Systemprogrammierung

Grundlage von Betriebssystemen

Sachwortverzeichnis

©Wolfgang Schröder-Preikschat

8. März 2025

Die in der Vorlesung (mündlich oder schriftlich) verwendeten Akronyme und Sachworte sind in der nachfolgenden Aufzählung zusammengefasst. Dabei werden die Akronyme nach den Sachworten, für die sie die Verkürzung bilden, aufgeschlüsselt. Für englischsprachige Sachwörter werden, soweit bekannt, die deutschsprachigen Entsprechungen angegeben. In der Beschreibung durch das ↑-Zeichen angeführte Sachwörter zeigen einen Kreuzverweis an. Jedes Sachwort wird erklärt, wobei dies im Zusammenhang mit dem hier relevanten Kontext der ↑Systemprogrammierung und in Bezug auf ↑Betriebssysteme geschieht. Die Formulierungen erheben nicht den Anspruch auf Gültigkeit auch für andere Fachrichtungen in der Informatik. Ebenso erhebt die Aufzählung nicht den Anspruch auf Vollständigkeit für das in der Vorlesung behandelte Fachgebiet.

Der vorliegende Text ist ein *Nachschlagewerk*, dessen Lektüre, im Gegensatz zu einem Lehrbuch mit durchgehendem roten Faden, für gewöhnlich nicht von vorne nach hinten empfohlen ist. Vielmehr ist der Text Begleitmaterial zu den Vorlesungsfolien. Neben Systemprogrammierung sind hier insbesondere die Lehrveranstaltungen Betriebssysteme, Betriebssystemtechnik, Nebenläufige Systeme und ↑Echtzeitsysteme eingeschlossen. Darüber hinaus kommen aber auch Lehrveranstaltungen zu den Themen Rechnerorganisation und ↑Rechnerarchitektur als ↑Bezugspunkt in Frage. Viele der dort (mündlich oder schriftlich) verwendeten Begriffe finden hier ihre Erklärung aus Sicht und mit Verständnis der systemnahen Programmierung. Ebenso dient der Text aber auch als Ergänzung zu (englisch- oder deutschsprachigen) Fach- oder Lehrbüchern zu diesen Themen, um dort verwendete Begriffe zu vertiefen, in einem größeren Kontext zu erfahren oder gar überhaupt erklärt zu bekommen.

Einige der behandelten Begriffe haben ihren Ursprung in der Erklärung eines anderen Begriffs, sie tauchen also nicht zwingend auch als Begriff in der jeweils verwendeten Primärliteratur auf. Diese Sachworte sind zwar nur durch Kreuzverweise in das ↑Verzeichnis gelangt, haben aber sehr wohl einen thematischen Bezug zur Systemprogrammierung. Ein Beispiel dafür ist die ↑Abfangung, nämlich als bildhafte Darstellung einer Hilfskonstruktion (in ↑CPU und ↑Betriebssystem) zur Sicherung von einem ↑Programmablauf, der eine ↑synchrone Ausnahme hervorruft (↑*trap*).

Fragen, Hinweise, Korrekturen oder Kommentare nimmt der Autor gerne entgegen, am besten per ↑Nachricht an `wosch@cs.fau.de`.

A-trap Jargon (Apple). Bezeichnung für einen ↑Maschinenbefehl, der von ↑Mac OS für Prozessoren der ↑m68k-Familie zur Auslösung eines ↑Systemaufrufs erwartet wurde. Die höherwertigen vier Bits des ↑Operationskodes eines solchen Befehls besaßen den Wert 1010_2, der im hexadezimalen Zahlensystem der Ziffer *A* in ↑ASCII entspricht. Bei Ausführung eines Maschinenbefehls, dessen Operationskode ein solches Bitmuster am Anfang hatte, wurde der

betreffende ↑Prozess abgefangen (↑*trap*) und es kam zur ↑Ausnahmebehandlung, in deren weiteren Verlauf schließlich die ↑partielle Interpretation des ↑Systemaufrufbefehls erfolgte.

ABA Problem Bezeichnung für einen Effekt und dadurch möglicherweise verursachten ↑Fehler, der bei der ↑Synchronisation im Zusammenhang mit ↑CAS auftreten kann. Dabei handelt es sich um die *falsch positive Ausführung* einer CAS-basierten Spekulation auf eine gemeinsame ↑Speicherstelle, an der eine ↑gemeinsame Variable liegt: die erfolgreiche Ausführung von CAS zeigt an, dass die beiden zu vergleichenden Operanden identisch sind und somit das Vorhandensein einer bestimmten globalen Bedingung vorgeben (positiv), aber diese globale Bedingung ist in Wirklichkeit nicht gegeben (falsch).
Angenommen Prozesse P_1 und P_2 greifen zugleich auf Speicherstelle L_i zu, um deren Inhalt zu verrechnen und zu aktualisieren. Der dabei von P_1 aus L_i gelesene Wert A sei ein Hinweis auf einen bestimmten globalen Zustand S_1. Aber P_1 verzögert sich im weiteren Verlauf, beispielsweise durch ↑Verdrängung, bevor er einen neuen Wert an L_i ablegen kann. In der Zwischenzeit ändert P_2 den Wert von L_i in B und dann wieder in A, wodurch ein neuer globaler Zustand $S_2 \neq S_1$ definiert wird. P_1 nimmt die ↑Arbeit wieder auf, stellt fest, dass der Wert von L_i gleich A ist, und geht daher von der Annahme aus, dass der globale Zustand nach wie vor S_1 sein muss — was jedoch nicht mehr zutrifft.

Exkurs Gegeben sei eine ↑verkettete Liste, die gemäß ↑LIFO verarbeitet wird, also einem ↑Stack entspricht. Der Listenzustand sei: $L_i \Rightarrow A \Rightarrow B \Rightarrow C$. Des Weiteren sei folgende Implementierung der Entnahmeoperation (*pull*) gegeben, in der CAS-basierte ↑nichtblockierende Synchronisation zur Anwendung kommt:

```
chain_t *pull(stack_t *this) {
    chain_t *item;
    do if ((item = this->link) == 0) break;          /* item=this->link */
    while (!CAS(&this->link, item, item->link)); /* this->link=item->link */
    return item;
}
```

Zum besseren Verständnis sind in Kommentaren die beiden relevanten funktionalen Schritte angegeben, um das Kopfelement aus der Liste zu entfernen und den Kopfzeiger auf dessen Nachfolger zu setzen. Für ein ↑sequentielles Programm sind diese Schritte ausreichend und korrekt, nicht jedoch für ein ↑nichtsequentielles Programm!
Die Operationsfolge bei gleichzeitiger Ausführung dieser Funktion durch P_1 und P_2 sei sodann wie folgt:

P_1 liest A aus L_i, vermerkt A in `item`, evaluiert B als Nachfolger (`item->link`) von A, verzögert kurz vor CAS, definiert $S_1 = A \rightsquigarrow B$ $\qquad L_i \Rightarrow A \Rightarrow B \Rightarrow C$

P_2 entfernt A aus der Liste ($L_i \Rightarrow B \Rightarrow C$), entfernt B aus der Liste ($L_i \Rightarrow C$), platziert A auf die Liste, definiert $S_2 = A \rightsquigarrow C$ $\qquad L_i \Rightarrow A \Rightarrow C$

P_1 setzt fort mit vermerktem A und evaluiertem B, führt CAS erfolgreich aus, da der Vergleich gelingt, denn `item` $= L_i = A$ $\qquad L_i \Rightarrow B \Rightarrow \circlearrowleft$

CAS wurde fälschlicherweise ($S_1 \neq S_2$) ausgeführt. Die Folge davon ist, dass (a) die von P_2 korrekt hinterlassene Liste $L_i \Rightarrow A \Rightarrow C$ verloren gegangen ist und (b) der Nachfolger des nunmehr einzigen und letzten Elements auf der Liste als undefiniert gelten muss, da zwischenzeitlich B von P_2 in beliebiger Weise verwendet (z.B. gelöscht oder auf eine andere Liste gesetzt) worden sein kann.
Die gängigste Methode zur Behandlung dieses Problems ist das Hinzufügen einer Änderungsnummer zu der gemeinsamen Variablen (hier: L_i), um damit eine Art Versionierung des CAS zu erreichen. Diese Nummer ist vor jedem CAS-Versuch für diese Variable um eins zu erhöhen, genauer: hier beim Einfügen eines neuen Kettenglieds in L_i. Die dafür geeignete Technik hängt von dem Wertebereich ab, der für die Änderungsnummer vorgesehen werden muss. Allgemeiner Ansatz ist es, für die Änderungsnummer ein eigenes Maschinenwort vorzusehen. In dem Fall müssen aber zwei Maschinenworte, unteilbar aktualisiert werden, wofür

eben auch ein spezieller atomarer Maschinenbefehl erforderlich ist: ↑DCAS.
Ein anderer Ansatz besteht darin, ungenutzte Bits in der gemeinsamen Variablen zu nutzen. Dies bietet sich im Falle von ↑Zeigern oder ↑Referenzen, also Adressen, zu aneinandergereihten ↑Daten an. Der Trick liegt dabei in der Auffüllung (*padding*) einer Datenstruktur bis zur nächsten Zweierpotenzgröße. Eine Adresse von solch einem ↑Objekt der Größe 2^n ↑Bytes enthält dann immer $n-1$ niederwertige Bits mit dem jeweiligen Wert 0. Diese $n-1$ Bits können als Platzhalter für die Änderungsnummer verwendet werden, sofern der Wertebereich $[0, 2^{n-1}]$ ausreicht. Bei diesem Ansatz muss nach wie vor nur ein Maschinenwort unteilbar aktualisiert werden, was mit dem einfachen CAS gegeben ist.
Für beide Ansätze sind allerdings immer noch Zählerüberläufe zu beachten — weshalb das eigentliche CAS-Problem nicht grundsätzlich vom Tisch ist, sondern nur mehr oder weniger unwahrscheinlich geworden ist. Mit ↑LL/SC besteht diese Problematik nicht.
Nachfolgendes Beispiel demonstriert die ↑Behelfslösung zur Zeigerversionierung, bei der die Änderungsnummer im Zeiger selbst kodiert ist, nämlich in dem ungenutzten niederwertigen Bitfeld (*pointer tag*):

```
inline void *raw(void *this, long mask) {
    return (void *)((long)this & ~mask);
}
```

Diese Funktion bereinigt einen mit einer Änderungsnummer markierten Zeiger, in dem sichergestellt ist, dass das betreffende Bitfeld komplett genullt vorliegt. Nur ein so bereinigter Zeiger darf wieder als Adresse verwendet werden! Dann noch die Funktion, die die Änderungsnummer eines markierten Zeigers um eins erhöht:

```
inline void *tag(void *this, long mask) {
    return (void *)((long)raw(this, mask) | ((long)this + 1) & mask);
}
```

Der betreffende Zeiger stellt letztlich eine Ganzzahl dar, deren Wert inkrementiert und in das Bitfeld der Änderungsnummer eingetragen wird. Dabei ist zu beachten, dass die Breite der Maske durch die Objektgröße als Zweierpotenz bestimmt sein muss. Daher nachfolgende zwei Definitionen:

```
typedef chain_t chain_l;                                /* labelled pointer */

#define TFM (sizeof(chain_l) - 1)                        /* tag-field mask */
```

Zum einen die Einführung eines speziellen Typen für den mit einer Änderungsnummer versehenen Zeiger. Zum anderen die zugehörige Bitfeldmaske, die den Abschnitt der niederwertigen Bits eines solchen Zeigers spezifiziert und damit die Zone für die Änderungsnummer festlegt. Nur um die prinzipielle Methode zu erläutern, entspricht die Objektgröße hier beispielhaft der Größe eines Kettenglieds und ist damit vergleichsweise klein: acht Bytes bei einem 64-Bit System, damit auch nur drei Bits für die Maske und $[0, 7]$ als Wertebereich für die Änderungsnummer.
Angewendet für das `pull`-Beispiel oben ergibt sich sodann nachfolgend skizziertes Bild:

```
typedef chain_t chain_l;                      chain_l *pull(stack_t *this) {
    chain_l *item;
    do if (raw((item = this->link), TFM) == 0) break;
    while (!CAS(&this->link, item, ((chain_t *)raw(item, TFM))->link));
    return item;
}
```

Alle Kettenglieder in der Liste sind nunmehr markierte Zeiger (`chain_l`). Bei der Herausnahme eines Kettenglieds aus dieser Liste, wird der Kopfzeiger lediglich auf den Nachfolger eben dieses Kettenglieds gesetzt. Dieser Nachfolger enthält ebenfalls einen markierten Zeiger, dessen Änderungsnummer hier nicht inkrementiert wird, da kein neues Listenelement eingefügt wird. Dies ist erst mit der entsprechenden Einfügeoperation der Fall:

```
void push(stack_t *this, chain_l *item) {
    do ((chain_t *)raw(item, TFM))->link = this->link;
    while (!CAS(&this->link, ((chain_t *)raw(item, TFM))->link, tag(item, TFM));
}
```

Im Gegensatz zu vorher wird mit CAS ein neues („von außen“ kommendes) Kettenglied in die Liste eingetragen und nicht ein bereits auf der Liste stehendes. Daher wird für dieses Kettenglied (`item`) die Änderungsnummer inkrementiert.

Wie die beiden Implementierungen andeuten, ist der Ansatz mit der Änderungsnummer intransparent für die Programme, die die betreffenden Funktionen benutzen. Um dies deutlich zu machen, wurde auch ein neuer Zeigertyp (`chain_l`) eingeführt. Wenn die Hardware nur CAS-artige Maschinenbefehle bietet, lassen sich die teils schnell komplex werdenden Lösungen zur nichtblockierenden Synchronisation bestimmter Datenstrukturen für den vorliegenden Fall nicht komplett verbergen. Denn es ist festzuhalten und zu beachten, dass das hier behandelte Problem einer CAS-basierten Spekulation zwar grundsätzlicher Natur ist, aber nicht in jedem Anwendungsfall von CAS auch auftritt.

ABB Abkürzung für (en.) ↑*atomic basic block*.

Abfangung (en.) ↑*interception*. Konstruktion, die einen stattfindenden ↑Prozess gegen „Kippen“ sichert (in Anlehnung an das ↑Bauwesen) und ihn nicht unkontrolliert scheitern lässt. Gemeint ist eine im ↑Prozessor existierende Vorrichtung, durch die in einem bestimmten Systemzustand eine ↑Aktion, die der Prozess selbst verursacht, abgefangen wird: an der ↑Fangstelle wird eine ↑synchrone Ausnahme erhoben. Zweck der damit verbundenen Funktion ist es beispielsweise, ↑Maschinenbefehle, ↑Adressen oder ↑Daten auf ihre Gültigkeit hin abzuprüfen und den Prozess zu unterbrechen, ihn in eine sichere Falle (↑*trap*) tappen zu lassen, um eine ungültige Aktion zu verhindern.

abgesetzter Betrieb (en.) ↑*remote operation*. Bezeichnung für eine ↑Betriebsart, bei der die ↑Ein-/Ausgabe physisch getrennt und damit auch logisch entkoppelt von der damit im Zusammenhang stehenden Berechnung erfolgt.

Die Leistungsfähigkeit von einem ↑Rechensystem, in dem Berechnungen und Ein-/Ausgabe in „Personalunion“ durch ein einzelnes von der ↑CPU ausgeführtes ↑Programm bewerkstelligt wird, ist begrenzt durch die Geschwindigkeit, mit der die Ein-/Ausgabe durchgeführt werden kann. Grund dafür ist die im Vergleich zur CPU für gewöhnlich sehr viel langsamer arbeitende ↑Peripherie. So liegt die ↑Latenzzeit für den Zugriff auf ein Bandlaufwerk im Sekunden- oder Minutenbereich, ein Festplattenlaufwerk bei 10 ms (über 2 Größenordnungen schneller), ein ↑SSD bei 25 μs (3 Größenordnungen schneller), den ↑RAM bei 50 ns (3 Größenordnungen schneller): das heißt, eine CPU könnte in der Zeit einen Durchsatz von mehr als 10^3 (SSD), 10^6 (Festplattenlaufwerk) oder 10^8 (Bandlaufwerk) Operationen pro Sekunde leisten (Stand 2025). Darüberhinaus ist in bestimmten Fällen gerade die Dauer für Eingabe oftmals nicht vorhersehbar und damit unbestimmt, etwa die Latenzzeit bis zu einem Tastendruck (Zeicheneingabe) oder Mausklick (Signaleingabe): also überall dort, wo ein unberechenbarer „externer Prozess“ (z.B. der Mensch) für die Eingabe verantwortlich ist.

Diese Ein-/Ausgabeabhängigkeit in der Rechenleistung wird wesentlich reduziert, wenn jeder für die CPU bestimmte Auftrag (↑*job*) immer über das schnellste ↑Peripheriegerät eingespeist und die eigentliche Zusammenstellung dieser Aufträge sowie die dazu benötigte Ein-/Ausgabe unabhängig von der CPU (*off-line*) bewerkstelligt wird. Das Rechensystem besteht dazu aus einerseits ↑Satellitenrechner für die eigentliche Bedienung der Peripherie und zur Durchführung der vergleichsweise langsamen Ein-/Ausgabe und andererseits dem ↑Hauptrechner zur Durchführung der eigentlichen Berechnungen. Die Ein-/Ausgabe (Satellitenrechner) wird abgesetzt von Berechnungen (Hauptrechner) betrieben oder umgekehrt. Dazu werden auf dem Satellitenrechner, über die jeweils benötigten und dort angeschlossenen Eingabegeräte, die zu bearbeitenden Aufträge samt aller Eingabedaten auf ein schnelles ↑Speichermedium überspielt, das dann zum Hauptrechner (manuell) übertragen wird, um dort die gestapelten

Arbeitsaufträge nacheinander auszuführen. Für die Ausgabe wird umgekehrt verfahren: sie gelangt zunächst auf ein schnelles Speichermedium am Hauptrechner und wird danach an einen oder mehrere Satellitenrechner, an dem die benötigten Ausgabegeräte angeschlossen sind, (manuell) weitergegeben. Die Aufträge werden sodann vom Hauptrechner in ↑Stapelverarbeitung ausgeführt.

Abgrenzung (en.) ↑*fencing*. Schutzmechanismus, bei dem ein ↑Adressraum an einer bestimmten ↑Adresse, der ↑Gatteradresse, in zwei Zonen aufgeteilt wird: eine geschützte Zone für das ↑Betriebssystem und eine ungeschützte Zone für das ↑Maschinenprogramm. Generiert ein ↑Prozess der ungeschützten Zone eine Adresse der geschützten Zone, wird er abgefangen (↑*trap*). Umgekehrt gilt dies nicht zwingend, das heißt, das Betriebssystem hat potentiell freien Zugriff auf den gesamten Adressraum und damit eben auch auf den vom Maschinenprogramm belegten Bereich (ungeschützte Zone).
Ursprünglich ist ein ↑realer Adressraum und damit der ↑Hauptspeicher mit solch einer Grenzlinie versehen worden, um ein ↑residentes Steuerprogramm zu schützen. Aber auch ein ↑logischer Adressraum lässt sich auf diese Weise partitionieren und in verschiedene Schutzzonen aufteilen, wie etwa das ↑Mehradressraummodell von ↑Linux oder ↑Windows NT zeigt.
Bevor die von dem Prozess im Maschinenprogramm gebildete effektive (reale/logische) Adresse zum ↑Adressbus geht, wird sie von der ↑CPU mit der Gatteradresse verglichen. Die vom Prozess generierte (effektive) Adresse passiert nur dann den Adressbus (d.h., der Zugriff gelingt nur), wenn sie nicht innerhalb des durch die Gatteradresse markierten ↑Adressbereichs liegt. Dazu führt die CPU eine einfache Vergleichsoperation ($<$ oder $\geq$) aus, je nachdem, ob die geschützte Zone den vorderen oder hinteren Adressbereich ausmacht. Das Ergebnis des Vergleichs hat jedoch nur Auswirkung bei Ausführung des Maschinenprogramms, um nämlich Zugriffe auf das Betriebssystem zu unterbinden. Ist das Betriebssystem und damit ein ↑Programm jenseits der Gatteradresse aktiv, besteht für die CPU sonst mit jedem ↑Maschinenbefehl fortlaufend eine ↑Ausnahmesituation — deren Behandlung, die nämlich innerhalb des Betriebssystem geschieht, sofort wieder eine ↑Ausnahme hervorbringt. Daher wechselt die CPU den ↑Arbeitsmodus, je nachdem in welcher Zone sie gerade operiert: Systemmodus (↑*system mode*) für die geschützte und Benutzermodus (↑*user mode*) für die ungeschützte Zone. Normalerweise ist der Benutzermodus aktiv und das Maschinenprogramm wird ausgeführt. Jede Ausnahme aktiviert den Systemmodus. Darüberhinaus ist ein Spezialbefehl erforderlich, auch in Form eines ↑Systemaufrufs, um bewusst den Wechsel in den Systemmodus zu erreichen und so die Dienste des Betriebssystems überhaupt in Anspruch nehmen zu können.
Diese einfache Technik ist zugleich ein sehr restriktiver Ansatz, da keine der beiden Zonen zum einen über eine bestimmte Grenze hinweg expandieren und zum anderen den nicht ausgeschöpften Bereich der jeweils anderen Zone für den eigenen Bedarf nutzen kann (↑interne Fragmentierung). Zudem ist der Ansatz nur für ↑Einprogrammbetrieb tauglich, da der Schutz durch den Adressvergleich (größer/kleiner) immer nur unidirektional greift.

Ablage (en.) ↑*filing*. Bezeichnung für einen ↑Speicher zur mittelfristigen, dauerhaften Aufbewahrung von Informationen; Vorrichtung in einem ↑Rechensystem, wo eine ↑Datei abgelegt wird, ein ↑nichtflüchtiger Speicher. Die technische Speicherung erfolgt mitlaufend (*on-line*) zu dem ↑Prozess, der die Informationen ablegt oder auf (von ihm selbst oder anderen Prozessen) abgelegte Informationen zugreift. Auch ↑Sekundärspeicher genannt.

Ablagesystem (en.) ↑*filing system*. Bezeichnung für ein dem Ablegen von ↑Dokumenten dienendes System. Die ↑Ablage oder das ↑Archiv in einem ↑Rechensystem, implementiert durch ein ↑Dateisystem.

Ablaufinvarianz (en.) ↑*reentrancy*. Merkmal von einem ↑Programm, mehr als einen ↑Handlungsstrang gleichzeitig zuzulassen, ohne sich in den daraus resultierenden zeitlich überlappenden Abläufen gegenseitig beeinflussen zu können. Eine solche Überlappung von Abläufen zeigt sich beispielsweise bei der ↑Ausnahmebehandlung, bei der nämlich der normale Ablauf eines

Programms überlappt wird von dem unnormalen Ablauf durch einen ↑Ausnahmehandhaber. Variante davon ist das Programm zur ↑Unterbrechungsbehandlung, um im Betriebssystem auf eine ↑Unterbrechungsanforderung der Hardware reagieren zu können. Ähnliche Abläufe resultieren aus der Möglichkeit zur ↑Parallelverarbeitung ein und desselben Programms. Damit derartige Abläufe frei von gegenseitiger Beeinflussung sind, muss für den betreffenden Abschnitt ↑Eintrittsinvarianz gelten. Für einen ↑Prozess in diesem Abschnitt bedeutet dies: keine Verwendung statischer oder globaler ↑Variablen, keine Veränderung an seinem ↑Text und kein Aufruf von einem nicht eintrittsinvarianten ↑Unterprogramm.

Ablaufplan (en.) ↑*schedule*. Ergebnis der ↑Ablaufplanung.

Ablaufplanung (en.) ↑*scheduling*. Verfahren, das die Zuteilung von einen ↑Prozessor an einen ↑Prozess oder eine Gruppe von Prozessen regelt. Das Verfahren arbeitet rechnerunabhängig (*off-line*) oder mitlaufend (*on-line*) zu den Prozessen, es liefert dementsprechend einen statischen oder dynamischen ↑Ablaufplan, der zur ↑Laufzeit der Prozesse unverändert befolgt wird oder aktuellen Bedürf- und Geschehnissen angepasst werden kann. Grundlage bildet ein auf das jeweilige Verfahren zugeschnittener ↑Einplanungsalgorithmus.

Ablaufsteuerung (en.) ↑*flow control*. Steuerung von einem ↑Programmablauf mithilfe aufeinanderfolgender Befehle oder Bedingungen (Duden), um einem ↑Prozess einen bestimmten ↑Kontrollfluss zu vermitteln.

Abmeldung (en.) ↑*log-out*. Prozedur, nach der eine Person oder ein externer ↑Prozess dem ↑Betriebssystem gegenüber die Teilnahme am Rechenbetrieb aufkündigt: das Sichabmelden aus einem ↑Rechner oder ↑Rechnernetz. Für gewöhnlich werden dabei alle während der individuellen Teilnahme dynamisch angeforderten ↑Betriebsmittel wieder freigegeben.

Abruf- und Ausführungszyklus (en.) ↑*fetch-execute-cycle*. Bezeichnung für den ↑Befehlszyklus von einem ↑Interpreter: (1) den nächsten Befehl aus dem ↑Programm lesen, (2) ihn dekodieren, gegebenenfalls in Einzeloperationen zerlegen, alle benötigten Operanden entsprechend der jeweils spezifizierten ↑Adressierungsart laden und (3) die Operation/en durchführen. Schließlich (4) die Abfrage, ob während dieser Befehlsausführung eine ↑Ausnahme erhoben wurde und gegebenenfalls den ↑Unterbrechungszyklus einleiten. Führt eine ↑Aktion herbei.

absolute address (dt.). ↑absolute Adresse, ↑Maschinenadresse.

absolute Adresse (en.) ↑*absolute address*. Bezeichnung für eine ↑Adresse, die nicht relativ zu einem bestimmten ↑Bezugspunkt steht, sondern direkt eine bestimmte ↑Speicherstelle benennt. Auch als ↑direkte Adresse bezeichnet. Derartige Adressen sind der ↑Verlagerung zu unterziehen, sollte das sie enthaltene ↑Programm im jeweiligen ↑Adressraum verschoben werden müssen.

absolute loader (dt.) ↑Absolutlader.

Absolutlader (en.) ↑*absolute loader*. Bezeichnung für ein ↑Programm, das ein ausschließlich absolute Adressen verwendendes Programm in den ↑Hauptspeicher bringt; ein ↑Lader, der keine ↑Verlagerung durchführt. Durch solch einen Lader wird typischerweise das ↑Betriebssystem in den Hauptspeicher gebracht (↑*bootstrap*).

abstract machine (dt.) ↑abstrakte Maschine, abstrakter ↑Rechner.

abstrakte Maschine (en.) ↑*abstract machine*. Bezeichnung für eine Maschine, die nur gedanklich und nicht physisch existiert. Sie ist entweder ein als Modell geschaffenes Konstrukt, um komplexe Vorgänge innerhalb von Rechensystemen auf theoretischer Ebene zu untersuchen (Automat, Turing-Maschine) oder ein über eine wohldefinierte Schnittstelle zugängliches Softwaregebilde, das allein oder im Ensemble mit anderen Maschinen ihrer Art hilft, eine ↑semantische Lücke zu schließen (↑Betriebssystem). Letztere Variante meint ein ↑Programm, das zwar ein physisches ↑Betriebsmittel in Form von ↑Speicher belegt, als Software jedoch selbst ein „nicht technisch-physikalischer Funktionsbestandteil“ (Duden) darstellt.

Abstraktionsebene (en.) ↑*level of abstraction*. Ein bestimmtes Niveau einer *Modellvorstellung*, festgelegt durch die zur Modellierung herangezogenen (funktionalen, mathematischen) Abstraktionen. Beispielsweise die Modellierung eines ↑Gatters durch einen elektronischen Schaltkreis (Realität) oder durch eine boolesche Funktion (Theorie). Aber auch die Modellierung einer ↑Anwendung als Satz von ↑Maschinenprogrammen (Funktionen). Dabei ist eine solche Stufe geprägt durch eine bestimmte Genauigkeit der Modellierung, mit der aus dem Besonderen der jeweils gegebenen Hilfs- oder Ausdrucksmittel das Allgemeine entnommen wird (Abstrahieren). Dies geschieht für gewöhnlich durch fortlassen von Details. Die so gebildete „höhere“ Ebene ist durch eine „tiefere“ Ebene verkörpert, eine Beziehung, die gegebenenfalls als ↑funktionale Hierarchie zum Ausdruck kommt.

acceptance test (dt.) ↑Übernahmeprüfung.

access control list (dt.) ↑Zugriffskontrollliste.

access fault (dt.) ↑Zugriffsfehler.

access matrix (dt.) ↑Zugriffsmatrix.

access right (dt.) ↑Zugriffsrecht.

access time (dt.) ↑Zugriffszeit.

accounting (dt.) ↑Buchführung.

ACET Abkürzung für (en.) *average-case* ↑*execution time*, mittlere anzunehmende ↑Ausführungszeit.

ACL Abkürzung für (en.) ↑*access control list*.

action (dt.) ↑Aktion, Handlung.

action strand (dt.) ↑Handlungsstrang.

activation record (dt.) ↑Aktivierungsblock.

active register (dt.) ↑aktives Register.

active waiting (dt.) ↑aktives Warten.

actual parameter (dt.) ↑tatsächlicher Parameter.

address (dt.) ↑Adresse.

address bus (dt.) ↑Adressbus.

address mapping (dt.) ↑Adressabbildung.

address range (dt.) ↑Adressbereich.

address space (dt.) ↑Adressraum.

address space segment (dt.) ↑Adressraumsegment.

address width (dt.) ↑Adressbreite.

address-space isolation (dt.) ↑Adressraumisolation.

address-space layout (dt.) ↑Adressraumbelegungsplan.

address-space model (dt.) ↑Adressraummodell.

addressing mode (dt.) ↑Adressierungsart.

Adressabbildung (en.) ↑*address mapping*. Zuordnung, die für eine ↑logische Adresse genau eine ↑reale Adresse festlegt. Dabei hängt die Abbildungsfunktion davon ab, wie der ↑Adressraum einer ↑Prozessinkarnation physisch aufgebaut ist und durch die Hardware (↑MMU) technisch repräsentiert werden kann, sie ist aber immer tabellengesteuert.
Grundsätzlich unterscheidet die Abbildung zwischen ↑Seitenadressierung, die einen eindimensionalen Ansatz zur Organisation eines ↑Prozessadressraums verfolgt und auf ↑Seitentabellen basiert, und ↑Segmentierung, der ein zweidimensionales Modell zugrunde liegt und die ↑Segmenttabellen nutzt. Beide lassen sich in zwei verschiedenen Arten mit der Segmentierung als zentrales Konzept kombinieren, gliedern dann allerdings verschiedene ↑Entitäten in ↑Segmente auf. Die eine Art segmentiert die Seitentabellen und ermöglicht so eine ↑Ressourcen sparende Verwaltung der dann als *dynamisches* ↑*Feld* organisierten Zusammenstellung der ↑Seitendeskriptoren eines Prozessadressraums (↑*segmented paging*). Die andere Art segmentiert den Prozessadressraum selbst und erlaubt die ↑Seitenumlagerung der einzelnen Segmente in diesem Adressraum (↑*paged segmentation*).
Letztere ist vergleichsweise sehr aufwendig, auch die reine Segmentierung ist im Vergleich zur Seitenadressierung mit hohem Aufwand insbesondere in der ↑Speicherverwaltung verbunden (↑*placement policy*). So genießt die Seitenadressierung für gewöhnlich den Vorzug, wenn systemorientierte Kriterien zur ↑Performanz im Vordergrund stehen. Demgegenüber reflektiert die Segmentierung besser benutzerorientierte Kriterien, etwa bei der Repräsentation von ↑Programmstrukturen und ihrer Aufteilung in ↑Module.

Adressbereich (en.) ↑*address range*. Ein bestimmter Abschnitt stetig verlaufender ↑Adressen. Eine Menge von Adressen, in der jedes Element eine ↑lineare Adresse bildet.

Adressbreite (en.) ↑*address width*. Bitanzahl, die zur Darstellung einer ↑Adresse im ↑Dualsystem zur Verfügung steht. Typisch waren oder sind 16, 18, 20, 24, 32, 48 und 64 Bits pro Adresse (Stand 2025).

Adressbus (en.) ↑*address bus*. Verbindungssystem (↑*bus*) in einem ↑Rechner, über das eine ↑reale Adresse übermittelt wird.

Adresse (en.) ↑*address*. Allgemein die Angabe des ↑Namens und Orts einer ↑Entität, speziell die Nummer einer ↑Speicherstelle.

Adressierungsart (en.) ↑*addressing mode*. Lokalisierung der/des Operanden von einen ↑Maschinenbefehl. Gängig sind: Registeradressierung, der Befehl enthält den Namen des dem Operanden zugeordneten Prozessorregisters; unmittelbare Adressierung (Direktwertadressierung), der Befehl enthält den Operanden selbst; direkte (absolute) Adressierung, der Befehl enthält die Speicheradresse des Operanden; indirekte Adressierung, der Befehl enthält den Namen des die Speicheradresse des Operanden führenden Prozessorregisters; relative Adressierung, der Befehl enthält den auf eine Basisadresse bezogenen Abstand zum Operanden; indizierte Adressierung, der Befehl enthält den Namen des den zu einer Basisadresse bezogenen Abstand zum Operanden führenden Prozessorregisters.

Adressraum (en.) ↑*address space*. Menge von nichtnegativen ganzen Zahlen, endlich, wobei jedes Element darin eine ↑Adresse repräsentiert. Die Kardinalität dieser Menge ist variabel und bestimmt durch den jeweils zum Bezug genommenen ↑Prozess, aber nach oben begrenzt durch die ↑Adressbreite der (realen/virtuellen) Maschine.
Damit hat diese Menge eine dynamische oder statische Charakteristik, je nachdem welche Sicht (Prozess, Maschine) eingenommen wird. Steht der dynamische Aspekt im Vordergrund, wird oft auch von ↑Prozessadressraum gesprochen. Demgegenüber kennt der statische Aspekt für gewöhnlich drei verschiedene Auslegungen dieser Menge, nämlich als ↑realer Adressraum, ↑logischer Adressraum oder ↑virtueller Adressraum. Der Prozessadressraum ist durch eine dieser Arten in Anordnung und Anzahl der gültigen Adressen geprägt.

Adressraumbelegungsplan (en.) ↑*address-space layout*. Aufriss von einem ↑Adressraum hinsichtlich gültiger, ungültiger oder zu bestimmten Zwecken reservierter ↑Adressbereiche. Ausgewiesen sind nicht nur die mit dem ↑Speicherplan dargestellten ↑Speicherbereiche, sondern auch ↑Adressen, über die ↑speicherabgebildete Ein-/Ausgabe möglich ist oder die ins Leere verweisen. Letztere sind Lücken im Adressraum, die für gewöhnlich einen ↑Busfehler hervorbringen, wenn eine diesbezügliche Adresse von einem ↑Prozess appliziert wird. Aber auch nicht jeder der aufgeführten Speicherbereiche ist als ↑Hauptspeicher zur Ausführung von ↑Programmen inklusive dem ↑Betriebssystem verwendbar. Die zur freien Verfügung stehenden Hauptspeicherbereiche sind entsprechend kenntlich gemacht. Nachfolgend ein Beispiel:

	Adressbereich	Größe (KiB)	Verwendung	
1.	`00000000`–`0009ffff`	640	RAM	System
2.	`000a0000`–`000bffff`	128	Video RAM	Bildschirm
3.	`000c0000`–`000c7fff`	32	Video RAM	BIOS
4.	`000c8000`–`000dffff`	96	Lücke	keine
5.	`000e0000`–`000effff`	64	Video RAM	BIOS (*shadow*)
6.	`000f0000`–`000fffff`	64	RAM	BIOS (*shadow*)
7.	`00100000`–`090fffff`	147456	RAM	Erweiterung
8.	`09100000`–`fffdffff`	4045696	Lücke	keine
9.	`fffe0000`–`fffeffff`	64	SM-RAM	(*system management*)
10.	`ffff0000`–`ffffffff`	64	ROM	BIOS

Dargestellt ist ein ↑realer Adressraum (32-Bit ↑Rechner: Toshiba Tecra 730CDT, 1996), der nur mit dem System- und Erweiterungsbereich (1. und 7. Zeile) Hauptspeicher für das Betriebssystem und die ↑Maschinenprogramme verfügbar macht. In diesem Beispiel bleibt der weitaus größte Bereich des realen Adressraums ungenutzt (8. Zeile), nämlich etwa 3.86 von insgesamt maximal adressierbaren 4 GiB. Anders ausgedrückt: nur gut 3,55 % der für dieses 32-Bit System möglichen 2^{32} realen Adressen sind wirklich durch die Hardware definiert. Für die heutzutage (2025) verfügbaren 64-Bit Systeme ergibt sich ein noch viel größeres Missverhältnis zwischen möglichen und definierten realen Adressen.
Die in dem beispielhaft hier betrachteten Rechner eingebaute ↑CPU (Intel Pentium) trennt Speicher- und Ein-/Ausgabeadressraum, womit die ↑Ein-/Ausgaberegister einerseits über Adressen eines eigenen Adressraums und andererseits mittels spezieller ↑Maschinenbefehle (`in`, `out`) angesprochen werden. Für gewöhnlich greift auf Basis solch einer CPU ↑isolierte Ein-/Ausgabe. Auch für den Ein-/Ausgabeadressraum gibt es eine Aufstellung von Adressen, die den jeweiligen Ein-/Ausgaberegistern entsprechen beziehungsweise an denen ↑Peripherie angeschlossen ist.

Adressraumisolation (en.) ↑*address-space isolation*. Abkapselung von einem ↑Prozess, indem sein ↑Adressraum physisch von den jeweiligen Adressräumen anderer Prozesse im ↑Rechensystem getrennt gehalten wird. Technische Grundlage dafür bildet der ↑Speicherschutz, in aller Regel hardwarebasiert unter Verwendung einer ↑MMU oder ↑MPU. Geschieht die Abkapselung *total*, betrifft sie also den kompletten Adressraum, gilt der Prozess als ↑schwergewichtiger Prozess. Greift sie dagegen *partiell*, bezieht sie sich nämlich nur auf den das ↑Stapelsegment eines Prozesses betreffenden ↑Adressbereich, ist der Prozess ein ↑leichtgewichtiger Prozess. Ein Prozess kann damit aus seinen isolierten Adressbereichen nicht ausbrechen und somit auch nicht in den Adressraum eines anderen Prozesses einbrechen. Genau genommen trifft letzteres allerdings nur im Falle einer totalen Abkapselung zu. Für die partielle Abkapselung wird bewusst von der gemeinsamen Benutzung zumindest eines Teils desselben Adressraums durch mehrere Prozesse ausgegangen (↑*shared memory*).

Adressraummodell (en.) ↑*address-space model*. Typ, Art der Ausführung des ↑Adressraums für die von einem ↑Betriebssystem verwalteten ↑Prozesse. Grundlage ist ein ↑logischer Adressraum, der, je nach ↑Betriebsart, erweitert als ↑virtueller Adressraum in Erscheinung treten kann. In beiden Fällen bilden ↑Seiten oder ↑Segmente die Organisationselemente des Adress-

raums und bestimmen damit auch die Verfahren im Umgang mit diesen Elementen (↑*paging*, ↑*segmentation*), gegebenenfalls auch in Kombination (↑*segmented paging*, ↑*paged segmentation*). Weitere charakteristische Eigenschaft ist, ob ein ↑realer Adressraum das Bild für mehr als einen logischen/virtuellen Adressraum gleichzeitig darstellt (↑*multi-address space*), die ↑Adressabbildung damit auch durch ein ↑Multiplexverfahren definiert ist, oder nicht (↑*single-address space*).

Adressraumsegment (en.) ↑*address space segment*. Gleichartiger Abschnitt eines gegliederten, nach einem bestimmten System *physisch* aufgeteilten ↑Adressraums. Der ↑Adressbereich eines solchen Abschnitts ist durch ein ↑Segment erfasst, womit der gesamte Adressraum als ↑segmentierter Adressraum in Erscheinung tritt. Für gewöhnlich ist auf solch ein Segment ein ↑Programmsegment abgebildet, entweder als Ganzes oder in Einzelheiten verteilt über mehrere Segmente der gleichen Art (↑Text, ↑Daten, ↑BSS.

Adresszähler (en.) ↑*location counter*. Bezeichnung für einen vom ↑Assemblierer pro ↑Segment (insb. ↑Text, ↑Daten, ↑BSS) verwalteten und beim ↑Assemblieren fortgeschriebenen Zähler einer ↑Assemblieranweisung. Der mit dem Wert 0 initialisierte Zähler repräsentiert gleichsam eine *relative* ↑*Adresse*, an der diese Anweisung den angeordneten ↑Maschinenkode generiert oder einen ↑Platzhalter für eine sonstige ↑Entität reserviert hat. Dieser Zähler wird beim Assemblieren um die Länge (in Bytes) der Assemblieranweisung erhöht.
Darüberhinaus kann jede Assemblieranweisung mit einer Marke (↑*label*) versehen sein, für gewöhnlich ein ↑Name gefolgt vom Doppelpunkt (`.LC0:`), wodurch dieser Anweisung eine ↑symbolische Adresse zugeordnet wird (vgl. ↑Übersetzungsprotokoll, S. 23):

```
Zeile  ↑ILC  Maschinenkode          Assemblieranweisung
  20                                .LC0:
  21   0026  48656C6C  6F20776F              .string "Hello world!"
  22         726C6421  00
```

Beim Assemblieren erhält der als ↑Symbol (`.LC0`) interpretierte Name der Marke einen Wert (0026_{16}), der dem Wert des Adresszählers für eben diese Anweisung entspricht. Damit ist der Wert des Symbols eine *relative Adresse* innerhalb des betreffenden Segments (hier: Text, S. 23). Beim ↑Binden wird zu dieser relativen Adresse (0026_{16}) dann die ↑Ladeadresse des Segments addiert, um die ↑absolute Adresse des durch die, ursprünglich symbolisch adressierte, Assemblieranweisung (`.string`) generierten ↑Maschinenkodes festzulegen.
Der Zählerwert — nämlich die entsprechende symbolische, relative oder absolute Adresse — kann als Operand in einer Assemblieranweisung verwendet werden. Typisch ist das Dollarzeichen (`$`): einem Symbol vorangestellt (`$.LC0`, S. 23) liefert es dessen Wert als Operand.

after you, after you (dt.) nach dir, nach dir. Sinnbild für einen möglichen Effekt bei der ↑Synchronisierung, der das ↑Verhungern von ↑Prozessen zur Folge haben kann.
Der Überlieferung nach (Dijkstra) eilen zwei Gentleman zielstrebig auf einen Türdurchgang zu, der zu eng ist, dass beide ihn zugleich nebeneinander passieren könnten. Als Mann von Anstand gewährt jeder dem anderen selbstverständlich den Vorrang, um daraufhin nacheinander seines Weges gehen zu können. Dazu zieht einer seinen Bowler und bittet den anderen, durch ein höfliches „nach Ihnen“, voranzugehen. Letzterer steht seinem Konkurrenten in nichts nach, zieht gleichzeitig seinen Bowler und bittet ersteren mit einem ebenso höflichen „nach Ihnen“ vor. Beide nehmen an, der jeweils andere hält inne, was jedoch nicht der Fall ist. So kommt es wie es kommen musste: eng nebeneinander gehend prallen sie zusammen gegen die Türpfosten, erkennen ihren Irrtum, ziehen sich zurück und, anständig wie sie sind, gewähren sie dem jeweils anderen erneut den Vorrang. Blind gegenüber anderem Sozialverhalten, das nämlich Fortschritt verspricht, verfallen beide immer wieder demselben Ablauf — und wenn sie nicht gestorben sind, dann „kreiseln“ sie noch heute vor der Tür.

ageing (dt.) ↑Alterung.

Aktion (en.) ↑*action*. Ausführung der Anweisung einer Maschine durch einen ↑Prozessor. Eine Anweisung, die eine Einzelaktion auf höherer Ebene beschreibt, kann als ↑Aktionsfolge auf tieferer Ebene stattfinden.

Exkurs Diese mögliche Auslegung einer für die verschiedenen Maschinen einer ↑Mehrebenenmaschine semantisch äquivalenten Operation einerseits als Einzelaktion und andererseits als Aktionsfolge ist nachfolgend an einem einfachen Beispiel dargestellt:

Ebene	Aktion	Aktionsfolge
5	`i++`	

4–3	`incl i`*	`movl i,%r`
		`addl $1,%r`*
		`movl %r,i`

	`addl $1,i`*	
2–1		* ***read*** *from memory into accumulator* ***modify*** *contents of accumulator* ***write*** *from accumulator into memory*

Ausgang ist die Anweisung (Postinkrement) in ↑C zum Erhöhen der ↑Variablen `i` um Eins. Diese eine Anweisung spezifiziert eine einzelne Aktion auf der Ebene 5. Der ↑Kompilierer übersetzt diese einzelne Anweisung entweder in einen einzelnen Maschinenbefehl, hier ein Inkrement- (`incl i`) oder ein Additionsbefehl (`addl $1, i`), oder drei Maschinenbefehle. Welche der gezeigten insgesamt drei Varianten für ↑x86 sich ergeben, bestimmt letztlich der Kompilierer in Abhängigkeit von dem Kontext, in dem die zu übersetzende Anweisung im ↑Programm steht und von der jeweils eingestellten Optimierungsstufe. Entsprechend ergibt sich für Ebene 4–3 entweder eine einzelne Aktion oder eine Aktionsfolge, die jeweils die eine Aktion der Ebene 5 implementieren.
Darüberhinaus sind die mit einem Asterisk (*) versehenen Maschinenbefehle typische Fälle von Einzelaktionen, die in der Hardware (Ebene 2–1) selbst jeweils als Aktionsfolge implementiert sind. Um nämlich mit diesen Befehlen die gewünschte Erhöhung des Operandenwerts um Eins zu erreichen, muss die ↑CPU in ihrem ↑Befehlszyklus den Operanden zunächst lesen, den gelesenen Wert dann (in der ALU) entsprechend modifizieren und das Ergebnis schließlich zurückgeschreiben (↑*read-modify-write*). Dabei ist es unerheblich, ob der Operand im ↑Arbeitsspeicher (`i`) oder im ↑Registerspeicher (`r`) liegt.
Die Tatsache, dass eine Einzelaktion auf höherer Ebene als Aktionsfolge auf tieferer Ebene stattfinden kann ist nur logisch bedeutsam, wenn ein ↑nichtsequentieller Prozess für den ↑Programmablauf sorgt. In dem Fall kann unvermutet auf höherer Ebene eine ↑Wettlaufsituation auf tieferer Ebene die Folge sein und Berechnungsfehler bedingen, denen zwingend durch ↑Nebenläufigkeitssteuerung vorgebeugt werden muss.

Aktionsfolge (en.) *pattern of* ↑*action*s. Reihe von nacheinander oder gleichzeitig stattfindenden Aktionen. Dabei kann jede ↑Aktion gänzlich oder in Teilen auch demselben ↑Handlungsstrang zugehören.

aktives Register (en.) ↑*active register*. Bezeichnung für ein ↑Prozessorregister, das von einem ↑Handlungsstrang benutzt wird. Im Rahmen seines Optimierungslaufs versucht der ↑Kompilierer, den häufig verwendeten Werten (von ↑Variablen) des zu übersetzenden ↑Programms schnellen ↑Registerspeicher zuzuteilen (↑*register allocation*). Ein Register wird aktiviert, wenn der ↑Prozessor auf Veranlassung des Handlungsstrangs den ↑Maschinenbefehl zur Aufnahme (*load*) eines Werts (Konstante, Inhalt einer Programmvariablen) ausführt. Das Register bleibt für eine bestimmte Programmvariable aktiv, solange der Prozessor (auf Veranlassung des Handlungsstrangs) noch nicht den Maschinenbefehl zum Umspeichern (*store*) des Registerinhalts ausgeführt hat.

aktives Warten (en.) ↑*active waiting*, ↑*busy waiting*. Verfahren, bei dem ein ↑Prozess tatkräftig, in geschäftiger Art und Weise mittels einer ↑Verzögerungsschleife ein bestimmtes ↑Ereignis

erwartet: im Gegensatz zu ↑passives Warten, gibt der Prozess seinen ↑Prozessor während der gesamten ↑Wartezeit nicht ab und bleibt von sich aus laufend (↑Prozesszustand).
Ein solches ↑Warteverhalten führt nur bedingt zur effizienten Nutzung der ↑CPU. Im Falle einer ↑Betriebsart, bei der die CPU zu einem Zeitpunkt grundsätzlich nur ein einziges ↑Programm (↑*uniprogramming*) ausführt, wird unnötigerweise Energie für Nichtstun (↑*idle*) verbraucht. Können dagegen mehrere Programme im ↑Arbeitsspeicher zugleich zur Ausführung bereitgestellt sein (↑*multiprogramming*) oder erlaubt ein einzelnes ↑nichtsequentielles Programm zugleich mehr als einen ↑Kontrollfluss, liegt die CPU trotz anstehender Arbeitslast unnötigerweise brach.
Einen Prozess tätig warten zu lassen, lohnt sich wenn überhaupt nur bei kurzer Wartezeit — die dem Prozess genau im Entscheidungsmoment zum Warten und damit vorher bekannt sein müsste, was sie für gewöhnlich jedoch nicht ist. Nur wenn das zeitliche Verhalten des externen Prozesses, der das Ereignis hervorruft, vorhersagbar ist, etwa die ↑Zwischenankunftszeit von Eingabedaten oder von Ausgabebestätigungen, lässt sich die Wartezeit des (internen) Prozesses bestimmen. Bei Maschinen oder allgemein physikalischen Prozessen mit periodischem Verhalten in Bezug auf Ein-/Ausgabe ist solch eine Vorhersage durchaus möglich, nicht jedoch bei aperiodischem Verhalten (z.B. wenn der Mensch selbst als externer Prozess ein Glied in der Verarbeitungskette bildet, nämlich eine Eingabe liefern oder die Ausgabe bestätigen muss).

Aktivierungsblock (en.) ↑*activation record*. Bezeichnung für den beim Aufruf eines ↑Unterprogramms jeweils benutzten Teilbereich auf dem ↑Laufzeitstapel einer ↑Prozessinkarnation. Dieser Teilbereich umfasst für gewöhnlich folgende Strukturelemente:

- die Argumente (↑*actual parameter*) für das Unterprogramm, sofern erforderlich,
- die Rücksprungadresse zu dem übergeordneten Unterprogramm, das das betreffende Unterprogramm aufgerufen hat,
- die Inhalte der ↑Prozessorregister, die im Unterprogramm verwendet werden, aber für das übergeordnete Unterprogramm invariant bleiben müssen (↑*callee-saved register*),
- Platzhalter für die lokalen ↑Variablen (*automatic variables* in ↑C) und
- dem Unterprogramm nur vorübergehend bereitgestellter Behelfsspeicher, beispielsweise Platz für invariant zu haltende Inhalte von Prozessorregister, die jedoch ungesichert in einem anderen, aufzurufenden Unterprogramm verwendet werden dürfen (↑*caller-saved register*).

Die Größe und Reihenfolge der einzelnen Strukturelemente eines solchen Teilbereichs legt der ↑Kompilierer fest und schlägt sich unter anderem auch in der durch ihn definierten ↑Aufrufkonvention für Unterprogramme nieder. Dieser Bereich wird beim Aufruf in ein Unterprogramm automatisch aufgebaut und vor dem Rücksprung aus dem Unterprogramm automatisch zurückgebaut.
Der zur Argumentenübergabe erforderliche Bereich oben auf dem Stapel des aufrufenden Unterprogramms überlagert sich praktisch mit dem entsprechenden Bereich des aufgerufenen Unterprogramms. Für die Übergabe selbst gibt es zwei Herangehensweisen. In dem einen Ansatz legt das aufrufende Unterprogramm die Argumente zum Aufruf einfach auf den Stapel ab (*push*) und macht den Bereich nach Rückkehr aus dem Unterprogramm wieder frei (*pop*). Im anderen Ansatz verwendet das aufrufende Unterprogramm einen Übergabebereich fester (vom Kompilierer berechneter) Größe, der zur Aufnahme der Argumente dient.

Algol Abkürzung für *algorithmic language*, eine Familie von imperativen, prozeduralen, strukturierten Programmiersprachen. Die wichtigsten Varianten waren Algol 58, ↑Algol 60 und Algol 68, benannt nach dem Jahr der Erstveröffentlichung ihrer Spezifikationen. Ein weiterer Ableger war Algol W (Wirth und Hoare, 1966), der die spätere Entwicklung von ↑Pascal stark beeinflusst hat. Für gewöhnlich ist Algol 60 gemeint, wenn der Begriff fällt.

Algol 60 Programmiersprache der ↑Algol-Familie (1960), erhielt 1963 ihre endgültige Fassung.

alignment (dt.) ↑Ausrichtung.

allgemeiner Semaphor (en.) *general* ↑*semaphore*. Ein ↑Semaphor, bei dem die ↑Ablaufsteuerung eine *Ganzzahlvariable* benutzt, der einen ganzzahligen elementaren ↑Datentyp (`int`) besitzt. Die Operationen ↑P und ↑V sind zählende Operationen, daher auch *zählender Semaphor*. Zustandsänderungen des Semaphors geschehen als ↑Elementaroperation, wobei aber nur die arithmetischen ↑Aktionen jeweils als ↑atomare Operation ausgelegt sein können.
Die Zahlensemantik des im Semaphor integrierten Zählers ist entweder ganzzahlig ($\mathbb{Z}$) oder nichtnegativ ganzzahlig ($\mathbb{N}_0$). Entsprechend funktioniert die Ablaufsteuerung unterschiedlich. Bei der $\mathbb{Z}$ Variante dekementiert P beziehungsweise inkrementiert V unbedingt, wohingegen die $\mathbb{N}_0$ Variante die Zählerarithmetik bedingt durchführt und eben nicht im negativen Wertebereich operiert. Beiden Varianten gemeinsam ist allerdings die durch P erfolgende ↑Prozessblockade, wenn der Zähler auf null steht ($\mathbb{N}_0$) beziehungsweise null durchläuft ($\mathbb{Z}$). Darüber hinaus erlaubt die $\mathbb{Z}$ Variante die Anzahl der schlafenden Prozesse vom Zählerwert abzulesen, diese entspricht exakt dem Betrag des negativen Werts. Damit hat es V sehr leicht festzustellen, ob ein schlafender Prozess aufzuwecken ist. Demgegenüber muss dazu die $\mathbb{N}_0$ Variante die ↑Warteliste schlafender Prozesse prüfen: sie ist damit in der zwischen P und V durchzuführenden ↑Bedingungssynchronisation nicht unabhängig von den Funktionen, die die Warteliste verwalten und bereitstellen. Dies schränkt den Spielraum zur Implementierung des Semaphors ein. Die $\mathbb{N}_0$ Variante verhindert damit insbesondere eine Lösung, bei der ein ↑binärer Semaphor die Grundlage bildet (s. unten, Exkurs).
Typischer Anwendungsfall dieser Semaphorart ist die Verwaltung „zählbarer“ ↑Betriebsmittel, nämlich die Vergabeprozedur sowohl für ein ↑konsumierbares Betriebsmittel (↑*producer-consumer problem*) als auch für ein ↑wiederverwendbares Betriebsmittel (↑*bounded buffer*; begrenzte Anzahl von Hardware-/Softwareeinheiten jeder Art). Mit dieser Funktion ist zudem eine weitere Ausprägung als ↑privater Semaphor gebräuchlich, der lediglich eine Spielart des zählenden Semaphors ($\mathbb{Z}$ oder $\mathbb{N}_0$) darstellt.

Exkurs Nachfolgend wird die Idee aufgegriffen, den zählenden Semaphor auf Grundlage des binären Semaphors zu implementieren. Dabei kommt letzterer für die Bedingungssynchronisation zwischen der abwärts (*down*) und aufwärts (*up*) zählenden Primitive zum Einsatz. Zur besseren Abgrenzung zum binären Semaphor, für den P und V die relevanten Primitiven bilden, wird nachfolgend mit den dazu korrespondierenden Primitiven `down` und `up` eine entsprechend synonyme Symbolik verwendet. Der Datentyp für solch eine „Zählvorrichtung“ (*numerator*) sei wie folgt aufgebaut:

```
typedef struct {
    int load;               /* # of available resources/waiting processes */
    semaphore_t wait;                     /* process waitlist; initial: 0 */
} numerator_t;
```

Die Zählervariable (`load`) gibt einen Hinweis auf die mit dem Semaphor verwaltete „Last“ in Bezug auf freie ↑Ressourcen ($\mathbb{N}^*$) beziehungsweise schlafende Prozesse ($\mathbb{Z}^*_-$). Der mit dem Wert null vorzubelegende integrierte binäre Semaphor (`wait`) dient dazu, einen abwärts zählenden Prozess für Ergebnisse kleiner oder gleich null schlafen zu legen und diesen durch einen aufwärts zählenden Prozess wieder aufzuwecken, wenn das Ergebnis kleiner null ist. Das Abwärtszählen (`down`) ist wie folgt implementiert:

```
void down(numerator_t *sema) {
    if (FAA(&sema->load, -1) <= 0)             /* any resource available? */
        P(&sema->wait);                           /* no, wait for release */
}
```

Die verwendete Zähloperation, ↑FAA, tut zweierlei. Zum einen ruft sie den Wert des Zählers ab (*fetch*), bevor dieser verändert wird. Zum anderen addiert sie zu dem abgerufenen Wert eine Zahl (*add*), bevor die Summe in der Zählervariablen gespeichert wird. Beide Schritte zusammen bilden eine atomare Operation (*fetch and add*).

Ist der abgerufene Wert nicht positiv ($\mathbb{Z}_-$), also kleiner oder gleich null, legt sich der Prozess mit Hilfe des binären Sempahors (`wait`) in P schlafen. Hierbei ist die speichernde Semantik eines jeden Semaphors zu beachten, denn diese bewirkt, dass für die hier gezeigte Konstruktion ein zwischenzeitliches V (in `up`, s. später) und damit der Weckruf für einen sich schlafen legenden (noch nicht in P befindlichen) Prozess nicht verloren geht (↑*lost wake-up*).
Das Aufwärtszählen (`up`) geschieht nach einem sehr ähnlich Muster, nur ist hier für den Weckruf zu sorgen, wenn es wenigstens einen (anscheinend) schlafenden Prozess gibt:

```
void up(numerator_t *sema) {
    if (FAA(&sema->load, +1) < 0)                /* any waiting process? */
        V(&sema->wait);                           /* yes, signal release */
}
```

Der durch das V des integrierten binären Semaphors (`wait`) umgesetzte Weckruf geschieht nur, wenn der mit FAA abgerufene Wert negativ ($\mathbb{Z}^*_-$) ist. Dann, und nur dann, schläft ein (`down` durchführender) Prozess in P oder ist auf direktem Wege dahin.
Die hier gezeigte Lösung,) nämlich das Zählen durch eine atomare Operation zu bewerkstelligen und die Bedingungssynchronisation zwischen den zählenden Prozessen durch einen binären Semaphor zu erledigen, schränkt echte ↑Parallelverarbeitung auf der mit dem zählenden Semaphor eingeführten ↑Abstraktionsebene nicht ein. Diese Ebene praktiziert letztlich ↑nichtblockierende Synchronisation, um die beiden (a) zählenden und (b) Prozesse steuernden Primitiven, `down` und `up`, abzusichern. Dabei steht die Funktionsweise tieferer Ebenen, insbesondere die des verwendeten binären Semaphors beziehungsweise der Primitiven P und V, nicht im Vordergrund. Dass P zu einer regulären Prozessblockade führen kann, bedeutet nicht, dass damit zur Implementierung des zählenden Semaphors ↑blockierende Synchronisation geschieht! Vielmehr ist die hier stattfindende Bedingungssynchronisation prinzipielles Merkmal eines jeden Semaphors. Allerdings legt eine auf höherer Ebene bereits nichtblockierend ausgelegte Ablaufsteuerung ↑gleichzeitiger Prozesse die Grundlage, um eine solches Vorgehen auf tieferen Ebenen nicht als zwecklose Maßnahme erscheinen zu lassen, Denn wenn bereits „von oben ausgehend“ ein ↑sequentieller Prozess stattfindet, macht es für gewöhnlich wenig Sinn, diesen „unten laufend“ mit gleichzeitigen Prozessen, die in dem Kontext nicht mehr geschehen können, zu synchronisieren. Daraus ergibt sich letztlich auch der besondere Anreiz, den hier verwendeten binären Semaphor, sowie alle anderen tieferliegenden Funktionen, ebenfalls einer nichtblockierend synchronisierten Ablaufsteuerung zu unterziehen — was zwar nicht leicht, aber eben auch nicht unmöglich ist.

Alpha Bezeichnung für den als Nachfolger zur ↑VAX von ↑DEC entwickelten Mikroprozessor (1992): 64-Bit, ↑RISC, betrieben durch ↑OpenVMS und verschiedene ↑UNIX Dialekte.

Alterung (en.) ↑*ageing*. Bezeichnung für eine bestimmte Technik bei der ↑Prozesseinplanung, um möglichem ↑Verhungern vorzubeugen. Dabei fließt die ↑Wartezeit eines ↑Prozesses auf der ↑Bereitliste ein in die Entscheidungsfindung zur ↑Einlastung der ↑CPU. Typisches Beispiel dafür ist ↑HRRN und ↑MLFQ.

ambient noise (dt.) ↑Umgebungsrauschen.

AMD 64 Prozessorarchitektur, 64-Bit, abwärtskompatibel zu ↑x86, AMD (2000). Erste Produktfreigabe in 2003 (AMD Opteron „K8“).

AMP Abkürzung für (en.) ↑*asymmetric multiprocessing* und (en.) ↑*asymmetric multiprocessor*.

Angriffssicherheit (en.) ↑*security*. Zustand des Geschütztseins einer ↑Entität in einem ↑Rechensystem vor Gefahr oder Schaden durch einen böswilligen ↑Prozess (in Anlehnung an den Duden). Um diesen Zustand zu gewährleisten, sorgt ein ↑Betriebssystem für die Unwirksamkeit jeder ↑Aktion, die zur Verletzung der ↑Schutzdomäne anderer Prozesse führen kann. Grundsätzlich bedeutet dies, unautorisiertes Eindringen eines Prozesses in eine fremde Schutzdomäne zu erkennen und abzufangen (↑Ausnahmesituation). Dazu ist einem Prozess

mindestens der unautorisierte Zugriff auf bestimmte Entitäten zu verwehren, was im Falle von Zugriffen mittels dazu extra vorgesehenen Operationen (z.B. lesen/schreiben einer ↑Datei oder von einem ↑Speicherwort) noch vergleichsweise einfach ist. Ungleich schwieriger ist die Zugriffsabwehr, wenn ein Prozess mit legitimen Operationen Informationen, die nicht für ihn bestimmt sind, unbemerkt abgreifen kann (↑*covered channel*).
Jede ↑Schutzverletzung, die den Zustand oder die Funktion einer Entität (wie oben erwähnt) verändert, ist nicht nur illegitim, sonder kann insbesondere auch zur Beeinträchtigung der ↑Betriebssicherheit führen. Umgekehrt gilt dies nicht.

Ankunftsprozess (en.) ↑*arrival process*. Bezeichnung für einen (internen oder externen) ↑Prozess, der die Ankunft eines ↑Auftrags an einer ↑Bedienstation beschreibt. In einem von ↑Unterbrechungen und ↑Interferenz betroffenen ↑Rechensystem ist die Auftragsankunft für gewöhnlich durch einen *Zufallsprozess* charakterisiert.

Ankunftszeit (en.) ↑*arrival time*, a_i. Bezeichnung für eine ↑Prozessgröße, die den Zeitpunkt festlegt, zu dem eine ↑Aufgabe zur Verarbeitung zur Verfügung steht.
Eine Aufgabe ist zur Verarbeitung durch einen ↑Prozess verfügbar, nachdem sie ausgelöst wurde. Die Aufgabe ist im System vorhanden, liegt zur sofortigen Durchführung vor, muss aber als eine von möglichen vielen anderen Aufgaben/Teilaufgaben aus logischen (↑*causality*) oder physischen (↑*uniprocessor*) Gründen nacheinander verarbeitet und dazu entsprechend noch bereitgestellt werden. Damit ergibt sich für den Zeitpunkt, zu dem eine Aufgabe verfügbar ist für die Verarbeitung, folgendes Bild:

$$r_i^0 < a_i < r_i^+,$$

mit r_i^0 als ↑Auslösezeit und r_i^+ als ↑Bereitzeit. Typisches technisches Szenario ist etwa eine Aufgabe, die im Rahmen einer ↑Unterbrechungsbehandlung als Folge der zugehörigen ↑Unterbrechungsanforderung ausgelöst wird (r_i^0). Im weiteren Verlauf wird ein Prozess, der für die Verarbeitung dieser Aufgabe zuständig sein soll, der ↑Ablaufplanung zugeführt (a_i). Dieser Prozess kommt sodann gemäß ↑Einplanungsalgorithmus auf die ↑Bereitliste und wartet die Zuteilung des/eines ↑Prozessors ab (r_i^+).
Die Übergänge zwischen den einzelnen Vorgängen verlaufen fließend und die ↑Aktionsfolge für jeden Übergang nimmt (nicht nur) Zeit in Anspruch. Damit finden diese Übergänge unter realen Bedingungen auch zu unterschiedlichen Zeitpunkten statt, die betreffenden Vorgänge starten zu verschiedenen Zeiten. Für theoretische Betrachtungen ist diese Differenzierung jedoch nicht immer relevant, weshalb gelegentlich auch die Gleichsetzung der drei Zeitpunkte gilt: $r_i^0 = a_i = r_i^+$. Wenn etwa davon ausgegangen werden kann/muss, dass die gesamte ↑Verarbeitungskette ab Entgegennahme der Aufgabe an einer „Eingabestelle“ (z.B. ↑Peripheriegerät) bis hin zur Bereitstellung als vermeintlich ↑atomare Operation geschieht, besteht in logischer Hinsicht ↑Gleichzeitigkeit für diese drei Zeitpunkte — sie können dann durch einen einzigen Zeitpunkt modelliert werden.

Anmeldung (en.) ↑*login*. Prozedur, nach der sich eine Person oder ein externer ↑Prozess dem ↑Betriebssystem gegenüber zur Teilnahme am Rechenbetrieb und Aufnahme einer bestimmten Arbeit ankündigt. Die sich anmeldende ↑Entität wird authentifiziert und zugelassen, wenn sie berechtigt zur Teilnahme ist und ihr eine Mindestmenge an ↑Betriebsmittel in dem Moment bereitgestellt werden kann. Ist sie nicht berechtigt, wird ihr der Zugang zum ↑Rechensystem verwehrt. Besteht temporärer Betriebsmittelmangel, wird die Zulassung solange aufgeschoben, bis die Zusicherung der zur Arbeitsaufnahme benötigten Betriebsmittel in Aussicht steht (↑*long-term scheduling*). Nach erfolgter Zulassung wird (für den internen Prozess) eine ↑Prozessinkarnation angelegt, die die Entität im Rechensystem repräsentiert. Dieser Prozess nimmt seine Arbeit in einem bestimmten ↑Namensraum (↑*root file system*) auf, dort beginnend in seinem ↑Heimatverzeichnis.

anticipatory paging (dt.) ↑Vorausladen.

Antwortzeit (en.) ↑*response time*. Zeitspanne zwischen Absetzen eines Auftrags durch einen ↑Prozess und der Entgegennahme der Antwort daraufhin im selben Prozess. Beispielsweise die zur ↑Laufzeit im ↑Maschinenprogramm anfallende ↑Latenzzeit einer per ↑Systemaufruf abgeforderten und im ↑Betriebssystem durchgeführten ↑Systemfunktion.

Anwenderprogramm siehe ↑Anwendungsprogramm.

Anwendung (en.) ↑*application*. Bezeichnung für mindestens ein ↑Maschinenprogramm, das die für einen bestimmten ↑Anwendungsfall erforderlichen Funktionen festlegt. Dabei handelt es sich im Allgemeinen eben um nicht systemtechnische Funktionen, die nämlich nicht durch ↑Systemsoftware implementiert sind.
Im Falle von mehreren zusammengehörigen Maschinenprogrammen enthalten diese für gewöhnlich auch Anweisungen, die eine Kommunikation untereinander ermöglichen und der Kooperation bei der arbeitsteiligen Erfüllung von den mit den nicht systemtechnischen Funktionen verknüpften Aufgaben dienen. Dies bedeutet dann aber auch, dass mehr als ein ↑Prozess zugleich stattfinden muss, um die gewünschte nicht systemtechnische Funktionalität bereitzustellen. Allerdings sind dafür mehrere Programme alles andere als ein zwingendes Merkmal, denn die Anwendungsfunktionalität kann auch gut durch ein einzelnes ↑nichtsequentielles Programm beschrieben sein. Ebenso ist natürlich auch ein Mix aus sequentiellen und nichtsequentiellen Programmen möglich. Einzig die nicht systemtechnische Funktion bildet das charakteristische Merkmal und nicht etwa der Aspekt, wie diese Funktion systemtechnisch ausgelegt ist, also etwa ob sie als (ein- oder mehrelementige Menge von) ↑Faden, ↑Faser oder gar ↑Fäserchen in Erscheinung tritt.
Zu beachten ist, dass die Formulierung „nicht systemtechnische Funktion“ genau genommen relativistisch zu verstehen ist und oftmals ein bestimmtes, jedoch nicht immer explizit benanntes ↑Bezugssystem vor Augen hat. So beziehen sich beispielsweise Programme, die eine Datenbankanwendung bilden, auf ein Datenbanksystem, dessen konstituierenden Programme ihrerseits eine Betriebssystemanwendung darstellen. Noch weiter oben in der Hierarchie kann die Datenbankanwendung selbst als ein System von Dienstnutzer/-leister (*client-server model*) ausgelegt sein, wobei letztere durch Programme implementiert sind, die im Verbund eine bestimmte Webanwendung ausmachen.

Anwendungsfaden (en.) ↑*user thread*. Bezeichnung für einen ↑Faden der ↑Benutzerebene (↑*user-level thread*), der (a) von sich aus immer im ↑Benutzerbereich bleibt, im ↑Maschinenprogramm residiert, ausschließlich dessen Kode ausführt und (b) nur ausnahmsweise (↑*trap*, (↑*interrupt*) auf die ↑Betriebssystemebene in den privilegierten ↑Arbeitsmodus (↑*kernel mode*) wechselt. Gegensatz zu ↑Systemkernfaden.
Der Faden ist im Maschinenprogramm implementiert und entsteht nicht erst durch einen Faden im ↑Betriebssystemkern. Sämtliche für solch einen Faden erforderlichen Betriebsmittel sowie für seine Verwaltung nötigen Funktionen stellt das als ↑nichtsequentielles Programm ausgelegte Maschinenprogramm. Dies betrifft insbesondere den ↑Laufzeitkontext eines Fadens und die Funktionen zur ↑Ablaufplanung, ↑Einlastung und ↑Synchronisation.
Ein ↑Prozesswechsel, um innerhalb des Maschinenprogramms vom aktuellen zu einem anderen Faden umzuschalten, verläuft im lokalen ↑Adressraum und auch unter ↑Speicherschutz ohne ↑Systemaufruf. Jeder dieser Fäden ist vom Bautyp her ein ↑federgewichtiger Prozess, für den zur Steuerung und Überwachung keine kostspielige ↑Systemfunktion erforderlich ist und er selbst dazu auch keine benutzt. Für das ↑Betriebssystem ist ein solcher Faden kein ↑Objekt erster Klasse, das heißt, der durch den Faden konkretisierte ↑Prozess ist dem Betriebssystem gänzlich unbekannt.
Eine vom Maschinenprogramm aus (mittels Systemaufruf) beanspruchte Systemfunktion läuft damit nicht im Namen des effektiv beanspruchenden Fadens, sondern nur im Namen derjenigen ↑Entität ab, die im Betriebssystem das jeweils in Ausführung befindliche Maschinenprogramm repräsentiert. Ist dies bespielsweise ein ↑schwergewichtiger Prozess oder ein ↑leichtgewichtiger Prozess und wird dieser dann durch die Systemfunktion blockiert (↑Prozesszustand), blockieren implizit alle Fäden, die unbekannterweise eben auf diese eine

↑Prozessinkarnation durch eine ↑Aktion im Maschinenprogramm abgebildet wurden (↑*thread blocking anomaly*.

Anwendungsfall (en.) ↑*use case*. Beschreibung funktionaler und, wenn möglich, auch nicht-funktionaler Merkmale eines geplanten oder existierenden Systems. Dokumentiert wird insbesondere das nach außen (für den oder die Nutzer) sichtbare Verhalten dieses Systems. Je nach Detaillierungsgrad der Beschreibung sind Einzelheiten des Anwendungsverhalten mehr oder weniger abstrakt dargestellt. Mit zunehmend konkreter werdender Darstellung können Informationen gewonnen werden, die als ↑Vorwissen zu der jeweiligen ↑Anwendung in einem ↑Betriebssystem nutzbar sind. Beispielsweise kann dies Wissen über die Anzahl möglicher ↑Prozesse, der Art ihrer Interaktion (↑*shared memory*, ↑*message passing*), deren Begrenztheit, jeweilige Dauer (↑WCET), ↑Periode oder Bedarf an ↑Betriebsmitteln, sowie Kenntnis über ↑Systemaufrufe und damit zu Art und Zahl der überhaupt zur Anwendungsunterstützung benötigten ↑Systemfunktionen umfassen.

Anwendungsprogramm (en.) ↑*application program*, ↑*user program*. Bezeichnung einer ↑Anwendung, die aus genau einem ↑Maschinenprogramm besteht.

Anwesenheitsbit (en.) ↑*present bit*. Attribut einer ↑Seite oder von einem ↑Segment, gespeichert im entsprechenden Deskriptor der ↑MMU, das einen Hinweis über das Dasein (der Seite/des Segments) im ↑Prozessadressraum gibt. Eine Schaltvariable als Ausprägung eines booleschen Datentyps, die zwischen anwesend (`true`, Zugriff erlaubt) und abwesend (`false`, Zugriff verwehrt) unterscheidet. Die MMU signalisiert der ↑CPU, die durch den ↑Maschinenbefehl beschriebene ↑Aktion zu unterbrechen (↑*trap*), sobald dieser, spezifiziert durch die ↑Adressierungsart, Zugriffe auf abwesende Seiten/Segmente ausübt.
Ursprünglich wird durch dieses Attribut ↑virtueller Speicher in einem Prozessadressraum gekennzeichnet, nämlich alle für einen ↑Prozess gültige Seiten/Segmente, die zur Zeit jedoch nicht im ↑Hauptspeicher liegen. Allerdings kann es auch Verwendung finden, wann immer das ↑Betriebssystem den Zugriff auf einen bestimmten ↑Adressbereich eines Prozesses sofort erfahren möchte, unabhängig davon, ob die diesem Bereich zugeordneten Seiten/Segmente im Hauptspeicher residieren.

aperiodic task (dt.) ↑aperiodische Aufgabe.

aperiodische Aufgabe (en.) ↑*aperiodic task*. Eine in ↑Echtzeit zu bearbeitende ↑Aufgabe mit weicher (schwacher) oder fester ↑Termineinhaltung. Gegenteil von ↑sporadische Aufgabe.

APIC Abkürzung für (en.) *advanced* ↑PIC (Intel; erste Ausführung 82489DX, 1994). Die wesentliche Erweiterung im Verglich zum sonst gebräuchlichen PIC (Intel 8259, 1976; 8259A, 1981) besteht in der Möglichkeit, die ↑Unterbrechungsanforderungen der ↑Peripherie gezielt an bestimmte ↑Rechenkerne weiterleiten zu können. Dies schließt ein, dass Rechenkerne sich selbst gegenseitig Unterbrechungsanforderungen (↑IPI) zusenden können.
Dieses spezielle ↑Steuerwerk besteht aus zwei funktionellen Komponenten. Einerseits die lokale ↑Unterbrechungssteuereinheit (*local APIC*), die in jedem Rechenkern integriert ist und durch ↑speicherabgebildete Ein-/Ausgabe angesprochen wird (vgl. S. 247). Der gewöhnliche PIC (Intel) dagegen ist ein eigener ↑Halbleiterbaustein, der nebst ↑CPU und anderen Bausteinen typischerweise auf der ↑Hauptplatine eines ↑PC untergebracht ist und durch ↑isolierte Ein-/Ausgabe über den (System-) ↑Bus betrieben wird. Andererseits die optionale Unterbrechungssteuereinheit für die ↑Peripheriegeräte (↑*I/O APIC*) als eigener Halbleiterbaustein, die im wesentlichen die ursprüngliche Rolle des PIC übernimmt. Zur Weitergabe der Unterbrechungsanforderung eines Geräts an einen Rechenkern wird diese jedoch an die betreffende lokale Steuereinheit vermittelt, die schließlich die ↑Unterbrechung entsprechend den zuvor vom ↑Betriebssystem getroffenen Einstellungen bewirkt.
Ausgelöst durch Software (d.h., einen ↑Maschinenbefehl) kann die lokale Steuereinheit nicht nur Unterbrechungsanforderungen an Rechenkerne versenden und von anderen Rechenkernen oder (externen) Unterbrechungssteuereinheiten entgegennehmen, um sie ihrem Rechenkern

zuzustellen. Darüber hinaus läuft über sie die Vergabe der ↑Prozessoridentifikation eines Rechenkerns, die dann zur Verwendung in der Software abgerufen werden kann (vgl. S. 247).

App Kurzbezeichnung für (en.) ↑*application*. Ursprünglich nur das Kürzel für eine ↑Anwendung für programmierbare Mobiltelefone (*smartphone*). Mittlerweile aber auch die Bezeichnung für ein ↑Anwendungsprogramm für einen gewöhnlichen ↑Rechner.

application (dt.) ↑Anwendung.

application program (dt.) ↑Anwenderprogramm, ↑Anwendungsprogramm, ↑Benutzerprogramm.

Applikationsprogramm siehe ↑Anwendungsprogramm.

Arbeit (en.) ↑*job*, J_i. Tätigkeit mit einzelnen Verrichtungen beispielsweise als Teil einer ↑Aufgabe, Ausführung eines ↑Auftrags (in Anlehnung an den Duden). Art und Weise einer Aufgabe, die einem ↑Prozess oder einer Gruppe von Prozessen gestellt wird; eine bestimmte Aufgabenstellung für ein ↑Rechensystem.
Im Kontext von ↑Echtzeitsystemen die Ausführung einer (periodischen) Aufgabe. Eine Einheit, die vom System geplant und ausgeführt wird; eine bestimmte Stufe (*instance*) der Durch- oder Ausführung einer Folge von Verarbeitungsschritten einer Aufgabe, ein Verfahrensabschnitt.

Arbeitsentzug (en.) ↑*work stealing*. Bezeichnung für eine Variante der ↑Einlastung, bei der sich (bildlich gesprochen) ein ↑Prozessor selbst um ↑Prozesse bemüht, um nicht in den ↑Leerlauf eintreten zu müssen. Im Gegensatz zu ↑Arbeitsteilung wird ein Prozess aus der ↑Bereitliste eines anderen Prozessors „gestohlen“, sollte die eigene Bereitliste im Moment eines sonst anstehenden ↑Prozesswechsels leer sein. Sollte kein Prozess zum „Stehlen“ gefunden werden können, bedeutet dies, dass auch insgesamt keine ↑Aufgaben zur Bearbeitung anstehen und ein Prozessor daher nicht unnötigerweise untätig wird. Das bei Arbeitsteilung bestehende Problem, solche Phasen der Untätigkeit eines Prozessors nicht sicher abwenden zu können, stellt sich hier nicht.

Arbeitslast (en.) ↑*workload*. Maß für die von einem ↑Prozessor zu tragende Belastung durch die von ihm zu erledigenden ↑Aufgaben und dabei zu verarbeitenden ↑Daten.

Arbeitsmenge (en.) ↑*working set*. Minimalmenge des im ↑Hauptspeicher zu einem Zeitpunkt vorzuhaltenden Bestands von ↑Text und ↑Daten von einem ↑Prozess, damit dieser effizient stattfinden und seine Arbeit verrichten kann. Zur Aufbewahrung des Gesamtbestands wird ↑seitennummerierter virtueller Speicher als Grundlage genommen. Die Bestandsgröße ist ganzzahlige Vielfache einer ↑Seite.
Die Erfassung solcher Seiten dient der Modellierung von Prozessverhalten und damit auch einer Abschätzung der im ↑Rechensystem voraussichtlich noch zu erwartenden Vorgänge in Bezug auf den Hauptspeicher sowie dessen Nutzung. In dem Modell wird davon ausgegangen, dass in einem ↑Maschinenprogramm keine Anweisungen kodiert sind, um dem ↑Betriebssystem per ↑Systemaufruf Hinweise über den im Hauptspeicher erforderlichen Seitenbestand zu übermitteln. Vielmehr ist es Aufgabe des Betriebssystems eigenständig diesen Bestand zu ermitteln. Grundlage dafür ist die zurückliegende ↑Referenzfolge eines Prozesses, das heißt, die sich daraus ergebende Menge der im letzten ↑Zeitfenster referenzierten Seiten. Diese ↑Betriebsseiten bilden eine Teilmenge des im Hauptspeicher liegenden Seitenbestands (↑*resident set*), Zugriffe darauf haben keine ↑Seitenfehler zur Folge. Ziel ist es, die Anzahl der residenten Seiten (d.h., den von einem Prozess belegten Hauptspeicherplatz) zu minimieren und dadurch den Grad an ↑Mehrprogrammbetrieb (d.h., die Anzahl von Prozessen) zu maximieren.
Die exakte Anzahl der Betriebsseiten eines Prozesses ist bestimmt durch zwei Faktoren: (1) die Rechenvorschrift seines ↑Programms und (2) die — für gewöhnlich erst zur ↑Laufzeit bekannten — Eingabedaten dazu. Beide Faktoren definieren einen Prozess erst und, was

besonders zu beachten ist, sie sind gänzlich unabhängig von der Funktionsweise eines Betriebssystems. Demgegenüber ist die Anzahl der residenten Seiten dieses Prozesses vor allem eben auch eine Funktion des Betriebssystems. Das Betriebssystem muss das Programm — idealerweise sogar die zu erwartenden Eingabewerte — dieses Prozesses „verstehen“ und dazu den ↑Programmablauf erfassen und deuten, um die jeweils relevanten Betriebsseiten des betreffenden Prozesses zu kennen und dadurch die Anzahl seiner residenten Seiten minimieren zu können, ohne dabei plötzlichen Leistungsabfall (↑*thrashing*) zu befürchten. Allerdings greift hier nur eine gute Schätzung, da kein Betriebssystem die Programme, deren Abläufe es steuern und überwachen soll, funktional wirklich verstehen kann — das einzige, was das Betriebssystem schaffen kann, ist es, Parameter zum nichtfunktionalen Verhalten der durch diese Programme definierten Prozesse (mehr oder weniger gut) abzuleiten.
Um Hauptspeicherplatz für andere Prozesse zu schaffen, lässt die ↑Seitenumlagerung einzelne Betriebsseiten unangetastet: eine Arbeitsmenge wird niemals auseinandergerissen, da sonst das ↑Flattern von Betriebsseiten droht. Stattdessen wird diese Menge (Betriebsseiten) gegebenenfalls komplett umgelagert (*working-set* ↑*swapping*) und der zugehörige Prozess wird für die Dauer, während all seine Betriebsseiten ausgelagert sind, suspendiert. Zur Fortsetzung des Prozesses werden all diese Seiten im Verbund eingelagert (↑*pre-paging*).

Anmerkung Auch wenn, wie oben angedeutet, dem Modell nach davon ausgegangen wird, dass ein Maschinenprogramm keine Anweisungen enthält, um dem Betriebssystem Hinweise über den im Hauptspeicher zukünftig zu erwartenden Seitenbestand zu übermitteln, bedeutet dies nicht, ein Betriebssystem könnte in seiner Funktionsweise nicht davon profitieren, wenn es solche Hinweise aus erster Hand erhält. So darf normalerweise schon zu Recht angenommen werden, dass Seiten, die als frei im ↑Haldenspeicher ausgewiesen sind, keine Betriebsseiten sind und daher auch keine residenten Seiten des betreffenden Prozesses zu sein brauchen. Ebenso kann ein Prozess in Abhängigkeit von seinem Fortschritt für gewöhnlich sehr gut wissen, welche Teile seines Programms und damit auch ↑Adressraums er normalerweise niemals oder nie wieder durchstreifen wird. Ein Hinweis in der Freigabeoperation (`free(3)`) oder an definierten Programmstellen hilft dem Betriebssystem, die Anzahl der residenten Seiten eines Prozesses zu minimieren. Derartige Hinweise können dem Betriebssystem durch einen dafür geeigneten Systemaufruf (schwergewichtig: vergleichsweise hoher Aufwand, aber synchron und sofort verwertbar) oder über einen mit dem Maschinenprogramm gemeinsamen ↑Speicherbereich (leichtgewichtig: sehr geringer Aufwand, aber asynchron und nur verzögert verwertbar) zugestellt werden.
Allerdings darf sich das Betriebssystem auf solche Hinweise, so sie denn der Prozess überhaupt abliefern kann, allein nicht verlassen, denn sonst könnten (bewusst oder unbewusst) falsche Hinweise leicht ↑Seitenflattern hervorrufen, damit Systemlast provozieren, das nichtfunktionale Verhalten des Rechensystems beeinflussen (↑*covered channel*) und gegebenenfalls ↑Anwendungen schädigen. Daher ist die Annahme des Modells schon von großer praktischer Bedeutung — jedoch heißt das nicht, darauf zu verzichten, dem Betriebssystem Hilfestellung zum besseren eigenen Vorankommen zu leisten und damit indirekt auch zum „Wohlbefinden“ des Gesamtsystems beizutragen.

Arbeitsmodus (en.) ↑*operating mode*. Art und Weise des Handelns, Tätigwerdens, von einem ↑Prozessor (↑CPU oder ↑Rechenkern) im Rahmen der Ausführung von einem ↑Programm; (lat.) *modus operandi* (in Anlehnung an den Duden). Zu einem Zeitpunkt handelt der Prozessor entweder für das ↑Betriebssystem oder für ein ↑Maschinenprogramm, aber niemals für beide zugleich — indem er jedoch für das Betriebssystem handelt, kann dadurch sehr wohl die Ausführung des Maschinenprogramms voranschreiten (↑partielle Interpretation).
Ist das Betriebssystem aktiv (↑*system mode*), handelt der Prozessor privilegiert, das heißt, er führt jeden korrekt kodierten ↑Maschinenbefehl aus (↑*privileged mode*). Synonyme Bezeichnungen für diese Arbeitsweise fokussieren auf das jeweils aktive Subsystem innerhalb eines Betriebssystems, das heißt, sie beziehen sich auf das ↑Hauptsteuerprogramm (↑*supervisor mode*) oder den ↑Betriebssystemkern (↑*kernel mode*). In diesem Modus kann insbesondere

durch Setzen einer ↑Unterbrechungssperre der Prozessor dazu veranlasst werden, zeitweilig ungestört von einer ↑Unterbrechung zu handeln (↑*uninterruptible mode*).
Ist das Betriebssystem inaktiv, aber Arbeit ohne ↑Leerlauf zu leisten, handelt der Prozessor für das Maschinenprogramm (↑*user mode*) und damit nicht privilegiert (↑*unprivileged mode*). Der Wechsel in den Systemmodus geschieht bei jeder ↑Ausnahme, einschließlich ↑Systemaufruf. Der Wechsel zurück in den Benutzermodus erfolgt mit Beendigung einer ↑Ausnahmebehandlung, etwa wenn die partielle Interpretation eines Maschinenbefehls durch das Betriebssystem abgeschlossen ist.
Die Inbetriebnahme von dem ↑Rechner findet ebenfalls im Systemmodus statt. Zur erstmaligen Aufnahme der Ausführung eines Maschinenprogramms ist demzufolge ein (durch einen offensichtlich nicht abgesetzten Systemraufruf hinterlassener) Systemzustand nachzubilden und aufzusetzen, um vom System- in den Benutzermodus wechseln zu können.

Arbeitsspeicher (en.) ↑*main memory*, *working storage*. Bezeichnung für den zur Ausführung von einem ↑Programm und nur zu dessen ↑Laufzeit benötigten ↑Speicher. Im Wesentlichen ↑Hauptspeicher, eventuell jedoch erweitert um einen peripheren Speicher (↑*secondary storage*) zur Ablage von zeitweilig für die Programmausführung nicht erforderliche Bestände von ↑Text oder ↑Daten. Die Erweiterung ist funktional transparent oder intransparent, nämlich indem die Zugriffe auf den Erweiterungsspeicher in den Programmen selbst nicht formuliert sind (↑virtueller Speicher) beziehungsweise explizit durch spezielle Anweisungen zum Ausdruck kommen (↑*overlay memory*).
Die Verwendung des Begriffs als Synonym zu Hauptspeicher ist mit Vorsicht zu betrachten, sie ist unpräzise. Ein ↑Prozess muss zwar im Hauptspeicher stattfinden, damit er seine Arbeit verrichten kann, jedoch kann er bei ↑Speichervirtualisierung weit mehr als nur die direkt über den Hauptspeicher verfügbaren Text- und Datenbestände bearbeiten. Hinter Hauptspeicher steht sowohl in logischer als auch in physischer Hinsicht ein anderer Speicherbegriff: logisch, da ohne ihn grundsätzlich kein (von Neumann) ↑Rechner funktioniert, aber ein solcher Rechner ohne oben genannten Erweiterungsspeicher seine Funktion sehr wohl erfüllen kann; physisch, da seine Größe durch die ↑Speicherausstattung und nicht etwa durch die ↑Adressbreite des Rechners oder sein ↑realer Adressraum limitiert ist.

Arbeitsteilung (en.) ↑*work sharing*. Bezeichnung für eine Variante der ↑Einplanung von ↑Prozessen, bei der die zu bearbeitenden ↑Aufgaben auf ↑Prozessoren verteilt werden. Im Gegensatz zu ↑Arbeitsentzug wird die einem Prozessor jeweils zugeordnete ↑Bereitliste durch einen übergeordneten ↑Planer mit Prozessen befüllt.
Solange mehr Aufgaben zur Bearbeitung anstehen als Prozessoren verfügbar sind, ist es das Ziel, die Befüllung der Bereitliste eines Prozessors zeitlich so zu gestalten, dass es nicht zum ↑Leerlauf des betreffenden Prozessors kommt und er stattdessen immer etwas (sinnvolles) zu tun hat. Dazu ist die Aufgabenverteilung rechtzeitig anzustoßen, nämlich bevor eine der Bereitlisten leer ist, insgesamt aber noch genug Aufgaben anstehen. Der Zeitpunkt dafür ist bestimmt aus (a) der zur Aufgabenverteilung zu erwartenden ↑Ausführungszeit einerseits und (b) der zur Aufgabenbearbeitung zu erwartenden ↑Bedienzeiten pro Prozessor andererseits: die Differenz $(b) - (a)$ bestimmt die relative ↑Startzeit für die Aufgabenverteilung, um einen drohenden Leerlauf zu verhindern. Da diese Zeiten für gewöhnlich nur Schätzwerte sind und nicht selten überhaupt Unbekannte bleiben, wird der Leerlauf von Prozessoren nicht sicher verhindert werden können.

Arbeitsverzeichnis (en.) ↑*current working directory*. Bezeichnung für ein ↑Verzeichnis, das den gegenwärtigen ↑Namenskontext für einen ↑Prozess bildet (↑*current working directory*). In ↑UNIX ist diesem Verzeichnis der Name „.“ (↑*dot*) zugeordnet. Der Eintrag erlaubt die einfache Bestimmung vom eigenen Namenskontext, ohne den wirklichen, im ↑Elterverzeichnis eingetragenen, Kontextnamen kennen zu müssen.

architecture (dt.) ↑Architektur.

architecture feature (dt.) ↑Architekturmerkmal.

Architektur (en.) ↑*architecture*. Anordnung der Teile eines Ganzen zueinander; gegliederter Aufbau, innere Gliederung (Duden). Nach ↑Vitruv ist eine damit gemeinte Konstruktion auf drei Prinzipien begründet: *venustas*, Anmut, Schönheit, Wunsch; *firmitas*, Solidität, Stabilität, Wesentlichkeit; *utilitas*, Zweckmäßigkeit, Nützlichkeit, Funktion, Gut.

Architekturmerkmal (en.) ↑*architecture feature*. Eigenschaft einer ↑Rechnerarchitektur. Beispielswiese die Fähigkeit des ↑Prozessors zur ↑Parallelverarbeitung, die Art oder der Grad von ↑Speicherkonsistenz im System, das Spezifikum eines ↑Zwischenspeichers und ob dieser bei einem ↑Multiprozessor beziehungsweise ↑Mehrkernprozessor ↑Speicherkohärenz sicherstellt oder die Besonderheit funktional dedizierter Verarbeitungseinheiten. Darüber hinaus aber auch jede charakteristische Eigenheit der Hardware, die im ↑Programmiermodell sichtbar ist und Auswirkungen auf die Ausführung von ↑Programmen haben kann. Neben dem ↑Befehlssatz beispielsweise die ↑Bytereihenfolge oder die Fähigkeit, einzelnen oder gar allen ↑Rechenkernen eine eigene ↑Taktfrequenz geben zu können, damit dann verschieden schnell zu laufen und unterschiedlich viel Energie zu verbrauchen.

Archiv (en.) ↑*archive*. Bezeichnung für einen ↑Speicher zur langfristigen, dauerhaften Aufbewahrung von Informationen; geordnete Sammlung von Software und ↑Daten jeglicher Art in digitaler Form; ↑nichtflüchtiger Speicher. Die Speicherung erfolgt rechnerunabhängig (*off-line*). Zum Zugriff auf die gespeicherten Informationen ist der auf ↑Wechseldatenträger vorrätiger Speicher, manuell oder maschinell (Roboter), zuerst an das ↑Rechensystem anzuschließen. Auch ↑Tertiärspeicher genannt.

archive (dt.) ↑Archiv.

arithmetic overflow (dt.) ↑arithmetischer Überlauf.

arithmetischer Überlauf (en.) ↑*arithmetic overflow*. Überschreitung der bei endlicher Zahlendarstellung definierten Kapazitätsgrenze des Operanden einer arithmetischen Operation. In technischer Hinsicht ist dieser Operand das ↑Exemplar eines ↑Datentyps fester Breite, jeder seiner möglichen Werte ist als Dualzahl mit einer festen Anzahl von Bits repräsentiert. Beispielsweise deckt in ↑C der Datentyp `int` einen Wertebereich von mindestens $[-32767, +32767]$ beziehungsweise höchstens $[-2147483647, +2147483647]$ ab — der jeweilige negative Grenzwert ist der im Standard erlaubten Darstellung eines Zahlenwerts als Einerkomplement geschuldet, das gebräuchliche Zweierkomplement verschiebt diese untere Grenze um einen weiteren Zähler. Ein Exemplar dieses Typs ist damit durch 16 oder 32 Bits zusammenhängend im ↑Speicher dargestellt. Der Typmodifizierer `short` wird gemeinhin in dem Zusammenhang genutzt, um die kleinere (d.h., 16 Bits breite) Darstellung festzulegen. Für größere Darstellungen ist der Modifizierer `long` anzugeben, nämlich einfach (mind. 32 Bits) oder zweifach (mind. 64 Bits). Eine kleinere Darstellung wird durch `char` (mind. 8 Bits) erreicht. Tritt bei einer Berechnung bezogen auf einen bestimmten Datentyp ein Überlauf auf, wird dies im ↑Statusregister angezeigt (*overflow bit*).

ARM Abkürzung für Advanced ↑RISC Machines Ltd. (früher Acorn RISC Machines). Ein Unternehmen, das RISC-Entwürfe entwickelt und die Fertigung der Hardware Lizenznehmern (z.B., AMD, Apple, IBM, Intel) überlässt.

ARPANET Abkürzung für (en.) *Advanced Research Projects Agency Network*, 1969–1990. Ursprünglich im Auftrag der US-Luftwaffe am MIT entwickeltes ↑Rechnernetz. Vorläufer des heutigen Internet.

array (dt.) ↑Feld.

arrival process (dt.) ↑Ankunftsprozess.

arrival time (dt.) ↑Ankunftszeit.

ASCII Abkürzung für (en.) *American standard code for information interchange* (1963). Ein 7-Bit Kode, der 128 Zeichen definiert: 33 Steuerzeichen (nicht druckbar: [$\texttt{00}_{16}$, $\texttt{1F}_{16}$], $\texttt{7F}_{16}$) und 95 druckbare Zeichen ([$\texttt{20}_{16}$, $\texttt{7E}_{16}$]).

ASM86 Bezeichnung für einen ↑Assemblierer, Intel, für ↑x86. Erste Version 1981.

assembler (dt.) ↑Assemblierer.

assembler listing (dt.) Assemblerprotokoll, ↑Übersetzungsprotokoll.

Assemblieranweisung (en.) ↑*assembly instruction*. Anordnung, Befehl an den ↑Assemblierer einen ↑Maschinenbefehl zu generieren, einen Datenwert (Konstante) abzusetzen oder einen ↑Platzhalter für Datenwerte (↑Variable) zu reservieren. Die jeweiligen Anweisungen sind in ↑Assembliersprache formuliert, sie werden vom Assemblierer interpretiert und ausgeführt. Bei der ↑Interpretation wird der ↑Adresszähler um die Länge (in Bytes) der jeweiligen Anweisung erhöht, für Maschinenbefehle und Datenwerte wird zudem ↑Maschinenkode erzeugt.

Assemblieren (en.) ↑*assembly*. Vorgang der ↑Übersetzung von einem ↑Programm durch einen ↑Assemblierer. Das in ↑Assembliersprache vorliegende Programm (↑*source module*) wird in eine semantisch äquivalente Repräsentation in ↑Maschinensprache (↑*object module*) umgewandelt, wobei sich die bislang symbolische Darstellung des Programms auflöst und in eine nunmehr numerische Darstellung (↑Maschinenkode) übergeht.
Neben der Generierung von Maschinenkode sammelt der Assemblierer alle in der ↑Übersetzungseinheit verwendeten ↑Symbole, weist ihnen Attribute oder Werte zu und trägt sie in die ↑Symboltabelle ein. Normalerweise ist jedes in der Übersetzungseinheit definierte Symbol nur dort lokal gültig. Um es auch global gültig zu machen, muss das Symbol explizit einem ↑Pseudobefehl (z.B. `.globl`) übergeben werden. Symbole, die nicht in der Übersetzungseinheit definiert wurden, sind für gewöhnlich in einer anderen Übersetzungseinheit zu definieren und als global sichtbar auszuweisen. Einem definierten Symbol wird normalerweise auch ein Wert zugewiesen, beispielsweise den des bei der Symboldefinition gültigen ↑Adresszählers. Die Symboltabelle wird zusammen mit dem generierten Maschinenkode im ↑Objektmodul gespeichert und später vom ↑Binder verwendet, um den global ausgewiesenen Symbolen schließlich die entgültigen Werte (↑Adressen) zuzuordnen.
Die weitere wichtige Funktion des Assemblierers besteht im Aufbau der ↑Verlagerungstabelle, um dem Binder oder dem ↑Lader die Stellen mitzuteilen, an denen noch eine ↑Verlagerung durchzuführen ist. In dieser Tabelle ist für jedes Symbol, das einer Operandenadresse entspricht, verzeichnet, an welcher Stelle beim ↑Binden oder ↑Laden noch eine Adresskorrektur vorzunehmen ist.

Exkurs Zur Verdeutlichung des Übersetzungsvorgangs wird das für die ↑Kompilation betrachtete Beispiel (vgl. S. 136) weiterverfolgt und dazu das ↑Übersetzungsprotokoll als Ausgangspunkt genommen. Nachfolgend das entsprechende, unter ↑Linux mit dem Kommando `as --32 -march=i586 -a -f -W --listing-lhs-width 2` generierte Protokoll:

```
Zeile  ↑ILC  Maschinenkode          ↑Assemblieranweisung
  1                                           .text
  2                                           .p2align 4,,15
  3                                           .globl main
  4                                           .type main, @function
  5                                 main:
  6    0000   8D4C2404                        leal 4(%esp), %ecx
  7    0004   83E4F0                          andl $-16, %esp
  8    0007   FF71FC                          pushl -4(%ecx)
  9    000a   55                              pushl %ebp
 10    000b   89E5                            movl %esp, %ebp
 11    000d   51                              pushl %ecx
 12    000e   83EC10                          subl $16, %esp
 13    0011   68260000  00                    pushl $.LC0
 14    0016   E8FCFFFF  FF                    call puts
 15    001b   8B4DFC                          movl -4(%ebp), %ecx
 16    001e   83C410                          addl $16, %esp
 17    0021   C9                              leave
 18    0022   8D61FC                          leal -4(%ecx), %esp
 19    0025   C3                              ret
 20                                  .LC0:
 21    0026   48656C6C  6F20776F              .string "Hello world!"
 22           726C6421  00
```

Von links nach rechts stehen die Rubriken für die Zeilennummer im Quelltext, den Wert des ↑Adresszählers, den zweispaltig aufgetragenen Maschinenkode und der interpretierten beziehungsweise übersetzten Quelltextanweisung. Bis auf Zeilen 13 und 14 zeigt der Maschinenkode in der entsprechend Rubrik immer einen gültigen Wert. Da das Symbol `puts` in der ↑Übersetzungseinheit unbekannt ist, hat der Assemblierer die Scheinadresse $\texttt{FFFFFFFC}_{16}$ (↑Bytereihenfolge im Protokoll ist *little endian*!) als Operand für die Sprunganweisung (`call`) generiert. Die korrekte Adresse wird der Binder einsetzen, wenn er das — für gewöhnlich in der ↑libc definierte — Symbol `puts` hat auflösen können.
Hat der Binder das Symbol `puts` in einer ↑Programmbibliothek gefunden und in einen Wert auflösen können, entnimmt er die Stelle, an der auf Grundlage dieses Werts die berechnete Adresse im ↑Maschinenprogramm geschrieben werden soll, der Verlagerungstabelle. Diese Stelle entspricht im gegebenen Beispiel Zeile 14, Adresszählerwert $\texttt{0017}_{16}$. Ebenso wird mit dem Adresswert für die Marke `.LC0` verfahren, der an der durch den Adresszählerwert $\texttt{0012}_{16}$ zu Zeile 13 bestimmten Stelle in Bezug auf die ↑Ladeadresse des Maschinenprogramms beim Binden oder Laden ebenfalls zu korrigieren ist: obwohl das Symbol `.LC0` einen definierten Wert besitzt, ist dieser als *relative Adresse* nur innerhalb des Segments, auf das sich das Symbol bezieht, gültig, nicht jedoch als ↑absolute Adresse innerhalb eines aus mehreren Segmenten bestehenden Maschinenprogramms.

Assemblierer (en.) ↑*assembler*. Softwareprozessor, der ein in ↑Assembliersprache formuliertes ↑Programm in ein semantisch äquivalentes Programm in ↑Maschinensprache umwandelt.

Assembliersprache (en.) ↑*assembly language*. Bezeichnung für eine *hardwarenahe Programmiersprache*, um alle Bestandteile, die letztlich ein ↑Maschinenprogramm bilden, zusammenzustellen (*assemble*). Die Sprachelemente dienen der Platzierung und Anordnung von ↑Text und ↑Daten in den entsprechenden ↑Segmenten eines umfassenden ↑Adressraums, der symbolischen Darstellung von einem ↑Maschinenbefehl, der Vergabe von Namen für eine ↑Adresse, der Zuordnung von Attributen zu den Namen (für den ↑Binder) und der Deklaration eines solchen Namens als global sichtbar außerhalb der ↑Übersetzungseinheit. Eine ↑Assemblieranweisung in dieser Sprache ist unterteilt in vier Feldern wie folgt:

```
[label] mnemonic [operands] [comment]
```

Optionale Angaben sind (hier) durch eckige Klammern kenntlich gemacht. Ansonsten bezeichnen die Einzelkomponenten eine Marke (↑*label*), die dem an dieser Stelle geltenden Adresswert (↑Adresszähler) einen Namen gibt; ein als Gedächtnisstütze dienendes ↑Mnemon (*mnemonic*), das entweder den Operationsteil von einem ↑Maschinenbefehl oder einen ↑Pseudobefehl für den ↑Assemblierer benennt; der ebenfalls mnemonisch formulierte Operandenteil (*operands*) des Maschinenbefehls; sowie ein Kommentar (*comment*), um die Anweisung im semantischen Zusammenhang zu erklären. Ein Beispiel für ↑GAS und ↑x86 ist folgendes ↑Unterprogramm, das als Funktionswert den ↑PC-Inhalt von dem ↑Prozess liefert, der dieses Unterprogramm aufgerufen hat:

```
    .text                       # apply/add following stuff to text segment
    .p2align 4,,15              # enforce 16 byte alignment
    .globl whereami             # make symbol globally visible
whereami:                       # assume call instruction for invocation
    movl (%esp), %eax           # fetch program counter saved by call
    rts                         # return to where I came from
    .size whereami, .-whereami  # determine length of this function
```

Die hier durchgängig genutzten Kommentarfelder erläutern die Bedeutung jeder einzelnen Anweisung. Dabei ist (für GAS) jedes Kommentarfeld durch das Rautensymbol (`#`) gekennzeichnet, wodurch der nach diesem Symbol auf der Zeile stehende Text vom Assemblierer ignoriert wird. Die ↑Signatur der hier dargestellten Funktion lautet in ↑C:

```
extern int whereami(); /* determine caller's location inside text segment */
```

Pseudobefehle sind dem Assemblierer durch den anführenden Punkt kenntlich gemacht. Zunächst wird durch `.text` deklariert, dass der Bezug aller nachfolgenden Anweisungen zum ↑Textsegment ausgelegt ist.
Der Befehl `.p2align` sorgt für die ↑Ausrichtung des mit dem nächsten Maschinenbefehl (`movl (%esp), %eax`) beginnenden Programmabschnitts. Diese Anweisung hat drei Operanden: der erste Operand (`4`) gibt an, bis zu welcher Zweierpotenzgröße der Adresszähler weitergeschaltet und das Textsegment an dieser Stelle aufgefüllt werden soll; der zweite Operand (leer) definiert den Wert, der als Füllung verwendet werden soll; der dritte Operand (`15`) steht für eine Byteanzahl, um die der Adresszähler höchstens weitergeschaltet wird. Der Effekt dieser Anweisung ist, dass der vom Assemblierer bestimmte Wert des Symbols `whereami` nächstes ganzzahlig Vielfaches relativ zum aktuellen Adresszählerwert sein wird: hat der Adresszähler beispielsweise den Wert 42, wird `whereami` den Wert $2^4 \times 3 = 48 \geq 42$ erhalten und damit eine Lücke von sechs Bytes im Textsegment geschaffen.
Um sicherzustellen, dass der ↑Prozessor bei Ausführung der das Textsegment bildenden Maschinenbefehle trotz möglicher (logisch) offener, leerer Stellen korrekt weiterarbeiten kann, ist eine solche Lücke (physisch) gegebenenfalls durch einen oder mehrere ↑Leerbefehle definiert. Ein derartiges Auffüllen ist jedoch nur dann wirklich notwendig, wenn die Lücke innerhalb einer sonst statisch zusammenhängenden Befehlsfolge entsteht. In dem hier gezeigten Beispiel ist der erste Maschinenbefehl durch eine ↑Sprungmarke (`whereami`) gekennzeichnet, die symbolisch das Sprungziel für einen möglicherweise erfolgenden ↑Unterprogrammaufruf repräsentiert. Davor befindet sich in der betreffenden Funktion kein Maschinenbefehl, der im Falle eines Aufrufs ausgeführt werden müsste. Daher ist hier ein Auffüllen der möglicherweise (durch `.align`) generierten Lücke überhaupt nicht erforderlich. Eine andere Situation ergibt sich für eine Befehlsfolge, die beispielsweise zu einer Programmschleife gehört. In dem Fall wird ein Teil der Befehlsfolge, der Schleifenrumpf, mehrfach durchlaufen. Der Anfang eines solchen Rumpfes ist ebenfalls durch eine Sprungmarke gekennzeichnet, vor der dann für gewöhnlich auch Maschinenbefehle stehen, die im Zusammenhang, nämlich beim Eintritt in die Programmschleife, zu durchlaufen sind. In solch einer Konstellation ist Auffüllen der Lücke durch einen Leerbefehl erforderlich.
Der Befehl `.global` stellt die globale Sichtbarkeit von Symbolen (hier: `whereami`) her. Ohne diese Angabe hätten alle in einer Übersetzungseinheit deklarierten Symbole nur lokale

Bedeutung, sie wären für andere Übersetzungseinheiten effektiv „unsichtbar“: der ↑Binder verwendet den Wert eines lokalen Symbols nicht, um eine ↑unaufgelöste Referenz gleichen Namens, die in einem anderen ↑Objektmodul vermerkt ist, aufzulösen. Indem `whereami` als globales Symbol ausgezeichnet ist, kann die Funktion dieses Namens im gesamten Maschinenprogramm letztlich auch aufgerufen werden.
Der Befehl `.size`, schließlich, gibt einem Symbol ein Größenattribut. Konkret führt der Assemblierer hier folgende Berechnung durch: $loc(.) - loc(whereami)$, wobei *loc* für einen Adresszählerwert steht und mit dem Punkt (in GAS) auf den jeweils aktuellen Adresszählerwert Bezug genommen wird. Ergebnis ist die von der Funktion namens `whereami` im Textsegment belegte Anzahl von Bytes, das heißt, den für diesen Textabschnitt nötigen ↑Speicherbereich.
Diese Pseudobefehle werden vom Assemblierer ausgeführt, wohingegen die beiden Maschinenbefehle (`mov` und `rts`) zur Ausführung durch die ↑CPU gedacht sind. Dazu generiert der Assemblierer für diese Maschinenbefehle den ↑Maschinenkode, der gegebenenfalls noch in einem weiteren Schritt durch den ↑Binder (in Abhängigkeit von der ↑Ladeadresse einerseits und den ↑Adressen der Operanden andererseits) korrigiert wird.

assembly (dt.) Versammlung, Aufstellung; Programmumwandlung; das ↑Assemblieren.

assembly instruction (dt.) ↑Assemblieranweisung.

assembly language (dt.) ↑Assembliersprache.

AST Abkürzung für (en.) ↑*asynchronous system trap*.

asymmetric multiprocessing (dt.) ↑asymmetrische Simultanverarbeitung.

asymmetric multiprocessor (dt.) asymmetrischer ↑Multiprozessor, auch ↑asymmetrisches Multiprozessorsystem.

asymmetric scheduling (dt.) ↑asymmetrische Planung.

asymmetrische Planung (en.) ↑*asymmetric scheduling*. Modell der ↑Ablaufplanung, die wenigstens einen ↑Multiprozessor oder ↑Mehrkernprozessor zur Grundlage hat. Anders als ↑symmetrische Planung, sorgen die Verfahren vordergründlich nicht für eine gleichmäßige Verteilung der ↑Prozesse über die einzelnen ↑Prozessoren (↑*asymmetric multiprocessing*).
Der natürliche Grund dafür kann ein ↑asymmetrisches Multiprozessorsystem und der damit gegebene asymmetrische Aufbau der zugrunde liegenden Hardware sein. So ist der von einem Prozess generierte Befehlsstrom immer prozessorabhängig. Dies gilt für gewöhnlich für das gesamte ↑Programm, das den Prozess definiert, kann jedoch auch abschnittweise festgelegt sein: in jedem Fall ist aber dem Prozess ein Prozessor zuzuteilen, der den von diesem Prozess jeweils generierten Befehlsstrom auszuführen vermag. Die ↑Einplanung muss auf die charakteristischen Merkmale der Prozesse und ihrer Prozessoren Rücksicht nehmen.
Andererseits kann allein das ↑Betriebssystem diese Form der Ablaufplanung vorgeben, insbesondere bei symmetrischer Auslegung der Hardware. Hier ist vor allem die ↑Architektur des Betriebssystem zu nennen, die etwa aus Gründen der ↑Synchronisation und, in dem Zusammenhang, zur Reduzierung von ↑Interferenz die Prozessoren logisch getrennt voneinander betreibt. Dazu ist jedem Prozessor eine eigene ↑Bereitliste zugeordnet, das heißt, auf die diese Liste implementierende Datenstruktur wird nicht von verschiedenen Prozessoren aus zugegriffen. Die Prozesse, die diese Listen letztlich füllen und leeren, sind nicht gekoppelt, sie agieren rein lokal, müssen sich nicht mit Prozessen anderer Prozessoren synchronisieren und können ungestört voranschreiten. Der Nachteil dieser Organisation besteht dann jedoch darin, dass ein Prozessor in den ↑Leerlauf eintreten kann, obwohl ein anderer Prozessor noch mehr als genug ↑Aufgaben zu erledigen hat.
Zur Abhilfe dieses Problems unausgewogener Arbeitslasten gibt es zwei Ansätze, ↑Arbeitsteilung und ↑Arbeitsentzug, die zwar nicht zu einer zwingend anhaltenden Kopplung der auf die Listen zugreifenden Prozesse führen, jedoch die betreffenden Prozesse nicht mehr ohne

gegenseitige Beeinflussung stattfinden lassen: erfolgen Arbeitsteilung oder -entzug gleichzeitig zu sonst lokalen Operationen, die den Zustand einer Bereitliste verändern können, müssen die beteiligten Prozesse sich synchronisieren und werden sich dadurch auch zwingend gegenseitig beeinflussen. Diese mögliche Beeinflussung greift jedoch erst, wenn (1) die Untätigkeit einzelner Prozessoren droht und (2) mehr Aufgaben im ↑Prozesszustand bereit sind als Prozessoren zur Verfügung stehen.
Ein anderer Aspekt, der zur asymmetrischen Nutzung einer sonst symmetrisch ausgelegten Hardware führt, ist die Reservierung von Prozessoren für Prozesse, die ganz bestimmte Funktionen ausüben. Dies kann Prozesse sowohl einer ↑Anwendung als auch des Betriebssystems betreffen, die an einen Prozessor verankert und diesen gegebenenfalls sogar exklusiv nutzen werden. An sich gibt es wenige ↑Systemfunktionen, die nicht derart auf eigene Prozessoren platziert werden könnten. Entscheidend ist die strukturelle Einbindung dieser Funktionen im Betriebssystem, wie häufig sie aktiviert werden und wie kostspielig dann ihr „Fernaufruf" im Vergleich zu ihrer ↑Ausführungszeit ist. Beispielsweise kann es sich ab einer bestimmten Anzahl wettstreitiger Prozesse lohnen, wenn bereits ein ↑kritischer Abschnitt einen eigenen Prozessor zur Ausführung zugeteilt bekommt. Der sequentielle Durchlauf der wettstreitigen Prozesse sorgt für starke Lokalität der ↑Daten im ↑Zwischenspeicher, wodurch die Systemleistung erheblich verbessert werden kann.

asymmetrische Simultanverarbeitung (en.) ↑*asymmetric multiprocessing*. Eine Variante der ↑Simultanverarbeitung, die ein ↑asymmetrisches Multiprozessorsystem zur Grundlage hat.

asymmetrisches Multiprozessorsystem (en.) ↑*asymmetric multiprocessor* (↑AMP). Bezeichnung für ein ↑Rechensystem, das aus wenigstens einen ↑Multiprozessor oder ↑Mehrkernprozessor aufgebaut ist und dessen Recheneinheiten asymmetrisch ausgelegt sind. Die Asymmetrie ist in der ↑Heterogenität der Hardware begründet, etwa wenn der ↑Befehlssatz der Prozessoren uneinheitlich ist und demzufolge Prozesse nur auf die für sie geeigneten Prozessoren stattfinden sollten (↑*asymmetric scheduling*). Typisches Beispiel dafür ist ein aus ↑CPU und ↑GPU bestehendes Rechensystem, wobei jede Klasse dieser ↑Prozessoren jeweils durch mehrere ↑Exemplare ausgeprägt sein kann.

asynchrone Ausnahme (en.) *asynchronous* ↑*exception*. Eine ↑Ausnahme, die von dem betrachteten ↑Prozess nicht selbst verursacht wurde. Der durch diesen Prozess beschriebene ↑Programmablauf wird unterbrochen (↑*interrupt*), dadurch verzögert, aber zu einem späteren Zeitpunkt fortgesetzt. Verursacher einer solchen Ausnahme ist ein in Bezug auf diesen Programmablauf abweichender (externer) Prozess, der gegebenenfalls auf einem anderen ↑Prozessor oder in der ↑Peripherie stattfindet. Die auf dem Prozessor dieses Programmablaufs entstehende ↑Ausnahmesituation ist unvorhersehbar und nicht reproduzierbar.
Typisches Beispiel dafür ist eine ↑Unterbrechungsanforderung (↑IRQ, ↑NMI), aber auch ein ↑Seitenfehler — wobei letzterer jedoch differenziert zu betrachten ist: Diese Ausnahme ist nur dann asynchron, wenn (a) ↑virtueller Speicher die Grundlage für den Prozess bildet und (b) die ↑Ersetzungsstrategie eine globale Wirkung ausübt (vgl. ↑synchrone Ausnahme).

asynchrone Systemfalle (en.) ↑*asynchronous system trap*, ↑AST. Eine ↑asynchrone Ausnahme im ↑Betriebssystem, Bezeichnung sowohl für einen Mechanismus in einem (realen/abstrakten) ↑Prozessor als auch für eine Art von ↑Ereignis auf ↑Systemebene: der plötzliche Aufruf eines ↑Unterprogramms im Betriebssystem außerhalb des Hauptausführungsstrangs. Die Idee besteht darin, als Folge einer bestimmten Operation (↑*system mode*) einen Vorgang asynchron im Betriebssystem stattfinden zu lassen, bevor die zuvor unterbrochene Ausführung (↑*user mode*) eines ↑Maschinenprogramms wieder aufgenommen und dadurch die ↑Benutzerebene reaktiviert wird. Ein in ↑RSX 11 erstmalig verwirklichtes Konzept für den ursprünglichen Zweck, eine in Bearbeitung befindliche ↑Aufgabe über das Auftreten eines bestimmten Ereignisses (z.B. Beendigung der ↑Ein-/Ausgabe) zu informieren.
Der Mechanismus ist dem herkömmlichen Aufruf eines ↑Unterprogramms nicht ganz unähnlich, nur dass dieser eben asynchron zugestellt wird (↑DPC) und einen speziellen ↑Ausnahmehandhaber aktiviert. Auslöser ist eine auf Systemebene und durch Software initiierte

↑Unterbrechungsanforderung, die aber erst behandelt wird, wenn das Betriebssystem seine Aufgabe (z.B. einen ↑Systemaufruf oder eine ↑Unterbrechungsbehandlung) beendet hat und beabsichtigt, wieder in Bereitschaft zu gehen (d.h., sich zu deaktivieren) und zur Benutzerebene zurückzukehren. Typisches Beispiel für solch eine Funktion ist die Auslösung einer Unterbrechungsbehandlung zweiter Stufe (↑SLIH).
Mit der ↑VAX, später mit dem ↑Alpha, wurde der Mechanismus direkt durch Hardware unterstützt. Um einen AST auszulösen muss die Nummer der gewünschten Privilegebene (auch Zugriffsmodus) in ein spezielles ↑Prozessorregister (`PR$_ASTLVL`: *processor AST level register*) eingetragen werden. Zusätzlich wird ein ↑Deskriptor, der den AST-Handhaber (SLIH) identifziert, in eine ↑Warteschlange eingereiht. Sobald die CPU aus der Unterbrechungsbehandlung zurückkehrt (`REI`: *return from exception or interrupt*), prüft sie den dabei zu reaktivierenden Zugriffsmodus. Hat der aktuelle Modus höhere Privilegien als der zu reaktivierende Modus und ist ein AST anhängig, aktiviert die CPU automatisch den entsprechenden Unterbrechungshandhaber und startet diesen auf ↑Unterbrechungsprioritätsebene 2. Dieser Handhaber arbeitet alle in der AST-Warteschlange enthaltenen Aufgaben ab.
Die ↑PDP 11 Familie bot rudimentäre Unterstützung durch im ↑PSW enthaltene Statusinformationen: dem T-Bit (für ↑*trap*) und einer Angabe darüber, welcher ↑Arbeitsmodus vor der Unterbrechung aktiv war (*previous mode*). Kommt es zu einer Unterbrechung, sichert die CPU automatisch unter anderem das aktive PSW auf den ↑Laufzeitstapel der Systemebene und wechselt den Arbeitsmodus entsprechend. Das T-Bit kann in jedem Arbeitsmodus gesetzt werden, um nach der Ausführung eines ↑Maschinenbefehls eine ↑Ausnahme hervorzubringen. Für gewöhnlich wird damit die Fehlersuche (*debugging*) unterstützt. Um nun einen AST auf der Systemebene auszulösen, muss das T-Bit im gesicherten PSW gesetzt werden. Dazu prüft das Betriebssystem vorher, in welchen Arbeitsmodus bei Beendigung der Unterbrechungsbehandlung (FLIH) zurückgekehrt wird. Ist im aktiven PSW die Benutzerebene (*user mode*) als voriger Arbeitsmodus ausgewiesen, wird das T-Bit im gesicherten PSW gesetzt. Im nächsten Schritt muss das Betriebssystem die CPU mittels `RTI` (*return from interrupt*) — nicht jedoch mit `RTT` (*return from trap*) — aus der Unterbrechungsbehandlung zurückkehren lassen, um nämlich bei gesetztem T-Bit im dann vom Laufzeitstapel wieder hergestellten PSW noch vor Ausführung des nächsten Maschinenbefehls (auf Benutzerebene) sofort wieder eine Unterbrechungsbehandlung aufzunehmen.
Fehlt jedoch Hardwarehilfe, die (a) die Rückkehr zur Benutzerebene erkennt sowie (b) im Falle dieser Rückkehr und des damit erfolgten vollständigen Abbau des Laufzeitstapels automatisch eine ↑Ausnahmebehandlung einleitet, lässt sich der Mechanismus ansatzweise in Software nachbilden. Eine Möglichkeit besteht darin, den im Moment einer ↑Ausnahme von der CPU auf den Laufzeitstapel der Systemebene gesicherten ↑Prozessorstatus zu ersetzen. Der neue Prozessorstatus enthält alle Informationen, um mit Beendigung der regulären Unterbrechungsbehandlung automatisch den Unterbrechungshandhaber für den AST zu starten anstatt sofort die unterbrochene Programmausführung auf Benutzerebene wieder aufzunehmen. Dazu ist zuvor jedoch der ursprüngliche Prozessorstatus zu sichern, um diesen später reaktivieren zu können, nachdem der AST behandelt wurde. Dies erfordert Kenntnis von der Prozessorstatuslage auf dem Laufzeitstapel, und zwar im Zuge der AST-Signalisierung. Da der AST erst wirksam wird, wenn zur unterbrochenen Programmausführung auf Benutzerebene zurückgekehrt werden soll, liegt der zu ersetzende Prozessorstatus immer auf dem Stapelboden. Die betreffende ↑Adresse im ↑Hauptspeicher ist bekannt, wenn die CPU über einen eigenen ↑Stapelzeiger — samt passend ausgerichteten und dazugehörigen Laufzeitstapel (vgl. S. 254) — für die Systemebene verfügt oder für das Betriebssystem ein eigener ↑logischer Adressraum eingerichtet ist.
Eine andere Möglichkeit besteht darin, den Aufruf des Unterbrechungshandhabers erster Stufe (↑FLIH) in eine spezielle ↑Mantelprozedur einzubetten. Diese Prozedur führt beispielsweise Buch über die aktiven Handhaberinkarnationen, das heißt, zählt deren Verschachtelungen, und löst den AST aus, wenn keine verschachtelte Inkarnation mehr vorliegt. In dem Fall existiert nur noch eine Handhaberinkarnation, nämlich diejenige, die der Ausgang für die Kaskadenbildung war. Normalerweise erfolgt bei Auflösung dieser ersten/letzten Inkar-

nation die Rückkehr zur Benutzerebene, ausnahmsweise wird in dem Moment jedoch bei anhängigem AST eine zusätzliche Unterbrechungsbehandlung durchgeführt.
Das Problem einer solchen Nachbildung ist es, die Verbuchung einer jeden Handhaberinkarnation sicherzustellen. Auch wenn dies durch eine einfache Zähloperation bewerkstelligt werden kann, ist jedoch bei weitem nicht garantiert, dass jede Zählung erfolgt: kritisch ist hier die ↑kaskadierte Unterbrechung. Sogar wenn die Zähloperation der erste Maschinenbefehl eines jeden Unterbrechungshandhabers ist, hängt es von der CPU ab, ob dieser Befehl unter allen Umständen zur Ausführung kommt. Erlaubt die CPU bereits die nächste Unterbrechung vor dem ersten Befehl im Unterbrechungshandhaber, wird die neue Inkarnation nicht gezählt. Demzufolge kann mehr als eine Inkarnation aktiv sein, ein AST dürfte somit nicht behandelt werden. Allerdings ist dieser Umstand aufgrund einer noch nicht geschehenen Zählung nicht immer erkennbar, weshalb fälschlicherweise ein AST im falschen Moment zur Behandlung kommt. Wenn dann zu dieser Behandlung noch eine niedrigere Unterbrechungsprioritätsebene eingestellt wird, was für gewöhnlich der Fall ist, kann der Laufzeitstapel im Betriebssystem überlaufen, wodurch möglicherweise ↑Panik entsteht.

Exkurs Für den ↑x86 kann trotz ↑NMI für gewöhnlich eine sichere Nachbildung erreicht werden. Zum einen deshalb, weil die CPU die Kaskadierung eines NMI nicht zulässt: erst wenn der Handhaber für den NMI seine Ausführung vollständig beendet hat (`iret`), wird der nächste NMI zugelassen. Zum anderen profitiert die Nachbildung von der Tatsache, dass die CPU, je nach Modell, kaskadierte Unterbrechungen der Art ↑IRQ entweder nicht unterstützt oder programmierbar macht. Unterbrechungsprioritätsebenen führt erst der ↑PIC ein, x86 und damit die CPU kennt nur eine Ebene. Vor diesem Hintergrund startet die Unterbrechungsbehandlung wie folgt:

```
#define __attribute__ ((interrupt)) void irqstub_t

irqstub_t irq000(train_t *this) { guide(flih000, this); }
irqstub_t irq001(train_t *this) { guide(flih001, this); }
/* ... */
irqstub_t irq256(train_t *this) { guide(flih256, this); }
```

Für jeden IRQ ist ein ↑Unterbrechungshandhaberstumpf definiert, der lediglich den Aufruf an die zentrale Mantelprozedur (`guide`) kodiert, um den weiteren Verlauf in die entsprechenden Bahnen zu lenken. Die Aufrufparameter identifizieren den eigentlichen Unterbrechungshandhaber (FLIH) und den auf dem Laufzeitstapel gesicherten (minimalen) ↑Prozessorstatus des unterbrochenen ↑Prozesses. Die dementsprechende Mantelprozedur für einen Unterbrechungshandhaber erster Stufe, die einen AST, nämlich den Unterbrechungshandhaber zweiter Stufe, letztlich durch das System leitet, sei wie folgt ausgelegt:

```
__attribute__ ((fastcall)) void guide(flih_t flih, train_t *this) {
    stage(this, ENTER);                  /* enter IRQ handler incarnation */

    admit(IRQ | EDGE);             /* take OS priority, if edge triggered */
    flih(this);                                     /* invoke actual FLIH */
    avert(IRQ | EDGE);       /* return to CPU priority, if edge triggered */

    if (stage(this, PROVE | USER)) {    /* on the way back to user level? */
        while (backlog(annex())) {               /* yes, any pending AST? */
            admit(IRQ);                          /* yes, take OS priority */
            flush();                              /* forward pending ASTs */
            avert(IRQ);                         /* return to CPU priority */
        }
    }

    stage(this, LEAVE);                  /* leave IRQ handler incarnation */
}
```

Für gewöhnlich beginnt die Ausführung dieser Prozedur mit implizit gesetzter ↑Unterbre-

chungssperre (*interrupts disabled*), das heißt, der durch die Hardware in dem Moment vorgegebenen Unterbrechungsprioritätsebene der CPU. Gegebenenfalls ist diese Prioritätsebene in der CPU oder dem PIC auch programmierbar, woraufhin die totale Unterbrechungssperre nicht zwingend ist. Keinesfalls darf jedoch eine unkontrollierte Rücknahme dieser vorgegebenen Sperre geschehen, da sonst, insbesondere bei ↑Pegelsteuerung, der Laufzeitstapel überlaufen und dadurch Panik hervorgerufen werden könnte. Den Prozedurrumpf klammert eine Operation (`stage`) ein, um bei fehlender Hardwarehilfe für einen AST dennoch (z.B. durch Zählen, s.o.) die Verbuchung von Handhaberinkarnationen zu ermöglichen. Bei gegebener Hardwarehilfe (x86 ab ↑i286) ist diese „Operationsklammer“ wirkungslos.
Nach dem Betreten der Handhaberinkarnation startet die eigentliche Unterbrechungsbehandlung erster Stufe durch Aufruf (`flih`) der dem FLIH entsprechenden Prozedur. Liegt dem IRQ ↑Flankensteuerung zugrunde, darf vor dem Aufruf — aber erst nach eventuell erforderlicher Inkarnationsverbuchung (`stage: ENTER`) — die Unterbrechungssperre aufgehoben, das heißt, zur Prioritätsebene des Betriebssystems gewechselt werden. Nach Rückkehr aus dem FLIH wird wieder zur Prioritätsebene der CPU zurück gegangen, um nämlich dem Problem (↑*lost wake-up*) vorzubeugen, einen beim Verlassen der Unterbrechungsbehandlung eventuell anhängig werdenden AST zu verpassen.
Darf ein AST geliefert werden, weil die Rückkehr zur Benutzerebene ansteht (`stage: PROVE`), wird der Auftragsbestand (`backlog`) geprüft und gegebenenfalls die Auftragswarteschlange (`annex`) abgearbeitet. Beim x86, und zwar ab dem Modell i286, kann für diese Prüfung der auf dem Laufzeitstapel beim ↑Unterbrechungszyklus gesicherte (und hier durch `this` erreichbare) ↑Segmentwähler für das ↑Textsegment genutzt werden. Die Benutzerebene ist normalerweise mit den geringsten Privilegien (↑DPL 3) ausgestattet. Gibt der gesicherte Segmentwähler eben diese Privilegstufe (mit dem Wert 3) vor, wird angenommen, dass bei Rückkehr aus der Mantelprozedur (`guide`) der Wechsel zur Benutzerebene stattfindet. Sind mangels Hardwarehilfe die Handhaberinkarnationen zu zählen, zeigt ein Zählerwert von 1 diesen Zustand an. In dem Fall ist jedoch vorauszusetzen, dass mittels NMI gestartete und damit kaskadierte Unterbrechungshandhaber nicht durch dieselbe Mantelprozedur gelenkt werden (s.o.: Problem der Nachbildung durch Zählen), sondern nur welche, die einen IRQ zur Ursache haben.
Die Abarbeitung der Auftragswarteschlange (`flush`) geschieht auf der für das Betriebssystem definierten Prioritätsebene, weshalb im Falle des x86 die Unterbrechungssperre aufgehoben wird. Da während der Abarbeitung jederzeit ein weiterer AST ausgelöst werden kann, ist vor Rückkehr aus der Unterbrechungsbehandlung der Auftragsbestand erneut zu prüfen. Hierzu ist aus zuvor bereits erwähntem Grund erneut die Prioritätsebene der CPU einzunehmen, die Unterbrechungssperre also zu setzen.
Gemeinhin ist ein Unterbrechungshandhaber erster Stufe einer Prozedur im programmiersprachlichen Sinn (↑*C function*) sehr ähnlich, allerdings erfolgen Aufruf von und Rückkehr aus dem entsprechenden Unterprogramm auf anderem Wege. Der ↑Aktivierungsblock dieses Unterprogramms unterscheidet sich stark von dem einer normalen Prozedur (keine Argumente, gesicherter Prozessorstatus anstatt gesicherte Rücksprungadresse), weshalb es auch nicht mit einem Prozedur- (`ret`), sondern Ausnahmerücksprungbefehl (`iret`) beendet wird. Da dieser Unterbrechungshandhaber aber durch eine Mantelprozedur gesteuert und ebenfalls als Prozedur aufgerufen wird, besitzt er einen ganz normalen Aktivierungsblock und führt am Ende auch einen Prozedurrücksprung aus:

```
void flih000(train_t *this) {
    static sequel_t slih = {0, {slih000, 42}};          /* AST descriptor */
    /* ... */
    defer(annex(), &slih);                                  /* release AST */
}
```

Die hier wichtigen Punkte sind einerseits die Programmierung eines ↑Deskriptors für einen AST, um den dementsprechenden Unterbrechungshandhaber zweiter Stufe zu erfassen, und anderseits Bereitstellung eben dieses Deskriptors, um den AST beziehungsweise SLIH zu

gegebener Zeit auszulösen. Letzteres geschieht durch Zurückstellung (`defer`) eines Prozeduraufrufs (DPC: `slih`). Ob überhaupt ein SLIH stattfindet und daher ein AST ausgelöst wird, ist Ermessenssache des FLIH. Darüberhinaus kann ein FLIH mehr als einen SLIH nach sich ziehen, indem er mehr als einen AST auslöst beziehungsweise Deskriptor zurückstellt.
Der Unterbrechungshandhaber zweiter Stufe ist ebenfalls eine gewöhnliche Prozedur. Es bietet sich an, diese Prozedur mit einem „generischen Argument“ zu versehen, durch das der FLIH Parameter an den SLIH übergeben kann. Diese Parameter (im gezeigten Beispiel der Wert `42`) sind ebenfalls im Deskriptor für den SLIH aufgeführt und werden ihm automatisch übergeben, wenn der betreffende Deskriptor (durch `flush`) der AST-Warteschlange entnommen wird. Nachfolgend das Skelett eines solchen SLIH:

```
void slih000(data_t data) {
    /* ... */
}
```

Als AST ausgelöst wird diese Prozedur asynchron zum gegenwärtigen Hauptausführungsstrang im Betriebssystem gestartet. Gegebenenfalls sind die von ihr angestoßenen ↑Aktionen daher mit dem sonst im Betriebssystem noch stattfindenden ↑Prozessen zu synchronisieren. Auch wenn mit einem AST letztlich die ↑Sequentialisierung bestimmter Vorgänge einher geht, finden damit längst nicht alle Vorgänge im Betriebssystem sequentialisiert statt.

asynchronous system trap (dt.) ↑asynchrone Systemfalle.

Atlas Großrechner (Univ. Manchester, Ferranti und Plessey, 1962): 48-Bit ↑Maschinenwort, 24-Bit breite ↑Adresse, 16 Kiloworte ↑Kernspeicher (↑RAM), 8 Kiloworte ↑Festwertspeicher, 96 Kiloworte ↑Trommelspeicher, ↑DMA, eigener ↑Registersatz zur Ausführung von ↑Systemfunktionen (sog. *extracode*, z.B. ↑Unterbrechungshandhaber). Ein Universalrechner, mit dem erstmalig ↑einstufiger Speicher verwirklicht wurde.

atomare Aktion (en.) ↑*atomic action*. Eine primitive oder komplexe ↑Aktion, deren Einzelschritte nach außen sichtbar im Verbund scheinbar gleichzeitig stattfinden. Indem die Aktion tatsächlich stattfindet, läuft eine ↑atomare Operation ab.

atomare Operation (en.) ↑*atomic operation*. Operation, für die ↑Unteilbarkeit gilt. Eine solche Operation kann weder durch eine ↑Unterbrechung aufgespalten werden, noch durch ↑Parallelverarbeitung gleichzeitig mehrfach stattfinden.
Für gewöhnlich ist ein ↑atomarer Befehl der Ursprung für eine solche Operation, nämlich kodiert als Einzelanweisung im ↑Maschinenprogramm, um im ↑Abruf- und Ausführungszyklus ungeteilt in der realen Maschine (↑CPU) abzulaufen. Beispiel für derartige Operationen sind ↑CAS, ↑FAA, ↑FAS und ↑TAS.
Besteht die Operation jedoch aus einer Folge von Einzelanweisungen, sorgt eine Sperre am Anfang des diese Anweisungssequenz bildenden ↑Programmabschnitts (↑*critical section*) für die geforderte ↑Atomizität. Normalerweise wird die Sperre beim Verlassen dieses Abschnitts wieder aufgehoben, und zwar nur von dem Prozess, der den Abschnitt vorher betreten hatte. Ruft die Ausführung dieses Abschnitts eine ↑Ausnahmesituation hervor, die zum Abbruch des Prozesses führen muss, gilt der Sperrzustand allgemein als undefiniert. Implizites Aufheben der Sperre beim Prozessabbruch, wenn aufgrund der Art der ↑Prozessinkarnation dafür nämlich das ↑Betriebssystem zuständig ist, kann mangels Kontextwissen gemeinhin nicht geboten sein — viel eher folgt ↑Panik.

atomarer Befehl (en.) ↑*atomic instruction*. Bezeichnung für einen ↑Maschinenbefehl, der eine ↑atomare Aktion des für ihn zutreffenden ↑Prozessors spezifiziert.

atomic action (dt.) ↑atomare Aktion.

atomic basic block (dt.) ↑unteilbarer Grundblock.

atomic instruction (dt.) ↑atomarer Befehl.

atomic operation (dt.) ↑atomare Operation.

atomicity (dt.) ↑Atomizität.

Atomizität (en.) ↑*atomicity*. Eigenschaft einer Sequenz von Einzelanweisungen vor dem Hintergrund von ↑Parallelverarbeitung entweder vollständig oder gar nicht ausgeführt zu werden. Für einen ↑Prozess stellt sich diese Sequenz als ↑Elementaroperation dar. Gelingt ihre Ausführung, ist dem Prozess gewiss, diese exklusiv bestritten zu haben. Tritt während der Ausführung eine Situation ein, die zum Scheitern der Operation führt (↑*exceptional situation*), kommt es zum Ausführungsabbruch, wobei der zum Operationsanfang gültige Zustand hinterlassen wird — so, als wenn die Operation eben noch nicht ausgeführt worden wäre. Falls geboten und technisch möglich, wird dieser Ausnahmefall dem Prozess geeignet mitgeteilt, entweder durch einen Fehlerkode oder der Erhebung (*raising*) einer ↑Ausnahme. Anderenfalls geht der Operationsabbruch mit einem Prozessabbruch einher.

Aufgabe (en.) ↑*task*, T_i. Etwas, was einem ↑Prozess zu tun aufgegeben ist (in Anlehnung an den Duden); der ↑Auftrag, eine bestimmte Funktion zu erfüllen. Dieses Etwas kann implementiert sein als ein ↑Unterprogramm, also keinen eigenständigen ↑Handlungsstrang definieren oder, umgekehrt, sehr wohl durch einen eigenständigen Handlungsstrang realisiert sein, der dann ein oder mehrere Unterprogramme autonom und in einer bestimmten Reihenfolge ausführt. Im Kontext von ↑Echtzeitsystemen eine ausführbare Einheit eines (sequentiellen) ↑Programms, die wenigstens durch eine ↑WCET und einen ↑Termin charakterisiert ist. Eine Reihe verwandter ↑Arbeiten, die gemeinsam eine bestimmte Funktion erbringen. Die einen ↑Prozess bestimmende Einheit einer ↑Ablaufplanung, wobei der Prozess durch mehrere solcher (identischen, gleichen oder verschiedenen) Einheiten definiert sein kann.

Aufrufbetrieb (en.) ↑*polling mode*. Ursprünglich eine bestimmte Verfahrensweise zur ↑Steuerung und Überwachung der Kommunikation zwischen Sende- und Empfangsstationen in einem ↑Rechnernetz. Dabei fordert eine zentrale Leitstation (*primary*) eine entfernte Station zum Senden (*master*) oder Empfangen (*slave*) auf.
Mit seiner englischsprachigen Bedeutung fällt dieser Begriff — dann allerdings mehr im Sinne von „Abfragebetrieb“ — oft auch im Zusammenhang mit einem ↑Prozess, der den Status bestimmter ↑Entitäten der Hardware oder Software eben durch eine *zyklische Abfrage* ermittelt. Ein solcher Prozess betreibt letztlich ↑aktives Warten.

Aufrufkonvention (en.) ↑*calling convention*. Methode, nach der ein ↑tatsächlicher Parameter einem ↑Unterprogramm übergeben wird. Ist das Unterprogramm eine Funktion, legt die Methode zusätzlich die Art und Weise der Rückgabe des Funktionswerts fest. ↑Platzhalter für die Über- beziehungsweise Rückgabewerte sind Prozessorregister oder liegen auf dem ↑Laufzeitstapel, mit fester Zuordnung der Parameterposition zu einem Prozessorregister oder Festlegung der Reihenfolge beim Stapeln der Parameter (↑cdecl: von rechts nach links). Darüber hinaus wird bestimmt, welche Prozessorregister innerhalb des Unterprogramms frei verwendbar sind. Hierzu erfolgt eine Differenzierung zwischen ↑flüchtiges Register und ↑nichtflüchtiges Register, wobei nur die Inhalte letzterer invariante Merkmale bei der Unterprorgammausführung darstellen.

Auftrag (en.) ↑*job*, ↑*task*. Weisung; einem ↑Rechensystem zur Erledigung übertragene ↑Aufgabe (in Anlehnung an den Duden). Die damit verbundene Aufforderung wird von einem ↑Kommandointerpreter, allgemein auch ↑Auftragnehmer, entgegengenommen, der die zu erledigende Aufgabe prüft und, falls gültig beziehungsweise zulässig, alle zur vollständigen Durchführung erforderlichen Handlungen veranlasst.
Einerseits ist eine solche Aufgabe durch ein eigenständiges ↑Programm implementiert, das dann durch einen eigenen ↑Prozess zur Ausführung gebracht wird. Andererseits kann die Aufgabe aber auch als ein ↑Unterprogramm des Kommandointerpreters ausgelegt sein, dass dann im Kontext eines bereits stattfindenden Prozesses (nämlich dem ↑Interpreter) aufgerufen wird und abläuft (↑*built-in command*). Die Aufgabe kann darin bestehen, ↑Betriebsmittel anzufordern, damit ein vorgesehenes Programm ausgeführt werden, dieses Programm zu

starten und mit bestimmten Geräten der ↑Peripherie zu verknüpfen, Zwischenergebnisse für ein nachgeschaltet auszuführendes Programm zu speichern, schließlich gestartete Programme zu beenden und angeforderte Betriebsmittel wieder freizugeben. Eine derart komplexe Aufgabe ist für gewöhnlich als Anweisungsfolge (↑*batch job*) beschrieben und wird durch ↑Stapelverarbeitung ausgeführt.

Auftraggeber (en.) ↑*client*. Bezeichnung für einen ↑Prozess, der einem ↑Auftragnehmer eine ↑Aufgabe zur Bearbeitung zukommen lässt. Je nach Art der Aufgabe oder auch technischer Umsetzung des Auftragnehmers, muss der beauftragende Prozess die Beendigung der Aufgabenbearbeitung abwarten oder kann unabhängig von der Aufgabenbearbeitung nach Abgabe der Aufgabe weiter voranschreiten. Verzögert wird der Prozess aber wenigstens für die Zeitspanne, die die Abgabe der Aufgabe benötigt. Diese Zeitspanne muss jedoch nicht zwingend auch die Annahme der Aufgabe durch den Auftragnehmer selbst umfassen. Stattdessen kann die Verzögerung für den Prozess bereits enden, wenn das ↑Betriebssystem die Aufgabe für den Auftragnehmer verbucht und vermerkt hat.

Auftragnehmer (en.) ↑*server*. Bezeichnung für einen ↑Prozess, der eine von einem ↑Auftraggeber abgesetzte ↑Aufgabe zur Bearbeitung übernimmt. Dabei muss der Prozess selbst nicht zwingend auch für die Aufgabenbearbeitung zuständig sein, er kann lediglich als ↑Zusteller agieren und sich schnellstmöglich wieder der Übernahme einer weiteren Aufgabe zuwenden.

Auftragssteuersprache (en.) ↑*job control language*. Siehe ↑Kommandosprache.

Ausführungszeit (en.) ↑*execution time*. Bezeichnung für eine ↑Prozessgröße, die die effektive Zeitspanne für die Durchführung einer ↑Aufgabe quantifiziert. Grundsätzlich wird zwischen der minimalen (↑BCET) und maximalen (↑WCET) Zeitspanne unterschieden. Darüber hinaus kann die Zeitspanne (ggf. auch zwischen diesen beiden Extremen, soweit diese bekannt sind) auch einem Durchschnittswert (↑ACET) entsprechen.
↑Bezugspunkt in allen Fällen ist die Anzahl von Taktzyklen, die bei der Aufgabendurchführung von der ↑CPU benötigt werden. Diese Anzahl lässt sich durch Laufzeitmessung oder Programmanalyse bestimmen: sie ist einerseits durch den kürzesten und längsten Pfad des diese Aufgabe entsprechenden ↑Programms gegeben und anderseits durch die Anzahl von ↑Unterbrechungen, den der betreffende ↑Prozess unterworfen sein kann. Muss die Aufgabe in ↑Echtzeit durchgeführt werden, ist, je nach ↑Echtzeitbedingung, für jeden dabei anfallenden ↑Befehlszyklus ↑Vorwissen zur maximalen Anzahl der zu erwartenden ↑Prozessortakte wünschenswert oder unabdinglich. Verrechnet mit der ↑Taktfrequenz der CPU ergibt sich daraus die für die Ausführung der Aufgabe zu erwartende absolute Zeit.

Auslastung (en.) ↑*utilisation*. Grad der anhaltenden Nutzung von einem ↑Betriebsmittel; der Bruchteil der Zeit, den dieses Betriebsmittel nicht brachliegt. In Summe auch der Grad der anhaltenden Nutzung eines ↑Rechensystems (s. a. ↑Prozessorauslastung).

Auslösezeit (en.) ↑*release time*, r_i. Bezeichnung für eine ↑Prozessgröße, die den Zeitpunkt festlegt, zu dem die Bearbeitung einer ↑Aufgabe frühestens ansteht.
Eine Aufgabe wird durch ein ↑Ereignis hervorgerufen und ist dann zur Verarbeitung durch einen ↑Prozess verfügbar. Die betreffende Aufgabe ist damit im System vorhanden, sie steht zur weiteren Bearbeitung bereit. So wird der Zeitpunkt der Auslösung einer Aufgabe (im deutschsprachigen Raum) auch gelegentlich als ↑Bereitzeit bezeichnet, also als einen Zeitpunkt, zu dem die Durchführung durch einen Prozess frühestens beginnen kann — allerdings steht die dazu erforderliche ↑Einplanung (des Prozesses bzw. der Aufgabe) noch aus.
In technischer Hinsicht sind jedoch zwei verschiedene Momente auseinanderzuhalten. Einerseits der Umstand, eine Aufgabe aus einem Ereignis heraus zu formulieren und für die weitere Bearbeitung durch einen Prozess geeignet aufzubereiten. Beispielsweise ist dies die Funktion einer ↑Unterbrechungsbehandlung, nachdem sie durch eine ↑Unterbrechungsanforderung (Ereignis) ausgelöst wurde, um die damit verknüpfte Aufgabe entweder sofort zu erledigen oder für die spätere Bearbeitung aufzubereiten und zurückzustellen. Ein anderes Beispiel wäre die

Zustellung einer ↑Nachricht (auslösendes Ereignis) an einen Empfangsprozess, durch einen Sendeprozess: die Bildung der Nachricht (↑*marshalling*) ist Aufgabe des Sendeprozesses, der damit aber auch eine vom Empfangsprozess zu leistende Aufgabe vorbereitet. Andererseits eben der Begleitumstand, die auf-/vorbereitete Aufgabe schließlich der weiteren Bearbeitung zuzuführen. Beispielsweise kann dies bedeuten, einen Prozess, der auf die Bearbeitung einer solchen Aufgabe wartet, vom ↑Prozesszustand blockiert nach bereit zu überführen. Genau genommen bezieht sich die Bereitzeit nur auf diesen letzten Schritt der Zustandstransformation eines Prozesses in einer ↑Verarbeitungskette dieser Aufgaben, nämlich den betreffenden Prozess auf die ↑Bereitliste gesetzt zu haben.
In den Fällen, wo eine Differenzierung zwischen den beiden Ereignissen, Auslösung einerseits und Bereitstellung andererseits, geboten oder nicht vernachlässigbar ist, bezeichnet $r_i^0 = r_i$ den Zeitpunkt der Auslösung und r_i^+ den Zeitpunkt der Bereitstellung, wobei gilt: $r_i^0 < r_i^+$. In diesen Zusammenhang ist auch die ↑Ankunftszeit einzuordnen.

Ausnahme (en.) ↑*exception*. Sonderfall, eine Abweichung vom normalen ↑Programmablauf. Der ↑Prozess wird unterbrochen und gegebenenfalls nach erfolgter ↑Ausnahmebehandlung wieder aufgenommen. Als ↑synchrone Ausnahme wird der Sonderfall von dem betreffenden Prozess selbst hervorgerufen, als ↑asynchrone Ausnahme ist die Ursache des Sonderfalls in einem anderen (ggf, externen) Prozess begründet.

Ausnahmebedingung (en.) ↑*exception condition*. Gegebenheit in einer ↑Ausnahmesituation, die eine ↑Aktion dazu bringt, eine ↑Ausnahme zu erheben. Im Falle der in einem ↑Maschinenprogramm beschriebenen Aktion beispielsweise ein ungültiger ↑Maschinenbefehl beziehungsweise ↑Systemaufruf, eine ↑Schutzverletzung oder ein ↑Seitenfehler beim ↑Abruf- und Ausführungszyklus.

Ausnahmebehandlung (en.) ↑*exception handling*. Vorgang zur Bearbeitung eines besonderen (eine ↑Ausnahme darstellenden) Falls bei der Programmausführung. Ursache ist eine ↑Ausnahmesituation in der (realen/virtuellen) Maschine, auf der ein ↑Programm abläuft: die in diesem Programm direkt/indirekt kodierte ↑Aktion oder ↑Aktionsfolge, beispielsweise ein Befehl der ↑CPU oder eine Betriebssystemfunktion ausgelöst durch einen ↑Systemaufruf, kann nicht normal vollendet werden. Für die Bearbeitung eines solchen Sonderfalls ist ein Programm (↑Ausnahmehandhaber) vorzusehen, das auf der Maschine, deren Aktion/Aktionsfolge die Ausnahme erhebt, zur Ausführung kommt. Typisches Beispiel ist eine Routine zur ↑Unterbrechungsbehandlung in oder zur Signalbehandlung auf einem Betriebssystem, also ein Programm für eine reale beziehungsweise ↑virtuelle Maschine.

Ausnahmehandhaber (en.) ↑*exception handler*. Bezeichnung für einen speziellen, ausschließlich zur ↑Ausnahmebehandlung ausgelegten und vorgesehenen ↑Handhaber.

Ausnahmesituation (en.) ↑*exceptional situation*. Bezeichnung für die eine ↑Ausnahmebedingung bestimmenden Verhältnisse, Umstände, in denen sich ein ↑Prozess augenblicklich befindet (in Anlehnung an den Duden).

Ausnahmezuteiler (en.) ↑*exception dispatcher*. Bezeichnung für eine spezielle ↑Mantelprozedur, die durch eine ↑Ausnahme aktiviert wird, daraufhin den zuständigen ↑Ausnahmehandhaber identifiziert und aufruft und gegebenenfalls, nämlich in Abhängigkeit von der Art der Ausnahme, für die Fortsetzung des ausnahmsweise unterbrochenen ↑Programmablaufs sorgt. Für gewöhnlich übernimmt diese Prozedur für eine Gruppe solcher Handhaber zentrale Verwaltungsaufgaben, beispielsweise die ↑Zustandssicherung und -wiederherstellung für den unterbrochenen ↑Prozess, die Protokollierung seiner ↑Benutzerzeit und ↑Systemzeit, sonstige ↑Buchführung oder gar die ↑Sequentialisierung von ↑Unterbrechungsbehandlungen zweiter Stufe (↑SLIH), wenn diese beispielsweise jeweils durch einen ↑AST ausgelöst worden sind. Typische Art einer solchen Mantelprozedur ist der ↑Systemaufrufzuteiler, der neben oben genannten Aspekten insbesondere für die Parameterübergabe von der ↑Benutzerebene zur

↑Systemebene sorgt und, in umgekehrter Richtung, den Rückgabewert beziehungsweise einen Fehlermerker transferiert.

Ausrichtung (en.) ↑*alignment*. Linienführung einer ↑Adresse entsprechend der Größe von einem an einer bestimmten ↑Speicherstelle liegenden ↑Exemplar von ↑Text oder ↑Daten. Die Führung kann durch die ↑CPU vorgegeben oder empfohlen sein, da sonst der Zugriff scheitert (↑*trap*) oder langsamer verläuft (Zwischenzyklus). Maßgeblich ist die ↑Informationsstruktur (insb. ↑Wortbreite) der CPU. Eine ↑Assembliersprache bietet daher (für gewöhnlich) Anweisungen, um eine bestimmte Anordnung von Text oder Daten festlegen zu können (↑Pseudobefehl, z.B. `.p2align`). Typischerweise erzeugt ein (optimierender) ↑Kompilierer solche Anweisungen und sorgt somit dafür, dass beispielsweise Zugriffe auf Datentypexemplare immer mit einer zur Exemplargröße ausgerichteten Adresse geschehen: `char` gerade/ungerade, 8-Bit; `short` gerade, 16-Bit; `int`/`long` gerade, 32-Bit; `long long` gerade, 64-Bit.

Austrittsprotokoll (en.) ↑*exit protocol*. Konvention zur ↑Synchronisierung ↑gleichzeitiger Prozesse, die ein ↑nichtsequentielles Programm durchziehen. Das Protokoll regelt den weiteren ↑Programmablauf, nachdem ein ↑kritischer Abschnitt von einem ↑Prozess verlassen wurde. Mit erfolgreichem Durchlauf des Protokolls hat der Prozess den kritischen Abschnitt „freigegeben“ (*release*).
Einen kritischen Abschnitt zu verlassen ist vergleichsweise einfach, wenn das ↑Eintrittsprotokoll für die ↑Sequentialisierung gleichzeitiger Prozesse ↑aktives Warten vorsieht. In dem Fall ist lediglich der kritische Abschnitt zu entriegeln (*unlock*). Anderenfalls, und zwar wenn ↑passives Warten betrieben wird, ist ein möglicherweise wartender Prozess durch zusätzliche Anweisung an den ↑Planer zu deblockieren. Welcher Prozess zu deblockieren ist bestimmt der Planer aber nur dann, wenn das Eintrittsprotokoll keine ↑Warteschlange von Prozessen aufgebaut hat. Wird beim erfolglosen Eintrittsversuch der betreffende Prozess einer solchen Warteschlange hinzugefügt, muss die Auswahlentscheidung das Protokoll beim Freiwerden des kritischen Abschnitts selbst treffen. Um der Gefahr von ↑Prioritätsverletzung vorzubeugen, muss diese Entscheidung in Einklang mit der ↑Prozesseinplanung stehen. Alles weitere in beiden Fällen (aktives/passives Warten) regelt das Eintrittsprotokoll.
Im Gegensatz zum zugehörigen Eintrittsprotokoll, das für denselben kritischen Abschnitt in ↑Konkurrenzsituation beim Eintrittsversuch mehrerer Prozesse verlaufen kann, bestreitet das Protokoll für den Austritt zu einem Zeitpunk normalerweise nur ein einziger Prozess. Gleichwohl kann dieser Prozess im ↑Wettstreit stehen mit Prozessen, die sich zum Zeitpunkt der Freigabe des kritischen Abschnitts im Eintrittsprotokoll befinden. So bilden in Bezug auf die Absicherung nur eines kritischen Abschnitts die Implementierungen beider Protokolle selbst ein mit bis zu $N + 1$ Prozessen gleichzeitig ausgeführtes nichtsequentielles Programm, N Prozesse eintrittsseitig und 1 Prozess austrittsseitig. Entsprechend sind darin Vorkehrungen zur ↑Koordinierung der ↑Aktionen dieser Prozesse zu treffen.

automated computer operation (dt.) ↑automatisierter Rechnerbetrieb.

automatisierter Rechnerbetrieb (en.) ↑*automated computer operation*. ↑Betriebsart, bei der das ↑Rechensystem gesteuert durch einen ↑Kommandointerpreter nacheinander mit Arbeitspaketen bestückt wird, wobei jedes einzelne Paket durch ein ↑Programm beschrieben ist, von dem zu einem Zeitpunkt stets nur eins zur Ausführung bereit im ↑Hauptspeicher liegt (↑*uniprogramming*). Das Rechensystem führt automatisch und ohne Eingriffe einer Bedienperson (↑*operator*) alle erforderlichen Schritte aus, um die zur Ausführung bestimmten Programme nacheinander einzulesen, zu übersetzen, zu laden und auszuführen. Die Bedienperson sorgt lediglich dafür, einen ganzen ↑Stapel von Rechenaufgaben (↑*batch job*), einen ↑Auftrag, am Eingang in Empfang zu nehmen, in das Rechensystem einzuspeisen, das in aller Regel auf ↑Tabellierpapier ausgedruckte Ergebnis der durchgeführten Berechnungen dem Drucker zu entnehmen und in den Ausgang zur Abholung zu legen.
Bedeutet ↑manueller Rechnerbetrieb noch, jeden einzelnen (oben genannten) Schritt durch Bedienpersonal auszulösen, ist der Mensch nunmehr weitestgehend aus der Verarbeitungskette herausgenommen. Der dadurch wesentlich gesteigerte ↑Durchsatz erfordert allerdings

einen für ein Systemprogramm (↑*resident monitor*) reservierten Platz im Hauptspeicher, um solche Programmstapel verarbeiten zu können. Da zudem nicht jeder Auftrag dieselbe Reihenfolgebildung von Programmen impliziert, ist diese für den jeweiligen ↑Stapelauftrag immer wieder explizit durch Anweisungen einer ↑Auftragssteuersprache (↑JCL) festzulegen. Darüber hinaus muss für eine in Bezug auf nachfolgende Programme im Stapel vernünftige Fehlerbehandlung gesorgt sein, sollte nämlich ein ↑Programmablauf scheitern. Alle dafür notwendigen Softwarevorkehrungen, in Ergänzung zum Kommandointerpreter und zu grundlegenden Ein-/Ausgabeprozeduren, resultieren in Programme, die in ihrer Gesamtheit ein „embryonales ↑Betriebssystem“ bilden (↑FMS).

average access time (dt.) ↑mittlere Zugriffszeit.

B 5000 Großrechner (1961, Burroughs Corp.), auch „*The Descriptor*“ genannt: 48-Bit ↑Wortbreite, wobei der ↑Hauptspeicher indirekt indiziert über ein 1024 ↑Worte umfassendes, verschiebbares statisches ↑Feld (*program reference table*, PRT) adressiert wurde. Dabei war ein Feldeintrag ein ↑Deskriptor, der entweder direkt einen Operandenwert enthielt oder auf ein ↑Segment im Haupt- oder ↑Hintergrundspeicher (↑*virtual memory*) verwies.
Ein auf einer ↑Stackarchitektur basierender und speziell für höhere Programmiersprachen (↑Algol 60) entwickelter ↑Rechner, der erstmalig ↑Segmentierung implementierte. Zudem bot er Unterstützung für echte ↑Parallelverarbeitung (↑*asymmetric multiprocessing*) durch mehrere Recheneinheiten. Das ↑Rechensystem wurde durch ↑MCP betrieben, es war wegbereitend für viele nachfolgende Rechnerentwicklungen.

background noise (dt.) ↑Hintergrundrauschen.

backoff (dt.) zurückweichen. Verfahren zur ↑Stauauflösung.

bad block (dt.) fehlerhafter, ↑defekter Block.

bad-block management (dt.) ↑Defektblockverwaltung.

Bandbreite (en.) ↑*bandwidth*. Breite eines Frequenzbandes beschränkt durch eine untere und obere Grenzfrequenz. Physikalische Kenngröße eines Übertragungskanals, die effektiv die Anzahl von ↑Daten festlegt, die in einer bestimmten Zeiteinheit durch den Kanal übertragen werden können.

bandwidth (dt.) ↑Bandbreite.

base/limit register (dt.) ↑Segmentregister.

basic block (dt.) Basisblock, ↑Grundblock.

batch (dt.) ↑Stapel.

batch job (dt.) ↑Stapelauftrag.

batch mode (dt.) ↑Stapelbetrieb.

batch monitor (dt.) ↑Stapelmonitor.

batch process (dt.) ↑Stapelprozess.

batch processing (dt.) ↑Stapelverarbeitung.

Bauwesen Gesamtheit dessen, was mit dem Errichten von Bauten zusammenhängt (Duden). Bei dem hier gemeinten Bau handelt es sich um das ↑Betriebssystem.

BCET Abkürzung für (en.) *best-case* ↑*execution time*, kleinste anzunehmende ↑Ausführungszeit.

BDOS Abkürzung für (en.) *Basic Disk Operating System*. Bezeichnung für das im ↑Festwertspeicher liegende ↑CP/M-Subsystem zur Verwaltung dauerhaft in einem ↑Dateisystem gespeicherter ↑Daten und der Durchführung wie auch Umlenkung von ↑Ein-/Ausgabe mittels ↑Dateien. Dabei stellt das ↑BIOS die grundlegenden Ein-/Ausgabeoperationen zur Verfügung.

Bearbeitungszeit (en.) ↑*processing time*. Bezeichnung für eine ↑Prozessgröße, die die Zeitspanne bis zur vollständigen Bearbeitung einer ↑Aufgabe quantifiziert. Häufig auch gleichgesetzt mit der ↑Durchlaufzeit von dem ↑Prozess, der diese Aufgabe entweder allein oder mit mehreren anderen (kooperierenden) Prozessen zusammen bearbeitet. Genau genommen ist diese Zeitspanne allerdings nur die Summe aller bei der Aufgabenbearbeitung durch die direkt beteiligten Prozesse anfallenden ↑Bedienzeiten: in der Durchlaufzeit eines Prozesses sind nämlich auch all seine ↑Wartezeiten enthalten. In logischer Hinsicht entspricht die kumulierte Zeit einem (relativen) Zeitraum, dessen Anfang und Ende auf einer (relativen) Zeitskala durch Zeitpunkte scheinbar derart festlegt ist, als wenn der Prozess bis zur Fertigstellung der Aufgabe durchläuft (↑*run to completion*).

Bedieneinheit (en.) ↑*service unit*. Bezeichnung einer Verarbeitungseinheit (↑Prozessor), die in der Lage ist, einen an sie gestellten ↑Auftrag zu verarbeiten. Für gewöhnlich sind die Aufträge pro Einheit gleichartig, sie bestimmen sich durch die funktionalen Merkmale der mit der Station (in Hard-/Software) assoziierten ↑Entität. So ist beispielsweise ein ↑Prozess ein Auftrag an die ↑CPU als Entität, ein ↑Programm zu verarbeiten: er befindet sich dazu im ↑Prozesszustand bereit. Demgegenüber ist ein blockierter Prozess als solch eine Entität mit einem Auftrag verknüpft, der ihn von einem anderen (internen/externen) Prozess zur Verarbeitung erst noch zugestellt werden muss: etwa ein Auftrag, der die Beendigung einer Ein-/Ausgabeoperation (↑Ein-/Ausgabestoß) anzeigt, mit dem ein ↑kritischer Abschnitt wieder zugänglich gemacht wird oder der ein ↑konsumierbares Betriebsmittel zu Verfügung stellt.
Sind zu einem Zeitpunkt mehr Aufträge anhängig als Einheiten zur Verfügung stehen, bildet sich eine ↑Warteschlange von Aufträgen, die nicht sofort der Verarbeitung zugeführt werden können. Nur in dieser Situation ist die ↑Einplanung von Aufträgen erforderlich, die damit über die Reihenfolge der ↑Einlastung einer jeweiligen Verarbeitungseinheit entscheidet.

Bedienpult (en.) ↑*operator panel*. Ursprünglicher Arbeitsplatz (↑Systemkonsole) des Betreuungspersonals (↑*operator*) von einem ↑Rechensystem.

Bedienstation (en.) ↑*operating station*. Eine Station, an der ein ↑Ankunftsprozess mit einer bestimmten Ankunftsrate und einer gegebenen Verteilung einen ↑Auftrag nach dem anderen erzeugt. An dieser Station befindet sich wenigstens eine ↑Bedieneinheit, die zu einem Zeitpunkt nur einen Auftrag verarbeiten kann. Stehen weniger Bedieneinheiten zur Verfügung als Aufträge zur Bearbeitung anstehen, bildet sich eine ↑Warteschlange, die nach und nach entsprechend einer bestimmten Strategie abzubauen ist.

Bedienzeit (en.) ↑*service time*. Bezeichnung für eine ↑Prozessgröße, die die Zeitspanne quantifiziert, in der eine ↑Aufgabe an einer ↑Bedieneinheit durchgeführt wird. Handelt es sich bei der Aufgabe um einen ↑Prozess und bei der Bedieneinheit um die ↑CPU, dann entspricht diese Zeitspanne dem ↑Rechenstoß des Prozesses bis zum nächsten ↑Prozesswechsel.

bedingte kritische Region (en.) ↑*conditional critical region*. Eine ↑kritische Region, in der die ↑gemeinsame Variable mit einer ↑Ereigniswarteschlange verknüpft ist. Jede durch einen ↑Prozess erfolgende Wertezuweisung an die gemeinsame Variable verursacht (*cause*) ein ↑Ereignis und veranlasst in dem Moment (↑*release time*) die Freistellung von Prozessen, die dieses Ereignis erwarten (*await*). Wie viele Prozesse beziehungsweise welcher Prozess aus dieser Menge freigestellt wird, bestimmt die ↑Signalisierungsdisziplin.
Der ursprünglichen Idee (Hansen, 1972) nach werden alle auf das Ereignis wartenden Prozesse freigestellt. Die dadurch vermiedene ↑Interferenz mit der ↑Prozesseinplanung in Bezug auf die Auswahl des Prozesses, der als nächster den ↑Prozessor erhalten soll, zieht allerdings

unnötig ↑Einlastungen mit ↑Prozessinkarnationen nach sich, die nach erneuter Ereignisüberprüfung gleich wieder in den Wartezustand verfallen und den Prozessor abgeben werden: von n das Ereignis erwartenden Prozessen darf nur einer nach Ereigniseintritt die kritische Region passieren; von n freigestellten Prozessen werden $n-1$ sofort wieder blockieren, nachdem das eine Ereignis von genau nur einem Prozess wahrgenommen (d.h., konsumiert) wurde.
Durch bedingte Anweisungen, die Ereignisse oder damit verknüpfte Zustände beziehungsweise Variablenwerte abfragen, lässt sich der weitere nichtsequentielle ↑Programmablauf explizit zwischen konkurrierenden Prozessen steuern. So kommt ein solches Ereignis im übertragenen Sinne auch einer ↑Bedingungsvariablen gleich und die sie umfassende Region (↑*atomic basic block*) trägt das Synchronisationsmerkmal (↑*mutual exclusion*) eines ↑Monitors — die hier betrachteten und im Zusammenhang mit (unbedingten) kritischen Regionen stehenden Techniken legen die Grundlage für Monitore.

Exkurs Zur Verdeutlichung der Formulierung und Funktionsweise von kritischen Regionen, in denen Prozesse bedingt warten können, dient ein klassisches Problem der ↑Betriebsmittelkontrolle (↑*readers-writers problem*) als Beispiel. Zwei Arten ↑gleichzeitiger Prozesse, bezeichnet als Leser (*reader*) und Schreiber (*writer*), teilen sich ein ↑Betriebsmittel (etwa einen ↑Puffer). Die Leser können das Betriebsmittel gleichzeitig verwenden, aber die Schreiber benötigen den exklusiven Zugriff darauf. Darüberhinaus soll ein Schreiber, der bereit zur Nutzung des Betriebsmittels ist, nicht unnötig lang verzögert werden. Im Folgenden geschehen die Zugriffe auf das fragliche Betriebsmittel im Rahmen der eigentlichen Operationen zum Lesen (`read`) und Schreiben (`write`).
Das Betriebsmittel ist durch einen Kontrollblock repräsentiert, mit dem Buch über die Anzahl der gegenwärtigen Leser und Schreiber geführt werden kann. Ein diesbezüglicher ↑Datentyp sei wie folgt (in der fiktiven Sprache *Concurrent* ↑C) skizziert:

```
typedef struct {                                  /* resource control block */
    shared struct {event int readers, writers; } ctrl;     /* enumerators */
    shared struct { … } data;                      /* subject of read/write */
    …                                                     /* any other data */
} rcb_t;
```

Ein Typqualifizierer verleiht einzelnen Strukturkomponenten der ↑Exemplare dieses Datentyps den Charakter von gemeinsamen Variablen (***shared***), die gegebenenfalls vor gleichzeitigen Zugriffen zu schützen sind, beziehungsweise von Ereignissen (***event***), die der ↑Kooperation der gleichzeitigen Prozesse dienen. Für den Leser ergibt sich daraufhin folgendes Bild:

```
void reader (rcb_t *rp) {                          /* reader side operation */
    region (rp->ctrl) {                        /* sequential reader control */
        await (rp->ctrl.writers == 0);        /* conditionally delay reader */
        elide rp->ctrl.readers++;    /* subscribe reader, stop next writer */
    }
    read(&rp->data);                               /* allow concurrent read */
    region (rp->ctrl) rp->ctrl.readers--;    /* unsubscribe, signal writer */
}
```

Das Beispiel zeigt zwei kritische Regionen, wobei die obere bedingt und die untere unbedingt von Prozessen passiert wird. Die Bedingungsanweisung (***await***) sorgt dafür, dass Leser warten müssen, sobald sich ein Schreiber für Arbeiten in der (durch `rp->ctrl`) ausgewiesenen kritischen Region angemeldet hat. Zur Kontrolle der Schreiber melden sich Leser vor ihrer Leseoperation (`read`) an und danach wieder ab. Die Zähloperationen sind kritisch, allerdings würde die damit einhergehende Wertezuweisung an die gemeinsame Variable (`readers`) im oberen Fall zur ineffektiven Wiederaufnahme eines möglicherweise wartenden Schreibers führen: eine zwar unkritische, aber dennoch ↑falsche Signalisierung wäre die Folge. Um dies zu verhindern, ist die betreffende Operation mit einer Auslassungsanweisung (***elide***) versehen. Diese Anweisung ist letztlich ein Hinweis an den ↑Kompilierer für eine mögliche Optimierungsmaßnahme, um nämlich einen wartenden Prozess davon abzuhalten, nutzlos seine

Wartebedingung abermalig überprüfen und sich wieder in Wartestellung begeben zu müssen. Nach der Leseoperation (`read`), die gegebenenfalls von mehreren Prozessen zugleich durchgeführt werden kann, meldet sich der Leser wieder ab. Diesmal bewirkt die Veränderung des Inhalts der gemeinsamen Variablen (`readers`) jedoch automatisch eine Signalisierung, und zwar des Prozesses, der die Wertezuweisung/Zustandsänderung erwartet.
Für einen Schreiber zeichnet sich ein etwas anderes Bild, wie die nachfolgend präsentierte Programmskizze darlegt:

```
void writer (rcb_t *rp) {                          /* writer side operation */
    region (rp->ctrl) {                         /* sequential writer control */
        elide rp->ctrl.writers++;    /* subscribe writer, stop next reader */
        await (rp->ctrl.readers == 0);       /* conditionally delay writer */
    }
    region (rp->data) write(&rp->data);          /* force sequential write */
    region (rp->ctrl) rp->ctrl.writers--;    /* unsubscribe, signal reader */
}
```

Hier sind drei kritische Regionen ausgewiesen, wobei die obere und untere zu denen der Leser korrespondiert und damit die ↑Kooperation zwischen den betreffenden gleichzeitigen Prozessen regelt. Zu beachten ist die Anmeldung eines Schreibers, bevor er die Bedingungsanweisung (*`await`*) ausführt. Diese Reihenfolge stellt sicher, dass ihm nachfolgende Leser warten müssen, bis er seine Tätigkeit in der (durch `rp->ctrl`) ausgewiesenen kritischen Region beendet hat. Die mittlere kritische Region sorgt für ausschließlich sequentielle Schreiboperationen (`write`).
Ein geschickter Kompilierer kann auf Grundlage solcher Sprachkonstrukte automatisch für die benötigte ↑Synchronisierung der beteiligten (gleichzeitigen) Prozesse sorgen. Allerdings gestaltet sich dieser Vorgang bedeutend schwieriger als bei einer einfachen (unbedingten) kritischen Region. Zunächst sind die Typqualifizierer (*`shared`*, *`event`*) aufzulösen, was folgende abgeänderte Datenstruktur für den Betriebsmittelkontrollblock ergibt:

```
typedef struct {                                   /* resource control block */
    struct {                              /* subject of reader/writer control */
        struct { int value; event_t event; } readers;  /* signalling enum. */
        struct { int value; event_t event; } writers;  /* signalling enum. */
        semaphore_t mutex;
    } ctrl;
    struct { …                                      /* subject of read/write */
        semaphore_t mutex;
    } data;
    …                                                        /* any other data */
} rcb_t;
```

Die beiden Zähler (`readers`, `writers`) wurden hierbei jeweils mit einer Bedingungsvariablen (`event`) assoziiert, die die Ereigniswarteschlange repräsentiert. Das Synchronisationsprotokoll von Lesern und Schreibern wird in wechselseitigem Ausschluss der beteiligten gleichzeitigen Prozesse betrieben, wozu ein ↑Semaphor (`mutex`) dient. Gleiches gilt zur Absicherung der eigentlichen Lese-/Schreiboperationen (`data`).
Auf Basis dieser erweiterten Datenstruktur ergibt sich für Leser die folgende Lösung, bei der zur Koordinierung ausschließlich Semaphore und Bedingungsvariablen Verwendung finden:

```
void __reader (__rcb_t *rp) {
    P(&rp->ctrl.mutex);                          /* enter critical section */
    while (!(rp->ctrl.writers.value == 0))          /* any writer pending? */
        await(&rp->ctrl.writers.event, &rp->ctrl.mutex);    /* yes, block */
    rp->ctrl.readers.value++;       /* subscribe reader, stop next writer */
    V(&rp->ctrl.mutex);                          /* leave critical section */

    read(&rp->data);                              /* allow concurrent read */

    P(&rp->ctrl.mutex);                          /* enter critical section */
    rp->ctrl.readers.value--;                        /* unsubscribe reader */
    cause(&rp->ctrl.readers.event);               /* signal writer, if any */
    V(&rp->ctrl.mutex);                          /* leave critical section */
}
```

Entsprechend der beiden (bedingten, unbedingten) kritischen Regionen der ursprünglichen Formulierung sind hier zwei kritische Abschnitte durch Paare von `P` und `V` ausgewiesen. Da sowohl Leser als auch Schreiber gleichzeitig auf die gemeinsamen Zählervariablen zugreifen und dabei deren Inhalte verändern können, müssen beide kritischen Abschnite durch dieselbe ↑Synchronisationsvariable (`ctrl.mutex`) abgesichert sein.
Der obere kritische Abschnitt steuert die Zulassung der Leser zu ihrer jeweiligen Leseoperation (`read`). Hier prüft ein solcher Prozess auf Konfliktfreiheit mit Schreibern, wartet gegebenenfalls, wenn wenigstens ein Schreiber angemeldet ist für eine Schreiboperation (`write`) und meldet sich selbst für die Leseoperation an. Jeder Leser wartet (`await`) außerhalb des kritischen Abschnitts! Der untere kritische Abschnitt meldet einen Leser ab und löst die Fortsetzung eines auf diese Abmeldung wartenden Schreibers aus (`cause`).
Die Schreiberseite ist nahezu symmetrisch ausgelegt. Unterschiede ergeben sich lediglich in Bezug auf die Zulassung von Schreibern und die Durchführung der Schreiboperation. Auch hier wird zur Koordination nur von Semaphor und Bedingungsvariable Gebrauch gemacht:

```
void __writer (__rcb_t *rp) {
    P(&rp->ctrl.mutex);                          /* enter critical section */
    rp->ctrl.writers.value++;       /* subscribe writer, stop next reader */
    while (!(rp->ctrl.readers.value == 0))          /* any reader pending? */
        await(&rp->ctrl.readers.event, &rp->ctrl.mutex);    /* yes, block */
    V(&rp->ctrl.mutex);                          /* leave critical section */

    P(&rp->data.mutex);                          /* enter critical section */
    write(&rp->data);                                 /* sequential write */
    V(&rp->data.mutex);                          /* leave critical section */

    P(&rp->ctrl.mutex);                          /* enter critical section */
    rp->ctrl.writers.value--;                        /* unsubscribe writer */
    cause(&rp->ctrl.writers.event);               /* signal reader, if any */
    V(&rp->ctrl.mutex);                          /* leave critical section */
}
```

Im Unterschied zur Leserseite meldet sich ein Schreiber erst für seine Schreiboperation (`write`) an, bevor er auf Konfliktfreiheit mit Lesern prüft und gegebenenfalls wartet. Dies führt dazu, dass nach Anmeldung eines Schreibers nachfolgende Leser solange warten müssen, bis dieser Schreiber sich nach erfolgter Schreiboperation wieder abgemeldet hat: Schreiber sind gegenüber Leser priorisiert! Der andere Unterschied besteht in der strikt sequentiellen Durchführung der Schreiboperation, die dazu einen eigenen kritischen Abschnitt bildet, geschützt durch einen eigenen binären Semaphor (`data.mutex`).
Leser wie auch Schreiber, die warten müssen, dürfen dies nicht innerhalb des kritischen Abschnitts (`ctrl.mutex`) tun. Wäre der kritische Abschnitt während einer solchen Wartephase von dem betreffenden Prozess noch besetzt, könnte sein Gegenpart nicht eintreten, um das erwartete Ereignis zu verursachen. Die Folge wäre die ↑Verklemmung der beteiligten Prozes-

se. Um dies zu vermeiden verwenden Leser und Schreiber jeweils eine Bedingungsvariable. Die mit solch einer Variablen verbundene Warteoperation (`await`) lässt den Prozess aus den kritischen Abschnitt (`ctrl.mutex`) austreten, bevor er blockiert und damit den Prozessor abgibt, und anschließend wieder eintreten, nachdem er den Prozessor erneut erhalten hat und seinen Verlauf fortsetzt.
Nach Beendigung einer Wartephase überprüft der betreffende Prozess, egal ob Leser oder Schreiber, erneut seine Wartebedingung. Beispielsweise könnte ein wartender Schreiber mehr als einen Leser vor sich haben. Jeder Leser signalisiert am Ende den Schreiber, aber erst wenn alle Leser durch sind, darf der Schreiber zur Schreiboperation (`write`) zugelassen werden. Für einen wartenden Leser ergibt sich ein ähnliches Bild. Aus diesem Grunde, aber auch zugunsten einer wesentlich einfacheren Programmstruktur und eines weniger komplexen Musters zur Generierung kritischer Abschnitte aus kritischen Regionen, ist die Warteoperation (`await`) in einer kopfgesteuerten Schleife (`while`) eingebettet.
Dieses Muster, nämlich die erneute Überprüfung der Wartebedingung eines Prozesses, obwohl diese für seine Wiederaufnahme (↑*resume*) zuvor anscheinend doch aufgehoben worden sein musste, ist typisch für die hier betrachtete Art des Zusammenspiels gekoppelter Prozesse. So können die Wartebedingungen für verschiedene Prozesse zwar auf dieselbe gemeinsame Variable bezogen sein, die Bedingungsauswertung kann allerdings immer nach unterschiedlichen algorithmischen Konstruktionen geschehen und dabei auch andere Variablen einbeziehen. Es ist gemeinhin davon auszugehen, dass nur der Prozess, der warten muss, auch weiß, warum dies so ist. Der Prozess, der die Wertezuweisung an die gemeinsame Variable vornimmt, besitzt für gewöhnlich aber keine Kenntnis darüber, ob und gegebenenfalls welchen anderen, auf dieses Zuweisungsereignis wartenden (`await`) Prozess er auslöst (`cause`) — dazu müsste er einen Überblick auf die wartenden Prozesse und einen Einblick in die zu treffende Auswahlentscheidung der Signalisierungsdisziplin besitzen. Positiver Nebeneffekt der erneuten Überprüfung der jeweiligen Wartebedingung eines Prozesses ist es zudem, falsche Auslösungen beziehungsweise Signalisierungen tolerieren zu können.

Bedingungssynchronisation siehe ↑unilaterale Synchronisation.

Bedingungsvariable (en.) ↑*condition variable*. Eine ↑Synchronisationsvariable, deren gespeicherter Wert unzugänglich ist für einen ↑Prozess; Programmierkonvention. Kommt für gewöhnlich in einem ↑Monitor zur Anwendung, um den gegenwärtigen Prozess auf ein ↑Ereignis zu synchronisieren und bei der Blockierung implizit den wechselseitigen Ausschluss aufzuheben. Macht die ↑unilaterale Synchronisation von Prozessen möglich, unter der Prämisse, dass zu einem Zeitpunkt höchstens ein Prozess im Monitor sein darf.
Auf der ↑Variablen sind nur zwei Operationen definiert: `wait` (a. `await`), um das durch die Variable repräsentierte Ereignis zu erwarten, den Prozess zu blockieren, und `signal` (a. `cause`), um eben dieses Ereignis anzuzeigen, einen Prozess zu deblockieren. Erwartet kein Prozess das Ereignis, ist `signal` wirkungslos. Dies bedeutet aber auch, dass eine Ereignisanzeige in Form eines Zustands für gewöhnlich nicht in der Variablen gespeichert wird. Im Gegensatz dazu speichert `wait` den Zustand, dass einer oder mehrere Prozesse in Erwartung auf ein Ereignis blockiert sind. In dem Fall ist mit jedem ↑Exemplar einer solchen Variablen eine ↑Ereigniswarteschlange assoziiert.
Untrennbar verbunden mit dem Monitor, das heißt, einem Konzept der *Typisierung* in höheren Programmiersprachen, indem nämlich die korrekte Verwendung von Operationen zur ↑Synchronisation durch einen ↑Kompilierer abprüfbar wird und so Programmierfehler vermieden werden können. Die zur ↑Bedingungssynchronisation erforderlichen Anweisungen erzeugt der Kompilierer, wie auch die Instruktionen, um den Monitor zeitweilig verlassen zu können, ohne diesen in der Zwischenzeit gesperrt zu halten. Mangels Sprachunterstützung ist dieses Konzept jedoch viel mehr zur Programmierkonvention entartet — das jedoch immer noch erleichtert, ein ↑nichtsequentielles Programm zu formulieren.

Exkurs Zur Implementierung einer solchen Synchronisationsvariablen bieten sich verschiedene Techniken an, die zwar funktional gleiche, aber in nichtfunktionaler Hinsicht teils sehr

unterschiedliche Lösungsvarianten hervorbringen. Die besondere Herausforderung dabei ist die als ↑kritischer Abschnitt ausgezeichnete Umgebung, in der die Synchronisationsvariable verwendet wird. Sollte ein Prozess bei Verwendung dieser Variablen blockieren und damit den ↑Prozessor an einen anderen Prozess abgeben, muss vorher der umfassende kritische Abschnitt verlassen und danach, wenn der betreffende Prozess nach seiner Deblockierung wieder den Prozessor erhalten hat und fortschreitet, wieder betreten werden. Dieser Ablauf muss mit ↑Wettlaufsituationen umgehen, denen durch geeignete algorithmische Konstruktionen vorzubeugen sind. Zunächst fällt dazu ein Blick auf den ↑Datentypen solch einer Synchronisationsvariablen:

```
typedef struct {
    semaphore_t post;                    /* binary signalling: initially 0 */
    int load;                       /* # of waiting processes: initially 0 */
} event_t;
```

Grundlage für die Implementierung ist der ↑binärer Semaphor, der die unilaterale Synchronisation zur Ereignismitteilung durch ein ↑Signal sicherstellt (`post`). Zur weiteren Kontrolle des Signalaustauschs dient ein Zähler, mit dem Buch über die Anzahl wartender Prozesse geführt wird (`load`).
Darauf aufbauend kann die Operation, um das Ereignis, nämlich die Aufhebung einer Wartebedingung in einem kritischen Abschnitt, abzuwarten (`await`), wie folgt verlaufen:

```
void await (event_t *event, semaphore_t *mutex) {
    FAA(&event->load, +1);                   /* subscribe waiting process */
    V(mutex);                                   /* leave critical section */
    P(&event->post);                                     /* acquire signal */
    FAA(&event->load, -1);                 /* unsubscribe waiting process */
    P(mutex);                                /* re-enter critical section */
}
```

Im ersten Schritt wird der das Ereignis erwartende Prozess verbucht. Dies muss eine ↑unteilbare Anweisung bilden, hier die ↑atomare Operation ↑FAA, um einer Wettlaufsituation (↑*lost wake-up*) mit dem das Ereignis anzeigenden Prozess vorzubeugen. Nachdem sich der noch „schwebend wartende“ Prozess angemeldet hat, verlässt er den umfassenden kritischen Abschnitt (`V`), aus dem heraus das hier betrachtete ↑Unterprogramm aufgerufen wurde. Um das Ereignis der Aufhebung seiner Wartebedingung wahrzunehmen, wartet der Prozess anschließend den Empfang des betreffenden Signals ab (`P`). Dabei ist die Auslegung als binärer Semaphor zu beachten, sie ist wesentlich, um nämlich ↑falsche Signalisierungen zu tolerieren — ein ↑allgemeiner Semaphor würde jedes Signal zählen und damit speichern, auch wenn die betreffende Signalisierung von einem ↑Fehler im nichtsequentiellen Programm herrührt. Wurde dem Prozess das Signal zugestellt und hat er einen Verlauf wieder aufgenommen, trägt er sich als wartender Prozess wieder aus (`FAA`) und bewirbt sich erneut (`P`) um den Eintritt in den kritischen Abschnitt, in dem er nach Rücksprung aus dem Unterprogramm weiter voranschreiten wird.
Die dazu korrespondierende Operation, mit der das Ereignis der Enträftung der Wartebedingung eines Prozesses verursacht wird, zeigt nachfolgende Skizze:

```
void cause (event_t *event) {
    if (event->load > 0)                            /* any waiting process */
        V(&event->post);                            /* yes, release signal */
}
```

Die Operation ist vergleichsweise einfach, ihre Funktion besteht lediglich darin, das Signal anzuzeigen, worauf sich ein wartender Prozess synchronisiert hat beziehungsweise haben könnte. Da das Zählen wartender Prozesse als ↑atomarer Befehl realisiert ist (FAA, s. `await`) und die An- und Abmeldung eines wartenden Prozesses von ihm selbst geschieht, ruft die Abfrage des Zählers (`load`) und Reaktion darauf keine Wettlaufsituation hervor! In dem ganzen Zusammenhang ist auch zu beachten, dass falschen Signalisierungen nicht vorgebeugt werden

könnte, indem die Signalisierungsoperation (`cause`) (a) als kritischer Abschnitt abläuft und (b) darin die Abmeldung eines wartenden Prozesses geschieht — also nicht der signalisierte Prozess (in `await`) selbst, sondern der ihn signalisierende Prozess die Abmeldung vornimmt.

Befähigung (en.) ↑*capability*. Gabe von einem ↑Subjekt, eine bestimmte Operation auf ein ↑Objekt durchzuführen. Ein übertragbares, fälschungssicheres Merkmal, das einen ↑Prozess zu einer bestimmten ↑Aktion ermächtigt. In technischer Hinsicht ist diesem Merkmal ein bestimmter Wert zugeschrieben, der die ↑Referenz auf ein Objekt einschließlich ↑Zugriffsrecht repräsentiert. Dieser Wert ist in einer ↑Variablen gespeichert, die das den betreffenden Prozess festlegende ↑Programm definiert: die Gabe ist damit ein physischer Bestandteil des Prozesses, sie ist mit dem Subjekt verknüpft und gegebenenfalls im jeweiligen ↑Prozessadressraum daselbst gespeichert (im Gegensatz zur ↑ACL). Ein Prozess kann über mehrere solcher Gaben für dasselbe Objekt oder verschiedene Objekte verfügen. Mit jeder dieser Gabe ist der Prozess sodann autorisiert, die damit jeweils verbundene Aktion stattfinden zu lassen. Bei Auslösung der Aktion bezogen auf ein bestimmtes Objekt muss der Prozess seine Gabe dazu explizit machen, er trägt den Schlüssel (*key*) für die gewünschte Aktion bei sich. Ist die Gabe unzureichend, tritt eine ↑Ausnahmesituation ein.
Anders als bei einer ↑Zugriffskontrollliste ist der Widerruf (*revocation*) eines Zugriffsrechts schwierig, da der Inhalt der Schlüsselvariablen zu invalidieren ist: die ↑Adresse dieser Variablen gehört gegebenenfalls (↑*multi-address space*) einem fremden ↑Adressbereich an (d.h., die Variable liegt in einem anderen ↑Adressraum isoliert von der widerrufenden Autorität) und ist eventuell sogar unbekannt. Noch komplizierter erweist sich der Widerruf aller Zugriffsrechte eines Prozesses auf ein gegebenes Objekt. Wird die ↑Adresse einer Schlüsselvariablen nicht im ↑Betriebssystem verbucht, ist in sämtlichen Prozessen eine ↑Ausnahme zu erheben (↑*broadcast*), deren jeweilige Behandlung die Invalidierung des Inhalts der Schlüsselvariablen zur Folge haben muss. Der Weg über die ↑Ausnahmebehandlung ist ebenfalls notwendig, sollte nur das Zugriffsrecht eines einzelnen Prozesses widerrufen werden müssen.

Befehlssatz (en.) ↑*instruction set*. Menge der Instruktionen, die ein ↑Prozessor ausführen kann. Jede einzelne Instruktion wird auch als ↑Maschinenbefehl bezeichnet.

Befehlssatzebene (en.) ↑*instruction set level*. Bezeichnung für die das ↑Programmiermodell definierende Schicht in einem als ↑Mehrebenenmaschine gekennzeichnetem ↑Rechensystem. Diese der Ausführung von ↑Maschinenbefehlen bestimmten Ebene (↑*conventional machine level*) legt vor allem das grundlegende ↑Speichermodell, den ↑Befehlssatz und die ↑Adressierungsarten fest und präsentiert das ↑Operationsprinzip des ↑Prozessors — sie definiert die ↑ISA. Typisch ist die ausschließlich *binäre Kodierung* der von dem Prozessor dieser Ebene auszuführenden ↑Programme.

Befehlszähler (en.) ↑*program counter*. Register der ↑CPU, das ihrer Befehlsfolgesteuerung (*instruction sequencer*) die gegenwärtige Position im ablaufenden ↑Programm anzeigt; auch ↑PC genannt. Diese Positionangabe ist eine ↑Adresse im ↑Arbeitsspeicher, an der der ↑Abruf- und Ausführungszyklus der CPU einen ↑Maschinenbefehl erwartet (*instruction pointer*) und die pro Zyklus implizit um die Befehlslänge (heute (2025): der Anzahl von Bytes, die der komplette Befehl umfasst) inkrementiert wird. Bei ↑CISC hängt diese Länge vom jeweiligen Befehl ab, bei ↑RISC ist sie normalerweise für alle Befehle gleich.
Je nach CPU geschieht die Weiterschaltung des PC vor (*pre-increment*) oder nach (*post-increment*) dem Abrufen eines Maschinenbefehls: das heißt, im Falle einer ↑Ausnahmesituation bei der Befehlsausführung zeigt der PC entweder auf den Befehl, der die ↑Ausnahme hervorgerufen hat (*pre*: ab Motorola 68020), oder auf seinen Nachfolger (*post*: Motorola 68000/10, IA-32) im Befehlsstrom. Letzterer Fall macht eine ↑Ausnahmebehandlung, die zur Wiederaufnahme der Befehlsausführung führt, gegebenenfalls unmöglich, da der dann im ↑Unterbrechungszyklus gesicherte Inhalt des PC den Nachfolger des gescheiterten Befehls adressiert und eine Rückrechnung der Adresse nur möglich wäre, wenn alle Befehle der CPU gleich lang sind — was für CISC (IA-32) im Allgemeinen nicht zutrifft und auch bei RISC

nicht sein muss. Abhilfe schafft hier eine Präventivmaßnahme der CPU, in der sie nämlich den jeweiligen Inhalt des PC vor Weiterschaltung vermerkt und in Abhängigkeit von der Ausnahme den vermerkten oder den aktuellen Wert im Unterbrechungszyklus sichert, sie also nach der Art der Ausnahme unterscheidet (*fault-class exceptions*, Intel 64 und IA-32). Zusätzlich zur impliziten Veränderung durch ein Inkrement, kann der PC auch explizit durch spezielle Maschinenbefehle einen Wert erhalten. Bei einer CPU, die den PC zum ↑Programmiermodell zugehörig definiert (↑PDP 11 Register R7, ARM7 Register R15), kann die Veränderung sogar durch normale Schreibbefehle geschehen. In beiden Fällen erfolgen damit Sprünge im Programmablauf, womit Fallunterscheidungen, Schleifen, Aufrufe von einem ↑Unterprogramm (ebenso ↑Unterbrechungshandhaber) und Rückkehr daraus erst möglich werden.

Befehlszyklus (en.) ↑*instruction cycle*. Hauptschleife eines ↑Interpreters. Solange der ↑Prozessor, den der Interpreter implementiert, in Betrieb (d.h., angeschaltet) ist, fortwährend den jeweils nächsten, durch den ↑Befehlszähler bestimmten ↑Maschinenbefehl abrufen und ausführen (↑*fetch-execute-cycle*). Dabei wird der Befehlszähler in Abhängigkeit von dem Maschinenbefehl entweder implizit um die Befehlslänge (kein Sprungbefehl) erhöht oder explizit durch den Befehlsoperanden (relativer/absoluter Sprungbefehl) verändert.

Befestigung (en.) ↑*mount*. Vorgang, um ein auf einen ↑Datenträger liegendes und zur Benutzung vorgesehenes ↑Dateisystem in die Haltevorrichtung (↑*mounting point*) eines bereits in Benutzung befindlichen Dateisystems zu hängen, dadurch daran für den Betrieb zu befestigen und verfügbar zu machen. Das Ergebnis des Vorgangs ist aber nicht unwiderruflich, die Montage wird für gewöhnlich gelöst (*dismount*), wenn das eingehängte Dateisystem nicht mehr benötigt beziehungsweise der betreffende Datenträger wieder ausgeworfen wird. Bei einem ↑Wechseldatenträger läuft der Befestugungsvorgang häufig auch automatisch ab, unterstützt durch das ↑Betriebssystem, wenn nämlich der ein (einhängbares) Dateisystem speichernde Datenträger über einen geeigneten Anschluss (z.B. ↑USB) für sein ↑Peripheriegerät verfügbar gemacht wird (↑*plug and play*).

Befestigungspunkt (en.) ↑*mount point*. Stelle im ↑Namensraum von einem ↑Dateisystem für die ↑Befestigung eines weiteren Dateisystems. Typischerweise der ↑Name von einem ↑Verzeichnis, an dessen Stelle sodann das ↑Wurzelverzeichnis des eingehängten Dateisystems tritt.

Behelf (en.) ↑*workaround*. Etwas, womit man sich in Ermangelung eines Besseren behelfen muss (Duden). Oft auch als Behelfslösung bezeichnet, durch die das eigentliche Problem zwar nicht behoben, sondern um es herum gearbeitet wird. Hierzu findet eine Hilfskonstruktion (in Hardware oder Software) Verwendung, die für gewöhnlich einen höheren Aufwand (↑*overhead*) für einen ↑Prozess verursacht, mit der aber das Problem letztlich umgangen wird. Beispiele dafür wären etwa der Trick, um einen ↑Hauptspeicherfehlzugriff zu erzwingen oder die Lösung zum ↑ABA Problem.

Benutzer / Benutzerin (en.) ↑*user*. In der Informatik für gewöhnlich eine Person, die ein ↑Rechensystem oder eine ↑Anwendungen darauf für bestimmte Zwecke nutzt.

Benutzerbereich (en.) ↑*user space*, ↑*userland*. Bezeichnung für den ↑Adressbereich, der einem ↑Maschinenprogramm vom ↑Betriebssystem zugestanden wird. Bezugsrahmen bildet ein bestimmter ↑logischer Adressraum oder ↑virtueller Adressraum, dessen Modell im Betriebssystem verankert ist. In diesem Gebiet handelt die ↑CPU für das Maschinenprogramm (↑*user mode*), darüberhinaus verfügt ein ↑Prozess darin nur über eingeschränkte Rechte (↑*unprivileged mode*).

Benutzerebene (en.) ↑*user level*. Bezeichnung für die Stufe, im Sinne von sowohl Privilegien (↑*unprivileged mode*) als auch Kontext (↑Maschinenprogramm), auf der ein ↑Prozess gegenwärtig stattfindet. Gegenteil von ↑Systemebene.

Benutzerkennung (en.) ↑*user identification*. Zeichen zur eindeutigen Identifizierung derjenigen ↑Entität, die das ↑Rechensystem benutzen darf oder benutzt; für gewöhnlich eine Nummer. Die Kennung wird bei der ↑Anmeldung initial zugeordnet und kann gegebenenfalls im weiteren Verlauf von einem ↑Prozess für sich selbst geändert werden (`setuid(2)`). Der mögliche ↑Kennungswechsel ist hochgradig abhängig vom ↑Betriebssystem.

Benutzerprogramm (en.) ↑*user program*. Bezeichnung für ein ↑Programm gebildet aus all jenen Anweisungen, die nicht der ↑Systemsoftware zuzurechnen sind. Dieses Programm beginnt mit `main` (↑C) und endet für gewöhnlich an der durch die ↑Programmbibliotheken des zugrunde liegenden Systems definierten Schnittstelle.

Benutzerprozess (en.) ↑*user process*. Bezeichnung für einen ↑Prozess, der durch ein ↑Benutzerprogramm definiert ist und für gewöhnlich auf ↑Benutzerebene stattfindet.

Benutzerzeit (en.) ↑*user time*. Bezeichnung für die Zeitspanne, die ein ↑Prozess im ↑Maschinenprogramm (↑*user mode*) verbracht hat. Ein über die gesamte Lebenszeit eines Prozesses akkumulierter Wert. Gegensatz zur ↑Systemzeit.

Benutzthierarchie siehe ↑funktionale Hierarchie.

Bereitliste (en.) ↑*ready list*. Aufstellung der von einem ↑Prozessor zu einem bestimmten Zeitpunkt zu leistenden Arbeit, wobei eine einzelne Arbeitseinheit (↑Aufgabe) durch einen ↑Prozess zu verrichten ist. Jeder der Prozesse auf dieser Liste steht bereit (↑Prozesszustand) zur ↑Einlastung der ↑CPU beziehungsweise eines ↑Rechenkerns.
Eine vom ↑Planer verwaltete ↑Warteschlange, die im Falle mitlaufender (*on-line*) ↑Ablaufplanung von ihm auch fortgeschrieben wird. Ob diese Aufstellung als statische (Tabelle) oder dynamische (einfach/doppelt ↑verkettete Liste) Datenstruktur oder in Kombination von beiden (Vorrang- oder Prioritätenliste) implementiert ist, hängt ganz vom ↑Einplanungsalgorithmus ab. Konkretes Listenelement in all diesen Varianten ist der ↑Prozesskontrollblock, der direkt (ein Verkettungsglied enthaltend) oder indirekt (referenziert durch ein Verkettungsglied) in die Liste aufgenommen wird.

Bereitzeit (en.) ↑*ready time*, r_i^+. Bezeichnung für eine ↑Prozessgröße, die den Zeitpunkt festlegt, zu dem die Durchführung einer ↑Aufgabe frühestens beginnen kann.
Dieser Zeitpunkt ist (im deutschsprachigen Raum) gelegentlich gleichgesetzt mit ↑Auslösezeit, da ab dem Moment die betreffende Aufgabe dem System bekannt ist und zur Durchführung einem bestimmten ↑Prozesses übertragen werden kann. Diese Gleichsetzung ist genau genommen aber nur bedingt zutreffend, und zwar wenn der Prozess sofort mit der Auslösung auch die vollständige Bearbeitung der Aufgabe aufnehmen kann. Für gewöhnlich ist dies in einem realen System aber nicht der Fall. Stattdessen wird mindestens zweistufig in einer ↑Verarbeitungskette vorangeschritten: (1) Auslösung, Formulierung und gegebenenfalls Vorverarbeitung sowie (2) Bereitstellung und Verarbeitung der Aufgabe. Beide Stufen sind *logisch* und *physisch* immer zu unterschiedlichen Zeiten aktiv, mögen aber *virtuell* zur selben Zeit stattfinden (↑*simultaneity*).
Typisches Beispiel dafür ist die Verarbeitung einer ↑Unterbrechungsanforderung. Die Gleichsetzung beider Zeitbegriffe würde bedeuten, dass ab dem Moment der Auslösung der betreffende Prozess (a) komplett durchlaufen (↑*run to completion*) und (b) sonst auch ohne jegliche ↑Unterbrechung voranschreiten kann. Dazu müsste zur Auslösezeit einer ↑Unterbrechungsanforderung implizit eine *totale* ↑*Unterbrechungssperre* durch die Hardware erhoben werden, genau genommen muss diese dann eben auch den ↑NMI einschließen. Da letzteres in Bezug auf den NMI grundsätzlich nicht zutrifft und in ↑Rechensystemen gemeinhin auch ↑kaskadierte Unterbrechungen vorgesehen sind, muss immer davon ausgegangen werden, dass zwischen Auslösung und Verarbeitung einer Aufgabe die Auslösung einer weiteren Aufgabe möglich ist. Damit wird die (erste) Aufgabe unterbrochen, verzögert und erst fortgesetzt, nachdem die Verarbeitung der unterbrechenden (zweiten) Aufgabe zum Abschluss kam. Für die spätere Wiederaufnahme der unterbrochenen Aufgabe wurde diese im ↑Unterbrechungszyklus

passend aufbereitet und (im übertragenen Sinn) schließlich auch implizit bereitgestellt, und zwar zur Auslösezeit der unterbrechenden Aufgabe — jedoch bezogen auf ein und dieselbe Aufgabe geschieht deren Auslösung und Bereitstellung immer zu unterschiedlichen Zeitpunkten.
Neben solch einer impliziten Bereitstellung einer Aufgabe, der ein Prozess im Moment einer Unterbrechungsanforderung gerade nachgeht, ist selbstverständlich auch die explizite Bereitstellung gebräuchlich. Um nämlich die ↑Unterbrechungslatenz möglichst klein zu halten, ist eine Unterbrechungsbehandlung üblicherweise in zwei Phasen unterteilt. Diese Phasen korrespondieren zu den beiden oben genannten Verarbeitungsstufen, und zwar (1) ↑FLIH beziehungsweise (2) ↑SLIH: der FLIH startet im Moment der Unterbrechungsanforderung (Auslösung), erstellt die Aufgabe, die er sodann dem SLIH zur Verarbeitung übergibt (Bereitstellung). Im weiteren Verlauf kann der SLIH wiederum die Verarbeitung der Aufgabe aussetzen und einem Prozess übertragen, wozu er für gewöhnlich die ↑mitlaufende Planung einbezieht (Bereitstellung). Diese Beispiele verdeutlichen, dass die Zeitpunkte für Auslösung und Bereitstellung einer Aufgabe in Wirklichkeit immer auseinander liegen.
In den Fällen, wo eine Differenzierung zwischen den beiden Ereignissen, Auslösung einerseits und Bereitstellung andererseits, geboten oder nicht vernachlässigbar ist, bezeichnet r_i^+ den Zeitpunkt der Bereitstellung und $r_i^0 = r_i$ den Zeitpunkt der Auslösung, wobei gilt: $r_i^+ > r_i^0$. In diesen Zusammenhang ist auch die ↑Ankunftszeit einzuordnen.

best fit (dt.) beste Passung, ↑Platzierungsalgorithmus.

Betriebsart (en.) ↑*operating mode*. Art und Weise des durch ein ↑Betriebssystem vorgesehenen oder ermöglichten Betriebs eines ↑Rechensystems, durch das „In-Funktion-Sein“, die Abwicklung von ↑Programmen in bestimmter Form zu steuern und zu überwachen. Die verschiedenen Ansätze unterscheiden unter anderem:

- die Anzahl der Teilnehmer am Rechnerbetrieb (↑*single-user mode*, ↑*multi-user mode*),
- die Anzahl der Programme, die ein Teilnehmer gleichzeitig in Betrieb haben kann (↑*uniprogramming*, ↑*multiprogramming*),
- die Anzahl der ↑Prozesse, die ein Programm zugleich zulässt (uni-, ↑*multithreading*),
- die Art der Verschränkung der Prozesse, nämlich ob sie pseudo- oder echt parallel stattfinden (↑*parallel processing*),
- Zeitgarantien, die diesen Prozessen eingeräumt werden (keine, ↑*real-time mode*),
- den Grad der Ansprechempfindlichkeit von Prozessen, abgestuft für eine interaktionslose oder interaktive Bedienung und Nutzung des Systems (↑*batch mode*, ↑*conversational mode*) oder
- Techniken zur Optimierung von Betriebsabläufen (↑*remote operation*, ↑*overlapped I/O*, ↑*single-stream batch monitor*, ↑*multi-stream batch monitor*).

Jede dieser Optionen, einzeln für sich genommen oder in geeigneten Kombinationen, begründet eine eigene Rechnerbetriebsart, für die das Betriebssystem jeweils bestimmte Systemfunktionen bereitstellen muss.

Betriebslast (en.) ↑*overhead*. ↑Gemeinkosten durch überschüssigen oder indirekten Bedarf beispielsweise an Rechenzeit, Speicher, Bandbreite oder Energie, um eine bestimmte Aufgabe zu erledigen.

Betriebsmittel (en.) ↑*resource*. Auch als ↑Ressource bezeichnetes etwas zum Betrieb eines ↑Prozesses, ohne das er die ihm gemäß ↑Programm auferlegte ↑Aufgabe nicht erfüllen kann. Ein durch das ↑Betriebssystem den Prozessen zur Verfügung gestelltes Mittel gegenständlicher Form (Hardware, Software) oder als physikalische Größe (Zeit, Energie). Typische gegenständliche Ressourcen der Hardware sind ↑Prozessor, ↑Speicher sowie die ↑Peripheriegeräte, die der Software sind gewisse Bestände von ↑Text oder ↑Daten. Die Ressourcen können

dauerhaft (persistent) oder nur vorübergehend vorhanden sein, entsprechend gelten sie für Prozesse als wiederverwendbar oder konsumierbar.
Als ↑wiederverwendbares Betriebsmittel kann eine Ressource geteilt (*shared*, *preemptive*) oder ungeteilt (*exclusive*, *non-preemptive*) den Prozessen zur Verfügung stehen. Demgegenüber ist sie als ↑konsumierbares Betriebsmittel vermittelbar, ein Kommunikationsmittel (einer Erzeuger-Verbraucher-Beziehung) für Prozesse, oder aufbrauchbar, durch den Fortschritt der Prozesse erschöpfbar. Sowohl die wiederverwendbaren (gegenständlichen) als auch die aufbrauchbaren (messbaren) Ressourcen sind grundsätzlich nur in begrenzter Menge vorhanden, die vermittelbaren (eine Kommunikation ermöglichen) dagegen unbegrenzt.
Allen Ressourcen ist ein bestimmter *Lebenszyklus* gemeinsam. Dieser bedeutet für gewöhnlich, dass jede einzelne Einheit einer Ressource zur Erfüllung einer ↑Aufgabe von einem Prozess angefordert, als Ergebnis einer ↑Systemfunktion dem Prozess zugeteilt/vermittelt, von ihm belegt, benutzt und, nach Aufgabendurchführung, freigegeben/verbraucht wird. Der Zyklus verläuft dynamisch, wobei einzelne Schritte darin jedoch auch einer statischen Auslegung entsprechen können. Dies kann beispielsweise alle Schritte bis auf die Benutzung betreffen, und zwar wenn durch umfängliches ↑Vorwissen die Betriebsmitteleinheiten spätestens zur ↑Ladezeit des betreffenden ↑Maschinenprogramms für den Prozess reserviert werden. Die Freigabe einer wiederverwendbaren Ressource ermöglicht deren erneute Nutzung durch denselben oder einen anderen Prozess. Ist diese Ressource als unteilbar klassifiziert, darf sie einem Prozess jedoch nur exklusiv zugeteilt werden. Für gewöhnlich erfordert dies die ↑multilaterale Synchronisation ↑gleichzeitiger Prozesse, wenn diese dieselbe Einheit einer solchen Ressource für sich beanspruchen. Die Freigabe einer konsumierbaren und vermittelbaren Ressource ermöglicht ebenfalls deren Nutzung, führt aber im weiteren Verlauf zur restlosen Entsorgung (Zerstörung). Zur Anforderung einer (vermittelbaren) Einheit davon ist für gewöhnlich die ↑unilaterale Synchronisation des anfordernden Prozesses erforderlich (↑*producer-consumer problem*).

Betriebsmittelkontrolle (en.) ↑*resource control*. Überprüfung, der ein ↑Prozess bei Zuteilung, Nutzung und Rückgabe von einem ↑Betriebsmittel durch das ↑Betriebssystem unterzogen wird; kontinuierliche Überwachung durch eine dedizierte Steuerung im Betriebssystem, der Prozess und Betriebsmittel unterstehen (↑*resource management*).

Betriebsmittelsperre (en.) ↑*resource lock*. Maßnahme zum Sperren eines ↑Betriebsmittels (in Anlehnung an den Duden), nämlich dass dieses einem ↑Prozess nicht vor Eintritt eines bestimmten ↑Ereignisses zugeteilt wird.

Betriebsmittelverwaltung (en.) ↑*resource management*. ↑Systemfunktion zur ↑Betriebsmittelkontrolle; Operationen im ↑Betriebssystem in ihrer Gesamtheit, um, im ↑Auftrag oder anstelle des eigentlichen besitzenden ↑Prozesses, ↑Betriebsmittel zu betreuen, in Obhut zu haben, in Ordnung zu halten (in Anlehnung an den Duden). Grundsätzliche Ziele dabei sind, anstehende Aufträge konfliktfrei abzuwickeln, diese in endlicher Zeit korrekt zu bearbeiten, eine gleichmäßige und maximierte ↑Auslastung der Betriebsmittel, hoher ↑Durchsatz, kurze ↑Durchlaufzeit und hohe Ausfallsicherheit. Des Weiteren sollte die Anforderung eines Betriebsmittels die ↑Verklemmung von Prozessen ausschließen und die „gleichzeitige“ Anforderung desselben Betriebsmittels durch gleichrangige „gleichzeitige“ Prozesse frei von ↑Verhungern sein. VL 11

Betriebsmodus (en.) ↑*mode of operation*. ↑Arbeitsmodus, in dem das ↑Rechensystem betrieben wird. Normalerweise wird ein ↑Maschinenprogramm direkt durch die ↑CPU ausgeführt (↑*unprivileged mode*). Demgegenüber kommt ein ↑Betriebssystem nur außer der Reihe zur Ausführung (↑*privileged mode*), als ↑Ausnahme (↑*trap*, ↑*interrupt*): zwingend zur Inbetriebnahme des Rechensystems (↑*power-on reset*) und als Option während des Rechenbetriebs, nämlich wenn ↑Adressraumisolation zur Wahrung der ↑Integrität des Betriebssystems vorgeschrieben ist. So erfolgt der ↑Moduswechsel immer auch nur ausnahmebedingt: vom unprivilegierten in den privilegierten Modus und, nach erfolgter ↑Ausnahmebehandlung, wieder

zurück. Dies schließt insbesondere jede Form von ↑Systemaufruf mit ein. Gleiches gilt für die Inbetriebnahme, bei der für den Urwechsel „zurück“ in den unprivilegierten Modus der Zustand einer vermeintlich vor dem Einschalten („Urknall“) ausnahmsweise unterbrochenen und jetzt wieder aufzunehmenden Ausführung eines Maschinenprogramms aufgesetzt wird.

Betriebsseite (en.) ↑*working page*. Element (↑Seite) einer ↑Arbeitsmenge.

Betriebssicherheit (en.) ↑*safety*. Zustand des Geschütztseins der Umgebung von einem ↑Prozess vor Gefahr oder Schaden durch ihn selbst (in Anlehnung an den Duden). Dabei ist mit Umgebung ein bestimmter Kreis von Prozessen gemeint, die intern (↑Programmablauf) oder extern (physikalisch-technischer Prozess) von dem jeweils als ↑Bezugssystem betrachteten ↑Rechensystem stattfinden. Um diesen Zustand zu gewährleisten, sorgt ein ↑Betriebssystem für die Unwirksamkeit jeder ↑Aktion, die zur Verletzung der ↑Schutzdomäne anderer Prozesse führen kann. Grundsätzlich bedeutet dies, unautorisiertes Verlassen der eigenen Schutzdomäne eines Prozesses zu erkennen und abzufangen (↑Ausnahmesituation). Im Falle von ↑Echtzeitbetrieb ist darüber hinaus noch dafür Sorge zu tragen, dass abnormales Zeitverhalten eines beliebigen Prozesses die ↑Echtzeitbedingung eines strikt zeitabhängigen Prozesses nicht verletzt.
Nicht selten ist der Nachweis zu erbringen, dass ein solcher Zustand niemals durch Prozesse, die innerhalb des Rechensystems stattfinden, aufgehoben werden kann. Dazu finden formale Methoden Verwendung, die die formale Korrektheit der Software prüfen und gegebenenfalls bestätigen. Für gewöhnlich durchläuft die Software dazu eine aufwendige Zertifizierung, mit der die Einhaltung spezifizierter Anforderungen durch eine autorisierte und unabhängige Zertifizierungsstelle (z.B. der TÜV) nachgewiesen wird.

Betriebssystem (en.) ↑*operating system*. Bezeichnung für ein ↑Programm oder eine Menge von Programmen, die Programme oder ↑Anwendungen unterstützen und die Programmierung oder Bedienung von einem ↑Rechensystem erleichtern sollen, den ↑Benutzern den Umgang mit dem Rechensystem ermöglichen, die Ausführung von Programmen überwachen und steuern, das Rechensystem nach einer bestimmten Art und Weise für einen ↑Anwendungsfall tätig werden lassen und damit die ↑Betriebsart festlegen, eine ↑abstrakte Maschine (auch: ↑virtuelle Maschine) implementieren. Eine ↑Softwareschicht, die die ↑Betriebsmittel eines Rechensystems verwaltet, die Betriebsmittelvergabe steuert und die Betriebsmittel nach gewissen Kriterien an mitbenutzende Rechenprozesse verteilt. Die für all diese Funktionen benötigten Programme implementieren die sogenannte ↑Betriebssystemebene in dem Rechensystem.

Betriebssystemebene (en.) ↑*operating system machine level*. Bezeichnung für eine oberhalb der ↑Befehlssatzebene angesiedelte Zwischenschicht in einem als ↑Mehrebenenmaschine gekennzeichnetem ↑Rechensystem. Diese Zwischenschicht definiert eine ↑abstrakte Maschine, die den ↑Befehlssatz der Befehlssatzebene um ↑Systemaufrufbefehle erweitert. Die Erweiterungen geben dem Rechensystem beispielsweise folgende Fähigkeiten:

- mehr als einen ↑Prozess zugleich auf der ↑CPU stattfinden zu lassen,
- sie räumlich zu isolieren, das heißt, einen Prozess im ↑Hauptspeicher einzugrenzen (↑logischer Adressraum) beziehungsweise auf einen sich über verschiedene, auseinanderliegende Orte ausdehnenden ↑Arbeitsspeicher abzubilden (↑virtueller Adressraum),
- dem Prozess ↑Ein-/Ausgabe zu ermöglichen, bereitzustellen und alle dazu erforderlichen Abläufe in Bezug auf die ↑Peripherie zu koordinieren,
- die dauerhafte Speicherung von ↑Daten in einem ↑Dateisystem zu erreichen, dies gegebenenfalls auch in einem zum ↑Netzwerk zusammengeschalteten System,
- einen Prozess im Falle eines selbstverschuldeten ↑Fehlers abzufangen (↑*trap*) oder nach einer ↑Unterbrechung unversehrt fortzusetzen und, grundsätzlich,
- den Prozessen die zum weiteren Vorankommen benötigten ↑Betriebsmittel in geeigneter Art und Weise (↑Koordination) verfügbar zu machen.

Typisch ist die ausschließlich *binäre Kodierung* (↑Maschinenkode) der diese Erweiterungen implementierenden ↑Programme (↑Systemfunktionen).
Zur Ausführung eines Programms sind Systemaufrufbefehle nicht zwingend, wohl aber die durch eine bestimmte Befehlssatzebene definierten ↑Maschinenbefehle. Anderenfalls könnte ein ↑Betriebssystem die ihm angestammte Funktion überhaupt nicht erfüllen, nämlich diese Systemaufrufbefehle zu interpretieren (↑*partial interpretation*). Grundsätzlich anders verhält es sich jedoch in Bezug auf ↑Anwendungsprogramme, die in einer ↑Benutzthierarchie zu Betriebssystemen stehen — obwohl ein solches Programm sogar frei von Systemaufrufbefehlen sein kann. Auch wenn ein Betriebssystem lediglich dazu da sein sollte, ein Anwendungsprogramm in den Hauptspeicher zu bringen (d.h., das Betriebssystem ggf. vielleicht nur aus dem ↑Lader besteht), die Hardware (z.B. ↑MMU, ↑MPU, ↑PIC oder ↑Peripheriegerät) für dieses Programm insbesondere so zu initialisieren, dass es sogar ↑Ein-/Ausgabe direkt und in eigener Regie zu führen in der Lage ist, um anschließend den durch dieses Programm definierten (ggf. bewusst niemals endenden) ↑Prozess zu starten, der dann in seinem Verlauf möglicherweise keinen einzigen Systemaufrufbefehl absetzen muss oder absetzt, so wird das korrekte Funktionieren des Anwendungsprogramm eben genau davon abhängen, dass all die dazu erforderlichen Voraussetzungen durch das Betriebssystem zuvor korrekt geschaffen wurden. Ein Anwendungsprogramm benutzt sowohl die Befehlssatzebene als auch das Betriebssystem, es setzt auf die ↑Maschinenprogrammebene auf.
Da das Betriebssystem als Programm oder Programmensemble, das diese Zwischenschicht implementiert, seinerseits direkt auf die Befehlssatzebene aufsetzt, ist es in der Lage, jeden beliebigen Maschinenbefehl der CPU uneingeschränkt zu benutzen. Dies gilt jedoch nicht für ein Anwendungsprogramm, dessen Prozess im Allgemeinen immer nur eine Teilmenge dieser Befehle direkt zur Ausführung bringen kann. So bleibt etwa ein ↑privilegierter Befehl dem Prozess im Anwendungsprogramm (↑*unprivileged mode*) verwehrt und ist ihm bestenfalls nur indirekt zugänglich, nämlich unter Kontrolle des Betriebssystems (↑*privileged mode*) als Folge eines ↑Systemaufrufs. Bezogen auf das ↑Maschinenprogramm als allgemeiner Repräsentant eines beliebigen Anwendungsprogramms bedeutet dies, dass ein Betriebssystem die privilegierten Befehle der Befehlssatzebene maskiert und nur die unprivilegierten Befehle der Befehlssatzebene durchlässt.

Betriebssystemkern (en.) ↑*operating-system kernel*. Herzstück von einem ↑Betriebssystem. Definiert als wesentliche Funktion das ↑Operationsprinzip für eine ↑abstrakte Maschine, die durch ein Betriebssystem implementiert wird (↑*operating system machine level*). Das Operationsprinzip bestimmt letztlich die Architektur des Betriebssystems und damit auch den Funktionsumfang seines Kerns. Darüberhinaus stellt die für den jeweiligen Anwendungsfall vom Betriebssystem zu unterstützende ↑Betriebsart gegebenenfalls weitere Anforderungen, die sich auch in zusätzlicher und vom Kern zu erbringender Funktionalität niederschlagen kann. So lässt sich ohne Angabe der Betriebsart der tatsächliche Funktionsumfang eines solchen Kerns ebenso wenig festlegen wie der eines Betriebssystems. Fordert die Betriebsart beispielsweise eine Form von ↑Simultanverarbeitung, wird der Kern wenigstens Funktionen zur ↑Prozesseinplanung, ↑Unterbrechungsbehandlung und ↑Synchronisierung umfassen. Läuft es zusätzlich noch auf ↑Mehrprogrammbetrieb hinaus, werden im Kern zumindest grundlegende Funktionen zum ↑Speicherschutz und zur ↑Ausnahmebehandlung sowie ein ↑Systemaufrufzuteiler hinzukommen.

Betriebssystemlatenz Bezeichnung für die von einem ↑Betriebssystem zur Durchführung einer bestimmten ↑Systemfunktion anfallenden ↑Latenzzeit. Je nach der ↑Betriebsart beziehungsweise dem ↑Operationsprinzip des Betriebssystems ist damit die Verzögerung, die ein ↑Handlungsstrang im ↑Maschinenprogramm erfahren kann, gegebenenfalls unbestimmt und nicht berechenbar. Ist die Stelle, an der eine solche Verzögerung wirkt, im Fall von einem ↑Systemaufruf im Maschinenprogramm noch bestimmbar, macht demgegenüber eine ↑Unterbrechung eine exakte Vorhersage unmöglich. Bestenfalls kann dann noch die ↑Zwischenankunftszeit von Unterbrechungen abgeschätzt werden, um Phasen der Unterbrechungs- und damit Verzögerungsfreiheit zu veranschlagen.

Bevorrechtigung (en.) ↑*preemption*. Einen ↑Prozess durch Gewähren besonderer Rechte bevorzugen, privilegieren (in Anlehnung an den Duden). Für gewöhnlich bedeutet dies die ↑Verdrängung eines anderen Prozesses, um den von ihm belegten Platz selbst einzunehmen.

Hinweis Der englischsprachige Begriff „*preemption*“ wird ins Deutsche irrtümlicherweise mit „Verdrängung“ übersetzt. Dies ist wohl dem besonderen Umstand geschuldet, wonach es in dem assoziierten Zusammenhang schließlich darum geht, einen Prozess von seinem Betriebsmittel zu verdrängen — um es selbst einzunehmen (in Anlehnung an den Duden). Wenn nämlich davon die Rede ist, dass ein Prozess „preempted“ wird, dann kommt ihm in dem Moment ein anderer, bevorrechtigter Prozess in der weiteren Nutzung des betreffenden Betriebsmittels zuvor (vgl. S. 372).

Bezugspunkt (en.) ↑*point of reference*. Ein für gewöhnlich durch Hardware oder Software repräsentiertes oder bei deren Betrieb daraus hervorgehendes Artefakt eines bestimmten ↑Bezugssystems.

Bezugsrahmen (en.) ↑*frame of reference*. Ein Kontext, ein Zusammenhang, der bestimmte Nebenumstände definiert; ein ↑Bezugssystem.

Bezugssystem (en.) ↑*frame of reference*. Zugrunde liegendes System, Ganzes von Beziehungen und ↑Entitäten (in Anlehnung an den Duden). Ein abstraktes Gebilde, das einen bestimmten Bezug, das heißt, einen Zusammenhang, eine sachliche Verbindung, eine bestimmte Verknüpftheit für eine bestimmte Menge von Entitäten definiert. Eine Gruppe von ↑Bezugspunkten, die einen inhaltlich festgelegten, genau umrissenen ↑Bezugsrahmen bilden.

Bibliothek (en.) ↑*library*. Raum mit Ablagen; ↑Programmbibliothek.

Binärarbitration (en.) ↑*binary arbitration*. Schiedswesen für oder Schlichtung (*arbitration*) von Streitigkeiten zweier ↑gekoppelter Prozesse um eine ↑Entität, mit der zu einem Zeitpunkt nur exklusiv einer der beiden Prozesse umgehen darf. Typischer Fall einer solchen Entität ist allgemein ein ↑unteilbares Betriebsmittel, aber insbesondere auch ein ↑kritischer Abschnitt. Das klassische Beispiel für ein Verfahren, das die Verwendung der betreffenden Entität frei von ↑Verklemmung und ↑Verhungern regelt, ist der Algorithmus von Kessels (vgl. S. 164).

binärer Semaphor (en.) *binary* ↑*semaphore*. Bezeichnung für einen ↑Semaphor, bei dem die ↑Ablaufsteuerung eine *binäre* ↑ *Variable* benutzt. Entweder realisiert als ↑allgemeiner Semaphor mit Wertebereich $[0, 1]$ oder speziell ausgelegt als boolescher elementarer ↑Datentyp (`bool`). Bei letzterer Variante sind die in den Operationen ↑P und ↑V erfolgenden Zustandsänderungen in Bezug auf die enthaltene *boolesche Variable* implizit jeweils eine ↑atomare Operation: *Zuweisung* von `false` (P) beziehungsweise `true` (V).
Die Ablaufsteuerung lässt einen ↑Prozess in P blockieren, wenn der Variablenwert `false` ist. Anderenfalls setzt P die (`true` enthaltene) Variable auf `false` und lässt den Prozess voranschreiten. Allein anhand des Wahrheitswerts ist dann nicht feststellbar, ob ein Prozess in P blockiert und durch V gegebenenfalls zu deblockieren ist. In dem Fall hätte V nur die Wahl, die Variable auf `true` zu setzen und den ↑Planer die ↑Prozesstabelle ablaufen zu lassen (damit höhere ↑Latenzzeit mit sich zu bringen), um gegebenfalls zu deblockierende Prozesse fest- und wieder bereitzustellen. Verwendet P jedoch eine semaphoreigene Datenstruktur als ↑Warteliste, kann V darüber einfach prüfen, ob Prozesse zur Deblockierung anhängig sind. Ist die Warteliste leer, setzt V den Variablenwert auf `true`. Anderenfalls streicht V den nächsten Prozess von der Warteliste (birgt damit dann aber auch Gefahr von ↑Interferenz mit dem Planer) und deblockiert ihn, ohne den Wahrheitswert zu verändern: dieser bleibt `false`, solange noch Prozesse am Semaphor warten.
Typischer Anwendungsfall dieser Art von Semaphor ist ein ↑kritischer Abschnitt oder ↑Monitor beziehungsweise überall dort, wo ↑wechselseitiger Ausschluss erforderlich ist. Aber auch ↑einseitige Synchronisation ist damit praktizierbar, solange die Anzahl der zu einem Zeitpunkt zu signalisierenden ↑Ereignisse nicht größer als 1 ist (vgl. S. 13).

Exkurs Zur Verdeutlichung der prinzipiellen Funktionsweise wird das (ab S. 293) bereits diskutierte Implementierungsmuster eines Semaphors wieder aufgegriffen. Der dort eingeführte Datentyp (`semaphore_t`) samt Operationen für die Statusänderung/-anzeige findet unverändert Verwendung, wobei eben die „binäre Variante“ gilt. Der wesentliche Aspekt bei der hier diskutierten Skizze ist vor allem die Absicherung von P und V, um gleichzeitige ↑Aktionen auf dasselbe ↑Exemplar des Semaphors koordiniert stattfinden zu lassen.
Da jede Art von Semaphor ein Instrument zur ↑Koordinierung ↑gleichzeitiger Prozesse darstellt, ist es natürlich, dass P und V jeweils ein ↑nichtsequentielles Programmteil bilden. Zum koordinierten Ablauf eines ↑Handlungsstrangs in den „Gefahrenzonen“ dieser Programmteile wäre ↑nichtblockierende Synchronisation prinzipiell von Vorteil, da ein Semaphor selbst dazu ausscheidet — denn P und V sind ja überhaupt erst zur Verfügung zu stellen, sie können für den denselben Typen von Semaphor nicht durch sich selbst geschützt werden. Nichblockierende Synchronisation zur Absicherung von P und V zu verwenden, bedingt jedoch einigermaßen komplexe und tiefgreifende Softwarestrukturen, die nicht leicht begreifbar sind. Stattdessen bilden hier Sperrtechniken die Grundlage und es wird darüberhinaus von einem ↑Uniprozessor ausgegangen.
Klassisch ist die ↑Unterbrechungssperre, die zeitweilig die hardwaretechnische Funktion zur Auslösung (pseudo-) gleichzeitiger Prozesse außer Kraft setzt. Damit wird allerdings die ↑Unterbrechungsbehandlung allgemein unterbunden, insbesondere auch jene, die überhaupt keine zu koordinierenden gleichzeitigen Prozesse hervorruft. Differenzierter funktioniert die ↑Wiedereintrittssperre, was allerdings immer noch auch ↑Aktionsfolgen ohne jeden Prozessbezug unterbindet. Als weitere Möglichkeit bietet sich die ↑Verdrängungssperre an. Diese Art von Sperre setzt lediglich den im weiteren Verlauf einer Unterbrechungsbehandlung vom Planer gegebenenfalls zu aktivierenden ↑Umschalter außer Kraft, weshalb unvorhersehbare ↑Prozesswechsel zeitweilig unterbunden werden.
Für die nachfolgende Betrachtung spielt die Art der Sperre allerdings nur eine untergeordnete Rolle, weshalb durch ein Makro (`ZONE`, S. 312) davon abstrahiert wird. Vor diesem Hintergrund lässt sich eine Implementierung für P wie folgt skizzieren:

```
void P(semaphore_t *sema) {
    enter(ZONE);                          /* go through the danger zone */
    if (capacity(sema)) {                /* semaphore still unacquired? */
        decrease(sema);                                  /* yes, take it */
        leave(ZONE);                     /* escape from the danger zone */
    } else /* sleep(&sema->wait) */ {      /* no, delay current process */
        allot(&sema->wait);                 /* label process, queue up */
        leave(ZONE);                     /* escape from the danger zone */
        block();                                       /* suspend process */
    }
}
```

Zwei Änderungen im Vergleich zur Mustervorlage (S. 294) sind auffällig. Zum einen die Anweisungen zum Betreten (`enter`) und Verlassen (`leave`) einer ↑Sperrzone. Beim Eintritt wird die Sperre gesetzt, beim Austritt wird die Sperre aufgehoben. Den Bereich dazwischen durchläuft ein Prozess logisch ungestört (↑*run to completion*), er wird nicht zur Abgabe seines ↑Prozessors gezwungen (*no* ↑*preemption*). Zum anderen die inzeilige Verwendung der Funktion zur ↑Prozessblockade, um den Prozess in den Schlaf (`sleep`, S. 243) zu versetzen. Offensichtlich blockiert ein Prozess innerhalb der Sperrzone. Diese ist jedoch zu verlassen, bevor der Prozessor den Prozessor abgibt. Anderenfalls bleibt die durch die Sperre abgeschaltete Funktion eine unbestimmt lange Zeit außer Kraft. Sind etwa ↑Unterbrechungsanforderungen oder ↑Verdrängungen gesperrt, ist ↑präemptive Planung wirkungslos. Wie auch in anderen Fällen, beispielsweise einer ↑Bedingungsvariablen, muss daher ein blockierter Prozess grundsätzlich außerhalb der von ihm betretenen kritischen Abschnitte warten.
Einen Prozess zu blockieren, ist eine recht aufwendige Operation. Die betreffende Prozedur (`block`) ist Teil des Planers, der den nächsten Prozess auswählt, zuvor gegebenenfalls noch

eine ↑Umplanung vornimmt und, sofern die ↑Bereitliste nicht leer ist, den Umschalter mit dem Prozesswechsel beauftragt. Um die ↑Einplanungslatenz für Prozesse, die als Folge einer Unterbrechungsanforderung ausgelöst werden würden, nicht unnötig zu vergrößern, wird die Sperrzone noch vor diesen Maßnahmen verlassen — aber erst, nachdem sich der Prozess für seinen Weckruf disponiert (`allot`) hat (↑*lost wake-up*).
Das gleiche Kriterium, nämlich nur eine möglichst kleine Sperrzone einzurichten, gilt ebenfalls für die Implementierung von V:

```
void V(semaphore_t *sema) {
    enter(ZONE);                            /* go through the danger zone */
    if (backlog(&sema->wait) == 0) {               /* empty waiting list? */
        increase(sema);                                 /* yes, take back */
        leave(ZONE);                        /* escape from the danger zone */
    } else {                            /* no, wake up a sleeping process */
        leave(ZONE);                        /* escape from the danger zone */
        wakeup(&sema->wait);                       /* unblock next process */
    }
}
```

Die aufwendige Operation hier ist das Aufwecken (`wakeup`, S. 243) eines schlafenden Prozesses. Der aufzuweckende Prozess muss ausgewählt und von der Warteliste (`wait`) gestrichen werden, um dann seine Bereitstellung durch den Planer zu erfahren. Letzteres wird durch einen mehr oder weniger komplexen ↑Einplanungsalgorithmus erreicht, der gegebenenfalls auch die Verdrängung des Prozesses bewirkt, der V aufgerufen hat. Daher wird auch in diesem Fall die Sperrzone noch vor Aktivierung des Planers wieder verlassen.
Es ist davon auszugehen, dass gerade die Operationen, die die Wartelistenverwaltung wie auch die Prozessblockade betreffen, kritische Abschnitte aufweisen. Dennoch sollten die für P und V geltenden Sperrzonen grundsätzlich nicht unnötig ausgedehnt werden. Jede Sperrzone, ↑kritische Region oder eben auch der kritische Abschnitt schränkt ↑Parallelverarbeitung ein. Jedes dieser Konstrukte definiert in jeder Beziehung ein ↑sequentielles Programmteil. Zu lange sequentielle Bereiche — und zwar nicht gemessen an der Anzahl von Programmzeilen, sondern in Bezug auf die zu erwartende ↑Laufzeit — tragen dazu bei, gerade das Leistungsvermögen moderner ↑Mehrkernprozessoren nur ungenügend auszuschöpfen. Darüber hinaus sind ↑Wettlaufsituationen oft auch sehr problemspezifisch, so dass nicht angenommen werden sollte, dass die hier für P und V durchaus praktikable Sperrzonentechnik auch für alle anderen Fälle die adäquate Lösung darstellt.

binary arbitration (dt.) ↑Binärarbitration.

Binden (en.) ↑*linking*. Vorgang, um eine ↑Bindung einzurichten. Ein ↑Binder liest ein ↑Objektmodul nach dem anderen ein, analysiert jedes darin verwendete ↑Symbol nach seinem Geltungsbereich (d.h., ob es lokal oder global sichtbar ist) und versucht jede gegebenenfalls bestehende ↑unaufgelöste Referenz zu löschen. Jedes Objektmodul, das ein global sichtbares (exportiertes) und in einem anderen Objektmodul benutztes (importiertes) Symbol enthält, fließt in das aufzustellende ↑Lademodul ein. Dabei wird entweder das gesamte Objektmodul oder nur die darin enthaltene und exportierte ↑Entität berücksichtigt, um das ↑Text- und ↑Datensegment für das am Ende gebildete ↑Maschinenprogramm jeweils sukzessive aufzubauen. Im ersten Fall kann damit das Lademodul mehr Entitäten als benötigt (d.h. referenziert) aufweisen und entsprechend größeren Platz im ↑Arbeitsspeicher beanspruchen. Die Module liegen als ↑Datei vor oder werden einer ↑Programmbibliothek entnommen.
Je nach ↑Bindezeit operiert ein Binder unterschiedlich. Für komplette Bindungen vor ↑Ladezeit eines Maschinenprogramms darf im Lademodul keine ↑Referenz mehr unaufgelöst sein, anderenfalls endet der Bindevorgang mit einer Fehlermeldung (*unresolved references*) und das betreffende ↑Programm gilt als nicht ablauffähig. Für die Erstellung einer Bindung zur Ladezeit kommt zusätzlich ein ↑bindender Lader ins Spiel, der die noch offenen Bestandteile des Maschinenprogramms einfügt und dabei direkt in den Arbeitsspeicher bringt. In

dem Fall hat der zur Generierung des Lademoduls genutzte (statische) Binder die Bindung in Bezug auf eine unaufgelöst gebliebene Referenz allerdings gültig markiert, das heißt, die referenzierte Entität ist bekannt im ↑Rechensystem und darf auch zur Komplettierung des Maschinenprogramms hinzugezogen und nachgeladen werden. Sehr ähnlich wird auch für die Erstellung einer Bindung zur ↑Laufzeit des Maschinenprogramms verfahren: die zur Laufzeit bestehenden unaufgelösten Referenzen sind gültig, sie werden jedoch erst im Moment ihrer erstmaligen Benutzung (d.h., wenn ein ↑Prozess in eine ↑Bindungsfalle tappt) endgültig gebunden. Hier greift ein ↑dynamischer Binder ein und komplettiert die Bindung, indem er die angeforderte Entität nachlädt.

bindender Lader (en.) ↑*linking loader*. Bezeichnung für einen ↑Lader, der eine in einem ↑Lademodul gegebenenfalls noch enthaltene ↑unaufgelöste Referenz selbst auflöst. Dabei bildet jede dieser Referenzen eine ↑symbolische Adresse von ↑Text oder ↑Daten vorrätig in einem ↑Objektmodul in einer ↑Bibliothek. Die Bibliothek mit dem Objektmodul, das die gesuchte symbolische Adresse in der ↑Symboltabelle listet, wird zur Herstellung einer ↑Bindung herangezogen. Steht in keiner Bibliothek ein solches Objektmodul zur Verfügung, scheitert der Ladevorgang.

Binder (en.) ↑*linker*. Bezeichnung für ein ↑Dienstprogramm, das mehrere Objektmodule zu einem ↑Objektmodul oder ↑Lademodul zusammenfügt.

Bindezeit (en.) ↑*binding time*, ↑*link time*. Zeitpunkt, zu dem ein ↑Programm gebunden wird, nämlich für jedes von diesem Programm verwendete ↑Symbol die ↑Bindung mit einer ↑Adresse stattfindet.

binding time (dt.) ↑Bindezeit.

Bindung (en.) ↑*linkage*. Beziehung zwischen einem ↑Symbol und einer ↑Adresse, repräsentiert durch einen Eintrag in der ↑Symboltabelle. Die Beziehung wird zur ↑Bindezeit eingerichtet, nämlich wenn das mit dem Symbol bezeichnete ↑Objekt (↑Modul, ↑Unterprogramm, ↑Exemplar eines ↑Datentyps) vom ↑Binder gefunden und der Art (↑Text, ↑Daten, ↑BSS) entsprechend dem jeweiligen ↑Segment hinzugefügt wurde, wodurch das Segment wächst.
Gemäß der (zur Segmentbasis) relativen Objektlage in dem Segment, sowie der absoluten Lage des Segments im vom ↑Betriebssystem für ein ↑Maschinenprogramm vorgesehenen ↑Adressraum, kann schließlich das jeweilige Symbol in eine konkrete (reale, logische, virtuelle) ↑Ladeadresse aufgelöst werden: auf jede zur Basis 0 relativen Adresse eines solchen Objekts wird die Anfangsadresse des das Objekt enthaltenen Segments als ↑Verlagerungskonstante addiert. Abschließend wird an jeder Stelle im Maschinenprogramm, an der eine solche Objektreferenz verwendet wird, der korrigierte (verschobene) Adresswert eingetragen. Diese Stellen sind in der ↑Verlagerungstabelle verzeichnet, die zusammen mit der ↑Symboltabelle beiden wichtigsten und pro ↑Objektmodul vom ↑Assemblierer für den Binder erstellten Datenstrukturen. Auch ein ↑bindender Lader richtet solche Bindungen ein, jedoch zur ↑Ladezeit eines Programms. Ein Objektmodul nach dem anderen wird nach den zu aufzulösenden Symbolen abgesucht, wobei diese Module eben erst von einem ↑Übersetzer erzeugt wurden oder einer ↑Bibliothek entnommen werden.

Bindungsfalle (en.) ↑*link trap*. Bezeichnung für eine spezielle ↑Abfangung im ↑Betriebssystem, um bei Bedarf ↑dynamisches Binden durchzusetzen, sobald ein ↑Prozess nämlich eine für ihn zwar gültige jedoch noch ↑unaufgelöste Referenz benutzen sollte.

Bindungsfehler (en.) ↑*linkage fault*. ↑Ausnahmesituation beim Zugriff auf ein ↑Text oder ↑Daten enthaltendes ↑Segment im ↑Arbeitsspeicher. Die von einem ↑Prozess beim Zugriff verwendete ↑symbolische Adresse des Segments ist gültig, jedoch im Moment des Zugriffs hat es noch keinen Platz im Arbeitsspeicher dieses Prozesses. Die Bindung des Segmentnamens (↑Symbol) an eine Segmentadresse geschieht erst zur ↑Laufzeit von dem betreffenden ↑Maschinenprogramm, sie ist dynamisch. Das Betriebssystem enthält dazu einen ↑Binder, der die Auflösung

des Namens in eine ↑Adresse und damit verbunden die Platzierung des Segments im Rahmen einer ↑Ausnahmebehandlung durchführt und den beim Zugriff abgefangenen Prozess danach weiter fortsetzt: ↑partielle Interpretation des Zugriffs.

BIOS Abkürzung für (en.) *Basic Input/Output System.* Ein typischerweise im ↑Festwertspeicher liegendes und für gewöhnlich direkt daraus ausgeführtes ↑residentes Steuerprogramm, das also zum Ablauf nicht erst durch ein ↑Urladeprogramm in den ↑Hauptspeicher gebracht werden muss. Dieses Steuerprogramm gehört zur ↑Firmware eines Rechners und setzt direkt auf der ↑Befehlssatzebene auf.
Ursprünglich wurde damit nur ein in ↑CP/M für ↑Ein-/Ausgabe zuständiges Subsystem bezeichnet, das den maschinenspezifischen Teil des ↑Betriebssystems einschließlich ↑Urlader kapselte. Letzteres steht auch heute (2025) noch im Vordergrund, wobei Betriebssysteme dadurch aber längst nicht als maschinenunabhängig gelten.

bit flipping (dt.) ↑Bitkipper.

bit string (dt.) Bitfolge, Bitkette, ↑Bitleiste.

Bitkipper (en.) ↑*bit flipping.* Bezeichnung für einen im ↑Speicher oder bei der Übertragung von ↑Daten aufgetretener ↑Defekt, der den Wert eines einzelnen Bits verfälscht: aus 0 wurde 1 oder umgekehrt.

Bitleiste (en.) ↑*bit string.* Eine endliche Folge von Bits, in der Länge gemeinhin begrenzt durch die ↑Wortbreite des ↑Prozessors. Ein elementarer ↑Datentyp, auf dessen ↑Exemplare (d.h., ↑Maschinenbefehloperanden) einfache *logische Operationen* definiert sind.

BKL Abkürzung für (en.) *big kernel lock.* Die Bezeichnung einer in ↑Linux anfänglich benutzten ↑Umlaufsperre, durch die ↑wechselseitiger Ausschluss für das gesamte ↑Betriebssystem sichergestellt wurde. In aktuellen Versionen ist diese Sperre nicht mehr vorhanden.

Blattprozedur (en.) ↑*leaf procedure.* Ein ↑Unterprogramm, das kein weiteres Unterprogramm im selben ↑Adressraum aufruft, auch nicht sich selbst.

Block Abschnitt fester Größe in einem ↑Ablagesystem. Die Größe eines solchen Blocks hängt vom Bezugsrahmen ab, beispielsweise: 512 B, ↑Gerätetreiber (Platte: ↑Peripheriegerät); 1 KiB, ↑Betriebssystemkern (↑Linux: `vmstat(8)`); 4 KiB, ↑Dateisystem. Letzteres ist ein typischer Wert für eingangs genannte Arten von ↑Speicher.

block diagram (dt.) ↑Blockdiagramm.

block number (dt.) ↑Blocknummer.

Blockdiagramm (en.) ↑*block diagram.* Graphische Repräsentation der hauptsächlichen Bestandteile oder Funktionen, nämlich die Blöcke, eines bestimmten Systems sowohl in Bezug auf Software als auch Hardware oder einer Kombination davon. Verbindungslinien zwischen diesen Blöcken drücken gewisse Zusammenhänge aus, die auf eine ein- oder beidseitige Interaktion hinweisen. Häufig deutet diese Darstellung eine ↑hierarchische Struktur an, wobei jedoch die Relationen zwischen den Einzelbestandteilen des Gesamtsystems nicht selten uneinheitlich sind und damit verschiedenartige Zusammenhänge in derselben Skizze zum Ausdruck kommen.

blockierende Synchronisation (en.) *blocking* ↑*synchronisation.* Art der ↑Synchronisation, bei der ein ↑Prozess in einer ↑Konkurrenzsituation darauf warten muss, bis er an der Reihe ist, eine bestimmte ↑Aktion oder ↑Aktionsfolge durchzuführen. Die betreffende Aktion/Aktionsfolge ist als ↑kritischer Abschnitt beschrieben, für den eine ↑pessimistische Nebenläufigkeitssteuerung zur Absicherung erfolgt. Grundsätzlich ist diese Art der Synchronisation anfällig für ↑Verklemmungen.
Wesentlicher Aspekt dieser Synchronisationsform ist, dass ein wartender Prozess in logischer

und gegebenenfalls auch physischer Hinsicht die Kontrolle über seinen ↑Prozessor abgegeben hat und nicht mehr eigenständig ins weitere Geschehen eingreifen kann. Entweder modelliert der ↑Prozesszustand eine Situation, in der eine bestimmte, durch den Prozess selbst unveränderliche Bedingung gilt, die ihn als blockiert (*inaktiv*) ausweist. Oder der Prozess bleibt in dieser Situation im Zustand laufend beziehungsweise wechselt von sich aus zeitweilig in den Zustand bereit, solange er (*aktiv*) warten muss. Erst ein anderer Prozess muss die Wartebedingung entkräften und den blockierten Prozess in den bereit-Zustand überführen, wenn er inaktiv war, beziehungsweise freistellen, wenn er aktiv geblieben ist. Gegenteil von ↑nichtblockierende Synchronisation.

blocking time (dt.) ↑Sperrzeit.

Blocknummer (en.) ↑*block number*. Zahl, die einen ↑Block auf einem ↑Datenträger eindeutig kennzeichnet; eine ↑numerische Adresse.

boot block (dt.) ↑Startblock.

boot sector siehe (en.) ↑*boot block*.

bootstrap (dt.) ↑Urladen, ↑Urlader.

bootstrap loader (dt.) ↑Urladeprogramm.

bootstrapping (dt.) ↑Ureingabe.

boundary fence (dt.) ↑Einfriedung.

bounded buffer (dt.) begrenzter ↑Puffer; ↑Erzeuger-Verbraucher-Problem.

bounds register (dt.) ↑Grenzregister.

breakpoint (dt.) ↑Haltepunkt.

Briefkasten (en.) ↑*mailbox*. Ein Medium zur ↑Interprozesskommunikation. In technischer Hinsicht ein ↑Platzhalter mit begrenzter Aufnahmekapazität für ↑Nachrichten, die zwischen kommunizierenden ↑Prozessen ausgetauscht werden sollen (↑*bounded buffer*).

Exkurs Zur Verdeutlichung der prinzipiellen Funktionsweise dieses Mediums skizziert nachfolgendes Beispiel, wie durch vergleichsweise einfache Maßnahmen eine enggekoppelte Kommunikaton ↑gleichzeitiger Prozesse ermöglicht werden kann. Der dem Nachrichtenplatzhalter entsprechende ↑Datentyp ist dabei wie folgt ausgelegt:

```
typedef struct mailbox {          /* placeholder for electronic messages */
    buffer_t pipe;                                          /* bounded buffer */
} mailbox_t;
```

Die hier dargestellte ↑Informationsstruktur zum Botschaftenaustausch (↑*message passing*) erfasst lediglich die ↑Zeiger der zur Weiterleitung (↑*pipe*) bestimmten Nachrichten mittels eines Ringpuffers begrenzter Größe (vgl. S. 86), nicht jedoch die Nachrichten selbst. Letztere werden, wie die Kommunikationsprimitiven unten zeigen, in einer ↑Halde verwaltet. Eine einzelne Nachricht ist dabei von folgender Struktur:

```
typedef struct mail {                                /* electronic message */
    size_t size;                                  /* number of data entries */
    char data[0];                              /* open array of data entries */
} mail_t;
```

Demnach umfasst die Nachricht eine bestimmte Anzahl (`size`) von ↑Bytes, die ihrerseits als „offenes Feld“ (`data`) organisiert sind: die Feldgröße 0 bedeutet hier nicht, dass das betreffende Feld etwa keinen Platz belegt, sondern dass der von dem Feld `data` effektiv beanspruchte Platz erst noch durch den Längenparameter `size` zur ↑Laufzeit bestimmt wird.

Um eine Nachricht aufzugeben, ist zunächst Platz für sie in der Halde zu schaffen. Wenn dies gelingt, wird dorthin der Nachrichteninhalt abgelegt und der Nachrichtenzeiger abschließend im Ringpuffer vermerkt. Die entsprechende Operation (`post`) verläuft folgt:

```
mail_t *post(mailbox_t *shed, const void *data, size_t size) {
    mail_t *mail = (mail_t *)engross(sizeof(mail_t) + size);
    if (mail) {                                /* mail memory allocated? */
        mail->size = size;                      /* yes, define mail size */
        memcpy(&mail->data, data, size);                /* copy mail data */
        put(&shed->pipe, (data_t)(void*)mail);      /* store mail pointer */
    }
    return mail;                            /* indicate success or failure */
}
```

Typischerweise wird die Nachricht auf Anweisung des Absenders in einen ↑Zwischenspeicher geschrieben und dort solange aufbewahrt, bis der Empfänger die Anweisung erteilt, die Nachricht auszulesen. Zur Aufbewahrung beansprucht (`engross`) der Absender ein genügend großes ↑Speicherstück und füllt dies mit ↑Daten zur Nachrichtenlänge (`size`) und zum Nachrichteninhalt (`data`, `memcpy`). Die benötigte Speicherstückgröße ergibt sich aus der Anzahl der Bytes für den Längenparameter und den Nachrichteninhalt. Anschließend wird der Nachrichtenzeiger dem Ringpuffer hinzugefügt (`put`). Sollte der Ringpuffer komplett gefüllt sein, wartet der die Operation (`post`) ausführende Prozess (in `put`, vgl. S. 86) solange, bis er an der Reihe ist, einen der nächsten im Ringpuffer freigewordenen Einträge für sich nutzen zu können (↑*producer-consumer problem*).

Um umgekehrt eine Nachricht aufzunehmen, spezifiziert der Empfänger einen ↑Speicherbereich in seinem ↑Adressraum, in den der Nachrichteninhalt (ganz oder teilweise) abgelegt werden kann. Dazu wird der Zeiger der zu transferierenden Nachricht dem Ringpuffer entnommen und die Nachrichtenlänge mit der Größe des Speicherbereichs abgeglichen. Die entsprechende Operation (`pick`) gestaltet sich wie folgt:

```
size_t pick(mailbox_t *shed, void *data, size_t size) {
    mail_t *mail = (mail_t *)get(&shed->pipe).that;       /* receive mail */

    size_t sent = mail->size;                          /* notice mail size */
    size_t able = size < sent ? size : sent;           /* notice copy size */

    memcpy(data, &mail->data, able);                     /* copy mail data */
    dispose(mail);                                   /* release mail memory */

    return sent;                                      /* indicate mail size */
}
```

Die Operation geht davon aus, dass dem Empfänger immer eine Nachricht zugestellt wird. Sollte keine Nachricht verfügbar sein, was einen leeren Ringpuffer impliziert, wartet der betreffende Prozess (in `get`, vgl. S. 86) solange, bis er an der Reihe ist, einen der nächsten in den Ringpuffer eingehenden Einträge entgegen nehmen zu können. Nach Erhalt einer Nachricht (`mail`) wird die Größe des Datensatzes bestimmt, der in den angegebenen Speicherbereich (`data`) transferiert werden kann. Je nach Größe (`size`) des Speicherbereichs nimmt dieser ganz oder nur teilweise die Nachricht auf. Nach erfolgtem Nachrichtentransfer (`memcpy`) wird die Nachricht verworfen und das zugehörige Speicherstück in der Halde freigegeben (`dispose`). Schlussendlich wird die Länge der eingegangenen Nachricht (`sent`) zurückgeliefert. Damit kann der die Operation (`pick`) durchführende Prozess prüfen, ob die empfangene Nachricht nur teilweise (`size < sent`) im Speicherbereich (`data`) Platz fand.

Der Einfachheit halber erlauben die hier skizzierten Operationen keinen schubweisen Transfer von Nachrichteninhalten. Dies würde bedeuten, (a) die Sendeoperation (`post`) zu unterbrechen, sobald kein ausreichend großer Zwischenspeicher für die Nachricht mehr verfügbar und (b) die Empfangsoperation (`pick`) zu unterbrechen, sobald die der Größe des angegebenen Speicherbereichs entsprechende Aufnahmekapazität erschöpft ist. Wohlgemerkt werden nicht

die Sende- und Empfangsprozesse unterbrochen, sondern nur die von ihnen durchgeführten Sende- und Empfangsoperationen. Beim jeweils nächsten Durchlauf würde jede dieser Operationen (`post` bzw. `pick`) dort weitermachen, wo sie zuvor mit dem Schreiben (`post`) oder Lesen (`pick`) einer Nachricht mangels Speicherplatz hatte aufhören müssen, um schließlich den jeweiligen Rest einer Nachricht entsprechend noch zu verarbeiten.
Abschließend sei noch erwähnt, dass die Haldenoperationen (`engross`, `dispose`) denen aus ↑C bekannten (`malloc`, `free`) entsprechen können:

```
#include <stdlib.h>

inline void *engross(size_t size) { return malloc(size); }
inline void  dispose(void *data)  { free(data); }
```

Wichtig ist lediglich der Aspekt, dass die miteinander kommunizierenden gleichzeitigen Prozesse dieselbe Halde nutzen müssen, da in dieser die auszutauschenden Nachrichten zwischengespeichert vorliegen. Allerdings kann diese Halde sehr problemspezifisch für das hier betrachtete Szenario ausgelegt sein und muss überhaupt nicht mit der für C übereinstimmen. So bietet sich eine *funktionale Abstraktion* von den diesbezüglichen Funktionen in C an.

broadcast (dt.) ↑Rundruf, ↑Sammelruf.

BSD Mehrplatz-/Mehrbenutzerbetriebssystem. Abkürzung für (en.) *Berkeley Software Distribution*, Variante von ↑UNIX, erste Installation 1977 (↑PDP 11).

BSS Abkürzung für (en.) *block started by symbol*, (dt.) Block eingeleitet durch ein ↑Symbol. Der Begriff steht für ausgespartem Raum in einem in ↑Assembliersprache formulierten ↑Programm. Der jeweilige Raum wird durch eine Marke (Symbol) gekennzeichnet, er erhält einen Namen. Seine Größe erhöht den ↑Adresszähler beim ↑Assemblieren entsprechend, ohne jedoch einen Inhalt an seiner Stelle zu hinterlassen: der Raum bleibt leer. Erst beim ↑Laden wird das diesen Raum beanspruchende ↑Datensegment im ↑Arbeitsspeicher initialisiert, indem jedes ↑Speicherwort in diesem Segment den Wert 0 erhält.

Buchführung (en.) ↑*accounting*. Aufzeichnung der von einem ↑Prozess oder einer ↑Prozessinkarnation genutzten ↑Betriebsmittel. Je nach Art des Betriebsmittels werden dabei unterschiedliche Werte erfasst. Für ein ↑wiederverwendbares Betriebsmittel ist etwa die Nutzungsdauer ein wichtiger Wert, wohingegen für ein ↑konsumierbares Betriebsmittel eher die Anzahl relevant ist. Die über die Zeit akkumulierten Werte werden entweder direkt im ↑Prozesskontrollblock gehalten oder indirekt darüber in eigenen Datenstrukturen verwaltet. Die aufgezeichneten ↑Daten dienen einerseits bestimmten strategischen Verfahren in einem ↑Betriebssystem, um einen effizienten Rechnerbetrieb zu gewährleisten, und andererseits aber auch kommerziellen Zwecken, wenn nämlich die Inanspruchnahme allgemein von dem ↑Rechensystem nur gegen Entgelt erfolgt.

buffer (dt.) ↑Puffer.

buffering (dt.) ↑Puffern.

bug (dt.) Störung; technischer Jargon. Der Etymologie nach in der Informatik die Bezeichnung für einen (kleinen) ↑Defekt in der Hardware oder Software eines ↑Rechensystems. Überliefert ist etwa die Geschichte von einer toten Motte gefangen (↑*trapped*) in einem Relais des ↑Mark II, die letztlich zur Fehlfunktion der Hardware geführt hat. Nach Entfernen der Motte (↑*debug*) funktionierte der ↑Rechner wieder. Aber bereits vor der Rechnerära (19. Jh.) wurden mit dem Begriff auch Pannen bezeichnet, etwa in mechanischen Konstruktionen.

build management (dt.) Erstellungsprozess. Vorgang bei der Softwareentwicklung, durch Konfigurierung, Generierung, ↑Übersetzung und ↑Binden ein ↑Programm automatisch zu erzeugen (↑*build pipeline*). Eine der einfachsten Varianten davon ist `make(1)`.

build pipeline (dt.) Erstellungssequenz. Folge einzelner ↑Verarbeitungsketten in der automatisierten Entwicklung des fertigen ↑Lademoduls eines ↑Programms oder Programmsystems (↑*build management*). Dies schließt die Generierung eines zum ↑Urladen bestimmten ↑Betriebssystems mit ein.

built-in command (dt.) ↑integrierter Befehl.

burst (dt.) ↑Häufung, Stoß.

burst mode (dt.) ↑Stoßbetrieb.

burst time (dt.) ↑Stoßzeit.

Bus (en.) ↑*bus*. Sammelleitung zur Übertragung von ↑Daten zwischen mehreren Funktionseinheiten eines ↑Rechners (in Anlehnung an den Duden).

bus (dt.) ↑Bus; Übertragungsweg, Verteilerschiene, Vielfachleitung.

bus error (dt.) ↑Busfehler.

bus lock (dt.) ↑Bussperre.

bus-lock burst (dt.) ↑Häufung von ↑Bussperren. Phase von sehr dicht aufeinander folgenden Bussperren durch anhaltende Ausführung ↑atomarer Befehle vor allem ↑gleichzeitiger Prozesse verschiedener ↑Rechenkerne. Während einer solchen Phase gelingt es den ↑Prozessoren, die keine atomaren Befehle ausführen müssen, kaum, ↑Daten über den ↑Datenbus (lesend oder schreibend) zu transferieren. Die Dauer dieser Phase ist bestimmt durch die ↑Prozesslokalitäten, der in dieser ↑Konkurrenzsituation zur Zeit befindlichen Prozessoren.
Als ↑Bezugssystem kommt ein ↑nichtsequentielles Programm in Frage, in dem für die ↑Synchronisation ↑unteilbare Anweisungen der Befehlssatzebene genutzt werden, das heißt, ↑atomare Operationen. Zudem bildet ein ↑speichergekoppelter Multiprozessor die Ausführungsplattform, über die die gleichzeitigen Prozesse hinweg verteilt stattfinden. ↑Pseudoparallelität ist also nicht der Normalfall, sondern die Prozesse bringen die atomaren Operationen von mehreren Prozessoren zugleich zur Ausführung. In solch einer Konstellation ist es nicht ungewöhnlich, dass atomare Operationen bei ihrer Ausführung in hoher Frequenz und kurzen Zeitabständen gehäuft den gemeinsamen Bus gesperrt, unteilbar nutzen.
Mit Mitteln der ↑Befehlssatzebene lassen sich nur die Symptome einer solche Häufung bekämpfen. Für gewöhnlich kommen hier Verfahren zum Einsatz, die den betroffenen Prozessen ↑Ruhezeiten zwischen den zu wiederholenden atomaren Befehlen verordnen. Dadurch entstehen mehr oder weniger lange Zeitspannen, die frei von Bussperren sind und somit unbeteiligte Prozesse nicht am Buszugang hindern (↑*backoff*).

Busfehler (en.) ↑*bus error*. Bezeichnung einer ↑Ausnahme, die durch eine für den ↑Adressbus ungültige ↑Adresse hervorgerufen wird. Zu dieser Adresse gibt es keinen Adressaten, das heißt, weder ein ↑Speicherwort noch ein ↑Peripheriegerät kann dadurch von der ↑CPU angesprochen werden.

Bussperre (en.) ↑*bus lock*. Maßnahme eines Gerätes zum Sperren eines ↑Busses. Eine Vorrichtung, die verhindert, dass Geräte, die diese Sperre nicht gesetzt haben, den Bus zum Zugriff auf andere Geräte nutzen können. Ein typisches Gerät in dem Zusammenhang ist der ↑Prozessor, insbesondere wenn ein ↑speichergekoppelter Multiprozessor das ↑Bezugssystem darstellt.

busy waiting (dt.) ↑aktives Warten, ↑geschäftiges Warten.

Byte (en.) ↑*byte*. Maßeinheit für die Anzahl der Bits zur Kodierung eines einzelnen Schriftzeichens in einem ↑Rechensystem. Gebräuchliche Längen waren/sind 5, 6, 7 (ASCII), 8 (EBCDIC), 9 oder 12 Bits. Darüberhinaus gab es ↑Rechner mit $\approx 5,644$ (Radix-50) oder 1–36 Bits (einstellbar, ↑DEC PDP-10) pro Schriftzeichen. Wenn nicht anders angegeben, dann werden heute (2025) typischerweise 8 Bits zur Zeichenkodierung angenommen, auch als ↑Oktett bezeichnet.

byte (dt.) ↑Byte, ↑Oktett.

Bytereihenfolge (en.) ↑*endianness*. Reihung der einzelnen ↑Bytes in einem ↑Speicherwort, das als Einheit aus mehr als einem Byte zusammengesetzt ist. So kann entweder das höchstwertigste (*big endian*, Motorola-Format) oder das niederwertigste (*little endian*, Intel-Format) Byte in diesem Wort zuerst gespeichert sein und somit an der kleinsten ↑Adresse in diesem Wort liegen, gefolgt von den jeweils nächsten Bytes desselben Worts. Bei einem 32 Bits breitem Speicherwort von zwei 16-Bit Halbworten, je zwei Bytes, kann die Reihung sowohl in Bezug auf die Halbwörter als auch den Bytes darin verschieden sein (*middle endian*, PDP-Format). Einige ↑Prozessoren erlauben die Einstellung der Reihung (*bi-endian*, z.B. ↑PowerPC).
Zur Verdeutlichung sei angenommen, einer ↑Variablen `value` vom Datentyp `long` (↑C) wird die Dezimalzahl `3 735 943 886` zugewiesen, hexadezimal `DEADFACE` (↑*hexspeak*). Je nach Art der Reihung kann das 32-Bit Speicherwort, das als ↑Platzhalter für die Variable `value` verwendet wird, damit folgende Belegungsmuster aufweisen:

Adresse	*endianness*		
	big	*little*	*middle*
BAFF0000	DE	CE	AD
BAFF0001	AD	FA	DE
BAFF0002	FA	AD	CE
BAFF0003	CE	DE	FA

Wenn nun dieses beispielsweise von einem „Großender“ beschriebene Speicherwort von ihm selbst und darüber hinaus auch von einem „Kleinender“ und einem „Mittelender“ gelesen wird, deuten die betreffenden Prozessoren den Inhalt des Speicherworts unterschiedlich: nämlich als den Zahlenwert `3 735 943 886` (*big*, richtig), `3 472 535 006` (*little*, falsch) und `2 917 060 346` (*middle*, falsch). Sollten also auf gemeinsame ↑Daten zugreifende ↑Prozesse auf Prozessoren mit solch unterschiedlichen Sichten stattfinden, müssen die betreffenden ↑Programme selbst sicherstellen, dass derartige Missdeutungen ausgeschlossen sind.

C Programmiersprache: imperativ, prozedural (1969).

C++ Programmiersprache: imperativ, objektorientiert (1979). Erweiterung von ↑C.

cache (dt.) ↑Zwischenspeicher.

cache coherence (dt.) Zwischenspeicherkohärenz: allg. ↑Speicherkohärenz.

cache line (dt.) ↑Zwischenspeicherzeile.

cache miss (dt.) ↑Zwischenspeicherfehlzugriff.

cache thrashing (dt.) ↑Flattern einer ↑Zwischenspeicherzeile (↑*cache*).

callee-saved register siehe (en.) ↑*non-volatile register*.

caller-saved register siehe (en.) ↑*volatile register*.

calling convention (dt.) ↑Aufrufkonvention.

capability (dt.) ↑Befähigung.

card punch (dt.) ↑Kartenstanzer.

card reader (dt.) ↑Kartenleser.

carriage return (dt.) ↑Wagenrücklauf.

CAS Abkürzung für (en.) ↑*compare and swap*; Bezeichnung für eine ↑atomare Operation mit drei Operanden: `bool CAS(void*, word_t, word_t)`. Der erste Operand ist die ↑Adresse eines ↑Speicherworts, dessen Inhalt mit dem Wert des zweiten Operanden verglichen und abhängig vom Vergleichsausgang mit dem Wert des dritten Operanden definiert wird. Nur bei Übereinstimmung wird der Inhalt des adressierten Speicherworts mit dem Wert des dritten Operanden überschrieben und die Operation gelingt (`true`). Anderenfalls scheitert die Operation (`false`):

```
bool CAS(void *ref, word_t old, word_t new) {
    atomic {                                    /* enforce indivisibility */
        if (*ref != old)                            /* state as expected? */
            return false                                     /* no, fail */
        *ref = new;                                  /* write to memory */
        return true;                                          /* succeed */
    }
}
```

Mit `__sync_bool_compare_and_swap(ref, old, new)` definiert ↑GCC für diese Operation eine Standardfunktion (*atomic built-in*), die als ↑atomarer Befehl ausgelegt ist und beispielsweise für ↑x86 als `lock cmpxchg` erscheint.

Beachte Eine Anwendung dieser Operation ist möglicherweise mit dem ↑ABA Problem konfrontiert. Dabei wird ein Wart in das Speicherwort geschrieben, und zwar in der falschen Annahme, dass zwischenzeitlich keine Änderung geschah, da der Vergleich gelingt – weil eben der ursprüngliche Wert in der Zwischenzeit zurückgeschrieben wurde.
Dieses potentielle Problem betrifft für gewöhnlich jedoch nur die als ↑Maschinenbefehl implementierte Operation, und zwar wenn diese direkt eine Instruktion der ↑Befehlssatzebene ist. Im Falle etwa einer ↑LL/SC-basierten Nachbildung der Operation kann das ABA Problem nicht mehr in Erscheinung treten. Nachfolgend ein Beispiel (vgl. S. 168):

```
inline bool CAS(void *ref, word_t old, word_t new) {
    return (LL(ref) == old) && SC(ref, new);
}
```

Entscheidend hier ist das auf eine Adressreservierung basierende Prinzip, die der Maschinenbefehl ↑LL leistet. Hinzu kommt, dass ein weiteres LL desselben oder eines anderen ↑Prozessors die bestehende Reservierung aufhebt, mit der Folge, dass der Maschinenbefehl ↑SC scheitern wird. Das heißt, wenn man davon ausgeht, dass dem wiederholten und erfolgreichen Schreiben des Datums A in der Folge ABA ein Lesen (LL) vorausgegangen sein muss, hat eben dieses Lesen die bestehende Reservierung gebrochen. Dadurch wird das SC für den konkurrierenden ↑Prozess scheitern, womit dieser auch keinen inkonsistenten Datenzustand wegen einer falsch-positiven Annahme verursacht.

cascaded interrupt (dt.) ↑kaskadierte Unterbrechung.

causal order (dt.) ↑Kausalordnung.

causality (dt.) ↑Kausalität, Kausalzusammenhang.

CCITT Abkürzung für (fr.) *Comité Consultatif International Téléphonique et Télégraphique.*

CD Akürzung für (en.) *compact disc*, (dt.) Kompaktscheibe.

cdecl ↑Aufrufkonvention von ↑C.

ceiling (dt.) ↑Obergrenze.

ceiling blocking (dt.) ↑Obergrenzensperre.

central processing unit (dt.) ↑Zentraleinheit. Abkürzung ↑CPU.

chain blocking (dt.) ↑Kettenblockierung.

channel (dt.) ↑Kanal.

channel program (dt.) ↑Kanalprogramm.

chip (dt.) ↑Halbleiterbaustein.

circular first fit siehe ↑*next fit*.

CISC Abkürzung für (en.) *complex instruction set computer*, (dt.) ↑Rechner mit vielschichtigem (komplexen) ↑Befehlssatz. Der ↑Prozessor in solch einem Rechner kann einen einzelnen ↑Maschinenbefehl als mehrere Einzeloperationen (z.B. Operanden laden, Berechnung durchführen, Ergebnis speichern) auf niedriger Ebene direkt ausführen oder ist zu mehrstufigen Operationen oder ↑Adressierungsarten innerhalb eines einzelnen Maschinenbefehls in der Lage. Für gewöhnlich variieren die Befehlslängen, einerseits wegen verschieden langer Befehlskodes und andererseits wegen der für einen Befehl erlaubten Adressierungsarten. Entsprechend variiert die Anzahl der bei Ausführung benötigten Taktzyklen von Befehl zu Befehl. Anders als beim ↑RISC ist der Grundgedanke hinter der ↑Rechnerarchitektur, die ↑semantische Lücke auch durch Maßnahmen in Hardware zu verkleinern und dafür in Bezug auf ↑Performanz eher einen Kompromiss einzugehen. Typisches Beispiel ist ↑Intel 64.

class (dt.) ↑Klasse.

class hierarchie (dt.) ↑Klassenhierarchie.

CLI Abkürzung für (en.) ↑*command line* ↑*interpreter*. Ein besonderer ↑Kommandointerpreter.

client (dt.) ↑Auftraggeber.

clock rate (dt.) ↑Taktfrequenz.

clock speed siehe (en.) ↑*clock rate*.

command (dt.) ↑Kommando.

command interpreter (dt.) ↑Kommandointerpreter.

command line (dt.) ↑Kommandozeile.

compare and swap (dt.) vergleichen und tauschen; Abkürzung ↑CAS.

competition (dt.) ↑Konkurrenz.

compilation (dt.) ↑Kompilation; Kompilat.

compiler (dt.) ↑Kompilierer.

compiling (dt.) Programmübersetzung, ↑Kompilation.

completion time (dt.) ↑Endzeit.

complex command (dt.) ↑Komplexbefehl.

computer (dt.) ↑Rechner.

computer installation siehe ↑*computer system*.

computer instruction (dt.) ↑Maschinenbefehl.

computer operation (dt.) ↑Programmablauf.

computer science folklore (dt.) ↑Informatikfolklore.

computer system (dt.) ↑Rechenanlage.

concurrency (dt.) ↑Nebenläufigkeit.

concurrency control (dt.) ↑Nebenläufigkeitssteuerung.

Concurrent Pascal Programmiersprache: imperativ, klassenbasiert (Hansen, 1975). Spezialisierung von ↑Pascal in zweierlei Hinsicht. Einerseits eine Erweiterung um Sprachkonstrukte zur Formulierung von ↑Prozessen und ↑Monitoren, um insbesondere Software für ↑Betriebssysteme und zur ↑Prozesssteuerung strukturiert als ↑nichtsequentielles Programm zum Ausdruck bringen zu können. Andererseits aber auch eine Einschränkung auf Sprachkonstrukte zur Formulierung *typsicherer* (sequentieller/nichtsequentieller) ↑Programme (↑Sequential Pascal). Ursprünglich war der ↑Kompilierer selbst in Pascal implementiert und erzeugte ↑Maschinenkode für eine ↑virtuelle Maschine (↑CSIM, *threaded code*).

concurrent process (dt.) ↑nebenläufiger Prozess.

condition variable (dt.) ↑Bedingungsvariable.

condition-code register (dt.) ↑Statusregister.

conditional critical region (dt.) ↑bedingte kritische Region.

congestion resolution (dt.) ↑Stauauflösung.

console log (dt.) ↑Konsolenprotokoll.

constant folding (dt.) ↑Konstantenfaltung.

constant propagation (dt.) ↑Konstantenpropagation.

consumable resource (dt.) ↑konsumierbares Betriebsmittel.

contention (dt.) Wettbewerb, ↑Wettstreit, ↑Konkurrenzsituation.

context (dt.) ↑Kontext.

context switch (dt.) ↑Kontextwechsel.

continuation (dt.) ↑Fortsetzung.

continuous paper (dt.) Endlospapier, ↑Tabellierpapier.

control (dt.) ↑Steuerung.

control bus (dt.) ↑Steuerbus.

control flow (dt.) ↑Kontrollfluss.

control language (dt.) ↑Steuersprache.

control loop (dt.) Regelschleife, ↑Regelkreis.

control structure (dt.) ↑Kontrollstruktur.

control unit (dt.) ↑Steuerwerk.

controller (dt.) Kontroller; ↑Steuereinheit, Steuereinrichtung, ↑Steuerung.

conventional machine level siehe (en.) ↑*instruction set level*.

conversational mode (dt.) ↑Dialogbetrieb.

convoi effect (dt.) ↑Konvoieffekt.

cooperation (dt.) ↑Kooperation.

cooperative scheduling (dt.) ↑kooperative Planung.

coordination (dt.) ↑Koordinierung, ↑Koordination.

core (dt.) ↑Rechenkern.

core dump (dt.) ↑Kernspeicherabzug; auch *memory dump* oder *system dump*.

core memory (dt.) ↑Kernspeicher.

coroutine (dt.) ↑Koroutine.

Cortex Familie von ↑Prozessorkernen geistigen Eigentums (*intellectual property core*), ↑ARM, 32/64-Bit (teils auch 16-Bit, *Thumb*).

covered channel (dt.) ↑verdeckter Kanal.

CP/M Abkürzung für (en.) *Control Program/Monitor*, ein ursprünglich für Mikrorechner auf Basis ↑i8080 entwickeltes ↑Betriebssystem (Digital Research, Inc., 1974). Zentrale Komponenten waren/sind das ↑BIOS und ↑BDOS, beide in ↑Festwertspeichern gehalten, sowie ein für eine ↑Dialogstation konzipierter ↑Kommandointerpreter (CCP, *Console Command Processor*) zur Interaktion mit dem ↑Rechensystem.

CPU Abkürzung für (en.) ↑*central processing unit*.

CPU *bound* (dt.) ↑CPU-belastend. Bezeichnung für eine rechenintensive ↑Aufgabe, einen wenig/nicht interaktiven ↑Prozess. Gegenteil von ↑I/O *bound*.

CPU *burst* (dt.) ↑CPU-Stoß.

CPU *protection* (dt.) ↑CPU-Schutz.

CPU *time* (dt.) ↑Rechenzeit.

CPU-Schutz (en.) ↑CPU *protection*, Vorkehrung in einem ↑Betriebssystem, durch die die Monopolisierung der ↑CPU durch einen einzelnen ↑Prozess verhindert wird. Dazu erhält jeder Prozess die CPU nur für maximal eine ↑Zeitscheibe zugeteilt. Durch Ablauf der Zeitscheibe kommt eine ↑präemptive Planung zur Wirkung, die im weiteren Verlauf dem dadurch jeweils unterbrochenen Prozess die CPU entziehen kann.

CPU-Stoß siehe ↑Rechenstoß.

CR Abkürzung für (en.) ↑*carriage return*. Steuerzeichen, kodiert als $\texttt{0D}_{16}$ in ↑ASCII, ↑EBCDIC und ↑*Unicode*.

critical region (dt.) ↑kritische Region.

critical section (dt.) ↑kritischer Abschnitt.

cross-cutting concern (dt.) ↑Querschnittsbelang.

CSIM Abkürzung für (en.) *complete software interpreter machine*, (dt.) vollständig in Software realisierter ↑Interpreter.

CSMA/CD Abkürzung für (en.) *Carrier Sense Multiple Access/Collision Detection*. Ein asynchrones Medienzugriffsverfahren in der Kommunikationstechnik, das einen Mehrfachzugriff mit Trägerprüfung und Kollisionserkennung durchführt. Bekanntes Beispiel ist ↑Ethernet.

CTSS Mehrplatz-/Mehrbenutzerbetriebssystem. Abkürzung für „*Compatible Time-Sharing System*“. Führte das ↑Zeitteilverfahren ein (↑MLFQ), um einem ↑Prozess die lllusion vom eigenen ↑Prozessor zu verschaffen (↑partielle Virtualisierung) und so mehrere Prozesse zugleich stattfinden lassen zu können (↑Simultanverarbeitung). Das System war kompatibel mit ↑FMS. Erste Installation im Jahr 1961 (modifizierte IBM 7094).

current working directory (dt.) ↑Arbeitsverzeichnis.

cursor (dt.) ↑Schreibmarke.

cycle period (dt.) ↑Periode.

cycle stealing (dt.) ↑Taktentzug.

cycle time (dt.) ↑Durchlaufzeit, ↑Zykluszeit.

Dämon (en.) ↑*demon*. Bezeichnung für einen ↑Prozess, der anhaltend im Hintergrund stattfindet und dabei bestimmte ↑Systemfunktionen erfüllt (z.B. `cron(8)` oder `lpd(8)`). Der Prozess findet in einem dedizierten ↑Maschinenprogramm (↑*system program*), also oberhalb eines Betriebssystems, oder in dem Betriebssystem selbst statt.

dangling pointer (dt.) ↑hängender Zeiger.

dark silicon (dt.) dunkles Silizium. Menge von Schaltkreisen einer integrierten Schaltung, die aufgrund des vorgegebenen Grenzwerts für die thermische Verlustleistung (*thermal design power*, TDP) elektrischer Bauteile nicht mit der nominellen Betriebsspannung (d.h., oberen Nennspannung) versorgt sein darf: Bereiche von Transistoren in einem ↑Prozessor, die durch Überhitzungsschutz abzuschalten oder abgeschaltet sind. Für eine als ↑Mehrkernprozessor ausgelegte ↑CPU bedeutet dies beispielsweise, dass nicht jeder ↑Rechenkern durchgängig in Betrieb ist. Die jeweils abgeschalteten Kerne bilden eine Kühlfläche für angeschaltete Kerne, deren Abwärme dann über diese Fläche abgeführt werden kann.

Darwin Mehrplatz-/Mehrbenutzerbetriebssystem, von ↑UNIX abstammend. Erste Installation März 1999 (PowerPC), Basis für ↑macOS. Programmiert in ↑Objective-C, ↑C++ und ↑C.

data (dt.) ↑Daten.

data bus (dt.) ↑Datenbus.

data flow diagram (dt.) ↑Datenflussdiagramm.

data processing system (dt.) Datenverarbeitungssystem, ↑Rechensystem.

data segment (dt.) ↑Datensegment.

data type (dt.) ↑Datentyp.

Datei (en.) ↑*file*. Kunstwort, von Datenkartei; Bestand von sachlich zusammenhängenden ↑Daten, der dauerhaft im ↑Speicher vorrätig sein kann. Je nach Art der Daten, grob unterschieden in *ausführbare* und *nichtausführbare Datei*. Erstere speichert ein ↑Programm für einen ↑Interpreter. Letztere enthält Daten, die erst durch einen ↑Prozess der Verarbeitung unterzogen werden. Beide Arten sind von einer auch als „echte Datei“ bezeichneten Klasse, im Gegensatz zur ↑Pseudodatei, die nicht der ↑EDV im eigentlichen Sinne dient.

Dateibaum (en.) ↑*file tree*. Struktur, die der ↑Namensraum in einem ↑Dateisystem bildet (↑hierarchischer Namensraum). Die Besonderheit dabei ist die Darstellung als mit der Wurzel (↑*root directory*) nach oben an einer Befestigung (↑*mounting point*) eingehängter Baum.

Dateiname (en.) ↑*file name*. ↑Name, der eine ↑Datei in einem ↑Dateisystem identifziert. Der Name ist für gewöhnlich eindeutig immer nur in Bezug auf den für ihn gültigen ↑Namenskontext. Dieselbe Datei kann (i) innerhalb desselben Namenskontextes über verschiedene oder (ii) in verschiedenen Namenskontexten über denselben Namen identifiziert werden (↑*hard link*).

Dateisystem (en.) ↑*file system*. Prinzip, nach dem Informationen in einem ↑Ablagesystem gegliedert und geordnet sind; Einheit von (dauerhaft) zu speichernde Informationen und ↑Datenträger. Das grundlegende, interne Strukturelement für einen solchen Datenträger ist der ↑Block, aber das nach außen sichtbare ist die ↑Datei, in der ↑Nutzdaten zu beliebigen Zwecken gespeichert sind. Um diese Daten jedoch auf dem Datenträger so zu organisieren, dass sie in effizienter Weise wieder abrufbar und gegebenenfalls auch aktualisierbar sind, bedarf es ↑Verwaltungsdaten. Zu den Verwaltungsdaten zählt eine Zusammenstellung des auf dem Datenträger selbst genutzten Gebiets (↑*partition*) von Blöcken, der in diesem Gebiet (i) freien Blöcke für Nutz- aber auch weitere Verwaltungsdaten und (ii) reservierten Blöcke für Dateidatenstrukturen (↑*inode table*) und Listen freier Einzelstücke dieser Strukturen wie auch fehlerhafter Blöcke (↑*bad block*). Derartige Informationen, zumeist indirekt über Verweise (d.h., Anfangsblock und Blockanzahl in dem Gebiet) zum Ausdruck gebracht, sind Attribute einer zentralen Datenstruktur (↑*superblock*), die redundant über den Datenträger abgelegt ist, um im Fehlerfall die Nutz- und Verwaltungsdaten weiterhin verfügbar zu haben.

Daten (en.) ↑*data*. Zahlenwerte oder (zweckdienlich) kodierte Informationen, die auf Beobachtungen, Messungen, Erhebungen, Angaben oder Berechnungen beruhen und elektronisch (digital) in einem ↑Speicher abgelegt werden können.

Datenbus (en.) ↑*data bus*. Verbindungssystem (↑*bus*) in einem ↑Rechner, über das ↑Daten übermittelt werden.

Datenflussdiagramm (en.) ↑*data flow diagram*. Graphische Repräsentation des Flusses der ↑Daten durch ein ↑Informationssystem. Für gewöhnlich dient eine solche Darstellung der Visualisierung der hauptsächlichen Schritte im datenflussorientierten Zusammenspiel verschiedener Komponenten eines bestimmten (Teil-)Systems.

Datensegment (en.) ↑*data segment*. Bezeichnung für ein ↑Segment, das die ↑Daten eines↑Programms speichert. Je nach ↑Betriebsart ist ein solches Segment gegebenenfalls nicht schreibbar durch „fremde“ ↑Prozesse, das heißt, Prozesse, die nicht der ↑Prozessinkarnation entsprechen, die für das betreffende Programm durch das ↑Betriebssystem eingerichtet wurde.

Datenträger Bezeichnung für ein ↑Speichermedium.

Datentyp (en.) ↑*data type*. Beschreibung einer bestimmten Menge von ↑Daten derselben Struktur und der auf diesen Daten definierten Operationen. Jedes einzelne Datum als Element dieser Menge ist Beispiel (↑*instance*), ↑Exemplar, einer konkreten Ausprägung derselben Art, desselben Typs. Vor oder zur ↑Laufzeit wird sichergestellt, dass die für das Datum beabsichtigte Operation gültig, typkonform ist. Im Fehlerfall, scheitert die ↑Kompilation von dem betreffenden ↑Programm (vor Laufzeit) oder es tritt eine ↑Ausnahmesituation ein (zur Laufzeit).

dauerhaftes Betriebsmittel siehe ↑wiederverwendbares Betriebsmittel.

DCAS Abkürzung für (en.) *double* ↑*compare and swap*. Ein ↑CAS-Maschinenbefehl, der zwei ↑Speicherworte zugleich atomar aktualisieren kann. Die beiden Speicherworte werden entweder über zwei ↑Adressen angesprochen, können damit auch verteilt im ↑Arbeitsspeicher liegen oder haben eine einzige Adresse, liegen dann als Doppelwort vor. Entsprechendes gilt für die weiteren Operanden eines solchen Befehls.

deadline (dt.) ↑Frist, ↑Termin.

deadlock (dt.) ↑Verklemmung.

deadlock avoidance (dt.) ↑Verklemmungsvermeidung.

deadlock prevention (dt.) ↑Verklemmungsvorbeugung.

debug (dt.) entflohen, austesten, ↑Fehler beseitigen.

debugger (dt.) ↑Fehleraufspürer; technischer Jargon für ein Fehlersuchprogramm.

debugging (dt.) ↑Fehlerbereinigung.

DEC Abkürzung für *Digital Equipment Corp.*, auch Digital (1957–1998, Übern. von Compaq).

Defekt (en.) ↑*fault*. Bezeichnung für eine bestimmte Art von ↑Gefahr, die durch Schaden in der Hardware (↑Bitkipper, ↑*spurious interrupt*) oder Software (↑arithmetischer Überlauf, ↑*dangling pointer*) einen ↑Fehler verursachen kann. Anfangsglied in einer ↑Gefahrenkette. Der englischsprachigen Bedeutung nach auch der Mangel an etwas (↑Seite, ↑Segment), nämlich im Sinne von (en.) ↑*page fault* oder (en.) ↑*segment fault*.

Defektblockverwaltung (en.) ↑*bad-block management*. Funktion im ↑Dateisystem oder in der Firmware einer ↑Plattensteuerung, mit der ein ↑defekter Block ausgeblendet und durch einen fehlerfreien Block ersetzt wird. Dazu werden in einem reservierten Bereich auf dem ↑Datenträger Ersatzblöcke vorgehalten. Eine ebenfalls auf diesem Medium liegende Zurodnungstabelle bildet die defekten auf die fehlerfreien Blöcke ab. Diese Tabelle ist entsprechend zu verwalten.

defekter Block (en.) ↑*bad block*. Bezeichnung für einen ↑Block, der unbrauchbar (geworden) ist. Ursache können im ↑Datenträger manifestierte Fertigungsfehler sein, aber auch Fehler die durch Abnutzung oder Alterung auftreten. Ein solcher bei einer Ein-/Ausgabeoperation erkannter Block wird ausgeblendet und durch einen fehlerfreien Block ersetzt (↑*bad-block management*).

deferred procedure call (dt.) ↑zurückgestellter Prozeduraufruf.

Defragmentierung Verfahren zur Auflösung der ↑Fragmentierung. Die im ↑Hauptspeicher belegten Abschnitte werden geeignet verschoben, um einen einzigen zusammenhängenden freien Abschnitt zu schaffen. Voraussetzung dafür ist ein ↑logischer Adressraum für jeden ↑Prozess, dessen im Hauptspeicher liegende Anteile von ↑Text und ↑Daten von der Verschiebung betroffen sind: die ↑reale Adresse eines jeden verschobenen Abschnitts ändert sich, nicht jedoch dessen ↑logische Adresse. Nach jedem Verschiebevorgang wird jeder ↑Seitendeskriptor oder ↑Segmentdeskriptor, der einen verschobenen Abschnitt referenzierte, mit der neuen realen Adresse programmiert. Am Ende des Vorgangs hat die ↑Freispeicherliste nur noch einen Eintrag.

delay loop (dt.) ↑Verzögerungsschleife.

demon (dt.) ↑Dämon.

descriptor (dt.) Beschreiber, ↑Deskriptor.

descriptor privilege level (dt.) durch einen ↑Deskriptor definierte Privilegstufe. Gemeinhin die für ↑x86 (ab Modell ↑i286) gebräuchliche Bezeichnung für ein bestimmtes Vor- oder Sonderrecht, das den Zugriffsmodus eines ↑Subjekts auf ein bestimmtes ↑Objekt festlegt. Für x86 sind vier Stufen möglich, sehr ähnlich zur ↑VAX, wobei Stufe 0 mit den höchsten (↑*system level*: ↑*kernel mode*) und Stufe 3 mit den niedrigsten (↑*user level*) Privilegien verknüpft ist. Jede dieser Stufen ist gleichbedeutend mit einem ↑Schutzring.

Deskriptor (en.) ↑*descriptor*. Kenn- oder Schlüsselwort, durch das der Inhalt einer Information charakterisiert wird und das zur Bestimmung von ↑Text oder ↑Daten im ↑Speicher von einem ↑Rechensystem dient (in Anlehnung an den Duden).
Der Begriff steht auch für den ↑B 5000, der mit solch einen Beschreiber entweder den Wert eines skalaren Elements oder die Lage (↑Adresse des ersten ↑Worts im ↑Kernspeicher oder im ↑Trommelspeicher), Größe (Anzahl der ↑Speicherworte) und Attribute (Typ (↑Text, ↑Daten, ↑E/A), ↑*present bit*) eines ↑Segments des ↑Maschinenprogramms in Hardware erfasste. Letzteres bildet in Form des ↑Segmentdeskriptors die Grundlage von ↑Segmentierung überhaupt, hat als ↑Seitendeskriptor aber auch eine feste Bedeutung bei der ↑Seitenadressierung.

Desktop (en.) ↑*desktop*. Schreibtischmetapher, bei der die für einen ↑Rechner dargestellte Arbeitsfläche einem Schreibtisch nachempfunden ist und die hinterste Ebene bildet, über die dann einzelne Fenster als Bildschirmelemente dargestellt sind.

desktop (dt.) Arbeitsoberfläche, ↑Desktop.

deterministic scheduling (dt.) ↑deterministische Planung.

deterministische Planung (en.) ↑*deterministic scheduling*. Modell der ↑Ablaufplanung, die eine kausale Vorbestimmung allen Geschehens oder Handelns der ↑Prozesse vornimmt. Die Prozessreihenfolge ist durch Vorbedingungen im Voraus eindeutig festgelegt: sie ist, anders als ↑probabilistische Planung, zufallsunabhängig.
Charakteristisches Merkmal ist die ↑vorlaufende Planung, die nämlich über das nötige ↑Vorwissen verfügt, um einen ↑Ablaufplan liefern zu können, der während des Betriebs zu jedem Zeitpunkt bestimmt, mit welchem Prozess im ↑Rechensystem weitergefahren wird. Anderenfalls lassen sich weder kausal vorbestimmte Prozessfolgen herleiten noch eindeutige Aussagen zum Zusammenspiel dieser Prozesse treffen. Das Vorwissen betrifft die Anzahl der einzuplanenden Prozesse, ihre jeweilige ↑Dringlichkeit, eventuelle Abhängigkeiten untereinander sowie Art und Anzahl der jeweils benötigten ↑Betriebsmittel. Für jeden Prozess ist zudem seine ↑Ankunftszeit, ↑Bearbeitungszeit und ↑Frist bekannt. Basierend auf derartigen Parametern erhalten die Prozesse eine bestimmte ↑Priorität, nach der sie schließlich im Rechensystem vorangetrieben werden (↑*fixed-priority scheduling*).

device driver (dt.) ↑Gerätetreiber.

device file (dt.) ↑Gerätedatei.

device number (dt.) ↑Gerätenummer.

device register (dt.) ↑Geräteregister.

devirtualisation (dt.) ↑Entvirtualisierung.

DFS Abkürzung für (en.) *dynamic frequency scaling*.

Dialogbetrieb (en.) ↑*conversational mode*. Bezeichnung für eine ↑Betriebsart, bei der der Austausch von Fragen und Antworten zwischen einem Menschen und dem ↑Rechensystem über eine ↑Dialogstation im Vordergrund steht. Die dabei stattfindende Mensch-Maschine-Interaktion wird durch einen ↑Dialogprozess geführt. Ziel für das ↑Betriebssystem ist es, die ↑Antwortzeit der jeweils gestellten Anfrage wie auch die ↑Durchlaufzeit der damit verbundenen Arbeitsaufträge zu minimieren. Gegebenenfalls ist auch ↑Termineinhaltung zu gewährleisten, wenn nämlich die Antwort zu einer Anfrage innerhalb einer bestimmten Frist vorliegen muss. Letzteres bedeutet, dass das Betriebssystem dann einer vorgegebenen ↑Echtzeitbedingung zu genügen hat. Eine gewisse ↑Vorhersagbarkeit des Systemverhaltens bei Ausführung der Anfragen ist zumindest wünschenswert, in bestimmten Anwendungsfällen sogar gefordert.

Dialogprozess ↑Programmablauf in einem ↑Rechensystem, der die wechselseitige Kommunikation mit einen typischerweise in Gestalt eines Menschen erscheinenden „externen Prozess“ durch Verwendung einer ↑Dialogstation führt. Kontrolliert wird der Programmablauf durch einen ↑Kommandointerpreter, der die Anfragen an das Rechensystem entgegennimmt und ausführt. Dabei kann die Bedienoberfläche textorientiert (↑*shell*), grafisch (↑*desktop*) oder in geeigneter Kombination beider Arten ausgelegt sein.

Dialogstation (en.) ↑*terminal*. ↑Peripheriegerät zur wechselseitigen Kommunikation (Absetzen von Anfragen, Entgegennahme von Antworten) zwischen Mensch und ↑Rechensystem. Für gewöhnlich zwei Geräte vereint: einerseits die Rechnertastatur als Eingabegerät und andererseits der Rechnerbildschirm (↑*terminal*) oder -zeilendrucker (↑TTY) als Ausgabegerät. Je nach technischer Ausführung sind dafür im ↑Betriebssystem gegebenenfalls auch zwei ↑Gerätetreiber erforderlich und nicht nur einer.

die (dt.) ↑Halbleiterplättchen.

Dienst Bündelung von Funktionen in einem ↑Rechensystem nach einer bestimmten fachlichen, thematischen Ausrichtung. Das „Funktionsbündel“ verfügt über eine klar definierte Schnittstelle, durch die ein ↑Auftrag an das Rechensystem abgegeben und das Ergebnis der Auftragsausführung entgegengenommen werden kann. Beispiele sind Informations-, Datenbank-, Buchungs-, Bezahl-, Abrechnungs-, Web-, Netzwerk- oder Systemdienste.

Dienstprogramm (en.) ↑*utility*. Bezeichnung von einem ↑Programm für systemnahe Aufgaben, meist zum ↑Betriebssystem gehörend. Solche Aufgaben bestehen beispielsweise darin, Objektmodule zu einem ↑Lademodul zusammenzubinden (`ld(1)`), Dateiverzeichnisse aufzulisten (`ls(1)`), Dateien zu kopieren (`cp(1)`), Druckaufträge „aufzuspulen“ (`lp(1)`), den Zustand von Prozessen darzustellen (`ps(1)`), Parameter der Netzwerkschnittstelle zu konfigurieren (`ifconfig (8)`) oder Statusdaten im Betriebssystem zu manipulieren (`sysctl (8)`).

direct memory access (dt.) ↑Speicherdirektzugriff.

directory (dt.) ↑Verzeichnis.

directory entry (dt.) ↑Verzeichniseintrag.

direkte Adresse siehe ↑absolute Adresse.

Direktzugriffsspeicher (en.) ↑*random access memory*. Bezeichnung für einen ↑Speicher mit wahlfreiem/direktem Zugriff auf jedes ↑Speicherwort über eine jeweils eindeutig zugeordnete ↑Adresse. Für gewöhnlich auch als ↑RAM abgekürzt, womit aber gleichfalls die Auslegung immer als Schreib-lese-Speicher gemeint ist.

dirty bit siehe (en.) ↑*modified bit*.

disc control (dt.) ↑Plattensteuerung.

disc controller (dt.) Gerät zur ↑Plattensteuerung.

disc memory (dt.) ↑Plattenspeicher.

dispatcher (dt.) ↑Umschalter, Zuteiler.

dispatching (dt.) ↑Einlastung.

dispatching latency (dt.) ↑Einlastungslatenz.

displacement (dt.) ↑Verschiebung, ↑Versatz.

distributed shared memory (dt.) ↑verteilter gemeinsamer Speicher.

DLL Abkürzung für (en.) ↑*dynamic link library*. Kurzbezeichnung für eine ↑dynamische Programmbibliothek mit ↑Bezugspunkt ↑Windows.

DM Abkürzung für (en.) *deadline monotonic*. Eine Methode zur ↑Einplanung, die einer ↑Aufgabe ihre ↑Priorität in Abhängigkeit von ihrem *relativen* ↑Termin fest zuordnet (↑*off-line scheduling*); ein Verfahren zur ↑Prioritätszuweisung an ↑Echtzeitprozesse. Dabei nimmt die Priorität zu, je näher der Termin liegt. Der aus dem Verfahren hervorgehende ↑Ablaufplan wird nachgeschaltet abgearbeitet (↑*fixed-priority scheduling*).
Eine Verallgemeinerung von ↑RM. In besonderen Fällen sogar identisch damit, nämlich falls relativer Termin und Periodenlänge für alle betrachteten Prozesse gleich sind.

DMA Abkürzung für (en.) ↑*direct memory access*.

DMA *controller* (dt.) ↑Steuereinheit für ↑DMA.

document (dt.) ↑Dokument.

Dokument (en.) ↑*document*. Bezeichnung für eine strukturierte, als Einheit erstellte und gespeicherte Menge von ↑Daten (Duden).

dot (dt.) ↑Name vom ↑Arbeitsverzeichnis (↑UNIX).

dot dot (dt.) ↑Name vom ↑Elterverzeichnis (↑UNIX).

DPC Abkürzung für (en.) ↑*deferred procedure call*.

DPL Abkürzung für (en.) ↑*descriptor privilege level*.

DRAM Abkürzung für (en.) *dynamic RAM*, (dt.) dynamischer ↑RAM.

Dringlichkeit (en.) ↑*priority*, ↑*urgency*. Grad, in dem die Bearbeitung eines Auftrags drängt, in dem ein ↑Prozess den ↑Prozessor zugeteilt bekommen oder abgeschlossen werden muss. Typischerweise ein Attribut eines Prozesses, um ↑Vorrangplanung betreiben zu können. Entsprechend ist dieses Attribut durch eine Komponente im ↑Prozesskontrollblock repräsentiert.

Exkurs Die technische Auslegung eines ↑Datentyps für diese Komponente ist problemspezifisch, und zwar abhängig von der Art der im ↑Betriebssystem verwirklichten Vorrangplanung (↑*on-line scheduling*). Im einfachsten Fall geschieht die Modellierung des ↑Rangs eines Prozesses durch eine *Ganzzahl*, deren einzelne Werte unterschiedlichen Rangstufen entsprechen. In bestimmten Fällen ist jedoch eine weitere Strukturierung dieses Datentyps hilfreich, wie beispielsweise nachfolgend skizziert:

```
typedef union urgency {
    int unit;                                          /* plain integer */
    struct {                                               /* bit field */
        unsigned int clan : 8;                   /* static absolute rank */
        unsigned int face : 8;                  /* dynamic absolute rank */
        unsigned int data : 16;            /* user-defined relative rank */
    } part;
} urgency_t;
```

Die statische Komponente (`clan`) repräsentiert die *Stammebene* des Prozesses, ihr Wert identifiziert die Stufe, auf der die ↑Warteschlange des betreffenden Prozesses liegt: Je höher der Wert, desto niedriger der Rang. Verwendet das Verfahren zur Vorrangplanung eine mehrstufig und als Feld organisierte ↑Bereitliste, wie etwa ↑MLQ, ist jeder Stufe eine eigene Warteschlange von gleichrangigen Prozessen zugeordnet. Diese Stufe lässt sich sodann durch die statische Komponente als Feldindex selektieren. Die dynamische Komponente (`face`) repräsentiert die *Wirkebene* des Prozesses, ihr Wert geht von dem der statischen Komponente aus, kann allerdings im Verlauf eines Prozesses variieren. Diese Komponente erlaubt es, den

Rang eines Prozesses phasenweise zu erhöhen oder abzusenken. Im Falle einer ↑präemptiv ausgelegten Vorrangplanung bildet sie den ↑Bezugspunkt für die mögliche ↑Verdrängung des laufenden Prozesses von seinem Prozessor. Die dritte Komponente (`data`) kann beliebig verwendet werden, sie erlaubt insbesondere eine relative Abstufung innerhalb derselben Stamm- oder Wirkebene. Als Sortierschlüssel verwendet lässt sich damit eine von ↑FCFS abweichende Reihenfolge von Prozessen innerhalb der Warteschlange eines jeweiligen Rangs definieren.
Auf diesem Datentyp seien zudem spezielle Zugriffsoperationen definiert, um einerseits eine Abstraktion von der inneren Struktur der jeweiligen ↑Exemplare zu erreichen und andererseits die Operationen auf den einzelnen Komponenten kontrollieren zu können. Der ganzzahlige Wert eines solchen Exemplars liefert folgende Funktion:

```
inline int value(const urgency_t *this) {
    return this->unit;
}
```

Für den initialen Wert identifizieren Stamm- und Wirkebene denselben Rang und der benutzerdefinierte Anteil ist Null. Die entsprechenden Werte der einzelnen Komponente werden wie folgt aufgestellt:

```
inline void range(urgency_t *this, int8_t rank) {
    this->part.clan = rank;
    this->part.face = rank;
    this->part.data = 0;
}
```

Nach einer mit dieser Operation durchführbaren Initialisierung eines diesbezüglichen Exemplars ist die statische Komponente, das heißt, der Wert der Stammebene, nur noch lesbar:

```
inline int8_t level(const urgency_t *this) {
    return this->part.clan;
}
```

Demgegenüber ist die dynamische Komponente schreib- wie auch lesbar, nämlich um den aktuellen Wert der Wirkebene neu einstellen (`angle`) beziehungsweise abmessen (`gauge`) zu können:

```
inline void angle(urgency_t *this, int8_t rank) {
    this->part.face = rank;
}
inline int8_t gauge(const urgency_t *this) {
    return this->part.face;
}
```

Ebenso ist die benutzerdefinierte Komponente schreib- und lesbar, um den Rang eines Prozesses weiter abzustufen (`grade`) beziehungsweise die damit eventuell zum Ausdruck gebrachte Nuance (`shade`) in Erfahrung zu bringen:

```
inline void grade(urgency_t *this, int16_t step) {
    this->part.data = step;
}
inline int16_t shade(const urgency_t *this) {
    return this->part.data;
}
```

Um zu prüfen, welches von zwei gegebenen Exemplaren einen höheren Rang beschreibt, dient folgende Funktion:

```
inline int prior(const urgency_t *this, const urgency_t *next) {
    return level(this) < gauge(next);
}
```

Die Operation liefert den Wahrheitswert „wahr“ (*true*), wenn der erste Parameter (`this`) einen Wert höherer Priorität vorgibt als der zweite Parameter (`next`), das heißt, wenn der

eine Prozess vorrangig (`prior`) gegenüber einem anderen Prozess ist. Dabei entsprechen numerisch kleinere Werte einem höheren Rang.
Schließlich noch eine Operation, mit der der Wert der dynamischen Komponente eines Exemplars schrittweise (nach oben oder unten) geändert werden kann:

```
inline void alter(urgency_t *this, int step) {
    angle(this, gauge(this) + step);
}
```

Mit dieser Operation lässt sich beispielsweise die Änderung der Wirkebene eines Prozesses einrichten, der der ↑Prozessverpflichtung zum Aufstöbern der im Zuge einer Prozessorübergabe möglicherweise bereitgestellten Prozesse nachkommt (bspw. S. 197, `watch` ⇝ `track` ⇝ `alter`). Gegebenenfalls ist die betreffende Suchfunktion mit einem Prozessrang durchzuführen, der höher ist als der Rang der Stammebene des Suchprozesses. Während der Suche verringert dieser Prozess den Rang seiner Wirkebene bei jedem Übergang zum nächsten Eintrag in dem Warteschlangenfeld, bis er seine Stammebene erreicht hat.

Drosseln (en.) ↑*throttling*. Absenkung der Arbeitsgeschwindigkeit der Hardware, um den Energiebedarf und, als Folge daraus, die Betriebstemperatur oder -kosten des ↑Rechensystems — und damit auch die verursachte Umweltbelastung — zu verringern. Die dazu verwendeten Techniken basieren auf Spannungs- (↑DVS) oder Frequenzskalierung (↑DFS).

drum address (dt.) ↑Trommeladresse.

drum machine (dt.) ↑Trommelmaschine.

drum memory (dt.) ↑Trommelspeicher.

DSM Abkürzung für (en.) ↑*distributed shared memory*.

Dualsystem Zahlensystem, das nur zwei Ziffern (0, 1) zur Darstellung aller Zahlen als Potenzen von 2 verwendet.

dump (dt.) ↑Speicherauszug.

Durchlaufzeit (en.) ↑*cycle time*, ↑*through-put time*. Zeitspanne zwischen der Ankunft (↑*arrival time*) von einem ↑Prozess an einer bestimmten Station im System und seinem Weggang von dieser Station nach vollständiger Beendigung (↑*completion time*) seiner dort anfallenden ↑Aufgabe(n). ↑Bezugspunkt für diesen Systemparameter ist für gewöhnlich der ↑Planer, der nämlich bestimmt, wann ein Prozess das erste Mal auf die ↑Bereitliste kommt, wann dieser Prozess zur ↑Einlastung des ↑Prozessors ansteht, wie oft der Prozess einer ↑Verdrängung unterzogen wird und wann der Prozess die ↑Ablaufplanung verlässt. In dem Fall entspricht der Prozessor einer solchen Station. Allgemein bezeichnet die Station ein ↑wiederverwendbares Betriebsmittel, für das ein ↑Auftrag zur Bearbeitung eingetroffen ist.

Durchsatz (en.) ↑*throughput*. Anzahl von Aufträgen, die ein ↑Rechensystem, ↑Betriebssystem oder ↑Prozess pro Zeiteinheit verarbeiten kann. Beispielsweise die Anzahl von ↑Ein-/Ausgabeaufträgen, abgesetzt von einem einzelnen Prozess oder einer Gruppe von Prozessen.

Durchschreibetechnik (en.) ↑*write-through*. Schreibstrategie in Bezug auf eine ↑Zwischenspeicherzeile, die im ↑Zwischenspeicher als Kopie vorliegt. Bei ↑schreibendem Zugriff auf die betreffende Zeile wird sie am Zwischenspeicher vorbei direkt zur nächst höheren Stufe der ↑Speicherhierarchie kopiert, wodurch der betreffende Eintrag im Zwischenspeicher seine Gültigkeit verliert. Dieser Vorgang verläuft über einen ↑Schreibpuffer, um den ↑Prozessor nicht warten zu lassen. Beim nächsten Lesezugriff wird die betreffende Zeile wieder aus der nächst höheren Speicherhierarchiestufe geladen.

DVD Abkürzung für (en.) *digital versatile disc*, (dt.) Universalscheibe für digitale Daten.

DVS Abkürzung für (en.) *dynamic voltage scaling.*

dynamic binding (dt.) ↑dynamisches Binden, auch (en.) ↑*dynamic linking.*

dynamic link library (dt.) ↑dynamische Koppelbibliothek; Abkürzung ↑DLL.

dynamic linker (dt.) ↑dynamischer Binder.

dynamic linking siehe (en.) ↑*dynamic binding.*

dynamic memory (dt.) ↑dynamischer Speicher.

dynamic scheduling (dt.) ↑dynamische Planung.

dynamische Koppelbibliothek (en.) ↑*dynamic link library.* Eine ↑Programmbibliothek, die ↑programmiertes Binden dynamisch zur ↑Laufzeit eines ↑Maschinenprogramms unterstützt.

dynamische Planung (en.) ↑*dynamic scheduling*, auch ↑mitlaufende Planung. Anders als ↑statische Planung stagniert der ↑Ablaufplan nicht während die betreffenden ↑Prozesse stattfinden.

dynamische Programmbibliothek (en.) *dynamic* ↑*program library.* Bezeichnung für eine ↑Programmbibliothek, aus der heraus Bestände von ↑Text oder ↑Daten mitlaufend mit dem betreffenden ↑Prozess in den ↑Arbeitsspeicher geladen werden, das heißt, die ↑Bindezeit von einem ↑Maschinenprogramm entspricht seiner ↑Ladezeit oder ↑Laufzeit. Zur Ladezeit kommt ein ↑bindender Lader ins Spiel, der automatisch die im ↑Lademodul noch als unaufgelöst geltenden Text- oder Datenreferenzen feststellt, die entsprechenden Bestände einer solchen Bibliothek entnimmt und sie mit dem Maschinenprogramm zusammen in den Arbeitsspeicher lädt, wenn sie nicht bereits geladen wurden. Zur Laufzeit setzt ein in dem (noch nicht komplett gebundenen) Maschinenprogramm stattfindender Prozess explizit einen ↑Systemaufruf ab, um die benötigte Programmbibliothek im Arbeitsspeicher zugänglich zu machen (↑UNIX: `dlopen(3)`; ↑Windows: `LoadLibrary`). Die dabei anfallende ↑Aktionsfolge bedeutet jedoch <u>kein</u> ↑dynamisches Binden, wo ein Fehlzugriff auf Text oder Daten den Auslöser bildet und die Einbindung einer solchen ↑Entität durch ↑partielle Interpretation des Zugriffs bewerkstelligt wird. Stattdessen wird die betreffende Entität durch ↑programmiertes Binden dem Maschinenprogramm hinzugefügt. Beide Fälle (Lade-/Laufzeit) führen für gewöhnlich zur Mitbenutzung einer schon im Hauptspeicher liegenden Bibliothek oder Teile davon durch mehrere Prozesse (↑*shared library*) und erfordern damit den Zugriff auf denselben ↑Speicherbereich (↑*shared memory*) durch den jeweiligen ↑Prozessadressraum.

dynamischer Binder (en.) ↑*dynamic linker.* Bezeichnung für einen ↑Binder, der als ↑Systemfunktion integriert in einem ↑Betriebssystem den Dienst vollbringt, eine ↑Bindung zu einem ↑Text- oder ↑Datensegment herzustellen (↑dynamisches Binden). Der Binder behandelt den in einem ↑Prozess aufgetretenen ↑Bindungsfehler und gliedert ein der ↑Ablage entnommenes Text- oder Datensegment in den ↑Adressraum des Prozesses ein. Aktiviert wird der Binder bei Bedarf, im Rahmen der ↑Ausnahmebehandlung bei einem ↑Hauptspeicherfehlzugriff über eine noch als ↑symbolische Adresse ausgelegte und spätere ↑logische Adresse oder ↑virtuelle Adresse von dem betreffenden ↑Segment.

dynamischer Speicher (en.) ↑*dynamic memory.* Bezeichnung für einen ↑Speicher, dessen räumliches Fassungsvermögen (Kapazität) veränderlich ist; Gegenteil von ↑statischer Speicher. Typische Beispiele dafür sind ↑Stapelspeicher und ↑Haldenspeicher. Ersterer wird implizit (automatisch) mit jedem Aufruf von einem ↑Unterprogramm vergrößert und bei Rückkehr davon wieder verkleinert. Letzterer wird explizit in Anspruch genommen, in ↑C beispielsweise durch `malloc(3)` und `free(3)`. Beiden Speichern gemeinsam ist die Zugehörigkeit zu einem ↑Programm, wobei sie aber selbst erst durch einen ↑Prozess dieses Programms, also zur ↑Laufzeit, in Erscheinung treten und zur Wirkung kommen.

dynamisches Binden (en.) ↑*dynamic binding*. ↑Bindung herstellen, und zwar zur ↑Laufzeit von dem betreffenden ↑Programm selbst. Für gewöhnlich wird dieser Mechanismus in heutigen ↑Rechensystemen mit zwei verschiedenen Ebenen (↑*multi-level machine*) in Verbindung gebracht, nämlich der Programmiersprachenebene und der ↑Maschinenprogrammebene. Erstere nimmt dabei Bezug auf eine ↑virtuelle Methode, deren konkrete Implementierung erst vorliegt und referenziert (d.h., gebunden) werden kann ab dem Moment der Erzeugung eines ↑Objekts, dessen Ausprägung unter anderem (↑*inheritance*) bestimmt ist durch die ↑Klasse, die die ↑Signatur dieser Methode enthält.

Der ursprünglichen Bedeutung (Maschinenprogrammebene) nach hat jedoch ein ↑Prozess in einem Programm eine ungebundene ↑symbolische Adresse verwendet und dadurch eine ↑Ausnahmesituation herbeigeführt: die ↑CPU unterbricht die Ausführung von dem ↑Maschinenbefehl, der diese ↑Adresse hervorgebacht hat (↑*trap*) und löst dessen ↑partielle Interpretation durch das ↑Betriebssystem aus. In der Folge lokalisiert ein ↑dynamischer Binder das adressierte ↑Text-/↑Datensegment, durchläuft die ↑Platzierungsstrategie, um das betreffende ↑Segment in den ↑Prozessadressraum einzublenden und eine ↑Ladeadresse zu erhalten und erstellt mit ihr die Bindung. Zum Abschluss wird die CPU instruiert, die Ausführung des betroffenen Maschinenbefehls zu wiederholen (↑*rerun*).

Grundlage für diese Technik bildet zumindest ein ↑logischer Adressraum, denn das in Frage kommende Segment ist in dem Programm durch ein ↑Symbol repräsentiert und damit auch in der, in dem ↑Lademodul dieses Programms enthaltenen, ↑Symboltabelle vermerkt. Dieses Symbol wurde bei der ↑Übersetzung des Programms erfasst: es steht beispielsweise für ein ↑Unterprogramm, auf das durch einen Maschinenbefehl zum Prozeduraufruf (`call`, x86) kodiert im Programm Bezug genommen; oder es steht etwa für ein ↑Exemplar eines beliebigen Datentypen, auf das lesend/schreibend durch einen Maschinenbefehl (`mov`, x86) kodiert im Programm zugegriffen wird. Die Gültigkeit der damit gegebenen symbolischen Adresse wurde also vor Laufzeit des Programms im Rahmen der in dieser Phase anfallenden Übersetzungs- und (statischen) Bindevorgänge bestätigt, nicht jedoch die Präsenz von ↑Text oder ↑Daten im Lademodul dazu.

Bildet ein ↑virtueller Adressraum die erweiterte Grundlage, so lässt sich letztendlich der ↑Arbeitsspeicher eines Prozesses über die komplette ↑Ablage im ↑Rechensystem ausdehnen. Die dynamisch eingebundenen und durch gewöhnliche Adressen referenzierten Text- und Datensegmente werden bereits im Falle eines logischen Adressraums automatisch und für den Prozess funktional nicht wahrnehmbar aus der Ablage geholt, sie liegen dort vielleicht jeweils in einer eigenen ↑Datei vor und wurden womöglich eben keiner ↑Programmbibliothek entnommen. Allerdings erfolgte der „transparente Zugriff“ nur lesend. Durch den virtuellen Adressraum erscheint dieser gesamte ↑Speicher jedoch als ↑virtueller Speicher, womit der „transparente Zugriff“ darauf sodann lesend und schreibend geschehen kann. Dies bedeutet insbesondere, dass jede Datei ein gewöhnliches und direkt im Programm adressierbares Segment darstellt: sie wird ohnehin für gewöhnlich symbolisch adressiert, dann eben implizit bei Zugriffen dynamisch eingebunden und damit ohne expliziten ↑Systemaufruf (`open(2)`, `read(2)`, `write(2)`, `close(2)`) zugänglich. Pionierarbeit dazu leistete ↑Multics, das diesen Ansatz erstmalig umgesetzt hat und so jede im Rechensystem gespeicherte Information jedem Prozess in kontrollierter Weise (durch partielle Interpretation der Speicherzugriffe, scheinbar) direkt zugänglich machte — ein Merkmal, das heutige (2025) Betriebssysteme bei all ihrer Komplexität in anderen Belangen vermissen lassen.

E/A Abkürzung für ↑Ein-/Ausgabe.

E/A-Register Abkürzung für ↑Ein-/Ausgaberegister.

E/A-Stoß Abkürzung für ↑Ein-/Ausgabestoß.

EBCDIC Abkürzung für (en.) *extended binary coded decimal interchange code*, IBM (1963/64). Ein 8-Bit Kode, dessen 256 mögliche Kodewörter jedoch nicht alle verwendet werden. Die

ersten 64 Kodewörter sind Steuerzeichen. Seine Besonderheit ist der enge Bezug zum Standardformat einer ↑Lochkarte (IBM: 80 Zeichen pro Zeile, 12 Zeilen) bei der die Buchstaben A–I, J–R und S–Z jeweils in der numerischen Zone (d.h., die Ziffern 0–9) kodiert werden. Der Kode wurde mit ↑IBM System/360 weitläufig eingeführt und findet vornehmlich in der Großrechnerdomäne (↑*mainframe*) Verwendung.

Echtzeit (en.) ↑*real time*. Bezeichnung für die tatsächlich in der Realität verstreichende Zeit. Im Falle eines ↑Echtzeitrechensystems die einem ↑Prozess im ↑Rechensystem vorgegebene Zeit, an die er sich (mehr oder weniger strikt) zu halten hat. Damit der Prozess seine Zeitvorgaben einhalten kann, ist ↑Echtzeitbetrieb erforderlich. Das bedeutet: Betrieb von einem Rechensystem, bei dem ein ↑Programm zur Verarbeitung anfallender Daten ständig betriebsbereit ist, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen (nach DIN ISO/IEC 2382).

Echtzeitachse (en.) ↑*real-time axis*. Als gerade Linie grafisch dargestellter Verlauf der ↑Echtzeit (in Anlehnung an den Duden). Für gewöhnlich die Abszissenachse (x) eines rechtshändigen kartesischen Koordinatensystems, auf der die einzelnen Größen bestimmte Punkte der Echtzeit angeben (t). Allgemein ein bestimmter zeitlicher Verlauf, wobei sich die Zeitpunkte auf die Echtzeit beziehen.

Echtzeitbedingung (en.) ↑*real-time condition*. Umtand in Bezug auf die vorgegebene Zeit, die ein ↑Prozess in der Realität verbrauchen darf. Für den Prozess ist ein Termin festgelegt, zu dem ein durch ihn selbst herbeigeführtes Berechnungsergebnis vorliegen sollte oder muss. Dieser Termin hat eine bestimmte *Kritikalität* und ist als weich (*soft*), fest (*firm*) oder hart (*hard*) qualifiziert; Synonyme für weich und hart sind schwach beziehungsweise strikt:

- Ein weicher/schwacher Termin darf von dem Prozess überschritten werden. Der Prozess wird bei Terminüberschreitung nicht abgebrochen. Ein von ihm verspätet geliefertes Berechnungsergebnis wird nicht verworfen, jedoch nimmt die Bedeutung dieses Ergebnisses mit zunehmender Verspätung immer weiter ab.
- Ein fester Termin darf von dem Prozess überschritten werden. Der Prozess wird bei Terminüberschreitung abgebrochen, aber erneut für den nächsten Durchlauf (↑Periode) bereitgestellt. Ein Berechnungsergebnis wird nicht geliefert.
- Ein harter/strikter Termin darf von dem Prozess nicht überschritten werden. Eine gegebenenfalls dennoch stattfindende Terminüberschreitung durch den Prozess ist eine ↑Ausnahmesituation, deren Behandlung im Kontext der Anwendung (↑Maschinenprogramm) zu erfolgen und das System in einen sicheren Zustand zu bringen hat. Ein Berechnungsergebnis wird nicht geliefert.

In jedem Fall überprüft das ↑Betriebssystem fortlaufend für jeden solchen zeitabhängigen Prozess, ob eine Terminüberschreitung vorliegt und leitet entsprechend der jeweiligen Kritikalitätsstufe die erforderlichen Maßnahmen ein. Nur im Falle der Überschreitung eines harten Termins übergibt das Betriebssystem die Zuständigkeit für den weiteren Rechenbetrieb an das Maschinenprogramm, da nur die Anwendung selbst den sicheren Zustand des Systems zu erkennen und herbeizuführen vermag — wenn überhaupt. Wird das System nicht in einen sicheren Zustand überführt, kann dies eine *Katastrophe* zur Folge haben: eine größere Gefährdungs- und Gefahrenlage oder ein Schadensereignis materieller oder wirtschaftlicher Natur oder bezogen auf ein oder mehrere Lebewesen (Mensch, Tier).
Hintergrund ist für gewöhnlich der ↑Echtzeitbetrieb, dass heißt, wenn die Ausführung von Maschinenprogrammen eine Abhängigkeit vom physikalischen Zeitpunkt der Erzeugung und Verwendung der Berechnungsergebnisse definiert. Solche Maschinenprogramme interagieren typischerweise mit „externen Prozessen“ im Rahmen einer Regelschleife (↑*control loop*). Für gewöhnlich handelt es sich dabei um technische (physikalische/chemische) Prozesse, die in technischen Anlagen oder Maschinen stattfinden. Aber auch der Mensch ist oft in solch einer

Regelschleife eingebunden und kann Schritte auslösen, die in Echtzeit zu bearbeiten sind. Beispiel ist eine manuell ausgelöste Reaktorschnellabschaltung (*scram*, für „verduften“), wenn das Bedienpersonal die Überschreitung kritischer Grenzwerte erkennt. Ein weiteres Beispiel ist die Verarbeitung von Datenströmen, hier insbesondere Audio- oder Videoströme (*streaming media*). Je nach Anwendungsfall gelten für solch eine Verarbeitung sehr unterschiedliche Kritikalitätsstufen: weich/fest für digitales Fernsehen (Einzelbildverlust ist unkritisch, wegen möglicher Qualitätseinbußen aber unerwünscht) und hart für autonomes Fahren (Einzelbildverlust ist kritisch, sollten dadurch Brems- und Lenkvorgänge zu spät ausgelöst werden).

Echtzeitbetrieb (en.) ↑*real-time mode*. Bezeichnung für eine ↑Betriebsart, die einen ↑Prozess im ↑Rechensystem in ↑Echtzeit stattfinden lässt. Ein oder mehrere Prozesse operieren unter wenigstens einer bestimmten ↑Echtzeitbedingung, die durch die jeweilige Umgebung des Rechensystems vorgegeben sind. Zeit stellt damit keine intrinsische Eigenschaft des Rechensystems dar, sondern ist durch die in der Umgebung gültigen Zeitskala definiert. Nach dieser Zeitskala haben sich bestimmte Abläufe innerhalb des Rechensystems zu richten.
Die Verarbeitung der zeitabhängigen Prozesse (↑*process control*) geschieht zeit- oder ereignisgesteuert. Bei zeitgesteuerter Verarbeitung finden die betreffenden Prozesse immer nur zu definierten Zeitpunkten statt (↑*time-triggered system*). Diese Zeitpunkte definiert zwar das Rechensystem (↑PIT), jedoch sind sie passend zu der in der Umgebung des Rechensystems existierenden Zeitskala gewählt. Die Zeitintervalle sind für gewöhnlich gleich lang (äquidistant) und fest. Die Ablaufplanung für die Prozesse geschieht rechnerunabhängig (*off-line*), sie liefert einen zur ↑Laufzeit statischen ↑Ablaufplan. Demgegenüber finden bei einer ereignisgesteuerten Verarbeitung die Prozesse entsprechend ihrer jeweiligen ↑Dringlichkeit zeitnah zu dem ihnen jeweils zugeordneten ↑Ereignis statt, die genauen Zeitpunkte für diese Prozesse sind jedoch unbekannt (↑*event-triggered system*). Die Ablaufplanung erfolgt mitlaufend (*on-line*) zu den jeweils einzuplanenden Prozessen, sie resultiert in einen dynamischen Ablaufplan, der zur Laufzeit variiert. Aus der Dringlichkeit eines Prozesses wird seine ↑Priorität abgeleitet, die ihm eine bestimmte Position auf der ↑Bereitliste einräumt und, in Abhängigkeit vom ↑Verdrängungsgrad, mehr oder weniger zügig und bestimmt (vorhersagbar) die ↑CPU zuteilt (↑*preemption*).
Die ereignisgesteuerte Verarbeitung von Prozessen bietet höhere Flexibilität, erschwert jedoch die ↑Vorhersagbarkeit der Abläufe im Rechensystem. Umgekehrt die zeitgesteuerte Verarbeitung, die aufgrund des statischen Ablaufplans kaum Flexibilität zeigt, dafür aber mit starker Vorhersagbarkeit dieser Abläufe aufwarten kann. Welche Verarbeitungsart zu bevorzugen oder gar gefordert ist, steht und fällt mit dem jeweiligen Anwendungsfall und dem jeweils verfügbaren Anwendungswissen. Zeitgesteuerte Verarbeitung erfordert *Vorabinformationen* zu den Prozessen: zu ihrer jeweiligen Dauer (↑WCET) wie auch zur jeweiligen Länge ihrer eventuellen ↑Periode, zu bestehenden Abhängigkeiten zwischen den Prozessen und über die von den Prozessen belegten ↑Betriebsmittel. Ist dieses *a priori* Wissen nicht verfügbar, bleibt nur die ereignisgesteuerte Verarbeitung der Prozesse. Ist dieses Wissen jedoch verfügbar, hat zeitgesteuerte Verarbeitung zudem den großen Vorteil, einen vergleichsweise schlanken Rechnerbetrieb zu ermöglichen — auch wenn vorhersagbares Verhalten keine Rolle spielt und daher ereignisgesteuerte Verarbeitung ebenfalls Option wäre.

Echtzeitprozess (en.) ↑*real-time process*. Ein ↑Prozess, für den wenigstens eine ↑Echtzeitbedingung gilt, der in ↑Echtzeit stattfindet oder stattfinden muss.

Echtzeitrechensystem (en.) ↑*real-time computing system*. Bezeichnung des in einem ↑Echtzeitsystem eingesetzten ↑Rechensystems zur ↑Prozessdatenverarbeitung.

Echtzeitsystem (en.) ↑*real-time system*. Einheit aus technischen Anlagen, Bauelementen, die eine gemeinsame Funktion haben (Duden) und in ↑Echtzeit betrieben werden beziehungsweise operieren müssen. Ein solcher Komplex dient für gewöhnlich der Implementierung eines oder mehrerer ↑Regelkreise, und zwar mechanisch, elektronisch oder elektro-mechanisch umgesetzt. Für eine auf Elektronik zurückgreifende Realisierung bildet gegebenenfalls ein ↑Echtzeitrechensystem eine der zentralen Komponenten.

Echtzeituhr (en.) ↑*real-time clock*. Bezeichnung für einen speziellen ↑Zeitgeber, mit dem die *physikalische Zeit* gemessen wird. Häufig wird diese Art von Zeitgeber über eine Ersatzstromquelle (z.B. Batterie) mit Energie versorgt, um auch dann noch Zeit erfassen zu können, wenn der ↑Rechner ausgeschaltet oder sein ↑Versagen festgestellt wurde. Typischerweise arbeitet der Oszillator dieser Uhr mit einer Frequenz von 32 768 kHz, das heißt, er generiert genau 2^{15} Takte pro Sekunde, was einem Zeitabstand von 30,517578125 μs entspricht.
Für gewöhnlich kann diese Basisfrequenz durch einen im Zeitgeber integrierten Teiler (*divider*) verändert werden. Dadurch lässt sich die Frequenz, mit der eine ↑Unterbrechungsanforderung von dem Zeitgeber generiert wird auch auf kleinere Werte einstellen. Standardmäßig ist die Unterbrechungsfrequenz 1024 Hz, jedoch kann die Basisfrequenz 15-fach geteilt werden hinunter bis auf 2 Hz. Aus praktischen Gründen erlauben die meisten ↑Rechensysteme allerdings nur eine Rate von höchstens 8192 Hz, also eine zyklische ↑Unterbrechung nach jeweils 122,0703125 μs.
Der Zeitgeber eignet sich besonders zur Auslösung regelmäßig zu wiederholender ↑Aufgaben, wovon ↑mitlaufende Planung eine der wichtigsten ist. Hier sind die Verfahren ↑RR und ↑HRRN hervorzuheben, bei denen das ↑Ereignis zur ↑Umplanung für gewöhnlich in jeweils unveränderten Intervallen hervorgerufen wird. Demgegenüber ist dieser Zeitgeber weniger geeignet für Verfahren wie ↑VRR oder ↑SRTF, wo deblockierte ↑Prozesse die ↑CPU für den jeweils noch verbleibenden Rest der sonst festen ↑Zeitscheibe zugeteilt bekommen können. Ebenso ↑MLFQ, da hier das zur Zeitscheibe korrespondierende Zeitintervall variiert, das heißt, mit den einzelnen Stufen von oben nach unten in dem System an Länge zunimmt. Für derartige Verfahren bildet ein flexibel einstellbarer Zeitgeber (↑PIT) die bessere Grundlage.

EDF Abkürzung für (en.) *earliest deadline first*. Ein bestimmter ↑Einplanungsalgorithmus in einem ↑Betriebssystem. Zeitbasiertes Verfahren, das dem ↑Prozess die höchste ↑Priorität gibt, dessen Termin als nächster ansteht. Für ↑Echtzeitbetrieb (↑*event-triggered system*) bedeutet dies insbesondere auch ↑Verdrängung, vorzugsweise mit einem ↑Verdrängungsgrad von 100 % (sog. volle Verdrängung, (en.) *full preemption*) und konstanter ↑Latenz.

edge-triggered interrupt (dt.) flankengesteuerte ↑Unterbrechung, ↑Flankensteuerung.

EDV Abkürzung für *elektronische Datenverarbeitung*. Sammelbegriff für den Umgang mit ↑Daten in einem ↑Rechensystem, um dadurch Informationen zu gewinnen und darzustellen und die gewonnenen Informationen für die Erzeugung weiterer Daten zu nutzen. Die zu verarbeitenden Daten liegen rechnerunabhängig (*off-line*) oder mitlaufend (*on-line*) zu einem ↑Prozess in einem ↑Speicher im Rechensystem vor.

EDVAC Abkürzung für (en.) *Electronic Discrete Variable Automatic Computer* (1949). Erster ↑Rechner, mit dem das von John von Neumann vorgeschlagene Prinzip einer universellen ↑Rechnerarchitektur verwirklicht wurde. Die von dem (auf Röhren- und Diodentechnik basierenden) Rechner auszuführenden Anweisungen waren demzufolge vorgegeben durch ein ↑Programm, das erstmalig binär kodiert war und zur Ausführung intern im ↑Hauptspeicher lag. Zur ↑Ein-/Ausgabe von ↑Daten standen Magnetbandgerät, ↑Kartenstanzer, ↑Kartenleser und ↑Fernschreiber (↑TTY) zur Verfügung. Programmiert wurde der Rechner direkt in ↑Maschinenkode.

EEPROM Abkürzung für (en.) *electrically erasable PROM*, (dt.) elektrisch löschbarer ↑PROM.

Eigenvariable (en.) ↑*own variable*. Bezeichnung einer ↑Variablen, deren Wert beim Verlassen des Grundblocks, in dem sie deklariert wurde, namentlich unbekannt wird, aber erhalten bleibt und beim Wiedereintritt in diesen Block automatisch zur Verfügung steht. Geht auf ein mit ↑Algol 60 eingeführtes Konzept zurück, das solche Variablen durch den Zusatz `own` bei der Deklaration auszeichnet. In ↑C wird gleiches durch `static` als Zusatz bei der Deklaration der lokalen Variablen einer Funktion erreicht.

Ein-/Ausgabe (en.) ↑*input/output*; Abkürzung ↑E/A. Bezeichnung einerseits für den Vorgang zur Kommunikation der innerhalb eines ↑Rechensystems stattfindenden ↑Prozesse mit ihrer

Außenwelt und andererseits für die Gesamtheit von ↑Daten, Informationen, die einem Prozess eingegeben und von ihm verarbeitet werden (*input*) beziehungsweise das Arbeitsergebnis eines Prozesses (*output*) bilden.

Ein-/Ausgaberegister (en.) ↑*input/output register*; Abkürzung ↑E/A-Register. Behältnis in einem ↑Peripheriegerät um ↑Daten ein-/auszugeben und die ↑Ein-/Ausgabe zu steuern und zu überwachen. Anzahl und Bedeutung dieser ↑Register sind gerätespezifisch. Typischerweise wird (logisch) unterschieden zwischen Kontroll-, Status- und Datenregister. Diese Register sind zugänglich entweder über eine gewöhnliche ↑Adresse (↑*memory-mapped* I/O) oder über einen speziellen Anschluss (↑*port-mapped* I/O).

Ein-/Ausgabestoß (en.) ↑I/O *burst*; Abkürzung ↑E/A-Stoß. Bezeichnung für eine ↑Häufung von Aktivität, die in engem Bezug mit der ↑Peripherie steht und vor allem der ↑Ein-/Ausgabe von ↑Daten dient. Ein solches Aktivitätsbündel umfasst sowohl die zur Durchführung der Ein-/Ausgabe notwendigen ↑Aktionen der ↑CPU als auch die daraufhin in der Peripherie ablaufenden Vorgänge. Die seitens der CPU dazu stattfindenden Aktionen beruhen auf ↑Maschinenbefehle, mit denen die ↑Steuerung und Überwachung der in der Peripherie stattfindenden Ein-/Ausgabeoperationen bewerkstelligt wird.
In dieser auf Ein-/Ausgabe fokussierten Phase führt die Peripherie eigenständig eine oder mehrere Ein-/Ausgabeoperationen für einen ↑Prozess und damit unabhängig von der CPU durch. Diese Operationen sind logisch, aber nicht zwingend physisch verknüpft mit dem Prozess, der sie ausgelöst hat. Der Prozess kann während der Zeit einen beliebigen ↑Prozesszustand haben, das heißt, er ist laufend, bereit oder blockiert. Im letzteren Fall kann es sein, dass der Prozess die Beendigung einer Ein-/Ausgabeoperation erwartet (↑passives Warten) und dadurch physisch mit der Ein-/Ausgabe gekoppelt ist. Eine solche Kopplung kann allerdings auch bestehen, wenn der Prozess laufend ist (↑aktives Warten). In den anderen Fällen findet der Prozess losgelöst von der oder den laufenden Ein-/Ausgabeoperationen statt, die er gegebenenfalls abgesetzt hat, er ist aber logisch mit seiner Ein-/Ausgabe gekoppelt. Wartet ein Prozess nicht die Beendigung seiner Ein-/Ausgabeoperation(en) ab, kann er weitere an dasselbe oder ein anderes ↑Peripheriegerät abgeben. All diese Operationen geschehen nebenläufig zum ↑Rechenstoß des Prozesses.

Einbenutzerbetrieb (en.) ↑*single-user mode*. Variante von ↑Dialogbetrieb, bei der das ↑Betriebssystem zu einem Zeitpunkt höchstens einem Teilnehmer den Rechenbetrieb ermöglicht (Gegenteil von ↑Mehrbenutzerbetrieb). Allerdings ist ↑Mehrprogrammbetrieb damit nicht ausgeschlossen: je nach Auslegung des Betriebssystems kann ein Teilnehmer sehr wohl in die Lage versetzt werden, während der ↑Sitzung mehr als ein ↑Programm zur Ausführung zu bringen (↑Teilnehmerbetrieb, ↑Teilhaberbetrieb).

Einfriedung (en.) ↑*boundary fence*. Bezeichnung für einen ↑Speicherschutz durch ↑Grenzregister. Generiert ein ↑Prozess eine ↑Adresse, die außerhalb der durch „seine“ Grenzregister festgelegten Zone in einem ↑Adressraum liegt, wird die diesbezügliche ↑Aktion eines lesenden oder schreibenden Zugriffs abgefangen (↑*trap*). Für gewöhnlich speichert jedes der Grenzregister eine ↑reale Adresse. Bevor die von dem Prozess gebildete effektive (reale) Adresse zum ↑Adressbus geht, wird diese auf Gültigkeit hin wie folgt überprüft:

$$BR_{lwb} \leq \textit{effective address} \leq BR_{upb},$$

mit BR (↑*bounds register*) für eins der beiden Grenzregister (*lower* bzw. *upper bound*: *lwb*/*upb*). Liegt die effektive Adresse außerhalb der Grenzen, erhebt die Hardware eine ↑Ausnahme und das ↑Betriebssystem bricht den Prozess in aller Regel ab (↑*segmentation fault*). Ist das Betriebssystem selbst eingegrenzt und generiert ein Prozess, der (als Folge von einem ↑Systemaufruf, einer ↑Unterbrechung oder gar ↑Schutzverletzung) im Betriebssystem stattfindet, eine Adresse, die außerhalb der Betriebssystemgrenzen liegt, wird ↑Panik ausgelöst: in dem Fall ist von einem gravierenden Programmierfehler im Betriebssystem auszugehen, der den weiteren gesicherten ↑Programmablauf unmöglich macht.

Diese Form von Speicherschutz ist typisch für eine ↑MPU. Im ↑Prozesskontrollblock einer ↑Prozessinkarnation werden die unteren und oberen Grenzadressen von dem entsprechenden ↑Prozessadressraum als Attribute gespeichert und beim ↑Prozesswechsel in die jeweiligen Grenzregister geschrieben. Da dieser Schreibzugriff eine privilegierte Operation ist und daher im Betriebssystem erfolgt (↑*privileged mode*), müssen im Falle eines selbst eingegrenzten Betriebssystems für den System- und Benutzermodus jeweils eigene Grenzregister vorhanden sein (↑Arbeitsmodus). Beim Prozesswechsel werden die Grenzadressen in die dem Benutzermodus zugeordneten Grenzregister geschrieben, die aber erst nach Verlassen des Systemmodus zur Wirkung kommen. Grundsätzlich werden mit jedem Wechsel vom Benutzer- zum Systemmodus und umgekehrt die dem jeweiligen Modus zugeordneten Grenzregister im ↑Prozessor (bzw. in der MPU) wirksam. Ist das Betriebssystem dagegen nicht eingegrenzt, sind die Grenzregister nur einfach ausgelegt. In dem Fall ist die Überprüfung von Grenzen (*bounds check*) im Systemmodus entweder wirkungslos oder abgeschaltet, die Grenzregister kommen dann nur im Benutzermodus zur Wirkung.

Eingabetaste (en.) ↑*enter key.* Taste auf einer Rechnertastatur zur Bestätigung einer Eingabeaufforderung. Bewirkt beim zeilenorientierten Betrieb einen kombinierten ↑Wagenrücklauf und ↑Zeilenvorschub.

Einlastung (en.) ↑*dispatching.* Vorgang der Belegung einer Verarbeitungseinheit mit einem Auftrag. Die Verarbeitungseinheit ist ein ↑wiederverwendbares Betriebsmittel bestimmter Art, das nach Gebrauch für einen weiteren Auftrag entsprechender Art weiterhin verwendet werden kann. Typisches Beispiel für einen Auftrag ist eine ↑Aufgabe, für deren Erledigung ein ↑Prozess stattfinden und dazu wiederum ein ↑Prozessor zugewiesen, das heißt, unter Last gesetzt werden muss. Nach Gebrauch des Prozessors ist dieser frei für einen weiteren Prozess (↑Prozesswechsel). Anderes Beispiel ist Ein-/Ausgabe, bei der ein ↑Peripheriegerät durch einen Prozess den Auftrag zur Lieferung von Eingabe- oder Annahme von Ausgabedaten erhält. Nach Gebrauch des Geräts ist dieses frei für einen weiteren Ein-/Ausgabeauftrag.

Einlastungslatenz (en.) ↑*dispatching latency.* Bezeichnung für die ↑Latenzzeit bis zur erfolgten ↑Einlastung von einem ↑Prozessor, und zwar relativ zum Zeitpunkt der Feststellung des diesbezüglichen Auftrags zur Ausführung einer bestimmten ↑Aufgabe. Beschreibt der Auftrag einen ↑Prozess und handelt es sich bei dem Prozessor um die ↑CPU, dann entspricht diese Latenzzeit dem Interval, um den Prozess vom Zustand bereit in den Zustand laufend zu überführen (s. a. ↑Prozesszustand), das heißt, einen ↑Prozesswechsel durchzuführen.
Ein sowohl für ↑vorlaufende Planung als auch ↑mitlaufende Planung anfallendes Zeitintervall, das zum einen durch die ↑Ausführungszeit des ↑Umschalters bestimmt ist und zum anderen jede zu seiner Aktivierung nötige ↑Aktionsfolge einschließt. Unabwendbare ↑Gemeinkosten, sobald ein Prozessor als Folge der ↑Einplanung mehr als einen Prozess zur Verarbeitung zugeteilt bekommen hat und demzufolge im ↑Multiplexverfahren betrieben werden muss.
Für gewöhnlich schlagen diese Gemeinkosten nur zu Buche, wenn der laufende Prozess in den Zustand blockiert wechselt oder die Einplanung feststellt, dass dieser Prozess von der CPU verdrängt werden muss (↑*preemption*). Jedoch nicht jede ↑präemptive Planung hat den Prozessorentzug für einen Prozess zwingend zur Folge: entscheidend ist, dass die ↑Dringlichkeit des einzuplanenden oder eingeplanten Prozesses höher sein muss als die des laufenden Prozesses. In dem Fall gehen diese Gemeinkosten einher auch mit jedem Zustandsübergang eines Prozesses von laufend nach bereit. Dies trifft ebenfalls auf das ↑Rundlaufverfahren zu (z.B. ↑RR), durch den ein bestehender ↑Ablaufplan im ↑Zeitteilverfahren verarbeitet wird.

Einplanung (en.) ↑*scheduling.* Eingeplantsein, etwas in der ↑Planung berücksichtigen (Duden). Einen ↑Auftrag zur Verarbeitung durch einen ↑Prozessor zeitlich einplanen (*schedule*), einen Zeitplan für die Auftragsverarbeitung festlegen. Das Moment, der ausschlaggebende Umstand für einen solchen Vorgang entspricht einem bestimmten ↑Ereignis, das die Notwendigkeit der Auftragserteilung bewirkt.
Mit diesem Ereignis wird die ↑Bereitzeit der mit dem Auftrag verbundenen ↑Aufgabe festgelegt. Die wirkliche ↑Startzeit dieser Aufgabe liegt jedoch später, sie hängt ab von der

Zeitspanne bis zur Entnahme des entsprechenden Auftrags aus der ↑Warteschlange für den Prozessor. Diese Zeitspanne trägt zur ↑Wartezeit des Auftrags bei und beeinflusst damit die ↑Latenz bis zum wirklichen Ausführungsbeginn der betreffenden Aufgabe.
Ein anderer wesentlicher Faktor zur Wartezeit bilden Ereignisse und daraus resultierende Maßnahmen, die zur ↑Unterbrechung der Entleerung oder, möglicherweise ursächlich eben in einer solchen Unterbrechung selbst begründet, vordringlichen Befüllung der Warteschlange führen. Sind diese Ereignisse nicht kontrollierbar, kann die Wartezeit eines Auftrags nicht begrenzt und damit für die betreffende Aufgabe auch keine Aussage zur Start- und effektiven ↑Endzeit, ganz geschweige zur ↑Ausführungszeit, getroffen werden. Dieser Aspekt ist besonders problematisch, wenn die Aufgabe mit einer ↑Frist verknüpft ist, das heißt, ihre Bearbeitung einer ↑Echtzeitbedingung unterworfen ist.

Einplanungsalgorithmus (en.) ↑*scheduling algorithm.* Lösungs- und Bearbeitungsschema, Handlungsvorschrift zur ↑Planung der Vergabe (↑*dispatching*) von einem ↑Auftrag an einen ↑Prozessor. Ist charakterisiert durch die Reihenfolge von Aufträgen in einer ↑Warteschlange und die Bedingungen, unter denen die Aufträge dort eingereiht werden. Die Bewertungsgrundlage (↑*scheduling criteria*) für diese Vorschrift unterscheidet dabei grob zwischen anwendungs- und systemorientierten Kriterien, das heißt, dem in einer Anwendung wahrnehmbaren Systemverhalten einerseits und der effektiven und effizienten Auslastung der ↑Betriebsmittel andererseits.
Beiden Orientierungen mit ein und demselben Verfahren gerecht zu werden, ist in aller Regel nicht möglich. Ein mehrstufiger Ansatz kann Abhilfe schaffen, wobei den einzelnen Stufen dann bestimmten Kriterien zugeordnet sind. In solch einem Szenario erfolgt dann jedoch immer eine Schwerpunktsetzung, die sich durch die Reihenfolge ergibt, in der die einzelnen Ebenen durchlaufen werden und dadurch ein Kriterium ein anderes dominiert.

Einplanungskriterium (en.) ↑*scheduling criteria.* Bewertungsgrundlage für einen ↑Einplanungsalgorithmus. Grob unterschieden wird dabei zwischen anwendungs- und systemorientierten Kriterien, das heißt, dem in einer Anwendung wahrnehmbaren Systemverhalten einerseits und der effektiven und effizienten Auslastung der ↑Betriebsmittel andererseits. Charakteristische Merkmale anwendungsorientierter Kriterien in Bezug auf das ↑Betriebssystem sind ↑Antwortzeit, ↑Durchlaufzeit, ↑Termineinhaltung und ↑Vorhersagbarkeit. Demgegenüber stehen wünschenswerte Merkmale systemorientierter Kriterien wie ↑Durchsatz, ↑Prozessorauslastung, ↑Gerechtigkeit, ↑Dringlichkeit und ↑Lastausgleich. In der Durchsetzung dieser Kriterien ist ↑Proportionalität ein zusätzlicher Aspekt, der die Verhältnismäßigkeit der mit einem Einplanungsalgorithmus möglichen Optimierungen und den sich daraus ergebenden Konsequenzen für den Entwurf und die Implementierung eines Betriebssystems in den Vordergrund stellt.

Einplanungslatenz (en.) ↑*scheduling latency.* Bezeichnung für die ↑Latenzzeit bis zur erfolgten ↑Einplanung von einem ↑Auftrag für einen ↑Prozessor. Beschreibt der Auftrag einen ↑Prozess und handelt es sich bei dem Prozessor um die ↑CPU, dann entspricht diese Latenzzeit dem Interval, um den Prozess vom Zustand <u>blockiert</u> in den Zustand <u>bereit</u> zu überführen (s. a. ↑Prozesszustand), das heißt, auf die ↑Bereitliste zu platzieren.
Ein nur für ↑mitlaufende Planung anfallendes Zeitintervall, das zum einen durch die ↑Ausführungszeit des ↑Planers bestimmt ist und zum anderen jede zu seiner Aktivierung nötige ↑Aktionsfolge einschließt. Ersteres ist maßgeblich durch den ↑Einplanungsalgorithmus und seiner konkreten Implementierung vorgegeben, wohingegen letzteres insbesondere davon abhängt, ob die ↑Einplanung nebenläufig, das heißt, pseudo- oder echt parallel zu dem gegenwärtig auf dem betreffenden Prozessor stattfindenden Prozess erfolgen kann.
Ist für ↑Synchronisation zu sorgen, wenn nämlich nebenläufige Planerausführung nicht in Frage kommt, fallen ↑Gemeinkosten nichtfunktionaler Art an (↑*ambient noise*). Verbietet der Planer beispielsweise den ↑Wiedereintritt, weil er etwa als ↑kritischer Abschnitt implementiert ist, und läuft bereits ein Planungsvorgang, verzögert sich die erneute Aktivierung des Planers, bis der laufende Planungsvorgang zum Abschluss kommt, das heißt, der kriti-

sche Abschnitt verlassen und damit freigegeben wird. Ohne ↑Vorwissen über die maximale Anzahl von Prozessen, die gleichzeitig im ↑Wettstreit um einen durch (in diesem Beispiel mittels) ↑blockierende Synchronisation abgesicherten Planer stehen können, bleibt diese Verzögerung unbestimmt. Dieses Vorwissen kann nur aus dem jeweiligen ↑Anwendungsfall, den das ↑Betriebssystem unterstützen soll, extrahiert werden. Kann dieses Wissen nicht zusammengestellt und für das Betriebssystem genutzt werden, ist die zu erwartende maximale Verzögerung nur noch bestimmbar, wenn der Planer auf ↑nichtblockierende Synchronisation mit ↑Wartefreiheit setzt.

Einprogrammbetrieb (en.) ↑*uniprogramming*. Bezeichnung für eine ↑Betriebsart, bei der zu einem Zeitpunkt nur ein ↑Maschinenprogramm im ↑Arbeitsspeicher zur Ausführung bereit steht. Die Inbetriebnahme des Maschinenprogramms geschieht von Hand (↑manueller Rechnerbetrieb) oder programmgesteuert (↑automatisierter Rechnerbetrieb), darüber hinaus kann es als ↑sequentielles Programm oder ↑nichtsequentielles Programm ausgelegt sein und damit sequentiell oder (echt/pseudo) parallel ausgeführt werden. Zur Entlastung der ↑CPU von Ein-/Ausgabe ist ↑abgesetzter Betrieb möglich, sofern das ↑Rechensystem eine entsprechende Hardwarekonfiguration aufweist. Ganz allgemein bietet ansonsten noch ↑überlappte Ein-/Ausgabe die Option, ↑Rechenstoß und ↑Ein-/Ausgabestoß des einen in Ausführung befindlichen Maschinenprogramms parallel stattfinden zu lassen und dadurch grundsätzlich die ↑Durchlaufzeit sowie Verweildauer im Arbeitsspeicher zu verringern. Eine weitere, den ↑Durchsatz des Rechensystems steigernde — jedoch ↑Speicher kostende und ↑Hintergrundrauschen erzeugende — Maßnahme besteht in einer Verkürzung der Wechselzeiten zwischen aufeinanderfolgenden Maschinenprogrammen durch die Zwischenpufferung der jeweils als nächstes zu bearbeitenden ↑Aufgabe (↑*single-stream batch monitor*).

Einschaltrückstellung (en.) ↑*power-on reset*. Bezeichnung für eine (schaltungstechnische) Vorrichtung oder den Vorgang, um einen elektronischen ↑Rechner in den Anfangszustand zurückzusetzen. Nach dem Anlegen der Versorgungsspannung, aber erst mit Erreichen eines bestimmten Nennwerts dieser Spannung, sorgt die Maßnahme zunächst für den definierten Zustand der elektronischen Bausteine. Anschließend wird von der Hardware eine ↑Ausnahme erhoben, um (in ↑ROM oder ↑EPROM) permanent residenter Software die weitere Initialisierung des Systems zu übertragen (*reset*: ↑*trap*). Im ↑Hauptspeicher flüchtige Software wird, soweit erforderlich, anschließend durch ↑Urladen ins System gebracht und gestartet.

einseitige Synchronisation siehe ↑unilaterale Synchronisation.

einstufiger Speicher (en.) ↑*single-level store*. Bezeichnung für einen ↑Speicher, bei dem die tatsächliche Lage seiner ↑Seiten, das heißt, ob diese aktuell im ↑Vordergrundspeicher oder im ↑Hintergrundspeicher vorrätig sind, für gewöhnlich unbedeutend ist für den jeweils stattfindenden ↑Prozess. Typischerweise ist ↑virtueller Speicher von dieser Art.

Eintrittsinvarianz (en.) ↑*reentrancy*. Charakteristik eines bestimmten Abschnitts in einem ↑Programm, zur ↑Laufzeit den ↑Wiedereintritt gefahrlos zu erlauben und damit ↑Ablaufinvarianz sicherzustellen.

Eintrittskonsistenz (en.) ↑*entry consistency*. Modell der ↑Speicherkonsistenz, für das ein ↑kritischer Abschnitt den Ausgangspunkt bildet. Variante der ↑Freigabekonsistenz, indem beim Eintritt in den betreffenden Abschnitt nur ↑Synchronisation der ↑Prozesse in Bezug auf die Beendigung von Schreiboperationen auf einzeln ausgewiesene ↑Speicherzellen erzielt wird. Spezifischer als Freigabekonsistenz, schwächer als ↑schwache Konsistenz.

Eintrittsprotokoll (en.) ↑*entry protocol*. Konvention zur ↑Synchronisierung ↑gleichzeitiger Prozesse, um ein ↑nichtsequentielles Programm partiell nur strickt sequentiell auszuführen. Typischer Grund für die ↑Sequentialisierung dieser ↑Prozesse ist ein ↑kritischer Abschnitt. Hierzu muss das Protokoll sicherstellen, dass nach Verlauf der darin beschriebenen Schritte am Ende nur noch ein einziger Prozess den fraglichen Abschnitt passieren kann. Mit erfolgreichem

Protokolldurchlauf hat der Prozess den kritischen Abschnitt „erworben“ (*aqcuire*).
Das dazu korrespondierende ↑Austrittsprotokoll regelt den weiteren Ablauf beim Verlassen des betreffenden kritischen Abschnitts. Ist ↑aktives Warten für die Sequentialisierung vorgesehen, entriegelt (*unlock*) der den kritischen Abschnitt verlassende Prozess lediglich eben diesen Abschnitt und zeigt dies durch ein ↑Ereignis an. Daraufhin wird beim Abschnittseintritt der erste Prozess, der dieses Ereignis wahrnimmt und es daraufhin schafft, den Abschnitt erneut zu verriegeln (*lock*), fortfahren können. In einer ↑Konkurrenzsituation darf dies genau nur einem der Konkurrenten gelingen. Ist ↑passives Warten vorgesehen, muss der den kritischen Abschnitt verlassende Prozess zusätzlich den ↑Planer anweisen, einen der dieses Ereignis gegebenenfalls erwartenden Prozesse zu deblockieren. Um dem Austrittsprotokoll die zu deblockierenden Prozesse zu vermitteln, bietet sich eine ↑Warteschlange der vergebens einen Eintrittsversuch geleisteten Prozesse an. In diese Warteschlange reiht sich eintrittsseitig der erfolglose Prozess selbst ein, um dann zu gegebener Zeit austrittsseitig von dem den kritischen Abschnitt verlassenden Prozess daraus wieder entfernt zu werden.
Die Anweisungen einerseits (Austritt) zum Deblockieren und andererseits (Eintritt) zum Verriegeln bewirken verschiedene Prozesse. Zwischen diesen beiden können weitere Prozesse eintreffen und mit dem deblockierten Prozess um den Eintritt in den kritischen Abschnitt konkurrieren. Daher muss sich ein durch das Austrittsprotokoll deblockierter Prozess für gewöhnlich erneut um den Eintritt bewerben. Nur wenn die ↑Unteilbarkeit der ↑Aktionsfolge vom deblockierenden zum verriegelnden Prozess sichergestellt ist, kann die erneute Bewerbung entfallen. In beiden Fällen (aktives/passives Warten) ist jeder dieser Konkurrenten in dem Moment ein ↑gekoppelter Prozess.
Beim Austritt ist keine Konkurrenzsituation zwischen gleichzeitigen Prozessen desselben kritischen Abschnitts zu befürchten, im Gegensatz zum Eintritt. Wohl aber kann der Prozess, der gerade den kritischen Abschnitt verlässt, in Konkurrenz stattfinden zu einem oder mehreren Prozessen, die den kritischen Abschnitt betreten wollen. So bilden in Bezug auf die Absicherung nur eines kritischen Abschnitts die Implementierungen beider Protokolle selbst ein mit bis zu $N + 1$ Prozessen gleichzeitig ausgeführtes nichtsequentielles Programm, N Prozesse eintrittsseitig und 1 Prozess austrittsseitig. Entsprechend sind darin Vorkehrungen zur ↑Koordinierung der ↑Aktionen dieser Prozesse zutreffen.

Einzeladressraum (en.) ↑*single-address space*. Charakteristische Eigenschaft eines logischen/virtuellen ↑Adressraums, der implizit von mehreren ↑Prozessen geteilt wird. Eine ↑Adressraumisolation für den einzelnen ↑Prozessadressraum gibt es nicht, alle ↑Adressen sind eindeutig für jeden Prozess in diesem einen Adressraum (Gegensatz zum ↑Mehradressraum).
In einem ↑Rechensystem mit dieser Eigenschaft wird ↑Schutz dadurch sichergestellt, dass kein Prozess eine ihm fremde Adresse (innerhalb einer bestimmten Zeitspanne) in Erfahrung bringen kann. Die Annahme dabei ist (a) ein allen Prozessen gemeinsamer, einziger und riesengroßer ↑logischer Adressraum und (b) dass jede vom ↑Betriebssystem zu vergebene ↑logische Adresse als Funktion der ↑Platzierungsstrategie randomisiert wird. Damit kann keine dieser Adressen von einem Prozess erraten werden und bloßes Ausprobieren einer Adresse ist wegen der Adressraumgröße impraktikabel.
Dieser Ansatz ist gangbar bei einer ↑Adressbreite ab 48 Bits beziehungsweise einen 2^{48} verschiedene logische Adressen umfassenden Adressraum: mit einer ↑Zugriffszeit von 1 ns pro ↑Byte würde ein kompletter Lauf über den gesamten Adressraum etwa 78 Stunden dauern, was für eine ↑Betriebsart mit kurzlebigen (interaktiven) Prozessen ausreichend Sicherheit bieten kann. Auf Grundlage heutiger (2025) 64-Bit-Technologie würde ein solcher Lauf etwa 585 Jahre dauern und damit allgemein vor Brachialgewalt (*brute force*) schützen, sollte ein Prozess fremde Bestände von ↑Text und ↑Daten zerstören wollen.
So ist Schutz vor unautorisierten Zugriffen gegeben, ohne jedoch erlaubte Zugriffe (z.B. nur lesen) weiter qualifizieren und unerlaubte Zugriffe (z.B. alle, außer lesen) abwehren zu können. Für letzteres ist eine Adresse zusätzlich als ↑Befähigung auszulegen. Bei solch einem Ansatz wird eine ↑MMU schließlich nur zur ↑Adressabbildung (d.h., ↑Verlagerung zur ↑Laufzeit) benötigt, nicht aber zur Adressraumisolation (im Gegensatz zum Mehradressraum).

Einzelanforderung (en.) ↑*on-demand paging.*. VL 12.3

Einzelruf (en.) ↑*unicast*. Bezeichnung für einen Ruf, der nur an genau eine ↑Entität innerhalb eines bestimmten ↑Bezugssystems geht. Gegenteil zum ↑Gruppenruf.

Einzelstromstapelmonitor (en.) ↑*single-stream batch monitor*. Bezeichnung für einen ↑Stapelmonitor, der die gestapelten Aufträge als einzelnen Verarbeitungsstrom ausführt. Grundlage dafür bildet ein ↑Maschinenprogramm, wovon zu einem Zeitpunkt immer nur eins im ↑Hauptspeicher liegend herrscht (↑*uniprogramming*) und zur ↑Laufzeit eine bestimmte Einzelarbeit (↑*job*) leistet. Gegebenenfalls liest der Stapelmonitor im Hintergrund der Ausführung dieses Maschinenprogramms bereits das nächste Maschinenprogramm in Folge ein (↑*overlapped I/O*) und puffert es zwischen, um den Maschinenprogrammwechsel frei von ↑Durchsatz mindernder ↑Wartezeit bewerkstelligen zu können. Sobald das Maschinenprogramm mangels ↑Betriebsmittel nicht weiter ausgeführt werden kann, stoppt der Stapelmonitor seinen Betrieb und erwartet die Beendigung von einem gegebenenfalls noch in der ↑Peripherie nebenläufigen ↑Ein-/Ausgabestoß. Gibt es einen solchen Stoß und wird er fertig, können die dadurch verfügbar werdenden Betriebsmittel zur Fortsetzung der Maschinenprogrammausführung führen. Gibt es keinen solchen Stoß oder werden keine Betriebsmittel durch seine Beendigung freigestellt, kann dies ↑Panik auslösen.

Elementaroperation (en.) ↑*elementary operation*. Primitive einer (realen/virtuellen) Maschine; grundlegende, wesentliche Operation in Bezug auf eine bestimmte Ebene der Abstraktion. Eine ↑Aktion, die auf dieser Ebene in einem Schritt (ungeteilt, atomar) stattzufinden scheint.

Exkurs Zählen ist eine Operation, die vor dem Hintergrund ↑gleichzeitiger Prozesse, die über eine gemeinsame ↑Variable miteinander gekoppelt sind, kritisch ist und leicht ein falsches Berechnungsergebnis verursachen kann. In solch einem Fall ist ↑Synchronisation zu gewährleisten. Grundlage bildet ein ↑nichtsequentielles Programm, in dem das Zählen selbst eben eine ↑Aktionsfolge darstellt und letztlich als ↑kritischer Abschnitt in Erscheinung tritt. Für gewöhnlich ist zur Absicherung eines solchen Abschnitts ↑wechselseitiger Ausschluss der betreffenden Prozesse sicherzustellen. Die Umsetzung dieser Maßnahme kann grundsätzlich auf zwei verschiedenen Ebenen erfolgen, nämlich auf der ↑Maschinenprogrammebene oder auf der ↑Befehlssatzebene. Zur Synchronisation werden somit entweder Abstraktionen des ↑Betriebssystems genutzt, was ↑Systemaufrufe nach sich zieht oder es kommen spezielle ↑Maschinenbefehle der ↑CPU zum Einsatz.
Angenommen es wird eine elementare Zähloperation benötigt, die eine Ganzzahlvariable um Eins erhöht und den vor dem Hochzählen gültigen Variablenwert als Funktionsergebnis liefert (*fetch and increment*, `FAI`). Die Signatur dieser Operation sei wie folgt spezifiziert:

```
int FAI(aint_t *);                                  /* fetch and increment */
```

Der formale Parameter ist ein ↑Zeiger auf eine Variable, deren ↑Datentyp eine unteilbare Ganzzahl (*atomic integer*) definiert. Das Ergebnis der Funktion ist eine typkonforme ganze Zahl. Eine mögliche Implementierung dieser Funktion durch Benutzung eines Betriebssystems, um dem gleichzeitigen Hochzählen ein und derselben Variable vorzubeugen, skizziert nachfolgendes Beispiel:

```
typedef struct aint {
    semaphore_t mutex;                          /* initial: 1 (unlocked) */
    int value;                                          /* actual number */
} aint_t;

inline int FAI(aint_t *this) {            /* OS-based fetch and increment */
    P(&this->mutex);                            /* enter critical section */
    int data = this->value++;                        /* increment counter */
    V(&this->mutex);                            /* leave critical section */

    return data;             /* deliver fetched (i.e., old) counter value */
}
```

Zum Schutz des kritischen Abschnitts dient ein ↑binärer Semaphor (`mutex`), dessen Operationen ↑P und ↑V als „Synchronisationsklammern“ passend platziert wurden. Der Initialwert des Semaphors bringt zum Ausdruck, dass sich zu einem Zeitpunkt höchstens ein Prozess im kritischen Abschnitt befinden kann — vorausgesetzt V ruft immer der Prozess auf, der zuvor P aufgerufen hat. Darüberhinaus muss jedes ↑Exemplar dieses Datentyps korrekt initialisiert werden, damit es zu keiner ↑Verklemmung der Prozesse kommt:

```
aint_t aint = { {1} };                    /* unlocked (1) atomic integer */
```

In funktionaler Hinsicht wird mit einer solchen Lösung sehr wohl die erforderliche Synchronisation erreicht. Jedoch stellt sich die grundsätzliche Frage, ob diese Implementierung auch in nichtfunktionaler Hinsicht für die gegebene Problemstellung, nämlich einfach nur einen Zähler um Eins weiterzuschalten, geeignet ist. Um ansatzweise eine Idee von den zu erwartenden ↑Gemeinkosten zu bekommen, ist ein Blick auf die vom ↑Kompilierer (↑GCC) generierten Maschinenbefehle (↑x86) sinnvoll (`gcc`-Schalter `-S`). Allein nur für den Anteil im ↑Maschinenprogramm fallen brutto wenigstens 8 Maschinenbefehle an, wobei nur die Aufrufe von P und V, nicht aber deren Implementierungen berücksichtigt wurden. Für die eigentliche Zähloperation wurden netto lediglich 3 Maschinenbefehle generiert, der 5 Maschinenbefehle umfassende Mehraufwand auf dieser Ebene kommt durch die Parameterversorgung sowie ↑Unterprogrammaufrufe zustande. Die Implementierungen von P und V bringen leicht hunderte weiterer Maschinenbefehle im Betriebssystem noch hinzu.
So naheliegend eine betriebssystembasierte Lösung auf den ersten Blick auch sein mag, sie hat für die gegebene Problemstellung einen enorm hohen Mehraufwand zur ↑Laufzeit zur Folge. Als Alternative für das hier betrachte Problem des Zählens bietet es sich daher an, sowohl die ↑ISA der zugrunde liegenden CPU in Augenschein zu nehmen als auch auf besondere Konstrukte (↑*built-in command*) des Kompilierers zu achten. Hier sei insbesondere auf letzteres zurückgegriffen, was dann zu folgender Implementierung führt:

```
typedef int aint_t;

inline int FAI(aint_t *this) {            /* CPU-based fetch and increment */
    return FAA(this, 1);                    /* map to atomic fetch and add */
}
```

Diese Implementierung verursacht brutto wie netto 2 Maschinenbefehle, vorausgesetzt der Kompilierer wird explizit dazu angewiesen, für die eigentliche Zähloperation eine spezielle Befehlskombination (`lock xaddl`) der zugrunde liegenden CPU zu verwenden:

```
#define FAA __sync_fetch_and_add
```

Mit dieser Lösung wird zwar nicht in ↑Assembliersprache programmiert, jedoch sind Kenntnisse von bestimmten Eigenschaften der CPU erforderlich und wie von diesen Eigenschaften durch den Kompilierer nutzbringend Gebrauch gemacht werden kann. Die Lösung ist damit zwar prozessorabhängig, allerdings auch um etliches effizienter zur Laufzeit. Hinzu kommt noch, dass der korrekte ↑Programmablauf in Bezug auf `FAI` von einem speziellen Initialwert der Ganzzahlvariablen letztlich vollkommen unabhängig ist.

elementary operation (dt.) ↑Elementaroperation.

Elop Abkürzung für (en.) ↑*elementary operation*.

Elterverzeichnis Bezeichnung für ein ↑Verzeichnis, in dem der ↑Name von dem ↑Arbeitsverzeichnis verzeichnet ist. In ↑UNIX ist diesem Verzeichnis der Name „`..`“ (↑*dot dot*) zugeordnet. Der Eintrag erlaubt die einfache Navigation zum übergeordneten Verzeichnis im ↑Namensraum, ohne den wirklichen Namen dieses Verzeichnisses kennen zu müssen.

endianness (dt.) ↑Bytereihenfolge.

Endzeit (en.) ↑*completion time*. Bezeichnung für eine ↑Prozessgröße, die den Zeitpunkt festlegt, zu dem die Durchführung einer ↑Aufgabe tatsächlich endet.

ENIAC Abkürzung für (en.) *Electronic Numerical Integrator and Computer* (1945). Erster programmierbarer, elektronische digitaler Universalrechnerr (Röhrentechnik). Genau genommen handelte es sich bei dem ↑Rechner um eine Sammlung von Rechenmaschinen, deren ↑Programme ursprünglich durch eine Kombination von Steckbrettverdrahtung und drei tragbaren Funktionstabellen (mit jeweils 1.200 Zehnfachschaltern) in die Maschine eingegeben wurden. Die Aufgabe, ein Rechenproblem auf die Maschine abzubilden, war komplex und dauerte in der Regel Wochen. Nachdem das Programm auf dem Papier ausgearbeitet war, konnte es Tage dauern, durch Manipulation der Schalter und Kabel den eigentlichen Rechenvorgang in Gang zu bringen.

enter key (dt.) ↑Eingabetaste.

Entität (en.) ↑*entity*. Seiendes, das heißt, ein konkreter oder abstrakter Gegenstand, aber auch das Wesen dieses Gegenstands. Sammelbegriff für verschiedene, eindeutig zu bestimmende Gegenstände oder Sachverhalte, über die Informationen zur Verarbeitung durch ein ↑Rechensystem gespeichert werden sollen. Gegenstände des Rechensystems selbst sind die ↑Prozessinkarnation (inkl. ↑Faden, ↑Faser, ↑Fäserchen), der ↑Adressraum, das ↑Segment, die ↑Seite, das ↑Dateisystem, die ↑Datei, der ↑Indexknoten und so weiter, für die Sachverhalte wie etwa Anzahl, Größe, Alter, Lokalität, Geltungsbereich, Eigentümerschaften oder Zustand vermerkt werden.

entity (dt.) ↑Entität.

entry consistency (dt.) ↑Eintrittskonsistenz.

entry protocol (dt.) ↑Eintrittsprotokoll.

Entvirtualisierung (en.) ↑*devirtualisation*. Vorgang, eine bestimmte Form der ↑Virtualisierung wieder rückgängig zu machen, eine ↑virtuelle Maschine aufzulösen, zu entfernen. Typisches Beispiel für einen solchen Vorgang ist die ↑Übersetzung, wenn diese nämlich ein ↑Programm, das in der Sprache für eine ↑virtuelle Maschine M_i formuliert vorliegt, umwandelt in eine (semantisch äquivalente) Darstellung in der Sprache für eine (reale/virtuelle) Maschine M_j, $j < i$. Auch wenn bei diesem Abbildungsvorgang die Zielsprache abermals die einer virtuellen Maschine sein sollte, wurde eine Ebene der Virtualisierung entfernt, nämlich Ebene i, um der wirklichen Darstellung des betreffenden Programms für die reale Maschine näher zu kommen. Dieses Vorgehen ist typisch in einer ↑Mehrebenenmaschine, wo für ein zur Ausführung zu bringendes Programm durch Übersetzung die Ebenen 5–4 stufenweise (in wenigstens zwei Schritten) aufgelöst werden müssen, damit es schließlich in ↑Maschinensprache repräsentiert für die Ebene 3 in ausführbereiter Form vorliegt.

Ein sehr ähnlicher Zusammenhang in Bezug auf Übersetzungstechniken zeigt sich für die ↑objektorientierte Programmierung, deren wesentliches Paradigma, nämlich die ↑Vererbung, in technischer Hinsicht unter anderem durch die ↑virtuelle Methode getragen wird. Welches konkrete ↑Exemplar einer solchen Methode zur Ausführung kommt, wird für gewöhnlich erst zur ↑Laufzeit beim Aufruf anhand des ↑Datentyps des applizierten ↑Objekts bestimmt. Dazu geschieht der Aufruf indirekt, nämlich über einen ↑Sprungvektor, dessen ↑Adresse (*virtual function table pointer*) Bestandteil des Objekts selbst ist. Für einen solchen Aufruf resultieren dann jedoch teils erhebliche ↑Gemeinkosten, die für bestimmte Programmkonstruktionen allerdings vermieden werden können. Erkennt der ↑Kompilierer zur Übersetzungszeit, welche Methode direkt aufgerufen kann, wird er einen gewöhnlichen Prozeduraufruf generieren und gegebenenfalls auch den zugehörigen Sprungvektor samt ↑Zeiger darauf entfernen. Die in der *Quellsprache* ausgewiesenen virtuellen Methoden sind nach der Übersetzung in der *Zielsprache* nicht mehr sichtbar.

Ebenfalls auflösbar ist die ↑partielle Virtualisierung und die ↑Vollvirtualisierung, wobei dies jedoch nicht durch Sprachübersetzung geschieht, sondern vielmehr die *dynamische Rekonfigurierung* des ↑Rechensystems impliziert und teils auch das ↑Betriebssystem involviert. Beispielsweise kann ↑Adressraumisolation, nämlich eine der erforderlichen Funktionen für die

partielle Virtualisierung des den ↑Prozessen jeweils zugestandenen physischen ↑Adressraums, ein transientes Merkmal des Betriebssystem sein, also nur dann zur Wirkung kommen, wenn Prozesse entstehen, die dieses Merkmal auch wirklich benötigen: für manche Prozesse ist die ↑MMU samt zugehöriger ↑Systemfunktionen außer Betrieb, für andere Prozesse erfüllt diese Hardwarekomponente sehr wohl die ihr zugedachte Funktion. Gleiches lässt sich in Bezug auf einen ↑VMM festhalten, um Vollvirtualisierung entweder in oder außer Betrieb zu haben — beide Formen sind jedoch alles andere als leicht umzusetzen und erfordern einen gezielt auf derartige Maßnahmen ausgerichteten Systementwurf.

EPROM Abkürzung für (en.) *erasable PROM*, (dt.) löschbarer ↑PROM.

Ereignis (en.) ↑*event*. Auftreten von einem Geschehnis hervorgerufen durch einen ↑Prozess, beobachtbar von dem Prozess selbst oder einem anderen Prozess. Das Geschehnis ist im ↑Rechensystem direkt beobachtbar, indem ein Prozess etwa die Veränderung des Inhalts von einem ↑Speicherwort durch Abfragen (*polling*) oder Abwarten (*waiting*) wahrnimmt. Letzteres wird durch ↑unilaterale Synchronisation erreicht, das heißt, der Prozess wartet (aktiv/geschäftig oder passiv/schlafend) solange, bis ihm die Veränderung durch einen anderen Prozess explizit angezeigt wird. Das Geschehnis kann aber auch indirekt beobachtbar sein, wenn es einem Prozess nämlich möglich ist, aus wahrnehmbarem Systemverhalten auf das Auftreten gewisser Vorgänge im Rechensystem zu schließen (↑*covered channel*).

ereignisbasiertes Betriebssystem (en.) ↑*event-based operating system*. Bezeichnung für ein ↑Betriebssystem, dessen grundlegendes Konzept zur Repräsentation einer autarken ↑Aktion oder ↑Aktionsfolge das ↑Ereignis einer ↑Unterbrechung ist: jeder mögliche ↑Handlungsstrang in dem System hat seinen Ursprung als ↑asynchrone Ausnahme. Die Besonderheit eines solchen Betriebssystems besteht darin, dass sich all diese Handlungsstränge denselben ↑Laufzeitstapel teilen. im Gegensatz dazu steht ein ↑prozessbasiertes Betriebssystem, das jedem Handlungsstrang einen eigenen Laufzeitstapel zugesteht.
Im Betriebssystem alle Handlungsstränge nur auf einen einzigen Laufzeitstapel ablaufen zu lassen reduziert den eigenen Speicherplatzbedarf erheblich, insbesondere wenn daran gedacht wird, dass ein solcher Strang die wesentliche Komponente einer ↑Prozessinkarnation bildet. Allerdings ist damit innerhalb des Betriebssystems ein ↑Prozesswechsel direkt als Folge einer ↑Unterbrechungsbehandlung dann nicht möglich. Stattdessen wird ein solcher Wechsel nur noch an ausgewählten ↑Verdrängungspunkten möglich sein. Das wiederum bedeutet die grundsätzliche Entkopplung von ↑Einplanung, die sehr wohl immer noch asynchron möglich ist, und ↑Einlastung, die nur noch synchron an den Verdrängungsstellen erfolgt.

ereignisgesteuertes System (en.) ↑*event-triggered system*. Ein ↑Rechensystem dessen ↑Echtzeitbetrieb jede anstehende ↑Aufgabe zu der jeweils für sie vorgegebenen (statischen/dynamischen) ↑Priorität durchführt.

Ereigniswarteliste (en.) ↑*event waitlist*. Eine ↑Warteliste für ein bestimmtes ↑Ereignis. Alle ↑Prozesse, die auf dieser Liste verzeichnet sind, befinden sich in einer ↑Prozessblockade.

Ereigniswarteschlange (en.) ↑*event queue*. Eine ↑Ereigniswarteliste, auf die die ↑Prozesse in einer bestimmten *Reihe* angeordnet sind (↑*queue*). Bildet für gewöhnlich die Grundlage einer ↑Bedingungsvariablen.

error (dt.) ↑Fehler.

Ersetzungsalgorithmus (en.) ↑*replacement algorithm*. Lösungs- und Bearbeitungsschema, Handlungsvorschrift zur ↑Verdrängung einer ↑Seite aus dem ↑Hauptspeicher, zentraler Rechenvorgang einer ↑Ersetzungsstrategie. Ist charakterisiert durch eine bestimmte ↑Heuristik in Bezug auf die von einem ↑Prozess in naher Zukunft benutzten Seiten. . VL 12.3

Ersetzungsstrategie (en.) ↑*replacement policy*. Verfahrensweise nach der ein im ↑Hauptspeicher von einem ↑Prozess belegter ↑Speicherbereich (↑*page frame*, ↑*main-memory chunk*), auch

↑Gebrauchsstück, für ein ↑Umlagerungsmittel zur Freigabe bestimmt wird, um ein im ↑Umlagerungsbereich liegendes Pendant desselben oder eines anderen Prozesses einlagern zu können. Ursache dafür ist in aller Regel ein voll belegter Hauptspeicher in dem Moment, wenn ein ↑Prozess bei einem ↑Hauptspeicherfehlzugriff den Ablauf der ↑Ladestrategie ausgelöst hat. Da bei vollem Hauptspeicher offensichtlich kein passendes ↑Loch für den Ladevorgang bleibt, ist wenigstens eins der Gebrauchsstücke von seinem Platz im Hauptspeicher zu verdrängen.
Hintergrund ist ↑virtueller Speicher, bei dem eine solche ↑Verdrängung durch das ↑Betriebssystem auch ohne explizit im ↑Maschinenprogramm kodierte und per ↑Systemaufruf übermittelte Hinweise auskommen muss. Das besondere Problem dabei ist, den in zeitlicher Hinsicht am besten geeigneten Platz herauszufinden, nämlich ein von keinem derzeitigen Prozess mehr benutztes aber eingelagertes Gebrauchsstück festzustellen. Allerdings ist damit gleichsam eine zu treffende Aussage über das zukünftige Verhalten eines Prozesses verbunden, nämlich dass ein als unbenutzt festgestelltes Gebrauchsstück nicht schon mit dem nächsten ↑Maschinenbefehl von einem Prozess gleich wieder benutzt wird. Welche zukünftige ↑Aktion ein Prozess tätigt, kann aber gerade in einem dynamischen System nicht exakt bestimmt werden. Bestenfalls lässt sich eine Schätzung vornehmen, die von dem vergangenen und gegenwärtigen Verhalten eines Prozesses ausgeht und daraufhin auf sein zukünftiges Verhalten schließt.
Die Verdrängung eines Gebrauchsstücks „minimalinvasiv“ für die Prozesse vorzunehmen, ist der Knackpunkt der Verfahrensweise. Ein erster wichtiger Schritt in diese Richtung ist die Beschränkung auf die ↑Seite als Umlagerungsmittel. Damit gestaltet sich die Suche nach dem zu füllenden Loch oder zu ersetzenden Platz im Hauptspeicher im Vergleich zum ↑Segment als nicht mehr so schwierig: alle ↑Seitenrahmen sind gleich groß, daher gemeinhin auch gleich gut bei der Auswahl, wohingegen Speicherbereiche für Segmente typischerweise verschieden groß sind und dadurch die Suche nach dem geeigneten Platz einigermaßen komplex ist (↑*placement policy*). Folglich bildet ein ↑seitennummerierter Adressraum die Basis für virtuellen Speicher, was allerdings ↑seitennummerierte Segmentierung sehr wohl einschließt: in beiden Fällen werden ausschließlich Seiten ersetzt, womit gar die Ersetzung eines kompletten seitennummerierten Segments gemeint sein kann. Die Bestimmung der zu ersetzenden Seiten folgt dann einem ↑Ersetzungsalgorithmus, für den zu Beginn seiner Berechnung niemals alle für eine optimale Lösung benötigen Eingabedaten vorliegen (Online-Algorithmus).
Die Verfahrensweise bei der Ersetzung kann <u>lokal</u> oder <u>global</u> ausgelegt sein. Im lokalen Fall wird immer nur eine Seite ausgewählt, die genau dem ↑Prozessadressraum zugeordnet ist, aus dem heraus der Zugriff auf ein nicht im Hauptspeicher liegender Bestand von ↑Text oder ↑Daten erfolgte. Der daraus resultierende ↑Zugriffsfehler ist für den betreffenden Prozess in logischer Hinsicht vorhersagbar und auch reproduzierbar (↑*trap*), Folge der Ersetzung ist damit eine ↑Ausnahmesituation keines „fremden“ Prozesses. Nicht jedoch so im globalen Fall, bei dem nämlich eine Seite für die Ersetzung ausgewählt werden könnte, die einem beliebigen Prozessadressraum und damit insbesondere einem „fremden“ (unverwandten) Prozess zugeordnet ist, der mit dem die Ersetzung auslösenden Prozess nicht in Beziehungen steht. Dieser (fremde) Prozess wird dann unvorhersagbar und auch nicht reproduzierbar einen Zugriffsfehler erfahren (↑*interrupt*).

Erzeuger-Verbraucher-Problem (en.) ↑*producer-consumer problem*. Bezeichnung für ein typisches Muster der ↑Synchronisierung ↑gleichzeitiger Prozesse, die durch eine gemeinsame Datenstruktur gekoppelt sind. Dabei hat die Datenstruktur vornehmlich die Funktion eines ↑Puffers und die Synchronisierung folgt einem bestimmten Protokoll, das die Zugriffsreihenfolge der ↑Daten erzeugenden und verbrauchenden Prozesse auf diesen Puffer regelt. Demnach wird ein Verbraucherprozess beim Zugriff, um dem Puffer ein Datum zu entnehmen, nur dann ungehindert voranschreiten, wenn der Puffer in dem Moment nicht leer ist. Demgegenüber wird ein Erzeugerprozess beim Zugriff, um dem Puffer ein Datum hinzuzufügen, nur dann ungehindert voranschreiten, wenn der Puffer in dem Moment nicht voll ist. Entsprechend deblockiert ein Verbraucherprozess einen blockierten Erzeugerprozess bei Entnahme

des nächsten Datums, da nämlich Platz zur Aufnahme eines weiteren Datums geschaffen wird. Umgekehrt deblockiert ein Erzeugerprozess einen blockierten Verbraucherprozess bei Aufnahme des nächsten Datums, da nämlich ein freier Platz mit diesem Datum zur Entnahme belegt werden konnte.
Jedes vom Erzeugerprozess bereitgestellte Datum ist ein ↑konsumierbares Betriebsmittel für den Verbraucherprozess und jeder für das ↑Puffern eines solchen Datums benötigte Platz ist ein von beiden Prozessen zu benutzendes ↑wiederverwendbares Betriebsmittel im Sinne eines ↑Zwischenspeichers. Die Prozesse gehen eine ↑logische Synchronisation ein, da zum einen der Verbraucher von der Verfügbarkeit eines produzierten Datums und zum anderen der Erzeuger von der Verfügbarkeit eines Speicherplatzes für das zu produzierende Datum abhängt.

Exkurs Das Problem steht stellvertretend für die Kommunikation zwischen Prozessen auf Grundlage eines Puffers begrenzter Kapazität (↑*bounded buffer*). Dazu ist zunächst einmal sicherzustellen, dass die Erzeugerprozesse keinen *Pufferüberlauf* und die Verbraucherprozesse keinen *Pufferunterlauf* mit ihren jeweiligen ↑Aktionen zur Folge haben können. Zur diesbezüglichen ↑Koordinierung der Erzeuger-/Verbraucherprozesse dienen zwei ↑Koordinierungsmittel (je ein ↑allgemeiner Semaphor) zum Zählen der Verfügbarkeit von ↑Ressourcen. Der entsprechende ↑Datentyp eines solchen Puffers gestaltet sich dann wie folgt:

```
typedef struct buffer {
    semaphore_t free;                          /* initial: STORAGE_SIZE */
    semaphore_t full;                                     /* initial: 0 */
    storage_t cask;                                   /* buffer memory */
} buffer_t;
```

Der eine Semaphor (`free`) zählt die Anzahl der im Puffer noch freien Plätze, sein Initialwert ist die Anzahl der im Lager (`storage`) verfügbaren Plätze. Der andere Semaphor (`full`) zählt die Anzahl der im Lager bereits belegten Plätze, sein Initialwert ist Null. Die Operation, um ein Datum in den Puffer abzulegen, lässt sich sodann wie folgt darstellen:

```
void put(buffer_t *pipe, data_t item) {          /* put data into buffer */
    P(&pipe->free);                                   /* prevent overflow */
    store(&pipe->cask, item);                       /* make item available */
    V(&pipe->full);                                   /* signal consumable */
}
```

Der Erzeugerprozess wird in ↑P aufgehalten (d.h., blockieren), wenn der Puffer voll ist. Damit wird einem Pufferüberlauf vorgebeugt. Hat der Prozess die P-Operation hinter sich gelassen, ist damit sichergestellt, dass im Puffer noch wenigstens ein Platz zur Aufnahme des Datums vorhanden ist. Das Datum wird gespeichert (`store`). Abschließend wird durch ↑V die Verfügbarkeit des Datums für einen Verbraucherprozess angezeigt. Die V-Operation bewirkt zugleich die Deblockierung eines gegebenenfalls wartenden Verbraucherprozesses, für den die nachfolgend skizzierte Operation gilt:

```
data_t get(buffer_t *pipe) {                     /* get data from buffer */
    P(&pipe->full);                                   /* prevent underflow */
    data_t item = fetch(&pipe->cask);               /* read available item */
    V(&pipe->free);                             /* signal reusable space */

    return item;                                          /* deliver data */
}
```

Der Verbraucherprozess wird in P aufgehalten (d.h., blockieren), wenn der Puffer leer ist. Damit wird einem Pufferunterlauf vorgebeugt. Hat der Prozess die P-Operation hinter sich gelassen, ist damit sichergestellt, dass im Puffer wenigstens ein Datum zur Entnahme vorhanden ist. Das Datum wird dem Puffer entnommen (`fetch`). Abschließend wird durch V die Verfügbarkeit eines Pufferplatzes für einen Erzeugerprozess angezeigt, was zugleich die Deblockierung eines gegebenenfalls darauf wartenden Prozesses bewirkt.
Zusätzlich zur Abwehr möglicher Pufferüber- und -unterläufe ist sicherzustellen, dass ein

und derselbe *Pufferplatz* nicht zugleich von mehreren Erzeugerprozessen zur Ablage eines Datums genutzt beziehungsweise ein und dasselbe Datum eines solchen Platzes nicht zugleich an mehrere Verbraucherprozesse herausgegeben werden kann, denn:

- die Anzahl gleichzeitig erlaubter Erzeugerprozesse ist bestimmt durch die Anzahl freier Pufferplätze und
- die Anzahl gleichzeitig erlaubter Verbraucherprozesse ist bestimmt durch die Anzahl belegter Pufferplätze.

Aus diesem Grunde ist sowohl das Hinzufügen als auch das Entfernen eines Datums jeweils als ↑Elementaroperation zu formulieren. Die zur Buchführung der Lagerbelegung entsprechende Datenstruktur hat den nachfolgend skizzierten Aufbau:

```
typedef struct storage {
    aint_t get, put;                                    /* index variables */
    data_t bay[STORAGE_SIZE];                              /* memory space */
} storage_t;
```

Die beiden Laufzählerattribute (`get`, `put`) sind ↑Exemplare eines Datentyps, der das Schema einer unteilbaren Ganzzahl (*atomic integer*) definiert. Die auf diesen Datentyp definierte Elementaroperation (`FAI`, S. 81) sorgt dafür, dass auch im Konfliktfall (↑*contention*) jeder auf das Arbeitsfeld (`bay`) desselben Lagers gleichzeitig zugreifende Prozess einen eindeutigen Laufzähler für die Indizierung erhält. Diese Elementaroperation wird von den beiden Lageroperationen wie folgt benutzt:

```
inline void store(storage_t *cask, data_t item) {
    cask->bay[FAI(&cask->put) % STORAGE_SIZE] = item;    /* insert element */
}

inline data_t fetch(storage_t *cask) {
    return cask->bay[FAI(&cask->get) % STORAGE_SIZE];    /* remove element */
}
```

Das Arbeitsfeld ist als *Ringpuffer* organisiert, was durch den Modulooperator (`%`) angewendet einerseits auf den jeweils bestimmten Laufzählerwert zur Indizierung und anderseits der als Zweierpotenz vorausgesetzten Arbeitsfeldgröße des Lagers (`STORAGE_SIZE`) zum Ausdruck kommt. Beim Zugriff auf einen einzelnen Eintrag auf das Lagerfeld bildet `FAI` (*fetch and increment*) die zentrale Operation (vgl. S. 81). Diese sorgt für die benötigte ↑Unteilbarkeit der Berechnung des Laufzählerwerts, nämlich damit niemals mehrere Prozesse zugleich denselben Eintrag in diesem Feld lesen oder schreiben können. Unteilbar findet die folgende ↑Aktionsfolge statt:

1. Auslesen des ↑Platzhalters und den gelesenen Wert als Rückgabewert der Operation vorhalten (*fetch*).
2. Erhöhung des gelesenen Werts um Eins und das Ergebnis zurück in den Platzhalter schreiben (*increment*).

Damit wird der Laufindex für die Aktionen zum Einspeichern (`store`) und zum Abrufen (`fetch`) eines Datums trotz eventueller gleichzeitiger Erzeuger- und Verbraucherprozesse immer atomar verändert. Als Konsequenz daraus können gleichzeitige Prozesse niemals denselben Pufferplatz zugleich adressieren.
Wie an anderer Stelle (S. 81) diskutiert, kann die Elementarität von `FAI` in Abhängigkeit von der jeweils zum Bezug genommenen ↑Abstraktionsebene auf unterschiedliche Art und Weise durchgesetzt werden. Entsprechend der sich daraus ergebenden Variationsmöglichkeiten hat dies Konsequenzen für die Initialisierung eines ganzen Pufferexemplars. Nachfolgend sind zwei grundlegende Variationen formuliert (die jedoch so für gewöhnlich nicht in derselben ↑Übersetzungseinheit stehen):

```
buffer_t hwb = { {STORAGE_SIZE}, {0}, {{1}, {1}}}; /* heavy-weight buffer */
buffer_t lwb = { {STORAGE_SIZE}, {0}};             /* light-weight buffer */
```

In der ersten Variante (`hwb`) sind die Laufzählerattribute (`get`, `put`) der Lagerdatenstruktur (`storage`) des Puffers durch Abstraktionen des ↑Betriebssystems geschützt, konkret wird hier der jeweiligen Ganzzahl ein ↑binärer Semaphor zur Seite gestellt, der korrekt mit Eins vorzubelegen ist, damit `FAI` keine ↑Verklemmung verursacht. Stattdessen basiert die zweite Variante (`lwb`) zur Absicherung dieser beiden Attribute einzig auf Abstraktionen der ↑CPU, konkret auf eine ↑atomare Operation, auf die `FAI` abgebildet wird und die sonst keine zusätzlichen Attribute erforderlich macht. Beiden Varianten ist jedoch als Betriebssystemabstraktion der allgemeine (d.h. zählende) Semaphor grundlegend, um die logische Synchronisation von Erzeuger- und Verbraucherprozessen durchzusetzen. Benötigt werden immer zwei dieser Semaphore, deren Vorbelegung in beiden Fällen auch identisch ist.

escape character (dt.) ↑Fluchtsymbol.

Ethernet Eine Technologiefamilie zum Aufbau von ↑Rechnernetzen, deren Übertragungstechnik auf ↑CSMA/CD basiert (Xerox PARC, 1973). In Analogie zum — mittlerweile widerlegten — lichterzeugenden/-ausbreitendem „Äther“ ein allgegenwärtiges, vollkommen passives Medium zur Verbreitung elektromagnetischer Wellen.

event (dt.) ↑Ereignis.

event queue (dt.) ↑Ereigniswarteschlange.

event waitlist (dt.) ↑Ereigniswarteliste.

event-based operating system (dt.) ↑ereignisbasiertes Betriebssystem.

event-triggered system (dt.) ↑ereignisgesteuertes System.

exception (dt.) ↑Ausnahme.

exception condition (dt.) ↑Ausnahmebedingung.

exception dispatcher (dt.) ↑Ausnahmezuteiler.

exception handler (dt.) ↑Ausnahmehandhaber.

exception handling (dt.) ↑Ausnahmebehandlung.

exceptional situation (dt.) ↑Ausnahmesituation.

exclusion zone (dt.) ↑Sperrzone.

execution time (dt.) ↑Ausführungszeit.

Exemplar (en.) ↑*instance*. Einzelstück aus einer Menge gleichartiger Stücke (Duden). Die Stücke sind gleichartig, weil ihnen dasselbe Modell (↑*data type*), dieselbe Bauart zugrunde liegt: sie sind vom selben *Bautyp*. So ist eine ↑Variable in einem ↑Programm ein solches Einzelstück, ebenso wie ein ↑Objekt allgemein.

exit protocol (dt.) ↑Austrittsprotokoll.

exploit (dt.) Glanzleistung, Tat. Ein spezielles ↑Programm zum Zwecke der systematischen Ausnutzung einer Schwachstelle in einem ↑Rechensystem; ↑Schadsoftware.

exponential backoff (dt.) gemäß einer Exponentialfunktion verlaufend zurückweichen (durch Anpassung der ↑Ruhezeit; in Anlehnung an den Duden). Ein auf *Rückkopplung* basierendes Verfahren zur ↑Stauauflösung, das die Rate bestimmter ↑Aktionen eines ↑Prozesses in Bezug auf ein ↑gemeinsames Betriebsmittel im Moment des Zugriffs multiplikativ verringert, um schrittweise eine für das ↑Bezugssystem akzeptable Rate zu erreichen (↑*backoff*).

exponentielle Glättung Bezeichnung für ein Verfahren der ↑Zeitreihenanalyse, mit dem anhand eines gemessenen Werts x_t der jüngsten Vergangenheit und eines prognostizierten Werts s_t der Gegenwart ein zu erwartender Wert s_{t+1} für die nahe Zukunft abgeschätzt wird. Zur Gewichtung des Einflusses des gemessenen und prognostizierten Wertes auf die Schätzung findet ein Glättungsfaktor α Verwendung. Eine solche Glättung (1. Ordnung) wird typischerweise mit folgender rekursiven Formel berechnet:

$$s_{t+1} = \alpha \cdot x_t + (1 - \alpha) \cdot s_t, \text{ mit } 0 \leq \alpha \leq 1.$$

Mit dem Ansatz kann für die ↑mitlaufende Planung bei Bedarf und sehr einfach die Länge des nächsten zu erwartenden ↑Rechenstoßes eines ↑Prozesses abgeschätzt werden (vgl. ↑SPN).

external fragmentation (dt.) ↑externe Fragmentierung.

external process (dt.) ↑externer Prozess.

externe Fragmentierung (en.) ↑*external fragmentation*. Gesamtheit an ↑Verschnitt im ↑Hauptspeicher. Der Grad der ↑Fragmentierung ist so hoch, dass kein einziger freier Abschnitt (↑*hole*) für die ↑Speicherzuteilung nutzbar ist. Oft sind in der Situation zwar genügend viele ↑Bytes im Hauptspeicher frei, sie liegen jedoch zu stark verstreut vor und bilden damit keinen zusammenhängenden Abschnitt angeforderter Größe. Der Verschnitt lässt sich durch ↑Defragmentierung auflösen.

externer Prozess (en.) ↑*external process*. Ein in der ↑Peripherie stattfindender und eindeutig gekennzeichneter Vorgang zur ↑Ein-/Ausgabe von ↑Daten. Der ↑Prozessor, auf dem ein solcher ↑Prozess abläuft, ist ein ↑Peripheriegerät.

FAA Abkürzung für (en.) *fetch and add*; Bezeichnung für eine ↑atomare Operation mit zwei Operanden: `word_t FAA(word_t *, word_t)`. Der erste Operand ist die ↑Adresse eines ↑Speicherworts, dessen Wert vor dem Addieren mit dem ganzzahligen Wert ($\mathbb{Z}$) des zweiten Operanden gelesen und zurückgeliefert wird (Postinkrement):

```
word_t FAA(word_t *ref, word_t val) {
    word_t aux;                                  /* auxiliary variable */
    atomic {                                  /* enforce indivisibility */
        aux = *ref;                              /* read current value */
        *ref += val;                          /* modify memory contents */
    }
    return aux;                               /* deliver previous value */
}
```

Mit `__sync_fetch_and_add(ref, val)` definiert ↑GCC für diese Operation eine Standardfunktion (*atomic built-in*), die als ↑atomarer Befehl ausgelegt ist und erscheint beispielsweise für ↑x86 als `lock xadd` (bei Verwendung als Funktion, d.h., einen Wert liefernd, `int`) beziehungsweise `lock add` oder `lock inc` (bei Verwendung als Prozedur, d.h., keinen Wert liefernd, `void`).

Faden (en.) ↑*thread*. Bezeichnung für einen ↑Handlungsstrang, der einen eigenen ↑Laufzeitkontext besitzt und typischerweise mitlaufender (*on-line*) ↑Ablaufplanung unterliegt. Grob differenziert in ↑Anwendungsfaden und ↑Systemkernfaden.

failure (dt.) Fehlfunktion, ↑Versagen.

fairness (dt.) ↑Gerechtigkeit.

falsche Signalisierung (en.) ↑*wrong signalling*. Bezeichnung für einen ↑Defekt, den ein inkorrektes ↑nichtsequentielles Programm aufweist oder einen ↑Fehler, den ein ↑nichtsequentieller Prozess trotz korrektem Programm verursacht. Fehlverhalten eines ↑Prozesses, das in Zusammenhang mit der zugrunde liegenden ↑Signalisierungsdisziplin zu betrachten ist.

Ausgegangen wird dabei von einem Prozess, der auf eine erfüllte Wartebedingung getroffen ist und daraufhin warten muss, bis diese durch einen anderen Prozess aufgehoben wird. Dazu erwartet der Prozess ein ↑Ereignis, das ihm die Aufhebung dieser Wartebedingung meldet. Diesem Ereignis zugeordnet ist eine ↑Synchronisationsvariable, die der betreffenden Wartebeziehungsweise Meldeoperation für gewöhnlich als Parameter übergeben wird.
Für einen Prozess können mehrere, verschiedene Wartebedingungen durch das nichtsequentielle Programm vorgeschrieben sein. Entsprechend werden mehrere, zu diesen Wartebedingungen jeweils korrespondierende, aber eben verschiedene Ereignisse zum Ausdruck bringende Meldeoperationen formuliert sein. Jedem Ereignis ist dabei eine eigene Synchronisationsvariable zugeordnet.
Angenommen, für einen Prozess P_a ist Wartebedingung C_1 erfüllt, er wartet daraufhin auf das Eintreten von Ereignis E_1 durch Verwendung der Synchronisationsvariablen V_1. Weiterhin sei angenommen, für einen beliebigen anderen, ebenfalls wartenden und Synchronisationsvariable V_2 nutzenden Prozess P_b gilt eine erfüllte Wartebedingung C_2, deren Aufhebung aber Ereignis E_2 bedeutet. Ein Prozess P_c hebt nun Wartebedingung C_2 auf, verwendet anstatt V_2 dann aber fälschlicherweise Synchronisationsvariable V_1, um Ereignis E_2 zu melden. In dem Moment wird Prozess P_a signalisiert (nicht P_b, wie es richtig wäre), damit aus seinem Wartezustand befreit, und seine Tätigkeit wieder aufnehmen, obwohl für ihn Wartebedingung C_1 nach wie vor erfüllt ist. Prozess P_a schreitet aber voran in der irrtümlichen Annahme, Ereignis E_1 sei eingetreten, seine Wartebedingung C_1 sei aufgehoben. Hinzu kommt, dass das korrekterweise für Prozess P_b vorgesehene Signal verloren ist. Gegebenenfalls wartet dieser Prozess nun ewig auf ein Signal, das nur einmal gemeldet werden würde.
Wenn solch ein mögliches Fehlverhalten eines signalisierenden Prozesses nicht ausgeschlossen werden kann, dann sollte ein signalisierter Prozess nach Wiederaufnahme eigenständig seine Wartebedingung erneut überprüfen, bevor er seiner eigentlichen Tätigkeit nachgeht. Gegebenenfalls wird der Prozess daraufhin abermals in den Wartezustand versetzt, wenn nämlich seine Wartebedingung immer noch erfüllt ist. Der signalisierte Prozess verhält sich damit *fehlertolerant*, er toleriert die irrtümliche Meldung eines Ereignisses. Allerdings ist mit dieser Maßnahme nur einer Seite geholfen. Ein möglicherweise wartender Prozess, der stattdessen hätte signalisiert werden müssen, bleibt dabei unberücksichtigt, er ließe sich auch nicht durch den tolerant agierenden Prozess identifzieren.

false sharing (dt.) ↑irrige Mitbenutzung.

Fangstelle (en.) ↑*trap*. Bezugnehmend auf einen bestimmten ↑Prozess der Punkt, an dem die Beanstandung (↑*interception*) einer ↑Aktion geschieht, eine ↑Ausnahme hervorgebracht und als Folge dessen die Aktion vom ausführenden ↑Prozessor abgefangen wird. Ist der Prozessor die ↑CPU, entspricht dieser Punkt der ↑Adresse des abgefangenen ↑Maschinenbefehls im ↑Maschinenprogramm — an der der Prozess dem Prozessor in die „Falle“ gegangen ist.

FAS Abkürzung für (en.) *fetch and store*; Bezeichnung für eine ↑atomare Operation mit zwei Operanden: `word_t FAS(word_t *, word_t)`. Der erste Operand ist die ↑Adresse eines ↑Speicherworts, dessen Wert vor dem Überschreiben mit dem Wert des zweiten Operanden gelesen und zurückgeliefert wird:

```
word_t FAS(word_t *ref, word_t val) {
    word_t aux;                                  /* auxiliary variable */
    atomic {                                  /* enforce indivisibility */
        aux = *ref;                              /* read current value */
        *ref = val;                                 /* write new value */
    }
    return aux;                               /* deliver previous value */
}
```

Mit `__sync_lock_test_and_set(ref, val)` definiert ↑GCC für diese Operation eine Standardfunktion (*atomic built-in*), die als ↑atomarer Befehl ausgelegt ist und beispielsweise für ↑x86 als `xchg` erscheint (s. auch ↑TAS).

Faser (en.) ↑*fiber*. Bezeichnung für einen ↑Faden, der strikt kooperativer ↑Ablaufplanung unterliegt und damit nicht gleichzeitig mit anderen seiner Art abläuft. Auch *leichtgewichtiger Faden*, da bei dieser Art der Ablaufplanung die Kontrolle durch das ↑Betriebssystem entfällt, der Aufwand für die ↑Zustandssicherung auf ein Minimum reduziert werden kann und ↑Synchronisation implizit sichergestellt ist. Verschiedentlich auch einer ↑Koroutine gleichgestellt.

Fäserchen (en.) ↑*fibril*. Bezeichnung für einen ↑Faden, der ausschließlich im ↑Betriebssystemkern (↑Linux) residiert. Ein solcher Faden bildet die technische Grundlage, um eine ↑Systemfunktion, etwa ausgelöst durch einen asynchronen ↑Systemaufruf, unabhängig von anderen Vorgängen stattfinden zu lassen.

fault (dt.) ↑Defekt.

fault-error-failure chain (dt.) ↑Gefahrenkette.

FB Abkürzung für (en.) *feedback*. Bezeichnung einer Kategorie von Strategien für die ↑mitlaufende Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor mithilfe von *Rückkopplung*. Dabei werden eine oder mehrere von einem ↑Prozess generierte ↑Prozessgrößen in die Planung zurückgeführt, um für den nächsten Durchlauf den relativen ↑Rang des betreffenden Prozesses auf der ↑Bereitliste zu bestimmen. Typisches Beispiel einer solchen Größe ist die ↑Bedienzeit, die einen Aufschluss über den Interaktivitätsgrad eines Prozesses geben kann. Als dialogorientiert eingestufte Prozesse (eher kurze ↑Rechenstoßlängen im Vergleich zu den ↑Ein-/Ausgabestoßlängen) erhalten dann einen höheren Rang, andere dagegen fallen in ihrem Rang ab. In ähnlicher Weise lässt sich der ↑Alterung eines Prozesses entgegenwirken, indem zunehmende ↑Wartezeit auf den Prozessor den Prozessrang erhöhen kann. Solche Verfahren können rein analytisch ausgelegt sein (↑SPN, ↑HRRN) oder durch allein konstruktive Maßnahmen ihre Funktion erfüllen (↑MLFQ).

FCFS Abkürzung für (en.) ↑*first-come, first-served*. Bezeichnung für eine Strategie zur ↑Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor in der Reihenfolge ihrer ↑Ankunftszeit. Sowohl ↑vorlaufende Planung als auch ↑mitlaufende Planung kann nach diesem Schema vorgehen. Charakterisiert im zuletzt genannten Fall die Aufgabenbearbeitung einen bestimmten ↑Prozess, so ist das Verfahren zudem ein Beispiel für die ↑kooperative Planung: es funktioniert dann nach dem ↑Windhundprinzip und greift immer, sobald eine der Aufgaben in den ↑Prozesszustand <u>bereit</u> gebracht wird. In Bezug auf die Reihung handelt es sich um eine zu ↑FIFO analoge Methode, die allerdings weder das ↑Puffern von ↑Daten noch die Abfederung gegenläufiger Wirkungskräfte bezweckt.

Die Bereitstellung von Aufgaben geschieht synchron und, je nach ↑Operationsprinzip des ↑Betriebssystems, asynchron zum jeweils auf dem betreffenden Prozessor stattfindenden Prozess. Im synchronen Fall werden die Aufgaben direkt oder indirekt durch ↑Systemaufrufe und damit von dem jeweils stattfindenden Prozess selbst eingeplant. Demgegenüber ist die Aufgabenplanung im optionalen asynchronen Fall die mögliche Folge einer ↑Unterbrechungsanforderung. Allerdings ändert letztere nichts an der Bearbeitungsreihenfolge der Aufgaben, sondern bedeutet lediglich ein zweites Moment der Einspeisung zur Bearbeitung bereiter Aufgaben: ein durch die Unterbrechungsanforderung möglicher Entzug des Prozessors findet grundsätzlich nicht statt.

Für die Prozesse allgemein ist dies eine faire Planungsstrategie, jedoch kann der dem Verfahren innewohnende ↑Konvoieffekt die Auslastung von ↑Betriebsmitteln, die insbesondere zur Verwaltung oder Durchführung von ↑Ein-/Ausgabe anfallen, erheblich beeinträchtigen. Demgegenüber ist die Implementierung des Verfahrens alles andere als kompliziert, sie ermöglicht zudem eine sehr kurze ↑Einplanungslatenz. Wenn der Prozessmix durchgehend (nahezu) homogen ist, wird das Verfahren auch zu einer guten Auslastung der Betriebsmittel führen können. Allerdings ist ein solcher Mix gerade bei ↑Simultanverarbeitung im Allgemeinen nicht gegeben, weshalb andere Einplanungsverfahren erforderlich sind, die dann auch an Komplexität zunehmen.

Exkurs Grundlage für die Implementierung des Verfahrens ist für gewöhnlich ein dynamischer ↑Datentyp, mit dem eine ↑Schlange der ein- und ausgehenden Aufgaben geformt werden kann. Damit ist gerade im Falle mitlaufender Planung die Reihung der Prozesse in konstanter ↑Laufzeit möglich, wie nachfolgend beispielhaft gezeigt wird:

```
void ready(process_t *task) {                /* schedule according to FCFS */
    process_t *self = being(ONESELF);        /* use own process descriptor */

    assert((task != self) || valid(&self->mood, PENDING | IDLE));

    state(&task->mood, READY);                  /* set process ready to run */
    enqueue(labor(), &task->line);     /* add its descriptor to ready list */
}
```

Normalerweise setzt sich ein Prozess nicht unkontrolliert von selbst auf die ↑Bereitliste, da er sonst fälschlicherweise in einer eventuellen Wartesituation ungewollt oder bei ↑Simultanverarbeitung mehrfach voranschreiten könnte. Gegebenenfalls würde eine solche Maßnahme sogar dem Planungsverfahren widersprechen, wenn nämlich (z.B. bei ↑SRTF oder ↑EDF, aber ggf. auch ↑SPN oder ↑HRRN) der laufende Prozess per Definition den Prozessor solange zu behalten hat, bis er entweder von sich aus blockiert (↑*run to completion*) oder zugunsten eines anderen Prozesses vom Prozessor durch den ↑Planer verdrängt wird. Jedoch bilden der ↑Leerlauf und bestimmte Sonderfälle (`PENDING`: s. `pause`, unten) zulässige Ausnahmen. So wird die Bereitstellung für gewöhnlich (d.h., ohne Übersetzungsschalter `-DNDEBUG`) und nur ausnahmsweise für diese Sonderfälle gelingen.
Die Einreihungsoperation (`enqueue`, vgl. S. 277) platziert den einzuplanenden Prozess immer ans Ende der Schlange (`labor`). Wie bereits erwähnt kann die Bereitstellung (`ready`) synchron oder asynchron ausgelöst erfolgen. Ist jedoch letzteres im Betriebssystem vorgesehen, sind sämtliche Operationen des mitlaufenden Planers, zusätzlich zu ihrer eigentlichen Funktion, der ↑Synchronisation zu unterziehen (↑*cross-cutting concern*).
Neben der Bereitstellung umfasst jeder mitlaufende Planer noch weitere Operationen, die allgemein und in verschiedener Hinsicht die ↑Umplanung betreffen. Beispielsweise eine Operation, die es einem Prozess in kooperativer Weise ermöglicht, den Prozessor zugunsten eines anderen Prozesses abzugeben:

```
void pause() {                           /* voluntarily release processor */
    process_t *next = elect(labor());      /* select ready-to-run process */
    if (next) {                                          /* any available? */
        process_t *self = being(ONESELF);              /* yes, use own PCB */
        state(&self->mood, PENDING);             /* enter transition state */
        ready(self);                         /* put oneself onto ready list */
        seize(next);               /* switch to next ready-to-run process */
    }
}
```

Mit dieser Operation (`pause`) kann ein Prozess von Zeit zu Zeit seinen Prozessor zur Disposition stellen und einen ↑Prozesswechsel erbeten. Jedoch pausiert der Prozess nur, wenn die Auswahl (`elect`) einen Nachfolger für ihn auf der ↑Bereitliste (`labor`) hat finden können. Ist dies der Fall, betritt der Prozess einen Zwischenzustand (`PENDING`), um damit anzuzeigen, dass die nachfolgende Platzierung von sich selbst auf die Bereitliste schon seine Richtigkeit hat (vgl. `ready`: Sonderfall, `assert`). Der damit eintretende ↑Schwebezustand, der den Prozess nämlich als <u>laufend</u> <u>bereit</u> ausweist, wird beim anstehenden Prozesswechsel (`seize`) wieder aufgehoben.
Zu diesem Punkt der Prozesspause kommt noch ein weiterer Aspekt hinzu, nämlich Unterstützung für die ↑logische Synchronisation. Diese ergibt sich zwangsläufig, wenn Prozesse sich nicht nur dem ↑Betriebssystem gegenüber kooperativ verhalten müssen, sondern auch untereinander. Eine zentrale Operation dazu ist, einen Prozess, der auf ein bestimmtes ↑Ereignis warten muss, zu blockieren:

```
void block() {                          /* release and reschedule processor */
    process_t *next, *self = being(ONESELF);

    state(&self->mood, BLOCKED);                    /* mark myself blocked */

    next = quest(labor());           /* search for next process to execute */
    if (next == self)                         /* myself already unblocked? */
        state(&self->mood, FLUSH | RUNNING);              /* yes, continue */
    else                                          /* no, release processor */
        seize(next);                             /* switch to next process */
}
```

Zu beachten ist auch hier der Schwebezustand, den ein blockierender Prozess einnimmt: in dem Moment nämlich, wo dieser logisch in den ↑Prozesszustand blockiert (*blocked*) übergeht, ist er immer noch physisch laufend (*running*). Dieser Schwebezustand endet erst, wenn der betreffende Prozess den Prozessor entweder behalten darf (dann-Zweig der Fallunterscheidung) odert wirklich abgibt (`seize`).
Die Blockierung eines Prozesses impliziert die Auswahl (`quest`) eines Gleichgesinnten, der als nächster der ↑Einlastung (`seize`) zuzuführen ist. Dieser nächste Prozess kann jedoch derjenige sein, der sich gerade auf dem Wege zur Prozessorabgabe befindet: denn das Ereignis, das dieser Prozess erwartet, kann zwischenzeitlich eingetreten und beispielsweise durch eine ↑Unterbrechungsanforderung angezeigt worden sein. Im Rahmen der ↑Unterbrechungsbehandlung kann dieser Prozess sodann wieder bereitgestellt werden, so dass er sich (in `quest`) selbst auf der Bereitliste wiederfindet.
Daher wird der Prozesswechsel (`seize`) nur bedingt geschehen. Das Bestreben, einen bereitgestellten Prozess zu liefern, kann jedoch mit Leerlauf einhergehen, nämlich wenn in dem Moment die Bereitliste keine Prozesseinträge enthält. Die betreffende Funktion (`quest`) muss somit die Tatsache berücksichtigen, gegebenenfalls auf die Bereitstellung eines Prozesses unbestimmt lang warten zu müssen:

```
process_t *quest(backlog_t *list) {    /* seek for a ready-to-run process */
    process_t *next;

    while ((next = elect(list)) == 0)        /* any ready-to-run process? */
        stall(batch(list));                     /* no, shutdown operation */

    return next;                           /* ready-to-run process elected */
}
```

In diesem Beispiel übernimmt immer der Prozess, der im Falle einer leeren Bereitliste blockiert, die Rolle des ↑Leerlaufprozesses (`stall`). Für gewöhnlich versetzt dieser den Prozessor in den ↑Schlafzustand, der erst durch eine Unterbrechungsanforderung beendet wird. Kommt es als Folge einer solchen Anforderung zur Bereitstellung (`ready`) eines Prozesses, wird dieser nach Wiederaufnahme des Leerlaufprozesses gefunden (`elect`) und als Ergebnis zurückgeliefert. Dabei ist zu beachten, dass dieses Ergebnis eben auch genau jener Prozess sein kann, der nicht nur diese Schleife, sondern auch den Leerlauf kontrolliert hat.
Zu guter Letzt noch einen Hinweis zur Auswahlfunktion (`elect`), die einen bereitgestellten Prozess der Bereitliste entnehmen soll. Für das hier erklärte Verfahren entfernt diese Funktion lediglich das Kopfelement von der betreffenden Schlange (`dequeue`):

```
process_t *elect(backlog_t *list) {        /* choose ready-to-run process */
    return forge(dequeue(list));
}
```

Jedoch ergibt sich dabei ein Typkonflikt, da das Kettenglied (`chain_t`, S. 374) zwar im ↑Prozesskontrollblock (`process_t`) enthalten ist, dieser selbst aber kein Kettenglied ist. Daher ist aus dem der Schlange entnommenen Kettenglied noch ein Prozesskontrollblock zu formen:

```
process_t *forge(chain_t *item) {                /* make process pointer */
    return (process_t *)coerce(item, (int)&((process_t *)0)->line);
}
```

Diese „magische“ Anweisung sorgt für eine dynamische Typumwandlung: sie nötigt (`coerce`) einen Zeiger (`chain_t *`) auf ein Kettenglied als Zeiger (`process_t *`) auf einen Prozesskontrollblock zu erscheinen, wobei letzterer das Kettenglied (`line`) als Attribut enthalten muss. Derartige Formulierungen sind in ↑C mangels Spracheigenschaften hin und wieder leider nicht zu vermeiden. Dahinter steckt eine Zeigermanipulation folgender Art:

```
inline void *coerce(void *ptr, int off) {          /* ugly, but needed... */
    return ptr ? (void *)((unsigned)ptr - off) : 0;
}
```

Anderen Sprachen (↑C++) bieten dazu einen speziellen Operator (`dynamic_cast`), wobei dann der ↑Kompilierer eine Anweisung eben gezeigter Form generiert.

feather-weight process (dt.) ↑federgewichtiger Prozess.

federgewichtiger Prozess (en.) ↑*feather-weight process*. Bezeichnung für einen ↑Prozess, der mit anderen Prozessen seiner Art zusammen im gemeinsamen und durch ↑Speicherschutz isolierten ↑Adressraum stattfindet. Ein solcher Prozess kann als ↑Anwendungsfaden oder als ↑Systemkernfaden technisch ausgeprägt sein, das heißt, er residiert entweder auf der ↑Maschinenprogrammebene oder innerhalb des ↑Betriebssystemkerns.

feedback control (dt.) ↑Regelung.

Fehler (en.) ↑*error*. Bezeichnung für eine bestimmte Art von ↑Gefahr, die als Folge eines ↑Defekts zum ↑Versagen des Systems führen kann. Abweichung vom spezifizierten Richtigen, die erst zur ↑Laufzeit eines ↑Programms wahrgenommen wird (↑*bug*). Mittelglied in einer ↑Gefahrenkette.

Fehleraufspürer (en.) ↑*debugger*. Ein spezielles ↑Maschinenprogramm zur ↑Fehlerbereinigung. Mit Hilfe dieses Programms können ↑Fehler, die bei einem ↑Programmablauf festgestellt wurden, nachträglich aufgespürt, bestimmt und gegebenenfalls in ihren weiteren Auswirkungen gedeutet werden. Für gewöhnlich wird damit aber der festgestellte Programmfehler nicht beseitigt, sondern lediglich die Stelle im betreffenden ↑Quellmodul, an der danach die Korrektur vorgenommen werden sollte oder muss, mehr oder weniger genau eingegrenzt.
Ein solches Fehlersuchprogramm ist immer zugeschnitten für eine bestimmte ↑virtuelle Maschine und damit insbesondere abhängig von der ↑Abstraktionsebene des Programms, bei dessen Ausführung der Fehler aufgetreten ist. In der üblichen Organisation eines ↑Rechensystems als ↑Mehrebenenmaschine kann es somit auch mehrere dieser Suchprogramme geben, und zwar für jede Abstraktionsebene oberhalb der ↑Befehlssatzebene. Ein typisches Beispiel ist `gdb(1)`, der ↑GNU *Debugger* für eine bestimmte ↑Maschinenprogrammebene.
Als spezieller ↑Kommandointerpreter zur Fehlersuche ermöglicht dieses Maschinenprogramm, den ↑Prozessorstatus des zu kontrollierenden Programmablaufs einzusehen und zu verändern, ebenso die Inhalte von ↑Speicherzellen im ↑Arbeitsspeicher zu lesen oder zu schreiben. Darüber hinaus können ↑Haltepunkte gesetzt und gelöscht werden, um den der Fehlersuche unterzogenen Programmablauf in definierten Schritten stattfinden zu lassen und so die sich entwickelnden Zustände schrittweise zu verfolgen und zu deuten.

Fehlerbereinigung (en.) ↑*debugging*. Vorgang zur Berichtigung von ↑Programmen, für gewöhnlich unterstützt durch einen ↑Fehleraufspürer. Dabei wird der zu untersuchende ↑Programmablauf im Original betrachtet, das heißt, ohne bloß zum Zwecke der Fehlersuche Änderungen an den entsprechenden ↑Quellmodulen vorgenommen zu haben.
Allerdings ist letzteres der nicht selten zuerst praktizierte Ansatz, nämlich durch eingestreute Ausgabeanweisungen (`printf(3)`) schrittweise den Fortschritt und die dabei erfolgenden Zustandsänderungen des Programmablaufs anzuzeigen. Diese Technik, die letztlich eine bestimmte Form von ↑Instrumentierung darstellt, greift jedoch massiv in das originale Programm ein, führt damit bei der Fehlersuche auch zu stark veränderten Programmabläufen, und setzt überhaupt die Verfügbarkeit von Quellmodulen voraus. Eine Fehlersuche, die etwa

das ↑Laufzeitsystem mit einschließt, ist so nicht praktikabel.
Das Aufspüren von Fehlern ist nicht mit Maßnahmen zur ↑Profilbildung zu verwechseln, obwohl dadurch auch Fehlverhalten aufgedeckt werden kann. Profilbildung dient einem anderen Zweck, beispielsweise der Suche nach Leistungsengpässen in einem Programmablauf. Derartige Engpässe sind „nichtfunktionale“ Mängel, die bei der Fehlersuche nicht im Vordergrund stehen: Fehlersuche soll „funktionale“ Mängel aufspüren.

Feld (en.) ↑*array*. Bestimmte Anordnung von (für gewöhnlich) mehreren gleichartig strukturierten ↑Daten, die jeweils ↑Exemplare ein und desselben ↑Datentyps sind. ↑Adressierungsart, um auf die einzelnen Feldelemente zuzugreifen, ist typischerweise die indizierte Adressierung.

fence (dt.) ↑Schutzgatter.

fence address (dt.) ↑Gatteradresse.

fence register (dt.) ↑Schutzgatterregister.

fencing (dt.) ↑Abgrenzung.

Fernschreiber (en.) ↑*teletypewriter*. Gerät, schreibmaschinenähnlich, das der Aufnahme und Übermittlung von Schriftzeichen dient (Duden) und mit ↑Tabellierpapier bestückt ist. Auch als ↑Dialogstation genutzt.

Fernschreibkode (en.) ↑*telegraph code*. Schlüssel, mit dessen Hilfe chiffrierte ↑Daten übertragen werden können. Ein 5-Bit Kode, in zwei Versionen standardisiert durch das ↑CCITT: Kode Nr. 1, auch Baudot-Kode, nach Jean-Maurice-Émile Baudot (1845–1903) und Kode Nr. 2, auch Murray-Kode, nach Donald Murray (1865–1945). Letzterer findet als Umschaltkode breite Verwendung in der ↑EDV.
Durch zwei Steuerzeichen wird zwischen Buchstaben- oder Ziffernmodus gewählt: `LS` (*letter shift*), 11111_2, Buchstabenumschaltung und `FS` (*figure shift*), 11011_2, Ziffernumschaltung. So werden 26 Bitkombinationen, die für die Buchstaben im lateinischen Alphabet stehen, doppelt belegt. Im Ziffernmodus sind jedoch drei Bitkombinationen keinem Zeichen zugewiesen (reserviert). Weitere (in beiden Modi gültige) Steuerzeichen sorgen für ↑Zeilenvorschub (01000_2), ↑Wagenrücklauf (00010_2) und Zwischenraum (*space*, 00100_2). Die Bitkombination 00000_2 wird nicht verwendet. Ausgenommen LS, FS und den vier unbelegten Bitkombinationen umfasst der Kode damit 52 Zeichen.

Festspeicher (en.) ↑*permanent memory*. ↑Speicher, dessen Inhalt nichtflüchtig ist (↑*non-volatile memory*). Im Unterschied zu herkömmlichem ↑Festwertspeicher, sind die ↑Speicherzellen nicht nur einzeln lesbar, sondern können auch während des ↑Programmablaufs im ↑Befehlszyklus der ↑CPU beschrieben werden.
Gängige Vertreter sind ↑Flashspeicher, deren ↑Speicherzellen jedoch nicht in allen Fällen zum Schreiben einzeln adressierbar sind. Darüberhinaus bildet eine einzelne Schreiboperation, auch wenn diese eine einzelne Speicherzelle betrifft, eine ↑Aktionsfolge von Befehlen an den Speicherbaustein, sie wird nicht durch einen elementaren ↑Maschinenbefehl der CPU geleistet. Dies ist nicht zu verwechseln mit ↑NVRAM, dessen Speicherzellen einzelnen im Maschinenbefehl adressierbar sind sowohl zum Lesen als auch zum Schreiben.

Festwertspeicher (en.) ↑*read-only memory*. ↑Speicher, dessen Inhalt nach dem Beschreiben nur noch abgerufen, aber nicht mehr verändert werden kann (in Anlehnung an den Duden).

fetch policy (dt.) ↑Ladestrategie.

fetch-execute-cycle (dt.) ↑Abruf- und Ausführungszyklus.

fiber (dt.) ↑Faser.

fibril (dt.) ↑Fäserchen.

FIFO Abkürzung für (en.) *first in, first out*. Bezeichnung für ein Vorgehen bei der Organisation eines ↑Puffers und Verarbeitung der darin gespeicherten ↑Daten. In Bezug auf die Reihung handelt es sich um eine zu ↑FCFS analoge Methode, die allerdings weder die ↑Planung von ↑Aufgaben noch die Bildung einer Reihenfolge von ↑Prozessen bezweckt.

file (dt.) ↑Datei.

file name (dt.) ↑Dateiname.

file system (dt.) ↑Dateisystem.

file tree (dt.) ↑Dateibaum.

filing (dt.) ↑Ablage.

filing system (dt.) ↑Ablagesystem.

Firmware Kunstwort aus (en.) *firm* (fest) und (en.) *ware* (Ware, Produkt). Bezeichnung von Software, die in ↑Festwertspeichern liegt und dort auch direkt zur Ausführung kommt. Ursprünglich waren damit nur ↑Mikroprogramme gemeint. Erst mit der Einführung von Mikroprozessoren (ab 1971) kamen „gewöhnliche“ ↑Programme hinzu, beispielsweise das ↑BIOS oder ein ↑BDOS.

first fit (dt.) erstbeste Passung, ↑Platzierungsalgorithmus.

first-class object (dt.) ↑Objekt erster Klasse.

first-come, first-served (dt.) ↑Windhundprinzip, Abkürzung ↑FCFS.

fixed-priority scheduling (dt.) ↑Ablaufplanung von ↑Echtzeitprozessen mit fester ↑Priorität; eine bestimmte Form von ↑Vorrangplanung. Damit sehr eng verbunden ist die ↑deterministische Planung von Prozessen, die für gewöhnlich ↑periodische Aufgaben unterbrechbar (↑*preemptive*) zu bewältigen haben. Die Festlegung der jeweiligen Priorität geschieht statisch (↑*off-line scheduling*), greift auf ↑Vorwissen zurück. Dazu kommen analytische Verfahren (↑RM, ↑DM) zum Einsatz, die vor ↑Laufzeit der durchzuführenden ↑Aufgaben den dafür zu befolgenden ↑Ablaufplan liefern.

Die ↑Prioritätszuweisung ist jedoch nur ein, und zwar der statische Aspekt des Verfahrens. Demgegenüber geschieht die Aufgabendurchführung immer mit einem dynamischen Hintergrund, und zwar durch Prozesse, deren einzelne Priorität die der durch sie jeweils geleisteten Aufgabe entspricht. In dieser Phase laufen keine Berechnungen mehr ab, um die Priorität eines Prozessen den jeweils gegebenen Umständen anzupassen (↑*static scheduling*). Faktisch trifft der ↑Planer zur ↑Laufzeit des Prozesssystems keine strategisch relevanten Entscheidungen mehr. Seine Hauptfunktion besteht stattdessen darin, entsprechend der vor Laufzeit erfolgten Berechnungen zur Erstellung des ↑Ablaufplans und der Prioritätszuweisung rechtzeitig den ↑Umschalter zu aktivieren. Dies schließt die Behandlung von ↑Prozessblockaden ein, wie auch die ↑Bevorrechtigung eines (im Vergleich zum laufenden Prozess) höher priorisierten Prozesses, sobald dieser (synchron vom laufenden Prozess selbst oder asynchron als Folge einer ↑Unterbrechungsanforderung) zur ↑Einlastung bereitgestellt wird.

Gängig ist ein tabellenbasiertes Verfahren. Ein Tabelleneintrag verweist dann auf den ↑Prozesskontrollblock des Prozesses, der eine bestimmte Aufgabe abzuarbeiten hat. Im einfachsten Fall kann die ↑Prozesstabelle selbst Gegenstand des Verfahrens sein. Bei solch einer Umsetzung spiegelt die Priorität eines Prozesses die Position des ihn betreffenden Eintrags in der Tabelle wider, wobei die Priorität von vorne nach hinten abnimmt. Für die gebräuchliche Implementierung dieser Tabelle als ↑Feld, entspricht die Position eines Tabelleneintrags einem Feldindex, dessen Wert damit gleichsam einen Prioritätswert repräsentiert: je kleiner dieser Indexwert, desto höher die Priorität das darüber indizierten Prozesses.

Um nun mit solch einer Organisation bereitgestellter Prozesse immer denjenigen mit der

höchsten Priorität auszuwählen und zu bedienen, wenn nämlich wegen einer Prozessblockade eine Nachfolge für den aktuellen Prozesses zu finden ist, genügt beispielsweise eine Zählschleife (`for`-*loop*), und zwar mit dem Feldindex aufsteigend ab der Stelle, die dem zur Zeit noch laufenden Prozess entspricht. Dabei ist die mögliche Bevorrechtigung eines bereitzustellenden Prozesses zu beachten. In solch einer Situation rutscht die Startposition für die Nachfolgesuche automatisch weiter zurück: sie nimmt einen kleineren Wert für den Feldindex ein, wenn der bereitzustellende Prozess eine höhere Priorität besitzt als der laufende Prozess. Wird ein Prozess bevorrechtigt (↑*full preemptive*) behandelt, verdrängt er den laufenden Prozess vom ↑Prozessor. Der dann laufende Prozess hat eine höhere Priorität als sein Vorgänger, damit beginnt die nächste Nachfolgesuche mit einem kleineren Indexwert.
Aus all dem geht hervor, dass der Berechnungs- und Zeitaufwand bei der Nachfolgesuche durch die feste Tabellengröße nach oben begrenzt ist (↑WCET). Darüberhinaus hat die Bereitstellung eines nicht zu bevorrechtigenden Prozesses einen konstanten Aufwand, denn dies bedeutet lediglich, in dem betreffenden Kontrollblock den ↑Prozesszustand auf <u>bereit</u> zu setzen. Demgegenüber ist der Aufwand allerdings höher, aber immer noch nach oben beschränkt (WCET), wenn die Bevorrechtigung geschieht: der laufende Prozess wird bereitgestellt und ein ↑Prozesswechsel ist durchzuführen.
Die Alternative zum Feld, also zu einer statischen Datenstruktur, ist die Verwendung von dynamischen Datenstrukturen. Hierzu kann eine gewöhnliche ↑Schlange Verwendung finden, wobei spätestens zur Prozessorzuteilung die Reihung der Einträge, gemeinhin die Prozesskontrollblöcke, stimmig zu den Prozessprioritäten sein muss: am Kopf der Schlange steht der Prozess mit der höchsten Priorität, ihm folgen Prozesse mit jeweils absteigender Priorität. Um diese Reihung sicherzustellen, bieten sich zwei Herangehensweisen an.
Der eine Ansatz reiht die Prozesse entsprechend ihrer Priorität immer in dem Moment, wenn ein neuer Eintrag in die Schlange (*enqueue*) geschieht. Damit fällt bei jeder Eintragung ein Such- und Einsortieraufwand an. Wird davon ausgegangen, dass die Anzahl der zu einem Zeitpunkt möglichen Prozesse normalerweise begrenzt ist, ergibt sich auch hier eine obere Schranke (WCET) — und zwar in etwa entsprechend der Feldvariante. Demgegenüber ist der Aufwand, um die Nachfolge eines Prozesses zu bestimmen, sehr gering: diese ist immer der Prozess am Schlangenanfang. Auch im Falle der Bevorrechtigung eines eintreffenden Prozesses geschehen in Bezug auf die Schlange wenig aufwendige Dinge: der Kontrollblock des laufenden Prozessen kommt einfach an den Anfang der Schlange. Hier besteht der eigentliche Aufwand im Prozesswechsel. Problematisch ist vielmehr die Tatsache, dass jeder einzelne Vorgang, um den Kontrollblock eines nicht zu bevorrechtigenden Prozesses in der Schlange zu platzieren, vergleichsweise aufwendig ist.
Anders wäre es, wenn ein solcher Kontrollblock immer gemäß der sonst für eine Schlange üblichen Reihungsdisziplin behandelt wird, nämlich ↑FIFO. Damit ist der Aufwand, um einen Prozess bereitzustellen, ähnlich gering wie in der Feldvariante. Allerdings muss dann in dem Moment, wenn der laufende Prozess den Prozessor abgeben möchte, die zu den Prozessprioritäten stimmige Reihung der in der Schlange stehenden Kontrollblöcke nachgezogen werden: die komplette Schlange ist umzusortieren. Der dafür anfallende Aufwand schlägt jedoch immer nur bei freiwilliger Abgabe des Prozessors zu Buche, das heißt, bei einer Prozessblockade. Trifft ein zu bevorrechtigender Prozess ein, kommt der Kontrollblock des laufenden Prozesses einfach an den Anfang der Schlange (↑LIFO) und es geschieht der Prozesswechsel.
Der Aufwand für die Umsortierung der Schlange lässt sich allerdings verstecken, wenn die dafür erforderlichen Schritte im Hintergrund geschehen können. Erreichen ließe sich dies beispielsweise durch einen dedizierten ↑Rechenkern, der parallel zu den auf ihren eigenen (anderen) Rechenkernen stattfindenden Prozessen die ↑Umplanung der in der Schlange stehenden Prozesse vornimmt. Die entsprechende ↑Systemfunktion greift bei Bedarf ins Geschehen ein und stellt die Schlange stimmig zu den Prioritäten der verbuchten Einträge um. Eine solche, durch die ↑Rechnerarchitektur des Prozessors ermöglichte Maßnahme bedingt dann jedoch eine andere ↑Synchronisierung der auf der gemeinsamen dynamischen Datenstruktur (↑*shared variable*) der Schlange definierten Operationen. Da diese Operationen aber in Anbetracht be-

vorrechtigter Prozesse bereits auch ohne solche architektonischen Hilfsmittel synchronisiert ablaufen müssen, sind die „neuralgischen Punkte“ schon bekannt und abgesichert gegenüber Zugriffe (pseudo-) ↑gleichzeitiger Prozesss als Folge von Unterbrechungsanforderungen.

flag bit (dt.) ↑Markierungsbit.

Flankensteuerung (en.) ↑*edge-triggered interrupt*. Auslöser für die ↑Unterbrechung ist ein Wechsel des Pegelstands auf der Signalleitung (↑*interrupt line*) zur ↑CPU, hervorgerufen durch die ↑Peripherie. Der Impuls zu diesem Wechsel ist nur lang genug, um von der CPU als ↑Unterbrechungsanforderung erkannt zu werden. Gängig ist die Zwischenspeicherung (↑*latch*) des Signals, da die CPU normalerweise nur zwischen zwei Durchläufen des ↑Abruf- und Ausführungszyklus, also nach der Ausführung von einem ↑Maschinenbefehl, den Unterbrechungszyklus fährt. Jedoch erkennt die CPU nur höchstens einen Impuls pro Durchlauf, wiederholte Impulse (*reassertion*) gehen unerkannt verloren. Eine ↑Unterbrechungssperre verlängert dieses Intervall und erhöht die Wahrscheinlichkeit verpasster Unterbrechungsanforderungen. Dieser Aspekt ist besonders virulent bei Mitbenutzung der Störleitung (*interrupt sharing*), wenn nämlich mehr als ein ↑Peripheriegerät entweder direkt oder indirekt, im letzteren Fall durch einen ↑PIC, an die CPU angeschlossen werden soll. Wiederholte Impulse auf der Störleitung sind durch die verschiedenen und voneinander unabhängigen Peripheriegeräte in solch einer Konstellation Normalität. Damit ist der flankengesteuerte Ansatz recht anfällig für den Verlust von Unterbrechungsanforderungen, im Gegensatz zur ↑Pegelsteuerung.

flash memory (dt.) ↑Flashspeicher.

Flashspeicher (en.) ↑*flash memory*. ↑Festspeicher. Besondere Form von ↑EEPROM, bei der die ↑Speicherzellen zwar einzeln (zum Lesen und ggf. auch Schreiben) adressierbar sind, jedoch nicht einzeln überschrieben beziehungsweise gelöscht werden können. Löschen und gegebenenfalls auch Schreiben geschieht blockweise, auf Basis sogenannter Sektoren. Letztere umfassen typischerweise $1/n$, $n = 4, 8, 16, \ldots$, der Speicherkapzität des betreffenden Geräts.

Flattern (en.) ↑*thrashing*. Phänomen andauernder ↑Umlagerung des Inhalts einer bestimmten, und zwar ein und derselben ↑Speicherpartie zwischen verschiedenen Speicherorten beziehungsweise Ebenen in der ↑Speicherhierarchie. Ursprünglich nur ein eine ↑Seite betreffendes Phänomen anhaltender ↑Seitenumlagerung (↑*page thrashing*), aber in gleicher Weise auch eine mögliche Erscheinung bezüglich ein und derselben ↑Zwischenspeicherzeile (↑*cache thrashing*), deren Inhalt anhaltend zwischen verschiedenen ↑Prozessoren hin und her transferiert wird. Ebenso kann die anhaltende Umlagerung ein und denselben Eintrag in einem ↑TLB (bzw. in der ↑Seitentabelle für ein und demselben ↑Adressraum: ↑TLB *thrashing*) oder die ↑Arbeitsmenge ein und desselben ↑Prozesses (↑*process thrashing*) betreffen. Als Effekt zeigt sich ein plötzlicher Leistungsabfall, der aber auch ebenso plötzlich wieder verschwindet. Grundsätzlich liegt im Moment des Leistungsabfalls eine starke Überlastung des betreffenden Speichersystems vor.

Ein solches Verhalten kann zwei Ursachen haben: (1) zu hohe Systemlast bei unzureichend freien speicherbezogenen ↑Ressourcen oder (2) ↑irrige Mitbenutzung der betreffenden Speicherpartie. Für gewöhnlich steht der erste Punkt in engem Zusammenhang einerseits mit der Größe des ↑Hauptspeichers, insbesondere in Bezug auf die Anzahl freier ↑Seitenrahmen (↑*virtual memory*) und andererseits mit Umfang, Aufbau und Struktur des ↑Zwischenspeichers. Ist der Haupt-/Zwischenspeicher zu klein für die jeweils gegebene Systemlast, müssen gegebenenfalls zu viele logische/virtuelle Einheiten (Seite, Zwischenspeicherzeile auf Ebene des ↑Prozessadressraums) auf zu wenig physische Einheiten (Seitenrahmen, Zwischenspeicherzeile auf Ebene des Zwischenspeichers) abgebildet werden.

Der zweite Punkt (↑*false sharing*) ist typisch für ein ↑paralleles Programm, bei dem ↑Daten oder Datenstrukturen, die von ↑Prozessen verschiedener Prozessoren oder ↑Rechenkerne unabhängig voneinander verarbeitet werden, derselben Speicherpartie (Seite, Zwischenspeicherzeile) zugeordnet sind. In logischer Hinsicht, nämlich entsprechend der Rechenvorschrift des parallelen Programms, geschieht keine Mitbenutzung von ↑Variablen durch diese Prozesse.

Indirekt, in physischer Hinsicht teilen sich die Prozesse jedoch sehr wohl dieselbe Speicherpartie, wenn diese eben die ↑Platzhalter der fraglichen Variablen beherbergt. Finden nun die Prozesse auf verschiedenen Prozessoren oder Rechenkernen statt, wird der komplette Inhalt einer solchen Partie (Seite, Zwischenspeicherzeile) bei Lese-/Schreibzugriffen zum jeweiligen Prozessort (dem lokalen Hauptspeicher seines Prozessors beziehungsweise Zwischenspeicher seines Rechenkerns) transferiert.

Exkurs Beispielhaft sei das Phänomen für eine Seite erläutert, und zwar unter der Annahme, dass ↑virtueller Speicher die Grundlage bildet. Verursacht ein Prozess einen ↑Seitenfehler, ist in dem Moment der Hauptspeicher komplett belegt, nämlich kein Seitenrahmen zur Zeit frei, und soll der Prozess für seinen Fortschritt nicht von der Seitenrahmenrückgabe anderer Prozesse abhängig sein, muss das ↑Betriebssystem zur Einlagerung der angeforderten Seite Platz durch Auslagerung einer Seite (desselben oder eines anderen Prozesses) schaffen: es sorgt für die ↑Verdrängung einer Seite aus ihrem Seitenrahmen. Die eben erst verdrängte/ausgelagerte Seite wird (ggf. nach erfolgtem ↑Prozesswechsel) jedoch sofort wieder von ihrem Prozess referenziert, woraufhin erneut ein Seitenfehler auftritt. Ist der Hauptspeicher in dem Moment immer noch komplett belegt, wiederholt sich der Vorgang der Seitenverdrängung: eine eingelagerte Seite (desselben oder eines anderen Prozesses) wird ausgelagert, um die unlängst verdrängte Seite wieder einzulagern, die dann wiederum verdrängt wird, um erneut ausgelagert zu werden und so weiter. Die verdrängte Seite flattert ständig zwischen ↑Vordergrundspeicher und ↑Hintergrundspeicher hin und her, solange kein einziger Seitenrahmen frei ist.
Der Prozess der flatternden Seite kommt nur noch extrem langsam voran, da das Betriebssystem viel mehr Zeit mit Ein-/Ausgabe für die diesen Prozess betreffende und andauernde Seitenumlagerung beansprucht. Erst wenn (ggf. unbeteiligte) Prozesse Seitenrahmen an das Betriebssystem zurückgeben, kann die fortlaufende Umlagerung einer solchen Seite ein Ende finden und der Prozess wieder mit voller Leistung voranschreiten. Das Phänomen kann auftreten, sobald nur eine einzige ↑Betriebsseite eines Prozesses ausgelagert und damit seine Arbeitsmenge auseinandergerissen wird. Es betrifft zumeist auch mehr als einen Prozess und sorgt für starken Leistungseinbruch im gesamten ↑Rechensystem.

Anekdote Ein an einer ↑PDP 11/40 angeschlossenes externes Festplattenlaufwerk, untergebracht in einem eigenen Gehäuse auf Rollen (↑DEC RM03), diente als Hintergrundspeicher. Das Rechensystem wurde durch ↑UNIX betrieben. Im Moment der Überlastung durch anhaltende Transfers zwischen Vorder- und Hintergrundspeicher (↑*swapping*) begann das Gehäuse sich wie von selbst durch den Rechnerraum zu bewegen. Die anhaltenden, schnellen und verschieden langen Vor- und Rückbewegungen der auf einem Arm montierten Lese-/Schreibköpfe des Festplattenlaufwerks erzeugten Schwingungen, die sich auf das Gehäuse übertrugen, das daraufhin selbst in Bewegung geriet. Die „Fahrt“ endete, sobald die ursächlich dafür verantwortliche Überlastung nicht mehr bestand.

FLIH Abkürzung für (en.) *first-level interrupt handler*. Bezeichnung für den ↑Unterbrechungshandhaber erster Stufe. Dieser ↑Handhaber wird direkt durch eine ↑Unterbrechungsanforderung gestartet und beginnt seine Ausführung auf der durch die Hardware (↑CPU, ↑PIC oder ↑Peripherie) bestimmten ↑Unterbrechungsprioritätsebene. Die durch ihn erfolgende ↑Unterbrechungsbehandlung findet grundsätzlich asynchron zum auf ↑Benutzerebene oder ↑Systemebene unterbrochenen ↑Prozess statt.
Der Wirkungskreis des Handhabers ist stark eingeschränkt und erstreckt sich in erster Linie nur auf die Bedienung des Geräts, das die ↑Unterbrechung hervorgerufen hat. Seine ↑Aufgabe ist die explizite Bestätigung der Unterbrechungsanforderung an dem betreffenden Gerät (was bei ↑Pegelsteuerung zwingend erforderlich ist), das Lesen oder Schreiben von ↑Daten über ↑Ein-/Ausgaberegister und, sofern ↑E/A-Aufträge vorliegen, den nächsten ↑Ein-/Ausgabestoß abzusetzen. All diese ↑Aktionen bedeuten ↑programmierte Ein-/Ausgabe (auch wenn für den eventuellen Datentransfer ↑DMA zum Einsatz kam), sie müssen schnell, zügig und vor allem in endlicher Zeit erfolgen, damit der unterbrochene Prozess nur auf das Kürzeste

und darüber hinaus begrenzt verzögert wird.
Insbesondere wegen der Asynchronität zu Prozessen der Systemebene ist beim Aufruf einer ↑Systemfunktion durch den Handhaber größte Vorsicht geboten. Nur wenn alle wettlaufgefährdeten ↑Aktionsfolgen im Betriebssystem (a) jeweils als ↑kritischer Abschnitt ausgelegt und (b) mittels ↑Unterbrechungssperren abgesichert sind, darf ein solcher Aufruf überhaupt in Erwägung gezogen werden. Finden solche Sperren im Betriebssystem nämlich Verwendung, konnte kein kritischer Abschnitt aktiv gewesen sein, wenn der Handhaber zur Ausführung gelangt ist — womit allerdings nicht zum Ausdruck gebracht werden soll, Unterbrechungssperren seien die zu bevorzugenden Mechanismen zur ↑Synchronisation.
Für eine ↑Betriebsart, die auf einen ↑Uniprozessor ausgerichtet ist, wäre mit solchen Sperren ein gangbarer Weg eröffnet — allerdings nur, wenn ein ↑IRQ die Unterbrechungsursache war. In dem Fall ist aber vor Aufruf einer Systemfunktion die Unterbrechungspriorität wieder zu senken, um die ↑Unterbrechungslatenz für Handhaber derselben Prioritätsebene nicht unnötig zu verlängern. Dazu ist auf die zum Zeitpunkt der Handhaberaktivierung gültigen Prioritätsebene zurückzugehen, mit der möglichen Konsequenz, dass derselbe Handhaber, während er noch aktiv ist, durch eine nachfolgende Unterbrechungsanforderung erneut zur Ausführung gelangt (indirekte ↑Rekursion). All diese Reinkarnationen des einen Handhabers verwenden denselben Laufzeitstapel, der niemals vor Erreichen der Abbruchbedingung der Rekursion überlaufen darf. Je tiefer der Handhaber jedoch durch direkte/indirekte Aufrufe von Systemfunktionen ins Betriebssystem eindringt, umso stärker dehnt sich sein Laufzeitstapel aus. Gegebenenfalls bieten CPU oder Betriebssystem sogar nur einen einzigen Laufzeitstapel für alle Handhaber im System, womit die Stapelkapazität sehr schnell an ihre Grenze kommt. Wird diese Grenze überschritten, verursacht dies für gewöhnlich ↑Panik.
Der ↑NMI, der grundsätzlich nicht an der CPU maskiert werden kann, erhöht noch das Risiko eines möglichen Stapelüberlaufs. Ein entsprechender Handhaber wird immer mit der höchsten Priorität ausgeführt, wobei es es hier eben keine Prioritätsobergrenze gibt: gemeinhin kann jeder Handhaber dieses Unterbrechungstyps in eine indirekte Rekursion verfallen. Sollte ein solcher Handhaber eine Systemfunktion aufrufen, hilft für gewöhnlich nur noch ↑nichtblockierende Synchronisation mit wartefreier ↑Fortschrittsgarantie.
Findet ein ↑Multiprozessor Verwendung, gelten alle vorher genannten Aspekte weiterhin. Darüberhinaus muss dann jedoch die Unterbrechungssperre an einer bestimmten CPU insbesondere auch durch Prozesse der Systemebene von jeder anderen CPU aus verhängt werden können. Für gewöhnlich ist dies nicht ohne weiteres aus der Ferne möglich und benötigt entweder Spezialhardware oder einen ↑PIC, dessen IRQ-Ausgang einer Maskierung unterzogen werden kann. Letzteres bedeutet, dass die in dem PIC typischerweise integrierte ↑Leitweglenkung von Unterbrechungsanforderungen (↑*interrupt* ↑*routing*) komplett ab- und wieder anstellbar von jedem ↑Rechenkern aus ist. Aber auch in dem Fall ist jede Unterbrechungssperre immer noch mit dem grundsätzlichen Problem behaftet, Unterbrechungsanforderungen zu unterbinden, deren jeweilige Behandlung in überhaupt keiner Abhängigkeit zum gegenwärtigen Prozess steht und daher auch nebenläufig stattfinden könnte.
Aus all den genannten Gründen ruft ein Unterbrechungshandhaber erster Stufe eine Systemfunktion daher für gewöhnlich niemals direkt auf, sondern indirekt im Rahmen einer nachgeschalteten Unterbrechungsbehandlung (↑SLIH). Diese wird durch den Handhaber nur ausgelöst, das heißt, zur späteren Ausführung hinterlassen, und durch einen ↑Ausnahmezuteiler zu gegebener Zeit gestartet. Der richtige Zeitpunkt zum Starten solcher zurückgestellten Aufrufe (↑DPC) ist dann gegeben, wenn zum Zeitpunkt der Beendigung einer Unterbrechungsbehandlung kein Handhaber der ersten Stufe mehr aktiv und demzufolge der Laufzeitstapel komplett abgebaut ist.

flow control (dt.) ↑Ablaufsteuerung, Flusskontrolle.

Fluchtsymbol (en.) ↑*escape character*. Sonderzeichen, mit dem nachfolgende Zeichen in einer Zeichenkette abweichend vom Normalfall gedeutet werden. Die Bedeutung nachfolgender Zeichen wird dadurch maskiert, daher auch als *Maskierungszeichen* bezeichnet. Beispielsweise bedeutet die Zeichenkombination `^C` das Ende einer Nachricht im ↑ASCII-Zeichensatz

($0x3_{16}$), wohingegen in ↑UNIX mit dieser Kombination einem ↑Prozess das ↑Signal zur Termination zugestellt wird. In beiden Fällen jedoch maskiert das ^-Zeichen die normale Bedeutung des Buchstabens `C`.

flüchtiges Register (en.) ↑*volatile register*. Konzept der ↑Aufrufkonvention: der Inhalt eines solchen Prozessorregisters gilt als unbeständig. Vorkehrungen zur Sicherung und Wiederherstellung des Registerinhalts werden im ↑Unterprogramm nicht getroffen, sondern gegebenenfalls im aufrufenden Programm (*caller-saved*). So definiert beispielsweise ↑cdecl für ↑x86 die Register `EAX`, `ECX` und `EDX` als flüchtig, wobei `EAX` den Funktionsrückgabewert speichert.

FMS Abkürzung für „*FORTRAN Monitor System*". Gilt als erstes wirkliches ↑Betriebssystem, basierend auf Magnetbandgeräte, das automatisch mehr als ein ↑Programm nacheinander im Stapel (*batch*) verarbeiten konnte. Die Programme konnten zudem in ↑FORTRAN formuliert sein und wurden dann vor Ausführung automatisch der ↑Kompilation unterzogen. Erste Installation im Jahr 1955 (IBM 704).

formal parameter (dt.) ↑formaler Parameter.

formaler Parameter (en.) ↑*formal parameter*. Bestandteil der formalen Schnittstelle (Signatur) von einem ↑Unterprogramm. Bezeichner eines ↑Platzhalters zur Übernahme eines zuweisungskompatiblen Werts (↑tatsächlicher Parameter) als Argument eines Unterprogrammaufrufs.

FORTRAN Abkürzung für (en.) „*formula translation*"; Programmiersprache: prozedural, imperativ (1957).

Fortschrittsgarantie (en.) ↑*progress guarantee*. Zusicherung über das Vorankommen eines ↑Prozesses oder Prozesssystems, wobei ein gemeinsames ↑nichtsequentielles Programm die Grundlage bildet. Ursprünglicherweise wird ↑nichtblockierende Synchronisation für die Koordinierung ↑gekoppelter Prozesse angenommen, allerdings muss auch ↑blockierende Synchronisation das Vorankommen dieser Prozesse in bestimmter Form gewährleisten. Typischerweise werden drei Striktheitsgrade unterschieden:

behinderungsfrei (en. *obstruction-free*) Ein einzelner, in Isolation stattfindender Prozess wird seine ↑Aktion oder ↑Aktionsfolge in einer begrenzten Anzahl von Schritten durchführen und beenden. Der Prozess findet isoliert statt, sofern alle anderen Prozesse, die ihn behindern könnten, zeitweilig (z.B. durch die ↑Ablaufplanung) zurückgestellt sind.

sperrfrei (en. *lock-free*) Jeder Schritt eines Prozesses trägt dazu bei, dass die Ausführung des nichtsequentiellen Programms insgesamt voranschreitet. Damit ist systemweiter Fortschritt garantiert, jedoch können einzelne Prozesse dem ↑Verhungern unterliegen. Umfasst Behinderungsfreiheit.

wartefrei (en. *wait-free*) Die Anzahl der zur Durchführung und Beendigung einer Aktion oder Aktionsfolge auszuführenden Schritte durch einen Prozess ist konstant oder zumindest nach oben begrenzt. Damit ist der Fortschritt jedes einzelnen Prozesses garantiert. Umfasst Sperrfreiheit.

Entsprechend des Striktheitsgrads nimmt nicht selten die Schwierigkeit in Entwurf und Implementierung eines nichtblockierend auf Prozesse wirkenden Koordinierungsverfahrens zu. Für gewöhnlich ist Sperrfreiheit in vielen Fällen für ein ↑Betriebssystem noch vergleichsweise einfach umzusetzen, wohingegen wartefreie Lösungen zuweilen sehr herausfordernd sind und auf Konstruktionshilfen in der Software angewiesen sind.

Fortsetzung (en.) ↑*continuation*. Mechanismus zur Übergabe der Kontrolle über den ↑Rechenkern an einen anderen ↑Handlungsstrang. Letzterer ist in dem Fall nicht als ↑Faden implementiert (auch in keiner Variante davon) und besitzt daher keinen eigenen ↑Laufzeitstapel, um *implizit* darauf beim ↑Prozesswechsel den Zustand zur Wiederaufnahme des Programmablaufs sichern zu können. Stattdessen teilen sich alle Handlungsstränge ein und denselben

Stapel und sichern diesen Zustand *explizit* in ein ↑Objekt erster Klasse. Dieses Objekt liegt in einer Form vor, die es erlaubt, es als Funktion aufrufen zu können. Beim Aufruf sichert der Handlungsstrang seinen aktuellen Zustand und gibt sich einen neuen Zustand, den er einem zweiten Objekt dieser Art entnimmt: der Handlungsstrang vollzieht letztlich einen Wechsel zwischen *primitiven* ↑Koroutinen (vgl. S 145). Soweit es den ↑Prozessorstatus betrifft, ist der Zustand nur definiert durch „beständige“ ↑Prozessorregister (↑nichtflüchtiges Register) und insbesondere den ↑Befehlszähler. Damit wird der Handlungsstrang nicht an die Stelle des Funktionsaufrufes zurückkehren, sondern immer an eine Stelle, die durch den Befehlszähler des wiederhergestellten Zustands bestimmt ist.

FPU Abkürzung für (en.) *floating point unit*, (dt.) Gleitkommaprozessor.

fragmentation (dt.) ↑Fragmentierung.

Fragmentierung (en.) ↑*fragmentation*. Zerstückelung von ↑Speicher in kleine freie Abschnitte mit zwischengelagerten belegten Abschnitten. Ein Phänomen, bei dem Speicherplatz ineffizient genutzt wird, wodurch Kapazität oder Leistung und häufig beides verringert werden. Je höher der Zerstückelungsgrad, umso mehr freie Abschnitte liegen vor, umso kleiner sind diese Abschnitte und umso höher ist die Wahrscheinlichkeit, dass die ↑Speicherzuteilung an einen ↑Prozess scheitert. Die nicht zuteilungsfähigen Abschnitte sind letztlich ↑Verschnitt. Ein Problem insbesondere in Bezug auf den ↑Hauptspeicher, aber auch im Zusammenhang mit dem ↑Ablagespeicher.

frame of reference (dt.) ↑Bezugsrahmen, ↑Bezugssystem.

free list (dt.) ↑Freispeicherliste.

FreeBSD Variante von ↑BSD, erste Installation 1993 (Intel 80386). *Freie Software*, bei deren Empfang gleichsam die vollen Nutzungsrechte uneingeschränkt entgegengenommen werden.

Freigabekonsistenz (en.) ↑*release consistency*. Modell der ↑Speicherkonsistenz, für das ein ↑kritischer Abschnitt den Ausgangspunkt bildet. Die Auswirkung jeder innerhalb dieses Abschnitts von einem einzelnen ↑Prozessor auf den ↑Arbeitsspeicher durchgeführte Schreiboperation ist für andere Prozessoren erst nach Verlassen eben dieses Abschnitts sichtbar. Jedoch werden beim Austritt die betreffenden ↑Speicherzellen für alle Prozessoren lediglich wieder zugänglich gemacht. Beim Eintritt in den zugehörigen Abschnitt synchronisieren sich die ↑Prozesse daher auf die Beendigung der Schreiboperationen für diese Zellen. Allgemeiner als ↑Eintrittskonsistenz, schwächer als ↑schwache Konsistenz.

Freispeicherliste (en.) ↑*free list*. Zusammenstellung freier Abschnitte im ↑Hauptspeicher, wobei jeder Abschnitt einen von einem ↑Prozess derzeit nicht genutzten ↑Adressbereich (↑*hole*) entspricht. Die Liste ist in Abhängigkeit vom jeweils gewählten ↑Platzierungsalgorithmus unterschiedlich technisch ausgeprägt, üblich ist jedoch eine *dynamische Datenstruktur*. Darüberhinaus sind zwei Varianten der Abspeicherung dieser Datenstruktur gebräuchlich: getrennt von oder vereint in den durch sie gelisteten freien Abschnitten. Die erste Variante ist zweckmäßig vor dem Hintergrund von ↑Speicherschutz, wenn nämlich der die freien Abschnitte verwaltende Prozess in einem ↑Adressraum residiert, der von dem des einen freien Abschnitt in seinem Adressraum erzeugenden Prozesses getrennt ist (↑Adressraumisolation). Dies ist meist der Fall für das ↑Betriebssystem, das normalerweise die ↑Speicherzuteilung global für jedes ↑Programm im ↑Rechensystem durchführt. Ausnahme davon bildet ein Betriebssystem, dessen Adressraum zugleich ↑realer Adressraum ist (↑*identity mapping*): in diesem Fall können die Knoten der Datenstruktur zur Erfassung aller freien Abschnitte des Hauptspeichers in diesen Abschnitten selbst liegen. Diese Repräsentation der Liste ist darüberhinaus typisch für die Verwaltung der freien Abschnitte in einem ↑Haldenspeicher, da hier freie und belegte Abschnitte grundsätzlich demselben Adressraum zugehören. Sie bedeutet aber auch, dass die Größe eines bei der ↑Speicherzuteilung einem Prozess zugewiesenen Abschnitts mindestens

die Größe eines Knotens der Datenstruktur zur Implementierung der Liste freier Hauptspeicherabschnitte haben muss. Für eine einfach ↑verkettete Liste enthält ein solcher Knoten zwei Attribute: die Größe (`unsigned int`: 2 – 8 Byte) des durch ihn repräsentierten freien Abschnitts und die ↑Adresse (`void*`: 4 – 8 Byte) des Folgeknotens, zusammen also 6 bis16 Byte (je nach ↑CPU).

Frist (en.) ↑*deadline*, d_i (absolut) oder D_i (relativ). Bezeichnung für eine ↑Prozessgröße, die den Zeitpunkt festlegt, zu dem die Durchführung einer ↑Aufgabe spätestens beendet sein muss. Ein ↑Termin, der nicht überschritten werden sollte oder darf.

full preemptive (dt.) voll-↑präemptiv. Steigerungsform, die zum Ausdruck bringen soll, dass einem ↑Prozess ein von ihm zurzeit benutztes ↑wiederverwendbares Betriebsmittel *uneingeschränkt* entzogen werden kann, um es einem anderen, zur Nutzung dieses ↑Betriebsmittels bevorrechtigten Prozess zuzuteilen. Die damit einhergehende ↑Umplanung der Betriebsmittelnutzung kann insbesondere *jederzeit* geschehen, das heißt, der ↑Verdrängungsgrad im ↑Betriebssystem beträgt (abgesehen von ↑Kausalitäten) 100 %. Der ↑Informatikfolklore ist auch eine eher abgeschwächte Deutung des Begriffs zu entnehmen, wonach hier Umplanung nur vollumfänglich in Bezug auf bestimmte ↑Systemfunktionen gemeint ist.

full virtualisation (dt.) ↑Vollvirtualisierung.

functional hierarchy (dt.) ↑funktionale Hierarchie.

funktionale Hierarchie (en.) ↑*functional hierarchy*. Gesamtheit von in einer Rangfolge stehenden Ebenen, wobei jede Ebene durch eine bestimmte Menge von Funktionen definiert ist und die Ebenenanordnung eine spezielle ↑hierarchische Struktur widerspiegelt. Der Idee nach stellt eine Ebene in einer solchen Anordnung eine (reale/virtuelle) Maschine für die nächst höhere Ebene zur Verfügung. Im zugehörigen Entwurf wird eine Ebene oberhalb einer anderen Ebene platziert, um zum Ausdruck zu bringen, dass das korrekte Funktionieren ersterer von der Existenz einer korrekten Implementierung der Funktionen letzterer abhängt: dass die obere Ebene die untere „benutzt“ (↑*uses hierarchy*).

GAS Abkürzung für (en.) *GNU assembler*, (dt.) ↑GNU ↑Assemblierer.

Gastbetriebssystem (en.) ↑*guest operating system*. Ein ↑Betriebssystem, das in einer ↑Benutzthierarchie auf einen ↑VMM oder ein ↑Wirtsbetriebssystem aufsetzt. In reiner Ausführung nimmt das Gastsystem diese Benutzung nicht wahr: jede in diesem System verursachte ↑Ausnahme wird vom jeweiligen Wirt abgefangen (↑*trap*) und in funktionaler Hinsicht „transparent“ behandelt. Demgegenüber kann das Gastsystem in der Realisierung einer eigenen ↑Systemfunktion jedoch auch vom Wirtssystem profitieren, allerdings impliziert dies Änderungen im Gastsystem (unreine Ausführung).

gate (dt.) ↑Gatter.

Gatter (en.) ↑*gate*. Schaltkreis, der elementare logische Verknüpfungen realisiert (Duden). Diese logischen Verknüpfungen sind *Boolesche Funktionen*, die eine bestimmte Anzahl von digitalen Eingangswerten auf einen digitalen Ausgangswert abbilden. Einfache Verknüpfungen sind AND ($\wedge$), OR ($\vee$) und XOR ($\oplus$, d.h., exklusiv-oder) mit jeweils zwei Eingängen und NOT ($\neg$) mit nur einem Eingang.
Für die technische Umsetzung solcher Schlatkreise kommen heute (2025) *Transistoren* zum Einsatz. Je nach Art des Schaltkreises werden jeweils 2-5 Transistoren benötigt, um die gewünschte Funktion zu implementieren. Hochgerechnet auf einen modernen ↑Prozessor von heute — ein Apple A8X (ARM64) mit 3 ↑Rechenkernen umfasst um die 3.3 Mrd. Transistoren, ein 12-kerniger IBM POWER8 (↑PowerPC) 4.2 Mrd., ein 22-kerniger Intel Xeon Broadwell 7.2 Mrd. und ein 32-kerniger SPARC M7 mehr als 10 Mrd. — ergeben sich bei einem Durchschnitt von angenommen 3 Transistoren pro Schaltkreis 1.1 bis über 3.3 Mrd. Boolesche Funktionen integriert auf ein und demselben ↑Halbleiterbaustein.

Gatteradresse (en.) ↑*fence address*. Bezeichnung für eine ↑Adresse, die in einem bestimmten ↑Adressbereich eine geschützte Zone im ↑Arbeitsspeicher von einer ungeschützten Zone trennt (↑*fencing*). Ursprünglich bildet ein ↑realer Adressraum die Obermenge für diesen Adressbereich — aber auch ein ↑logischer Adressraum oder ↑virtueller Adressraum kann so in zwei Zonen unterteilt sein, in Abhängigkeit von dem durch das ↑Betriebssystem im Zusammenspiel mit einer ↑MMU jeweils implementierte Modell von dem ↑Adressraum für sich selbst und einem ↑Maschinenprogramm (↑*userland*).

GCC Abkürzung für (en.) *GNU Compiler Collection*, (dt.) ↑GNU ↑C ↑Kompilierer.

GE 645 Großrechner (1967, General Electric Corp.): für ↑Multics modifizierter GE 635 ↑Prozessor mit 36-Bit ↑Wortbreite. Die Veränderungen bezogen sich vor allem auf Erweiterungen und betrafen insbesondere eine ↑Adressbreite von 24-Bits, ↑Schutzringe, ↑seitennummerierte Segmentierung samt ↑Übersetzungspuffer, ein ↑Hauptspeicherzugriffe zählender ↑Zeitgeber und Hardwarelogik zur entfernten Systemkonfigurierung. Damit war das System weitestgehend abwärtskompatibel zum GE 635.

Gebrauchsstück (en.) ↑*utility piece*. Ein im ↑Hauptspeicher von einem ↑Prozess belegter ↑Speicherbereich (↑*page frame*, ↑*main-memory chunk*). Gegenteil zu einem ↑Loch.
Ergebnis der ↑Speicherzuteilung, die ein zur Speicheranforderung passendes Loch im Hauptspeicher gefunden und dieses zum Gebrauch durch den anfordernden Prozess im betreffenden ↑Prozessadressraum zur Verfügung gestellt hat.

Gefahr (en.) ↑*threat*. Möglichkeit, dass einem ↑Prozess oder Wesen seiner Umgebung (innerhalb oder außerhalb des ↑Rechensystems) etwas zustößt, dass ein Schaden eintritt; drohendes Unheil (in Anlehnung an den Duden). Das Möglichsein eines ↑Defekts, der gegebenenfalls einen ↑Fehler verursacht, aus dem schließlich das ↑Versagen eines Systems resultieren kann (↑*fault-error-failure chain*). Beispielsweise kann ein ↑hängender Zeiger (Defekt) durch einen ↑Bitkipper (Defekt) verursacht sein und bei Dereferenzierung (Fehler) im ↑Betriebssystem sodann ↑Panik (Versagen) herbeiführen.

Gefahrenkette (en.) *chain of threats*, ↑*fault-error-failure chain*. Bezeichnung für den Ablauf eines Geschehens, an dessen Anfang ein ↑Defekt steht, der einen ↑Fehler verursacht, aus dem schließlich das ↑Versagen eines Systems von Hard- oder Software hervorgeht. Dabei ist zu beachten, dass nicht jeder Defekt zwingend einen Fehler und nicht jeder Fehler zwingend zum Versagen führt. Ist es jedoch zum Versagen gekommen, wurde dieses durch einen Fehler als Folge eines Defekts verursacht.

gekoppelter Prozess (en.) ↑*interacting process*. Ein ↑gleichzeitiger Prozess, für den in Bezug auf wenigstens einen anderen gleichzeitigen Prozess im selben ↑Rechensystem ↑Nebenläufigkeit <u>nicht</u> gilt. Der betreffende Prozess ist abhängig von einem ↑Ereignis, das durch einen anderen gleichzeitigen Prozess bewirkt werden muss. Hat dieses Ereignis noch nicht stattgefunden, muss der davon abhängige Prozess auf den Ereigniseintritt warten.
Für das Warten gibt es zwei grundsätzliche Verfahrensweisen, die in Abhängigkeit von (a) der ↑Abstraktionsebene, auf die der betreffende Prozess aufsetzt, und (b) der zu erwartenden ↑Wartezeit einzuschlagen sind. Ist von einer unbekannten Wartezeit auszugehen, wird üblicherweise die Wahl auf ↑passives Warten fallen. Dies setzt jedoch Mechanismen zum ↑Prozesswechsel voraus, die, bezogen auf die ↑funktionale Hierarchie, erst ab einer bestimmten Stufe der ↑Betriebssystemebene zur Verfügung stehen. Sind derlei Mechanismen (in Bezug auf eine bestimmte Hierarchiestufe) nicht verfügbar, muss ↑aktives Warten betrieben werden. Diese Wartetechnik kann aber auch unabhängig von diesen Mechanismen praktikabel sein, allerdings nur bei abzusehender kurzer und vor allem endlicher Wartezeit.
Für gewöhnlich ist für die Kopplung der Prozesse wenigstens eine ↑gemeinsame Variable verantwortlich, über die die Prozesse direkt oder indirekt in *Interaktion* treten: der eine Prozess (*reader*) liest einen von einem gemeinhin anderen Prozess (*writer*) zuvor geschriebenen Wert. Dabei bildet das Schreiben — gegebenenfalls sogar eines bestimmten Werts — die

das Ereignis schließlich herbeiführende ↑Aktion. Solch eine ↑Interprozesskommunikation ist ein typisches Beispiel für die Notwendigkeit der ↑Synchronisation gleichzeitiger Aktionen auf einen, mehreren Prozessen gemeinsamen ↑Datenbestand.
Allgemein sind die betreffenden Prozesse über ein ↑gemeinsames Betriebsmittel gekoppelt, auf das sie zugleich zugreifen wollen und das von sehr unterschiedlicher Ausprägung sein kann. Jedoch erfolgt die Verwaltung jedes dieser Betriebsmittel gemeinhin über eine eigene ↑Variable, einen ↑Deskriptor, die neben den *Zustand* gegebenenfalls auch weitere Attribute (z.B. Typ, Besitzer, Zugriffsrecht) festhält. Auf diese Variable, die im Grunde genommen ein bestimmtes Betriebsmittel im ↑Betriebssystem repräsentiert, greifen die gekoppelten Prozesse zugleich zu und koordinieren sich dabei in ihren Operationen.
Im Falle einer gegenseitigen Abhängigkeit gleichzeitiger Prozesse, wenn also wenigstens zwei Prozesse zugleich miteinander (in eben dargestellter Weise) gekoppelt sein können, besteht die Gefahr einer ↑Verklemmung. Einer solchen Gefahr muss durch geeignete algorithmische Formulierungen, das heißt, konstruktive Maßnahmen unbedingt vorgebeugt werden.

Exkurs Als Beispiel für die Verklemmungsproblematik wird ↑wechselseitiger Ausschluss mittels einer ↑Lese-/Schreibsperre betrachtet, hier ein Verfahren, mit dem ein ↑kritischer Abschnitt vor gleichzeitig auf zwei Prozessoren stattfindenden Prozessen geschützt werden soll. Der Grundgedanke dabei ist, dass ein Prozess zunächst die Zulassung zum Eintritt in den kritischen Abschnitt anfordert und dafür eine Reservierung vornimmt. Falls der kritische Abschnitt von einem (anderen) Prozess aber bereits belegt beziehungsweise reserviert ist, wartet der anfordernde Prozess, bis der betreffende Abschnitt verlassen, das heißt, die zugehörige Reservierung zurückgenommen wurde. Die Datenstruktur des für zwei Prozessoren definierten ↑Eintrittsprotokolls (`lockup`) sei wie folgt skizziert:

```
#define NCPU 2                          /* max. # of processors supported */

typedef volatile struct lock {                  /* read/write memory lock */
    bool join[NCPU];                                 /* reservation flags */
} lock_t;
```

Initial verbucht ein diesbezügliches ↑Exemplar für gewöhnlich keine Reservierung eines Prozesses, die Sperre für sich erheben und halten zu wollen:

```
lock_t lock = { false, false };              /* no reservation, initially */
```

Ein naheliegender, allerdings auch sehr problematischer, Entwurf des gegebenenfalls von zwei Prozessen gleichzeitig durchlaufenen Eintrittsprotokolls zeigt folgendes Beispiel:

```
void lockup(lock_t *lock) {          /* acquire read/write lock: deadlock */
    unsigned self = aegis();                      /* read own processor id */

    lock->join[self] = true;                         /* make a reservation */
    while (lock->join[self^1]);      /* spin, if peer holds a reservation */
}
```

Zunächst wird bestimmt, welcher der beiden Prozessoren die „Schirmherrschaft“ (*aegis*) über den den kritischen Abschnitt (nach `lockup`) betreten wollenden Prozess besitzt. Mit der (mittels `aegis`, S. 247) gelieferten ↑Prozessoridentifikation (`self` $\in \{0, 1\}$) wird der für den zugehörigen Prozessor zuständige Merker selektiert und gesetzt, um einem gleichzeitigen Prozess auf dem anderen Prozessor den Wunsch zum Betreten des kritischen Abschnitts anzuzeigen. Anschließend wird geprüft, ob eben ein solcher Prozess selbst bereits den kritischen Abschnitt für sich beansprucht hat. Hierzu wird der für den anderen Prozessor (`self^1`) zuständige Merker abgefragt. Ist auch dieser Merker gesetzt, besteht eine ↑Konkurrenzsituation und der den Abschnitt betreten wollende Prozess wartet (aktiv) auf die Freigabe des Abschnitts durch den anderen Prozess (mittels `unlock`).
Diese Lösung funktioniert und bewirkt keine Verklemmung, wenn sich die beiden gleichzeitigen Prozesse nicht zwischen dem Setzen des eigenen und Abfragen des fremden Merkers überlappen. Die betreffenden Prozesse finden dann zwar gleichzeitig statt, sie operieren je-

doch etwas zeitversetzt. In diesem Fall ist auch in einer eventuellen Konkurrenzsituation für beide Prozesse Fortschritt durch das Eintrittsprotokoll garantiert. Treten beide Prozesse jedoch Anweisung für Anweisung nahezu synchron auf, ist Verklemmung vorprogrammiert:

1. als entscheidend problematische Aktion (↑*race condition*) setzen beide Prozesse zugleich ihren eigenen Merker und
2. stellen dann in der Warteschleife fest, dass der jeweils andere Prozess seinen Merker gesetzt hat — den keiner von beiden aber zurücksetzen wird.

In dieser Situation verweilen beide Prozesse für immer, bis einem von beiden „von außen" der Prozessor entzogen und der zugehörige Merker zurückgesetzt wird. Die Prozesse sind miteinander gekoppelt, ohne dass sie diesen Sachverhalt erkennen und aus dieser ausweglosen Situation von selbst ausbrechen können.
Der Vollständigkeit halber sei nachfolgend noch kurz das ↑Austrittsprotokoll skizziert, das lediglich den Merker zurücksetzt, damit durch Rücknahme der Reservierung die Sperre aufhebt und einen kritischen Abschnitt als nicht belegt auszeichnet:

```
void unlock(lock_t *lock) {                    /* release read/write lock */
    lock->join[aegis()] = false;                    /* cancel reservation */
}
```

Abschließend soll nicht unerwähnt bleiben, dass es für das skizzierte Problem selbstverständlich verklemmungsfreie Lösungen gibt, sowohl für den hier betrachteten Spezialfall von zwei Prozessoren als auch für darüber hinausgehende Fälle. Zentraler Punkt bei dem oben skizzierten Einstiegsprotokoll ist das „programmierte Beharren" auf den Anspruch, den kritischen Abschnitt zu betreten. Keiner der beiden beteiligten Prozesse nimmt in der Konkurrenzsituation diesen Anspruch zurück und verwehrt seinem Konkurrenten dadurch die Möglichkeit, den kritischen Abschnitt zu belegen. Eine Variante, in der die Prozesse einen solchen Verzicht ausüben, zeigt folgendes Beispiel:

```
void lockup(lock_t *lock) {                    /* acquire read/write lock */
    unsigned self = aegis();                     /* read own processor id */

    do {                                /* try to block competing process */
        lock->join[self] = true;                     /* make a reservation */
        if (lock->join[self^1])                /* peer holds reservation? */
            lock->join[self] = false;       /* yes, cancel own reservation */
    } while (!lock->join[self^1]);    /* retry, if peer released the lock */
}
```

Auf den ersten Blick scheint hier kein Unterschied zur vorangegangenen Lösung zu bestehen, da bei schrittweise synchronem Verlauf der gleichzeitigen Prozesse beide in logischer Hinsicht immer noch unaufhörlich in der Schleife verweilen können. Werden jedoch — mit an Sicherheit grenzender Wahrscheinlichkeit auftretende — physikalische Effekte in Betracht gezogen, die zu einer irgendwie gearteten zeitlichen Verzögerung von nur einen der beiden Prozesse führen, ergibt sich ein anderes Bild. Die Anspruchrücknahme erlaubt es dem schnelleren Prozess, dem langsameren Prozess im nächsten Durchlauf zuvorzukommen und den kritischen Abschnitt zu belegen. Ein solcher Effekt wäre beispielsweise in einer ↑Unterbrechungsanforderung begründet. Allerdings ist es höchst fragwürdig, das korrekte Funktionieren des Eintrittsprotokolls (`lockup`) von derartigen physikalisch bedingten Auswirkungen abhängig zu machen. Vielmehr muss die algorithmische Formulierung des Eintrittsprotokolls nachweislich die Korrektheit unabhängig von solchen Effekten zeigen — was andere Lösungen (Algorithmen von Dekker, Peterson oder Kessels: vgl. S. 161ff.) auch tun.

Gemeinkosten (en.) ↑*overhead*. Unkosten, indirekte Kosten, einer ↑Systemfunktion — aber auch allgemein jeder von einem ↑Rechensystem zu erfüllenden ↑Aufgabe. Zuschlag bezüglich Rechenzeit, Energieverbrauch oder Speicherplatz, der bei Durchführung dieser Aufgabe in Kauf zu nehmen ist, um von der dadurch dann bereitgestellten Funktion zu profitieren.

gemeinsame Variable (en.) ↑*shared variable*. Eine mehreren (gleichzeitigen) ↑Prozessen gemeinsame ↑Variable. Die Prozesse sind bestimmt durch ein ↑nichtsequentielles Programm, in dem diese Variable deklariert ist. Für gewöhnlich dient eine solche Variable dem Informationsaustausch zwischen den durch das nichtsequentielle Programm ermöglichten Prozessen.

gemeinsamer Speicher (en.) ↑*shared memory*. ↑Speicher, der von mehr als einen ↑Prozess zugleich genutzt werden kann. Im Falle von ↑Arbeitsspeicher ist typischerweise zu differenzieren, ob die gemeinsame Nutzung ein ↑Textsegment betrifft (↑*shared code*, ↑*shared library*) oder ein ↑Datensegment, bis hin zu einem einzelnen ↑Speicherwort darin (↑*shared data*), zur Grundlage hat. ↑Text ist zur ↑Laufzeit in der Regel nur lesbar, kann also von den Prozessen nicht verändert werden. Ganz im Gegensatz dazu stehen ↑Daten, die zwischen den Prozessen ausgetauscht werden und dazu eine ↑Aktion oder ↑Aktionsfolge zum Lesen und Schreiben von verschiedenen Prozessen ausgehend zugleich auf ein oder einer Folge von Speicherwörtern stattfinden muss. In dem Fall ist ↑Synchronisation zu erzielen, um Datenkonsistenz sicherzustellen.

gemeinsames Betriebsmittel (dt.) ↑*shared resource*. Ein mehreren ↑Prozessen in gleicher oder unterschiedlicher Weise gehörendes ↑Betriebsmittel. So greifen die in Gemeinschaft agierenden Prozesse entweder nur lesend oder nur schreibend oder teils lesend und teils schreibend auf dasselbe Betriebsmittel zu. Die Zugriffe können zeitgleich geschehen (↑*simultaneous process*) oder sie erfolgen zu verschiedenen Zeitpunkten, ohne dass die Prozesse zwingend zugleich im ↑Rechensystem stattfinden müssen.
Jedes Betriebsmittel, das explizit der ↑Interprozesskommunikation dient, fällt in diese Kategorie (↑*shared variable*, ↑*bounded buffer*, ↑*pipe*). Aber auch die ↑CPU oder ↑Peripheriegeräte, beispielsweise ein ↑Zeilendrucker, sind Betriebsmittel, die von mehreren Prozessen gemeinsam, aber nicht immer zeitgleich genutzt werden können.

Gemeinschaftsbibliothek (en.) ↑*shared library*. Bezeichnung für eine ↑Bibliothek, die zu einem Zeitpunkt von mehr als einen ↑Prozess zugleich benutzt wird (↑*shared memory*).

general-purpose operating system (dt.) ↑Universalbetriebssystem.

Gerätedatei (en.) ↑*device file*. Beispiel für eine ↑Pseudodatei, über die einem ↑Prozess im ↑Maschinenprogramm die direkte Interaktion mit dem ↑Gerätetreiber für ein ausgewähltes ↑Peripheriegerät möglich ist. Typisches Merkmal für ein ↑UNIX-artiges ↑Betriebssystem.

Gerätenummer (en.) ↑*device number*. Eine Zahl, die innerhalb eines ↑Betriebssystems ein ↑Peripheriegerät kennzeichnet. In ↑UNIX ein Zweitupel, um (1) die Geräteklasse beziehungsweise den ↑Gerätetreiber (↑*major device number*) und (2) ein Gerät (↑*minor device number*) dieser Klasse zu identifizieren.
Nach außen sichtbar wird ein Gerät (in UNIX) über den ↑Dateinamen einer ↑Gerätedatei, und zwar durch den ↑Namensraum des ↑Dateisystems. Wichtiges Attribut des ↑Indexknotens dieser Datei ist die ein Zweitupel bildende Gerätekennzeichnung, mit deren Hilfe im Verlauf eines ↑Systemaufrufs zur Ein-/Ausgabe — nämlich zum Einleiten (`open(2)`), Durchführen (`read(2)`, `write(2)`) oder Abschließen (`close(2)`) der gewünschten Operation — sowohl der Gerätetreiber als auch das Gerät erreichbar ist. Diese Kennzeichnung wird für gewöhnlich im Rahmen der Systemkonfigurierung vorgenommen, und zwar mit Erzeugung des entsprechenden Indexknotens (`mknod(2)`).

Geräteregister (en.) ↑*device register*. Allgemeine Bezeichnung für die ↑Ein-/Ausgaberegister eines ↑Peripheriegeräts, durch die ein ↑Gerätetreiber mit dem betreffenden Gerät kommunizieren kann. Neben dem Transfer von ↑Daten werden darüber nicht nur Einstellungen am Gerät vorgenommen, sondern auch Zustände des Gerätes abgefragt.

Gerätetreiber (en.) ↑*device driver*. ↑Unterprogramm in einem ↑Betriebssystem, durch das ein bestimmtes ↑Peripheriegerät verwaltet und bedient wird. Nach außen implementiert solch ein Unterprogramm typischerweise eine für eine ↑Klasse von Peripheriegeräten einheitliche

Schnittstelle, nach innen ist es speziell zugeschnitten auf das jeweilige Gerät dieser Klasse.

Exkurs In ↑UNIX-artigen Betriebssystemen hat ein Treiber einen eindeutigen numerischen ↑Namen (↑*major device number*), der zugleich als Index in ein ↑Feld von Treiberzugangspunkten dient. Jeder dieser Zugangspunkte ist durch einen die Treiberschnittstelle festlegenden ↑Verbund definiert. Wesentliche Verbundattribute dabei sind (a) ↑Sprungvektoren, um die ↑Systemfunktionen des Treibers zu erreichen und gegebenenfalls (b) ein ↑Zeiger auf eine all diesen Systemfunktionen gemeinsame Datenstruktur — eine ↑virtuelle Methode ist einer derartigen Auslegung einer Schnittstellenfunktion nicht unähnlich. Je nach Ausgestaltung eines solchen Verbunds ist es darüber beispielsweise möglich, ein Gerät der betreffenden Klasse in Betrieb zu nehmen (`open`) oder außer Betrieb zu setzen (`close`), ↑Ein-/Ausgabe der in einem ↑Puffer vorliegenden ↑Daten zu tätigen, also Informationen zu lesen (`read`) oder zu schreiben (`write`), den Gerätebetrieb zu steuern (`control`) oder eine ↑Unterbrechungsanforderung des Geräts zu behandeln (`treat`).
Auch die Geräte selbst haben (in UNIX) einen eindeutigen numerischen Namen (↑*minor device number*), mit dem aus einem Feld von Gerätezugangspunkten die ebenfalls als Verbund organisierten Attribute eines spezifischen Geräts selektiert werden können. Dadurch ist es dem Treiber einer bestimmten Geräteklasse möglich, zwischen verschiedenen Geräten derselben Geräteklasse zu unterscheiden. Beide Komponenten (*major/minor device number*) zusammen bilden die ↑Gerätenummer.

Gerechtigkeit (en.) ↑*fairness*. Prinzip des Verhaltens von einem ↑Betriebssystem, das jedem ↑Prozess gleichermaßen Fortschritt zusichert. Fairness in der ↑Synchronisation von Prozessen oder der Vergabe von einem oder mehrerer ↑Betriebsmittel an einen Prozess.

gereihte Umlaufsperre (en.) ↑*queued spinlock*. Bezeichnung für eine Sperre (*lock*), die in Reihenfolge der ↑Ankunftszeit eines ↑Prozesses in einer ↑Warteschlange vermerkt (↑FIFO) und befolgt wird (↑FCFS), wobei die Gültigkeit des Vermerks eben dieser Prozess durch Kreiseln (*spin*) anhaltend überprüft; ↑wechselseitiger Ausschluss.
Der Prozess betreibt ↑geschäftiges Warten bis der Sperrvermerk für ihn gelöscht ist. Ein auf Grundlage eines ↑Schlossalgorithmus operierendes Verfahren zur ↑Synchronisation von Prozessen auf einem ↑Multiprozessor oder ↑Mehrkernprozessor. Wie bei der ↑Ticketsperre wird ein an derselben Sperre wettstreitiger Prozess um die Zeit verzögert, die alle vor ihm bereits eingetroffenen Prozesse zum Passieren benötigen (↑*proportional backoff*). Im Unterschied ist die Überprüfung der durch den Sperrvermerk postulierten Wartebedingung durch einen wettstreitigen Prozess jedoch eine ↑Aktion, die keine ↑Interferenz mit den anderen an derselben Sperre wartenden Prozesse verursacht. Allerdings können durch die Abarbeitungsdisziplin der Sperren Interferenzen mit dem ↑Planer entstehen, wenn letzterer eine von FCFS abweichende ↑Planung der Prozesse vorsieht.
Das Verfahren verwendet pro Prozess beziehungsweise ↑Prozessor eine „Karte“, die ihn eindeutig repräsentiert und auf der sowohl sein Sperrvermerk als auch Nachfolger im kritischen Abschnitt notiert ist. Diese Karte kommt im ↑Eintrittsprotokoll auf einen, einer jeden Sperre eigenen ↑Stapel. Liegt in dem Moment bereits wenigstens eine weitere Karte auf dem Stapel, markiert der Prozess seinen angehenden Wartezustand (Sperrvermerk), vermerkt seine Karte als Hinweis auf den Nachfolger des Prozesses, dessen Karte eben noch oben auf dem Stapel lag und achtet darauf (↑*busy waiting*), dass der auf seiner Karte gerade angezeigte Wartehinweis wieder gelöscht wird. Letzteres geschieht im ↑Austrittsprotokoll, wenn der den kritischen Abschnitt verlassende Prozess auf seiner Karte die Karte eines Nachfolgeprozess vermerkt vorfindet. In dem Fall wird der auf der Karte des Nachfolgeprozesses notierte Wartehinweis (Sperrvermerk) gelöscht, so dass der betreffende Nachfolgeprozess sein geschäftiges Warten aufgeben und in den kritischen Abschnitt eintreten kann.

Exkurs Der Stapel ist als einfach ↑verkettete Liste ausgelegt, in die die einzelnen Karten im Eintrittsprotokoll nacheinander eingehängt werden. Dazu und um den Wartezustand wie auch indirekt den Nachfolgeprozess zu vermerken ist eine Karte wie folgt als ↑Verbund re-

präsentiert:

```
typedef volatile struct mark {                          /* queued-lock entry */
    chain_t list;           /* lock list linkage: must be first attribute */
    bool wait;                                                /* lock state */
    volatile struct mark *next;                  /* lock to be served next */
} mark_t;
```

Da jedes ↑Exemplar dieses ↑Datentyps von verschiedenen Prozessen (auf verschiedenen Prozessoren) aus genutzt wird, sind die einzelnen Attribute als flüchtig (`volatile`) auszuweisen, um die permanente Zwischenspeicherung in ↑Prozessorregistern zu unterbinden. Die Sperre selbst ist nur noch durch einen Listenzeiger repräsentiert:

```
typedef struct lock {                                       /* queued lock */
    mark_t *tail;                                     /* list of lock marks */
} lock_t;
```

Im Vergleich zu anderen Sperrverfahren, die eine globale ↑Schlossvariable verwenden, über die sich die betreffenden Prozesse synchronisieren, bringt in dem hier betrachten Verfahren jeder Prozess seine eigene Schlossvariable in das Ein- und Austrittsprotokoll ein. Diese ↑Variable ist ein Attribut des oben skizzierten Kartendatentyps. Entsprechend ist die Sperroperation (`lockup`) um einen zweiten Parameter (`mark`) erweitert:

```
void lockup(lock_t *lock, mark_t *mark) {      /* pausing queued spinlock */
    mark->next = 0;                   /* next lock requester still unknown */
    mark_t *hook = (mark_t *)FAS(&lock->tail, mark);   /* store lock request
*/
    if (hook != 0) {            /* is any other process allready waiting? */
        mark->wait = true;                /* yes, indicate wait condition */
        hook->next = mark;                            /* announce successor */
        while (mark->wait)             /* do I have to wait to be served? */
            ease();                            /* yes, pause for a moment */
    }
}
```

Zunächst wird auf der Karte notiert, dass ein Nachfolgeprozess für den kritischen Abschnitt unbekannt sei. Ein solcher Prozess kommt erst ins Spiel, wenn auf dem Stapel, auf dem diese Karte abzulegen ist, bereits wenigstens eine andere Karte vorliegt. Für das Ablegen auf dem Stapel sorgt ↑FAS. Diese Operation liest (*fetch*) den ↑Zeiger (`tail`) auf die oben auf dem Stapel liegende Karte und definiert (*store*) diesen mit der ↑Adresse der hinzukommenden Karte (`mark`) in einem Schritt (↑*atomic instruction*). Ist der gelesene Wert nicht Null, befindet sich der Prozess in Konkurrenzsituation mit wenigstens einem anderen Prozess, um den kritischen Abschnitt zu durchlaufen. In dem Fall muss der neu hinzugekommene Prozess warten, markiert dazu seinen Wartezustand auf seiner Karte, verknüpft dann seine *Wartekarte* mit der, dessen Zeiger (`hook`) er als Rückgabewert aus FAS erhalten hat und wartet, bis er aufgerufen wird. In der Wartephase gibt er seinem Prozessor den Hinweis, sich „entspannen“ zu können (`ease`, vgl. S. 380). Ist der durch FAS gelesene Wert dagegen Null, befindet sich kein anderer ↑gleichzeitiger Prozess im kritischen Abschnitt — oder auf dem Weg dahin.

Genau genommen baut das Eintrittsprotokoll (`lockup`) zwei einfach verkettete Listen auf. Die eine Liste bildet den mittels FAS entstehenden Kartenstapel, der einerseits anzeigt, ob ein Prozess in die Wartephase eintreten muss (`hook != 0`) und der andererseits den Haken (`hook`) zum Einhängen (der Karte) dieses Prozesses liefert. Letzterer ist der Nachfolgeprozess des Prozesses, dessen Karte oben auf dem Stapel liegt. Die zweite Liste verbucht die Wartekarten in Ankunftszeitreihenfolge (FIFO) ihrer Prozesse und legt damit die Reihenfolge des Eintritts in den kritischen Abschnitt (FCFS) fest. Diese Liste entsteht dadurch, indem die Karte des bald wartenden Prozesses auf der Karte seines Vorgängers vermerkt wird (`hook->next = mark`). Die Sperre (`lock`) enthält damit einen Zeiger auf die Wartekarte des Prozesses, der das Ende (`tail`) der Liste wartender Prozesse markiert. Der Anfang

dieser Liste ist auf der Wartekarte des Prozesses verzeichnet, der den kritischen Abschnitt gerade belegt und beim Verlassen desselben seinem Nachfolger (`next`) durch Löschen des Sperrvermerks Zutritt gewährt.
Auch die Entsperroperation (`unlock`), die das Austrittsprotokoll implementiert, erhält einen zweiten Parameter (`mark`), damit sich ein Prozess, der den (durch `lock` geschützten) kritischen Abschnitt verlässt, gegebenenfalls mit seinem Nachfolgeprozess synchronisieren kann:

```
void unlock(lock_t *lock, mark_t *mark) {      /* release queued spinlock */
    if (mark->next == 0) {                          /* is no one waiting? */
        if (CAS(&lock->tail, mark, 0)     /* yes, am I still the only one? */
            return;                              /* yes, nothing to be done */
        while (mark->next == 0); /* successor announcement still pending? */
    }
    mark->next->wait = false;                        /* release successor */
}
```

Wenn im Moment der Freigabe des kritischen Abschnitts kein Nachfolgeprozess bekannt ist (`mark->next == 0`) und sich zwischenzeitlich auch nicht bekannt gemacht hat, löscht der austretende Prozess seine Karte vom Kartenstapel. Hierzu kommt mit ↑CAS ein atomarer Befehl zum Einsatz, um sicherzustellen, dass die Karte (`mark`) wirklich nur dann gelöscht wird, wenn zwischenzeitlich kein Nachfolgeprozess aufgelaufen ist (↑*lost wake-up*). Enthielt der Stapel nur diese eine Karte (CAS gelang), ist der kritische Abschnitt frei und das Austrittsprotokoll beendet (`return`). Andererseits (CAS scheiterte) ist zwischenzeitlich ein Nachfolgeprozess eingetroffen, dessen Wartekarte zwar auf dem Stapel liegt, der sich aber gegebenenfalls noch nicht als Nachfolger hat erklären können. In dieser ↑Wettlaufsituation wartet der im Austrittsprotkoll (`unlock`) befindliche Prozess darauf, dass sich der Prozess im Eintrittsprotokoll (`lockup`) als Nachfolger bekannt gibt. Letzteres ist geschehen, wenn der den Abschnittseintritt anstrebende Prozess seine Wartekarte auf der Karte des den Abschnitt verlassenden Prozesses hat vermerken können (`lockup`: `hook->next = mark`). Auch hier synchronisiert sich der Prozess durch geschäftiges Warten, und zwar solange wie er auf seiner eigenen, lokalen Karte noch keinen Hinweis auf den bald zu erwartenden Nachfolgeprozess vorfindet (`mark->next == 0`).
In logischer Hinsicht ist diese Wartephase im Austrittsprotokoll durch das Eintrittsprotokoll wohl bestimmt und zudem sehr kurz (`lockup`): dem FAS folgend bis zur Kopfschleife (`while`) drei Anweisungen in C, die für ↑x86 in vier ↑Maschinenbefehle übersetzt werden. In Wirklichkeit kann die ↑Latenz für den aus den kritischen Abschnitt austretenden Prozess jedoch eine unbestimmte oder sogar unbestimmbare Größe annehmen. Dies ist dann möglich, wenn der Prozess im Eintrittsprotokoll ↑Unterbrechungen erfährt oder gar scheitert. Durch Unterbrechungen bedingte Verzögerungen können gegebenenfalls vorgebeugt werden, indem im Eintrittsprotokoll für die betreffende Befehlsfolge eine ↑Unterbrechungssperre erhoben wird — was jedoch zum Verlust von ↑Unterbrechungsanforderungen führen kann, gemeinhin nur für ↑IRQ, nicht aber für ↑NMI praktikabel ist und aus diesen Gründen prinzipiell zu überdenken wäre. Muss aber nun mit Unterbrechungen gerechnet werden, so sind diese zu tolerieren und es es bietet sich nur noch an, wie schon im Eintrittsprotokoll auch hier beim geschäftigen Warten dem Prozessor „Entspannung“ (`ease`, vgl. S. 380) zu bringen.
Trotz dieser im Austrittsprotokoll vorgesehenen aktiven Wartephase von möglicherweise unbestimmter Dauer gibt das Verfahren selbst den Prozessen aber immer noch eine ↑Fortschrittsgarantie. Weder Unterbrechungen noch das Scheitern von Prozessen im Eintrittsprotokoll (`lockup`) sind durch das Sperrverfahren an sich begründet, weshalb — in logischer Hinsicht — auch keine Abstriche an der Fortschrittsgarantie vorgenommen werden können.

geschäftiges Warten (en.) ↑*busy waiting*. Situation, in der ein ↑Prozess durch anhaltendes Abfragen (*polling*) einer ↑Variablen ein bestimmtes ↑Ereignis erwartet. Im einfachsten Fall gleicht der Vorgang einer ↑Verzögerungsschleife, wobei der Prozess jedoch nicht auf den Ablauf einer von ihm selbst festgelegten Zeit wartet (vgl. S. 380), sondern auf eine durch äußere Einflüsse bewirkte Zustandsänderung:

```
inline void probe(some_t *eve, some_t val) {
    while(*eve != val);                          /* state change happened? */
        ease();                                          /* no, relax CPU */
}
```

Sollte die gewünschte Zustandsänderung ausgeblieben sein, versucht sich der Prozess im Schleifenrumpf zu „entspannen“ (`ease`), indem er die ↑CPU in die Lage versetzt, prozessorinterne Optimierungsmaßnahmen durchzuführen (vgl. S. 380). Alternativ unterbricht sich der wartende Prozess nach jedem Abfrageschritt, um einem anderen Prozess zu begünstigen (`favor`). Dies geschieht unter Kontrolle des ↑Planers, für dessen Operationen Wissen über die Ereignisvariable (`eve`) — als Indikator, worauf der Prozess wartet — hilfreich sein kann:

```
inline void probe(some_t *eve, some_t val) {
    while (*eve != val)                          /* state change happened? */
        favor(eve);                                  /* no, relinquish CPU */
}
```

Der wartende Prozess wechselt jedoch nur dann von laufend nach bereit (↑Prozesszustand), wenn es in dem Moment einen bereitgestellten (anderen) Prozess gibt und dieser gemäß ↑Prozesseinplanung keine niedrigere Priorität besitzt. In logischer Hinsicht behält der wartende Prozess den Prozessor, obwohl er dabei selbst keine produktive Arbeit verrichten kann. Behält der Prozess den Prozessor auch in physischer Hinsicht, trägt er, sofern ein ↑Uniprozessor die Grundlage bildet, selbst zur Verlängerung seiner Wartezeit bei, da kein anderer Prozess desselben Prozessors diesen zugeteilt bekommt, um so das erwartete Ereignis herbeiführen zu können. Letzteres trifft auch dann zu, wenn die Prozesseinplanung nach einem ↑Zeitteilverfahren arbeitet: dann ist die Länge des dem Prozess zugebilligten Zeitintervalls maßgeblich dafür, wann ihn ein anderer Prozess verdrängt, der dann eventuell das erwartete Ereignis bewirkt.
Der zuletzt genannte Aspekt macht deutlich, dass ↑aktives Warten nur bedingt zur ↑Synchronisation eines Prozesses auf den Eintritt eines bestimmten Ereignisses brauchbar ist. Die beteiligten Prozesse, einerseits der das Ereignis erwartende und andererseits der das Ereignis anzeigende (externe) Prozess, sollten auf verschiedenen Prozessoren stattfinden. Darüberhinaus sollte der das Ereignis erwartende Prozess durchgehend seinen Prozessor behalten, was für gewöhnlich bedeutet, aktiv mit gesetzter ↑Unterbrechungssperre zu warten.

GID Abkürzung für (en.) ↑*group ID*.

gleichzeitiger Prozess (en.) ↑*simultaneous process*. Bezeichnung für einen ↑Prozess, der gleichzeitig mit wenigstens einem anderen Prozess

(a) gemeinsam im selben ↑Arbeitsspeicher und

(b) auf demselben oder einen anderen ↑Prozessor stattfindet.

Da jeder dieser Prozesse durch eine eigene ↑Aktionsfolge charakterisiert ist, überlappen sich mehrere solcher Folgen in (a) Raum und (b) Zeit.
Ein Prozess, der gleichzeitig mit einem anderen Prozess stattfindet, ist nicht automatisch auch ein ↑nebenläufiger Prozess. Umgekehrt gilt dies aber sehr wohl. Um auch nebenläufig stattfinden zu können, muss der betreffende Prozess unabhängig von ↑Ereignissen sein, die ein anderer Prozess gleichzeitig verursacht. Besteht diese Unabhängigkeit nicht, wird der betreffende Prozess auch als ↑gekoppelter Prozess bezeichnet.

Gleichzeitigkeit (en.) ↑*simultaneity*. Stattfinden zweier ↑Ereignisse zur gleichen Zeit. In physikalischer Hinsicht ist dieses Konzept niemals absolut, sondern hängt von dem ↑Bezugssystem eines Beobachters (↑Prozess) dieser Ereignisse ab. Demnach bedeutet die Simultanität zweiter entfernter Ereignisse Unterschiedliches für zwei verschiedene Beobachter, wenn letztere sich in Bezug aufeinander bewegen (d.h., die Prozesse voranschreiten) und damit verschiedene Standpunkte einnehmen.
Bildlich lässt sich dies durch ein Gedankenexperiment (Einstein) verdeutlichen, bei dem sich

ein Beobachter auf einer bewegenden Plattform (Zug) und ein anderer Beobachter auf einer feststehenden Plattform (Bahnsteig) befindet. Auf der bewegenden Plattform wird mittig ein Lichtblitz ausgelöst, der sich somit sowohl in direkter als auch entgegengesetzter Bewegungsrichtung ausbreitet, bis beide Strahlen die Plattformbegrenzung erreichen. Für den ebenfalls mittig auf der bewegenden Plattform platzierten Beobachter erscheinen die Lichtstrahlen gleichzeitig ihren jeweiligen Endpunkt zu erreichen. Einen anderen Eindruck erfährt der auf der feststehenden Plattform platzierte Beobachter, für den der sich in Fahrtrichtung ausbreitende Lichtstrahl seinen Endpunkt früher erreicht als der sich entgegengesetzt der Fahrtrichtung ausbreitende Lichtstrahl:

- Vom Beobachtungsstandpunkt der sich bewegenden Plattform aus gesehen überwinden beide Lichtstrahlen dieselbe Entfernung und benötigen dafür dieselbe Zeit, sie erreichen ihre Ziele gleichzeitig; beide Ereignisse geschehen zugleich.
- Vom Beobachtungsstandpunkt der feststehenden Plattform aus gesehen überwindet der rückwärtsgerichtete Lichtstrahl eine längere Entfernung als der vorwärtsgerichtete Lichtstrahl, weshalb letzterer weniger Zeit für seinen Weg benötigt als ersterer und sie damit ihre Ziele nicht gleichzeitig erreichen; beide Ereignisse geschehen nacheinander.

Dieses Gedankenexperiment lässt sich auf informatische Belange übertragen. Beispielsweise läuft eine ↑Elementaroperation „von oben betrachtet“ logisch immer in einem Schritt ab, auch wenn diese „von unten gesehen“ aus mehreren ↑Maschinenbefehlen besteht und demzufolge real mehrere Einzelschritte zur ↑Ausführungszeit benötigt. Vor dem Hintergrund ↑gleichzeitiger Prozesse ist damit die ↑Unteilbarkeit der Ausführung einer solchen Operation niemals implizit gegeben. Eine Elementaroperation ist also nicht automatisch auch eine ↑atomare Operation, wohl aber umgekehrt. Je nach ↑Anwendungsfall muss oder muss nicht für eine Elementaroperation die (scheinbare) Simultanität der mit ihren Einzelschritten jeweils gegebenen Ereignissen von Maschinenbefehlsausführungen durch einen ↑Prozessor gelten. Simultanität mehrerer solcher Ereignisse ein und desselben ↑Prozesses ist dann hergestellt, wenn ↑Synchronisation der diese Ereignisse hervorrufenden ↑Aktionen ↑gekoppelter Prozesse erreicht ist.
Scheinbare Simultanität mehrerer zusammenhängender Ereignisse durch Synchronisation zu erreichen und damit die Unteilbarkeit einer ↑Aktionsfolge zu gewährleisten, ist ein Aspekt. Ein anderer Aspekt ist es, gleichzeitige Prozesse durch ↑Simultanverarbeitung stattfinden zu lassen und dadurch überhaupt erst auch Synchronisation gegebenenfalls fordern zu müssen. Im Falle ↑nebenläufiger Prozesse finden die durch sie jeweils hervorgebrachten Ereignisse „von außen betrachtet“ dann zweifelsfrei gleichzeitig statt, auch wenn jeder einzelne davon als ↑sequentieller Prozess agiert. Dabei ist es unerheblich, ob die betreffenden Prozesse durch Mehrfachnutzung (↑*partial virtualisation*) oder Vervielfachung des zugrunde liegenden Prozessors zustande kommen. In beiden Fällen werden die Aktionsfolgen der betreffenden Prozesse (scheinbar) gleichzeitig auftreten können.

GNU Abkürzung für (en.) *GNU is Not Unix*, ein rekursives Akronym. Ursprünglich nur die Bezeichnung für ein Mehrplatz-/Mehrbenutzerbetriebssystem, allgemein jedoch bekannt als Sammlung von (freier) Software, und zwar von Anwendungs- und Dienstprogrammen. Gleichwohl als ↑UNIX-ähnliches ↑Betriebssystem konzipiert, aber auf ↑Mach basierend (auch als „*GNU Hurd*“ bezeichnet). In Entwicklung seit 1984 (Intel 80386), programmiert in ↑Assembliersprache (↑GAS) und ↑C.

GPU Abkürzung für (en.) *graphics processing unit*, ↑Grafikprozessor.

Grafikprozessor (en.) ↑*GPU*. Bezeichnung für einen ↑Prozessor, der ursprünglich nur speziell für die Berechnung von Grafiken ausgelegt war. Allerdings hat sich mittlerweile (2025) auch eine Klasse solcher Prozessoren herausgebildet, die allgemein zur ↑Parallelverarbeitung bestimmter rechenintensiver Probleme jenseits von Grafiken nutzbar sind: beispielsweise der Nvidia Tesla mit 5 760 (K80) oder der Intel Xeon Phi mit 72 realen/288 logischen (Knights Landing) ↑Rechenkernen.

Granularität (en.) ↑*granularity*. Anzahl von Untergliederungen eines Elements (Duden). Beispielsweise die Byteanzahl pro ↑Speicherwort, Speicherwortanzahl pro ↑Zwischenspeicherzeile, Zwischenspeicherzeilenanzahl pro ↑Seite, Seitenanzahl pro ↑Segment oder Segmentanzahl pro ↑Adressraum, wenn insbesondere verschiedene Ebenen in der ↑Speicherhierarchie zusammenhängend betrachtet werden.

granularity (dt.) ↑Granularität.

Grenzpriorität (en.) ↑*limit priority*. Bezeichnung für die ↑Obergrenze der (statischen) ↑Priorität bezogen auf ein bestimmtes ↑unteilbares Betriebsmittel.

Grenzregister (en.) ↑*bounds register*. Schutzvorkehrung mit der die ↑Einfriedung von dem für einen ↑Prozess gültigen ↑Adressbereich bewerkstelligt wird. Spezielle ↑Prozessorregister, die als Paar die untere und obere (d.h., die erste und letzte gültige) ↑Adresse in diesem Bereich festlegen. Der Inhalt des jeweiligen Registers gleicht einer ↑Gatteradresse, das heißt, der ↑Bezugspunkt ist für gewöhnlich ein ↑realer Adressraum.
Im Unterschied zum ↑Schutzgatterregister wird der Adressraum jedoch nicht in nur zwei Gebiete (↑Betriebssystem einerseits, ↑Maschinenprogramm andererseits) unterteilt, in denen abwechselnd zu einem Zeitpunkt immer nur ein Prozess stattfinden kann (↑*uniprogramming*). Stattdessen sind in dem Adressraum mehrere eingegrenzte Gebiete möglich, wovon ein Gebiet für das Betriebssystem und die anderen Gebiete jeweils für ein Maschinenprogramm vorgesehen sind (↑*multiprogramming*). Gleichwohl „erbt“ jedes der beiden Register die typischen Merkmale eines Schutzgatterregisters: der Zugriff darauf ist eine privilegierte Operation (↑*privileged mode*), für einen stattfindenden Prozess sind die gespeicherten Gatteradressen fest und ein ↑verschiebender Lader bringt das jeweilige Maschinenprogramm in den ↑Hauptspeicher.

Großrechner (en.) ↑*mainframe*. Bezeichnung für ein komplexes und umfangreiches ↑Rechensystem, das für gewöhnlich einen zentralen ↑Hauptrechner und verschiedene, spezialisierte ↑Satellitenrechner umfasst.

group ID Abkürzung für (en.) ↑*group identification*.

group identification (dt.) ↑Gruppenkennung.

Grundblock (en.) ↑*basic block*. Abschnitt in einem ↑Programm, der einen linearen ↑Kontrollfluss beschreibt. Linearität des Kontrollflusses bedeutet, dass der betreffende Block fortlaufender Anweisungen genau einen Ein- und einen Ausstiegspunkt aufweist. Dazwischen befindet sich keine Anweisung, die den Kontrollfluss zum Anhalten oder Verzweigen bringt. Stattdessen markiert eine solche Anweisung die Blockgrenze, nämlich den Ausstiegspunkt des einen und, bei einer Verzweigung, den Einstiegspunkt des oder eines nachfolgenden Blocks.

Gruppenkennung (en.) ↑*group identification*. Zeichen zur eindeutigen Identifizierung der Gruppe, der eine durch ↑Benutzerkennung autorisierte ↑Entität angehört; für gewöhnlich eine Nummer. Die Kennung wird bei der ↑Anmeldung initial zugeordnet und kann gegebenenfalls im weiteren Verlauf von einem ↑Prozess für sich selbst geändert werden (`setgid(2)`). Der mögliche ↑Kennungswechsel ist hochgradig abhängig vom ↑Betriebssystem.

Gruppenruf (en.) ↑*multicast*. Bezeichnung für einen Ruf, der an eine ausgewählte Teilmenge, einer Gruppe von ↑Entitäten innerhalb eines bestimmten ↑Bezugssystems geht. Unterscheidet sich vom ↑Rundruf, indem grundsätzlich nicht alle Teilnehmer dieses Systems angesprochen sind.

guest operating system (dt.) ↑Gastbetriebssystem.

GUI Abkürzung für (en.) *graphical user interface*. Fensterbasierte Oberfläche eines ↑Betriebssystems, mit der die Benutzung und Bedienung eines ↑Rechensystems durch grafische Symbole und Steuerelemente (*widgets*) geschieht. Erstmalig um- und eingesetzt für den ↑Xerox Alto.

hängender Zeiger (en.) ↑*dangling pointer*. Bezeichnung für einen ungültigen ↑Zeiger.

Häufung (en.) ↑*burst*. Ansammlung, häufiges Vorkommen von ↑Aktionen oder ↑Ereignissen (in Anlehnung an den Duden). In technischer Hinsicht kann sich diese Anhäufung als Bitbündel, Signalfolge/-block oder in Form plötzlicher Schwingungen äußern. Ein in jeder Hinsicht *stoßartiger Vorgang*, in ↑Betriebssystemen insbesondere auch die gebündelte Ausführung von ↑Maschinenbefehlen einerseits zur Durchführung von Berechnungen und allen damit verbundenen Lese-/Schreiboperationen, um ↑Daten zwischen ↑Hauptspeicher und ↑CPU zu bewegen (↑CPU *burst*) und andererseits zur Kommunikation mit der ↑Peripherie zwecks ↑Ein-/Ausgabe von Daten (↑I/O *burst*). Ebenso ein ↑atomarer Befehl, wenn dieser allein oder in Kombination mit anderen Befehlen dieser Art zur ↑Synchronisation wiederholt zur Ausführung kommen muss (↑*bus-lock burst*).

Halbleiterbaustein (en.) ↑*chip*. Bezeichnung für einen elektronischen Baustein, der aus Halbleitern besteht. Typisches Beispiel dafür ist eine ↑CPU, der ↑RAM oder eine andere Art von ↑Hauptspeicher, sowie eine Einrichtung zur Kommunikation mit einem ↑Peripheriegerät. Der Baustein bildet zugleich das Gehäuse für möglicherweise mehrere ↑Halbleiterplättchen.

Halbleiterplättchen (en.) ↑*die*. Bezeichnung für eine einzelne (a) ungehäuste Einheit eines ↑Halbleiterbausteins und (b) zumeist in rechteckiger Gestalt auf eine ↑Halbleiterscheibe aufgebrachte Formation aus einer Anordnung von Schaltkreisen bestimmter Funktion(en). Letztere (b) werden durch Herausschneiden aus der Halbleiterscheibe zu den Stücken, die als erstere (a) bestimmte funktionale Bestandteile des Halbleiterbausteins bilden.

Halbleiterscheibe (en.) ↑*wafer*. Bezeichnung für eine dünne Scheibe aus Halbleitermaterial, auf die integrierte Schaltungen aufgebracht werden (Duden). Dabei ist eine solche Schaltung für gewöhnlich auf ein ↑Halbleiterplättchen konzentriert, wovon mehreren davon dann in einer regelmäßigen Anordnung über die Scheibe verteilt vorliegen. Die Anzahl dieser Plättchen pro Scheibe hängt ab von der Plättchengröße ($l \times w$, mit l für die Länge und w für die Breite), der Scheibenfäche ($\pi \times r^2$, mit r für den Radius) und der aus Fertigungsgründen vorzusehenden Randbreite bis zur Scheibenkante. Die Flächenparameter (l, w, r) ergeben sich einerseits durch die Funktion der Schaltung und damit aber auch durch die Anzahl von Transistoren, die auf einem Plättchen platziert werden sollen und andererseits durch die Fertigungstechnologie beziehungsweise Strukturgröße (Gatterlänge) eines Transistors, die heute (2025) etwa 14 nm beträgt.

Halde (en.) ↑*heap*. Aufschüttung von zurzeit nicht erwerbbaren Vorräten an ↑Speicher (in Anlehnung an den Duden). Eine dynamische Datenstruktur, die der Speicherung von Datenelementen unterschiedlicher Anzahl und Größe dient. Jedes dieser Elemente ist ↑Exemplar eines bestimmten Datentyps und durch einen ↑Prozess belegt (daher auch nicht mehr erwerbbar).

Haldenspeicher (en.) ↑*heap memory*. Bezeichnung für einen ↑Speicher, der als Haufen (↑*heap*) organisiert ist; eine Ansammlung von Speicherbereichen im ↑Adressraum von einem ↑Maschinenprogramm, die durch einen ↑Prozess in diesem ↑Programm belegt oder belegbar (frei) sind. Ein ↑dynamischer Speicher, der in erster Linie der Verwaltung dynamischer Datenstrukturen dient. Seine Kapazität ist begrenzt, aber bis zu dieser Grenze variabel.

Haltebefehl Bezeichnung für einen ↑Maschinenbefehl, bei dessen Ausführung die ↑CPU bis zur nächsten ↑Unterbrechungsanforderung angehalten wird. Der Befehl kommt für gewöhnlich zur Ausführung, wenn sonst keine direkte Arbeit zur Erledigung mehr ansteht, das heißt, wenn das System sich im ↑Leerlauf befindet. Typischerweise ist dieser Befehl nur im privilegierten ↑Arbeitsmodus ausführbar. Der Versuch, diesen Befehl unprivilegiert auszuführen, führt zur ↑Ausnahmesituation.

Halteproblem (en.) ↑*halting problem*. Bezeichnung einer Fragestellung der Algorithmentheorie zur Endscheidbarkeit, nämiich ob ein Algorithmus konstruierbar ist, der einen beliebigen anderen Algorithmus als Eingabe erhält und feststellt, dass letzterer bei Verarbeitung seiner

eigenen Eingabedaten terminiert. Dieses Problem gilt als *algorithmisch nicht entscheidbar*. Es bedeutet, dass es demzufolge auch kein ↑Programm geben kann, das einen ↑Programmablauf auf seine Beendigung hin prüft. Beispielsweise kann ein ↑Übersetzer im Allgemeinen nicht entscheiden, ob das von ihm übersetzte Programm eine Endlosschleife enthält. Ebenso kann ein ↑Betriebssystem nicht entscheiden, ob ein ↑Prozess den ↑Leerlauf beenden wird.

Haltepunkt (en.) ↑*breakpoint*. Bezeichnung für eine bestimmte Stelle in einem ↑Programm, an der ein damit in Beziehung stehender ↑Programmablauf kontrolliert angehalten werden soll. Für gewöhnlich entspricht diese Stelle der ↑Adresse einer bestimmten Anweisung in dem betreffenden Programm, beispielsweise eine Zuweisungsoperation, eine Fallunterscheidung, ein Schleifeneintritt, -durchlauf oder -austritt oder ein ↑Unterprogrammaufruf oder -rücksprung. Die genaue Bedeutung dieser Stelle hängt jedoch ab von der ↑Abstraktionsebene, der die für die Ausführung des Programms verantwortliche ↑virtuelle Maschine zugeordnet ist.
Auf der ↑Maschinenprogrammebene korrespondiert diese Anweisung zu einem gemeinen ↑Maschinenbefehl, bei dessen ↑Befehlszyklus eine ↑Ausnahme durch die ↑CPU zu erheben ist (*breakpoint* ↑*trap*). Da die dadurch bewirkte ↑Unterbrechung des Programmablaufs normalerweise im ursprünglichen Programm aber nicht formuliert ist, wird der betreffende Maschinenbefehl vor Ausführungsbeginn durch einen ↑Unterbrechungsbefehl ersetzt. Das Setzen, Verwalten und Entfernen dieser Unterbrechungsbefehle ist für gewöhnlich Bestandteil der ↑Fehlerbereinigung. Findet nun ein Programmablauf unter Kontrolle eines ↑Fehleraufspürers statt, so wird letzterer im Rahmen der durch den Unterbrechungsbefehl erzwungenen ↑Unterbrechungsbehandlung automatisch durch das ↑Betriebssystem aktiviert, um die Fehlersuche im unterbrochenen Programmablauf in interaktiver Art und Weise (↑*partial interpretation*) zu ermöglichen.

halting problem (dt.) ↑Halteproblem.

Handhabe (en.) ↑*handle*. Bezeichnung für etwas, was ein auf ein bestimmtes Ziel gerichtetes Vorgehen ermöglicht, erlaubt (Duden). Eine opake ↑Referenz zu einer ↑Entität.

Handhaber (en.) ↑*handler*. Bezeichnung für ein ↑Unterprogramm zur Bedienung oder Steuerung eines Geräts (↑Peripherie) oder Instruments (Hardware, Software), aber auch zur Behandlung eines bestimmten ↑Ereignisses, etwa der Umgang mit einer ↑Ausnahme (↑*trap*, ↑*interrupt*).

handle (dt.) ↑Handhabe.

handler (dt.) ↑Handhaber (dem dt. Patentwesen entnommene fachbegriffliche Übersetzung).

Handlungssperre (en.) *action disable*. Maßnahme zum Sperren einer Handlung, das heißt, einer bestimmten Abfolge von zusammenhängenden, miteinander verketteten ↑Ereignissen oder Vorgängen. Eine ↑Unterbrechungssperre unterbindet die ↑Unterbrechungsbehandlung (sperrt einen ↑FLIH), eine ↑Wiedereintrittssperre wehrt die Weiterverfolgung der begonnenen, dann aber zurückgestellten Unterbrechungsbehandlung ab (sperrt ggf. einen ↑SLIH) und eine ↑Verdrängungssperre setzt die ↑Einlastung des ↑Prozessors durch einen ↑Prozess außerstand. All diese Maßnahmen blockieren einen bestimmten Vorgang für gewöhnlich nur zeitweilig, sie dienen der ↑Synchronisierung von Ereignissen in Bezug auf eine bestimmte ↑Sperrzone — eine Art ↑kritischer Abschnitt, für den ↑wechselseitiger Ausschluss nicht praktizierbar ist.

Handlungsstrang (en.) ↑*action strand*). ↑Aktionsfolge, die der Bearbeitung einer komplexen und zusammenhängen ↑Aufgabe entspricht. Mehrere solcher Stränge können ein und dieselbe Aktionsfolge gemeinsam haben (↑Exemplare desselben ↑Datentyps), nur in Teilen der Aktionsfolge überschneiden (Exemplare verschiedener Typen, aber mit wenigstens einem gemeinsamen ↑Unterprogramm) oder komplett verschiedene Aktionsfolgen ausmachen (Exemplare verschiedener Typen).

hard link (dt.) ↑Verknüpfung.

Harvard architecture (dt.) ↑Harvard-Architektur.

Harvard-Architektur (en.) ↑*Harvard architecture*. Bezeichnung für eine (im ↑Mark I verwirklichte) ↑Rechnerarchitektur mit getrennten Speicher- und Signalpfaden für Anweisungen (↑Text) und ↑Daten — im Gegensatz zur ↑von Neumann Architektur, bei der sich Programmanweisungen und Daten denselben Speicher und dieselben Signalwege teilen. Typischerweise ist durch diese Trennung ein zweigeteilter ↑realer Adressraum des ↑Rechners die Folge: einerseits ein ↑Adressraum für den Text- und andererseits einer für den Datenbereich. In modifizierten Formen einer solchen ↑Architektur ist es nicht ungewöhnlich, auch ↑Ein-/Ausgabe über einen eigenen, weiteren Adressraum zu gestallten (↑*port-mapped* I/O, ↑x86).

Hauptgerätenummer (en.) ↑*major device number*. Auf ↑UNIX zurückgehender darstellbarer abstrakter Begriff, eine Zahl, mit dessen Hilfe die ↑Klasse eines ↑Peripheriegeräts identifiziert wird. Komponente einer ↑Gerätenummer.
Wie ↑Objekte desselben ↑Objekttyps, so besitzen alle Geräte derselben Geräteklasse dieselben (technischen) Merkmale. Diese Geräte unterscheiden sich für gewöhnlich lediglich in ihren ↑Adressen beziehungsweise ↑Geräteregistern, über die sie durch den gemeinsamen ↑Gerätetreiber erreichbar sind. Der Gerätetreiber unterscheidet die von ihm zu bedienenden Geräte dann mit Hilfe einer ↑Nebengerätenummer.

Hauptplatine (en.) ↑*main board*. Bezeichnung für eine ↑Trägerleiterplatte, auf der die zentralen Bauteile von einem ↑Rechner platziert sind. Die einzelnen (oftmals in Sockeln steckenden) Bauteile sind Halbleiterbausteine (*chip*) für ↑CPU, ↑RAM und ↑ROM (jew. in den verschiedensten Ausprägungen), zum Anschluss von ↑Peripherie, sowie Steckplätze für Erweiterungskarten unterschiedlicher Art. Eine mögliche Implementierung der ↑Zentraleinheit eines Rechners..

Hauptprogramm Bezeichnung für ein ↑Programm, das in logischer Hinsicht von keinem anderen Programm aufgerufen wird: es beginnt seine Ausführung, indem es vom ↑Betriebssystem gestartet wird. Technisch geschieht dies in bestimmten Fällen sehr wohl durch einen Aufruf, beispielsweise einen in der Form `main(argc, argv, envp)` bei ↑C. Hier erfolgt der Aufruf durch eine spezielle ↑Aktionsfolge zur Inbetriebnahme (*startup*) des Programms überhaupt, und zwar im Namen von der vom Betriebssystem vorher dazu eingerichteten ↑Prozessinkarnation. Dieser Aufruf bewirkt auch, dass `main()` sehr wohl zurückkehren kann. So wird `main()` letztlich von einer speziellen ↑Mantelprozedur aufgerufen, die zudem für die gegebenenfalls benötigte Parameterversorgung (`argc`, `argv`, `envp`) sorgt. Nach Rückkehr aus `main()` zu dieser Mantelprozedur verlässt der ↑Prozess per ↑Systemaufruf (`_exit(2)`) das ↑Maschinenprogramm, betritt das Betriebssystem und terminiert dort.

Hauptrechner (en.) ↑*host computer*. Rechenanlage, die von einer anderen (für gewöhnlich kleineren) Rechenanlage vorbereitete Aufgaben entgegennimmt, durchführt und nach Erledigung an diese (ggf. sogar eine andere, kleinere Rechenanlage) zur Nachbearbeitung abgibt. Zentrale Komponente eines ↑Großrechners.
Historisch auch als Inbegriff für die ↑Zentraleinheit, die in ↑Stapelverarbeitung die über ↑Satellitenrechner bereitgestellten Aufträge ausführt (↑*remote operation*). An einer solchen Anlage ist, neben der ↑Systemkonsole, für gewöhnlich nur ein schnelles ↑Peripheriegerät angeschlossen, über das ↑automatisierter Rechnerbetrieb und die Entsorgung (d.h., Ausgabe der Berechnungsergebnisse) geschieht. Dieses Gerät war früher (ab Ende 1950) ein Bandlaufwerk, zwischenzeitlich (ab Mitte 1960) ein Wechselplattenlaufwerk und ist heute (2025) zumeist auch ein Steuergerät zur Hochgeschwindigkeitskommunikation.

Hauptspeicher (en.) ↑*main memory*. Bezeichnung für den ↑Speicher in dem ↑Programm liegen muss, damit es von der ↑CPU ausgeführt werden kann. Die für die Ausführung erforderlichen Bestände von ↑Text und ↑Daten des Programms liegen flüchtig (temporär: ↑DRAM,

↑SRAM) oder nichtflüchtig (permanent: ↑ROM, ↑PROM; semi-permanent: ↑EPROM, ↑EEPROM, ↑NVRAM) vor. Als ↑Direktzugriffsspeicher (↑RAM) organisiert, im Allgemeinen als flüchtig ausgelegt verstanden in Form von Schreib-lese-Speicher.

Hauptspeicherfehlzugriff (en.) ↑*mainstore miss.* Fehlzugriff auf eine im ↑Hauptspeicher gewähnte Informationseinheit (↑Maschinenbefehl, ↑Daten). Die für den Zugriff von der ↑CPU im ↑Abruf- und Ausführungszyklus applizierte ↑virtuelle Adresse ist der ↑Teilinterpretation durch das ↑Betriebssystem zu unterziehen. Der Zyklus wird unterbrochen, abgefangen (↑*trap*) und nach erfolgter Behandlung des Fehlzugriffs im Betriebssystem später von der CPU wieder aufgenommen (↑*rerun*). Typisch für einen ↑Seitenfehler — jedoch gibt ein ↑virtueller Adressraum, der Voraussetzung für die Erkennung eines solchen Fehlzugriffs ist, dem Betriebssystem damit einen allgemeinen Mechanismus in die Hand, um gezielt die Kontrolle zurückzugewinnen, wenn ein ↑Prozess einen bestimmten ↑Adressbereich durchstreift. Dazu programmiert das Betriebssystem die ↑MMU, beim Zugriffsversuch auf eine bestimmte ↑Seite oder ein bestimmtes ↑Segment den Prozess zu unterbrechen. Eine übliche Maßnahme ist es, die Seite/das Segment nicht vorhanden erscheinen zu lassen (↑*present bit*). Unterstützt die MMU solch eine Maßnahme nicht direkt, lässt sich das gewünschte Verhalten gegebenenfalls durch eine ↑Behelfslösung herbeiführen, indem im ↑Seitendeskriptor/↑Segmentdeskriptor gespeicherte Attribute zweckentfremdet werden (z.B. alle Zugriffsrechte löschen).

Hauptspeicherstück (en.) ↑*main-memory chunk.* Ein ↑Speicherstück bestimmter Größe im ↑Hauptspeicher (↑*physical address space*), damit insbesondere ein ↑Seitenrahmen, wenn ein ↑seitennummerierter Adressraum für die ↑Prozesse eingerichtet ist oder allgemein ein ↑Segment, sonst.

Hauptsteuerprogramm (en.) ↑*supervisor.* ↑Programm im ↑Betriebssystemkern, das alle wesentlichen Funktionen zur Mehrfachnutzung der zugrunde liegenden (realen/virtuellen) Maschine entsprechend der gewünschten ↑Betriebsart umfasst. Mindestens enthalten im Funktionsumfang sind ↑Prozesseinplanung und ↑Ausnahmebehandlung. Eine typische weitere Funktion ist der ↑Speicherschutz, sofern die bereitzustellende Betriebsart dies erfordert.

heap (dt.) ↑Halde.

heap memory (dt.) ↑Haldenspeicher.

heavy-weight process (dt.) ↑schwergewichtiger Prozess.

Heimatverzeichnis (en.) ↑*home directory.* Bezeichnung für ein ↑Verzeichnis, das den ↑Namenskontext für einen ↑Prozess direkt nach erfolgter Systemanmeldung (↑*login*) bildet. Es ist gleichsam das ↑Arbeitsverzeichnis, in dem der Prozess seine Arbeit aufnimmt. Dieses Verzeichnis ist (unter ↑UNIX) auch häufig die Stelle, an der ein ↑Dienstprogramm unterstützende ↑Daten platziert. Diese in Form einer ↑Datei indirekt in einem weiteren ↑Verzeichnis gespeicherten Daten werden namentlich durch einen Punkt angeführt.

Hertz Maßeinheit der Frequenz, Zeichen: Hz (Duden). Anzahl sich regelmäßig wiederholender Vorgänge pro Sekunde.

heterogeneity (dt.) ↑Heterogenität.

Heterogenität Verschiedenartigkeit, Ungleichartigkeit, Uneinheitlichkeit im Aufbau, in der Zusammensetzung (Duden); ein ↑Architekturmerkmal. Gegenteil von ↑Homogenität.

Heuristik Methode einer Anleitung für ein analytisches Verfahren, das auf Grundlage unvollständiger Informationen und in endlicher Zeit zu brauchbaren Ergebnissen kommt.

hexspeak Bezeichnung für eine auf Hexadezimalziffern beruhende Schreibweise, um vor allem einprägsame „magische Zahlen“ zu konstruieren.

hierarchical structure (dt.) ↑hierarchische Struktur.

hierarchische Struktur (en.) ↑*hierarchical structure*. Anordnung der Teile eines Ganzen, die durch eine Methode der Aufteilung eines Systems in seine Einzelbestandteile und die Art der Relation zwischen Teilepaaren bestimmt ist. Strukturen sind hierarchisch, wenn eine solche Relation $R(\alpha, \beta)$ Ebenen entstehen lässt. Dabei ist Ebene_0 eine Menge von Teilen α, so dass es kein β gibt mit $R(\alpha, \beta)$. Ebene_i, für $i > 0$, ist Menge von Teilen α, so dass gilt: es existiert ein β auf Ebene_{i-1} mit $R(\alpha, \beta)$ und falls $R(\alpha, \gamma)$, dann liegt γ auf Ebene_{i-1}. Dabei sind folgende Arten von R geläufig: „benutzt“, wenn das korrekte Funktionieren von α die Existenz wenigstens einer korrekt funktionierenden Implementierung von β voraussetzt (↑Benutzthierarchie, ↑funktionale Hierarchie); „ruft“, wenn α einen Aufruf an ↑Unterprogramm β enthält (Aufrufhierarchie); „beauftragt“, wenn α einen Auftrag an β abgibt und beide Vorgänge dabei unabhängig von der relativen Geschwindigkeit des jeweils anderen Vorgangs operieren (Prozesshierarchie).

hierarchischer Namensraum ↑Namensraum, der eine ↑hierarchische Struktur aufweist. Die einer solchen Struktur zugrundeliegende, Ebenen entstehen lassende, Relation $R(\alpha, \beta)$ zwischen Teilepaaren α und β bedeutet hier „definiert“: der ↑Namenskontext α bestimmt den Bezugsrahmen, in dem ↑Name β eindeutig ist. Um eine aus vielen Ebenen bestehende Struktur zu bilden, kann der in einem Namenskontext verzeichnete Name seinerseits einen Namenskontext auf tiefer gelegener Ebene bezeichnen. Die sich dadurch ergebende Struktur ist baumartig (↑*file tree*).

Hintergrundrauschen (en.) ↑*background noise*, in anderen Fällen auch als ↑Umgebungsrauschen bezeichnet. Entspricht typischerweise der Gesamtheit aller messbaren Störgrößen beziehungsweise ↑Gemeinkosten in einem ↑Betriebssystem, die im Hintergrund von einem normalen ↑Programmablauf anfallen. Beispiel dafür ist ein ↑Dämon, der in regelmäßigen Abständen bestimmte Verwaltungsaufgaben durchführt (↑*spooler*, ↑*pager*). Ebenso ein im Betriebssystem mitlaufender ↑Planer, der im ↑Zeitteilverfahren arbeitet und gegebenenfalls die ↑Verdrängung von einem ↑Prozess bewirkt. Ferner die durch einen ↑Zeitgeber regelmäßig ausgelöste ↑Systemfunktion zur Bestimmung der ↑Arbeitsmenge eines Prozesses. Jeder weitere Fall einer ↑Unterbrechungsbehandlung macht eine solche Störgröße aus. Die plötzliche Ersetzung einer ↑Seite oder ↑Zwischenspeicherzeile kann einen Fehlzugriff durch einen Prozess auslösen, der die referenzierte ↑Entität eben noch direkt zugreifbar wähnte: die Behandlung des Fehlzugriffs verzögert den Prozess ungewollt. Je nach der ↑Betriebsart fallen Anzahl, Häufigkeit und Höhe dieser Störgrößen und Unkosten verschieden aus. Jedoch lassen sich die verschiedenen Betriebsarten nicht so einfach in Bezug auf ihr „Rauschen“ klassifizieren. Hier müssen im Einzelfall Messungen und Analysen der ↑Laufzeit vorgenommen werden, wenn die Parameter relevant für den Ablauf von einem ↑Maschinenprogramm sind.

Hintergrundspeicher (en.) ↑*secondary storage*. Gegenteil von ↑Vordergrundspeicher. ↑Speicher zur Auslagerung von ↑Programmen und ↑Daten; auch ↑Sekundärspeicher. Peripherer Speicher, der indirekt über Ein-/Ausgabeoperationen durch das ↑Betriebssystem bedient wird. Die Speicherzugriffe geschehen mitlaufend (*on-line*) zu den ↑Prozessen, die diese Zugriffe veranlassen.

hole (dt.) ↑Loch, Lücke, Leerstelle.

hole list (dt.) ↑Löcherliste.

home directory (dt.) ↑Heimatverzeichnis.

homogeneity (dt.) ↑Homogenität.

Homogenität Gleichartigkeit, Gleichmäßigkeit im Aufbau, in der Zusammensetzung (in Anlehnung an den Duden); ein ↑Architekturmerkmal. Gegenteil von ↑Heterogenität.

host computer (dt.) ↑Hauptrechner.

host operating system (dt.) ↑Wirtsbetriebssystem.

HRRN Abkürzung für (en.) *highest response ratio next*. Bezeichnung einer Strategie für die nicht ↑präemptive Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor in einem *Vorrangverfahren*. Aufgaben mit kürzerer ↑Bedienzeit werden bevorzugt, jedoch fließen die ↑Wartezeiten der die Aufgaben jeweils bearbeitenden ↑Prozesse in die Ausarbeitung des ↑Ablaufplans mit ein (↑*ageing*).
Grundlage für die Auswahl der als nächstes zu bearbeitenden Aufgabe ist ein Verhältniswert R (*ratio*), $1 \leq R < \infty$, der wie folgt definiert ist:

$$R = \frac{t_{wait} + t_{burst}}{t_{burst}} = 1 + \frac{t_{wait}}{t_{burst}}.$$

Dabei ist t_{burst} die gemäß ↑SPN abgeschätzte Bedienzeit als Dauer des nächsten ↑Rechenstoßes des betreffenden Prozesses. Die Wartezeit t_{wait} ist, ausgehend von einem bestimmten *Beobachtungszeitpunkt*, der Zeitunterschied zur ↑Ankunftszeit einer Aufgabe, nämlich wenn der dieser Aufgabe entsprechende Prozess auf die ↑Bereitliste kommt:

$$t_{wait} = |p_{observe} - p_{arrive}|.$$

Zur Ankunftszeit gilt für die betreffende Aufgabe $t_{wait} = 0$. Damit wird der dieser Aufgabe entsprechende Prozess mit $R = 1$ auf die Bereitliste gesetzt. Je größer dieser Wert ist, umso stärker ist der Vorrang des betreffenden Prozesses gegenüber anderen Prozessen: die Bereitliste ist nach absteigenden Werten von R sortiert. Ein bereitzustellender Prozess wird daher immer ans Ende der Bereitliste gelangen, da jeder andere auf dieser Liste bereits platzierte Prozess eine Wartezeit $t_{wait} > 0$ besitzt und damit $R > 1$ gilt. Das wiederum bedeutet eine ↑Aktionsfolge zum Einfügen, die bei geeigneter Auslegung der Bereitliste (↑*queue*) mit konstantem Zeitaufwand geschehen kann.
Mit voranschreitender Verweildauer auf dieser Liste nimmt auch die Wartezeit der Aufgabe eines bereitgestellten Prozesses zu, die Bedienzeit dieser Aufgabe bleibt jedoch unverändert. Als Folge vergrößert sich R für jeden dieser Prozesse. Werden fortlaufend weitere Prozesse bereitgestellt, gelangen diese zunächst ans Listenende. In Abhängigkeit von ihrer Bedienzeit können diese Prozesse zu einem späteren Zeitpunkt, während sie warten müssen, nach und nach jedoch vordere Plätze in der Liste erreichen und damit von entsprechend stärkerem Vorrang profitieren. Dies ist typisch für Prozesse mit eher kurzer Bedien- und immer länger werdender Wartezeit.
Für den Beobachtungszeitpunkt, um R für jeden Prozess auf der Bereitliste zu berechnen, gibt es zwei Optionen: dieser Zeitpunkt ist entweder das Moment der ↑Einlastung eines Prozesses oder ein in regelmäßigen Zeitabständen auftretendes ↑Ereignis. In beiden Fällen ist die gesamte Bereitliste zu durchlaufen und gegebenenfalls umzusortieren. Folglich hat die erste Option eine vergleichsweise hohe ↑Einlastungslatenz zur Konsequenz — deren Höhe zudem von der jeweiligen Listenlänge abhängt, damit nicht nur variiert, sondern für gewöhnlich auch nicht vorhergesagt werden kann. Im anderen Fall hängt die Vorgehensweise von der zugrunde liegenden ↑Rechnerarchitektur und deren Nutzung durch das ↑Betriebssystem ab:

Multiplexbetrieb Ist ein ↑Uniprozessor gegeben, wird die Berechnung typischerweise durch einen ↑Zeitgeber ausgelöst und im Rahmen der entsprechenden ↑Unterbrechungsbehandlung durchgeführt. Da auch in dem Fall sowohl der Durchlauf als auch die Neuordnung der Bereitliste zeitlich variiert, muss die Berechnung durch einen ↑Unterbrechungshandhaber zweiter Stufe (↑SLIH) erfolgen, um den jeweils unterbrochenen Prozess nicht zwingend unbestimmt lang zu verzögern. Die dann erforderlichen regelmäßigen ↑Unterbrechungsanforderungen sorgen jedoch für zusätzliches ↑Hintergrundrauschen.

Parallelbetrieb Liegt ein ↑Multiprozessor vor, bietet sich als zusätzliche Variante die Auslagerung dieser Berechnung auf einen dedizierten und nicht für die ein- oder umzu-

planenden Prozesse bestimmten ↑Rechenkern an. Die benötigten zyklischen Unterbrechungsanforderungen gehen alle an diesen Rechenkern und verzögern daher die zu planenden Prozesse nicht.

Der Vorteil dieser jeweils unterbrechungsgesteuerten und im Hintergrund stattfindenden Berechnung besteht darin, dass die im Vordergrund von jedem Prozess selbst zu tätigende Auswahl seines Nachfolgers auf dem Prozessor zeitlich begrenzt und sogar in (nahezu) konstantem Zeitaufwand geschieht: die Einlastungslatenz, das heißt, die Verzögerung des auswählenden Prozesses ist minimal und begrenzt. Dieses Argument ist vielfach entscheidend, weshalb das Verfahren in der Umsetzung typischerweise auch dieser zweiten Option (Hintergrundausführung) folgt.
Mit SPN gemeinsame Merkmale sind (a) die ↑probabilistische Planung, da die jeweiligen Bearbeitungszeiten für gewöhnlich nur abgeschätzt werden können (↑*time series analysis*) und (b) die nicht ↑präemptive Planung: eintreffende Prozesse bewirken den Prozessorentzug ebenso wenig wie ein Prozess, der während seiner Wartezeit den Listenanfang erreicht haben sollte. Im Unterschied zu SPN sind die Prozesse nicht von ↑Verhungern betroffen, weil jeder neu bereitgestellte und dadurch zunächst immer ans Ende der Bereitliste platzierte Prozess mit zunehmender Wartezeit (↑Alterung) an den vor ihm stehenden kürzeren, aber jüngeren Prozessen nach und nach vorbeiziehen kann.

Exkurs Als Erweiterung zu SPN fallen im Wesentlichen in zweierlei Hinsicht Änderungen an. Zum einen ist das Sortierkriterium nicht mehr bloß die Länge t_{burst} des zu erwartenden nächsten Rechenstoßes eines Prozesses, sondern der Verhältniswert R zur Prozesswartezeit t_{wait}. Die Bereitstellung eines Prozesses wird daraufhin gemäß ↑FCFS geschehen können, nur dass die Ankunftszeit des Prozesses festzuhalten ist, um damit zum Beobachtungszeitpunkt seine jeweilige Wartezeit bestimmen zu können. Zum anderen ist ein Unterbrechungshandhaber zweiter Stufe (im Multiplex- oder Parallelbetrieb) erforderlich, der in regelmäßigen Zeitabständen die Bereitliste durchläuft und gegebenenfalls umstellt. Für den ersten Aspekt ergibt sich folgende Bereitstellungsoperation (vgl. SPN, S. 317):

```
void ready(process_t *task) {                /* schedule according to HRRN */
    if (task != being(ONESELF))                                /* not myself? */
        task->time.guess = guess(task);      /* yes, estimate burst length */

    state(&task->mood, READY);                 /* set process ready to run */

    ticket(&task->line, 1);                        /* define initial ratio */
    task->time.ready = timestamp();                 /* remember ready time */
    enqueue(labor(), &task->line.next);            /* append to ready list */
}
```

Der auf zweiter Stufe nachgeschaltete Unterbrechungshandhaber bestimmt für jeden Prozess auf der Bereitliste den Verhältniswert R und ordnet den betreffenden Prozess entsprechend neu ein. Analog zum für ↑RR vorgestellten Vorgehen sei angenommen, dass dieser Handhaber regelmäßig als Folge der Unterbrechungsanforderung der ↑Echtzeituhr durch den ihr zugeordneten (↑RTC-) FLIH als ↑AST ausgelöst wird (vgl. S. 272). Nachfolgende Prozedur implementiert diesen Handhaber:

```
void rtc_slih(data_t none) {      /* real-time clock second-level handler */
    /* if need be, do some additional handling of the clock device */

    hrrn();                                     /* request ready-list update */
}
```

Dieser Unterbrechungshandhaber zweiter Stufe läuft im Hintergrund, asynchron zu den anderen Aktivitäten im ↑Betriebssystem und aktualisiert die Bereitliste gemäß des für jeden darauf befindlichen Prozess neu berechneten Verhältniswerts R. Die Prozedur (`hrrn`), die diesen Listendurchlauf vornimmt, sei wie folgt skizziert:

```
void hrrn() {                        /* update ready list according to HRRN */
    process_t *task = forge(ahead(labor()));           /* read first task */
    if (task) {                                  /* any task ready to run? */
        queue_t list = {0};                  /* newly assembled task queue */
        chain_t *next;                     /* successor task on ready list */
        time_t time = timestamp();    /* reference point for observation time
*/
        do {                                    /* walk through ready list */
            next = chain(&task->line.next);          /* remember successor */
            ticket(&task->line, judge(task, time));        /* assign ratio */
            enlist(&list.head, &task->line);       /* add task to new list */
            if (chain(&task->line.next) == 0)    /* trailing list element? */
                list.tail = &task->line.next;  /* yes, adjust tail pointer */
            task = forge(next);          /* make it a process/task pointer */
        } while (task);              /* update next process, if applicable */
        *labor() = list;                              /* update ready list */
    }
}
```

Bevor der Listendurchlauf startet, wird der für alle Listeneinträge (`task`) identische Beobachtungszeitpunkt (`time`) festgehalten. Die für einen gelisteten Prozess verstrichene Wartezeit ergibt sich relativ zu diesem Zeitpunkt. Verrechnet mit der jeweils erwarteten Rechenstoßlänge wird sodann jeder dieser Prozesse neu bewertet (`judge`) und entsprechend seiner Bewertung in die neue Bereitliste (`list`) einsortiert (`enlist`). Wenn ein so behandelter Prozesseintrag ans Ende dieser Liste gelangt ist, wird der Schlangenendezeiger (`list.tail`) entsprechend aktualisiert. Nach erfolgtem Listendurchlauf geschieht die Aktualisierung der globalen Bereitliste (`labor`), womit schließlich die ↑Umplanung stattgefunden hat.
Die Neubewertung eines jeden Prozesses folgt der oben vorgestellten Berechnung des Verhältniswerts R. Die entsprechende Funktion dazu ergibt sich wie folgt:

```
rank_t judge(process_t *task, time_t time) {   /* value according to HRRN */
    time_t wait = time - task->time.ready;         /* settle waiting time */
    return (wait + task->time.guess) / task->time.guess; /* compute ratio */
}
```

Der berechnete Verhältniswert bestimmt den Rank, den der Prozess auf der Bereitliste erhält. Diesem Rank entsprechend wird der Prozess einsortiert (`enlist`), wozu das Verkettungsglied im betreffenden Prozesskontrollblock zuvor noch zu definieren ist (`ticket`, S. 387).
Es sei nochmals darauf hingewiesen, dass die dieses Verfahren mit sich bringende Umplanung von Prozessen für gewöhnlich im Hintergrund laufend stattfindet, nämlich im Rahmen einer zyklischen Unterbrechungsbehandlung. Auch wenn, wie hier gezeigt, dafür ein Unterbrechungshandhaber zweiter Stufe zum Einsatz kommt, geschieht die Umplanung immer noch im Namen des jeweils unterbrochenen Prozesses. Von der damit einhergehenden Verzögerung kann jeder beliebige Prozess betroffen sein. Daher sollten zur Implementierung der Bereitliste für das hier erläuterte Verfahren Datenstrukturen verwendet werden, die in Algorithmen mit geringem und begrenztem Laufzeitaufwand für den Listendurchlauf resultieren. Die einfachheitshalber hier gewählte Schlange ist mit Bedacht zu verwenden und bietet sich nur bei einer eher kleinen Anzahl von Prozessen an.

HTML Abkürzung für (en.) *hypertext markup language*. Beschreibungssprache, die Hypertextdokumente im *World Wide Web* mithilfe spezieller Auszeichner (*tags*) kodiert.

hybrid address space (dt.) ↑hybrider Adressraum.

hybrider Adressraum (en.) ↑*hybrid address space*. Bezeichnung für einen ↑Adressraum, der so-

wohl ↑realer Adressraum, ↑logischer Adressraum als auch ↑virtueller Adressraum ist (↑*identity mapping*). Er vereint alle Eigenschaften eines logischen/virtuellen Adressraums mit der besonderen (eher ungewöhnlichen) Eigenschaft, einem ↑Prozess den kompletten ↑Hauptspeicher direkt zugänglich zu machen. Der Prozess hat damit direkten Zugriff auf alle freien, aber auch auf alle von allen anderen Prozessen belegten Abschnitte.
Ein solcher Adressraum ist (im Falle von ↑Mehrprogrammbetrieb) nur einem ↑Betriebssystem zuzubilligen — jedoch auch dann bedenklich, da jede ↑Aktion im Betriebssystem direkten Zugriff auf jedes ↑Speicherwort und damit jeden beliebigen Abschnitt von ↑Text und ↑Daten im Hauptspeicher hat.

Hypervisor ↑Programm, das eine ↑virtuelle Maschine steuert und überwacht; entspricht einem ↑VMM, Typ I oder II. Üblicherweise stellt dieses Programm mehrere Exemplare desselben Bautyps einer virtuellen Machine zur Verfügung und verwaltet diese entsprechend (Mehrfachnutzung). Legendarisch als Steuerprogramm für das „Leitsystem“ (↑*supervisor*) von ↑VM/370 bestimmt, woraus sich die Namensgebung ableitet.

I/O Abkürzung für (en.) ↑*input/output*.

I/O *bound* (dt.) ↑E/A-gebunden. Bezeichnung für eine ein-/ausgabeintensive ↑Aufgabe, einen interaktiven ↑Prozess. Gegenteil von ↑CPU *bound*.

I/O *burst* (dt.) ↑Ein-/Ausgabestoß.

I/O *register* Abkürzung für (en.) ↑*input/output register*.

i286 Intel 80286, 16-Bit Mikroprozessor (1982). Führte den Schutzmodus (*protected mode*) für ↑x86-Prozessoren ein, indem ein ↑segmentierter Adressraum den ↑Speicherschutz ermöglichte.

i386 Intel 80386, 32-Bit Mikroprozessor (1985). Nachfolger des ↑i286, wobei der Schutzmodus wahlweise als ↑segmentierter Adressraum oder ↑seitennummerierter Adressraum ausgeführt ist. Darüberhinaus kann die ↑Segmentadressierungseinheit mit der ↑Seitenadressierungseinheit kombiniert werden, um damit ↑segmentierte Seitenadressierung zu ermöglichen.

i8080 Intel 8080, 8-bit Mikroprozessor (1974–1990). Erweiterte und verbesserte Variante des früheren i8008 (1972–1983), dem ersten 8-Bit Mikroprozessor von Intel und Ausgangspunkt für den i8086 (1978–1998), dem ersten 16-Bit Mikroprozessor von Intel. Letzterer führte zur ↑x86-Architektur.

IBM 701 Großrechner (1952), auch *Defense Calculator* genannt: Röhrentechnik, 36-Bit ↑Maschinenwort, 18-Bit breite 1-Adressbefehle, ↑Trommelspeicher. Erster Universalrechner für die Massenproduktion, mit besonderen Merkmalen zur Lösung wissenschaftlicher Problemstellungen und einem ↑Assemblierer (↑symbolischer Maschinenkode).

IBM 709 Großrechner (1958): Röhrentechnik, 36-Bit ↑Maschinenwort, ↑überlappte Ein-/Ausgabe, Ein-/Ausgabekanäle (↑Kanalprogramm).

IBM System/360 Großrechner (1964): 32-Bit ↑Maschinenwort, 24-Bit ↑Adressbreite, byteorientierter ↑Hauptspeicher (8-Bit ↑Byte), mikroprogrammierte ↑CPU, ↑EBCDIC Zeichensatz.

IBM z Systems Großrechner (2008): 64-Bit System, bis zu 141 Haupt- und 640 Hilfsprozessoren, „*zero downtime*“. Volle Rückwärtskompatibilität bis ↑IBM System/360.

identity mapping (dt.) identische Abbildung: ↑hybrider Adressraum.

idle (dt.) ↑Leerlauf.

idle loop (dt.) ↑Leerlaufschleife.

idle process (dt.) ↑Leerlaufprozess.

IDT Abkürzung für (en.) ↑*interrupt descriptor table.*

ILC Abkürzung für (en.) *instruction* ↑*location counter.*

index node (dt.) ↑Indexknoten. Abkürzung ↑*inode.*

Indexknoten (en.) ↑*inode.* ↑Informationsstruktur in einem ↑Dateisystem; ↑Deskriptor einer auf einem ↑Datenträger liegender ↑Datei. Typische Attribute dieser Struktur sind (↑UNIX): Eigentümer (↑UID), Gruppenzugehörigkeit (↑GID), Rechte (lesen, schreiben, ausführen: für Eigentümer, Gruppe und die Welt), Zeitstempel (letzter Zugriff, letzte Änderung), Anzahl der Verweise (↑*hard link*) und Typ. Letzteres Attribut klassifiziert eine Datei als: ↑Verzeichnis, ↑symbolische Verknüpfung, Kommunkationskanal (*pipe*, *named pipe*), Sockel (*socket*) zur ↑Interprozesskommunikation, ↑Gerätedatei, ↑Pseudodatei oder reguläre Datei.
In Abhängigkeit vom Typwert (*tag*) werden bestimmte Strukturelemente unterschiedlich gedeutet (*tagged union*). Im Falle einer regulären Datei etwa führt der Deskriptor Buch über die Dateigröße (Vielfaches von ↑Byte) und jeden ↑Block (in Form der jeweiligen ↑Blocknummer), der zur Speicherung der ↑Nutzdaten verwendet wird. Bezeichnet der Typwert dagegen eine Gerätedatei, speichert der erste Eintrag im ↑Feld von Blocknummern das Zweitupel einer ↑Gerätenummer.

Indexknotennummer (en.) ↑*inode number.* Zahl, die einen ↑Indexknoten im ↑Dateisystem kennzeichnet; eine ↑numerische Adresse. Mit Hilfe dieser Zahl wird der zu verwendende Eintrag in der ↑Indexknotentabelle selektiert, sie ist technisch ein ↑Feldindex und als ↑Adresse nur gültig in Bezug auf den ↑Namensraum dieses einen ↑Dateisystems.

Indexknotentabelle (en.) ↑*inode table.* Zusammenstellung von ↑Indexknoten, listenförmig, pro ↑Dateisystem. Ihre Größe wird bei Formatierung des Dateisystems festgelegt. Jeder Eintrag ist der ↑Deskriptor einer ↑Datei auf dem ↑Datenträger. Die Größe der Tabelle bestimmt damit die maximale Anzahl von Dateien pro Dateisystem.

indivisable resource (dt.) ↑unteilbares Betriebsmittel.

indivisibility (dt.) ↑Unteilbarkeit.

indivisible statement (dt.) ↑unteilbare Anweisung.

Informatikfolklore (en.) ↑*computer science folklore.* Lehrüberlieferung in der Informatik, eine für gewöhnlich auch als „ungeschriebenes Wort“ weitergegebende Feststellung oder Erklärung zu einem Sachverhalt. Nicht selten gibt es keinen oder nur einen lückenhaften Beleg für eine so getroffene Aussage, woraufhin der betreffende Sachverhalt auch kontrovers aufgefasst und falsch verstanden werden kann. Beispiele aus (dem Umfeld) der ↑Systemprogrammierung sind:

- Ein ↑Prozess ist die konkrete ↑Instanziierung eines ↑Programms.
- Der Prozess hat einen geschützten ↑Adressraum.
- Nachteil an einem ↑Semaphor ist, dass der Prozess, der ↑P aufruft, nicht auch derselbe sein muss, der ↑V aufruft.
- Ein Prozess scheitert, wenn er einen ↑Mutex, den er nicht besitzt (d.h., nicht angefordert und belegt hat), dennoch versucht freizugeben.
- Der Mutex ist ein „spezialisierter ↑binärer Semaphor“, mit dem ↑wechselseitiger Ausschluss durch ↑passives Warten betrieben wird.
- Eine ↑Umlaufsperre eignet sich zur ↑Synchronisation, insbesondere wenn ein kurzer ↑kritischer Abschnitt zu schützen ist.
- Vollverdrängende ↑Ablaufplanung von Prozessen (↑*full preemptive* ↑*scheduling*) geschieht nur bei Aktivierung, Blockierung, Signalisierung oder Terminierung einer ↑Aufgabe, bei Freigabe eines ↑Betriebsmittels oder bei Wiederaufnahme eines Prozesses nach Rückkehr aus der ↑Unterbrechungsbehandlung.

- ↑Java bietet ↑Monitorobjekte.
- Bildet ↑virtueller Speicher die Grundlage für einen Prozess im ↑Maschinenprogramm, ist die explizite Rückgabe (`free(3)`) eines zuvor zugeteilt erhaltenen ↑Speicherbereichs (`malloc(3)`) an das ↑Betriebssystem vollkommen unnötig.
- Die ↑Arbeitsmenge eines Prozesses entspricht seiner ↑Residenzmenge.

Derartige Pauschalaussagen sind gemeinhin unzutreffend, sie erfordern eine differenzierte Betrachtung des konkreten Sachverhalts.

information structure (dt.) ↑Informationsstruktur.

information system (dt.) ↑Informationssystem.

Informationsstruktur (en.) ↑*information structure*. Tripel von Typen von Informationsträgern, ihrer Repräsentation und der Menge der auf sie anwendbaren Operationen. Auch zu verstehen als Menge von abstrakten Datentypen einer (realen/virtuellen) Maschine, ein Aspekt in ihrem ↑Operationsprinzip.

Informationssystem (en.) ↑*information system*. Bezeichnung eines in besonderer Weise zur Speicherung, Wiedergewinnung und Verarbeitung von Informationen organisierten Systems (in Anlehnung an den Duden). Typischerweise besteht das System aus einer bestimmten Gruppe von Komponenten oder ↑Entitäten, die zur Erzeugung von Information interagieren.
Je nach Abstraktionsgrad ergeben sich dabei in Bezug auf ↑Rechensysteme teils sehr verschiedene Sichten. Auf *Makroebene* könnten die Entitäten beispielsweise durch ↑Prozesse einer bestimmten ↑Anwendung repräsentiert sein, die möglicherweise über ein ↑Netzwerk von ↑Rechnern samt ↑Peripherie große Bestände von ↑Daten zu verarbeiten haben. Demgegenüber könnten auf *Mikroebene* die einzelnen Funktionseinheiten (z.B. Rechen-, Speicher-, Ein-/Ausgabewerk) eines solchen Rechners, sowie die Kontroll- und Datenwege zwischen den Einheiten, im Vordergrund stehen. Wesentlich dabei ist weniger die Granularität, als vielmehr die Tatsache, dass die Untergliederungen des Systems in einem bestimmten logischen Zusammenhang stehen und die Gesamtfunktion nur durch Interaktion erreicht wird.

inheritance (dt.) ↑Vererbung.

inode Abkürzung für (en.) ↑*index node*.

inode number (dt.) ↑Indexknotennummer.

inode table (dt.) ↑Indexknotentabelle.

input/output (dt.) ↑Ein-/Ausgabe, Abkürzung ↑I/O.

input/output register (dt.) ↑Ein-/Ausgaberegister; Abkürzung (en.) ↑I/O *register*.

instance (dt.) Beispiel, ↑Exemplar; ↑Instanz.

instance variable (dt.) ↑Objektkomponente, auch Mitgliedsvariable (*member variable*). Eine in einer ↑Klasse deklarierte/definierte ↑Variable.

Instanz (en.) ↑*instance*. Ursprünglich der Begriff für einen Verfahrensabschnitt vor einem Gericht. Aus dem Englischen allerdings falsch übersetzt übernommen in die ↑Informatikfolklore als die Bezeichnung für eine einzelne Ausprägung, ein ↑Exemplar aus einer ↑Klasse von ↑Objekten (in Anlehnung an den Duden).

instruction cycle (dt.) ↑Befehlszyklus.

instruction set (dt.) ↑Befehlssatz.

instruction set level (dt.) ↑Befehlssatzebene, auch bezeichnet als (en.) ↑ISA *level*.

instrumentation (dt.) ↑Instrumentierung.

Instrumentierung (en.) ↑*instrumentation*. Bezeichnung für den Vorgang, ein ↑Programm mit speziellen Instrumenten oder Mitteln zu versehen, um dadurch mitlaufend (*on-line*) oder abgetrennt (*off-line*) bestimmte Aussagen über das Verhalten oder Erscheinungsbild eines zugehörigen ↑Programmablaufs treffen zu können. Für gewöhnlich erfolgen hierzu Änderungen in Form zusätzlicher Anweisungen auf der Ebene von ↑Quellmodulen. Aber auch Modifikationen auf der Ebene von ↑Maschinenkode, also direkt im ↑Maschinenprogramm, sind durchaus gebräuchlich. Die durchgeführten Maßnahmen geben Aufschluss über die ↑Performanz, dienen der ↑Profilbildung oder erlauben eine Diagnose möglicher ↑Fehler eines oder mehrerer Programmabläufe. Dabei ist jedoch zu beachten, dass diese Maßnahmen immer auch ↑Gemeinkosten und ↑Profile hervorbringen, die sonst nicht entstehen — sie verfälschen mehr oder weniger stark den „Abdruck“ eines normalen Ablaufs des unbearbeiteten (*raw*) Programms.

integrierter Befehl (en.) ↑*built-in command*. Bezeichnung für einen ↑Auftrag, den der in einem ↑Kommandointerpreter jeweils stattfindende ↑Prozess von selbst und direkt durchführt. Technisch ausgelegt zumeist als ↑Unterprogramm.

Integrität Unverletzlichkeit von Programmtext und -daten, die korrekte Funktionsweise eines Systems gemäß Spezifikation.

Intel 64 Prozessorarchitektur, 64-Bit, ursprüngliche Spezifikation als ↑AMD 64. Zwischenzeitlich zunächst bezeichnet als IA32e, dann als EM64T (*extended memory 64 technology*). Erste Produktfreigabe in 2004 (Intel Xeon „Nocona“).

interacting process (dt.) ↑gekoppelter Prozess.

interarrival time (dt.) ↑Zwischenankunftszeit.

interception (dt.) ↑Abfangung.

interference (dt.) ↑Interferenz.

Interferenz (en.) ↑*interference*. Überlagerung beim Zusammentreffen zweier oder mehrerer Vorgänge, die sich dadurch möglicherweise gegenseitig beeinflussen. Jeder Vorgang entspricht dabei einem ↑Prozess, der wenigstens ein ↑Betriebsmittel mit einen anderen Prozess gemeinsam hat. Teilen sich zwei Prozesse beispielsweise die ↑CPU im Zeitmultiplexverfahren (↑partielle Virtualisierung), kann der eine Prozess durch den anderen verdrängt werden. Der verdrängende Prozess überlagert den verdrängten Prozess und verzögert diesen dadurch. Darf das Betriebsmittel nur exklusiv genutzt werden, ist im Falle mehrerer Prozesse, die gleichzeitig darauf zugreifen, ↑Synchronisation explizit sicherzustellen. Ein Prozess, der ein solches Betriebsmittel hält, beeinflusst andere Prozesse, die dieses Betriebsmittel gleichzeitig beanspruchen und sich synchronisieren müssen, wodurch letztere sich gegenseitig selbst beeinflussen. Spezielle Ausprägung davon ist ein ↑kritischer Abschnitt. Grundsätzlich kann ↑Synchronisierung die ↑Ablaufplanung überlagern, was Störungen im ↑Ablaufplan zur Folge haben kann; genauer: ↑blockierende Synchronisation, die eine Prozessreihenfolge festlegt und dadurch der Entscheidung der Ablaufplanung möglicherweise entgegenwirkt.

internal fragmentation (dt.) ↑interne Fragmentierung.

interne Fragmentierung (en.) ↑*internal fragmentation*. Gesamtheit von ↑Verschnitt verteilt über gegebenenfalls mehrere zugeteilte ↑Speicherstücke. Normalerweise bezogen auf ↑Seiten, betrifft aber jeden Typ ↑Speicherbereich, dessen Größe echtes Vielfaches der Größe eines ↑Byte ist und am Ende ein ↑Loch aufweist. Typisch ist ein solcher Verschnitt wenn ein ↑seitennummerierter Adressraum die Grundlage für die Verwaltung von ↑Arbeitsspeicher bildet. In dem Fall wird die ↑Speicherzuteilung einem ↑Prozess immer einen Abschnitt

zuteilen, dessen Größe Vielfaches der Größe einer Seite ist. Damit kann am Ende der letzten/einzigen Seite des zugeteilten Abschnitts einer linearen Folge von Seiten ein Loch der Größe $hole = size \mod sizeof(page)$ entstehen, wobei $size$ die Größe des angeforderten Speicherbereichs ist. Alle Seiten dieses Abschnitts werden in Folge in den ↑Adressraum des Prozesses platziert, der Prozess erhält damit auch mehr als angefordert zugeteilt. In logischer Hinsicht dürfte dieser Prozess nicht auf den dem Loch entsprechenden Seitenabschnitt zugreifen. Physisch kann er an einen solchen Zugriff allerdings nicht gehindert werden — obwohl er sich in dem Fall fehlerhaft verhalten würde, da er das Loch nämlich nicht kennen dürfte: mit der ↑MMU kann dieser ↑Zugriffsfehler nicht festgestellt werden.

Interpretation Vorgang der Deutung und bedingten Ausführung der Anweisungen in einem ↑Programm durch einen ↑Interpreter. Ausgeführt werden nur gültige Anweisungen. Ob und unter welchen Bedingungen eine Anweisung gültig ist und was eine ungültige Anweisung zur Folge hat, legt das ↑Operationsprinzip der (realen/virtuellen) Maschine fest, die der Interpreter implementiert. Der ↑Befehlssatz dieser Maschine definiert die Menge der allgemein gültigen Anweisungen (d.h., Befehle), aus denen die Programme für diese Maschine formuliert werden. Ungültige Anweisungen werden von der Maschine ignoriert oder bewirken eine ↑Ausnahme, die von der Maschine erhoben wird. Eine solche Ausnahme kann die ↑partielle Interpretation der betroffenen Anweisung bedeuten.

Interpreter (en.) ↑*interpreter*. Soft-, Firm- oder Hardwareprozessor, der die Anweisungen von einem ↑Programm deutet und direkt ausführt. Gegebenenfalls erfolgt zuvor eine ↑Vorübersetzung durch einen ↑Kompilierer, um die ↑semantische Lücke zwischen der Quellsprache, in der das Programm formuliert ist, und der ↑Interpretersprache, in der das zu deutende Programm vorliegen muss, zu verringern und dadurch die Interpretation zu beschleunigen.

interpreter (dt.) ↑Interpreter, Interpretierer; ↑Übersetzer.

interpreter language (dt.) ↑Interpretersprache.

interpreter system (dt.) ↑Interpretersystem.

Interpretersprache (en.) ↑*interpreter language*. Bezeichnung der Programmiersprache für ein ↑Programm, dessen Ausführung durch ↑Interpretation in Software (↑CSIM) bestimmt ist. Beispiele dafür sind BASIC, Forth, Perl, Python oder PHP. Verwandte Form ist die ↑Skriptsprache.

Interpretersystem (en.) ↑*interpreter system*. Prinzip nach dem die ↑Interpretation von einem ↑Programm geschieht, aber auch die dafür (in Hard- und/oder Software) benötigte, abgegrenzte Einheit von Komponenten oder Menge von Funktionen eines ↑Rechensystems. Normalerweise erfolgt die Interpretation *total*, also vollständig durch einen ↑Interpreter, der die (reale/virtuelle) Maschine, für die das Programm bestimmt ist, implementiert. Im Falle einer ↑Ausnahme kann jedoch ein anderer Interpreter helfend eingreifen und teilweise die Programmausführung übernehmen (↑partielle Interpretation).

interprocess communication (dt.) ↑Interprozesskommunikation.

Interprozesskommunikation (en.) ↑*interprocess communication*. Verständigung unter ↑Prozessen eines ↑Rechensystems mithilfe von ↑Daten. Die Kommunikation kann speichergekoppelt (↑*shared memory*) durch ↑gemeinsame Variablen oder nachrichtengekoppelt mittels Botschaftenaustauch (↑*message passing*) geschehen. In beiden Fällen übernimmt jeder der Kommunikationspartner im Moment der ↑Aktion zur Kommunikation eine bestimmte Rolle als ↑gekoppelter Prozess, und zwar Leser oder Schreiber im speichergekoppelten Modell beziehungsweise Sender oder Empfänger im nachrichtengekoppelten Modell.
Die Kopplung der kommunizierenden Prozesse kann eng oder lose ausgelegt sein. Bei einer

engen Kopplung finden die Prozesse zugleich statt (↑*simultaneous process*) und die Kommunikation zwischen ihnen geschieht mitlaufend (*on-line*), effektiv immer durch Lese-/Schreibaktionen bezogen auf den ↑Arbeitsspeicher: Übertragung einer ↑Nachricht durch einen ↑Platzhalter, auf den die kommunizierenden Prozesse direkten Zugriff haben. Demgegenüber wird bei einer losen Kopplung davon ausgegangen, dass die betreffenden Prozesse nicht zugleich stattfinden und vielmehr zeitversetzt ihrer jeweiligen ↑Aufgabe nachkommen. Die Kommunikation zwischen den Prozessen verläuft abgetrennt (*off-line*) von den dafür erforderlichen Aktionen der betroffenen Prozesse. Typisches Beispiel dafür ist die Kommunikation durch eine ↑Datei als Nachrichtenplatzhalter. Gleiches trifft zu auf einen ↑Briefkasten, der ebenfalls als Medium den Kommunikationspartnern zur Verfügung stehen kann.

interrupt (dt.) ↑Unterbrechung, unterbrechen.

interrupt command (dt.) ↑Unterbrechungsbefehl.

interrupt controller (dt.) ↑Unterbrechungssteuereinheit.

interrupt cycle (dt.) ↑Unterbrechungszyklus.

interrupt descriptor table (dt.) ↑Unterbrechungsdeskriptortabelle.

interrupt handler (dt.) ↑Unterbrechungshandhaber.

interrupt handler stub (dt.) ↑Unterbrechungshandhaberstumpf.

interrupt handling (dt.) ↑Unterbrechungsbehandlung.

interrupt latency (dt.) ↑Unterbrechungslatenz.

interrupt line (dt.) ↑Störleitung.

interrupt lock (dt.) ↑Unterbrechungssperre.

interrupt mask (dt.) ↑Unterbrechungsmaske.

interrupt mode (dt.) ↑Unterbrecherbetrieb.

interrupt priority level (dt.) ↑Unterbrechungsprioritätsebene.

interrupt request (dt.) ↑Unterbrechungsanforderung.

interrupt vector (dt.) ↑Unterbrechungsvektor.

interrupt vector table (dt.) ↑Unterbrechungsvektortabelle.

interval timer (dt.) ↑Intervallzeitgeber.

Intervallzeitgeber (en.) ↑*interval timer*. Bezeichnung für einen ↑Zeitgeber, mit dem der zwischen zwei Zeitpunkten liegende Zeitraum gemessen oder angezeigt werden kann. Jeder der beiden Zeitpunkte definiert jeweils ein ↑Ereignis, das für gewöhnlich mit einem bestimmten Vorgang verknüpft ist. Typisches Beispiel für solch einen Vorgang ist das Ausmessen oder Einstellen der Ausführungsdauer eines (Abschnitts eines) ↑Programms. Beim Ausmessen stehen die beiden Ereignisse dann für zwei ↑Aktionen, nämlich die Zeitmessung zu starten und zu beenden. Als Ergebnis wird die verstrichene Zeit an dem Gerät sodann ablesbar sein. Beim Einstellen wird der Zeitgeber mit einem vorher gemessenen oder berechneten Zeitintervall programmiert. Darüber hinaus kann er im Zuge einer solchen Einstellung instruiert werden, beim Ablauf des Zeitintervalls eine ↑Unterbrechungsanforderung an die ↑CPU zu senden. Hier entspricht das erste Ereignis der vorgenommenen Zeiteinstellung und das zweite Ereignis der verursachten ↑Unterbrechung. Die Kommunikation mit dem Zeitgeber verläuft dabei normalerweise über ↑Geräteregister.

inverted page table (dt.) ↑invertierte Seitentabelle.

invertierte Seitentabelle (en.) ↑*inverted page table*. Datenstruktur einer ↑MMU, durch die ein bestimmter ↑seitennummerierter Adressbereich im ↑Hauptspeicher erfasst wird. Genau genommen ist der Bezugsrahmen jedoch ein ↑realer Adressraum, in dem der Hauptspeicher entsprechend ↑Adressraumbelegungsplan angeordnet vorliegt und eben auch ↑speicherabgebildete Dinge vermöge dieser Datenstruktur adressierbar sind.
Im Gegensatz zur gewöhnlichen ↑Seitentabelle richtet sich die Tabellengröße hier nach der Anzahl der ↑Seitenrahmen, die zur Abbildung von ↑Seiten einem bestimmten ↑Bezugssystem für die ↑Seitenadressierung insgesamt zur Verfügung stehen. Dieses Bezugssystem ist entweder allein durch den Adressraumbelegungsplan definiert oder zusätzlich durch eine jeweilige ↑Prozessinkarnation gegeben. Im ersten Fall gibt es eine zentrale Tabelle fester Länge, die sich alle ↑Prozesse desselben Systems teilen. Im zweiten Fall gibt es pro Prozessinkarnation eine eigene, lokale Tabelle variabler Länge.
Der Seitenadressierung entsprechend bildet jede vom ↑Prozess generierte ↑logische Adresse a ein Paar (p, o) mit ↑Seitennummer p und ↑Versatz o innerhalb von Seite p, wobei das Paar, wie üblich, nach außen als ↑lineare Adresse erscheint. Bei der Adressierung spielt p jedoch nicht mehr die Rolle als Tabellenindex, sondern als Suchschlüssel, und zwar um den Tabelleneintrag zur lokalisieren, der die Seite p abbildet auf einen Seitenrahmen. Aus dem Index des gefundenen Tabelleneintrags (zentrale Tabelle) oder aus dem Eintrag selbst (lokale Tabelle) lässt sich dann die ↑Seitenrahmennummer, f, ableiten, aus der die ↑reale Adresse einer Seite gebildet wird.
Die Einträge der zentralen Tabelle lassen sich nicht eindeutig unterscheiden, wenn nur allein die Seitennummer p als Suchschlüssel genommen wird: ist mehr als ein ↑Prozessadressraum zu verwalten (↑*multiprogramming*), wird derselbe Wert für p mehrfach in der Tabelle vorhanden, die betreffende Seite für gewöhnlich jedoch auf verschiedene Seitenrahmen abgebildet sein. In dem Fall wird der Suchschlüssel dargestellt als Paar $k = (r, p)$, mit r als eindeutige Kennung des p zugehörenden Prozessadressraums als Bezugssystem. Im Fall einer lokalen Tabelle entspricht r der Anfangsadresse der diese Tabelle implementierenden Datenstruktur. Gespeichert wird r typischerweise in einem speziellen ↑Prozessorregister. Erfolgt ein Wechsel des Prozessadressraums, ist der Inhalt des r speichernden Prozessorregisters zu aktualisieren.
….

Suche: linear, Streuwerttabelle

iOS Bezeichnung für das auf ↑XNU basierende ↑Betriebssystem für Mobilgeräte von Apple (d.h., iPhone, iPad und iPod Touch). Erste Installation 2007, läuft auf ↑Prozessoren der ARM-Familie.

IP Abkürzung für (en.) *Internet Protocol*, seit 1974. Ermöglicht die Kommunikation von Nachrichtenpaketen fester Länge in einem ↑Netzwerk. Grundfunktionen sind (1) Adressierung der Netzknoten, (2) Kapselung, Formatierung und Verpackung der ↑Daten, (3) Stückung und Wiederzusammenfügung von Datagrammen und (4) Leitweglenkung (*routing*).

IPC Abkürzung für (en.) ↑*interprocess communication*.

IPI Abkürzung für (en.) *interprocessor interrupt*. Eine ↑Unterbrechungsanforderung, die von der ↑Unterbrechungssteuereinheit eines ↑Rechenkerns zu einem anderen Rechenkern gesendet wird. Bei ↑Prozessoren von Intel wird diese Funktion durch den ↑APIC bereitgestellt.
Für ein ↑Betriebssystem ist eine solche Funktionen ein wichtiges Mittel zur Durchführung kernspezifischer Operationen. Wenn beispielsweise der ↑Leerlauf eines Rechenkerns beendet werden kann, weil der ↑Planer von einem anderen Rechenkern ausgehend einen ↑Prozess zur ↑Einlastung bereitgestellt hat, dann ermöglicht diese durch Software (Betriebssystem) ausgelöste Unterbrechungsanforderung, diesem Prozess seinen Prozessor zu geben. In dem Beispiel wird der Leerlauf des betreffenden Rechenkerns beendet, woraufhin der dortige ↑Leerlaufprozess den bereitgestellten Prozess erkennen und einen ↑Prozesswechsel zu ihm hin durchführen kann.

IPL Abkürzung für (en.) ↑*interrupt priority level.*

IRQ Abkürzung für (en.) ↑*interrupt request.*

irrige Mitbenutzung (en.) ↑*false sharing.* Zugriffsmuster ↑gleichzeitiger Prozesse auf verschiedene Strukturelemente (↑Speicherwort) der kleinsten Verwaltungseinheit einer Speicherressource (↑Zwischenspeicherzeile, ↑Seite), wobei nur letztere der Mitbenutzung (*sharing*) unterliegt. Auch wenn simultane Schreib-lese-Vorgänge verschiedene Strukturelemente betreffen, also logisch nicht zueinander in Konflikt stehen, weil sie etwa unabhängige ↑Variablen betreffen, besteht in solch einem Fall dennoch ein physischer Schreib-lese-Konflikt, nämlich in Bezug auf die umfassende, gemeinsame Verwaltungseinheit, wenn diese Variablen also auf derselben Seite oder in derselben Zwischenspeicherzeile liegen.
Zur Konsistenzwahrung zwischengespeicherter globaler Daten bewirkt jeder Schreibvorgang entweder die Ausspülung (*write-invalidate*) oder die Aktualisierung (*write-update*) der betreffenden Speicherressource (↑*cache*, ↑*page frame*). Dies hat unnötige leistungsmindernde Blocktransfers zur Folge, wenn in logischer Hinsicht keine Zugriffskonflikte auf die einzelnen Strukturelemente bestehen.

ISA Abkürzung für (en.) *instruction set architecture*, oft auch als Synonym für die ↑Befehlssatzebene gebraucht.

isolierte Ein-/Ausgabe (en.) ↑*port-mapped* I/O. Art von Ein-/Ausgabe, bei der die ↑Ein-/Ausgaberegister von einem ↑Peripheriegerät über einen speziellen Anschluss (*port*) zugänglich sind. Der Anschluss wird über eine Nummer identifiziert, die einer ↑Adresse gleicht, jedoch keine ↑Speicherstelle im eigentlichen Sinn adressiert. Zur Durchführung eines Ein-/Ausgabevorgangs ist ein spezieller ↑Maschinenbefehl (`in`/`out` bei x86) erforderlich, der ↑Daten zwischen einem ↑Prozessorregister und dem ausgewählten Anschluss transferiert.

ITS Mehrplatz-/Mehrbenutzerbetriebssystem. Abkürzung für (en.) *Incompatible Time-Sharing System*, als Seitenhieb auf ↑CTSS. Implementiert in ↑Assembliersprache, erste Installation um 1969 (PDP-6, später PDP-10). Leistete ↑Mehrprogrammbetrieb ohne ↑Speicherschutz, machte virtuelle Geräte (zur geräteunabhängigen Ein-/Ausgabe) verfügbar, erlaubte netzwerktransparenten Dateizugriff über ↑ARPANET, ermöglichte die Suspendierung von dem jeweils im Vordergrund stattfindenden ↑Prozess (`^Z`) und bot einen Mechanismus, um einen ↑Systemaufruf sicher unterbrechen zu können (↑PCLSRing). Alle diese Konzepte, bis auf den fehlenden Speicherschutz, sind für ↑UNIX übernommen worden. Das System hatte wesentlichen Einfluss auf die Hackerkultur.

IVT Abkürzung für (en.) ↑*interrupt vector table.*

Java Programmiersprache: imperativ, ↑objektorientiert (1995), eigentlich ↑Interpretersprache.

JCL Abkürzung für (en.) ↑*job control language.*

job (dt.) Tätigkeit, ↑Arbeit.

job control language (dt.) ↑Auftragssteuersprache.

jump label (dt.) ↑Sprungmarke.

jump vector (dt.) ↑Sprungvektor.

JVM Abkürzung für (en.) *Java Virtual Machine*, Teil der Laufzeitumgebung für ein in ↑Java formuliertes ↑Programm. Typischerweise als ↑CSIM realisierte ↑virtuelle Maschine, die ↑Zwischenkode (konkret: ↑Java Bytekode) direkt ausführt. Jedes Programm läuft auf seiner eigenen virtuellen Maschine.

Kachel siehe ↑Seitenrahmen.

Kachelprozessor (en.) ↑*tile processor*. Bezeichnung für einen ↑Mehrkernprozessor, der aus mehreren (norm. identischen, aber auch verschiedenartig strukturierten/aufgebauten) integrierten Baugliedern (↑*tile*) besteht, wobei diese Bauglieder eine oder mehrere Verarbeitungseinheiten (d.h., jeweils ein ↑Rechenkern), einen ↑Zwischenspeicher, gegebenenfalls auch lokalen ↑Hauptspeicher und eine Leitungs- oder Speichervermittlungsstelle (*switch*) umfassen können. Innerhalb des durch ein solches Bauglied abgeschlossenen Gebiets ist für gewöhnlich ↑Speicherkohärenz sichergestellt. Darüberhinaus können manche dieser Bauglieder auch nur globalen Hauptspeicher (↑*memory tile*) oder technische Möglichkeiten zur ↑Ein-/Ausgabe bereitstellen. Nicht alle diese architektonischen Bauglieder müssen identisch ausgelegt sein, die zugrunde liegende ↑Rechnerarchitektur kann ein homogene oder heterogene Struktur definieren.

Kanal (en.) ↑*channel*. Steuereinheit, die die Operationen von einem ↑Peripheriegerät kontrolliert, wobei sich die Betriebsüberwachung (*sphere of control*) auf mehr als ein Gerät derselben Art oder verschiedener Arten erstrecken kann. Wesentliche Funktion dieser Einheit ist es, den bei der Ein-/Ausgabe anfallenden Transfer von ↑Daten zwischen ↑Hauptspeicher und ↑Peripherie unabhängig von der ↑CPU durchzuführen. Dazu nutzt die Einheit ↑Speicherdirektzugriff und sendet zu gegebener Zeit eine ↑Unterbrechungsanforderung an die CPU, um die Beendigung der asynchron geleisteten Ein-/Ausgabearbeit anzuzeigen. Diese als ↑Prozessor fungierende Einheit ist durch ein ↑Kanalprogramm, das die jeweils zu leistende Arbeit genau beschreibt, zwar sehr flexibel einsetzbar, sie dient jedoch vornehmlich dem Sonderzweck (*special purpose*) der Entlastung der CPU von jeder zeitaufwendigen ↑Aufgabe, die mit Ein-/Ausgabe in Verbindung steht. Seit Einführung mit ↑IBM 709, nach wie vor die in einem Großrechner von IBM vorherrschende Ein-/Ausgabetechnik.

Kanalprogramm (en.) ↑*channel program*. Folge von Anweisungen für eine Steuereinheit zur eigenständigen, von der ↑CPU unabhängigen Ein-/Ausgabe von ↑Daten. Die Steuereinheit ist ein dedizierter ↑Prozessor, der einen Ein-/Ausgabekanal (↑*channel*) implementiert. Über diesen ↑Kanal wird der Datentransfer zwischen ↑Hauptspeicher und ↑Peripherie direkt abgewickelt, gesteuert durch ein vom ↑Betriebssystem dazu jeweils dem Prozessor zur Ausführung übergebenes ↑Programm. Eingeführt mit ↑IBM 709, weiterentwickelt für ↑IBM System/360 und nach wie vor zentrale Ein-/Ausgabetechnik der ↑IBM z Systems.

Kartenleser (en.) ↑*card reader*. ↑Peripheriegerät, das auf Knopfdruck automatisch eine ↑Lochkarte nach der anderen einliest, die darauf kodierten ↑Daten (mechanisch oder optisch) abfragt und mittels ↑Ein-/Ausgaberegister dem zugehörigen ↑Gerätetreiber zur Eingabe bereitstellt. Der Gerätetreiber ist für gewöhnlich ein ↑Unterprogramm von einem Steuerprogramm zur Organisation und Überwachung des Rechnerbetriebs (↑*resident monitor*). Falls ↑abgesetzter Betrieb geführt wird, ist das Gerät selbst am ↑Satellitenrechner angeschlossen, ansonsten am ↑Hauptrechner.

Kartenlocher (en.) ↑*key punch*. Gerät zur Erfassung von ↑Daten und deren direkte Speicherung auf einer ↑Lochkarte in kodierter Form. Über eine Zuführvorrichtung werden nach und nach die leeren Lochkarten zur Erfassung automatisch bereitgestellt. Eine Ablagevorrichtung nimmt abschließend die jeweils bearbeitete Lochkarte auf. Die Dateneingabe erfolgt durch eine Tastatur, die der von einem ↑Fernschreiber oder einer Schreibmaschine gleicht. Bei Betätigung der Taste eines alphanumerischen Zeichens (d.h., Buchstabe, Ziffer oder Interpunktionszeichen), erzeugt eine elektromechanisch arbeitende Eingabeeinheit eine Lochung in der aktuellen Spalte einer eingelegten Lochkarte und transportiert die Karte automatisch um eine Spalte weiter nach links. Die Lochung entspricht dem für das jeweilige Zeichen verwendeten Kode (↑EBCDIC). Der Wechsel zu nächsten Karte erfolgt automatisch, wenn eine komplette Zeile (d.h., 80 Spalten) abgelocht wurde, oder durch Betätigung der Taste zum ↑Wagenrücklauf. Bei Geräten mir Kopiervorrichtung, können bestimmte oder alle Spaltenlochungen von einer vorhergehenden Karte mittels Kopiertaste automatisch auf die nachfolgende Karte übertragen werden. Gegebenenfalls wird jedes eingegebene Zeichen an

der Oberkante der Karte, über seiner Lochung in der Spalte, in für den Menschen lesbarer Form zusätzlich dargestellt.

Kartenstanzer (en.) ↑*card punch*. ↑Peripheriegerät, das durch einen zugehörigen ↑Gerätetreiber die zur Ausgabe nacheinander mittels ↑Ein-/Ausgaberegister bereitgestellten ↑Daten automatisch auf eine ↑Lochkarte nach der anderen in entsprechend kodierter Form überträgt. Der Gerätetreiber ist für gewöhnlich ein ↑Unterprogramm von einem Steuerprogramm zur Organisation und Überwachung des Rechnerbetriebs (↑*resident monitor*). Falls ↑abgesetzter Betrieb geführt wird, ist das Gerät selbst am ↑Satellitenrechner angeschlossen, ansonsten am ↑Hauptrechner.

kaskadierte Unterbrechung (en.) ↑*cascaded interrupt*. Bezeichnung für eine stufenförmig angeordnete ↑Unterbrechung, wobei jede Stufe einer bestimmten ↑Unterbrechungsprioritätsebene entspricht. Die Kaskade bildet sich, sobald eine ↑Unterbrechungsbehandlung niedriger Priorität durch eine ↑Unterbrechungsanforderung höherer Priorität unterbrochen wird. In dem Fall wird die Behandlung auf niedrigerer Prioritätsebene aus- und erst wieder fortgesetzt, wenn die durch die Anforderung ausgelöste Behandlung auf höherer Prioritätsebene zum Abschluss gekommen ist. Diese Kaskadierung kann sich wiederholen, in Abhängigkeit von den Eigenschaften der Hardware (↑CPU, ↑PIC): sie bildet einen Stapel von laufenden Unterbrechungsbehandlungen, wobei aber immer nur die oben auf dem Stapel liegende Behandlung voranschreitet und alle im Stapel liegenden Behandlungen sich verzögern.
Typisch ist eine solche Kaskade unterschiedlicher Prioritätsebenen für den Unterbrechungstyp ↑IRQ. Sie bildet sich aber auch im Falle eines ↑NMI, wobei dann jedoch nicht mit jeder Unterbrechungsanforderung dieser Art eine weitere Prioritätsebene entsteht, wohl aber eine weitere Inkarnation eines ↑Unterbrechungshandhabers. Ein NMI ist nicht nur nicht maskierbar, sondern aktiviert auch immer die höchste Prioritätsebene: jede Inkarnation des entsprechenden Handhabers wird mit derselben Priorität zur Ausführung kommen. So erweitert der NMI die durch den IRQ möglichen Prioritätsebenen effektiv nur um eine weitere Ebene, jedoch bleibt dadurch die Anzahl der Handhaberinkarnationen unbestimmt:

$$N_{level} = \sum_{l=0}^{n} IRQ_l + NMI \qquad \text{versus} \qquad N_{incarnation} = \sum_{l=0}^{n} IRQ_l + \sum_{l=0}^{\infty} NMI_l$$

Diese Unbestimmtheit der Anzahl möglicher Handhaberinkarnationen, wenn der NMI mit in die Betrachtung einbezogen wird, stellt ein grundsätzliches Problem der ↑Systemprogrammierung für die „nackte Hardware“ dar. Sollte etwa die Hardware einen ↑Defekt in Bezug auf den NMI zeigen, der eine hochfrequente „unechte Unterbrechung“ (↑*spurious interrupt*) als ↑Fehler hervorbringt, dann besteht die große Gefahr für das totale ↑Versagen des ↑Rechensystems. Um ↑Gefahren dieser Art unwahrscheinlich zu machen, wird der NMI nicht für den normalen ↑Unterbrecherbetrieb genutzt, sondern nur, um ein schwerwiegendes Problem als ↑Ereignis an das ↑Betriebssystem zu signalisieren. Darüberhinaus wird für diese Signalisierung für gewöhnlich die ↑Flankensteuerung genutzt, um nämlich die Wahrscheinlichkeit zu senken, dass der ↑Laufzeitstapel im Betriebssystem überläuft und dadurch ↑Panik entsteht — was bei ↑Pegelsteuerung leicht der Fall wäre, da der NMI-Handhaber niemals die Unterbrechungsanforderung quittieren sowie abstellen könnte, da er nämlich sofort nach seinem Start wieder unterbrochen wird und im weiteren Verlauf in eine (logisch) unendliche indirekte ↑Rekursion verfällt. Gleiches könnte übrigens geschehen, sollte ein IRQ-Handhaber während seines Verlaufs von sich aus auf eine niedrigere Prioritätsebene wechseln und er dadurch Gefahr läuft, mit einer der nächsten Unterbrechungsanforderungen sich selbst in der Ausführung rekursiv zu überlappen.
Aus dieser Gründen findet eine Unterbrechungsbehandlung für gewöhnlich in zwei Phasen statt. Zum einen ist jede der Unterbrechungsbehandlungen durch einen Unterbrechungshandhaber „erster Stufe“ (↑FLIH) repräsentiert, der mit der durch die Hardware definierten Unterbrechungspriorität ausgeführt wird. Zum anderen kommen Unterbrechungshandhaber „zweiter Stufe“ (↑SLIH) zur Ausführung, um ↑Aufgaben von geringerer und durch Software

definierter Priorität zu bewerkstelligen. Letztere starten aber erst dann, wenn eine möglicherweise vorhandene Kaskade ersterer komplett wieder abgebaut wurde. Die Kaskadierung ist und bleibt damit in erster Linie ein Merkmal der Hardware.

Kausalität (en.) ↑*causality*. Ursächlichkeit, kausaler Zusammenhang (Duden). Bezeichnet allgemein die Beziehung zwischen *Ursache* und *Wirkung*. In Bezug auf ↑Prozesse, die insbesondere gleichzeitig stattfinden können, ist darin ein bestimmtes wechselseitiges Verhältnis zu verstehen, nämlich welche ↑Aktion auf welches ↑Betriebsmittel in welcher Art zugreift und welche andere Aktion dieses Betriebsmittel bereitstellt. Gemeinhin ist gerade für ein ↑nichtsequentielles Programm — vor allem in Form eines ↑Betriebssystems — das Festsetzen eines solchen Verhältnisses im Voraus schwierig oder gar unmöglich, wenn die diese Aktionen auslösenden ↑Ereignisse zu unvorhersagbaren Zeiten auftreten. In solchen Fällen sind die betreffenden Prozesse explizit einer ↑Nebenläufigkeitssteuerung zu unterwerfen, um die Aktionen konsistent stattfinden zu lassen.

Kausalordnung (en.) ↑*causal order*. Bezeichnung für eine die ↑Kausalität bestimmende Reihenfolge innerhalb einer Menge von ↑Ereignissen. Ein Ereignis B, das erst stattfinden kann (Wirkung), nachdem ein anderes Ereignis A bereits stattgefunden hat (Ursache), ist kausal abhängig von diesem anderen Ereignis ($A < B$ bzw. $B > A$): Ereignis B wird durch Ereignis A beeinflusst, kann nicht unabhängig von A geschehen.

Kennungswechsel Veränderung der ↑Benutzerkennung oder ↑Gruppenkennung von einem ↑Prozess. Der Wechsel geschieht explizit durch einen ↑Systemaufruf (z.B. `setuid(2)`) oder implizit bei Gebrauch von einem ↑Objekt, das dazu mit einem speziellen Attribut versehen sein muss. Beispiel für letzteres ist das *setuid*-Recht einer ausführbaren ↑Datei, etwa eine Datei, die ein ↑Maschinenprogramm enthält. Wird dieses Maschinenprogramm zur Ausführung gebracht, findet der zugehörige Prozess, zusätzlich zu seinen ursprünglichen Rechten, mit den Rechten derjenigen ↑Entität statt, der die Datei gehört.
Typischerweise wird — in einem ↑UNIX-artigen ↑Betriebssystem — zwischen echter (*real*), effektiver (*effective*) und geretteter (*saved*) Kennung eines Prozesses unterschieden. Die echte Kennung wird bei der ↑Anmeldung zugewiesen und definiert die ursprüngliche (*primary*) Kennung eines Prozesses. Die effektive Kennung nutzt ein Prozess, um zeitweise mit höheren Privilegien oder Rechten stattfinden zu können. Mit ihr erfolgt in den überwiegenden Fällen die Rechteprüfung im Betriebssystem. Die gerettete Kennung erlaubt einem zeitweise mit höheren Privilegien/Rechten stattfindenden Prozess, temporär mit niedrigeren Privilegien/Rechten voranzuschreiten, ohne seinen erhöhten Rechtestatus zu verlieren. Initial haben alle drei Kennungen denselben Wert, nämlich den der echten Kennung.

kernel siehe (en.) ↑*operating-system kernel*.

kernel mode siehe ↑*privileged mode*.

kernel process (dt.) ↑Prozess des ↑Betriebssystemkerns.

kernel space siehe ↑*system space*.

kernel thread (dt.) ↑Systemkernfaden.

kernel-level thread (dt.) ↑Faden des ↑Betriebssystemkerns. Ein ↑Systemkernfaden, der einen ↑Kontrollfluss der ↑Betriebssystemebene repräsentiert, ausschließlich auf dieser verwaltet wird, ebenda verankert ist, immer im privilegierten ↑Arbeitsmodus (↑*kernel mode*) abläuft, aber einen Kontrollfluss höherer Ebene (↑*user-level thread*) tragen kann.

Kernspeicher (en.) ↑*core memory*. Eine Vorrichtung zur nichtflüchtigen Aufbewahrung von Informationen (↑*non-volatile memory*) auf Grundlage von Ringkernen in Form von Dauermagneten aus Ferrit, auch als *Magnetkernspeicher* bezeichnet. Dabei repräsentiert ein einzelner Ringkern den Informationsträger für genau ein Bit.

Die Ringkerne sind mittels Selktions-, Abtast- und Blockierdrähten zu einer Gitterstruktur in Form einer Matrix aufgefädelt, wobei die zeitverschobenen Funktionen zum Abtasten und Blockieren auch ein und denselben Draht nutzen können. Die beiden in X- und Y-Richtung verlaufenden Drähte legen das Gitter fest und bilden die eigentliche *Selektionsleitung*.
Lesen und Schreiben einer Information (0 bzw. 1) geschieht immer in ein und demselben Zyklus. Um einen Ringkern anzusprechen, werden die X- und Y-Drähte der Selektionsleitung jeweils hälftig mit Strom versorgt, so dass dem Ringkern am Kreuzungspunkt die ganze elektrische Leistung trifft. Bei der Selektion, die auch dem *Lesezyklus* entspricht, wird so der Informationsgehalt eines Ringkerns allerdings immer auf den logischen Wert 0 gesetzt (*destructive read*). Folglich ist der ursprüngliche Informationsgehalt durch einen nachfolgenden Schritt wieder herzustellen, aber nur, falls der Ringkern den Wert logisch 1 enthielt. Ein Wechsel der Polarität des Ringkerns von logisch 1 nach 0 erzeugt einen wahrnehmbaren Impuls, der über den Abtastdraht geht und somit auf den letzten Informationsgehalt schließen lässt: logisch 1, wenn ein Impuls festgestellt wird, logisch 0 sonst. Um im *Schreibzyklus* logisch 1 zu hinterlassen, wird die Polarität des selektierten Ringkerns geändert. Dies geschieht durch einen Strom in umgekehrter Richtung zum Lesezyklus. Demgegenüber wird ein Informationsgehalt von logisch 0 im Ringkern erzeugt, indem dieselbe Menge des zur Selektion gebrauchten Strom zusätzlich über den Blockierdraht geht. Dadurch halbiert sich der effektiv durch den Ringkern fließende Strom, was zur Sperrung der Polaritätsänderung führt und den Wert logisch 0 dort belässt.
Diese um 1949 entwickelte Technik ist heute (2025) nur noch vereinzelt anzufinden, beispielsweise in Altanlagen der Bahntechnik. So lässt sich in der Signaltechnik der Zustand von Fahrsignalen auch ohne Betriebsstrom zuverlässig speichern. Überhaupt verhält sich diese Technik vergleichsweise robust gegenüber elektromagnetischen Impulsen und Strahlung. Für gewöhnlich taucht diese Speicherform begrifflich jedoch nur noch im Zusammenhang mit einem ↑Kernspeicherabzug auf.

Kernspeicherabzug (en.) ↑*core dump*. Historisch die Bezeichnung für das Leeren (*dump*) des ↑Kernspeichers als Folge der Feststellung das ↑Versagens von einem ↑Programmablauf. Zum Zwecke der anschließenden Fehlerbeseitigung (↑*debug*) wurde der Kernspeicherinhalt für gewöhnlich mittels ↑Zeilendrucker auf ↑Tabellierpapier ausgegeben. Mit der sonst nicht unüblichen Bezeichnung von Kernspeicher gemeinhin auch als Zentral- oder ↑Hauptspeicher geschieht heute diese Ausgabe in eine ↑Datei — sofern der Programmablauf nicht wegen ↑Panik im ↑Dateisystem (innerhalb des ↑Betriebssystems) versagt hat. So ist die von ↑UNIX-Systemen bekannte Fehlermeldung

↑*segmentation fault* (*core dumped*)

eben im letzteren Sinn zu verstehen, sie deutet keineswegs eine bestimmte technische Auslegung des Hauptspeichers an. Falls freigegeben, wird in solch einem Fall im ↑Arbeitsverzeichnis eines gescheiterten ↑Prozesses die Datei `core` erzeugt (`core(5)`).
Der Speicherabzug liegt in numerischer Form vor, er entspricht der auf der ↑Maschinenprogrammebene gebräuchlichen Darstellungsweise. Gespeichert wird üblicherweise der ↑Prozessorstatus und der Inhalt des ↑Prozessadressraums in Abhängigkeit von dem ↑Arbeitsmodus, der im Moment des Versagens galt. War das Betriebssystem aktiv (↑*system mode*), geht die Ausgabe über die im Falle des gescheiterten Ablaufs eines ↑Maschinenprogramms (↑*user mode*) sonst generierten Informationen hinaus und beschreibt noch zusätzlich den nur für privilegierte Vorgänge (↑*privileged mode*) zugänglichen Hardwarezustand.

Kettenblockierung (en.) ↑*chain blocking*. Andauerndes verhängen und aufheben einer ↑Prozesssperre für denselben ↑Prozess. Damit einhergehend ↑Umplanung und ↑Prozesswechsel, nämlich wenn der Prozess (1) bei Anforderung eines gesperrten ↑Betriebsmittels blockieren und (2) bei Freigabe dieses Betriebsmittels deblockiert sowie der freigebende Prozess dadurch vom Prozessor verdrängt wird.
Diese Abfolge hat ihre Ursache in ↑Betriebsmittelsperren, die verschiedene Prozesse niedriger

aber unterschiedlicher ↑Priorität jeweils für ein anderes ↑unteilbares Betriebsmittel verhängt haben (↑*transitive blocking*). Ein Prozess, dessen Priorität höher ist all diese Prozesse, der aber die durch diese Prozesse gesperrten Betriebsmittel nacheinander anfordert, verfällt bei verdrängender ↑Vorrangplanung zwangsweise in ein Hin und Her in Bezug auf die Nutzung des ↑Prozessors für das eigene dringliche Vorankommen.
Das Verhängen und Aufheben der Sperren, die Umplanung und die Prozesswechsel bedingen beträchtliche ↑Gemeinkosten, die zur Erhöhung der ↑Sperrzeiten des hochprioren Prozesses beitragen. Ein solcher Effekt, der bei Mitbenutzung (*sharing*) unteilbarer Betriebsmittel durch mehrere Prozesse unterschiedlicher Priorität entstehen kann, ist typisch für die ↑Prioritätsvererbung und sorgt dafür, dass die totale Sperr-/Blockierungszeit des jeweils höchst prioren Prozesses nicht minimal ist und auch nur schwer (wenn überhaupt) im Voraus berechnet oder gar nur abgeschätzt werden kann.

key punch (dt.) ↑Kartenlocher.

Klasse (en.) ↑*class*. Bezeichnung für eine Schablone, Vorlage, aus der ↑Objekte durch spezielle Operationen (`create`, `new`) erzeugt werden können. Ein ↑Objekttyp, das heißt, eine für gewöhnlich auch als ↑Datentyp verstandene „Blaupause“ (*blueprint*), um mehreren Objekten dieselben Operationen, auch als Methoden bezeichnet, und gleichartiges Verhalten zu geben. Derartige Blaupausen haben eine oder mehrere Schnittstellen, die die auf die Objekte anwendbaren Operationen beziehungsweise ↑Signaturen der entsprechenden Methoden aufführen. Der jeweilige Klassenrumpf spezifiziert die Implementierung dieser Operationen in Form von ↑Unterprogrammen, die demselben deklarativen Bereich (↑*name space*) angehören.

Klassenhierarchie (en.) ↑*class hierarchie*. Bezeichnung für die Klassifikation von ↑Datentypen, die ↑Objekte als ↑Exemplare von ↑Klassen sprachlich bezeichnen, wobei die Klassen in einer bestimmten Wechselbeziehung zueinander stehen. Diese Beziehung ist definiert durch eine ↑hierarchische Struktur von Klassen, sie entsteht durch ↑Vererbung oder Erweiterung bestehender Klassen, drückt die Abstraktion von etwas aus oder repräsentiert eine Schnittstellendefinition.

Kommando (en.) ↑*command*. Ein durch einen (kurzen) Befehl erteilter ↑Auftrag, der einen ↑Prozess zur Übernahme einer bestimmten ↑Aufgabe verpflichtet. Die einzelne Anweisung in einer ↑Kommandosprache.

Kommandointerpreter (en.) ↑*command interpreter*. Bezeichnung für ein ↑Programm, bei dessen Ablauf die in ↑Kommandosprache formulierten Anweisungen an ein ↑Rechensystem interpretiert werden. Ein ↑Interpreter von Kommandos, die ihm als Befehlsstrom in einem ↑Stapel (↑*batch processing*) oder interaktiv über eine ↑Dialogstation (↑*time sharing*) zugestellt werden: letzteres, um eine Sitzung mit dem ↑Betriebssystem zu bestreiten (↑*shell*). Ursprünglich (um 1955) jedoch Inbegriff für das zentrale Steuerprogramm zur ↑Stapelverarbeitung und wesentlicher Bestandteil von einem „embryonalen Betriebssystem“ (↑*resident monitor*, ↑FMS).

Kommandosprache (en.) ↑*job control language*. Sprache, in der ein ↑Auftrag an ein in ↑Stapelbetrieb laufendes ↑Rechensystem formuliert wird. Ursprünglich eine reine ↑Steuersprache, um den Rechnerbetrieb automatisiert durch einen ↑Kommandointerpreter zu erreichen (↑*automated computer operation*). Als eine Art ↑Skriptsprache heutzutage (2025) jedoch auch ein Ausdrucksmittel, um wiederkehrende oder interaktionslose Arbeiten am ↑Rechner zu verrichten.
Nachfolgend zwei Beispiele, links ein (fiktiver) Auftrag für ↑FMS zur Ausführung von einem ↑Programm geschrieben in ↑FORTRAN und rechts ein Auftrag für eine(n) ↑*shell* (`bash(1)`) zur Ausführung eines semantisch äquivalenten Programms in ↑C:

```
*JOB, 42, ARTHUR DENT
*XEQ
*FORTRAN
 PRINT *, "Hello world!"
 END
*END
```

```
echo '
int main() {
    printf("Hello world!\n");
}
' | gcc -x c -
./a.out
```

Beiden Beispielen gemeinsam ist die vollständige Beschreibung (in einer domänenspezifischen ↑JCL) des von einen Kommandointerpreter abzuwickelnden Auftrags, um diesen dann ohne weiteren Eingriff ausführen zu können (↑*batch processing*). Im Beispiel links ist ein für den Kommandointerpreter bestimmtes ↑Kommando durch das ↑Fluchtsymbol `*` markiert. Bei Verwendung einer ↑Lochkarte sind solche Fluchtsymbole gelegentlich einer speziellen (z.B. der ersten) Spalte vorbehalten, um dadurch die Auswertung aller nachfolgenden Zeichen dieser Zeile dem Kommandointerpreter zu übertragen. Im Beispiel rechts dagegen erfolgt durch das Fluchtsymbol `'` eine Auswertungsunterdrückung (*quoting*), um die bis zum folgenden Zeichen `'` stehenden Anweisungen dem Kommandointerpreter vorzuenthalten. Der für FMS bestimmte Auftrag beschreibt eine durch `JOB` im ↑Betriebssystem mit der Kennung `42` zur Abrechnung für Benutzer `ARTHUR DENT` identifizierte ↑Arbeit zur Ausführung (`XEQ`) des Übersetzers (`FORTRAN`), der die nachfolgenden Zeilen in ein anschließend sofort auszuführendes ↑Maschinenprogramm transformieren soll. Dabei markiert das erste `END` einerseits das Ende der Übersetzungseinheit und andererseits den Start der Programmausführung, wohingegen das zweite `END` das Auftragsende für den Kommandointerpreter anzeigt. Der *shell*-Auftrag leitet mittels `|` (↑*pipe*-Kommando) die in C formulierten, durch `'` zunächst unterdrückten und dann aber wiedergegebenen (`echo`), Anweisungen in den Übersetzer (`gcc`), um das ↑Lademodul (`a.out`) erzeugt zu bekommen und daraufhin das generierte Maschinenprogramm im ↑Arbeitsverzeichnis zu starten (`./a.out`).

Kommandozeile (en.) ↑*command line*. Reihe von nebeneinanderstehenden Wörtern, wobei das erste Wort für gewöhnlich ein ↑Kommando bezeichnet und die nachfolgenden Wörter Optionen oder Parameter dazu benennen. Die korrekte Verknüpfung der sprachlichen Einheiten in einem Satz (d.h., die Syntax) gibt die verwendete ↑Kommandosprache vor. Gegebenenfalls erlaubt die Sprache mehrere solcher Kommandos pro Zeile, jeweils getrennt durch ein spezielles Satzzeichen. Mit abgeschlossener Eingabe der Reihe (↑Wagenrücklauf) beginnt die Ausführung der Kommandos durch ein spezielles Steuerprogramm (↑CLI).

Kompilation (en.) ↑*compilation*, ↑*compiling*. Vorgang der ↑Übersetzung von einem ↑Programm durch einen ↑Kompilierer. Das in einer höheren Programmiersprache vorliegende Programm wird in eine semantisch äquivalente Repräsentation in eine andere Programmier- oder in ↑Assembliersprache umgewandelt. Die symbolische Darstellung des Programms wird beibehalten: sowohl das ursprüngliche als auch das generierte Programm sind beides ↑Quellmodule, für gewöhnlich wird nur das *Sprachniveau* (von hoch nach niedrig) gewechselt.
Bei der Transformation des Programms von der *Quellsprache* zur *Zielsprache* kann der Niveauwechsel innerhalb der höheren Programmiersprachenebene geschehen oder mit einem Ebenenwechsel einhergehen (↑*multi-level machine*). Beispiel für die erste Variante wäre die Übersetzung von ↑C++ nach ↑C, wie im Falle des `cfront` (1983–1993) oder Comeau C/C++, wohingegen die gebräuchliche zweite Variante die Quellsprachen C oder C++ etwa in die Assembliersprache ↑ASM86 übersetzt und damit von der problemorientierten Programmiersprachenebene eine Stufe tiefer zur Assembliersprachenebene wechselt.

Exkurs In dem hier betrachteten Zusammenhang von ↑Mehrebenenmaschinen ist der Kompilierer ein Werkzeug zur ↑Entvirtualisierung, indem er Programme, die für eine bestimmte ↑virtuelle Maschine M_i gedacht sind, in Programme einer anderen, in der Hierarchie tiefer liegenden (virtuellen) Maschine M_j, $j < i$, übersetzt und dadurch die mit Ebene i eigentlich erreichte ↑Virtualisierung auflöst. Dieser Übersetzungsvorgang, in mehreren Schritten wiederholt angewendet, führt letztlich dazu, die betreffenden Programme für die vorgesehene *reale Maschine* so aufbereitet zu haben, dass sie direkt von dieser ausführ-

bar sind. Nachfolgend ist das (zur besseren Lesbarkeit von Hand aufbereitete) Ergebnis eines solchen einstufigen Vorgangs (C ↦ ASM) beispielhaft dargestellt, das einen kompletten Ebenenwechsel veranschaulicht und den Unterschied im Sprachniveau begreiflich machen soll:

```c
#include <stdio.h>

int main() {
    printf("Hello world!\n");
}
```

```
        .text
        .p2align 4,,15
        .globl main
        .type main, @function
main:
        leal 4(%esp), %ecx
        andl $-16, %esp
        pushl -4(%ecx)
        pushl %ebp
        movl %esp, %ebp
        pushl %ecx
        subl $16, %esp
        pushl $.LC0
        call puts
        movl -4(%ebp), %ecx
        addl $16, %esp
        leave
        leal -4(%ecx), %esp
        ret
.LC0:
        .string "Hello world!"
```

In dem Beispiel wurde das oben stehende, in der Quellsprache C formulierte Programm mit der Anweisung `gcc -O -m32 -S` in das rechts abgebildete Programm kompiliert, und zwar in einer für ↑GAS gültigen Zielsprache für ↑x86. Der Unterschied im Sprachniveau ist deutlich, außer dem ↑Namen `main` und der Zeichenkette „`Hello world!`“ zeigen sich keine weiteren Merkmale, die eine Wiederkennung ermöglichen. Da in der auszugebenden Zeichenkette keine (durch das `%`-Zeichen spezifizierten) Formatangaben enthalten sind, `printf` also ausschließlich Klartext (*plain characters*) zu verarbeiten hätte, hat der Kompilierer den Aufruf an die Spezialfunktion `puts` generiert, ebenfalls ein in der ↑libc enthaltenes ↑Unterprogramm. Dieses schließt die Ausgabe implizit mit einem Zeilenumbruch ab, weshalb der Kompilierer die Zeichenkombination „\n“ aus der Zeichenkette entfernt hat. Der am Kopf stehende (generierte) ↑Pseudobefehl `.text` sorgt dafür, dass sowohl das `main`-Programm als auch die mit der Marke (↑*label*) `.LC0` ausgezeichnete Zeichenkette ins ↑Textsegment platziert wird. Diese, wie auch die anderen mit dem `.`-Zeichen angeführten Pseudobefehle, sind jedoch Anweisungen an den ↑Assemblierer, nämlich den von im zu generierenden ↑Maschinenkode entsprechend für den ↑Binder auszuweisen und zusammenzustellen.

Kompilierer (en.) ↑*compiler*. Softwareprozessor, der ein ↑Programm, das in einer Quellsprache formuliert vorliegt, in ein semantisch äquivalentes Programm einer Zielsprache übersetzt.

Komplexbefehl (en.) ↑*complex command*. Art der technischen Auslegung eines ↑Systemaufrufbefehls, der zum Abruf seines ↑Operationskodes und gegebenenfalls zugehöriger Operanden einen komplexen ↑Befehlszyklus im ↑Systemaufrufzuteiler bedingt (vgl. S. 342). Aufbau und Struktur eines solchen Befehls, dessen Ausführung einen ↑Systemaufruf bewirkt, ähneln einem für ↑CISC typischen ↑Maschinenbefehl.
Gegensatz zum ↑Primitivbefehl trotz dem gemeinsamen Merkmal einer bestimmten ↑Aktionsfolge zur Operandenauswertung, der bei ↑Mehrfädigkeit im ↑Maschinenprogramm eine konsistente Sicht auf die Operanden zuzusichern ist. Da die Auswertung jedoch auf der ↑Systemebene stattfindet und, anders als beim Primitivbefehl, dieser der Kontext der entsprechenden Aktionsfolge bekannt ist, können implizit Maßnahmen zur Konsistenzsicherung durch das ↑Betriebssystem greifen. Der die Anweisungen zur Operandenauswertung umfassende Systemaufrufbefehl bildet eine ↑Elementaroperation, die entweder direkt durch ↑Synchronisierung oder indirekt durch ↑kooperative Planung im Betriebssystem abgesichert und atomar gestaltet werden kann. Vor dem Hintergrund gilt nur der Abruf des Systemaufrufbefehls aus dem Maschinenprogramm gegebenenfalls als ↑atomare Operation, nicht aber die Ausführung der diesem Befehl entsprechenden ↑Systemfunktion.
Der ↑Prozess im Maschinenprogramm (genauer: im ↑Systemaufrufstumpf) führt für gewöhnlich nur den ↑Maschinenbefehl zum Wechsel von der ↑Benutzerebene hin zur Systemebene

aus und unterbricht sich dadurch von selbst, um den Systemaufruf auszulösen. Dazu nutzt er einen ↑Unterbrechungsbefehl (`sos`). Die Parameter für den Systemaufruf folgen diesem Maschinenbefehl im Befehlsstrom (↑Textsegment) für das Betriebssystem direkt als Operanden. Nachfolgend ein Beispiel für ein fiktives Betriebssystem (↑x86):

```
sos                    # switch to operating system: trap assumed, here
.long opc              # operation code for the operating system
.long op1              # 1st operand
.long op2              # 2nd operand
…                      …
.long opn              # nth operand
↑Systemaufruffehler?   # aftercare of the system call
```

Der erste Maschinenbefehl nach der Operandenliste, deren Länge vom jeweiligen, durch den ↑Operationskode *opc* bestimmten Systemaufruf abhängt, prüft für gewöhnlich ab, ob ein Systemaufruffehler aufgetreten ist. Gegebenenfalls wird dann die Fehlerbehandlung im Kontext des Maschinenprogramms eingeleitet (vgl. S. 338).
Diese Kodierungsform impliziert einen Systemaufrufzuteiler, der zur Ausführung des Systemaufrufs die Operanden aus den Befehlsstrom lesen und dabei den vom Betriebssystem verwalteten ↑Befehlszähler des Maschinenprogramms, PC_{os}, Schritt für Schritt weiterschalten muss, so dass bei Rückkehr zur Benutzerebene der Maschinenbefehl zur Fehlerabfrage als nächster ausgeführt wird. Damit ähnelt ein solcher Systemaufrufbefehl einem gewöhnlichen Maschinenbefehl, nur dass er durch das Betriebssystem und nicht von der ↑CPU interpretiert wird. Die für seine ↑partielle Interpretation erforderlichen Schritte folgen weitestgehend dem ↑Befehlszyklus einer CPU, die durch den von ihr verwalteten Befehlszähler, PC_{cpu}, auch zunächst den Operationskode lesen und deuten muss, um so einerseits die Operation und anderseits die Adressierungsart des oder der Operanden zu identifzieren und nach Befehlsausführung im ↑Statusregister die Zustandsanzeige (bspw. *carry bit*, C) in Bezug auf die durchgeführte Berechnung zu hinterlassen. Im Fehlerfall ($C = 1$) wird ein bestimmtes Prozessorregister (`eax`) den Fehlerkode enthalten, anderenfalls ($C = 0$) steht in diesem ↑Register der Rückgabewert der ausgeführten Systemfunktion.
Das Weiterschalten von PC_{os} geschieht, nachdem der Systemaufrufzuteiler den Systemaufrufbefehl vollständig abgerufen hat. Hierbei ist zu beachten, dass die CPU ihren ↑Prozessorstatus, damit den Wert von PC_{cpu}, im ↑Unterbrechungszyklus in dem Moment auf den ↑Laufzeitstapel des Betriebssystems sichert, wenn die Ausführung des Systemaufrufbefehls startet: dadurch erhält PC_{os} seinen initialen Wert, nämlich den von PC_{cpu}. Dieser Zyklus wurde durch einen Prozess im Maschinenprogramm selbst ausgelöst (↑*software interrupt*), indem die CPU den einen Systemaufruf einleitenden ↑Unterbrechungsbefehl (`sos`: hier expandiert als `int`-Befehl) ausführt. Der Systemaufrufzuteiler liest PC_{os} (d.h., der auf dem Laufzeitstapel gesicherte Wert von PC_{cpu}) aus und, da der gelesene Wert eine ↑Adresse im Textsegment des Maschinenprogramms repräsentiert, initialisiert damit einen ↑Zeiger, um darüber dann den Operationskode nebst Operanden aus dem Maschinenprogramm zu lesen. Analog zur CPU, die PC_{cpu} nach jeden dieser Abrufschritte implizit weiterschaltet, funktioniert der Systemaufrufzuteiler: schrittweise setzt er den Zeiger automatisch weiter, so dass am Ende, wenn der komplette Systemaufrufbefehl gelesen wurde, der diesem Befehl nachfolgende Maschinenbefehl adressiert wird.
Der nach Abruf des Systemaufrufbefehls im Systemaufrufzuteiler gültige Zeigerwert ersetzt sodann den im Laufzeitstapel gesicherten Inhalt von PC_{cpu}: das heißt, der Systemaufrufzuteiler überschreibt PC_{os} einfach. Bei Rückkehr vom Systemaufruf beziehungsweise aus der damit verknüpften ↑Unterbrechungsbehandlung (`iret`) stellt die CPU ihren zuvor gesicherten Prozessorstatus wieder her, womit sie PC_{cpu} den Wert von PC_{os} zuweist. Die Folge davon ist, dass die CPU die Ausführung des Maschinenprogramms mit dem Maschinenbefehl wiederaufnimmt, der den Operanden des vom Betriebssystem verarbeiteten Systemaufrufbefehls folgt.
Die Argumente für den Systemaufruf gehören zu zwei Sorten von Parametern: *Wertparameter*

und *Referenzparameter*. Von der Wertparametersorte sind unmittelbar adressierte Operanden, die dem Unterbrechungsbefehl im Befehlsstrom folgen und deren Auswertung vor ↑Laufzeit des Maschinenprogramms geschehen ist. Demgegenüber umfasst die Referenzparametersorte direkt (absolut) adressierte Operanden, deren Adressen dem Unterbrechungsbefehl im Befehlsstrom folgen und die zur Laufzeit ausgewertet werden. Für jeden unmittelbar adressierten Operanden ist lediglich sein Wert in den Befehlsstrom einzufügen, für jeden direkt adressierten Operanden aber seine Adresse. Diese Direkt- und Adresswerte sind Konstanten, zumal sie im für gewöhnlich schreibgeschützten Textsegment des Prozesses liegen. So ist der Operandenzugriff für den Systemaufrufzuteiler einerseits einfach, wenn er durch unmittelbare Adressierung geschieht und andererseits aufwendiger, wenn er auf direkte Adressierung zugreifen muss. Darüberhinaus muss der Zuteiler Kenntnis darüber besitzen, welcher Operand wie zu adressieren ist. Angenommen, das den Operationskode tragende Argument (`opc`) ist 32 Bits breit, wovon der Operationskode die niederwertigen 8 Bits beansprucht. Dann bleiben 24 Bits zur Klassifizierung von bis zu 24 Operanden beispielsweise mit folgender Bedeutung: 0 für unmittelbare und 1 für direkte Adressierung.
Eine andere Variante besteht darin, grundsätzlich zwischen sogenannten direkten und indirekten Systemaufrufen zu unterscheiden, wie etwa im Falle von ↑UNIX V6 (vgl. S. 341). Direkte Systemaufrufe verwenden die unmittelbare Adressierung, die Operanden sind Teil des Systemaufrufbefehls und liegen im Textsegment. Indirekte Systemaufrufe geschehen durch zwei verknüpfte Systemaufrufbefehle. Der erste Befehl (Textsegment) verwendet die direkte Adressierung, um einen ↑Deskriptor für den zweiten Befehl (Datensegment) zu referenzieren. Dieser *Hauptbefehl* hat den Operationskode mit Nummer 0 und veranlasst den Systemaufrufzuteiler dazu, den direkt adressierten und im Deskriptor kodierten *Nebenbefehl*, der die eigentlich zu startende Systemfunktion identifiziert, zur Ausführung zu bringen. Die Auswertung der somit durch einen Referenzparameter (nämlich der einzige Operand des Hauptbefehls) adressierten Operanden des Nebenbefehls kann zur Laufzeit geschehen.
Im Gegensatz zum Primitivbefehl, bei dem die Operandenauswertung im Systemaufrufstumpf geschieht, erfolgt diese Auswertung für die hier betrachtete Variante im Systemaufrufzuteiler. Wären Stumpf wie auch Zuteiler demselben ↑Adressraum zugeordnet, ergäbe sich zwischen beiden Varianten kaum ein Unterschied. Jedoch liegt zwischen diesen beiden Komponenten nicht nur eine logische, sondern auch eine physische Grenze, die durch die Art der im System etablierten ↑Adressraumisolation definiert ist. Folgt die Abkapselung eines Prozesses einem ↑Mehradressraummodell, bestimmt letztlich der Grad der den Prozessen damit gegebenen Privatspähre, ob die Operandenauswertung auch für den Systemaufrufzuteiler vergleichsweise einfach ist. Dies ist etwa der Fall, wenn die Adressräume für die auf Benutzerebene stattfindenden Prozesse *teilprivat* und damit für ↑Aktionen der Systemebene, eben die Operandenauswertung im Systemaufrufzuteiler, direkt zugänglich sind (↑Linux, ↑Windows NT). Im Falle von *strikt privat* eingerichteten Adressräumen (UNIX V6, ↑macOS) muss zum Abruf des Systemaufrufbefehls und seiner Operanden durch den Systemaufrufzuteiler der Adressraum des jeweiligen Benutzerebenenprozesses erst zugänglich gemacht werden. Die dafür notwendigen Schritte sind vergleichsweise zeitaufwendig und erhöhen die ↑Latenz für einen Systemaufruf durchaus erheblich.

komplexe Koroutine Bezeichnung für eine ↑Koroutine, die getrennt von anderen ihrer Art einen eigenen ↑Laufzeitstapel besitzt. Die ↑Handhabe für einen ↑Handlungsstrang innerhalb einer solchen Koroutine ist ein ↑Stapelzeigerwert. Dieser Wert ist (logisch nicht auflösbar, zusammenfassend) der Zeiger auf einen Zeiger auf den ↑Maschinenbefehl, der nach erfolgtem ↑Koroutinenwechsel als nächster ausgeführt wird. Anders als eine ↑primitive Koroutine.

Konkurrenz (en.) ↑*competition*. Möglicher Begleitumstand ↑gleichzeitiger Prozesse, wenn diese in ↑Wettstreit um ein ↑unteilbares Betriebsmittel geraten.

Konkurrenzsituation (en.) ↑*contention*. Kritischer Umstand ↑gleichzeitiger Prozesse, die zur gleichen Zeit Anspruch auf ein ↑gemeinsames Betriebsmittel erheben, das auch als ↑unteilbares Betriebsmittel eingestuft ist. Im Moment der Beanspruchung übernimmt jeder dieser

↑Prozesse die Rolle als ↑gekoppelter Prozess, der durch dieses ↑Betriebsmittel mit anderen Prozessen aus seiner Gruppe verbunden und zum Zusammenwirken aufgefordert ist.
Die betreffenden Prozesse stehen in ↑Konkurrenz um die Zuteilung des Betriebsmittels, das allerdings nur exklusiv vergeben werden darf. Folglich wird in dieser Lage nur einer der konkurrierenden Prozesse das Betriebsmittel erhalten. Alle anderen Prozesse müssen warten, bis das Betriebsmittel wieder freigegeben wird und sie an die Reihe kommen, das freigegebene Betriebsmittel zu belegen. Für die Reihenfolgebildung unterliegen die konkurrierenden Prozesse der ↑Synchronisierung. Typisches Beispiel dafür ist ein ↑kritischer Abschnitt.

Konsolenprotokoll (en.) ↑*console log*. Aufzeichnung der auf einem ↑Rechner ablaufenden Vorgänge (Duden) zur Darstellung auf der ↑Systemkonsole und Aufzeichnung in einer ↑Datei oder auf Druckpapier.

Konstantenfaltung (en.) ↑*constant folding*. Bezeichnung für eine Optimierungstechnik bei der ↑Kompilation. Dabei werden konstante Ausdrücke in einem ↑Programm vom ↑Kompilierer bereits zur Übersetzungszeit erkannt und ausgewertet, anstatt sie erst zur ↑Laufzeit des Programms durch den ↑Prozessor berechnen zu lassen.

Konstantenpropagation (en.) ↑*constant propagation*. Bezeichnung für eine Optimierungstechnik bei der ↑Kompilation. Dabei werden Werte von Konstanten in Ausdrücken in einem ↑Programm vom ↑Kompilierer zur Übersetzungszeit ersetzt. Diese Konstanten sind entweder im Programm daselbst kodiert oder wurden durch ↑Konstantenfaltung bestimmt.

konsumierbares Betriebsmittel (en.) ↑*consumable resource*. Bezeichnung für ein ↑Betriebsmittel, das, anders als ein ↑wiederverwendbares Betriebsmittel, nur vorübergehend (flüchtig) vorhanden ist. Eine Einheit davon gilt als *vermittelbar*, wenn sie ein Kommunikationsmittel (↑Nachricht, ↑Signal) für ↑Prozesse darstellt, oder *aufbrauchbar*, wenn sie eine durch Prozesse erschöpfbare physikalische Größe (Zeit, Energie) bildet. Erstere sind logisch in unbegrenzter Anzahl (transitorisch) vorhanden, jede Einheit davon ist verfügbar. Letztere sind (wie wiederverwendbare Betriebsmittel) nur in begrenzter Anzahl vorhanden, es gibt davon nur eine bestimmte Menge von Einheiten.
Wird eine konsumierbare Betriebsmitteleinheit von einem Prozess beansprucht oder erworben (*acquire*), hört sie in dem Moment auf zu bestehen, sie erlischt, verliert Gültigkeit, wird implizit zerstört. Einheiten eines solchen Betriebsmittels müssen erst freigesetzt (*release*) und damit erzeugt werden, um sie später konsumieren zu können. Der ein solches (insb. vermittelbares) Betriebsmittel erzeugende Prozess wird auch als *Produzent* bezeichnet, sein Gegenstück *Konsument* (Verbraucher). Die Erzeugung (insb. vermittelbarer) Betriebsmittel kann zu beliebigen Zeitpunkten geschehen, sofern der betreffende Prozess seinerseits nicht den Erwerb einer solchen Betriebsmitteleinheit erwartet und demzufolge blockiert ist. Jede freigesetzte Einheit des Betriebsmittels wird sofort verfügbar. Beispiele sind Signale der Hardware (↑IRQ, ↑NMI) oder Software (↑Semaphor, `signal(2)`), Nachrichten oder jede Form von ↑Daten, die im weitesten Sinne ↑Ein-/Ausgabe eines Prozesses darstellen.
Eine aufbrauchbare Betriebsmitteleinheit ist allgemein eine physikalische Größe. Typisch dafür ist die Zeit, für die ein Prozess vor allem ein wiederverwendbares Hardwarebetriebsmittel (↑Prozessor, ↑Hauptspeicher, ↑Peripherie) zugeteilt bekommt oder der Energiebedarf, den er bei Inanspruchnahme dieses Betriebsmittels generiert. Indem der Prozess abläuft, verstreicht Zeit und die dabei beanspruchten Hardwarebetriebsmittel „verbrauchen“ elektrische Energie, wandeln diese um in Wärme, verursachen Kosten bei der Elektrizitätsversorgung. Je nach Art des Energieträgers oder wenn eine eigenständige Gewinnung (*energy harvesting*) ausscheidet steht elektrische Energie für gewöhnlich nicht unbegrenzt zur Verfügung, was insbesondere für batteriebetriebene, mobile ↑Rechensysteme relevant ist. Dies betrifft allerdings auch stationäre, ans Elektrizitätsnetz fest angeschlossene Rechensysteme, wenn nämlich eine „ungeplante Versorgungsunterbrechung“ (Stromausfall) dem Rechensystem nur noch ein Restenergiefenster begrenzter Größe für Sicherungsmaßnahmen zur Verfügung steht.

Anmerkung Die hier vorgenommene Unterklassifizierung in aufbrauchbare und vermittelbare Betriebsmittel ist eher ungewöhnlich, verfeinert aber die herkömmliche Einteilung und passt diese insbesondere vor dem Hintergrund eines nachhaltigen Rechnerbetriebs (Stichwort: *Green IT*) entsprechend an. Als konsumierbar gelten in der klassischen Fachliteratur gemeinhin lediglich die hier als vermittelbar bezeichneten Kommunikationsbetriebsmittel wie Nachrichten oder Signale. Sofern nicht weiter spezifiziert, gilt diese „einfache Sichtweise" auch weiterhin. Jedoch wird gerade elektrische Energie zunehmend als ein zwingend vorhandener Bestand an Ressourcen für den Rechnerbetrieb begriffen, der an relevanten Stellen im ↑Betriebssystem zu verwalten ist. Dies betrifft allgemein die ↑Betriebsmittelverwaltung, aber spezielle auch die ↑Prozesseinplanung. Letztere kann gezielt einen energiebewussten Rechnerbetrieb unterstützen. So wird etwa mit „*race to sleep*" ein Ansatz verfolgt, Prozesse bei höchstmöglicher Prozessorgeschwindigkeit (↑*clock speed*) stattfinden zu lassen, um im Abschluss in einen (langen) Ruhezustand mit geringem Stromverbrauch einzutreten (↑*sleep state*). Im Gegensatz dazu wird bei „*crawl to sleep*" der Prozessor eher bei niedriger Geschwindigkeit betrieben, um den durchschnittlichen Energiebedarf reduzieren zu können. Welche dieser Verfahrensweisen zur Minimierung des Energiebedarfs zu bevorzugen ist, hängt nicht nur ab von der Hardware, sondern auch von der Software, also dem Programm, das den betreffenden Prozess bestimmt. Derartige Aspekte sind sowohl bei mobilen als auch stationären Rechensystemen bedeutsam, sie haben unter anderem Einfluss auf die Batterielaufzeit beziehungsweise Dimensionierung eines Kühlungssystems. Umgekehrt sollten Prozesse daher auch entsprechend solcher Randbedingungen Energie und Zeit konsumieren, aufbrauchen dürfen.

Kontext (en.) ↑*context*. Umgebendes Etwas einer ↑Entität, beispielsweise (a) der ↑Handlungsstrang eines ↑Prozesses, (b) ein ↑kritischer Abschnitt, die ↑kritische Region oder der ↑Monitor, worin sich der Prozess gegenwärtig befindet, (c) die ↑Schutzdomäne oder der ↑Adressraum des Prozesses, (d) sein ↑Laufzeitkontext, (e) der den Prozess beherbergenden ↑Prozessor oder ↑Rechenkern oder (f) auch das ↑Rechensystem, in dem der Prozess stattfindet.
Ein bestimmter inhaltlicher Sinnzusammenhang, in dem die ↑Aktion oder ↑Aktionsfolge eines Prozesses (insb. ↑gleichzeitiger Prozesse) steht, und der Sach- und Situationszusammenhang, aus dem heraus sie verstanden werden muss (in Anlehnung an den Duden).

Kontextwechsel (en.) ↑*context switch*. Eine gegebenenfalls nach gewissen Gesetzen öfter oder immer wieder vor sich gehende Veränderung in bestimmten Geschehnissen (in Anlehnung an den Duden), die in eigenen ↑Kontexten stattfinden. Typisches Beispiel ist ein ↑Prozesswechsel (Veränderung), wodurch sich für gewöhnlich ein anderer ↑Kontrollfluss (Geschehnis) ergibt, der bestimmte ↑Ereignisse (Geschehnis) verursacht. Ein anderes Beispiel ist der ↑Systemaufruf (Veränderung), durch den ein ↑Handlungsstrang (Geschehnis) des ↑Maschinenprogramms bestimmte ↑Aktionsfolgen (Geschehnis) im ↑Betriebssystem bewirkt. Ebenso der Aufruf (Veränderung) eines ↑Unterprogramms, der einen Handlungsstrang (Geschehnis) auf tieferer ↑Abstraktionsebene hervorbringt.
Wichtiger Aspekt bei dem Vorgang ist die Sicherstellung der *Konsistenz* des Kontextes, der durch den Wechsel jeweils verlassen wird. Dies kann implizit durch eine geeignete Struktur und den Berechnungsvorschriften eines ↑Programms gegeben sein (↑*reentrancy*). Für gewöhnlich ist jedoch explizit für eine *Kontextsicherung* zu sorgen, um zu gegebener Zeit den an der „Wechselstelle" gültigen alten Zustand wieder herstellen und, nach dem Wechsel zurück, im zuvor hinterlassenen Kontext weiter operieren zu können. Im Falle eines Unterprogrammaufrufs geschieht solch eine Sicherung durch den ↑Aktivierungsblock, wohingegen beim Prozesswechsel für gewöhnlich der ↑Koroutinenstatus des den ↑Prozessor abgebenden ↑Prozesses in oder über seinen ↑Prozesskontrollblock vermerkt wird. Im Zuge eines solchen Vorgangs wird immer der zuvor gesicherte Kontext wiederhergestellt, wodurch das ↑Hauptprogramm wieder ins Spiel kommt oder eine andere ↑Koroutine desselben oder eines anderen Prozesses zur Ausführung gelangt.
Die Kontextsicherung geschieht immer in dem Kontext, der durch den Wechsel verlassen wird. Demgegenüber kann die Wiederherstellung eines Kontextes entweder in ihm selbst

geschehen, und zwar direkt nachdem er aktiviert wurde (vgl. S. 251), oder sie ist die letzte ↑Aktion des Kontextes, der durch den Wechsel verlassen werden soll. Die erste Variante sieht sowohl in der Datenstruktur als auch den Operationen, um einen Kontext zu sichern und wieder herzustellen, strikt kontextspezifische Merkmale, die von Kontext zu Kontext sehr wohl verschieden sein können: das Wechselverfahren ist heterogen, spezialisiert für den jeweiligen Kontext. Im Gegensatz dazu sind diese Merkmale in der zweiten Variante für alle Kontexte gleich: das Wechselverfahren ist homogen, allgemein für alle Kontexte brauchbar. Dabei ist allerdings zu beachten, dass in beiden Varianten der Wechsel zwischen Kontexten derselben Abstraktionsebene gemeint ist. Werden beispielsweise mehrere solcher Ebenen übereinander angeordnet betrachtet, so kann auf tieferer Ebene ein allgemeines Wechselverfahren vorliegen, das auf höherer Ebene ein spezialisiertes Wechselverfahren ermöglicht. Ein Beispiel dafür ist ein allgemein ausgelegter ↑Koroutinenwechsel tieferer Ebene (S. 151) als Grundlage für einen spezialisierten Prozesswechsel höherer Ebene (S. 251).

Kontrollfluss (en.) ↑*control flow*. Bezeichnung für die zeitliche Abfolge von ↑Maschinenbefehlen, die sich bei der Ausführung eines ↑Programms ergeben kann. Für ein ↑sequentielles Programm ist diese Abfolge allein durch die Reihenfolge der Befehle innerhalb des Programms vorgegeben. In dem Fall ist die Befehlsabfolge vor ↑Laufzeit des Programms analytisch bestimmbar, sofern für die Analyse dann auch die von dem Programm zu verarbeitenden ↑Daten bekannt sind. Demgegenüber ist die Befehlsabfolge für ein ↑nichtsequentielles Programm für gewöhnlich nicht im Vorhinein bestimmbar, da hierzu auch ↑Vorwissen zu jedem einzelnen möglichen ↑Handlungsstrang erforderlich ist. Dies impliziert Wissen über den exakten Zeitpunkt, wann welcher ↑Prozess stattfindet, wie lange er seiner jeweiligen Tätigkeit nachkommt, ob er einem bestimmten Prozess nachfolgt oder gleichzeitig mit anderen Prozessen geschieht und wie letzteres genau ausgeprägt ist, nämlich ob die gleichzeitigen Prozesse nebeneinander auf verschiedenen Prozessoren beziehungsweise verschränkt oder überlappend auf demselben Prozessor ihren ↑Aufgaben nachkommen. Hier sind insbesondere ↑asynchrone Ausnahmen jeder Art eingeschlossen. Solch umfängliche Kenntnis über die dynamischen Geschehnisse innerhalb eines nichtsequentiellen Programms bleibt vor Laufzeit vollkommen ungewiss und ergibt sich erst, wenn überhaupt, bei Ausführung dieses Programms.

Kontrollstruktur (en.) ↑*control structure*. Spezifikation der für die ↑Interpretation benötigten Algorithmen und der Transformation der Informationsträger einer (realen/virtuellen) Maschine, ein Aspekt in ihrem ↑Operationsprinzip.

Konvoieffekt (en.) ↑*convoi effect*. Bild für die Auswirkung einer nach ↑FCFS vorgehenden ↑Ablaufplanung, wenn ein Mix von kurzen und langen ↑Prozessen gegeben ist. Dabei entspricht die Prozesslänge der Dauer von einem ↑Rechenstoß, das heißt, der jeweils (zu erwartenden) ↑Bedienzeit des Prozesses durch den ↑Prozessor. Dem Effekt liegt die Annahme zugrunde, kurze Rechenstöße seien ein-/ausgabeintensive (d.h., interaktive, ↑I/O *bound*) und lange Rechenstöße seien rechenintensive (d.h., nicht interaktive, ↑CPU *bound*) Prozesse.
Wenn nun Prozesse kurzer Rechenstöße vergleichsweise lange auf der ↑Bereitliste stehen, weil sie erst die Freigabe des Prozessors durch einen Prozess mit vergleichsweise langem Rechenstoß abwarten müssen, kann in der jeweiligen ↑Wartezeit kein ↑Ein-/Ausgabestoß von ihnen gestartet werden: die von ihnen beanspruchte ↑Peripherie liegt in der Zeit brach, wenn sie nicht noch mit einer vorangegangenen ↑Aufgabe beschäftigt ist. Gibt der lange Prozess den Prozessor ab, werden die kurzen Prozesse nach und nach ihre Zeit nutzen, vornehmlich Ein-/Ausgabestöße zu starten, um daraufhin ihrerseits den Prozessor wieder abzugeben. In der Zwischenzeit sei ein langer Prozess, gegebenfalls derselbe wie zuvor, wieder an der Reihe. Dieser belegt den Prozesor, nachdem der letzte der kurzen Prozesse die Kontrolle abgegeben hat. Während der lange Prozess rechnet werden die kurzen Prozesse wieder bereitgestellt, müssen abermals auf die Zuteilung des Prozessors warten, können daher ihre Ein-/Ausgabestöße nicht absetzen und lassen die Peripherie damit erneut unter- oder unbeschäftigt. Dieser Zyklus wiederholt sich, solange der bestehende Prozessmix unverändert bleibt. Die Folge ist, dass der Prozessor zwar durchgehend beschäftigt ist, die Peripherie jedoch phasen-

weise immer nichts zu tun bekommt — dies obwohl Prozesse bereitstehen, die der Peripherie Ein-/Ausgabeaufträge zustellen würden, jedoch nicht können, weil sie eben noch nicht an der Reihe sind, den Prozessor zu erhalten.
Die kurzen Prozesse bilden einen Verband von zusammenhängenden Aufgaben für die Peripherie, der allerdings immer erst dann vorankommt, wenn der lange Prozess den Prozessor verlässt. Der Prozessor gleicht einer einspurigen Straße (↑wiederverwendbares Betriebsmittel), auf der die schnelleren Autos (kurze Prozesse) darauf warten müssen, dass der langsamere Laster (langer Prozess) vor ihnen sie vorbeilässt (eine Ausweichstelle benutzt). Dieses Verkehrsszenario ist der Ursprung für die Namensgebung des beschriebenen Effekts.

Kooperation (en.) ↑*cooperation*. Zusammenarbeit von ↑Prozessen, um gemeinsam eine bestimmte Sache zu bewirken oder ↑Aufgabe zu erledigen. Zusammenbringen von Handlungen zweier oder mehrerer Prozesse/Systeme, sodass die Wirkungen der Handlungen zum Nutzen aller dieser Prozesse/Systeme führen (in Anlehnung an das Wiktionary).
Ein typisches Beispiel liefert die ↑kooperative Planung, bei der die Prozesse bereitwillig die Kontrolle über den ↑Prozessor an andere Prozesse weitergeben, um allen Prozessen im System Fortschritt zu ermöglichen. Überhaupt setzen alle Verfahren zur ↑Synchronisation auf kooperatives Verhalten der beteiligten Prozesse.

kooperative Planung (en.) ↑*cooperative scheduling*. Modell der ↑Ablaufplanung, die (im Gegensatz zu ↑präemptive Planung) davon ausgeht, dass ↑Prozesse stets bereitwillig und in Zusammenarbeit mit dem ↑Betriebssystem die Kontrolle über den ↑Prozessor ab- oder weitergeben. Dazu sichern die Prozesse logisch zu, von Zeit zu Zeit einen ↑Systemaufruf zu tätigen, dadurch dem Betriebssystem die Möglichkeit zur ↑Umplanung zu geben und somit anderen Prozessen den Prozessor zukommen zu lassen — dies bedingt jedoch eine einen Systemaufruf bewirkende ↑Aktion kodiert im ↑Maschinenprogramm daselbst. Dieser Systemaufruf ist entweder direkt der Ablaufplanung gewidmet oder betrifft die Durchführung einer beliebigen anderen ↑Systemfunktion. Im letzteren Fall wird (insb. vom ↑Systemaufrufzuteiler) jedoch immer auch die Ablaufplanung gestreift, die dann indirekt durch den Systemaufruf ihre Funktion ausüben kann.
Diese strikt kooperative Vorgehensweise schließt allerdings durch ↑Unterbrechungen ausgelöste asynchrone Abläufe im Betriebssystem nicht aus. Auch kann die Ablaufplanung als Folge solcher Unterbrechungen sehr wohl tätig werden, jedoch wird sie niemals sofort die ↑Einlastung der ↑CPU mit einer ↑Prozessinkarnation, das heißt, einen ↑Prozesswechsel nach sich ziehen: ↑Einplanung und Einlastung sind zwar räumlich gekoppelt, aber zeitlich entkoppelt. Sollte die Einplanung einen anderen als den derzeit als <u>laufend</u> (↑Prozesszustand) bekannten Prozess bevorzugen, wird die Einlastung und damit der Prozesswechsel nicht vor dem nächsten Systemaufruf vollzogen. Bleibt ein Systemaufruf aus, erhält kein anderer Prozess die CPU zugeteilt: der gegenwärtig stattfindende Prozess monopolisiert damit die CPU.

Koordination (en.) ↑*coordination*. Aufeinander abgestimmt sein von Vorgängen, bestimmte gemeinsame Ordnung innerhalb einer Gruppe von ↑Prozessen. Ergebnis der erfolgreichen ↑Koordinierung ↑gleichzeitiger Prozesse.
Die dazu in einem ↑Rechensystem vorhandenen Maßnahmen greifen unterschiedlich. Zum einen die ↑Ablaufplanung, die einen Prozess anderen Prozessen nebenordnet bevor dieser seine Tätigkeit (erneut) aufnimmt. Sind für die Prozesse alle Daten-, Kontrollfluss- und Zeitabhängigkeiten vorab bekannt, ist ein (statischer) ↑Ablaufplan möglich, durch den die Prozesse implizit koordiniert stattfinden werden. Fehlt jedoch dieses Wissen im Moment der ↑Prozesseinplanung oder verbietet eine zu hohe Berechnungskomplexität die mitlaufende (*on-line*) Erstellung und Fortschreibung eines solchen Plans, ist zum anderen explizite ↑Nebenläufigkeitssteuerung der dann stattfindenden Prozesse erforderlich. Letztere unternehmen die Prozesse selbst, indem sie Einvernehmen über den weiteren Ablauf in Abhängigkeit der von ihnen gerade beabsichtigen ↑Aktion oder ↑Aktionsfolge erzielen und sich so nebenordnen.

Koordinierung (en.) ↑*coordination*. Herstellung von ↑Koordination innerhalb einer Gruppe von ↑Prozessen. Dies geschieht einerseits durch ↑Prozesseinplanung. In dem Fall sorgt der ↑Planer dafür, dass für die betreffenden Prozesse Koordination *implizit* bei deren Ablauf sichergestellt ist. Hierzu können die Prozesse jeweils ein ↑sequentielles Programm zur Grundlage haben, das heißt, die die Prozesse definierenden ↑Programme enthalten keine Konstrukte zur Formulierung explizit paralleler Abläufe. Demgegenüber setzt implizit sichergestellte Koordination ↑Vorwissen über den zeitlichen Verlauf der betreffenden Prozesse voraus. Fehlt diesbezügliches Vorwissen haben andererseits die Prozesse selbst, und zwar *explizit* durch entsprechende ↑Aktionen, für Koordination sorgen. Das jedoch impliziert, dass die Prozesse durch ein ↑nichtsequentielles Programm definiert sein müssen.

Koordinierungsmittel Bezeichnung für ein Instrument oder eine Maßnahme, um gleichzeitige Prozesse, die über eine gemeinsame ↑Ressource unter sich gekoppelt sind, aufeinander abstimmen, miteinander in Einklang bringen, nebenordnen zu können. Typische Beispiele sind der ↑Semaphor, die ↑Umlaufsperre, eine ↑Handlungssperre, aber auch eine ↑atomare Operation (z.B. ↑CAS, ↑FAA, ↑FAS, ↑TAS) oder Spezialbefehle (z.B. ↑LL und ↑SC) der ↑CPU.

Koroutine (en.) ↑*coroutine*. Bezeichnung für eine spezielle ↑Routine, die zusammen mit (lat. *con*) anderen Routinen ihrer Art auf derselben Stufe steht, einen eigenständigen Programmfluss beschreibt. Eine Art „Nebenprogramm“, das zwar die Rolle von einem ↑Hauptprogramm einnimmt, aber in Wirklichkeit kein solches Hauptprogramm darstellt: um abzulaufen, wird keine dieser Routinen aufgerufen, jedoch kann, wechselseitig und in kooperativer Art und Weise, jede die andere fortsetzen (↑*resume*).

Zur Fortsetzung einer so gleichgestellten Routine unterbricht die laufende Routine (↑Prozess) ihre Ausführung, um diese auf Veranlassung einer anderen Routine dieser Art später wieder aufzunehmen. Dabei definiert der Pfad bis zum jeweils nächsten Unterbrechungspunkt innerhalb einer solchen Routine einen autonomen ↑Handlungsstrang. Je nach der inneren Struktur sind mehrere dieser Stränge innerhalb einer solchen Routine möglich, die in Abhängigkeit von dem jeweils definierenden Pfad verschieden sein können.

Ein einfaches Beispiel zeigen die folgenden beiden ↑Programme in ↑C. Hier ist der Rumpf einer solchen wechselseitig betreibbaren Routine programmiersprachlich als ↑Unterprogramm (`niam`) ausformuliert, wobei die ↑Handhabe für den entsprechenden Handlungsstrang innerhalb dieser Routine als untypisierter Zeiger (`void*`) ausgelegt ist:

```
typedef void* coroutine_t;              /* handle type */

coroutine_t __attribute__((noreturn)) niam (coroutine_t root) {
    static volatile coroutine_t peer;  /* own variable */
    peer = root;                       /* remember origin */
    printf("niam(%p) at %p\n", root, niam);
    for (;;) {
        printf("niam: resume %p\n", peer);
        peer = resume(peer);           /* switch coroutine */
    }
}
```

Diese Routine beschreibt zwei Handlungsstränge. Der erste Strang reicht vom Routinenanfang bis zum ersten Aufruf der Funktion `resume` (↑Koroutinenwechsel) innerhalb der Endlosschleife und der zweite Strang reicht von der Rückkehr aus dieser Funktion bis zum erneuten Aufruf (einzelner Schleifendurchlauf). Zwischen diesen Handlungssträngen ist die betreffende Routine (`niam`) inaktiv.

Das Attribut `noreturn` hat sowohl einen dokumentarischen als auch einen technischen Zweck. Einerseits weist es darauf hin, dass gleichgestellte Routinen (wie `niam`) einen Aufruf nicht durch eine Rückkehr beantworten dürfen. Andererseits bedingt dieses Attribut eine Warnmeldung durch den Kompilierer (`gcc`), sollte dennoch fälschlicherweise ein Rückkehrpfad aus der Routine existieren. Eingangsparameter (`root`) ist eine Handhabe, die die erzeugende und

initial aufrufende Routine identifiziert. Mit dieser kann der wechselseitige Betrieb aufgenommen werden. Der lokale Zustand des Handlungsstrangs ist in einer ↑Eigenvariablen (`peer`) gespeichert und bleibt damit während Phasen seiner Inaktivität erhalten. Im Rahmen einer Endlosschleife wird diese ↑Variable hier lediglich als Handhabe genutzt, um die jeweils den Handlungsstrang (in `niam`) aktivierende ↑Entität sicher reaktivieren zu können.
Nachfolgende Programmskizze zeigt das entsprechende Pendant, sie beschreibt das Hauptprogramm (`main`) im eigentlichen Sinne beziehungsweise die ursprüngliche (`niam`) gleichgestelle Routine:

```
int main (int argc, char *argv[]) {
    static volatile coroutine_t next;  /* own variable */
    printf("main(%d) at %p\n", argc, main);
    next = assume(niam);               /* make coroutine from routine */
    while (argc--) {
        printf("main: resume %p\n", next);
        next = resume(next);           /* switch coroutine */
    }
}
```

Diese Routine beschreibt vier Handlungsstränge. Der erste Strang reicht vom Routinenanfang bis zum Aufruf der Funktion `assume` (↑Koroutinenaufbau). Mit Rückkehr aus dieser Funktion startet der zweite Handlungsstrang, der bis zum ersten Aufruf der Funktion `resume` innerhalb der Kopfschleife reicht. Der dritte Strang reicht von der Rückkehr aus und bis zum wiederholten Aufruf von `resume`, sofern die Kopfschleife mehr als einmal durchlaufen wird ($argc > 0$). Schließlich der vierte Handlungsstrang, der mit der Rückkehr aus der Funktion `resume` beginnt, die Kopfschleife verlässt ($argc = 0$) und die Ausführung des Programms beendet. Zwischen diesen Handlungssträngen ist die Routine (`main`) inaktiv.
Auch hier wird zur Speicherung des lokalen Datenzustands eine Eigenvariable (`next`) verwendet: wie im anderen Fall auch, enthält diese Variable eine Handhabe für den jeweils als nächstes zu aktivierenden Handlungsstrang. Damit in einer Routine autonome Handlungsstränge überhaupt möglich sind, muss diese die Eigenschaft zum wechselseitigen Betrieb übernehmen: ein Aufruf der Funktion `assume` bewirkt, dass die als ↑tatsächlicher Parameter (`niam`) spezifizierte Routine eben diese Eigenschaft übertragen wird (↑Koroutinenaufbau). Im konkreten Beispiel erhält die Routine `niam` diese Eigenschaft, die jedem ihrer Handlungsstränge dadurch ermöglicht, mit einem Handlungsstrang in der Routine `main` wechselseitig, durch Verwendung der Funktion `resume`, ablaufen zu können.
Da keine dieser gleichgestellten Routinen gewöhnlich aufgerufen wird, können sie auch nirgends hin zurückkehren: der Rückkehrversuch stellt eine ↑Ausnahmesituation dar, die den weiteren Verlauf des entsprechenden Handlungsstrangs undefiniert lässt. Beabsichtigt eine solche Routine dennoch die Beendigung, macht sie dies durch einen entsprechenden globalen Zustand deutlich, unterbricht die Ausführung und gibt dadurch Möglichkeit zur Entsorgung von anderer Stelle (durch eine andere Routine).

Koroutinenaufbau Errichtung einer ↑Koroutine auf Grundlage einer gewöhnlichen ↑Routine, der dazu die Fähigkeit zur gleichgestellten Ausführung im Zusammenspiel mit anderen Koroutinen derselben Art übertragen wird. Wenn möglich sind Vorkehrungen zur ↑Abfangung der errichteten Koroutine zu treffen, sollte diese unerwarteterweise einen Rückkehrpfad aus der sie definierenden Routine nehmen. Nach erfolgter Übernahme der für diese Form der Ausführung nötigen Eigenschaften schreitet die Ko-/Routine immer wechselseitig und in kooperativer Weise mit ihresgleichen voran (↑*resume*): sie unterbricht ihre Ausführung von Zeit zu Zeit, beendet sich jedoch nicht von selbst. Damit diese wechselseitigen Abläufe möglich sind, ist der den Ausgangspunkt bildenden *Basisroutine*:

1. die Fortsetzungsadresse einer bereits bestehenden Koroutine zu vermitteln, um zu wissen, wohin sie im Falle ihrer Ausführungsunterbrechung wechseln kann und

2. ein Zustand zu geben, der die Fortsetzung ihrer unterbrochenen Ausführung ermöglicht.

Wünschenswertes Kriterium der Maßnahmen ist es, den ↑Koroutinenwechsel unabhängig von seiner jeweiligen Abfolge stattfinden zu lassen, nämlich nicht zwischen den ersten (initialen) und allen weiteren (normalen) Wechseln unterscheiden zu müssen. Dazu muss der Routine der ↑Koroutinenstatus einer scheinbar zuvor bereits gelaufenen und vorübergehend unterbrochenen Koroutine gegeben werden. Folglich unterscheidet sich der durchzuführende beziehungsweise zu hinterlassene Aufbau je nach Art der zu errichtenden Koroutine (↑primitive Koroutine, ↑komplexe Koroutine).

Primitive Koroutine Wesentliches Merkmal dieser Art von Koroutine ist der allen ↑Exemplaren eben dieses elementaren Typs gemeinsame ↑Laufzeitstapel. Dadurch eröffnen sich grundsätzlich zwei Wege für die Koroutineneinrichtung, die dann nämlich einerseits fremd- und anderseits selbstgesteuert geschehen kann. Im fremdgesteuerten Fall bringt eine beliebige andere Ko-/Routine die Basisroutine dazu, zukünftig als Koroutine zu funktionieren. Dazu wird die Basisroutine derart durch eine ↑Mantelprozedur aufgerufen, dass sie ihre Ausführung unterbrechen und zu einer anderen Koroutine umschalten kann. Die ↑Handhabe für den ↑Handlungsstrang dieser Koroutine hat die Basisroutine beim initialen Aufruf als Parameter übergeben bekommen.
Im selbstgesteuerten Fall sorgt die Basisroutine von sich aus für die Metamorphose. Dazu wird diese Routine bereits durch einen regulären Koroutinenwechsel aktiviert und muss dann allerdings die Handhabe des Handlungsstrangs einer anderen Koroutine selbst in Erfahrung bringen, um zu letzterer irgendwann hinschalten zu können: ein Handlungsstrang innerhalb einer Koroutine hat für gewöhnlich keinen Parameter, weshalb im Fall der initialen Aktivierung die benötigte Handhabe, anders als bei der fremdgesteuerten Variante, auch nicht als Aufrufparameter übergeben werden kann. Stattdessen ist diese Handhabe durch Ausführung einer speziellen *Übernahmefunktion* in Erfahrung zu bringen. Der Einfachheit wegen bietet es sich dann an, die Handhabe desjenigen Handlungsstrangs derjenigen Koroutine zu liefern, aus dem heraus der initiale Koroutinenwechsel geschah.
Die Beschreibung macht deutlich, dass die Einrichtung einer Koroutine keine einfache Operation ist. Typischerweise gestaltet sich diese Operation schwieriger als der Koroutinenwechsel selbst. Ein Beispiel für eine solche Operation liefert nachfolgende Skizze einer entsprechenden Funktion, die für gewöhnlich ein in ↑Assembliersprache (↑GAS) formuliertes ↑Unterprogramm erfordert, da die durchzuführenden ↑Aktionen vom jeweiligen ↑Programmiermodell des zugrunde liegenden Prozessors (↑x86) abhängen:

```
assume:                 # use subroutine call: peer = assume(0 or <name>)
    cmpl $0, 4(%esp)    # check for role of calling co-/routine
    je 1f               # bypass in case of self-controlled setup
    pushl %ebx          # save non-volatile processor state
    pushl %ebp          # "
    pushl %esi          # "
    pushl %edi          # "
    movl %esp, %ebp     # save top of stack
    movl $2f, %eax      # pass home address for self-controlled setup
    pushl %eax          # do similar for pilot-controlled setup
    call *24(%esp)      # perform initial activation for both cases
    xorl %eax, %eax     # should never return to here: error code
2:                      # come here in the case of resumption
    movl %ebp, %esp     # restore top of stack
    popl %edi           # restore non-volatile processor state
    popl %esi           # "
    popl %ebp           # "
    popl %ebx           # "
1:                      # come here in the case of bypass
    ret                 # continue as coroutine or fail
```

Die gezeigte Funktion (`assume`) ist sowohl für die fremd- als auch für die selbstgesteuerte Umwandlung einer Routine in eine Koroutine geeignet. Ein Beispiel für den fremdgesteuerten Ansatz ist bereits an anderer Stelle (S. 144) behandelt worden. Wird dieser Weg gegangen, bildet `assume` die Mantelprozedur für den Aufruf der Basisroutine (`call *24(%esp)`).
Im selbstgesteuerten Ansatz wird davon ausgegangen, dass die Basisroutine durch eine `resume`-Aktion gestartet wurde. In diesem Fall entspricht die von dieser Routine auzurufende Funktion `assume` praktisch einem ↑Leerbefehl, sie umgeht die ↑Aktionsfolge zur fremdgesteuerten Einrichtung einer Koroutine und kehrt sofort wieder zurück (↑Sprungmarke 1). Hier ist zu beachten, dass in diesem Fall die `assume`-Funktion dann genau den Wert zurück liefert, den eine von diesem Handlungsstrang sonst aufgerufene `resume`-Funktion zurückgeliefert hätte — letztere mangels Eingabeparameter für `resume` in der Initialisierungsphase der neuen Koroutine jedoch noch nicht aufgerufen werden kann. Dieser Rückgabewert ist die Handhabe desjenigen Koroutinenhandlungsstrangs, der `resume` aufgerufen hat, um den nunmehr laufenden Handlungsstrang der gerade neu entstehenden Koroutine zu aktivieren.
Um `assume` im fremdgesteuerten Ansatz auch unabhängig davon benutzen zu können, welchem Model die Basisroutine nun folgt, wird sowohl der Rückgabewert für die selbstgesteuerte Übernahme (`assume(0)`) aufgesetzt (`movl $2f, %eax`) als auch der Eingabeparameter für die fremdgesteuerte Übernahme (`assume(<name>)`) übergeben (`pushl %eax`). Darüber hinaus liefert die Funktion (durch `xorl %eax, %eax`) den Rückgabewert null, wenn die initial aufgerufene Routine, die eine Koroutine werden sollte, aus dem Aufruf zurückkehrt.

Komplexe Koroutine Anders als im Fall der zuvor behandelten Koroutinenart verfügen die Exemplare einer komplexen Koroutine über einen eigenen ↑Laufzeitstapel. Dies hat zur Folge, dass nur noch die fremdgesteuerte Einrichtung praktiziert werden kann, da eine komplexe Koroutine ohne eigenen Laufzeitstapel nicht ablauffähg ist und daher auch der Aufbau aus eigener Kraft, wie für den selbstgesteuerten Fall notwendig, ausscheidet.
Der koroutineneigene Laufzeitstapel legt die Grundlage für die physische Entkopplung der Handlungsstränge verschiedener Koroutinen. Die Entkopplung zeigt sich insbesondere in der Eigenschaft, die Koroutine initial durch einen gewöhnlichen Prozeduraufruf betreten zu können, der ihr damit einen Rückkehrweg eröffnet. Letzterer bedeutet jedoch lediglich, den betreffenden Handlungsstrang einer solchen Koroutine „einzufangen“ und, wenn möglich, kontrolliert anzuhalten oder eine ↑Ausnahme zu erheben. Auf diesem Stapel werden zudem alle für die Koroutine relevanten ↑Eigenvariablen abgelegt.
Wie nachfolgendes Beispiel (GAS, x86) zeigt, hat die Operation zur Einrichtung einer kom-

plexen Koroutine gewisse Gemeinsamkeiten mit dem Aufbau einer primitiven Koroutine. Dies betrifft jedoch nur die Struktur in drei Blöcken von Maschinenbefehlen, wobei die beiden äußeren zum Sichern und Wiederherstellen des relevanten ↑Prozessorstatus' identisch sind mit der zuvor behandelten Funktionsvariante. Der mittlere Befehlsblock ist spezifisch für den Aufbau einer komplexen Koroutine, er sorgt für (1) den mit eigenem Laufzeitstapel versehenen Aufruf der Basisroutine, (2) die Vermittlung der Fortsetzungsadresse der laufenden Ko-/Routine, wohin die aufgebaute Koroutine durch einen Koroutinenwechsel umschalten kann und (3) die Abfangung, sollte die Basisroutine aus ihrem Aufruf zurückkehren:

```
assign:                 # use subroutine call: peer = assign(name, hook)
    pushl %ebx          # save non-volatile processor state
    pushl %ebp          # "
    pushl %esi          # "
    pushl %edi          # "
    pushl $2f           # setup own resumption address
    movl %esp, %ebx     # setup and backup own coroutine handle
    movl 28(%esp), %esp # switch to stack of new (virgin) coroutine
    pushl %ebx          # pass handle of parent coroutine
3:                      # come here after return from halt: re-enter
    call *24(%ebx)      # perform initial coroutine activation
    hlt                 # should never return to here: halt
    jmp 3b              # should never return from halt: loop
2:                      # come here in the case of resumption
    popl %edi           # restore non-volatile processor state
    popl %esi           # "
    popl %ebp           # "
    popl %ebx           # "
    ret                 # continue as coroutine
```

Zunächst generiert die die `assume-Funktion` aufrufende Ko-/Routine ihre eigene Fortsetzungsadresse und hinterlässt diese auf ihrem Laufzeitstapel (`pushl $2f`). Daraufhin definiert sie die Handhabe (`movl %esp, %ebx`), die später ↑tatsächlicher Parameter der Basisroutine wird. Anschließend erfolgt die Umschaltung hin zum Laufzeitstapel der aufzubauende Koroutine (`movl 28(%esp), %esp`), die Parameterübergabe (`pushl %ebx`) an die Basisroutine sowie ihr Aufruf (`call *24(%ebx)`). Mit dem Basisroutinenaufruf startet der erste Handlungsstrang in der neu eingerichteten Koroutine.
Sollte die Basis-/Koroutine unerwarteterweise aus dem Aufruf zurückkehren, wird der betreffende Handlungsstrang abgefangen. Die radikale Lösung hierfür ist es, die CPU einen ↑Haltebefehl (`hlt`) ausführen zu lassen. Auf der hier betrachteten ↑Abstraktionsebene gibt es keine andere Möglichkeit zu diesem Schritt, da Wissen über die im System vorhandenen Koroutinen fehlt. Dieses Wissen ist erst auf höherer Ebene vorhanden, wenn nämlich Koroutinen die Basis für ↑Prozessinkarnationen bilden, die dann von einem ↑Planer verwaltet werden und dazu, unter anderem, auf der ↑Bereitliste geführt werden. Wenn die CPU ihren Haltezustand verlässt ist es daher auch sinnvoll, die Basis-/Koroutine erneut zu betreten (`jmp 3b`), um auf höherer Ebene Kenntnis von nunmehr gegebenenfalls vorhandenen anderen Koroutinen zu erhalten und zu einer auf dieser Liste wechseln zu können.

Koroutinenstatus Gesamtheit von ↑Daten, die den statischen Zustand einer ↑Koroutine manifestieren. Dieser Zustand umfasst alle Daten, die zur Wiederaufnahme und zum Fortsetzen einer unterbrochenen Koroutinenausführung erforderlich sind.
Unerlässliches Merkmal dieses Datenzustands ist die Fortsetzungsadresse, das heißt, der Wert des ↑Befehlszählers zum Zeitpunkt der Kontrollabgabe der Koroutine über den ↑Prozessor. Dieser Wert (↑Adresse) markiert die Stelle im ↑Programm, an der die Koroutine ihre Ausführung zuletzt unterbrochen hatte und gegebenenfalls wiederaufnehmen wird. Zusätzlich kann dieser Zustand durch die Inhalte weiterer ↑Prozessorregister wie auch lokaler

↑Variablen definiert sein, je nachdem, welcher ↑Aufgabe die Koroutine seit ihrer letzten Ausführungswiederaufnahme nachgekommen ist. All diese Zustandsdaten müssen in Phasen der Inaktivität einer Koroutine invariant sein: jede Variable, die diesen Zustand mit definiert, ist eine ↑Eigenvariable.
Zur Aufbewahrung von Eigenvariablen kann ↑statischer Speicher oder ↑dynamischer Speicher genutzt werden. Letzteres bietet sich vor allem an, wenn das Koroutinenkonzept zur Implementierung von ↑Prozessen zum Einsatz kommen soll (↑komplexe Koroutine). In diesem Fall ist insbesondere der ↑Stapelspeicher ein bedeutsames Hilfsmittel: jede Koroutine verfügt dann über einen eigenen ↑Laufzeitstapel, in dem die Eigenvariablen liegen. Bei einem ↑Koroutinenwechsel wird sodann der ↑Stapelzeiger umgesetzt, wodurch der Datenzustand für die jeweils aktivierte Koroutine automatisch wieder zur Verfügung steht.

Exkurs Einen wesentlichen Anteil an den Eigenvariablen einer Koroutine hat der ↑Prozessorstatus, genauer die Prozessorregister, die von der Koroutine im Moment des ↑Koroutinenwechsels gerade belegt sind (↑aktives Register). Dies bedeutet eine mehr oder weniger umfangreiche Umspeicherung von ↑Daten, je nachdem, wie viele diese Register im Moment des Wechsels für die Koroutine gerade aktiv sind.
Das Wissen über die gerade aktiven Register hat der ↑Kompilierer. Ist in der Programmiersprache ein Koroutinenkonzept verankert, so wird der Kompilierer bei der ↑Übersetzung nur die zur Sicherung und Wiederherstellung wirklich erforderlichen ↑Maschinenbefehle absetzen, nämlich nur um die Inhalte der im Moment des Koroutinenwechsels aktiven Register bewahren. Allerdings kennt ↑C kein solches Konzept, daher ist die Sicherung und Wiederherstellung dieser Register selbst zu programmieren. Wie viele Register dabei zu berücksichtigen sind, hängt davon ab, welcher ↑Abstraktionsebene eine Koroutine jeweils zugeordnet ist.
Typischerweise sind hier zwei Ebenen relevant, nämlich die der höheren Programmiersprache und die des ↑Maschinenprogramms. Erfolgt der Wechsel zwischen Koroutinen derselben höheren Programmiersprache, gibt die ↑Aufrufkonvention dieser Sprache den Hinweis über die unbedingt zu sichernden und wiederherzustellenden Register. Anderenfalls muss das ↑Programmiermodell des Prozessors herangezogen werden. Daher bietet sich die parametergesteuerte Registerbehandlung an, wie folgendes Beispiel einer in C für ↑x86 formulierten Sicherungsoperation zeigt:

```
inline void unload(trim_t trim, void *tank) {
    if ((trim & ~CDECL) == INNER) {                /* use run-time stack? */
        if (!(trim & CDECL)) {       /* yes, consider volatile registers? */
            asm volatile (               /* yes, backup to run-time stack */
                "pushl %eax\n\t"
                "pushl %ecx\n\t"
                "pushl %edx");
        }
        asm volatile ( /* backup non-volatile registers to run-time stack */
            "pushl %ebx\n\t"
            "pushl %ebp\n\t"
            "pushl %esi\n\t"
            "pushl %edi");
    } else if ((trim & ~CDECL) == PLAIN) {           /* use buffer store? */
        asm volatile (              /* yes, backup non-volatile registers */
            "movl %%edi, 0(%0)\n\t"
            "movl %%esi, 4(%0)\n\t"
            "movl %%ebp, 8(%0)\n\t"
            "movl %%ebx, 12(%0)" : : "r" (tank));
        if (!(trim & CDECL)) {            /* consider volatile registers? */
            asm volatile (                 /* yes, backup to buffer store */
                "movl %%edx, 16(%0)\n\t"
                "movl %%ecx, 20(%0)\n\t"
```

```
                "movl %%eax, 24(%0)" : : "r" (tank));
        }
    }
}
```

Dabei ist der ↑Datentyp `trim_t` wie folgt definiert:

```
typedef enum trim {
    NAKED, INNER, PLAIN, CDECL=(1<<31)
} trim_t;
```

Die Umspeicherung von Registerinhalten entlastet (`unload`) den Prozessor. Dazu sind letztlich drei Varianten beschrieben: `INNER`, sichert in den Laufzeitstapel; `PLAIN`, sichert in einen Pufferspeicher (`tank`); weder noch beziehungsweise `NAKED`, sichert nirgendwo hin. Darüber hinaus wird spezifiziert, ob entsprechend der Aufrufkonvention von C (↑cdecl) vorgegangen werden soll (`CDECL`). Ist dies der Fall, betrifft die Entlastung des Prozessors nur einen Teil seiner Register (↑*non-volatile register*). Anderenfalls werden zusätzlich noch die Inhalte der flüchtigen Register (↑*volatile register*) gesichert.
Die Anweisungen sind prozessorabhängig. Durch Konstantenfaltung (*constant folding*) wertet der Kompilierer die Fallunterscheidungen bereits bei der Übersetzung des ↑Unterprogramms (`unload`) aus und generiert eine ↑Aktionsfolge, die frei von Programmverzweigungen ist. Darüber hinaus hinterlässt inzeiliges assemblieren (*inline assembly*) keine weiteren Spuren im Maschinenprogramm.
Die inverse Operation folgt demselben Muster, hier sei jedoch nur ihre Signatur angegeben:

```
inline void reload(trim_t trim, void *tank) { ... }
```

Beide Operationen rahmen typischerweise die zentrale, für gewöhnlich von allen Koroutinen verwendete Anweisung zur Ausführungsunterbrechung und damit zur Prozessorabgabe ein. Ein Beispiel, bei dem dann der Zustand der nichtflüchtigen Register im Laufzeitstapel gesichert vorliegt, zeigt folgender Programmausschnitt:

```
unload(INNER|CDECL, 0);       /* backup non-volatile registers, use stack */
last = resume(next);                                  /* switch coroutines */
reload(INNER|CDECL, 0);       /* define non-volatile registers, use stack */
```

Zu beachten ist hierbei, dass eine Koroutine (`last`) die Funktion `resume` aufruft und eine andere Koroutine (`next`) aus dem Aufruf, den sie irgendwann vorher selbst getätigt hat, zurückkehrt. Der Funktionswert identifiziert dann die Koroutine, die `resume` gerade eben aufgerufen hatte. Jede dieser Koroutinen sichert ihre aktiven Register nicht nur selbst, sondern definiert sie auch für sich selbst. Keine Koroutine muss damit die aktiven Register der jeweils anderen Koroutine kennen.

Koroutinenwechsel Veränderung in dem ↑Programmablauf, Wechsel von einer ↑Koroutine zu einer anderen Koroutine: ↑*resume*. Je nach Art der Koroutine erfolgt dieser Wechsel unterschiedlich, und zwar abhängig von dem zur Adressierung eines ↑Handlungsstrangs verwendeten ↑Prozessorregister im Steuerwerk einer ↑CPU. Für eine ↑primitive Koroutine wird der Handlungsstrang durch den ↑PC adressiert, der Wechsel verläuft direkt. Im Gegensatz dazu ist der Schalthebel für eine ↑komplexe Koroutine (um letztlich ebenfalls den PC umzusetzen) der ↑SP, der Wechsel verläuft indirekt. Passend zur Wechseltechnik ist für einen initialen ↑Koroutinenaufbau zu sorgen, der es beispielsweise einer gewöhnlichen ↑Routine gestattet, als Koroutine gleichgestellt mit anderen Routinen ausgeführt werden zu können. In beiden Fällen ist die Signatur einer entsprechenden Operation jedoch gleich, beispielsweise:

```
coroutine_t __attribute__((fastcall)) resume(coroutine_t);
```

Das angegebene Attribut (`fastcall`) spezifiziert die Parameterübergabe durch Verwendung eines ↑Prozessorregisters (hier: `ecx`, ↑x86).
Um den Handlungsstrang einer zeitweilig unterbrochenen Koroutinenausführung wieder auf-

nehmen zu können, muss die Wechseloperation jede Koroutine dazu veranlassen, eine ↑Handhabe zur Wiederaufnahme der Ausführung eigenständig zu sichern und wiederherzustellen. In Bezug auf den bei Bedarf zusätzlich noch zu berücksichtigenden ↑Prozessorstatus ist jedoch auch dessen fremdgesteuerte Wiederherstellung möglich: die den Wechsel auslösende Koroutine setzt den Prozessorstatus der nachfolgenden Koroutine auf, bevor letztere ihren Handlungsstrang wieder aufnimmt. Das bedeutet aber auch, dass dann in dieser einen Wechseloperation Wissen über den Prozessorstatusumfang der jeweils nachfolgenden Koroutine verankert sein muss. Dieses Wissen liegt für gewöhnlich nicht vor, weshalb in dem Fall und unter Kontrolle der jeweils vorangehenden Koroutine immer der komplette Prozessorstatus gesichert und wiederhergestellt werden müsste.
Da Koroutinen von den Prozessorregistern unterschiedlichen Gebrauch machen können, ist auch nur die Sicherung und Wiederherstellung der Inhalte jener Prozessorregister nötig, die die jeweilige Koroutine im Moment ihrer Ausführungsunterbrechung belegt. Der nachfolgend gezeigte `resume`-Mechanism trägt diesem Aspekt Rechnung.

Primitive Koroutine Ohne entsprechende Eigenschaften der CPU (↑orthogonaler Befehlssatz) bildet die Operation eine ↑Aktionsfolge, was zudem für gewöhnlich die Formulierung in ↑Assembliersprache erfordert. Ein Beispiel für ↑GAS und x86 ist nachfolgend skizziertes ↑Unterprogramm, das eine Funktion implementiert, die (a) zu einer Koroutine hinschaltet, deren Adresse in einem Prozessorregister enthalten ist und (b) als Wert die Fortsetzungsadresse des Handlungsstrangs der wegschaltenden Koroutine liefert:

```
resume:                 # use subroutine call: last = resume(next)
    popl %eax           # return address becomes function return value
    jmp *(%ecx)         # continue other coroutine
```

Insgesamt sind damit jedoch wenigstens vier Maschinenbefehle für den Wechsel von solch einem Handlungsstrang notwendig, nicht nur zwei. Die beiden zusätzlichen Befehle bewirken den Unterprogrammaufruf:

```
    movl <next>, %ecx   # pass continuation address as parameter
    call resume         # and perform the coroutine switch
```

Dabei steht `<next>` für einen Operanden mit der Fortsetzungsadresse des Handlungsstrangs einer Koroutine, zu der hingeschaltet werden soll. Die ↑Gemeinkosten dieser Nachbildung einer ↑Elementaroperation zum Ablaufwechsel sind vergleichsweise hoch. Alternativ kann diese Operation als ↑Makrobefehl ausgelegt werden, um den Aufwand auf zwei Maschinenbefehle zu begrenzen:

```
.macro resume next      # input next (continuation address), spoil %eax
    movl $1f, %eax      # generate output value: continuation address
    jmp *(\next)        # actually switch to (i.e., continue) coroutine
    .p2align 3          # enforce aligned continuation address
1:                      # point of return, i.e., resumption
.endm                   # %eax holds the continuation address
```

Der erste (`movl`) Befehl liefert die Fortsetzungsadresse (`$1f`, d.h., Marke 1 vorwärts) der laufenden Handlungsstrangs als Ausgabeparameter (`%eax`). Diese Adresse ist der vom ↑Binder berechnete Wert einer ↑Sprungmarke (`1:`), nämlich die Stelle, an der die wegschaltende Koroutine später die Wiederaufnahme ihrer Ausführung erwartet. Der zweite (`jmp`) Befehl überträgt die Kontrolle an die Koroutine, deren Fortsetzungsadresse als Parameter (`next`) übergeben wurde. Die in diesem Makro definierte Befehlsfolge lässt sich nahezu direkt in ↑C übernehmen, und zwar durch Anweisungen zum inzeiligen Assemblieren (*inline assembly*):

```
inline coroutine_t resume(coroutine_t next) {
    register coroutine_t last asm("eax");    /* enforce register variable */
    asm volatile(                                /* start inline assembly */
        "movl $1f, %%eax\n\t"                      /* define return value */
        "jmp *%0\n\t"                          /* switch to next coroutine */
        ".p2align 3\n"              /* enforce alignment of following text */
        "1:"                                          /* come back to here */
        : : "g" (next) : "%eax");                /* specify input parameter */
    return last;                       /* deliver pointer to last coroutine */
}
```

Vorteil dieser Lösung ist die nahtlose Einbindung von Maschinenbefehlen (GAS/x86) in eine in Hochsprache (C) formulierte ↑Routine. Dabei setzt der Kompilierer (`gcc`) die in der Funktion (`resume`) verwendeten Variablen (`next`, `last`) mit passender ↑Adressierungsart automatisch als Operanden ein.
Die Implementierungen greifen das Konzept des ↑Verbindungsregisters auf, um die Fortsetzungsadresse einer Koroutine zu speichern. Mehr Statusinformationen sind auch für eine gewöhnliche Routine nicht notwendig, um nämlich die Stelle zu vermerken, an der sie die Kontrolle an eine andere Routine durch einen Maschinenbefehl zum Unterprogrammaufruf (`call`) abgegeben hat und zurückerhalten kann. Wie auch im Falle herkömmlicher Unterprogramme wird der wirkliche Datenzustand davon abhängig sein, welcher Funktion die betreffende Koroutine dient. Ist diese Funktion der ↑Prozesswechsel, muss typischerweise ein erweiterter ↑Koroutinenstatus berücksichtigt werden. Darüber hinaus ist, jeweils passend zu den hier gezeigten Funktionen zum Koroutinenwechsel, für die Einrichtung einer Koroutine derart zu sorgen, dass zu dieser auch umgeschaltet werden kann, obwohl sie noch keine Möglichkeit hatte, selbst einen Koroutinenwechsel durchzuführen (↑Koroutinenaufbau).

Komplexe Koroutine Im Gegensatz zum vorangegangem Modell, erfolgt hier die Umschaltung des ↑Stapelzeigers, um zwischen Handlungssträngen verschiedener Koroutinen hin und her zu wechseln. Beispiel einer entsprechenden Operation (x86) ist nachfolgend skizziertes Unterprogramm. Die darin beschriebene Funktion (a) schaltet hin zu einer Koroutine, deren Stapelzeiger in einem Prozessorregister enthalten ist und (b) liefert als Wert den Stapelzeiger dieser zuvor eben noch gelaufenen Koroutine:

```
coroutine_t __attribute__((fastcall)) resume(coroutine_t next) {
    coroutine_t last;                          /* return value placeholder */
    asm volatile(                                /* start inline assembly */
        "movl %%esp, %0\n\t"                       /* define return value */
        "movl %1, %%esp"                         /* switch run-time stack */
        : "=r" (last) : "r" (next));  /* specifiy input/output constraints */
    return last;                       /* deliver pointer to last coroutine */
}
```

Wird dieses Unterprogramm aufgerufen, gelangt automatisch die Fortsetzungsadresse des Handlungsstrangs der wegschaltenden Koroutine auf den Stapel: diese ist nämlich die beim Aufruf hinterlassene Rücksprungadresse. Die ↑Speicherstelle, an der diese Adresse vom Prozessor (x86) abgelegt wurde, identifiziert der Stapelzeiger (`esp`). Der für die zu aktivierende Koroutine gültige Stapelzeiger wird vor dem Aufruf noch als ↑tatsächlicher Parameter in einem ↑Prozessorregister (`fastcall`: `ecx`) übergeben.
Zu beachten dabei ist die Bedeutung des Rückgabewertes der Funktion `resume`. Dieser Wert entspricht dem Stapelzeiger der soeben weggeschalteten Koroutine und identifiziert damit die Stelle, an der für gewöhnlich auch der ↑Laufzeitkontext dieser Koroutine gesichert vorliegt. Mit diesem Rückgabewert als Parameter für einen späteren `resume`-Aufruf wird die betreffende Koroutine wieder in den Zustand gebracht, den sie zuvor inne hatte. Im Gegensatz zur primitiven Koroutine, deren Handhabe ein Befehlszählerwert ist, ist für den hier gezeigten Koroutinenwechsel allerdings auch eine dazu passende Einrichtungsprozedur zu druchlaufen

(↑Koroutinenaufbau).

kritische Region (en.) ↑*critical region*. Durch den Lese-Schreib-Zugriff ↑gleichzeitiger Prozesse auf eine ↑gemeinsame Variable gekennzeichneter räumlicher Bereich, der ein ↑nichtsequentielles Programm strukturiert. Prozesse in solchen Regionen, die sich auf dieselbe Variable beziehen, schließen sich für gewöhnlich gegenseitig aus (↑*mutual exclusion*), ein jeder aus dieser Menge gilt als ↑gekoppelter Prozess. Typischerweise ist eine solche Region als ↑kritischer Abschnitt verwirklicht.
Dieses Konzept ermöglicht beweiskräftige Aussagen über die Auswirkungen von gleichzeitigen Berechnungen. Wenn diese jedoch zu einer gemeinsamen ↑Aufgabe zusammenarbeiten, müssen sie auch warten können, bis bestimmte Bedingungen von anderen Prozessen erfüllt werden. Derartige Situationen werden durch eine ↑bedingte kritische Region behandelt.
Der ursprünglichen Idee (Hansen, 1972) nach wird eine solche Region im nichtsequentiellen Programm durch Sprachkonstrukte explizit gemacht, sie definiert damit einen bestimmten räumlichen Bereich, in dem die ↑Synchronisation gleichzeitiger Prozesse, die auf gemeinsame Variablen zugreifen, geschieht. Im weiteren Schritt generiert ein ↑Kompilierer automatisch die Synchronisationsanweisungen und sorgt dafür, dass ein der betreffenden Region gleichzusetzender kritischer Abschnitt nachweislich korrekt abgesichert wird. Programmierfehler bei der ↑Synchronisierung sind ausgeschlossen.
Funktional gleichen sich die beiden Konzepte zur Kennzeichnung solcher Bereiche — „Regionen“ beziehungsweise “Abschnitte“ — in einem nichtsequentiellen Programm, sie sind jedoch unterschiedlichen ↑Abstraktionsebenen zuzurechnen. Zwischen diesen Ebenen besteht eine ↑semantische Lücke, die vor allem durch die Verschiedenheit der sprachlichen Ausdruckshilfsmittel zur Bereichskennzeichnung definiert ist, nämlich die eher hochsprachlichen Konstrukte (`region { … }`) für die kritischen „Regionen“ einerseits und die eher niedersprachlich einzustufenden, logisch der ↑Maschinenprogrammebene und gegebenenfalls noch tieferen Ebenen zuzurechnenden Primitiven (`P/V`, `lockup/unlock`, `avert/admit` usw.) für die kritischen „Abschnitte“ andererseits.

Exkurs Zur Verdeutlichung des Unterschieds soll folgendes nichtsequentielles ↑Unterprogramm (formuliert in der fiktiven Sprache *Concurrent* ↑C) betrachtet werden:

```
void meter (shared int *ref, int val) {
    region (*ref) *ref += val;          /* indivisible read-modify-write */
}
```

Ein Typqualifizierer (***shared***) weist ein ↑Exemplar (`*ref`) des betreffenden Datentyps (`int`) als gemeinsame Variable aus. Die sich auf den räumlichen Bereich (***region***) dieser Variablen beziehende Programmanweisung (`*ref += val`) soll sequentiell durchlaufen werden, so dass Werteberechnung und -zuweisung (↑*read-modify-write*) logisch unteilbar geschehen.
Auf Grundlage solcher Sprachkonstrukte lässt sich ein solcher Bereich in einen kritischen Abschnitt umwandeln, zu dessen Schutz sich ein ↑binärer Semaphor anbietet. Zunächst wird hierzu die ↑Adresse der gemeinsamen Variablen (`ref`) zusammen mit einer passenden ↑Synchronisationsvariablen, mit der ↑wechselseitiger Ausschluss praktizierbar ist, als ↑Verbund repräsentiert:

```
typedef struct {
    int *ref;                           /* shared variable reference */
    semaphore_t mutex;                  /* critical-region ressource */
} __int_t
```

Darauf basierend ergibt sich für das Unterprogramm (a) eine leicht veränderte ↑Signatur und (b) ein entsprechend angepasster Prozedurrumpf wie folgt skizziert:

```
void __meter (__int_t *obj, int val) {
    P(&obj->mutex);                              /* enter critical section */
    *obj->ref += val;            /* safely do read-modify-write operation */
    V(&obj->mutex);                              /* leave critical section */
}
```

Bei dieser Lösung ist zu beachten, dass der Kompilierer an allen Stellen der Verwendung der betreffenden gemeinsamen Variablen den elementaren (`int`) durch den komplexen (`__int_t`) ↑Datentyp ersetzen muss. Der Typqualifizierer (*`shared`*) gibt ihm den entsprechenden Hinweis in der Signatur der ursprünglichen Formulierung des Unterprogramms.
In ähnlicher Art und Weise ließe sich auch die Region selbst qualifizieren, um etwa andere Synchronisationsverfahren oder -techniken zum Ausdruck zu bringen. Beispielsweise wäre es gut vorstellbar, die zur Diskussion stehende Anweisung wirklich atomar (*`atomic`*) ablaufen zu lassen, wenn möglich, sie auf einen einzigen ↑Maschinenbefehl zu reduzieren:

```
void meter (shared int *ref, int val) {
    atomic region (*ref) *ref += val;    /* indivisible read-modify-write */
}
```

Ein geschickter Kompilierer kann auf Grundlage solcher Sprachkonstrukte derartig einfach aufgebaute kritische Anweisungsfolgen leicht auf eine ↑Elementaroperation abbilden, hier ↑FAA, sie also als ↑atomare Operation auslegen:

```
void __meter (int *ref, int val) {
    FAA(ref, val);                                          /* fetch and add */
}
```

Ebenso ließe sich eine Region formulieren, deren Ausführung ungestört (*`quiet`*) von ↑Unterbrechungsanforderungen geschehen soll:

```
void meter (shared int *ref, int val) {
    quiet region (*ref) *ref += val;     /* indivisible read-modify-write */
}
```

Der Kompilierer setzt daraufhin Anweisungen ab, um beim Eintritt in den kritischen Abschnitt eine ↑Unterbrechungssperre zu erheben und beim Austritt wieder zurückzunehmen:

```
void __meter (int *ref, int val) {
    avert(IRQ);                                             /* disable IRQ */
    *ref += val;                 /* safely do read-modify-write operation */
    admit(IRQ);                                              /* enable IRQ */
}
```

Als weitere Spezialisierung bietet sich an, selektiv Sperren für einzelne oder Gruppen von Unterbrechungsprioritätsebenen zu definieren. Beispielsweise folgendermaßen:

```
    quiet[3] region (var) RMW(var);            /* disable IRQ levels 1..3 */
    quiet[6..9] region (var) RMW(var);         /* disable IRQ levels 6..9 */
```

Hier soll eine beliebige Lese-, Änderungs- und Schreiboperation (`RMW`) ungestört nur von Unterbrechungsanforderungen durchgeführt werden, die bestimmten Unterbrechungsprioritäten zugeordnet sind. Damit könnten Unterbrechungssperren für ↑Prozessoren mit mehreren Prioritätsebenen (8 bei ↑m68k, bis zu 256 bei ↑Cortex-M3) oder externen ↑Unterbrechungssteuereinheiten (↑PIC für ↑x86) bereichsweise zum Ausdruck gebracht werden.
Eine nicht ganz so grobe Methode, die nämlich Unterbrechungsanforderungen sehr wohl zulässt, dafür aber den möglichen Prozessorentzug als Folge einer ↑Umplanung verhindert, ließe sich wie folgt zum Ausdruck bringen:

```
    nonpreemptive region (var) RMW(var);                /* execute as NPCS */
```

Indem der Prozess, der sich gerade im kritischen Abschnitt (`RMW`) befindet, nicht vom Prozessor verdrängt wird, den Abschnitt also von Anfang bis Ende durchlaufen kann (↑*run to completion*), ist es auch keinem Prozess (desselben Prozessors) möglich, in dieser Zeitspanne

den Abschnitt für sich selbst beanspruchen zu können (↑NPCS).
Nicht nur für die ↑Systemprogrammierung wäre eine Einflussnahme auf das jeweils zu verwendende Synchronisationsverfahren des kritischen Abschnitts vorteilhaft. So ließe sich die Verwendung von Semaphore für Szenarien ausschließen, bei denen ↑blockierende Synchronisation durch wechselseitigen Ausschluss nicht praktikabel ist, weil es nahezu sicher zur ↑Verklemmung eines bestimmten Prozesses kann. Ein typisches Beispiel dafür wäre ein kritischer Abschnitt, der einerseits durch ↑Prozessinkarnationen und andererseits im Rahmen einer ↑Unterbrechungsbehandlung passiert werden soll — in beiden Fällen finden Prozesse statt, aber ein direkt (durch die Hardware) in eine Unterbrechungsbehandlung gezwungener Prozess sollte tunlichst nicht blockieren.

kritischer Abschnitt (en.) ↑*critical section*. Bezeichnung für einen bestimmten räumlichen Bereich (↑Text, ↑Daten) in einem ↑Programm, der eine ↑Wettlaufsituation ↑gleichzeitiger Prozesse möglich macht und dadurch eine in sich inkonsistente ↑Aktionsfolge hervorbringen kann. Dadurch können falsche Berechungsergebnisse, widersprüchliche ↑Daten oder gefährliche Kontrollflüsse (↑*lost wake-up*) hervorgerufen werden.
Der ursprünglichen Bedeutung (Dijkstra, 1965) nach darf ein solcher Abschnitt zu jedem Augenblick nur von höchstens einen Prozess besetzt sein (↑*mutual exclusion*). Gemeinhin ist jedoch jede Form von ↑Nebenläufigkeitssteuerung zulässig, solange damit zu jeder Zeit ein in sich konsistenter ↑Programmablauf in diesem Abschnitt sichergestellt ist. Eine Abstraktion davon stellt die ↑kritische Region dar.

Kurvenschreiber (en.) ↑*plotter*. ↑Peripheriegerät zur Ausgabe von Funktionsgraphen, technische Zeichnungen oder Grafiken; X-Y-Schreiber. Messgerät mit einer Schreibeinrichtung, um den zeitlichen Verlauf eingetragener Messgrößen festzuhalten. Die durch einen zugehörigen ↑Gerätetreiber zur Ausgabe nacheinander mittels ↑Ein-/Ausgaberegister bereitgestellten ↑Daten definieren die jeweilige Lage der Schreibeinrichtung in einer Ebene (Koordinate). Je nach Geräteart ist die Schreibeinrichtung ein Stift (Papier), Messer (Folie), Laserstrahl (Folie) oder Lichtkopf (Film). Der Gerätetreiber ist für gewöhnlich ein ↑Unterprogramm von einem Steuerprogramm zur Organisation und Überwachung des Rechnerbetriebs (↑*resident monitor*). Falls ↑abgesetzter Betrieb geführt wird, ist das Gerät selbst am ↑Satellitenrechner angeschlossen, ansonsten am ↑Hauptrechner.

kurzfristige Planung (en.) ↑*short-term scheduling*. ↑Ablaufplanung auf kurze Sicht, typischerweise im Mikro- oder Millisekundenbereich. Eine ↑Systemfunktion der Prozessverwaltung (*on-line* ↑*scheduling*). Festlegung der Reihenfolge, nach der die ↑Einlastung der CPU geschehen soll. Obligatorische Maßnahme zur ↑Simultanverarbeitung. VL 9.2

label (dt.) Kennzeichen, Marke, ↑Name. Bezeichnung für eine ↑symbolische Adresse.

Ladeadresse (en.) ↑*load address*. Bezeichnung für eine ↑Adresse, an der ein Element (↑Modul, ↑Unterprogramm, ↑Maschinenbefehl, ↑Exemplar eines ↑Datentyps) in dem für ein ↑Maschinenprogramm einzurichtenden (realen, logischen, virtuellen) ↑Adressraum zu platzieren ist. Eine solche Adresse wird zur ↑Bindezeit des Maschinenprogramms für jedes einzelne darin enthaltende ↑Objekt (↑Text, ↑Daten, ↑BSS) bestimmt, sie ist gegebenenfalls nachträglich zur ↑Ladezeit oder sogar zur ↑Laufzeit noch zu korrigieren (↑*program relocation*).

Ladedistanz (en.) ↑*load distance*. Der räumliche Abstand zwischen der alten und neuen ↑Ladeadresse eines ganzen ↑Maschinenprogramms oder auch nur einzelner Bestandteile davon. Je nach Richtung der ↑Verlagerung ein positiver oder negativer Korrekturwert.

Lademodul (en.) ↑*load module*. ↑Datei mit der Eingabe für einen ↑Lader, Ausgabedatei von einem ↑Binder.

Laden (en.) ↑*loading*. Vorgang, um ↑Text oder ↑Daten von einem ↑Datenträger in den ↑Arbeitsspeicher zu übertragen. Dabei ist ein ↑verschiebender Lader weitestgehend frei in der Wahl

der ↑Adresse im Arbeitsspeicher, an der das zu ladende ↑Maschinenprogramm platziert werden kann. Ein ↑bindender Lader hat zunächst die gleiche Wahl und sorgt aber zusätzlich noch dafür, jede eventuell noch bestehende ↑unaufgelöste Referenz zu beseitigen. Dazu wird die ↑Bindung mit der ↑Ladeadresse der referenzierten ↑Entität komplettiert, wozu letztere gegebenenfalls noch selbst in den Arbeitsspeicher zu bringen ist. Ohne solch eine Wahl ist der ↑Lader an die durch das ↑Lademodul sodann vorgegebene Ladeadresse gebunden. Passend zur ↑Betriebsart wird der ↑Adressraum für das Maschinenprogramm etabliert, eine ↑Prozessinkarnation eingerichtet, für die Zuteilung weiterer ↑Betriebsmittel gesorgt und der zugehörige ↑Prozess bereitgestellt (↑*scheduling*).

Lader (en.) ↑*loader*. Bezeichnung für eine ↑Systemfunktion, die das durch ein ↑Lademodul beschriebene ↑Maschinenprogramm in den ↑Arbeitsspeicher platziert und abschließend einen ↑Prozess damit verknüpft. Dieser Prozess existiert bereits und hat den ↑Systemaufruf zum Laden selbst abgesetzt (UNIX `exec`) oder er wird beim Ladevorgang erzeugt (VMS `run`, Windows `spawn`).

Ladestrategie (en.) ↑*fetch policy*. Verfahrensweise nach der das Moment des Zugriffs auf ein im ↑Umlagerungsbereich von einem ↑Prozess belegtes ↑Umlagerungsmittel bestimmt wird. Ein solcher Zugriff impliziert die Einlagerung des Umlagerungsmittels in den ↑Hauptspeicher. Diese Einlagerung geschieht bei Bedarf (*on demand*) oder im Voraus (*anticipatory*), das heißt, entweder bei oder vor dem Zugriff auf das ausgelagerte Umlagerungsmittel. Im ersten Fall kommt zur Ausführung von einem ↑Maschinenbefehl durch die ↑CPU eine ↑virtuelle Adresse zur Geltung, die zwar gültig ist, jedoch nicht auf ↑Text oder ↑Daten im Hauptspeicher abgebildet werden kann. Es kommt zu einen ↑Hauptspeicherfehlzugriff. Bildet ↑virtueller Speicher den Bezug, betrifft der Fehlzugriff in aller Regel eine ↑Seite und es kommt zur ↑Seitenumlagerung durch das ↑Betriebssystem (↑*pager*). Der Fehlzugriff kann sich jedoch auch auf ein ↑Text- oder ↑Datensegment beziehen, das noch nicht in den ↑Prozessadressraum eingebunden wurde (↑*dynamic binding*). In dem Fall wird ein ↑dynamischer Binder im Betriebssystem das referenzierte ↑Segment komplett (↑*segmentation*) oder teilweise (↑*segmented paging*) in den Hauptspeicher bringen und entsprechend im Umlagerungsbereich verbuchen. Erstere Variante ist gleichsam ein Beispiel für die Einlagerung im Voraus, im Falle seitennummerierter Segmentierung kann dies insbesondere den ↑Seitenvorabruf für alle Seiten des Segments auslösen. Ist die ↑Arbeitsmenge des den Fehlzugriff auslösenden Prozesses bekannt, werden wenigstens alle zu dieser Menge zählenden Seiten eingelagert. Damit die Einlagerung vollzogen werden kann, bestimmt die ↑Platzierungsstrategie den dafür benötigten ↑Speicherbereich im Hauptspeicher. Ist nicht genügend freier Hauptspeicher verfügbar, kommt entweder die ↑Ersetzungsstrategie ins Spiel oder der Prozess wird vorläufig durch den ↑Planer suspendiert (↑Prozesswechsel), bis ausreichend Hauptspeicherplatz zur Verfügung steht.

Ladezeit (en.) ↑*loading time*. Zeitpunkt zu dem ein ↑Programm zur Ausführung in den ↑Arbeitsspeicher geladen wird oder Zeitspanne (↑*load time*) eben dieses Ladevorgangs.

langfristige Planung (en.) ↑*long-term scheduling*. ↑Ablaufplanung auf lange Sicht, typischerweise im Sekunden- oder Minutenbereich. Eine ↑Systemfunktion der Lastkontrolle im ↑Rechensystem, steuert den Grad an ↑Mehrprogrammbetrieb und bestimmt dazu den Zeitpunkt der Zulassung von einem ↑Prozess und damit den Moment seiner Teilnahme am Rechenbetrieb. Für gewöhnlich greift diese Form von ↑Einplanung bei der ↑Anmeldung, zur ↑Umlagerung kompletter ↑Programme (↑*swapping*) oder zum Zwecke der ↑Verklemmungsvermeidung.

Lastausgleich (en.) ↑*load balance*. Ergebnis einer ↑Lastverteilung, setzt für gewöhnlich ↑Parallelverarbeitung voraus.

Lastverteilung (en.) ↑*load balancing*. Art und Weise, in der ↑Aufgaben auf mehr als einen ↑Rechenkern von einer ↑CPU oder einem ↑Multiprozessor verteilt werden. Ein Vorgang, dessen Ziel es ist, ↑Lastausgleich herbeizuführen.

latch (dt.) Auffangregister; zustandsgesteuertes FlipFlop (1-Bit ↑Speicher).

latency (dt.) ↑Latenz.

latency period (dt.) ↑Latenzzeit.

Latenz (en.) ↑*latency*. Gemeinhin die Verzögerungszeit (↑*latency period*) zwischen einem ↑Ereignis als Ursache und der darauf folgenden ↑Aktion als Wirkung (↑*causality*).
Vorhandensein einer Sache, die noch nicht in Erscheinung getreten ist (Duden). Im Kontext von einem ↑Betriebssystem beispielsweise die Sache, dass eine ↑Unterbrechungsbehandlung erfolgen wird, da die ↑CPU eine ↑Unterbrechungsanforderung erkannt hat, diese Behandlung wegen einer (implizit durch die CPU oder explizit durch das Betriebssystem) gesetzten ↑Unterbrechungssperre allerdings noch nicht möglich ist. Analog dazu die Sache, dass ein ↑Prozesswechsel geschehen wird, da der ↑Planer den Vorrang für einen ausgelösten ↑Prozess festgestellt hat, die dafür notwendige ↑Einlastung der CPU wegen einer (bedingt durch das ↑Operationsprinzip des Betriebssystems, implizit oder explizit) gesetzten ↑Verdrängungssperre jedoch unterbunden ist.
Allgemein jedes Ereignis, das (gemäß ↑Programm) als logische Folge eines vorausgegangenen Ereignisses absehbar ist, aber auf Grund eines bestimmten Systemzustands noch nicht sofort herbeigeführt werden darf. Die Zeit zwischen dem ursächlichen und dem wirkenden Ereignis ist die ↑Latenzzeit.

Latenzzeit (en.) ↑*latency period*, auch kurz (en.) ↑*latency*. Bezeichnung für die Zeitspanne einer technisch bedingten Verzögerung.

Laufzeit (en.) ↑*run-time*. Zeitpunkt zu dem ein ↑Prozess stattfindet beziehungsweise Zeitspanne einer ↑Aktion, ↑Aufgabe oder eines ↑Programmablaufs.

Laufzeitkontext (en.) ↑*run-time context*. Gesamtheit an Zustandsdaten der (realen/virtuellen) Maschine, die einen ↑Prozess definieren; der ↑Kontext eines ↑Programmablaufs. Neben dem ↑Laufzeitstapel ist ein weiteres wesentliches Bestandteil dieser Daten der ↑Prozessorstatus. Kommt es zur Unterbrechung eines Prozesses, spiegelt der in dem Moment gültige Prozessorstatus den Kontext zur späteren Fortsetzung eben dieses Prozesses wider. Um den Prozess fortsetzen zu können, als wenn seine Unterbrechung nie geschehen wäre, ist der Prozessorstatus invariant zu halten. Dies wird erreicht durch eine zeitweilige ↑Zustandssicherung in eine dem Prozess eigene Datenstruktur im ↑Hauptspeicher: Der Prozessorstatus wird bei der Unterbrechung in diese Datenstruktur gesichert und zur Fortsetzung wieder daraus geladen.

Laufzeitstapel (en.) ↑*run-time stack*. ↑Stapelspeicher von einem ↑Handlungsstrang; dient vornehmlich der zeitweiligen Lagerung lokaler Daten.

Laufzeitsystem (en.) ↑*run-time system*. Menge von Funktionen, die in Abhängigkeit von der jeweils verwendeten Programmiersprache eine Ablaufunterstützung für ein ↑Maschinenprogramm bildet. Typische Beispiele solcher Funktionen für ↑C sind formatierte Ein-/Ausgabe (`scanf(3)`, `printf(3)`), kopieren von Speicherbereichen (`memcpy(3)`, `strcpy(3)`), Verarbeitung von Zeichenketten (`string(3)`), Ein-/Ausgabe von Binärdatenströmen (`fread(3)`), Handhabung von Signalen (`signal(3)`), Kontrollfluss- und Kontextverwaltung (`setjmp(3)`), Verwaltung von ↑Haldenspeicher (`malloc(3)`) und mehr. Technisch sind die Funktionen jeweils als ↑Unterprogramm aus- und in einer ↑Programmbibliothek abgelegt (u.a. ↑libc). Nicht wenige dieser Funktionen dienen insbesondere der Interaktion mit dem ↑Betriebssystem und versuchen gerade in dem Zusammenhang, die durch einen ↑Systemaufruf bedingte ↑Latenzzeit zu kaschieren.

LCFS Abkürzung für (en.) *last come, first served*, (dt.) wer zuletzt kommt, mahlt zuerst; Reihungsverfahren, gleichbedeutend mit ↑LIFO.

lead time (dt.) ↑Vorlaufzeit.

leaf procedure (dt.) ↑Blattprozedur.

Leerbefehl (en.) ↑*no-op instruction*. Bezeichnung für einen wirkungslosen ↑Maschinenbefehl, der keinen Effekt hat, außer Platz im ↑Hauptspeicher zu belegen, ↑Ausführungszeit zu benötigen und Energie zu beanspruchen.

Exkurs Typische Bezeichnung für einen solchen Befehl ist `NOP` (*no operation*). Mit Ausnahme des ↑PC bleibt bei Ausführung dieses Befehls der im ↑Programmiermodell definierte Zustand einer ↑CPU unverändert. Im Falle von ↑x86 hat `NOP` den ↑Operationskode `0x90`, der dem Maschinenbefehl

```
xchg %eax, %eax
```

entspricht und bedeutet, den Inhalt von ↑Register `eax` mit seinem eigenen Inhalt auszutauschen — also trotz geleisteter Arbeit unverändert zu belassen. Dieses Beispiel verdeutlicht, dass mit `NOP` schon noch eine bestimmte Operation verbunden sein kann. Allerdings hat diese Operation eben keinen Effekt auf den Zustand der CPU, den PC ausgenommen.

Leerlauf (en.) ↑*idle*. Zustand der Untätigkeit, nämlich wenn auf einem ↑Prozessor kein ↑Prozess stattfindet. Entweder wird der Prozessor gerade eben durch einen auf einem anderen Prozessor stattfindenden Prozess hochgefahren, durchläuft seine Initialisierungsprozedur, und hat vom ↑Planer noch keinen Prozess zugewiesen bekommen (↑Multiprozessor) oder im Moment der Blockierung des auf ihm stattfindenden Prozesses steht kein anderer Prozess zur ↑Einlastung mehr bereit.
Der Prozessor kann nur noch durch einen externen Prozess aus diesen Zustand befreit werden, indem ihm „von außen“ ein Prozess zur Einlastung bereitgestellt wird. Dies kann durch eine ↑Unterbrechungsanforderung geschehen, die die Deblockierung eines blockierten Prozesses bewirkt oder, vorausgesetzt der Prozessor ist aktiv untätig, indem er auf der ↑Bereitliste plötzlich einen Prozess vorfindet, den der Planer von einem anderen Prozessor aus dort abgelegt hat. Im letzteren Fall ist ein ↑Leerlaufprozess aktiv wartend darauf, dass die Bereitliste wieder gefüllt wird (↑*busy waiting*). Normalerweise jedoch wird ein Prozessor in solch einer Situation in den ↑Schlafzustand versetzt, um nicht unnötig das ↑Rechensystem zu strapazieren, Energie zu verbrauchen, Abwärme zu produzieren — damit Unkosten zu verursachen und die Umwelt zu belasten.

Exkurs Auf den ersten Blick erscheint der Vorgang, einen Prozessor in den Schlafzustand zu versetzen, keine besonders komplizierte ↑Aktion zu sein. Der Übergang in diesen Zustand ist eine *bedingte Anweisung*, die den Schlafzustand nur aktiviert, wenn die Bedingung dafür noch gilt. Gegebenenfalls kann diese *Schlafbedingung* im Moment des Schlafengehens aber bereits entkräftet worden sein, ohne dass der diesen Übergang steuernde Prozess etwas davon mitbekommen haben muss (↑*lost wake-up*). Gleichzeitige Prozesse, die Auswirkungen auf die Aktivierung des Schlafzustands des Prozessors haben können, sind daher unbedingt zu koordinieren:

```
void relax(const void *sign) {     /* retire CPU, prepare for sleep state */
    arena(ENTER);                                     /* join danger zone */
    if (*(long *)sign == 0)                          /* still out of work? */
        drift();                                  /* yes, let CPU drift off */
    arena(LEAVE);                                     /* quit danger zone */
}
```

Die hier gezeigte Klammerung der dem kritischen Pfad entsprechenden „Kampfbahn“ (`arena`) beugt der Gefahr vor, die ↑CPU möglicherweise in den ewigen Schlaf zu treiben (`drift`). Den Schlafzustand bewirkt typischerweise ein spezieller ↑Maschinenbefehl (z.B. `hlt`, ↑x86), dessen Ausführung die CPU nämlich erst mit der nächsten Unterbrechungsanforderung beendet. Mit Ausführungsbeginn dieses Befehls hebt die CPU zwar implizit eine eventuell bestehende ↑Unterbrechungssperre auf, allerdings ist damit noch längst nicht garantiert, dass die ↑Peripherie auch eine weitere Unterbrechungsanforderung zur CPU sendet. Sollte der CPU also

eine solche Anforderung entgangen sein, schläft sie möglicherweise für immer — es sei denn, das ↑Betriebssystem nutzt einen periodischen ↑Zeitgeber, der jedoch nicht für alle ↑Betriebsarten nötig und somit alles andere als Standard ist.
Gängige Lösung zur Absicherung dieses Pfads ist es, beim Betreten (`ENTER`) eine *totale Unterbrechungssperre* zu setzen und diese beim Verlassen (`LEAVE`) wieder aufzuheben. Die entsprechenden Anweisungen dazu zeigt nachfolgendes Beispiel:

```
inline void arena(plan_t plan) {                    /* bracketing construct */
    if (plan == ENTER)                                    /* start of path? */
        asm volatile ("cli": : : "cc");                 /* yes, disable IRQ */
    else if (plan == LEAVE)                                /* end of path? */
        asm volatile ("sti": : : "cc");                  /* yes, enable IRQ */
}
inline void drift() {                     /* suspend CPU, force sleep state */
    asm volatile ("hlt");
}
```

Da der Haltebefehl (in `drift`: `hlt`) trotz Sperre eine Unterbrechungsanforderung zulässt und bei korrekter Klammerung (s.o., `relax`) bis dahin der Prozess nicht unterbrochen werden kann, ist softwareseitig dafür Sorge getragen, den erwarteten Weckruf nicht zu verpassen. Dies setzt allerdings voraus, dass bei ↑Flankensteuerung die Hardware wenigstens eine solche zwischenzeitlich eingehende Anforderung zwischenspeichert.
Alternativ ist ein *sperrfreies Verfahren* möglich, das beim Betreten des Pfads einen *Rücksetzpunkt* definiert und diesen beim Verlassen des Pfads wieder ungültig macht. Ist ein solcher Rücksetzpunkt im Moment einer dann jederzeit möglichen Unterbrechungsanforderung gültig und kommt es im Verlauf der ↑Unterbrechungsbehandlung zur Bereitstellung eines Prozesses, setzt der unterbrochene Leerlaufprozess bei Rückkehr aus dem ↑Unterbrechungshandhaber genau an dieser Stelle wieder auf, nicht etwa an seiner Unterbrechungsstelle. Als Folge dieses Umstands wertet der Leerlaufprozess die Wartebedingung erneut aus, stellt fest (`*sign` $\neq$ `0`), die CPU nicht in den Schlafzustand versetzen zu müssen und verlässt seinen Ruheplatz (`relax`). Dieser Ansatz bedeutet letztlich, beim Betreten des Pfads eine ↑Koroutine einzurichten und eine dazu passende ↑Handhabe bekanntzugeben, um darüber die Unterbrechungsbehandlung mit einem ↑Koroutinenwechsel abschließen zu können:

```
inline void arena(plan_t plan) {                    /* bracketing construct */
    extern coroutine_t *idle;                           /* coroutine handle */
    if (plan == ENTER)                                    /* start of path? */
        asm volatile (                          /* yes, define restart point */
            "pushl $1f\n\t"                        /* setup point of restart */
            "movl %%esp, %0\n"                     /* setup coroutine handle */
            "1:"                               /* come here upon resume(idle) */
            : "=m" (idle));
    else if (plan == LEAVE)                                /* end of path? */
        if (FAS(&idle, 0))                /* yes, cancel restart; interrupted?*/
            asm volatile ("addl $4, %esp");    /* no, remove restart point */
}
```

Beim Verlassen des Pfads muss ein ungültiger Rücksetzpunkt hinterlassen werden, um nicht fälschlicherweise noch bei der nächsten Unterbrechungsanforderung einen Koroutinenwechsel hin zum Leerlaufprozess abzusetzen. Wurde der Leerlaufprozess durch einen Koroutinenwechsel fort- beziehungsweise zurückgesetzt, ist die Koroutinenhandhabe bereits ungültig (`idle` = `0`) und es ist keine „Aufräumarbeit“ zu leisten. Anderenfalls (`idle` $\neq$ `0`) wurde weder der Leerlaufprozess unterbrochen noch die CPU in den Schlafzustand versetzt, so dass der noch bestehende Hinweis auf den Rücksetzpunkt entfernt werden muss. Um einer möglichen ↑Wettlaufsituation vorzubeugen, ist das Lesen und Löschen des Wertes der Koroutinenhandhabe als ↑atomare Operation durchzuführen (↑FAS).

Das Beispiel geht davon aus, dass dem Koroutinenwechsel eine *komplexe Koroutine* zugrunde liegt. Stößt die Unterbrechungsbehandlung auf den Sonderfall, zum Leerlaufprozess zurückzusetzen (nämlich wenn gilt: `idle` $\neq$ `0`), ist als letzte Anweisung lediglich der Funktionsaufruf `resume(idle)` auszuführen. Da jedoch der Unterbrechungshandhaber auch in dem Fall immer nur verlassen und nicht als Koroutine unterbrochen wird, ist nur für die Fortsetzung der Koroutine des Leerlaufprozesses zu sorgen. Folgendes Beispiel skizziert die dazu erforderlichen letzten Maschinenbefehle einer Unterbrechungsbehandlung:

```
    ...                 # at this point, the processor status was restored
    cmpl $0, idle       # idle process interrupted?
    je 1f               # no, perform normal exit
    movl idle, %esp     # yes, switch stack to idle-process coroutine
    movl $0, idle       # avoid spurious resume: cancel restart point
    ret                 # resume execution of idle-process coroutine
1:                      # come here for normal exit
    iret                # return from interrupt
```

Für gewöhnlich läuft diese Befehlsfolge in demselben Modus, den die CPU mit Aufnahme der Unterbrechungsbehandlung inne hatte: maskierbare ↑Unterbrechungsanforderungen (↑IRQ) sind maskiert. Damit bildet diese ↑Aktionsfolge eine ↑unteilbare Anweisung auf dieser CPU. Des Weiteren ist zu beachten, dass all diese Maßnahmen dann als Teil einer Phase stattfinden, in der die CPU logisch untätig ist (↑*idle process*, S. 160) — somit die im Vergleich zum sperrenden Ansatz etwas höheren ↑Gemeinkosten kaum oder nicht ins Gewicht fallen und damit die erreichte Sperrfreiheit nicht zwingend zur Erhöhung der ↑Betriebslast beiträgt.

Leerlaufprozess (en.) ↑*idle process*. Bezeichnung für einen ↑Prozess, der den ↑Leerlauf von einem ↑Prozessor kontrolliert; der zwar logisch untätig ist, aber physisch für das System tätig sein kann. Es gibt zwei grundsätzlich verschiedene Modelle für einen solchen Prozess.
In dem einen Fall steht für den Leerlauf eine eigene ↑Prozessinkarnation zur Verfügung. Entsprechend wird der Leerlauf sodann durch einen physischen ↑Prozesswechsel gestartet, und zwar immer dann, wenn im Moment der Blockierung eines Prozesses die ↑Bereitliste keinen anderen Prozess für den Prozessor enthält. In dem Modell zieht der Leerlauf genau genommen immer zwei physische Prozesswechsel nach sich: erstens vom blockierenden zum leerlaufenden Prozess und zweitens vom leerlaufenden zum bereitgestellten Prozess. Dies geschieht auch dann, wenn der bereitgestellte Prozess sich als derjenige erweist, der vorher zum leerlaufenden Prozess gewechselt ist.
In dem anderen Fall übernimmt immer jener Prozess die Rolle der Leerlaufkontrolle, der jüngst zurückliegend gerade blockiert ist und in dem Moment keinen anderen für seinen Prozessor lauffähigen Prozess in der Bereitliste vorfindet. Damit kann ein beliebiger Prozess in diese Rolle gebracht werden, die Identität des leerlaufenden Prozesses ist nicht eindeutig bestimmt. Jedoch eröffnet sich so die Option, dem physischen Prozesswechsel vorzubeugen, wenn nämlich der Prozess in seiner Leerlaufphase deblockiert wird. Zu beachten ist, dass lediglich ein logischer Prozesswechsel vollzogen wird, nämlich vom ↑Prozesszustand <u>laufend</u> nach <u>untätig</u>: ein physischer Wechsel der Prozessinkarnation erfolgt nicht.
Für gewöhnlich versetzt der den Leerlauf kontrollierende Prozess seinen Prozessor in einen speziellen Bereitschaftsbetrieb (*stand-by mode*), der zur Absenkung des Energiebedarfs der Hardware beiträgt. Hierzu muss der Prozessor allerdings über einen speziellen, privilegierten ↑Maschinenbefehl (z.B. `hlt`, ↑x86) verfügen, der dann vom Betriebssystem zu gegebener Zeit zur Ausführung zu bringen ist. Mit Ausführung dieses Befehls geht faktisch der Stillstand (*stall*) des Prozesses einher, indem nämlich der Leerlauf dann im Prozessor selbst, also in der Hardware, stattfindet. Der Prozess, der diesen Befehl zur Ausführung bringt, wird erst wieder aufgenommen, wenn die Befehlsausführung zum Abschluss gekommen ist. Der Befehlsabschluss geschieht in diesem Fall gemeinhin durch eine ↑Unterbrechungsanforderung. Die dadurch ausgelöste ↑Unterbrechungsbehandlung kann im Betriebssystem sodann ein ↑Ereignis hervorrufen, das die Beendigung des Leerlaufs anzeigt: beispielsweise wurde als Folge

der Unterbrechungsbehandlung ein Prozess auf die Bereitliste gesetzt. Nimmt der Prozess sodann seine Tätigkeit mit Abschluss des Spezialbefehls wieder auf, muss er prüfen, ob das Ereignis zur Beendigung des Leerlaufs wirklich vorliegt. Falls dieses Ereignis noch nicht eingetreten ist, bringt der Prozess den Spezialbefehl abermals zur Ausführung und den (eben erst aufgeweckten) Prozessor wieder in den Bereitsschaftsbetrieb.
Diese ↑Leerlaufschleife definiert die Art der Untätigkeit des Betriebssystems beziehungsweise des Prozessors, indem der Schleifenrumpf den Prozessor gegebenenfalls in Bereitschaft setzt und ansonsten beliebige ↑Aufgaben im Betriebssystem erledigt. So wurde etwa in manchen (frühen) Versionen von ↑UNIX in solch einem Schleifenrumpf unter anderem eine millionenstellige Approximation für die Eulersche Zahl e berechnet, da das Betriebssystem mangels laufbereiter Prozesse dem Prozessor sonst keine Arbeit zuweisen konnte. Anstatt bei ausstehenden (laufbereiten) Prozessen derartige Berechnungen durchzuführen, weil sich der Prozessor vielleicht nicht in Bereitschaft setzen lässt, kann dieser Schleifenrumpf allerdings auch zur Überprüfung des Funktionszustands des Betriebssystems genutzt werden, der Suche nach ↑Verklemmungen gewidmet sein oder anderen Handlungen zur Validierung dienen.

Exkurs Die Maßnahmen für den Leerlauf spielen sich letztlich auf zwei verschiedenen Ebenen ab, der prozessbezogenen Ebene einerseits und der prozessorbezogenen Ebene andererseits. Ein Beispiel für die prozessbezogene Ebene, mit einer für diesen Fall speerfreien Leerlaufschleife (vgl. S. 161) und einem potentiell sperrfreien Schleifenrumpf (`relax`, S. 157), ist nachfolgend skizziert:

```
void stall(const void *sign) {              /* process reduced to inaction */
    process_t *self = being(ONESELF);            /* own process descriptor */

    state(&self->mood, IDLE);                          /* enter idle state */

    while (*(const long *)sign == 0)            /* idle loop: out of work? */
        relax(sign);                         /* yes, be sleep at the switch */

    state(&self->mood, CLEAR|IDLE);                    /* leave idle state */
}
```

Die gezeigte Operation (`stall`) bringt den gegenwärtig stattfindenden Prozess (`ONESELF`) dazu, den Leerlauf (`IDLE`) zu kontrollieren, effektiv stillzustehen, und belässt ihn in dieser Rolle, solange dies geboten ist. Für die Abbruchbedingung der Leerlaufschleife verweist der hier applizierte ↑Zeiger (`sign`) typischerweise auf eine ↑Speicherstelle, die einen Hinweis auf den „Füllstand“ der Bereitliste gibt. Diese Stelle ist entweder ein Zeiger auf das Kopfelement der Bereitliste oder ein Zähler der zur Zeit insgesamt bereitstehenden Prozesse: der Wert null zeigt in beiden Fällen an, dass kein anderer Prozess zur Beschäftigung des Prozessors verfügbar ist. In dieser Rolle instruiert der „stillstehende“ Prozess seinen Prozessor, sich auszuruhen (`relax`, vgl. S. 157). Damit wechselt der Prozess auf die prozessorbezogene Ebene, die den eigentlichen Leerlauf herbeiführt: die ↑CPU in den ↑Schlafzustand versetzen. Diese ↑Aktion ist eine *bedingte Anweisung*, die nur ausgeführt werden darf, wenn die Schlafbedingung noch gilt, das heißt, falls zwischenzeitlich kein Prozess auf die Bereitliste gesetzt wurde. Um die Schlafbedingung überprüfen zu können, wird der prozessorbezogenen Ebene die Adresse der Stelle auf der Bereitliste mitgeteilt, wo der nächste Schub (`batch`) von Prozessen zu erwarten ist. Nimmt der kontrollierende Prozess eine mittlerweile wieder aufgefüllte Bereitlste war, beendet er seine Phase „scheinbarer Untätigkeit“.

Leerlaufschleife (en.) ↑*idle loop*. Bezeichnung für eine ↑Kontrollstruktur im ↑Betriebssystem, durch die wiederholt der ↑Leerlauf herbeigeführt wird, nämlich solange die Schleifenbedingung gültig bleibt (Laufbedingung) beziehungsweise bis sie aufgehoben ist (Abbruchbedingung). Die Schleifenbedingung nimmt Bezug auf ein bestimmtes ↑Ereignis, das die Beendigung des Leerlaufs anzeigt.

Exkurs Eine sehr einfache Variante, die den ↑Prozessor (↑x86) bei gültiger Schleifenbedingung wiederholt in Betriebsbereitschaft (*stand-by mode*) und damit den ↑Leerlaufprozess

in den Stillstand (*stall*) versetzt, zeigt folgendes Beispiel (vgl. auch S. 157):

```
void stall(const void *sign) {            /* if need be, yielding the CPU */
    asm volatile("cli");                              /* disable interrupts */

    while (*(const long *)sign == 0)         /* check idle-loop condition */
        asm volatile("hlt");     /* halt CPU until next interrupt request */

    asm volatile("sti");                        /* (re-)enable interrupts */
}
```

Besonders zu beachten ist die durch den ↑Maschinenbefehl `cli` erhobene ↑Unterbrechungssperre. Diese beugt der ↑Wettlaufsituation vor, nämlich dass der Prozessor trotz mittlerweile aufgehobener Schleifenbedingung seine Tätigkeit einstellt, anhält (`hlt`) und gegebenenfalls seine Betriebsbereitschaft niemals beendet (↑*lost wake-up*). Entsprechend wird die Unterbrechungssperre wieder aufgehoben (`sti`), wenn der Leerlauf als beendet gilt. Während der Prozessor hält (`hlt`) ist die Unterbrechungssperre jedoch implizit aufgehoben, damit nach Eingang einer ↑Unterbrechungsanforderung im Zuge der ↑Unterbrechungsbehandlung gegebenenfalls die Abbruchbedingung der Schleife herbeigeführt werden kann.

leichtgewichtiger Prozess (en.) ↑*light-weight process*. Bezeichnung für einen ↑Prozess, der als ↑Systemkernfaden mit anderen Prozessen seiner Art zusammen im gemeinsamen und durch ↑Speicherschutz isolierten ↑Adressraum stattfindet.

Leitweglenkung (en.) ↑*routing*. Vorrichtung (↑Systemfunktion) zum Festlegen der Route (Leitweg) für ein ↑Signal oder eine ↑Nachricht von seiner Quelle zum Ziel. Ursprünglich eine Einrichtung in der Telekommunikation, um die Wegewahl für Nachrichtenströmen bei der Übertragung in einem vermaschten ↑Netzwerk zu regeln. Im Falle von ↑Mehrkernprozessoren führt jedoch auch ein ↑PIC im Prinzip eine solche Wegewahl in Bezug auf eine ↑Unterbrechungsanforderung durch. Die Verfahren zur Wegewahl unterscheiden sich allerdings sehr stark vom Einsatzbereich einer solchen Vorrichtung.

Leitwerk siehe ↑Steuerwerk.

Lese-/Schreibsperre (dt.) ↑*read/write lock*. Bezeichnung für eine ↑Umlaufsperre mit der ↑wechselseitiger Ausschluss durch einfache, *elementare Lese-/Schreibzugriffe* auf einzelne ↑Speicherworte erreicht wird. Verfahren dieser Art benötigen für gewöhnlich ↑Vorwissen über die maximale Anzahl an dieser Sperre ↑gekoppelter Prozesse, hängen allerdings nicht davon ab, dass ein ↑atomarer Befehl zur Verwaltung der Sperre erforderlich ist.
Trotz der Einschränkung auf die nach oben begrenzte Anzahl von Prozessen, für die mit dieser Art von Verfahren ↑Synchronisation erzielt werden kann, sind die nur mit primitiven Lese-/Schreiboperationen auskommenden Lösungen attraktiv. Zum einen erfordern sie keine speziellen, komplexen ↑Maschinenbefehle mit unteilbarem ↑Befehlszyklus zum Lesen, Modifizieren und Schreiben (↑*read-modify-write*) von ↑Daten. Dies trägt allgemein zur Portabilität bei, es macht die Lösungen einsetzbar sowohl für ↑CISC- als auch ↑RISC-Maschinen. Zum anderen sind diese Lösungen (bei kleiner Maximalanzahl gekoppelter Prozesse) vergleichsweise performant, wenn nämlich ein freier ↑kritischer Abschnitt durch den Prozess betreten wird, dass heißt, keine ↑Konkurrenzsituation vorliegt.

Exkurs Nachfolgend werden drei klassische Varianten einer solchen Sperre vorgestellt, die die Algorithmen von Dekker, Peterson und Kessels implementieren. Jede dieser Lösungen beugt dem ↑Verhungern gekoppelter Prozesse vor, im Gegensatz zu den ursprünglich von Dijkstra zur Darstellung des Synchronisationsproblems bei wechselseitigem Ausschluss diskutierten Ansätzen (vgl. S. 105). In allen Fällen wird von maximal zwei in der Konkurrenzsituation gekoppelten Prozesse ausgegangen (s. Definition von `NCPU`, S. 105).
Die Verfahren von Dekker und Peterson verwenden denselben ↑Datentypen, um das ↑Exemplar einer Sperre zu modellieren. Dieser erweitert den für die Lösung von Dijkstra benötigten Datentypen (S. 105) um eine Mitgliedsvariable (`turn`), die anzeigt, welcher Prozess in der

Konkurrenzsituation als nächster bevorzugt die Sperre halten soll:

```
typedef volatile struct lock {                    /* read/write memory lock */
    bool join[NCPU];                                   /* reservation flags */
    unsigned turn;                                /* preferenced procecessor */
} lock_t;
```

Initial verbucht ein diesbezügliches Exemplar für gewöhnlich keine Reservierung eines Prozesses, die Sperre für sich erheben und halten zu wollen. Allerdings muss für die Lösung von Dekker einem der beiden Prozessoren der Vortritt eingeräumt werden:

```
lock_t lock = { {false, false}, 0 };   /* defer to processor 0, initially */
```

Genau genommen wird der Prozess, der als nächster in der Konkurrenzsituation den Vorzug erhalten soll, durch die ↑Prozessoridentifikation des Prozessors bestimmt, auf dem dieser Prozess stattfindet. Dies aber impliziert, dass ein Prozess beim Versuch, die Sperre für sich zu erheben, keine ↑Verdrängung von seinem Prozessor durch einen anderen mit ihm örtlich gemeinsam stattfindenden Prozess erfahren darf. Anderenfalls besteht die Gefahr, dass mehrere an derselben Sperre gekoppelten Prozesse desselben Prozessors diese Sperre jeweils für sich beanspruchen und auch erheben können. Zur korrekten Verwendung aller nachfolgend behandelten Lösungen ist also zusätzlich noch eine ↑Verdrängungssperre zu erheben, aber nur, wenn die beteiligten Prozessoren in ↑Simultanverarbeitung betrieben werden.
Diese Vorgehensweise ist auch geboten, nicht aber gefordert, für Umlaufsperren, die komplexe atomare Maschinenbefehle verwenden. Der Grund dort sind beträchtliche Leistungseinbußen, die entstehen können, sollte der eine Sperre haltende Prozess vom Prozessor verdrängt werden (↑*lock holder preemption*).
Der Algorithmus von Dekker orientiert sich an der Vorlage von Dijkstra. Hebt ein Prozess die Sperre auf, muss er jedoch dem anderen Prozessor, der gegebenenfalls einen konkurrierenden Prozess beherbergt, der Vortritt geben. Anschließend nimmt er die Reservierung für seine Sperre zurück:

```
void unlock(lock_t *lock) {                       /* release read/write lock */
    unsigned self = aegis();                        /* read own processor id */

    lock->turn = self^1;                        /* defer to competing process */
    lock->join[self] = false;                       /* unban competing process */
}
```

Der Unterschied zur Vorlage von Dijkstra (S. 106) ist lediglich die zusätzliche Anweisung, dem jeweils anderen Prozessor zur Auflösung einer möglichen nächsten Konkurrenzsituation Vorzug einzuräumen.
Das Verfahren, um eine Sperre zu erheben und damit den jeweils anderen Prozess auszusperren (`lockup`), gestaltet sich nach gleichem Beginn im weiteren Verlauf jedoch etwas komplizierter (vgl. S. 106). Zunächst macht P_0, ein Prozess auf Prozessor 0, seinen Wunsch deutlich, die Sperre für sich zu beanspruchen. Anschließend prüft er, ob die Sperre bereits von P_1, einem Prozess auf Prozessor 1, gehalten wird. Ist dies der Fall, nimmt P_0 seine Reservierung zurück. Im weiteren Verlauf schlägt die Lösung von Dekker aber einen etwas anderen Weg ein, wie nachfolgend skizziert:

```
void lockup(lock_t *lock) {                       /* acquire read/write lock */
    unsigned self = aegis();                        /* read own processor id */

    lock->join[self] = true;                            /* make a reservation */
    while (lock->join[self^1])                    /* peer holds reservation? */
        if (lock->turn != self) {                 /* other processor's turn? */
            lock->join[self] = false;             /* yes, cancel reservation */
            while (lock->turn != self);            /* it's not my turn, spin */
            lock->join[self] = true;                /* re-assert reservation */
        }
}
```

Anstatt wie bei der Vorlage von Dijkstra nur darauf zu warten, dass P_1 seine Reservierung zurücknimmt, wartet P_0 bei der Lösung von Dekker in dieser Konkurrenzsituation zunächst ab, bis er (bzw. sein Prozessor) an der Reihe ist: P_0, nachdem er seine Reservierung storniert hat, betreibt ↑aktives Warten (innere `while`-Schleife), solange sein Prozessor noch nicht aufgerufen wurde. Hat P_1 die Sperre aufgehoben, verlässt P_0 seine Wartestellung, um seine Reservierung zu erneuern. Anschließend muss P_0 jedoch nochmals prüfen, ob die Sperre zwischenzeitlich wieder von P_1 angefordert wurde (äußere `while`-Schleife). Diese erneute Überprüfung ist erforderlich, da P_1 (bzw. sein Prozessor), obgleich er die Sperre gerade erst aufgehoben hat, schneller sein kann als P_0 (bzw. dessen Prozessor), der eben noch aktiv wartend tätig war: P_1 könnte P_0 überholen, das heißt, abermals die Sperre (durch ein für ihn erfolgreiches `lockup`) erheben.
Die Überholung von P_0 durch P_1 und umgekehrt kann an jeder Stelle in dem Programmbeispiel (`lockup`) geschehen. Heikel ist die Stelle, nachdem P_0 die innere Warteschleife verlassen hat und bevor er seine Reservierung erneuern konnte: genau an dieser Stelle wird P_0 nun verzögert und P_1 versucht erneut (durch `lockup`), die Sperre zu erheben. In der Situation hat P_0 zwar Vortritt, sonst hätte er die innere Schleife nicht verlassen können, allerdings würde P_1, wenn er P_0 überholt, in der Auswertung der Bedingung der äußeren Schleife feststellen, dass für die Sperre eben keine Reservierung durch P_0 vorliegt. In dieser Konstellation hat P_1 aber selbst die Sperre für sich schon wieder reserviert. Folglich schreitet P_1 mit erhobener Sperre voran. Nimmt P_0 seine Tätigkeit erst wieder auf, nachdem P_1 die Bedingung der äußeren Schleife hat auswerten können, kann er selbst, trotz Vortritt, die Sperre nicht mehr für sich erheben und damit P_1 aussperren. Stattdessen wartet P_0 in der äußeren Schleife das Aufheben der Sperre durch P_1 ab.
Dieser Überholungsvorgang kann jedoch nur einmal geschehen, sofern P_0 es schafft, nach Verlassen der inneren Warteschleife seine Reservierung zu erneuern. In dem Fall würde P_1 beim nächsten Versuch, die Sperre (durch einen erneuten Aufruf von `lockup`) für sich zu beanspruchen, feststellen, dass P_0 (1) eine Reservierung vorgenommen hat und (2) als nächster die Sperre erheben darf. Daraufhin zieht P_1 seine Reservierung zurück und wartet (innere `while`-Schleife), bis er beziehungsweise sein Prozessor als nächster an der Reihe ist. Verlässt P_0 schließlich die äußere Warteschleife, ist er der Schlosshalter.
Die Lösung von Peterson vermeidet die in der vorigen Variante von Dekker noch mögliche Überholung eines Prozesses. Überhaupt besticht dieser Algorithmus durch seine vergleichsweise sehr einfache Struktur:

```
void lockup(lock_t *lock) {                    /* acquire read/write lock */
    unsigned self = aegis();                     /* read own processor id */

    lock->join[self] = true;                        /* make a reservation */
    lock->turn = self;                                /* defer to myself */
    while (lock->join[self^1] && (lock->turn == self));    /* contention? */
}
```

Nachdem ein Prozess seine Reservierung für die Sperre vorgenommen hat, gibt er sich eigennützig selbst den Vortritt. Im Falle einer Konkurrenzsituation, wenn also P_0 und P_1 dieselbe Sperre zugleich erheben wollen, wird immer derjenige von beiden, dessen Schreiboperation zur Festlegung des Vortrittsrechts zuletzt zur Wirkung kam, auch den Vortritt erhalten. Angenommen, P_1 konnte als letzter das Vortrittsrecht festschreiben. Sowohl P_0 als auch P_1 haben in dieser Konkurrenzsituation ihre Reservierung durchgeführt. Damit muss P_0 warten, da die Reservierung von P_1 noch gilt und P_1 auch den Vortritt hat. Obwohl die Reservierung von P_0 ebenfalls gilt, passiert P_1 jedoch die Warteschleife, da P_0 eben nicht den Vortritt hat. Da ein Prozess, während er warten muss, seine Reservierung nicht zurücknimmt, lässt es diese Konstruktion auch nicht zu, dass im Verlauf der Auflösung der Konkorrenzsituation der eine Prozess von dem anderen Prozess überholt wird.
Die Aufhebung der Sperre (`unlock`) entspricht der Originallösung von Dijkstra (vgl. S. 106). Da ein Prozess beim Versuch, die Sperre zu erheben (`lockup`), sich ja immer selbst den Vortritt gibt, entfällt hier die bei der Lösung von Dekker noch benötigte diesbezügliche An-

weisung beim Aufheben der Sperre.
Die bisher vorgestellten Lösungen von Dekker und Peterson haben gemeinsam, dass jeder der Synchronisation anstrebenden Prozesse lesend und schreibend auf die ↑gemeinsame Variable zugreift, die den als nächsten in einer Konkurrenzsituation zu bevorzugenden Prozessor bestimmt (`turn`). Ein solcher *Multischreiberansatz* strapaziert den ↑Zwischenspeicher in besonderem Maße und zieht für gewöhnlich Leistungseinbußen im Falle wettstreitiger Prozesse nach sich. Aus diesem Grunde verfolgt der Algorithmus von Kessels einen *Einzelschreiberansatz*. Hierzu verfügt jeder Prozessor über eine eigene Variable (`turn`) für den gegebenenfalls zu bevorzugenden Prozessor. Jeder der beiden Prozesse führt Lesezugriffe sowohl auf die eigene als auch auf die fremde Variable aus, aber Schreibzugriffe betreffen immer nur die eigene Variable. Entsprechend ist die zur Modellierung der Sperre benötigte Datenstruktur zu erweitern, wie nachfolgendes Beispiel zeigt:

```
typedef volatile struct lock {                    /* read/write memory lock */
    bool join[NCPU];                                   /* reservation flags */
    unsigned turn[NCPU];                           /* preferenced processor */
} lock_t;
```

Im Wesentlichen folgt die Lösung nach Kessels der von Peterson vorgeschlagenen Struktur (s. oben). Da nun aber nicht mehr eine einzelne Variable den zu bevorzugenden Prozessor benennt, sondern ein Feld solcher Variablen, besteht der Trick beim Festlegen eben dieses Prozessors darin, den Wert des eigenen Feldeintrags auf Grundlage des Werts des Feldeintrags des jeweils anderen Prozessors u berechnen. Wie bei Peterson geschieht dies, nachdem ein Prozess seine Reservierung für die Sperre vorgenommen hat:

```
void lockup(lock_t *lock) {                      /* acquire read/write lock */
    unsigned self = aegis();                       /* read own processor id */

    lock->join[self] = true;                          /* make a reservation */
    lock->turn[self] = ((lock->turn[self^1] + self) % 2);
    while (lock->join[self^1]  /* peer has reserved and not my turn, spin */
       && (lock->turn[self] == ((lock->turn[self^1] + self) % 2)));
}
```

Dabei bestimmt der Ausdruck `((lock->turn[self^1] + self) % 2)` den jeweils zu bevorzugenden Prozessor, und zwar aus eigener (`self`) Sicht des Prozesses, der die Sperre für sich beansprucht. Initial sei das Exemplar der Sperre wie folgt mit Werten vorbelegt:

```
lock_t lock = { {false, false}, {0, 0} };
```

Beim Versuch, die Sperre zu erheben (`lockup`), bekommt der eigene (`self`) Feldeintrag für den Prozessor eines Prozesses einen Wert zugewiesen, der nicht nur, wie im Falle von Peterson, immer seinen eigenen Prozessor identifiziert, sondern zusätzlich abhängig ist vom Wert des Feldeintrags des Prozessors des jeweils anderen Prozesses (`self^1`). Dieser Wert regelt den weiteren Verlauf nur im Falle der Konkurrenzsituation, also wenn beide Prozesse, P_0 und P_1, ihre Reservierung vorgenommen haben (Kopfschleife, erste Bedingung). Ein Prozeß muss dann warten, wenn in dieser Situation der Wert seines Feldeintrags unverändert geblieben ist (Kopfschleife, zweite Bedingung). Dies bedeutet dann aber, dass der jeweils andere konkurrierende Prozess es nach seiner Reservierung noch nicht geschafft hat, seinen eigenen Feldeintrag zu setzen. Sobald dieser Prozess seinem Feldeintrag allerdings einen Wert zuweisen konnte, erhält der auf dieses ↑Ereignis wartende Prozess den Vortritt und verlässt die Warteschleife.
Jeder von den beiden zu einem Zeitpunkt konkurrierenden Prozesse berechnet den Wert seines eigenen Feldeintrag aus der eigenen Prozessornummer (`self` $\in \{0, 1\}$) und dem Wert des Feldeintrags seines jeweiligen Konkurrenten (`self^1` $\in \{0, 1\}$). Für den dann durch ↑Binärarbitration aufgelösten möglichen Konflikt zwischen den beiden Prozessen ergeben sich schließlich folgende Konstellationen:

- P_0 muss auf die Zuweisung von P_1 warten, wenn das Tupel (`turn`) den Wert $(0, 0)$ oder

$(1,1)$ hat. P_1 wird in dieser Situation seinen Feldeintrag auf den gegensätzlichen Wert des Feldeintrags von P_0 setzen, das heißt, $(0,0) \rightsquigarrow (0,1)$ beziehungsweise $(1,1) \rightsquigarrow (1,0)$. P_0 verlässt die Warteschleife und hält dann die Sperre. P_1 muss anschließend darauf warten, dass P_0 die Sperre wieder aufhebt (`unlock`) und erhält dann die Sperre.

- P_1 muss auf P_0 warten, wenn das Tupel (`turn`) den Wert $(0,1)$ oder $(1,0)$ hat. P_0 wird in dieser Situation seinen Feldeintrag auf den Wert des Feldeintrags von P_1 setzen, das heißt, $(0,1) \rightsquigarrow (1,1)$ beziehungsweise $(1,0) \rightsquigarrow (0,0)$. P_1 verlässt die Warteschleife und hält dann die Sperre. P_0 muss anschließend darauf warten, dass P_1 die Sperre wieder aufhebt (`unlock`) und erhält dann die Sperre.

Die Aufhebung der Sperre (`unlock`) entspricht der Originallösung von Dijkstra (vgl. S. 106). Grundsätzlich wird in all den hier vorgestellten Lösungen davon ausgegangen, dass immer nur der Prozess, der die Sperre erhoben hat, diese später auch wieder aufheben wird. Ein anderes Verhalten deutet auf einen Programmierfehler. Um sowohl zu erkennen als auch zu verhindern, dass der falsche Prozess die Sperre aufhebt, wäre beim Erheben lediglich der sperrende Prozess (*lock holder*) zu vermerken und beim Aufheben der entsperrende Prozess auf Übereinstimmung zu überprüfen (vgl. ↑Mutex).

lesen (en.) ↑*read*. ↑Daten aus einem ↑Speicher erkennend entnehmen (in Anlehnung an den Duden). Hier das Kopieren von ↑Daten, die mit einer eindeutigen ↑Adresse innerhalb der ↑Speicherhierarchie versehenen sind. Die ↑Aktion legt eine Kopie auf wenigstens einer tieferen Hierarchiestufe an beziehungsweise führt die Daten bestimmten Verarbeitungseinheiten des ↑Prozessors zu. Sei ↑virtueller Speicher angenommen, dann bedeutet ein solcher Lesevorgang beispielsweise im Falle eines ↑Seitenfehlers, das adressierte Datum aus dem ↑Umlagerungsbereich (Stufe 6/5) zu holen, es im ↑Hauptspeicher (Stufe 4) zu platzieren, in einer ↑Zwischenspeicherzeile (Stufe 3) abzulegen, um es schließlich in einem internen ↑Register (Stufe 1) der ALU vorliegen zu haben. Anders als ↑schreiben wird abwärts jeweils ein weiteres ↑Exemplar auf tieferer Stufe abgelegt, und zwar nach der auf höherer Stufe existierenden Vorlage.

Leser (en.) ↑*reader*. Ein ↑Daten ↑lesender und verarbeitender ↑Prozess, der gegebenenfalls in ↑Konkurrenz mit anderen seiner Art oder ↑Schreibern agiert.

Leser-Schreiber-Problem (en.) ↑*readers-writers problem*. Beispiel für eine allgemeine, komplizierte Fragestellung zum ↑Programmablauf bei ↑Nebenläufigkeit im ↑Rechensystem. Im Vordergrund steht ↑wechselseitiger Ausschluss, und zwar zur ↑Koordinierung mehrerer ↑nebenläufiger Prozesse beim gleichzeitigen Zugriff auf einen bestimmten Programmbereich, ausgewiesen als ↑kritischer Abschnitt. In dem Szenario gibt es zwei unterschiedliche Klassen solcher Prozesse, die als ↑Leser und ↑Schreiber bezeichnet werden. Die Leser können den Abschnitt gemeinsam nutzen, aber die Schreiber müssen exklusiven Zugriff haben.
Das Szenario wird in verschiedenen Varianten beschrieben, die (seid erstmaliger Diskussion, 1972) jeweils gesonderte Lösungsvorschläge hervorgebracht haben. Zur Prozesskoordinierung findet für gewöhnlich der ↑Semaphor Verwendung, wobei Anzahl wie auch Konstellation je nach Szenario variiert. Diese Unterschiede stellen sich wie folgt dar:

Problem 1 Kein Leser wartet beim Eintritt in den kritischen Abschnitt, es sei denn, ein Schreiber hat bereits die Erlaubnis erhalten, eben diesen Abschnitt zu betreten; kein Leser sollte einfach warten müssen, nur weil ein Schreiber darauf wartet, bis andere Leser den Abschnitt passiert haben. Leser genießen Vorrang vor Schreibern.

```
Reader:
    P(mutex);
    readers++;
    if (readers == 1) P(w);
    V(mutex);

    read(…);

    P(mutex);
    readers--;
    if (readers == 0) V(w);
    V(mutex);
```

```
Writer:
    P(w);

    write(…);

    V(w);
```

Problem 2 Sobald ein Schreiber zum Eintritt in den kritischen Abschnitt bereit ist, tritt er auch so schnell wie möglich ein; ein Leser, der eintrifft, nachdem ein Schreiber seinen Eintritt in den kritischen Abschnitt angekündigt hat, muss warten, auch wenn dieser Schreiber wartet. Schreiber genießen Vorrang vor Lesern.

```
Reader:
    P(mutex3);
    P(r);
    P(mutex1);
    readers++;
    if (readers == 1) P(w);
    V(mutex1);
    V(r);
    V(mutex3);

    read(…);

    P(mutex1);
    readers--;
    if (readers == 0) V(w);
    V(mutex1);
```

```
Writer:
    P(mutex2);
    writers++;
    if (writers == 1) P(r);
    V(mutex2);
    P(w);

    write(…);

    V(w);
    P(mutex2);
    writers--;
    if (writers == 0) V(r);
    V(mutex2);
```

Problem 3 Allen Lesern und Schreibern wird der Zugang zum kritischen Abschnitt in der Reihenfolge ihres Eintreffens gewährt. Wenn ein Schreiber ankommt, während sich bereits ein oder mehrere Leser in dem Abschnitt befinden, wartet er, bis diese Leser den Abschnitt verlassen haben. Neue Leser, die in der Zwischenzeit ankommen, müssen warten, bis der Schreiber den kritischen Abschnitt verlassen hat.

```
Reader:
    P(queue);
    P(mutex);
    if (readers == 0) P(entry);
    readers++;
    V(queue);
    V(mutex);

    read(…);

    P(mutex);
    readers--;
    if (readers == 0) V(entry);
    V(mutex);
```

```
Writer:
    P(queue);
    P(entry);
    V(queue);

    write(…);

    V(entry);
```

Angemerkt sei, dass die geforderte Zulassungsreihenfolge (↑FCFS) der Prozesse nicht

durch die algorithmische Konstruktion gesichert ist, sondern auf die Reihungsdisziplin der ↑Warteschlange im Semaphor (`queue`) zurückgeht. Eine von FCFS abweichende Disziplin, beispielsweise um ↑Prioritätsverletzungen gegenüber prioritätsorientierter ↑Prozesseinplanung vorzubeugen, garantiert keinen Zugang zum kritischen Abschnitt (`read`, `write`) in der Reihenfolge, wie die Leser oder Schreiber ihr jeweiliges Zugangsprotokoll (eingangs mit dem ersten `P`) aufnehmen.

Diese Beispiele machen nicht nur die Schwierigkeiten deutlich, ein ↑nichtsequentielles Programm überhaupt zu entwickeln und die durch die Programmanweisungen ermöglichten Prozesse zu verstehen. Vielmehr zeigen sie auch die dabei bestehende besondere Herausforderung, nämlich sicherzustellen, in einem solchen Programm für korrekte Prozesssynchronisation zu sorgen — so übernimmt der Autor auch keine Gewähr für Korrektheit und Vollständigkeit der präsentierten Programmskizzen. `;-)` Darüber hinaus lassen diese Beispiele zurecht vermuten, dass einfache Lösungen auf höherer ↑Abstraktionsebene als sie mit dem Semaphor gegeben ist, etwa durch ↑kritische Regionen oder dem ↑Monitor, nicht existieren. Schließlich ist festzuhalten, dass der ↑Mutex allein, ganz im Gegensatz zum Semaphor, in dem ganzen Zusammenhang keine wirkliche Erleichterung bringt, da er nämlich keine direkten Wege für die benötigte ↑einseitige Synchronisation bietet.

level of abstraction (dt.) ↑Abstraktionsebene.

level-triggered interrupt (dt.) pegelgesteuerte ↑Unterbrechung, ↑Pegelsteuerung.

LF Abkürzung für (en.) ↑*line feed*. Steuerzeichen, kodiert als $\texttt{0A}_{16}$ in ↑ASCII.

libc Bezeichnung einer ↑Bibliothek für ↑C, auch „*C standard library*“. Umfasst Makros, Typdefinitionen und Funktionen für die verschiedensten Zwecke: Zeichenkettenverarbeitung, mathematische Berechnungen, Ein-/Ausgabe, Speicherverwaltung, sowie betriebssystemnahe Operationen.

library (dt.) ↑Bibliothek.

LIFO Abkürzung für (en.) *last in, first out*, (dt.) der umgekehrten Reihe nach; Lagerungsverfahren, gleichbedeutend mit ↑LCFS.

light-weight process (dt.) ↑leichtgewichtiger Prozess.

limit priority (dt.) ↑Grenzpriorität.

line feed (dt.) ↑Zeilenvorschub.

line printer (dt.) ↑Zeilendrucker.

linear address (dt.) ↑lineare Adresse.

lineare Adresse (en.) ↑*linear address*. Bezeichnung für eine eindimensionale ↑Adresse.

link register (dt.) ↑Verbindungsregister.

link time siehe (en.) ↑*binding time*.

link trap (dt.) ↑Bindungsfalle; Mechanismus in ↑Multics.

linkage (dt.) ↑Bindung.

linkage fault (dt.) ↑Bindungsfehler.

linked list (dt.) ↑verkettete Liste.

linker (dt.) ↑Binder.

linking (dt.) ↑Binden.

linking loader (dt.) ↑bindender Lader.

Linux Mehrplatz-/Mehrbenutzerbetriebssystem, von ↑UNIX abstammend. Erste Installation September 1991 (Intel 80386), programmiert in ↑C.

LL Abkürzung für (en.) *load linked*. Bezeichnung für eine ↑Elementaroperation mit einem Operanden: `word_t LL(word_t *)`. Der Operand ist die ↑Adresse eines ↑Speicherworts, dessen Inhalt gelesen und als Ergebnis geliefert wird. Dabei verbucht die Operation für den ausführenden ↑Prozessor eine Reservierung auf die angegebene Adresse. Eine gegebenenfalls für einen anderen Prozessor bestehende Reservierung auf diese Adresse wird gelöscht.
Ein solcher ↑Maschinenbefehl ist nur typisch für ↑RISC. In Kombination mit ↑SC erlaubt er die ↑Synchronisierung von gleichzeitigen ↑Prozessen, die über eine gemeinsame ↑Variable miteinander gekoppelt sind (s. auch ↑LL/SC).

LL/SC Bezeichnung für ein Paar von Spezialbefehlen, ↑LL und ↑SC, zur ↑Synchronisierung gleichzeitiger und über eine gemeinsame ↑Variable miteinander gekoppelter ↑Prozesse. Diese Befehle sind nur typisch für ↑RISC, sie erlauben aber die Nachbildung der für ↑CISC typischen unteilbaren (*atomic* ↑*read-modify-write*) ↑Maschinenbefehle.

Exkurs Gängige Befehle eben genannter Art, mit der sich viele grundlegende Probleme zur ↑Synchronisation gekoppelter Prozesse allein auf ↑Befehlssatzebene lösen lassen, sind ↑FAA, ↑CAS und ↑TAS. Für diese Operationen sind beispielhaft die nachfolgenden Implementierungsmuster angegeben:

```
word_t faa(word_t *ref, word_t val) {                    /* fetch and add */
    word_t aux;                                            /* placeholder */

    do aux = LL(ref);                        /* fetch and make reservation */
    while (!SC(ref, aux + val));    /* add and, if open reservation, store */

    return aux                              /* deliver fetched (old) value */
}
```

Dieses Muster ist zugleich ein einfaches Beispiel für die ↑nichtblockierende Synchronisation, mit der das FAA durch eine ↑Transaktion nachgebildet wird. Die Transaktion beginnt mit LL, um den aktuellen Inhalt der gemeinsamen Variablen (`*ref`) zu lesen. Sie endet, wenn SC gelingt und damit der Variablen ein neuer Wert (`aux + val`) zugewiesen wurde. Die Transaktion wird wiederholt, wenn SC scheitert. Allerdings hat damit diese Lösung eine andere ↑Fortschrittsgarantie als mit einem FAA, das eine echte ↑atomare Operation des zugrunde liegenden ↑Prozessors bildet. Die hier skizzierte Implemenierung ist lediglich *sperrfrei*, nicht aber *wartefrei*. Im Gegensatz dazu schwächt die Nachbildung eines CAS die ursprünglich wartefreie Fortschrittsgarantie nicht ab:

```
int cas(word_t *ref, word_t old, word_t new) {        /* compare and swap */
    return (LL(ref) == old) && SC(ref, new);
}
```

Das nachgebildete CAS scheitert entweder wenn der mit LL gelesene Wert der gemeinsamen Variablen nicht mehr dem ursprünglichen Wert (`old`) entspricht oder wenn SC scheitert, weil die eben zuvor mit LL vorgenommene Reservierung durch einen anderen Prozess bereits schon wieder aufgehoben wurde. Beide Fälle bedeuten, dass ein ↑gleichzeitiger Prozess auf die gemeinsame Variable mittels LL oder SC erfolgreich zugegriffen hat. Auf Basis dieser Lösung lässt sich zugleich und sehr einfach ein TAS skizzieren:

```
int tas(word_t *ref) {                                    /* test and set */
    return cas(ref, 0, 1);           /* deliver: *ref == 0 ? *ref = 1 : 0 */
}
```

Für diese Nachbildung ist der durch (ein nachgebildetes oder echtes) CAS erreichte bedingte

Schreibzugriff auf die adressierte gemeinsame Variable bemerkenswert. Gemeinhin schreibt ein echtes TAS immer den Wert 1, nachdem es den alten Wert aus der Variablen gelesen und als Rückgabewert zwischengespeichert hat. Dadurch ruft das echte TAS wesentlich stärkere ↑Interferenz hervor, verursacht durch das mit jedem Schreibvorgang ablaufende Protokoll zur Wahrung der ↑Speicherkohärenz im ↑Zwischenspeicher. Die hier gezeigte Lösung erzeugt in einer ↑Konkurrenzsituation wesentlich weniger ↑Hintergrundrauschen.

load (dt.) ↑Systemlast.

load address (dt.) ↑Ladeadresse.

load balance (dt.) ↑Lastausgleich.

load balancing (dt.) ↑Lastverteilung.

load distance (dt.) ↑Ladedistanz.

load module (dt.) ↑Lademodul.

load time siehe (en.) ↑*loading time*.

loader (dt.) ↑Lader.

loading (dt.) ↑Laden.

loading time (dt.) ↑Ladezeit.

location counter (dt.) ↑Adresszähler.

Loch (en.) ↑*hole*. Hohlraum im ↑Arbeitsspeicher, freier ↑Speicherbereich. Unbenutzter ↑Adressbereich bestimmter Länge, der zwar zur Speicherung von ↑Text oder ↑Daten zur Verfügung stehen könnte, aber (in logischer Hinsicht) in keinem ↑Prozessadressraum verfügbar ist. Gegenteil zu einem ↑Gebrauchsstück.

Lochkarte (en.) ↑*numeric punch card*. Bezeichnung für ein ↑Speichermedium, auf dem ↑Daten mit Hilfe von Lochungen permanent aufbewahrt werden. IBM hatte eine Karte von 80 Spalten, 12 Zeilen und mit rechteckigen Löchern zur Kodierung patentieren lassen (1928), die das Standardformat in der ↑Stapelverarbeitung darstellt — ein Format, das übrigens auch für Bildschirmfenster mit einem Vorgabewert von 80×24 Zeichen (d.h., zwei Karten untereinander) nach wie vor präsent ist. Pro Spalte wird für gewöhnlich nur ein Zeichen kodiert. Zur Kodierung ganzzahliger Werte $[0, 9]$ werden die unteren 10 Lochpositionen (d.h., Zeilen; auch als „numerische Zone“ bezeichnet), von oben nach unten um eine Spalte nach rechts versetzt, genutzt. Die drei obersten Zeilen dienen der Zonenlochung (*zone punches*), ursprünglich nur um Vorzeichen und die Ziffer `0` zu kodieren: positives Vorzeichen in Zone (Zeile) 12, negatives Vorzeichen in Zone (Zeile) 11, `0` in Zone (Zeile) 10. Durch die Zonenbildung ist die Mehrfachlochung möglich, etwa um Ziffern von Buchstaben und Sonderzeichen zu unterschieden. Beispielsweise hat ein Buchstabe zwei Lochungen, die erste in Zone 12 bis 10 und die zweite für eine Ziffer $[1, 9]$. Derart kodiert ein Loch in Zone 12 und ein weiteres in der numerischen Zone die Buchstaben `A` bis `I`, mit `A` als Ziffer 1 und `I` als Ziffer 9. Entsprechendes für die Buchstaben `J` bis `R` mit einem Loch in Zone 11 und `S` bis `Z` mit einem Loch in Zone 10 (oberste Zeile der numerischen Zone), hier jedoch nur noch die Ziffern $[2, 9]$. Die Kodierung von Satz- und Sonderzeichen geschieht analog. Diese Form der Informationsrepräsentation bildet nach wie vor die Grundlage für ↑EBCDIC. An der Oberkante der Karte wird oft das so in einer Spalte kodierte Zeichen zusätzlich noch für den Menschen lesbar dargestellt, so es sich um ein druckbares Zeichen handelt.
Für gewöhnlich trifft ein ↑Programm, das die auf der Karte kodierten Informationen verarbeiten soll, bestimmte Annahmen über die Zeilen-/Spaltenorganisation. So definiert(e) der ↑Übersetzer für ↑FORTRAN etwa folgenden Kartenaufbau:

Spalte	Leitkarte	Folgekarten
1–5	Anweisungsnummer	
6	leer oder die Ziffer `0`	eine Ziffer ungleich `0`
7–72	Anweisung	Anweisungsfortsetzung
73–80	optionale Kartenidentifikation	

Steht in der ersten Spalte das ↑Fluchtsymbol `C`, sind die Spalten 2–80 frei für einen Kommentar: der Übersetzer ignoriert alle folgenden Zeichen bis zum Zeilenende. Anderenfalls bieten Spalten 1–5 Platz zur Kodierung der Nummer einer bestimmten Anweisung innerhalb des Programms. Diese Nummer ist die Spungmarke für eine Sprunganweisung (`goto`) und legt damit die Anweisung fest, mit der die Programmabarbeitung im Falle einer Verzweigung fortgesetzt wird. Spalte 6 dient der Unterscheidung zwischen Leit- und Folgekarte. Mit der Leitkarte beginnt eine neue FORTRAN-Anweisung. Sollte diese Anweisung mehr als 65 Zeichen zur Kodierung benötigen, wird sie auf der jeweils nachfolgenden Karte fortgesetzt. Insgesamt kann eine solche Anweisung somit bis zu 10 aufeinanderfolgende Karten belegen. Die Kartenidentifikation in den Spalten 73–80 dient der durchgehenden Nummerierung aller Karten des Programms, einschließlich der Kommentarkarten. Für gewöhnlich erfolgt die Nummerierung beispielsweise in 10er Schritten: damit wird die Korrektur von Programmierfehlern erleichtert, wenn nämlich durch Änderung der Anweisung neue Karten in einen bestehenden ↑Stapel eingefügt werden müssen (diese werden dann in jeweils dazwischen liegenden 1er Schritten nummeriert).

Lochkartenleseprogramm Bezeichnung für ein ↑Programm, das eine ↑Lochkarte nach der anderen einliest und die darauf kodierten Informationen in den ↑Hauptspeicher bringt. Dazu muss das Programm selbst im Hauptspeicher zur Ausführung bereit stehen, wohin es (vor der Ära ↑nichtflüchtiger Speicher) von Hand geladen wird (↑*bootstrapping*). Nachdem das Programm durch ↑Ureingabe ins System eingespeist und gestartet worden ist, liest es für gewöhnlich ein auf Lochkarten gespeichertes und direkt von der ↑CPU ausführbares (d.h., binär kodiertes) Steuerprogramm an einen vorgegebenen Platz in den Hauptspeicher ein. Dieses Steuerprogramm nutzt sodann das urgeladene Programm oder verfügt über ein eigenes ↑Unterprogramm, um im normalen Betrieb Lochkarten einzulesen.

Lochstreifen (en.) ↑*punched paper tape*. Vorläufer der ↑Lochkarte. Anfangs ein Papierstreifen mit fünf Kanälen zur Übertragung von ↑Daten im ↑Fernschreibkode. Später um drei Kanäle erweitert für 8-Bit breite Kodes, insbesondere auch für ↑ASCII. Beiden Ausführungen gemeinsam ist in jeder Spalte (auch Reihe genannt) ein Führungsloch zwischen dem dritten und vierten Datenloch (von unten), wodurch 5-Kanal-Streifen auch durch 8-Kanal-Geräte verarbeitet (gelesen, gelocht) werden können. Die Kodierung erfolgt durch für gewöhnlich runde Löcher innerhalb eines quadratischen Rasters von 1/10 Zoll, womit pro Zoll 10 Zeichen (eins pro Spalte) dargestellt werden können. Bei einer Gesamtlänge von etwa 350 m können bis zu 120 000 Zeichen gespeichert werden. Neben Papier als Herstellungsmaterial, sind die Streifen aus Kunstoff oder einem Laminat von Kunststoff und Metall gefertigt. Durch Zusammenfügen der beiden Enden eines Streifens, lassen sich endlos laufende Steuerstreifen konstruieren.

lock algorithm (dt.) ↑Schlossalgorithmus.

lock holder preemption (dt.) ↑Bevorrechtigung eines ↑Prozesses zur Nutzung des ↑Prozessors trotz erhobener ↑Umlaufsperre. Bezeichnung für einen Vorgang, bei dem ein ↑kritischer Abschnitt durch einen Prozess gesperrt ist und dieser Prozess jedoch zugunsten eines anderen Prozesses vom ↑Prozessor verdrängt wird (↑*preemption*). Folge ist die Verzögerung des die Umlaufsperre haltenden Prozesses (*lock holder*), der dadurch nicht den kritischen Abschnitt verlassen und die Sperre auch nicht aufheben kann. Im Falle von ↑Wettstreit um den Eintritt in den kritischen Abschnitt, erleiden sodann alle wettstreitigen Prozesse desselben oder auch eines anderen Prozessors oder ↑Rechenkerns ebenfalls diese Verzögerung. Dadurch kann die ↑Performanz des gesamten ↑Rechensystems stark absinken.

Die Lösung dieses Problems besteht immer darin. dem Prozessor, der einen kritischen Abschnitt ausführen soll, im ↑Eintrittsprotokoll den Wunsch nach störungsfreiem Ablauf eben dieses Abschnitts mitzuteilen und im ↑Austrittsprotokoll diesen Wunsch wieder zurückzunehmen. Da in Abhängigkeit von der ↑Betriebsart das Problem grundsätzlich verschiedene Ebenen im Rechensystem betrifft, sind auch *ebenenspezifische Lösungen* erforderlich:

1. Ein ↑nichtsequentieller Prozess im ↑Maschinenprogramm. In dem Fall scheidet die rein auf ↑Benutzerebene ablaufende und ohne Unterstützung durch das ↑Betriebssystem auskommende Umlaufsperre aus. Stattdessen sind ↑Systemaufrufe erforderlich, um dem Betriebssystem das benötigte Wissen über die störungsfreie Zone des Maschinenprogramms zu vermitteln. Die damit einhergehenden und teils recht hohen ↑Gemeinkosten könnten nur durch *speicherabgebildete Kommunikation* mit dem Betriebssystem vermieden werden. Dazu ist im ↑Adressraum des Maschinenprogramms ein ↑Adressbereich vorzusehen, der entweder statisch im ↑Mehradressraummodell des Betriebssystems verankert ist oder dynamisch durch einen Systemaufruf eingerichtet wird. In beiden Fällen kann dann durch einfache Lese-/Schreiboperationen dem Betriebssystem signalisiert werden, dass der Prozess eine störungsfreie Zone betreten beziehungsweise verlassen hat. Für gewöhnlich sind derartige Systemaufrufe jedoch nicht gebräuchlich.
2. Die ↑partielle Virtualisierung durch ein Betriebssystem. Der gängige Ansatz besteht darin, mit der Umlaufsperre zugleich auch die eventuelle ↑Unterbrechung oder, als Folge davon, die mögliche ↑Verdrängung des den kritischen Abschnitt durchlaufenden Prozesses zu unterbinden. Hier hat für gewöhnlich die ↑Unterbrechungssperre den Vorzug gegenüber einer ↑Verdrängungssperre, da nur dadurch wirklich (↑IRQ bedingte) Verzögerungen verhindert werden können — ↑NMI ausgenommen. Genau diese Lösung ist dann auch in der ↑Informatikfolklore gemeint, wenn nämlich Umlaufsperren gerade für kurze kritische Abschnitte postuliert werden: nicht die Umlaufsperre, sondern die Unterbrechungssperre ist in dem Fall allerdings in den Vordergrund zu rücken!
3. Bei ↑Paravirtualisierung durch einen ↑VMM oder ein ↑Wirtsbetriebssystem. In solch einem Fall wird zwischen ↑Gastbetriebssystemen herumgeschaltet. Jedes dieser Betriebssysteme verwaltet dann eine ↑virtuelle Maschine, was jedoch keinem Betriebssystem bewusst ist. Angenommen (1) die der virtuellen Maschine zugrunde liegende reale Maschine ist ein ↑Mehrkernprozessor, (2) nur eins dieser Betriebssysteme führt keine partielle Virtualisierung der Rechenkerne seiner virtuellen Maschine durch und (3) verzichtet daher (zu Recht) auf die Unterbrechungssperre in der Umlaufsperre. Die (wie schon bei der partiellen Virtualisierung) typischerweise durch ein ↑Zeitteilverfahren bewirkte Bevorrechtigung einer virtuellen Maschine kann dann zum Leistungseinbruch in dem unter (2) erwähnten Betriebssystem führen. Da weder Gastbetriebssystem bewusst ist, dass es auf einer virtuellen Maschine läuft noch dem eine virtuelle Maschine bereitstellenden VMM bewusst ist, was auf einer virtuellen Maschine läuft, ist das Problem für ↑Vollvirtualisierung übrigens nicht lösbar.

Jeder dieser Ansätze bedeutet letztlich, die Unterbrechung eines Prozesses geeignet zu unterbinden. Kommt die Anforderung dazu, wenn der Prozessor in einem nicht privilegierten ↑Arbeitsmodus tätig ist (↑*user mode*), so darf nicht darauf vertraut werden, dass der betreffende Prozess das Verlassen der störungsfreien Zone auch anzeigt und dadurch der Prozessor in seinen Normalzustand zurückversetzt werden kann. Dies betrifft insbesondere die Fälle 1 und 3, für die daher eine ↑Zeitsperre vorzunehmen ist, um die Rückkehr in den Normalzustand auch sicherstellen zu können. Die einer solche Sperre zuzuordnenden Zeitspanne sollte sich idealerweise nach der zu erwartenden ↑Ausführungszeit des die störungsfreie Zone definierenden kritischen Abschnitts richten und wäre daher als ↑Vorwissen dem jeweiligen System (Betriebssystem, VMM) im Eintrittsprotokoll der Umlaufsperre zu vermitteln, darf aber einen bestimmten vorgegebenen Wert eben auch nicht überschreiten. Auch für Fall 2 ist eine solche Maßnahme angebracht, um einer eventuellen Fehlfunktion (↑*failure*) des Betriebssystems vorzubeugen.

Anmerkung Die wörtliche Übersetzung des Begriffs ließe vermuten, dass der die Sperre haltende Prozess „bevorrechtigt“ wird. Dies ist aber nicht zutreffend! Bevorrechtigt wird ein beliebiger anderer Prozess, der den sperrenden Prozess vom Prozessor „verdrängt“ und gegebenenfalls auch überhaupt nicht den gesperrten kritischen Abschnitt durchläuft. Der Begriff steht für eine ↑Aktionsfolge, die eine unerwünschte Auswirkung als Folge der Verdrängung des „Schlosshalters“ zur Bevorrechtigung eines anderen Prozesses haben kann.

lock variable (dt.) ↑Schlossvariable.

lock-freedom (dt.) ↑Sperrfreiheit.

Löcherliste (en.) ↑*hole list*. Bezeichnung für eine ↑Freispeicherliste, auf der jedes verzeichnete ↑Loch einem freien ↑Speicherbereich im ↑Hauptspeicher entspricht.

log-out (dt.) ↑Abmeldung.

logical address (dt.) ↑logische Adresse.

logical address space (dt.) ↑logischer Adressraum.

login (dt.) ↑Anmeldung.

logische Adresse (en.) ↑*logical address*. Bezeichnung für eine ↑Adresse, die von der Lage eines ↑Speicherworts im ↑Hauptspeicher abstrahiert. Der Definitionsbereich dieser Adresse ist ein bestimmter ↑logischer Adressraum.
Von Ausnahmen (↑*identity mapping*) einmal abgesehen, ist nicht davon auszugehen, dass die durch eine logische Adresse angegebene Lage einer ↑Speicherstelle im Hauptspeicher identisch ist mit ihrer wirklichen Lokalität (↑reale Adresse). Daher ist eine ↑Adressabbildung erforderlich, bevor der Zugriff auf die betreffende Speicherstelle zulässig ist. Wird eine solche Adresse von der ↑CPU im ↑Abruf- und Ausführungszyklus appliziert, ist ein ↑Busfehler in aller Regel ausgeschlossen. Jedoch kann es zu einer ↑Schutzverletzung kommen, nämlich wenn die Adresse außerhalb ihres Adressraums liegt (↑*segmentation fault*).

logische Synchronisation siehe ↑unilaterale Synchronisation.

logischer Adressraum (en.) ↑*logical address space*. Bezeichnung für einen ↑Adressraum, der von der Lage eines ↑Prozesses im ↑Hauptspeicher abstrahiert. Eine bestimmte Menge von Nummern, wobei jede einzelne davon eine ↑logische Adresse repräsentiert.
Ein solcher Adressraum ist durch eine ↑abstrakte Maschine (↑Kompilierer, ↑Betriebssystem) definiert. Diese Maschine sieht jede ↑Adresse in diesem Adressraum als gültig an für einen in diesem Adressraum daselbst sich bewegenden und eine der Adressen daraus erzeugenden Prozess. Verwendet der Prozess dagegen eine Adresse, die außerhalb seines definierten Adressraums liegt, wird der Zugriff damit von der abstrakten Maschine abgefangen (↑*exception*) und eine ↑Ausnahmebehandlung vorgenommen.
Eine logische Adress ist vor oder zur ↑Laufzeit von dem ↑Programm, das den Prozess beschreibt, auf eine ↑reale Adresse abzubilden. Vor Laufzeit meint Abbildung durch ↑Übersetzung, ↑Bindung oder spätestens zur ↑Ladezeit des Programms. Demgegenüber beansprucht die Abbildung zur Laufzeit eine ↑MMU, deren Datenstrukturen auf Veranlassung durch den ↑Lader vom Betriebssystem einzurichten sind: nämlich pro Adressraum mindestens eine ↑Seitentabelle oder ↑Segmenttabelle anlegen, je nach Art der MMU, und für jeden ↑Seitendeskriptor beziehungsweise ↑Segmentdeskriptor ein ↑Exemplar in entsprechender Anzahl gemäß der im ↑Lademodul enthaltenen und durch das Betriebssystem vorgegebenen Parameter für das ↑Text-, ↑Daten- und ↑Stapelsegment programmieren.
Die Zielmenge der Abbildung ist in beiden Fällen ein ↑realer Adressraum, wobei dieselbe reale Adresse mindestens eine logische Adresse als Urbild hat (Surjektivität): mehrere logische Adressen (in der Regel) verschiedener Adressräume können auf dieselbe reale Adresse abbilden (↑*shared memory*). Für die abzubildenden Adressen kann ein ↑seitennummerierter

Adressbereich den Rahmen bilden, wobei dieser entweder den gesamten (logischen) Adressraum abdeckt oder nur pro ↑Segment definiert ist. Die Art bestimmt die MMU, wodurch ein ↑seitennummerierter Adressraum oder ↑segmentierter Adressraum vorgegeben ist. Jedes einzelne Segment muss komplett und zusammenhängend im ↑Hauptspeicher vorliegen, damit das betreffende Programm ausgeführt werden kann. Jedoch ist die durch die ↑Ladeadresse vorgegebene Lokalität jeder ↑Seite/jedes Segments zur ↑Laufzeit des Programms veränderlich: die jeweiligen Strukturelemente (Seite, Segment) von dem ↑Prozessadressraum können im Hauptspeicher verschoben werden, ohne dass dies funktionelle Auswirkungen auf den Prozess hätte — er bewegt sich in einem oder mehreren Adressbereichen, die zwar möglicherweise jeweils wachsen oder schrumpfen, aber deren Adressen darin stets gleich bleiben.

Lokalitätsprinzip (en.) ↑*principle of locality*. Bezeichnung für ein bestimmtes Schema, nach dem die von einem ↑Prozess generierte ↑Referenzfolge aufgebaut ist. Unterschieden wird *zeitliche* und *räumliche Lokalität*. Erstere bezieht sich auf die Wiederverwendung von ↑Daten oder ↑Betriebsmittel innerhalb einer vergleichsweise kleinen Zeitspanne, wohingegen letztere Bezug nimmt auf relativ eng benachbarte ↑Adressen. Eine spezielle Variante räumlicher Lokalität bildet die *sequentielle Lokalität*, wenn sich nämlich die Adressen in der betrachteten Zeitspanne linear entwickeln. Eine stark linear ausgeprägte Referenzfolge ist typisch für Schleifen (`while`, `for`) und ↑Rekursionen, die bei ↑Programmablauf nahezu stetige Adressmengen nicht nur im ↑Textsegment hervorbringen, sondern auch im ↑Datensegment (Durchlauf bei Feldern oder verketteten Datenstrukturen) oder im ↑Stapelsegment (↑Aktivierungsblock bei rekursiven Durchläufen in mehreren Inkarnationen „nahtlos“ stapeln).
Dieses Prinzip erlaubt bis zu einem gewissen Grad die Vorhersage des Verhaltens der Prozesse in einem ↑Rechensystem und liefert damit die Grundlage nicht nur zur Optimierung des Rechnerbetriebs, sondern auch zur Durchsetzung bestimmter Gütemerkmale. Ergibt sich eine starke ↑Prozesslokalität, ist dies beispielsweise für die Abschätzung der ↑Arbeitsmenge, ↑Umlagerung logisch zusammenhängender ↑Seiten oder Berechnung der (für ↑SPN, ↑HRRN oder ↑SRTF benötigten) Länge des nächstens ↑Rechenstoßes eines Prozesses sehr förderlich.

long-term scheduling (dt.) ↑langfristige Planung.

lost wake-up (dt.) entgangenes Aufwachen. Bezeichnung für eine ↑Wettlaufsituation, in der ein sich schlafen legender ↑Prozess den für ihn bestimmten Weckruf (*wake-up call*) verpasst. Dabei handelt es sich um eine Variante des auch als „*non-repeatable read*“ bezeichneten Problems von sich zeitlich überlappenden Lese-Schreiboperationen (vgl. S. 390, *read-write conflict*). Typische Ursache für ein sich daraus ergebendes Fehlverhalten des Prozesses ist eine *bedingte Anweisung* nachfolgend skizzierten Musters:

```
if (!occur(&event))                          /* do I have to go to bed? */
    †                                     /* yes, hopefully undisturbed! */
    sleep(&event);             /* bed down: release CPU, enter sleep state */
```

Der Prozess stellt fest, dass ein für seinen weiteren Fortschritt relevantes ↑Ereignis (`event`) noch nicht stattgefunden hat (`occur`: 1. Leseoperation, logisch/wirklich) und wird sich daraufhin bis zum Ereigniseintritt schlafen legen (`sleep`: 2. Leseoperation, logisch). Wenn das Ereignis, angezeigt durch einen anderen Prozess (Schreiboperation), aber genau in der Zeitspanne zwischen der Feststellung der Wartebedingung und dem erfolgten Übergang in den Wartezustand eintritt (†), entgeht dem die Wartebedingung hat prüfenden Prozess das Wecksignal, da er eben noch nicht als wartend (d.h., schlafend) verzeichnet ist: dem das Ereignis anzeigenden (externen) Prozess ist der sich schlafen legende Prozess unbekannt und der sich schlafen legende (das Ereignis nach wie vor erwartende) Prozess kann auch durch erneutes Prüfen der Wartebedingung sein mögliches Fehlerverhalten nicht verhindern.
Einem solchen Fehlverhalten kann in verschiedener Weise vorgebeugt werden. Ein Ansatz besteht darin sicherzustellen, dass das erwartete Ereignis ab dem Zeitpunkt der Abfrage (`occur`) und bis zum Zeitpunkt der Reaktion darauf (`sleep`) nicht angezeigt werden kann. Gemeinhin ist hierfür vor der Abfrage entweder eine ↑Unterbrechungssperre oder eine ↑Ver-

drängungssperre zu setzen, die dann zurückzunehmen ist, wenn die Wartebedingung nicht zutrifft (weil das Ereignis stattgefunden hat) oder sobald der Prozess schläft (er sich als dieses Ereignis erwartend bekannt gemacht hat):

- Die Unterbrechungssperre sorgt dafür, dass die ↑CPU eine an sie gestellte ↑Unterbrechungsanforderung nicht erfährt und damit auch nicht die ↑Unterbrechungsbehandlung stattfinden kann, in deren Verlauf gegebenenfalls das für den Prozess relevante Ereignis hervorgebracht wird. Gleichfalls kann allerdings, bei ↑Flankensteuerung, das von der ↑Peripherie gesandte Signal unwiederbringlich verloren gehen.
- Demgegenüber sorgt die Verdrängungssperre lediglich für die Aussetzung beziehungsweise Verzögerung der ↑Einlastung der CPU mit dem Prozess, der gegebenenfalls das von dem anderen Prozess erwartete Ereignis verursacht.

Grundsätzlich betreffen diese Sperren aber auch Vorgänge (d.h., Unterbrechungsbehandlungen und Prozesse), die überhaupt keinen Bezug zu dem das Ereignis erwartenden Prozess haben müssen. Dadurch wird ↑Nebenläufigkeit im System unnötig eingeschränkt. Darüber hinaus beugen diese Sperren nur der Anzeige eines Ereigniseintritts durch einen Vorgang vor, der sich mit dem Prozess auf demselben Prozessor zeitlich überlappt.
Ein anderer Ansatz ist es, dass der Prozess, bevor er die Ereignisabfrage tätigt, seine Absicht anzeigt, sich in absehbarer Zeit gegebenenfalls schlafen zu legen. Ab dem Moment der Anzeige kann dem Prozess jederzeit das Wecksignal zugestellt werden, auch wenn er selbst noch nicht schläft. Sollte die Wartebedingung zutreffen, wird sich der Prozess nur dann wirklich schlafen legen, wenn das Wecksignal bisher ausgeblieben ist. Im Gegensatz zum ersten Ansatz setzt dieses *sperrfreie Verfahren* nicht voraus, dass sowohl der ereigniserwartende als auch der -anzeigende Prozess auf demselben Prozessor residieren.
Ein typisches Lösungsmuster für beide Ansätze zeigt folgendes Beispiel, wobei Operationen zur Sperrung/Ankündigung (`catch`) und Entsperrung/Abkündigung (`annul`) den wettlaufkritischen Pfad im ↑Programm entsprechend eingrenzen:

```
catch(&event))                          /* be poised for a wake-up call */
if (!occur(&event))                          /* do I have to go to bed? */
    sleep(&event);                              /* yes, bed myself down */
else                                                  /* no, stay awake */
    annul(&event);                      /* wake-up call no longer needed */
```

Der Prozess gibt bekannt, die Anzeige eines Ereignisses einfangen zu wollen (`catch`) und erklärt diese Anzeige für ungültig (`annul`), wenn das Ereignis eingetreten sein sollte. Letzteres geschieht implizit beim Schlafenlegen, sobald der Prozess gefahrlos die Kontrolle über den Prozessor abgegeben hat. Der Trick des sperrfreien Ansatzes besteht nun darin, dass der ein Ereignis einfangen wollende Prozess durch die Ereignisanzeige auf die ↑Bereitliste gelangt — auch dann, wenn er selbst noch nicht schläft: der betreffende Prozess wird immer der ↑Einplanung zugeführt. Damit geht dieser Prozess für die Einlastung nicht verloren. Der mögliche ↑Schwebezustand, den der Prozess durch die Bekanntgabe bereitwillig toleriert, nämlich tendenziell schlafen zu gehen und gleichzeitig auf der Bereitliste zu stehen, muss auch in anderen Fällen beachtet werden — beispielsweise bei der Blockierung eines Prozesses und seiner erneuten Bereitstellung vor der eigentlich beabsichtigten Prozessorabgabe oder die Bereitstellung eines Prozesses, der die Rolle als ↑Leerlaufprozess übernommen hat. Für ihn ist daher für gewöhnlich keine Sonderbehandlung erforderlich.

LRU Abkürzung für (en.) *least recently used*. Bezeichnung einer ↑Ersetzungsstrategie, um den Inhalt eines bestimmten Bereichs im ↑Hauptspeicher oder ↑Zwischenspeicher durch den Inhalt eines gleich großen, anderen Bereichs zu ersetzen. Im Falle von Hauptspeicher wird eine in einem ↑Seitenrahmen platzierte ↑Seite durch eine andere Seite ersetzt (↑virtueller Speicher). Im Falle von Zwischenspeicher wird eine darin platzierte ↑Zwischenspeicherzeile durch eine andere (aus dem Hauptspeicher gelesene) ersetzt. Ersetzt wird die Seite/Zeile, die kürzlich am wenigsten genutzt wurde.

m68k Kürzel für eine Prozessorserie (1979–1994) von Motorola, die vier Generationen von 16/32-Bit Prozessoren umfasste: 68000, 68020, 68040 und 68060. Die Modelle 68010 und 68030 waren lediglich kleinere Revisionen und etablierten keine neue Prozessorgenerationen. Das Modell 68050 erreichte niemals Produktreife. Der 68070 war eine lizensierte Variante des 68000 und wurde von Philips produziert. Die Prozessoren kamen bevorzugt in Arbeitsplatzrechner (↑PC) zum Einsatz (insb. ↑Macintosh, Amiga, Atari) und sind nach wie vor in eingebetteten Systemen weit verbreitet (Freescale).

Mach ↑Betriebssystemkern, erste Installation 1985 (↑DEC VAX). Entwicklung bis 1994 an der Carnegie Mellon University, USA, zur Forschung auf dem Gebiet der ↑Systemprogrammierung. Bildet die Basis unter anderem für ↑Darwin beziehungsweise ↑XNU.

machine language (dt.) ↑Maschinensprache.

machine word (dt.) ↑Maschinenwort.

Macintosh Arbeitsplatzrechner (↑PC), 1984, entwickelt von Apple auf Basis von ↑Prozessoren der ↑m68k-Familie. Maßgeblichen Einfluss auf die Entwicklung hatte der ↑Xerox Alto, der bahnbrechend für die grafischen Benutzungsoberflächen (↑GUI) nachfolgender Rechnergenerationen war. Betrieben wurde das System durch ↑Mac OS.

Mac OS Abkürzung für (en.) *Macintosh Operating System* und von 1996 bis 2016 die Bezeichnung für das ↑Betriebssystem für den ↑Macintosh. Auch bekannt als Mac OS Classic. Erste Installation 1984 (System, ↑m68k), Prozessorwechsel in 1994 (System 7.1.2, ↑PowerPC mit m68k Emulation bis 1997). Anfangs ein an der Benutzerschnittstelle rein grafisch arbeitendes Einbenutzerbetriebssystem, ab Mac OS X ein auf ↑XNU basierendes Mehrplatz-/Mehrbenutzerbetriebssystem. Erste Ausprägung dieser Art in 2001 (Cheetah, ↑PowerPC), abermaliger Prozessorwechsel in 2005 (Tiger, ↑x86) und Wechsel von 32-Bit auf 64-Bit Technologie in 2009 (Snow Leopard).

macOS Ab Herbst 2016 gültige Bezeichnung von ↑Mac OS.

macro instruction (dt.) ↑Makrobefehl.

mailbox (dt.) elektronischer ↑Briefkasten.

main board (dt.) ↑Hauptplatine.

main memory (dt.) ↑Hauptspeicher.

main-memory chunk (dt.) ↑Hauptspeicherstück.

mainframe (dt.) ↑Großrechner.

mainstore miss (dt.) ↑Hauptspeicherfehlzugriff.

major device number (dt.) ↑Hauptgerätenummer.

Makrobefehl (en.) ↑*macro instruction*. Bezeichnung für eine zu einer Einheit zusammengefasste Folge von Befehlen; Kurzform: Makro (Duden). Bei der ↑Übersetzung wird der in einem ↑Programm kodierte Makroaufruf durch die definierte Befehlsfolge ersetzt. Diese Makroexpansion übernimmt gewöhnlich ein Makroprozessor.

malware (dt.) ↑Schadsoftware. Kunstwort, gebildet aus (en.) *malicious software*.

Mantelprozedur (en.) ↑*wrapper procedure*. Umhüllung für ein ↑Unterprogramm, um es in andere Software zu integrieren. Beispiel ist etwa ein in ↑Assembliersprache zu formulierender Aufruf eines in ↑C vorliegenden Unterprogramms zur ↑Unterbrechungsbehandlung, der entsprechend ↑Aufrufkonvention die Inhalte der im Unterprogramm frei verwendbaren Register (↑nichtflüchtiges Register, *callee-saved*) sichert und wiederherstellt. Allgemein jede Art von Software zur Adaptation der formalen Schnittstelle eines Unterprogramms.

manual computer operation (dt.) ↑manueller Rechnerbetrieb.

manueller Rechnerbetrieb (en.) ↑*manual computer operation*. ↑Betriebsart, bei der das ↑Rechensystem gesteuert durch Programmierpersonal nacheinander mit Arbeitspaketen bestückt wird, wobei jedes einzelne Paket durch ein ↑Programm beschrieben ist, von dem zu einem Zeitpunkt stets nur eins zur Ausführung bereit im ↑Hauptspeicher liegt (↑*uniprogramming*). Die Person führt für gewöhnlich folgende Schritte nacheinander aus:

1. die Anweisungen des auszuführenden Programms samt ↑Daten mit dem ↑Kartenlocher auf ↑Lochkarte kodieren und zur Eingabe in das Rechensystem aufbereiten
2. die Lochkarten zum Einlegen in den ↑Kartenleser bereithalten und je nach Art der Programmrepräsentation wie folgt verfahren:
   (a) ist das Programm von der ↑CPU ausführbar (d.h., binär kodiert), weiter bei 3
   (b) liegt das Programm dagegen in Quelltext vor:
      i. zuerst die Lochkarten mit dem direkt von der ↑CPU ausführbaren (d.h., binär kodierten) ↑Übersetzer der ↑Programmbibliothek entnehmen und einlegen
      ii. den Lochkartenleser starten und die Übertragung des Übersetzers in den Hauptspeicher abwarten
      iii. die Ausführung des Übersetzers, der das übersetzte Programm im Haupt- oder ↑Trommelspeicher belässt, durch Knopfdruck starten
      iv. sofern zweckmäßig, das übersetzte und ausführbare Programm zur Ablage in der Programmbibliothek gemäß 7a auf Lochkarten speichern
3. die Lochkarten mit dem (ggf. zu übersetzenden) Programm (ggf. als Eingabe für den Übersetzer) in den Kartenleser einlegen
4. den Kartenleser starten und die Übertragung des Programms in den Haupt- oder Trommelspeicher abwarten (ggf. nach vorheriger Übersetzung)
5. die Ausführung des Programms durch Knopfdruck starten
6. sofern erforderlich, die Lochkarten mit den von dem Programm zu verarbeitenden ↑Daten einlegen und den Kartenleser starten
7. je nach Art der gewünschten Ausgabe, Vorkehrungen zur Darstellung oder Aufbewahrung der Berechnungsergebnisse treffen:
   (a) soll die Ausgabe in maschinenlesbarer Form in eine ↑Bibliothek abgelegt oder durch ein anderes ↑Peripheriegerät noch weiterverarbeitet werden:
      i. zur Aufnahme der Ausgabedaten für eine ausreichende Anzahl leerer Lochkarten sorgen und in einen ↑Kartenstanzer einlegen
      ii. durch Knopfdruck einerseits den Stanzer aktivieren und andererseits die Ausgabe starten, anschließend die Beendigung des Stanzvorgangs abwarten
      iii. die gestanzten Lochkarten dem Stanzgerät entnehmen und zur Aufbewahung in einer Bibliothek geeignet verpacken oder weiter bei 7b
   (b) sofern gewünscht, die Ausgabe in eine für den Menschen lesbare Form bringen:
      i. für ↑Tabellierpapier sorgen und den ↑Zeilendrucker aktivieren oder
      ii. für Zeichenpapier/-material sorgen und den ↑Kurvenschreiber aktivieren
8. die Ausgabe abwarten, von der jeweiligen Ausgabestation abholen und ihrer Bestimmung zuführen

Wesentliche Arbeitserleichterung bringt ↑automatisierter Rechnerbetrieb, bei dem die Schritte 2 bis einschließlich 7 als ↑Auftrag beschrieben sind, dessen automatische Bearbeitung (↑*batch processing*) sodann durch Bedienpersonal (↑*operator*) überwacht abläuft.

Mark I Elektromechanischer ↑Rechner (Harvard, 1944), aufgebaut aus schnellen elektromagnetischen Relais. Der Rechner besaß keinen ↑Hauptspeicher, sondern las einen ↑Maschinenbefehl nach dem anderen (wie bei der ↑Z1) von einem ↑Lochstreifen und führte ihn sofort aus. Die damit verbundene technische Trennung von ↑Text und ↑Daten eines ↑Programms hat den Begriff ↑Harvard-Architektur geprägt.

Mark II Elektromechanischer ↑Rechner (Harvard, 1947), verbesserte Variante des ↑Mark I — aus den Annalen der Informatikwissenschaft bekannt als Ursprung des Begriffs „↑*bug*".

Markierungsbit (en.) ↑*flag bit*. Boole'sche Binärziffer, wobei der Ziffernwert einen bestimmten Zustand widerspiegelt, den es festzuhalten gilt. Auch kurz als *Merker* bezeichnet.

marshalling (dt.) Bildung einer ↑Nachricht aus einer Menge von ↑Daten. Dem Bahnwesen entlehnt, das mit dem Begriff eine *Zugbildung* meint, also die Anordnung oder Aufstellung eines Zugs aus einer Menge von Wagen.

Maschinenadresse (en.) ↑*absolute address*. Bezeichnung für eine gültige ↑reale Adresse im ↑Rechensystem, wobei ein vorgegebener ↑realer Adressraum das ↑Bezugssystem bildet.

Maschinenbefehl (en.) ↑*computer instruction*, *machine instruction*. Instruktion, elementare Anweisung in einem ↑Maschinenprogramm. Eine solche Anweisung besteht aus einen obligatorischen *Operationsteil*, der die ↑Aktion der Maschine bezeichnet, und einen optionalen *Operadenteil*, der diese Aktion mit den von der Maschine zu verarbeitenden Informationen verknüpft. Für gewöhnlich ist jede dieser Anweisungen durch einen eindeutigen *Binärkode* repräsentiert, auch als ↑Maschinenkode bezeichnet.
In Abhängigkeit von der in Bezug genommenen ↑Abstraktionsebene haben diese Anweisungen teils sehr unterschiedliche Formate, und zwar in vertikaler (d.h., betreffs verschiedener Maschinen verschiedener Ebenen) wie auch horizontaler (d.h., betreffs verschiedener Maschinenausprägungen innerhalb derselben Ebene) Hinsicht. Auf ↑Befehlssatzebene ist, je nach Maschinenart beziehungsweise ↑CPU, beispielsweise eine unterschiedliche Anzahl und Form von Adressangaben typisch:

- 0-Adressbefehl, ausschließlich implizite Adressierung der Operanden (Stapelmaschine). Der Befehl nimmt die Operanden von (*pop*) und legt das Operationsergebnis auf (*push*) dem ↑Stapelspeicher ab.
- 1-Adressbefehl, implizite Adressierung des ersten und explizite Adressierung des zweiten Operanden (Akkumulatormaschine). Die ↑Adresse des expliziten Operanden verweist in den ↑Arbeitsspeicher. Der Befehl greift auf den Operanden im Arbeitsspeicher zu, liest den implizit adressierten Operanden aus dem Akkumulator und hinterlässt hier auch das Operationsergebnis. Bei einfachen Transferbefehlen ist der Akkumulator entweder Quell- (*store*) oder Zieloperand (*load*).
- 2-Adressbefehl, explizite Adressierung beider Operanden, wobei einer nur Quell- und ein anderer sowohl Quell- als auch Zieloperand ist (↑CISC). Die ↑Adressen der Operanden verweisen in den ↑Arbeitsspeicher oder den ↑Registerspeicher. Der Befehl liest die Operanden von ihren Adressen (*read*) und hinterlässt das Operationsergebnis an der Adresse des Zieloperanden (*write*).
- 3-Adressbefehl, explizite Adressierung der beiden Quell- und des einen Zieloperanden (↑RISC). Die ↑Adressen der Operanden verweisen in den ↑Arbeitsspeicher oder den ↑Registerspeicher. Der Befehl liest die Quelloperanden von ihren Adressen (*read*) und hinterlässt das Operationsergebnis an der Adresse des Zieloperanden (*write*).

Demgegenüber erweitert die ↑Betriebssystemebene die Menge der zur Verfügung stehenden Anweisungen eines ↑Rechners um ↑Systemaufrufbefehle, die ihrerseits von ↑Betriebssystem zu Betriebssystem in Anzahl und Form variieren. Beide Arten von Anweisungen zusammen bilden die Grundlage für die Formulierung von Maschinenprogrammen.

Maschinenkode (en.) *machine code*, ↑*object code*. Verschlüsselung von einem ↑Maschinenbefehl. Üblich ist der *Binärkode*, allerdings erfolgt die Darstellung eines solchen Befehls gemeinhin als Hexadezimalzahl: so bedeutet 05_{16} bei einem ↑x86-kompatiblen ↑Prozessor die Addition mit dem Akkumulator. Früher war auch die Darstellung als Oktalzahl verbreitet (↑Rechner der PDP-Familie).

Maschinenprogramm (en.) ↑*object program*, *machine program*. Ein durch einen ↑Prozessor ausführbares ↑Programm, dessen kontrollierter Ablauf für gewöhnlich durch ein ↑Betriebssystem gewährleistet ist. Jede Instruktion in diesem Programm entspricht einem ↑Maschinenbefehl, den eine ↑virtuelle Maschine der ↑Maschinenprogrammebene durch ↑partielle Interpretation ausführt. Eine solche Instruktion kann als ↑Systemaufruf *explizit* eine bestimmte ↑Aktion eines Betriebssystems auslösen beziehungsweise wird allein oder unter Mitwirkung des Betriebssystems von der ↑CPU ausgeführt. Das Programm ist zudem *implizit* abhängig von einem Betriebssystem, und zwar wenn es von der Gültigkeit eines Systemzustands ausgeht, die es nicht selbst durch Systemaufrufe anfordert, sondern zur ↑Ladezeit vom Betriebssystem zugesichert wird und zur ↑Laufzeit bestehen bleibt.

Maschinenprogrammebene (en.) ↑*object program level*. Bezeichnung für die oberhalb der ↑Befehlssatzebene angesiedelte und von einem ↑Betriebssystem abhängige Schicht in einem als ↑Mehrebenenmaschine gekennzeichnetem ↑Rechensystem. Diese vor allem durch die ↑Betriebssystemebene gekennzeichnete Schicht erweitert das ↑Programmiermodell und den ↑Befehlssatz einer ↑CPU um Merkmale und Abstraktionen, die von einem Betriebssystem bereitgestellt werden. Typisch ist die ausschließlich *binäre Kodierung* der von dem — aus CPU und Betriebssystem bestehenden, eine ↑abstrakte Maschine bildenden — ↑Prozessor dieser Ebene auszuführenden ↑Programme. Die Programmausführung geschieht einerseits (normalerweise) durch die CPU und andererseits (ausnahmsweise) über ↑partielle Interpretation durch das Betriebssystem.
Dem Konzept nach besteht der Befehlssatz dieser Ebene einerseits aus den sogenannten unprivilegierten Befehlen der Befehlssatzebene und andererseits aus den ↑Systemaufrufbefehlen der Betriebssystemebene. Typischerweise gehört ein ↑privilegierter Befehl der Befehlssatzebene nicht zur Menge der auf dieser Ebene gültigen ↑Maschinenbefehle. Ein ↑Maschinenprogramm wird vom Betriebssystem daher derart (↑*unprivileged mode*) zur Ausführung gebracht, dass ein in dem Programm möglicherweise (bewusst oder unbewusst, aber eben fälschlicherweise) kodierter privilegierter Befehl bei der Ausführung von der CPU abgefangen wird (↑*trap*) und nicht zur Wirkung kommen kann — vorausgesetzt, die CPU verfügt über das benötigte technische Merkmal, zwischen privilegiertem und unprivilegiertem ↑Arbeitsmodus unterscheiden zu können.

Maschinensprache (en.) ↑*machine language*. Programmiersprache, deren Sprachelemente in Form von ↑Maschinenkode repräsentiert sind. Die eigentliche Programmiersprache einer ↑CPU.

Maschinenwort (en.) ↑*machine word*. Informationseinheit in einem ↑Prozessor (↑CPU). Allgemein ausgelegt als Bitvektor, dessen Länge die ↑Wortbreite des Prozessors definiert.

mass storage (dt.) ↑Massenspeicher.

Massenspeicher (en.) ↑*mass storage*. ↑Speicher zur dauerhaften Ablage sehr großer Mengen von ↑Daten aller Art. Oft Synonym für ↑Sekundärspeicher. Umfasst jedoch auch ↑Tertiärspeicher, der nicht permanent im ↑Rechensystem angeschlossen ist und sich in Form von Archiven zeigt. In dem Fall geschehen die Speicherzugriffe komplett entkoppelt (*off-line*) von den ↑Prozessen, die diese Zugriffe veranlassen und kommen erst durch den Eingriff von Bedienpersonal (↑*operator*) oder Roboter zustande.

MCP Abkürzung für (en.) *master control program*. Bezeichnung des ↑Betriebssystems für den ↑B 5000, das als erstes vollkommen in Hochsprache (↑Algol 60) implementiert war.

MCU Abkürzung für (en.) ↑*microcontroller unit*.

medium-term scheduling (dt.) ↑mittelfristige Planung.

Mehradressraum (en.) ↑*multi-address space*. Charakteristische Eigenschaft eines logischen/virtuellen ↑Adressraums, mit anderen seiner Art zugleich im selben ↑Rechensystem definiert und durch ein ↑Betriebssystem unter Zuhilfenahme einer ↑MMU oder ↑MPU eingerichtet zu sein. Die gleichzeitig vorhandenen Adressräume können verschieden groß sein und decken für gewöhnlich denselben ↑Adressbereich ab. Ein im Betriebssystem verwirklichtes ↑Multiplexverfahren sorgt dafür, dass ein ↑realer Adressraum gleichzeitig das Bild für mehr als einen logischen/virtuellen Adressraum darstellt (↑*partial virtualisation*). Als Folge dieses Verfahrens sind die in den vervielfachten Adressräumen definierten ↑Adressen mehrdeutig: dieselbe logische/virtuelle Adresse ist für gewöhnlich mehrfach vorhanden.

Diese Eigenschaft begründet zudem eine bestimmte Art von ↑Schutz, der sicherstellt, dass ein ↑Prozess nicht aus seinen ↑Prozessadressraum ausbrechen kann, das heißt, in seinem Adressraum eingesperrt ist. Genau genommen ist hier ein ↑schwergewichtiger Prozess gemeint: ein ↑leichtgewichtiger Prozess oder ↑federgewichtiger Prozess teilt sich mit anderen seiner Art implizit denselben globalen Adressraum, er überlappt sich damit in Teilen seines eigenen Adressraums mit Teilen der eigenen Adressräume gleichgestellter (leicht-/federgewichtiger) Prozesse in diesem globalen Adressraum. Der globale Adressraum unterliegt dem ↑Speicherschutz, hier konkret auf Grundlage einer MMU oder MPU. Wesentlicher Aspekt dabei ist die ↑Adressraumisolation (im Gegensatz zum ↑Einzeladressraum), durch die ein Ausbrechen aus dem eigenen und damit gegebenenfalls Einbrechen in einen fremden Adressraum für jeden (schwergewichtigen) Prozess unterbunden wird.

Diese Isolation kann stark oder schwach ausgelegt sein. Ersterer Fall meint, dass der Prozessadressraum als *strikt privat* gilt, und zwar bezogen auf jedes im Rechensystem zur Ausführung kommende ↑Programm, nämlich ↑Maschinenprogramm und Betriebssystem (wie bei ↑macOS bis Leopard, 32-Bit). Demgegenüber ist bei letzterem Fall der Prozessadressraum *teilprivat* (wie bei ↑Windows NT und ↑Linux, 32/64-Bit, sowie macOS ab Snow Leopard, 64-Bit): agiert der Prozess im Maschinenprogramm (↑*user mode*), ist sein Adressraum beschränkt durch die Berechnungsvorschrift nur dieses Programms; agiert der Prozess im Betriebssystem (↑*system mode*), erweitert sich sein Adressraum und ist zusätzlich beschränkt um die Berechnungsvorschrift des Programms „Betriebssystem“. Bei dieser Variante ist also der durch ein Maschinenprogramm belegte ↑Speicherbereich (ggf. auch mehrere davon) inhärenter Bestandteil des für das Betriebssystem definierten Adressraums. Technisch wird dies erreicht, indem der durch die ↑Adressbreite insgesamt mögliche Adressraum partitioniert wird. Gebräuchlich ist eine gleichmäßige (32-Bit Windows NT: 2 GiB jeweils für Benutzer- und Systemmodus) oder ungleichmäßige (32-Bit Windows NT *Enterprise Edition* und Linux: 3 GiB für Benutzer- und 1 GiB für Systemmodus) Aufteilung. Die Grenze zwischen beiden Adressraumpartitionen entspricht einem ↑Schutzgatter.

Im Gegensatz dazu steht bei starker Isolation jedem Programm (also Maschinenprogrammen und Betriebssystem) jeweils ein Adressraum mit voller Adressbreite (virtuell) zur Verfügung. Neben eines größeren Adressraums ist der Vorteil dieses Ansatz gegenüber der schwachen Isolation, dass im Betriebssystem verborgene Programmierfehler die Integrität der Maschinenprogramme nicht verletzen können. Andererseits gestalten sich Zugriffe des Betriebssystems auf den Speicherbereich eines Maschinenprogramms, etwa als Folge von einem ↑Systemaufruf, schwieriger und mit weitaus höherer ↑Latenz: der Speicherbereich, auf den zugegriffen werden soll, muss explizit in den Betriebssystemadressraum eingeblendet und nach erfolgtem Zugriff wieder ausgeblendet werden. Bei schwacher Isolation dagegen ist die mit einem Systemaufruf übergebene Adresse auch im Betriebssystemdressraum gültig und direkt verwendbar. Jedoch muss das Betriebssystem hier vor Verwendung einer solchen Adresse eine *Integritätsprüfung* durchführen, um unautorisierten Zugriffen auf den für die Adressraumpartition des Betriebssystems definierten ↑Adressbereich vorzubeugen.

Im Grunde ist diese Schutzart ein Beispiel für ↑partielle Virtualisierung, indem nämlich vor allem Adressen beziehungsweise Adressbereiche virtualisiert werden und nicht notwendigerweise auch gleich noch der ↑Hauptspeicher. Adressen sind dadurch nicht mehr systemweit

eindeutig (im Gegensatz zum Einzeladressraum), sondern nur noch bedeutsam in einem bestimmten ↑Bezugssystem, das als ↑virtueller Adressraum definiert ist. Jeder schwergewichtige Prozess erhält einen eigenen virtuellen Adressraum, den er nicht aus eigenen Kräften nach Belieben ausdehnen kann und der ihn daher auch immun gegenüber unautorisierten Zugriffen anderer Prozesse macht (↑*protection domain*) — vorausgesetzt, das Betriebssystem genügt seiner Spezifikation und funktioniert damit in korrekter Weise.

mehradrig (en.) ↑*multi-core*. In der Natur einer ↑CPU liegende Eigenart, aus der sich bestimmte Fähigkeiten zur ↑Parallelverarbeitung oder Anfälligkeiten für ↑Wettlaufsituationen ergeben (in Anlehnung an den Duden).

Mehraufgabenbetrieb (en.) ↑*multitasking*. Art von ↑Mehrprogrammbetrieb, wobei die durch ein ↑Programm jeweils beschriebene ↑Aufgabe im Vordergrund steht. Jedes Programm kann entweder nur eine oder mehrere Aufgaben beschreiben. Sind es mehrere Aufgaben, können diese von derselben oder verschiedenen Arten (Typen) sein. Zudem steht die wirkliche Ausprägung einer Aufgabe nicht im Vordergrund: eine Aufgabe kann einen eigenständigen ↑Handlungsstrang darstellen oder teilt sich einen solchen mit anderen Aufgaben.

Mehrbenutzerbetrieb (en.) ↑*multi-user mode*. Variante von ↑Dialogbetrieb, bei der das ↑Betriebssystem zu einem Zeitpunkt mindestens einem Teilnehmer den Rechenbetrieb ermöglicht (Gegenteil von ↑Einbenutzerbetrieb). Für gewöhnlich geht diese Variante mit ↑Mehrprogrammbetrieb einher, etwa indem jedem Teilnehmer ein eigener ↑Kommandointerpreter zur Dialogführung bereitgestellt wird. Allerdings kann allen Teilnehmern auch ein- und dasselbe ↑Exemplar eines solchen Interpreters zugeordnet sein, der dann als ↑nichtsequentielles Programm ausgelegt eben alle Teilnehmer bedient. Eine solche Dialogführung im ↑Einprogrammbetrieb unterstützt zwar mehrere Teilnehmer (↑Teilnehmerbetrieb, ↑Teilhaberbetrieb), kommt jedoch nicht dem ↑Schutz teilnehmerspezifischer ↑Daten entgegen.

Mehrebenenmaschine (en.) ↑*multi-level machine*. Gegliederter, durch eine ↑funktionale Hierarchie geprägter Aufbau eines ↑Rechners in Form von mehreren, logisch übereinander liegenden Ebenen. Jede dieser Ebene ist definiert durch eine ↑virtuelle Maschine, hier auch zu verstehen als ↑abstrakte Maschine, deren Merkmale und Funktionen der Verwirklichung einer der Semantik bestimmter Operationen betreffenden und in der Hierarchie höher gestellten Ebene dient. Die Operationen können durch Anweisungen eines ↑Programms in Software formuliert oder durch Komponenten elektronischer Bauteile in Hardware repräsentiert sein. Mit jeder Ebene steht ein ↑Prozessor zur Durchführung dieser Operationen bereit, wodurch die ↑semantische Lücke zwischen einer höher und einer tiefer gelegenen Ebene überbrückt beziehungsweise aufgelöst wird. Dabei müssen beide Ebenen nicht zwingend aneinandergrenzen, das heißt, der Prozessor vermag in seiner Abbildungsfunktion durchaus mehrere Ebenen zusammenfassend zu betrachten.

Die Anzahl und Bedeutung der Ebenen reflektiert einen bestimmten Technologiestand (*state of the art*) eines ↑Rechensystem. Beides hat sich im Laufe der Zeit seit des ersten ↑Rechners, der durch ein im ↑Hauptspeicher zur Ausführung bereitstehendes und binär kodiertes ↑Programm betrieben wurde (↑EDVAC), nach und nach den jeweiligen technologischen Gegebenheiten angepasst. Nachfolgende Aufstellung zeigt die für heutige (2025) Rechensysteme typischen Ebenen sowie deren Anordnung zueinander, wobei auf den (nicht nur) für ↑Systemprogrammierung relevanten Ebenen 1–3 besonderes Augenwerk liegt:

Ebene	Bezeichnung	Abbildung	Prozessor
5	Programmiersprachenebene	↑Übersetzung	↑Kompilierer
4	Assembliersprachenebene	↑Übersetzung	↑Assemblierer, ↑Binder
3	↑Maschinenprogrammebene	↑Teilinterpretation	↑Betriebssystem
2	↑Befehlssatzebene	↑Interpretation	↑CPU
1	↑Mikroarchitekturebene	↑Interpretation	↑Mikroprogrammsteuerwerk
0	digitale Logikebene	Ausführung	↑Gatter: $\wedge$, $\vee$, $\neg$, $\oplus$

Anfänglich umfasste ein Rechner (EDVAC) nur die Ebenen 0–2, wobei Ebene 0 zunächst aus Röhren-, danach (um 1953) aus Transistortechnik bestand und erst später (um 1956) der dann in gleicher Technik aufgebaute Programmspeicher hinzukam. Im weiteren Verlauf der Rechnerentwicklung kamen Ebene 4 (um 1950, ↑IBM 701), Ebene 3 (um 1955, ↑FMS) und Ebene 5 (um 1957, ↑FORTRAN) hinzu. Das mit Ebene 1 eingeführte Paradigma von ↑Mikroprogrammen war zwar bereits früh (um 1951) ein Begriff, fand jedoch erst mit ↑IBM System/360 (um 1964) weite Verbreitung.
Ebene 2 definiert die Grenze zwischen Hardware (Ebenen 0–2) und Software (Ebenen 3–5). Ebene 1 ist geprägt durch Mikroprogramme, die vor allem die architektonischen Merkmale des Rechners bestimmen. Für gewöhnlich sind diese Merkmale durchgängig (oberhalb von Ebene 1) sichtbar und beeinflussen teils auch wesentlich den Aufbau und die Struktur der Programme. Demgegenüber werden die durch Ebene 2 hinzugekommenen „syntaktischen Merkmale“, die nämlich in dem ↑Programmiermodell verankert sind (d.h., ↑Befehlssatz, ↑Prozessorregister), durch Ebene 5 verdeckt. Die Mikroprogramme bilden die sogenannte ↑Firmware — die hier jedoch nicht mit dem (direkt auf Ebene 2 aufsetzenden) ↑BIOS zu verwechseln ist. Nicht jeder Rechner ist allerdings mikroprogrammiert, weshalb Ebene 1 auch komplett in Hardware ausgelegt sein kann.
Ein anderer grundlegender Bruch zeigt sich zwischen Ebenen 3 und 4: Ebenen 4–5 (ggf. auch höher) sind ↑Anwendungsprogrammen vorbehalten, wohingegen die tieferliegenden Ebenen das Gebiet der ↑Systemprogrammierung darstellen, insbesondere Ebenen 2–3. Zudem werden alle höheren Ebenen ab einschließlich Ebene 4 übersetzt und alle tieferen Ebenen einschließlich Ebene 3 interpretiert. Neben diesen unterschiedlichen Methoden, nämlich wie eine Ebene letztlich getragen wird, definiert auch die Art der Programmrepräsentation eine deutliche Trennlinie. Die ↑Maschinensprache der Ebenen 0–3 ist numerisch, die der Ebenen 4–5 ist symbolisch. So ist jede Art von Software, die zur Ausführung gebracht werden soll, letztlich ein in ↑Maschinenkode der Ebene 2 repräsentiertes Programm.
Die Ebenen 4–5 haben ihrer Zweck für die Entwicklung von Software beziehungsweise Programmen, sie existieren für ein gegebenes Programm wirklich (real) nur zur Entwicklungszeit. Dagegen sind die Ebenen 0–3 für dieses Programm physisch (real) immer präsent, sie sind die eigentlichen operativen Komponenten zur ↑Laufzeit jeglicher Sorte von Programmen. Oberhalb von Ebene 5 befinden sich in dieser logischen Konstruktion von scheinbaren (virtuellen) und wirklichen (realen) Maschinen schließlich die ↑Anwendungen. Hier gibt es insbesondere auch solche Anwendungen, die der (Weiter-)Entwicklung der Ebenen 0–5 zukünftiger Rechensysteme dienen.

mehrfädiger Zusteller (en.) ↑*multi-threaded server*. Bezeichnung für einen ↑Zusteller, dessen Funktion durch ein ↑nichtsequentielles Programm definiert ist.

Mehrfädigkeit (en.) ↑*multithreading*. Fähigkeit einer (realen/virtuellen) Maschine mehr als einen ↑Faden zugleich durch ↑Parallelverarbeitung ausführen zu können. Grundlage bildet ein ↑nichtsequentielles Programm, das die einem jeden Faden zukommende ↑Aufgabe beschreibt.

Mehrkernprozessor (en.) ↑*multi-core processor*. Bezeichnung für eine ↑CPU, die aus mehr als einen ↑Rechenkern besteht. Damit ist die CPU ↑mehradrig ausgelegt und ermöglicht dem

↑Betriebssystem dadurch, ↑Prozesse echt gleichzeitig auf demselben ↑Halbleiterbaustein (genauer: ↑Halbleiterplättchen) stattfinden zu lassen.

Mehrprogrammbetrieb (en.) ↑*multiprogramming*, ↑*multitasking*. Bezeichnung für die ↑Betriebsart eines Rechensystems, bei der mehr als ein ↑Programm zugleich im ↑Arbeitsspeicher zur Ausführung zur Verfügung stehen. Die Programme werden im Grunde solange ausgeführt, bis sie von selbst die Kontrolle über den ↑Prozessor abgeben (*run to completion*). Dabei erfolgt die Kontrollübergabe in solchen Situationen, die die weitere Benutzung des Prozessors logisch bedingt im Programm nicht mehr erfordert: nämlich wenn ein ↑wiederverwendbares Betriebsmittel oder ein ↑konsumierbares Betriebsmittel (inkl. Ein-/Ausgabe) zum weiteren Fortschritt benötigt wird, aber nicht zur Verfügung steht, oder wenn die Programmausführung endet. Mehrfachnutzung des Prozessors wird zwar unterstützt, jedoch nur in räumlicher Hinsicht und in kooperativer Art und Weise eines Programmablaufs. Erst die Erweiterung um ↑Simultanverarbeitung sorgt für die Mehrfachnutzung in Raum und Zeit.

Mehrprozessbetrieb (en.) ↑*multitasking*. Art von ↑Mehraufgabenbetrieb, bei der jede zu bearbeitende ↑Aufgabe durch eine eigene ↑Prozessinkarnation repräsentiert ist.

mehrseitige Synchronisation siehe ↑multilaterale Synchronisation.

Mehrstromstapelmonitor (en.) ↑*multi-stream batch monitor*. Bezeichnung für einen ↑Stapelmonitor, der die gestapelten Aufträge als mehrfachen Verarbeitungsstrom ausführt. Jeder Verarbeitungsstrom entspricht einem ↑Maschinenprogramm, wovon mehrere verschiedene zugleich im ↑Hauptspeicher residieren (↑*multiprogramming*) und jedes einzelne zur ↑Laufzeit eine bestimmte Einzelarbeit (↑*job*) leistet. Sobald ein Maschinenprogramm aus Mangel an ↑Betriebsmittel nicht weiter ausgeführt werden kann, schaltet der Stapelmonitor um zu einen anderen Verarbeitungsstrom und nimmt damit die Ausführung eines anderen Maschinenprogramms auf: kooperative und dynamische ↑Ablaufplanung. Der Ausführungswechsel zwischen den verschiedenen Maschinenprogrammen geschieht implizit mit einer Betriebsmittelanforderung, wenn diese nämlich in dem Moment, wo sie von dem ↑Prozess gestellt wird, nicht erfüllbar ist. Die zusätzliche Last für das ↑Betriebssystem, im Vergleich zum ↑Einzelstromstapelmonitor, besteht in der ↑Betriebsmittelkontrolle und dem ↑Speicherschutz in Bezug auf den einzelnen (in seinem jeweiligen Maschinenprogramm stattfindenden) Prozess beziehungsweise ↑Prozessadressraum.

memory (dt.) ↑Speicher.

memory allocation (dt.) ↑Speicherzuteilung.

memory area (dt.) ↑Speicherbereich.

memory attribute (dt.) ↑Speicherattribut.

memory barrier (dt.) ↑Speicherbarriere.

memory cell (dt.) ↑Speicherzelle.

memory coherency (dt.) ↑Speicherkohärenz.

memory consistency (dt.) ↑Speicherkonsistenz.

memory controller (dt.) ↑Speichersteuerung.

memory equipment (dt.) ↑Speicherausstattung.

memory footprint (dt.) ↑Speichergrundfläche.

memory hierarchy (dt.) ↑Speicherhierarchie.

memory item (dt.) ↑Speicherpartie, Speichergegenstand.

memory location (dt.) ↑Speicherstelle.

memory management (dt.) ↑Speicherverwaltung.

memory management unit (dt.) ↑Speicherverwaltungseinheit.

memory map (dt.) ↑Speicherplan.

memory model (dt.) ↑Speichermodell.

memory protection (dt.) ↑Speicherschutz.

memory pyramid (dt.) ↑Speicherpyramide.

memory tile (dt.) ↑Speicherkachel.

memory virtualisation (dt.) ↑Speichervirtualisierung.

memory word (dt.) ↑Speicherwort.

memory-mapped (dt.) ↑speicherabgebildet.

***memory-mapped* I/O** (dt.) ↑speicherabgebildete Ein-/Ausgabe.

memory-protection unit (dt.) ↑Speicherschutzeinheit, Abkürzung ↑MPU.

Mesa Programmiersprache, imperativ, strenge statische Typung (1976); ↑Algol ähnlich. Bekannt für ihr ↑Monitorkonzept (↑*signal and continue*). Systemprogrammiersprache für ↑Pilot, wobei die Sprache selbst für einen Großteil ihrer Laufzeitunterstützung von Pilot abhängig ist.

message (dt.) ↑Nachricht, Botschaft.

message passing (dt.) ↑Nachrichtenversenden, Botschaftenaustausch.

microarchitecture level (dt.) ↑Mikroarchitekturebene.

microcode siehe (en.) ↑*microprogram*.

microcontroller (dt.) ↑Mikrokontroller.

microinstruction (dt.) ↑Mikrobefehl.

microprogram (dt.) ↑Mikroprogramm.

microprogram control unit (dt.) ↑Mikroprogrammsteuerwerk.

Mikroarchitekturebene (en.) ↑*microarchitecture level*. Bezeichnung für die den ↑Befehlssatz und das ↑Programmiermodell eines ↑Prozessors implementierende Schicht in einem als ↑Mehrebenenmaschine gekennzeichneten ↑Rechensystem. Diese Ebene ist definiert durch die funktionellen Bestandteile des Prozessors und der Art, wie diese „Blöcke“ untereinander über ↑Daten- und ↑Signalwege verbunden sind beziehungsweise ihre Zusammenarbeit organisiert ist, um die für den Prozessor vorgesehene ↑ISA zu verwirklichen. Dem entsprechend geschieht die Beschreibung dieser Ebene für gewöhnlich mit ↑Blockdiagrammen, die gegebenenfalls durch ↑Datenflussdiagramme ergänzt werden.
Ein wesentlicher Blocktyp dieser Ebene ist die Ausführungseinheit (*execution unit*), wovon es für gewöhnlich verschiedene Ausprägungen gibt, etwa zur Ganzzahl- oder Fließkommaarithmetik, zum Laden und Speichern von Daten oder zur Sprungvorhersage. Ebenso verankert diese Ebene die Entwurfsentscheidung zur Integration von (einfachen) ↑Peripheriegeräten oder Steuerbausteinen für den ↑Hauptspeicher (*memory controller*). Die dabei geltenden Entwurfsrandbedingungen sind weniger, einen bestimmten Leistungsgrad zu erreichen, sondern betreffen vielmehr Fläche/Kosten des ↑Halbleiterbausteins, sein Energieverbrauch und

die damit einhergehende Hitzeabgabe, der Logikumfang, die Verbindungsfähigkeit, die Herstellbarkeit und, nicht zuletzt, leichte Testbarkeit und ↑Fehlerbeseitigung.
In technischer Hinsicht ist der Aufbau dieser Ebene festverdrahtet (typisch für ↑RISC), durch ein ↑Mikroprogramm bestimmt (typisch für ↑CISC) oder eine Kombination von beiden. Die mikroprogrammierte Umsetzung erlaubt in vergleichsweise einfacher Weise die Wiederverwendung derselben Blöcke zur Ausführung verschiedener ISA-Varianten. Unterstützt der Prozessor die Änderung des Mikroprogramms gar im laufenden Betrieb (z.B. ↑PDP 11/40e), kann dadurch beispielsweise ein ↑Prozesswechsel mit einem Wechsel der ISA einhergehen — das heißt, einem Prozess sein eigener spezieller ↑Befehlssatz bereitgestellt werden.

Mikrobefehl (en.) ↑*microinstruction*. Eine Anweisung, auch als ↑Mikroinstruktion bezeichnet, mit der auf sehr grundlegender Ebene die Schaltkreise des ↑Prozessors gesteuert und dadurch seine Operationen ausgelöst werden. Dazu ist das Verhalten dieser Schaltkeise beschrieben durch voneinander unabhängige, in einem Takt durchführbare *Mikrooperationen*. Eine Mikroinstruktion umfasst sodann eine bestimmte Gruppe von Mikrooperationen, die im selben Takt stattfinden. Je breiter eine solche Instruktion ist, je mehr Bits sie umfasst, umso mehr Mikrooperationen können damit gleichzeitig ausgelöst werden. Dabei ist die ↑Wortbreite einer Mikroinstruktion nicht notwendigerweise Vielfaches der Größe eines ↑Bytes, im Gegensatz zum ↑Maschinenbefehl.
Jede dieser Anweisungen weist ein einheitliches Format auf, das aus Adress-, Auswahl- und Steuerteil besteht. Das Adressteil bestimmt die nächste auszuführende Mikroinstruktion, es enthält eine ↑Adresse in den *Mikroprogrammspeicher*. Während der Ausführung einer Mikroinstruktion wird diese Folgeadresse in den *Mikrobefehlszähler* übernommen. Der Inhalt dieses ↑Registers lässt sich bedingt verändern, und zwar über die im Auswahlteil zusammengefassten *Bitschalter*. Jeder dieser Schalter selektiert ein zum ↑Mikroprogrammsteuerwerk gehörendes Register, dessen Inhalt gegebenenfalls zur Änderung des Mikrobefehlszählerwerts herangezogen wird. Darüber lassen sich bedingte Anweisungen beziehungsweise Verzweigungen zum Ausdruck bringen. Das Steuerteil schließlich bestimmt die für einen bestimmten Abschnitt zur Ausführung eines Maschinenbefehls durchzuführenden Mikrooperationen. Über die hier ebenfalls enthaltenen Bitschalter werden die einzelnen Rechen- und Speichereinheiten des Prozessors angesteuert, sowie Übertragungswege zwischen diesen Einheiten und Registern des Mikroprogrammsteuerwerks freigeschaltet.
Jeder einzelne Bitschalter entspricht damit einer diskreten Schaltanweisung, auch als *Pikobefehl* bezeichnet. Diese Schaltanweisungen gelten zudem als unabhängig, weshalb sie auch durch Einzelbitsteuerung zur Wirkung gebracht werden können: eine derart aufgebaute Mikroinstruktion heißt daher *horizontal kodiert*. Ist solch eine unabhängige Einzelbitsteuerung nicht möglich oder geboten, etwa wegen einer bestimmten (ggf. Zwischentakte erfordernden) zeitlichen Abfolge der Mikrooperationen beziehungsweise da ein Multiplexer für die Aktivierung bestimmter Funktionseinheiten des Prozessors sorgt, wird die Schaltanweisung durch einen mehrstelligen Binärkode in einer ↑Bitleiste repräsentiert. Jeder dieser Binärkodes entspricht dann einem sogenannten *Nanobefehl*, die Mikroinstruktion ist in dem Fall *vertikal kodiert*. Darüber hinaus können sowohl horizontal als auch vertikal kodierte Abschnitte in ein und derselben Mikroinstruktion definiert sein. Eine solche Mikroinstruktion wird dann auch als *diagonal kodiert* bezeichnet. Dadurch können in einem *Mikrobefehlszyklus* zugleich mehrere Piko- und Nanobefehle aktiv sein.

Mikroinstruktion siehe ↑Mikrobefehl.

Mikrokontroller (en.) ↑*microcontroller*. Ganzheit von ↑Steuereinheit und ↑CPU. Für gewöhnlich ein in seinen wesentlichen Funktionen komplettes ↑Rechensystem für einen besonderen ↑Anwendungsfall integriert in ein und demselben ↑Halbleiterbaustein, die ↑MCU.

Mikroprogramm (en.) ↑*microcode*, ↑*microprogram*. Bezeichnung für ein ↑Programm, dessen einzelne Anweisungen ↑Mikrobefehle darstellen. Dieses Programm beschreibt grundlegende *architektonische Merkmale*, die der ↑Prozessor, in dem es als Firmware — für gewöhnlich

nichtflüchtig, aber durchaus auch änderbar — gespeichert vorliegt, aufweisen soll. Das Programm legt wesentliche funktionale Eigenschaften des Prozessors fest und schafft damit unter anderem die Grundlage für die Ausprägung als ↑CISC — ein ↑RISC ist für gewöhnlich nicht mikroprogrammiert, da seine vergleichsweise einfachen ↑Maschinenbefehle direkt die Funktionseinheiten des Prozessors ansprechen können und dies nicht algorithmisch in Form eines Programms geschehen muss. Zwischen der zugrunde liegenden Hardware und der durch die ↑Befehlssatzebene sichtbaren Architektur des Prozessors besteht dadurch kein feststehender Zusammenhang.

Mikroprogrammsteuerwerk (en.) ↑*microprogram control unit*. Bezeichnung für das ↑Steuerwerk in einem ↑Prozessor, das auf Grundlage eines ↑Mikroprogramms die Funktionsweise eben dieses Prozessors festlegt, die Ausführung eines durch die ↑Befehlssatzebene definierten ↑Maschinenbefehls steuert und den im ↑Programmiermodell sichtbaren Prozessorzustand fortschreibt.

minor device number (dt.) ↑Nebengerätenummer.

mitlaufende Planung (en.) ↑*on-line scheduling*. Modell der ↑Ablaufplanung, die ört- und zeitlich gekoppelt mit den einzuplanenden ↑Prozessen geschieht: sie findet im Hintergrund mitlaufend statt und erzeugt einen gemeinhin dynamischen ↑Ablaufplan, der zu einem gegenwärtigen Zeitpunkt (d.h., zeitnah zur Aufstellung) vom ↑Betriebssystem abgearbeitet wird (↑*dynamic scheduling*). Der ↑Planer ist fester Bestandteil von oder aufgesetzter Teil (↑*user-level scheduling*) bei einem Betriebssystem (örtlich gekoppelt). Anders als ↑vorlaufende Planung wird der Ablaufplan in direkter Verbindung zu den jeweils darauf stehenden Prozessen aktualisiert und fortgeschrieben (zeitlich gekoppelt).
Diese Form von ↑Planung ist bei fehlendem ↑Vorwissen zu den einzuplanenden Prozessen obligatorisch, dass heißt, wenn ↑Prozessgrößen oder die zu erwartende Prozessanzahl unbekannt sind. Aber auch trotz Vorhandensein solchen Wissens kann es geboten sein, ein mitlaufendes Verfahren einer möglichen vorlaufenden Variante vorzuziehen, nämlich wenn Flexibilität im Vordergrund steht und der Dynamik im ↑Rechensystem Rechnung zu tragen ist. Letzterer Aspekt ist typisches Merkmal eines ↑Universalbetriebssystems.

mittelfristige Planung (en.) ↑*medium-term scheduling*. ↑Ablaufplanung auf mittlere Sicht, typischerweise im Millisekunden- oder Sekundenbereich. Eine in der Speicherverwaltung verankerte ↑Systemfunktion, die festlegt, welche Bereiche im ↑Hauptspeicher zu gegebener Zeit für einen ↑Prozess freizumachen oder zu belegen sind (↑*swapping*). VL 12.3

mittlere Zugriffszeit (en.) ↑*average access time*.. VL 12.3

MLFQ Abkürzung für (en.) *multi-level feedback queue*. Bezeichnung einer auf *Rückkopplung* basierenden Strategie für die auf ↑Bevorrechtigung basierende ↑mitlaufende Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor. Diese Strategie vereint ↑MLQ mit ↑FB, wobei die verschiedenen ↑Warteschlangen aber nicht nach Aufgabentypen unterscheiden, sondern nach der Länge der für eine Gruppe von ↑Prozessen geltenden ↑Zeitscheibe. Auf oberster Ebene (höchster ↑Rang) liegt eine Warteschlange von Prozessen mit der kürzesten im ↑Betriebssystem definierten Zeitscheibe. Absteigend (niedrigere Ränge) nehmen die Zeitscheiben ebenenweise an Länge zu, bis auf unterster Ebene (niedrigster Rang) angelangt die Zeitscheibenlänge konstant bleibt und die sich dort befindenden Prozesse reihum bedient werden.
Für gewöhnlich werden eintreffende Prozesse entsprechend ihrer ↑Ankunftszeit in einer dieser Warteschlangen eingereiht und gemäß ↑FCFS bedient. Ein Prozess, der erstmalig eingeplant wird, nimmt seinen Platz immer in der obersten Warteschlange ein. Bei der ↑Einlastung erhält der betreffende Prozess sodann den Prozessor für die Dauer der mit seiner Warteschlange geltenden Zeitscheibe zugeteilt. Beendet der Prozess seinen ↑Rechenstoß vor Ablauf der Zeitscheibe, gibt er also „freiwillig“ den Prozessor ab, etwa weil er seinen ↑Ein-/Ausgabestoß abwarten muss, bleibt er auf dieser Ebene und wird bei erneuter Bereitstellung wieder in dieselbe Warteschlage eingereiht. Läuft für den Prozess jedoch die Zeitscheibe ab, wird er

einer Warteschlange tieferer Ebene (längerer Zeitscheibe) hinzugefügt und zusehends vom Prozessor verdrängt. Die Warteschlange dafür kann implizit immer die der nächst tieferen Ebene sein oder mit jeder Ebene ist explizit vermerkt, in welche Warteschlange der Prozess herabzustufen ist.
Ist die Warteschlange auf der Ebene des Prozesses, dessen Zeitscheibe ablief, noch gefüllt, wird dem nächsten darin in Ankunftszeitreihenfolge (FCFS) der Prozessor zugeteilt. Anderenfalls geschieht absteigend ein Ebenenwechsel, bis eine nichtleere Warteschlange gefunden und dort der Prozess für die Prozessorzuteilung entnommen wird (FCFS). Auf der untersten Ebene angelangt werden alle sich dort befindenden Prozesse nach ↑RR bedient. Bleibt die Suche nach laufbereiten Prozessen allerdings erfolglos, kommt es zum ↑Leerlauf.
Vorstellbar und dem Prinzip einer stetig zunehmenden Zeitscheibenlänge beim Ebenenabstieg folgend wäre auch eine unendlich große Zeitscheibe auf unterster Ebene, mit der die dort schließlich angelangten Prozesse nicht mehr zugunsten anderer Prozesse dieser Ebene vom Prozessor verdrängt werden (↑*run to completion*). Innerhalb jeder Ebene sind die Prozesse jedoch gleichrangig. Dies gilt für alle Ebenen und bedeutet insbesondere auch, gleichrangigen Prozessen Fortschritt zu garantieren und dazu jedem von ihnen dieselbe ebenenbezogene Zeitscheibe zuzugestehen (↑*fairness*). Folglich werden auf unterster Ebene entsprechend der dort geltenden festen maximalen Zeitscheibe die Prozesse reihum (RR) bedient.
Das Verfahren belohnt Prozesse, die den Prozessor vor Ablauf ihrer Zeitscheibe zugunsten eines anderen Prozesses (derselben Ebene) abgeben und bestraft Prozesse, die dies nicht tun, das heißt, den Prozessor für sich behalten und mit dem Zeitscheibenablauf erst zur Prozessorabgabe gezwungen werden müssen (*penalization*). Mit anderen Worten: rechenintensive (↑CPU *bound*) Prozesse werden gegenüber ein-/ausgabeintensiven (d.h., interaktive, ↑I/O *bound*) Prozessen zurückgestuft. Ob ein Prozess rechen- oder ein-/ausgabeintensiv ist, bestimmt letztlich das ↑Programm, das den Prozess definiert, in Abhängigkeit von den zu verarbeitenden ↑Daten. Für gewöhnlich ändert sich die ↑Prozesslokalität über die Zeit, je nachdem, welchen Pfad der Prozess durch sein Programm nimmt und wie lange er etwa in Programmschleifen verweilt. Somit kann der Prozess in einer Phase rechen- und in einer anderen Phase ein-/ausgabeintensiv agieren und umgekehrt: ein bis zur untersten Ebene abgestufter rechenintensiver Prozess muss somit die Chance zum Wiederaufstieg erhalten, wenn er in eine ein-/ausgabeintensive Phase eintritt.
Um Prozesse wieder hochzustufen, wird deren ↑Alterung in Betracht gezogen. Hierzu reiht das Verfahren einen Prozess, der seine jüngsten Rechenstöße vor Ablauf seiner Zeitscheibe beendet hatte, bei seiner nächsten Bereitstellung wieder weiter oben in eine Warteschlange höherer Ebene ein. In welche Warteschlange der Prozess hochgestuft werden soll ist entweder explizit mit jeder Ebene vermerkt oder ergibt sich aus der Mittelung der letzten N Rechenstoßlängen des Prozesses nach oben gerundet zur nächsten Zeitscheibengröße. Darüberhinaus kann die Hochstufung präemptiv oder nichtpräemptiv wirken, das heißt:

(a) dem zuvor einer Warteschlange höherer Ebene entnommenen und jetzt stattfindenden Prozess einen Platz in dieser Warteschlange zuweisen und den Prozessor zu Gunsten des hochgestuften Prozesses entziehen oder

(b) dem hochgestuften Prozess lediglich eine Warteschlange höherer Ebene zuweisen, auch wenn diese Ebene im Rang höher liegt als die der Warteschlange, der der stattfindende Prozess zuvor entnommen wurde.

Im letzten Fall der nichtpräemptiven Alterung erhält ein „verjüngter“ (d.h., hochgestufter) Prozess frühestens dann den Prozessor, wenn der stattfindende Prozess vor oder bei Ablauf seiner Zeitscheibe den Prozessor abgibt. In dem Fall schreitet Verdrängung wie im Verfahren üblich „von oben nach unten“ voran, wohingegen im anderen Fall der präemptiven Alterung Verdrängung sich zusätzlich auch „von unten nach oben“ ziehend auswirken kann.
Dieses mehrstufige Verfahren approximiert ↑SPN, benötigt dazu jedoch kein ↑Vorwissen über die zu erwartende Länge des nächsten Rechenstosses eines Prozesses. Stattdessen „beobachtet“ das Verfahren die Prozessverläufe lediglich und sorgt durch Bestrafung und Belohnung

für eine geeignete Rangordnung der Prozesse untereinander. Dieser Ansatz wurde erstmals mit ↑CTSS umgesetzt, später in ↑Multics übernommen, für ↑Solaris stark verfeinert (60 Warteschlangen, Zeitscheiben von 20ms bis einige 100ms, Alterungskontrolle im Sekundenrhythmus) und findet heute in abgewandelter Form Verwendung in ↑Windows NT und ↑Mac OS.

MLQ Abkürzung für (en.) *multi-level queue*. Bezeichnung für eine bestimmte Form der Organisation einer ↑Warteschlange wie auch Strategie zur ↑Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor. Dabei ist jede Aufgabe von einem bestimmten *Typ*, der für sie zutreffend geglaubte Eigenschaften zusammenfasst. Solche Eigenschaften beziehen sich beispielsweise auf den Vorgang (Editieren, Übersetzen, Rechnen), das Profil (↑CPU *bound*, ↑I/O *bound*), die Betriebsart (↑*real-time mode*, ↑*conversational mode*, ↑*batch mode*) die Domäne (Benutzer, Dienste, System) oder gar die Statusgruppe (Professoren, Mittelbau, Studierende) einer jeweiligen Aufgabe. Für jede Kategorie gibt es eine eigene Warteschlange, versehen mit einer bestimmten ↑Priorität, die jeder Warteschlange einer bestimmten *Prioritätsebene* zuordnet und dadurch den relativen ↑Rang in der Aufgabenbearbeitung festlegt. Die Einordnung in eine der von dem System jeweils angebotenen Kategorien geschieht spätestens bei Erstellung der betreffenden Aufgabe, das heißt, wenn die Aufgabe ins System eingespeist und diesem damit bekannt gegeben wird.
Zur Bearbeitung der Aufgaben ist dann ein zweistufiger Plan des Vorgehens vorgesehen, der für jede Warteschlange eine eigene, möglicherweise unterschiedliche *lokale Strategie* verfolgt und für den Wechsel zwischen den Warteschlangen eine bestimmte *globale Strategie* in Betracht zieht. Als lokale Strategie kommt nahezu jeder andere ↑Einplanungsalgorithmus in Frage. Beispiele dafür könnten sein ↑EDF, ↑SRTF, ↑HRRN und ↑FCFS, die dann eventuell auch in dieser Reihenfolge in der Hierarchie von oben (hohe Priorität) nach unten (niedrige Priorität) angeordnet sind. Die globale Strategie kann dann *statisch* oder *dynamisch* sein. Im statischen Fall werden die Warteschlangen von oben nach unten abgearbeitet und der Wechsel zur nächst tieferen Ebene geschieht nur, wenn oberhalb dieser keine weitere Aufgabe zur Bearbeitung ansteht. Damit besteht die Gefahr des ↑Verhungerns von Aufgaben in tiefer liegenden Warteschlangen — je nach lokaler Strategie kann diese Gefahr aber auch bereits Aufgaben derselben, höheren Ebene betreffen. Der dynamische Fall kann nach einem ↑Zeitteilverfahren arbeiten, wodurch dann ein Rundlauf (↑*round robin*) in vertikaler Hinsicht zustande kommt, durch den jeder Ebene für eine bestimmte Zeit den Prozessor zur Aufgabenbearbeitung zugeteilt bekommt. Ein solches Verfahren kann beispielsweise gemäß ↑RR vorgehen oder es teilt den verschiedenen Ebenen unterschiedliche ↑Rechenzeiten zu, etwa: 40 % Systemprozesse, 40 % Dialogprozesse, 20 % Stapelprozesse. Eine andere Form des ↑präemptiven Wechsels zwischen den Ebene kann allein nur durch die der Ebene einer eintreffenden Aufgabe zugeordneten Priorität gegeben sein. Trifft eine Aufgabe höherer Prioritätsebene zur Bearbeitung ein, während eine Aufgabe niedrigerer Prioritätsebene in Bearbeitung ist, wird erstere gestartet und damit zur höheren Ebene gewechselt.
Jeder Eintrag in einer der Warteschlangen steht für eine bestimmte Aufgabe, die von einem ↑Prozess zu erledigen ist. Diese Aufgaben können demselben oder verschiedenen Prozessen zur Erledigung aufgegeben sein, auch kann jede Aufgabe selbst einer eigenen ↑Prozessinkarnation entsprechen. Letztere Variante impliziert dann immer einen ↑Prozesswechsel, wenn derselbe Prozessor mit einer Aufgabe aus einer der Warteschlangen versorgt werden soll. So ist die Zuordnung von Aufgabe zu Prozessinkarnation abhängig von der jeweils definierten lokalen oder globalen Strategie. Nur im Falle einer präemptiven Verfahrensweise für eine lokale Strategie ist die Repräsentation einer Aufgabe als Prozessinkarnation obligatorisch. Ähnliches gilt für die globale Strategie, wenn diese präemptiv arbeitet: dann ist wenigstens eine Prozessinkarnation pro Ebene (d.h., lokale Strategie) vorzusehen, um zwischen diesen Ebenen (d.h., den jeweiligen lokalen Strategien) nach Belieben umschalten zu können. Eine solche für jede Ebene dann normalerweise vorzuhaltende und als ↑Zusteller agierende Prozessinkarnation kann jedoch entfallen, wenn die diesbezügliche lokale Strategie dieser Ebene bereits eine präemptive Vorgehensweise in der Aufgabenbearbeitung verfolgt.

Exkurs Für die Implementierung des Verfahrens bietet sich ein Feld (*array*) von ↑Deskriptoren an, wobei jeder Deskriptor zumindest eine Warteschlange (Anfangs-/Endzeiger), die lokale Strategie (Prozedurzeigerfeld, pro Operation des ↑Planers einen Zeiger) und gegebenenfalls auch einen Rechenzeitanteil ($\mathbb{N}^*$) definiert. Dabei repräsentiert der erste Deskriptorfeldeintrag die höchste und der letzte die niedrigste Prioritätsebene. Die Auswahl der ersten auszuführenden Aufgabe der nächst tieferen Ebene kann dann einfach durch eine Laufanweisung (`for`-*statement*) geschehen, nämlich wenn alle Aufgaben der aktuellen Ebene bearbeitet wurden (statische globale Strategie) oder das ↑Zeitquantum dieser Ebene (dynamische globale Strategie) erschöpft ist. Zur Bereitstellung einer Aufgabe ist zunächst die zugehörige Ebene zu identifizieren (globale Strategie) und anschließend innerhalb dieser Ebene die betreffende Aufgabe auf die Warteschlange zu setzen (lokale Strategie). Im Falle einer präemptiven, dynamischen globalen Strategie wird die Bereitstellung einer Aufgabe, die einer im Vergleich zur laufenden Aufgabe höheren Ebene zugeordnet ist, für gewöhnlich einen Prozesswechsel nach sich ziehen.
Das nachfolgende Beispiel veranschaulicht den Rahmen einer solchen Lösung, wobei zunächst eine nichtpräemptive und anschließend eine präemptive Variante vorgestellt wird. Insbesondere letztere gibt eine Idee vom Aufbau eines ↑Planers zum ↑Echtzeitbetrieb, wobei die *statische Priorität* der von einem Prozess zu bearbeitenden Aufgabe dem Rang und damit der Ebene einer Warteschlange entspricht. Beiden Varianten gemeinsam ist folgende zentrale Datenstruktur zur Erfassung des jeweiligen Bestands:

```
typedef struct supply {
    heat_t list[NTIER];                    /* array of queue descriptors */
    int rest;                        /* total # of ready-to-run processes */
} supply_t;
```

Zur Beendigung eines gegebenenfalls aktiven ↑Leerlaufprozesses ist ein Hinweis zum Eingang eines laufbereiten Prozesses erforderlich. Verwendet der Planer für die ↑Bereitliste nur eine einzige Warteschlange, ergibt sich dieser Hinweis leicht durch den Zustand des damit gegebenen einzigen Kopfzeigers: Null bedeutet weiterhin Untätigkeit (vgl. auch S. 93). Bei mehreren Warteschlangen, wie im Fall hier, würde reihum die Abfrage jedes einzelnen Kopfzeigers anfallen und damit der mögliche ↑Schlafzustand der ↑CPU verhindert oder zumindest stark eingeschränkt. Daher kommt ein Zähler zum Einsatz, der Aufschluss über den aktuellen Restbestand (`rest`) bereitgestellter Prozesse gibt und wie folgt zugänglich ist:

```
inline const int *raring() {
    extern supply_t work;                        /* global MLQ descriptor */
    return &work.rest;                              /* refer to MLQ meter */
}
```

Dieser Zähler ist zu aktualisieren, wann immer ein Prozess eine der Warteschlangen hinzugefügt oder aus einer davon entfernt wird. Da die Bereitstellung von Prozessen, damit jeweils das Hinzufügen zu einer der Warteschlangen, auch in der nichtpräemptiven Variante entweder lokal aus einer ↑Unterbrechungsbehandlung heraus oder entfernt von einem anderen Prozessor oder ↑Rechenkern aus geschehen kann, also möglicherweise gleichzeitig stattfindet, muss das Aufrechnen (`reckon`) der Prozesse als ↑atomare Operation durchgeführt werden:

```
inline int reckon(int step) {
    extern supply_t work;                        /* global MLQ descriptor */
    return FAA(&work.rest, step);          /* meter ready-to-run processes */
}
```

↑FAA ist die passende Operation, um den Prozesszähler jederzeit konsistent aktualisieren zu können. Nur wenn sichergestellt ist, dass eine gleichzeitige Veränderung des Zustands der mehrstufigen Bereitsliste niemals erfolgt, kann die atomare Operation entfallen. Die Warteschlange für die Prozesse einer bestimmten Stufe ($0 \leq tier < NTIER$) beziehungsweise Priorität macht nachfolgend skizzierte Zugriffsoperation zugänglich:

```
inline heat_t *lineup(tier_t tier) {
    extern supply_t work;                        /* global MLQ descriptor */
    assert(tier < NTIER);                            /* index bound check */
    return &work.list[tier];              /* deliver local MLQ descriptor */
}
```

Jede Warteschlangenebene definiert einen Lauf (`heat`) von gleichrangigen Prozessen. Jeder dieser Läufe hat eine eigene Warteschlange, mit der gegebenenfalls eine ebenenspezifische Reihungsdisziplin der Prozesse verknüpft ist. So könnten mehrere gleichrangige Prozesse beispielsweise entsprechend ihrer ↑Ankunftszeit (↑FCFS) oder erwarteten ↑Stoßzeit (↑SPN) in der Warteschlange aufgestellt sein. Vor diesem Hintergrund bietet es sich an, den Warteschlangendeskriptor wie folgt auszulegen:

```
typedef struct heat {
    queue_t chart;                    /* available ready-to-run processes */
    void       (*quote)(queue_t *, process_t *);     /* enqueue discipline */
    process_t *(*elect)(queue_t *);                  /* dequeue discipline */
} heat_t;
```

Auf diesen grundlegenden Datenstrukturen und Zugriffsoperationen aufbauend, soll nun zuerst das Einstufen (`class`) eines Prozesses und damit sein Platzieren auf die für ihn geltende Warteschlange betrachtet werden. Die betreffende Operation setzt sich wie folgt zusammen:

```
inline void class(process_t *task, mood_t mood) {
    offer(lineup(level(&task->rank)), task, mood);
}
```

Jeder Prozess ist fest einer bestimmten Ebene (`level`, S. 69) in der mehrstufig organisierten ↑Ablaufplanung zugewiesen. Die Nummer dieser Ebene, der sogenannten *Stammebene* (vgl. S. 68), entspricht dem festen Rang des Prozesses und identifiziert den für diesen Prozess zu verwendenden Warteschlangendeskriptor (`lineup`) in dem Warteschlangenfeld. Mit diesen Parametern in der Hand wird der Prozess der späteren Prozessorzuteilung angeboten:

```
inline void offer(heat_t *heat, process_t *task, mood_t mood) {
    state(&task->mood, mood);                      /* define process state */
    (*heat->quote)(&heat->chart, task);                 /* enqueue process */
}
```

Zusätzlich zur Einstufung erfolgt die Zählung, um den betreffenden Prozess gänzlich als Kandidat für die Prozessorzuteilung aufzunehmen:

```
inline void house(process_t *task, mood_t mood) {
    class(task, mood);                      /* put process onto ready list */
    reckon(1);                                /* one more to be considered */
}
```

Diese Aufnahmeoperation bildet die Grundlage für die Bereitstellung lauffähiger Prozesse. Für die nichtpräemptiv erfolgende Prozessbereitstellung ist lediglich noch für die Parameterversorgung zu sorgen:

```
void ready(process_t *task) {  /* schedule according to nonpreemptive MLQ */
    process_t *self = being(ONESELF);           /* own process descriptor */

    assert((task != self) || valid(&self->mood, IDLE|PENDING));

    house(task, READY);                                /* add to ready list */
}
```

Die Zusicherung (`assert`) besagt, dass ein Prozess sich selbst nur dann bereitstellen darf, falls seine Prozessorabgabe (als soeben blockierter Prozess, s. `block`) noch bevorsteht (`PENDING`) oder er bereits als Leerlaufprozess agiert (`IDLE`). In beiden Fällen ist die Bereitstellung des laufenden Prozesses als Folge einer Unterbrechungsbehandlung oder ausgelöst von einem andern Prozessor eine korrekte Operation. Anderenfalls ist zu verhindern, dass ein Prozess, der

möglicherweise bereits auf der Bereitliste steht, nochmals auf diese gesetzt wird — um so nämlich einem inkonsistenten Zustand der betreffenden Warteschlange vorzubeugen.
Normalerweise geschieht die Bereitstellung eines Prozesses, um dessen vorangegangene Blockierung aufzuheben. Letzteres ist für gewöhnlich der Tatsache geschuldet, dass der Prozess erst den Eintritt eines bestimmten ↑Ereignisses abwarten muss, um weiter voranschreiten zu können, und dazu den Prozessor zugunsten eines anderen Prozesses abgibt (↑passives Warten). Dieser Prozess ist Ausgabe (`outgo`) einer Auswahloperation, die die Suche nach dem geeigneten Kandidaten für die Prozessorzuteilung in dem Rang (`level`) des blockierenden Prozesses (`self`) aufnimmt. Steht kein gleichrangiger Prozess bereit, startet ein Suchlauf über das die Bereitliste repräsentierende Warteschlangenfeld entsprechend der festgelegten globalen Strategie. Anschließend ist die Wiederaufnahme des Nachfolgeprozess einzuleiten:

```
void block() {                          /* schedule and release processor */
    process_t *self = being(ONESELF);           /* own process descriptor */
    process_t *next = outgo(level(&self->rank)); /* start scan at my rank */
    if (next == self)                        /* myself already unblocked? */
        state(&self->mood, FLUSH|RUNNING);               /* yes, continue */
    else {                                        /* no, release processor */
        state(&self->mood, BLOCKED|PENDING);       /* define process state */
        seize(next);                            /* switch to next process */
    }
}
```

Im Ergebnis wurde genau ein Prozess von der Bereitliste genommen (`outgo`), wodurch sich die Anzahl der zur ↑Einlastung des Prozessors verfügbaren Prozesse entsprechend verringert:

```
inline process_t outgo(tier_t tier) {
    process_t *next = trace(tier);            /* start scan at given rank */
    assert(next != 0);            /* never come here devoid of a process! */
    reckon(-1);                               /* one less to be considered */
    return next;                               /* deliver selected process */
}
```

Es wird immer ein Prozess aufgespürt (`trace`, s. unten), was durch die Zusicherung zum Ausdruck kommt. Die Suche nach dem nächsten Prozess startet auf dem angegebenen Rang (`tier`). Falls derzeit kein lauffähiger Prozess zur Verfügung steht, die Bereitliste also leer ist, wird das ↑Rechensystem in den ↑Leerlauf versetzt. Dieser Zustand wird erst verlassen, wenn zwischenzeitlich wenigstens ein Prozess wieder auf die Bereitliste gekommen ist.
Darüber hinaus kann der Fall eintreten, dass der stattfindende Prozess sich selbst von der Bereitliste holt (`block`, dann-Zweig): Das von dem blockierenden Prozess erwartete Ereignis ist zwischenzeitlich eingetreten und hat zu seiner Bereitstellung geführt, obwohl der Prozess den Prozessor (a) noch hält oder (b) als Leerlaufprozess eben auch nicht abgeben wird. Der Prozesswechsel geschieht daher nur, wenn ein anderer als der noch stattfindende Prozess von der Bereitliste genommen wurde.
Hinter der Operation, um den Nachfolger eines Prozesses aufzuspüren (`trace`) und von der Bereitlste zu nehmen, verbirgt sich eine typische, notwendige Folge zusammenhängender Ereignisse wie folgt: einerseits der *Suchlauf*, um einen zur Prozessorzuteilung verfügbaren Prozess zu finden, und andererseits der *Leerlauf*, wenn die Suche keinen verfügbaren Prozess hervorbringen sollte. Die Operation endet erst mit gefundenem, zuteilungsbereiten Prozess:

```
process_t *trace(tier_t tier) {    /* seek nonpreemptive runnable process */
    process_t *next;                        /* candidate to be switched on */

    while ((next = track(tier)) == 0) {           /* any candidate found? */
        stall(raring());                        /* no, become idle process */
        tier = 0;                       /* restart scan from the beginning */
    }

    return next;                                     /* deliver candidate */
}
```

Im Grunde unterscheidet sich diese Operation vor dem Hintergrund einer mehrstufig organisierten Bereitliste nur unwesentlich von der enstprechenden Operation für eine einstufige Bereitliste (vgl. S. 93). Das Aufspüren (`trace`) oder die Suche (`quest`) nach einen zuteilungsbereiten Prozess kann grundsätzlich unbestimmt lange andauern, wenn es zum Leerlauf kommt. Dann wird ein Prozess nämlich erst als Folge eines externen Ereignisses, verursacht durch ein ↑Peripheriegerät (↑*interrupt*) oder einen anderen Prozessor, auf die Bereitliste gelangen können. Tritt ein solches Ereignis niemals ein, liegt eine totale ↑Verklemmung im ↑Rechensystem vor. Allerdings ist die Frage, ob nun ein solches Ereignis noch eintritt und die Operation daher beendet werden kann, algorithmisch nicht entscheidbar (↑Halteproblem) — weshalb es hier auch nicht angebracht ist, in ↑Panik zu verfallen.

Das Aufstöbern (`track`) des nächst verfügbaren Prozesses startet auf der Ebene der Warteschlange des noch auf dem Prozessor stattfindenden Prozesses und versucht damit, zunächst einen gleichrangigen Kandidaten für die Prozessorzuteilung zu finden. Sollte der Suchlauf vergebens gewesen sein, kommt der suchende Prozess zum Stillstand (`stall`, vgl. auch S. 160). Nachdem dieser Prozess wieder Fahrt aufnimmt, startet er den jeweils nächsten Suchlauf immer von der ersten Ebene (d.h., dem höchsten Rang, $tier = 0$) ausgehend. Der dann schließlich aufgespürte Prozess wird als Suchergebnis zurückgeliefert.

Ausgehend von der Wurzelebene (`root`) wechselt die Suchfunktion (`track`) im weiteren Verlauf zur jeweils nächsten Ebene, wann immer eine leere Warteschlange vorgefunden wird. Schlimmstenfalls erfolgt ein kompletter Tabellendurchlauf. Blieb die Suche auch nach dem kompletten Durchlauf erfolglos, liefert die Funktion Null zurück. Ansonsten zeigt das Funktionsergebnis auf den als nächsten zur Einlastung bereitstehenden Prozess, der dazu seiner Warteschlange auch entnommen wurde:

```
process_t *track(tier_t root) {            /* scan by circular table walk */
    tier_t tier = root;                          /* starting point of scan */
    do {                                             /* perform table walk */
        process_t *next = catch(lineup(tier));           /* select process */
        if (next)                          /* anyone present at this level? */
            return next;                         /* yes, deliver candidate */

        tier = (tier + 1) % NTIER;                   /* step to next level */
    } while (tier != root);                             /* complete cycle? */

    return 0;                                     /* yes, no candidate found */
}
```

Dabei kommt für jede Ebene die jeweilige lokale Strategie des Verfahrens zur Wirkung, um den nächsten dort gegebenenfalls bereitstehenden Prozess zu ergreifen (`catch`) und als Ergebnis des Suchlaufs abzuliefern:

```
inline process_t *catch(heat_t *heat) {   /* remove process from waitlist */
    return (*heat->elect)(&heat->chart);              /* deliver candidate */
}
```

Den Laufzählerwert für die nächste jeweils zu prüfende Ebene beziehungsweise Warteschlange berechnet eine Modulooperation (`%`) mit der maximalen Anzahl von Feldeinträgen (`NTIER`) als Operand. Damit bildet das Feld praktisch einen Ring von Warteschlangen, der reihum abgelaufen wird solange kein Prozess gefunden wurde und der Umlauf noch nicht kom-

plett erfolgt ist. Demnach werden gegebenenfalls (bei größeren Laufzählerwerten) zunächst rangniedere Warteschlangen geprüft, bevor (mit kleineren Laufzählerwerten) auf ranghöhere Warteschlangen gewechselt wird. Hierbei ist zu beachten, dass gerade wegen der nichtpräemptiven Arbeitsweise dieser Lösungsvariante Prozesse bei Bereitstellung (`ready`) lediglich auf die ihrer Ebene entsprechenden Warteschlange kommen, nicht aber sofort auch gestartet werden, obwohl der Rang ihrer Ebene dies vorgeben würde. Um dann dem Verhungern (↑*starvation*) von Prozessen vorzubeugen, verläuft die Suche reihum in abnehmender Rangfolge und beginnt nicht immer am Anfang des Warteschlangenfelds.
Für kooperative Abläufe in ↑Maschinenprogrammen kann freiwilliges Pausieren des laufenden Prozesses hilfreich sein, ohne dabei jedoch explizit ↑logische Synchronisation betreiben zu müssen, das heißt, den betreffenden Prozess kurzzeitig zu blockieren, um ihn danach gleich wieder bereitzustellen. Stattdessen erklärt sich der Prozess als nur noch laufbereit, reiht sich selbst in die Bereitliste ein und gibt den Prozessor zugunsten eines anderen Prozesses ab, sofern es einen anderen laufbereiten Prozess gibt. Der betreffende Prozess bleibt also zuteilungsfähig und hat lediglich seinen ↑Prozesszustand von laufend in bereit geändert, ohne dabei den Umweg über blockiert zu nehmen. Im Falle von Vorrangplanung ist eine solche Operation jedoch nicht immer sinnvoll, denn der laufende Prozess hatte als höchstrangiger Prozess den Prozessor zugeteilt bekommen. Insbesondere bei einem präemptiv arbeitenden Verfahren ist davon auszugehen, dass der vorne auf der Bereitliste stehende Prozess keinen höheren Rang hat als der laufende Prozess, denn sonst hätte ersterer schon den Prozessor erhalten. Im Falle nichtpräemptiver Vorrangplanung kann es Situationen geben, die das Pausieren des laufenden Prozesses nicht ermöglichen, obwohl eine solche Maßnahme sehr wohl im Sinne der Strategie wäre (s. hierzu die Diskussion zu SPN, S. 316). In dem hier betrachteten Fall einer mehrstufig organisierten Bereitliste, wo jede Stufe nicht nur einem bestimmten Rang entspricht, sondern durch die stufeneigene Warteschlange mehrere Prozesse desselben Rangs bereit zur Prozessorzuteilung stehen können, zeigt sich jedoch ein anderes Bild, insbesondere wenn die jeweiligen Warteschlangen nach FCFS betrieben werden. Eine diesbezügliche Operation zum Pausieren des laufenden Prozesses umfasst nachfolgend skizzierte Schritte (vgl. S. 92):

```
void pause() {          /* voluntarily release processor according to MLQ */
    process_t *self = being(ONESELF);           /* own process descriptor */
    heat_t *heat = lineup(level(&self->rank));    /* get queue descriptor */
    process_t *next = catch(heat);                     /* fetch successor */
    if (next) {                              /* any process on same rank? */
        offer(heat, self, READY|PENDING);          /* yes, provide myself */
        seize(next);                                    /* switch process */
    }
}
```

Die Operation durchläuft ausschließlich die lokale Strategie (`catch`) des betreffenden Rangs (`level`) des laufenden Prozesses (`self`). Gibt es einen zum laufenden Prozess gleichrangigen Kandidaten, kommt es zur Prozessorübergabe: der laufende Prozess setzt sich laufbereit und platziert sich auf die seinem Rang entsprechende Warteschlange (`offer`), um anschließend den Prozesswechsel auszulösen (`seize`). Dabei liegt es letztlich ganz in der Hand der lokalen Auswahlstrategie (`elect`) einen möglichen Kandidaten auch zurückzuhalten, obwohl die betreffende Warteschlange nicht leer ist. Dies lässt dann etwa auch eine SPN betreibende lokale Strategie zu, für die aber die Pausierung des laufenden Prozesses keine Option darstellt und die demzufolge in dem Fall grundsätzlich Null als Ergebnis der Auswahlfunktion liefert.

In der präemptiven Lösungsvariante wird Verhungern bewusst in Kauf genommen, da ranghöhere Prozesse immer den Vorzug gegenüber rangniederen Prozessen erhalten. Ausgelöst wird die Verdrängung des laufenden Prozesses hier jedoch ausschließlich als Folge der Anforderung zur Bereitstellung eines Prozess, dessen Wartebedingung aufgehoben wurde, das heißt, der sich im Prozesszustand blockiert befindet. Der bereitzustellende Prozess ist ge-

genüber einen bereits laufenden Prozess bevorrechtigt (↑*preemptive*), wenn letzterer einen niedrigeren Rang innehat. Im dem Fall wird der laufende Prozess zugunsten des bereitzustellenden Prozesses vom Prozessor verdrängt. Dabei ist zu beachten, dass in Bezug auf ein und denselben Prozess eine solche Anforderung niemals mehrfach gestellt wird (vgl. SRTF, S. 322) — vorausgesetzt, das ↑Betriebssystem verhält sich korrekt. Darüberhinaus ist, wie in anderen präemptiven Verfahren der ↑Vorrangplanung auch, besondere Vorsicht geboten, wenn nämlich jederzeit die Bereitstellung von Prozessen geschehen kann für eben den Prozessor, auf dem ein stattfindender Prozess geradewegs einen ↑Prozesswechsel ansteuert (vgl. SRTF, S. 321). Soll ↑präemptive Planung im Betriebssystem für vorranggesteuerte Prozesse uneingeschränkt (↑*full preemptive*) durchgesetzt werden, sind folgende neuralgische Stellen zu beachten (höhere Rangstufe gleich kleinerer Laufzählerwert bzw. Feldindex):

- Angenommen P^4_{run}, der stattfindende und bei ↑Vorrangsteuerung in dem Moment auch ranghöchste Prozess, sei Rang 4 zugeordnet und wird durch eine Bereitstellungsanforderung für P^3_{rdy}, einem Prozess höherer Rangstufe 3, unterbrochen. Folglich muss P^4_{run} den Prozessor an P^3_{rdy} abgeben. Dazu wird P^4_{run} gezwungen, sich selbst auf die seinem Rang entsprechende Warteschlange der mehrstufigen Bereitliste zu setzen, $P^4_{run} \mapsto P^4_{run:rdy}$, und den Prozesswechsel an P^3_{rdy} einzuleiten. Der Zwang dazu geschieht im Rahmen der Unterbrechungsbehandlung, die immer im Namen des jeweils unterbrochenen Prozesses, hier P^4_{run}, stattfindet und diesen (indirekt mittels Aufruf von `ready`) schließlich zum Prozesswechsel bringt. Auf dem Weg zum Prozesswechsel wird der noch laufende, aber bereits auf der Bereitliste stehende Prozess ($P^4_{run:rdy}$) abermals unterbrochen, diesmal durch eine Bereitstellungsanforderung für P^2_{rdy}, dessen Rang 2 ebenfalls höher ist als der von $P^4_{run:rdy}$. Folglich durchläuft $P^4_{run:rdy}$ dieselbe Prozedur wie zuvor bereits P^4_{run}, obwohl er bereits auf der Bereitliste steht. Einer solchen wiederholten Bereitstellung desselben Prozesses, als Folge der indirekten ↑Rekursion der Bereitstellungsoperation (`ready`), ist vorzubeugen.
- Angenommen die Bereitstellungsanforderung für P^2_{rdy} trifft vor der für P^3_{rdy} ein, das heißt, die P^2_{rdy} betreffende Unterbrechungsbehandlung wird unterbrochen und durch die P^3_{rdy} betreffende Unterbrechungsbehandlung überlagert (↑*cascaded interrupt*). In dem Fall kann nicht nur $P^4_{run:rdy}$ wiederholt auf die Bereitliste kommen, sondern darüberhinaus erhält P^3_{rdy} den Prozessor vor P^2_{rdy} zugeteilt, sobald nämlich der Prozesswechsel vollzogen wird. Dieser Wechsel von $P^4_{run:rdy}$ hin zu P^3_{rdy} ist eine Form von ↑Prioritätsverletzung, die nach erfolgter Rangprüfung und Feststellung, dass P^4_{run} den Prozessor zugunsten von P^2_{rdy} abgeben muss, noch vermieden werden kann. Wird nämlich eben dieser besondere Zustand von $P^4_{run:rdy}$ erkannt, kommt P^3_{rdy} in die seinem Rang entsprechende Warteschlange der mehrstufigen Bereitliste.
- Angenommen die indirekte Rekursion der Bereitstellungsoperation (`ready`) geschieht bereits bevor P^4_{run} überhaupt die Möglichkeit zur Rangprüfung hatte. Eine Bereitstellungsanforderung für einen Prozess bedeutet zunächst nichts weiter als den betreffenden Prozess bereitstellen zu wollen. Dabei liegt dieser Prozess, konkret P^2_{rdy}, ganz allein in der Hand des laufenden Prozesses (P^4_{run}), etwa repräsentiert als *lokale* ↑*Variable* oder ↑tatsächlicher Parameter der Bereitstellungsoperation. In dieser Situation lässt sich die ↑Prioritätsverletzung durch den Prozesswechsel hin zu P^3_{rdy} jedoch nur vorbeugen, wenn im Moment der Feststellung der Bereitstellungsanforderung für P^2_{rdy}, als das Betriebssystem das von P^2_{rdy} erwartete ↑Ereignis wahrnahm, entweder eine ↑Verdrängungssperre oder eine ↑Prozesssperre mit dem Rang von P^2_{rdy} erhoben wird. Letztere entspricht der ↑Prioritätsvererbung, indem P^4_{run} zeitweilig der Rang 2 zugewiesen wird und erst nach erfolgtem Prozesswechsel hin zu P^2_{rdy} seinen ursprünglichen Rang 4 wieder erhält. Diesen ursprünglichen Rang des Vorgängerprozesses wieder einzustellen wäre ↑Prozessverpflichtung von dem dann laufenden P^2_{run}.
- Angenommen P^4_{run} blockiert und hat bereits seinen Nachfolgeprozess, P^7_{nxt}, der den niedrigeren Rang 7 einimmt, aus der Bereitliste entfernt (`elect`). Geradewegs zum

Prozesswechsel hin zu P^7_{nxt} kommt es zur Bereitstellungsanforderung für einen Prozess, P^6_{rdy}, dessen Rang 6 höher ist als der von P^7_{nxt}. Da jedoch der Rang von P^4_{run} eine Verdrängung nicht zulässt, kommt P^6_{rdy} in die seinem Rang entsprechende Warteschlange der mehrstufigen Bereitliste. Sodann erhält mit P^7_{nxt} ein Prozess den Prozessor, obwohl ein Prozess höheren Rangs (P^6_{rdy}) auf Zuteilung wartet. Daher ist es die Prozessverpflichtung von P^7_{nxt}, nach Aufnahme seiner Tätigkeit auf die Richtigkeit der Zuteilungsentscheidung hin zu prüfen und eventuell den Prozessor wieder abzugeben.

Um eine möglichst uneingeschränkte (*full preemptive*) Arbeitsweise im Zusammenhang mit der Bereitstellung von Prozessen zu ermöglichen, sind ↑Wettlaufsituationen der eben genannten Art durch ↑Koordination der möglichen gleichzeitigen Abläufe geeignet zu begegnen. Dadurch nimmt die präemptive Lösungsvariante zum nichtpräemptiven Gegenstück erheblich an Komplexität zu. Hervorzuheben ist dabei, dass die geschilderten Problempunkte allein durch ↑Unterbrechungsanforderungen verursacht sind und ausschließlich Abläufe auf ein und demselben Prozessor oder Rechenkern betreffen, also unabhängig von einem ↑Multiprozessor oder ↑Mehrkernprozessor geschehen können. Bei genauerer Betrachtung der Problematik ist sogar festzustellen, dass Lösungen dazu in Bezug auf den ↑Kontrollfluss im Betriebssystem für *echte* ↑Parallelität teils sogar einfacher sind als die hier diskutierte Lösung für *unechte*, das heißt, ↑Pseudoparallelität. Die Schwierigkeit des präemptiven Ansatzes stellt sich also vor allem beim ↑Uniprozessor, und zwar im Falle von ↑Unterbrecherbetrieb, nämlich der ↑Betriebsart, die eben überhaupt erst die technische Voraussetzung für die Verdrängung eines Prozesses von seinem Prozessor liefert.
Der hier betrachtete Ansatz zur Lösung der geschilderten Problematik zieht Änderungen an mehreren Stellen (`ready`, `block`, `trace`, `track`, `pause` und `seize`) nach sich, die nachfolgend behandelt. Diesen Änderungen gemeinsam ist eine differenzierende Auslegung des der ↑Dringlichkeit eines Prozesses entsprechenden Prozessattributs, wofür ein entsprechend strukturierter ↑Datentyp ins Spiel kommt (vgl. S. 68).
Zunächst die Bereitstellung eines Prozesses, die nunmehr zur ↑Verdrängung des laufenden Prozesses, unter dessen Kontrolle eben diese Operation geschieht (d.h., der bereitstellende Prozess), führen kann. Darüberhinaus muss diese Operation mit einer möglichen indirekt rekursiven Aktivierung umgehen, die nämlich durch „gleichzeitige“ Bereitstellung verschiedener Prozesse geschehen kann. Das Problem dabei ist jedoch nur der Fall, wenn jeder dieser Prozesse den laufenden Prozess verdrängen müsste. Im Vergleich zur nichtpräemptiven Variante (S. 189) ergeben sich gravierende Änderungen:

```
void ready(process_t *task) {     /* schedule according to preemptive MLQ */
    process_t *self = being(ONESELF);           /* own process descriptor */

    if (!prior(&task->rank, &self->rank) || !avail(self, READY|PENDING))
        house(task, READY);         /* add provided process to ready list */
    else                 /* higher ranked process, prepare for preemption */
        watch(seize(task));  /* switch process, fulfil process obligation */
}
```

Auffallend ist der Verzicht auf die bei der nichtprämptiven Lösung noch vorhandenen Zusicherung, dass ein Prozess sich selbst nur unter bestimmten Bedingungen bereitstellen kann. Demgegenüber muss hier bei der präemptiven Lösung sogar die wiederholte Aufforderung, sich selbst bereitzustellen, als völlig normal angesehen werden. Der einfache Fall ist nun der, wenn der bereitzustellende Prozess (`task`) einen niedrigeren Rang hat als der bereitstellende/laufende Prozess (`self`). Für diese Überprüfung (`prior`, S. 69) wird der Rang der sogenannten *Wirkebene* (vgl. S. 68) des laufenden und der Rang der Stammebene des bereitzustellenden Prozesses herangezogen. Ist die Wirkebene von höherem Rang, kommt der bereitzustellende Prozess in die ihm zugewiesene Warteschlange und der Prozess, unter dessen Kontrolle die Bereitstellung geschieht, fährt fort. Anderenfalls, wenn die Wirkebene des laufenden Prozesses einen niedrigeren oder den gleichen Rang hat, darf der in Frage kommende Verdrängungsvorgang auch im indirekt rekursivem Fall den laufenden Prozess nur einmal

auf die Bereitliste setzen. Dies stellt die erweiterte Prozessaufnahmeoperation (`avail`) sicher:

```
inline int avail(process_t *task, mood_t mood) {
    int done;                           /* placeholder for function value */
    if ((done = blend(&task->mood, mood)))     /* state change succeeded? */
        house(task, mood);                      /* yes, add to ready list */
    return done;                          /* indicate success or failure */
}
```

Nur wenn das Vermengen (`blend`, S. 259) des gewünschten Prozesszustands (`mood`) mit dem des gegebenen Prozesses (`task`) gelingt, wird die Aufnahme analog zur nichtpräemptiven Variante (`house`, S. 189) vorgenommen. Der damit einhergehende Zustandsübergang setzt jedoch voraus, dass sich der betreffende Prozess nicht bereits in dem Zielzustand befindet. Im konkreten Beispiel (s. `ready`) wird also der Prozess (`task`) nur dann eingestuft, das heißt, von der Bereitstellung Gebrauch gemacht (`avail`), wenn sein gegenwärtiger Prozesszustand nicht bereits das Attribut `READY|PENDING` enthält. Um auch im Falle „gleichzeitiger“ (indirekt rekursiver) Bereitstellungen bevorrechtigter Prozesse den Verdrängungsvorgang konsistent durchführen zu können, wird der Zustandsübergang durch eine ↑Elementaroperation vollzogen. Dazu greift die in Frage kommende Operation (`blend`) auf ↑CAS zurück und nimmt lediglich die dazu benötigte Operandenversorgung vor (vgl. S. 259).
Die der Bereitstellung normalerweise vorangehenden Operation ist die Blockierung des laufenden Prozesses. Auch diese Funktion muss gegenüber der nichtpräemptiven Variante abgeändert werden. Jedoch ist nur ein kleiner Eingriff nötig, der dafür sorgt, dass sofort nach Wiederaufnahme des deblockierten Prozesses dieser der oben genannten Prozessverpflichtung (`watch`, S. 197) nachkommen kann. Dies bedingt jedoch zusätzlich eine Abänderung der Prozesswechseloperation (`seize`), die nunmehr einen Funktionswert liefert, nämlich um den Vorgängerprozess, dem die Verpflichtung gilt, identifizieren zu können:

```
void block() {                          /* schedule and release processor */
    process_t *self = being(ONESELF);           /* own process descriptor */
    process_t *next = outgo(level(&self->rank)); /* start scan at my rank */
    if (next == self)                        /* myself already unblocked? */
        state(&self->mood, FLUSH|RUNNING);                /* yes, continue */
    else {                                     /* no, release processor */
        state(&self->mood, BLOCKED|PENDING);      /* define process state */

        watch(seize(next));  /* switch process, fulfil process obligation */
    }
}
```

Ausgehend von der Stammebene (`level`) sucht der noch laufende, aber demnächst den Prozessor abgeben wollende, Prozess (`self`) daselbst seinen Nachfolger und erhält diesen als Ausgabe (`outgo`) der Suchfunktion. Im Unterschied zur nichtpräemptiven Variante (S. 190) kann der Nachfolgerprozess nicht von höherem Rang sein, denn sonst hätte er im Moment seiner Bereitstellung den laufenden Prozess vom Prozessor verdrängt. Folglich hat der Nachfolger den gleichen oder einen niedrigeren Rang. Sobald der Prozess, der als nächstes den Prozessor erhalten soll (`next`), gefunden wurde, ist es zudem möglich, dass weitere Bereitstellungsanforderungen einen oder mehrere andere Prozesse auf die Bereitliste gesetzt haben. Hervorzuheben sind dabei all jene Prozesse, deren jeweiliger Rang der Stammebene niedriger ist als der der Wirkebene des noch laufenden, den Prozessor allerdings gleich abgebenden Prozesses (`self`), aber höher ist als der Rang der Stammebene des in Kürze den Prozessor erhaltenden Prozesses (`next`). Die in solch einem Szenario grundsätzlich unabwendbare Prioritätsverletzung, die nach der Auswahl (`outgo`) eines rangniedrigeren Prozesses (Nachfolger) geradewegs hin zum Prozesswechsel (`seize`) unter Kontrolle des ranghöheren Vorgängerprozesses aufgetreten sein kann, lässt sich nur im Rahmen der Prozessverpflichtung (`watch`, s. unten) des Nachfolgers bereinigen.
Die im Rahmen der Auswahloperation (`outgo`, S. 190) stattfindende Nachfolgersuche (`trace`)

geht mit logischem Stillstand (`stall`) des schwebend blockierten Prozesses einher, solange das Aufstöbern (`track`) eines laufbereiten Prozesses erfolglos ist. Anders als im nichtpräemptiven Fall werden immer nur die Warteschlangen höherer Ebenen (tieferer Ränge) ausgehend von der Stammebene des laufenden Prozesses (`tier`) abgesucht. Prozesse niedrigerer Ebene (höherem Rang) gelangen präemptiv an den Prozessor:

```
process_t *trace(tier_t tier) {       /* seek preemptive runnable process */
    process_t *next;                       /* candidate to be switched on */

    while ((next = track(tier, NTIER, 0)) == 0)   /* any candidate found? */
        stall(raring());                       /* no, become idle process */

    return next;                                     /* deliver candidate */
}
```

Die präemptive Lösung verkürzt die ↑Latenzzeit, bis ein ranghöherer Prozess den Prozessor erhält, wie auch die Nachfolgersuche eines jeden Prozesses, der beabsichtigt, den Prozessor abzugeben. Beide Aspekte schlagen sich in der abgeänderten Funktion zum Aufstöbern (`track`) solcher Prozesse nieder:

```
process_t *track(tier_t root, tier_t last, bool step) { /* ascending scan */
    process_t *self = being(ONESELF);           /* own process descriptor */

    for (heat_t *heat = lineup(root); heat < lineup(last); heat++) {
        alter(&self->rank, step);       /* if so, pass to next lower rank */

        process_t *next = catch(heat);                   /* select process */
        if (next)                        /* anyone present at this level? */
            return next;                        /* yes, deliver candidate */
    }

    return 0;                                       /* no candidate found */
}
```

Im Gegensatz zur nichtpräemptiven Lösung (vgl. S. 191) ist nicht nur der Anfang (`root`), sondern auch das Ende (`last`) des Suchbereichs vorgegeben. Letzteres bedingt die Prozessverpflichtung, nämlich die im Verlauf der Prozessorübergabe eventuell zurückgestellten Prozesse, deren jeweiliger Stammebenenrang im Vergleich mit dem Wirkebenenrang des den Prozessor (a) abgebenden Prozesses niedriger und (b) entgegennehmenden Prozesses höher ist, nachträglich zu bevorrechtigen (vgl. `watch`, unten). Bereitstellungsanforderungen für Prozesse mit höherem Rang als der Prozess, unter dessen Kontrolle diese Suche läuft, bewirken normalerweise eine Verdrängung. Daher wird das Feld nur mit aufsteigendem Laufindex (höherer Ebene, aber abnehmendem Rang) abgesucht. Gegebenenfalls verringert der Prozess, der den Suchlauf kontrolliert, dabei den Rang seiner Wirkebene (`alter`, S. 70). Die Absenkung des Wirkebenenrangs geschieht aber nur, wenn dem Prozess zuvor ein höherer Rang als der seiner Stammebene zugewiesen wurde ($step = true$), um nämlich die ihm auferlegte Prozessverpflichtung nachzukommen (`watch` ⇝ `track`). In dem Fall endet die Suche spätestens mit Erreichen der Stammebene dieses Prozesses. Geschieht die Suche im Verlauf der Blockierung (`block` ⇝ `outgo` ⇝ `trace` ⇝ `track`) und damit im Namen des den Prozessor abgebenden Prozesses, bleibt der Rang unverändert ($step = false$). In dem Fall schreitet die Suche bis zum Ende des Warteschlangenfelds voran und endet spätestens mit dem Rang der Stammebene der gemächlichsten ($NTIER - 1$) Prozesse.

Nach Aufnahme der Tätigkeit eines Prozess, das heißt, nach Rückkehr aus der von ihm zu einem früheren Zeitpunkt (in `block`, `ready` oder `pause`) begangenen Prozesswechseloperation (`seize`), geht dieser seine Prozessverpflichtung ein, auf die Verfügbarkeit zwischenzeitlich bereitgestellter höherrangiger Prozesse hin zu prüfen. Dazu muss der betreffende Prozess entsprechend der globalen Strategie vorgehen, um den höchstrangigsten dieser Prozesse zu finden. Dies wiederum bedeutet hier, die Suche mit der vordersten Warteschlange der mehrstufig organisierten Bereitliste aufzunehmen:

```
inline void watch(process_t *task) {         /* fulfil process obligation */
    process_t *next, *self = being(ONESELF);
    clean(task);                    /* adjust rank of predecessor process */
    angle(&self-rank, 0);                   /* prevent priority violation */
    if ((next = track(0, level(&self->rank), 1))) { /* any higher ranked? */
        class(self, READY|PENDING);             /* yes, make me available */
        watch(seize(next));               /* resume higher ranked process */
    }
}
```

Trotz präemptiver Herangehensweise ist es durchaus möglich, dass Prozesse von höherem Rang als der laufende Prozess im Zuge ihrer Bereitstellung nicht immer den Prozessor zugeteilt bekommen können. Dies ist kein Fehler des Verfahrens, sondern ein — wenngleich auch nicht wünschenswertes — Merkmal. Im Zusammenhang mit der Blockierung (`block`) und Bereitstellung (`ready`) wurde bereits darauf hingewiesen. Insbesondere kann eine indirekt rekursive Bereitstellungsanforderung sogar die Zurückstellung des überhaupt höchstrangigsten Prozesses zur Folge haben. Daher beginnt die Suche (`track`) mit der für Rang 0 zuständigen Warteschlange und sie endet mit dem Rang der Stammebene (`level`) des seiner Verpflichtung nachkommenden Prozesses (`self`).

Hierbei ist besonders darauf zu achten, dass es nach Auffinden eines solchen zurückgestellten Prozesses nicht zu einer Prioritätsverletzung kommt. Angenommen der seiner Verpflichtung gerade nachkommende Prozess (`self`) hat einen zurückgestellten höherrangigen Prozess (`next`) aufgestöbert (`track`), der Bereitliste entnommen, und wird sodann vom Prozessor verdrängt durch eine Bereitstellungsanforderung für einen Prozess, dessen Rang niedriger als der des aufgestöberten Prozesses und eben höher als sein eigener Rang ist. Ob dieser aufgestöberte, hochrangige Prozess den Prozessor erhält, ist überhaupt ungewiss, denn er steht nicht mehr auf der Bereitliste und kann demzufolge bei keiner anderen Prozessverpflichtung aufgestöbert werden. Der rangniedere Prozess (`self`) muss selbst erst wieder den Prozessor erhalten, damit der Wechsel hin zum ranghöheren Prozess (`next`) geschehen kann.

Um einer möglichen Prioritätsverletzung vorzubeugen begibt sich der die Prozessverpflichtung nachkommende Prozess (`self`) auf den Rang der ersten zu prüfenden Warteschlange (Rang 0), er ändert den Rang seiner Wirkebene (`angle`) entsprechend. Beim Stöbern (`track`) über die Bereitliste verringert der Prozess sodann den Rang seiner Wirkebene schrittweise ($step = true$) mit jedem Wechsel zu einer anderen, rangniedrigeren Warteschlange (`track` $\rightsquigarrow$ `alter`). Damit kommt ein Prozess, dessen Rang höher als der der Stammebene des laufenden (stöbernden) Prozesses (`self`) und niedriger als der Rang des aufgestöberten Prozesses (`next`) ist, vorerst nicht mehr zum Zuge. Wenn nun ein ranghöherer Prozess aufgestöbert wurde, findet ein Prozesswechsel (`seize`) statt, wobei dann der Rang der Wirkebene des den Prozessor abgebenden Prozesses (`self`) höher ist als der Rang seiner Stammebene.

Diese mögliche Inkonsistenz des Rangs von Stamm- und Wirkebene wird ebenfalls im Rahmen der Prozessverpflichtung bereinigt (`clean`), und zwar in Bezug auf den jeweiligen Vorgängerprozess (`task`). Dieser Prozess hat gegebenenfalls selbst als Folge seiner Prozessverpflichtung den Prozessor abgegeben, mit der eben dargestellten Konsequenz. Eine solche Rangveränderung hatte jedoch nur Bedeutung im letzten Zyklus (d.h., bis zur Prozessorabgabe) des betreffenden Prozesses:

```
inline void clean(process_t *task) {            /* clearing up of process */
    angle(&task->rank, level(&task->rank));         /* reset dynamic rank */
}
```

Der Rang der Wirkebene des angegebenen Prozesses wird für den nächsten Zyklus neu eingestellt (`angle`, S. 69) und damit auf den mit der jeweiligen Stammebene (`level`, S. 69) definierten Standard (*default*) zurückgesetzt.

Als Folge der Rangerhöhung im Rahmen der Prozessverpflichtung und der Tatsache, dass der betreffende Prozess bereits nach erstmaliger Absenkung seines höheren Rangs bei der

Suche nach einen ranghöheren Prozess (`track`) verdrängt werden kann, ist eine Änderung in der Einstufungsoperation (`class`, S. 189) vorzunehmen: Die Einstufung des Prozesses darf nicht mehr gemäß des Rangs seiner Stammebene (`level`) erfolgen, sondern muss sich an dem Rang seiner Wirkebene (`gauge`) orientieren. Anderenfalls kann es ebenfalls zu der oben geschilderten Prioritätsverletzung kommen, wenn nämlich der laufende Prozess (`self`) seinen Nachfolger (`next`) bereits in der Hand hat und geradewegs (direkt oder indirekt) die Prozessorabgabe betreibt, die mit seiner Einstufung (`class`) einhergeht.
Wenn es somit bei der Ausübung einer Prozessverpflichtung (direkt oder indirekt) zum Prozesswechsel (`seize`) kommt, wird der den Prozessor abgebende (die Prozessverpflichtung durchführende) Prozess genau in diesem Kontext wieder fortgesetzt, nachdem er den Prozessor zurückerhalten hat. Die geschehene Wiederaufnahme dieses Prozesses impliziert aber auch, dass er bei der Auswahl zur Einlastung des Prozessors der in dem Moment ranghöchste Prozess war und damit ganz vorne auf der Bereitliste stand. Darüberhinaus ist zu beachten, dass dann auch der Rang seiner Wirkebene wieder dem Rang seiner Stammebene entspricht (`clean`). Der *endrekursive Aufruf* von `watch`, den der ↑Kompilierer für gewöhnlich auf eine einfache Programmschleife abbildet, sorgt sodann dafür, dass alle zwischenzeitlich doch noch eingetroffenen ranghöheren Prozesse zum Zuge gekommen sind, bevor der hier betrachtete Prozess (`self`) die Prozessverpflichtung erfüllt hat und seines weiteren Weges zieht.
Auch die Operation zum Pausieren (`pause`, S. 192) erfordert eine kleine Änderung, um für die präemptive Verfahrensweise gerüstet zu sein. Wie zuvor bereits bei der Blockierung (`block`, S. 195) und der Bereitstellung (`ready`, S. 194) gezeigt, muss die Operation zum Prozesswechsel (`seize`) auch hier direkt in die Prozessverpflichtung (`watch`, S. 197) münden.
Abschließend noch der Hinweis, dass die oben skizzierte präemptive Verfahrensweise gerade in Bezug auf die Absuche der mehrstufig organisierten Bereitliste im Rahmen der Prozessverpflichtung (`watch`) verbessert werden kann — allerdings auf Kosten weiterer Änderungen an anderer Stelle. Anstatt nämlich immer die Suche mit der Rang 0 zugeordneten Warteschlange zu beginnen, genügt es, die erste nichtleere Warteschlange zum Ausgangspunkt zu nehmen. Jedoch erfordert diese, den Rang des höchstrangigen Prozesses, der nicht zur Verdrängung führen durfte und demzufolge zurückgestellt werden musste, festzuhalten. Dazu wäre wenigstens die Operation zum Anordnen (`house`, S. 189) und zur Ausgabe (`outgo`, S. 190) eines Prozesses entsprechend abzuändern, nämlich um Buch über den aktuell höchsten Rang eines bereitgestellten Prozesses zu führen. Vereinfachungen an anderer Stelle ergeben sich dadurch allerdings nicht — die strukturelle Komplexität des Verfahrens nimmt dann aber weiter zu. Alternativ dazu kann auch eine grundsätzlich andere Datenstruktur zur Repräsentation der Bereitliste in Erwägung gezogen werden, die letztlich ein schnelleres Aufstöbern zwischenzeitlich zurückgestellter Prozesses ermöglicht. Aber auch dies zieht weitestgehend querschneidende Änderungen nach sich, die dann zu Lasten der Einstufung von Prozessen geht.

MMU Abkürzung für (en.) ↑*memory management unit.*

Mnemon (gr.) erinnern.

mode change (dt.) ↑Moduswechsel.

mode of operation (dt.) ↑Betriebsmodus. Siehe auch (en.) ↑*operating mode.*

modified bit (dt.) ↑Veränderungsbit (Patentwesen).

Modul (en.) ↑*module.* Softwareeinheit mit eigenem Datenmodell und eigenem Satz von Operationen darauf. Die Daten sind nur über die moduleigenen Operationen zugänglich (*information hiding*).

module (dt.) Baugruppe, ↑Modul, Serieneinheit.

Moduswechsel (en.) ↑*mode change.* Übergang von einen ↑Arbeitsmodus in einen anderen.

Monitor (en.) ↑*monitor*. Datentyp, Klasse mit impliziten Eigenschaften zur ↑Synchronisation; Programmierkonvention. Für alle Operationen, die auf ein ↑Exemplar eines solchen Datentyps angewendet werden, gilt ↑wechselseitiger Ausschluss. Diese Operationen müssen in der Datentypschnittstelle spezifiziert sein. Per Definition ist jedes ↑Unterprogramm, das eine solche Operation implementiert, ein ↑kritischer Abschnitt. Innerhalb eines solchen Abschnitts kann ein ↑Prozess eine ↑Bedingungsvariable verwenden, um sich auf ein ↑Ereignis zu synchronisieren beziehungsweise ein solches anzuzeigen. Mit dieser Variablen assoziiert ist eine ↑Signalisierungsdisziplin, die nicht nur eine von mehreren möglichen Semantiken zur ↑Bedingungssynchronisation begründet, sondern auch ein unterschiedliches Verhalten der in der ↑Kooperation beteiligten Prozesse festlegt.
Ursprünglich ein Konzept der *Typisierung* in höheren Programmiersprachen, indem nämlich die korrekte Verwendung von Operationen zur Synchronisation durch einen ↑Kompilierer abprüfbar wird und so Programmierfehler vermieden werden können. Die zum wechselseitigen Ausschluss erforderlichen Anweisungen erzeugt der Kompilierer, wie auch die Instruktionen, um einen kritischen Abschnitt zeitweilig verlassen zu können, ohne diesen in der Zwischenzeit gesperrt zu halten. Mangels Sprachunterstützung ist dieses Konzept jedoch viel mehr zur Programmierkonvention entartet, das heißt, dem Menschen als ↑Prozessor wird es erleichtert, ein (vermeintlich) korrektes ↑nichtsequentielles Programm zu formulieren.

monitor (dt.) ↑Monitor, Wächter.

motherboard (dt.) ↑Trägerleiterplatte.

mount (dt.) Anbau, ↑Befestigung, Fassung, Halterung.

mount point (dt.) Punkt, an dem `mount(2)` ein ↑Dateisystem aufpfropft (↑*mounting point*).

mounting point (dt.) ↑Befestigungspunkt.

MPU Abkürzung für (en.) ↑*memory-protection unit*.

MS-DOS Akbürzung für „*Microsoft Disk Operating System*“, Einplatz-/Einbenutzerbetriebssystem, erste Installation 1981 (Intel 8086).

multi-address space (dt.) ↑Mehradressraum.

multi-core (dt.) ↑mehradrig.

multi-core processor (dt.) ↑Mehrkernprozessor, Mehrkerner (Jargon); ↑mehradriger ↑Prozessor.

multi-level machine (dt.) ↑Mehrebenenmaschine.

multi-stream batch monitor (dt.) ↑Mehrstromstapelmonitor.

multi-threaded server (dt.) ↑mehrfädiger Zusteller.

multi-user mode (dt.) ↑Mehrbenutzerbetrieb.

multicast (dt.) ↑Gruppenruf.

Multics Mehrplatz-/Mehrbenutzerbetriebssystem, Abkürzung für (en.) „*Multiplexed Information and Computing Service*“. Konzeptpapiere im Jahr 1965, erste echte Installation Dezember 1967 (↑GE 645), programmiert in EPL (*early* ↑PL/1), in Betrieb bis Oktober 2000. Das System bot einige zukunftsweisende Konzepte: ein nicht mehr zwischen ↑Dateien (bez. als ↑Segmente in der ↑Ablage) und herkömmlichen ↑Hauptspeicher unterscheidender ↑einstufiger Speicher, ein ↑hierarchischer Namensraum für das ↑Dateisystem, ↑dynamisches Binden, ein interaktiver ↑Kommandointerpreter als gewöhnliches ↑Maschinenprogramm (↑*shell*), mitlaufende (*on-line*) ↑Rekonfigurierung und eine in Bezug auf ↑Angriffssicherheit durchgängige Ausgestaltung der ↑Systemfunktionen.

multilateral synchronisation (dt.) ↑multilaterale Synchronisation.

multilaterale Synchronisation (en.) ↑*multilateral synchronisation*. Bezeichnung einer ↑Synchronisation, die sich auf jeden beteiligten ↑Prozess auswirken kann. Jeder der betroffenen Prozesse bestreitet dasselbe Protokoll, um Synchronisation in Bezug auf eine bestimmte ↑Aktion oder ↑Aktionsfolge zu erzielen. Kommt hierzu ↑blockierende Synchronisation zur Geltung, wird jeder dieser Prozesse durch das Protokoll blockiert werden können. Für ↑nichtblockierende Synchronisation lässt dieses Protokoll jeden der beteiligten Prozesse die in ↑Konkurrenzsituation gescheiterte Aktion/Aktionsfolge wiederholen.
Diese Art der Synchronisation ist typisch, wenn Prozesse gleichzeitig ein ↑wiederverwendbares Betriebsmittel anfordern, das jedoch nur exklusiv belegt und genutzt werden darf. Eine besondere Variante davon ist ein ↑kritischer Abschnitt, für den ↑wechselseitiger Ausschluss von Prozessen sicherzustellen ist. Alle diese Verfahren wirken mehrseitig.

multiplexing (dt.) ↑Multiplexverfahren.

Multiplexverfahren (en.) ↑*multiplexing*. Methode, deren Ursprung in der Nachrichtentechnik liegt, nämlich um mehrere Signale (↑konsumierbares Betriebsmittel) simultan über ein Medium (↑wiederverwendbares Betriebsmittel) übertragen zu können. In einem ↑Betriebssystemkern findet diese Methode Anwendung, um mehr als einen ↑Prozess zugleich auf dem ↑Prozessor einer (realen/virtuellen) Maschine stattfinden lassen zu können. Grundlage dafür bilden entsprechende Verfahren zur ↑Prozesseinplanung, die ↑Simultanverarbeitung ermöglichen.

multiprocessing (dt.) ↑Simultanverarbeitung.

multiprocessor (dt.) ↑Multiprozessor.

multiprogramming (dt.) ↑Mehrprogrammbetrieb.

Multiprozessor (en.) ↑*multiprocessor*. Bezeichnung für einen ↑Rechner, der aus mehr als einer ↑CPU besteht.

multitasking (dt.) ↑Mehraufgabenbetrieb, ↑Mehrprogrammbetrieb, ↑Mehrprozessbetrieb.

multithreading (dt.) ↑Mehrfädigkeit.

Mutex Kunstwort, Abkürzung für (en.) ↑*mutual exclusion*; Sperrmechanismus, mit dem ↑wechselseitiger Ausschluss ausgeübt wird. Der Mechanismus sichert zu, dass nur der ↑Prozess die Sperre aufheben kann, der diese zuvor auch gesetzt hat. Typischerweise wird mit diesem Konstrukt ein ↑kritischer Abschnitt eingefasst. Einen damit eingefassten Abschnitt kann nur der Prozess entsperren, der diesen zuletzt sperrte, betrat und jetzt verlässt. Für jeden anderen Prozess scheitert die ↑Aktion — was in dem Fall auf einen Programmierfehler hindeutet und eigentlich eine ↑Ausnahmesituation darstellt, im Allgemeinen jedoch ohne Auswirkung für den offensichtlichen fehlgeleiteten Prozess bleibt. Grundlage bildet typischerweise ein ↑binärer Semaphor, eine ↑Umlaufsperre oder eine ↑Lese-/Schreibsperre: In der Sperroperation (`lockup`) wird nach dem Erwerb (*`acquire`*) der Sperre der gegenwärtige Prozess vermerkt, dessen Identität in der Entsperroperation (`unlock`) noch vor der Abgabe (*`release`*) der Sperre mit demjenigen Prozess überprüft wird, der den kritischen Abschnitt verlässt:

```
void lockup(mutex) {
    acquire(mutex.token);
    mutex.owner = self();
}
```

```
void unlock(mutex) {
    if (mutex.owner != self())
        raise(EPERM);
    mutex.owner = NULL;
    release(mutex.token);
}
```

In dem Beispiel liefert `self()` entweder eine ↑PID oder den ↑Prozesszeiger, und mit `raise()` wird eine ↑Ausnahme erhoben, nämlich dass der gegenwärtige Prozess eine für ihn unerlaubte Operation durchführt. Hinter dem Erwerb (*`acquire`*) und der Abgabe

(*`release`*) der Sperre verbirgt sich die semantisch äquivalente Operation des als Grundlage angenommenen Konzepts zum wechselseitigen Ausschluss (`P` und `V` im Falle des binären Semaphors oder `lockup` und `unlock` im Falle der Umlauf- oder Lese-/Schreibsperre). Dazu passend entspricht der „Spielstein“ (`token`) dem ↑Exemplar eines binären Semaphors beziehungsweise einer Umlauf- oder Lese-/Schreibsperre.
Dieses Beispiel geht davon aus, dass die ↑Ausnahmebehandlung richtigerweise die Termination des fehlgeleiteten Prozesses erzwingt und damit die Ausführung der gescheiterten Entsperroperation nicht wieder aufnimmt.

mutual exclusion (dt.) gegenseitiger oder ↑wechselseitiger Ausschluss.

Nachricht (en.) ↑*message*. Mitteilung, mit der ein ↑Prozess einem anderen Prozess in Bezug auf einen bestimmten Umstand die Kenntnis des neuesten Sachverhalts vermittelt (in Anlehnung an den Duden). Für gewöhnlich umfasst diese Mitteilung eine gewisse Menge von ↑Daten, denen ein bestimmter ↑Speicherbereich zugeordnet ist. Die Mitteilung kann aber auch „datenlos“ erfolgen, beispielsweise indem ein wartender Prozess deblockiert wird und dieser dann bei Wiederaufnahme eben davon ausgehen darf, dass ein anderer Prozess ihm die Möglichkeit zur Fortsetzung vermittelt hat. Letztere Form wird auch als ↑Signal bezeichnet.

Nachrichtenversenden (en.) ↑*message passing*. Art der Kommunikation, bei der ein ↑Prozess einem anderen Prozess eine ↑Nachricht explizit zusendet. Eine zwingende Maßnahme, wenn den kommunizierenden Prozessen kein ↑gemeinsamer Speicher für den Austausch von ↑Daten zur Verfügung steht. Dies ist typisch für ein ↑Rechnernetz. Aber auch trotz gemeinsamem Speicher kann die nachrichtenorientierte Interaktion von Prozessen geboten sein, nämlich im Falle einer zu schwachen ↑Speicherkonsistenz. Wenn der logische Ablauf einer bestimmten Folge von Speicheroperationen indeterministisch ist, müssen die betreffenden Prozesse durch ein wenigstens in ihrer Gruppe einheitliches Protokoll explizit den Zugriff auf die gespeicherten Daten regeln. Im Zuge der notwendigen ↑Synchronisierung, was häufig den Hauptanteil an ↑Gemeinkosten ausmacht, senden die beteiligten Prozesse die Daten (für die Konsistenz innerhalb der Gruppe gelten muss) oder auch nur die ↑Referenz darauf gleich mit.
Der Vorgang, eine Nachricht zu versenden, kann *synchron* oder *asynchron* für den betreffenden Prozess erfolgen. Damit ist gemeint, dass der Prozess auf die Entgegennahme der Nachricht warten (synchron) oder nicht warten (asynchron) muss. Entgegengenommen wird die Nachricht entweder von dem Prozess, der die weitere Verarbeitung vornehmen wird (synchon), oder von einer ↑Entität, die lediglich für eine Zwischenspeicherung sorgt (asynchron). Ein Prozess, der eine Nachricht entgegennimmt, tut dies für gewöhnlich immer synchron: nämlich in dem Moment, sobald er den Auftrag dazu angenommen hat.
Zusätzlich zum Aspekt der Synchronizität kann eine Nachricht *gepuffert* oder *ungepuffert* von einem zum anderen Prozess übertragen werden. Letzteres bedeutet üblicherweise, dass der mit der Kommunikation verbundene Datentransfer durchgehend (*end-to-end*) erfolgt, und zwar über eine Art ↑Speicherdirektzugriff direkt heraus aus dem (ggf. durch ↑Speicherschutz isolierten) ↑Adressraum des die Nachricht versendenden hinein in den (ggf. ebenso isolierten) Adressraum des diese Nachricht entgegennehmenden Prozesses. Das Kommunikationsmodell sieht keine Pufferung der kompletten Nachricht vor, gleichwohl kann für die Durchführung des Transfers aus technischen Gründen eine Zwischenspeicherung von Nachrichtenfragementen erfolgen. Solche Gründe ergeben sich für gewöhnlich bei der Kommunikation über ein Rechnernetz, aber beispielsweise auch zur Unterstützung von ↑Umlagerung: um den eine Nachricht entgegennehmenden Prozess unabhängig von ihrer Anwesenheit im ↑Arbeitsspeicher voranschreiten zu lassen, kann ihre Zwischenspeicherung in einen vom ↑Betriebssystem dafür vorgesehenen ↑Speicherbereich zweckmäßig sein. Demgegenüber bedeutet die gepufferte Übertragung einer (gesamten) Nachricht, keine Zwischenspeicherung aus technischen Gründen. Vielmehr ist die Pufferung im Kommunikationsmodell verankert.
Gepufferter Nachrichtentransfer dient vornehmlich der zeitlichen Entkopplung von Sende- und Empfangsprozess: der Sendeprozess muss damit nicht die Bereitwilligkeit des Empfangsprozesses abwarten, eine Nachricht entgegenzunehmen. So wird dadurch insbesondere auch

der asynchrone Versandt einer Nachricht ermöglicht (s.o.). Jedoch ist für letzteres die Nachrichtenpufferung nicht zwingend. Stattdessen kann das Kommunikationsmodell den zur Zusammenstellung einer zu sendenden Nachricht benötigten Speicherbereich als ↑wiederverwendbares Betriebsmittel explizit machen. Nach dem asynchronen Versenden der Nachricht kann der Sendeprozess durch ↑einseitige Synchronisation mit dem ↑Ereignis, das die Beendigung des Datentransfers anzeigt, auch ohne Zwischenspeicherung zeitlich unabhängig vom Empfangsprozess voranschreiten. Der Sendeprozess fragt von Zeit zu Zeit ab, ob dieses Ereignis bereits eingetreten ist und geht dazwischen anderen Dingen nach oder er blockiert sich auf die Ereignisanzeige durch den Empfangsprozess.

Name (en.) ↑*name*, ↑*label*. Bezeichnung; allgemein die kennzeichnende Benennung von einer bestimmten ↑Entität, insbesondere auch von dem ↑Exemplar eines Programmtextes oder -datums (d.h., von einem ↑Unterprogramm oder einer Programmvariablen). Die Benennung kann eine Nummer oder ein ↑Symbol sein.

name (dt.) Benennung, Bezeichnung, ↑Name; ↑Handhabe.

name binding (dt.) ↑Namensbindung.

name resolution (dt.) ↑Namensauflösung.

name space (dt.) ↑Namensraum.

Namensauflösung (en.) ↑*name resolution*. Vorgang, bei dem ein ↑Name umgewandelt wird in eine ↑numerische Adresse. Hier ist insbesondere der ↑Pfadname Ausgangspunkt für die Aufschlüsselung, die letztlich zur Lokalisierung der die gesuchte ↑Datei beschreibenden Datenstruktur (↑*inode*) im ↑Dateisystem führt.
Beginnt der Pfadname mit einem ↑Separator, verläuft die Suche von der Wurzel des Dateisystems (↑*root directory*) ausgehend: absolute Bezeichnung/Adressierung der Datei. Anderenfalls startet die Suche an der Stelle im ↑Namensraum, wo sich der ↑Prozess gegenwärtig aufhält (↑*current working directory*): relative Bezeichnung/Adressierung der Datei. Nacheinander werden die einzelnen Pfadabschnitte, jeder ein ↑Name gefolgt von einem Separator oder dem Textende, in dem jeweiligen Verzeichnis gesucht.
Jedes dieser Verzeichnisse ist eine spezielle Datei, die mit Hilfe der in ihrem ↑Indexknoten enthaltenen Informationen ausgelesen wird. So wird ein ↑Verzeichniseintrag nacheinander offengelegt und mit dem gesuchten Namen verglichen. Den Anfang nimmt der Knoten von dem ↑Wurzelverzeichnis beziehungsweise ↑Arbeitsverzeichnis. Entspricht der gesuchte Name einem Verzeichniseintrag, wird mit der verzeichneten ↑Indexknotennummer der nächste Indexknoten geladen. Ist der Indexknoten ↑Deskriptor einer weiteren Verzeichnisdatei und kann die Suche fortgesetzt werden, weil das Textende noch nicht erreicht ist, wiederholt sich der Ablauf. Im Fehlerfall (unbekannter Name, falscher Dateityp, ↑defekter Block) bricht die Suche ab. Am Textende angekommen, spiegelt der zuletzt geladene Indexknoten die gesuchte Datei wider.

Namensbindung (en.) ↑*name binding*. Vorgang, bei dem eine bindende Beziehung zwischen ↑Name und ↑numerische Adresse eingegangen wird; Verbindung zwischen ↑symbolische Adresse und numerische Adresse herstellen. Hier insbesondere die Einrichtung einer ↑Verknüpfung zwischen ↑Dateiname und einem ↑Exemplar der die betreffende ↑Datei beschreibenden Datenstruktur (↑*inode*) im ↑Dateisystem. Dazu ist ein ↑Verzeichniseintrag anzulegen und diesem Eintrag ist ein freier ↑Indexknoten zuzuordnen. Letzteres meint, die ↑Indexknotennummer in den Eintrag zu speichern und dadurch die ↑Bindung zu manifestieren.

Namenskontext (en.) ↑*naming context*. Sinnzusammenhang, in dem ein ↑Name steht; Bezugsrahmen von einem Namen. Der Kontext, in dem ein Name eindeutig ist. Die Eindeutigkeit sichert der Name selbst zu, indem er nicht noch einmal in dem Kontext definiert ist.

Namensraum (en.) ↑*name space*. Bezeichnung für eine Menge von eindeutigen Bezeichnungen, endlich. Jedes Element in dem in sich abgeschlossenen Raum ist ein ↑Name für eine ↑Entität. Dieser Raum entspricht einem ↑Adressraum, der den Kontext für eine ↑symbolische Adresse definiert und damit einen bestimmten, umfassenden ↑Namenskontext bildet. In dem Sinne ist ein Name, der letztlich ein ↑Symbol in einem noch zu übersetzenden/bindenden ↑Programm darstellt, gleichsam Synonym für eine besondere ↑virtuelle Adresse im korrespondierenden ↑Maschinenprogramm.
Die Namensmenge kann einen gegliederten oder ungegliederten Aufbau haben. Im gegliederten Fall ist die Menge als ↑hierarchischer Namensraum ausgeprägt, in dem die Teilepaare jeweils Namenskontexte sind und ein ↑Pfadname die bezeichnete Entität eindeutig identifiziert. Demgegenüber ist im ungegliederten Fall eine „flache Struktur“ typisch. In dem Fall muss der Name selbst die eindeutige Identifizierung der Entität sicherstellen.
Die hierarchische Auslegung ist in Anbetracht der typischerweise sehr großen Anzahl zu benennender Entitäten (↑Datei) insbesondere für ein ↑Dateisystem äußerst zweckmäßig. Hier müssen zudem die gespeicherten Informationen zur Struktur und ↑Verknüpfung von Namenskontexten auch in das Dateisystem selbst integriert sein (↑Persistenz). Erst dadurch ist es möglich, nach erfolgter ↑Befestigung eines Dateisystems die darin verzeichneten Dateien zu erkennen und zu gebrauchen.

naming context (dt.) ↑Namenskontext.

Nebengerätenummer (en.) ↑*minor device number*. Auf ↑UNIX zurückgehender darstellbarer abstrakter Begriff, eine Zahl, mit dessen Hilfe ein ↑Peripheriegeräts einer bestimmten ↑Klasse identifiziert wird. Diese Zahl ermöglicht es dem durch eine ↑Hauptgerätenummer identifizierten ↑Gerätetreiber dieser Klasse zwischen den verschiedenen, aber gleichartigen Geräten zu unterscheiden. Komponente einer ↑Gerätenummer.
Die allen Geräten derselben Klasse gemeinsamen Attribute sind für gewöhnlich in einem ↑Verbund zusammengefasst. Typische Attribute bilden die ↑Ein-/Ausgaberegister eines Geräts, dass heißt, deren ↑Adressen (↑*memory-mapped* I/O) oder ↑Namen (↑*port-mapped* I/O). Weitere Attribute geben beispielsweise Auskunft über den Gerätezustand, spezifizieren den ↑Puffer zur Weiterleitung von ↑Daten, die bei der ↑Ein-/Ausgabe anfallen oder identifizieren die ↑Prozessinkarnation, die das betreffende Gerät, gemeinhin ein ↑wiederverwendbares Betriebsmittel, aktiv in Benutzung hat.

nebenläufiger Prozess (en.) ↑*concurrent process*. Ein ↑gleichzeitiger Prozess, für den in Bezug auf alle anderen gleichzeitigen Prozesse im selben ↑Rechensystem ↑Nebenläufigkeit gilt.

Nebenläufigkeit (en.) ↑*concurrency*. Verhältnis von nicht kausal abhängigen Ereignissen, die sich also nicht beeinflussen. Ein Ereignis ist nebenläufig zu einem anderen, wenn es im Anderswo (d.h., weder in der Zukunft noch in der Vergangenheit) des anderen Ereignisses liegt, weder Ursache noch Wirkung ist: wenn zu dem anderen Ereignis keine Daten-, Kontrollfluss- oder Zeitabhängigkeit besteht.

Nebenläufigkeitssteuerung (en.) ↑*concurrency control*. Verfahren, das im Falle der ↑Nebenläufigkeit einzelner ↑Aktionen oder ↑Aktionsfolgen die Entwicklung korrekter Berechnungen durch den ↑Prozess sicherstellt. Wenigstens ein ↑gleichzeitiger Prozess nimmt zumindest streckenweise an den Berechnungen teil und bewirkt gegebenenfalls inkonsistente Berechnungszustände, wenn keine ↑Koordination der kritischen Einzelaktionen vorliegt.
Zur ↑Koordinierung dieser Aktionen ist ↑optimistische Nebenläufigkeitssteuerung geboten, wenn in dem Verfahren eine möglichst starke logische Entkopplung der aufeinander abzustimmenden Prozesse zu geschehen hat. Ein Beispiel dafür wäre die Anforderung, plötzliches ↑Versagen der in der Koordinierung selbst involvierten Prozesse besser oder überhaupt tolerieren zu können. Alternativ kann aber immer auch eine ↑pessimistische Nebenläufigkeitssteuerung zum Einsatz kommen — umgekehrt gilt dies nicht. Der pessimistische Ansatz ist insbesondere erforderlich, wenn die Art des von den zu koordinierenden Aktionen beanspruchten ↑Betriebsmittels die ↑Unteilbarkeit bestimmter Aktionsfolgen bedingt.

Netzwerk Vernetzung mehrerer voneinander unabhängiger elektronischer Geräte, die den Datenaustausch zwischen diesen ermöglicht; Kurzform: Netz (in Anlehnung an den Duden).

next fit (dt.) nächstbeste Passung, ↑Platzierungsalgorithmus.

nichtblockierende Synchronisation (en.) *non-blocking* ↑*synchronisation*. Art der ↑Synchronisation, bei der ein ↑Prozess in einer ↑Konkurrenzsituation *nicht* darauf warten muss, bis er an der Reihe ist, eine bestimmte ↑Aktion oder ↑Aktionsfolge durchzuführen. Die betreffende Aktion/Aktionsfolge ist für gewöhnlich als ↑Transaktion beschrieben, für die eine ↑optimistische Nebenläufigkeitssteuerung zur Absicherung erfolgt. Grundsätzlich ist diese Art der Synchronisation frei von ↑Verklemmungen.
Wesentlicher Aspekt dieser Synchronisationsform ist, dass ein in der Konkurrenzsituation mit anderen Prozessen ↑gekoppelter Prozess niemals oder nur zeitweilig die Kontrolle über seinen ↑Prozessor abgibt und in logischer Hinsicht weiterhin eigenständig ins weitere Geschehen eingreifen kann: sein ↑Prozesszustand modelliert eine Situation, in der eine bestimmte, durch den Prozess selbst unveränderliche Bedingung gilt, die ihn entweder als laufend oder als bereit ausweist. Ausschließlich die ↑mitlaufende Planung bewirkt für den betreffenden Prozess die eventuelle Abgabe der Kontrolle über den Prozessor und damit einen Zustandswechsel von laufend nach bereit, niemals aber nach blockiert. Gegenteil von ↑blockierende Synchronisation.

Exkurs Als Beispiel, um das Prinzip wie auch besondere Schwierigkeiten dieser Synchronisationstechnik zu verdeutlichen, dient eine ↑verkettete Liste, insbesondere die darauf definierten Operationen (vgl. S. 374). Präsentiert werden ↑CAS-basierte Lösungen, die, um sich auf das Wesentliche dieser Synchronisationstechnik besser konzentrieren zu können, bewusst nicht dem ↑ABA Problem vorbeugen.
Als erste Operation soll das Aneinanderfügen (*join*) zweier Kettenglieder betrachtet werden (für die sequentielle Variante vgl. S. 375):

```
chain_t *join(chain_t *this, chain_t *item) {
    do item->link = this->link;     /* create private copy of global link */
    while (!CAS(&this->link, item->link, item));  /* try to commit change */
    return item;                        /* point at the joined chain link */
}
```

Diese Implementierung geht davon aus, dass das einzufügende Kettenglied (`item`) im Moment der Verkettung eine ↑Eigenvariable darstellt, die von keinem anderen Prozess benutzt wird. Damit kann dieses Kettenglied zum einen die Kopie des zu manipulierenden globalen Kettenglieds aufnehmen und zum anderen zugleich seinen Nachfolger in der Liste speichern. Das anschließende CAS fügt dieses Kettenglied in die Liste ein, aber auch nur dann, wenn das Gelenk (`link`) an der Einfügestelle (`this`) noch unverändert ist. Scheitert das CAS, ist anzunehmen, dass zwischenzeitlich ein anderer Prozess das Gelenk genau an dieser Stelle bereits erfolgreich verändert hat. In dem Fall wiederholt der Prozess seinen Einfügeversuch, wobei dann das neue Gelenk als private Kopie in die Eigenvariable (`item`) als Referenzwert übernommen wird.
So eingängig diese Implementierung vielleicht auch sein mag, sie ist allerdings in zweierlei Hinsicht problematisch. Einerseits wurde eine bestimmte Strategie der Zusammenstellung zweier Kettenglieder angenommen, nämlich ↑LIFO, womit, wenn es um die Liste insgesamt geht, immer nur an einem Ende, dem Kopfelement (`this`), manipuliert wird. Allerdings kann mit dieser Operation die Manipulation auch an jeder anderen Stelle in der Liste geschehen. Damit ergibt sich, zweitens, das eigentliche Problem: das Verkettungsglied (`this`), an das das neue Element (`item`) durch einen Prozess angehängt werden soll, könnte gleichzeitig durch einen anderen Prozess gelöscht werden. Die Folge davon wäre, dass das angehängte Element mit dem gelöschten Element verschwindet und eben nicht wie gedacht auf der Liste verzeichnet ist.
Die Lösung zu dieser „einfüge-löschen“ ↑Wettlaufsituation besteht darin, die Aktionsfolge zum Löschen eines Kettenglieds für andere Prozesse explizit zu machen und auzuzeigen. Eine

Möglichkeit besteht etwa darin, indem das Gelenk (`link`) eben des zu löschenden Listenelements (`this`) beim Löschen als „zerbrechlich“ (*frail*) markiert wird. Um solche Markierungen setzen, erkennen und auch wieder aufheben zu können, dienen folgende Hilfskonstrukte:

```
typedef enum flag {
    FRAIL = (1 << 0)                        /* pointer should not be used */
} flag_t;

inline void *mark(void *item, flag_t flag) {
    return (void *)((long)item | flag);        /* deliver flagged pointer */
}
inline bool just(void *item, flag_t flag) {
    return (long)item & flag;                   /* deliver pointer marker */
}
inline void *pure(void *item) {
    return (void *)((long)item & ~FRAIL);     /* deliver purified pointer */
}
```

Mit dieser Hilfskonstruktion ist es möglich, einen ↑Zeiger zu markieren (*mark*), ihn damit aber auch zu verfälschen, seinen Zustand, den er gerade (*just*) inne hat, abzufragen und ihn auch wieder als reines, echtes (*pure*) ↑Exemplar wie gewohnt zu verwenden. Für die Markierung wird das niederwertigste Bit eines Zeigerwertes verwendet, damit können alle Zeiger behandelt werden, die auf Objekte verweisen, deren jeweilige Größe zwei ↑Bytes und mehr entspricht und die jeweils eine gerade ↑Adresse haben. Für `chain_t`-Objekte wird dies hier angenommen — da ein optimierender ↑Kompilierer für gewöhnlich Vorkehrungen dafür treffen wird, dass diese Objekte eine gerade Speicheradresse erhalten (↑*alignment*).
Damit würde die eingangs skizzierte Operation zum Aneinanderfügen (*join*) zweier Kettenglieder wie folgend gezeigt korrigiert:

```
chain_t *join(chain_t *this, chain_t *item) {
    do {                            /* try safe insertion of a chain link */
        item->link = this->link;    /* create private copy of global link */
        if (just(item->link, FRAIL))                  /* valid chain link? */
            return 0;                /* no, indicate failure of operation */
    } while (!CAS(&this->link, item->link, item));       /* try to commit */
    return item;                        /* point at the joined chain link */
}
```

Sollte das kritische Gelenk (`item->link`) markiert sein, das heißt, wird sein Kettenglied (`this`) gerade gelöscht, scheitert die Operation. Hier wird der Einfachheit halber davon ausgegangen, dass die aufrufende Ebene in dem Fall darauf mit der Wiederholung dieser Operation reagiert — und dies so oft tut, bis die Operation gelingt.
Damit lässt sich nunmehr die Löschoperation formulieren, die effektiv nichts weiter tut, als das zu löschende Kettenglied in der Liste zu übergehen (*skip*):

```
chain_t *skip(chain_t *this) {
    chain_t *item;                          /* chain link to be removed */
    for (;;) {                            /* try to unhook a chain link */
        if ((item = pure(this->link)) == 0))     /* nothing more to skip? */
            break;                        /* yes, someone beat me to it */
        chain_t *next = pure(item->link);           /* point at successor */
        CAS(&item->link, next, mark(next, FRAIL));        /* mark at link */
        if (CAS(&this->link, item, next))                /* try to unhook */
            break;                            /* chain link was unhooked */
        CAS(&item->link, mark(next, FRAIL), next); /* not unhooked, unmark */
    }
    return item;                                /* deliver removed entry */
}
```

Die eigentliche Operation, die das fragliche Kettenglied aus der Liste entfernt, besteht aus den drei CAS-basierten Schritten. Mit dem ersten CAS wird das auszuhängende Kettenglied (`item`) markiert. Ab dem Moment wird das Einhängen eines anderen Kettenglieds an dieser Stelle scheitern. Mit dem zweiten CAS wird das auszuhängende Kettenglied übersprungen und damit letztlich aus dem Listenabschnitt (`this`) entfernt. Schließlich wird mit dem dritten CAS die Markierung des nun ausgehängten Kettenglieds wieder gelöscht.
Auffällig ist, dass die Markierung nur gelöscht wird, wenn das fragliche Kettenglied nicht ausgehängt werden konnte (d.h., wenn das zweite CAS scheitert): da das Kettenglied auf der Liste bleibt, muss die Markierung auch wieder zurückgenommen werden. Wurde das Kettenglied jedoch ausgehängt, bleibt die Markierung bestehen. In dem Fall wird davon ausgegangen, dass eine Wiederverwendung dieses Kettenglieds seinem Gelenk (`link`) eine normale (unmarkierte) Adresse zuweist und den Zeiger damit wieder stabilisiert.
Jedes einzelne CAS könnte scheitern und damit keine Veränderung herbeiführen. Scheitert das erste CAS, dann ist anzunehmen, dass das betreffende Kettenglied (`item`) vor einem anderen Prozess schon entfernt und woanders (in eine andere Liste) eingehängt wurde. In dem Fall darf auch keine Markierung vorgenommen werden. Als Folge würde zudem das zweite CAS scheitern, da das Gelenk (`link`) an der Aushängestelle (`this`) nicht mehr auf das zwischenzeitlich entfernte Kettenglied verweisen würde. Scheitert das dritte CAS, dann könnte es (wie möglicherweise schon zuvor, s.o.) zur Überholung der Aushängeoperation durch einen anderen Prozess gekommen sein. Auch dann darf das Kettenglied nicht verändert werden.
Die Aufrechterhaltung der Markierung eines ausgehängten Kettenglieds hat zur Folge, dass, im Falle gleichzeitiger Aushängeversuche zweier direkt benachbarter Kettenglieder, der (zeitversetzte) Prozess, der das zweite (nachgestellte) Kettenglied appliziert, solange in der Schleife verbleibt, wie sich der Gelenkzustand eben dieses Kettenglieds nicht ändert, das heißt, „zerbrechlich“ bleibt. Der Zeiger auf dieses Kettenglied wird dem anderen (vorangehenden) Prozess jedoch als Ergebnis geliefert — womit sich eine Wettlaufsituation ergeben kann, wenn dieser Prozess das erhaltene (soeben ausgehängte) Kettenglied beispielsweise einer anderen Liste hinzufügt und damit dem Gelenk einen normalen (unmarkierten) Zeigerwert zuweist. In dem Moment würde der zeitversetzte Prozess seinen wiederholten Aushängeversuch durchaus erfolgreich beenden können — und damit gegebenenfalls fälschlicherweise ein Kettenglied aus einer beliebigen anderen Liste entfernen. Eine Lösung dieser Wettlaufsituation ist mangels Kontextwissen (bspw. zum Vorgänger des applizierten Kettenglieds) hier auf dieser Ebene nicht möglich — aber auf der aufrufenden Ebene, indem zur konsistenten Wiederverwendung des fraglichen Kettenglieds im Falle eines „zerbrechlichen“ Gelenkzustands ein Protokoll verfolgt wird, das die alte Nachbarschaftsbeziehung (hier: `this` und `this->link`) nach erfolgter Aushängeoperation noch untersuchen kann und im Konfliktfall den betreffenden Listenabschnitt in geeigneter Weise korrigiert. TBC
Die nächste Operation, das Zerlegen (*part*) einer Liste in zwei Teile, gestaltet sich gegenüber dem Einfügen (*join*) und Aushängen (*skip*) eines Kettenglieds vergleichsweise einfach:

```
inline chain_t *part(chain_t *this) {
    chain_t tail;
    do tail = this->link;           /* create private copy of global link */
    while (!CAS(&this->link, tail, NULL));        /* try to commit change */
    return pure(tail);             /* point to rear detached list section */
}
```

Hier zeigt sich die typische Struktur einer einfachen transaktional ausgelegten Datenmanipulation: zunächst wird eine lokale Kopie des zu verändernden globalen Datums angelegt, um dann, bei unverändertem Zustand, die Änderung durchzuführen und zu bestärtigen. Als Ergebnis wird der abgetrennte hintere Teil der Liste geliefert.
Diese Implementierung lässt sich allerdings stark vereinfachen. Bei genauer Betrachtung finden hier nur zwei wesentliche Aktionen statt: zum einen das Abrufen (*fetch*) des aktuellen und zum anderen das Einspeichern (*store*) eines neuen Wertes des Gelenks (`link`) im applizierten Kettenglied (`this`). Diese beiden Aktionen müssen jedoch eine ↑unteilbare Anweisung

bilden, was mit ↑FAS der Fall ist. Entsprechend lässt sich die Funktion umschreiben wie folgt:

```
inline chain_t *part(chain_t *this) {
    return pure(FAS(&this->link, NULL));   /* separate & deliver rear end */
}
```

Im Gegensatz zu allen vorher betrachteten Lösungen, die den Prozessen lediglich ↑Sperrfreiheit garantieren, bietet diese hier mit ↑Wartefreiheit eine stärke ↑Fortschrittsgarantie.
Schließlich noch die Operation, um das Kettenglied anhand eines als Gelenk angegebenen Attributs aufzufinden (*find*):

```
inline chain_t *find(chain_t *this, chain_t *item) {
    chain_t *next, *root = this;
    while (this && ((next = this->link) != item))
        this = just(next, FRAIL) ? root : next;
    return this;              /* deliver chain link of given item or NULL */
}
```

Die Suche selbst nimmt keine Veränderungen an der Liste vor, muss allerdings damit umgehen können, dass bei den Suchläufen Änderungen durch andere Prozesse vorgenommen werden. Besonderes Augenmerk gilt möglicherweise gleichzeitig stattfindenden Aktionen zum Aushängen (*skip*) eines oder mehrerer Kettenglieder. Hierzu wird geprüft, ob das jeweils betrachtete Kettenglied als „zerbrechlich“ (`FRAIL`) markiert ist. Wird ein solches Kettenglied erfasst, beginnt die Suche erneut von vorne (`root`).
Andere Lösungen, die im Falle gleichzeitiger Löschoperationen, nicht mit der Suche erneut starten, sind komplizierter und gegebenenfalls auch nicht effizienter. Eine einfache Alternative besteht etwa im Abbruch der Suche mit einem Fehlerkode — mit dem Effekt, dass dann die aufrufende Ebene die Suche erneut auslöst, damit aber auch die Kontrolle über die Anzahl derartiger Versuche erhält.
Dieser Exkurs zeigt, dass ein auf nichtblockierende transaktionale Konzepte basierendes ↑nichtsequentielles Programm nicht unbedingt kompliziert sein muss (wie etwa das FAS-basierte *part*, aber auch durchaus *join* und *find*), dass dies aber eben auch kein Regelfall ist. Nicht nur der allgemein höhere Schwierigkeitsgrad, mit dem die Entwicklung solcher Programme konfrontiert ist, bereitet Probleme, auch die zur ↑Laufzeit dann gegebenenfalls gehäuft (↑*bus-lock burst*) zur Ausführung kommenden atomaren Befehle (hier: CAS und FAS), die sich negativ auf die Performanz auswirken und starke ↑Interferenz verursachen können. Damit soll jedoch nicht etwa ein Plädoyer für blockierende Synchronisation gehalten werden, denn auch diese ist auf tieferer Ebene mit genau denselben Problemen behaftet. Als Frage stellt sich daher vielmehr, ab welcher ↑Abstraktionsebene die Untiefen möglicher ↑Gleichzeitigkeit bei der Programmausführung verborgen gehalten werden soll.

nichtflüchtiger Speicher (en.) ↑*non-volatile memory*. Klassifikation für einen ↑Speicher, der seinen Inhalt auch ohne Anliegen einer Betriebsspannung für längere Zeit erhält; ↑Kernspeicher, ↑Festwertspeicher; ↑Sekundärspeicher oder ↑Tertiärspeicher. Neben ↑Wechseldatenträger kommen davon heute (2025) vor allem gerade Magnetplatte und ↑SSD als fest in einem ↑Rechensystem eingebaute Datenträger zum Einsatz.

nichtflüchtiges Register (en.) ↑*non-volatile register*. Konzept der ↑Aufrufkonvention: der Inhalt eines solchen Prozessorregisters gilt als beständig. Vorkehrungen zur Sicherung und Wiederherstellung des Registerinhalts werden im ↑Unterprogramm bei Bedarf getroffen (*callee-saved*).

nichtperiodische Aufgabe (en.) ↑*non-periodic task*. Eine termingebundene ↑Aufgabe, die keiner ↑Periode fest zugeordnet ist oder werden kann und deren ↑Termineinhaltung locker (d.h., weich oder fest) erfolgen kann oder strikt vorgegeben ist.. Gegenteil von ↑periodische Aufgabe; zusammenfassende Bezeichnung für ↑aperiodische Aufgabe oder ↑sporadische Aufgabe.

nichtsequentieller Prozess (en.) ↑*non-sequential process*. Ein ↑Prozess, der durch ein ↑nichtsequentielles Programm definiert ist und mehr als einen ↑Handlungsstrang aufweist. Jeder

dieser Handlungsstränge ist im Grunde genommen ein ↑sequentieller Prozess, der allerdings zeitversetzt oder gleichzeitig mit anderen zusammen stattfindet.

nichtsequentielles Programm (en.) ↑*non-sequential program*. Bezeichnung für ein ↑Programm, das Konstrukte zur Formulierung explizit paralleler Abläufe verwendet. Die Konstrukte können in der Programmiersprache, in der das Programm formuliert wurde, enthalten sein oder werden sprachunabhängig durch eine Programmbibliothek (z.B. *POSIX Threads* bzw. *pthreads*; aber auch UNIX Signale, `signal(3)`) bereitgestellt.

NMI Abkürzung für (en.) *non-maskable* ↑*interrupt*.

no-op instruction (dt.) ↑Leerbefehl.

non-preemptive critical section (dt.) ↑unverdrängbarer kritischer Abschnitt.

non-sequential process (dt.) ↑nichtsequentieller Prozess.

non-sequential program (dt.) ↑nichtsequentielles Programm.

non-volatile memory (dt.) ↑nichtflüchtiger Speicher.

non-volatile register (dt.) ↑nichtflüchtiges Register.

non-write-allocate siehe (en.) ↑*write-allocate*.

non-periodic task (dt.) ↑nichtperiodische Aufgabe.

Notbehelf (en.) ↑*workaround*. Etwas für einen bestimmten Zweck nur bedingt Geeignetes, was ersatzweise benutzt wird, wenn etwas Besseres nicht verfügbar ist (Duden).

Notizblockspeicher (en.) ↑*scratchpad memory*. Bezeichnung für einen ↑Speicher in der ↑CPU, der komplett durch Software verwaltet wird. Einem ↑Zwischenspeicher sehr ähnlich, jedoch mit dem ↑Speicherwort als kleinste Verwaltungseinheit. Neben Einzelwortzugriffe ermöglicht die CPU typischerweise durch ↑DMA den Transfer großer zusammenhängender Blöcke von ↑Daten vom/zum ↑Hauptspeicher Die ↑Zugriffszeit auf ein einzelnes Datum entspricht der ↑Taktfrequenz der CPU. Seine Kapazität reicht von 64 ↑Byte (Fairchild F8, 1975) bis zu einem als Notizblock- oder Zwischenspeicher konfigurierbaren Anteil von 16 GiB (Intel Knights Landing, 2013).

NPCS Abkürzung für (en.) ↑*non-preemptive critical section*.

numeric punch card (dt.) ↑Lochkarte.

numerical address (dt.) ↑numerische Adresse.

numerische Adresse (en.) ↑*numerical address*. Eine sich nur aus Ziffern zusammensetzende und eine Zahl darstellende↑Adresse. Beispiele dafür sind ↑reale Adresse, ↑logische Adresse oder ↑virtuelle Adresse und ↑Gerätenummer, ↑Indexknotennummer oder ↑Blocknummer.

Nutzdaten Bezeichnung für ↑Daten, die bei Durchführung einer bestimmten ↑Aufgabe gebraucht (insb. auch Eingabe), verarbeitet, erzeugt, gespeichert und dargestellt (insb. auch Ausgabe) werden.

NVRAM Abkürzung für (en.) *non-volatile RAM*, (dt.) nichtflüchtiger ↑RAM.

Obergrenze (en.) ↑*ceiling*. Im Zusammenhang mit den auf ↑Prioritätsobergrenzen basierenden Verfahren zur Vorbeugung der *unkontrollierten* ↑Prioritätsumkehr die Bezeichnung für die obere Grenze der ↑Priorität für ein bestimmtes ↑unteilbares Betriebsmittel, R. Ein Betriebsmittelattribut, das der Priorität des höchst priorisierten ↑Prozesses, der R sperren könnte,

entspricht (↑Grenzpriorität). Dieser Prozess gehört der Menge von Prozesse an, die R gemeinsam haben.
Die Bestimmung des auch *Betriebsmittelobergrenze* genannten Werts benötigt ↑Vorwissen zu den Prozessen und den von ihnen angeforderten unteilbaren Betriebsmitteln. Für gewöhnlich geschieht die Festlegung der jeweiligen Grenzwerte (*capping*) vor ↑Laufzeit der betreffenden ↑Anwendungsprogramme: die Werte bleiben zur Laufzeit konstant (statisch).
Demgegenüber bestimmt der höchste Grenzwert aller gegnwärtig gesperrten unteilbaren Betriebsmittel die sogenannte *Systemobergrenze*. Dieser Grenzwert ist variabel (dynamisch), seine Einstellung geschieht zu den Bedingungen des auf Prioritätsobergrenzen basierenden Protokolls zur Laufzeit, nämlich wenn es zur Sperrung eines unteilbaren Betriebsmittels kommt. Ist keine einzige ↑Betriebsmittelsperre aktiv, liegt die Systemobergrenze unterhalb der niedrigsten Prozesspriorität.

Obergrenzensperre (en.) ↑*ceiling blocking*. Im Zusammenhang mit den auf ↑Prioritätsobergrenzen basierenden Verfahren zur Vorbeugung der *unkontrollierten* ↑Prioritätsumkehr die Bezeichnung für eine ↑Prozesssperre, erhoben bei der Betriebsmittelzuteilung, wenn nämlich ein freies (ungesperrtes) ↑unteilbares Betriebsmittel dem anfordernden ↑Prozess nicht zugeteilt wird. Diese Sperre wird gesetzt, falls (1) die ↑Priorität des anfordernden Prozesses nicht größer ist als die ↑Obergrenze des Systems und (2) dieser Prozess kein unteilbares Betriebsmittel belegt (sperrt), dessen Obergrenze der des Systems entspricht. Die Sperre wird aufgehoben, sobald die Obergrenze des Systems abgefallen und kleiner oder gleich der Priorität des gesperrten Prozesses ist.

object (dt.) Ding, Einheit, Gegenstand, ↑Objekt.

object code (dt.) ↑Maschinenkode.

object module (dt.) ↑Objektmodul.

object program (dt.) ↑Maschinenprogramm, Objektprogramm, Zielprogramm.

object program level (dt.) ↑Maschinenprogrammebene.

object type (dt.) ↑Objekttyp.

object-oriented (dt.) ↑objektorientiert.

Objective-C Programmiersprache: imperativ, objektorientiert (1981). Obermenge von ↑C.

Objekt (en.) ↑*object*. Gegenstand, auf den das Handeln von einem ↑Subjekt gerichtet ist (in Anlehnung an den Duden); eine (nicht näher beschriebene) ↑Entität, die eine bestimmte physische Ausdehnung und eine bestimmte Form hat.
Im programmiersprachlichen Sinne ist mit dieser *Einheit* sowohl ein Satz von *Operationen* als auch ein *Zustand* verbunden. Das Ergebnis dieser Operationen ist damit nicht nur bestimmt durch deren Operanden (d.h., Argumente, Parameter), sondern insbesondere auch durch den im Moment der Operation definierten Zustand. Diesbezüglich unterscheiden sich diese Operationen von herkömmlichen Funktionen, die nämlich gemeinhin kein „Gedächtnis“ haben und vollständig durch ihre Argumente bestimmt sind.
Für gewöhnlich bildet diese Einheit das ↑Exemplar eines bestimmten (ordinalen, elementaren oder komplexen bzw. zusammengesetzten) ↑Datentyps. Jedes solcher Exemplare belegt eine bestimmte Region im ↑Speicher, es hat damit eine definierte ↑Adresse und erstreckt sich über einen bestimmten, zusammenhängenden ↑Adressbereich.

Objekt erster Klasse (en.) ↑*first-class object*. Exemplar eines bestimmten Bautyps, das im ↑Arbeitsspeicher abgelegt (d.h., einer ↑Variablen zugewiesen) werden, ↑tatsächlicher Parameter beziehungsweise Funktionsergebnis sein oder zur ↑Laufzeit erstellt werden kann und eine eigene Identität besitzt. Für solch ein Objekt oder Konstrukt gibt es in seinem ↑Bezugssystem (d.h., ↑Programm) keine Einschränkung in der Erzeugung oder Nutzung.

Objektkode (en.) ↑*object code*. Resultat der ↑Übersetzung, insbesondere durch ↑Assemblieren, von den in einem ↑Quellmodul enthaltenen Konstrukten für ein ↑Programm. Anweisungen, die überwiegend in Form von ↑Maschinenkode vorliegen, aber auch vorinitialisierte Daten umfassen. Der für ein ↑Maschinenprogramm zur Ausführung (in den ↑Arbeitsspeicher zu ladende) relevante Anteil von Text- und Datenbeständen in einem ↑Objektmodul.

Objektmodul (en.) ↑*object module*. ↑Datei mit einem ↑Programm als Eingabe für einen ↑Binder, gleichfalls aber auch Ausgabedatei von einem ↑Assemblierer. Neben ↑Text oder ↑Daten, ist die ↑Symboltabelle wichtigstes Merkmal dieser Datei.

objektorientiert (en.) ↑*object-oriented*. Eigenschaft eines informatischen Systems, nämlich eines ↑Programms, auf der Idee, Verwendung von (gleichberechtigten, selbstständig arbeitenden) ↑Objekten zu beruhen (in Anlehnung an den Duden). Genau genommen sind damit jedoch drei wesentliche und ineinandergreifende Prinzipien gemeint:

1. Objekte. ↑Entitäten, mit denen ein wohldefinierter Satz von Operationen verknüpft ist und deren Zustand den Effekt dieser Operationen speichert.
2. ↑Klassen. Vorlagen, aus denen die Objekte als ↑Exemplare eines wohldefinierten↑Datentyps explizit erzeugt werden können.
3. ↑Vererbung. Ein Mechanismus, mit dem eine ↑Klassenhierarchie gebildet wird, um die Mitbenutzung (*sharing*) von ↑Ressourcen oder Merkmalen einer Oberklasse in einer Unterklasse zu ermöglichen.

Fehlen das Klassenkonzept, um gleichartige Objekte zu ermöglichen, und der Vererbungsmechanismus, um Klassenhierarchien inkrementell entstehen zu lassen, ist das betreffende System bestenfalls *objektbasiert*. Klassen wie auch Vererbung müssen daher in der Programmiersprache, in der das informatische System formuliert ist, verankert sein. Beispiele solcher Sprachen sind ↑Java und ↑C++, nicht jedoch ↑C, die lediglich dazu beiträgt, objektbasierte Programme zu formulieren — sehr wohl aber Zielsprache sein kann für einen ↑Kompilierer (z.B. `cfront`), der Programme einer objektorientierten Programmiersprache übersetzt.

Objekttyp (en.) ↑*object type*. Synonym von ↑Klasse.

off time (dt.) ↑Ruhezeit.

off-line scheduling (dt.) ↑vorlaufende Planung.

offset (en.) ↑Versatz.

Oktett (en.) ↑*byte*. Bezeichnung für ein 8-Bit ↑Byte.

on-demand paging (dt.) ↑Einzelanforderung.

on-line scheduling (dt.) ↑mitlaufende Planung.

opcode Abkürzung für (en.) ↑*operation code*.

OpenVMS Mehrplatz-/Mehrbenutzerbetriebssystem, erste Installation 1977 (↑DEC ↑VAX). Eine Weiterentwicklung von ↑VMS insbesondere für Prozessoren der Klasse ↑Alpha. Darüberhinaus erfolgten Portierungen auf 64-Bit Prozessoren von Intel (Itanium) und AMD (x86-64).

operating mode (dt.) ↑Arbeitsmodus, ↑Betriebsart; ↑Betriebsmodus.

operating station (dt.) ↑Bedienstation.

operating system (dt.) ↑Betriebssystem.

operating system machine level (dt.) ↑Betriebssystemebene.

operating-system kernel (dt.) ↑Betriebssystemkern.

operation code (dt.) ↑Operationskode. Abkürzung ↑*opcode*.

operation principle (dt.) ↑Operationsprinzip.

Operationskode (en.) ↑*operation code*. Identifikation für einen Befehl einer (realen/virtuellen) Maschine. Beispielsweise der ↑Maschinenkode, der den ↑Maschinenbefehl einer CPU spezifiziert. Allgemein aber die Nummer des Befehls, der von einer Maschine durchgeführt werden soll.

Operationsprinzip (en.) ↑*operation principle*. Definition des funktionellen Verhaltens der ↑Architektur einer (realen/virtuellen) Maschine durch Festlegung einer ↑Informationsstruktur für diese Maschine und einer ↑Kontrollstruktur für den Ablauf von einem ↑Programm. Beide Strukturformen definieren die ↑Rechnerarchitektur.

operator (dt.) Bedienungsperson. Jemand, dessen Aufgabe es ist, maschinelle Anlagen oder Rechner zu kontrollieren und zu bedienen (Duden); Operateur.

operator panel (dt.) ↑Bedienpult, Bedienungsfeld.

optimistische Nebenläufigkeitssteuerung (en.) *optimistic* ↑*concurrency control*. Bezeichnung für eine ↑Nebenläufigkeitssteuerung, in der jeder ↑Prozess die kritische ↑Aktion oder ↑Aktionsfolge als eine ↑Transaktion stattfinden lässt. Scheitert die Transaktion für einen Prozess, wird sie von ihm wiederholt bis sie gelingt oder der Prozess eine ↑Ausnahme erhebt. Typisches Beispiel für solch ein Ablaufmuster ist die ↑nichtblockierende Synchronisation. Dem Ansatz liegt die Annahme zugrunde, wonach ein Prozess mit anderen Prozessen nicht in Konflikt gerät, da eine gemeinsame und gleichzeitige Transaktion eher unwahrscheinlich ist. Gegenteil von ↑pessimistische Nebenläufigkeitssteuerung.

orthogonaler Befehlssatz Bezeichnung für einen ↑Befehlssatz (↑ISA), bei der jeder ↑Maschinenbefehl jede ↑Adressierungsart verwenden kann. In solch einer ↑Rechnerarchitektur variieren Befehlstyp und Adressierungsart unabhängig voneinander. Ihr Befehlssatz macht es nicht zur Auflage, dass bestimmte Befehle nur spezielle ↑Register verwenden dürfen.

Ortstransparenz Eigenschaft, durch die einem ↑Subjekt (↑Prozess) der tatsächliche Ort von einem ↑Objekt verborgen bleibt. Der für den Zugriff verwendete ↑Name enthält keinen Hinweis darüber, ob das Objekt sich mit dem Subjekt denselben ↑Adressraum, ↑Rechenkern oder ↑Rechner teilt. Ein solcher Name repräsentiert eine ↑symbolische Adresse.

OS Abkürzung für (en.) ↑*operating system*.

overhead (dt.) ↑Betriebslast, ↑Gemeinkosten.

overlapped I/O (dt.) ↑überlappte Ein-/Ausgabe.

overlay (dt.) ↑Überlagerung.

overlay memory (dt.) ↑Überlagerungsspeicher.

own variable (dt.) ↑Eigenvariable.

P Abkürzung für (hol., Kunstwort) *prolaag*, für (dt.) erniedrigen; (en.) *down*, *wait*, *acquire*. Neben ↑V die andere elementare Operation eines ↑Semaphors.

page (dt.) ↑Seite.

page descriptor (dt.) ↑Seitendeskriptor.

page fault (dt.) ↑Seitenfehler.

page fault probability (dt.) ↑Seitenfehlerwahrscheinlichkeit.

page frame (dt.) ↑Seitenrahmen.

page frame number (en.) ↑Seitenrahmennummer.

page number (en.) ↑Seitennummer.

page table (dt.) ↑Seitentabelle.

page thrashing (dt.) ↑Seitenflattern.

paged address range (dt.) ↑seitennummerierter Adressbereich.

paged segmentation (dt.) ↑seitennummerierte Segmentierung.

paged virtual memory (dt.) ↑seitennummerierter virtueller Speicher.

pager (dt.) ↑Seitenabrufer.

paging (dt.) ↑Seitenverfahren; ↑Seitenumlagerung; ↑Seitenadressierung, -nummerierung.

paging unit (dt.) ↑Seitenadressierungseinheit.

panic (dt.) ↑Panik.

Panik (en.) ↑*panic*. Gefahr für das ↑Rechensystem bedeutende und bei einer ↑Aktion im ↑Betriebssystem aufgetretene ↑Ausnahmesituation, die nicht behandelbar ist und den sofortigen Stillstand des Rechensystems bewirkt.

parallel computer (dt.) ↑Parallelrechner.

parallel processing (dt.) ↑Parallelverarbeitung.

paralleler Prozess siehe ↑nichtsequentieller Prozess.

paralleles Programm siehe ↑nichtsequentielles Programm.

Parallelität ↑Nebenläufigkeit, wenn in einem ↑Rechensystem die Unabhängigkeit von Vorgängen gegeben ist. Dabei ist ein solcher Vorgang ein ↑Prozess, der durch Vervielfachung oder Mehrfachnutzung von einem ↑Prozessor echt- oder pseudoparallel zu anderen Prozessen stattfindet. Voraussetzung dabei ist die Fähigkeit zur ↑Parallelverarbeitung durch ein ↑Betriebssystem.

Parallelrechner (en.) ↑*parallel computer*. Bezeichnung für einen ↑Rechner, der aus zwei oder mehreren realen ↑Prozessoren aufgebaut ist und echte ↑Parallelverarbeitung unterstützt. Neben der Anzahl der Prozessoren, die millionenfache ↑Parallelität bedeuten kann — etwa Tihane-2: 32 000 Mehrkernprozessoren (Intel Xeon) jeweils 12-fach parallel (384 000 Kerne) plus 48 000 Hochleistungsprozessoren (Intel Xeon Phi) jeweils 57-fach parallel (2 736 000 Kerne), zusammen 3 120 000 Kerne — und einen entsprechend komplexen Systemaufbau mit sich bringt, spielt die Organisation von dem ↑Arbeitsspeicher eine zentrale Rolle in solchen Rechnern. Die Prozessoren können speicher- oder nachrichtengekoppelt sein. Letzteres impliziert für jeden ↑Prozess, dass der Austausch von ↑Daten mit anderen Prozessen immer nur indirekt geschehen kann (↑*message passing*). Demgegenüber bedeutet die speichergekoppelte Organisation der Prozessoren die Möglichkeit zum direkten Datenaustausch (↑*shared memory*), also ohne zwingend ↑Nachrichtenversenden betreiben zu müssen. In diesem Fall ist jedoch auf die ↑Speicherkonsistenz zu achten: für gewöhnlich ist in solchen Konstellationen nicht mehr garantiert, dass die Speicheroperationen von allen Prozessoren in derselben (sequentiellen) Reihenfolge sichtbar sind. So kann es zum Konflikt (↑*race condition*) kommen, wenn von verschiedenen Prozessoren aus auf dieselbe (Inkohärenz) oder eine andere (Inkonsistenz) gemeinsame ↑Variablen zugegriffen wird. Ein ↑nichtsequentielles Programm für

solche Rechner muss daher Grad und Art der durch die Hardware zugesicherten *Kohärenz* im ↑Zwischenspeicher und *Konsistenz* im ↑Arbeitsspeicher reflektieren. Sind die Zusicherungen zu schwach, müssen in dem Programm selbst Vorkehrungen zur Durchsetzung des jeweils geforderten Stärkegrads getroffen werden. Dies kann so weit gehen, speichergekoppelte Rechner als ein nachrichtengekoppeltes (verteiltes) System aufzufassen und grundsätzlich direktem Datenaustausch vorzubeugen.

Parallelverarbeitung (en.) ↑*parallel processing.* ↑Betriebsart von einem ↑Rechensystem, durch die mehrere Abläufe von demselben ↑Programm oder verschiedener Programme parallel stattfinden können. Erreicht wird dies durch Vervielfachung oder Mehrfachnutzung des Prozessors, das heißt, räumliche oder zeitliche Partitionierung seiner Verarbeitungseinheiten. Ersteres begründet beispielsweise einen ↑Multiprozessor oder ↑Mehrkernprozessor, letzteres ist durch ↑partielle Virtualisierung eben nur der Verarbeitungseinheit erreichbar. Sind mehrere Abläufe in demselben Programm möglich, wird letzteres als *nichtsequentiell* bezeichnet.

paravirtualisation (dt.) ↑Paravirtualisierung.

Paravirtualisierung (en.) ↑*paravirtualisation.* Form der ↑Selbstvirtualisierung: die ↑Virtualisierung erfolgt „nebenher“ oder „entlang“ (gr. *para*) der gewöhnlichen Funktion von einem ↑Betriebssystem. Im Unterschied zur ↑Vollvirtualisierung ist dem Betriebssystem die Tatsache bekannt, dass die Hardware (↑CPU, ↑Peripherie), die es eigentlich verwaltet, virtualisiert wird. So setzt das Betriebssystem, wenn ein ↑sensitiver Befehl durch die (virtuelle) CPU ausgeführt werden soll, einen Aufruf an den ↑Hypervisor ab, der die Ausführung dieses Befehls dann so vornimmt, dass keine andere ↑virtuelle Maschine dadurch beeinträchtigt wird. Beispiel für solch einen Befehl ist etwa die ↑Unterbrechungssperre einer CPU (`cli` bei ↑x86): wäre ein solcher Befehl nicht virtualisierbar, würde seine direkte Ausführung in einer virtuellen Maschine die Unterbrechungsperre der realen Maschine setzen und damit den weiteren Betrieb anderer virtueller Maschinen zeitweilig aussetzen. In dem Fall sperrt der Hypervisor eine ↑Unterbrechung nur in der virtuellen Maschine, für die die Unterbrechungssperre wirken soll. Der Hypervisor unterbindet damit also keine ↑Unterbrechungsanforderung, die an die reale Maschine (d.h., CPU) geht. Andere Beispiele dieser Virtualisierungsart liefern ↑Gerätetreiber, aber auch die ↑Ablaufplanung in dem Betriebssystem: muss die virtuelle Maschine ↑Echtzeit zusichern, ist dem Hypervisor nötiges Wissen zu übermitteln, damit dieser die virtuelle Maschine rechtzeitig wieder in Betrieb nehmen kann, so dass jeder auf dieser Maschine stattfindende ↑Prozess seine ↑Echtzeitbedingung einhalten kann. Auch wenn diese Form der Virtualisierung demnach intransparent ist für das Betriebssystem, so ist sie transparent für jedes ↑Maschinenprogramm.

partial interpretation (dt.) ↑partielle Interpretation.

partial virtualisation (dt.) ↑partielle Virtualisierung, ↑Teilvirtualisierung.

partielle Interpretation (en.) ↑*partial interpretation.* Ausführung der Anweisung einer (realen/virtuellen) Maschine durch verschiedene ↑Interpreter, deren Anordnung zueinander durch eine ↑funktionale Hierarchie definiert ist. In diesem arbeitsteiligen und strikt stapelorientierten Vorgang kooperieren Interpreter, die einerseits eine reale und andererseits wenigstens eine ↑virtuelle Maschine implementieren. Dabei definiert der Interpreter der realen Maschine die unterste Ebene in der Hierarchie, er startet den ↑Abruf- und Ausführungszyklus zur Interpretation. Eine ↑Ausnahmesituation, die zur ↑Unterbrechung der gegenwärtigen ↑Aktion auf tieferer Ebene führt, kann bewirken, dass ein Interpreter auf nächst höherer Ebene gestartet und somit eine virtuelle Maschine aktiviert wird. Dieser nachgeschaltete Interpreter läuft als Teil der ↑Ausnahmebehandlung, er sorgt letztlich dafür, dass die unterbrochene Aktion fortgesetzt wird. Entweder vollendet er diese Aktion, beendet damit auch die durch ihn bereitgestellte virtuelle Maschine, oder er leitet seinerseits eine Ausnahmebehandlung ein, aktiviert damit eine weitere virtuelle Maschine (Rekursion). Beendigung einer virtuellen Maschine bedeutet in solch einem Szenario, dass die (reale/virtuelle) Maschine, die zuvor

aktiv war, die Kontrolle über die weitere Programmausführung zurück erhält. Typische Beispiele für diese Art der Interpretation sind der ↑Systemaufruf und ein ↑Seitenfehler, mit dem ↑Betriebssystem als Interpreter. Ein anderes Beispiel liefert ein ↑sensitiver Befehl, der durch einen ↑Hypervisor interpretiert werden muss. In jedem dieser Fälle fährt zeitweilig eine virtuelle Maschine hoch, um unterbrochene Aktionen zu vollenden und sich dann alsbald wieder abzuschalten.

partielle Virtualisierung (en.) ↑*partial virtualisation.* Mehrfachnutzung einer Hardwareeinheit in einem ↑Rechensystem durch ein ↑Zeitteilverfahren. Ursprünglich nur bezogen auf die ↑CPU, um nämlich einem ↑Prozess die Illusion von einem eigenen Prozessor zu geben. Technische Grundlage dafür sind Systemfunktionen zur ↑Simultanverarbeitung. So können mehrere Prozesse zugleich stattfinden, indem jedem ein eigener virtueller Prozessor zur Verfügung steht. Das Konzept ist aber übertragbar auf andere Hardwareeinheiten, insbesondere dem ↑Hauptspeicher. Hier kann durch zeitweilige ↑Umlagerung ganzer Programme in den ↑Hintergrundspeicher derselbe Hauptspeicherbereich zur zeitweiligen Mehrfachnutzung durch mehrere Programme bereitgestellt werden.

Partition (en.) ↑*partition.* Teilbereich bestimmter Größe auf einem ↑Datenträger. Eine solche Aufteilung wird üblicherweise für die ↑Ablage genutzt und besteht dann aus einem ↑Dateisystem, sie kann aber auch exklusiv als ↑Umlagerungsbereich konfiguriert sein. Auf demselben Datenträger können mehrere solcher Teilbereiche (ggf. unterschiedlicher Größe) eingerichtet sein.

partition (dt.) Teilung, ↑Partition.

Pascal Programmiersprache: imperativ, prozedural (Wirth, 1971). Weiterentwicklung von ↑Algol 60 vor allem in Hinblick auf *starke Typisierung.*

passive waiting (dt.) ↑passives Warten.

passives Warten (en.) ↑*passive waiting.* Verfahren, bei dem ein ↑Prozess untätig ein bestimmtes ↑Ereignis erwartet: im Gegensatz zu ↑aktives Warten, gibt der Prozess seinen ↑Prozessor während der gesamten ↑Wartezeit ab und blockiert von sich aus (↑Prozesszustand). Wenn im Moment der Blockierung ein anderer Prozess bereit steht, löst der im ↑Betriebssystem mitlaufende ↑Planer einen ↑Prozesswechsel aus. Tritt das erwartete Ereignis ein, wird der betreffende Prozess dadurch (genauer: durch den das Ereignis herbeiführenden (externen) Prozess) deblockiert und von dem Planer bereit gestellt oder sofort (d.h., als laufend) eingesetzt, je nach dem Verfahren zur ↑Prozesseinplanung (d.h.,) und ↑Operationsprinzip (d.h., erlaubter ↑Verdrängungsgrad) des Betriebssystems.
Dieses ↑Warteverhalten trägt zur Maximierung der ↑Auslastung einerseits der ↑CPU und andererseits der ↑Peripherie bei. Im Falle der CPU wird ein anderer Prozess produktive Arbeit (↑Rechenstoß) vollbringen können, dabei gegebenenfalls auch einen weiteren ↑Ein-/Ausgabestoß auslösen und somit ein ↑Peripheriegerät beanspruchen. Im Falle der Peripherie werden von verschiedenen Prozessen abgesetzte Ein-/Ausgabestöße mehrere Peripheriegeräte zugleich beschäftigen können. Letzterer Aspekt bedeutet aber auch, dass der auf der CPU gegenwärtig stattfindende Prozess durch eine ↑Unterbrechungsanforderung verzögert werden dann, die ihre Ursache in der Beendigung des Ein-/Ausgabestoßes eines anderen (vorigen) Prozesses hat. Sogar mehrere solcher Stöße könnten nebenläufig zum gegenwärtigen Prozess, der womöglich keinen eigenen Ein-/Ausgabestoß auslöst, stattfinden und alle könnten am Ende diesen Prozess unterbrechen. Damit beeinflussen sich Prozesse bei diesem Warteverfahren indirekt, obwohl sie nicht direkt gekoppelt sind und gegebenenfalls auch nichts voneinander wissen: es besteht ein hohes Potential für ↑Interferenz kausal unabhängiger Prozesse. Zudem ist der im Vergleich zu aktivem Warten anfallende Mehraufwand (↑*overhead*) für die gesamte ↑Aktionsfolge bis einschließlich Prozesswechsel nicht unerheblich. Dieser erhöht grundsätzlich die ↑Latenzzeit für jeden Prozess, der warten muss und er geht erst bei genügend viel

produktive Arbeit, die während der Wartezeit vollbracht werden kann, im ↑Hintergrundrauschen unter. Latenzzeit wie auch Produktivitätsmaß sind noch bestimmbar, einerseits durch statische Analyse von dem zum Prozesswechsel führenden ↑Programm im Betriebssystem und andererseits durch dynamische Analyse der ↑Bereitliste, jedoch trifft dies nur bedingt auf die Wartezeit zu (vgl. ↑aktives Warten). Daher ist die Entscheidung, aktiv oder passiv zu warten auch nur bedingt eine effiziente Lösung. Gleiches gilt auch, wenn zunächst nur für eine kurze aktiv und dann, wenn das Ereignis immer noch nicht eingetreten ist, passiv gewartet wird.

path name (dt.) ↑Pfadname.

pausenloser Ablauf (en.) ↑*run to completion*. Merkmal einer ↑Ablaufsteuerung, die sicherstellt, dass ein ↑Prozess die mit ihm verbundene ↑Aufgabe zügig erfüllen kann, ohne durch ↑Verdrängung unterbrochen zu werden. Nach erfolgter ↑Einlastung schreitet der Prozess pausenlos voran und gibt den ↑Prozessor erst wieder ab, wenn er:

1. entweder durch ↑logische Synchronisation mit einem anderen (ggf. externen) Prozess das Ergebnis einer benötigten Zuarbeit abwarten und dazu ein diesbezügliches ↑konsumierbares Betriebsmittel erwerben (*acquire*) muss, oder
2. die Aufgabe komplett erfüllt hat.

Der erste Punkt impliziert eine ↑Ablaufplanung, die nur den zum Fortschritt des logisch synchronisierten (d.h., kausal abhängigen) Prozesses beitragenden anderen Prozess bevorzugt den/einen Prozessor zuteilt, nämlich wenn der ↑Prozesszustand dieses anderen Prozesses anzeigt, dass er bereit zur Freigabe (*release*) des zu konsumierenden Betriebsmittels ist. Dieser Aspekt wird jedoch nicht immer als zwingend erforderliches Merkmal angesehen, mit der Folge, dass der betreffende Prozess, trotz ↑kooperative Planung im Grundsatz, durch kausal nicht zusammenhängende Prozesse unerwünschte Verzögerung erfahren kann. Zur Minimierung der ↑Bearbeitungszeit des logisch synchronisierten Prozesses spielt jedoch der kausale Zusammenhang mit anderen Prozessen eine entscheidende Rolle. Dies bedeutet dann allerdings auch ↑Vorwissen zu den Daten- und gegebenenfalls auch Zeitabhängigkeiten zumindest jener Gruppe von Prozessen, für die diese Art von Ablaufsteuerung gelten soll. So ist für gewöhnlich nur der zweite Punkt wirklich ausschlaggebend.

PC Abkürzung für (en.) ↑*program counter* oder ↑*personal computer*.

PCB Abkürzung für (en.) ↑*process control block*.

PCLSRing Ausgesprochen (en.) *program counter lusering*. Der Mechanismus in ↑ITS, um einen ↑Systemaufruf jederzeit quasiatomar ablaufen lassen zu können. In ITS scheint die durch einen Systemaufruf aktivierte ↑Systemfunktion auch trotz einer möglichen ↑Unterbrechungsanforderung immer ununterbrechbar stattzufinden. Kommt es zur ↑Unterbrechung, findet eine von zwei Maßnahmen statt:

1. Der Systemaufruf wird zurückgezogen, indem der ↑Befehlszähler auf den Anfang der Aufrufausführung im ↑Betriebssystem zurückgesetzt wird. Der mittlerweile bereits geänderte Zustand wird gesichert, so dass bei Wiederaufnahme der Ausführung der Systemaufruf dort fortfährt, wo er zuvor unterbrochen worden ist.
2. Der Systemaufruf schließt ab, wenn dies mit nur noch wenigen Befehlen möglich ist.

Dadurch wird sichergestellt, dass kein ↑Prozess einen anderen Prozess (inkl. er selbst), der sich in einem ↑Systemaufruf befindet, observieren kann. In anderen Systemen (↑Mach) findet diese Technik Verwendung, um die ↑Unteilbarkeit einer ↑Aktion (↑Elementaroperation) die emuliert eine teilbare ↑Aktionsfolge bildet, logisch durchzusetzen. Ein ↑kritischer Abschnitt kann darüber ebenfalls abgesichert werden, insbesondere wenn in dem Abschnitt explizit ein ↑Prozesswechsel programmiert ist: nach Wiederaufnahme des weggeschalteten Prozesses, wird dieser in einen Wiederanlauf (*restart*) des kritischen Abschnitts gezwungen.

PCP Abkürzung für (en.) ↑*priority ceiling protocol*.

PDP 11 Kleinrechnerfamilie (1970–1990, DEC): 16-Bit, wortorientiert, mikroprogrammierbar; nahezu ↑orthogonaler Befehlssatz, ↑speicherabgebildete Ein-/Ausgabe; acht ↑Unterbrechungsprioritätsebenen, jeder ↑Unterbrechungsvektor bildet ein Tupel von ↑PC und ↑PSW, das bei einer ↑Unterbrechung in die entsprechenden ↑Prozessorregister geladen wird. Die ↑Rechner fanden insbesondere zum ↑Echtzeitbetrieb weitläufig Verwendung, hier insbesondere unter ↑RSX 11.

PDP 11/40 Kleinrechner (1970) aus der ↑PDP 11 Familie. Für das ↑Betriebssystem standen firmeneigene (insb. ↑RSX 11), aber insbesondere auch eine große Anzahl nicht-proprietärer Lösungen, vor allem ↑UNIX, zur Verfügung.

PDV Abkürzung für ↑Prozessdatenverarbeitung.

Pegelsteuerung (en.) ↑*level-triggered interrupt*. Auslöser für die ↑Unterbrechung ist eine Mindestzeit an Beständigkeit des Pegelstands auf der Signalleitung (↑*interrupt line*) zur ↑CPU. Für eine ↑Unterbrechungsanforderung erniedrigt (erhöht) die ↑Peripherie den Pegelstand und hält diesen Zustand, bis der ↑Unterbrechungshandhaber der anfordernden Peripherie die Unterbrechung bestätigt. Im Moment dieser Bestätigung, und zwar an dem entsprechenden ↑Peripheriegerät, wird der dort eingestellte Pegelstand zurückgenommen. Stellt dasselbe Gerät sofort nach der Bestätigung und noch während die ↑Unterbrechungsbehandlung läuft eine weitere Anforderung, kommt diese normalerweise implizit bei der Rückkehr aus der Unterbrechungsbehandlung zur Geltung — oder auch bereits vorher, wozu der Unterbrechungshandhaber jedoch die für die CPU in dem Moment gültige ↑Unterbrechungsprioritätsebene explizit herabsetzen muss (↑privilegierter Befehl). Damit werden auch wiederholte Anforderungen (*reassertion*) nicht verpasst, im Gegensatz zur ↑Flankensteuerung. Jedoch darf der Unterbrechungshandhaber die Bestätigung nicht auslassen, da er sonst bei Rückkehr aus der Unterbrechungsbehandlung sofort wieder aufgerufen wird, damit seinen erneuten Aufruf nicht etwa mit einer weiteren Unterbrechung in Verbindung bringen kann und, was viel gravierender ist, der unterbrochene ↑Prozess womöglich niemals fortgesetzt wird. Auslassen der Bestätigung einer pegelgesteuerten Unterbrechung kann ↑Panik auslösen.

performance (dt.) Leistung, Leistungsfähigkeit; ↑Performanz.

Performanz (en.) ↑*performance*. Leistungsfähigkeit eines einzelnen Systembestandteils, eines Subsystems oder eines ganzen Systems, Hardware oder Software.

Periode (en.) ↑*cycle period*. Zeitraum, der zwischen zwei gleichen ↑Aufgaben eines sich wiederholenden (internen) ↑Prozesses liegt. In diesem Zeitraum ist die Bearbeitung einer oder mehrerer verschiedener Aufgaben vorgesehen, jede davon auch als ↑periodische Aufgabe bezeichnet. Die Länge des Zeitraums ist für gewöhnlich fest und bestimmt durch den in ↑Echtzeit zu steuernden (externen) Prozess.

periodic server (dt.) ↑periodischer Zusteller.

periodic task (dt.) ↑periodische Aufgabe.

periodische Aufgabe (en.) ↑*periodic task*. Eine wiederkehrende ↑Aufgabe, deren Bearbeitung an einer ↑Periode gebunden ist. Gegenteil von ↑nichtperiodische Aufgabe.

periodischer Zusteller (en.) ↑*periodic server*. Bezeichnung für einen speziellen ↑Zusteller, der ↑nichtperiodische Aufgaben in einen periodischen Ablauf einordnet. Grundlage bildet ein ↑Echtzeitbetrieb, der vordergründig ↑periodische Aufgaben bewältigen muss, aber gleichfalls auch ↑aperiodische Aufgaben oder ↑sporadische Aufgaben zu bearbeiten hat, ohne dabei den periodischen Ablauf zu stören.
Dem Zusteller wird ein bestimmtes Aufnahmevermögen (*capacity*) an nichtperiodischen Aufgaben zugesprochen, dessen Umfang so bemessen ist, dass die rechtzeitige Bearbeitung der

strikt periodischen Aufgaben trotz Zustelleraktivität weiterhin stets gewährleistet ist. Dieses Aufnahmevermögen entspricht einem *Zeitbudget*, das die maximale ↑Ausführungszeit (↑WCET) des Zustellers in seinem jeweiligen Durchlauf (↑Periode) festlegt. Dieses Budget nimmt mit voranschreitender Aufgabenbearbeitung ab, und zwar um die bei der Bearbeitung einer Aufgabe verstreichenden (realen) Zeit. Ist das Budget vollständig aufgebraucht, wird die in dem Moment stattfindende Aufgabenbearbeitung unterbrochen, zurückgestellt und erst im nächsten Durchlauf des Zustellers wieder aufgenommen.
Es gibt verschiedene Arten solcher Zusteller. Sie unterscheiden sich zum einen in der Art und Weise, wie in ihrem jeweiligen Durchlauf mit dem verfügbaren Zeitbudget im Falle einer leeren oder abgearbeiteten Aufgabenliste umgegangen wird und zum anderen in dem Prinzip, wie die Wiederauffüllung (*replenishment*) ihres Zeitbudgets geschieht:

abfragender Zusteller (*polling server*). Die ↑Bereitzeit des Zustellers entspricht die einer periodischen Aufgabe und ist definiert durch den Beginn des ihr zugeordneten Durchlaufs, sie ist unabhängig vom Zustand der Aufgabenliste des Zustellers. Zur ↑Startzeit in seinem Durchlauf fragt der Zusteller den auf seiner Aufgabenliste verzeichneten Aufgabenbestand ab. Ist eine zu bearbeitende Aufgabe auf der Liste verzeichnet, beginnt der Zusteller mit der Abarbeitung der Aufgabenliste für die Dauer des ihm zur Verfügung stehenden Zeitbudgets. Ist die Liste (anfangs oder mittlerweile) leer, beendet der Zusteller von selbst seine Tätigkeit in dem Durchlauf. Ein in dem Durchlauf (a) dem Zusteller noch zustehendes restliches Zeitbudget verfällt und (b) danach noch eintreffende nichtperiodische Aufgaben bleiben unberücksichtigt bis zum nächsten Zustellerdurchlauf. Im ungünstigen Fall verlängert sich somit die ↑Antwortzeit für die Bearbeitung nichtperiodischer Aufgaben. Für den nächsten Zustellerdurchlauf wird das Zeitbudget des betreffenden Zustellers immer komplett aufgefüllt.

aufschiebbarer Zusteller (*deferrable server*). Die Bereitzeit des Zustellers entspricht die einer periodischen Aufgabe und ist definiert durch den Beginn des ihr zugeordneten Durchlaufs, sie ist abhängig vom Zustand der Aufgabenliste des Zustellers. Der Zusteller startet seine Tätigkeit in dem ihm zugeordneten Durchlauf überhaupt erst, wenn seine Aufgabenliste einen Aufgabenbestand aufweist. Konnte die Liste in dem Durchlauf vollständig abgearbeitet werden, beendet der Zusteller von selbst seine Tätigkeit. Dabei verfällt ein ihm gegebenenfalls noch zustehendes restliches Zeitbudget nicht. Sobald in dem Durchlauf eine weitere Aufgabe eintrifft und noch Zeitbudget vorhanden ist, nimmt der Zusteller ihre Bearbeitung unverzüglich auf. Damit verkürzt sich die Antwortzeit für die Bearbeitung nichtperiodischer Aufgaben. Ein nicht aufgebrauchtes Zeitbudget des Zustellers verfällt erst am Ende des jeweiligen Durchlaufs. Für den nächsten Zustellerdurchlauf wird das Zeitbudget des betreffenden Zustellers immer komplett aufgefüllt. Ergab sich eine Aufschiebung des Zeitbudget des Zustellers ans Ende seines aktuellen Durchlaufs und ist der Zusteller im direkt nachfolgenden Durchlauf abermals eingeplant, kommt es zum sogenannten Doppelschlag (*double hit*), wenn bei Duchlaufwechsel die Aufgabenliste noch gefüllt ist. Eine solche mögliche Anhäufung von Zeitbudget beeinträchtigt die Bearbeitung periodischer Aufgaben.

sporadischer Zusteller (*sporadic server*). Verfeinerung des aufschiebbaren Zustellers, die der Anhäufung von Zeitbudget durch feste Auffüllzeitpunkte vorbeugt und zugleich eine Verbesserung der mittleren Antwortzeit für die Bearbeitung nichtperiodischer Aufgaben ermöglicht, ohne dabei die Kapazitätsgrenze für die periodischen Aufgaben herabzusetzen. Dazu wird das Zeitbudget im „korrekten“ Zeitabstand vergütet und somit in Folge nie mehr Aufnahmevermögen als vorgesehen eingeräumt. Dies wird dadurch erreicht, indem die Wiederauffüllung des Zeitbudgets nicht immer zu Beginn des jeweiligen Durchlaufs des Zustellers geschieht, sondern unregelmäßig (*sporadic*) während seines Durchlaufs stattfindet. Dabei wird allerdings nur ein bestimmtes Stück (*chunk*) seines totalen Zeitbudgets aufgefüllt. Jede übrige Zeitspanne, in der keine Bearbeitung periodischer Aufgaben geschieht, ist für die Bearbeitung nichtperiodischer Aufgaben im Moment ihrer ↑Ankunftszeit verfügbar.

Anders als seine Bezeichnung vermuten lässt, bearbeitet der sporadische Zusteller keinesfalls nur sporadische Aufgaben. Sehr wohl können auch aperiodische Aufgaben auf seiner Aufgabenliste verzeichnet sein. Dies trifft ebenso zu auf die anderen beiden Zustellerarten. Stattdessen bezieht sich dieses Adjektiv auf die im Vergleich zum abfragenden oder aufschiebbaren Zusteller unregelmäßige Wiederauffüllung des Zeitbudgets.
Allen drei Zustellern gemeinsam ist, dass sporadische Aufgaben vor ihrer Bereitstellung (d.h., Aufnahme auf die Aufgabenliste) einer ↑Übernahmeprüfung zu unterziehen sind. Erst wenn diese Prüfung positiv verlaufen ist, wird die betreffende sporadische Aufgabe als nichtperiodische Aufgabe gemäß eines dieser Zustellerkonzepte behandelt.
Die Aufgabenliste ist für gewöhnlich als ↑Vorrangwarteschlange ausgeprägt, um insbesondere bei einem Mix von aperiodischen und sporadischen Aufgaben letzteren die durch die Übernahmeprüfung bestätigte freie Rechenkapazität schließlich auch zusichern zu können. So werden sporadische Aufgaben immer vor den aperiodischen Aufgaben bearbeitet. Als Folge daraus kann ↑Termineinhaltung bei der Bearbeitung aperiodischer Aufgaben nicht garantiert werden: die Bearbeitung aperiodischer Aufgaben mit festen Terminen wird dann verworfen, falls sie nicht mehr rechtzeitig zum Abschluss kommen kann, oder abgebrochen.

peripheral (dt.) ↑Peripheriegerät.

peripheral equipment (dt.) ↑Peripherie.

Peripherie (en.) ↑*peripheral equipment*. Gesamtheit der an einer ↑Zentraleinheit angeschlossenen Geräte, jedes davon auch als ↑Peripheriegerät bezeichnet.

Peripheriegerät (en.) ↑*peripheral*. Bezeichnung für ein Gerät, das die ↑Steuerung durch einen ↑Prozessor benötigt. Jede Form von Ein-/Ausgabegerät (z.B. Tastatur, Bildschirm, Maus, Platte; Sensoren, Aktoren) in einem ↑Rechensystem, aber auch Erweiterungskarten beziehungsweise Anschlüsse zur seriellen/parallelen ↑Ein-/Ausgabe von ↑Daten.

permanent memory (dt.) ↑Festspeicher.

persistentes Betriebsmittel siehe ↑dauerhaftes Betriebsmittel.

Persistenz Speicherbarkeit einer ↑Entität; die Fähigkeit von einem System, seine innere Struktur sowie die Zusammengehörigkeit und gegliederte Zusammenstellung seiner Einzelbestandteile dauerhaft speichern zu können. Hierzu kommt die ↑Ablage zum Einsatz.

personal computer (dt.) Arbeitsplatzrechner; Abkürzung ↑PC.

pessimistische Nebenläufigkeitssteuerung (en.) *pessimistic* ↑*concurrency control*. Bezeichnung für eine ↑Nebenläufigkeitssteuerung, die die Annahme trifft, dass ein ↑Prozess mit wenigstens einem anderen Prozess wahrscheinlich in Konflikt gerät, wenn eine gemeinsame ↑Aktion oder ↑Aktionsfolge gleichzeitig stattfindet. Dem möglichen Konflikt wird vorgebeugt, indem die Prozesse die betreffende Aktion/Aktionsfolge nur als Ganzes nacheinander durchführen können. Dies lässt sich durch ↑blockierende Synchronisation erreichen oder auf Basis einer ↑Ablaufplanung, die die Prozesse (in der kritischen Phase) strikt nacheinander stattfinden lässt. Gegenteil von ↑optimistische Nebenläufigkeitssteuerung.

Pfadname (en.) ↑*path name*. Benennung einer ↑Entität durch Angabe des Pfads zu ihr im ↑Namensraum. Der Pfad bildet eine Zeichenfolge, in der ↑Name und Trenntext (↑Separator) abwechselnd auftreten, beispielsweise:

- `Multics>ist>aktueller>denn>je` oder
- `UNIX/ist/ohne/Multics/undenkbar` und `lebt/weiter/in/Linux` oder
- `Windows\hat\viel\mehr\von\VMS\als\allgemein\bekannt\ist`

Der abschließende Name, nämlich das Blatt eines baumartig aufgebauten Namensraums, bezeichnet die Entität (im Beispiel: `je`, `undenkbar`, `Linux`, `ist`). Die Namen davor, durch den Trenntext jeweils separiert, bezeichnen einen ↑Namenskontext, der für gewöhnlich — insbesondere in einem ↑Dateisystem — als ↑Verzeichnis implementiert ist.

PGAS Abkürzung für (en.) *partitioned global address space*. Ein ↑virtueller Adressraum, der auseinanderliegende ↑Hauptspeicher verschiedener ↑Rechner zusammenschließt: für gewöhnlich liegen die Hauptspeicher über ein ↑Rechnernetz verteilt vor (↑DSM). In diesem Adressraum ist jeder einzelne Hauptspeicher einem bestimmten ↑Adressbereich eindeutig zugeordnet.

physical address (dt.) ↑reale Adresse.

physical address space (dt.) ↑realer Adressraum.

PIC Abkürzung für (en.) ↑*position independent code* und (en.) *programmable* ↑*interrupt controller*.

PID Abkürzung für (en.) ↑*process identifaction*, *process identifier*.

Pilot Einplatz-/Einbenutzerbetriebssystem, 1981, für vernetzte Arbeitsplatzrechner (↑Xerox Alto). Geschrieben in ↑Mesa, wobei dem gesamten ↑Rechensystem überhaupt ein einsprachiger Ansatz (*single-language approach*) zugrunde liegt: Das ↑Betriebssystem kann als sehr leistungsfähiges ↑Laufzeitsystem für Mesa betrachtet werden.

PIO Abkürzung für (en.) ↑*programmed input/output*.

PIP Abkürzung für (en.) ↑*priority inheritance protocol*.

pipe (dt.) Kanal, Leitung, Rohr. Einrichtung eines ↑Betriebssystems, die vor allem dazu dient, ↑Daten weiterzuleiten (in Anlehnung an den Duden). Technische Maßnahme zum Aufbau einer ↑Verarbeitungskette, deren Funktionseinheiten als ↑Prozessinkarnationen ausgelegt sind. Weit verbreitet als ↑UNIX `pipe(2)`.

pipeline (dt.) ↑Verarbeitungskette.

PIT Abkürzung für (en.) ↑*programmable interval timer*.

PL/1 Abkürzung für (en.) *programming language # 1*, prozedurale, imperative Programmiersprache (1964).

placeholder (dt.) ↑Platzhalter.

placement algorithm (dt.) ↑Platzierungsalgorithmus.

placement policy (dt.) ↑Platzierungsstrategie.

Planer (en.) ↑*scheduler*. Bezeichnung für ein ↑Programm zur Erstellung und Verwaltung von einem ↑Ablaufplan. Dieses Programm kommt nach zwei verschiedenen Modellen zum Einsatz: vor (*off-line*, statisch) oder zur (*on-line*, dynamisch) ↑Laufzeit der Abarbeitung des Ablaufplans. Auch eine Kombination beider Varianten ist möglich. In dem Fall reflektiert der statische Ablaufplan für jeden einzelnen darin aufgeführten ↑Prozess Vorwissen zu Daten- und Kontrollflussabhängigkeiten, das dem ↑Betriebssystem initial für eine Gruppe von Prozessen übergeben wird. Zur Laufzeit wird dieser Plan dann entsprechend der dynamischen Vorgänge im ↑Rechensystem fortgeschrieben.
Der statische Ansatz findet bevorzugt bei Spezialsystemen Verwendung, sofern das benötigte Vorwissen vorhanden ist und die Aufstellung des Plans in angemessener Zeit möglich ist: die Erstellung eines solchen Plans ist typischerweise ein NP-schweres Entscheidungsproblem, das in Abhängigkeit von der Prozessanzahl wie auch dem Grad der Prozessabhängigkeiten gegebenenfalls eine unvertretbar hohe Berechnungszeit bedingt.
Demgegenüber ist der dynamische Ansatz in Spezial- und Universalsystemen verbreitet, in

letzteren wegen dem dort oft nicht vorhandenem Vorwissen eben auch unabdinglich. In diesem Fall der (mit den zu planenden Prozessen) im Betriebssystem mitlaufenden Ablaufplanung wird letztere als ↑Unterprogramm aufgerufen, wann immer Gründe für eine Aktualisierung des Plans vorliegen. Diese Gründe sind die Zulassung, Bereitstellung, Blockierung, Unterbrechung, Verdrängung, Suspendierung oder Beendigung eines Prozesses.

Planung (en.) ↑*scheduling*. Ausarbeitung eines Plans, der festlegt, wann ein ↑Auftrag einem ↑Prozessor zur Verarbeitung übergeben werden soll (↑*dispatching*). Je nach Art des Prozessors unterscheiden sich für gewöhnlich die Planungsverfahren, indem ihre Methoden auch technische Merkmale der jeweiligen ↑Bedieneinheit berücksichtigen. Darüber hinaus ist es nicht ungewöhnlich, für dieselbe Prozessorklasse oder dasselbe Prozessorexemplar mehrere solcher Verfahren zu haben.
Ist der Auftrag ein ↑Prozess und der Prozessor eine ↑CPU oder ein ↑Rechenkern, wird auch von ↑Ablaufplanung gesprochen. Für gewöhnlich werden hier drei verschiedene Ebenen unterschieden (von oben nach unten): ↑langfristige Planung, ↑mittelfristige Planung und ↑kurzfristige Planung. Aufträge zur Ein- oder Ausgabe von ↑Daten unterliegen ebenfalls einer Planung, die zudem sehr stark durch die technischen Merkmale von dem jeweiligen ↑Peripheriegerät beeinflusst ist. Grundsätzlich ist derartige Planung der Auftragsverarbeitung notwendig, wenn zu einem Zeitpunkt mehr Aufträge anstehen können als Prozessoren für deren Verarbeitung verfügbar sind. Eine zentrale Bedeutung kommt dabei dem ↑Einplanungsalgorithmus zu, der über die Reihenfolge und Häufigkeit der ↑Einlastung eines Prozessors entscheidet.

Planungswarteschlange (en.) ↑*scheduling queue*. Eine ↑Warteliste von ↑Prozessen, die zum sofortigen, mit den strategischen Entscheidungen der ↑Einplanung konform gehenden ↑Kontextwechsel führen kann, sobald das ↑Ereignis gemeldet wird, dessen Eintritt wenigstens einer der gelisteten Prozesse erwartet. Die originäre Idee (Hansen, 1972) sah für jede ↑gemeinsame Variable, über die die Interaktion ↑gekoppelter Prozesse laufen soll, solch eine Warteliste vor. Wertzuweisungen an eine derart „funktional erweiterte“ Variable konnten sodann gezielt die Fortsetzung eines oder mehrerer wartender Prozesse bewirken (*context switching queue*). Die Prozesse waren damit bis zu einem beliebigen Grad durch ein ↑Anwendungsprogramm steuerbar, indem die Zuordnung einer jeden Warteliste entweder zu einer Gruppe von Prozessen oder einem einzelnen Prozess geschah.
Benötigen die Prozesse nur noch den ↑Prozessor, um fortfahren zu können, dann erwarten sie als Ereignis die Prozessorzuteilung (↑*dispatching*): sie befinden sich im ↑Prozesszustand <u>bereit</u>, stehen auf der ↑Bereitliste. Die Listenposition eines Prozesses ist dann durch das bei der Einplanung jeweils zu berücksichtigende ↑Einplanungskriterium bestimmt.
Warten die Prozesse nicht nur auf den Prozessor, sondern darüber hinaus auch auf die Verfügbarkeit weiterer ↑Ressourcen, dann befinden sie sich in einer ↑Prozessblockade, ihr Prozesszustand weist sie als <u>blockiert</u> aus. In dem Fall stehen die Prozesse für gewöhnlich auf einer Warteliste für das Ereignis, das die Verfügbarkeit der betreffenden Ressource verzeichnet (↑*event waitlist*). Um ↑Interferenzen vorzubeugen oder zumindest gering zu halten, sollte die Prozessposition auf dieser Warteliste konform gehen mit dem Verfahren zum Führen oder Weiterführen des Prozessplans (↑*scheduling*).
Trifft das Ereignis ein, auf das mehr als ein Prozess wartet, sollte zuerst der Prozess von der ↑Ereigniswarteliste gestrichen werden, der im Falle der Prozessorzuteilung von der Bereitliste gestrichen werden würde. Denn indem das von mehreren Prozessen erwartete Ereignis eintritt, ist immer eine Auswahlentscheidung zu treffen, nämlich welcher von den wartenden Prozesse zuerst die Prozessorzuteilung erfahren wird. Auch wenn dieser Prozess zunächst auf die Bereitliste kommt und immer noch mit anderen dort bereits verzeichneten Prozessen um den Prozessor konkurriert, so wurde er aufgrund bestimmter Kriterien aus der Menge der auf der Ereigniswarteliste stehenden Prozessen bevorzugt.
Das bedeutet, in der konkreten technischen Umsetzung der Ereigniswarteliste sollte das Kriterium für die Listenposition eines darauf verzeichneten Prozesses in Relation zu seinen Konkurrenten um das betreffende Ereignis stimmig sein zu dem Kriterium für seine

relative Position auf der Bereitliste in Bezug auf seine Konkurrenten um den Prozessor. Angenommen die Liste soll eine ↑Ereigniswarteschlange bilden, das heißt, als *dynamische Datenstruktur* eine herkömmliche ↑Schlange darstellen, dann ist in Bezug auf die jeweilige Einplanungsentscheidung folgendes zu beachten:

↑FCFS, ↑RR Die mit einer Schlange gemeinhin verknüpfte Bedienungsstrategie (↑FIFO) ist stimmig. Es sind keine zusätzlichen Maßnahmen erforderlich, um die Prozesse in die korrekte Reihenfolge zu bringen.

↑MLQ, ↑MLFQ Die relative Listenposition eines wartenden Prozesses ist bestimmt durch seine ↑Priorität, die der Typ seines ↑Programms beziehungsweise der betreffende Index in einem ↑Feld von Bereitlisten festlegt. Der Prozess ist in aufsteigender Reihenfolge einzusortieren, der Sortierschlüssel ist der für ihn zutreffende Indexwert (`level`, S. 69).

↑VRR Die relative Listenposition eines wartenden Prozesses ist bestimmt durch einen Wert für eine Zeitdauer, und zwar einen Anteil seiner ↑Zeitscheibe. Der Prozess ist in aufsteigender Reihenfolge einzusortieren, der Sortierschlüssel ist die Größe des ihm in der Zeitscheibe noch zur Verfügung stehenden Schlupfes (`slack`, S. 386) oder die Länge des von ihm bereits getätigten ↑Rechenstoßes (`usage`, S. 387). Ein größerer Schlupf beziehungsweise kürzerer Rechenstoß könnte ein Prozess mit höherer Interaktionsrate sein.

↑SPN, ↑SRTF, ↑HRRN Die relative Listenposition eines wartenden Prozesses bestimmt sich durch einen Wert für eine Zeitdauer, und zwar einen Schätzwert für die Länge seines nächsten Rechenstoßes. Der Prozess ist in aufsteigender Reihenfolge einzusortieren, der Sortierschlüssel ist dieser Schätzwert (`guess`, S. 316).

Der am Kopf einer solchen ↑Warteschlange stehende Prozess ist dann derjenige, der in gleicher Konstellation auch im Falle einer Bereitliste vorne stehen würde und als nächster den Prozessor zugeteilt bekäme. Allerdings ist damit erst ein erster Schritt getan. Um einen Kontextwechsel auszulösen, muss dieser Prozess vor allem noch dem gegenwärtig stattfindenden Prozess gegenüber bevorrechtigt (↑*preemptive*) sein. Trifft dies nicht zu, wird der aus der Ereigniswarteschlange genommene Prozess auf die Bereitliste gesetzt.
Die beschriebene Vorgehensweise berücksichtigt bestimmte Mechanismen für die Umsetzung der Kriterien zur Prozesseinplanung in der Verwaltung einer gewöhnlichen Schlange (FIFO), um sicherzustellen, dass die wartenden Prozesse übereinstimmend mit der Reihungsdisziplin der Bereitliste angeordnet sind. Bis auf die FIFO-ähnlichen Verfahren (FCFS, RR) fallen dann allerdings vergleichsweise kostspielige ↑Aktionen an, wenn ein wartender Prozess einzuschlängeln (*enqueue*) ist.
Alternativ bietet sich auch als mögliche Lösung an, die Prozesse gemäß FIFO zu reihen, damit einen wartenden Prozess sehr effizient (sogar mit konstantem Aufwand, vgl. S. 277) in die Schlange zu stellen, und dann, sobald das Ereignis eingetreten ist, alle darauf wartenden Prozesse in die Bereitliste zu übernehmen, in die sie dann *en bloc* entsprechend der jeweiligen Strategie zur Prozesseinplanung behandelt werden (Hansen, 1972). Konsequenz daraus ist jedoch die Fortsetzung gegebenenfalls mehrerer Prozesse, obwohl nur ein einzelnes Fortsetzungsereignis eintrat. Wenn dieses Verhalten für die gegebene ↑Signalisierungsdisziplin unerwünscht ist oder zu Fehlverhalten führen kann (↑*signal and urgent wait*), dann müssen diese Prozesse, sobald sie aus der Blockierungsoperation zurückkehren, ihre Wartebedingung erneut überprüfen (bspw. S. 300).
Ein ganz anderer Ansatz besteht darin, die Warteliste tabellenorientiert auszulegen, das heißt, eine *statische Datenstruktur* (↑*array*) zur Grundlage zu nehmen. Angenommen jeder der Prozesse besitzt eine *eindeutige statische Priorität* und wird mit einem dazu passenden Verfahren (↑*fixed-priority scheduling*) behandelt. Damit ergibt sich eine zu MLQ sehr ähnliche Organisation wartender Prozesse. Allerdings sorgt die Randbedingung „eindeutige Priorität“ dafür, dass pro Ebene nur maximal ein Prozess in der Warteschlange für ein Ereignis stehen kann (*single-process queue*). Dies wiederum lässt sich ausnutzen für eine äußerst raumsparende Art und Weise der Repräsentation solcher Warteschlangen: ein ↑Markierungsbit pro

Ereignis, mehrere davon zusammengefasst in einer ↑Bitleiste pro ↑Prozesskontrollblock genügt, um festzuhalten, ob und gegebenenfalls worauf ein Prozess wartet. Damit ist sehr leicht eine Signalisierungsdisziplin umsetzbar, bei der immer nur höchstens ein Prozess aus seiner Blockierungsoperation zurückkehren kann — woraus aber nicht geschlossen werden darf, dass mit dieser klassischen Lösung (Hansen, 74) auf die erneute Überprüfung der Wartebedingung durch den deblockierten Prozess verzichtet werden kann (↑*signal and return*).

Plattenspeicher (en.) ↑*disc memory*. ↑Datenträger, für gewöhnlich in Form einer Scheibe mit einer magnetischen Oberfläche (*platter*). Im Betrieb rotiert die Scheibe mit hoher Geschwindigkeit. Die ↑Daten werden blockweise durch Lese-/Schreibköpfe von der Plattenoberfläche abgerufen beziehungsweise auf die Plattenoberfläche aufgebracht.

Plattensteuerung (en.) ↑*disc control*, ↑*disc controller*. Funktion zur ↑Steuerung beziehungsweise Steuereinheit (↑*controller*) von einem ↑Plattenspeicher.

Platzhalter (en.) ↑*placeholder*. Bezeichnung für eine ↑Speicherstelle, die für eine ↑Variable eines ↑Programms reserviert ist. An dieser Stelle hat genau ein ↑Exemplar des ↑Datentyps dieser Variable seinen Platz. Für dieselbe Variable können zu einem Zeitpunkt mehrere solcher Stellen auf verschiedenen Stufen der ↑Speicherhierarchie vorgemerkt sein.

Platzierungsalgorithmus (en.) ↑*placement algorithm*. Lösungs- und Bearbeitungsschema, Handlungsvorschrift zur ↑Speicherzuteilung, zentraler Rechenvorgang einer ↑Platzierungsstrategie. Ist charakterisiert durch die Reihenfolge von Einträgen auf einer Liste, die freie Abschnitte beliebiger Länge im ↑Hauptspeicher erfasst (↑*hole list*), und einem oder mehrerer Kriterien, nach denen genau einer dieser Einträge für die Speicherzuteilung auf Anforderung durch einen ↑Prozess ausgewählt wird. Die Einträge können einerseits der Größe nach auf- oder absteigend oder anderseits der ↑Adresse nach und dann in aller Regel nur aufsteigend sortiert sein. ↑Bezugspunkt dabei ist jeweils die Größe beziehungsweise Adresse des durch einen Eintrag repräsentierten freien Hauptspeicherabschnitts.
Bei Sortierung nach aufsteigender Größe (↑*best fit*) ist das Ziel, den ↑Verschnitt möglichst klein zu halten: hier wird der Eintrag gewählt, der die beste Passung, also den kleinsten Rest, bringt. Bleibt ein Rest übrig, muss dieser mit einem zweiten Suchlauf durch die Liste neu einsortiert werden. Da jeder übrig bleibende Rest darüberhinaus einen Eintrag mit noch kleinerer Größe erzeugt, wird ↑externe Fragmentierung immer mehr wahrscheinlich. Dieses Problem wird abgeschwächt mit einer Sortierung der Listeneinträge nach absteigender Größe (↑*worst fit*). Hier wird der Eintrag gewählt, der die schlechteste Passung hat, also den größten Rest liefert. Die Annahme dabei ist, dass ein gegebenenfalls übrig bleibender Rest wahrscheinlich immer noch groß genug für eine nachfolgende Speicherzuteilung und damit also brauchbar ist. Bleibt ein Rest übrig, muss auch dieser mit einem zweiten Suchlauf durch die Liste neu einsortiert werden.
Wiederholte Suchläufe werden vermieden, wenn die Einträge der Adresse nach sortiert sind und die Speicherzuteilung immer den hinteren Teil eines genügend großen Abschnitts zurück liefert. Ein sehr einfaches Verfahren ist es dann, die Liste immer vom Anfang her nach dem erstbesten Abschnitt (↑*first fit*) zu durchsuchen. Dieses Verfahren hinterlässt vorne eher kleinere Einträge und führt somit dazu, dass nachfolgende Suchläufe wahrscheinlich länger benötigen, um überhaupt in die Nähe brauchbar großer Abschnitte zu kommen. Das lässt sich vermeiden, indem ein Suchlauf immer nach dem Eintrag startet, der im vorangegangenen Suchlauf für die Speicherzuteilung verwendet wurde: ab dieser Stelle wird der nächstbeste Abschnitt (↑*next fit*) auf der Liste gesucht. Am Ende der Liste angekommen wird ab Listenanfang bis zum Startpunkt weitergesucht (↑*circular first fit*).
Wurde die Liste komplett abgesucht, ohne einen passenden freien Abschnitt zu finden, scheitert bei allen Verfahren die Speicherzuteilung. Solche Fehlläufe werden noch durch das Phänomen der ↑Fragmentierung begünstigt, mit dem alle Verfahren konfrontiert sind, die mit freien Speicherabschnitten beliebiger und unterschiedlicher Länge umgehen müssen. Fragmentierung kann jedoch durch ↑Verschmelzung eines frei werdenden Abschnitts mit einem

oder zwei auf der Liste bereits verzeichneten freien Abschnitten vermindert werden. Dadurch werden einerseits größere und andererseits weniger freie Abschnitte generiert, was die Wahrscheinlichkeit von Fehlläufen senkt (größere freie Abschnitte) und den Aufwand für Suchläufe reduziert (weniger Listeneinträge).

Platzierungsstrategie (en.) ↑*placement policy*. Verfahrensweise nach der die ↑Speicherzuteilung geschieht. Sie bestimmt die ↑Adresse von einem freien Bereich (↑*hole*) im ↑Arbeitsspeicher, wo nach erfolgter Speicherzuteilung an einen ↑Prozess Bestände von ↑Text oder ↑Daten entsprechend der Bereichsgröße von dem Prozess selbst abgelegt werden können. Mit Bereichsadresse und -größe hat die Strategie einen Platz von möglicherweise vielen anderen Plätzen ausgewählt und dem Prozess zur freien Verfügung zugeteilt.
Die freien Bereiche stehen auf einer Liste (↑*hole list*), deren Organisation und Repräsentation einerseits vom ↑Platzierungsalgorithmus und andererseits vom Geltungsbereich der Speicherzuteilung abhängig ist, das heißt, ob diese auf der ↑Maschinenprogrammebene durch ein ↑Laufzeitsystem oder auf der ↑Befehlssatzebene durch das ↑Betriebssystem geschieht. Im Falle des Betriebssystems bestimmt die Art des Strukturelements (↑Seite, ↑Segment) von einem ↑Prozessadressraum maßgeblich die technische Auslegung dieser Liste.
Liegt dem ↑Prozess ein ↑seitennummerierter Adressraum zugrunde, erfolgt die Speicherzuteilung seitenweise und damit in einer Größe, die immer Vielfaches der Größe einer ↑Seite ist. Da jede Seite immer nur in exakt einen ↑Seitenrahmen „eingespannt“ (platziert) wird und alle Seitenrahmen gleich groß (wie auch die Seiten) sind, ist jeder freie Seitenrahmen gut zur Platzierung einer Seite im ↑Hauptspeicher. Um zu verbuchen, ob ein Seitenrahmen frei oder belegt ist, reicht ein Bit: 1 = `true` = frei (Loch), 0 = `false` = belegt. Alle Seitenrahmen des Hauptspeichers werden dann über eine ↑Bitleiste erfasst, die als Tabelle (*bit map*) gespeichert ist. Für eine Speicherzuteilung von n_b ↑Byte sind dann $n_p = (n_b + sizeof(page) - 1)/sizeof(page)$ beliebige Bits in dieser Bitleiste zu finden, die jeweils einen freien Seitenrahmen markieren (d.h., auf 1 gesetzt sind).
Bestimmt dagegen ein ↑segmentierter Adressraum das Herrschaftsgebiet von einem Prozess, erfolgt die Speicherzuteilung segmentorientiert und damit in einer Größe, die immer Vielfaches der Größe des Strukturelements eines Segments ist. Handelt es sich bei diesem Element um eine Seite (↑*segmented paging*), ist der Fall allerdings einfach und die Zuteilung kann auf Basis einer Bitleiste wie eben zuvor beschrieben geschehen. Bei „reiner“ ↑Segmentierung jedoch ist das Byte das Strukturelement eines Segments. In dem Fall kann die Größe jedes Segments einer beliebigen Anzahl von Byte im Bereich $[0, 2^n - 1]$ entsprechen.
Die Segmentgröße ist einerseits durch das ↑Programm und andererseits durch einen Prozess in dem Programm bestimmt. Verschiedene Prozesse (verschiedener Programme) beanspruchen damit verschieden große Segmente im Hauptspeicher, um stattfinden zu können. Enden Prozesse und werden daraufhin die durch sie beanspruchten Segmente im Hauptspeicher freigegeben, entstehen freie Bereiche (Löcher) unterschiedlicher Größe.
Eine Bitleiste zur Verwaltung dieser freien Bereiche scheidet wegen dem sehr ungünstigen Kosten-Nutzen-Verhältnis aus, da pro Byte ein Bit notwendig wäre: bei einem 4 GiB großen Bereich im Hauptspeicher könnte (bei erlaubter kleinster Segmentgröße von einem Byte) jedes zweite Byte belegt/frei sein, was eine Bitleiste von 2^{31} Bits Länge beziehungsweise Bittabelle von 256 MiB Größe beanspruchen würde. Um die Bitleiste auch hier effizient nutzen zu können, müsste einem Segment ein viel größeres Strukturelement (z.B. eine Seite) zugrunde gelegt werden. Bleibt es jedoch bei dem Byte, muss ein Eintrag auf der Liste freier Bereiche im Hauptspeicher die ↑Adresse und die Länge (in Byte) eines solchen Bereichs verzeichnen. Da die Anzahl dieser Einträge variiert, ist diese ↑Freispeicherliste gemeinhin als *dynamische Datenstruktur* ausgeprägt. Dabei sind die Einträge in dieser Datenstruktur nach Kriterien sortiert, die der Platzierungsalgorithmus jeweils vorgibt.
Für gewöhnlich wird der Ansatz als dynamische Datenstruktur nicht nur von einem Betriebssystem verfolgt, wenn segmentierte Adressräume zu verwalten sind, sondern auch von einem Laufzeitsystem, wenn nämlich der ↑Haldenspeicher innerhalb eines Prozessadressraums verwaltet werden muss.

plotter (dt.) ↑Kurvenschreiber.

plug and play (dt.) sofort betriebsbereit; automatische Inbetriebnahme von einem ↑Peripheriegerät allgemein im Moment seines physischen Anschlusses an das ↑Rechensystem, speziell auch beim Einlegen von einen diesbezüglichen ↑Wechseldatenträger (z.B. ein Speicherstab, ↑USB) und der implizit folgenden ↑Befestigung von dem darauf gespeicherten ↑Dateisystem.

point of reference (dt.) ↑Bezugspunkt.

pointer (dt.) ↑Zeiger.

polling mode (dt.) ↑Aufrufbetrieb.

***port-mapped* I/O** (dt.) ↑isolierte Ein-/Ausgabe, mittelbar durch einen Anschluss (*port*).

position independent code (dt.) ↑positionsunabhängiger Kode.

positionsunabhängiger Kode (en.) ↑*position independent code*. Bezeichnung für den Kode (d.h., ↑Text) eines ↑Maschinenprogramms, der unabhängig von seiner Lage innerhalb eines ↑Adressraums vom ↑Prozessor ausgeführt werden kann. Ein solcher Kode verzichtet für gewöhnlich auf ↑absolute Adressen für alle programmbezogenen ↑Referenzen und unterliegt daher auch nicht der ↑Verlagerung durch spezielle Hilfsmittel der Software (↑Binder, ↑verschiebender Lader) oder Hardware (↑MMU). Stattdessen basiert der Programmkode auf ↑relative Adressen einerseits und, soweit vom Prozessur unterstützt, spezielle ↑Adressierungsarten andererseits, um bei der Programmausführung die absolute Adresse einer Referenz berechnen zu können Die Verschiebbarkeit eines solchen Kodes hat für gewöhnlich jedoch höhere ↑Gemeinkosten des betreffenden Programms zur Folge.

Exkurs Aufgrund dieser Gemeinkosten wird ein solcher Kode durch spezielle Übersetzungsschalter (z.B. `-f ↑PIC` für `gcc`) bei der ↑Kompilation erzeugt. ….

Kodebeispiel: Assembler

power-on reset (dt.) ↑Einschaltrückstellung.

PowerPC ↑Rechnerarchitektur einer von den Unternehmen Apple, IBM und Motorola gemeinsam spezifizierten ↑CPU (1991). Ein 64-Bit ↑RISC, von dem jedoch auch 32-Bit Varianten vor allem für den Bereich eingebetteter Systeme existieren.

präemptiv (en.) ↑*preemptive*. Einer sich bereits abzeichnenden Entwicklung zuvorkommend, vorsorglich, vorbeugend (Duden). Je nach Betrachtungsweise die Eigenschaft eines ↑Betriebssystems, einer ↑Ablaufplanung oder eines ↑Programmablaufs einen ↑Prozess bevorrechtigt den ↑Prozessor zuzuteilen (↑*dispatching*) und den laufenden Vorgang davon zu verdrängen (↑*preemption*). Dabei bilden der ↑Verdrängungsgrad, die ↑Einplanungslatenz und die ↑Einlastungslatenz des Betriebssystems, wesentliche Faktoren für die relative ↑Startzeit des betreffenden (bevorrechtigten) Prozesses.

präemptive Planung (en.) ↑*preemptive scheduling*. Modell der ↑Ablaufplanung, durch die ↑Prozesse gezwungen werden können, die Kontrolle über den ↑Prozessor ab- und an einen anderen Prozess weiterzugeben. Anders als ↑kooperative Planung kann einem im ↑Prozesszustand laufend befindlichen Prozess sein Prozessor zugunsten eines auf der ↑Bereitliste stehenden Prozesses entzogen werden (↑*preemption*). Beispiele für solch ein Verfahren sind ↑RR, ↑VRR, ↑SRTF und ↑MLFQ.
Je nach dem vor allem durch Struktur und Aufbau des ↑Betriebssystems möglichen ↑Verdrängungsgrad wird der erzwungene ↑Prozesswechsel mehr oder weniger zeitnah zu dem ↑Ereignis stattfinden, das die Ursache für die Neuzuteilung des Prozessors ist. Typischerweise ruft eine ↑Unterbrechungsanforderung dieses Ereignis hervor, in deren weiteren Behandlungsverlauf es dann zum Aufruf des ↑Planers kommen kann. ↑Einplanung und ↑Einlastung sind sowohl räumlich als auch zeitlich gekoppelt, wobei letztere jedoch immer nur dann zum zeitnahen Prozesswechsel führt, wenn erstere dies auch vorgibt. Kommt es zur Entscheidung, den

Planer aufzurufen, bestimmen ↑Einplanungslatenz und ↑Einlastungslatenz zusammen die Zeitspanne bis zum Prozessorentzug beziehungsweise zur Prozessorzuführung.

pre-paging (dt.) ↑Seitenvorabruf.

precompilation (dt.) ↑Vorübersetzung.

predictability (dt.) ↑Vorhersagbarkeit.

preemption (dt.) ↑Bevorrechtigung.

preemption level (dt.) ↑Verdrängungsebene, ↑Verdrängungsgrad.

preemption locking (dt.) ↑Verdrängungssperre.

preemption point (dt.) ↑Verdrängungspunkt.

preemptive (dt.) bevorrechtigt, ↑präemptiv.

preemptive scheduling (dt.) ↑präemptive Planung.

present bit (dt.) ↑Anwesenheitsbit.

Primärspeicher (en.) ↑*primary storage*. Klassifikation für ↑Registerspeicher, ↑Zwischenspeicher, ↑Notizblockspeicher und ↑Hauptspeicher/↑Arbeitsspeicher; „erstrangiger Speicher“, der zwar über keine große Kapazität verfügt, dafür jedoch eine niedrige ↑Zugriffszeit mit sich bringt.

primary storage (dt.) ↑Primärspeicher, ↑Vordergrundspeicher.

Primitivbefehl (en.) ↑*primitive command*. Art der technischen Auslegung eines ↑Systemaufrufbefehls, der zum Abruf seines ↑Operationskodes und gegebenenfalls zugehöriger Operanden nur einen vergleichsweise einfachen (primitiven) ↑Befehlszyklus im ↑Systemaufrufzuteiler bedingt (s. w. u.). Aufbau und Struktur eines solchen Befehls, dessen Ausführung einen ↑Systemaufruf bewirkt, ähneln einem für ↑RISC typischen ↑Maschinenbefehl.
Gegensatz zum ↑Komplexbefehl trotz dem gemeinsamen Merkmal einer bestimmten ↑Aktionsfolge zur Operandenauswertung, der bei ↑Mehrfädigkeit im ↑Maschinenprogramm eine konsistente Sicht auf die Operanden zuzusichern ist. Anders als beim Komplexbefehl geschieht diese Auswertung jedoch im ↑Systemaufrufstumpf, auf ↑Benutzerebene. Nur dieser ist der Kontext und die Semantik der Aktionsfolge bekannt, weshalb als implizite Maßnahme zur Konsistenzsicherung durch das ↑Betriebssystem hier nur noch ↑kooperative Planung helfen könnte. Der die Anweisungen zur Operandenauswertung umfassende Systemaufrufbefehl bildet eine ↑Elementaroperation, die, falls erforderlich und das Betriebssystem kooperative Planung nicht zur Verfügung stellt, nur direkt durch ↑Synchronisierung im Maschinenprogramm selbst abgesichert und atomar gestaltet werden muss.
Vor dem Wechsel von der Benutzerebene hin zur ↑Systemebene durchläuft der ↑Prozess im Maschinenprogramm (genauer: im Systemaufrufstumpf) für gewöhnlich eine Sequenz von Maschinenbefehlen, um sowohl die Parameter als auch den Operationskode als Argumente zur Übergabe an das Betriebssystem aufzubereiten. Typischerweise kommen hier ↑Prozessorregister als Behälter zum Einsatz, um damit im Moment des Wechsels der Systemebene die entsprechenden Argumente zur direkten Übernahme und Weiterverarbeitung bereitstellen zu können. Dazu wird im Allgemeinen eine feste Zuordnung zwischen Argument und Prozessorregister getroffen, die aus Gründen der Effizienz für alle Systemaufrufe gilt. Nachfolgend ein Beispiel, dessen Argumentenübergabe an ↑Linux angelehnt ist (↑x86):

```
movl op6, %ebp          # 6th operand
movl op5, %edi          # 5th operand
movl op4, %esi          # 4th operand
movl op3, %edx          # 3rd operand
movl op2, %ecx          # 2nd operand
movl op1, %ebx          # 1st operand
movl opc, %eax          # operation code for the operating system
sos                     # switch to operating system
↑Systemaufruffehler?    # aftercare of the system call
```

Der dem ↑Unterbrechungsbefehl (`sos`) folgende Maschinenbefehl prüft für gewöhnlich ab, ob ein Systemaufruffehler aufgetreten ist. Gegebenenfalls wird dann die Fehlerbehandlung im Kontext des Maschinenprogramms eingeleitet (vgl. S. 338).
Die für die hier betrachtete Art von Systemaufrufbefehl erfolgende Operandenauswertung auf Benutzerebene bedeutet, dass nicht der Systemaufrufzuteiler, sondern der Systemaufrufstumpf den Abruf der Operanden bewerkstelligt. Durch die direkte Einbindung des Stumpfes in das Maschinenprogramm und dem daraus resultierenden gemeinsamen ↑Adressraum ist der Operandenzugriff sehr einfach. Die Systemaufrufparameter sind ausnahmslos *Wertparameter*, die bei der Operandenauswertung gewonnen und als Kopien an den Systemaufrufzuteiler in Prozessorregistern übergeben werden (*call-by-value*).
Das Gegenstück im Systemaufrufzuteiler legt nur die Inhalte der Prozessorregister auf dem ↑Laufzeitstapel des Betriebssystems ab und komplettiert damit praktisch die Parameterübergabe an die Systemfunktion, bevor diese als gewöhnliches ↑Unterprogramm aufgerufen wird. Dabei identifiziert der Operationskode, genauer die in ihm enthaltene ↑Systemaufrufnummer, letztlich die aufzurufende Systemfunktion. Die Funktionsadresse steht in einem ↑Sprungvektor, der mit der Systemaufrufnummer als Index einem Zeigerfeld entnommen wird. Nachfolgend die entsprechende Sequenz von Maschinenbefehlen, um eine Systemfunktion mit den tatsächlichen Parametern zu versorgen und aufzurufen:

```
pushl %ebp              # pass 6th operand or just save register
pushl %edi              # pass 5th operand or       "
pushl %esi              # pass 4th operand or       "
pushl %edx              # pass 3rd operand or       "
pushl %ecx              # pass 2nd operand or       "
pushl %ebx              # pass 1st operand or       "
movzbl %al, %eax        # isolate system-call number, an array index
call *vector(,%eax,4)   # invoke system function
```

Auch wenn nicht jede Systemfunktion dieselbe Parameteranzahl aufweisen wird, ist der hier praktizierte Ansatz, nämlich die Inhalte aller zur Parameterübergabe verfügbaren Prozessorregister unabhängig von der Systemaufrufnummer zu stapeln, der einfachste Weg — dem auch Linux folgt. Nicht nur die Parameter werden damit übergeben, sondern es erfolgt gleichsam die Sicherung der Inhalte jener Prozessorregister, die im Systemaufrufstumpf nicht zur Parameterübergabe benötigt wurden. Anschließend wird die Systemaufrufnummer aus dem Operationskode isoliert, um schließlich die Systemfunktion indirekt über ihren Sprungvektor aufzurufen. Die Vektortabelle hat 2^v Einträge, wobei der Wert für v der Länge der ↑Bitleiste für die Systemaufrufnummer im Operationskode entspricht. Hier gilt $v = 8$, was bis zu 256 Systemaufrufe ermöglicht — eine für gewöhnlich ausreichende Anzahl: Ausnahmen bestätigen die Regel, wie Linux (dessen Wert für `__NR_syscalls` einigermaßen variiert, die Zahl 376 dafür aber nicht ungewöhnlich ist) zeigt. Ungenutzte Vektoren verweisen auf eine Systemfunktion (`nosys` in ↑UNIX, `sys_ni_syscall` in Linux), die nichts tut außer einen Fehlerkode (`ENOSYS`) zu liefern.
Im Moment der durch einen Systemaufruf bewirkten ↑Unterbrechung eines Prozesses spiegeln die Inhalte der Prozessorregister für gewöhnlich den aktuellen ↑Prozessorstatus eben dieses Prozesses wider. Dieser Prozessorstatus ist jedoch nicht identisch mit dem Prozessorstatus, den der Prozess besitzt, wenn er den Systemaufrufbefehl zur Ausführung bringt und damit

beispielsweise auch die Domäne oder den Kontext seines ↑Benutzerprogramms verlässt. Die Inhalte der in diesem Befehl zur Argumentenübergabe an den Systemaufrufzuteiler verwendeten Prozessorregister sind im Laufzeitstapel des Prozesses auf Benutzerebene zu sichern und vor Beendigung dieses Befehls von dort wieder herzustellen (vgl. `write`, S. 340). Dies ist ein einfacher Vorgang, der jedoch dazu führt, dass der auf Systemebene gleich zum Anfang gesicherte Prozessorstatus (s. oben) nicht mit dem Prozessorstatus des Prozesses im Benutzerprogrammkontext gleichzusetzen ist. Werden Prozessorregister, deren Inhalte zum Status des Benutzerprogramms gehören, in den Stümpfen zur Argumentenübergabe genutzt, verteilt sich letztlich dieser gesicherte Status über zwei Laufzeitstapel je nach Systemaufruf mehr oder weniger stark: nämlich dem Stapel der Benutzerebene für die zur Argumentenübergabe freigemachten Prozessorregister einerseits und dem der Systemebene für alle anderen Prozessorregister andererseits. Die oben gezeigte Sequenz von `push`-Operationen sorgt für einen Schnappschuss, der zwar den Prozessorstatus der Benutzerebene festhält, jedoch nicht zwingend auch dem Kontext des Benutzerprogramms zuzurechnen ist.
In logischer Hinsicht geschieht somit bereits der Wechsel zur Systemebene, sobald der Prozess den Systemaufrufstumpf betritt und damit ein gewisser Teil von seinem Prozessorstatus auf den Stapel (der Benutzerebene) gesichert wird. In physischer Hinsicht, also wirklich, erfolgt dieser Wechsel aber erst mit Ausführung des Unterbrechungsbefehls (`sos`) durch die ↑CPU. Damit das Betriebssystem Buch führen kann über den Prozessorstaus eines Prozesses im Benutzerprogrammkontext, ist dieser logische Schritt zu beachten und die in diesem Moment in Abhängigkeit vom effektiven Systemaufruf im zugehörigen Stumpf gesicherten Prozessorregisterinhalte. Dies betrifft nicht nur Arbeitsregister, sondern insbesondere auch den als Rücksprungadresse (der den Systemaufrufstumpf repräsentierenden Prozedur im Maschinenprogramm) für gewöhnlich bereits zuvor gesicherten Inhalt des ↑Befehlszählers.
Dieses registerbasierte Verfahren macht eine Einschränkung in der Anzahl von Argumenten, die einem Systemaufruf direkt übergeben werden können: im gegebenen Fall nämlich sechs Parameter. Sollten mehr Parameter übergeben werden müssen als Prozessorregister dafür zur Verfügung stehen, fällt ein zusätzlicher Schritt im Systemaufrufzuteiler an. Damit kommt letztlich ein *indirekter Systemaufruf* zur Wirkung, indem die Parameter (allerdings anders als bspw. gängig für Komplexbefehle, vgl. S. 341) indirekt über den zum Prozessorstatus der Benutzerebene gehörenden ↑Stapelzeiger (↑*user mode* ↑*stack pointer*) zugänglich gemacht werden. Je nach Auslegung eines solchen Systemaufrufs werden in solch einer Situation entweder alle Parameter indirekt übergeben/abgerufen oder nur die Parameter, für die keine Prozessorregister zur direkten Übergabe mehr zur Verfügung stehen.
Der Systemaufrufstumpf ist in programmiersprachlicher Hinsicht ein Unterprogramm, dem bereits alle Parameter für den Systemaufruf zu übergeben sind. Für den hier zugrunde gelegten Prozessor (x86), der als ↑CISC nämlich eine gewöhnliche Stapelmaschine darstellt, bedeutet dies, dass all diese Parameter zunächst auf den Laufzeitstapel des den Systemaufrumpfstumpf aktivierenden Prozesses der Benutzerebene gelangen. Des Weiteren ist der Operationskode beziehungsweise die Systemaufrufnummer ein eindeutiger Indikator eben auch für die Parameteranzahl eines Systemaufrufs. Erkennt der Systemaufrufzuteiler die Notwendigkeit, mehr als sechs Argumente an die Systemfunktion zu übergeben, kopiert er diese direkt vom Laufzeitstapel der Benutzerebene auf den Laufzeitstapel der Systemebene. In dieser Situation verfährt der Zuteiler nicht unähnlich zur Parameterübergabe bei einem Komplexbefehl, nur dass die Argumente dann nicht aus dem ↑Textsegment, sondern aus dem ↑Stapelsegment des betreffenden Prozesses der Benutzerebene zu lesen sind (vgl. S. 137).

primitive command (dt.) ↑Primitivbefehl.

primitive Koroutine Bezeichnung für eine ↑Koroutine, die zusammen mit anderen ihrer Art ein und denselben ↑Laufzeitstapel besitzt (analog zur ↑Routine). Die ↑Handhabe für einen ↑Handlungsstrang innerhalb einer solchen Koroutine ist ein ↑Befehlszählerwert. Dieser Wert ist der Zeiger auf den ↑Maschinenbefehl, der nach erfolgtem ↑Koroutinenwechsel als nächster ausgeführt wird. Anders als eine ↑komplexe Koroutine.

principle of locality (dt.) ↑Lokalitätsprinzip.

prior knowledge (dt.) ↑Vorwissen.

Priorität (en.) ↑*priority.* Stellenwert, den ein ↑Prozess innerhalb einer Rangfolge einnimmt (in Anlehnung an den Duden). Dieser Wert kann *statisch* (unveränderlich) oder *dynamisch* sein. Im statischen Fall, erhält der Prozess spätestens bei seiner Erzeugung einen bestimmten ↑Rang und behält diesen bis zu seiner Zerstörung. Demgegenüber wird im dynamischen Fall der Rang eines Prozesses während seiner Lebenszeit vom ↑Einplanungsalgorithmus aktualisiert. Beispielsweise kann bei ↑Echtzeitbetrieb der Rang eines Prozesses umso mehr ansteigen, je weniger Zeit dem Prozess bis zum Termin bleibt (↑EDF). Im ↑Dialogbetrieb kann der Rang eines Prozesses umso mehr ansteigen, je kürzer sein ↑Rechenstoß ist. Dem liegt die Annahme zugrunde, dass ein kurzer Rechenstoß auf einen ein-/ausgaberührigen und damit interaktiven Prozess hindeutet, dem eine möglichst kurze ↑Antwortzeit zu geben ist. Umgekehrt kann der Rang eines Prozesses umso mehr abnehmen, je länger sein Rechenstoß und damit auch seine Phase der Interaktionslosigkeit dauert.

Prioritätsobergrenze (en.) ↑*priority ceiling.* Maßnahme zur Vorbeugung einer *unkontrollierten* ↑Prioritätsumkehr (*unbounded* ↑*priority inversion*). Das darauf basierende Protokoll (↑PCP) verhindert nicht nur unnötige Ketten der Blockierung (↑*chain blocking*) eines ↑Prozesses, die durch Verhängung und Aufhebung von ↑Prozesssperren entstehen können, sondern beugt zusätzlich auch noch möglichen ↑Verklemmungen vor (↑*deadlock prevention*). Im Grunde ein auf ↑Prioritätsvererbung zurückgehendes Verfahren, dass jedoch gerade in Hinblick auf diese beiden zuletzt genannten Aspekte bezüglich Prozesssperren und Verklemmungen eine wesentliche Erweiterung darstellt.
Voraussetzung für das Verfahren ist jedoch ↑Vorwissen zur Mächtigkeit der Menge jener Prozesse, die ein bestimmtes ↑unteilbares Betriebsmittel, R, gemeinsam haben. Für jedes einzelne R ist diese Menge zu bestimmen, aus der dann jeweils die ↑Priorität des höchst priorisierten Prozesses die ↑Obergrenze definiert, die R als Betriebsmittelattribut fest zugeordnet wird. Diese *Betriebsmittelobergrenze* wird auch als ↑Grenzpriorität bezeichnet.
Neben der im Voraus festgelegten Betriebsmittelobergrenze basiert das Verfahren auf eine zur Laufzeit variierende *Systemobergrenze*, nämlich der höchsten Obergrenze aller zum Zeitpunkt gesperrten unteilbaren Betriebsmittel. Die Verwaltung der Systemobergrenze verwendet eine *globale Gebrauchsliste*, auf der die gesperrten Betriebsmittel mit abnehmender Grenzpriorität sortiert aufgeführt sind. Damit definiert die Grenzpriorität des jeweils am Listenanfang stehenden Betriebsmittels die Systemobergrenze. Die Betriebsmittelzuteilung legt einen neuen Listeneintrag an, die Systemobergrenze kann sich erhöhen, und die Betriebsmittelfreigabe streicht einen Listeneintrag, die Systemobergrenze kann sich absenken.
Das allgemeine Regelwerk (*original* PCP) zur Betriebsmittelvergabe an den momentan stattfindenden Prozess (P_h) sieht sodann folgende Schritte vor:

1. R ist gesperrt, dann wird P_h blockiert und seine Priorität an den R sperrenden Prozess (P_l) vererbt: $P_l \mapsto P_l^h$. Die Systemobergrenze bleibt unverändert.
2. R ist nicht gesperrt, dann erhält P_h das Betriebsmittel unter bestimmten Voraussetzungen zugeteilt. Hierzu muss eine der beiden folgenden Bedingungen zutreffen:
   (a) die Priorität von P_h ist größer als die Obergrenze des Systems oder
   (b) P_h muss mindestens ein unteilbares Betriebsmittel, dessen Obergrenze der des Systems entspricht, bereits belegen und damit gesperrt haben.

   Trifft Bedingung 2a zu, erhöht sich die Systemobergrenze. Trifft keine dieser Bedingungen zu, bleibt R ungesperrt und, da keine Betriebsmittelzuteilung erfolgte, die Systemobergrenze demzufolge unverändert (so auch, falls nur Bdg. 2b zutrifft). In der Situation vererbt P_h seine Priorität an den Prozess (P_l), der mit P_h das angeforderte Betriebsmittel (R) gemeinsam benutzt, jedoch R vor Bereitstellung von P_h hat belegen können: $P_l \mapsto P_l^h$. P_h wird nach Vererbung seiner Priorität suspendiert. Dazu kommt er zurück

auf die ↑Bereitliste, und zwar, da für Prozesse derselben Priorität ↑FCFS gilt, nach P_l^h, der zuvor von einem Prozess höherer Priorität vom Prozessor verdrängt worden sein musste und daher selbst auf dieser Liste steht.

P_l^h nimmt seine ursprüngliche Priorität wieder an, sobald er alle Betriebsmittel, deren jeweilige Obergrenze größer oder gleich seiner effektiven/geerbten (d.h., P_h entsprechenden) Priorität ist, freigegeben hat. Hinweis dazu geben die auf der Gebrauchsliste stehenden gesperrten Betriebsmittel, die dort entsprechend ihrer jeweiligen Grenzpriorität nach abnehmenden Werten sortiert vermerkt sind. Auf dieser Grundlage geschieht sodann die Rücknahme einer erhöhten (geerbten) Priorität, indem nun geprüft werden kann, wann die effektive Priorität des ein Betriebsmittel freigebenden Prozesses (P_l^h) größer ist als die Grenzpriorität des oben auf der Gebrauchsliste stehenden Betriebsmittels. Sobald durch erfolgende Betriebsmittelfreigabe der erste Listeneintrag eine kleinere Grenzpriorität aufweist, nimmt der freigebende Prozess wieder seine ursprüngliche Priorität an: $P_l^h \mapsto P_l$. Als direkte Folge kann in dem Moment die Systemobergrenze auch einen niedrigeren Wert erhalten, nämlich wenn durch die Betriebsmittelfreigabe der oberste Eintrag auf der Gebrauchsliste gestrichen wurde.
Dass ein nicht gesperrtes Betriebsmittel bei Anforderung nicht vergeben und der anfordernde Prozess stattdessen suspendiert wird, begründet eine weitere Form der Blockierung (↑*ceiling blocking*). Dadurch wird sowohl der Kettenbildung von Blockierungs- und Deblockierungsvorgängen als auch einer möglichen Verklemmung der Prozesse untereinander vorgebeugt. Mit dieser Vorgehensweise kommt allerdings auch das (im Gegensatz zu ↑PIP) *pessimistische Verfahren* zum Vorschein, nämlich manchmal eine in Bezug auf den Betriebsmittelzustand eigentlich unnötige ↑Prozesssperre zu verhängen. Grundsätzlich ist festzuhalten, dass der jeweils höchst priorisierte Prozess (P_h) höchstens für die Dauer der längsten, ihn betreffenden Betriebsmittelsperre den ↑Prozessor nicht zugeteilt bekommt, weil er entweder blockiert oder suspendiert wurde.
Das beschriebene Originalverfahren resultiert in eine laufzeitaufwendige Implementierung, insbesondere aufgrund des komplexen Regelwerks und der mitzuführenden Listen für die gesperrten Betriebsmittel einerseits und die suspendierten Prozesse andererseits, deren Einträge zudem nach absteigenden Prioritätswerten zu sortieren sind. Jede einzelne Betriebsmittelanforderung oder -freigabe erfordert einen Suchlauf auf der Gebrauchsliste und jede Prozesssuspendierung erfordert einen Suchlauf auf der Bereitliste. Dies verursacht ↑Gemeinkosten, die in zeitlicher Hinsicht wegen der im Voraus bekannten Prozess- und Betriebsmittelanzahl zwar nach oben beschränkt sind, jedoch zu Laufzeitvarianzen und ↑Latenzen führen. Weitere Gemeinkosten ergeben sich wegen unnötiger ↑Prozesswechsel, verursacht entweder durch direkte Blockierung (Bdg. 1) oder Suspendierung (Bdg. 2a und 2b).
Eine Vereinfachung der Implementierung des Verfahrens besteht nun darin, die Priorität eines Prozesses unmittelbar im Moment der Betriebsmittelanforderung auf die Grenzpriorität des dann zugeteilten unteilbaren Betriebsmittels zu setzen (*immediate* PCP). Dadurch kann dieser Prozess nicht mehr durch höher priorisierte Prozesse, die dieses Betriebsmittel ebenfalls gebrauchen wollen, vom Prozessor verdrängt werden. Als direkte Konsequenz daraus sind weniger Prozesswechsel zu erwarten. Eine Absenkung der Prozesspriorität erfolgt dann bei der Freigabe des letzten Betriebsmittels, dessen Grenzpriorität die Prioritätserhöhung bewirkte. Die dazu notwendige Buchführung basiert auf einen *Betriebsmittelstoß* (↑*stack*) gesperrter Betriebsmittel, die bei der Zuteilung darauf abgelegt (*push*) und bei der Freigabe davon wieder entfernt (*pop*) werden. Ist nach Freigabe die Grenzpriorität des dann oben auf dem Stoß liegenden Betriebsmittels kleiner als die aktuelle Prozesspriorität, erhält der aktuelle Prozess seine vorige Priorität zurück. Dazu ist die Prozesspriorität bei der Betriebsmittelzuteilung zu sichern. Da jedes dieser Betriebsmittel zu einem Zeitpunkt nur von einem Prozess exklusiv belegt sein kann, bietet sich zur Sicherung dieser Priorität der ↑Deskriptor des jeweils zugeteilten Betriebsmittels an.
Die Variante zur unmittelbaren Zuweisung der Grenzpriorität an den zu dem Zeitpunkt gleichfalls höchst priorisierten Prozeß lässt sich verschlanken, indem eine enge Verzahnung dieser Art der Priorisierung mit der ↑Prozesseinplanung geschieht (*stack-based* PCP). In

der dann strikt betriebsmittelbezogen stapelbasiert ablaufenden ↑Einlastung des Prozessors werden die Prozesse selbst, nicht die gesperrten Betriebsmittel, auf eine einen *Prozessstoß* implementierende Bereitliste gesetzt. Die Listenposition eines Prozesses ist dann bestimmt durch die höchste Grenzpriorität aller Betriebsmittel, die dieser Prozess jeweils benötigt. Die Priorität eines Prozesses ist damit selbst durch eine Obergrenze bestimmt und sie ist fest. Je weiter oben auf dem Stoß ein Prozess liegt, desto höher ist seine Priorität. Für den nächsten vom Stoß zu nehmenden Prozess gilt sodann, dass alle von ihm benötigten Betriebsmittel frei (d.h., nicht gesperrt) sind. Das bedeutet jedoch nicht, dass bei der Einlastung implizit eine Vorbelegung dieser Betriebsmittel geschehen muss. Die Betriebsmittel können immer noch bei Bedarf angefordert und wieder freigegeben werden (↑SRP).
Davon ausgehend nimmt ein Prozeß immer erst dann seine ↑Aufgabe auf, wenn die Systemobergrenze unter die Priorität des oben auf dem Stoß liegenden Prozesses gefallen ist. Sobald nun der Prozessor frei wird für den nächsten Prozess, sind auch alle von diesem Prozess benötigten Betriebsmittel erklärtermaßen frei. Der Prozess erhält den Prozessor zugeteilt, woraufhin die Systemobergrenze den Wert der Obergrenze/Priorität eben dieses Prozesses annimmt. Hat der Prozess seine Aufgabe fertiggestellt, so erwartet er die nächste Aufgabenstellung indem er sich suspendiert und dadurch den Prozessor für den nächsten oben auf dem Stoß liegenden Prozess freigibt, woraufhin die Systemobergrenze aktualisiert wird.
Dieses Verfahren arbeitet ebenfalls verdrängend. Wird ein Prozess bereitgestellt, dessen Obergrenze/Priorität höher ist als die Systemobergrenze, verdrängt dieser den gegenwärtigen Prozess vom Prozessor: der verdrängte Prozess setzt seine Aufgabe aus, kommt oben auf den Stoß (↑LIFO bzw. ↑LCFS), und der verdrängende Prozess nimmt sofort seine Aufgabe auf. Gleichwohl geht jeder Prozess von sich aus, nämlich gemäß Vorschrift des ihn bestimmenden ↑Programms, einer Aufgabe immer bis zur Fertigstellung nach (↑*run to completion*).

Prioritätsumkehr (en.) ↑*priority inversion*. Auswirkung eines bestimmten Geschehens, das einem nachrangigen ↑Prozess den Vorzug gegenüber einen vorrangigen Prozess gibt und dadurch Abläufe entgegen der durch der zugrunde liegenden ↑Vorrangplanung eigentlich vorgegebenen Reihenfolge zustande kommen. Im Ergebnis behält ein Prozess niedriger ↑Priorität den ↑Prozessor, obwohl ein Prozess höherer Priorität darauf wartet, den Prozessor zugeteilt zu bekommen.
Die Ursache dafür ist möglicherweise eine zuvor geschehene ↑Prioritätsverletzung bei der Zuteilung eines ↑Betriebsmittels an einen darauf wartenden Prozess. Für gewöhnlich wird ein solcher umgekehrter Ablauf jedoch nicht mit einer irrtümlichen Entscheidung etwa als Folge eines Programmierfehlers (↑*bug*) in Verbindung gebracht, sondern er ist vielmehr ein Merkmal (*feature*) eines vorranggesteuerten ↑Rechensystems, in dem wenigstens ein ↑unteilbares Betriebsmittel zugleich von mehreren Prozessen unterschiedlicher Prioritäten in unvorhersehbarer Reihenfolge angefordert werden kann.
Angenommen, ein Prozess niedriger Priorität, P_l, hat ein unteilbares Betriebsmittel, R, zugeteilt bekommen. P_l belegt R und geht normal seiner ↑Aufgabe nach. Bevor nun P_l das Betriebsmittel (R) freigibt und damit wieder zur Verfügung stellt, wird ein Prozess höherer Priorität, P_h, bereitgestellt. Für die *relative Priorität* in Bezug auf P_l und P_h gilt: $P_l < P_h$, das heißt, P_h dominiert P_l. Bei Vorrangplanung bewirkt dies die ↑Verdrängung von P_l, dem damit der Prozessor, aber nicht R entzogen wird. P_h fordert nun seinerseits R an, das jedoch im Besitz von P_l ist und weiterhin bleibt. Da P_h allerdings R selbst für das eigene Vorankommen benötigt, wird er auf die Freigabe durch P_l warten müssen. P_h blockiert, gibt den Prozessor frei, so dass P_l wieder fortgesetzt werden kann. In dem Moment kommt es zur Umkehrung der relativen Priorität: $P_l > P_h$, das heißt, P_h wird von P_l dominiert.
Diese *direkte* Form der Umkehrung kann durch ein geeignetes Protokoll zwischen P_l und P_h so kontrolliert werden, dass P_l das Betriebsmittel (R) unverzüglich an P_h abtritt und letzterer damit nicht übermäßig lang blockiert bleibt. Beispielsweise könnten ↑Verdrängungspunkte vorgesehen werden, an denen P_l prüft, ob eine Bedarfsanmeldung durch P_h vorliegt, dann gegebenenfalls einen Wiederaufsetzpunkt (↑*continuation*) für sich definiert und R freigibt. Im Moment der Freigabe von R würde P_l sodann durch P_h verdrängt werden, der dann zügig

R belegt und somit seine Aufgabe noch rechtzeitig erfüllen kann. Allerdings hilft ein solches Protokoll nicht gegen die *indirekte* Variante, die nämlich eine *unkontrollierte Umkehrung* (*unbounded* ↑*priority inversion*) der relativen Prioritäten herbeiführen kann.
Angenommen, ein dritter Prozess mittlerer Priorität, P_m, kommt ins Spiel: $P_l < P_m < P_h$, das heißt, P_h dominiert P_m, der seinerseits P_l dominiert. P_m wird bereitgestellt nachdem P_h wegen P_l, wie zuvor beschrieben, blockiert ist und bevor P_l das von P_h benötigte Betriebsmittel (R) freigibt. Zwischen P_m einerseits und P_l und P_h anderseits besteht keine Abhängigkeit in Bezug auf R, das nur von P_l und P_h gemeinsam benutzte unteilbare Betriebsmittel. Wohl aber wird bei Vorrangplanung P_l durch P_m vom Prozessor verdrängt. Solange P_m den Prozessor nicht abgibt, fährt P_l nicht weiter fort und wird somit auch davon abgehalten, das von P_h benötigte Betriebsmittel R freizugeben. In Bezug auf die relative Priorität bedeutet dies: $P_m > P_l > P_h$, das heißt, P_h wird von P_m dominiert. Ein direkt zwischen P_l und P_h vereinbartes Protokoll zur prompten Freigabe von R bleibt wirkungslos, da P_m dieses Betriebsmittel nicht benötigt und daher auch keine Kenntnis von diesem Protokoll hat. Durch P_l und P_h lassen sich die ↑Aktionen von P_m nicht kontrollieren, so dass die ↑Wartezeit von P_h auf Freigabe von R unbestimmt bleibt.
Wenn die ↑Bereitzeiten von P_l, P_m und P_h von unvorhersehbaren ↑Ereignissen abhängig sind (↑*event-triggered system*), lässt sich ein solches Szenario allein durch Vorrangplanung nicht verhindern. Besonders problematisch ist dies für den ↑Echtzeitbetrieb, der in der Priorität eines Prozesses für gewöhnlich auch die Wichtigkeit einer ↑Termineinhaltung zum Ausdruck bringt. So kann P_h Gefahr laufen, durch die ihm wegen P_m auferlegte Wartezeit, einen wichtigen ↑Termin zu verpassen. Für die Lösung dieses Problems gibt es im Grunde die beiden folgenden Ansätze:

1. Im Moment der Zuteilung von R an P_l wird eine ↑Verdrängungssperre erhoben. Folglich kann P_l nicht durch P_m vom Prozessor verdrängt werden, wodurch P_h auch keine unbestimmte Verzögerung durch P_m erfährt. Diese Sperre wird aufgehoben, sobald P_l das Betriebsmittel (R) abgibt. Die Wartezeit von P_h ist bestimmbar, wenn für die ↑Ausführungszeit der Aufgabe von P_l bis zur Freigabe von R der ungünstigste Fall (↑WCET) in Betracht gezogen werden kann. Ist jedoch die ↑Zwischenankunftszeit von ↑Unterbrechungen unbekannt, bleibt die Wartezeit unbestimmt. In dem Fall muss, anstatt der Verdrängungssperre, eine ↑Unterbrechungssperre gesetzt werden. Grundsätzlich implizieren beide Techniken aber die Aussperrung von R unabhängigen Prozessen, deren Prioritäten insbesondere auch noch höher sein können als die von P_h.
2. Im Moment der Anforderung von R erfolgt eine gezielte Prioritätserhöhung, die P_m davon abhält, den Prozessor zugeteilt zu bekommen. Hierzu gibt es zwei prinzipielle Verfahrensweisen, nämlich ↑Prioritätsvererbung oder das Setzen einer ↑Prioritätsobergrenze. In beiden Varianten wird die Priorität von P_l erhöht, und zwar wenn P_h das Betriebsmittel (R) anfordert, ihm die Zuteilung von R aber verwehrt werden muss und er daraufhin blockiert:
   (a) weil R bereits von P_l belegt ist, dann wird die Priorität von P_h an P_l vererbt, oder
   (b) weil die Priorität von P_h nicht höher ist als die höchste Prioritätsobergrenze aller in dem Moment belegten Betriebsmittel im System und er auch kein Betriebsmittel hält, das diese *Systemobergrenze* definiert hat (nur die zweite Variante).

   Die zweite Variante geht über die noch recht einfache Vererbung hinaus und verbindet mit R eine Prioritätsobergrenze, die bestimmt ist durch die höchste Priorität aller von der Verfügbarkeit von R abhängigen Prozesse. Im Falle von P_l und P_h, die R gemeinsam haben, erhält R als Prioritätsobergrenze somit die Priorität von P_h. Sobald nun einer dieser Prozesse R anfordert und R noch frei ist, wird die Prioritätsobergrenze von R als Systemobergrenze geltend gemacht — aber nur, falls Bedingung 2b <u>nicht</u> zutrifft (↑*deadlock prevention*) und R damit überhaupt zugeteilt werden dürfte. Im Gegensatz zur bloßen Prioritätsvererbung ist zur Bestimmung einer Prioritätsobergrenze ↑Vorwissen über die von den Prozessen beanspruchten Betriebsmittel erforderlich. Beide

Varianten sorgen dafür, dass nur direkt oder indirekt voneinander abhängige Prozesse zeitweilig von der Prozessorzuteilung ausgenommen werden.

Für Prozesse, die in ↑Echtzeit stattfinden müssen, ist die Nichtbeachtung der möglichen Umkehrung relativer Prozessprioritäten bei Mitbenutzung (*sharing*) unteilbarer Betriebsmittel ein ↑Fehler, der schwerwiegende Folgen nach sich ziehen kann. Für einen Prozess anderer Art ergibt sich daraus vielleicht nur eine Anomalie, die in Anbetracht der durch die Lösungsansätze bedingten Nebenwirkungen (↑*background noise*) gut tolerierbar ist.

Prioritätsvererbung (en.) ↑*priority inheritance*. Maßnahme zur Vorbeugung einer *unkontrollierten* ↑Prioritätsumkehr (*unbounded* ↑*priority inversion*). Das darauf basierende Protokoll (↑PIP) wirkt nur auf ↑Prozesse, die über wenigstens ein ↑unteilbares Betriebsmittel miteinander gekoppelt sind oder den Fortschritt solcher gekoppelten Prozesse durch ↑Interferenz beeinträchtigen könnten.
Voraussetzung für diese unerwünschte Form der Inversion sind wenigstens drei Prozesse, P_l, P_m und P_h, unterschiedlicher ↑Prioritäten, wobei aber nur P_l und P_h ein unteilbares Betriebsmittel, R, gemeinsam benutzen. Für die *relative Priorität* in Bezug auf die drei Prozess gilt: $P_l < P_m < P_h$, das heißt, P_h dominiert sowohl P_m als auch P_l und P_m dominiert P_l. Fällt die ↑Bereitzeit von P_h oder P_m in die Zeitspanne während P_l den ↑Prozessor benutzt, wird P_l der Prozessor zugunsten von P_h oder P_m entzogen (↑*preemption*). Dieses Prinzip der dem Ansatz zugrunde liegenden und verdrängend wirkenden ↑Vorrangplanung gilt unabhängig von R.
Die unkontrollierte Prioritätsumkehr wird nun verhütet, indem P_l, wenn er R bereits belegt und folglich für P_h sperrt, im Moment der Anforderung von R durch P_h auf die Priorität von P_h angehoben wird: $P_l \mapsto P_l^h$. Dadurch kann P_l^h nicht mehr durch P_m vom Prozessor verdrängt werden, sobald P_h seine Betriebsmittelanforderung betreffs R gestellt hat. Für die relative Priorität gilt: $P_m < P_l^h \leq P_h$. Da R durch P_l gesperrt bleibt, erwartet P_h das ↑Ereignis der Betriebsmittelfreigabe im ↑Prozesszustand <u>blockiert</u>. Diese Situation hält an, bis R von P_l^h freigegeben wird. In Moment der Betriebsmittelfreigabe betreffs R kehrt P_l^h zur ursprünglichen Priorität zurück: $P_l^h \mapsto P_l$. Als Folge kommt es zur Deblockierung und damit Bereitstellung von P_h. Da P_h die höhere Priorität besitzt, wird die Vorrangplanung P_l zugunsten von P_h vom Prozessor verdrängen.
Damit sind zwei Arten der Blockierung eines höher priorisierten Prozesses möglich. Einerseits kann ein Prozess *direkt* blockieren, nämlich wenn er ein gesperrtes Betriebsmittel anfordert. Dieser Fall trifft auf P_h zu. Andererseits kann ein Prozess *indirekt* blockiert werden (*inheritance blocking*). Dies tritt im Fall von P_m auf, der durch P_l^h von der Zuteilung des Prozessors ausgesperrt wird.
Die Vererbung einer Prozesspriorität wirkt *transitiv*, und zwar im Falle von gestapelten Anforderungen höher priorisierter Prozesse an unteilbare Betriebsmittel, die von demselben niederprioren Prozess gehalten werden und damit für andere Prozesse gesperrt sind. Daraus folgt gleichfalls die indirekte (auch als transitiv bezeichnete) Blockierung aller mittel priorisierten Prozesse durch den „niederprioren“ Prozess, der nämlich aufgrund des Vererbungsvorgangs „ausnahmsweise“ eine höhere *effektive Priorität* erhalten hat.
Um dies zu verdeutlich sei einmal angenommen, zu den drei oben genannten Prozessen kommen zwei unteilbare Betriebsmittel, B_1 und B_2, ins Spiel. P_l schreitet voran, belegt B_1 und wird von P_m vom Prozessor verdrängt. P_m schreitet voran, belegt B_2 und wird von P_h vom Prozessor verdrängt. P_h schreitet voran, fordert B_2 an und blockiert, woraufhin seine Priorität (h) an P_m vererbt wird: $P_m \mapsto P_m^h$. Folglich nimmt P_m^h seine Aktivität wieder auf. P_m^h fordert sodann B_1, das aber gesperrt ist, an und blockiert, woraufhin seine geerbte/effektive Priorität (h) an P_l weitervererbt wird: $P_l \mapsto P_l^h$.
Die durch ein belegtes unteilbares Betriebsmittel hervorgerufene ↑Sperrzeit für Prozesse, die dieses Betriebsmittel anfordern, ist nach oben begrenzt, wenn (a) die Anzahl der auf die Zuteilung dieses Betriebsmittels wartenden Prozesse begrenzt ist und (b) diese Prozesse keine ↑Verklemmungen erleiden. Die *obere Grenze* leitet sich folgt ab:

- Für jedes einzelne gesperrte Betriebsmittel, R, wird ein dieses Betriebsmittel anfordernder höher priorisierter Prozess (P_h) bis zur Aufhebung der bestehenden ↑Betriebsmittelsperre von dem niedrig priorisierten Prozess (P_l) nur einmal gesperrt und damit auch nur einmal direkt blockiert ($P_l \mapsto P_l^h$). Sobald R freigegeben wird, deblockiert P_h und verdrängt den dadurch in seiner Priorität wieder abgestuften Prozess $P_l^h \mapsto P_l$ vom Prozessor. Sollte P_h im weiteren Verlauf nach erfolgter Freigabe von R dieses Betriebsmittel erneut anfordern, konnte kein anderer Prozess niedrigerer Priorität (weder P_m noch P_l) in der Lage gewesen sein, R für sich selbst zu beanspruchen und damit für P_h wieder zu sperren.
- Für die indirekte Blockierung von P_m (wie auch jedem anderen mittel priorisierten Prozess) in Bezug auf R und einen durch Vererbung hochgestuften Prozess $P_l \mapsto P_l^h$ gilt ähnliches. Diese hält bis zur Rückstufung von $P_l^h \mapsto P_l$ auf die Ausgangspriorität an und endet im Moment der Freigabe von R. P_m ist dann nicht mehr wegen R indirekt blockiert, wird aber wegen P_h ganz normal nachrangig bei der Vorrangplanung behandelt und bekommt daher wegen seiner niedrigeren Priorität den Prozessor nicht zugeteilt. P_m könnte erst dann wieder in die indirekte Blockierung verfallen, wenn P_h wegen ein anderes Betriebsmittel R' gesperrt und dadurch direkt blockiert wird. Dann gilt in Bezug auf $R^{'}$ der im vorigen Aufzählungspunkt behandelte Aspekt für $R = R^{'}$.
- Der für diese ↑Prozesssperre verantwortliche Prozess niedriger Priorität (P_l) blockiert einen Prozess hoher Priorität (P_h) nur höchstens für die Dauer der durch ihn (P_l) bewirkten Betriebsmittelsperre. Diese Dauer ist begrenzt durch die *absolute* ↑Ausführungszeit (↑WCET) der mit dem Betriebsmittel umgehenden ↑Aufgabe des niederprioren Prozesses (P_l). In dieser Ausführungszeit sind allerdings Verzögerungen enthalten, die beispielsweise in der Verdrängung von P_l durch höher priorisierte Prozesse, die dann gegebenenfalls selbst wegen P_l blockieren werden, ihre Ursache haben und damit die Sperrzeit verlängern. Solche für gewöhnlich unvorhersehbaren Ereignisse erschweren die Berechnung der voraussichtlichen Sperrzeit von P_h, aber sie führen nicht dazu, dass P_h unvorhersehbare Blockierungen erdulden müsste.

Insgesamt kann in dem Szenario ein Prozess sehr wohl mehrmals blockieren, aber eben nur einmal pro gesperrtes Betriebsmittel und damit auch nur einmal durch einen anderen, niederprioren Prozess. Jedoch kann es zu einer Kette von Blockierungen und Deblockierungen kommen (↑*chain blocking*).
Angenommen $n > 1$ Prozesse unterschiedlicher Priorität sind im ↑Wettstreit um n unteilbare Betriebsmittel R_i, $1 \leq i \leq n$. Die Priorität eines jeweiligen Prozesses sei umgekehrt proportional zu $0 \leq k < n$, das heißt, P_0 hat die höchste und P_k hat die niedrigste Priorität. P_k startet, belegt R_i und wird anschließend von P_{k-1} verdrängt. P_{k-1} schreitet voran, belegt R_{i-1} und wird anschließend von P_{k-2} verdrängt. Dieser Ablauf setzt sich fort bis zu P_1, der P_2 verdrängt und dann R_1 belegt hat. Nun wird P_1 verdrängt durch P_0, der sodann R_i in der Reihenfolge $i = 1, 2, 3, \ldots, n$ anfordert. Dadurch kommt es zu einer Verkettung einzelner Vererbungsvorgänge in der Reihenfolge:

$$P_1 \mapsto P_1^0 \ \{A_1\} \ P_2 \mapsto P_2^0 \ \{A_2\} \ P_3 \mapsto P_3^0 \ \{A_3\} \ldots \{A_{k-1}\} \ P_k \mapsto P_k^0 \ \{A_k\},$$

mit A_i als die ↑Aktionen der Aufgabe, die der eigentlich niederpriore Prozess P_i^0 sodann mit höherer effektiver Priorität erledigt. Die Länge dieser Kette hinsichtlich der Anzahl logisch folgerichtiger Vererbungs-, Blockierungs- und Deblockierungsschritte bezüglich P_0 lässt sich noch gut bestimmen. Der für P_0 betrachtete schlimmste Fall begrenzt die Anzahl dieser Schritte auf $min(n, k)$, mit n für die Anzahl der von P_0 benötigten unteilbaren Betriebsmittel und k für die Anzahl der mit P_0 im Wettstreit um diese Betriebsmittel stehenden niederprioren Prozesse. Allerdings ist mit diesem Fall noch längst nicht die Zeit bestimmt, die P_0 insgesamt verzögert werden kann.
Mit jedem Vererbungsschritt ist die Blockierung von P_0 verbunden, einschließlich ↑Prozesswechsel hin zum jeweils niederprioren Prozess. Letzterer gibt mit Beendigung seiner Aufgabe

(A_i) das von P_0 benötigte Betriebsmittel frei, er wird dadurch zugunsten von P_0 verdrängt (Prozesswechsel). Im Vergleich zur eigentlich zu bearbeitenden Aufgabe versuchen diese Prozesswechsel wie auch die dazu stattfindende ↑Umplanung beträchtliche ↑Gemeinkosten. Hinzu kommt noch die WCET der Aktionen, die der jeweils niederpriore Prozess bis zur Freigabe des von P_0 benötigten Betriebsmittels verursacht und deren Abschätzung für gewöhnlich auf eine *Überapproximation* beruht. Dadurch erhöht sich die Sperrzeit für einen Prozess allgemein in nicht unerheblichem Maße. Sollte der (jeweils) höchst priorisierte Prozess (P_0) in solch eine Situation gelangen, ist das besonders problematisch und kritisch, da damit die Gefahr der Überschreitung eines von ihm einzuhaltenden wichtigen ↑Termins droht. Das Hin und Her von P_0 und somit die Welle von Prozesswechseln endet erst, nachdem P_0 seine eigene Aufgabe mit R_n zum Abschluss gebracht hat. Diese noch unnötig hohe Sperrzeit, die das Verfahren einem Prozess bescheren kann, wird durch die Einführung sogenannter ↑Prioritätsobergrenzen vermieden — mit dem zusätzlichen und höchst vorteilhaften Nebeneffekt der ↑Verklemmungsvorbeugung.

Prioritätsverletzung (en.) ↑*priority violation.* Fehlverhalten bei der Auswahl des ↑Prozesses, der als nächster ein ↑Betriebsmittel zugeteilt bekommen soll. Ein der ↑Vorrangplanung widersprechendes Verhalten, das sich dadurch äußert, dass der ausgewählte Prozess eben nicht der mit der höchsten Priorität auf der Auswahlliste war.
Jede Form von Vorrangplanung sortiert Prozesse (bzw. die von ihnen durchzuführenden ↑Aufgaben) nach der ↑Dringlichkeit ihrer Bearbeitung durch einen ↑Prozessor (Betriebsmittel). Je dringlicher ein Prozess, desto höher seine Priorität anderen Prozessen gegenüber. Für gewöhnlich steht der Prozess mit der höchsten Priorität immer am Anfang der Liste, gefolgt von Prozessen niederer Prioritäten. Die Auswahl fällt dann immer auf den Prozess am Listenanfang, der erklärtermaßen die höchste Priorität haben müsste.
Angenommen die ↑Warteliste, auf der ein Prozess stehen kann, wird nach ↑FCFS bedient und die ↑Bereitliste, auf die der Prozess später kommen soll, wird entsprechend einer Vorrangplanung verwaltet. Das bedeutet, auf der Warteliste werden die Prozesse entsprechend ihrer ↑Ankunftszeit geführt, wohingegen auf der Bereitliste die Prozesse nach ihrer Priorität aufgestellt sind. Der ganz oben auf der Warteliste stehende Prozess hat die in dem Moment jüngste Ankunftszeit, muss aber in Bezug auf die Vorrangplanung nicht die höchste Priorität besitzen. Dennoch wird er zuerst von der Warteliste genommen (FCFS) und auf die Bereitliste gesetzt, obwohl ihm möglicherweise ein wartender Prozess höherer Priorität folgt. Beispielsweise ruft jeder ↑Semaphor, dessen Wartelistenstrategie (Ankunftszeit) im Widerspruch zur Bereitlistenstrategie (Priorität) steht, dieses Problem hervor.
Mögliche Folge dieses Fehlverhaltens ist ↑Prioritätsumkehr, nämlich dass ein Prozess niedrigerer Priorität den Prozessor nutzt, obwohl ein Prozess höherer Priorität eigentlich in den Vorzug der Prozessorzuteilung hätte kommen müssen. Diese Auswirkung zeigt sich jedoch erst im Moment des ↑Prozesswechsels, wenn dem fälschlicherweise ausgewählten Prozess der Prozessor zugeteilt wird. Für Prozesse, die in ↑Echtzeit stattfinden müssen, ist die Verletzung ihrer Priorität ein ↑Defekt, der schwerwiegende Folgen nach sich ziehen kann. Für einen Prozess anderer Art ergibt sich daraus vielleicht nur eine Anomalie, die kaum Aufsehen erregt.

Prioritätszuweisung (en.) ↑*priority assignment* Der Vorgang, um einer genau spezifizierten ↑Aufgabe eine bestimmte ↑Priorität für die spätere Bearbeitung durch einen ↑Prozess zu geben. Dabei ist die betrachtete Aufgabe im Regelfall nur eine von vielen verschiedenen Aufgaben (*task set*). Jede dieser Aufgaben erhält eine passende Priorität, um die ↑Dringlichkeit ihrer Bearbeitung im Vergleich zu den anderen Aufgaben zum Ausdruck zu bringen. Daraus resultiert schließlich ein ↑Ablaufplan und folglich eine wohl definierte Bearbeitungsreihenfolge der Aufgaben. Jedoch geschieht die Ablaufplanerstellung entkoppelt von der Aufgabenbearbeitung (↑*off-line scheduling*). Erst letztere impliziert die Prozesse, die plangemäß im ↑Rechensystem stattfinden müssen (↑*fixed-priority scheduling*). Typische Beispiele für solch eine Vergabe von Prioritäten an Aufgaben beziehungsweise Prozessen sind ↑RM und ↑DM.

priority (dt.) ↑Priorität, Rangfolge, ↑Dringlichkeit.

priority assignment (dt.) ↑Prioritätszuweisung.

priority ceiling (dt.) ↑Prioritätsobergrenze.

priority control (dt.) ↑Vorrangsteuerung.

priority inheritance (dt.) ↑Prioritätsvererbung.

priority inversion (dt.) ↑Prioritätsumkehr.

priority queue (dt.) ↑Vorrangwarteschlange.

priority scheduling (dt.) ↑Vorrangplanung.

priority violation (dt.) ↑Prioritätsverletzung.

private semaphore (dt.) ↑privater Semaphor.

privater Semaphor (en.) ↑*private semaphore*. Bezeichnung für einen ↑Semaphor, bei dem ↑P nur für denjenigen ↑Prozess möglich ist, der Eigentümer dieses Semaphors ist. Demgegenüber ist ↑V jedem anderen Prozess möglich. Originär ein ↑allgemeiner Semaphor mit Wertebereich $[-1, 1]$, implizit vorbelegt mit dem Wert 0. Alternativ kann auch ein ↑binärer Semaphor die Basis bilden, mit `false` als Initialwert.
Hauptzweck dieses Semaphors besteht in der ↑Synchronisation eines Prozesses auf ein ↑Ereignis, das den weiteren Verlauf des Prozesses bestimmt. Das Ereignis kann von einem beliebigen Prozess, auch ihm selbst, signalisiert werden, oft als Ergebnis der bei einer Berechnung erreichten Zustandsänderung. Varianten erlauben die Speicherung von mehr als einem Ereignis und umfassen demzufolge den Wertebereich $[-1, n]$, mit $n \geq 1$.

Exkurs Dem Prinzip nach funktioniert dieser Semaphor wie ein Auffangregister, er entspricht einem „Software-↑*latch*“ und kann damit Signale für den ihn besitzenden Prozess aufnehmen und zwischenspeichern. Der Mechanismus ist sehr einfach, ein ↑allgemeiner Semaphor bildet dafür eine geeignete Grundlage. Die Entgegennahme eines aufgefangenen Signals lässt sich wie folgt umsetzen:

```
void latch() {
    down(&being(ONESELF)->shut);            /* acquire (private) resource */
}
```

Die Lösung begreift den verwendeten zählenden Semaphor als Teil des Wesens (`being`) eines Prozesses, das heißt, der Semaphor ist Bestandteil eines ↑Prozesskontrollblocks und damit nur implizit (indirekt) adressierbar durch den ↑Prozesszeiger des ↑Prozessors des Prozesses, der ein aufgefangenes Signal abrufen möchte. Jeder Prozess besitzt seinen eigenen Prozesskontrollblock, dessen ↑Adresse damit ein eindeutiger Wert für den Prozesszeiger ist. Da es einem Prozes immer nur möglich ist, den Zählerwert des ihn angehenden (d.h., öffentlich nicht zugänglichen) Semaphors zu verringern und zudem bei voriger null blockieren wird (`latch`), kann die Differenz nicht kleiner als -1 sein.
Indem nun der Prozess den Wert des nur ihm zugänglichen Semaphors abwärts zählt (`down`), ruft er ein gespeichertes Signal ab, konsumiert es. Die dazu korrespondierende Operation, um ein Signal zu speichern, es zu produzieren, zeigt naschfolgende Skizze:

```
void serve(pid_t name) {
    process_t *peer = being(name);                /* grab process pointer */
    if (peer->name == name)                        /* valid process name? */
        up(&peer->shut);                           /* yes, signal release */
}
```

Hier wird der Prozesszeiger nicht implizit, sondern explizit bestimmt anhand der ↑Prozessidentifikation des Prozesses, der mit einem Signal bedient werden soll (`serve`). Nur wenn der so verfügbar gemachte Prozesskontrollblock auch tatsächlich den zu signalisierenden Prozess

modelliert, also die angegebene Prozessidentifikation enthält, wird der Wert des in dieser prozesslokalen Datenstruktur enthaltenen Semaphors aufwärts gezählt (`up`). Im Falle einer falschen Prozessidentifikation geht das Signal verloren.

privileged instruction (dt.) ↑privilegierter Befehl.

privileged mode siehe ↑*system mode*.

privilegierter Befehl (en.) ↑*privileged instruction*. ↑Maschinenbefehl, dessen direkte Ausführung durch eine (reale/virtuelle) Maschine nur erlaubt ist, wenn die ↑CPU im privilegierten ↑Arbeitsmodus tätig ist. Beispiele dafür sind Befehle zum Setzen oder Aufheben einer totalen ↑Unterbrechungssperre, Wechseln der ↑Unterbrechungsprioritätsebene, Zugriff auf ↑Unterbrechungsvektoren, Auslesen oder Beschreiben der ↑Geräteregister und Programmieren einer ↑MMU, ↑MPU oder eines ↑PIC. Für gewöhnlich wird ein ↑Prozess, der derartige Befehle im unprivilegierten Arbeitsmodus der CPU absetzt, eine ↑Ausnahme herbeiführen, von der CPU abgefangen (↑*trap*) und daraufhin vom ↑Betriebssystem abgebrochen.

probabilistic scheduling (dt.) ↑probabilistische Planung.

probabilistische Planung (en.) ↑*probabilistic scheduling*. Modell der ↑Ablaufplanung unter Berücksichtigung von Unsicherheiten oder Wahrscheinlichkeiten zu einer oder mehrerer ↑Prozessgrößen. Im Gegensatz zu ↑deterministische Planung geschehen die ↑Prozesse zufällig, zu unvorhersehbaren Zeitpunkten und oft auch für unvorhersehbare Zeitspannen. Die Prozessreihenfolge ist nicht durch Vorbedingungen im Voraus festgelegt, sondern entsteht erst zur ↑Laufzeit und als Folge von ↑Ereignissen.
Charakteristisches Merkmal ist die ↑mitlaufende Planung, da ↑Vorwissen zur Bildung und Aktualisierung der Prozessreihenfolge nicht zur Verfügung steht und die benötigten Informationen nur durch Beobachtung der Prozesse selbst gesammelt werden können. Grundlage bildet somit *dynamisches Wissen* über den gegenwärtigen Systemzustand, von dem ausgehend Entscheidungen für zukünftige Prozesse getroffen werden. Die mit diesen Entscheidungen einhergehende Optimierung in Bezug auf das ↑Einplanungskriterium für einen Prozess ist nur näherungsweise möglich. Die entstehende Prozessreihenfolge reflektiert dieses Wissen entsprechend eines oder mehrerer dieser Kriterien und wird dem angestrebten Optimierungsziel wahrscheinlich nahe kommen. Typische Beispiele sind ↑SPN, ↑HRRN und ↑SRTF.

process (dt.) Ablauf, Arbeitsablauf/-gang, ↑Prozess, Verfahren, Vorgang.

process address space (dt.) ↑Prozessadressraum.

process blockade (dt.) ↑Prozessblockade.

process control (dt.) ↑Prozesssteuerung, Prozessverarbeitung.

process control block (dt.) ↑Prozesskontrollblock.

process data (dt.) ↑Prozessdaten.

process data processing (dt.) ↑Prozessdatenverarbeitung.

process descriptor siehe (en.) ↑*process control block*.

process domain (dt.) ↑Prozessdomäne.

process factor (dt.) ↑Prozessgröße.

process favoring (dt.) ↑Prozessbegünstigung.

process identifaction (dt.) ↑Prozessidentifikation.

process incarnation (dt.) ↑Prozessinkarnation.

process instance (dt.) ↑Prozessexemplar.

process locality (dt.) ↑Prozesslokalität.

process lock (dt.) ↑Prozesssperre.

process obligation (dt.) ↑Prozessverpflichtung.

process pointer (dt.) ↑Prozesszeiger.

process scheduling (dt.) ↑Prozesseinplanung.

process switch (dt.) ↑Prozesswechsel.

process table (dt.) ↑Prozesstabelle.

process thrashing (dt.) ↑Flattern einer ↑Arbeitsmenge (↑*virtual memory*).

process-based operating system (dt.) ↑prozessbasiertes Betriebssystem.

processing time (dt.) ↑Bearbeitungszeit.

processor (dt.) ↑Prozessor, gemeinhin auch ↑CPU.

processor clock (dt.) ↑Prozessortakt.

processor consistency (dt.) ↑Prozessorkonsistenz.

processor cycle (dt.) ↑Prozessorzyklus.

processor identifaction (dt.) ↑Prozessoridentifikation.

processor register (dt.) ↑Prozessorregister.

processor status (dt.) ↑Prozessorstatus.

processor status word (dt.) ↑Prozessorstatuswort.

processor utilisation (dt.) ↑Prozessorauslastung.

producer-consumer problem (dt.) ↑Erzeuger-Verbraucher-Problem.

Produktionsbetrieb Bezeichnung für den Betrieb eines ↑Rechensystems, um Leistungsdaten zu produzieren. Neben einer Funktionsprüfung geschieht dabei auch die Bestimmung relevanter Systemparameter (d.h., nichtfunktionaler Merkmale) in Abhängigkeit von einem vorgegebenen *Anwendungsprofil*. Je nach ↑Betriebsart des Rechensystems finden diese Parameter Berücksichtigung als ↑Einplanungskriterium, hier insbesondere ↑Antwortzeit, ↑Durchlaufzeit, ↑Durchsatz und ↑Prozessorauslastung.

Profil (en.) ↑*profile*. Gesamtheit von Eigenschaften, die unverwechselbar typisch für einen bestimmten ↑Programmablauf sind (in Anlehnung an den Duden). Das Ergebnis einer ↑Profilbildung unter Berücksichtigung einer bestimmen ↑Arbeitslast. Ein ganz bestimmtes Erscheinungsbild, das spezifische *nichtfunktionale Eigenschaften* des Programmablaufs zum Ausdruck bringt. Beispielsweise die ↑Speichergrundfläche, die ↑Ausführungszeit insgesamt oder nur bestimmter ↑Aufgaben, der bei Ausführung des betreffenden ↑Programms anfallende Energiebedarf, ein Muster von bekannten und prägenden ↑Ereignissen, die Frequenz von ↑Systemaufrufen, die Größe beziehungsweise der Umfang von ↑Ein-/Ausgabe oder gar die Anzahl hervorgerufener ↑paralleler Prozesse.
Diese Eigenschaften geben Aufschluss über ein bestimmtes Verhalten, das ein Programmablauf hervorruft: beispielsweise speicherhungrig, berechnungsintensiv oder energiearm zu sein. Jedoch kann bei Beobachtung eines solchen Verhaltens für gewöhnlich nicht auf das Programm geschlossen werden, dessen Programmablauf eben dieses Verhalten verursacht. Verschiedene Programme zur Ausführung gebracht bewirken verschiedene Programmabläufe, deren charakteristische Eigenschaften aber durchaus gleich sein können.

Profilbildung (en.) ↑*profiling*. Eine für bestimmte Zwecke nutzbare Darstellung des Gesamtbildes eines ↑Programmablaufs (in Anlehnung an den Duden), die Erstellung seines charakteristischen, als ↑Profil bezeichneten, Erscheinungsbildes geprägt durch bestimmte *nichtfunktionale Eigenschaften*. Für gewöhnlich kommen hierzu spezielle Werkzeuge (↑*profiler*) zur ↑Instrumentierung zum Einsatz, und zwar um das ↑Programm, das den zu untersuchenden Programmablauf definiert, gezielt zu ergänzen um Anweisungen zur Aufzeichnung der ↑Daten, die zur späteren Erstellung des gewünschten Profilbildes benötigt werden. Nicht zu verwechseln mit ↑Fehlerbereinigung, mit der unerwünschte *funktionale Eigenschaften* eines Programmablaufs beseitigt werden.

profile (dt.) ↑Profil.

profiler (dt.) ↑Profilersteller.

Profilersteller (en.) ↑*profiler*. Bezeichnung eines speziellen, der ↑Profilbildung eines ↑Programmablaufs dienenden ↑Programms. Typische Beispiele für solche Programme sind `perf(1)` für ↑Linux, `sample(1)` und `instruments(1)` für ↑macOS oder `VSPerfCmd` (als Bestandteil von *Visual Studio*) für ↑Windows.

profiling (dt.) ↑Profilbildung.

program (dt.) ↑Programm.

program counter (dt.) ↑Befehlszähler; Abkürzung ↑PC.

program library (dt.) ↑Programmbibliothek.

program relocation (dt.) ↑Programmverlagerung.

program segment (dt.) ↑Programmsegment.

program status word (dt.) ↑Programmstatuswort.

Programm (en.) ↑*program*. Festlegung einer Folge von Anweisungen für einen ↑Prozessor, nach der die zur Bearbeitung einer (durch einen Algorithmus wohldefinierten) Handlungsvorschrift erforderlichen ↑Aktionen stattfinden sollen; Konkretisierung eines Algorithmus.
Eine solche Folge beschreibt einen Vorgang, einen ↑Prozess, der sich über eine gewisse Zeit erstrecken und bei dem ein bestimmtes Berechnungsergebnis entstehen soll. Sie ist von statischer Natur, die den Vorgang (dynamisch) erst dann zur Entfaltung bringt, wenn sie durch den für sie bestimmten Prozessor als ↑Programmablauf zur Ausführung kommt.
Für gewöhnlich ist mit der Anweisungsfolge ein *starrer Rechenplan* für den Prozessor definiert, das heißt, sie selbst enthält keine Anweisungen, die diesen Rechenplan, also sich selbst, bei der Ausführung verändern könnten. Im Gegensatz dazu bezeichnet ein *freier Rechenplan* eine zwar immer noch statische Anweisungsfolge, bei deren Ausführung aber bestimmte Anweisungen in ihr selbst modifiziert werden (↑*self-modifying code*).

Programmablauf (en.) ↑*computer operation*. Ein auf einem ↑Rechner stattfindender ↑Prozess, der allein (↑*uniprogramming*) oder in Konkurrenz mit anderen (↑*multiprogramming*) strikt nacheinander (↑*run to completion*) beziehungsweise simultan (↑*multiprocessing*) den ↑Prozessor zum eigenen Fortschritt nutzt. Ist der Prozess — horizontal (auf derselben ↑Abstraktionsebene) wie auch vertikal (über mehrere Abstraktionsebenen) — durch ein ↑sequentielles Programm bestimmt, finden seine ↑Aktionen total geordnet statt: er repräsentiert einen sequentiellen ↑Kontrollfluss. Liegt dem Prozess jedoch ein ↑nichtsequentielles Programm zugrunde, so sind seine Aktionen bestenfalls partiell geordnet. In dem Fall ist ein ↑nichtsequentieller Prozess möglich, der mehrere parallele Kontrollflüsse hervorbringen kann.

programmable interval timer (dt.) programmierbarer ↑Intervallzeitgeber.

Programmbibliothek (en.) ↑*program library*. Gesamtheit mehrerer häufig verwendeter Softwarebestände (↑Modul, ↑Unterprogramm, ↑Exemplar eines Datentyps), die in wenigstens einem ↑Objektmodul enthalten sind. Typischerweise sind jedoch mehrere Objektmodule in der ↑Bibliothek zusammengefasst, wie beispielsweise im Falle der ↑libc. In einer solchen Bibliothek ist jedem einzelnen Softwarebestand ein ↑Name (↑Symbol) eindeutig zugeordnet. Die Bibliothek selbst ist in Form einer ↑Datei in der ↑Ablage vorrätig.

programmed binding (dt.) ↑programmiertes Binden.

programmed input/output (dt.) ↑programmierte Ein-/Ausgabe; Abkürzung ↑PIO.

Programmiermodell (en.) ↑*programming model*. Bezeichnung für den in einem ↑Programm direkt sicht-, nutz- oder programmierbaren ↑Registersatz von einem ↑Prozessor einschließlich dessen *Ausführungsmodell*. Letzteres spezifiziert, in welcher Art und Weise die durch einen ↑Maschinenbefehl ausgelösten Operationen stattfinden, das heißt, ob sie sequentiell, nichtsequentiell, außer der Reihe (*out of order*), unteilbar (↑*atomic operation*) ausgeführt werden und inwiefern ↑Speicherkonsistenz oder ↑Speicherkohärenz dabei gewahrt ist.

programmierte Ein-/Ausgabe (en.) ↑*programmed input/output*. Methode des Transfers von ↑Daten zwischen ↑Peripherie und ↑Hauptspeicher unter Hilfestellung durch die ↑CPU. Im Gegensatz zu ↑DMA führt die CPU ein ↑Programm aus, um ein ↑Byte nach dem anderen zu transferieren. Die damit verbundene Ein-/Ausgabe in Bezug auf einen bestimmten ↑Speicherbereich läuft voll und ganz unter Kontrolle der CPU ab. Folglich kann die CPU in der Zeit keine anderen Berechnungen durchführen: Ein-/Ausgabe und Berechnungen überlappen sich nicht.

programmiertes Binden (en.) ↑*programmed binding*. Geplantes ↑Binden bei Bedarf, und zwar explizit kodiert im ↑Maschinenprogramm daselbst, ausgelöst durch einen ↑Systemaufruf (↑UNIX: `dlopen(3)`; ↑Windows: `LoadLibrary`). Das Binden der ↑Text- oder ↑Datenbestände geschieht dynamisch, zur ↑Laufzeit des Maschinenprogramms. Anders als ↑dynamisches Binden, das in ursprünglicher Form durch ↑Teilinterpretation der Zugriffe eines ↑Prozesses auf diese Bestände ausgelöst wird, ist die programmierte Variante (in funktionaler Hinsicht) allerdings intransparent für diesen Prozess.

programming model (dt.) ↑Programmiermodell.

Programmsegment (en.) ↑*program segment*. Gleichartiger Abschnitt eines gegliederten, nach einem bestimmten System *logisch* aufgeteilten ↑Maschinenprogramms. Ein solcher Abschnitt enthält ↑Texte oder ↑Daten oder bildet eine, gemeinhin für Daten reservierte, Aussparung (↑BSS) in dem vom Maschinenprogramm belegten ↑Speicherbereich. Entsprechend wird zwischen zwei Kategorien unterschieden: dem ↑Textsegment und dem ↑Datensegment. Jedes davon kann auf ein oder mehrere voneinander getrennte ↑Adressraumsegmente eines zweidimensionalen ↑Adressraums abgebildet sein (↑*segmentation*) oder beide belegen verschiedene ↑Adressbereiche im gemeinsamen (eindimensionalen) Adressraum.

Programmstatuswort Abkürzung ↑PSW, Jargon (IBM) für die Kombination von ↑Prozessorstatuswort und ↑Befehlszähler.

Programmverlagerung (en.) ↑*program relocation*. Der bei ↑Verlagerung eines bereits mit einer ↑Ladeadresse versehenen ↑Maschinenprogramms erforderliche Vorgang, ↑absolute Adressen um eine bestimmte ↑Ladedistanz zu verschieben.

progress guarantee (dt.) ↑Fortschrittsgarantie.

PROM Abkürzung für (en.) *programmable ROM*, (dt.) programmierbarer ↑ROM.

proportional backoff (dt.) verhältnisgleich zurückweichen (durch Anpassung der ↑Ruhezeit; in Anlehnung an den Duden). Ein auf *Rückkopplung* basierendes Verfahren, das die Rate bestimmter ↑Aktionen eines ↑Prozesses in Bezug auf ein ↑gemeinsames Betriebsmittel in Abhängigkeit von der Anzahl der im Moment des Zugriffs darum konkurrierenden Prozesse verringert (↑*backoff*). Typisch für die ↑Ticketsperre.

Proportionalität Verhältnismäßigkeit, Angemessenheit (Duden). In Bezug auf ein ↑Betriebssystem ist damit insbesondere das Verhältnis zwischen der für einen Anwendungsfall akzeptablen Leistung einerseits und der technisch erreichbaren effektiven Leistung andererseits gemeint. Nicht immer stößt das technisch Machbare auf anstandslose Akzeptanz beim Menschen. Die Konsequenz daraus kann sein, ein ↑Rechensystem für bestimmte Benutzer/innen gezielt so so betreiben, wie sie es bislang gewohnt sind, obwohl dadurch die Leistungsfähigkeit des Systems bei weitem nicht ausgeschöpft wird. Typische Beispiele sind etwa die bewusste Verlängerung von ↑Antwortzeiten oder ↑Durchlaufzeiten für die von den betreffenden Personen ausgelösten Aufträge an das Betriebssystem. Allgemein besagt dieser Aspekt, ein ↑Einplanungskriterium gerade noch gut genug für die jeweils gegebene ↑Anwendung zu erfüllen und damit ein zufriedenstellendes Maß an Leistungsfähigkeit zu gewährleisten.

protection (dt.) ↑Schutz.

protection domain (dt.) ↑Schutzdomäne.

protection error (dt.) ↑Schutzfehler.

protection fault (dt.) ↑Schutzverletzung.

protection ring (dt.) ↑Schutzring.

protection system (dt.) ↑Schutzsystem.

Prozedurfernaufruf (en.) ↑*remote procedure call*. Aufruf, der den aktuellen ↑Handlungsstrang verlässt und ein ↑Unterprogramm in einem gegebenenfalls anderen, dem ↑Speicherschutz unterliegenden ↑Adressraum zur Ausführung bringt. Ursprünglich ein Konzept ausschließlich für kooperiende Programme, die verteilt auf einem ↑Rechnernetz ablaufen. In abgewandelter Form jedoch auch anwendbar zur lokalen Kommunikation zwischen Handlungssträngen desselben oder verschiedener Adressräume desselben Rechners.

Prozess (en.) ↑*process*, P_i. Ein ↑Programm in Ausführung — wobei für gewöhnlich keine Annahme darüber getroffen wird, in welcher Art und Weise diese Ausführung geschieht und in welcher Form sie erscheint (im Gegensatz zum ↑Prozessexemplar, s.w.u.). Der ursprünglichen Begrifflichkeit nach ein Ort der Kontrolle innerhalb einer Befehlssequenz, das heißt, eine abstrakte ↑Entität, die die Anweisungen eines Programms durchläuft, wenn das Programm von einem ↑Prozessor ausgeführt wird. Damit auch ein sich über eine gewisse Zeit erstreckender und durch die Anweisungen eines Programms beschriebener Vorgang, bei dem ein bestimmtes Berechnungsergebnis entstehen soll. Ein solcher Vorgang wird erst bewirkt durch einen ↑Programmablauf, eine ↑Aktionsfolge, er ist inhaltlich festgelegt durch das Programm einerseits und den gemeinhin erst zur ↑Laufzeit bekannten Eingabedaten andererseits.
Gelegentlich wird solch ein Vorgang auch als *intern* oder *extern* stattfindend qualifiziert. ↑Bezugspunkt dabei ist das ↑Rechensystem des Prozessors, der das betreffende Programm ausführt und dadurch den Vorgang stattfinden lässt. Die internen Vorgänge finden innerhalb desselben Rechensystems statt, wohingegen externe Vorgänge außerhalb dieses Rechensystems geschehen. Gleichwohl kann zwischen diesen eine Interaktion stattfinden, beispielsweise indem ein interner Vorgang einen externen Vorgang veranlasst oder umgekehrt (↑Ein-/Ausgabe) und ein externer Vorgang (etwa mittels einer ↑Unterbrechungsanforderung) auf einen internen Vorgang einwirkt.

Hinweis Abweichend von der ursprünglichen Definition einer abstrakten Entität begreift

die ↑Informatikfolklore mit diesem Terminus gemeinhin auch einen in sich abgeschlossenen, geschützten ↑Prozessadressraum: nämlich ein unter ↑Adressraumisolation stattfindender Programmablauf. Diese Sicht, die eine bestimmte Art, Weise und Form der Ausführung eines Programms vorgibt, ist mit ↑Multics etabliert worden, sie repräsentiert jedoch nur eine von mehreren möglichen Verkörperungen eines Programmablaufs in Form einer sogenannten ↑Prozessinkarnation und ruft damit Mehrdeutigkeit wie auch Missverständnisse hervor. Als Folge dieser Gleichsetzung und dem Bedürfnis nach einer Differenzierung in Bezug auf die konkrete Implementierung haben sich alternative technische Auslegungen wie ↑Faden, ↑Faser und ↑Fäserchen ergeben, womit allesamt verschiedene Programmabläufe innerhalb desselben isolierten Adressraums zum Ausdruck gebracht werden (vgl. auch S. 244). Allen Auslegungen gemeinsam (vor Multics) war und (nach Multics) ist, einen Ort der Kontrolle innerhalb der bei Ausführung eines Programms entstehenden Befehlssequenz zu verkörpern, diesen als *autonomen* ↑Kontrollfluss zu repräsentieren.
Um den Bezug zu räumlicher Isolation und ↑Schutz explizit zu machen, ist daher auch eine Einordnung in *Gewichtsklassen* gängig. Existiert etwa ein solcher Kontrollfluss immer nur allein innerhalb eines geschützten Adressraums, wird er nämlich als ↑schwergewichtiger Prozess verstanden: dies entspricht der mit Multics eingeführten und über viele Jahre durch ↑UNIX einschließlich Doppelgänger (*look-alike*) propagierten Begrifflichkeit. Demgegenüber steht ein ↑leichtgewichtiger Prozess für einen Kontrollfluss, der sich mit anderen Kontrollflüssen seiner Art denselben geschützten Adressraum teilt. Dabei werden die ↑Exemplare dieser beiden Klassen von Prozessinkarnationen durch das Betriebssystem verwaltet: eingeführt mit ↑Thoth, wo sich eine Gruppe (↑*team*) von Prozessexemplaren denselben geschützten Adressraum teilt. Im Vergleich dazu ist ein ↑federgewichtiger Prozess, der als Kontrollfluss ebenfalls gemeinsam mit anderen seiner Art im selben geschützten Adressraum residiert, dem Betriebssystem unbekannt. Mit dieser Klassifikation werden grob die beim Wechsel zwischen den einzelnen Kontrollflüssen zu erwartenden ↑Gemeinkosten eingestuft: je höher diese liegen, je schwerer also ein Kontrollfluss wiegt und er damit auch schwerfälliger in seinen Bewegungen sein wird, desto seltener sollte der Kontrollflusswechsel erfolgen. Gemeinhin steigen diese Gemeinkosten mit zunehmender Striktheit der Adressraumisolation.
So wird deutlich, dass die (durch Multics unbeabsichtigt begründete) „folkloristische“ Sicht von einem Programm in Ausführung und dem damit zu verbindendem Begriff problematisch ist. Nicht die Art und Weise ist entscheidend, wie ein Programmablauf in seinem Adressraum wirkt, dass heißt, allein oder mit anderen gemeinsam darin stattfindet und dem Betriebssystem bekannt oder unbekannt ist. Entscheidend ist die durch den Programmablauf gegebene Dynamik, nämlich dass sich entsprechend Handlungsvorschrift und Datenzustand des Programms der Ort der Kontrolle innerhalb einer Befehlssequenz fortlaufend ändert.

Prozessadressraum (en.) ↑*process address space*. Bezeichnung für den durch einen ↑Prozess definierten ↑Adressraum. Wenn die ↑Betriebsart es zulässt oder gar erfordert, ist mit diesem Adressraum gleichfalls die Durchsetzung der Vereinzelung oder Abkapselung eines Prozesses innerhalb einer Gruppe von Prozessen verbunden (↑*address-space isolation*).

prozessbasiertes Betriebssystem (en.) ↑*process-based operating system*. Bezeichnung für ein ↑Betriebssystem, dessen grundlegendes Konzept zur Repräsentation einer autarken ↑Aktion oder ↑Aktionsfolge der ↑Prozess ist: jeder mögliche ↑Handlungsstrang in dem System benötigt eine ↑Prozessinkarnation, um stattzufinden. Die Besonderheit eines solchen Betriebssystems besteht darin, jeden dieser Stränge mit einem eigenen ↑Laufzeitstapel zu versehen. Im Gegensatz dazu steht ein ↑ereignisbasiertes Betriebssystem, das alle Handlungsstränge auf ein und demselben Laufzeitstapel ablaufen lässt.
Im Betriebssystem jedem Handlungsstrang einen eigenen Laufzeitstapel zuzuordnen, trägt wesentlich dazu bei, sowohl die ↑Einplanungslatenz als auch die ↑Einlastungslatenz für einen Prozess reduzieren zu können. Einer engen räumlichen und zeitlichen Kopplung von ↑Einplanung und ↑Einlastung steht nichts entgegen, zumindest soweit es die ↑Architektur des Betriebssystems betrifft. Kommt ein Prozess nach erfolgter Einplanung gegebenenfalls

trotzdem nicht sofort zum Zuge, liegt dies in erster Linie an der Art und Weise der ↑Ablaufplanung: wenn im Betriebssystem ↑präemptive Planung stattfindet, ist damit die grundlegende Voraussetzung zur zeitnahen Einlastung eintreffender Prozesse gegeben. In zweiter Linie könnten nur noch Maßnahmen zur ↑Synchronisation einen möglichen ↑Prozesswechsel innerhalb des Betriebssystems verhindern beziehungsweise verzögern.

Prozessbegünstigung (en.) ↑*process favoring*. Bevorteilung eines ↑Prozesses aus einer Gruppe von Prozessen, die dasselbe ↑Ereignis erwarten. Ein solches Ereignis besteht beispielsweise in der ↑Einlastung des ↑Prozessors mit einem sich im ↑Prozesszustand bereit befindlichen Prozess. Ein anderes typisches Beispiel ist die Zuteilung eines beliebig anderen ↑Betriebsmittels an einen Prozess, der sich im Zustand blockiert befindet. Gibt es zu einem Zeitpunkt mehr Prozesse als für sie verfügbare Betriebsmittel, kommen die Prozesse, die in dem Moment nicht bedient werden können, auf eine ↑Warteliste des von ihnen benötigten und beanspruchten Betriebsmittels. Für den Prozessor (auch ↑Rechenkern) handelt es sich bei solch einer Warteliste um die sogenannte ↑Bereitliste, da alle Prozesse darauf bereit dazu sind, fortgesetzt werden zu können und dazu nur noch einen Prozessor benötigen. Anstatt nun die Prozesse in Ankunftsreihenfolge (↑FCFS) zu bedienen, wird aufgrund bestimmter Kriterien ein Prozess dem anderen gegenüber bevorteilt von der Warteliste gestrichen.
Im Falle der Prozessorzuteilung ist die Bevorteilung von Prozessen für gewöhnlich ein optionales Merkmal der ↑Einplanung, die dazu analytische oder konstruktive Maßnahmen in den Vordergrund stellt. Bei analytischer Herangehensweise ist beispielsweise der ↑Rechenstoß oder die ↑Zwischenankunftszeit eines jedes Prozesses Gegenstand der Untersuchung, so dass etwa der Prozess mit der jeweils kürzesten Stoßlänge (↑SPN, ↑HRRN, ↑SRTF), höchsten Ankunftsrate (↑RM) oder mit den kürzesten relativen ↑Terminen (↑DM) bevorzugt eingelastet wird (↑*priority scheduling*). Demgegenüber greift eine konstruktive Herangehensweise auf strukturelle Mittel zurück, um einem Prozess Vorrang gegenüber anderen Prozessen zu geben. Beispielsweise eine Vorzugsliste, um interaktive Prozesse vorrangig einzulasten (↑VRR) oder eine Hierarchie von Bereitlisten, in der Prozesse zunächst oben (hohe Priorität) einsteigen, dann aber mit länger werdenden Rechenstößen schrittweise nach unten (niedrigere Priorität) durchgereicht werden (↑FB, ↑MLFQ).
Im Vergleich dazu sind Wartelisten für die Zuteilung anderer ↑Ressourcen als Prozessoren nicht selten als einfache ↑Schlangen repräsentiert. Damit werden die in den Schlangen verzeichneten Prozessanfragen dann aber auch in Ankunftsreihenfolge bedient (↑FIFO). Diese Vorgehensweise ist problematisch, wenn die Prozessorzuteilung eine andere Bedienungsart verfolgt. Dabei muss die Abweichung in der Reihungsdisziplin längst nicht nur zu Lasten der ↑Performanz des ↑Rechensystems gehen, kritischer kann eine dadurch mögliche ↑Prioritätsverletzung sein, wenn nämlich die Prozessorzuteilung nach den Prozessen zugeordneten ↑Prioritäten geschieht. Um in solchen Fällen Fehlverhalten bei der Auswahl des Prozesses vorzubeugen, dessen Anfrage als nächste bedient werden soll, ist eine mit der ↑Planung der Prozessorzuteilung konform gehende Disziplin zur Reihung der Einträge in der Schlange zu verwenden (↑*scheduling queue*).

Prozessblockade (en.) ↑*process blockade*. Vorübergehender Ausfall bestimmter ↑Prozesse (in Anlehnung an den Duden) bedingt durch das Fehlen oder Nichtstattfinden wenigstens eines ↑Ereignisses, von dessen Eintritt das weitere Vorankommen eines jeweiligen Prozesses abhängt. Der betreffende Prozess befindet sich im ↑Prozesszustand blockiert. Zudem ist in seinem ↑Prozesskontrollblock das Ereignis benannt, das zu seiner Deblockierung führt.
Technisch für gewöhnlich bewirkt durch eine ↑Prozesssperre, die bei passender Gelegenheit durch einen anderen (nicht gesperrten, ggf. auch externen) Prozess wieder aufzuheben ist. Gibt es diesen anderen Prozess jedoch nicht oder ist dieser anhaltend verhindert, das Ereignis zur Deblockierung herbeizuführen und damit die Sperre aufzuheben, wird die Blockade aus dem Kreis der betroffenen, zusammenhängenden Prozesse nicht durchbrochen werden können. Ein solches Szenario bedeutet die ↑Verklemmung des oder der blockierten Prozesse.

Exkurs Ein Prozess, der sich sich schlafen legt (`sleep`), damit blockiert, um zu gegebener

Zeit von einem anderen Prozess aufgeweckt zu werden (`wakeup`), sodann seine Tätigkeit fortsetzen kann, ist nicht ungewöhnlich. Ein solches Szenario ist typisch für die ↑Synchronisation ↑gleichzeitiger Prozesse. Schlafende Prozesse stehen auf einer ↑Warteliste:

```
typedef struct {
    virtual queue_t list;                /* regular or computed waitlist */
} waitlist_t;
```

Je nach Modell liegt die Warteliste als reguläre dynamische Datenstruktur (↑*queue*) vor oder sie wird bei Bedarf immer wieder durch Ablauf der ↑Prozesstabelle neu berechnet — sie ist in jedem Fall aber „virtuell“ (*virtual*) präsent. Der ↑Tabellenlauf ist eine Funktion des ↑Planers und würde damit grundsätzlich eine, im Auslegungsfall als ↑Schlange, mögliche ↑Prioritätsverletzung bei der Auswahl des aufzuweckenden Prozesses ausschließen. Andererseits kann die Neuberechnung im Mittel eine höhere ↑Einplanungslatenz mit sich bringen. Ist dies nicht tolerierbar und ↑Interferenz mit dem Planer vorzubeugen, sollte die Warteliste (a) als dynamische Datenstruktur vorliegen und (b) entsprechend der ↑Ablaufplanung verwaltet werden.
Um nun einen Prozess in Schlaf zu versetzen, fallen gemeinhin zwei Hauptschritte an:

```
void sleep(waitlist_t *bell) {
    allot(bell);                              /* label process, queue up */
    block();                        /* suspend process until event occurs */
}
```

Im ersten Schritt benennt der noch laufende Prozess selbst den Grund für seine anstehende Blockierung (`block`), er disponiert (`allot`) seinen Prozesskontrollblock für das zur Fortsetzung seiner Tätigkeit benötigte Ereignis (`bell`) und macht dazu einen entsprechenden Vermerk. Damit steht der Prozess auf der Warteliste (`list`), er ist auffindbar, sobald das entsprechende Ereignis eintritt und daraufhin von einem anderen Prozess, der dieses Ereignis hervorgerufen hat, der Weckruf (`wakeup`) geschieht. Hierzu wird das benannte Ereignis, das „Klingelzeichen“ (`bell`) zum Wachwerden, im Prozesskontrollblock des laufenden Prozesses eingetragen. Wenn mehrere Prozesse dasselbe Ereignis erwarten, gemeinsam schlafen, stehen sie alle auf einer Warteliste, die durch das Weckrufzeichen (`bell`) gekennzeichnet ist. Sollte diese Liste konkret als dynamische Datenstruktur ausgeprägt sein, bedeutet disponieren (`allot`) des Prozesses nicht nur, einen Prozesskontrollblock mit einer Marke zu versehen, sondern diesen auch in eine mit dieser Marke assoziierten ↑Schlange einzureihen.
Im zweiten Schritt (`block`) setzt sich der Prozess sodann für eine gewisse Zeit außer Funktion, er blockiert und erklärt sich dem Planer gegenüber bereit, freiwillig den ↑Prozessor abzugeben. Ob es in dem Moment wirklich zur Prozessorabgabe und einem damit verbundenen ↑Prozesswechsel kommt, hängt von zwei Bedingungen ab: erstens, seit vermerken (`allot`) des Ereignisses ist der Weckruf (`wakeup`) noch nicht geschehen; zweitens, der Planer hat den von der Warteliste (`list`) gestrichenen, aufgeweckten Prozess zwar auf die ↑Bereitliste gesetzt, aber einen anderen Prozess bevorzugt.
Die Operation zum Weckruf streicht einen Prozess von der mit dem Weckrufzeichen (`bell`) identifizierten Warteliste. Dazu wird geprüft (`check`), ob Prozesse auf der Warteliste verzeichnet sind und welcher davon gestrichen werden sollte. War die Warteliste nicht leer, wird der gelieferte Prozess für die Zuteilung des Prozessors bereitgestellt (`ready`):

```
void wakeup(waitlist_t *bell) {
    process_t *next = check(bell);        /* select process based on label */
    if (next)                           /* anyone waiting on labeled event? */
        ready(next);                   /* yes, provide process and schedule */
}
```

Für das Zusammenspiel dieser beiden Operationen (`sleep`, `wakeup`) ist das Disponieren (`allot`) des sich schlafen legenden Prozesses von besonderer Bedeutung. Erst mit diesem Schritt ist der betreffende Prozess als „schlafende ↑Entität“ verbucht, er kann damit aufgeweckt werden und (durch `ready`) wieder auf die Bereitliste kommen, und zwar *bevor* er

sich blockiert (`block`). Dies bedeutet aber auch, dass der betreffende Prozess möglicherweise überhaupt nicht den Prozessor abgibt, da die Bereitstellung ihn ganz oben auf die Bereitliste platziert und er bei der (im Zuge seiner Blockierung stattfindenden) Auswahl des ihn nachfolgenden Prozesses sich selbst als Nachfolger von dieser Liste wieder heruntergenommen haben kann. Entsprechende Berücksichtigung dieses Aspekts (bspw. S. 50) eröffnet einen Weg zur Vorbeugung der für gewöhnlich kritischen ↑Wettlaufsituation, nämlich den für einen sich schlafen legenden Prozess bestimmten Weckruf zu verpassen (↑*lost wake-up*).

Prozessdaten (en.) ↑*process data*. Informationen, analoge oder digitale ↑Daten, die mittels Sensoren einem bestimmten physikalisch-technischen Prozess entnommen und zur ↑Prozesssteuerung verwendet werden. Die Daten repräsentieren Messwerte unterschiedlichster Art, beispielsweise Temperatur, Druck, Füllstand, Spannung, Zerfall, Geschwindigkeit, Höhe, Position, aber auch Umsatz.

Prozessdatenverarbeitung (en.) ↑*process data processing*. ↑Steuerung oder ↑Regelung von physikalisch-technischen ↑Prozessen durch ein unter ↑Echtzeitbetrieb laufendes ↑Rechensystem (↑*real-time computing system*). Die in ↑Echtzeit zu verarbeitenden ↑Prozessdaten (↑*real-time data processing*) repräsentieren den aktuellen Zustand des physikalisch-technischen Prozesses. Als Folge der Verarbeitung dieser ↑Daten wird der physikalisch-technische Prozess in einen neuen Zustand überführt.

Prozessdomäne (en.) ↑*process domain*. Herrschaftsbereich von einem ↑Prozess, definiert durch das ihn bestimmende ↑Programm und repräsentiert durch seinen ↑Adressraum. Bei zusätzlichem ↑Speicherschutz (optional) ist es dem Prozess unmöglich, seine Domäne unkontrolliert zu verlassen und in die eines anderen Prozesses einzudringen.

Prozesseinplanung (en.) ↑*process scheduling*. Vorgang der ↑Ablaufplanung, die auf Grundlage einer ↑Prozessinkarnation operiert und (relativen) Zeitpunkt wie auch Reihenfolge der ↑Einlastung der ↑CPU damit festlegt.

Prozessexemplar (en.) ↑*process instance*. Ausfertigung eines ↑Prozesses in Abhängigkeit von der ↑Betriebsart, ein bestimmtes ↑Exemplar in Form einer ↑Prozessinkarnation. Ein ↑Programmablauf in dinghafter Ausprägung und mit wirklichen Eigenschaften bestimmt durch das ↑Betriebssystem. Verkörperung eines Prozesses, indem ein bestimmter ↑Laufzeitkontext, eine Menge (virtueller) ↑Speicher, gegebenenfalls auch eine eigene ↑Schutzdomäne und möglicherweise weitere (virtuelle) ↑Ressourcen vom Betriebssystem eingerichtet wurden. Anders als Prozess eine konkrete ↑Entität, die die Anweisungen eines ↑Programms durchlaufen kann, wenn das Programm von einem ↑Prozessor kontrolliert durch das Betriebssystem ausgeführt wird (vgl. auch S. 240).
Für ein bestimmtes algorithmisches Problem kann es mehr als ein ↑Programm zur Beschreibung der Lösung durch einen Prozess geben und jeder dieser Prozesse kann in verschiedener Gestalt in Form einer Prozessinkarnation verkörpert sein. Letztere unterscheiden sich dann in Art und Umfang der Ressourcen, die dem Prozess, damit er stattfinden kann, durch das Betriebssystem bereitgestellt werden müssen. Ein solches Exemplar könnte als ↑Fäserchen, ↑Faser oder ↑Faden gemeinsam mit anderen ihrer Art (die ganz andere Probleme lösen) im selben isolierten ↑Adressraum umgesetzt sein oder es besitzt einen eigenen isolierten ↑Prozessadressraum. Die für diese verschiedenen Ausprägungen von einem Betriebssystem dann bereitzustellenden Ressourcen und zu leistenden ↑Systemfunktionen sind in Art, Umfang und Auslegung dann schon sehr verschieden. Eine typische Ressource, die aber allen Ausprägungen von Prozessen gemein ist, ist der ↑Prozesskontrollblock — der allerdings für die unterschiedlichen Arten von Prozessverkörperung für gewöhnlich auch eine *artspezifische Datenstruktur* darstellt, durch einen eigenen ↑Datentypen modelliert ist.

Hinweis Gemeinhin kann der Unterschied zum Begriff Prozess vernachlässigt werden, beide sind landläufig von gleicher oder ähnlicher Bedeutung, sinnverwandt. Wenn jedoch eine spezielle Ausprägung eines Prozesses im Vordergrund steht, sollte auch der Begriff Prozess

qualifiziert Verwendung finden, indem die besonderen, ausschlaggebenden Merkmale, die der Prozess aufweist, angegeben werden. Prozess allein ist kontextabhängig, je nachdem ob seine abstrakte Entität (d.h., der Ort der Kontrolle in einer Befehlssequenz) oder seine konkrete Entität (d.h., ein solcher Ort mit ↑Adressraumisolation) gemeint ist.

Prozessgröße (en.) ↑*process factor*. Maß der Ausdehnung von einem ↑Prozess in Raum und Zeit, quantitative Eigenschaft oder technisches Merkmal einer ↑Prozessinkarnation. Als Raumgröße gilt typischerweise die ↑Speichergrundfläche, wohingegen eine Zeitgröße die ↑Auslösezeit, ↑Ankunftszeit, ↑Bereitzeit, ↑Startzeit, ↑Bedienzeit, ↑Bearbeitungszeit, ↑Ausführungszeit, ↑Wartezeit, ↑Endzeit oder eine ↑Frist meint.

Prozessidentifikation (en.) ↑*process identifaction*, Abkürzung ↑PID. Nummer (auch: ↑Name) von einem ↑Prozess, genauer gesagt einer ↑Prozessinkarnation. Für gewöhnlich eine als *elementarer* ↑*Datentyp* beschriebene Größe, die im ↑Betriebssystem dazu genutzt wird, prozessspezifische Datenstrukturen zu verwalten und zu identifizieren.

Exkurs Nach außen beschreibt dieser Datentyp zumeist eine einfache Zahlenfolge, etwa eine Ganzzahl aus einem Wertebereich, der durch die ↑Wortbreite vorgegebenen ist:

```
typedef long int pid_t;
```

Nach innen, für das Betriebssystem, bietet es sich gegebenenfalls an, jedem der Zahlenwerte eine bestimmte Struktur aufzuerlegen. Beispielsweise erweist es sich als hilfreich, aus der PID schnell den zugehörigen ↑Prozesskontrollblock lokalisieren zu können. Wenn diese im Betriebssystem sehr zentralen Datenstrukturen in einem Feld, der ↑Prozesstabelle, organisiert sind, wäre es sehr nützlich, in der PID eine *Kennziffer* als Feldindex zu kodieren. Für die Selektion eines Eintrags in der Prozesstabelle muss dann lediglich diese Kennziffer aus der PID extrahiert werden:

```
inline int ratio(pid_t name) { return name % NPROC; }
```

Dabei definiert `NPROC` die Anzahl der in der Prozesstabelle enthaltenen Prozesskontrollblöcke. Bei der Erzeugung einer Prozessinkarnation entnimmt das Betriebssystem der Prozesstabelle den nächsten freien PCB und erfährt damit zugleich die Kennziffer (den Feldindex) des betreffenden Prozesses. Da jedoch davon auszugehen ist, dass derselbe Tabelleneintrag über die Zeit mehrfach für verschiedene Prozesse genutzt wird, muss die jeweilige Kennziffer eindeutig gemacht werden. Dies kann einfach durch einen *Generationszähler* erreicht werden, der neben der Kennziffer die zweite Strukturkomponente der PID darstellt. Mit jeder Wiederverwendung einer PID wird dieser Zähler dann wie folgt weitergedreht:

```
inline pid_t wheel(pid_t name) { return name + NPROC; }
```

Angenommen eine solche PID muss für jeden ↑Faden eindeutig sein und das Betriebssystem bietet bis zu 2048 solcher Fäden: $NPROC = 2048 = 2^{12}$. Im Falle einer 32-Bit ↑CPU wären damit bis zu 2^{20} Generationen derselben PID möglich, bevor diese mehrdeutig wird: das heißt, ein und derselbe PCB wurde dann verwendet, um nacheinander 2^{20} Fäden im laufenden Betrieb zu repräsentieren. Für kurzfristige ↑Laufzeiten eines ↑Rechensystems ist eine 32 Bits breite PID gemeinhin ausreichend, für den mittel- und erst recht langfristigen Betrieb sind 64 Bits (`long long int`) geboten, unabhängig von `NPROC` — und unabhängig von der hier vorgestellten möglichen inneren Struktur einer PID.
Abschließend noch eine spezielle PID, über die sich ein Prozess selbst identifizieren beziehungsweise seinen eigenen PCB schnell lokalisieren kann:

```
#define ONESELF (-1 -(NPROC - 1))          /* last generation of process 0 */
```

Verwendet wird dieses Makro, wenn ein Prozess seinen eigenen ↑Prozesszeiger in Erfahrung bringen will (vgl. `being`, S. 252), um eben darüber auf seinen PCB zugreifen zu können.

Prozessinkarnation (en.) ↑*process incarnation*. Verkörperung eines ↑Prozesses, ein konkretes ↑Prozessexemplar (Synonym).

Prozesskontrollblock (en.) ↑*process control block*. Datenstruktur zur Beschreibung einer ↑Prozessinkarnation; ↑PCB. Zentrales Objekt, um für einen ↑Prozess alle von ihm belegten oder benötigten ↑Betriebsmittel, seinen ↑Prozesszustand, seinen ↑Laufzeitkontext, seine Rechte und seine Verwandtschaftsbeziehungen zu anderen Prozessen zu erfassen. Muss ein Prozess auf die Zuteilung eines Betriebsmittels oder aus anderen Gründen warten, erfasst der PCB den Grund des Wartens und enthält gegebenenfalls auch entsprechende Attribute für seine Platzierung auf einer ↑Warteschlange. Um einen Prozess erzeugen, das heißt, eine Prozessinkarnation einrichten zu können, muss das ↑Betriebssystem noch einen PCB im ↑Hauptspeicher belegen können. Für jeden existierenden Prozess gibt es eine solche Datenstruktur. Typischerweise sind diese alle in der ↑Prozesstabelle des Betriebssystems zusammengefasst.

Prozesslokalität (en.) ↑*process locality*. Örtlichkeit eines ↑Prozesses innerhalb eines bestimmten Raums in seiner Lage (in einem ↑Adressraum) und Beschaffenheit (der Folge von ↑Adressen). Dabei legt die Verteilung von Punkten, das heißt, Zugriffen auf „Elementen“ in diesem Raum den Grad der Lokalität eines Prozesses fest. Die Lokalität gilt als stark, wenn die Raumpunkte einen relativ zusammenhängenden Komplex bilden. Anderenfalls ist sie eher schwach (↑*principle of locality*).
Aufbau und Struktur des den Prozess definierenden ↑Maschinenprogramms spielen bei dieser Sache eine wichtige Rolle für den Lokalitätsgrad. Zum einen betrifft dies den Quelltext des ↑Programms, nämlich die Art und Weise des Einsatzes von Sprachkonstrukten zur Formulierung der Lösung für ein algorithmisches Problem. Dabei hilft insbesondere *strukturierte Progammierung*, um dem durch das Programm schließlich definierten (später stattfindenden) Prozess eine starke Lokalität zu verschaffen. Zum anderen kann ein ↑Kompilierer den generierten ↑Maschinenkode samt Datenstrukturen geeignet anordnen (↑*alignment*) und damit die Lokalität des Prozesses stärken, auch wenn dies dem Quelltext nicht anzusehen ist — was jedoch nicht bedeutet, strukturierte Programmierung vernachlässigen zu können.
Entsprechend des Programms beziehungsweise seiner Programmpfade und -abschnitte ist somit die Lokalität des dadurch definierten Prozesses mehr oder weniger stark ausgeprägt. Beispielsweise wird ein gänzlich schleifenloses und ohne ↑Rekursion auskommendes Programm bestenfalls sequentielle Lokalität hervorbringen: der betreffende Prozess beginnt am Programmanfang und schreitet geradewegs bis zum Programmende voran, ohne dabei einen Programmabschnitt zu wiederholen. Andererseits kann ein und dasselbe Programm einem Prozess phasenweise sowohl eine starke als auch eine schwache Lokalität geben, eben in Abhängigkeit von der Lage und Beschaffenheit des Raums (d.h., Programmabschnitts), in dem sich der Prozess gerade befindet.

Prozessor (en.) ↑*processor*. Verarbeitungseinheit; aktive Komponente in einem ↑Rechensystem, die eine Transformation auf ihren ↑Daten vornimmt. Die Vorschrift für diese Transformation liefert ein ↑Programm, das in Form von Hard-, Firm- oder Software vorliegt.

Prozessorauslastung (en.) ↑*processor utilisation*. Zeitanteil der produktiven Arbeit von einem ↑Prozessor, seine ↑Auslastung. Unproduktiv ist der Prozessor typischerweise, wenn er sich im ↑Leerlauf befindet — und dadurch nur mit Nichtstun ausgelastet ist.

Prozessoridentifikation (en.) ↑*processor identifaction*. Nummer (auch: ↑Name) von einem ↑Prozessor oder ↑Rechenkern. Für gewöhnlich eine hardwareabhängige Größe, die im ↑Betriebssystem oder ↑Maschinenprogramm dazu genutzt werden kann, prozessorspezifische Datenstrukturen zu verwalten und zu identifizieren.

Exkurs Die naheliegende Lösung, um einem ↑Prozess die Identifikation oder Kennziffer seines gegenwärtigen Rechenkerns zugreifbar machen zu können, wäre ein reserviertes ↑Prozessorregister, in dem die diesbezügliche Nummer abgelegt ist. Dieses ↑Register wäre Bestandteil des ↑Programmiermodells, allerdings muss die Reservierung nicht zwingend in der Hardware begründet liegen, sondern kann auch durch den ↑Kompilierer geschehen, der dann bestimmte *Arbeitsregister* nicht zur Speicherung gewöhnlicher ↑Daten eines ↑Programms verwendet. Für solch einen Fall genügt eine einfache Zugriffsfunktion auf dieses Register, um

den Rechenkern zu identifizieren, der momentan die „Schirmherrschaft“ (*aegis*) über den diese Funktion aufrufenden Prozess besitzt:

```
inline unsigned aegis() {                    /* readout unique processor ID */
    unsigned name;

    asm volatile("movl %%<coreid>, %0" : "=g" (name));

    return name;
}
```

Hierbei steht `<coreid>` für das die Nummer des Rechenkerns enthaltene Prozessorregister. Die ↑Wortbreite des Registers sollte zweckmäßigerweise Nummern im Wertebereich $[0, N-1]$ zulassen, mit N gleich der Anzahl von Rechenkernen auf dem Prozessor. Damit lässt sich dann beispielsweise direkt ein Index zur Selektion prozessorspezifischer Datenstrukturen aus einem Feld ableiten. Typisches Beispiel für solch eine Selektion ist etwa der Zugriff auf den ↑Prozesszeiger, falls die ↑symmetrische Simultanverarbeitung auf einem ↑Mehrkernprozessor dazu ein Zeigerfeld benutzt.
Wird für die Aufbewahrung kein spezielles, sondern ein für allgemeine Berechnungszwecke nutzbares Prozessorregister verwendet, ist zu beachten, dass dieses nicht Teil des bei ↑Prozesswechseln gesicherten und wiederhergestellten ↑Prozessorstatus sein sollte oder darf. Ansonsten wäre die Wanderung (*migration*) von Prozessen zwischen Prozessorkernen problematisch beziehungsweise unmöglich, sollten die Prozesse eben auch (direkt oder indirekt) Kenntnis von ihrem aktuellen Standort haben müssen.
Wenn die Hardware eine solche Folge von Rechenkernnummern nicht von selbst definiert, ist die Funktion, Rechenkerne eindeutig zu benennen, für gewöhnlich eine initiale ↑Aufgabe des Betriebssystems oder sogar des ↑Urladers:

```
void setup() {                               /* define unique processor ID */
    unsigned name = label();                        /* compute processor ID */

    asm volatile("movl %0, %%<coreid>" : : "g" (name)); /* setup register */
}
```

Der Aufwand für diese Maßnahme steht und fällt mit der Benennungsfunktion (`label`) beziehungsweise mit den in der Hardware jeweils een Merkmalen zur eindeutigen Unterscheidung einzelner Rechenkerne. So kann es sich leicht ergeben, dass diese Funktion entweder komplett oder anteilig nur als ↑privilegierter Befehl zur Ausführung kommen kann und damit nicht direkt im Maschinenprogramm ablaufbar ist. In jedem Fall ist die Berechnung aber hardwareabhängig. Nachfolgend ein Beispiel, dass für Prozessoren von Intel ab Pentium 4 gelten sollte:

```
unsigned label() {                           /* compute unique processor ID */

    /* Readout hardware configuration (e.g. using dip switches or other
     * technical provision, or through the local APIC, if applicable) and
     * map the data obtained onto a linear range of values [0, N-1],
     * with N being the total number of processors in this domain. */

    return (LAPIC_IDR >> 24) & 0xff;         /* use local APIC ID register */
}
```

Im betrachteten Beispiel wird die Nummer des Rechenkerns aus dem Inhalt des Kennregisters seines ↑APIC berechnet. Die Register des APIC können durch ↑speicherabgebildete Ein-/Ausgabe gelesen und geschrieben werden. Hierzu gelten folgende Definitionen für die Basisadresse und das Kennregister des APIC:

```
#define LAPIC_BASE  0xfee00000                /* base within address space */
#define LAPIC_IDR   *(uint32_t *)(LAPIC_BASE + 0x20) /* LAPIC ID register */
```

Sieht nun das Programmiermodell kein solches Prozessorregister als Platzhalter für die Nummer eines Rechenkerns vor, kann natürlich die Benennungsfunktion auch stets von neuem ausgeführt werden, und zwar wenn ein Prozess den ihn vorantreibenden Rechenkern identi-

fizieren muss:

```
inline unsigned aegis() {            /* (re-) compute unique processor ID */
    return label();                              /* deliver computed value */
}
```

Einmal abgesehen davon, dass diese Operation dann gegebenenfalls einen ↑Systemaufruf impliziert, erzeugt der immer wieder anfallende Berechnungsaufwand nicht unerhebliche ↑Gemeinkosten. Für bestimmte Fälle bietet sich daher auch ein ↑Notbehelf an, der den ↑Adressraum eines Prozesses einbezieht und durch eine geeignet ausgelegte ↑Adressabbildung prozessorspezifische Daten auch den Prozessen im Maschinenprogramm direkt zugänglich macht. In dem Fall richtet das Betriebssystem für jeden Prozess, je nach Art der Adressabbildung, entweder eine spezielle ↑Seite oder ein spezielles ↑Segment ein, worüber dann systemspezifische Informationen in den ↑Prozessadressraum eingeblendet werden. An einer vorgegebenen ↑Adresse darin sieht das Betriebssystem sodann ein ↑Speicherwort vor, in dem es die Nummer des den betreffenden Prozess veranstaltenden Rechenkerns ablegt:

```
inline unsigned aegis() {                  /* readout unique processor ID */
    return COREID;               /* assume OS defined paged address space */
}
```

Im hier angenommenen Fall verbirgt sich hinter `COREID` eine Adresse im Prozessadressraum, an der die Kennung des Rechenkerns des Prozesses vermerkt ist:

```
#define COREID *(unsigned *)0x42        /* location in user address space */
```

Wandert der Prozess auf einen anderen Rechenkern, was typischerweise eine Operation des Betriebssystems ist, dann wird das Betriebssystem für den betreffenden Prozess die an dieser Adresse gespeicherte Kennung auch aktualisieren müssen, und zwar noch bevor der Prozess auf dem neuen Rechenkern seine Tätigkeit in dem dort für ihn gültigen Adressraum fortsetzt, spätestens bevor der Prozess (mittels `aegis`) seine Rechenkernkennung abruft.
Der Nachteil dieser Lösung ist jedoch einerseits die Abhängigkeit von einer solchen Adressabbildung sowohl durch das Betriebssystem als auch unterstützt durch eine ↑MMU und andererseits die Notwendigkeit der Aktualisierung dieses Speicherworts, wann immer der Prozess den Rechenkern wechselt. Darüberhinaus muss jeder Prozess, und zwar unabhängig von seiner konkreten technischen Auslegung (als ↑*thread*, ↑*fiber*, ↑*fibril* o.a.), über eine solche Adressabbildung verfügen, womit Prozesswechsel dann zumindest in Teilen auch einen Adressraumwechsel nach sich ziehen und an Aufwand teils erheblich zunehmen.

Prozessorkonsistenz (en.) ↑*processor consistency*. Modell der ↑Speicherkonsistenz, nach dem jede von einem einzelnen ↑Prozessor auf den ↑Arbeitsspeicher durchgeführte Schreiboperation für andere Prozessoren in genau der Reihenfolge sichtbar ist, wie sie von ihm ausgegeben wurde. Allerdings können Schreiboperationen, die von verschiedenen Prozessoren ausgegeben wurden, von diesen Prozessoren in unterschiedlicher Reihenfolge sichtbar sein. Schwächer als ↑sequentielle Konsistenz.

Prozessorregister (en.) ↑*processor register*. Behältnis (↑*register*) in der ↑CPU, um ↑Daten zu speichern. Jedes ↑Exemplar davon ist höchstens eine ↑Wortbreite groß. Es dient der eher kurzzeitigen Aufbewahrung von einem ↑Speicherwort/↑Maschinenwort oder Teilen davon, um in der CPU durch einen ↑Maschinenbefehl verarbeitet werden zu können. Die ↑Zugriffszeit auf das aufbewahrte Datum entspricht der ↑Taktfrequenz der CPU.

Prozessorstatus (en.) ↑*processor status*. Bezeichnung des im ↑Programmiermodell der ↑CPU für einen ↑Prozessor definierten Zustands.

Prozessorstatuswort (en.) ↑*processor status word*. Bezeichnung für ein spezielles ↑Speicherwort, dessen Inhalt im ↑Unterbrechungszyklus zusammengestellt wird und das im ↑Stapelspeicher lokalisiert ist. Neben der Kopie vom ↑Statusregister umfasst dieses Wort typischerweise weitere ↑Daten zur Statusüberwachung der ↑CPU und zur Steuerung ihrer Operation am Ende

von dem Unterbrechungszyklus, der zur ↑Ausnahmebehandlung im ↑Betriebssystem geführt hat.
Der unter dem Akronym ↑PSW bekannte Begriff steht ebenfalls für ein spezielles ↑Prozessorregister, das die um systemrelevante Statusinformationen erweiterte Funktion eines Statusregisters bereitstellt. Die zusätzlichen Informationen sind typischerweise eine ↑Unterbrechungsmaske und der ↑Arbeitsmodus. Während die Manipulation der im Statusregister enthaltenen Informationen für gewöhnlich keine besonderen Privilegien erfordert (↑*unprivileged mode*), ist der erweiterte Status nur auf ↑Systemebene änderbar (↑*privileged mode*).

Prozessortakt (en.) ↑*processor clock*. Bezeichnung für die kleinste Phase im Rythmus synchronisierter Vorgänge in einem ↑Prozessor (in Anlehnung an den Duden). Bestimmt zusammen mit der ↑Taktfrequenz die für einen ↑Befehlszyklus jeweils zu veranschlagende Zeit.

Prozessorzyklus (en.) ↑*processor cycle*. Kreislauf reiner regelmäßig wiederkehrenden, in sich geschlossenen Folge zusammengehöriger Vorgänge innerhalb eines ↑Prozessors zur Verarbeitung eines ↑Maschinenbefehls (in Anlehnung an den Duden). Entspricht einem ↑Befehlszyklus. Den Beginn der einzelnen Verarbeitungsschritte gibt jeweils der ↑Prozessortakt vor. Die Anzahl der für einen Zyklus benötigten Takte ist ein *relatives Maß* für die ↑Ausführungszeit eines Maschinenbefehls oder einer Folge von Maschinenbefehlen, unabhängig von der ↑Taktfrequenz, mit der der Prozessor operiert.

Prozesssperre (en.) ↑*process lock*. Maßnahme zum Sperren eines ↑Prozesses (in Anlehnung an den Duden), nämlich dass dieser den ↑Prozessor nicht erneut vor Eintritt eines bestimmten ↑Ereignisses zugeteilt bekommt.

Prozesssteuerung (en.) ↑*process control*. Führung eines externen (physikalisch-technischen) Prozesses durch ein ↑Rechensystem, bei der ein fortlaufender Informationsaustausch erfolgt, um diesen Prozess zu überwachen, zu steuern und/oder zu regeln. Dazu werden die Vorgänge innerhalb des Rechensystems durch ↑Echtzeitbetrieb gelenkt.
Der externe Prozess ist nicht mit einem „internen ↑Prozess“ zu verwechseln: ersterer ist kein ↑Programm in Ausführung, gleichwohl ist ein solches Programm für seine ↑Steuerung oder ↑Regelung durch ein Rechensystem erforderlich. Bei ↑Programmablauf werden ↑Prozessdaten verarbeitet, wodurch für gewöhnlich der externe Prozess eine bewusste, gewollte und gezielte Beeinflussung erfährt (↑PDV).

Prozesstabelle (en.) ↑*process table*. Feld begrenzter Länge im ↑Betriebssystem, wobei jedes Feldelement ein ↑Prozesskontrollblock ist. Normalerweise eine statische Datenstruktur, deren Größe (d.h., Anzahl der Feldelemente) ein *Systemparameter* zur Konfigurierung des Betriebssystem bildet. Ist jeder einzelne Tabelleneintrag für eine ↑Prozessinkarnation belegt, kann ein weiterer ↑Prozess erst dann erzeugt und eingerichtet werden, wenn ein anderer Prozess terminiert, seine Prozessinkarnation gelöscht und dadurch ein Tabelleneintrag frei wurde.

Prozessverpflichtung (en.) ↑*process obligation*. Tätigkeit, zu der ein ↑Prozess verpflichtet ist (in Anlehnung an den Duden). Der betreffende Prozess kommt der Verpflichtung nach, eine bestimmte Schuld, die ein anderer Prozess hinterlassen hat, zu begleichen. Den Ausgangspunkt bilden durch ein ↑nichtsequentielles Programm direkt oder indirekt gekoppelte Prozesse, die entweder durch ↑Aktionen auf eine gemeinsame Datenstruktur (↑*shared data*) oder allgemein durch eine bestimmte ↑Kontrollstruktur eng miteinander verflochten sind. Wenn es dann einem Prozess, der aufgrund eines für ihn momentan gültigen Zustands unaufhaltsam einem bestimmten ↑Ereignis zustrebt, unmöglich ist, die getroffene Entscheidung für sein weiteres Fortgehen zu revidieren, übernimmt ein anderer Prozess die Verpflichtung, sein eigenes weiteres Voranschreiten auf Richtigkeit hin zu prüfen und eine eventuell aufgetretene Unstimmigkeit zu beseitigen.
Angenommen Prozesse unterliegen der ↑Vorrangplanung und der gegenwärtig auf dem ↑Prozessor stattfindende Prozess, P_α, der die höchste ↑Priorität hat, erwartet ein bestimmtes

Ereignis und blockiert daraufhin. Für den anstehenden ↑Prozesswechsel wählt P_α für gewöhnlich selbst seinen Nachfolgeprozess, P_γ, dessen Priorität niedriger als die von P_α sein kann, aber die höchste aller bereitgestellten Prozesse ist. Auf dem Wege hin zum Prozesswechsel tritt nunmehr das erwartete Ereignis ein, das die Blockierungsbedingung für P_α aufhebt. Da P_α allerdings die Bedingungsauswertung bereits passiert hat, kann er die Entscheidung zum Prozesswechsel hin zu P_γ nicht mehr rückgängig machen. Daher kommt P_γ, nachdem er seine Tätigkeit aufgenommen hat, der Verpflichtung nach, die von P_α zuvor getroffene Auswahlentscheidung auf Richtigkeit in Bezug auf die Vorrangplanung hin zu prüfen. P_γ fährt nur dann fort, wenn er immer noch der höchstpriorisierte aller bereitgestellten Prozesse ist. Anderenfalls leitet er den sofortigen Prozesswechsel ein.
Angenommen die Vorrangplanung sieht die ↑Verdrängung eines Prozesses von seinem Prozessor vor und der gegenwärtig auf dem Prozessor stattfindende Prozess, P_α, niedrigerer Priorität stellt den höher priorisierten Prozess P_γ bereit. Dann muss es zur Verdrängung von P_α durch P_γ kommen: sobald P_α dies erkennt, leitet er den Prozesswechsel hin zu P_γ. Wenn nun auf dem Wege von P_α hin zur Übergabe des Prozessors an P_γ eine ↑Unterbrechungsanforderung eintrifft, die P_α unterbricht und deren Behandlung die Bereitstellung eines Prozesses, P_δ, noch höherer Priorität als P_γ nach sich zieht, dann kann P_α nicht mehr dazu veranlasst werden, P_δ anstatt von P_γ aufzunehmen. Stattdessen wird P_δ vermerkt, P_α mit Beendigung der ↑Unterbrechungsbehandlung fortgesetzt und demzufolge auch zu P_γ gewechselt. Sobald P_γ seine Tätigkeit aufnimmt, überprüft er die Richtigkeit seines Daseins: P_γ fährt nur dann fort, wenn er immer noch der höchstpriorisierte aller bereitgestellten Prozesse ist. Anderenfalls leitet er den sofortigen Prozesswechsel ein.
Offensichtlich erfordern solche Szenarien eine geeignete ↑Nebenläufigkeitssteuerung. Dabei ist jedoch zu beachten, dass (nicht nur) für die hier geschilderten typischen Fälle das *Muster* solcher Verpflichtungen gang und gäbe ist. Hinter jeder Art ↑Synchronisation ↑gleichzeitiger Prozesse verbirgt sich solch ein Schema, nur die dafür verwendeten Protokolle sind zuweilen höchst verschieden. Beispielsweise kann durch eine Aussperrung der ↑Aktionsfolge, die die Ungültigkeit der getroffenen Auswahlentscheidung zur Folge hat, der drohenden Verletzung der Zuteilungsstrategie vorgebeugt werden. Einmal von den unerwünschten Eigenschaften solch einer Methode abgesehen (erhöhte/unbestimmte ↑Unterbrechungslatenz bzw. ↑Antwortzeit der Unterbrechungsbehandlung bei ↑Pegelsteuerung, verpasste Unterbrechungsanforderung bei ↑Flankensteuerung), so ist in dem Fall die Aufhebung der Sperre, die der den Prozessor abgebende Prozess (P_α) dann beizeiten zu erheben hat, nichts anderes als wiederum eine Verpflichtung, nämlich des den Prozessor zugeteilt erhaltenen Prozesses (P_γ). Für gewöhnlich ist genau in dieser Weise ein ↑kritischer Abschnitt, innerhalb von dem ein Prozesswechsel geschieht, zu handhaben. Unverkennbar überträgt jede Form von ↑blockierende Synchronisation die Verpflichtung zum Verlassen eines solchen kritischen Abschnitts an einen anderen Prozess. Für ↑nichtblockierende Synchronisation ergibt sich kein anderes Bild, zumal die oben geschilderten Fälle nichts weiter als eine spezielle Variante dieser Art der ↑Synchronisierung darstellen. Die dafür benötigten Synchronisationsprotokolle verwenden herkömmliche Kontrollstrukturen in einem nichtsequentiellen Programm und kommen nicht selten auch ohne ↑atomare Operationen aus, wie etwa die im Zusammenhang mit ↑SRTF diskutierten „neuralgischen Punkte“ (z.B. `ready`, S. 324 und `block`, S. 326) belegen.

Prozesswechsel (en.) ↑*process switch.* ↑Aktionsfolge, die den aktuellen ↑Prozess aus- und einem ausgewählten Prozess einsetzt. Die Aktionsfolge sichert den ↑Laufzeitkontext des aktuellen Prozesses und stellt dann den Laufzeitkontext des einzusetzenden Prozesses wieder her. Darüberhinaus ist der ↑Prozesszeiger zu aktualisieren. Dieser Zeiger ist die ↑Adresse vom ↑PCB des zum Zeitpunkt auf dem ↑Prozessor stattfindenden Prozesses: für das ↑Betriebssystem hat der Prozesszeiger eine ähnliche Funktion, wie der ↑PC für die ↑CPU. Die Umschaltung zwischen Prozessen ist die zentrale Funktion der ↑Einlastung, für sie ist der ↑Umschalter zuständig.

Exkurs Einen Prozess aus- und einen anderen Prozess einzusetzen, also den Prozessor

mit einem Prozess zu belegen (`seize`), ist ein nicht allzu komplizierter Vorgang — wenn von einem ↑präemptiven Ansatz einmal abgesehen wird. In letzter Konsequenz ist die hierfür erforderliche Aktionsfolge aber immer prozessorabhängig. Nachfolgend ein Beispiel, das den prinzipiellen Ablauf eines solchen Wechsels für den nichtpräemptiven Fall veranschaulicht:

```
process_t *seize(process_t *task) {                    /* dispatch process */
    process_t *self = being(ONESELF);            /* own process descriptor */

    state(&self->mood, CLEAR|RUNNING);       /* cancel running state, only */
    process_t *back = shift(task);         /* switch process, save context */
    state(&self->mood, FLUSH|RUNNING);       /* define running state, also */

    return back;                             /* deliver predecessor handle */
}
```

Den Zugriff auf den PCB des jeweils stattfindenden Prozesses ermöglicht eine Funktion (`being`), die die Adresse der dem PCB entsprechenden Datenstruktur berechnet und als Prozesszeiger zurückliefert. Damit kann der ↑Prozesszustand laufend des betreffenden Prozesses gelöscht werden. Anschließend erfolgt die wirkliche Umschaltung (`shift`) und Zuweisung des Prozessors an den anderen Prozess (`task`). Diese Operation bewirkt den physischen Wechsel, sie sorgt dafür, einen zeitweilig ausgesetzten Prozess später wieder aufnehmen zu können. Dazu sichert sie seinen ↑Prozessorstatus und vermerkt eine diesbezügliche ↑Handhabe im PCB des betreffenden Prozesses (s. unten). Wird der weggeschaltete Prozess (`self`) später wieder aufgenommen, liefert die entsprechende Operation den Zeiger auf den PCB des Prozesses, der die Umschaltung zu ihm hin veranlasst hat. Diese Handhabe für den Vorgängerprozess (`back`) geht als Funktionswert zurück (`return`). Zuvor wird der Prozesszustand abgeglichen (`FLUSH`) und wieder auf laufend gesetzt.
In dem Beispiel bildet eine komplexe ↑Koroutine die Basis der ↑Prozessinkarnation: jedem Prozess ist damit ein eigener ↑Laufzeitstapel zugeordnet. Darüber hinaus besteht der invariant zu haltende ↑Koroutinenstatus aus jenen ↑Prozessorregistern, deren Inhalte über Funktionsaufrufe hinweg erhalten bleiben (↑*non-volatile register*). Damit kann die Prozessumschaltung auf einen ↑Koroutinenwechsel wie folgt abgebildet werden:

```
process_t *shift(process_t *next) {                     /* switch process */
    process_t *self = throw(next);                  /* swap process handle */

    unload(INNER|CDECL, 0);                  /* backup own processor state */
    coroutine_t *back = resume(next->flow.crux);       /* switch coroutine */
    reload(INNER|CDECL, 0);                  /* define own processor state */

    process_t *task= educe(back);                 /* derive process handle */
    task->flow.crux = back;                       /* save coroutine handle */

    return task;                                 /* deliver process handle */
}
```

Die erste Operation (`throw`) liest und definiert den Prozesszeiger in einem logischen Schritt, als ↑Elementaroperation. Allerdings ist der zweite Schritt, nämlich die Definition des neuen Zeigers (`next`) optional: er entfällt, wenn in der jeweiligen Konfiguration des ↑Betriebssystems der Prozesszeiger aus dem ↑Stapelzeiger eines Prozesses hervorgebracht wird und dazu dann die betreffende Funktion (`educe`) entsprechend ausgelegt ist.
Prozessumschaltung (`shift`) wie auch Koroutinenwechsel (`resume`) sind hier jeweils eine Funktion, die einem wieder aufgenommenen Handlungsstrang die Handhabe des vorangegangenen (`back` bzw. `task`) Handlungsstrangs liefert: letzterer hat die Einlastung des Prozessors veranlasst. Diese Handhabe (nämlich `back`) ermöglicht die spätere Fortsetzung des zeitweilig ausgesetzten Prozesses, sie identifiziert seinen gesicherten Prozessorstatus und ist daher im PCB des betreffenden Prozesses (`task`) einzutragen. Der Zeiger auf diesen PCB wird aus der gelieferten Handhabe hervorgebracht (`educe`. s. aber den Hinweis oben), um ihn dann auch als Funktionsergebnis zurückzuliefern.
Wie das Beispiel (`shift`) zeigt, stellt jeder Prozess eigenständig den Prozessorstatus wie-

der her (`reload`), den er zuvor selbst gesichert hat (`unload`). Als Sicherungsspeicher dient der ↑Laufzeitstapel des Prozesses (`INNER`, wobei `CDECL` auch nur die nichtflüchtigen Register einbezieht). Der Wechsel zwischen Prozessen ist damit prozessspezifisch ausgelegt, und zwar hinsichtlich Aufbau und Struktur von dem jeweiligen Prozessorstatus. Den Prozessen gemeinsam ist lediglich das jeweils verwendete Koroutinenkonzept.
Normalerweise erfolgt der Wechsel vom aktuellen hin zu einen anderen Prozess. Jedoch kann es auch Situationen geben, wo beide Prozesse identisch sind: beispielsweise wenn der aktuelle Prozess als ↑Leerlaufprozess agiert, also logisch blockierte, jedoch durch seine Deblockierung zwischenzeitlich wieder bereitgestellt und vom ↑Planer als nächster einzusetzender Prozess ausgewählt wurde. Ob ein Prozess zu sich selbst wechseln kann, hängt ganz von der Schnittstelle und der Implementierung von dem ↑Unterprogramm ab, das typischerweise für solch einen Wechsel aufzurufen ist. Dieses Unterprogramm benötigt dann zwei Referenzparameter, die jeweils einen ↑PCB adressieren, und es muss den PCB des aktuellen Prozesses nach erfolgter Kontextsicherung aktualisiert haben, bevor auf den PCB des einzusetzenden Prozesses zugegriffen wird. Auch wenn technisch möglich: Ein Prozesswechsel zu sich selbst bringt für den aktuellen Prozess keinen Fortschritt und bedeutet daher nur Unkosten.

Prozesszeiger (en.) ↑*process pointer*. Zeiger auf den ↑Prozesskontrollblock von dem ↑Prozess, der gegenwärtig auf einem ↑Rechenkern stattfindet.

Exkurs Im Allgemeinen verwaltet ein ↑Betriebssystem pro Rechenkern einen solchen Zeiger. Um den zum Zeitpunkt stattfindenden Prozess zu lokalisieren, gibt es für gewöhnlich zwei Herangehensweisen. Eine davon macht diesen Zeiger als wirkliches ↑Exemplar im Betriebssystem entbehrlich und berechnet ihn stattdessen auf Grundlage eines speziellen ↑Prozessorregisters. Ein anderer, naheliegender Ansatz ist ein Zeigerfeld mit soviel Einträgen wie Rechenkerne mit Prozessen betrieben werden sollen. Die Identifikation (Nummer) des Rechenkerns dient dann als Feldindex, um auf den zugeordneten Zeiger zugreifen zu können.
Bevor diese beiden Herangehensweise nun im Detail vorgestellt werden, ein Satz von Funktionen, die unabhängig von der technischen Auslegung eines solchen Zeigers sind und der Abstraktion von nicht offensichtlichen Problemen im Umgang damit dienen. Ausgangspunkt für die nachfolgende Betrachtung sei zunächst nachfolgend skizzierte, zentrale Funktion:

```
inline process_t *being(pid_t name) {          /* compute process pointer */
    return (name == ONESELF) ? cycle() : point(name);
}
```

Funktionswert ist ein Zeiger, der entweder den auf dem ↑Prozessor gerade stattfindenden (`ONESELF`, S. 245) oder einen anderen (↑Prozessidentifikation ungleich `ONESELF`) Prozess lokalisiert. Im letzteren Fall wird der ↑PCB auf einer Liste nachgeschlagen (`point`), die typischerweise in Form der ↑Prozesstabelle implementiert ist:

```
inline process_t *point(pid_t name) {           /* map PID to PCB pointer */
    extern process_t folk[];                                /* process table */
    process_t *this = &folk[ratio(name)];                     /* locate PCB */
    return (this->name == name) ? this : 0;  /* check PID, return pointer */
}
```

Hier wird davon ausgegangen, dass die Prozessidentifikation (`name`) eine Kennziffer (`ratio`) enthält, die einem Feldindex für die Prozesstabelle entspricht und den zugehörigen PCB leicht lokalisierbar macht. Jedoch liefert die Funktion nur dann einen gültigen Zeigerwert (`this`), wenn die angegebene ↑PID auch tatsächlich mit dem lokalisierten PCB verknüpft ist. Dazu wird die präsentierte PID mit der als Attribut im PCB vermerkten wirklichen PID des betreffenden Prozesses verglichen. Nur bei Gleichheit kommt die ↑Adresse, an der der PCB im ↑Adressraum des Betriebssystems liegt, als Zeigerwert zurück.
Eine weitere, von der technischen Auslegung des Zeigers unabhängige Funktion sorgt für die Umschaltung des Zeigerwerts. Diese Funktion kommt beim ↑Prozesswechsel zur Geltung:

```
inline process_t *throw(process_t task) {   /* fetch & store process pointer
*/
    process_t *self = being(ONESELF);                    /* fetch pointer */
    apply(task);                          /* store pointer, if applicable */
    return self;              /* deliver pointer to still running process */
}
```

Funktionsergebnis ist immer der Zeiger auf den gegenwärtig auf dem Prozessor stattfindenden Prozess (`self`). Gegebenenfalls wird dieser Zeigerwert aktualisiert (`apply`), falls der Zeiger dediziertes Exemplar einer eigens dafür vorgesehenen globalen ↑Variablen ist.

Zunächst nun der Ansatz, der pro Prozessor eine globale Variable verwendet, um den PCB des gegenwärtig stattfindenden Prozesses zu adressieren. Die Variablen sind in einem Zeigerfeld zusammengefasst, das mit einer ↑Prozessoridentifikation indiziert wird, um den jeweils für einen bestimmten Prozessor vorgesehenen Eintrag zu selektieren:

```
extern process_t *life[];       /* one process pointer per processor core */
```

Bei jedem Prozesswechsel ist der dem betreffenden Prozessor zugehörige Eintrag mit dem Zeiger auf den PCB des einzusetzenden Prozesses zu aktualisieren. Diese Aktualisierung geschieht vor oder nach dem damit einhergehenden ↑Koroutinenwechsel:

```
inline void apply(process_t *task) {            /* define process pointer */
    life[aegis()] = task;
}
```

Von dem exakten Moment der Zeigeraktualisierung einmal abgesehen, so bedeutet dies aber, dass direkt vor oder nach der Operation zum Koroutinenwechsel (`resume`, S. 149) eine Koroutine läuft, die nicht durch den dann jeweils aktuellen Zeiger auf dem PCB eindeutig identifiziert werden kann. Die beiden Einzelschritte, nämlich Zeigeraktualisierung und Koroutinenwechsel, bilden keine ↑unteilbare Anweisung, sie können eine ↑Wettlaufsituation hervorbringen, wenn ↑präemptive Planung im Hintergrund läuft. Bei diesem Ansatz muss daher durch geeignete ↑Nebenläufigkeitssteuerung direkt in oder im Kontext der Operation zum *physischen Prozesswechsel* (`shift`, S. 251) einem möglichen Fehlverhalten vorgebeugt werden. Typischerweise geschieht dies durch eine ↑Verdrängungssperre, die vor dem ersten dieser Einzelschritte in der den Prozessor abgebenden Koroutine zu erheben und nach erfolgter Prozessorübergabe in der dann aufgenommenen Koroutine aufzuheben ist. Letzteres geschieht dann im Rahmen einer ↑Prozessverpflichtung.
Auf Basis einer so applizierten Zeigervariablen, lässt sich sodann der zur Zeit auf dem Prozessor stattfindende Arbeitsgang einfach dadurch in Erfahrung bringen, indem der Zeiger auf den zugehörigen PCB über das Zeigerfeld selektiert wird:

```
inline process_t *cycle() {                     /* return process pointer */
    return life[aegis()];
}
```

Ist der PCB nicht Bestandteil der Prozesstabelle, sondern in einer anderen prozessbezogenen Datenstruktur enthalten, wird sein Zeiger aus der Adresse eben dieser Datenstruktur abgeleitet. Wie weiter unten noch als Alternative zum Zeigerfeld betrachtet wird, handelt es sich bei dieser Datenstruktur um den ↑Laufzeitstapel eines Prozesses. Die Buchführung des von einem Prozessor jeweils veranstalteten Prozesses auf Grundlage eines Zeigerfelds macht die Ableitungsfunktion jedoch obsolet, weshalb sie im gegebenen Fall hier den Wert null liefert — für gewöhnlich wird diese Funktion auch nur in eben dieser alternativen Auslegung des Zeigers genutzt:

```
inline process_t *educe(void *none) {           /* derive process pointer */
    return 0;                                                   /* unused! */
}
```

Wie oben erwähnt wird das Zeigerfeld mit der Prozessoridentifikation indiziert, genauer mit einer daraus extrahierten Kennziffer, die den Prozessor beziehungsweise Rechenkern, der die

Schirmherrschaft (`aegis`) für den die Indexoperation durchführenden Prozess innehat, eindeutig bestimmt. Im Falle eines ↑Uniprozessors ist diese Maßnahme unkritisch, nicht jedoch, wenn ein ↑Multiprozessor die Grundlage bildet und dieser insbesondere durch ↑symmetrische Simultanverarbeitung betrieben werden soll. Letzteres kann eine weitere Wettlaufsituation hervorbringen, nämlich wenn ein Prozess die Kennziffer seines Prozessors zur Selektion seines PCB über das Zeigerfeld bereits in der Hand hat — das heißt, die Funktion `aegis` wurde ausgeführt und ihr Wert wurde prozesslokal (z.B. in einem ↑Prozessorregister) zwischengespeichert —, dann aber vom Betriebssystem auf einen anderen Prozessor überführt wird. Dem anderen Prozessor ist eine andere Kennziffer zugeordnet und damit auch einem anderen Eintrag in dem Zeigerfeld. Vollendet der auf einen anderen Prozessor überführte Prozess dann seinen Feldzugriff, ist es es möglich, dass er den falschen Feldeintrag selektiert, nämlich entweder überschreibt (`apply`) oder ausliest (`cycle`). Beiden Situationen ist durch Maßnahmen zur Nebenläufigkeitssteuerung vorzubeugen, was entweder auf eine ↑Umlaufsperre hinausläuft oder ↑nichtblockierende Synchronisation nach sich ziehen muss.

Der zweite Ansatz zur Verwaltung des Zeigers auf den PCB des jeweils laufenden Prozesses ist frei von den beiden Wettlaufsituationen, die den ersten (ein Zeigerfeld nutzender) Ansatz noch betreffen. Als Alternative bietet sich nämlich an, den Zeigerwert für einen PCB immer explizit abzuleiten aus dem Wert eines prozessorlokalen ↑Registers, das implizit mit jedem Koroutinenwechsel beziehungsweise jeder Prozessüberführung aktualisiert wird. Hierfür bietet sich der ↑Stapelzeiger eines als *komplexe Koroutine* ausgelegten Prozesses an. Da der Wechsel zwischen dieser Art von Koroutinen immer auch auf den Austausch des Stapelzeigerinhalts hinausläuft, wird damit dann automatisch auch und atomar der Prozesszeiger umgesetzt. Dazu ist der PCB oder ein Zeiger darauf entweder am Anfang oder am Ende des ↑Stapelsegments eines jeweiligen Prozesses zu platzieren. Wenn darüber hinausgehend die Stapelsegmente aller Prozesse eine feste Zweierpotenzgröße besitzen, kann durch Maskierung des Inhalts des ↑SP die Adresse des PCB leicht bestimmt werden:

```
inline process_t *educe(void *sp) {             /* derive process pointer */
    return (process_t *)(((word_t)sp & PMASK) + PFILL - sizeof(process_t));
}
```

Als ↑tatsächlicher Parameter der Funktion (`educe`) ist von einem gültigen Stapelzeigerwert (`sp`) auszugehen. Die Zweierpotenzgröße des Stapelsegments sei mit `SSIZE` bezeichnet. Der Wert von `PMASK` ist dann `-SSIZE`. Nun angenommen der Wert von `PFILL` sei definiert als `sizeof(process_t)`, dann liefert die Funktion die Adresse des am Anfang des Stapelsegments liegenden PCB: die CPU führt mit dem Stapelzeigerwert als Operand die einfache bitweise Operation *UND* aus. Ist der Wert von `PFILL` dagegen `SSIZE` gleichgesetzt, zeigt die gelieferte Adresse auf den PCB am Ende des Stapelsegments: die CPU addiert zur zwischenberechneten Basisadresse die Differenz $SSIZE - sizeof(PCB)$. So ergeben sich zwei verschiedene Varianten der Platzierung des PCB im Stapelsegment, die jedoch beide eine recht effiziente Berechnung des Zeigers auf den PCB eines Prozesses ermöglichen und eine konstruktive Maßnahme bilden, damit bestimmte Wettlaufsituationen beim Umgang mit solch einem Zeiger gar nicht erst auftreten können.
Im Gegensatz zum ersten Ansatz mit einer Zeigervariablen (d.h., einem Zeigerfeld von nur einem Eintrag), die für gewöhnlich mit einem einzigen ↑Maschinenbefehl gelesen werden kann, sind hier nunmehr, neben dem Lesen des Stapelzeigerregisters, noch ein oder zwei weitere Maschinenbefehle notwendig. Dies relativiert sich bereits, wenn ein echtes Zeigerfeld (das also mehr als einen Eintrag aufweist) zugrunde gelegt wird: dann sind schon wenigstens zwei Maschinenbefehle zum Auslesen des Zeigers nötig. Aber in beiden Fällen besteht dann bei präemptiver Planung keine Notwendigkeit zur Verdrängungssperre, die wenigstens drei bis vier Maschinenbefehle erfordert und, was schwerer wiegt, darüber hinaus gleichzeitige Prozesse zeitweilig unterbindet. Ist dann sogar noch eine Umlaufsperre in Betracht zu ziehen, erweist sich die Stapelzeigerlösung unzweifelhaft als die Variante mit den geringeren ↑Gemeinkosten: die Lösung mit der Zeigervariablen ist nur von Vorteil, wenn strikt ↑kooperative

Planung im Hintergrund läuft.
Der Vollständigkeit halber noch die beiden anderen, zur Abstraktion von der technischen Auslegung des Zeigers auf den aktuellen Prozess eingeführten Funktionen:

```
inline process_t *cycle() {                    /* derive process pointer */
    return educe(sp());           /* take stack pointer register as basis */
}
inline void apply(process_t *none) {}  /* define process pointer: unused! */
```

Da der Zeiger in dieser Variante berechnet und nicht explizit durch eine Variable repräsentiert wird, bleibt die Operation zur Definition des Zeigerwerts leer — der ↑Kompilierer erzeugt hier für gewöhnlich keinen ↑Text und damit auch keine überflüssigen Gemeinkosten. Der Zeiger auf den aktuell auf einem Prozessor laufenden Arbeitsgang (`cycle`) wird nun vom Wert des Stapelzeigerregisters eben dieses Prozessors abgeleitet. Den Wert dieses Prozessorregisters liefert eine spezielle Funktion (`sp`), die dazu in ↑Assembliersprache formuliert ist:

```
inline void *sp() {                     /* readout stack pointer register */
    void *aux;                                        /* auxiliary variable */
    asm volatile("movl %%esp, %0" : "=g" (aux))        /* inline assembly */
    return aux;                  /* deliver pointer value to top of stack */
}
```

Vor allem für ein ↑prozessbasiertes Betriebssystem ist die Stapelzeigerlösung sehr zweckmäßig, zumal die feste Zweierpotenzgröße und ↑Ausrichtung des Laufzeitstapels eines jeweiligen Prozesses keine wirkliche Hürde darstellt. Die mögliche maximale Ausdehnung des Laufzeitstapels eines Prozesses, der sich im Betriebssystem bewegt, kann für gewöhnlich auf Grundlage von ↑Vorwissen zum stapelspeicherintensivsten Programmpfad abgeschätzt oder sogar recht genau bestimmt werden. Dazu wird eine (manuelle/automatisierte) statische Programmanalyse des Betriebssystems durchgeführt, in die dann Wissen zur minimalen ↑Zwischenankunftszeit von ↑Unterbrechungsanforderungen einerseits und möglichen Verschachtelung von ↑Unterbrechungshandhaber verschiedener ↑Unterbrechungsprioritätsebenen andererseits einfließen muss. Die im Zusammenhang mit ↑Unterbrechungen stehenden Punkte bedeuten letztlich nicht programmierte Aufrufe von ↑Unterprogrammen, deren jeweilige Ausführung dann zusätzlichen Bedarf an Stapelspeicher impliziert. Der nächst größere Zweierpotenzwert der schlimmstmöglichen Ausdehnung bezogen auf die effektiv benötigte Anzahl von ↑Bytes ergibt den für jeden einzelnen ↑Systemkernfaden zu reservierenden Stapelspeicher. Dabei sind etwa Größen um 8 KiB pro Laufzeitstapel für ein System wie ↑Linux nicht ungewöhnlich.

Prozesszustand Situation, in der sich ein ↑Prozess augenblicklich befindet. Der Prozess ist bereit (*ready*) zur ↑Einlastung, wenn alle von ihm benötigten ↑Betriebsmittel bis auf den ↑Prozessor zur Verfügung stehen und letzteren aber mit anderen Prozessen teilen muss. In solch einem Fall muss der Prozess innehalten, bis der für ihn geeignete Prozessor frei wird. Ein Prozess, der stattfindet, ist laufend (*running*) auf einem Prozessor. Dieser Prozess kann zugunsten eines anderen Prozesses pausieren, dem dann der Prozessor zugeteilt wird. Der damit einhergehende ↑Prozesswechsel geschieht freiwillig oder unfreiwillig (*präemptiv*). Benötigt ein stattfindender Prozess ein weiteres Betriebsmittel, das in dem Moment jedoch nicht zur Verfügung steht, wird er blockiert (*blocked*). Dem folgt ein Prozesswechsel, wenn der Prozess sich den Prozessor mit anderen Prozessen teilen muss und wenigstens einer der anderen Prozesse in Bereitschaft steht. Steht in dem Moment kein anderer Prozess zur Einlastung bereit, wird der Prozessor untätig (↑*idle*). In der Situation wird nur ein externer Prozess dem Prozessor wieder eine Tätigkeit verschaffen können, beispielsweise durch eine ↑Unterbrechungsanforderung. Mit Zuteilung des benötigten Betriebsmittels wird ein blockierter Prozess deblockiert und wieder bereitgestellt. Diese Zustandsübergänge kontrolliert ein im ↑Betriebssystem mitlaufender (*on-line*) ↑Planer.
Darüber hinaus kann sich ein Prozess auch noch in einem ↑Schwebezustand befinden, der für gewöhnlich jedoch nur aufgrund bestimmter ↑Betriebsarten möglich ist. Typische Beispiele

sind im Zusammenhang mit der ↑Umlagerung von Prozessen zu finden, aber gerade auch beim Zusammenspiel von ↑Unterbrechungsbehandlung und ↑mitlaufende Planung.

Exkurs Die Art und Weise der Zustandsverwaltung zur ↑Prozesseinplanung ist nicht unwesentlich bestimmt durch die jeweilige Betriebsart. Wenn etwa strikt ↑kooperative Planung die Grundlage bildet, dabei dann sogar Bereitstellungen von Prozessen aus der Unterbrechungsbehandlung heraus unzulässig sind und das alles nur für einen ↑Uniprozessor ausgelegt werden muss, dann bildet jeder Zustandsübergang jedes Prozesses implizit eine synchrone Operation. Dies ändert sich, wenn ↑präemptive Planung in Betracht zu ziehen ist, da dann eine Unterbrechungsanforderung den Planer ins Geschehen bringen kann: folglich ist auch im Uniprozessorfall jeder Zustandsübergang schon durch eine ↑Elementaroperation zu gewährleisten. Die Art der ↑Synchronisation der betreffenden ↑Aktionen kann hinfällig sein und ist durch eine andere Variante zu ersetzen, wenn ein ↑Multiprozessor als Basis angenommen werden muss. Daher ist es geboten, die betreffenden Operationen durch eine geeignete *funktionale Abstraktion* zu verallgemeinern.
Die technische Repräsentation des typischerweise im ↑Prozesskontrollblock enthaltenen Zustandsattributs zur Prozesseinplanung kann kompakt in Form einer ↑Bitleiste ausgelegt sein. In dem Fall sind komplexe Operationen (↑*read-modify-write*) erforderlich, um die Zustandsänderungen für einen Prozess herbeizuführen. Finden diese Operationen vor dem Hintergrund von ↑Parallelverarbeitung statt, ist deren ↑Unteilbarkeit gefordert. In diesem Fall bietet sich als Alternative die Repräsentation als *Zeichenfeld* an, wodurch die Zustandsänderungen auf Lese-/Schreiboperationen zurückgeführt werden können und allein durch ↑Speicherkohärenz der korrekte Ablauf sichergestellt ist. Nachfolgend werden beide Varianten kurz vorgestellt, wobei für beide dieselben Definitionen zugrunde gelegt werden:

```
typedef enum mood {                        /* scheduling states of a process */
    /* main states */
    READY    = 0,                            /* expects processor allocation */
    RUNNING  = 1,                                      /* owns the processor */
    BLOCKED  = 2,                                        /* expects an event */
    IDLE     = 3,                                    /* damned to inactivity */
    /* states of suspense */
    PENDING  = 4,                                     /* in state transition */
    BARRED   = 5,                                      /* runs to completion */
    /* following are commands, only, for state management */
    CHECK    = (1 << 29),                        /* read whole state variable */
    FLUSH    = (1 << 30),                      /* define whole state variable */
    CLEAR    = (1 << 31)             /* cancel selected state-variable flag */
} mood_t;

#define SUSPENSE ((1 << PENDING) | (1 << BARRED))
```

Dieser ↑Datentyp definiert sowohl vier *Zustandswerte* entsprechend eingangs beschriebener Standardsituationen eines Prozesses, zwei Werte für den *Schwebezustand* (`PENDING`, `BARRED`) als auch *Operationsmerker* zur Zustandsmanipulation. Mit den Operationsmerkern wird angezeigt, den aktuellen Zustand zum Prüfen (`CHECK`) zu lesen, ihn zu leeren und mit einem Zustandsmerker zu definieren (`FLUSH`) oder ihn um einen Zustandsmerker zu bereinigen (`CLEAR`). Ist bei der Operation auf dem Zustand kein Operationsmerker spezifiziert, wird der angegebene Zustandsmerker als Element in die Zustandsmenge aufgenommen. Die Verwendung dieser Merker wird weiter unten im Zusammenhang mit der funktionalen Abstraktion (`state`) zur Zustandsmanipulation deutlich.
Die vier Zustandswerte entsprechen entweder einer Bitposition in der Bitleiste oder einer Zeichenposition im Zeichenfeld, an der jeweils der Wahrheitswert zur Gültigkeit (*true*, 1) oder Ungültigkeit (*false*, 0) des betreffenden Zustands gespeichert ist. Im gegebenen Beispiel kodieren damit vier Bits beziehungsweise ↑Bytes den als *Menge* formulierten Zustand eines Prozesses, das Zeichenfeld hat damit vier Einträge (`NMOOD`) vom Typ `int8_t` und ist

so platzkompatibel zum Typ `int32_t`. Zur Kodierung dieser vier Positionswerte sind zwei Bits erforderlich, aus denen sodann mit einer Maskenoperation (`MOOD`) der Index für das Zeichenfeld bestimmt wird. Nachfolgend die dementsprechenden Deklarationen:

```
#define NMOOD sizeof(long)              /* number of states or state bytes */
#define MOOD(m) (m & (NMOOD - 1))      /* relevant bits per state variable */

typedef union state {                        /* scheduling-state attribute */
    int32_t unit;                             /* read-modify-write variant */
    int8_t quad[NMOOD];                              /* load/store variant */
} state_t;
```

Zur Verallgemeinerung der verschiedenen Operationsvarianten zur Zustandsverwaltung ist der Datentyp einer Zustandsvariablen als Vereinigung (*union*) ausgelegt, wodurch die beiden hier gezeigten Komponenten (`unit`, `quad`) dieselbe ↑Adresse zugewiesen bekommen. Entsprechend dieser beiden grundlegenden, auf die Datenstruktur bezogenen Varianten lassen sich verschiedene Funktionsvarianten bilden, wovon zwei davon weiter unten vorgestellt werden. Der Schwebezustand (`SUSPENSE`) ist eine Ergänzung zu den Hauptzuständen (insb. `READY`, `BLOCKED`, `RUNNING`) und wird daher durch eine Bitposition in der Bitleiste modelliert.
Nachfolgend werden nun zwei Funktionsvarianten zur Zustandsmanipulation vorgestellt, die erste nur zum Gebrauch auf einem Uniprozessor bei strikt synchronen Abläufen und die zweite für Uni- und Multiprozessor. Zunächst die erste Variante, die sich vor allem dadurch auszeichnet, dass die Zustandsmanipulationen durch komplexe Operationen (*read-modify-write*) bewerkstelligt werden:

```
inline int32_t state(state_t *this, mood_t mood) {   /* read-modify-write */
    if (!(mood & CHECK)) {
        if (mood & CLEAR)
            this->unit &= ~((1 << MOOD(mood)) | (mood & SUSPENSE));
        else if (mood & FLUSH)
            this->unit = (1 << MOOD(mood);
        else
            this->unit |= ((1 << MOOD(mood) | (mood & SUSPENSE));
    }
    return this->unit;
}
```

Diese Funktionsvariante operiert nur auf der Bitleiste, entsprechend der Operationsmerker löscht sie ein einzelnes Zustandsbit (`CLEAR`: logisches UND), definiert die komplette Zustandsvariable (`FLUSH`: Zuweisung) oder setzt ein einzelnes Zustandsbit (sonst: logisches ODER). Jede dieser Operation samt Zuweisung muss unteilbar durchgeführt werden, sollte ein Prozess im Verlauf einer Unterbrechungsbehandlung oder von einem anderen Prozessor ausgehend bereitgestellt werden können. Dies würde eine weitere Funktionsvariante ergeben, bei der, in Abhängigkeit vom ↑Befehlssatz der ↑CPU oder geeigneten Standardfunktionen des ↑Kompilierers, die betreffenden Anweisungen (`&=`, `|=`) durch atomare ϕ-*and-fetch*-Operationen, mit ϕ gleich `andl` beziehungsweise `orl`, ersetzt sind.
Zu beachten ist, dass der Kompilierer alle bedingten Anweisungen auflöst, da die Funktion (`state`) inzeilig (`inline`) übersetzt und durch ↑Konstantenpropagation schließlich die relevante Operation bereits zur Übersetzungszeit nicht nur festgelegt, sondern auch durchgeführt (d.h., der Ausdruck berechnet) wird. Um also hier die Anweisung des längsten Pfads im ↑Quellmodul (`|=`) in Aktion zu bringen, wird die CPU nicht alle Fälle überprüfen müssen, sondern kann stattdessen direkt den relevanten ↑Maschinenbefehl (`orl`) ausführen.
Zusätzlich zu dieser Hauptfunktion sind weitere Hilfsfunktionen sinnvoll, um für verschiedene Anwendungsfälle die Handhabung von Zustandsvariablen zu erleichtern. Nachfolgend zwei solcher Funktionen, die abhängig von der konkreten Vereinigungsvariante sind (hier für die Repräsentation als Bitleiste gezeigt):

```
inline int valid(state_t *this, mood_t mood) {   /* check state attribute */
    return this->unit == ((1 << MOOD(mood)) | (mood & SUSPENSE));
}
```

Die Funktion `valid` prüft, ob das spezifizierte Zustandsattribut (`mood`) in der referenzierten Zustandsvariablen (`this`) enthalten ist. Wohingegen die nächste Funktion (`draft`) einen Zustandswert, wie er in einer Zustandsvariablen gespeichert wäre, auf Grundlage des angegebenen Zustandsattributs zusammenstellt:

```
inline int32_t draft(mood_t mood) {                /* compose state value */
    return ((1 << MOOD(mood)) | (mood & SUSPENSE));
}
```

Beide Hilfsfunktionen haben jeweils eine Entsprechung für die Repräsentation der Zustandsvariablen als Zeichenfeld. Für diese, wie auch insbesondere die Hauptfunktion (`state`), gilt die oben erwähnte Optimierungsart der Konstantenpropagation ebenso. Das besondere Merkmal dieser zweiten Auslegungsvariante besteht jedoch darin, die Zustandsmanipulationen nur mit einfachen Lese-/Schreibbefehlen durchzuführen:

```
inline int32_t state(state_t *this, mood_t mood) {          /* load/store */
    if (!(mood & CHECK)) {
        if (mood & CLEAR)
            this->quad[MOOD(mood)] = 0;
        else if (mood & FLUSH)
            this->unit = (1 << (MOOD(mood) * 8));
        else
            this->quad[MOOD(mood)] = (1 | (mood & SUSPENSE));
    }
    return this->unit;
}
```

Das Zustandsattribut (`mood`) gibt hierbei den Indexwert für das Zeichenfeld, durch den der zu manipulierende Zustandsmerker (`quad[...]`) selektiert wird. Bei der `FLUSH`-Anweisung ist zudem die ↑Bytereihenfolge der CPU zu berücksichtigen und sicherzustellen, dass bei der Zuweisung an die komplette Zustandsvariable tatsächlich auch die dem Zeichenfeldeintrag entsprechende Byteposition definiert wird. Für ↑x86 (*little endian*) ist die hier gezeigte „Normierung“ durch Multiplikation des Indexwerts (d.h., Zustandsattribut `mood`) mit der Bitanzahl (`8`) pro Byte stimmig: es wird das jeweils i-te Byte in der Bitleiste (`unit`) oder dem Zeichenfeld (`quad`) auf den Wert Eins gesetzt.
Voraussetzung dafür, eben nicht auf Komplexbefehlen entsprechenden Anweisungen zurückgreifen zu müssen, die gegebenenfalls jeweils in atomarer Ausfertigung vorzuliegen haben, ist die Darstellung der Zustandsmenge als Zeichenfeld. Unter der Annahme von Speicherkohärenz ist damit insbesondere bei der möglichen asynchronen Bereitstellung eines Prozesses für die korrekten Zustandsübergänge gesorgt. Diese Variante empfiehlt sich bei einem ↑RISC, dessen Befehlssatz gemeinhin überhaupt keine komplexen arithmetisch-logischen Operationen anbietet, die direkt im ↑Hauptspeicher wirken. Aber auch bei einem ↑CISC (wie x86), der solche Operationen im Befehlssatz aufweist, kann die zweite hier diskutierte Funktionsvariante von Vorteil sein, da einfache Zuweisungsoperationen für gewöhnlich schneller ablaufen als komplexe Rechenoperationen.
Die Hilfsfunktionen in Bezug auf die Zustandsrepräsentation als Zeichenfeld, um einerseits auf das Enthaltensein eines Zustandsattributs hin zu prüfen (`valid`) und andererseits einen Zustandswert zusammenzustellen (`draft`), sind nachfolgend skizziert:

```
inline int valid(state_t *this, mood_t mood) {   /* check state attribute */
    return this->quad[MOOD(mood)] == (1 | (mood & SUSPENSE));
}
```

```
inline int32_t draft(mood_t mood) {                /* compose state value */
    return ((1 << (MOOD(mood) * 8)) | ((mood & SUSPENSE) << (MOOD(mood) * 8)));
}
```

In funktionaler Hinsicht entspricht die Bedeutung dieser Funktionen der der oben bereits erläuterten Varianten für die Repräsentation einer Zustandsvariablen als Bitleiste. Der Unterschied ist rein nichtfunktional und betrifft die Art und Weise der Abbildung der Mengenelemente des (logischen) Prozesszustands auf ein einzelnes ↑Speicherwort.
Abschließend noch eine Funktion, die einen bedingten Zustandsübergang als ↑atomare Operation durchführt und dies unabhängig von der jeweiligen Repräsentationsart der Zustandsvariablen erreicht:

```
inline int blend(state_t *this, mood_t mood) {   /* atomic state transfer */
    state_t tune = *this;
    return CAS(&this->unit, state(&tune, CLEAR | mood), draft(mood));
}
```

Demnach erfolgt der Zustandsübergang nur, wenn das gewünschte Zustandsattribut (`mood`) nicht bereits in der Zustandsvariablen (`this`) enthalten ist. Um dies auf Grundlage eines ↑CAS zu bewerkstelligen, wird in einer lokalen Kopie (`tune`) der Zustandsvariablen das zu setzende Zustandsattribut zunächst entfernt (`CLEAR`). Der um dieses Attribut reduzierte Wert der Kopie wird sodann mit dem aktuellen Wert der Zustandsvariablen verglichen. Bei Gleichheit, und zwar nur wenn auch in der Zustandsvariablen dieses Attribut nicht gesetzt ist, erfolgt der Zustandsübergang in Bezug auf dieses Attribut. Diese kombinierte Operation von Vergleich und bedingte Zuweisung wird letztlich allein von CAS geleistet, wobei die zur Aufnahme der Kopie dienende Hilfsvariable lediglich der Modellierung des für den Zustandsübergang vorausgesetzten Ausgangszustands dient. Liegt dieser Ausgangszustand nicht vor, scheitert CAS und damit scheitert das Vermengen (`blend`) der Zustandsattribute.

pseudo file (dt.) ↑Pseudodatei.

pseudo instruction (dt.) ↑Pseudobefehl.

pseudo parallelism (dt.) ↑Pseudoparallelität.

Pseudobefehl (en.) ↑*pseudo instruction*. Anweisung an einen ↑Assemblierer. Beispielsweise zur Angabe des Programmabschnitts (↑Text, ↑Daten, ↑BSS), auf die sich die nachfolgenden Anweisungen beziehen, um den ↑Adresszähler des aktuellen Segments auf eine bestimmte Grenze auszurichten, einem ↑Symbol einen Wert zuzuweisen oder Informationen für den ↑Binder aufzubereiten.

Pseudodatei (en.) ↑*pseudo file*. Bezeichnung für eine ↑Datei, deren eigentliche Bedeutung lediglich nachgeahmt ist, um im ↑Betriebssystem verfügbare ↑Daten oder ↑Betriebsmittel auch einem ↑Prozess im ↑Maschinenprogramm kontrolliert zugänglich machen zu können. Im Falle von ↑UNIX-artigen Betriebssystemen ist das beispielsweise die ↑Gerätedatei oder `proc(5)`.

Pseudoparallelität (en.) ↑*pseudo parallelism*. Unechte ↑Parallelität, die nicht durch Verfielfachung, sondern durch Mehrfachnutzung (↑*time sharing*) des ↑Prozessors zustande kommt.

PSW Abkürzung für (en.) ↑*processor status word* oder ↑*program status word*.

Puffer (en.) ↑*buffer*. Ursprünglich die Bezeichnung für eine federnde Vorrichtung an Vorder- und Rückseite eines Schienenfahrzeugs zum Auffangen von Stößen (Duden). Im übertragenen Sinne ein ↑wiederverwendbares Betriebsmittel (Vorrichtung) zum Ausgleich der durch die unterschiedlichen Geschwindigkeiten zweier oder mehrerer ↑Prozesse (Schienenfahrzeuge) hervorgerufenen Wirkungskräfte (Auffangen von Stößen).
Entgegen der landläufigen Nutzung dieses Mittels zur Zwischenspeicherung von ↑Daten, steht aber eben dieser Aspekt der Aufbewahrung von Informationen hier gerade nicht im Vordergrund: der ↑Speicherbereich ist lediglich Mittel zum Zweck, nämlich eine sich auf

einen bestimmten Prozess auswirkende ↑Aktionsfolge eines anderen Prozesses derart zu puffern, dass beide keine oder nur geringe Verzögerung erfahren. Typischerweise kooperieren die betreffenden Prozesse in der Durchführung einer Berechnung. Dabei markieren bestimmte Zwischenschritte jeweils ein ↑Ereignis, das eine kausale Abhängigkeit zwischen den Prozessen definiert. Jedes dieser Ereignisse ist von dem einen Prozess (Produzent) zu dem anderen Prozess (Konsument) zu kommunizieren und wird dazu in Form bestimmter Daten (↑Signal, ↑Nachricht), die jeweils ein ↑konsumierbares Betriebsmittel repräsentieren, kodiert und zur Bearbeitung bereitgestellt (↑*producer-consumer problem*).
Eine derartige Vorrichtung ist nicht nur in Software gebräuchlich, um Geschwindigkeitsunterschiede bei kooperierenden Prozesses auszugleichen. Typisches Beispiel für die Hardware ist der ↑Schreibpuffer, um zwei mit unterschiedlichen Geschwindigkeiten arbeitenden und einander angrenzenden (unteren) Stufen in einer ↑Speicherhierarchie zu koppeln.

Puffern (en.) ↑*buffering*. Konzept einer Methode, die Auswirkung eines Vorgangs auf einen Gegenstand oder anderen Vorgang mildern, abschwächen zu können (in Anlehnung an den Duden). Ursprüngliche Bedeutung ist die Entkopplung von (externen/internen) ↑Prozessen unterschiedlicher Geschwindigkeiten. Diese Entkopplung wird erreicht, indem der eine Prozess ↑Daten, die ein bestimmtes ↑Ereignis repräsentieren, zur späteren Verarbeitung durch den anderen Prozess zwischenspeichert. Der dafür benötigte ↑Speicherbereich wird als ↑Puffer bezeichnet. Dabei steht die Speicherung selbst nicht im Vordergrund, sie ist lediglich Mittel zum Zweck für den Ausgleich verschiedener Wirkungskräfte.

punched paper tape (dt.) ↑Lochstreifen.

Quellmodul (en.) ↑*source module*. ↑Datei mit einem ↑Programm als Eingabe für einen ↑Übersetzer. Gleichfalls aber auch Ausgabedatei eines Instruments zur Programmerzeugung, das heißt, eines Editors, Übersetzers (insb. ein ↑Kompilierer) oder Generators.

Querschnittsbelang (en.) ↑*cross-cutting concern*. Bezeichnung für einen bedeutsamen Aspekt in der Software, der erst wie bei einem in Querrichtung durch einen Körper geführten Schnitt sichtbar werden würde (in Anlehnung an den Duden). Ein in einem ↑Programm beschriebenes charakteristisches Merkmal, das nicht an einer Stelle der Software konzentriert in Erscheinung tritt, sondern mehrfach in identischer oder leicht abgewandelter Form vorhanden ist. Dieses Merkmal kann mangels geeigneter Modularisierungskonzepte in der gewählten Programmiersprache nicht einfach als einzelnes Modul separat dargestellt werden. Meist bringt ein solches Merkmal eine bestimmte nichtfunktionale Eigenschaft zum Ausdruck.
Typisches Beispiel für solch einen Belang sind Programmstellen, an denen unter bestimmten Umständen für die zusätzliche ↑Synchronisierung von ↑Prozessen zu sorgen ist. Aber auch die Zielmenge für reguläre Rückgabewerte der durch ↑Systemaufrufe aktivierten ↑Systemfunktionen fällt darunter, die nämlich in Abhängigkeit von der Art und Weise der Auslegung eines ↑Systemaufrufbefehls definiert ist (s. insb. ↑Primitivbefehl).

queue (dt.) ↑Schlange, ↑Warteschlange.

queued lock siehe ↑*queued spinlock*.

queued spinlock (dt.) ↑gereihte Umlaufsperre.

race condition (dt.) ↑Wettlaufsituation.

race hazard (dt.) Laufgefahr, ↑Wettlaufsituation.

RAM Abkürzung für (en.) ↑*random access memory*.

random access memory (dt.) ↑Direktzugriffsspeicher.

random device siehe ↑*random generator*.

random generator (dt.) ↑Zufallsgenerator.

Rang (en.) ↑*rank*, *tier*. Bestimmte Stufe, Stellung, die ein ↑Prozess in einem hierarchisch gegliederten Verfahren zur ↑Planung der ↑Einlastung eines ↑Prozessors innehat; Stellenwert im Vergleich zu anderen Prozessen (in Anlehnung an den Duden).

rank (dt.) ↑Rang.

re-entrance (dt.) ↑Wiedereintritt.

re-entrant (dt.) ablaufinvariant, eintrittsinvariant; fähig zum ↑Wiedereintritt.

re-entry lock (dt.) ↑Wiedereintrittssperre.

read (dt.) ↑lesen.

read-modify-write Bezeichnung für einen aus drei Teilschritten bestehenden ↑Befehlszyklus bei der Ausführung von einem komplexen ↑Maschinenbefehl: erstens, ein ↑Speicherwort auslesen und in den ↑Prozessor laden; zweitens, den geladenen Wert im Prozessor verändern; drittens, den geänderten Wert in dasselbe Speicherwort zurückschreiben. Der Maschinenbefehl beschreibt eine ↑Aktionsfolge, die zwar eine logisch zusammenhängende Einheit bildet, zwangsweise jedoch keine ↑atomare Operation darstellt. Das bedeutet, nach jedem Teilschritt kann ein anderer Prozessor denselben Maschinenbefehl angewendet auf dasselbe ↑Maschinenwort ausführen. Folge davon ist eine möglicherweise inkonsistente Berechnung.

read-only memory (dt.) ↑Festwertspeicher. Abkürzung ↑ROM.

read/write lock (dt.) ↑Lese-/Schreibsperre.

reader (dt.) ↑Leser.

readers-writers problem (dt.) ↑Leser-Schreiber-Problem.

ready list (dt.) ↑Bereitliste.

ready time (dt.) ↑Bereitzeit.

real time (dt.) ↑Echtzeit.

real-time axis (dt.) ↑Echtzeitachse.

real-time clock (dt.) ↑Echtzeituhr.

real-time computing system (dt.) ↑Echtzeitrechensystem.

real-time condition (dt.) ↑Echtzeitbedingung.

real-time data processing siehe (en.) ↑*process data processing*.

real-time mode (dt.) ↑Echtzeitbetrieb.

real-time process (dt.) ↑Echtzeitprozess.

real-time system (dt.) ↑Echtzeitsystem.

reale Adresse (en.) ↑*physical address*. Bezeichnung für eine ↑Adresse, deren Definitionsbereich ein ↑realer Adressraum $\mathcal{R} = [0, 2^n - 1]$ ist, mit n gleich der ↑Adressbreite des zugrunde liegenden ↑Prozessors. Eine solche Adresse ermöglicht der ↑CPU den Zugriff auf ein bestimmtes ↑Speicherwort im ↑Hauptspeicher oder auf ein spezielles ↑Ein-/Ausgaberegister von einem ↑Peripheriegerät (↑*memory-mapped* I/O) über den ↑Datenbus.

realer Adressraum (en.) ↑*physical address space*. Bezeichnung für den wirklichen (physischen) ↑Adressraum, der durch die reale Maschine (Hardware) definiert ist. Dieser Adressraum ist nicht zwingend linear aufgebaut, auch wenn die ↑CPU entsprechend ihrer ↑Adressbreite n und dem daraus resultierenden ↑Adressbereich $\mathcal{A} = [0, 2^n - 1]$ die Generierung einer beliebigen ↑Adresse in $\mathcal{A}$ und damit in dem gesamten Adressraum zulässt. Bestimmte Gebiete in diesem Adressraum können undefiniert sein, Lücken bilden. Konsequenz daraus ist, dass nicht jede von der CPU applizierte Adresse auch gültig ist, das heißt, einen Adressaten (↑Speicherwort, ↑Peripheriegerät) hat. Die Verwendung einer solchen ungültigen Adresse in einem ↑Prozess ist für gewöhnlich ein schwerwiegender Fehler (↑*bus error*). Welche Gebiete in $\mathcal{A}$ gültig beziehungsweise ungültig sind, bestimmt der ↑Adressraumbelegungsplan.
Damit ein Prozess in einem solchen Adressraum nicht scheitert, muss sein ↑Programm auf die Adressbereiche mit gültigen Adressen abgebildet worden sein. Die Abbildung ist spätestens zur ↑Bindezeit zu leisten. Verfügt die CPU über eine ↑MPU, steckt das ↑Betriebssystem dazu obere und untere Grenzen für das Programm ab, so dass der zugehörige Prozess nur für ihn gültige Adressen generieren und so aus sein ↑Text-, ↑Daten- und ↑Stapelsegment nicht ausbrechen kann. Gleichwohl muss jedes einzelne ↑Segment komplett und zusammenhängend im ↑Hauptspeicher vorliegen. Darüberhinaus ist die durch die ↑Ladeadresse vorgegebene Lokalität jedes dieser Segmente zur ↑Laufzeit des Programms unveränderlich.

Rechenanlage (en.) ↑*computer system*. Gesamtheit der ein ↑Rechensystem konstituierenden Hardware (↑Rechner und ↑Peripherie). Eine durch (wenigstens) ein ↑Programm gesteuerte, ↑Daten verarbeitende Anlage, deren Ausmaß und ↑Betriebsart durch den jeweiligen Einsatzzweck bestimmt ist.

Rechenkern (en.) ↑*core*. Recheneinheit einer ↑CPU, auf der ein ↑Prozess stattfinden kann.

Rechenstoß (en.) ↑CPU *burst*. Bezeichnung für eine ↑Häufung von Aktivität, die in engem Bezug mit der ↑CPU steht und vor allem der Verrichtung festgelegter Berechnungen dient. Ein solches Aktivitätsbündel umfasst alle notwendigen ↑Aktionen zur Durchführung bestimmter arithmetisch-logischer Operationen wie auch zum dazu erforderlichen Transfer von ↑Daten (d.h., Operanden) vom/zum ↑Hauptspeicher.
In dieser auf Rechenaktivitäten fokussierten Phase führt die CPU für einen ↑Prozess, gemäß seines ↑Programms, einen ↑Maschinenbefehl nach dem anderen aus: sie verarbeitet einen Stoß von Maschinenbefehlen bezogen auf einen bestimmten Prozess. Der betreffende Prozess ist laufend (↑Prozesszustand).
Für Anfang und Ende von einem solchen „Ausführungsbündel“ gibt es eine physische und eine logische Sicht. Die physische (einfachere) Sicht nimmt die Position der realen Maschine ein: der Operationsstoß beginnt mit jedem Neustart (*reset*) oder Erwachen aus dem ↑Schlafzustand der CPU und endet mit jedem Eintritt in diesen. Demgegenüber gestaltet sich die logische Sicht, die eine ↑abstrakte Maschine in Form von dem ↑Betriebssystem als Grundlage nimmt, etwas schwieriger. Letztlich führt hier die Differenzierung zwischen Prozess und ↑Prozessinkarnation, beziehungsweise der jeweilige ↑Arbeitsmodus der CPU, genau genommen zu verschiedenen Arten und damit auch Start- und Endpunkten für einen derartigen Stoß (nachfolgend im Uhrzeigersinn verdeutlicht).
Demzufolge nimmt ein Prozess seinen Operationsstoß beim physischen ↑Prozesswechsel auf, also wenn von einer ↑Prozessinkarnation zu einer anderen umgeschaltet wird (00:00 Uhr). Im Moment dieser Umschaltung (↑*dispatching*) handelt die CPU für das Betriebssystem (↑*privileged mode* bzw. ↑*system mode*). Dieser Stoß setzt sich fort bis zum Verlassen des Betriebssystems und zur erstmaligen oder wiederholten Aufnahme der Ausführung von dem ↑Maschinenprogramm, für das die Prozessinkarnation eingerichtet wurde. Im Moment von dem damit verbundenen ↑Moduswechsel wird die CPU ihre Handlung für das Betriebssystem beenden und für das Maschinenprogramm (↑*unprivileged mode* bzw. ↑*user mode*) beginnen (06:00 Uhr). Die Prozessinkarnation vollzieht mit dieser Umschaltung letztlich einen logischen Prozesswechsel, da durch Aufnahme der Ausführung des Maschinenprogramms die Entwicklung einer neuen Folge von Maschinenbefehlen eines anderen Programms und damit

auch in einem anderen ↑Adressraum beginnt. Diese Entwicklung setzt sich bis zur Unterbrechung des Prozesses fort, nämlich wenn das Maschinenprogramm durch eine ↑synchrone Ausnahme oder ↑asynchrone Ausnahme verlassen und das Betriebssystem wieder betreten wird. Im Moment des damit verbundenen erneuten Moduswechsels wird die CPU ihre Handlung für das Maschinenprogramm beenden und für das Betriebssystem beginnen (18:00 Uhr). Erneut kommt es zum logischen Prozesswechsel, da durch Aktivierung des Betriebssystems die Entwicklung einer neuen Folge von Maschinenbefehlen eines anderen Programms in einem anderen Adressraum beginnt. Dieser neue Operationsstoß entwickelt sich weiter bis zum (a) Verlassen des Betriebssystems und zur Rückkehr zum Maschinenprogamm (06:00) oder (b) physischen Prozesswechsel, nämlich wenn die zugehörige Prozessinkarnation weggeschaltet wird (24:00 Uhr).
Dieser Kreislauf zeigt eine große Häufung von Aktivität für die Prozessinkarnation, die jedoch aus wenigstens drei kleineren Stößen für die logisch verschiedenen Prozesse dieser Inkarnation zusammengesetzt ist: letztere bilden Stoßabschnitte, die verschiedenen Phasen einer Prozessinkarnation entsprechen. Bestimmte Abschnitte (18:00–06:00 ohne und 00:00–06:00 sowie 18:00–24:00 mit Wechsel der Prozessinkarnation) sind dem Betriebssystem, ein anderer Abschnitt (06:00–18:00) ist ein- oder mehrfach dem Maschinenprogramm zuzurechnen. Dies gilt grundsätzlich für jede Prozessinkarnation, die durch ein Betriebssystem zur Ausführung eines Maschinenprogramms bereitgestellt wird. Wissen über die jeweilige ↑Stoßzeit gibt Aufschluss über die von einer Prozessinkarnation selbst verrichteten und anderen übertragenen Arbeit, letzteres insbesondere in Bezug auf den ↑Ein-/Ausgabestoß, den der Prozess während seiner Verweildauer im Betriebssystem ein- oder mehrfach auslösen kann. Die Daten dienen allgemein Abrechnungszwecken (↑*accounting*) und liefern Hinweise über den Ausnutzungsgrad von einem ↑Rechensystem (z.B. ↑UNIX: `getrusage(2)`).

Rechensystem (en.) ↑*data processing system*. Einheit aus technischen Anlagen, Bauelementen und Programmen zur Verarbeitung von ↑Daten und Durchführung von Berechnungen. Zentrales Konzept ist der ↑Rechner, der allein oder im Verbund vernetzt mit anderen (gleichen/ungleichen) Einheiten seiner Art eine bestimmte Funktion für einen bestimmten Anwendungsfall ausübt. Dieser Anwendungsfall kann in besonderem Maße auf einen ganz bestimmten Zusammenhang ausgerichtet sein oder verschiedenste Bereiche umfassen, das heißt, einerseits ein Spezialsystem (*special-purpose system*) oder anderseits ein Universalsystem (*general-purpose system*) bedingen.

Rechenzeit (en.) ↑CPU *time*. Zeit, die für die Operationen eines ↑Rechensystems zur Bewältigung einer ↑Aufgabe benötigt wird (in Anlehnung an den Duden). Die zur ↑Laufzeit von einem ↑Programm beanspruchte Zeit zur Durchführung von Berechnungen.
Grundlage bilden die ↑Stoßzeiten eines ↑Prozesses. Dann entspricht die von dem Prozess benötigte Zeit für seine (bisherigen) Berechnungen der Summe all seiner Stosszeiten:

$$t_{cpu} = \sum_{n=0}^{N-1} t_{burst}^{n},$$

mit t_{burst} der für einen bestimmten ↑Rechenstoß n benötigten Zeit. Die Summe dieser Einzelzeiten ergibt die (bisher) von dem Prozess im ↑Prozesszustand laufend auf der ↑CPU verbrachten Gesamtzeit.

Rechner (en.) ↑*computer*. Programmgesteuerte, elektronische Rechenanlage.

Rechnerarchitektur ↑Architektur von einem ↑Rechner oder ↑Rechensystem. Diese ist bestimmt durch ein ↑Operationsprinzip für die Hardware und einer Struktur gegeben durch Art/Anzahl der ↑Betriebsmittel (Hardware) und die sie verbindenden Kommunikationseinrichtungen, einschließlich der dafür definierten Kommunikations- und Kooperationsregeln.

Rechnernetz Zusammenschluss mehrerer voneinander unabhängiger ↑Rechner über ein ↑Netzwerk, der die Kommunikation der zusammengeschlossenen Einheiten ermöglicht. Dabei folgt

die Kommunikation zwischen den verschiedenen Rechnern einem bestimmten Protokoll, das für gewöhnlich wenigstens paarweise einheitlich ist. Im gesamten Netz können mehrere Protokolle möglich und zugleich aktiv sein (z.B. ↑UDP, ↑TCP und ↑IP). Die Struktur der Verbindungen (*Topologie*) der Rechner kann sehr unterschiedlich ausgeprägt sein. Typische Formen sind Linie, Ring, Stern, Baum, Bus, (teil-) vermascht und vollvermascht (jeder Rechner mit jedem anderen). Auch hier können im Netz mehrere Formen zugleich ausgeprägt sein.

Rechtzeitigkeit (en.) ↑*timeliness*. Fähigkeit eines ↑Rechensystems, eine ihm aufgetragene ↑Aufgabe innerhalb einer bestimmten Zeitspanne oder bis zu einem vorgegebenen Zeitpunkt zu vollenden und damit ↑Termineinhaltung zuzusichern.

reconfiguration (dt.) ↑Rekonfigurierung.

reentrancy (dt.) ↑Eintrittsinvarianz, ↑Ablaufinvarianz.

reference (dt.) ↑Referenz.

reference bit (dt.) ↑Referenzbit.

reference string (dt.) ↑Referenzfolge.

Referenz (en.) ↑*reference*. Eine ↑Speicherstelle, auf die verwiesen wird, weil sie Auskunft über etwas geben kann (in Anlehnung an den Duden). Der Bezug auf einen bestimmten, im ↑Speicher vorrätigen Bestand von ↑Text oder ↑Daten (dem Speicherinhalt an der Speicherstellle). In technischer Hinsicht für gewöhnlich verkörpert durch einen ↑Zeiger, der jedoch niemals ungültig ist, das heißt, immer auf eine gültige Speicherstelle verweist.

Referenzbit (en.) ↑*reference bit*. ↑Speichermarke im ↑Seitendeskriptor, die von der ↑MMU beim Zugriff (ausführen, lesen, schreiben) auf die betreffende ↑Seite gesetzt wird.

Referenzfolge (en.) ↑*reference string*. Reihe von zeitlich aufeinanderfolgenden, von einem ↑Prozess generierte Referenzen auf den ↑Arbeitsspeicher. Der Verlauf dieser Reihe ist bestimmt durch die Struktur von dem ↑Programm, das den Prozess festlegt und der ausgeführten Funktion in Abhängigkeit von den Eingabedaten. Jede Referenz entspricht einer ↑Adresse.

Regelkreis (en.) ↑*control loop*. Wirkungsablauf, der die Beeinflussung einer physikalischen Größe (Regelgröße) in einem bestimmten technischen Prozess erreicht und dabei Abweichungen von einem vorgegebenen Sollwert (Führungsgröße) kontinuierlich und für gewöhnlich in Schritten (Stellgröße) entgegenwirkt; ein sich selbst regulierendes geschlossenes System (Duden). Dabei bildet ein technischer Prozess die „Gesamtheit von aufeinander einwirkenden Vorgängen in einem System, durch die Materie, Energie und Information umgeformt, transportiert und gespeichert wird“ (DIN IEC 60050-351). Typische Beispiele für solche Prozesse finden sich in der Fertigungs-, Verfahrens- oder Mess- und Prüftechnik, bei der technische Anlagen und Maschinen (d.h., technische Arbeitsmittel) durch ein ↑Rechensystem bedient, gesteuert und überwacht werden.
Ein solcher Kreislauf zur ↑Regelung des technischen Prozesses muss stabil sein, soll ein gutes Führungs- und Störverhalten aufweisen und soll robust sein. Der Kreislauf ist stabil, wenn die Regelung keine Dauerschwingungen oder aufklingende Schwingungen erzeugt. Das Führungsverhalten beschreibt die Auswirkung von Aufschaltungen der (positiven oder negativen) Führungsgröße auf die Regelgröße. Dem wirken für gewöhnlich Störgrößen aus der Umgebung der Regelstrecke entgegen (Störverhalten), die so auszuregeln sind, dass die Regelgröße wieder den Wert der Führungsgröße annimmt. Der Kreislauf ist robust, wenn der durch Dauerbelastung und einhergehender Materialermüdung hervorgerufene Einfluss von Parameteränderungen auf Regler und Regelstrecke in vorgegebenen Toleranzbereichen bleibt.

Regelung (en.) ↑*feedback control*. Vorgang in einem ↑Regelkreis, bei dem durch ständige Kontrolle und Korrektur eine physikalische, technische oder ähnliche Größe auf einem konstanten Wert gehalten wird (Duden). Dazu ist der Istwert der zu beeinflussenden Größe, als *Regelgröße* bezeichnet, stetig zu messen und mit einem Sollwert, die sogenannte *Führungsgröße*, zu vergleichen. Die Messung erfolgt direkt in oder an der *Regelstrecke*, das heißt, dem die zu beeinflussende Größe enthaltenden Teil des Regelkreises. Beispiele für eine Regelstrecke sind Stoffströme (d.h., Gas- oder Flüssigkeitsströme, Papier-, Kunststoff- oder Metallbahnen), Wärmeerzeugungs- und Kühlungsanlagen, Wärmetauscher, chemische Reaktoren, Kernreaktoren, Fahr- oder Flugzeuge, elektrische Maschinen und nachrichtentechnische Geräte (d.h., Rundfunkgeräte) — aber auch eine ↑CPU: nämlich um Überhitzung zu vermeiden und Durchschmelzen vorzubeugen (↑*dark silicon*). Abweichungen zwischen Regel- und Führungsgröße ergeben jeweils eine bestimmte *Regeldifferenz*, aus der ein *Regler* eine *Stellgröße* berechnet. Letztere wirkt stellend (erhöhen, verringern) auf die Regelgröße ein, und zwar derart, dass trotz (unbekannter) *Störgrößen* die Abweichung verringert und schließlich minimal gehalten wird Eine solche Rückkopplung fehlt bei der ↑Steuerung. Der gesamte Vorgang ist zyklisch und unterliegt Zeitvorgaben, die durch die physikalischen Eigenschaften der Regelstrecke definiert sind und eine bestimmte ↑Echtzeitbedingung manifestieren.

Register (en.) ↑*register*. Behältnis in einem ↑Prozessor oder in einem ↑Peripheriegerät, nämlich als ↑Speicher oder Übertragungsmedium von ↑Daten: daher auch differenziert in ↑Prozessorregister und ↑Ein-/Ausgaberegister. Ohne weitere Qualifizierung ist erstere Art implizit gemeint, eine ↑Speicherzelle innerhalb der ↑CPU für ein ↑Maschinenwort.

register (dt.) ↑Register, Zählwerk.

register allocation (dt.) ↑Registerzuteilung.

register set (dt.) ↑Registersatz.

Registersatz (en.) ↑*register set*. Menge aller ↑Prozessorregister. Typischerweise differenziert nach folgenden Arten: Datenregister, zur Speicherung von Operanden und Rechenergebnissen; Adressregister, zur Adressierung von Operanden und ↑Maschinenbefehlen; Spezialregister, zur Betriebsstandanzeige (↑Statusregister) oder Adressierung bestimmter ↑Speicherbereiche (↑Segmentregister). Neben den dem ↑Programmiermodell zugerechneten ↑Registern umfasst die Menge auch interne Register, auf die von einem ↑Programm heraus nicht zugegriffen werden kann.

Registerspeicher Bezeichnung von ↑Speicher in der ↑CPU mit sehr geringer Kapazität (2 bis 1024 ↑Prozessorregister) und extrem kurzer ↑Zugriffszeit für die Zwischenspeicherung von ↑Daten. Die Gesamtheit aller Prozessorregister bildet den ↑Registersatz.

Registerzuteilung (en.) ↑*register allocation*. Optimierungsvorgang bei der ↑Kompilation, der einem später beim ↑Programmablauf relevanten ↑Platzhalter (zusätzlich) ein ↑Prozessorregister in der ↑Speicherhierarchie zuordnet. Relevant ist ein Platzhalter, wenn er häufig verwendet wird und durch die Zuordnung zu ↑Registerspeicher eine kurze ↑Ausführungszeit zu erwarten ist. Für die (Zwischen-) Speicherung von Werten wird dann in dem betreffenden Programmabschnitt (insb. ↑Grundblock, ↑Unterprogramm) der ursprünglich dem ↑Arbeitsspeicher zugeordnete Platzhalter nicht genutzt.
Für gewöhnlich ist die durch das Programm bestimmte Platzhalteranzahl für die berechneten Werte größer als die durch das ↑Programmiermodell definierte Anzahl von Prozessorregistern. Demzufolge muss der ↑Kompilierer eine Platzhalterauswahl treffen, um damit dann die beste Entscheidung in Bezug auf die verfügbaren ↑Register zu treffen. Für den Programmablauf benötigte Platzhalter, denen keine Register zugewiesen werden können, verbleiben dann im Arbeitsspeicher. Aufgrund der gemeinhin geringen Registeranzahl kann die Auswahlentscheidung in Bezug auf verschiedene Abschnitte eines zu kompilierenden Programms sodann die Wiederverwendung desselben Registers für einen anderen Platzhalter zur Folge haben.

In solch einer Situation generiert der Kompilierer zusätzlich ↑Maschinenbefehle zum Sichern und Wiederherstellen des Inhalts des Registers, das für den auszutauschenden Platzhalter genutzt wird. Für diese Sicherungsmaßnahmen wird ein ↑Laufzeitstapel genutzt.

Rekonfigurierung (en.) ↑*reconfiguration*. Eine Maschine (ggf. durch Ergänzung von Modulen) an neue Anforderungen anpassen, ein (ggf. defektes) System neu oder anders zusammenstellen, um die Funktionsfähigkeit zu erhalten, wiederherzustellen oder den geänderten Bedürfnissen oder Bedingungen zu entsprechen (in Anlehnung an den Duden).

Rekursion Bezeichnung für den Vorgang in einem ↑Programmablauf, wenn ein ↑Unterprogramm in seiner Definition selbst nochmals aufgerufen wird. Dieser rekursive Aufruf kann direkt in dem betreffenden Unterprogramm oder indirekt in einem anderen Unterprogramm kodiert sein, wobei letzteres auf einem Aufrufpfad heraus aus ersterem Unterprogramm erreichbar ist. Indirekt rekursive Aufrufe sind *wechselseitig rekursiv*, da die betreffenden Unterprogramme durch (direkte oder indirekte) wechselseitige Verwendung voneinander definiert sind. Ein wichtiger Aspekt dabei ist die Existenz einer Abbruchbedingung für die Entfaltung der rekursiven Definition eines Unterprogramms. Diese Bedingung muss sicherstellen, dass ein rekursiver Aufruf des Unterprogramms (spätestens zur ↑Laufzeit) in endlich vielen Schritten aufgelöst werden kann.
In ↑Betriebssystemen ist diese Art des Aufrufs von Unterprogrammen eher selten anzufinden und wenn dort überhaupt, dann nur mit größter Vorsicht zu verwenden. Grund dafür ist die für gewöhnlich starke und gegebenenfalls durch statische Programmanalyse auch nicht mehr bestimmbare maximale Ausdehnung des ↑Laufzeitstapels eines ↑Prozesses, wenn ein solcher Aufruf durch den ↑Kompilierer nicht in eine herkömmliche Programmschleife umgewandelt werden kann. Die Ausdehnung des Laufzeitstapels über seine Grenze hinweg löst für gewöhnlich ↑Panik aus. Beispiele für diese Aufrufart finden sich unter anderem bei der ↑Unterbrechungsbehandlung (insb. ↑FLIH).

relative address (dt.) ↑relative Adresse.

relative Adresse (en.) ↑*relative address*. Bezeichnung für eine ↑Adresse, die relativ zu einem bestimmten ↑Bezugspunkt steht. Solche Adressen sind positionsunabhängige ↑Referenzen.

release consistency (dt.) ↑Freigabekonsistenz.

release time (dt.) ↑Auslösezeit, ↑Bereitzeit.

relocating loader (dt.) ↑verschiebender Lader.

relocation (dt.) ↑Verlagerung.

relocation constant (dt.) ↑Verlagerungskonstante.

relocation table (dt.) ↑Verlagerungstabelle.

remote operation (dt.) ↑abgesetzter Betrieb.

remote procedure call (dt.) ↑Prozedurfernaufruf, Abkürzung ↑RPC.

removable medium (dt.) ↑Wechseldatenträger.

replacement algorithm (dt.) ↑Ersetzungsalgorithmus.

replacement policy (dt.) ↑Ersetzungsstrategie.

rerun (dt.) ↑Wiederholungslauf.

rerun bit Steuerungsbit im ↑Prozessorstatuswort, Schaltvariable als Ausprägung eines booleschen Datentyps: im Falle von `true` wird die ↑CPU angewiesen, einen ↑Wiederholungslauf durchzuführen; anderenfalls (`false`) ruft die CPU ganz normal den nächsten ↑Maschinenbefehl in Folge ab und beginnt mit seiner Ausführung. Typischerweise deutet die CPU dieses Bit im Moment der Wiederaufnahme von einem zuvor unterbrochenen ↑Prozess, nämlich am Ende vom zugehörigen ↑Unterbrechungszyklus.

rescheduling (dt.) ↑Umplanung.

resident monitor (dt.) ↑residentes Steuerprogramm.

resident set (dt.) ↑Residenzmenge.

residentes Steuerprogramm (en.) ↑*resident monitor*. Bezeichnung für ein ↑Programm, das den Ablauf anderer Programme organisiert und überwacht (Duden). Der im ↑Hauptspeicher residente, ein „embryonales ↑Betriebssystem“ bildende Teil einer ↑Systemsoftware. Dieses Programm wird für gewöhnlich durch ein ↑Urladeprogramm selbst in den Hauptspeicher gebracht, wenn der betreffende ↑Rechner hochfährt, und verbleibt dort solange, bis dieser Rechner herunterfährt. Die Eigenschaft, resident zu sein, besteht daher lediglich im besonderen Verhältnis zum ↑Maschinenprogramm, das nämlich eins nach dem anderen (mit für gewöhnlich nicht immer derselben Funktion) das ↑Rechensystem durchläuft und damit die eigentliche transiente Komponente im System darstellt.

Die Konsequenz aus dieser Eigenschaft ist, dass sowohl für eine logische als auch physische Entkopplung von Maschinen- und Steuerprogramm gesorgt werden muss: um das Maschinenprogramm nachträglich laden zu können, ist es logisch vom Steuerprogramm zu entkoppeln; um das Steuerprogramm für die gesamte Betriebsdauer unbeschadet zu belassen, ist es physisch vom Maschinenprogramm zu entkoppeln. Beide Arten stellen für sich eigenständige Programme dar, die jeweils mit wohldefinierten Funktionen versehen sind.

Logische Entkopplung meint, dass das ↑Binden dieser Programm weder räumlich noch zeitlich zusammenhängend geschieht. Damit ist allerdings dem ↑Binder die ↑absolute Adresse von einem im Maschinenprogramm benutzten ↑Unterprogramm des Steuerprogramms für gewöhnlich nicht bekannt: die ↑symbolische Adresse dieses Unterprogramms kann nicht aufgelöst werden, womit die für ein ausführbares Maschinenprogramm erforderliche ↑Bindung nicht möglich ist. Um zwar logisch entkoppelt dennoch Funktionen des Steuerprogramms in Anspruch nehmen zu können, werden die Adressen all dieser „exportierten Funktionen“ in eine Sprungtabelle eingetragen. Dies geschieht, wenn das Steuerprogramm gebunden wird oder es sich initialisiert. Die Adresse dieser Tabelle ist dem Binder bekannt oder er hinterlässt beim Binden den Hinweis, an welcher Adresse sie im Hauptspeicher vom Steuerprogramm zu platzieren ist. Darüber hinaus erhält jede dieser exportierten Funktion eine Nummer, die den Eintrag der zugehörigen Funktionsadresse in der Tabelle identifiziert. Der Aufruf einer Funktion des Steuerprogramms erfolgt sodann indirekt über die Sprungtabelle. Dieses Verfahren entspricht in den wesentlichen Grundzügen der programmiersprachlichen Auslegung von einem ↑Systemaufruf.

Demgegenüber meint physische Entkopplung vor allem ↑Speicherschutz für das Steuerprogramm (↑*fence register*, ↑*bounds register* oder ↑*base/limit register*), um eben dafür zu sorgen, dass fehlerhafte Maschinenprogramme schlimmstenfalls immer nur lokale Auswirkungen auf sich selbst, das heißt, auf ↑Text und ↑Daten in ihrem jeweils eigenen ↑Adressraum haben. Jedes Maschinenprogramm kommt strikt räumlich isoliert vom Steuerprogramm zur Ausführung. Damit ist aber ein Aufruf der Steuerprogrammfunktionen über die Sprungtabelle nicht mehr in der Form möglich, wie die logische Entkopplung des Maschinenprogramms es vorsah: die Tabelle liegt logisch genau zwischen zwei Programmen, die nunmehr aber durch Speicherschutz physisch voneinander getrennt sind. Da Hardware die Grenze zwischen Maschinen- und Steuerprogramm sichert, ist ein spezieller ↑Maschinenbefehl für den Aufruf von Steuerfunktionen aus dem Maschinenprogramm heraus erforderlich. Gleiches gilt für die Rückkehr zum Maschinenprogramm, heraus aus dem Steuerprogramm. Die Technik dafür

liefert die ↑partielle Interpretation. Dabei wandert die Sprungtabelle in den Adressraum des Steuerprogramms, sie wird von einem ↑Systemaufrufzuteiler genutzt, um die mittels des speziellen Maschinenbefehls angeforderte Steuerprogrammfunktion aufzurufen. Der Aufruf im Maschinenprogramm verläuft nunmehr über einen ↑Systemaufrufstumpf, der die gegebene ↑Aufrufkonvention entsprechend der von ↑CPU und Steuerprogramm vorgegebenen Merkmale abbildet. Damit ist der Systemaufruf wie in seiner für gewöhnlich technischen Umsetzung realisiert.

Residenzmenge (en.) ↑*resident set*. Menge des im ↑Hauptspeicher zu einem Zeitpunkt für einen ↑Prozess vorgehaltenen Text- und Datenbestands, wobei zur Aufbewahrung des Gesamtbestands ↑seitennummerierter virtueller Speicher als Grundlage genommen wird. Die Bestandsgröße in beiden Fällen ist ganzzahliges Vielfaches einer ↑Seite und bildet eine abzählbar endliche Menge der Seiten des virtuellen Adressraums eines ↑Prozessexemplars, die gegenwärtig auf ↑Seitenrahmen abgebildet ist und damit im Hauptspeicher platziert vorliegt. Zu beachten ist der Unterschied zur ↑Arbeitsmenge eines Prozesses, deren Seiten zwar im Hauptspeicher residieren, die jedoch nur einen Teil aller residenten Seiten des Prozesses ausmachen: die Arbeitsmenge eines Prozesses ist Teilmenge der Residenzmenge des Prozesses. Darüberhinaus unterliegt jede Seite zuletzt genannter Sorte der Möglichkeit der individuellen ↑Verdrängung aus ihrem Seitenrahmen, solange sie (nämlich als ↑Betriebsseite) nicht auch der Arbeitsmenge des Prozesses angehört: eine Residenzmengenseite kann sehr wohl einzeln verdrängt werden, wohingegen eine Arbeitsmengenseite immer nur im Verbund derselben Arbeitsmenge verdrängt, das heißt, aus- und eingelagert wird — es sei denn, die Mächtigkeit der Arbeitsmenge ist 1.. VL 12.3

resource (dt.) ↑Betriebsmittel, ↑Ressource.

resource control (dt.) ↑Betriebsmittelkontrolle.

resource lock (dt.) ↑Betriebsmittelsperre.

resource management (dt.) ↑Betriebsmittelverwaltung.

response time (dt.) ↑Antwortzeit.

Ressource (en.) ↑*resource*. Lehnwort aus dem Französischen: (dt.) Auskommen, Mittel, Hilfsmittel. Auch als ↑Betriebsmittel bezeichnetes etwas, zur Verfügung gestellt durch das ↑Betriebssystem, um einen ↑Prozess zu betreiben. Grundsätzlich benötigt jeder Prozess (in einem ↑Rechensystem) jeweils wenigstens ein ↑Exemplar von drei Hardwareklassen solcher Mittel:

1. den ↑Hauptspeicher, in dem das ↑Programm für den Prozess vorliegen muss,
2. einen ↑Prozessor, um den Prozess überhaupt stattfinden lassen zu können und
3. die ↑Peripherie, damit der Prozess mit dem „Außenwelt“ kommunizieren kann.

Für gewöhnlich kommen für jedes dieser Exemplare weitere Mittel (Software) hinzu, die in der von einem Prozess zu leistenden Arbeit begründet sind. Für die Klasse ↑wiederverwendbares Betriebsmittel sind etwa dynamische Datenstrukturen wie auch Datenpuffer, Auftragsdeskriptoren oder Kontrollblöcke typische Beispiele, für die Klasse ↑konsumierbares Betriebsmittel sind dies ↑Signale oder ↑Nachrichten wie auch Zeit und Energie.

Restlaufzeit (en.) *remaining term*. Zeit, die ein ↑Prozess noch in Betrieb sein darf, die sein ↑Rechenstoß noch gültig ist (in Anlehnung an den Duden). ↑Bezugspunkt bildet ein ↑Termin oder allgemein ein in zeitlicher Hinsicht vorhersehbares ↑Ereignis, das den Endzeitpunkt des Rechenstoßes markiert. Das Zeitintervall vom gegenwärtigen Zeitpunkt (jetzt) bis zum Zeitpunkt, der für den Bezugspunkt gilt oder erwartet wird (später), definiert die *relative Zeit*, die letztlich ein bestimmtes ↑Betriebsmittel einem Prozess noch zugeteilt bleiben darf. Auch wenn hier jede Art ↑wiederverwendbares Betriebsmittel in Betracht gezogen werden

kann, ist typischerweise jedoch nur die ↑CPU gemeint. Formal ist dieses Interval wie folgt definiert:

$$t_{rest} = t_{burst} - |p_{now} - p_{start}|.$$

Dabei bestimmt t_{burst} die für den Prozess im Voraus veranschlagte ↑Stoßzeit. Diese kann durch einen festen Parameter wie etwa einer ↑Zeitscheibe oder einen variablen Wert als Ergebnis einer Prognose mittels ↑Zeitreihenanalyse bestimmt sein (vgl. auch ↑SRTF). Der aktuelle Zeitpunkt des Prozesses ist durch p_{now} gegeben, wohingegen p_{start} den Zeitpunkt definiert, zu dem der Prozess seinen Rechenstoß im Rahmen eines ↑Prozesswechsels aufgenommen hat. Die Differenz $|p_{now} - p_{start}|$ ergibt den bisher vom Prozess verbrauchten Zeitanteil seines Rechenstoßes.
Diese Zeitspanne (t_{rest}) bestimmt damit gemeinhin die Dauer, die der Prozess noch im ↑Prozesszustand laufend voraussichtlich verbringen kann oder wird, bevor sein Rechenstoß endet. Sie entspricht damit der einem Prozess zugebilligten effektiven Arbeitszeit, das heißt, sie enthält insbesondere keine ↑Schlupfzeit, also keinen für den Bedarfs- oder Notfall vorsorglich zurückbehaltenen ↑Zeitpuffer.

resume (dt.) fortsetzen; die Ausführung einer ↑Koroutine übernehmen. Einen ↑Koroutinenwechsel durchführen, indem von einem ↑Handlungsstrang der die Kontrolle über den ↑Prozessor abgebenden Koroutine hin zu einem Handlungsstrang der diese Kontrolle annehmenden Koroutine umgeschaltet wird.
Liegt der ↑CPU ein ↑orthogonaler Befehlssatz zugrunde, kann diese Operation leicht durch einen einzigen ↑Maschinenbefehl ausgedrückt werden. Beispiel dafür ist/war die ↑PDP 11/40, bei der die ↑Aktion `JSR PC, @(SP)+` einem Koroutinenwechsel entspricht. Diese Aktion (*jump to subroutine*):

1. sichert die auf dem ↑Laufzeitstapel liegende und von dort (mittels `@(SP)+`) entfernte Fortsetzungsadresse der zu aktivierenden Koroutine in ein internes ↑Prozessorregister,
2. legt den Inhalt des ↑Verbindungsregisters (`PC`) auf den Stapel ab und sichert damit die Fortsetzungsadresse der die Kontrolle über die CPU abgebenden Koroutine,
3. sichert die Adresse des (`JSR`) folgenden Maschinenbefehls als Rücksprungadresse in das angegebene Verbindungsregister (`PC`: diesbez. ein ↑Leerbefehl im gg. Fall) und
4. weist dem Befehlszähler die im ersten Schritt gesicherte Zieladresse zu, führt damit den Sprung aus und schaltet so zur adressierten Koroutine um.

Am Beispiel dieses Befehls wird vor allem deutlich, dass, analog zu ↑Routinen, Koroutinen in urwüchsiger Ausprägung (Reinform, *minimale Basis*) denselben Laufzeitstapel gemeinsam haben: ↑primitive Koroutine. Demgegenüber gibt es Modelle (*minimale Erweiterungen*), die jeder Koroutine einen eigenen Laufzeitstapel zugestehen: ↑komplexe Koroutine.

reusable resource (dt.) ↑wiederverwendbares Betriebsmittel.

ring-oriented protection (dt.) ↑ringorientierter Schutz.

ringorientierter Schutz (en.) ↑*ring-oriented protection*. Abschirmung einer einzelnen ↑Entität oder einer Menge von Enitäten durch einen ↑Schutzring. Dazu bildet eine solche Enität ein ↑Objekt, das einen bestimmten Schutzring als Attribut besitzt und damit logisch auf eben diesen Ring residiert. In dem Ansatz liegen mehrere Schutzringe konzentrisch übereinander, wobei der unterste (innerste), Ring 0, die höchste Privilegstufe definiert und jeder darüber liegende (äußere) Ring in Folge jeweils eine niedrigere Privilegstufe hat.
Für jeden ↑Prozess gelten festgelegte Regeln, damit der Zugriff auf ein Objekt gelingt. Dazu ist neben der Ringzugehörigkeit des Objekts auch die des Prozesses von Bedeutung. Maßgeblich für die Ringzugehörigkeit des Prozesses ist der Kontext, in dem sich der Prozess jeweils befindet: nämlich der den Prozess definierende ↑Text eines bestimmten Objektes und damit

der Ring dieses Objektes. Nur Zugriffe innerhalb eines Rings oder von einen inneren auf einen äußeren Ring sind zulässig. Zugriffe von einen äußeren auf einen inneren Ring werden abgefangen (↑*trap*) und der ↑Teilinterpretation unterzogen. Die Folge davon ist entweder ein ↑Schutzfehler mit anschließendem Prozessabbruch oder die Zugriffserlaubnis. Wird dem Prozess der Zugriff direkt oder indirekt (durch Teilinterpretation) gewährt, ist er in der Lage, die ↑Daten eines Objekts zu lesen oder zu schreiben beziehungsweise den Objekttext auszuführen. Letzteres ermöglicht den Prozess zum Schutzringwechsel: der Prozess agiert danach auf dem Ring, der durch das Objekt definiert ist, dessen Text den weiteren Prozessverlauf nunmehr vorschreibt.
Diese Technik wurde erstmals mit ↑Multics vorgestellt, in dem Fall mit 64 logischen (durch das ↑Betriebssystem implementierten) und 8 realen (durch die Hardware bereitgestellten) Ringen. Intel hatte (lange nach Multics) mit dem ↑i286 eine CPU mit vier Ringen auf den Markt gebracht. Alle nachfolgenden Prozessoren dieser Familie stellen dieses Merkmal weiterhin zur Verfügung.

RISC Abkürzung für (en.) *reduced instruction set computer*, (dt.) ↑Rechner mit vermindertem (primitiven) ↑Befehlssatz. Der ↑Prozessor in solch einem Rechner verfügt über vergleichsweise wenige, dafür aber hoch optimierte Befehle. Ein weiteres typisches Merkmal ist, dass Berechnungen ausschließlich ↑Prozessorregister als ↑Platzhalter für die Operanden verwenden, wozu letztere explizit durch Ladebefehle aus dem ↑Arbeitsspeicher zu lesen sind und das Berechungsergebnis explizit durch einen Speicherbefehl dahin zurückzuschreiben ist (*load/store architecture*). Für gewöhnlich sind die Befehlslängen gleich, so dass jeder Maschinenbefehl gleich viel Platz im Arbeitsspeicher belegt. Darüber hinaus ist bei dieser ↑Rechnerarchitektur angestrebt, jeden Befehl in gleicher Anzahl von Taktzyklen auszuführen. Anders als beim ↑CISC ist der Grundgedanke hinter der ↑Rechnerarchitektur, ↑Performanz vor allem durch Schlichtheit zu steigern und die ↑semantische Lücke nur durch Maßnahmen in Software zu verkleinern. Typische Beispiele sind die Prozessoren der ↑PowerPC-Familie.

RM Abkürzung für (en.) *rate monotonic*. Eine Methode zur ↑Einplanung, die einer ↑Aufgabe ihre ↑Priorität in Abhängigkeit von ihrer ↑Periode fest zuordnet (↑*off-line scheduling*); ein Verfahren zur ↑Prioritätszuweisung an ↑Echtzeitprozesse. Dabei nimmt die Priorität zu, je kürzer die Periode und je höher damit die Ankunftsrate ist. Der aus dem Verfahren hervorgehende statische ↑Ablaufplan wird nachgeschaltet abgearbeitet (↑*fixed-priority scheduling*).
Eine Spezialisierung von ↑DM, da die Periodenlänge auch als *relativer* ↑Termin aufgefasst werden kann.

memory chunk (dt.) ↑Speicherstück.

ROM Abkürzung für (en.) ↑*read-only memory*.

root directory (dt.) ↑Wurzelverzeichnis.

root file system (dt.) ↑Stammdateisystem.

round robin (dt.) Ringmodell, Reihummethode, ↑Rundlaufverfahren. Abgeleitet von (frz.) *ruban rond*, dem Sinn nach (dt.) Kreisband. Ursprünglich die Bezeichnung für einen Bitt- und Beschwerdebrief (an den frz. König, 17./18. Jhd.), der von den Verfassern in Kreisform unterschrieben wurde, um dadurch Spekulationen über eine mögliche Rangfolge oder der Identifikation von „Rädelsführern“ zu erschweren und so der eventuellen Hinrichtung vorzubeugen. Übernommen als bildhafte Beschreibung für ein ↑Zeitteilverfahren, das die ↑präemptive Planung von ↑Prozessen ermöglicht (Kleinrock, 1964).

Routine (en.) ↑*routine*. Bezeichnung für ein kleineres ↑Programm oder Teil eines Programms mit einer bestimmten, gewöhnlich häufiger benötigten Funktion (Duden). Zuweilen abstrahiert der Begriff auch von der konkreten Ausprägung des Programms, also ob es als ↑Unterprogramm in Erscheinung tritt oder als ↑Makrobefehl ausgelegt ist. Gemeinhin wird darunter aber eine Prozedur im programmiersprachlichen Sinn verstanden.

routine (dt.) Prozedur, ↑Routine.

routing (dt.) Wegewahl, ↑Leitweglenkung.

RPC Abkürzung für (en.) ↑*remote procedure call.*

RR Abkürzung für (en.) ↑*round robin.* Bezeichnung für eine Strategie zur ↑Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor im ↑Rundlaufverfahren. Dem grundlegenden Schema nach ↑FCFS, jedoch steht die ↑präemptive Planung im Zentrum des Verfahrens, womit auch die Einstufung als ↑mitlaufende Planung begründet ist. Der die Aufgaben verarbeitende Prozessor ist allgemeint charakterisiert als ↑wiederverwendbares Betriebsmittel, das im ↑Zeitteilverfahren benutztbar sein muss: das heißt, dieser Prozessor kann kein ↑unteilbares Betriebsmittel sein. Damit kann einer laufenden Aufgabe zu gegebener Zeit zugunsten einer bereitstehenden Aufgabe der Prozessor entzogen und später wieder zugeteilt werden (↑Verdrängung). Typischerweise die von einem Prozessor zu bearbeitende Aufgabe einem ↑Prozess und der Prozessor selbst entspricht der ↑CPU beziehungsweise einer ihrer möglicherweise mehreren ↑Rechenkerne.
Bei dem Verfahren erhält ein Prozess den Prozessor immer nur für eine ↑Zeitscheibe lang zugeteilt, dabei ist die Länge der Zeitscheibe für gewöhnlich für alle Prozesse gleich. Innerhalb dieses Zeitintervalls endet der ↑Rechenstoß des Prozesses entweder freiwillig oder unfreiwillig, letzteres als Folge einer ↑Unterbrechungsanforderung beim Ablauf der Zeitscheibe. Die Unterbrechung des Rechenstosses führt zur erneuten ↑Einplanung (FCFS) des betreffenden Prozesses. Letzteres geschieht jedoch nur dann, wenn im Moment des Zeitscheibenablaufs wenigstens ein anderer Prozess (auf der ↑Bereitliste stehend) die Prozessorzuteilung erwartet. Ist dies der Fall, wird der der nächste bereitstehende Prozess in Folge (FCFS) der ↑Einlastung des Prozessors zugeführt. Dieser Vorgang wiederholt sich fortlaufend.
Das mit FCFS bestehende Problem des ↑Konvoieffekts wird stark abgeschwächt. Da im Allgemeinen davon auszugehen ist, dass ein-/ausgabeintensive Prozesse ihre Rechenstöße noch vor Ablauf der Zeitscheide beenden, den Prozessor von selbst abgeben, und nur rechenintensive Prozesse durch Ablauf ihrer Zeitscheibe unterbrochen werden, ist dieser Effekt allerdings nicht gänzlich vermeidbar. Jedoch ist die Schwere des Problems nicht mehr durch die für gewöhnlich unbekannte Rechenstoßlänge bestimmt, sondern durch die bekannte Zeitscheibenlänge und damit auch begrenzt. Je größer letztere allerdings ist, umso stärker entartet das Verfahren zum bloßem FCFS und umso mehr gewinnt der Konvoieffekt an Bedeutung; je kleiner sie ist, desto häufiger erfolgen programmierte Unterbrechungen und desto höher sind die ↑Gemeinkosten des Verfahrens. Als Faustregel gilt daher, dass die Länge der Zeitscheibe etwas größer sein soll als die Dauer eines typischen Rechenstoßes. Genau genommen ist dieser Parameter jedoch abhängig von Art und Aufbau einer jeweiligen ↑Anwendung und ist durch Testläufe zu bestimmen, er basiert aber nicht selten auch nur auf Erfahrungswerte: typische Standards sind 100ms (↑BSD, ↑Linux) oder Abstufungen von 20ms, 40ms, 60ms und 120ms (↑Windows NT, je nach Konfiguration).

Exkurs Im Vergleich zu FCFS (vgl. S. 92) besteht der Unterschied in der Implementierung vor allem in der Notwendigkeit (a) einen ↑Zeitgeber samt ↑Gerätetreiber und diesbezüglicher ↑Unterbrechungsbehandlung vorzusehen sowie (b) zur ↑Synchronisation der Operationen des mitlaufenden ↑Planers. Genau genommen ist jedoch zwischen der synchronen und asynchronen Auslegung von FCFS zu differenzieren: nur wenn strikt synchrones FCFS die Basis für das hier vorgestellte präemptive Verfahren bildet, sind entsprechende Erweiterungen (nichtfunktionaler Art) vorzunehmen, um nämlich die benötigte asynchrone Nutzung zu ermöglichen. Die asynchrone Form von FCFS setzt für gewöhnlich bereits synchronisierte Planeroperationen voraus. Hier ist dann nur noch eine Vorkehrung zu treffen, die den gegenwärtigen (d.h., unterbrochenen) Prozess zwingt, den Prozessor freiwillig abzugeben. Dazu muss im Rahmen der durch Ablauf der Zeitscheibe ausgelösten Unterbrechungsbehandlung eine Planeroperation zur bedingten Prozessorabgabe (`check`) abgesetzt werden:

```
void check() {                              /* try to reschedule processor */
    backlog_t *list = labor();                            /* use ready list */
    if (ahead(list))                         /* any other process present? */
        if (FAA(&list->omen, 1) == 0)    /* yes, scheduler already active? */
            pause();                                      /* no, reschedule */
}
```

Sind die Planeroperationen durch ↑nichtblockierende Synchronisation geschützt, können sie jederzeit auch (vom Gerätetreiber ausgehend) asynchron aufgerufen und durchgeführt werden. Anderenfalls darf die Operation zur unbedingten Prozessorabgabe (`pause`, S. 92) nur dann asynchron zur Wirkung kommen, wenn im Moment des Zeitscheibenablaufs der Planer inaktiv ist.

Eine einfache Lösung, die allerdings nur vor Asynchronität durch Ablauf der Zeitscheibe auf einem ↑Uniprozessor schützt, zeigt obiges Beispiel (`check`). Dabei wird mit ↑FAA geprüft, ob der Planer bereits aktiv ist und gleichzeitig der Aktivzustand ($omen > 0$) des Planers definiert. Beide Schritte geschehen atomar. Ist der Planer bereits aktiv, endet die Operation ohne Prozessorabgabe. In dem Fall wird jedoch der Zustand ($omen > 1$) hinterlassen, dass nämlich die Prozessorabgabe angefordert wurde. Alle anderen Planeroperationen müssen dann nach folgendem Muster umgestaltet werden:

```
{
    backlog_t *list = labor();                            /* use ready list */
    FAA(&list->omen, 1);                                 /* scheduler active */

    /* do respective scheduler operation */

    if (FAS(&list->omen, 0) > 1)        /* scheduler inactive: preemption? */
        pause();                                   /* yes, release processor */
}
```

Hier wird der Verdrängungsoperation (`check`) durch FAA angezeigt, dass der Planer aktiv ist und die Prozessverdrängung aufzuschieben ist. Letztere wird am Ende der so geschützten Planeroperation (↑*critical section*) gegebenenfalls nachgeholt. Hierzu wird, ebenfalls atomar, durch ↑FAS einerseits der kritische Abschnitt verlassen ($omen = 0$) und andererseits ein Wert geliefert, der Auskunft darüber gibt, ob zwischenzeitlich die ↑Verdrängung des laufenden Prozesses angefordert wurde: $omen > 1$.

In dem Beispiel werden alle kritischen Planeroperationen durch eine ↑Verdrängungssperre geschützt. Da im gesperrten Fall Verdrängungswünsche aber nicht ins Vergessen geraten sollten, wird am Ende jeder kritischen Operation auf eine solche zwischenzeitlich eingegangene Anforderung geprüft und gegebenenfalls die Verdrängung (`pause`) nachgezogen.

Die Verdrängungsoperation (`check`) ist als ↑AST zu verstehen, der durch den ↑Unterbrechungshandhaber erster Stufe (↑FLIH) des Zeitgebers ausgelöst wird und als diesbezügliche Aktivität zweiter Stufe (↑SLIH) stattfindet. Angenommen, der Zeitgeber (↑RTC) sendet zyklische ↑Unterbrechungsanforderungen über den ↑Unterbrechungsvektor mit der Nummer 8 an die ↑CPU. Mit diesem Vektor sei folgender Handhaber verknüpft (vgl. S. 28):

```
__attribute__ ((interrupt)) void irq008(train_t *this) {
    guide(rtc_flih, this);           /* control real-time clock interrupt */
}
```

Der gezeigte ↑Unterbrechungshandhaberstumpf aktiviert den dem Zeitgeber zugehörigen FLIH, der einen AST auslöst, durch dessen Behandlung schließlich der korrespondiere SLIH zur Ausführung kommt. Während der FLIH sofort zur Ausführung kommt, wird der SLIH erst bei Rückkehr aus der Unterbrechungsbehandlung zur ↑Benutzerebene gestartet (`guide`). Der FLIH selbst hat die Aufgabe, den Zeitgeber zu verwalten und gegebenenfalls die von ihm ausgegangene Unterbrechungsanforderung zu bestätigen. Darüber hinaus muss er den SLIH bereitstellen, um damit schließlich die Prozessverdrängung durch den Planer zu bewirken:

```
void rtc_flih(train_t *this) {     /* real-time clock first-level handler */
    static sequel_t slih = {0, {rtc_slih, 0}};          /* AST descriptor */
    /* if need be, acknowledge IRQ at the clock device */
    defer(annex(), &slih);                                  /* release AST */
}
```

Jeder SLIH wird zeitverzögert zu dem ihn auslösenden FLIH ausgeführt. Dazu findet ein ↑Deskriptor Verwendung, der (als AST) den SLIH repräsentiert und in einer ↑Warteschlange verbleibt, bis der richtige Moment zur Ausführung gekommen ist. Im gegebenen Fall aktiviert der FLIH die Prozessverdrängungsoperation (`check`) im Planer:

```
void rtc_slih(data_t none) {      /* real-time clock second-level handler */
    /* if need be, do some additional handling of the clock device */
    check();                                          /* request preemption */
}
```

Der richtige Moment der Ausführung des SLIH ist gegeben, wenn der ↑Laufzeitstapel, auf dem die Unterbrechungsbehandlung stattgefunden hat, komplett zurückgebaut wurde. Sollte dann ein AST anhängig sein, wird dieser behandelt noch bevor der unterbrochene Prozess zur Benutzerebene zurückkehrt. Die Aktivierung des SLIH ist ein solcher AST, sie kommt in einer Art und Weise zur Geltung, als wenn der unterbrochene Prozess dazu einen ↑Systemaufruf abgesetzt hätte — und dadurch „freiwillig“ den Prozessor zugunsten eines anderen Prozesses abgeben möchte.
Entsprechung der Semantik von einem AST, nämlich dass dieser asynchron zur Behandlung kommt, muss die Verdrängungsoperation in ↑Synchronisation mit dem gegebenenfalls im Planer bereits stattfindenden Prozess gebracht werden. Hierzu kommt eine Verdrängungssperre zum Einsatz (vgl. `check`, oben). Ist die Sperre nicht gesetzt, wird der laufende/unterbrochene Prozess verdrängt. Anderenfalls wird die Verdrängungsanforderung vermerkt und der betreffende Prozess erst nach Aufhebung der Sperre zur Prozessorabgabe geführt. Dieses Protokoll wird durch einen Zähler gesteuert, dessen Manipulation und Zustandsabfrage auf ↑atomare Operationen (FAA, FAS) zurückgreift. Die hier beispielhaft erklärte FLIH-AST-SLIH-Folge dient damit keinesfalls der Synchronisation, sondern lediglich dazu, ↑Unterbrechungslatenz zu senken und dabei möglichem Laufzeitstapelüberlauf vorzubeugen.

RSX 11 Familie von (Mehrplatz-/Mehrbenutzer-) Echtzeitbetriebssystemen für ↑Rechner der PDP-11 Familie (↑DEC), erste Installation 1972 (↑PDP 11/40). Abkürzung ursprünglich für „*Real-Time System Executive*“, später für „*Resource Sharing Executive*“. Bewährte Prinzipien von RSX-11M (1973, u.a. ↑AST) wurden später für ↑VMS übernommen, das unter derselben Leitung (David Cutler) entstand.

RTC Abkürzung für (en.) ↑*real-time clock*.

Rückschreibetechnik (en.) ↑*write-back*. Schreibstrategie in Bezug auf eine ↑Zwischenspeicherzeile, die im ↑Zwischenspeicher als Kopie vorliegt. Bei ↑schreibendem Zugriff geschieht die Aktualisierung der betreffende Zeile nur im Zwischenspeicher. Der zugehörige Eintrag wird jedoch markiert (↑*dirty bit*), um damit anzuzeigen, dass die Zeile im Zwischenspeicher mit der der nächst höheren Stufe der ↑Speicherhierarchie inkonsistent ist. Erst bei Bedarf, das heißt, entweder programmiert und damit vorhersehbar per ↑Maschinenbefehl (*flush*) oder nicht programmiert und daher unvorhersehbar bei einer plötzlich erforderlichen Ersetzung der zwischengespeicherten Zeile, geschieht das Zurückschreiben der markierten Einträge auf die nächst höhere Speicherhierarchiestufe.

Ruhezeit (en.) ↑*off time*. Zeit der Ruhe eines ↑Prozesses (in Anlehnung an den Duden). Der Prozess unterlässt ↑Aktionen, die ↑Interferenz durch Zugriffe auf ein mit anderen Prozessen gemeinsam genutztes ↑Betriebsmittel hervorrufen könnten. Ein typisches Betriebsmittel in dem Zusammenhang ist der ↑Bus, insbesondere wenn ein ↑speichergekoppelter Multiprozessor die Grundlage für diese Prozesse ist. Um ↑Interferenz zu vermeiden oder zumindest

abzuschwächen wird ein Prozess dann für eine gewisse Zeit von selbst von weiteren Buszugriffen Abstand nehmen und zurückweichen (↑*backoff*). Als besonders problematisch und Interferenz verstärkend erweisen sich dabei gehäufte, unteilbare Buszugriffe (↑*bus-lock burst*.

Exkurs Die Zeit, wie lange die Prozesse derartige Zugriffe meiden sollen, wird *statisch* oder *dynamisch* bestimmt. Im ersteren Fall ist diese Zeit ein Parameter des ↑Prozessors, auf dem der betreffende Prozess stattfindet. Die jeweiligen Größen sollten dann verschieden sein, damit die verschiedenen Prozesse auch verschieden lang Ruhe bewahren und sie dadurch nach ihrer Ruhephase eher nicht erneut in Wettstreit geraten. Nachfolgend ein Beispiel für eine Funktion, die einen solchen prozessorabhängigen Parameter liefert:

```
time_t offtime() {                    /* initial backoff parameter: static */
    extern time_t __offtime[];              /* off-times of all processors */
    return __offtime[aegis()];              /* off-time for this processor */
}
```

Anhand der ↑Prozessoridentifikation wird die „Schirmherrschaft“ (*aegis*) des diese Funktion nutzenden Prozesses bestimmt, um daraufhin einem Parameterfeld die dem betreffenden Prozessor zugewiesene Größe zu entnehmen.
Im dynamischen Fall wird, ausgehend von einer für gewöhnlich zufällig gewählten Größe, die Dauer der jeweils nächsten Ruhephase eines Prozesses anhand der Dauer seiner vorangegangenen Ruhephase berechnet. Diese Verlängerung seiner Ruhephase nimmt der Prozess immer dann vor, nachdem er einen Konflikt beim Zugriff auf das gemeinsame Betriebsmittel — beziehungsweise die ↑gemeinsame Variable — erkannt hat.
Initial ist die Ruhephase des Prozesses bei dynamischen Verfahren zur Vermeidung von zukünftigen Wettstreits von zufälliger Dauer. Für gewöhnlich wird die diesbezügliche Zeitgröße unter Verwendung eines ↑Zufallsgenerators bestimmt. Je nach technischer Ausführung dieses Generators gestaltet sich die Bildung dieser Größe unterschiedlich. Liefert der Generator beispielsweise mehrstellige Zufallszahlen, deren Wertebereich durch die ↑Wortbreite eines ↑Maschinenworts definiert ist, dann kann die Bildung des initialen Werts wie folgt geschehen:

```
time_t offtime() {                          /* initial backoff parameter */
    return anyold(sizeof(time_t) / 2);            /* randomised off-time */
}
```

Generiert wird eine x-beliebige (*any-old*) natürliche Zahl $\mathbb{N}_0$ im Wertebereich $[0, 2^n - 1]$, mit $n =$ `(sizeof(time_t) / 2) * 8`, wenn von einem 8-Bit breitem ↑Byte ausgegangen wird (vgl. auch S. 402). Die Zahl liegt im niederwertigen n-Bit breiten Abschnitt einer doppelt so breiten ↑Bitleiste und entspricht damit eher einem kleineren Wert als durch ein ↑Exemplar des ↑Datentyps `time_t` gespeichert werden kann. Als Startwert für die Dauer der Ruhephase eines Prozesses wird also eher von einer kürzeren Zeit ausgegangen, die dann in dem dynamischen Verfahren, um den Prozess effektiv zur Ruhe zu bringen, noch angepasst werden kann oder schrittweise verlängert wird (vgl. S. 359 bzw. S. 330).
Liefert der Zufallsgenerator nur einen zufälligen Wahrheitswert, also eine Zufallszahl im Wertebereich $[0, 1]$, reicht dies für einen initialen Zeitwert nicht aus. Ein üblicher Ansatz besteht dann darin, für eine festgelegte Anzahl niederwertiger Bits den Wert jedes einzelnen Bits dieser Menge zufällig auf 0 oder 1 zu setzen. Auch diese Vorgehensweise resultiert in einen nicht all zu großen Startwert, da die so zusammengebaute Größe als gewöhnliche mehrstellige natürliche Zahl $\mathbb{N}_0$ betrachtet in ihrer Binärrepräsentation eine gewisse Anzahl höherwertiger Bits immer auf 0 gesetzt hat. Dazu wird die eben betrachtete Funktion wie folgt abgeändert:

```
#define TIMEBITS (sizeof(time_t) * 8) / 2    /* width of initial off-time */

time_t offtime() {                  /* initial backoff parameter: bitwise */
    time_t time = 0;                                          /* placeholder */
    for (int bit = 0; bit < TIMEBITS; bit++)   /* assemble off-time value */
        time |= (anyold(0) << bit);                /* record bit by chance */
    return time;                                    /* randomised off-time */
}
```

Auch wenn dieses „händische“ Zusammenbauen eines zufälligen Startwerts vergleichsweise aufwendig ist auf Basis eines n-Bit Zufallsgenerators, $n > 1$ (s. oben), so hätte die Lösung dennoch den Charme, dass damit eben eine beliebig breite Bitleiste für den Wertebereich der Zufallszahl vorgegeben werden kann. Die Alternative dazu wäre die Maskierung einer Bitleiste maximaler Wortbreite, indem alle Positionen des gewünschten Abschnitts höherwertiger Bits auf null gesetzt werden. Das geschieht dann durch ein einfaches logisches Und (`&`) mit dem Wert `~((1 << TIMEBITS) - 1)`, wobei `TIMEBITS` einem beliebigen Wert im Bereich [1, `(sizeof(time_t) * 8) - 1`] entspricht — aber diese Option scheidet eben mit einem 1-Bit Zufallsgenerator aus.

run to completion (dt.) laufen bis zur Fertigstellung; ↑pausenloser Ablauf.

run-time (dt.) ↑Laufzeit.

run-time context (dt.) ↑Laufzeitkontext.

run-time stack (dt.) ↑Laufzeitstapel.

run-time system (dt.) ↑Laufzeitsystem.

Rundlaufverfahren (en.) ↑*round robin*, Abkürzung ↑RR. Reihummethode der ↑Ablaufplanung. bei der jeder einzelne ↑Prozess im stetigen Wechsel mit anderen Prozessen solange die ↑CPU zugeteilt bekommt, bis er seine ↑Aufgabe erfüllt hat. Dabei wird der Wechsel durch das ↑Betriebssystem erzwungen (↑*preemption*), als Folge der ↑Unterbrechung des gegenwärtig auf der CPU stattfindenden Prozesses. Der unterbrochene Prozess wird daraufhin neu eingeplant.

Rundruf (en.) ↑*broadcast*. Bezeichnung für einen Ruf, der an jede ↑Entität innerhalb eines bestimmten ↑Bezugssystems geht. Anders als ↑Gruppenruf, ein umfassender „Wurf“ derselben Mitteilung (↑*signal*, ↑*message*) an alle.

safety (dt.) ↑Betriebssicherheit.

Sammelruf (en.) ↑*broadcast*. Ruf zum Sammeln (Duden), ein ↑Rundruf.

SASOS Abkürzung für (en.) ↑*single-address space* ↑*operating system*.

satellite computer (dt.) ↑Satellitenrechner.

Satellitenrechner (en.) ↑*satellite computer*. Rechenanlage, die für eine andere (für gewöhnlich größere) Rechenanlage vor- und nachbereitende Aufgaben wahrnimmt. Eine solche Anlage vermittelt einem ↑Hauptrechner den Zugang zur ↑Peripherie, an ihr sind verschiedene Ein-/Ausgabegeräte unterschiedlicher Art und Geschwindigkeit angeschlossen. Eingaben werden entgegengenommen und samt verarbeitende Programme zu einem ↑Stapel zusammengefasst, der dann über ein schnelles ↑Peripheriegerät (Band-/Wechselplattenlaufwerk, Steuergerät zur Hochgeschwindigkeitskommunikation) zu gegebener Zeit zum Hauptrechner übertragen wird. Über ein solches Gerät werden ebenfalls die vom Hauptrechner erzeugten Ausgaben entgegengenommen und den zugehörigen Ausgabegeräten zugestellt.

SC Abkürzung für (en.) *store conditional*. Bezeichnung für eine ↑Elementaroperation mit zwei Operanden: `bool SC(word_t *, word_t)`. Der erste Operand ist die ↑Adresse eines ↑Speicherworts, das mit dem Wert des zweiten Operanden beschrieben werden soll. Besteht für den ausführenden ↑Prozessor eine eigene (zuvor durch ↑LL vorgenommene) Reservierung für diese Adresse, gelingt die Schreiboperation (`true`) und die bestehende Reservierung wird zurückgenommen. Anderenfalls scheitert die Schreiboperation (`false`), ohne jedoch den Inhalt des adressierten Speicherworts zu verändern oder die gegebenenfalls bestehende andere Reservierung zu löschen.
Ein solcher ↑Maschinenbefehl ist nur typisch für ↑RISC. In Kombination mit ↑LL erlaubt er die ↑Synchronisierung von gleichzeitigen ↑Prozessen, die über eine gemeinsame ↑Variable miteinander gekoppelt sind (s. auch ↑LL/SC).

Schadsoftware (en.) ↑*malware*. Software, die Schäden in anderer Software, in elektronischen Systemen verursacht (Duden). Ein ↑Programm, das sich — bei Ausführung auf einen ↑Prozessor (d.h., als ↑Prozessinkarnation im ↑Prozesszustand laufend) — gezielt bösartig (*malicious*) gegenüber anderen Programmen beziehungsweise dem ↑Rechensystem verhält. Dazu greift ein solches Schadprogramm *direkt* oder *indirekt* heimtückisch in andere Programme oder ↑Prozesse ein. Im direkten Fall vermag es reguläre (allgemein gespeicherte oder im ↑Arbeitsspeicher laufende) Programme, wozu insbesondere auch das ↑Betriebssystem zählt, „von außen“ so zu verändern, dass diese bei ihrer Ausführung eigentlich nicht intendierte Operationen ausführen (eine „Tat“ begehen: ↑*exploit*), das heißt, ↑Daten offenlegen oder selbst zum Schadprogramm werden. Im indirekten Fall nutzt das Schadprogramm reguläre Wege, um an sonst private Informationen zu gelangen (↑*covered channel*) oder durch ↑Systemlast Störungen (↑*interference*) des Rechenbetriebs zu bewirken und dadurch gegebenenfalls auch ↑Echtzeitprozesse zu beeinträchtigen oder gar zum Scheitern zu bringen.

schedule (dt.) ↑Ablaufplan.

scheduler (dt.) ↑Planer.

scheduling (dt.) ↑Ablaufplanung, ↑Einplanung, ↑Planung des zeitlichen Ablaufs.

scheduling algorithm (dt.) ↑Einplanungsalgorithmus.

scheduling criteria (dt.) ↑Einplanungskriterium.

scheduling latency (dt.) ↑Einplanungslatenz.

scheduling queue (dt.) ↑Planungswarteschlange.

Schlafzustand (en.) ↑*sleep state*. Systemzustand, der die einem ↑Prozessor bereitgestellte Energie, ihm angelegte Leistung beschreibt. Je tiefer dieser Zustand, desto geringer der Energieverbrauch beziehungsweise die Leistungsaufnahme des Prozessors, umso länger dauert seine Aufwachphase und umso umfassender sind die Maßnahmen in einem ↑Betriebssystem zur Aufrechterhaltung von Hardwarekontext (insb. ↑CPU, ↑Zwischenspeicher und ↑Hauptspeicher). Der Grad der Abstufung ist prozessorabhängig, heutige (2025) Prozessoren beinhalten eine bis fünf Stufen. Das Betriebssystem bringt den Prozessor zum Schlafen, sobald für ihn ↑Leerlauf festgestellt wurde. Der Prozessor weckt wieder auf, wenn er eine ↑Unterbrechungsanforderung erhält.

Schlange (en.) ↑*queue*. Bezeichnung für eine Datenstruktur zur gemeinhin vorübergehenden Aufbewahrung von ↑Objekten, deren Anzahl über die Zeit verschieden und gegebenenfalls auch nicht im Voraus bestimmbar ist. Mit dieser Datenstruktur ist eine typische Art und Weise der Reihung verbunden, bei der ein aufzubewahrendes Objekt ans Ende (*tail*) angefügt (*enqueue*) und ein aufbewahrtes Objekt vom Anfang (*head*) entfernt (*dequeue*) wird. Gebräuchlich ist die Implementierung als ↑verkettete Liste.

Exkurs Hier ein Beispiel in ↑C, das die Einreihung des Endelements mit konstantem und

das Entfernen des Kopfelements mit begrenztem Aufwand durchführt (vgl. auch S. 374):

```
typedef struct queue {
    chain_t head;
    chain_t *tail;
} queue_t;
```

Um den konstanten Einreihungsaufwand zu erreichen verwendet diese Implementierung einen Zeiger (`tail`) auf die Stelle, an der das jeweils nächste Element eingehängt werden kann. Initial ist diese Stelle der Kopfzeiger (`head`).

Nachfolgend die dazu korrespondierende Initialisierungsprozedur:

```
inline void zeroize(queue_t *this) {
    this->head = 0;                                  /* define empty queue */
    this->tail = &this->head;          /* refer to initial last chain link */
}
```

Dadurch wird beim Einfügen des ersten Elements in eine leere Liste implizit der Kopfzeiger mit initialisiert. Als Folge entfällt die diesbezügliche Fallunterscheidung, die sonst einen nur noch nach oben begrenzten Berechnungsaufwand mit sich bringt. Beim Entfernen des letzten Elements ist, wie sonst üblich, dem Endzeiger einen definierten Wert zu geben, der im gegebenen Beispiel, anstatt mit Null, nunmehr mit der ↑Adresse des Kopfzeigers initialisiert wird. Die betreffenden Operationen gestalten sich wie folgt (s. auch S. 375):

```
inline void enqueue(queue_t *this, chain_t *item) {
    this->tail = join(&this->tail, item);      /* add to and advance tail */
}
inline chain_t *dequeue(queue_t *this) {
    chain_t *item = this->head.link;      /* fetch stored element, if any */
    if (item && (this->head.link = item->link) == 0) /* last one fetched? */
        this->tail = &this->head;           /* yes, reset linkage pointer */
    return item;                          /* deliver fetched element or 0 */
}
inline chain_t *backlog(queue_t *this) {
    return this->head.link;            /* indicate stock of items, if any */
}
```

Dieses Prinzip entspricht der auch als ↑FCFS oder ↑FIFO bezeichneten Vorgehensweise zur Bearbeitung anstehender ↑Aufgaben oder ↑Daten..

Schlitzzeit (en.) ↑*slot time*. Die Länge von einem ↑Zeitschlitz. Sowohl Begriff als auch Parameter von ↑Rechnernetzen, eine Zeitgröße. Ihr Maß entspricht wenigstens der doppelten ↑Laufzeit eines elektronischen Impulses über die Länge des maximalen theoretischen Abstands zwischen zwei Netzknoten. Im Falle von ↑CSMA/CD bestimmt die Zeitgröße (a) die Obergrenze für die Übernahme des Mediums, (b) eine Beschränkung für die Länge eines bei einer Kollision erzeugten Paketfragments und (c) das Maß zur ↑Einplanung der Sendewiederholung von ↑Daten. Eine ↑Wartezeit, die benötigt wird, bis das Medium übertragungsfrei ist und für eine neue Übertragung wieder genutzt werden kann. Die aufgrund physikalisch-technischer Gegebenheiten kleinste Größe einer ↑Zeitnische.

Schlossalgorithmus (en.) ↑*lock algorithm*. Lösungs- und Bearbeitungsschema, Handlungsvorschrift zur ↑Synchronisierung ↑gekoppelter Prozesse. Allen Implementierungen solcher Algorithmen gemeinsam ist ↑wechselseitiger Ausschluss der beteiligten Prozesse, wobei dieser im Falle der ↑Konkurrenzsituation für gewöhnlich durch ↑geschäftiges Warten erreicht wird.

Schlossvariable (en.) ↑*lock variable*. Die in einem ↑Schlossalgorithmus verwendete ↑Variable zur Zustandsbeschreibung einer Sperre, die ↑Prozessen das weitere Vorankommen unterbinden soll. Im einfachsten Fall eine Boolesche Variable, deren Wert `true` bedeutet, dass die Sperre verhängt wurde. Demgegenüber zeigt der Wert `false` an, dass die Sperre aufgehoben ist. Durch Abprüfen dieser Variable erfährt ein Prozess somit, ob er weiter voranschreiten darf. Was er im Falle einer verhängten Sperre tut, also ob er (a) ↑aktives Warten betreibt, (b) ↑passives Warten bevorzugt oder (c) sich zeitweilig einer anderen ↑Aufgabe zuwendet, bleibt

letztlich dem abfragenden Prozess überlassen. Das Verhalten des Prozesses in Abhängigkeit vom Zustand der Sperre ist im Schossalgorithmus festgelegt.

Exkurs Ein durch solch einer Sperre mit anderen Prozessen ↑gekoppelter Prozess muss die zugehörige Variable jederzeit ändern können. Daher scheidet ein ↑Prozessorregister zur Zwischenspeicherung der Variablen für gewöhnlich aus, da dieses Teil des bei ↑Prozesswechseln invariant gehaltenen ↑Prozessorstatus ist. Somit bietet sich folgende Definition eines ↑Datentyp für diese Variable an:

```
typedef volatile bool lock_t;
```

Das Schlüsselwort `volatile` (*flüchtig*) weist den ↑Kompilierer an, eine so attributierte Variable dauerhaft im ↑Arbeitsspeicher zu halten. Eine Werteänderung, zu der ein Prozessorregister im Zuge der Berechnung und zur Zwischenspeicherung verwendet wurde oder werden musste, ist sofort im Arbeitsspeicher zu manifestieren. Allerdings ist mit diesem Attribut nicht automatisch auch ↑Speicherkohärenz garantiert.

Schlupfzeit (en.) ↑*slack time*. Zeit, die ein ↑Prozess noch in Reserve hat, um die von ihm zu bearbeitende ↑Aufgabe noch rechtzeitig abschließen zu können. Formal ist dieses Intervall wie folgt definiert:

$$t_{slack} = |p_{due} - p_{start}| - t_{rest},$$

mit p_{due} als vorgesehener Endzeitpunkt für die Aufgabenbearbeitung, p_{start} dem Startzeitpunkt und t_{rest} der ↑Restlaufzeit des Prozesses. Ein für den Bedarfs- oder Notfall eines Prozesses vorsorglich zurückbehaltener ↑Zeitpuffer, der mit zunehmender ↑Wartezeit dieses Prozesses allerdings immer kleiner wird.

schreiben (en.) ↑*write*. Irgendwo festhalten, eintragen, verbuchen von ↑Daten (in Anlehnung an den Duden). Hier das Kopieren von Daten, die mit einer eindeutigen ↑Adresse innerhalb der ↑Speicherhierarchie versehenen sind. Die ↑Aktion entnimmt die Daten bestimmten Verarbeitungseinheiten des ↑Prozessors beziehungsweise legt eine Kopie auf wenigstens einer höherer Hierarchiestufe an. Um beispielsweise das Berechnungsergebnis einer arithmetischen Operation abzuspeichern, kommt das betreffende Datum aus einem internen ↑Register (Stufe 1) der ALU hinein in den↑Zwischenspeicher (Stufe 3: ↑*write-back*, ↑*write-allocate*), um gegebenenfalls sofort weiter (↑*write-through*) in den ↑Hauptspeicher (Stufe 4) zu gelangen. Anders als ↑lesen wird aufwärts jeweils ein weiteres ↑Exemplar auf höherer Stufe angelegt oder aktualisiert, und zwar nach seiner auf tieferer Stufe existierenden Vorlage.

Schreiber (en.) ↑*writer*. Ein ↑Daten verarbeitender und ↑schreibender ↑Prozess, der gegebenenfalls in ↑Konkurrenz mit anderen seiner Art oder ↑Lesern agiert.

Schreibmarke (en.) ↑*cursor*. Bezeichnung der Bildschirmmarke für die aktuelle Bearbeitungsposition.

Schreibpuffer (en.) ↑*write buffer*. Ein ↑Puffer (Hardware) zwischen dem ↑Zwischenspeicher und der zu ihm nächst höheren Stufe in der ↑Speicherhierarchie, um für die zu ↑schreibende ↑Zwischenspeicherzeile die bei der betreffenden ↑Aktion anfallende ↑Latenzzeit zu verbergen.

Schreibzuteilungstechnik (en.) ↑*write-allocate*, ↑*non-write-allocate*. Schreibstrategien in Bezug auf eine ↑Zwischenspeicherzeile, die *nicht* im ↑Zwischenspeicher vorliegt. Beim Fehlzugriff (↑*cache miss*) wegen eines zu ↑schreibenden Operanden wird bei der ersten Variante (*write-allocate*) die betreffende Zeile aus der nächst höheren Stufe der ↑Speicherhierarchie geholt und in den Zwischenspeicher kopiert. Anschließend wird der dem Operandenzugriff entsprechende Bereich in der Zeilenkopie aktualisiert und der Eintrag im Zwischenspeicher als verändert markiert (↑*dirty bit*). Im Gegensatz dazu schreibt die zweite Variante (*non-write-allocate*) am Zwischenspeicher vorbei direkt in die nächst höhere Stufe der Speicherhierarchie (↑*write-through*). Dadurch wird die Ersetzung zwischengespeicherter Zeilen, um Platz für eine Zeilenkopie zu schaffen, unwahrscheinlicher.

Schutz (en.) ↑*protection*. Vorrichtung, die eine Gefährdung von einem ↑Prozess abhält oder einen Schaden in einem ↑Rechensystem abwehrt (in Anlehnung an den Duden). Die hierzu im Grundsatz erforderlichen Maßnahmen haben einerseits einen räumlichen und zeitlichen Aspekt der Isolation von Prozessen und sind andererseits bestimmt durch die jeweilige ↑Betriebsart des Rechensystems: sie sind im Allgemeinen nur typisch für ↑Mehrprogrammbetrieb und bei ↑Simultanverarbeitung. In räumlicher Hinsicht ist (bei Mehrprogrammbetrieb) die Integrität der einem Prozess eigenen Anteile von ↑Text und ↑Daten sicherzustellen. Dies wird dadurch erreicht, indem entweder die jeweilige ↑Adresse eines solchen Anteils von keinem anderen Prozess in Erfahrung gebracht werden oder kein Prozess aus seinem ↑Prozessadressraum ausbrechen kann. Ersterer Ansatz geht von einen ↑Einzeladressraum aus, wohingegen letzterer Ansatz einen ↑Mehradressraum zur Grundlage hat. Beide Modelle sind zentraler Bestandteil der ↑Schutzdomäne von einem Prozess.
In zeitlicher Hinsicht ist (zur Simultanverarbeitung) zusätzlich sicherzustellen, dass kein Prozess die ↑CPU monopolisieren und damit ununterbrochen nur noch für sich selbst belegen kann. Dies stellt die Anforderung an die ↑Prozesseinplanung, ein ↑Zeitteilverfahren zur Grundlage zu nehmen. Ist als Betriebsart ein Mischbetrieb vorgesehen, bei dem unter anderem der ↑Echtzeitbetrieb von bestimmten Prozessen unterstützt werden soll, muss diese Gruppe von Prozessen weitestgehend vor störenden Einflüssen (↑Interferenz) in ihrem Ablauf geschützt werden. Dies betrifft vor allem Verfahren zur ↑Virtualisierung, wobei hier die besondere Schwierigkeit besteht, dass das zugrunde liegende Steuerprogramm (↑VMM) für gewöhnlich keine Kenntnis über die Prozesse hat, die eine ↑virtuelle Maschine ermöglicht: weder eine reale noch eine virtuelle Maschine kennt die Bedeutung von dem ↑Programm, das sie ausführt; ihr ist damit auch nicht bewusst, dass dieses Programm ein Betriebssystem sein könnte und möglicherweise ↑Ablaufplanung nach besonderen benutzerorientierten, zeitlichen Kriterien durchsetzen muss. Gilt zudem umgekehrt, dass kein Prozess in Erfahrung bringen kann, ob er auf einer realen oder virtuellen Maschine stattfindet, ist die Durchsetzung solcher zeitlichen Kriterien bei gleichzeitig (auf derselben realen Maschine) laufenden virtuellen Maschinen nicht mehr gewährleistet: das zur Virtualisierung einer realen Maschine notwendige ↑Multiplexverfahren wird in aller Regel einen Konfliktherd zu der jeweils auf einer virtuellen Maschine laufenden ↑Prozesseinplanung bilden.

Schutzdomäne (en.) ↑*protection domain*. Gebiet, auf dem für einen ↑Prozess ein bestimmter ↑Schutz gewährleistet ist. Jedes diesem Gebiet zugeordnete ↑Objekt ist dem Prozess (↑Subjekt) in bestimmter Weise zugänglich.
Für gewöhnlich ist das Gebiet bestimmt durch eine Menge von Objekten, auf die ein Prozess zugreifen kann. Mit jedem Objekt darin ist wiederum eine Menge von Rechten verbunden, die das ↑Zugriffsrecht des Prozesses festlegt. Prozesse können solche Domänen wechseln, jedoch nur unter Kontrolle der (realen/virtuellen) Maschine, auf die er stattfindet. Typisches Beispiel für solch einen Domänenwechsel ist der ↑Systemaufruf, bei dem der Prozess (a) seinen ↑Prozessor in den privilegierten ↑Arbeitsmodus bringt und (b) in einen anderen oder erweiterten ↑Adressraum, je nach ↑Mehradressraummodell, wechselt.
Zusätzlich zu dem jeweiligen ↑Prozessadressraum wird — bei einem ↑UNIX-artigen ↑Betriebssystem — dieses Gebiet durch die jeweilige ↑UID und ↑GID eines Prozesses abgesteckt. Dazu erhält jedes vom Betriebssystem verwaltete und Prozessen zugängliche Objekt ein {UID, GID}-Paar bei seiner Erzeugung zugewiesen. Dieses Paar nimmt für gewöhnlich die Werte der entsprechenden Kennungen des Prozesses an, der die Objekterzeugung effektiv ausgelöst (z.B. mit `creat(2)` eine ↑Datei angelegt) hat. So lässt sich für jede gegebene {UID, GID}-Kombination eine Menge von Objekten erstellen, auf die ein Prozess zugreifen kann. Alle Prozesse mit der gleichen Kombination von {UID, GID}-Werten besitzen Zugriff auf exakt die gleiche Menge von Objekten. Diese Mengen (d.h., Domänen) sind verschieden für Prozesse mit unterschiedlichen {UID, GID}-Kombinationen.

Schutzfehler (en.) ↑*protection error*. ↑Zugriffsfehler in Bezug auf das ↑Schutzsystem einer (realen/virtuellen) Maschine. Betrifft die Verletzung von ↑Speicherschutz, aber auch von Rech-

ten, die ein ↑privilegierter Befehl von einem ↑Prozess erfordert. Stellt eine ↑Ausnahmesituation dar.

Schutzgatter (en.) ↑*fence*. ↑Abgrenzung durch eine unveränderliche ↑Gatteradresse. Die Adresse ist „fest verdrahtet“ (*hard-wired*) in der Hardware und bestimmt Position wie auch Größe der geschützten Zone für das ↑Betriebssystem im ↑Hauptspeicher. Nach dem ↑Urladen des Betriebssystems ist der unbelegte Bereich in dieser Zone entweder ↑Verschnitt oder als ↑dynamischer Speicher für das Betriebssystem nutzbar. Die Gatteradresse ist dem ↑Binder bekannt, der damit dann die ↑Verlagerung von dem jeweiligen ↑Maschinenprogramm durchführt, bevor es vom ↑Lader des Betriebssystems in die ungeschützte Zone gebracht wird.
Dieses aus den Anfängen der Betriebssystementwicklung stammende Schutzkonzept findet genau genommen auch heute (2025) noch in ↑Linux und ↑Windows Verwendung. Hier ist der ↑Adressbereich $A = [0, 2^N - 1]$, den eine ↑CPU mit ↑Adressbreite N in logischer Hinsicht abdecken kann, ebenfalls zweigeteilt, wenn nämlich ein ↑logischer Adressraum oder ein ↑virtueller Adressraum durch das Betriebssystem bereitgestellt werden soll. Für $N = 32$ (4 GiB ↑Adressraum), beispielsweise, wird dem Maschinenprogramm eine Zone (vorderer Adressbereich) zugeteilt, die entweder 50% (Windows) oder 75% (Windows /3GB-Schalter, Linux) dieses Adressbereichs entspricht. Der übrige Adressbereich, der letztlich die geschützte Zone (hinterer Adressbereich) für das Betriebssystem ausmacht, ist einem Maschinenprogramm unter Linux/Windows nicht zugänglich. Andererseits ist Linux/Windows die ungeschützte Zone sehr wohl zugänglich. Dieser 50% beziehungsweise 75% Schnitt in dem Adressbereich definiert letztlich eine Gatteradresse in einem logischen/virtuellen Adressraum.

Schutzgatterregister (en.) ↑*fence register*. ↑Abgrenzung durch eine veränderliche ↑Gatteradresse. Die Adresse ist in einem speziellen ↑Prozessorregister der ↑CPU gespeichert. Der Registerinhalt bestimmt Position wie auch Größe der geschützten Zone für das ↑Betriebssystem im ↑Hauptspeicher. Nach dem ↑Urladen des Betriebssystems markiert dessen Anfangs- oder Endadresse im Hauptspeicher die Gatteradresse, je nachdem, ob die geschützte Zone den hinteren oder vorderen ↑Adressbereich ausmacht. Diese Adresse wird sodann in das ↑Register geschrieben, bevor das Betriebssystem dem ↑Maschinenprogramm in der ungeschützten Zone die Kontrolle übergibt. Das Beschreiben dieses Registers ist eine privilegierte Operation, ihre Ausführung heraus aus der ungeschützten Zone abgefangen (↑*trap*). Diese Operation ist nur dem Betriebssystem erlaubt (↑*operating mode*), da das Maschinenprogramm sonst die geschützte Zone auf den eigenen Adressbereich ausdehnen kann. Während der Ausführung von einem Maschinenprogramm ist die Gatteradresse fest. Bevor das nächste Maschinenprogramm geladen wird und zur Ausführung kommt, kann das Betriebssystem die Gatteradresse verändern, um sich mehr oder weniger Platz im Hauptspeicher zu geben. Da in dem Ansatz die Gatteradresse zur ↑Bindezeit eines Maschinenprogramms als undefiniert gilt, muss im Betriebssystem ein ↑verschiebender Lader tätig sein, um die Maschinenprogramme in die ungeschützte Zone des Hauptspeichers zu bringen. Dieser Lader nimmt die zur ↑Ladezeit gültige Gatteradresse als ↑Verlagerungskonstante und richtet das Maschinenprogramm entsprechend aus.

Schutzring (en.) ↑*protection ring*. Bezeichnung für eine bestimmte Art von ↑Schutzdomäne einer ↑Entität — beziehungsweise von einem ↑Subjekt oder ↑Objekt, je nachdem, ob eine aktive oder passive Haltung eingenommen wird. Der Ring bildet einen Mechanismus, um eine bestimmte Menge solcher Entitäten vor möglichen Auswirkungen von Fehlern (↑*safety*) oder böswilligem Verhalten (↑*security*) anderer Prozesse zu bewahren. Durch diesen Mechanismus wird ↑ringorientierter Schutz ermöglicht.

Schutzsystem (en.) ↑*protection system*. Gesamtheit von technischen Maßnahmen, um ↑Schutz zu gewährleisten.

Schutzverletzung (en.) ↑*protection fault*. ↑Aktion, die einen ↑Schutzfehler (insb. Verletzung von ↑Speicherschutz) verursacht.

schwache Konsistenz (en.) ↑*weak consistency*. Modell der ↑Speicherkonsistenz, für das eine von einem ↑Prozessor explizit einzurichtende Absperrung (↑*memory barrier*) zur ↑Synchronisation laufender Operationen auf dem ↑Arbeitsspeicher die Grundlage bildet. Die Absperrung bewirkt, dass alle ausstehenden Speicheroperationen aller Prozessoren zum Abschluss gebracht und neue Speicheroperationen erst nach erfolgter Synchronisation wieder ausgegeben werden.
Schwächer als ↑Prozessorkonsistenz. Allgemein aber auch die Bezeichnung für jede Form von Speicherkonsistenz, die schwächer ist als ↑sequentielle Konsistenz.

Schwebezustand Bezeichnung für den Zustand der Unklarheit, der Unsicherheit, der Unentschiedenheit über einen bestimmten ↑Prozesszustand (in Anlehnung an den Duden). Je nach ↑Betriebsart können für einen ↑Prozess bestimmte Zwischenzustände definiert sein, die Einfluss auf seine ↑Einplanung beziehungsweise ↑Einlastung ausüben.
Ermöglicht das ↑Betriebssystem die ↑Umlagerung der Inhalte von ↑Speicherbereichen, kann der Zustand des betreffenden Prozesses zusätzlich zu bereit oder blockiert jeweils auch schwebend (*pending*) sein. Dies ist dann der Fall, wenn sämtliche von einem ↑Prozessadressraum abgedeckten Bereiche im ↑Hauptspeicher komplett im ↑Hintergrundspeicher ausgelagert sind. Ein solcher Prozess kann zwar weiterhin eingeplant, aber nicht eingelastet werden: letzteres erfordert zunächst die Einlagerung seiner Hauptspeicherbereiche. Demgegenüber kann ein Prozess, der schwebend blockiert ist, sehr wohl in den Zustand schwebend bereit überführt werden, ohne dass er dazu erst eingelagert werden müsste. Diese Zustandsüberführung hat nämlich nur zur Folge, den (nicht umgelagerten) ↑Prozesskontrollblock des betreffenden Prozesses auf die ↑Bereitliste zu setzen.
Eine weitere Quelle von Zwischenzuständen ist die ↑mitlaufende Planung einerseits und die gleichzeitig dazu stattfindende Bereitstellung eines Prozesses andererseits, wobei letzteres entweder indirekt im Rahmen einer ↑Unterbrechungsbehandlung geschieht oder unterbrechungsfrei direkt von einem anderen ↑Prozessor oder ↑Rechenkern aus erfolgt. Vor solch einem Hintergrund können sich folgende Situationen ergeben:

- Ein Prozess, der zum weiteren Fortschritt erst den Eintritt eines bestimmten ↑Ereignisses erwarten muss (↑logische Synchronisation), wird in den Zustand blockiert übergehen. Diese ↑Aktion führt der Prozess selbständig durch, bevor er den Prozessor abgibt, das heißt, während er noch im Zustand laufend ist. Prozesse können demnach laufend blockiert sein.
- Ist in diesem Moment kein anderer Prozess im Zustand bereit, die Bereitliste also leer, kann der Prozess die Rolle als ↑Leerlaufprozess übernehmen. Anstatt den Prozessor abzugeben wird der Prozess untätig. Prozesse können demnach laufend blockiert und untätig sein.
- Ein als blockiert ausgewiesener Prozess kann in den Zustand bereit übergehen und damit auf die Bereitliste gelangen, nämlich sobald das von ihm erwartete Ereignis eintritt. Das kann insbesondere geschehen, obwohl es der Prozess noch nicht geschafft hat, den Prozessor abzugeben. Prozesse können demnach laufend bereit und, im Falle der Rolle als Leerlaufprozess, dabei auch noch untätig sein.

Entsprechend bietet es sich an, den Prozesszustand als eine *Menge* von Statusattributen und nicht als Aufzählung von Zahlenwerten zu kodieren. Ist diese Menge sodann kompakt als ↑Bitleiste gespeichert, sind vergleichsweise komplexe Mengenoperationen (↑*read-modify-write*) notwendig, nämlich zur Bildung der Vereinigungs- oder Schnittmenge, um einen bestimmten Zwischenzustand zu definieren (Disjunktion, `or`) beziehungsweise aufzuheben (Konjunktion, `and`). Bei Speicherung als *Zeichenfeld* (z.B., `char[4]`) entarten die Mengenoperationen jedoch zu einfachen Setz- und Rücksetzbefehlen (Zuweisung, `mov`). Letzteres ist gerade bei ↑Parallelverarbeitung besonders vorteilhaft, da die dann erforderlichen atomaren Mengenoperationen allein mit Wahrung der ↑Speicherkohärenz durch die ↑CPU gegeben sein können.

schwergewichtiger Prozess (en.) ↑*heavy-weight process*. Bezeichnung für einen ↑Prozess, der als ↑Systemkernfaden allein im eigenen und durch ↑Speicherschutz isolierten ↑Adressraum stattfindet. Auf ↑Multics zurückgehendes Verständnis einer ↑Prozessinkarnation, nämlich der Gleichsetzung von ↑Prozess und physisch abgeschottetem Adressraum.

scratchpad memory (dt.) ↑Notizblockspeicher.

scripting language (dt.) ↑Skriptsprache.

secondary storage (dt.) ↑Sekundärspeicher, ↑Hintergrundspeicher.

security (dt.) ↑Angriffssicherheit.

segfault Abkürzung für (en.) ↑*segmentation fault*.

Segment (en.) ↑*segment*. Unterteilung im ↑Prozessadressraum, wobei der ↑Adressbereich dieser Unterteilung auf den ↑Speicher eines Rechensystems abgebildet ist, genauer: dem ↑Vordergrundspeicher und ↑Hintergrundspeicher. In diesem Bereich liegt Programmtext oder -daten, auch bezeichnet als ↑Textsegment und ↑Datensegment. Ein solcher Adressraum kann mehrere dieser Bereiche umfassen, die sich nicht überlappen und von fester oder variabler Größe sind. Beispiele für Segmente sind grobkörnige Bereiche wie ganze Dateien oder der gesamte Text- oder Datenbereich von einem ↑Programm oder feinkörnige Gebilde wie ein einzelnes ↑Unterprogramm, Datenstrukturen, ↑Objekte oder ↑Variablen.

segment (dt.) Abschnitt, Teilstück; ↑Segment.

segment descriptor (dt.) ↑Segmentdeskriptor.

segment fault (dt.) ↑Segmentfehler.

segment name (en.) ↑Segmentname.

segment number (dt.) ↑Segmentnummer.

segment selector (dt.) ↑Segmentwähler.

segment table (dt.) ↑Segmenttabelle.

Segmentadressierung (en.) ↑*segmented addressing*. Siehe ↑Segmentierung.

Segmentadressierungseinheit (en.) ↑*segmentation unit*. Funktionsbaustein einer ↑MMU, der eine ↑logische Adresse in eine ↑reale Adresse umsetzt. Definitionsbereich der logischen Adresse ist ein ↑segmentierter Adressraum. Die Abbildung dieses Adressraums geschieht durch eine ↑Segmenttabelle, der Bildbereich stellt sich als ↑realer Adressraum dar.

segmentation (dt.) ↑Segmentierung, ↑Segmentumlagerung.

segmentation fault (dt.) ↑Segmentierungsfehler, ↑Speicherzugriffsfehler.

segmentation unit (dt.) ↑Segmentadressierungseinheit.

Segmentdeskriptor (en.) ↑*segment descriptor*. Bezeichnung für einen ↑Deskriptor von einem ↑Segment. Datenstruktur einer ↑MMU (↑*segmentation unit*), um eine ↑logische Adresse in eine ↑reale Adresse umzusetzen (↑Adressabbildung). Der Deskriptor kennzeichnet ein Segment einerseits durch den im ↑Hauptspeicher beanspruchten Platz (vgl. ↑Segmentregister) und andererseits durch ↑Speicherattribute, die die mit dem Segment erlaubten Operationen definieren sowie ihren Systemstatus festhalten.

segmented address space (en.) ↑segmentierter Adressraum.

segmented addressing (dt.) ↑Segmentadressierung.

segmented paging (dt.) ↑segmentierte Seitenadressierung.

segmented virtual memory (dt.) ↑segmentierter virtueller Speicher.

Segmentfehler (en.) ↑*segment fault*. Fehlgriff auf ein ↑Segment, verursacht einen ↑Bindungsfehler und hat ↑dynamisches Binden zur Folge. Zu dem Segment besteht eine ↑unaufgelöste Referenz, die im Moment der Verwendung eine ↑Ausnahme erhebt (↑*link trap*).

segmentierte Seitenadressierung (en.) ↑*segmented paging*. Methode der ↑Seitenadressierung bei der die ↑Seitentabellen segmentiert sind; Kombination von Seitenadressierung mit ↑Segmentierung. In dieser Variante der ↑Adressabbildung bildet jede Seitentabelle ein in einem eigenen ↑Segment liegendes *dynamisches* ↑*Feld* von ↑Seitendeskriptoren. Größen wie auch Anzahl dieser Segmente beziehungsweise Seitentabellen sind nach oben begrenzte variable Werte, die durch die Ausdehnung eines jeweiligen ↑Prozessadressraums bestimmt und über die ↑Laufzeit entsprechend veränderbar ist.
Anders als ↑seitennummerierte Segmentierung ist der Prozessadressraum selbst nach wie vor seitennummeriert, er ist nicht segmentiert. Seine linear angeordneten Bestandteile sind ↑Seiten, die nichtlinear auf ↑Seitenrahmen abgebildet im ↑Hauptspeicher liegen können und gegebenenfalls (↑*virtual memory*) auch der ↑Seitenumlagerung unterzogen sind.
Wie bei Seitenadressierung üblich ist jede vom ↑Prozessor generierte ↑logische Adresse a eine ↑lineare Adresse. Diese bildet ein Paar $a = (p, o)$, mit ↑Seitennummer p und ↑Versatz o innerhalb von Seite p. Im Unterschied zur gewöhnlichen Seitenadressierung ist jedoch p ebenfalls als Paar $p' = (s, d)$ dargestellt, mit ↑Segmentnummer s und ↑Verschiebung d innerhalb des Seitentabellensegments s, um den darin enthaltenen ↑Seitendeskriptor für p herauszugreifen. Damit ist die Abbildung einer logischen Adresse $a = ((s, d), o)$ auf ihre ↑reale Adresse a_r wie folgt definiert (vgl. S. 289):

$$a_r = \begin{cases} segmenttable[s].pagetable[d].pageframe * \mathcal{P} + o, & \text{falls } d < segmenttable[s].size \\ \uparrow segfault, & \text{sonst} \end{cases}$$

mit fester Seitengröße $\mathcal{P} = 2^w$ und $0 \leq o < \mathcal{P}$. Wie üblich ergeben sich die in a enthaltenen Werte für Seitennummer und Versatz zu $p = a \text{ div } \mathcal{P}$ und $o = a \bmod \mathcal{P}$, wobei $p = p' = (s, d)$ von zweiteiliger Struktur ist (vgl. S. 286). Dieser Struktur entsprechend gelten für die Verschiebung $0 \leq d < \mathcal{D}$ bei maximaler Segmentlänge $\mathcal{D} = 2^l$ und für die Segmentnummer $0 \leq s < \mathcal{S}$ bei fester Anzahl von Segmenttabelleneinträgen $\mathcal{S} = 2^n - \mathcal{D} - \mathcal{P}$, mit n als ↑Adressbreite von a. Dabei geben w und l jeweils die Länge der für o beziehungsweise d durch die ↑MMU festgelegten ↑Bitleisten in a vor.
Zur Dimensionierung wie auch Lokalisierung einer verschiebbaren (*relocatable*) Segmenttabelle dient ein ↑Segmentregister, dessen Inhalt und Verwendung unter Kontrolle des ↑Betriebssystems liegt. Dabei geschieht die Grenzwertüberprüfung für s entsprechend zur gewöhnlichen Segmentierung (vgl. S. 284).

segmentierter Adressraum (en.) ↑*segmented address space*. Bezeichnung für einen ↑Adressraum, der zweidimensional ausgelegt ist: die erste Dimension benennt das jeweilige ↑Segment und die zweite Dimension benennt die ↑Speicherzelle innerhalb dieses Segments. Entsprechend bildet eine ↑Adresse in solch einem Adressraum ein Zweitupel $\langle s, d \rangle$ mit ↑Segmentname s und ↑Verschiebung d innerhalb von Segment s. Dabei kann ein solches Segment als ↑seitennummerierter Adressbereich ausgelegt sein, vorausgesetzt das ↑Betriebssystem und die zugrundeliegende ↑MMU, die zur Abbildung des zweidimensionalen Adressraums erforderlich ist, unterstützt eine derartige Organisation (↑*paged segmentation*).

segmentierter virtueller Speicher (en.) ↑*segmented virtual memory*. Bezeichnung für einen als ↑virtueller Speicher implementierten ↑Arbeitsspeicher, dessen Strukturelement und kleinste Verwaltungseinheit das ↑Segment darstellt. Der gesamte Speicherbereich ist in Form solcher Segmente organisiert.

Segmentierung (en.) ↑*segmentation*. Zerlegung eines komplexen Ganzen in einzelne Abschnitte (in Anlehnung an den Duden). Jeder Abschnitt bildet ein eigenständiges ↑Segment, das für gewöhnlich ↑Text oder ↑Daten enthält, aber auch beliebige andere ↑Entitäten erfassen kann, solange diese *logisch* im ↑Hauptspeicher (genauer: ↑realer Adressraum) organisiert (↑*memory-mapped*) vorliegen. Die Segmente können nur logische, durch Software definierte Abschnitte bilden oder sie sind zusätzlich physisch existent. Letzteres erfordert gemeinhin spezielle funktionale Merkmale des ↑Betriebssystems und der Hardware (↑MMU), um die Segmente und schließlich die darin enthaltenen Entitäten zu adressieren. Das in einzelne Abschnitte, in Form physischer Segmente zerlegte komplexe Ganze bildet einen ↑Adressraum, auch als ↑segmentierter Adressraum bezeichnet.
Mit dieser Methode ist ein ↑Prozessadressraum selbst segmentiert. das heißt, jede vom ↑Prozess generierte ↑logische Adresse a bildet ein Zweitupel $\langle s, d\rangle$ mit ↑Segmentnummer s und ↑Verschiebung d innerhalb von Segment s. Damit ist a keine ↑lineare Adresse, sondern eine Zweikomponentenadresse. In solch einem Fall geschieht die ↑Adressabbildung, um aus der logischen Adresse a die korrespondierende ↑reale Adresse a_r herzuleiten, wie folgt:

$$a_r = \begin{cases} segmenttable[s].base + d, & \text{falls } d < segmenttable[s].size \\ \uparrow segfault, & \text{sonst} \end{cases}$$

Für gewöhnlich ist die ↑Segmenttabelle ein verschiebbares (*relocatable*) ↑Feld von ↑Segmentdeskriptoren, dessen Lage und, bei variabler räumlicher Ausdehnung, auch Größe im ↑Arbeitsspeicher durch ↑Segmentregister erfasst wird. Die Segmentnummer ist dann der Feldindex, um den ↑Deskriptor für Segment s aus dem Feld herauszugreifen. Dabei gelten für die Segmentnummer $0 \leq s < \mathcal{S}$ bei maximaler Anzahl von Segmenttabelleneinträgen $\mathcal{S} = 2^m$, mit $m \in \{3, 10, 13, 18\}$ (↑PDP 11, ↑B 5000, ↑x86 und ↑GE 645 entspr.) und für die Verschiebung $0 \leq d < \mathcal{D}$ bei maximaler Segmentlänge $\mathcal{D} = 2^n$, mit n gleich der ↑Adressbreite.
Im Falle eines dynamischen Felds führt die MMU gemeinhin ebenenfalls eine Grenzwertüberprüfung (*bounds check*) für s durch und greift einen Segmentdeskriptor nur heraus, falls $s < \mathcal{S}$. Stellt die MMU einen Indexfehler ($s \geq \mathcal{S}$) fest oder ist der ausgewählte Segmentdeskriptor nicht gültig (bspw. ausgewiesen durch ein spezielles Attribut (*invalid bit*) oder spezielle Werte für die Basisadresse oder Länge), zeigt sie einen ↑Segmentierungsfehler an. Bei einem statischen Feld wird ein solcher Fehler beim Feldzugriff nur bei ungültigen Deskriptoren signalisiert. In diesem Fall umfasst das Feld immer s Einträge.

Segmentierungsfehler (en.) ↑*segmentation fault*. Bezeichnung einer ↑Schutzverletzung durch einen ↑Prozess beim Zugriff mit einer ↑Adresse, die außerhalb von dem für sie definierten ↑Adressbereich liegt. Der Adressbereich beschreibt die für ein ↑Segment gültigen Adressen. Jede Adresse außerhalb des Segments wird im Zugriffsfall eine ↑Ausnahme herbeiführen.
Typischerweise werden solche Segmente durch eine ↑MMU oder ↑MPU abgesichert, die der ↑CPU im Falle eines ↑Zugriffsfehlers signalisiert, den in Ausführung befindlichen ↑Maschinenbefehl abzufangen (↑*trap*). Eine solche Schutzverletzung gilt allgemein als schwerwiegender ↑Fehler und zieht gegebenenfalls einen ↑Kernspeicherabzug nach sich.

Segmentname (en.) ↑*segment name*. Kennzeichnende Benennung eines ↑Segments, durch die es von anderen seiner Art unterschieden wird (in Anlehnung an den Duden). Für gewöhnlich ein aus Ziffern gebildeter ↑Name, auch ↑Segmentnummer.

Segmentnummer (en.) ↑*segment number*. Eine natürliche Zahl, die ein ↑Segment kennzeichnet; gemeinhin auch Synonym für ↑Segmentname.

Segmentregister (en.) ↑*base/limit register*. Vorkehrung mit der ein ↑Segment im ↑Hauptspeicher festgelegt wird. Spezielle ↑Prozessorregister, die als Paar die Lage (↑reale Adresse) und die Länge (Anzahl von ↑Speicherzellen) des Segments definieren: nämlich das Basisregister (*base register*) und das Längenregister (*limit register*). In dem Segment liegt entweder ein ↑Maschinenprogramm oder, falls ein ↑segmentierter Adressraum die ↑Prozessdomäne bildet, eine ↑Segmenttabelle, die verschiedene Segmente eines Maschinenprogramms erfasst: vereinfacht

ist die Segmenttabelle ein *dynamisches* ↑*Feld* von Paaren solcher in ↑Segmentdeskriptoren enthaltenen Adress- und Längenwerte.
Gegebenenfalls ist das Registerpaar im ↑Programmiermodell nicht direkt sichtbar, sondern nur indirekt durch spezielle ↑Maschinenbefehle zugänglich. Beispielsweise sorgt bei ↑x86 (ab ↑i386) nur ein ↑privilegierter Befehl, `lgdt` (*load global descriptor table register*), dafür, dem ↑Prozessor die Lage (*base*) und Länge (*limit*) der Segmenttabelle im ↑Hauptspeicher mitzuteilen. Dabei adressiert der Befehlsoperand ein sechs (32-Bit Modus: $4+2$) beziehungsweise zehn (64-Bit Modus: $8+2$) ↑Bytes großes ↑Objekt, das ebenfalls im Hauptspeicher liegt, um die gewünschten ↑Daten für die Basisadresse (32/64-Bit) und Länge (16-Bit, Byteanzahl) der Tabelle in den Prozessor zu laden. Für gewöhnlich besitzt dieses Objekt eine ↑logische Adresse (im ↑Betriebssystem), es beschreibt allerdings ein Objekt mit einer realen Adresse (*base*) im Hauptspeicher, nämlich die Segmenttabelle.
Liegt in dem durch das Registerpaar beschriebene Segment ein Maschinenprogramm, sind, im Gegensatz zu ↑Grenzregister, die Registerinhalte selbst für einen stattfindenden ↑Prozess (d.h., ↑Prozesszustand laufend) veränderlich und es ist auch kein ↑verschiebender Lader erforderlich, um ein als Segment ausgebildetes Maschinenprogramm in den Hauptspeicher zu bringen. Der entscheidende Grund dafür ist, dass jede vom Prozess generierte effektive ↑Adresse eine logische Adresse, a_l, darstellt, die mit Hilfe des Basisregisters zur ↑Laufzeit vom Prozessor erst noch auf eine reale Adresse abgebildet wird: $a_r = a_l + \textit{base register}$, mit $a_l = d$ auch als ↑Verschiebung relativ zur Basisadresse im Segment adressierten Speicherzelle bezeichnet. Der Basisregisterinhalt gilt in dem Fall als ↑Verlagerungskonstante zumindest während eines einzelnen ↑Abruf- und Ausführungszyklusses des Prozessors.
Die ↑Verlagerung einer logischen Adresse zur Laufzeit geschieht jedoch nur, wenn die betreffende Adresse gültig ist, das heißt, eine Speicherzelle in dem Segment referenziert. Vor jeder Verlagerung muss für die logische Adresse a_l gelten: $0 \leq a_l < \textit{limit register}$. Ist der Adresswert größer oder gleich dem Inhalt des Längenregisters, wird ein Zugriff über diese Adresse abgefangen (↑*segmentation fault*). Jedes solcher Segmente ist damit ein ↑logischer Adressraum mit Wertebereich $[0, N-1]$, wobei N die Länge des jeweiligen Segments angibt.
Zur Erzeugung von dem ↑Lademodul für ein Maschinenprogramm weist der ↑Binder jedem ↑Symbol dieses Programms eine zur Basis 0 relative Adresse zu. Um das Maschinenprogramm zur Ausführung zu bringen, fordert der ↑Lader die ↑Speicherzuteilung an und erhält im Erfolgsfall eine ↑Ladeadresse, die gleichfalls Basisadresse des Segments dieses Programms ist. An diese Ladeadresse wird das Maschinenprogramm in den Hauptspeicher platziert. Für die daraufhin (vom Lader) eingerichtete ↑Prozessinkarnation werden im ↑Prozesskontrollblock Basisadresse und Länge des entsprechenden Segments als Attribute verbucht. Bei einem ↑Prozesswechsel werden diese Attribute in das Basis- und Längenregister geschrieben.
Da dieser Schreibzugriff beim Prozesswechsel als privilegierte Operation im ↑Betriebssystem erfolgt (↑*privileged mode*), müssen im Falle eines durch ein eigenes Segment geschütztes Betriebssystem für den System- und Benutzermodus jeweils eigene Basis-/Längenregister vorhanden sein (↑Arbeitsmodus). Beim Prozesswechsel werden die entsprechenden Segmentattribute (Basis, Länge) in die dem Benutzermodus zugeordneten Basis-/Längenregister geschrieben, die aber erst nach Verlassen des Systemmodus zur Wirkung kommen. Grundsätzlich werden mit jedem Wechsel vom Benutzer- zum Systemmodus und umgekehrt die dem jeweiligen Modus zugeordneten Basis-/Längenregister im Prozessor wirksam. Ist das Betriebssystem dagegen nicht im eigenen Segment isoliert, sind die Basis-/Längenregister nur einfach ausgelegt. In dem Fall ist die Überprüfung der logischen Adresse (*limit check*) im Systemmodus entweder wirkungslos oder abgeschaltet, die Basis-/Längenregister kommen dann nur im Benutzermodus zur Wirkung.

Segmenttabelle (en.) ↑*segment table*. Datenstruktur, durch die ein ↑segmentierter Adressraum definiert wird. Eine listenförmige Zusammenstellung von ↑Segmentdeskriptoren, für gewöhnlich repräsentiert als ↑Feld. Die Anzahl der mit dieser Datenstruktur erfassten ↑Segmente ist abhängig von dem ↑Prozess, der einen solchen ↑Adressraum ausfüllt. Zudem gibt es für jeden Prozess eine eigene Datenstruktur dieser Art, die beim ↑Prozesswechsel sodann ebenfalls

umgeschaltet werden muss. Die ↑MMU lokalisiert und dimensioniert diese Datenstruktur für gewöhnlich durch ein ↑Segmentregister, dessen Inhalt unter Kontrolle des ↑Betriebssystems steht und beim Prozesswechsel aktualisiert wird.

VL 6.2, 6.3 oder 12.1

Segmentumlagerung (en.) ↑*segmentation*. Funktion in einem ↑Betriebssystem, durch die ein bei der Ausführung von einem ↑Programm referenziertes, aber nicht im ↑Hauptspeicher (Vordergrund) vorliegendes ↑Segment vom ↑Hintergrundspeicher automatisch eingelagert wird. Wenn noch Platz für die Einlagerung zu schaffen ist, werden gegebenenfalls ein oder mehrere im Hauptspeicher vorliegende Segmente vorher auf den Hintergrundspeicher ausgelagert. Grundlage dafür bildet ↑segmentierter virtueller Speicher und die ↑Segmentadressierung. Für den anfallenden Transfer aller betroffenen Segmente im Vorder- und Hintergrundspeicher sorgt das Betriebssystem.

Segmentwähler (en.) ↑*segment selector*. Spezielles ↑Prozessorregister bei ↑x86, das unter anderem einen ↑Segmentnamen sowie, ab Modell ↑i286, die mit dem betreffenden ↑Segment verbundene Privilegstufe (↑DPL) speichert.

Seite (en.) ↑*page*. ↑Adressbereich fester Größe, die immer eine Zweierpotenz bildet und ganzzahlige Vielfache der Größe eines ↑Speicherwort ist und heute (2025) für gewöhnlich einer ↑Oktettanzahl entspricht. Die effektive Größe hängt ab von verschiedenen Faktoren: Umfang der ↑Seitentabelle, verfügbarer Platz im ↑Übersetzungspuffer, ↑Verschnitt und der ↑Zugriffszeit auf den ↑Hintergrundspeicher.
In vielen Fällen ist die Seitengröße daher in der ↑MMU einstellbar, typische Werte sind 4, 8 oder 16 KiB (für ↑Linux, ↑macOS oder ↑Windows und mehr (z.B. bis zu 256 MiB bei ↑Prozessoren mit einer ↑Adressbreite von 64 Bits).

Seiten-Kachel-Tabelle siehe ↑Seitentabelle.

Seitenabrufer (en.) ↑*pager*. ↑Programm zur ↑Seitenumlagerung. Intern im ↑Betriebssystem (Systemebene) oder extern dazu oberhalb (Benutzerebene), gegebenenfalls sogar in jedem ↑Maschinenprogramm, angesiedelte Funktion, mit deren Hilfe ↑virtueller Speicher durchgesetzt wird. Abgerufen werden Programmteile aus dem ↑Hintergrundspeicher (Einlagerung) oder dem ↑Vordergrundspeicher (Auslagerung), jeweils im Moment des Zugriffs durch einen ↑Prozess oder vorausschauend. Einheit dafür ist eine ↑Seite, die einzeln oder als Teil einer Folge transferiert wird.

Seitenadressierung (en.) ↑*paging*. Methode einer auf ↑Seiten als Strukturelement basierenden ↑Adressabbildung, für gewöhnlich in Hardware (↑*paging unit*) realisiert durch eine unter Kontrolle eines ↑Betriebssystem stehenden ↑MMU. Der Abbildung liegt grundsäzlich ein ↑seitennummerierter Adressraum zugrunde.
Bei dieser Methode ist ein ↑Prozessadressraum selbst seitennummeriert. Dies bedeutet, jede vom ↑Prozess generierte ↑logische Adresse a bildet ein Paar (p, o) mit ↑Seitennummer p und ↑Versatz o innerhalb von Seite p. Damit ist a eine ↑lineare Adresse, im Gegensatz zur ↑Segmentierung. In solch einem Fall geschieht die Adressabbildung, um aus der logischen Adresse a die korrespondierende ↑reale Adresse a_r herzuleiten, wie folgt:

$$a_r = pagetable[p].pageframe * \mathcal{P} + o$$

mit fester Seitengröße $\mathcal{P} = 2^w$ und $0 \leq o < \mathcal{P}$, wobei w der Länge der für o durch die MMU festgelegten ↑Bitleiste in a entspricht (typisch ist $m = 12$, das heißt, eine Seitengröße von 4096 ↑Bytes). Die in $a = (p, o)$ enthaltenen Werte für die Seitennummer innerhalb des Prozessadressraums einerseits und dem Versatz (↑Oktettnummer) innerhalb der Seite andererseits ergeben sich zu $p = a \text{ div } \mathcal{P}$ und $o = a \bmod \mathcal{P}$. Dabei entsprechen die beiden Divisionsoperationen der (von der MMU durchgeführten) Extraktion von p und o aus a mit den Masken $(2^n - 1) - (\mathcal{P} - 1)$ für div und $\mathcal{P} - 1$ für mod, mit n als ↑Adressbreite.
Bei dem Ansatz hat die Seitentabelle grundsätzlich eine feste Größe, die durch die Obergrenze des Wertebereichs $[0, 2^n - \mathcal{P} - 1]$ für die Seitennummer p bestimmt und über die ↑Laufzeit

unveränderbar ist. Beim Indizieren in die Seitentabelle überprüft die MMU für gewöhnlich nicht die Indexgrenze für die Seitennummer p. Damit die MMU die Abbildung ohne Risiko eines ↑Fehlers für das ↑Rechensystem leisten kann, muss demzufolge die Seitentabelle immer vollumfänglich als ↑Exemplar vorhanden sein, auch wenn ein Prozessadressraum nur einen Bruchteil der durch diese Datenstruktur lokalisierbaren Seiten umfasst.
Für große Werte dieser Obergrenze bietet sich daher eine mehrstufige Abbildung an, und zwar über kleinere, dafür aber mehrere Tabellen in Abhängigkeit von der Stufenanzahl. Letztere ist abhängig von der Adressbreite n beziehungsweise der durch $(2^n - 1) - (\mathcal{P} - 1)$ definierten Bitleistenbreite in der abzubildenden Adresse a. Bei $\mathcal{P} = 4096$ und $n = 32$ ist ein zweistufiges Modell üblich, wohingegen bei $n = 64$ vier oder gar fünf Stufen pro Prozessadressraum implementiert sind. In der mehrstufigen Variante ist zunächst die Seitentabelle der letzten Stufe zu lokalisieren, um dann den effektiven Seitendeskriptor für den ↑Text- oder ↑Datenbestand eines ↑Prozesses herauszugreifen. Dies geschieht durch Indexoperationen auf vorgeschalteten Seitentabellen, wobei für jede Stufe eine Untergliederung der Seitennummer p vorgenommen wird, um die Indexwerte der Tabelle der jeweiligen Stufe zu erhalten. Für gewöhnlich bilden dazu die Abschnitte, in die p untergliedert sind, gleich große Bitleisten (bspw. 10 Bits für $n = 32$ oder 9 Bits für $n = 64$, wie bei ↑Intel 64).
Angenommen, der Abbildung liegt ein zweistufiges Verfahren zugrunde. In dem Fall ist eine Seitennummer dargestellt als Paar $p = (t, e)$, mit t als Nummer der Seitentabelle der zweiten Stufe und e dem Eintrag (↑Seitendeskriptor) in Tabelle t. Die reale Adresse für $a = ((t, e), o)$ lässt sich dann wie folgt bilden:

$$a_r = tabledirectory[t].pagetable[e].pageframe * \mathcal{P} + o$$

mit fester Länge für die benötigten Tabellen $\mathcal{T} = 2^l$ und $0 \leq l < \mathcal{T}$. Die Länge l der für t und e festgelegten Bitleisten in p ist durch die MMU festgelegt, ein typischer Wert (bei $\mathcal{P} = 4096$) ist $l = 10$. Die in $p = (t, e)$ enthaltenen Werte für das Tabellenverzeichnis (*table directory*) einerseits und dem Eintrag in der Seitentabelle andererseits ergeben sich zu $t = p \text{ div } \mathcal{T}$ und $e = p \bmod \mathcal{P}$, was wiederum einer einfachen Extraktion von zwei Bitleisten entspricht und dem Muster zur oben behandelten Herleitung der Werte für p und o folgt.
Dadurch kann der Bedarf an Speicherplatz für die zur Abbildung benötigten und in einer Seitentabelle aufgelisteten Seitendeskriptoren eines Prozessadressraums erheblich reduziert werden. Sei $n = 32$ und $w = 12$ (d.h., eine 32-Bit MMU, ein 32-Bit ↑Prozessor, mit einer typischen Seitengröße von 4 KiB), dann wäre p als eine $n - w = 20$ Bits umfassende Bitleiste in der Hardware implementiert. Für eine einstufige Abbildung müssten in der Seitentabelle sodann 2^{20} Seitendeskriptoren vorgehalten sein, auch wenn der betreffende Prozessadressraum nicht mehr als drei Seitendeskriptoren benötigt, um ↑Text, ↑Daten und den ↑Laufzeitstapel abzubilden — und dies für jedes Prozessexemplar (↑*heavy-weight process*). Demgegenüber sind bei einer zweistufigen Abbildung und angenommener gleichmäßigen Aufteilung der Bitleiste für p in jeweils 10 Bits für t und e nur $4 * 2^{10} = 4096$ Seitendeskriptoren nötig, also 2^{10} Stück jeweils für das Tabellenverzeichnis der ersten Stufe und die darüber erreichbaren drei Seitentabellen der zweiten Stufe für Text, Daten und Laufzeitstapel — sofern letztere Bereiche getrennt voneinander durch eigene Seitentabellen erfasst werden sollen. Sieht das durch das Betriebssystem vorgegebene ↑Adressraummodell dagegen keine solche Trennung vor, sind lediglich $2 * 2^{10} = 2048$ Seitendeskriptoren vorzuhalten.
Mit diesem Ansatz werden Seitentabellen intabuliert und belegen auch nur dann Speicherplatz, wenn sie im (übergeordneten) Tabellenverzeichnis verbucht sind. In Hinblick auf den kompletten Prozessadressraum deckt jede Seitentabelle (der zweiten Stufe) nur ein Teilstück eines größeren Ganzen ab, wobei all diese Teilstücke allerdings gleich groß sind und jeweils Platz für 2^e Einträge von Seitendeskriptoren haben. Ungenutzte Tabelleneinträge, das heißt, ungültige Einträge ausgewiesen durch ein spezielles Attribut (*invalid bit*) oder einen speziellen Wert für die Seitenrahmennummer bewirken einen ↑Zugriffsfehler (↑*segmentation fault*), wenn sie im Zuge der Adressabbildung herausgegriffen werden.
Eine andere Sicht ergibt sich, wenn ein Prozessadressraum nur durch ein einziges solches

Teilstück komplett beschrieben ist. Die Teilstücke, das heißt, die Seitentabellen können damit dann auch verschieden groß sein, je nach Umfang des jeweiligen Prozessadressraums. Mit anderen Worten ist ein solches Teilstück dann ein ↑Segment von Seitendeskriptoren: die Seitentabellen sind segmentiert (↑*segmented paging*). Im einfachsten Fall lässt sich dazu die Seitentabelle eines Prozessadressraums durch ein ↑Segmentregister vollständig erfassen. Bei segmentierten Seitentabellen überprüft die MMU im Zuge der Adressabbildung die Indexgrenze für die Seitennummer. Überschreitet die Seitennummer diese Grenze und damit die Länge des Seitentabellensegments, signalisiert die MMU einen ↑Segmentierungsfehler.

Seitenadressierungseinheit (en.) ↑*paging unit*. Funktionsbaustein einer ↑MMU, der eine ↑logische Adresse in eine ↑reale Adresse umsetzt. Definitionsbereich der logischen Adresse ist ein ↑seitennummerierter Adressraum. Die Abbildung dieses Adressraums geschieht tabellenbasiert (↑*page table*), der Bildbereich stellt sich als ↑realer Adressraum dar.

Seitendeskriptor (en.) ↑*page descriptor*. Bezeichnung für einen ↑Deskriptor von einer ↑Seite. Datenstruktur einer ↑MMU (↑*paging unit*), um eine ↑logische Adresse in eine ↑reale Adresse umzusetzen (↑Adressabbildung). Der Deskriptor kennzeichnet eine Seite einerseits durch ihren im ↑Hauptspeicher beanspruchten Platz (↑Seitenrahmen) und andererseits durch ↑Speicherattribute, die die mit der Seite erlaubten Operationen definieren sowie ihren Systemstatus festhalten.

Seiteneffekt (en.) ↑*side effect*. Nebenwirkung einer in einem ↑Unterprogramm formulierten ↑Aktion, die außerhalb ihres Gültigkeitsbereichs eine Zustandsveränderung herbeiführt. Darunter fällt auch jede für den ↑Prozess beobachtbare Interaktion mit dem aufrufenden ↑Programm oder der „Außenwelt“ neben der Rückgabe regulärer (deklarierter/definierter) Parameter.

Seitenfehler (en.) ↑*page fault*. Art von ↑Zugriffsfehler. Grundlage bildet ↑seitennummerierter virtueller Speicher. Im Moment des Zugriffs auf ein ↑Speicherwort im virtuellen Speicher stellt der ↑Prozessor fest, dass die ↑Seite, die dieses Wort umfasst, nicht im ↑Hauptspeicher eingelagert ist. Der Prozessor stellt daraufhin eine ↑Ausnahmesituation fest, die vom Betriebssystem (↑Seitenabrufer) behandelt wird und zur ↑Seitenumlagerung führt: die Seite, auf der das referenzierte Wort steht, wird eingelagert.

Seitenfehlerwahrscheinlichkeit (en.) ↑*page fault probability*.. VL 12.3

Seitenflattern (en.) ↑*page thrashing*. ↑Flattern einer ↑Seite (↑*virtual memory*).

Seitennummer (en.) ↑*page number*. Eine natürliche Zahl, die eine ↑Seite kennzeichnet.

seitennummerierte Segmentierung (en.) ↑*paged segmentation*. Methode der ↑Segmentierung bei der die ↑Segmente seitennummeriert sind; Kombination von Segmentierung mit ↑Seitenadressierung. In dieser Variante der ↑Adressabbildung sind nicht ↑Bytes die linear angeordneten Bestandteile eines Segments, sondern ↑Seiten, die nichtlinear auf ↑Seitenrahmen abgebildet im ↑Hauptspeicher liegen und gegebenenfalls (↑*virtual memory*) auch der ↑Seitenumlagerung unterzogen sind.
Anders als ↑segmentierte Seitenadressierung ist der ↑Prozessadressraum selbst nach wie vor segmentiert — je nach technischer Auslegung (s. nachfolgender Abschnitt) gilt dies aber dennoch auch für die Seitentabelle. Wie bei Segmentierung üblich ist damit jede vom ↑Prozess generierte ↑logische Adresse a ein Zweitupel $\langle s, d\rangle$ mit ↑Segmentnummer s und ↑Verschiebung d innerhalb von Segment s (vgl. auch S. 284): a ist keine ↑lineare Adresse.
Diese Variante der Adressabbildung kennt zwei verschiedene technische Auslegungen. In der einen Auslegung lokalisiert und dimensioniert der ↑Segmentdeskriptor die ↑Seitentabelle des betreffenden Segments (↑GE 645). Damit ist jede Seitentabelle ein *dynamisches* ↑*Feld* von ↑Seitendeskriptoren — sie ist segmentiert. Demzufolge hat eine Seitentabelle eine variable Größe, die durch die Ausdehnung eines jeweiligen Segments bestimmt und über die ↑Laufzeit entsprechend veränderbar ist. Im Unterschied zur gewöhnlichen Segmentierung ist jedoch die Verschiebung innerhalb eines Segments, d, als Paar (p, o) dargestellt, mit ↑Seitennummer

p und ↑Versatz o innerhalb der Seite p. Damit ist die Abbildung einer logischen Adresse $a = \langle s, (p, o) \rangle$ auf ihre ↑reale Adresse a_r wie folgt definiert (vgl. S. 283):

$$a_r = \begin{cases} segmenttable[s].pagetable[p].pageframe * \mathcal{P} + o, & \text{falls } p < segmenttable[s].size \\ \uparrow segfault, & \text{sonst} \end{cases}$$

mit fester Seitengröße $\mathcal{P} = 2^w$, wobei m der Länge der für o durch die ↑MMU festgelegten ↑Bitleiste in d entspricht, und $0 \leq o < \mathcal{P}$. Wie bei Seitenadressierung üblich ergeben sich die in $d = (p, o)$ enthaltenen Werte für Seitennummer und Versatz zu $p = d \operatorname{div} \mathcal{P}$ und $o = d \bmod \mathcal{P}$ (vgl. S. 286). Für die Segmentnummer gilt $0 \leq s < \mathcal{S}$ bei maximaler Anzahl von Segmenttabelleneinträgen $\mathcal{S} = 2^m$, mit m als hardwareabhängigen Wert (vgl. 284).
In der anderen Auslegung lokalisiert und dimensioniert der Segmentdeskriptor ein Segment im seitennummerierten Prozessadressraum (↑x86 ab ↑i386). Hierzu wird zunächst in einem ersten Schritt die (zweidimensionale) logische Adresse $a = \langle s, d \rangle$ von einer ↑Segmentadressierungseinheit aufgelöst und in eine (eindimensionale) lineare Adresse a_l übersetzt, wie es von der reinen Segmentierung her bekannt ist (vgl. S. 284):

$$a_l = \begin{cases} segmenttable[s].base + d, & \text{falls } d < segmenttable[s].size \\ \uparrow segfault, & \text{sonst} \end{cases}$$

Diese lineare Adresse wird sodann in einem zweiten Schritt einer ↑Seitenadressierungseinheit zugeführt, die a_l als Paar (p, o) dargestellt deutet und in die reale Adress a_r entsprechend der bei Seitenadressierung üblichen Vorgehensweise übersetzt (vgl. S. 286):

$$a_r = pagetable[p].pageframe * \mathcal{P} + o$$

Im Gegensatz zur ersten Auslegung, die eine Seitentabelle durch einen Segmentdeskriptor erfasst und ihr damit Dynamik verleiht, ist hier nun jede Seitentabelle ein *statisches Feld* von Seitendeskriptoren. Dieses Feld hat eine feste Größe, die durch die Obergrenze $2^n - \mathcal{P}$ (n gleich der ↑Adressbreite) für die Seitendeskriptoranzahl bestimmt, über die Laufzeit unveränderbar ist und gegebenenfalls mehrstufig ausgelegt sein muss (vgl. S. 287).
Demgegenüber hat die Segmenttabelle (für beide Auslegungsarten) für gewöhnlich eine variable Größe. Zur Dimensionierung wie auch Lokalisierung dieser Tabelle im ↑Arbeitsspeicher dient ein ↑Segmentregister, dessen Inhalt und Verwendung unter Kontrolle des ↑Betriebssystems liegt. Dabei geschieht die Grenzwertüberprüfung für s entsprechend zur gewöhnlichen Segmentierung (vgl. S. 284).
Zu beachten dabei ist der feine Unterschied, dass im ersten Ansatz die ↑Seite und im zweiten Ansatz immer noch das ↑Byte (↑Oktett) das Strukturelement eines Segments bildet: letzterer erlaubt somit die Aufteilung desselben Seitenrahmens auf zwei Segmente, indem ein vorderer Abschnitt im Seitenrahmen (der letzten Seite) am Ende des einen Segments und der restliche hintere Abschnitt dieses Seitenrahmens (in der ersten Seite) am Anfang des anderen Segments liegt — sofern von der MMU unterstützt (z.B. i386 im 16-Bit Schutzmodus). Mit solch einer Option ließe sich ↑interne Fragmentierung vermeiden, da der Rest des letzten Seitenrahmens eines Segments nicht mehr brachliegen müsste.

seitennummerierter Adressbereich (en.) ↑*paged address range*. Ein bestimmter ↑Adressbereich, dessen Strukturelement eine ↑Seite bildet, der demzufolge mit fortlaufenden ↑Seitennummern versehen ist und nicht, wie sonst üblich, allein ↑Oktettnummern als Ordnungskriterium ansetzt. Die Größe dieses Bereichs ist ganzzahlige Vielfache der Größe einer Seite, wobei die Seitengröße durch die zugrundeliegende ↑MMU vorgegeben ist.

seitennummerierter Adressraum (en.) *paged* ↑*address space*. Ein ↑seitennummerierter Adressbereich, der vollumfänglich einen ↑Adressraum erfasst. Dieser Adressraum ist eindimensional, der in der Länge fest durch die ↑Adressbreite eingegrenzt ist. Nach außen sichtbar sind ↑lineare Adressen von ↑Speicherzellen oder ↑speicherabgebildeter ↑Entitäten endlicher Anzahl,

jedoch ist sein wesentliches inneres Strukturelement die ↑Seite. Die Lokalisierung dieser ↑Objekte geschieht durch ↑Seitenadressierung.
Die innere Gliederung eines solchen Adressraums ist durch die von einer ↑MMU vorgegebene Seitengröße bestimmt. Zur Bestimmung der Seite, auf der das zu lokalisierende Objekt liegt, wird eine ↑Seitentabelle verwendet. Diese Tabelle liegt als ↑Feld von ↑Seitendeskriptoren im ↑Arbeitsspeicher, die zentrale Datenstruktur für die zur Lokalisierung der Objekte erforderliche ↑Adressabbildung.

seitennummerierter virtueller Speicher (en.) ↑*paged virtual memory*. Bezeichnung für einen als ↑virtueller Speicher implementierten ↑Arbeitsspeicher, dessen Strukturelement und kleinste Verwaltungseinheit die ↑Seite darstellt. Der gesamte Speicherbereich ist linear in Form solcher Seiten organisiert.

Seitenrahmen (en.) ↑*page frame*. Abschnitt fester Größe im ↑Hauptspeicher (↑realer Adressraum) zur Aufnahme von exakt einer ↑Seite: die effektive Rahmengröße ist definiert durch die Seitengröße. Die Nummer eines solchen Rahmens bildet die ↑reale Adresse einer für gewöhnlich nur durch eine ↑logische Adresse oder ↑virtuelle Adresse erreichbaren Seite. Im Hintergrund steht ein ↑seitennummerierter Adressraum, dessen einzelne Seiten auf korrespondierende Seitenrahmen abzubilden sind. Die hierfür nötige Adressumsetzung leistet eine ↑MMU, die dazu vom ↑Betriebssystem vorher entsprechend programmiert werden muss (↑Seitentabelle).

Seitenrahmennummer (en.) ↑*page frame number*. Eine natürliche Zahl, die eindeutig einen ↑Seitenrahmen kennzeichnet. Multipliziert mit der Größe einer ↑Seite ist das Ergebnis die ↑reale Adresse, an der eine Seite im (physischen, realen) ↑Adressraum platziert werden kann. Diese Adresse verweist für gewöhnlich auf eine Stelle im ↑Hauptspeicher, sie kann allerdings auch einen Bereich für ↑speicherabgebildete Dinge (insb. ↑Ein-/Ausgaberegister) identifizieren.

Seitentabelle (en.) ↑*page table*. Datenstruktur einer ↑MMU, durch die ein bestimmter, *linear* ↑seitennummerierter Adressbereich definiert wird und diesen auf meist nichtlinear angeordnete ↑Speicherbereiche im ↑Arbeitsspeicher abbildet. Der Adressbereich kann einen kompletten ↑Prozessadressraum oder nur einzelne Bestandteile (d.h., ↑Segmente) davon erfassen. Dazu bildet diese Datenstruktur eine listenförmige Zusammenstellung von ↑Seitendeskriptoren, für gewöhnlich repräsentiert als ↑Feld.
Die Tabelle liegt verschiebbar (*relocatable*) im Arbeitsspeicher an einer in aller Regel vom ↑Betriebssystem bestimmten ↑Adresse, die zur Erfassung eines vollständigen Prozessadressraums für gewöhnlich in einem speziellen Adressregister (z.B. CR3 bei ↑x86 ab ↑i386) des ↑Prozessors gehalten wird. Impliziert ein ↑Prozesswechsel auch einen Adressraumwechsel, ist der Inhalt dieses Adressregisters entsprechend mit der Anfangsadresse der betreffenden Tabelle des zu aktivierenden Prozessadressraums zu aktualisieren. Definiert diese Tabelle nur einen Teil des Prozessadressraums, ist ihre Adresse für gewöhnlich in einer übergeordneten Datenstruktur (derselben oder anderer Art, einer ↑Segmenttabelle) intabuliert.
Die maximale Länge des die Tabelle repräsentierenden Felds ist bestimmt einerseits durch die von der MMU vorgegebenen Größe einer ↑Seite als Zweierpotenz und andererseits aus der ↑Adressbreite der ↑CPU. Sei n diese Adressbreite und $A = [0, 2^n - 1]$ der durch die CPU definierte ↑Adressraum. Sei ferner $w = log_2(\mathcal{P})$ die Adressbreite, um ein beliebiges ↑Oktett (für gewöhnlich eine ↑Speicherzelle, aber auch andere ↑speicherabgebildete Dinge) innerhalb einer exakt $\mathcal{P} = 2^w$ ↑Bytes großen Seite zu selektieren. Dann umfasst dieser Adressraum genau 2^{n-w} Seiten. Die innere Struktur einer Adresse $a \in A$ ist dann dargestellt als Paar $a = (p, o)$, mit ↑Seitennummer $p \in [0, 2^{n-w} - 1]$ und ↑Versatz $o \in [0, 2^w - 1]$ innerhalb von Seite p. Bei der ↑Seitenadressierung ist sodann p der Tabellenindex, um in dem Feld den Seitendeskriptor für Seite p zu selektieren:

$$\Delta = \begin{cases} pagetable[p], & \text{falls } pagetable[p] \text{ gültig ist} \\ \uparrow segfault, & \text{sonst} \end{cases}$$

Für einen ungültigen Tabelleneintrag ist der Seitendeskriptor entweder mit einem entsprechenden Hinweis (*invalid bit*) etikettiert oder mit einer speziellen ↑Seitenrahmennummer

versehen. Jedenfalls wird beim Zugriff auf einen ungültigen Eintrag durch die MMU ein ↑Segmentierungsfehler hervorgerufen (↑*trap*), der gemeinhin zum Abbruch des betreffenden ↑Prozesses durch das Betriebssystem führt.
Ein typischer Wert für w ist 12, was eine Seitengröße von $2^{12} = 4096$ Bytes ergibt. Bei einer Adressbreite von $n = 32$ Bits resultiert dies in $2^{n-w} = 2^{20}$ Seiten, die allein ein einziger Prozessadressraum umfassen kann. Entsprechend würden 2^{20} Seitendeskriptoren erforderlich sein, um all diese Seiten auf Hauptspeicherplätze abzubilden — ↑virtueller Speicher vorausgesetzt. Das heißt, das Feld von Seitendeskriptoren müsste 1 048 576 Einträge aufweisen. Eine typische Größe für den Seitendeskriptor eines 32-Bit Prozessors sind 32 Bytes: damit beansprucht das Feld 32 MiB — pro Prozessadressraum. Bei einer Adressbreite von $n = 64$ Bits hätte die Feldgröße ein riesiges Ausmaß, auch wenn die Seitengröße (z.B. auf 8 oder 16 KiB, wie für ↑Windows beziehungsweise ↑Linux) zunehmen würde.
Allerdings beansprucht ein Prozessadressraum in Wirklichkeit eher selten 2^{n-w} Seiten, schon gar nicht für $n = 64$. Die Tabelle ist häufig nur dünn besetzt mit solchen Seitendeskriptoren, die eine für den betreffenden Prozessadressraum gültige Abbildung von Seiten auf den Arbeitsspeicher definiert. Um die Tabellenlänge und damit den Platzbedarf für das Deskriptorfeld besser auf die wirkliche, effektive Größe eines Prozessadressraums abzustimmen, entwickelten sich verschiedene Darstellungsformen, die damit auch Varianten der ↑Seitenadressierung nach sich zogen. So kann die Tabelle segmentiert (↑*segmented paging*), mehrstufig (S. 287) oder nach ↑Seitenrahmen (↑*inverted page table*) organisiert sein.

Seitenumlagerung (en.) ↑*paging*. Funktion in einem ↑Betriebssystem, durch die eine bei der Ausführung von einem ↑Programm referenzierte, aber nicht im ↑Hauptspeicher (Vordergrund) vorliegende ↑Seite vom ↑Hintergrundspeicher automatisch eingelagert wird. Wenn noch Platz für die Einlagerung zu schaffen ist, wird dazu gegebenenfalls eine im Hauptspeicher vorliegende Seite vorher auf den Hintergrundspeicher ausgelagert. Grundlage dafür bildet ↑seitennummerierter virtueller Speicher und die ↑Seitenadressierung. Für den anfallenden Transfer aller betroffenen Seiten im Vorder- und Hintergrundspeicher sorgt das Betriebssystem.

Seitenverfahren (en.) ↑*paging*. Eine Methode zur Verwaltung des — gegebenenfalls (↑*paged segmentation*) in mehrere Teilstücke verschiedener Länge (logisch) zerlegten, segmentierten — ↑Arbeitsspeichers durch das ↑Betriebssystem auf Grundlage der ↑Seitenadressierung und unter Einbeziehung von ↑Seitenumlagerung. Dabei sind alle verwalteten ↑Speicherbereiche von einheitlicher Länge, je nach ↑Abstraktionsebene bezeichnet als ↑Seiten (↑*virtual address space*) oder ↑Seitenrahmen (↑*physical address space*).

Seitenvorabruf (en.) ↑*pre-paging*. ↑Seitenumlagerung, die vorausschauend funktioniert und nicht erst im Moment des Zugriffs auf eine ↑Speicherstelle. Das ↑Betriebssystem hat exaktes oder heuristisches Wissen darüber, welche ↑Seite als nächste bei der Ausführung von einem ↑Programm referenziert wird.

Sekundärspeicher (en.) ↑*secondary storage*. Erste Stufe ↑Massenspeicher. Klassifikation für die ↑Ablage; „zweitrangiger Speicher“, der zwar über eine große Kapazität verfügt, dafür jedoch auch eine hohe ↑Zugriffszeit mit sich bringt.

selbstmodifizierender Kode (en.) ↑*self-modifying code*. Ein bestimmter Abschnitt in einem ↑Programm, der Anweisungen enthält, bei deren Ausführung Teile des dem Abschnitt eigenen Kodes gezielt verändert werden. Für gewöhnlich liegen solche Abschnitte in ↑Assembliersprache formuliert vor, um den ↑Maschinenkode einzelner ↑Maschinenbefehle modifizieren zu können. Die Selbstmodifikation ist beabsichtigt und deutet weder auf ↑Fehler noch auf ↑Schadsoftware in dem betreffenden Programm hin. Mit ihr lassen sich beispielsweise Einschränkungen im ↑Befehlssatz überwinden oder wiederholte, überflüssige Anweisungen zum Testen einer einmaligen Bedingung (*first-time condition*) nachträglich vermeiden

Exkurs Nachfolgend ein Beispiel für ein Programm, das bei Ausführung seinen Kode selbst modifiziert, um Befehlssatzeinschränkungen zu bewältigen. Dabei handelt es sich um

eine ↑Routine zum Einlesen eines Datums über eine Eingabeschnittstelle (↑*port-mapped* I/O), wobei die Kennung der Schnittstelle ein ↑tatsächlicher Parameter dieser Routine sein soll. Prozessor sei der ↑i8080, dessen ↑Maschinenbefehl für die Eingabe (`in`) die Kennung der Eingabeschnittstelle allerdings nur als Direktwert kodiert im Operandenfeld zulässt. Damit nun die Routine ein Datum von einer beliebigen Eingabeschnittstelle einlesen kann, wird die als Parameter (im Akkumulator) übergebene Schnittstellenkennung, die ein Zahlenwert ist, an die Stelle des Operanden des Eingabebefehls (`in`) geschrieben (`sta`), bevor dieser Befehl als nächster in der Folge zur Ausführung kommt. Die ↑Adresse des Operanden (`ugh+1`) ist um eins größer als die ↑symbolische Adresse (`ugh`) des Eingabebefehls in dem Programm: der Operand folgt dem ↑Operationskode im nächsten ↑Byte. Der im Beispiel angegebene Operand für die Schnittstellennummer (`0`) ist lediglich ein ↑Platzhalter, er wird bei Ausführung des Programms durch den Parameterwert ersetzt.

```
in_by_port_variable:
     sta ugh+1     ; define port number
ugh: in 0          ; read data via port
     ret
```

Diese Programmiertechnik setzt allerdings ein schreibbares ↑Textsegment voraus. Damit ist sie grundsätzlich nicht nutzbar für Programme, die im ↑Festwertspeicher liegen. Auch bei Prozessoren mit *Harvard-Architektur*, bei der nämlich ↑Text und ↑Daten eines Programms strikt voneinander getrennt in verschiedenen ↑Hauptspeichern liegen, lässt sich solch eine Kodemodifikation nicht durchführen: jede Schreiboperation geht implizit zum Datenspeicher, vermeintlich dort liegende und zu modifizierende Maschinenbefehle blieben grundsätzlich wirkungslos, da der Prozessor zum Befehlsabruf prinzipiell den Text- beziehungsweise Programmspeicher nutzt. Von solch einer speziellen ↑Rechnerarchitektur einmal abgesehen sind bei bestimmten ↑Betriebsarten die Textsegmente durch das ↑Betriebssystem und unter Mitwirkung einer ↑MMU auch aus Sicherheitsgründen schreibgeschützt.

Moderne Prozessoren geben auch eigentlich keinen Anlass mehr, den eigenen Programmkode aus Gründen von Befehlssatzeinschränkungen modifizieren zu wollen. Ausgenommen Schadsoftware, so gibt es Bedarf dafür nur noch zur (*on the fly*) Optimierung des Programmkodes zur ↑Laufzeit. Dies aber händisch zu tun, gilt als schlechter Programmierstil — daher auch der „Aufschrei“ (`ugh`) im präsentierten Programmbeispiel oben.

Selbstvirtualisierung (en.) ↑*self-virtualisation*. Form der ↑Virtualisierung, die die eigene (reale) Maschine betrifft. Dabei stellt die jeweils bereitgestellte ↑virtuelle Maschine ein vollständiges Abbild der realen Maschine dar, dass heißt, das ↑Programmiermodell der virtuellen Maschine ist mit dem der realen Maschine identisch. Für einen im ↑Maschinenprogramm residierenden ↑Prozess ist es damit auch nicht feststellbar, ob er auf einer realen oder virtuellen Maschine stattfindet. Typische Ausprägungen dieser Virtualisierungsart sind die ↑Vollvirtualisierung und ↑Paravirtualisierung.

self-modifying code (dt.) ↑selbstmodifizierender Kode.

self-virtualisation (dt.) ↑Selbstvirtualisierung.

semantic gap (dt.) ↑semantische Lücke.

semantische Lücke (en.) ↑*semantic gap*. Bedeutungsbezogener Unterschied zwischen den Beschreibungen von zwei Sprachebenen in einem mehrschichtig organisierten ↑Rechensystem, für das eine bestimmte ↑hierarchische Struktur definiert ist. Ursprünglich begründet in der Diskrepanz zwischen den komplexen Operationen der Konstrukte einer höherer Programmiersprache und den einfachen Operationen (↑*instruction set*) einer ↑CPU.

Semaphor (en.) ↑*semaphore*. Mittel zur ↑Koordination unterschiedlicher Art, um die Vergabe einer (↑binärer Semaphor) oder gegebenenfalls mehrerer (↑allgemeiner Semaphor) ↑Ressourcen desselben Typs zu regeln; eine ↑Synchronisationsvariable als ↑Exemplar eines speziellen ↑Datentyps auf dem im Wesentlichen die beiden Operationen ↑P und ↑V definiert sind.

Mit P synchronisiert sich ein ↑Prozess auf ein bestimmtes ↑Ereignis, das etwas über die Verfügbarkeit einer bestimmten Ressource aussagt und dessen Auftreten durch V entweder von

ihm selbst (↑unilaterale Synchronisation) oder von einem anderen (uni- und ↑multilaterale Synchronisation) Prozess angezeigt wird. Das heißt, P kann den aufrufenden Prozess blockieren und V kann einen (durch P) blockierten Prozess deblockieren. Da beide Operationen insbesondere auch von verschiedenen Prozessen benutzt werden, müssen sie einer ↑Nebenläufigkeitssteuerung unterliegen: P und V gehören zur Klasse der ↑Elementaroperation, deren Durchführung auch bei ↑Parallelverarbeitung ein konsistentes Ergebnis liefern muss — das bedeutet jedoch nicht, sie als ↑atomare Operation auslegen zu müssen.
Zentrale Bedeutung für den Fortschritt von einem Prozess hat die jeweils in P und V formulierte ↑Ablaufsteuerung. Diese kann in P eine ↑Prozessblockade bewirken, die erst durch ein V wieder aufgehoben wird; sie stellt sich für einen binären Semaphor etwas anders dar als für einen allgemeinen Semaphor. Beiden gemeinsam ist aber, dass die in P blockierten Prozesse auf einer ↑Warteliste bilden, die durch V abgebaut wird. Diese Reihe von wartenden Prozessen ist entweder als dynamische Datenstruktur (bspw. in Form des ↑Deskriptors einer ↑Schlange) ein Attribut des Semaphordatentyps, damit pro ↑Exemplar vorhanden und wird auch nach eigenen Kriterien verwaltet (gemeinhin ↑FCFS), oder sie wird im Moment der (durch V veranlassten) Deblockierung eines Prozesses durch den ↑Planer nach seinen Kriterien berechnet. Erstere Variante ist anfällig für ↑Interferenz mit dem Planer, da die Abarbeitung der Warteliste und das Bedienungsverfahren der ↑Bereitliste in aller Regel verschieden sind. Letztere Variante ist anfällig für Varianzen in der zur Deblockierung anfallenden ↑Latenzzeit, da die ↑Prozesstabelle erst nach Einträgen abgesucht werden muss, die mit bestimmten Exemplaren des Semaphordatentyps verknüpft sind.
Wichtiges Merkmal — weder Fehler noch Unzulänglichkeit, wie gelegentlich kolportiert — beider Operationen (P und V) darüberhinaus ist, dass sie von jedem Prozess gleichberechtigt genutzt werden können. Anderenfalls gestaltet sich die ↑Koordinierung von Prozessen in nicht wenigen Fällen schwierig (z.B. Hoare-/Hansen-Typ von ↑Monitor) oder gar unmöglich (z.B. ↑Erzeuger-Verbraucher-Problem, Planer als ↑kritischer Abschnitt). Gleichwohl haben sich spezielle Semaphorvarianten herausgebildet, wie etwa ↑privater Semaphor und ↑Mutex, durch die bestimmte Muster der Synchronisation unterstützt werden und die helfen, Programmierfehler zu vermeiden oder zu erkennen.

Exkurs An sich sind die Operationen eines Semaphors in funktionaler Hinsicht einfach, sie müssen lediglich Buch führen über den Zustand einer ↑Prozesssperre und einer Liste gegebenenfalls gesperrter Prozesse. Ein dafür geeigneter und für verschiedene Semaphorarten geeigneter Datentyp kann wie folgt definiert sein:

```
typedef struct {
    note_t note;                                /* status indication */
    guard_t wait;                        /* expected event, waiting list */
} semaphore_t;
```

Der Unterschied zwischen binärem und allgemeinem Semaphor liegt vor allem in den Operationen zur Statusanzeige (`note`), die einerseits mit Booleschen und anderseits mit ganzzahlige Werten umgehen müssen. Entsprechend ist der betreffende ↑Datentyp (`note_t`) entweder als `bool` oder als `int` definiert. Für den binären Semaphor sind die Funktionen zur Statusänderung wie folgt ausgelegt:

```
inline int decrease(semaphore_t *sema) { return sema->note = false; }
inline int increase(semaphore_t *sema) { return sema->note = true; }
```

Demgegenüber ist die Zahlensemantik des allgemeinen Semaphors nicht binär, sondern ganzzahlig ($\mathbb{Z}$) oder nichtnegativ ganzzahlig ($\mathbb{N}_0$). Diese schließt die vergleichsweise doch sehr einfachen, lediglich Lese-/Schreibbefehle erfordernden Binäroperationen zwar mit ein, bedingt allerdings komplexere arithmetische (↑*read-modify-write*) Operationen zur Statusänderung:

```
inline int decrease(semaphore_t *sema) { return --sema->note; }
inline int increase(semaphore_t *sema) { return sema->note++; }
```

Schließlich wird noch eine Operation zur Statusanzeige benötigt. Diese ist unabhängig von

der Auslegung als binärer oder allgemeiner Semaphor:

```
inline int capacity(semaphore_t *sema) { return sema->note; }
```

Auf Basis dieser drei Operationen lässt sich für beide Semaphorarten, gegebenenfalls auch noch darüber hinausgehend, ein gemeinsames Implementierungsmuster entwickeln. Damit kann durch *bedingte* ↑*Kompilation* in einfacher Weise die jeweils gewünschte Semaphorart generiert werden. Dieses Muster wird nachfolgend erklärt.
Zunächst die Semaphorprimitive, mit der sich ein Prozess auf die Verfügbarkeit von Ressourcen synchronisieren kann und die dazu eine *bedingte Prozesssperre* formuliert. Die Prozesssperre ist aktiv, wenn der Semaphor (`note`) den Wert null hat. Nachfolgende Prozesse müssen dann blockieren und kommen dazu auf eine ↑Warteliste (`wait`) für ein Geschehnis, das die Aufhebung der Prozessblockade bewirkt. Die diesbezügliche P-Operation stellt sich wie folgt dar:

```
void P(semaphore_t *sema) {
    when (capacity(sema) > 0)           /* are resources still available? */
        decrease(sema);                                   /* yes, take one */
    else                                     /* no, delay current process */
        sleep(&sema->wait);                            /* wait for release */
}
```

Lässt die Aufnahmefähigkeit (`capacity`) des Semaphors eine weitere Vergabe der von ihm verwalteten Ressourcen zu (d.h., `note` ist positiv beziehungsweise *true*), erniedrigt der Semaphor seinen Wert und lässt den Prozess fortfahren. Anderenfalls wird der anfordernde Prozess schlafen gelegt (`sleep`). Die inverse Operation dazu gestaltet sich wie folgt:

```
void V(semaphore_t *sema) {
    if (backlog(&sema->wait) == 0)                 /* empty waiting list? */
        increase(sema);                                 /* yes, take back */
    else                                          /* no, continue process */
        wakeup(&sema->wait);                           /* unblock next one */
}
```

Weist die Warteliste keinen Rückstand (`backlog`, vgl. S. 277) von schlafenden Prozessen auf, erhöht der Semaphor seinen Wert. Ansonsten wird genau einer aus dieser Menge aufgeweckt (`wakeup`). In beiden Fällen fährt der Prozess fort.
So funktional einfach die hier skizzierten Operationen (P und V) auf den ersten Blick auch sein mögen, sie sind problematisch im Zusammenspiel (a) mit dem Planer und (b) vor dem Hintergrund ↑gleichzeitiger Prozesse. Dem Aufwecken genau eines schlafenden Prozesses geht eine Auswahlentscheidung zu den auf der Warteliste (`wait`) stehenden Prozesse voraus. Diese Entscheidung kann im Konflikt zum Planer stehen, das heißt, die mögliche Ursache für eine ↑Prioritätsverletzung bilden. Alle schlafenden Prozesses aufzuwecken, würde dieses Problem beseitigen, da ausschließlich der Planer die Prozesse dann in die korrekte Reihenfolge bringen wird. In dem Fall müsste allerdings jeder aufgeweckte Prozess in P seine Wartebedingung nochmals überprüfen (***when*** ≡ `while`), da nur ein einziger aus dieser Runde den Semaphor passieren darf. Dies bedeutet aber auch, dass neu in P ankommende Prozesse aufgeweckte Prozesse überholen könnten (↑*starvation*). Eine andere Lösung bestände darin, die Warteliste (`wait`) nicht als Datenstruktur zu begreifen, sondern sie etwa durch eine Zählschleife (`for`) über die Liste aller Prozesse bei Bedarf immer neu zu berechnen. Dann erfolgt beim Schlafenlegen (`sleep`) lediglich der Vermerk, dass der betreffende Prozess auf das durch einen Adresswert (`&sema->wait`) eindeutig bezeichnete Geschehnis wartet. Allerdings müsste dann zur Feststellung des Bestands (`backlog`) an Prozessen, die dieses Geschehnis erwarten, der erste davon basierend auf der Menge aller Prozesse erst (in einer Schleife) berechnet werden. Gleiches würde zum Aufwecken (`wakeup`) erforderlich sein. Derartige Berechnungen führt der Planer selbst durch, womit eine Prioritätsverletzung bei der Auswahl eben des einen aufzuweckenden Prozesses ausgeschlossen wäre. Folglich bräuchte der jeweils aufgeweckte Prozess in P seine Wartebedingung nicht erneut zu überprüfen (***when*** ≡ `if`),

Das andere Problemfeld ergibt sich durch die ↑Wettlaufsituationen, die sich an verschiedenen Stellen der beiden Operationen zeigen können. Hier sind zunächst die ↑Aktionen (`decrease`, `increase`) zur Werteänderung des Semaphors zu nennen. So müssten für einen binären Semaphor die betreffenden Lese-/Schreibbefehle und für einen allgemeinen Semaphor die jeweiligen Zählbefehle unteilbar sein. Des Weiteren ist in P die ↑Aktionsfolge zur Überprüfung der Wartebedingung und das Schlafenlegen des betreffenden Prozesses kritisch. Die skizzierte Lösung birgt das Risiko, dass ein sich in P (mittels `sleep`) schlafen legender Prozess das für ihn bestimmte, durch ein gleichzeitiges V (genauer: `wakeup`) hervorgerufene Aufwachereignis verpasst (↑*lost wake-up*) und damit gegebenenfalls überhaupt nicht mehr aufgeweckt wird, weil ein weiteres V auf den betreffenden Semaphor ausbleibt. Das gleiche Problem zeigt sich in V in Bezug auf die Aktionsfolge zur Überprüfung der Weckbedingung (`backlog`) und der Statusänderung (`increase`) des Semaphors: kommt in dem Moment ein P zur Ausführung und lässt die Kapazität des Semaphors keine weitere Ressourcenvergabe zu, wird der betreffende Prozess schlafen gelegt, obwohl V (durch `increase`) gleich eine verfügbare Ressource anzeigen wird. Darüberhinaus könnten mehr Prozesse als erlaubt den Semaphor passieren, obwohl dies nur eine Teilmenge (allgemeiner Semaphor) oder sogar nur einem einzigen (binärer Semaphor) davon möglich sein dürfte. Dies könnte geschehen, wenn mehrere Prozesse zugleich in P feststellen, dass ihre Wartebedingung nicht erfüllt ist und sie diese dann erst (im `then`-Zweig, durch `decrease`) für andere Prozesse erfüllen: Den Semaphorwert (`note`) zu testen und ihn dann einzustellen sind wenigstens zwei Schritte, die atomar verlaufen müssten (↑TAS oder ↑FAA).

Jeder der beiden gezeigten Prozedurrümpfe für sich allein als auch beide zusammen genommen müssten wiedereintrittsfähig (↑*re-entrant*) formuliert sein, und zwar in Bezug auf beliebige Prozesskonstellationen. Für gewöhnlich werden beide Operationen als Bestandteil desselben kritischen Abschnitts verstanden, der dann durch ↑blockierende Synchronisation vor gleichzeitigen Prozessen geschützt wird. Für den allgemeinen Semaphor ist hierzu durchaus der binäre Semaphor verwendbar, wobei aber wenigstens für letzteren immer eine Technik verschieden vom Semaphor zum Einsatz kommen muss (↑Unterbrechungssperre, ↑Verdrängungssperre, ↑Umlaufsperre). Alternativ bieten sich auch Lösungen an, die auf die ↑nichtblockierende Synchronisation zurückgreifen, dazu aber Hilfestellung durch Funktionen zur speziellen Kontrolle des ↑Prozesszustands benötigen.

semaphore (dt.) ↑Semaphor; Winker, Signalmast, Formsignal; (gr.-nlat.) Zeichenträger.

sensitiver Befehl Bezeichnung für einen ↑Maschinenbefehl, dessen direkte Ausführung durch eine ↑virtuelle Maschine nicht tolerierbar ist. Ein solcher Befehl gilt als *störungsempfindlich*, wenn er den Zustand der ↑Systemsteuerung, reservierter ↑Prozessorregister oder einer reservierten ↑Speicherstelle ändert/abfragt, das ↑Schutzsystem für Programme, die privilegiert ablaufen müssen, referenziert oder Ein-/Ausgabe tätigt.

Separator Vorrichtung, um einzelne Bestandteile in der mehrwortig ausgeführten Bezeichnung einer ↑Entität zu trennen. Ein spezieller Trenntext, der einzelne oder eine Gruppe von Worten separiert. Dabei wird der jeweils separierte Abschnitt als ↑Name aufgefasst, der sich auf einen bestimmten (vor dem Trenntext bezeichneten) ↑Namenskontext bezieht. Beispiele solcher Trenntexte sind: `>` (*greater-than*, ↑Multics), `/` (*slash*, ↑UNIX) oder `\` (*backslash*, ↑Windows) — in der Reihenfolge ihrer historischen Entwicklung.

sequence point (dt.) ↑Sequenzpunkt.

sequencing (dt.) Reihenfolgebildung, ↑Sequentialisierung.

sequential consistency (dt.) ↑sequentielle Konsistenz.

Sequential Pascal Programmiersprache: imperativ, prozedural (Hansen, 1975). Spezialisierung von ↑Pascal, und zwar Einschränkung auf Sprachkonstrukte zur Formulierung *typsicherer* (sequentieller) ↑Programme. Sprachbasis von ↑Concurrent Pascal).

sequential process (dt.) ↑sequentieller Prozess.

sequential program (dt.) ↑sequentielles Programm.

Sequentialisierung (en.) ↑*sequencing*. Schaffung einer bestimmten Reihung in einer ↑Aktionsfolge entlang einer ↑Kausalordnung.
Handelt es sich bei den in Reihe zu bringenden Aktionsfolgen jeweils um ein ↑Prozessexemplar, das immer komplett und als eine in sich abgeschlossene Einheit zu betrachten ist, dann bestimmt für gewöhnlich ein ↑Ablaufplan deren Aufeinanderfolge. Solch ein Fall kann sodann zur Konsequenz haben, dass die betreffenden ↑Prozesse selbst nicht mehr für ↑Koordinierung sorgen müssen: sie sind durch den Ablaufplan *impliziert* koordiniert. Anderenfalls, wenn ↑Planung allein nicht für koordiniert stattfindende Aktionsfolgen sorgen kann, müssen die Prozesse *explizit*, und zwar durch ↑Synchronisierung ihrer ↑Aktionen, für die korrekte Reihenfolgebildung sorgen.

sequentielle Konsistenz (en.) ↑*sequential consistency*. Modell der ↑Speicherkonsistenz, nach dem jeder ↑Prozessor jede Operation auf den ↑Arbeitsspeicher immer in der durch das (nichtsequentielle) ↑Programm spezifizierten Reihenfolge durchführt. Schwächer als ↑strikte Konsistenz.

sequentieller Prozess (en.) ↑*sequential process*. Ein ↑Prozess, der für gewöhnlich durch ein ↑sequentielles Programm definiert ist und nur einen einzigen ↑Handlungsstrang aufweist.
Allerdings kann ein solcher Prozess auch durch ein ↑nichtsequentielles Programm bestimmt sein, dann allerdings nur phasenweise und partiell, und zwar wenn in diesem ↑Programm ein ↑kritischer Abschnitt formuliert ist. Ein solcher Abschnitt erlaubt für sich zu einem Zeitpunkt nur einen Handlungsstrang, er darf nur sequentiell von einem Prozess durchlaufen werden — allerdings könnte diesem Prozess ein von ihm gänzlich unabhängiger, ↑nebenläufiger Prozess definiert durch das (beiden) gemeinsame Programm zur Seite stehen. Mit dem kompletten nichtsequentiellen Programm als ↑Bezugssystem ist grundsätzlich also immer noch ein ↑nichtsequentieller Prozess beschrieben.

sequentielles Programm (en.) ↑*sequential program*. Ein ↑Programm, das ausschließlich Konstrukte zur Formulierung sequentieller Abläufe verwendet. Diese Konstrukte sind in der Programmiersprache enthalten, in der das Programm formuliert ist. Jedoch ist zu beachten, dass ein und derselbe ↑Programmablauf auf einer ↑Abstraktionsebene (höhere) sequentiell und einer anderen (tiefere) parallel sein kann: nämlich wenn auf höherer Ebene ein ↑Unterprogramm einer tieferen Ebene aufgerufen wird und das Unterprogramm ein ↑nichtsequentielles Programm darstellt.

Sequenzpunkt (en.) ↑*sequence point*. Eine beliebige Stelle in einem ↑Prozess, an der zugesichert ist, dass alle ↑Seiteneffekte vorangegangener ↑Aktionen verrichtet wurden und noch kein Seiteneffekt nachfolgender Aktionen auftreten konnte. In ↑C sind solche Punkte an folgenden Stellen definiert: zwischen der Auswertung der linken und rechten Operanden des logischen Und (`&&`) beziehungsweise Oder (`||`), nach der Auswertung des ersten Teils (Bedingung) der aus drei Teilen bestehenden bedingten Zuweisung (`?:`), am Ende eines vollständigen Ausdrucks, vor Eintritt in ein ↑Unterprogramm durch einen Unterprogrammaufruf, nach Rückkehr davon und Übernahme zurückgegebener Parameter, am Ende einer Initialisierung, zwischen den Vereinbarungszeichen einer Sequenz von Vereinbarungen und nach jeder mit einem Formatbezeichner zur ↑Ein-/Ausgabe (`printf`) verbundenen Konversion.

server (dt.) ↑Auftragnehmer, ↑Zusteller.

service time (dt.) ↑Bedienzeit.

service unit (dt.) ↑Bedieneinheit.

session (dt.) ↑Sitzung.

shared code (dt.) gemeinsam genutztes ↑Programm oder ↑Unterprogramm. ↑Text und möglicherweise auch ↑Daten liegen in einem von mehr als einen ↑Prozess zugleich zugreifbaren Bereich im ↑Arbeitsspeicher (↑*shared memory*). Gilt für diesen Programmbereich keine ↑Ablaufinvarianz oder besteht unter den Prozessen zwar eine Kausalordnung, deren Erhaltung durch die ↑Prozesseinplanung jedoch nicht zugesichert werden kann, ist zur Vorbeugung eventueller Inkonsistenzen ↑Synchronisation unter den Prozessen zu erzielen.

shared data (dt.) gemeinsam genutzte ↑Daten. Die Daten liegen in wenigstens einem ↑Speicherwort, auf das von mehr als einem ↑Prozess zugleich zugegriffen werden kann (↑*shared memory*). Um im Falle der gleichzeitig möglichen Schreibzugriffe (Lesen-Schreiben oder Schreiben-Schreiben) einem inkonsistenten Datenbestand vorzubeugen, ist ↑Synchronisation unter den Prozessen zu erzielen.

shared library (dt.) ↑Gemeinschaftsbibliothek.

shared memory (dt.) ↑gemeinsamer Speicher.

shared resource (dt.) ↑gemeinsames Betriebsmittel.

shared variable (dt.) ↑gemeinsame Variable.

shared-memory processor (dt.) ↑speichergekoppelter Multiprozessor.

shell (dt.) Außenhaut, Randzone, Ummantelung. Mit ↑Multics (um 1964) eingeführte Bezeichnung für den über eine ↑Dialogstation genutzten ↑Kommandointerpreter zur Interaktion mit dem ↑Betriebssystem. Vorreiter dafür war `RUNCOM` (um 1963) von ↑CTSS, ein ↑Programm zur Verarbeitung von in ↑Skriptsprache formulierter Kommandos nebst Parametersubstitution.

short-term scheduling (dt.) ↑kurzfristige Planung.

side effect (dt.) ↑Seiteneffekt.

Signal (en.) ↑*signal*. Bezeichnung für ein Zeichen mit einer bestimmten Bedeutung, der Träger einer Information (in Anlehnung an den Duden). Gemeinhin erklärt auch als ↑Nachricht mit „Nulllänge“, die jedoch nicht inhaltslos ist, der Meldung eines ↑Ereignisses entspricht.

signal (dt.) Meldung, ↑Signal.

signal and continue (dt.) signalisieren und fortfahren. ↑Signalisierungsdisziplin einer *nichtblockierenden* ↑Bedingungsvariablen, bei der der signalisierende ↑Prozess die Kontrolle über den ↑Monitor behält (Besitzwahrung). Allerdings erst wenn dieser Prozess den Monitor verlässt, wird ein signalisierter Prozess seine ↑Aufgabe im Monitor wieder aufnehmen und fortsetzen können. Eingeführt mit dem Monitorkonzept von ↑Mesa.

Bezeichnend für diese Regel ist die Besitzwahrung des Monitors für den signalisierenden Prozess, der seine Tätigkeit nach Abgabe des Signals ununterbrochen im Monitor fortsetzt. Als Folge daraus kann der Prozess während seines Monitoraufenthalts mehrere Signale abgeben, dadurch auch mehrere blockierte Prozesse, die verschiedene ↑Ereignisse erwarten, signalisieren, deblockieren und so ihre erneute Bereitstellung durch den ↑Planer veranlassen. In Bezug auf ein einzelnes Signal, deblockiert ja nach Verfahren genau nur ein Prozess (↑*unicast*) oder alle auf der ↑Warteliste stehenden Prozesse (↑*multicast*).

Da es weder den augenblicklichen noch einen atomaren Wechsel zwischen signalisierenden und signalisiertem Prozess gibt, kann letzterer bei der Wiederaufnahme seiner Tätigkeit nicht davon ausgehen, dass seine Wartebedingung immer noch als aufgehoben gilt. Sollten alle Prozesse auf der Warteliste durch das eine Signal aufgeweckt worden sein, was der signalisierte Prozess nicht wissen kann, oder kam es seit Verlassen des Monitors durch den signalisierenden Prozess zum Neueintritt in den Monitor durch wenigstens einen weiteren Prozess, können die geschehenen Zustandsänderungen bereits wieder zur Erfüllung dieser

wie auch anderer Wartebedingungen geführt haben. Daher muss ein signalisierter, aufgeweckter Prozess seine Wartebedingung erneut überprüfen. Daraus ergibt sich nachfolgend skizziertes typisches Muster beim Umgang mit der Bedingungsvariablen:

```
while (wait condition)                  | do cancel wait condition;
    wait(event);                        | notify(event, way); /* uni-/multicast */
```

Der signalisierende Prozess (rechts) gibt die Art und Weise (`way`) der Meldung des spezifizierten Ereignisses (`event`) vor, das heißt, ob genau einer oder alle auf der mit dem Ereignis verknüpften Warteliste stehenden Prozesse aufgeweckt werden. Gebräuchlich sind auch zwei verschiedene Primitiven, je eine pro Art der Meldung (alle: `broadcast` in Mesa, `notifyAll` in Java). Soll nur ein Prozess aufgeweckt werden, ist im Zuge der Meldung (`notify`) eine Auswahlentscheidung zu treffen: für gewöhnlich ↑FIFO, die Warteliste bildet *keine* ↑Planungswarteschlange. Unabhängig von der Anzahl der aufgrund des Ereignisse aufgeweckten Prozesse, muss jeder signalisierte Prozess (links) in einer Schleife (`while`) die Gültigkeit der Aufhebung der Wartebedingung überprüfen, da sie alle untereinander und mit anderen bereitgestellten Prozessen um die Fortsetzungsmöglichkeit ihrer Tätigkeit konkurrieren (↑*signal scattering and continue*). Diese Maßnahme lässt den Prozess zusätzlich ↑falsche Signalisierungen tolerieren.

signal and return (dt.) signalisieren und zurückkehren. ↑Signalisierungsdisziplin einer *nichtblockierenden* ↑Bedingungsvariablen, bei der der signalisierende ↑Prozess die Kontrolle über den ↑Monitor aufgibt und diesen sofort verlässt. Der signalisierte Prozess wird daraufhin sofort seine ↑Aufgabe im Monitor wieder aufnehmen und fortsetzen können. Eingeführt mit dem Monitorkonzept von ↑Concurrent Pascal.
Bezeichnend für diese Regel ist die logische Kontrollübergabe des Monitors vom signalisierenden zum signalisierten Prozess. Letzterer hat die Zusicherung, dass nur er signalisiert wurde (↑*unicast*). Der signalisierende Prozess hat eine bestimmte Wartebedingung aufgehoben und verlässt den Monitor mit der Aufforderung, einen Prozess, dessen Fortgang sich im Monitor eben wegen dieser Bedingung verzögert (*delay*), an seiner statt fortzusetzen (*continue*). Dabei ist es jedoch möglich, dass ein anderer Prozess zwischenzeitlich den Monitor betreten, durchlaufen und wieder verlassen hat. Dem Modell nach kann somit die Wartebedingung des signalisierten Prozesses abermals zutreffen, sobald er seine Tätigkeit wieder aufnimmt, er muss diese Bedingung erneut überprüfen. Daraus ergibt sich als typisches Interaktionsmuster für die betreffenden Prozesse folgendes Bild:

```
while (wait condition)                  | do cancel wait condition;
    delay(event);                       | continue(event);   /* final statement */
```

Dem ursprünglichen Ansatz (Hansen, 1975) nach kann es pro Warteliste für ein bestimmtes Ereignis (`event`) nur maximal einen verzögerten Prozess geben (*single-process queue*). Dies folgt einerseits aus dem zugrunde liegenden Verfahren für die ↑kurzfristige Planung der Prozesse (↑*fixed-priority scheduling*) und andererseits aus der dazu korrespondieren Funktionsweise der als ↑Planungswarteschlange repräsentierten Warteliste. Die Warteliste für das gegebene Ereignis ist von oben nach unten mit absteigender ↑Priorität der Prozesseinträge sortiert. Diese Sortierung entspricht auch der der Prozesseinträge in der ↑Prozesstabelle. Bei der Signalisierung (`continue`) wird die Warteliste dieser Sortierung folgend nach dem ersten auf das Ereignis wartenden Prozess abgesucht. Würde die Warteliste eines Ereignisses allerdings eine dynamische Datenstruktur (↑*queue*) bilden und nicht als Tabelle (↑*array*) repräsentiert sein, könnten sehr wohl mehrere Prozesse darauf verzeichnet sein. Aber auch in diesem Fall müssen die Prozesseinträge entsprechend ihrer Priorität aufgereiht sein. So käme es in beiden Fällen zur Auswahl desselben wartenden Prozesses.

signal and urgent wait (dt.) signalisieren und dringendes warten. ↑Signalisierungsdisziplin einer *blockierenden* ↑Bedingungsvariablen, bei der der signalisierende ↑Prozess die Kontrolle über den ↑Monitor an den signalisierten Prozess abgibt (Besitzwechsel), blockiert und bevorzugt zurückerhält, sobald letzterer im Monitor erneut warten muss oder diesen verlässt.

Der signalisierte Prozess, dessen Wartebedingung eben durch den signalisierenden Prozess aufgehoben wurde, nimmt sofort seine ↑Aufgabe im Monitor wieder auf. Eine für den von Hoare (um 1974) eingeführten Monitor typische Vorgehensweise.
Bezeichnend für diese Regel ist die logische ↑Unteilbarkeit der ↑Aktionsfolge für den Wechsel vom signalisierenden zum signalisierten Prozess. Letzterer hat die Zusicherung, dass nur er signalisiert wurde (↑*unicast*) und auch kein anderer Prozess zwischenzeitlich den Monitor betreten kann. Dem Modell nach gilt die Wartebedingung des signalisierten Prozesses immer noch als aufgehoben, sobald er seine Tätigkeit wieder aufnimmt, er muss diese Bedingung nicht erneut überprüfen. Daraus ergibt sich als typisches Interaktionsmuster für die betreffenden Prozesse folgendes Bild:

```
if (wait condition)                | do cancel wait condition;
    wait(event);                   | signal(event);
```

Der signalisierende Prozess (rechts) kommt im Zuge der Wechselaktion auf eine *Vorzugswarteschlange*, die ausschließlich Prozesse enthält, die bei Meldung (`signal`) des Ereignisses der Aufhebung einer bestimmten Wartebedingung den Monitor abgeben mussten. Prozesse in dieser Warteschlange erhalten nacheinander den Vorzug zum Wiedereintritt in den Monitor, sobald dieser freigegeben wurde: all diese Prozesse werden jenen vorgezogen, die den Monitor „von außen“ betreten wollen. Der signalisierte Prozess (links) überprüft die Gültigkeit der Aufhebung der Wartebedingung nicht erneut, da zwischenzeitlich kein anderer Prozess den Monitor hat betreten können. Er führt gewöhnlich nur eine bedingte Anweisung (`if`) durch, was dann allerdings zur Folge haben kann, ↑falsche Signalisierungen nicht zu tolerieren.

signal and wait (dt.) signalisieren und warten. ↑Signalisierungsdisziplin einer *blockierenden* ↑Bedingungsvariablen, bei der der signalisierende ↑Prozess die Kontrolle über den ↑Monitor an einen signalisierten Prozess abgibt (Besitzwechsel), blockiert und bevorzugt zurückerhält, sobald letzterer im Monitor erneut warten muss oder diesen verlässt. Einer der signalisierten Prozesse, deren jeweilige Wartebedingung durch den signalisierenden Prozess aufgehoben wurde, nimmt bevorzugt seine ↑Aufgabe im Monitor wieder auf. Eine für den von Hansen (um 1973) eingeführten Monitor typische Vorgehensweise.
Bezeichnend für diese Regel ist, dass alle Prozesse, die die Aufhebung der betreffenden Wartebedingung erwarten, signalisiert (↑*multicast*) und damit zur Fortführung ihrer Tätigkeiten bereitgestellt werden. Der Prozess, der die Wartebedingung aufgehoben hat, wechselt zu einen der signalisierten Prozesse. Jeder dieser Prozesse kann nach Wiederaufnahme seiner Tätigkeit im Monitor eine Zustandsänderung vornehmen, die die Wartebedingung für nachfolgende Prozesse erneut erfüllt. Folglich muss ein signalisierter Prozess, wenn er seine Tätigkeit im Monitor wieder aufnimmt, seine Wartebedingung erneut überprüfen, da er nicht wissen kann, der wievielte aufgeweckte Prozess bezogen auf dasselbe Ereignis er ist. Daraus ergibt sich als typisches Interaktionsmuster für die betreffenden Prozesse folgendes Bild:

```
while (wait condition)             | do cancel wait condition;
    wait(event);                   | signal(event);
```

Der signalisierende Prozess (rechts) kommt im Zuge der Wechselaktion auf eine *Vorzugswarteschlange*, die ausschließlich Prozesse enthält, die bei Meldung (`signal`) des Ereignisses der Aufhebung einer bestimmten Wartebedingung den Monitor abgeben mussten. Prozesse in dieser Warteschlange erhalten nacheinander den Vorzug zum Wiedereintritt in den Monitor, sobald dieser freigegeben wurde: all diese Prozesse werden jenen vorgezogen, die den Monitor „von außen“ betreten wollen. Demgegenüber muss jeder signalisierte Prozess (links) in einer Schleife (`while`) die Gültigkeit der Aufhebung der Wartebedingung überprüfen. Diese Maßnahme lässt den Prozess zusätzlich ↑falsche Signalisierungen tolerieren.

signal scattering and continue (dt.) Signalstreuen und fortfahren. ↑Signalisierungsdisziplin im Zusammenhang mit einer ↑Synchronisationsvariablen für eine ↑bedingte kritische Region. Der signalisierende ↑Prozess behält die Kontrolle über die (bedingt) ↑kritische Region, in der er sich gerade befindet (Besitzwahrung). Die dieser Synchronisationsvariablen zugehörigen ↑Er-

eigniswarteschlange wird komplett geleert, das heißt, alle darin gelisteten Prozesse werden signalisiert und bereitgestellt, um ihre jeweilige ↑Aufgabe in der kritischen Region wieder aufnehmen und fortsetzen, mit dem signalisierenden Prozess um den ↑Prozessor konkurrieren zu können. Eine für die von Hansen (1972) vorgestellte bedingt kritische Region typische Vorgehensweise, die dann in leicht abgewandelter Form (↑*signal and wait*) Einzug in den von ihm (um 1973) eingeführten ↑Monitor gefunden hat.
Bezeichnend für diese Regel ist zum einen die Besitzwahrung der kritischen Region für den signalisierenden Prozess und zum anderen die Meldung des Ereignisses an alle in der Ereigniswarteschlange stehenden Prozesse. Jede einzelne dieser Eigenschaften hat zur Folge, dass ein signalisierter Prozess, sobald er seine Tätigkeit in der kritischen Region wieder aufnimmt, die vor seiner Blockierung erfüllte Wartebedingung erneut überprüfen muss.
Verlässt der signalisierende Prozess die kritische Region, kann ein beliebig anderer Prozess in diese eintreten: (a) ein Prozess, der über den Hauptpfad „vorne“ auf die Region trifft, und (b) einer der signalisierten und damit fortgesetzten Prozesse über einen Seitenpfad. Fall (a) schließt Prozesse ein, die eben erst diese Region „hinten“ regulär verlassen haben. Jeder einzelne dieser Prozesse ist in der Lage durch Zustandsänderungen innerhalb der Region die Wartebedingung des Prozesses, der als nächster aus der Ereigniswarteschlange heraus die Region wieder betritt, erneut zu erfüllen. Demzufolge muss ein signalisierter Prozess erneut seine Wartebedingung überprüfen, nachdem er die Ereigniswarteschlange verlassen hat und bevor er seine Tätigkeit in der Region effektiv wieder aufnimmt. Die Interaktion zwischen dem signalisierenden Prozess einerseits und einem signalisierten Prozess andererseits stellt sich daher wie folgt dar:

```
while (wait condition)          | do cancel wait condition;
    await(event);               | cause(event);
```

Der signalisierende Prozess (rechts) veranlasst (`cause`) die Bereitstellung aller Prozesse, die das spezifizierte Ereignis (`event`) erwarten. Für die Bereitstellung sorgt die ↑Prozesseinplanung, die die signalisierten Prozesse der ↑Bereitliste hinzufügt und dabei gegebenenfalls eine ↑Umplanung vornimmt. So müssen alle dasselbe Ereignis (`event`) erwartenden Prozesse sowohl untereinander als auch mit den bereits auf der Bereitliste stehenden Prozessen um die Möglichkeit konkurrieren, ihre Tätigkeit fortsetzen zu können: ihre jeweilige Position in der Ereigniswarteschlange hat keine Bedeutung.
Auch wenn die Prozesse, die ihre kritischen Regionen aus Ereigniswarteschlangen und damit über einen Seitenpfad erneut betreten, Vorrang vor Prozessen erhalten sollten, die direkt über den Hauptpfad in kritische Regionen eintreffen, können immer noch signalisierte Prozesse untereinander in Konkurrenz um den Wiedereintritt stehen. Da dem signalisierten Prozess (links) nicht bekannt ist, wieviele seiner Konkurrenten sich seit der Signalisierung (`cause`) vor ihm in der kritischen Region befanden, prüft er die Gültigkeit der Aufhebung seiner Wartebedingung erneut ab (`while`), sobald sein abwarten (`await`) des Ereignisses (`event`) ein Ende hat. Diese Maßnahme lässt den Prozess zusätzlich ↑falsche Signalisierungen tolerieren.

Signalisierungsdisziplin (dt.) ↑*signalling discipline*. Einhalten von bestimmten Vorschriften, vorgeschriebenen Verhaltensregeln zur Übermittlung eines ↑Signals (in Anlehnung an den Duden), das ein von gegebenenfalls mehreren ↑Prozessen erwartetes ↑Ereignis anzeigt. Das Ereignis ist die Enträftung beziehungsweise Aufhebung einer bestimmten Wartebedingung durch einen anderen Prozess, das Signal (↑*consumable resource*) ist das Kommunikationsmittel, um dieses Geschehnis den betreffenden Prozessen mitzuteilen. Wird das Ereignis von keinem Prozess erwartet, ist das zugehörige Signal wirkungslos, es wird nicht vermerkt.
Je nach Modell kann der Vorgang für den signalisierenden Prozess (Sender) blockierend (↑*signal and wait*, ↑*signal and urgent wait*) oder nichtbockierend (↑*signal scattering and continue*, ↑*signal and continue*, ↑*signal and return*) ablaufen. Das heißt, entweder muss der betreffende Prozess den ↑Prozessor zugunsten des signalisierten Prozesses (Empfänger) abgeben, damit also die Ausführung seiner ↑Aufgabe unterbrechen, oder er setzt nach geschehener Ereignisanzeige unverzüglich mit seiner Tätigkeit fort.

Die Regeln sind nicht nur von zentraler Bedeutung für ↑bedingte kritische Regionen (*signal and compete*), sondern insbesondere auch für die solche Bereiche in abgeänderter Form zur Grundlage habenden ↑Monitore (*signal and continue/return/wait/urgent wait*). Gerade die verschiedenen Monitorkonzepte manifestieren sich in diesen Regeln, die damit die wesentlichen Unterscheidungsmerkmale begründen. Aber für beide Konzepte gilt, dass ein *innerhalb* eines solchen Bereichs blockierender Prozess, immer *außerhalb* dieses Bereichs die erneute Zuteilung des Prozessors (passiv oder aktiv) entgegensehen muss.
Allen Varianten gemeinsam ist das Prinzip der ↑Bedingungssynchronisation, bei der für den einen Prozess eine Wartebedingung erfüllt ist, die ein anderer Prozess aufheben muss. Trifft für einen Prozess eine solche Bedingung zu, muss er den Empfang eines Signals abwarten (`wait`) und wird erst fortfahren, nachdem ein anderer Prozess durch Senden des Signals die erfolgte Aufhebung der Wartebedingung meldet (`signal`). Daraus leitet sich ein typisches Muster der Interaktion ↑gekoppelter Prozesse ab:

```
when (wait condition)          | do cancel wait condition;
    wait(event);               | signal(event);
```

Der das Ereignis erwartende Prozess (links) blockiert (`wait`), wenn die Wartebedingung dies vorschreibt. Er tut dies unter Verwendung einer ↑Bedingungsvariablen (`event`), die das von ihm erwartete Ereignis repräsentiert. Diese Bedingungsvariable verwendet der das Ereignis anzeigende Prozess (rechts) zur Signalübermittlung (`signal`), nachdem er die Wartebedingung aufgehoben hat.
Nach Wiederaufnahme des Prozesses, dem die Aufhebung seiner Wartebedingung gemeldet wurde, ist sein Verhalten abhängig von der jeweils geltenden Regel zur Übermittlung des zugehörigen Signals und dem Zusammenspiel mit dem Prozess, der das Ereignis der Aufhebung Wartebedingung gemeldet hat. Das im gezeigten Interaktionsmuster auftretende *Konditional* (`when`) führt je nach Verhalten entweder zu einer bedingten Anweisung (`if`) oder wiederholten Überprüfung der Wartebedingung (`while`). Die Auslegung als bedingte Anweisung erfordert eine blockierende Bedingungsvariable, bei der höchstens ein Prozess signalisiert und dieser auch augenblicklich fortgesetzt wird (*signal and urgent wait*). Alle anderen Fälle benötigen die wiederholte Wartebedingungsüberprüfung durch den signalisierten Prozess, der sich nämlich nicht sicher sein kann, dass ihm zwischenzeitlich kein anderer Prozess zuvor gekommen ist und die Wartebedingung erneut erfüllt hat. Darüber hinaus werden dadurch ↑falsche Signalisierungen toleriert.

signalling discipline (dt.) ↑Signalisierungsdisziplin.

Signatur (en.) ↑*signature*. Bezeichnung für die formale Schnittstelle eines ↑Unterprogramms. Für gewöhnlich umfasst diese den ↑Namen und eine Liste beziehungsweise Reihenfolge der ↑Datentypen der Parameter des Unterprogramms. Ist das Unterprogramm eine Funktion, gehört der Datentyp des Rückgabewerts ebenfalls zu dieser Schnittstellenspezifikation.

signature (dt.) ↑Signatur.

simultaneity (dt.) ↑Gleichzeitigkeit, Simultanität.

simultaneous process (dt.) ↑gleichzeitiger Prozess.

Simultanverarbeitung (en.) ↑*multiprocessing*. Fähigkeit von einem ↑Betriebssystem zur ↑Parallelverarbeitung im ↑Mehrprogrammbetrieb. Manifestiert sich vor allem in speziellen Verfahren zur ↑Prozesseinplanung und durch ↑partielle Virtualisierung der ↑CPU.

single-address space (dt.) ↑Einzeladressraum.

single-level store (dt.) ↑einstufiger Speicher.

single-stream batch monitor (dt.) ↑Einzelstromstapelmonitor.

single-user mode (dt.) ↑Einbenutzerbetrieb.

Sitzung (en.) ↑*session*. Abschnitt im Zeitverlauf zwischen ↑Anmeldung und ↑Abmeldung, in dem das ↑Rechensystem bestimmte Anforderungen für eine Person oder für einen externen ↑Prozess nachgeht. Eine solche Anforderung kann einen einzelnen ↑Auftrag, einen ↑Dienst oder eine beliebige Abfolge von beiden beinhalten.

SJF Abkürzung für (en.) *shortest job first*. Strategie zur ↑Planung der von einem ↑Rechensystem zu erledigenden ↑Arbeit (↑*job*) in einem *Prioritätsverfahren*. Dabei wird jede Arbeit nach der Bearbeitungszeit sortiert und die Arbeit bevorzugt, deren Bearbeitung vermeintlich am schnellsten geschehen kann: je kürzer die Bearbeitungsdauer einer ↑Aufgabe, desto höher die ↑Priorität, wobei der Prioritätswert einen Zeitwert darstellt.
In dieser Grundidee stimmt das Verfahren mit ↑SPN überein — jedoch ist das der Planung zugrunde liegende Kriterium nicht die Dauer eines jeweiligen ↑Rechenstoßes eines ↑Prozesses, sondern ein kompletter ↑Stapelauftrag. Der entscheidende Punkt liegt darin, dass die Aufgaben für gewöhnlich nicht interaktiv bearbeitet werden (↑*batch mode*) und als ↑Stapel gepackt dem ↑Rechensystem im Ganzen zur Bearbeitung übergeben werden (↑*batch job*). Das Planungskriterium ist die für die Arbeit zu veranschlagende totale ↑Laufzeit. Darin enthalten sind alle ↑Wartezeiten, die ein zur Bewältigung der Aufgabe angesetzter Prozess (gegebenenfalls in Kooperation mit anderen Prozessen) erfahren wird.
Ein weiterer wichtiger Unterschied zu SPN ist die ↑vorlaufende Planung. Die dafür im Voraus bekannt zu gebenden Bearbeitungszeiten der Aufgaben werden damit nicht im normalen Betrieb ermittelt. Die Bestimmung einer solchen Bearbeitungszeit kann beispielsweise durch die in ↑Kommandosprache formulierte explizite Angabe einer ↑Frist (*time limit*) im ↑Stapelauftrag erfolgen, deren Quantifizierung vielleicht nur auf Erfahrungswissen zurückgreift. Ein anderer Ansatz wäre ein *Probelauf* des Stapelauftrags unter realistischen Bedingungen, das heißt, die Durchführung eines ↑Produktionsbetriebs. Auf Basis der so ermittelten Laufzeitparameter kann durch eine anschließende ↑Zeitreihenanalyse die voraussichtlich optimale Reihenfolge der Aufgabenbearbeitung bestimmt werden.

SJN Abkürzung für (en.) *shortest job next* ↦ ↑SJF.

Skriptsprache (en.) ↑*scripting language*. Programmiersprache zur Formulierung von einem zumeist kurzen und kompakten ↑Programm, das für gewöhnlich direkt durch einen ↑Interpreter zur Ausführung kommt (↑CSIM). Ein Hauptaspekt einer solchen Sprache liegt in der formalen Beschreibung von Anweisungsfolgen, um auf bestehende und in dem jeweils gegebenen ↑Rechensystem verfügbare Programme zugreifen und gegebenenfalls auch zu einen umfassenden Programmkomplex verknüpfen zu können. Ein weiterer, häufiger Einsatz besteht im Musterbau (*prototyping*) insbesondere von Anwendungsprogrammen.

slack time (dt.) ↑Schlupfzeit.

sleep state (dt.) ↑Schlafzustand.

SLIH Abkürzung für (en.) *second-level interrupt handler*. Bezeichnung für den ↑Unterbrechungshandhaber zweiter Stufe. Dieser ↑Handhaber wird indirekt durch eine ↑Unterbrechungsanforderung gestartet und beginnt seine Ausführung auf der durch das ↑Betriebssystem bestimmten ↑Unterbrechungsprioritätsebene. Für gewöhnlich ist dies die Prioritätsebene 0, das heißt, alle Unterbrechungsanforderungen sind zugelassen (*interrupts enabled*). Da ein solcher Handhaber als ↑Fortsetzung der entsprechenden ↑Unterbrechungsbehandlung erster Ebene (↑FLIH) fungiert und letztere durch die Hardware definiert auf Prioritätsebene l_{flih}, $l_{flih} \geq 0$ stattfand, ist die ihm zugebilligte Prioritätsebene des Betriebssystems l_{slih}, $0 \leq l_{slih} \leq l_{flih}$ explizit einzunehmen. Dies bedeutet zweierlei:

1. Sicherstellen, dass sämtliche Unterbrechungsbehandlungen der ersten Ebene abgeschlossen sind, das heißt, der durch solche Behandlungen aufgebaute ↑Laufzeitstapel wieder abgebaut wurde. Hier sind insbesondere ↑kaskadierte Unterbrechungen in Erwägung zu ziehen. Da die Behandlung von einem ↑IRQ oder einem ↑NMI jederzeit durch einen

(weiteren) NMI unterbrochen werden kann, ist auch im Falle einer ↑CPU, die nur über einen einstufigen IRQ verfügt (wie im Falle von ↑x86), gegebenenfalls eine mögliche Kaskadierung zu berücksichtigen.

2. Falls nicht bereits geschehen, die CPU oder den ↑PIC dazu veranlassen, nur die Unterbrechungsanforderungen ab Prioritätsebene l_{slih}, normalerweise $l_{slih} = 0$, zuzulassen. Da diese ↑Aktion aus einem Handhaber erster Stufe heraus erfolgt, muss dieser für die Quittierung der von ihm behandelten Unterbrechungsanforderung gesorgt haben. Anderenfalls besteht (vor allem bei ↑Pegelsteuerung) die Gefahr des sofortigen ↑Wiedereintritts in dieselbe Unterbrechungsbehandlung, mit der möglichen Folge einer unkontrollierbaren und gegebenenfalls ↑Panik verursachenden ↑Rekursion.

Diese beiden Schritte bewerkstelligt ein spezieller ↑Ausnahmezuteiler, der damit letztlich eine kontrollierte ↑Sequentialisierung von Handhaberausführungen betreibt. Jede dieser Ausführungen wird allerdings erst veranlasst, wenn von der durch eine ↑Unterbrechung betretenen ↑Systemebene zurück zur ↑Benutzerebene gewechselt werden sollte (↑AST).
Die durch den Handhaber erfolgende ↑Unterbrechungsbehandlung geschieht asynchron zum auf Benutzerebene unterbrochenen ↑Prozess. Je nach ↑Operationsprinzip des Betriebssystems und der damit verbundenen Art der ↑Synchronisation von Prozessen auf Systemebene, läuft die Unterbrechungsbehandlung der zweiten Ebene ebenso wie die der ersten Ebene asynchron ab oder sie erfolgt, anders als die der ersten Ebene, synchron (↑Wiedereintrittssperre, ↑nichtblockierende Synchronisation) zu bestimmten ↑Aktionen im Betriebssystem. In jedem Fall geht seine Ausführung immer im Namen des gerade stattfindenden, aber unterbrochenen Prozesses vor. Da dem Handhaber damit kein eigenständiger und eindeutiger Ablaufkontext (d.h., keine eigene ↑Prozessinkarnation) zur Verfügung steht, dürfen im Zuge seiner Ausführung weder direkt noch indirekt Operationen zur Wirkung kommen, die den unterbrochenen Prozess in die ↑blockierende Synchronisation zwingen. Sehr wohl aber kann letzterer für die ↑unilaterale Synchronisation in die Rolle als *Erzeugerprozess* gebracht werden, um nämlich das ↑Ereignis herbeizuführen, auf dessen Eintritt ein anderer Prozess gegebenenfalls wartet: ein signalisierender, ↑allgemeiner Semaphor kann benutzt werden, solange für die Operation dazu (↑V) nirgends ein ↑kritischer Abschnitt passiert werden muss.
Der Wirkungskreis des Handhabers ist nur bedingt eingeschränkt: er erstreckt sich sowohl auf das die Unterbrechung verursachende Gerät der ↑Peripherie als auch auf das Betriebssystem. Die ↑Aufgabe des Handhabers besteht in erster Linie darin, gepufferte ↑Daten zwischen Prozess und Gerät zu transferieren (↑Ein-/Ausgabe), den Prozessen die von den Geräten verursachten ↑Ereignisse mitzuteilen, Prozesse dadurch gegebenenfalls bereitzustellen und von den Prozessen Aufträge zum Absetzen eines ↑Ein-/Ausgabestoßes entgegenzunehmen. Vor allem aber ist sein Zweck, ↑Unterbrechungslatenz zu minimieren.

slot (dt.) ↑Zeitnische; Schlitz.

slot time (dt.) ↑Schlitzzeit (dem dt. Patentwesen entnommene fachbegriffliche Übersetzung).

SMP Abkürzung für (en.) ↑*symmetric multiprocessing*, (en.) ↑*symmetric multiprocessor* und (en.) ↑*shared-memory processor* — wobei letzterer einen ↑Multiprozessor oder ↑Mehrkernprozessor bezeichnet, der nicht zwingend symmetrisch aufgebaut sein muss, sondern insbesondere auch eine asymmetrische Auslegung haben kann (↑AMP).

software interrupt (dt.) durch Software hervorgerufene ↑Unterbrechung, (en.) ↑*trap*. Bezeichnung für eine explizit durch Ausführung eines ↑Unterbrechungsbefehls erzwungene Unterbrechung der Ausführung eines ↑Maschinenprogramms, das daraufhin eine ↑synchrone Ausnahme hervorbringt.

software layer (dt.) ↑Softwareschicht.

Softwareschicht (en.) ↑*software layer* Sammlung von ↑Text und ↑Daten (zusammengefasst als ein ↑Programm oder eine Menge davon) auf einer bestimmten Ebene in einem System, das

eine ↑hierarchische Struktur aufzeigt. Dabei müssen nicht alle Ebenen dieses Systems in Software vorliegen, das Gesamtsystem kann einen Komplex aus Soft-, Firm- und Hardware bilden. Programme, die in einer Schicht zusammengefasst sind, teilen sich nicht zwingend dasselbe Wissen über die dortigen Datenstrukturen, ↑Modul und Schicht sind zwei voneinander unabhängige Konzepte: Eine Schicht kann ein oder mehrere Module enthalten, ein Modul kann sich über eine oder mehrere Schichten erstrecken. So definiert vor allem die umfassende hierarchische Struktur, welche Programme zusammen eine Schicht ausmachen.

Solaris Mehrplatz-/Mehrbenutzerbetriebssystem, erste Installation 1982 (Sun Microsystems, zuerst ↑m68k, später insb. SPARC), auf Basis von ↑UNIX V7. Bis zur Umstellung der Quelltextbasis auf *Unix System V Release 4* geführt unter dem Namen ↑SunOS. Seit 2010 weiterentwickelt und vertrieben von Oracle — scheint auszusterben (Stand 2025).

source module (dt.) ↑Quellmodul.

SP Abkürzung für (en.) ↑*stack pointer*; Kürzel der Lehrveranstaltung ↑Systemprogrammierung.

special-purpose operating system (dt.) ↑Spezialbetriebssystem.

Speicher (en.) ↑*memory*, ↑*store*. Vorrichtung in einem ↑Rechensystem zur Aufbewahrung von Informationen auf einem ↑Speichermedium.

speicherabgebildet (en.) ↑*memory-mapped*. Eigenschaft einer ↑Entität, der für gewöhnlich keine ↑Adresse im ↑Arbeitsspeicher zugeordnet ist, durch ↑Maschinenbefehle im ↑Prozessadressraum dennoch zugänglich zu sein. So wird beispielsweise ein ↑Ein-/Ausgaberegister eines ↑Peripheriegeräts in den (logischen) ↑Adressraum eingeblendet, es erhält vom ↑Betriebssystem eine ↑logische Adresse, um mittels einfacher Lese-/Schreibbefehle der ↑Befehlssatzebene ↑speicherabgebildete Ein-/Ausgabe bewerkstelligen zu können.

speicherabgebildete Ein-/Ausgabe (en.) ↑*memory-mapped* I/O. Art von Ein-/Ausgabe, bei der die ↑Ein-/Ausgaberegister von einem ↑Peripheriegerät über eine normale ↑Adresse zugänglich sind. Jeder ↑Maschinenbefehl, der in Abhängigkeit von der ↑Adressierungsart wenigstens einen eine ↑Speicherstelle adressierenden Operanden hat, ist zur Durchführung von Ein-/Ausgabevorgängen geeignet: Eingabe entspricht Lesen und Ausgabe entspricht Schreiben, jeweils in Bezug auf eine Speicherstelle.

Speicherattribut (en.) ↑*memory attribute*. Kennzeichen eines bestimmten ↑Speicherbereichs, die die in dem korrespondierenden ↑Adressbereich erlaubten Operationen definieren sowie ihren Systemstatus festhalten. Dabei wird der Speicherbereich typischerweise durch einen ↑Seitendeskriptor oder einen ↑Segmentdeskriptor erfasst, je nach der ↑Adressabbildung.
Die erlaubten Operationen haben einen engen Bezug zum ↑Abruf- und Ausführungszyklus der ↑CPU: ausführen (*execute*) erlaubt den Abruf von einem ↑Maschinenbefehl, der Speicherbereich enhält ↑Text; lesen (*read*) erlaubt das Auslesen von einem ↑Speicherwort, der Speicherbereich enhält Text oder ↑Daten; schreiben (*write*) erlaubt die Veränderung des Inhalts eines Speicherworts, der Speicherbereich enhält Daten.
Als Statusinformation wird für gewöhnlich festgehalten, ob der Speicherbereich benutzt (↑*used bit*) oder beschrieben (↑*dirty bit*) wurde und ob dieser Bereich auf den ↑Hauptspeicher abbildet (↑*present bit*). Letzteres Statusbit unterstützt die ↑Ladestrategie, es wird vom ↑Betriebssystem verwaltet. Die anderen beiden Bits unterstützen die ↑Ersetzungsstrategie, sie werden von der MMU gesetzt und vom Betriebssystem gelöscht („klebrige Bits“).

Speicherausstattung (en.) ↑*memory equipment*. Einrichtung von ↑RAM oder ↑ROM (inkl. der verschiedenen Arten davon) in einem ↑Rechner (↑*main board*, ↑*motherboard*), wobei der Rechner<u>hersteller</u> Art und Umfang der jeweiligen Ausrüstung festlegt. Grundlage für diese Festlegung bildet dabei immer auch ein (durch die ↑CPU definierter) ↑realer Adressraum, indem nämlich jedem dieser Speicherbausteine ein bestimmter ↑Adressbereich zugewiesen wird (↑*memory map*). Jede in diesen Adressbereichen fallende ↑reale Adresse, die zur/bei

Ausführung von einem ↑Maschinenbefehl von der CPU auf den ↑Adressbus ausgegeben wird, selektiert einen vorhandenen Speicherbaustein. Reale Adressen, die außerhalb dieser Bereiche liegen, bedeuten einen schwerwiegenden Fehler (↑*bus error*) und werden abgefangen (↑*trap*).

Speicherauszug (en.) ↑*dump*. Das Sichtbarmachen eines zusammenhängenden Teils eines ↑Speichers (Duden). Für gewöhnlich geschieht die Darstellung des Speicherinhalts in numerischer Form, sehr ähnlich zum ↑Kernspeicherabzug. Jedoch bezieht sich der Vorgang nicht allein auf den ↑Hauptspeicher, sondern auch auf jede andere Art von Speicher eines ↑Rechensystems, das heißt, die ↑Register, dem ↑Arbeitsspeicher oder ↑Dateien in der ↑Ablage. Allen Speicherarten gemein ist die digitale Repräsentation der jeweils enthaltenen ↑Daten in Einheiten, die jeweils eine ↑Bitleiste fester Länge bilden, heute in aller Regel ein ↑Oktett. Dargestellt und gelistet werden die numerischen Werte dieser Einheiten in einem bestimmten Zahlensystem (gew. oktal, dezimal, hexdezimal), gegebenenfalls auch ergänzt um eine bestimmte Zeichenkodierung (gew. ↑ASCII). Typische ↑Dienstprogramme dafür sind `hexdump(1)` beziehungsweise `od(1)`).

Speicherbandbreite Maß der Transferrate von ↑Daten zwischen ↑Hauptspeicher und ↑CPU. Im Allgemeinen das Produkt aus ↑Wortbreite und ↑Taktfrequenz.

Speicherbarriere (en.) ↑*memory barrier*. ↑Maschinenbefehl, der eine bestimmte Ordnungsbedingung für die Speicheroperationen einer ↑CPU durchsetzt. Für gewöhnlich besagt diese Bedingung, dass Operationen, die vor der Barriere ausgegeben wurden, garantiert ausgeführt werden vor Operationen, die nach der Barriere ausgegeben werden. Beispiele eines solchen Maschinenbefehls sind `sync` (↑PowerPC) und `mfence` (↑x86).

Speicherbereich (en.) ↑*memory area*. Abschnitt bestimmter Länge im ↑Arbeitsspeicher. Der Bereich kann statisch oder dynamisch angelegt worden sein, also vor oder zur ↑Laufzeit der diesen Bereich benutzenden Entität (↑Prozessinkarnation). Im statischen Fall bestimmen Programmierpersonal, ↑Kompilierer, ↑Assemblierer, ↑Binder oder ↑Lader (d.h., ↑Betriebssystem) den Bereich. Dabei ist jeder Verfahrensschritt in der Lage Lokalität oder ↑Adresse des Bereichs vorzugeben, wohingegen nur die ersten drei genannten die Bereichslänge festlegen. Die Länge ist typischerweise ganzahlige Vielfache der Größe von einem ↑Speicherwort. Im dynamischen Fall bestimmen ein ↑Prozess selbst oder das Betriebssystem den Bereich, wobei der Prozess in aller Regel nur die Länge vorgibt und das Betriebssystem die (reale, logische, virtuelle) ↑Adresse für einen mindestens so großen Abschnitt im Rahmen der ↑Speicherzuteilung ermittelt und dem Prozess zuweist.

Speicherdirektzugriff (en.) ↑*direct memory access*. Auch kurz als ↑DMA bezeichnete Methode des Transfers von ↑Daten zwischen ↑Peripheriegerät und ↑Hauptspeicher ohne Hilfestellung durch die ↑CPU. Im Gegensatz zu ↑PIO beauftragt die CPU eine spezielle Steuereinheit (↑DMA *controller*), um die Daten transferiert zu bekommen. Die damit verbundene Ein-/Ausgabe in Bezug auf einen bestimmten ↑Speicherbereich läuft ohne Kontrolle der CPU ab. Folglich kann die CPU in der Zeit andere Berechnungen durchführen: Ein-/Ausgabe und Berechnungen überlappen sich. Zur Durchführung des Datentransfers durch die Steuereinheit sind verschiedene Arbeitsweisen gebräuchlich: ↑Stoßbetrieb, ↑Taktentzug, ↑Transparentbetrieb. Je nach Steuereinheit ist eine einzelne Transfereinheit das ↑Byte (d.h., ↑Oktett), ↑Wort (32-Bit) oder Halbwort (16-Bit). Darüberhinaus kann der Zugriff der Steuereinheit auf den Hauptspeicher über eine ↑reale Adresse, ↑logische Adresse oder ↑virtuelle Adresse erfolgen. Erwartet die Steuereinheit eine reale Adresse, muss diese das ↑Betriebssystem vor Absetzen des Transferauftrags gegebenenfalls zunächst aus einer vorgegebenen logischen/virtuellen Adresse herleiten, nämlich wenn ein ↑logischer Adressraum oder ↑virtueller Adressraum Quelle oder Ziel des Transfervorgangs ist. Kann die Steuereinheit jedoch mit einer logischen/virtuellen Adresse umgehen, fordert sie selbst bei der ↑MMU die Umwandlung in die korrespondierende reale Adresse an. In beiden Fällen muss aber das Betriebssystem dafür sorgen, dass der Speicherbereich für den Datentransfer reserviert ist, das heißt, der Bereich muss zeitweilig von ↑Überlagerung oder ↑Umlagerung beziehungsweise ↑Seitenumlagerung

ausgenommen sein. Das Ende des Transfers wird dem Betriebssystem für gewöhnlich durch eine ↑Unterbrechungsanforderung mitgeteilt.

speichergekoppelter Multiprozessor (en.) ↑*shared-memory processor*. ↑Parallelrechner, dessen Prozessoren einen gemeinsamen ↑Hauptspeicher besitzen und diesen mitbenutzen können (↑*shared memory*). Je nach Anzahl und Art der gekoppelten ↑Rechner, jeder davon gegebenenfalls auch nur ein einzelner ↑Rechenkern, ist der Hauptspeicher mehr oder weniger gleichförmig ausgelegt und in zeitlicher Hinsicht einheitlich zugänglich (*uniform*). Für gewöhnlich ist bei größeren Systemen der Hauptspeicher nur noch mit unterschiedlicher ↑Latenz zugänglich, gerade auch bezogen auf die Distanz der Strecke „hinter“ dem ↑Zwischenspeicher: nicht jeder ↑Prozessor ist gleich weit entfernt von einer gegebenenfalls gemeinsam und zugleich adressierten ↑Speicherzelle. Damit wird bei einem Fehlzugriff auf den Zwischenspeicher (↑*cache miss*) die Einlagerung der entsprechenden ↑Zwischenspeicherzeile unterschiedlich lange dauern können. So ist wenigstens die ↑Zugriffszeit auf dieselbe Speicherzelle von der Lokalität des Kerns abhängig, von der aus ein ↑Prozess den Zugriff ausübt (*non-uniform*). Darüber hinaus ist, in Abhängigkeit von der Konstruktionskomplexität des jeweiligen Systems, nicht zwingend sichergestellt, dass der gleichzeitige Zugriff auf dieselbe Speicherzelle von verschiedenen Lokalitäten aus auch immer für alle betreffenden Prozesse denselben Wert liefert (↑*cache coherence*). Auch wenn in solch einem ↑Rechensystem den Prozessen ein einheitlicher ↑Adressraum geboten wird, um direkt auf den gemeinsamen Hauptspeicher zugreifen zu können, bedeutet dies längst nicht, dass damit ein ↑nichtsequentielles Programm leicht von der Hand geht.

Speichergrundfläche (en.) ↑*memory footprint*. Bezeichnung für eine ↑Prozessgröße, die die Ausdehnung von einem ↑Prozess, genauer der entsprechenden ↑Prozessinkarnation, im ↑Hauptspeicher oder im ↑Arbeitsspeicher quantifiziert. Der Unterschied zwischen den beiden Betrachtungsweisen liegt in der Art des als ↑Bezugssystem genommenen ↑Prozessadressraums, das heißt, einerseits ↑realer Adressraum beziehungsweise ↑logischer Adressraum oder andererseits ↑virtueller Adressraum. Ersterer Fall definiert eine bestimmte Grundfläche nur im ↑Vordergrundspeicher, wohingegen letzterer Fall diese Fläche um Bereiche im ↑Hintergrundspeicher erweitert. In beiden Fällen umfasst die Ausdehnungsfläche jedoch jeden einzelnen ↑Speicherbereich, der dem Prozess zu einem Zeitpunkt statisch oder dynamisch zugeordnet ist. Gemeinhin bezieht sich diese Prozessgröße aber auf den Hauptspeicher.
Neben der einer Prozessinkarnation jeweils zugeschriebenen Grundflächengröße (im Hauptspeicher) ist gerade auch die des ↑Betriebssystems ein bedeutender Faktor, um nämlich die ↑Gemeinkosten (in Bezug auf Hauptspeicher) für eine bestimmte ↑Betriebsart zu quantifizieren. Für ein ↑Spezialbetriebssystem können diese Kosten weniger als ein Kilobyte betragen, wohingegen ein ↑Universalbetriebssystem leicht mehrere Megabytes mit Beschlag belegt.

Speicherhierarchie (en.) ↑*memory hierarchy*. Gesamtheit der in einer bestimmten Rangordnung stehenden ↑Speicher in einem ↑Rechensystem. Die ↑hierarchische Struktur ist durch eine Relation zwischen Speicherpaaren definiert, die von oben nach unten eine Zunahme an Speicherkapazität und, als Konsequenz daraus, ↑Zugriffszeit beschreibt.
Für die ↑Systemsoftware zeigt sich in logischer Hinsicht eine siebenstufige Organisation des Speichersystems, wobei jede dieser Stufen in funktionaler Hinsicht eben auch durch eine bestimmte Art und Weise des Zugriffs auf eine Speichereinheit und der dafür charakteristischen ↑Latenz ausgezeichnet ist. Demnach stellt sich die Hierarchie wie folgt dar:

	Speicherart	Geschwindigkeit	Zugriffsmethode	Akteur
1.	↑Registerspeicher	1	↑Maschinenbefehl	↑CPU
2.	↑Notizblockspeicher	2		
3.	↑Zwischenspeicher	2 – 10		
4.	↑Hauptspeicher	100		
5.	↑Festspeicher	10^3 – 10^4	↑Systemfunktion	↑Betriebssystem
6.	↑Ablage	10^3 – 10^7		
7.	↑Archiv	10^7 – 10^{12}	↑Systemprogramm	Mensch/Roboter

Die Speichergeschwindigkeit ist hier als relativer Wert aufgetragen, mit höherer Stufe spiegelt sie in etwa wider, um wieviel langsamer die Zugriffe auf die einzelnen Speichereinheiten geschehen. Die Verlangsamung ist vor allem technologiebedingt. ↑Bezugspunkt (1.) ist der ↑Prozessortakt, der den schnellstmöglichen Zugriff von einer Zeiteinheit (im oberen Pikosekundenbereich) vorgibt. Darüberhinaus sind dynamische Vorgänge in Betracht zu ziehen, etwa (3.) bedingt durch das Protokoll zur ↑Speicherkohärenz des Zwischenspeichers oder, sofern ↑virtueller Speicher vorhanden ist, durch ↑Seitenumlagerung — jenachdem, ob die zugegriffene ↑Seite (4.) ein- oder (5. – 6.) ausgelagert ist. Auch die Frage der technischen Auslegung des ↑Massenspeichers (5./6.), beispielsweise als ↑SSD oder ↑Plattenspeicher, spielt ebenso eine Rolle wie der Punkt, ob sogar (7.) Eingriffe von außen notwendig sind, um das ↑Speichermedium überhaupt erst verfügbar zu machen.
Die untersten vier Stufen (Register- bis Hauptsspeicher) umfassen den ↑Primärspeicher, auch ↑Vordergrundspeicher. Demgegenüber bilden Festspeicher und Ablage den ↑Sekundärspeicher und Archiv den ↑Tertiärspeicher, zusammen für gewöhnlich auch der ↑Hintergrundspeicher. In solch einer Anordnung erstreckt sich virtueller Speicher über den gesamten Hauptspeicher und einem vergleichsweise kleinen Teil (↑*swap area*) des Festspeichers oder der Ablage — zusammen der ↑Arbeitsspeicher.
Für gewöhnlich verwaltet das Betriebssystem die oberen vier Stufen (Haupt-/Festspeicher, Ablage und Archiv). Der Notizblockspeicher wird zumeist direkt vom ↑Maschinenprogramm beherrscht, der Zwischenspeicher ist unter Kontrolle der ↑CPU und Registerspeicher wird vom ↑Kompilierer oder, im Falle der Programmierung in ↑Assembliersprache, vom Maschinenprogramm selbst zugeteilt (↑*register allocation*).
Die Speichergeschwindigkeit nimmt von der untersten zur obersten Stufe ab beziehungsweise die Zugriffszeit auf die jeweiligen Speichereinheiten nimmt zu — ebenso vergrößert sich für gewöhnlich die Speicherkapazität. Dieser Sachverhalt führt zur Darstellung der Rangordnung innerhalb eines Speichersystems in Form einer ↑Speicherpyramide, wobei allerdings die Spitze die unterste Hierarchiestufe repräsentiert.

Speicherkachel (en.) ↑*memory tile*. Gemeinhin die Bezeichnung für ein bestimmtes architektonisches Bauglied eines kachelartig organisierten ↑Mehrkernprozessors (↑*tile processor*). Dieses Bauglied bildet einen Teil oder die Gesamtheit des mehreren ↑Rechenkernen eines solchen ↑Prozessors gemeinsamen, globalen ↑Hauptspeichers (↑*shared memory*). Ein derartig organisierter Mehrkernprozessor kann eine oder mehrere solcher Bauglieder aufweisen, je nach definierter ↑Rechnerarchitektur.
In anderen Zusammenhängen (↑*virtual memory*) steht der Begriff aber auch für eine Einheit fester Größe im Hauptspeicher, einer sogenannten ↑Kachel, die den Platz für genau einen ↑Seitenrahmen bestimmt. Diese Größe ist immer ganzzahlige Vielfache der Größe von einem ↑Speicherwort.

Speicherkohärenz (en.) ↑*memory coherency*. Grad und Art der zusammenhängenden (kohärenten) Ausführung gleichzeitiger Operationen auf den ↑Arbeitsspeicher, wobei die Operationen dieselbe ↑Speicherzelle referenzieren (Unterschied zur ↑Speicherkonsistenz) und jede Operation von einer anderen ↑CPU (in einem ↑Multiprozessor) oder einem anderen ↑Rechenkern (in einem ↑Mehrkernprozessor) ausgegeben wird. Die mit diesen Operationen verbundenen gleichzeitigen Zugriffe auf die Speicherzelle verlaufen kohärent, wenn jede einzelne ↑Aktion dazu denselben Wert liefert. Dies ist ein typisches Merkmal von einem ↑Zwischenspeicher in solchen Systemen.
Für gewöhnlich bestimmt die Konstruktionskomplexität des allen Prozessoren gemeinsamen Arbeitsspeichers (↑*shared memory*) den Grad/die Art der jeweils zugesicherten Kohärenz. So ist für einfachere Systeme mit einer eher geringen, im Zehnerbereich liegenden Prozessor-/Kernanzahl Kohärenz zumeist durch die Hardware sichergestellt. Steigt jedoch die Anzahl stark an, sichert die Hardware gegebenenfalls nur noch für bestimmte Gebiete (↑*tile*) kohärente Zugriffe auf gespeicherte ↑Daten zu, wenn überhaupt. In solchen Fällen ist ein ↑nichtsequentielles Programm gefordert, das in seiner eigenen Berechnungsvorschrift die kohärente

Sicht, die ein ↑nichtsequentieller Prozess sodann auf die betreffenden Speicherzellen haben soll, zusichert.

Speicherkonsistenz (en.) ↑*memory consistency.* Grad und Art der relativen Ordnung von Operationen auf den ↑Arbeitsspeicher, wobei die Operationen nicht dieselbe ↑Speicherzelle referenzieren (Unterschied zur ↑Speicherkohärenz). Je nach Grad/Art der *Konsistenz* wird zwischen verschiedenen Modellen unterschieden, die gängigen Varianten sind (in der Reihenfolge abnehmender Stärke der Konsistenz): ↑strikte Konsistenz, ↑sequentielle Konsistenz, ↑Prozessorkonsistenz, ↑schwache Konsistenz und ↑Freigabekonsistenz.
Grundlage bildet ↑gemeinsamer Speicher, der real oder virtuell zur Verfügung steht. Letzteres meint einen ↑Arbeitsspeicher, der sich physisch über mehrere lokale ↑Hauptspeicher in einem ↑Multiprozessor oder ↑Mehrkernprozessor erstreckt, in logischer Hinsicht aber als ein gemeinsamer ↑Speicherbereich in Erscheinung tritt (↑DSM). Demgegenüber ist mit real ein Arbeitsspeicher gemeint, dessen Hauptspeicheranteil physisch allen Prozessoren gemeinsamen ist. In beiden Fällen kann für gewöhnlich nur noch eine abgeschwächte Form der Konsistenz bei gleichzeitigen Speicherzugriffen gewährleistet werden.

Speichermarke Marke, die von der ↑MMU nur gesetzt, aber nicht wieder gelöscht wird („klebriges Bit“). Erst das ↑Betriebssystem löscht ein solche Marke, beispielsweise beim Durchlauf einer bestimmten ↑Ersetzungsstrategie für eine ↑Seite.

Speichermedium (en.) ↑*storage medium.* Träger für Informationen, Datenträger. Die Informationen sind fotografisch (Film), mechanisch (Lochkarte/-streifen), magnetisch (Band, Platte), optisch (CD, DVD) oder elektronisch (Halbleiter) gespeichert.

Speichermodell (en.) ↑*memory model.* Gebilde, das die inneren Beziehungen und Funktionen von ↑Speicher abbildet beziehungsweise schematisch veranschaulicht und gegebenenfalls vereinfacht, idealisiert (in Anlehnung an den Duden). Dabei unterscheidet sich die dadurch auch vorgegebene Form oder Beschaffenheit dieses Gebildes durchaus von Schicht zu Schicht in dem typischerweise eine ↑hierarchische Struktur aufweisenden ↑Rechensystem. Auf der einem ↑Programmablauf direkt betreffenden untersten Schicht (↑*instruction set level*) bilden solch elementare Dinge wie Größe und Struktur eines ↑Bytes, ↑Speicherworts und der ↑Register, einschließlich der darin festgelegten ↑Bytereihenfolge, die relevanten Aspekte, wie auch die Organisation des ↑Zwischenspeichers und einer ↑Zwischenspeicherzeile. Darüberhinaus ist die von der jeweiligen Schicht zugesicherten ↑Speicherkohärenz beziehungsweise ↑Speicherkonsistenz ein weiteres kennzeichnendes und ganz wesentliches Merkmal des Modells.

Speicherpartie (en.) ↑*memory item.* Teil, Abschnitt, Ausschnitt aus einem größeren Ganzen im ↑Speicher (in Anlehnung an den Duden). Die Partie entspricht gemeinhin einer ↑Seite oder einer ↑Zwischenspeicherzeile, kann aber auch einen Eintrag im ↑TLB oder die ↑Arbeitsmenge eines ↑Prozesses bezeichnen.

Speicherplan (en.) ↑*memory map.* Aufstellung der im ↑Adressraumbelegungsplan angegebenen ↑Adressbereiche, die sowohl die flüchtigen (alle Arten von ↑RAM) als auch nichtflüchtigen (alle Arten von ↑ROM) Teile des ↑Hauptspeichers erfassen:

	Adressbereich	Größe (KiB)	Verwendung	
1.	`00000000`–`0009ffff`	640	RAM	System
2.	`000f0000`–`000fffff`	64	RAM	BIOS (*shadow*)
3.	`00100000`–`090fffff`	147456	RAM	Erweiterung
4.	`fffe0000`–`fffeffff`	64	SM-RAM	(*system management*)
5.	`ffff0000`–`ffffffff`	64	ROM	BIOS

Auszugsweise dargestellt ist ein ↑realer Adressraum (32-Bit ↑Rechner: Toshiba Tecra 730CDT, 1996), und zwar nur die darin mit RAM und ROM bestückten Adressbereiche. Der im Schatten (*shadow*) liegende Bereich enthält das ↑BIOS als Kopie aus dem ROM (5. Zeile). Diese Kopie wird beim Neustart (*reset*) des ↑Rechners angelegt, um das BIOS sodann immer aus

dem RAM (2. Zeile) ausführen zu können und dadurch eine erhebliche Leistungssteigerung zu erreichen (↑Zykluszeit hier: 170 ns ROM vs. 60 ns RAM).

Speicherpyramide (en.) ↑*memory pyramid.* Art der Darstellung der in einem ↑Rechensystem typischerweise existierenden ↑Speicherhierarchie, eine Pyramide stilisierend, zumeist gezeichnet als gleichschenkliges Dreieck in der Ebene. Die Höhe des Dreiecks reflektiert die Anzahl der übereinander liegenden Systeme oder Sorten von ↑Speicher. In Bezug auf diese Speicher nehmen ↑Zugriffszeit und Kapazität von der Spitze bis zur Basis des Dreiecks zu.

Speicherschutz (en.) ↑*memory protection.* Maßnahmen in einem ↑Rechensystem, die eine Gefährdung der Integrität der Inhalte von ↑Speicherzellen abhalten oder diesbezügliche mögliche Integritätsschäden abwehren (in Anlehnung an den Duden). In konkreter Weise eine Vorrichtung, die zum ↑Schutz gegen unautorisierte Zugriffe auf den ↑Arbeitsspeicher konstruiert ist. Technische Grundlage ist zumeist eine ↑MMU oder ↑MPU, die vom ↑Betriebssystem entsprechend seines Schutzmodells so programmiert wird, dass ein ↑Prozess nur auf die ihm jeweils zugewiesenen Speicherbereiche zugreifen kann. Neben solchen hardwarebasierten Techniken gibt es auch softwarebasierte Ansätze, die ↑Typsicherheit von Programmabläufen voraussetzen. In solch einem Fall stellt der ↑Kompilierer sicher, dass das von ihm generierte Programm zur ↑Laufzeit (logisch) keine unautorisierten Speicherzugriffe hervorrufen kann.

Speicherschutzeinheit (en.) ↑*memory-protection unit.* Gerät zum ↑Speicherschutz durch ↑Einfriedung eines zu schützenden ↑Adressbereichs.

Speicherstelle (en.) ↑*memory location.* Ort, lokalisierbarer Bereich, in einem ↑Speicher.

Speichersteuerung (en.) ↑*memory controller.* Ein spezieller ↑Halbleiterbaustein, der den Transfer von ↑Daten zwischen ↑Prozessor und ↑Hauptspeicher steuert. Der Baustein ist entweder im Prozessor integriert oder auf der ↑Hauptplatine angebracht. Ersteres verkürzt die ↑Latenzzeit beim Speicherzugriff, macht den Prozessor aber abhängig von der Art der Speicherbausteine, die von der Steuerung behandelt werden können. Letzteres sorgt dagegen für eine stärkere Speicherunabhängigkeit des Prozessors, bedingt aber eine höhere Speicherzugriffslatenz.

Speicherstück (en.) ↑*memory chunk.* Einheit bildender Teil (↑Speicherbereich) eines Ganzen (↑Arbeitsspeicher). Jede Einheit ist gleich groß (↑Speicherwort, ↑Seite) oder von verschiedener Größe (↑Segment von Speicherwörtern oder Seiten).

Speicherverwaltung (en.) ↑*memory management.* Funktionseinheit der ↑Systemsoftware, die ↑Speicher im Auftrag oder anstelle des eigentlichen Besitzers betreut, in Obhut hat oder in Ordnung hält (in Anlehnung an den Duden). Besitzer ist ein ↑Prozessexemplar, dessen ↑Programm zur Ausführung durch die ↑CPU ganz oder teilweise im ↑Arbeitsspeicher vorliegen muss. Verwaltet wird der Arbeitsspeicher für ein solches Exemplar, und zwar kraft des ↑Betriebssystems und gegebenenfalls auch eines zusätzlichen ↑Laufzeitsystems.
Obligatorisches Merkmal dieser Funktionseinheit ist eine ↑Platzierungsstrategie, die bei Bedarf die ↑Speicherzuteilung für einen Speicher anfordernden ↑Prozess leistet. Demgegenüber sind sowohl ↑Ladestrategie als auch ↑Ersetzungsstrategie, zwei weitere Charakteristika dieser Funktionseinheit, optionale Merkmale, die für gewöhnlich nur im Falle von ↑Speichervirtualisierung unabdingliche ↑Systemfunktionen bilden.

Speicherverwaltungseinheit (en.) ↑*memory management unit.* Auch kurz als ↑MMU bezeichnetes Bauelement oder Bauteil in einem ↑Rechner, das für die ↑Adressabbildung zuständig ist. Dabei führt die MMU nicht wirklich die Verwaltung von ↑Speicher durch, sondern sorgt einerseits für die Überprüfung der Gültigkeit des durch die ↑CPU erfolgenden Zugriffs auf den ↑Hauptspeicher und andererseits dafür, dass eine gültige ↑logische Adresse in eine ↑reale Adresse umgewandelt wird, um einen als gültig erkannten Zugriff letztlich zuzulassen. Ist der Zugriff jedoch ungültig, konstatiert die MMU eine ↑Ausnahme, die den in dem Moment auf der CPU stattfindenden ↑Prozess entweder synchron (↑*trap*) oder asynchron (↑*interrupt*)

unterbricht. Die Verwaltung des Speichers ist ebenso wie die Programmierung der MMU eine zentrale ↑Aufgabe des Betriebssystems.

Speichervirtualisierung (en.) ↑*memory virtualisation.* Funktion von einem ↑Betriebssystem, um ↑Hauptspeicher entsprechend seiner Anlage einem ↑Prozess scheinbar komplett und exklusiv zur Verfügung zu stellen. Die Maßnahme schafft die Illusion nicht nur von einem Hauptspeichermonopol, sondern auch von einer Hauptspeicherkapazität, die der wirklich vorhandenen um Größenordnungen übersteigen kann: bei einer ↑CPU mit 64-Bit ↑Wortbreite kann der ↑Adressraum eines Prozesses einen ↑Adressbereich von $[0, 2^{64} - 1]$ abdecken — bei einer Hauptspeichergröße der Winzigkeit von vielleicht gerade einmal 8 GiB (d.h., 2^{33} ↑Byte). Grundlage dafür ist ein ↑virtueller Adressraum für den Prozess und ↑virtueller Speicher. Letzteres benötigt vor allem eine ↑Ladestrategie und eine ↑Ersetzungsstrategie, um den Hauptspeicher im Wesentlichen nur noch als ↑Zwischenspeicher zwischen CPU und ↑Umlagerungsbereich erscheinen zu lassen.

Speicherwort (en.) ↑*memory word.* Organisationseinheit in einem ↑Speicher (genauer: ↑Hauptspeicher), wobei jeder dieser Einheit eine ↑Adresse zugeordnet ist. Größe und Struktur einer solchen Einheit sind einem ↑Maschinenwort entsprechend ausgelegt.

Speicherzelle (en.) ↑*memory cell.* Gemeinhin die kleinste adressierbare Einheit im ↑Arbeitsspeicher oder im ↑Prozessor (↑*register*). Heute (2025) typischerweise ein ↑Byte (↑Oktett) in einem ↑Speicherwort: der Arbeitsspeicher ist wortorientiert, seine Größe ist ein ganzzahlig Vielfaches der ↑Wortbreite, aber er ist byteweise adressierbar.
Für gewöhnlich folgt die ↑Adresse eines Speicherworts jedoch einer gewissen Linienführung (↑*alignment*), um der ↑CPU einen schnellen Zugriff auf den ↑Hauptspeicher zuzusichern. Eine Wortadresse orientiert sich daher, wie schon die Kapazität des Arbeitsspeichers, an der jeweiligen Wortbreite, ihr Wert ist also Vielfaches von 1 (byteorientiert, gerade/ungerade) beziehungsweise 2, 4 oder 8 (wortorientiert, gerade).
Größe wie auch Anforderungen zur ↑Ausrichtung einer solchen Einheit hängen letztlich vom jeweils betrachteten ↑Bezugssystem ab. Die zuvor genannten Aspekte sind typisch für die ↑Befehlssatzebene, wohingegen bereits die ↑Maschinenprogrammebene vor allem durch das ↑Betriebssystem Abweichungen davon festlegen kann. Beispielsweise lässt sich für eine ↑CPU, die unausgerichtete Zugriffe nicht zulässt (*alignment* ↑*trap*), durch eine im Betriebssystem dann gegebenenfalls erfolgende ↑Ausnahmebehandlung die Abstraktion von der Notwendigkeit zur passenden Ausrichtung der ↑Daten im Hauptspeicher erreichen. Ein anderes Beispiel liefert die ↑Halde, die, in Abhängigkeit von ihrer Auslegung, eine minimale Zellengröße nicht selten als kleines Vielfaches der Wortbreite vorgibt: etwa die Größe des ↑Deskriptors eines ↑Lochs in der Halde, sofern jeder dieser Deskriptoren in dem durch ihn erfassten Loch daselbst Platz haben soll und damit keinen extra Speicherplatz belegt.

Speicherzugriffsfehler (en.) ↑*segmentation fault.* Ein beim Zugriff auf den ↑Hauptspeicher hervorgerufener ↑Segmentierungsfehler.

Speicherzuteilung (en.) ↑*memory allocation.* Allokation von einem ↑Speicherbereich, Zuweisung des Bereichs als ↑wiederverwendbares Betriebsmittel an eine ↑Prozessinkarnation. Vorgang beziehungsweise Ergebnis der ↑Platzierungsstrategie einer ↑Speicherverwaltung. Je nach Art der Anfrage durch einen ↑Prozess beziehungsweise abhängig von der Spezies des zuzuteilenden ↑Speichers läuft der Vorgang auf verschiedenen Ebenen und in unterschiedlicher Weise ab. Hinzu kommt, dass neben der eigentlichen Funktion, nämlich einem Prozess weiteren Speicher zuzuteilen, für gewöhnlich auch eine Anpassung des ↑Prozessadressraums an die veränderte Speicherbelegung vorzunehmen ist.
Abstrahiert das ↑Betriebssystem von der Lage eines Prozesses im ↑Hauptspeicher, nämlich indem mindestens ein ↑logischer Adressraum als ↑Prozessdomäne eingerichtet ist und damit die Abstraktionsebene bildet, wird dieser für gewöhnlich nur um die Größe des ↑Adressbereichs erweitert, der dem zuzuteilenden Hauptspeicherbereich entspricht — so denn das Betriebssystem eine Erweiterung um diese Größe noch zulässt. Diese Erweiterung geschieht

wenigstens um den bei der Zuteilungsanforderung angegebenen Wert, allerdings kann dem Prozess auch ein größeres Stück als angefordert zugeteilt werden (↑*placement policy*). Der zuzuteilende Hauptspeicherbereich wird allerdings auf physischer Ebene bestimmt (s. u.).
Abstrahiert das Betriebssystem von der Lage eines Prozesses im ↑Arbeitsspeicher, kommt, ergänzend zum logischen, ein ↑virtueller Adressraum als Abstraktionsebene beziehungsweise Prozessdomäne ins Spiel. Dieser Adressraum ist durch „generalisierte ↑logische Adressen" definiert, das heißt, er verleiht seinem logischen Vorbild funktional die Fähigkeit zur Abstraktion von den Örtlichkeiten (↑*virtual address*) beliebiger ↑Objekte innerhalb eines ↑Rechensystems. Geschieht die Speicherallokation nun auf Ebene des virtuellen Adressraums, so wird dieser analog der ihm zugrunde liegenden Ebene des logischen Adressraums erweitert: es läuft die gleiche Operation wie eben skizziert ab, der virtuelle Adressraum wird wenigstens um die Größe des der Zuteilungsanforderung entsprechenden Adressbereichs vergrößert. Für die Zuteilung eines Speicherbereichs reicht es allerdings, diesen nur im ↑Umlagerungsbereich (des Arbeitsspeichers) zu allozieren. Wirklicher Platz im Hauptspeicher wird dann erst bei Bedarf geschaffen, spätestens wenn der Zugriff darauf erfolgt (↑*fetch policy*). Dem betreffenden Prozess wird also ↑virtueller Speicher zugewiesen.
Die Zuweisung realen Hauptspeichers geschieht auf physischer Ebene (↑realer Adressraum), mit oder ohne Adressraumabstraktion durch das Betriebssystem. Auf dieser Ebene ist letztlich ein Hauptspeicherbereich zu finden, dessen Größe mindestens dem in der Zuteilungsanforderung angegebenen Wert entspricht. Auch dabei handelt es sich um einen Adressbereich, der jedoch im Gegensatz zu logischen/virtuellen Adressräumen immer mit Speicher an genau der durch seine ↑reale Adresse benannten Stelle verknüpft ist. So findet auf dieser Ebene die eigentliche, wirkliche Allokation von Hauptspeicher statt. Dazu sucht das Betriebssystem nach einem passenden und noch unbenutzten Adress-/Speicherbereich, auch als ↑Loch bezeichnet. Buch über solche Löcher zu führen und diese bei Bedarf für die Zuweisung von freien Hauptspeicher zu nutzen oder, umgekehrt, auch wieder verfügbar zu machen, ist die zentrale Aufgabe der Platzierungsstrategie einer Speicherverwaltung.
Im Falle von ↑Mehrprogrammbetrieb geschieht die Speicherzuweisung typischerweise per ↑Systemaufruf (`mmap(2)`, `brk(2)`), und zwar unabhängig vom Abstraktionsgrad, den das Betriebssystem in Bezug auf Prozessadressräume implementiert. Solch ein Betrieb erfordert die programmübergreifende, *globale* Verwaltung sowohl des Hauptspeichers als auch der Prozessadressräume, beides zentrale Aufgaben eines Betriebssystems. Betreibt das Betriebssystem ↑Adressraumisolation, ein typisches Merkmal logischer/virtueller Adressräume, dann kommt für gewöhnlich noch eine *lokale* Dimension der Speicherverwaltung hinzu, die innerhalb eines Prozessadressraums und oberhalb des Betriebssystems wirkt, und zwar im jeweiligen ↑Maschinenprogramm daselbst.
Auf Maschinenprogrammebene ruft der Prozess gemeinhin ein ↑Unterprogramm (`malloc(3)`) auf, um ↑Haldenspeicher zugeteilt zu bekommen. Dieses Unterprogramm ist typischerweise Bestandteil von dem mit einer Programmiersprache verbundenem ↑Laufzeitsystem. Der Unterprogrammaufruf kann in einen Systemaufruf münden, wenn nämlich in der ↑Halde kein Loch (d.h., freier Abschnitt) entsprechend der Größe des angeforderten Speicherbereichs vorhanden ist. Ein solcher Systemaufruf (z.B. `sbrk(2)`) schafft dann einen freien Abschnitt, mit dem ganz oder teilweise die Speicheranforderung bedient wird. In beiden Fällen sorgt eine bestimmte Platzierungsstrategie für die Verortung des angeforderten Speicherbereichs, wobei durch das Betriebssystem zuerst die Zuweisung an den betreffenden Prozessadressraum geschieht und dann durch das Laufzeitsystem innerhalb dieses Adressraums die Zuweisung an den anfordernden ↑Handlungsstrang erfolgt. Die dabei verfolgten Strategien für die Auswahl eines passenden Lochs — das heißt, der ↑Ressource zur Platzierung zu speichernder ↑Daten — sind für gewöhnlich ebenenspezifisch ausgelegt in Bezug auf Betriebssystem einerseits und Laufzeitsystem andererseits.

Sperrfreiheit (en.) ↑*lock-freedom.* ↑Fortschrittsgarantie für ↑nichtblockierende Synchronisation. Diese Garantie besagt, dass irgendein ↑Prozess eine Operation in einer endlichen Anzahl von Schritten vollenden kann, ohne Rücksicht auf die relativen Geschwindigkeiten der anderen

Prozesse. Eine im Vergleich zu ↑Wartefreiheit schwächere Zusicherung, die zwar einem System (von Prozessen), nicht jedoch jedem einzelnen Prozess darin Fortschritt garantiert: gerät ein Prozess in eine ↑Konkurrenzsituation, birgt dies die Gefahr von ↑Verhungern.
Der Vorteil dieser Form von nichtblockierender Synchronisation liegt in der für gewöhnlich einfacheren Umsetzung, als dies im wartefreien Fall getan ist. Gilt für das ↑Betriebssystem keine ↑Echtzeitbedingung, ist sperrfreie nichtblockierende Synchronisation für gewöhnlich unproblematisch und liefert in vielen Beispielen eine praktikable Lösung.

Sperrzeit (dt.) ↑*blocking time*. Bezeichnung für die Zeitspanne, während der ein ↑unteilbares Betriebsmittel, R, von einem ↑Prozess (*owner*), P_o, belegt und damit für andere Prozesse gesperrt ist. R steht in der Zeit *exklusiv* P_o zur Verfügung. Dabei ist die Zeitspanne der exklusiven Nutzung durch P_o wie folgt definiert:

$$t_{detent} = |p_{redemption} - p_{allocation}|,$$

mit $p_{allocation}$ als Zeitpunkt der Zuteilung und $p_{redemption}$ als Zeitpunkt der Rücknahme von R, wobei der Zuteilung die Anforderung (*acquire*) und der Rücknahme die Freigabe (*release*) von R durch P_o vorausgeht.
Die Zeitspanne hat direkten Einfluss auf die ↑Wartezeit eines Prozesses, der bei Anforderung von R, und zwar wenn R durch P_o gesperrt ist, blockiert und in diesem ↑Prozesszustand verharrt, bis ihm R zugeteilt wird. Diese Wartezeit hängt ab von der Anzahl der Prozesse, die im ↑Wettstreit um R stehen und deren Zuteilung erwarten:

$$t_{block} \leq N_r * t_{detent},$$

mit $N_r \geq 1$ gleich der Anzahl der auf der Warteliste für R stehenden Prozesse, die das durch P_o gesperrte Betriebsmittel (R) angefordert hatten. Im Falle einer begrenzten Anzahl von Prozessen, die dieses Betriebsmittel in logischer Hinsicht gemeinsam benutzen, ergibt sich damit eine *obere Grenze* für die Zeitspanne der ein Prozess von der Zuteilung des ↑Prozessors aufgrund der Anforderung eines gesperrten unteilbaren Betriebsmittels ausgesperrt sein kann — sofern die Möglichkeit einer ↑Verklemmung einmal außer Acht gelassen wird.

Sperrzone (en.) ↑*exclusion zone*. Nicht für andere ↑Prozesse zugänglicher abgegrenzter räumlicher Bereich, ein besonderer ↑kritischer Abschnitt. Die Ausgrenzung geschieht durch Setzen einer Sperre, wodurch eine Funktion, die die Auslösung ↑gleichzeitiger Prozesse bewirken kann, außer Kraft gesetzt wird. Diese Funktion bleibt bis zur Aufhebung der Sperre abgeschaltet. Typisches Beispiel ist die ↑Unterbrechungsbehandlung oder der ↑Umschalter; eine entsprechende Vorrichtung, um zu verhindern, dass diese Funktionen zur Ausführung gelangen, ist eine dem ↑Operationsprinzip des ↑Betriebssystemkerns zudienende ↑Handlungssperre.

Exkurs Eine solche Zone kann durch eine vergleichsweise einfache Maßnahme zur Absicherung (*safeguard*) eingegrenzt werden. Nachfolgend ein Beispiel, mit dem je nach Wunsch eine bestimmte Handlungssperre erhoben und wieder zurückgenommen werden kann. Die jeweils unterstützen Sperren definiert folgender ↑Datentyp:

```
typedef enum safeguard {
    /* exclusion zones */
    UICS = 1,                          /* uninterrupted critical section */
    NRCS = 2,                          /* non-reentrant critical section */
    NPCS = 3,                         /* non-preemptive critical section */
    /* bit fields used to encode attributes */
    SGZF = 0x00ff,                              /* zone field, mandatory */
    SGQF = ~SGZF                             /* qualifier field, optional */
} safeguard_t;

#define ZONE NPCS             /* default zone, if not redefined elsewhere */
```

Ausgangspunkt ist ein ↑nichtsequentielles Programm mit wenigstens einem Bereich, der (a) eine Ausführung gleichzeitiger Prozesse nicht zulässt und für den (b) ↑wechselseitiger Ausschluss nicht praktizierbar ist. Ein solcher Bereich (bspw. S. 50) wird wie folgt betreten:

```
inline void enter(safeguard_t mode) {
    int zone = mode & SGZF;
    if (zone == UICS)
        avert(IRQ);
    else if (zone == NRCS)
        guard(mode >> 8);
    else if (zone == NPCS)
        state(&being(ONESELF)->mood, BARRED);
}
```

Je nach Angabe wird eine der oben genannten Sperren erhoben — oder eben überhaupt keine Sperre, falls keine Bedingung erfüllt ist. Letzteres wird hier bewusst als Merkmal und nicht als ↑Fehler eingestuft, damit ein ↑nichtsequentieller Prozess durch ein ↑nichtsequentielles Programm abstrahierend von seinem wirklichen Erscheinungsbild beschrieben werden kann: denn wenn sich jeder ↑Handlungsstrang eines solchen Prozesses strikt *kooperativ* verhält, kann auf explizite ↑Synchronisation verzichtet werden. Im gegebenen Fall ist dies eine *Konfigurationsentscheidung*, die zur Übersetzungszeit getroffen werden können soll.
Unabhängig davon beziehen sich ↑Unterbrechungssperre (UICS) und ↑Wiedereintrittssperre (NRCS) auf ↑asynchrone Ausnahmen der ersten (↑FLIH) beziehungsweise zweiten (↑SLIH) Stufe. Die ↑Verdrängungssperre (NPCS) kommt durch Manipulation des ↑Prozesszustands zustande, nämlich indem über den ↑Prozesszeiger (`being`) des laufenden Prozesses dieser (`ONESELF`) als gesperrt markiert wird. Ein so markierter Prozess behält seinen Prozessor bis zum Abschluss der in dem Bereich durchzuführenden ↑Aufgabe (↑*run to completion*), das heißt, bis er den abgesicherten Bereich verlässt:

```
inline void leave(safeguard_t mode) {
    int zone = mode & SGZF;
    if (zone == UICS)
        admit(IRQ);
    else if (zone == NRCS)
        clear(mode >> 8);
    else if (zone == NPCS)
        state(&being(ONESELF)->mood, CLEAR|BARRED);
}
```

Für beide Operationen ist die inzeilige (*inline*) Verwendung wichtig. Dadurch kann der ↑Kompilierer zur Übersetzungszeit die bedingte Anweisung berechnen und direkt die gewünschte Operation absetzen. Die Fallunterscheidungen werden nicht von den Prozessen selbst, das heißt, nicht zur ↑Laufzeit des nichtsequentiellen Programms ausgewertet.

Spezialbetriebssystem (en.) ↑*special-purpose operating system*. Bezeichnung für ein ↑Betriebssystem, das nur speziell und für gewöhnlich sehr eingeschränkt eingesetzt werden kann und damit nur einzelnen Anwendungszwecken dient; Gegenteil von ↑Universalbetriebssystem. Die Funktionen eines solchen Betriebssystems sind typischerweise in Hinblick auf spezielle Bedürfnisse einer bestimmten Klasse von Anwendungen optimiert. Dies betrifft insbesondere auch den Umfang der von dem Betriebssystem zu leistenden Aufgaben: nur die Funktion, die unbedingt erforderlich ist, wird durch das Betriebssystem auch bereitgestellt beziehungsweise ist im Betriebssystem überhaupt realisiert. Schon allein damit ist ein solches Betriebssystem in räumlicher Hinsicht (z.B. Platzbedarf im ↑Hauptspeicher) für gewöhnlich schlanker als ein Universalbetriebssystem. Auch in zeitlichen Belangen (z.B. ↑Latenzzeit oder ↑Laufzeit) zeigt sich zumeist ein anderes Bild: das Zeitverhalten eines solchen Betriebssystems ist oftmals nicht nur besser, sondern auch vorhersagbar. So ist ein typischer Anwendungsfall insbesondere der ↑Echtzeitbetrieb.

spinlock (dt.) ↑Umlaufsperre.

SPN Abkürzung für (en.) *shortest process next*. Bezeichnung einer Strategie für die nicht ↑präemptive Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor in einem *Vorrangverfahren*. Die Aufgaben werden nach zunehmender ↑Bedienzeit sortiert. Es erhält die Aufgabe den Vorzug, deren Bearbeitung am schnellsten geschehen kann: je kürzer die Bearbeitungsdauer einer Aufgabe durch einen ↑Prozess, desto höher seine ↑Priorität, wobei der Prioritätswert einen Zeitwert darstellt.
Im Unterschied zu ↑SJN zielt diese Strategie gemeinhin auf die Förderung von Interaktion in einem ↑Rechensystem ab (↑*conversational mode*). Die dazu erforderliche ↑mitlaufende Planung setzt die Bearbeitungszeit einer Aufgabe der Dauer des einzelnen ↑Rechenstoßes eines Prozesses gleich. Das Verfahren steht vornehmlich im Zusammenhang mit der Planung interaktiver Prozesse, deren meist kurze Rechenstöße häufig nur dazu dienen, ↑Ein-/Ausgabe anzustoßen und damit jeweils einen Dialog mit einem „externen“ Prozess zu führen.
Mit ↑SJN gemeinsames Merkmal ist die ↑probabilistische Planung, da die jeweiligen Bearbeitungszeiten für gewöhnlich nur abgeschätzt werden können (↑*time series analysis*). Darüberhinaus wirkt das Verfahren nichtpräemptiv, da ein eintreffender Prozess niemals den Prozessorentzug bewirkt: auch dann nicht, wenn die abgeschätzte Ausführungszeit des eintreffenden Prozesses kürzer ist als die des gegenwärtig auf dem Prozessor stattfindenden Prozesses. Wie bei ↑FCFS, ↑RR oder ↑VRR auch, werden eintreffende Prozesse lediglich der ↑Einplanung zugeführt. Die ↑Einlastung eines Prozesses erfolgt immer erst dann, wenn der gegenwärtig stattfindende Prozess die Bearbeitung seiner (Teil-) Aufgabe beendet.

Exkurs Dreh- und Angelpunkt zur Abschätzung der Länge des zukünftigen Rechenstoßes eines Prozesses ist eine fortlaufend stattfindende Zeitmessung zur Bestimmung der Dauer eines jeden Rechenstoßes, den der Prozess bisher verrichtet hat. Im Fokus steht dabei die Operation zum ↑Prozesswechsel. Diese Operation beendet den Rechenstoß des Prozesses, der den ↑Prozessor abgibt und startet den Rechenstoß des Prozesses, der den Prozessor zugeteilt erhalten hat. In dieser Operation sind die ↑Zeitstempel zu setzen, um zu gegebener Zeit die jeweilige ↑Stoßzeit eines Prozesses bestimmen zu können. In ↑C sei die entsprechende Funktion wie folgt formuliert:

```
time_t burst(burst_t *this) {                    /* normalised CPU burst */
    return spent(this, this->close);
}
```

Dabei definiert `close` den Zeitpunkt (Endzeit), zu dem der Rechenstoß abgeschlossen wurde und die Funktion `spent` liefert die bisher von dem Prozess verbrauchte Zeit als Nettobetrag:

```
time_t spent(burst_t *this, time_t time) {           /* net time consumed */
    return time - this->start - this->delay;
}
```

Hier markiert `start` den Zeitpunkt (Startzeit), zu dem der Rechenstoß gestartet wurde. Beide Werte (`close` und `start`) repräsentieren Zeitstempel und sind damit auf sehr lange Sicht eindeutig gültige Markierungen, weshalb bei der Differenzbildung auch nicht der absolute Betrag berechnet werden muss. Attribut `delay` definiert die aufgelaufenen Verzögerungszeiten seit dem letzten Prozesswechsel. Im betrachteten Fall wird dieser Wert in der zentralen Operation zur Lenkung (`guide`, ↑AST) von Unterbrechungsbehandlungen ermittelt, die dazu dem Zweck entsprechend wie folgt anzupassen ist (vgl. S. 28):

```
__attribute__ ((fastcall)) void guide(flih_t flih, train_t *this) {
    stage(this, ENTER);            /* enter interrupt handler incarnation */

    time_t start;                              /* user-level offset time */
    if (stage(this, PROVE|USER))     /* user-level CPU-burst interrupted? */
        start = timestamp();                 /* yes, remember offset time */

    /* control first- and second-level interrupt handler */

    if (stage(this, PROVE|USER))          /* resume user-level CPU burst? */
        being(ONESELF)->time.delay += (timestamp() - start); /* yes, size */

    stage(this, ENTER);            /* leave interrupt handler incarnation */
}
```

Die Operation prüft, welcher Ebene (`stage`) der unterbrochene Prozess entsprang. Fand dieser Prozess vor seiner ↑Unterbrechung eben noch auf der ↑Benutzerebene (`PROVE|USER`) statt, wird die Startzeit für seine Verzögerung genommen (`start`). Kehrt der Prozess nach erfolgter Unterbrechungsbehandlung auf der ↑Systemebene zurück zur Benutzerebene, wird die Endzeit für seine Verzögerung genommen sowie aus beiden Zeitstempeln die Verzögerungszeit berechnet und im ↑Prozesskontrollblock des betreffenden Prozesses aufsummiert (`time.delay`: bei 64-Bit Zahlendarstellung, wie hier angenommen, ist ein ↑arithmetischer Überlauf während der Lebenszeit des ↑Rechners nicht zu erwarten). Dazwischen läuft der Vorgang zur Steuerung und Überwachung von Unterbrechungsbehandlungen unverändert ab.

Die so akkumulierte Verzögerungszeit eines Prozesses ist in der Operation zum Prozesswechsel auf Null zu setzen. Diese Operation beendet den Rechenstoß des Prozesses, der den ↑Prozessor abgibt und startet den Rechenstoß des Prozesses, der den Prozessor zugeteilt erhalten hat. Demzufolge kann für den seinen Rechenstoß aufnehmenden Prozess noch keine Verzögerungszeit aufgelaufen sein. Nachfolgend ein Beispiel für diese Operation (vgl. S. 251):

```
void seize(process_t *task) {                          /* dispatch process */
    process_t *last = being(ONESELF);      /* access process control block */

    state(&last->mood, CLEAR|RUNNING);       /* cancel running state, only */

    last->time.close = timestamp();                    /* end of CPU burst */
    last->flow.crux = shift(task);         /* switch process, save context */
    task->time.start = timestamp();                  /* begin of CPU burst */
    task->time.delay = 0;                            /* no time delay, yet */

    state(&last->mood, FLUSH|RUNNING);       /* define running state, only */
}
```

Wenn ein Prozess seine Tätigkeit aufnimmt, wird die Startzeit (`time.start`) in seinem Prozesskontrollblock vermerkt und auch die Verzögerungszeit (`time.delay`) zurückgesetzt. Beendet der Prozess diese Tätigkeit, um kurz danach (hier durch einen ↑Koroutinenwechsel) die Kontrolle über den Prozessor abzugeben (`shift`), wird die Endzeit (`time.close`) genommen. Die Differenz aus beiden Zeitwerten ergibt die wirkliche Länge des letzten Rechenstoßes eines jeweiligen Prozesses, sie wird aber erst zur Prognose der Länge des zukünftigen Rechenstoßes gebildet (s. nächsten Absatz). Unter der Annahme des ↑Lokalitätsprinzips wird die Länge des nächsten (unbekannten, abzuschätzenden) Rechenstoßes eines Prozesses nach seiner Wiederaufnahme mit einer gewissen Wahrscheinlichkeit der Länge seines letzten (bekannten) Rechenstoßes entsprechen.

Um nun die Länge des nächsten Rechenstoßes eines Prozesses zu prognostizieren, wäre ein naheliegender Ansatz, die gemessenen Längen zu akkumulieren, um aus der damit berechneten prozessspezifischen totalen Rechenzeit das *arithmetische Mittel* zu bilden und als Schätzwert heranzuziehen. Nachfolgende Funktion skizziert die grundlegenden Berechnungsschritte:

```
time_t guess(process_t *task) {                /* weighted burst average */
    time_t burst = burst(&task->time);                        /* fix length */

    task->time.total += burst;                        /* settle this burst */
    task->time.count += 1;                          /* book the calculation */

    return decay(task->time.total / task->time.count, burst); /* estimate */
}
```

Für einen Prozess die totale Rechenzeit sowie die Anzahl der durchgeführten Akkumulationen festzuhalten, ermöglicht es, ihn in Bezug auf seine durchschnittliche Rechenstoßlänge zu charakterisieren: kleinere Werte stufen ihn als interaktiv oder ein-/ausgabeintensiv ein, größere Werte beschreiben ihn als rechenintensiv. Allerdings bleibt auf Basis allein des arithmetischen Mittels die Vorhersage der als nächstes zu erwartenden Rechenstoßlänge recht ungenau. Dies ist auch dann immer noch der Fall, wenn, wie hier skizziert, dieser Mittelwert (durch `decay`, s.u.) mit der zuletzt gemessenen Rechenstoßlänge (`burst`) gewichtet verrechnet wird, um das Ergebnis dann als neuen Schätzwert für den Prozess festzuhalten.
Schwachstelle der gezeigten Lösung ist die effektiv fehlende Rückkopplung der Schätzung in Bezug auf das gegenwärtige Prozessverhalten. Hierzu muss der jeweils bestimmte Schätzwert in die „Prozessgeschichte“ eingehen, nämlich vermerkt und in die nächste Schätzung wieder eingespeist werden. Eine Umsetzung zeigt nachfolgendes Beispiel, wobei hier die nunmehr als Option zu betrachtende Buchführung zur totalen Rechenzeit und Charakterisierung eines Prozesses weggelassen wurde:

```
time_t guess(process_t *task) {              /* weighted burst estimation */
    return decay(task->time.guess, burst(&task->time));
}
```

In diesem Beispiel — bei Verwendung dieser Funktion im Verlauf der Bereitstellung eines Prozesses (`ready`, s.u.) und dabei zur Aktualisierung des prozessspezifischen Schätzwerts (`time.guess`) — ergibt sich die erwartete Länge des nächsten Rechenstoßes eines Prozesses als ↑exponentielle Glättung des Durchschnittswerts von der vorangegangenen Schätzung sowie der aktuellen Messung, wobei diese beiden Werte einer Wichtung unterzogen werden. Die Glättung sorgt dafür, dass die Bedeutung eines zurückliegenden Rechenstoßes mit zunehmendem „Alter“ abnimmt und des aktuellen Rechenstoßes hoch bleibt. Erreicht wird dies durch nachfolgend skizzierte Filterfunktion (*decay filter*), bei der die Stärke der Glättung durch einen *Wichtungsfaktor* einstellbar ist:

```
time_t decay(time_t guess, time_t burst) {/* exponential weighted average */
    extern float trend();                 /* weighting factor 0 <= f <= 1 */
    return (guess * (1 - trend())) + (burst * trend());
}
```

Hat der Wichtungsfaktor (`trend`) den Wert 1, wird die bisherige Entwicklung der Rechenstöße komplett ausgeblendet. Für andere Werte ($0 \leq$ `trend` < 1) dominieren die bisher vollbrachten Rechenstöße eines Prozesses mehr oder weniger stark. Diesen Wichtungsfaktor zu bestimmen und eine diesbezügliche Wichtungsfunktion herzuleiten, die dem wirklichen ↑Programmablauf sehr nahe kommt, ist der eigentliche Knackpunkt des Verfahrens
Da die Zeitmessungen erst beim Prozesswechsel (`seize`) geschehen, ist die Abschätzung der Länge des nächsten Rechenstoßes des gerade stattfindenden Prozesses vor dem Wechsel zwecklos: die Endzeit seines Rechenstoßes wurde noch nicht festgehalten, das heißt, der gegenwärtige Rechenstoß des Prozesses ist noch nicht abgeschlossen. Geschieht der Umschaltvorgang (`seize`), weil der Prozess auf ein ↑Ereignis warten muss und daher in den ↑Prozesszustand <u>blockiert</u> übergegangen ist, wird die ↑Einplanung immer auf Veranlassung eines anderen Prozesses, den ↑Leerlaufprozess eingeschlossen, erfolgen: also nach dem Prozesswechsel. In dem Fall liegen Start- und Endzeit des letzten Rechenstoßes des einzuplanenden Prozesses vor, womit die Abschätzung des nächsten Rechenstoßes dieses Prozesses im Moment seiner erneuten Bereitstellung möglich ist.
Wie eingangs erwähnt arbeitet das Verfahren <u>nicht</u> präemptiv, womit es insbesondere auch

nicht auf ↑Zeitscheiben basiert: die für ↑RR benötigte Funktion (`check`, S. 272) zum regelmäßigen Prozessorentzug entfällt, womit die in diesem Fall bedingt erfolgende erneute Bereitstellung des Prozesses durch sich selbst eben auch nicht geschehen kann und daher die zur Aufnahme auf die ↑Bereitliste nötige Rechenstoßlängenabschätzung noch vor dem Prozesswechsel nicht in Frage kommt. Anders verhält es sich jedoch, wenn der Prozess von sich aus pausiert (`pause`, S. 92). Die Annahme, ein solches Vorgehen ist wirkungslos und bringt nichts außer Zeitverschwendung, da der pausierende Prozess ja nach wie vor der kürzeste ist, muss allerdings nicht stimmen.
Nicht präemptiv zu arbeiten bedeutet nicht, eine Prozessbereitstellung im Verlauf einer ↑Unterbrechungsbehandlung sei nicht möglich. Diese Arbeitsweise bedeutet lediglich, dass es aus einen solchen Verlauf heraus nicht zur ↑Einlastung eines zeitnah bereitgestellten Prozesses kommt. Folglich kann das Verfahren sehr wohl Prozesse einplanen, deren Rechenstoßlängen kürzer sind als die des im Zustand laufend befindlichen Prozesses. Würde letzterer nun pausieren wollen, könnte demnach das Szenario eintreten, dass sich der stattfindende Prozess zugunsten eines kürzeren Prozesses selbst bereitstellt und dazu noch vor Prozessorabgabe die erwartete Länge seines nächsten Rechenstoßes bekannt sein müsste.
Die einfachste Lösung dazu wäre, einem Prozess keine Möglichkeit zum Pausieren zu eröffnen, indem etwa dem ↑Maschinenprogramm dazu kein (und auf `pause` hinauslaufender) ↑Systemaufruf zur Verfügung steht — ein nicht ganz untypischer Ansatz. Alternativ kann immer dann, wenn die Endzeit des aktuellen Rechenstoßes noch nicht bekannt ist, der zuletzt für den betreffenden Prozess zur Bereitstellung herangezogene Schätzwert erneut verwendet werden. Diese Lösung zeigt nachfolgendes Beispiel (vgl. S. 92):

```
void ready(process_t *task) {                /* schedule according to SPN */
    process_t *self = being(ONESELF);       /* use own process descriptor */

    assert((task != self) || valid(&self->mood, PENDING | IDLE));

    if (task != self)                                        /* not myself? */
        task->time.guess = guess(task);     /* yes, estimate burst length */

    state(&task->mood, READY);                /* set process ready to run */

    ticket(&task->line, task->time.guess);              /* define sort key */
    enlist(labor(), &task->line);      /* sort descriptor into ready list */
}
```

Der Schätzwert (`time.guess`) wird nicht neu berechnet und damit auch nicht aktualisiert, wenn der gegenwärtige Prozess (`ONESELF`) sich selbst (`task`) bereitstellen wollte. Damit fließt die aus aktuellem Prozessverhalten resultierende Rechenstoßlänge für einen Prozess, der pausieren und somit sich selbst auf die Bereitliste setzen müsste, nicht in die Berechnung der zu erwartenden Länge seines nächsten Rechenstoßes ein. Nur wenn die Bereitstellung durch einen anderen Prozess erfolgt, kann die Länge des jüngsten Rechenstoßes des jeweils bereitzustellenden Prozesses auch Berücksichtigung finden. Bei der Einplanung nicht immer und vor allem auf aktuelle Rechenstoßlängen zurückgreifen zu können, ist der eigentliche Nachteil dieser Lösung. Denn gerade die Berücksichtigung aktueller Werte ist für die möglichst genaue Vorhersage zukünftigen Prozessverhaltens besonders wichtig.
Eine andere Lösung besteht darin, die Zeitnahme für einen Prozess vorzuziehen. Der späteste Zeitpunkt, um die Rechenstoßlänge für einen pausierenden Prozess noch zu bestimmen, ist eine Zeitnahme an der Schnittstelle zum Planer, also noch vor dem eigentlichen Prozesswechsel durch den ↑Umschalter. In dem Fall geht der Zeitaufwand für alle Berechnungen, die durch die Einplanung selbst durchzuführen sind, nicht in die Rechenstoßlänge eines Prozesses ein. Gleiches gilt für die Durchführung der Zeitnahme bereits an der Schnittstelle zum ↑Betriebssystem (d.h., bei jeder ↑Ausnahme), wie es etwa zur Bestimmung von ↑Benutzerzeit und ↑Systemzeit gemeinhin praktiziert wird. In beiden Fällen kann es jedoch zu einer Verzerrung der Rechenstoßlängenabschätzung kommen, da nicht für jeden Prozess der Zeitaufwand innerhalb des Betriebssystems beziehungsweise Planers gleich ist. Die im Hauptausführungsstrang eines Prozesses innerhalb des Betriebssystems stattfindenden ↑Aktionen

sind insbesondere durch die beim ↑Systemaufruf übergebenen Parameterwerte bestimmt. Diese Aktionen finden synchron zum Prozess selbst statt, sie sind überwiegend prozessbezogen. Durch eventuelle ↑Unterbrechungen hervorgerufene Nebenausführungsstränge sind demgegenüber gemeinhin nicht prozessbezogen, sondern beziehen sich auf die ↑Peripherie und generieren bestimmte (von Prozessen ggf. erwartete) ↑Ereignisse. Die dabei anfallenden Zeitaufwände (d.h., Verzögerungszeiten) fließen daher, wie oben bereits skiziziert (`train`), auch nicht in die Berechnung der Rechenstoßlänge eines Prozesses ein.
Die Zeitnahme beim Prozesswechsel umgeht die eben geschilderte Problematik und ermöglicht zudem eine weitere Abstufung der Systemzeit. Wenn diese Lösung jedoch auch für pausierende Prozesse funktionieren soll, sind diese in einen bestimmten ↑Schwebezustand zu versetzen, der es anderen Prozessen ermöglicht, die noch ausstehende Bereitstellung und damit Einsortierung in die Bereitliste zu vollenden. Dieser Schwebezustand beginnt mit dem Wunsch zu pausieren, erstreckt sich über den Wechsel hinweg zum nächsten Prozess und endet damit, wenn letzterer seinen Vorgänger (nämlich der pausierte Prozess) auf die Bereitliste setzt. Indem der Nachfolgeprozess die Bereitstellung des pausierten Prozesses vornimmt, ist durch den erfolgten Prozesswechsel die Zeitnahme erfolgt, wodurch schließlich die Abschätzung der Rechenstoßlänge des Vorgängerprozesses möglich wird und somit der pausierte Prozess noch seinen Platz auf der Bereitliste finden kann.

spool (dt.) Spule. Abkürzung für (en.) *simultaneous peripheral operations online*.

spool area (dt.) ↑Spulbereich.

spooler (dt.) Spulmaschine, Druckpuffer.

spooling (dt.) ↑Spulen.

sporadic task (dt.) ↑sporadische Aufgabe.

sporadische Aufgabe (en.) ↑*sporadic task*. Eine in ↑Echtzeit zu bearbeitende ↑Aufgabe mit harter (strikter) ↑Termineinhaltung. Gegenteil von ↑aperiodische Aufgabe.

Sprungmarke (en.) ↑*jump label*. Bezeichnung für die ↑symbolische Adresse von einem ↑Maschinenbefehl, der als Ziel einer Programmverzweigung Verwendung finden kann.

Sprungvektor (en.) ↑*jump vector*. Bezeichnung für eine definierte ↑Adresse im ↑Arbeitsspeicher, an der die ↑Referenz (↑Zeiger) auf ein bestimmtes ↑Unterprogramm verzeichnet ist. Gemeinhin liegen solche Vektoren zusammengefasst in einem Feld (*array*), auf das mit einem Index zugegriffen wird, um einen bestimmten Vektor darin zu selektieren.

Spulbereich (en.) ↑*spool area*. Pufferplatz im ↑Hintergrundspeicher, um zwischengepufferte ↑Daten zur Eingabe an einen (schnellen) ↑Prozess oder Ausgabe an ein (langsames) ↑Peripheriegerät wieder „abspulen“ zu können. Ursprüngliches ↑Speichermedium hierfür ist das Magnetband (↑*spool*) und es kommen Magnetbandspulgeräte zur Ein-/Ausgabe zum Einsatz — woraus sich auch der Begriff „↑*spooling*“ ableitet. Moderne Ansätze nutzen für diese Vorgehendsweise jede Form von Fest- oder Wechselplatten als Speichermedium.

Spulen (en.) ↑*spooling*. ↑Stapelverarbeitung von Ein-/Ausgabeaufträgen. Die ein-/auszugebenden ↑Daten gelangen zunächst in einen Pufferbereich (↑*spool area*). Ein ↑Dämon (↑*spooler*) entnimmt diesem die Ein-/Ausgabeaufträge und leitet sie nacheinander an das jeweilige ↑Peripheriegerät weiter. Für gewöhnlich werden die Aufträge nach gewissen Kriterien sortiert auf einer Warteschlange gehalten (↑Ablaufplanung), bis die zur Abwicklung des Auftrags benötigte ↑Entität mit dem nächsten Auftrag bestückt werden kann (↑Einlastung). Typischer Anwendungsfall für solch eine Vorgehensweise zur Ausgabe ist das Drucken (`lpr(1)`, `lpd(8)`), wobei die Entität einem ↑Peripheriegerät („externer Prozess“ Drucker) entspricht. Für langsame Eingabegeräte und -medien (z.B. ↑Lochkarte) führt der Dämon eine Liste von wartenden Prozessen, die ausgelöst werden, sobald Eingabe für sie bereitsteht. In diesem

Szenario stellt der jeweilige (interne) Prozess, damit sein ↑Prozessor, die mit einer weiteren ↑Aufgabe einzulastende Entität dar.

spurious interrupt (dt.) unechte, unberechtigte ↑Unterbrechung.

SRAM Abkürzung für (en.) *static RAM*, (dt.) statischer ↑RAM.

SRP Abkürzung für (en.) *stack resource policy*. Bezeichnung für eine Strategie sowohl zur ↑Vorrangplanung von ↑Aufgaben durch einen ↑Prozessor als auch zur Vorbeugung einer *unkontrollierten* ↑Prioritätsumkehr (*unbounded* ↑*priority inversion*). Eine Verfeinerung und zugleich auch eine wesentliche Vereinfachung von ↑PCP.
Dem Ansatz nach hat ein ↑Prozess eine oder mehrere Aufgaben zu bearbeiten. Der Prozess verläuft *periodisch*, falls das Intervall zwischen aufeinanderfolgenden Anforderungen zur Aufgabenausführung konstant ist. Dieses Intervall wird auch als ↑Periode bezeichnet. Anderenfalls verläuft der Prozess *aperiodisch*. Die Aufgaben eines Prozesses sind im Voraus bekannt, ihre Anzahl ist endlich.
Jede Aufgabe ist einer bestimmten ↑Verdrängungsebene fest zugeordnet. Mehrere Aufgaben können dabei durchaus auf derselben Verdrängungsebene liegen, diese werden dann nacheinander (↑FCFS) von dem Prozess abgearbeitet. Wird eine Aufgabe, A_α, zur Ausführung bereitgestellt, geschieht die Verdrängung der laufenden Aufgabe, A_γ, vom Prozessor nur dann, wenn gilt: $level(A_\alpha) > level(A_\gamma)$.
Die Verdrängungsebene einer Aufgabe orientiert sich an den *relativen* ↑*Termin* dieser Aufgabe. Der relative Termin hat einen festen Wert, der angibt, innerhalb welcher Zeit ab Ausführungsanforderung (↑Bereitzeit) der Prozess diese Aufgabe beendet haben muss. Die einer Verdrängungsebene zugeordnete Nummer entspricht einem Wert, der umgekehrt proportional zum relativen Termin einer Aufgabe dieser Ebene ausfällt. Dementsprechend sind die Ebenen hierarchisch angeordnet.
Die Anforderung, eine bestimmte Aufgabe auszuführen, ist mit einer ↑Priorität verbunden. Je dringlicher die Ausführungsanforderung einer Aufgabe ist, desto höher ist die Anforderungspriorität. Für gewöhnlich entspricht die Priorität der Nummer ($\mathbb{N}_+$) der Verdrängungsebene einer Aufgabe. Hat die Ausführungsanforderung für Aufgabe A_α eine höhere Priorität als die Ausführungsanforderung für Aufgabe A_γ, wobei aber die ↑Ankunftszeit ersterer (für A_α) später liegt als die letzterer (für A_γ), dann muss A_α auf einer höheren Verdrängungsebene liegen als A_γ. Die Bedeutung der Priorität liegt in der *Terminkontrolle*, insbesondere auch durch ↑Prioritätsvererbung einen blockierten Prozess indirekt vorantreiben zu können, indem der ihn blockierende Prozess (auf höherer effektiver Priorität) beschleunigt fortgeführt wird. Jedoch ist die damit verbundene *direkte Blockierung* eines Prozesses nur vor ↑Startzeit seiner Aufgabe möglich: das heißt, wenn eine Aufgabe startet ist ein von ihr benötigtes ↑unteilbares Betriebsmittel niemals gesperrt. Da Betriebsmittelanforderungen niemals blockieren, fallen folglich auch keine diesbezüglich zusätzlichen ↑Prozesswechsel an. Letztere, und zwar höchstens zwei, sind nur durch ↑Verdrängung in Kauf zu nehmen: nämlich (1) um den verdrängenden Prozess zu starten, diesen durchzuführen und, sobald er seine Aufgabe erfüllt hat, (2) den verdrängten Prozess wieder aufzunehmen.
In Bezug auf die Verdrängung wirkt das Verfahren wie eine ↑kaskadierte Unterbrechung, das heißt, die vom Prozessor jeweils verdrängte Ausführung einer Aufgabe (A_γ) wird oben auf einem Stoß (↑*stack*) abgelegt, falls eine andere Aufgabe (A_α) bestimmter ↑Dringlichkeit zur Ausführung zu bringen ist ($level(A_\alpha) > level(A_\gamma)$) — diese Ausführungen sind Prozesse unterschiedlicher Verdrängungsebenen. Die zugrunde liegende ↑präemptive Planung verläuft damit stapelbasiert (↑LCFS). Gilt bei Bereitstellung von A_α, und zwar während A_γ in Bearbeitung ist, $level(A_\alpha) \leq level(A_\gamma)$, dann wird A_α entsprechend ihrer Priorität in den Aufgabenstoß einsortiert und die Bearbeitung von A_γ wird fortgesetzt.
Das Verfahren setzt voraus, dass kein Prozess während der Bearbeitung einer Aufgabe absichtlich wartet, indem er sich etwa bis zum Eintritt eines bestimmten ↑Ereignisses suspendiert. Endet mit der Periode die Aufgabe des Prozesses, darf er aussetzen (↑*run to completion*). Mit Bereitstellung derselben oder einer anderen Aufgabe kann der Prozess dann

wieder seine Tätigkeit aufnehmen. Dies gilt für alle Prozesse, die nach diesem Schema eingeplant und der ↑Einlastung zugeführt werden. Als Konsequenz daraus können alle Prozesse denselben ↑Laufzeitstapel gemeinsam nutzen, wodurch die ↑Speichergrundfläche des Systems wesentlich minimiert wird.
Eine weitere Konsequenz des Verfahrens ist, dass zum Zeitpunkt der Einlastung des Prozessors mit einem Prozess alle zur Bearbeitung seiner Aufgabe(n) benötigten Betriebsmittel implizit frei sind. Die einem Prozess durch seine Aufgabe implizit gegebene Verdrängungsebene definiert dann die aktuelle ↑Obergrenze des Systems. Allerdings bedeutet dies nicht automatisch die Vorbelegung dieser Betriebsmittel im Moment der Einlastung. Das Verfahren teilt die Betriebsmittel immer noch auf Anforderung zu und gibt diese auf Anforderung wieder frei. Im Falle eines aus mehreren Einheiten bestehenden unteilbaren Betriebsmittels kann dann die Anforderung einer Einheit immer noch die direkte Blockierung eines Prozesses bewirken. Allerdings ist in dem Fall der blockierte Prozess einer tieferen und der ihn blockierende Prozess einer höheren Verdrängungsebene zuzuordnen. Typisches Beispiel für solch ein Betriebsmittel ist ein von allem Prozessen gemeinsam benutzter Laufzeitstapel, den zu ermöglichen (nicht aber zu erzwingen) ein wichtiger Aspekt des Verfahrens überhaupt ist.
Angenommen ein niederpriorer Prozess, P_l, bearbeitet Aufgabe A_γ auf Verdrängungsebene L_π und belegt dazu eine Region bestimmter Größe, R_γ, oben auf dem Laufzeitstapel. P_l wird durch einen höher priorisierten Prozess, P_h, unterbrochen. P_h verdrängt P_l vom Prozessor, nimmt Aufgabe A_α auf Verdrängungsebene $L_{\pi+1}$ auf und belegt dadurch seinerseits eine Region bestimmter Größe, R_α, direkt angrenzend oberhalb von R_γ auf dem mit P_l gemeinsam benutzten Laufzeitstapel. In dem Moment wird P_l durch P_h dominiert, indem P_l durch P_h direkt blockiert und damit von der Prozessorzuteilung ausgenommen ist:

$$L_{\pi+1} > L_\pi \Longleftrightarrow A_\alpha^{\pi+1} > A_\gamma^\pi \Longleftrightarrow P_h > P_l$$

Würde P_l den Prozessor zugeteilt bekommen, bevor P_h seine Aufgabe (A_α) erledigt hat, besteht die Gefahr, dass durch die dann erfolgende weitere Bearbeitung von A_γ die von ihr beanspruchte Region (R_γ) in die von A_α beanspruchte Region (R_α) expandiert — sollte nämlich die Ausführung von A_γ (d.h., P_l) weitere ↑Daten auf dem Laufzeitstapel ablegen (*push*). Diese beiden Regionen (R_α und R_γ) sind zwei verschiedene Einheiten eines aus mehreren solcher Einheiten bestehenden unteilbaren Betriebsmittels, nämlich dem von P_h und P_l gemeinsam benutzen Laufzeitstapel.

SRTF Abkürzung für (en.) *shortest remaining time first*. Bezeichnung für eine Strategie zur ↑Vorrangplanung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor. Dabei werden die Aufgaben nach zunehmender verbleibender ↑Bedienzeit sortiert. Die Aufgabe mit der kürzesten ↑Restlaufzeit erhält den Vorzug bei der Prozessorzuteilung: je kürzer die verbleibende Bearbeitungsdauer einer Aufgabe durch einen ↑Prozess, desto höher seine ↑Priorität, wobei der Prioritätswert einen Zeitwert darstellt. Im Grunde ein Verfahren, das alle Eigenschaften von ↑SPN hat, im Kern allerdings die ↑präemptive Planung verfolgt.
Dem Verfahren liegt die Annahme zugrunde, dass Prozesse mit eher kürzeren ↑Rechenstoßlängen auf stärkere Interaktivität gerade auch in Hinblick auf die ↑Peripherie hindeuten (↑I/O *bound*). Eine Bevorzugung dieser Prozesse könnte die Frequenz, mit der ↑Ein-/Ausgabe geschieht, erhöhen und kann dadurch zu einer gemeinhin höheren Auslastung der ↑Peripheriegeräte beitragen. Im Falle von ↑Dialogstationen bedeutet dies insbesondere eine Stärkung der Interaktion zwischen Mensch und ↑Rechensystem. Damit teilt das Verfahren die mit ↑VRR verfolgte Vorgehensweise, basiert dabei jedoch auf *analytische Methoden* zur Voraussage der zu erwartenden Rechenstoßlängen und damit Charakterisierung von Prozessen.
Im Moment der Bereitstellung eines Prozesses, P_{rdy}, wird die für ihn (gemäß SPN) abgeschätzte nächste ↑Stoßzeit, t_{job}, der zu erwartenden Restlaufzeit, t_{odd} des gegenwärtig stattfindenden Prozesses, P_{run}, gegenübergestellt. Wenn dann $t_{job} \geq t_{odd}$ gilt, wird P_{rdy} auf die ↑Bereitliste gesetzt und P_{run} fährt fort. Anderenfalls ($t_{job} < t_{odd}$) kommt P_{run} auf die Bereitliste und P_{rdy} erhält den Prozessor zugeteilt: P_{run} wird dem Prozessor zugunsten von P_{rdy} entzogen (↑*preemption*).

Genau genommen ist dieser Ablauf in der Form jedoch nur bei hundertprozentigem ↑Verdrängungsgrad möglich, wenn nämlich der Prozessorentzug wirklich jederzeit und damit zwischen der Ausführung von zwei beliebigen ↑Maschinenbefehlen in P_{run} geschehen kann. Für das ↑Betriebssystem grundsätzlich unproblematisch ist aber nur der Fall, wenn im Moment der Bereitstellung von P_{rdy} der zu verdrängende Prozess (P_{run}) auf ↑Benutzerebene stattfindet und letzterer erst mit der vorausgegangenen ↑Unterbrechung auf die ↑Systemebene gezwungen wurde. In dieser Situation war das Betriebssystem noch nicht aktiv, sondern es nimmt Aktivität erst auf mit der Unterbrechung von P_{run}, in dessen Namen sodann auf Systemebene die Bereitstellung von P_{rdy} geschieht. Damit ist mit diesen Ersteintritt ins Betriebssystem keine ↑Wettlaufsituation zu befürchten.
Problematisch ist allerdings der Fall, wenn P_{run} seine Unterbrechung auf Systemebene erfährt, also das Betriebssystem bereits aktiv ist (z.B. als Folge eines ↑Systemaufrufs oder einer Unterbrechung) und es zum ↑Wiedereintritt kommt. Unabhängig von den dann für gewöhnlich notwendigen Maßnahmen zur ↑Nebenläufigkeitssteuerung, um möglichem Fehlverhalten im Betriebssystem vorzubeugen, ist in dem Fall ein hundertprozentiger Verdrängungsgrad auf Systemebene in der Praxis grundsätzlich nicht erreichbar. Setzt die Nebenläufigkeitssteuerung auf ↑blockierende Synchronisation, erscheint diese Feststellung auf den ersten Blick zunächst sehr plausibel. Jedoch spielt in dem Zusammenhang schon die Art der damit einhergehenden ↑Prozesssperre eine große Rolle. Schließlich bildet nur eine ↑Handlungssperre die Voraussetzung, um einen Prozess vor dem möglichen Entzug seines Prozessors wirklich bewahren zu können. Wird dagegen ↑wechselseitiger Ausschluss betrieben und benutzt dieser eine ↑Umlaufsperre, wird die Verdrängung des sperrenden Prozesses (↑*lock holder preemption*) nur dann unterbunden, wenn aus Gründen der ↑Performanz zugleich noch eine ↑Unterbrechungssperre zur Wirkung kommt. Findet anstatt Umlaufsperre ein ↑binärer Semaphor oder ↑Mutex Verwendung, ist Verdrängbarkeit nur dann gegeben, wenn die betreffenden Abstraktionen ohne ↑Verdrängungssperre implementiert sind.
Aber auch ↑nichtblockierende Synchronisation impliziert längst keine hundertprozentige, *volle Verdrängbarkeit* im Betriebssystem. Eine nur eingeschränkt mögliche Verdrängbarkeit kommt immer dann zum Vorschein, wenn der gegenwärtig auf dem Prozessor stattfindende Prozess seinen Prozessor abgeben wird, weil ein anderer Prozess ihn verdrängt, der abgebende Prozess dazu bereits seinen Nachfolgeprozess, der nicht (mehr) auf der Bereitliste steht, „in der Hand“ hat und unaufhaltsam auf den ↑Prozesswechsel hin zusteuert.
Angenommen der stattfindende Prozess P_{run} hat die Entscheidung getroffen, einen bereitgestellten Prozess P^*_{rdy} zu aktivieren. Dabei sei P^*_{rdy} repräsentiert als *lokale* ↑ *Variable* oder ↑tatsächlicher Parameter des P_{run} bestimmenden ↑Unterprogramms mit eben dieser Entscheidung zum Prozesswechsel. Zwischen der Entscheidung zum Prozesswechsel und dem Prozesswechsel selbst wird P_{run} nun unterbrochen und soll durch einen Prozess, P^{**}_{rdy}, vom Prozessor verdrängt werden, da die für P^{**}_{rdy} erwartete Stoßzeit, t^{**}_{job}, kürzer ist als die Restlaufzeit von P_{run}: $t^{**}_{job} < t_{odd}$. Ob jedoch diese Stoßzeit von P^{**}_{rdy} auch kleiner ist als die erwartete Stoßzeit, t^*_{job}, von P^*_{rdy}, der sich in Hand von P_{run} befindet und sodann implizit mit P_{run} den Prozessor abgeben wird, lässt sich nicht bestimmen. Sollte nämlich $t^*_{job} < t^{**}_{job}$ gelten, müsste P^*_{rdy} den Vorzug gegenüber P^{**}_{rdy} erhalten und daher dürfte P_{run} nicht durch P^{**}_{rdy} vom Prozessor verdrängt werden. Da in der gegebenen Situation aber P_{run} wegen $t^{**}_{job} < t_{odd}$ die Kontrolle an P^{**}_{rdy} abgeben würde, kann dies zu einer Fehlentscheidung gegenüber P^*_{rdy} führen. Diese Fehlentscheidung entspricht einer ↑Prioritätsverletzung, der unbedingt vorzubeugen ist, um die Durchsetzung der Strategie zur Vorrangplanung zu erreichen. Die Lösung dieses Problems bedeutet zweierlei:

1. P_{run} nicht in der kritischen Phase zwischen der Entscheidung zum Prozesswechsel an P^*_{rdy} und dem Prozesswechsel selbst zu verdrängen, stattdessen aber P^{**}_{rdy} auf die Bereitliste setzen und
2. P^*_{rdy} die ↑Prozessverpflichtung auferlegen, direkt nach dem Prozesswechsel auf die Richtigkeit der von P_{run} getroffenen Entscheidung hin zu prüfen und gegebenenfalls unverzüglich die Kontrolle des Prozessors an P^{**}_{rdy} abzugeben.

Damit wird die eventuelle ↑Einlastung von P^{**}_{rdy} zunächst aufgeschoben, bis P^{*}_{rdy} den Prozessor erhalten hat und die aufgeschobene Einlastung sofort nachholt.
Auf den ersten Blick scheint es daher naheliegend zu sein, dass P_{run} zu Beginn der kritischen Phase eine Verdrängungssperre erhebt. Allerdings darf diese Sperre erst nach erfolgtem Prozesswechsel von P^{*}_{rdy} wieder aufgehoben werden. Demzufolge wäre in diesem Fall die P^{*}_{rdy} auferlegte Prozessverpflichtung sogar noch zu erweitern, nämlich die Verdrängungssperre aufzuheben. Denn trotz Verdrängungssperre bleibt P_{run} immer noch unterbrechbar, somit können sehr wohl Verdrängungsanforderungen durchkommen. Der in jeder dieser Anforderungen jeweils adressierte Prozess ist sodann wie P^{**}_{rdy} zu behandeln und, um nicht verloren zu gehen, auf die Bereitliste zu setzen. Trotz Verdrängungssperre ist daher nach wie vor die Bereitliste im Rahmen der Prozessverpflichtung von P^{*}_{rdy} zu überprüfen. Die Überprüfung kann nur dann entfallen, wenn eine Unterbrechungssperre genutzt wird. Aber auch dann ist die Prozessverpflichtung für P^{*}_{rdy} unverzichtbar, da auch diese Sperre immer noch von P_{run} erhoben und von P^{*}_{rdy} aufgehoben werden müsste.
Die Konsequenz daraus aber ist, dass das Betriebsystem nunmehr einen ↑Verdrängungspunkt auf Systemebene definieren muss. Solange dieser Punk von P_{run} beziehungsweise P^{*}_{rdy} noch nicht erreicht ist, kann der betreffende Prozess (P_{run} oder P^{*}_{rdy}), wenn er nämlich vom Prozessor zu verdrängen ist, nicht zurück auf die Bereitliste gesetzt (↑*scheduling latency*) und der Prozessor mit P^{**}_{rdy} auch nicht eingelastet (↑*dispatching latency*) werden.
Mit dem dargestellten Problem ist in der Praxis grundsätzlich jede Form von (a) präemptiver und (b) prioritätsbasierter Planung konfrontiert. Nur präemptive Planung allein, wenn also sonst alle Prozesse als gleichberechtigte ↑Entitäten betrachtet werden, ruft dieses Problem nicht hervor und muss daher auch keine Gegenmaßnahmen treffen. Im Falle einer solchen Kombination ist für jede Operation, die (1) eine Prozessauswahl und (2) einen anschließenden Prozesswechsel als ↑Aktionsfolge enthält, dieses Problem virulent. Typisches Beispiel einer solchen Operationen ist die Bereitstellung (`ready`) eines Prozesses.

Exkurs Für die hier betrachtete Klasse von Verfahren (präemptiv und prioritätsbasiert) bildet die Operation, einen Prozess in Bereitschaft (`ready`) zu setzen, die einzige Aktionsfolge im Betriebssystem, die Verdrängung eines Prozesses zu bewirken. Grundsätzlich wird die Bereitstellungsoperation, wie auch in anderen Verfahren (↑FCFS, ↑RR, ↑VRR, SPN, ↑HRRN), indirekt von Prozessen bewirkt, um aufgrund eines eingetretenen ↑Ereignises einen auf den Ereigniseintritt passiv wartenden Prozess zu deblockieren. Dieser Aufruf kann sowohl aus einem synchron (↑*trap*: ↑Systemfunktion) als auch aus einem asynchron (↑*interrupt*: ↑Unterbrechungshandhaber) stattfindenden ↑Handlungsstrang heraus geschehen. Jedoch in keinen dieser (eben genannten) Verfahren wird im Moment der Ausführung dieser Operation dem im ↑Prozesszustand laufend befindlichen Vorgang der Prozessor entzogen — auch im Falle von RR oder VRR, die beide Prozessorentzug beim Ablauf der ↑Zeitscheibe bewirken, bedeutet „in Bereitschaft setzen“ eines Prozesses lediglich ↑Einplanung, nämlich die Einreihung des zugehörigen ↑Prozesskontrollblocks in die Bereitliste, nicht aber zugleich auch Einlastung, also die Verdrängung des laufenden Vorgangs durch direkt folgendem ↑Prozesswechsel.
Der oben thematisierte Aspekt zur ↑Koordination der durch das potentiell verdrängend wirkende Verfahren möglichen gleichzeitigen Vorgänge, mit der die ↑mitlaufende Planung nunmehr konfrontiert ist, betrifft zuallererst die durch die Bereitstellungsoperation festgelegten ↑Aktionen. Hierzu sind insbesondere drei Punkte zu beachten:

- Ein Prozess wird nicht mehrfach bereitgestellt. Zwischen den einzelnen Bereitstellungen ein und desselben Prozesses wurde diesem nicht nur der Prozessor zugeteilt, sondern der Prozessor wurde mit diesem Prozess auch eingelastet (↑*dispatching*: `seize`, S. 315). Im gegebenen Fall musste der Prozess, damit er bereitgestellt werden kann, im ↑Prozesszustand blockiert gewesen sein. Demzufolge finden die auf einen Prozesskontrollblock bezogenen Aktionen implizit in (logischer) Isolation voneinander statt, sie unterliegen damit nicht der Nebenläufigkeitssteuerung.
- Gleichwohl können aber mehrere, verschiedene Prozesse zugleich bereitgestellt werden.

Die auf die ↑Bereitliste (↑*shared data*) bezogenen Aktionen finden gekoppelt, das heißt, nicht in (logischer) Isolation voneinander statt. Ähnliches gilt auch in Bezug auf den Prozesswechsel (`seize`, S. 315). Diese Aktionen unterliegen daher der Nebenläufigkeitssteuerung, die jedoch unabhängig voneinander und in den betreffenden Basisoperationen eigenständig bewerkstelligt werden kann.

- Ein und derselbe wegschaltende Prozess kann zum Wiedereintritt in die Bereitstellungsoperation gezwungen werden und dadurch indirekt rekursiv gegebenenfalls abermals seine eigene Verdrängung vom Prozessor veranlassen wollen. Dies ist möglich mit jeder ↑Unterbrechungsanforderung, deren Behandlung einen Prozess deblockiert, dessen voraussichtliche Rechenstoßlänge kürzer ist als die Rechenstoßrestlänge des unterbrochenen Prozesses. Die ↑Rekursion durch die ausgelösten Verdrängungsanforderungen ist mit dem beim Ersteintritt bereits in Gang gesetzten ursprünglichen Verdrängungsvorgang zu koordinieren, indem der dann aufgenommene Prozess in die Pflicht genommen wird, einem eventuell (rekursiv) aufgelaufenen kürzeren Prozess nachträglich zu bevorzugen.

Abgesehen von möglichen rekursiven Verdrängungsvorgängen, ist auf der hier zunächst betrachteten ↑Abstraktionsebene sonst nicht für ↑Synchronisation zu sorgen. Eine umfassende und mehrere Prozessoren betreffende Synchronisation geschieht entweder auf einer höheren Ebene, von der aus die Bereitstellungsoperation zur Ausführung gebracht wird. Im dem Fall kommt für gewöhnlich blockierende Synchronisation zum Einsatz, durch die ein ↑kritischer Abschnitt, der die Bereitstellungsoperation enthält, abgesichert wird. Oder die Synchronisation geschieht innerhalb der eventuell kritischen Basisoperationen, nämlich zur Einreihung eines Prozesses in die Bereitliste einerseits und zum Prozesswechsel andererseits. In dem Fall kann auch nichtblockierende Synchronisation zum Einsatz kommen.
Dagegen ist der Umgang mit dem möglichen Wiedereintritt in die Bereitstellungsoperation durch ein und demselben Prozess schon etwas knifflig, sofern dieser nicht durch eine Unterbrechungssperre verhindert werden soll. Eine solche Sperre trägt insbesondere hier nicht nur zu einer erheblichen Erhöhung der ↑Unterbrechungslatenz bei, sondern wird aufgrund der davon betroffenen Aktionsfolge zur Einsortierung eines Prozesses in die Bereitliste unbestimmt lang erhoben bleiben. Daher scheidet eine solche Maßnahme zur Abwehr des möglichen Wiedereintritts in die Bereitstellungsoperation und daraus folgender rekursiver Verdrängungen für gewöhnlich aus. Stattdessen bietet sich für eine derartige Situation, in der nämlich ein Prozess unaufhaltsam seinen Prozessor zugunsten eines anderen Prozesses abgeben wird, ein *Hilfsschema* an. Dieses setzt auf eine Prozessverpflichtung für den dann den Prozessor erhaltenen Prozess, nämlich den Prozessor gegebenenfalls sofort wieder abzugeben.
Die hier zunächst betrachtete Operation (`ready`) berücksichtigt nun, dass der Prozess, unter dessen Namen ein anderer Prozess direkt (Systemaufruf) oder indirekt (Unterbrechungshandhaber) in Bereitschaft gesetzt wird, möglicherweise auch den Prozessor abgeben muss. Eine Skizze dazu zeigt nachfolgendes Beispiel (vgl. S. 317):

```
void ready(process_t *task) {                /* schedule according to SRTF */
    process_t *self = being(ONESELF);            /* own process descriptor */
    assert((task != self) || valid(&self->mood, IDLE));
    if (task != self)                                        /* not myself? */
        task->time.guess = guess(task);      /* yes, estimate burst length */
    time_t left = spare(&self->time, timestamp());  /* set remaining term */
    if ((left <= task->time.guess) || valid(&self->mood, PENDING | RUNNING))
        allow(task, READY, task->time.guess);              /* normal ready */
    else {                                              /* too long, preempt */
        allow(self, PENDING | READY, left);                /* touchy ready */
        seize(task);                             /* perform process switch */
        watch();                              /* fulfil process obligation */
    }
}
```

Im Vergleich zu SPN, die nicht verdrängend arbeitende Variante des Verfahrens, bildet die Fallunterscheidung in Bezug auf die Restlänge des Rechenstoßes des die Operation durchführenden (d.h., laufenden) Prozesses das wesentliche Unterscheidungsmerkmal — außerdem ist der indirekt rekursiven Verdrängung vorzubeugen, falls nämlich der Prozesszustand des laufenden Vorgangs als schwebend laufend erkannt wurde (`valid`). Die Berechnung der Restlänge des Rechenstoßes gestaltet sich dabei wie folgt:

```
time_t spare(burst_t *this, time_t time) {
    return burst(this) - spent(this, time);          /* net remaining term */
}
```

Für den gegebenen Fall wird also der Nettobetrag (`spent`, S. 314) der seit Aufnahme des laufenden Prozesses (`this`) verstrichenen Zeit gebildet (s. auch das zugehörige `seize`, S. 315). Dieser Wert entspricht der einem Prozess voraussichtlich noch zur Verfügung stehenden Zeit (`spare`) bis zum Ende seines Rechenstoßes. Ist die Restlänge nicht größer als die erwartete Rechenstoßlänge des in Bereitschaft zu setzenden Prozesses, wird wie bei SPN verfahren: der betreffende Prozess (`task`) kommt auf die Bereitliste. Anderenfalls (sonst-Zweig) wird der laufende Prozess auf die Bereitliste gesetzt, vom Prozessor verdrängt (`seize`) und kommt bei Wiederaufnahme seiner Prozessverpflichtung nach (`watch`). Die in beiden Fällen durchgeführte Bereitstellung eines Prozesses, verläuft wie folgt:

```
void allow(process_t *task, mood_t mood, time_t left) {
    state(&task->mood, mood);                       /* define ready state */
    ticket(&task->line, left);                         /* define sort key */
    enlist(labor(), &task->line);                         /* sort process */
}
```

Die Bereitstellungsoperation (`allow`) kommt auch bei Vermeidung einer indirekt rekursiven Verdrängung zur Geltung. Sollte also der bereitzustellende Prozess (`task`) einen Vorranganspruch haben, weil seine Rechenstoßlänge kürzer ist als die Restlänge des stattfindenden Prozesses, dann kommt ersterer dennoch auf die Bereitliste, und zwar wenn der Zustand letzteren Prozesses eben darauf hindeutet (`valid`), dass dieser sich bereits auf dem Wege zur Prozessorabgabe befindet.

Zu beachten ist der Zwischenzustand (`PENDING`), in den der vom Prozessor zu verdrängende Prozess gebracht wird. Der zu Beginn der Verdrängungsphase (`ready`, sonst-Zweig) noch im Zustand laufend agierende, demnächst jedoch den Prozessor abgebende Prozess kommt vor dem Prozesswechsel auf die Bereitliste. Sobald ein Prozess aber auf der Bereitliste steht, kann er zur Einlastung des Prozessors herangezogen werden. Im gegebenen Fall besteht diese Möglichkeit allerdings nur bei echter (d.h., durch Vervielfachung einer Recheneinheit ermöglichte) ↑Parallelverarbeitung, nämlich wenn ein anderer Prozessor, auf den sich der anstehende Prozesswechsel nicht bezieht, eingelastet werden soll. Das hier betrachtete Ver-

fahren arbeitet zwar verdrängend, nicht jedoch durch ein ↑Zeitteilverfahren. Daher kann der laufende Prozess zwischen Listeneintragung (`allow`: `enlist`) und Prozesswechsel (`ready`: `seize`) nicht für seinen Prozessor zur Einlastung ausgewählt werden: für diesen Prozess ist keine ↑Zeitscheibe definiert, er gibt den Prozessor nur nach Vollendung der ihm aufgetragenen Aufgabe ab (↑*run to completion*). Die Bereitstellung wird immer einen Prozess auf die Bereitliste setzen, nämlich entweder den „von außen“ vorgegebenen oder den laufenden Prozess, niemals aber einen Prozess davon entfernen.
Von einen anderen Prozessor ausgehend wäre es jedoch grundsätzlich möglich, den soeben auf die Bereitliste platzierten Prozess zur Einlastung auszuwählen und daher auch von dieser Liste zu entfernen. Diese Auswahl geschieht immer dann, wenn ein Prozess für seinen weiteren Fortschritt erst noch ein bestimmtes ↑Ereignis erwarten und dazu seinen ↑Rechenstoß von selbst beenden muss, der betreffende Prozess also selbst den Nachfolger für seinen Prozessor festlegt (`block`, s.u.). Um nun zu verhindern, dass in solch einer Situation, ein bereits bereitgestellter, aber noch laufender Prozess fälschlicherweise zugleich auf einen anderen Prozessor wieder aufgenommen wird, nimmt der zu verdrängende Prozess den Zwischenzustand (`PENDING`) an. Ein Prozess, der sich in diesem Zwischenzustand befindet, wird bei einer möglicherweise überlappend stattfindenden Auswahl bewusst übergangen.
Dieser Zwischenzustand verhindert gleichsam rekursive Verdrängungsanforderungen. Hat ein Prozess den Verdrängungsvorgang bereits eingeleitet, geht er diesen konsequent weiter und sorgt aber dafür, dass während dieses Vorgangs bereitgestellte Prozesse höherer Priorität als er selbst nicht verloren gehen (`ready`, dann-Zweig). Trotz Vorrang gegenüber dem stattfindenden Prozess darf allerdings nicht davon ausgegangen werden, dass dieser Anspruch auch in Bezug auf den ganz oben auf der Bereitliste stehenden Prozess ebenso gilt. Der die Bereitliste anführende Prozess kann sehr wohl die höchste Priorität aller Prozesse haben, den stattfindenden Prozess eingeschlossen, falls er selbst zuvor aus Gründen der Abwehr einer indirekt rekursiven Verdrängung zurückgestellt werden musste. Daher darf bei der Zurückstellung eines eingetroffenen Prozesses, dessen Rechenstoßlänge kürzer als die Rechenstoßrestlänge des stattfindenden Prozesses ist, ersterer nicht etwa automatisch an den Anfang der Bereitliste gesetzt und der Einfachheit halber nach ↑LCFS behandelt werden. Vielmehr muss dieser Prozess auch in der Situation eine Listenposition erhalten, die seinem Rank relativ zu allen anderen bereits gelisteten Prozessen entspricht: der Prozess ist mittels *Suchlauf* einzusortieren (`enlist`).
Der Verdrängungsvorgang endet mit erfolgter Übergabe des Prozessors an den beim Ersteintritt in die Bereitstellungsoperation (`ready`) angegebenen Prozess (`task`), der daraufhin seiner Prozessverpflichtung nachkommt und die Richtigkeit seines Fortgangs überprüft. Nach erfolgter Prozessorübergabe fährt dieser Prozess an der Stelle fort, an der er zuvor selbst die Kontrolle über den Prozessor abgegeben hat. Diese Stelle befindet sich einerseits in der Bereitstellungsoperation (Verdrängungsphase), aber andererseits auch in der Blockierungsoperation (`block`, s.u.). Jedoch in beiden Fällen bedeutet diese Prozessverpflichtung, die erwartete Rechenstoßlänge des oben auf der Bereitliste stehenden Prozesses der Rechenstoßrestlänge des stattfindenden, gerade erst angelaufenen Prozesse gegenüberzustellen und gegebenenfalls die sofortige Prozessorabgabe in Erwägung zu ziehen:

```
void watch() {                                  /* fulfil process obligation */
    process_t *task, *self = being(ONESELF);          /* process pointers */

    do {                        /* check ready list and, if need be, preempt */
        state(&self->mood, PENDING);              /* don't preempt myself */
        if ((task = elect(labor()))) {    /* urgent ready-to-run process? */
            if (task->time.guess >= self->time.guess) {    /* less urgent? */
                enlist(labor(), &task->line);     /* yes, put process back */
                break;                                  /* leave loopback */
            }
            allow(self, READY, spare(&self->time, timestamp()); /* touchy */
            seize(task);                          /* perform process switch */
        }
    } while (task);

    state(&self->mood, CLEAR | PENDING);       /* may be preempted, again */
}
```

Der soeben wieder aufgenommene Prozess (`self`) fährt nur dann fort, wenn er tatsächlich die höchste Priorität (d.h., den kürzesten Rechenstoß) besitzt, das heißt, während des soeben durchgeführten Vorgangs der Prozessorübergabe kein höher priorisierter Prozess (`task`) ankam. Allerdings könnte während dieses Vorgangs, und zwar in der kritischen Phase, sehr wohl mehr als ein Prozess höherer Priorität (d.h., mit kürzeren Rechenstoßlägen) eintreffen. Daher ist eine Schleife zu durchlaufen, um eventuelle Vorranganspruche weiterer auf der Bereitliste stehenden Prozesse zu prüfen. Dazu wird der am Listenanfang stehende Prozess zunächst von der Liste genommen (`elect`), um ihn isoliert von gleichzeitigen Vorgängen (desselben oder eines anderen Prozessors) derselben Art prüfen und behandeln zu können. Sollte für den Prozess kein Vorranganspruch bestehen, kommt er zurück auf die Liste (`enlist`) und die Schleife endet (`break`). Anderenfalls wird der laufende Prozess entsprechend seiner Rechenstoßrestlänge (`spare`) bereitgestellt (`allow`) und mit einem Prozesswechsel (`seize`) vom Prozessor verdrängt.

Die der Bereitstellung entgegengesetzte Operation betrifft das Abblocken eines gegenwärtig auf dem Prozessor stattfindenden Prozesses. Entweder ist der Prozess mit seiner Aufgabe vollständig fertig geworden oder er muss aussetzen, weil er für sein weiteres Voranschreiten vom Eintritt eines bestimmten Ereignisses abhängig ist und dieses Ereignis noch nicht stattgefunden hat. In jedem Fall findet der Rechenstoß des Prozesses ein logisches Ende:

```
void block() {                        /* release and reschedule processor */
    process_t *next, *self = being(ONESELF);

    state(&self->mood, BLOCKED);                    /* mark myself blocked */

    next = quest(labor());          /* search for next process to execute */
    if (next == self)                         /* myself already unblocked? */
        state(&self->mood, FLUSH | RUNNING);              /* yes, continue */
    else {                                        /* no, release processor */
        seize(next);                            /* switch to next process */

        watch();                              /* fulfil process obligation */
    }
}
```

Im Grunde sind, wie auch in den anderen Verfahren, nur zwei zentrale Schritte zu bewältigen, erstens den Nachfolgeprozess suchen (`quest`) und zweitens diesem den Prozessor ergreifen (`seize`) lassen. Im Gegensatz zur Einplanung, die eine Einsortierung von Prozessen in Bezug auf den für sie zu erwartenden Rechenstoßlängen durchführt, ist die Auswahl zur Einlastung des Prozessors mit dem Nachfolgeprozess sehr einfach, sie geschieht gemäß FCFS vom Anfang der Bereitliste ausgehend (*first served*, FS). Hatte der Nachfolgeprozess durch Abblocken (`block`) den Prozessor abgegeben, dann kehrt er hier aus der Prozesswechsel-

operation (`seize`) zurück. Wurde der Nachfolgeprozess durch Bereitstellen eines anderen Prozesses verdrängt, dann kehrt er im Kontext der entsprechenden Bereitstellungsoperation (`ready`) aus der dortigen Prozesswechseloperation zurück. In beiden Fällen muss aber nach vollzogenem Prozesswechsel (d.h., Rückkehr aus `seize`) der betreffende Prozess seiner Verpflichtung nachkommen (`watch`) und auf Richtigkeit der Aufnahme seiner Tätigkeit hin prüfen.
Das Abblocken eines Prozesses ist keine Operation, die die Verdrängung eines Prozesses hervorrufen kann. Obwohl der laufende Prozess seinen Prozessor abgibt, geschieht dies nicht etwa aufgrund „höherer Gewalt", nämlich indem ein Prozess höherer Priorität den Prozessor zugeteilt bekommen muss. Vielmehr gibt der abblockende Prozess aufgrund seines eigenen logischen Verlaufs immer freiwillig den Prozessor ab. Der gefundene Nachfolgeprozess kann daher auch keine höhere Priorität (d.h., kürzere Rechenstoßlänge) als der laufende, sich abblockende Prozess besitzen, sonst wäre letzterer nämlich bei der Bereitstellung (`ready`) seines Nachfolgers schon längst vom Prozessor verdrängt worden. Damit ergibt sich hier eine grundsätzlich andere Situation als für die Bereitstellungsoperation, nur diese ist mit dem Problem einer möglicherweise indirekt rekursiv ausgelösten Prozessverdrängung konfrontiert: die Abblockungsoperation (`block`) verläuft *voll verdrängbar*.

SSD Abkürzung für (en.) *solid state disk/drive*, (dt.) Festkörperlaufwerk, Halbleiterlaufwerk. Die Bezeichnung als Platte/Laufwerk ist sinnbildlich zu sehen: das Gerät enthält keine beweglichen Teile (wie etwa rotierende Scheiben oder ein- und ausfahrende Lese-/Schreibköpfe); es handelt sich weder um eine Platte, noch um ein Laufwerk für eine Platte.
Der Ausdruck begründet sich in der Ablösung herkömmlicher ↑Plattenspeicher durch ↑Festspeicher in Form von ↑Halbleiterplättchen sowie der damit verbundenen Nutzung als ↑Ablagespeicher. Technische Grundlage ist ↑Flashspeicher, der ursprünglich/gemeinhin, wie Plattenspeicher, nur in Blöcken (als Sektoren bezeichnet) und nicht in einzeln adressierbaren ↑Speicherzellen beschrieben werden kann.

Stack (en.) ↑*stack*. Bezeichnung eines ↑Datentyps für eine Sammlung von ↑Daten, auf die zwei grundlegende Operationen definiert sind: *push* für (dt.) vordrängen, um ein Datum oben auf dem Stoß abzulegen und *pull* oder *pop* für (dt.) herausholen, um ein Datum oben von dem Stoß zu entfernen. Eine spezielle Art von dynamischer Datenstruktur, bei der die Elemente umgekehrt zur Einreihungsfolge entfernt werden (↑LIFO).

Exkurs Angenommen, die Grundlage für diese dynamische Datenstruktur sei eine ↑verkettete Liste: `typedef chain_t stack_t`. Dann ergeben sich für die oben genannten Basisoperationen folgende einfache Implementierungen, zunächst die Einfügreoperation:

```
inline void push(stack_t *this, chain_t *item) {
    join(this, item);                        /* add item to top of stack */
}
```

Wie auf S. 375 ersichtlich, hängt `join()` das Kettenglied `item` an das Kettenglied `this`, das zugleich Kopfzeiger der Liste ist, an und lässt diesen abschließend auf `item` verweisen. Analog die Entnahmeoperation:

```
inline chain_t *pull(stack_t *this) {
    return skip(this);                      /* pop item from top of stack */
}
```

Hier wird das am Kopf stehende Kettenglied, so die Liste nicht leer ist, einfach übersprungen, das heißt, aus der Liste entfernt und ein ↑Zeiger darauf als Ergebnis geliefert. Entsprechend wurde der Kopfzeiger auf den Nachfolger des entfernten Kettenglieds oder `NULL` gesetzt.

stack (dt.) ↑Stack, ↑Stapel, Stoß.

stack memory (dt.) ↑Stapelspeicher.

stack pointer (dt.) ↑Stapelzeiger.

stack segment (dt.) ↑Stapelsegment.

Stammdateisystem (en.) ↑*root file system*. Bezeichnung für ein ↑Dateisystem, das auf dem ↑Datenträger liegt, von dem das ↑Betriebssystem urgeladen wird (↑*bootstrap*).

Stapel (en.) ↑*batch*. Stoß übereingelegter gleicher Dinge (in Anlehnung an den Duden). Dem Ursprung nach ist jedes dieser Dinge eine ↑Lochkarte, wobei ein bestimmter Satz davon einen ↑Auftrag beschreibt. Dabei bildet dieser Satz durch übereinanderstapeln von Lochkarten eine wohldefinierte Folge von Anweisungen an ein ↑Rechensystem. Diese Anweisungen sind in der Reihenfolge, wie sie gestapelt und dadurch zusammengestellt wurden (d.h., von unten nach oben bzw. von vorne nach hinten) zu verarbeiten und nicht etwa in der Reihenfolge, wie gestapelte Dinge von einem solchen Stoß (↑*stack*) wieder entfernt werden.
Mittlerweile ist die Lochkarte jedoch weniger das Instrument, um auch eine bestimmte Sequenz solcher Anweisungen zu definieren, sondern stellvertretend nur noch der Datenträger (↑Speichermedium) für ein ↑Kommando. So lässt sich ein solcher Anweisungsstoß eben auch als ↑Datei speichern, wobei die einzelnen darin enthaltenen Kommandos dann unverändert weiterhin ihren Zweck erfüllen. Für gewöhnlich ist die Formulierung für einen ↑Stapelauftrag in genau dieser Form heute (2025) üblich, indem nämlich das in ↑Kommandosprache formulierte ↑Programm als Datei der ↑Stapelverarbeitung zugeführt wird. Ein modernes Beispiel dafür liefert ein für einen bestimmten ↑Kommandointerpreter vorgesehenes ↑*shell*-Skript.

Stapelauftrag (en.) ↑*batch job*. Aufeinanderfolge von Schritten in einem ↑Stapelprozess, wobei jeder Schritt für gewöhnlich durch ein ↑Programm implementiert ist. Ein ↑Auftrag zur Ausführung einer Reihe von Programmen auf einen bestimmten Satz (↑*batch*) von Eingaben von ↑Daten, anstatt auf nur einer einzelnen Eingabe.

Stapelbetrieb (en.) ↑*batch mode*. ↑Betriebsart zur ↑Stapelverarbeitung.

Stapelmonitor (en.) ↑*batch monitor*. Steuerprogramm zur ↑Stapelverarbeitung. Das ↑Programm führt jeden eingehenden ↑Stapelauftrag dem ↑Kommandointerpreter zu, überwacht die Ausführung von jeden einzelnen in dem ↑Stapel beschriebenen ↑Auftrag und erstellt suksessive das ↑Konsolenprotokoll. Die den ↑Stapelbetrieb bestimmende Funktionseinheit in einem ↑Betriebssystem, das speziell für diese ↑Betriebsart ausgelegt ist.

Stapelprozess (en.) ↑*batch process*. Vorgang der ↑Stapelverarbeitung.

Stapelsegment (en.) ↑*stack segment*. Bezeichnung von einem ↑Datensegment, das den ↑Stapelspeicher von einem ↑Prozess bildet.

Stapelspeicher (en.) ↑*stack memory*. ↑Speicher, der als Stapel (dynamische Datenstruktur) organisiert und nach ↑LIFO betrieben wird.

Stapelverarbeitung (en.) ↑*batch processing*. Verfahren zur Durchführung der einem ↑Rechensystem übergebenen Aufträge, nämlich um ein ↑Programm nach dem anderen automatisch zur Ausführung zu bringen und vollständig abzuarbeiten, dazu jeweils die benötigten ↑Daten zur Eingabe bereitzustellen und die Ausgabedaten zwischen- oder endzuspeichern beziehungsweise darzustellen oder auszudrucken. Die vollständig beschriebenen Aufträge sind in einem ↑Stapel hinterlegt, sie werden ohne weitere Interaktion mit der beauftragenden externen ↑Entität von einem ↑Stapelmonitor abgearbeitet. Dem dabei von dem Stapelmonitor erstellten ↑Konsolenprotokoll kann nach erfolgter Abarbeitung des Stapels der Verlauf und das jeweilige Ergebnis der durchgeführten Arbeitsschritte entnommen werden. Das bevorzugte Verarbeitungsverfahren für Routineaufgaben, beispielsweise um am Tagesende Zahlungseingänge einzubuchen (Rechnungswesen), am Wochenende Verkaufsstatistiken zu generieren (Einzelhandel), am Monatsende Gehaltsabrechnungen zu erstellen (Personalwesen), am Quartalsende Daten einer Aktiengesellschaft aufzubereiten (Berichtswesen) und am Jahresende den rechnerischen Abschluss eines kaufmännischen Geschäftsjahres vorzulegen

(Wirtschaftswesen). Aber auch in technischen Bereichen finden sich viele Anwendungsbeispiele, etwa die Digitalisierung von Umkehr- und Negativfilm sowie die „Entwicklung“ der generierten Rohdaten (Fotobearbeitung), die Bildung einer Fertigungsreihenfolge von Produktionsaufträgen (Sequenzierung), das Erstellen von Kopien gespeicherter Informationen (Datensicherung) beziehungsweise die Restauration der Originaldaten nach einem Systemausfall (Datenwiederherstellung) und so weiter — bis hin zur automatischen Fertigung von einem ↑Maschinenprogramm oder ↑Betriebssystem (↑*build management*).

Stapelzeiger (en.) ↑*stack pointer*. ↑Register der ↑CPU, das ihrem Steuerwerk (*control unit*) das gegenwärtige obere Ende in einem ↑Laufzeitstapel (*stack top*) anzeigt: auch ↑SP genannt. Diese Angabe ist eine ↑Adresse im ↑Arbeitsspeicher, an der die ↑Speicherstelle entweder von dem zuletzt gestapelten oder für das nächste zu stapelnde Datum zu finden ist. Mit jeder Stapeloperation wird der Registerinhalt automatisch um die Größe (Vielfaches von einem ↑Byte) des zu stapelnden Datums verändert, und zwar vor oder nach der Operation — was die beiden Möglichkeiten, worauf genau die im Register enthaltene Adresse verweist, begründet. Je nach CPU ist diese Größe fest (typisch für ↑RISC) oder variabel (typisch für ↑CISC), im letzteren Fall aber nur bis zu einer durch die ↑Wortbreite bestimmten oberen Grenze:

$$1 \leq quantity \leq (width(word) + width(byte) - 1))/width(byte),$$

wobei *quantity* eine Byteanzahl repräsentiert und die Funktion *width* eine Bitanzahl liefert. Darüber hinaus kann bei der Adressierung dieser Speicherstelle die Randbedingung gelten, dass der Adresswert im Register ganzzahlig Vielfaches der Größe (Byteanzahl) des dort zu lesenden/schreibenden Operanden sein muss (↑Ausrichtung). In Abhängigkeit von der durch die CPU definierten Expansionsrichtung des Stapels wird der Registerinhalt bei der Datumsablage (*push*) dekrementiert (*top-down stack*) oder inkrementiert (*bottom-up stack*), das heißt, der Stapel wächst entweder von hohen zu niedrigen oder von niedrigen zu hohen Adressen. Die inverse Operation der Datumsentnahme (*pop*) wirkt umgekehrt.

starke Konsistenz (en.) ↑*strong consistency*. Für gewöhnlich ↑strikte Konsistenz, jedoch auch die Bezeichnung für jede Form von ↑Speicherkonsistenz, die stärker ist als ↑schwache Konsistenz.

start time (dt.) ↑Startzeit.

Startblock (en.) ↑*boot block*. Bezeichnung für den ↑Block auf einem ↑Datenträger, in dem der ↑Urlader liegt.

Startzeit (en.) ↑*start time*, s_i. Bezeichnung für eine ↑Prozessgröße, die den Zeitpunkt festlegt, zu dem die Durchführung einer ↑Aufgabe tatsächlich beginnt.

starvation (dt.) ↑Verhungern.

static memory (dt.) ↑statischer Speicher.

static scheduling (dt.) ↑statische Planung.

statische Planung (en.) ↑*static scheduling*, auch ↑vorlaufende Planung, Anders als ↑dynamische Planung stagniert der ↑Ablaufplan während die betreffenden ↑Prozesse stattfinden.

statischer Speicher (en.) ↑*static memory*. Bezeichnung für einen ↑Speicher, dessen räumliches Fassungsvermögen (Kapazität) unveränderlich ist; Gegenteil von ↑dynamischer Speicher. Typisches Beispiel dafür ist das ↑BSS- oder ↑Datensegment von einem (↑UNIX) ↑Prozess. Beide bilden jeweils eine Zusammenfassung der ↑Platzhalter für die verwendeten Programmvariablen, initialisiert (Daten) oder nicht initialisiert (BSS, jedoch zur ↑Ladezeit mit 0 vorbelegt), und dienen zur ↑Laufzeit der Speicherung von Berechnungsergebnissen. Die Größe von diesen Segmenten wird zur ↑Bindezeit festgelegt und verändert sich zur Laufzeit eines Programms nicht mehr.

Statusregister (en.) ↑*condition-code register*. Behältnis in der ↑CPU, in dem eine Sammlung von Zustandsanzeigen (*status-flag bits*) vermerkt ist. Diese Anzeigen werden als Nebeneffekt bei der Ausführung von einem ↑Maschinenbefehl aktualisiert, etwa bei arithmetisch-logischen Operationen (typisch) oder beim Laden von einem ↑Speicherwort in ein ↑Prozessorregister (untypisch, MOS Technology 6502); sie können aber auch explizit durch spezielle Maschinenbefehle gelöscht oder gesetzt werden. Der jeweilige Zustand wird durch ein ↑Markierungsbit in diesem ↑Register dargestellt, wobei folgende Reihe von Bits typisch ist: Übertrag (*carry*), Überlauf (*overflow*), Vorzeichen (*sign*), Null (*zero*) und Parität (*parity*). Auf Basis dieser Zustandsanzeigen lassen sich in ↑Assembliersprache bedingte Anweisungen beziehungsweise Sprünge formulieren und so Fallunterscheidungen und verschiedene Formen von Schleifen in einem Programm umsetzen.

Stauauflösung (en.) ↑*congestion resolution*. Eine durch Behinderung von ↑Prozessen bewirkte Ansammlung von ↑Aufgaben nicht länger bestehen lassen (in Anlehnung an den Duden). Behinderungsanlass ist für gewöhnlich ein ↑wiederverwendbares Betriebsmittel, das zur Bearbeitung einer oder mehrerer Aufgaben für eine bestimmte Zeiteinheit von einer zu hohen Anzahl ↑gleichzeitiger Prozesse beansprucht wird.
Für ein wiederverwendbares, ↑unteilbares Betriebsmittel ist zudem auf algorithmischer Ebene der Zugriff typischerweise als ↑kritischer Abschnitt oder ↑atomare Operation formuliert. Hier kann allein die Notwendigkeit zur ↑Synchronisation ↑gekoppelter Prozesse bei zu hohem Prozessaufkommen zum Stau führen. Aufgelöst wird ein solcher Stau, indem die beteiligten Prozesse gezwungen werden, Abstand von der wiederholten Durchführung der Synchronisatonsoperation zu nehmen.
Jedem der betroffenen Prozesse wird in solch einer ↑Konkurrenzsituation, in der zu viele gleichzeitige Zugriffe auf ein ↑gemeinsames Betriebsmittel geschehen, eine bestimmte ↑Ruhezeit verordnet, und zwar bis er den nächsten Versuch einer wettstreitigen ↑Aktion angeht. Typischer ↑Anwendungsfall einer solchen prozessspezifischen Verzögerung ist die ↑Umlaufsperre. Die Prozessverzögerung führt zur Verringerung der Rate des Zugriffs auf die Sperre (*lock*) und sorgt damit für eine Verkehrsberuhigung auf bestimmten Verbindungswegen oder in relevanten Hardwareeinheiten, um weitere Störungen im ↑Programmablauf zu reduzieren oder vorzubeugen. Diese Rate verringert sich durch programmiertes, eigenständiges, temporäres zurückhalten (↑*backoff*) von der Ausübung weiterer Zugriffe des Prozesses auf das mit anderen Prozesse gemeinsame Betriebsmittel.
Das Verfahren greift immer nur im Falle einer Konkurrenzsituation gekoppelter Prozesse, die das fragliche ↑Betriebsmittel zum jeweils eigenen Fortschritt belegen müssen, dies aber nur strikt nacheinander dürfen (↑*indivisable resource*). Gemeinsames Merkmal des jeweiligen Prozessverhaltens in diesen Situationen ist ↑aktives Warten auf die Freigabe des beanspruchten Betriebsmittels. Neben ↑Rechnernetzen (insb. ↑Ethernet) stehen hier vor allem auch ↑Busse ↑speichergekoppelter Multiprozessoren im Vordergrund. Die Verfahren sorgen dafür, dass wettstreitige Prozesse für eine bestimmte Zeit von selbst von weiteren Buszugriffen Abstand nehmen, dadurch das Prozessaufkommen pro Zeiteinheit reduzieren und effektiv damit eine Auflösung des Staus betreiben.

Exkurs Abstand gewinnen in einer Konkurrenzsituation beruht auf freiwilligen Aktionen von jedem der beteiligten Prozesse. Für jeden einzelnen wettstreitigen Prozess bedeutet es jedoch lediglich, Zeit zu verbrauchen:

```
time_t backoff(time_t time) { /* make an active break, delay own progress */
    dwell(time / sweep());       /* estimate # of delay loops & stay calm */
    return time * 2                /* indicate next backoff delay, if any */
}
```

Hier wird davon ausgegangen, dass die Einheit der Ruhezeit (`time`) des Prozesses der ↑Prozessorzyklus darstellt. Der Prozess selbst verweilt (`dwell`) kraft einer ↑Verzögerungsschleife, indem er nichts tut.
Die gezeigte Rückzugsoperation (`backoff`) verzögert den Prozess, beruhigt dabei auch den

Prozessor in Bezug auf seine Busaktivitäten und liefert als Ergebnis die nächste Ruhezeit für den Prozess zurück, sollte dieser abermals im selben Kontext in eine Konkurrenzsituation geraten: die eingangs übergebene Ruhezeit wird ausgangs verdoppelt (↑*exponential backoff*). So liegt es beispielsweise ganz in der Hand des für eine Umlaufsperre praktizierten Verfahrens, die Ruhezeit als einen statischen oder dynamischen Parameter aufzufassen und im nächsten Versuch entweder den „von oben“ gegebenen ursprünglichen oder „von unten“ gelieferten fortgeschrittenen Wert zu verwenden (vgl. S. 359), in beiden Fällen sind aber von Durchlauf zu Durchlauf immer Wertekorrekturen „von außen“ möglich.
Um nun die Anzahl der benötigte Schleifendurchläufe abzuschätzen, wird der Zeitwert (`time`) für die Ruhephase normalisiert, das heißt, in die benötigte Anzahl von Schleifendurchläufen umgerechnet. Die in der Verzögerungsschleife verwendete Zählanweisung (vgl. S. 380) dominiert den Aufwand eines einzelnen Durchlaufs (*sweep*). Die Anzahl der bei Ausführung dieser Anweisung voraussichtlich anfallenden Prozessorzyklen bestimmt folgende Funktion:

```
inline unsigned sweep() {                /* determine backoff-loop costs */
    return YANETUT;     /* estimated # of processor cycles per loop sweep */
}
```

Im gegebenen Beispiel ist der durch einen ↑Makrobefehl definierte Rückgabewert (`YANETUT`) eine Konstante, die durch (a) Analyse der vom ↑Kompilierer generierten ↑Assemblieranweisungen und (b) Abschätzung der pro Anweisung anfallenden Anzahl von Operationszyklen für den gegebenen Prozessor bestimmt wird: die hier betrachtete Verzögerungsschleife (`dwell`, S. 380)) beansprucht 3 – 5 ↑Maschinenbefehle, je nachdem ob ein Spezialbefehl der ↑CPU zur „Entspannung“ gewisser Vorgänge in der Hardware genutzt werden kann oder nicht. Eine sich als nicht ganz so schlecht erwiesene Abschätzung für die zu erwartende Anzahl von Operationszyklen, und zwar für eine bereits wenige Male durchlaufene Schleife (d.h., die Befehle liegen im ↑Zwischenspeicher und die ↑Verarbeitungskette im Prozessor bricht nicht ab), ergibt allgemein folgender Wert:

```
#define YANETUT sizeof(time_t) /* you are not expected to understand this */
```

Alternativ kann dieser Wert auch durch eine Messung ermittelt werden. In dem Fall ist der Makrobefehl entsprechend anzupassen und beispielsweise als Aufruf einer problemspezifischen Messfunktion zu formulieren.
Der Einfachheit halber nimmt die gezeigte Lösung an, dass systembedingte Verzögerungen die Ruhezeit des Prozesses nur unwesentlich beeinflussen. Solche Verzögerungen können durch ↑Unterbrechungsbehandlungen hervorgerufen werden, insbesondere wenn bei der Schleifenausführung (`dwell`) ↑Unterbrechungsanforderungen eintreffen. Um derartiges zu verhindern oder zu tolerieren, muss der Prozess entweder mit erhobener ↑Unterbrechungssperre ruhen — was allerdings nur eine theoretische Lösung sein kann — oder es ist dafür zu sorgen, dass in der Schleife eventuelle systembedingte Verzögerungszeiten ausgeblendet werden. Letzteres setzt beispielsweise die Akkumulation der durch ↑Unterbrechung eines Prozesses entstehenden Verzögerungszeiten voraus, so dass im Bedarfsfall die Normalisierung der pro Schleifendurchlauf benötigten Zeit möglich ist. Dies würde bedeuten, den Schleifenzähler um einen variablen Wert zu erniedrigen, und zwar um eine Größe, die der Anzahl von Schleifendurchläufen bezogen auf die aktuelle Verzögerung des Prozesses in der Schleife entspricht.
Ein anderer Ansatz zur Überprüfung des Ablaufs der Ruhezeit besteht darin, auf einen ↑Zeitgeber zurückzugreifen. Dieses autonom zum Prozessor tätige Gerät wird dann direkt vor der Warteschleife (a) mit der Dauer der Ruhezeit programmiert und (b) in den ↑Aufrufbetrieb versetzt, um dann durch ↑aktives Warten (`probe`, S. 111) des Prozesses den Ablauf des Zeitintervalls abzupassen. Eine diesbezügliche Anpassung zeigt folgende Skizze:

```
time_t backoff(time_t time) {    /* make an active break: real-time clock */
    time_t *hand = (time_t *)timer(time, POLLING_MODE, MEMORY_MAPPED);
    probe(hand, 0)                        /* busy wait on timer expiration */
    return time * 2                                  /* next backoff delay */
}
```

Dieses Beispiel geht davon aus, dass die Interaktion mit dem Zeitgeber durch ↑speicherabgebildete Ein-/Ausgabe geschieht. Damit läuft die Ruhezeit des Prozesses im Hintergrund zu den sonstigen Aktivitäten seines Prozessors ab und der Prozess beendet seine Ruhephase, wenn ein bestimmtes ↑Speicherwort den Wert null enthält. An der ↑Adresse (`hand`) dieses Speicherworts liegt das in den ↑Adressraum des Prozesses eingeblendete ↑Ein-/Ausgaberegister, durch das der Zeitgeber den aktuellen Wert eines Rückwärtszählers mitteilt. Durch ↑geschäftiges Warten verzögert sich der Prozess und prüft (`probe`) dabei, ob der Rückwärtszähler den Wert null angenommen hat,
Aber auch in diesem Fall kann der Prozess in seiner Warteschleife immer noch durch Unterbrechungen verzögert werden. Die Folge davon wäre auch hier, dass der Prozess möglicherweise länger warten wird als seine Ruhezeit vorgibt. Daraus folgt weiter, dass die Ruhezeiten der wettstreitigen Prozesse (in einem realen System) immer nur mehr oder weniger genaue Annäherungswerte sein können — womit der leistungsmindernden Auswirkung von ↑Wettstreit unter den Prozessen nicht sicher mit solchen Verfahren vorgebeugt werden kann.

Steuerbus (en.) ↑*control bus*. Verbindungssystem (↑*bus*) in einem ↑Rechner, über das Kontroll- und Statussignale zwischen ↑CPU, ↑Hauptspeicher und ↑Peripherie übermittelt werden.

Steuereinheit (en.) ↑*controller*. Zentraler Bestandteil eines ↑Rechensystems, in dem Rechenoperationen ablaufen und der die ↑Steuerung weiterer Funktionen übernimmt (in Anlehnung an den Duden). Diese Einheit kann in Hard-, Firm- oder Software verwirklicht vorliegen. Typische Beispiele solcher Steuerfunktionen finden sich im Zusammenhang mit der Interaktion zwischen einem ↑Peripheriegerät und der ↑CPU, dem ↑Speicherdirektzugriff (↑DMA *controller*) und der Weiterleitung von ↑Unterbrechungsanforderungen (↑PIC). Für gewöhnlich sind diese Funktionen in einem eigenen, dem ↑Betriebssystem durch ↑Geräteregister zugänglichen ↑Halbleiterbaustein gekapselt. Allerdings können solche Steuerfunktionen, gegebenenfalls sogar ergänzt um ↑Speicher (d.h., Varianten von ↑RAM oder ↑ROM), auch mit der CPU zusammen im selben Halbleiterbaustein integriert sein (↑MCU).

Steuersprache (en.) ↑*control language*. Bezeichnung für eine Programmiersprache zur Formulierung eines ↑Stapelauftrags, eine ↑Auftragssteuersprache. Als Abkürzung CL auch die Bezeichnung für die in einigen ↑Betriebssystemen von IBM verwendete Programmiersprache zur Automatisierung von Systemabläufen, Konfigurationssteuerung oder zum Parametrisieren und Starten von ↑Programmen zur Generierung kaufmännischer Listen.

Steuerung (en.) ↑*control*. Vorgang in einem System, bei dem eine oder mehrere Größen als Eingangsgrößen andere Größen als Ausgangsgrößen aufgrund der dem System eigentümlichen Gesetzmäßigkeiten beeinflussen (DIN IEC 60050-351). Unterschied zur ↑Regelung ist der offene Wirkungsweg: Ausgangsgrößen wirken nicht auf Eingangsgrößen zurück, wodurch Störgrößen nicht ausgeglichen werden können. Gleichwohl unterliegt der Vorgang einer bestimmten ↑Echtzeitbedingung, die durch den zu steuernden physikalisch-technischen Prozess vorgegeben ist.

Steuerwerk (en.) ↑*control unit*. Eine auch als ↑Leitwerk bezeichnete Funktionseinheit im ↑Prozessor, die für den koordinierten Ablauf der Verarbeitung von ↑Maschinenbefehlen sorgt. Kernfunktion dieser Einheit ist der ↑Abruf- und Ausführungszyklus, der die ↑Interpretation der Maschinenbefehle bewirkt und durch Steuersignale den Betrieb der anderen Funktionseinheiten des ↑Rechners (Speicherwerk, Rechenwerk und Ein-/Ausgabewerk) leitet. Darüber hinaus regelt diese Einheit die Handhabung von ↑Unterbrechungsanforderungen, je nach ↑Rechnerarchitektur als integrale Funktion des Prozessors selbst oder im Zusammenspiel mit externen Hardwarekomponenten (↑PIC, ↑APIC).
Das Leitwerk kann festverdrahtet im Prozessor integriert sein oder es entnimmt die Vorschrift zu seiner Vorgehensweise einem ↑Mikroprogramm. Letzteres ist die für ↑CISC typische Umsetzung (↑*microprogram control unit*), wohingegen ersterer Ansatz kennzeichnend für einen ↑RISC ist. Der Vorteil festverdrahteter Leitwerke ist der für gewöhnlich dadurch erreichbare schnellere Ablauf eines einzelnen Maschinenbefehls. In dem Fall ist jedoch die Komplexität

der jeweiligen Befehle oft stark eingeschränkt, da die direkte Umsetzung in Hardware schwierig ist und die Realisierung umfangreicher Funktionen damit eher ausscheidet. Demgegenüber bedeutet der mikroprogrammierte Ansatz die Umsetzung von Befehlen in Firmware und damit auch Lösungen von nahezu beliebiger Komplexität, die dann allerdings auch einen für gewöhnlich langsameren Ablauf der Befehlsausführung impliziert. Für dieselbe, von einem ↑Prozess durchzuführende Operation können demnach mehr (RISC) oder weniger (CISC) Maschinenbefehle anfallen, was die gleiche, eine längere oder auch eine kürzere ↑Ausführungszeit für den betreffenden Abschnitt im ↑Maschinenprogramm nach sich ziehen kann. In funktionaler Hinsicht bleiben diese unterschiedlichen Auslegungen eines Leitwerks oberhalb der ↑Befehlssatzebene im Allgemeinen jedoch verborgen.

Störleitung (en.) ↑*interrupt line*. Übertragungsweg (Kabel, Draht, Leiterbahn), auf dem das ↑Signal für eine ↑Unterbrechungsanforderung von einem ↑Peripheriegerät zur ↑CPU gesendet wird. Der Eingang des Signals ruft eine Störung (*interruption*) des durch die CPU betriebenen ↑Programmablaufs hervor.

storage medium (dt.) ↑Speichermedium.

store siehe (en.) ↑*memory*.

Stoßbetrieb (en.) ↑*burst mode*. Verfahren zum ↑Speicherdirektzugriff. Die Steuereinheit (↑DMA *controller*) transferiert einen Block von ↑Daten in einem Stück, sobald ihr Zugriff auf den ↑Systembus gewährt wurde. Während des Transfers bleibt der ↑CPU der Zugriff auf den Systembus verwehrt, ihre Leistung wird wesentlich beeinträchtigt.

Stoßzeit (en.) ↑*burst time*. Maß für die Dauer des ↑Rechenstoßes eines ↑Prozesses. Zur Bestimmung dieses Intervalls ist die *Start-* und *Endzeit* eines Rechenstoßes aufzuzeichnen. Die entsprechenden ↑Zeitstempel sind beim ↑Prozesswechsel zu setzen, wobei dann für den wegschaltenden Prozess seine Endzeit und für den anschaltenden Prozess seine Startzeit im jeweiligen ↑Prozesskontrollblock zu vermerken ist. Aus beiden gemessenen Zeiten lässt sich durch Differenzbildung der Zeitverbrauch eines Rechenstoßes sehr einfach ermitteln:

$$t_{use} = |p_{end} - p_{start}|.$$

Allerdings umfasst dieser Wert sämtliche Verzögerungszeiten, die der Prozess während seines Rechenstoßes erfahren hat. Dies sind insbesondere Zeiten, die mit jeder einzelnen ↑Unterbrechungsbehandlung zu Buche schlagen. Diese Zeiten sind pro Prozess zu akkumulieren und die resultierende Summe ist von der festgestellten Zeitdifferenz abzuziehen:

$$t_{burst} = t_{use} - t_{delay}.$$

Dies ergibt die *normalisierte Rechenstoßlänge* eines Prozesses, in die durch eventuelle ↑Unterbrechungsanforderungen entstehende ↑Gemeinkosten nicht einfließen.
Diese Verzögerungszeiten sind bei der Unterbrechungsbehandlung zu messen. Die Prozessverzögerung startet im Moment des Übergangs von der ↑Benutzerebene zur ↑Systemebene und endet im Moment der Rückkehr zur Benutzerebene. Hierzu sind in jedem ↑Unterbrechungshandhaber erster Stufe (↑FLIH) die Zeitstempel für die entsprechenden Start- und Endzeitpunkte der Verzögerung zu setzen (vgl. S. 315). Der Betragswert der Differenz ergibt die während des laufenden Rechenstoßes eines Prozesses auf Systemebene (durch ↑Unterbrechungen bedingt) verstrichene Zeit, die der bisher angefallenen und im Prozesskontrollblock des betreffenden Prozesses vermerkten Verzögerungszeit (t_{delay}) hinzuzuzählen ist. Diese Verzögerungszeit ist auf 0 zu setzen, wann immer der Prozess angeschaltet wird (s.o.) und damit seinen nächsten Rechenstoß startet (t_{start}).

strict consistency (dt.) ↑strikte Konsistenz.

strikte Konsistenz (en.) ↑*strict consistency*. Modell der ↑Speicherkonsistenz, nach dem jeder ↑Prozessor bei jeder Leseoperation aus dem ↑Arbeitsspeicher immer den Wert erhält, der von einem beliebigen (anderen) Prozessor zuletzt an dieselbe ↑Adresse geschrieben wurde. Stärkste Form der Speicherkonsistenz (↑*strong consistency*).

strong consistency (dt.) ↑starke Konsistenz.

subject (dt.) Arbeitsstück, Gegenstand, Medium, ↑Subjekt.

Subjekt (en.) ↑*subject*. Bezeichnung für ein mit Bewusstsein ausgestattetes, denkendes, erkennendes, handelndes Wesen (Duden). Ein Individuum (↑Prozess), das ein ↑Zugriffsrecht auf ein ↑Objekt wünscht oder hat.

subprogram (dt.) ↑Unterprogramm.

SunOS siehe ↑Solaris.

Superblock Satz von ↑Verwaltungsdaten, die ein ↑Dateisystem charakterisieren (↑UNIX).

superblock (dt.) ↑Superblock.

supervisor (dt.) ↑Hauptsteuerprogramm.

supervisor mode siehe ↑*privileged mode*.

swap area (dt.) ↑Umlagerungsbereich.

swapping (dt.) ↑Umlagerung.

swapping means (dt.) Tauschmittel, ↑Umlagerungsmittel.

Symbol Sinnbild einer ↑Adresse, deren symbolhafte Bezeichnung.

symbol table (dt.) ↑Symboltabelle.

symbolic link (dt.) ↑symbolische Verknüpfung.

symbolische Adresse (en.) ↑*label*. Bezeichnung einer ↑Adresse, die als ↑Symbol für ein Strukturmerkmal eines Programms (d.h., ein Sprungziel oder der ↑Platzhalter einer ↑Variablen oder Konstanten) steht und zeichenhaftig (mnemonisch) dargestellt ist. Das Symbol ist durch einen ↑Binder in eine Nummer aufzulösen, bevor ein direkter Zugriff über die damit verknüpfte Adresse möglich ist: eine ↑Bindung ist herzustellen. Diese Nummer ist ein Element im ↑Adressraum von dem ↑Prozess, der durch eben dieses Programm bestimmt ist und den Zugriff bewirkt. Dabei kann die Auflösung vor oder zur ↑Ladezeit beziehungsweise sogar erst zur ↑Laufzeit des Programms stattfinden, die Bindung ist damit statisch oder dynamisch.

symbolische Verknüpfung Paarung von ↑Dateiname und ↑Indexknoten durch eine ↑symbolische Adresse. Im Unterschied zur ↑Verknüpfung mit einer ↑Indexknotennummer wird ein ↑Pfadname genutzt, um die Beziehung zu einer ↑Datei zu erstellen. Der Pfadname ist selbst als spezielle Datei gespeichert, was durch ein Attribut im Indexknoten kenntlich gemacht ist. Bei der ↑Namensauflösung wird der zum Dateinamen gesuchte endgültige Indexknoten sodann über den gespeicherten Pfadnamen ermittelt. Damit lassen sich beliebige Verknüpfungen erstellen, insbesondere solche, die die sonst azyklische Struktur von dem ↑Dateibaum durchbrechen (indem ein ↑Verzeichnis adressiert wird) oder auf eine Datei in einem anderen (an einen ↑Befestigungspunkt eingehängtes) ↑Dateisystem verweisen.

symbolischer Maschinenkode Zeichenhaftige, mnemonische Darstellung von ↑Maschinenkode, indem jedem Befehl seine Bedeutung durch eine Merkhilfe als ↑Mnemon gegeben wird. Beispielsweise steht `ADD EAX` für die Addition mit dem 32-Bit breiten Akkumulator von einem ↑x86-kompatiblen ↑Prozessor.

Symboltabelle (en.) ↑*symbol table*. Eine beim ↑Assemblieren erstellte Datenstruktur, die jedem ↑Symbol in einem ↑Programm Attribute zuordnet, die dann beim ↑Binden ausgewertet werden. Die Attribute geben Hinweise beispielsweise auf Typ, Größe, Gültigkeitsbereich (lokal/global) und Lage (absolut/relativ) des mit dem Symbol bezeichneten Abschnitts von ↑Text/↑Daten eines Programms.

symmetric multiprocessing (dt.) ↑symmetrische Simultanverarbeitung.

symmetric multiprocessor (dt.) symmetrischer ↑Multiprozessor, auch ↑symmetrisches Multiprozessorsystem.

symmetric scheduling (dt.) ↑symmetrische Planung.

symmetrische Planung (en.) ↑*symmetric scheduling*. Modell der ↑Ablaufplanung für ein ↑symmetrisches Multiprozessorsystem. Anders als ↑asymmetrische Planung kann grundsätzlich jeder ↑Prozess auf jeden ↑Prozessor des Systems stattfinden, da ↑Homogenität der Hardware das grundlegende Prinzip bildet. Technisch basiert diese Form der Ablaufplanung auf eine allen Prozessoren gemeinsame ↑Bereitliste, das heißt, auf die diese Liste implementierende Datenstruktur wird von verschiedenen Prozessoren aus zugegriffen. Jeder dieser Prozessoren wird zur Ausführung des ↑Planers herangezogen, wodurch die damit erfolgende ↑Einplanung auch implizit jedem dieser Prozessoren zu Gute kommt und einen Prozess zur ↑Einlastung bereit stellt. Steht ein ↑Prozesswechsel an, wird sich (bildlich gesprochen) jeder Prozessor selbst um einen anderen Prozess bemühen. Demnach wird ein Prozessor nur dann in den ↑Leerlauf eintreten müssen, wenn insgesamt weniger ↑Aufgaben zur Bearbeitung anstehen als Prozessoren in dem System verfügbar sind.
Problematisch an diesem Modell ist die vergleichsweise intensive Kopplung ↑gleichzeitiger Prozesse bei der Einplanung (befüllen/umsortieren der Bereitliste) und Einlastung (leeren der Bereitliste): alle Operationen, die den Zustand der Bereitliste veränderen können, unterliegen der ↑Synchronisation. Dies erzeugt ↑Interferenz wann immer Prozesse von verschiedenen Prozessoren ausgehend andere Prozesse bereitstellen (↑*scheduling*) oder abfertigen (↑*dispatching*) und beschränkt sich nicht nur auf die Momente der ↑Arbeitsteilung oder des ↑Arbeitsentzugs, wie etwa bei der asymmetrischen Planung. Letztere Formen der Arbeitsversorgung helfen zudem nur, um Imbalancen bei mehreren Bereitlisten auszugleichen und bleiben für eine allen gemeinsame Bereitliste ohne Bedeutung. Eine Abschwächung von Interferenzen bei konkurrierenden Listenoperationen ist hier nur noch durch Synchronisationsverfahren erreichbar, die möglichst wenig Störung auf die betroffenen Prozesse ausüben.

symmetrische Simultanverarbeitung (en.) ↑*symmetric multiprocessing*. ↑Betriebsart für zwei oder mehr gleiche (identische) Recheneinheiten, eng gekoppelt über ein gemeinsames Verbindungssystem. Typischerweise ist ein solches System von Recheneinheiten durch einen homogenen ↑Multiprozessor oder ↑Mehrkernprozessor repräsentiert. Auf jeder einzelnen Recheneinheit kann wenigstens ein ↑Prozess stattfinden, echt simultan mit Prozessen anderer Recheneinheiten. Zudem könnten einzelne oder alle Recheneinheiten einem ↑Multiplexverfahren unterzogen sein und somit jeweils mehr als einen Prozess pseudo-simultan ermöglichen.

symmetrisches Multiprozessorsystem (en.) ↑*symmetric multiprocessor* (↑SMP). Bezeichnung für ein ↑Rechensystem, das aus wenigstens einen ↑Multiprozessor oder ↑Mehrkernprozessor aufgebaut ist und dessen Recheneinheiten symmetrisch ausgelegt sind. Die Symmetrie ist in der ↑Homogenität der Hardware begründet, wodurch insbesondere ein für alle Prozessoren einheitlicher ↑Befehlssatz definiert und damit die Grundlage für ↑symmetrische Simultanverarbeitung gegeben ist: die Prozesse können auf jeden beliebigen Prozessor des Systems stattfinden (↑*symmetric scheduling*).

synchrone Ausnahme (en.) *synhronous* ↑*exception*. Eine im ↑Prozess selbst begründete ↑Ausnahme. Bleiben das ↑Programm, das den Prozess in statischer Hinsicht bestimmt, und die Eingabedaten, die den Prozess in dynamischer Hinsicht bestimmen, unverändert, wird der

Prozess immer wieder an derselben Stelle (↑Adresse in seinem ↑Adressraum) in dieselbe Falle (↑*trap*) tappen: die ↑Ausnahmesituation ist vorhersehbar und reproduzierbar.
Beispiele dafür sind ein ungültiger ↑Maschinenbefehl, eine falsche ↑Adressierungsart, ein ↑Schutzfehler oder ein ↑Seitenfehler — wobei letzterer jedoch differenziert zu betrachten ist. Kommt es zum Seitenfehler, weil (a) ↑virtueller Speicher die Grundlage für den Prozess bildet und (b) die ↑Ersetzungsstrategie eine globale Wirkung auf Prozesse ausübt, dann ist diese Ausnahme für gewöhnlich weder vorherseh- noch reproduzierbar, sondern asynchron hervorgerufen (vgl. ↑asynchrone Ausnahme).

Exkurs Die Ursache des Seitenfehlers in solch einer Konstellation liegt dann nicht in dem Prozess selbst, auch wenn dieser synchron gemäß Programmvorschrift auf den betreffenden ↑Maschinenbefehl zugreift und diesen damit zur Ausführung durch die ↑CPU bringt (↑*fetch-execute-cycle*). Vielmehr liegt die Ursache in der ↑Seitenumlagerung der Ersetzungsstrategie, wenn letztere nämlich nicht gleichlaufend (also asynchron) zum fraglichen Prozess diesem einen ↑Seitenrahmen entzieht und damit eine Seite auslagert, auf die der Prozess gegebenenfalls zugreifen wird. Dann ist zwar der Seitenzugriff des Prozesses synchron, nicht aber die ihn in diesem Zusammenhang möglicherweise treffende Ausnahme und damit ↑Unterbrechung. Steht jedoch eine lokale Ersetzungsstrategie hinter der Seitenumlagerung, könnte der Prozess in dem Fall und durch geschicktes Vorgehen möglichen Seitenfehlern vorbeugen — sofern das ihn bestimmende Programm (a) Kenntnis von dem in der Größe begrenzten Verfügungsrahmen von Seitenrahmen für den Prozess hat, (b) die Ersetzungsstrategie (bevorzugt ↑FCFS) nachvollzieht und (c) so organisiert ist, dass es beim Zugriff auf eine Seite entweder sicher oder niemals zur Seitenumlagerung kommt. Dann wäre die durch einen Seitenfehler verursachte Ausnahme synchron.

Synchronisation (en.) ↑*synchronisation*. Ergebnis der ↑Synchronisierung. Die beteiligten ↑Prozesse unterliegen einer ↑Koordination der ↑Kooperation und ↑Konkurrenz untereinander.

synchronisation (dt.) ↑Synchronisierung, ↑Synchronisation.

synchronisation variable (dt.) ↑Synchronisationsvariable.

Synchronisationsvariable (en.) ↑*synchronisation variable*. Eine ↑Variable, die ein ↑nichtsequentielles Programm deklariert/definiert und ein ↑nichtsequentieller Prozess benutzt, um die ↑Synchronisation gleichzeitiger ↑Aktionen herzustellen. Diese Variable ist für gewöhnlich Bestandteil des ↑Maschinenprogramms, wenn für einzelne oder alle (Teil-) ↑Prozesse eine ↑Koordinierung ihrer Aktionen erforderlich ist. Darüberhinaus kann eine solche Variable aber auch durch das ↑Betriebssystem bereitgestellt und verwaltet werden, insbesondere falls mehrere unter ↑Adressraumisolation in ↑Simultanverarbeitung ablaufende Maschinenprogramme sich untereinander synchronisieren müssen.
Für gewöhnlich speichert eine solche Variable keinen Wert, der für den sie benutzenden Prozess zugänglich ist. Typische Beispiele derartige „abstrakte Behälter“ sind die ↑Bedingungsvariable und der ↑Semaphor.

Synchronisierung (en.) ↑*synchronisation*. Vorgänge zeitlich aufeinander abstimmen, sie in einer bestimmten Reihenfolge stattfinden lassen, ↑Synchronisation herstellen. Die Maßnahmen dazu können eine ↑blockierende Synchronisation oder ↑nichtblockierende Synchronisation der betroffenen Vorgänge bewirken.

system bus (dt.) ↑Systembus.

system call (dt.) ↑Systemaufruf.

system call error (dt.) ↑Systemaufruffehler.

system console (dt.) ↑Systemkonsole.

system control (dt.) ↑Systemsteuerung.

system function (dt.) ↑Systemfunktion.

system level (dt.) ↑Systemebene.

system mode (dt.) Systemmodus: ↑Arbeitsmodus.

system process (dt.) ↑Systemprozess.

system program (dt.) ↑Systemprogramm.

system programming (dt.) ↑Systemprogrammierung.

system programming language (dt.) ↑Systemprogrammiersprache.

system software (dt.) ↑Systemsoftware.

system space (dt.) ↑Systembereich, -gebiet, -gelände.

system time (dt.) ↑Systemzeit.

system-call dispatcher (dt.) ↑Systemaufrufzuteiler.

system-call instruction (dt.) ↑Systemaufrufbefehl.

system-call number (dt.) ↑Systemaufrufnummer.

system-call parameter (dt.) ↑Systemaufrufparameter.

system-call stub (dt.) ↑Systemaufrufstumpf.

Systemaufruf (en.) ↑*system call*. Aufruf von einem im ↑Betriebssystem liegenden ↑Unterprogramm, initiiert durch eine in einem ↑Maschinenprogramm kodierte ↑Aktion. Letztere kommt durch einen ↑Systemaufrufbefehl zum Ausdruck.

Systemaufrufbefehl (en.) ↑*system-call instruction*. Anweisung an das ↑Betriebssystem zur Ausführung einer bestimmten Operation (in Anlehnung an den Duden). Diese Anweisung beschreibt eine ↑Aktionsfolge, die zum ↑Systemaufruf führt: sie umfasst nicht nur den ↑Unterbrechungsbefehl, um einen ↑Moduswechsel zu bewirken und das Betriebssystem zu aktivieren, sondern auch den ↑Operationskode und, soweit benötigt, die Operanden, um damit die ↑Systemfunktion zu identifizieren und mit Argumenten zu versehen. Technisch umgesetzt im ↑Maschinenprogramm, und zwar als ↑Komplexbefehl oder ↑Primitivbefehl der ↑Betriebssystemebene, bildet die betreffende Anweisung für gewöhnlich eine Sequenz semantisch zusammenhängender ↑Maschinenbefehle der ↑Befehlssatzebene.

Systemaufruffehler (en.) ↑*system call error*. Ein bei der Durchführung eines ↑Systemaufrufs im ↑Betriebssystem erkannter ↑Fehler. Dieser Fehler bezieht sich immer auf den ↑Prozess, der den Systemaufruf abgesetzt hat und deutet nicht etwa auf eine Fehlfunktion des Betriebssystems hin. Typische Beispiele dafür sind eine unbekannte ↑Systemaufrufnummer, ungültige ↑Systemaufrufparameter, unzureichende ↑Zugriffsrechte beziehungsweise fehlende ↑Befähigung des Prozesses oder gar eine Erschöpfung prozesslokaler ↑Ressourcen.
Ein solcher Fehler wird das Betriebssystem nicht in ↑Panik versetzen und stattdessen zur Fortsetzung des betreffenden Prozesses führen, damit dieser nach Beendigung/Abbruch des Systemaufrufs eine Fehlerbehandlung durchführen kann. Es handelt sich dabei immer um eine ↑Ausnahmesituation des den Systemaufruf hat absetzenden Prozesses, und zwar während dieser im Betriebssystem tätig ist.
Eine diesbezügliche Fehlerreaktion geschieht für gewöhnlich immer innerhalb des ↑Maschinenprogramms, in dem der Systemaufruf kodiert vorliegt und das den Prozess definiert, in dessen Namen (↑*process identifaction*) das Betriebssystem tätig geworden ist. Je nach Typ des Systemaufrufs, also ob dieser durch einen ↑Komplexbefehl oder einen ↑Primitivbefehl ausgelöst wurde, sowie der Art der Fehlermeldung und des jeweiligen Fehlerkodes, ergeben

sich verschiedene Muster von ↑Aktionsfolgen innerhalb eines ↑Systemaufrufstumpfes, um auf den angezeigten Fehler zu reagieren. Allen Mustern gemein sind Anweisungen, die den Konventionen der Programmiersprache, in der das Maschinenprogramm formuliert wurde, beziehungsweise des zugehörigen ↑Übersetzers einerseits und des Betriebssystems andererseits entsprechen. Diese Konventionen legen fest, ob und gegebenenfalls wie ein Fehlerkode des Betriebssystems vermerkt und an die den Systemfunktionsstumpf hat aufgerufene Prozedur zurückgeliefert werden soll. Für Programmiersprachen mit integriertem Konzept zur ↑Ausnahmebehandlung könnten diese Anweisungen auch zur Erhebung (*raising*) einer ↑Ausnahme führen, mit dem Fehlerkode als ↑tatsächlicher Parameter des ↑Ausnahmehandhabers.

Exkurs In den nachfolgenden Beispielen wird ↑C angenommen und damit eine ↑Abstraktionsebene ohne Sprachkonzepte zur Ausnahmebehandlung. Zunächst das für Komplexbefehle typische Anweisungsmuster (vgl. S. 137):

```
sos                     # switch to operating system
↑Pseudobefehle          # system-call parameters: operand list
jc __syscall_error      # if error code, handle that error; else, continue
```

Auf Grundlage dieser Art von ↑Systemaufrufbefehl geschieht die Fehlermeldung des Betriebssystems für gewöhnlich mittels Zustandsmerker im ↑Statusregister, hier der Übertragsmerker (*carry bit*, C). Ist dieses Bit gesetzt, signalisiert das Betriebssystem damit einen bei der ↑Teilinterpretation des Systemaufrufs aufgetretenen Fehler. Die bedingte Sprunganweisung (`jc`) prüft den Übertragsmerker und lässt den aus dem Betriebssystem soeben erst zurückgekehrten Prozess gegebenenfalls ($C == 1$) zur Fehlerbehandlung verzweigen. Anderenfalls ($C == 0$) schreitet der Prozess fehlerlos fort. Setzen oder Löschen dieses Merkers ist für gewöhnlich die Aufgabe des ↑Systemaufrufzuteilers (vgl. S. 343).
Das Anweisungsmuster für Primitivbefehle gestaltet sich sehr ähnlich, nur stehen hier vor dem ↑Unterbrechungsbefehl (`sos`) die Anweisungen zur Auswertung und Bereitstellung der Systemaufrufparameter (vgl. S. 226):

```
↑Maschinenbefehle       # system-call parameters: operand evaluation
sos                     # switch to operating system
jc __syscall_error      # if error code, handle that error; else, continue
```

Wenn allerdings das Betriebssystem einen gescheiterten Systemaufruf nicht durch einen Zustandsmerker im Statusregister anzeigt, sondern durch einen speziellen Wert der angeforderten Systemfunktion mitteilt, muss nach Rückkehr zur ↑Benutzerebene daselbst geprüft werden, ob aus dem gelieferten Rückgabewert des Systemaufrufs eine Fehlermeldung hervorgeht. Gemeinhin ist dazu die Zielmenge für den Rückgabewert zweigeteilt, beispielsweise:

$$\mathbb{N}_0 = \{0; 1; 2; \ldots; n\} \cup \{n+1; \ldots 2^k - 1\},$$

mit k gleich der ↑Wortbreite der ↑CPU. Normale Rückgabewerte liegen im Bereich $[0, n]$, wohingegen einen Fehler anzeigende Rückgabewerte im Bereich $[n+1, 2^k - 1]$ liegen. Die Anzahl von Fehlerkodes ist vergleichsweise gering, so dass für normale Rückgabewerte ein wesentlich größerer Bereich definiert ist und n damit „gering kleiner“ als $2^k - 1$ ausfällt. Werden die Rückgabewerte nun als Ganzzahl $\mathbb{Z}$ gedeutet, wird der Fehlerkodebereich eine vergleichsweise kleine Menge der größten negativen Zahlen abdecken: $[-n+1, -1]$. Diese Lösung wird beispielsweise mit ↑Linux verfolgt, wobei $n = 4096$ gilt. Demzufolge gestaltet sich das Anweisungsmuster für einen Systemaufruf etwas anders — zudem sind Systemaufrufe in Linux als Primitivbefehl ausgelegt:

```
↑Maschinenbefehle
                        # system-call parameters: operand evaluation
sos                     # switch to operating system
cmpl $-4095, %eax       # check return value as to error-code range
jae                     # if error code, handle that error; else, continue
__syscall_error_neg
```

Dieser Ansatz erfordert, dass zum Abschluss eines Systemaufrufs keine Systemfunktion einen

regulären Rückgabewert größer als $|n|$ liefert. So ist die Implementierung von Systemfunktionen grundsätzlich abhängig von der Art und Weise wie Fehlerkodes von der System- zur Benutzerebene zurückgeliefert werden (↑*cross-cutting concern*).
Das ↑Laufzeitsystem der Programmiersprache C (↑libc) liefert für einen gescheiterten Systemaufruf den Wert -1, wohingegen der Fehlerkode des Systemaufrufs selbst differenziert Auskunft über den Scheiterungsgrund gibt. Im Fehlerfall finden dazu folgende Schritte statt:

```
__syscall_error_neg:   # come here for negative error code
    neg %eax           # make absolute value and join common exit

__syscall_error:       # general error return sequence
    movl %eax, errno   # remember specific error code
    movl $-1, %eax     # provide unspecific error code
    ret                # unsuccessful return
```

Diese allgemeine Ausstiegssequenz von Maschinenbefehlen in einem Systemaufrufstumpf korrespondiert zu den zwei Arten der Fehlermeldung durch ein Betriebssystem, nämlich Zustandsmerker im Statusregister einerseits (`__syscall_error`) beziehungsweise einen bestimmten Bereich größter negativer Zahlen aus der Zielmenge für den Rückgabewert der angeforderten Systemfunktion andererseits (`__syscall_error_neg`). Erfolgt nun die Fehleranzeige des Betriebssystems nicht über Merker im Statusregister, sondern sind diese durch explizite Abfrage (`cmpl $-4095, %eax; jae __syscall_error_neg`), ob der Rückgabewert im Fehlerbereich liegt, im Maschinenprogramm selbst zu setzen, ist vor Sicherung des Fehlerkodes noch sein absoluter Betrag zu bilden (`neg`).
Beiden Varianten gemeinsam ist, den im Rückgaberegister (`eax`) enthaltenen differenzierenden Fehlerkode der per Systemaufruf angeforderten Systemfunktion lediglich in einer globalen ↑Variablen (`errno`) zu vermerken und den Wert -1 als Rückgabewert der als Systemaufrufstumpf repräsentierten Funktion zu liefern. Dies ist die seit ↑UNIX typische Vorgehensweise für POSIX-kompatible Systemaufrufe (vgl. `intro(2)`).
Der Unterschied zwischen diesen beiden gebräuchlichen Fehlermeldungsarten ist weniger in der geringeren Anzahl von Maschinenbefehlen zu sehen, die im Maschinenprogramm selbst zur Erkennung und Behandlung der Ausnahmesituation anfallen: 0 bei Fehlermeldung mittels Statusregister oder 2 bei Abfrage des Fehlerbereichs. Auch der betriebssystemseitige Aufwand, um eine Fehlermeldung zur Benutzerebene zu propagieren, ist nicht mit sehr viel Aufwand verbunden: 0 Maschinenbefehle bei Mitteilung über den Fehlerbereich oder $3 \leq n$ Maschinenbefehle zum Vermerk des ↑Stapelzeigers im Moment der ↑Unterbrechungsbehandlung sowie zum Laden des ↑Zeigers für die Durchführung einer `or`- oder `and`-Operation zur Manipulation des Zustandsmerkers in der im Laufzeitstapel des Betriebssystems liegenden Sicherungskopie des Statusregisterinhalts.
Viel stärker ins Gewicht fällt der ↑Querschnittsbelang beider Varianten und damit der Einfluss auf den Aufbau und die Struktur der Betriebssystemsoftware. Eine Fehlermitteilung zu ermöglichen, indem der normale Wertebereich der von Systemfunktionen durchzuführenden Berechnungen durch Abtrennung eines Fehlerbereichs beschnitten wird, ist in logischer Hinsicht grundsätzlich problematisch, aber erst bei kleiner ↑Wortbreite wirklich kritisch. Die das Statusregister nutzende Alternative beschneidet den Wertebereich solcher Berechnungen nicht: der von `write` gelieferte Wert (`int`), die Anzahl der geschriebenen ↑Daten, kann dem als dritten Parameter übergebenen Größenwert (`int`), die Anzahl der zu schreibenden Daten, in jeder Hinsicht entsprechen. Darüber hinaus spiegelt dieser Ansatz das ↑Operationsprinzip einer CPU in Bezug auf ihr ↑Programmiermodell wider, durch ihn gleicht die ↑partielle Interpretation des Systemaufrufbefehls durch das Betriebssystem der Interpretation eines Maschinenbefehls durch die CPU — eine Analogie, die jedoch bei Auslegung als Komplexbefehl noch stärker ausgeprägt ist.

Systemaufrufnummer (en.) ↑*system-call number*. Ein bestimmter ↑Operationskode, der den ↑Systemaufruf an ein ↑Betriebssystem identifiziert und normalerweise als fortlaufende Nummer dargestellt ist. Diese Nummer wird vom ↑Systemaufrufzuteiler auf eine interne Funktion

des Betriebssystems abgebildet.

Systemaufrufparameter (en.) ↑*system-call parameter*. Ein Argument für ein ↑Unterprogramm, das durch einen ↑Systemaufruf aktiviert wird. Für gewöhnlich ein ↑tatsächlicher Parameter einer ↑Systemfunktion, die die mit dem Systemaufruf bezweckte Operation im ↑Betriebssystem implementiert.

Systemaufrufstumpf (en.) ↑*system-call stub*. Programmabschnitt, der einen ↑Systemaufruf absetzt und das dazu erforderliche Protokoll zwischen dem umfassenden ↑Maschinenprogramm und dem aufzurufenden ↑Betriebssystem abwickelt. Der Systemaufruf wird kodiert, die ↑Systemaufrufparameter werden zur Übergabe ans Betriebssystem bereitgestellt, die ↑CPU wird angewiesen, die ↑partielle Interpretation des Maschinenprogramms durch das Betriebssystem zu ermöglichen und nach Rückkehr vom Systemaufruf wird auf eine ↑Ausnahmesituation geprüft. Dieser Programmabschnitt entspricht einer ↑Blattprozedur des Maschinenprogramms, die einen Aufruf des in Form einer ↑Wurzelprozedur im Betriebssystem repräsentierten ↑Systemaufrufzuteilers kodiert.

Exkurs Solche Aufrufstümpfe werden für gewöhnlich generiert aus der ↑Signatur der ↑Systemfunktion, die durch einen Systemaufruf zur Ausführung kommen soll. Im Wesentlichen kapselt ein Stumpf den ↑Systemaufrufbefehl an das Betriebssystem und Anweisungen zur Fehlerbehandlung. Gegebenenfalls fallen auch Anweisungen an, um den ↑Prozessorstatus des den Systemaufruf absetzenden ↑Prozesses konsistent zu halten. Letzteres ist insbesondere erforderlich, wenn der Systemaufrufbefehl festgelegte ↑Prozessorregister zur Argumentenübergabe an das Betriebssystem verwendet (↑Primitivbefehl). Nachfolgendes Beispiel skizziert einen Systemaufrufstumpf zur Signatur `int write(int, const char *, int)` eines ↑UNIX-ähnlichen Betriebssystems auf Grundlage eines solchen Befehls (↑x86):

```
write:                          # int write(int, const char *, int)
    pushl %ebx                  # save non-volatile register
    movl 16(%esp), %edx         # 3rd operand
    movl 12(%esp), %ecx         # 2nd operand
    movl 8(%esp), %ebx          # 1st operand
    movl $4, %eax               # operation code for the operating system
    sos                         # switch to operating system
    popl %ebx                   # restore non-volatile register
    jc __syscall_error          # if error flag set, unsuccessful return
    ret                         # successful return
```

Da der Systemaufrufbefehl zur Übergabe des ersten Parameters ein ↑nichtflüchtiges Register (`ebx`) vorschreibt, ist dessen Inhalt vor dem Befehl zu sichern (`push`) und nach Rückkehr vom Betriebssystem beziehungsweise vor Rücksprung zum ↑Benutzerprogramm wieder herzustellen (`pop`). Der eigentliche Systemaufrufbefehl umfasst die fünf Anweisungen zwischen diesen beiden Operationen auf dem ↑Laufzeitstapel, er schließt mit dem ↑Unterbrechungsbefehl (`sos`) ab. Das Beispiel geht davon aus, dass das Betriebssystem zur Anzeige von ↑Systemaufruffehlern den Übertragsmerker (*carry bit*, C) im ↑Statusregister nutzt. Entsprechend gestaltet sich die Fehlerabfrage durch eine bedingte Anweisung (`jc`): Nur im Fehlerfall ($C == 1$) kommt es zu einer Nachbehandlung des gescheiterten Systemaufrufs (vgl. S.339). Je nach Ausgang des Systemaufrufs enthält ein Rückgaberegister (`eax`) den Fehlerkode oder das Ergebnis der Systemfunktion.

Systemaufrufzuteiler (en.) ↑*system-call dispatcher*. Gegenstück zu einem ↑Systemaufrufstumpf; Programmabschnitt im ↑Betriebssystem, der die ↑Aktionsfolge festlegt, um einen ↑Systemaufruf anzunehmen und abzuwickeln: den Systemaufruf total und das ↑Maschinenprogramm partiell zu interpretieren. Das bedeutet, erstens, den Systemaufruf dekodieren und auf Gültigkeit prüfen. Ein ungültiger Systemaufruf begründet eine ↑Ausnahmesituation. Zweitens, die Weitergabe der ↑Systemaufrufparameter an das aufzurufende ↑Unterprogramm, welches

die angeforderte Systemfunktion implementiert. Abhängig vom ↑Operationsprinzip des Betriebssystems erfolgt die Überprüfung der Parameter hier bei ihrer Weitergabe oder später im Moment ihrer Verwendung. Ungültige Parameter begründen eine Ausnahmesituation. Drittens, die durch den Systemaufuf ausgelöste ↑partielle Interpretation des Maschinenprogramms beenden und die ↑CPU anweisen, die Interpretation des Maschinenprogramms wieder aufzunehmen. Dieser Programmabschnitt ist eine ↑Wurzelprozedur des Betriebssystems.

Exkurs Zur Verdeutlichung der Arbeitsschritte wird der in der ursprünglichen Fassung von ↑UNIX (V6) verwendete ↑Ausnahmezuteiler vorgestellt, aber nur soweit es die Entgegennahme und Ausführung von Systemaufrufen betrifft (↑*fetch-execute-cycle*). Neben den Systemaufrufen war dieser Zuteiler zusätzlich noch für alle anderen durch die Hardware hervorgerufenen Ausnahmen synchroner Natur (↑*trap*) zuständig.
Das besondere Merkmal der hier betrachteten Systemaufrufe ist der Verzicht auf die Nutzung von ↑Prozessorregister zur Parameterübergabe, im Gegensatz etwa zu ↑Linux und anderen zeitgenössischen Betriebssystemen. Jeder von diesem Zuteiler verarbeitete ↑Systemaufrufbefehl entspricht einem ↑Komplexbefehl, der je nach Auslegung *direkte* und *indirekte Systemaufrufe* ermöglchte. Der Unterschied zwischen diesen beiden Aufrufarten liegt in der für die Parameterübergabe verwendete ↑Adressierungsart der Befehlsopereranden. Im direkten Fall folgen alle zu übergebenden Parameter dem zur Auslösung des Systemaufrufs verwendeten ↑Unterbrechungsbefehl (`trap`, ↑PDP 11; entspricht `int`, ↑x86), sie sind die Operanden des Systemaufrufbefehls. Da diese Parameter im schreibgeschützten ↑Textsegment liegen, sind sie zugleich Konstante des ↑Maschinenprogramms. Ihre Werte wurden vor ↑Laufzeit des betreffenden Maschinenprogramms ausgewertet, sodass diese Parameter durch *unmittelbare Adressierung* zugreifbar sind. Demgegenüber folgt im indirekten Fall dem Unterbrechungsbefehl die ↑Adresse von einem *Systemaufrufdeskriptor* im ↑Datensegment. Hier ist nur die Deskriptoradresse eine Maschinenprogrammkonstante, während die eigentliche Paramterisierung variabel gestaltet ist, das heißt, die Operandenauswertung zur Laufzeit geschehen kann. Auf diesen ↑Deskriptor wird durch *direkte Adressierung* zugegriffen.
Beiden Fällen gemeinsam ist jedoch der Ansatz, den ↑Systemaufrufbefehl als erweiterten Maschinenbefehl (Komplexbefehl) aufzufassen und so auch im Programm entsprechend zu repräsentieren. Nachfolgend sind die Kodierungen dieser beiden Arten gegenübergestellt:

```
.text
trap 4 # direct system call
.long 1
.long L_.str
.long 12
```

```
    .text
    trap 0      # indirect system call
    .long proxy # descriptor address

    .data
proxy:
    .short 0104404
    .long 1
    .long L_.str
    .long 12
```

Das Beispiel links zeigt einen *direkten Systemaufruf* in ↑Assembliersprache, rechts ein Beispiel für den *indirekten Systemaufruf*. Allerdings bewirken beide Fälle den Aufruf derselben ↑Systemfunktion, nämlich `write(1, "Hello world!", 12)`. Das ↑Symbol `L_.str` steht für die Adresse der Zeichenfolge `Hello world!`, es wurde automatisch vom ↑Kompilierer (hier für ↑C) generiert. Diese Zeichenfolge liegt ebenfalls im Textsegment (hier nicht gezeigt). Da in diesem konkreten Fall alle Parameter zur Übersetzungszeit bekannt sind (der Systemaufruf `write` hat seit jeher in allen UNIX-ähnlichen Systemen die ↑Systemaufrufnummer 4), kann der Übersetzer den Kode für den direkten Systemaufruf vollständig generieren.
Eine Formulierung für denselben Systemaufruf, jedoch in indirekter Ausfertigung, zeigt das Beispiel rechts. Hier sind praktisch zwei Systemaufrufe hintereinandergeschaltet. Der im Textsegment liegende erste Systemaufrufbefehl (1-Adressbefehl) mit der speziellen Systemaufrufnummer 0 aktiviert die ↑Systemebene und sorgt sonst lediglich dafür, dass der im ↑Datensegment liegende zweite Systemaufrufbefehl (3-Adressbefehl) mit dem ↑Operationskode

0104404 abgerufen und ausgeführt wird. Das (zweiwortige) erste Glied bildet den *Hauptbefehl*, der das (vierwortige) zweite Glied als *Nebenbefehl* nachzieht. Der Operationskode des Nebenbefehls ist die Oktahlzahlendarstellung von `trap 4`, also ein wirklicher Maschinenbefehl (wie im linken Beispiel), der jedoch nicht von der CPU, sondern vom Betriebssystem interpretiert wird. Da der Nebenbefehl im Datensegment liegt, kann er zur Laufzeit von dem Maschinenprogramm selbst immer wieder neu zusammengestellt werden. Der Hauptbefehl ist immer ein 1-Adressbefehl, der Nebenbefehl ist (für das betreffende System, UNIX V6) ein n-Adressbefehl, mit $0 \leq n \leq 5$. Letzterer zum Nebenbefehl festgehaltene Aspekt trifft auch auf jeden Systemaufrufbefehl zu, der zum direkten Systemaufruf führt (linkes Beispiel).
Die Skizze des nachfolgend vorgestellten Zuteilers orientiert sich an die in UNIX V6 für die PDP 11 ursprünglich festgelegten Schritte. Dabei wird x86 als Plattform angenommen und dementsprechend auch von einem Systemaufrufbefehl ausgegangen, der die oben gezeigte Struktur hat und den Unterbrechungsbefehl wie folgt nachbildet:

```
.macro trap scn     # define trap instruction
    .byte 204       # 1-byte software interrupt (int3)
    .byte \scn      # system-call number (operation code)
.endm               # done
```

Die ↑partielle Interpretation des Systemaufrufbefehls beginnt mit dem Abruf des Operationskodes aus dem Maschinenprogramm. Zu diesem Zweck wird eine Zugriffsfunktion (`fubyte`) genutzt, die ein ↑Byte, nämlich die Systemaufrufnummer, aus dem ↑Adressraum des den Systemaufruf absetzenden ↑Prozesses der ↑Benutzerebene liest:

```
__attribute__ ((interrupt)) void syscall(train_t *this) {
    syscall_t *duty;
    word_t *data, name;

    duty = &sysentry[fubyte(this->ip, this->cs) & 077];
    this->ip += 1;                                /* advance user-level PC */

    if (duty != &sysentry[0]) {                     /* direct system call? */
        data = (word_t *)this->ip;             /* yes, locate operand part */
        name = this->cs;                                /* at text segment */
        this->ip += duty->argc * sizeof(word_t);        /* advance user PC */
    } else {                          /* no, indirect system call requested */
        sysorder_t *link;
        short code;

        link = (sysorder_t *)fuword(this->ip, this->cs);   /* read operand */
        this->ip += sizeof(link);                       /* advance user PC */

        code = fuword(&link->code, UDS);    /* read code from data segment */
        if ((code & ~077) != SYS) code = SYS   /* if need be, correct code */

        duty = &sysentry[code & 077];        /* select function descriptor */
        data = (word_t *)&link->argv[0];            /* locate operand part */
        name = UDS;                                     /* at data segment */
    }

    for (int argc = 0; argc < duty->argc; argc++, data++)    /* copy args. */
        u.argv[argc] = fuword(data, name);       /* from text/data segment */

    sysreturn(this, sysexecute(this, duty));
}
```

Die Adresse (`ip`) im Textsegment (`cs`) entstammt dem beim Systemaufruf im ↑Unterbrechungszyklus auf dem ↑Laufzeitstapel des Betriebssystems gesicherten ↑Prozessorstatus. Dies ist der während der partiellen Interpretation vom Betriebssystem verwaltete Wert des ↑Befehlszählers PC_{os} des unterbrochenen Prozesses im Maschinenprogramm. Der Operaktionskode (in den sechs niederwertigen Bits des abgerufenen Bytes, Maskierung mit `077` bzw.

$3f_{16}$) ist ein Index in ein Vektorfeld von ↑Exemplaren einer Struktur von zwei Attributen: Adresse und Argumentenanzahl der dem Systemaufruf zugeordneten ↑Systemfunktion. Die Adresse eines solchen Exemplars (`duty`) identifiziert die aufzurufende Systemfunktion. Nach erfolgtem Operationskodeabruf wird PC_{os} ein Byte weitergesetzt.
Wurde nicht der erste Eintrag im Vektorfeld selektiert (Operationskode > 0), handelt es sich um einen direkten Systemaufruf. In dem Fall markiert PC_{os} den Listenanfang (`data`) der aus dem Textsegment (`name`) zu lesenden Parameter. Darüber hinaus ist PC_{os} um die Bytelänge dieser Parameterliste weiterzusetzen.
Demgegenüber zeigt die Selektion des ersten Eintrags (Operationskode 0) im Vektorfeld einen indirekten Systemaufruf an. In dem Fall wird die Adresse (`link`) des den Nebenbefehl identifizierenden Systemaufrufdeskriptors aus dem Adressraum des den Systemaufruf absetzenden Prozesses der Benutzerebene gelesen (`fuword`) und PC_{os} um die Breite dieser Adresse entsprechend weitergesetzt. Der in diesem (im Datensegment (`UDS`) liegende) ↑Deskriptor vermerkte Operationskode (`code`) identifiziert die Argumtenenanzahl und die Adresse der im Vektorfeld verzeichneten Systemfunktion. Sollte dort die Kodierung eines unbekannten Unterbrechungsbefehls gespeichert sein, was auf einen ↑Fehler hinweist, wird mit der Kodierung des Unterbrechungsbefehls (PDP 11) für einen indirekten Systemaufruf weitergearbeiteit (`SYS` = 0104400). Wie beim direkten Systemaufruf wird der Operationskode (`code`) verwendet, um in dem Vektorfeld den Eintrag für die aufzurufende Systemfunktion zu selektieren. Die Parameter für diese Funktion liegen im Datensegment (`UDS`, `name`), genauer im Argumentenvektor (`data`) des Systemaufrufdeskriptors.
Nachdem nun bestimmt worden ist, aus welchem ↑Segment und von welcher Adresse die Systemaufrufparameter zu lesen sind, geschieht die eigentliche Parameterübergabe durch eine Zählschleife (`for`-*loop*). Wortweise (`fuword`) gelangt die entsprechend benötigte Anzahl von Argumenten von der Benutzer- zur Systemebene (`u.argv`). Anschließend wird die Systemfunktion — oder, falls der Nebenbefehl einen unbekannten Operationskode aufwies (s.o.), eine Nulloperation (`nullsys`) — ausgeführt:

```
inline int sysexecute(train_t *this, syscall_t *duty) {
    u.error = 0;                        /* clear (system level) error code */

    int done = (*duty->funp)(u.argv);           /* invoke system function */

    if (u.error == 0)                            /* any error indication? */
        this->flags &= ~EBIT;                     /* no, reset error flag */
    else {                                        /* yes, error reporting */
        this->flags |= EBIT;                            /* set error flag */
        done = u.error;                             /* provide error code */
    }

    return done                    /* successful or unsuccessful returning */
}
```

Diese ↑Mantelprozedur startet die gewünschte Systemfunktion, prüft eine eventuell bestehende Fehlermeldung (`u.error` $\neq 0$) und setzt oder löscht den Fehlermerker (`EBIT`) in der auf dem Laufzeitstapel liegenden Sicherungskopie des ↑Statusregisters. Im gegebenen Fall wird der Übertragsmerker (*carry flag*) verwendet, um Erfolg ($C = 0$) oder Misserfolg ($C = 1$) der Ausführung mitzuteilen. Der Rückgabewert oder der Fehlerkode der Systemfunktion wird durch ein Prozessorregister zur Benutzerebene übertragen:

```
inline void sysereturn(train_t *this, int done) {
    ((word_t *)this)[R0] = done;
}
```

Konkret wird hier die ↑Speicherstelle im Laufzeitstapel überschrieben, von der der Inhalt des für Rückgabewerte vorgesehenen und zuvor als Folge der ↑Unterbrechung gesicherten Prozessorregisters (`R0` entspricht `eax`) bei Rückkehr aus der ↑Unterbrechungsbehandlung wiederhergestellt wird.

Abschließend noch kurz eine Anmerkung zu den Operationen, um ↑Daten aus dem Adressraum der Benutzerebene zu lesen (`fubyte/word`: *fetch user byte/word*), in dem der den Systemaufruf getätigte Prozess isoliert ist. Die Implementierung dieser Operationen ist abhängig vom ↑Mehradressraummodell. Sind für die Benutzerebene teilprivate Adressräume definiert (Linux, ↑Windows NT), erfolgen die Zugriffe, ohne zuvor den betreffenden Adressraum erst zugänglich machen zu müssen. Anderenfalls, nämlich bei strikt privaten Adressräumen (UNIX V6, ↑macOS), ist die ↑MMU extra für einen Adressraumgrenzen überschreitenden Zugriff zu instruieren, damit die Leseoperation gelingen kann. Dazu ist entweder ein spezieller Maschinenbefehl (UNIX V6, PDP 11: `mfpi/d`, *move from previous instruction/data space*) erforderlich oder ein zwischen Benutzer- und Systemebene ↑gemeinsamer Speicherbereich (macOS) zeitweilig einzurichten.

Systembereich (en.) ↑*system space*, ↑*kernel space*. Bezeichnung für den ↑Adressbereich, den sich ein ↑Betriebssystemkern selbst zugesteht. Bezugsrahmen bildet ein bestimmter ↑logischer Adressraum oder ↑virtueller Adressraum, dessen Modell im ↑Betriebssystem verankert ist. In diesem Gebiet handelt die ↑CPU für das Betriebssystem (↑*system mode*), darüberhinaus verfügt ein ↑Prozess darin für gewöhnlich über uneingeschränkte Rechte (↑*privileged mode*).

Systembus (en.) ↑*system bus*. Gesamtheit der in einem ↑Rechner vorhandenen Sammelschienen, über die ↑CPU, ↑Hauptspeicher und ↑Peripherie gekoppelt sind: nämlich der ↑Adressbus, ↑Datenbus und ↑Steuerbus.

Systemebene (en.) ↑*system level*. Bezeichnung für die Stufe, im Sinne von sowohl Privilegien (↑*privileged mode*) als auch Kontext (↑Betriebssystem), auf der ein ↑Prozess gegenwärtig stattfindet. Gegenteil von ↑Benutzerebene.

Systemfunktion (en.) ↑*system function*. ↑Unterprogramm im ↑Betriebssystem. Ein solches ↑Programm implementiert beispielsweise die durch einen ↑Systemaufruf vom ↑Maschinenprogramm angeforderte Operation. Im Falle von ↑UNIX ist für jede mit `man(1)` unter Abschnitt 2 „*System calls*“ gelistete Funktion wenigstens ein Unterprogramm des Betriebssystems verknüpft. Darüberhinaus enthält ein Betriebssystem Unterprogramme, die nicht als Folge von Systemaufrufen zur Ausführung kommen, also ihren Ursprung in einer ↑Aktion des Maschinenprogramms haben, sondern bei denen der Aufruf originär von der Hardware ausgelöst wird, nämlich bei einer ↑Unterbrechungsanforderung oder allgemein aufgrund einer ↑Ausnahmesituation. Ferner kann ein im Betriebssystem autonom stattfindender ↑Prozess, eventuell sogar mehrere davon, als ↑Dämon Anlass für solche Aufrufe sein.

Systemkernfaden (en.) ↑*kernel thread*. Bezeichnung für einen ↑Faden der ↑Betriebssystemebene (↑*kernel-level thread*), der von sich aus immer im ↑Systembereich bleibt, im ↑Betriebssystemkern residiert und ausschließlich dessen Kode ausführt. Gegensatz zu ↑Anwendungsfaden.
Ein Faden im ↑Betriebssystem, der durch eine eigene ↑Prozessinkarnation im Betriebssystemkern implementiert ist. Sämtliche für solch einen Faden erforderlichen Betriebsmittel sowie für seine Verwaltung nötigen Funktionen stellt das Betriebssystem. Dies betrifft den ↑Laufzeitkontext des Fadens und die Funktionen zur ↑Ablaufplanung, ↑Einlastung und ↑Synchronisation: All diese Funktionen, insbesondere auch der ↑Prozesswechsel, verlaufen unter Kontrolle des Betriebssystemkerns.
Ein solcher Faden kann, je nach Zweckbestimmung, einen Faden der ↑Benutzerebene tragen, den er dann auf Betriebssystemebene repräsentiert und implementiert — gegebenenfalls kann er damit auch indirekt auf ↑Maschinenprogrammebene wirken, das heißt, als Abbild in Form eines gewöhnlichen Kontrollflusses der Benutzerebene (↑*user-level thread*) ebenda laufen und in entsprechender Funktion einen oder mehrere Anwendungsfäden erzeugen. Letztere Art erfordert dann einen ↑Systemaufruf, wenn innerhalb des betreffenden ↑Maschinenprogramms, durch das dieses Kontrollflussabbild definiert ist, vom aktuellen zu einem anderen Faden umgeschaltet werden soll: setzt dieser Kontrollfluss einen Systemaufruf ab, läuft er als den ihn tragenden Betriebssystemkernfaden (↑*kernel-level thread*) weiter.
In dem Fall ist jeder dieser Fäden aus Sichtweise des Maschinenprogramms vom Bautyp her

ein ↑leichtgewichtiger Prozess, da zwar vergleichsweise aufwendige Systemaufrufe zu seiner Kontrolle erforderlich sind, jedoch bei einem Fadenwechsel im selben Maschinenprogramm derselbe gegebenenfalls unter ↑Speicherschutz stehende ↑Adressraum aktiv bleibt. Innerhalb des Betriebssystems ist ein solcher Faden jedoch immer vom Bautyp her ein ↑federgewichtiger Prozess. Dies gilt insbesondere auch für solche Ausprägungen von Betriebssystemfäden, die (definiert durch ein Maschinenprogramm) auf der Benutzerebene wirken sollen.
Für das Betriebssystem ist ein solcher Faden, auch wenn dieser auf Maschinenprogrammebene wirkt, ein ↑Objekt erster Klasse, das heißt, der durch den Faden konkretisierte ↑Prozess ist insbesondere auch dem Betriebssystemkern bekannt. Eine vom Maschinenprogramm aus beanspruchte ↑Systemfunktion läuft damit immer im Namen dieses Fadens ab. Wird dieser dann durch die Systemfunktion blockiert (↑Prozesszustand), kann der Betriebssystemkern von selbst zu einen anderen Faden dieser Art im selben Maschinenprogramm umschalten.

Systemkonsole (en.) ↑*system console*. Ausrüstung, mit der die manuelle Steuerung und Überwachung von Vorgängen in einem ↑Rechner möglich ist (↑*operator panel*). Anfänglich ein aus vielen Schaltern und verschiedenen Lampenfeldern bestehendes Gerät, über das neben der Eingabe von Kommandos zum Starten, Laden, Anhalten, Fortsetzen und Herunterfahren des Rechners auch bitweise Inhalte der ↑Prozessorregister und vom ↑Hauptspeicher angezeigt und verändert werden konnten sowie allgemein der Systemzustand abgerufen wurde. Zwischenzeitlich auch ergänzt um ein schreibmaschinenähnliches ↑Peripheriegerät (↑TTY) mit ↑Tabellierpapier oder eine spezielle ↑Dialogstation, worüber der Rechner seine Anweisungen in Form einer ↑Kommandozeile erhielt. Seit der ↑PC Einzug gehalten hat auch Inbegriff für die Kombination aus Datensicht- und -eingabegerät (↑*terminal*), über das der Rechner gesteuert und überwacht wird.

Systemlast (en.) ↑*load*. Allgemein die Bezeichnung für die momentan oder zu einem gegebenen Zeitpunkt im ↑Rechensystem herrschende ↑Arbeitslast, hervorgerufen durch laufende oder auf bestimmte ↑Ereignisse wartende ↑Prozesse.

Systemprogramm (en.) ↑*system program*. Bezeichnung für ein ↑Programm oder ↑Unterprogramm der ↑Systemsoftware. Als Unterprogramm entstammt es typischerweise der ↑Programmbibliotheken des zugrunde liegenden Systems.

Systemprogrammiersprache (en.) ↑*system programming language*. Bezeichnung für eine Hochsprache zur ↑Systemprogrammierung. Eine formale Programmiersprache, mit deren Sprachkonstrukte ein ↑Programm zur gezielten Kontrolle der Hardware (↑CPU, ↑Peripherie, ↑Speicher) ausgedrückt werden kann. Zweck einer solchen Programmiersprache ist bewusst nicht die Abstraktion von der Hardware, sondern die problemorientierte und schon auf höherer ↑Abstraktionsebene stattfindende Formulierung von Algorithmen und Datenstrukturen bei direkter Durchgriffsmöglichkeit auf die Hardware (↑ISA) in funktionaler (↑Maschinenbefehl) und nichtfunktionaler (↑Programmiermodell, ↑reale Adresse, ↑Ausrichtung) Hinsicht. Eine ↑Assembliersprache zählt für gewöhnlich nicht in diese Kategorie, da sie ausschließlich — dafür aber allumfassend — lediglich den Durchgriff auf die Hardware ermöglicht. Von den höheren Programmiersprachen ist beispielsweise ↑C oder ↑C++ dieser Kategorie zuzurechnen, obgleich in einigen Fällen (↑Ausnahmebehandlung, ↑Prozesswechsel, ↑Synchronisation) mangels geeigneter Sprachkonstrukte vereinzelt auf Assembliersprache zurückgregriffen werden muss. Eine Hilfskonstruktion (*workaround*) dabei ist oftmals, Assembliersprachenanweisungen entweder inzeilig (*inline*) an den benötigten Stellen im Hochsprachenprogramm zu kodieren oder als getrennt zu übersetzendes ↑Unterprogramm dem Hochsprachenprogramm zum Aufruf beiseitezustellen. Sind die hardwarenahen Operationen ausschließlich durch (in Assembliersprache formulierte, getrennt übersetzte) Unterprogramme zugänglich, handelt es sich bei der Programmiersprache, in der das diese Unterprogramme aufrufen müssende ↑Hauptprogramm formuliert ist, im Allgemeinen nicht um eine Hochsprache zur Systemprogrammierung.

Systemprogrammierung (en.) ↑*system programming*. Programmierung von ↑Systemsoftware. Diese Software ist entweder Teil von einem ↑Betriebssystem oder muss in engem Zusammenspiel mit dem Betriebssystem und der Hardware (↑CPU, ↑Peripherie) funktionieren. Zur Formulierung der Systemsoftware kommt für gewöhnlich eine ↑Systemprogrammiersprache zum Einsatz. Gegenstand eines diesbezüglichen Lehrgebiets ist der Aufbau, die Struktur und die Funktionsweise von Software, die als ↑Programm hilft, ein ↑Rechensystem für einen bestimmten Anwendungszweck zu betreiben. Dabei kann dieses Programm für sich allein genommen durchaus ziemlich wirkungslos sein: es kommt möglicherweise erst dann zur Geltung, wenn es als ↑Unterprogramm (Systemsoftware) in ein ↑Hauptprogramm (Anwendungssoftware) eingebunden ist und von dort aufgerufen werden muss. Typisches Beispiel dafür ist eine ↑Programmbibliothek, deren Softwarebestände die Schnittstelle zu einem Betriebssystem darstellen.

Systemprozess (en.) ↑*system process*. Bezeichnung für einen ↑Prozess, der durch ein ↑Systemprogramm definiert ist. Je nach Art und Auslegung dieses ↑Programms findet der Prozess entweder auf der ↑Benutzerebene oder der ↑Systemebene statt. Letzteres ist für gewöhnlich der Fall, wenn das Programm Teil des ↑Betriebssystems ist.

Systemsoftware (en.) ↑*system software*. Gesamtheit von in Software vorliegender Programme, die eine ↑Rechenanlage betriebsbereit machen (in Anlehnung an den Duden). Dazu zählt insbesondere das ↑Betriebssystem, aber auch die mit einem Betriebssystem typischerweise ausgelieferte ↑Programmbibliothek ist hier eingeschlossen. Typisches Beispiel für letzteres ist die ↑libc, die für gewöhnlich die Programmierschnittstelle zu einem Betriebssystem (z.B. ↑UNIX) bereitstellt. Allgemein wird darunter all jene Software zusammengefasst, die die Grundlage für die Ausführung von Anwendungsprogrammen legt.

Systemsteuerung (en.) ↑*system control*. Gesamtheit der zur ↑Steuerung der Abläufe in einem ↑Rechensystem erforderlichen technischen Bestandteile in Bezug auf einen bestimmten ↑Arbeitsmodus.

Systemzeit (en.) ↑*system time*. Bezeichnung für die Zeitspanne, die ein ↑Prozess im ↑Betriebssystem (↑*system mode*) verbracht hat. Ein über die gesamte Lebenszeit eines Prozesses akkumulierter Wert. Gegensatz zur ↑Benutzerzeit.

Tabellenlauf (en.) ↑*table walk*. Durchgang in einem Verlauf zum Auffinden eines bestimmten Eintrags in einer als Tabelle (↑*array*) organisierten Datenstruktur. Ein solcher Durchgang kann im Zuge der ↑Adressabbildung erforderlich sein, um beispielsweise die eine ↑logische Adresse abbildende ↑Seite in einer ↑Seitentabelle zu finden (*page table walk*). Ein anderes typisches Beispiel ist die Suche nach dem zu einen bestimmten Merkmal (z.B. ↑PID, ↑Prozesszustand oder erwartetes ↑Ereignis) passenden ↑Prozesskontrollblock in der ↑Prozesstabelle.

Tabellierpapier (en.) ↑*continuous paper*. Endlosdruckapier, das keine Einzelblätter aufweist, sondern eine lange Bahn von (bis zu 2000 Stück) durch Perforation voneinander getrennter Blätter bildet. Ursprünglich für eine *Tabelliermaschine* verwendetes Ausgabemedium.

table walk (dt.) ↑Tabellenlauf.

Taktentzug (en.) ↑*cycle stealing*. Verfahren zum ↑Speicherdirektzugriff. Die Steuereinheit (↑DMA *controller*) transferiert einen Block von ↑Daten in Einzelschritten, für die sie jeweils den ↑Systembus nur kurzzeitig belegt: einen Buszyklus „unbemerkt“ in ihren Besitz bringt (↑*cycle stealing*). Zwischen den Einzelschritten hat die CPU Zugriff auf den Systembus; jedoch wird ihre Leistung während des laufenden Transfers beeinträchtigt, wenn sie auf den ↑Bus zugreifen muss.

Taktfrequenz (en.) ↑*clock rate*, ↑*clock speed*. Geschwindigkeit, mit der ↑Daten in der ↑CPU verarbeitet werden können; auch Taktung oder Taktrate von einem ↑Prozessor. Frequenz, mit

der die CPU ihre Steuerbefehle erteilen kann. Angabe in ↑Hertz und zumeist mit einem Präfix versehen, der eine ganzzahlige Vielfache von 10^3 für diese Einheit angibt: Kilo-, Mega-, Gigahertz.

TAS Abkürzung für (en.) ↑*test and set*; Bezeichnung für eine ↑atomare Operation mit einen Operanden: `bool TAS(void *)`. Der Operand ist die ↑Adresse einer ↑Speicherzelle, deren Inhalt im ersten Schritt gelesen und im zweiten Schritt mit dem Wert 1 (`true`) definiert wird. Der gelesene Wert (d.h., `true` oder `false`) ist das Ergebnis dieser Operation:

```
bool TAS(void *ref) {
    bool data;                                  /* auxiliary variable */
    atomic {                                 /* enforce indivisibility */
        data = *(bool *)ref;                     /* read logical value */
        *ref = true;                            /* write logical value */
    }
    return data;                        /* deliver previous logical value */
}
```

Mit `__sync_lock_test_and_set(ref, 1)` definiert ↑GCC für diese Operation eine Standardfunktion (*atomic built-in*), die als ↑atomarer Befehl ausgelegt ist und beispielsweise für ↑x86 als `xchg` erscheint (s. auch ↑FAS).

task (dt.) ↑Aufgabe.

tatsächlicher Parameter (en.) ↑*actual parameter*. Argument, das einem ↑Unterprogramm beim Aufruf tatsächlich übergeben wird. Ein dazu korrespondierender ↑formaler Parameter bestimmt den Typ des Wertes, der übergeben werden kann.

TCP Abkürzung für (en.) *Transmission Control Protocol*, seit 1974. Ermöglicht verbindungsorientierte, zuverlässige, gesicherte und geschützte Kommunikation von Nachrichtenströmen variabler Länge oberhalb von ↑IP. Zusatzfunktionen sind (1) Multiplexen der transferierten ↑Daten, (2) Handhabung und Verpackung der Daten als Nachrichten, (3) Transfer dieser Nachrichten, (4) Bereitstellung einer gewissen Dienstgüte und von Zuverlässigkeit sowie (5) Flusskontrolle und Stauvermeidung.

Team (en.) ↑*team*. Gruppe von Vorgängen an einer gemeinsamen ↑Aufgabe (in Anlehnung an den Duden). Logisch entspricht jeder Vorgang einem ↑Prozess, der physisch als ↑Faden im ↑Maschinenprogramm in Erscheinung tritt und im ↑Betriebssystem durch eine ↑Ablaufplanung kontrolliert wird. Wesentliches Merkmal ist die Repräsentation als ↑nichtsequentielles Programm, das die Basis für den gemeinsamen geschützten ↑Adressraum legt, in dem die Fäden operieren und sich dabei auch sonst *implizit* gemeinsame ↑Ressourcen teilen; ein Kollektiv von logisch verbundenen Prozessen. Unwesentlich ist die Repräsentation des Fadens im Betriebssystem, solange sichergestellt ist, dass ein Faden in einer sich für ihn abzeichnenden Entwicklung zuvorkommend (*präemptiv*) zur Ausführung gelangen kann. Letzteres bedeutet keinesfalls, den Faden als ↑Systemkernfaden implementieren zu müssen. Andere Formen sind ebenso möglich, beispielsweise eine ↑Fortsetzung. Ein mit ↑Thoth eingeführtes Prozesskonzept.

team (dt.) Arbeitsgemeinschaft, Arbeitsgruppe, ↑Team.

Teilhaberbetrieb (en.) ↑*transaction mode*. Bezeichnung für eine ↑Betriebsart, bei der ein ↑Dialogprozess über mehr als eine ↑Dialogstation den Rechnerbetrieb führt. Für gewöhnlich bietet die Dialogstation einem Menschen den Zugang zum ↑Rechensystem. Im Gegensatz zum ↑Teilnehmerbetrieb erfüllt der Dialogprozess typischerweise nur einen bestimmten ↑Dienst, jedoch können mehrere solcher Prozesse verschiedener Dienste zugleich bereitstehen. Der Dialogprozess besteht über die Dauer einer ↑Sitzung hinweg, er führt mehrere Sitzungen gegebenenfalls verschiedener Teilhaber nacheinander.

Teilhabersystem Bezeichnung von einem ↑Rechensystem für ↑Teilhaberbetrieb.

Teilinterpretation siehe ↑partielle Interpretation.

Teilnehmerbetrieb (en.) ↑*time sharing*. Bezeichnung für eine ↑Betriebsart, bei der wenigstens ein ↑Dialogprozess pro ↑Dialogstation den Rechnerbetrieb führt. Für gewöhnlich bietet die Dialogstation einem Menschen den Zugang zum ↑Rechensystem. Im Gegensatz zum ↑Teilhaberbetrieb erfüllt der Dialogprozess keinen bestimmten ↑Dienst, sondern bildet lediglich die „Außenhaut“ (↑*shell*) des Rechensystems, um einen ↑Kommandointerpreter mit Anweisungen zu versorgen. Der Dialogprozess besteht nur für die Dauer einer ↑Sitzung.

Teilnehmersystem Bezeichnung von einem ↑Rechensystem für ↑Teilnehmerbetrieb.

Teilvirtualisierung siehe ↑partielle Virtualisierung.

telegraph code (dt.) ↑Fernschreibkode.

teletypewriter (dt.) ↑Fernschreiber, Abkürzung ↑TTY.

Termin (en.) ↑*deadline*. Eine Zeitbeschränkung (*time constraint*). Ein festgelegter absoluter (d_i) oder relativer Zeitpunkt (D_i), bis zu dem oder an dem etwas geschehen soll (in Anlehnung an den Duden). Für einen absoluten Termin gilt:

$$d_i = r_i + D_i,$$

mit r_i gleich der ↑Bereitzeit (einer ↑Aufgabe) von ↑Prozess P_i.

terminal (dt.) Datenstation, Endgerät; ↑Dialogstation.

Termineinhaltung Befolgung oder Grad der Erfüllung eines vorgegebenen ↑Termins, zu dem eine bestimmte ↑Aufgabe fällig ist (↑*timeliness*). Erfordert die Vereinbarung eines festgelegten (absoluten oder relativen) Zeitpunkts, bis zu dem ein Berechnungsergebnis vorliegen soll oder muss. Der Termin ist typischerweise abgestuft klassifiziert wie folgt:

weich (*soft*), auch schwach, wenn die Terminverletzung tolerierbar ist, mit jeder weiteren Verzögerung das Berechnungsergebnis aber an Wert verliert;

fest (*firm*), wenn die Terminverletzung tolerierbar ist, das Berechnungsergebnis aber keinen Wert mehr hat und die Aufgabe daher abgebrochen werden muss;

hart (*hard*), auch strikt, wenn die Terminverletzung wegen sonst drohender Gefahr für Leib und Leben oder hohem Risiko für wirtschaftlichen Verlust nicht tolerierbar ist, eine ↑Ausnahmesituation, auf die die ↑Anwendung rechtzeitig reagieren können muss.

Der jeweilige Grad ist charakteristisches Merkmal für ein ↑Maschinenprogramm, das unter ↑Echtzeit ablaufen muss und stellt damit bestimmte funktionale (↑Planung) und nichtfunktionale (↑Hintergrundrauschen) Anforderungen, die von einem ↑Betriebssystem zu berücksichtigen sind.

Tertiärspeicher (en.) ↑*tertiary storage*. Zweite Stufe ↑Massenspeicher. Klassifikation für das ↑Archiv; „drittrangiger Speicher“, der zwar über eine enorm große Kapazität verfügt, dafür jedoch auch eine sehr hohe ↑Zugriffszeit mit sich bringt.

tertiary storage (dt.) ↑Tertiärspeicher.

test and set (dt.) testen und einstellen; Abkürzung ↑TAS.

Text (en.) ↑*text*. Im Wortlaut festgelegte, inhaltlich zusammenhängende Folge von Anweisungen (in Anlehnung an den Duden); Fassung von Befehlen in einem ↑Programm. Oft auch als Kode (*code*) bezeichnet, der nach Übersetzung des Programms letztlich in Form von ↑Maschinenkode vorliegt und damit die von einem ↑Prozessor auszuführenden Befehle, nicht jedoch die zu verarbeitenden ↑Daten, beschreibt.

text (dt.) ↑Text.

text segment (dt.) ↑Textsegment.

Textsegment (en.) ↑*text segment*. Bezeichnung für ein ↑Segment, das den ↑Text eines ↑Programms speichert. Je nach ↑Betriebsart ist ein solches Segment gegebenenfalls nicht mehr schreibbar, sobald eine ↑Prozessinkarnation des betreffenden Programms durch das ↑Betriebssystem eingerichtet wurde. Auch ↑selbstmodifizierender Kode wäre dann in solch einem Segment nicht mehr handhabbar. Eventuell ist das Segment auch nicht les- oder ausführbar durch „fremde“ ↑Prozesse, das heißt, Prozesse, die nicht der Prozessinkarnation entsprechen, die das Betriebssystem für das betreffende Programm eingerichtet hat.

Thoth Echtzeitbetriebssystem, University of Waterloo, Ontario, Kanada. Erste Installation im Jahr 1976 (Data General Nova, Texas Instruments 990). Jeder ↑Prozess kann einer Gruppe (↑*team*) von Prozessen oder ↑Prozessexemplaren angehören, die sich einen gemeinsamen ↑Adressraum und eine gemeinsame ↑Freispeicherliste teilen (↑*light-weight process*). Prozesse in derselben Gruppe können ↑Daten gemeinsam nutzen. Prozesse verschiedener Gruppen können dies nicht (↑*address-space isolation*), jedoch durch Botschaftenaustausch (↑*message passing*) kommunizieren. Diese grundlegenden Konzepte wurden später von QNX, V und insbesondere ↑Mach übernommen und weiterentwickelt.

thrashing (dt.) ↑Flattern, Überlastung.

thread (dt.) ↑Faden.

thread blocking anomaly (dt.) Anomalie bei Fadenlahmlegung. Bezeichnung für eine bestimmte Abnormität eines als ↑nichtsequentieller Prozess ablaufenden ↑Maschinenprogramms, die sich darin äußert, dass wenigstens ein ↑Anwendungsfaden eben dieses Programms unerwartet nicht weiter ausgeführt werden kann. Für gewöhnlich ist die Ursache ein ↑Systemaufruf, der zur Blockierung des ↑Systemkernfadens führt, unter dessen Kontrolle die betreffenden Anwendungsfäden dieses Maschinenprogramms ablaufen. Allerdings kann dieser Effekt auch im Zuge der ↑Abfangung oder ↑Unterbrechung eines Anwendungsfadens auftreten. Wird als Folge dieser ↑Ausnahmen beziehungsweise des Systemaufrufs der untergelagerte Systemkernfaden durch den ↑Planer zur Prozessorabgabe gezwungen, ist allen durch diesen Faden letztlich repräsentierten Benutzerfäden ebenfalls kein Prozessor mehr zugeteilt. Erst die erneute ↑Einplanung dieses Systemkernfadens wird dazu führen können, dass die von ihm kontrollierten Anwendungsfäden auf ↑Maschinenprogrammebene den Prozessor erneut erhalten, um dann ihre Ausführung wieder aufzunehmen. Bleibt die Einplanung des Systemkernfadens aus, etwa weil seine weitere Ausführung bedingt durch einen ↑Fehler abgebrochen werden musste, verwaisen die mit ihm verknüpften Benutzerfäden, die dann jedoch brachliegend weiter im Maschinenprogramm existieren können.

threat (dt.) ↑Gefahr.

throttling (dt.) ↑Drosseln, Drosselung.

through-put time (dt.) ↑Durchlaufzeit.

throughput (dt.) ↑Durchsatz.

ticket lock (dt.) Kartenschloss, ↑Ticketsperre.

ticket spinlock siehe ↑*ticket lock*.

Ticketsperre (en.) ↑*ticket lock*. Bezeichnung für eine Sperre (*lock*), bei der ein ↑gekoppelter Prozess in einem zeitlich ausgewogenen, fairen und endlichen Verhältnis zu anderen wettstreitigen Prozessen stehend eine ↑Konkurrenzsituation überwindet; ↑wechselseitiger Ausschluss. Die Sperre ist gültig für alle Prozesse, die, sinnbildlich dargestellt, nicht im Besitz einer Karte

(*ticket*) mit einer bestimmten *Sequenznummer* sind. Es kann nur genau einen Prozess geben, der diese Karte entweder bereits besitzt oder noch ziehen wird. Durchlass an der Sperre wird gewährt in Reihenfolge der ↑Ankunftszeit eines jeweiligen Prozesses (↑FCFS). Im Gegensatz zur (gewöhnlichen) ↑Umlaufsperre ist ↑Gerechtigkeit unter wettstreitigen Prozessen zugesichert und damit auch deren ↑Fortschrittsgarantie an der Sperre gegeben.
Das Verfahren folgt dem Prinzip des ↑Schlossalgorithmus und dient damit der ↑Synchronisation von Prozessen auf einem ↑Multiprozessor oder ↑Mehrkernprozessor. Typischerweise wird mit dieser Technik ein ↑kritischer Abschnitt abgesichert. Vorlage für das Verfahren ist der sogenannte Bäckerladenalgorithmus (*bakery algorithm*, Lamport, 1974) — der allerdings ohne ↑atomare Operationen auskommt, stattdessen auf ↑gemeinsame Variablen ausschließlich durch gewöhnliche Lesebefehle zugreift und schreibende Zugriffe nur auf die privaten Variablen der beteiligten Prozesse durchführt.
Entlehnt einer Kundenverkehrssteuerung bestehend aus Wartemarkenspender und Personenaufrufanlage. Dementsprechend sieht das ↑Eintrittsprotokoll für den kritischen Abschnitt vor, dass die Prozesse einem „Wartemarkenspender“ zunächst eine „Wartemarke“ mit fortlaufender Nummer entnehmen müssen. Diese Marke ist das Ticket für einen Prozess, um den kritischen Abschnitt passieren zu können und die auf der Marke notierte *Wartenummer* bestimmt, wann dieser Prozess passieren darf. Die wettstreitigen Prozesse betreiben schließlich ↑geschäftiges Warten, bis eine „Aufrufanlage“ die Wartenummer desjenigen anzeigt, dem als nächstes Eintritt in den kritischen Abschnitt gewährt wird. Der betreffende Prozess ist dadurch aufgefordert, den kritischen Abschnitt zu durchlaufen und schließlich das ↑Austrittsprotokoll zu befolgen. Als Funktion des Austrittsprotokolls wird nicht nur der kritische Abschnitt freigegeben, sondern gleichfalls die Wartenummer des als nächstes zu bedienenden Prozesses angezeigt.
Indem die an derselben Sperre aufgelaufenen, wettstreitigen (gekoppelten) Prozesse die Anzeige ihrer jeweiligen Sequenznummer abwarten, tun sie dies alle implizit mit verschiedenen ↑Ruhezeiten. Für die ungefähre Ruhezeit ϵ_i eines dieser Prozesse P_i, $1 < i \leq n$, mit n gleich der Anzahl der in dem Moment wettstreitigen Prozesse, gilt dann:

$$\epsilon_i = i * \mathrm{ET}(\textit{critical section}),$$

wobei ET ein Maß liefert für die zu erwartende ↑Ausführungszeit des kritischen Abschnitts, den jeder dieser Prozesse betreten möchte. Wenn einmal davon ausgegangen wird, dass ET für jeden Prozess gleich ist, dann stehen die Ruhezeiten in einem ausgewogenen Verhältnis zur Anzahl der an derselben Sperre aufgelaufenen Prozesse, sie steigen linear an, je mehr Prozesse auflaufen (↑*proportional backoff*). Lässt der kritische Abschnitt allerdings verschiedene Programmpfade zu, dann muss ET schon der ↑WCET dieses Abschnitts entsprechen, um ausgewogene Ruhezeiten zu erhalten. Dies setzt dann weiter voraus, dass die Ausführung des kritischen Abschnitts entweder nicht durch ↑Unterbrechungen verzögert werden kann oder in virtueller Zeit geschieht.

Exkurs Die Entnahme der Wartemarke mit fortlaufender Sequenznummer lässt sich einfach durch eine Zähloperation als Teil des Entrittsprotokolls nachbilden. Entsprechend gestaltet sich die Anzeige der Wartemarke des als nächstes zu bedienenden Prozesses, die sich einfach durch eine Zähloperation als Teil des Austrittsprotokolls nachbilden lässt. Pro Sperre sind damit zwei ↑Variablen erforderlich, die sich durch einen strukturierten ↑Datentypen wie folgt repräsentieren lassen:

```
typedef struct lock {                                  /* ticket lock */
    unsigned next;                     /* number that will be served next */
    volatile unsigned this;              /* number being currently served */
} lock_t;
```

Die Zähloperation zur Entnahme der Wartemarke ist eine ↑Aktion, die potentiell von mehreren Prozessen zugleich durchgeführt wird. Dies ist genau in einer Wettstreitsituation der Fall, wenn nämlich mehrere Prozesse zugleich Eintritt in denselben kritischen Abschnitt erhalten wollen. Um sicherzustellen, dass auch in diesem Fall ein Prozess eine Wartemarke

mit eindeutiger Sequenznummer (`next`) erhält, ist die Zähloperation als ↑atomarer Befehl implementiert (↑FAA). Demgegenüber wird pro Sperre die Zähloperation zur Anzeige der Wartemarke des als nächstes zu bedienenden Prozesses immer nur von einem Prozess durchlaufen, nämlich der Prozess, der sich als einziger in dem durch diese Sperre geschützten kritischen Abschnitt befand. Allerdings wird die betreffende Variable (`this`) im Wettstreitfall von wenigstens einem anderen Prozess gleichzeitig gelesen, weshalb sicherzustellen ist, dass jeder dieser Prozesse immer den aktuell gespeicherten Wert (↑*shared memory*) liest. Eine solche, vor dem Hintergrund ↑gleichzeitiger Prozesse „brisante" (`volatile`) Variable ist daher extra auszuweisen.
Diese beiden eben genannten Aktionen, Generierung einer eindeutigen Sequenznummer und (wiederholtes) Lesen der der Wartemarkenanzeige entsprechenden Variablen, bilden den Kern des Eintrittsprotokolls. Nachfolgend eine dazu passende Implementierungsskizze:

```
void lockup(lock_t *lock) {                          /* ticket spinlock */
    unsigned self = FAA(&lock->next, 1);          /* draw sequence number */
    while (self != lock->this);                      /* wait to be served */
}
```

FAA generiert eine eindeutige Sequenznummer, sofern die Anzahl der an derselben Sperre wettstreitigen Prozesse im Wertebereich der Zählervariablen (`next`) bleibt — was bereits für eine 16-Bit (`short`) Variable sehr wahrscheinlich ist (mindestens $2^{16} = 65536$ wettstreitige Prozesse) und für eine 32-Bit (hier) Variable sicher gilt (mindestens $2^{32} > 4$ Mrd. wettstreitige Prozesse). Die Kopfschleife (`while`) repräsentiert den Warteraum, in dem ein Prozess geduldig darauf wartet, dass die Sequenznummer seiner Wartemarke angezeigt wird, er damit das Eintrittsprotokoll (`lockup`) verlassen und den (durch `lock` geschützten) kritischen Abschnitt betreten kann.
Das Austrittsprotokoll gestaltet sich sehr einfach, es besteht lediglich aus der Zähloperation zum Weiterschalten der Wartemarkenanzeige (`this`):

```
void unlock(lock_t *lock) {
    lock->this += 1;                    /* call on next process to proceed */
}
```

Da für jede dieser Sperren das Austrittsprotokoll immer nur von einem Prozess zugleich durchlaufen wird, weil das Eintrittsprotokoll ja für eine ↑Sequentialisierung möglicher gleichzeitiger Prozesse gesorgt hat, ist eine Absicherung der Zähloperation beispielsweise durch FAA hier nicht erforderlich.
Die gezeigten Lösungen für das Ein- und Austrittsprotokoll sind vergleichsweise effektiv, zumal sie bestechend einfach sind. Allerdings bietet sich für die Wartephase wettstreitiger Prozesse noch eine Verbesserung an, indem dem ↑Prozessor ein Hinweis über die gegenwärtige Situation eines Prozesses gegeben wird:

```
void lockup(lock_t *lock) {                     /* pausing ticket spinlock */
    unsigned self = FAA(&lock->next, 1);          /* draw sequence number */
    while (self != lock->this)                       /* wait to be served */
        ease();                                     /* pause for a moment */
}
```

Der Rumpf der Kopfschleife teilt dem Prozessor mit, dass er sich beruhigen (`ease`) und gegebenenfalls auch in einen speziellen Zustand der Bereithaltung (*stand-by*) eintreten kann (vgl. S. 380). Bei vorab bekannter Ausführungszeit (ET, s.o.) des kritischen Abschnitts könnte die voraussichtliche Dauer der Bereithaltung mitgegeben werden, indem die um einen Zähler verringerte Differenz zwischen eigener und angezeigter Sequenznummer multipliziert mit ET bestimmt wird. Diese Differenz entspricht der Anzahl der wettstreitigen Prozesse, die vor einem selbst noch in den kritischen Abschnitt eintreten werden.
Das hier gezeigte Austrittprotokoll geht davon aus, dass es für eine gegebene Sperre (`lock`) immer nur sequentiell durchlaufen wird. Laufzeitfehler, die etwa zur Folge haben können, mehrere Prozesse zugleich dieses Protokoll bezogen auf ein und dieselbe Sperre durchlaufen

zu lassen, sind damit höchst problematisch. Eine einfache Spezialisierung schafft hier Abhilfe:

1. Nach Erwerb der Sperre, nämlich als letzte Anweisung der Sperroperation (`lockup`), wird der gegenwärtige Prozess, der in den kritischen Abschnitt eintreten darf, in dem ↑Objekt, das die Sperre repräsentiert (`lock`), vermerkt.
2. Vor Abgabe der Sperre, nämlich als erste Anweisung der Entsperroperation (`unlock`), wird überprüft, ob der gegenwärtige Prozess, der die Entsperrung vornimmt, auch der Prozess ist, der die Sperre erwarb. Ist der entsperrende Prozess nicht im Besitz der Sperre, wird eine ↑Ausnahme erhoben (vgl. S. 200).

Eine solche Spezialisierung lässt sich einfach durch ↑Mantelprozeduren erreichen, die die beiden oben gezeigten Funktionen (`lockup`, `unlock`) jeweils mit den entsprechenden Anweisungen ummanteln. Darüberhinaus ist der Datentyp für ↑Exemplare des Sperrobjekts um ein Attribut zu erweitern, das einen Prozess eindeutig identifiziert. Dieses Attribut ist ein ↑Prozesszeiger oder eine ↑Prozessidentifikation, je nachdem, ob die Operationen innerhalb oder außerhalb des ↑Betriebssystemkerns stattfinden).
Grundsätzliches Problem des Verfahrens ist, dass im Wettstreitfall die betroffenen Prozessoren kreiselnd dieselbe gemeinsame Variable (`this`) lesen. Dieses programmierte Verhalten der wettstreitigen Prozesse strapaziert die in ↑Zwischenspeichern benötigten Verfahren zur ↑Speicherkohärenz (↑*cache coherence*) in dem Moment stark, wenn der eine Prozess den kritischen Abschnitt verlässt und dies durch Anzeige der nächsten zu bedienenden Wartemarke anzeigt, die betreffende Variable (`this`) im Ausstrittsprotokoll (`unlock`) also inkrementiert. Die damit verbundene Schreiboperation invalidiert die ↑Zwischenspeicherzeile für alle in dem Wettstreit involvierten Prozessoren, mit der Folge, dass beim nächsten Lesezugriff *jeder* der betroffenen Prozessoren diese Zeile wieder aus dem ↑Arbeitsspeicher lesen muss — obwohl nur einer davon als nächster die Sperre erhalten wird. Wenn jede Anforderung der neuen Zwischenspeicherzeile seriell (von der Hardware) bedient wird, dann steigt die ↑Zugriffszeit auf die Zeile linear mit der Anzahl der wartenden Prozessoren. Dieses Problem löst die ↑gereihte Umlaufsperre.

tile (dt.) Kachel, Fliese; hier im Sinne von ↑*tile processor*.

tile processor (dt.) ↑Kachelprozessor.

time buffer (dt.) ↑Zeitpuffer.

time quantum (dt.) ↑Zeitquantum.

time series (dt.) ↑Zeitreihe.

time series analysis (dt.) ↑Zeitreihenanalyse.

time sharing (dt.) ↑Teilnehmerbetrieb, ↑Zeitteilverfahren.

time slice (dt.) ↑Zeitscheibe.

time slot (dt.) ↑Zeitschlitz.

time stamp (dt.) ↑Zeitstempel.

time unit (dt.) ↑Zeiteinheit.

time-triggered system (dt.) ↑zeitgesteuertes System.

timeliness (dt.) ↑Rechtzeitigkeit.

timeout (dt.) ↑Zeitsperre.

timer (dt.) ↑Zeitgeber.

TLB Abkürzung für (en.) ↑*translation look-aside buffer*.

TLB *thrashing* (dt.) ↑Flattern eines ↑Seitendeskriptors (↑TLB).

Trägerleiterplatte (en.) ↑*motherboard*. Platine, die als Träger für elektronische Bauteile dient und dazu über geeignete mechanische Befestigungen (Sockel) und elektrische Verbindungen verfügt. Typisches Beispiel ist die ↑Hauptplatine in einem ↑Rechner.

transaction (dt.) ↑Transaktion.

transaction mode (dt.) ↑Teilhaberbetrieb.

Transaktion (en.) ↑*transaction*. Eine durch ein ↑Programm definierte ↑Aktionsfolge, die eine *logische Einheit* bildet und als Ganzes entweder gelingt oder scheitert. Gelingt sie, wurde die Berechnung vollständig korrekt durchgeführt und der damit verbundene Datenbestand im konsistenten Zustand hinterlassen. Scheitert sie, bleibt die Aktionsfolge ohne Auswirkung, als wenn sie nie stattgefunden hätte.
Ist das ↑Bezugssystem gebildet durch ein ↑nichtsequentielles Programm, versucht ein durch die darin beschriebene Aktionsfolge ↑gekoppelter Prozess ungehindert die Durchführung der zugehörigen Operationen, auch wenn dies gleichzeitig durch mehrere Prozesse geschehen sollte. Dabei darf die mit dieser Aktionsfolge bezweckten Berechnung eines bestimmten globalen Zustands zunächst jedoch nur lokale Signifikanz für jeden der beteiligten Prozesse haben. Um globale Signifikanz zu erreichen und damit eine Zustandsänderung für alle Prozesse sichtbar zu machen, ist jeder beteiligte Prozess verpflichtet, sich vorher abzusichern (*validate*), um seine lokale Berechnung nach außen bestätigen zu können (*commit*). Die Bestätigung gelingt nur, wenn kein anderer Prozess diese zwischenzeitig bereits erreicht hat. Anderenfalls scheitert die Bestätigung (*abort*) und der jeweils betroffene Prozess wiederholt die Aktionsfolge. Auch als ↑optimistische Nebenläufigkeitssteuerung bezeichnet.

Exkurs Wie nachfolgendes Beispiel demonstriert, ist eine solche Aktionsfolge in einfachster Form Teil einer (fußgesteuerten) Schleife, die es zulässt, dass Prozesse gleichzeitig (↑*simultaneous process*) in ihr stattfinden und operieren können:

```
ERLEDIGE Transaktion:
    WIEDERHOLE
        erstelle die lokale Kopie des Datums an einer globalen Adresse;
        verwende diese Kopie, um ein neues Datum zu berechnen;
        versuche, das neue Datum an der globalen Adresse zu bestätigen;
    SOLANGE die Bestätigung gescheitert ist;
BASTA.
```

Über die an einer globalen ↑Adresse liegende ↑gemeinsame Variable sind die gleichzeitigen Prozesse logisch gekoppelt. Jeder der Prozesse ist in der Lage, das in dieser Variablen enthaltene Datum auszulesen und eine lokale Kopie zu erstellen – zwischengespeichert beispielsweise im eigenen ↑Laufzeitstapel, in ein eigenes ↑Prozessorregister oder in einer ↑Eigenvariablen. Diese Kopie können die betreffenden Prozesse ungehindert, da nunmehr ungekoppelt, verwenden, um die gewünschte Berechnung durchführen. Abschließend versucht jeder dieser Prozesse sein berechnetes Ergebnis in die gemeinsame Variable zu speichern, was jedoch nur gelingt, wenn sich der Zustand dieser Variablen zwischenzeitlich nicht geändert hat. Der erste Prozess, der den unveränderten Zustand erkennt, überträgt sein berechnetes Ergebnis und bestätigt die Zustandsänderung.
Diese Schritte von prüfen, gegebenenfalls übertragen und bestätigen beim Änderungsversuch bilden eine ↑atomare Operation, so dass auch nur ein einziger der gleichzeitigen Prozesse erfolgreich mit der Zustandsänderung der gemeinsamen Variablen sein kann. Für alle anderen gleichzeitigen Prozess wird in dem Moment diese atomare Operation scheitern. In der Folge wiederholen sie ihre Änderungsversuche, indem sie die Aktionsfolge erneut durchlaufen, diesmal dann mit dem neuen Wert der gemeinsamen Variablen. Jeder der betroffenen Prozess durchläuft diese Schleife solange, bis sein Änderungsversuch erfolgreich war.

In diesem Beispiel laufen die gekoppelten Prozesse Gefahr, ihre Änderungsoperation bezüglich der gemeinsamen Variablen überhaupt nicht vollenden zu können und schlimmstenfalls für immer in der Schleife zu verharren: letztlich betreiben sie eine Art ↑geschäftiges Warten auf erfolgreiche Durchführung der Zustandsaktualisierung einer Variablen. Lösungen, die dem damit potentiellen ↑Verhungern eines Prozesses vorbeugen, sind nicht trivial und bereits schnell sehr komplex für Variablen, deren Größe mehr als ein ↑Maschinenwort umfasst. Dieses Muster ist typisch für die ↑nichtblockierende Synchronisation.
Für die erwähnte atomare Operation leisten im einfachsten Fall ↑Maschinenbefehle wie ↑LL/SC oder ↑CAS die erforderliche Funktionalität. Ersterer Ansatz ist für ↑Prozessoren der ↑RISC-Klasse typisch und arbeitet auf Grundlage einer Adressreservierung beim Lesen (↑LL), deren Bestand beim Schreibversuch (↑SC) überprüft wird. Demgegenüber ist letzterer Ansatz bei ↑CISC anzufinden. Hier wird ein Vergleich mit dem ursprünglichen Inhalt der gemeinsamen Variablen durchgeführt und der neue Wert nur geschrieben, wenn der alte Variablenzustand noch gültig ist.

transitive Blockierung (en.) ↑*transitive blocking*. Bezeichnung für die auf einen ↑Prozess indirekt übergehende Blockierung eines anderen Prozesses. Diese Form der Blockierung gilt, wenn ein Prozess P blockiert ist durch einen anderen Prozess P_i, der seinerseits blockiert ist durch Prozess P_j und so weiter. Die Blockierung von P_i durch P_j wirkt auch auf P, sie geht in die Blockierung von P über.
Jeder der beteiligten Prozesse ist blockiert, weil er auf die Zuteilung eines ↑Betriebsmittels wartet. Dabei erwarten die Prozesse die Verfügbarkeit desselben Betriebsmittels oder verschiedener Betriebsmittel. Im letzteren Fall kann in ungünstiger Konstellation der Prozesse eine ↑Kettenblockierung die Folge sein.

transitive blocking (dt.) ↑transitive Blockierung.

translation (dt.) ↑Übersetzung.

translation look-aside buffer (dt.) ↑Übersetzungspuffer.

translator (dt.) ↑Übersetzer.

transparent mode (dt.) ↑Transparentbetrieb.

Transparentbetrieb (en.) ↑*transparent mode*. Verfahren zum ↑Speicherdirektzugriff. Die Steuereinheit (↑DMA *controller*) transferiert einen Block von ↑Daten in Teilen, nämlich wenn die ↑CPU den ↑Systembus nicht benötigt. Die CPU wird in ihrer Leistung nicht beeinträchtigt.

trap (dt.) Falle, ↑Fangstelle; ↑Abfangung, abfangen. Bezeichnung für die bedingt selbstverursachte Aussetzung des normalen Verlaufs eines ↑Prozesses. Dieser Verlauf ist durch das diesen Prozess definierende ↑Programm vorgegeben und wird durch eine ↑Aktion des Prozesses selbst unterbrochen: der Prozess wird bei dieser Aktion durch einen Mechanismus in der ↑CPU abgefangen und der ↑Ausnahmebehandlung durch das ↑Betriebssystem zugeführt.

Trommeladresse (en.) ↑*drum address*. Bezeichnung für eine ↑Adresse im ↑Trommelspeicher.

Trommelmaschine (en.) ↑*drum machine*. ↑Rechner, bei dem der ↑Hauptspeicher ausschließlich aus ↑Trommelspeicher besteht. Solche Rechner wurden bis etwa 1970 hergestellt.

Trommelspeicher (en.) ↑*drum memory*. ↑Peripheriegerät zur elektromagnetischen Aufzeichnung und Wiedergabe von ↑Daten. Die Außenseite eines im Betrieb schnell rotierenden Metallzylinders ist mit ferromagnetischem Material beschichtet, das der eigentlichen Informationsspeicherung dient. Der Zylinder hat mehrere Spuren und pro Spur ist wenigstens ein eigener (oftmals feststehender) Lese-/Schreibkopf vorhanden, was eine mittlere ↑Zugriffszeit (bereits in den 1950er Jahren) um 10 ms ermöglicht. Ursprünglich der ↑Hauptspeicher in einem ↑Rechner (↑*drum machine*), wobei dann der ↑Maschinenbefehl die ↑Trommeladresse des jeweiligen Operanden enthält. Später als Erweiterung für die Zwischenspeicherung von

Daten oder Teilen einer ↑Programmbibliothek, bis in die 1980er Jahre auch zur schnellen ↑Umlagerung von ↑Text und Daten (z.B. in ↑UNIX: `/dev/drum`) in Gebrauch (↑*swapping*). Im Falle von Erweiterungsspeicher besorgt ein ↑Gerätetreiber die Ein- oder Ausgabe mittels ↑Ein-/Ausgaberegister, die nebst Daten auch die Trommeladresse enthalten beziehungsweise dem Gerät übermitteln. Der Gerätetreiber ist für gewöhnlich ein ↑Unterprogramm von einem Steuerprogramm zur Organisation und Überwachung des Rechnerbetriebs (↑*resident monitor*).

TTY Abkürzung für (en.) ↑*teletypewriter*.

Typfehler Folge einer ↑Typverletzung.

Typsicherheit Ausmaß der Verhinderung von ↑Typfehler durch eine Programmiersprache beziehungsweise einen ↑Kompilierer. Die Maßnahmen dazu greifen statisch (bei der ↑Kompilation), dynamisch (zur ↑Laufzeit) oder in Kombination. Je nach Stärke der Typisierung der Programmiersprache wird der Kompilierer mehr oder weniger viel Typkonflikte unbeanstandet durchlassen, die dann jedoch durch generierte Anweisungen zur Typkontrolle zur Laufzeit abgefangen werden und als ↑Ausnahmesituation enden.

Typverletzung Unstimmigkeit bei Operationen auf verschiedenen Datentypen. Beispielsweise eine Ganzzahl (`int`) als Zeiger auf ein Zeichen (`char*`) oder als Universalzeiger (`void*`) behandeln — dadurch eine beliebige ↑Adresse konstruieren und (logisch) unautorisiert auf den ↑Arbeitsspeicher zugreifen können.

UDP Abkürzung für (en.) *User Datagram Protocol*, seit 1980. Ermöglicht verbindungslose, unzuverlässige, ungesicherte und ungeschützte Kommunikation in einem ↑Netzwerk. Das Protokoll verwendet Prüfsummen zur Überprüfung der Integrität transferierter ↑Daten und Anschlussnummern (*port number*), um darüber verschiedene Funktionen im Quell- und Zielknoten eines Datagramms zu adressieren.

Überlagerung (en.) ↑*overlay*. Programmteil (↑Text oder ↑Daten), dessen Platz im ↑Hauptspeicher wechselweise über die Zeit von mehreren solcher Teile gemeinsam belegt wird.

Überlagerungsspeicher (en.) ↑*overlay memory*. Bereich in einem peripheren ↑Speicher, in dem ↑Überlagerung eines Programms abgelegt sind.

überlappte Ein-/Ausgabe (en.) ↑*overlapped I/O*. ↑Betriebsart, bei der Ein-/Ausgabe die damit im Zusammenhang stehende oder eine beliebige andere Berechnung zeitlich überlagert. Dazu setzt ein ↑Prozess zunächst eine Ein-/Ausgabeoperation an das ↑Betriebssystem ab. Im weiteren Verlauf löst diese Operation, effektiv verursacht durch einen ↑Gerätetreiber, einen ↑Ein-/Ausgabestoß bei einem ↑Peripheriegerät aus. Dieser Stoß kann gleichzeitig mit dem jeweils von der ↑CPU generierten ↑Rechenstoß eines Prozesses stattfinden. Wartet der Prozess, der die Ein-/Ausgabe angestoßen hat, nicht auf die Beendigung der dafür ursächlichen Operation, überlagern sich der durch ihn selbst herbeigeführte Rechen- und Ein-/Ausgabestoß. Begeht der Prozess dagegen ↑passives Warten, führt dies für gewöhnlich zum ↑Prozesswechsel und damit dazu, dass sich Rechen- und Ein-/Ausgabestöße, die aus verschiedenen Prozessen resultieren, zeitlich überlagern können. Da die Ein-/Ausgabe unabhängig von dem Prozess stattfindet, der die Operation dazu abgesetzt hat, sendet das betreffende Peripheriegerät eine ↑Unterbrechungsanforderung an die CPU, um dem Betriebssystem die Beendigung der entsprechenden Operation anzuzeigen. Aus demselben Grund erfolgt der Transfer ein- oder auszugebender ↑Daten durch ↑Speicherdirektzugriff.
Allgemein betrachtet tragen diese Maßnahmen wesentlich zur Steigerung von ↑Durchsatz bei, sie bewirken allerdings auch ↑Interferenz: Prozesse werden verzögert oder unterbrochen, die nichts mit der laufenden oder beendeten Ein-/Ausgabeoperation zu tun haben müssen. Wegen der strikt asynchronen Vorgehensweise ist der Zeitpunkt oder die Zeitspanne der Beeinflussung zudem unvorhersehbar für die Prozesse. Je nach Art eines Prozesses ist

diese Eigenschaft mehr oder weniger kritisch. Alle Prozesse, für die eine bestimmte ↑Termineinhaltung gilt, sind davon betroffen. Für gewöhnlich kann diese Beeinflussung seitens der ↑Peripherie nicht unterbunden werden, was insbesondere auf ↑DMA zutrifft: auch wenn die Technik selbst einen Prozess nicht unterbricht, trägt sie (mit Ausnahme von ↑Transparentbetrieb) durch abknapsen von ↑Speicherbandbreite jedoch zu seiner Verzögerung bei, wenn er zugleich auf den ↑Hauptspeicher zugreift. Für Prozesse, die eine solche Beeinflussung nicht tolerieren, muss das Betriebssystem Vorkehrungen treffen, die einen vorhersehbaren Verlauf zusichern. Das kann im Extremfall bedeuten, Unterbrechungen oder DMA phasenweise oder für „sensitive Prozesse“ ganz zu unterbinden.

Übernahmeprüfung (en.) ↑*acceptance test*. Verfahren beim ↑Echtzeitbetrieb, um zu entscheiden, ob eine ↑sporadische Aufgabe zur Bearbeitung durch das ↑Rechensystem zugelassen werden darf. Dabei wird geprüft, ob bis zum ↑Termin der Fertigstellung dieser Aufgabe noch genügend freie Rechenkapazität verfügbar ist: Zur ↑Ankunftszeit der Aufgabe muss die Summe der ↑Schlupfzeiten im Rechensystem bis zum Aufgabentermin größer oder gleich der maximalen Aufgabenbearbeitungszeit (↑WCET) sein. Reicht die Rechenkapazität nicht aus, wird die Bearbeitung dieser Aufgabe zurückgewiesen und eine ↑Ausnahme erhoben.

Übersetzer (en.) ↑*translator*. ↑Dienstprogramm, das die Anweisungen eines in einer bestimmten Sprache formulierten ↑Programms in semantisch äquivalente Anweisungen eines anderen Programms einer gegebenenfalls anderen Sprache umwandelt. Typisch dafür sind ↑Kompilierer, ↑Assemblierer und ↑Interpreter, allerdings nur letztere führen die in dem zu übersetzenden Programm enthaltenen Anweisungen auch direkt aus.

Übersetzung (en.) ↑*translation*. Bezeichnung für die ↑Kompilation oder das ↑Assemblieren von einem ↑Programm. In beiden Fällen erfolgt die Umwandlung eines in Quellsprache (z.B. ↑C oder ↑ASM86) vorliegenden Programms in eine bestimmte Zielsprache (z.B. ↑ASM86 oder ↑Maschinensprache).

Übersetzungseinheit Quellmodul für einen ↑Übersetzer, seine Eingabe.

Übersetzungsprotokoll (en.) ↑*assembler listing*. Aufstellung des vom ↑Assemblierer verarbeiteten und in ↑Assembliersprache formulierten ↑Programms, typischerweise in mehreren blockartig gesetzten (ggf. gleich langen, mehrspaltigen) Rubriken, wobei Anzahl und Bedeutung dieser Rubriken von System zu System durchaus variiert. Für ↑GAS sind folgende Rubriken typisch (zeilenweise, von links nach rechts; vgl. auch S. 23):

1. Verweis auf die Zeilennummer im ↑Quellmodul.
2. Wert des ↑Adresszählers des für diese Zeile gültigen ↑Segments.
3. Aufstellung des generierten ↑Maschinenkodes, in Abhängigkeit vom Übersetzerschalter gegebenenfalls mehrspaltig.
4. Angabe der vom Assemblierer interpretierten beziehungsweise übersetzten Anweisung entsprechend Rubrik 1. Für gewöhnlich steht hier ein ↑Pseudobefehl, eine ↑Assembleranweisung oder eine ↑Sprungmarke.

Möglicherweise zeigt die letzte Rubrik (4.), die den übersetzten Quelltext dokumentiert, alternativ eine Anweisung einer höheren Programmiersprache (z.B. ↑C). In dem Fall wurde das Protokoll direkt vom ↑Kompilierer erstellt. Auch können hier beide Arten symbolisch repräsentierter Anweisungen, nämlich sowohl der Programmier- als auch der durch ↑Kompilation generierten Assembliersprache, angegeben sein.

Übersetzungspuffer (en.) ↑*translation look-aside buffer*. Auch kurz als ↑TLB bezeichnete Funktionseinheit zur schnellen ↑Adressabbildung, integriert in einer ↑MMU oder ↑CPU. Beim Zugriff auf den ↑Arbeitsspeicher ermöglicht diese Einheit mit einem kurzen „Blick zur Seite“ (*look aside*) festzustellen, ob eine von der CPU verwendete ↑logische Adresse direkt in eine ↑reale Adresse übersetzt werden kann. Die für die Übersetzung erforderlichen Hinweise werden entweder von der MMU oder dem ↑Betriebssystem in dem Puffer zwischengespeichert.

UID Abkürzung für (en.) ↑*user ID*.

Umgebungsrauschen (en.) ↑*ambient noise*. Siehe ↑Hintergrundrauschen.

Umlagerung (en.) ↑*swapping*. Übertragung von Speicherinhalten zwischen verschiedenen Ebenen der ↑Speicherhierarchie in einem ↑Rechensystem. Typisch dafür ist etwa die Auslagerung von einem kompletten ↑Prozess vom ↑Vordergrundspeicher in den ↑Hintergrundspeicher beziehungsweise umgekehrt die Einlagerung.

Umlagerungsbereich (en.) ↑*swap area*. Bereich in der ↑Ablage, der vom ↑Betriebssystem für die Sicherungskopie (*backup*) des Inhalts von einem ↑Prozessadressraum vorgesehen ist. ↑Hintergrundspeicher, ohne dem ↑virtueller Speicher für einen ↑Prozess nicht funktioniert. Die Größe dieses Bereichs richtet sich nach der maximalen Anzahl, der zu einem Zeitpunkt vom Betriebssystem unterstützten Prozesse und der maximalen Größe des Adressraums eines jeden dieser Prozesse (↑schwergewichtiger Prozess).
Aus diesem Bereich werden gegebenenfalls (↑*paging*) nur die Anteile von ↑Text und ↑Daten eines Prozesses in den ↑Hauptspeicher kopiert (umgelagert), die für sein effizientes Vorankommen gerade benötigt werden (↑*working set*). Umgekehrt werden bei Verdrängung aus dem Hauptspeicher nur die Anteile in diesen Bereich zurück kopiert (umgelagert), die von dem Prozess seit ihrer Einlagerung zwischenzeitlich verändert worden sind.

Umlagerungsmittel (en.) ↑*swapping means*. Handhabe zur ↑Umlagerung von ↑Text oder ↑Daten zwischen ↑Vordergrundspeicher und ↑Hintergrundspeicher. Die Text-/Datenbestände sind in den Strukturierungselementen des ↑Prozessadressraums zusammengefasst, das heißt, sie liegen entweder in einer ↑Seite oder in einem ↑Segment. Je nach ↑Adressraummodell bilden letztere die Handhabe zur Umlagerung der Text- und Datenbestände. Dabei kann die Umlagerung nur einzelne Seiten (↑*paging*) oder Segmente (↑*segmentation*) betreffen, logisch zusammenhängende Seiten (↑*working set*) in Betracht ziehen oder den gesamten ↑Speicherbereich des Prozessadressraums umfassen (↑*swapping*).

Umlaufsperre (en.) ↑*spinlock*. Bezeichnung für eine Sperre (*lock*), deren Gültigkeit von einem ↑Prozess durch Kreiseln (*spin*) anhaltend überprüft wird; ↑wechselseitiger Ausschluss.
Der Prozess betreibt ↑geschäftiges Warten bis die Sperre aufgehoben ist. Ein auf Grundlage eines ↑Schlossalgorithmus operierendes Verfahren zur ↑Synchronisation von Prozessen auf einem ↑Multiprozessor oder ↑Mehrkernprozessor. Dabei sollte darauf geachtet werden, dass jeder der kreiselnden Prozesse auf einem eigenen ↑Prozessor stattfindet und der die Sperre haltende Prozess keine ↑Verdrängung erfährt (↑*lock holder preemption*). Ansonsten drohen erhebliche Leistungseinbußen, da die kreiselnden Prozesse noch zusätzlich durch die ↑Wartezeit des von seinem Prozessor verdrängten und die Sperre haltenden Prozesses verzögert werden. Typischerweise wird mit dieser Technik ein ↑kritischer Abschnitt abgesichert.
Für gewöhnlich gibt das Verfahren keine ↑Fortschrittsgarantie: die an derselben Sperre wettstreitig agierenden Prozesse laufen Gefahr zu ↑Verhungern (s. auch nachfolgende Beispiele). Verbesserung schaffen Lösungen, die beispielsweise die ↑Ankunftszeit eines jeden Prozesses an der Sperre berücksichtigen und für einen entsprechend geordneten Durchlass sorgen (↑*ticket spinlock*, ↑*queued lock*).

Exkurs Im Grunde genommen ist das Verfahren in logischer Hinsicht sehr einfach umzusetzen — wenn gewisse physikalische Gegebenheiten und Effekte der Hardware einmal außen vor gelassen werden. Folgende Lösung skizziert das Prinzip:

```
void lockup(lock_t *lock) {
    while (*lock);                          /* busy-test lock variable */
    *lock = true;                       /* reset done, set lock variable */
}
```

Der Ansatz nutzt eine ↑Schlossvariable, die den Zustand der Sperre widerspiegelt und dazu

nur Boolesche Werte annimmt (S. 278):

„Sperre aufgehoben“ $\mapsto$ `false`
„Sperre verhängt“ $\mapsto$ `true`

Ein durch diese Schlossvariable ↑gekoppelter Prozess wird beim Abschließen (*lock up*) der als Schloss (*lock*) verstandenen Sperre solange aufgehalten, bis ein anderer (gekoppelter) Prozess die Maßnahme zum Aufschließen (*unlock*) ergriffen hat:

```
void unlock(lock_t *lock) {
    *lock = false;                                  /* reset lock variable */
}
```

So einfach diese Lösung auf den ersten Blick auch erscheinen mag, sie ist falsch, da sie in einer ↑Konkurrenzsituation mehrerer über dieselbe Variable gekoppelter Prozesse dazu führen kann, dass alle konkurrierenden Prozesse die Sperre passieren können. Im Falle von N konkurrienden Prozessen müssen normalerweise $N - 1$ Prozesse an der Sperre aufgehalten werden, nur einer darf die Sperre überwinden. Die gezeigte Lösung, und zwar nur diejenige, um die Sperre zu verhängen (`lockup`), erlaubt es, dass die konkurrierenden Prozesse in eine ↑Wettlaufsituation (*write-read conflict*) geraten.
Angenommen, die konkurrierenden Prozesse schreiten synchron (d.h., Anweisung für Anweisung in `lockup`) voran und die Sperre ist aufgehoben. Dann wird jeder dieser Prozesse die Sperre als aufgehoben (`false`) wahrnehmen, damit die Kopfschleife (`while`) nicht betreten und die Sperre verhängen (`true`), ohne dass dies einer seiner Konkurrenten mitbekommt. Bildet diese Lösung etwa den Kern des ↑Eintrittsprotokolls für einen kritischen Abschnitt, kann die Integrität dieses Abschnitts (im Falle von Konkurrenz) nicht zugesichert werden. Erwähnt sei noch, dass das Voranschreiten der konkurrierenden Prozesse in beinahe „Gleichtakt“ ganz einfach durch ↑Simultanverarbeitung geschehen kann.
Korrekte Lösungen, die den Sperrzustand weiterhin durch eine Schlossvariable als einziges Instrument repräsentieren, müssen sicherstellen, dass die für ein berechtigtes Abschließen benötigte ↑Aktionsfolge zum Abfragen und Setzen der Schlossvariablen als ↑atomarer Befehl implementiert ist. Eine solche Lösung zeigt nachfolgendes Beispiel:

```
void lockup(lock_t *lock) {                  /* test and set lock variable */
    while (TAS(lock));
}
```

Mittels ↑TAS wird die ↑Unteilbarkeit der genannten Aktionsfolge sichergestellt. Die Implementierung dieses ↑Maschinenbefehls sorgt für (scheinbare) ↑Gleichzeitigkeit der beiden relevanten ↑Aktionen zum Lesen — um den Test durchführen zu können — und Schreiben der Schlossvariablen.
Diese Implementierung ist korrekt, aber kann gerade bei hoher Konkurrenz ↑gleichzeitiger Prozesse, die nicht nur über dieselbe Schlossvariable miteinander gekoppelt sein müssen, starke ↑Interferenz als auch Leistungseinbrüche zur Folge haben. Ein Knackpunkt ist die Konstruktion der Kopfschleife, die durch die ↑Atomizität von TAS ein gravierendes Problem hervorruft. Gerade bei hoher Konkurrenz ist der ↑Bus nahezu anhaltend gesperrt für andere Prozesse, da jeder der konkurrierenden Prozesse, während sie alle sehr geschäftigt warten (d.h. „schleifen“), mit jedem TAS den Bus für sich selbst beansprucht (↑*bus-lock burst*). Abhilfe schafft daher eine Schleifenkonstruktion, mit der das eigentliche Warten darauf, dass die Sperre aufgehoben wird, keinen atomaren Befehl benutzt:

```
void lockup(lock_t *lock) {                 /* test & test and set (TATAS) */
    while (*lock || TAS(lock));
}
```

Verlässt ein Prozess die Warteschleife, weil die Schlossvariable durch einen anderen Prozess zurückgesetzt wurde (`unlock`), kann er aber — wie eingangs beschrieben — nicht sicher sein, dass er der einzige (gleichzeitige/gekoppelte) Prozess in dieser Situation ist. Er muss also nach dem Versuch, die Schlossvariable für sich zu setzen, gegebenenfalls erneut in die

Warteschleife eintreten. Dieses korrekte Verhalten wird durch das logische Oder (`||`) in der Abbruchbedingung sichergestellt, wodurch TAS immer nur anderenfalls (*or else*), also wenn die Schlossvariable zurückgesetzt wurde, zur Ausführung kommt.
Ein weiteres Problem ist, dass die gängigen (Hardware-) Implementierungen von TAS immer schreiben (vgl. S. 347), auch wenn dadurch der Wert der Schlossvariablen nicht verändert wird. Zudem sorgt TAS als atomarer Befehl auch immer sofort für Speicherkohärenz. Da für das hier thematisierte Sperrverfahren für gewöhnlich ein ↑speichergekoppelter Multiprozessor die Grundlage bildet, sind die ↑Zwischenspeicher der einzelnen ↑Rechenkerne mit jedem einzelnen TAS auf den aktuellen Stand zu bringen. Dies sorgt für erhöhten Verkehr vor allem auf dem ↑Datenbus, wodurch ↑Bandbreite für die Aktivitäten anderer Prozesse verloren geht — insbesondere ↑nebenläufiger Prozesse. Abhilfe für dieses Problem schafft daher eine Variante, die den Schreibvorgang nur bedingt durchführt:

```
void lockup(lock_t *lock) {            /* test & compare and swap (TACAS) */
    while (*lock || !CAS(lock, false, true));
}
```

Der atomare Befehl ↑CAS trifft eine Fallunterscheidung und schreibt nur, wenn eine bestimmte Bedingung erfüllt ist. Im gegebenen Fall erhält die Schlossvariable (`*lock`) nur dann den neuen Wert `true`, wenn ihr gültiger Wert im Moment der Befehlsausführung dem erwarteten Wert `false` gleicht. Von N konkurrierenden (gleichzeitigen/gekoppelten) Prozessen wird damit nur ein einziger die Schreiboperation bewirken und einen Abgleich der Zwischenspeicher auslösen: im Verglich zu den N Schreiboperationen, die mit TAS ausgelöst werden, verringert sich das Verkehrsaufkommen wesentlich — und Anfälligkeit für Interferenz sinkt.
Mit den bisher betrachteten TA-Φ-AS Lösungsvarianten, mit $\Phi \in \{\mathrm{T}, \mathrm{C}\}$, werden allerdings nur bedingt zu lange Stöße von ↑Bussperren vermieden. Sollte etwa ein mit diesem Verfahren abgesicherter kritischer Abschnitt sehr kurz sein, kann leicht die Situation eintreten, dass keiner oder nur wenige der konkurrierenden Prozesse in der Warteschleife verharren. Dies betrifft insbesondere kritische Abschnitte, deren ↑Ausführungszeiten kaum größer sind als die Zeit, die ein Durchlauf der Warteschleife beansprucht — womit ↑Informatikfolklore mit Vorsicht zu übernehmen ist, nämlich dass geschäftiges Warten dieser Art gerade für kurze kritische Abschnitte geeignet sein soll. Damit solche Stöße gerade unbeteiligte (d.h., nebenläufige) Prozesse nicht zu stark beeinträchtigen, sollten die Intervalle zwischen den einzelnen atomaren Befehlen ausreichend Spielraum für normale (nicht atomare) Befehle lassen. Dies aber bedeutet, dass die konkurrierenden Prozesse eine bestimmte ↑Ruhezeit einlegen und von Aktionen mit globalen Auswirkungen zurückweichen (↑*backoff*) müssen, bevor sie den nächsten atomaren Befehl zur Ausführung bringen.
Die Wahl (a) einer Größe für die Ruhezeit eines um das Setzen der Schlossvariablen konkurrierenden Prozesses und (b) der Stelle in der Implementierung des Schlossalgorithmus, an der die Ruhezeit dann schließlich gelten soll, ist schwieriger als auf den ersten Blick vermutet. Beide Fälle sind stark vom Wettstreitaufkommen durch konkurrierende Prozesse und der jeweils zugrunde liegenden Hardwareplattform abhängig. Die Ruhezeit kann einerseits statisch oder dynamisch bestimmt werden und andererseits nach jedem oder nur dem atomaren Zugriff auf die Schlossvariable greifen. Eine Lösungsvariante, in der nach jedem erfolglosen Schlossvariablenzugriff ein konkurrierender Prozess für eine bestimmte Ruhezeit ϵ innehält, zeigt folgendes Beispiel:

```
void lockup(lock_t *lock) {          /* TACAS with backoff on every access */
    while (*lock || !CAS(lock, false, true))
        ϵ = backoff(ϵ);                                /* pause for a moment */
}
```

Die Anweisung `ϵ = backoff(ϵ)` lässt einerseits einen wettstreitigen Prozess pausieren und ermöglicht andererseits eine Ruhezeitanpassung. Auch nach jedem erfolglosen nicht atomaren Lesezugriff (*spin on read*) auf die Schlossvariable den betreffenden Prozess für eine bestimmte Zeit ruhen zu lassen, reduziert Wettstreit vor allem für Prozessoren, die keine Zwischenspeicher besitzen, das heißt, bei denen jeder Speicherzugriff über eine (zentrale)

↑Speichersteuerung läuft. Die Schlossvariable wird letztlich mit geringerer Frequenz geprüft, was insbesondere auch auf CAS zutrifft. Dagegen ist im Falle von zwischenspeicherbasierten Prozessoren nur eben dieser letzte Aspekt von Bedeutung. Nachfolgende Programmskizze wäre eine Lösungsvariante dafür:

```
void lockup(lock_t *lock) {          /* TACAS with backoff on abortive set */
    while (*lock || (!CAS(lock, false, true) && (ϵ = backoff(ϵ), true)));
}
```

Hier beginnt die Ruhezeit eines Prozesses nach dem Scheitern des CAS, was durch die Konstruktion der Wiederholungsbedingung mit dem logischen Und (`&&`, *and then*) erreicht wird. Der anschließende Ausdruck, bei dessen Auswertung der Prozess dann für eine bestimmte Zeit pausiert (`backoff`), gelingt mit dem letzten ↑Sequenzpunkt immer und führt damit zur Wiederholung der Kopfschleife. Knackpunkt dieser und der vorherigen Lösungsvariante aber ist, die effektive Größe ϵ für die jeweilige Ruhezeit eines Prozesses zu bestimmen.
Ein Ansatz besteht darin, die Ruhezeit (ϵ) *statisch* auszulegen. Ein wettstreitiger Prozess setzt dann für ein festes Zeitintervall aus, bis er den nächsten Versuch zum Setzen der Schlossvariablen unternimmt. Dabei ist es sinnvoll, den verschiedenen wettstreitigen Prozessen auch verschiedene Ruhezeiten zuzuordnen. Anderenfalls würden sie beim nächsten Versuch mit hoher Wahrscheinlichkeit alle wieder zugleich in Wettstreit geraten. In dem Ansatz ist jedem Prozessor eine feste ↑Zeitnische zugewiesen, die durch das Warteverhalten und die Ruhezeiten der wettstreitigen Prozesse einem Prozessor indirekt eingeräumt wird. Dennoch ergibt sich damit kein ↑zeitgesteuertes System, da die Zeitnischen nicht an fest vorher definierten Punkten auf der ↑Echtzeitachse liegen. Indem die Prozesse unterschiedlich lange innehalten, ergibt sich für einem ihrer Prozessoren eine Zeitnische zur kollisionsfreien Nutzung des Busses. Wenige lehre Zeitnischen bedeuten dann noch gute Latenz, wohingegen wenige volle Zeitnischen auf geringen Wettstreit hinweisen. Allgemein ist von einer guten Leistung auszugehen, wenn bei wenigen Zeitnischen nur wenige Prozesse beziehungsweise bei vielen Zeitnischen auch viele Prozesse pausieren.
Der andere Ansatz ist *dynamisch* ausgerichtet und verordnet den wettstreitigen Prozessen variable Verzögerungen. Für gewöhnlich hängt dabei die Ruhezeit (ϵ) eines Prozesses von der Anzahl der sich in der Konkurrenzsituation befindlichen wettstreitigen Prozesse ab. Dadurch ergibt sich die Zeitnische eines Prozessors als Funktion des Kollisionsgrads der an einer bestimmten Schlossvariable gekoppelten Prozesse. Je nach Verfahren wird im Moment der Kollision mit einer festen oder zufälligen Größe für die Ruhezeit gestartet, die sich dann bei weiteren Kollisionen exponentiell für gewöhnlich bis zu einer bestimmten Obergrenze verlängert (↑*exponential backoff*).
Unabhängig davon, ob die Ruhezeiten der Prozesse/Prozessoren statisch oder dynamisch bestimmt werden: beide Verfahren sind ungünstig bei kritischen Abschnitten mit vergleichsweise langen Ausführungszeiten. Gegebenenfalls erhöhen sich die Ruhezeiten an einem kritischen Abschnitt dann übermäßig stark, während die zugehörige Schlossvariable noch gesetzt ist. Hinzu kommt, dass ein Scheitern des atomaren Befehls (TAS oder CAS) nicht zwingend auch ein Anzeichen vieler wettstreitiger Prozesse sein muss. Grundsätzlich müssen die Ruhezeiten begrenzt sein und sollten dann auch verschiedene Endgrößen annehmen. Mit zunehmender Ruhezeit im Moment der Erkenntnis, dass die Schlossvariable noch besetzt ist, senkt sich der ↑Durchsatz an dem betreffenden kritischen Abschnitt.
Alle oben vorgestellten Lösungen sind problematisch, sobald sich unter den wettstreitigen Prozessen wenigstens ein ↑Echtzeitprozess befindet. Grundsätzlich besteht dann die Gefahr der ↑Prioritätsumkehr, für die ↑blockierende Synchronisation, wie hier behandelt, in der Regel höchst anfällig ist und zum ↑Versagen solcher Echtzeitprozesse führen kann. Darüberhinaus ist zu beachten, dass die gestaffelten (statischen/dynamischen) Ruhezeiten letztlich zu einer Reihung der Prozesse führen kann, die im Widerspruch zur ↑Ablaufplanung steht (↑*interference*). Als Folge davon kann sich eine ↑Prioritätsverletzung für die durch ihre Ruhezeit fälschlicherweise nachrangig aufgereihten Echtzeitprozesse ergeben, was ebenfalls zum Scheitern dieser Prozesse führen kann. Die Verletzung der Priorität eines Echtzeitprozesses

lässt sich vermeiden indem die *effektive Ruhezeit* ϵ_i eines Prozesses P_i in Abhängigkeit von der jeweiligen Priorität bestimmt wird:

$$\epsilon_i = \epsilon * \mathrm{PRI}(P_i),$$

mit ϵ als Parameter für die prozessorspezifische Ruhezeit von P_i und PRI der ↑Priorität des angegebenen Prozesses. Je kleiner der Wert von PRI, desto höher die Priorität des betreffenden Prozesses. Handelt es sich bei P_i um einen Echtzeitprozess, dann entspricht PRI der statischen/dynamischen Priorität von P_i. Anderenfalls hat PRI einen Wert, der größer ist als der Wert der zur Zeit niedrigsten Priorität eines Echtzeitprozesses im ↑Bezugssystem, das heißt, Prozesse ohne ↑Echtzeitbedingungen haben die niedrigste Priorität überhaupt.
In logischer Hinsicht ergäbe sich für die wettstreitigen Prozesse schon eine prioritätskonforme Reihung an der Sperre, allerdings ist damit längst nicht einem möglichen Scheitern der Echtzeitprozesse vorgebeugt. Die prozessorspezifische Ruhezeit (ϵ) als Skalierungsfaktor kann immerhin dazu führen, dass ein Echtzeitprozess P_i, sobald er die Sperre passiert und weiter voranschreitet, dennoch nicht mehr den für ihn geltenden ↑Termin D_i erreicht:

$$\sigma_i(a_i) + \epsilon_i > D_i,$$

mit $\sigma_i(a_i)$ gleich der ↑Schlupfzeit zur ↑Ankunftszeit a_i des Prozesses an der Sperre. Echtzeitfähig wäre dieser Ansatz daher nur, wenn sichergestellt werden kann, dass die Ruhezeit eines Echtzeitprozesses stets kleiner oder gleich seiner Schlupfzeit ist. Indem dieser Prozess an der Sperre wartet, nimmt seine Schlupfzeit allerdings nach und nach mit seiner Ruhezeit ab. Daher müssen alle Ruhezeiten eines Echtzeitprozesses schon vorher bekannt sein, um ihn so einplanen zu können, dass er seine ↑Aufgaben auch rechzeitig erledigen kann. Dies erfordert umfangreiches ↑Vorwissen zu allen Prozessen, die zu beliebigen Zeitpunkten miteinander in Wettstreit geraten können — was in einem durch Dynamik geprägten ↑Rechensystem bestenfalls nur unvollständig vorhanden ist. Unter realen Bedingungen ist damit die auf Ruhezeiten basierende *Stauauflösung* wettstreitiger Prozesse untauglich für ↑Echtzeitsysteme.

Umplanung (en.) ↑*rescheduling*. Abänderung eines existierenden Plans, nach dem ein ↑Auftrag einem ↑Prozessor zur Verarbeitung übergeben werden soll. Typischer Fall ist ein ↑Ablaufplan, der zur ↑Laufzeit an geänderten Randbedingungen angepasst werden muss. Dadurch kann ein bereits eingeplanter ↑Prozess nunmehr früher oder später für die ↑Einlastung der ↑CPU in Frage kommen. Voraussetzung dafür jedoch ist die ↑mitlaufende Planung, um überhaupt auf sich dynamisch ergebende, für gewöhnlich auch unvorhersehbare und Prozesse beeinflussende ↑Ereignisse reagieren zu können. Beispiele dafür sind ↑HRRN, ↑SRTF, ↑VRR und ↑FB beziehungsweise ↑MLFQ.
Zu beachten ist, dass umplanende Verfahren nicht zwingend auch die ↑Verdrängung der Prozesse von ihrem Prozessor zur Folge haben müssen. Zuallererst steht die Neuordnung der ↑Bereitliste entsprechend aktueller Randbedingungen im Vordergrund. Gekoppelt mit einem Verfahren, das ↑präemptive Planung verfolgt, kann der die Liste anführende Prozess abschließend dann Vorzug vor dem gegenwärtig auf der CPU stattfindenden Prozess genießen und diesen verdrängen.

Umschalter (en.) ↑*dispatcher*. Bezeichnung für ein ↑Programm, das die ↑Systemfunktion zum ↑Prozesswechsel implementiert.

unaufgelöste Referenz Bezeichnung für eine ↑Referenz, deren ↑Adresse noch unbekannt ist. Eine solche Referenz wird beim ↑Binden gelöscht und durch eine ↑Bindung ersetzt.

unicast (dt.) ↑Einzelruf.

Unicode Universalzeichensatz (1991), bei dem ein bis vier ↑Byte zur Kodierung eines einzelnen Zeichens verwendet werden. Unterteilt in 17 Ebenen, mit jeweils bis zu 2^{16} Zeichen. Obermenge von ↑ASCII, die die ersten 128 Zeichen entsprechend definiert.

unilateral synchronisation (dt.) ↑unilaterale Synchronisation.

unilaterale Synchronisation (en.) ↑*unilateral synchronisation*. Bezeichnung einer ↑Synchronisation, die sich nur in einer bestimmten Rolle blockierend für einen ↑Prozess auswirken kann; auch ↑logische Synchronisation oder ↑Bedingungssynchronisation.
Zwischen den Prozessen besteht eine Leser-Schreiber- oder Erzeuger-Verbraucher-Beziehung allgemein in Bezug auf ein ↑konsumierbares Betriebsmittel. Der Schreiber-/Erzeugerprozess zeigt das ↑Ereignis an, ein Betriebsmittel zur Verfügung gestellt zu haben und muss zur Durchführung dieser Anzeige (Signalisierung) selbst nicht warten. Demgegenüber ist der Leser-/Verbraucherprozess in seinem Vorankommen von einem solchen Ereignis abhängig. Er wird blockieren, wenn das Ereignis (Signalisierung) noch nicht stattgefunden hat.
Solch ein Muster der Kooperation von Prozessen, formuliert als ↑nichtsequentielles Programm, definiert die konsequente ↑Aktionsfolge für einen bestimmten Sachzusammenhang, das daher auch als logische Synchronisation bezeichnet wird. Darüberhinaus ist ein derartiges Ereignis auch Beispiel für die Entkräftung der Wartebedingung eines Prozesses, der mit dieser Betrachtungsweise dann der Bedingungssynchronisation unterliegt. Grundlage für dieses Muster der Synchronisation bildet ein ↑allgemeiner Semaphor, der ein bestimmtes Betriebsmittel repräsentiert. Die Anzahl der mit ↑V durch einen Prozess freigesetzten Betriebsmitteleinheiten ergibt sich durch den positiven Wert des Semaphors. Mit ↑P wird diese Anzahl durch einen für gewöhnlich anderen Prozess reduziert, der damit eine Betriebsmitteleinheit konsumiert. Der Absolutwert eines negativen Werts des Semaphors gibt die Anzahl von Betriebsmitteleinheiten an, auf deren Bereitstellung durch `V` Prozesse in `P` warten.
Dieses Synchronisationsmuster ist aber nicht auf die Anwendung von Semaphore beschränkt, es zeigt sich überall dort in einem nichtsequentiellen Programm, wo das Vorankommen eines Prozesses abhängig gemacht werden muss von einer Zustandsänderung durch einen anderen Prozess. Beispielsweise kann solch eine Prozessabhängigkeit auch durch eine ↑Bedingungsvariable zum Ausdruck gebracht werden. Aber auch ↑geschäftiges Warten auf einen Ereigniseintritt ist eine typische Form dieses Musters.

uninterruptible mode (dt.) ↑unterbrechungsfreier Betrieb.

uniprocessor (dt.) ↑Uniprozessor.

uniprogramming (dt.) ↑Einprogrammbetrieb.

Uniprozessor (en.) ↑*uniprocessor*. Bezeichnung für einen ↑Prozessor, der nur aus einer ↑CPU besteht beziehungsweise einen einzigen ↑Rechenkern enthält. Für die ↑Parallelverarbeitung von ↑Prozessen ist damit die zeitliche Partitionierung des Prozessors (↑partielle Virtualisierung) Voraussetzung. Überhaupt lässt sich ↑Nebenläufigkeit dann nur mit Hilfe von ↑Unterbrechungsanforderungen zum Ausdruck bringen.

Universalbetriebssystem (en.) ↑*general-purpose operating system*. Bezeichnung für ein ↑Betriebssystem, das allgemein und umfassend eingesetzt werden kann und verschiedenen Anwendungszwecken dient; Gegenteil von ↑Spezialbetriebssystem. Trugschluss ist, dass ein solches Betriebssystem allen Anwendungszwecken gleichermaßen gut dienen kann. Vielmehr wird im Allgemeinen auf Kompromisslösungen etwa bei der ↑Betriebsmittelverwaltung zurückgegriffen, die voraussichtlich für viele Anwendungsbereiche akzeptable Leistungen von dem ↑Rechensystem erwarten lassen. Das schließt ein umfassendes Funktionsangebot ein, auch wenn nicht alle Funktionen von jeder Anwendung zwingend erfordert werden. Typische Beispiele dafür sind etwa ↑virtueller Speicher, ↑Verdrängung, ↑Nachrichtenversenden, ↑Virtualisierung, ↑Schutz, aber auch das ↑Dateisystem oder der ↑Mehrprogrammbetrieb. Daraus resultierende latente, nichtfunktionale Merkmale eines Rechensystems in zeitlicher (z.B. ↑Latenzzeit oder ↑Laufzeit) wie auch räumlicher (z.B. Platzbedarf im ↑Hauptspeicher) Hinsicht können Anwendungen sogar verhindern.

UNIX Mehrplatz-/Mehrbenutzerbetriebssystem. Erste Installation im Jahr 1969 (↑DEC PDP-7), zunächst programmiert in ↑Assemblersprache, später (1972) umgearbeitet unter Verwendung von ↑C. In der Anfangsphase (1969) unter dem Namen UNICS geführt, als Abkürzung für (en.) „*Uniplexed Information and Computing Service*“ und Wortspiel in Bezug auf ↑Multics. Ursprung vieler Betriebssystementwicklungen, insbesondere ↑Linux und ↑Darwin.

unprivileged mode siehe ↑*user mode*.

unresolved reference (dt.) ↑unaufgelöste Referenz.

unteilbare Anweisung (en.) ↑*indivisible statement*. Formulierung einer ↑Aktion, für die ↑Unteilbarkeit gilt. Eine solche Aktion findet scheinbar „zeitlos“ statt. Während die Aktion stattfindet, ändert sich der Zustand von dem ↑Programm, in dem die betreffende Anweisung kodiert ist, anscheinend nicht.

unteilbarer Grundblock (en.) ↑*atomic basic block*. Bezeichnung für einen ↑Grundblock, für den ↑Unteilbarkeit gilt.

unteilbares Betriebsmittel (en.) ↑*indivisable resource*. Ausschließlichkeit (↑*mutual exclusion*) erforderndes ↑wiederverwendbares Betriebsmittel, dessen Einheiten zu einem Zeitpunkt nur von maximal einen ↑Prozess erworben (*acquire*) und für die Zeitdauer bis zur Freisetzung (*release*) belegt werden dürfen.
Typisches Beispiel eines solchen Betriebsmittels ist jede Form von Drucker: für die Dauer eines Druckvorgangs steht das Druckgerät keinem anderen Druckprozess zur Verfügung, ansonsten können unlesbare oder unverständliche Ausdrucke die Folge sein. In diese Kategorie ist aber auch ein einzelnes ↑Speicherwort einzuordnen, wenn nämlich durch ↑Parallelverarbeitung nicht tolerierbare Lese-Schreib-Konflikte für eine an der ↑Adresse dieses Worts liegende ↑gemeinsame Variable auftreten können. Im übertragenen Sinn passt auch ein ↑kritischer Abschnitt dazu, bei dem letztlich ja die von seinem Programmtext belegten Speicherworte — nicht etwa die von dem Text referenzierten Daten! — zeitweilig der exklusiven Nutzung nur durch einen Prozess unterzogen sind (↑unteilbarer Grundblock).

Unteilbarkeit (en.) ↑*indivisibility*. Umstand, der die Aufsplittung einer ↑Aktion oder ↑Aktionsfolge und Verteilung der Einzelmaßnahmen auf verschiedene Zeitintervalle verhindert. Die (Schrittfolge einer) Aktion beziehungsweise Aktionsfolge findet scheinbar gleichzeitig statt, sie ist atomar, kann sich nicht mit einem weiteren Exemplar ihrer selbst zeitlich überlappen.

Unterbrecherbetrieb (en.) ↑*interrupt mode*. Ausprägung einer ↑Betriebsart, die bestimmte ↑Aktionen eines ↑Betriebssystems als Folge einer ↑Unterbrechung auslöst und zeitlich überlappt zum laufenden ↑Prozess stattfinden lässt. Voraussetzung dafür ist die ↑Unterbrechungsanforderung durch ein Gerät (für gewöhnlich ↑Peripherie).

Unterbrechung (en.) ↑*interrupt*. Störung von einem ↑Prozess, einer ↑Aktionsfolge oder ↑Aktion, verursacht durch ein in Bezug auf diese Handlungen externes Ereignis. Das Ereignis findet asynchron zu der Handlung statt, unvorhersagbar und nicht reproduzierbar. Bestenfalls lässt sich noch die ↑Zwischenankunftszeit der Ereignisse abschätzen, womit jedoch nur eine Aussage über die Frequenz und nicht über den Zeitpunkt möglich ist. Eine solche Störung kann maskierbar (↑IRQ) oder nichtmaskierbar (↑NMI) sein, das heißt, das ↑Betriebssystem vermag sie entweder zu unterbinden (↑*interrupt lock*), um sie nach einer gewissen Zeitspanne wieder zuzulassen, oder grundsätzlich nicht abzuwehren.

Unterbrechungsanforderung (en.) ↑*interrupt request*. Auch kurz als ↑IRQ bezeichnete Forderung der ↑Peripherie an die ↑CPU, eine ↑Unterbrechungsbehandlung einzuleiten und durchzuführen.

Unterbrechungsbefehl (en.) ↑*interrupt command*. Spezielle Art von ↑Maschinenbefehl eines ↑Maschinenprogramms, mit dem die ↑Unterbrechung der Ausführung eben dieses ↑Programms durch einen darin stattfindenden ↑Prozess erzwungen wird. Der Zweck dieses Maschinenbefehls besteht darin, einen Wechsel von der ↑Benutzerebene hin zur ↑Systemebene zu erreichen. Typischerweise leitet ein solcher Befehl einen ↑Systemaufruf ein oder definiert einen im Rahmen der ↑Fehlerbereinigung platzierten ↑Haltepunkt.

Exkurs Je nach ↑CPU gibt es für diesen Befehl unterschiedliche Bezeichnungen (`int`: ↑x86; `trap`: ↑PDP 11, ↑m68k; `sc`: ↑PowerPC; `svc`: ↑VM/370) für die gleiche Bedeutung. Verschiedentlich wurden auch *illegale* ↑*Operationskodes* genutzt, etwa der sogenannte ↑*A-trap* von m68k. Allen gemeinsam ist, die Ausführung des gegenwärtig stattfindenden ↑Prozesses zu unterbrechen, nur um ihn daraufhin mit *Systemprivilegien* wiederaufzunehmen. Nachfolgend mit einer ↑Assemblieranweisung als ↑Makrobefehl (`sos`) für x86 definiert:

```
.macro sos          # switch to operating system
    int $42         # mode change: trap through vector 42
.endm               # done
```

Der Wechsel von Benutzer- zu Systemebene muss nicht zwingend dadurch geschehen, eine vom ↑Betriebssystem zu behandelnde ↑synchrone Ausnahme hervorzubringen. So bewirkt beispielsweise der für x86 definierte spezielle Maschinenbefehl `sysenter` den gewünschten Ebenenwechsel, ohne den Umweg über eine solche vergleichsweise aufwendige ↑Ausnahme gehen zu müssen. Dadurch lässt sich die mit einem Systemaufruf verbundene ↑Latenz wesentlich verringern.

Unterbrechungsbehandlung (en.) ↑*interrupt handling*. Vorgang zur Analyse und Bearbeitung der bei Ausführung von einem ↑Maschinenprogramm aufgetretenen Störung (↑*interrupt*). Handelt es sich um eine berechtigte (echte) Störung, wird diese bearbeitet und die Ausführung des unterbrochenen Programms anschließend wieder aufgenommen. Im Falle einer unberechtigten Störung (↑*spurious interrupt*), wird diese vermerkt. Zuviele unberechtigte Störungen innerhalb kurzer Zeit deuten auf einen Fehler in der Hardware hin und könnten ↑Panik auslösen. Anderenfalls wird die Programmausführung fortgesetzt.

Die ↑Aufgabe zur Behandlung einer ↑Unterbrechung übernimmt ein ↑Unterbrechungshandhaber. Je nach Art der Unterbrechung gibt es verschiedene solcher Handhaber, jeder davon repräsentiert die ↑Wurzelprozedur einer bestimmten ↑Systemfunktion. So münden beispielsweise ↑IRQ und ↑NMI aufgrund ihrer verschiedenartigen Charakteristik typischerweise nicht in dieselbe Wurzelprozedur. In ähnlicher Weise werden ↑Unterbrechungsanforderungen verschiedener Geräteklassen (↑*major device number*) bedient, so das ↑Betriebssystem derartige Klassen unterscheidet und verwaltet, wie es beispielsweise mit ↑UNIX-artigen Installationen der Fall ist. Für gewöhnlich verfügt jede Geräteklasse über ihren speziellen Unterbrechungshandhaber und allgemein über einen spezifischen ↑Gerätetreiber. Nimmt der ↑Prozessor eine Unterbrechungsanforderung entgegen, wird der der Quelle dieser Anforderung zugeordnete Handhaber aktiviert. Dieser Handhaber kennt seine Geräteklasse — nicht unähnlich wie eine ↑virtuelle Methode das ↑Objekt kennt, auf das sie operiert.

Je nach Beschaffenheit der Hardware gibt es für die Feststellung des passenden Handhabers verschiedene Herangehensweisen. Im einfachsten Fall für den ↑Unterbrecherbetrieb ist es, wenn die Hardware es erlaubt, jeder möglichen Unterbrechungsanforderung die ↑Adresse ihres jeweiligen Unterbrechungshandhabers zuzuordnen. Dies geschieht gemeinhin auf Grundlage einer ↑Unterbrechungsvektortabelle, sofern die Anzahl der Tabelleneinträge die der Quellen von Unterbrechungsanforderungen nicht unterschreitet. In solch einer Konstellation sendet die Quelle im ↑Unterbrechungszyklus eine Unterbrechungsnummer an den Prozessor, der diese sodann zur *indizierten Adressierung* der Vektortabelle verwendet, damit die Adresse des Unterbrechungshandhabers der Tabelle entnimmt und diesen schließlich aktiviert.

Verfügt der Prozessor nicht über solch eine Tabelle oder ist die Anzahl der zur Anforderungsbehandlung nutzbaren Tabelleneinträge zu klein, arbeitet ein per Default aktivierter allgemeiner Unterbrechungshandhaber im ↑Aufrufbetrieb und fragt der Reihe nach den Zu-

stand der möglichen Unterbrechungsquellen (z.B. ↑Peripheriegeräte) ab. Die erste Quelle, die anzeigt, eine Unterbrechungsanforderung gemeldet zu haben, bestimmt den Unterbrechungshandhaber beziehungsweise die Geräteklasse (so das Betriebssystem diese implementiert). Technisch ist diese Quelle für gewöhnlich als ↑Ein-/Ausgaberegister repräsentiert, aus deren Adresse sich der Handhaber / die Geräteklasse ableiten lässt. Das Betriebssystem führt Buch darüber, welche dieser Ein-/Ausgaberegister mit welchem Handhaber / welcher Geräteklasse assoziiert sind und kann so die benötigte Abbildung leisten, um im Aufrufbetrieb schlussendlich die gerätespezifische Unterbrechungsbehandlung auszulösen.
Für gewöhnlich stellt jede dieser Unterbrechungen eine ↑asynchrone Ausnahme dar, was zur Folge hat, dass der betreffende Unterbrechungshandhaber zu unvorhersehbaren und nicht reproduzierbaren Zeitpunkten zur Ausführung kommt. Programmabschnitte im Betriebssystem, die empfindlich gegenüber solchen plötzlichen Unterbrechungen sind und gegebenenfalls sogar zu ↑Fehlern führen können, müssen sich daher durch geeignete Maßnahmen zur ↑Synchronisation schützen. Nicht selten bedeutet dies das Setzen von ↑Unterbrechungssperren an den relevanten Programmstellen. Alternativ bietet sich auch ↑nichtblockierende Synchronisation an, um der sonst gegebenenfalls zu hohen ↑Unterbrechungslatenz bei solch einer Sperre vorzubeugen.

Unterbrechungsdeskriptortabelle Jargon (Intel) für eine ↑Unterbrechungsvektortabelle, deren Einträge ↑Deskriptoren für die ↑Ausnahmehandhaber sind.

unterbrechungsfreier Betrieb (en.) ↑*uninterruptible mode*. ↑Arbeitsmodus von einem ↑Prozessor (↑CPU, ↑Rechenkern), um ↑Synchronisation herzustellen. In dem Modus lässt der Prozessor eine ↑Unterbrechungsanforderung nicht durch. Der Prozessor handelt für das ↑Betriebssystem (↑*privileged mode*), unterbindet dabei aber gleichzeitig eine eventuelle ↑Unterbrechungsbehandlung. Dadurch wird eine mögliche nebenläufige ↑Aktionsfolge für den in dem Modus handelnden Prozessor ausgeschlossen. Der Prozessor wechselt in diesen Modus entweder implizit, nämlich wenn er in seinem ↑Unterbrechungszyklus eine anhängende Unterbrechungsanforderung erkennt, oder explizit, durch Setzen einer ↑Unterbrechungssperre. Während der Prozessor in dem Modus agiert, ist sein Unterbrechungszyklus faktisch außer Kraft. Der Prozessor verlässt den Modus ebenfalls implizit oder explizit, nämlich beim Rücksprung aus der Unterbrechungsbehandlung (wenn der zu einem früheren Zeitpunkt im Unterbrechungszyklus gesicherte und jetzt wiederhergestellte ↑Prozessorstatus dies bestimmt) oder Aufheben der Unterbrechungssperre. Je nach Art der Zustellung des der Unterbrechungsanforderung zugrunde liegenden Digitalsignals, dessen Frequenz und der Dauer des Modus ist es nach seiner Beendigung möglich, eine solche zwischenzeitlich gestellte Anforderung entweder verpasst zu haben (↑Flankensteuerung) oder nachträglich anzunehmen und zu behandeln (↑Pegelsteuerung).

Unterbrechungshandhaber (en.) ↑*interrupt handler*. Bezeichnung für einen speziellen, ausschließlich zur ↑Unterbrechungsbehandlung ausgelegten und vorgesehenen ↑Handhaber.

Unterbrechungshandhaberstumpf (en.) ↑*interrupt handler stub*. Bezeichnung für einen ↑Unterbrechungshandhaber begrenzter, fester Größe. Ein solcher Handhaber ist typisch für eine ↑CPU der ↑RISC Klasse. Je nach Art der ↑Unterbrechung ist in dem Stumpf die ↑Unterbrechungsbehandlung entweder komplett (↑SLIH) oder anteilig (↑FLIH) möglich. Gegebenenfalls verzweigt dieser Handhaber auch nur über eine in Software implementierte Vektortabelle (↑IVT oder ↑IDT).

Unterbrechungslatenz (en.) ↑*interrupt latency*. ↑Latenzzeit ab Zeitpunkt der Wahrnehmung der ↑Unterbrechung in der ↑CPU bis zum Zeitpunkt der Aufnahme der ↑Unterbrechungsbehandlung im Betriebssystem. Wurde keine ↑Unterbrechungssperre angefordert, ist die Verzögerungszeit bestimmt durch die restliche Ausführungszeit von dem ↑Maschinenbefehl, der im Moment der ↑Unterbrechung aktiv war. Wurde jedoch eine Unterbrechungssperre gesetzt, ergibt sich die Verzögerung aus der Restlaufzeit bis zur Aufhebung der Sperre durch das Betriebssystem.

Unterbrechungsmaske (en.) ↑*interrupt mask*. Bezeichnung für eine Vorrichtung in der ↑CPU, um die ↑Unterbrechung eines ↑Programmablaufs zeitweilig nicht zuzulassen. In logischer Hinsicht gibt die Maske eine bestimmte ↑Unterbrechungsprioritätsebene vor, auf der die CPU operiert. Unterbrechungen mit einer Priorität kleiner oder gleich dieser Ebene sind unterbunden. Das Setzen der Maske ist für gewöhnlich eine privilegierte Operation (↑*privileged mode*).

In technischer Hinsicht ist eine solche Maske eine ↑Bitleiste von unterschiedlicher Länge, je nach CPU. Für ↑x86 sowie allen gegenwärtigen 64-Bit Varianten dieser Familie (von Intel oder AMD) besteht die Bitleiste aus genau einem Bit (*interrupt flag*) im ↑Statusregister, womit nur eine Unterbrechungsprioritätsebene definiert ist. Bei Prozessoren der ↑PDP 11 oder ↑m68k Familie umfasst(e) die Bitleiste drei Bits und damit acht Prioritätsebenen.

Unterbrechungsprioritätsebene (en.) ↑*interrupt priority level*. Attribut des aktuellen Systemzustands, das die Priorität der momentan von der ↑CPU akzeptierten ↑Unterbrechungsanforderung anzeigt. Typischerweise werden Anforderungen mit einer Priorität kleiner oder gleich der Priorität der CPU nicht akzeptiert, diese sind gesperrt. Im Normalbetrieb läuft die CPU auf Prioritätsebene 0 und jede Unterbrechungsanforderung hat eine Priorität größer als 0. Mit jeder akzeptierten Unterbrechungsanforderung wird die Priorität der CPU auf die Ebene dieser Anforderung gehoben. Bei Rückkehr aus der ↑Unterbrechungsbehandlung kehrt die Priorität der CPU wieder auf die vorige Ebene zurück. Im ↑Unterbrechungszyklus führt die CPU damit implizit eine ↑Unterbrechungssperre für die Prioritätsebene durch, die mit der Unterbrechungsanforderung assoziiert ist. Eine solche Sperre kann jedoch auch explizit gesetzt werden, um der ↑Unterbrechung eines Programmablaufs vorzubeugen (↑kritischer Abschnitt). Darüber hinaus kann diese Sperre im Rahmen der Unterbrechungsbehandlung explizit aufgehoben werden, um noch vor Rückkehr daraus auf Unterbrechungsanforderungen reagieren zu können und damit die ↑Unterbrechungslatenz zu verkürzen. Läuft die Unterbrechung jedoch über ↑Pegelsteuerung und hat der ↑Unterbrechungshandhaber vor Aufhebung der Sperre die Unterbrechung noch nicht am ↑Peripheriegerät bestätigt, gilt die zugehörige Anforderung immer noch als anhängig und es kommt zum indirekt rekursiven Aufruf des Handhabers durch die CPU. Dies kann ↑Panik auslösen, da davon auszugehen ist, dass ein solcher Kreislauf im Unterbrechungshandhaber nicht durchbrochen wird beziehungsweise werden kann.

Unterbrechungssperre (en.) ↑*interrupt lock*. Vorkehrung zur Abwehr einer ↑Unterbrechung, stellt ↑Synchronisation her. Implementiert als ↑privilegierter Befehl, der nur lokale Auswirkung auf seinen ↑Prozessor hat und eine an ihn gestellte ↑Unterbrechungsanforderung der ↑Peripherie nicht durchlässt. Dazu muss diese Anforderung am Prozessor maskierbar sein (↑IRQ). Eine nichtmaskierbare Anforderung (↑NMI) dagegen kann nicht gesperrt werden: Gegebenenfalls lässt sich die Peripherie jedoch zeitweilig umprogrammieren, einen NMI nicht anzufordern. Zur Aufhebung der Sperre kommt eine entsprechende inverse Operation zum Einsatz. Bei der Unterbrechungsabwehr besonders zu beachten ist die Art der Zustellung des Digitalsignals (↑Pegelsteuerung, ↑Flankensteuerung).

Exkurs Für ↑x86, eine ↑CPU mit nur einer ↑Unterbrechungsprioritätsebene, gestalten sich die Maßnahmen, um Unterbrechungsanforderungen abzuwenden (*avert*) und wieder zuzulassen (*admit*), vergleichsweise einfach. Um jedoch auch Prozessoren mit mehreren Prioritätsebenen (8 bei ↑m68k, bis zu 256 bei ↑Cortex-M3) oder externen ↑Unterbrechungssteuereinheiten (↑PIC für x86) behandeln zu können, ist eine generalisierte Schnittstelle für die Abwehr-/Zulassungsoperationen zweckmäßig. Hierzu zunächst eine Typdefinition, um Art beziehungsweise Wirkungsfeld der Sperre spezifizieren zu können

```
typedef enum level {
    INO = ((1 << 8) - 1), /* 0..255 = IRQ number */
    IRQ = (1 << 8),
    NMI = (1 << 9),
    ALL = (IRQ | NMI))
} level_t;
```

Darauf aufbauend kann die Operation zum Abwehr von Unterbrechungsanforderungen (für x86) wie folgt formuliert sein, wobei einfachheitshalber hier nur die IRQ-Sperre direkt am Prozessor betrachtet wird:

```
inline void avert (level_t ban) {
    if (ban & IRQ) {                                        /* disable IRQ? */
        if (!(ban & INO)) asm volatile ("cli"); /* no IRQ number, clear IF */
        else …                      /* IRQ number specified, disable at PIC */
    }
    if (ban & NMI)                                          /* disable NMI? */
        …                                     /* turning-off of all devices */
}
```

Die hier durch eine inzeilige Anweisung als ↑symbolischer Maschinenkode spezifizierte ↑Maschinenbefehl ist ein ↑privilegierter Befehl, der die CPU dazu veranlasst, eingehende ↑Signale auf der Unterbrechungsleitung zu ignorieren (`cli`). Effektiv wird im ↑Statusregister der CPU lediglich ein ↑Markierungsbit (*interrupt flag*, IF) gelöscht beziehungsweise gesetzt. Die inverse Operation, nämlich solche Signale wieder wahrzunehmen (`sti`), zeigt folgende Skizze:

```
inline void admit (level_t ban) {
    if (ban & IRQ) {                                         /* enable IRQ? */
        if (!(ban & INO)) asm volatile ("sti"); /* no IRQ number, store IF */
        else …                      /* IRQ number specified, enable at PIC */
    }
    if (ban & NMI)                                           /* enable NMI? */
        …                                      /* turning-on of all devices */
}
```

Mit beiden hier beispielhaft ausformulierten Operationen wird eine *totale Sperre* erhoben oder zurückgenommen, ein Vorgehen, das für gewöhnlich nur erforderlich wird, wenn ein ↑kritischer Abschnitt zwischen ↑Betriebssystemkern und ↑Unterbrechungshandhaber zu schützen ist — und dies nur mit solch einer „Holzhammermethode“ bewerkstelligt werden kann. In dem Fall verübt der Betriebssystemkern eine Art ↑blockierende Synchronisation mit dem Unterbrechungshandhaber, um einer zeit- und räumlich überlappten Ausführung des kritischen Abschnitts vorzubeugen. Im Falle einer erhobenen Sperre muss der Unterbrechungshaber nämlich warten, er ist blockiert, bis die Sperre wieder zurückgenommen wird und er erst dann an der Reihe ist, in den kritischen Abschnitt einzutreten.

Unterbrechungssteuereinheit (en.) ↑*interrupt controller*. Bezeichnung für ein gewöhnlicherweise durch das ↑Betriebssystem programmierbares Bauelement oder Bauteil (↑PIC) in einem ↑Rechner, das die von einem ↑Peripheriegerät gestellte ↑Unterbrechungsanforderung an die ↑CPU weiterleitet. Typischerweise ist damit einstellbar, ob die Weiterleitung zeitweilig zurückzuhalten ist (↑*interrupt lock*), in welcher Art und Weise die ↑Störleitung zur CPU bedient werden soll (↑*edge-triggered interrupt*, ↑*level-triggered interrupt*), auf welcher ↑Unterbrechungsprioritätsebene die Weiterleitung zu erfolgen hat und über welchen ↑Unterbrechungsvektor schließlich die ↑Unterbrechungsbehandlung geschieht.

Unterbrechungsvektor (en.) ↑*interrupt vector*. Bezeichnung für eine definierte ↑Adresse im ↑Hauptspeicher, an der entweder ↑Daten oder ↑Maschinenbefehle zur Auslösung einer ↑Unterbrechungsbehandlung liegen. Gemeinhin sind solche Vektoren in einer Tabelle (↑IVT oder ↑IDT) zusammengefasst.

Enthält der Vektor Daten (typisch für ↑CISC), handelt es sich dabei in der einfachsten

Ausfertigung lediglich um die Adresse (↑PC) des von der ↑CPU automatisch im ↑Unterbrechungszyklus zu startenden ↑Unterbrechungshandhabers. Ein treffendes Beispiel dafür ist ↑m68k. Bei einer CPU mit segmentierter Adressierung ist für gewöhnlich zusätzlich noch der ↑Segmentname Teil des Vektors, wie etwa im Falle des ↑x86 bis zum 80286. Ab 80386 ist ein solcher Vektor ein ↑Deskriptor (8 Bytes), durch den insbesondere die Art des ↑Ausnahmehandhabers spezifiziert ist und welchen Tortyp die CPU für seine Aktivierung verwenden soll (↑*interrupt*, ↑*trap* oder ↑*task gate*). Bei Prozessoren der ↑PDP 11 Familie besteht der Vektor aus einem Tupelwert, der bei Annahme der ↑Unterbrechung von der CPU als PC und ↑PSW übernommen wurde.
Enthält der Vektor Maschinenbefehle (typisch für ↑RISC), beginnt die CPU die Ausführung mit der Unterbrechungsbehandlung direkt an dieser Stelle. Ein solcher Vektor enthält lediglich einen ↑Unterbrechungshandhaberstumpf von fester, begrenzter Größe (z.B. 256 Bytes beim ↑PowerPC). Durch diesem Stumpf ist nicht nur der weitere Verlauf der Unterbrechungsbehandlung bestimmt, sondern auch die Art und Weise, wie die jeweilige Behandlung problemspezifisch im ↑Betriebssystem eingebettet und zu aktivieren ist (z.B. als ↑Unterprogramm, ↑Koroutine oder gar in Form einer ↑Prozessinkarnation).

Unterbrechungsvektortabelle (en.) ↑*interrupt vector table*. Vorrichtung in einer ↑CPU zur listenförmigen Zusammenstellung der verfügbaren ↑Unterbrechungsvektoren. Wenn die CPU ihren ↑Unterbrechungszyklus durchführt, liest sie eine (vom ↑Peripheriegerät oder ↑PIC gesendete) Unterbrechungsnummer vom ↑Bus und führt mit dem gelesenen Wert sowie der Tabellenadresse als Operanden eine *indizierte Adressierung* durch, um den der ↑Unterbrechung zugeordneten Vektor zu laden und damit die ↑Unterbrechungsbehandlung zu starten. Die Tabellengröße ist prozessorspezifisch, für ↑x86 sind 256 Einträge festgelegt.

Unterbrechungszyklus (en.) ↑*interrupt cycle*. Phase im ↑Abruf- und Ausführungszyklus, wird typischerweise am Ende der Befehlsausführung durchlaufen und umfasst folgenden *Vorspann* an Teilschritten, um die ↑Unterbrechungsbehandlung — allgemein: ↑Ausnahmebehandlung, denn für die ↑Abfangung gilt dasselbe — einzuleiten: (1) den ↑Prozessorstatus sichern, mindestens davon die Inhalte von ↑Statusregister und ↑Befehlszähler, (2) den Befehlszähler mit der ↑Adresse vom ↑Unterbrechungshandhaber laden und (3) in den privilegierten ↑Betriebsmodus wechseln, sofern von der ↑CPU implementiert.
Der *Nachspann* der Unterbrechungsbehandlung kommt durch einen speziellen ↑Maschinenbefehl in ↑Aktion, der ganz normal im Abruf- und Ausführungszyklus verarbeitet wird und folgenden wesentlichen Schritt macht: (5) den Prozessorstatus wieder herstellen. Hier ist zu beachten, dass die Schritte (2) und (5) jeweils den Befehlszähler verändern und damit für den nächsten Durchlauf vom Abruf- und Ausführungszyklus vorgeben, von welcher Adresse der nächste Maschinenbefehl zur Ausführung abgerufen werden soll. Diese Schritte bewirken letztlich den Hinsprung (2) zur Unterbrechungsbehandlung und den Rücksprung (5) zu dem ↑Prozess, der unterbrochen/abgefangen wurde.
Da der Betriebsmodus der CPU in ihrem Statusregister vermerkt ist, wird mit Wiederherstellung vom Prozessorstatus implizit zu dem Betriebsmodus umgeschaltet (5), der zum Zeitpunkt der ↑Unterbrechung aktiv war. Zur Sicherung (1) und Wiederherstellung (5) all dieser Statusdaten findet typischerweise ein ↑Stapelspeicher Verwendung.

Unterprogramm (en.) ↑*subprogram*. ↑Programm, dessen Aufruf in demselben (Rekursion), einem anderen oder mehrerer anderer Programme kodiert ist. Je nach Programmiersprache und -paradigma technisch verschieden umgesetzt und unterschiedlich bezeichnet: Operation, Abschnitt, ↑Routine, Subroutine, Prozedur, Funktion, Methode oder Makro.

unverdrängbarer kritischer Abschnitt (en.) ↑*non-preemptive critical section*. Ein ↑kritischer Abschnitt der sicherstellt, dass dem sich zur Zeit darin befindlichen ↑Prozess der ↑Prozessor <u>nicht</u> entzogen wird. Beim Eintritt in den kritischen Abschnitt erhebt der betreffende Prozess eine ↑Verdrängungssperre, die er beim Austritt wieder aufhebt. Damit durchläuft der Prozess den Abschnitt „ungestört“ bis zum Ende (↑*run to completion*), er wird vom

↑Umschalter frühestens beim Verlassen des Abschnitts zugunsten eines anderen Prozesses vom Prozessor verdrängt. Innerhalb eines solchen Abschnitts kann ein Prozess somit auch nicht durch andere, vom ↑Planer zwischenzeitlich dazwischengeschobene Prozesse verzögert werden. Allerdings sind damit Verzögerungen aufgrund von ↑Unterbrechungsanforderungen nicht zwingend ausgeschlossen.

Ureingabe (en.) ↑*bootstrapping*. Vorgang zur Inbetriebnahme von einem ↑Rechner, indem der ↑Urlader über eine als Schalttafel ausgelegte ↑Systemkonsole manuell in den ↑Hauptspeicher gebracht wird. Die wesentlichen Schritte dabei sind (am Beispiel ↑PDP 11/40):

1. die Hardware des Rechners betriebsbereit machen:
   (a) den ENABLE/HALT Schalter in die HALT Position bringen, damit die Funktionen der Systemkonsolle freigeben
   (b) sicherstellen, dass der OFF/POWER/LOCK Schalter in der POWER Position steht
2. das ↑Urladeprogramm in den Hauptspeicher bringen:
   (a) die ↑Adresse der ersten zu beschreibenden Stelle (↑Speicherwort) im Hauptspeicher als binär kodierten (16-stelligen) Wert am Schaltregister einstellen
   (b) den eingestellten Wert durch Betätigung der LOAD ADRS Taste in das Busadressregister (BAR) laden
   (c) das als nächstes in den Hauptspeicher zu transferierende Datum als binär kodierten (16-stelligen) Wert am Schaltregister einstellen
   (d) den eingestellten Wert durch Betätigung der DEP Taste in das durch BAR adressierte Speicherwort laden, BAR wird dabei automatisch um 2 erhöht
   (e) die Schritte ab 2c wiederholen, solange das Urladeprrogramm noch nicht vollständig eingegeben worden ist
3. den Rechner hochfahren:
   (a) das Bandlaufwerk anschließen und das Band mit dem gewünschten ↑Absolutlader in das Laufwerk einlegen
   (b) die Schritte 2a und 2b zur Eingabe der Adresse, an der die Ausführung des Urladeprogramms starten soll, erneut durchführen
   (c) den ENABLE/HALT Schalter in die ENABLE Position bringen und mit der START Taste das Urladeprogramm schließlich zur Ausführung bringen
   (d) abwarten, dass das Bandlaufwerk zum Stillstand kommt und der Absolutlader damit seine Aufgabe (das ↑Betriebssystem zu laden) erfüllt hat
   (e) den OFF/POWER/LOCK Schalter in die LOCK Position bringen

Typischerweise ist das Urladeprogramm nicht sehr umfangreich: im betrachteten Beispiel umfasst es lediglich etwa 14 Maschinenbefehle beziehungsweise Speicherworte, die nach und nach in der Schleife um Schritt 2c einzugeben sind. Insgesamt ist der Ablauf jedoch sehr spezifisch auf den jeweiligen Rechner ausgelegt und variiert damit durchaus auch stark von Hersteller zu Hersteller. In (vom Rechnerhersteller mitgelieferten) Festwertspeicher kodiert, kann der Vorgang allerdings weitestgehend automatisiert ablaufen.

urgency (dt.) ↑Dringlichkeit.

Urladen (en.) ↑*bootstrap*. Vorgang bei der Inbetriebnahme von einem ↑Rechner, nämlich das ↑Betriebssystem in den ↑Hauptspeicher zu bringen.

Urladeprogramm (en.) ↑*bootstrap loader*. Bezeichnung für ein ↑Programm) zum ↑Urladen von einem ↑Betriebssystem, bestimmt die Funktions- und Arbeitsweise von dem ↑Urlader.

Urlader (en.) ↑*bootstrap*. Bezeichnung eines ↑Laders, der zur Inbetriebnahme des ↑Rechners das ↑Betriebssystem in den ↑Hauptspeicher bringt und startet. Der Ladevorgang ist üblicherweise zweistufig organisiert. Zunächst wird ein auf das Betriebssystem zugeschnittener Lader vom gewählten ↑Datenträger heruntergeladen, typischerweise aus dem ersten ↑Block (↑*boot block*), und gestartet. Dieser Lader lädt das Betriebssystem und bietet dazu häufig eine Wahl der ↑Betriebsart an, in der das geladene Betriebssystem den Rechner betreiben soll: im Falle von ↑UNIX läuft der Rechner zunächst im ↑Einbenutzerbetrieb und wird dann (ggf. auch erst bei Bedarf) in den ↑Mehrbenutzerbetrieb überführt.

USB Abkürzung für (en.) *universal serial bus*, (dt.) vielseitiger serieller ↑Bus.

use case (dt.) ↑Anwendungsfall.

used bit siehe (en.) ↑*reference bit*.

user (dt.) ↑Benutzer.

user ID Abkürzung für (en.) ↑*user identification*.

user identification (dt.) ↑Benutzerkennung.

user level (dt.) ↑Benutzerebene.

user mode (dt.) Benutzermodus: ↑Arbeitsmodus.

user process (dt.) ↑Benutzerprozess.

user program (dt.) ↑Anwenderprogramm, ↑Anwendungsprogramm, ↑Benutzerprogramm.

user space (dt.) ↑Benutzerbereich, -gebiet, -gelände.

user thread (dt.) ↑Anwendungsfaden.

user time (dt.) ↑Benutzerzeit.

user-level scheduling (dt.) Planung auf Benutzerebene. Bezeichnung für eine ↑mitlaufende Planung, die eng gekoppelt mit dem ↑Betriebssystem als Funktion direkt im Kontext eines ↑Benutzerprogramms stattfindet. Der ↑Planer als dedizierter Betriebssystemaufsatz ist ein ↑Unterprogramm in dem Benutzerprogramm. Dieser Ansatz gibt jedem Benutzerprogramm die Option zu einem eigenen Planer, der speziell auf die Bedürfnisse der ↑Anwendung zugeschnitten ist und dazu ein anwendungsorientiertes ↑Einplanungskriterium implementiert.
Mit verschiedenen Benutzerprogrammen können damit zugleich verschiedene Planer oberhalb des Betriebssystems tätig sein. Die Herausforderung dabei ist, ↑Simultanverarbeitung verschiedener Benutzerprogramme durch das Betriebssystem möglichst frei von ↑Interferenz mit den verschiedenen Planern zu ermöglichen. Zu beachten ist hier, dass die Planer in den Benutzerprogrammen nicht voneinander wissen und sie nur aktiv werden können, wenn das Betriebssystem die Ausführung ihres Benutzerprogramms veranlasst. Für gewöhnlich betreibt das Betriebssystem die ↑CPU dazu in einem ↑Zeitteilverfahren, das die Planer auf der Benutzerebene im ↑Rundlaufverfahren aktiviert.

user-level thread (dt.) ↑Faden der ↑Benutzerebene. Ein ↑Anwendungsfaden, der einen ↑Kontrollfluss der Benutzerebene repräsentiert, ausschließlich auf dieser verwaltet wird, ebenda verankert ist, für gewöhnlich immer im nichtprivilegierten ↑Arbeitsmodus (↑*user mode*) abläuft und auf einen Kontrollfluss der ↑Betriebssystemebene (↑*kernel-level thread*) basiert.

userland siehe ↑*user space*.

uses hierarchy (dt.) ↑Benutzthierarchie.

utilisation (dt.) ↑Auslastung.

utility (dt.) ↑Dienstprogramm.

utility piece (dt.) ↑Gebrauchsstück.

V Abkürzung für (hol.) *verhoog*, (dt.) erhöhen; (en.) *up*, *signal*, *release*. Neben ↑P die andere elementare Operation eines ↑Semaphors.

Variable (en.) ↑*variable*. Eine veränderliche Größe (Duden), der ein Platz bestimmter räumlicher Ausdehnung im ↑Arbeitsspeicher zugewiesen ist. Für gewöhnlich bemisst sich die Ausdehnung dieser Größe nach der Anzahl von ↑Bytes, die das ↑Exemplar eines bestimmten ↑Datentyps beansprucht. Dieses Exemplar besitzt einen ↑Namen, über den ein ↑Prozess die Größe abfragen oder verändern kann.

variable (dt.) Regelgröße, ↑Variable, Veränderliche.

VAX Abkürzung für „*virtual address extension*“ und Bezeichnung für eine von ↑DEC entwickelte ↑ISA als Nachfolge zur ↑PDP 11: 32-Bit, ↑CISC, vier Privilegstufen (*kernel*, *executive*, *supervisor*, *user*), 32 ↑Unterbrechungsprioritätsebenen, hardwareunterstützten ↑AST, erstmalig umgesetzt mit der VAX-11/780 (1977) und betrieben durch ↑VMS, später ↑OpenVMS.

Veränderungsbit (en.) ↑*modified bit*. Ursprünglich eine ↑Speichermarke nur im ↑Seitendeskriptor, die von der ↑MMU beim Schreibzugriff auf die betreffende ↑Seite gesetzt wird. Für ↑Zwischenspeicher mit ↑Rückschreibetechnik der demselben Zweck dienende Statusindikator für eine zwischengespeicherte ↑Zwischenspeicherzeile, um nämlich zu vermerken, ob seit Einlagerung der Zeile ein Schreibzugriff darauf geschah.

Verarbeitungskette (en.) ↑*pipeline*. Einrichtung oder Vorgang zur fließbandartigen Bearbeitung einer in Stufen oder Phasen zerlegten ↑Aufgabe durch verschiedene, hintereinander geschaltete Funktionseinheiten, realisiert in Hardware oder Software oder geeigneter Kombination von beiden. In einer solchen Kette bildet die Ausgabe einer Funktionseinheit die Eingabe einer nachgeschalteten Funktionseinheit. Für verschiedene Aufgaben werden die Aufgabenstufen durch die zuständigen Funktionseinheiten echt- oder pseudoparallel bearbeitet (↑*parallel processing*), sofern die Aufgaben in ↑Kausalordnung vorliegen.
Typisches Beispiel einer solchen Vorgehensweise ist die Bearbeitung eines ↑Maschinenbefehls durch heutige (2025) ↑Prozessoren: (1) den Befehl aus dem ↑Arbeitsspeicher laden, (2) im Prozessor dekodieren und (3) ausführen, um (4) abschließend die Ausführung durch Ablieferung eines Ergebnisses geeignet zu bestätigen. Diese vier *Verarbeitungsstufen* können für aufeinander im *Befehlsstrom* folgende, unabhängige Maschinenbefehle parallel arbeiten. Sehr ähnlich operieren Grafikprozessoren, die verschiedene arithmetische Einheiten zur photorealistischen Visualisierung (*rendering*) von ↑Daten enthalten. In Software gestaltet sich die Kette als Konstruktion von ↑Prozessinkarnationen, deren Operationen Ausgaben vorangestellter Verarbeitungsstufen als Eingaben entgegennehmen und ihrerseits Ausgaben an nachgeschaltete Einheiten abgeben (↑UNIX ↑*pipe*).

Verbindungsregister (en.) ↑*link register*. Bezeichnung für ein ↑Prozessorregister, das die Rücksprung- oder Fortsetzungsadresse des gegenwärtig auf dem ↑Prozessor ausgeführten ↑Unterprogramms enthält. Beim Unterprogrammaufruf wird diese Adresse <u>nicht</u> automatisch auf dem ↑Stapelspeicher gesichert, wodurch für den Hin- und Rücksprung kostspielige Schreib-/Leseoperationen für den ↑Hauptspeicher entfallen. Nur im Falle von geschachtelten Aufrufen kommt es zu solchen Haupstpeicherzugriffen, die dann durch explizit abzusetzende, aber gewöhnliche ↑Maschinenbefehle zum Schreiben/Lesen eines ↑Maschinenworts bewerkstelligt werden. Dieses Register ist typisches Merkmal des ↑Programmiermodells von einem ↑RISC.

Verbund ↑Datentyp, der aus einem oder mehreren (identischen oder verschiedenen) Datentypen zusammengesetzt ist. Die einzelnen Elemente (Attribute) dieses Datentyps sind entweder

nacheinander im ↑Speicher angeordnet und durch ihre jeweilige ↑Adresse eindeutig unterscheidbar oder überlagern sich im Speicher und besitzen alle dieselbe Adresse: `struct` beziehungsweise `union` in ↑C/↑C++. Jedes dieser Elemente kann selbst wieder einen solchen Zusammenschluss von Datenstrukturen bilden.

verdeckter Kanal (en.) ↑*covered channel*. Variante von einem Sicherheitsangriff auf ein ↑Rechensystem, die es ermöglicht, Informationen von einem ↑Prozess abzugreifen und zu einem anderen Prozess zu transferieren, ohne dass beide zur direkten Kommunikation berechtigt wären. Der Kanal ist nicht zum Informationstransfer bestimmt: er benutzt keine der von einem ↑Betriebssystem angebotenen legitimen Mechanismen zum Datentransfer, sondern zweckentfremdet andere legitime und per regulärem ↑Systemaufruf angebotene Mechanismen dafür. Beispiel ist der durch bestimmte Berechnungen eines Prozesses bewirkte Effekt auf die Systemlast, die von einem anderen Prozess gemessen werden kann. Die Lastschwankungen geben Anlass zu Spekulationen über die in dem Moment stattfindenden Vorgänge beziehungsweise werden gezielt zur Informationskodierung genutzt.

Verdrängung (en.) *displacement*. Maßnahme zur ↑Bevorrechtigung eines ↑Prozesses, nämlich um ein von einem anderen Prozess belegtes ↑wiederverwendbares Betriebsmittel selbst zu nutzen. Der das betreffende ↑Betriebsmittel derzeit besitzende Prozess wird davon zugunsten eines zu bevorzugenden Prozesses verdrängt. Eine ↑Aktionsfolge, durch die ein wiederverwendbares Betriebsmittel letztlich den Besitzer wechselt. Dem Prozess, der dieses Betriebsmittel gerade besitzt, wird es entzogen und einem anderen Prozess zugeteilt, der dann zum neuen Besitzer wird. Typisches Beispiel eines solchen Betriebsmittels ist die ↑CPU oder ein einzelner ↑Rechenkern, wenn nämlich die ↑Prozesseinplanung nach einem ↑Zeitteilverfahren arbeitet.

Hinweis Mit dem Begriff ist gemeinhin die deutsche Übersetzung von (en.) ↑*preemption* verbunden. Dies ist insofern nachvollziehbar, weil es in dem assoziierten Zusammenhang um die Bevorrechtigung eines Prozesses in der Nutzung eines bestimmten Betriebsmittels geht und der möglichen Folge, einen anderen Prozess von diesem Betriebsmittel zu verdrängen. Allerdings ist dabei die Rollenverteilung zu beachten: nicht der bevorrechtige Prozess, sondern der *nicht bevorrechtigte* Prozess wird verdrängt. Wenn nun davon die Rede ist, ein Prozess „preempted“ etwas, dann wird er die betreffende ↑Entität, die ein anderer Prozess in dem Moment besitzt, durch Bevorrechtigung erwerben (vgl. S. 49).

Verdrängungsebene (en.) ↑*preemption level*. Bezeichnung für den grundlegenden Mechanismuns von ↑SRP, durch den die Bearbeitung von ↑Aufgaben durch einen ↑Prozess gesteuert wird. Dieser Mechanismus bestimmt, ob eine ↑Aufgabe, A_α, eine andere Aufgabe, A_γ, von der Bearbeitung durch einen Prozess vom ↑Prozessor verdrängen kann. A_α verdrängt A_γ nur, wenn A_α auf einer höheren Ebene als A_γ angesiedelt ist.
Die Ebene wird durch eine natürliche Zahl ($\mathbb{N}_+$) identifiziert, deren Wert umgekehrt proportional zum relativen ↑Termin der betreffenden Aufgabe ist. Dabei entspricht der relative Termin einer festen Zeitspanne, innerhalb der ab einem bestimmten Zeitpunkt die Aufgabe bearbeitet sein muss. Dieser Zeitpunkt ist durch die ↑Bereitzeit der Aufgabe festgelegt.
Dem Prinzip nach besteht eine starke Analogie zur ↑Unterbrechungsprioritätsebene. Diese lässt eine ↑kaskadierte Unterbrechung nur dann zu, wenn die diesbezügliche ↑Unterbrechungsanforderung einer höheren Prioritätsebene zugeordnet ist als die Prioritätsebene, auf der die ↑Unterbrechungsbehandlung in dem Moment stattfindet. Dabei entspricht die Unterbrechungsanforderung der Bereitstellung einer Aufgabe (A_α) und die Unterbrechungsbehandlung der Bearbeitung einer Aufgabe (A_γ).

Verdrängungsgrad (en.) ↑*preemption level*. Abstufung des Vorhandenseins der Eigenschaft von einem ↑Betriebssystem, ↑Verdrängung frei von ↑Latenz zu gestalten.
Benutzt das Betriebssystem etwa eine ↑Unterbrechungssperre zur ↑Synchronisation, verzögert sich die mögliche Verdrängung von dem gerade auf (einem bestimmten ↑Rechenkern) der ↑CPU stattfindenden Prozess wenigstens für die Dauer dieser Sperre. Gleiches gilt im

Falle einer ↑Verdrängungssperre, die das Betriebssystem entweder zusätzlich oder alternativ für die Synchronisation gebraucht. Benutzt das Betriebssystem eine ↑Umlaufsperre, um Synchronisation für den ↑Planer in einem ↑Mehrkernprozessor oder ↑Multiprozessor zu erzielen, verzögert sich auch hier die mögliche Verdrängung eines in dem Fall aber fernen Prozesses: beispielsweise um den ↑Rechenkern dieses Prozesses für einen von einem anderen Rechenkern aus deblockierten vorrangigen Prozess zu nutzen. Zwischen den ↑Aktionen zum Aufheben der einen und Setzen einer gegebenenfalls anderen Sperre liegt ein Stück ↑nichtsequentielles Programm bestimmter Länge. Diese Länge variiert im Allgemeinen, wie auch die Häufigkeit solcher Stücke verteilt über das gesamte ↑Programm — bestimmt durch das ↑Operationsprinzip des Betriebssystems.
Ein Betriebssystem, das ↑präemptive Planung vorsieht, operiert für gewöhnlich *verdrängend* (↑*preemptive*) zwischen den gesperrten Programmabschnitten (↑*critical section*), vorausgesetzt sein Operationsprinzip impliziert keine allumfassende Sperre (↑BKL plus totale Unterbrechungssperre) und definiert damit seinen gesamten Komplex nicht tatsächlich als ↑sequentielles Programm. Ist das Betriebssystem dagegen vollumfänglich frei von solchen Sperren, operiert es *voll verdrängend* (↑*full preemptive*) und bietet damit die besten Voraussetzungen für das größtmögliche Maß an ↑Parallelität. Die genaue Abstufung der Intervalle, in der Verdrängung stattfinden kann, lässt sich demnach durch folgendes Verhältnis quantifizieren (*degree of preemption*):

$$dop = \frac{length(nonsequential\ code)}{length(total\ code)}$$

↑Bezugspunkt für die Bestimmung der jeweiligen Länge ist der einzelne ↑Maschinenbefehl der zugrunde liegenden CPU. Nur ein Betriebssystem, das einen ↑Prozesswechsel nach jeden einzelnen Maschinenbefehl erlaubt, hat einen Quotienten von Eins beziehungsweise lässt Verdrängung jederzeit, das heißt, 100 Prozent zu. In allen anderen Fällen liegt der Quotientenwert im Bereich $[0, 1[$ und der Grad unter 100 %. Damit würde Parallelität wie auch die Wahrscheinlichkeit von besserer Auslastung und Leistung von dem ↑Rechensystem entsprechend verringert.

Verdrängungspunkt (en.) ↑*preemption point*. Stelle in einem ↑Programm, an der die ↑Verdrängung eines ↑Prozesses, der nämlich durch eben dieses Programm definiert ist, stattfinden kann. Diese Stelle zeichnet sich aus durch eine Anweisung, die die Bereitwilligkeit eines Prozesses zur Abgabe des von ihm vorübergehend kontrollierten ↑Prozessors dem ↑Planer anzeigt. Der kooperativ von dem Prozess ins Spiel gebrachte Planer kann daraufhin eine ↑Umplanung vornehmen, deren Folge die ↑Einlastung des Prozessors durch einen anderen Prozess sein kann.

Verdrängungssperre (en.) ↑*preemption locking*. Sperren einer möglichen ↑Verdrängung eines ↑Prozesses von seinem ↑Prozessor, stellt ↑Synchronisation her (↑NPCS). Implementiert als ↑Aktionsfolge im ↑Betriebssystem, die einen ↑Prozesswechsel zeitweilig unterbindet. Entweder wird die ↑Ablaufplanung oder die ↑Einlastung von einem Prozess, dessen Auslösung durch ein bestimmtes ↑Ereignis verursacht ist, zurückgestellt.
Je nach Implementierung wird der angeforderte Prozesswechsel somit komplett ignoriert (die Aufnahme eines oder mehrerer Prozesse in den ↑Ablaufplan verfällt) oder erst verzögert wirksam (ein oder mehrere Prozesse wurden zwar eingeplant, aber der Einlastungswunsch nur vermerkt: „Software↑*latch*“). Der verzögerte Prozesswechsel geschieht dann im Moment der Aufhebung der entsprechenden Sperre. Die Technik ist einer ↑Unterbrechungssperre sehr ähnlich, jedoch wird nicht der ↑Arbeitsmodus der ↑CPU geändert, sondern der Modus, in dem das ↑Betriebssystem operiert.

Vererbung (en.) ↑*inheritance*. Mechanismus der ↑objektorientierten Programmierung zur Mitbenutzung (*sharing*) bestimmter ↑Ressourcen in einer Rangfolge, die eine bestimmte ↑hierarchische Struktur definiert. Diese Struktur ist entweder als ↑Objekt- oder ↑Klassenhierarchie

ausgeprägt, wobei dann eine *prototypische* beziehungsweise *klassenbasierte Vererbung* die Grundlage bildet. Erstere ist auch als *Delegation* bekannt.
Insbesondere die klassenbasierte Form ist ein Mechanismus zur Wiederverwendung von ↑Text und ↑Daten (*code/data reuse*), sie erlaubt zudem die Erweiterung von Altsoftware (*legacy software*) mittels öffentlicher ↑Klassen und Schnittstellen. Dabei wird eine erbende Klasse als Kind-, Unter- oder abgeleitete Klasse (*child class*, *subclass*, *derived class*) bezeichnet, die korrespondierende vererbende Klasse ist die Eltern-, Ober- oder Basisklasse (*parent class*, *super class*, *base class*). Diese Beziehung kann einfach oder mehrfach ausgelegt sein, das heißt, relativ zu einer Stufe in der Klassenhierarchie erbt eine Unterklasse die Merkmale (Text, Daten) immer nur genau einer Oberklasse (↑Java) oder möglicherweise auch mehrerer Oberklassen (↑C++). Entsprechend wird dann von Einfach- (*single*) und Mehrfachvererbung (*multiple inheritance*) gesprochen.
In Ober- und Unterklasse kann dieselbe ↑Signatur einer Methode spezifiziert sein, womit letztlich verschiedene Implementierungen derselben Schnittstelle möglich werden. Welche Implementierung bei einem Methodenaufruf dann zur Wirkung kommt, hängt ab von der Art der Methode. Handelt es sich etwa um die ↑virtuelle Methode einer Oberklasse, dann wird ihre Implementierung durch die Methode mit identischer Signatur in der Unterklasse ersetzt: beim Aufruf kommt dann die Implementierung zur Geltung, die der letzten Unterklasse mit identischer Methodensignatur angehört. Anderenfalls, wenn keine virtuelle Methode spezifiziert ist, existieren möglicherweise zugleich mehrere Implementierungen von Methoden derselben Signatur, wobei jede dieser Methodenimplementierungen dann weiterhin Geltung hat und durch entsprechende Aufrufe zur Ausführung kommen kann. In Java ist jede Methode eine virtuelle Methode, in C++ muss dazu die betreffende Methode mit dem Spezifikationssymbol `virtual` explizit ausgewiesen sein. Die Entscheidung, welche der virtuellen Methoden effektiv zur Wirkung kommen, fällt für gewöhnlich erst zur ↑Laufzeit (↑*dynamic binding*).

Verhungern (en.) ↑*starvation*. Bezeichnung eines Problems bei der ↑Ablaufplanung oder ↑Synchronisation, wodurch ein ↑Prozess die für sein weiteres Vorankommen erforderliche ↑Ressource fortwährend versagt bleibt. Der Prozess unterliegt keiner ↑Verklemmung, da er nicht blockiert, sondern laufend oder bereit ist (↑Prozesszustand). Er kommt nicht voran, obwohl die von ihm benötigte Ressource zeitweilig frei ist, sie ihm jedoch von einem anderen Prozess immer wieder weggeschnappt wird, bevor er sie für sich beanspruchen kann.
In Bezug auf bestimmte Implementierungsvarianten einer ↑Umlaufsperre wird dieses Problem auch als ↑*after you, after you*-Blockierung bezeichnet.

verkettete Liste (en.) ↑*linked list*. Eine Datenstruktur, die eine für gewöhnlich dynamische Folge von Elementen in linearem ↑Speicherbereich verwaltet. Die Reihenfolge der Elemente wird durch ↑Zeiger oder ↑Referenzen festgelegt, die auch als Kettenglied (*chain link*) bezeichnet werden. Eine solche Liste ist typischerweise einfach oder doppelt verkettet, das heißt, jedes Element enthält entweder nur ein Kettenglied, um nur auf das Nachfolgerelement (*successor*) zu verweisen oder zwei Kettenglieder, um sowohl auf das Nachfolger- als auch auf das Vorgängerelement (*predecessor* zu verweisen.

Exkurs Nachfolgend ein Beispiel in ↑C, programmiersprachlich eine einfache (als Verbund repräsentierte) *rekursive Datenstruktur*, die entweder kein Listenelement oder eine Folge von Listenelementen desselben ↑Datentyps (`chain_t`) enthält:

```
typedef struct chain {
    struct chain *link;
} chain_t;
```

Das Ende einer solcher Folge wird durch einen bestimmten Zeiger-/Referenzwert (des Verbundmitglieds `link`) im letzten Kettenglied der Liste markiert, gemeinhin `NULL` in C.

Auf Grundlage eines so definierten Kettenglieds lassen sich elementare Operationen bilden, mit denen bereits komplexe dynamische Datenstrukturen (z.B. ↑Schlangen) aufgebaut und verwaltet werden können. Etwa eine Operation zum Verketten zweier Listenelemente:

```
inline chain_t *join(chain_t *this, chain_t *item) {
    item->link = this->link;          /* let item refer to this' successor */
    return this->link = item;         /* let this' successor refer to item */
}
```

Zeigt `this` auf den Kopf einer leeren oder gefüllten Liste, dann wird das Element, auf das `item` zeigt, zum ersten beziehungsweise vordersten Eintrag in dieser Liste. Eine Anweisung der Art `list.link = join(&list, &elem)` platziert sodann das Element `elem` an den Listenanfang und aktualisiert den Zeiger darauf.
Die inverse Operation, nämlich ein Element aus der Liste auszutragen, zeigt folgendes Beispiel:

```
inline chain_t *skip(chain_t *this) {
    chain_t *item = this->link;           /* the chain link to be skipped */
    if (item != 0)                                     /* any successor? */
        this->link = item->link;                  /* yes, skip this entry */
    return item;                                 /* deliver removed entry */
}
```

Praktisch ist auch eine Operation, die eine Liste in zwei Teile zerlegt, sie an einer vorgegebenen Stelle trennt. Nachfolgend eine mögliche Implementierung:

```
inline chain_t *part(chain_t *this) {
    chain_t *tail = this->link;   /* beginning of the list to be detached */
    this->link = 0;                     /* termination of the origin list */
    return tail;                             /* deliver back list section */
}
```

Die Operation geht davon aus, dass der Kopf der Ursprungsliste im aufrufenden Kontext weiterhin bekannt ist. Des Weiteren nimmt sie der Einfachheit halber bewusst an, wie auch die beiden vorigen Operationen, dass der applizierte Zeiger auf das jeweils zu manipulierende Kettenglied gültig (d.h., ungleich `NULL`) ist. Die Gültigkeit des betreffenden Zeigers lässt sich leicht im aufrufenden Kontext der jeweiligen Operation sicherstellen beziehungsweise überprüfen, bevor die empfindlichen Speicherzugriffe zur Wirkung kommen.
So könnte die Stelle, an der die eben betrachtete Aufspaltung einer Liste geschehen soll, zuvor durch die Suche nach einem bestimmten Element lokalisiert worden sein. Eine solche Operation, mit der in einer gegebenen Liste ein Element anhand eines bestimmten Attributs gesucht werden kann, zeigt folgendes Beispiel:

```
inline chain_t *find(chain_t *this, chain_t *item) {
    chain_t *node = this;           /* start at the beginning of the list */
    while (node & node->link != item)            /* matching entry found? */
        node = node->link;                   /* no, investigate successor */
    return node;                  /* point to found entry or deliver NULL */
}
```

Die Anweisung `find(&list, NULL)` liefert dann entweder den letzten Eintrag in einer gefüllten Liste oder den Zeiger auf das Kopfelement einer leeren Liste. Konnte das Attribut `item` nicht gefunden werden, liefert die Operation `NULL`.

Verklemmung (en.) ↑*deadlock*. Bezeichnung für die gegenseitige Blockierung gleichzeitig, aber unabhängig voneinander stattfindender ↑Prozesse (in Anlehnung an den Duden). VL 11

Verklemmungsvermeidung (en.) ↑*deadlock avoidance*. Maßnahmen zum Vermeiden einer ↑Verklemmung durch *analytische Methoden*; es nicht zu einer Verklemmung kommen lassen, ihr aus dem Wege gehen (in Anlehnung an den Duden). Nicht zu verwechseln mit ↑Verklemmungsvorbeugung, die konstruktive Methoden zur Verhütung von Verklemmungen nutzt. VL 11

Verklemmungsvorbeugung (en.) ↑*deadlock prevention*. Maßnahmen zur Verhütung einer ↑Verklemmung durch *konstruktive Methoden*; Prophylaxe (in Anlehnung an der Duden). Nicht zu verwechseln mit ↑Verklemmungsvermeidung, die analytische Methoden zum Vermeiden

von Verklemmungen nutzt.

VL 11

Verknüpfung (en.) ↑*hard link*. Paarung von ↑Dateiname und ↑Indexknoten durch eine ↑numerische Adresse (↑*inode number*), gespeichert im ↑Verzeichniseintrag für die benannte ↑Datei. Damit kann weder eine Beziehung zu einem ↑Verzeichnis auf demselben noch zu einer Datei auf einem anderen (an einen ↑Befestigungspunkt eingehängtes) ↑Dateisystem hergestellt werden. Die zu assoziierende Datei ist eine ↑Entität desselben Dateisystems.

Verlagerung (en.) ↑*relocation*. Bezeichnung für den Vorgang, ein ↑Maschinenprogramm mit einer ↑Ladeadresse zu versehen und damit elementaren Programmbestandteilen (z.B. ↑Segmenten, ↑Seiten oder ↑Worten, genauer gesagt ↑Maschinenbefehlen, ↑Variablen oder Konstanten) ↑absolute Adressen zuzuordnen. Bestimmte ↑Referenzen innerhalb des Maschinenprogramms gleichen ↑Ladedistanzen, um die der Wert der Ladeadresse *vor* beziehungsweise *zur* ↑Laufzeit des Programms zu korrigieren ist — als wenn das referenzierte Etwas von einem bisher innegehabten Ort an einen anderen Ort gelegt wurde. Die Ladedistanz gibt die ↑Verschiebung zu der durch die Ladeadresse festgelegten Basis des Maschinenprogramms an.
Geschieht das Verlagern vor Laufzeit des Maschinenprogramms, *statisch*, also zur ↑Bindezeit oder zur ↑Ladezeit, unterliegt jede Referenz der Korrektur durch ein ↑Dienstprogramm. Für gewöhnlich erfolgt diese Korrektur auf ↑Befehlssatzebene. Die Korrekturstellen sind in der von einem ↑Assemblierer pro ↑Objektmodul anteilig erzeugten und von einem ↑Binder für das ↑Lademodul aus den Anteilen zusammengestellten ↑Verlagerungstabelle verzeichnet. An diesen Stellen wird die dort benötigte absolute Adresse als Summe von Ladeadresse und Ladedistanz eingetragen. Zur Ladezeit sorgt ein ↑verschiebender Lader für die Korrektur.
↑Bezugspunkt für den statischen Ansatz ist die sogenannte ↑Verlagerungskonstante, die eine *feste Ladeadresse* für das Maschinenprogramm vorgibt. Alle als Ladedistanzen ausgewiesene Referenzen innerhalb des Maschinenprogramms bilden relative Adresswerte zu dieser Basis. Damit ist die absolute Adresse a einer solchen Referenz bestimmt durch:

$$a = \mathcal{B} + d,$$

mit der durch die Verlagerungskonstante vorgegebenen Basis $\mathcal{B}$ und der Verschiebung d zu dieser Basis beziehungsweise der absoluten Ladeadresse des ersten Worts ($\mathcal{B}$) des Maschinenprogramms und der Ladedistanz (d) dazu.
Für gewöhnlich ist den betreffenden Referenzen in dem Maschinenprogramm jeweils ein ↑Symbol zugeordnet, sie haben ↑symbolische Adressen (bspw. der ↑Name einer Variablen oder Konstanten, eines ↑Unterprogramms oder eine ↑Sprungmarke), die vom Assemblierer in relativ zur Basis eines bestimmten ↑Programmsegments ausgerichtete ↑numerische Adressen aufgelöst werden. Der Wert einer solchen numerischen Adresse (↑*displacement*) wird zusammen mit dem Symbol als Paar in der vom Assemblierer pro Objektmodul erzeugten ↑Symboltabelle abgelegt. Bei Zusammenstellung des Lademoduls, das heißt, (1) dem Zueinanderstellen der in den Objektmodulen enthaltenden Texte und Daten zu Programmsegmenten und (2) der Anordnung dieser Segmente passend zum Modell des ↑Prozessadressraums des ↑Betriebssystems, ändern sich die relativen Positionen der einzelnen Bestandteile des Maschinenprogramms. Dies betrifft sowohl den inneren Aufbau der Programmsegmente als auch die Programmsegmente untereinander.
Indem die Text- und Datenbestände aller benötigten Objektmodule zueinander angeordnet werden, vergrößern sich die Ladedistanzen für die weiter hinten platzierten Elemente entsprechend um die Längen der vorne platzierten Einheiten. In der als Bestandteil des Lademoduls bei dem Vorgang neu entstehenden Symboltabelle wird daher die mit einem Symbol verknüpfte numerische Adresse entsprechend aktualisiert, und zwar nach Platzierung der dieses Symbol definierenden Einheit auf die neue Ladedistanz gesetzt. Wurden alle Einheiten platziert, ist jede zu einem Symbol verzeichnete Ladedistanz ein Verschiebungswert zum Anfang des Programmsegments passend zur Kategorie (Text, Daten, BSS) des Symbols. In einem weiteren Schritt wird die Verlagerungskonstante als Basisadresse zu allen verzeichneten Ladedistanzen des ersten Programmsegments addiert, um damit die absoluten

Adressen zu vergeben. Für gegebenenfalls nachfolgende Programmsegmente wird ebenso verfahren, allerdings erhöht sich die Basisadresse für diese schrittweise um die Länge des jeweils vorangegangenen Segments. Dieser abschließende Schritt zur Festlegung absoluter Adressen im Maschinenprogramm erfolgt entweder noch zur Bindezeit oder erst zur Ladezeit.
Geschieht das Verlagern zur Laufzeit des Maschinenprogramms, *dynamisch*, unterliegt jede Referenz der Korrektur durch den ↑Prozessor, genauer gesagt einer↑MMU — wenn ↑positionsunabhängiger Kode, der kein Beispiel für die hier betrachteten Verlagerungsmethoden ist, einmal außer Acht gelassen wird. Vielmehr beruht der Vorgang in dem Fall auf eine geeignete Methode der ↑Adressabbildung, nämlich ↑Segmentierung oder ↑Seitenadressierung beziehungsweise Kombinationen (↑*paged segmentation*, ↑*segmented paging*) davon. Referenzen, deren Bezugsrahmen ein ↑logischer Adressraum ist, werden damit verlagert auf eine tiefere Ebene, für die nur noch ein ↑realer Adressraum definiert ist.
Mit Segmentierung als Grundlage erhält jedes Programmsegment ein eigenes ↑Adressraumsegment. Innerhalb dieses Segments sind alle vom Binder bestimmten Ladedistanzen zu den Referenzen gewöhnliche Verschiebungen zur Segmentbasis. Im betreffenden ↑Segmentdeskriptor muss das Betriebssystem lediglich die Ladeadresse des Segments als Basiswert eintragen. Bei jedem ↑Befehlszyklus wird dann die (segmentierte) MMU Referenzen, die jetzt nämlich ↑logische Adressen darstellen, verlagern und die benötigte absolute, ↑reale Adresse generieren (vgl. S. 284).
Mit Seitenadressierung als Grundlage geschieht das Verlagern von Referenzen in durchaus ähnlicher Art und Weise, nur dass die Ladeadresse der ↑Seite, auf der die fragliche Referenz nunmehr liegt, einem ↑Seitendeskriptor entnommen wird (vgl. S. 286). Auch hierzu hat das Betriebssystem die Ladeadresse der betreffenden Seite, das heißt, die reale Adresse des zugeordneten ↑Seitenrahmens, zuvor in den Seitendeskriptor als Basiswert eingetragen. Im Unterschied zur Segmentierung müssen allerdings zusätzlich noch die Programmsegmente auf den eindimensionalen Adressraum abgebildet worden sein. Dies bedeutet, vom Anfang eines Prozessadressraums ausgehend, die Programmsegmente nacheinander auf ein ganzzahlig Vielfaches von Seiten entsprechend der jeweiligen Segmentgröße zu verteilen. Dabei ist der Anfang des Prozessadressraums definiert durch die Verlagerungskonstante eines (eindimensionalen) logischen Adressraums, deren Wert für gewöhnlich durch das Betriebssystem vorgegeben ist. Diese Verteilung der Programmsegmente innerhalb eines solchen logischen Adressraums sowie die Generierung korrekter absoluter Adressen für diesen Adressraum geschieht in aller Regel bereits vor Laufzeit, nämlich, wie für den statischen Fall oben beschrieben, als Funktion eines Binders oder verschiebenden Laders. Im Unterschied zum statischen Fall sind die absoluten Adressen hier jedoch logische Adressen von Referenzen, die schließlich in den realen Adressraum verlagert werden.

Verlagerungskonstante (en.) ↑*relocation constant*. Festwert für die Justierung von ↑Text oder ↑Daten beim ↑Binden oder ↑Laden. Bezugnehmend auf diesen Wert wird bei der ↑Verlagerung jede in einem ↑Maschinenprogramm verwendete ↑Referenz auf eigene Programmstellen korrigiert, so dass danach eine gültige ↑absolute Adresse entsteht. Die Korrekturstellen sind in der beim ↑Assemblieren erzeugten ↑Verlagerungstabelle verzeichnet.
Der Wert dieser Konstante hängt vor allem davon ab, in welcher Art von ↑Adressraum ein Maschinenprogramm ablaufen soll. Ist durch die ↑Betriebsart ein ↑realer Adressraum vorgegeben, definiert die Konstante die ↑reale Adresse im ↑Hauptspeicher, an der das Programm geladen werden kann. Wird dagegen ein ↑logischer Adressraum oder ↑virtueller Adressraum durch die Betriebsart vorgegeben, bestimmt das ↑Betriebssystem den Wert dieser Konstanten, die dann die ↑logische Adresse beziehungsweise ↑virtuelle Adresse vorgibt, an der das Programm im ↑Prozessadressraum liegen soll.

Verlagerungstabelle (en.) ↑*relocation table*. Liste von ↑Zeigern auf die zu ändernden Stellen im ↑Objektkode, um ein ↑Maschinenprogramm vom ↑Binder zusammenzufügen, zu binden, oder durch den ↑Lader nachträglich noch verschieben zu können (↑*relocation*). Jede dieser Stelle gibt an, wo die ↑Ladeadresse eines referenzierten Programmelements (d.h., eines ↑Ma-

schinenbefehls, einer ↑Variablen oder Konstanten) einzutragen, eine ↑absolute Adresse zu korrigieren ist.

Versagen (en.) ↑*failure*. Bezeichnung für eine bestimmte Art von ↑Gefahr, die als Folge eines ↑Fehlers zum Fehlverhalten im System führt und den Ausfall des Systems hervorrufen kann. Endglied in einer ↑Gefahrenkette.

Versatz (en.) ↑*offset*. Für gewöhnlich eine Ganzzahl, die den Abstand zwischen einem ↑Bezugspunkt und einem beliebigen anderen Punkt innerhalb desselben Bezugsrahmens kennzeichnet. Typisches Beispiel ist die Distanz zu einer bestimmten ↑Speicherzelle innerhalb einer ↑Seite, ausgehend vom Seitenanfang (↑*page offset*). Sei 2^n die Anzahl der Speicherzellen pro Seite, dann ist der Wertebereich dieser Zahl $o = [0, 2^n - 1]$.

Hinweis Eine ähnliche Bedeutung hat die ↑Verschiebung, das heißt, wenn ein ↑segmentierter Adressraum und dort die Lokalisierung einer Speicherzelle innerhalb eines ↑Segments betrachtet wird. Verschiedentlich werden beide Arten des Abstands einer Speicherzelle vom Anfang ihres Bezugsrahmens (Seite oder Segment) daher auch synonym zueinander verwendet. Allerdings hat der Bezugsrahmen einerseits eine feste Größe (Seite) und andererseits eine variable Größe (Segment), zudem ist die Größe einer Seite im Vergleich zur maximalen Größe eines Segments gemessen an der Anzahl von Speicherzellen sehr unterschiedlich: bezugnehmend auf einen Wertebereich $[0, 2^n - 1]$ gilt für eine Seite $9 \leq n \leq 30$ und für ein Segment $32 \leq n \leq 64$, je nach zugrundeliegender Hardware.

verschiebender Lader (en.) ↑*relocating loader*. ↑Lader, der beim ↑Laden von einem ↑Maschinenprogramm zugleich für die ↑Verlagerung von eben diesem ↑Programm sorgt.

Verschiebung (en.) ↑*displacement*. Eine natürliche Zahl, die für ein bestimmtes ↑Segment den Abstand einer ↑Speicherzelle vom Segmentanfang (d.h., relativ zur ersten Speicherzelle des Segments) kennzeichnet. Sei N die maximale Anzahl von Speicherzellen, die ein Segment enthalten kann. Dann ist der Wertebereich dieser Zahl $d = [0, N - 1]$, wobei N durch die ↑Wortbreite bestimmt ist.

Verschmelzung Vereinigung von einem frei werdenden ↑Adressbereich mit einem oder zwei angrenzenden Adressbereichen zu einem einzigen großen Adressbereich, wobei die Länge eines jeden dieser Bereiche bekannt ist. Optionales Merkmal von einem ↑Platzierungsalgorithmus, um den Grad von ↑Fragmentierung vermindern zu können. Ein frei werdender ↑Speicherbereich ist seiner Länge (↑*best fit*/↑*worst fit*) oder ↑Adresse (↑*first fit*/↑*next fit*) nach auf die Liste freier Speicherabschnitte (↑*hole list*) zu platzieren. Dabei wird überprüft, ab dieser Abschnitt direkt zu Abschnitten, die bereits auf der Liste stehen, benachbart ist. Liegt der durch einen ↑Prozess frei werdende Abschnitt ($hole_p$) im ↑Hauptspeicher direkt vor einem bereits auf der Liste stehenden Abschnitt ($hole_l$), gilt also $address(hole_p) + sizeof(hole_p) = address(hole_l)$, oder direkt danach, gilt also $address(hole_l) + sizeof(hole_l) = address(hole_p)$, wird der bereits auf der Liste stehende Abschnitt entsprechend um die Länge des frei werdenden Abschnitts korrigiert: ein größerer freier Abschnitt wird erzeugt und es kommt kein weiterer Listeneintrag hinzu. Liegt der frei werdende Abschnitt genau zwischen zwei bereits auf der Liste stehenden Abschnitte, kommt es zum Lückenschluss: ein noch größerer freier Abschnitt wird erzeugt und es wird ein Listeneintrag gestrichen. Diese Maßnahme ist einfach durchzuführen, wenn die Listeneinträge der Adresse nach sortiert sind. In dem Fall ändert sich nicht die Listenposition des Eintrags, sondern lediglich der im Eintrag verzeichnete Längenwert. Sind die Listeneinträge jedoch der Größe nach sortiert, erzeugt ein Verschmelzungsschritt einen größeren freien Abschnitt, der in einem zweiten Suchlauf wieder einzusortieren ist, um dabei durch einen weiteren möglichen Verschmelzungsschritt zu einem noch größeren Abschnitt zu werden, der schließlich einen dritten Suchlauf zum Einsortieren nach sich zieht.

Verschnitt (en.) ↑*waste*. Abfall, bei der ↑Speicherzuteilung anfallender Rest. Typischerweise gibt ein ↑Prozess die Anzahl von ↑Byte an, die ein ihm zuzuteilender ↑Speicherbereich umfassen

soll. Wenn die Speicherzuteilung jedoch nicht byteorientiert arbeitet, weil die Größe ihrer kleinsten Verwaltungseinheit (↑Speicherwort, ↑Seite) ein Vielfaches der Größe eines Byte ausmacht, erhält der Prozess einen größeren Bereich als angefordert zugeteilt. In aller Regel bleibt der überschüssige Anteil in diesem Speicherbereich von dem Prozess ungenutzt (↑interne Fragmentierung). Arbeitet die Speicherzuteilung jedoch byteorientiert und teilt sie einem Prozess einen Speicherbereich zu, indem sie ein verfügbares ↑Loch ausreichender Größe verkleinert aber nicht eliminiert, bleibt ein kleineres Loch übrig, das gegebenenfalls für weitere Zuteilungen unbrauchbar ist (↑externe Fragmentierung).

verteilter gemeinsamer Speicher (en.) ↑*distributed shared memory.* ↑Arbeitsspeicher, der innerhalb derselben Ebene der ↑Speicherhierarchie über mehrere ↑Rechner verteilt vorliegt. Typischerweise ist dieser Arbeitsspeicher als auseinanderliegender globaler ↑Hauptspeicher organisiert, dessen Einzelteile die lokalen Hauptspeicher verschiedener Rechner bilden. Zugriffe auf einen entfernten ↑Speicherbereich sind nur über ein ↑Rechnernetz möglich und für gewöhnlich mit abgeschwächter ↑Speicherkonsistenz behaftet.
Mittels ↑Speichervirtualisierung können die verteilten Hauptspeicher in logischer Hinsicht als Einheit durch ein ↑Betriebssystem zur Verfügung gestellt werden. Grundlage dafür ist ein hinreichend großer ↑seitennummerierter Adressraum, über den die im Rechnernetz verfügbaren Hauptspeicher zugänglich gemacht werden können (↑PGAS). Die ↑Umlagerung von Speicherinhalten, ausgelöst durch einen ↑Seitenfehler beim Zugriff auf einen entfernten Hauptspeicher, geschieht sodann nicht zwischen verschiedenen Ebenen (d.h., ↑Vordergrundspeicher und ↑Hintergrundspeicher) der Speicherhierarchie, sondern innerhalb derselben Hauptspeicherebene.

Verwaltungsdaten Bezeichnung für ↑Daten, die zur Bewirtschaftung von ↑Nutzdaten erforderlich sind (↑*overhead*).

Verzeichnis (en.) ↑*directory.* Ordner; nach einem bestimmten System geordnete Aufstellung mehrerer unter einem bestimmten Gesichtspunkt zusammengehörender ↑Daten (in Anlehnung an den Duden). Die Daten sind in einer ↑Datei erfasst, deren ↑Name in dem Ordner verzeichnet ist. In Bezug auf diesen Namen definiert der Ordner gleichsam einen ↑Namenskontext, der die ↑Bindung zu einer Dateidatenstruktur beschreibt (↑*directory entry*). Ein solcher Ordner ist letztlich selbst eine Datei, die im ↑Dateisystem zur Speicherung eben dieser Bindungen vorgesehen ist. Die genaue Auslegung dieser speziellen Datei gibt das ↑Betriebssystem vor. Im Falle von ↑UNIX sind in jeder solchen Datei zwei vordefinierte Verzeichnitseinträge vorgesehen: ein Eintrag (↑*dot*) definiert die Bindung zur Ordnerdatei selbst und ein zweiter Eintrag (↑*dot dot*) verweist auf den Elterordner. Ansonsten werden keine weiteren Vorgaben gemacht.

Verzeichniseintrag (en.) ↑*directory entry.* Eintrag in einem ↑Verzeichnis, der die ↑Bindung zu einer namentlich identifizierten Dateidatenstruktur beschreibt (↑*hard link*). Dieser Eintrag bildet die ↑symbolische Adresse einer ↑Datei ab auf die ↑numerische Adresse der dieser Datei im ↑Dateisystem zugeordneten Datenstruktur: wesentliches Merkmal dabei ist das Tupel $(Symbol, Adresse)$, wobei ↑Adresse (im Falle von ↑UNIX) eine ↑Indexknotennummer ist. Die Datenstruktur entspricht dem auf dem ↑Datenträger daselbst gespeicherten ↑Deskriptor (↑*inode*) für diese Datei.

Verzögerungsschleife (en.) ↑*delay loop.* Folge von Anweisungen oder Befehlen eines ↑Programms, die mehrmals hintereinander durchlaufen werden kann (Duden), um gezielt die Verzögerung des zugehörigen ↑Prozesses zu bewirken. Der Prozess hält für eine gewisse Zeit inne (↑*active waiting*), bleibt in seinem ↑Warteverhalten beschäftigt (↑*busy waiting*).

Exkurs Zweck der Anweisungen ist es, dass sich ein Prozess selbst aufhält (↑*backoff*). Für gewöhnlich geschieht dies durch eine bestimmte Anzahl von Schleifendurchläufen, in denen der Prozess nichts tut außer Zeit verstreichen zu lassen, er verweilt beim Zählen (`dwell`):

```
inline void dwell(/*volatile*/ long term) {                    /* delay loop */
    while (term--);                                  /* # of passes to be done */
        ease();                                                  /* relax CPU */
}
```

Die Kopfschleife zählt die zu verstreichende Zeit (`term`) und gibt dem Prozess im Falle einer Restzeit die Möglichkeit, sich im Schleifenrumpf zu entspannen (`ease`):

```
inline void ease() {                                             /* relax CPU */
    asm("pause": : : "memory");                        /* give spin loop hint */
}
```

Der verwendete, hier für ↑x86 definierte Spezialbefehl (`pause`) weist den ↑Prozessor auf einen sich im Kreiseln (*spin*) befindlichen Prozess hin und ermöglich dadurch bestimmte prozessorinterne Optimierungsmaßnahmen, beispielsweise eine Verletzung der Speicherzugriffsreihenfolge (*memory ordering*) zu vermeiden oder die Leistungsaufnahme (*power consumption*) zu reduzieren. Alle Aktionen der Schleife beanspruchen normalerweise nur den ↑Zwischenspeicher, gehen somit nicht über den Bus und sorgen dadurch für eine „logische Abkopplung“ des Prozessors aus Sicht des zurückweichenden Prozesses.
Sollte der Prozessor keinen vergleichbaren Spezialbefehl besitzen, hat die „Entspannungsoperation“ (`ease`) keinen Effekt, sie ist in dem Fall leer, enthält keine ↑Maschinenbefehle:

```
inline void ease() {}                                          /* no operation */
```

Um dann aber sicherzustellen, dass der ↑Kompilierer überhaupt Maschinenbefehle für die Verweiloperation (`dwell`) eines Prozesses und insbesondere die diese Operation bestimmende Schleife erzeugt, gilt der tatsächliche Parameter (`term`) als „unberechenbar“ (`volatile`). Im gegebenen Fall erzwingt diese Maßnahme überhaupt die Schleifengenerierung, obwohl die zugehörigen Anweisungen den Berechnungszustand des betreffenden Prozesses unverändert lassen und funktional (anscheinend) nichts bewirken. Die sonst übliche Optimierung des Kompilierers wird durch diesen Trick für dieses ↑Unterprogramm praktisch außer Kraft gesetzt. Dies bedeutet aber andererseits, dass bei Verwendung eines Spezialbefehls wie oben gezeigt, dieser Trick wegen der inzeilig abgesetzten ↑Assemblieranweisung (`asm`) unnötig ist. Daher ist in dem Fall die Typspezialisierung (`volatile`) in der ↑Signatur verzichtbar.
In dem Beispiel bestimmt ein ↑tatsächlicher Parameter (`term`) die dem Prozess eigene Verzögerung. Der Parameterwert kann einer vorher bestimmten relativen Zeitspanne entsprechen, die die ↑Ausführungszeit eines einzelnen Schleifendurchlaufs als Maßeinheit zu Grunde legt. Diese Maßeinheit ist nicht nur abhängig von der Geschwindigkeit, mit der die ↑CPU operiert (↑*processor clock*), sondern auch von der Anzahl der Maschinenbefehle, die der Kompilierer für einen einzelnen Schleifendurchlauf generiert.
Eine solche Verzögerung hat aber immer nur *lokale Signifikanz*, sie gibt einen unteren Grenzwert für die Zeit vor, für die der betreffende Prozess eigenständig innehält. Während der Prozess schleift, können dynamische Vorgänge von außen (↑*interrupt*, ↑*interference*) zu weiteren Verzögerungen führen. Exakte, absolute ↑Wartezeiten lassen sich darüber für einen Prozess nicht einstellen. Für gewöhnlich erfordern diese einen geeigneten ↑Zeitgeber.

virtual address (dt.) ↑virtuelle Adresse.

virtual address space (dt.) ↑virtueller Adressraum.

virtual function (dt.) ↑virtuelle Methode.

virtual machine (dt.) ↑virtuelle Maschine.

virtual memory (dt.) ↑virtueller Speicher.

virtualisation (dt.) ↑Virtualisierung.

virtualisation system (dt.) ↑Virtualisierungssystem.

Virtualisierung (en.) ↑*virtualisation*. Nachbildung einer (realen) Maschine oder Teile davon. Die komplette Nachbildung wird auch als ↑Vollvirtualisierung bezeichnet, ansonsten ist der Begriff ↑Teilvirtualisierung angebracht.
Grundsätzlich kann die Nachbildung allein durch einen in Software implementierten ↑Interpreter geleistet werden (↑CSIM). Dies wird für gewöhnlich jedoch nur praktiziert, wenn die nachgebildete Maschine (↑Prozessor) unterschiedlich ist von jenem Prozessor, auf dem dieser Interpreter zur Ausführung kommt. Die ↑JVM ist ein weit verbreitetes Beispiel dafür. In den anderen Fällen findet ein ↑Hypervisor Verwendung, der lediglich die ↑partielle Interpretation bestimmter Prozessorbefehle (↑sensitiver Befehl) vornimmt.

Virtualisierungssystem (en.) ↑*virtualisation system*. ↑Interpretersystem zur Nachbildung einer (realen) Maschine, typischerweise realisiert durch einen ↑VMM. Der VMM ist ein für eine ↑Wirtsmaschine bestimmtes Steuerprogramm, er erschafft die Ablaufumgebung für eine ↑virtuelle Maschine.
Unterschieden wird zwischen *Typ I* und *Typ II*. Ein Typ I VMM läuft auf der bloßen Hardware (Wirtsmaschine), direkt auf der ↑CPU, wohingegen ein Typ II VMM zusätzlich noch ein ↑Wirtsbetriebssystem benutzt. Zentrale Funktion von einem VMM ist es, die direkte Ausführung eines „störungsempfindlichen Befehls“ (↑sensitiver Befehl) durch die virtuelle Maschine zu verhindern. Ein solcher Befehl wird vom VMM abgefangen (↑*trap*) und speziell behandelt (↑partielle Interpretation).
Typisches Beispiel eines sensitiven Befehls ist die ↑Unterbrechungssperre. Wird diese durch eine ↑Aktion in einem ↑Gastbetriebssystem veranlasst, darf nur eine ↑Unterbrechungsanforderung an die virtuelle Maschine, die eben dieses Gastbetriebssystem ausführt, maskiert werden. In solch einem Fall fängt der VMM den entsprechenden ↑Maschinenbefehl (`cli`, x86) ab und unterbindet die mögliche ↑Unterbrechung einer auf der virtuellen Maschine stattfindenden Aktion. Der Zustand der ↑Wirtsmaschine bleibt diesbezüglich unverändert, das heißt, für sie gilt keine Unterbrechungssperre. Kommt der inverse Befehl (`sti`, x86) zur Ausführung durch die virtuelle Maschine, agiert der VMM dementsprechend.

virtuelle Adresse (en.) ↑*virtual address*. Bezeichnung einer ↑Adresse, die von der Lage eines ↑Speicherworts im ↑Arbeitsspeicher abstrahiert. So ist bei der Verwendung einer solchen Adresse immer davon auszugehen, dass der Inhalt des betreffenden Speicherworts möglicherweise nicht jederzeit im ↑Hauptspeicher residiert.
Im Grunde handelt es sich dabei um eine ↑logische Adresse (genauer: eine Erweiterung davon), die nämlich die wirkliche Lokalität (↑Vordergrundspeicher oder ↑Hintergrundspeicher) der durch sie bezeichneten ↑Speicherstelle im Arbeitsspeicher generalisiert. Wird eine solche Adresse von der ↑CPU im ↑Abruf- und Ausführungszyklus appliziert, kann ein Zugriff darüber den Hauptspeicher verfehlen (↑*mainstore miss*). Ein solcher Fehlzugriff (↑*trap*) führt jedoch lediglich dazu, dass der betreffende ↑Prozess unterbrochen und später wieder aufgenommen wird, nachdem das ↑Betriebssystem die ↑Teilinterpretation der abgefangenen Operation erledigt hat. Je nach ↑Operationsprinzip des Betriebssystems können über solche Adressen auf nahezu beliebige ↑Entitäten, die an ebenso beliebigen Stellen im ↑Rechensystem platziert sind, funktional transparent für einen Prozess zugegriffen werden.

virtuelle Maschine (en.) ↑*virtual machine*. Maschine, die nicht in Wirklichkeit existiert, aber echt (real) erscheint. Der ↑Prozessor dieser Maschine ist entweder ein ↑Interpreter oder ein ↑Übersetzer. Ersterer implementiert ein ↑Virtualisierungssystem bestimmter Art, das ein ↑Programm dieser Maschine direkt ausführen kann, wohingegen letzterer das auszuführende Programm in ein Programm einer anderen (realen/virtuellen) Maschine transformiert, um es durch diese ausführen zu lassen.

virtuelle Methode (en.) ↑*virtual function*, *virtual method*. Auch bezeichnet als *polymorphe Funktion*, die in verschiedenerlei Gestalt jeweils als ↑Unterprogramm in einem ↑Maschinenprogramm vorkommt, wobei alle ↑Exemplare der betreffenden Funktion dieselbe ↑Signatur haben. Diese Art von Methode ist verbreitet in der ↑objektorientierten Programmierung, sie

ermöglicht die *Redefinition* und damit Spezialisierung einer bereits existierenden Methodenimplementierung. Dabei ist die ursprüngliche (gemeinhin auch allgemeine) Fassung der Methode Attribut einer *Oberklasse* und die spezialisierte Fassung ist Attribut einer *Unterklasse*, wobei letztere dann durch ↑Vererbung von ersterer (direkt oder indirekt) abgeleitet wurde und beide damit über einen bestimmten Satz gemeinsamer Merkmale verfügen.
Gegebenenfalls wird in der Oberklasse nur die Signatur, aber keine Implementierung der Methode vorgegeben (*pure virtual function* in ↑C++). In dem Fall muss eine Unterklasse die Implementierung bereitstellen, um ein Objekt, das auch die Merkmale der Oberklasse tragen soll, überhaupt erzeugen zu können. Grundsätzlich geschieht die Zuordnung der Implementierung einer solchen reinen (*pure*) oder gewöhnlichen „gedachten" Methode zu einem Objekt immer im Moment der Objekterzeugung (↑*dynamic binding*).

virtueller Adressraum (en.) ↑*virtual address space*. Bezeichnung für einen ↑Adressraum, der von der Lage eines ↑Prozesses im ↑Arbeitsspeicher abstrahiert. Eine bestimmte Menge von Nummern, wobei jede einzelne davon eine ↑virtuelle Adresse repräsentiert.
Typischer Anwendungsfall für einen solchen Adressraum ist ↑virtueller Speicher, wodurch ein ↑Programm ablaufen kann, obwohl zur ↑Laufzeit nur Einzelbestandteile von ↑Text und ↑Daten davon im ↑Hauptspeicher vorzuliegen brauchen. Erst virtuelle Adressen erlauben die Abstraktion von der Lokalität von Informationseinheiten in Bezug auf die ↑Speicherhierarchie. Ein ↑logischer Adressraum hat diese Eigenschaft nicht, eine Adresse darin abstrahiert lediglich von der Lokalität solcher Einheiten innerhalb des Hauptspeichers: ein Programm muss hier also immer komplett im Hauptspeicher vorliegen, damit es ausgeführt werden kann. Zur Adressraumüberwachung nutzt das ↑Betriebssystem bei Bedarf den ↑Hauptspeicherfehlzugriff durch einen Prozess als Mechanismus. Dadurch erst kann es in den Prozess (ohne sein Wissen, aber dennoch zeitlich wahrnehmbar für ihn) eingreifen und nicht im Hauptspeicher eingelagerte Text- und Datenbestände automatisch nachladen.

virtueller Speicher (en.) ↑*virtual memory*. ↑Arbeitsspeicher, der „unecht" vorhanden ist, nicht in Wirklichkeit existiert, aber echt erscheint. Eine durch ein ↑Betriebssystem bereitgestellte Vorrichtung zum Speichern von Informationen, die anteilig auf den im ↑Rechensystem verfügbaren ↑Vordergrundspeicher (zur direkten Verarbeitung, ↑Hauptspeicher) und ↑Hintergrundspeicher (zur Zwischenspeicherung, ↑Ablage) gehalten werden. Dabei ist diese Aufteilung den ↑Prozessen in funktioneller Hinsicht unbewusst (↑*single-level store*).
Zunächst vorgelegt als Grundprinzip des Entwurfs (*design rationale*) für ein Rechensystem, dessen ↑Trommelspeicher für ein ↑Programm funktional transparent erscheint (Güntsch, Logischer Entwurf eines digitalen Rechengerätes mit mehreren asynchron laufenden Trommeln und automatischem Schnellspeicherbetrieb, Diss., 1956). Dieses Konzept sah vor, dass die Hardware eigenständig und automatisch die 100 ↑Speicherworte umfassenden Informationsblöcke zwischen dem Hauptspeicher (6 Blöcke) und der Trommel (1000 Blöcke) hin- und herbewegt. Erstmalig umgesetzt wurde diese Technik allerdings mit ↑Atlas, wobei die Transfers der Informationsblöcke jedoch nicht Hardware, sondern Software (*extracode*) regelte und damit den nach wie vor gültigen Standard setzte.

Vitruv (Marcus Vitruvius Pollio, 80–70 bis etwa 15 v. Chr.) altrömischer Baumeister, Bauingenieur, Architekt, Autor.

VM/370 Einplatz-/Einbenutzerbetriebssystem für eine ↑virtuelle Maschine im ↑Mehrprogrammbetrieb. Erste Installation 1972 (IBM System/370).

VMM Abkürzung für (en.) *virtual machine monitor*, (dt.) Steuerprogramm für eine ↑virtuelle Maschine, zentraler Bestandteil von einem ↑Virtualisierungssystem.

VMS Mehrplatz-/Mehrbenutzerbetriebssystem, Abkürzung für „*Virtual Memory System*", erste Installation 1977 (↑DEC VAX-11). Hatte maßgeblichen Einfluss auf die Entwicklung von ↑Windows NT, das ab 1988 unter derselben Leitung (David Cutler) entstand.

volatile register (dt.) ↑flüchtiges Register.

Vollvirtualisierung (en.) ↑*full virtualisation*. Form der ↑Selbstvirtualisierung: die ↑Virtualisierung erfolgt ausschließlich durch einen ↑Hypervisor. Im Unterschied zur ↑Paravirtualisierung ist keinem ↑Prozess, weder im ↑Maschinenprogramm noch im ↑Betriebssystem, die Tatsache bekannt, dass die Hardware (↑CPU, ↑Peripherie), auf die er stattfindet, virtualisiert wird. Dies setzt jedoch virtualisierbare Hardware voraus, insbesondere die Fähigkeit der CPU, dass ein ↑sensitiver Befehl abgefangen werden kann (↑*trap*) und dann durch den Hypervisor zur Ausführung kommt (↑partielle Interpretation). Eine Hardware gilt als voll virtualisierbar, wenn ausnahmslos jeder sensitive Befehl derart behandelt werden kann.

volume (dt.) ↑Datenträger.

von Neumann Architecture (dt.) ↑von Neumann Architektur.

von Neumann Architektur (en.) ↑*von Neumann Architecture*. Bezeichnung für eine ↑Rechnerarchitektur mit gemeinsamen Speicher- und Signalpfaden für Anweisungen (↑Text) und ↑Daten — im Gegensatz zur ↑Harvard-Architektur, bei der sich Programmanweisungen und Daten nicht denselben Speicher und nicht dieselben Signalwege teilen. Der wesentliche Aspekt dieser ↑Architektur war/ist die freie Programmierbarkeit des ↑Rechners über einen zentralen ↑Hauptspeicher. Ursprünglich mit dem ↑EDVAC verwirklicht, zuvor waren die ↑Programme der Rechner entweder „händisch verdrahtet“ (↑ENIAC) oder sie wurden während der Ausführung anweisungsweise von einem ↑Lochstreifen eingelesen (↑Z1, ↑Mark I.

Vorausladen (en.) ↑*anticipatory paging.*. VL 12.3

Vordergrundspeicher (en.) ↑*primary storage*. Gegenteil von ↑Hintergrundspeicher. ↑Hauptspeicher, auch ↑Primärspeicher. Gemeinhin flüchtiger ↑Speicher, der direkt über Lese-/Schreiboperationen der ↑CPU bedient wird.

Vorhersagbarkeit (en.) ↑*predictability*. Grad, in dem eine korrekte Prädiktion (Vorhersage) des Zustands, Verhaltens oder Platz-, Energie- oder Zeitbedarfs von einem ↑Rechensystem möglich ist, in qualitativer oder quantitativer Hinsicht. Diese unterliegt immer bestimmten und vorher zu treffenden Annahmen. Insbesondere für ↑Echtzeit sind dabei folgende Aspekte bedeutsam: die Granularität eines Termins und Laxheit (weich, fest, hart) einer ↑Aufgabe, die Striktheit (weich, fest, hart) eines Termins, Zuverlässigkeitsanforderungen, die Größe des Systems und der Grad an Interaktion (↑Koordination) zwischen den Komponenten, sowie die Charakteristik der Umgebung, in der das System operieren muss.

vorlaufende Planung (en.) ↑*off-line scheduling*. Modell der ↑Ablaufplanung, die ört- und zeitlich entkoppelt von den einzuplanenden ↑Prozessen geschieht: sie findet im Voraus statt und erzeugt einen gemeinhin statischen ↑Ablaufplan, der zu einem späteren Zeitpunkt vom ↑Betriebssystem abgearbeitet wird (↑*static scheduling*). Allerdings kann dieser Plan auch den Initialzustand einer ↑Bereitliste definieren, die dann durch eine anschließende ↑mitlaufende Planung aktualisiert und fortgeschrieben wird. In dem Fall wirken zwei ↑Planer: einer im Voraus (statisch) und ein anderer im Nachhinein (dynamisch). Anders als mitlaufende Planung ist im strikt statischen Ansatz der Planer nicht Bestandteil des Betriebssystems (örtlich entkoppelt), sondern getrennt davon verwirklicht als ↑Dienstprogramm, das zur Aufstellung des Ablaufplans und damit vor den eigentlich vorgesehenen Prozessen erst noch zur Ausführung gebracht werden muss (zeitlich entkoppelt).
Diese Form von ↑Planung ist geboten, wenn die Berechnungskomplexität zur Erstellung des Ablaufplans (für eine gegebene Menge von Prozessen) einen Einsatz im laufenden Betrieb impraktikabel macht. Planung verursacht ↑Gemeinkosten, die in Grenzen zu halten sind, um, entsprechend der jeweils geforderten ↑Betriebsart, die eigentlich zu verwirklichenden Prozesse der ↑Maschinenprogramme beziehungsweise die ↑Peripherie bedienen zu können: ein Betriebssystem ist für gewöhnlich nur ausnahmsweise aktiv, was insbesondere bedeutet, nicht beliebig viel ↑Rechenzeit für sich selbst zu beanspruchen. Darüber hinaus geschieht

Planung sehr oft vorlaufend, um jederzeit (möglichst exakte) Aussagen zur ↑Vorhersagbarkeit von Prozessen treffen zu können. Typischer Fall ist die ↑deterministische Planung. In allen Fällen ist zur Erstellung des Plans ↑Vorwissen unabdinglich.

Vorlaufzeit (en.) ↑*lead time*. Zeitspanne zwischen der Auslösung (↑*release time*) und dem tatsächlichen Beginn (↑*start time*) eines ↑Prozesses. Eine ↑Latenzzeit, die durch (a) die Summe der ↑Durchlaufzeiten aller bereits vorrangig anstehenden Prozesse und (b) das ↑Hintergrundrauschen im ↑Rechensystem bestimmt ist.

Vorrangplanung (en.) ↑*priority scheduling*. Bezeichnung für eine ↑Ablaufplanung, die ↑Prozesse entsprechend ihres jeweiligen ↑Rangs für die ↑Einlastung eines ↑Prozessors vorsieht. Durch den Rang wird die ↑Priorität eines Prozesses zum Ausdruck gebracht. Die Rangzuweisung an einen Prozess kann einmalig und damit fest sein (↑*off-line scheduling*) oder sie geschieht initial vor Laufzeit und wird zur Laufzeit entsprechend des jeweiligen Systemzustands aktualisiert (↑*off-line scheduling*).
Der Rang eines Prozesses definiert in erster Linie den Platz auf der ↑Bereitliste und damit eine, in Bezug auf die anderen dort verzeichneten Prozesse, bestimmte Stufe einer nach qualitativen Gesichtspunkten gegliederten Reihenfolge (Ordnung). Für gewöhnlich wird diese Reihenfolge beim Eingang (Eintreffen), also bei der Bereitsstellung eines Prozesses aufgebaut oder aktualisiert, um die relative Position des eingetroffenen Prozesses auf der Bereitliste festzulegen. Die Auswahl des nächsten Prozesses für die Einlastung geschieht sodann ausgehend vom Listenanfang. Im Gegensatz etwa zu ↑FCFS kommt es pro (ein- oder ausgehenden) Prozess damit zu einem *Suchlauf*, der je nach Suchverfahren und technischer Auslegung der Bereitliste entweder im Moment der Bereitstellung oder der Auswahl erfolgt. Ist die Bereitliste als ↑Vorrangwarteschlange ausgelegt, findet der Suchlauf für gewöhnlich bei der Bereitstellung eines Prozesses statt, so dass bei der Auswahl der höchstrangige Prozess immer dem Listenanfang einfach entnommen werden kann. Ist demgegenüber die Bereitliste als Feld (*array*) ausgelegt, geschieht der Suchlauf im Moment der Prozessauswahl. In dem Fall ist die Bereitstellung eines Prozesses eine sehr einfache ↑Aktion und bedeutet lediglich, das in dem betreffenden ↑Prozesskontrollblock, der in diesem Feld liegt, enthaltene Attribut zum ↑Prozesszustand auf <u>bereit</u> zu setzen.
In zweiter Linie bestimmt der Rang, ob der eintreffende Prozess den im Moment der Bereitstellung stattfindenden Prozess sogar vom Prozessor verdrängen darf. Dieser zusätzliche Aspekt kommt aber nur ins Spiel, wenn die rangbasierte Ablaufplanung gleichfalls als ↑präemptive Planung funktioniert: rangbasierte Ablaufplanung wirkt nicht automatisch verdrängend und präemptive Ablaufplanung verfährt nicht automatisch rangbasiert. So sind beispielsweise ↑VRR, ↑SPN, ↑SRP, ↑SRTF, ↑RM, ↑DM und ↑EDF alle rangbasiert, aber nur SRP, SRTF, RM, DM und EDF verdrängen einen nachrangigen Prozess auch vom Prozessor.

Vorrangsteuerung (en.) ↑*priority control*. Führung von ↑Prozessen gemäß der ihnen jeweils zugewiesenen statischen oder dynamischen ↑Prioritäten. Grundlage dafür bildet eine bestimmte Form der ↑Vorrangplanung.

Vorrangwarteschlange (en.) ↑*priority queue*. Bezeichnung einer ↑Warteschlange, deren Einträge entsprechend ihres ↑Rangs sortiert vorliegen. Der Rang eines jeweiligen Eintrags entspricht der ↑Priorität, nach der ein Eintrag in Relation zu den anderen Einträgen derselben Warteschlange verarbeitet werden soll. Ein Eintrag höherer Priorität steht weiter vorne, ein Eintrag niedrigerer Priorität steht weiter hinten auf dieser Warteschlange. Einträge gleicher Priorität sind dann für gewöhnlich entsprechend ihrer ↑Ankunftszeit zueinander in Relation gesetzt, so dass ein früher aufgenommener Eintrag in dieser Gruppe gleichrangiger Einträge auch zuerst verarbeitet wird (↑FCFS).
Jeder Eintrag auf dieser Wartschlange steht für eine bestimmte ↑Aufgabe, die von einem ↑Prozess zu erledigen ist. Diese Aufgaben können demselben oder verschiedenen Prozessen zur Erledigung aufgegeben sein, auch kann jede Aufgabe selbst einer eigenen ↑Prozessinkarnation entsprechen. Letztere Variante impliziert dann immer einen ↑Prozesswechsel, wenn derselbe ↑Prozessor mit einer Aufgabe aus dieser Warteschlange versorgt werden soll.

Vorübersetzung (en.) ↑*precompilation*. Form der ↑Kompilation, um eine für die schnelle ↑Interpretation in Software besser geeignete Repräsentation von einem ↑Programm zu erhalten.

Vorwissen (en.) ↑*prior knowledge*. Kenntnis von etwas haben, mit der (lat.) *a priori* zuverlässige Aussagen möglich sind. Im Fokus stehen einerseits ↑Prozessgrößen, insbesondere Zeitpunkte, -intervalle und ↑Fristen. Andererseits kann dieses Wissen bevorzugt auch Beziehungen zwischen ↑Prozessen und ↑Betriebsmitteln beschreiben und damit letztlich auch Prozessabhängigkeiten repräsentieren. In all diesen Fällen sind die Informationen jedoch im Voraus bekannt und fließen als Parameter in die ↑Ablaufplanung ein.
Diese Größen und logischen Verknüpfungen sind dem ↑Planer „von außen“ vorgegeben, sie werden nicht zur ↑Laufzeit der zu planenden Prozesse erfasst. Sie beruhen auf Erfahrungswissen, empirische Untersuchungen, entwurfsbegleitende Modellanalyse, werkzeuggestützte Programmflussanalyse, messbasierte Ansätze (in Bezug auf Testläufe) oder Kombinationen davon. Kenntnis von solchen Informationen zu besitzen, ist obligatorisch für ↑deterministische Planung und optional, aber wünschenswert, für jede andere Form von ↑Planung.

VRR Abkürzung für (en.) *virtual* ↑*round robin*. Bezeichnung für eine Strategie zur ↑Planung der Bearbeitung von ↑Aufgaben durch einen ↑Prozessor im ↑Rundlaufverfahren mit Bevorzugung interaktiver ↑Prozesse. Eine Spezialisierung von ↑RR, mit der dem noch (in abgeschwächter Form) verbleibenden ↑Konvoieffekt vorgebeugt werden kann. Dazu wird neben der ↑Bereitliste eine weitere ↑Warteschlange eingeführt, in die Prozesse gelangen, die:

1. ihren letzten ↑Rechenstoß beendeten, indem sie einen ↑Ein-/Ausgabestoß, dessen Beendigung sie erwarten müssen, ausgelöst haben und
2. durch Beendigung ihrer ↑Ein-/Ausgabe wieder bereitgestellt wurden.

Die Prozesse in dieser zusätzlichen Warteschlange sind also nicht durch Ablauf ihrer ↑Zeitscheibe zur Abgabe des ↑Prozessors gezwungen wurden, sie gelten daher nicht als rechenintensiv (↑CPU *bound*), sondern sind vermutlich ein-/ausgabeintensiv (↑I/O *bound*). Steht erneut die ↑Einlastung an, wird zunächst versucht, aus der Warteschlange ein-/ausgabeintensiver Prozesse die nächste Aufgabe für den Prozessor zu entnehmen. Ist diese *Vorzugsliste* leer, kommt die nächste Aufgabe aus der Bereitliste und damit dann ein als vermutlich rechenintensiv eingestufter Prozess zur Wirkung.
Einziges Moment, durch das die ↑Verdrängung eines stattfindenden Prozesses erreicht wird, ist der mittels einer ↑Unterbrechungsanforderung angezeigte Zeitscheibenablauf. Folge der diesbezüglichen ↑Unterbrechungsbehandlung wird die Verdrängung des unterbrochenen Prozesses sein, wenn wenigstens ein anderer Prozess auf der Vorzugs- oder Bereitliste steht. Die das Ende einer Ein-/Ausgabe anzeigende Unterbrechungsanforderung führt demgegenüber nicht zur zeitnahen Einlastung des betreffenden Prozesses, sondern lediglich zu seiner ↑Einplanung unter Verwendung der Vorzugsliste. Frühestens mit Ablauf der aktuellen Zeitscheibe wird diesem Prozess dann der Prozessor zugewiesen, nämlich wenn dem Prozess bei der Einplanung die höchste ↑Dringlichkeit zuerkannt wurde.
Die Bearbeitung der Aufgaben in der Bereitliste erfolgt strikt nach RR, ihre Reihungstrategie ist ↑FCFS und benötigt daher eine konstante ↑Laufzeit für das Einsortieren. Demgegenüber sind die Aufgaben in der Vorzugsliste entsprechend der Restdauer der Zeitscheibe des jeweils zugehörigen Prozesses geordnet, womit einsortieren einen linear ansteigenden Laufzeitaufwand mit sich bringt. Für beide Listen gilt jedoch, dass die jeweils nächste Aufgabe am Kopfende (d.h., vorne) und damit in konstantem Aufwand entfernt wird.
Unabhängig davon, welcher der beiden Listen die nächste Aufgabe beziehungsweise der zugehörige Prozess entstammt, der Prozessor wird nur maximal eine Zeitscheibe lang für die Aufgabenbearbeitung belegt: kein Prozess kann daher den Prozessor bewusst oder unbewusst monopolisieren (↑CPU *protection*). Wurde der Prozess der Bereitliste entnommen, erhält er die ganze Zeitscheibe an Rechenzeit zugeteilt. Kam der Prozess allerdings aus der Vorzugsliste, belegt er den Prozessor längstens für die Restdauer seiner „angebrochenen“ Zeitscheibe.

Der ↑Zeitgeber wird, im Gegensatz zu RR, also nicht mit einem konstanten Wert für alle Prozesse programmiert, sondern mit einem variablen Wert, der prozessabhängig ist und damit ein Attribut des jeweiligen ↑Prozesskontrollblocks bildet. Grundsätzlich gilt: Wann immer ein Prozess durch Ablauf seiner (vormals ganzen oder zerstückelten) Zeitscheibe eine Unterbrechung erfährt und dadurch zur Prozessorabgabe gezwungen wird, so kommt er zurück auf die Bereitliste.

Exkurs Ausgehend von den Implementierungsskizzen für FCFS und RR sind an einigen Stellen Änderungen in vergleichsweise geringem Umfang erforderlich. Im Fokus steht die Verwaltung von nunmehr zwei ↑verkettete Listen, deren Einträge zur Einlastung bereite Aufgaben beziehungsweise Prozesse repräsentieren:

```
typedef struct backlog {
    queue_t ready;              /* normal list of ready-to-run processes */
    chain_t favor;            /* priority list of ready-to-run processes */
} backlog_t;
```

Diese Datenstruktur ist eine einfache Erweiterung des vorher bereits für FCFS oder RR benutzten ↑Datentyps, der dort jedoch nur die normale Bereitliste (`ready`) als Attribut hat. Hinzu kommt nun die Vorzugsliste (`favor`), wobei die Listeneinträge nach aufsteigender Restdauer der jeweiligen Zeitscheibe sortiert sind. Die Einträge in dieser Liste sind ↑Exemplare des folgenden Datentyps:

```
typedef struct entry {
    chain_t next;                           /* successor on some waitlist */
    rank_t rank;                                /* sort key of this entry */
} entry_t;
```

Neben dem Attribut zur Verkettung (`next`) enthält jeder Eintrag einen Sortierschlüssel (`rank`), der letztlich die Position des vorzuziehenden Prozesses in Abhängigkeit von den anderen Prozessen seiner Art in der Liste festlegt. Im gegebenen Fall ist der Sortierschlüssel gleich dem Wert der Restdauer der Zeitscheibe des Prozesses, dessen ↑Prozesskontrollblock (`process_t`) einen solchen Eintrag als Attribut (`line`) umfasst. Im Moment der Beendigung der Ein-/Ausgabe wird ein auf dieses ↑Ereignis wartender Prozess bereitgestellt, indem sein Prozesskontrollblock entsprechend seines Ranks auf die Vorzugsliste kommt:

```
void favor(process_t *task) {         /* schedule according to accuteness */
    assert(task != being(ONESELF));     /* don't put oneself ready to run */

    state(&task->mood, FLUSH|READY);          /* set process ready to run */

    ticket(&task->line, slack(task));                   /* define sort key */
    enlist(&labor()->favor, &task->line);      /* sort into priority list */
}
```

Prozesse, die der Vorzugsliste entnommen werden, belegen den Prozessor maximal für die Restdauer ihrer Zeitscheibe, das heißt, für den ihnen in der Zeitscheibe jeweils noch zur Verfügung stehenden Schlupf (`slack`):

```
time_t slack(process_t *task) {         /* unused period within timeslice */
    return slice() - task->time.usage;
}
```

Zur Differenzbildung ist die seit Wiederaufnahme des Prozesses verstrichene Zeit (`usage`) heranzuziehen. Der berechnete Wert bestimmt letztlich den Rank, den der Prozess auf der Vorzugsliste erhält. Diesem Rank entsprechend wird der Prozess einsortiert (`enlist`), wozu das Verkettungsglied im betreffenden Prozesskontrollblock zuvor noch zu definieren ist:

```
rank_t ticket(entry_t *item, rank_t rank) {     /* fetch and set sort key */
    rank_t aux = item->rank;                            /* pluck old rank */
    item->rank = rank;                                 /* define new rank */
    return aux;                                       /* deliver old rank */
}
```

Zur Berechnung der Restlaufzeit eines Prozesses innerhalb der Zeitscheibe ist noch seine Startzeit (`time.start`) festzuhalten. Dazu wird beim ↑Prozesswechsel (`seize`) ein ↑Zeitstempel gesetzt (`timestamp`), der den Zeitpunkt für den Beginn des Rechenstoßes des fortgesetzten Prozesses markiert (vgl. S. 251):

```
void seize(process_t *task) {                         /* dispatch process */
    process_t *last = being(ONESELF);     /* access process control block */

    state(&last->mood, CLEAR|RUNNING);      /* cancel running state, only */

    last->flow.crux = shift(task);        /* switch process, save context */
    task->time.start = timestamp();                  /* begin of CPU burst */
    lapse(task->time.slice);                              /* set the timer */

    state(&last->mood, FLUSH|RUNNING);      /* define running state, only */
}
```

Da die Zeitscheibenlänge davon abhängt, ob der wiederaufgenomme Prozess aus der Bereit- oder Vorzugsliste kam (s. `elect`, unten), ist der Zeitgeber (↑PIT) prozessabhängig zu programmieren (`lapse`).
Ein weiterer Zeitstempel wird in dem Moment gesetzt, wenn der Prozess auf die Beendigung seines Ein-/Ausgabestoßes warten muss und dazu den Prozessor abgeben wird. In Bezug auf den ↑Prozesszustand wechselt der Prozess von laufend nach blockiert. Diese zweite Markierung führt nicht die Operation (`seize`) zum Prozesswechsel durch, die nämlich auch bei Prozessverdrängung zur Wirkung kommt und der Prozess dabei nicht blockiert, sondern vom Zustand laufend in den Zustand bereit übergeht. Die Operation zur Blockierung eines Prozesses und Auswahl seines Nachfolgers ergibt sich demnach wie folgt (vgl. S. 93):

```
void block() {                        /* release and reschedule processor */
    process_t *next, *self = being(ONESELF);

    state(&self->mood, BLOCKED);                    /* mark myself blocked */

    next = quest(labor());          /* search for next process to execute */
    if (next == self)                         /* myself already unblocked? */
        state(&self->mood, FLUSH | RUNNING);              /* yes, continue */
    else {                                          /* no, release processor */
        self->time.usage = usage(self);
        seize(next);                              /* switch to next process */
    }
}
```

Die Differenz zwischen der bei Suspendierung (`block`) des Prozesses genommenen Endzeit (`timestamp`) und seiner Startzeit ergibt die Länge seines letzten Rechenstoßes, das heißt, den Verbrauch (`usage`) des Prozesses von seiner Zeitscheibe:

```
time_t usage(process_t *task) {           /* used period within timeslice */
    return timestamp() - task->time.start;
}
```

Der berechnete Wert bestimmt die relative Position des Prozesses auf der Vorzugsliste, nämlich wenn der Prozess deblockiert und wieder bereitgestellt wird (`favor`, s. oben).
Wann immer im Moment des Ablaufs einer Zeitscheibe wenigstens ein Prozess die Prozessorzuteilung erwartet (`check`, S. 272) wird der gegenwärtig stattfindende Prozess zurück auf die normale Bereitliste gesetzt. Gegenüber RR beziehungsweise FCFS unterscheidet sich die Operation dazu nur durch die Wahl der Liste bereitgestellter Prozesse (vgl. S. 92):

```
void ready(process_t *task) {                /* schedule according to FCFS */
    process_t *self = being(ONESELF);        /* use own process descriptor */

    assert((task != self) || valid(&self->mood, PENDING | IDLE));

    state(&task->mood, READY);                 /* set process ready to run */
    enqueue(&labor()->ready, &task->line);           /* add to ready list */
}
```

Dies ist übrigens ebenfalls die Vorgehensweise für jeden Prozess, der erstmalig (d.h., nach seiner Erzeugung) bereitgestellt wird. Im Gegensatz zur Vorzugsliste belegt jeder Prozess, der der Bereitliste entnommen wird, den Prozessor für maximal eine volle Zeitscheibe. Für die Bestimmung eines zur Einlastung des Prozessors bereitstehenden Prozesses wird sodann aus einer der beiden Listen ausgewählt. Darüber hinaus wird in Abhängigkeit von der gewählten Liste die Länge der nächsten Zeitscheibe bestimmt und für den zeitnah erfolgenden Prozesswechsel (`seize`, s. oben) prozessspezifisch (`time.slice`) festgehalten:

```
process_t *elect(backlog_t *list) {        /* choose ready-to-run process */
    process_t *next;

    if ((next = forge(unbag(&list->favor))))     /* priority list filled? */
        next->time.slice = rating(&next->line);     /* yes, residual term */
    else                                        /* no, use normal ready list */
        if ((next = forge(dequeue(&list->ready))))  /* ready list filled? */
            next->time.slice = slice();                 /* yes, full term */

    return next;
}
```

Demnach wird der Prozess mit dem kleinsten Zeitscheibenwert (`rating`) bevorzugt, wenn die Vorzugsliste nicht leer ist (`unbag`). Der entsprechende Eintrag steht gleich oben auf der Liste. Ist dagegen die Vorzugsliste leer, wird der nächste Prozess, dem dann eine ganze Zeitscheibe (`slice`) zuzuteilen ist, dem Kopf der Bereitliste entnommen (`dequeue`) — in beiden Fällen ist aus dem Kettengliedzeiger noch ein Prozesszeiger zu formen (`forge`: vgl. FCFS, S. 93). Die Programmierung des Zeitgebers mit dem prozessabhängigen Zeitscheibenwert geschieht bei der Einlastung (`seize`), im Gegensatz zu RR, wo die Zeitscheibenlänge für alle Prozesse identisch ist.

wafer (dt.) ↑Halbleiterscheibe.

Wagenrücklauf (en.) ↑*carriage return*. Vorgang der Rückführung des Läufers (↑*cursor*) an den Zeilenanfang einer ↑Dialogstation. Insbesondere auch zur Auslösung einer meist durch Betätigung der ↑Eingabetaste abgegebenen Anweisung zur Ausführung einer ↑Kommandozeile. Der Begriff hat seinen Ursprung in der Schreibmaschine beziehungsweise dem ↑Fernschreiber, wo am Zeilenende manuell (durch den Zeilenschalthebel der Schreibmaschine) oder automatisch (durch Tastendruck oder ein Steuerzeichen beim Fernschreiber, ↑CR) der Wagen (*carriage*) mit den Typenhebeln oder dem Kugelkopf auf den Anfang einer neuen Zeile positioniert werden musste.

wait behaviour (dt.) ↑Warteverhalten.

wait-freedom (dt.) ↑Wartefreiheit.

waiting period (dt.) ↑Wartezeit.

waitlist (dt.) ↑Warteliste.

Wartefreiheit (en.) ↑*wait-freedom*. ↑Fortschrittsgarantie für ↑nichtblockierende Synchronisation. Diese Garantie besagt, dass jeder ↑Prozess jede Operation in einer endlichen Anzahl von Schritten vollenden kann, ohne Rücksicht auf die relativen Geschwindigkeiten der anderen

Prozesse. Eine stärkere Zusicherung als ↑Sperrfreiheit, da jedem einzelnen Prozess in einer ↑Konkurrenzsituation Fortschritt garantiert wird.

Warteliste (en.) ↑*waitlist*. Liste von ↑Prozessen, die darauf warten, ein bestimmtes ↑Betriebsmittel zugeteilt zu bekommen. Die wartenden Prozesse sind für gewöhnlich im ↑Prozesszustand blockiert, das heißt, sie gelten als nicht lauffähig, da ihnen neben der ↑CPU noch wenigstens ein anderes Betriebsmittel zum Vorankommen fehlt (↑*process blockade*). Prozesse, die nur noch die Zuteilung der CPU als einzig fehlendes Betriebsmittel benötigen, warten im Zustand bereit auf der sogenannten ↑Bereitliste.
Von der konkreten technischen Lösung, die sich hinter der Implementierung einer solchen Liste verbergen kann, wird abstrahiert. Je nach Problemstellung können sich verschiedene Lösungsvarianten ergeben, die zwar funktional alle identisch sind, aber nichtfunktional (bspw. Speicher- oder Zeitbedarf) voneinander abweichen. So ist solch eine Liste nicht zwingend als *Reihe* von wartenden Prozessen (↑*queue*) zu verstehen.

Warteschlange (en.) ↑*queue*. Konstrukt für aufgelaufene, unerfüllte und zeitweilig zurückgestellte Aufträge. Bildet sich, wenn in einem Zeitintervall mehr Aufträge eintreffen, als in demselben Intervall verarbeitet werden können. Eine ↑Warteliste, die eine bestimmte *Reihe* solcher Aufträge zum Ausdruck bringt: sie „stehen in einer Schlange“.
Ist ein solcher Auftrag etwa ein ↑Prozess, so bedeutet beispielsweise eine gefüllte ↑Bereitliste, dass zu wenig ↑Betriebsmittel der Art ↑Prozessor zur Verfügung stehen. Hinzufügen weiterer Prozessoren kann zum Abbau der Liste beitragen, oder die gerade verfügbaren Prozessoren werden im ↑Zeitteilverfahren entsprechend einer bestimmten Strategie bedient, bis die Liste geleert werden konnte. Die Strategie ist im ↑Einplanungsalgorithmus verankert.

Warteverhalten (en.) ↑*wait behaviour*. Art und Weise, wie ein ↑Prozess dem Eintreffen von einem ↑Ereignis entgegensieht: ↑aktives Warten oder ↑passives Warten. Das Ereignis, auf das der Prozess wartet, definiert einen bestimmten Zustand, der gelten muss, damit der Prozess in seinem ↑Programm weiter voranschreiten kann. Typisches Beispiel ist der von einem Prozess ausgelöste ↑Ein-/Ausgabestoß. Hängt der Prozess von einzugebenden ↑Daten ab, muss er die Beendigung des diesbezüglichen Eingabestoßes abwarten: um Fortschritt erzielen zu können, benötigt der Prozess Eingabedaten, die ihm am Ende dieses Stoßes zur Verfügung stehen und dann durch ein Signal, ein ↑konsumierbares Betriebsmittel, angezeigt werden. Möchte der Prozess Daten ausgeben und ist jedoch der dazu benötigte Ausgabepuffer voll, muss der Prozess die Beendigung des diesbezüglichen Ausgabestoßes abwarten: um Fortschritt erzielen zu können, benötigt der Prozess ein ↑wiederverwendbares Betriebsmittel, den Puffer, der durch den Ausgabestoß geleert wird (↑DMA) und mit Ende dieses Stoßes wieder Platz für weitere auszugebende Daten hat. Aber auch ohne nebenläufigem Ein-/Ausgabestoß — die ↑Nebenläufigkeit bezieht sich auf den jeweiligen Stoß, kausal abhängig ist der Prozess auf das dem Stoßende entsprechenden Ereignis — kann ein Prozess in Wartesituationen kommen, nämlich wenn er allgemein ↑Synchronisation mit anderen Prozessen erzielen muss.

Wartezeit (en.) ↑*waiting period*. Bezeichnung für eine ↑Prozessgröße, die die Zeitspanne von der Unterbrechung bis zur Fortsetzung einer Operation oder Operationsfolge quantifiziert.

waste (dt.) ↑Verschnitt.

WCET Abkürzung für (en.) *worst-case* ↑*execution time*, größte anzunehmende ↑Ausführungszeit.

weak consistency (dt.) ↑schwache Konsistenz.

Wechseldatenträger (en.) ↑*removable medium*. ↑Speichermedium, das nicht fest in einem ↑Rechensystem eingebaut ist. Das Medium ist austauschbar, es wird durch ein am Rechensystem angeschlossenes ↑Peripheriegerät zum Speichern und Abrufen (gespeicherter) Informationen zugänglich gemacht. Das ↑Betriebssystem ermöglicht einem ↑Prozess den Zugang zu diesem Gerät per ↑Systemaufruf. Die im Betriebssystem stattfindende Interaktion mit dem Gerät bewerkstelligt ein ↑Gerätetreiber, der speziell auf die Eigenschaften dieses Geräts

zugeschnitten ist. Beispiele für solche heute (2025) eingesetzten Medien sind der Lochstreifen, die Lochkarte, das Magnetband, die Magnetplatte, flexible Magnetplatte (*floppy disc*), optische Speicherplatte (↑CD, ↑DVD), Speicherkarte (*flash/memory card*, ↑SSD) oder der Speicherstab (*memory stick*, ↑USB).

wechselseitiger Ausschluss (en.) ↑*mutual exclusion*. Form von ↑multilaterale Synchronisation. Sorgt dafür, dass ein ↑kritischer Abschnitt stets nur von höchstens einen ↑Prozess betreten und durchstreift werden kann. Inbegriff für ↑blockierende Synchronisation.
Jeder der beteiligten Prozesse erwartet *aktiv* (↑Lese-/Schreibsperre, ↑Umlaufsperre, ↑Ticketsperre, ↑gereihte Umlaufsperre) oder *passiv* (↑binärer Semaphor) das ↑Ereignis, dass der von einem Prozess erworbene kritische Abschnitt wieder freigesetzt wird. Ein Prozess wartet aktiv, wenn er durch anhaltendes Abfragen (*polling*) seiner Wartebedingung während seiner Wartezeit beschäftigt bleibt (↑*busy waiting*). Im Gegensatz dazu wartet ein Prozess passiv, wenn er blockiert (↑Prozesszustand) und dem ↑Ereignis, dass der kritische Abschnitt verlassen wird, „schlafend“ entgegensieht.
Als ↑Entität ist mit dieser Form der multilateralen Synchronisation der ↑Mutex eng verbunden. Anders als die ↑Informatikfolklore jedoch verlauten lässt, ist damit allerdings nicht zwingend ein „spezialisierter binärer Semaphor“ gemeint. Vielmehr lässt sich jede Variante des wechselseitigen Auschlusses als Mutex auslegen.

Wettlaufsituation (en.) ↑*race condition*, ↑*race hazard*. Umstand, bei dem ein ↑nichtsequentielles Programm eine ↑Aktion oder ↑Aktionsfolge mit unbestimmten zeitlichen Verhalten zulässt. Konsequenz daraus ist möglicheweise ein inkorrekter ↑Programmablauf, der ein Fehlverhalten oder eine falsche Berechnung bewirkt. Diesem Umstand zugrunde liegt immer wenigstens ein asynchrones ↑Ereignis, das den Programmablauf nicht nur unvorhersagbar, sondern auch nicht reproduzierbar macht. In Bezug auf den betrachteten Programmablauf hat ein solches Ereignis immer die Ursache in wenigstens einer gleichzeitig stattfindenden Aktion, nämlich hervorgerufenen durch einen anderen, zeitlich überlappend, das heißt, parallel stattfindenden Programmablauf (↑paralleler Prozess, ↑Unterbrechungsbehandlung). Zur Vorbeugung möglicher Fehlfunktionen sind Maßnahmen zur ↑Nebenläufigkeitssteuerung erforderlich.
Solch eine Situation beruht immer auf einen Konflikt beim Zugriff auf wenigstens eine, den parallelen Programmabläufen gemeinsame ↑Speicherstelle. Gemeinhin liegt an dieser Speicherstelle eine ↑Variable (↑*shared variable*), die von den Programmabläufen wechselseitig entweder gelesen und geschrieben (*read-write conflict* bzw. *write-read conflict*) oder geschrieben (*write-write conflict*) wird. Im Einzelnen äußern sich diese Konflikte, wenn:

read-write dieselbe Leseoperation, nacheinander mit derselben gemeinsamen Variablen als Operand durchgeführt von demselben Programmablauf, verschiedene Werte liefert (*non-repeatable read*) — zwischen den beiden Leseoperationen desselben Programmablaufs geschieht wenigstens eine Schreiboperation eines anderen Programmablaufs, wobei all dieses Operationen dieselbe gemeinsame Variable als Operand haben;

write-read dieselbe noch nicht zurückgeschriebene gemeinsame Variable gleichzeitig von einem Programmablauf gelesen und von einem anderen Programmablauf geschrieben wird (*dirty read*) — nach der Lese-, aber noch vor der zugehörigen Schreiboperation desselben Programmablaufs geschieht wenigstens eine weitere zusammenhängende Sequenz einzelner Lese-/Schreiboperationen eines anderen Programmablaufs, wobei all diese Operationen dieselbe gemeinsame Variable als Operand haben;

write-write dieselbe noch nicht zurückgeschriebene gemeinsame Variable gleichzeitig von verschiedenen Programmabläufen geschrieben wird, ohne die zuvor ausgeübte Schreiboperation des jeweils anderen Programmablaufs mitzubekommen (*lost update*) — die mit der ersten Schreiboperation des einen Programmablaufs bezweckte Aktualisierung des Variablenwerts geht mit der zweiten Schreiboperation des anderen Programmablaufs zur selben gemeinsamen Variablen verloren.

Bis auf den letzten Fall (*write-write*) sind die beiden anderen Fälle *serialisierbar*, das heißt, der mögliche Konflikt lässt sich durch geeignete Maßnahmen zur Nebenläufigkeitssteuerung verhindern. Eine Maßnahme wäre, die verschiedenen Programmabläufe für die Dauer der jeweils zusammenhängenden Lese-Leseoperationen (*read-write*) oder Lese-Schreiboperationen (*write-read*) wechselseitig auszuschließen (↑*mutual exclusion*). Eine andere Maßnahme besteht darin, die kritische Aktionsfolge als ↑Transaktion zu formulieren und damit Programmabläufe nicht zwingend von der Durchführung der betreffenden Lese-/Schreiboperationen zeitweilig auszuschließen (↑nichtblockierende Synchronisation).
Eine Serialisierung des Schreib-Schreibkonflikts ist unmöglich, da alle Schreiboperationen zur selben gemeinsamen Variablen dasselbe Gewicht haben und immer der zufällig, unvorhersehbar jeweils zuletzt schreibende Programmablauf den Variablenwert festlegen wird. Aber spätere Schreiboperationen könnten unterbunden werden, wenn die erste Schreiboperation einen diesbezüglichen Hinweis hinterlässt (↑TAS, ↑CAS). Es liegt dann ganz in der Hand des diese Schreiboperationen umfassenden nichtsequentiellen Programms, den Umstand für die nächste nicht mehr zu unterbindende und damit wieder wirksame Schreiboperation auf die betreffende gemeinsame Variable zu bestimmen.

Exkurs Nachfolgend soll anhand von zwei Fällen zum einen der Lese-Schreibkonflikt und zum anderen der Schreib-Lesekonflikt verdeutlich werden. Der erste Fall ist der ↑Prozesseinplanung entnommen, genauer ↑HRRN, und betrifft die Berechnung des Verhältniswerts aus verstrichener ↑Wartezeit und erwarteter ↑Laufzeit (↑Rechenstoß) von einem ↑Prozess. Dieser Verhältniswert drückt die relative ↑Dringlichkeit eines Prozesses aus, die ↑CPU zugeteilt zu bekommen und bestimmt damit die Position des Prozesses auf der ↑Bereitliste. Die entsprechende Berechnung führt folgende Funktion aus (vgl. S. 121):

```
rank_t judge(process_t *task, time_t time) {   /* value according to HRRN */
    time_t wait = time - task->time.ready;         /* settle waiting time */
    return (wait + task->time.guess) / task->time.guess; /* compute ratio */
}
```

Beim Lese-Schreibkonflikt führt ein Prozess nacheinander zwei Leseoperation auf einer gemeinsamen Variablen aus, wobei die Lesesequenz überlappt wird von (wenigstens) einer Schreiboperation auf dieselbe Variable, allerdings hervorgerufen durch einen anderen Prozess. Die zweite Leseoperation läuft dann Gefahr, einen Wert zu liefern, der ungleich zum Wert der ersten Leseoperation ist. Eine solche Lesesequenz zeigt die `return`-Anweisung, die zwei aufeinanderfolgende Leseoperationen des `guess`-Attributs ein und desselben ↑Prozesskontrollblocks (`task`) spezifiziert. Wenn nun die Attribute des Prozesskontrollblocks als flüchtig (`volatile`) gelten, was beispielsweise für HRRN geboten sein kann, dann wird jeder in der Hochsprache (↑C) formulierte Lesezugriff auf `guess` auch immer direkt die Speicherstelle, an der `guess` liegt, adressieren und nicht etwa auf einen beim ersten Lesezugriff möglicherweise in einem ↑Prozessorregister zwischengespeicherten Wert zurückgreifen. Eine zwischenzeitliche Veränderung des Wertes von `guess`, beispielsweise um dem zugehörigen Prozess mehr Gewicht zu geben und ihn auf der Bereitliste explizit nach vorne zu drücken, kann ein zur angegebenen Berechnungsvorschrift (`judge`) für den korrekten Verhältniswert inkonsistentes Ergebnis liefern.
Der eben betrachtete Fall kann dazu führen, dass eine bestimmte Maßnahme zur Prozesseinplanung nicht immer korrekt durchsetzbar ist. Dieser ↑Defekt muss aber keine für das System kritische Auswirkungen haben, er kann im weiteren Verlauf wirkungslos bleiben und durch eine wiederholte Berechnung des Verhältniswerts korrigiert werden — was jedoch nicht bedeutet, den möglichen Lese-Schreibkonflikt einfach zu ignorieren.
Anders verhält es sich im zweiten Fall, der Schreib-Lesekonflikt, der beispielsweise im Zusammenhang mit Lösungen zum ↑Erzeuger-Verbraucher-Problem auftreten kann. Hier konkret gemeint sind die Aktionen, um ein Datum in einen ↑Puffer zu schreiben beziehungsweise daraus zu lesen und dabei sicherzustellen, dass derselbe *Pufferplatz* nicht zugleich von mehreren Prozessen adressiert werden kann (vgl. S. 87). Der Puffer ist ein *Feld* und ein

Pufferplatz darin wird eindeutig durch einen *Feldindex*, einem Zähler, identifiziert. Um den Puffer schrittweise zu füllen beziehungsweise zu leeren, ist lediglich der betreffende Zähler um Eins zu erhöhen. Hierzu wird aus gutem Grunde die ↑Elementaroperation `FAI` (vgl. S. 81) verwendet. Angenommen, FAI wäre keine solche Operation, sondern lediglich definiert als:

```
inline int FAI(int *this) {                          /* fetch and increment */
    return (*this)++;          /* increase counter by 1, deliver old value */
}
```

Angewendet in den Operationen zum Befüllen (`store`) beziehungsweise Entleeren (`fetch`) des Puffers (vgl. S. 87), generiert der ↑Kompilierer folgende ↑Maschinenbefehle für FAI (↑x86):

```
I1: movl (%ecx), %eax      # read counter from RAM, register result
I2: leal 1(%eax), %edx     # modify its value by +1, register temporary
I3: movl %edx, (%ecx)      # write temporary back to counter into RAM
```

Diese Sequenz ist *semantisch äquivalent* zur Einzelaktion `(*this)++` auf der Hochsprachenebene, sie implementiert diese einzelne Anweisung als logisch zusammenhängende Aktionsfolge, die jedoch nicht in einem unteilbaren Einzelschritt, sondern teilbar in drei Teilschritten (↑*read-modify-write*) durchgeführt führt. In dem genannten Anwendungsbeispiel kann damit nach jedem Teilschritt eines Erzeugers oder Verbrauchers, Prozess P_1, ein Teilschritt eines anderen Erzeugers beziehungsweise Verbrauchers, P_2, zur Ausführung gelangen. Eine mögliche Abfolge der Teilschritte sowie die schrittweise Veränderung des in einer gemeinsamen Variablen enthaltenen und durch einen ↑Zeiger (`this` bzw. `%ecx`) indirekt adressierten Zählerwerts zeigt nachfolgende Aufstellung:

Teil-	P_1			P_2			Zähler
schritt	Befehl	`%eax`	`%edx`	Befehl	`%eax`	`%edx`	`(%ecx)`
1	`I1`	42	0815				42
2	`I2`	42	43				42
3				`I1`	42	4711	42
4				`I2`	42	43	42
5				`I3`	42	43	43
6	`I3`	42	43				43
Ergebnis		42			42		43

Dabei ist zu beachten, dass die in den Prozessorregistern (hier: `%eax`, `%ecx`, `%edx`) gespeicherten Werte den ↑Prozessorstatus des jeweiligen Prozesses (P_1, P_2) definieren. Die betreffenden Register sind damit Platzhalter für *prozesseigene Variablen*, die niemals mehreren Prozessen zugleich zugänglich sind. Beim ↑Prozesswechsel wird der Prozessorstatus des die CPU abgebenden Prozesses (direkt/indirekt in dessen Prozesskontrollblock) gesichert und der des die CPU erhaltenen Prozesses (direkt/indirekt aus dessen Prozesskontrollblock) hergestellt. Dies betrifft jedoch nicht die, durch den im Prozessorregister (`%ecx`) gehaltenen Zeiger (`this`) identifizierte, beiden Prozessen (P_1, P_2) gemeinsame Zählervariable.

Diese mögliche Abfolge von Teilschritten zeigt, dass sowohl P_1 als auch P_2 denselben Zählerwert (42) als Feldindex verwenden, damit fälschlicherweise auf denselben Pufferplatz lesend oder schreibend zugreifen, mit dem sich daraus ergebenden ↑Fehler (als Folge ausbleibender ↑Unteilbarkeit der Zähloperation), entweder denselben gepufferten Wert zu erhalten oder einen bereits gepufferten Wert durch Überschreiben zu löschen. Darüberhinaus berechnen beide Prozesse fälschlicherweise denselben nächsten Zählerwert (nämlich 43, anstatt 44): obwohl logisch zwei Prozesse jeweils eine Zähloperation durchgeführt haben, infolgedessen der Zähler korrekterweise um Zwei hätte erhöht werden müssen, geschah fälschlicherweise die Erhöhung des Zählers aber nur um Eins.

Fehlverhalten und Berechnungsfehler sind der Tatsache geschuldet, dass, bevor P_1 den von ihn um Eins erhöhten Zählerwert in den als globale, gemeinsame Variable ausgelegten Zähler zurückschreiben kann, P_2 noch den alten Zählerwert liest (Schreib-Lesekonflikt: *dirty read*). Derselbe Fehler würde daher auch auftreten, sollte P_2 seine Aktionsfolge sofort nach

Teilschritt 1 starten. Zur Lösung dieses Problems ist somit zu verhindern, dass P_2 im Konfliktfall auf den Zähler zugreifen kann oder den Zähler inkonsistent verwendet, und zwar in der Zeitspanne (↑kritischer Abschnitt) ab Beginn der Lese- und bis Ende der zugehörigen Schreiboperation durch P_1. Für den gegebenen Anwendungsfall ist dazu die gesamte Aktionsfolge (d.h., Befehle `I1`–`I3`) als Elementaroperation auszulegen (vgl. S. 82).

Wettstreit (en.) ↑*contention*. Bemühen ↑gekoppelter Prozesse, einander in etwas zu übertreffen, einander den Vorrang streitig zu machen (in Anlehnung an den Duden). Die betreffenden ↑Prozesse befinden sich in einer ↑Konkurrenzsituation.
Unter den beteiligten Prozessen herrscht mehr oder weniger starke ↑Konkurrenz, je nachdem, wieviele Prozesse zugleich dasselbe ↑Betriebsmittel beanspruchen. Die dazu anfallenden gleichzeitigen Zugriffe der Prozesse betreffen ein ↑gemeinsames Betriebsmittel, das zu einem Zeitpunkt nur exklusiv von einem Prozess belegt werden darf (↑*indivisable resource*) und dessen Einheiten nur in begrenzter Anzahl vorrätig sind (↑*reusable resource*). Die Prozesse sind indirekt über dieselbe, aber strikt nacheinander zu benutzende Einheit einer solcher Art von Betriebsmittel gekoppelt. Die daraus folgende ↑Sequentialisierung bestimmter ↑Aktionen dient auch der Vorbeugung einer ↑Wettlaufsituation der konkurrierenden Prozesse.

Wiedereintritt (en.) ↑*re-entrance*. Moment, in dem ein ↑Programm erneut, und zwar verschachtelt, zur Ausführung kommt. Ursache ist eine vorangegangene ↑Ausnahme, erhoben während der Programmausführung, wobei die ↑Ausnahmebehandlung Bestandteil des Programms selbst ist. Im Zuge dieser Behandlung wird ein in seiner Ausführung zuvor unterbrochener Programmabschnitt (einer indirekten ↑Rekursion nicht unähnlich) wiederholt betreten. Zur korrekten Ausführung ist für das einen solchen Abschnitt enthaltene Programm ↑Ablaufinvarianz gefordert.

Wiedereintrittssperre (en.) ↑*re-entry lock*. Vorkehrung zur Abwehr des möglichen ↑Wiedereintritts eines ↑Handlungsstrangs in einen für wenigstens eine ↑Wettlaufsituation anfälligen ↑Programmabschnitt (↑*critical section*); definiert eine ↑Sperrzone. Ausgelöst wird der Handlungsstrang durch eine asynchron gestartete ↑Aktion oder ↑Aktionsfolge. Typisches Beispiel ist die zeitweilige Aussetzung der weiteren Abarbeitung ↑zurückgestellter Prozeduraufrufe im Rahmen einer ↑Unterbrechungsbehandlung. Ein bereits laufender Handlungsstrang wird nach Setzen einer solchen Sperre jedoch bis zum Abschluss durchgeführt (↑*run to completion*), die Sperre wirkt erst beim Übergang zum nächsten anstehenden Behandlungsauftrag.

Exkurs Ein Ansatz zur Implementierung einer solchen Sperre ist es, die Anforderungen zum Wiedereintritt in einen bestimmten Programmabschnitt als zurückgestellten Prozeduraufruf aufzufassen. Dabei sorgt ein Wächter (*guard*) einerseits für die Zurückstellung bestimmter Prozeduraufrufe, sobald ein Handlungsstrang den betreffenden Programmabschnitt betreten hat und andererseits für die Wiederaufnahme dieser Aufrufe, wenn dieser Abschnitt verlassen wurde. Der ↑Datentyp dafür modelliert einen Merker (*flag*) für den Aktivitätszustand des Abschnitts und eine ↑Schlange zurückgestellter Prozeduraufrufe:

```
typedef struct {
    bool flag;              /* true, guard is active; otherwise, inactive */
    queue_t wait;           /* waiting list of accumulated entry attempts */
} guard_t;
```

Für die zu ergreifenden Maßnahmen zur Abschnittsüberwachung ist die mögliche Herkunft der Anforderung zum Wiedereintritt zu beachten. Der Wiedereintritt kann einerseits die Folge einer ↑Unterbrechungsbehandlung sein und andererseits durch einen Handlungsstrang eines anderen ↑Prozessors oder ↑Rechenkerns zustande kommen. Der Verlauf zum „Durchschleusen“ der Prozeduraufrufe ist in beiden Fällen recht ähnlich, wenngleich schon verschiedene (spezialisierte) Verfahren Verwendung finden können. Nachfolgend steht die Unterbrechungsbehandlung als Ursache im Vordergrund, weshalb Angaben zur Betriebsweise (*mode*) der Operationen der Einfachheit halber ungenutzt bleiben. In diesem Sinne wird die Abschnittsüberwachung wie nachfolgend skizziert ausgelöst:

```
inline void guard(char __attribute__ ((unused))mode) {
    cover(true);                        /* acquire non-reentrant section */
}
```

Diese Operation aktiviert die Abschnittsüberwachung, sie markiert den Anfang (*true*) des direkt angrenzenden und zu überwachenden Abschnitts. Am Abschnittsende wird die Marke zurückgenommen (*false*). Beide Maßnahmen klammern den betreffenden Programmabschnitt, sie ummanteln (*cover*) ihn und versehen ihn so mit einem Schutz vor Aktionen, die einen Wiedereintritt in den Abschnitt zur Folge haben können:

```
inline void cover(bool flag) {
    court()->flag = flag;                           /* set area protection */
}
```

Solche von den Operationen zum Setzen und Löschen eines Merkers umschlossenen Bereiche bilden einen Gesamtkomplex im Programm, einen Hof (*court*), für den ein bestimmter Wächter zuständig ist. Im gegebenen Fall wird davon ausgegangen, dass dieser Hof durch den Gesamtkomplex eines ↑Betriebssystemkerns definiert ist:

```
inline guard_t *court() {
    extern guard_t bkg;                                /* big kernel guard */
    return &bkg;
}
```

Damit ist für den Betriebssystemkern ein einziger Wächter zuständig. Die darüber zum Ausdruck gebrachte Sperre ist jedoch nicht mit dem sogenannten ↑BKL zu verwechseln, denn sie führt nicht dazu, dass ein ↑gleichzeitiger Prozess blockieren könnte, sondern bewirkt lediglich die Zurückstellung bestimmter asynchron ausgelöster Aktionen eines solchen Prozesses. Der betreffende Prozess selbst wird nicht geblockt, er kann weiter voranschreiten. Für eine Unterbrechungsbehandlung ist ein solches Verhalten wesentlich, vor allem um ↑Verklemmungen von Prozessen vorzubeugen.
Befindet sich nun ein Prozess innerhalb eines derart überwachten Bereichs, werden Prozeduraufrufe, die zeit- und räumliche Überlappungen von kritischen Aktionsfolgen mit diesem Prozess bewirken könnten, zurückgestellt, in eine Schlange eingereiht. Diese Schlange wird geräumt (*clear*) sobald der Prozess den überwachten Abschnitt verlässt:

```
inline void clear(char __attribute__ ((unused))mode) {
    cover(false);                       /* release non-reentrant section */
    if (backlog(&court()->wait))                  /* any pending request? */
        treat();                          /* yes, process procedure list */
}
```

Die Betriebsweise (*mode*) auch dieser Operation (vgl. `guard` oben) kann je nach Herkunft einer Wiedereintrittsanforderung verschieden ausgelegt sein. Jedoch sollen Angaben dazu hier ebenfalls ungenutzt bleiben, da einfachheitshalber nur eine Variante (Unterbrechungsbehandlung) den Gesamtzusammenhang bildet. Allen Varianten gemeinsam ist allerdings, unerfüllte Wiedereintrittsanforderungen zu behandeln (*treat*) und die zugehörigen zurückgestellten Prozeduraufrufe nachträglich auszuführen:

```
void treat() {
    sequel_t *item;
    while ((item = next(&court()->wait)))            /* next noted sequel */
        enact(&item->call);                  /* put procedure into effect */
}
```

Mit der Ausführung eines jeden zurückgestellten Prozeduraufrufs kommt es zur Fortsetzung einer bestimmten Sache (*sequel*, vgl. S. 404), die für gewöhnlich Auswirkungen auf den Systemzustand der zum selben Hof gehörenden überwachten Abschnitte hat. Jeder darüber zur Ausführung kommenden Prozedur wird dieselbe Schutzmaßnahme verordnet (*enact*), die auch für den soeben noch verlassenen überwachten Abschnitt galt:

```
inline void enact(thunk_t *call) {
    cover(true);                          /* acquire non-reentrant section */
    bleep(call);                               /* run interposed procedure */
    cover(false);                         /* release non-reentrant section */
}
```

Ein zurückgestellter Prozeduraufruf wird „angepiebst“ (*bleep*, vgl. S. 403), ummantelt (*cover*) und kommt schließlich synchronisiert mit dem stattfindenden Prozess zur Ausführung. Die Zurückstellung der Prozeduraufrufe geschieht im Kontext einer Unterbrechungsbehandlung, genauer als Folge eines ↑AST. Letztlich wird der Aufruf weitergeleitet (*relay*) an den (für den Betriebssystemkern, d.h., Hof) zuständigen Wächter. Ist dieser eingeschaltet (*going*), wird der Prozeduraufruf zurückgestellt, andererseits durchgeführt:

```
inline void relay(sequel_t *item) {
    guard_t *gate = court();
    if (going() || backlog(&gate->wait))                /* section active? */
        enqueue(&gate->wait, &item->next);    /* yes, defer procedure call */
    else                                        /* no, go ahead and invoke */
        enact(&item->call);                   /* put procedure into effect */
}
```

Da das Leeren (*clear*, s.o.) der Schlange zurückgestellter Prozeduraufrufe außerhalb des jeweils geschützten Abschnitts geschieht, könnten zwischenzeitig ausgelöste Prozeduraufrufe zurückgestellte überholen. Um dies zu verhindern, wird ein Prozeduraufruf auch im Falle eines Rückstands (*backlog*) anhängender Aufrufe zurückgestellt, auch wenn in dem Moment der Wächter nicht eingeschaltet, das heißt, kein überwachter Abschnitt aktiv ist. Zur Überprüfung des Betriebszustands des Wächters dient folgende einfache Operation:

```
inline bool going() {
    return court()->flag;                       /* deliver operating state */
}
```

Die mögliche Zurückstellung von Prozeduraufrufen, obwohl der Wächter nicht angeschaltet beziehungsweise kein überwachter Abschnitt aktiv ist, bedeutet nicht, dass die betreffenden Prozeduren nicht zur Ausführung gelangen. Vielmehr deutet diese Situation auf den laufenden Vorgang zum Leeren (*clear*, s.o.) der Aufrufschlange hin: die Unterbrechungsbehandlung, die einen Prozeduraufruf weitergibt (*relay*), läuft im Namen des Prozesses, der einen überwachten Abschnitt verlässt und dazu die Aufrufschlange abarbeitet. Dieser Prozess wurde nämlich in genau der Phase unterbrochen, in der er eben diese Schlange bereits verarbeitetet (*treat*). Wenn die Schlangenoperationen zum Einreihen und Entfernen zurückgestellter Prozeduraufrufe (*sequel*) korrekt synchronisiert sind, geht keiner dieser Aufrufe verloren.

Dieses Verfahren geht von der Aufteilung einer Unterbrechungsbehandlung in zwei Phasen aus, die sich am besten als *Prolog* und *Epilog* begreifen lassen. In der Prologphase läuft die Unterbrechungsbehandlung der ersten (↑FLIH) und zweiten (↑SLIH) Stufe, wobei letztere als AST abgefertigt wird. Diese Phase verläuft asynchron sowohl zum System als auch zum unterbrochenen Prozess. Ist das System nicht durch eine ↑Benutzthierarchie bestimmt, in der ausnahmslos entweder ↑nichtblockierende Synchronisation oder ↑Unterbrechungssperren Verwendung finden, dann darf aus der Prologphase heraus für gewöhnlich keine einzige ↑Systemfunktion direkt aufgerufen werden. Letzteres ist dann nur der Epilogphase vorbehalten, die zwar immer noch asynchron zum System, aber synchron zum jeweils stattfindenden Prozess verläuft. Dieser eine Prozess sorgt für die serialisierte Ausführung nicht wiedereintrittsfähiger Programmabschnitte, indem er die komplette Aufrufschlange Eintrag für Eintrag nacheinander verarbeitet, und hinterlässt so einen konsistenten Systemzustand.

Wird der jeweilige Prozess, der die Aufrufschlange abarbeitet, in eben dieser Phase immer wieder unterbrochen und setzt die Unterbrechungsbehandlung daraufhin weitere zurückzustellende Prozeduraufrufe ab, besteht durch die dann mögliche zwischenzeitliche Aufnahme weiterer Einträge in die Aufrufschlange offensichtlich der Anschein, dass der Prozess unbe-

grenzt verzögert werden kann und damit dem ↑Verhungern ausgesetzt ist. Dies aber ist kein grundsätzliches Problem des hier beschriebenen Sperrverfahrens. Vielmehr deutet eine solche Situation auf hochfrequente ↑Unterbrechungen hin, die so oder so einzeln zu behandeln sind. Auch mit einer anderen Art von Sperrverfahren beziehungsweise ↑Synchronisation würde jedem Prozess ein solches Schicksal ereilen können. Wenn die ↑Peripherie keine unechten Unterbrechungen (↑*spurious interrupt*) hervorruft, dann ist solch eine Situation nur durch einen schnelleren Prozessor beizukommen, der den Prozessen zwischen den Unterbrechungen mehr Spielraum zur normalen Programmausführung zugesteht.

Wiederholungslauf (en.) ↑*rerun*. Erneute ↑Interpretation von einem ↑Maschinenbefehl, bei dessen Verarbeitung zuvor im ↑Abruf- und Ausführungszyklus der ↑CPU eine ↑Ausnahmesituation eingetreten ist. Das ↑Betriebssystem hat die mit dem Maschinenbefehl verbundene ↑Aktion abgefangen (↑*trap*), die erhobene ↑Ausnahme behandelt (↑partielle Interpretation) und hinterlässt für die CPU einen ↑Prozessorstatus, der nach der Beendigung der ↑Ausnahmebehandlung zur Wiederholung der Ausführung des abgefangenen Maschinenbefehls führt (↑*rerun bit*).

wiederverwendbares Betriebsmittel (en.) ↑*reusable resource*. Bezeichnung für ein ↑Betriebsmittel, das nur in begrenzter Anzahl und, anders als ein ↑konsumierbares Betriebsmittel, dauerhaft (persistent) vorhanden ist, von dem es eine feste Anzahl von Einheiten gibt. Ein Betriebsmittel in gegenständlicher Form der Hardware (insb. ↑CPU, ↑RAM, ↑Ein-/Ausgabegerät) oder Software (↑Variable, Platzhalter).
Jede dieser Einheiten ist entweder verfügbar oder belegt, das heißt, keinem oder einem ↑Prozess zugewiesen. Mit erfolgter Zuweisung hat ein Prozess die von ihm angeforderte Betriebsmitteleinheit erworben (*acquire*). Ist die Mitbenutzung desselben Betriebsmittels ausgeschlossen (↑unteilbares Betriebsmittel), können Einheiten davon zu einem Zeitpunkt nur maximal einem Prozess zugewiesen sein. Ein Prozess kann beliebige von ihm belegte Einheiten von Betriebsmitteln freisetzen (*release*), sofern er seinerseits nicht den Erwerb einer Betriebsmitteleinheit erwartet und demzufolge blockiert ist. Erworbene Einheiten können dem Prozess nicht entzogen werden, sie sind erst wieder verfügbar, nachdem sie durch den Prozess explizit freigesetzt wurden.

Windhundprinzip (en.) ↑*first-come, first-served*. Bezeichnung für ein Planungsverfahren, das ein ↑wiederverwendbares Betriebsmittel ausschließlich nur in der zeitlichen Reihenfolge des Eintreffens von Bedarfsanmeldungen zuteilt („wer zuerst kommt, mahlt zuerst“).

Windows Mehrplatz-/Mehrbenutzerbetriebssystem, erste Installation 1985 (Intel 8086), zunächst lediglich als fensterbasierte graphische Benutzeroberfläche zur Erleichterung des Umgangs mit ↑MS-DOS. Erst seit ↑Windows NT mehrplatz-/mehrbenutzerfähig.

Windows NT Mehrplatz-/Mehrbenutzerbetriebssystem, erste Installation 1993 (Intel 80286). Steht für ↑Windows „New Technology“ — oder Halbgeschwister von ↑VMS, dem unter derselben Leitung (David Cutler) entstandenem ↑Betriebssystem: WNT entwickelt sich sich aus VMS, indem positionsweise ein Schritt im Alphabet weitergegangen wird.

Wirtsbetriebssystem (en.) ↑*host operating system*. Ein ↑Betriebssystem, das ein ↑Gastbetriebssystem bewirtet. Letzteres kann in der Bereitstellung einer eigenen ↑Systemfunktion von den (per ↑Systemaufruf zugänglichen) Systemfunktionen des Wirtssystems profitieren. Darüberhinaus kann mehr als ein Gastsystem von dem Wirtssystem bedient werden, ohne dass ein ↑Prozess des Gastsystems dies in funktionaler Hinsicht wahrnimmt. Eine solche Mehrfachnutzung kann sich *homogen* (gleichartige Gastsysteme) oder *heterogen* (verschiedenartige Gastsysteme) darstellen, ohne dass dies wiederum dem Wirtssystem bewusst ist.

Wirtsmaschine Maschine realer oder virtueller Natur, die eine ↑virtuelle Maschine bewirtet: letztere benutzt (↑Benutzthierarchie) erstere. Dies bedeutet in erster Linie, dass das ↑Programmiermodell der benutzten Maschine ebenfalls für die benutzende Maschine gilt. Typi-

scherweise läuft der ↑VMM als einziges ↑Programm auf dieser Maschine, um eben eine oder mehrere virtuelle Maschinen zu realisieren.

word length (dt.) ↑Wortbreite.

work sharing (dt.) ↑Arbeitsteilung.

work stealing (dt.) ↑Arbeitsentzug.

workaround (dt.) Abhilfe, ↑Behelf, Hilfskonstruktion, ↑Notbehelf, Notlösung.

working page (dt.) ↑Betriebsseite.

working set (dt.) ↑Arbeitsmenge.

workload (dt.) ↑Arbeitslast.

worst fit (dt.) schlechteste Passung, ↑Platzierungsalgorithmus.

Wort Kurzbezeichnung für ein ↑Maschinenwort oder ↑Speicherwort.

Wortbreite (en.) ↑*word length*. Größe (Bitanzahl) von dem ↑Maschinenwort einer ↑CPU. Typisch waren oder sind 8, 9, 12, 16, 18, 24, 32, 36, 39, 40, 48, 60 und 64 Bits pro Wort (Stand 2025).

wrapper procedure (dt.) ↑Mantelprozedur.

write (dt.) ↑schreiben.

write buffer (dt.) ↑Schreibpuffer.

write-allocate (dt.) ↑Schreibzuteilungstechnik.

write-back (dt.) ↑Rückschreibetechnik.

write-through (dt.) ↑Durchschreibetechnik.

writer (dt.) ↑Schreiber.

wrong signalling (dt.) ↑falsche Signalisierung.

Wurzelprozedur Oberprogramm, auch übergeordnetes ↑Programm, das ein ↑Unterprogramm aufruft, aber selbst nicht als Unterprogramm aufgerufen wird.

Wurzelsegment Bezeichnung für ein ↑Segment, das die Ansatzstelle (Wurzel) für die hardwaregestützte ↑Segmentierung von einem ↑Adressraum bildet (↑segmentierter Adressraum). In diesem Segment liegt die globale, jedem ↑Prozessadressraum gemeinsame ↑Segmenttabelle (↑Multics).

Wurzelverzeichnis (en.) ↑*root directory*. Bezeichnung für ein ↑Verzeichnis, das die Ansatzstelle (Wurzel) von einem ↑Dateisystem bildet. Von dieser Stelle aus ist jede in dem Dateisystem erfasste ↑Entität erreichbar, was insbesondere auch für die ↑Befestigung eines Dateisystems von Bedeutung ist. In dem Fall nämlich wird der ↑Befestigungspunkt (ein Verzeichnis) in einem Dateisystem zur Wurzel des eingehängten Dateisystems.

x86 Prozessorarchitektur, 16/32-Bit, abwärtskompatibel zum Intel 8086 (1978).

Xerox Alto Arbeitsplatzrechner (↑PC), 1973, entwickelt von Xerox PARC (*Palo Alto Research Center*) auf Basis einer eigens gefertigten ↑CPU in 16-Bit *bit-slice*-Technik. Der erste ↑Rechner mit einer grafischen Benutzungsoberfläche (↑GUI), die per Maus angesteuert wurde. Auf Basis von ↑Ethernet bildeten mehrere dieser Rechner ein ↑Rechnernetz, das unter anderem durch ↑Pilot betrieben wurde. Das System war prägend für den ↑Macintosh, mit dem grafische Benutzungsoberflächen erst eine Dekade später den Durchbruch erreichten.

XNU Abkürzung für (en.) *X is Not Unix*. Verkörperung des auf ↑Mach und ↑FreeBSD zurückgehenden ↑Betriebssystemkerns für NeXTSTEP, einem von NeXT 1989 (für ↑m68k) herausgegebenen und bis 1995 (u.a. auch für ↑x86 und SPARC) vertriebenen Mehrplatz-/Mehrbenutzerbetriebssystem. Zentraler Bestandteil von ↑macOS ab Version X und ↑iOS.

Z1 Mechanischer ↑Rechner (Konrad Zuse, 1937) mit getrennten Prozessor- und Speichereinheiten (wie bei einer ↑von Neumann Architektur), wobei das Programm in einem ↑Lochstreifen (genauer: gelochter Filmstreifen) untergebracht und so erstmalig frei programmierbar war.

Zeiger (en.) ↑*pointer*. Gemeinhin ein besonderer Wert, der als ↑Adresse zu interpretieren ist und auf einen anderen Wert (↑Text, ↑Daten) verweist. Um letzteren zu erhalten ist ersterer zunächst zu dereferenzieren (Operatoren `()` bzw. `*` und `->` in ↑C). Eingeführt mit ↑PL/1.

Zeilendrucker (en.) ↑*line printer*. ↑Peripheriegerät das alle Zeichen einer Zeile gleichzeitig druckt. Die durch einen zugehörigen ↑Gerätetreiber zur Ausgabe nacheinander mittels ↑Ein-/Ausgaberegister bereitgestellten ↑Daten werden zeilenweise in dem Gerät gepuffert und dann auf ↑Tabellierpapier ausgegeben. Der Gerätetreiber ist für gewöhnlich ein ↑Unterprogramm von einem Steuerprogramm zur Organisation und Überwachung des Rechnerbetriebs (↑*resident monitor*). Falls ↑abgesetzter Betrieb geführt wird, ist das Gerät selbst am ↑Satellitenrechner angeschlossen, ansonsten am ↑Hauptrechner.

Zeilenvorschub (en.) ↑*line feed*. Setzen der ↑Schreibmarke auf die nächste Zeile, ohne dabei die aktuelle Position des Läufers (↑*cursor*), die er innerhalb der vorigen Zeile besaß, zu verändern.
Ursprüngliche Bezeichnung für das Drehen der Walze bei einer Schreibmaschine, um das Papier eine Zeile vorwärts zu transportieren. Für den ↑Fernschreiber wurde für einen solchen Zeilenwechsel ein spezielles Steuerzeichen (↑LF) eingeführt, das nach wie vor den Zeilenumbruch in einer ↑Dialogstation bewirkt.

Zeiteinheit (en.) ↑*time unit*. Bezeichnung für die der Zeitmessung im ↑Rechensystem zugrunde liegenden Einheit. Die kleinste Auflösung ist für gewöhnlich durch die Fähigkeiten des in diesem System vorhandenen ↑Zeitgebers vorgegeben, lässt sich zumeist jedoch durch das ↑Betriebssystem noch für gröbere Werte skalieren.

Zeitfenster Zeitspanne fester Länge. Ein begrenztes Zeitkontingent für ein bestimmtes ↑Ereignis oder eine Folge festgelegter Geschehnisse.

Zeitgeber (en.) ↑*real-time clock*, ↑*timer*. Bezeichnung für ein spezielles ↑Peripheriegerät, mit dem zeitlich gebundene Vorgänge geregelt werden können (↑RTC, ↑PIT). Typischerweise sendet das Gerät eine ↑Unterbrechungsanforderung an die ↑CPU, wenn ein vom ↑Gerätetreiber vorprogrammiertes Zeitintervall (↑*interval timer*) abgelaufen ist. Für gewöhnlich lässt das Gerät sich aber auch durch zyklische Abfrage betreiben (↑*polling mode*).

zeitgesteuertes System (dt.) ↑*time-triggered system*. Ein ↑Rechensystem dessen ↑Echtzeitbetrieb jede anstehende ↑Aufgabe zu dem jeweils für sie fest vorgegebenen Zeitpunkt durchführt. Auch bezeichnet als taktgesteuertes System.

Zeitnische (en.) ↑*slot*. Ursprünglich ein Begriff aus der Luftfahrt für ein vorher festgelegtes Zeitfenster, in dem eine Fluggesellschaft für eins ihrer Flugzeuge einen Flughafen zum Starten oder Landen benutzen kann. In die Informations- und Kommunikationstechnik übernommen bezeichnet der Begriff ein bestimmtes Zeitfenster, während dessen ↑Daten über ein gemeinsames Medium übertragen werden können. Zeitpunkt und Breite (in Zeiteinheiten: ↑*slot time*) dieses Fensters können fest (*statisch*) oder variabel (*dynamisch*) vorgegeben sein.

Zeitpuffer (en.) ↑*time buffer*. Als Zeitreserve eingeplanter freier zeitlicher Zwischenraum zwischen ↑Terminen, Punkten eines Zeitplans (Duden). In Bezug auf einen ↑Prozess entspricht die damit definierte Zeitspanne einer bestimmten ↑Schlupfzeit.

Zeitquantum (en.) ↑*time quantum*. Bestimmte, einem ↑Prozess oder einer Gruppe von Prozessen zukommende Menge von ↑Zeiteinheiten, die ein ↑wiederverwendbares Betriebsmittel exklusiv zur freien Nutzung Verfügung steht, bevor das ↑Betriebssystem über die Neuzuteilung entscheidet. Im Falle des ↑Prozessors entspricht eine solche Menge der für manche Strategien zur ↑Ablaufplanung benötigten ↑Zeitscheibe (z.B. ↑RR oder ↑VRR).

Zeitreihe Bezeichnung für eine Folge von Zeitgrößen, die nach einer bestimmten Gesetzmäßigkeit, in einem bestimmten regelmäßigen Abstand aufeinanderfolgen (in Anlehnung an den Duden). Eine geordnete Aufstellung von Zeitlängen. Typisches Beispiel einer solchen Reihe wäre eine Aufstellung der Stoßlängen (↑Rechenstoß oder ↑Ein-/Ausgabestoß), die ein jeder ↑Prozess innerhalb einer bestimmt Zeitspanne hervorbringt.

Zeitreihenanalyse Ermittlung der Einzelbestandteile einer ↑Zeitreihe mittels empirischer Methoden, mit dem Ziel, durch inferenzstatistische Untersuchung eine Vorhersage über die weitere Entwicklung dieser Folge treffen zu können. Die einzelnen Zeitgrößen beruhen auf Beobachtung oder Erfahrung, sie repräsentieren Werte, die bei der Nutzung von ↑Betriebsmitteln erhoben und pro ↑Prozess erfasst werden.
Typisches Beispiel eines solchen Nutzungswerts ist die jeweilige Stoßlänge eines Prozesses, und zwar sowohl ↑Rechenstoß als auch ↑Ein-/Ausgabestoß. Für gewöhnlich bringt ein Prozess mehrere solcher Stöße abwechselnd hervor. Mit jeder Stoßart wird der Prozess eine eigene Zeitreihe bilden, deren jeweilige Gliederanzahl prozessabhängig ist und daher auch als verschieden beziehungsweise variabel gilt. Zeitreihen mit eher langen Rechenstößen beziehungsweise kurzen Ein-/Ausgabestößen deuten auf rechenintensive Prozesse, die den ↑Prozessor häufig lange für Berechnungen belegen (↑CPU *bound*). Demgegenüber deuten Zeitreihen mit eher langen Ein-/Ausgabestößen an, dass hier ein-/ausgabeintensive Prozesse am Werke sind (↑I/O *bound*). Schließlich lassen Zeitreihen mit eher kurzen Rechenstößen vermuten, vermehrt interaktive Prozesse am Laufen zu haben. Diese durch Untersuchung einzelner Zeitreihen mögliche Charakterisierung von Prozessen kann von der ↑Ablaufplanung in verschiedener Hinsicht gewinnbringend genutzt werden, beispielsweise um die Auslastung der ↑CPU oder der ↑Peripherie zu maximieren oder die Ansprechempfindlichkeit des ↑Betriebssystems zu verbessern.
Ist der Nutzungswert eines von einem Prozess zu belegenden Betriebsmittels unbekannt, entscheidet dieser Wert jedoch über die ↑Dringlichkeit der Betriebsmittelzuteilung, kann aus entsprechend beobachteten Werten in der Vergangenheit — oder während eines ↑Produktionsbetriebs — auf den zu veranschlagenden Wert in der Zukunft geschlossen werden. Dabei wird angenommen, dass das Nutzungsverhalten eines Prozesses in Bezug auf ein bestimmtes Betriebsmittel mit gewisser Wahrscheinlichkeit auch zukünftig gleich bleibt. So liegt es nahe, den zu erwartenden Nutzungswert s_{n+1} als *arithmetisches Mittel* der bisher wirklich gemessenen Nutzungswerte s_i, $1 \leq i \leq n$, zu definieren:

$$s_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} t_i$$

Der Nachteil dieser Lösung ist jedoch, dass sie eben alle Phasen, in denen ein Prozess ein bestimmtes Betriebsmittel genutzt hat, gleich bewertet. Lange zurückliegende Aktivitäten des Prozesses haben damit die gleiche Wichtigkeit wie seine zum gegenwärtigen Zeitpunkt stattfindenden Aktivitäten. Ein besserer Ansatz ist es daher, Nutzungswerte mit zunehmenden Abstand („Alter“) vom gegenwärtigen Zeitpunkt schwächer zu gewichten:

$$s^*_{n+1} = \frac{1}{w} \cdot s_{n+1} + \frac{w-1}{w} \cdot s^*_n,$$

wobei $1/w$ den *Wichtungsfaktor* α beschreibt. Die Prognose wird jedoch viel direkter das aktuelle Prozessverhalten reflektieren, wenn die Mittlung nicht den Mittelwert s_{n+1} aller bisher gemessenen Nutzungswerte betrifft, sondern lediglich den jeweils zuletzt gemessenen

Wert t_n (↑exponentielle Glättung):

$$s^*_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot s^*_n, \text{ mit } 0 \leq \alpha \leq 1$$

Abschätzungen dieser Art sind typisch für die ↑mitlaufende Planung von Prozessen auf Grundlage ihrer jeweiligen Rechenstoßlängen, beispielsweise wie im Falle von ↑SPN.

Zeitscheibe (en.) ↑*time slice*. Zeitintervall, für das ein ↑Prozess den ↑Prozessor vom ↑Planer zugeteilt bekommt.

Zeitschlitz (en.) ↑*time slot*. Ein periodischer Zeitabschnitt fester Länge zur regelmäßig wiederkehrenden Nutzung von ↑Betriebsmitteln. Eine ↑Zeitscheibe.

Zeitsperre (en.) ↑*timeout*. Zeitbeschränkung für einen ↑Prozess, zum Ausdruck gebacht durch eine vorgegebene Zeitspanne, für die bis zum Eintritt von einem bestimmten ↑Ereignis gewartet werden soll.

Zeitstempel (en.) ↑*time stamp*. Markierung des Zeitpunkts einer Eingabe, Veränderung, Speicherung oder Ähnliche von ↑Daten (Duden). Für gewöhnlich wird erwartet, dass diese Markierung in Bezug auf ihre Stelle im Zeitverlauf eindeutig ist, so dass eine ↑Kausalordnung von ↑Ereignissen definiert werden kann. Dazu muss keine Uhrzeit genommen werden, es genügt ein Zähler. Nicht selten ist letzterer sogar besser geeignet, da dann die Unterscheidbarkeit zweier solcher nacheinander vorgenommenen Markierungen nicht von der Genauigkeit der Uhr abhängt. Allerdings muss der Wertebereich des Zählers ausreichen, um ausnahmslos jedes relevante Ereignis aufzählen zu können.

Exkurs Für ↑x86 (ab Modell Pentium) kann der in einem 64-Bit breiten ↑Prozessorregister enthaltene Zeitstempelzähler (*time-stamp counter*, TSC) durch einen ↑Maschinenbefehl ausgelesen werden. Der TSC zählt die Prozessorzyklen seit Rückstellung (*reset*) des Prozessors. Die Operation dazu gestaltet sich wie folgt:

```
inline time_t timestamp() {              /* read time-stamp counter (TSC) */
    time_t tick;

    asm volatile("rdtsc" : "=A" (tick) : : "%eax", "%edx");

    return tick;                         /* deliver 64-bit time-stamp value */
}
```

Als inzeilige Anweisung resultiert diese Funktion in nur einen Maschinenbefehl (`rdtsc`), der im hier angenommenen 32-Bit Modus den Zählerwert in das Prozessorregisterpaar `edx:eax` überträgt, die höherwertigen 32 Bits in `edx` und die niederwertigen 32 Bits in `eax`.
Die ↑Zykluszeit ist definiert als 1/↑Taktfrequenz. Bei einer Frequenz von 1 GHz, wird der TSC nicht vor 584 Jahren durchgehenden Betrieb seit der letzten Rückstellung überlaufen — im konkreten Fall des Prozessors (Intel Core i7, 2,2 GHz), auf dem dieses Dokument erstellt wurde, kann ab dem Zeitpunkt beruhigt 265 Jahre in die Zukunft geblickt werden.
Der theoretisch mögliche Zählerüberlauf wird damit in praktischer Hinsicht kein Problem bereiten, um aus einem Zeitstempel, der letztlich einer Prozessortaktnummer entspricht, eine Zeit oder einen Zeitpunkt zu berechnen. Allerdings ergeben sich Schwierigkeiten bei einer solchen Zeitrechnung, wenn die Taktfrequenz des Prozessors variiert. Aus Gründen der Energieeffizienz, aber auch um zu starkem thermischen Stress oder gar Überhitzung vorzubeugen, wird die Prozessorgeschwindigkeit verschiedentlich gedrosselt (↑*throttling*). Als Folge verlängert sich die Zykluszeit. Ist allein das ↑Betriebssystem für die Einstellung der Taktfrequenz verantwortlich, kann pro ↑Prozess vermerkt werden, in welcher Phase dieser mit welcher Drosselstufe stattgefunden hat. Jedoch ändern manche Prozessoren ihre Taktfrequenz auch eigenständig, wodurch eine Zeitrechnung allein auf Zeitstempelbasis ungenau sein kann.

Zeitteilverfahren (en.) ↑*time sharing*. ↑Multiplexverfahren, das jedem ↑Prozess immer nur für ein bestimmtes Zeitintervall (↑*time slice*) den ↑Prozessor zuteilt. Technische Grundlage dafür bildet ein von dem ↑Planer benutzter ↑Zeitgeber. Der Ablauf des Zeitintervalls bewirkt

eine ↑Unterbrechung des laufenden Prozesses, woraufhin der Planer als Teil der ↑Unterbrechungsbehandlung aufgerufen wird. In dieser Situation wird der Planer dem unterbrochenen Prozess bedingt den Prozessor entziehen, das heißt, eventuell einen ↑Prozesswechsel erzwingen: der unterbrochene Prozess wird vom Prozessor verdrängt (↑*preemption*). Dies geschieht jedoch nur, wenn neben dem logisch noch laufenden (↑Prozesszustand), aber tatsächlich unterbrochenen, Prozess wenigstens ein weiterer Prozess bereit zur ↑Einlastung zur Verfügung steht und dieser keine niedrigere Priorität als der unterbrochene Prozess besitzt. In dem Fall wird der unterbrochene Prozess auf die ↑Bereitliste gesetzt und ein anderer Prozess (mit gleicher/höherer Priorität), zu dem dann als nächster gewechselt werden soll, davon herunter genommen. Gibt es keinen solchen Prozess, wird der unterbrochene Prozess für ein weiteres Zeitintervall fortgesetzt. Je nach Verfahren ist das Zeitintervall fest oder variabel.

Zentraleinheit (en.) ↑*central processing unit*. Mittelpunkt von einem ↑Rechensystem in logischer Hinsicht, durch dem alle anderen Komponenten der ↑Peripherie kontrolliert und gesteuert werden.

Zufallsgenerator (en.) ↑*random generator*. Gerät (↑*random device*) oder ↑Programm, das etwas, besonders Zahlen, mithilfe der Zufallsauswahl auswählt (Duden). Für gewöhnlich definiert der Generator einen bestimmten Bereich, aus dem die Zufallsauswahl getroffen wird. Im Falle von Zahlen ist dies ein Wertebereich mit fester unterer und oberer Grenze.
Die Zufallsauswahl kann *deterministisch* oder *nicht-deterministisch* geschehen. Erstere Verfahrensweise trifft bei gleichem Ausgangszustand immer dieselbe Zufallsauswahl in Folge, im Gegensatz zur nicht-deterministischen Variante. Dies ist typisch für einen in Programmform vorliegenden Generator, der daher einen ersten Zufallswert „von außen“ erhalten muss. Dieser Wert bildet dann den Keim (*seed*) für die weitere Zufallsauswahl. Ein nicht-deterministischer Generator greift von selbst auf einen externen Vorgang zurück, typischerweise ein *physikalischer Prozess* wie beispielsweise thermisches Rauschen oder radioaktiver Zerfall. Dieser Vorgang liefert einen gemeinhin unvorhersehbaren numerischen Wert als Keim.
Für die Keimbildung eines nicht-deterministisch arbeitenden Generators ist oftmals eine Operation zur ↑Ein-/Ausgabe durchzuführen. Je nach Einbindung der ↑Ein-/Ausgaberegister in den ↑Adressraum eines ↑Prozesses ist diese Operation gegebenenfalls nur als ↑privilegierter Befehl ausgelegt und muss dann vom ↑Betriebssystem als ↑Systemaufruf verfügbar gemacht werden. Demgegenüber bieten manche ↑Prozessoren spezielle ↑Maschinenbefehle für die Zufallsauswahl an, beispielsweise der mit ↑x86 (IA-32, Intel 64) für 16-, 32- und 64-Bit Operanden definierte Befehl `rdrand`. Mit solch einem Befehl kann entweder der Keimwert für den Generator gebildet oder der Generator selbst repräsentiert werden. ↑GCC definiert dafür die Standardfunktion (*intrinsic function*) `__builtin_ia32_rdrand`Φ`_step(ref)`, mit $\Phi \in \{16, 32, 64\}$, und bildet diese entsprechend ab.

Exkurs Beispielhaft wird hier ein Generator für x86 vorgestellt, der den über eine Standardfunktion nutzbaren `rdrand` Befehl verwendet. Der ↑Datentyp für die von diesem Generator ausgewählten Zahlen ist wie folgt definiert:

```
typedef uint64_t chance_t;                /* assume largest values margin */
```

Ein ↑Exemplar der ausgewählten Zufallszahl entspricht damit dem größtmöglichen ↑Speicherwort, das dieser Generator bedienen kann. Für die verschiedenen ↑Wortbreiten gibt es verschiedene Befehle beziehungsweise Standardfunktionen, die durch folgenden ↑Makrobefehl zugänglich gemacht werden:

```
#define coin(ref, phi) __builtin_ia32_rdrand ## phi ## _step(ref)
```

Die `##` Operatoren sorgen für die Verkettung der jeweils links und rechts stehenden Zeichenketten. So dient die als Argument (`phi`) spezifizierte Wortbreite des jeweiligen Befehls der Vervollständigung des ↑Namens der zu verwendenden Standardfunktion.
Die hier vorgenommene *funktionale Abstraktion* erleichtert nicht nur den Umgang mit den verschiedenen Standardfunktionen, sondern verdeckt überhaupt auch die Abhängigkeit von

speziellen Merkmalen eines ↑Kompilierers. Letztlich führt diese Abstraktion eine allgemeine Plattformunabhängigkeit ein, sodass eben auch spezielle ↑Assemblieranweisungen, Zugriffe auf Ein-/Ausgaberegister oder ein Systemaufruf in dem Makro formuliert sein können.
Ergebnis der Konstruktion ist schließlich ein *hybrider Generator*, der einerseits einen „physikalischen Prozess“ im Prozessor bemüht und andererseits als Programm implementiert ist. Letzteres ist hier sehr einfach gehalten, es ruft lediglich einen x-beliebigen (*any-old*) Zahlenwert ab und liefert diesen als Ergebnis der Zufallsauswahl. Dazu wird zur Verwendung und Parametrisierung des Makros (`coin`) eine Fallunterscheidung wie folgt getroffen:

```
inline chance_t anyold(int size) {                      /* random device */
    chance_t luck;                                        /* placeholder */
    if (size == sizeof(int64_t))                              /* 64-bit? */
        coin(64, (uint64_t *)&luck);          /* yes, toss the proper coin */
    else if (size == sizeof(int32_t))                         /* 32-bit? */
        coin(32, (uint32_t *)&luck);          /* yes, toss the proper coin */
    else                                      /* must be 16-bit or smaller */
        coin(16, (uint16_t *)&luck);               /* toss the proper coin */
    return !size ? luck & 1 : luck;                      /* random number */
}
```

Durch *inzeiliges kompilieren* (`inline`) wertet der Kompilierer bereits das Argument (`size`) aus und führt auch selbst die Fallunterscheidung durch, anstatt dies zur ↑Laufzeit des kompilierten Programms der ↑CPU zu überlassen. Ein „Aufruf“ dieser Funktion (`coin`) mündet somit direkt in die Ausführung der Variante des Maschinenbefehls (`rdrand`), die durch die Typgröße als Argument spezifiziert wird. Bei Typgröße 0 liefert die Funktion einen zufälligen Boolschen Wert.

Zugriffsfehler (en.) ↑*access fault*. Allgemein der Fehlgriff eines ↑Prozesses bei der Adressierung einer ↑Ressource, für gewöhnlich eine ↑Ausnahmesituation für den Prozess. Typische Fehlgriffe speziell in Bezug auf einen ↑Prozessadressraum sind etwa ↑Seitenfehler, ↑Segmentfehler, ↑Segmentierungsfehler, ↑Speicherzugriffsfehler, ↑Schutzfehler, ↑Bindungsfehler, aber auch ↑Busfehler.

Zugriffskontrollliste (en.) ↑*access control list*. Datenstruktur, die für jedes ↑Subjekt das ↑Zugriffsrecht auf ein und dasselbe ↑Objekt speichert: die Liste ist physischer Bestandteil des Objekts und nicht des Subjekts (im Gegensatz zur ↑Befähigung). Dabei ist ein Listeneintrag ein Paar von ↑Prozessidentifikation (Subjekt) und Zugriffsrecht. Bei Auslösung einer ↑Aktion bezogen auf ein bestimmtes Objekt wird der Listeneintrag anhand der Identifikation für den betreffenden ↑Prozess aufgesucht: der Auftrag des Prozesses für die gewünschte Aktion durchläuft eine Eingangsprüfung. Schlägt die Suche fehl oder ist das im gefundenen Listeneintrag verbuchte Zugriffsrecht für den Prozess unzureichend, tritt eine ↑Ausnahmesituation ein. Steht der Prozess gegebenenfalls mehrmals auf der Liste, wird für gewöhnlich der zuerst gefundene Listeneintrag für ihn zur Rechteüberprüfung herangezogen: der zuerst gelistete Eintrag für einen Prozess dominiert nachfolgende Einträge desselben Prozesses. Der Widerruf (*revocation*) eines Zugriffsrechts ist einfach und bedeutet lediglich, einen bestimmten Listeneintrag zu löschen. Für gewöhnlich wird in dem Fall immer der dominierende Listeneintrag gelöscht. Auch der Widerruf aller Zugriffsrechte eines Prozesses auf dasselbe Objekt ist leicht möglich, indem lediglich alle Listeneinträge dieses Prozesses für das Objekt zu löschen sind.

Zugriffsmatrix (en.) ↑*access matrix*. System, das einzelne Faktoren von ↑Subjekt und ↑Objekt darstellt und zur verkürzten Beschreibung von Zugriffsrechten dient. Typischerweise sind die Spalten der Matrix durch eine endliche Menge von Objekten und die Zeilen der Matrix durch eine endliche Menge von Subjekten definiert. Jeder Matrixeintrag beschreibt dann die Menge von Zugriffsrechten, die ein Subjekt auf ein Objekt besitzt.
Für den praktischen Einsatz in einem ↑Betriebssystem, um nämlich die Zugriffsrechte von

jedem ↑Prozess (Subjekt) auf die vorhandenen ↑Betriebsmittel (Objekte) zu beschreiben, ist die Matrixdarstellung ungeeignet. Da die Prozess für gewöhnlich nur auf einen Teil der Betriebsmittel zugreifen, wäre die Matrix nur sehr dünn besetzt. Um Speicherplatz einzusparen, wird die Matrix daher entweder zeilen- oder spaltenweise abgelegt. Im Falle der zeilenweise Speicherung bedeutet dies dann, dass jedem Prozess eine ↑Befähigung auf die von ihm verwendeten Betriebsmittel gegeben wird. Wohingegen die spaltenweise Speicherung jedem Betriebsmittel eine ↑Zugriffskontrollliste zuordnet, in der dann nur die Prozesse verzeichnet sind, die dieses Betriebsmittel verwenden wollen.

Zugriffsrecht (en.) ↑*access right*. Erlaubnis, Dinge oder Rechte zu nutzen, die nicht allgemein zugänglich sind (Duden). Das Ding ist ein ↑Objekt und das Recht gibt an, dass und gegebenenfalls auch in welcher Weise auf dieses Objekt durch einen ↑Prozess zugegriffen werden darf. Bereits Kenntnis von der ↑Adresse eines Objekts kann einem Prozess das Recht zum Zugriff darauf gewähren (↑Einzeladressraum). Für gewöhnlich trifft das jedoch nur auf Adressen zu, die (a) nur schwer oder nicht erraten werden können und (b) einem ↑Adressbereich angehören, dessen Größe ein bloßes Ausprobieren von Adressen impraktikabel macht. Das Recht allein nur zum Zugriff kann einem Prozess genommen oder gezielt erteilt werden (↑logischer Adressraum). Eine weitere Differenzierung ist über die Art des Zugriffs möglich, beispielsweise lesen, schreiben und ausführen für sehr einfache Objekte als ↑Seite oder ↑Segment in einem ↑Prozessadressraum. Darüberhinaus können komplexe Operationen wie etwa erzeugen, eröffnen, vergrößern, verkleinern, verschmelzen, aufteilen, duplizieren oder zerstören weitere Objektrechte definieren. Die möglichen Rechte sind somit von dem Typ des Objektes selbst abhängig: so wird eine einzelne ↑Speicherzelle oder ein ↑Maschinenwort nur gelesen oder beschrieben werden können, für einen ↑Verbund oder eine ↑Datei dagegen können weit umfangreichere Rechte gelten.

Zugriffszeit (en.) ↑*access time*. Zeitspanne von der Auslösung bis zur Ausführung einer Operation oder Operationsfolge.

zurückgestellter Prozeduraufruf (en.) ↑*deferred procedure call*. Mechanismus in einem ↑Betriebssystem (urspr. ↑Windows NT), um vorangigen ↑Prozessen die nachgeschaltete Durchführung von nachrangigen ↑Aufgaben zu ermöglichen und diese dadurch nicht unnötig zu verzögern. Ein typisches Beispiel ist das Zusammenspiel zwischen einem ↑FLIH und seinem ↑SLIH, wobei letzterer eine Prozedur darstellt, deren Aufruf zurückgestellt ist.

Exkurs Die mit diesem Mechanismus verknüpfte Prozedur ist nichts anderes als ein ↑Unterprogramm, mit dem eine zusätzliche Berechnung in ein anderes (Unter-) ↑Programm eingefügt werden kann (auch als „*thunk*“ bezeichnet). Diese Berechnung lässt sich am einfachsten durch einen ↑Deskriptor repräsentieren, der die Startadresse und einen (optionalen) Parameter der entsprechenden Prozedur enthält:

```
typedef struct {
    void (*code)(data_t);                          /* address of procedure */
    data_t data;                                 /* parameter of procedure */
} thunk_t;
```

Der Aufruf einer solchen Prozedur geschieht immer indirekt über den zugehörigen und passend definierten Deskriptor. Damit als ↑Objekt ausgelegt, wartet die Prozedur darauf, dass ihre Arbeitsaufnahme durch „anpiepsen“ (*bleep*) geschieht:

```
inline void bleep(thunk_t *call) {
    (*call->code)(call->data);                            /* call procedure */
}
```

Um die Deskriptordefinition nicht nur statisch, sondern auch dynamisch erfolgen lassen zu können, bietet sich ein als Prozedur ausgelegter *Konstruktor* an, der die für den späteren Aufruf benötigten Informationen passend verpackt (*crate*):

```
inline void crate(thunk_t *item, void (*text)(data_t), data_t data) {
    item->code = text;
    item->data = data;
}
```

Zur Zurückstellung des Aufrufs solch einer Prozedur ist der zugehörige Deskriptor nur noch für eine Verkettung (*chain*, s. auch S. 374) aufzubereiten und ensprechend zu erweitern, wodurch zu gegebener Zeit seine Einreihung in eine ↑Schlange erreichbar ist:

```
typedef struct {
    chain_t next;                                /* subsequent queue entry */
    thunk_t call;                                  /* procedure descriptor */
} sequel_t;
```

Im übertragenen Sinne ist damit die mögliche Fortsetzung (*sequel*) einer bestimmten Berechnung als Datenstruktur beschrieben, wobei offen bleibt, ob und gegebenenfalls wann diese Fortsetzung erfolgt. Der Aufruf der Prozedur, die diese Fortsetzung implementiert, geschieht asynchron und ist abhängig von bestimmten ↑Ereignissen, die den jeweils stattfindenden Prozess betreffen (↑*asynchronous system trap*, ↑*re-entry lock*).

Zustandssicherung Maßnahme zum Sicherstellen der augenblicklichen physischen Beschaffenheit von einem ↑Prozess in Bezug auf den ihm zugeteilten ↑Prozessor. Die betrifft wenigstens den ↑Prozessorstatus, kann jedoch auch bestimmte Datenbestände umfassen, die über den ↑Adressraum eines Prozesses zugänglich sind. Beispiel für letzteres ist das Ziehen einer Sicherungskopie (*backup*) von im ↑Hauptspeicher liegende Datenstrukturen, um ein Zurückrollen (*rollback*) des Prozesses im Rahmen der Erholung (*recovery*) nach dem Scheitern einer ↑Aktion oder einem Systemausfall duchführen zu können. Im einfachsten Fall wird mit der Maßnahme nur der Prozessorstatus gesichert, um einen unterbrochenen Prozess vom Prozessor wegschalten und später auf demselben oder einem anderen Prozessor, jedoch mit demselben ↑Programmiermodell, wieder fortsetzen zu können.

Zusteller (en.) ↑*server*. Bezeichnung für einen speziellen ↑Prozess, der eine eingegangene ↑Aufgabe dem dafür geeigneten ↑Prozessor zur Bearbeitung zustellt. Ein solcher Prozess nimmt beispielsweise Anfragen zu ↑Dateien (↑*file server*), zur ↑Ein-/Ausgabe (↑I/O *server*) oder zu ↑HTML-Dokumenten (*web server*) entgegen oder er sorgt im ↑Echtzeitbetrieb etwa dafür, eine ↑nichtperiodische Aufgabe in einen periodischen Ablauf zu integrieren (↑*periodic server*). Dabei ist gemeinhin für jede dieser Arten eine eigene ↑Prozessinkarnation im ↑Rechensystem eingerichtet, zumeist auch nur auf ↑Benutzerebene.
Der die angefragte Aufgabe zu bearbeitende Prozessor ist die ↑CPU (Berechungsaufgaben), ein ↑Peripheriegerät (Ein-/Ausgabeaufgaben) oder auch ein anderer Prozess, genau genommen eine andere Prozessinkarnation (Entkopplung von Aufgabenempfang und -bearbeitung). Durch Verwendung mehrerer interner Prozessinkarnationen (↑*multi-threaded server*) wird die ↑Parallelverarbeitung von Aufgaben ermöglicht und damit die sonst einem einzelnen ↑Handlungsstrang inhärente Sequentialität aufgelöst. Allerdings steht und fällt der Grad der erreichbaren ↑Parallelität mit den direkten und indirekten kausalen Abhängigkeiten zwischen den zugleich zur Bearbeitung bereits anstehenden und den zur Erfüllung einer angefangenen Aufgabenbearbeitung gegebenenfalls noch erforderlichen Aufgabeneingängen. Beispiel für eine Aufgabe, die solche Abhängigkeiten zeigt, ist das Lesen von ↑Daten aus einer gemeinsamen benutzten (*shared*) ↑Datei, die durch wenigstens eine andere Aufgabe, mit der das Schreiben geschieht, erst noch mit ausreichendem Inhalt gefüllt werden muss (↑*producer-consumer problem*). Darüberhinaus muss, um überhaupt das Schreiben zu ermöglichen, auch noch freier Platz in der Datei verfügbar sein.
Derartige kausale Abhängigkeit definieren bestimmte Reihenfolgen der Aufgabenbearbeitung und schränken mögliche ↑Nebenläufigkeit dabei ein. Dieser Aspekt ist vollkommen unabhängig von der Anzahl der im Rechensystem verfügbaren Prozessoren oder ↑Rechenkerne. Ein ↑Multiprozessor oder ↑Mehrkernprozessor allein ist damit längst nicht Garant für höhere Leistungsfähigkeit bei der Aufgabenbearbeitung.

Zwischenankunftszeit (en.) ↑*interarrival time*. Zeitspanne zwischen zwei aufeinanderfolgenden Aufträgen an einer ↑Bedienstation, allgemein zwischen zwei Ereignissen derselben Art. Gemessen über mehrere solcher Aufträge/Ereignisse wird gegebenenfalls unterschieden zwischen Minimal-, Durchschnitts- und Maximalwerten für diese Zeitspanne. So ist beispielsweise im Falle einer ↑Unterbrechung zwar nicht ihr exakter Zeitpunkt vorhersehbar, nicht selten jedoch die *minimale Zeitspanne* zwischen zwei solcher Ereignisse.

Zwischenkode ↑Maschinenkode für eine ↑virtuelle Maschine, der durch einen ↑Interpreter, der in Software realisiert ist (↑CSIM), ausgeführt wird.

Zwischenspeicher (en.) ↑*cache*. Hilfsspeicher: schneller Pufferspeicher zur Verbergung der ↑Latenzzeit für Zugriffe auf den im Vergleich zur ↑Zykluszeit der ↑CPU langsameren ↑Hauptspeicher. Die Puffergröße ist ganzzahlige Vielfache der Größe einer ↑Zwischenspeicherzeile.
Die Originale der ↑Daten liegen im Hauptspeicher, eventuelle Kopien davon sind Bestandteil einer oder auch mehrerer Zwischenspeicherzeilen in diesem Hilfsspeicher. Als Konsequenz daraus kann es zu Inkonistenzen zwischen den Originaldaten einerseits und den zwischengespeicherten Daten andererseits kommen, sollten ↑schreibende Zugriffe unkontrolliert geschehen. Um vor dem Hintergrund einer solchen Speicherorganisation eine konsistente Sicht auf die Daten zu erhalten, erfordern Schreibzugriffe bestimmte Techniken (↑*write-through*, ↑*write-back*, ↑*write-allocate*), die für die Sicherstellung der ↑Speicherkohärenz sorgen.

Zwischenspeicherfehlzugriff (en.) ↑*cache miss*. Fehlzugriff (*miss*) auf eine im ↑Zwischenspeicher gewähnte Informationseinheit (↑Text, ↑Daten). Für die bei dem Zugriff von der ↑CPU im ↑Abruf- und Ausführungszyklus applizierte ↑Adresse ist kein zugehöriger Eintrag im Zwischenspeicher vermerkt. In dem Fall liest der Zwischenspeicher eigenständig den Inhalt von dem ↑Speicherbereich (im ↑Hauptspeicher) ein, in den diese Adresse fällt. Dieser Bereich hat die Größe einer ↑Zwischenspeicherzeile. Ist der Zwischenspeicher voll, überschreibt der Einlesevorgang eine der Zwischenspeicherzeilen, die in jüngster Zeit am wenigsten referenziert wurde (↑LRU). Wurde ein in dieser Zeile liegendes ↑Speicherwort verändert, muss vor Überschreibung der Zeile ihr aktueller Inhalt zurück in den Hauptspeicher gesichert werden. Der Fehlzugriff ist in funktionaler Hinsicht nicht von einem ↑Prozess wahrnehmbar, wohl aber nichtfunktional: ist das Datum aus dem Hauptspeicher zu lesen, kann in dem Fall (ohne Zurückschreiben) leicht eine um den Faktor 40 höhere ↑Latenz (4 vs. 150 Zyklen) zu Buche schlagen (die mit Zurückschreiben doppelt so hoch sein kann).

Zwischenspeicherzeile (en.) ↑*cache line*. Verwaltungseinheit im ↑Zwischenspeicher, deren Größe ganzzahlige Vielfache der Größe von einem ↑Speicherwort ist. Auch die Bezeichnung für die Übertragungseinheit von ↑Daten zwischen Zwischen- und ↑Hauptspeicher. Typische Zeilenlängen sind 32 (↑Cortex-A, Mindestlänge), 64 (↑x86) oder 128 (↑PowerPC) ↑Bytes.
Jede dieser zwischengespeicherten Einheiten ist die Kopie des Inhalts eines entsprechend großen Bereichs im Hauptspeicher. Der Zugriff auf den Inhalt eines nicht im Zwischenspeicher (auf einer solchen Zeile) liegenden Speicherworts bewirkt den blockweisen Transfer der assoziierten Verwaltungseinheit (Zeile). Wichtiger Aspekt für die ↑Performanz dabei ist die ↑Granularität der Zeile und die jeweiligen Eigentümerschaften der Originalspeicherworte in ihr (↑*false sharing*).

Zykluszeit (en.) ↑*cycle time*. Zeitspanne vom Start bis zur Vollendung einer Operation oder Operationsfolge.