Überblick

Vorrangsteuerung
- Prioritäten
- Prioritätsabbildung
- Repräsentation von Ablaufplänen
- Prioritätsverletzung und Prioritätsumkehr

Ereignisorientierte Einplanung
(engl. *event-driven scheduling*)

Einplanung von Arbeitsaufträgen erfolgt zu Ereigniszeitpunkten, die zufällig auftreten und somit nicht vorhersehbar sind
- die Ereignisverarbeitung unterliegt einer gewissen Dringlichkeit
- Ereignisauslöser sind kontrollierte Objekte oder andere Arbeitsaufträge

Ereignisse haben Prioritäten, die dem Ereignisauslöser und/oder der Ereignisverarbeitung zugeordnet sind

feste Zuordnung $\mapsto$ Ereignisverarbeitung/-auslöser
- gibt Arbeitsaufträgen eine absolute Priorität

variable Zuordnung $\mapsto$ Ereignisverarbeitung
- gibt Arbeitsaufträgen eine relative Priorität

☞ auch: prioritätsorientierte Einplanung (engl. *priority-driven scheduling*)

Ereignisorientierter Planer
(engl. *event-driven scheduler*)

Einplanung ereignisbedingt ausgelöster Arbeitsaufträge resultiert in eine dynamische Datenstruktur $\mapsto$ „sortierte Liste“
- der Vorgang ist gekoppelt mit der Einlastung: *online scheduling*
- kritisch ist die Berechnungskomplexität und wann sie anfällt
  - konstant oder variabel, dann jedoch mit oberer Schranke $\mapsto$ WCET
  - zum Auslöse- oder Auswahlzeitpunkt von Arbeitsaufträgen

Sortierschlüssel (engl. *sort key*) zur Reihung eines Arbeitsauftrags ist die ihm zugeordnete Priorität
- ergibt sich ggf. erst zum Ereigniszeitpunkt aus der Priorität der von ihm zu verarbeitenden Klasse von Ereignissen
- ist eindeutig abzubilden auf einen endlichen Wertebereich

☞ auch: prioritätsorientierter Planer (engl. *priority-driven scheduler*)

Berechnungskomplexität
Zeitpunkte ihrer Wirksamwerdung

Auslösezeitpunkt (nach dem Ereigniszeitpunkt)
- konstanter Aufwand, im Falle einer Ablauftabelle
  - Jobs durch indizierte Adressierung in die Tabelle aufnehmen
  - ggf. ist ein Tabelleneintrag eine FCFS-Liste von Jobs gleicher Priorität
- (beschränkter) linearer Aufwand, im Falle einer Ablaufliste
  - Vorabwissen zur WCET des Sortiervorgangs ist gefordert

Auswahlzeitpunkt (nach dem Auslöse- und vor dem Einlastungszeitpunkt)
- nahezu konstanter Aufwand, im Falle einer Ablaufliste
  - Jobs vom Kopf her der (ggf. einfach verketteten) Liste entnehmen
- beschränkter linearer Aufwand, im Falle eine Ablauftabelle
  - Vorabwissen zur WCET des Suchvorgangs ist gefordert
  - Tabelleneinträge können leer sein und sind zu überspringen

Berechnungskomplexität (Forts.)

Ablaufliste vs. Ablauftabelle

```
template<class Job>
class Schedule {
    Job* list;
public:
    void release (Job& item) {
        Job* last = 0;
        Job* tail = list;

        while (tail && tail->outrank(item))
            tail = (last = tail)->next();

        if (!last) {
            item.next(list); list = &item;
        } else {
            item.next(tail); last->next(&item);
        }
    }

    Job* extract () {
        Job* item;
        if ((item = list)) list = item->next();
        return item;
    }
};
```

`release` $O(n)$

`extract` nahezu $O(1)$

```
template<class Job, unsigned Jobs>
class Schedule {
    Job table[Jobs];
public:
    void release (Job& item) {
        assert((&item >= &table[0])
            && (&item <= &table[Jobs - 1]));
        item.state(Job::Ready);
    }

    Job* extract () {
        for (unsigned slot = 0; slot < Jobs; slot++)
            if (table[slot].state() == Job::Ready) {
                table[slot].state(Job::Selected);
                return &table[slot];
            }

        return 0;
    }
};
```

`release` $O(1)$

`extract` $O(n)$, obere Schranke `Jobs`

Prioritätsorientierte Algorithmen

Klassifikation

Verfahren zur prioritätsorientierten Einplanung periodischer Jobs werden in zwei Gruppen eingeteilt:

feste Priorität (engl. *fixed priority*)

- alle Jobs einer Task haben dieselbe Priorität
- d.h., die Taskpriorität ist relativ zu anderen Tasks fest, unabhängig von der Auslösung bzw. Beendigung von Jobs

dynamische Priorität (engl. *dynamic priority*)

- die Jobs einer Task können verschiedene Prioritäten haben
- d.h., die Taskpriorität variiert relativ zu anderen Tasks, wenn Jobs ausgelöst bzw. beendet werden

Unterschiede ergeben sich demnach in der Art und Weise, wie Prioritäten den Jobs zugeordnet werden

- nämlich statisch, wie im Falle fester Prioritäten, oder dynamisch...

Verfeinerte Klassifikation

Ein Frage der Betrachtungsebene...

Praxisrelevanz haben Verfahren, die den Jobs feste Prioritäten zuweisen

- die Zuweisung erfolgt jedoch zum Auslösezeitpunkt eines Jobs
  - d.h., wenn er ereignisbedingt auf die Bereitliste (engl. *ready list*) kommt
- einmal zugewiesen, bleibt die Priorität eines ausgelösten Jobs gleich
  - jedoch immer nur in Relation zu allen anderen Jobs auf der Bereitliste
- auf Jobebene sind die Prioritäten fest, auf Taskebene sind sie variabel

Konsequenz daraus sind drei Kategorien von Einplanungsalgorithmen:

feste Priorität wie gehabt (S. 6- 6)

dynamische Priorität auf Taskebene (engl. *task-level dynamic-priority*) und
feste Priorität auf Jobebene (engl. *job-level fixed-priority*)
dynamische Priorität auf Jobebene (engl. *job-level dynamic-priority*)

☞ dynamische Priorität $\mapsto$ dynamisch auf Task- und fest auf Jobebene

Mehrdeutigkeit von Prioritäten

Anwendungsebene vs. Systemebene

Echtzeitrechensysteme unterstützen typischerweise nur eine begrenzte Anzahl von Prioritätsebenen:

8 im IEEE 802.5 *token ring* [16]

32 im alten QNX, im neuen („Neutrino“) 256 [17]

140 in Linux 2.5 (mit Ebenen 1–100 reserviert für Echtzeitprozesse)

256 in VxWorks [18] und vielen anderen Echtzeitbetriebssystemen

- implementierungsbedingter begrenzter Wertebereich: Bitfeld, `char`

Echtzeitanwendungen können jedoch mehr Prioritätsebenen benötigen, als die gegebene Systemplattform unterstützt

uneindeutige Prioritäten (engl. *nondistinct priorities*) sind die Folge

- die Anzahl unterschiedlicher (d.h., eindeutiger) Task-/Jobprioritäten übersteigt die Anzahl unterschiedlicher Prioritäten im System
- die Task-/Jobprioritäten lassen sich nicht eindeutig abbilden

Prinzip der Prioritätsabbildung
Prioritätsraster (engl. *priority grid*)

ω_n Anzahl (an eine Task/einen Job) zugewiesener Prioritäten
- $1, 2, \ldots, \omega_n$ mit 1 als höchste und ω_n als niedrigste Priorität

ω_s Anzahl der Prioritäten des Systems
- $\pi_1, \pi_2, \ldots, \pi_{\omega_s}$ mit π_k $(1 \leq k \leq \omega_s)$ im Bereich $[1, \omega_n]$
- zusätzlich gilt: $\pi_j < \pi_k$ wenn $j < k$

Menge $\{\pi_1, \pi_2, \ldots, \pi_{\omega_s}\}$ ist Prioritätsraster Π, auf das die zugewiesenen Prioritäten wie folgt abgebildet werden:
- zugewiesene Prioritäten größer gleich π_1 auf π_1
- zugewiesene Prioritäten im Bereich $(\pi_{k-1}, \pi_k]$ auf π_k für $1 < k \leq \omega_s$

☞ die Abbildung kann gleichmäßig oder ungleichmäßig definiert sein

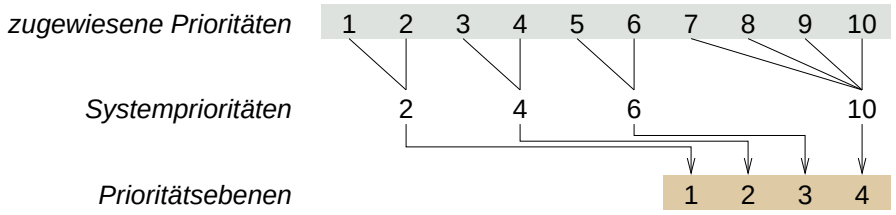
Abbildung durch gleichmäßige Verteilung
(engl. *uniform mapping*)

Prioritätsraster Π uniform auf den Bereich zugewiesener Prioritäten legen
- sei Q definiert als Ganzzahl $\lfloor \omega_n / \omega_s \rfloor$, dann ist die Systempriorität $\pi_k = kQ$ für $k = 1, 2, \ldots, \omega_s - 1$ und $\pi_{\omega_s} = \omega_n$
- das bedeutet für einen Block von max. Q zugewiesenen Prioritäten:
  - die ersten Q höchsten $1, 2, \ldots, Q$ werden abgebildet auf $\pi_1 = Q$
  - die nächsten Q höchsten werden abgebildet auf $\pi_2 = 2Q$
  - usw., bis alle zugewiesenen Prioritäten „gerastert“ worden sind
- Tasks werden dann entsprechend ihrer Systempriorität π_k abgearbeitet

Tasks verschiedener logischer (d.h., zugewiesener) Prioritäten erhalten dieselbe physische Systempriorität, liegen auf einer Prioritätsebene
- die Jobs dieser Tasks sind einer linearen Abbildung unterworfen
- Wichtung erhalten sie durch ihre Position in der linearen Ordnung

Abbildung durch gleichmäßige Verteilung (Forts.)
Querschneidender Belang von Anwendung und System

Beispiel: 10 Tasks mit zugewiesenen Prioritäten $1, 2, \ldots, 10$ und ein System, das nur die Prioritätsebenen $1, 2, 3, 4$ unterstützt



- $[1,\ 2] \mapsto \pi_1 = 2$
- $[3,\ 4] \mapsto \pi_2 = 4$
- $[5,\ 6] \mapsto \pi_3 = 6$
- $[7, 10] \mapsto \pi_4 = 10$

Problem „Fairness“: Tasks höherer zugewiesener Prioritäten werden gleich behandelt wie Tasks mit niedrigeren zugewiesenen Prioritäten.

Abbildung durch ungleichmäßige Verteilung
(engl. *non-uniform mapping*)

Prioritätsraster Π derart auf den Bereich zugewiesener Prioritäten legen, so dass das Verhältnis $(\pi_{i-1} + 1)/\pi_i$ für $i = 2, 3, \ldots, \omega_s$ gleich bleibt
- die Methode wird auch als *constant ratio mapping* [19] bezeichnet
- für höhere zugewiesene Prioritäten werden mehr Prioritätsebenen reserviert als für niedrigere zugewiesene Prioritäten
- resultiert in eine bessere Feinabstufung höher priorisierter Tasks

Beispiel (wie gehabt, S. 6- 11): $\omega_n = 10$, $\omega_s = 4$
- $[1,\ 1] \mapsto \pi_1 = 1$
- $[2,\ 3] \mapsto \pi_2 = 3$
- $[4,\ 6] \mapsto \pi_3 = 6$
- $[7, 10] \mapsto \pi_4 = 10$

- $(\pi_1 + 1)/\pi_2 = 2/3$
- $(\pi_2 + 1)/\pi_3 = 2/3$
- $(\pi_3 + 1)/\pi_4 \approx 2/3$

- 1:1 $\mapsto$ Ebene 1
- 2:1 $\mapsto$ Ebene 2
- 3:1 $\mapsto$ Ebene 3
- 4:1 $\mapsto$ Ebene 4

Relative Planbarkeit

Einfluss der Anzahl von Systemprioritäten auf die Planbarkeit eines Systems

Verschlechterung der Planbarkeit ist zu erwarten, wenn insgesamt zu wenig Systemprioritäten zur Verfügung stehen, d.h., wenn gilt: $\omega_n > \omega_s$

- sei g das Minimum von Verhältniswerten des Prioritätsrasters
  - d.h., $g = \min_{2 \leq i \leq \omega_s}(\pi_{i-1} + 1)/\pi_i$
- im Falle von RM für große n und $D_i = p_i$ für alle i wurde gezeigt [19], dass für die planbare Auslastung (engl. *schedulable utilization*) gilt:

$$\begin{cases} ln(2g) + 1 - g & \text{falls } g > 1/2 \\ g & \text{falls } g \leq 1/2 \end{cases}$$

- das Verhältnis dieser Auslastung zu $ln2$ ist ein Maß für die relative Planbarkeit des gegebenen Systems

Beispiel: 100 000 Tasks (und evtl. noch vielmehr Jobs), d.h., $\omega_n = 100\,000$

- die relative Planbarkeit bei $\omega_s = 256$ ist gleich 0.9986

☞ RM: für komplexeste Tasksysteme reichen bereits 256 Prioritätsebenen

Reihung von Arbeitsaufträgen

Logische Zusammenhänge im Lichte systembedingter Einschränkungen

Ablaufsteuerung zusammen mit Prioritätsabbildung resultieren in einen zweistufigen Ansatz zur Reihung von Arbeitsaufträgen:

1. je nach Ablaufsteuerungsverfahren werden Tasks (bzw. ihren Jobs) eindeutige Prioritäten zugewiesen

   feste Prioritäten $\rightsquigarrow$ RM, DM

   dynamische Prioritäten $\rightsquigarrow$ EDF, LRT, LST

   - Prioritätsschlange (engl. *priority queue*), als Tabelle von ω_n Joblisten
   - eine Schlange/Liste pro Task, zur Erfassung ausgelöster Jobs der Task
2. die durch logische Zwänge in der Anwendung gegebene Reihung wird unter Berücksichtigung der Systemeinschränkungen umgesetzt
   - die Tabellenlänge ggf. auf die Anzahl der Prioritätsebenen begrenzen
   - Schlangen/Listen aufeinanderfolgender Tabelleneinträge vereinen
   - dabei die den Tasks entsprechenden „Jobgruppen“ beibehalten

☞ den Ablaufplan zweidimensional, als indizierte Bereitliste repräsentieren

Indizierte Bereitliste

Prioritätswert als strukturierter Sortierschlüssel

Ablaufsteuerungsverfahren erzeugen eine lineare Ordnung von Tasks/Jobs

- jedes (der betrachteten) Verfahren hat sein eigenes Sortierkriterium:
  - Periode (RM), relativer Termin (DM)
  - absoluter Termin (EDF), Auslösezeit (LRT), Schlupfzeit (LST)
- der Sortierschlüssel besteht aus mehreren Strukturelementen:
  1. Index in eine Tabelle von Listendeskriptoren $O(1)$

     $\omega_n \leq \omega_s \mapsto$ zugewiesene Priorität der Task

     $\omega_n > \omega_s \mapsto$ Systempriorität bzw. Prioritätsebene der Task
  2. relative Position innerhalb der (indizierten) Liste

     $\omega_n \leq \omega_s \mapsto$ zugewiesene Priorität des Jobs $\rightsquigarrow$ LIFO/FIFO $O(1)$

     $\omega_n > \omega_s \mapsto$ zweigeteilte Ordnungszahl des Jobs $O(n)$

     (a) die zugewiesene Priorität seiner Task

     (b) die ihm zugewiesene Priorität innerhalb seiner Task
- der Sortieraufwand ist insgesamt gesehen eher sublinear, $o(n)$

☞ Vorrangsteuerung mit konstantem Aufwand ist nur bedingt möglich !!!

Indizierte Bereitliste (Forts.)

Ablaufplan segmentieren $\rightsquigarrow$ Aufwand begrenzen

1. Dimension $\mapsto$ statische Datenstruktur: Tabelle von Zeiger(paare)n
   - konstanter Aufwand beim Zugriff auf einen Tabelleneintrag
     - Indizierung der Tabelle, Index ist die Prioritätsebene eines Jobs
   - linearer Aufwand bei der Auswahl des nächsten Tabelleneintrags
     - begrenzt durch ω_s bzw. die Anzahl von Prioritätsebenen
     - kleine und feste obere Schranke

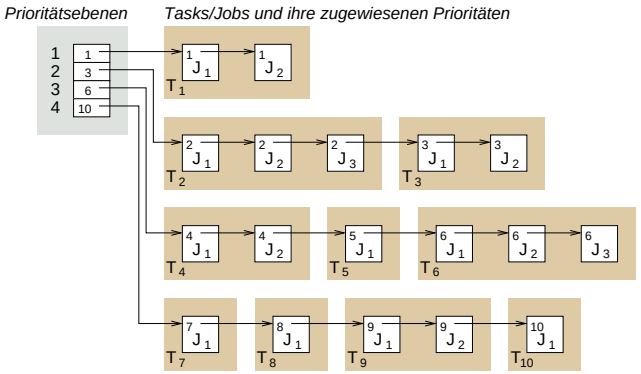
2. Dimension $\mapsto$ dynamische Datenstruktur: Liste/Schlange von Jobs
   - konstanter Aufwand bei der Auswahl des nächsten Jobs
     - Fallunterscheidung entfällt: die Liste/Schlage ist nicht leer
   - linearer Aufwand bei der Bereitstellung eines Jobs
     - begrenzt durch das Verfahren zur Prioritätsabbildung
     - feste obere Schranke ω_n (schlimmster Fall)

☞ auf mehrere Ebenen aufgeteilte Liste/Schlange: *multi-level queue*, MLQ

Indizierte Bereitliste und Prioritätsabbildung

Lineare Ordnung von Tasks/Jobs hinweg über alle Prioritätsebenen

Beispiel (S. 6- 12): angenommen, die 10 Tasks kommen mit 18 Jobs...



Die relative Ordnung zwischen den Jobs verschiedener Tasks derselben Prioritätsebene muss aufrechterhalten werden, ebenso wie die der ggf. mehreren Jobs innerhalb einer Task:

- LIFO/FIFO ist nicht garantiert
- nur bedingt $O(1)$

☞ ω_s steht hier für die Anzahl der Prioritätsebenen = Tabelleneinträge

Indizierte Bereitliste und Prioritätsabbildung (Forts.)

Verschmelzung von Ablaufliste und Ablauftabelle (S. 6- 5)

```
template<class Job, unsigned Levels>
class Schedule {
    Job* table[Levels];
public:
    void release (Job& item) {
        assert(item.level() < Levels);

        Job* last = 0;
        Job* tail = table[item.level()];

        while (tail && tail->outrank(item)) tail = (last = tail)->next();

        if (!last) {
            item.next(table[item.level()]); table[item.level()] = &item;
        } else {
            item.next(tail); last->next(&item);
        }
    }

    Job* extract () {
        Job* item;
        for (Job** slot = &table[0]; slot < &table[Levels]; slot++)
            if (item = *slot) {
                *slot = item->next();
                return item;
            }
        return 0;
    }
};
```

- $O(1)$
- $O(n)$, begrenzt
- $O(n)$, begrenzt
- $O(1)$

Prioritätsorientierter $O(1)$-*Scheduler* !?

Die Tücke liegt oft im Detail...

Jobauslösung mit konstantem Aufwand, $O(1)$, ist möglich, sofern gilt:

1. der Ablaufplan ist eine auf mehrere Prioritätsebenen aufgeteilte dynamische Datenstruktur, repräsentiert als Tabelle von...
   - Wartelisten $\rightsquigarrow$ LIFO
   - Warteschlangen $\rightsquigarrow$ FIFO
2. alle Jobs, die über denselben Tabelleneintrag erfasst werden, besitzen auch dieselbe Priorität $\rightsquigarrow$ Prioritätsschlange
   - sonst könnte LIFO/FIFO Prioritätsverletzung zur Folge haben
3. die Anzahl der Tabelleneinträge entspricht mindestens der Anzahl zugewiesener Jobprioritäten: $\omega_s \geq \omega_n$
   - ggf. werden dann nahezu alle Tabelleneinträge nur einen Job erfassen
   - hängt ab von der Echtzeitanwendung und dem Einplanungsverfahren

Jobauswahl ist unter diesen Bedingungen nicht in $O(1)$ möglich:

- begrenzt viele leere Tabelleneinträge sind ggf. zu überspringen

Prioritätsorientierter $O(1)$-*Scheduler* (Forts.) !?

Eine Abwägungsfrage...

Vorrangsteuerung ist mit einem grundsätzlichen Konflikt konfrontiert:

- entweder Jobauslösung oder Jobauswahl mit $O(1)$ zu versehen
  - beides zugleich geht nicht

...für Jobauslösung in $O(1)$ spricht:

- ereignisgesteuerte Einplanung & Einlastung benötigen konstante Zeit
  - als Folge eines *Interrupts* oder der Zustellung eines „Zeitsignals“
  - bedeutsam für voll-verdrängbare (engl. *full preemptive*) Systeme
- ereignisbedingte Verzögerungen von Jobs lassen sich exakt bestimmen

...für Jobauswahl in $O(1)$ spricht:

- der Übergang zum jeweils nachfolgenden Job benötigt konstante Zeit
  - wenn z.B. der aktuelle Job durchgelaufen ist oder blockiert

☞ Linux 2.6, Mach, QNX, ..., VxWorks verhelfen Jobauslösung zu $O(1)$

Querschneidender Belang „Priorität“

Nichtfunktionale Eigenschaften anderer als einplanender Systemfunktionen

Abstraktion...

de.wikipedia.org/wiki ∼ (von lat. abstrahere: abziehen, wegziehen)
bezeichnet den Prozess rationaler Verarbeitung von konkretem Sinnesmaterial, wobei von bestimmten äußeren, individuellen oder zufälligen Merkmalen, Eigenschaften und Beziehungen des betreffenden Objekts abgesehen wird, andere, allgemeingültige (unsichtbare) strukturelle Eigenschaften dagegen als wesentlich herausgehoben und zugleich variabel oder modellhaft betrachtet werden.

... ist im Kontext von Echtzeitrechensystemen mit Bedacht umzusetzen

- ein Beispiel lieferte die Diskussion zum $O(1)$-Scheduler auf S. 6- 19:
  - die Gefahr von Prioritätsverletzung bei LIFO/FIFO, wenn nicht alle Jobs in der Liste dieselbe Priorität besitzen
  - den Listenanfang hat dann nicht zwingend der höchst priorisierte Job
- überall, wo Jobs auf Warteschlangen stehen, ist dieses Problem akut

Fehlgeleitete Auslösung

Prioritätsverletzung (engl. *priority violation*)

Beispiel: Vergabe wiederverwendbarer/konsumierbarer Betriebsmittel

- jede Betriebsmittelklasse K_i verwaltet eine Warteliste für suspendierte Jobs entsprechend FIFO (bzw. FCFS)
  - z.B. durch einen Semaphor (engl. *semaphore* [20])
- ein anfordernder Job J_l wird suspendiert, wenn zum Zeitpunkt der Anforderung kein Betriebsmittel in K_i verfügbar ist
  - J_l hat eine niedrige Priorität, nimmt Platz i der Warteliste ein
- ereignisbedingt wird ein Job J_h ausgelöst und eingelastet, der ebenfalls wegen Betriebsmittelmangel in K_i bei Anforderung suspendiert wird
  - J_h hat eine hohe Priorität, nimmt Platz $j > i$ der Warteliste ein
- bei Freigabe eines Betriebsmittels in K_i wird der nächste auf dieses Ereignis wartende Job ausgelöst
  - J_l wird vor J_h ausgewählt, die Priorität von J_h wird verletzt

☞ Jobwarteschlangen sind konform zum Einplanungsverfahren zu bedienen

Fehlgeleitete Auslösung (Forts.)

Nichtfunktionale Eigenschaft im Beispiel „FIFO-Semaphor“

Funktion eines Semaphors ist die Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen[17]: Prozess ↦ Job

- ein abstrakter Datentyp, auf dem zwei Operationen definiert sind:
  - P signalisiert die Blockade eines Prozesses, wenn zum gegebenen Zeitpunkt $s = 0$ gilt
    - s wird sonst um 1 erniedrigt oder auf 0 gesetzt
  - V signalisiert die Deblockade eines Prozesses, wenn zum gegebenen Zeitpunkt ein auf s blockierter Prozess existiert
    - s wird sonst um 1 erhöht oder auf 1 gesetzt
- Aufgabe ist nicht die Umsetzung von Prozessblockade/-deblockade
  - ebensowenig, wie Buch über die auf s blockierten Prozesse zu führen
- Buchführung blockierter Prozesse ist eine Optimierungsmaßnahme
  - bei der eine FIFO-Warteschlange für die Semaphorfunktion ideal ist

[17] Prozesse, deren Ausführung sich zeitlich überschneidet.

Abhängigkeit von Aufgaben niedriger Priorität

Selbstaussetzung (engl. *self-suspension*)

Beispiel: ein Fernaufrauf (engl. *remote procedure call* [21]) bzw. aus Klient/Anbieter (engl. *client/server*) bestehende Rechensysteme

Klient ↦ ein Job höherer Priorität
- ruft eine Funktion auf, die von einem anderen Job (ggf. auf einem anderen Rechner) ausgeführt wird
- setzt „freiwillig“ (indirekt) seine weitere Ausführung aus

Anbieter ↦ ein Job niedrigerer Priorität
- führt eine Funktion aus, die von einem anderen Job (ggf. auf einem anderen Rechner) aufgerufen wird
- verzögert den aufrufenden Job ggf. unbestimmt lang

- ein Rückruf (engl. *callback*) ist mit ähnlicher Problematik behaftet
  - beide Konzepte sind nur bedingt transparent für die involvierten Jobs

☞ Prioritätsvererbung (engl. *priority inheritance*) muss betrieben werden

Intervalle von Unverdrängbarkeit

Blockierung, Hemmung (engl. *blocking*)

Beispiel: ein (zeit-) kritischer Abschnitt (engl. *critical section*)

- eine Folge von Anweisungen, deren Ausführung einen gegenseitigen Ausschluss erfordern ⇝ mehrseitige Synchronisation
  - (a) sich vor Überlappung schützen ↦ binärer Semaphor
  - (b) sich vor Verdrängung schützen ↦ Einlastung abschalten
- manche Betriebssysteme unterbinden während der Ausführung von Systemaufrufen die Verdrängung des laufenden Prozesses
  - klassisches UNIX-Modell (`runrun`-Flag) und das vieler UNIX *look-alikes*
- Job J_l läuft auf solch einer Plattform und tätigt einen Systemaufruf
  - J_l hat eine niedrige Priorität, durchläuft unverdrängbar den Kern
- während des Systemaufrufs, wird Job J_h ereignisbedingt ausgelöst
  - J_h hat eine hohe Priorität, wird eingeplant aber nicht eingelastet
- J_l blockiert bzw. hemmt J_h, die Priorität von J_h wird verletzt

☞ Synchronisation ist nicht-funktionale Eigenschaft eines Systemaufrufs

Nebenläufige Zugriffe auf gemeinsame Betriebsmittel

Prioritätsumkehr (engl. *priority inversion*)

Prioritätsumkehr [22] ist Folge der Blockierung eines höher priorisierten Jobs durch einen niedriger priorisierten Job

1. der niedrig priorisierte Job durchläuft einen kritischen Abschnitt und wird vom höher priorisierten Job verdrängt
2. der höher priorisierte Job möchte denselben kritischen Abschnitt betreten, wird vom niedrig priorisierten Job jedoch daran gehindert
3. der niedrig priorisierte Job kann weiter ausgeführt werden, obwohl ein höher priorisierter Job wartet

. . . die das Problem verschärfende Variante bringt weitere Jobs ins Spiel:

4. mittel priorisierte Jobs verdrängen den niedrig priorisierten Job und blockieren indirekt den höher priosisierten Job noch länger

☞ kritischer Abschnitt oder Betriebsmittel: Unteilbarkeit ist das Problem

What really happened on Mars?

Prioritätsumkehr beim *Mars Pathfinder* [23, 24]

bc_sched ↦ Task mit höchster Priorität (mit Ausnahme der VxWorks Task „tExec“)

- kontrollierte den Aufbau der Transaktionen über den „1553“-Bus
- dieser Bus koppelte Fahr- und Landeeinheit der Raumsonde

bc_dist ↦ Task mit dritthöchster Priorität

- steuerte die Einsammlung der Transaktionsergebnisse
- Dateneingabe über doppelt gepufferten gemeinsamen Speicher

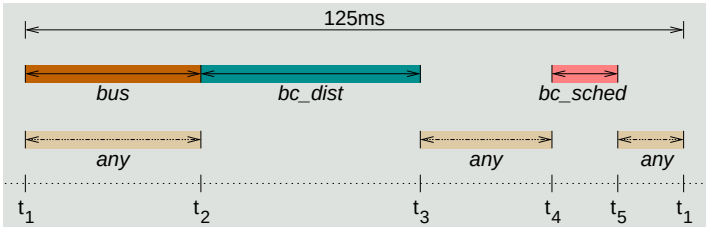
ASI/MET ↦ Task mit sehr niedriger Priorität

- sammelte in seltenen Durchläufen meteorologische Daten ein
- interoperierte mit *bc_dist* (blockierend) auf IPC-Basis

☞ die Hardware gab eine Periodenlänge von 8 Hz (d.h., 125 ms) vor

What really happened on Mars? (Forts.)

Aufbau eines Buszyklus



t_1 Transaktion startet hardware-kontrolliert an einer 8 Hz Grenze

t_2 Busverkehr ist zur Ruhe gekommen, *bc_dist* wird ausgelöst

t_3 *bc_dist* hat die Datenverteilung abgeschlossen

t_4 *bc_sched* wird ausgelöst, setzt Transaktion für nächsten Buszyklus auf

t_5 *bc_sched* hat seine Aufgabe für diesen Zyklus beendet

☞ Intervalle $[t_1, t_2), [t_3, t_4), [t_5, t_1)$ standen u.a. *ASI/MET* zur Verfügung

What really happened on Mars? (Forts.)
Feste Randbedingung

bc_dist muss die Datenverteilung abgeschlossen haben, wenn *bc_sched* ausgelöst wird, um die Transaktion des nächsten Zyklus aufzusetzen:

- stellt *bc_sched* fest, dass *bc_dist* noch nicht abgeschlossen ist, wird ein Total-*reset* durchgeführt
- der *reset* hat die Initialisierung der gesamten Hard- und Software zur Folge, insbesondere den Abbruch aller bodengesteuerten Aktivitäten
  - bereits aufgezeichnete wiss. Daten sind dann zwar gesichert, aber die noch anstehende Tagesarbeit kann nicht mehr vollbracht werden

Kategorie „feste Echtzeit“ (engl. *firm real-time*); zur Erinnerung (S. 2- 7):

fest (engl. *firm*) auch „stark“
- das Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist wertlos und wird verworfen
- Terminverletzung ist tolerierbar, führt zum Arbeitsabbruch

What really happened on Mars? (Forts.)
Fehlersituation

ASI/MET (niedrige Priorität) hat *bc_dist* (hohe Priorität) blockiert:
- *ASI/MET* belegte ein wiederverwendbares, unteilbares Betriebsmittel
  - das von *bc_dist* angefordert wurde, bevor *ASI/MET* es wieder frei gab
- im weiteren Verlauf verdrängten Tasks mittlerer Priorität *ASI/MET*
  - dadurch verlängerte sich die Blockierungszeit für *bc_dist*
  - als Folge war *bc_dist* noch nicht abgeschlossen als *bc_sched* startete
- *bc_sched* stellte die Zeitverletzung fest und löste einen *reset* aus

Fehlererkennung und -beseitigung:
- die Semaphorinitialisierung war in VxWorks falsch eingestellt
- sie wurde bodengesteuert (durch ein Skriptprogramm) korrigiert
  - der Semaphor wurde auf Prioritätsvererbung umgestellt

Resümee

Prioritäten gebunden an Ereignisauslöser oder -verarbeiter
- fest auf Task- und/oder Jobebene, ggf. dynamisch auf Taskebene
- Einplanungsaufwand: Auslöse- vs. Auswahlzeitpunkt von Jobs

Prioritätsabbildung mangels Systemprioritäten bzw. Priorätsebenen
- gleichmäßig oder ungleichmäßig, *constant ratio mapping*
- Einfluss auf die Planbarkeit (engl. *schedulability*) eines Systems

Repräsentation von Ablaufplänen $\mapsto$ Prioritätsschlange auf Tabellenbasis
- indizierte Bereitliste (*multi-level queue*), Aufwand begrenzen
- $O(1)$-*Scheduler*: Jobsauslösung vs. Jobauswahl

Prioritätsverletzung und Prioritätsumkehr Selbstaussetzung/Hemmung
- kritische Abschnitte, unteilbare Betriebsmittel, Fernaufrufe
- Abstraktion „*considered harmful*“, nichtfunktionale Eigenschaften