

# Systemprogrammierung

## Betriebssystemkonzepte: Prozesse

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

25. November 2010

## Programm in Ausführung

Kontrolliert durch Programme, exekuiert auf einen Prozessor

**Prozess**, kann die Ausführung mehrerer Programme bedeuten

- ein **Anwendungsprogramm** ruft ein **Betriebssystemprogramm** auf
  - Systemaufruf (engl. *system call*)
  - Programmunterbrechung (engl. *trap, interrupt*)
- ein Prozess ist **Aktivitätsträger** von ggf. mehreren Programmen
  - Adressraumüberlagerung mit einem anderem Programm (*exec(2)*)

**Programm**, kann von mehreren Prozessen ausgeführt werden

- **pseudo-paralleles Programm**, im Falle von Uniprozessorsystemen
  - präemptive (d.h. verdrängende) Programmverarbeitung
  - Aufgabe (engl. *task*), Faden (engl. *thread*)
- **paralleles Programm** im Falle von Multi(kern)prozessorsystemen

## Gliederung

- 1 **Ausführungsstrang**
  - Prozess
  - Prozessmodelle
  - Einplanung
  - Systemfunktionen
- 2 **Koordinationsmittel**
  - Konkurrenz
  - Koordinationsvariable
  - Systemfunktionen
- 3 **Kommunikationsmittel**
  - Botschaftenaustausch
  - Systemfunktionen
- 4 **Zusammenfassung**

## Prozess $\neq$ Programm

Programm ist statisch, Prozess ist dynamisch

Wissen über das gegenwärtig ausgeführte Programm sagt nicht viel aus über die zu dem Zeitpunkt im System stattfindende Aktivität

- Welches Zugriffsrecht besitzt das Programm zur Zeit?
  - auf ein Adressraumsegment, auf eine Datei, auf ein Gerät, ...
  - allgemein: auf ein Betriebsmittel
- Welcher Kontrollfluss ist im mehrfädigen Programm zur Zeit aktiv?
  - Uni- vs. Multiprozessorsystem (SMP)
- Wieviel Programmunterbrechungen sind zur Zeit gestapelt?
- 
- 

*Im Betriebssystemkontext ist das Konzept „Prozess“ daher nützlicher als das Konzept „Programm“, um Abläufe zu beschreiben und zu verwalten.*

## Prozess ≠ Prozessinstanz

Analogie zu Typ oder Klasse einerseits und Exemplar bzw. Objekt andererseits

### Prozess, ein abstraktes Gebilde

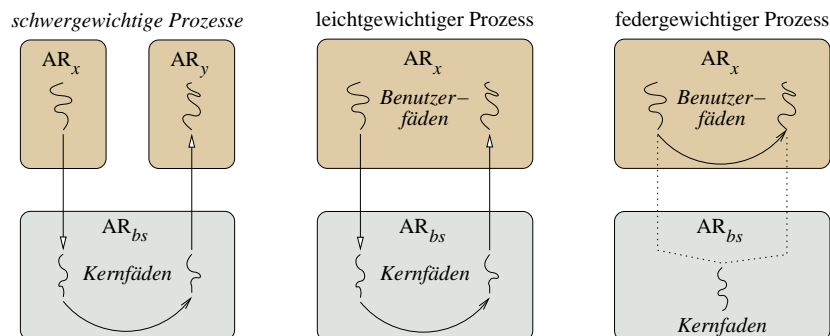
- ein „Programm in Ausführung“ ☺, sequentieller Kontrollfluss ☺
- ein „Ablauf“ ☺, der eine Verwaltungseinheit ist ☺

### Prozessinkarnation, ein konkretes Gebilde

- das „physische Exemplar“ des abstrakten Gebildes „Prozess“
  - an Betriebsmittel (Ressource; engl. *resource*) gebunden
  - die **Identität** (engl. *identity*) einer **Programmausführung**
- die einen Prozess repräsentierende **Verwaltungseinheit**
  - „dynamische Datenstruktur“ verschiedenartiger Strukturelemente

- synonyme Verwendung der Begriffe kann zu Missverständnissen führen

## Schwer- vs. leicht- vs. federgewichtige Prozesse



Adressraumwechsel sind (je nach MMU) mehr oder weniger „teuer“

- die zur Adressumsetzung benötigten Deskriptoren werden mit jedem Wechselsvorgang aus dem Zwischenspeicher (engl. *cache*) verdrängt
- erneute Adressraumaktivierung hat zur Folge, dass die MMU die Adressraumdeskriptoren erst wieder zwischenspeichern muss

## Gewichtsklassen von Prozessinkarnationen

### schwergewichtiger Prozess (engl. *heavyweight process*)

- Prozessinkarnation und Benutzeradressraum bilden eine Einheit
- Prozesswechsel  $\rightsquigarrow$  zwei Adressraumwechsel:  $AR_x \Rightarrow BS \Rightarrow AR_y$ 
  - „klassischer“ UNIX Prozess

### leichtgewichtiger Prozess (engl. *lightweight process*)

- Prozessinkarnation und Adressraum sind voneinander entkoppelt
- Prozesswechsel  $\rightsquigarrow$  einen Adressraumwechsel:  $AR_x \Rightarrow BS \Rightarrow AR_x$ 
  - **Kernfaden** (engl. *kernel thread*): Faden auf Kernebene

### federgewichtiger Prozess (engl. *featherweight process*)

- Prozessinkarnationen und Adressraum bilden eine Einheit
- Prozesswechsel  $\rightsquigarrow$  kein Adressraumwechsel:  $AR_x \Rightarrow AR_x$ 
  - **Benutzerfaden** (engl. *user thread*): Faden auf Benutzerebene

- Kern-/Benutzerfaden  $\Rightarrow$  Betriebssystem-/Benutzerprogramm

## Benutzthierarchie von Prozessinkarnationen

Implementierung von Prozessen

### schwergewichtiger Prozess



**Basis:** federgewichtiger Prozess

- der eigentliche **Kontrollfluss**
- Steuerbefehle sind Prozeduren des laufenden Programms
  - erzeugen, wechseln, zerstören

**Erweiterungen** zum Mehrprogrammbetrieb bedeuten „Gewichtszunahme“

- leichtgewichtiger Prozess: **vertikale Isolation** vom Betriebssystem
  - Steuerbefehle sind Systemaufrufe an den Betriebssystemkern
- schwergewichtiger Prozess: **horizontale Isolation** von anderen Fäden
  - jeder Faden besitzt seinen eigenen (logischen/virtuellen) Adressraum

**Implementierungskonzept** von Prozess(inkarnation)en ist die **Koroutine** [2]

- in mehr oder weniger stark funktional angereicherter Form

**Prozesseinplanung** (engl. *process scheduling*) stellt sich allgemein zwei grundsätzlichen Fragestellungen:

- 1 Zu welchem **Zeitpunkt** sollen Prozesse in das Rechensystem eingespeist werden?
- 2 In welcher **Reihenfolge** sollen eingespeiste Prozesse ablaufen?

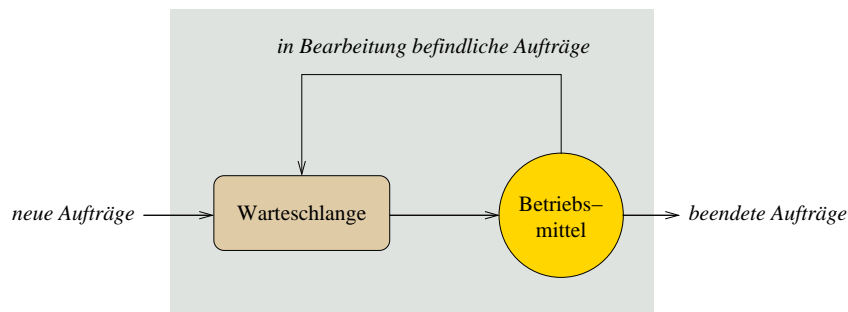
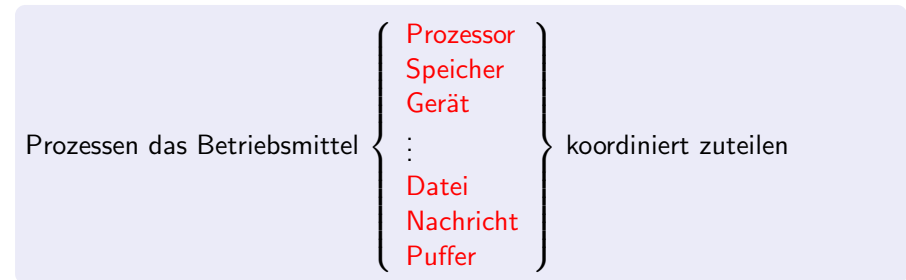
Zuteilung von **Betriebsmitteln** an **konkurrierende Prozesse** kontrollieren

### Einplanungsalgorithmus (engl. *scheduling algorithm*)

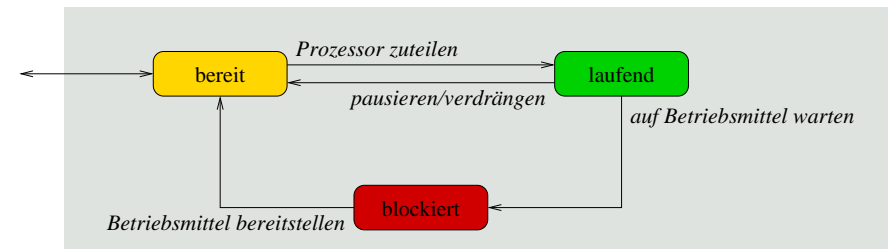
Implementiert die **Strategie**, nach der ein von einem Rechnersystem zu leistender **Ablaufplan** zur Erfüllung der jeweiligen **Anwendungsanforderungen** entsprechend aufzustellen und zu aktualisieren ist.

**Ablaufplan** (engl. *schedule*) zur Betriebsmittelzuteilung erstellen

- geordnet nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, ...
- entsprechend der jeweiligen Einplanungsstrategie
- zur Unterstützung einer bestimmten Rechnerbetriebsart



Ein einzelner Einplanungsalgorithmus charakterisiert sich durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereicht werden. [8]



Prozessverarbeitung impliziert die Verwaltung mehrerer **Warteschlangen**:

- häufig sind Betriebsmitteln eigene Warteschlangen zugeordnet
  - in denen Prozesse auf Zuteilung des jew. Betriebsmittels warten
- im Regelfall sind in Warteschlangen stehende Prozesse blockiert...
  - mit Ausnahme der **Bereitliste** (engl. *ready list*)
  - die auf Zuteilung der CPU wartenden Prozesse sind lauffähig

## Warteschlangentheorie

Theoretische Grundlagen des Scheduling

Betriebssysteme durch die „theoretische/mathematische Brille“ gesehen:

- R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
- E. G. Coffman, P. J. Denning. *Operating System Theory*.
- L. Kleinrock. *Queuing Systems, Volume I: Theory*.

Einplanungsverfahren stehen und fallen mit Vorgaben der **Zieldomäne**

- die „Eier-legende Wollmilchsau“ kann es nicht geben
- Kompromisslösungen sind geläufig
  - aber nicht in allen Fällen tragfähig

- Scheduling ist ein **Querschnittsbelang** (engl. *cross-cutting concern*)

## UNIX Systemfunktionen

Operationen auf Prozesse und Prozessadressräume

Linux, MacOS, SunOS

(vgl. S. 41)

```
pid = fork()
pid = wait(status)
void _exit(status)
pid = getpid()
pid = getppid()
ok = nice(incr)
err = execv(path, argv)
err = execve(path, argv, envp)
:
```

## UNIX Scheduling

Charakteristische Eigenschaften — Ausnahmen bestätigen die Regel

Linux, MacOS, SunOS

- die Verfahren wirken **verdrängend** (engl. *preemptive*)
  - Prozesse können das Betriebsmittel „CPU“ nicht monopolisieren
  - dem laufenden Prozess kann die CPU entzogen werden (CPU-Schutz)
- der fortgeschriebene Ablaufplan ist **nicht-deterministisch**
  - nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
  - die exakte Vorhersage der Prozessorauslastung ist nicht möglich
- Prozessausführung und -einplanung sind **gekoppelt** (engl. *online*)
  - dynamische Prozesseinplanung während der Programmausführung
  - Planungsziel: Antwortzeiten minimieren, Interaktivität fördern
- das System arbeitet im **Zeitmultiplexbetrieb** (engl. *time sharing*)

## Gliederung

- 1 Ausführungsstrang
  - Prozess
  - Prozessmodelle
  - Einplanung
  - Systemfunktionen
- 2 Koordinationsmittel
  - Konkurrenz
  - Koordinationsvariable
  - Systemfunktionen
- 3 Kommunikationsmittel
  - Botschaftenaustausch
  - Systemfunktionen
- 4 Zusammenfassung

Interaktion von Prozessen ist zwingend, um in einem Mehrprozesssystem Fortschritte in der Programmverarbeitung zu erreichen, und zwar:

**implizit** innerhalb des Betriebssystems

- (pseudo-) parallele Ausführung mehrfädiger Systemprogramme:
  - asynchrone Programmunterbrechungen
  - verdrängende Prozesseinplanung
  - ggf. auch SMP (engl. *symmetric multiprocessing*)
- die Prozesse **konkurrieren** um die Betriebsmittelzuteilung

**explizit** innerhalb des Anwendungssystems

- arbeitsteilige Ausführung eines Programms durch mehrere Fäden
  - paralleles/verteiltes Programm
- die Prozesse **kooperieren** zur gemeinsamen Programmausführung

```
typedef struct chainlink {
    struct chainlink *link;
} chainlink;

void chain (chainlink **next, chainlink *item) {
    *next = (*next)->link = item;
}
```

Anweisung zum Anhängen eines Kettenglieds in zwei Schritten:

- 1 Element einfügen: `temp = (*next)->link = item`
- 2 nächste Einfügestelle vermerken: `*next = temp`

**Annahme: Verdrängung dazwischen und Mitbenutzung**

- Verkettungsglied `*next` wird zur kritischen Variablen

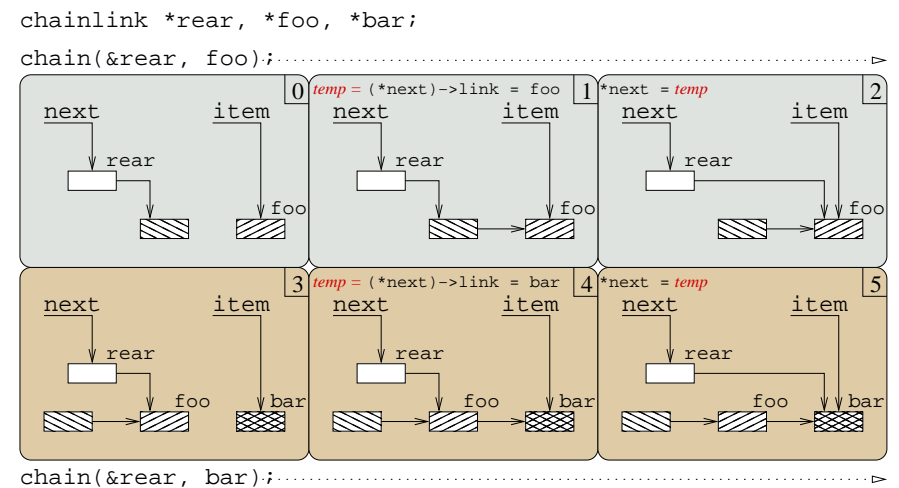
Rückblick: **einander überlappendes Zählen**

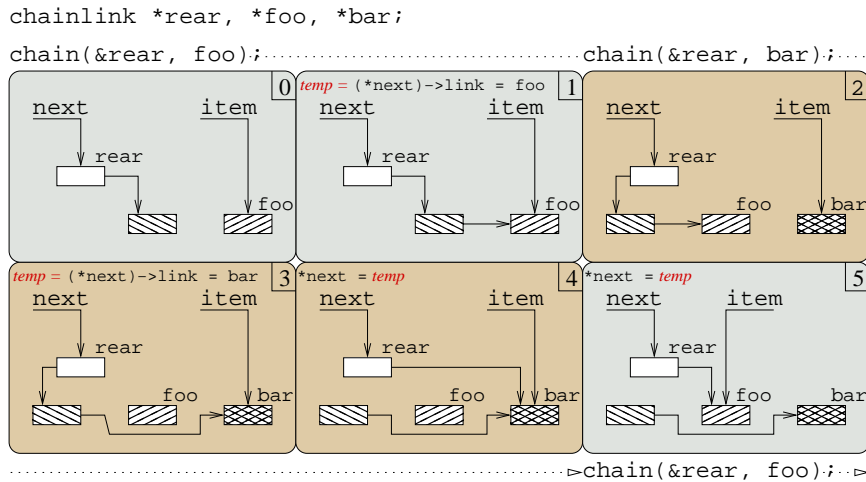
- **wheel++** ist nicht immer eine unteilbare Operation
  - diese Operation der Ebene<sub>5</sub> ist nur „scheinbar elementar“
  - ggf. bildet sie eine Sequenz von Elementaroperationen der Ebene<sub>4</sub>
  - in dem Fall wäre sie eine teilbare Operation und damit kritisch
- unterbrechungsbedingte Überlappungs(d)effekte möglich

**Warteschlangen**, z.B., stellen andere potentielle „Brennpunkte“ dar

- oft ist eine beliebige Permutation der **Zugriffsoperationen** möglich
  - eintragen überlappt austragen bzw. sich selbst, und umgekehrt
- auch die Auslegung der **Datenstruktur** „Schlange“ ist von Bedeutung

• „Untiefen“ dieser Art gibt es einige in Betriebssystemen...





**Synchronisation** (engl. *synchronization*) bringt die Aktivitäten von verschiedenen Prozessen in eine Reihenfolge [5, S. 26]:

- dadurch wird prozessübergreifend das erreicht, wofür innerhalb eines Ausführungsstrangs die Sequentialität von Aktivitäten sorgt
- Nebenläufigkeit bzw. Parallelität wird damit gezielt unterbunden

**Gegenseitiger Ausschluss der Listenmanipulation<sup>a</sup>**

<sup>a</sup>P() und V() klammern den kritischen Abschnitt. Durch P() wird erreicht, dass die Anweisungen bis zum V() nicht zugleich von mehreren Prozessen ausführbar sind.

```

void chain (chainlink **next, chainlink *item) {
    P();
    *next = (*next)->link = item;
    V();
}

```

Eine „nicht-negative ganze Zahl“, für die zwei **unteilbare Operationen** definiert sind [4]:

- P** (hol. *prolaag*, „erniedrige“; auch *down, wait*)
  - hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
  - ansonsten wird der Semaphor um 1 dekrementiert
- V** (hol. *verhoog*, erhöhe; auch *up, signal*)
  - inkrementiert den Semaphor um 1
  - auf den Semaphor ggf. blockierte Prozesse werden deblockiert

Ein **abstrakter Datentyp** zur **Signalisierung von Ereignissen** zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet).

**Linux, MacOS, SunOS** (vgl. S. 43)

```

id = semget(key, nsem, flag)
val = semctl(id, semnum, cmd, ...)
ok = semop(id, sembuf, nops)
:

```

**Sequenzen** von Semaphoroperationen werden unteilbar ausgeführt:

- technisch ist die Sequenz als ein `sembuf`-Feld repräsentiert, wobei die Reihenfolge der Feldelemente die Operationsreihenfolge definiert
- jedes `sembuf`-Exemplar beschreibt eine (ggf. andere) auszuführende Operation
- Bitschalter modifizieren das Verhalten der spezifizierten `sembuf`-Operation

```

struct sembuf {
    u_short sem_num;
    short sem_op;
    short sem_flg;
};

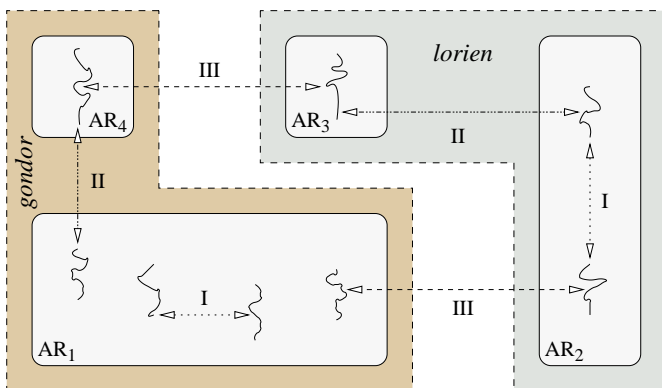
```

## Gliederung

- 1 Ausführungsstrang
  - Prozess
  - Prozessmodelle
  - Einplanung
  - Systemfunktionen
- 2 Koordinationsmittel
  - Konkurrenz
  - Koordinationsvariable
  - Systemfunktionen
- 3 Kommunikationsmittel
  - Botschaftenaustausch
  - Systemfunktionen
- 4 Zusammenfassung

## Problemdomänen der Kommunikation

Notwendigkeit domänenspezifischer Kommunikationsmechanismen



- i innerhalb desselben Adressraums
- ii zwischen verschiedenen Adressräumen desselben Rechner
- iii zwischen verschiedenen Rechnern

## Motivation

- Konsequenz der **physikalischen Adressraumtrennung** durch eine MMU:
- in Ausführung befindliche Programme sind abgeschottet
    - Prozesse sind in (log./virt.) Adressräumen eingesperrte „Gefangene“
    - sie können nicht ohne weiteres mit der „Außenwelt“ kommunizieren
  - Kooperation muss **Adressraumgrenzen** überwinden können
- Konsequenz **mehrerer Ausführungskontexte** innerhalb eines Adressraums:
- Programme laufen ggf. mehrfädig (engl. *multi-threaded*) ab
    - Fäden (engl. *threads*) sind eigene Kontrollflüsse im Programm
    - sie können nicht ohne weiteres mit anderen Fäden kommunizieren
  - Kooperation muss **Kontrollflussgrenzen** überwinden können

### Semaphor ~ Zeitsignal

Ein Semaphor eignet sich zur Anzeige des Ereignisses, dass Daten den einen Prozess verlassen haben und bei einem anderen Prozess eingetroffen sind. Den Datenaustausch selbst bewerkstelligt ein Semaphor nicht.

## Interprozesskommunikation

Prinzipielle Aktionen

### Datentransfer vom Sende- zum Empfangsadressraum

- über einen den Prozessen gemeinsamen Kommunikationskanal

### Synchronisation von Sende- und Empfangsprozess

- Fortschritt des Empfangsprozesses hängt ab vom Sendeprozess
  - die Nachricht ist ein **konsumierbares Betriebsmittel**
  - Empfangsprozess ist **Konsument**, Sendeprozess ist **Produzent**
  - konsumiert werden kann nur, nachdem produziert worden ist
- Fortschritt des Sendeprozesses hängt ab vom Empfangsprozess
  - der Nachrichtenpuffer ist ein **wiederverwendbares Betriebsmittel**
  - Sendeprozess füllt, Empfangsprozess leert den Puffer
  - gefüllt werden kann nur, wenn noch Platz ist ( $\leftrightarrow$  leeren)
- die **Koordination** geschieht implizit mit der angewandten Primitive

## Kommunikationssemantiken [7]

**Sendep primitiven** wirken unterschiedlich auf den ausführenden Prozess, je nach **Grad der Synchronisation** mit dem Empfangsprozess:

*no-wait send* Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist

- Interprozesskommunikation **im Vorübergehen** (durch Pufferung)

*synchronization send* Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist

- **Rendezvous** zwischen Sender und Empfänger (ohne Pufferung)

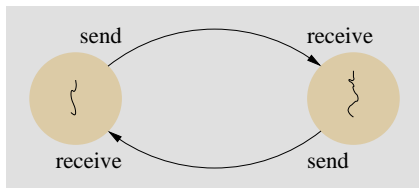
*remote-invocation send* Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist

- **Fernaufruf** einer vom Empfangsprozess auszuführenden Funktion

**Empfangsprimitiven** wirken (im Regelfall) gleich auf den ausführenden Prozess: er wartet, bis eine Nachricht von einem Sendeprozess eintrifft.

## Gleichberechtigte Kommunikation

*no-wait send* oder *synchronization send*



Die an der Kommunikation beteiligten Prozesse sind in ihrer Rollenfunktion gleichzeitig Produzent und Konsument von Nachrichten.

**send** Bereitstellung eines konsumierbaren Betriebsmittels

- in* Identifikation des Empfängers (Konsument)
- in* Basis/Länge der Nachricht

**receive** Anforderung eines konsumierbaren Betriebsmittels

- in* Basis/Länge eines Empfangspuffers
- out* Identifikation des Senders (Produzent)

## Kommunikationsmodelle

### Gleichberechtigte Kommunikation

Die miteinander kommunizierenden Prozesse spielen **dieselbe Rolle**; zwei Kommunikationspartner,  $P_1/P_2$ , sind sowohl Sender als auch Empfänger:

$$P_1 \left\{ \begin{array}{l} \text{send} \longrightarrow \text{receive} \\ \text{receive} \longleftarrow \text{send} \end{array} \right\} P_2$$

### Ungleichberechtigte Kommunikation

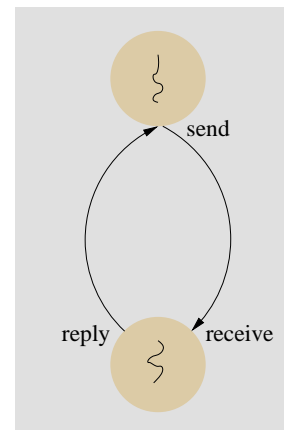
Die miteinander kommunizierenden Prozesse spielen **verschiedene Rollen**; ein Kommunikationspartner,  $P_2$ , ist **Dienstgeber** (engl. *server*), der andere,  $P_1$ , ist **Dienstnehmer** (engl. *client*):

$$(\text{Klient}) P_1 \left\{ \begin{array}{l} \text{send} \longrightarrow \text{receive} \\ \text{reply} \longleftarrow \end{array} \right\} P_2 (\text{Anbieter})$$

## Ungleichberechtigte Kommunikation

*remote-invocation send*

Die an der Kommunikation beteiligten Prozesse besitzen unterschiedliche Rollenfunktionen, **Klient** einerseits und **Anbieter** andererseits:



**send** einer Anforderungsnachricht

- in* Identifikation des Anbieters
- in* Basis/Länge der Nachricht
- in* Basis/Länge des Empfangspuffers
- out* Identifikation eines Anbieters

**receive** einer Anforderungsnachricht

- in* Basis/Länge des Empfangspuffers
- out* Identifikation des Klienten

**reply** einer Antwortnachricht

- in* Identifikation des Klienten
- in* Basis/Länge der Nachricht



- Faden** (engl. *thread*) **Konsument** der Nachricht
- direkte Adresse der die Nachricht verarbeitenden Instanz
  - die Prozessidentifikation (PID)
- Tor** (engl. *port*) **Anschluss zur Weiterleitung/Zustellung** von Nachrichten, der einem bestimmten Prozess zugeordnet ist
- Prozesse können mehrere solcher Anschlüsse besitzen
    - Ein- und/oder Ausgangstore für Nachrichten
  - die Zuordnung ist statisch oder dynamisch
- Briefkasten** (engl. *mailbox*) **Zwischenspeicher** für Nachrichten, der durch Senden gefüllt und Empfangen geleert wird
- der Pufferbereich ist keinem Prozess zugeordnet
  - *N* Prozesse können dahin senden und daraus empfangen

- IPC nutzt (in dem Fall) **Torverbindungen** und verläuft in drei Phasen:
- Aufbauphase** plant die zur Durchsetzung der jeweils angeforderten Gütemerkmale notwendigen Betriebsmittel ein
- Puffer, Fäden, Bandbreite, . . . , Protokoll
- Nutzungsphase** Botschaftenaustausch gemäß Gütemerkmale
- Abbauphase** gibt die reservierten (eingeplanten) Betriebsmittel frei und löst die Verbindung auf

Richtung/Betriebsart verbindungsorientierter Kommunikation

Richtung			Betriebsart
unidirektional	Tor <sub>s</sub>	→	Tor <sub>r</sub> halbduplex
bidirektional	Tor <sub>sr</sub>	↔	Tor <sub>rs</sub> voll duplex

Prozesse synchronisieren sich zur Kommunikation, indem sie Betriebsmittel anfordern und bereitstellen (S. 28):

- Sender** benötigt das wiederverwendbare Betriebsmittel „Puffer“
- synchrone IPC ⇒ der Zielpuffer (des Empfängers)
  - asynchrone IPC ⇒ ein Zwischenpuffer
- Empfänger** benötigt das konsumierbare Betriebsmittel „Nachricht“
- asynchrone IPC ⇒ ein Zwischenpuffer
  - synchrone IPC ⇒ der Quellpuffer (des Senders)

**Betriebsmittelmangel**

Ursache für die **Blockierung** der Prozesse bei der Kommunikation:

- Empfänger erwartet Nachricht, Sender erwartet freien Puffer
- „asynchron“ bedeutet nicht „nicht-blockierend“ oder „wartefrei“

```
Linux, MacOS, SunOS
s = socket(domain, type, protocol)
ok = bind(s, name, namelen)
num = recvfrom(s, buf, buflen, flags, from, fromlen)
num = sendto(s, msg, msglen, flags, to, tolen)
ok = connect(s, name, namelen)
ok = listen(s, backlog)
d = accept(s, addr, addrlen)
num = recv(d, buf, buflen, flags)
num = send(s, msg, msglen, flags)
ptr = gethostbyname(name)
:
:
```

## Gliederung

- 1 Ausführungsstrang
  - Prozess
  - Prozessmodelle
  - Einplanung
  - Systemfunktionen
- 2 Koordinationsmittel
  - Konkurrenz
  - Koordinationsvariable
  - Systemfunktionen
- 3 Kommunikationsmittel
  - Botschaftenaustausch
  - Systemfunktionen
- 4 Zusammenfassung

## Literaturverzeichnis

- [1] COFFMAN, E. G. ; DENNING, P. J.:  
*Operating System Theory*.  
Prentice Hall, Inc., 1973
- [2] CONWAY, M. E.:  
Design of a Separable Transition-Diagram Compiler.  
In: *Communications of the ACM* 6 (1963), Nr. 7, S. 396–408
- [3] CONWAY, R. W. ; MAXWELL, L. W. ; MILLNER, L. W.:  
*Theory of Scheduling*.  
Addison-Wesley, 1967
- [4] DIJKSTRA, E. W.:  
Cooperating Sequential Processes / Technische Universiteit Eindhoven.  
Eindhoven, The Netherlands, 1965. –  
Forschungsbericht. –  
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

## Resümee

- die **Prozessinkarnation** ist Exemplar eines Prozesses
  - sie lässt sich verschiedenen Gewichtsklassen zuordnen:
    - feder-, leicht- und schwergewichtige Prozesse
    - hinter denen sich unterschiedliche Prozessmodelle verbergen
- die **Einplanung** legt Zeitpunkt und Reihenfolge von Prozessen fest
  - Prozessinkarnationen werden Verarbeitungszustände zugeschrieben
    - Zustandsübergänge (in EBNF):  $\{\{\text{bereit, laufend}\} -, [\text{blockiert}]\}$
    - UNIX: verdrängend, nicht-deterministisch, gekoppelt, Zeitmultiplex
- gleichzeitige Prozesse ziehen **Interaktion** nach sich:
  - implizit bei einander unbekanntem Prozessen im Betriebssystem
  - explizit bei einander bekannten Prozessen im Maschinenprogramm
- in **Konkurrenz** sind Prozesse zu koordinieren bzw. synchronisieren
  - durch den Austausch von Zeitsignalen mittels Koordinationsvariable
  - aber ebenso durch Botschaftenaustausch: Interprozesskommunikation

## Literaturverzeichnis (Forts.)

- [5] HERTTWICH, R. G. ; HOMMEL, G. :  
*Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme*.  
Springer-Verlag, 1989. –  
ISBN 3–540–51701–4
- [6] KLEINROCK, L. :  
*Queuing Systems*. Bd. I: Theory.  
John Wiley & Sons, 1975
- [7] LISKOV, B. J. H.:  
Primitives for Distributed Computing.  
In: *Proceedings of the Seventh Symposium on Operating System Principles (SOSP 1979)*,  
December 10–12, 1979, Pacific Grove, California, USA, ACM, 1979. –  
ISBN 0–89791–009–5, S. 33–42
- [8] LISTER, A. M. ; EAGER, R. D.:  
*Fundamentals of Operating Systems*.  
The Macmillan Press Ltd., 1993. –  
ISBN 0–333–59848–2

## Parthenogenese in UNIX

### Prozess aufspalten, abwarten und beenden

(← Aufgabe 3)

```
#include <sys/types.h>
#include <sys/wait.h>

char parent[] = "Elter: ", child[] = "Kind: ";

main() {
    pid_t pid;
    int zwerg;

    switch ((pid = fork())) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            printf("%sHier ist der Kindprozess, meine PID ist %d.\n", child, getpid());
            printf("%sDie PID meines Elterprozesses ist %d.\n", child, getppid());
            printf("%sGib mir einen (kleinen Wert als) Exitstatus: ", child);
            scanf("%d", &zwerg);
            printf("%sDanke und Tschüss!\n", child);
            exit(zwerg);
        default:
            printf("%sHier ist ein Elterprozess, meine PID ist %d.\n", parent, getpid());
            printf("%sDie PID meines Kindprozesses ist %d..\n", parent, pid);
            wait(&zwerg);
            printf("%sDer Exitstatus meines Kindprozesses ist %d.\n", parent, WEXITSTATUS(zwerg));
            printf("%sHollaröhdulliöh!\n", parent);
    }
}
```

## Einrichten und initialisieren einer Koordinationsvariablen

```
#include <sys/sem.h>

typedef struct {
    int sid; /* Semaphordeskriptor/-kennung */
    struct sembuf sop; /* Deskriptor der Semaphoreoperation */
} sema_t;

sema_t sema; /* Semaphorexemplar */

int main () {
    ...
    sema.sid = semget(IPC_PRIVATE, 1, IPC_CREAT);
    if (sema.sid == -1) perror("semget");
    else if (semctl(sema.sid, 0, SETVAL, 1) == -1)
        perror("semctl");
    ...
}
```

## Ablaufprotokoll der Interaktion Elter ↔ Kind

```
wosch@gondor 71$ gcc -O6 -o fork fork.c
wosch@gondor 72$ ./fork
Elter: Hier ist ein Elterprozess, meine PID ist 1984.
Elter: Die PID meines Kindprozesses ist 1985...
Kind: Hier ist der Kindprozess, meine PID ist 1985.
Kind: Die PID meines Elterprozesses ist 1984.
Kind: Gib mir einen (kleinen Wert als) Exitstatus: 42
Kind: Danke und Tschüss!
Elter: Der Exitstatus meines Kindprozesses ist 42.
Elter: Hollaröhdulliöh!
wosch@gondor 73$
```

## Nachbildung von P und V

```
void prolaag (sema_t *s) {
    int ok;

    assert(s);

    s->sop.sem_op = -1;
    ok = semop(s->sid, &s->sop, 1);

    assert(ok != -1);
}

inline void P () { prolaag(&sema); }
```

```
void verhoog (sema_t *s) {
    int ok;

    assert(s);

    s->sop.sem_op = 1;
    ok = semop(s->sid, &s->sop, 1);

    assert(ok != -1);
}

inline void V () { verhoog(&sema); }
```

Scheitern von P/V sieht die klassische Definition [4] nicht vor:

- durch **Zusicherung** (engl. *assertion*) wird „Gelingen“ garantiert

### Achtung

- DNDEBUG bzw. #define NDEBUG stellen Zusicherungen ab