

2. Prozesse (Stand 05.06.2024)

Ein Prozess ist der Ablauf eines Programms in einer Rechenanlage

Programm + zeitliche Dimension (Zustand) = **Prozess**



statisch



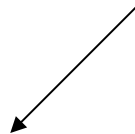
dynamisch

Aus Anwendersicht: Prozess ist Handlungsträger

Aus BS-Sicht: Prozess ist Objekt, dem der Prozessor zugeteilt wird

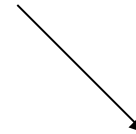
Alle modernen Betriebssysteme können zu einem Zeitpunkt mehrere Prozesse verwalten.

Mehrprozessbetrieb (multiprogramming, multitasking)



Multiprozessorsystem

echtes Multitasking

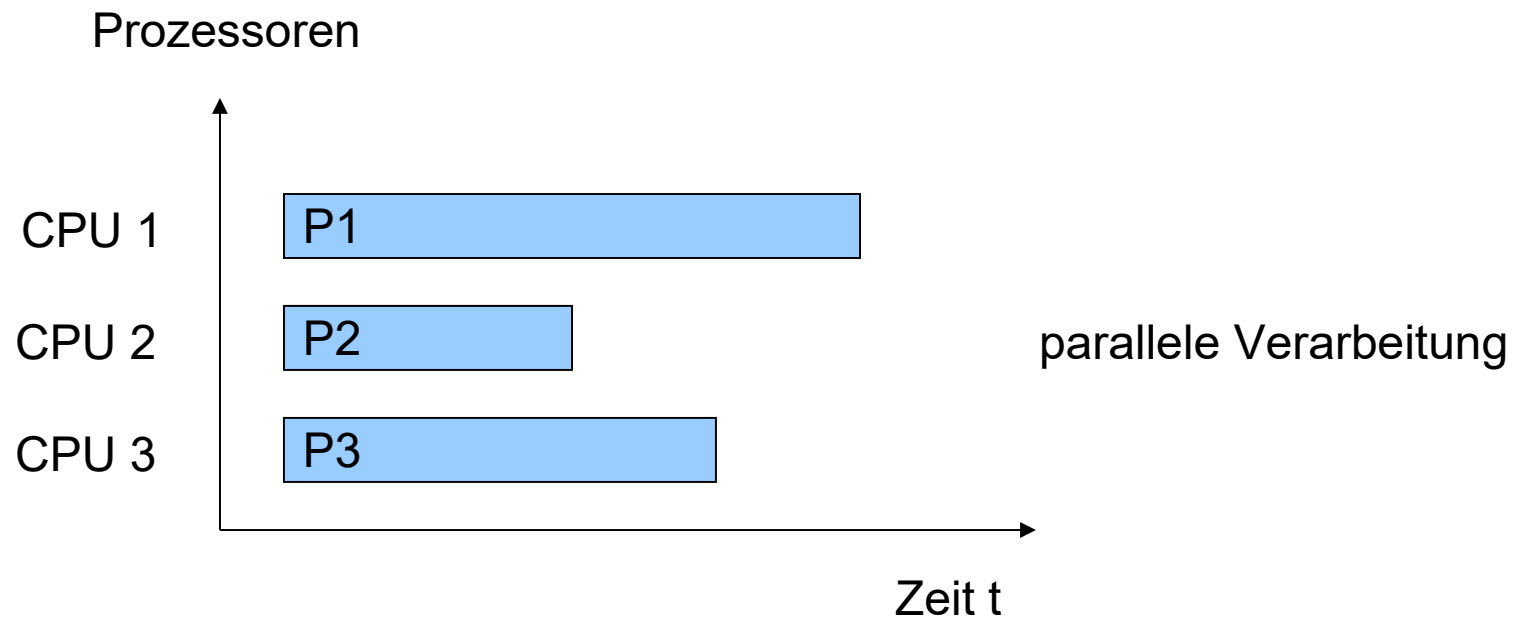


Einprozessorsystem

Prozesse müssen
sequentialisiert werden
(Nebenläufigkeit, concurrency)

Ursprüngliche Idee von Multitasking: bessere Prozessorauslastung

Heute: Komfort und Flexibilität



sequentielle Verarbeitung



überlappende Verarbeitung



Bei voneinander unabhängigen Prozessen hängen die Ergebnisse der Prozesse nicht von deren Bearbeitungsreihenfolge ab.

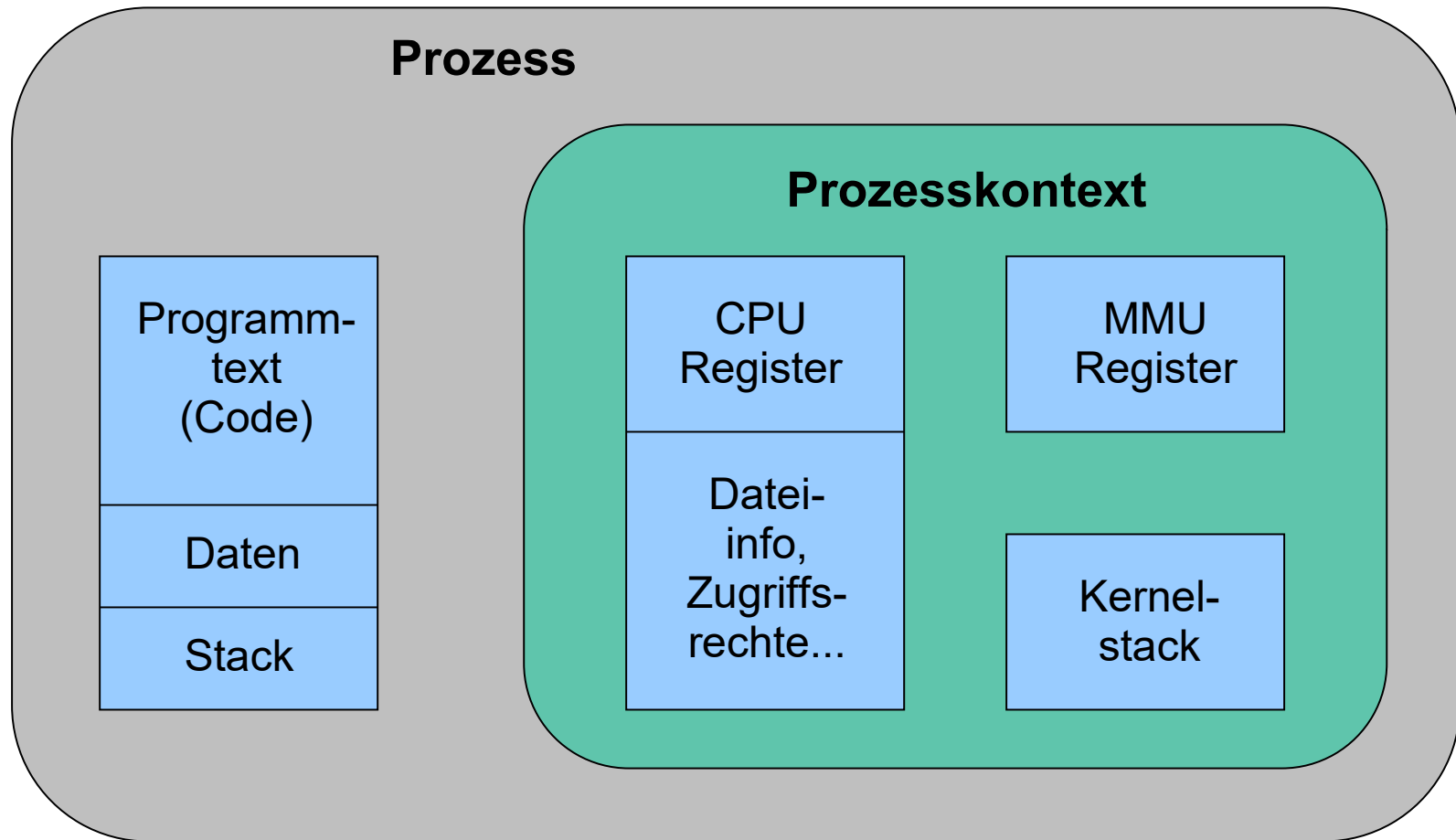
Multitasking spart auch Bearbeitungszeit:

Textverarbeitung + Ausdruck von Texten kann parallel erfolgen.

Bsp.: Windows 3.x + Word (lang ist es her)



Woran lag die schlechte Performanz?



Der Prozesskontext beschreibt einen virtuellen Prozessor

Der Prozesskontext ist in zwei Teile aufgeteilt:

1) **Speicherresidenter process control block** (PCB)

- Scheduling - Parameter
- Speicherreferenzen: Code-, Daten- und Stackadressen
(im Haupt- bzw. Massenspeicher)
- Signaldaten wie Masken
- erwartetes Ereignis
- Timerzustand
- PID, PID der Eltern, User/Group IDs

2) **Auslagerbarer Benutzerkontext** (swappable user structure)

- Prozessorzustand: Register, FPU-Register
- Systemaufruf: Parameter, bisherige Ergebnisse
- Dateiinfo - Tabelle (file descriptor table)
- Benutzungsinfo: CPU-Zeit, max. Stackgröße, ...
- Kernel - Stack: Stackplatz für Systemaufrufe des Prozesses

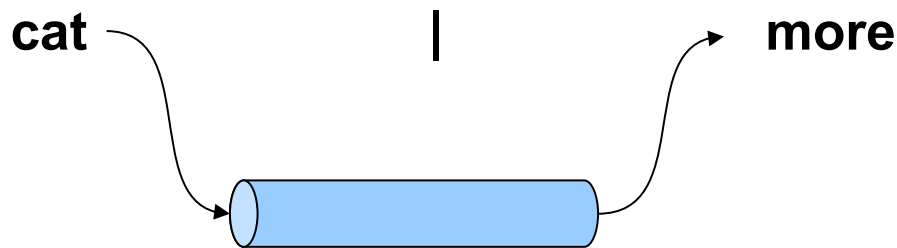
Der PCB lässt sich nur indirekt über Systemaufrufe abfragen und ändern.

Die user structure lässt sich direkt mit Unix-Befehlen bearbeiten.

Ein Programm (Job) kann mehrere Prozesse erzeugen.

\$ cat gedicht1 gedicht2 gedicht3 | more

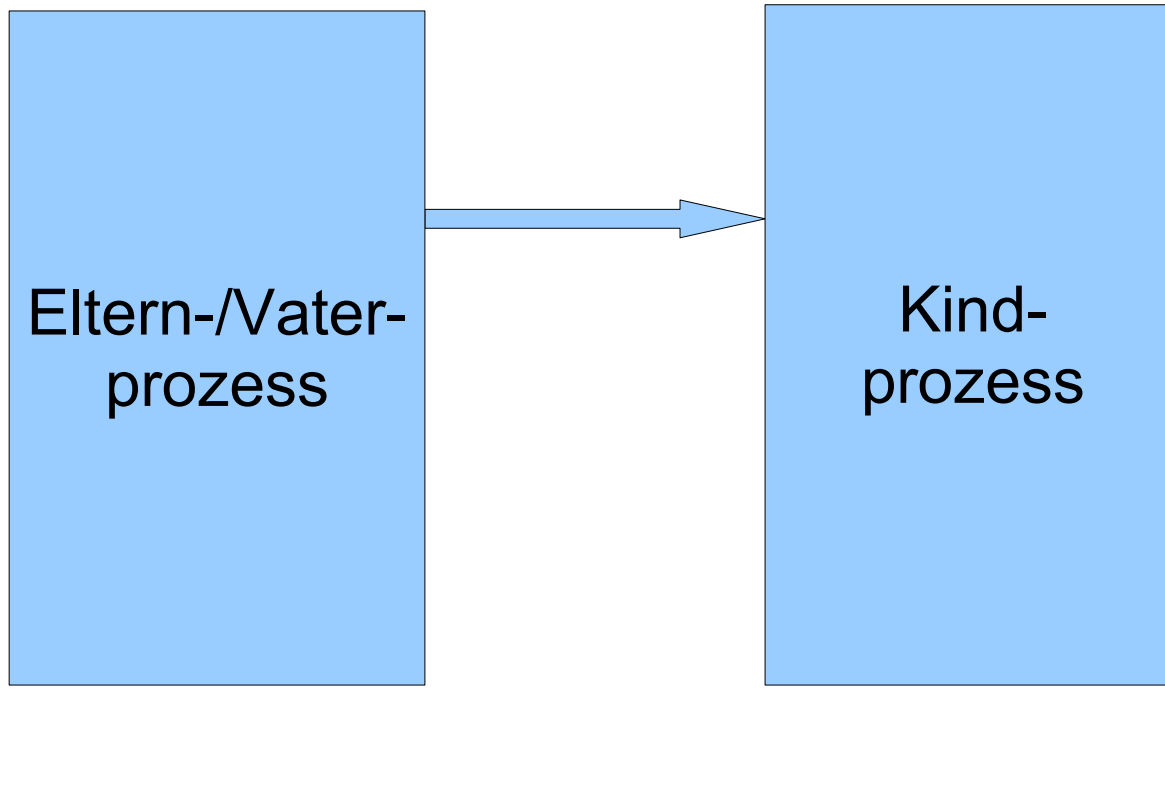
Shell startet 2 Prozesse (cat, more) und lenkt die Ausgabe des ersten zur Eingabe des zweiten um.



| ist Pipe-Zeichen

t

Ein Prozess kann einen anderen Prozess erzeugen.



Linux fork()

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main( ){
    pid_t child_pid;
    child_pid = fork (); // Create a new child process;
    if (child_pid < 0) {
        printf("fork failed");
        return 1;
    }
    else if (child_pid == 0) {
        printf ("child process successfully created!");
    }
    else {
        printf ("parent process still running!");
    }
    return 0;
}
```

Windows CreateProcessA

```
BOOL CreateProcessA(  
    [in, optional]      LPCSTR          lpApplicationName,  
    [in, out, optional] LPSTR           lpCommandLine,  
    [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    [in]                BOOL             bInheritHandles,  
    [in]                DWORD            dwCreationFlags,  
    [in, optional]      LPVOID           lpEnvironment,  
    [in, optional]      LPCSTR           lpCurrentDirectory,  
    [in]                LPSTARTUPINFOA   lpStartupInfo,  
    [out]               LPPROCESS_INFORMATION lpProcessInformation  
)
```

2.1 Prozesszustände

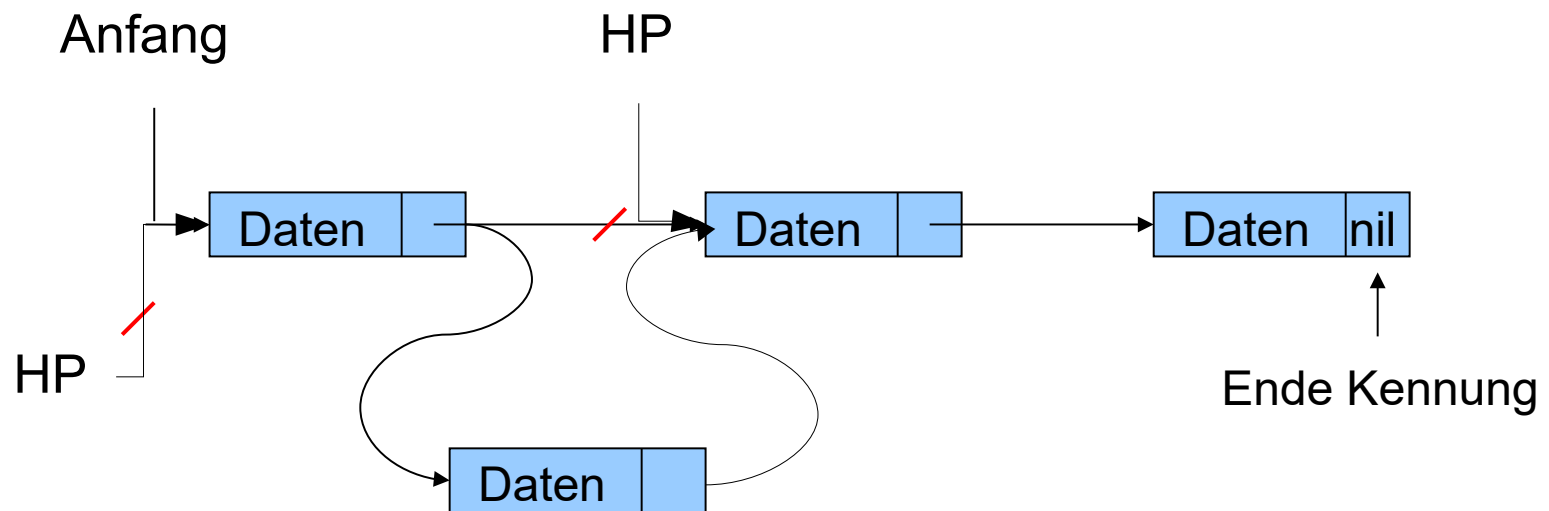
Im Monoprozessorrechner ist ein Prozess „aktiv“ (running), die anderen sind angehalten, blockiert, wartend.

worauf wird gewartet:

1. aktiv den Prozessor zu erhalten (sind sonst bereit (ready))
2. eine Nachricht (message) von einem anderen Prozess zu erhalten (siehe Prozesskommunikation später)
3. ein Signal von einem Zeitgeber (timer) zu erhalten (siehe Prozesssynchronisation später)
4. Daten eines Ein-/Ausgabegerätes zu erhalten (I/O)

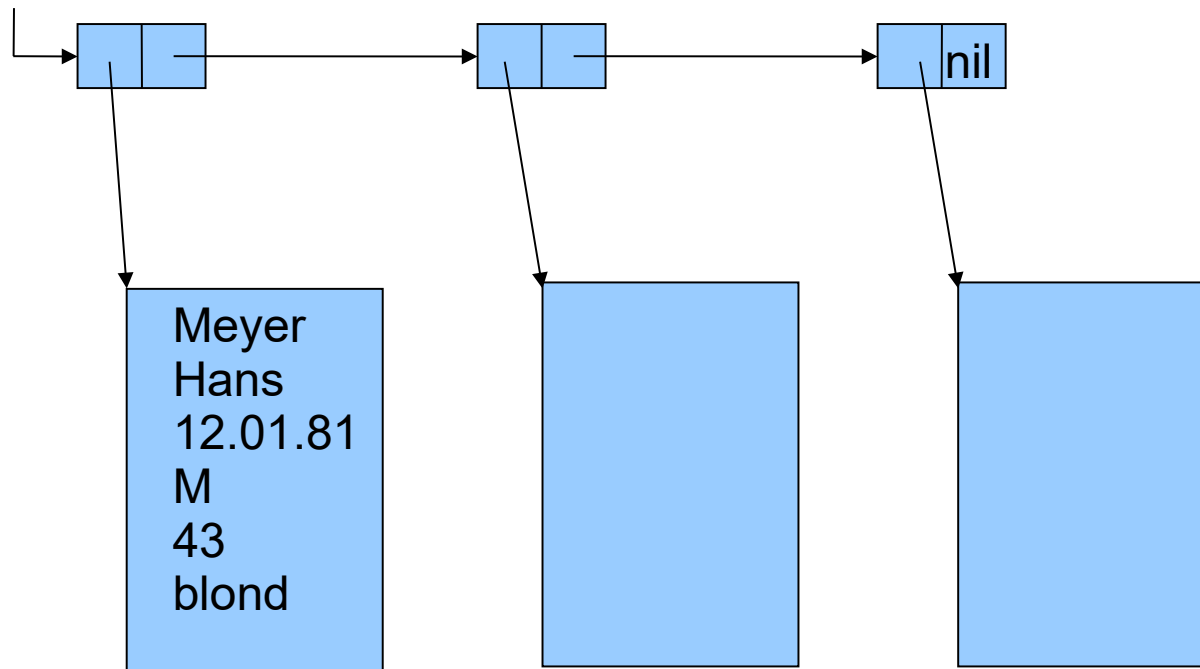
Falls durch ein Ereignis ein Prozess (2.-4.) entblockt wird, wird er in die Bereit - Liste (1. , ready-queue) verschoben. Nur aus der Bereit - Liste gehen Prozesse in den aktiven Zustand über.

Oft benötigte Datenstruktur: verkettete Liste

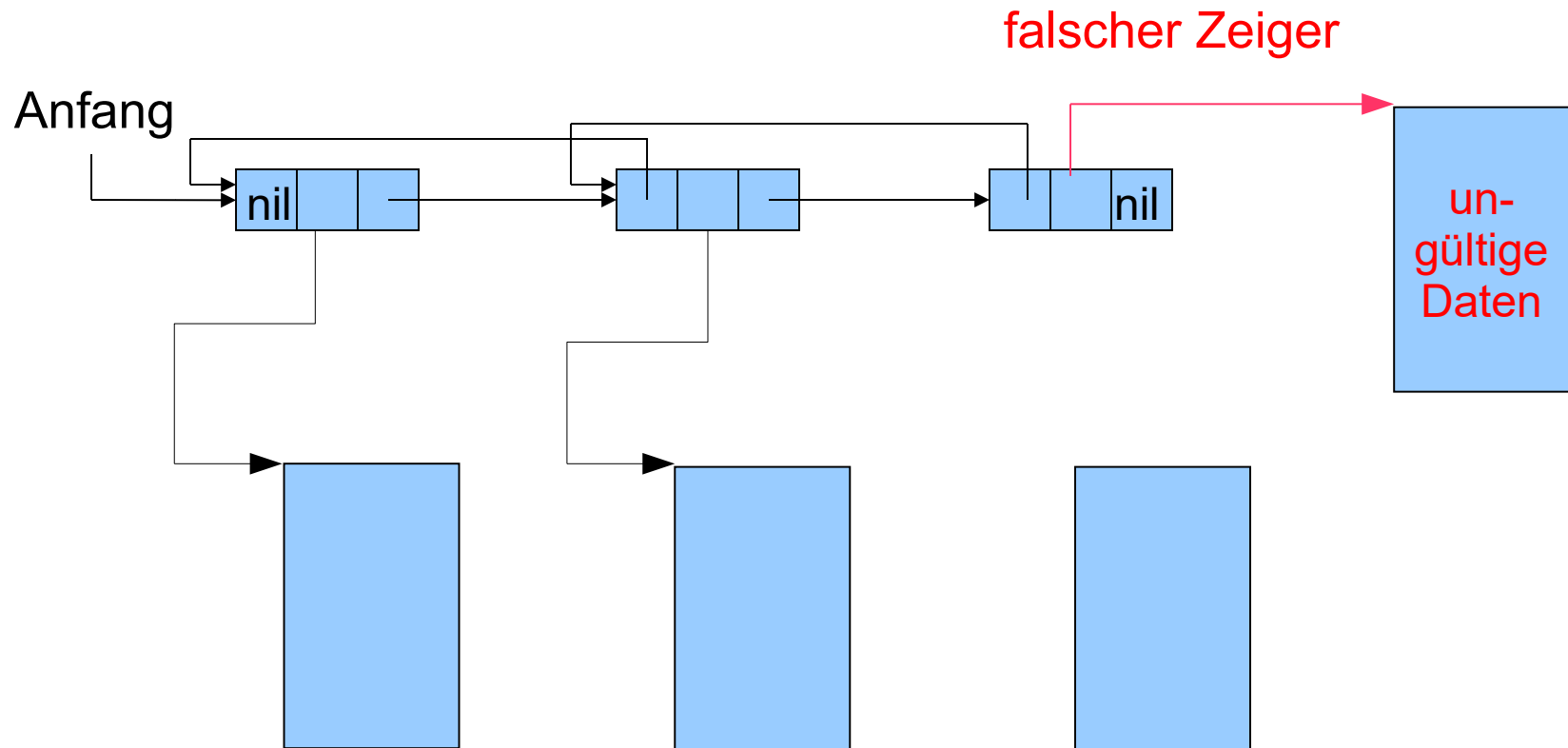


Daten über Zeiger

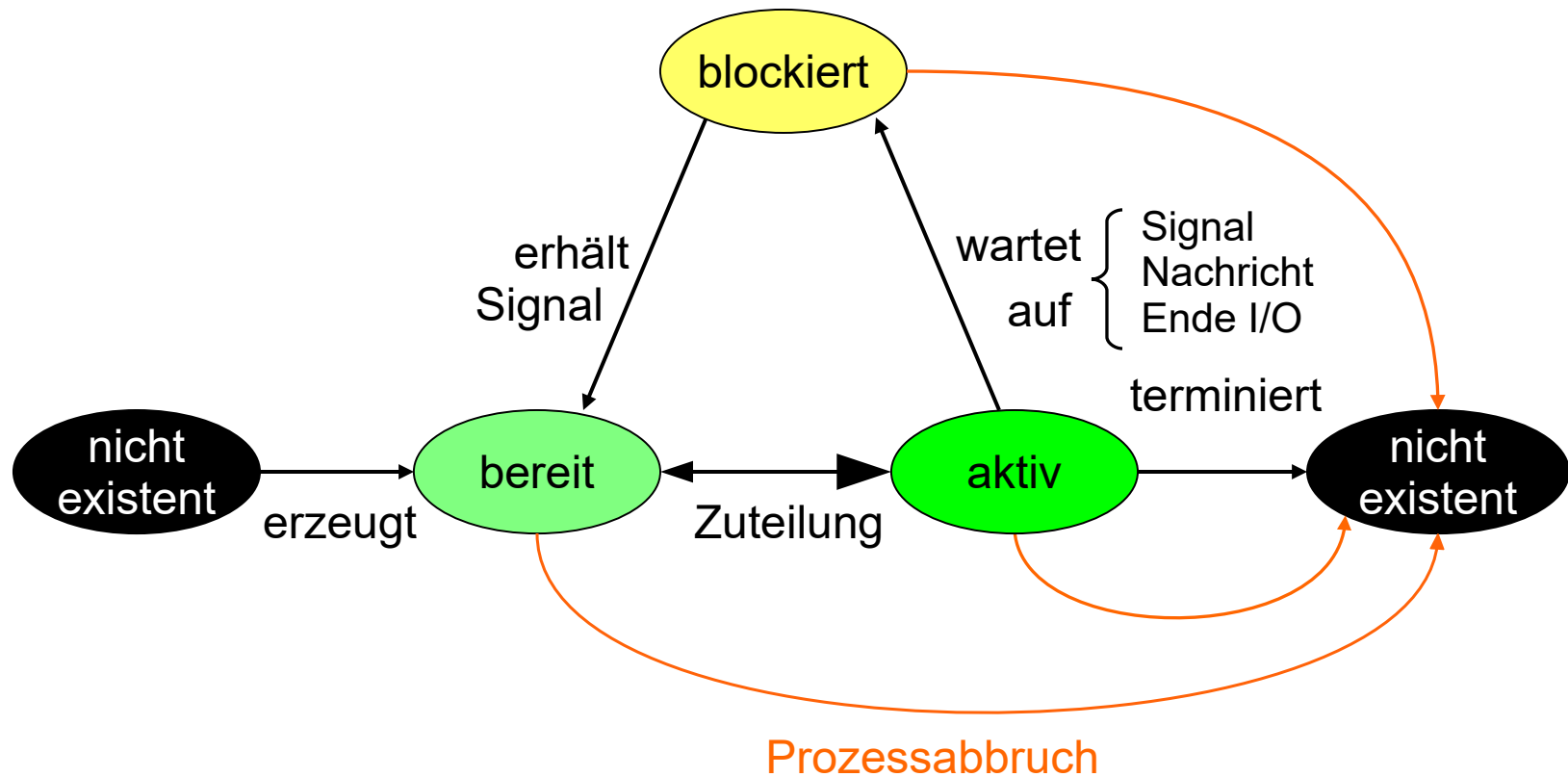
Anfang



Kleiner Fehler, große Auswirkung



2.1.1 Prozesszustände und Übergänge



Die Überführung von Zustand zu Zustand übernimmt eine zentrale Betriebssysteminstanz - der Scheduler. Aufgaben:

- Zustellen der Signale
- Abspeichern der Prozessdaten
- Einordnen in die Warteschlangen

Über Betriebssystemaufrufe meldet ein Prozess Wünsche beim Scheduler an, der diese mit anderen Prozesswünschen koordiniert.

Blockiert ein Prozess muss der letzte Zustand der CPU
(= virtueller Prozessor) gerettet werden.

Ein neuer virtueller Prozessor wird geladen.
(Prozess- / Kontextwechsel: **context switch**)

2.1.2 Betriebssysteme differieren

- Zahl der Ereignisse, auf die gewartet werden kann
- Anzahl und Typen von Warteschlangen, in denen gewartet werden kann
- Strategien zum Erzeugen und Terminieren von Prozessen

Begriffsklärung:

Scheduling



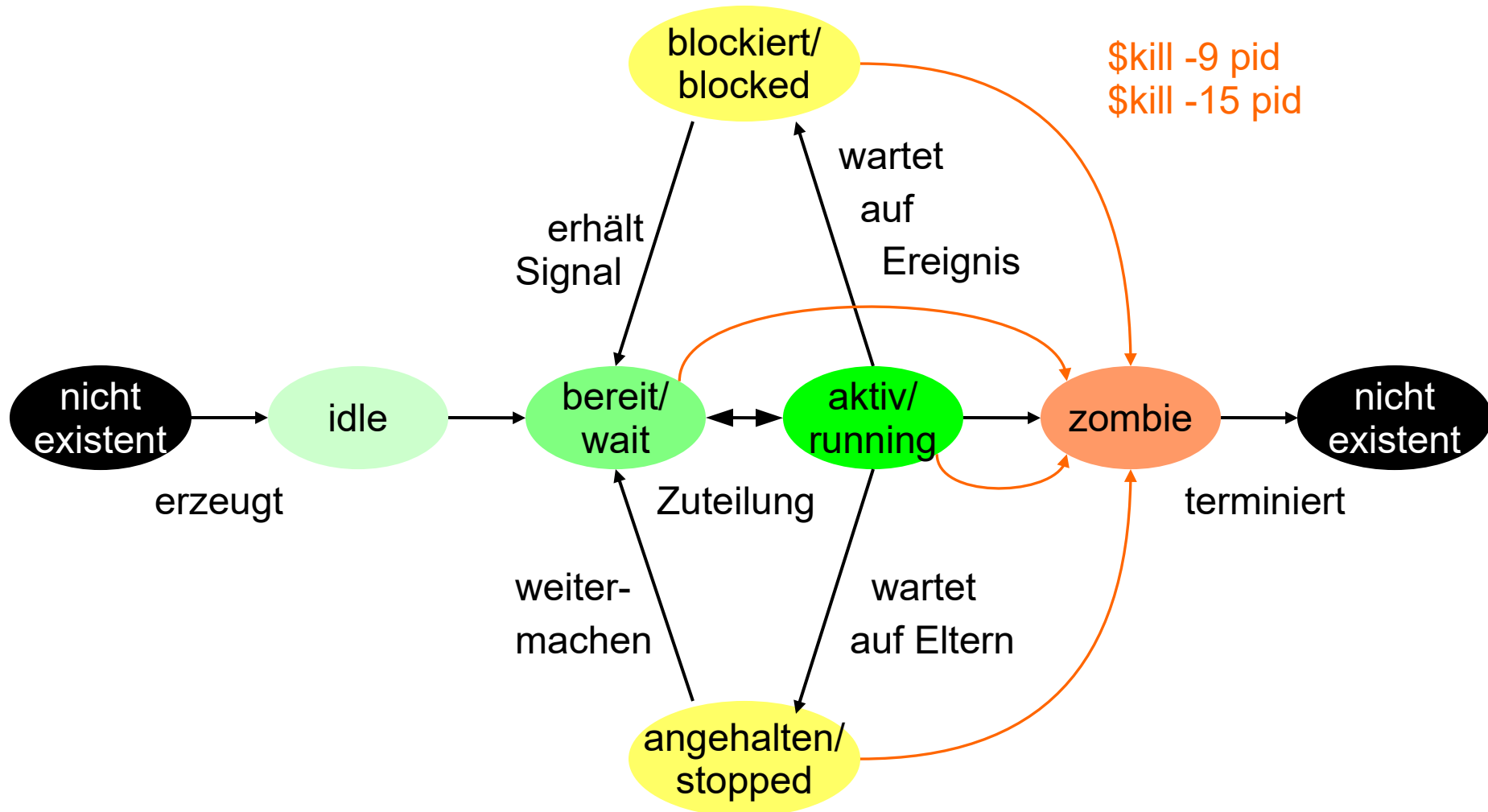
Planen der Zuteilung
der Betriebsmittel

Dispatching



tatsächliches
Zuteilen

2.1.3 Beispiel Unix



Den Übergang von einem Zustand in den nächsten erreicht ein Prozess in der Regel durch Anfragen (Systemaufrufe).

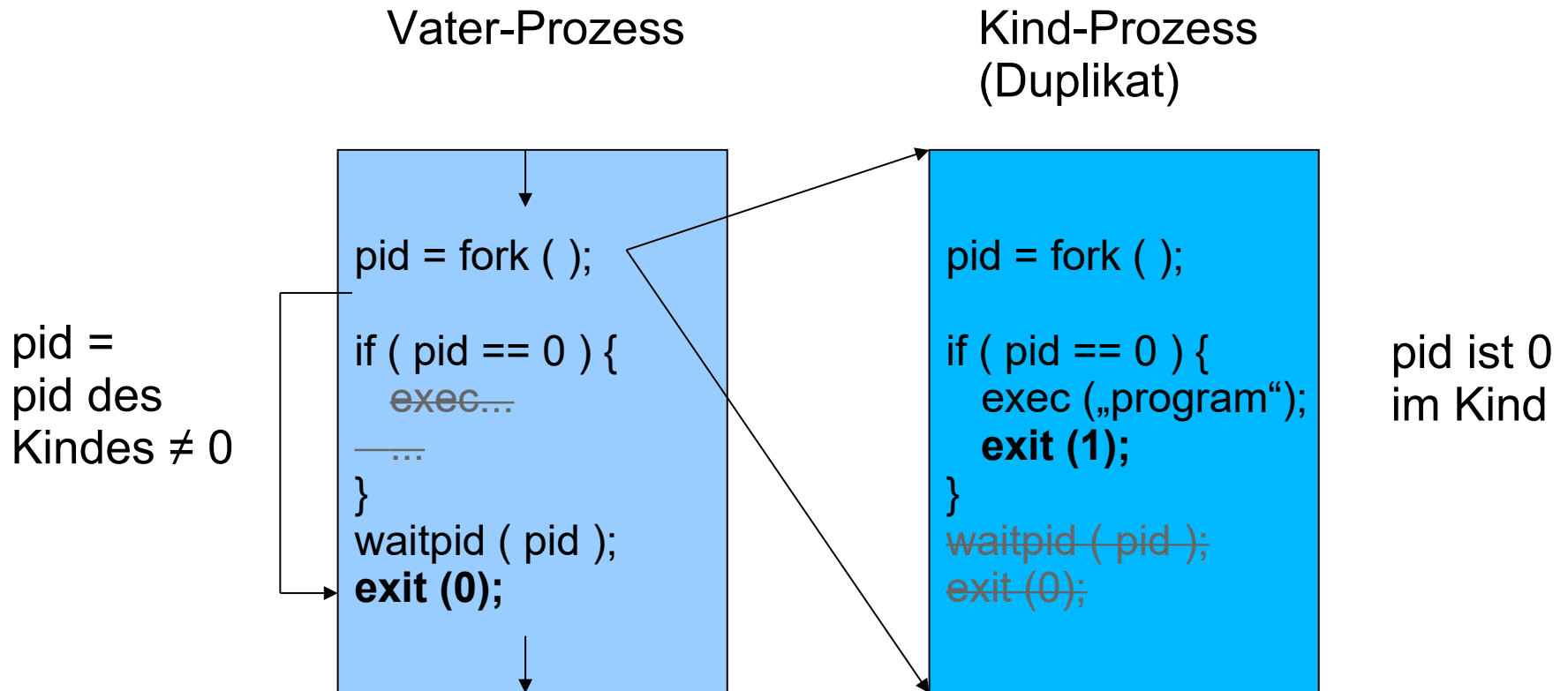
Prozess erzeugen:

Mit der Systemfunktion **fork()** wird vom aufrufenden Prozess eine Kopie erstellt, die in die Bereit-Liste eingehängt wird.

Alle Prozesse in Unix stammen direkt oder indirekt vom Init-Prozess (pid ist 1) ab.

Prozess beenden:

Ein Prozess beendet sich selbst durch die Systemfunktion **exit()**. Er teilt dem System seinen Exit-Status mit, den der Prozess-erzeuger (sein Vaterprozess) lesen muss, damit der Prozess endgültig aus dem System entfernt wird.

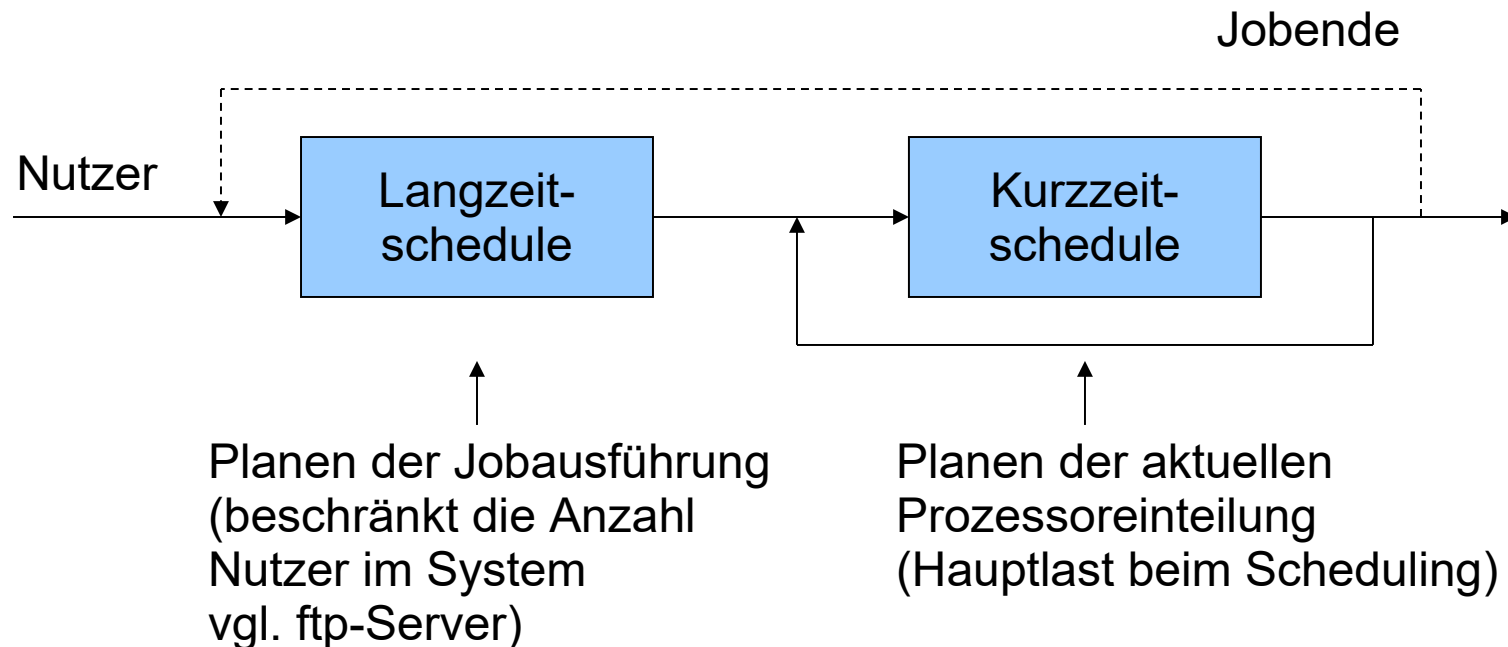


Kann die pid kleiner 0 werden? Wenn ja, wann? Wenn nein, warum nicht?

2.2 Prozessscheduling

Zugriffskoordination der Betriebsmittel (BM) ist erforderlich, falls der Bedarf größer als der Vorrat an BM ist.

Langzeit- und Kurzzeitscheduling



2.2.1 Ziele, Zielkonflikte

- Auslastung der CPU:
 - Ziel: 100%
 - Normal: 40% bis 90%
- Durchsatz (throughput) :
 - Zahl der Jobs pro Zeiteinheit soll maximal sein (wichtiges Maß).
- Faire Behandlung (fairness):
 - Jeder Benutzer soll im Mittel den gleichen CPU-Anteil erhalten.

- Ausführungszeit (turn around time)
Zeitspanne von Jobbeginn bis Jobende soll minimal sein.
Ausführungszeit = Bedienzeit
+ warten auf I/O-Ende
+ Zeit in Warteschlange bereit
- Wartezeit
kann der Scheduler selbst beeinflussen, soll minimal sein.
- Antwortzeit (response time)
in interaktiven Systemen wichtig
Die Antwortzeit, die Zeit zwischen einer Eingabe und der Antwort an die Ausgabegeräte, soll minimal sein.

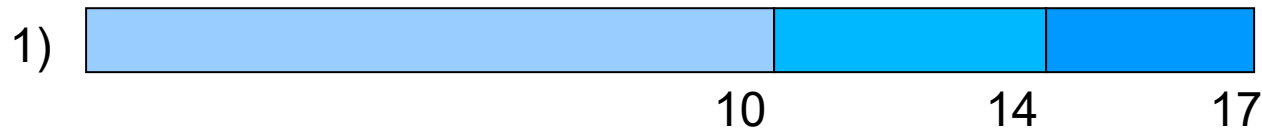
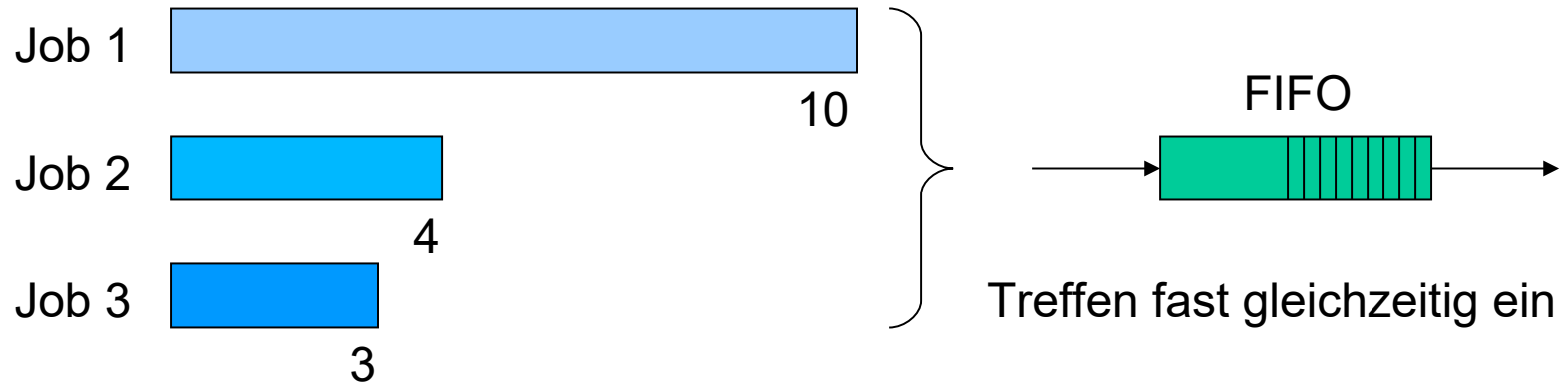
2.2.2 non-preemptives Scheduling

liegt vor, wenn Prozesse nicht vorzeitig unterbrochen werden können.
Sinnvoll, wenn man die Charakteristika der Prozesse kennt
(z.B. Dauer einer Transaktion in einem DB-Programm).

Strategien

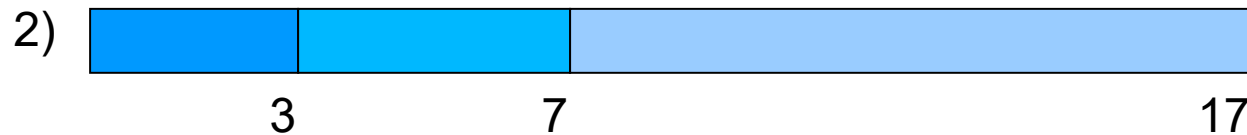
- First Come First Serve (FCFS)
Implementierung durch FIFO-Warteschlange (First In First Out)

Bsp.



Ausführungszeiten (Jobbeginn bis Jobende)

$$\text{mittlere Ausführungszeit} = \frac{10 + 14 + 17}{3} \sim 14$$

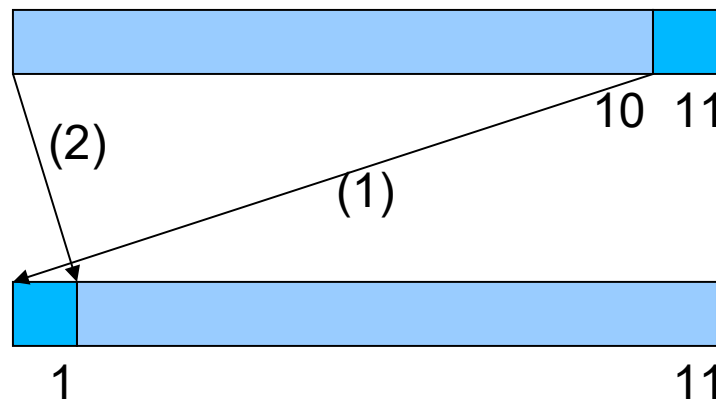


$$\text{mittlere Ausführungszeit} = \frac{3 + 7 + 17}{3} = 9 < 14 \quad !!$$

- Shortest Job First (SJF)

Der Prozess mit der geschätzt kürzesten Bedienzeit wird vorgezogen.

Bevorzugt stark interaktive Prozesse, die meist auf I/O warten.
SJF minimiert die mittlere Wartezeit.



(1) verkürzt Wartezeit stark
(2) verlängert Wartezeit kaum

Nachteil von SJF:

Verhungern von großen Jobs (starvation)

- Highest Response Ratio Next (HRN)

$$H = \frac{\text{Antwortzeit}}{\text{Bedienzeit}} \quad \text{soll groß sein}$$

Ausführungszeit = Bedienzeit
+ warten auf I/O-Ende
+ Zeit in Warteschlange bereit

$$H = \frac{A}{B}$$

H groß, wenn

- (1) B klein, d.h. zieht Jobs mit kurzer Bedienzeit vor
- (2) A groß, d.h. begrenzt Wartezeit von Jobs mit langer Bedienzeit

- Priority Scheduling (PS)

Jeder Prozess hat eine Priorität. Neue Prozesse werden über die Priorität in die Warteschlange eingeordnet.

Bei gleicher Priorität:

andere Strategie, z. B. FCFS

Achtung: Verhungern möglich!

=> Prioritätenanpassung (dynamische Priorität)

Beispiel:

p_t = aktueller Wert zum Zeitpunkt t

p_{t+1} = nächster Wert

$$p_{t+1} = \alpha * p_t(1-\alpha)^0 + \alpha * p_{t-1}(1-\alpha)^1 + \alpha * p_{t-2}(1-\alpha)^2 \\ + \dots + \alpha * p_1(1-\alpha)^{n-1} + \alpha * p_0(1-\alpha)^n$$

$\alpha < 1 \Rightarrow$ Einfluss der Vorwerte nimmt exponentiell ab

2.2.3 Preemptives Scheduling

Prozesse dürfen aus dem aktiven Zustand verdrängt werden.

Situation: verschiedene Benutzer
verschiedene Jobs + 1 Job blockiert alle anderen
→ Ärger

↙ Jobs müssen vorzeitig unterbrochen werden können.

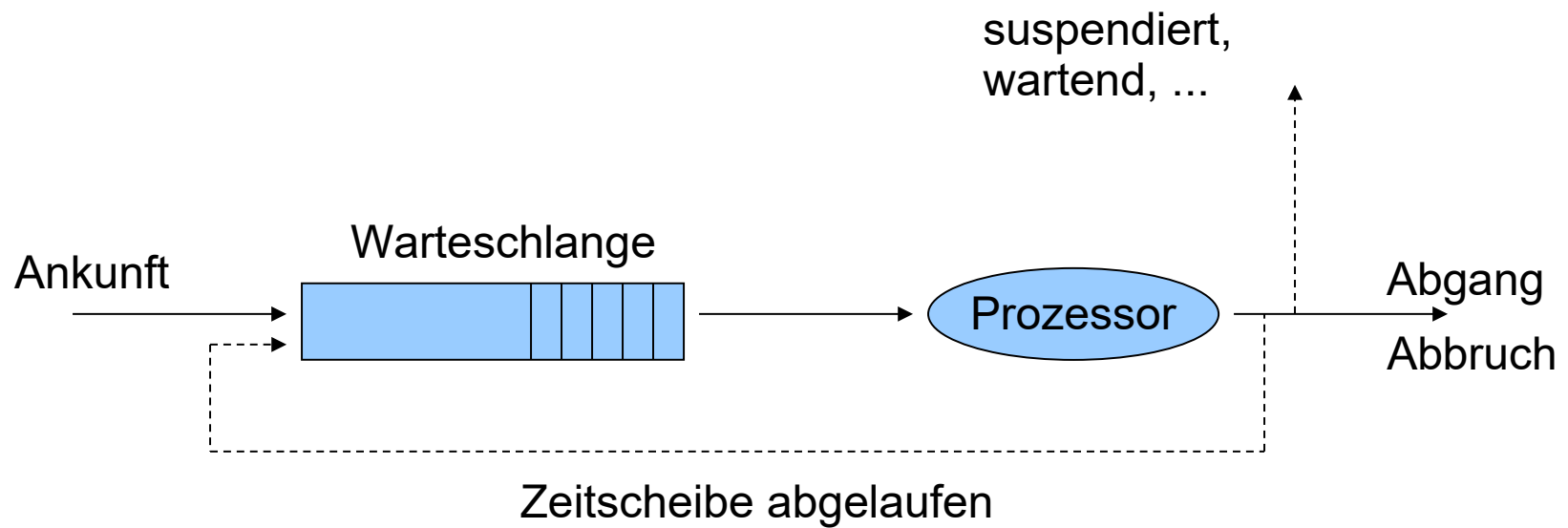
WICHTIGSTE STRATEGIE: Zeitscheibenverfahren

verfügbare Zeit für das **Betriebsmittel** CPU in einzelne, gleich große Zeitabschnitte aufteilen.

Beginnt eine neue **Zeitscheibe**, unterbricht der **Dispatcher** den aktuellen Prozess und wählt den nächsten vom Warteschlangen-anfang der **Bereit-Warteschlange** aus.

Der unterbrochene Prozess kommt in die Bereit-WS.

Preemptives Scheduling



Strategien

- Round Robin (RR)
 - RR = FCFS-Strategie + Zeitscheibenverfahren
 - Antwortzeit ist proportional zu der mittleren Bedienzeit.

Einfluss der Länge der Zeitscheibe:

1) Zeitscheibe unendlich lang => einfaches FCFS

Anmerkung: Wann ist eine Zeitscheibe unendlich lang?

2) Zeitscheibe sehr kurz => alle n Jobs erhalten die Prozessorleistung zu $1/n$, wenn der Prozesskontextwechsel ohne Zeitverlust verläuft (z.B. durch Hardwaremechanismus - das ist aber nicht der Normalfall).

Folge: $\frac{\text{Arbeitszeit}}{\text{Umschaltzeit}}$ wird kleiner und damit sinkt der Durchsatz.

Daumenregel: Wähle Zeitscheibe in der Größe des mittleren **CPU-Zeitbedarfs** zwischen zwei I/O-Aktivitäten (**CPU-burst**) von **80%** der Jobs (**~ 100 ms**)

- **Dynamic Priority Round Robin (DPRR)**
 - RR erhält als Vorstufe eine prioritätsgesteuerte WS
 - Priorität steigt mit der Zeit
 - wenn Schwellpriorität erreicht ist, kommt der Prozess in die RR-Warteschlange
- **Shortest Remaining Time First (SRTF)**
 - SJF wird zu SRTF
 - Job mit kürzester Restbedienzeit wird vorgezogen.
Frage: wo liegt das Problem in der Implementierung von SJF und SRTF?

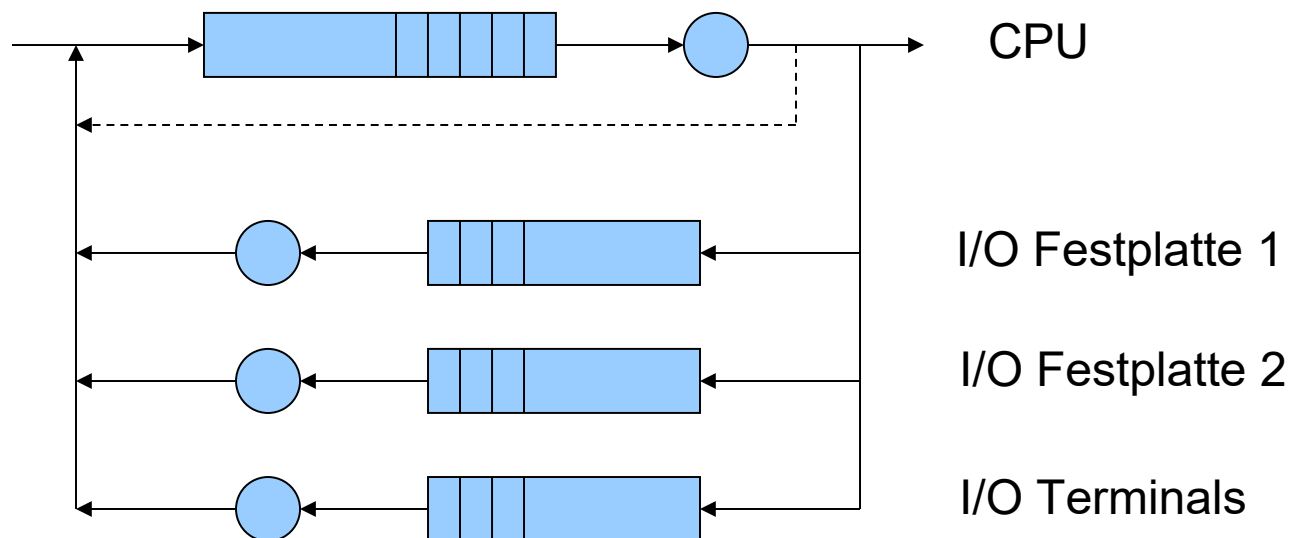
- **Priority Scheduling (PS)**

Ein Prozess wird durch einen neu ankommenden Prozess verdrängt, wenn der neue Prozess höhere Priorität hat.

- **Kombination:**
FCFS-Strategie der RR-WS durch Prioritätsstrategie ersetzen.

Multiple Warteschlangen

1. Fall: Systeme haben zwar nur eine CPU aber z.B. mehrere DMA-Kontroller (DMA: Direct Memory Access). Der DMA-Kontroller ist ein unabhängiges Betriebsmittel. Für jedes BM wird eine Warteschlange eingerichtet. Scheduling mit multiplen WS:



2. Fall: Mehrere Jobs mit unterschiedlicher Priorität.
Für jede Jobart eine eigene Warteschlange.

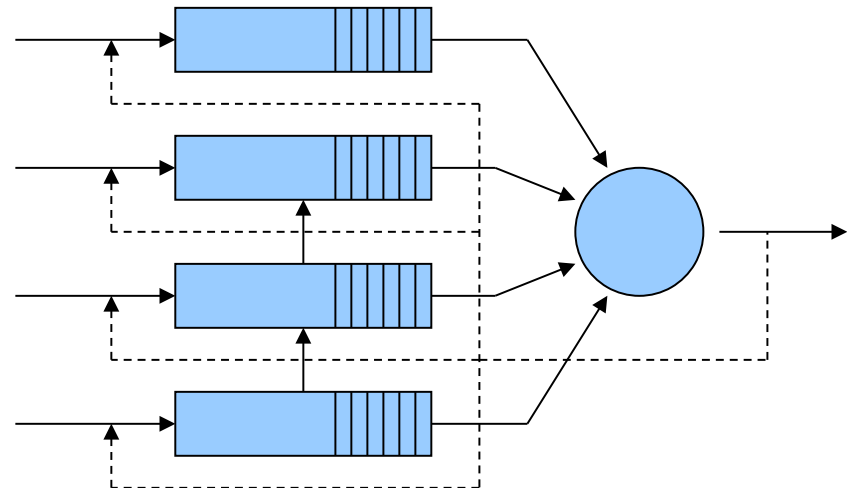
Multi-level-Scheduling

PRIOR 0: Systemprozesse

PRIOR 1: Interaktive Jobs

PRIOR 2: Allgemeine Jobs

PRIOR 3: Rechenintensive Jobs



Können die Jobs bei längerer Wartezeit in eine Warteschlange höherer Ordnung überwechseln, so spricht man von multi-level feed back Scheduling.

Multiple Scheduler

Bisher vernachlässigt: nicht alle Prozesse sind im HSP.

Das Kopieren vom HSP → Festplatte → HSP
ganzer Prozesse
heißt swapping.

Swapping verlangsamt den Prozesskontextwechsel erheblich.
Man erhält eine Optimierung für den Prozesskontextwechsel,
wenn der neu ausgewählte Prozess bereits im HSP ist.

Realisierung:

Mittelzeitscheduler (zweiter/dritter Scheduler) regelt die Zuordnung
der Prozesse zum BM HSP.

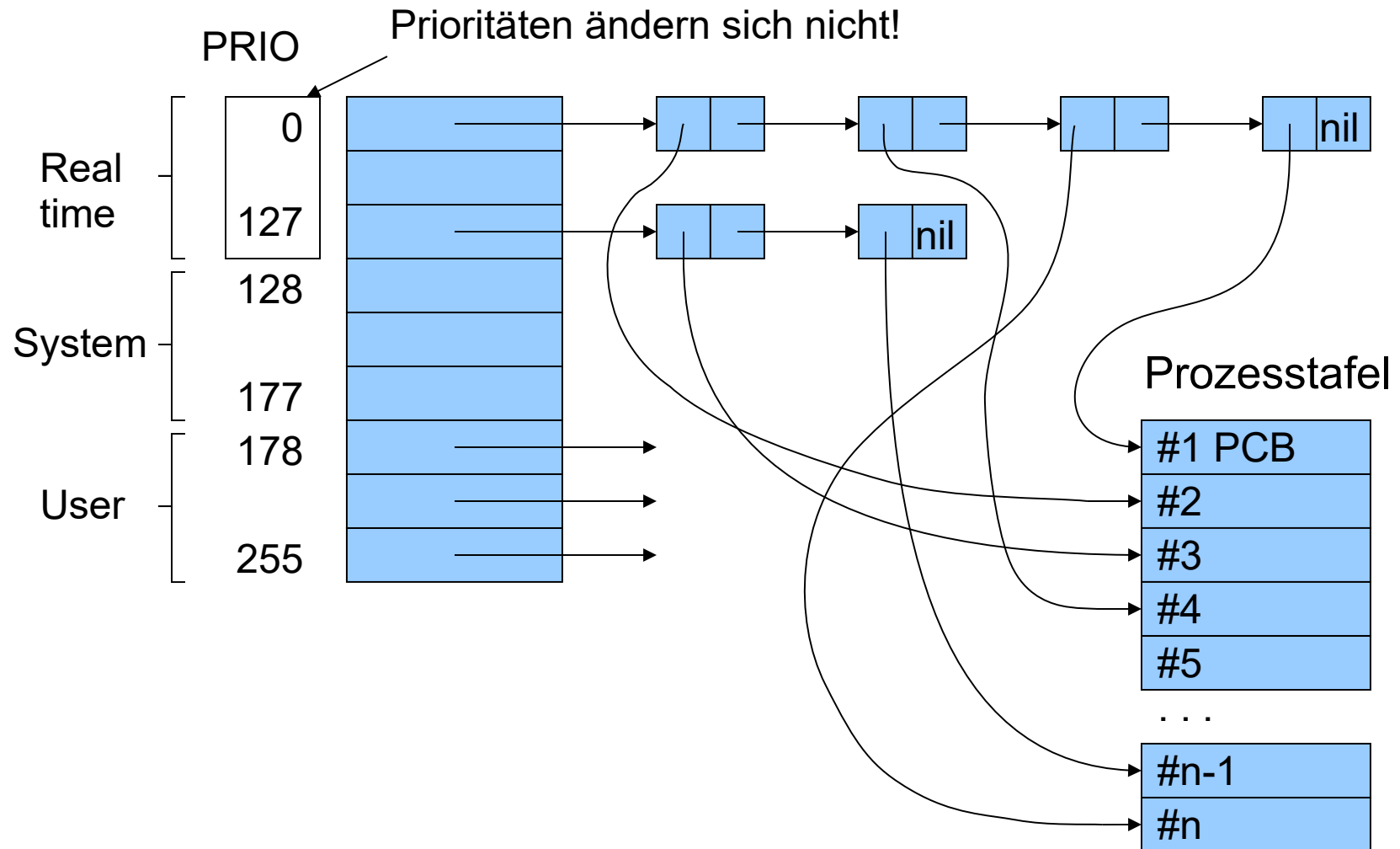
2.2.4 Beispiel Unix: Scheduling

RR mit multi-level feedback Scheduling

Jeder Prozess hat eine initiale Priorität. Sie erhöht sich mit der Zeit. Für alle Prozesse mit der gleichen Priorität gibt es eine FCFS-WS.

Die WS wird durch eine verkettete Liste realisiert. Der Informations-
teil der Listenelemente zeigt auf einen PCB in der Prozesstafel.

Beispiel HP-Unix



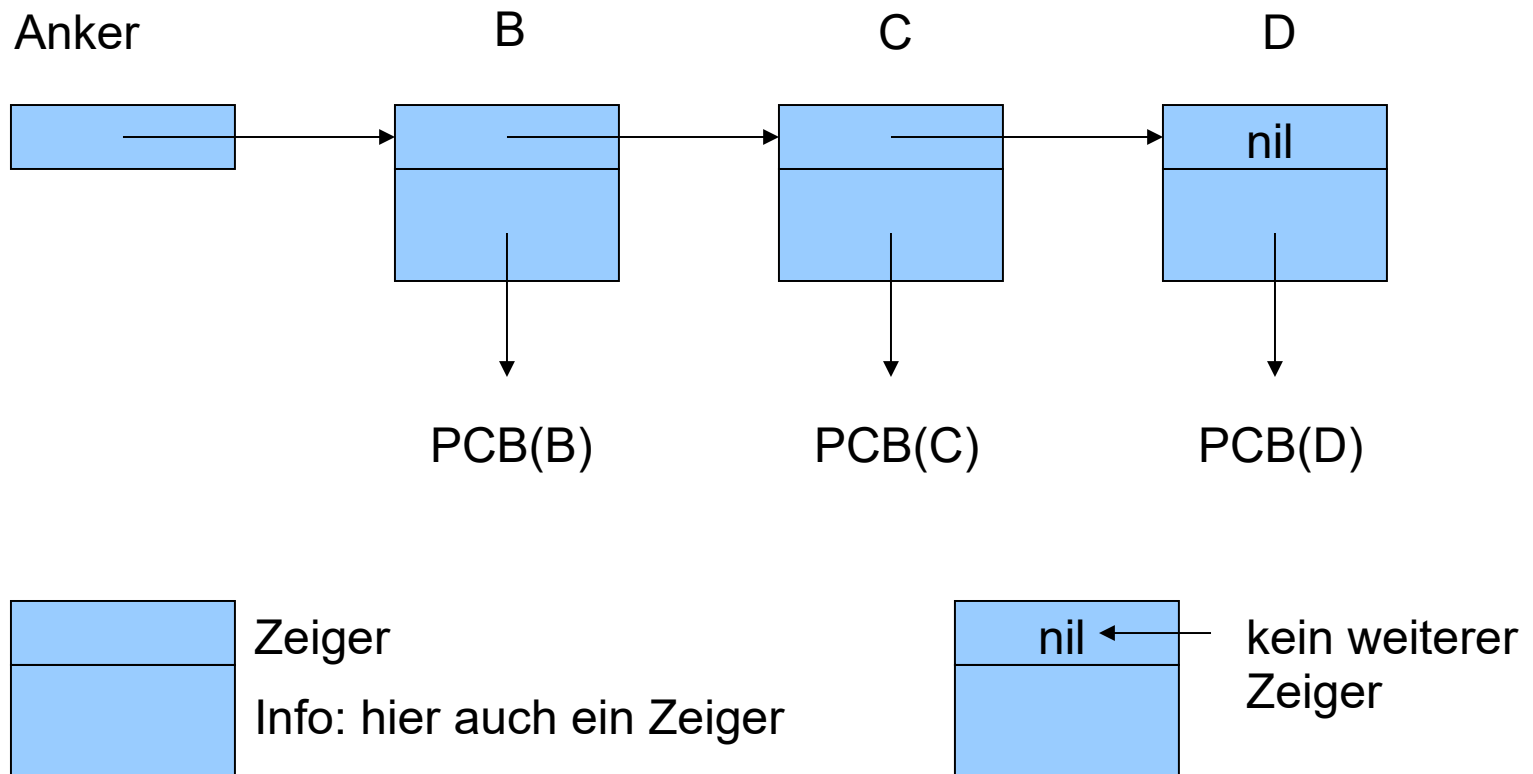
2.3 Prozesssynchronisation

Prozesse sind quasi-parallel ausführbare Programmeinheiten

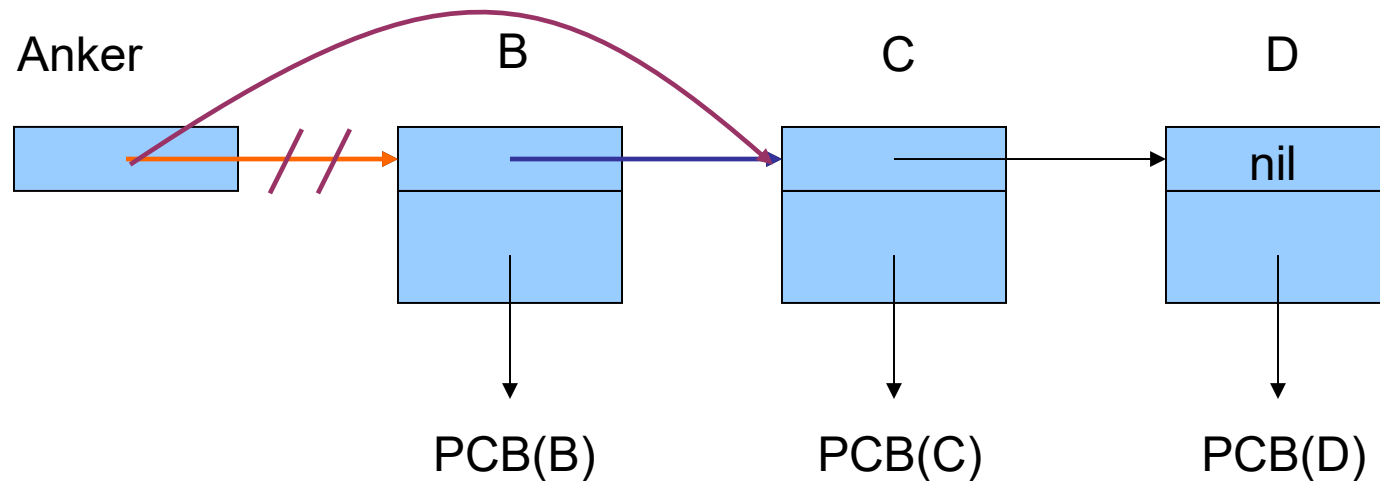
=>	Flexibilität	+	+	+			
	Effizienz	+	+	+			
	Probleme	-	-	-	-	-	-

2.3.1 Race conditions und kritische Abschnitte

LIFO-Warteschlange

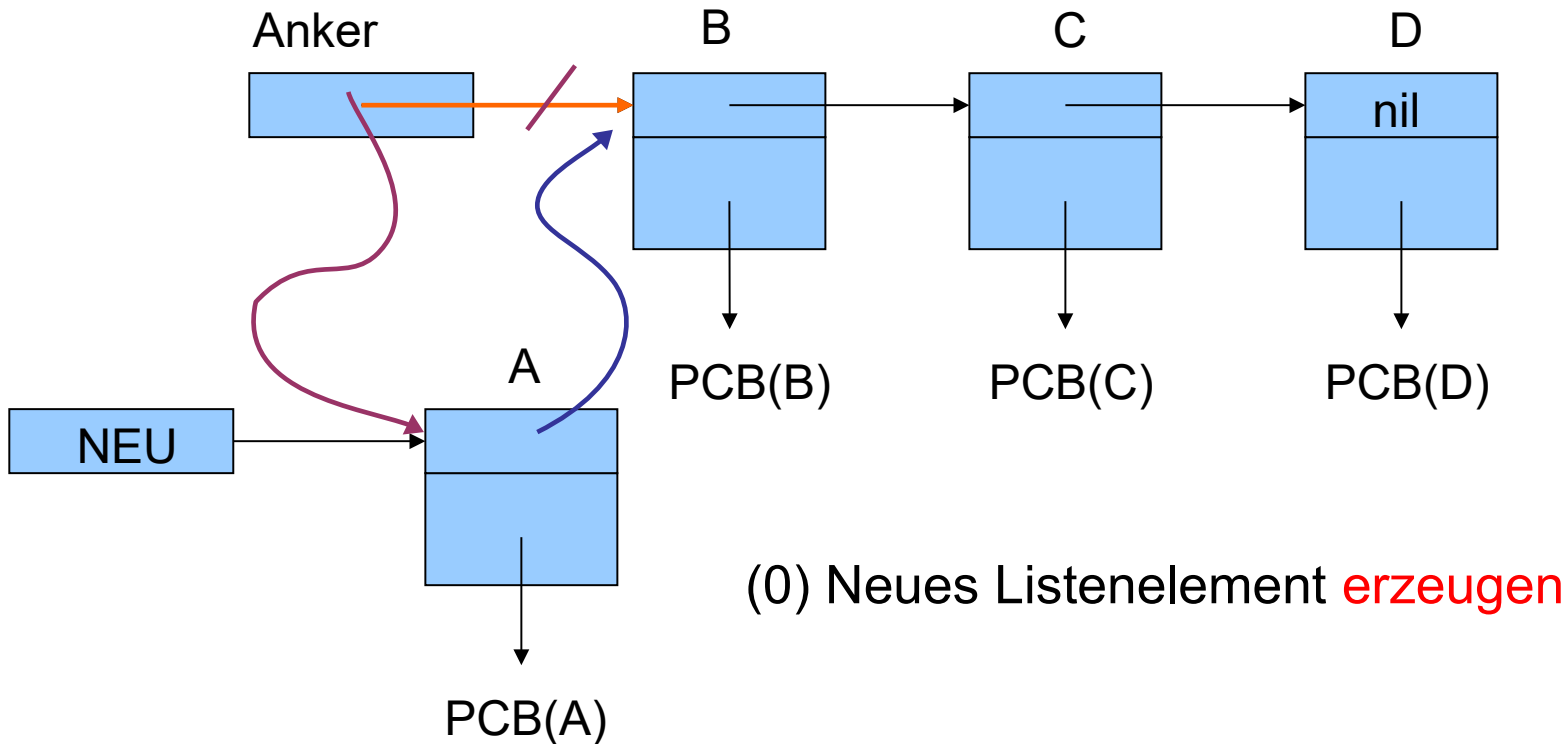


Aushängen



- (1) Lesen des **Ankers**: zeigt auf B →
- (2) Lesen des **Zeigers** von B: zeigt auf C →
- (3) Setzen des **Ankers**: zeigt auf C →

Einhängen



(1) Lesen des **Ankers**: zeigt auf B



(2) Setzen des **Zeigers** von A: zeigt auf B

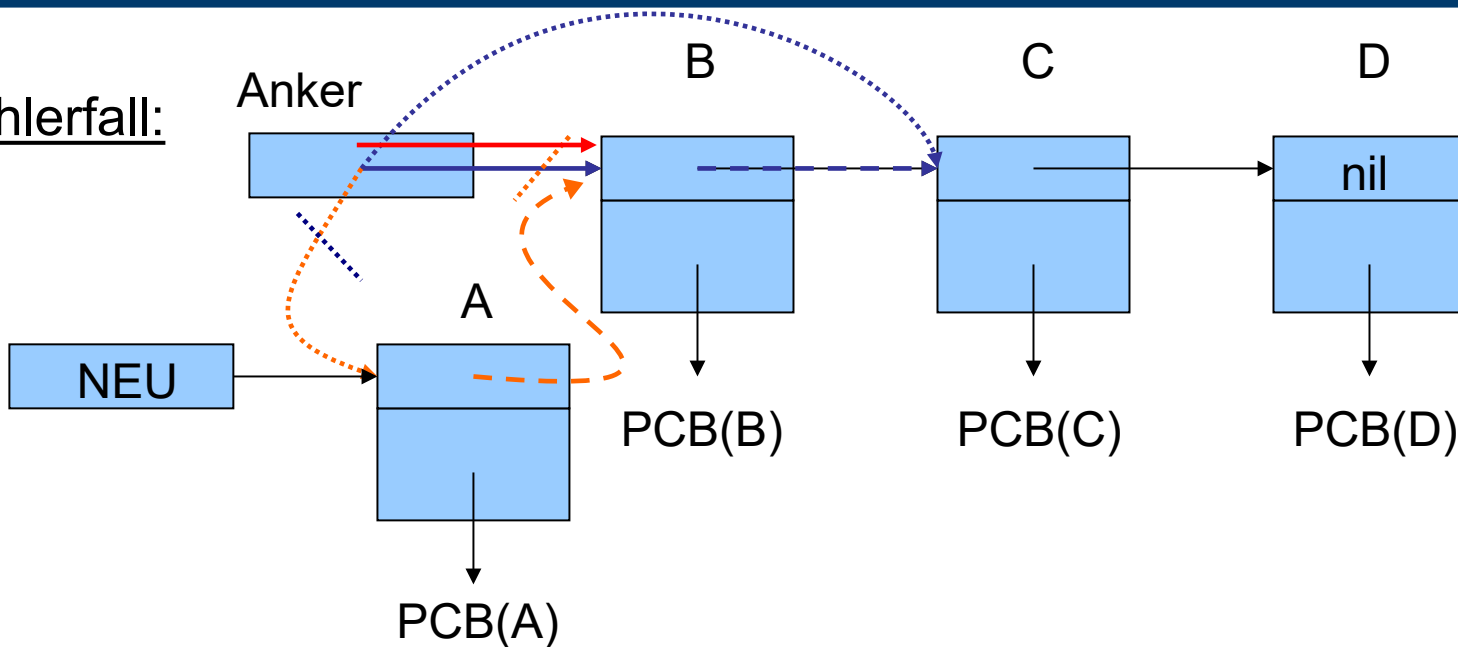




(3) Setzen des **Ankers**: zeigt auf A






Einhängen und Aushängen funktionieren,
solange die Schritte **komplett** abgearbeitet werden.

Fehlerfall:



Aushängen: (1) Lesen des Ankers: zeigt auf B 
 (2) Lesen des Zeigers von B: zeigt auf C 
 Unterbrechung

Einhängen: (1) Lesen des Ankers: zeigt auf B 
 (2) Setzen des Zeigers von A: zeigt auf B 
 (3) Setzen des Ankers: zeigt auf A 

Aushängen: (3) Setzen des Ankers: zeigt auf C  

Fehlerfall

- Fehler muss nicht immer auftreten.
- Fehler tritt sporadisch, „unerklärlich“, nicht reproduzierbar auf.
- Fehler ist an Nebenbedingungen (z.B. große Systemlast) gebunden
- Fehler tritt auf, wenn ein Prozess den anderen überholt; man nennt dies race condition.
- Codeabschnitte, in denen der Fehler entstehen kann, nennt man kritische Abschnitte (critical sections).
- Kritische Abschnitte dürfen nicht unterbrochen werden, sonst kommt es zum Fehler.

Lösung für das Problem **muss** folgende **Anforderungen erfüllen**:

- Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Abschnitten sein. (mutual exclusion)
- Es dürfen keine Annahmen über die Abarbeitungsgeschwindigkeit oder Anzahl der Prozesse bzw. Prozessoren gemacht werden.
- Kein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess blockieren.
- Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss irgendwann den Abschnitt auch betreten dürfen: Ein ewiges Warten muss ausgeschlossen sein. (fairness condition)

2.3.2 Signale, Semaphore und atomare Aktionen

Zunächst **einfache Idee**:

Prozess wartet vor kritischem Abschnitt solange, bis der Abschnitt wieder frei ist.

(1)

Prozess 1
.
.
while (nextp != 1) { /* aktives Warten, busy waiting, spin locks */ };
/* kritischer Abschnitt ... */
nextp = 2;

Lösung ?

Nein !

Prozess 2
.
.
while (nextp != 2) { /* ... */ };
/* kritischer Abschnitt ... */
nextp = 1;

Fehler: ein Prozess ..., den kritischen, Abschnitt wieder zu betreten

(2)

Prozess 1
<pre>#define TRUE (0==0) #define FALSE (0!=0) nextp1 = FALSE; while (nextp2 ##/TRUE) { ... } nextp1 = TRUE;</pre>
/* kritischer Abschnitt */
<pre>nextp1 = FALSE;</pre>

Lösung ?

Nein !

Prozess 2
<pre>nextp2 = FALSE; while (nextp1 ##/TRUE) { ... } nextp2 = TRUE;</pre>
/* kritischer Abschnitt */
<pre>nextp2 = FALSE;</pre>

Fehler: was passiert, wenn beide die Variablen nextp1 bzw. nextp2 gleichzeitig testen?

(3)

Prozess 1 ...
nextp1 = TRUE; while (nextp2 == TRUE) { /* warten */ }
/* kritischer Abschnitt */
nextp1 = FALSE;

Lösung ?

Nein !

Prozess 2 ...
nextp2 = TRUE; while (nextp1 == TRUE) { /* warten */ }
/* kritischer Abschnitt */
nextp2 = FALSE;

Fehler: beide Prozesse können für immer warten; warum?

Lösung (Peterson 1981)

Prozess 1 ...
<pre>interesep1 = TRUE; nextp = 2; while ((nextp == 2) && (interesep2)) { /* warten */ }</pre>
/* krit. Abschnitt */
<pre>interesep1 = FALSE;</pre>

Prozess 2 ...
<pre>interesep2 = TRUE; nextp = 1; while ((nextp == 1) && (interesep1)) { /* warten */ }</pre>
/* krit. Abschnitt */
<pre>interesep2 = FALSE;</pre>

Derjenige Prozess, der zuletzt nextp belegt, wartet!

```
interessep1 = TRUE;  
nextp = 2;  
while((nextp == 2) && (interessep2 == TRUE)) ...
```

Dieses Konzept kann man kompakter mit Signalen schreiben

waitfor (signal);

→ Prozess blockiert, wenn das Signal nicht gesetzt war. Ansonsten betritt er den kritischen Abschnitt und setzt das Signal zurück.

send (signal);

→ Prozess **setzt das Signal**


aktiviert anderen Prozess

Das Signal signal ist **anfänglich gesetzt**; was würde sonst passieren?

Dijkstra (1965) nannte seine Signale Semaphore
zwei elementare Operationen $p(s)$, $v(s)$

Passieren $p(s)$: Beim Eintritt in den kritischen Abschnitt wird $p(s)$ aufgerufen. Der aufrufende Prozess wartet, falls sich ein anderer Prozess in dem mit dem Semaphore korrespondierenden Abschnitt befindet.

Verlassen $v(s)$: Beim Verlassen des kritischen Abschnittes ruft der Prozess $v(s)$ auf und bewirkt damit, dass einer der evtl. wartenden Prozesse aktiviert wird und den kritischen Abschnitt betreten darf.

Beispiel: Hochzählen einer Variablen durch mehrere Prozesse

kritischer Abschnitt `p(s) ;` `z = z+1 ;` `printf ("%d\n", z) ;` `v(s) ;`

||

sind selbst kritische Abschnitte, die sich einfach mit atomaren Aktionen realisieren lassen.

Eine **atomare Aktion** wird immer entweder vollständig (alle Operationen des Abschnittes) oder gar nicht (keine einzige Operation) durchgeführt.

roll back

Hardwarelösung atomarer Aktionen:

1. Interrupts abschalten

Problematisch, da man den Timer-Interrupt und den power failure-Interrupt nicht abschalten sollte. Etwas besser: dem Prozess im kritischen Abschnitt die höchstmögliche Priorität geben.

2. Atomare Instruktionsfolgen

die test and set lock - Instruktion tsl liest den Inhalt einer Speicherzelle aus und ersetzt ihn innerhalb desselben Speicherzyklus durch einen anderen Wert.

```
int tsl (int *target) {  
    int temp = *target;  
    *target = 1;  
    return temp;  
}
```

Die Lösung von Peterson hat ein großes Problem:

Prozess muss aktiv warten

busy waiting
spin locks

Problembehebung: der wartende Prozess legt sich schlafen.

Warum ist das eine Lösung?

Eine Softwarelösung

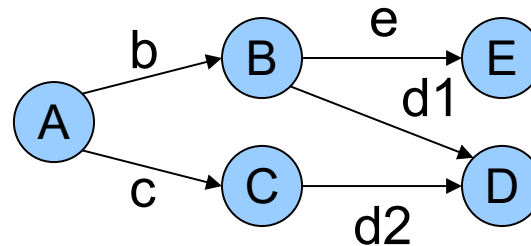
```
typedef struct {  
    int value;  
    processlist list; /* Liste der wartenden Prozesse */  
} semaphor;
```

```
void p (semaphor s) {  
    s.value = s.value - 1;  
    if (s.value < 0) {  
        einhaengen (myid, s.list);  
        sleep ();  
    }  
}  
  
void v (semaphor s) {  
    if (s.value < 0) {  
        pid = aushaengen (s.list);  
        wakeup (pid);  
    }  
    s.value = s.value + 1;  
}
```

2.3.3 Anwendungen

1. Synchronisieren von Prozessen

Der Präzedenzgraph der Tasks A, B, C, D, E sei folgendermaßen gegeben:



Dann lassen sich die Prozesse einzeln so formulieren:

```
void process_a (void) {      void process_b (void) {
    taskbody_a(); v(b); v(c);    p(b); taskbody_b(); v(d1); v(e);
}                               }
```

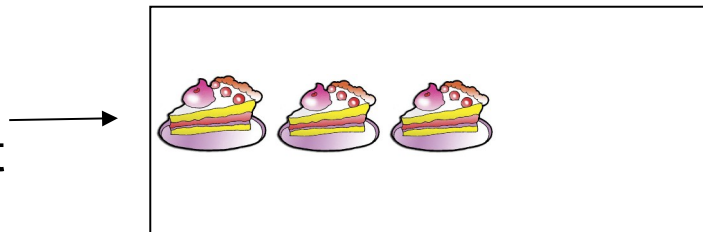


```
void process_c (void) {  
    p(c); taskbody_c(); v(d2);  
}  
  
void process_d (void) {  
    p(d1); p(d2); taskbody_d();  
}  
  
void process_e (void) {  
    p(e); taskbody_e();  
}
```

Die Semaphore b, c, d1, d2 und e müssen als globale Variable definiert sein. Sie sind mit dem Wert 0 zu initialisieren.

Das Erzeuger-Verbraucher-Problem (bounded buffer)

Erzeuger
erzeugt
und schreibt
in Puffer



Verbraucher
liest aus
Puffer und
verarbeitet

beschränkt großer
Puffer

Erzeuger

```
for (;;) {  
    produce (&item);  
    if (used == n) sleep();  
    putin_buffer(item);  
    used = used + 1;  
    if (used == 1)  
        wakeup (Verbraucher);  
}
```

Verbraucher

```
for (;;) {  
    if (used == 0) sleep();  
    getfrom_buffer (&item);  
    used = used - 1;  
    if (used == n-1)  
        wakeup (Erzeuger);  
    consume(item);  
}
```

Puffergröße ist n, global Variable used zu Anfang 0

Race-Condition:

Puffer ist leer, d.h. `used == 0` ist wahr.

Verbraucher hat gerade `used == 0` getestet und es wird auf Erzeuger umgeschaltet.

Erzeuger produziert, `used` wird 1 und weckt Verbraucher.

Verbraucher führt die **sleep**-Anweisung aus.

Erzeuger füllt den Puffer und führt die **sleep**-Anweisung aus

Folge: beide schlafen für immer.

Lösung mit Semaphoren:

Erzeuger

```
for (;;) {  
    produce(&item) ;  
    p(frei) ;  
    p(mutex) ;  
    putin_buffer(item) ;  
    v(mutex) ;  
    v(belegt) ;  
}
```

Verbraucher

```
for (;;) {  
    p(belegt) ;  
    p(mutex) ;  
    getfrom_buffer(&item) ;  
    v(mutex) ;  
    v(frei) ;  
    consume(item) ;  
}
```

mutex: wechselseitiger Ausschluss

frei + belegt: zur Prozesssynchronisation

2.3.4 Verklemmungen

Ein Beispiel mit zwei Betriebsmitteln

Prozess 1 sammelt Daten
aktualisiert Einträge in einer Datei
protokolliert dies auf Drucker

Prozess 2 druckt die gesamte Datei aus

BM: Drucker, Datei mit wechselseitigem Ausschluss geschützt.

- P1: Nach dem Datensammeln **sperrt** Prozess **P1** die **Datei**, aktualisiert einen Eintrag, will drucken vor der Aktualisierung des zweiten Eintrags.
- P2: Prozess **P2** **sperrt Drucker**, druckt Überschrift, beantragt Datei.

Prozess 1 und 2 legen sich schlafen. Beide schlafen ewig.

↘ Verklemmung (Deadlock)

Notwendige und hinreichende Verklemmungsbedingungen

- (1) Beschränkte Belegung (mutual exclusion)
Jedes involvierte Betriebsmittel (BM) ist entweder exklusiv belegt oder aber frei.
- (2) Zusätzliche Belegung (hold-and-wait)
Prozesse haben bereits BM, wollen zusätzliche und warten darauf, dass sie frei werden.
- (3) Keine vorzeitige Rückgabe (no preemption)
die bereits belegten BM können dem Prozess nicht einfach wieder entzogen werden.
- (4) Gegenseitiges Warten (circular wait)
Es muss eine geschlossene Kette von ≥ 2 Prozessen geben, bei denen jeweils einer BM vom nächsten haben möchte, die dieser schon belegt hat.

Verklemmungen bei logischen Einheiten

vgl. Erzeuger-Verbraucher-Synchronisation

Erzeuger

...

p (frei)

...

Verbraucher

...

p (belegt)

...

irrtümlicherweise frei = belegt = 0

↙ beide Prozesse schlafen für immer

Behandlung von Verklemmungen

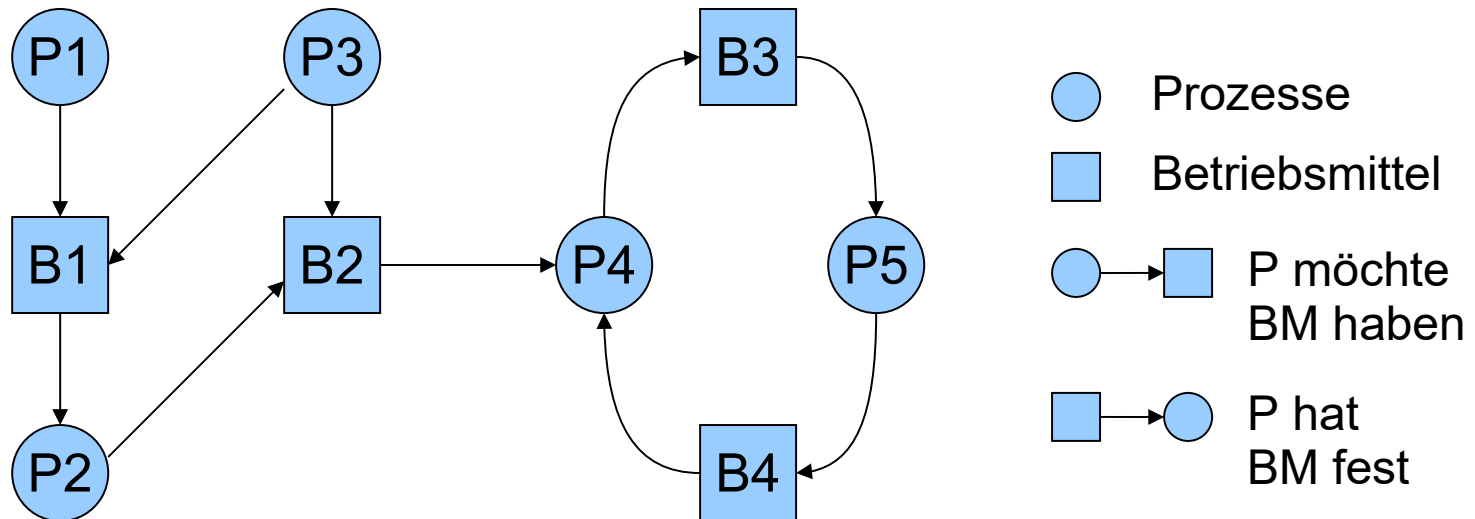
- a) das Problem ignorieren (Vogel-Strauss-Methode)
- b) Verklemmungen erkennen und beseitigen
- c) Verklemmungen vermeiden
- d) Verklemmungen unmöglich machen, indem man eine der Bedingungen (1) - (4) verhindert

zu a) weitere Störungsmöglichkeiten sind häufiger

Hardware: schlechte Lötstellen (kalte Lötstellen)
wackelige Kabel (Wackelkontakt)
Chipausfälle
Netzspannungsschwankungen

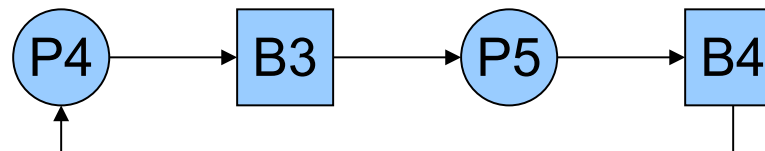
Software: Fehler im Betriebssystem
unterschiedliche Compilerversionen
Umkonfiguration \$HOME
falsche Dateneingabe

zu b) Betriebsmittelgraph (zum Erkennen + Beseitigen)



Notwendige und hinreichende Verklemmungsbedingung:

Zyklen im Graphen:

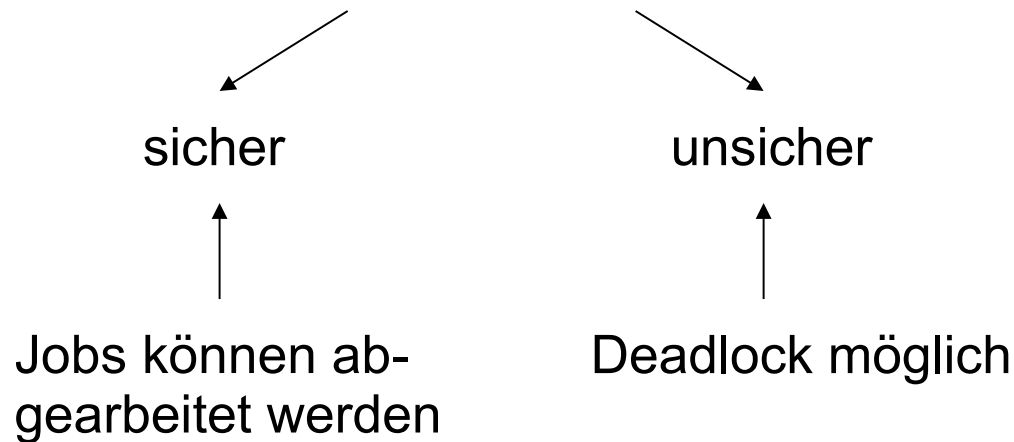


Verklemmungen beseitigen:

- Prozess abbrechen
- Prozess zurücksetzen
- BM dem Prozess entziehen

zu c) Verklemmungen vermeiden

Idee: Zustände einteilen



Lösung: sich nur in sicheren Zuständen bewegen.

Dijkstra entwickelte 1965 den Banker-Algorithmus.

BM (begrenztes Kapital)

P (Kreditwünsche)

zu d) Verklemmungen unmöglich machen

- 1) mutual exclusion:
Konkurrenz für BM abschaffen
Druckerdaemon + Warteschlange
geht nicht immer: z.B. für Semaphore
- 2) hold and wait:
alle BM auf einmal anfordern
- 3) no preemption:
vorzeitiger Entzug möglich = preemption
- 4) circular wait
Den Zyklus direkt bei der Anforderung von BM verhindern.
Definition einer Ordnungsrelation auf den BMn => Linearisierung.
Prozesse dürfen BM nur in aufsteigender Nummerierung belegen.

Ergänzung zu b – vgl. WertheApp)

Es gebe mehrere BM vom Typ s .

E ist ein Vektor, der die Zahl der existierenden BM vom Typ s angibt.

$E = (6 \ 3 \ 4 \ 2)$ heißt:

$E_1 = 6$ z.B. 6 tapes

$E_2 = 3$ z.B. 3 plotter

$E_3 = 4$ z.B. 4 printer

$E_4 = 2$ z.B. 2 CD-ROM-LW

E_s ist die Zahl der existierenden Einheiten pro BM vom Typ s .

B_{ks} Zahl der Einheiten vom Typ s , die Prozess k belegt hat.

C_{ks} Zahl der Einheiten vom Typ s , die Prozess k zusätzlich anfordert.

A_s Anzahl der noch freien BM vom Typ s . $A_s = E_s - \sum_{\text{über } k} B_{ks}$

Algorithmus zur Verklemmungserkennung

- (0) Alle Prozesse sind unmarkiert
- (1) **Suche** einen unmarkierten Prozess P_k , bei dem für alle BM s gilt $A_s \geq C_{ks}$
- (2) Falls ein solcher Prozess existiert, markiere ihn und bilde $A_s := A_s - C_{ks} + B_{ks} + C_{ks} = A_s + B_{ks}$
- (3) Gab es **keinen** solchen Prozess, Fertig \searrow **Verklemmung**
- (4) Gibt es noch unmarkierte Prozesse, dann **gehe zu (1)**, sonst Fertig \searrow **KEINE Verklemmung**

Beispiel: $E = (6 \ 3 \ 4 \ 2)$

bestehende Belegung

tapes plotter printer CD-ROM-LW

$B =$	(P_1)	3	0	1	1
	(P_2)	0	1	0	0
	(P_3)	1	1	1	0
	(P_4)	1	1	0	1
	(P_5)	0	0	0	0

Gesamt-
belegung

5	3	2	2
---	---	---	---

E

6	3	4	2
---	---	---	---

$E - B \rightarrow A$

1	0	2	0
---	---	---	---

Wünsche

$$C = \begin{matrix} (P_1) \\ (P_2) \\ (P_3) \\ (P_4) \\ (P_5) \end{matrix} \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 1 & 2 \\ \hline 3 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 2 & 1 & 1 & 0 \\ \hline \end{array}$$

A	1	0	2	0
$C_{(P4)} \checkmark$	0	0	1	0
B_4	1	1	0	1

A	2	1	2	1
$C_{(P1)} \checkmark$	1	1	0	0
B_1	3	0	1	1

A	2	1	2	1	Übertrag
$C_{(P1)} \checkmark$	1	1	0	0	
B_1	3	0	1	1	
<hr/>					
A	5	1	3	2	
$C_{(P2)} \checkmark$	0	1	1	2	
B_2	0	1	0	0	
<hr/>					
A	5	2	3	2	
$C_{(P3)} \checkmark$	3	1	0	0	
B_3	1	1	1	0	
<hr/>					
A	6	3	4	2	
$C_{(P5)} \checkmark$	2	1	1	0	
B_5	0	0	0	0	
<hr/>					
	6	3	4	2	

Alle Prozesse sind markiert \searrow fertig: keine Verklemmung.

C =

(P_1) ✓	1	1	0	0
(P_2) ✓	0	1	1	2
(P_3) ✓	3	1	0	0
(P_4) ✓	0	0	1	0
(P_5) ✓	2	1	1	0

2.4 Prozesskommunikation

Programme sind jetzt Bausteine.

Durch koordinierendes Zusammenwirken entsteht ein neues Programm.

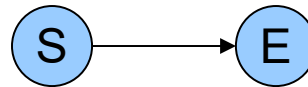
Die Interprozesskommunikation als Betriebssystemdienst ist erst mit Unix eingeführt worden. Sie ist die Grundlage für das Arbeiten über Rechengrenzen hinweg.

2.4.1 Kommunikation mit Nachrichten

Beim Nachrichtenaustausch gibt es prinzipiell 3 Verbindungsarten.

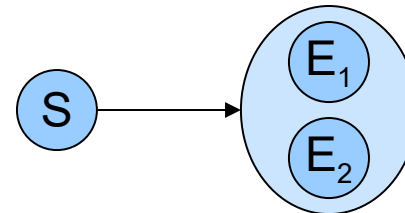
a) unicast:

1 Sender, 1 Empfänger



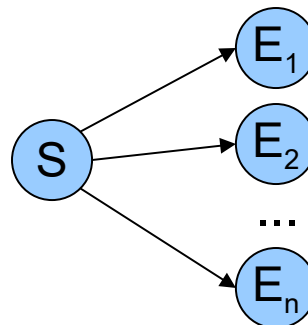
b) multicast:

1 Sender, mehrere Empfänger



c) broadcast:

1 Sender, alle Empfänger



Alle Verbindungen lassen sich ineinander überführen.

multicast, broadcast → unicast:

Angabe einer Empfängeradresse

unicast → multicast, broadcast:

Sender sendet nacheinander an alle Empfänger
oder

Sender übergibt Empfängerliste an ersten Empfänger,
dieser sendet die Nachricht weiter (streicht den
Empfänger aus der Liste)

2.4.2 Verbindungsorientierte Kommunikation

ähnlich einer Telefonverbindung

- open connection (Adresse)
Feststellen, ob der Empfänger existiert und bereit ist; Aufbau eines Kanals zur „synchronen“ Übertragung.
- send (message) / receive (message)
Nachrichtenaustausch
- close connection
Leeren der Nachrichtenpuffer; Beenden der Verbindung

Aufwand für Aufbau und Unterhalt des Kanals ist hoch!

2.4.3 Verbindungslose Kommunikation

Ähnlich der Briefpost.

Nachrichten werden „asynchron“ verschickt.

send (Adresse, message) / receive (Adresse, message)

Es ist nicht sichergestellt, dass der Empfänger die Nachricht erhält!



empfangene Nachrichten quittieren.

Was passiert, wenn die Quittung verloren geht?

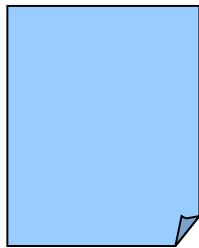
Sender sendet erneut die (überflüssige) Nachricht!

Der Empfänger muss diese Nachricht als alt erkennen.

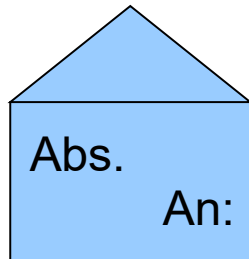
Dies ist über eine fortlaufende Nr. der Nachrichten möglich.

Wie sieht eine Nachricht aus?

vgl. Brief



Inhalt
auf
Papier



Umschlag
mit Versende-
daten

Nachricht (message)



header: Empfänger-Adr.
Sender-Adr.
Nachrichtentyp
Sequenz-Nr.
Länge
Daten (Zeiger)

Der Aufbau des headers muss Sender + Empfänger bekannt sein.

Art des Senden & Empfangen

blockierend,
synchron

Nach dem Senden
wird der Sender
solange verzögert,
bis die Rückmeldung
erfolgt ist.

nicht blockierend,
asynchron

System puffert die
Nachricht und kümmert
sich um die korrekte
Ablieferung. Sender kann
weiterarbeiten.
! Sender muss verzögert
werden, wenn Puffer voll ist.

Adressierung

Punkt-zu-Punkt-Verbindung: über Prozessnummer



versch. Rechner: über Prozessnummer + Rechnername



physikalische Namensgebung,
nicht immer bekannt!



logische Namen

z.B. PID des Druckprozesses wird durch logischen Druckernamen ersetzt (Zuordnung durch Tabelle im BS-Kern!)

Ausdehnung auf allg. Kommunikation über Rechnernetze:

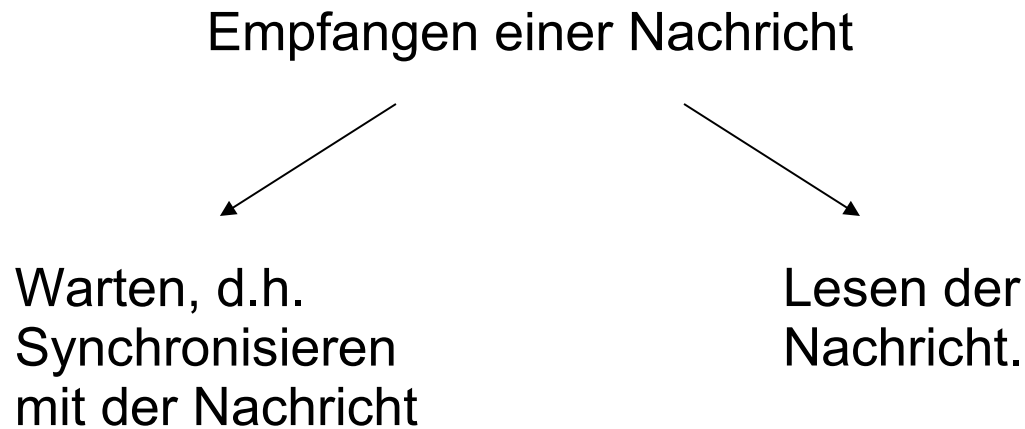
Informationen über Prozesse und Rechner werden nicht auf jedem Rechner gespeichert, sondern auf sog. name server, die eine Adressauflösung durchführen.

- Mailboxen

Beim asynchronen Nachrichtenaustausch müssen Nachrichten, die der Empfänger nicht sofort liest, gepuffert werden. Gibt man dem Puffer einen Namen, braucht man den Prozess nicht mehr zu kennen, der die Nachricht lesen soll.

2.4.4 Prozesssynchronisation durch Kommunikation

Synchronisation erreicht man durch Warten auf sehr kurze Nachrichten (Signale).



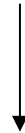
Nachricht der Länge Null → Hauptaufgabe Synchronisation

send (message)



send (signal)

receive (message)



waitfor (signal)

↓ „lässt sich abbilden.“

Signale (kurze Nachrichten) in Unix

werden zum Signalisieren an Prozesse genutzt:

Melden das Auftreten von Ereignissen, z.B. Fehlern wie

- unbekannte Speicheradresse
- Division durch Null