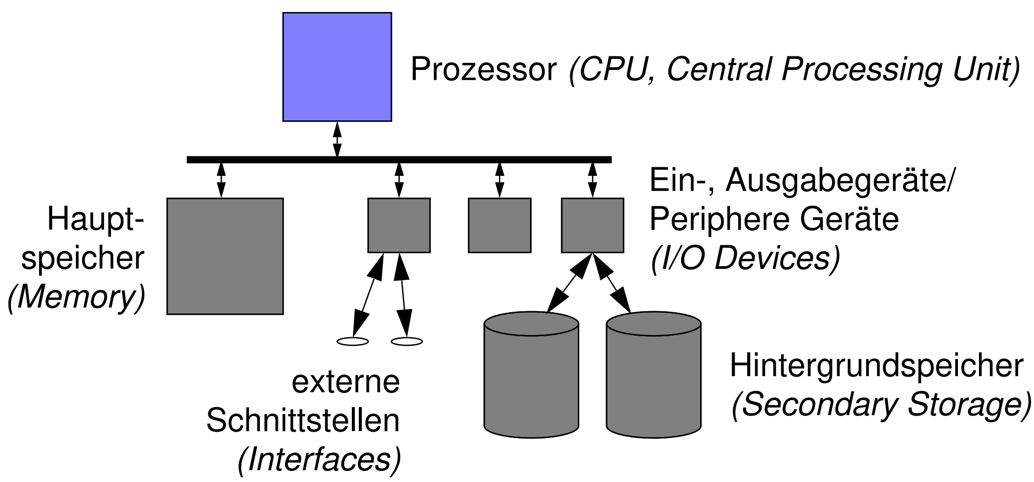
D Prozesse und Nebenläufigkeit

- Einordnung



1 Prozessor

- Register
  - Prozessor besitzt Steuer- und Vielzweckregister
  - Steuerregister:
    - Programmzähler *(Instruction Pointer)*
    - Stapelregister *(Stack Pointer)*
    - Statusregister
    - etc.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
  - Instruktion wird geladen und
  - ausgeführt
  - Programmzähler wird inkrementiert
  - dieser Vorgang wird ständig wiederholt

1 Prozessor (2)

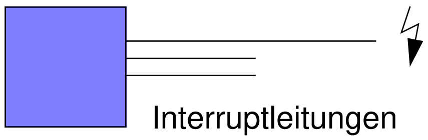
- Beispiel für Instruktionen

```
...
0010 5510000000   movl DS:$10, %ebx
0015 5614000000   movl DS:$14, %eax
001a 8a           addl %eax, %ebx
001b 5a18000000   movl %ebx, DS:$18
...
```

- Prozessor arbeitet in einem bestimmten Modus
  - Benutzermodus: eingeschränkter Befehlssatz
  - privilegierter Modus: erlaubt Ausführung privilegierter Befehle
    - Konfigurationsänderungen des Prozessors
    - Moduswechsel
    - spezielle Ein-, Ausgabebefehle

1 Prozessor (3)

- Unterbrechungen *(Interrupts)*



  - ◆ Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
  - ◆ vorher werden alle Register einschließlich Programmzähler gesichert (z.B. auf dem Stack)
  - ◆ nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden
  - ◆ Unterbrechungen werden im privilegierten Modus bearbeitet

1 Prozessor (4)

- Systemaufrufe *(Traps; User Interrupts)*
  - ◆ Wie kommt man kontrolliert vom Benutzermodus in den privilegierten Modus?
  - ◆ spezielle Befehle zum Eintritt in den privilegierten Modus
  - ◆ Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
  - ◆ solche Befehle werden dazu genutzt die Betriebssystemschnittstelle zu implementieren *(Supervisor Calls)*
  - ◆ Parameter werden nach einer Konvention übergeben (z.B. auf dem Stack)

2 Prozesse

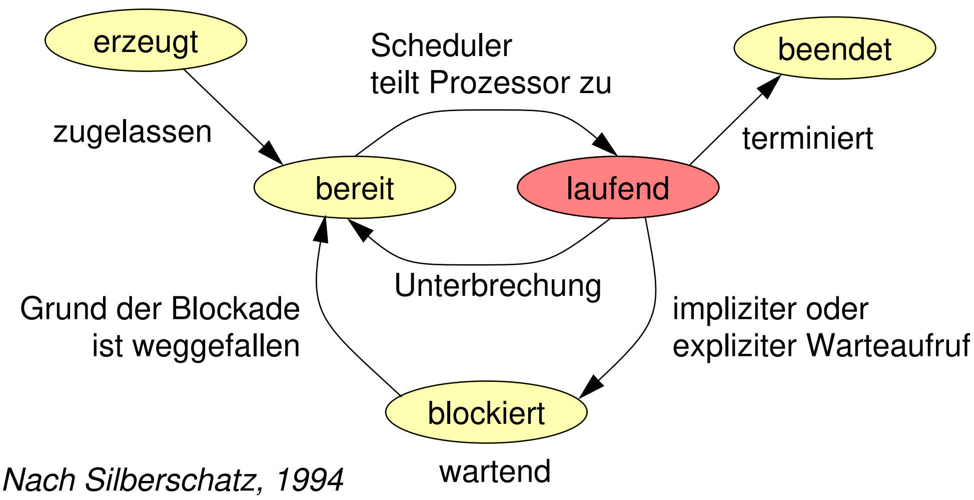
- Stapelsysteme *(Batch Systems)*
  - ein Programm läuft auf dem Prozessor von Anfang bis Ende
- Heutige Systeme *(Time Sharing Systems)*
  - mehrere Programme laufen gleichzeitig
  - Prozessorzeit muss den Programmen zugeteilt werden
  - Programme laufen nebenläufig
- Terminologie
  - **Programm:** Folge von Anweisungen
    (hinterlegt beispielsweise als Datei auf dem Hintergrundspeicher)
  - **Prozess:** Programm, das sich in Ausführung befindet, und seine Daten
    (*Beachte*: ein Programm kann sich mehrfach in Ausführung befinden)

2.1 Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** *(New)*
    Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** *(Ready)*
    Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** *(Running)*
    Prozess wird vom realen Prozessor ausgeführt
  - **Blockiert** *(Blocked/Waiting)*
    Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - **Beendet** *(Terminated)*
    Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

2.1 Prozesszustände (2)

- Zustandsdiagramm

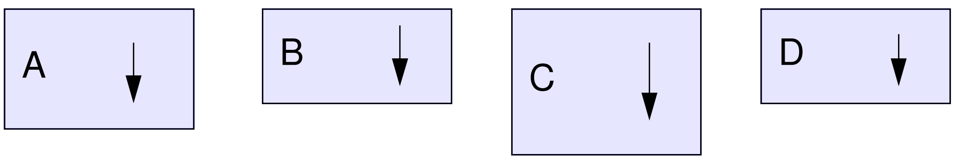


Nach Silberschatz, 1994

◆ Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors vornimmt.

2.2 Prozesswechsel

- Konzeptionelles Modell



vier Prozesse mit eigenständigen Befehlszählern

- Umschaltung *(Context Switch)*

◆ Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),

◆ Auswahl des neuen Prozesses,

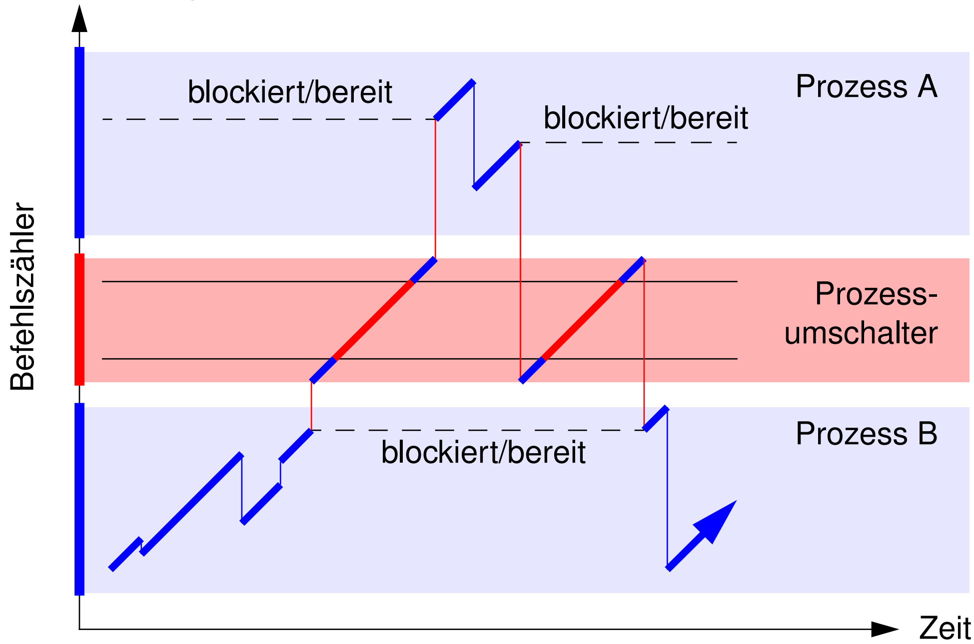
◆ Ablaufumgebung des neuen Prozesses herstellen (z.B. Speicherabbildung, etc.),

◆ gesicherte Register laden und

◆ Prozessor aufsetzen.

2.2 Prozesswechsel (2)

- Umschaltung



2.2 Prozesswechsel (3)

- Prozesskontrollblock *(Process Control Block; PCB)*
  - ◆ Datenstruktur, die alle nötigen Daten für einen Prozess hält. Beispielsweise in UNIX:
    - Prozessnummer *(PID)*
    - verbrauchte Rechenzeit
    - Erzeugungszeitpunkt
    - Kontext (Register etc.)
    - Speicherabbildung
    - Eigentümer *(UID, GID)*
    - Wurzelkatalog, aktueller Katalog
    - offene Dateien
    - ...

2.2 Prozesswechsel (4)

- Prozesswechsel unter Kontrolle des Betriebssystems
  - Mögliche Eingriffspunkte:
    - Systemaufrufe
    - Unterbrechungen
  - Wechsel nach/in Systemaufrufen
    - Warten auf Ereignisse
      (z.B. Zeitpunkt, Nachricht, Lesen eines Plattenblock)
    - Terminieren des Prozesses
  - Wechsel nach Unterbrechungen
    - Ablauf einer Zeitscheibe
    - bevorzugter Prozess wurde laufbereit
- Auswahlstrategie zur Wahl des nächsten Prozesses
  - *Scheduler*-Komponente

2.3 Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

2.3 Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - ◆ Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Kind

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```


2.3 Prozesserzeugung (2)

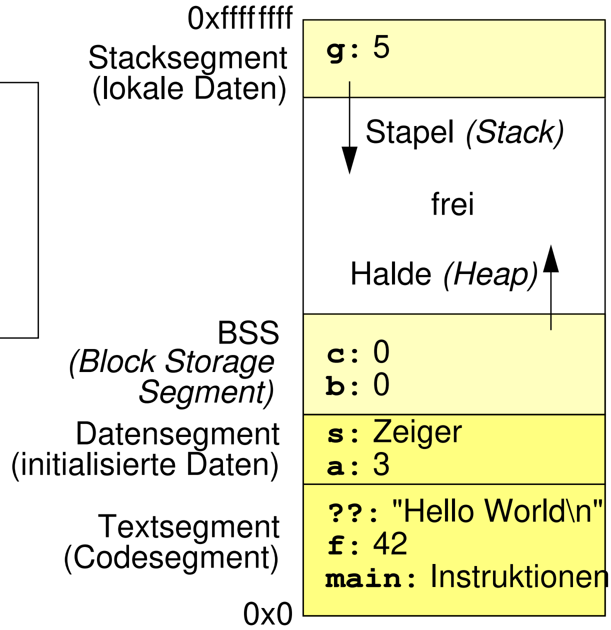
◆ Der Kind-Prozess ist eine perfekte Kopie des Vaters
- Gleiches Programm
- Gleiche Daten (gleiche Werte in Variablen)
- Gleicher Programmzähler (nach der Kopie)
- Gleicher Eigentümer
- Gleiches aktuelles Verzeichnis
- Gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
- ...

◆ Unterschiede:
- Verschiedene PIDs
- **fork()** liefert verschiedene Werte als Ergebnis für Vater und Kind

2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
int a= 3, b, c= 0;
const int f= 42;
const char *s= "Hello World\n";

int main( ... ) {
    int g= 5;
}
```

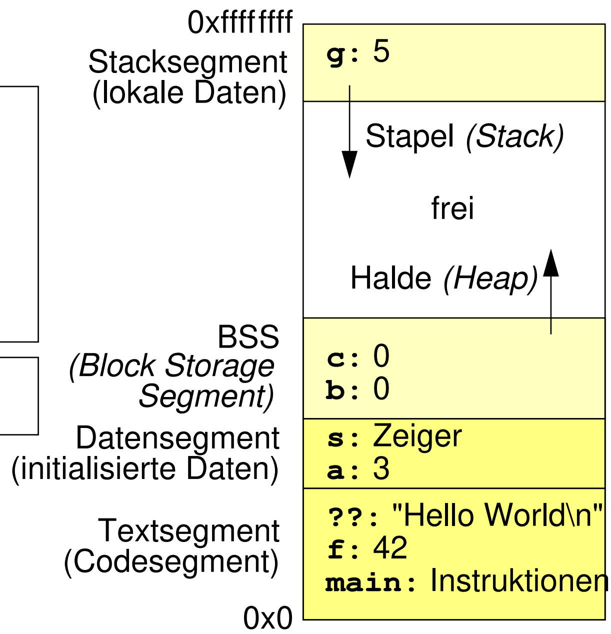


2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
int a= 3, b, c= 0;
const int f= 42;
const char *s= "Hello World\n";

int main( ... ) {
    int g= 5;
}
```


```
s[1]= 'a';
f= 2;
```

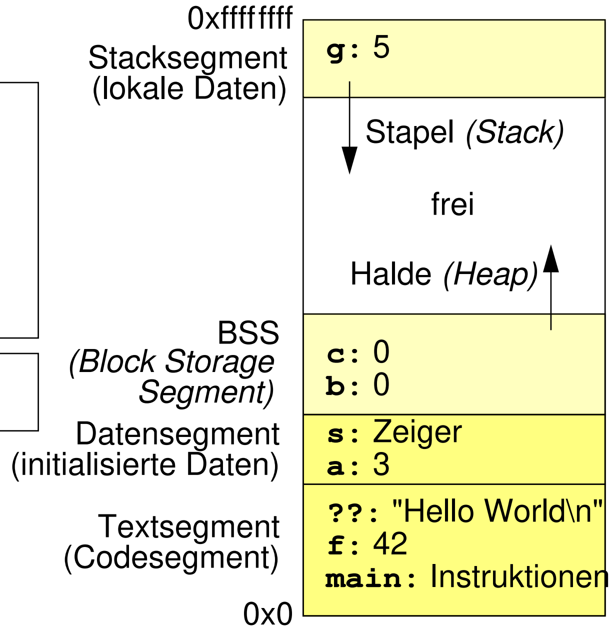


2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
int a= 3, b, c= 0;
const int f= 42;
const char *s= "Hello World\n";

int main( ... ) {
    int g= 5;
}
```

```
s[1]= 'a';          /* cc error */
f= 2;               /* cc error */
```

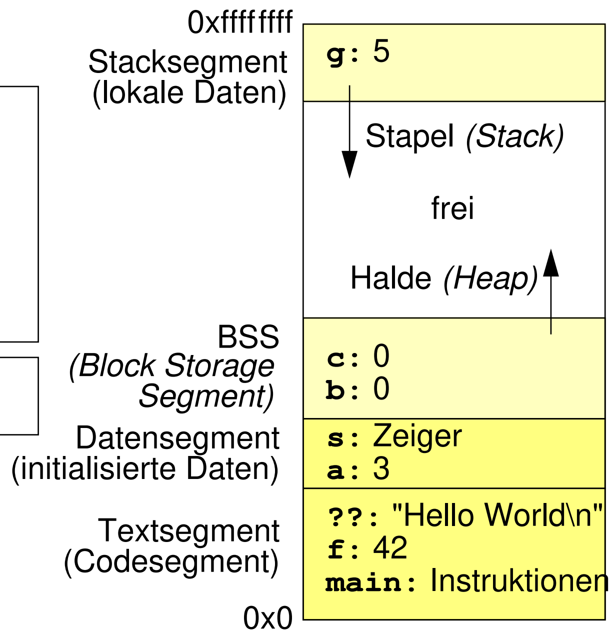


2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
int a= 3, b, c= 0;
const int f= 42;
const char *s= "Hello World\n";

int main( ... ) {
    int g= 5;
}
```

```
((char*)s)[1]= 'a';
*((int *)&f)= 2;
```

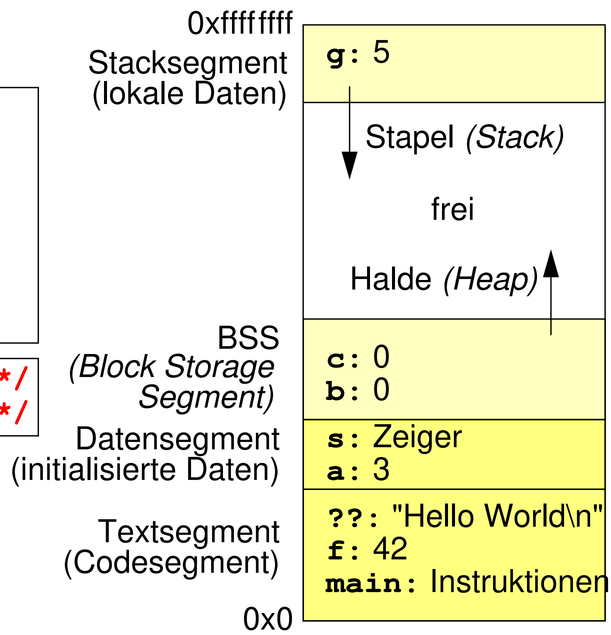


2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
int a= 3, b, c= 0;
const int f= 42;
const char *s= "Hello World\n";

int main( ... ) {
    int g= 5;
}
```

```
((char*)s)[1]= 'a';  /* SIGSEGV */
*((int *)&f)= 2;     /* SIGSEGV */
```



2.5 Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execve( const char *path, char *const argv[],
            char *const envp[] );
```

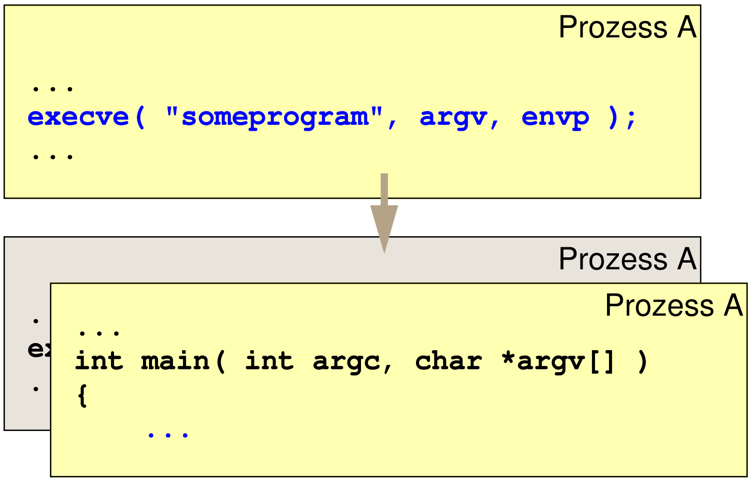
Prozess A

```
...
execve( "someprogram", argv, envp );
...
```

2.5 Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execve( const char *path, char *const argv[],
            char *const envp[] );
```



Altes ausgeführtes Programm ist endgültig beendet.

2.6 Operationen auf Prozessen (UNIX)

◆ Prozess beenden

```
void _exit( int status );
[ void exit( int status ); ]
```

◆ Prozessidentifikator

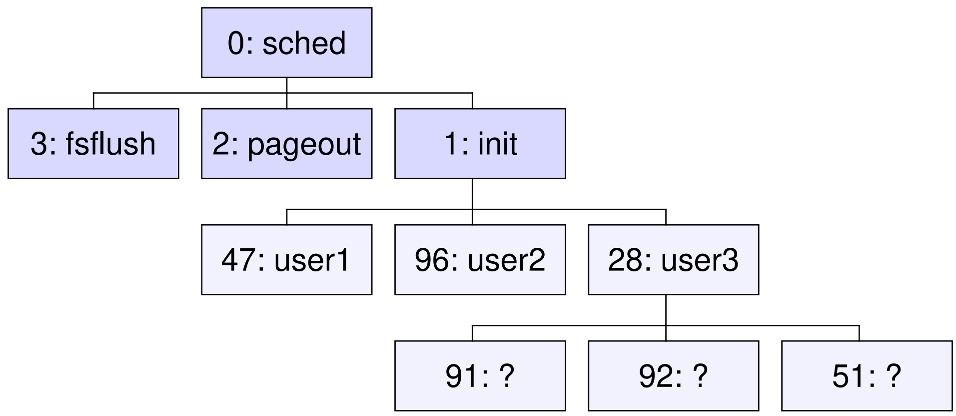
```
pid_t getpid( void );       /* eigene PID */
pid_t getppid( void );      /* PID des Vaterprozesses */
```

◆ Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

2.7 Prozesshierarchie (Solaris)

- Hierarchie wird durch Vater-Kind–Beziehung erzeugt



Frei nach Silberschatz 1994

◆ Nur der Vater kann auf das Kind warten

◆ Init-Prozess adoptiert verwaiste Kinder

3 Auswahlstrategien

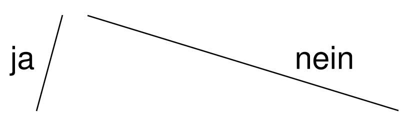
- Strategien zur Auswahl des nächsten Prozesses *(Scheduling Strategies)*

◆ Mögliche Stellen zum Treffen von Scheduling-Entscheidungen

1. Prozess wechselt vom Zustand „laufend“ zum Zustand „blockiert“ (z.B. Ein-, Ausgabeoperation)
2. Prozess wechselt von „laufend“ nach „bereit“ (z.B. bei einer Unterbrechung des Prozessors)
3. Prozess wechselt von „blockiert“ nach „bereit“
4. Prozess terminiert

◆ Keine Wahl bei 1. und 4.

◆ Wahl bei 2. und 3.

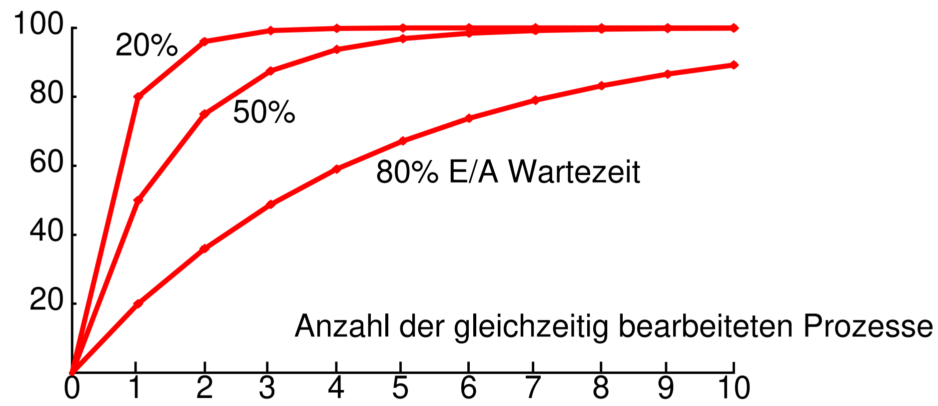


verdrängend nicht verdrängend

3 Auswahlstrategien (2)

- CPU Auslastung
  - ◆ CPU soll möglichst vollständig ausgelastet sein

★ CPU-Nutzung in Prozent, abhängig von der Anzahl der Prozesse und deren prozentualer Wartezeit



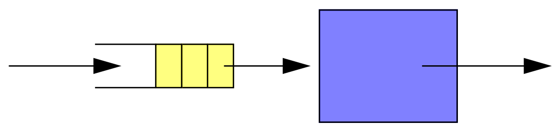
Nach Tanenbaum, 1995

3 Auswahlstrategien (3)

- Durchsatz
  - ◆ Möglichst hohe Anzahl bearbeiteter Prozesse pro Zeiteinheit
- Verweilzeit
  - ◆ Gesamtzeit des Prozesses in der Rechenanlage soll so gering wie möglich sein
- Wartezeit
  - ◆ Möglichst kurze Gesamtzeit, in der der Prozess im Zustand „bereit“ ist
- Antwortzeit
  - ◆ Möglichst kurze Reaktionszeit des Prozesses im interaktiven Betrieb

3.1 First Come, First Served

- Der erste Prozess wird zuerst bearbeitet *(FCFS)*
  - „Wer zuerst kommt ...“
  - Nicht-verdrängend
- Warteschlange zum Zustand „bereit“
  - Prozesse werden hinten eingereiht
  - Prozesse werden vorne entnommen



- Bewertung
  - fair (?)
  - Wartezeiten nicht minimal
  - nicht für Time-Sharing-Betrieb geeignet

3.1 First Come, First Served (2)

- Beispiel zur Betrachtung der Wartezeiten

Prozess 1:	24	Zeiteinheiten
Prozess 2:	3	
Prozess 3:	3	

- Reihenfolge: P1, P2, P3

P1	P2	P3

mittlere Wartezeit: (0+24+27)/3 = 17

- Reihenfolge: P2, P3, P1

P2	P3	P1

mittlere Wartezeit: (6+0+3)/3 = 3

3.2 Shortest Job First

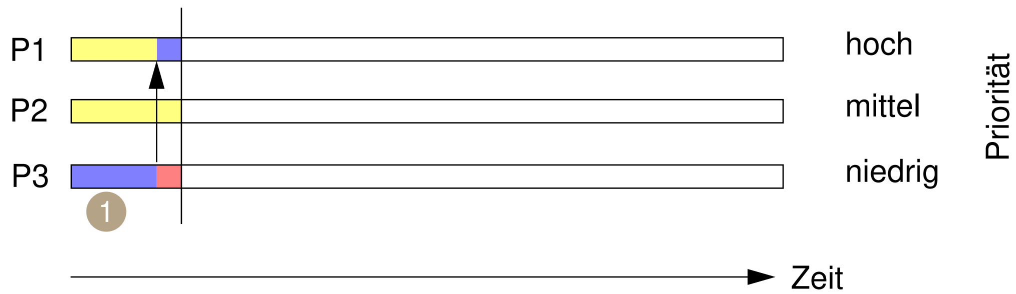
- Kürzester Job wird ausgewählt *(SJF)*
  - Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. Ein-, Ausgabe)
- „bereit“-Warteschlange wird nach Länge der nächsten Rechenphase sortiert
  - Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, exponentielle Glättung)
  - ... Protokollierung der Länge der vorherigen Rechenphase
- SJF optimiert die mittlere Wartezeit
  - Da Länge der Rechenphase in der Regel nicht genau vorhersagbar, nicht ganz optimal.
- Varianten: verdrängend *(PSJF)* und nicht-verdrängend

3.3 Prioritäten

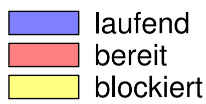
- Prozess mit höchster Priorität wird ausgewählt
  - dynamisch — statisch
    (z.B. SJF: dynamische Vergabe von Prioritäten gemäß Länge der nächsten Rechenphase)
    (z.B. statische Prioritäten in Echtzeitsystemen; Vorhersagbarkeit von Reaktionszeiten)
  - verdrängend — nicht-verdrängend
- Probleme
  - Aushungerung
    Ein Prozess kommt nie zum Zuge, da immer andere mit höherer Priorität vorhanden sind.
  - Prioritätenumkehr *(Priority Inversion)*

3.3 Prioritäten (2)

- Prioritätenumkehr
  - ◆ hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben

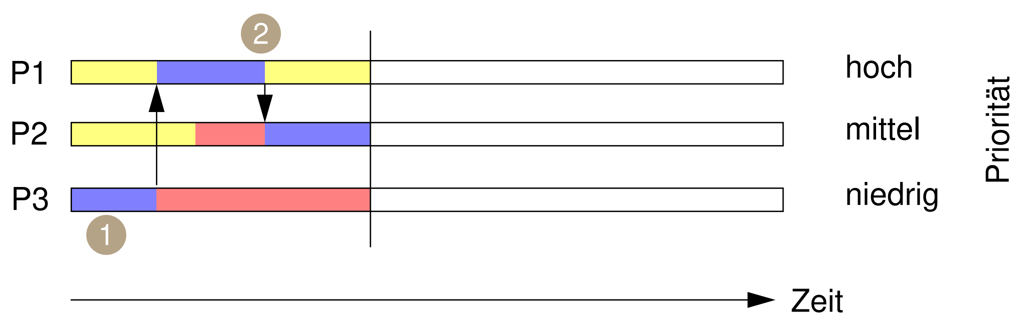


1. P3 belegt ein Betriebsmittel

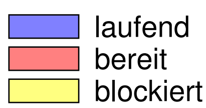


3.3 Prioritäten (2)

- Prioritätenumkehr
  - ◆ hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben

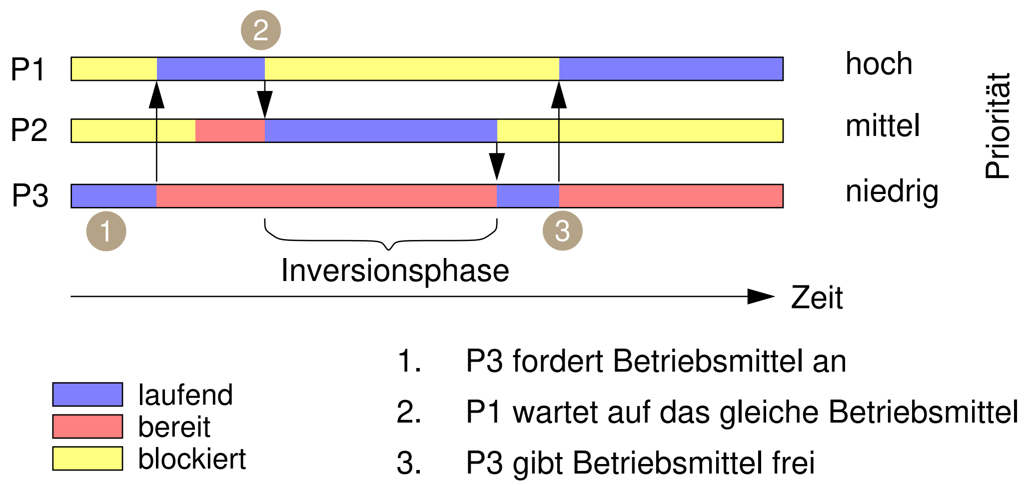


1. P3 belegt ein Betriebsmittel
2. P1 wartet auf das gleiche Betriebsmittel



3.3 Prioritäten (2)

- Prioritätenumkehr
  - hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben



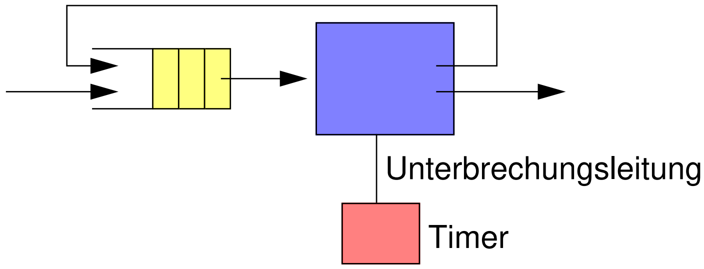
1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei

3.3 Prioritäten (3)

★ Lösungen
- zur Prioritätenumkehr:
  dynamische Anhebung der Priorität für kritische Prozesse
- zur Aushungerung:
  dynamische Anhebung der Priorität für lange wartende Prozesse (Alterung, *Aging*)

3.4 Round-Robin Scheduling

- Zuteilung und Auswahl erfolgt reihum
  - ähnlich FCFS aber mit Verdrängung
  - Zeitquant *(Time Quantum)* oder Zeitscheibe *(Time Slice)* wird zugeteilt
  - geeignet für *Time-Sharing*-Betrieb



  - Wartezeit ist jedoch eventuell relativ lang

3.4 Round-Robin Scheduling (2)

- Beispiel zur Betrachtung der Wartezeiten

$$\left.\begin{array}{ll}\text{Prozess 1:} & 24\\ \text{Prozess 2:} & 3\\ \text{Prozess 3:} & 3\end{array}\right\}\text{Zeiteinheiten}$$

  - Zeitquant ist 4 Zeiteinheiten
  - Reihenfolge in der „bereit“-Warteschlange: P1, P2, P3

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

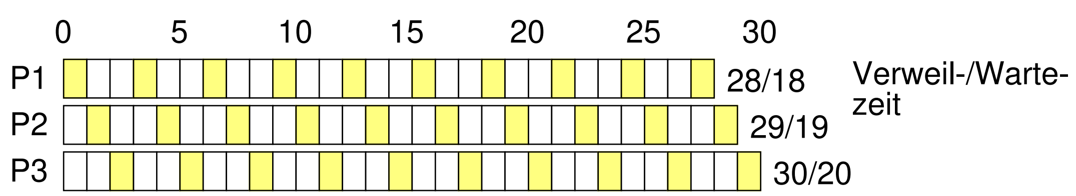
mittlere Wartezeit: (6+4+7)/3 = 5.7

3.4 Round-Robin Scheduling (3)

- Effizienz hängt von der Größe der Zeitscheibe ab
  - ◆ kurze Zeitscheiben: Zeit zum Kontextwechsel wird dominant
  - ◆ lange Zeitscheiben: Round Robin nähert sich FCFS an
- Verweilzeit und Wartezeit hängt ebenfalls von der Zeitscheibengröße ab
  - ◆ Beispiel: 3 Prozesse mit je 10 Zeiteinheiten Rechenbedarf
    - Zeitscheibengröße 1
    - Zeitscheibengröße 10

3.4 Round-Robin Scheduling (4)

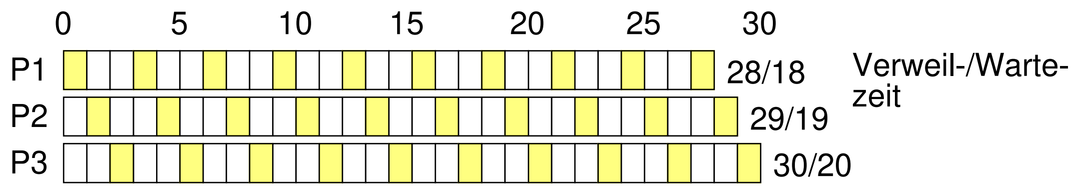
- ◆ Zeitscheibengröße 1:



durchschnittliche Verweilzeit: 29 Zeiteinheiten = (28+29+30)/3
durchschnittliche Wartezeit: 19 Zeiteinheiten = (18+19+20)/3

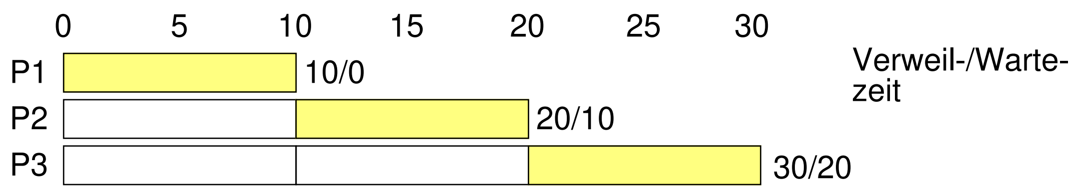
3.4 Round-Robin Scheduling (4)

◆ Zeitscheibengröße 1:



durchschnittliche Verweilzeit: 29 Zeiteinheiten = (28+29+30)/3
durchschnittliche Wartezeit: 19 Zeiteinheiten = (18+19+20)/3

◆ Zeitscheibengröße 10:



durchschnittliche Verweilzeit: 20 Zeiteinheiten = (10+20+30)/3
durchschnittliche Wartezeit: 10 Zeiteinheiten = (0+10+20)/3

3.5 Multilevel-Queue Scheduling

- Verschiedene Schedulingklassen
  - ◆ z.B. Hintergrundprozesse (Batch) und Vordergrundprozesse (interaktive Prozesse)
  - ◆ jede Klasse besitzt ihre eigenen Warteschlangen und verwaltet diese nach einem eigenen Algorithmus
  - ◆ zwischen den Klassen gibt es ebenfalls ein Schedulingalgorithmus z.B. feste Prioritäten (Vordergrundprozesse immer vor Hintergrundprozessen)
- Beispiel: Solaris
  - ◆ Schedulingklassen
    - Systemprozesse
    - Real-Time Prozesse
    - Time-Sharing Prozesse
    - interaktive Prozesse

3.5 Multilevel-Queue Scheduling (2)

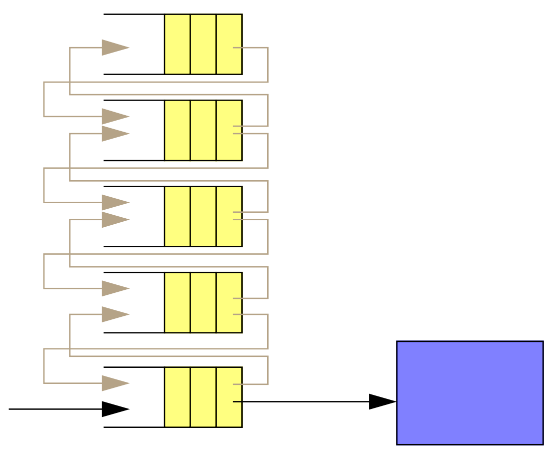
- ◆ Scheduling zwischen den Klassen mit fester Priorität (z.B. Real-Time-Prozesse vor Time-Sharing-Prozessen)
- ◆ In jeder Klasse wird ein eigener Algorithmus benutzt:
  - Systemprozesse: FCFS
  - Real-Time Prozesse: statische Prioritäten
  - Time-Sharing und interaktive Prozesse:
    ausgefeiltes Verfahren zur Sicherung von:
    - kurzen Reaktionszeiten
    - fairer Zeitaufteilung zwischen rechenintensiven und I/O-intensiven Prozessen
    - gewisser Benutzersteuerung

★ Multilevel Feedback Queue Scheduling

3.6 Multilevel-Feedback-Queue Scheduling

■ Mehrere Warteschlangen *(MLFB)*

- ◆ jede Warteschlange mit eigener Behandlung
- ◆ Prozesse können von einer zur anderen Warteschlange transferiert werden



3.6 Multilevel Feedback Queue Scheduling (2)

- Beispiel:
  - mehrere Warteschlangen mit Prioritäten (wie bei Multilevel Queue)
  - Prozesse, die lange rechnen, wandern langsam in Warteschlangen mit niedrigerer Priorität (bevorzugt interaktive Prozesse)
  - Prozesse, die lange warten müssen, wandern langsam wieder in höherpriore Warteschlangen *(Aging)*

3.7 Beispiel: Time Sharing Scheduling in Solaris

- 60 Warteschlangen, Tabellensteuerung

Level	ts_quantum	ts_tqexp	ts_maxwait	ts_lwait	ts_slpret
0	200	0	0	50	50
1	200	0	0	50	50
2	200	0	0	50	50
3	200	0	0	50	50
4	200	0	0	50	50
5	200	0	0	50	50
...					
44	40	34	0	55	55
45	40	35	0	56	56
46	40	36	0	57	57
47	40	37	0	58	58
48	40	38	0	58	58
49	40	39	0	59	58
50	40	40	0	59	58
51	40	41	0	59	58
52	40	42	0	59	58
53	40	43	0	59	58
54	40	44	0	59	58
55	40	45	0	59	58
56	40	46	0	59	58
57	40	47	0	59	58
58	40	48	0	59	58
59	20	49	32000	59	59

3.7 Beispiel: TS Scheduling in Solaris (2)

- Tabelleninhalt
  - kann ausgelesen und gesetzt werden (Auslesen: **`dispadmin -c TS -g`**)
  - **`Level:`** Nummer der Warteschlange
    Hohe Nummer = hohe Priorität
  - **`ts_quantum:`** maximale Zeitscheibe für den Prozess (in Millisek.)
  - **`ts_tqexp:`** Warteschlangennummer, falls der Prozess die Zeitscheibe aufbraucht
  - **`ts_maxwait:`** maximale Zeit für den Prozess in der Warteschlange ohne Bedienung (in Sekunden; Minimum ist eine Sekunde)
  - **`ts_lwait:`** Warteschlangennummer, falls Prozess zulange in dieser Schlange
  - **`ts_slpret:`** Warteschlangennummer für das Wiedereinreihen nach einer blockierenden Aktion

3.7 Beispiel: TS Scheduling in Solaris (3)

- Beispielprozess:
  - 1000ms Rechnen am Stück
  - 5 E/A Operationen mit jeweils Rechenzeiten von 1ms dazwischen

#	Warteschlange	Rechenzeit	Prozesswechsel weil ...
1	59	20	Zeitquant abgelaufen
2	49	40	Zeitquant abgelaufen
3	39	80	Zeitquant abgelaufen
4	29	120	Zeitquant abgelaufen
5	19	160	Zeitquant abgelaufen
6	9	200	Zeitquant abgelaufen
7	0	200	Zeitquant abgelaufen
8	0	180	E/A Operation
9	50	1	E/A Operation
10	58	1	E/A Operation
11	58	1	E/A Operation
12	58	1	E/A Operation

3.7 Beispiel: TS Scheduling in Solaris (4)

- Tabelle gilt nur unter der folgenden Bedingung:
  - ◆ Prozess läuft fast alleine, andernfalls
    - könnte er durch höherpriore Prozesse verdrängt und/oder ausgebremst werden,
    - wird er bei langem Warten in der Priorität wieder angehoben.
- Beispiel:

#	Warteschlange	Rechenzeit	Prozesswechsel weil ...
...			
6	9	200	Zeitquant abgelaufen
7	0	20	Wartezeit von 1s abgelaufen
8	50	40	Zeitquant abgelaufen
9	40	40	Zeitquant abgelaufen
10	30	80	Zeitquant abgelaufen
11	20	120	Zeitquant abgelaufen
...			

3.7 Beispiel: TS Scheduling in Solaris (5)

- Weitere Einflussmöglichkeiten
  - ◆ Anwender und Administratoren können Prioritätenoffsets vergeben
  - ◆ Die Offsets werden auf die Tabellenwerte addiert und ergeben die wirklich verwendete Warteschlange
  - ◆ positive Offsets: Prozess wird bevorzugt
  - ◆ negative Offsets: Prozess wird benachteiligt
  - ◆ Außerdem können obere Schranken angegeben werden
- Systemaufruf
  - ◆ Verändern der eigenen Prozesspriorität

```
int nice( int incr );
```

(positives Inkrement: niedrigere Priorität;
negatives Inkrement: höhere Priorität)

4 Prozesskommunikation

- *Inter-Process-Communication (IPC)*
  - ◆ Mehrere Prozesse bearbeiten eine Aufgabe
    - gleichzeitige Nutzung von zur Verfügung stehender Information durch mehrere Prozesse
    - Verkürzung der Bearbeitungszeit durch Parallelisierung
- Kommunikation durch Nachrichten
  - ◆ Nachrichten werden zwischen Prozessen ausgetauscht
- Kommunikation durch gemeinsamen Speicher
  - ◆ F. Hofmann nennt dies Kooperation (kooperierende Prozesse)

4 Prozesskommunikation (2)

- Klassifikation nachrichtenbasierter Kommunikation
  - ◆ Klassen
    - Kanäle *(Pipes)*
    - Kommunikationsendpunkte *(Sockets, Ports)*
    - Briefkästen, Nachrichtenpuffer *(Queues)*
    - Unterbrechungen *(Signals)*
  - ◆ Übertragungsrichtung
    - unidirektional
    - bidirektional (voll-duplex, halb-duplex)

4 Prozesskommunikation (3)

◆ Übertragungs- und Aufrufeigenschaften

- zuverlässig — unzuverlässig
- gepuffert — ungepuffert
- blockierend — nichtblockierend
- stromorientiert — nachrichtenorientiert — RPC

◆ Adressierung

- implizit: UNIX Pipes
- explizit: Sockets
- globale Adressierung: Sockets, Ports
- Gruppenadressierung: Multicast, Broadcast
- funktionale Adressierung: Dienste

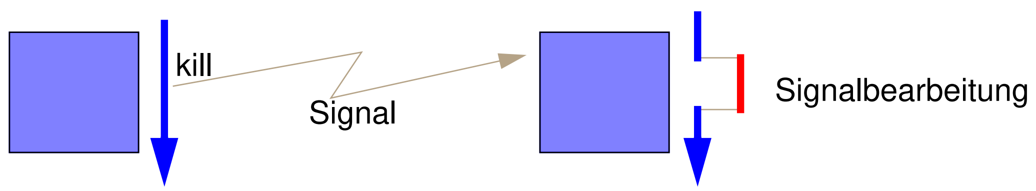
4.1 UNIX Signale

■ Signale sind Unterbrechungen ähnlich denen eines Prozessors

◆ Sender:
- Betriebssystem bei bestimmten Ereignissen
- Prozesse mit Hilfe des Systemaufrufs **`kill`**

◆ Empfänger-Prozess führt eine definierte Signalbehandlung durch

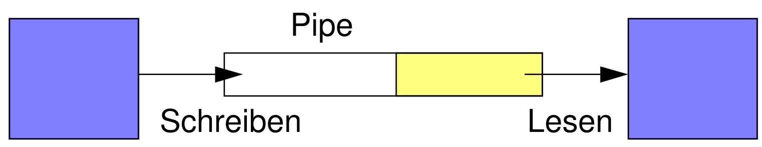
- Ignorieren
- Terminierung des Prozesses
- Aufruf einer Funktion (Signalbearbeitung)
  Nach der Behandlung läuft Prozess an unterbrochener Stelle weiter



◆ kill + Signalbearbeitung = minimale Prozesskomunikation (1 Bit)

4.2 Pipes

- Kanal zwischen zwei Kommunikationspartnern
  - unidirektional
    (heute gleichzeitige Erzeugung zweier Pipes je eine pro Richtung)
  - gepuffert (feste Puffergröße), zuverlässig, stromorientiert



- Operationen: Schreiben und Lesen
  - Ordnung der Zeichen bleibt erhalten (Zeichenstrom)
  - Blockierung bei voller Pipe (Schreiben) und leerer Pipe (Lesen)

4.2 Pipes (2)

- Systemaufruf unter Solaris
  - Öffnen einer Pipe

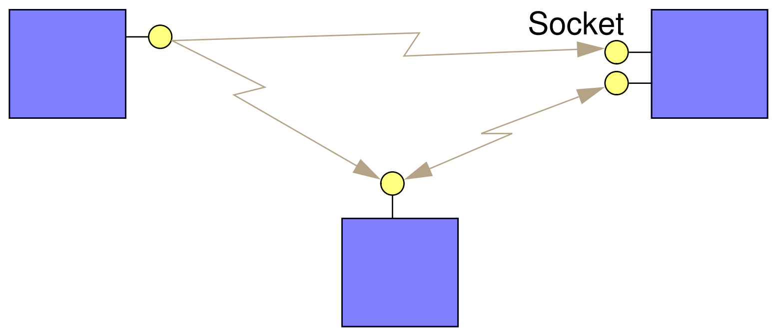
```
int pipe( int fdes[2] );
```

  - Es werden eigentlich zwei Pipes geöffnet
    `fdes[0]` liest aus Pipe 1 und schreibt in Pipe 2
    `fdes[1]` liest aus Pipe 2 und schreibt in Pipe 1
  - Zugriff auf Pipes wie auf eine Datei: **`read`** und **`write`**, **`readv`** und **`writev`**

- Named-Pipes
  - Pipes können auch als Spezialdateien ins Dateisystem gelegt werden.
  - Standardfunktionen zum Lesen und Schreiben können dann verwendet werden.

4.3 Sockets

- **Allgemeine Kommunikationsendpunkte**
  - ◆ bidirektional, gepuffert



  - ◆ Auswahl einer Protokollfamilie
    - z.B. Internet (TCP/IP), UNIX (innerhalb von Prozessen der gleichen Maschine), ISO, Appletalk, DECnet, SNA, ...
    - durch die Protokollfamilie wird gleichzeitig die Adressfamilie festgelegt (Struktur zur Bezeichnung von Protokolladressen)

4.3 Sockets (2)

- ◆ Auswahl eines Sockettyps für Protokolle mit folgenden Eigenschaften:
  - stromorientiert, verbindungsorientiert und gesichert
  - nachrichtenorientiert und ungesichert (Datagramm)
  - nachrichtenorientiert und gesichert
- ◆ Auswahl eines Protokolls der Familie
  - z.B. UDP
- ◆ explizite Adressierung
  - Unicast: genau ein Kommunikationspartner
  - Multicast: eine Gruppe
  - Broadcast: alle möglichen Adressaten
- ◆ Sockets können blockierend und nichtblockierend betrieben werden.

4.3 Sockets (3)

- UNIX-Domain
  - ◆ UNIX-Domain-Sockets verhalten sich wie bidirektionale Pipes.
  - ◆ Anlage als Spezialdatei im Dateisystem möglich
- Internet-Domain
  - ◆ Protokolle:
    - TCP/IP (strom- und verbindungsorientiert, gesichert)
    - UDP/IP (nachrichtenorientiert, verbindungslos, ungesichert)
      – Nachrichten können verloren oder dupliziert werden
      – Reihenfolge kann durcheinander geraten
      – Paketgrenzen bleiben erhalten
      (Datagramm-Protokoll)
  - ◆ Adressen: IP-Adressen und Port-Nummern

4.3 Sockets (4)

- Anlegen von Sockets
  - ◆ Generieren eines Sockets mit (Rückgabewert ist ein Filedeskriptor)

```
int socket( int domain, int type, int proto );
```

  - ◆ Adresszuteilung
    - Sockets werden ohne Adressen generiert
    - Adressenzuteilung erfolgt automatisch oder durch:

```
int bind( int socket, const struct sockaddr *address,
       size_t address_len);
```

4.3 Sockets (5)

- Datagramm-Sockets
  - kein Verbindungsaufbau notwendig
  - Datagramm senden

```
ssize_t sendto( int socket, const void *message,
        size_t length, int flags,
        const struct sockaddr *dest_addr, size_t dest_len);
```

  - Datagramm empfangen

```
ssize_t recvfrom( int socket, void *buffer,
        size_t length, int flags, struct sockaddr *address,
        size_t *address_len);
```


4.3 Sockets (6)

- Stromorientierte Sockets
  - Verbindungsaufbau notwendig
  - *Client* (Benutzer, Benutzerprogramm) will zu einem *Server* (Dienstanbieter) eine Kommunikationsverbindung aufbauen
- Client: Verbindungsaufbau bei stromorientierten Sockets
  - Verbinden des Sockets mit

```
int connect( int socket, const struct sockaddr *address,
        size_t address_len);
```

  - Senden und Empfangen mit **`write`** und **`read`** (**`send`** und **`recv`**)
  - Beenden der Verbindung mit **`close`** (schließt den Socket)

4.3 Sockets (7)

- Server
  - bindet Socket an eine Adresse (sonst nicht zugreifbar)
  - bereitet Socket auf Verbindungsanforderungen vor durch

    ```
    int listen(int s, int backlog);
    ```

  - akzeptiert einzelne Verbindungsanforderungen durch

    ```
    int accept(int s, struct sockaddr *addr, int *addrlen);
    ```

    - gibt einen neuen Socket zurück, der mit dem Client verbunden ist
    - blockiert, falls kein Verbindungswunsch vorhanden
  - liest Daten mit **`read`** und führt den angebotenen Dienst aus
  - schickt das Ergebnis mit **`write`** zurück zum Sender
  - schließt den neuen Socket

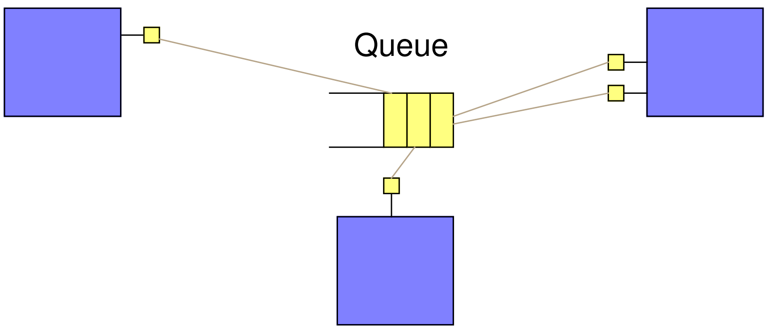
4.4 UNIX Queues

- Nachrichtenpuffer *(Queue, FIFO)*
  - rechnerlokale Adresse *(Key)* dient zur Identifikation eines Puffers
  - prozesslokale Nummer *(MSQID)* ähnlich dem Filedeskriptor
    (wird bei allen Operationen benötigt)
  - Zugriffsrechte wie auf Dateien
  - ungerichtete Kommunikation, gepuffert (einstellbare Größe pro Queue)
  - Nachrichten haben einen Typ (**`long`**-Wert)
  - Operationen zum Senden und Empfangen einer Nachricht
  - blockierend — nichtblockierend
  - alle Nachrichten — nur ein bestimmter Typ

4.4 UNIX Queues (2)

- Systemaufrufe unter Solaris 2.5
  - Erzeugen einer Queue bzw. Holen einer MSQID

```
int msgget( key_t key, int msgflg );
```



  - Alle kommunizierenden Prozesse müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann keine Queue mit gleichem Key erzeugt werden

4.4 UNIX Queues (3)

- Es können Queues ohne Key erzeugt werden (private Queues)
  - Nicht-private Queues sind persistent
  - Sie müssen explizit gelöscht werden

```
int msgctl( int msqid, int cmd, struct msqid_ds *buf );
```

- Systemkommandos zum Behandeln von Queues
  - Listen aktiver Message-Queues

```
ipcs -q
```

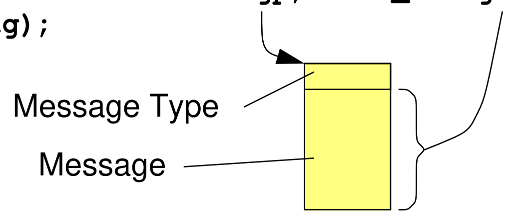
  - Löschen von Queues

```
ipcrm -Q <key>
```

4.4 UNIX Queues (4)

- Operationen auf Queues
  - ◆ Senden einer Nachricht

```
int msgsnd( int msqid, const void *msgp, size_t msgsz,
            int msgflg);
```



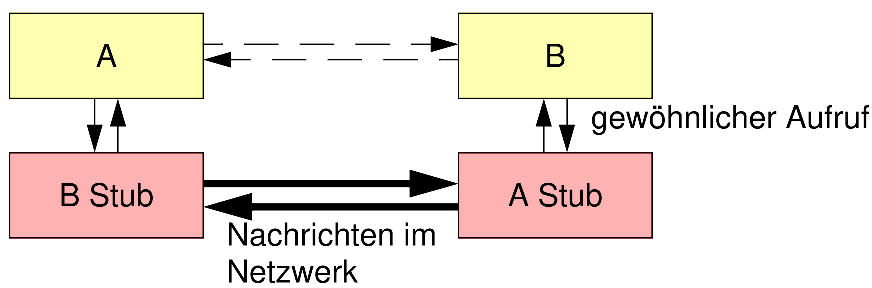
  - ◆ Empfangen einer Nachricht

```
int msgrcv( int msqid, void *msgp, size_t msgsz,
            long msgtype, int msgflg);
```

  - ◆ Zugriffsrechte werden beachtet

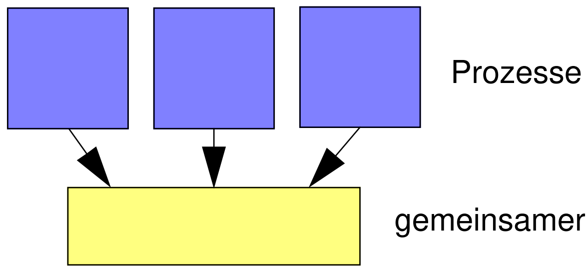
4.5 Fernaufruf *(RPC)*

- Funktionsaufruf über Prozessgrenzen hinweg *(Remote procedure call)*
  - ◆ hoher Abstraktionsgrad
  - ◆ selten wird Fernaufruf direkt vom System angeboten; benötigt Abbildung auf andere Kommunikationsformen z.B. auf Nachrichten
  - ◆ Abbildung auf mehrere Nachrichten
    - Auftragsnachricht transportiert Aufrufabsicht und Parameter.
    - Ergebnisnachricht transportiert Ergebnisse des Aufrufs.



4.6 Gemeinsamer Speicher

- Zwei Prozesse können auf einen gemeinsamen Speicherbereich zugreifen
  - gemeinsame Variablen oder Datenstrukturen



- Einrichten von gemeinsamem Speicher erst im Abschnitt E.5.

5 Aktivitätsträger *(Threads)*

- Mehrere Prozesse zur Strukturierung von Problemlösungen
  - Aufgaben eines Prozesses leichter modellierbar, wenn in mehrere kooperierende Prozesse unterteilt
    z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
    z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
  - Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
    z.B. wissenschaftliches Hochleistungsrechnen (Aerodynamik etc.)
  - Client-Server-Anwendungen unter UNIX: pro Anfrage wird ein neuer Prozess gestartet
    z.B. Webserver

5.1 Prozesse mit gemeinsamem Speicher

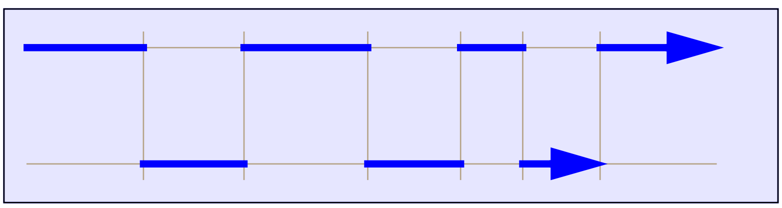
- Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse

- Nachteile
  - viele Betriebsmittel zur Verwaltung eines Prozesses notwendig
    - Dateideskriptoren
    - Speicherabbildung
    - Prozesskontrollblock
  - Prozessumschaltungen sind aufwendig.

- ★ Vorteil
  - In Multiprozessorsystemen sind echt parallele Abläufe möglich.

5.2 Koroutinen

- Einsatz von Koroutinen
  - einige Anwendungen lassen sich mit Hilfe von Koroutinen (auf Benutzerebene) innerhalb eines Prozesses gut realisieren



ein Prozess
zwei Koroutinen

- Nachteile:
  - Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
  - in Multiprozessorsystemen keine parallelen Abläufe möglich
  - Wird eine Koroutine in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert.

5.3 Aktivitätsträger

★ **Lösungsansatz:**
Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(Lightweighted Processes, LWPs)*

- Eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln.
  - Instruktionen
  - Datenbereiche
  - Dateien, Semaphoren etc.
- Jeder Thread repräsentiert eine eigene Aktivität:
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack

5.3 Aktivitätsträger (2)

- Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.
  - Es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
  - Speicherabbildung muss nicht gewechselt werden.
  - Alle Systemressourcen bleiben verfügbar.

■ Ein UNIX-Prozess ist ein Adressraum mit <u>einem</u> Thread

- Solaris: Prozess kann mehrere Threads besitzen

■ Implementierungen von Threads

- User-level Threads
- Kernel-level Threads

5.4 User-Level-Threads

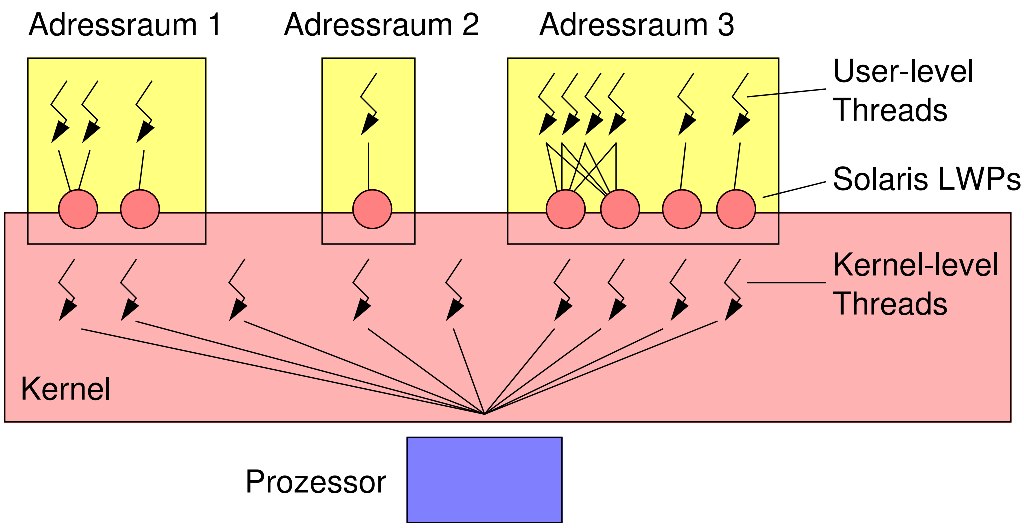
- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - Betriebssystem sieht nur einen Thread
- Vorteile
  - keine Systemaufrufe zum Umschalten erforderlich
  - effiziente Umschaltung
  - Schedulingstrategie in der Hand des Anwenders
- Nachteile
  - Bei blockierenden Systemaufrufen bleiben alle User-Level-Threads stehen.
  - Kein Ausnutzen eines Multiprozessors möglich

5.5 Kernel-Level-Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level-Threads
  - Betriebssystem schaltet Threads um
- Vorteile
  - kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
- Nachteile
  - weniger effizientes Umschalten
  - Fairnessverhalten nötig (zwischen Prozessen mit vielen und solchen mit wenigen Threads)
  - Schedulingstrategie meist vorgegeben

5.6 Beispiel: LWPs und Threads (Solaris)

- Solaris kennt Kernel-, User-Level-Threads und LWPs



Nach Silberschatz, 1994

6 Koordinierung

- Beispiel: Beobachter und Protokollierer
  - ◆ Mittels Induktionsschleife werden Fahrzeuge gezählt. Alle 10min druckt der Protokollierer die im letzten Zeitraum vorbeigekommene Anzahl aus.

```
int cnt= 0;

Observer
    on indication
        cnt= cnt+1;

Logger
    every 10 minutes
        printf( "Count= %d\n", cnt );
        cnt= 0;
```

6 Koordinierung (2)

- Effekte:
  - Fahrzeuge gehen „verloren“
  - Fahrzeuge werden doppelt gezählt
- Ursachen:
  - Befehle in C werden nicht unteilbar (atomar) abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden.
  - In C werden keinesfalls mehrere Anweisungen zusammen atomar abgearbeitet.
  - Prozesswechsel innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen können zu Inkonsistenzen führen.

6 Koordinierung (3)

- Fahrzeuge gehen „verloren“
  - Nach dem Drucken wird der Protokollierer unterbrochen. Beobachter zählt weitere Fahrzeuge. Anzahl wird danach ohne Beachtung vom Protokollierer auf Null gesetzt.

Observer	Logger
	`printf( "Count= %d\n", cnt );`
`cnt= cnt+1;`	
`cnt= cnt+1;`	
	`cnt= 0;`

6 Koordinierung (4)

- Fahrzeuge werden doppelt gezählt:
  - Beobachter will Zähler erhöhen und holt sich diesen dazu in ein Register. Er wird unterbrochen und der Protokollierer setzt Anzahl auf Null. Beobachter erhöht Registerwert und schreibt diesen zurück. Dieser Wert wird erneut vom Protokollierer registriert.

Observer	Logger
`mov DS:$cnt, %r1` `add $1, %r1`	
	`printf( "Count= %d\n", cnt );` `cnt= 0;`
`mov %r1, DS:$cnt`	

```
[cnt= cnt+1;]
```


6 Koordinierung (5)

- Gemeinsame Nutzung von Daten oder Betriebsmitteln
  - kritische Abschnitte:
    - nur einer soll Zugang zu Daten oder Betriebsmitteln haben (gegenseitiger Ausschluss, *Mutual Exclusion*, *Mutex*)
    - kritische Abschnitte erscheinen allen anderen als zeitlich unteilbar
  - Wie kann der gegenseitige Ausschluss in kritischen Abschnitten erzielt werden?

- Koordinierung allgemein:
  - Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern

★ Hinweis:
  - Im Folgenden wird immer von Prozessen die Rede sein. Koordinierung kann/muss selbstverständlich auch zwischen Threads stattfinden.

6.1 Gegenseitiger Ausschluss

- Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten
  - Annahme: Maschinenbefehle sind unteilbar (atomar)
- 1. Versuch

```
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    while( turn == 1 );
    ...
    /* critical sec. */
    ...
    turn= 1;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    while( turn == 0);
    ...
    /* critical sec. */
    ...
    turn= 0;

    ... /* uncritical */
}
```


6.1 Gegenseitiger Ausschluss (2)

- Probleme der Lösung
  - nur alternierendes Betreten des kritischen Abschnitts durch P_0 und P_1 möglich
  - Implementierung ist unvollständig
  - aktives Warten
- Ersetzen von **`turn`** durch zwei Variablen **`ready0`** und **`ready1`**
  - **`ready0`** zeigt an, dass Prozess 0 bereit für den kritischen Abschnitt ist
  - **`ready1`** zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist

6.1 Gegenseitiger Ausschluss (3)

- 2. Versuch

```
bool ready0= FALSE;
bool ready1= FALSE;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    while( ready1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    while( ready0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```


6.1 Gegenseitiger Ausschluss (4)

- Gegenseitiger Ausschluss wird erreicht
  - leicht nachweisbar durch Zustände von **`ready0`** und **`ready1`**

- Probleme der Lösung
  - aktives Warten
  - Verklemmung möglich

6.1 Gegenseitiger Ausschluss (5)

- Betrachtung der nebenläufigen Abfolgen

P_0	P_1
`ready0= TRUE;`	`ready1= TRUE;`
`while( ready1 );` +	`while( ready0 );` +
`<critical>` +	`<critical>` +
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical>` +	`<noncritical>` +
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

\+ = mehrfach, mind. einmal

* = mehrfach oder gar nicht

- ◆ Durchspielen aller möglichen Durchmischungen

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

P_0 P_1 **1**

ausgeführte
Anweisungen

P_0	P_1
`ready0= TRUE;`	`ready1= TRUE;`
`while( ready1 );` +	`while( ready0 );` +
`<critical>` +	`<critical>` +
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical>` +	`<noncritical>` +
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

2

ausgeführte Anweisungen

P_0	P_1
`ready0= TRUE;`	
`while( ready1 ); +`	`ready1= TRUE;`
`<critical> +`	`while( ready0 ); +`
`ready0= FALSE;`	`<critical> +`
`<noncritical> +`	`ready1= FALSE;`
`ready0= TRUE;`	`<noncritical> +`
`...`	`ready1= TRUE;`
	`...`

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

3

ausgeführte Anweisungen

P_0	P_1
`ready0= TRUE;`	
`while( ready1 );`	
`<critical> +`	`ready1= TRUE;`
`ready0= FALSE;`	`while( ready0 ); +`
`<noncritical> +`	`<critical> +`
`ready0= TRUE;`	`ready1= FALSE;`
`...`	`<noncritical> +`
	`ready1= TRUE;`
	`...`

Systemprogrammierung I
D – 79
© 1997-2003, F. J. Hauck, W. Schröder-Preikschat, Inf 4, FAU Erlangen-Nürnberg[D-Proc.fm, 2003-11-26 18.32]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

4

ausgeführte Anweisungen

P_0	P_1
ready0= TRUE;	
while(ready1);	
<critical>	
<critical> *	ready1= TRUE;
ready0= FALSE;	while(ready0); +
<noncritical> +	<critical> +
ready0= TRUE;	ready1= FALSE;
...	<noncritical> +
	ready1= TRUE;
	...

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

5

ausgeführte Anweisungen

P_0	P_1
ready0= TRUE;	
while(ready1);	
<critical>	
	ready1= TRUE;
<critical> *	while(ready0); +
ready0= FALSE;	<critical> +
<noncritical> +	ready1= FALSE;
ready0= TRUE;	<noncritical> +
...	ready1= TRUE;
	...

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

```
    P0                          P1                              6

    ready0= TRUE;
    while( ready1 );
    <critical>
                                                         ausgeführte
                                ready1= TRUE;            Anweisungen
                                while( ready0 );
-----------------------------------------------------------------
    <critical> *                while( ready0 ); *
    ready0= FALSE;              <critical> +
    <noncritical> +             ready1= FALSE;
    ready0= TRUE;               <noncritical> +
    ...                         ready1= TRUE;
                                ...
```


6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

```
    P0                          P1                              7

    ...
    <critical>
                                ready1= TRUE;               aktives
                                while( ready0 );            Warten
                                while( ready0 );
-----------------------------------------------------------------
    <critical> *                while( ready0 ); *
    ready0= FALSE;              <critical> +
    <noncritical> +             ready1= FALSE;
    ready0= TRUE;               <noncritical> +
    ...                         ready1= TRUE;
                                ...
```

6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

8

```
P0                          P1
...
                            ready1= TRUE;
                            while( ready0 );
                            while( ready0 );
ready0= FALSE;
--------------------------------------------------------
<noncritical> +             while( ready0 ); *
ready0= TRUE;               <critical> +
...                         ready1= FALSE;
                            <noncritical> +
                            ready1= TRUE;
                            ...
```


6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

9

```
P0                          P1
                            ready1= TRUE;
                            while( ready0 );
                            while( ready0 );
ready0= FALSE;
                            while( ready0 );
--------------------------------------------------------
<noncritical> +             <critical> +
ready0= TRUE;               ready1= FALSE;
...                         <noncritical> +
                            ready1= TRUE;
                            ...
```

6.1 Gegenseitiger Ausschluss (7)

- Verklemmung

1

ausgeführte Anweisungen

```
P0                          P1
----------------------------------------------------
ready0= TRUE;               ready1= TRUE;
while( ready1 ); +          while( ready0 ); +
<critical> +                <critical> +
ready0= FALSE;              ready1= FALSE;
<noncritical> +             <noncritical> +
ready0= TRUE;               ready1= TRUE;
...                         ...
```


6.1 Gegenseitiger Ausschluss (7)

- Verklemmung

2

ausgeführte Anweisungen

```
P0                          P1
ready0= TRUE;
----------------------------------------------------
while( ready1 ); +          ready1= TRUE;
<critical> +                while( ready0 ); +
ready0= FALSE;              <critical> +
<noncritical> +             ready1= FALSE;
ready0= TRUE;               <noncritical> +
...                         ready1= TRUE;
                            ...
```

Systemprogrammierung I
D – 80
© 1997-2003, F. J. Hauck, W. Schröder-Preikschat, Inf 4, FAU Erlangen-Nürnberg[D-Proc.fm, 2003-11-26 18.32]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

6.1 Gegenseitiger Ausschluss (7)

- Verklemmung

3

ausgeführte Anweisungen

P_0	P_1
`ready0= TRUE;`	
	`ready1= TRUE;`
`while( ready1 );` +	`while( ready0 );` +
`<critical>` +	`<critical>` +
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical>` +	`<noncritical>` +
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

6.1 Gegenseitiger Ausschluss (7)

- Verklemmung

4

ausgeführte Anweisungen

P_0	P_1
`ready0= TRUE;`	
	`ready1= TRUE;`
`while( ready1 );`	
`while( ready1 );` *	`while( ready0 );` +
`<critical>` +	`<critical>` +
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical>` +	`<noncritical>` +
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

6.1 Gegenseitiger Ausschluss (7)

- Verklemmung *(Lifelock)*

```
P0                          P1                              5

ready0= TRUE;                                        beide Prozesse
                            ready1= TRUE;            warten aktiv
while( ready1 );
                            while( ready0 );
------------------------------------------------------------
while( ready1 ); *          while( ready0 ); *
<critical> +                <critical> +
ready0= FALSE;              ready1= FALSE;
<noncritical> +             <noncritical> +
ready0= TRUE;               ready1= TRUE;
...                         ...
```


6.1 Gegenseitiger Ausschluss (8)

- 3. Versuch (Algorithmus von Peterson, 1981)

```
bool ready0= FALSE;
bool ready1= FALSE;
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    turn= 1;
    while( ready1 &&
           turn == 1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    turn= 0;
    while( ready0 &&
           turn == 0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```

6.1 Gegenseitiger Ausschluss (9)

- Algorithmus implementiert gegenseitigen Ausschluss
  - vollständige und sichere Implementierung
  - **`turn`** entscheidet für den kritischen Fall von Versuch 2, welcher Prozess nun wirklich den kritischen Abschnitt betreten darf
  - in allen anderen Fällen ist **`turn`** unbedeutend
- ▲ Problem der Lösung
  - aktives Warten
- ★ Algorithmus auch für mehrere Prozesse erweiterbar
  - Lösung ist relativ aufwendig

6.2 Spezielle Maschinenbefehle

- Spezielle Maschinenbefehle können die Programmierung kritischer Abschnitte unterstützen und vereinfachen
  - *Test-and-Set* Instruktion
  - *Swap* Instruktion
- Test-and-set
  - Maschinenbefehl mit folgender Wirkung

```
bool test_and_set( bool *plock )
{
    bool tmp= *plock;
    *plock= TRUE;
    return tmp;
}
```

  - Ausführung ist atomar

6.2 Spezielle Maschinenbefehle (2)

- Kritische Abschnitte mit Test-and-Set Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

6.2 Spezielle Maschinenbefehle (3)

Swap

- Maschinenbefehl mit folgender Wirkung

```
void swap( bool *ptr1, bool *ptr2)
{
    bool tmp= *ptr1;
    *ptr1= *ptr2;
    *ptr2= tmp;
}
```

- Ausführung ist atomar

6.2 Spezielle Maschinenbefehle (4)

- Kritische Abschnitte mit Swap-Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;
  ... /* uncritical */
}
```

Prozess 1

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;
  ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

6.3 Kritik an den bisherigen Verfahren

★ Spinlock
- ◆ bisherige Verfahren werden auch Spinlocks genannt
- ◆ aktives Warten

▲ Problem des aktiven Wartens
- ◆ Verbrauch von Rechenzeit ohne Nutzen
- ◆ Behinderung „nützlicher“ Prozesse
- ◆ Abhängigkeit von der Schedulingstrategie
  - nicht anwendbar bei nicht-verdrängenden Strategien
  - schlechte Effizienz bei langen Zeitscheiben

- Spinlocks kommen heute fast ausschließlich in Multiprozessorsystemen zum Einsatz
  - ◆ bei kurzen kritischen Abschnitten effizient
  - ◆ Koordinierung zwischen Prozessen von mehreren Prozessoren

6.4 Sperrung von Unterbrechungen

- Sperrung der Systemunterbrechungen im Betriebssystems

Prozess 0
```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

Prozess 1
```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

- ◆ nur für kurze Abschnitte geeignet
  - sonst Datenverluste möglich
- ◆ nur innerhalb des Betriebssystems möglich
  - privilegierter Modus nötig
- ◆ nur für Monoprozessoren anwendbar
  - bei Multiprozessoren arbeiten andere Prozesse echt parallel

6.5 Semaphor

- Ein Semaphor (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach *Dijkstra*)
  - ◆ P-Operation *(proberen; passeren; wait; down)*
    - wartet bis Zugang frei

```
void P( int *s )
{
    while( *s <= 0 );
    *s= *s-1;
}
```

atomare Funktion

  - ◆ V-Operation *(verhogen; vrijgeven; signal; up)*
    - macht Zugang für anderen Prozess frei

```
void V( int *s )
{
    *s= *s+1;
}
```

atomare Funktion

6.5 Semaphor (2)

- Implementierung kritischer Abschnitte mit einem Semaphor

```
int lock= 1;
```

Prozess 0

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

Prozess 1

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

▲ Problem:

◆ Implementierung von P und V

6.5 Semaphor (3)

- Implementierung im Betriebssystem (Monoprozessor)

P-Operation

Unterbrechungen sperren	
`*s <= 0` — ja	nein
`ok = FALSE;`	`*s = *s-1` `ok = TRUE;`
laufenden Prozess in Zustand *blockiert* versetzen und in Warteschlange aufnehmen	
Prozessumschalter ansteuern und Unterbrechungen freigeben	Unterbrechungen freigeben
`until: ok == TRUE`	

ok ist eine prozesslokale Variable

◆ jeder Semaphor besitzt Warteschlange, die blockierte Prozesse aufnimmt

6.5 Semaphor (4)

V-Operation

Unterbrechungen sperren
`*s = *s+1`
alle Prozess aus der Warteschlange in den Zustand *bereit* versetzen
Unterbrechungen freigeben
Prozessumschalter ansteuern

- ◆ Prozesse probieren immer wieder, die P-Operation erfolgreich abzuschließen
- ◆ Schedulingstrategie entscheidet über Reihenfolge und Fairness
  - leichte Ineffizienz durch Aufwecken aller Prozesse
  - mit Einbezug der Schedulingstrategie effizientere Implementierungen möglich

6.5 Semaphor (5)

★ Vorteile einer Semaphor-Implementierung im Betriebssystem

- ◆ Einbeziehen des Schedulers in die Semaphor-Operationen
- ◆ kein aktives Warten; Ausnutzen der Blockierzeit durch andere Prozesse

■ Implementierung einer Synchronisierung

- ◆ zwei Prozesse P_1 und P_2
- ◆ Anweisung S_1 in P_1 soll vor Anweisung S_2 in P_2 stattfinden

```
int lock= 0;
```

Prozess 1

```
...
S1;
V( &lock );
...
```

Prozess 2

```
...
P( &lock );
S2;
...
```

★ Zählende Semaphore

6.5 Semaphor (6)

- Abstrakte Beschreibung von zählenden Semaphoren (PV System)
  - für jede Operation wird eine Bedingung angegeben
    - falls Bedingung nicht erfüllt, wird die Operation blockiert
  - für den Fall, dass die Bedingung erfüllt wird, wird eine Anweisung definiert, die ausgeführt wird
- Beispiel: zählende Semaphore

Operation	Bedingung	Anweisung
P(S)	$S > 0$	$S := S - 1$
V(S)	TRUE	$S := S + 1$

7 Klassische Koordinierungsprobleme

- Reihe von bedeutenden Koordinierungsproblemen
  - Gegenseitiger Ausschluss *(Mutual exclusion)*
    - nur ein Prozess darf bestimmte Anweisungen ausführen
  - Puffer fester Größe *(Bounded buffers)*
    - Blockieren der lesenden und schreibenden Prozesse, falls Puffer leer oder voll
  - Leser-Schreiber-Problem *(Reader-writer problem)*
    - Leser können nebenläufig arbeiten; Schreiber darf nur alleine zugreifen
  - Philosophenproblem *(Dining-philosopher problem)*
    - im Kreis sitzende Philosophen benötigen das Besteck der Nachbarn zum Essen
  - Schlafende Friseure *(Sleeping-barber problem)*
    - Friseure schlafen solange keine Kunden da sind

7.1 Gegenseitiger Ausschluss

- Semaphor
  - ◆ eigentlich reicht ein Semaphor mit zwei Zuständen: binärer Semaphor

```
void P( int *s )
{
    while( *s == 0 );
    *s= 0;
}
```

atomare Funktion

```
void V( int *s )
{
    *s= 1;
}
```

atomare Funktion

  - ◆ zum Teil effizienter implementierbar

7.1 Gegenseitiger Ausschluss (2)

- Abstrakte Beschreibung: binäre Semaphore

Operation	Bedingung	Anweisung
P(S)	$S \neq 0$	S := 0
V(S)	TRUE	S := 1

7.1 Gegenseitiger Ausschluss (3)

- Problem der Klammerung kritischer Abschnitte
  - Programmierer müssen Konvention der Klammerung einhalten
  - Fehler bei Klammerung sind fatal

```
P( &lock );
... /* critical sec. */
P( &lock );
```

führt zu Verklemmung (Deadlock)

```
V( &lock );
... /* critical sec. */
V( &lock );
```

führt zu unerwünschter Nebenläufigkeit

7.1 Gegenseitiger Ausschluss (3)

- Automatische Klammerung wünschenswert
  - Beispiel: Java

```
synchronized( lock ) {
... /* critical sec. */
}
```

7.2 Bounded Buffers

- Puffer fester Größe
  - mehrere Prozesse lesen und beschreiben den Puffer
  - beispielsweise Erzeuger und Verbraucher (Erzeuger-Verbraucher-Problem)
    (z.B. Erzeuger liest einen Katalog; Verbraucher zählt Zeilen; Gesamtanwendung zählt Einträge in einem Katalog)
  - UNIX-Pipe ist solch ein Puffer
- Problem
  - Koordinierung von Leser und Schreiber
    - gegenseitiger Ausschluss beim Pufferzugriff
    - Blockierung des Lesers bei leerem Puffer
    - Blockierung des Schreibers bei vollem Puffer

7.2 Bounded Buffers (2)

- Implementierung mit zählenden Semaphoren
  - zwei Funktionen zum Zugriff auf den Puffer
    - **`put`** stellt Zeichen in den Puffer
    - **`get`** liest ein Zeichen vom Puffer
  - Puffer wird durch ein Feld implementiert, das als Ringpuffer wirkt
    - zwei Integer-Variablen enthalten Feldindizes auf den Anfang und das Ende des Ringpuffers
  - ein Semaphor für den gegenseitigen Ausschluss
  - je einen Semaphor für das Blockieren an den Bedingungen „Puffer voll“ und „Puffer leer“
    - Semaphor **`full`** zählt wieviele Zeichen noch in den Puffer passen
    - Semaphor **`empty`** zählt wieviele Zeichen im Puffer sind

7.2 Bounded Buffers (3)

```
char buffer[N];
int inslot= 0, outslot= 0;
semaphor mutex= 1, empty= 0, full= N;
```

```
void put( char c )
{
    P( &full );
    P( &mutex );
    buffer[inslot]= c;
    if( ++inslot >= N )
        inslot= 0;
    V( &mutex );
    V( &empty );
}
```

```
char get( void )
{
    char c;

    P( &empty );
    P( &mutex );
    c= buffer[outslot];
    if( ++outslot >= N )
        outslot= 0;
    V( &mutex );
    V( &full );
    return c;
}
```


7.3 Erstes Leser-Schreiber-Problem

- Lesende und schreibende Prozesse
  - Leser können nebenläufig zugreifen (Leser ändern keine Daten)
  - Schreiber können nur exklusiv zugreifen (Daten sonst inkonsistent)
- Erstes Leser-Schreiber-Problem (nach *Courtois* et.al. 1971)
  - Kein Leser soll warten müssen, es sei denn ein Schreiber ist gerade aktiv
- Realisierung mit zählenden (binären) Semaphoren
  - Semaphor für gegenseitigen Ausschluss von Schreibern untereinander und von Schreiber gegen Leser: **`write`**
  - Zählen der nebenläufig tätigen Leser: Variable **`readcount`**
  - Semaphor für gegenseitigen Ausschluss beim Zugriff auf **`readcount`**: **`mutex`**

7.3 Erstes Leser-Schreiber-Problem (2)

```
semaphore mutex= 1, write= 1;
int readcount= 0;
```

Leser

```
...
P( &mutex );
if( ++readcount == 1 )
    P( &write );
V( &mutex );

... /* reading */

P( &mutex );
if( --readcount == 0 )
    V( &write );
V( &mutex );
...
```

Schreiber

```
...
P( &write );

... /* writing */

V( &write );
...
```


7.3 Erstes Leser-Schreiber-Problem (3)

- Vereinfachung der Implementierung durch spezielle Semaphore?
  - ◆ PV-Chunk Semaphore:
    führen quasi mehrere P- oder V-Operationen atomar aus
    - zweiter Parameter gibt Anzahl an
- Abstrakte Beschreibung für PV-Chunk Semaphore:

Operation	Bedingung	Anweisung
P(S, k)	$S \geq k$	$S := S - k$
V(S, k)	TRUE	$S := S + k$

7.3 Erstes Leser-Schreiber-Problem (4)

- Implementierung mit PV-Chunk:
  - Annahme: es gibt maximal **N** Leser

```
PV_chunk_semaphore mutex= N;
```

Leser

```
...
Pc( &mutex, 1 );

... /* reading */

Vc( &mutex, 1 );
...
```

Schreiber

```
...
Pc( &mutex, N );

... /* writing */

Vc( &mutex, N );
...
```


7.4 Zweites Leser-Schreiber-Problem

- Wie das erste Problem aber: (nach Courtois et.al., 1971)
  - Schreiboperationen sollen so schnell wie möglich durchgeführt werden
- Implementierung mit zählenden Semaphoren
  - Zählen der nebenläufig tätigen Leser: Variable **readcount**
  - Zählen der anstehenden Schreiber: Variable **writecount**
  - Semaphor für gegenseitigen Ausschluss beim Zugriff auf **readcount**: **mutexR**
  - Semaphor für gegenseitigen Ausschluss beim Zugriff auf **writecount**: **mutexW**
  - Semaphor für gegenseitigen Ausschluss von Schreibern untereinander und von Schreibern gegen Leser: **write**
  - Semaphor für den Ausschluss von Lesern, falls Schreiber vorhanden: **read**
  - Semaphor zum Klammern des Leservorspanns: **mutex**

7.4 Zweites Leser-Schreiber-Problem (2)

```
semaphore mutexR= 1, mutexW= 1, mutex= 1;
semaphore write= 1, read= 1;
int readcount= 0, writecount= 0;
```

Bitte nicht versuchen, dies zu verstehen!!

Leser

```
...
P( &mutex ); P( &read );
  P( &mutexR );
  if( ++readcount == 1 )
    P( &write );
  V( &mutexR );
V( &read ); V( &mutex );

... /* reading */

P( &mutexR );
if( --readcount == 0 )
    V( &write );
V( &mutexR );
...
```

Schreiber

```
...
P( &mutexW );
if( ++writecount == 1 )
  P( &read );
V( &mutexW );
P( &write );

... /* writing */

V( &write );
P( &mutexW );
if( --writecount == 0 )
    V( &read );
V( &mutexW );
...
```


7.4 Zweites Leser-Schreiber-Problem (3)

- Vereinfachung der Implementierung durch spezielle Semaphore?
  - Up-Down–Semaphore:
    - zwei Operationen *up* und *down*, die den Semaphor hoch- und runterzählen
    - Nichtblockierungsbedingung für beide Operationen, definiert auf einer Menge von Semaphoren

7.4 Zweites Leser-Schreiber-Problem (4)

- Abstrakte Beschreibung für Up-down–Semaphore

Operation	Bedingung	Anweisung
up(S, { S_i })	$\sum_i S_i \geq 0$	S := S + 1
down(S, { S_i })	$\sum_i S_i \geq 0$	S := S – 1

7.4 Zweites Leser-Schreiber-Problem (5)

- Implementierung mit Up-Down–Semaphoren:

```
up_down_semaphor mutexw= 0, reader= 0, writer= 0;
```

Leser

```
...
down( &reader, 1, &writer );

... /* reading */

up( &reader, 0 );
...
```

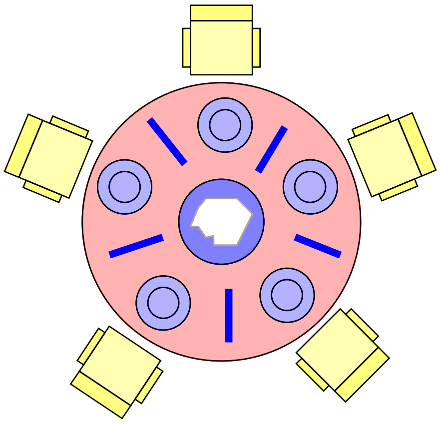
Schreiber

```
...
down( &writer, 0 );
down( &mutexw,
  2, &mutexw,&reader );

... /* writing */

up( &mutexw, 0 );
up( &writer, 0 );
...
```

- ◆ Zähler für Leser: **`reader`** (zählt negativ)
- ◆ Zähler für anstehende Schreiber: **`writer`** (zählt negativ)
- ◆ Semaphor für gegenseitigen Ausschluss der Schreiber: **`mutexw`**

7.5 Philosophenproblem

- Fünf Philosophen am runden Tisch



- Philosophen denken oder essen
  "The life of a philosopher consists of an alternation of thinking and eating." (Dijkstra, 1971)
- zum Essen benötigen sie zwei Gabeln, die jeweils zwischen zwei benachbarten Philosophen abgelegt sind

▲ Problem

- Gleichzeitiges Belegen mehrerer Betriebsmittel (hier Gabeln)
- Verklemmung und Aushungerung

7.5 Philosophenproblem (2)

- Naive Implementierung
  - eine Semaphor pro Gabel

```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

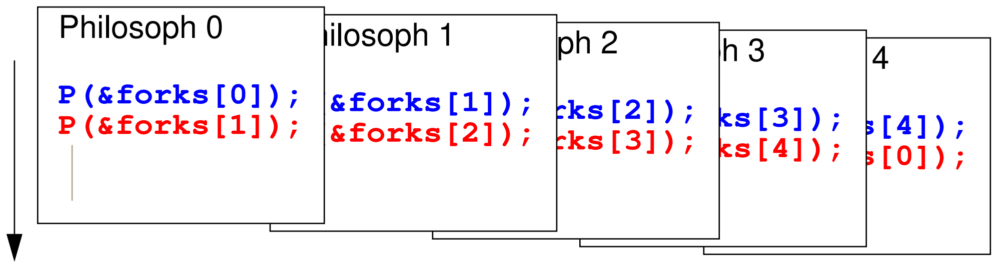
Philosoph i, i ∈ [0,4]

```
while( 1 ) {
    ... /* think */

    P( &forks[i] );
    P( &forks[(i+1)%5] );

    ... /* eat */

    V( &forks[i] );
    V( &forks[(i+1)%5] );
}
```

7.5 Philosophenproblem (3)

■ Problem der Verklemmung

- alle Philosophen nehmen gleichzeitig die linke Gabel auf und versuchen dann die rechte Gabel aufzunehmen



- System ist verklemmt
  - Philosophen warten alle auf ihre Nachbarn

7.5 Philosophenproblem (4)

■ Lösung 1: gleichzeitiges Aufnehmen der Gabeln

- Implementierung mit binären oder zählenden Semaphoren ist nicht trivial
- Zusatzvariablen erforderlich
- unübersichtliche Lösung

★ Einsatz von speziellen Semaphoren: PV-multiple–Semaphore

- gleichzeitiges und atomares Belegen mehrerer Semaphoren
- Abstrakte Beschreibung:

Operation	Bedingung	Anweisung
$P(\{ S_i \})$	$\forall i, S_i > 0$	$\forall i, S_i = S_i - 1$
$V(\{ S_i \})$	TRUE	$\forall i, S_i = S_i + 1$

7.5 Philosophenproblem (5)

◆ Implementierung mit PV-multiple–Semaphoren

```
PV_mult_semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, $i \in [0,4]$

```
while( 1 ) {
    ... /* think */

    Pm( 2, &forks[i], &forks[(i+1)%5] );

    ... /* eat */

    Vm( 2, &forks[i], &forks[(i+1)%5] );
}
```


7.5 Philosophenproblem (6)

- Lösung 2: einer der Philosophen muss erst die andere Gabel aufnehmen

```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, $i \in [0,3]$

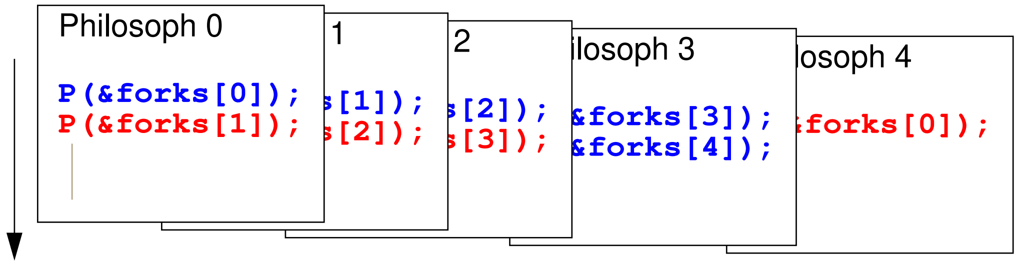
```
while( 1 ) {
    ... /* think */

    P( &forks[i] );
    P( &forks[(i+1)%5] );

    ... /* eat */

    V( &forks[i] );
    V( &forks[(i+1)%5] );
}
```

Philosoph 4

```
while( 1 ) {
    ... /* think */

    P( &forks[0] );
    P( &forks[4] );

    ... /* eat */

    V( &forks[0] );
    V( &forks[4] );
}
```

7.5 Philosophenproblem (7)

- ◆ Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- ◆ System verklemmt sich nicht

7.6 Schlafende Friseure

- Friseurladen mit N freien Wartestühlen
  - ◆ Friseure schlafen solange kein Kunde da ist
  - ◆ eintretende Kunden warten bis ein Friseur frei ist; gegebenenfalls wird einer der Friseure von einem Kunden aufgeweckt
  - ◆ sind keine Wartestühle mehr frei, verlassen die Kunden den Laden
- Problem:
  - ◆ Mehrere Bearbeitungsstationen sollen exklusive Bearbeitungen durchführen
- Implementierung mit zählenden Semaphoren
  - ◆ Semaphore zum Schutz der Variablen zum Zählen der Kunden: **`mutex`**
  - ◆ Semaphore zum Zählen der Friseure: **`barbers`**
  - ◆ Semaphore zum Zählen der Kunden: **`customers`**

7.6 Schlafende Friseure (2)

- Implementierung mit zählenden Semaphoren (PV System)

```
semaphor customers= 0, barbers= 0, mutex= 1;
int waiting= 0;
```

Barber

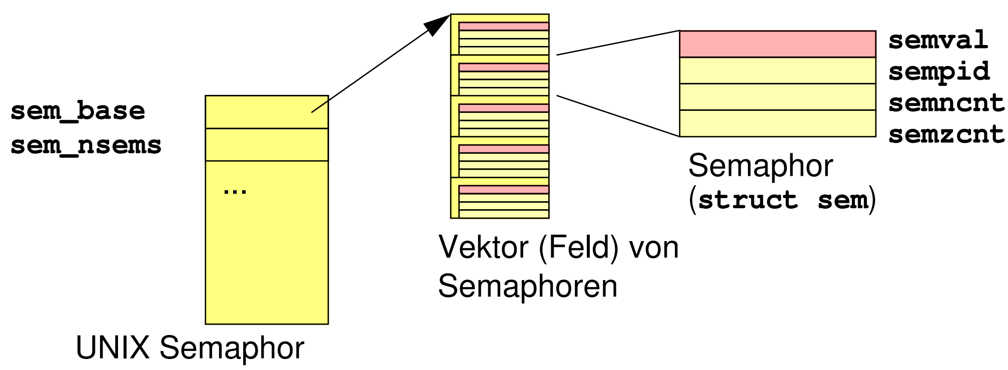
```
while( 1 ) {
    P( &customers );
    P( &mutex );
    waiting--;
    V( &barbers );
    V( &mutex );

    ... /* cut hair */
}
```

Customer

```
P( &mutex );
if( waiting < N ) {
    waiting++
    V( &customers );
    V( &mutex );
    P( &barbers );

    ... /* get hair cut */
}
else {
    V( &mutex );
}
```


8 UNIX-Semaphor

- Ein UNIX-Semaphor entspricht einem Vektor von Einzelsemaphoren (erweitertes Vektoradditionssystem)



◆ Gleichzeitige und atomare Operationen auf mehreren Semaphoren im Vektor möglich

8.1 Erzeugen einer UNIX-Semaphore

- UNIX-Semphore haben systemweit eindeutige Identifikation (Key)
  - ◆ Erzeugen und Aufnehmen der Verbindung zu einer Semaphore

```
int semget( key_t key, int nsems, int semflg );
```

Anzahl d. Semaphoren
im Vektor

Identifikation
– neue für Erzeugung
– bestehende für Verbindungsaufnahme

Zugriffssrechte;
Erzeugung oder
Verbindungsaufnahme

  - ◆ Ergebnis ist eine Semaphore ID ähnlich wie ein Filedescriptor
    - Semaphore ID muss bei allen Operationen verwendet werden
  - ◆ Zugriffsrechte: Lesen, Verändern
    - einstellbar für Besitzer, Gruppe und alle anderen (ähnlich wie bei Dateien)

8.1 Erzeugen einer UNIX-Semaphore (2)

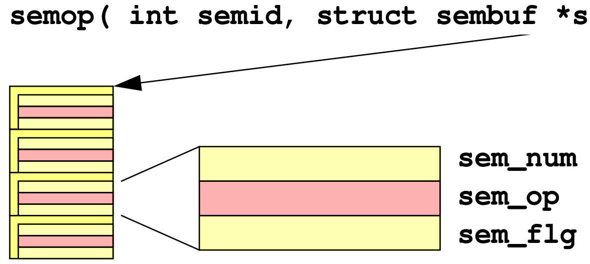
- Verwendung des Keys
  - ◆ Alle Prozesse, die auf die Semaphore zugreifen wollen, müssen den Key kennen
  - ◆ Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - ◆ Ist ein Key bereits vergeben, kann keine Semaphore mit gleichem Key erzeugt werden
  - ◆ Ist ein Key bekannt, kann auf die Semaphore zugegriffen werden
    - gesetzte Zugriffsberechtigungen werden allerdings beachtet
  - ◆ Private Semaphoren (ohne Key) können erzeugt werden
- Semaphore sind persistent
  - ◆ Explizites Löschen notwendig

```
ipcrm -S <key>
```

8.2 Operationen auf UNIX-Semphoren

■ Operationen auf mehreren der Semaphoren im Vektor

```
int semop( int semid, struct sembuf *sops, size_t nsops );
```



Anzahl der Einzel-operationen

◆ Operationen

- **sem_num**: Nummer des Semaphor im Vektor
- **sem_op < 0**: ähnlich P-Operation – Herunterzählen des Semaphor (blockierend oder mit Fehlerstatus, je nach **sem_flg**)
- **sem_op > 0**: ähnlich V-Operation – Hochzählen des Semaphore
- **sem_op == 0**: Test auf 0 (blockierend oder mit Fehlerstatus, je nach **sem_flg**)

8.2 Operationen auf UNIX-Semaphoren (2)

■ Kontrolloperationen

```
int semctl( int semid, int semnum, int cmd,
            [ union semun arg ] );
```

◆ explizites Setzen von Werten (einen, alle)

◆ Abfragen von Werten (einen, alle)

◆ Abfragen von Zusatzinformationen

- welcher Prozess hat letzte Operation erfolgreich durchgeführt
- wann wurde letzte Operation durchgeführt
- Zugriffsrechte
- Anzahl der blockierten Prozesse

◆ Löschen des Semaphor

8.3 Beispiel: Philosophenproblem

- Ein UNIX-Semaphor mit fünf Elementen (entsprechen Gabeln)
  - Deklarationen

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

int i;                          /* number of philosopher */
int j;
int semid;                        /* semaphor ID */
struct sembuf pbuf[2], vbuf[2];  /* operation buffer */

union semun {                   /* UNION for semctl */
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;

...
```


8.3 Beispiel: Philosophenproblem (2)

- Erzeuge Semaphor

```
...

semid= semget( IPC_PRIVATE, 5, IPC_CREAT|SEM_A|SEM_R );
if( semid < 0 ) { ... /* error */ }

for( j= 0; j < 5; j++ ) {    /* set all values to 1 */
    arg.val= 1;
    if( semctl( semid, j, SETVAL, arg ) < 0 ) {
        ... /* error */
    }
}

...
```

8.3 Beispiel: Philosophenproblem (3)

- Erzeugen der Prozesse

```
...

for( i=0; i<=3; i++ ) {       /* start children i= 0..3; */
    pid_t pid= fork();

    if( pid < (pid_t)0 ) { ... /* error */ }
    if( pid ==(pid_t)0 ) {
        /* child */

        break;
    }
}                             /* parent: i= 4; */

...
```


8.3 Beispiel: Philosophenproblem (4)

- Initialisierungen

```
...     /* we are philosopher i */

/* initialize buffer for P operation */

pbuf[0].sem_num= i; pbuf[1].sem_num= (i+1)%5;
pbuf[0].sem_op= pbuf[1].sem_op= -1;
pbuf[0].sem_flg= pbuf[1].sem_flg= 0;

/* initialize buffer for V operation */

vbuf[0].sem_num= i; vbuf[1].sem_num= (i+1)%5;
vbuf[0].sem_op= vbuf[1].sem_op= 1;
vbuf[0].sem_flg= vbuf[1].sem_flg= 0;

...
```

8.3 Beispiel: Philosophenproblem (5)

- Philosoph

```
...

while( 1 ) {
    ...      /* thinking */

    if( semop( semid, pbuf, 2 ) < 0 ) { ... /* error */ }

    ...      /* eating */

    if( semop( semid, vbuf, 2 ) < 0 ) { ... /* error */ }
}
```


9 Zusammenfassung

- Programmiermodell: Prozess
  - Zerlegung von Anwendungen in Prozesse oder Threads
  - Ausnutzen von Wartezeiten; Time sharing–Betrieb
  - Prozess hat verschiedene Zustände: laufend, bereit, blockiert etc.
- Auswahlstrategien für Prozesse
  - FCFS, SJF, PSJF, RR, MLFB
- Prozesskommunikation
  - Pipes, Queues, Signals, Sockets, Shared memory, RPC
- Koordinierung von Prozessen
  - Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern

9 Zusammenfassung (2)

- Gegenseitiger Ausschluss mit Spinlocks
- Klassische Koordinierungsprobleme und deren Lösung mit Semaphoren
  - Gegenseitiger Ausschluss
  - Bounded buffers
  - Leser-Schreiber-Probleme
  - Philosophenproblem
  - Schlafende Friseure

9 Zusammenfassung (3)

- UNIX Systemaufrufe
  - fork, exec, wait, nice
  - pipe, socket, bind, recvfrom, sendto, listen, accept
  - msgget, msgsnd, msgrcv
  - signal, kill, sigaction
  - semget, semop, semctl