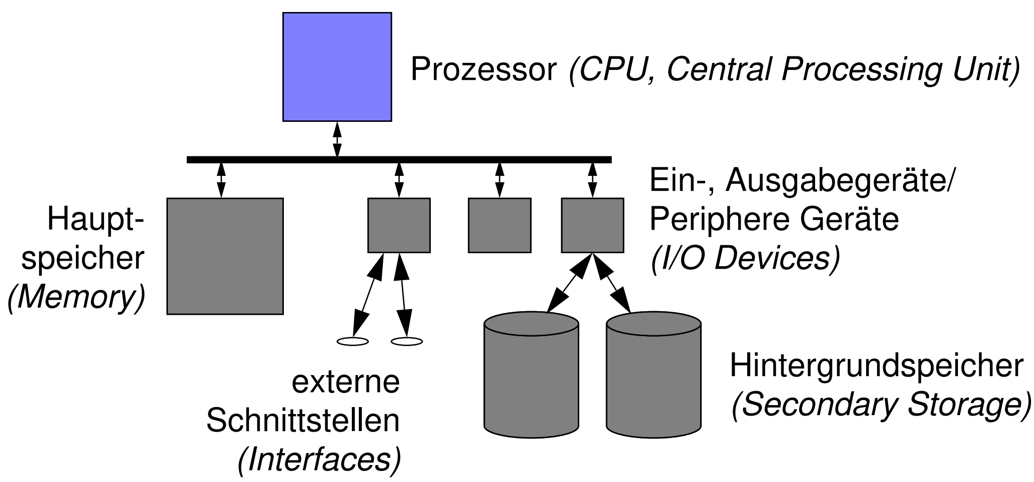
D Prozesse und Nebenläufigkeit

D Prozesse und Nebenläufigkeit

- Einordnung



© 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.

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)

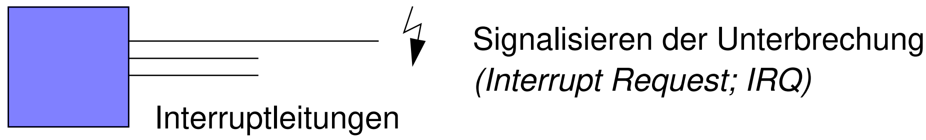
- Beispiel für Instruktionen

```
...
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

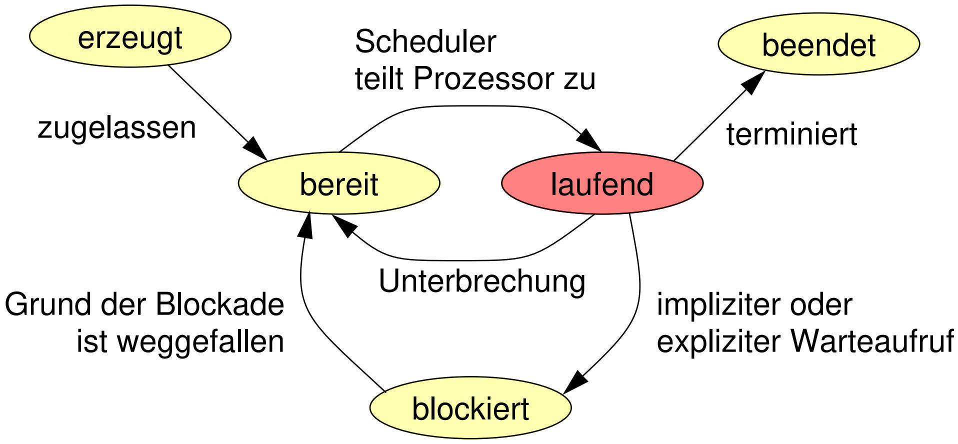
- Stapelsysteme *(Batch Systems)*
  - ein Programm läuft auf dem Prozessor von Anfang bis Ende
- Heutige Systeme *(Time Sharing Systems)*
  - mehrere Programme laufen gleichzeitig
  - Prozessorzeit muss den Programmen zugeteilt werden
  - Programme laufen nebenläufig
- Terminologie
  - **Programm:**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

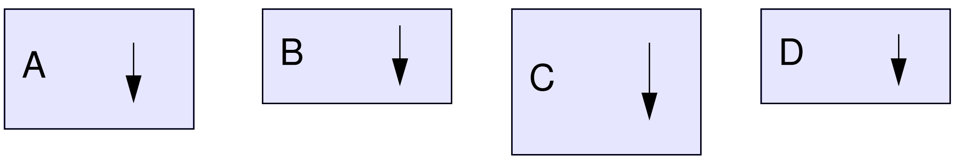


Nach Silberschatz, 1994

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

2.2 Prozesswechsel

- Konzeptionelles Modell

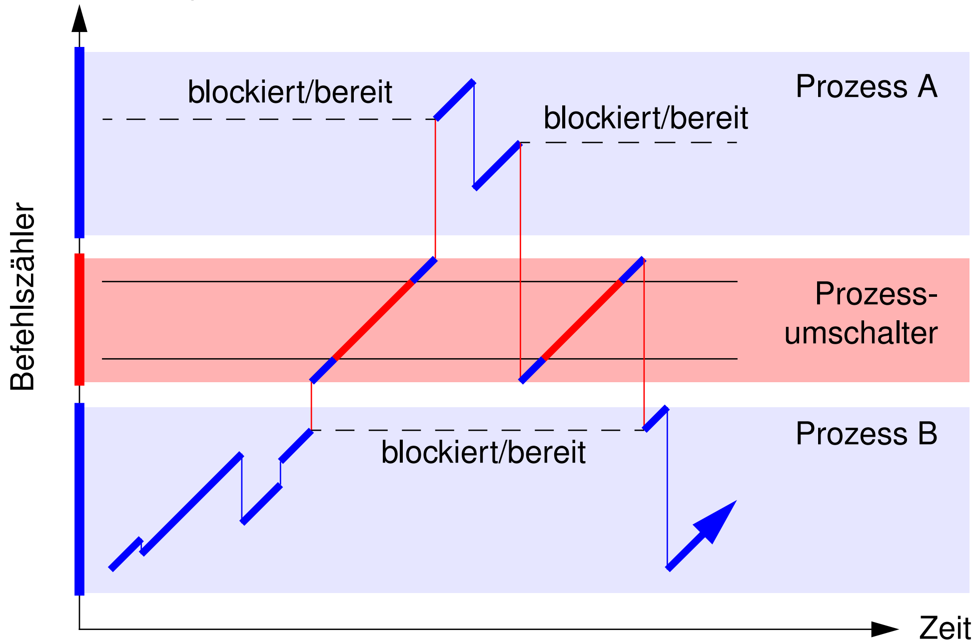


vier Prozesse mit eigenständigen Befehlszählern

- 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:

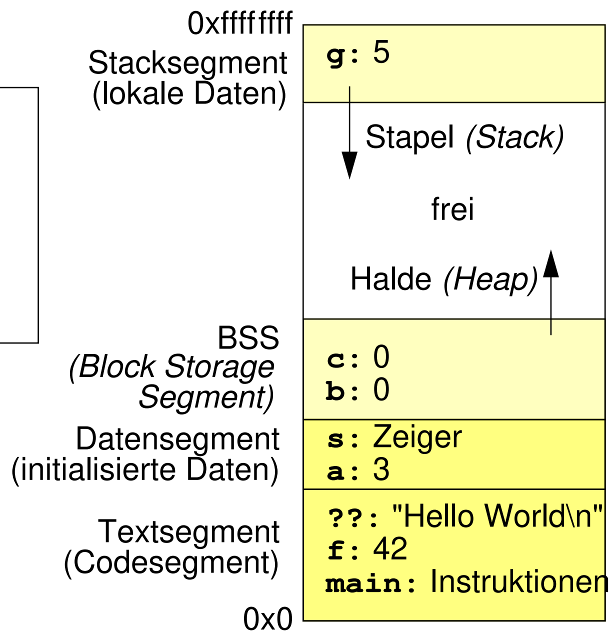
- Verschiedene PIDs
- `fork()` liefert verschiedene Werte als Ergebnis für Vater und Kind

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;
}
```



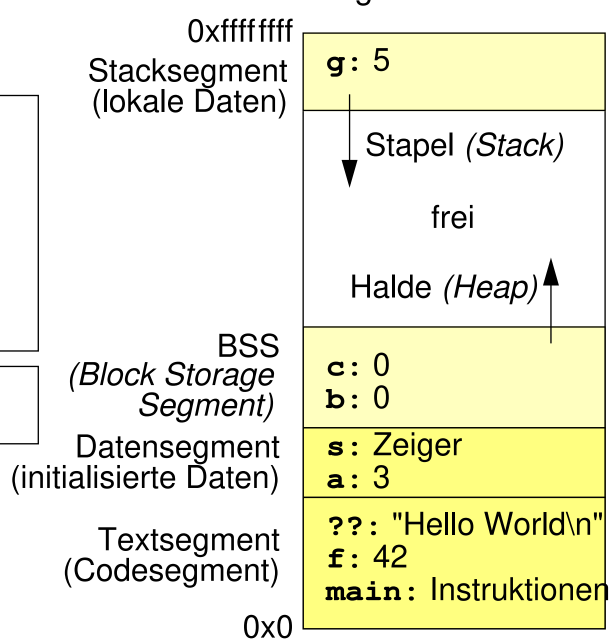
2.4 Speicheraufbau eines Prozesses (UNIX)

- 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;
```



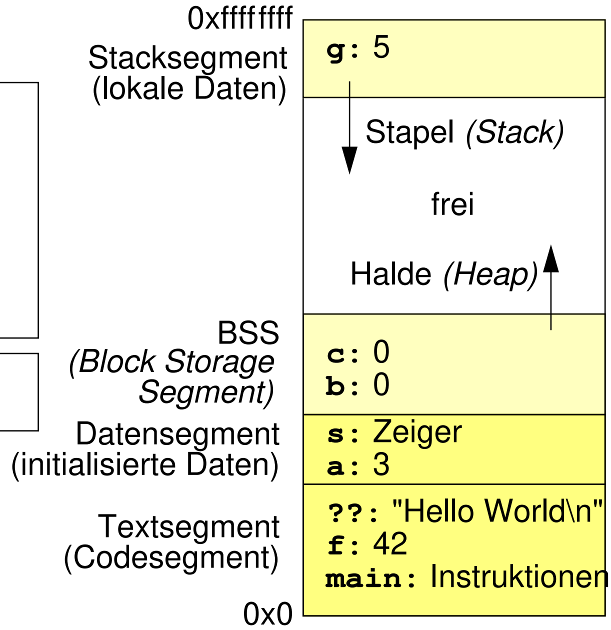
2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

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

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

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



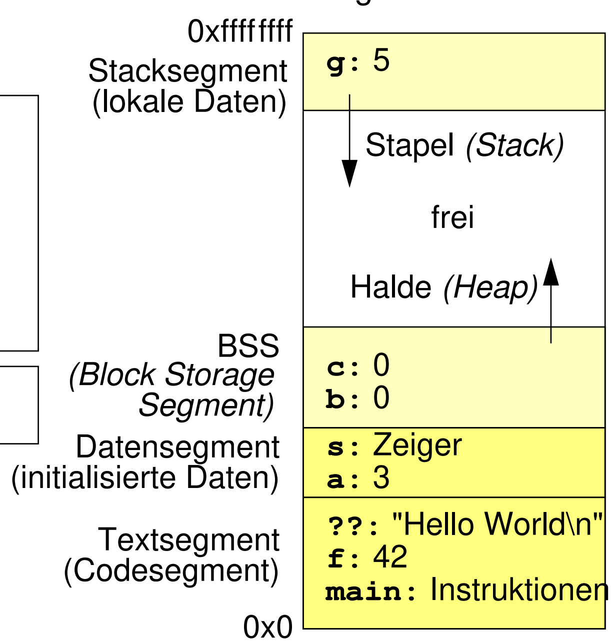
2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

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

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

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



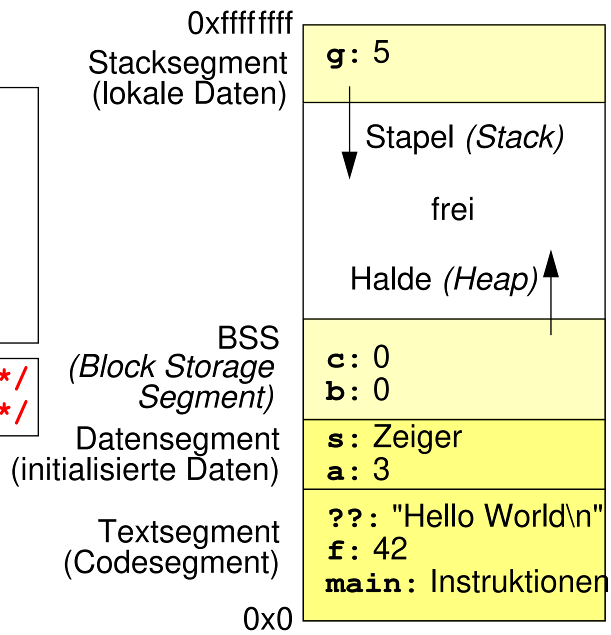
2.4 Speicheraufbau eines Prozesses (UNIX)

- 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[] );
```

Prozess A

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

Prozess A

Prozess A

```
...
int main( int argc, char *argv[] )
{
    ...
```

Altes ausgeführtes Programm ist endgültig beendet.

2.6 Operationen auf Prozessen (UNIX)

- Prozess beenden

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

- Prozessidentifikator

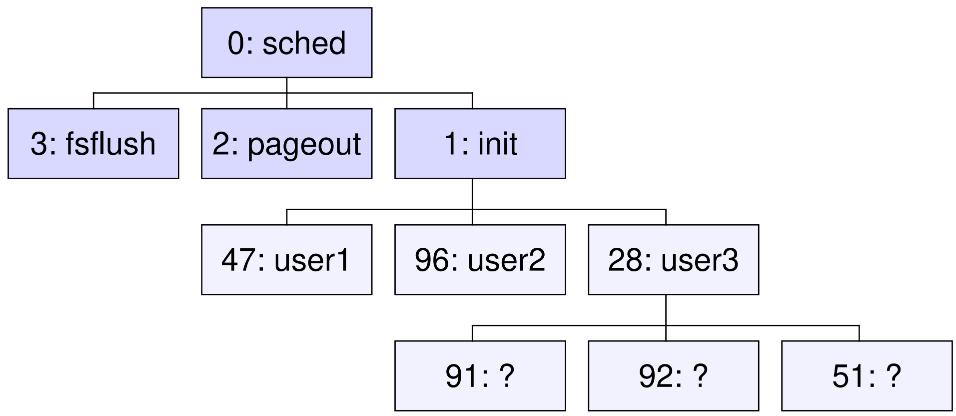
```
pid_t getpid( void );      /* eigene PID */
pid_t getppid( void );     /* PID des Vaterprozesses */
```

- 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.

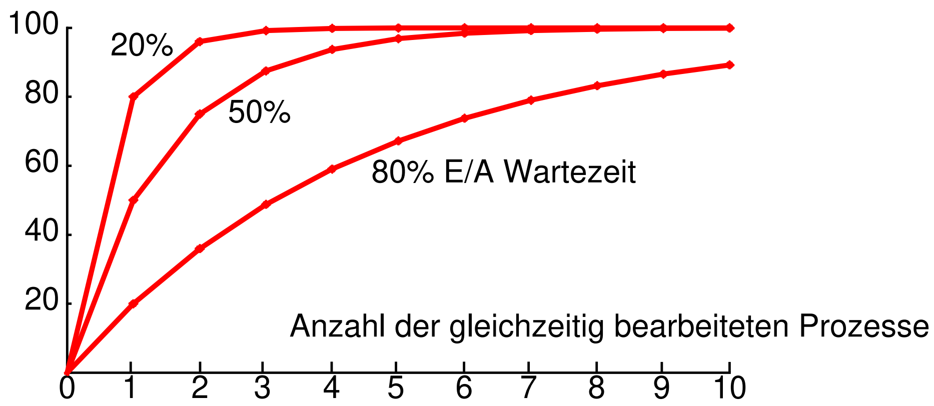
ja → verdrängend

nein → 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



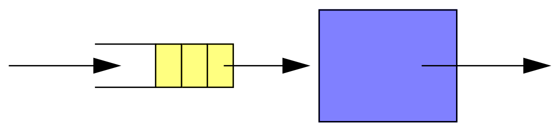
Nach Tanenbaum, 1995

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

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

- 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

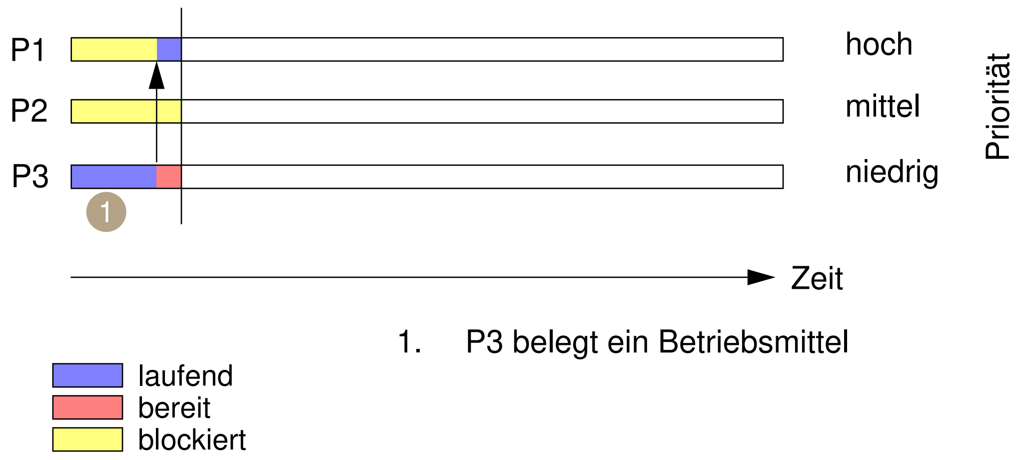
- Kürzester Job wird ausgewählt *(SJF)*
  - Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. Ein-, Ausgabe)
- „bereit"-Warteschlange wird nach Länge der nächsten Rechenphase sortiert
  - Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, 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)

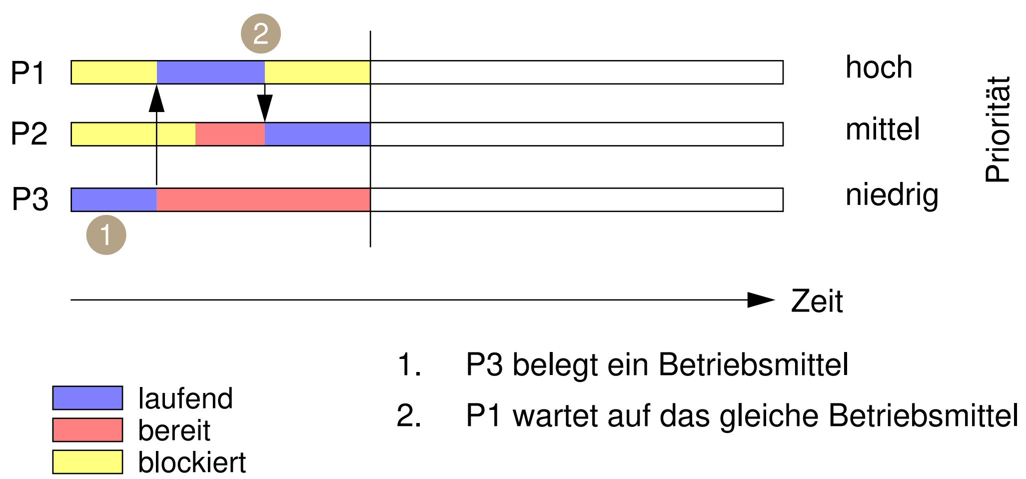
- Prioritätenumkehr
  - hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben



1. P3 belegt ein Betriebsmittel

3.3 Prioritäten (2)

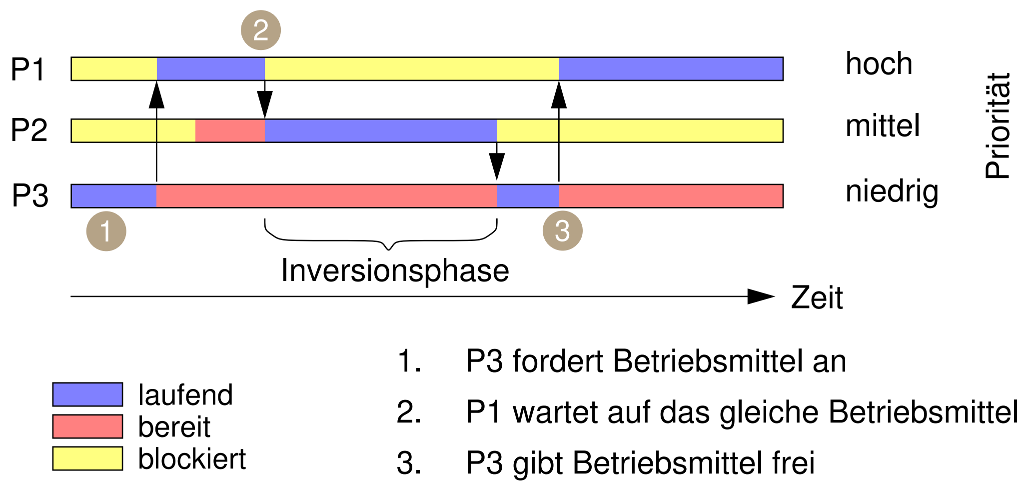
- Prioritätenumkehr
  - hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben



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



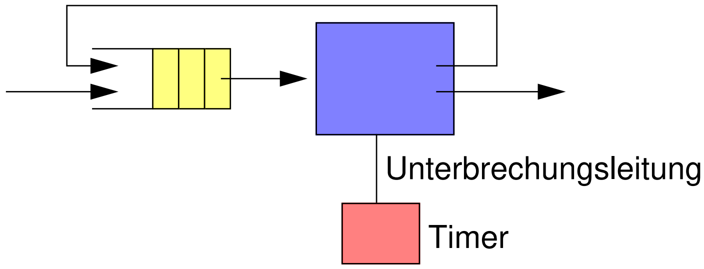
1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei

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

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

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

P1	P2	P3	P1	P1	P1	P1	P1
0 – 4	4 – 7	7 – 10	10 – 14	14 – 18	18 – 22	22 – 26	26 – 30

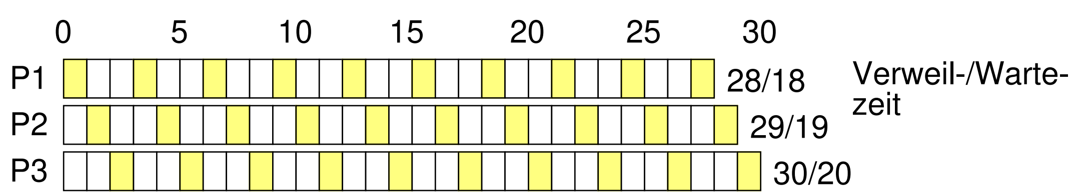
mittlere Wartezeit: (6+4+7)/3 = 5.7

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)

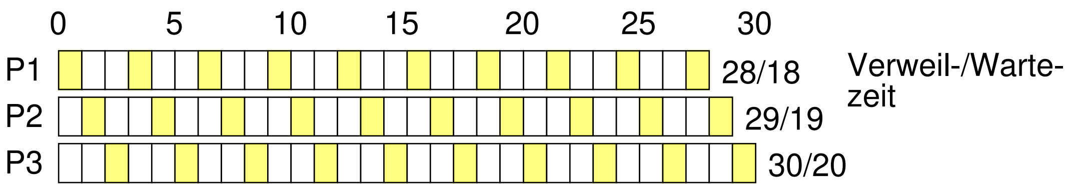
◆ Zeitscheibengröße 1:



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

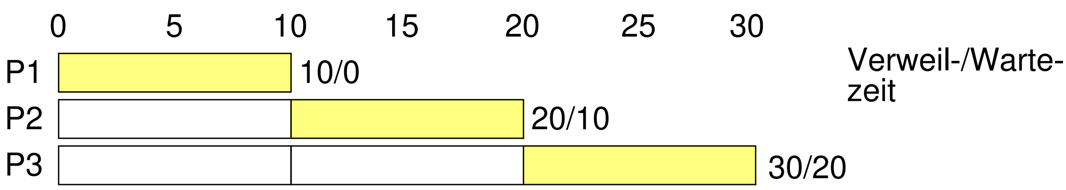
3.4 Round-Robin Scheduling (4)

◆ 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)

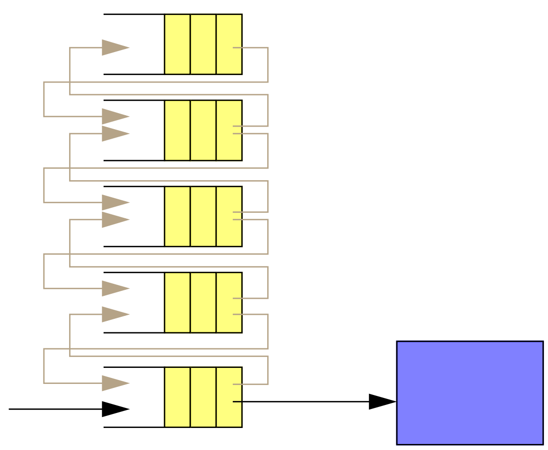
- ◆ Scheduling zwischen den Klassen mit fester Priorität (z.B. Real-Time-Prozesse vor Time-Sharing-Prozessen)
- ◆ In jeder Klasse wird ein eigener Algorithmus benutzt:
  - Systemprozesse: FCFS
  - Real-Time Prozesse: statische Prioritäten
  - Time-Sharing und interaktive Prozesse: ausgefeiltes Verfahren 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)