

# Grundlagen der Informatik für Ingenieure I

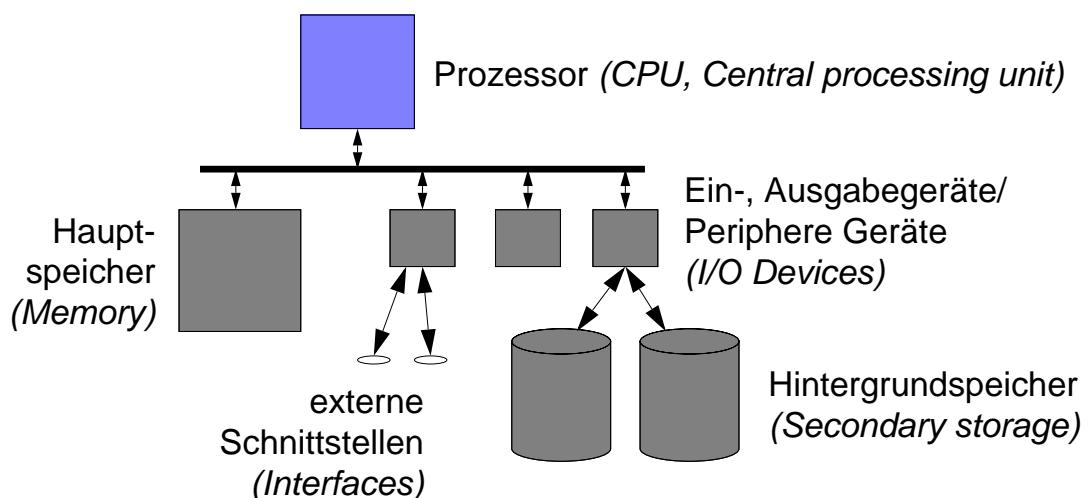
## Background:

### 5. Prozessverwaltung - Einführung

- 5.1 Wichtige Komponenten eines Prozessors
- 5.2 Betriebsstrategien von Rechensystemen
- 5.2.1 Aktivitätsträger, Prozesse, Threads
  - 5.2.1.1 Prozesszustände
  - 5.2.1.2 Prozesswechsel
- 5.2.2 Operationen auf Prozesse
- 5.2.3 Prozesshierarchie
- 5.2.4 Threads
- 5.2.5 Auswahlstrategien

## 5. Prozessverwaltungssystem

### ■ Einordnung



# 5.1 Wichtige Komponenten eines Prozessors

(Siehe auch Kapitel B3 - Rechnerarchitektur!)

## ■ Register

- ◆ Prozessor besitzt Steuer- und Vielzweckregister (Rechenregister)
- ◆ Steuerregister:
  - Befehlszähler (*Instruction pointer*)
  - Stapelregister (*Stack pointer*)
  - Statusregister
  - etc.

## ■ Befehlszähler (Programmadressregister, Programmzähler) enthält Speicherstelle der nächsten Instruktion

- ◆ Instruktion wird geladen und
- ◆ ausgeführt
- ◆ Programmzähler wird inkrementiert
- ◆ dieser Vorgang wird ständig wiederholt

# 5.1 Wichtige Komponenten eines Prozessors

## ■ 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
...

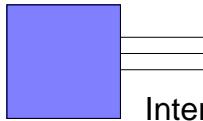
```

## ■ Prozessor arbeitet in einem bestimmten **Modus** (mindestens 2)

- ◆ **Benutzermodus**: eingeschränkter Befehlssatz
- ◆ **privilegierter Modus**: erlaubt zusätzlich Ausführung privilegierter Befehle
  - Konfigurationsänderungen des Prozessors
  - Moduswechsel
  - spezielle Ein-, Ausgabebefehle

## 5.1 Wichtige Komponenten eines Prozessors

### ■ Unterbrechungen (Interrupts)



Signalisieren der Unterbrechung  
(*Interrupt request; IRQ*)

Interruptleitungen

- ◆ Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus
- ◆ vorher werden alle Register einschließlich Programmzähler gesichert (z.B. auf dem Stack oder Registersatz-Umschaltung)
- ◆ nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden (Scheduler-Entscheidung)
- ◆ Unterbrechungen werden im privilegierten Modus bearbeitet
- ◆ geschachtelte Unterbrechungen werden in der Regel vermieden

## 5.1 Wichtige Komponenten eines Prozessors

### ■ Systemaufrufe (Traps; User interrupts)

- ◆ Wie kommt man kontrolliert vom Benutzermodus in einen privilegierten Modus?
- ◆ spezielle Befehle zum Eintritt in den privilegierten Modus
- ◆ Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus
- ◆ solche Befehle werden dazu genutzt, die Betriebssystemschnittstelle zu implementieren (*Supervisor calls (SVC)*)
- ◆ Parameter werden nach einer Konvention übergeben (z.B. auf dem Stack; über ausgezeichnete Register) und vom Betriebssystem überprüft!

## 5.2 Betriebsstrategien von Rechnesystemen

### ■ **Stapelsysteme** (*Batch systems*)

- ◆ ein Programm läuft auf dem Prozessor von Anfang bis Ende  
(üblich bei "numerischen Langläufern" auf Hochleistungssystemen)

### ■ **Zeitscheiben - Systeme** (*Time sharing systems*)

- ◆ mehrere Programme laufen (quasi) gleichzeitig
- ◆ Prozessorzeit muss den Programmen zugeteilt werden
- ◆ Programme laufen nebenläufig

### ■ **Echtzeit - Systeme** (*Realtime systems*)

- ◆ Garantiertes Reaktionsverhalten auf externe Ereignisse

## 1 Aktivitätsträger, Prozesse, Threads

### ■ Terminologie:

- ◆ **Programm:** Folge von Anweisungen

### ◆ **Prozess:**

- Programm, das sich in Ausführung befindet, und seine Daten und seine "Ausführungsumgebung" (*Schutzumgebung*).
- Ein Programm kann sich mehrfach in Ausführung befinden.

### ◆ **Aktivitätsträger:**

- Ein sich in Ausführung befindliches Programm entspricht der Abarbeitung der Maschinenbefehlssequenzen, die sich aus dem Programmcode ergeben.
- Diese Sequenzen ziehen sich - bildlich gesprochen - wie ein Faden (*Thread*) durch den Code.

# 1 Aktivitätsträger, Prozesse, Threads

## ◆ Aktivitätsträger (cont.):

- Ein Programmcode kann von mehreren Threads abgewickelt werden:
  - quasi parallel: zeitlich verschränkt auf einem Prozessor
  - oder tatsächlich parallel: auf mehreren Prozessoren

## ◆ Threads:

- **Threads** eines Prozesses laufen in der Schutzumgebung **EINES Prozesses** ab.
- Man nennt sie deswegen auch ***light-weighted processes* (LWP)**.
- Ein Sonderfall ist der Prozess mit genau einem Thread (***heavy-weighted processes***).

## 5.2.1.1 Prozesszustände

- Ein Prozess befindet sich üblicherweise in einem der folgenden Zustände:

### ◆ Erzeugt (New)

Prozess wurde erzeugt; Prozess besitzt noch nicht alle Betriebsmittel zum Laufen

### ◆ Bereit (Ready)

Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen

### ◆ Laufend (Running)

Prozess wird vom realen Prozessor ausgeführt

### ◆ Blockiert/Wartend (Blocked/Waiting)

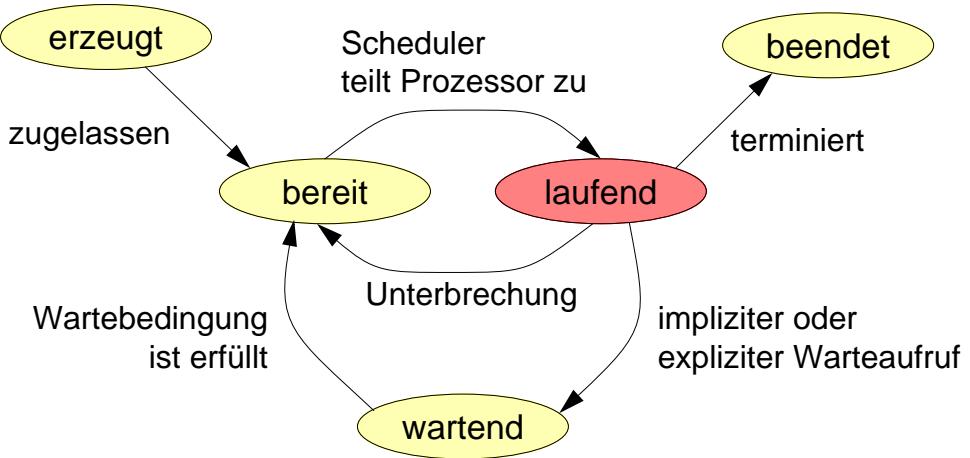
Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht)

### ◆ Beendet (Terminated)

Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

## 5.2.1.1 Prozesszustände

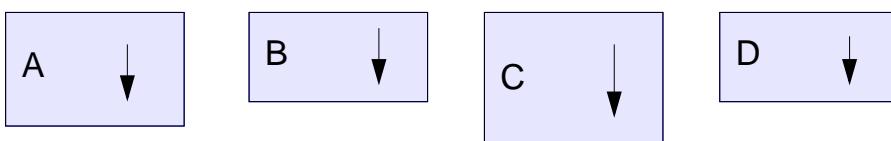
### ■ Zustandsdiagramm



- ◆ Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors (ggf. mehrere) vornimmt.

## 5.2.1.2 Prozesswechsel

### ■ Konzeptionelles Modell



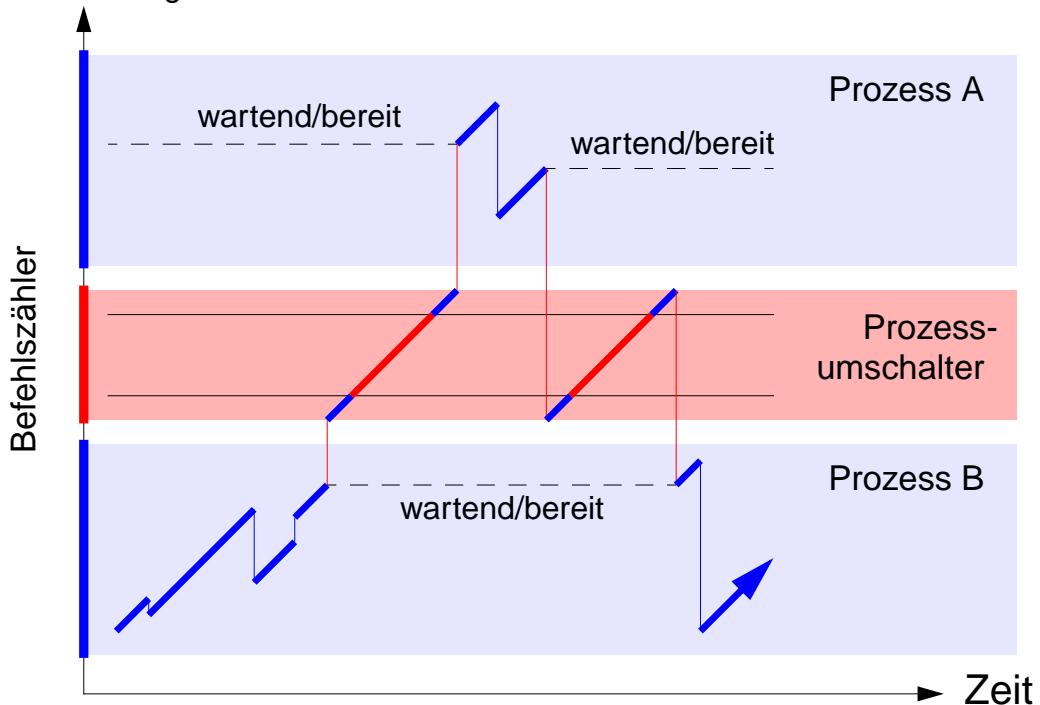
vier Prozesse mit eigenständigen Befehlszählern

### ■ 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.

## 5.2.1.2 Prozesswechsel

### ■ Umschaltung



## 5.2.1.2 Prozesswechsel

### ■ Prozesskontrollblock (Process control block; PCB):

- ◆ Datenstruktur, die alle nötigen Daten für die Verwaltung eines Prozesses enthält.

Beispielsweise in UNIX:

- Prozessnummer (*PID*)
- verbrauchte Rechenzeit
- Erzeugungszeitpunkt
- Kontext (Register etc.)
- Speicherabbildung
- Eigentümer (*UID*, *GID*)
- Wurzelkatalog, aktueller Katalog
- offene Dateien
- ...

## 5.2.1.2 Prozesswechsel

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

## 2 Operationen auf Prozessen

- Typische Operationen eines UNIX Betriebssystems:
  - ◆ **Erzeugen** eines neuen Prozesses:
    - dupliziert den gerade laufenden Prozess)
    - aus einem Vaterprozess wird ein Kindprozess erzeugt

```
pid_t fork( void );
```
  - ◆ **Ausführen** eines Programms:
 

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

```
/* Es gibt noch weitere Varianten dieses Systemcalls */
```
  - ◆ Prozess **beendet** sich:
 

```
void exit( int status );
```

## 2 Operationen auf Prozessen (cont.)

◆ Prozessidentifikator:

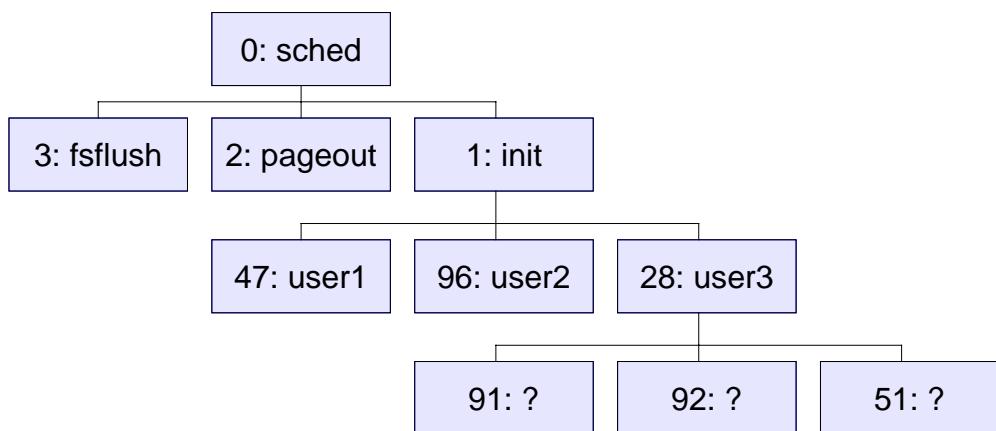
```
pid_t getpid( void );           /* eigene PID */
pid_t getppid( void );         /* PID des Vaterprozesses */
```

◆ Warten auf Beendigung eines Kindprozesses:

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

## 3 Prozesshierarchie (Solaris)

- Hierarchie wird durch Vater-Kind-Beziehung erzeugt



- ◆ Nur der Vater kann auf das Kind warten
- ◆ Init-Prozess adoptiert verwaiste Kinder

## 4 Threads

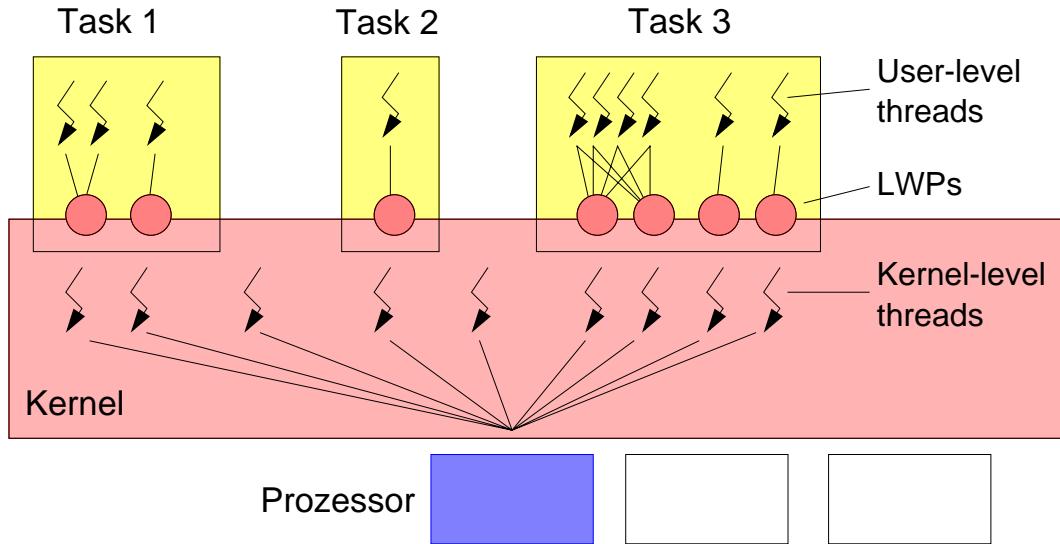
- Thread als leichtgewichtiger Prozess (*Light weight process, LWP*):
  - ◆ keine eigene Schutzumgebung
  - ◆ mehrere Threads teilen sich einen Speicherbereich und arbeiten im gleichen Instruktionen- und Datenbereich
  - ◆ Gruppe von Threads einschließlich ihrer Speicherabbildung wird oft als *Task* bezeichnet
- Ein Prozess ist ein Task mit einem Thread !
- Implementierungen von Threads
  - ◆ User-level threads
  - ◆ Kernel-level threads

## 4 Threads

- **User-level threads:**
  - ◆ Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - ◆ Betriebssystem sieht nur einen Thread
    - keine Systemaufrufe zum Umschalten erforderlich
    - effiziente Umschaltung
  - ◆ Bei blockierenden Systemaufrufen bleiben alle User-level threads stehen
- **Kernel-level threads:**
  - ◆ Betriebssystem schaltet Threads um
    - weniger effizientes Umschalten
  - ◆ Kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
  - ◆ Fairnessverhalten nötig (zwischen Prozess und Task mit vielen Threads)

## 4 Beispiel: LWPs und Threads (Solaris)

- Solaris kennt Kernel- und User-level threads



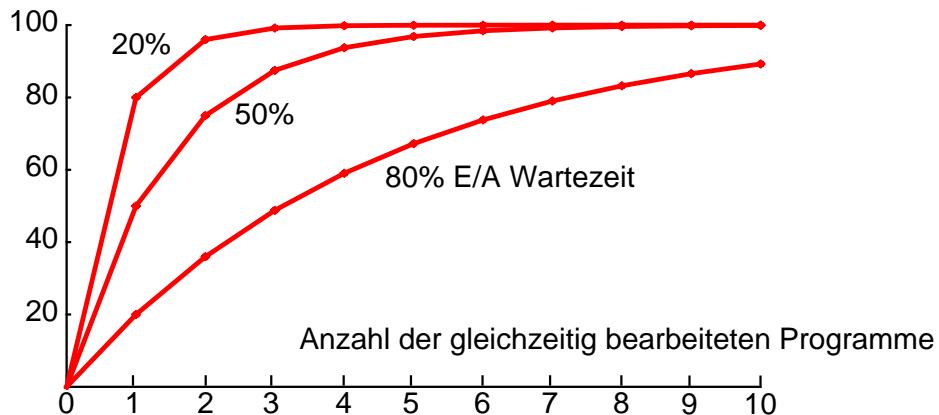
Nach Silberschatz, 1994

## 5 Auswahlstrategien

- Strategien zur Auswahl des nächsten Prozesses (*Scheduling strategies*):
  - ◆ Mögliche Stellen zum Treffen von Schedulingentscheidungen:
    1. Prozess wechselt vom Zustand „laufend“ zum Zustand „wartend“ (z.B. Ein-, Ausgabeoperation)
    2. Prozess wechselt von „laufend“ nach „bereit“ (z.B. bei einer Unterbrechung des Prozessors)
    3. Prozess wechselt von „wartend“ nach „bereit“
    4. Prozess terminiert
  - ◆ Bei 1. und 4. gibt es immer einen Prozesswechsel
  - ◆ Bei 2. und 3. gibt es nur einen Prozesswechsel bei verdrängender Strategie
    - Verdrängend (preemptive): laufender Prozess wird unterbrochen und in den Zustand "bereit" gebracht
    - Nicht verdrängend (nonpreemptive): laufender Prozess wird vollständig bearbeitet und darf nicht unterbrochen werden.

## 5 Auswahlstrategien

- ◆ CPU-Auslastung (betrachtet werden *Time sharing systems*)
  - CPU soll möglichst gut ausgelastet sein, jedoch NICHT wirklich zu 100%!!
  - CPU-Auslastung in Prozent, abhängig von der Anzahl der Programme und deren prozentualer Wartezeit



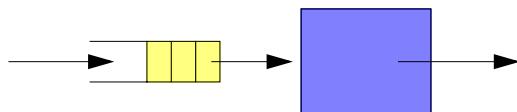
Nach Tanenbaum, 1995

## 5 Auswahlstrategien

- Weitere Kriterien für die Auswahlstrategie:
  - ◆ 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

## 5.2.5.1 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



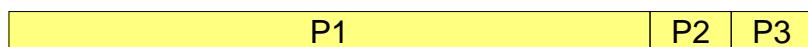
- Bewertung
  - ◆ fair (?)
  - ◆ Wartezeiten nicht minimal
  - ◆ nicht für Time sharing-Betrieb geeignet

## 5.2.5.1 First Come First Served

- Beispiel zur Betrachtung der Wartezeiten

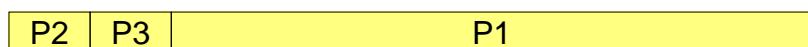
Prozess 1:	24	}	Zeiteinheiten
Prozess 2:	3		
Prozess 3:	3		

- ◆ Reihenfolge: P1, P2, P3



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

- ◆ Reihenfolge: P2, P3, P1



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

## 5.2.5.2 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.
- Variante: verdrängend (PSJF = Preemptive Shortest Job First) und nicht-verdrängend (SJF)

## 5.2.5.3 Prioritäten

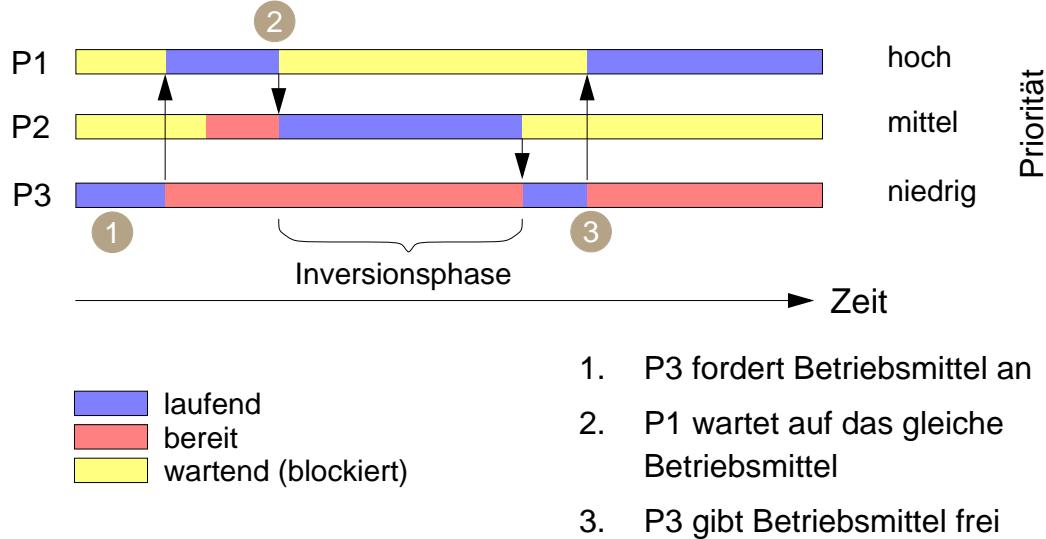
- Prozess mit höchster Priorität wird ausgewählt
  - ◆ dynamisch — statisch
    - (z.B. SJF: dynamische Vergabe von Prioritäten gemäß Länge der nächsten Rechenphase)
    - (z.B. statische Prioritäten in Echtzeitsystemen; Vorhersagbarkeit von Reaktionszeiten)
  - ◆ verdrängend — nicht-verdrängend
- Probleme:
  - ◆ Aushungerung
 

Ein Prozess kommt nie zum Zuge, da immer andere mit höherer Priorität vorhanden sind.
  - ◆ Prioritätenumkehr (*Priority inversion*)

## 5.2.5.3 Prioritäten

### ■ 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



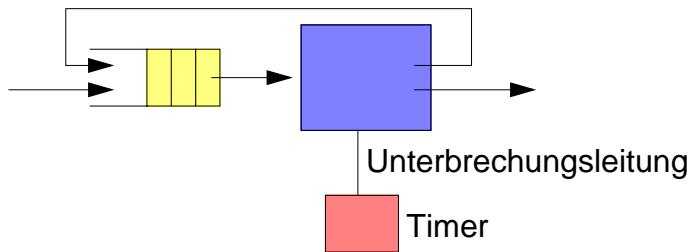
## 5.2.5.3 Prioritäten

### ■ Lösungen:

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

## 5.2.5.4 Round Robin Scheduling

- Zuteilung und Auswahl erfolgt reihum
  - ◆ ähnlich FCFS aber mit Verdrängung
  - ◆ Zeitquantum (*Time quantum*) oder Zeitscheibe (*Time slice*) wird zugeteilt
  - ◆ geeignet für *Time sharing*-Betrieb



- ◆ Wartezeit ist jedoch eventuell relativ lang

## 5.2.5.5 Multilevel Feedback Queue Scheduling

- Mehrere Warteschlangen (*MLFB*)
  - ◆ jede Warteschlange mit eigener Behandlung
  - ◆ Prozesse können von einer zur anderen Warteschlange transferiert werden

