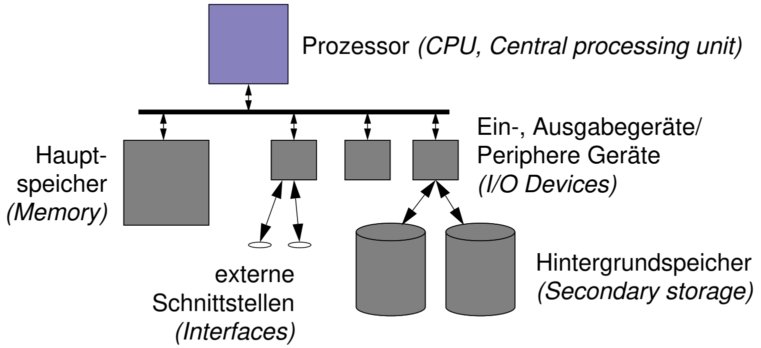
F Prozesse und Nebenläufigkeit

- Einordnung



F.1 UNIX — Prozeßverwaltung

1 Prozeß

- Ausführung eines Programms in der durch die Prozeßdatenstrukturen definierten Umgebung

2 Datenstrukturen zur Ausführung eines Programms

- Programmkode und Programmdaten werden in **Segmenten *(Regions)*** organisiert
- abhängig vom gerade auszuführenden Programm, können diese während der Lebenszeit eines Prozesses gewechselt werden

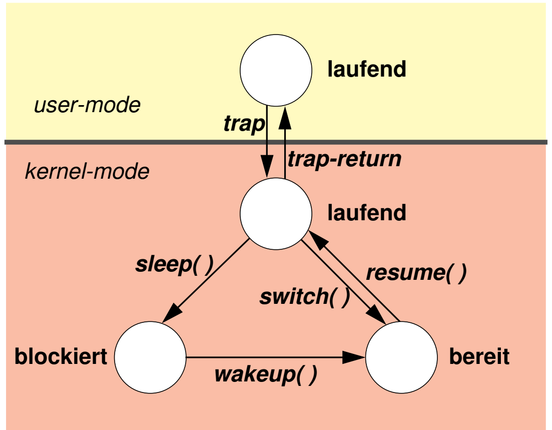
2 Datenstrukturen zur Ausführung eines Programms (2)

- **Text-Segment**
  Maschineninstruktionen des Programms
  (in der Regel schreibgeschützt, wird von mehreren Prozessen gemeinsam benutzt)
- **Daten-Segment**
  Daten des Programms (global und *static*), dynamisch erweiterbar (*malloc(3), sbrk(2)*)
- **Stack-Segment**
  lokale Daten und Aufrufparameter von Funktionen
  sowie Sicherungsbereiche für Registerinhalte und Rücksprungadressen
  — wächst bei Bedarf
- **shared Daten-Segment (optional)**
  gemeinsamer Datenbereich für mehrere Prozesse

3 Multiprogramming, Scheduling

- UNIX erlaubt die (quasi-)gleichzeitige Abwicklung mehrerer Prozesse
- Umschaltung zwischen Prozessen durch **Prozeßwechsel *(context switching)***
- Prozeßwechsel erfolgen
  - wenn Prozesse warten müssen (z. B. auf E/A), oder
  - nach einer bestimmten Laufzeit — **Zeitscheibe *(time slice)***
- die Entscheidung, welcher Prozeß als nächstes den Prozessor zugeteilt bekommt ***(Scheduling)*** erfolgt auf der Basis **dynamischer Prioritäten *(multi-level feedback)***

4 Prozeßzustände



5 Prozeßdatenstrukturen

- 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
    - ...

6 Prozeßwechsel

- 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

7 Erzeugen von Prozessen

- jeder UNIX-Prozeß kann mit dem Systemaufruf ***fork(2)*** einen neuen Prozeß erzeugen
  - *fork( )* erzeugt eine nahezu identische Kopie des aufrufenden Prozesses
  - der *fork( )* aufrufende Prozeß wird **Vaterprozeß *(parent process)*** genannt
  - der durch *fork( )* neu erzeugte Prozeß wird als **Sohnprozeß *(child process)*** bezeichnet
  - der Sohnprozeß erbt alle Rechte und Einschränkungen vom Vaterprozeß
  - wesentliche Unterschiede zwischen Vater- und Sohnprozeß:

fork-Ergebnis:	Vaterprozeß: *pid* des Sohnes, Sohnprozeß: 0
PID:	Sohnprozeß erhält nächste freie *pid*

7 Erzeugen von Prozessen (2)

- Beispiel:

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


7 Erzeugen von Prozessen (3)

- Beispiel:

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

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


8 Ausführen eines Programms

- ein UNIX-Prozeß kann die Ausführung eines Programms durch ein anderes Programm ersetzen — ***exec(2), execve(2)***
  - ◆ durch Laden eines neuen Programms werden die zuvor von dem Prozeß bearbeiteten Programm-Datenstrukturen zerstört
    ➥ **im Erfolgsfall gibt es kein return aus *exec( )***
  - ◆ durch Laden eines neuen Programms entsteht **kein neuer Prozeß**
- *execve( )* werden die Argumente ***(argv)*** und ein Environment ***(envp)*** für das neue Programm mitgegeben
- Details:
  - ◆ programmspezifische Daten des Prozesses (Segmente, Signalbehandlungsfunktionen) werden initialisiert
  - ◆ Zugriffsrechte des Prozesses werden auf Eigentümer/Gruppe der Programm-Datei geändert, wenn bei dieser ein ***s-bit*** gesetzt ist

8 Ausführen eines Programms (2)

- … Details:
  - ◆ alle anderen Prozeßparameter bleiben unverändert (*pid, current working directory, ...*)
  - ◆ **offene Dateideskriptoren bleiben erhalten**, es sei denn, sie wurden als *close-on-exec* (siehe ***fcntl(2)***) markiert
- Programmier-Beispiel:

```
if ( (pid = fork()) < 0 ) {
    perror("fork");
    exit(1)
} else if (pid == 0) {          /* child process */
    execl("/bin/cp", "cp",    "/tmp/a", "/tmp/b", (char *)0);
    perror("exec");
    exit(1);
} else {                         /* parent process */
          ...
}
```

9 Terminieren von Prozessen

- Prozesse terminieren, wenn
  - ◆ sie den Systemdienst ***exit(2)*** aufrufen
  - ◆ ein Signal an den Prozeß zugestellt wurde, für das keine Signalbearbeitungsfunktion vorgesehen ist
- die Startumgebung für C-Programme ruft nach einem *return* aus der Funktion *main* automatisch *exit(0)* auf
- dem *exit*-Aufruf kann ein Status-Wert (1 Byte) mitgegeben werden, der durch den Vaterprozeß abgefragt werden kann (→ *wait(2)*)
- ein Vaterprozeß kann auf das Terminieren von Sohnprozessen warten und deren exit-Status abfragen — ***wait(2)***

10 Systemaufruf *wait( )*

- ein Prozeß kann warten, bis ein Sohnprozeß terminiert oder gestoppt wird und dabei den Status des Sohnprozesses abfragen
  ➥ übliche Arbeitsweise einer Shell bei Vordergrundprozessen
- ***wait(2)*** blockiert den aufrufenden Prozeß so lange, bis ein Sohnprozeß im Zustand ZOMBIE existiert oder ein Sohnprozeß gestoppt wird
  - ◆ *pid* dieses Sohnprozesses wird als Ergebnis geliefert
  - ◆ als Parameter kann ein Zeiger auf einen *int*-Wert mitgegeben werden, in dem der Status (16 Bit) des Kind-Prozesses abgelegt wird
  - ◆ in den Status-Bits wird eingetragen "was dem Kind-Prozess zugestossen ist", Details können über Makros abgefragt werden: (Beispiele)
    - ➤ Prozess "normal" mit exit() terminiert: `WIFEXITED(status)`
    - ➤ exit-Parameter (nur das unterste Byte): `WEXITSTATUS(status)`
    - ➤ Prozess durch Signal abgebrochen: `WIFSIGNALED(status)`
    - ➤ Nummer des Signals, das Abbruch verursacht hat: `WTERMSIG(status)`
    - ➤ weitere siehe `man 2 wait` bzw. `man wstat` (je nach System)

10 Systemaufruf *wait( )* (2)

- Beispiel:

Vater

```
int status;
int p, c;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
    exit(1);
} else if( p != -1 ) {
    /* parent */
    c = wait(&status);
} else {
    /* error */
    ...
}
```

Kind

```
int status;
int p, c;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
    exit(1);
} else if( p != -1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```


11 Systemaufrufe — Überblick

- Prozeß erzeugen

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

- Ausführen eines Programms

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

- Prozeß beenden

```
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 );
```

F.2 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.

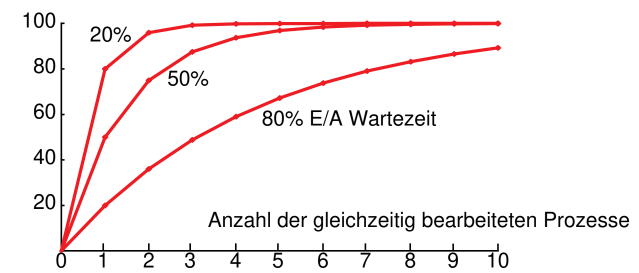
ja — verdrängend *(preemptive)*

nein — nicht verdrängend

F.2 Auswahlstrategien (2)

- CPU Auslastung
  - ◆ CPU soll möglichst vollständig ausgelastet sein

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



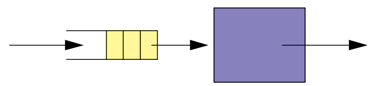
Nach Tanenbaum, 1995

F.2 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

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

1 First Come, First Served (2)

- Beispiel zur Betrachtung der Wartezeiten

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

2 Shortest Job First

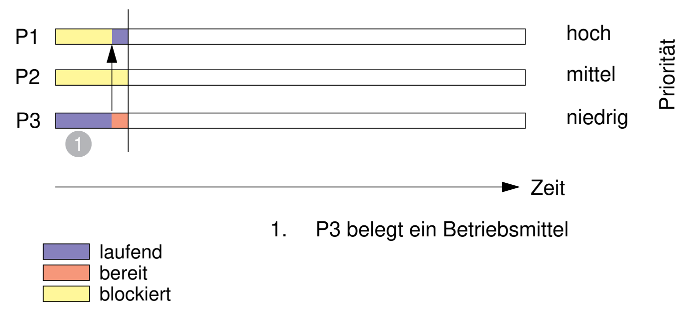
- Kürzester Job wird ausgewählt *(SJF)*
  ◆ Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. Ein-, Ausgabe)
- „bereit"-Warteschlange wird nach Länge der nächsten Rechenphase sortiert
  ◆ Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert,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

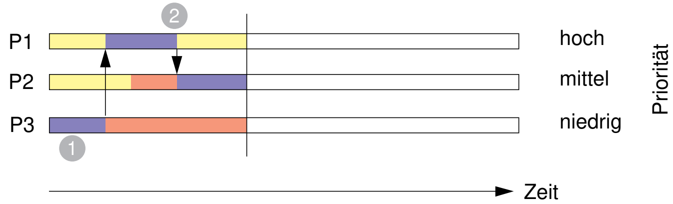
3 Prioritäten (2)

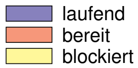
- Prioritätenumkehr
  ◆ hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben



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

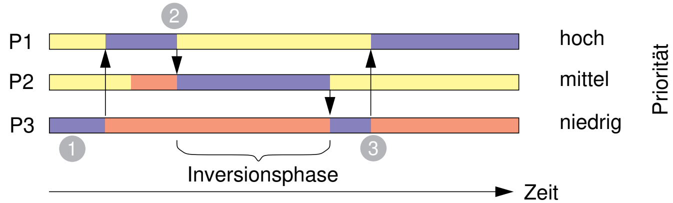


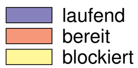


1. P3 belegt ein Betriebsmittel
2. P1 wartet auf das gleiche Betriebsmittel

3 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





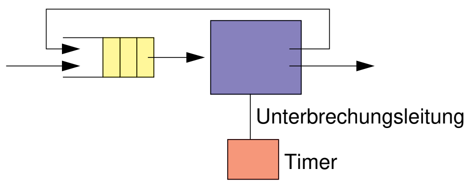
1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei

3 Prioritäten (3)

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

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

4 Round-Robin Scheduling (2)

- Beispiel zur Betrachtung der Wartezeiten

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

◆ 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

Ende: 30

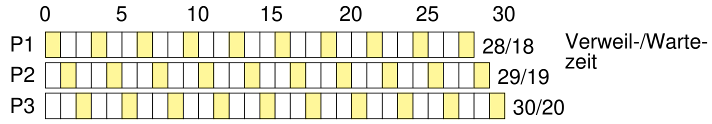
mittlere Wartezeit: (6+4+7)/3 = 5.7

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

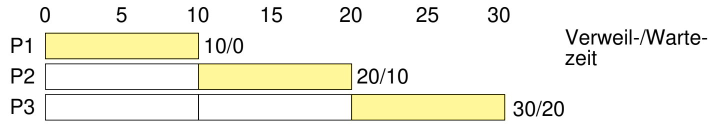
4 Round-Robin Scheduling (4)

◆ 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

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

5 Multilevel-Queue Scheduling (2)

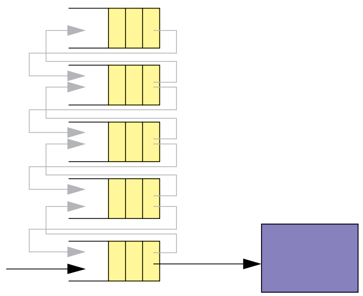
- ◆ Scheduling zwischen den Klassen mit fester Priorität (z.B. Real-Time-Prozesse vor Time-Sharing-Prozessen)
- ◆ In jeder Klasse wird ein eigener Algorithmus benutzt:
  - Systemprozesse: FCFS
  - Real-Time Prozesse: statische Prioritäten
  - Time-Sharing und interaktive Prozesse:
    ausgefeiltes Verfahren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

★ Multilevel Feedback Queue Scheduling

6 Multilevel-Feedback-Queue Scheduling

■ Mehrere Warteschlangen *(MLFB)*

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



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)*

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

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:`** Warteschlangenummer für das Wiedereinreihen nach einer blockierenden Aktion

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

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
...			

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)

F.3 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

1 Prozesse mit gemeinsamem Speicher

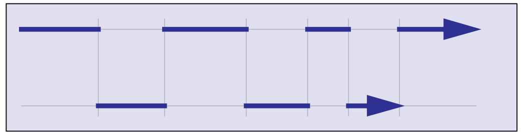
- Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse

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

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

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.

3 Aktivitätsträger

★ **Lösungsansatz:**
Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(Lightweight 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

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

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 Kernel-Level-Threads

■ Implementierung

- ◆ Betriebssystem kennt Kernel-Level-Threads
- ◆ Betriebssystem schaltet Threads um

★ Vorteile

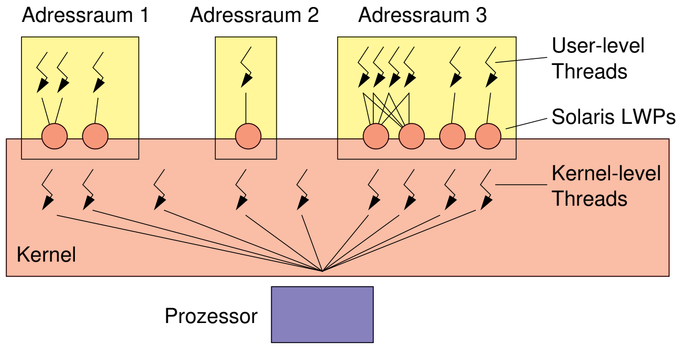
- ◆ kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen

▲ Nachteile

- ◆ weniger effizientes Umschalten
- ◆ Fairnessverhalten nötig (zwischen Prozessen mit vielen und solchen mit wenigen Threads)
- ◆ Schedulingstrategie meist vorgegeben

6 Beispiel: LWPs und Threads (Solaris)

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



Nach Silberschatz, 1994

F.4 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;
```


F.4 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.

F.4 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;
```


F.4 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;]
```

F.4 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.

1 Gegenseitiger Ausschluss

- Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten
  - ◆ Annahme: Maschinenbefehle sind unteilbar (atomar)
- Vollständige und sichere (ohne Verklemmungen) Implementierung nur mit Algorithmus von Petersen

▲ Problem der Lösung

- ◆ aktives Warten

★ Algorithmus auch für mehrere Prozesse erweiterbar

- ◆ Lösung ist relativ aufwändig

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

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

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

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

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

4 Sperrung von Unterbrechungen

- Sperrung der Systemunterbrechungen im Betriebssystem

Prozess 0

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();
```

Prozess 1

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();
```

  - 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

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

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
  - Im Betriebssystem mit gesperrten Unterbrechungen
  - Wartende Prozesse werden blockiert

5 Semaphor (3)

- ★ 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

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	$S := S + 1$

F.5 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

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

1 Gegenseitiger Ausschluss (2)

- Abstrakte Beschreibung: binäre Semaphore

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

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

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

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;
}
```


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**

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 );
...
```

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	$S := S + k$

3 Erstes Leser-Schreiber-Problem (4)

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

```
PV_chunk_semaphore mutex= N;
```

Leser

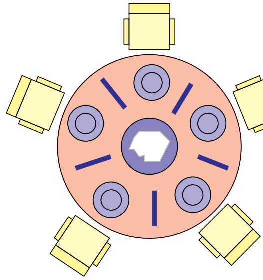
```
...
Pc( &mutex, 1 );

... /* reading */

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

Schreiber

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

... /* writing */

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


4 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

4 Philosophenproblem (2)

- Naive Implementierung
  - ◆ eine Semaphor pro Gabel

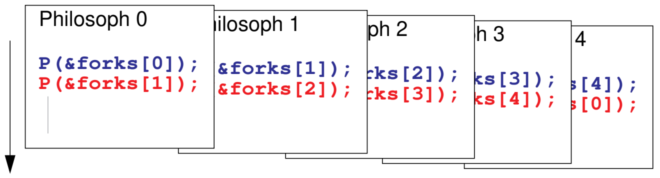
```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

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

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

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

    ... /* eat */

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

4 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

4 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	$\forall i, S_i = S_i + 1$

4 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] );
}
```


4 Philosophenproblem (6)

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

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

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

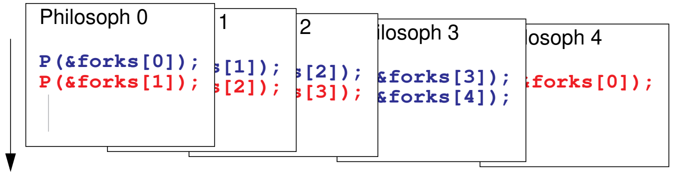
```
while( 1 ) {
    ... /* think */

    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] );
}
```

4 Philosophenproblem (7)

- ◆ Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- ◆ System verklemmt sich nicht

F.6 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
- Koordinierung von Prozessen
  - ◆ Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern
  - ◆ Gegenseitiger Ausschluss mit Spinlocks
- Klassische Koordinierungsprobleme und deren Lösung mit Semaphoren
  - ◆ Gegenseitiger Ausschluss, Bounded buffers, Leser-Schreiber-Probleme, Philosophenproblem