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
6	200	0	0	50	50
7	200	0	0	50	50
8	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 Sek.; min. 1)
  - **`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)

D.5 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)

D.5 Prozeßkommunikation (2)

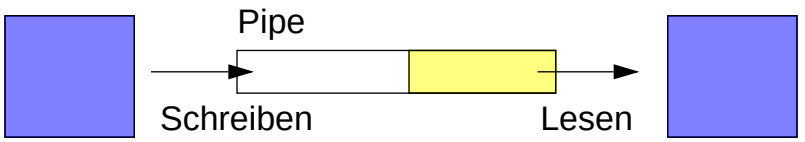
- Klassifikation nachrichtenbasierter Kommunikation
  - ◆ Klassen
    - Kanäle *(Pipes)*
    - Kommunikationsendpunkte *(Sockets, Ports)*
    - Briefkästen, Nachrichtenpuffer *(Queues)*
    - Unterbrechungen *(Signals)*
  - ◆ Übertragunsrichtung
    - unidirektional
    - bidirektional (voll-duplex,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

D.5 Prozeßkommunikation (3)

- ◆ Übertraguns- 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

1 Pipes

- Kanal zwischen zwei Kommunikationspartnern
  - unidirektional
  - gepuffert (feste Puffergröße), zuverlässig, stromorientiert



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

1 Pipes (2)

- Systemaufruf unter Solaris
  - Öffnen einer Pipe

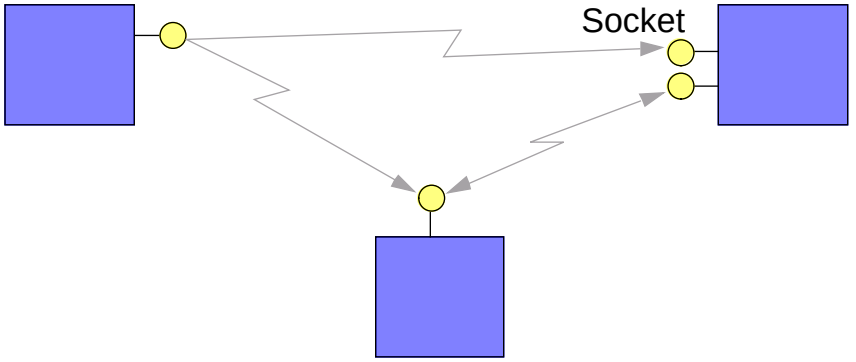
```
int pipe( int fdes[2] );
```

  - 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`**

2 Sockets

- Allgemeine Kommunikationsendpunkte
  - bidirektional, gepuffert



  - Auswahl einer Protokollfamilie
    - z.B. TCP/IP, UNIX (innerhalb von Prozessen der gleichen Maschine)

2 Sockets (2)

- Auswahl eines Protokolls der Familie
  - z.B. UDP
  - Wahl zwischen zuverlässigen und unzuverlässigen Protokollen
  - Wahl zwischen stromorientierten (verbindungsorientierten) und nachrichtenorientierten Protokollen
- explizite Adressierung
  - Unicast: genau ein Kommunikationspartner
  - Multicast: eine Gruppe
  - Broadcast: alle möglichen Adressaten
- können blockierend und nichtblockierend betrieben werden

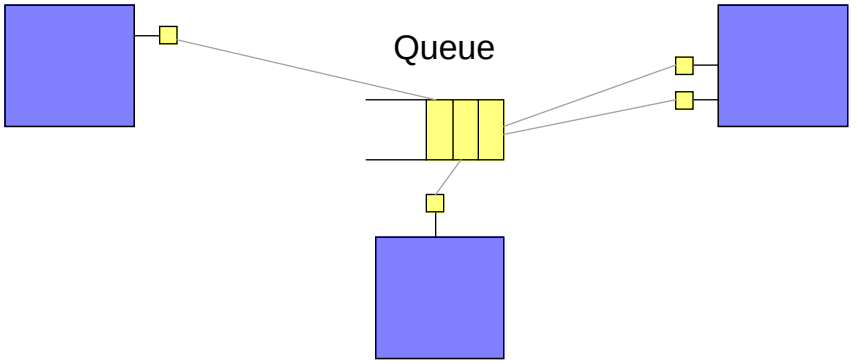
3 UNIX Queues

- 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

3 UNIX Queues (2)

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

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



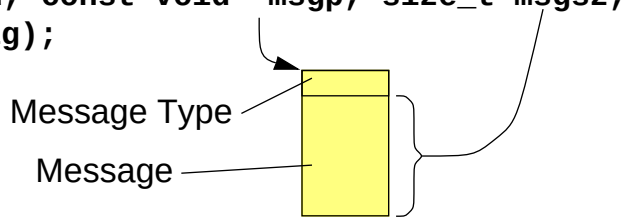
  - Alle kommunizierenden Prozesse müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann keine Queue mit gleichem Key erzeugt werden

3 UNIX Queues (3)

◆ Es können Queues ohne Key erzeugt werden (private 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

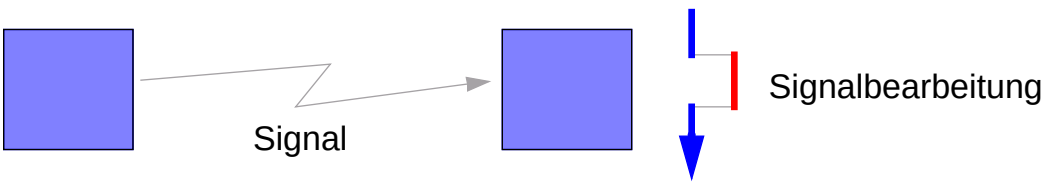
4 UNIX Signale

■ Signale sind Unterbrechungen ähnlich denen eines Prozessors

◆ Prozeß führt eine definierte Signalbehandlung durch

- Ignorieren
- Terminierung des Prozesses
- Aufruf einer Funktion

◆ Nach der Behandlung läuft Prozeß an unterbrochener Stelle weiter



4 UNIX Signale (2)

- Kommunikation über Signale: Signalisierung von Ereignissen

Signale (Beispiele)			Voreingestelltes Verhalten
◆ Terminaleingabe			
• **SIGINT**	Interrupt	^C	*Prozeß terminiert*
• **SIGQUIT**	Quit	^|	*Prozeß terminiert, schreibt Core dump*
◆ Systemsignale ausgelöst durch den Prozeß selbst			
• **SIGBUS**	Bus error		*Prozeß terminiert, schreibt Core dump*
• **SIGSEGV**	Segmentation fault		*Prozeß terminiert, schreibt Core dump*
◆ Systemsignale ausgelöst durch Betriebssystem			
• **SIGALRM**	Alarmzeitgeber		*wird ignoriert*
• **SIGCHLD**	Kindprozeßstatus		*wird ignoriert*
◆ Benutzerdefinierte Signale			
• **SIGUSR1, SIGUSR2**	frei für Benutzerkommunikation, z.B. für Start und Ende einer Bearbeitung		*werden ignoriert*

4 UNIX Signale (3)

- Signalbehandlung kann eingestellt werden:
  - ◆ SIG_IGN: Ignorieren des Signals
  - ◆ SIG_DFL: Defaultverhalten einstellen
  - ◆ *Funktionsadresse:* Funktion wird in der Signalbehandlung aufgerufen und ausgeführt
- UNIX Systemaufrufe
  - ◆ Einfangen von Signalen

```
void (*signal( int sig, void (*disp)( int ) ))( int );
```

  - ◆ Zustellen von Signalen

```
int kill( pid_t pid, int sig );
```

4 UNIX Signale (4)

- Signalsemantik unterschiedlich bei verschiedenen UNIX Systemen
  - **System V:** Rücksetzen der Signalbehandlung beim Einfangen eines Signals
    - Beim Einfangen eines Signals wird implizit **`signal( ..., SIG_DFL)`** aufgerufen
    - Im Signalhandler muß der Handler selbst wieder eingesetzt werden
    - kurze Zeitspanne ohne Signalhandler
  - **System VR4:** Unterbrechung von Systemaufrufen
    - Fast alle „langsamen“ Systemaufrufe können durch die Signalbehandlung unterbrochen werden
    - **`errno`** wird auf **`EINTR`** gesetzt und der Systemaufruf terminiert mit **`-1`**
  - **BSD, Posix:** Blockieren weiterer Signale während der Behandlung
    - Beim Einfangen werden weitere gleichartige Signale blockiert (maximal wird ein Signal gespeichert)
    - Sobald die Behandlung fertig ist, wird die Blockierung wieder freigegeben

4 UNIX Signale (5)

- Moderne UNIX Systeme implementieren alle Variationen
  - Systemaufruf statt **`signal`**:

```
int sigaction( int sig, const struct sigaction *act,
               struct sigaction *oact );
```

  - Rücksetzen auf Defaulthandler einstellbar
  - Liste von Signalen einstellbar, die beim Einfangen eines Signals blockiert werden soll
  - Automatischer Wiederanlauf von unterbrochenen Systemaufrufen einstellbar

★ **Wichtig:** Sie müssen die Semantik der Signalbehandlung auf dem entsprechenden UNIX System kennen!

4 UNIX Signale (6)

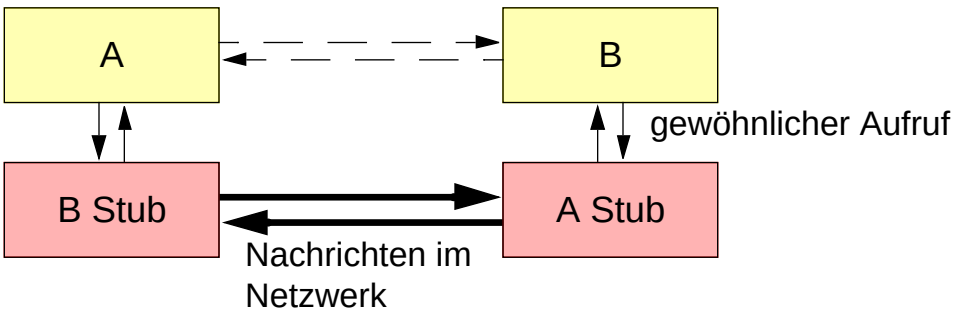
- Unterbrechung von Systemaufrufen
  - Wenn kein automatischer Wiederanlauf nach einer Unterbrechung durchgeführt wird, muß der Anwender auf den Fehler **EINTR** reagieren.

```
...
cnt= write( fd, buf, 100 );
...
```

```
...
do {
    cnt= write( fd, buf, 100 );
}
while( cnt < 0 && errno == EINTR );
...
```


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



6 Gemeinsamer Speicher

- Zwei Prozesse können auf einen gemeinsamen Speicherbereich zugreifen
  - gemeinsame Variablen und Datenstrukturen (ähnlich wie bei Threads des selben Prozesses)
- Näheres erst im Abschnitt E.5

D.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;
```

D.6 Koordinierung (2)

- Effekte:
  - ◆ Fahrzeuge gehen „verloren“
  - ◆ Fahrzeuge werden doppelt gezählt
- Ursachen:
  - ◆ Befehle in C werden nicht atomar abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden.
  - ◆ Fahrzeuge gehen „verloren“:
    Nach dem Drucken wird der Protokollierer unterbrochen. Beobachter zählt weitere Fahrzeuge. Anzahl wird danach ohne Beachtung vom Protokollierer auf Null gesetzt.
  - ◆ 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.

D.6 Koordinierung (3)

◆ Fahrzeuge gehen „verloren“:

Observer

```
cnt= cnt+1;
cnt= cnt+1;
```

Logger

```
printf( "Count= %d\n", cnt );

cnt= 0;
```

◆ Fahrzeuge werden doppelt gezählt:

Observer

```
mov DS:$cnt, %r1
add $1, %r1

mov %r1, DS:$cnt

[cnt= cnt+1;]
```

Logger

```
printf( "Count= %d\n", cnt );
cnt= 0;
```

D.6 Koordinierung (4)

- 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

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 */
}
```

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

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 */
}
```

1 Gegenseitiger Ausschluß (4)

- Gegenseitiger Ausschluß wird erreicht
- Probleme der Lösung
  - aktives Warten
  - Verklemmung möglich *(Lifelock)*

P_0

```
ready0= TRUE;

while( ready1 );
```

P_1

```
ready1= TRUE;

while( ready0 );
```

- Kombination beider Algorithmen führt zu Versuch 3

1 Gegenseitiger Ausschluß (5)

- 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 */
}
```

1 Gegenseitiger Ausschluß (6)

- 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

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

▲ 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 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

5 Semaphore

- 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 Semaphore (2)

- Implementierung kritischer Abschnitte mit Semaphore

```
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

5 Semaphore (3)

- Implementierung im Betriebssystem (Monoprozessor)

P-Operation

Unterbrechungen sperren	
ja: **`*s <= 0`**	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

- jede Semaphore besitzt Warteschlange, die blockierte Prozesse aufnimmt

5 Semaphore (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

5 Semaphore (5)

★ Vorteile einer Semaphore-Implementierung im Betriebssystem

- ◆ Einbeziehen des Schedulers in die Semaphore-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 Semaphore (6)

- Abstrakte Beschreibung von zählenden Semaphore (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: für zählende Semaphore

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

D.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

1 Gegenseitiger Ausschluß

- Semaphore
  - eigentlich reicht eine Semaphore mit zwei Zuständen: binäre Semaphore

```
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)

- 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

1 Gegenseitiger Ausschluß (3)

- Automatische Klammerung wünschenswert
  - Beispiel: Java

```
synchronized( lock ) {

... /* critical sec. */

}
```


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, der als Ringpuffer wirkt
    - zwei Integer-Variablen enthalten Feldindizes auf den Anfang und das Ende des Ringpuffers
  - eine Semaphore für den gegenseitigen Ausschluß
  - je eine Semaphore für das Blockieren an den Bedingungen „Puffer voll“ und „Puffer leer“
    - Semaphore **`full`** zählt wieviele Zeichen noch in den Puffer passen
    - Semaphore **`empty`** zählt wieviele Zeichen im Puffer sind

2 Bounded Buffers (3)

```
char buffer[N];
int inslot= 0, outslot= 0;
semaphore 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
  - ◆ Zählen der nebenläufig tätigen Leser: Variable **`readcount`**
  - ◆ Semaphore für gegenseitigen Ausschluß beim Zugriff auf **`readcount`**: **`mutex`**
  - ◆ Semaphore für gegenseitigen Ausschluß von Schreibern untereinander und von Schreibern gegen Leser: **`write`**

3 Erstes Leser-Schreiber-Problem (2)

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

Leser

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

... /* reading */

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

Schreiber

```
...
P( &writer );

... /* writing */

V( &writer );
...
```

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 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`**
  - Semaphore für gegenseitigen Ausschluß beim Zugriff auf **`readcount`**: **`mutexR`**
  - Semaphore für gegenseitigen Ausschluß beim Zugriff auf **`writecount`**: **`mutexW`**
  - Semaphore für gegenseitigen Ausschluß von Schreibern untereinander und von Schreibern gegen Leser: **`write`**
  - Semaphore für den Ausschluß von Lesern, falls Schreiber vorhanden: **`read`**
  - Semaphore zum Klammern des Leservorspanns: **`mutex`**

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 );
...
```

4 Zweites Leser-Schreiber-Problem (3)

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

4 Zweites Leser-Schreiber-Problem (4)

- Implementierung mit Up-down–Semaphore:

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

Leser

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

... /* reading */

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

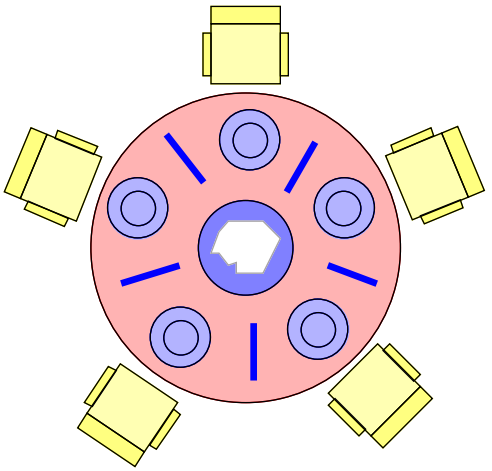
Schreiber

```
...
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)
- ◆ Semaphore für gegenseitigen Ausschluß der Schreiber: **`mutexw`**

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

5 Philosophenproblem (2)

- Naive Implementierung
  - eine Semaphore pro Gabel

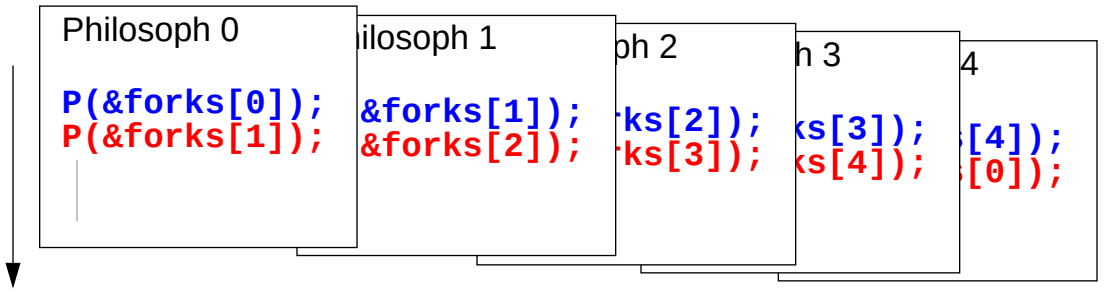
```
semaphore forks[5]= { 1, 1, 1, 1, 1 };
```

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] );
}
```

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

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 Semaphore
  - Abstrakte Beschreibung:

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

5 Philosophenproblem (5)

◆ Implementierung mit PV-multiple–Semaphore

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

Philosoph i, i ∈ [0,4]

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

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

    ... /* eat */

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


5 Philosophenproblem (6)

■ Lösung 2: einer der Philosophen muß erst die andere Gabel aufnehmen

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

Philosoph i, i ∈ [0,3]

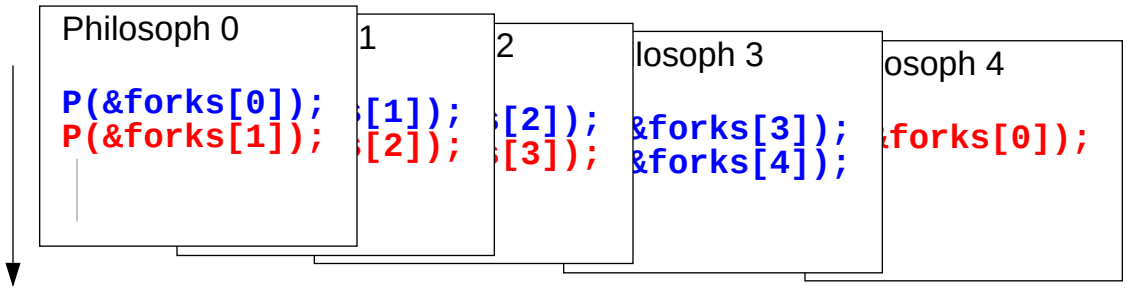
```
while( 1 ) {
    ... /* think */

    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] );
}
```

5 Philosophenproblem (7)

- Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- System verklemmt sich nicht

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`**

6 Schlafende Friseure (2)

- Implementierung mit zählenden Semaphoren (PV System)

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

Barber

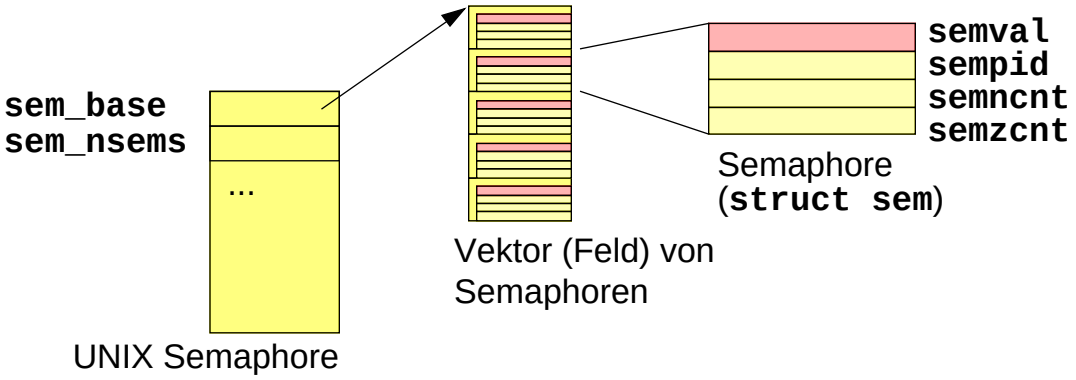
```
while( 1 ) {
    P( &customers );
    P( &mutex );
    waiting--;
    V( &barbers );
    V( &mutex );

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

Customer

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

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


D.8 UNIX Semaphore

- Vektor von Semaphoren (erweitertes Vektoradditionssystem)

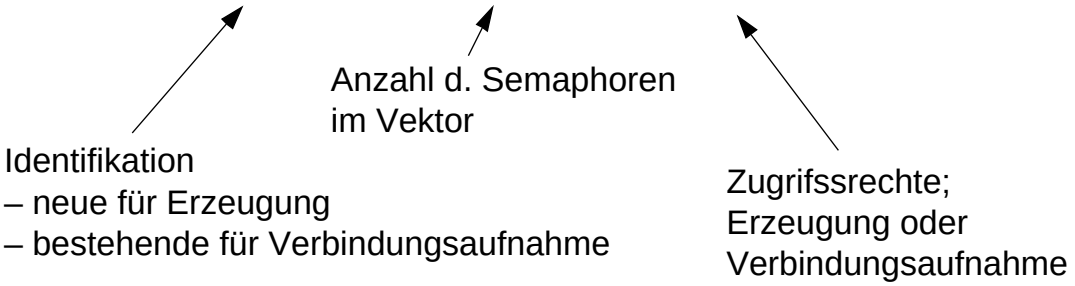


- Gleichzeitige und atomare Operationen auf mehreren Semaphoren im Vektor möglich

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 );
```



  - 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)

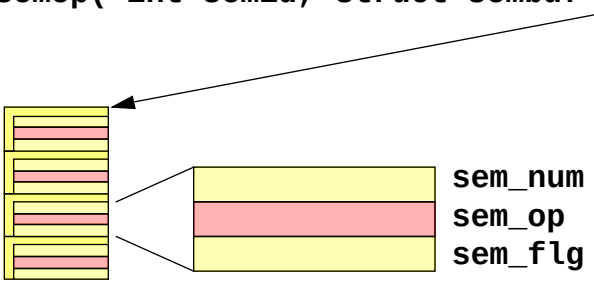
1 Erzeugen einer UNIX Semaphore (2)

- 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

2 Operationen auf UNIX Semphoren

- Operationen auf mehreren der Semaphoren im Vektor

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



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

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

3 Beispiel: Philosophenproblem

- Eine UNIX Semaphore 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;                          /* semaphore ID */
struct sembuf pbuf[2], vbuf[2];  /* operation buffer */

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

...
```


3 Beispiel: Philosophenproblem (2)

- Erzeuge Semaphore

```
...

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 */
    }
}

...
```

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; */

...
```


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;

...
```

3 Beispiel: Philosophenproblem (5)

- Philosoph

```
...

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

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

    ...     /* eating */

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


D.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