7.1 Gegenseitiger Ausschluss

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

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

atomare Funktion

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

atomare Funktion

  - zum Teil effizienter implementierbar

7.1 Gegenseitiger Ausschluss (2)

- Abstrakte Beschreibung: binäre Semaphore

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

7.1 Gegenseitiger Ausschluss (3)

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

```
P( &lock );

... /* critical sec. */

P( &lock );
```

führt zu Verklemmung (Deadlock)

```
V( &lock );

... /* critical sec. */

V( &lock );
```

führt zu unerwünschter Nebenläufigkeit

7.1 Gegenseitiger Ausschluss (3)

- Automatische Klammerung wünschenswert
  - Beispiel: Java

```
synchronized( lock ) {

... /* critical sec. */

}
```

7.2 Bounded Buffers

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

7.2 Bounded Buffers (2)

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

7.2 Bounded Buffers (3)

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

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

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

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


7.3 Erstes Leser-Schreiber-Problem

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

7.3 Erstes Leser-Schreiber-Problem (2)

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

Leser

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

... /* reading */

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

Schreiber

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

... /* writing */

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


7.3 Erstes Leser-Schreiber-Problem (3)

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

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

7.3 Erstes Leser-Schreiber-Problem (4)

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

```
PV_chunk_semaphore mutex= N;
```

Leser

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

... /* reading */

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

Schreiber

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

... /* writing */

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


7.4 Zweites Leser-Schreiber-Problem

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

7.4 Zweites Leser-Schreiber-Problem (2)

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

Bitte nicht versuchen, dies zu verstehen!!

Leser

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

... /* reading */

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

Schreiber

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

... /* writing */

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


7.4 Zweites Leser-Schreiber-Problem (3)

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

7.4 Zweites Leser-Schreiber-Problem (4)

- Abstrakte Beschreibung für Up-down–Semaphore

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

7.4 Zweites Leser-Schreiber-Problem (5)

- Implementierung mit Up-Down–Semaphoren:

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

Leser

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

... /* reading */

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

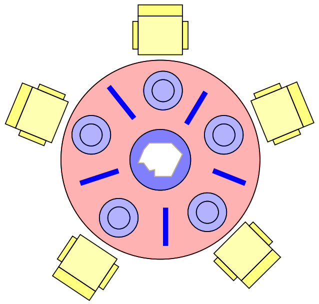
Schreiber

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

... /* writing */

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

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

7.5 Philosophenproblem

- Fünf Philosophen am runden Tisch



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

▲ Problem

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

7.5 Philosophenproblem (2)

- Naive Implementierung
  - ◆ eine Semaphor pro Gabel

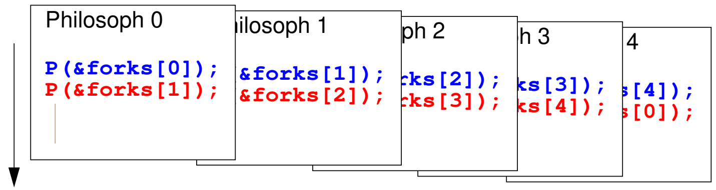
```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, i ∈ [0,4]

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

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

    ... /* eat */

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


7.5 Philosophenproblem (3)

- Problem der Verklemmung
  - ◆ alle Philosophen nehmen gleichzeitig die linke Gabel auf und versuchen dann die rechte Gabel aufzunehmen



- ◆ System ist verklemmt
  - • Philosophen warten alle auf ihre Nachbarn

7.5 Philosophenproblem (4)

- Lösung 1: gleichzeitiges Aufnehmen der Gabeln
  - ◆ Implementierung mit binären oder zählenden Semaphoren ist nicht trivial
  - ◆ Zusatzvariablen erforderlich
  - ◆ unübersichtliche Lösung

★ Einsatz von speziellen Semaphoren: PV-multiple–Semaphore

- ◆ gleichzeitiges und atomares Belegen mehrerer Semaphoren
- ◆ Abstrakte Beschreibung:

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

7.5 Philosophenproblem (5)

◆ Implementierung mit PV-multiple–Semaphoren

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

Philosoph i, i ∈ [0,4]

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

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

    ... /* eat */

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


7.5 Philosophenproblem (6)

- Lösung 2: <u>einer</u> der Philosophen muss erst die andere Gabel aufnehmen

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

Philosoph i, i ∈ [0,3]

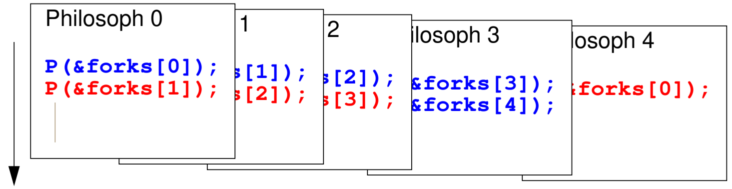
```
while( 1 ) {
    ... /* think */

    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] );
}
```


7.5 Philosophenproblem (7)

◆ Ablauf der asymmetrischen Lösung im ungünstigsten Fall



◆ System verklemmt sich nicht

7.6 Schlafende Friseure

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

7.6 Schlafende Friseure (2)

- Implementierung mit zählenden Semaphoren (PV System)

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

Barber

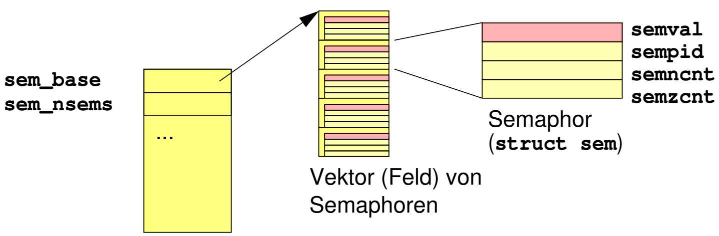
```
while( 1 ) {
     P( &customers );
     P( &mutex );
     waiting--;
     V( &barbers );
     V( &mutex );

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

Customer

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

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


8 UNIX-Semaphor

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



UNIX Semaphor

- ◆ Gleichzeitige und atomare Operationen auf mehreren Semaphoren im Vektor möglich

8.1 Erzeugen einer UNIX-Semaphore

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

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

Identifikation
– neue für Erzeugung
– bestehende für Verbindungsaufnahme

Anzahl d. Semaphoren
im Vektor

Zugriffssrechte;
Erzeugung oder
Verbindungsaufnahme

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

8.1 Erzeugen einer UNIX-Semaphore (2)

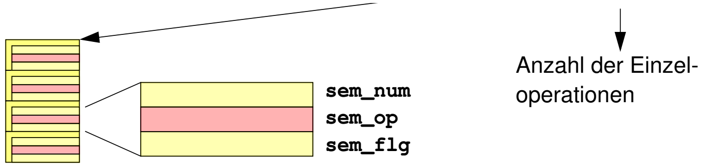
- Verwendung des Keys
  - ◆ Alle Prozesse, die auf die Semaphore zugreifen wollen, müssen den Key kennen
  - ◆ Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - ◆ Ist ein Key bereits vergeben, kann keine Semaphore mit gleichem Key erzeugt werden
  - ◆ Ist ein Key bekannt, kann auf die Semaphore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

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

8.2 Operationen auf UNIX-Semphoren

- Operationen auf mehreren der Semaphoren im Vektor

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



◆ Operationen

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

8.2 Operationen auf UNIX-Semaphoren (2)

- Kontrolloperationen

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

◆ explizites Setzen von Werten (einen, alle)

◆ Abfragen von Werten (einen, alle)

◆ Abfragen von Zusatzinformationen

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

◆ Löschen des Semaphor

8.3 Beispiel: Philosophenproblem

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

◆ Deklarationen

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

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

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

...
```


8.3 Beispiel: Philosophenproblem (2)

◆ Erzeuge Semaphor

```
...

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

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

...
```

8.3 Beispiel: Philosophenproblem (3)

◆ Erzeugen der Prozesse

```
...

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

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

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

...
```


8.3 Beispiel: Philosophenproblem (4)

◆ Initialisierungen

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

/* initialize buffer for P operation */

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

/* initialize buffer for V operation */

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

...
```


8.3 Beispiel: Philosophenproblem (5)

◆ Philosoph

```
...

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

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

    ...     /* eating */

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


9 Zusammenfassung

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

9 Zusammenfassung (2)

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

9 Zusammenfassung (3)

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