

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;
```

```
bool key;          Prozeß 0
...
while( 1 ) {
    key= TRUE;
    while( key == TRUE )
        swap( &lock, &key );

    ... /* critical sec. */

    lock= FALSE;

    ... /* uncritical */
}
```

```
bool key;          Prozeß 1
...
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 Betriebssystem

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 */
}
```

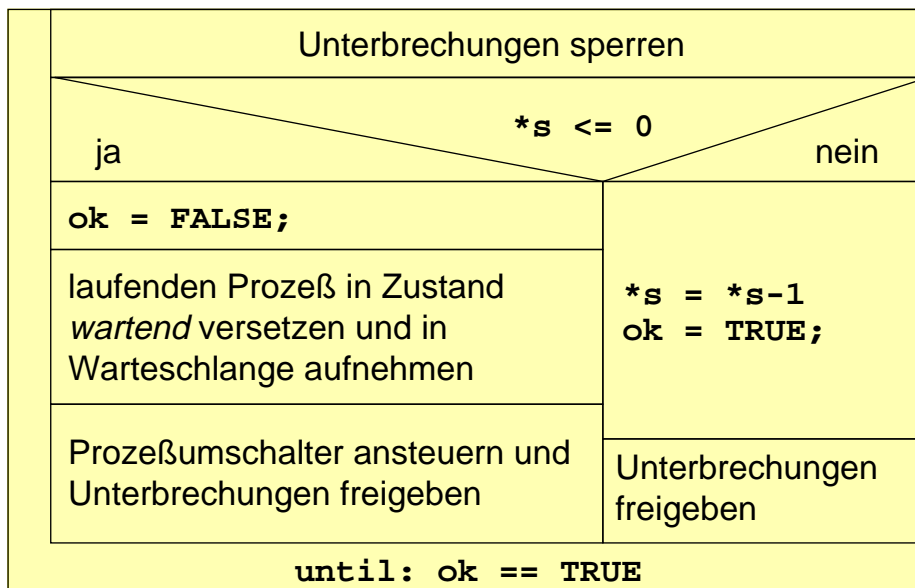
▲ Problem:

◆ Implementierung von P und V

5 Semaphore (3)

■ Implementierung im Betriebssystem (Monoprozessor)

P-Operation



`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 <i>bereit</i> 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 Wartezeit 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

SP I

Systemprogrammierung I

© Franz J. Hauck, Universität Erlangen-Nürnberg, IMMD IV, 1997

D-Proc.doc 1997-11-25 15.44

D.74

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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	$S := S + 1$

SP I

Systemprogrammierung I

© Franz J. Hauck, Universität Erlangen-Nürnberg, IMMD IV, 1997

D-Proc.doc 1997-11-25 15.44

D.75

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

D.6 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= N, full= 0;
```

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

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

    P( &full );
    P( &mutex );
    c= buffer[outslot];
    if( ++outslot >= N )
        outslot= 0;
    V( &mutex );
    V( &empty );
    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 **writcount**
- ◆ Semaphore für gegenseitigen Ausschluß beim Zugriff auf **readcount**:
mutexR
- ◆ Semaphore für gegenseitigen Ausschluß beim Zugriff auf **writcount**:
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 );
...
```

SP I

Systemprogrammierung I

© Franz J. Hauck, Universität Erlangen-Nürnberg, IMMD IV, 1997

D-Proc.doc 1997-11-25 15.44

D.88

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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
$up(S, \{ S_i \})$	$\sum_i S_i \geq 0$	$S := S + 1$
$down(S, \{ S_i \})$	$\sum_i S_i \geq 0$	$S := S - 1$

SP I

Systemprogrammierung I

© Franz J. Hauck, Universität Erlangen-Nürnberg, IMMD IV, 1997

D-Proc.doc 1997-11-25 15.44

D.89

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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 );  
...
```

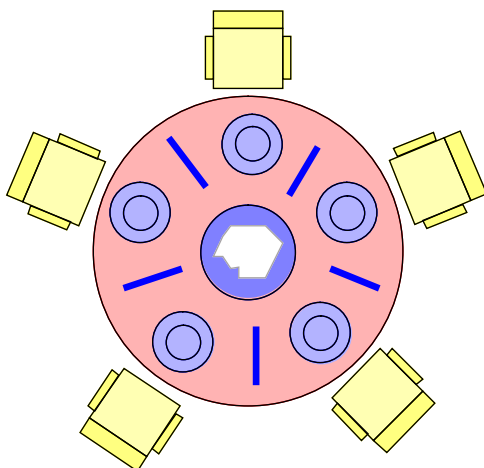
Schreiber

```
...  
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

- ◆ 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 };
```

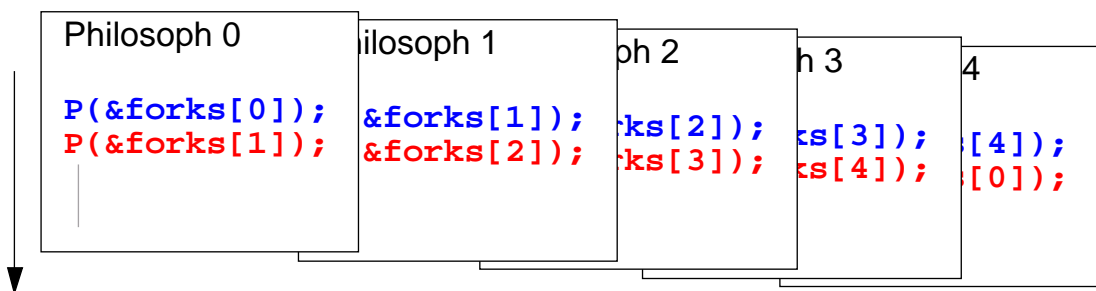
Philosoph i , $i \in [0,4]$

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

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

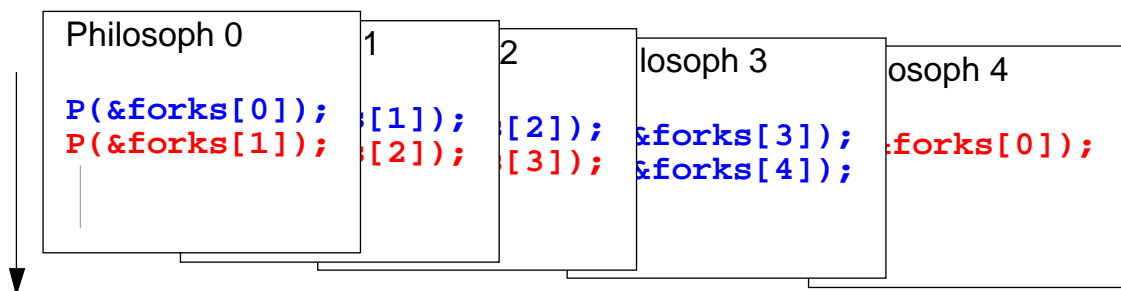
```
while( 1 ) {  
    ... /* think */  
  
    P( &forks[i] );  
    P( &forks[(i+1)%5] );  
  
    ... /* eat */  
  
    V( &forks[i] );  
    V( &forks[(i+1)%5] );  
}
```

Philosoph 4

```
while( 1 ) {  
    ... /* think */  
  
    P( &forks[0] );  
    P( &forks[4] );  
  
    ... /* eat */  
  
    V( &forks[0] );  
    V( &forks[4] );  
}
```

5 Philosophenproblem (7)

- ◆ Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- ◆ System verklemmt sich nicht

6 Schlafende Friseure

■ Firseurladen 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

■ 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 );
}
```