9.3 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	$S := S + k$

© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.3 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 );
...
```


© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.4 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`**

9.4 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 );
...
```

9.4 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

© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.4 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

© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.4 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 );
...
```

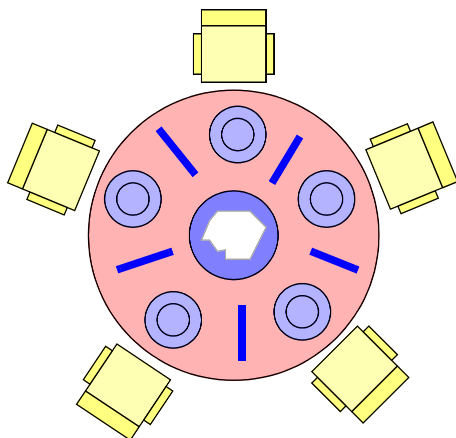
Schreiber

```
...
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**

9.5 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

9.5 Philosophenproblem (2)

- Naive Implementierung
  - ◆ eine Semaphor pro Gabel

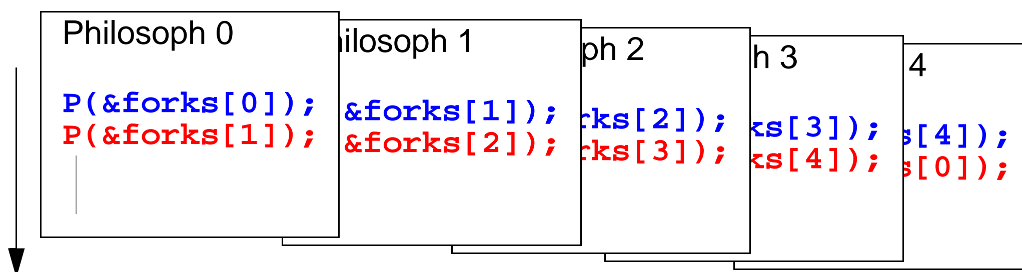
```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

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] );
}
```


9.5 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

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

© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.5 Philosophenproblem (5)

- Implementierung mit PV-multiple–Semaphoren

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

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

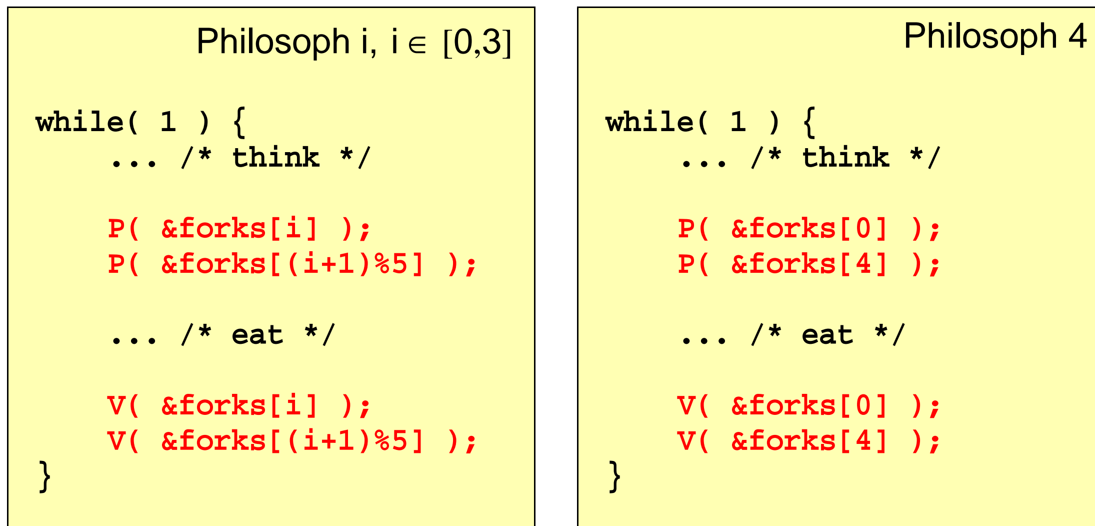
```
while( 1 ) {
     ... /* think */

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

     ... /* eat */

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


© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.5 Philosophenproblem (6)

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

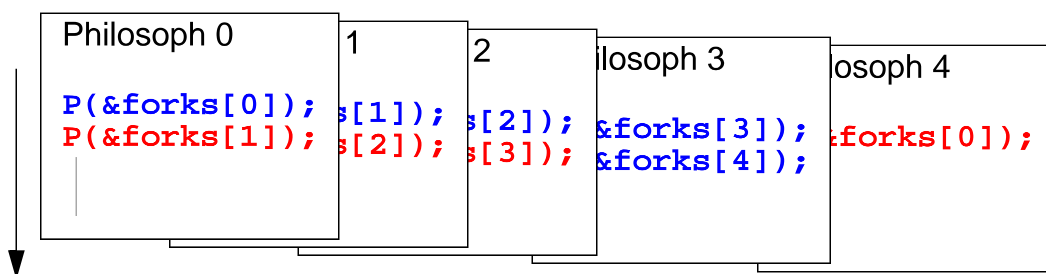
```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```



© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

9.5 Philosophenproblem (7)

- Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- System verklemmt sich nicht

© 1997-2003, Franz J. Hauck, Verteilte Systeme, Univ. Ulm
Reproduktion oder Verwendung dieser Unterlage bedarf in jedem Fall der Zustimmung des Autors.

10 Monitore

- Einfaches Konzept zur Koordinierung (B. Hansen/Hoare 1975)
  - Ziel: Vermeidung von Koordinierungsfehlern durch falsche Platzierung von Semaphoren
- Monitor als programmiersprachliches Konstrukt mit
  - privaten Variablen
  - Prozeduren (nur im gegenseitigen Ausschluss aufrufbar)
  - Condition-Variables zum Blockieren
    - Aufruf von `wait(c)` führt zur Blockade an Condition-Variable `c`
    - Aufruf von `signal(c)` führt zur Deblockade eines Prozesses blockiert an Condition-Variable `c`
    - Blockieren und Deblockieren garantiert dennoch gegenseitigen Ausschluss

10.1 Beispiel: Erzeuger-Verbraucher

```
monitor
{
    char buffer[N];
    int inslot= 0, outslot= 0, count= 0;
    condition notFull, notEmpty;

    void put( char c )              char get( void )
    {                               {
        while( count == N )             char c;
            wait( notFull );
                                        while( count == 0 )
        buffer[inslot]= c;                  wait( notEmpty );
        inslot= (inslot+1)%N;
        count++;                        c= buffer[outslot];
        signal( notEmpty );             outslot= (outslot+1)%N;
    }                                   count--;
                                        signal( notFull );
                                        return c;
}                                   }
```

10.2 Beispiel: Java

- Threaderzeugung

```
class MyThread extends Thread
{
    public void run()
    {
        ... /* thread code */
    }
}
```

```
...
    Thread t= new MyThread();
...
```

◆ weitere Möglichkeiten ohne Vererbung (siehe Java-Dokumentation)

10.2 Beispiel: Java (2)

```
class buffer
{
    char buffer[]= new char[N];
    int inslot= 0, outslot= 0, count= 0;

    synchronized public
    void put( char c )
    {
        while( count == N )
            try { wait(); }
            catch( ... ) {}

        buffer[inslot]= c;
        inslot=(inslot+1)%N;
        count++;
        notifyAll();
    }

    synchronized public
    char get( void )
    {
        char c;

        while( count == 0 )
            try { wait(); }
            catch( ... ) {}

        c= buffer[outslot];
        outslot= (outslot+1)%N;
        count--;
        notifyAll();
        return c;
    }
}
```

10.2 Beispiel: Java (3)

- Jedes Objekt stellt Monitor dar
  - nur für **`synchronized`** Methoden
  - expliziter Eintritt in den Monitor (lock) durch **`synchronized( obj ) { ... }`**
- Nur eine implizite Condition-Variable
  - **`wait`** und **`notify`** statt **`wait`** und **`signal`**
  - **`notifyAll`** deblockiert alle am Objekt (Monitor) blockierten Threads

★ Näheres: siehe Java-Dokumentation