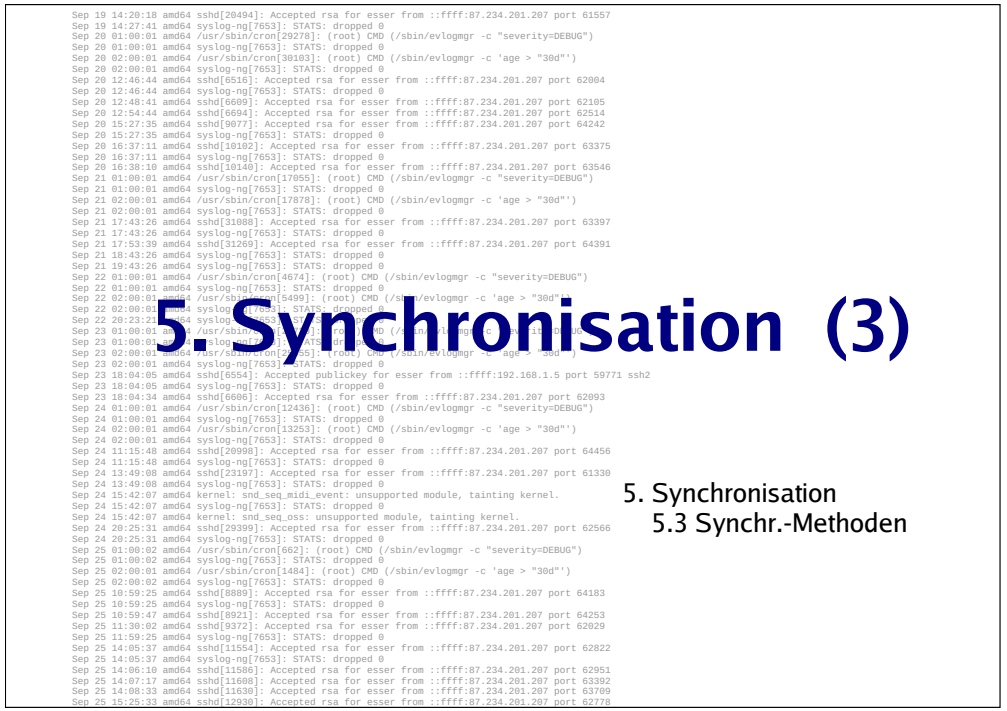
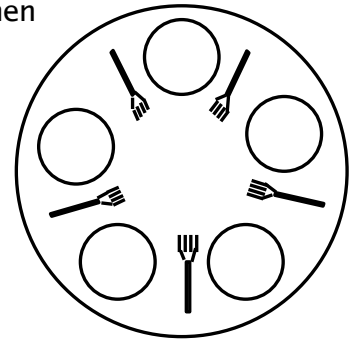


5. Synchronisation (3)

5. Synchronisation
5.3 Synchr.-Methoden

Philosophenproblem (1)

- Fünf Philosophen an einem Tisch (Dijkstra 1965)
  - Jeder Philosoph hat vor sich einen Teller Spaghetti.
  - Zwischen je zwei Tellern liegt eine Gabel.
  - Jeder Philosoph wechselt ab zwischen Denken und Essen.
  - Zum Essen benötigt ein Philosoph die beiden Gabeln (bzw. Eßstäbchen) rechts und links von seinem Teller.



- Aufgabe: – Philosophen nicht verhungern lassen
  – und maximale Parallelität erreichen

Philosophenproblem (2)

- Erste Idee:

```
#define  N  5                    // Anzahl der Philosophen
philosopher (int i) {            // i: Nummer des Philosophen (0...N-1)
  while (TRUE)  {
    think();                     // Der Philosoph denkt
    take_fork(i);                // Linke Gabel aufnehmen
    take_fork( (i+1)%N );        // Rechte Gabel aufnehmen (%=modulo)
    eat();                       // Der Philosoph isst
    put_fork (i);                // Linke Gabel zurücklegen
    put_fork ( (i+1)%N );        // Rechte Gabel zurücklegen
  }
}
```

  - take_fork(i) blockiert den Thread, bis die Gabel frei ist
  - Lösung ist falsch: Deadlock, wenn alle Philosophen gleichzeitig die linke Gabel nehmen

Philosophenproblem (3)

- 1. Korrekturversuch:
  - Jeder Philosoph prüft (nach dem Aufnehmen der linken Gabel), ob die rechte verfügbar ist
  - Ist die rechte Gabel nicht da, legt er seine linke zurück, wartet kurz und nimmt sie wieder
- Wenn die Wartezeit zufällig ist, könnte das „oft“ funktionieren (nicht gut genug)
- Bei gleicher Wartezeit könnte Endlosschleife enstehen (alle Phil. nehmen linke Gabel, legen sie wieder hin, nehmen sie wieder etc.) ➡ „starvation“

Philosophenproblem (4)

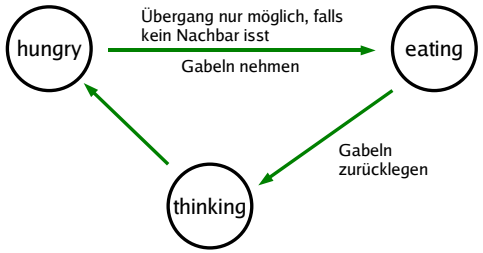
- 2. Korrekturversuch:
  - Ganzer Block vom Aufnehmen der ersten Gabel bis zum Ablegen aller Gabeln durch Mutex schützen, also

```
while (TRUE)  {
    think();
    wait (mutex);      // Anfang kritischer Bereich
    take_fork(i);
    take_fork( (i+1)%N );
    eat();
    put_fork (i);
    put_fork ( (i+1)%N );
    signal (mutex);    // Ende kritischer Bereich
  }
```

  - OK, aber nicht effizient: Es kann immer nur ein Philosoph essen – fünf Gabeln reichen für zwei

Philosophenproblem (5)

- Korrekte Lösung:
  - Zustände der Philosophen in Array state[ ] speichern



  - Semaphor sem[i] für jeden Philosophen: blockiert, wenn nicht beide Gabeln verfügbar sind

Philosophenproblem (6)

```
#define N 5                 // Anzahl der Philosophen
#define LEFT  (i-1)%N       // Nummer des linken Nachbarn
#define RIGHT (i+1)%N       // Nummer des rechten Nachbarn
#define THINKING 0          // der Philosoph denkt
#define HUNGRY   1          // Philosoph versucht, Gabeln zu nehmen
#define EATING   2          // der Philosoph isst

typedef int semaphore;      // Semaphore sind besondere "int"
int       state[N];         // Vektor für Zustände
semaphore mutex=1;          // Semaphor zum gegenseitigen Ausschluss
                            // beim Zugriff auf den Vektor state
semaphore sem[N]={0};       // ein Semaphor pro Philosoph

philosopher (int i)         // i: welcher Philosoph (0 bis N-1)
{
  while (TRUE)  {           // Endlosschleife
    think ();               // der Philosoph denkt
    take_forks (i);         // nimm beide Gabeln auf oder blockiere
    eat ();                 // iss die Nudeln
    put_forks (i);          // lege beide Gabeln zurück
  }
}
```


Philosophenproblem (7)

```
take_forks (int i) {     // i: welcher Philosoph (0 bis N-1)
  wait (mutex);          // kritischen Abschnitt betreten
  state[i] = HUNGRY;     // der Philosoph ist hungrig
  test (i);              // versuche beide Gabeln zu bekommen
  signal (mutex);        // Kritischen Abschnitt verlassen
  wait (sem[i]);         // Blockieren, falls Gabeln nicht frei
}

put_forks (int i) {      // i: welcher Philosoph (0 bis N-1)
  wait (mutex);          // kritischen Abschnitt betreten
  state[i] = THINKING;   // der Philosoph beendet das Essen
  test (LEFT);           // Testen, ob linker Phil. essen will + kann
  test (RIGHT);          // Testen, ob rechter Phil. essen will + kann
  signal (mutex);        // Kritischen Abschnitt verlassen
}

test (int i) {           // Testen, ob Philosoph i essen will + kann
  if ( state[i]     == HUNGRY &&
       state[LEFT]  != EATING &&
       state[RIGHT] != EATING ) {
    state[i] = EATING;
    signal (sem[i]);     // Philosoph i kann jetzt essen, darum wecken
  }
}
```

Beispiel: Phil. 2 und 3 wollen essen

```
// i=2, sem[2]=0

think();
take_forks (2);
   wait (mutex);
   state[2] = HUNGRY;
   test (2);
      state[2] == HUNGRY?  yes
      state[1] != EATING?  yes
      state[3] != EATING?  yes
      ->
      state[2] = EATING;
      signal (sem[2]);   // sem[2]=1
   signal (mutex);
   wait (sem[2]);   // Sem. ist 1, wird 0
eat ();
put_forks (2);
   wait (mutex);
   state[2] = THINKING;
   test (1);  // ggf. andere aufwecken
   test (3);
   signal (mutex);
loop...
```

```
// i=3, sem[3]=0

think();
take_forks (3);
   wait (mutex);
   state[3] = HUNGRY;
   test (3);
      state[3] == HUNGRY?  yes
      state[2] != EATING?  NO !!
      state[4] != EATING?  yes
      -> nichts tun (kein signal()-Aufruf)
   signal (mutex);
   wait (sem[3]);   // ist 0, blockieren!

   [blockiert, solange Phil. 2 isst]

eat ();
put_forks (3);
   wait (mutex);
   state[3] = THINKING;
   test (2);  // ggf. andere aufwecken
   test (4);
   signal (mutex);
loop...
```


Monitore (1)

Motivation

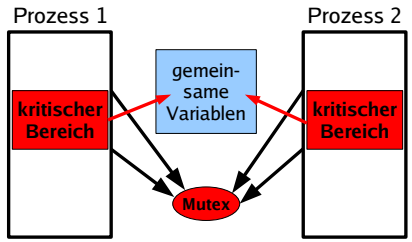
- Arbeit mit Semaphoren und Mutexen zwingt den Programmierer, vor und nach jedem kritischen Bereich wait() und signal() aufzurufen
- Wird dies ein einziges Mal vergessen, funktioniert die Synchronisation nicht mehr
- **Monitor** kapselt die kritischen Bereiche

Monitore (2)

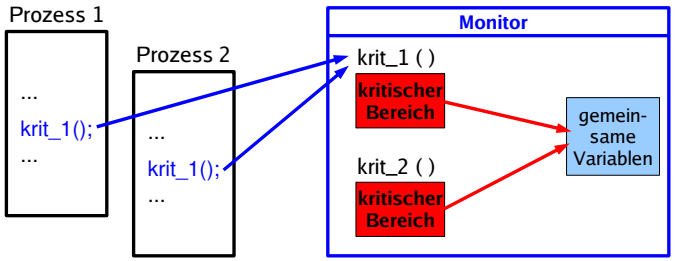
- **Monitor:** Sammlung von Prozeduren, Variablen, speziellen Bedingungsvariablen und Datenstrukturen:
  - Prozesse können die Prozeduren des Monitors aufrufen, können aber nicht von außerhalb des Monitors auf dessen Datenstrukturen zugreifen.
  - Zu jedem Zeitpunkt kann nur ein einziger Prozess aktiv im Monitor sein (d. h.: eine Monitor-Prozedur ausführen).
- Monitor wird durch Verlassen der Monitorprozedur frei gegeben

Monitore (3)

Mutex



Monitor



Monitore (4)

Einfaches Beispiel: Zugriff auf eine Festplatte; mit Mutex

```
mutex disk_access = 1;
```

```
wait (disk_access);
// Daten von der Platte lesen
signal (disk_access);

wait (disk_access);
// Daten auf die Platte schreiben
signal (disk_access);
```

Gleiches Beispiel, mit Monitor

```
monitor disk {
  entry read (diskaddr, memaddr) {
    // Daten von der Platte lesen
  };
  entry write (diskaddr, memaddr) {
    // Daten auf die Platte schreiben
  };
  init () {
    // Gerät initialisieren
  };
};
```

```
disk.read (da, ma);

disk.write (da, ma);
```


Monitor (5)

- Monitor ist ein Konstrukt, das Teil einer Programmiersprache ist
- Compiler – und nicht der Programmierer – ist für gegenseitigen Ausschluss zuständig
- Umsetzung (durch den Compiler) z. B. mit Semaphor/Mutex:

```
- monitor disk          ➞  semaphore m_disk = 1;

- entry funktion () {   ➞  void funktion () {
    /* Code */                wait (m_disk);
  }                           /* Code */
                              signal (m_disk);
                            }

- disk.funktion();      ➞  funktion();
```


Monitor (6)

- Monitor-Konzept erinnert an
  - Klassen (objektorientierte Programmierung)
  - Module (modulare Programmierung)
- Kapselung der Prozeduren und Variablen (außer über als public deklarierte Prozeduren kein Zugriff auf Monitor)
- Einfaches und übersichtliches Verfahren, um kritische Bereiche zu schützen, aber:
- Busy waiting ➞ Schlafen/Wecken wäre besser

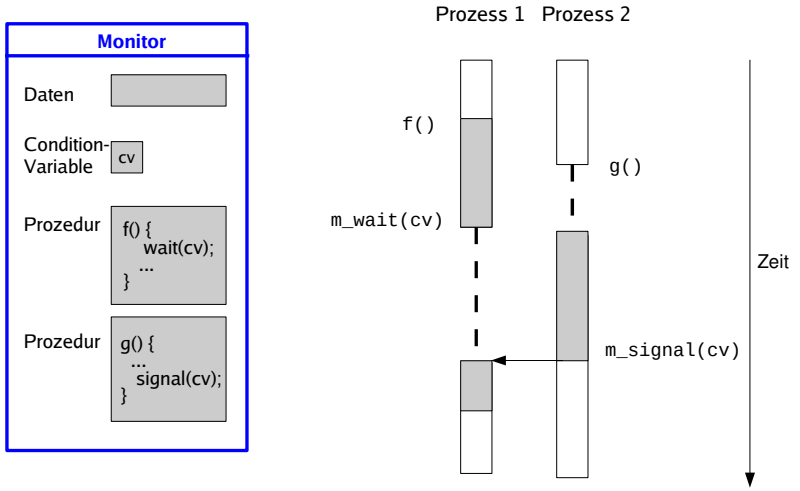
Monitor (7)

Zustandsvariablen (condition variables)

Für jede Zustandsvariable Wait- und Signal-Funktionen:

- m_wait (var): aufrufenden Prozess sperren (er gibt den Monitor frei)
- m_signal (var): gesperrten Prozess entsperren (weckt einen Prozess, der den Monitor mit m_wait() verlassen hat);
  erfolgt unmittelbar vor Verlassen des Monitors

Monitore (8)



Monitore (9)

- Gesperrte Prozesse landen in einer Warteschlange, die der Zustandsvariable zugeordnet ist
- status (cv) gibt Anzahl der wartenden Prozesse zurück
- Interne Warteschlangen haben Vorrang vor Prozessen, die von außen kommen

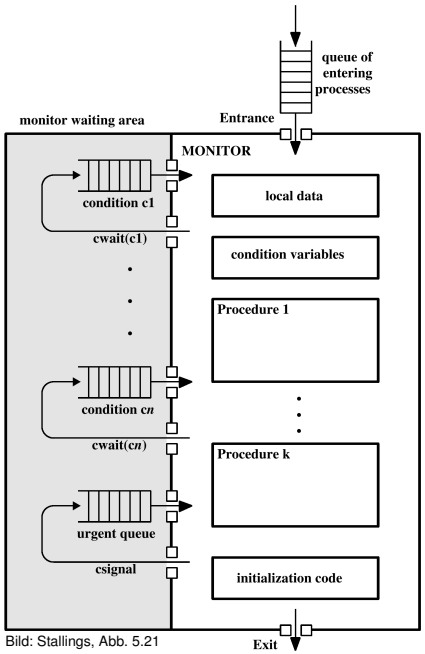


Bild: Stallings, Abb. 5.21

Monitore (10)

Producer-Consumer-Problem mit Monitor

```
monitor iostream {
  item buffer;
  int count;
  condition nonempty, nonfull;

  entry append(item x) {
    if (count == 1) m_wait(nonfull);
    put(buffer, x); // put ist lokale Prozedur
    count = 1;
    m_signal(nonempty);
  }

  entry remove(item x) {
    if (count == 0) m_wait(nonempty);
    get(buffer, x); // get ist lokale Prozedur
    count = 0;
    m_signal(nonfull);
  }

  init() {
    count = 0;  // Initialisierung
  }
}
```

Quelle: Prof. Scheidig, Univ. Saarbrücken, http://hssun5.cs.uni-sb.de/lehrstuhl/WS0607/Vorlesung_Betriebssysteme/ - angepasst an C-artige Syntax

Java und Monitore (1)

- Java verwendet Monitore zur Synchronisation
- Schlüsselwort „synchronized“
- Klasse, in der alle Methoden synchronized sind, ist ein Monitor
- Keine benannten Zustandsvariablen
- Warteschlangen:
  - m_wait: wait
  - m_signal: notify (weckt einen Prozess)
    notifyAll (weckt alle Prozesse)

Java und Monitore (2)

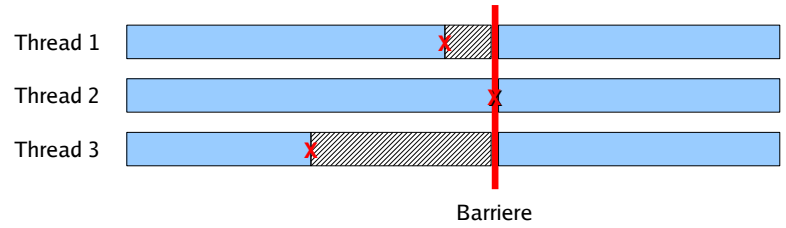
```
class BoundedBuffer extends MyObject {
   private int size = 0;
   private double[] buf = null;
   private int front = 0, rear = 0,
     count = 0;

   public BoundedBuffer(int size) {
      this.size = size;
      buf = new double[size];
   }

   public synchronized void
   deposit(double data) {
      while (count == size) wait();
      buf[rear] = data;
      rear = (rear+1) % size;
      count++;
      if (count == 1) notify();
   }

   public synchronized double fetch() {
      double result;
      while (count == 0) wait();
      result = buf[front];
      front = (front+1) % size;
      count--;
      if (count == size-1) notify();
      return result;
   }
}
```

Quelle: http://www.mcs.drexel.edu/~shartley/ConcProgJava/Monitors/bbse.java

Barrieren (1)

- Idee: Komplexere Berechnung in mehrere Phasen unterteilen
- Vor Eintritt in eine neue Phase warten alle Threads, bis jeder einzelne die alte Phase abgeschlossen hat
- Dann kann z.B. ein Austausch der Zwischenergebnisse erfolgen
- Schließlich setzen die Threads (unabhängig) ihre Berechnungen fort – bis zur nächsten Barriere

Barrieren (2)



- Threads rufen barrier()-Funktion auf und blockieren
- Erst wenn alle (Mitglieder einer Gruppe) barrier() aufgerufen haben, geht es weiter

Locking (1)

Locking erweitert die Funktionalität von Mutexen, indem es verschiedene Lock-Modi (Zugriffsarten) unterscheidet, und deren „Verträglichkeit“ miteinander festlegt:

- Concurrent Read: Lesezugriff, andere Schreiber erlaubt.
- Concurrent Write: Schreibzugriff, andere Schreiber erlaubt.
- Protected Read: Lesezugriff, andere Leser erlaubt, aber keine Schreiber (share lock)
- Protected Write: Schreibzugriff, andere Leser erlaubt, aber kein weiterer Schreiber (update lock)
- Exclusive: Schreibzugriff, keine anderen Zugriffe erlaubt

Locking (2)

	concurrent read	concurrent write	protected read	protected write	exclusive
concurrent read	X	X	X	X	-
concurrent write	X	X	-	-	-
protected read	X	-	X	-	-
protected write	X	-	-	-	-
exclusive	-	-	-	-	-

Locking (3)

- Thread fordert Lock in bestimmtem Modus an.
  - Ist der Lock-Modus mit den vorhandenen Locks anderer Threads verträglich, wird das Lock gewährt.
  - Ist der Lock-Modus zu einem Lock eines anderen Threads unverträglich, blockiert der Thread, bis das Lock gewährt werden kann.
- Locking-Mechanismen werden implementiert
  - vom Betriebssystem
  - von Anwendungsprogrammen (speziell Datenbanken)

Vorschau

Nächstes Mal:

- Rest von 5.3 (Synchr.-Methoden: Nachrichten)
- 5.4 Synchronisation unter Linux/Unix