

# Betriebssysteme Theorie

WS 2011/12

Hans-Georg Eßer  
Dipl.-Math., Dipl.-Inform.

Foliensatz D (11.11.2011)  
Synchronisation



## Einführung (2)

- Synchronisation: Probleme mit „gleichzeitigem“ Zugriff auf Datenstrukturen
- Beispiel: Zwei Threads erhöhen einen Zähler

```

erhoehe_zaehler( )   Ausgangssituation: w=10
{
  w=read(Adresse);   P1:                               P2:
  w=w+1;              w=read(Adresse); // 10
  write(Adresse,w);  w=w+1;          // 11
}
                                w=read(Adresse); // 10
                                w=w+1;          // 11
                                write(Adresse,w); // 11
write(Adresse,w); // 11 !!
    
```

Ergebnis nach P1, P2: w=11 – nicht 12!

## Einführung (1)

- Es gibt Prozesse (oder Threads oder Kernel-Funktionen) mit gemeinsamem Zugriff auf bestimmte Daten, z. B.
  - Threads des gleichen Prozesses: gemeinsamer Speicher
  - Prozesse mit gemeinsamer Memory-Mapped-Datei
  - Prozesse / Threads öffnen die gleiche Datei zum Lesen / Schreiben
  - SMP-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

## Einführung (3)

- Gewünscht wäre eine der folgenden Reihenfolgen:

<pre> Ausgangssituation: w=10 P1:                               P2: w=read(Adr); // 10 w=w+1;          // 11 write(Adr,w); // 11     </pre>	<pre> Ausgangssituation: w=10 P1:                               P2: w=read(Adr); // 10 w=w+1;          // 11 write(Adr,w); // 11     </pre>
<pre> w=read(Adr); // 11 w=w+1;          // 12 write(Adr,w); // 12     </pre>	<pre> w=read(Adr); // 11 w=w+1;          // 12 write(Adr,w); // 12     </pre>
Ergebnis nach P1, P2: w=12	Ergebnis nach P1, P2: w=12

## Einführung (4)

- Ursache: `erhoehe_zaebler()` arbeitet nicht **atomar**:
  - Scheduler kann die Funktion unterbrechen
  - Funktion kann auf mehreren CPUs gleichzeitig laufen
- Lösung: Finde alle Code-Teile, die auf gemeinsame Daten zugreifen, und stelle sicher, dass immer nur ein Prozess auf diese Daten zugreift (gegenseitiger Ausschluss, mutual exclusion)

## Einführung (6)

### Race Condition:

- Mehrere parallele Threads / Prozesse nutzen eine gemeinsame Ressource
- Zustand hängt von Reihenfolge der Ausführung ab
- Race: die Threads liefern sich „ein Rennen“ um den ersten / schnellsten Zugriff

## Einführung (5)

- Analoges Problem bei Datenbanken:

```
exec sql CONNECT ...
exec sql SELECT kontostand INTO $var FROM KONTO
      WHERE kontonummer = $knr
$var = $var - abhebung
exec sql UPDATE Konto SET kontostand = $var
      WHERE kontonummer = $knr
exec sql DISCONNECT
```

Bei parallelem Zugriff auf gleichen Datensatz kann es zu Fehlern kommen

- Definition der (Datenbank-) **Transaktion**, die u. a. **atomar und isoliert** erfolgen muss

## Einführung (7)

### Warum Race Conditions vermeiden?

- Ergebnisse von parallelen Berechnungen sind nicht eindeutig (d. h. potenziell falsch)
- Bei Programmtests könnte (durch Zufall) immer eine „korrekte“ Ausführreihenfolge auftreten; später beim Praxiseinsatz dann aber gelegentlich eine „falsche“.
- Race Conditions sind auch Sicherheitslücken

### Race Condition als Sicherheitslücke

- Wird von Angreifern genutzt
- Einfaches Beispiel: Primitive Shell

```
for (;;) {
  read(command);
  f = creat ("/tmp/script"); // Datei erzeugen
  write (f,command); // Befehl rein schreiben
  f.close (); // speichern/schließen
  chmod ("/tmp/script","a+x"); // ausführbar machen
  system ("/tmp/script"); // Skript ausführen
}
```

Annahme: Dateisystem ohne Zugriffsrechte (z. B. VFAT)

Angreifer ändert Dateiinhalt vor dem `chmod`; Programm läuft mit Rechten des Opfers

- Nicht alle Zugriffe sind problematisch:
  - Gleichzeitiges Lesen von Daten stört nicht
  - Prozesse, die „disjunkt“ sind (d. h.: die keine gemeinsamen Daten haben) können ohne Schutz zugreifen
- Sobald mehrere Prozesse/Threads/... gemeinsam auf ein Objekt zugreifen – und mindestens einer davon schreibend –, ist das Verhalten des Gesamtsystems **unvorhersehbar und nicht reproduzierbar**.

- Idee: Zugriff via Lock auf einen Prozess (Thread, ...) beschränken:

```
erhoehe_zaebler( ) {
  flag=read(Lock);
  if (flag == LOCK_UNSET) {
    set(Lock);
    /* Anfang des „kritischen Bereichs“ */
    w=read(Adresse);
    w=w+1;
    write(Adresse,w);
    /* Ende des „kritischen Bereichs“ */
    release(Lock);
  }
}
```

- Problem: Lock-Variable nicht geschützt

- Einführung, Race Conditions
- Kritische Abschnitte und gegenseitiger Ausschluss
- Synchronisationsmethoden, Standard-Primitive:
  - Mutexe
  - Semaphore
  - Monitore, Java: `synchronized`

## Kritische Bereiche (1)

- Programmteil, der auf gemeinsame Daten zugreift
  - Müssen nicht verschiedene Programme sein: auch mehrere Instanzen des gleichen Programms!
- Block zwischen erstem und letztem Zugriff
- Nicht den Code schützen, sondern die Daten
- Formulierung: kritischen Bereich „betreten“ und „verlassen“ (enter / leave critical section)

## Kritische Bereiche (3)

- Anforderung an parallele Threads:
  - Es darf maximal ein Thread gleichzeitig im kritischen Bereich sein
  - Kein Thread, der außerhalb kritischer Bereiche ist, darf einen anderen blockieren
  - Kein Thread soll ewig auf das Betreten eines kritischen Bereichs warten
  - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen kritischen Bereichen und blockieren sich gegenseitig)

## Kritische Bereiche (2)

- Bestimmen des kritischen Bereichs nicht ganz eindeutig:

```
void test () {
    z = global[i];
    z = z + 1;
    global[i] = z;
    // was anderes tun
    z = global[j];
    z = z - 1;
    global[j] = z;
}
```

- zwei kritische Bereiche oder nur einer?

## Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: **mutual exclusion**, kurz: mutex)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmierfehlern

## Test-and-Set-Lock (TSL) (1)

- **Maschineninstruktion** (z. B. mit dem Namen **TSL = Test and Set Lock**), die **atomic** eine Lock-Variable liest und setzt, also ohne dazwischen unterbrochen werden zu können.

```

enter:
    tsl register, flag ; Variablenwert in Register kopieren und
                        ; dann Variable auf 1 setzen
    cmp register, 0    ; War die Variable 0?
    jnz enter         ; Nicht 0: Lock war gesetzt, also
    Schleife
    ret

leave:
    mov flag, 0       ; 0 in flag speichern: Lock freigeben
    ret
    
```

## Aktives / passives Warten (1)

- **Aktives Warten (busy waiting):**
  - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
  - Der **Thread ist bereit** und **belegt die CPU**.
  - Die Variable muss von einem anderen Thread gesetzt werden.
    - (Großes) Problem, wenn der andere Thread endet.
    - (Großes) Problem, wenn der andere Thread – z. B. wegen niedriger Priorität – nicht dazu kommt, die Variable zu setzen.

## Test-and-Set-Lock (TSL) (2)

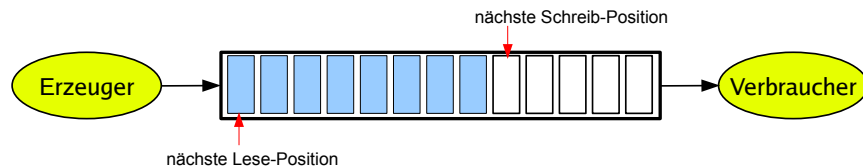
- **TSL** muss zwei Dinge leisten:
  - Interrupts ausschalten, damit der Test-und-Setzen-Vorgang nicht durch einen anderen Prozess unterbrochen wird
  - Im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU (deren Interrupts nicht gesperrt sind!) auf die gleiche Variable zugreifen kann

## Aktives / passives Warten (2)

- **Passives Warten (sleep and wake):**
  - Ein **Thread blockiert** und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit“ versetzt.
  - Blockierter Thread **verschwendet keine CPU-Zeit**.
  - Ein anderer Thread muss das Eintreten des Ereignisses bewirken.
    - (Kleines) Problem, wenn der andere Thread endet.
  - Bei Eintreten des Ereignisses muss der blockierte Thread geweckt werden, z. B.
    - explizit durch einen anderen Thread,
    - durch Mechanismen des Betriebssystems.

## Erzeuger-Verbraucher-Problem (1)

- Beim **Erzeuger-Verbraucher-Problem** (producer consumer problem, bounded buffer problem) gibt es zwei kooperierende Threads:
  - Der Erzeuger speichert Informationen in einem **beschränkten Puffer**.
  - Der Verbraucher liest Informationen aus diesem Puffer.



11.11.2011 Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer Folie D-21

## Erzeuger-Verbraucher-Problem (3)

- Realisierung mit passivem Warten:
  - Eine gemeinsam benutzte Variable „count“ zählt die belegten Positionen im Puffer.
  - Wenn der Erzeuger eine Information einstellt und der Puffer leer war (count == 0), weckt er den Verbraucher; bei vollem Puffer blockiert er.
  - Wenn der Verbraucher eine Information abholt und der Puffer voll war (count == max), weckt er den Erzeuger; bei leerem Puffer blockiert er.

11.11.2011 Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer Folie D-23

## Erzeuger-Verbraucher-Problem (2)

- **Synchronisation**
  - **Puffer nicht überfüllen:**  
Wenn der Puffer voll ist, muss der Erzeuger warten, bis der Verbraucher eine Information aus dem Puffer abgeholt hat, und erst dann weiter arbeiten.
  - **Nicht aus leerem Puffer lesen:**  
Wenn der Puffer leer ist, muss der Verbraucher warten, bis der Erzeuger eine Information im Puffer abgelegt hat, und erst dann weiter arbeiten.

11.11.2011 Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer Folie D-22

## Erzeuger-Verbraucher-Problem mit sleep / wake

```

#define N 100 // Anzahl der Plätze im Puffer
int count = 0; // Anzahl der belegten Plätze im Puffer

producer () {
    while (TRUE) { // Endlosschleife
        produce_item (item); // Erzeuge etwas für den Puffer
        if (count == N) sleep(); // Wenn Puffer voll: schlafen legen
        enter_item (item); // In den Puffer einstellen
        count = count + 1; // Zahl der belegten Plätze inkrementieren
        if (count == 1) wake(consumer); // war der Puffer vorher leer?
    }
}

consumer () {
    while (TRUE) { // Endlosschleife
        if (count == 0) sleep(); // Wenn Puffer leer: schlafen legen
        remove_item (item); // Etwas aus dem Puffer entnehmen
        count = count - 1; // Zahl der belegten Plätze dekrementieren
        if (count == N-1) wake(producer); // war der Puffer vorher voll?
        consume_item (item); // Verarbeiten
    }
}
    
```

11.11.2011 Betriebssysteme-Theorie, WS 2011/12, Hans-Georg Eßer Folie D-24

## Deadlock-Problem bei sleep / wake (1)

- Das Programm enthält eine race condition, die zu einem Deadlock führen kann, z. B. wie folgt:
  - Verbraucher liest Variable count, die den Wert 0 hat.
  - Kontextwechsel zum Erzeuger.
  - Erzeuger stellt etwas in den Puffer ein, erhöht count und weckt den Verbraucher, da count vorher 0 war.
  - Verbraucher legt sich schlafen, da er für count noch den Wert 0 gespeichert hat (der zwischenzeitlich erhöht wurde).
  - Erzeuger schreibt den Puffer voll und legt sich dann auch schlafen.


## Deadlock-Problem bei sleep / wake (3)

- Lösungsmöglichkeit: Systemaufrufe *sleep* und *wake* verwenden ein „**wakeup pending bit**“:
    - Bei *wake()* für einen nicht schlafenden Thread dessen wakeup pending bit setzen.
    - Bei *sleep()* das wakeup pending bit des Threads überprüfen – wenn es gesetzt ist, den Thread nicht schlafen legen.
- Aber: Lösung lässt sich nicht verallgemeinern (mehrere zu synchronisierende Prozesse benötigen evtl. zusätzliche solche Bits)

## Deadlock-Problem bei sleep / wake (2)

- **Problemursache:**  
Wakeup-Signal für einen – noch nicht – schlafenden Prozess wird ignoriert
- Falsche Reihenfolge
- Weckruf „irgendwie“ für spätere Verwendung aufbewahren...

VERBRAUCHER	ERZEUGER
n=read(count);	..
..	produce_item();
..	n=read(count);
..	/* n=0 */
..	n=n+1;
..	write(n,count);
..	<b>wake</b> (VERBRAUCHER);
/* n=0 */	..
sleep();	..



## Semaphore (1)

- Ein **Semaphor** ist eine Integer- (Zähler-) Variable, die man wie folgt verwendet:
- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
  - Beim **Anfordern** eines Semaphors (P- oder **Wait-Operation**): P = (niederl.) probeer
    - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
    - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

## Semaphore (2)

- Bei **Freigabe** eines Semaphors (V- oder **Signal-Operation**): V = (niederl.) vrijgeven
  - einen Thread aus der Warteschlange wecken, falls diese nicht leer ist,
  - Semaphor-Wert um 1 erhöhen (wenn es keinen auf den Semaphor wartenden Thread gibt)
- Code sieht dann immer so aus:
 

```
wait (&sem);
/* Code, der die Ressource nutzt */
signal (&sem);
```
- in vielen Büchern: **P** (&sem), **V** (&sem)

## Semaphore (4)

- Pseudo-Code für Semaphor-Operationen

```
wait (sem) {
    if (sem>0)
        sem--;
    else
        BLOCK_CALLER;
}

signal (sem) {
    if (P in QUEUE(sem)) {
        wakeup (P);
        remove (P, QUEUE);
    }
    else
        sem++;
}
```

## Semaphore (3)

- Variante: Negative Semaphor-Werte
  - Semaphor zählt Anzahl der wartenden Threads
  - **Anfordern** (WAIT):
    - Semaphor-Wert um 1 erniedrigen
    - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert  $\leq 0$  ist.
  - **Freigabe** (SIGNAL):
    - Thread aus der Warteschlange wecken (falls nicht leer)
    - Semaphor-Wert um 1 erhöhen

## Mutexe (1)

- **Mutex**: boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert
  - true: Zugang erlaubt
  - false: Zugang verboten
- **blockierend**: Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert  $\rightarrow$  Warteschlange
- Bei Freigabe:
  - Warteschlange enthält Threads  $\rightarrow$  einen wecken
  - Warteschlange leer: Mutex auf true setzen



## Mutexe (2)

- **Mutex (mutual exclusion) = binärer Semaphor**, also ein Semaphor, der nur die Werte 0 / 1 annehmen kann. Pseudo-Code:

```

wait (mutex) {
    if (mutex==1)
        mutex=0;
    else
        BLOCK_CALLER;
}

signal (mutex) {
    if (P in QUEUE(mutex)) {
        wakeup (P);
        remove (P, QUEUE);
    }
    else
        mutex=1;
}
    
```

- Neue Interpretation: wait → lock  
signal → unlock
- Mutexe für exklusiven Zugriff (kritische Bereiche)

## Atomare Operationen

- Bei Mutexen / Semaphoren müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:

Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

## Blockieren?

- Betriebssysteme können Mutexe und Semaphoren **blockierend** oder **nicht-blockierend** implementieren
- blockierend:  
wenn der Versuch, den Zähler zu erniedrigen, scheitert  
→ warten
- nicht blockierend:  
wenn der Versuch scheitert  
→ vielleicht etwas anderes tun

## Warteschlangen

- Mutexe / Semaphore verwalten Warteschlangen (der Prozesse, die schlafen gelegt wurden)
- Beim Aufruf von signal() muss evtl. ein Prozess geweckt werden
- Auswahl des zu weckenden Prozesses ist ein ähnliches Problem wie die Prozess-Auswahl im Scheduler
  - FIFO: **starker** Semaphor / Mutex
  - zufällig: **schwacher** Semaphor / Mutex

# Erzeuger-Verbraucher-Problem mit Semaphoren und Mutexen

```

typedef int semaphore;
semaphore mutex = 1;           // Kontrolliert Zugriff auf Puffer
semaphore empty = N;          // Zählt freie Plätze im Puffer
semaphore full = 0;           // Zählt belegte Plätze im Puffer

producer() {
    while (TRUE) {            // Endlosschleife
        produce_item(item);   // Erzeuge etwas für den Puffer
        wait (empty);         // Leere Plätze dekrementieren bzw. blockieren
        wait (mutex);         // Eintritt in den kritischen Bereich
        enter_item (item);    // In den Puffer einstellen
        signal (mutex);       // Kritischen Bereich verlassen
        signal (full);        // Belegte Plätze erhöhen, evtl. consumer wecken
    }
}

consumer() {
    while (TRUE) {           // Endlosschleife
        wait (full);          // Belegte Plätze dekrementieren bzw. blockieren
        wait (mutex);         // Eintritt in den kritischen Bereich
        remove_item(item);    // Aus dem Puffer entnehmen
        signal (mutex);       // Kritischen Bereich verlassen
        signal (empty);       // Freie Plätze erhöhen, evtl. producer wecken
        consume_entry (item); // Verbrauchen
    }
}
    
```

# Monitore (2)

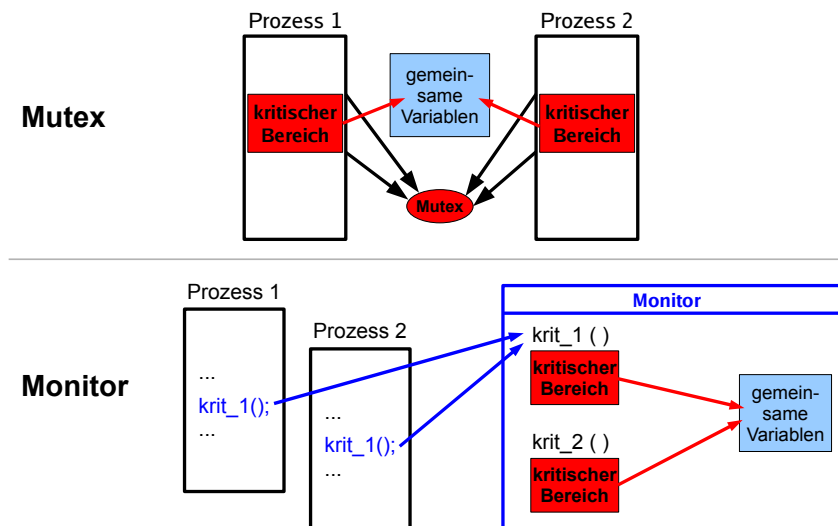
- **Monitor**: Sammlung von Prozeduren, Variablen und speziellen **Bedingungsvariablen**:
- Prozesse können die Prozeduren des Monitors aufrufen, können aber nicht von außerhalb des Monitors auf dessen Datenstrukturen zugreifen.
- Zu jedem Zeitpunkt kann **nur ein einziger Prozess aktiv im Monitor**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

# Monitore (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
- Monitor muss von Programmiersprache unterstützt werden (z.B. Java, Concurrent Pascal)

# Monitore (3)



## 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 (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:
- Was tun, wenn ein Prozess im Monitor blockieren muss?

## 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();
```

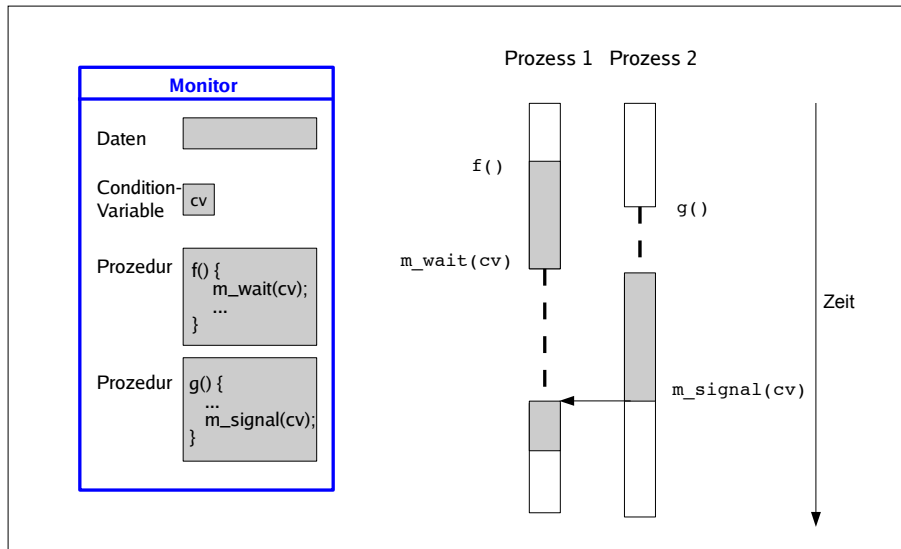
## Monitor (7)

### Zustandsvariablen (condition variables)

Idee: Prozess in Monitor muss darauf warten, dass eine bestimmte Bedingung (condition) erfüllt ist. 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 (10)

### Erzeuger-Verbraucher-Problem mit Monitor

```

monitor iostream {
    item buffer;
    int count;
    const int bufsize = 64;
    condition nonempty, nonfull;

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

    entry remove(item x) {
        while (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/lehstuhl/>  
WS0607/Vorlesung\_Betriebssysteme/  
- angepasst an C-artige Syntax

## Monitore (9)

- Gesperrte Prozesse landen in einer Warteschlange, die der Zustandsvariable zugeordnet ist
- Interne Warteschlangen haben Vorrang vor Prozessen, die von außen kommen
- Implementation mit Mutex/Semaphor:

```

conditionVariable {
    int queueSize = 0;
    mutex m;
    semaphore waiting;

    wait() {
        m.lock();
        queueSize++;
        m.release();
        waiting.down();
    }

    signal() {
        m.lock();
        while (queueSize > 0){
            // alle wecken
            queueSize--;
            waiting.up();
        }
        m.release();
    }
}
    
```

## 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>