Synchronisation

Gliederung

1. Einführung und Übersicht
2. Prozesse und Threads
3. Interrupts
4. Scheduling
5. **Synchronisation**
6. Interprozesskommunikation
7. Speicherverwaltung

Synchronisation

Übersicht:

- Einführung
- Race-Condition
- Kritische Abschnitte
- Mechanismen des BS und Standard-Primitive
- Tips zur praktischen Anwendung:
  - Deadlocks vermeiden
  - Mutex-Objekt

Einführung (1)

Wozu Synchronisation ?

- **Threads und z.T. Prozesse haben gemeinsamen Zugriff auf bestimmte Daten, z.B.**
  - Threads des gleichen Prozesses:
    - gemeinsamer virtueller Speicher
    - öffnen der gleichen Datei zum Lesen/Schreiben
  - Prozesse mit Shared-Memory (--> IPC)
  - SMP-System: Scheduler (je einer pro CPU):
    - Zugriff auf gleiche Prozesslisten / Warteschlangen
  - Datenbanken:
    - Zugriff über eine globale Datenbank-Verbindung (*DB-Connect*)

Einführung (2)

Beispiel: gleichzeitiger Zugriff auf Datenstruktur:

- **zwei Threads erhöhen einen gemeinsamen Zähler:**

```
erhoehe_zaehler( )
{
  w=read(Adresse);
  w=w+1;
  write(Adresse,w);
}
```

Ausgangssituation: w=10

P1:	P2:
w=read(Adresse); // 10	
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 (3)

Beispiel: gleichzeitiger Zugriff auf Datenstruktur:

- **gewünscht: eine der folgenden koordinierten Reihenfolgen:**

Ausgangssituation: w=10

P1:	**P2:**
w=read(Adr); // 10	
w=w+1; // 11	
write(Adr,w); // 11	
— — — — — — — — — — —	
	w=read(Adr); // 11
	w=w+1; // 12
	write(Adr,w); // 12

Ergebnis nach P1, P2: w=12

Ausgangssituation: w=10

P1:	**P2:**
	w=read(Adr); // 10
	w=w+1; // 11
	write(Adr,w); // 11
— — — — — — — — — — —	
w=read(Adr); // 11	
w=w+1; // 12	
write(Adr,w); // 12	

Ergebnis nach P1, P2: w=12

Einführung (4)

Beispiel: gleichzeitiger Zugriff auf Datenstruktur:

- **Ursache:**
  - `erhoehe_zaehler()` arbeitet nicht **atomar**:
    - Scheduler kann Funktion unterbrechen (anderer Thread arbeitet weiter)
    - Funktion kann auf mehreren CPUs gleichzeitig laufen
- **Lösung:** stelle sicher, dass
  - immer nur ein Prozess/Thread gleichzeitig auf gemeinsame Daten zugreift
  - bis Vorgang abgeschlossen ist
  - == gegenseitiger Ausschluß (*Mutual Exclusion*)

Einführung (5)

Beispiel: Datenbanken:

- **Datenbank: analoges Problem:**

```
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
```

  - paralleler Zugriff auf gleichen Datensatz potentiell fehlerhaft

- --> Definition der (Datenbank-) Transaktion, die

  - u.a. atomar und isoliert erfolgen muss

Race-Condition (1)

Eigenschaften:

- **Race Condition**
  - mehrere parallele Threads/Prozesse nutzen gemeinsame Ressource
  - Zustand hängt von Reihenfolge der Ausführung ab:
    - Ergebnis nicht vorhersagbar / nicht reproduzierbar
  - Race: Threads liefern sich ein „Rennen“ um den ersten/schnellsten Zugriff

Warum Race-Conditions vermeiden?

- Ergebnisse paralleler Berechnungen sind potentiell falsch
  - Programmtests können „funktionieren“, später die Anwendung aber versagen

Race-Condition (2)

Race-Condition als Sicherheitslücke

- Race-Conditions sind auch Sicherheitslücken
- wird vom Angreifer genutzt
- einfaches Beispiel:

```
read( command );
f = open( "/tmp/script", "w" );
write( f, command );
close( f );
chmod( "/tmp/script", "a+x" );
system( "/tmp/script" );
```

- Angreifer ändert Dateiinhalt vor dem chmod()

  --> Programm läuft mit Rechten des Opfers

Race-Condition (3)

Aspekte einer Lösung

- **Idee:**

  Zugriff via Flag / Lock auf Thread/Prozess beschränken:

```
erhoehe_zaehler() {
    flag = read( lock );
    if ( flag == LOCK_NOT_SET ) {
        set( lock );
        //Start kritischer Bereich
        w = read( adr );
        w = w + 1;
        //Ende kritischer Bereich
        write( adr, w );
        release( lock );
    }
}
```

- **Problem:** Lock-Variable nicht geschützt

- **kein Problem:**
  - gleichzeitiges Lesen von Daten
  - Threads, die “disjunkt“ sind, d.h.
    - keine gemeinsame Daten haben / nutzen
- **problematisch:**
  - mehrere Prozesse/Threads
  - greifen gemeinsam auf Objekt zu,
  - davon mindestens einer schreibend

Kritischer Bereich

Was ist ein „kritischer Bereich“ ?

- Programmteil, der auf gemeinsame Daten zugreift
- Block zwischen erstem und letztem Zugriff
- Formulierung: kritischen Bereich
  - betreten / verlassen (enter / *leave critical section*)

Anforderungen an parallele Threads:

- maximal ein Thread gleichzeitig in kritischem Bereich
- kein Thread außerhalb kritischem Bereich darf anderen blockieren (--> potentiell Deadlock)
- kein Thread soll ewig auf Freigabe eines kritischen Bereichs warten
- Deadlocks zu vermeiden, z.B.
  - zwei Threads in verschiedenen kritischen Bereichen blockieren sich gegenseitig

Gegenseitiger Ausschluß

- Gegenseitiger Ausschluß (**Mutual Exclusion**, kurz **Mutex**):
  - nie mehr als ein Thread betritt kritischen Bereich
  - es ist Aufgabe des Programmierers, dies zu garantieren
  - das BS bietet Mechanismen und Hilfsmittel, gegenseitigen Ausschluß durchzusetzen
  - Verantwortung für Fehlerfreiheit liegt beim Programmierer:
    - Compiler können i.d.R. Fehler aus Nebenläufigkeit nicht erkennen

Programmtechnische Synchronisation

- **Idee** war:
  - Zugriff via Flag / Lock auf Thread/Prozess beschränken
- **Problem:**
  - Lock-Variable nicht geschützt
- **Lösung:**
  - eine Lock-Variable **atomar testen und setzen**
  - dies kann per Programmlogik über mehrere Variable erreicht werden (--> Lit.: Dekker1966, Petersons Algorithmus)
  - ist kompliziert bei mehr als zwei Threads/Prozessen
- besser: Nutzung von „standard“ BS-Mechanismen

Synchronisation

Mechanismen und Standard-Primitive

Test-and-Set-Lock (TSL)

- **Maschineninstruktion (moderner CPUs)**
  - mit dem Namen **TSL = Test and Set Lock**
  - die atomar eine Lock-Variable liest (testet) und setzt, also garantiert ohne Unterbrechung
  - im Fall mehrerer CPUs:
    - TSL muss Speicherbus sperren, damit kein Thread auf anderer CPU in gleicher Weise zugreifen kann

```
enter:
    tsl register, flag  ; Variable in Register kopieren und
                        ; dann Variable auf 1 setzen
    cmp register, 0     ; war Variable 0 ?
    jnz enter           ; nicht 0: Lock war gesetzt, also Schleife(pollen)
    ret

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

Aktives und passives Warten (1)

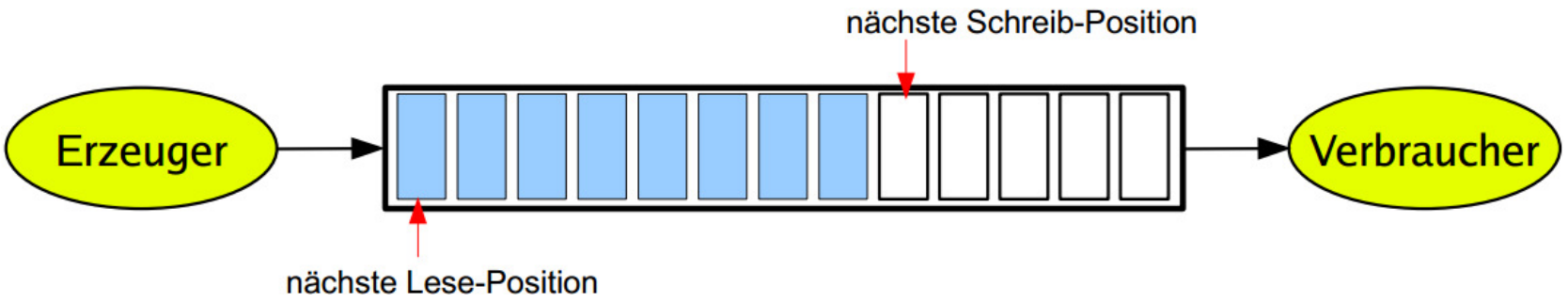
- **Aktives Warten (busy waiting):**
  - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt
  - Thread ist bereit und belegt die CPU
  - Variable muss von einem anderen Thread gesetzt werden
    - (großes) Problem, wenn der andere Thread endet
    - (großes) Problem, wenn anderer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

Aktives und passives Warten (2)

- **Passives Warten (sleep and wake):**
  - Thread blockiert und wartet auf Ereignis, das ihn in den Zustand „bereit“ versetzt
  - blockierter Thread verschwendet keine CPU-Zeit
  - anderer Thread muss Eintreten des Ereignisses bewirken
    - (kleines) Problem, wenn anderer Thread endet
  - bei Ereignis muss blockierter Thread geweckt werden
    - explizit durch anderen Thread
    - durch Mechanismen des BS

Erzeuger-Verbraucher-Problem (1)

- **Erzeuger-Verbraucher-Problem** (*producer-consumer problem, bounded-buffer problem*)
  - zwei kooperierende Threads:
    - Erzeuger speichert Informationspaket in beschränktem Puffer
    - Verbraucher liest Informationen aus diesem Puffer



Erzeuger-Verbraucher-Problem (2)

- **Synchronisation:**
  - Puffer nicht überfüllen:
    - wenn Puffer voll, muß Erzeuger warten, bis Verbraucher ein weiteres Paket abgeholt hat
  - nicht aus leerem Puffer lesen:
    - wenn Puffer leer, muß Verbraucher warten, bis Erzeuger ein weiteres Paket abgelegt hat
- **Realisierung mit passivem Warten:**
  - eine gemeinsame Variable „count“ zählt belegte Positionen im Puffer
  - wenn Erzeuger ein Paket einstellt und Puffer leer war (count == 0)
    --> wecken des Verbrauchers
  - wenn Verbraucher ein Paket abholt und Puffer voll war (count == max)
    --> wecken des Erzeugers

Erzeuger-Verbraucher 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 );             // Erzeuge etwas für den Puffer
      if (count == N) sleep();     // Wenn Puffer voll: schlafen legen
      enter( item );               // In den Puffer einstellen
      count = count + 1;           //Zahl belegter 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 belegter Plätze dekrementieren
      if (count == N-1) wake(producer);  //war der Puffer vorher voll?
      consume_item (item);         // Verarbeiten
   }
}
```

Deadlock-Problem bei sleep / wake (1)

- **Race-Condition im vorigen Programm --> potentieller Deadlock, z.B.**
  - Verbraucher liest Variable count, die den Wert 0 hat
  - Kontextwechsel zum Erzeuger:
    - Erzeuger stellt etwas in den Puffer,
    - erhöht count und
    - weckt Verbraucher, da count==0 war
  - Kontextwechsel zum Verbraucher:

    Verbraucher legt sich schlafen, da count==0 gelesen wurde
  - Erzeuger schreibt Puffer voll und legt sich auch schlafen

Deadlock-Problem bei sleep / wake (2)

- **Ursache des Problems:**
  - Wakeup-Signal für einen -- noch nicht -- schlafenden Prozess wird ignoriert
  - --> Weckaufruf „irgendwie“ aufbewahren
- **Lösungsmöglichkeit:**
  - Systemaufrufe sleep() und wake() verwenden „wakeup pending bit“
    - bei wake() für nicht schlafenden Thread dessen wakeup-pending-bit setzen
    - bei sleep() das wakeup-pending-bit des Threads prüfen und falls gesetzt, nicht schlafen legen
- aber:
  - Lösung lässt sich nicht verallgemeinern (mehrere Prozesse benötigen weitere Bits).

Standardprimitive zur Synchronisation

Welche Standardprimitive zur Synchronisation gibt es ?

- **Mutex** (mutual exclusion) = binärer Semaphor
- **Semaphor**
- **Event** (ähnlich Condition-Variable)
- **Monitor**
- **Locking**

Mutex (1)

- **Mutex** (mutual exclusion) = binärer Semaphor
  - zur Synchronisation eines kritischen Bereichs bzw. gemeinsamer Daten
  - kann nur 2 Zustände / Werte annehmen:
    - true / frei: Zugang erlaubt
    - false / gesperrt: Zugang gesperrt
  - Anforderungs- und Freigabe-Operationen:
    - Anforderung: wait() / lock() / get()
    - Freigabe: signal() / unlock() / release()
  - Anforderung:
    ein Thread, der eine bereits vergebene Mutex anfordert, blockiert --> Warteschlange
  - Freigabe:
    - Warteschlange enthält Threads --> einen wecken
    - Warteschlange leer --> Mutex auf true

Mutex (2)

- wait() und signal()-Operationen sind selbst kritische Abschnitte --> atomar realisiert
  - Implementierung als System-Calls und Verhinderung von Kontext-Wechseln (z.B. durch kurzzeitiges Ausschalten von Interrupts)

```
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;
}
```

Semaphor (1)

- **Semaphor**:
  - Integer- (Zähler-) Variable
  - mit festgelegtem Anfangswert N („Anzahl verfügbarer Ressourcen“)
  - Anforderung (wait()):
    - falls >= 1: Wert um 1 reduzieren
    - falls == 0: Thread blockieren --> Warteschlange
  - Freigabe (signal()):
    - falls Warteschlange nicht leer: einen Thread wecken
    - falls Warteschlange leer: Wert um 1 erhöhen

```
wait( &sem );
    // Kode, der die Ressource nutzt
signal( &sem );
```

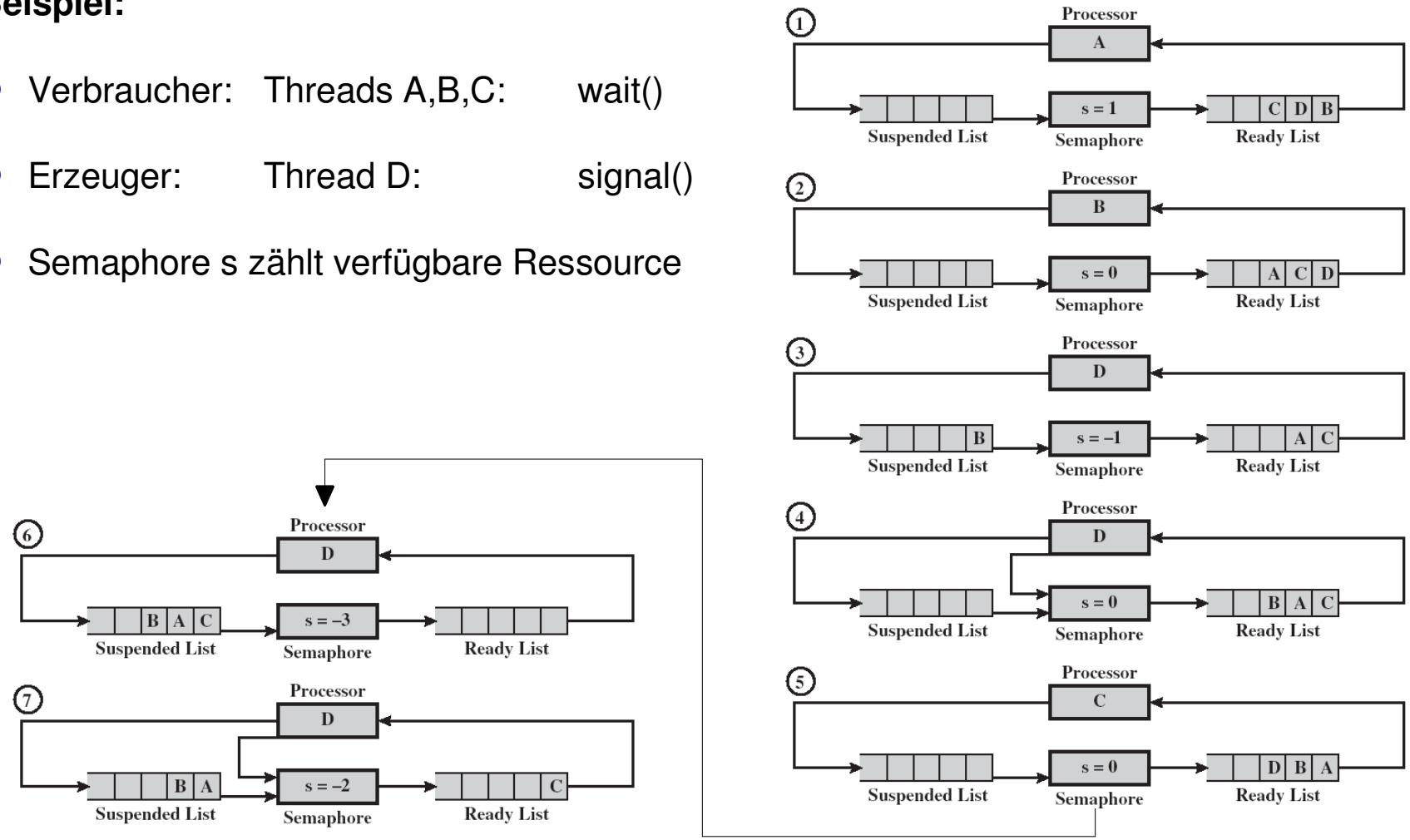
Semaphor (2)

- **Varianten:**
  - negative Semaphor-Werte:
    - Anforderung (wait()):
      - Wert immer um 1 reduzieren
      - --> Wert entspricht Zahl blockierter Threads in Warteschlange
    - Freigabe (signal()):
      - Wert immer um 1 erhöhen
  - nicht-blockierend: z.B. bool ret = try_lock();
  - pthreads-Semaphore:
    - sind auch zwischen Prozessen verwendbar (Standard: nur Threads)
  - SysV-IPC Semaphore:
    - arbeiten Prozessübergreifend
    - sind nach Prozessende noch vorhanden

Semaphor (3)

Beispiel:

- Verbraucher: Threads A,B,C: wait()
- Erzeuger: Thread D: signal()
- Semaphore s zählt verfügbare Ressource



Erzeuger-Verbraucher-Problem mit Semaphoren

```
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
    }  }
```

Events (1)

- **Events**:
  - kein Thread besitzt einen Event (<-> Mutex)
  - wait() blockiert, falls Event nicht im signalisierten Zustand

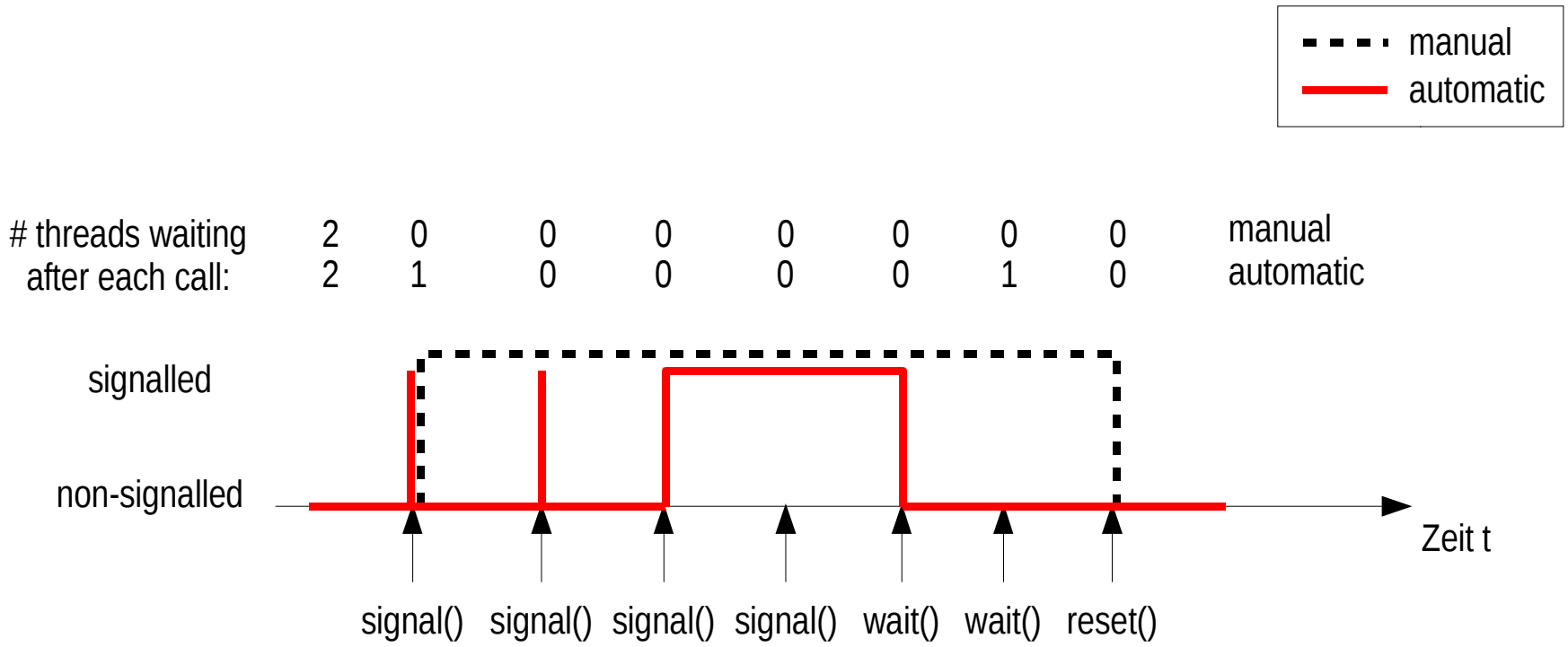
```
   thread A                thread B
       .                       .
event.wait()                   .
       .            event.signal();
       .                       .
```

t

  - automatischer Event:
    - jedes wait() setzt Event automatisch zurück (reset)
    - falls mehrere Threads warten:
      bei einem signal() kehrt nur genau ein Thread von seinem wait() zurück
  - manueller Event:
    - Event muß manuell zurückgesetzt werden (reset)
    - falls mehrere Threads warten:
      bei einem signal() kehren alle Threads von ihren wait() zurück

Events (2)

Beispiel:

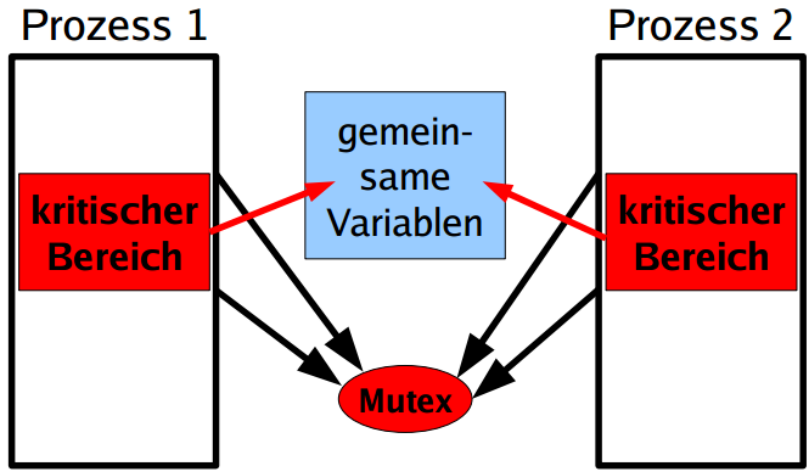


Monitor (1)

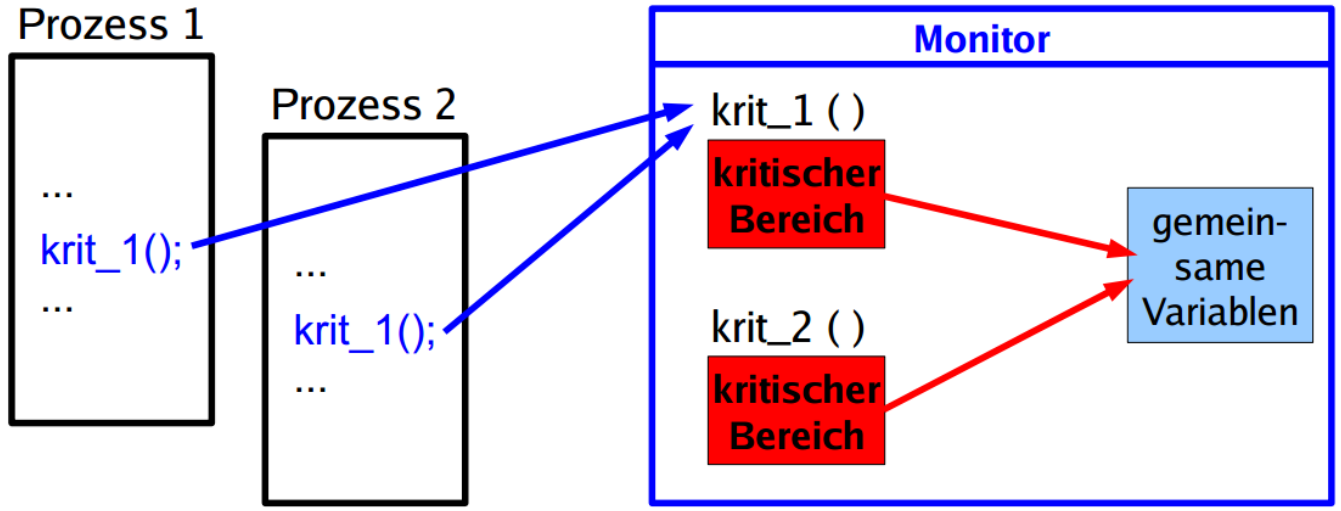
- Problem bei Arbeit mit Mutexen und Semaphoren:
  - Programmierer gezwungen, kritische Bereiche mit wait() und signal() abzusichern
  - schon bei einem Fehler --> Synchronisierung versagt
- **Monitor**:
  - muss von Programmiersprache unterstützt werden (z.B. Java, Concurrent-Pascal)
  - Sammlung von Prozeduren, Variablen, speziellen Bedingungsvariablen:
    - Prozesse können Prozeduren des Monitors aufrufen, aber
    - nicht auf dessen Datenstrukturen
  - kapselt kritische Bereiche
    - zu jedem Zeitpunkt nur ein einziger Prozess im Monitor aktiv (Monitor-Prozedur ausführen)
    - Freigabe durch Verlassen der Monitor-Prozedur

Monitor (2)

Mutex



Monitor



Monitor (3)

- Monitor-Konzept erinnert an
  - Klassen (Objektorientierung)
  - Module
- Kapselung: nur Zugriff über public-Prozeduren

Beispiel: Zugriff auf Festplatte mit Mutex:

```
mutex disc_access = 1;
```

```
wait( disc_access );
    // Daten lesen
signal( disc_access );

wait( disc_access );
    // Daten schreiben
signal( disc_access );
```

gleiches Beispiel: mit Monitor:

```
monitor disc {
  entry read( discaddr, memaddr ) {
    // Daten lesen
  };
  entry write( discaddr, memaddr ) {
    //Daten schreiben
  };
  init() {
    // Gerät initialisieren
  };
};
```

```
monitor disc;

disc.read( da, ma );

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

Monitor (4)

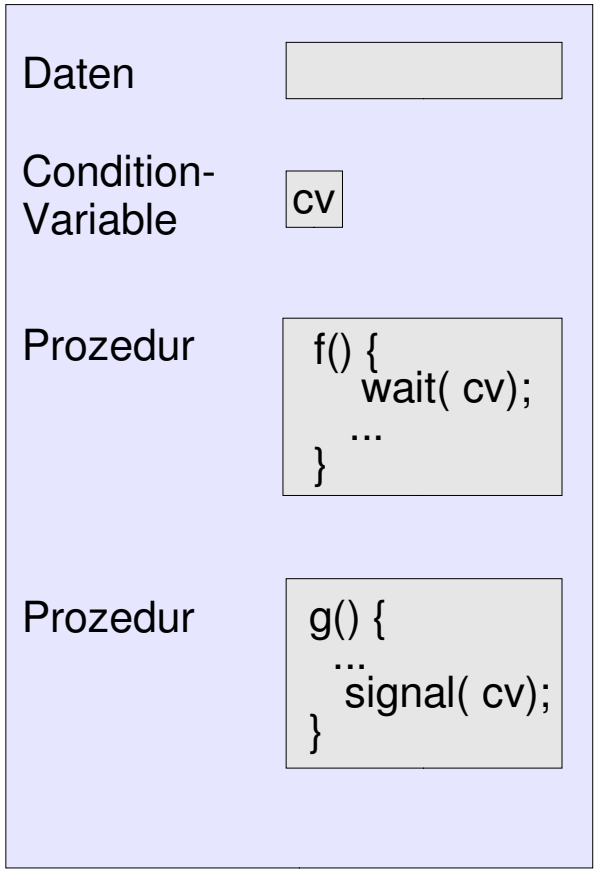
- Monitor wird von Programmiersprache / Compiler bereitgestellt
  - nicht der Programmierer ist für gegenseitigen Ausschluß verantwortlich
  - „Klasse“, bei der jede public-Methode synchronisiert ist

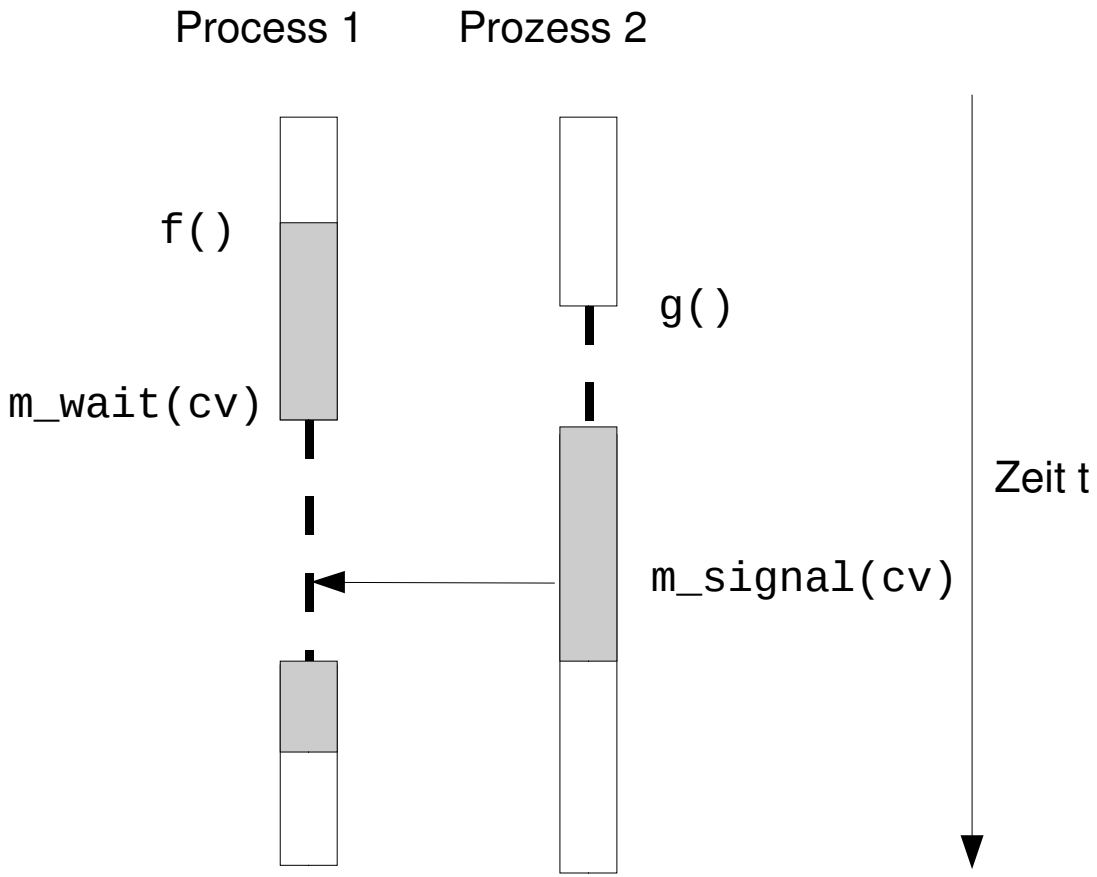
Was tun, wenn Prozess im Monitor blockieren muss ?

- **Condition-Variables** (Zustandsvariable):
  - **Idee:** Prozess muss auf Eintreten einer Bedingung (*Condition*) warten
  - **m_wait**(var): aufrufenden Prozess sperren (er gibt den Monitor frei)
  - **m_signal**(var):
    - gesperrte(n) Prozess(e) wecken
    - erfolgt unmittelbar vor Verlassen des Monitors

Monitor (5)

Monitor





Monitor (6)

- Gesperrte Prozesse landen in Warteschlange zu entsprechender Condition-Variable
- Interne Warteschlangen haben Vorrang vor Prozesszugriffen von außen
- Implementierung 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();
  }
};
```

Erzeuger-Verbraucher-Problem mit Monitor

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

    entry append( item x ) {
        while (count == N-1) m_wait(nonfull);
        put(buffer, x); //
        count += 1;
        m_signal(nonempty);
    }

    entry remove(item x) {
        while (count == 0) m_wait(nonempty);
        get(buffer, x); //
        count -= 1;
        m_signal(nonfull);
    }

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

Locking (1)

- **Locking**
  - erweitert Funktionalität von Mutexen
  - durch Unterscheidung verschiedener Lock-Modi (Zugriffsarten)
  - und Festlegung derer „Verträglichkeit“
    - 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: Lese/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	-	-	-	-	-

Praxisbeispiele

Praxis: Übersicht

- **Synchronisation im Linux-Kernel**
  - Atomare Operationen
    - auf Integer-Variablen
    - Bit-Operationen auf Bitvektoren
  - Spin-Locks
  - Reader-Writer-Locks
  - Semaphore / Reader-Writer Semaphore
- **Synchronisation in C++**
  - Threads, Mutexe
  - atomare Befehle
  - Mutex-Objekt in C++

Praxis / Linux-Kernel: Atomare Integer-Operationen

- Typ ***atomic_t*** (24 Bit Integer):
  - Initialisierung: **atomic_t** var = **ATOMIC_INIT**(0);
  - Wert setzen: **atomic_set**(&var, wert);
  - Addieren: **atomic_add**(wert, &var);
  - ++: **atomic_inc**(&var);
  - Subtrahieren: **atomic_sub**(wert, &var);
  - --: **atomic_dec**(&var);
  - Auslesen: int i = **atomic_read**(&var);
- res = **atomic_sub_and_test**(i, &var);
  - subtrahiert i atomar von var
  - return true, falls Ergebnis 0, sonst false
- res = **atomic_add_negative**(i, &var);
  - addiert i atomar zu var
  - return true, falls Ergebnis negativ, sonst false

Praxis / Linux-Kernel: Atomare Bit-Operationen

- Einzelne Bits in Bitvektoren setzen
- Datentyp: beliebig, z.B. unsigned long bitvektor = 0;
  - nur über Pointer anzusprechen
  - Anzahl der nutzbaren Bits abhängig vom verwendeten Datentyp
  - Test-and-Set-Operationen geben zusätzlich vorherigen Wert des jeweiligen Bits zurück

set_bit(i, &bv);	i-tes Bit setzen	b = **test_and_set_bit**(i, &bv);
clear_bit(i, &bv);	i-tes Bil löschen	b = **test_and_clear_bit**(i, &bv);
change_bit(i, &bv);	i-tes Bit kippen	b = **test_abd_change_bit**(i, &bv);

- Einzelne Bits auslesen:
  - b = **test_bit**(i, &bv);
- Suchfunktionen:
  - pos = **find_first_bit**(&bv, length);
  - pos = **find_first_zero_bit**(&bv, length);

Praxis / Linux-Kernel: Spin-Locks (1)

- **Spin-Lock:**
  - Lock mit Mutex-Funktion: gegenseitiger Ausschluß
  - Code, der Spin-Lock anfordert und nicht erhält
    - blockiert nicht (kein aufwendiger Wechsel in Kernel-Mode)
    - sondern läuft weiter (spinning), bis Lock verfügbar
  - nur zu verwenden
    - bei kurzen „Wartezeiten“ auf Lock,
    - bei Mehrprozessorsystemen
  - sind nicht „rekursiv“, also z.B. nicht in rekursiven Funktionen verwendbar

- Typ ***spinlock_t***

```
spinlock_t slock = SPIN_LOCK_UNLOCKED

spin_lock( &slock ):
    /* kritischer Abschnitt */
spin_unlock( &slock );
```

Praxis / Linux-Kernel: Spin-Locks (2)

- da Spin-Locks nicht schlafen/blockieren, sind diese in Interrupt-Handlern verwendbar
- in diesem Fall: zusätzlich Interrupts sperren:

```
spinlock_t slock = SPIN_LOCK_UNBLOCKED;
unsigned long flags;

spin_lock_irqsave( &slock, flags );          // aktuelle Interrupts sichern
    /* ktitischer Abschnitt */               // dann sperren
spin_unlock_irq_restore( &slock, flags );    // ursprüngl. Zustand restaurieren
```

- wenn zu Beginn alle Interrupts aktiviert sind, geht es auch einfacher:

```
spinlock_t slock = SPIN_LOCK_UNBLOCKED;

spin_lock_irq( &slock );                     // aktuelle Interrupts sperren
    /* ktitischer Abschnitt */
spin_unlock_irq( &slock );
```

Praxis / Linux-Kernel: Reader-Writer-Locks

- **Reader-Writer-Locks:**
  - Alternative zu normalen Locks, die mehrere Lesezugriffe zulässt,
  - aber bei schreibendem Zugriff exklusiv ist (wie normaler Lock)

	Es gibt schon einen Leser	Es gibt schon einen Schreiber	Noch keine Sperre
`read_lock(&lck)`	erfolgreich	schlägt fehl	erfolgreich
`write_lock(&lck)`	schlägt fehl	schlägt fehl	erfolgreich

- auch Varianten für Interrupt-Behandlung:

  - read_lock_irq / read_unlock_irq
  - read_lock_irqsave / read_unlock_irqrestore
  - write_lock_irq / write_unlock_irq
  - write_lock_irw_save / write_unlock_irqrestore

Praxis / Linux-Kernel: Semaphore (1)

- **Kernel-Semaphore**
  - sind „schlafende“ Locks
    - wartende Prozesse werden in Warteschlange gestellt, bei Freigabe wird erster geweckt
  - eignen sich für Sperren, die über längeren Zeitraum gehalten werden (<-> Spin-Lock)
  - sind nur im Prozess-Kontext, nicht in Interrupt-Handlern einsetzbar
    (Interrupt-Handler werden nicht vom Scheduler behandelt)
  - Code, der Semaphore verwendet, darf nicht bereits normalen Lock besitzen
    (Semaphore-Zugriff kann zum „schlafen-legen“ führen

Praxis / Linux-Kernel: Semaphore (2)

- Typ: ***semaphore***
  - statische Deklaration:

```
static DECLARE_SEMAPHORE_GENERIC( name, count );
static DECLARE_MUTEX( name );                          // count = 1
```

  - dynamische Deklaration:

```
sema_init( &sem, count );
init_MUTEX( &sem );                                    // count = 1;
```

  - Verwendung mit up() und down():

```
down( &sem );
    /* kritischer bereich */
up( &sem );
```

Praxis / Linux-Kernel: Semaphore (3)

- **Reader-Writer-Semaphore:**
  - analog zu Reader-Writer-Locks: Typ *rw_semaphore*, der spezielle Up- und Down-Operationen für Lese- und Schreibzugriffe erlaubt
  - alle RW-Semaphore sind Mutexe (bei Initialisierung count=1)

Lesender Code:

```
static DECLARE_RWSEM( rwsem );
init_rwsem( &rwsem );

down_read( &rwsem );
  //read-only Abschnitt
up_read( &rwsem );
```

Schreibender Code:

```
down_write( &rwsem );
  //lesen+schreiben-Abschnitt
up_write( &rwsem );
```

	Es gibt schon einen Leser	Es gibt schon einen Schreiber	Noch keine Sperre
down_read(&sem)	erfolgreich	schlägt fehl	erfolgreich
down_write(&sem)	schlägt fehl	schlägt fehl	erfolgreich

Praxis: Synchronisation in C++ (1)

- **Threads, Mutexe, usw.:**
  - **pthreads-Bibliothek (UL-Threads):**
    - quasi-Standard in Unix/Linux
    - wenig gebräuchlich unter WIndows (<-> Win-API)

    `#include <pthreads.h>`, linken mit `libpthreads`
  - **C++0x/C++11-Standard:**
    - Thread/Mutex-API: `#include <thread>`
  - **OpenMP**-Standard:
    - Parallelisierung von Schleifen mittels Threads

```
#include <omp.h>
#pragma omp parallel for num_threads(NCPU)
        for ( long y = ylow; y <= yhigh; ++y ) {...}
```

Praxis: Synchronisation in C++ (2)

Nachbildung eines **Monitors** mittels pthreads

```
struct ProducerConsumer {
   pthread_mutex_t count_mtx;    //zu initialisieren
   pthread_cond_t  full, empty;
   int             count;

   void enter() {
     pthread_mutex_lock( &count_mtx );
     if ( count == N-1 ) pthread_cond_wait( &full );
     enter_item();
     count++;
     if ( count == 0 ) pthread_cond_signal( &empty );
     pthread_mutex_unlock( &count_mtx );
   }

   void remove() {
     pthread_mutex_lock( &count_mtx );
     if ( count == 0 ) pthread_cond_wait( &empty );
     remove_item();
     count--;
     if (count == N-1) pthread_cond_signal( &full );
     pthread_mutex_unlock( &count_mtx );
   }
};
```

Erzeuger-Verbraucherprogramm

```
ProducerConsumer pc;

void producer() {
    produce_item();
    pc.enter();
}

void consumer() {
    pc.remove();
    consume_item();
}
```

Praxis: Synchronisation in C++ (3)

- **Atomare Befehle** (--> Literatur)**:**
  - **C++0x/C++11-Standard:**
    - **Atomic-**API: `#include <atomic>`
  - **OpenMP**-Standard:
    - #pragma omp atomic newline
      statement_expression
  - **Compiler-Unterstützung** (z.B. g++):
    - __sync_lock_release(&_lock); // testAndSet(_lock, 0);

Praxis: Synchronisation in C++ (4)

- **Mutex-Objekt:**

```
int RefCount::dec_refcount() {
    pthread_mutex_lock( &lock );
    --refcount;
    pthread_mutex_unlock( &lock );

    return  refcount; //not ok!
}
```

Kontextwechsel vor return

--> Vor Rückgabe Veränderung von refcount durch andere Threads möglich

- **Lösung:** Mutex in Konstruktor und Destruktor

```
class MutexObj {
private:
    MutexObj( pthread_mutex_t & lock )
    : _lock(lock) {
    pthread_mutex_lock( &_lock );
}
~MutexObj() {
    pthread_mutex_unlock( &_lock );
}

pthread_mutex_t & _lock;
};
```

```
int RefCount::dec_refcount() {
    MutexObj lock( refcount_lock );
    return  --refcount;    //ok
}
```

Synchronisation von refcount bis nach lokaler Kopie bei return

Aber: nur ok bei **return-per-value** !