5. Synchronisation (2)

5. Synchronisation

5.1 Einführung
5.2 Kritische Abschnitte
5.3 Synchr.-Methoden
5.4 Deadlocks

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

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 (1)

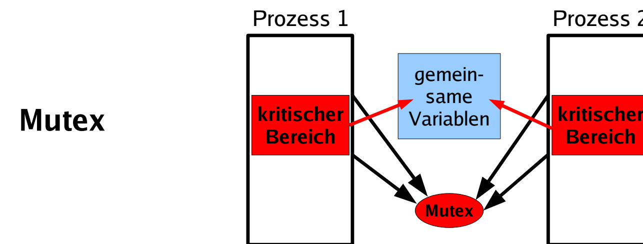
Motivation

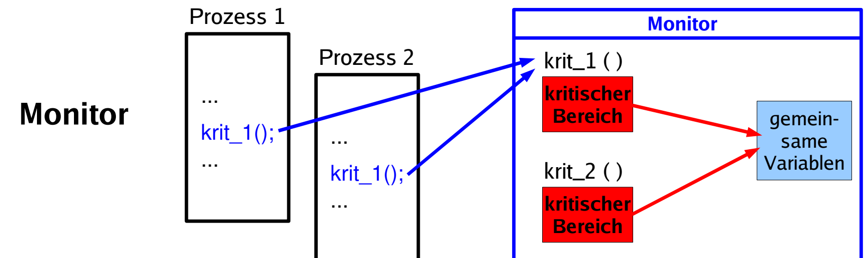
- Arbeit mit Semaphoren und Mutexen zwingt den Programmierer, vor und nach jedem kritischen Bereich wait() und signal() aufzurufen
- Wird dies ein einziges Mal vergessen, funktioniert die Synchronisation nicht mehr
- **Monitor** kapselt die kritischen Bereiche
- Monitor muss von Programmiersprache unterstützt werden (z.B. Java, Concurrent Pascal)

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

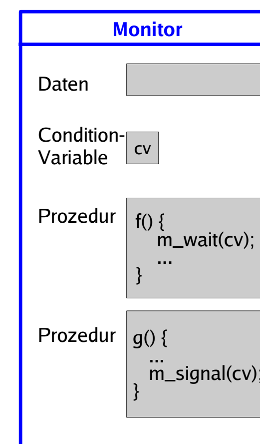
Monitor (7)

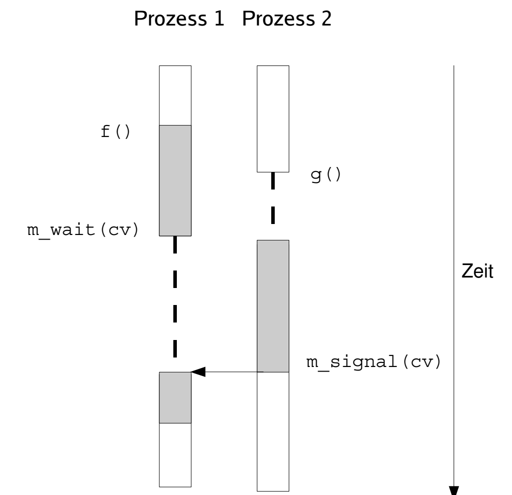
Zustandsvariablen (condition variables)

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 (9)

- Gesperrte Prozesse landen in einer Warte-schlange, 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();
  }
}
```


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/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

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)

Nachrichten (1)

- Nachrichtenaustausch über zwei Systemaufrufe
  - `send    (destination, &message);`
  - `receive (source,      &message);`
- Synchrone Kommunikation:
  Threads blockieren, wenn *send* bzw. *receive* nicht sofort ausgeführt werden können, z. B. weil
  - die Gegenseite keinen entsprechenden Befehl abgesetzt hat,
  - ein Zwischenpuffer für die Nachrichten voll bzw. leer ist.

Nachrichten (2)

- Vorteil: funktioniert auch bei Systemen ohne gemeinsamen Hauptspeicher (distributed systems, client-server-computing)
- Nachteile:
  - aufwändiger durch Duplizieren der Daten
  - Verwaltung der Namen für Quelle und Ziel nötig
  - Vorkehrungen gegen Verlust der Meldung nötig
- Implementierung z. B. durch Pipes oder Mailslots (Windows) oder RPCs.
- Auch Broadcast an mehrere Prozesse möglich

Nachrichten (3)

- **synchron** vs. **asynchron**
  - synchron: *send* / *receive* blockieren, bis zugehörige Operation auf Gegenseite abgeschlossen ist
  - asynchron: *send*-Call kehrt sofort zurück; Erfolg des Versands ist evtl. überprüfbar, z. B.:
    - Gegenseite schickt explizit Antwort
    - Messaging-System sendet Signal bei Zustellung
- **verbindungsorientiert** vs. **verbindungslos**
  - verbindungsorientiert: „stehende Verbindung“ (TCP)
  - verbindungslos (vgl. UDP)

Beispiel für Nachrichten

Zwei Prozesse wechseln sich im Zugriff ab

```
void funktion (int id) {
  int otherid = 1    id;
  char message[10] = "";
  // ein Prozess darf zuerst; ID 0
  if (id==0) {
    message = "go";
  }
  // unkritische Befehle
  while message != "go" {
    receive (otherid, &message);
  }
  // kritischer Bereich
  send (otherid, "go");
  message = "";
  // mehr unkritische Befehle
}
```

p0 ruft *funktion(0)* auf,
p1 ruft *funktion(1)* auf.

p0 darf zuerst in den kritischen Bereich

Linux: Synchronisation im Kernel

Synchronisation im Linux-Kernel

- Atomare Operationen
  - auf Integer-Variablen (atomic_set, atomic_add, atomic_inc, ...)
  - Bit-Operationen auf Bitvektoren (set_bit, clear_bit, test_and_set, ...)
- Spin Locks / Reader-Writer Spin Locks
- Semaphore / Reader-Writer-Semaphore

Atomare Integer-Operationen (1)

- Neuer 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);*

Atomare Integer-Operationen (2)

- *res =* ***atomic_sub_and_test*** *(i, &var);*
  zieht atomar i von var ab.
  - Rückgabewert true, falls Ergebnis 0 ist;
  - Rückgabewert false, falls Ergebnis nicht 0
- *res =* ***atomic_dec_and_test*** *(&var);*
  res = ***atomic_inc_and_test*** *(&var);*
  führt atomar *var--;* bzw. *var++;* aus.
  - Rückgabewert true, falls Ergebnis 0 ist;
  - Rückgabewert false, falls Ergebnis nicht 0

Atomare Integer-Operationen (3)

- *res =* ***atomic_add_negative*** *(i, &var);*
  addiert atomar *i* zu *var*.
  - Rückgabewert true, falls Ergebnis negativ ist;
  - Rückgabewert false, falls Ergebnis ≥ 0 ist

Atomare Bit-Operationen (1)

- Einzelne Bits in Bitvektoren setzen
- Datentyp: beliebig, z. B.
  unsigned long bitvektor = 0;
  - nur über Pointer ansprechen
  - Anzahl der setz-/testbaren Bits hängt von Größe des verwendeten Datentyps ab
- ***set_bit*** *(i, &bitvektor);* *i*-tes Bit setzen
- ***clear_bit*** *(i, &bitvektor);* *i*-tes Bit löschen
- ***change_bit*** *(i, &bitvektor);* *i*-tes Bit kippen

Atomare Bit-Operationen (2)

- Test-and-Set-Operationen geben zusätzlich den vorherigen Wert des jeweiligen Bits zurück
  - *b =* ***test_and_set_bit*** *(i, &bitvektor);*
  - *b =* ***test_and_clear_bit*** *(i, &bitvektor);*
  - *b =* ***test_and_change_bit*** *(i, &bitvektor);*
- Einzelne Bits auslesen
  - *b =* ***test_bit*** *(i, &bitvektor);*
- Suchfunktionen
  - *pos =* ***find_first_bit*** *(&bitvektor, laenge);*
  - *pos =* ***find_first_zero_bit*** *(&bitvektor, laenge);*

Spin Locks (1)

- Lock mit Mutex-Funktion: Gegenseitiger Ausschluss
- Code, der ein Spin Lock anfordert und nicht erhält, läuft in Schleife weiter, bis das Lock verfügbar wird („spinning“)
- Typ: *spinlock_t*

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock (&xy_lock);
/* kritischer Abschnitt */
spin_unlock (&xy_lock);
```


Spin Locks (2)

- Da Spin Locks nicht schlafen, kann man sie in Interrupt-Handlern verwenden
- In dem Fall: zusätzlich Interrupts sperren:

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED
unsigned long flags;

spin_lock_irqsave (&xy_lock, flags);
/* kritischer Abschnitt */
spin_unlock_irqrestore (&xy_lock, flags);
```

(aktuelle Interrupts in *flags* sichern, dann sperren bzw. ursprünglichen Zustand wiederherstellen)

Spin Locks (3)

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

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock_irq (&xy_lock);
/* kritischer Abschnitt */
spin_unlock_irq (&xy_lock);
```

  schaltet alle Interrupts aus bzw. wieder an

- Spin Locks sind nicht „rekursiv“,
  d.h.: es ist nicht möglich, das gleiche Spin Lock zweimal nacheinander anzufordern, etwa beim rekursiven Aufruf einer Funktion

Spin Locks (4)

- Um Blockieren zu vermeiden, ist Lock-Abfrage mit *spin_is_locked (&xy_lock);* möglich
- Locking-Versuch mit *spin_try_lock*:

```
if ( spin_try_lock (&xy_lock) ) {
  /* kritischer Abschnitt */
  spin_unlock (&xy_lock);
} else {
  /* durfte nicht in den kritischen Abschnitt */
}
```

- Beide Funktionen sollte man nicht verwenden: Entweder braucht man das Lock (und muss dann ggf. warten), oder man braucht es nicht...

Reader Writer Locks (1)

- Alternative zu normalen Locks, die mehrere Lesezugriffe zulässt – bei schreibendem Zugriff aber exklusiv (wie ein normales Lock) ist

```
rwlock_t xy_rwlock = RW_LOCK_UNLOCKED;
```

Lesender Code

```
read_lock (&xy_rwlock) ) {
  /* kritischer Abschnitt,
     read-only */
read_unlock (&xy_rwlock);
```

Schreibender Code

```
write_lock (&xy_rwlock) ) {
  /* kritischer Abschnitt,
     read & write */
write_unlock (&xy_rwlock);
```

- Nur bei klarer Trennung zwischen lesenden / schreibenden Programmteilen!

Reader Writer Locks (2)

-

	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 hier Varianten für Interrupt-Behandlung:
  - read_lock_irq read_unlock_irq
  - read_lock_irqsave read_unlock_irqrestore
  - write_lock_irq write_unlock_irq
  - write_lock_irqsave write_unpock_irqrestore

Semaphore (1)

- Kernel-Semaphore sind „schlafende" Locks
- Ist ein Semaphor schon gelockt, werden weitere Interessenten in eine Warteschlange eingereiht.
- Bei Freigabe eines Semaphors wird der erste wartende Thread in der Warteschlange geweckt
- Semaphore eignen sich für Sperren, die über einen längeren Zeitraum gehalten werden
  - keine Verschwendung von Rechenzeit

Semaphore (2)

- Semaphore sind nur im Prozess-Kontext einsetzbar, nicht in Interrupt-Handlern (Interrupt-Handler werden nicht vom Scheduler behandelt)
- Code, der einen Semaphor verwenden will, darf nicht bereits ein normales Lock besitzen (Semaphor-Zugriff kann dazu führen, dass der Thread sich schlafen legt.)
- Semaphore können auch mehr als einen Thread auf die Ressource zugreifen lassen

Semaphore (3)

Typ: *semaphore*

Statische Deklaration

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

Dynamische Semaphor-Erzeugung

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

- Verwendung mit *up()* und *down()*

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


Semaphore (4)

- Varianten von *down()*
  - *down (&sem);*
    nicht unterbrechbarer Schlaf, falls Semaphor nicht verfügbar
  - *down_interruptible (&sem);*
    unterbrechbarer Schlaf, falls Sem. nicht verfügbar
  - *down_trylock (&sem);*
    versucht, den Semaphor zu erhalten – falls das nicht gelingt, kehrt die Funktion sofort mit False-Wert zurück

Semaphore (5)

- Beispiel für *down_trylock()*

```
/* Auszug aus /usr/src/linux/kernel/printk.c */

if (!down_trylock(&console_sem)) {
      console_locked = 1;
      /*
       * We own the drivers.  We can drop the spinlock and let
       * release_console_sem() print the text
       */
      spin_unlock_irqrestore(&logbuf_lock, flags);
      console_may_schedule = 0;
      release_console_sem();
      /* Funktion release_console_sem() führt up(&console_sem); aus */
} else {
      /*
       * Someone else owns the drivers.  We drop the spinlock, which
       * allows the semaphore holder to proceed and to call the
       * console drivers with the output which we just produced.
       */
      spin_unlock_irqrestore(&logbuf_lock, flags);
}
```


Reader-Writer-Semaphore (1)

- Analog zu Reader Writer Locks:
  Typ *rw_semaphore*, der spezielle Up- und Down-Operationen für Lese- und Schreibzugriff erlaubt
- Alle Reader-Writer-Semaphore sind Mutexe (Zähler ist bei Initialisierung immer 1)

Statische Deklaration

```
static DECLARE_RWSEM (name);
```

Dynamische Semaphor-Erzeugung

```
init_rwsem (&sem);
```


Reader-Writer-Semaphore (2)

```
static DECLARE_RWSEM (xy_rwsem);
```

Lesender Code

```
down_read (&xy_rwsem) ) {
  /* kritischer Abschnitt,
     read-only */
up_read (&xy_rwsem);
```

Schreibender Code

```
down_write (&xy_rwsem) ) {
  /* kritischer Abschnitt,
     lesen und schreiben */
up_write (&xy_rwsem);
```

Genau wie bei Reader Writer Locks:

	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