Vorlesung Betriebssysteme II

Thema 2: Synchronisation

Robert Baumgartl

17. 03. 2014

Wiederholung: Semaphore (nicht gähnen!)

- Abstrakter Datentyp (Zähler und Warteschlange)
- Operationen:
  - **P()**: „probieren und eventuell warten“,
  - **V()**: „verlassen und eventuell wecken“,
  - **Init()**
  - **Try()**: nichtblockierendes Auslesen
- Synonyme: `down()`/`up()`, `wait()`/`signal()`
- binär/nichtbinär
- selbst kritischer Abschnitt, Schutz z.B. mit *Spinlocks*

Semaphore, cont'd

Probleme beim Einsatz von Semaphoren:

- Komplexität der Programmierung, z.B. Nachweis der Deadlockfreiheit schwierig
- keine festgelegte Strategie beim Aktivieren innerhalb der **V()**-Operation
- Threadsicherheit

Behandelte Fragestellungen (BS I):

- Steuerung kritischer Abschnitte
- Problem der speisenden Philosophen (zumindest angerissen)

Implementierung (Pseudocode) der P()-Operation

```
void P()
{
  static int sem;

  if (sem > 0) {
    sem--;
    return;
  }
  else {
    /* Prozess -> wartend setzen */
}
```

Problem: Unterbrechung zwischen `if` und `sem--` *und* weiterer Prozess ruft ebenfalls P() über *gleichem* Semaphor → 2 Prozesse im kritischen Abschnitt → *Race Condition*.

Implementierung (Pseudocode) der P()-Operation

Rekursion: Datenstruktur zum Schutz kritischer Abschnitte ist selbst kritischer Abschnitt!

- kann nicht mit sich selbst geschützt werden
- Schutz (z. B.) durch sog. Spinlocks

verbesserte Implementierung der P-Operation (Pseudocode):

```
void P()
{
    static int sem;
    spinlock sl;
    lock(sl);
    if (sem > 0) {
      sem--;
      unlock(sl);
      return;
    }
    else {
       unlock(sl);
       /* Prozess -> wartend setzen */
    }
```

Spinlocks („Schlossvariablen“)

- Abstrakter Datentyp (ADT), Grundidee:
  - vor Eintritt in k. A. : bestimmten Wert in Datenstruktur schreiben (Lock schließen, `enter_cs()`)
  - bei geschlossenem Lock: kontinuierlicher Test, ob Lock wieder geöffnet wurde (*busy waiting, spinning*)
  - gleichzeitige Schreiboperation durch Hardware serialisieren (d. h. , einer gewinnt – zufällig)
  - nach k. A.: Öffnung des Locks nicht vergessen (`leave_cs()`)

Spinlock-Implementierung, Versuch 1

```
            .globl enter_cs, leave_cs
            .equ locked,0
            .equ unlocked,1

            .data
lock:       .long unlocked

            .text
enter_cs:
            movl (lock),%eax
            cmpl $locked,%eax
            je enter_cs                              # try again
            movl $locked,(lock)
; *** k.A. ***
leave_cs:
            movl $unlocked,(lock)
```

(i386, AT&T-Assembler-Syntax)

Analyse der Implementierung

- Unterbrechung (z. B.) zwischen `movl` und `cmpl` kann dazu führen, dass mehr als ein Prozess k.A. betritt
- analog Diskussion Implementierung Semaphor
- Ursache: Lesen der Variable `lock`, nachfolgender Vergleich und Schreiben nicht atomar
- Abhilfe: kurzes Sperren der Interrupts (suboptimal – warum?)
- Alternative: Implementierung mittels Maschineninstruktionen, die *gleichzeitig* ein Datum lesen und schreiben können

Analyse der Implementierung

- Unterbrechung (z. B.) zwischen `movl` und `cmpl` kann dazu führen, dass mehr als ein Prozess k.A. betritt
- analog Diskussion Implementierung Semaphor
- Ursache: Lesen der Variable `lock`, nachfolgender Vergleich und Schreiben nicht atomar
- Abhilfe: kurzes Sperren der Interrupts (suboptimal – warum?)
- Alternative: Implementierung mittels Maschineninstruktionen, die *gleichzeitig* ein Datum lesen und schreiben können

Analyse der Implementierung

- Unterbrechung (z. B.) zwischen `movl` und `cmpl` kann dazu führen, dass mehr als ein Prozess k.A. betritt
- analog Diskussion Implementierung Semaphor
- Ursache: Lesen der Variable `lock`, nachfolgender Vergleich und Schreiben nicht atomar
- Abhilfe: kurzes Sperren der Interrupts (suboptimal – warum?)
- Alternative: Implementierung mittels Maschineninstruktionen, die *gleichzeitig* ein Datum lesen und schreiben können

Analyse der Implementierung

- Unterbrechung (z. B.) zwischen `movl` und `cmpl` kann dazu führen, dass mehr als ein Prozess k.A. betritt
- analog Diskussion Implementierung Semaphor
- Ursache: Lesen der Variable `lock`, nachfolgender Vergleich und Schreiben nicht atomar
- Abhilfe: kurzes Sperren der Interrupts (suboptimal – warum?)
- Alternative: Implementierung mittels Maschineninstruktionen, die *gleichzeitig* ein Datum lesen und schreiben können

Analyse der Implementierung

- Unterbrechung (z. B.) zwischen `movl` und `cmpl` kann dazu führen, dass mehr als ein Prozess k.A. betritt
- analog Diskussion Implementierung Semaphor
- Ursache: Lesen der Variable `lock`, nachfolgender Vergleich und Schreiben nicht atomar
- Abhilfe: kurzes Sperren der Interrupts (suboptimal – warum?)
- Alternative: Implementierung mittels Maschineninstruktionen, die *gleichzeitig* ein Datum lesen und schreiben können

Atomare Lese-Schreib-Instruktionen der IA32[1]

Instruktion	*Semantik*
`btr`	Bit Test and Reset (Memory)
`bts`	Bit Test and Set (Memory)
`xchg`	Exchange (Memory $\leftrightarrow$ Register)
`cmpxchg`	Compare and Swap (Memory $\leftrightarrow$ Register)

Instruktionen sind nicht gleichwertig, man kann mit ihnen noch viele interessante Dinge treiben (z. B. nichtblockierende Zugriffe realisieren $\rightarrow$ später).

[1] Intel-32-Bit-Architektur

Spinlock-Implementierung, Versuch 2

```
            .globl enter_cs, leave_cs
            .equ locked,0
            .equ unlocked,1

            .data
lock:       .long unlocked

            .text
enter_cs:
            movl $locked,%eax
            xchgl %eax,(lock)
            cmpl $locked,%eax
            je enter_cs:                        # try again
; *** k.A. ***
leave_cs:
            movl $unlocked,(lock)
```

Analyse von Versuch 2

- `xchg` sperrt den Bus ($\rightarrow$ keine gleichzeitige Ausführung bei mehreren Prozessoren/Cores)
- Inkonsistenzen unmöglich $\rightarrow$ korrekte Implementierung eines Spinlocks
- kann zentral (durch das BS) oder dezentral implementiert werden
- normalerweise nicht für Nutzerprozesse angeboten (aktives Warten!)
- eingesetzt für *kurze kritische Abschnitte* im BS
- Auf SMP-Maschinen nicht effizient, da Spinlock enthaltende Cacheline auf allen Prozessoren durch Schreiboperation invalidiert werden (was bedeutet das?)

Auflösung: Warum nicht einfach Interrupts verbieten?

- effizient (1 Instruktion)
- funktioniert nicht auf SMP-Maschinen, da auf allen anderen Prozessoren Interrupts noch erlaubt (Sperren geschieht lokal)
- Erhöhung der Interrupt-Latenz
- $\rightarrow$ Gefahr des Verpassens von Interrupts (können nicht gespeichert werden)
- $\rightarrow$ nur unter bestimmten Voraussetzungen sinnvoll

Mehrere API für Semaphore unter Unix

1. Teil der sog. „System V"-IPC:
   - vergleichsweise alte API
   - relativ komplizierte Semantik
   - nicht threadsicher
   - `man 7 svipc`
2. POSIX-Semaphore:
   - `man 7 sem_overview`
   - relativ neu
   - threadsicher
   - `sem_wait()`, `sem_post()`, `sem_open()`, `sem_close()`
3. Mutexe der Pthreads-Bibliothek
   - Konzept des Mutex (**Mut**ual **Ex**clusion device)
   - u. a. `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`

Weitere Semaphor-API

Win32:

- CreateSemaphore(), DeleteSemaphore(), WaitForSingleObject(), WaitForMultipleObjects(), ReleaseSemaphore()

Linux-Kern:

- Semaphore → Mutex
- kompliziert (vgl. `kernel/mutex.c`)
- nur für Kernel-Mode (z. B. Gerätetreiber)
- `mutex_lock()`, `mutex_unlock()`, ...

Semaphor-API nach System V

Überblick

- Es werden Mengen (*Sets*) von Semaphoren erzeugt ($\rightarrow$ Anzahl Semaphore pro Set angeben!)
- Semaphore besitzen 3x3 Zugriffsbits (wie Dateien)
- Kommandos `ipcs`, `ipcrm`
- Semaphore sind **nicht** automatisch initialisiert

Systemruf	*Semantik*
`semget`	Erzeugen eines Semaphor-Sets
`semop`	Zustandsmanipulation (P(), V())
`semctl`	Management (z. B. Init, Statusabfrage, Löschen)

Beispiel: erzver-sysv.c (extern)

Identifikation der Semaphore

Identifikation der Semaphore auf zwei Arten möglich:

- benannte Semaphore:
  - (systemweit einheitlichen) Key bei `semget` übergeben (Integer-Zahl)
  - jeder Prozess, der auf Semaphore zugreifen will, muss `semget` mit diesem Key aufrufen
  - Rechteprüfung durch das System
  - Key z. B. über gemeinsames Headerfile verteilbar
- unbenannte Semaphore:
  - `IPC_PRIVATE` bei `semget` als „Key“ übergeben
  - resultierenden Identifikator über `fork()` vererbbar ($\rightarrow$ nur zwischen verwandten Prozessen)

Semaphorausgleichswert

Problem: Was passiert, wenn ein Prozess einen Semaphor sperrt und dann (fehlerhaft) abbricht?

- Semaphor nicht mehr zu öffnen?
- andere Prozesse u. U. in Mitleidenschaft gezogen?!

Lösung: *Semaphor Adjustment Value* = Semaphorausgleichswert

- bei `semop` das Flag `SEM_UNDO` angeben
- *jede* Zustandsänderung eines Semaphors (P()/V()) wird für *jeden* Prozess kumulativ in einer extra Variable gespeichert
- Beispiel: Führt der Prozess dreimal P() ($= -3$) und zweimal V() ($= +2$) aus, so enthält die Variable `semadj` den Wert $+1$ ($-(-3+2) = +1$), da *Umkehrung* der getätigten Operationen.
- Bei Prozessbeendigung (regulär oder vorzeitig) wird der Wert der Variable auf den betroffenen Semaphor addiert und
- $\rightarrow$ die durch den beendeten Prozess durchgeführten Semaphoroperationen somit rückgängig gemacht.

Weitere Aspekte der System-V-Semaphoren

- nichtblockierende P()-Operation mittels Flag `IPC_NOWAIT`
  - verhindert Deadlock
  - (bzw. ersetzt ihn durch Livelock)
- Funktion `ftok()` zur Generierung eines systemweit eindeutigen Keys

Nachteile der API:

- Anlegen und Initialisierung der Semaphore nicht atomar
- stets Mengen von Semaphoren bearbeitet, auch wenn nur eine benötigt ($\rightarrow$ 2 verschiedene Identifikatoren/Indizes benötigt)
- verschiedene Operationen durch einen Systemruf abgebildet (`semop()`, `semctl()`)
- Typdefinition `union semun` im Usercode nötig
- explizite Löschung erforderlich

Motivation: Verdeutlichung der Komplexität der Programmierung mit (mehreren) Semaphoren

Gegeben: n Leseprozesse, m Schreibprozesse, gemeinsame Datenbasis

Randbedingungen:

- gleichzeitiges Lesen zulässig (und erwünscht),
- Schreibzugriffe exklusiv, d.h., es darf immer nur ein Prozess schreibend auf die Datenbasis zugreifen,
- kein Lesen während Schreiboperation (um keine inkonsistenten Daten zu lesen).

Gesucht: korrekte Synchronisation

```
program readerswriters1;
var
   mutex : semaphore;
process type reader;
begin
   wait(mutex);
   (* lesender Zugriff *)
   signal(mutex);
end    (* reader *)

process type writer;
begin
   wait(mutex);
   (* schreibender Zugriff *)
   signal(mutex);
end    (* writer *)

begin
  initial(mutex, 1);
   ...
```

Diskussion:

- fair
- kein paralleles Lesen möglich
- Sprache ist Pascal-FC

Listing: src/readwrite2.pfc (extern)

Diskussion:

- „Standardlösung“ (in den meisten Lehrbüchern diskutiert
- z. B. im Tanenbaum; im Stallings, S. 249, . . .)
- unfair: später eintreffende Leser blockieren u.U. wartende Schreiber („drängeln sich vor“)

Listing: src/readwrite2.pfc (extern)

Diskussion:

- „Standardlösung“ (in den meisten Lehrbüchern diskutiert
- z. B. im Tanenbaum; im Stallings, S. 249, ...)
- unfair: später eintreffende Leser blockieren u.U. wartende Schreiber („drängeln sich vor“)

Listing: src/readwrite3.pfc (extern)

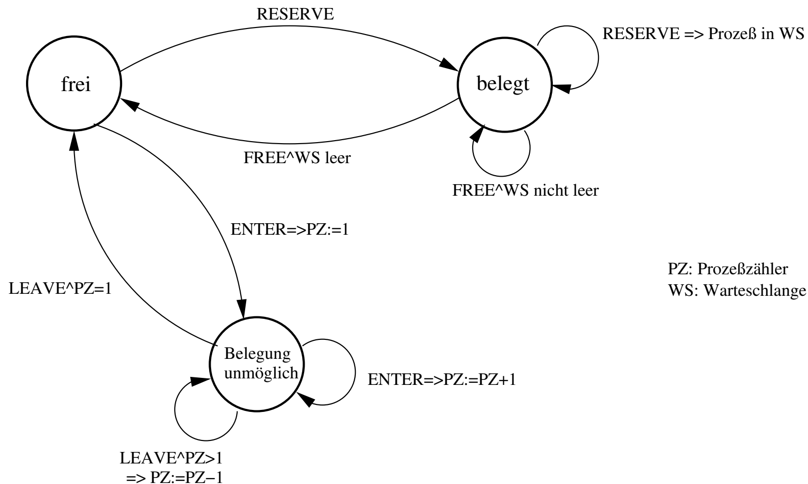
Diskussion:

- 5 Semaphore, 2 Variable
- unfair: neue Schreiber blockieren Leser
- kaum noch zu überblicken (!)

Diese sowie eine weitere Implementierung führen wir uns im Praktikum zu Gemüte.

- 3 Zustände: *belegt*, *frei* und *Belegung nicht möglich*
- 4 Operationen über BV möglich:
  - `RESERVE` → Variable geht in Zustand *belegt* oder Prozeß wird suspendiert (Semantik von *P()*)
  - `FREE` → wartender Prozeß wird fortgesetzt oder Variable geht in Zustand *frei* (Semantik von *V()*)
  - `ENTER` → wenn Variable *belegt*, dann wird Prozeß suspendiert; wenn Variable *frei* oder *Belegung nicht möglich*, dann Übergang in *Belegung nicht möglich* und Prozeßzähler++
  - `LEAVE` → wenn Prozeßzähler > 1 dann Verbleib in *Belegung nicht möglich*, ansonsten Übergang in *frei*
- zählt Anzahl der ENTER- bzw. LEAVE-Aufrufe

Zustandsdiagramm einer Boltvariable



Leser-Schreiber-Problem in PEARL

```
PROBLEM;
  DCL Boltvar BOLT;

Writer: TASK;
  REPEAT
    RESERVE Boltvar;
      ! Schreibender Zugriff
    FREE Boltvar;
  END; ! Loop
END; ! Consumer

Reader: TASK;
  REPEAT
    ENTER Boltvar;
      ! Lesender Zugriff
    LEAVE Boltvar;
  END; ! Loop
END; ! Producer 1
...
```

- Ziel: Erhöhung des Parallelitätsgrades im Kern

1. Leser-Schreiber-Semaphore
   - `struct rw_semaphore`
   - Definition in `/usr/src/linux/include/linux/rwsem.h`
   - Funktionen: `init_rwsem()` `down_read()`, `up_read()`, `down_write()`, `up_write()`
   - identische Semantik zu Bolt-Variablen
   - faires Verhalten: FIFO; entweder ein Schreiber oder alle Leser werden aufgeweckt, wenn ein Prozeß den Semaphor verläßt (abhängig von Position in Warteschlange)

2. Leser-Schreiber-Spinlocks
    - Typ `rwlock_t` (int)
    - Definition in `/usr/src/linux/include/linux/spinlock.h` vgl. auch `rwsem_spinlock.h`
    - Funktionen (Makros): `read_lock()`, `read_unlock()`, `write_lock()`, `write_unlock()`
    - initial stets offen

Implementierung des Leser-Schreiber-Spinlocks

- `rwlp` sei vom Typ `rwlock_t` (Integer, 32 Bit)
- `rwlp`$=0 \rightarrow$ Lock offen
- `rwlp`$= n, n > 0 \rightarrow n$ Leser anwesend
- MSB von `rwlp` gesetzt $\rightarrow$ Schreiber anwesend

`read_lock(rwlp)` expandiert zu (i386, AT&T-Syntax):

```
1: lock; incl rwlp
   jns 3f
   lock; decl rwlp
2: cmpl $0, rwlp
   js 2b
   jmp 1b
3:
```

Implementierung des Leser-Schreiber-Spinlocks

write_lock

`write_lock(rwlp)` expandiert zu:

```
1: lock; btsl $31, rwlp
   jc 2f
   testl $0x7fffffff,rwlp
   je 3f
   lock; btrl $31, rwlp
2: cmp $0, rwlp
   jne 2b
   jmp 1b
3:
```

Zwischenfazit

Zwei Lehren:

1. Das Leser-Schreiber-Problem ist keine abgehobene, unbrauchbare Betriebssysteme-Theorie sondern modelliert den lesenden und schreibenden Zugriff vieler Prozesse auf exklusive Ressourcen (alltägliche Situation in parallelen Systemen).
2. Problemstellung hat praktische Inkarnation gefunden in Form von
   2.1 Datentypen in Programmiersprachen (Bolt-Variable in PEARL)
   2.2 Datenstrukturen in Betriebssystemen (Leser-Schreiber-Semaphore bzw. -Spinlocks)

Motivation: Reduktion der Komplexität der Programmierung im Vergleich zum Einsatz von Semaphoren

- Monitor = Menge an (Monitor-)Prozeduren, gemeinsamen Variablen und Datenstrukturen, „Bedingungsvariablen“ ($\rightsquigarrow$ ADT)
- Grundregel: Exakt 1 Prozess kann im Monitor aktiv sein
- Konstrukt der Programmiersprache $\Rightarrow$ Compiler für wechselseitigen Ausschluß verantwortlich
- u.a. Concurrent Pascal, Pascal-FC, Modula II, Java
- Synchronisationsmechanismus „höherer Ebene“
- Implementation z.B. mittels Semaphore

Zum „Umschalten“ zwischen Prozessen im Monitor gibt es Bedingungsvariablen:

- repräsentieren *jeweils* eine Prozesswarteschlange
- zwei (grundlegende) Operationen: `delay()` und `resume()` (bzw. manchmal auch `wait()` und `signal()` genannt)

Bedingungsvariablen

delay() und resume()

delay(cond_var) $\rightarrow$ Prozess wird blockiert (in Warteschlange `cond_var` aufgenommen

resume(cond_var) $\rightarrow$

a) Prozeßwarteschlange (von `cond_var`) leer $\Rightarrow$ keine Reaktion, Rückkehr

b) Prozeßwarteschlange enthält genau 1 Prozeß $\Rightarrow$ dieser Prozeß deblockiert, setzt nach `delay()` fort

c) Prozeßwarteschlange enthält mehrere Prozesse $\Rightarrow$ ein Prozeß zur Deblockierung vom Scheduler ausgewählt, dieser setzt nach letztem `delay()` fort

Widerspruch: Ein Prozess A aktiviert durch `resume()` einen Prozeß B im Monitor. $\longleftrightarrow$ Es darf stets nur ein Prozeß im Monitor aktiv sein.

Lösung: Es existieren verschiedene Semantiken, die man kennen und unterscheiden muss:

A) “resume-and-continue”

- A wird weiter ausgeführt
- B setzt erst fort, wenn A Monitor verlassen hat
- z.B. Programmiersprache Mesa, Java

B) “resume-and-exit”

- A verläßt zwangsweise den Monitor (z.B., indem `resume()` nur als letzte Anweisung einer Monitorprozedur erlaubt ist)
- B kann sofort Arbeit in Monitor aufnehmen
- z.B. in Concurrent Pascal (Brinch Hansen)

C) “resume-and-urgent-wait”

- A betritt eine besondere Warteschlange, die *Chivalry Queue*
- diese hat höhere Priorität als alle anderen Warteschlangen
- B kann sofort Arbeit in Monitor aufnehmen
- A wird fortgesetzt, sobald ein Prozeß den Monitor verläßt (und kein weiterer Prozeß in *Chivalry Queue* existiert)
- z.B. in Pascal-FC (Hoare, 1973)

Bedingungsvariablen in Pthreads

Bedingungsvariablen existieren auch alleinstehend, z.B. in der Bibliothek POSIX-Threads (aka *pthreads*):

- Datentyp `pthread_cond_t`

Funktion	*Semantik*
`pthread_cond_init()`	Kreieren und Init einer BV.
`pthread_cond_signal()`	Wecken eines Threads
`pthread_cond_broadcast()`	Wecken *aller* Threads
`pthread_cond_wait()`	Threadblockierung an BV.
`pthread_cond_timedwait()`	Blockierung mit Timeout
`pthread_cond_destroy()`	Löschung der BV.

Rückbesinnung: Monitore

Mögliche Zustandstransitionen von Prozessen innerhalb von Monitoren:

- Ein Prozeß, der eine Prozedur eines Monitors betritt, in dem bereits ein anderer aktiv ist, blockiert an der Eintrittswarteschlange (*boundary queue*)
- Ein Prozeß, der (innerhalb eines Monitors) `delay(cond_var)` aufruft, wird an der Warteschlange von `cond_var` blockiert.
- Wenn ein Prozeß `resume(cond_var)` aufruft, und die Warteschlange von `cond_var` ist nicht leer, so wird *genau ein* dort wartender Prozeß aktiv gesetzt[2]
- Verläßt ein Prozeß eine Monitorprozedur, und es gibt blockierte Prozesse in der Eintritts- oder Vorrangwarteschlange, so wird genau einer daraus aktiviert. Die Vorrangwarteschlange ist hierbei priorisiert.

[2] Beachten Sie aber die konkrete Semantik von resume() wie oben erläutert!

Struktur eines Monitors

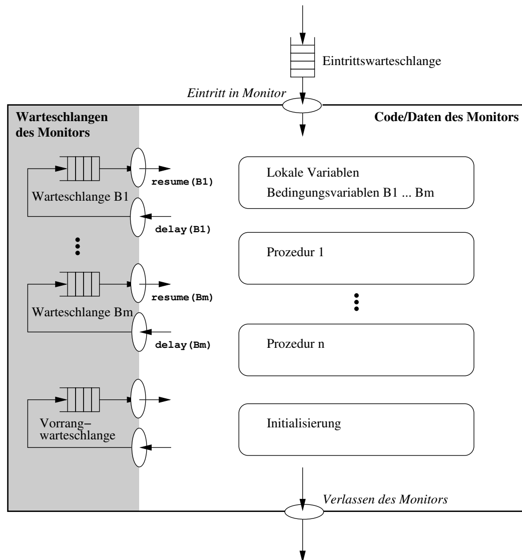


Abbildung: (nach William Stallings, Operating Systems)

Anwendung von Monitoren

1. Mutex mit Monitoren:
   Listing (Fragment): src/monitormutex.pfc (extern)
2. Monitor für das Erzeuger-Verbraucher-Problem (Hoare, 1973)
   Listing: src/pcmonitor.pfc (extern)
   - 2 verschiedene Warte„anlässe“ $\rightsquigarrow$ 2 Bedingungsvariablen
   - bewachter Puffer ist Teil des Monitors
   - Zugriff auf Monitor-eigene Variablen inhärent exklusiv
   - Nachteil: Typgebundenheit

Anwendung von Monitoren, contd.

3. Lösung des Leser-Schreiber-Problems mit Hilfe von Monitoren
   Listing: src/rwmonitor.pfc (extern)
   - zusätzliche Funktion `empty(cond_var)`; liefert `true`, wenn kein Prozeß bezüglich `cond_var` wartet, ansonsten `false`
   - Leser bevorzugt:
     - kontinuierlicher Strom von Lesern führt zum Verhungern wartender Schreiber
     - nach jedem Schreibvorgang Test, ob Leser warten (wenn ja, Aktivierung eines [und damit aller] wartenden Lesers)
     - $\leadsto$ kein Verhungern von Lesern möglich

Fazit: Bewertung des Monitor-Konzepts

Vorteile:

- erzwingt strikte Trennung von Synchronisation und eigentlichem Programmcode
- Komplexität der Synchronisation von Anwendungsprogramm in Compiler verlagert
- Es resultiert weitaus übersichtlicherer Programmcode.

Nachteile:

- Verschachtelung von Monitorprozeduren problematisch
- Prozessabbrüche?!
- Kein gleichzeitiger Zugriff auf Ressource (z. B. paralleles Lesen) mehr möglich. Ist dies gewünscht, so kann die Ressource nicht im Monitor eingebettet werden. Dies führt wiederum zur Aufweichung der Kapselung (Zugriff auf Ressource ohne Monitorprozedur möglich).