Betriebssysteme
6. Synchronisation von Prozessen und Threads

Tobias Lauer

Nebenläufigkeit

- Prozess/Thread[1]-Konzept in Betriebssystemen ermöglicht den parallelen Ablauf von Programmen:
  - Einprozessorsystem: virtuelle Parallelität
  - Mehrprozessorsystem: reale Parallelität
- Dies ist ein Vorteil, beinhaltet aber auch eine inhärente „Unsicherheit“
- Zentraler Begriff: „Nebenläufigkeit“ (engl. concurrency)
  - Programme schreiten unabhängig voneinander in der Zeit voran
  - keine Aussage über den jeweiligen Grad des Programmfortschritts über die Zeit hinweg möglich
  - Ausführung kann beliebig oft, an beliebigen Stellen, beliebig lange verzögert werden (Beispiele: E/A-Unterbrechungen, Round-Robin-Verdrängungen, Speicher-Nachladevorgänge, etc.)

⇨ Es ist bei nebenläufigen Prozessen/Threads keine a-priori Vorhersage möglich, welche Code-Teile in welcher relativen Reihenfolge zueinander ausgeführt werden: „Indeterminismus“ der Abarbeitung

[1]: in den folgenden Folien: der Einfachheit halber nur „Prozesse“

Ist Nebenläufigkeit ein Problem?

- Nebenläufigkeit ist KEIN Problem, wenn die Prozesse/Threads völlig unabhängig voneinander sind

- Nebenläufigkeit IST ein Problem,
  - wenn die Prozesse auf gemeinsame Betriebsmittel zugreifen
  - wenn die Prozesse gemeinsame Daten verwalten
  - wenn die Prozesse miteinander kooperieren müssen

- Klassische Probleme bei Nebenläufigkeit
  - Prozesse streiten sich um Betriebsmittel (z.B. Hardware)
    Gefahr von „Unfairness“, „Verhungern“ (*Starvation*)
  - Prozesse können gemeinsame Daten (z.B Files, Datenstrukturen) beschädigen („*Race Conditions*“, inkonsistente Datenzustände)
  - Prozesse können sich gegenseitig blockieren („*Deadlocks*“)

Beispiel für Inkonsistenz durch Nebenläufigkeit

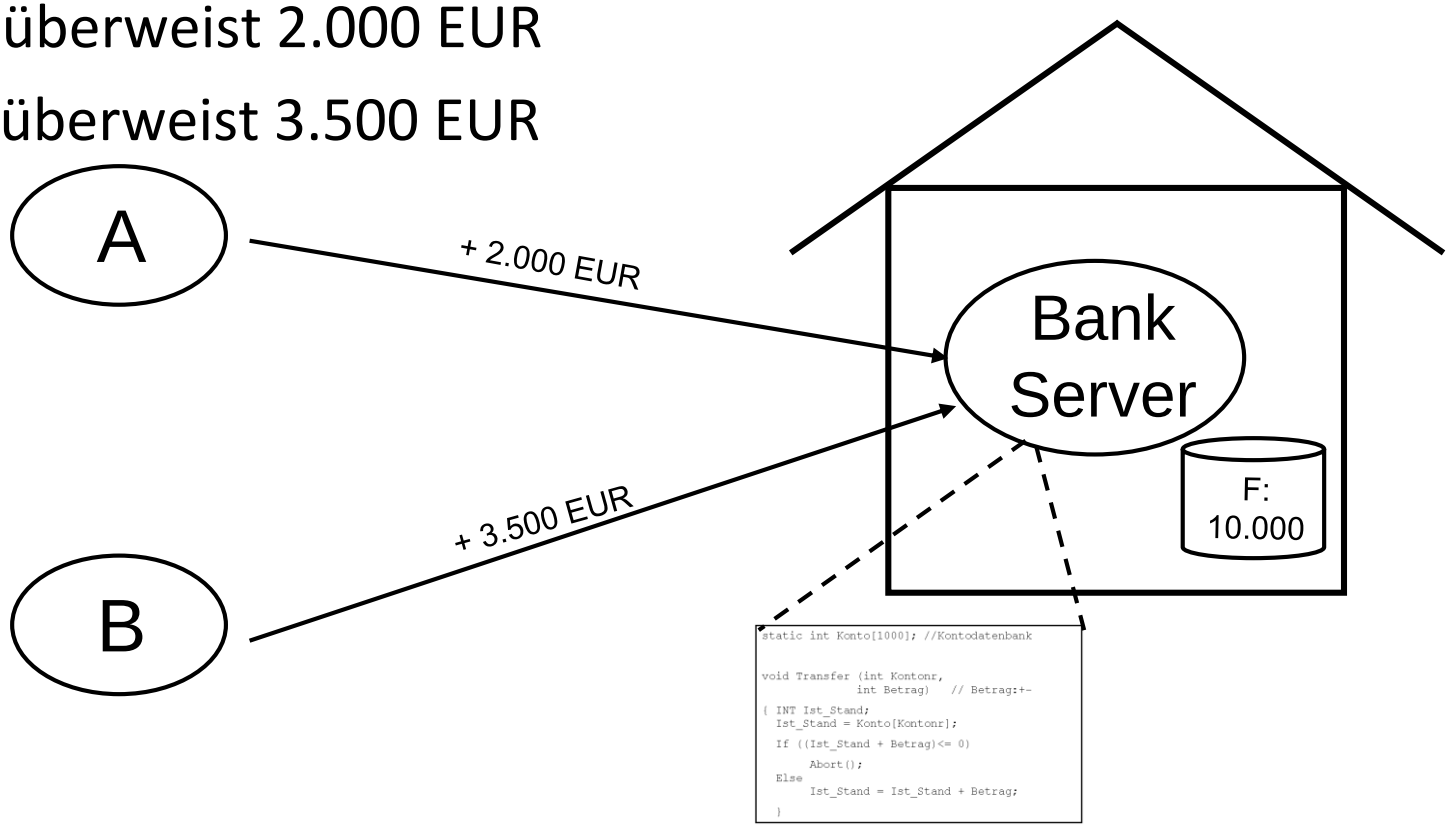
- Homebanking mit Bank-Server
- Transferoperation wird als Prozedur bereitgestellt:

```
int Konto[1000]; // Kontodatenbank


void Transfer (int Kontonr,
               int Betrag) // Überweisung (+/-)
{ int Ist_Stand, Soll_Stand;
  Ist_Stand = Konto[Kontonr];
  if ((Ist_Stand + Betrag)< 0)
       Abort();
  else
       Soll_Stand = Ist_Stand + Betrag;
  Konto[Kontonr] = Soll_Stand;
}
```

- Für jeden Homebanking-Auftrag wird auf dem Server ein separater Thread erzeugt

Beispiel

- Kunde A und B bezahlen gleichzeitig Rechnungen an eine Firma F mit Kontonummer 999
- F hatte zuvor einen Kontostand von 10.000 EUR
- Kunde A überweist 2.000 EUR
- Kunde B überweist 3.500 EUR



Beispiel

```
Kunde A: Transfer (999, 2000);
```

```
Kunde B: Transfer (999, 3500);
```

Thread 1:

```
void Transfer (int Kontonr,
               int Betrag)

{ int Ist_Stand, Soll_Stand;
  Ist_Stand = Konto[Kontonr];

  if ((Ist_Stand + Betrag)< 0)
       Abort();
  else
       Soll_Stand
        = Ist_Stand + Betrag;
   Konto[Kontonr] = Soll_Stand;
```

Thread 2:

```
void Transfer (int Kontonr,
               int Betrag)

{ int Ist_Stand, Soll_Stand;
  Ist_Stand = Konto[Kontonr];
  if ((Ist_Stand + Betrag)< 0)
       Abort();
  else
       Soll_Stand
        = Ist_Stand + Betrag;
  Konto[Kontonr] = Soll_Stand;
```

Fehler!

Was fehlt?

- Kritische Ausführungsteile der Threads müssen sich gegenseitig ausschließen
- „Kritischer Abschnitt“ („Critical Section/Region“) eines Programms:
  - Abfolge von Befehlen, die nicht durch andere Programme unterbrochen werden dürfen
  - Wechselseitiger Ausschluss („Mutual Exclusion“)

Mutex

```
void Transfer (int Kontonr,
               int Betrag)

{ int Ist_Stand, Soll_Stand;
  Ist_Stand = Konto[Kontonr];

  if ((Ist_Stand + Betrag)< 0)

       Abort();

  else
       Soll_Stand = Ist_Stand + Betrag;


  Konto[Kontonr] = Soll_Stand;
```

Realisierung kritischer Abschnitte

- Einfachste Möglichkeit: Während kritischer Abschnitte Interrupts sperren

DI

EI

```
void Transfer (int Kontonr,
               int Betrag)

{ INT Ist_Stand, Soll_Stand;
  Ist_Stand = Konto[Kontonr];
  If ((Ist_Stand + Betrag)< 0)
       Abort();
  Else
       Soll_Stand = Ist_Stand + Betrag;
  }
  F[Kontonr] = Soll_Stand;
```

- Keine Unterbrechungen (weder durch E/A, noch durch Round-Robin-Scheduler, etc.)
- Kritische Abschnitte sind automatisch geschützt
- Nachteile
  - Blockade von (evtl. zeitkritischen) E/A-Operationen
  - Reduzierte Effizienz (DI stört Scheduling-Strategie)
  - Im Mehrprozessorfall: Interrupt-Sperrung betrifft nur denjenigen Prozessor, auf dem das Programm gerade läuft → nicht einsetzbar

Versuch einer einfachen Software-Lösung

- Ansatz: Programm setzt Soft-Interrupt-Sperre in Form einer Variablen
- Speichern / Lesen einer einfachen Variable sind nicht unterbrechbar
- globale Variable „flag“: flag=0 : Unterbrechung erlaubt
  flag=1 : Unterbrechung verboten
- Warte aktiv, bis Soft-Interrupt frei ist.

Begin_Region();

```
void Transfer (int Kontonr,
               int Betrag)
{ INT Ist_Stand, Soll_Stand;
  Ist_Stand = Konto[Kontonr];
  If ((Ist_Stand + Betrag)< 0)
       Abort();
  Else
       Soll_Stand = Ist_Stand + Betrag;
  }
  F[Kontonr] = Soll_Stand;
```

End_Region();

```
static int flag=0; // initial frei

void Begin_Region () {
 while (flag == 1) { }; //wait loop
 flag = 1;
}

void End_Region () {
 flag = 0;
}
```

Der Peterson-Algorithmus für 2 Prozesse P_0, P_1

```
static int flag[2];          // Anzeige des Begin_Region
static int turn;             // wer ist priorisiert im
                                Fall von Kollisionen


void Begin_Region (int p) {
   int other = 1-p;                // Nr. des anderen Prozesses
   flag[p] = 1;                    // Anzeige, dass Region beantragt
   turn = other;                   // Gib anderem den Vortritt
   while ((turn == other) && (flag[other] == 1)) {
         // Wait in loop, ODER: SwitchToThread (yield)
   }
}

void End_Region (int p) {
   flag[p] = 0;
}
```

Beweis für Peterson-Algorithmus

s. Tafel

Noch eine Möglichkeit: Hardware-Unterstützung

- Atomarer (nicht unterbrechbarer) „Test-And-Set“-Befehl

Pseudocode (in C):

ATOMAR

```
int test_and_set (int *i)  {    // Testet Variable i
                                // Returnwerte: 0 (success), 1 (failed)

  if (*i==1)
    return (1);                 // Fall I: noch blockiert

  else
    { *i = 1; return (0); }     // Fall II: war frei; neu gesetzt

}
```

- Verwendung:

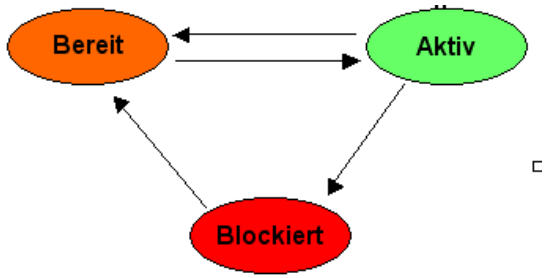
Dies funktioniert!

```
static int flag=0; // initial frei


void Begin_Region (int *flag) {

  while ( test_and_set(flag) == 1)
    { };    // wait in loop
}
```

```
void End_Region (int *flag) {

  *flag = 0;

}
```

Diskussion der bisherigen Lösungen

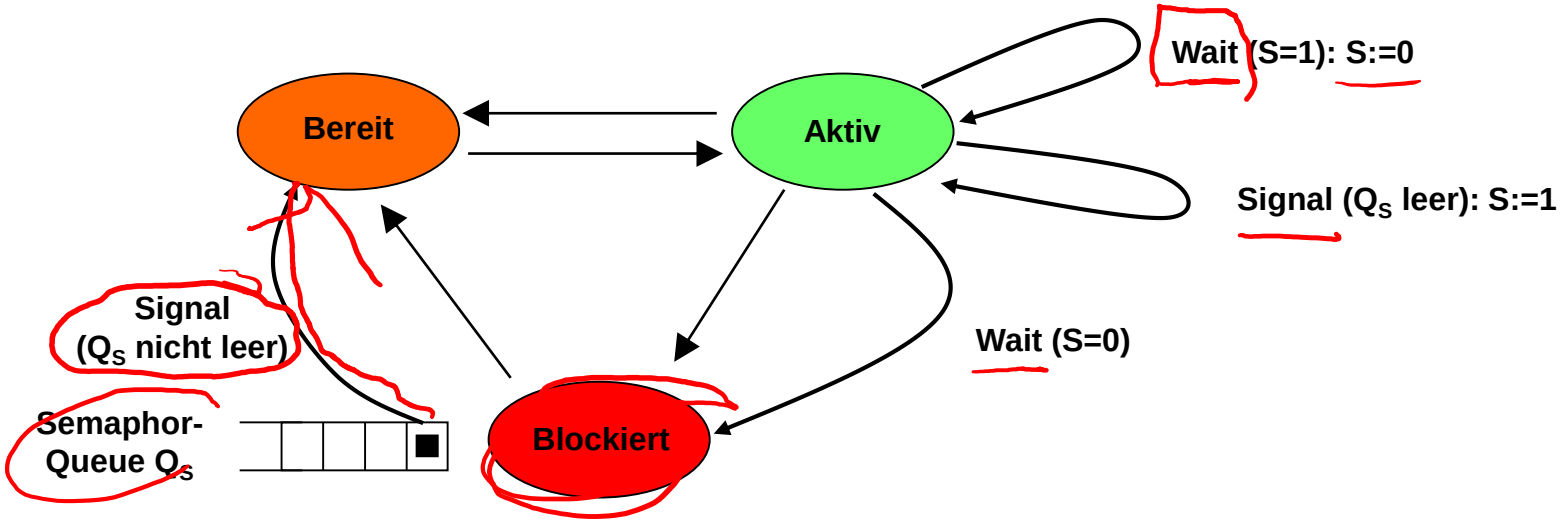
- Vorteil: Es gibt Verfahren, die in der Lage sind, kritische Abschnitte zu garantieren
- Nachteil: „Busy Waiting“ vor Eintritt in den kritischen Abschnitt verschwendet CPU-Ressourcen
- Nachteil: Gefahr der Prioritätsinvertierung
  (Busy Wait verhindert Abschluss des kritischen Abschnitts)
- Nachteil: Gefahr des Verhungerns (wer bekommt den Abschnitt?)



- Grundsätzlich neuer Ansatz: Einbeziehung des Schedulers zur Synchronisation von Prozessen
- Prinzip:
  - Prozesse, die auf kritische Abschnitte warten, werden blockiert.
  - Das Beenden von kritischen Abschnitten bringt wartende Prozesse nach „bereit“
  - Der Scheduler bringt diese bereiten Prozesse nach „aktiv“

Semaphore

- Synchronisationskonzept zwischen Prozessen (Dijkstra, 1965)
- Semaphor = Synchronisationsvariable + Operationen
- Binärer Semaphor S: Werte 0 und 1; Initialwert 1
- Operationen (interne Implementierung):
  - Wait(S): Falls S=1 → Setze S=0; aufrufender Prozess darf weitermachen
    Falls S=0 → Prozess blockieren (auf Semaphor-Warteliste setzen)
  - Signal(S): Fall 1: Es gibt blockierte Prozesse → 1 Prozess deblockieren (welchen?)
    Fall 2: Keine blockierten Prozesse → Setze S=1

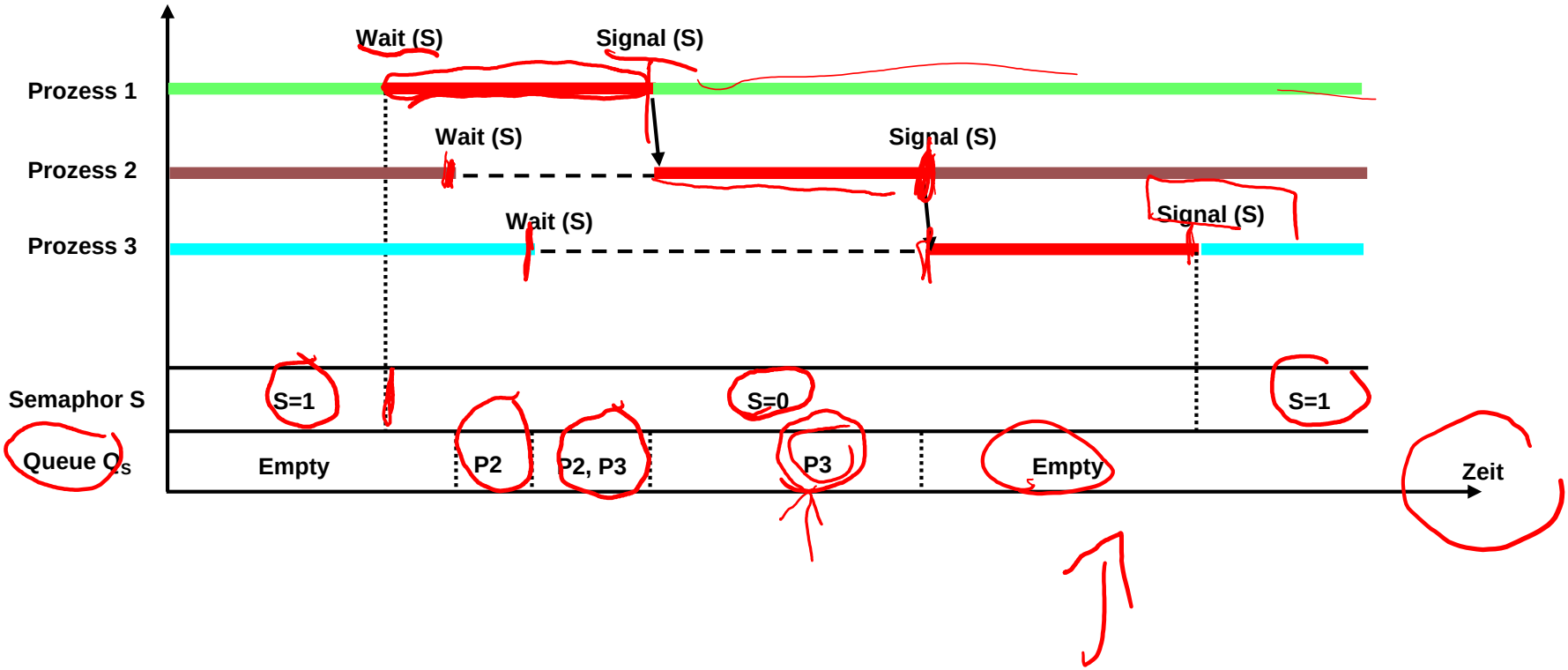


Semaphore zur Realisierung kritischer Abschnitte

- Statt Begin_Region (..): Wait (my_semaphore)
- Statt End_Region (..): Signal (my_semaphore)

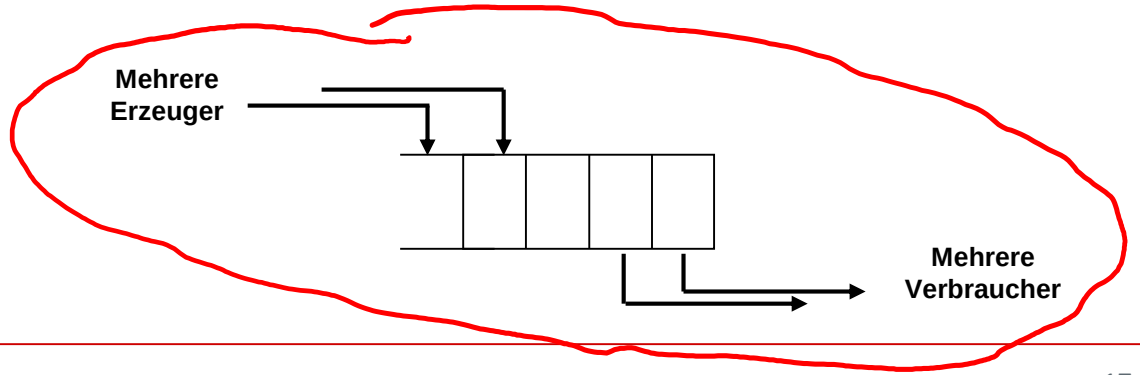
```
static BIN_SEMAPHORE S = 1;

void Transfer (int Kontonr,
               int Betrag)

{ INT Ist_Stand, Soll_Stand;
  Wait (S);
  Ist_Stand = Konto[Kontonr];
  If ((Ist_Stand + Betrag)< 0)
       Abort();
  Else
       Soll_Stand = Ist_Stand + Betrag;
  }
  F[Kontonr] = Soll_Stand;
  Signal (S);
```

Beispiel


Wait (S)
Signal (S)
Prozess 1
Wait (S)
Signal (S)
Prozess 2
Wait (S)
Signal (S)
Prozess 3
Semaphor S
S=1
S=0
S=1
Queue Qs
Empty
P2
P2, P3
P3
Empty
Zeit

Allgemeine (nicht-binäre) Semaphore

- Semaphor-Variable *S* nimmt Werte an von -*M* bis + *N*
- Operationen:
  - Wait (S): Setze S := S-1;
    Fall 1 (S ≥ 0): <Prozess darf weitermachen>
    Fall 2 (S < 0): <Prozess Blockieren>
  - Signal (S): Setze S:= S+1;
    Fall 1 (S ≤ 0): <Einen Prozess deblockieren>
    Fall 2 (S > 0): < Do nothing >
- Geeignet für weitergehende Synchronisationsaufgaben zwischen Prozessen
- Beispiel: Erzeuger-Verbraucher-Synchronisation über gemeinsamem Puffer



Anwendungen von Semaphoren

- Gegenseiter Ausschluss (mutual exclusion)
  Krit. Abschnitt sichern
- Synchronisation von Abläufen (barrier)
  - Jeder Thread/Prozess einer Gruppe muss warten, bis alle anderen an der Barriere angekommen sind, bevor alle weitermachen dürfen
- Produzenten & Konsumenten
  s. Tafel & Code
  - Puffer zur Speicherung zwischen Erzeugen und Konsumieren
- Leser & Schreiber
  - Lesen und Schreiben gemeinsamer Variablen

Erzeuger-Verbraucher-Schema (Producer-Consumer)

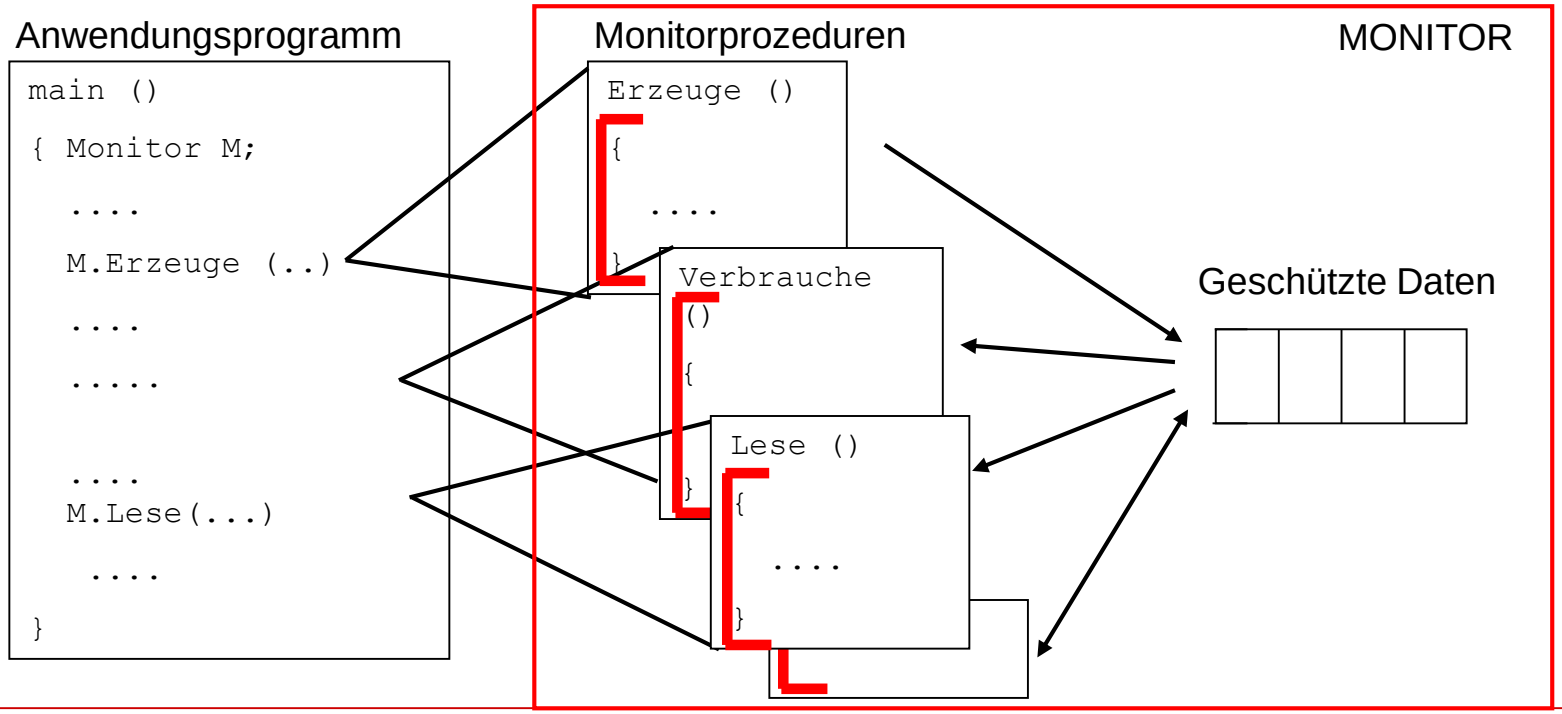
Situation: Menge von *m* Erzeugern, die „Dinge“ produzieren
Menge von *n* Verbrauchern, die „Dinge“ verwenden
gemeinsamer Puffer zum Ablegen von „Dingen“

Nachteile von Semaphoren

- Möglichkeiten für den Anwendungsprogrammierer, Fehler zu machen
  - Zu viele oder zu wenige „Signal“ Operationen
    (z.B.: Im IF-Teil vorhanden, aber im ELSE-Teil vergessen)
  - Zu viele oder zu wenige „Wait“ Operationen
  - Verklemmung durch falsche Verschachtelung
  - Race condition durch fälschliches Freigeben (bei Verwendung als Mutex)
- Erfordert starke Programmierdisziplin und Übersicht, vor allem bei Änderungen im Code
- Probleme treten i.d.R. erst zur Laufzeit auf
  - „Programm hängt“ → unklar, was passiert ist
  - Probleme sind i.d.R. nicht-deterministisch (race conditions) und nicht einfach reproduzierbar
- DESWEGEN: Semaphore – wenn möglich – nur an „übersichtlichen Stellen“ einsetzen (nicht datenflussabhängig)
- ALTERNATIVE: Monitore

Synchronisationskonzept: MONITORE

- „Objektorientierter“ Ansatz für Synchronisation
- Monitor = Softwaremodul mit 1..N Prozeduren/Methoden
  - Nur die Prozeduren/Methoden können auf die lokalen Daten zugreifen
  - Nur jeweils ein Prozess kann zu einer Zeit im Monitor sein
- D.h. Monitorprozeduren haben „automatisch eingebaute“ kritische Abschnitte



Monitore

- Vorteil gegenüber Semaphoren:
  - Benutzer kann weniger Fehler machen (1 Eintritt und 1 Austrittspunkt aus kritischem Abschnitt)
  - Monitor kann gleichzeitig Information Hiding für Daten (Objekte) realisieren
  - Debugging/Fehlersuche ist erleichtert

Kapselung

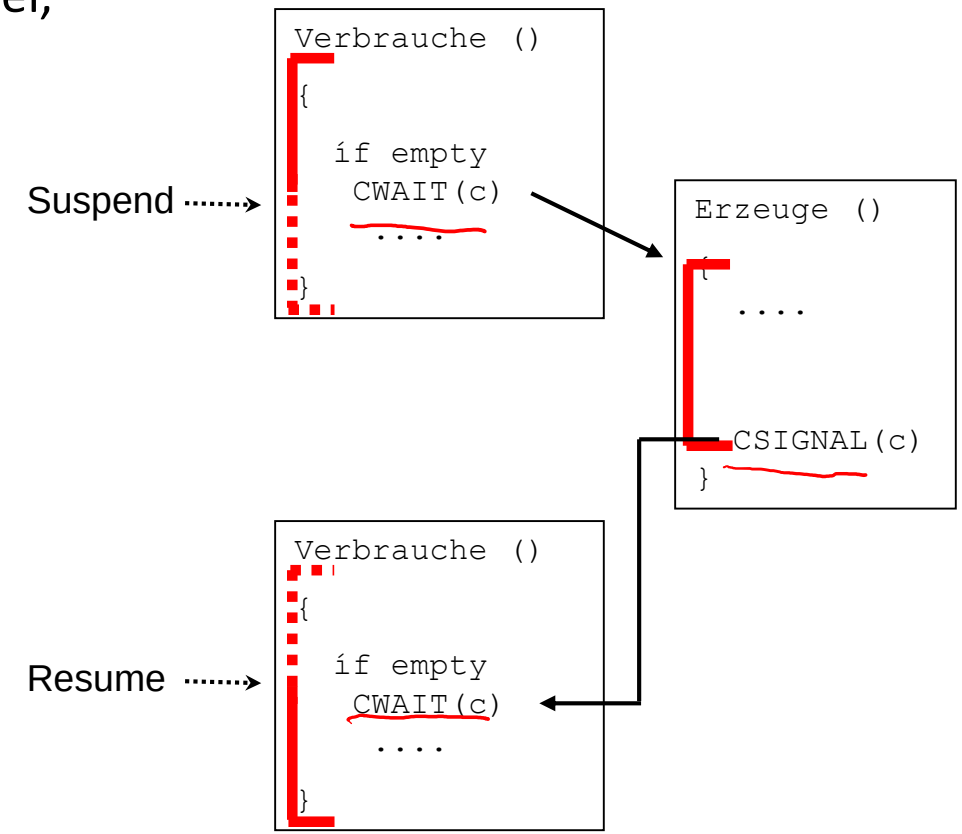
- Was fehlt noch?
  - Synchronisationsmechanismen zur Kooperation zwischen Prozessen
  - Beispiel: Verbraucher muss sich im Monitor blockieren, bis Erzeuger neue Daten geliefert hat

- Condition_Wait (CWAIT), Condition_Signal (CSIGNAL)

- Achtung: CWAIT muss Monitor zwischenzeitig freigeben, sonst Verklemmung (Deadlock)

Monitore – Zusammenspiel CSIGNAL-CWAIT

- Monitorprozeduren können sich über Bedingungsvariablen (Conditions) synchronisieren
- Monitorprozedur gibt Monitor frei, wenn sie blockiert wird
- Monitore häufig in Programmiersprachen eingebettet
- Beispiele:
  - ADA, Modula
  - Java (keyword „synchronized“)
  - C# (keyword "lock")



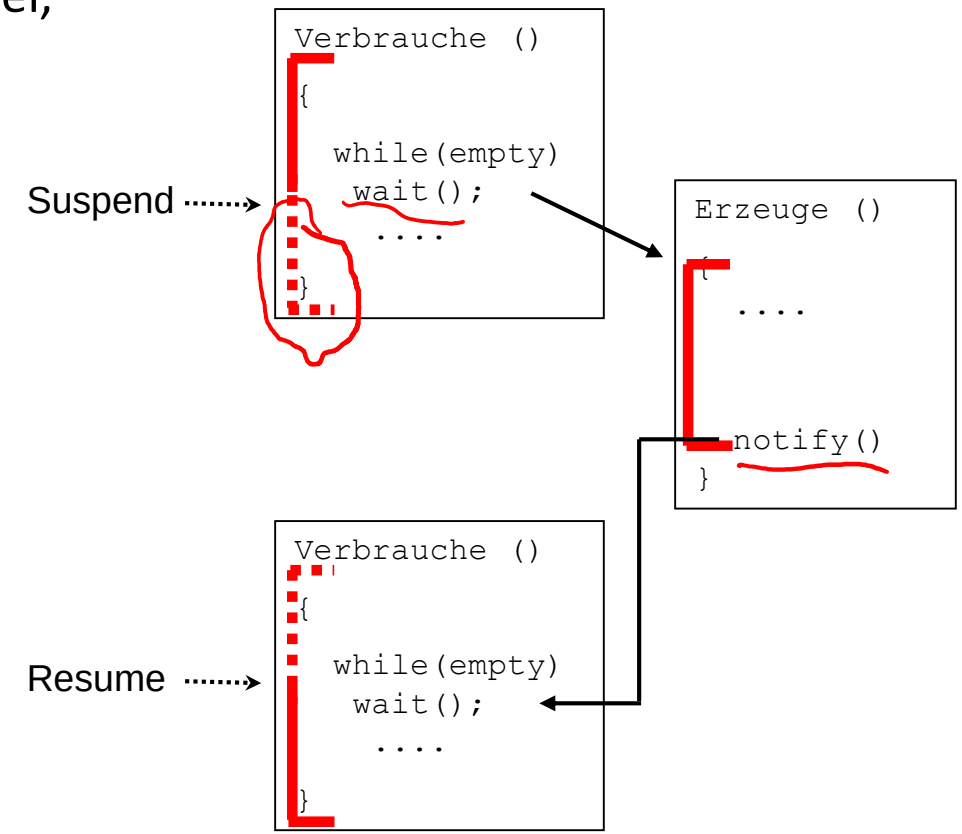
Monitore – Zusammenspiel Java (wait – notify)

- Monitorprozeduren können sich über Bedingungen (Conditions) synchronisierien

- Monitorprozedur gibt Monitor frei, wenn sie blockiert wird

- Monitore häufig in Programmiersprachen eingebettet

- Java:
  Schlüsselwort „synchronized“
  - Methoden wait() und notify()



Beispiele für Synchronisationsdienste in UNIX/LINUX

- GNU/LINUX Thread-**Mutex**
  - aus der POSIX-Bibliothek
  - Mutex = kritischer Abschnitt
  - erster Thread, der Mutex sperrt, kann kritischen Abschnitt ausführen
  - alle weiteren Threads werden bei Sperroperation blockiert (in Mehrprozessorumgebungen evtl. auch Spin-Lock)
  - Nach Beendigung des kritischen Abschnitts wird einer der warten-den Threads aufgeweckt und erhält als nächster den Eintritt
  - Mutex-Variable vom Typ pthread_mutex_t
  - Dienste: „pthread_mutex_lock(), pthread_mutex_unlock(),..“
  - Auch nichtblockierende Sperren: „pthread_mutex_trylock()“

```
#include <pthread.h>

pthread_mutex_t my_mutex
        = PTHREAD_MUTEX_INITIALIZER;


void* thread_function (void* arg)

{ while (1)

  {   ....
      pthread_mutex_lock(&my_mutex);

      ...
      // kritischer Abschnitt

      ...
      pthread_mutex_unlock(&my_mutex);
      ...
  }
```

Beispiele für Synchronisationsdienste in UNIX/LINUX

- **Prozess-Semaphore**
  - semget(), semctl(): allokieren, initialisieren und deallokieren ein Semaphor
  - semop(): für Signal/Wait-Operationen auf dem Semaphor
  - semop() kann Semaphorvariable
    a) um N erhöhen (~ Signal)
    b) um N reduzieren (~ Wait)
  - Würde bei „b)“ ein Wert < 0 entstehen, blockiert der Prozess
  - Er wacht wieder auf, wenn Wert der Semaphorvariable ≥ N ist (also wenn „b)“ ohne Blockade ausgeführt werden kann)
  - Wait-Operation kann auch nichtblockierend (Test-Wait) verwendet werden

Beispiele für Synchronisationsdienste in Windows

- **Critical Sections** ~ Posix Mutexe
- Nur für Threadsynchronisation (nicht für Prozesssynchronisation) effizienter als Semaphore
- Operationen: InitializeCriticalSection(), EnterCriticalSection(), LeaveCriticalSection(), TryEnterCriticalSection()
- SetCriticalSectionSpinCount() erlaubt in Mehrprozessorumgebungen zu definieren, wie lang der Thread „busy waiting“ praktiziert, bevor er blockiert wird.

Beispiel:

```
// Global variable
CRITICAL_SECTION CriticalSection;

void main()
{
    ...

    // Initialize the critical section one time only.
    if (!InitializeCriticalSectionAndSpinCount(&CriticalSection,
        0x80000400) )
        return;
    ...

    // Release resources used by the critical section object.
    DeleteCriticalSection(&CriticalSection)
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
    ...

    // Request ownership of the critical section.
    EnterCriticalSection(&CriticalSection);

    // Access the shared resource.

    // Release ownership of the critical section.
    LeaveCriticalSection(&CriticalSection);

    ...
}
```

Beispiele für Synchronisationsdienste in Windows

- **Mutexe** (WaitOne() .... ReleaseMutex() für kritische Abschnitte; auch über Prozessgrenzen hinweg einsetzbar: "named mutex")
- **Semaphore** (WaitOne(), Release() für Wait/Signal-Operationen ähnlich wie UNIX/LINUX)
- **Asynchronous Procedure Calls** (APCs, „QueueUserAPC()“)
  - erlauben asynchrone Ausführung z.B. von FileRead, FileWrite Operationen
  - FileRead/FileWrite kommt sofort zurück zum aufrufenden Programm
  - Nach Abschluss der Operation wird eine benutzerdefinierte Prozedur aufgerufen
- **Timer Queues** („CreateTimerQueue()“)
  - spezielle Unterstützung von User Timeout Handling
  - Programm spezifiziert Prozedur, die nach Ablauf aufgerufen wird