LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prinzipien, Modelle und Algorithmen der Nebenläufigen Programmierung
Wintersemester 2019/20

Programmierprimitiven Teil II

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 27. November 2019

Übersicht

Vor- und Nachteile von Semaphore

Vorteile

- Viele Probleme der nebenläufigen Programmieren lassen sich mit Semaphore lösen
- In vielen Programmiersprachen sind Semaphore implementiert

Vor- und Nachteile von Semaphore (2)

Nachteile

- Explizites Setzen und Entfernen der Locks
- Fehleranfällig:
  - Vergisst der Programmierer einen `signal`-Aufruf kann ein Deadlock auftreten
  - Vergisst der Programmieren einen `wait`-Aufruf kann der exklusive Zugriff verletzt werden

Monitore

- Monitore versuchen die Schwächen der Semaphore zu beheben
- Ermöglichen strukturierte Programmierung
- Daten und Zugriff auf die Daten werden in einem Monitor gekapselt
- passen zu dem Modell der Objektorientierung

Monitore (2)

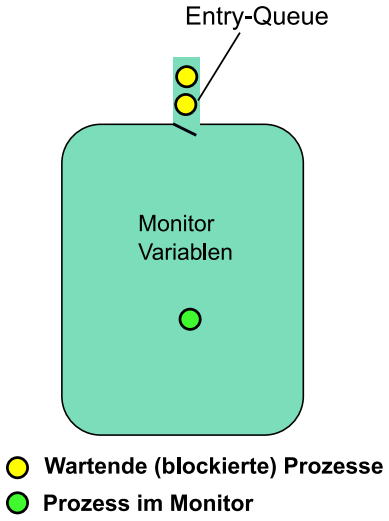
```
monitor Konto {
  int Saldo;
  int Kontonummer;
  int KundenId

  abheben(int x) {
    Saldo := Saldo - x;
  }

  zubuchen(int x) {
    Saldo := Saldo + x;
  }
}
```

- Kapselung von Daten und Methoden
- Kein direkter Zugriff auf Attribute
- Zugriff nur über die Methoden
- Nur ein Prozess kann zu einer Zeit im Monitor sein
- D.h. nur eine Methode von einem Prozess zu einer Zeit am Ausführen
- Andere Prozesse werden blockiert

Monitore (3)



- Wartende Prozesse entweder in FIFO-Schlange oder
- Menge, d.h. keine Starvation-Freiheit
- Hängt von der Implementierung ab

Mutual-Exclusion mit Monitor

```
monitor Mutex {
    ...
    Variablen des kritischen Abschnitts
    ...
    inKritischenAbschnitt(...) {
        Kritischer Abschnitt;
    }
}
```

Prozess i:

```
loop forever
(1)   Restlicher Code
(2)   Mutex.inKritischenAbschnitt()
end loop
```

Vor- und Nachteile bisher

Vorteil gegenüber Semaphore

Setzen / Entfernen des Locks geschieht implizit

Nachteil gegenüber Semaphore

Viele Probleme benötigen explizites Blockieren von Prozessen

Beispiele:

- Bounded Buffer: Erzeuger muss blockiert werden, wenn Buffer voll ist und Verbraucher muss blockiert werden, wenn Buffer leer ist
- Readers- und Writer: Es sollen mehrere Leser-Prozesse erlaubt sein, und Writer-Prozesse müssen blockiert werden solange gelesen wird usw.

Bei Readers- und Writers: Lesende Prozesse dürfen beim Lesen nicht im Monitor verbleiben, sonst kann immer nur ein Prozess lesen!

Erweiterung der Monitordefinition

- Bisherige Monitordefinition zu schwach (im Vergleich mit Semaphore)
- Deshalb: Erweiterung um Condition Variables oder Condition Expressions
- Zwei verschiedene Ansätze, beide kommen in Programmiersprachen vor
- Im folgenden: erst Condition Variables dann (kurz) Condition Expressions

Condition Variables

- FIFO-Queue (meistens) mit Operationen
- Name der Condition Variables wird meistens so gewählt, dass er die wahr werdene Bedingung erläutert, aber
- Die Bedingung wird nicht vom Laufzeitsystem geprüft. Es ist nur ein Name
- Operationen für Condition Variable cond:
  - `waitC`(cond)
  - `signalC`(cond)
  - manchmal: `empty`(cond)
- Achtung: Semantik verschieden von `signal` und `wait` bei Semaphore!

Condition Variables (2)

Simulation eines Semaphors mit einem Monitor

```
monitor Semaphore {
    int s := k;
    condition notZero;

    wait() {
        if s = 0 then
            waitC(notZero)
        s := s - 1
    }

    signal() {
        s := s + 1
        signalC(notZero)
    }
}
```

Semantik von Condition Variables

- Sei P der aufrufende Prozess, cond eine Condition Variable im Monitor monitor.
- Erinnerung: cond ist eine FIFO-Queue (Liste)
- Sei lock der implizite Lock des Monitors (Lock zur Entry-Queue)

Semantik von waitC

```
waitC(cond) {
    cond := append(cond,P); //Prozess wird zur Queue hinzugefügt
    P.state := blocked; //Prozess blockiert
    monitor.lock := release; // Monitor-Lock wird freigegeben
}
```


Semantik von Condition Variables (2)

Semantik von signalC

```
signalC(cond) {
    if cond ≠ empty then
        Q := head(cond); //Erster Prozess der Queue
        cond := tail(cond); //wird aus Queue entfernt
        Q.state := ready; //und entblockiert
}
```

Achtung:

- Die Semantik so ist noch unterspezifiziert:
- Anscheinlich: Aufrufender Prozess und entblockierter Prozess gleichzeitig im Monitor
- Klärung später, Annahme erstmal: Aufrufender Prozess verlässt Monitor

Semantik von Condition Variables (3)

Semantik von empty

```
empty(cond) {
    return(cond = empty); }
```


Semaphoresimulation – nochmal

```
monitor Semaphore {
    int s := k;
    condition notZero;

    wait() {
        if s = 0 then
            waitC(notZero)
        s := s - 1
    }

    signal() {
        s := s + 1
        signalC(notZero)
    }
}
```

Vergleich wait, signal, waitC, signalC

Semaphore Sem	Monitore (Condition Variable cond)
`wait`(Sem) kann zum Blockieren führen, muss aber nicht	`waitC`(cond) blockiert den Prozess stets
`signal`(Sem) hat stets einen Effekt: Entblockieren eines Prozesses oder Erhöhen von Sem.V	`signalC`(cond) kann effektlos sein: Entweder Prozess in cond wird entblockiert, oder effektlos, wenn cond leer ist

Erzeuger / Verbraucher mit Monitor

```
monitor BoundedBuffer {
   bufferType buffer := empty;
   condition notEmpty;
   condition notFull;

   produce(v) {
      if length(buffer) = N then
         waitC(notFull);
      buffer := append(v,buffer);
      signalC(notEmpty);
   }

   consume() {
      if length(buffer) = 0 then
         waitC(notEmpty)
      w := head(buffer);
      buffer := tail(buffer);
      signalC(notFull)
      return(w);    }
}
```

Code des Erzeugers:

```
loop forever
(1)  erzeuge e
(2)  BoundedBuffer.produce(e)
end loop
```

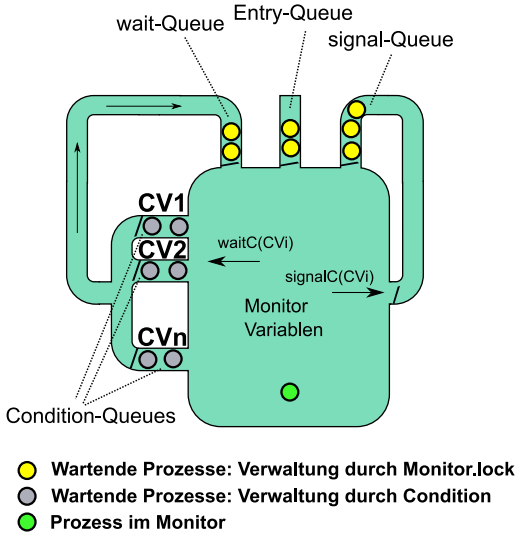
Code des Verbrauchers:

```
loop forever
(1)  e := BoundedBuffer.consume();
(2)  verbrauche e;
end loop
```


Genauere Modellierung von Monitoren

- Erinnerung: Beim Durchführen von `signalC` ist nicht eindeutig, wer den Monitor-Lock anschließend haben soll.
- Es gibt verschiedene Varianten
- Wenn gewecketer Prozess weiterrechnen darf: Wo wartet der signalisierende Prozess?
- Deshalb genauere Modellierung:
- Condition Variable besteht nicht aus einer Queue, sondern aus drei Queues :
  - Condition-Queue
  - wait-Queue
  - signal-Queue

Monitor-Modellierung mit drei Queues für die Condition Variable



Semantik der Queues

Condition-Queues: Dort warten die mit `waitC` blockierten Prozesse.

Wait-Queue: Wird ein `signalC` ausgeführt, so wird der erste wartende Prozess der Condition-Queue in die Wait-Queue eingefügt.

Signal-Queue: Der Prozess der `signalC` ausführt, wird in die Signal-Queue eingereiht, falls er einen Prozess der Condition-Queue entblockiert hat.

Entry-Queue: Wartende Prozesse, die den Monitor betreten möchten

Problem

Welche der drei Queues am Monitor-Lock hat Vorrang vor den anderen?

Prioritäten für die Queues

Problem

Welche der drei Queues am Monitor-Lock hat Vorrang vor den anderen?

- Je nach Implementierung gibt es verschiedene Strategien
- Wir schreiben E, W, S für die Prioritäten der Entry-, Wait- und Signal-Queue

Prioritäten für die Queues (2)

13 Möglichkeiten

1	$E = W = S$	
2	$E = W < S$	Wait and Notify
3	$E = S < W$	Signal and Wait
4	$E < W = S$	
5	$E < W < S$	Signal and Continue
6	$E < S < W$	Klassische Definition
7	$E > W = S$	nicht sinnvoll
8	$E = S > W$	nicht sinnvoll
9	$S > E > W$	nicht sinnvoll
10	$E = W > S$	nicht sinnvoll
11	$W > E > S$	nicht sinnvoll
12	$E > S > W$	nicht sinnvoll
13	$E > W > S$	nicht sinnvoll

Monitore mit Condition Expressions

- Anstelle von Condition Variablen:
- Es dürfen echte Boolesche Ausdrücke verwendet werden bei `waitCE`
  (wir schreiben `waitCE` statt `waitC`)
- Die Operation `signalC` gibt es nicht
- Das Laufzeitsystem prüft, ob die Ausdrücke wahr werden und entblockiert dann automatisch.

Erzeuger / Verbraucher mit Condition Expressions

```
monitor BoundedBuffer {
   bufferType buffer := empty;
   constant length := N;

   produce(v) {
      waitCE(length(buffer) < N); // Puffer voll?
      buffer := append(v,buffer);
   }

   consume() {
      waitCE(length(buffer) > 0); // Puffer leer?
      w := head(buffer);
      buffer := tail(buffer);
      return(w);     }
}
```

Code des Erzeugers:

```
loop forever
(1)   erzeuge e
(2)   BoundedBuffer.produce(e)
end loop
```

Code des Verbrauchers:

```
loop forever
(1)   e := BoundedBuffer.consume();
(2)   verbrauche e;
end loop
```


Im Folgenden:

- Einige der bekannten Anwendungsbeispiele kodiert mit Monitoren
- Wir benutzen Monitore mit Condition Variables
- Mit der klassischen Definition $E < S < W$ (Signal and Urgent Wait)

Readers & Writers mit Monitoren

```
monitor RW {
   int countR, countW := 0;
   condition okToWrite, okToRead;

   startRead() {
      if countW ≠ 0 or not(empty(okToWrite))
      then waitC(okToRead);
      countR := countR + 1;
      signalC(okToRead);}

   endRead() {
      countR := countR - 1;
      if countR = 0 then signalC(okToWrite);}

...}
```


Readers & Writers mit Monitoren (2)

```
...
   startWrite() {
      if countR ≠ 0 or countW ≠ 0
      then waitC(okToWrite);
      countW := 1;}

   endWrite() {
      countW := 0;
      if empty(okToRead)
      then signalC(okToWrite);
      else signalC(okToRead);}
```

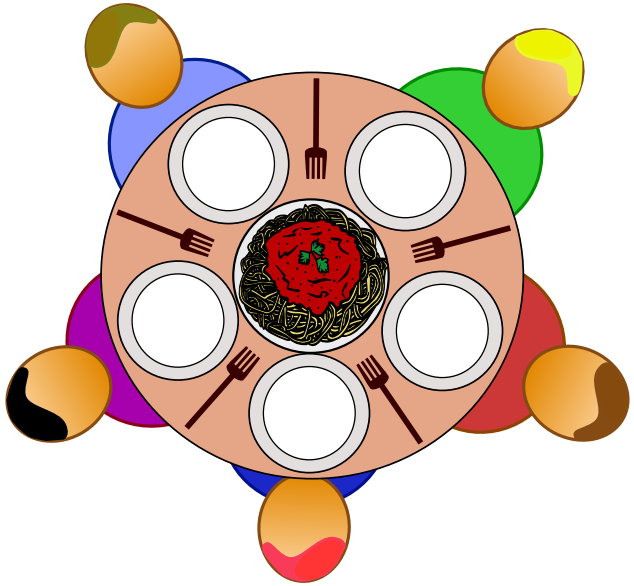
Code des Readers:

```
(1)   RW.startRead();
(2)   Lese;
(3)   RW.endRead();
```

Code des Writers:

```
(1)   RW.startWrite();
(2)   Schreibe;
(3)   RW.endWrite();
```

Speisende Philosophen mit Monitoren



Speisende Philosophen mit Monitoren (2)

- Modellierung nicht mit Semaphore
- Gabeln werden nur implizit benutzt
- Stattdessen: Für jeden Philosoph wird notiert, wieviele Gabeln vor ihm liegen (0,1 oder 2)
- Philosoph isst nur dann, wenn er zwei Gabeln hat
- Modifikation der Anzahl Gabeln:
  - Gabeln vor Philosoph i, werden modifiziert durch: linker und rechter Nachbar von Philosoph i

Speisende Philosophen mit Monitoren (3)

```
monitor Forks {
    constant anzahlPhil := N;
    int array [1,...,anzahlPhil] forks := [2,...,2];
    condition array [1,..., anzahlPhil] okToEat;

    takeForks(i) {
        if forks[i] ≠ 2 then waitC(okToEat[i]);
        forks[i+1] := forks[i+1]-1;
        forks[i-1] := forks[i-1]-1;}

    releaseForks(i) {
        forks[i+1] := forks[i+1]+1;
        forks[i-1] := forks[i-1]+1;}

        if forks[i+1] = 2 then signalC(okToEat[i+1]);
        if forks[i-1] = 2 then signalC(okToEat[i-1]);}
}
```

Anmerkung: i+1, i-1 sind „modulo N“ gemeint

Speisende Philosophen mit Monitoren (3)

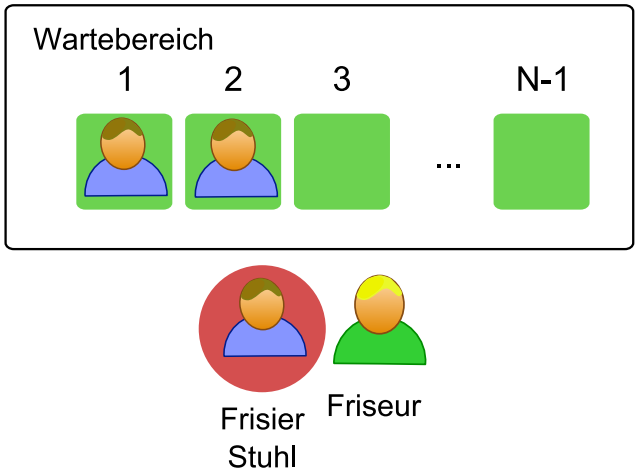
Philosoph i:

```
loop forever
(1)  Denke;
(2)  Forks.takeForks(i);
(3)  Esse;
(4)  Forks.releaseForks(i);
end loop
```

Eigenschaften

- Lösung ist Deadlock-frei
- aber nicht Starvation-frei!

Sleeping Barber mit Monitoren



Sleeping Barber mit Monitoren (2)

```
monitor Barbershop {
 int wartend:= 0;
 condition kundenVorhanden;
 condition friseurBereit;
 condition ladenVerlassen;
 // Methoden für den Friseur
 nehmeKunden() {
 if wartend = 0 then
    waitC(kundenVorhanden);
 wartend := wartend - 1;
 signalC(friseurBereit)}
 beendeKunden() {
 waitC(ladenVerlassen)}
 ...
```


Sleeping Barber mit Monitoren (3)

```
...
// Methoden für den Kunden
bekommeFrisur() {
if wartend < N then
   wartend := wartend + 1;
   signalC(kundenVorhanden)
   waitC(friseurBereit)
   return(True);
else return(False);}
verlasseLaden(){
   signalC(ladenVerlassen)}
}
```

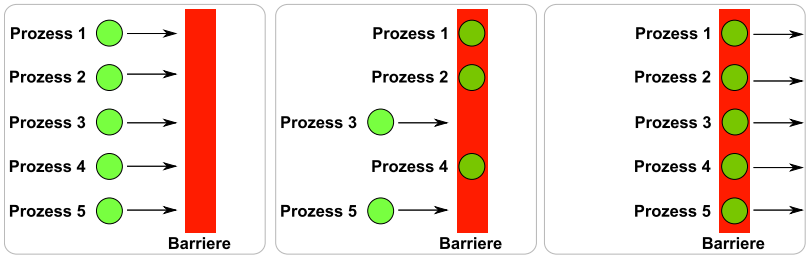
Friseur:

```
 loop forever
    Barbershop.nehmeKunden();
    Frisiere;
    Barbershop.beendeKunden()
```

Kunde

```
 if Barbershop.bekommeFrisur()
 then
     erhalte Frisur;
     verlasseLaden();
```


Barrier mit Monitor



Barrier mit Monitor (2)

```
monitor Barrier {
 int angekommen:= 0;
 int maxProzesse := N;
 condition alleAngekommen;
 barrier (){
  angekommen := angekommen + 1;
  if angekommen < maxProzesse then
     waitC(alleAngekommen);
  else
     angekommen := 0;
     signalAllC(alleAngekommen);
  }
}
```

Prozess i:

```
loop forever
  Code vor dem Barrier
  Barrier.barrier()
  Code nach dem Barrier
end loop
```

`signalAllC`: neue Operation: Alle wartenden Prozesse werden entblockiert

Monitore in Java

- Keine expliziten Monitore, aber
- synchronized Methoden
- Idee dabei: Jedes Objekt hat eine Lock-Variable
- Werden Methoden mit `synchronized` modifiziert, dann ist der exklusive Zugriff auf das Objekt über diese Methode sichergestellt

Monitore in Java (2)

```
class MonitoredClass {
 ... Attribute ...
  synchronized method1 {...}

  synchronized method2 {...}
```

- Nur ein Thread gleichzeitig führt Methode aus
- wenn method1 im Rumpf method2 aufruft, dann wird der Lock beibehalten.

Monitore in Java (3)

Erlaubt:

```
class MonitoredClass {
 ... Attribute ...
  synchronized method1 {...}

  method2 {...}
```

- method2 ist nicht synchronisiert.
- method2 kann jederzeit aufgerufen werden

Monitore in Java (4)

Statt Condition Variables

- Operationen `wait`, `notify`, `notifyAll`
- Nur eine Queue pro Objekt (zwischen verschiedenen Condition Variablen kann nicht unterschieden werden)
- `wait()`: Thread wartet an der Queue des Objekts
- `notify()`: Ein wartender Thread wird entblockiert, aber: Aufrufender Prozess behält Lock!
- `notifyAll()`: Alle wartende Threads werden entblockiert, aber:
  Aufrufender Prozess behält Lock!
- Wartende Threads haben gleiche Priorität wie neue!
- Entspricht $W = E < S$

Monitore in Java (5)

- Ein entblockierter Prozess kann möglicherweise erst dann den Lock erhalten, wenn Bedingung wieder falsch!
- Lösung: Bedingung erneut prüfen!

```
while (not Bedingung)
    wait();
```


Monitore in Java (6)

Wie programmiert man mit mehreren Condition Variablen?

Gewünscht, aber geht nicht:

```
synchronized method1() {
  while (x==0)
     wait();...        }

synchronized method2() {
  while (y==0)
     wait();...        }
...
if some condition
     notifyAll();
else
     notifyAll();
```


Monitore in Java (7)

- `synchronized` kann nicht nur für Methoden, sondern für beliebige Codeblöcke benutzt werden.
- Angabe des Objekts notwendig

```
Object obj = new Object();

synchronized (obj) {
   Kritischer Abschnitt
}
```

- Auch bestehende Objekte können verwendet werden.

Beispiele zu synchronized (1)

```
class EinThread implements Runnable {
 public void run() { printText(); }
 synchronized void printText () {
  System.out.println("(1) Thread " + (Thread.currentThread()).getId());
  System.out.println("(2) Thread " + (Thread.currentThread()).getId());
  System.out.println("(3) Thread " + (Thread.currentThread()).getId());
 }
}

public class Main {
 public static void main(String args[]) {
  for (int k = 1; k <= 10; k++) {
   (new Thread(new EinThread())).start();
  }
 }
}
```

Falsch: Da jedes EinThread-Objekt ein eigener Monitor ist

Beispiele zu synchronized (2)

```
class PrintMonitor {
 synchronized void printText () {
  System.out.println("(1) Thread " + (Thread.currentThread()).getId());
  System.out.println("(2) Thread " + (Thread.currentThread()).getId());
  System.out.println("(3) Thread " + (Thread.currentThread()).getId());
 }
}
class EinThread implements Runnable {
 PrintMonitor pm;
  EinThread(PrintMonitor pm) { this.pm = pm;}
  public void run() { pm.printText(); }
}
public class Main {
 public static void main(String args[]) {
  PrintMonitor pm = new PrintMonitor();
  for (int k = 1; k <= 10; k++) {
   (new Thread(new EinThread(pm))).start();
  }
 }
}
```


Beispiele zu synchronized (3)

```
// ...  PrintMonitor und EinThread wie zuvor

class NochEinThread implements Runnable {
 PrintMonitor pm;
 NochEinThread(PrintMonitor pm) { this.pm = pm; }
 public void run() {
  synchronized  (pm) {  // Synchronized am selben Objekt
   System.out.println("(A) Thread " + (Thread.currentThread()).getId());
   System.out.println("(B) Thread " + (Thread.currentThread()).getId());
   System.out.println("(C) Thread " + (Thread.currentThread()).getId());
  }
 }
}
public class Main {
 public static void main(String args[]) {
  PrintMonitor pm = new PrintMonitor();
  for (int k = 1; k <= 10; k++) {
   (new Thread(new EinThread(pm))).start();
   (new Thread(new NochEinThread(pm))).start();
  }}}
```


Beispiele in Java: Producer / Consumer (1)

```
import java.util.LinkedList;

// Die Klasse fuer den Buffer

class BBufferMon<V> {
 private LinkedList<V> buffer = new LinkedList<V>();
 private Integer length; // Fuellstand
 private Integer size;   // maximale Gr"oesse

 BBufferMon(Integer n){
  size = n;
  length = 0;
 }
```

Beispiele in Java: Producer / Consumer (2)

```
 synchronized void produce(V elem) {
      while (length == size) {
        try {wait();} catch (InterruptedException e) {};
      }
      buffer.add(elem);
      length++;
      notifyAll();
 }
 synchronized public V consume() {
      while (length == 0) {
        try {wait();} catch (InterruptedException e) {};
      }
      V e = buffer.removeFirst();
      length--;
      notifyAll();
      return e;
 }
}
```


Beispiele in Java: Producer / Consumer (3)

Klassen für Erzeuger und Verbraucher:

```
class Producer extends Thread {
  BBufferMon<Integer> buff;
  Integer number;
  Producer(BBufferMon<Integer> b, Integer i) { buff = b;  number = i;}
  public void run() { for (int i = 1; i <= 10; i++) {buff.produce(i);} }
}

class Consumer extends Thread {
  BBufferMon<Integer> buff;
  Integer number;
  Consumer(BBufferMon<Integer> b,Integer i) { buff = b; number = i;}
  public void run() { for (int i = 1; i <= 50; i++)
                        {Integer e = buff.consume();} }
}
```


Beispiele in Java: Producer / Consumer (4)

Main-Klasse und Methode:

```
class Main {
public static void main(String[] args) {
 // Puffer-Objekt erzeugen
      BBufferMon<Integer> b = new BBufferMon<Integer>(5);
 // Erzeuger-Threads erzeugen
      for (int i=1; i <= 50; i++) {
        Producer q = new Producer(b,i);
        q.start();
       }
 // Verbraucher-Threads erzeugen
      for (int i=1; i <= 10; i++) {
        Consumer q = new Consumer(b,i);
        q.start();
       }
      while (true) {}
    }
}
```


Beispiele in Java: Reader/Writers (aus Ben-Ari Buch)

```
class RWMonitor {
  volatile int readers = 0;
  volatile boolean writing = false;

  synchronized void StartRead() {
    while (writing)
      try {
         wait();
      } catch (InterruptedException e) {}
    readers = readers + 1;
    notifyAll();
  }

  synchronized void EndRead() {
    readers = readers - 1;
    if (readers == 0) notifyAll();
  }
...
```

Beispiele in Java: Reader/Writers

```
  synchronized void StartWrite() {
    while (writing || (readers != 0))
      try {
         wait();
      } catch (InterruptedException e) {}
    writing = true;
  }
  synchronized void EndWrite() {
    writing = false;
    notifyAll();
  }
}
```