

## Programmierprimitiven: Monitore

Prof. Dr. David Sabel

LFE Theoretische Informatik



- 1 Monitore
  - Einführung
  - Condition Variables
  - Condition Expressions
- 2 Einige Anwendungsbeispiele mit Monitoren
  - Readers&Writers
  - Philosophen
  - Sleeping Barber
  - Barrieren
- 3 Monitore in Java

## Vorteile

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

## 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 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)

---

```
class Konto {  
    int Saldo;  
    int Kontonummer;  
    int KundenId  
  
    abheben(int x) {  
        Saldo := Saldo - x;  
    }  
  
    zubuchen(int x) {  
        Saldo := Saldo + x;  
    }  
}
```

## 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

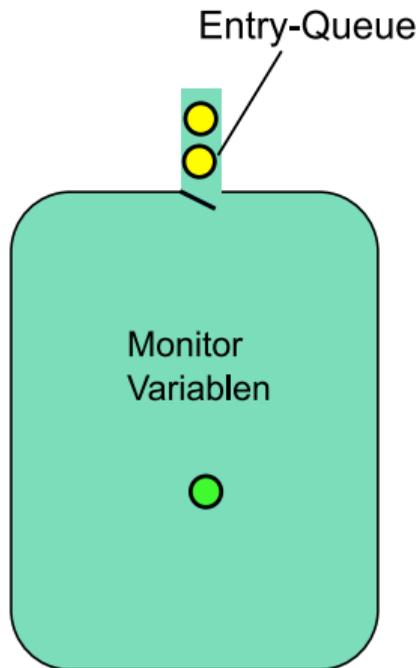
## Monitore (2)

---

```
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 (blockierte) Prozesse**
- **Prozess im Monitor**

- 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  
    ...  
    inKritischemAbschnitt(...) {  
        Kritischer Abschnitt;  
    }  
}
```

Prozess i:

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

### **Vorteil gegenüber Semaphore**

Setzen / Entfernen des Locks geschieht implizit

## Vor- und Nachteile bisher

---

### **Vorteil gegenüber Semaphore**

Setzen / Entfernen des Locks geschieht implizit

### **Nachteil gegenüber Semaphore**

Viele Probleme benötigen explizites Blockieren von Prozessen

### **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.

### **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 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  $\neq$  empty then  
    Q := head(cond); // Erster Prozess der Queue  
    cond := tail(cond); // wird aus Queue entfernt  
    Q.state := ready; // und entblockiert  
}
```

### Semantik von signalC

```
signalC(cond) {  
  if cond  $\neq$  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 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

`wait(Sem)` kann zum Blockieren führen,  
muss aber nicht

Monitore (Condition Variable cond)

# Vergleich wait, signal, waitC, signalC

---

Semaphore Sem

wait(Sem) kann zum Blockieren führen,  
muss aber nicht

Monitore (Condition Variable cond)

waitC(cond) blockiert den Prozess **stets**

## Vergleich wait, signal, waitC, signalC

---

Semaphore Sem

wait(Sem) kann zum Blockieren führen,  
muss aber nicht

signal(Sem) hat stets einen Effekt: Ent-  
blockieren eines Prozesses oder Erhöhen von  
Sem.V

Monitore (Condition Variable cond)

waitC(cond) blockiert den Prozess **stets**

## Vergleich wait, signal, waitC, signalC

---

### Semaphore Sem

`wait(Sem)` kann zum Blockieren führen, muss aber nicht

`signal(Sem)` hat stets einen Effekt: Entblockieren eines Prozesses oder Erhöhen von  $Sem.V$

### Monitore (Condition Variable cond)

`waitC(cond)` blockiert den Prozess **stets**

`signalC(cond)` kann effektivlos sein: Entweder Prozess in cond wird entblockiert, oder effektivlos, 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

---

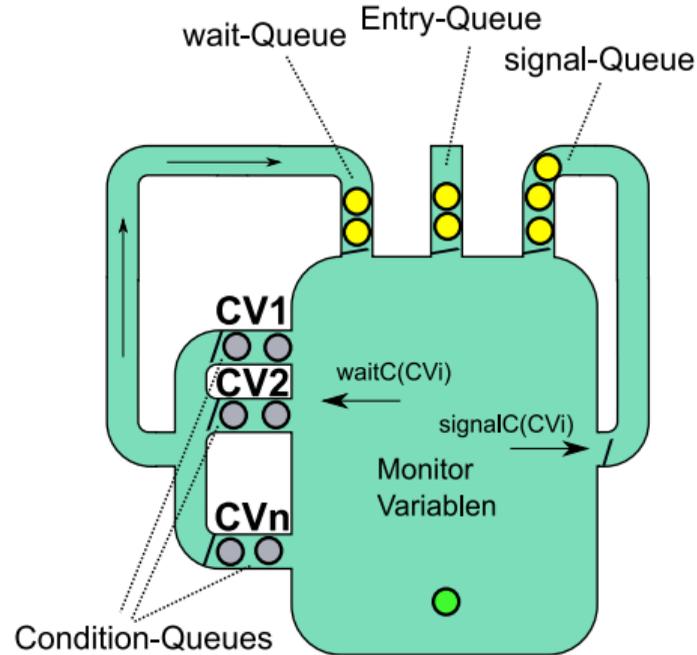
- Erinnerung: Beim Durchführen von `signalC` ist nicht eindeutig, wer den Monitor-Lock anschließend haben soll
- Es gibt verschiedene Varianten
- Wenn gewecketer Prozess weiterrechnen darf:  
Wo wartet der signalisierende Prozess?

# 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:
- Es gibt mehrere Queues:
  - Condition-Queue (eine pro Condition-Variable)
  - wait-Queue (eine pro Monitor)
  - signal-Queue (eine pro Monitor)
  - entry-Queue (eine pro Monitor)

# Monitor-Modellierung mit drei Queues für die Condition Variable



- **Wartende Prozesse: Verwaltung durch Monitor.lock**
- **Wartende Prozesse: Verwaltung durch Condition**
- **Prozess im Monitor**

# 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

# 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?

## 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  $E = W < S$

3  $E = S < W$

4  $E < W = S$

5  $E < W < S$

6  $E < S < W$

7  $E > W = S$

8  $E = S > W$

9  $S > E > W$

10  $E = W > S$

11  $W > E > S$

12  $E > S > W$

13  $E > W > S$

## Prioritäten für die Queues (2)

### 13 Möglichkeiten

1  $E = W = S$

2  $E = W < S$

3  $E = S < W$

4  $E < W = S$

5  $E < W < S$

6  $E < S < W$

7  $E > W = S$  nicht sinnvoll

8  $E = S > W$

9  $S > E > W$

10  $E = W > S$

11  $W > E > S$

12  $E > S > W$  nicht sinnvoll

13  $E > W > S$  nicht sinnvoll

- Neue Prozesse würden alten vorgezogen
- W und S Prozesse müssen evtl. beliebig lange Warten
- Die W und S Queues werden evtl. immer voller

### 13 Möglichkeiten

- 1  $E = W = S$
- 2  $E = W < S$
- 3  $E = S < W$
- 4  $E < W = S$
- 5  $E < W < S$
- 6  $E < S < W$
- 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

- Ähnliches Problem
- Neue Prozesse würden alten vorgezogen
- $W$  bzw.  $S$  Prozesse müssen evtl. beliebig lange Warten
- Die  $W$  bzw.  $S$  Queues werden evtl. immer voller
- $\Rightarrow E$  sollte nicht größere als  $W$  oder  $S$  sein

## 13 Möglichkeiten

- 1  $E = W = S$
- 2  $E = W < S$
- 3  $E = S < W$
- 4  $E < W = S$
- 5  $E < W < S$
- 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

- Erst darf der wartende Prozess weiterrechnen
- Dann der signalisierende
- Am Schluss neue Prozesse
- Bezeichnungen:

Signal and Urgent Wait  
(Hoare 1974)

oder auch

Immediate Resumption  
Requirement

## Prioritäten für die Queues (2)

---

### 13 Möglichkeiten

- 1  $E = W = S$
- 2  $E = W < S$
- 3  $E = S < W$
- 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

## Prioritäten für die Queues (2)

---

### 13 Möglichkeiten

- 1  $E = W = S$
- 2  $E = W < S$
- 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

### 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

- Verwendet in Java
- Problematisch: Bedingung zur Condition Variable kann wieder falsch sein, bis der Prozess wirklich wieder aktiv im Monitor ist
- Lösung: Zirkuläres Prüfen und Warten:

Beispiel: Erniedrige  $s$  erst dann, wenn  $s \neq 0$

```
while s = 0 do
    waitC(slstNichtNull);
    s := s - 1;
```

# 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  $\neq$  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  $\neq$  0 or countW  $\neq$  0  
    then waitC(okToWrite);  
    countW := 1;}
```

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

Code des Readers:

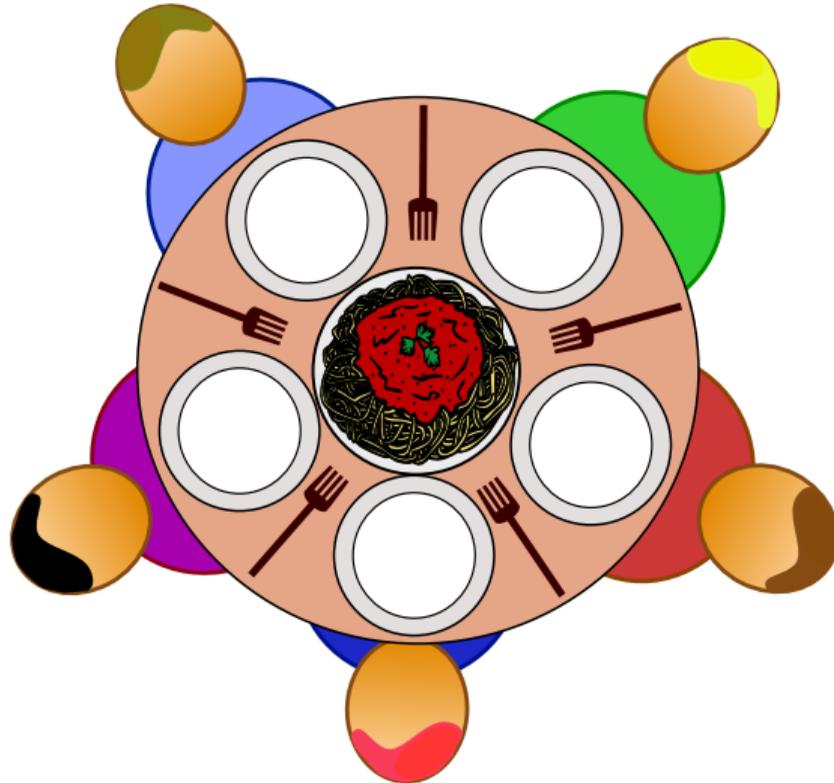
- (1) RW.startRead();
- (2) Lese;
- (3) RW.endRead();

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]  $\neq$  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;
```

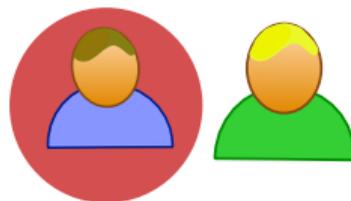
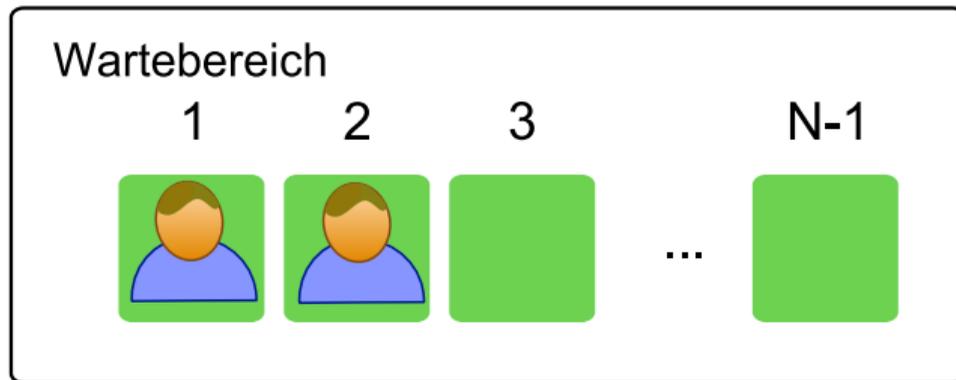
```
(4)  Forks.releaseForks(i);
```

```
end loop
```

### Eigenschaften

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

# Sleeping Barber mit Monitoren



Frisier  
Stuhl      Friseur

## 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)

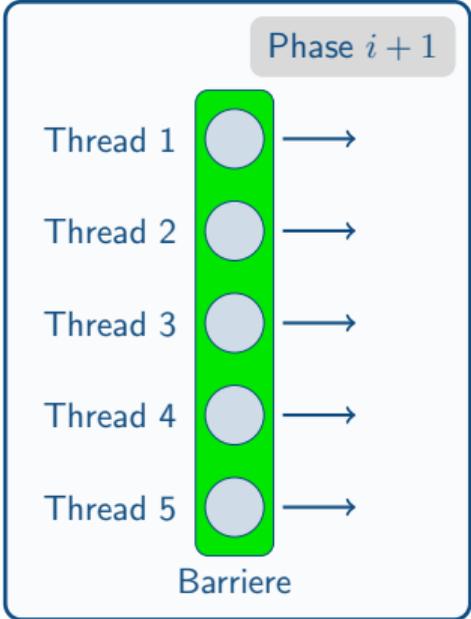
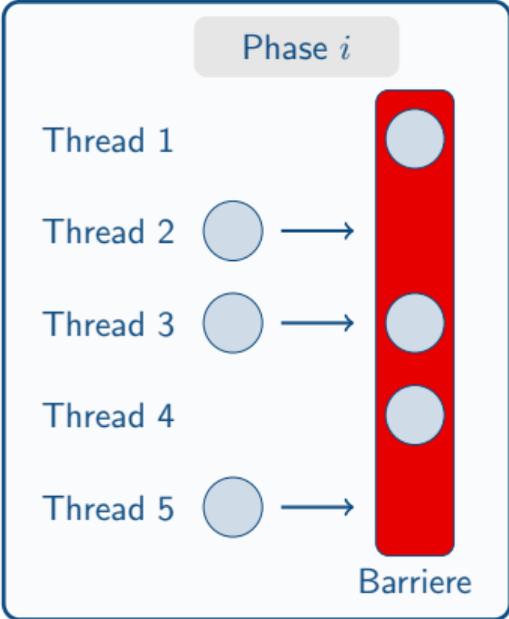
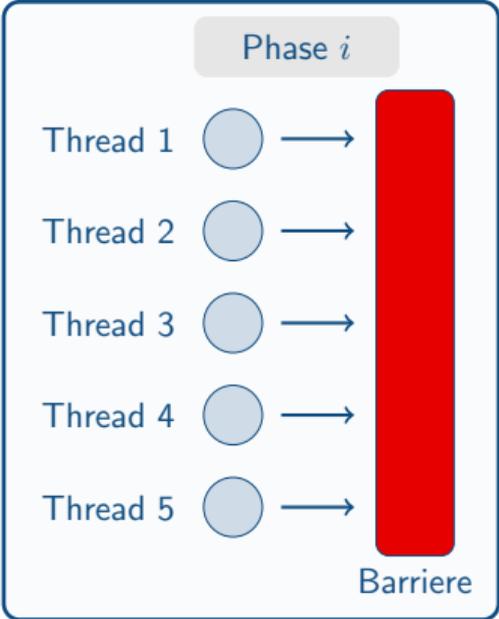
---

```
...  
// 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();
```

# Barriere mit Monitor



## Barriere 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 der Barriere
    Barrier.barrier()
    Code nach der Barriere
  end loop
```

signalAllC: neue Operation: Alle wartenden Prozesse werden entblockiert

- 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.

### Erlaubt:

```
class MonitoredClass {  
    ... Attribute ...  
  
    synchronized method1 {...}  
  
    method2 {...}
```

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

## Statt Condition Variables

- Operationen `wait`, `notify`, `notifyAll`
- Nur eine Queue pro Objekt  
(zwischen verschiedenen Condition Variablen kann nicht unterschieden werden)

### 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!

### 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() {  
    if (x==0)  
        wait();...    }
```

```
synchronized method2() {  
    if (y==0)  
        wait();...    }
```

...

```
if some condition  
    notify(auf  $x \neq 0$  wartende Prozesse)  
else  
    notify(auf  $y \neq 0$  wartende Prozesse)
```

## Monitore in Java (6)

---

Wie programmiert man mit mehreren Condition Variablen?

```
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();
        }
    }
}
```

## 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();
}
}
```