

# Kapitel 5

## Monitore und Synchronisationsbedingungen

Prof. Dr. Rolf Hennicker

22.06.2017

## 5.1 Synchronisationsbedingungen

*Bisherige Verwendung von Monitoren:*

Verkapselung von Daten, Zugriffsoperationen unter wechselseitigem Ausschluss.

*Jetzt zusätzlich:*

Synchronisationsbedingungen zur Prozesskooperation.

*Idee:*

Bestimmte Monitoroperationen dürfen erst dann ausgeführt werden, wenn eine bestimmte Bedingung (abhängig vom Monitorzustand) erfüllt ist.

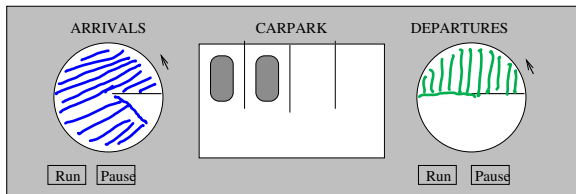
Z.B.

Puffer nicht voll: erst dann Ablegen eines Elements möglich,

Puffer nicht leer: erst dann Holen eines Elements möglich.

## Beispiel (Carpark):

Simulation eines Parkhauses mit beschränkter Kapazität



Keine Ankunft, wenn Parkhaus voll ist.

Keine Abfahrt, wenn Parkhaus leer ist.

## 5.2 Modellierung von Synchronisationsbedingungen

### Beispiel (Modellierung des Carpark):

ARRIVALS = (arrive → ARRIVALS).

DEPARTURES = (depart → DEPARTURES).

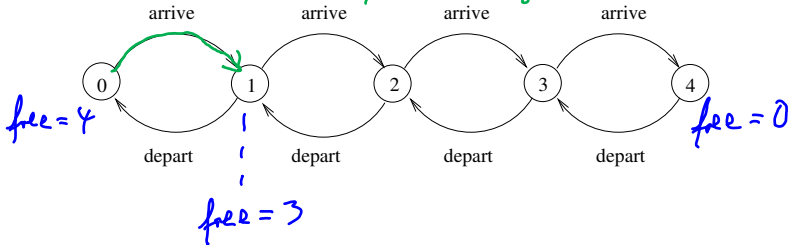
CARPARKCONTROL(N=4) = SPACES[N],

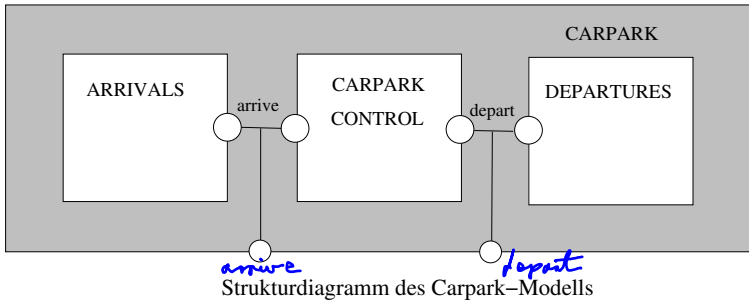
SPACES[free:0..N] = (when  $\text{free} > 0$  arrive → SPACES[free-1]  
| when  $\text{free} < N$  depart → SPACES[free+1]).

*aufangs freie Plätze*

||CARPARK = (ARRIVALS || CARPARKCONTROL(4) || DEPARTURES).

*atomar in FSP, nicht in Java*





## Schema zur Modellierung von Synchronisationsbedingungen *Pattern*

$P1 = (op1 \rightarrow P1).$

$P2 = (op2 \rightarrow P2).$

$MONITOR(N=...) = MON[init],$

$MON[state:Data] =$

$(when (cond1) op1 \rightarrow MON[nextState1]$

$| when (cond2) op2 \rightarrow MON[nextState2]).$

$\parallel SYS = (P1 \parallel P2 \parallel MONITOR).$

*Anfangszustand des Monitors*

*Monitorzustand*

*Monitoroperationen  
mit Zustandswechsel*

### Bemerkung:

- Die Synchronisationsbedingungen werden durch Wächter für Aktionen ausgedrückt.
- Die Bedingungen hängen vom Monitorzustand ab.
- Die Monitorzustände werden durch Prozessindizes (in lokalen Prozessen) modelliert.

## Klassifizierung von Prozessen

- ▶ Im Carpark-Beispiel sind ARRIVALS und DEPARTURES *aktive* Komponenten. Sie initiieren die Aktionen "arrive" und "depart" (Output-Aktionen).
- ▶ CARPARKCONTROL ist eine *passive* Komponente, die auf Aktionen reagiert (Input-Aktionen).
- ▶ Aktive Komponenten werden durch Threads implementiert. Passive Komponenten, die zur Überwachung von Synchronisationsbedingungen dienen, werden durch Monitore implementiert.

## 5.3 Implementierung von Synchronisationsbedingungen in Java

Zur Realisierung von Synchronisationsbedingungen verwenden wir die Methoden

```
public final void wait() throws InterruptedException
public final void notify() bzw. notifyAll()
```

der allgemeinsten Klasse Object.

### Wirkungsweise:

Jedes Objekt obj besitzt neben seiner Sperre eine Menge w von wartenden Threads. Die folgenden Operationen werden in einer unteilbaren Aktion ausgeführt:

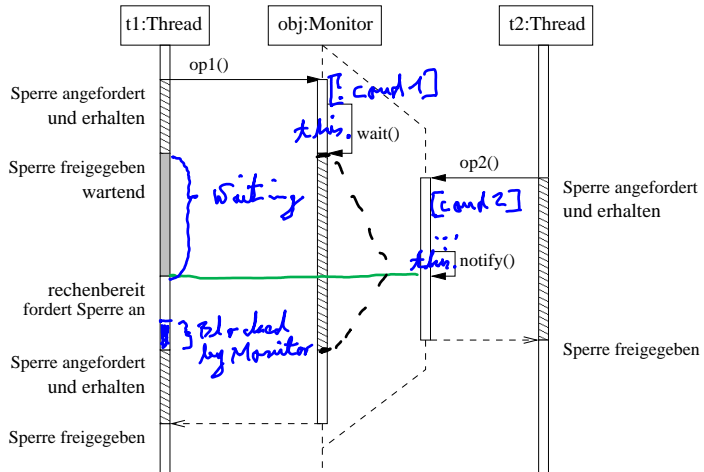
- ▶ obj.wait(): *meint this.wait();*
  - ▶ Der gerade ausgeführte Thread t gibt die Sperre von obj frei.
  - ▶ t wird in die Wartemenge w von obj eingereiht.
- ▶ obj.notify(): *meint this.notify();*
  - ▶ Ein beliebiger Thread wird aus w freigelassen ("geweckt") und ist wieder rechenbereit. *Renumberable*
- ▶ obj.notifyAll():
  - ▶ Alle Threads werden aus der Wartemenge w von obj freigelassen.



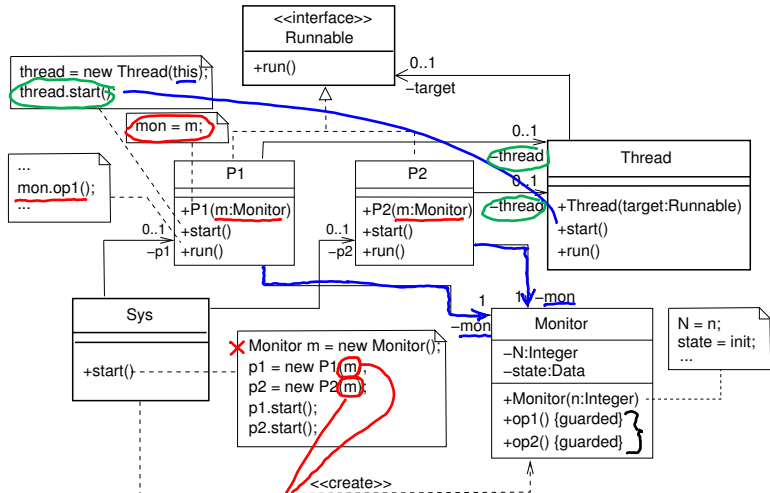
**Beachte:**

1. `wait` und `notify` (bzw. `notifyAll`) dürfen nur innerhalb von synchronisierten Blöcken aufgerufen werden.
2. Wird ein Thread `t` geweckt, dann muss er sich wieder um die Sperre von `obj` bewerben. Die Sperre bleibt solange bei demjenigen Thread, innerhalb dessen "notify" aufgerufen wurde, bis dieser sie freigibt.  
(“Signal and Continue“-Prinzip)
3. Sobald ein geweckter Thread `t` die Sperre von `obj` wieder erhält, fährt er dort fort, wo er aufgehört hat.

## Sequenzdiagramm mit "wait" und "notify"



# Schema zur Implementierung von Synchronisationsbedingungen



```
public class Monitor {
    private int N;
    private Data state;

    public Monitor(int n) {
        N = n;
        state = init;
    }

    public synchronized void op1() throws InterruptedException {
        while (!cond1aka. wait());

        // modify monitor state

        notify(); // or notifyAll();
    }

    public synchronized void op2() throws InterruptedException {
        while (!cond2 wait());

        // modify monitor state

        notify(); // or notifyAll();
    }
}
```

## Regel zur Implementierung von Synchronisationsbedingungen mit Monitoren

FSP: `when (cond) op → MONITOR[nextState]`

```
Java: public synchronized void op() throws InterruptedException {
    while (!cond) wait();
    ... // monitor state = nextState
    notifyAll(); oder notify();
}
```

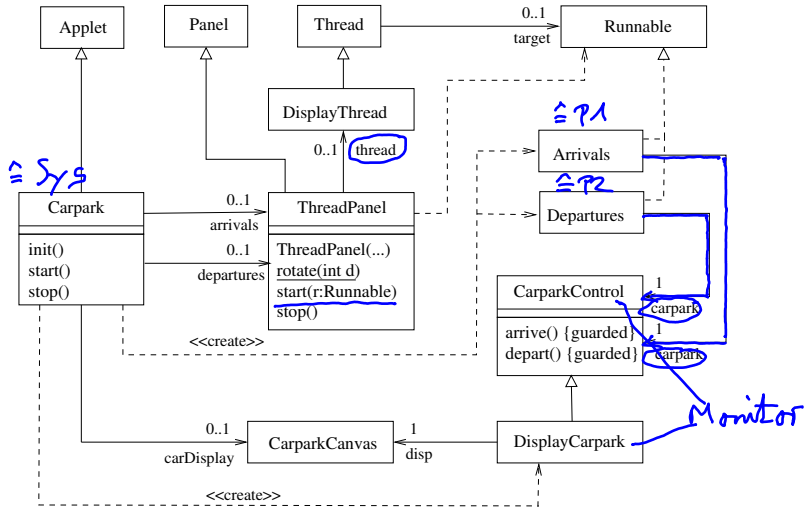
### Beachte:

Die `while`-Schleife ist notwendig, weil nach Verlassen des Waiting-Zustands nicht garantiert ist, dass die Bedingung "cond" gilt. Insbesondere ist dies nicht garantiert, wenn der Thread wieder im Zustand Running ist.

Gründe können sein:

- ▶ Die Bedingung wurde zwischenzeitlich gar nicht verändert.
- ▶ Die Wartebedingung "!"cond" war zwar zwischenzeitlich nicht mehr erfüllt, wurde aber bis der geweckte Thread wieder an die Reihe kommt von einem anderen Thread wieder auf "wahr" gesetzt.

# Beispiel (Implementierung des Carpark):



Realisierung des FSP-Monitors CARPARKCONTROL  
 durch die Klasse CarparkControl:

```

class CarparkControl {
    protected int capacity; // capacity = N in FSP
    protected int free; //freie Plaetze

    CarparkControl(int n) {
        capacity = n; free = n;
    }
    synchronized void arrive() throws InterruptedException {
        while (free <= 0) wait();
        free--;
        notify(); // hier kann nur ein anderer Prozess wartend sein
    }
    synchronized void depart() throws InterruptedException {
        while (free >= capacity) wait();
        free++;
        notify(); // hier kann nur ein anderer Prozess wartend sein
    }
}
  
```

*reine nach dem Schema implementiert*

*≡ init ≡ N im Bsp.*

*!(free > 0)*

*!(free < capacity)*

Realisierung der aktiven Komponenten ARRIVALS und DEPARTURES durch die Klassen Arrivals und Departures:

```
class Arrivals implements Runnable {
    private CarparkControl carpark; //Monitor

    Arrivals(CarparkControl c) {
        carpark = c;
    }
    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(330);
                ✕ carpark.arrive(); //Monitoraufruf
                ThreadPanel.rotate(30);
            }
        } catch (InterruptedException e) {}
    }
}
```



```
class Departures implements Runnable {
    private CarparkControl carpark; //Monitor

    Departures(CarparkControl c) {
        carpark = c;
    }
    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(180);
                ✕ carpark.depart(); //Monitorkauf
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e) {}
    }
}
```

Realisierung des Applets Carpark:

$\hat{=}$  Sys

```
class Carpark extends Applet {
    ThreadPanel arrivals;
    ThreadPanel departures;
    ...
    public void start() {
        CarparkControl (C) = new DisplayCarpark(carDisplay, places);
        arrivals.start(new Arrivals(C));
        departures.start(new Departures(C));
    }
    public void stop() {
        arrivals.stop();
        departures.stop();
    }
}
```

*shared  
Monitorobjekt*

## 5.4 Semaphore

## Das Semaphore-Konzept [Dijkstra 1968]

Ein Semaphore  $s$  besteht aus

- ▶ einer Integer-Variablen  $v$  zusammen mit
- ▶ einer Warteschlange  $w$  für die bezüglich des Semaphors blockierten Prozesse.

$$v = 0 \quad w = \langle P_1 \dots P_m \rangle$$

Es gibt genau zwei Operationen, die von einem Prozess  $P$  auf ein Semaphore  $s$  angewendet werden können:

$\text{down}(s):$  if  $v > 0$  then  $v = v - 1$ ;  $P$  ist im krit. Bereich  
 else  $w = w \cup \{P\}$ , d.h.  $P$  ist blockiert  
 $\text{up}(s):$  if  $w = \emptyset$  then  $v = v + 1$ ;  
 else  $w = w \setminus \{Q\}$  für einen vorher blockierten Prozess  $Q$

## Bemerkung

- ▶ Es wird vorausgesetzt, dass ein Semaphore mit einem Wert  $\geq 0$  (für  $v$ ) initialisiert ist.

Der Wert von  $v$  gibt an wieviele Prozesse noch in dem kritischen Bereich dürfen.

$Q$  ist dann sofort im kritischen Bereich

## Simulation von Semaphoren in FSP

const Max = 3

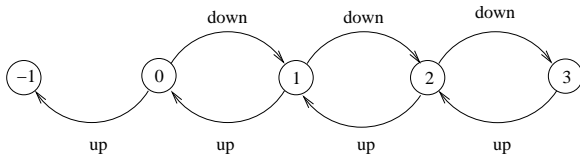
range Int = 0..Max (*simuliert die Integers in FSP*)

SEMAPHORE(initial=3) = SEMA[initial],

SEMA[v:Int] = ( when (v > 0) down → SEMA[v-1]  
 | up → SEMA[v+1]),

SEMA[Max+1] = ERROR.

Zugehöriges LTS:



initial: Gibt an, wieviele Prozesse von Beginn an in den kritischen Bereich dürfen.

## Wechselseitiger Ausschluss mit Semaphoren

$A = (a.do \rightarrow a.down \rightarrow a.critical \rightarrow a.up \rightarrow A).$

$B = (b.do \rightarrow b.down \rightarrow b.critical \rightarrow b.up \rightarrow B).$

$\parallel SYS = (A \parallel B \parallel \{a,b\}::SEMAPHORE(1)).$

## Simulation von Semaphoren in Java

```
class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }
    public synchronized void down() throws InterruptedException {
        while (value==0) wait();
        value--;
    }
    public synchronized void up() {
        value++;
        notifyAll();
    }
}
```