LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prof. Dr. R. Hennicker
M.Sc. Annabelle Klarl

# Entwurf und Implementierung paralleler Programme

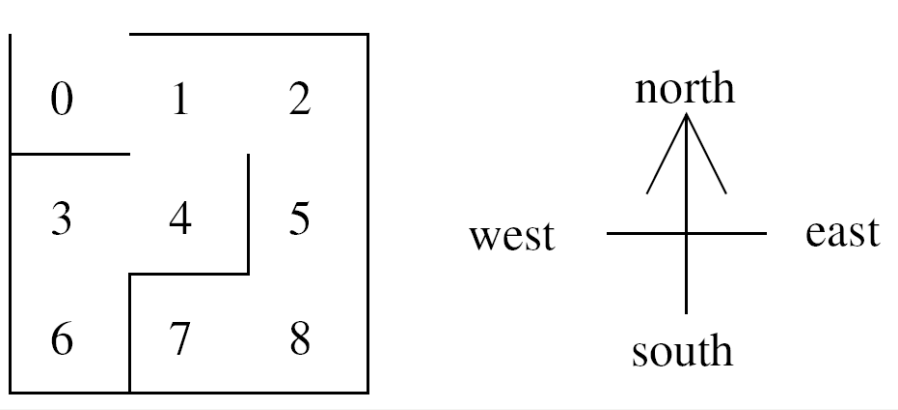
SoSe 2016

Übungsblatt 8

# Übungsblatt 8

## Aufgabe 1

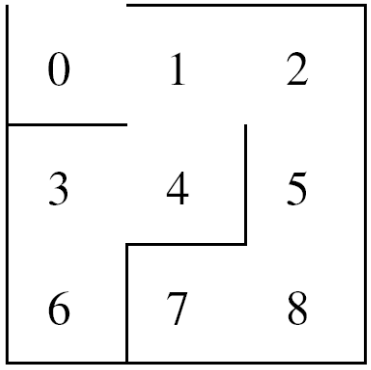
Gegeben sei das folgende Labyrinth.

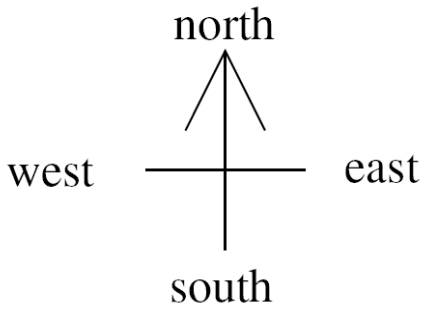



Geben Sie einen parametrisierten FSP-Prozess LABYRINTH(Start = …) an, so dass für **jeden Startpunkt S** durch Deadlock-Analyse des Prozesses LABYRINTH(S) der kürzeste Weg aus dem Labyrinth gefunden werden kann.

# Modellierung des Labyrinths

- **Idee**:
  - Modelliere einen Prozess mit einem Index für die aktuelle Position
  - Modelliere das Labyrinth durch die möglichen Bewegungen je Position (in Himmelsrichtungen), z.B. ist in Position 7 nur die Bewegung nach OSTEN möglich, mit der man Position 8 erreicht
- Bewegung aus dem Labyrinth heraus (also von 0 nach NORDEN) ist als STOP (="Deadlock") modelliert




LÖSUNG

# FSP-Prozess

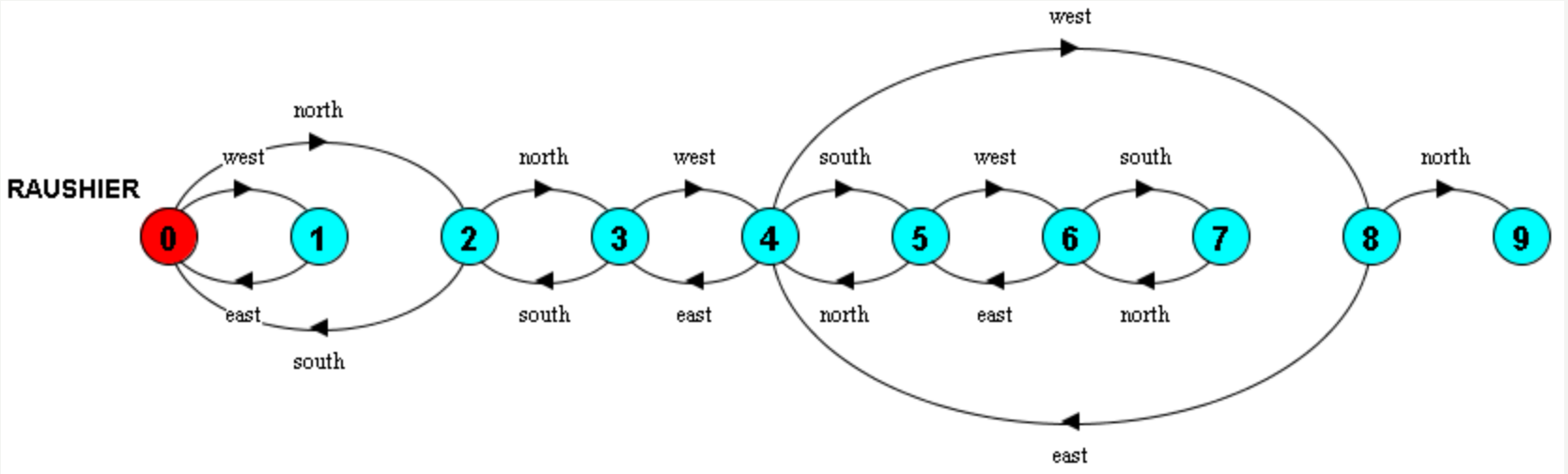
```
LABYRINTH(Start=8) = P[Start],
P[0]= (north->STOP|east->P[1]),
P[1]= (west->P[0]|east->P[2]|south->P[4]),
P[2]= (west->P[1]|south->P[5]),
P[3]= (east->P[4]|south->P[6]),
P[4]= (north->P[1]|west->P[3]),
P[5]= (north->P[2]|south->P[8]),
P[6]= (north->P[3]),
P[7]= (east->P[8]),
P[8]= (west->P[7]|north->P[5]).

||RAUSHIER = LABYRINTH(8).
```

# Sicherheits-Analyse

**LABYRINTH(8)**

Trace to DEADLOCK:

north
north
west
west
north



# Übungsblatt 8

## Aufgabe 2

Die folgende Klasse (nächste Folie) simuliert das Verhalten von Semaphoren in Java.

# Semaphoren in Java

```
public class Semaphore {
    private int value;
    public Semaphore (int initial) {
      value = initial;
    }
    public synchronized void down() throws IE... {
      while (value==0) wait();
      value--;
    }
    public synchronized void up() {
      value++;
      notify();
    }
}
```

Eine Semaphore stellt sicher, dass nur eine bestimmte Anzahl an Prozessen gleichzeitig auf eine kritische Ressource zugreifen dürfen.

Dabei stellt sie nur sicher, dass ihr Wert nur bis maximal 0 erniedrigt werden kann; die Erhöhung wird nicht kontrolliert.

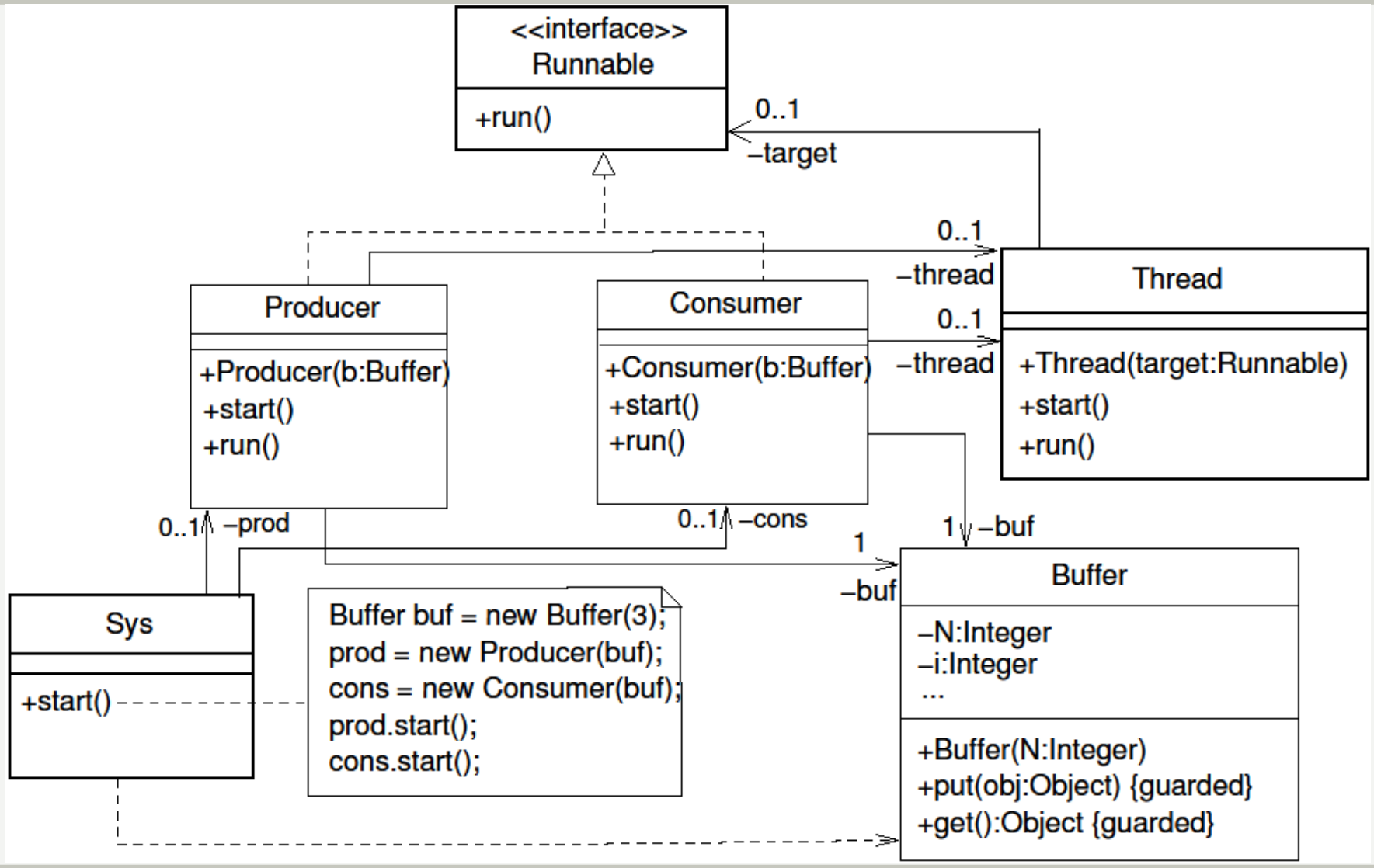
# Übungsblatt 8

## Aufgabe 2, Fortsetzung

Die folgende Klasse (nächste Folie) simuliert das Verhalten von Semaphoren in Java.

Geben Sie eine Java-Implementierung des Erzeuger-Verbraucher-Systems von Übungsblatt 7 an, so dass die Prozesskoordination über zwei Semaphoren *freePlaces* und *occupiedPlaces* realisiert wird und dafür die Klasse *Buffer* auf *wait* und *notify* Aufrufe verzichtet. Geben Sie auch ein FSP-Modell für Ihre Implementierung an und analysieren Sie das Deadlock-Verhalten.

# Modell aus Übungsblatt 7



# Aufgabe

## Die zwei Semaphoren

- **sollen nicht** den wechselseitigen Ausschluss beim Aufruf der Methoden `put` und `get` garantieren (=maximal ein Prozess greift gleichzeitig auf Puffer zu),
- **sollen** zur Prozesskoordination dienen, d.h.
  - `put` ist immer möglich, außer wenn es keine freien Plätze mehr gibt,
  - `get` ist immer möglich, außer wenn es nur freie Plätze gibt.

## Warum sind zwei Semaphoren nötig?

- Wdh: Eine Semaphore stellt nur sicher, dass ihr Wert nur bis maximal 0 erniedrigt werden kann; die Erhöhung wird nicht kontrolliert.
- Wir müssen hier aber die Anzahl der Plätze sowohl für **die obere als auch die untere** Grenze kontrollieren.

# Aufgabe

## Realisierung des Puffers durch **zwei Semaphoren**:

1) Die Anzahl "freier Plätze" im Puffer: `freePlaces`
2) Die Anzahl "belegter Plätze" im Puffer: `occupiedPlaces`

## Im Puffer:

- "`put` ist immer möglich, außer wenn es keine freien Plätze mehr gibt."
  - warten, solange `freePlaces == 0`
  - `freePlaces` erniedrigen, `occupiedPlaces` erhöhen
- "`get` ist immer möglich, außer wenn es nur freie Plätze gibt."

  => "`get` ist immer möglich, außer wenn es keine belegten Plätze gibt."
  - warten, solange `occupiedPlaces == 0`
  - `occupiedPlaces` erniedrigen, `freePlaces` erhöhen

LÖSUNG

# Implementierung von Buffer

## Klasse `Buffer`

- Attribute u.a.:
  - Größe `N`
  - Semaphore `freePlaces` initialisiert mit `N`,
  - Semaphore `occupiedPlaces` intialisiert mit `0`
- `synchronized void put(Object obj)`
  - erniedrige `freePlaces` (wartet intern, falls `freePlaces == 0`)
  - füge Element dem Puffer hinzu
  - erhöhe `occupiedPlaces`
- `synchronized Object get()`
  - erniedrige `occupiedPlaces` (wartet intern, falls `occupiedPlaces == 0`)
  - entferne Element aus dem Puffer
  - erhöhe `freePlaces`

# Probleme...

## Buffer

```
public synchronized void put(Object obj){
      freePlaces.down();
      buf[in] = obj;
      in = (in + 1) % N;
      occupiedPlaces.up();
}

public synchronized Object get() {
      occupiedPlaces.down();
      Object obj = buf[out];
      buf[out] = "_";
      out = (out + 1) % N;
      freePlaces.up();
      return obj;
}
```

## Semaphore

```
public synchronized void up(){
    value++;
    notify();
}

public synchronized void down(){
    while (value == 0)
        wait();
    value--;
}
```

**“Nested Monitor Problem”**

# Nested Monitor Problem

Beispielszenario: `get` wird vor dem ersten Aufruf von `put` aufgerufen.

- Der Aufrufer von `get` erhält zunächst das LOCK auf den Buffer.
- Der Aufrufer von `get` ruft die Methode `down` der Semaphore `occupiedPlaces` auf und erhält das LOCK auf die Semaphore.
- Die Semaphore `occupiedPlaces` hat den Wert 0 und muss warten.
- `wait()` gibt aber nur das LOCK des aktuellen Objekts frei, d.h. das LOCK der Semaphore `occupiedPlaces`.
- Demzufolger behält der Aufrufer von `get` das LOCK auf den Buffer, während er auf neue Einträge wartet.

=> Ein Aufrufer von `put` kann das LOCK auf den Buffer nie erhalten.

**Nested Monitor Problem**

# FSP-Modell

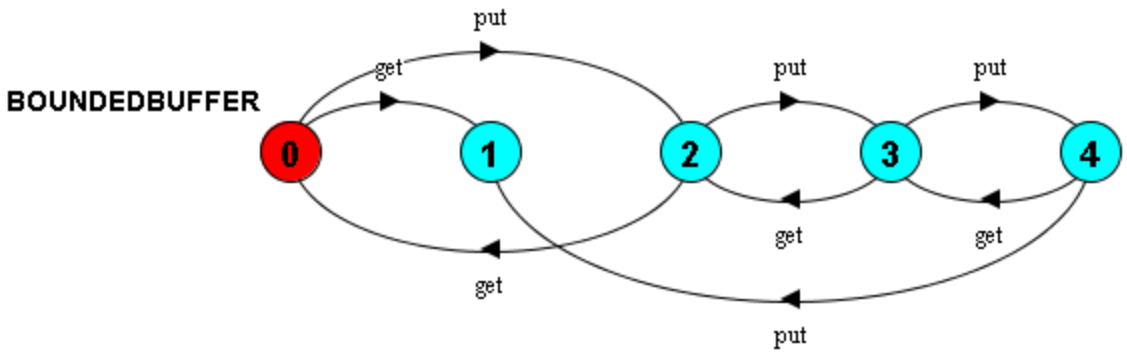
```
const MAX= 3
range Int = 0..MAX

SEMAPHORE(Initial=3) = SEMA[Initial],
SEMA[v:Int]              = ( when (v>0) down -> SEMA[v-1]
                           | up -> SEMA[v+1]),
SEMA[MAX+1]              = ERROR.

BUFFER(N=3) = COUNT[0],
COUNT[i:0..N] = ( put -> freePlaces.down -> occupiedPlaces.up -> COUNT[i+1]
                | get -> occupiedPlaces.down -> freePlaces.up -> COUNT[i-1]).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER || BUFFER(3) || CONSUMER
                    || freePlaces: SEMAPHORE(MAX)
                    || occupiedPlaces : SEMAPHORE(0)) @ {put,get}.
```

# Analyse

```
Composition:
BOUNDEDBUFFER = PRODUCER || BUFFER(3) || CONSUMER || freePlaces:SEMAPHORE(3)
    occupiedPlaces:SEMAPHORE(0)
State Space:
 1 * 1 * 28 * 4 * 4 = 2 ** 9
Composing...
  potential DEADLOCK
-- States: 24 Transitions: 26 Memory used: 7706K
Composed in 54ms
```

**Trace to DEADLOCK:**

**get**



# Problemlösung

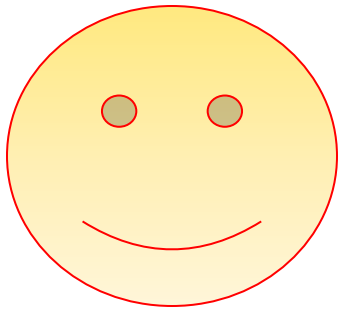
- Nested Monitors machen Probleme
- Lösung: Locks **nicht** gleichzeitig halten!
  - Stattdessen sequentiell vorgehen
  - Ein Lock nach dem Anderen anfordern
- Im Beispiel: put/get von Buffer:
  - Zunächst Lock der Semaphore verwenden und wieder freigeben
  - Danach Lock des Buffers verwenden und wieder freigeben
  - Zuletzt wieder Lock der (zweiten) Semaphore verwenden.
- D.h. Locks der Semaphore werden außerhalb des Buffer-Bereichs verwendet

LÖSUNG

# Lösung in Java

## Buffer

```
public void put(Object obj) {
        freePlaces.down();
        synchronized(this) {
                buf[in] = obj;
                in = (in + 1) % N;
        }
        occupiedPlaces.up();
}

public Object get() {
        occupiedPlaces.down();
        Object obj = null;
        synchronized(this) {
                obj = buf[out];
                buf[out] = "_";
                out = (out + 1) % N;
        }
        freePlaces.up();
        return obj;
}
```



# Lösung in FSP

```
const MAX = 3
range Int = 0..MAX
SEMAPHORE(N=0) = …

LOCK = (acquire -> release -> LOCK).

PUT = ( put -> freePlaces.down ->
        put.acquire -> putting -> put.release ->
        occupiedPlaces.up -> PUT).
GET = ( get -> occupiedPlaces.down ->
        get.acquire -> getting -> get.release ->
        freePlaces.up -> GET).
||BUFFER = (PUT||GET||{put,get}::LOCK).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = …
```