

## Programmierprimitiven: Prozessmodell und Semaphore

Prof. Dr. David Sabel

LFE Theoretische Informatik



- 1 Einleitung
- 2 Erweitertes Prozessmodell
- 3 Semaphore
  - Definition
  - Mutual-Exclusion mithilfe von Semaphore
- 4 Semaphore in Java
- 5 Anwendungsbeispiele

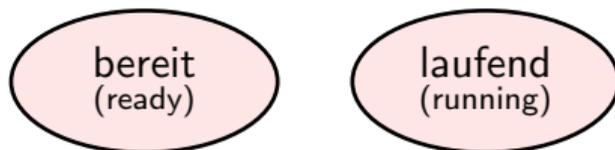
- Primitive atomare Operationen, die durch Hardware implementiert sind
- Lösungen für das Mutual-Exclusion-Problem mit den verschiedenen Primitiven

- „Softwarelösungen“
- Primitive, die durch nebenläufige Programmiersprachen bereit gestellt werden
- Anwendungen (klassische Probleme)

# Prozessmodell (1)

---

Prozesse  $P$  haben einen Zustand  $P.state$ :

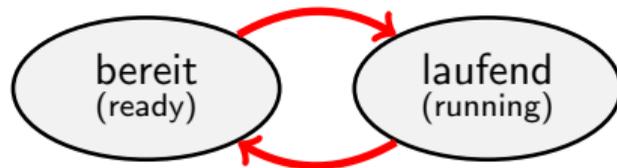


- **laufend/running**: Prozess führt Schritte aus
- **bereit/ready**: Prozess will Schritte ausführen, darf aber nicht
- mindestens ein Prozess läuft immer (z.B. **Leerlaufprozess**)

# Prozessmodell (1)

---

Prozesse  $P$  haben einen Zustand  $P.state$ :

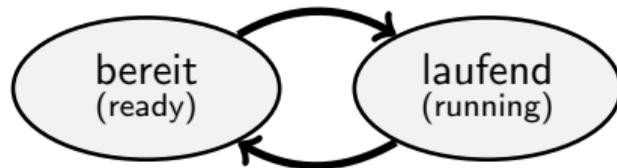


- **laufend/running**: Prozess führt Schritte aus
- **bereit/ready**: Prozess will Schritte ausführen, darf aber nicht
- mindestens ein Prozess läuft immer (z.B. **Leerlaufprozess**)
- **Scheduler** führt **Context-Switch** aus:  
Bereite Prozesse werden zu laufenden und umgekehrt

# Prozessmodell (1)

---

Prozesse  $P$  haben einen Zustand  $P.state$ :

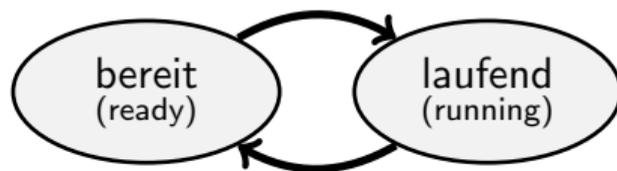


- **laufend/running**: Prozess führt Schritte aus
- **bereit/ready**: Prozess will Schritte ausführen, darf aber nicht
- mindestens ein Prozess läuft immer (z.B. **Leerlaufprozess**)
- **Scheduler** führt **Context-Switch** aus:  
Bereite Prozesse werden zu laufenden und umgekehrt
- **Fairness**: Jeder bereite Prozess wird nach endlich vielen Schritten laufend

## Prozessmodell (2)

---

Prozesse  $P$  haben einen Zustand  $P.state$ :



## Prozessmodell (2)

---

Prozesse  $P$  haben einen Zustand  $P.state$ :

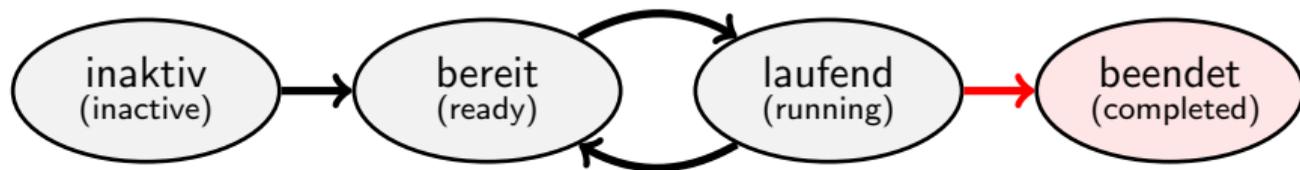


- **inaktiv**: noch nicht bereit (z.B. Code wird geladen)

## Prozessmodell (2)

---

Prozesse  $P$  haben einen Zustand  $P.state$ :

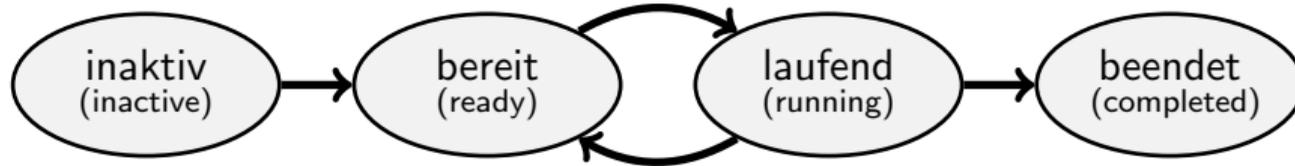


- **inaktiv**: noch nicht bereit (z.B. Code wird geladen)
- **beendet/completed**: Prozess terminiert

# Prozessmodell (3)

---

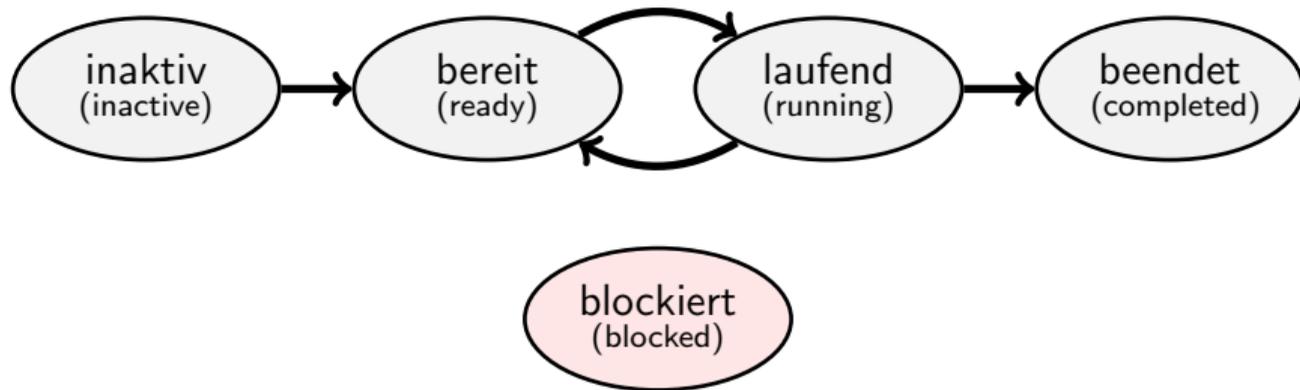
Prozesse  $P$  haben einen Zustand  $P.state$ :



## Prozessmodell (3)

---

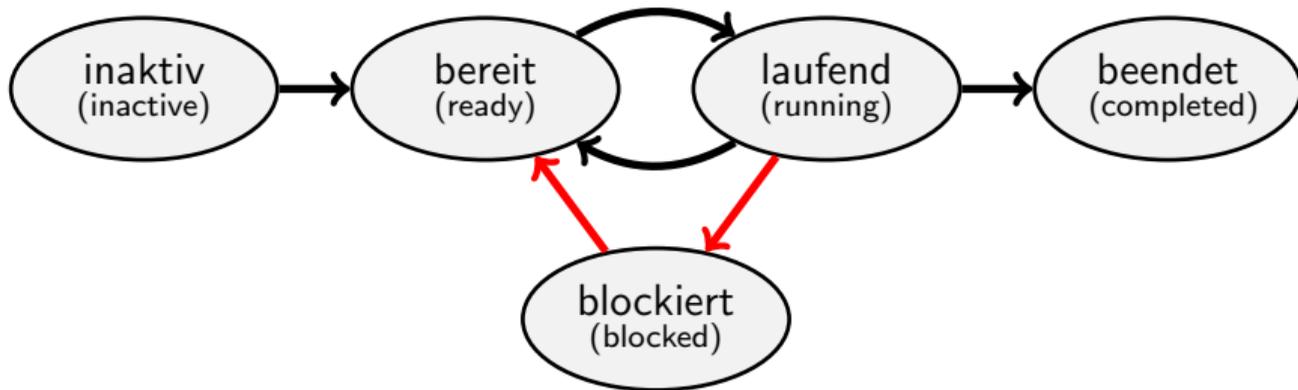
Prozesse  $P$  haben einen Zustand  $P.state$ :



- blockiert: Prozess darf keine Schritte ausführen

## Prozessmodell (3)

Prozesse  $P$  haben einen Zustand  $P.state$ :



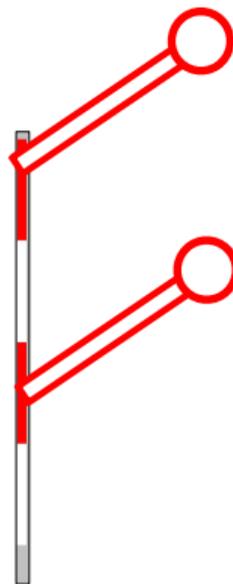
- blockiert: Prozess darf keine Schritte ausführen
- blockieren / entblockieren durch Programmbefehle, **nicht** durch Scheduler

# Semaphore

---

## **Begriffsherkunft:**

Semaphor =  
Mechanischer Signalgeber im Bahnverkehr



# Semaphore

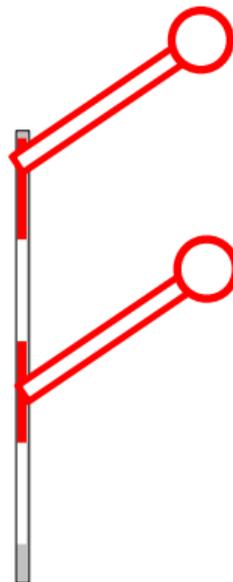
---

## **Begriffsherkunft:**

Semaphor =  
Mechanischer Signalgeber im Bahnverkehr

## **In der Informatik:**

Abstrakter Datentyp mit Operationen



# Semaphor $S$

---

## Attribute (i.a.):

- $V$  = Nicht-negative Ganzzahl
- $M$  = Menge von Prozessen

Schreibweise für Semaphor  $S$ :  $S.V$  und  $S.M$

## Attribute (i.a.):

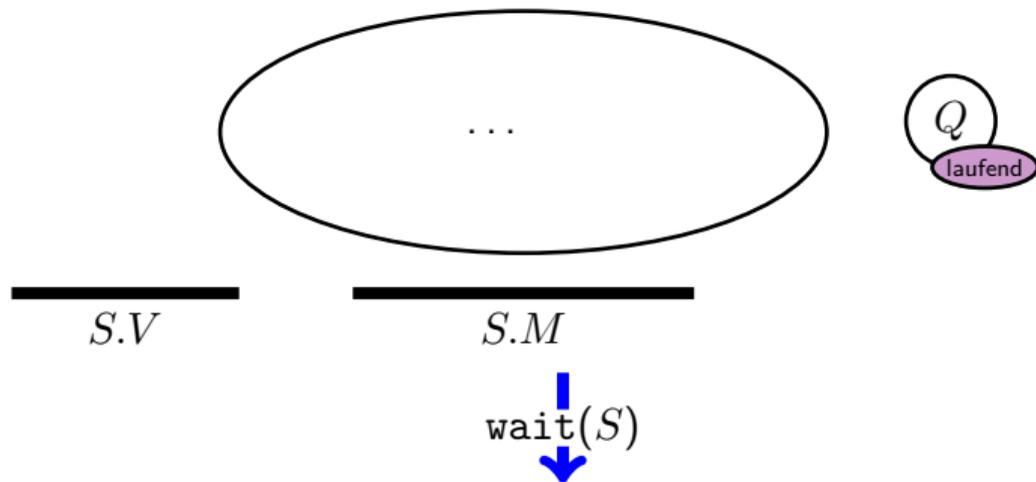
- $V$  = Nicht-negative Ganzzahl
- $M$  = Menge von Prozessen

Schreibweise für Semaphor  $S$ :  $S.V$  und  $S.M$

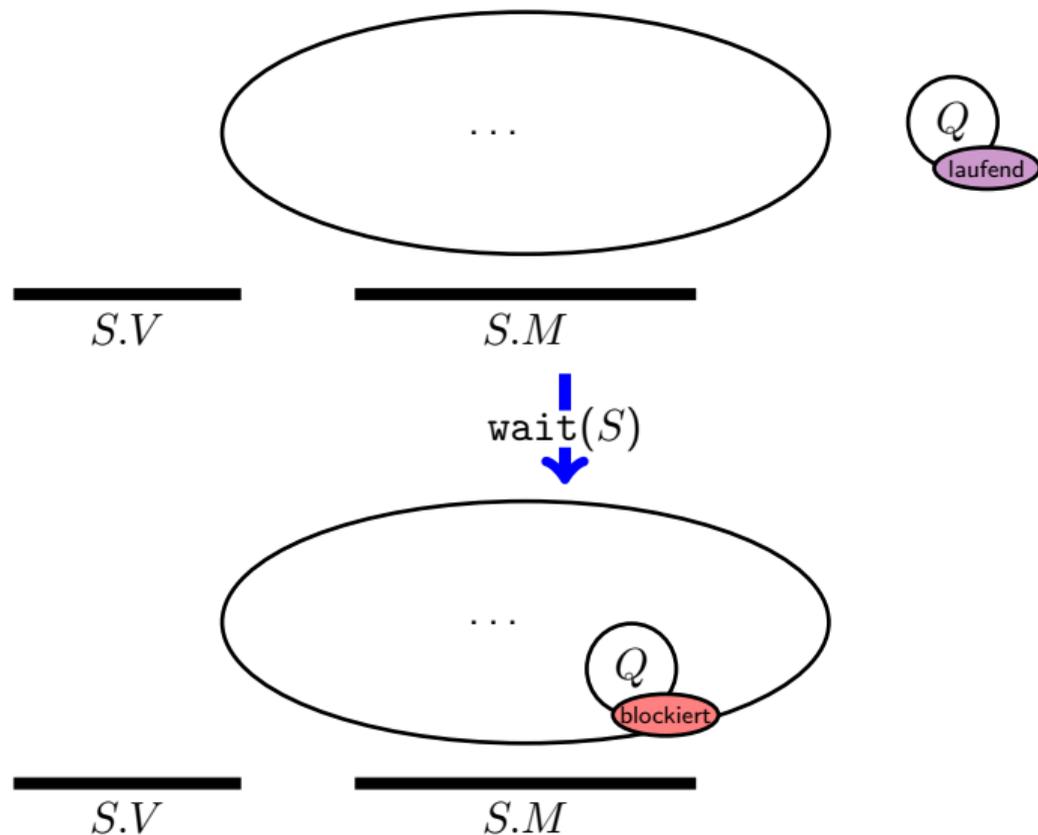
## Operationen:

- `newSem( $k$ )` erzeugt neuen Semaphor mit  $S.V = k$  und  $S.M = \emptyset$
- `wait( $S$ )` (alternativ: `P( $S$ )` (Dijkstra, prolaag (Kunstwort, anstelle von verlaag (niederl. erniedrige) oder `down( $S$ )`))
- `signal( $S$ )` (alternativ: `V( $S$ )` (Dijkstra, verhoog (niederl. erhöhe)) oder `up( $S$ )`)
- werden `atomic` ausgeführt (aus Sicht des Programmieres, d.h. Programmiersprache sorgt für „richtige“ Implementierung)

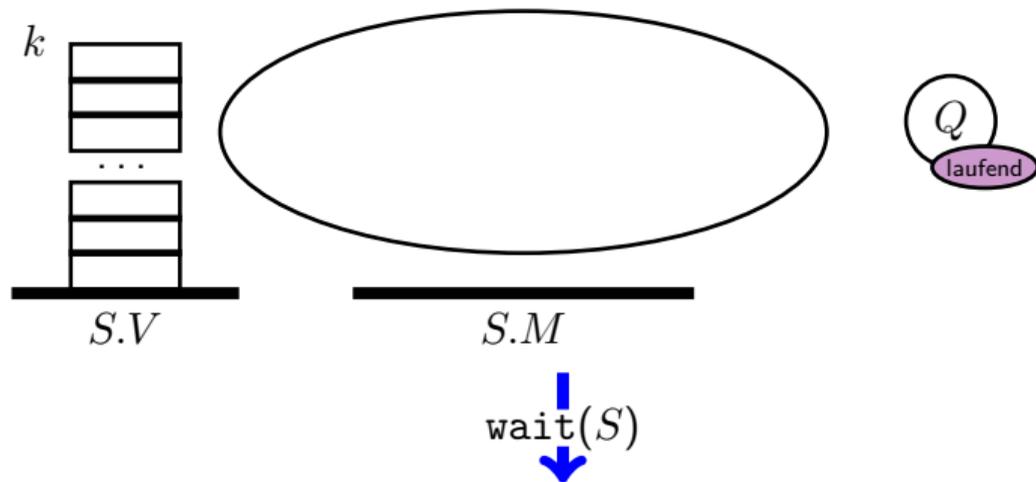
# Semaphor: Wait von Prozess $Q$ , wenn $S.V = 0$



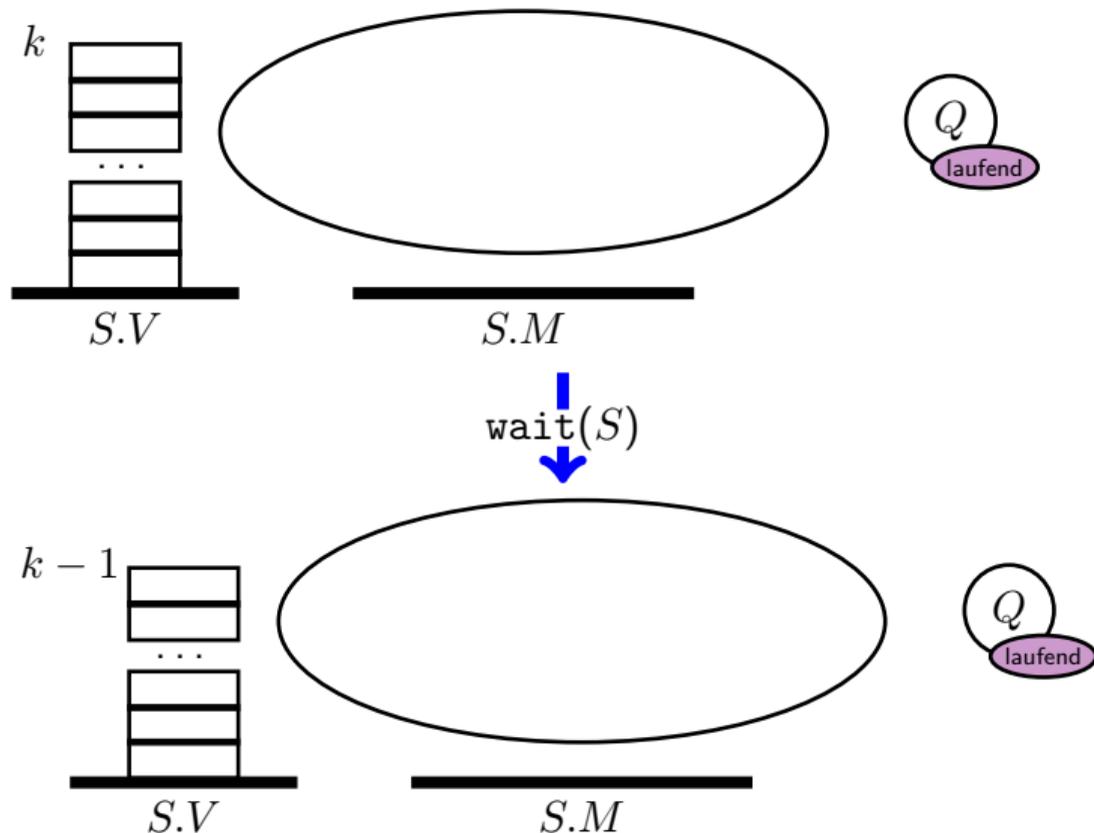
# Semaphor: Wait von Prozess $Q$ , wenn $S.V = 0$



# Semaphor: Wait von Prozess $Q$ , wenn $S.V > 0$



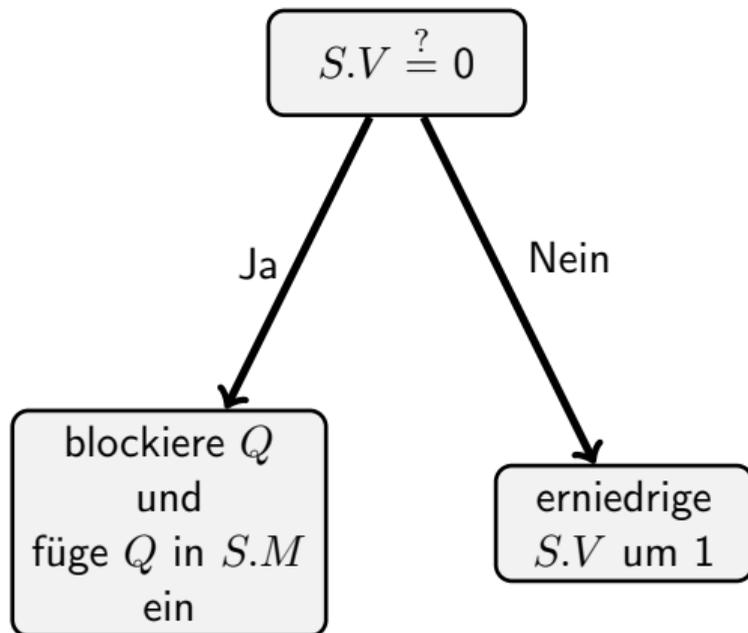
# Semaphor: Wait von Prozess $Q$ , wenn $S.V > 0$



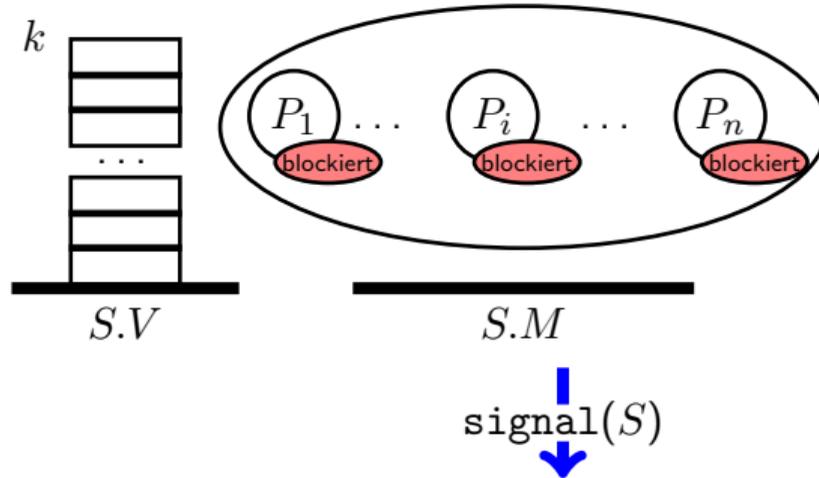
# wait( $S$ )

Sei  $Q$  der aufrufende Prozess:

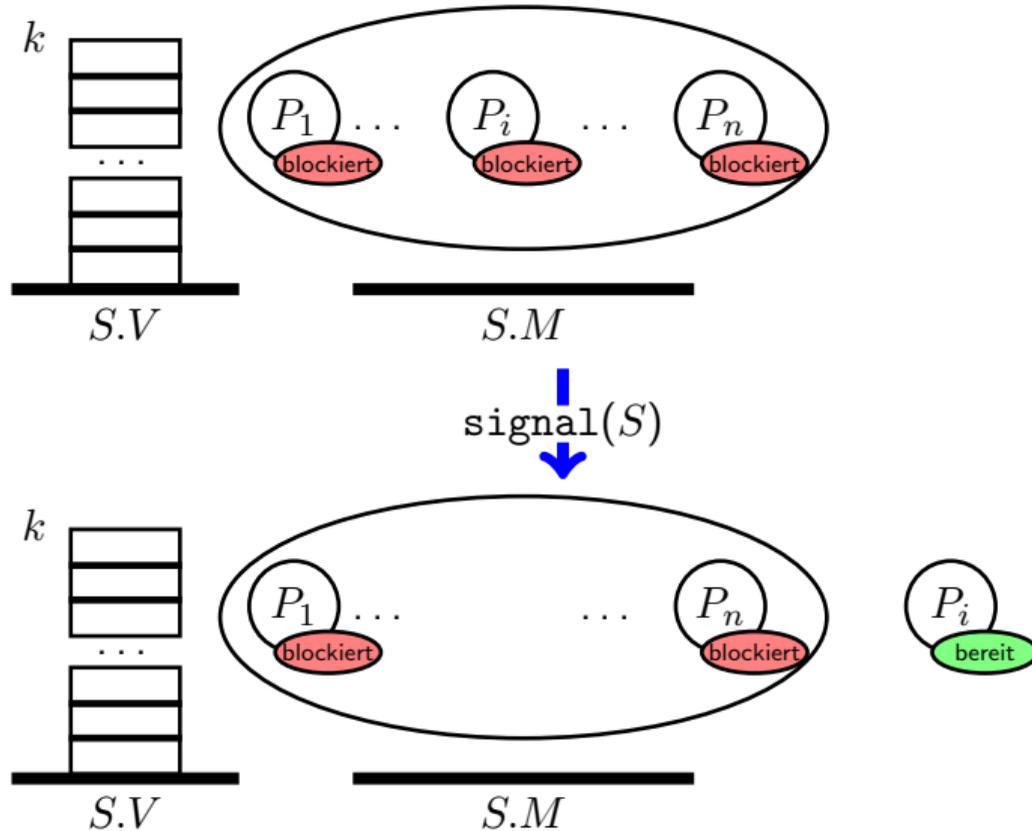
```
procedure wait( $S$ )  
  if  $S.V > 0$  then  
     $S.V := S.V - 1$ ;  
  else  
     $S.M := S.M \cup \{Q\}$ ;  
     $Q.state := blocked$ ;
```



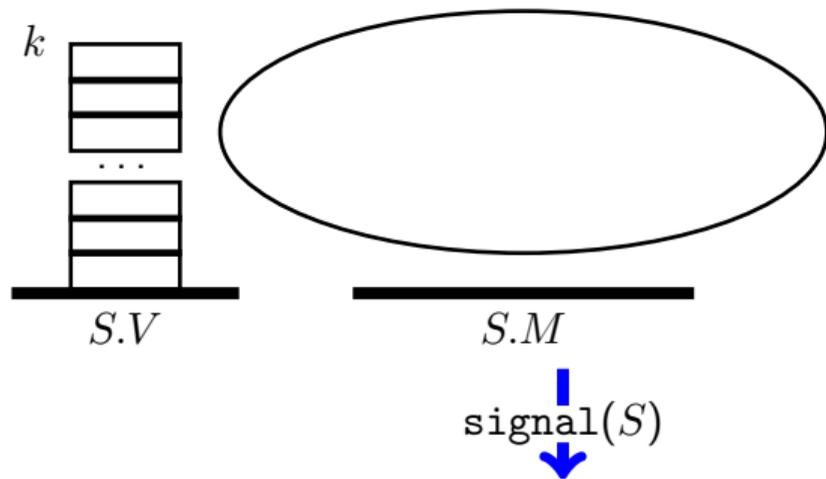
# Semaphor: Signal, wenn $S.M \neq \emptyset$



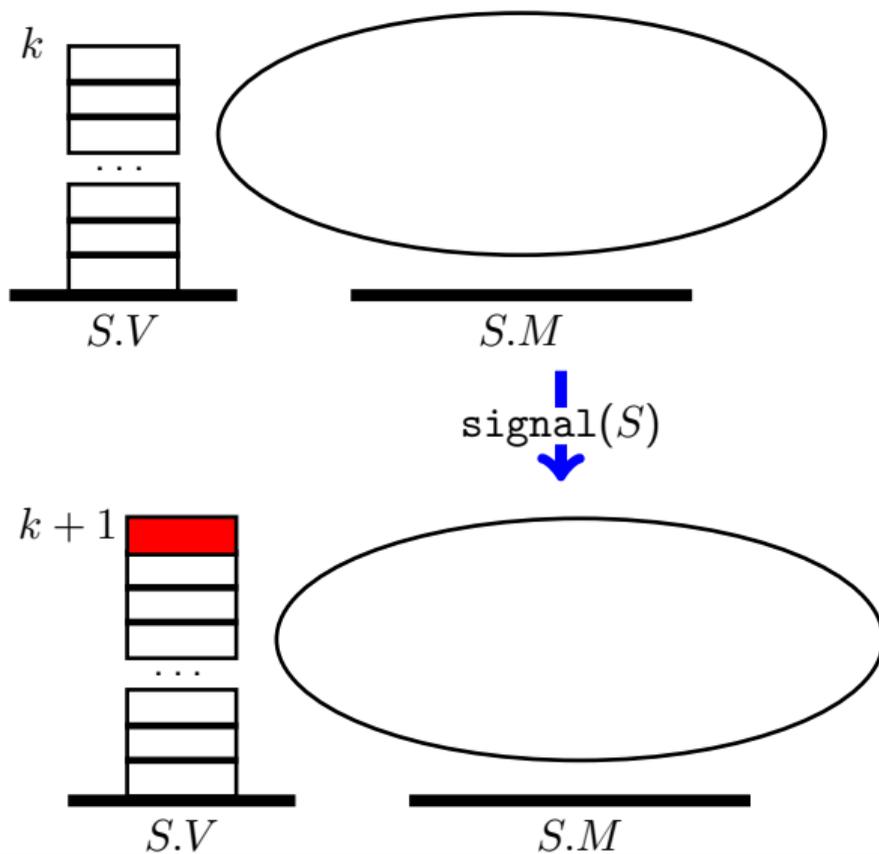
# Semaphor: Signal, wenn $S.M \neq \emptyset$



# Semaphor: Signal, wenn $S.M = \emptyset$

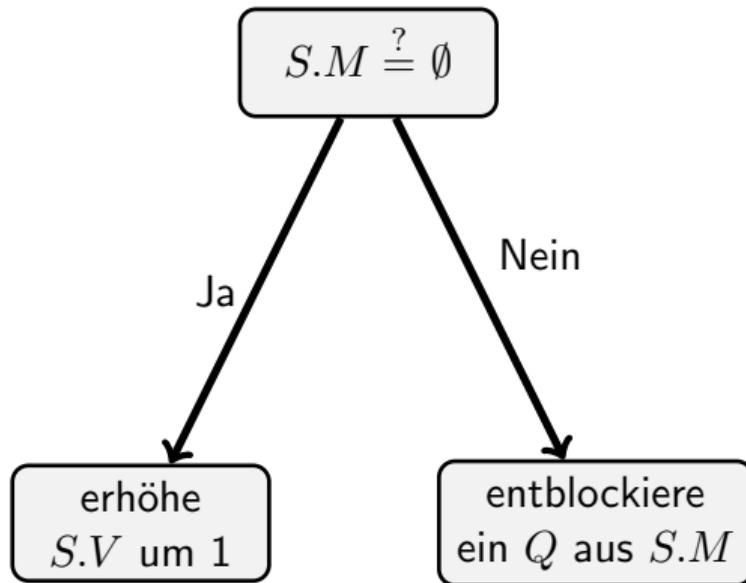


# Semaphor: Signal, wenn $S.M = \emptyset$



# signal(*S*)

```
procedure signal(S)  
  if  $S.M = \emptyset$  then  
     $S.V := S.V + 1$ ;  
  else  
    wähle ein Element  $Q$  aus  $S.M$ ;  
     $S.M := S.M \setminus \{Q\}$ ;  
     $Q.state := ready$ ;
```



Sei  $S$  mit  $k$  initialisierter Semaphor. Nach Ausführung jeder Auswertungsfolge  $\mathcal{P}$  gilt:

- $S.V \geq 0$
- $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P})$

wobei

- $\#_{signal}(S, \mathcal{P}) =$  Anzahl `signal`-Operationen in  $\mathcal{P}$
- $\#_{wait}(S, \mathcal{P}) =$  Anzahl `wait`-Operationen in  $\mathcal{P}$



# Invarianten

Sei  $S$  mit  $k$  initialisierter Semaphor. Nach Ausführung jeder Auswertungsfolge  $\mathcal{P}$  gilt:

- $S.V \geq 0$
- $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P})$   

-1		+1	wait(S)
	+1	+1	
+1		+1	signal(S)
	-1	+1	

wobei

- $\#_{signal}(S, \mathcal{P}) =$  Anzahl signal-Operationen in  $\mathcal{P}$
- $\#_{wait}(S, \mathcal{P}) =$  Anzahl wait-Operationen in  $\mathcal{P}$

# Binäre Semaphore

---

- bisher: **Generelle** Semaphore
- bei **binären Semaphore**:
  - $0 \leq S.V \leq 1$
  - wait unverändert
  - signal darf nicht beliebig erhöhen:

```
procedure signal(S)
  if S.V = 1 then
    undefined
  else if S.M =  $\emptyset$  then
    S.V := 1
  else
    wähle ein Element Q aus S.M;
    S.M := S.M \ {Q};
    Q.state := ready;
```

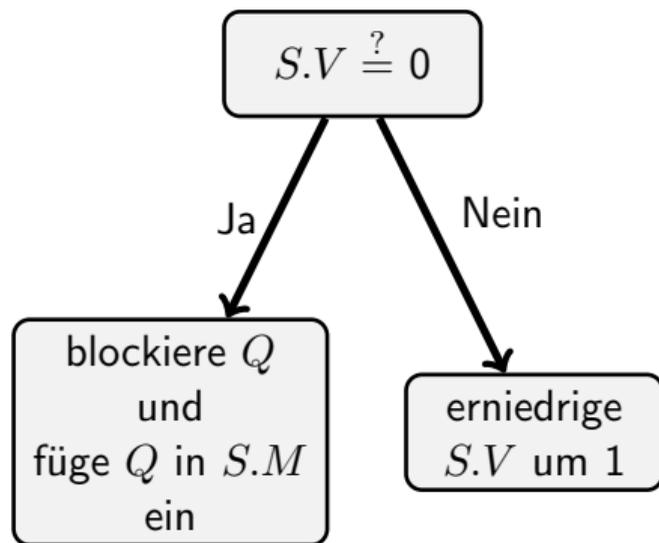
## Binäre Semaphore (2)

---

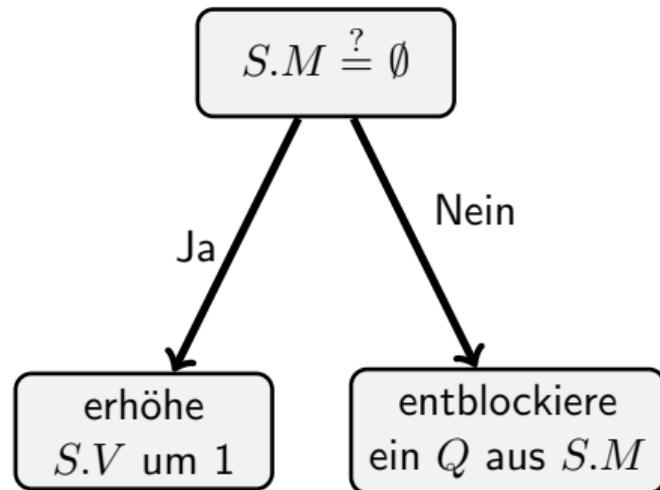
- werden in Programmiersprachen oft als `mutex` bezeichnet
- Invarianten gelten weiterhin, `wenn` vor jedem `signal` ein zugehöriges `wait` ausgeführt wird

## Zusammenfassend nochmal: Semaphore

$Q$  ruft `wait(S)` auf



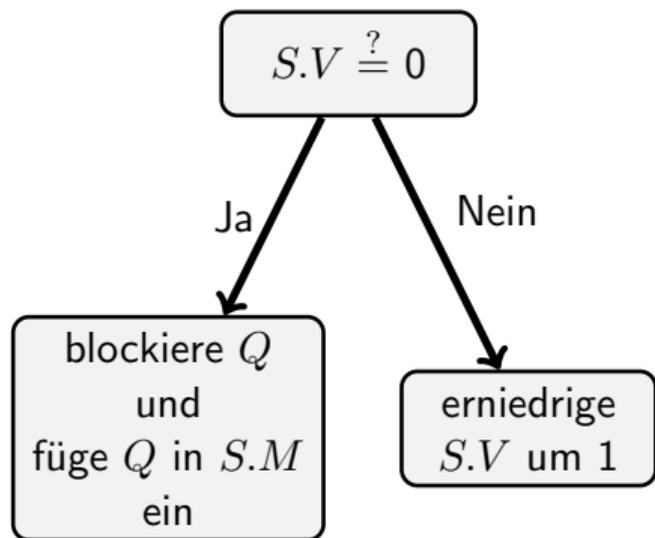
`signal(S)`



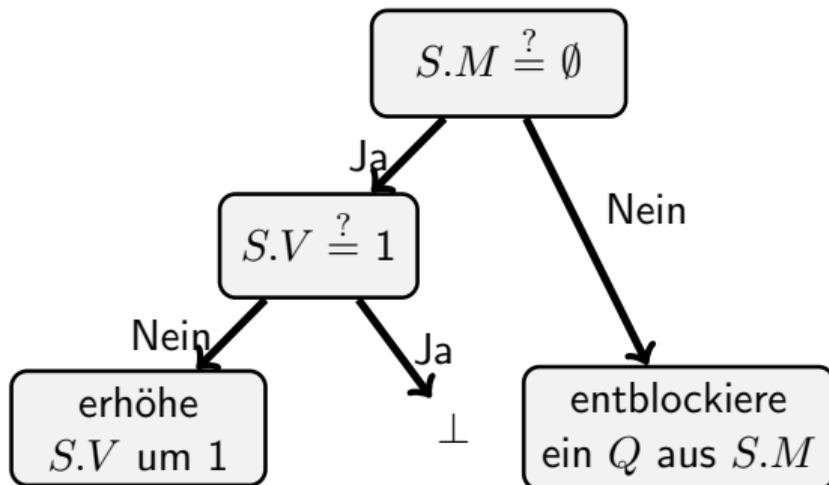
# Zusammenfassend: Binärer Semaphor

$V$  nur 0 oder 1

$Q$  ruft `wait(S)` auf



`signal(S)`



# Mutual-Exclusion mit binärem Semaphor

---

Initial:  $S$  sei ein binärer Semaphor, initialisiert mit 1

Programm des  $i$ . Prozesses

```
loop forever
(P1)  restlicher Code
(P2)  wait(S)
(P3)  Kritischer Abschnitt
(P4)  signal(S)
end loop
```

## Theorem

Der Algorithmus garantiert wechselseitigen Ausschluss und ist Deadlock-frei.

Beweis: Mutual-Exclusion

- $\#_{KA}(\mathcal{P}) =$  Anzahl von Prozessen im Kritischen Abschnitt nach Ausführung von  $\mathcal{P}$
- $\#_{KA}(\mathcal{P}) = \#_{wait}(S, \mathcal{P}) - \#_{signal}(S, \mathcal{P}) - |S.M|$
- Mit Invariante  $S.V = k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P})$  ergibt das

$$\begin{aligned}\#_{KA}(\mathcal{P}) + S.V &= \#_{wait}(S, \mathcal{P}) - \#_{signal}(S, \mathcal{P}) - |S.M| \\ &\quad + k + |S.M| + \#_{signal}(S, \mathcal{P}) - \#_{wait}(S, \mathcal{P}) \\ &= k\end{aligned}$$

- Da  $k = 1$  muss gelten:  $\#_{KA}(\mathcal{P}) \leq 1$  (d.h. wechselseitiger Ausschluss)

### Theorem

Der Algorithmus garantiert wechselseitigen Ausschluss und ist Deadlock-frei.

Beweis: Deadlock-Freiheit

- $\mathcal{P}$  = unendlich lange Auswertungsfolge, so dass Deadlock auftritt
- $\mathcal{P}_1$  = Präfix von  $\mathcal{P}$ , so dass ab  $\mathcal{P}_1$ :
  - Kein Prozess im Kritischen Abschnitt, d.h.  $\#_{KA}(\mathcal{P}_1) = 0$
  - Mind. ein Prozess  $P$  wartet (ist blockiert), d.h.  $S.V = 0$  und  $P \in S.M$
- Unmöglich, da  $\#_{KA}(\mathcal{P}_1) + S.V = 1$

## Weitere Eigenschaften

---

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	<code>wait(S)</code>	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	<code>wait(S)</code>	rest. Code
0	$\{2\}$	Krit. Abschnitt	(blockiert)	<code>wait(S)</code>

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)
0	{3}	rest. Code	Krit. Abschnitt	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)
0	{3}	rest. Code	Krit. Abschnitt	(blockiert)
0	{1, 3}	wait( $S$ )	Krit. Abschnitt	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)
0	{3}	rest. Code	Krit. Abschnitt	(blockiert)
0	{1, 3}	wait( $S$ )	Krit. Abschnitt	(blockiert)
0	{1, 3}	(blockiert)	signal( $S$ )	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)
0	{3}	rest. Code	Krit. Abschnitt	(blockiert)
0	{1, 3}	wait( $S$ )	Krit. Abschnitt	(blockiert)
0	{1, 3}	(blockiert)	signal( $S$ )	(blockiert)
0	{3}	Krit. Abschnitt	rest. Code	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)
0	{3}	rest. Code	Krit. Abschnitt	(blockiert)
0	{1, 3}	wait( $S$ )	Krit. Abschnitt	(blockiert)
0	{1, 3}	(blockiert)	signal( $S$ )	(blockiert)
0	{3}	Krit. Abschnitt	rest. Code	(blockiert)
0	{3}	Krit. Abschnitt	wait( $S$ )	(blockiert)

## Weitere Eigenschaften

- Bei maximal 2 Prozessen: Algorithmus ist Starvation-frei.
- Bei mehr Prozessen: **nicht Starvation-frei:**

$S.V$	$S.M$	Prozess 1	Prozess 2	Prozess 3
1	$\emptyset$	rest. Code	rest. Code	rest. Code
1	$\emptyset$	wait( $S$ )	rest. Code	rest. Code
0	$\emptyset$	Krit. Abschnitt	wait( $S$ )	rest. Code
0	{2}	Krit. Abschnitt	(blockiert)	wait( $S$ )
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)
0	{2, 3}	signal( $S$ )	(blockiert)	(blockiert)
0	{3}	rest. Code	Krit. Abschnitt	(blockiert)
0	{1, 3}	wait( $S$ )	Krit. Abschnitt	(blockiert)
0	{1, 3}	(blockiert)	signal( $S$ )	(blockiert)
0	{3}	Krit. Abschnitt	rest. Code	(blockiert)
0	{3}	Krit. Abschnitt	wait( $S$ )	(blockiert)
0	{2, 3}	Krit. Abschnitt	(blockiert)	(blockiert)

## Weitere Arten von Semaphore

- Bisherige Semaphore sind **schwache Semaphore**, da Auswahl des zu entblockierenden Prozesses beliebig.
- **Starke Semaphore**: FIFO-Reihenfolge

Queue / Liste  $S.L$  statt Menge  $S.M$

```
procedure wait( $S$ )
  if  $S.V > 0$  then
     $S.V := S.V - 1$ ;
  else
     $S.L := append(S.L, P)$ ;
     $P.state := blocked$ ;
```

```
procedure signal( $S$ )
  if isEmpty( $S.L$ ) then
     $S.V := S.V + 1$ ;
  else
     $Q := head(S.L)$ ;
     $S.L := tail(S.L)$ ;
     $Q.state := ready$ ;
```

- Mit starkem Semaphor: Algorithmus ist Starvation-frei & erfüllt FIFO-Eigenschaft

## Weitere Arten von Semaphore (2)

---

- **Unfaire** bzw. **Busy-Wait** Semaphore
- gar keine Eigenschaft, wann ein Prozess entblockiert wird
- blockiert = busy-waiting

Keine  $S.M$  Komponente nur  $S.V$

```
procedure wait( $S$ )  
  await  $S.V > 0$ ;  
   $S.V := S.V - 1$ ;
```

```
procedure signal( $S$ )  
   $S.V := S.V + 1$ ;
```

- Algorithmus selbst bei 2 Prozessen nicht Starvation-frei.

# Semaphore in Java

---

- Im Package `java.util.concurrent` ist die Klasse `Semaphore` definiert.
- Konstruktor `Semaphore(i)` initialisiert den Semaphor mit Wert  $i$
- Negatives  $i$  erlaubt
- `wait` heißt `acquire`
- `signal` heißt `release`
- Exceptions können auftreten und müssen abgefangen werden (bei `acquire` `InterruptedException`)
- zweiter Konstruktor `Semaphore(i,fair)`
  - $i$  = initialer Wert
  - `fair` = Boolescher Wert. Wenn falsch, dann busy-wait Semaphor, sonst starker Semaphor

## Beispiel aus Ben-Ari Buch

---

```
import java.util.concurrent.Semaphore;
class CountSem extends Thread {
    static volatile int n = 0; // globale atomare Variable
    static Semaphore s = new Semaphore(1);

    public void run() {
        int temp;
        for (int i = 0; i < 10; i++) {
            try {
                s.acquire();
            }
            catch (InterruptedException e) {}
            temp = n;
            n = temp + 1;
            s.release();
        }
    }
    ...
}
```

## Beispiel aus Ben-Ari Buch (2)

---

```
...
public static void main(String[] args) {
    CountSem p = new CountSem();
    CountSem q = new CountSem();
    p.start(); // startet Thread p
    q.start(); // startet Thread q
    try {
        p.join();// wartet auf Terminierung von Thread p
        q.join();// wartet auf Terminierung von Thread q
    }
    catch (InterruptedException e) { }
    System.out.println("The value of n is " + n);
}
}
```

Im Folgenden:

# Anwendungsbeispiele und Problemlösungen mit Semaphore

Koordination - Beispiel: Merge-Sort

Erzeuger / Verbraucher: Infinite / bounded Buffer

Speisende Philosophen

The Sleeping Barber

Cigarette Smoker's Problem

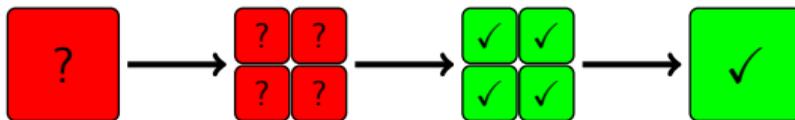
Reader & Writers

# Fork-Join-Pattern zum Parallelisieren

---

Idee analog zu Divide-and-Conquer:

- Löse Teilprobleme mit Threads
- Warte auf Beenden der Threads
- Setze Gesamtlösung zusammen



# Mergesort: Koordination der Reihenfolge

---

Parallelisierung von Mergesort mit Fork-Join-Pattern:

- Teile Eingabe in zwei Hälften
- Sortiere beide Hälften (rekursiv) nebenläufig (fork)
- Mische anschließend das Ergebnis (join)

Problem: Mische erst **nachdem** die beiden Hälften fertig sortiert sind.

# Mergesort mit binären Semaphore

---

Initial: left, right: Binärer Semaphore mit 0 initialisiert

merge-Prozess:

- (1) `wait(left);`
- (2) `wait(right);`
- (3) `merge`

Prozess für linke Hälfte

- (1) `sortiere linke Hälfte;`
- (2) `signal(left);`

Prozess für rechte Hälfte

- (1) `sortiere rechte Hälfte;`
- (2) `signal(right);`

Achtung: 2 Semaphore pro Rekursionsschritt!

Probleme:

- Naive Parallelisierung führt zu **Slow-Down**
- Zu viele Threads für wenige Prozessorkerne

Lösungen:

- Wenige Threads erzeugen
- Anzahl abschätzen bzw. testen
- Noch besser: Thread-Pool verwenden  
(Anzahl der Threads begrenzen, aber Threads können mehrere Aufgaben hintereinander bearbeiten)

Problem:

- Erzeuger produzieren Daten
- Verbraucher konsumieren Daten
- Beispiel: Tastatur / Betriebssystem usw.

Austausch über Puffer:

- Queue / Liste
- Erzeuger schreibt hinten auf die Liste
- Verbraucher konsumiert vorne von der Liste
- Zwei Varianten:
  - beliebig lange Liste (infinite buffer)
  - begrenzter Platz (bounded buffer)

# Erzeuger / Verbraucher mit infinite Buffer

---

Anforderungen:

- Lesen / Schreiben auf den Puffer **korrekt** (atomar)
- Verbraucher braucht Schutz für den Fall, dass der Puffer leer ist

## Erzeuger / Verbraucher mit infinite Buffer (2)

---

Initial: notEmpty: Genereller Semaphor, initialisiert mit 0  
mutex: Binärer Semaphor, initialisiert mit 1  
l: Liste

Erzeuger (erzeugt e)

- (1) erzeuge e;
- (2) wait(mutex);
- (3) l := append(l,e);
- (4) signal(notEmpty);
- (5) signal(mutex);

Verbraucher (verbraucht e)

- (1) wait(notEmpty);
- (2) wait(mutex);
- (3) e := head(l);
- (4) l := tail(l);
- (5) signal(mutex);
- (6) verbrauche e;

Liste am Anfang leer  $\implies$  Invariante:  $\text{notEmpty}.V = \text{length}(l)$

# Infinite Buffer mit Semaphore in Java

---

```
import java.util.concurrent.Semaphore;
import java.util.LinkedList;

class InfBuffer<V> {
    Semaphore notEmpty    = new Semaphore(0);
    Semaphore mutex       = new Semaphore(1);
    LinkedList<V> buffer = new LinkedList<V>();

    public void produce(V elem) {
        try {mutex.acquire();} catch (InterruptedException e) {};
        buffer.add(elem);
        notEmpty.release();
        mutex.release();
    }
    public V consume() {
        try {notEmpty.acquire();} catch (InterruptedException e) {};
        try {mutex.acquire();} catch (InterruptedException e) {};
        V e = buffer.removeFirst();
        mutex.release();
        return e;
    }
}
```

## Infinite Buffer mit Semaphore in Java (2)

---

```
class Producer extends Thread {
    static Random generator = new Random();
    InfBuffer<Integer> buff;
    Integer number;

    Producer(InfBuffer<Integer> b, Integer i) {
        buff = b;
        number = i;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {(Thread.currentThread()).sleep(Math.abs(generator.nextInt()%1000));}
            catch (InterruptedException e) { };
            buff.produce(i);
            PrintSem.print("Producer " + number + ": " + i + " produziert");
        }
    }
}
```

## Infinite Buffer mit Semaphore in Java (3)

---

```
class Consumer extends Thread {
    static Random generator = new Random();
    InfBuffer<Integer> buff;
    Integer number;

    Consumer(InfBuffer<Integer> b,Integer i) {
        buff = b;
        number = i;
    }

    public void run() {
        for (int i = 1; i <= 50; i++) {
            try {Thread.currentThread().sleep(Math.abs(generator.nextInt()%1000));}
            catch (InterruptedException e) { };
            Integer e = buff.consume();
            PrintSem.print("Consumer " + number + ": " + e + " konsumiert");
        }
    }
}
```

# Infinite Buffer mit Semaphore in Java (4)

---

```
final class PrintSem {
    static Semaphore mutex = new Semaphore(1);
    static void print(String str) {
        try {mutex.acquire();} catch (InterruptedException e) {};
        System.out.println(str);
        mutex.release();
    }
}

class Main {
    public static void main(String[] args) {
        InfBuffer<Integer> b = new InfBuffer<Integer>();
        for (int i=1; i <= 50; i++) {
            Producer q = new Producer(b,i);
            q.start();
        }
        for (int i=1; i <= 10; i++) {
            Consumer q = new Consumer(b,i);
            q.start();
        }
        while (true) {} // Endlosschleife
    }
}
```

# Erzeuger / Verbraucher mit bounded Buffer

---

Anforderungen:

- Lesen / Schreiben auf den Puffer **sicher** (atomar)
- Verbraucher braucht Schutz für den Fall, dass der Puffer leer ist
- Erzeuger braucht Schutz für den Fall, dass der Puffer voll ist

## Erzeuger / Verbraucher mit bounded Buffer (2)

Initial: notEmpty: Genereller Semaphor, initialisiert mit 0  
notFull: Genereller Semaphor, initialisiert mit  $N$   
mutex: Binärer Semaphor, initialisiert mit 1  
l: Liste

Erzeuger (erzeugt e)

- (1) erzeuge e;
- (2) wait(notFull);
- (3) wait(mutex);
- (4)  $l := \text{append}(l, e)$ ;
- (5) signal(notEmpty);
- (6) signal(mutex);

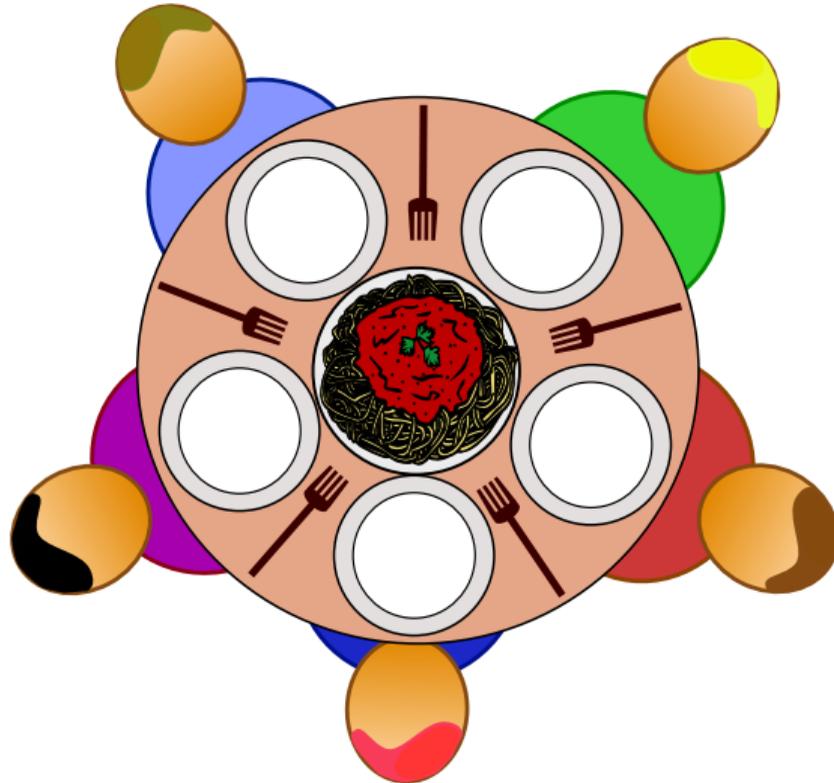
Verbraucher (verbraucht e)

- (1) wait(notEmpty);
- (2) wait(mutex);
- (3)  $e := \text{head}(l)$ ;
- (4)  $l := \text{tail}(l)$ ;
- (5) signal(notFull);
- (6) signal(mutex);
- (7) verbrauche e;

Invariante:  $\text{notEmpty}.V + \text{notFull}.V = N$   
„(notEmpty, notFull) = Split-Semaphor“

# Speisende Philosophen

---



## Speisende Philosophen (2)

---

### Situation

- Philosoph denkt oder isst Spaghetti, abwechselnd
- Philosoph braucht beide Gabeln zum Essen
- Philosoph nimmt Gabeln **nacheinander**

## Speisende Philosophen (2)

---

### Situation

- Philosoph denkt oder isst Spaghetti, abwechselnd
- Philosoph braucht beide Gabeln zum Essen
- Philosoph nimmt Gabeln **nacheinander**

### Anforderungen:

- Kein Deadlock: Irgendein Philosoph kann nach endlicher Zeit immer essen
- Kein Verhungern: Jeder Philosoph isst nach endlicher Zeit

## Speisende Philosophen (2)

---

### Situation

- Philosoph denkt oder isst Spaghetti, abwechselnd
- Philosoph braucht beide Gabeln zum Essen
- Philosoph nimmt Gabeln **nacheinander**

### Anforderungen:

- Kein Deadlock: Irgendein Philosoph kann nach endlicher Zeit immer essen
- Kein Verhungern: Jeder Philosoph isst nach endlicher Zeit

### Modellierung:

- Philosophen durchnummeriert  $i \in \{1, \dots, N\}$
- Gabel = Binärer Semaphor
- linke Gabel: `gabel[i]`, rechte Gabel: `gabel[i+1]` (modulo  $N$ )

Initial alle Gabeln mit 1 initialisiert

```
Philosoph i
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i]); // linke Gabel
(3) wait(gabel[i+1]); // rechte Gabel
(4) Philosoph isst
(5) signal(gabel[i + 1]);
(6) signal(gabel[i]);
end loop
```

# Beispiel

---

Philosoph 1

Philosoph 2

Philosoph 3

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code>		

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code>	

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code> „hat linke Gabel“	<code>wait(gabeln[3])</code>

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code> „hat linke Gabel“	<code>wait(gabeln[3])</code> „hat linke Gabel“

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code> „hat linke Gabel“	<code>wait(gabeln[3])</code> „hat linke Gabel“
<code>wait(gabeln[2])</code>		

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code> „hat linke Gabel“	<code>wait(gabeln[3])</code> „hat linke Gabel“
<code>wait(gabeln[2])</code> blockiert	<code>wait(gabeln[3])</code>	

# Beispiel

---

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code> „hat linke Gabel“	<code>wait(gabeln[3])</code> „hat linke Gabel“
<code>wait(gabeln[2])</code> blockiert	<code>wait(gabeln[3])</code> blockiert	<code>wait(gabeln[1])</code>

# Beispiel

Philosoph 1	Philosoph 2	Philosoph 3
<code>wait(gabeln[1])</code> „hat linke Gabel“	<code>wait(gabeln[2])</code> „hat linke Gabel“	<code>wait(gabeln[3])</code> „hat linke Gabel“
<code>wait(gabeln[2])</code> blockiert	<code>wait(gabeln[3])</code> blockiert	<code>wait(gabeln[1])</code> blockiert

Deadlock möglich: Alle haben die linke Gabel, keiner die rechte!

## Philosophen: Versuch 2

---

Initial alle Gabeln mit 1 initialisiert, **mutex**: Binärer Semaphor, mit 1 initialisiert

```
Philosoph i
  loop forever
  (1) Philosoph denkt;
  (2) wait(mutex);
  (3) wait(gabel[i]); // linke Gabel
  (4) wait(gabel[i+1]); // rechte Gabel
  (5) Philosoph isst
  (6) signal(gabel[i+1]);
  (7) signal(gabel[i]);
  (8) signal(mutex);
  end loop
```

## Philosophen: Versuch 2

---

Initial alle Gabeln mit 1 initialisiert, **mutex**: Binärer Semaphor, mit 1 initialisiert

```
Philosoph i
  loop forever
  (1) Philosoph denkt;
  (2) wait(mutex);
  (3) wait(gabel[i]); // linke Gabel
  (4) wait(gabel[i+1]); // rechte Gabel
  (5) Philosoph isst
  (6) signal(gabel[i+1]);
  (7) signal(gabel[i]);
  (8) signal(mutex);
  end loop
```

Deadlock-frei, aber Starvation möglich: Ein Philosoph wird immer wieder überholt.  
Zudem schlecht: Nur ein Philosoph isst gleichzeitig

## Philosophen: Versuch 3

---

Initial alle Gabeln mit 1 initialisiert, **raum**: genereller Semaphor, mit  $N - 1$  initialisiert

```
Philosoph i
loop forever
(1) Philosoph denkt;
(2) wait(raum);
(3) wait(gabel[i]); // linke Gabel
(4) wait(gabel[i+1]); // rechte Gabel
(5) Philosoph isst
(6) signal(gabel[i+1]);
(7) signal(gabel[i]);
(8) signal(raum);
end loop
```

raum lässt immer nur maximal  $N - 1$  Philosophen gleichzeitig an die Gabeln

## Philosophen: Versuch 3

---

Initial alle Gabeln mit 1 initialisiert, **raum**: genereller Semaphor, mit  $N - 1$  initialisiert

```
Philosoph i
loop forever
(1) Philosoph denkt;
(2) wait(raum);
(3) wait(gabel[i]); // linke Gabel
(4) wait(gabel[i+1]); // rechte Gabel
(5) Philosoph isst
(6) signal(gabel[i+1]);
(7) signal(gabel[i]);
(8) signal(raum);
end loop
```

raum lässt immer nur maximal  $N - 1$  Philosophen gleichzeitig an die Gabeln

Deadlock und Starvation-frei

Initial alle Gabeln mit 1 initialisiert

Philosoph  $i$ ,  $i < N$

```
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i]); // linke Gabel
(3) wait(gabel[i+1]); // rechte Gabel
(4) Philosoph isst
(5) signal(gabel[i]+1);
(6) signal(gabel[i]);
end loop
```

Philosoph  $N$

```
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i+1]); // rechte Gabel
(3) wait(gabel[i]); // linke Gabel
(4) Philosoph isst
(5) signal(gabel[i]);
(6) signal(gabel[i+1]);
end loop
```

Initial alle Gabeln mit 1 initialisiert

Philosoph  $i$ ,  $i < N$

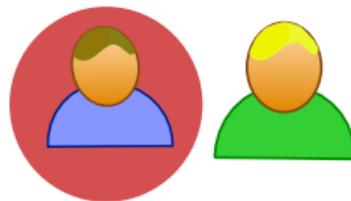
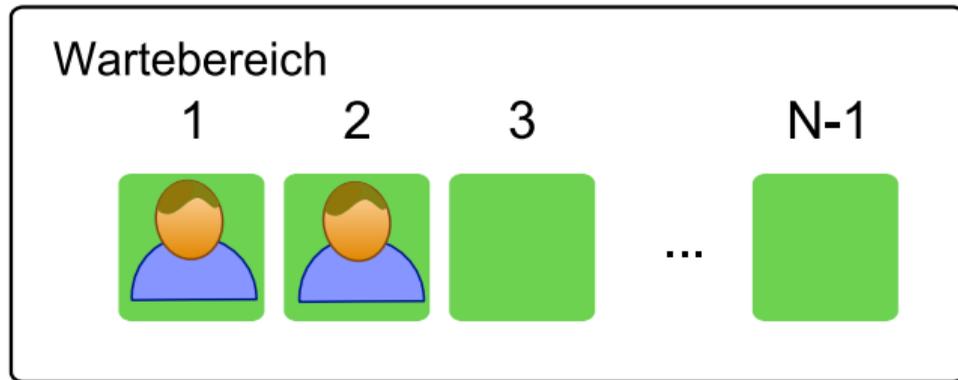
```
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i]); // linke Gabel
(3) wait(gabel[i+1]); // rechte Gabel
(4) Philosoph isst
(5) signal(gabel[i]+1);
(6) signal(gabel[i]);
end loop
```

Philosoph  $N$

```
loop forever
(1) Philosoph denkt;
(2) wait(gabel[i+1]); // rechte Gabel
(3) wait(gabel[i]); // linke Gabel
(4) Philosoph isst
(5) signal(gabel[i]);
(6) signal(gabel[i+1]);
end loop
```

Deadlock und Starvation-frei

# The Sleeping Barber



Frisier  
Stuhl      Friseur

# The Sleeping Barber

---

Situation:

- Friseur mit  $N - 1$  Warteplätzen und ein Frisierplatz
- Wenn Kunde da ist, wird er frisiert
- Wenn keine Kunde da ist, dann schläft Friseur, nächster Kunde weckt ihn
- Wenn Frisierplatz belegt, dann setzt sich Kunde auf Warteplatz
- Wenn alle Warteplätze belegt, dann geht Kunde sofort wieder.

# Lösung zum Sleeping Barber Problem

Initial: wartend: atomares Register, am Anfang 0  
kunden: genereller Semaphor, am Anfang 0  
mutex: binärer Semaphor, am Anfang 1  
synch,friseur: binärer Semaphor am Anfang 0

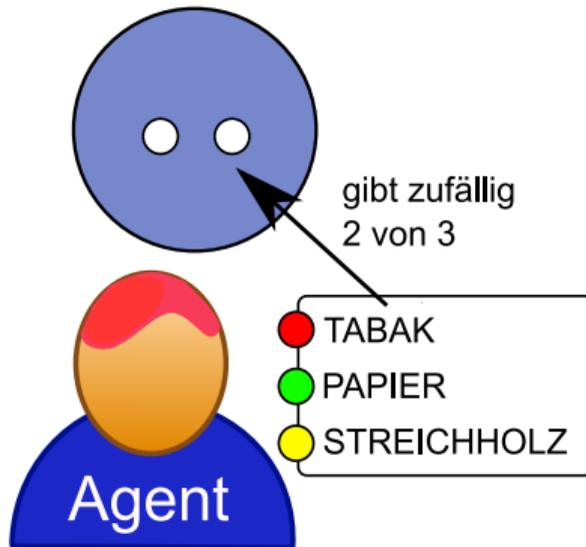
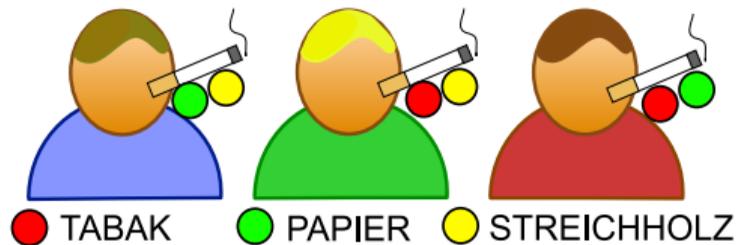
Friseur

```
loop forever
schlafe, solange keine Kunden:
(1) wait(kunden);
(2) wait(mutex);
(3) wartend := wartend -1;
nehme nächsten Kunden:
(4) signal(friseur);
(5) signal(mutex);
(6) schneide Haare;
warte, bis Kunde Laden verlässt:
(7) wait(synch);
end loop
```

Kunde

```
(1) wait(mutex);
(2) if wartend < N then
(3)   wartend := wartend + 1;
Wecke Friseur (bzw. erhöhe Kunden):
(4)   signal(kunden);
(5)   signal(mutex);
Warte bis Friseur bereit:
(6)   wait(friseur);
(7)   erhalte Frisur;
verlasse Laden:
(8)   signal(synch);
gehe sofort
(9) else signal(mutex);
```

# Cigarette Smoker's Problem



## Cigarette Smoker's Problem (2)

---

Problem-Beschreibung:

- 4 Personen: 3 Raucher, 1 Agent
- zum Rauchen einer Zigarette werden benötigt: Tabak, Papier, Streichhölzer
- die Raucher haben jeweils nur eine der Zutaten (unendlich viel davon)
- der Agent hat alle drei
- der Agent legt 2 der 3 Zutaten auf den Tisch
- der Raucher, der die dritte Zutat hat, nimmt die zwei weiteren und raucht
- Problem: Synchronisiere Raucher und Agenten

## Cigarette Smoker's Problem (3)

---

Modellierung durch 4 binäre Semaphore für den Agenten

- S[1] (Tabak), S[2] (Papier), S[3] (Streichholz):  
gibt an, ob Zutat auf dem Tisch liegt (initial 0)
- agent gibt an, ob der Agent Zutaten legt oder warten muss (initial 1)
- Programm des Agenten schon gegeben:  
loop forever  
    (1) wähle i und j zufällig aus {1, 2, 3};  
    (2) wait(agent);  
    (3) signal(S[i]);  
    (4) signal(S[j]);  
end loop

# Cigarette Smoker's Problem: Versuch 1

---

## Raucher mit Tabak

```
loop forever
(1) wait(S[2]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Papier

```
loop forever
(1) wait(S[1]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Streichholz

```
loop forever
(1) wait(S[1]);
(2) wait(S[2]);
(3) "rauche";
(4) signal(agent);
end loop
```

S[1].V   S[2].V   S[3].V  
0        0        0

# Cigarette Smoker's Problem: Versuch 1

---

## Raucher mit Tabak

```
loop forever
(1) wait(S[2]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Papier

```
loop forever
(1) wait(S[1]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Streichholz

```
loop forever
(1) wait(S[1]);
(2) wait(S[2]);
(3) "rauche";
(4) signal(agent);
end loop
```

Agent: Tabak und Papier (signal(S[1]) + signal(S[2]))

S[1].V	S[2].V	S[3].V
1	1	0

# Cigarette Smoker's Problem: Versuch 1

## Raucher mit Tabak

```
loop forever
(1) wait(S[2]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Papier

```
loop forever
(1) wait(S[1]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Streichholz

```
loop forever
(1) wait(S[1]);
(2) wait(S[2]);
(3) "rauche";
(4) signal(agent);
end loop
```

Agent: Tabak und Papier (signal(S[1]) + signal(S[2]))

S[1].V	S[2].V	S[3].V
0	1	0

# Cigarette Smoker's Problem: Versuch 1

---

## Raucher mit Tabak

```
loop forever
(1) wait(S[2]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Papier

```
loop forever
(1) wait(S[1]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Streichholz

```
loop forever
(1) wait(S[1]);
(2) wait(S[2]);
(3) "rauche";
(4) signal(agent);
end loop
```

Agent: Tabak und Papier (signal(S[1]) + signal(S[2]))

S[1].V	S[2].V	S[3].V
0	0	0

# Cigarette Smoker's Problem: Versuch 1

---

## Raucher mit Tabak

```
loop forever
(1) wait(S[2]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Papier

```
loop forever
(1) wait(S[1]);
(2) wait(S[3]);
(3) "rauche";
(4) signal(agent);
end loop
```

## Raucher mit Streichholz

```
loop forever
(1) wait(S[1]);
(2) wait(S[2]);
(3) "rauche";
(4) signal(agent);
end loop
```

Agent: Tabak und Papier (signal(S[1]) + signal(S[2]))

S[1].V   S[2].V   S[3].V  
0        0        0

**DEADLOCK!**

## Cigarette Smoker's Problem: Versuch 2

NEU:  $R[i]$ ,  $i = 1, \dots, 6$ , bin. Sem. (initial 0),  
mutex: bin. Sem. (initial 1),  
t: atom. Register (initial 0)

### Helfer (Tabak)

```
loop forever
(1) wait(S[1]);
(2) wait(mutex);
(3) t := t+1;
(4) if t ≠ 1 then
(5)   signal(R[t]);
(4) signal(mutex);
end loop
```

### Raucher mit Tabak

```
loop forever
(1) wait(R[6]);
(2) t := 0;
(3) "rauche";
(4) signal(agent);
```

### Helfer (Papier)

```
loop forever
(1) wait(S[2]);
(2) wait(mutex);
(3) t := t+2;
(4) if t ≠ 2 then
(5)   signal(R[t]);
(4) signal(mutex);
end loop
```

### Raucher mit Papier

```
loop forever
(1) wait(R[5]);
(2) t := 0;
(3) "rauche";
(4) signal(agent);
```

### Helfer (Streichholz)

```
loop forever
(1) wait(S[3]);
(2) wait(mutex);
(3) t := t+4;
(4) if t ≠ 4 then
(5)   signal(R[t]);
(4) signal(mutex);
end loop
```

### Raucher mit Streichholz

```
loop forever
(1) wait(R[3]);
(2) t := 0;
(3) "rauche";
(4) signal(agent);
```

## Der richtige Raucher wird geweckt ...

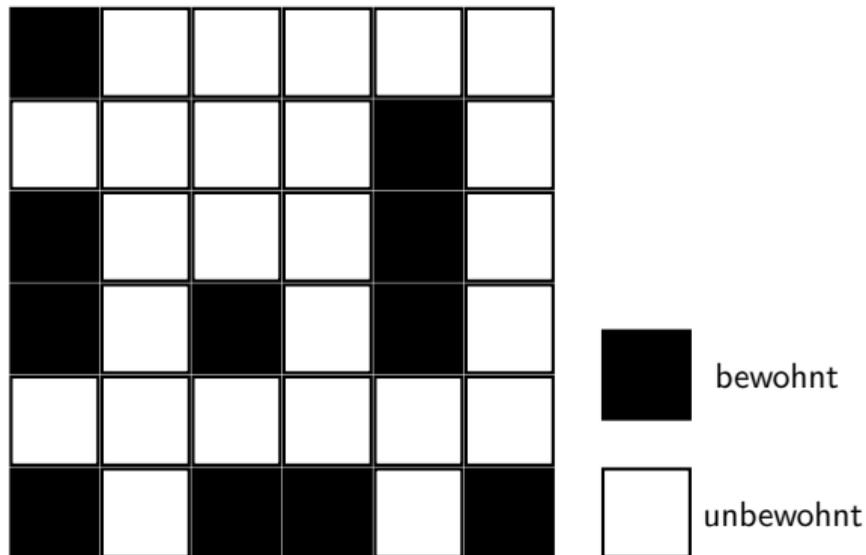
Zutaten auf dem Tisch	erster Helfer	zweiter (=weckender) Helfer	Wert von t	geweckter Raucher
Tabak & Papier	(Tabak)	(Papier)	$1+2 = 3$	R[3] (=Streichh.)
Tabak & Papier	(Papier)	(Tabak)	$2+1 = 3$	R[3] (=Streichh.)
Tabak & Streichh.	(Tabak)	(Streichh.)	$1+4 = 5$	R[5] (=Papier)
Tabak & Streichh.	(Streichh.)	(Tabak)	$4+1 = 5$	R[5] (=Papier)
Papier & Streichh.	(Papier)	(Streichh.)	$2+4 = 6$	R[6] (=Tabak)
Papier & Streichh.	(Streichh.)	(Papier)	$4+2 = 6$	R[6] (=Tabak)

- Manche Algorithmen erfordern “Phasen”
- D.h.: Die Prozesse führen Berechnungen durch, aber an einem Schritt warten alle Prozesse aufeinander
- Erst wenn alle an dieser Stelle angekommen sind, dürfen die Prozesse weiter rechnen
- Ähnlich war es beim Mergesort-Beispiel, dort wartet allerdings nur ein Prozess auf zwei weitere

# Anwendungsbeispiel: Game of Life

---

Spielfeld von Conways Game of Life:  $N \times N$ -Matrix

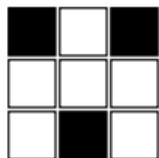


**Ziel:** Berechne stets die nächste Generation aus der aktuellen

# Spielregeln

---

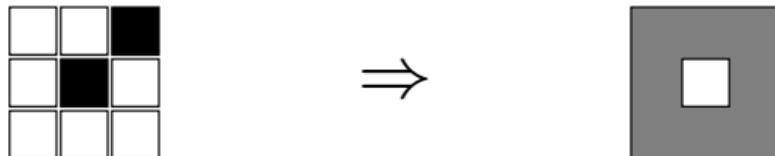
- Feld unbewohnt: Wieder bewohnt genau dann, wenn  $\#(\text{Nachbarn}) = 3$ , z.B.



# Spielregeln

– Feld bewohnt:

– Wenn  $\#(\text{Nachbarn}) < 2$ , dann unbewohnt (Unterpopulation), z.B.



– Wenn  $\#(\text{Nachbarn}) \in \{2, 3\}$ , dann weiterhin bewohnt. Z.B.

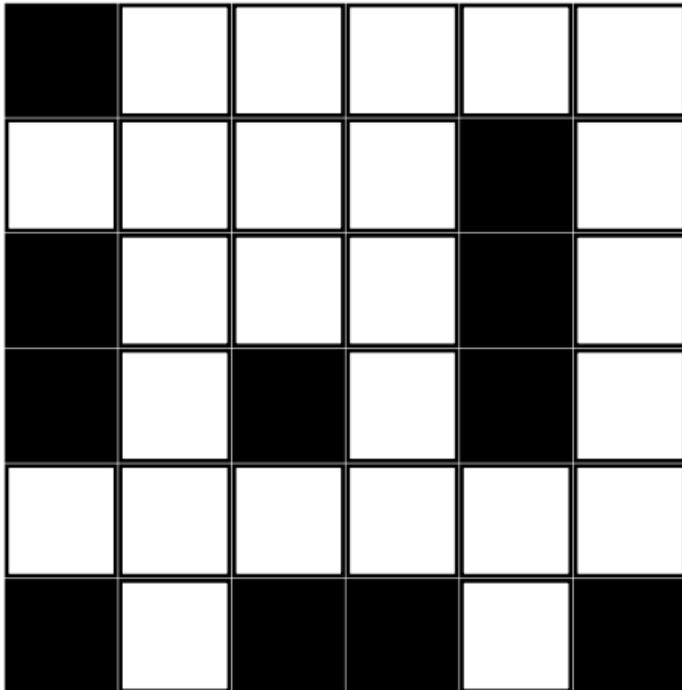


– Wenn  $\#(\text{Nachbarn}) > 3$ , dann unbewohnt danach (Überpopulation), z.B.



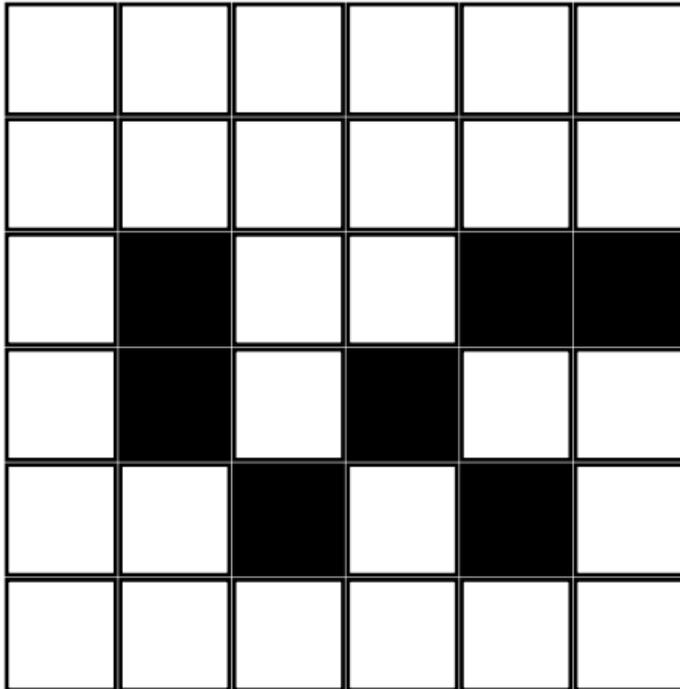
# Beispiel

---



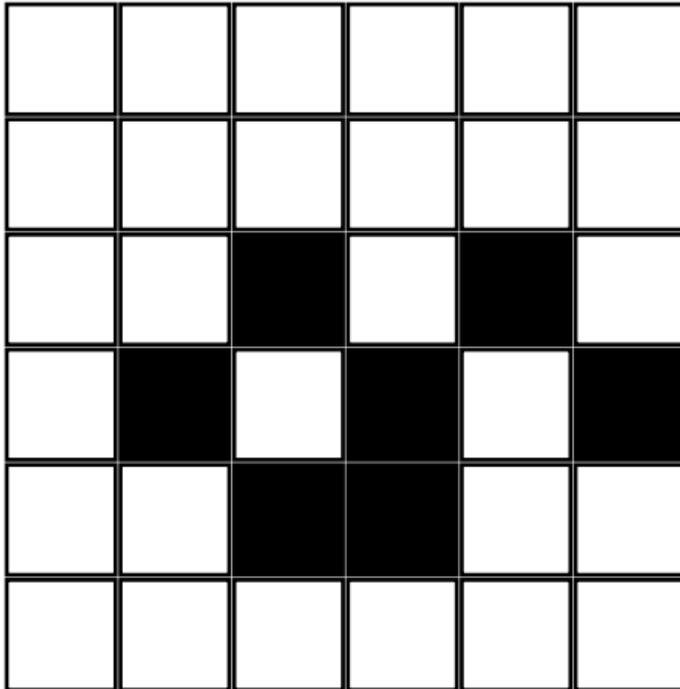
# Beispiel

---



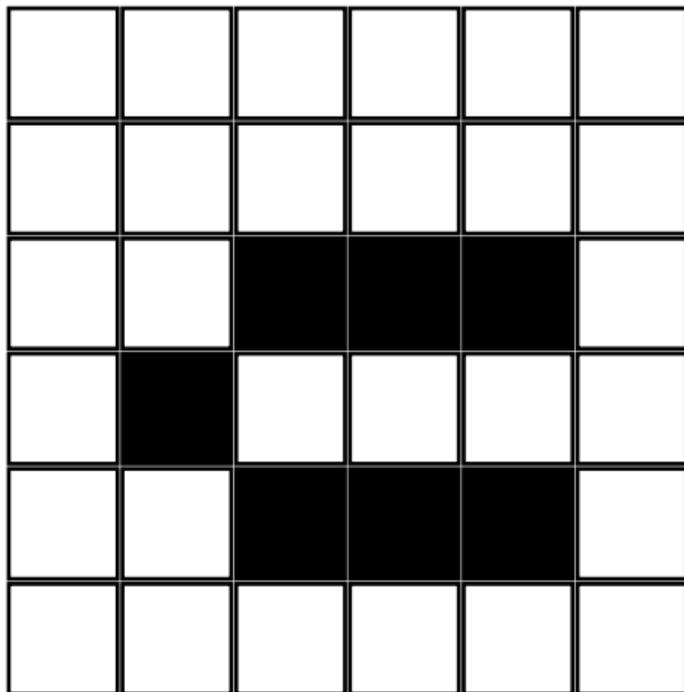
# Beispiel

---



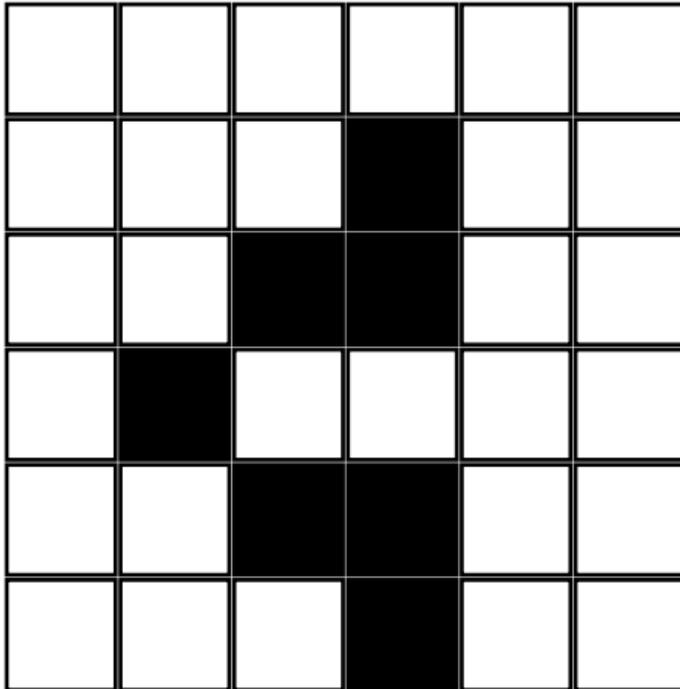
# Beispiel

---



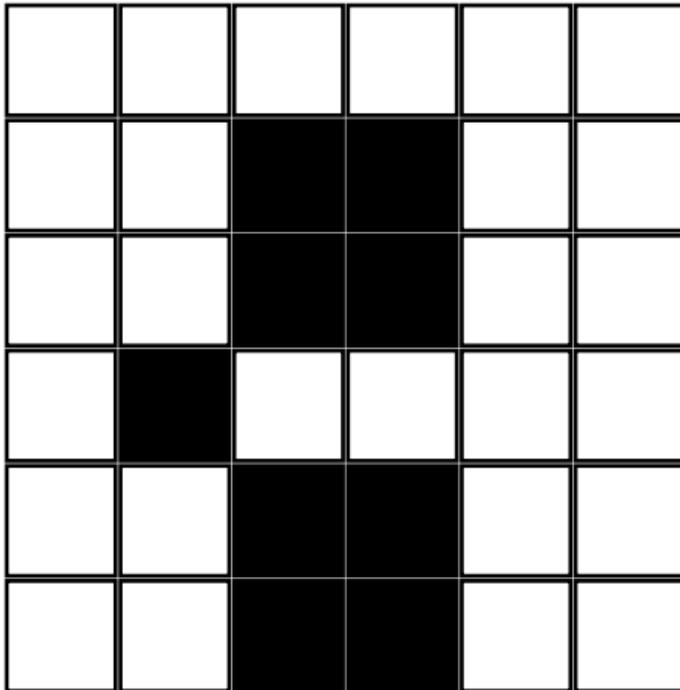
# Beispiel

---



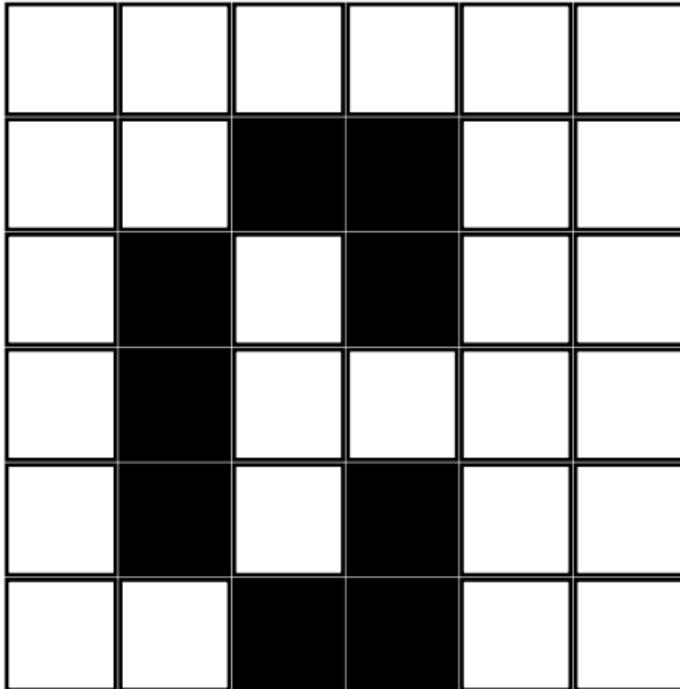
# Beispiel

---



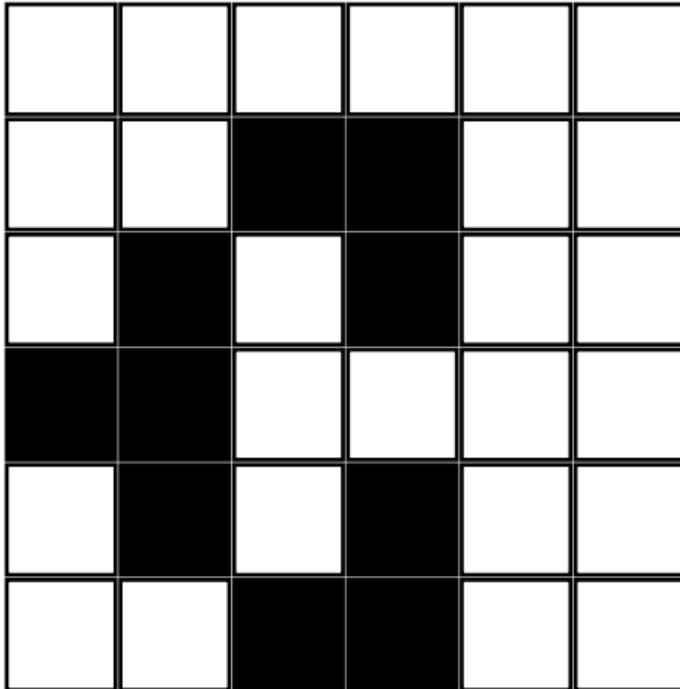
# Beispiel

---



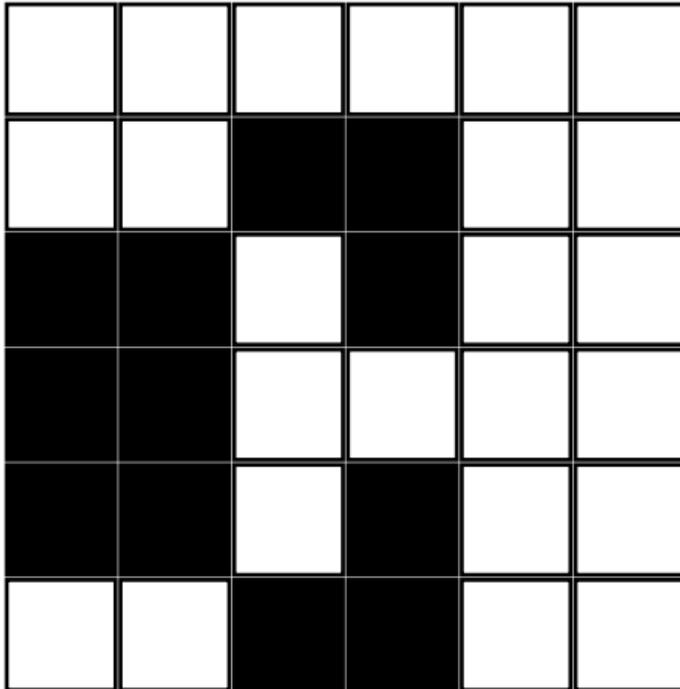
# Beispiel

---



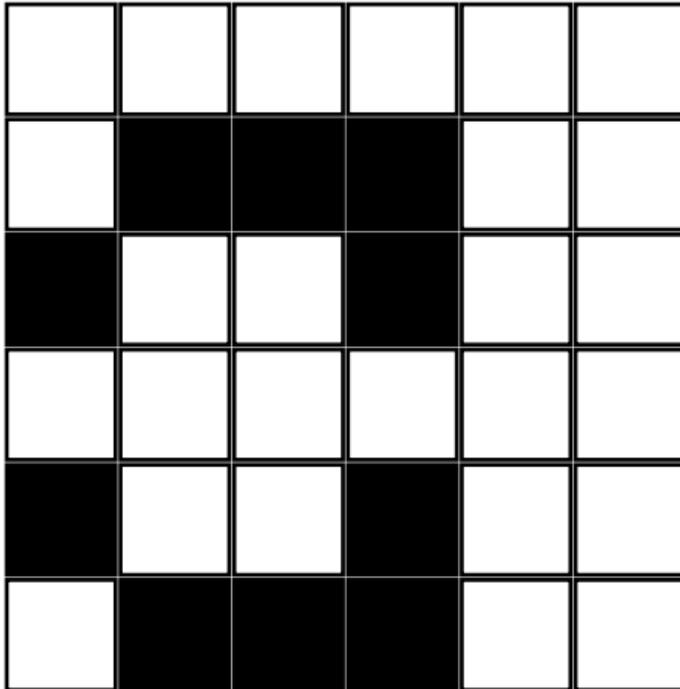
# Beispiel

---



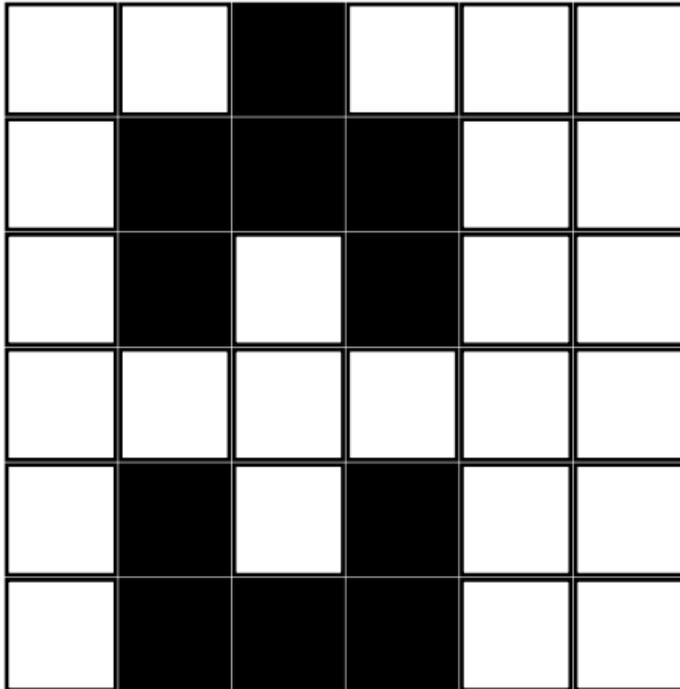
# Beispiel

---



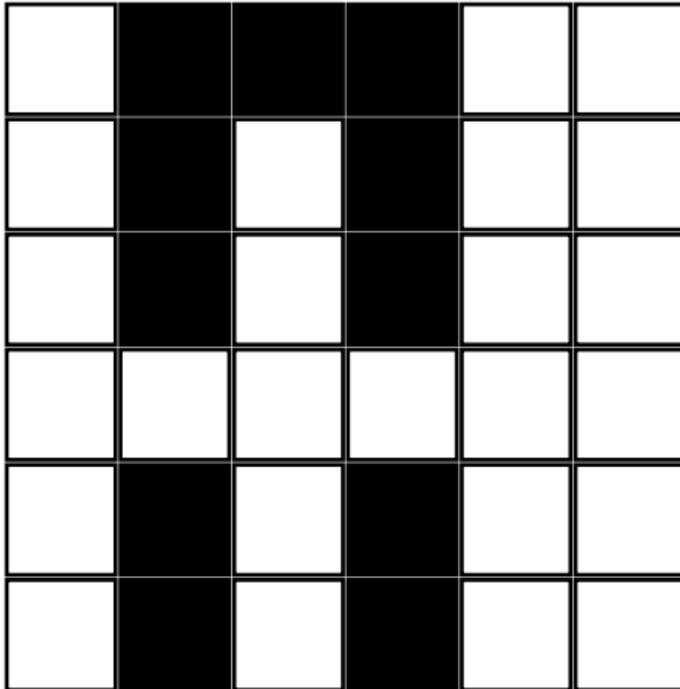
# Beispiel

---



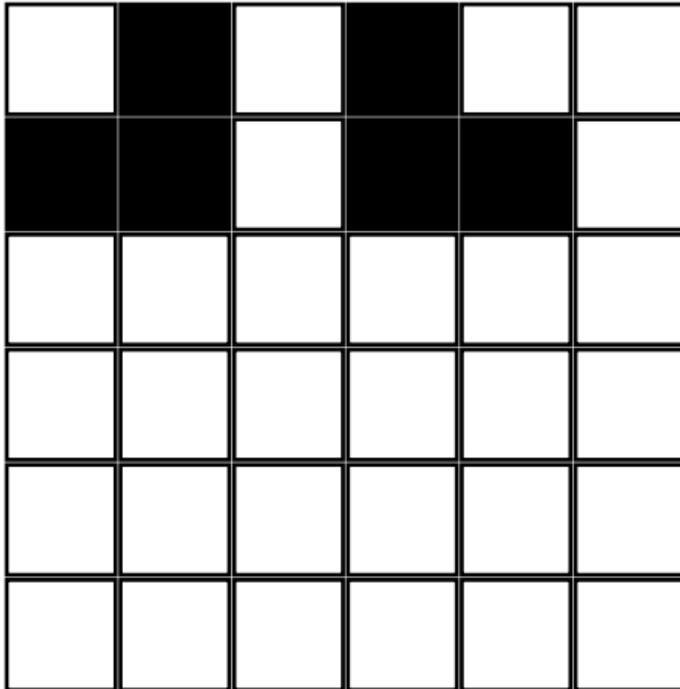
# Beispiel

---



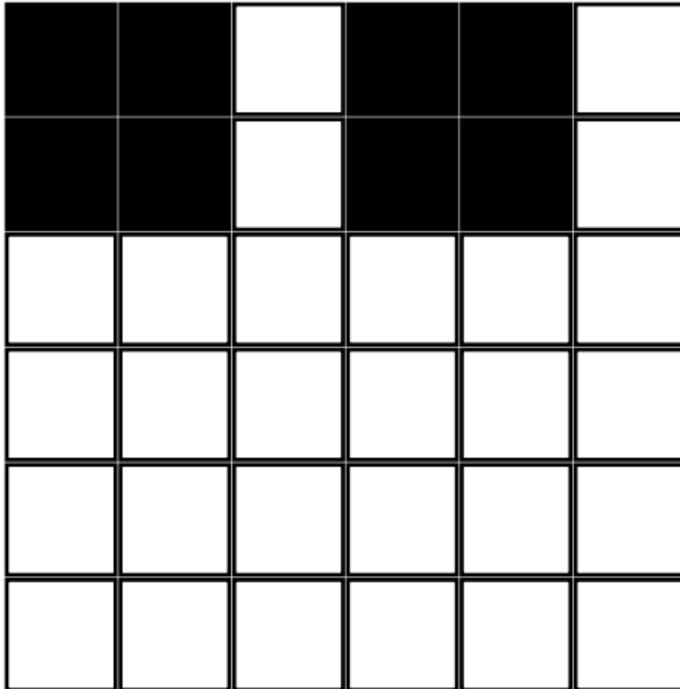
# Beispiel

---



# Beispiel

---



## Annahmen:

- Spielfeld in  $N \times N$ -Array mit Booleschen Einträgen
- `naechsterWert( $i, j, feld$ )`: berechne den nächsten Wert für Eintrag  $(i, j)$ : Lese alle Nachbarn und  $(i, j)$ -Eintrag, entscheide dann True oder False

## $k$ . Generation berechnen

`feld`: Initialisiertes  $N \times N$  Array, das das Spielfeld darstellt

`feld2`:  $N \times N$  Array zum Zwischenspeichern

Algorithmus:

```
for g:=1 to k do
  for i=1 to N do
    for j=1 to N do feld2[i,j] := naechsterWert(i,j,feld);
  for i=1 to N do
    for j=1 to N do feld[i,j] := feld2[i,j];
```

- Ein Prozess pro Feld  $(i,j)$ , berechnet den Eintrag für das Feld

## Paraller Algorithmus

feld: Initialisiertes  $N \times N$  Array, dass das Spielfeld darstellt

$(N \times N)$  Prozesse: jeweils einen pro Spielfeld

Programm für Prozess  $(i,j)$ :

```
for g:=1 to k
  v := naechsterWert(i,j,feld);
  feld[i,j]:=v;
```

Funktioniert nicht!

## Paraller Algorithmus

feld: Initialisiertes  $N \times N$  Array, dass das Spielfeld darstellt

$(N \times N)$  Prozesse: jeweils einen pro Spielfeld

Programm für Prozess  $(i,j)$ :

for  $g:=1$  to  $k$

$v := \text{naechsterWert}(i,j,\text{feld});$

    Warte bis alle Prozesse  $v$  berechnet haben

$\text{feld}[i,j] := v;$

    Warte bis alle Prozesse ihr update geschrieben haben

## Paraller Algorithmus

feld: Initialisiertes  $N \times N$  Array, dass das Spielfeld darstellt

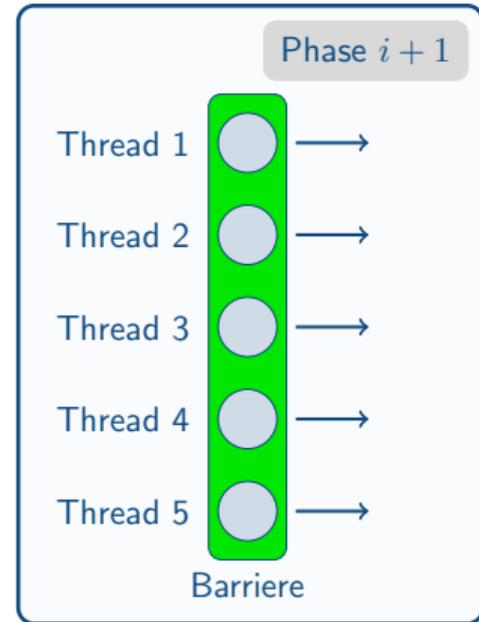
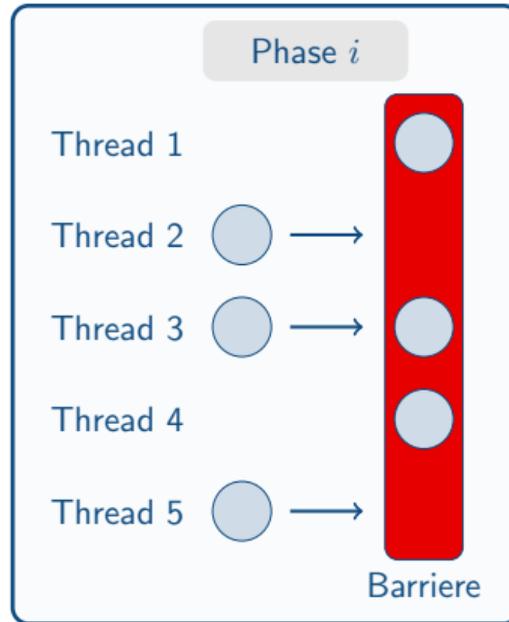
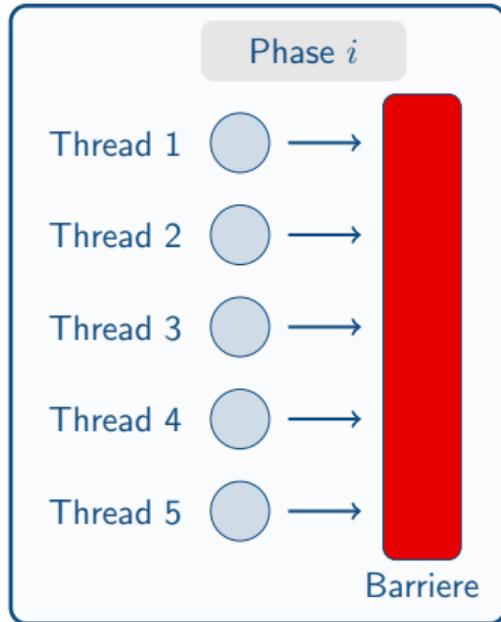
barrier: Barriere für  $N \times N$  Prozesse

$(N \times N)$  Prozesse: jeweils einen pro Spielfeld

Programm für Prozess  $(i,j)$ :

```
for g:=1 to k
  v := naechsterWert(i,j,feld);
  synchBarrier(barrier);
  feld[i,j] := v;
  synchBarrier(barrier);
```

# Allgemeines Schema



# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

## Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

## Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

## Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

## Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
1	∅		0	∅

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

## Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

## Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
1	∅		0	{P1}

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
1	∅		0	∅

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

# Barriere für 2 Prozesse

---

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

## Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

## Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

# Barriere für 2 Prozesse

Initial: p1ready, p2ready: binäre Semaphore am Anfang 0

## Programm für Prozess 1:

- (1) Berechnung vor der Barriere
- (2) `signal(p1ready);`
- (3) `wait(p2ready);`
- (4) Berechnung nach der Barriere

## Programm für Prozess 2:

- (1) Berechnung vor der Barriere
- (2) `signal(p2ready);`
- (3) `wait(p1ready);`
- (4) Berechnung nach der Barriere

p1ready.V	p1ready.M		p2ready.V	p2ready.M
0	∅		0	∅

## Barriere für $n$ Prozesse

---

Initial: ankommen, bin. Semaphor mit 1 initialisiert, verlassen, bin. Semaphor mit 0 initialisiert  
counter: atomares Register mit 0 initialisiert

### Programm für Prozess $i$ :

- (1) Berechnung vor der Barriere;
- (2) `wait(ankommen);`
- (3) `counter := counter + 1;`
- (4) `if counter <  $n$  // Sonderaufgabe für letzten Prozess`
- (5)     `then signal(ankommen);`
- (6)     `else signal(verlassen);`
- (7) `wait(verlassen);`
- (8) `counter := counter - 1;`
- (9) `if counter > 0 // Sonderaufgabe für letzten Prozess`
- (10)     `then signal(verlassen);`
- (11)     `else signal(ankommen);`
- (12) Berechnung nach der Barriere;

# Barrieren als abstrakter Datentyp

---

- Wir verwenden Barrieren auch als ADT
- Interne Darstellung z.B. als 4-Tupel  $(n, \text{ankommen}, \text{verlassen}, \text{counter})$

Operationen:

- `newBarrier( $k$ )`: Erzeugt eine Barriere für  $k$  Prozesse.  
(intern: zwei Semaphore für ankommen und verlassen und ein atomares Register counter erzeugen,  
Rückgabe ist 4-Tupel  $(n, \text{ankommen}, \text{verlassen}, \text{counter})$ )
- `synchBarrier( $B$ )`: Synchronisieren an der Barriere, d.h. die Zeilen (2) bis (12) werden ausgeführt für das 4-Tupel  $B$ .

# Barriere in Java

---

```
java.util.concurrent.CyclicBarrier
```

```
class CyclicBarrier
```

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point... The barrier is called cyclic because it can be re-used after the waiting threads are released.

Konstruktor:

```
CyclicBarrier(int parties)
```

Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action when the barrier is tripped.

Methoden:

```
public int await()
```

```
public int await()
```

```
    throws InterruptedException,  
           BrokenBarrierException
```

Waits until all parties have invoked await on this barrier.

## Gruppierung der Prozesse in

- **Readers:** Prozesse, die auf eine gemeinsame Ressource **lesend** zugreifen
- **Writers:** Prozesse, die auf die gemeinsame Ressource **schreibend** zugreifen

## Beispiel: Flugbuchungssystem

- Manche Zugriffe nur lesend: welche Flüge gibt es, wann usw.
- andere Zugriffe buchen Flüge, verändern damit die zur Verfügung stehenden Flüge

### **Erlaubt / Nicht erlaubt**

- Mehrere lesende Prozesse gleichzeitig, aber
- Nur ein Prozess schreibt gleichzeitig

### **Problem:**

- Löse den Zugriff so, dass viele gleichzeitig lesen, aber nie mehrere gleichzeitig schreiben.

### **Verschiedene Lösungen:**

- Priorität für Readers
- Priorität für Writers

# Priorität für Readers

Initial: countR: atomares Register, am Anfang 0  
mutex, mutexR, w: bin. Sem. am Anfang 1

## Programm für Reader

- (1) wait(mutexR);
- (2) countR := countR + 1;
- (3) if countR = 1 then
- (4)     wait(w);
- (5) signal(mutexR);
- (6) Kritischer Abschnitt
- (7) wait(mutexR);
- (8) countR := countR - 1;
- (9) if countR = 0 then
- (10)     signal(w);
- (11) signal(mutexR);

## Programm für Writer

- (1) wait(mutex);
- (2) wait(w);
- (3) signal(mutex);
- (4) Kritischer Abschnitt;
- (5) signal(w)

# Priorität für Readers

Initial: countR: atomares Register, am Anfang 0  
mutex, mutexR, w: bin. Sem. am Anfang 1

## Programm für Reader

```
(1) wait(mutexR);
(2) countR := countR + 1;
(3) if countR = 1 then
(4)     wait(w);
(5) signal(mutexR);
(6) Kritischer Abschnitt
(7) wait(mutexR);
(8) countR := countR - 1;
(9) if countR = 0 then
(10)     signal(w);
(11) signal(mutexR);
```

## Programm für Writer

```
(1) wait(mutex);
(2) wait(w);
(3) signal(mutex);
(4) Kritischer Abschnitt;
(5) signal(w)
```

- mutexR schützt Zugriff auf countR

# Priorität für Readers

Initial: countR: atomares Register, am Anfang 0  
mutex, mutexR, w: bin. Sem. am Anfang 1

## Programm für Reader

```
(1) wait(mutexR);
(2) countR := countR + 1;
(3) if countR = 1 then
(4)     wait(w);
(5) signal(mutexR);
(6) Kritischer Abschnitt
(7) wait(mutexR);
(8) countR := countR - 1;
(9) if countR = 0 then
(10)     signal(w);
(11) signal(mutexR);
```

## Programm für Writer

```
(1) wait(mutex);
(2) wait(w);
(3) signal(mutex);
(4) Kritischer Abschnitt;
(5) signal(w)
```

- mutexR schützt Zugriff auf countR
- **erster Leser blockiert Schreiber, bzw. wartet, dass Schreiber fertig wird**

# Priorität für Readers

Initial: countR: atomares Register, am Anfang 0  
mutex, mutexR, w: bin. Sem. am Anfang 1

## Programm für Reader

```
(1) wait(mutexR);
(2) countR := countR + 1;
(3) if countR = 1 then
(4)     wait(w);
(5) signal(mutexR);
(6) Kritischer Abschnitt
(7) wait(mutexR);
(8) countR := countR - 1;
(9) if countR = 0 then
(10)     signal(w);
(11) signal(mutexR);
```

## Programm für Writer

```
(1) wait(mutex);
(2) wait(w);
(3) signal(mutex);
(4) Kritischer Abschnitt;
(5) signal(w)
```

- mutexR schützt Zugriff auf countR
- erster Leser blockiert Schreiber, bzw. wartet, dass Schreiber fertig wird
- **letzter Leser entblockiert Schreiber**

# Priorität für Readers

Initial: countR: atomares Register, am Anfang 0  
mutex, mutexR, w: bin. Sem. am Anfang 1

## Programm für Reader

```
(1) wait(mutexR);
(2) countR := countR + 1;
(3) if countR = 1 then
(4)     wait(w);
(5) signal(mutexR);
(6) Kritischer Abschnitt
(7) wait(mutexR);
(8) countR := countR - 1;
(9) if countR = 0 then
(10)     signal(w);
(11) signal(mutexR);
```

## Programm für Writer

```
(1) wait(mutex);
(2) wait(w);
(3) signal(mutex);
(4) Kritischer Abschnitt;
(5) signal(w)
```

- mutexR schützt Zugriff auf countR
- erster Leser blockiert Schreiber, bzw. wartet, dass Schreiber fertig wird
- letzter Leser entblockiert Schreiber
- mutex sorgt dafür, dass maximal 1 Schreiber an w wartet

# Priorität für Writers

Initial: countR, countW: atomare Register, am Anfang 0  
mutexR, mutexW, mutex, w, r: bin. Sem. am Anfang 1

## Programm für Reader

```
(1) wait(mutex);
(2) wait(r);
(3) wait(mutexR);
(4) countR := countR + 1;
(5) if countR = 1 then wait(w);
(6) signal(mutexR);
(7) signal(r);
(8) signal(mutex);
(9) Kritischer Abschnitt;
(10) wait(mutexR);
(11) countR := countR - 1;
(12) if countR = 0 then signal(w);
(13) signal(mutexR);
```

## Programm für Writer

```
(1) wait(mutexW);
(2) countW := countW + 1;
(3) if countW = 1 then wait(r);
(4) signal(mutexW);
(5) wait(w);
(6) Kritischer Abschnitt;
(7) signal(w)
(8) wait(mutexW);
(9) countW := countW - 1;
(10) if countW = 0 then signal(r);
(11) signal(mutexW);
```

# Priorität für Writers

Initial: countR, countW: atomare Register, am Anfang 0  
mutexR, mutexW, mutex, w, r: bin. Sem. am Anfang 1

## Programm für Reader

- (1) wait(mutex);
- (2) wait(r);
- (3) wait(mutexR);
- (4) countR := countR + 1;
- (5) if countR = 1 then wait(w);
- (6) signal(mutexR);
- (7) signal(r);
- (8) signal(mutex);
- (9) Kritischer Abschnitt;
- (10) wait(mutexR);
- (11) countR := countR - 1;
- (12) if countR = 0 then signal(w);
- (13) signal(mutexR);

## Programm für Writer

- (1) wait(mutexW);
  - (2) countW := countW + 1;
  - (3) if countW = 1 then wait(r);
  - (4) signal(mutexW);
  - (5) wait(w);
  - (6) Kritischer Abschnitt;
  - (7) signal(w)
  - (8) wait(mutexW);
  - (9) countW := countW - 1;
  - (10) if countW = 0 then signal(r);
  - (11) signal(mutexW);
- mutexR, mutexW schützen countR, countW

# Priorität für Writers

Initial: countR, countW: atomare Register, am Anfang 0  
mutexR, mutexW, mutex, w, r: bin. Sem. am Anfang 1

## Programm für Reader

- (1) wait(mutex);
- (2) wait(r);
- (3) wait(mutexR);
- (4) countR := countR + 1;
- (5) **if countR = 1 then wait(w);**
- (6) signal(mutexR);
- (7) signal(r);
- (8) signal(mutex);
- (9) Kritischer Abschnitt;
- (10) wait(mutexR);
- (11) countR := countR - 1;
- (12) if countR = 0 then signal(w);
- (13) signal(mutexR);

## Programm für Writer

- (1) wait(mutexW);
  - (2) countW := countW + 1;
  - (3) if countW = 1 then wait(r);
  - (4) signal(mutexW);
  - (5) wait(w);
  - (6) Kritischer Abschnitt;
  - (7) signal(w)
  - (8) wait(mutexW);
  - (9) countW := countW - 1;
  - (10) if countW = 0 then signal(r);
  - (11) signal(mutexW);
- mutexR, mutexW schützen countR, countW
  - **erster Leser blockiert Schreiber**

# Priorität für Writers

Initial: countR, countW: atomare Register, am Anfang 0  
mutexR, mutexW, mutex, w, r: bin. Sem. am Anfang 1

## Programm für Reader

```
(1) wait(mutex);
(2) wait(r);
(3) wait(mutexR);
(4) countR := countR + 1;
(5) if countR = 1 then wait(w);
(6) signal(mutexR);
(7) signal(r);
(8) signal(mutex);
(9) Kritischer Abschnitt;
(10) wait(mutexR);
(11) countR := countR - 1;
(12) if countR = 0 then signal(w);
(13) signal(mutexR);
```

## Programm für Writer

```
(1) wait(mutexW);
(2) countW := countW + 1;
(3) if countW = 1 then wait(r);
(4) signal(mutexW);
(5) wait(w);
(6) Kritischer Abschnitt;
(7) signal(w)
(8) wait(mutexW);
(9) countW := countW - 1;
(10) if countW = 0 then signal(r);
(11) signal(mutexW);
```

- mutexR, mutexW schützen countR, countW
- erster Leser blockiert Schreiber
- **erster Schreiber blockiert Leser**

# Priorität für Writers

Initial: countR, countW: atomare Register, am Anfang 0  
mutexR, mutexW, mutex, w, r: bin. Sem. am Anfang 1

## Programm für Reader

- (1) `wait(mutex);`
- (2) `wait(r);`
- (3) `wait(mutexR);`
- (4) `countR := countR + 1;`
- (5) `if countR = 1 then wait(w);`
- (6) `signal(mutexR);`
- (7) `signal(r);`
- (8) `signal(mutex);`
- (9) Kritischer Abschnitt;
- (10) `wait(mutexR);`
- (11) `countR := countR - 1;`
- (12) `if countR = 0 then signal(w);`
- (13) `signal(mutexR);`

## Programm für Writer

- (1) `wait(mutexW);`
  - (2) `countW := countW + 1;`
  - (3) `if countW = 1 then wait(r);`
  - (4) `signal(mutexW);`
  - (5) `wait(w);`
  - (6) Kritischer Abschnitt;
  - (7) `signal(w)`
  - (8) `wait(mutexW);`
  - (9) `countW := countW - 1;`
  - (10) `if countW = 0 then signal(r);`
  - (11) `signal(mutexW);`
- mutexR, mutexW schützen countR, countW
  - erster Leser blockiert Schreiber
  - erster Schreiber blockiert Leser
  - `mutex` sorgt dafür, dass nur ein Leser an `r` warten kann