

# Kapitel 6

## Verklemmungen (Deadlocks)

Prof. Dr. Rolf Hennicker

29.06.2017

## 6.1 Charakterisierung von Deadlocks

Eine Verklemmung entsteht, wenn alle Prozesse in einem System blockiert sind. In diesem Fall sind keine weiteren Aktionen möglich. *Java: BlockedByMonitor oder waiting*

### Notwendige und hinreichende Bedingungen für Deadlocks:

(nach Coffmann et al. 1971) *1.  $\wedge$  2.  $\wedge$  3.  $\wedge$  4.  $\Leftrightarrow$  Deadlock*

#### 1. "Mutual Exclusion"

Es gibt mindestens zwei Ressourcen, die nur unter wechselseitigem Ausschluss benützt werden können.

#### 2. "Hold and Wait"

Prozesse belegen schon eine Ressource während sie darauf warten, eine weitere Ressource zu belegen.

#### 3. "No Preemption"

Prozesse können nicht "gezwungen" werden, eine Ressource wieder freizugeben.

#### 4. "Wait for Cycle" (*Circular Wait*)

Es gibt eine zyklische Folge von Prozessen, so dass jeder Prozess eine Ressource belegt, auf die sein Nachfolger wartet.

## Deadlock Vermeidung:

Eine der 4 Bedingungen verletzen!

1. + 2. können häufig nicht verletzt werden,  
da Anwendungs-inhärent
3. verletzen kann problematisch sein;  
vgl. Folie 8, 2.
4. verhindern ist häufig der beste Ansatz.

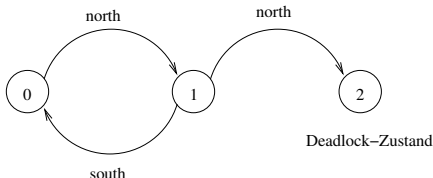
## 6.2 Deadlock-Analyse in FSP

Deadlocks werden in FSP repräsentiert durch den Prozess STOP, dessen LTS nur einen Zustand und keine Transitionen enthält.



### Beispiel:

MOVE = (north  $\rightarrow$  (south  $\rightarrow$  MOVE | north  $\rightarrow$  STOP)).



Ablauf zu Deadlock:

north  
north

*Bemerkung: Es gibt in FSP auch den konstanten Prozess aus dem die END zur Modellierung von gewünschter Terminierung.*

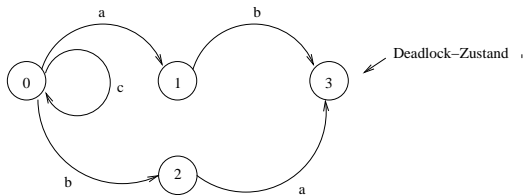
## Implizite Deadlock-Zustände

Deadlock-Zustände können sich bei der parallelen Komposition von Prozessen ergeben. Sie sind dann *nicht* explizit mit STOP (in einem lokalen Prozess) gekennzeichnet, also bei der Prozessdeklaration nicht offensichtlich.

### Beispiel:

$$A = (a \rightarrow \text{north} \rightarrow \text{south} \rightarrow A \mid c \rightarrow A).$$

$$B = (b \rightarrow \text{south} \rightarrow \text{north} \rightarrow B \mid c \rightarrow B).$$

$$\parallel AB = (A \parallel B).$$


Stark äquivalent ist:

$$D = (a \rightarrow b \rightarrow \text{STOP} \mid b \rightarrow a \rightarrow \text{STOP} \\ \mid c \rightarrow D) + \{\text{north, south}\}.$$

## Auffinden von Deadlock-Zuständen

Zum Auffinden von Deadlock-Zuständen *und* von hinführenden Abläufen (Traces) kann eine Breitensuche ("breadth-first search") ausgehend vom Anfangszustand durchgeführt werden. Falls ein Deadlock existiert, erhält man einen minimalen Weg dorthin.

### Beachte:

Da FSP nur endlich viele Zustände zulässt, ist die Deadlocksuche entscheidbar.

Beispiel: Betrachte den Prozess AB.

Beginne beim Zustand ①. Kein Deadlock

1. ①  $\xrightarrow{a}$  ② kein Deadlock bei ②; merke ②

2. ①  $\xrightarrow{b}$  ③ " " bei ③; merke ③

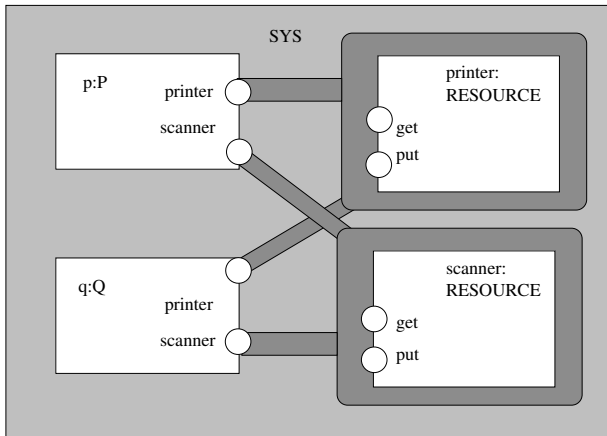
3. ①  $\xrightarrow{c}$  ① ① schon besucht

1.1 ②  $\xrightarrow{b}$  ④ Deadlock bei ④

Ein minimaler Ablauf zu Deadlock ist  $a b$

## Beispiel (Printer & Scanner):

Zwei Prozesse  $p:P$  und  $q:Q$  brauchen jeweils einen Printer und einen Scanner zum Kopieren.



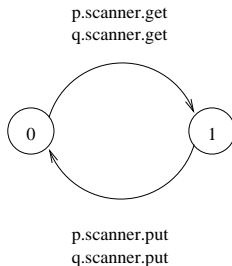
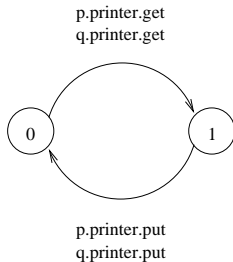
RESOURCE = (get  $\rightarrow$  put  $\rightarrow$  RESOURCE).

P = (printer.get  $\rightarrow$  scanner.get  $\rightarrow$  copy  $\rightarrow$  printer.put  $\rightarrow$  scanner.put  $\rightarrow$  P).

Q = (scanner.get  $\rightarrow$  printer.get  $\rightarrow$  copy  $\rightarrow$  scanner.put  $\rightarrow$  printer.put  $\rightarrow$  Q).

||SYS = (p:P || q:Q || {p,q}::printer:RESOURCE || {p,q}::scanner:RESOURCE).

LTS von {p,q}::printer:RESOURCE und {p,q}::scanner:RESOURCE:



Abläufe zu Deadlock:

p.printer.get  
q.scanner.get

oder

q.scanner.get  
p.printer.get



Möglichkeiten, um Deadlock zu vermeiden:

1. Die beiden Prozesse greifen in derselben Reihenfolge auf den Drucker und den Scanner zu.
2. Mit "Timeout": Nach einer bestimmten Zeitspanne des Wartens wird die Ressource wieder freigegeben. Gefahr: Kein echter Fortschritt!

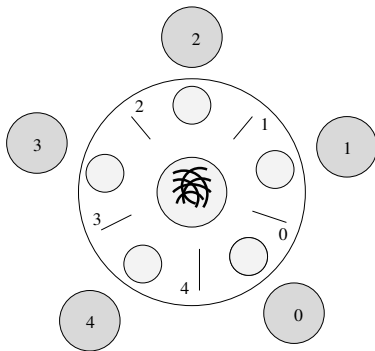
P = (printer.get  $\rightarrow$  GETSCANNER),  
 GETSCANNER = (scanner.get  $\rightarrow$  copy  $\rightarrow$  printer.put  $\rightarrow$  scanner.put  $\rightarrow$  P  
 | timeout  $\rightarrow$  printer.put  $\rightarrow$  P).

Q = (scanner.get  $\rightarrow$  GETPRINTER),  
 GETPRINTER = (printer.get  $\rightarrow$  copy  $\rightarrow$  scanner.put  $\rightarrow$  printer.put  $\rightarrow$  Q  
 | timeout  $\rightarrow$  scanner.put  $\rightarrow$  Q).

Problem: Zwar kein Deadlock aber  
 kein echter Fortschritt, bei  
 dem irgendwann "copy"  
 passiert, möglich!  
 $\rightarrow$  LTSA Animation

## 6.3 Die dinierenden Philosophen (Dijkstra)

Fünf Philosophen sitzen um einen runden Tisch. Jeder Philosoph verbringt sein Leben indem er abwechselnd denkt oder isst. Zum Essen benötigt er zwei Gabeln. Zwischen zwei Philosophen liegt jeweils eine Gabel. Die Philosophen haben vereinbart, dass sie nur die beiden Gabeln neben ihrem Teller benutzen.



## Modellierung in FSP

PHIL = (think → right.get → left.get → eat → right.put → left.put → PHIL).

FORK = (get → put → FORK).

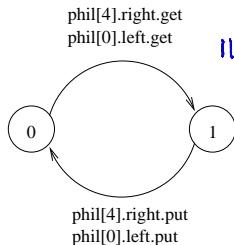
|| DINERS = forall[i:0..4] phil[i]:PHIL

|| forall[i:0..4] {phil[i].right, phil[(i+1)%5].left}::FORK).

*(phil[0]:PHIL || ... || phil[4]:PHIL)*

LTS für Gabel 4:

*{phil[0].right, phil[1].left} :: FORK*



*|| ...*

*{phil[4].right, phil[0].left} :: FORK*

Ablauf zu Deadlock im LTS von DINERS:

phil[0].think

phil[0].right.get

phil[1].think

phil[1].right.get

phil[2].think

phil[2].right.get

phil[3].think

phil[3].right.get

phil[4].think

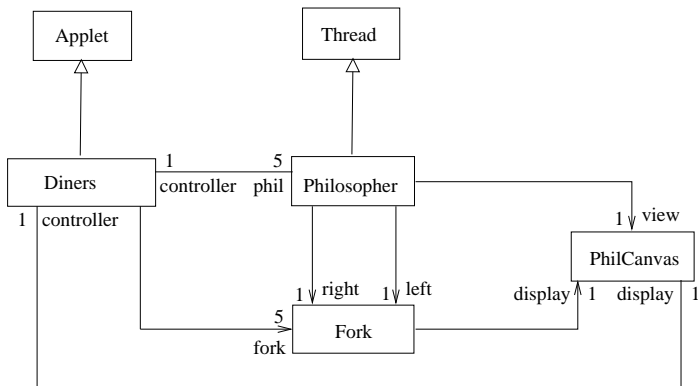
phil[4].right.get

DEADLOCK ! Kein left.get mehr möglich.

**Beachte:** Alle 4 Kriterien für Deadlock sind hier erfüllt (Ressource = Gabel)

## Simulation in Java

Philosophen werden durch Threads realisiert, Gabeln durch Monitore.



```
class Philosopher extends Thread {
    private Fork left, right;
    private PhilCanvas view;
    private Diners controller;
    private int identity;
    Philosopher(Diners ctr, int id, Fork l, Fork r) {
        controller = ctr; view = ctr.display;
        identity = id; left = l; right = r;
    }
    public void run() {
        try {
            while(true) {
                // thinking
                view.setPhil(identity,view.THINKING);
                Thread.sleep(controller.sleepTime());
                // hungry
                view.setPhil(identity,view.HUNGRY);
                right.get();
                // gotright fork
                view.setPhil(identity,view.GOTRIGHT);
                Thread.sleep(500);
                left.get():
                // eating
                view.setPhil(identity,view.EATING);
                Thread.sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (InterruptedException e) {}    }}
```

```

class Fork {
    private boolean taken = false; // Zustand der Gabel
    private PhilCanvas display;
    private int identity;
    Fork(PhilCanvas disp, int id) {
        display = disp; identity = id;
    }
    synchronized void get() throws InterruptedException {
        while (taken) wait();
        taken = true;
        display.setFork(identity,taken);
    }
    synchronized void put() {
        taken = false;
        display.setFork(identity,taken);
        notify();
    }
}

```

← notify nicht nötig,  
da kein Philosoph  
darauf warten wird,  
dass die Gabel genommen  
wird

while (! taken) wait ();  
hier weggelassen, da jedes Philosoph  
put nur aufruft, wenn er vorher  
get aufgerufen hat.

```
class Diners extends Applet {
    PhilCanvas display;
    Thread[] phil= new Thread[5];
    Fork[] fork = new Fork[5];
    ...
    public void start() {
        for (int i = 0; i < 5; ++i)
            fork[i] = new Fork(display,i);
        for (int i = 0; i < 5; ++i) {
            phil[i] = new Philosophier(this,i,
                fork[(i+4)%5],
                fork[i]);
            phil[i].start();
        }
    }
}
```

**Bemerkung:**

Das mögliche Deadlock ist im Programm nur schwer erkennbar!  
Modellierung ist wichtig!



## Deadlockfreie Philosophen

Es gibt verschiedene Möglichkeiten:

1. Nicht alle Philosophen nehmen die Gabeln in der gleichen Reihenfolge (Auflösen des "wait-for cycles").

Hier: Philosophen mit ungerader Nummer nehmen die linke Gabel zuerst:

```
PHIL(i=0) = (when (i%2 == 0) think → right.get → left.get → eat →
             right.put → left.put → PHIL
            | when (i%2 == 1) think → left.get → right.get → eat →
             right.put → left.put → PHIL).
```

```
|| DINERS = (phil[i:0..4]:PHIL(i) ||
             forall[i:0..4] {phil[i].right, phil[(i+1)%5].left}::FORK).
```

2. "right.get" und "left.get" zu einer atomaren Aktion zusammenfassen mit Hilfe einer Sperre.
3. Maximal 4 Philosophen dürfen an den Tisch (d.h. "right.get" hintereinander ausführen).

## Deadlockfreie Java Implementierung

```
class Philosopher extends Thread {
    private Fork left, right;
    private PhilCanvas view;
    private Diners controller;
    private int identity;

    Philosopher(Diners ctr, int id, Fork l, Fork r) {
        controller = ctr; view = ctr.display;
        identity = id; left = l; right = r;
    }

    public void run() {
        try {
            while(true) {
                // thinking
                view.setPhil(identity,view.THINKING);
                Thread.sleep(controller.sleepTime());
                // hungry
                view.setPhil(identity,view.HUNGRY);
            }
        }
    }
}
```

```
//get forks
if (identity%2 == 0) {
    right.get();
    // gotright fork
    view.setPhil(identity,view.GOTRIGHT);
}
else {
    left.get();
    // gotleft fork
    view.setPhil(identity,view.GOTLEFT);
}
Thread.sleep(500);
if (identity%2 == 0) {
    left.get();
}
else {
    right.get();
}
// eating
view.setPhil(identity,view.EATING);
Thread.sleep(controller.eatTime());
right.put();
left.put();
}
} catch (InterruptedException e) {}
}
```