Kapitel 6

Verklemmungen (Deadlocks)

Prof. Dr. Rolf Hennicker

16.06.2016

6.1 Charakterisierung von Deadlocks

genauer: BlockedByMonitor oder Waiting

Eine Verklemmung entsteht, wenn alle Prozesse in einem System blockiert (d.h. im Zustand "Non-Runnable") sind. In diesem Fall sind keine weiteren Aktionen möglich.

Notwendige und hinreichende Bedingungen für Deadlocks:
(nach Coffmann et al. 1971) 1. ∧ 2. ∧ 3. ∧ 4. ⇔ Es gibt 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"
   Es gibt eine zyklische Folge von Prozessen, so dass jeder Prozess eine Ressource belegt, auf die sein Nachfolger wartet.

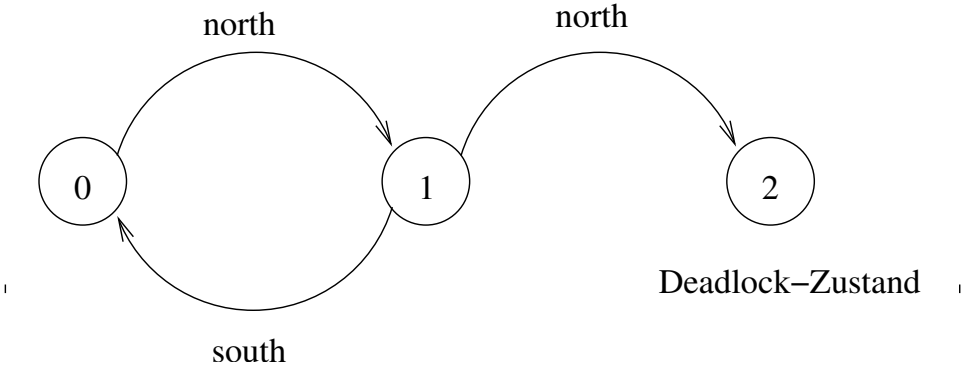
Deadlock Vermeidung: Eine der 4 Bedingungen verletzen.
1. + 2. kann meist nicht vermieden werden.
Vermeidung von 3. kann problematisch sein. Verstoße 4. zu vermeiden!

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 → (south → MOVE | north → STOP)).



Ablauf zu Deadlock:

north
north

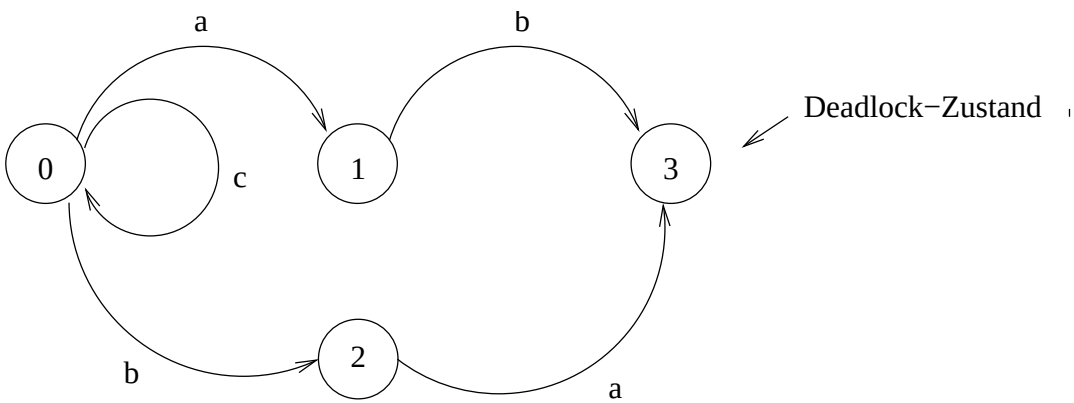
Bemerkung: Es gibt auch den konstanten FSP-Ausdruck END zur Modellierung von gewünschter Terminierung.

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:

$$A = (a \rightarrow north \rightarrow south \rightarrow A \mid c \rightarrow A).$$
$$B = (b \rightarrow south \rightarrow north \rightarrow B \mid c \rightarrow B).$$
$$||AB = (A||B).$$



Stark äquivalent ist:

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

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.

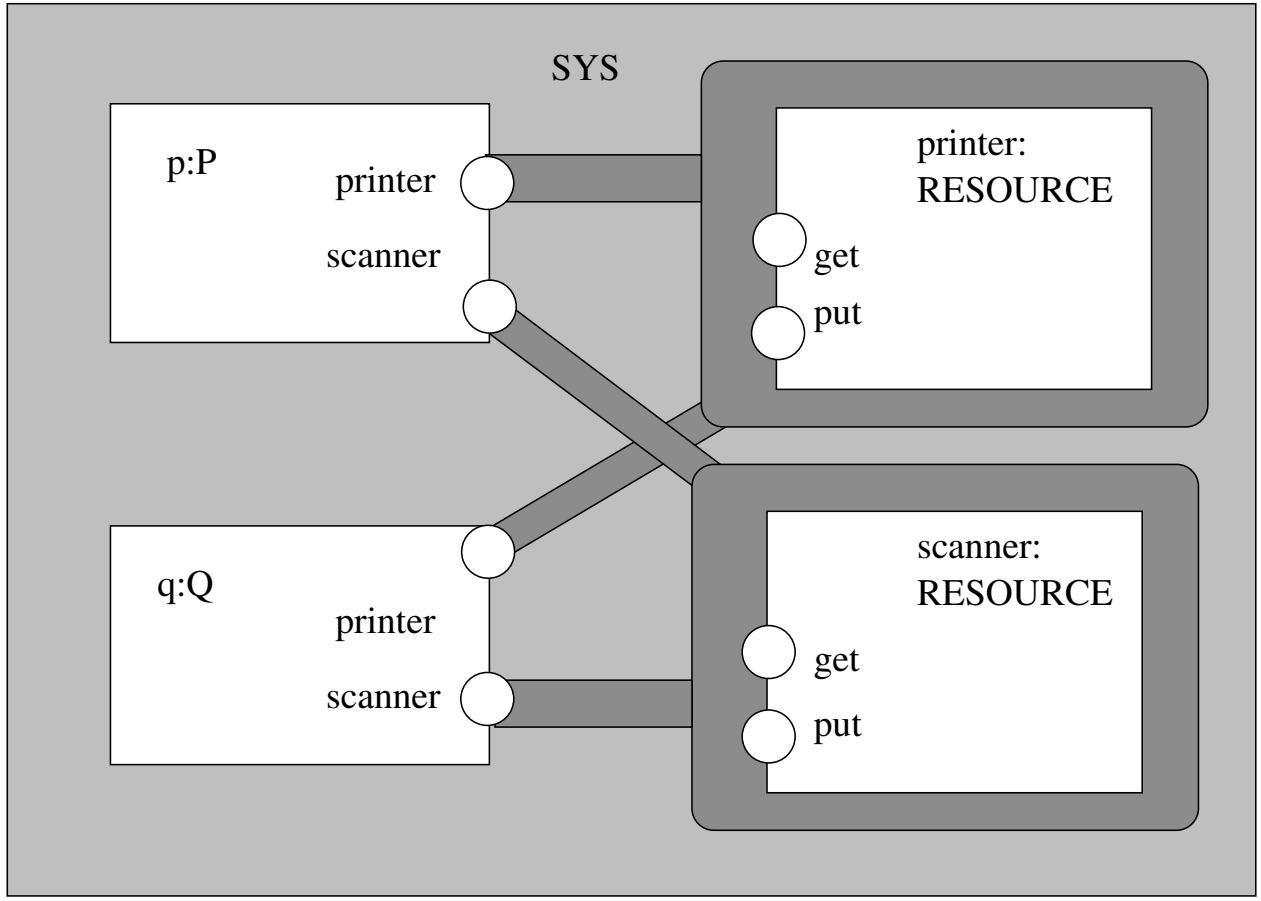
Bspl.: Betrachte den Prozess AB.
Beginne beim Anfangszustand ⓪. Kein Deadlock.

1. ⓪ —a→ ①: kein Deadlock; merke ① für weit. Weitersuche.
2. ① —c→ ⓪: kein Deadlock; ⓪ nicht merken, da schon besucht.
3. ⓪ —b→ ②: kein Deadlock; merke ② für weit. Weitersuche.

1.1 ① —b→ ③: Deadlock gefunden
Ein minimaler Ablauf dorthin ist a b

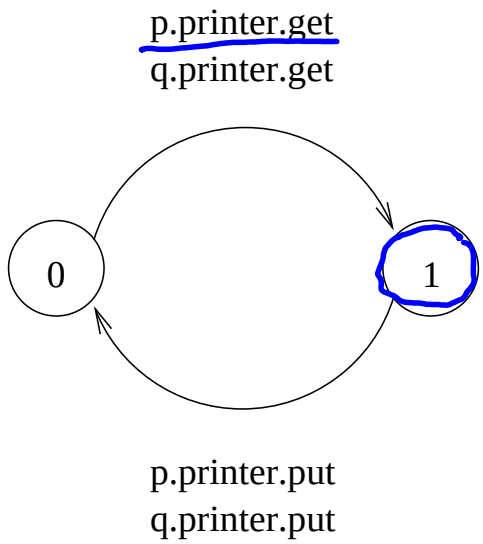
Beispiel (Printer & Scanner):

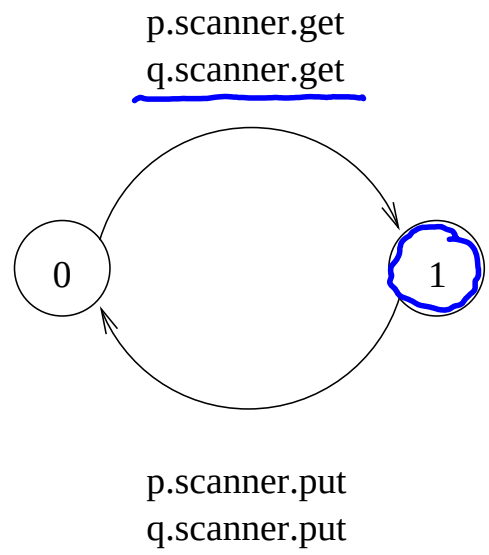
Zwei Prozesse p:P und q:Q brauchen jeweils einen Printer und einen Scanner zum Kopieren.



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

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





Abläufe zu Deadlock:

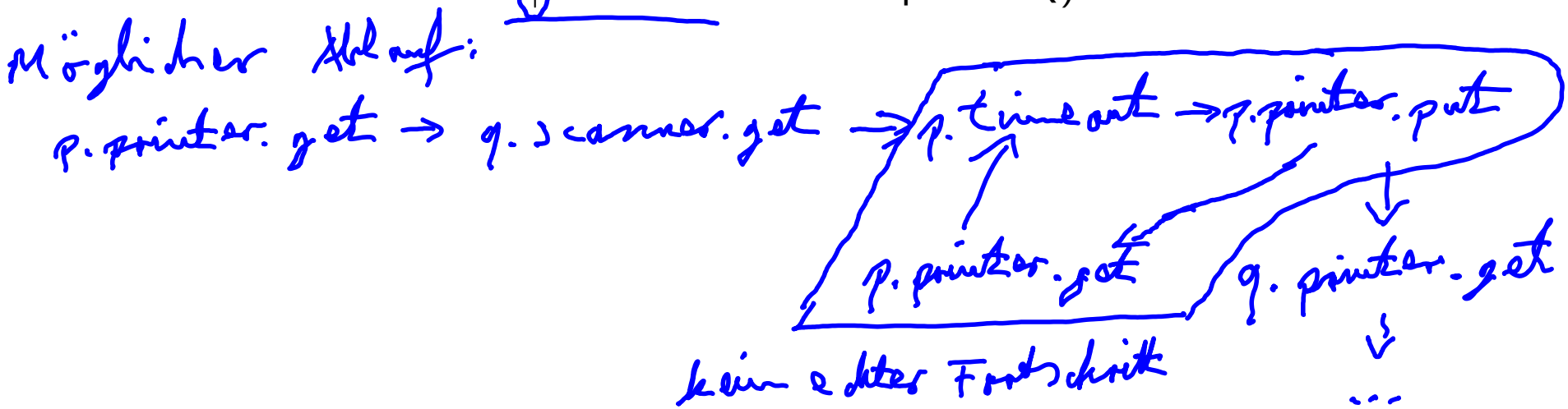
p.printer.get	oder	q.scanner.get
q.scanner.get		p.printer.get

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!

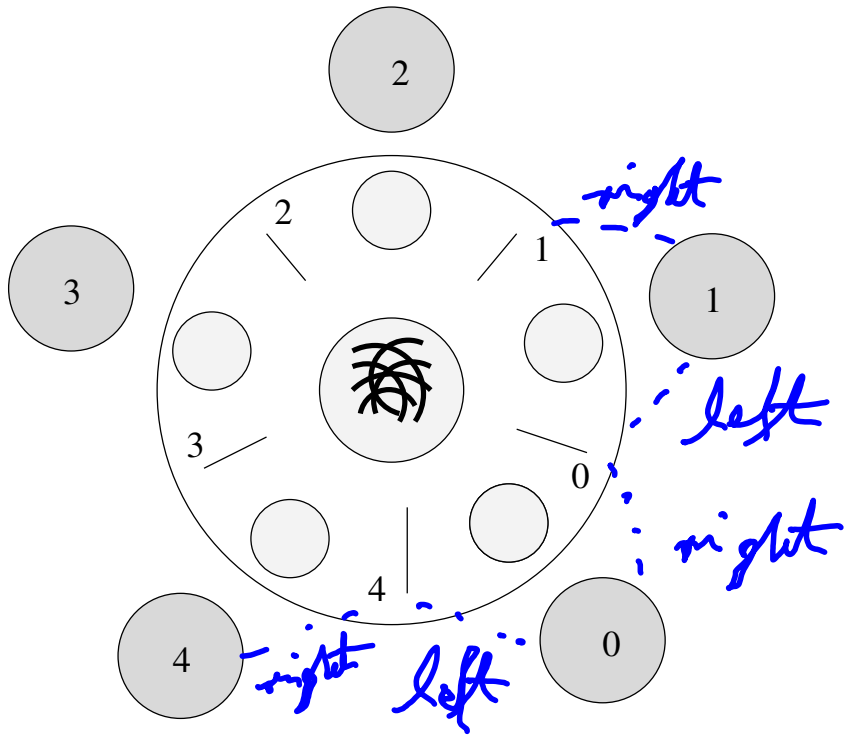
$$
\begin{aligned}
P &= (\text{printer.get} \rightarrow \text{GETSCANNER}),\\
\text{GETSCANNER} &= (\text{scanner.get} \rightarrow \text{copy} \rightarrow \text{printer.put} \rightarrow \text{scanner.put} \rightarrow P\\
&\quad \mid \text{timeout} \rightarrow \text{printer.put} \rightarrow P).
\end{aligned}
$$

$$
\begin{aligned}
Q &= (\text{scanner.get} \rightarrow \text{GETPRINTER}),\\
\text{GETPRINTER} &= (\text{printer.get} \rightarrow \text{copy} \rightarrow \text{scanner.put} \rightarrow \text{printer.put} \rightarrow Q\\
&\quad \mid \text{timeout} \rightarrow \text{scanner.put} \rightarrow Q).
\end{aligned}
$$



6.3 Die dinierenden Philosophen

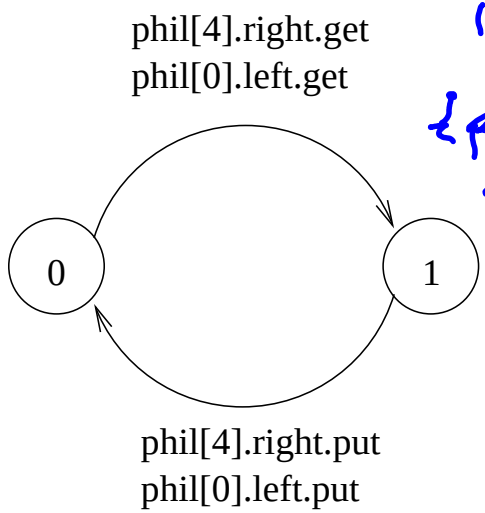
Fünf Philosophen sitzen um einen runden Tisch. Jeder Philosoph verbringt sein Leben indem er abwechselnd denkt oder isst. Zum Essen benötigt er zwei Gabeln. Zwischen zwei Philosophen liegt jeweils eine Gabel. Die Philosophen haben vereinbart, dass sie nur die beiden Gabeln neben ihrem Teller benützen.



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).

LTS für Gabel 4:



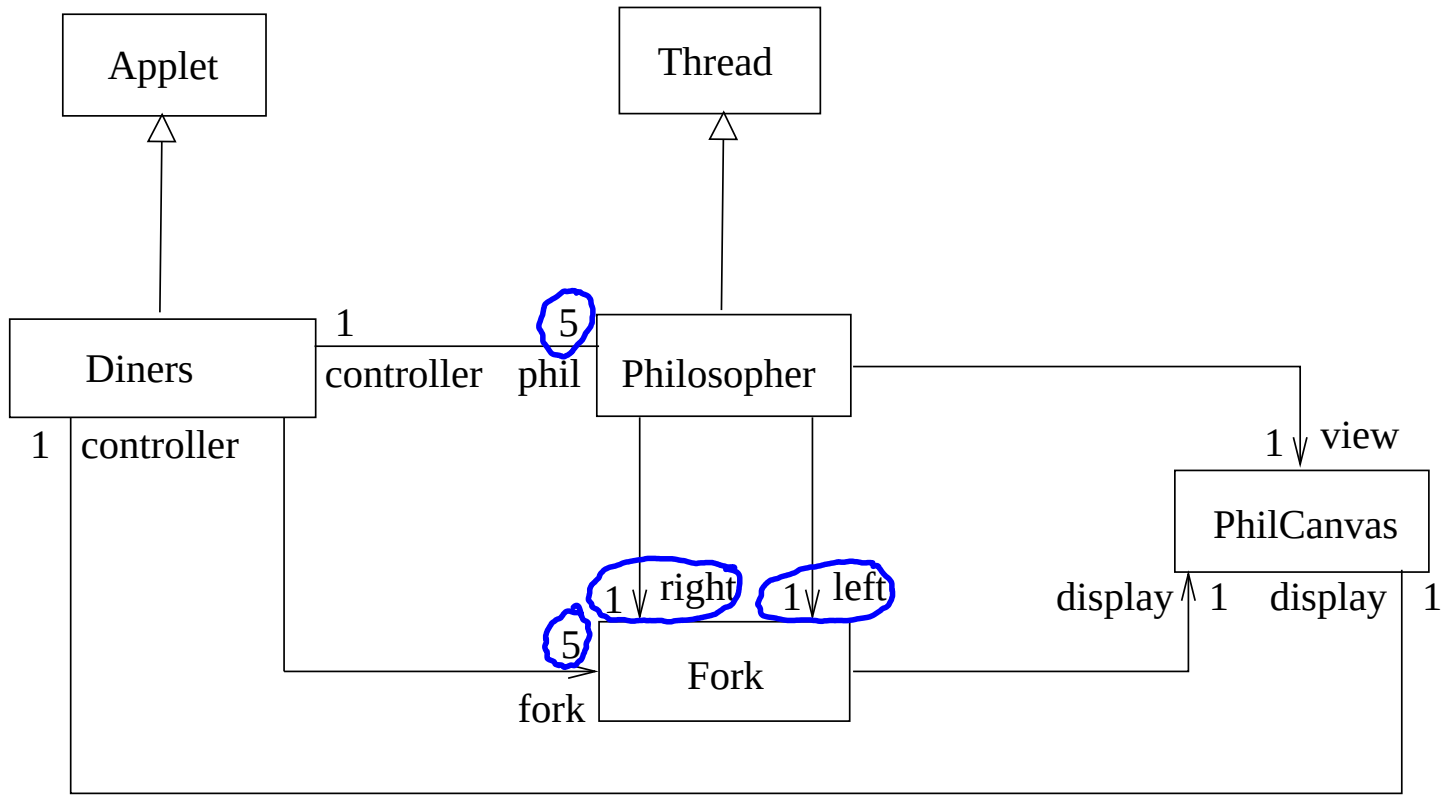
Ablauf zu Deadlock im LTS von DINERS:

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) {}   }}
```

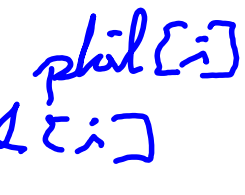
2 Monitore

```
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 hier nicht nötig, da kein Philosoph darauf warten wird, dass die Gabel genommen wurde.

while (!taken) wait();
hier nicht nötig, da jeder Philosoph "put" nur dann 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 Philosopher(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:

   $$\begin{aligned}\text{PHIL}(I{=}0) = (&\text{when } (I\%2 == 0)\ \text{think} \rightarrow \text{right.get} \rightarrow \text{left.get} \rightarrow \text{eat} \rightarrow \\ &\text{right.put} \rightarrow \text{left.put} \rightarrow \text{PHIL} \\ &|\ \text{when } (I\%2 == 1)\ \text{think} \rightarrow \text{left.get} \rightarrow \text{right.get} \rightarrow \text{eat} \rightarrow \\ &\text{right.put} \rightarrow \text{left.put} \rightarrow \text{PHIL}).\end{aligned}$$

   $$\begin{aligned}\|\text{DINERS} = (&\text{phil[i:0..4]:PHIL}(i) \ \| \\ &\text{forall[i:0..4]}\ \{\text{phil[i].right, phil[(i+1)\%5].left}\}::\text{FORK}).\end{aligned}$$

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) {}
    }
}
```

i gerade

i ungerade