

Entwurf und Implementierung paralleler Programme

Prof. Dr. Rolf Hennicker

25.06.2009

Kapitel 6

Verklemmungen

(Deadlocks)

6.1 Charakterisierung von Deadlocks

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. "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"

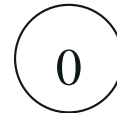
Prozesse können nicht "gezwungen" werden, eine Ressource wieder freizugeben.

4. "Wait for Cycle"

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

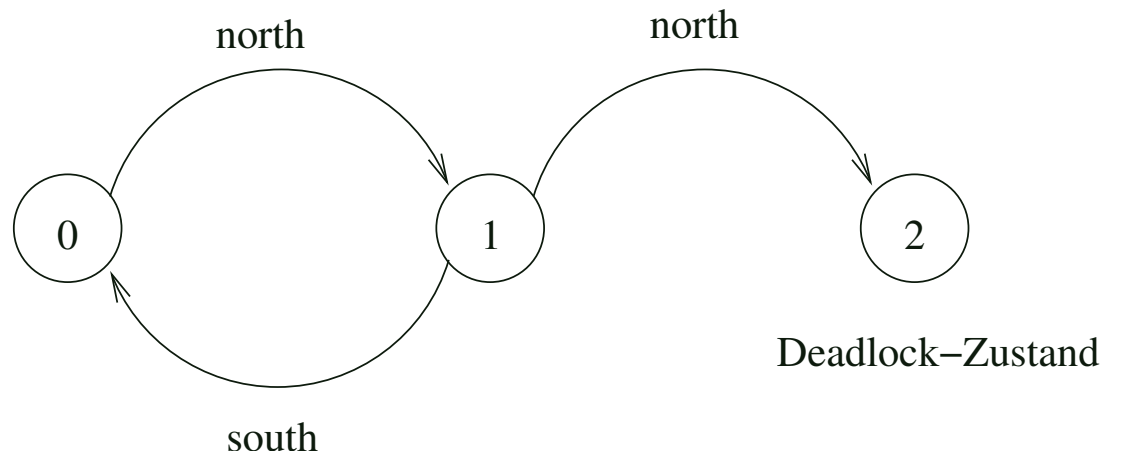
6.2 Deadlock-Analyse in FSP

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

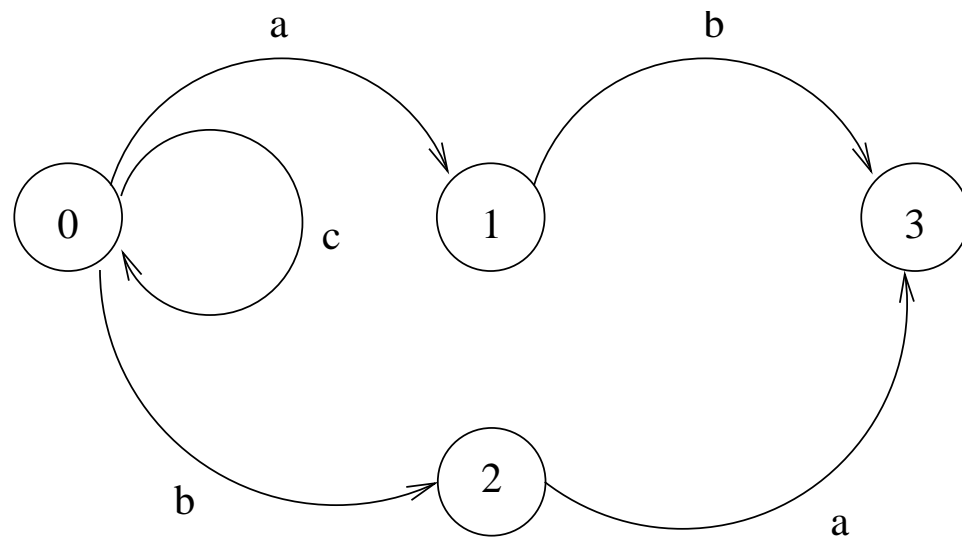
Implizite Deadlock-Zustände

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:

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

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

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


Deadlock-Zustand

Äquivalent ist:

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

Auffinden von Deadlock-Zuständen

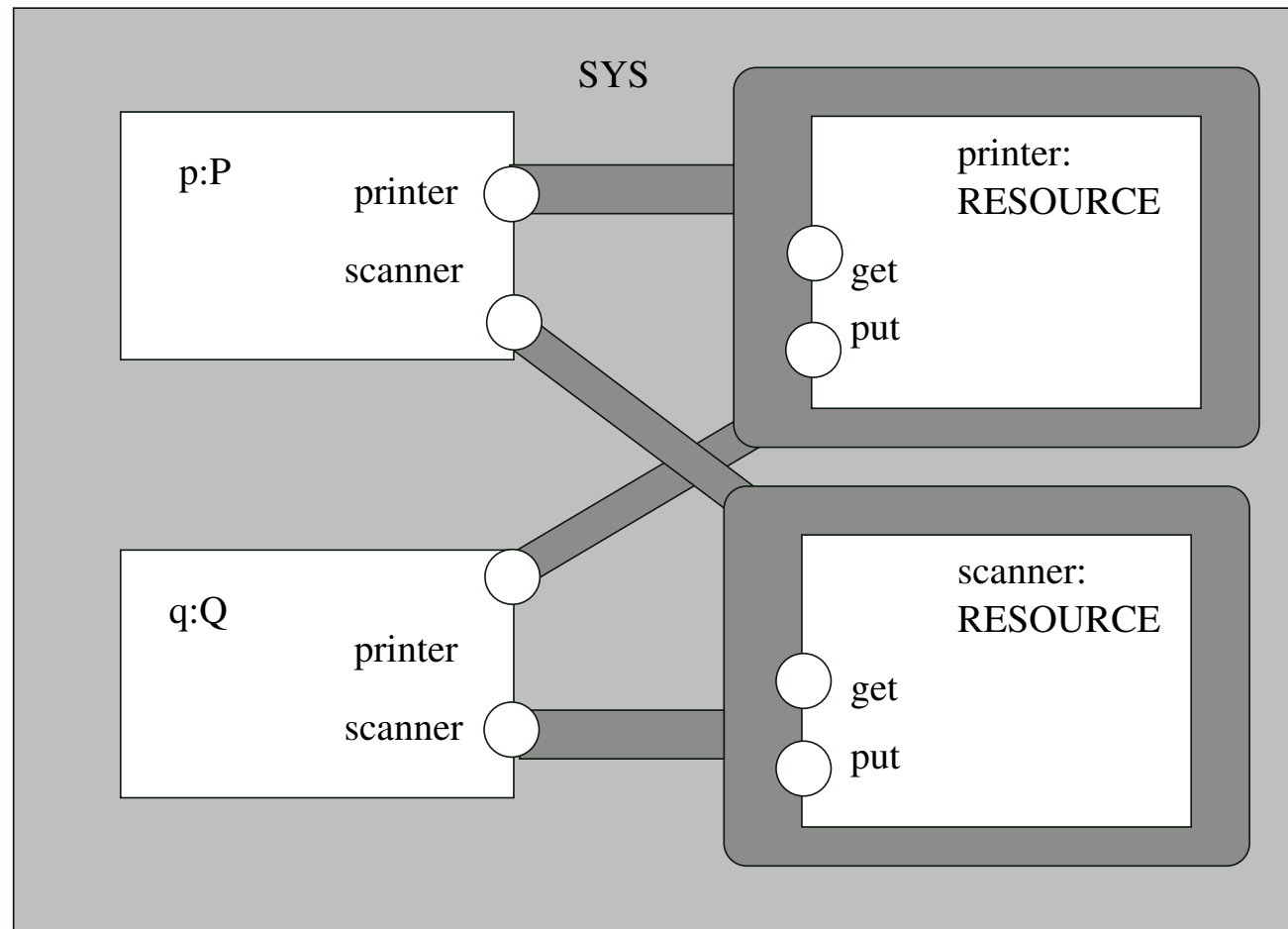
Zum Auffinden von Deadlock-Zuständen *und* von hinführenden Abläufen (Traces) kann eine Breitensuche (“breadth-first search”) ausgehend vom Anfangszustand durchgeführt werden. Falls ein Deadlock existiert, erhält man einen minimalen Weg dorthin.

Beachte:

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

Beispiel (Printer & Scanner):

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



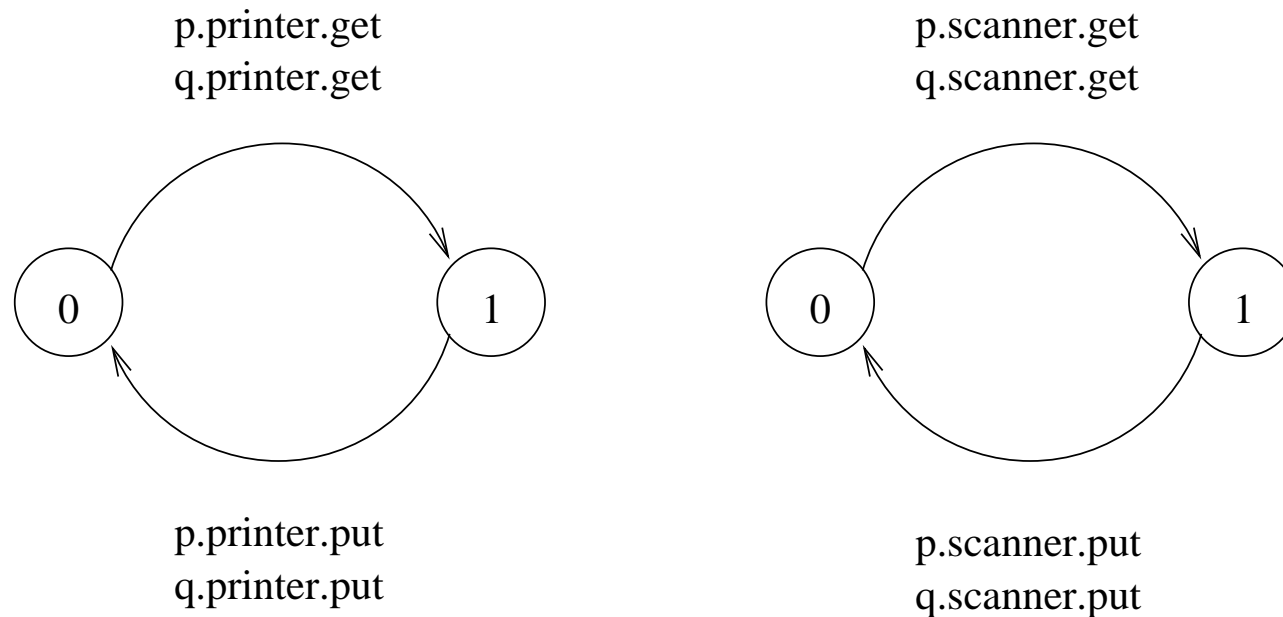
RESOURCE = (get \rightarrow put \rightarrow RESOURCE).

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

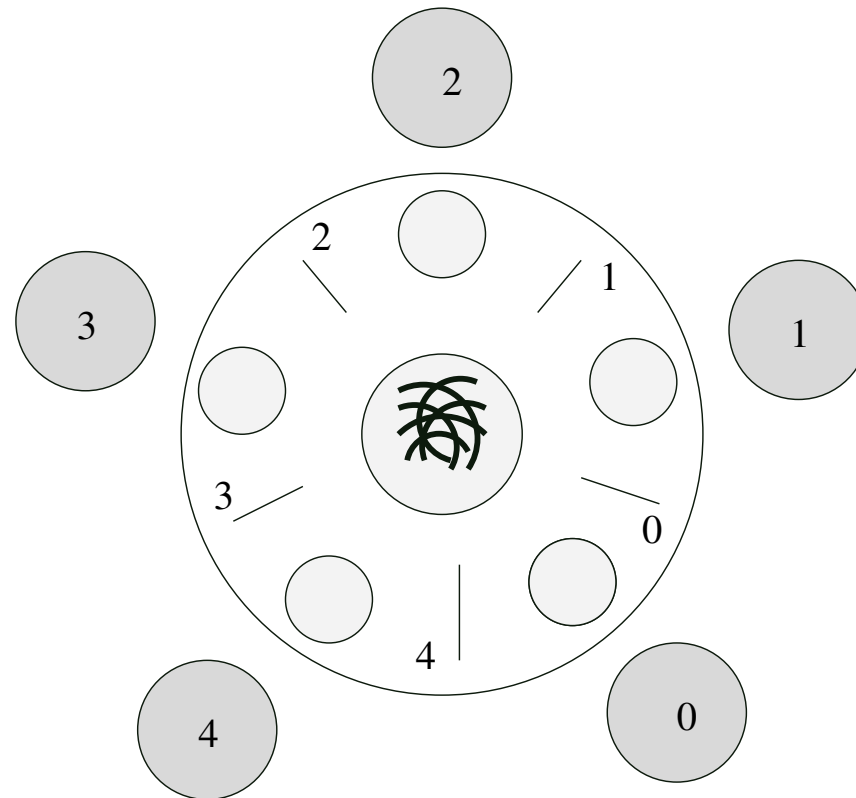
Möglichkeiten, um Deadlock zu vermeiden:

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!

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

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.



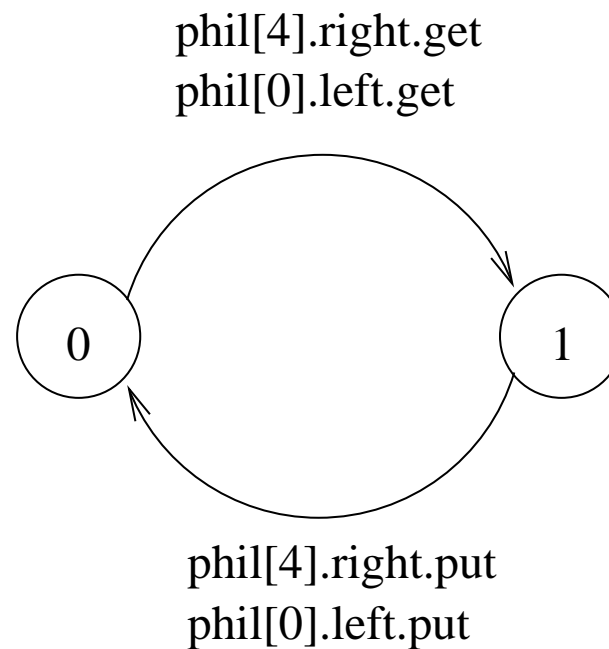
Modellierung in FSP

PHIL = (think \rightarrow right.get \rightarrow left.get \rightarrow eat \rightarrow right.put \rightarrow left.put \rightarrow PHIL).

FORK = (get \rightarrow put \rightarrow 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:



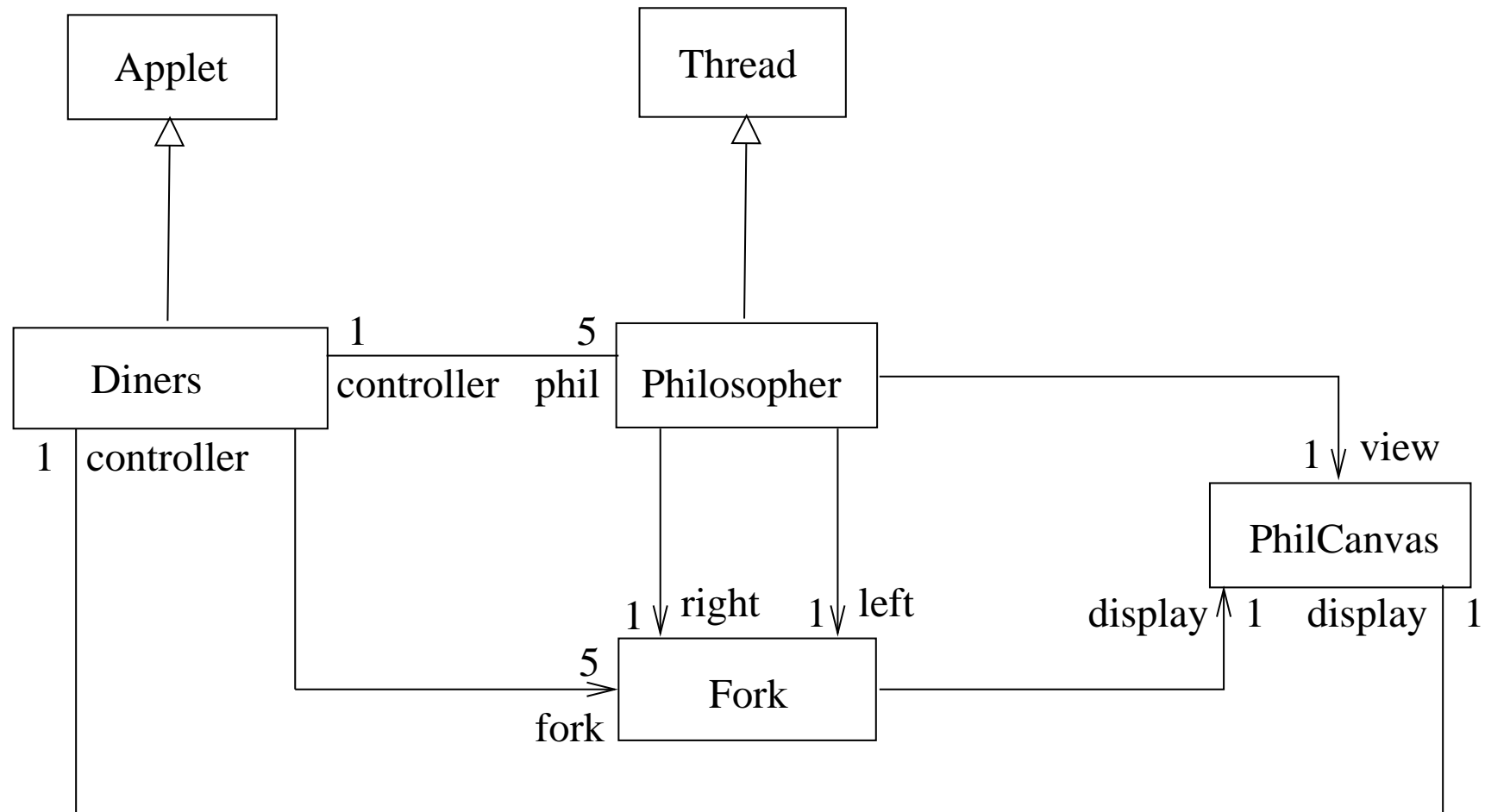
Ablauf zu Deadlock im LTS von DINERS:

```
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) {}
}
}
```

```
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();
    }
}
```



```
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.

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:

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

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

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) {}  
}  
}
```