

# Deadlock



# Deadlock

**Concepts:** system **deadlock:** no further progress  
four necessary & sufficient conditions

**Models:** deadlock - no eligible actions

**Practice:** blocked threads

**Aim:** deadlock avoidance - to design systems where deadlock cannot occur.

## Deadlock: four necessary and sufficient conditions

◆ **Serially reusable resources:**

*the processes involved share resources which they use under mutual exclusion.*

◆ **Incremental acquisition:**

*processes hold on to resources already allocated to them while waiting to acquire additional resources.*

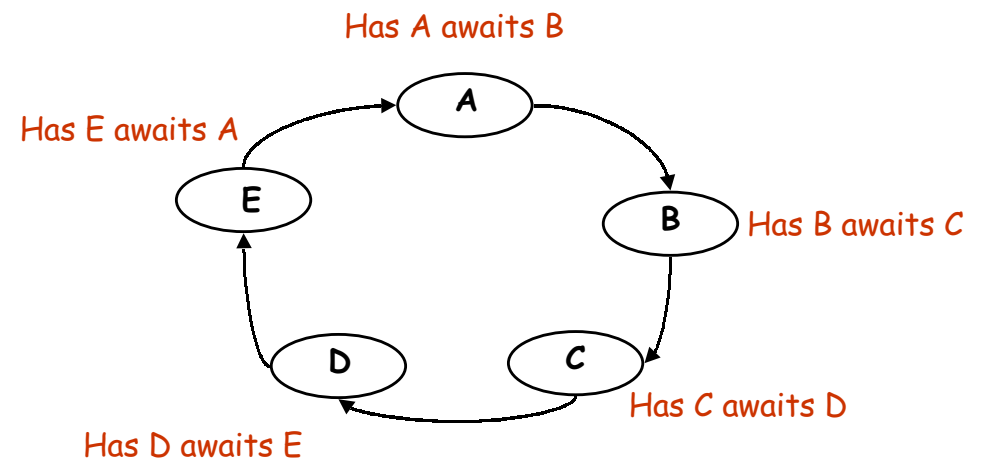
◆ **No pre-emption:**

*once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.*

◆ **Wait-for cycle:**

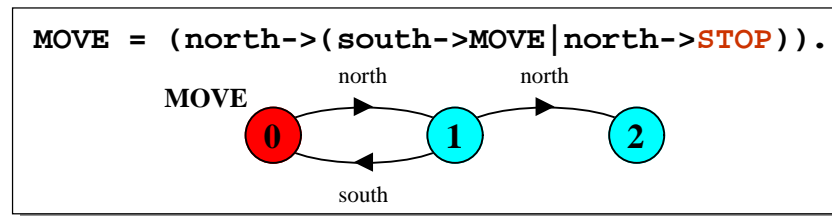
*a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.*

## Wait-for cycle



## 6.1 Deadlock analysis - primitive processes

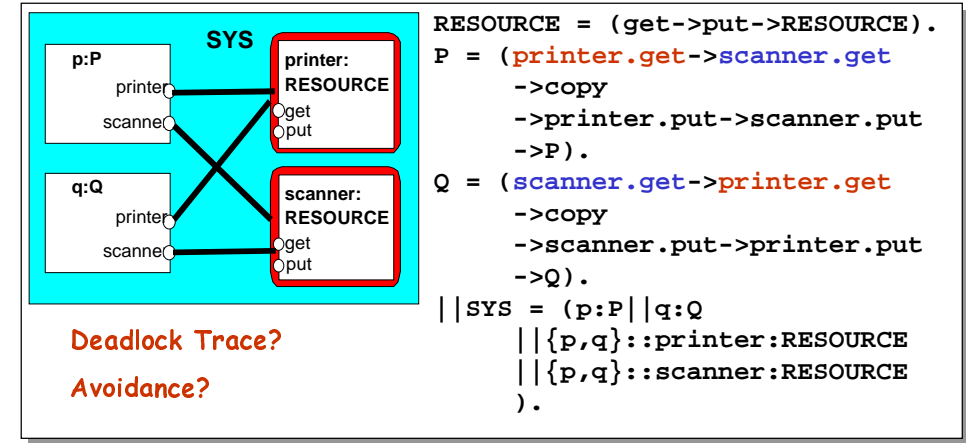
- ♦ deadlocked state is one with **no outgoing transitions**
- ♦ in FSP: **STOP** process



- ♦ animation to produce a trace.
- ♦ analysis using *LTSA*: Trace to **DEADLOCK**:  
(shortest trace to **STOP**)  
north  
north

## deadlock analysis - parallel composition

- ♦ in systems, deadlock may arise from the **parallel composition** of interacting processes.



## deadlock analysis - avoidance

- ♦ **acquire resources in the same order?**
- ♦ **Timeout:**

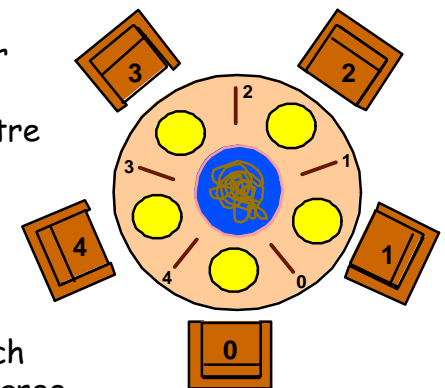
```
P = (printer.get-> GETSCANNER),  
GETSCANNER = (scanner.get->copy->printer.put  
->scanner.put->P  
| timeout -> printer.put->P  
).
```

```
Q = (scanner.get-> GETPRINTER),  
GETPRINTER = (printer.get->copy->printer.put  
->scanner.put->Q  
| timeout -> scanner.put->Q  
).
```

**Deadlock? Progress?**

## 6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately **thinking** and **eating**. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

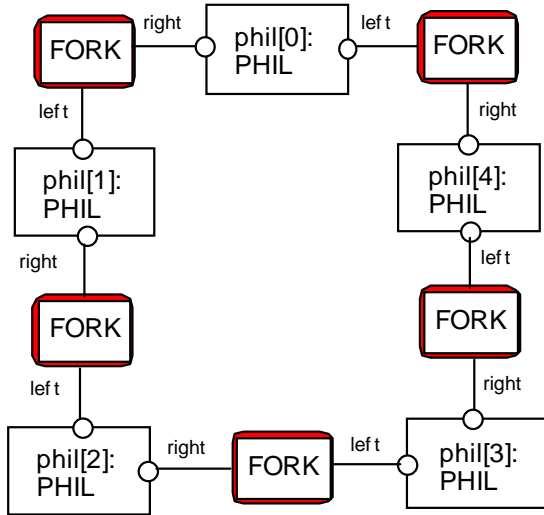


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

## Dining Philosophers - model structure diagram

Each FORK is a **shared resource** with actions **get** and **put**.

When hungry, each PHIL must first get his right and left forks before he can start eating.



Concurrency: Deadlock

9

©Magee/Kramer

## Dining Philosophers - model

```

FORK = (get -> put -> FORK).
PHIL = (sitdown ->right.get->left.get
        ->eat ->right.put->left.put
        ->arise->PHIL).
    
```

Table of philosophers:

```

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

*Can this system deadlock?*

Concurrency: Deadlock

10

©Magee/Kramer

## Dining Philosophers - model analysis

**Trace to DEADLOCK:**  
 phil.0.sitdown  
 phil.0.right.get  
 phil.1.sitdown  
 phil.1.right.get  
 phil.2.sitdown  
 phil.2.right.get  
 phil.3.sitdown  
 phil.3.right.get  
 phil.4.sitdown  
 phil.4.right.get

This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his **right**.

The system can make no further progress since each philosopher is waiting for a fork held by his neighbor i.e. a **wait-for cycle** exists!

Concurrency: Deadlock

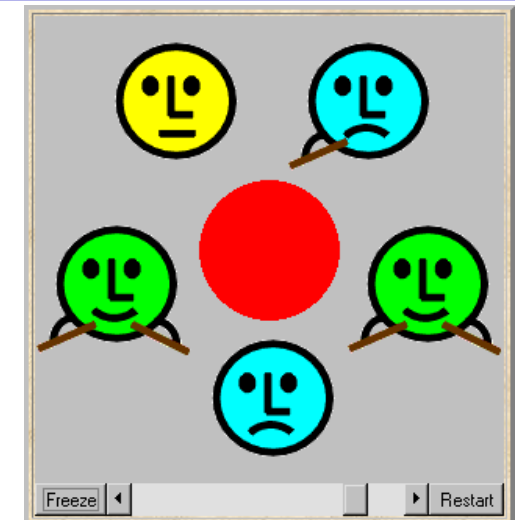
11

©Magee/Kramer

## Dining Philosophers

Deadlock is easily detected in our **model**.

*How easy is it to detect a potential deadlock in an implementation?*

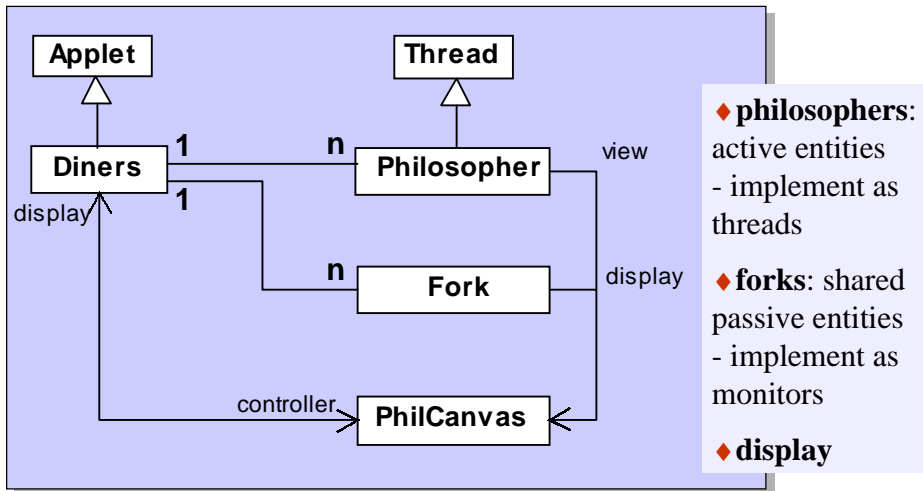


Concurrency: Deadlock

12

©Magee/Kramer

## Dining Philosophers - implementation in Java



Concurrency: Deadlock

13

©Magee/Kramer

## Dining Philosophers - Fork monitor

```
class Fork {
    private boolean taken=false;
    private PhilCanvas display;
    private int identity;

    Fork(PhilCanvas disp, int id)
        { display = disp; identity = id;}

    synchronized void put() {
        taken=false;
        display.setFork(identity,taken);
        notify();
    }

    synchronized void get()
        throws java.lang.InterruptedExcepion {
        while (taken) wait();
        taken=true;
        display.setFork(identity,taken);
    }
}
```

**taken**  
encodes the  
state of the  
fork

## Dining Philosophers - Philosopher implementation

```
class Philosopher extends Thread {
    ...
    public void run() {
        try {
            while (true) {
                view.setPhil(identity,view.THINKING);
                sleep(controller.sleepTime()); //hungry
                view.setPhil(identity,view.HUNGRY);
                right.get(); //gotright chopstick
                view.setPhil(identity,view.GOTRIGHT);
                sleep(500);
                left.get(); //eating
                view.setPhil(identity,view.EATING);
                sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (java.lang.InterruptedExcepion e){}
    }
}
```

Follows  
from the  
model  
(sitting  
down and  
leaving the  
table have  
been  
omitted).

## Dining Philosophers - implementation in Java

Code to create the philosopher  
threads and fork monitors:

```
for (int i =0; i<N; ++i)
    fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
    phil[i] =
        new Philosopher
            (this,i,fork[(i-1+N)%N],fork[i]);
    phil[i].start();
}
```

Concurrency: Deadlock

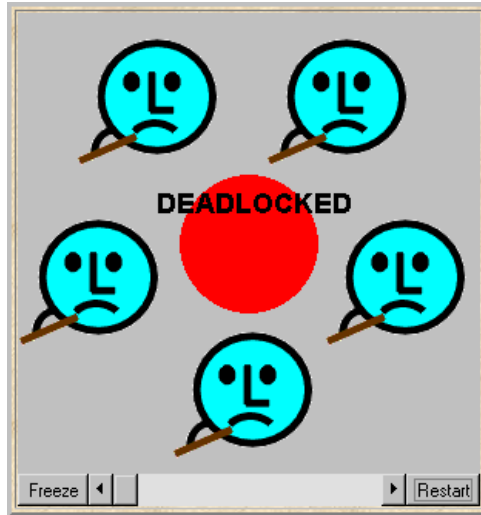
16

©Magee/Kramer

## Dining Philosophers

To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating.

This "speedup" increases the probability of deadlock occurring.



17

©Magee/Kramer

## Deadlock-free Philosophers

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist. *How?*

Introduce an *asymmetry* into our definition of philosophers.

Use the identity  $I$  of a philosopher to make *even* numbered philosophers get their *left* forks first, *odd* their *right* first.

*Other strategies?*

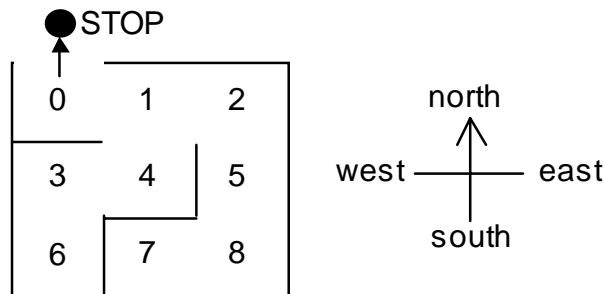
```
PHIL(I=0)
  = (when (I%2==0) sitdown
      ->left.get->right.get
      ->eat
      ->left.put->right.put
      ->arise->PHIL
    |when (I%2==1) sitdown
      ->right.get->left.get
      ->eat
      ->left.put->right.put
      ->arise->PHIL
  ).
```

18

©Magee/Kramer

## Maze example - shortest path to "deadlock"

We can exploit the shortest path trace produced by the deadlock detection mechanism of *L TSA* to find the *shortest path out of a maze* to the **STOP** process!



We must first model the **MAZE**.

Each position can be modelled by the moves that it permits. The **MAZE** parameter gives the starting position.

eg.  $\text{MAZE}(\text{Start}=8) = \text{P}[\text{Start}],$   
 $\text{P}[0] = (\text{north} \rightarrow \text{STOP} \mid \text{east} \rightarrow \text{P}[1]), \dots$

Concurrency: Deadlock

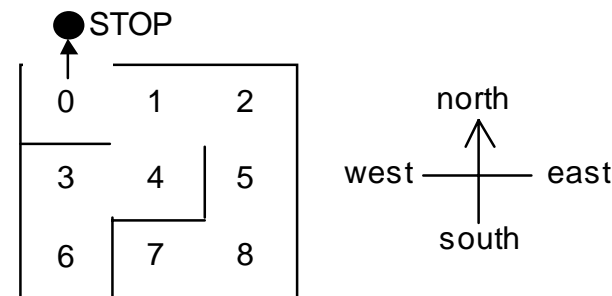
19

©Magee/Kramer

## Maze example - shortest path to "deadlock"

`||GETOUT = MAZE(7).`

Shortest path escape trace from position 7?



Trace to **DEADLOCK**:

east  
north  
north  
west  
west  
north

Concurrency: Deadlock

20

©Magee/Kramer

## Summary

---

### ◆ Concepts

- **deadlock**: no further progress
- four necessary and sufficient conditions:
  - ◆ serially reusable resources
  - ◆ incremental acquisition
  - ◆ no preemption
  - ◆ wait-for cycle

**Aim:** deadlock avoidance  
- to design systems where  
deadlock cannot occur.

### ◆ Models

- no eligible actions (analysis gives shortest path trace)

### ◆ Practice

- blocked threads