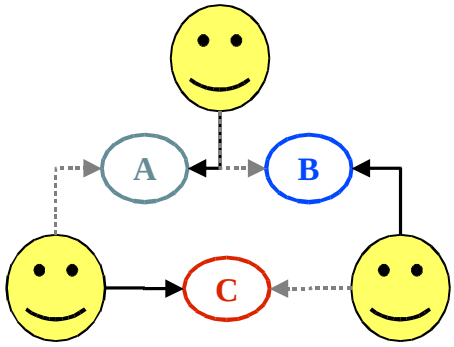
Concurrent Programming 19530-V (WS01)

Lecture 7: Deadlock

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Scenario

Process 1 gets the lock for object **A** and wants to lock object **B**



Process 2 gets the lock for object **C** and wants the lock for object **A**

Process 3 gets the lock for object **B** and wants the lock for object **C**

What happens next?

Deadlock Concept

When all processes in a system are waiting to acquire a shared resource (i.e., all of the processes are blocked), then the system is ***deadlocked***.

- When a system is deadlocked, it is not possible to execute any actions or make any progress
  - Each process is waiting for a resource to be released, but no process can make progress to release a held resource
- Deadlock is a serious issue in concurrent systems
- The goal is to design systems that are free from deadlock

Necessary Condition for Deadlock

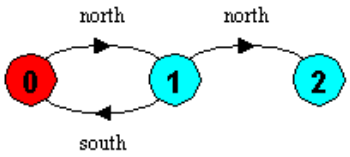
- *Serially reusable resources*
  - The processes involved share resources which they use under mutual exclusion.
- *Incremental acquisition*
  - Processes hold on to resources already allocated to them while waiting to acquire additional resources.
- *No preemption*
  - Once acquired by a process, resources cannot be preempted (forcibly withdrawn) but are only released voluntarily.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.

Primitive Deadlock Analysis

In an LTS graph deadlock is easily visible as a state with no outgoing arcs.

In FSP we can achieve deadlock using the **STOP** process:

```
MOVE = (north->(south->MOVE|north->STOP)).
```

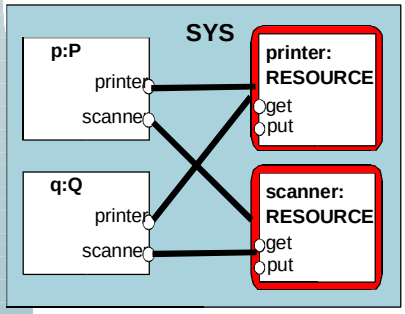


Using the LTSA, we can find the deadlock through safety analysis:

```
Trace to DEADLOCK:
     north
     north
```

Parallel Process Deadlock Analysis

In real systems, deadlock may arise from the parallel composition of interacting processes.



Deadlock Trace?

Avoidance?

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

Deadlock Avoidance

One potential technique for avoiding deadlock

If processes share different classes of resources, such as printers and scanners, a general purpose deadlock avoidance strategy is to order the resource classes so every process acquires them in the same order.

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

Deadlock Avoidance

Another potential technique for avoiding deadlock

It is also possible to use time-out values to avoid deadlock.

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

Deadlock in Java

Similar example in Java but without using locks

```
// Global space
Semaphore scanner = new Semaphore(1);
Semaphore printer = new Semaphore(1);
```

```
// Thread 1
public void run() {
  printer.down();

  scanner.down();

  // do work here…
  scanner.up();
  printer.up();
}
```

← Get printer

Get scanner →

← Try scanner

Try printer →

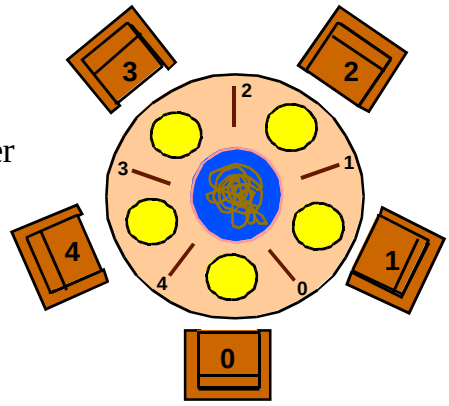
```
// Thread 2
public void run() {
  scanner.down();

  printer.down();

  // do work here…
  printer.up();
  scanner.up();
}
```

Deadlock in Java

- Why does deadlock happen in this example?
  - Because of how semaphores are implemented
    - The down() method of each semaphore will wait() until the semaphore value is non-zero
    - Since the two threads acquire the semaphores in the opposite order, it is possible to interleave their instructions such that one thread gets the scanner and one gets the printer and then they both must wait for each other to finish…which will never happen
  - This scenario can be resolved just like the model, use resource acquisition ordering

Dining Philosophers Example

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

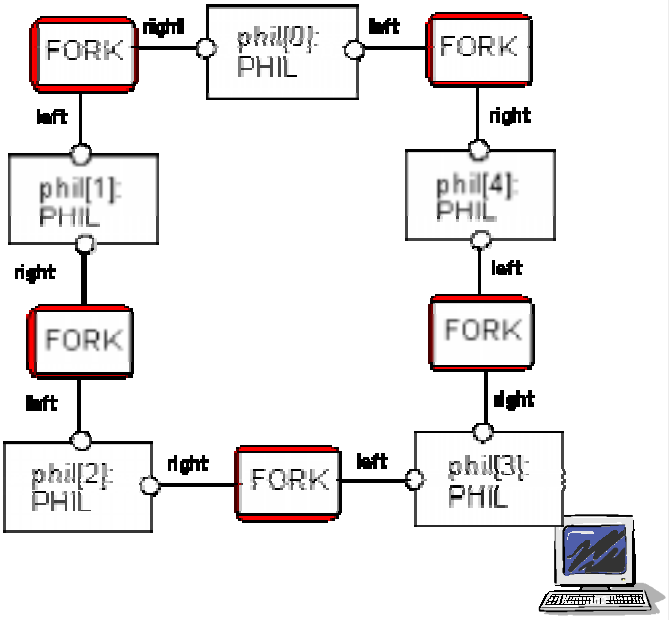


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

Dining Philosophers Structure Diagram

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

When hungry, each **`PHIL`** must first get his right fork and then his left fork before he can start eating.



Dining Philosopher 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?

Dining Philosopher 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
```

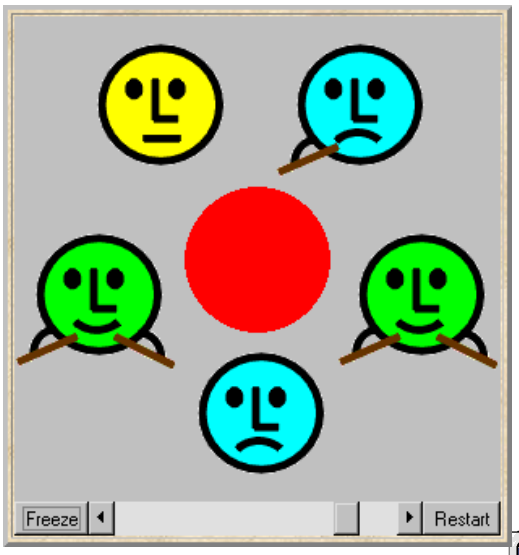
This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his **right**.

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!

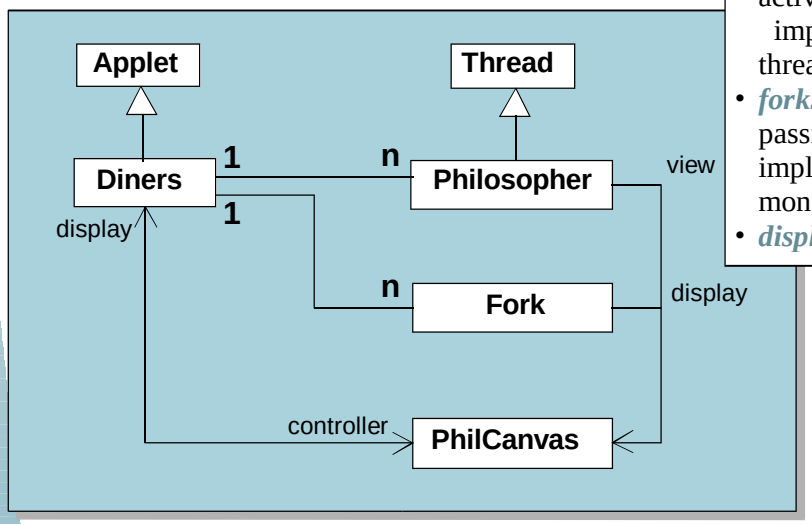
Dining Philosophers Implementation

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

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



Dining Philosophers Implementation



- ***philosophers***: active entities, implement as threads
- ***forks***: shared passive entities, implement as monitors
- ***display***

Dining Philosophers Implementation

```
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.InterruptedException {
    while (taken) wait();
    taken=true;
    display.setFork(identity,taken);
  }
}
```

taken encodes the state of the fork

Dining Philosophers Implementation

```
class Philosopher extends Thread {
  ...
  public void run() {
    try {
      while (true) {                    // thinking
        view.setPhil(identity,view.THINKING);
        sleep(controller.sleepTime()); // hungry
        view.setPhil(identity,view.HUNGRY);
        right.get();                    // got right fork
        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.InterruptedException e){}
  }
}
```

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

Dining Philosophers Implementation

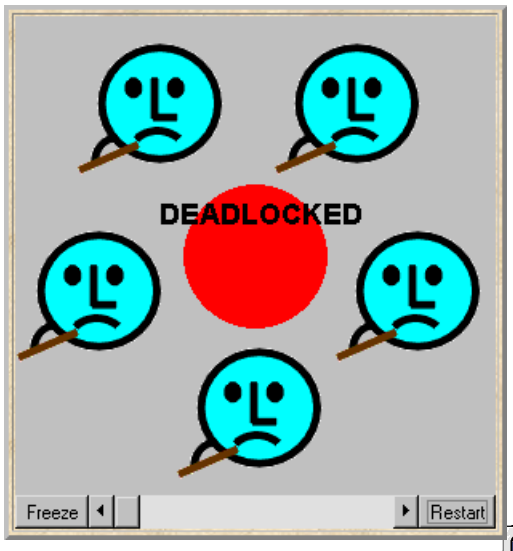
Code to create the philosopher threads and fork monitors

```
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();
}
```

Deadlock in Implementation

To ensure that 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.



Fixing the Model

We can fix the implementation by eliminating the *wait-for* cycle...
How?

Introduce an ***asymmetry*** into our definition of philosophers.

Use the identity of each 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
    ).
```

Deadlock in Real Systems

- Deadlock avoidance heuristics from this lecture
  - Ordered acquisition of different resources types
  - Asymmetric acquisition of same resource types
- Both of these heuristics apply to real world systems, but following them alone will not always lead to deadlock free systems
  - Sometimes *wait-for* cycles result from unexpected dependencies and circumstances of the system and environment's implementation
    - Requires testing and diligence