

Deadlock

INF2140 Parallel Programming: Lecture 6

March 07, 2012

Deadlock

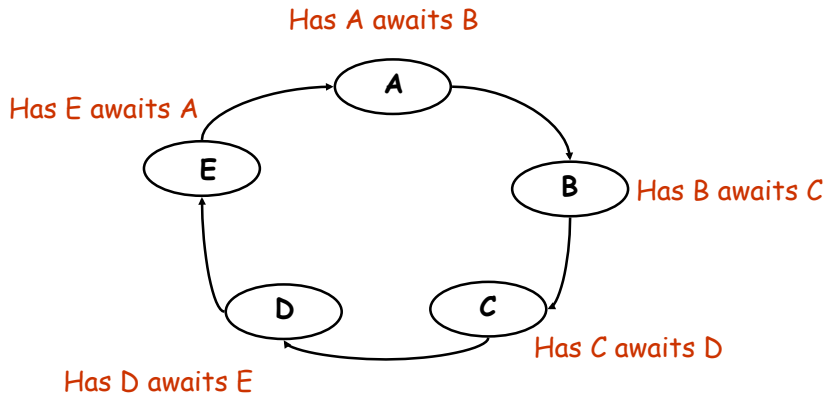
- **Concepts**
 - System deadlock: no further progress
 - Four necessary & sufficient conditions
- **Models**
 - Deadlock - no eligible actions
- **Practice**
 - Blocked threads

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once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.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

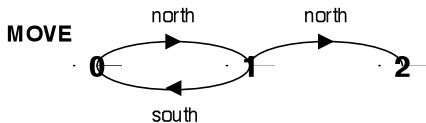
Wait-for Cycle



Deadlock Analysis - Primitive Processes

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

$\text{MOVE} = (\text{north} \rightarrow (\text{south} \rightarrow \text{MOVE} \mid \text{north} \rightarrow \text{STOP}))$.

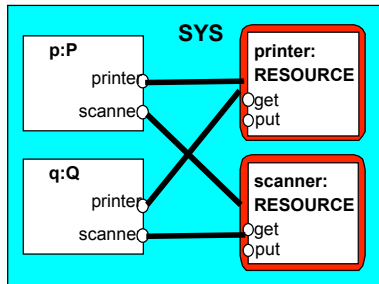


- Animation to produce a trace.
- Analysis using LTSA:
(shortest trace to STOP)

Trace to DEADLOCK:
north
north

Deadlock Analysis - Parallel Composition

- In composed systems, deadlock may arise from the *parallel composition* of interacting processes.



Deadlock Trace?
Avoidance?

```
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  
).
```

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
              ).
```

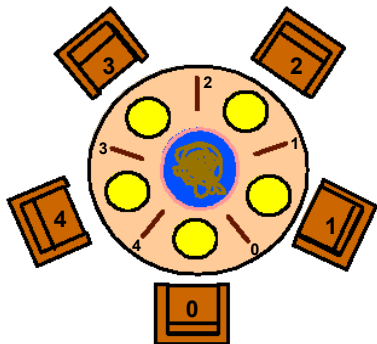
Deadlock?

Progress?

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.

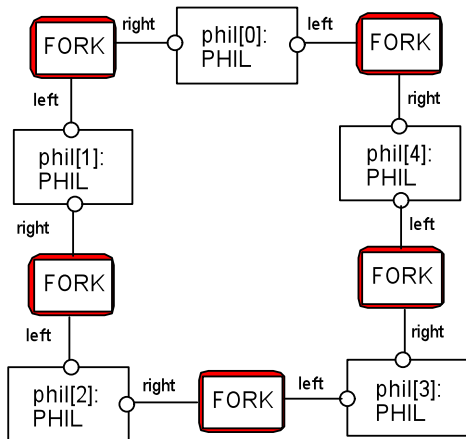
One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.



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.



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?

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
```

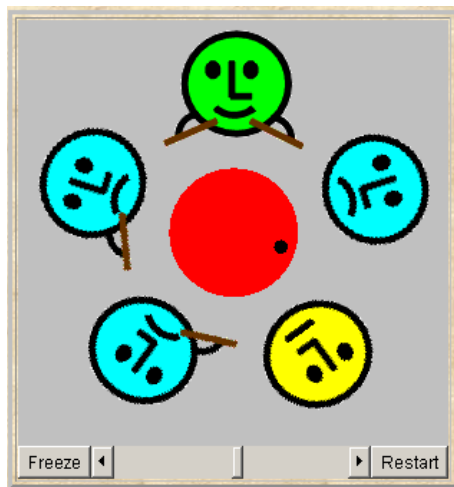
This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his **right**.

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!

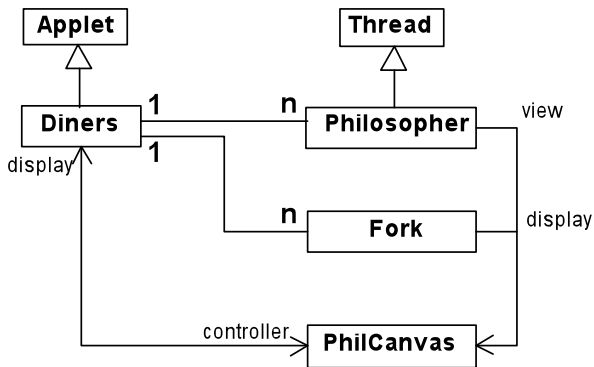
Dining Philosophers

Deadlock is easily detected in our **model** using **LTSA**

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



Dining Philosophers - Implementation in Java



- **Philosophers:** active entities —implement as *threads*
- **Forks:** shared passive entities —implement as *monitors*
- **Display**

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

taken:
encodes the
state of the
fork

Dining Philosophers - Philosopher implementation

```
class Philosopher extends Thread {
... // Follows from the model (we omit sitting down
public void run() {           // and leaving the table)
    try {
        while (true) { // thinking
            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.InterruptedException e){}}}
```

Dining Philosophers - implementation in Java

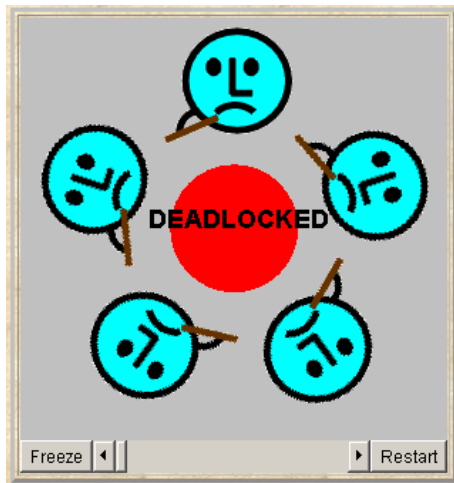
Code to create the philosopher threads and fork monitors:

```
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();
}
```


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.



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.

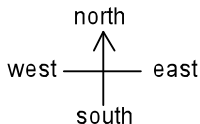
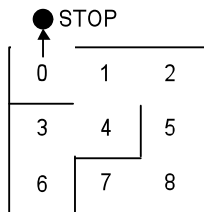
Use the identity I of a philosopher to make **even** numbered philosophers get their **left** forks first, **odd** their **right** first.

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
).
```

Maze Example - Shortest Path to “Deadlock”

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



We first model the MAZE.

Each position is modelled by the moves that it permits. The MAZE parameter gives the starting position.

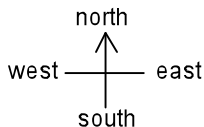
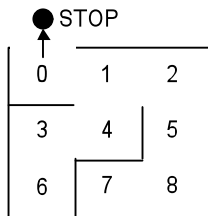
One example:

$\text{MAZE}(\text{Start}=8) = P[\text{Start}],$

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

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

Summary

- Concepts
 - Deadlock: no further progress
 - Four necessary and sufficient conditions:
 - Serially reusable resources
 - Incremental acquisition
 - No preemption
 - Wait-for cycle
- Models
 - No eligible actions (analysis gives shortest path trace)
- Practice
 - Blocked threads

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