Deadlocks
(second hour of Week 7, after midterm)

Abhik Roychoudhury
CS 3211
National University of Singapore

Reading material: Chapter 6 of Textbook.

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

- A. Serially reusable resources:

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

- B. Incremental acquisition:

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

- C. No pre-emption:

once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

- D. 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.

A. Serially re-usable resources

A monitor encapsulates resources which are accessed using mutual exclusion

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  synchronized public void up() {
     ++value;
     notify();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value== 0) wait();
    --value;
  }
}
```


B. Incremental acquisition

Nested Monitors --- Implement a bounded buffer as a monitor. Use semaphores (another monitor) to control access when buffer is full or empty.

```
class SemaBuffer implements Buffer {
  …
  Semaphore full;  //counts number of items
  Semaphore empty; //counts number of spaces

  SemaBuffer(int size) {
    this.size = size; buf = new Object[size];
    full = new Semaphore(0);
    empty= new Semaphore(size);
  }
…
}
```


Nested monitors – Incr. acquisition

```
synchronized public void put(Object o)
              throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)%size;
   full.up();
 }
 synchronized public Object get()
              throws InterruptedException{
   full.down();
   Object o =buf[out]; buf[out]=null;
   --count; out=(out+1)%size;
   empty.up();
   return (o);
 }
```

```
synchronized public void down()
     throws InterruptedException {
   while (value== 0) wait();
   --value;
 }
```

What was the deadlock scenario?

Initially buffer does not contain anything,
Integer protected by semaphore *full* is 0,
And integer protected by semaphore *empty* is non-zero

Consumer executes get()

Inside get(), the first line is full.down()

Inside down, the first line is
while (value == 0) wait()
// value is the integer protected by the semaphore monitor

Since *full* is 0, wait() is executed
Since wait() is encountered in a method for the *full* semaphore –
it releases the lock for *full*

The lock for the buffer whose get() called full.down() is not released!!

Deadlock scenario in this case

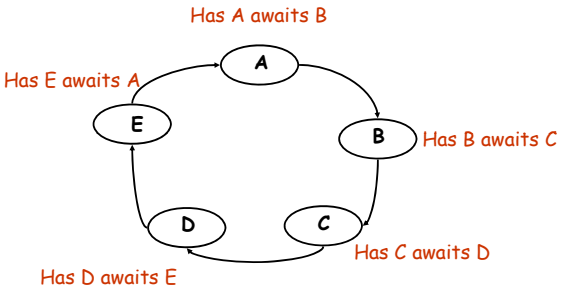
A. *Serially re-usable resource*: the buffer for example

B. *Incremental acquisition of resources*: acquire the lock to the buffer, and wait for items to be placed in the buffer.

C. ***No pre-emption***: All resources are released voluntarily.

D. *Circular Wait*: The producer is waiting for the lock to the buffer to be released, so that it can insert items. The consumer is waiting for the items to be inserted, so that it can consume and release the lock to the buffer.

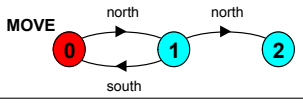
D. Wait-for cycle



6.1 Deadlock analysis - primitive processes

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

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



- animation to produce a trace.
- analysis using *LTSA*:

(shortest trace to **STOP**)

Trace to DEADLOCK:
north
north

A state with no outgoing actions

Note that such a "deadlocked" state will be obvious only when we construct the state model.
--- What is a ``deadlocked" state?
--- A state with no outgoing actions.

Moreover, even if such a deadlocked state exists, it may a state in the global state model.

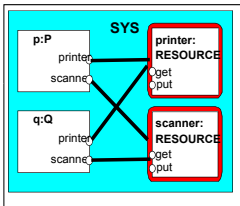
Suppose Sys = P1 || P2

There might be no deadlocked state in the state models of P1, P2.

But in the state model of Sys, we can encounter deadlocked states.

deadlock analysis - parallel composition

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



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 Trace?
Avoidance?

Deadlock Trace

```
p.printer.get
q.scanner.get
```

The problem meets all the four conditions of deadlock

A. Serial re-use: The printer and scanner are serially re-used.

B. Incremental acquisition: each process holds on to acquired resource (scanner/printer), while waiting for the other resource (printer/scanner).

C. No pre-emption: All resources are released voluntarily.

D. Wait for cycle: Process p has printer, waits for scanner from q
Process q has scanner, waits for printer from p.

deadlock analysis - avoidance

- all processes acquire resources in the same order.
- Introduce Timeouts:

```
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?

Deadlock avoidance - timeouts

- B. Incremental acquisition:

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

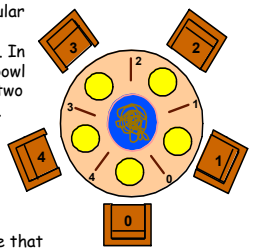
Having timeouts --- violates the above condition for deadlock, thereby avoiding deadlock.

Violates progress

Acquire the first resource, Fail to acquire second resource, Timeout ... (repeated forever).

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.

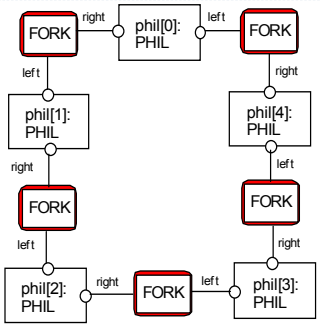


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

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.



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?

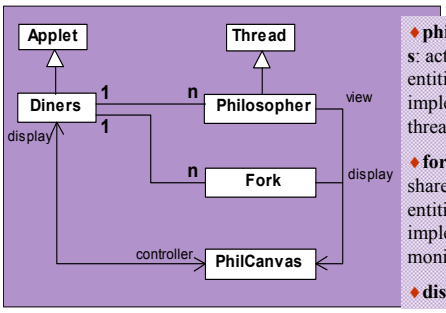
Dining Philosophers - model analysis

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 neighbour i.e. a ***wait-for cycle*** exists!

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 {
  ...
  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 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();
}
```


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.

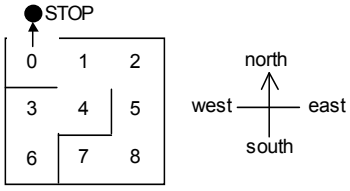
Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

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" (in tutorials next week)

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



We must first model the **MAZE**.

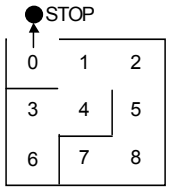
Each position can be modelled by the moves that it permits. The **MAZE** parameter gives the starting position.

eg. **MAZE(Start) = P[Start],**
P[0] = (north->STOP|east->P[1]),...

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 futher 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.