Deadlocks

Abhik Roychoudhury
CS 3211
National University of Singapore

Modified from Kramer and Magee’s lecture notes.
Reading material: Chapter 6 of Textbook.

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  once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.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

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.

```java
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

```java
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);
}
```
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(). 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

A state with no outgoing actions

Note that such a "deadlocked" state will be obvious only when we construct the state model. Moreover, even if such a deadlocked state exists, it may be 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.

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)

deadlock analysis - parallel composition

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

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),

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.

Table of philosophers:

| DINERS(N=5) = forall [i:0..N-1] (phi[i].PHIL | {phi[i].left,phi[(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

For (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        fork[i][j] = new Fork(display, identity, i, j);

for (int i = 0; i < N; ++i)
    Philosopher phi = new Philosopher(controller, identity, i, fork[i]);
    phi.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=8) = P[Start],
P[0] = (north->STOP|east->P[1]),...

| 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.