Deadlock

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 5828 — Lecture 21 — 03/30/2010
© University of Colorado, 2010
The remainder of the slides in this presentation were created by Magee and Kramer for Chapter 6 of the Concurrency textbook.
Chapter 6

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once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.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

Has A awaits B

Has E awaits A

Has D awaits E

Has B awaits C

Has C awaits D
6.1 Deadlock analysis - primitive processes

♦ 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**: Trace to **DEADLOCK**:

  (shortest trace to **STOP**)

  north

  north

Concurrency: Deadlock
deadlock analysis - parallel composition

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

\[
\text{RESOURCE} = (\text{get} \rightarrow \text{put} \rightarrow \text{RESOURCE}).
\]

\[
P = (\text{printer.get} \rightarrow \text{scanner.get}
\rightarrow \text{copy}
\rightarrow \text{printer.put} \rightarrow \text{scanner.put}
\rightarrow P).
\]

\[
Q = (\text{scanner.get} \rightarrow \text{printer.get}
\rightarrow \text{copy}
\rightarrow \text{scanner.put} \rightarrow \text{printer.put}
\rightarrow Q).
\]

\[
||\text{SYS} = (p:P || q:Q

\|\{p,q\}:printer:RESOURCE
\|\{p,q\}:scanner:RESOURCE

).
\]
deadlock analysis - avoidance

♦ acquire resources in the same order?

♦ Timeout:

\[
\begin{align*}
P &= \text{(printer.get->} \text{GETSCANNER)}, \\
\text{GETSCANNER} &= \text{(scanner.get->copy->printer.put} \\
&\quad \text{->scanner.put->P} \\
&\quad \mid \text{timeout} \to \text{printer.put->P} \\
Q &= \text{(scanner.get->} \text{GETPRINTER)}, \\
\text{GETPRINTER} &= \text{(printer.get->copy->printer.put} \\
&\quad \text{->scanner.put->Q} \\
&\quad \mid \text{timeout} \to \text{scanner.put->Q}
\end{align*}
\]

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.
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

\[
FORK = (\text{get} \rightarrow \text{put} \rightarrow \text{FORK}). \\
PHIL = (\text{sitdown} \rightarrow \text{right.get} \rightarrow \text{left.get} \rightarrow \text{right.put} \rightarrow \text{left.put} \rightarrow \text{arise} \rightarrow \text{PHIL}).
\]

Table of philosophers:

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

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!
Deadlock is easily detected in our model.

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

Concurrent dining philosophers are implemented in Java by creating threads for each philosopher and using monitors to control access to forks. Each philosopher can hold one fork at a time, and the forks are shared among all philosophers. The display is updated by the applet to reflect the state of the philosophers.
Dining Philosophers - Fork monitor

```java
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.InterruptedIOException {
        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); // 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.InterruptedIOException e) {} }
Dining Philosophers - implementation in Java

Code to create the philosopher threads and fork monitors:

```java
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?*

```plaintext
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!

eg. MAZE(Start=8) = P[Start],
    P[0] = (north->STOP|east->P[1]),...
Maze example - shortest path to “deadlock”

\[ ||\text{GETOUT} = \text{MAZE}(7). \]

Shortest path escape trace from position 7?

Trace to DEADLOCK:
- east
- north
- north
- west
- west
- north

Concurrent: Deadlock
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.