CSC 4103 - Operating Systems
Spring 2007

LECTURE - VIII
DEADLOCKS - I

Tevfik Koşar

Louisiana State University
February 13th, 2007

Roadmap

- Synchronization
  - Monitors
- Deadlocks
  - Deadlock Characterization
    - Resource Allocation Graphs
  - Deadlock Prevention

Dining Philosophers Problem

• Five philosophers spend their time eating and thinking.

• They are sitting in front of a round table with spagett served.

•There are five plates at the table and five chopsticks set between the plates.

• Eating the spaghetti requires the use of two chopsticks which the philosophers pick up one at a time.

•Philosophers do not talk to each other.

•Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher i:

```
Do {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );

        //  eat

    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

        //  think

} while (true) ;
```

Problems with Semaphores

- Wrong use of semaphore operations:
  - semaphores A and B, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

    ➔ Deadlock

  - signal (mutex) …. wait (mutex)
    ➔ violation of mutual exclusion

  - wait (mutex) … wait (mutex)
    ➔ Deadlock

  - Omitting of wait (mutex) or signal (mutex) (or both)
    ➔ violation of mutual exclusion or deadlock

Semaphores

- inadequate in dealing with deadlocks
- do not protect the programmer from the easy mistakes of taking a semaphore that is already held by the same process, and forgetting to release a semaphore that has been taken
- mostly used in low level code, eg. operating systems
- the trend in programming language development, though, is towards more structured forms of synchronization, such as monitors and channels

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
        …
    procedure Pn (…) {……}

    Initialization code ( ….) { … }
            …
    }
}
```

- A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition

---

Monitor - Example

As a simple example, consider a monitor for performing transactions on a bank account.
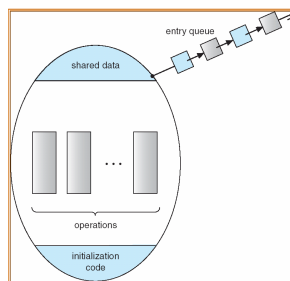
```
monitor account {
    int balance := 0

    function withdraw(int amount) {
        if amount < 0 then error "Amount may not be negative"
        else if balance < amount then error "Insufficient funds"
        else balance := balance - amount
    }

    function deposit(int amount) {
        if amount < 0 then error "Amount may not be negative"
        else balance := balance + amount
    }
}
```

Schematic view of a Monitor

Condition Variables

- Provide additional synchronization mechanism
- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process invoking this operation is suspended
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

  If no process suspended, x.signal() operation has no effect.

Solution to Dining Philosophers using Monitors

```
monitor DP
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];      //to delay philosopher when he is
                             hungry but unable to get chopsticks

initialization_code() {
        for (int i = 0; i < 5; i++)
                state[i] = THINKING;
    }


void pickup (int i) {
        state[i] = HUNGRY;
        test(i);//only if both neighbors are not eating
        if (state[i] != EATING) self [i].wait;
    }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
        if ((state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) &&
            (state[(i + 4) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
        }
    }

void putdown (int i) {
        state[i] = THINKING;
            // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```
➜ No two philosophers eat at the same time
➜ No deadlock
➜ But starvation can occur!
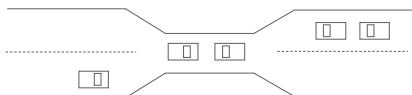
DEADLOCKS

The Deadlock Problem - *revisiting*

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

1. **Mutual exclusion:** nonshared resources; only one process at a time can use a specific resource
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

---

Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

4. **Circular wait:** there exists a set $\{P_0, P_1, …, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.


Resource-Allocation Graph

- Used to describe deadlocks
- Consists of a set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, …, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, …, R_m\}$, the set consisting of all resource types in the system.
- P requests R – directed edge $P_1 \rightarrow R_j$
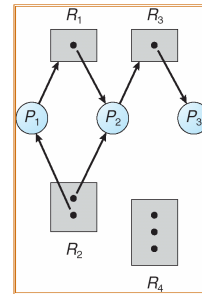- R is assigned to P – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

- Process

- Resource Type with 4 instances

- P_i requests instance of R_j

- P_i is holding an instance of R_j

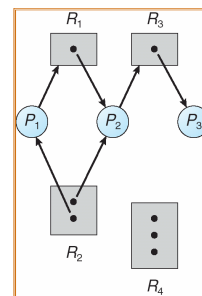### Example of a Resource Allocation Graph

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.

- If graph contains a cycle $\Rightarrow$ there may be a deadlock
  - if only one instance per resource type, then deadlock.
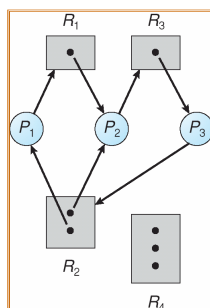 - if several instances per resource type, possibility of deadlock.

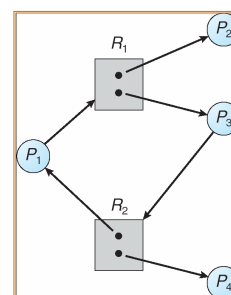### Resource Allocation Graph – Example 1

➔ No Cycle, no Deadlock

Resource Allocation Graph – example 2

➔ Deadlock

Which Processes deadlocked?

➔ P1 & P2 & P3

Resource Allocation Graph – Example 3

➔ Cycle, but no Deadlock

Rule of Thumb

- A cycle in the resource allocation graph
  - Is a necessary condition for a deadlock
  - But not a sufficient condition

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - → deadlock prevention or avoidance
- Allow the system to enter a deadlock state and then recover.
  - → deadlock detection
- Ignore the problem and pretend that deadlocks never occur in the system
  - → Programmers should handle deadlocks (UNIX, Windows)

Deadlock Prevention

→ Ensure one of the deadlock conditions cannot hold

→ Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
  - Eg. read-only files

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  1. Require process to request and be allocated all its resources before it begins execution
  2. or allow process to request resources only when the process has none.

  Example: Write from DVD to disk, then print.
  1. holds printer unnecessarily for the entire execution
     - Low resource utilization
  2. may never get the printer later
     - starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Any Questions?

Hmm..

Reading Assignment

- Read chapter 7 from Silberschatz.

Acknowledgements

- "Operating Systems Concepts" book and supplementary material by Silberschatz, Galvin and Gagne.

31

6