CSE 421/521 - Operating Systems
Fall 2012

Lecture - X
Deadlocks - I

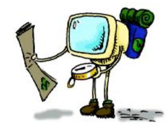
Tevfik Koşar

University at Buffalo

October 2nd, 2012

Roadmap

- Synchronization structures
  - Problems with Semaphores
  - Monitors
  - Condition Variables
- The Deadlock Problem
  - Characterization of Deadlock
  - Resource Allocation Graph
  - Deadlock Prevention
  - Deadlock Detection



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

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
 }
}
```

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!

Sleeping Barber Problem

- Based upon a hypothetical barber shop with one barber, one barber chair, and a number of chairs for waiting customers
- When there are no customers, the barber sits in his chair and sleeps
- As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs
- If all of the chairs are occupied, the newly arrived customer simply leaves

Solution

- Use three semaphores: one for any waiting customers, one for the barber (to see if he is idle), and a mutex
- When a customer arrives, he attempts to acquire the mutex, and waits until he has succeeded.
- The customer then checks to see if there is an empty chair for him (either one in the waiting room or the barber chair), and if none of these are empty, leaves.
- Otherwise the customer takes a seat – thus reducing the number available (a critical section).
- The customer then signals the barber to awaken through his semaphore, and the mutex is released to allow other customers (or the barber) the ability to acquire it.
- If the barber is not free, the customer then waits. The barber sits in a perpetual waiting loop, being awakened by any waiting customers. Once he is awoken, he signals the waiting customers through their semaphore, allowing them to get their hair cut one at a time.

Implementation:

+ Semaphore Customers
+ Semaphore Barber
+ Semaphore accessSeats (mutex)
+ int NumberOfFreeSeats

The Barber(Thread):

```
while(true) //runs in an infinite loop
{
  Customers.wait() //tries to acquire a customer - if none is available he's going to
     sleep
  accessSeats.wait() //at this time he has been awaken -> want to modify the number
     of available seats
  NumberOfFreeSeats++ //one chair gets free
  Barber.signal() // the barber is ready to cut
  accessSeats.signal() //we don't need the lock on the chairs anymore //here the
     barber is cutting hair
}
```

The Customer(Thread):

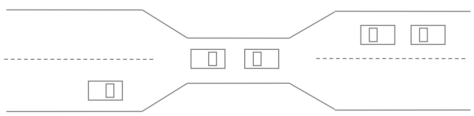
```
while (notCut) //as long as the customer is not cut
{
  accessSteats.wait() //tries to get access to the chairs
  if (NumberOfFreeSeats>0) { //if there are any free seats
    NumberOfFreeSeats -- //sitting down on a chair
    Customers.signal() //notify the barber, who's waiting that there is
    a customer
   accessSeats.signal() // don't need to lock the chairs anymore
   Barber.wait() // now it's this customers turn, but wait if the barber
    is busy
   notCut = false
  } else // there are no free seats //tough luck
  accessSeats.signal() //but don't forget to release the lock on the
    seats }
```


The Deadlock Problem

- 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 disk drives.
  - P_1 and P_2 each hold one disk 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)*

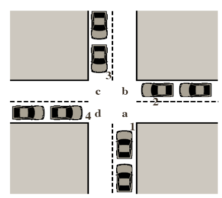
Bridge Crossing Example



- 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.

Deadlock vs Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes



(a) Deadlock possible

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

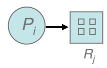
1. **Mutual exclusion:** nonshared resources; only one process at a time can use a specific resource
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption:** a resource can be released only voluntarily by the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

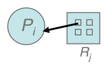
Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

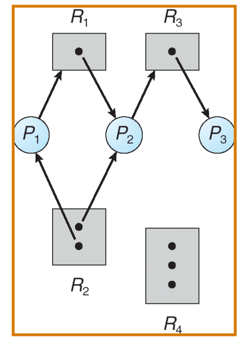
4. **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is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

Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_i 

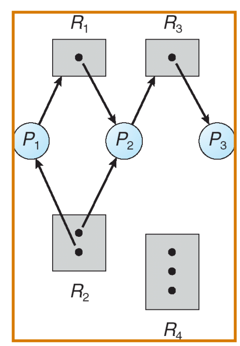
Example of a Resource Allocation Graph



Basic Facts

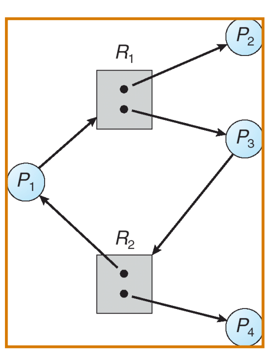
- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒ there may be a deadlock
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Resource Allocation Graph - Example 1



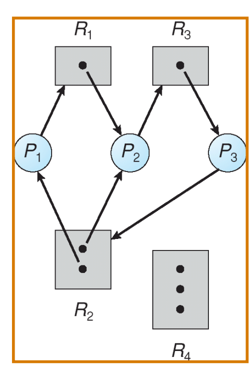
➔ No Cycle, no Deadlock

Resource Allocation Graph - Example 2



➔ Cycle, but no Deadlock

Resource Allocation Graph - example 3



➔ Deadlock

Which Processes deadlocked?

➔ P1 & P2 & P3

Exercise

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

void P0()	void P1()	void P2()
`{ while (true) { get(A); get(B); get(C); // critical region: // use A, B, C release(A); release(B); release(C); } }`	`{ while (true) { get(D); get(E); get(B); // critical region: // use D, E, B release(D); release(E); release(B); } }`	`{ while (true) { get(C); get(F); get(D); // critical region: // use C, F, D release(C); release(F); release(D); } }`

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: Read from DVD to memory, 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.

Exercise

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

b. Modify the order of some of the `get` requests to prevent the possiblity of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

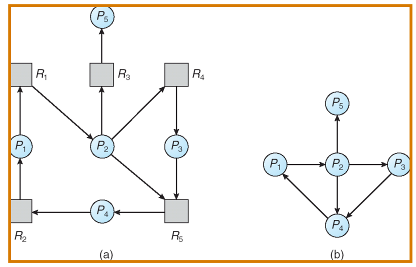
void P0()	void P1()	void P2()
`{ while (true) { get(A); get(B); get(C); // critical region: // use A, B, C release(A); release(B); release(C); } }`	`{ while (true) { get(D); get(E); get(B); // critical region: // use D, E, B release(D); release(E); release(B); } }`	`{ while (true) { get(C); get(F); get(D); // critical region: // use C, F, D release(C); release(F); release(D); } }`

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.



Resource-Allocation Graph   Corresponding wait-for graph

Single Instance of Each Resource Type

- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.
- Only good for single-instance resource allocation systems.

Several Instances of a Resource Type

- *Available*: A vector of length *m* indicates the number of available resources of each type.
- *Allocation*: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- *Request*: An *n* x *m* matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 0,2, \ldots, n-1$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.

Detection Algorithm (Cont.)

3. *Work* = *Work* + $Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $0 \leq i \leq n-1$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Summary

- Synchronization structures
  - Problems with Semaphores
  - Monitors
  - Condition Variables
- The Deadlock Problem
  - Characterization of Deadlock
  - Resource Allocation Graph
  - Deadlock Prevention
  - Deadlock Detection

- Next Lecture: Deadlocks - II

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR