Deadlocks

Deadlock problem. Necessary conditions for deadlock. Resource allocation graphs. Java deadlock examples. Deadlock prevention. Deadlock avoidance (safe state and Banker's algorithm). Deadlock detection and recovery.

NB: The concept of *safe state* and *Banker's algorithm* (Slides 7.22-37) will be covered during the Lab #3 class (17/2/2016)

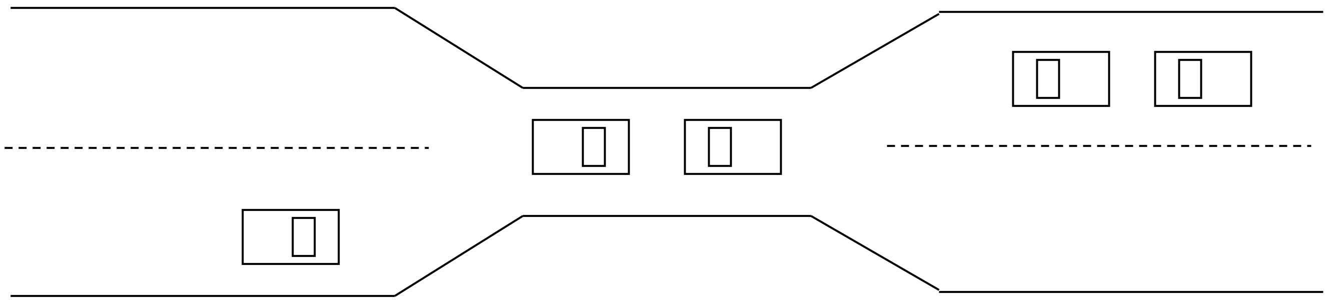
OS5: 16/2/2016
Textbook (SGG): Ch. 7.1-7.4, 7.5.1, 7.5.3, 7.6.2, 7.7

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 1
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one
- Example 2: (binary) semaphores *A* and *B*, initialized to 1

P_0	P_1
acquire(A);	acquire(B);
acquire(B);	acquire(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 back up in general
- Starvation is possible (cars in one direction only keep going)
- Note: most operating systems do not prevent or resolve deadlocks completely

General System Model

- Resource types R_1, R_2, . . ., R_m
  - Physical interpretation of resource types: CPU cycles, memory, I/O devices, etc
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Necessary Conditions for Deadlock

Deadlock *can* occur if these four conditions hold simultaneously:

- **Mutual exclusion:** Only one process at a time can use a resource
- **Hold and wait:** A process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** There exists a set $\{P_0, P_1, \ldots, P_n\}$ 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

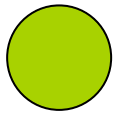
Resource-Allocation Graph

A directed graph, with set of vertices V and set of edges E

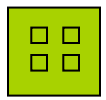
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set of all the resource types in the system
- **request edge**: directed edge $P_i \rightarrow R_j$
  - P_i wants to acquire an instance of R_j
- **assignment edge:** directed edge $R_j \rightarrow P_i$
  - An instance of R_j is being held by P_i

Resource-Allocation Graph (cont'd)

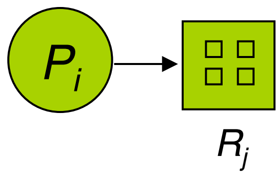
- Process



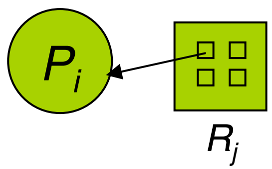
- Resource type with 4 instances



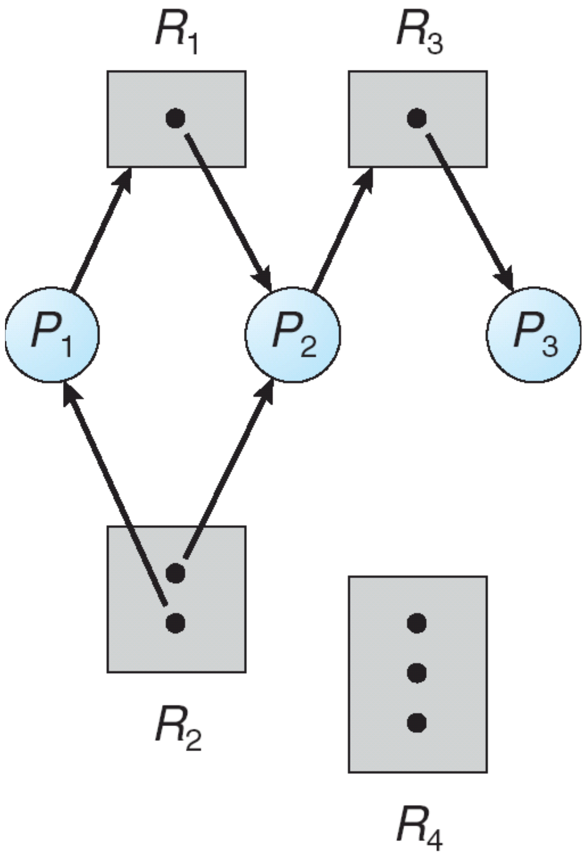
- P_i requests an instance of R_j



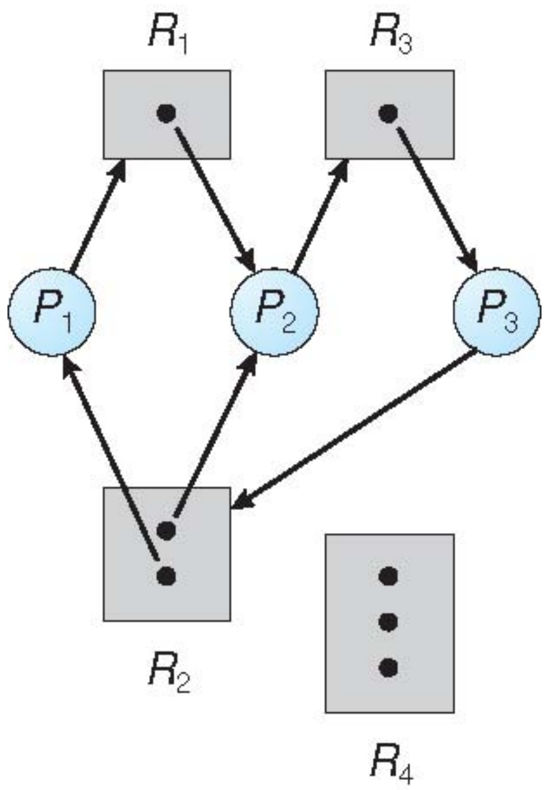
- P_i is holding an instance of R_j



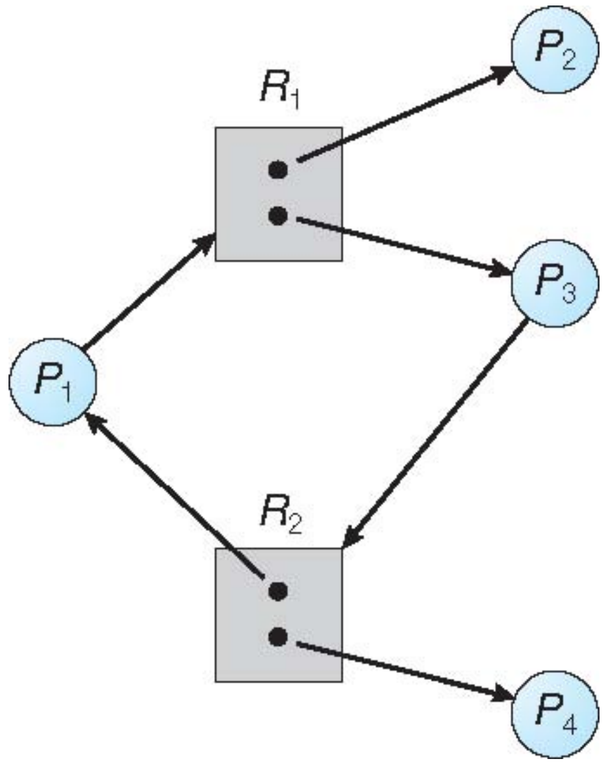
Example Resource Allocation Graph



Resource Allocation Graph with a Deadlock



Graph with A Cycle but No Deadlock



Graph Cycles and Deadlock

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph has a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, then *possibility* of deadlock

Java Deadlock Example

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}
```

Thread A

```
class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.unlock();
      }
   }
}
```

Thread B

Java Deadlock Example

```
public static void main(String arg[]) {
    Lock lockX = new ReentrantLock();
    Lock lockY = new ReentrantLock();

    Thread threadA = new Thread(new A(lockX,lockY));
    Thread threadB = new Thread(new B(lockX,lockY));

    threadA.start();
    threadB.start();
}
```

Assume that lockX and lockY are initially available. How can deadlock occur? (Demonstrate an interleaving of the two threads' execution.)

Handling Deadlocks in Java

```
public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
       while (true) {
          try {
             // sleep for 1 second
             Thread.sleep(1000);

             // repaint the date and time
             repaint();

             // see if we need to suspend ourself
             synchronized (mutex) {
                while (ok == false)
                   mutex.wait();
             }
          }
          catch (InterruptedException e) { }
       }
    }

    public void start() {
       //  Slide 7.18
    }

    public void stop() {
       //  Slide 7.18
    }

    public void paint(Graphics g) {
      g.drawString(new java.util.Date().toString(),10,30);
    }
}
```

- Applet creates clockThread to display wall clock time every second
- clockThread runs only if applet is visible (e.g., not minimized) – variable ok (shared by main Applet thread and clockThread) indicates visible or not
- clockThread blocks on visible condition whenever ok is false
- NB: repaint() method ultimately calls paint() to display current wall clock time

Handling Deadlocks in Java

```
// this method is called when the applet is
// started or we return to the applet
public void start() {
    ok = true;

    if (clockThread == null) {
        clockThread = new Thread(this);
        clockThread.start();
    }
    else {
        synchronized(mutex) {
            mutex.notify();
        }
    }
}

// this method is called when we
// leave the page the applet is on
public void stop() {
    synchronized(mutex) {
        ok = false;
    }
}
```

- When applet becomes visible, main Applet thread calls start() method
- When applet becomes invisible, main thread calls stop() method
- Note: per usual practice, must wait() and notify() visible condition *only if* mutex lock for ok is held

Methods for Handling Deadlocks

- Deadlock *prevention*
  - Impose conditions on resource requests to ensure that a request can *never* cause the system to enter a deadlock state
- Deadlock *avoidance*
  - Before granting a resource request, check that the request will not cause the system to enter a deadlock state
  - Requires advance knowledge of *future* resource needs (e.g., Banker's algorithm in your OS Lab #3)
- Deadlock *detection and recovery*
  - Detect deadlock after the fact, then recover from it (e.g., preempting held resources and rolling back processes)
- Pretend that deadlock will never occur in the system and don't deal with it – many operating systems, including UNIX, use this approach!

Deadlock Prevention

Constrain the ways requests can be made, in order not to allow *any* one of …

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, *or* allow process to request resources only when the process has none (i.e., previously held resources must be released before new resources are requested)
  - OS must support a system call for acquiring multiple resources, e.g., "acquireAll()" in addition to acquire() for semaphores
  - Disadvantages
    - Low resource utilization – why?
    - Starvation becomes more likely – why?

Deadlock Prevention (cont'd)

- **No Preemption** – if a process holding some resources requests another resource that cannot be immediately granted, then all the resources being held are released
  - Preempted resources are added to the list of resources for which the process needs to wait
  - Process will be restarted only when it can obtain all the resources it needs (both old and new)
  - Disadvantages: preemption costs and starvation
- **Circular Wait** – impose a total ordering of all the resource types, and require that each process requests resources according to that order
  - Disadvantage: Burden on programmer to ensure the order by design, without unnecessarily sacrificing utilization

Deadlock Avoidance

Requests can lead to deadlock, but don't grant those requests that can lead to it, based on additional *a priori* information available

- Simplest and most useful model requires each process to declare (in advance) the *maximum number* of resources of each type that it may need.
- At time of resource request, avoidance algorithm examines the system's resource allocation state to ensure that granting the request will never lead to a *circular-wait* condition.
- Resource allocation state is defined by the numbers of currently available and allocated resources, and the maximum number of resources that each process may need in the future.

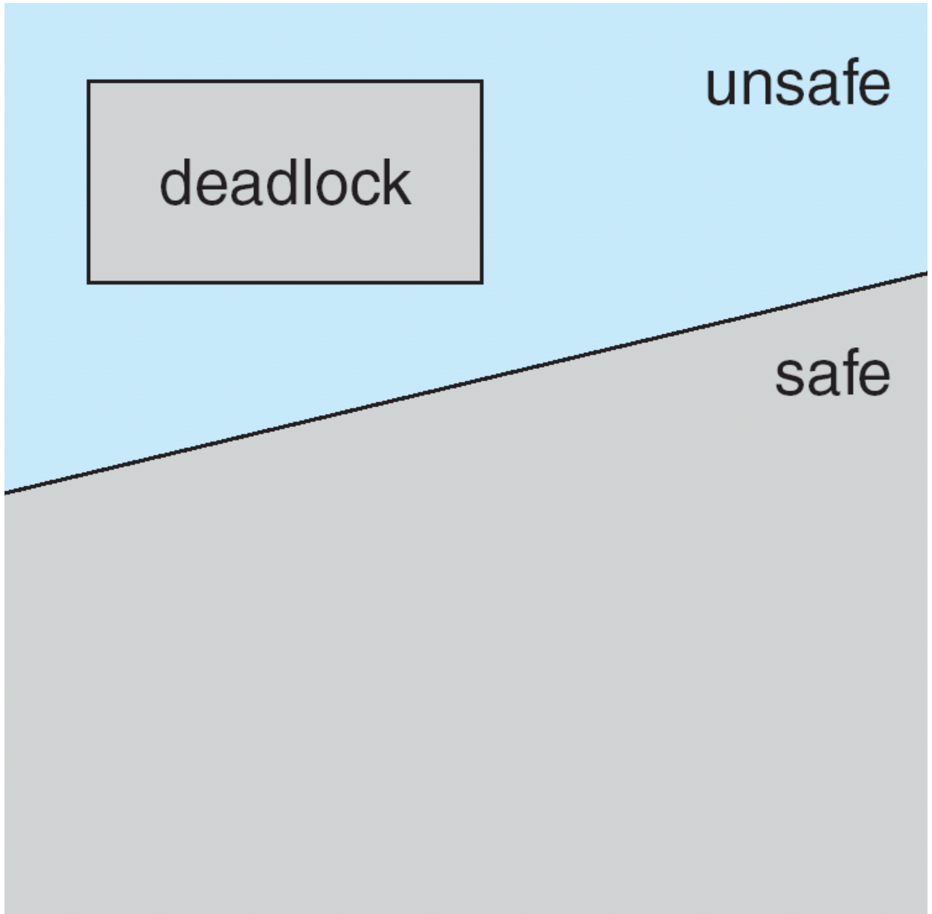
Safe State

- When a process requests an available resource, system must decide if granting the request will leave the system in a safe state.
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all the processes in the system such that for each P_i, the resources that P_i will ever need can be satisfied by the currently available resources *plus* the resources held by all the preceding P_j, with $j < i$.
- Rationale:
  - If P_i's resource needs are not immediately available, then P_i can wait until all the preceding P_j processes have finished.
  - When the preceding processes all finished, P_i can obtain its needed resources, do its job, return its allocated resources, and finish.
  - When P_i finishes, P_{i+1} can obtain its needed resources and finish, and so on.

Basic Facts

- If system is in safe state ⇒ no deadlocks
- If system is in unsafe state ⇒ *possibility* of deadlock
- Avoidance ⇒ ensure that system will never enter an unsafe state.

Safe, Unsafe, Deadlock States



Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
  - We won't cover this algorithm (limited applicability)
- Multiple instances of a resource type
  - Use the Banker's algorithm (your Lab #3)
  - Subsumes the single-instance problem

Banker's Algorithm (Lab 3)

- Multiple instances of each resource
- Each process must *a priori* claim its maximum use
- When a process requests a resource, it may have to wait
- After a process got all its resources, it must return them within a finite amount of time

Data Structures for Banker's Algorithm

Let n = number of processes, and m = number of resources types

- **Available**: Vector of length m. If *Available*[j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If *Max*[i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If *Allocation*[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish[*i*] = *false* for *i* = 0, 1, …, *n* - 1

2. Find an *i* such that:

   (a) *Finish*[*i*] = *false*

   (b) *Need*[*i*] ≤ *Work*

   If no such *i* exists, go to Step 4

3. *Work* = *Work* + *Allocation*[*i*]
   Finish[*i*] = *true*
   go to Step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Algorithm for Granting Request by Process P_i

Request[*i*] = request vector for process P_i. If $Request_i[i,j] = k$, then process P_i wants *k* instances of resource type R_j

1. If *Request*[*i*] ≤ *Need*[*i*] go to Step 2. Otherwise, raise error, since process has exceeded its maximum claim.

2. If *Request*[*i*] ≤ *Available*, go to Step 3. Otherwise P_i must wait, since the resources are not immediately available.

3. Try to allocate the requested resources to P_i by updating the resource allocation state as follows:

$$Available = Available - Request[i];$$

$$Allocation_i = Allocation[i] + Request[i];$$

$$Need[i] = Need[i] - Request[i];$$

- If new state is *safe* ⇒ the resources are allocated to P_i
- If new state is *unsafe* ⇒ P_i must wait, and the old resource-allocation state is restored

Activity 5.1: Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

  Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Is the system safe? If so, give an execution sequence of the processes that demonstrates the safety. If not, why not?

Activity 5.1 (cont'd): P_1 then requests (1,0,2)

- Check that Request ≤ Available, i.e., (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies the safety condition

- Can we now grant (i) request for (3,3,0) by $P_{4;}$ or (ii) request for (0,2,0) by P_0?

Deadlock Detection

- Allow system to enter deadlock state
- Detect occurrence of deadlock by *detection algorithm*
- Recover from the detected deadlock

Deadlock Detection: Multiple Instances

- **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.

Algorithm is similar to Banker's algorithm for deadlock avoidance.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   For *i* = 1,2, …, *n*, if *Allocation*[i] ≠ 0, then
   Finish[i] = false; else *Finish*[i] = *true*

2. Find an index *i* such that both:

   (a) *Finish*[*i*] == *false*

   (b) *Request*[*i*] ≤ *Work*

   If no such *i* exists, go to Step 4

3. *Work* = *Work* + *Allocation*[*i*]
   Finish[*i*] = *true*
   go to Step 2

4. If *Finish*[*i*] == *false* for some *i*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

NB: Complexity of algorithm is **O(*m* x *n*²)**

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*

Detection Example (cont'd)

- 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

- New state of the system?
  - Can reclaim resources held by process P_0, but still insufficient resources to satisfy the other processes' requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Deadlock Recovery: Abort Processes

- Abort all deadlocked processes (resources held are preempted)
- Abort one process at a time until the deadlock cycle is eliminated
- Which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer until completion?
  - Resources the process has used,
  - Resources the process needs to complete.
  - How many processes will system need to abort?
  - Is process interactive or batch?
  - etc …