Deadlocks

Electrical and Computer Engineering
Stephen Kim (dskim@iupui.edu)

Deadlocks

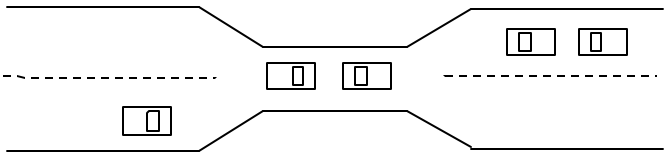
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

- A set of blocked processes each holding resources and waiting to acquire resources held by another process in the set.
- Example
  - System has 2 tape drives.
  - For two processes P_1 and P_2, each holds 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.

System Model

- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has n_i instances.
- Each process utilizes a resource as follows:
  - Request: If the request cannot be granted immediately, the requesting process must wait until it can acquire the resouce
  - Use:
  - Release: The process releases the resource after using it.
- The request and release are implemented as system calls in OS.
  - open file vs. close file
  - allocate memory vs. free memory

Deadlock Characterization

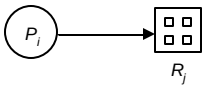
- **Deadlock can arise if 4 conditions hold simulatenously**
  - **Mutual exclusion:** only one process at a time can use a resource.
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
  - **No preemption:** a resource can 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 (1)

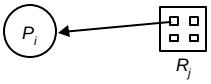
- The set of vertices V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- The set of edges E,
  - Request edge – directed edge $P_i \rightarrow R_j$.
  - Assignment edge – directed edge $R_j \rightarrow p_i$.

Resource-Allocation Graph (2)

- Process is represented by a circle.
- Each resource type is indicated by a rectangle.
  - If a resource type has more than one instance, the instance is represented by a dot within the rectangle.
  - A request-edge points to only the square, but
  - an assignment-edge must also designate one of the dots in the rectangle



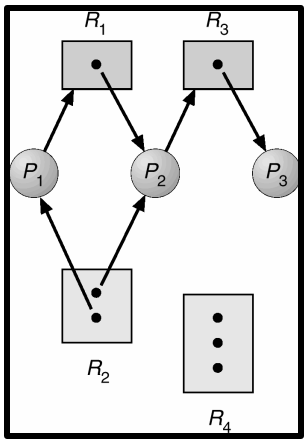
Request-edge



Assignment-edge

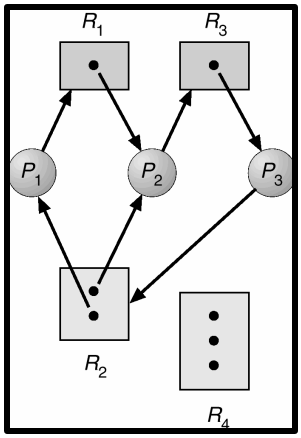
Example of a Resource Allocation Graph

- If the graph contains no cycles, then no process is deadlocked.



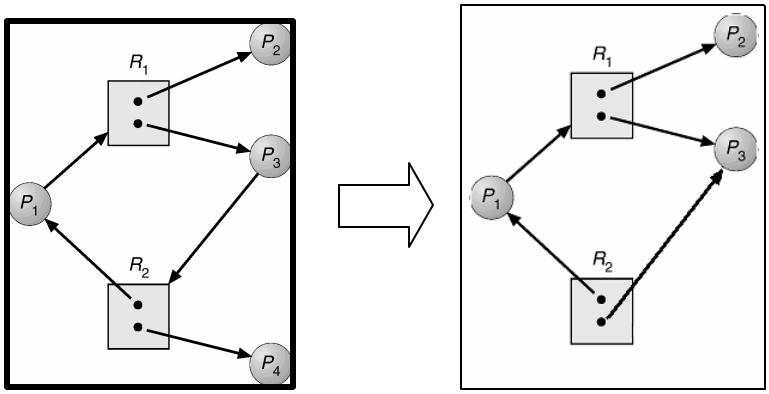
Resource Allocation Graph With A Deadlock

- If the graph does contain a cycle, then a deadlock *may* exist.



Resource Allocation Graph With A Cycle But No Deadlock

- If each resource type has several instances, then a cycle does not necessarily imply a deadlock
- Once P_4 releases an instance of resource type R_2, the resource can be allocated to P_3 so the cycle can be broken.



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, *may or may not have deadlocks*.

Methods for Handling Deadlocks

- Prevention and avoidance – to ensure that the system will *never* enter a deadlock state.
  - deadlock prevention – to make sure at least one of the necessary conditions never occur.
  - deadlock avoidance -- the OS has additional information in advance, and decides whether a process should wait or not a request.
- Recovery – to allow the system to enter a deadlock state, detect it and then recover.
- Ignorance – to ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention (1)

- **Mutual Exclusion**
  - sharable resources need no mutual exclusion (no need to wait for sharable resources)
  - must hold for nonsharable resources.
- **Hold and Wait**
  - must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - pre-allocation – to require process to request and be allocated all its resources before it begins execution,
  - single allocation – to allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention (2)

- **No Preemption**
  - If a process 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**
  - to impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

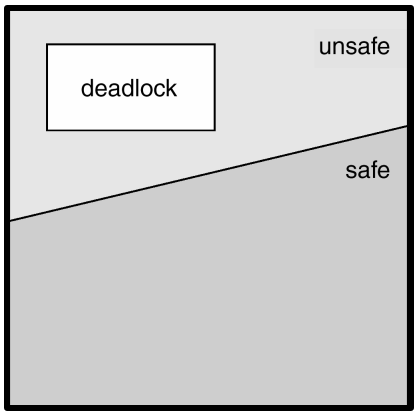
- Simplest and most useful model requiring that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state.*
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus resources held by all the P_j, with $j<i$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

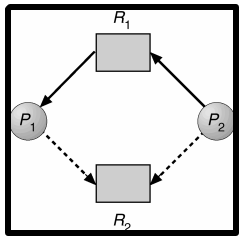
Safe, Unsafe , Deadlock State

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

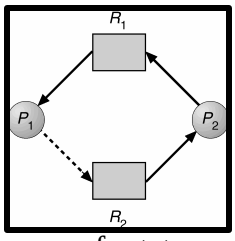


Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.
- This graph algorithm can solve the problem if each resource has a single instance



deadlock avoidance



unsafe state

Banker's Algorithm

- The previous resource allocation graph cannot solve the problem if the system has multiple instances of each type.
- Banker's algorithm to solve the multiple instance problem
- Assumption
  - Each process must a priori claim maximum use.
  - When a process requests a resource it may have to wait.
  - When a process gets all its resources it must return them in 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:* $n \times m$ matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j.
- *Need:* $n \times m$ 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 - 1,3, …, n.
2. Find an i such that both:
   (a) $Finish\ [i] = false$
   (b) $Need_i \leq Work$
   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.
1. If $Finish\ [i]$ == true for all i, then the system is in a safe state

Resource-Request Algorithm

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

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_{i;;}$$

- *If the resulting state is safe, then the resources are allocated to* P_i.
- *If the resulting state is unsafe* Þ P_i *must wait for* $Request_i$, *and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances),
  B (5instances, and C (7 instances).
- Snapshot at time T_0:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Example (Cont.)

- The content of the matrix. Need is defined to be *Max – Allocation*.

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

- The system is in a safe state since the sequence $<P_1,P_3,P_4,P_2,P_0>$ satisfies the safety criteria.

Example P_1 Request (1,0,2)

- Check that *Request* ≤ *Available* (that is, (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 safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

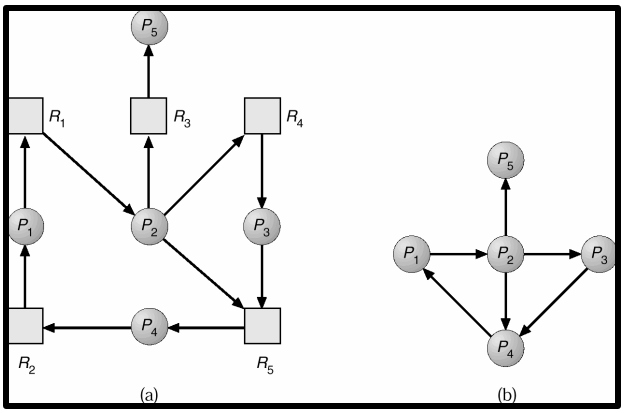
Deadlock Detection

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

Single Instance of a Resource Type

- Maintain *wait-for* graph
  - Nodes are processes only.
  - $P_i \rightarrow P_j$ iff P_i is waiting for *a resource* R_k *allocated to* P_j.
- 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.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length *m* indicates the number of available resources of each type.
- *Allocation:* An *n´m* matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An *n´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 $i = 1,2, \ldots, n$, 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.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

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*.
- Q: List all possible safe sequences.

Example (Cont.)

- P_2 requests an additional instance of type C.

	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 1	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	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.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.