CMSC 412

Deadlock

Announcements

- Reading
  - Chapter 8

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 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example - semaphores *A* and *B*, set to 1

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

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Four necessary conditions

- **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.

Deadlock Characterization

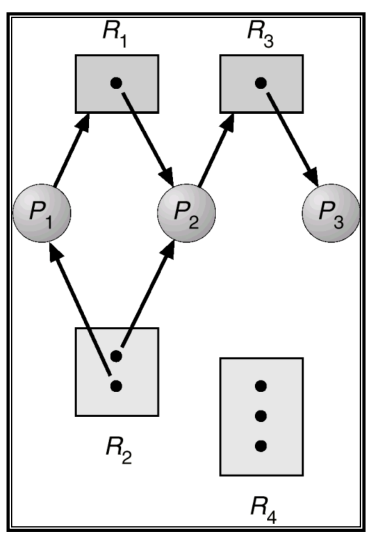
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that
  - P_0 is waiting for a resource that is held by P_1
  - P_1 is waiting for a resource that is held by P_2
  - …, P_{n-1} is waiting for a resource that is held by P_n, and
  - P_n is waiting for a resource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

Resource Allocation Graph

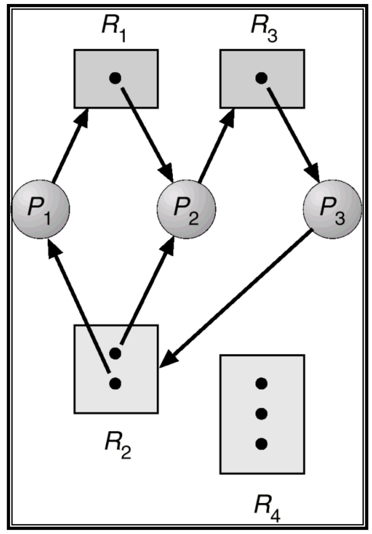
*A set of vertices **V** and a set of edges **E***

- **V** is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
- **E** has two types
  - request edge – directed edge P1 → Rj
  - assignment edge – directed edge Rj → Pi

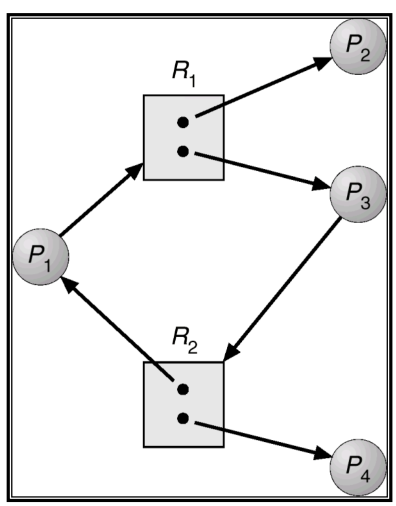
Example Resource Allocation Graph



Graph With A Deadlock



Graph With A Cycle, No Deadlock



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, possibility of deadlock.

Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways a request can be made

- Mutual Exclusion – Sharable resources do not require mutually exclusive access and cannot be involved in a deadlock.
- 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.
  - Low resource utilization; starvation possible.

Deadlock Prevention

- **No Preemption** – Virtualize resources and permit them to be preempted. For example, the CPU can be preempted.
- **Circular Wait** – Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

- Deadlock prevention restricts some large class of behaviors *a priori*
  - Some behaviors within this class might be legal in some circumstances
- Deadlock avoidance permits more behaviors, relying on dynamic checks
  - Actions that could possibly lead to deadlock are avoided

Deadlock Avoidance Approach

- Each process declares the maximum number of resources of each type that it may need.
- OS dynamically ensures that a request can never cause the resource-allocation state to eventually be in a circular-wait condition.
- Resource-allocation state is defined by
  - The number of available resources
  - The number of 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*.
- The state is safe if the resources could be allocated to the processes in *some order*.
  - I.e., system is in safe state if there exists a *safe sequence* of all processes.

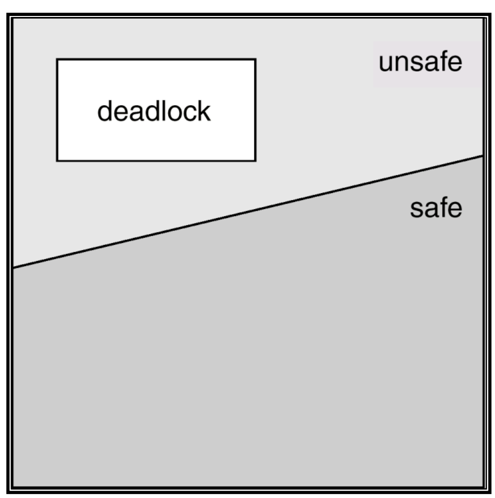
Safe Process Sequence

- Sequence <P_1, P_2, …, 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 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.

Basic Facts

- 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.

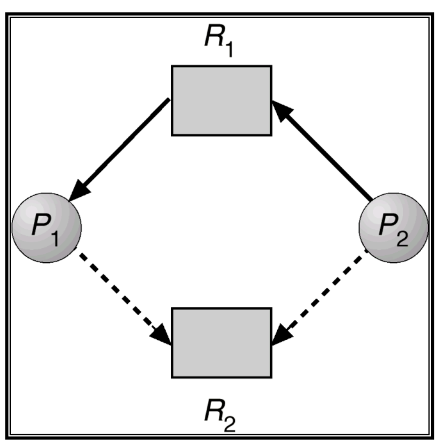
Safe, Unsafe, Deadlock State



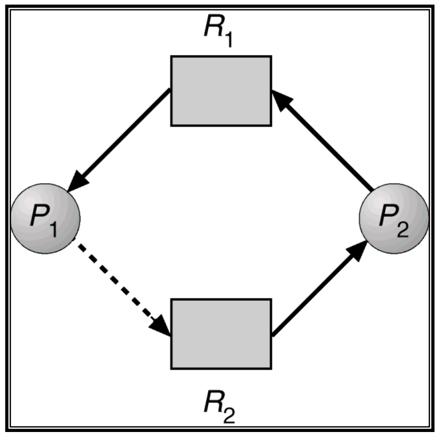
Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line. One instance per resource type.
- 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.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Multiple resource instances.
- Each process must *a priori* claim maximum resources in use at any time.
- 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.

12

Banker's Algorithm

- Variables:

  n is the number of processes

  m is the number of resource types

  - Available - vector of length *m* indicating the number of available resources of each type
  - Max - *n* by *m* matrix defining the maximum demand of each process
  - Allocation - *n* by m matrix defining number of resources of each type currently allocated to each process
  - Need: *n* by *m* matrix indicating remaining resource needs of each process
- $Need[i,j] = Max[i,j] - Allocation[i,j]$.

Safety Algorithm

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

   Work = *Available*

   Finish[*i*] = *false* for *i* = 1,2, …, *n*.
2. Find and *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.
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for P_i

$Request_i$ = request vector for process P_i.

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

Algorithm:

1. If $Request_i \leq Need_i$ go to step 2. Otherwise error: process has exceeded its maximum claim.
2. If $Request_i \leq Available$ go to step 3. Otherwise P_i waits, since resources are not available.

Resource-Request Algorithm for P_i

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 safe $\Rightarrow$ the resources are allocated to P_i.*
- *If unsafe $\Rightarrow$ P_i must wait, 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*
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	

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 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.

Example P_1 Requests

	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	

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

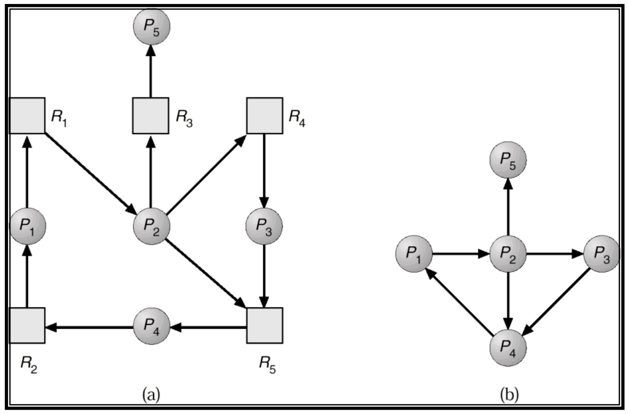
Deadlock Detection

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

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes.
  - $Pi \rightarrow Pj$ if Pi is waiting for Pj.
- 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* defines the number of available resources per 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 = 1,2, …, 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.

Detection Algorithm (Cont.)

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.

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

19

Example of Detection Algorithm

- Five processes P_0 through P_4
- Three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).

	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 cannot fulfill other processes' requests.
  - Deadlock with 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 rollbacks 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.