Chapter 7. Deadlocks

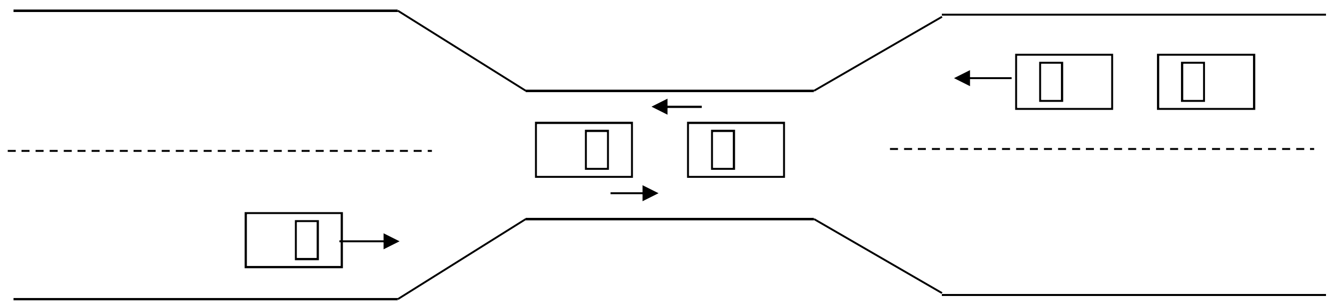
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

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 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.

1. 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

  - Example of file i/o
    - Open
    - Read/write
    - Close

2. Deadlock Characterization

Deadlock can, not will, arise if 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 $\{P_0, P_1, \ldots, P_n, P_0\}$ 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.

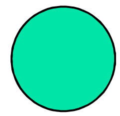
Resource-Allocation Graph

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

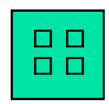
- *V* is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $P_1 \rightarrow R_j$
- Assignment edge – directed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

Resource-Allocation Graph – cont.

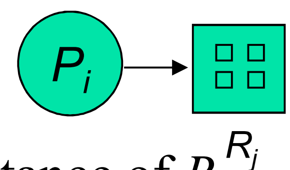
- Process



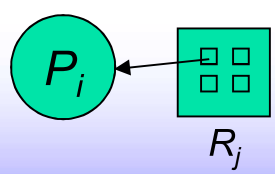
- Resource type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

- See Figure 7.1, “Resource-allocation graph”.
  - ☺ *Deadlock?*

Resource Allocation Graph

- See Figure 7.2, “Resource-allocation graph with a deadlock”.
  - ☺ *Deadlock?*

Resource Allocation Graph – cont.

- See Figure 7.3, “Resource-allocation graph with a cycle but no deadlock”.
  - ☺ *Deadlock?*

Basic Facts

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

3. Methods for 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.
  - ☺ *Why?*

- Prevention
- Avoidance
- Detection and recovery

4. Deadlock Prevention

Idea: Breaking one of four necessary conditions for deadlocks

- **Mutual Exclusion**
  - Not required for sharable resources
  - However, must hold for nonsharable resources.

- **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.
  - => However, low resource utilization; starvation possible.

Deadlock Prevention - continued

- **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.

Deadlock Prevention - continued

- **Circular Wait**
  - $R = \{R_1, R_2, \ldots, R_m\}$
  - $F: R \text{ -> } N$
  - **Protocol**
    - Each process requests resources in an increasing order of enumeration.
    - A process can request instances of resource type R_j iff $F(R_j) > F(R_i)$ for all R_i held by the process.
    - If several instances of the same resource type are needed, a single request for all of them must be issued.
    - Then, there will be no circular wait.
  - ☺ *Can you prove it?*
- **However, inefficient**
  - ☺ ***Why???***

5. Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires 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 + resources held by all the P_j, with $j < i$.
  - If the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
  - When all P_j have 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.

  - => ☺ *Acyclic graph always?*

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
- See Figure 7.8, “Safe, unsafe, and deadlocked state spaces”.

Resource-Allocation Graph Algorithm

- One instance for each resource type:
  - ***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.

- See Figure 7.9 – 7.10.

Banker's Algorithm

- Multiple instances for each resource type:
  - **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 the Banker's Algorithm

Let n = number of processes, and m = number of resource 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

Find out whether or not a system is in a safe state:

Try to find a safe sequence of all processes.

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

   $Work = Available$ - resources

   $Finish[i] = false$ for $i = 0, 1, 2, 3, \ldots, n\text{-}1$. - processes
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.
4. If $Finish[i]$ == true for all i, then the system is in a safe state.

Resource-Request Algorithm for Process 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.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, 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 safe, i.e., there is a safe sequence $\Rightarrow$ the resources are allocated to P_i.*
- *If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored*

However, inefficient

☺ ***Why???***

The above operation is very expensive because of high complexity, $O(mn^2)$, of the safety checking algorithm.

Example of 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	

Example - continued

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

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

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

Example P_1 Request (1,0,2) Cont.

- Check that Request$_1$ (1,0,2) $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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 2	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 Request (1,0,2) Cont.

Currently

-

	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 2	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?*
  - No, because $Need_4 > Available_4$
- ☺ *Can request for (0,2,0) by P_0 be granted?*
  - No, because *Available*:(2,3,0) => (2,1,0), and then no process can finish.

6. 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.
- Periodically invoke an **algorithm that searches for a cycle** in the graph.
- An algorithm to detect a cycle in a graph requires $O(n^2)$ operations, where n is the number of vertices in the graph.
- See Figure 7.11.

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

Try to find a safe sequence of all processes

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.

Detection Algorithm - continued

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.

4. If $Finish[i]$ == false, for some i, $1 \le i \le n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked.

Algorithm requires an order of O(mn^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 - continued

- P_2 requests an additional instance of type C.

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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.

Detection-Algorithm Usage

- ☺ ***When, and how often, to invoke???***
- It 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 we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

7. 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 the process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources the 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.