Chapter 7: Deadlocks

- ♦ The Deadlock Problem
- ♦ System Model (7.1)
- ♦ Deadlock Characterization (7.2)
- ♦ Methods for Handling Deadlocks (7.3)
- ♦ Deadlock Prevention (7.4)
- ♦ Deadlock Avoidance (7.5)

Chapter Objectives

- ♦ To develop a description of deadlocks, which prevents sets of concurrent processes from completing their tasks

- ♦ To present a number of different methods fro preventing or avoiding deadlocks in a computer system

The Deadlock Problem

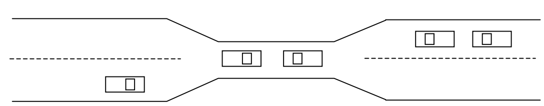
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

- **Example**

  – System has 2 tape drives.
  – P_1 and P_2 each hold one tape drive and each needs another one.

The Deadlock Problem (cont.)

- 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 (7.1)

- 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 (7.2)

Deadlock can arise if four conditions hold simultaneously (necessary condition).

Deadlock $\Rightarrow$ 4 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.

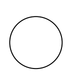
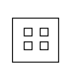
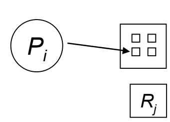
Deadlock Characterization (7.2) (cont.)

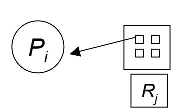
- Resource-Allocation Graph

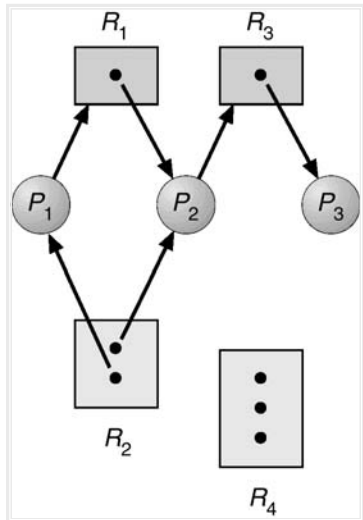
  A set of vertices *V* and a set of edges *E*.

  - 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 $P_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$

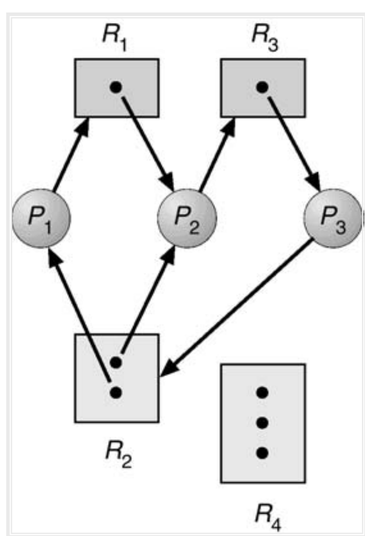
Deadlock Characterization (7.2) (cont.)

- 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 

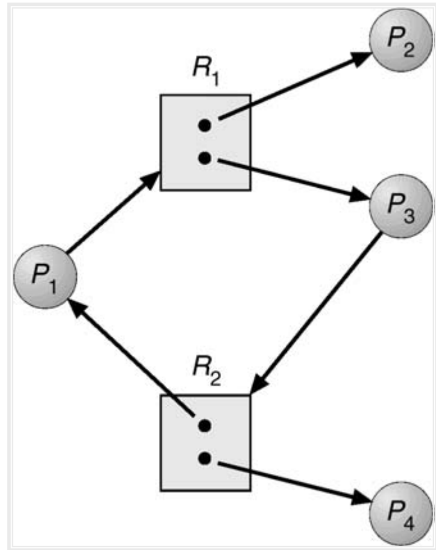


Example of a Resource Allocation Graph



P_2 req R_3 hold by
P_3 req R_2 hold by P_1
req R_1 hold by P_2

Resource Allocation Graph With a Deadlock



P_4 may release
Its resource that
will be allocated to P_3
Thus breaking the cycle

Resource Allocation Graph With A Cycle But No Deadlock

Deadlock Characterization (7.2) (cont.)

- 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.

Methods for Handling Deadlocks (7.3)

- 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 (7.4)

One of the conditions should not hold !

- Mutual Exclusion – not required for sharable resources (not involved in a deadlock); must hold for nonsharable resources (such as printer).
- Hold and Wait (never occurs)– 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 (tape-disk-printer), or allow process to request resources only when the process has none.
  - Disadvantages: low resource utilization; starvation is possible.

Deadlock Prevention (7.4) (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.

Deadlock Avoidance (7.5)

Requires that the operating 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.

Deadlock Avoidance (7.5) (cont.)

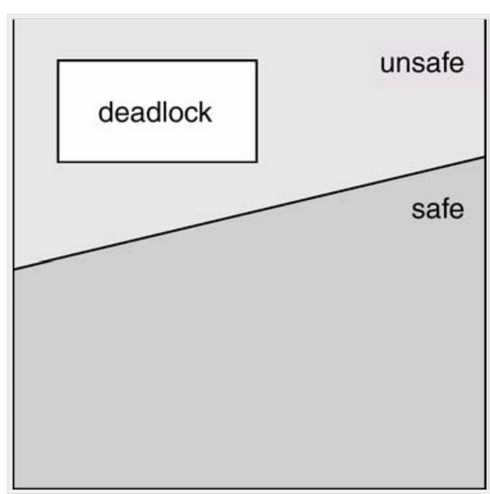
- Safe State
  - When a process requests an available resource, the operating system must decide if immediate allocation leaves the system in **a *safe state***.
  - System is in safe state **if there exists** a safe sequence of all processes.

Deadlock Avoidance (7.5) (cont.)

- Safe state (cont.)
  - Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that P_i can still request (the need) can be satisfied by currently available resources + resources held by all the P_j , with $j<i$ *(order is important!)*.
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j $(j<i)$ 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.

Deadlock Avoidance (7.5) (cont.)

- Safe state (cont.)
  - 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.

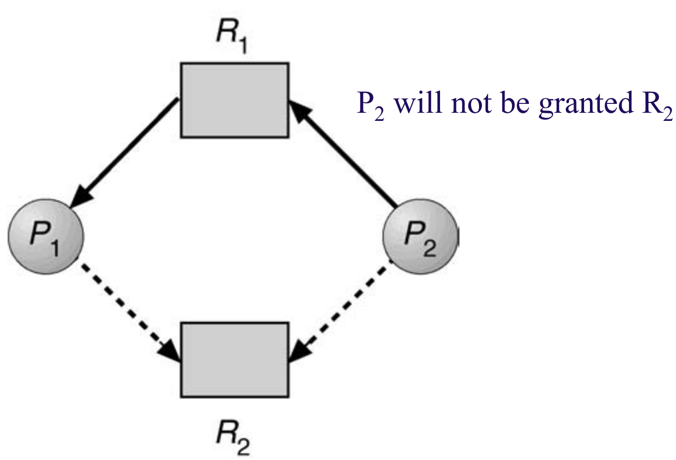


Safe, Unsafe , Deadlock State

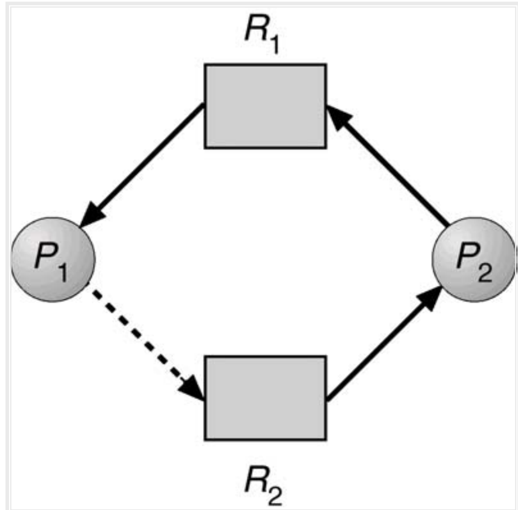
Deadlock Avoidance (7.5) (cont.)

- Resource-Allocation Graph (RAG) Algorithm: Case of one instance of resource types
  - *Claim edge* P_i ---> R_j indicated that process P_i 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.

<u>Principle</u>:
- P_i requests R_j. *It is granted only if the assignment* $R_j \rightarrow P_i$ *does not create a cycle in the RAG*



Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph

Deadlock Avoidance (7.5) (cont.)
Case of Multiple Instances of Resources

- ♦ Banker's Algorithm = update resources + test safety
  - – Multiple instances of resources (the RAG algorithm is not applicable in this case!).
  - – 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.

Deadlock Avoidance (7.5) (cont.)

- Data Structures for the 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 $\text{Allocation}[i,j] = k$ 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.

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

Deadlock Avoidance (7.5) (cont.)

- Banker's Algorithm (cont.)
  - 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 index i such that both:

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

   (b) $Need_i \le Work$

   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.

Deadlock Avoidance (7.5) (cont.)

- Banker's Algorithm (cont.)
  - Resource-Request Algorithm for Process P_i

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 safe* $\Rightarrow$ *the resources are allocated to* P_i. *(calls safety algorithm)*
- *If unsafe* $\Rightarrow$ P_i *must wait, and the old resource-allocation state is restored*

Deadlock Avoidance (7.5) (cont.)

- How does Banker's Algorithm Work ?
  - 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:
  - Using the update algorithm

avail = avail-alloc

	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	

Deadlock Avoidance (7.5) (cont.)

- Example (cont.)
  - The content of the matrix Need is related as: **Max – Allocation = Need.**

	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

<P_1,P_3,P_4,P_2,P_0> safe:
P_1 may request
1 2 2, covered by
current available 3 3 2
P_3 may request
0 1 1 covered by
current available 2 1 0 + alloc(P_1) 1 2 2

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

av:=av – req;
alloc:= alloc+req;
need= need -req

Deadlock Avoidance (7.5) (cont.)

- Example (cont.): P_1 Request (1,0,2)
  - Check that $\text{Request}_1 \leq$ Available (that is, $(1,0,2) \leq (3,3,2)$) $\Rightarrow$ *true* $\Rightarrow$ fulfillment of the request (but not yet granted!).

update

	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 $\Rightarrow$ P_1 request is granted.
  - Can request for (3,3,0) by P_4 be granted? Answer is “no” (resources not available !)
  - Can request for (0,2,0) by P_0 be granted? Answer is “no”(resulting state is unsafe since no safe sequence can be found!!!)