Chapter 8: Deadlocks

1

Chapter Topics

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

2

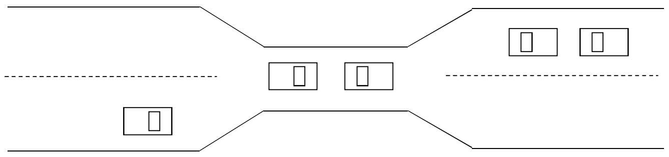
The Deadlock Problem

- A set of blocked processes each of which holds a resource and waits to acquire a resource held by another process in the set, such that they have to wait permanently.
- Examples
  - Each of two processes P_1 and P_2 locks one file and needs the other one.
  - Two semaphores *A* and *B*, initialized to 1.

```
        P1              P2
    wait (A);       wait(B);
    wait (B);       wait(A);
```

3

Bridge Crossing Example



- Traffic is allowed only in one direction at a time.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car is backed up (i.e., preempted resources and rolled back).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible (even if deadlock is resolved).
- Note: Most OSs do not prevent or avoid deadlocks.

4

System Model

- Resource types R_1, R_2, . . ., R_m
  - e.g., CPU cycles, memory space, I/O devices, data files.
- Each resource type R_i may have W_i identical instances.
- Each process utilizes a resource as follows:
  1. Request
  2. Use
  3. Release

5

Four Necessary Conditions for a Deadlock

Following four conditions must be effective for a deadlock to exist; i.e., if a deadlock happens, these conditions hold simultaneously.

- **Mutual exclusion:** There are resources held in a non-sharable mode; that is, only one process at a time can use the resource.
- **Hold and wait:** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
- **No preemption:** Resources cannot be preempted. Instead, a resource can only be released voluntarily by the process holding it, after that process has completed its use of the resource.
- **Circular wait:** There must exist a set of waiting processes {P_1, P_2, …, P_n} such that P_1 is waiting for a resource that is held by P_2, P_2 is waiting for a resource that is held by P_3, … , 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_1.

6

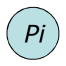
Resource Allocation Graph

- A resource allocation graph is a directed graph with a set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$ is the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$ is the set consisting of all resource types in the system.
- E is partitioned into two types:
  - request edge: directed edge $P_i \rightarrow R_j$
  - assignment edge: directed edge (an instance of) $R_j \rightarrow P_i$

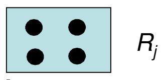
7

Resource Allocation Graph (cont'd)

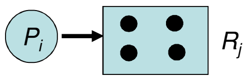
- Process



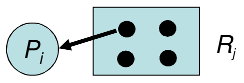
- Resource type with 4 instances



- P_i requests an instance of R_j: request edge

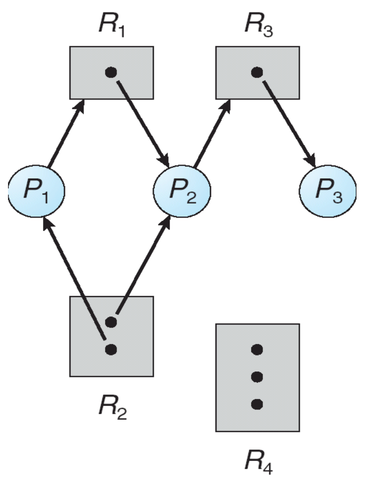


- P_i is holding an instance of R_j: assignment edge



8

Example of a Resource Allocation Graph

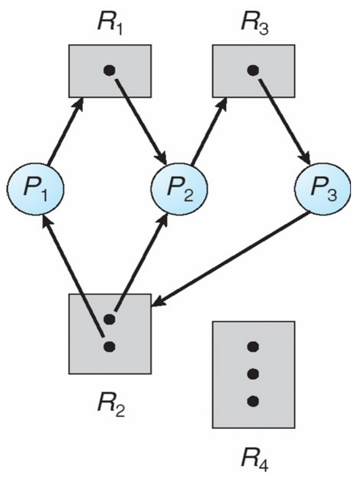


- As there is no circular wait (represented by a cycle), there is no deadlock in this case.

9

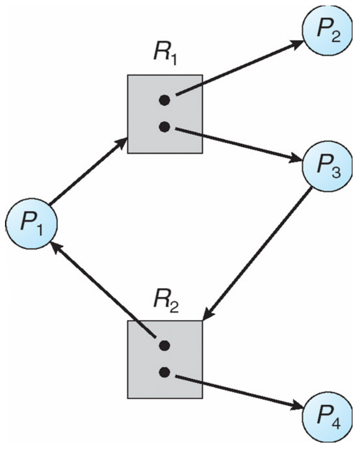
Resource Allocation Graph with a Deadlock

- Suppose that process P_3 requests an instance of resource type R_2.
  - Since no resource instance is available, a request edge (P_3, R_2) is added as shown in this graph.
- There are two simple cycles in this graph, and processes P_1, P_2, and P_3 are deadlocked.



10

Graph with a Cycle but No Deadlock



- P_4 may release an instance of R_2, and that instance can be allocated to P_3, which will break the cycle.

11

Basic Facts

- If a resource allocation graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then there is a deadlock.
  - if several instances per resource type, then a deadlock is possible.

12

Methods for Handling Deadlocks

- **Deadlock prevention**: condition a system to remove any possibility of deadlocks to occur.
- **Deadlock avoidance**: pretend the current request is granted, then examine the post-grant resource allocation state to ensure that there is a way all the processes can complete without causing a deadlock. If not, the current request is rejected.
- **Deadlock detection and recovery**: determine if a deadlock has occurred; and if yes, determine the processes and resources involved in the deadlock and clear the deadlock.
- **Ignore the deadlock problem:** pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

13

Deadlock Prevention

Idea: By ensuring that at least one of the four necessary conditions cannot hold, we can prevent the occurrence of a deadlock.

- **Denying the *Mutual Exclusion***: However, some resources are intrinsically non-sharable (e.g., semaphore).
- **Denying the *Hold and Wait*:** must guarantee that whenever a process requests a resource, it does not hold any other resources:

(A) Require each process to request and be allocated all its resources before it begins execution.

- – May cause low resource utilization because some of the allocated resources may not be used for a long time.

(B) Require each process to release all the resources currently allocated, before it can request any additional resources.

- – The process may need to request some of the currently allocated resources (after releasing them) together with new ones, and would be idle to release and request the resources repeatedly.

14

Deadlock Prevention (cont'd)

- **Denying the *No Preemption*:** Preempt the resources of the requesting process or the other hold-and-waiting process:

(A) Preempt the resources of requesting process: If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by the requesting process are released.

- – Preempted resources are added to the list of resources the requesting process is waiting for.
- – The requesting process will be resumed only when it can regain its old resources, as well as the new ones that it is requesting.

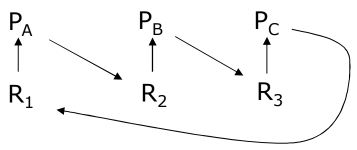
(B) Preempt the resources of hold-and-waiting process: If a process requests some resources which are allocated to some other process that is waiting for another resource, then preempt the desired resources from the hold-and waiting process and allocate them to the requesting process.

15

Deadlock Prevention (cont'd)

- **Denying the Circular Wait:** impose a total ordering of all resource types; that is, we assign a unique integer number to each resource type. Then, each process must request the resources in ascending order of their numbers.
  - – If several instances of the same resource type are needed, a single request for all those must be issued.

Proof by contradiction



Suppose that N_i denotes the unique integer value assigned to resource type R_i. If a circular wait exists as shown, then $N_1 < N_2 < N3 < N_1$, which cannot be true.

16

Deadlock Avoidance

- For each request made by a process, the system considers the resource allocation state to decide if the request can be granted or the process must wait to avoid a possible future deadlock.
  - Pretend the current request is granted, then examine the post-grant resource allocation state to ensure that there is a way all the processes can complete even though each process requests its maximum remaining need. If not, the current request is rejected.
- Requires each process to declare the *maximum number of instances* of each resource type that it may need.
  - If a process P_2 needs 3 instances of R_1 at time t_0 and 4 instances of R_1 later at time t_1, then P_2 needs maximum 7 instances of R_1.
- When deadlock avoidance is used, resource allocation state is defined by the number of available and allocated resources, and the maximum remaining need of the processes.

17

Safe State

- When a process requests available resources, system must decide if immediate allocation leaves the system in safe state or not.
- The system is in safe state if there exists a safe sequence $<P_1, P_2, \ldots, P_n>$ of all the processes in the systems, such that for each P_i, the resources that P_i may request (up to its maximum remaining need) can be satisfied by currently available resources plus the resources held by all P_j with $j < i$ (i.e., P_j comes before P_i in the safe sequence).
  - So, **a safe sequence is like a completion sequence in the worst case scenario.**
- That is:
  - If P_i's needed resources are not immediately available, then P_i can wait until all P_j (where $j < i$) have finished.
  - When all P_j (where $j < i$) are 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.
- If there is no such safe sequence of processes, the system is in unsafe state.

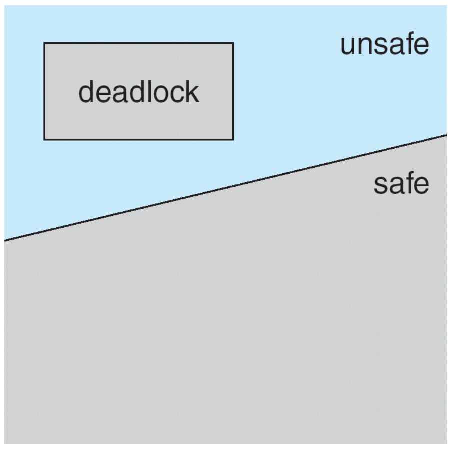
18

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
  - The OS cannot prevent processes from requesting resources in such a way that a deadlock occurs.
- Deadlock avoidance algorithm ensures that a system will never enter unsafe state.

19

Safe, Unsafe, and Deadlock State



20

Avoidance Algorithms

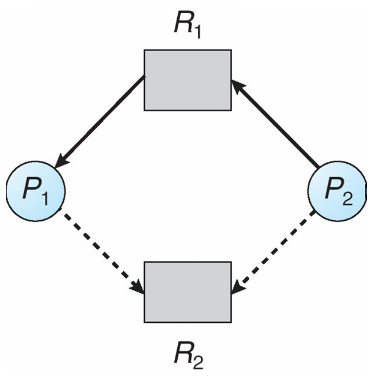
1. Single instance of each resource type.
   – Use a resource allocation graph.

2. Multiple instances of a resource type.
   – Use the Banker's algorithm.

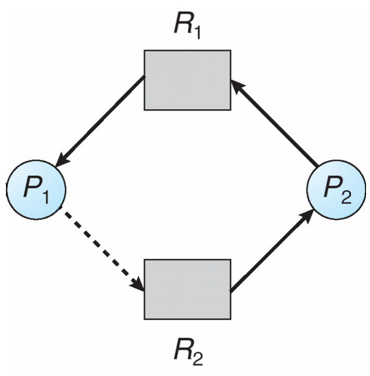
Single Instance of Each Resource Type
(Using the Resource Allocation Graph)

- A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j sometime in the future, and it is represented by a dashed line.
- A claim edge is converted to a request edge when the process requests a resource.
- If process P_i requests a resource R_j, the request can be granted only if converting the request edge to an assignment edge does not result in a cycle (including other claim edges) in the resource allocation graph.
- A request edge is converted to an assignment edge when the resource is actually allocated to the process.
- When a resource is released by a process, the assignment edge is reconverted to a claim edge.

Example



resource-allocation graph



If P_2 requests R_2, the post-grant state is unsafe. So, this request would be denied.

23

Banker's Algorithm

- Each resource type may have multiple instances.
- When a process enters the system, it must declare the maximum number of instances of each resource type it may need.
- Whenever a process requests a resource that is currently available, the system must decide if the resource can be assigned immediately or if the process must wait.
  – The request is granted only if it leaves the system in safe state.
- When a process gets all its requested resources, it must return them (obviously after using them) within a finite amount of time.

24

Data Structures for the Banker's Algorithm

- **Available**: Vector of length *m*, where *m* is the number of resource types. If *Available*[*j*] = *k*, there are *k* available instances of resource type R_j.
- **Max**: *n x m* matrix, where *n* is the number of processes. If *Max*[*i, j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.
  - If a process P_2 needs 3 instances of R_1 at time t_0 and 4 instances of R_1 later at time t_1, then *Max*[2, 1]= 7.
- **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, thus

Need[*i, j*] = *Max*[*i, j*] – *Allocation*[*i, j*]

25

Data Structures for the Banker's Algorithm (cont'd)

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

	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

- *Need* is defined as (*Max* – *Allocation*).

26

Notations

- *Allocation*$_i$: *i*-th row of matrix *Allocation*.

  Need$_i$: *i*-th row of matrix *Need*.

  Request$_i$: a vector (of length *m*) representing the current request of process P_i .

- X, Y: vectors of length *m*

  X ≤ Y iff X[i] ≤ Y[i] for all i= 1, 2, …, m.

  e.g., if X= (0, 3, 2), Y=(1, 7, 2), then X ≤ Y.

27

Banker's Algorithm

If process P_i wants *k* instances of resource type R_j , then

$Request_i[\ j\ ] = k$.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, we have an error because a process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait because the requested resources are not available.
3. Pretend to have allocated the requested resources to P_i by tentatively modifying the resource allocation state as follows:

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

4. Run the Safety algorithm:
   – if safe, then actually allocate the requested resources to P_i .
   – if unsafe, then P_i must wait for $Request_i$, and the previous resource allocation state is restored.

28

Safety Algorithm

1. Let *Work* be a vector of length *m,* and *Finish* be a vector of length *n*, and initialize them as:

   Work = *Available*

   Finish[*i*] = *false* for *i* = 1, 2, …, n.

2. Find any *i* such that:

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

   (b) $Need_i \le Work$

   If no such *i* exists, go to step 4.

3. $Work = Work - Need_i + (Need_i + Allocation_i)$

   $= Work + Allocation_i$

   Finish[*i*] = *true*

   Go to step 2

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

29

Example of the Safety Algorithm

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

	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

- *Need* is defined as (*Max* – *Allocation*).
- The system is in safe state because a safe sequence < P_1, P_3, P_4, P_2, P_0> exists.

30

Example of BA: P_1 Requests (1, 0, 2)

- Check if *Request*$_1$ (1, 0, 2) ≤ *Need*$_1$ (1, 2, 2).
- Since *Request*$_1$ (1, 0, 2) ≤ *Available* (3, 3, 2), we pretend this request is granted, and the post-grant state is created as follows:

	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 the safety algorithm shows that a safe sequence < P_1, P_3, P_4, P_0, P_2> exists in the post-grant state. So, P_1's request would be granted.

31

Deadlock Detection

- Allow the system to enter a deadlock state.
- Use a deadlock detection algorithm to detect a deadlock.
  1. If each resource has a single instance, we can check if there is a cycle in the wait-for graph (or in the resource allocation graph).
  2. If some resource has multiple instances, we can use a detection algorithm which is similar to the Banker's algorithm.
- Use a deadlock recovery scheme if there is a deadlock.

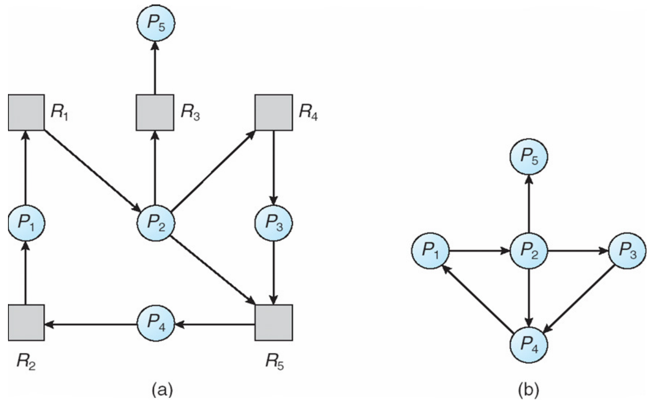
32

Single Instance of Each Resource Type

- Maintain a *wait-for graph* in which
  - Nodes represents the processes.
  - ($P_i \rightarrow P_j$) exists if process P_i is waiting for process P_j to release a resource.
- A wait-for graph is obtained from the resource allocation graph by removing the nodes of resource types and collapsing the request edge and the assignment edge for each resource type:
  - For example, ($P_i \rightarrow P_j$) exists in the wait-for graph iff the corresponding resource allocation graph contains two edges ($P_i \rightarrow R_q$) and ($R_q \rightarrow P_j$) for some resource type R_q.
- Periodically invoke an algorithm that searches for a cycle in the wait-for graph. If there is a cycle, a deadlock exists.

33

Resource Allocation Graph and Wait-for Graph



Resource allocation graph **Corresponding wait-for graph**

34

Several Instances of a Resource Type

Data Structures:

- **Available***:* A vector of length *m* indicating the number of available instances of each resource type.
- **Allocation***:* An *n* x *m* matrix indicating the number of instances of each resource type currently allocated to each process.
- **Request***:* An *n* x *m* matrix indicating 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.
- Maximum need of each process is not required because we don't care about the remaining need (i.e., future request) of each process.

35

Deadlock Detection Algorithm

Idea: A process is not deadlocked if its current request can be satisfied by using the available resources and the resources to be released by other processes (that can complete).

1. Let *Work* be a vector of length *m,* and *Finish* be a vector of length *n*, and initialize them as:
   (a) *Work* = *Available*
   (b) For *i* = 1, 2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[*i*] = false; otherwise, *Finish*[*i*] = *true* (i.e., not part of a deadlock).
2. Find an index *i* such that:
   (a) *Finish*[*i*] == *false*, and
   (b) $Request_i \leq Work$
   If no such *i* exists, go to step 4.

36

Deadlock Detection Algorithm (cont'd)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2
   – Note: we assume that P_i can be done eventually and return the resources currently allocated to it. If this is not the case, such that the process will be involved in a deadlock later, the deadlock will be detected after that.

4. If *Finish*[*i*] == false for some *i*, $1 \leq i \leq n$, then the system is in a deadlock state.
   – If *Finish*[*i*] == *false*, then P_i is deadlocked

37

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*. That means, it is a completion sequence in the deadlock detection algorithm. So, there is no deadlock.

38

Example of Detection Algorithm (cont'd)

- If P_2 requests an additional instance of type C, the state of system becomes as follows:

	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

– Can reclaim resources held by process P_0, but insufficient resources to satisfy any other process' request.

– So, a deadlock exists, consisting of processes P_1, P_2, P_3 and P_4.

39

Usage of a Deadlock Detection Algorithm

- When should we invoke the deadlock detection algorithm?
  1. Every time a request for a resource cannot be immediately granted.
     - Considerable overhead in computation time to run the deadlock detection algorithm.
  2. At certain time intervals.
  3. If CPU utilization is lower than certain value
     - Once a deadlock happens, it will grow as more and more processes and their resources are included in the deadlock, and eventually CPU utilization will drop as the number of processes that can use the CPU is reduced.

40

Recovery from a Deadlock: Process Termination

- Abort all deadlocked processes, or abort one process at a time until the deadlock is eliminated.
- To choose a process to terminate, the following factors can be considered:
  - Priority of the process.
  - How long the process has computed, and how much longer the process will compute before completion.
  - How many and what types of resources the process has used.
  - How many more resources the process needs to complete.
  - Whether the process is interactive or batch.

41

Recovery from Deadlock: Resource Preemption

- Selecting a victim process whose resource will be preempted:
  - We need to determine the order of preemption to minimize the cost.
- Roll back the selected victim process:
  - Roll back the process only as far as necessary to break the deadlock is desirable, but it requires the system to keep track of information about the state of all running processes. In general, we cannot roll back a process to a certain point in the past.
- Possible starvation of a process:
  - A process may be picked as a victim again and again.

42