Operating Systems - Deadlock

Mridul Sankar Barik

Jadavpur University

2024

System Model I

- A system consists of finite number of resources to be distributed among a number of cooperating processes
  - Resource types $R_1, R_2, \ldots, R_n$
    - CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
- A process may utilize a resource in the following sequence
  - Request
  - Use
  - Release
- A system table records whether each resource is free or allocated and if allocated, to which process
- If a process requests for a resource which is already allocated to another process then it is added to queue of waiting processes waiting for the resource

System Model II

- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set
- Events corresponds to the acquisition and release of resources
- Resources may be
  - Physical
    - Printers, memory space, CPU cycles etc.
  - Logical
    - Files, semaphores, monitors etc.
- Example1
  - A system has three tape drives
  - Three processes hold one of these tape drives
  - If each process now requests another tape drive, then deadlock occurs
- Example2
  - A system has one printer and one tape drive
  - Process P_i is holding the tape drive and P_j is holding the printer
  - If P_i requests the printer and P_j requests the tape drive, deadlock occurs

Deadlock Characterization

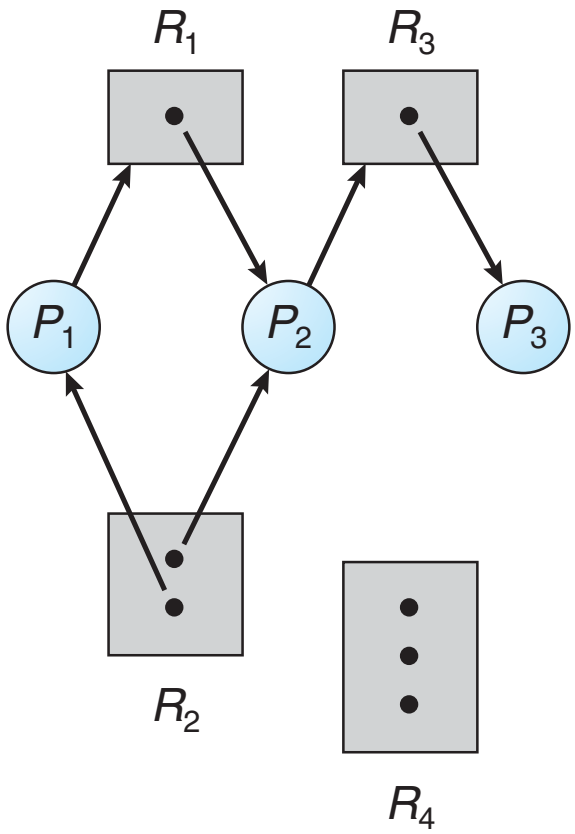
- Deadlock can arise if four conditions hold simultaneously
  1. **Mutual exclusion**: only one process at a time can use a resource
  2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
  3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  4. **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_{n-1}\}$ 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, ..., Pn–1 is waiting for a resource that is held by P_0.

Resource-Allocation Graph I

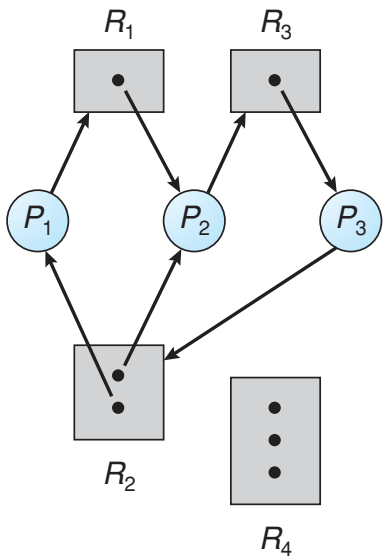
- Deadlocks can be described by a directed graph called system resource allocation graph
  - A set of vertices V and a set of edges E
- V is partitioned into two types
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Request edge
  - A directed edge $(P_i \rightarrow R_j)$ from process P_i to resource type R_j
  - Process P_i has requested an instance of resource type R_j and is currently waiting for that resource
- Assignment edge
  - A directed edge $(R_j \rightarrow P_i)$ from resource type R_j to process P_i
  - An instance of resource type R_j has been allocated to process P_i

Resource-Allocation Graph II

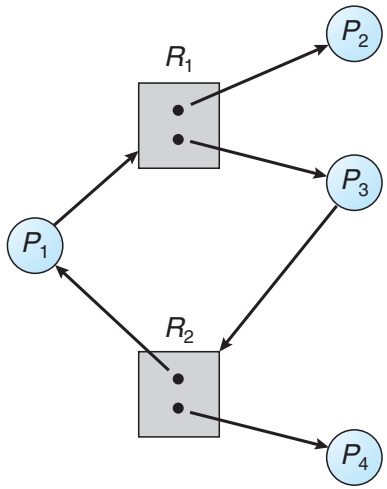
- Example of a Resource-Allocation Graph



Resource-Allocation Graph III



Resource-Allocation Graph With A Deadlock



Resource-Allocation Graph With A Cycle But No Deadlock

- If graph contains no cycles then 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 of Handling Deadlock

Deadlock problem can be solved in **three ways**

1. Ensure that the system will never enter a deadlock state
   - Deadlock Prevention
     - Ensuring that at least one of the necessary condition for deadlock cannot hold
   - Deadlock Avoidance
     - OS requires advance additional information concerning which resources a process will request in its lifetime
2. Allow the system to enter a deadlock state and then recover, the system must provide
   - An algorithm that examines whether deadlock has occurred
   - An algorithm to recover from the deadlock
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention I

Ensure that at least one of the four conditions of deadlock cannot hold

- Mutual Exclusion
  - Not required for sharable resources; must hold for non-sharable 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
  - Low resource utilization; starvation possible
- No Preemption
  - If a process that is holding some resources and 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

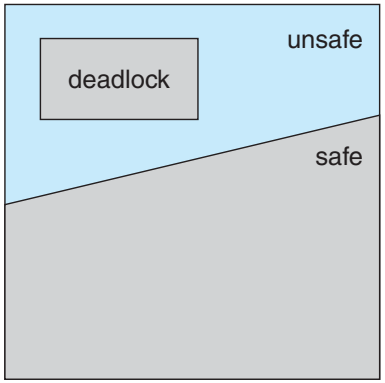
Deadlock Prevention II

- Circular Wait
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
    - Let $R = R_1, R_2, \ldots, R_m$ be the set of resource types
    - Assign each resource type a unique integer number using a one-to-one function $F : R \rightarrow N$ where N is the set of natural numbers
    - If process P_i is using R_i then it can request another resource R_j if $F(R_j) > F(R_i)$
  - Proof
    - Assume that circular wait exists
    - Let the set of processes involved in circular wait be $\{P_0, P_1, \ldots, P_n\}$ where P_i is waiting for resource R_i which is currently held by process P_{i+1}
    - Since process P_{i+1} is holding resource R_i while requesting resource R_{i+1} then we must have $F(R_i) < F(R_{i+1})$ for all i
    - This means $F(R_0) < F(R_1) < F(R_2) < \ldots < F(R_n) < F(R_0)$, which is impossible
    - Therefore there can be no circular wait

Deadlock Avoidance I

- Requires that the system has some additional apriori 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 P_i's resource needs are not immediately available, then P_i can wait until all P_j have finished
    - When they 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

Deadlock Avoidance II



- 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

- Example: Consider a system with 12 magnetic tape drives and 3 processes: P_0, P_1, P_2

	Max Needs	Current Allocation
P_0	10	5
P_1	4	2
P_2	9	2

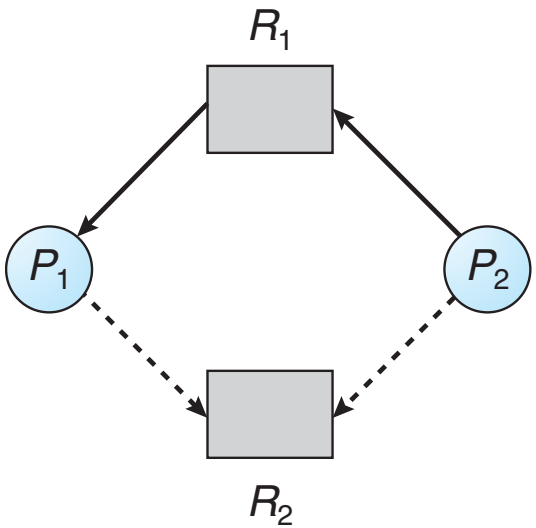
- At time t_0 process process P_0 is holding 5 tape drives, process P_1 is holding 2 tape drives, and process P_2 is holding 2 tape drives
- At time $t0$,the system is in a safe state. The sequence $< P_1, P_0, P_2 >$ satisfies the safety condition
- Suppose, at time t_1 , process P_2 requests and is allocated 1 more tape drive. The system is now no longer in safe state.
- Deadlock avoidance algorithm ensures that system always remains in a safe state

Avoidance Algorithms

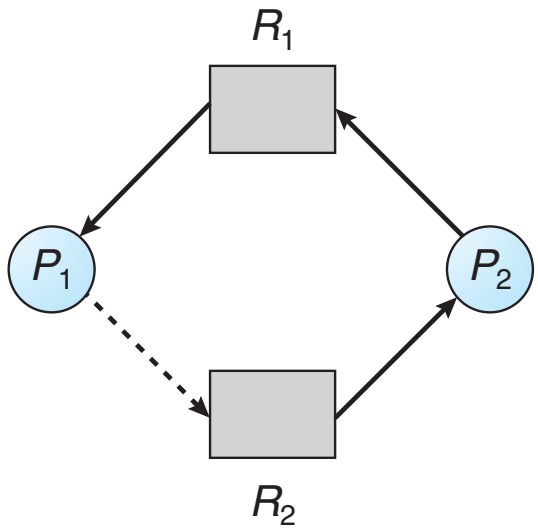
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Algorithm

- If we have a system with only one instance of each resource type, a variant of resource allocation graph can be used for deadlock avoidance
- In addition to request and assignment edges, a new edge is introduced
  - Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j sometime in future; 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
  - Suppose that process P_j requests resource R_j, The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $Rj \rightarrow P_i$ does not result in the formation of a cycle in the resource allocation graph



If P_2 requests R_2 and is allocated, deadlock occurs



Banker's Algorithm I

- Allows multiple instances of the same resource
- Each process must a priori declare maximum number of instances of each resource type that it may need
- When a process requests a set of resource the system must determine whether allocation will leave the system in a safe state
- When a process gets all its resources it must return them in a finite amount of time
- Data structures
  - 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

Banker's Algorithm II

Safety Algorithm

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

   $Work := Available$
   $Finish[i] = false$ for $i = 1, 3, \ldots, n$.

2. Find an i such that both:
   a. $Finish[i] = false$
   b. $Need_i \leq Work$

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.

Banker's Algorithm III

Resource-Request Algorithm

$Request_i$ = 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 the resulting resource-allocation state is 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 I

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	2			

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 of Banker's Algorithm II

Now, process P_1 requests resources (1, 0, 2). Check that $Request[1] \leq Available$ (that is, $(1,0,2) \leq (3,3,2)) \Rightarrow true$. Pretend that this request has been fulfilled, and the new state is:

	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.

Hence the request of process P_1 can be immediately granted.

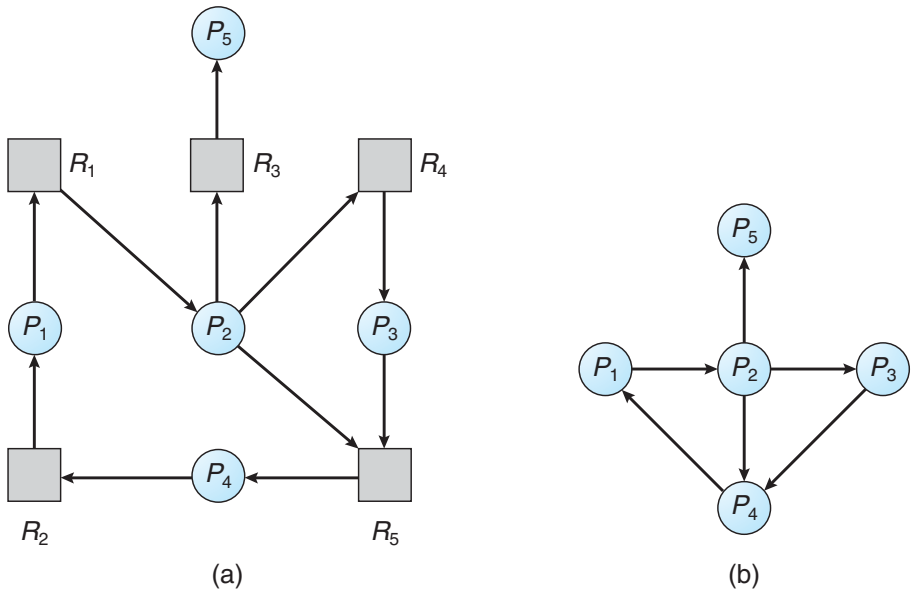
Can request for (3,3,0) by P_4 be granted?

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

Deadlock Detection I

Single instance of each resource type

- Maintain a wait-for graph Obtained from resource allocation graph by removing nodes of type resource and collapsing the appropriate edge
- $P_i \rightarrow P_j$ 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 an order of n^2 operations, where n is the number of vertices in the graph



(a) Resource-Allocation Graph
(b) Corresponding wait-for Graph

Deadlock Detection II

Several instances of each resource type

- Data Structures
  - *Available*: A vector of length m. It indicates the number of available resources of each type.
  - *Allocation*: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
  - *Request*: An $n \times m$ 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.

Deadlock Detection III

Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively Initialize $Work := Available$. 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. $Finish[i] = false$ and
   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.

This algorithm requires an order of $m \times n^2$ 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.

Now, 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.

Detection Algorithm Usage

- When, and how often, to invoke depends on two factors
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- In the extreme, detection algorithm is invoked every time a request for allocation cannot be granted immediately: incurs considerable time in computation overhead
- Invoke detection algorithm once per hour or when CPU utilization drops below 40 percent
- 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 I

- When a deadlock is detected several alternatives exist
  - Inform operator, let the operator deal with it manually
  - Let the system recover from deadlock automatically, two options
    - Process Termination
    - Resource Preemption
- Process Termination: abort one or more processes to break the deadlock and reclaim all the resources allocated to the aborted process
  - 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 II

- Resource Preemption: preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
  - Selecting a victim – minimize cost
  - Rollback – return to some safe state, restart process from that state
  - Starvation – same process may always be picked as victim, include number of rollback in cost factor