OPERATING SYSTEMS

Topic - Deadlock

Definition of Deadlock

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

Definition: **A deadlock** is a situation where a group of processes are permanently blocked as a result of each process having acquired a subset of resources needed for its completion and waiting for the release of the remaining resources held by other processes in the same group thus making it impossible for any of the process to proceed.

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.
  - CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.
  - If a process requests an instance of a resource type, the allocation of *any* instance of the type should satisfy the request. If it does not, then the instances are not identical.

System Model (cont.)

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system.

A process may utilize a resource in only the following sequence:

1. **Request.** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use.** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release.** The process releases the resource.

Deadlock Characterization

Four Necessary Conditions for the occurrence of deadlock

1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait.** A set $\{P_0, P_1, ..., P_{n-1}\}$ of waiting processes must exist such that P0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

Resource Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E.

The set of vertices V is partitioned into two different types of nodes:

- $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system and, each process P_i is pictorially represented *as a circle*
- $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system and *each resource type Rj* is pictorially represented *as a rectangle.*
  - *A resource type R_j may have more than one instance, we* represent each such instance as a dot within the rectangle.

Resource Allocation Graph (cont.)

Request Edge

- A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$
- It signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
- This directed edge $P_i \rightarrow R_j$ is called a request edge.

Assignment Edge

- A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$
- It signifies that an instance of resource type R_j has been allocated to process P_i .
- This directed edge $R_j \rightarrow P_i$ is called an assignment edge

Note: A request edge points to only the rectangle R_j, *whereas an assignment edge must also* designate one of the dots in the rectangle

Deadlock Prevention

Each of the four necessary conditions must hold for a deadlock to occur. The occurrence of a deadlock can be prevented by ensuring that at least one of these conditions cannot hold. This approach is elaborated by examining each of the four necessary conditions separately.

Mutual Exclusion: The mutual exclusion condition must hold. That is, at least one resource must be non-sharable.

- Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Deadlock Prevention (cont.)

Hold and Wait To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Deadlock Prevention (cont.)

No Preemption The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Deadlock Prevention (cont.)

Circular-wait The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

- To illustrate, we let $R = \{R_1, R_2, ..., R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:
  - F(tape drive) = 1
  - F(disk drive) = 5
  - F(printer) = 12

Deadlock Prevention (cont.)

- To prevent deadlocks, each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type say, R_i . After that, the process can request instances of resource type R_j *if* and only if $F(R_j) > F(R_i)$.
  - For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
- Alternatively, whenever a process requesting an instance of resource type R_j *it* must have released any resources R_i such that $F(R_i) \geq F(R_j)$.
  - *Note also that if several* instances of the same resource type are needed, a ***single request for all of them*** must be issued.
- If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Detection Single Instance Resource Type

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instances of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for graph.**

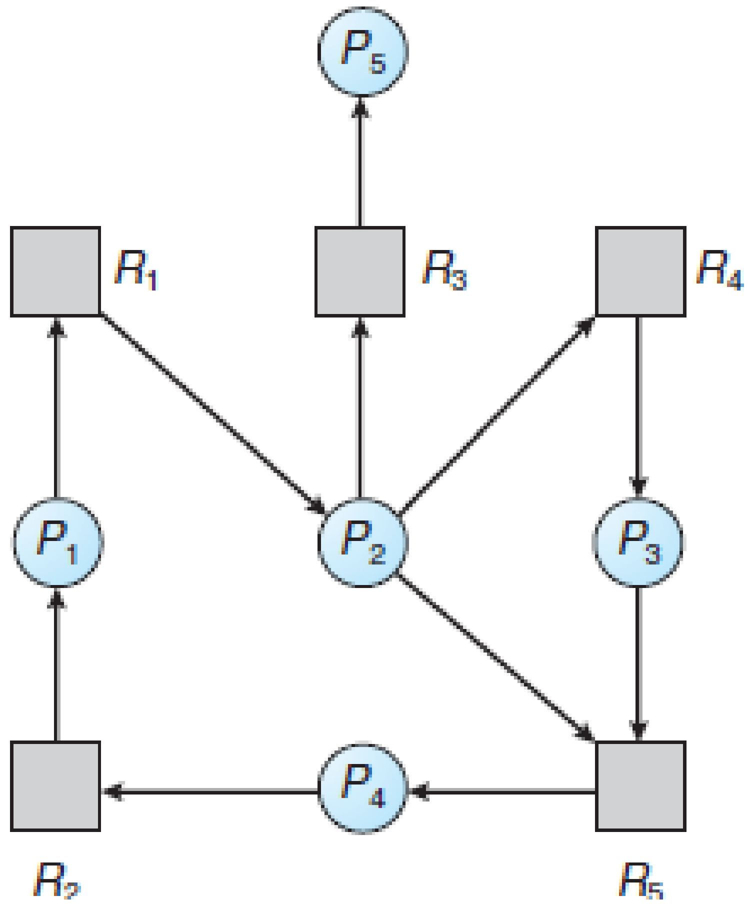
Wait graph is obtained from the corresponding resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

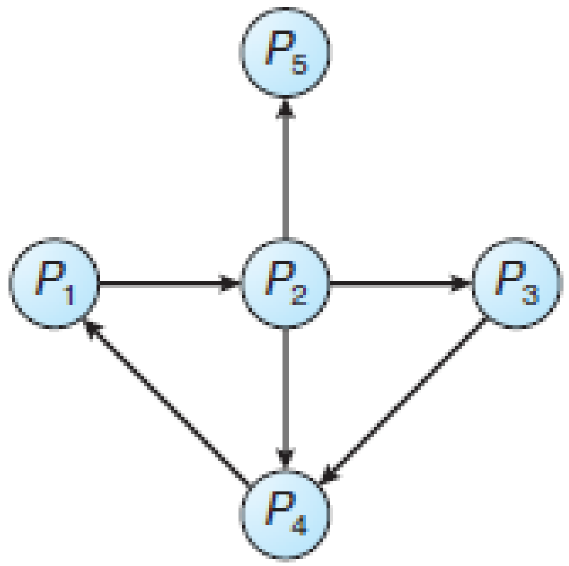
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q .

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait for graph and 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 n2 operations, where n is the number of vertices in the graph.

Resource Allocation graph along with corresponding wait graph





Several Instances of Each Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. A deadlock detection algorithm that is applicable to such a system employs several time-varying data structures

- **Available**: A vector of length m 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] equals k, then process P_i is requesting k more instances of resource type R_j .

Deadlock Detection Several Instances Resource Type

The ≤ relation between two vectors is defined as follows:

- Let *X and Y* be vectors of length *n.*
- $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i = 1, 2, ..., n.
- For example*, if* $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$*.*
- *In addition,* $Y < X$ *if* $Y \leq X$ *and* $Y \neq X$*.*

To simplify notation,

- Each row in the matrices ***Allocation and Request*** is treated as vectors and refer to them as $\textbf{\textit{Allocation}}_i$ ***and*** $\textbf{\textit{Request}}_i$ **.**
- The vector $\textbf{\textit{Allocation}}_i$ specifies the resources currently allocated to process *Pi*
- The vector $\textbf{\textit{Request}}_i$ specifies the current request of process *Pi to* complete its task.

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

Deadlock Detection Several Instances Resource Type

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

   Initialize ***Work*** = ***Available***.

   For i = 0, 1, ..., n–1,

   if $\boldsymbol{Allocation_i} \neq 0$, *then* ***Finish[i]*** = false *otherwise* ***Finish[i]*** = true.

2. Find an index *i such that both*

   a. ***Finish[i]*** == false

   b. $\boldsymbol{Request_i} \leq \boldsymbol{Work}$

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

3. ***Work*** =***Work*** + $\boldsymbol{Allocation_i}$***;***

   Finish[i] = true

   Go to step 2.

4. If ***Finish[i]*** ==false for some *i*, $1 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish[i]*** == false, then process $\boldsymbol{P_i}$ is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances.

Suppose that, at time T_0, we have the following resource-allocation state:

	Allocation *ABC*	Request ABC	Available ABC
P0	010	000	000
P1	200	202	
P2	303	000	
P3	211	100	
P4	002	002	

Work ABC	*FINESH*	
010	TRUE	(after 1st iteration of Step-2 & 3)
513	TRUE	(after 3rd iteration of Step-2 & 3)
313	TRUE	(after 2nd iteration of Step-2 & 3)
724	TRUE	(after 4th iteration of Step-2 & 3)
726	TRUE	(after 5th iteration of Step-2 & 3)

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <P0, P2, P1, P3, P4> results in the condition "Finish[i] == true for all i"

Suppose now that process *P2 makes one additional request for an instance* of type *C. The* ***Request matrix is modified as follows*****:**

.

	Allocation *ABC*	Request ABC	Available ABC
P0	010	000	000
P1	200	202	
P2	303	001	
P3	211	100	
P4	002	002	

Work ABC	*FINSH*	
010	TRUE	(after 1st iteration of Step-2 & 3)
	FALSE	
	FALSE	
	FALSE	
	FALSE	

Although we can reclaim the resources held by process *P0, the number of available resources is not sufficient* to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *P1, P2, P3, and P4.*

Deadlock Avoidance

- **Available:** A vector of length ***m*** indicates the number of available resources of each type. If ***Available*****[j]** equals ***k***, then ***k*** instances of resource type $\boldsymbol{R_j}$ are available.
- **Max:** An ***n* × *m*** matrix defines the maximum demand of each process**.** If ***Max[i][j]*** equals ***k,*** then process $\boldsymbol{P_i}$ may request at most ***k*** instances of resource type R_j .
- **Allocation:** An ***n* × *m*** matrix defines the number of resources of each type currently allocated to each process. If ***Allocation*****[i][j]** equals ***k,*** then process *Pi* is currently allocated *k instances of resource type* R_j .
- **Need.: An *n* × *m*** matrix indicates the remaining resource need of each process. If ***Need*****[*i*][j]** equals ***k,*** then process $\boldsymbol{P_i}$ ***may need k more instances*** of resource type *Rj to complete its task. Note that* ***Need*****[i][j]** equals ***Max* [*i*][j]** − ***Allocation*****[*i*][j].**

Deadlock Avoidance

- These data structures vary over time in both size and value.
- To simplify the presentation of the banker's algorithm, we next establish some notation.
- Let *X and Y be vectors of length n. We say that X ≤ Y if and* only if *X[i] ≤ Y[i] for all i = 1, 2, ..., n. For example, if X = (1,7,3,2) and Y =* (0,3,2,1), then *Y ≤ X. In addition, Y < X if Y ≤ X and Y = X.*
- We can treat each row in the matrices ***Allocation and Need as vectors*** and refer to them as ***Allocation$_i$ and Need$_i$. The vector Allocationi specifies*** the resources currently allocated to process *Pi ; the vector* ***Need$_i$ specifies the*** additional resources that process *P$_i$ may still request to complete its task.*

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state and described as follows:

1. Let ***Work and Finish*** be vectors of length ***m*** and ***n,*** respectively. Initialize ***Work*** = ***Available and Finish[i] = false for i = 0, 1, …, n – 1.***
2. **Find an index *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.***

This algorithm may require an order of *$m \times n^2$ operations to determine whether* a state is safe

Let ***Request***$_i$ be the request vector for process P_i . If Request$_i$ [j] == k, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1. If Request$_i$ ≤ Need$_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request$_i$ ≤ Available, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process 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, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i *must wait for* ***Request***$_i$ ***,*** and the old resource-allocation state is restored.

Illustrating Banker’s Algorithm

Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	Allocation ABC	Max ABC	Available ABC
P0	010	753	332
P1	200	322	
P2	302	902	
P3	211	222	
P4	002	433	

Need ABC
743
122
600
011
431

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One such alternative is to recover the system from the deadlock automatically. There are two options for breaking a deadlock.

- Abort one or more processes to break the circular wait.
- Preempt some resources from one or more of the deadlocked processes

Deadlock Recovery Through Process Termination

Process Termination

Two methods can be used to eliminate deadlocks by aborting a process. In both methods, the system reclaims all resources allocated to the terminated processes.

Method-I Abort all deadlocked processes.

- ✓ This clearly will break the deadlock cycle,
- ✘ At great expense as some process may be nearing completion.

Method –II Abort one process at a time until the deadlock cycle is eliminated.

- ✘ incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Deadlock Recovery Through Process Termination (cont.)

Factors that determines the selection of process which is to be aborted:

- What the priority of the process is?
- How long the process has computed and how much longer the process will compute before completing its designated task?
- How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
- How many more resources the process needs in order to complete
- How many processes will need to be terminated?
- Whether the process is interactive or batch?

Deadlock Recovery Using Resource Preemption

Successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Here three issues are need to be addressed:

1. **Selecting a victim**. Which resources and which processes are to be preempted?
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? The roll back the process to some safe state and restart it from that state.
3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?