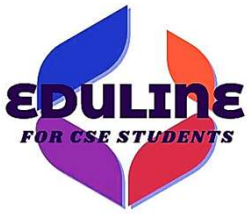
MODULE 3

CHAPTER 2 – DEAD LOCKS



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 wait state.
- Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources consist of some number of identical instances.
- Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types.
- If a system has two CPUs, then the resource type CPU has two instances.
- Under the normal mode of operation, a process may utilize a resource in only the following sequence:

❖**Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

❖**Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

❖**Release:** The process releases the resource.

CONDITIONS FOR DEDALOCK

➢A deadlock situation can arise if the following four conditions hold simultaneously in a system:

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 {P0, P1 , . . ., Pn } of waiting processes must exist such that Po is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn and finally Pn is waiting for a resource that is held by P0.

RESOURCE ALLOCATION GRAPH

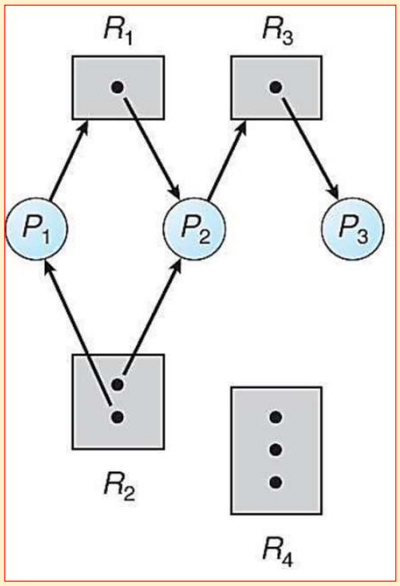
- A tool for recognizing deadlock is —Resource allocation graph (RAG). This graph consists of a set of vertices V and a set of edges E. Vertices are partitioned in to processes {P1, P2…..Pn} and resources {R1, R2…..Rm}. Its edges are of two types:

❖**Request edges:** This edge indicates the request of a process for acquiring a particular resource. It is a directed edge like Pi ⟶ Rj.

❖**Assignment edge:** This edge indicates the allocation of a resource to particular process. It is also a directed edge like Rj⟶Pi. This edge here indicates that an instance of resource Rj has been allocated to the process Pi.

- Circle represents processes and rectangles represent resources.

Resource-allocation graph



The sets P, R, and E:

- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- P3 is holding an instance of R3.

➢ If a cycle does not exist in RAG, then we can say that there is no deadlock exists.

➢ If a cycle exists in a RAG, there must be a possibility of deadlock.

➢ If a resource has only one instance and also a cycle exist, then deadlock must occur. In the case of multiple instances, the possibility of occurring deadlock is small.

- Now consider the following resource-allocation graph. In this example, we also have a cycle:

P1 → R1 → P3 → R2 → P1

However, there is no deadlock.

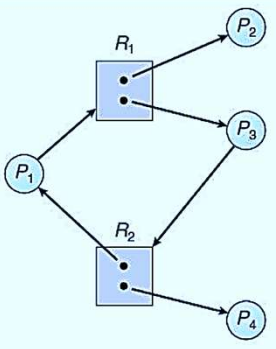


Figure 1: Resource-allocation graph with a cycle but no deadlock.

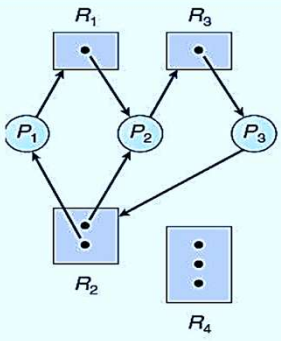


Figure 2: Resource allocation graph with deadlock.

- In figure1 ,P2 holds an instance of R1, and P4 holds an instance of R2. P2 and P4 can release their resources.
- If they do, then R1 and R2 will both have free instances, so there will be no deadlock, as those free instances can be assigned to P1 and P3, respectively, and the arrows will be reversed.
- The request edges will turn into assignment edges. Then the graph will be acyclic.

Multiple instances of resources	One instance of every resource
• Cyclic RAG need not always imply deadlock	• Cyclic RAG implies deadlock
• If there is a cycle in the RAG, it does not necessarily imply that a deadlock has occurred. Even though there is a cycle in the RAG, there is no deadlock.	• If the cycle in RAG involves only a set of resource types, each of which has a single instance, then deadlock must have occurred.

METHODS FOR HANDLING DEADLOCK

The four methods are:

- Prevention
- Avoidance
- Detection and recovery
- Ignore (used in UNIX, ignore and restart)

❖**DEADLOCK PREVENTION**

For a deadlock to occur, each of the four necessary conditions (mutual exclusion, Hold and wait, No preemption, Circular wait) must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock

1. Mutual Exclusion

- Make all the non-sharable resources sharable.
- The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes.
- Sharable resources, on the other hand, 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. We cannot prevent deadlocks by denying the mutual-exclusion condition because some resources are intrinsically non-sharable and mutual exclusion is one of the requirements of the critical section problems solution.

2. Hold and Wait

- To prevent Hold and wait condition from happening, we can have a rule that says
- A process may not request a resource if it is holding another resource.
- So, to take the print out of the contents of a file, you first request the disk, and then you get it, use it and release it. Then you request the printer, you get it, use it, and then you release it.
- Thus, it implies that a process should have released all its resources before it requests for additional resources.
- Disadvantage is starvation.

- Another rule can be there that "a process should request and acquire all the requested resources before its execution begins".
- For example, to take the print out of the contents of a file, the disk and printer should be requested beforehand.
- Disadvantage of this method is resource utilization will be low.

3. No Preemption

- To ensure that this condition does not happen, preemption of the resources should be allowed. Preemption can be added in two manners:

1. 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 currently being held 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.

2. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.

- If they are not available, we check whether they are allocated to some other process that is waiting for additional resources.
- If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

4. Circular Wait

- For avoiding circular wait we use numbering scheme. We can assign a number for all resources.
- So each process has requested in a particular order. If the set of resources types R includes tape drives, disk drives and printer, then the function F might be

F(tape drive)=1;

F(disk drive)=5;

F(printer)=12;

- A process can be request any resource based on the condition **f(Rj)>f(Ri).**

❖Ri: The resource number which is currently hold by the process Pi.

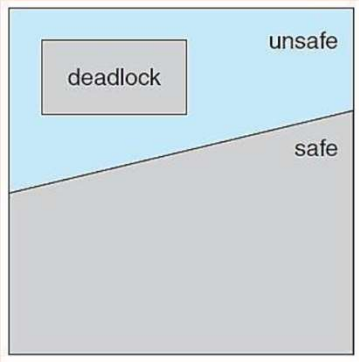
❖Rj: New request from process Pi.

- If a process holds the disk drive, it cannot be requested for a tape drive, because the number is less, but it can be give a request for printer, which has a greater number.

DEADLOCK AVOIDANCE

Safe State: A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A deadlock state is a form of unsafe state.

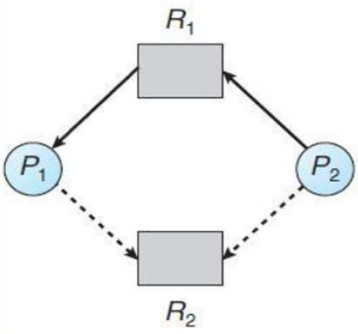
Safe, unsafe, and deadlocked state spaces

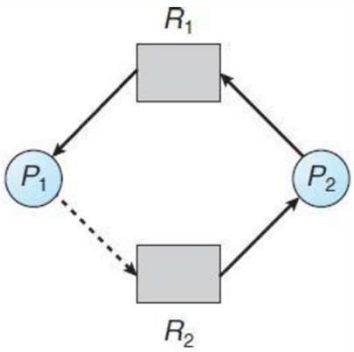


❖Resource-Allocation Graph Algorithm

- This algorithm is used for only single instance resource type.
- In addition to the request and assignment edge, a new type of edge called a **claim edge** is introduced.
- A claim edge **Pi ----> Rj** indicates that process Pi may request resource Rj at some time in the future.
- Claim edge is represented by a dashed line.
- When process Pi requests resource Rj, the claim edge Pi ---->Rj is converted to a request edge.
- Similarly, when a resource Rj is released by Pi the assignment edge Rj —> Pi is reconverted to a claim edge Pi ---->Rj

- Consider the resource allocation graph of fig(a). Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, Since this allocation will create a cycle in the graph (fig(b)).
- A cycle indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.





❖Banker's Algorithm

- This algorithm's purpose is deadlock avoidance.
- Several data structures must be maintained to implement the banker's algorithm.
- **Available:** A vector of length m indicates the number of available resources of each type.
- **Max:** The maximum demand of each process.
- **Allocation:** Ann x m matrix defines the number of resources of each type currently allocated to each process.
- **Need:** An n x m matrix indicates the remaining resource need of each process. **Need[i,j] = Max[i,j] - Allocation[i,j]**

Example:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2			
P1	2	0	0	3	2	2						
P2	3	0	2	9	0	2						
P3	2	1	1	2	2	2						
P4	0	0	2	4	3	3						

- First we find Need.

  Need = Max-Allocation

- Now compare need and available. First we take process P0. It is not selected because A need 7 but Available of A is 3. Second we take process P1. It is selected. Then the new

Available = (Available +Allocation)

= (3 3 2) + (2 0 0) = **(5 3 2)**

- Next we select P2. Compare its need and the new available. ie, between (6 0 0) and (5 3 2)
- Here A wants 6 but only 5 is available. So it is rejected.
- Next take P3 and compare its need and available.ie, between

  (0 1 1)and (5 3 2). It is accepted.

  Then new Available= (5 3 2)+(2 1 1)= **(7 4 3)**
- Next take P4 and compare with its need and available. ie, between (4 3 1) and (7 4 3). It is accepted.

  Then the new Available =(7 4 3)+(0 0 2)= **(7 4 5)**

- Next we take P0 and compare with its need and available. ie, between (7 4 3) and (7 4 5). It is accepted.

  Then new Available= (7 4 5)+ (0 1 0)= **(7 5 5)**
- Now we can take P2 and compare with its need and new available. ie, between (6 0 0) and(7 5 5).
- It is accepted. Then the new final available = (7 5 5)+(3 0 2)=(10,5,7)
- So the safe sequence is **P1, P3, P4, P0, P2.**

❖DEADLOCK DETECTION

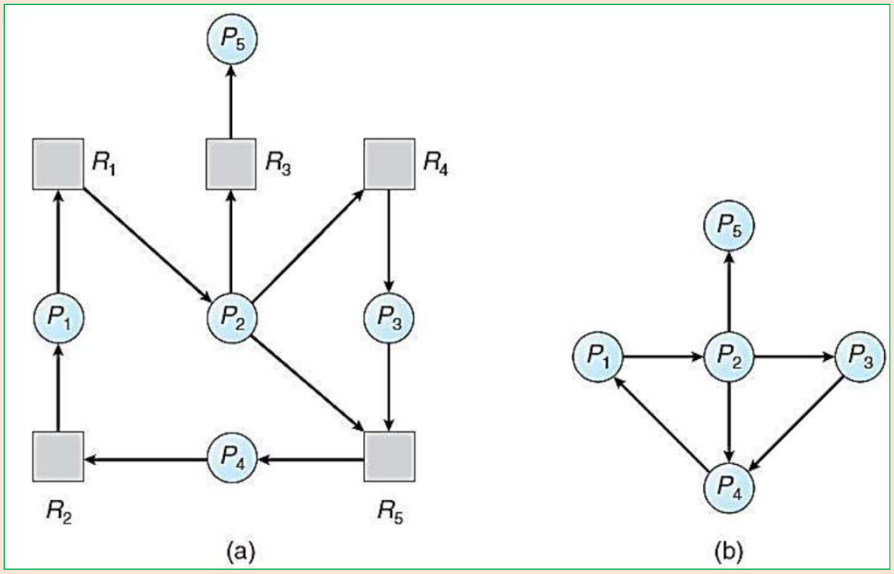
➢If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock
- Detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

➢Detection in Single Instance 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**.
- We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.
- If Pi ➡ Rj and Rj ➡ Pj, then Pi ➡ Pj.
- A deadlock exists in the system iff the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

Resource-allocation graph. (b) Corresponding wait-for graph



❖RECOVERY FROM DEADLOCK

- Recovery can be handled by manually or automatically.
- There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

➢Process Termination

- Here all processes are terminated which is in the deadlock state. Two methods for the process termination methods are:

✓Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at a great expense. If 5 processes are existed, these 5 processes must be terminated. So the used resources are wasted.

✓ Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Many factors may determine which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed, and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch

➢ **Resource Preemption**

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. The three conditions are:

1. **Selecting a victim:** Which resources and which processes are to be preempted?

   we must determine the order of preemption to minimize cost.

2. Rollback: If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state. Since, it is difficult to determine what a safe state is; the simplest solution is a total rollback: Abort the process and then restart it.

3. Starvation: In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation occurred. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.