UNIT-3

CHAPTER-2

DEADLOCKS

Unit-3-Chapter-2: Deadlocks

CONTENTS

DEADLOCK

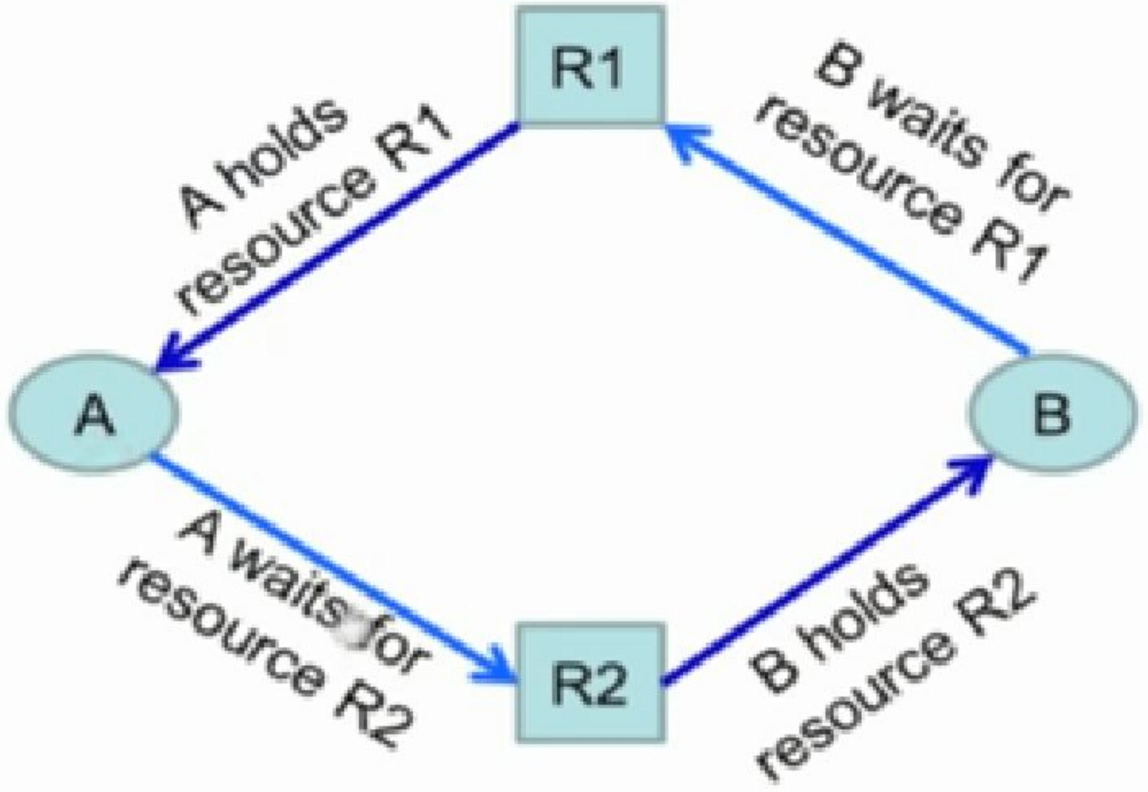
DEFNITION:

In a multiprogramming environment, several processes may compete for a finite number of resources.

A process requests resources; and 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”**

EXAMPLE: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Deadlocks


R1
A holds resource R1
B waits for resource R1
A
B
A waits for resource R2
R2
B holds resource R2

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.

EXAMPLE: Memory space, CPU cycles, files, and I/O devices (such as printers and DVD drives) etc.

If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

- 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.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**. 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.

- The request and release of resources are through system calls. Examples of system calls includes request () and release () device, open () and close () file, and allocate () and free () memory system calls.
- Request and release of resources that are not managed by the operating system can be accomplished through the wait () and signal () operations on semaphores or through acquisition and release of a mutex lock.
- For each use of a kernel managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource.
- A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

- 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. The events with which we are mainly concerned here are resource acquisition and release.
- The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).
- To illustrate a deadlock state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlock state. Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.
- Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process is holding the printer. If Pi requests the printer and Pi requests the DVD drive, a deadlock occurs.

Deadlock Characterization

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. In this section we discuss features that characterize deadlocks.

Necessary Conditions

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 held by P1, P1 is waiting for a resource held by P2 , ..., Pn-1 is waiting for a resource held by Pn , and Pn is waiting for a resource held by Po.

All four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

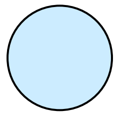
Resource-Allocation Graph

- 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 = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system.
  - R = {R1 , R2, ... , Rn}, the set consisting of all resource types in the system.

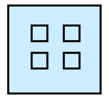
- **request edge** – directed edge $P_i \rightarrow R_j$ signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
- **assignment edge** – directed edge $R_j \rightarrow P_i$ signifies that an instance of resource type Rj has been allocated to process Pi.

Resource-Allocation Graph (Cont.)

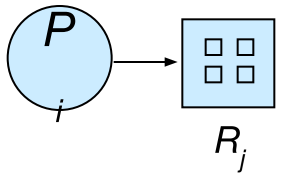
- Process



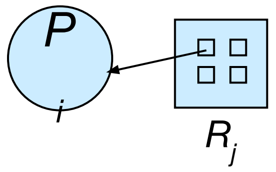
- Resource Type with 4 instances



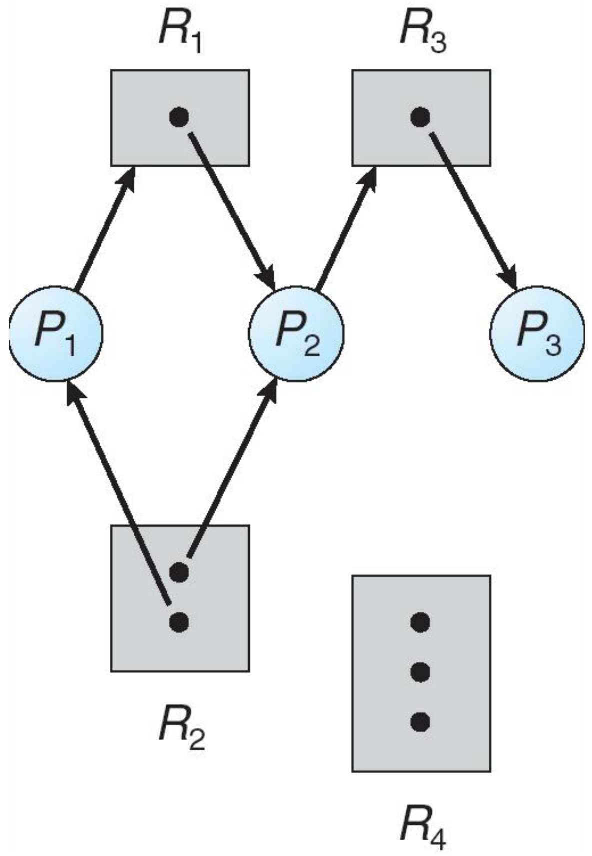
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



The resource-allocation graph shown in Figure depicts the following situation.

The sets P, R, and E:

- P = {PI, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 $\rightarrow$ R1, P2 $\rightarrow$ R3,R1 $\rightarrow$ P2 ,R2 $\rightarrow$ P2, R2 $\rightarrow$ P1, R3 $\rightarrow$ 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 Rj and an instance of R2 and is waiting for an instance of R3 .
- Process P3, is holding an instance of R3.

- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

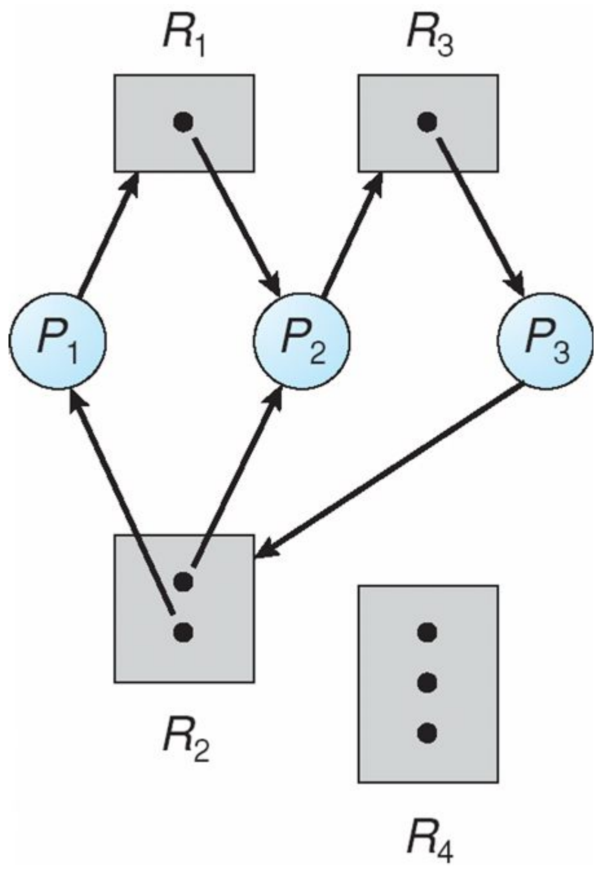
- Suppose that process P3 requests an instance of resource type R2, Since no resource instance is currently available, a request edge P3 □R2 is added to the graph.
- At this point, two minimal cycles exist in the system:

P1 $\rightarrow$ R1 $\rightarrow$ P2 $\rightarrow$ R3 $\rightarrow$ P3 $\rightarrow$ R2 $\rightarrow$ P1

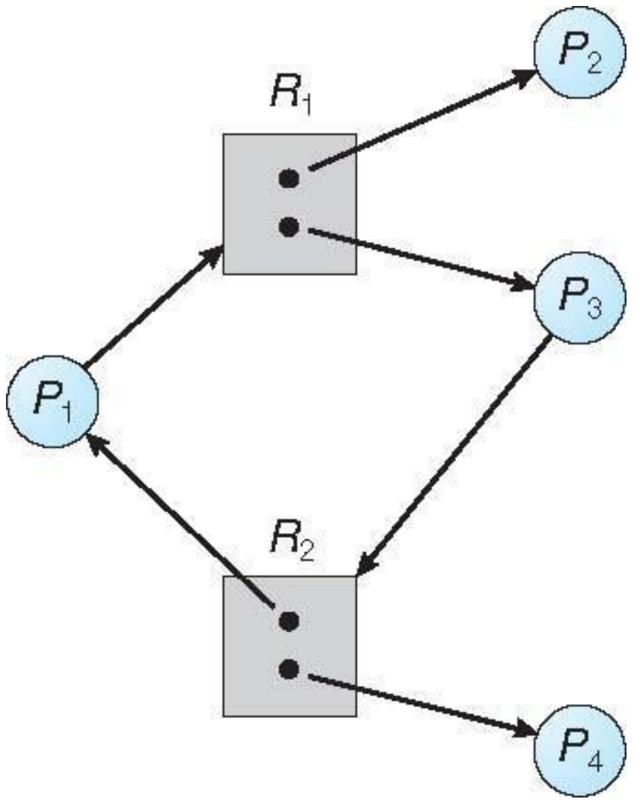
P2 $\rightarrow$ R3 $\rightarrow$ P3 $\rightarrow$ R2 $\rightarrow$ P2

Processes P1 , P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1

Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



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

$P1 \longrightarrow R1 \longrightarrow P3 \longrightarrow R2 \longrightarrow P1$

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2.That resource can then be allocated to P3 , breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem

Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

- we can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.
- Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).
- If a system neither ensures that a deadlock will never occur nor provides a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually

Deadlock prevention

- For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.
- Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. However, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

- 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.

- Two protocols can be used
  - Protocol-1:requires each process to request and be allocated all its resources before it begins execution.
  - Protocol-2: A Process request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated

Example:

- consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.
- Both these protocols have two main disadvantages:
  - Resource utilization may be low
  - starvation is possible.

No Preemption: The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.

- Following protocol can be used to ensure that no preemption condition will not occur.
- 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.
- 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.
- Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, 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. If the resources are neither available nor held by a waiting process, the requesting process must wait.

Circular Wait:

- 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 = {R1 , R2, ..., Rn} 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.
- Define a one-to-one function F: R □ N, where N is the set of natural numbers.
- 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

- we can consider the following protocol to prevent deadlocks:
- **Protocol-1:** 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, Rj. After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri)
- 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.
- **Protocol-2:** whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) ≥ F(Rj).
- If these two protocols are used, then the circular-wait condition cannot hold.

- We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction)
- Let the set of processes involved in the circular wait be { Po, P1 , ..., Pn}, where Pi is waiting for a resource Ri, which is held by process Pi +1 .
- Then, since process Pi+1 is holding resource Ri; while requesting resource Ri+1 , we must have F(Ri) < F(Ri+1), for all i. But this condition means that F(Ro) < F(R1) < ... < F(Rn) < F(Ro). By transitivity, F(Ro) < F(Ro), which is impossible. Therefore, there can be no circular wait.

Deadlock Avoidance

Requires that the 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

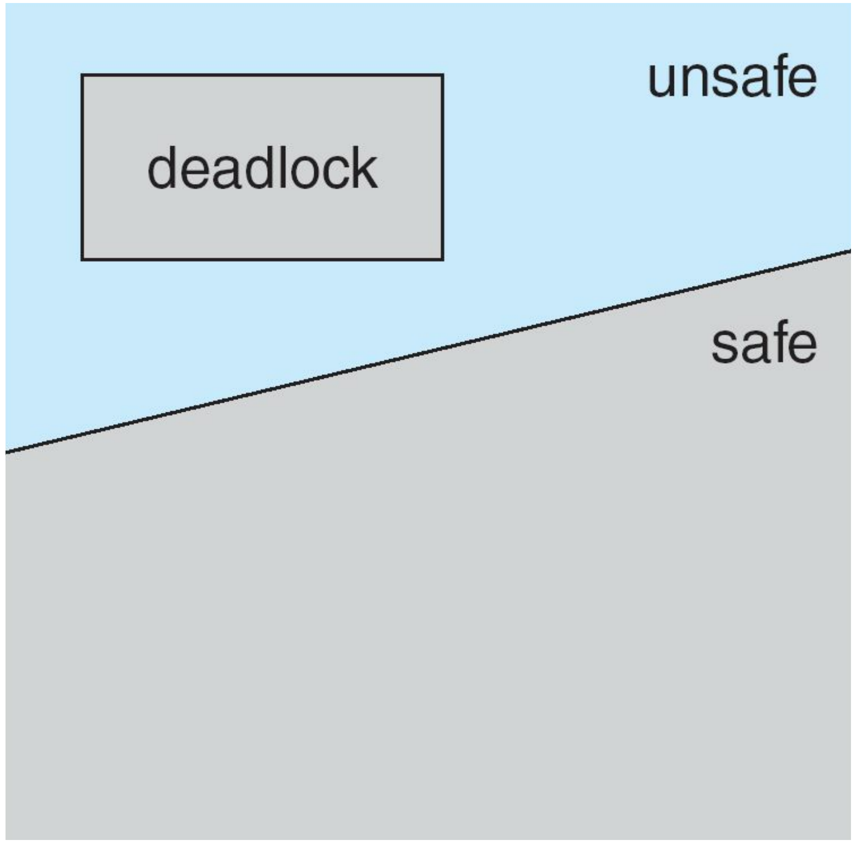
Safe State

- 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.
- 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 safety 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 can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j 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 .

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



Example:

consider a system with 12 magnetic tape drives and three processes: Po, P1, and P2 . Process Po requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives. Suppose that, at time *t0,* process Po is holding 5 tape drives, process P1 is holding 2 tape drives, and process P2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

processes	**Maximum needs**	**Current Allocation/Needs**
P0	10	5
P1	4	2
P2	9	2

At time to, the system is in a safe state. The sequence < P1, Po, P2 > satisfies the safety condition.

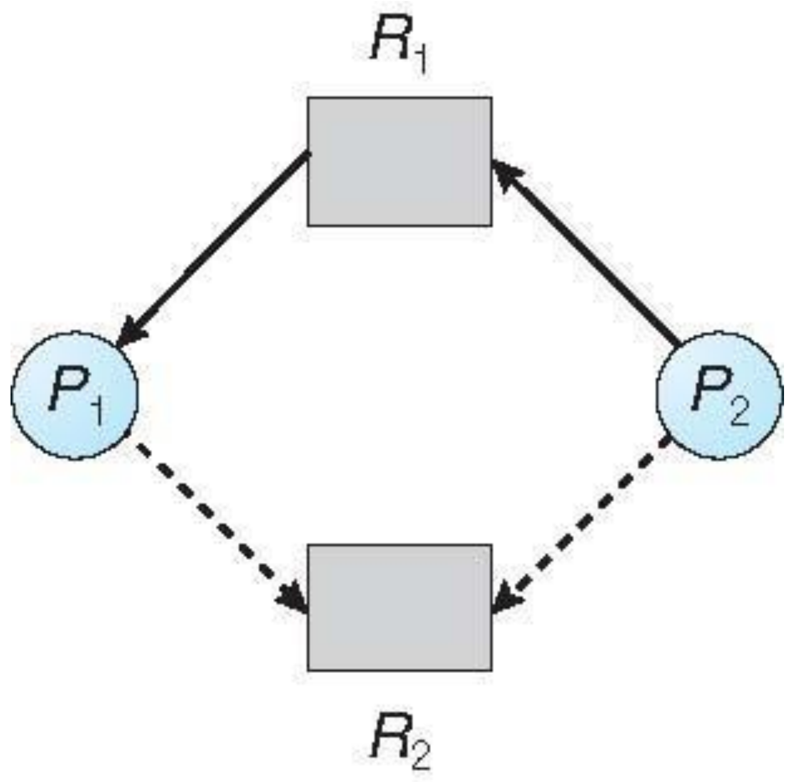
- Process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives);
- Process Po can get all its tape drives and return them (the system will then have 10 available tape drives);
- Finally process P2 can get all its tape drives and return them (the system will then have all 12 tape drives available).
- A system can go from a safe state to an unsafe state, If Suppose, at time t1, process P2 requests and is allocated with one more tape drive.
- At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives.
- Since process Po is allocated with 5 tape drives but it requires a maximum of 10 tape drives to finish its execution, it may request 5 more tape drives. Since they are unavailable, process Po must wait.
- Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.
- The problem associated here was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

- The key idea behind safety algorithm is to ensure that the system will always remain in a safe state.
- Initially the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

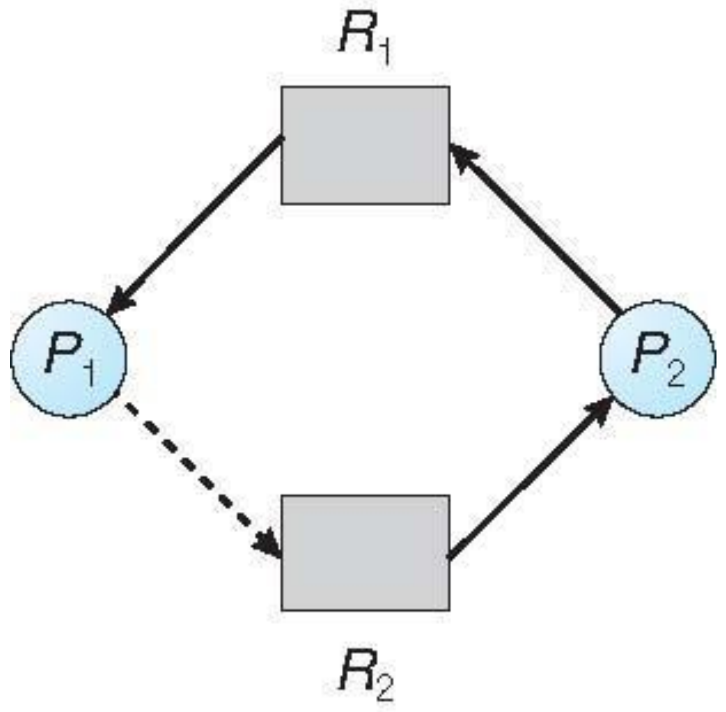
Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with only one instance of each resource type a variant of the resource-allocation graph discussed in the previous section can be used for deadlock avoidance.
- In addition to the request and assignment edges a new type of edge called a claim edge is introduced.
- A claim edge Pi $\longrightarrow$ Rj indicates that process Pi may request resource Rj at some time in the future.
- This edge resembles a request edge in direction but is represented in the graph 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
- Resources must be claimed a priori in the system. That is before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi ----> Rj to an assignment edge Rj ----> Pi does not result in the formation of a cycle in the resource-allocation graph.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found then the allocation will put the system in an unsafe state. Therefore, process Pi will have to wait for its requests to be satisfied

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- Banker's algorithm can be used for deadlock avoidance in case of a system with multiple instance of each resource type.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

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 x m matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **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

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

Safety Algorithm

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

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

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

   (b) ***Need**$_i$* **≤ *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

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i}$ **[*j*] = *k*** then process $\boldsymbol{P_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i} \leq \boldsymbol{Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $\boldsymbol{Request_i} \leq$ ***Available***, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ 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 $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 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:

	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	

Example (Cont.)

- 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: If P_1 Request (1,0,2) will the system be in safe state?

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	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

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

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

- You should be able to see however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted. since the resources are not available.
- Furthermore, a request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

Problems on Bankers Algorithm:

Problem-1: A single processor system has three resource types X, Y and Z, which are shared by three processes. There are 5 units of each resource type. Consider the following scenario, where the column Allocation denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST?

- P0
- P1
- P2
- None of the above since the system is in a deadlock

Allocation			Request			
	X	**Y**	**Z**	**X**	**Y**	**Z**
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

According to question-

Total = [X Y Z] = [5 5 5]

Total _Alloc = [X Y Z] = [5 4 3]

Now,

Available = Total – Total_Alloc

= [5 5 5] – [5 4 3]

= [0 1 2]

Step-01:

- With the instances available currently, only the requirement of the process P1 can be satisfied.
- So, process P1 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then, Available

= [0 1 2] + [2 0 1]

= [2 1 3]

- With the instances available currently, only the requirement of the process P0 can be satisfied.
- So, process P0 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [2 1 3] + [1 2 1]

= [3 3 4]

- With the instances available currently, the requirement of the process P2 can be satisfied.
- So, process P2 is allocated the requested resources.
- It completes its execution and then free up the instances of resources held by it.

Then-

Available

= [3 3 4] + [2 2 1]

= [5 5 5]

Thus,

There exists a safe sequence P1, P0, P2 in which all the processes can be executed.

So, the system is in a safe state.

Process P2 will be executed at last.

Thus, Option (C) is correct.

Problem-2: Assume there are 5 processes P0 through P4 and four types of resources. At time T0 we have the following state:

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Apply Bankers Algorithm to answer the following

(i) What is the content of Need Matrix?

(ii) Is the system in Safe state?

(iii) If a request from process P(0,4,2,0) arrives can it be granted?

Deadlock Detection

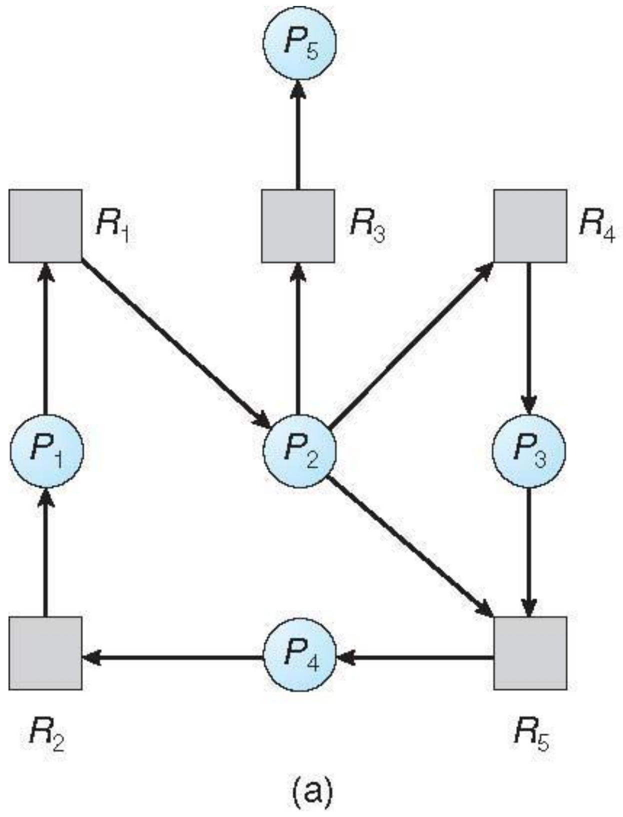
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

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

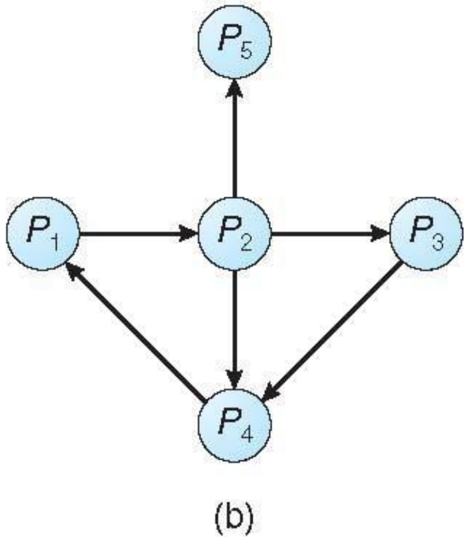
Single Instance of Each Resource Type

- 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 wait for graph.
- Wait for graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- An edge Pi $\longrightarrow$ Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi $\longrightarrow$ Rq and Rq $\longrightarrow$ Pj for some resource Rq.

Figure below shows the resource allocation graph and the corresponding wait for graph



Resource-Allocation Graph



Corresponding wait-for graph

- 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.

Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- Similar to bankers algorithm ,Following data structures are used by the algorithm:
  - **Available***:* A vector of length ***m*** indicates the number of available resources of each type
  - **Allocation***:* An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process
  - **Request***:* An ***n* x *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.

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *n***, if ***Allocation*** $_i$ **≠ 0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request*** $_i$ **≤ *Work***

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

Detection Algorithm (Cont.)

3. ***Work = Work + Allocation$_i$***
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P$_i$*** is deadlocked

Algorithm requires an order of O(*m* x *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***

Example (Cont.)

- $\boldsymbol{P_2}$ 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 $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- 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

When a detection algorithm determines that a deadlock exists, It can be handled in two ways:

- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal ,with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock:
  - One is simply to **abort one or more processes** to break the circular wait.
  - The other is to preempt some resources from one or more of the deadlocked processes.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes

- Abort one process at a time until the deadlock cycle is eliminated

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Thank You