# OPERATING SYSTEMS

Unit Objectives:

1. To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
2. To present a number of different methods for preventing or avoiding deadlocks in a computer system.

**Unit -4**

Deadlocks: System model; Deadlock characterization; Methods for handling deadlocks; Deadlock prevention; Deadlock avoidance; Deadlock detection and recovery from deadlock.

## System Model

- A System consists of a finite number of resources to be distributed among a number of competing processes.
- Resources are partitioned into several types each consisting of some number of identical instances.
- If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.
- The allocation of any instance of a resource type will satisfy the request of a process.
- A process must request a resource before using it and must release the resource after using it.
- Each process utilizes a resource as follows:
    - **Request** : The process requests the resource. If the resource cannot be granted immediately, then requesting process must wait until it can acquire the resource.
    - **Use** : The process can operate on the resource.
    - **Release** : The process releases the resource.
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Here events of concern are resource acquisition and resource release.
- The resources may be either physical (for example, printers, tape drives, memory space and CPU cycles) or logical (for example, files).

Illustration of a deadlocked 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 deadlocked state. Each is waiting for the event “CD RW is released”, which can be caused only by one of the other waiting processes.
- Consider a system with one printer and one DVD drive. Suppose that process $P_1$ is holding the DVD drive and process $P_2$ is holding the printer. If $P_1$ requests the printer and $P_2$ requests the DVD drive, a deadlock occurs.

## Deadlock Characterization

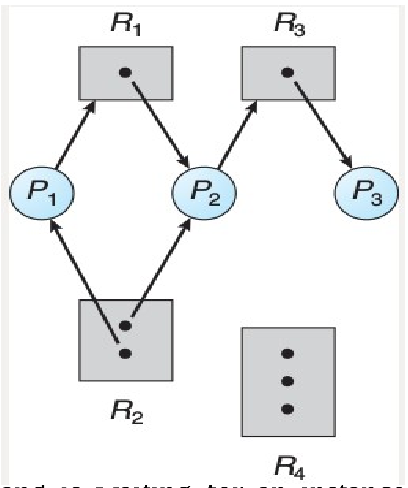
**Necessary Conditions: A Deadlock can arise if the following four conditions hold simultaneously.**

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; 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 is holding at least one resource and is waiting to acquire additional resources currently held by other processes.
3. **No preemption**: Resources cannot be preempted; 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 $\{P_0, P_1, \ldots, P_n\}$ 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 $P_2$, …, $P_{n-1}$ is waiting for a resource that is held by $P_n$, and $P_n$ is waiting for a resource that is held by $P_0$.

**Resource-Allocation Graph**

- A deadlock can be described more precisely by using a directed graph called resource-allocation graph.
- The graph consists of a set of vertices *V* and a set of edges *E*.
- The set 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.
- A directed edge from process $P_i$ to resource type $R_j$ is denoted by $\boldsymbol{P_i \rightarrow R_j}$; it signifies that process $P_i$ has requested an instance of resource type $R_j$ and is currently waiting for that resource.
- A directed edge from Resource type $R_j$ to process $P_i$ is denoted by $\boldsymbol{R_j \rightarrow P_i}$; it signifies that an instance of resource type $R_j$ has been allocated to process $P_i$.
- A directed edge $P_i \rightarrow R_j$ *is called a* ***request edge.***
- A directed edge $R_j \rightarrow P_i$ *is called an* ***assignment edge.***
- We represent each process $P_i$ as a circle and each resource type $R_j$ as a rectangle. We represent each instance of a resource type as a dot within the rectangle.
- A request edge points to only the rectangle $R_j$, whereas an assignment edge must also designate one of the dots in the rectangle.
- When a process $P_i$ requests an instance of resource type $R_j$, a request edge is inserted in the resource-allocation graph.
- When this request can be fulfilled, the request edge is transformed to an assignment edge.
- Assignment edge is deleted when process releases the resource.

**Example of a Resource Allocation Graph**



- The sets P, R and E:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - One instance of resource type $R_1$
  - Two instances of resource type $R_2$
  - One instance of resource type $R_3$
  - Three instances of resource type $R_4$
- Process states:
  - Process $P_1$ is holding an instance of resource type $R_2$ and is waiting for an instance of resource type $R_1$.
  - Process $P_2$ is holding an instance of $R_1$ and $R_2$ and is waiting for an instance of resource type $R_3$.
  - Process $P_3$ is holding an instance of $R_3$.
- If the graph contains no cycles, then no process in the system is deadlocked. If the graph contains a cycle, then a deadlock may exist.
- If the cycle involves only a set of resource types, each of which has a only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.

- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.
- In this case, cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

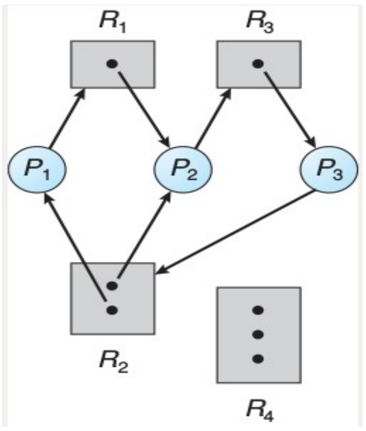
**Resource Allocation Graph With A Deadlock**

- According to this graph two minimal cycles exists in the system:

  $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

  $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Process $P_1$ is waiting for the resource $R_1$, which is held by process $P_2$.
- Process $P_2$ is waiting for the resource $R_3$ which is held by process $P_3$.
- Process $P_3$ is waiting for either $P_1$ or $P_2$ to release resource $R_2$.

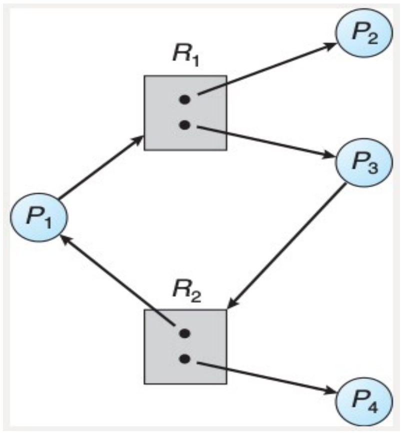
Processes $P_1$, $P_2$ and $P_3$ are deadlocked.



**Graph With A Cycle But No Deadlock**

- In this graph also there is cycle:

  $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- However there is no deadlock.
- Process $P_4$ may release its instance of resource type $R_2$. That resource can be allocated to $P_3$, breaking the cycle.
- 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.



## Methods for Handling Deadlocks

- **We can deal with deadlock problem in one of three ways:**
  - Use a protocol to **prevent or avoid** deadlocks, ensuring that the system will never enter a deadlocked state.
  - Allow the system to enter a deadlocked state, **detect it and recover**.
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems including UNIX and Windows.
- **Deadlock prevention** provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.

- **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.
- **Deadlock detection and recovery** provides an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.
- In the absence of above techniques, the undetected deadlock will result in deterioration of the system's performance. 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 system will stop functioning and need to be restarted manually.

## Deadlock Prevention

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

### Mutual Exclusion

- The mutual exclusion condition must hold for non-sharable resources. In contrast sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Read-only files are good examples of a sharable resource.
- We cannot prevent deadlocks by denying the mutual exclusion for non-sharable resources.

### 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 requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it is not holding any resource. A process must release all the resources before it can request any additional resources.

***Illustration to understand the difference between two protocols.***

- Consider a process that copies data from a DVD drive to a file on disk, sorts the the file, and then prints the result to a printer.
- According to the first protocol the process must initially request the DVD drive, disk file, and printer. Printer will be held for the entire execution, even though it is required only at the end.
- The second protocol allows the process to initially request only the DVD drive and disk file, Copies data from the DVD drive to the disk file, sorts the file and then releases both the DVD drive and the disk file.
- The process then requests the disk file and the printer. After copying the disk file to the printer, it releases these two and terminates.

***Disadvantages:***

- Both protocols result in **low resource utilization**, since resources may be allocated but unused for a long period.
- **Starvation**, A process that needs several popular resources may have to wait indefinitely.

### No Preemption

- To ensure that this condition does not hold, we can use the following protocol.

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released or preempted.
- 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.
- Alternatively if the requested resources are held by some other process that is waiting for additional resources. We preempt the desired resources from the waiting process and allocate them to the requesting process.

**Circular Wait**

- To ensure that this condition never holds we 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. 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 function F may be defined as follows:

  F(tape drive) = 1

  F(disk drive) = 5

  F(printer) = 12
- **Protocol to prevent deadlocks**: Each process can request resources only in an increasing order of enumeration. A process holding instances of resource type $R_i$ can request any instances of resource type $R_j$ if and only if $F(R_j) > F(R_i)$.
- If several instances of the same resource type are needed, a single request for all of them must be issued.
- For example a process that wants to use tape drive and printer at the same time must first request the tape drive and then request the printer.
- With this protocol circular wait condition cannot hold. We demonstrate this by assuming that a circular wait exists (proof by contradiction).
- Let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots, P_n\}$, where $P_i$ is waiting for a resource $R_{i+1}$, which is held by process $P_{i+1}$.
- Since process $P_i$ is holding resource $R_i$ while requesting resource $R_{i+1}$ we must have $F(R_i) < F(R_{i+1})$ for all i.
- This condition means that $\mathbf{F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)}$. By transitivity $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

## Deadlock Avoidance

- Deadlock prevention ensures that at least one of the necessary conditions for deadlock cannot hold and hence deadlocks cannot occur.
- Side effects of deadlock prevention are low device utilization and reduced system throughput.
- Deadlock avoidance requires the complete sequence of requests and releases for each process. With this knowledge system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- In making this decision the system consider the resources currently available, the resources currently allocated to each process and the future requests and releases of each process.

- 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.

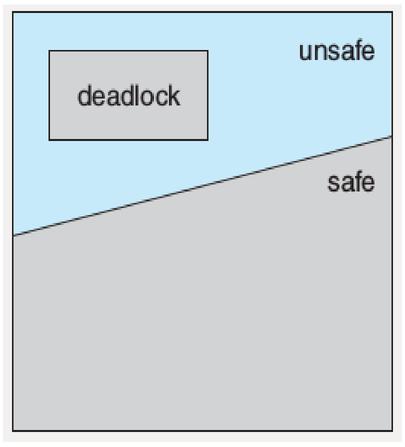
**Safe State**

- When a process requests a resource, the system must decide if immediate allocation of the resource leaves the system in a safe state.
- System is in a **safe state** if there exists a **safe sequence**. A sequence <$P_1$, $P_2$, …, $P_n$> of all the processes in the system is a safe sequence if, for each $P_i$, the resources that $P_i$ can still request can be satisfied by currently available resources plus the resources held by all $P_j$ , with $j < i$.
- That is:
  - If resources that $P_i$ 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 all needed resources, execute, return allocated resources, and terminate.
  - When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on.

If no such sequence exists, then the system state is said to be unsafe.

**Safe, Unsafe, Deadlock State**

- 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.



**Illustration**

- Consider a system with 12 magnetic tape drives and three processes: $P_0$ , $P_1$ and $P_2$ .
- $P_0$ requires 10 tape drives, $P_1$ may need as many as 4 tape drives and $P_2$ may need up to nine tape drives.
- At time $t_0$ , $P_0$ is holding five tape drives, $P_1$ is holding two tape drives and $P_2$ is holding two tape drives.

| | Maximum Needs | Current Needs |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | 7 |

- At time $t_0$ , system is in a safe state. The sequence <$P_1$ , $P_0$ , $P_2$ > satisfies the safety condition.

- Process $P_1$ can get all its tape drives and then return them (the system will then have five available tape drives).
- Then process $P_0$ can get all its tape drives and return them (the system will then have ten available tape drives).
- Finally process $P_2$ an get all its tape drives and return them (the system will then have twelve available tape drives).
- A system can go from a safe state to an unsafe state. Suppose, at time $t_1$, process $P_2$ requests and is allocated one more tape drive. The system is no longer in a safe state.
- At this point only process $P_1$ can be allocated all its tape drives. When it returns system will have only four available tape drives. With these four tape drives requests of $P_0$ and $P_2$ cannot be satisfied. So system is in unsafe state and may result in a deadlock.

**Avoidance Algorithms**

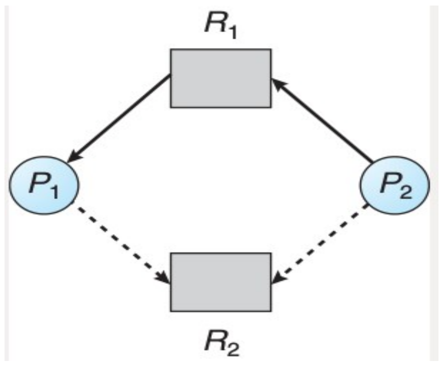
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

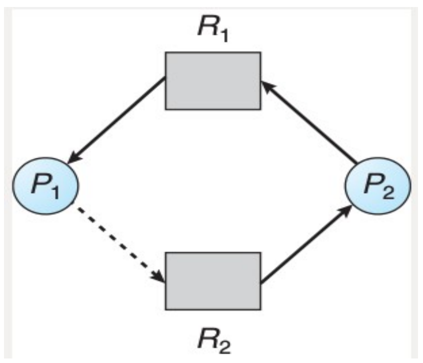
**Resource-Allocation Graph Algorithm**

- We introduce a new type of edge called claim edge, in addition to request and assignment edges.
- A claim edge $\boldsymbol{P_i \rightarrow R_j}$ indicates that process $P_i$ may request resource $R_j$ at some time in future, represented by a dashed line.
- When process $P_i$ requests resource $R_j$, the claim edge $\boldsymbol{P_i \rightarrow R_j}$ is converted to a request edge.
- Request edge $\boldsymbol{P_i \rightarrow R_j}$ is converted to an assignment edge $\boldsymbol{R_j \rightarrow P_i}$ when the resource is allocated to the process.
- When resource is released by the process, assignment edge is reconverted to a claim edge.
- Resources must be claimed a priori in the system. That is, before process $P_i$ starts executing, all its claim edges must appear in the resource allocation graph.
- The request can be granted only if converting the request edge $\boldsymbol{P_i \rightarrow R_j}$ to an assignment edge $\boldsymbol{R_j \rightarrow P_i}$ 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.

**Illustration**

- Consider the first resource allocation graph. Suppose that $P_2$ requests $R_2$. Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle in the graph, as shown in second graph.
- A cycle indicates that the system is in an unsafe state. If $P_1$ requests $R_2$ then a deadlock will occur.





## Banker's Algorithm

Resource-allocation graph algorithm is not applicable to a system with multiple instances of each resource type.

**Banker's algorithm works as follows:**

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- When resources are requested, the system must determine whether the allocation of these resources will leave the system in a safe state.
- Resources are allocated if the resulting state is safe, otherwise process must wait until some other process releases enough resources.

## Data Structures for the Banker's Algorithm

- Following data structures must be maintained to implement the banker's algorithm.
- Let ***n*** = number of processes, and ***m*** = number of resources types.
- **Available**: A vector of length m, indicates the number of available resources of each resource type. If *available[ j ] = k*, then k instances of resource type $R_j$ are available.
- **Max**: An *n x m* matrix, defines the maximum demand of each process. If *Max[ i ][ j ] = k*, then process $P_i$ may request at most k instances of resource type $R_j$.
- **Allocation**: An *n x m* matrix, defines the number of resources of each type currently allocated to each process. If *Allocation[ i ][ j ] = k* then $P_i$ is currently allocated k instances of resource type $R_j$.
- **Need**: An *n x m* matrix, indicates the remaining resource need of each process. 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 ]*

**Some notation:**

- Let X and Y be vectors of length n, we say that $X \leq Y$ 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 $Y \leq X$.
- We treat each row in the matrices allocation and need as vectors and refer to them as $\text{Allocation}_i$ and $\text{Need}_i$.

## Safety Algorithm

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) ***$\textbf{Need}_i \leq \textbf{Work}$***
3. If no such **i** exists, go to step 4

   ***$\textbf{Work} = \textbf{Work} + \textbf{Allocation}_i$***

***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**

- 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 ***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 resources are allocated to *P$_i$***

  **If unsafe => *P$_i$* must wait, and the old resource-allocation state is restored.**

**Example of Banker's Algorithm**

Consider a system with 5 processes $P_0$ through $P_4$ and 3 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 $T_0$ following snapshot of the system has been taken:

| | 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 | | | |

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 the safety criteria.
- Suppose now that process $P_1$ requests one additional instance of resource type A and two instances of resource type C, so *$Request_1$ = ( 1, 0, 2).*
- To decide whether this request can be immediately granted, we first check that *$Request_1$* ≤ *Available* – that is (1, 0, 2) ≤ (3, 3, 2), which is true.
- We pretend that this request has been fulfilled and we arrive at the following new state.

| | 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 | | | |

- We must determine whether this new system state is safe. Executing safety algorithm shows that sequence <***$P_1$*, *$P_3$*, *$P_4$*, *$P_0$*, *$P_2$***> satisfies safety requirement. Hence, we immediately grant the request of process $P_1$.
- When system is in this new state
  - Can request for (3,3,0) by ***$P_4$*** be granted?
  - Can request for (0,2,0) by ***$P_0$*** be granted?

## Deadlock Detection

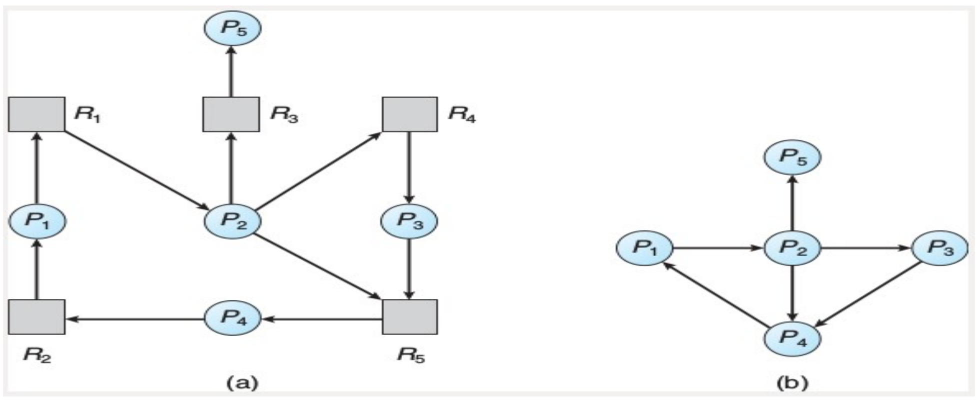
In this environment, the system provides:

- 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

- We 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 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.
- A deadlock exists if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system must maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.

### Resource-Allocation Graph and Wait-for Graph



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

**Several Instances of a Resource Type**

The algorithm uses several data structures that are similar to those used in the banker's algorithm.

- **Available**: A vector of length m, indicates the number of available resources of each resource 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, that indicates the current request of each process. If *Request[ i ][ j ] = k*, then $P_i$ 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: *Work = Available*, for i = 0, 1, 2, . . . , n-1. 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
3. ***Work = Work + Allocation***$_i$

   ***Finish[ i ] = true***

   go to step 2
4. If ***Finish[ i ] == false***, for some ***i***, 0 ≤ ***i*** < ***n***, then the system is in a deadlocked state. Moreover, if ***Finish[ i ] == false***, then ***P***$_i$ is deadlocked.

**Example of Detection Algorithm**

Consider a system with five processes $P_0$ through $P_4$ and three resource types A (7 instances), B (2 instances), and C (6 instances). Suppose that at time $T_0$ we have the following resource-allocation state:

| | 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 | | | |

- The system is not in a deadlocked state. The sequence <***$P_0$*, *$P_2$*, *$P_3$*, *$P_4$*, *$P_1$***> will result in ***Finish[i] = true*** for all ***i***.
- Suppose now that $P_2$ requests an additional instance of resource type C. The modified request matrix is as follows:

| | *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 |

- The system is now deadlocked.
- We can reclaim the resources held by process **$P_0$**, the number of resources is insufficient to fulfill the requests of other processes.
- Deadlock exists, consisting of processes ***$P_1$*, *$P_2$*, *$P_3$***, and ***$P_4$***

**Detection-Algorithm Usage**

- When and how often to invoke the detection algorithm depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- Deadlock occurs only when some process makes a request that cannot be granted immediately.
- We can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately.
- Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time.
- Less expensive approach is to invoke the algorithm at defined time intervals- for example once per hour or whenever CPU utilization drops below 40 percent.

## Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes:** Very expensive; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated:** More overhead; After each process is aborted a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
  - We must determine which deadlocked process(or processes) should be terminated. Many factors may affect which process is chosen, including:
    1. Priority of the process.
    2. How long the process has computed, and how much longer the process will compute before completing its designated task.

3. Resources the process has used.
4. How many more resources the process needs in order to complete.
5. How many processes will need to be terminated.
6. Is process interactive or batch?

## Recovery from Deadlock: Resource Preemption

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

**Three issues to be addressed for preemption.**

1. **Selecting a victim –** Which processes and which resources are to be preempted? Order of preemption must minimize the cost.
2. **Rollback –** After preempting a resource from a process, we must roll back the process to some safe state and later process must be restarted from that state.
    - Determining safe state is difficult, the simplest solution is a total rollback : abort the process and later restart it.
3. **Starvation** – We need to ensure that resources will not always be preempted from the same process.