

SYSTEM MODEL

A system consists of a finite number of resources to be distributed among a number of competing processes. Memory space, CPU cycles, files and I/O devices (such as printers and DVD drivers) are examples of resource types. Each resource type R_i has W_i instances.

A process must request a resource before using it and must release the resource after using it. A process cannot request three printers if the system has two printers.

Resources are sharable and non-sharable resources. For example, printer is non-sharable resource and reading a file is sharable resource.

Deadlock: A process requests a resource held by other process. For example, 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.

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, then the requesting process must wait until it can acquire the resources.

Use: The process can operate on the resource. For example, the resource is a printer, the process can print on the printer.

Release: The process releases the resource.

DEADLOCK CHARACTERIZATION

In a deadlock processes never finish executing, and system resources are tied up, preventing other from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at 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 $\{p_0, p_1, \dots, p_n\}$ of waiting processes must exist such that p_0 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 .

We emphasize that 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 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: $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource 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. A directed

edge from resource 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 . A directed edge $P_i \rightarrow R_j$ is called a **request edge**; A directed edge from $R_j \rightarrow P_i$ is called an **assignment edge**.

Pictorially we represent each process P_i as a circle and each resource type R_j as a rectangle. In the fig 1,

- 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 R1
 Two instance of resource type R2
 One instance of resource type R3
 Three instance 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.
- Process P3 is holding an instance of R3.

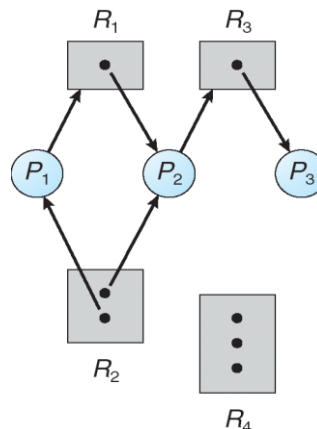


Figure: Resource allocation graph

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. 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.

To illustrate this concept, we return to the resource allocation graph (depicted in the figure below). Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. At this point, two minimal cycles exist in the system.

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

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 R3.

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

$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

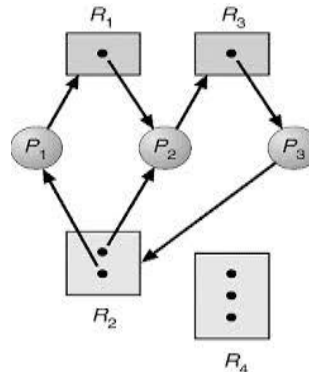


Figure: Resource allocation graph with deadlock.

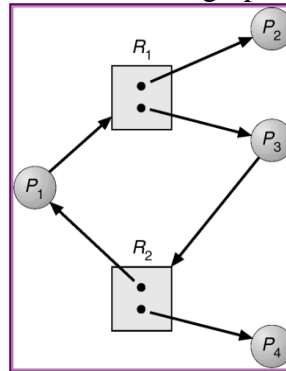


Figure: Resource-allocation graph with a cycle but no deadlock.

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 deadlock state. If there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with deadlock problem.

Methods for Handling Deadlocks

Generally speaking, 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 deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

DEADLOCK PREVENTION

To ensure that deadlock never occurred the system can use either a deadlock prevention (or) deadlock avoidance. Deadlock prevention provides a set of methods for ensuring that at least one of necessary conditions cannot hold.

1. **Mutual exclusion:** The mutual condition must hold for non-shareable resources. For example a printer cannot be simultaneously shared by several processes. Read only files are good examples of shareable resources. A process never needs to wait for a shareable resource. It

cannot prevent dead locks by denying the mutual exclusion condition, because some resources are intrinsically non-shareable resources.

2. **Hold and wait:** In Hold and Wait, If the Process is pointing towards Resource (Mutex) state then it is known as **Request**, i.e., Process is requesting some resource. Ex: Process1 is requesting for some DVD. If the Resource (Mutex) is pointing towards Process state then it is known as **Hold**, i.e., Resource is requesting for the Process. Ex: DVD is requesting Process P2.

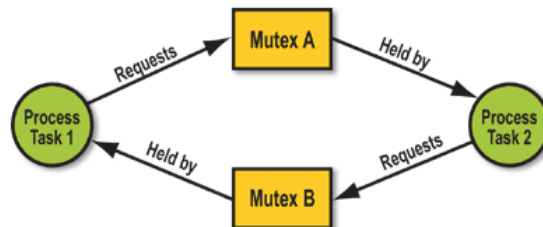


Figure: Deadlock with hold and wait

In Hold and Wait, If the Process is pointing towards Resource (Mutex) state then it is known as **Request**, i.e., Process is requesting some resource. Ex: Process1 is requesting for some DVD. If the Resource (Mutex) is pointing towards Process state then it is known as **Hold**, i.e., Resource is requesting for the Process. Ex: DVD is requesting Process P2.

The hold and wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all resources can be granted simultaneously.

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. To illustrate the difference between two methods, we consider a process copies data from DVD to a file on disk, sorts the file, and prints the file on printer.

Method 1: In first method, it requests all resources at the beginning of the process (DVD, disk & printer). It will hold the printer for its entire execution, even though it needs the printer only at the end.

Method 2: The second method allows process to request initially only the DVD drive and disk file. It copies the file from DVD to disk file, sorts the file and then release DVD and disk. The process must request the disk file and printer.

This approach is in efficient in several ways:

- A process may held up for a long time waiting for all its resource request to be filled. So starvation is possible.
 - Resources allocated to the process may remain unused for a long time, during which time they denied to other process.
 - It is very difficult to identify all the resources required by a process at the beginning.
3. **No preemption:** The third necessary condition for deadlocks is that there be no preemption of resources that have already allocated. If a process holding certain resources is denied a further request, that process must release its original resources and if necessary, request them again together with the additional resources. If a process requesting some resources, OS check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resources from waiting process and allocate them to the requesting process.

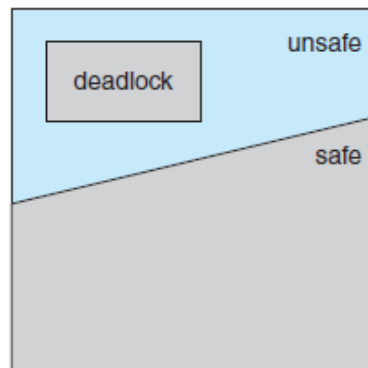
This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

4. **Circular-wait:** Circular wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resource of type R, then it may subsequently request only those resources of types following R in the ordering.

Assume resource R_i precedes R_j in the ordering $i < j$. Now suppose that two processes A and B are deadlocked because A has acquired R_i and requests R_j and B has acquired R_j and requests R_i . This condition is impossible it implies $i < j$ and $j < i$. The circular wait prevention may be inefficient, slowing down processes and denying resources access unnecessarily.

SAFE STATE

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ 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



Consider a system with 12 magnetic tape drives and three processes: P0, P1, and P2.

- Process P0 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 t_0 , process P0 is holding 5 tape drives, process P1 is holding 2 tape drives, and process P2 is holding 2 tape drives. (Thus, there are three free tape drives.)

	<u>Maximum Needs</u>	<u>Current Needs</u>
P_0	10	5
P_1	4	2
P_2	9	2

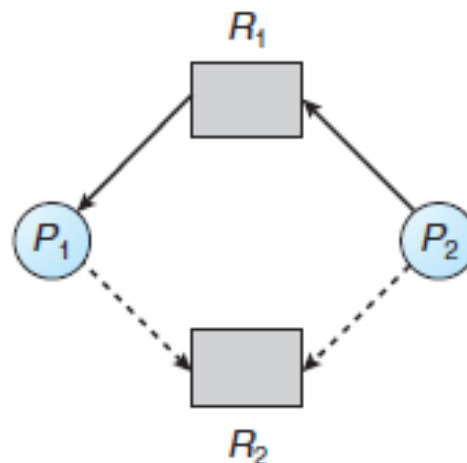
The sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition.

AVOIDANCE ALGORITHMS

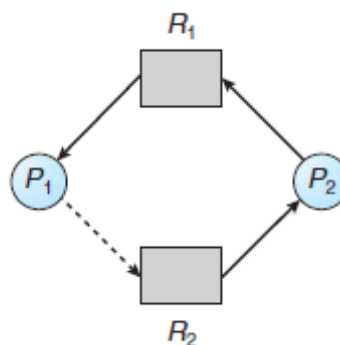
1. Single instance of a resource type
 - a. Use a resource-allocation graph
2. Multiple instances of a resource type
 - a. Use the banker's algorithm


Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system



- 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



 An unsafe state in a resource-allocation graph.

BANKER'S ALGORITHMS FOR DEADLOCK AVOIDANCE

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with a multiple instances of each resource type. The dead lock avoidance algorithms that we describe next are applicable to such a system but are less efficient than the resource allocation graph scheme. This algorithm is commonly known as bankers algorithms. The name was chosen because the algorithms could be used in banking system to ensure that bank never allocated its available cash in such a way 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 resource in the system. When a user requests a set of resources, the system must be determine whether the allocation of these resources will have 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.

Several data structures must be determine to implement the banker's algorithms. These data structures encode the state of the resources-allocation system. let n be the number of process in the system and m be the number of resources types. We need following data structures.

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

These data structure vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Lets x and y be vectors length n . we say that $x < y$ if and only if $x[i] < y[i]$ for all $i=1,2,\dots,n$. for example if $x=(1,7,3,2)$ and $y=(0.3,2,1)$, then $y < x$. $y < x$ if $y < x$ and $y \neq x$.

We can treat each row in the matrices allocation and need as vectors and refer to them as $allocation_i$ and $Need_i$. the vectors $allocation_i$ specifies the resources currently allocated to process p_i ; the vector $Need_i$ specifies the additional resources p_i may still request to complete its task.

Resource-Request Algorithm:

1. If $Request_i \leq Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$ Goto 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 is safe, the transaction is completed and process p_i is allocated its resources. However if the new state is unsafe p_i must wait for request I and old resource allocation is restored.

Safety Algorithm:

1. Let work and finish be vectors of length m and n, respectively. Initialize Work = Available. For $i=0,1,\dots,n-1$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i]=\text{false}$; otherwise, $\text{Finish}[i]=\text{true}$.
 2. Find an index I such that both
 - a. $\text{Finish}[i]=\text{false}$
 - b. $\text{Request}_i \leq \text{Work}$
 If no such i exists, go to step4.
 3. $\text{Work}=\text{Work}+ \text{Allocation}_i$
 $\text{Finish}[i]= \text{true}$.
 Go to step2.
 4. If $\text{Finish}[i]=\text{false}$, for some I, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $\text{Finish}[i]=\text{false}$, then process p_i is deadlocked.
- The algorithm may require an order of $m \times n^2$ operations to determine whether it is safe.

DEADLOCK DETECTION

In this situation, 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 single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph by a wait-for graph.

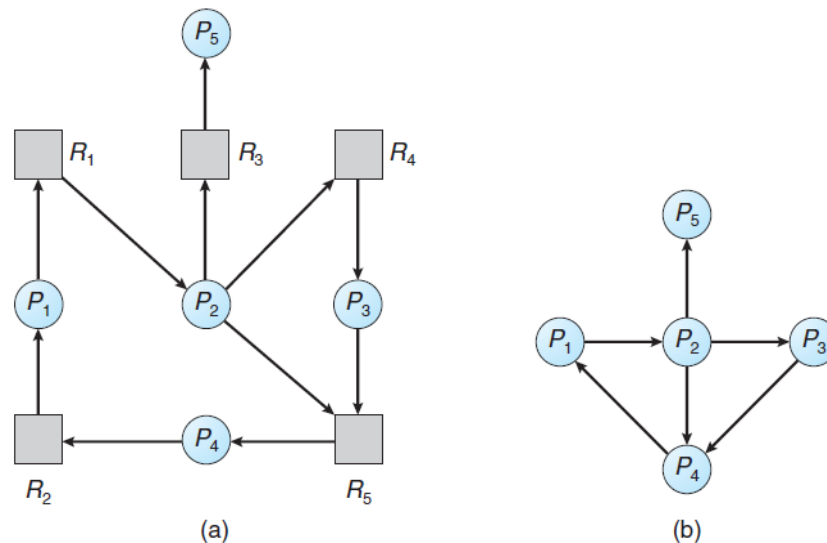


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

An edge $p_i \rightarrow p_j$ exists in wait for graph if and only if the correspond resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$.

Several Instance of a Resource Type:

- **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 $\text{Request}[i][j]$ equals k , then process P_i is requesting k more instances of resource type R_j .

Algorithm:

1. Let work and finish be vectors of length m and n , respectively. Initialize $\text{Work} = \text{Available}$. For $i=0,1,\dots,n-1$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i]=\text{false}$; otherwise, $\text{Finish}[i]=\text{true}$.
2. Find an index I such that both
 - a. $\text{Finish}[i]=\text{false}$
 - b. $\text{Request}_i \leq \text{Work}$
 If no such i exists, go to step 4.
3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$.
 Go to step 2.
4. If $\text{Finish}[i]=\text{false}$, for some $I, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $\text{Finish}[i]=\text{false}$, then process p_i is deadlocked.

The algorithm may require an order of $m \times n^2$ operations to determine whether it is safe.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives are available.

Process Termination: To eliminate deadlocks by aborting a process, we use one of two methods.

1. Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of the partial computations must be discarded and probably will have to be recomputed later.

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

Many factors may affect 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
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. If preemption is required to deal with deadlocks, then three issues need to be addressed:

Selecting a victim: As like in process termination, we must determine the order of preemption to minimize the cost.

Rollback: Clearly, 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 the state.

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 situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a finite number of times. The most common solution is to include the number of roll backs in the cost factor.

Bankers Algorithm Example:

- 1.) Consider a system with five processes P0 through P4 and three resource types A, B, and C. resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Find whether the system is in deadlock state or not.

	Allocation	Max	Available
	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	

The content of the matrix Need is defined to be Max-Allocation and is follows:

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

	Allocation	Max	Need	Available	
	A B C	A B C	A B C	A B C	
			3 3	2	
P0	0 1 0	7 5 3	7 4 3	5 3 2	-P1
P1	2 0 0	3 2 2	0 2 0	7 4 3	-P3
P2	3 0 2	9 0 2	6 0 0	7 4 5	-P4
P3	2 1 1	2 2 2	0 1 1	7 5 5	-P0
P4	0 0 2	4 3 3	4 3 1	10 5 7	-P2

We claim that the system is currently in a safe state. Indeed, the sequence $\langle P1, P3, P4, P2, P0 \rangle$ satisfies the safety criteria. Suppose now that process P1 request one additional instance of resource type A and two instances of resources type C, so request $t1 = (1, 0, 2)$. To decide whether this request can be immediately granted, we first check that $\text{Request}1 \leq \text{Available}$ that is, that $(1, 0, 2) \leq (3, 3, 2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

- 2.) Assume that there are 12 tape drives .Find whether the system is in safe state or not.

	Allocation	Max	Available
P0	5	10	3
P1	2	4	
P2	2	9	

The content of the matrix Need is defined to be Max-Allocation and is follows

	Need
P0	5
P1	2
P2	7

	Allocation	Max	Need	Available	
				3	
P0	5	10	5	5	-P1
P1	2	4	2	10	-P2
P2	2	9	7	12	-P3

We can conclude that the system is in safe state. Now consider that P2 requests and allocated one more tape drive then the allocation will be

	Allocation	Max	Available
P0	5	10	2
P1	2	4	
P2	3	9	

The content of the matrix Need is defined to be Max-Allocation and is follows

	Need
P0	5
P1	2
P2	6

	Allocation	Max	Need	Available	
				2	
P0	5	10	5	4	-P1
P1	2	4	2		
P2	3	9	6		

Here we can consider that the system is in deadlock state because $\text{Available} < \text{Need}$. So the system is in deadlock state.

Example:

Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	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	

Question1. What will be the content of the Need matrix?

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

So, the content of Need Matrix is:

Process	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

Question2. Is the system in safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,

m=3, n=5 Step 1 of Safety Algo
 Work = Available
 Work =

3	3	2
---	---	---

 0 1 2 3 4
 Finish =

false	false	false	false	false
-------	-------	-------	-------	-------

For i = 0 Step 2
 Need₀ = 7, 4, 3 ✗
 Finish [0] is false and Need₀ > Work
 So P₀ must wait But Need ≤ Work

For i = 1 Step 2
 Need₁ = 1, 2, 2 ✓
 Finish [1] is false and Need₁ < Work
 So P₁ must be kept in safe sequence

Step 3
 Work = Work + Allocation₁
 Work =

5	3	2
---	---	---

 0 1 2 3 4
 Finish =

false	true	false	false	false
-------	------	-------	-------	-------

For i = 2 Step 2
 Need₂ = 6, 0, 0 ✗
 Finish [2] is false and Need₂ > Work
 So P₂ must wait

For i = 3 Step 2
 Need₃ = 0, 1, 1 ✓
 Finish [3] is false and Need₃ < Work
 So P₃ must be kept in safe sequence

Step 3
 Work = Work + Allocation₃
 Work =

7	4	3
---	---	---

 0 1 2 3 4
 Finish =

false	true	false	true	false
-------	------	-------	------	-------

For i = 4 Step 2
 Need₄ = 4, 3, 1 ✓
 Finish [4] is false and Need₄ < Work
 So P₄ must be kept in safe sequence

Step 3
 Work = Work + Allocation₄
 Work =

7	4	5
---	---	---

 0 1 2 3 4
 Finish =

false	true	false	true	true
-------	------	-------	------	------

For i = 0 Step 2
 Need₀ = 7, 4, 3 ✓
 Finish [0] is false and Need₀ < Work
 So P₀ must be kept in safe sequence

Step 3
 Work = Work + Allocation₀
 Work =

7	5	5
---	---	---

 0 1 2 3 4
 Finish =

true	true	false	true	true
------	------	-------	------	------

For i = 2 Step 2
 Need₂ = 6, 0, 0 ✓
 Finish [2] is false and Need₂ < Work
 So P₂ must be kept in safe sequence

Step 3
 Work = Work + Allocation₂
 Work =

10	5	7
----	---	---

 0 1 2 3 4
 Finish =

true	true	true	true	true
------	------	------	------	------

Step 4
 Finish [i] = true for 0 ≤ i ≤ n
 Hence the system is in Safe state

The safe sequence is P₁, P₃, P₄, P₀, P₂

Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

A B C
Request₁ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

1, 0, 2 1, 2, 2 ✓
Request₁ < Need₁

Step 2

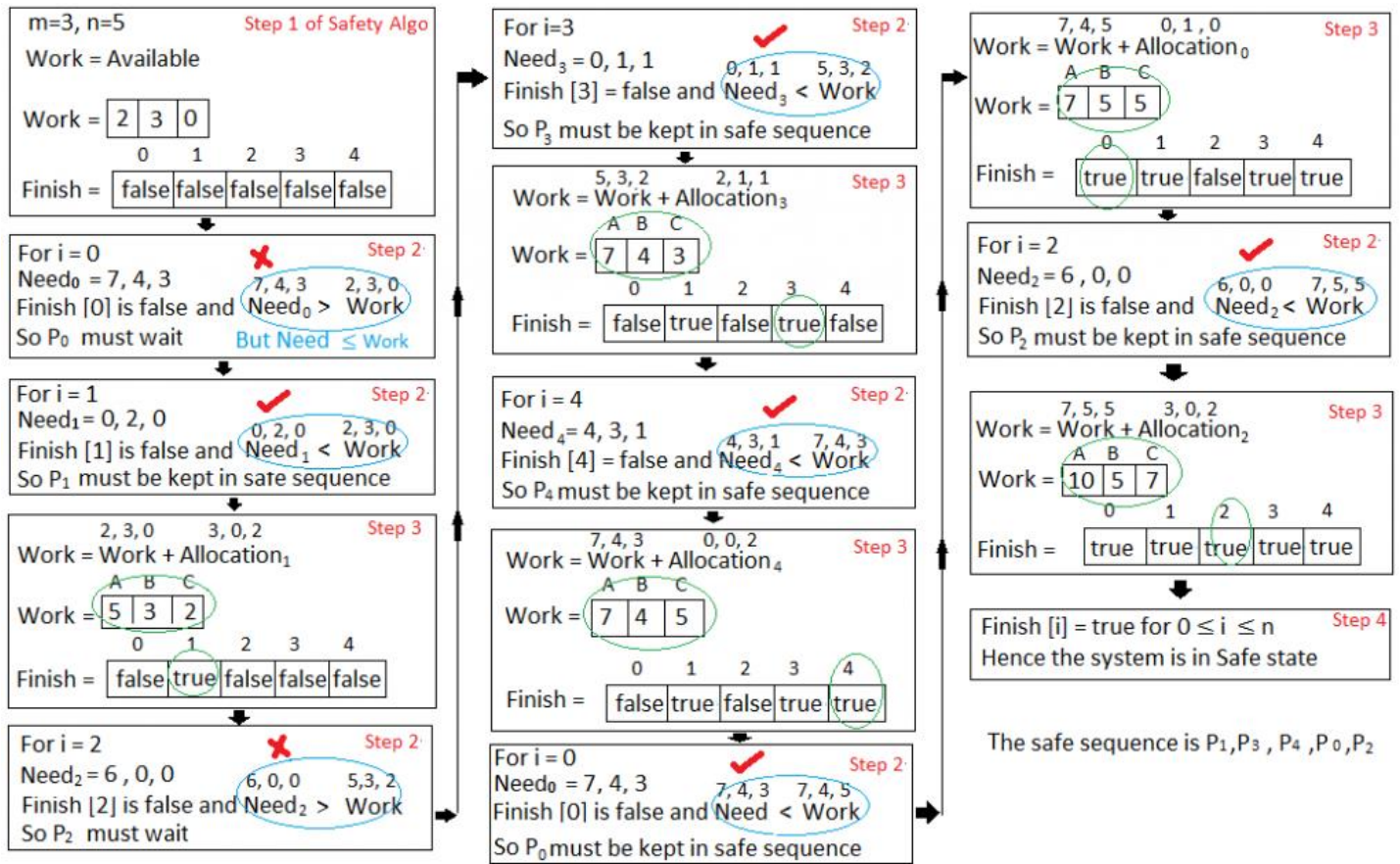
1, 0, 2 3, 3, 2 ✓
Request₁ < Available

Step 3

Available = Available – Request₁
Allocation₁ = Allocation₁ + Request₁
Need₁ = Need₁ - Request₁

Process	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. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process P₁.