

UNIT-3

Process Synchronization

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

The code executed by a process can be grouped into various sections. Some of these sections require access to shared resources while others do not.

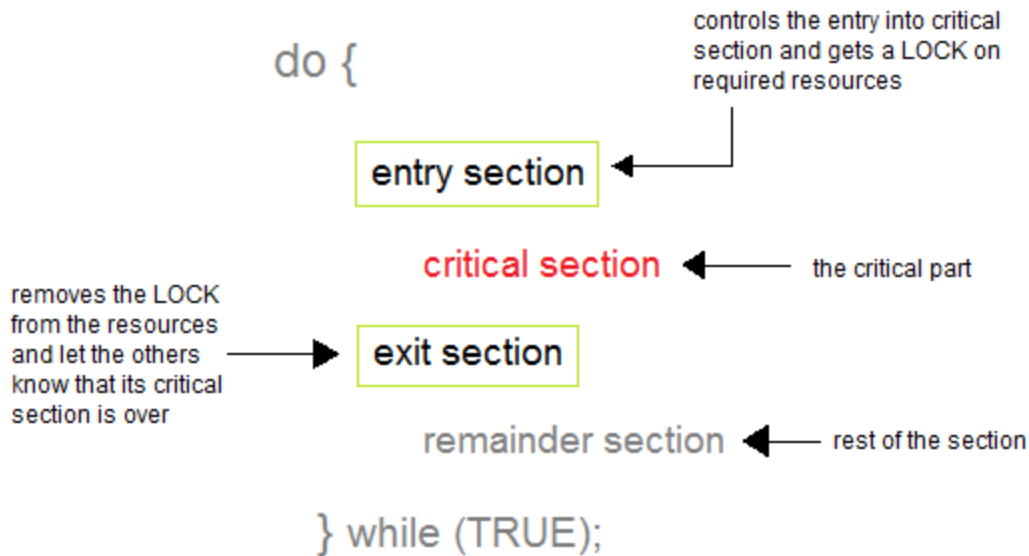
A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

Entry section- Each process must request for permission to enter its critical section. The code segment of a process that is executed when a process requests for an entry in its critical section constitutes the entry section. This section contains the code that checks whether a process is eligible to enter the critical section, otherwise it has to wait until it is allowed to enter.

Critical section – It is the code segment where a process can access shared resources. Only one process at a time can be allowed in the critical section.

Exit section- It is the segment of the process that is executes in preparation for leaving the critical section. Process must inform other processes by executing the exit section that it has indeed finished.

Remainder section- This section is the rest of the code of the process. When the process is in the remainder section, it implies that it is not waiting to enter its critical section.



Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

No two processes should be allowed to be inside their critical sections at the same time. So if a process P is currently executing in its critical section then no other cooperating processes must execute in their critical section until P leaves its critical section.

2. Progress

If no process is in its critical section, and if one or more process want to execute their critical section then any one of these processes must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows:

1. **Wait(P)**

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
Wait(s)
{
    While(s<=0);
    s--;
}
```

2. **Signal(V)**

The signal operation increments the value of its argument S.

```
Signal(s)
{
    S++;
}
```

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:

1. **Counting Semaphores**

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

2. **Binary Semaphores**

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

```
Wait(s)
If S>0
Then S=0
Else
```

Wait on S

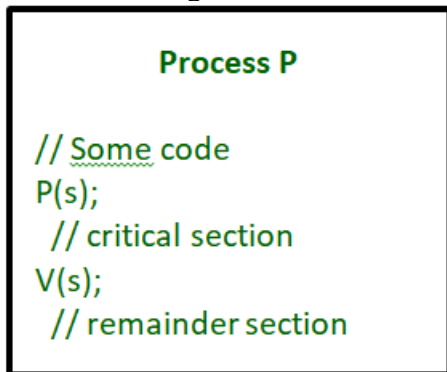
Signal(s)

If(one or more processes are waiting on S)

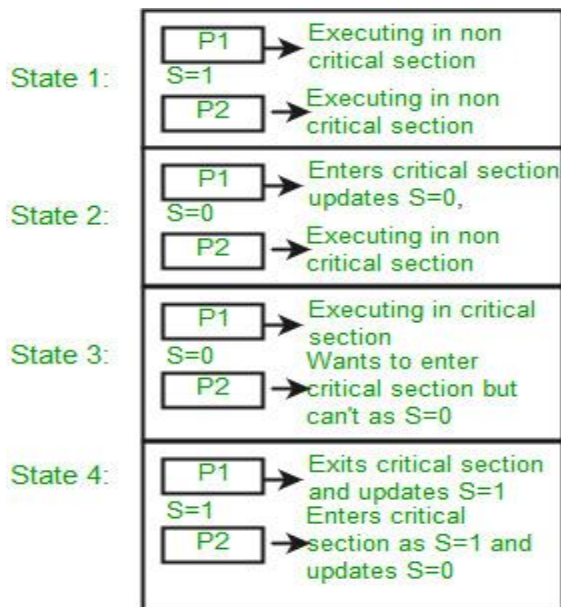
Then (let one of these processes to proceed)

Else S=1

1. See below image.critical section of Process P is in between P and V operation.



Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until $s > 0$, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.



Advantages of Semaphores

Some of the advantages of semaphores are as follows:

1. Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
2. There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
3. Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows:

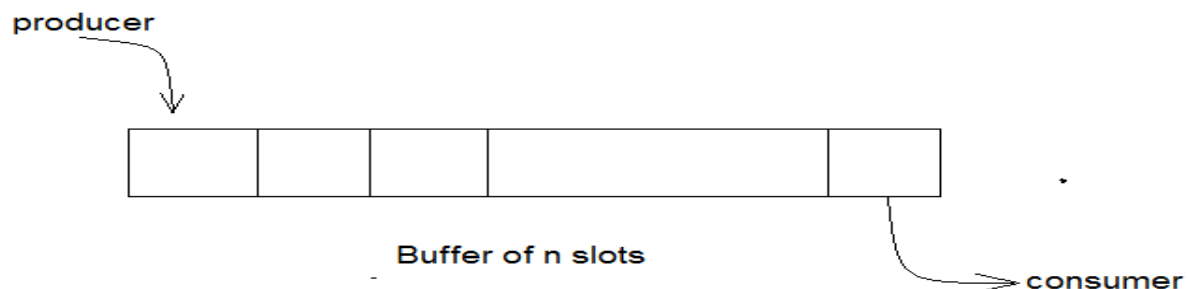
1. Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
2. Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
3. Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- **m**, a **binary semaphore** which is used to acquire and release the lock.
- **empty**, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a **counting semaphore** whose initial value is **0**.

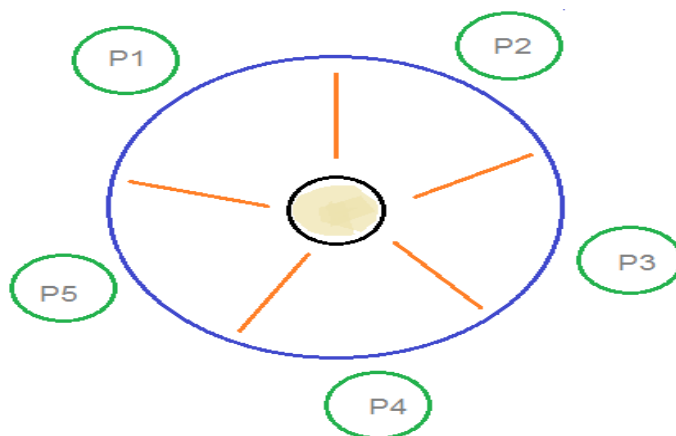
At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

Dining Philosophers Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, `stick[5]`, for each of the five chopsticks.

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

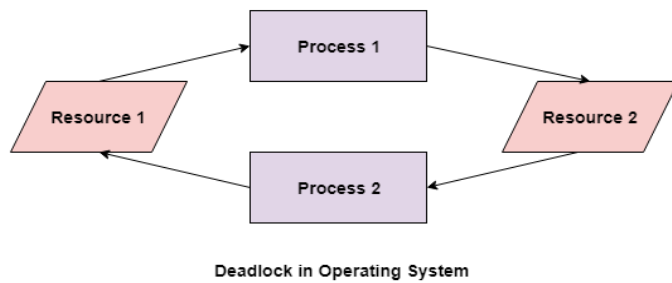
But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided

DEADLOCK

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

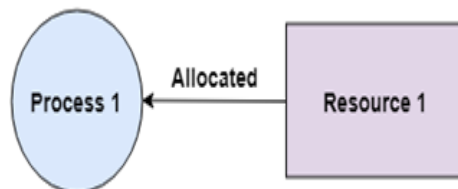
Coffman Conditions

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows:

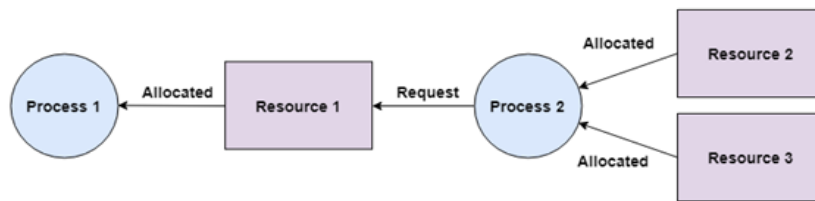
1. Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



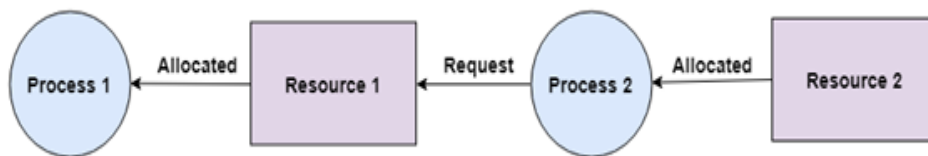
2. Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



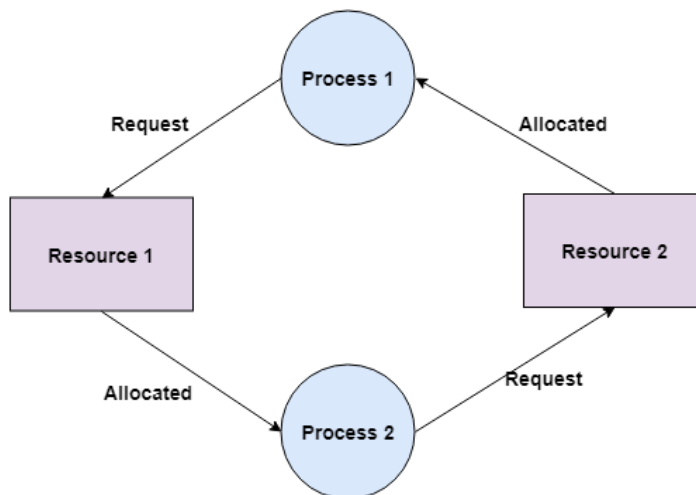
3. No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4. Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource 2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Deadlock in resource allocation

A process must request a resource before using it and must release the resources after using it. A process may request as many resources as it requires executing its designed task. Under the normal modes of operations, a process may utilize a resource in only the following sequence:

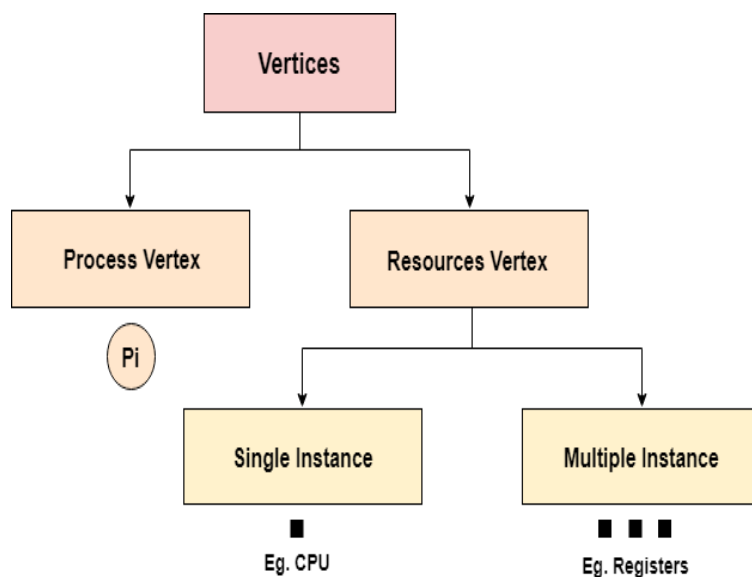
- (a) **Request:** First, the process must request the resource needed through a system call. If an instance of that resource is available, the operating system allocates the resource immediately. If the request cannot be granted immediately then the process must wait until it can acquire the resource.
- (b) **Allocation:** When the resource is allocated to a process, the process can operate on the resource i.e. use the services of the resource allocated. E.g. the resource is printer, when printer is allocated to a process A, the printer can print the print job of process A.
- (c) **Release:** When process has finished using the resource, it must release the resource through a system call. E.g. after printing, the process A should release the printer.

Resource Allocation Graph

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

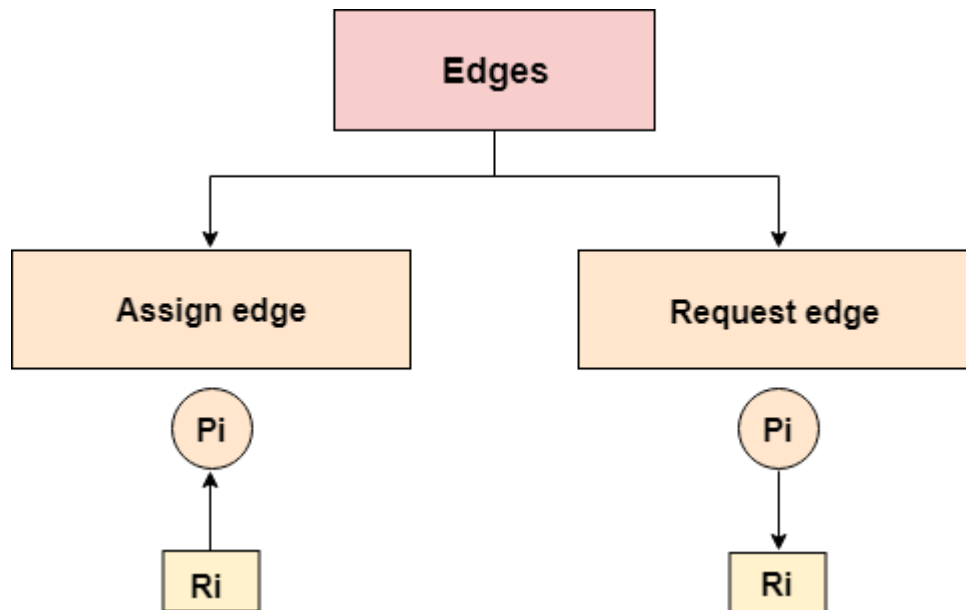
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.

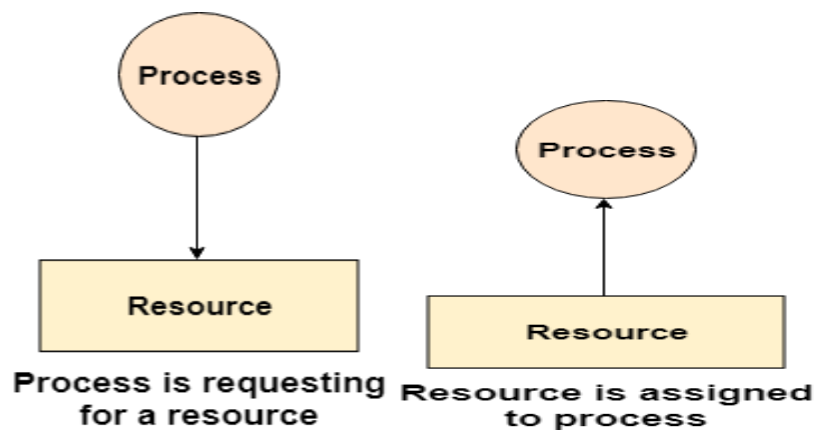


Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource.

A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them. A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process. A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

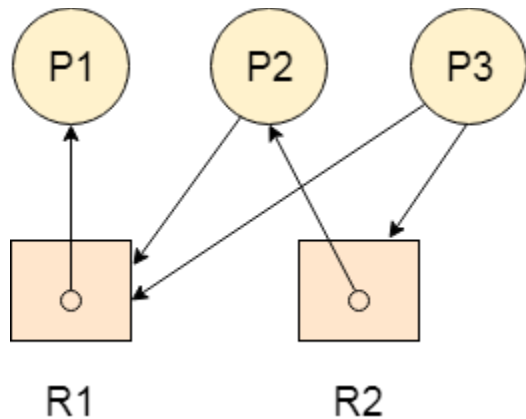


Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.

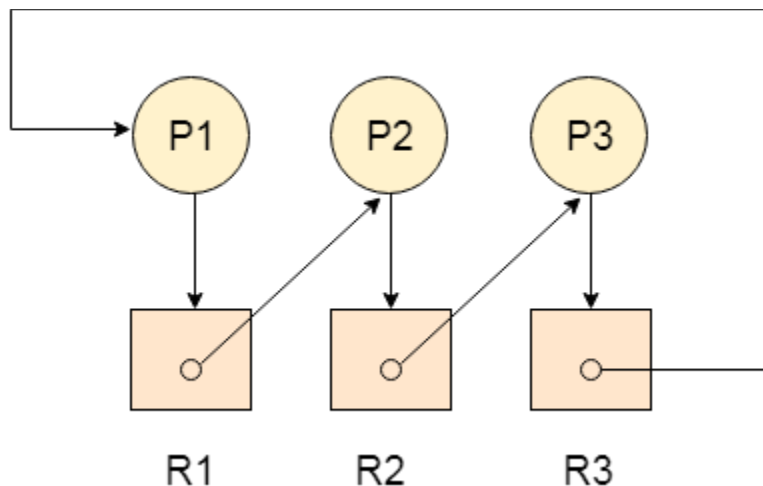


Deadlock Detection using RAG

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

The following example contains three processes P1, P2, P3 and three resources R1, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Allocation Matrix

Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, an entry is being made in front of P1 and below R3 since R3 is assigned to P1.

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0
P3	0	1	0

Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.

Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Deadlock Prevention

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

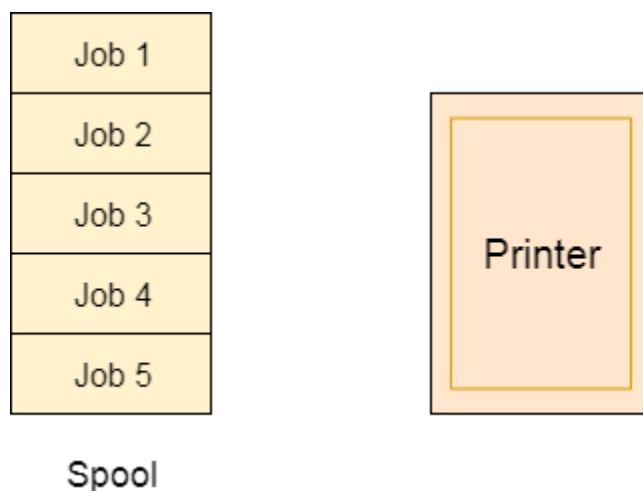
1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

3. No Preemption

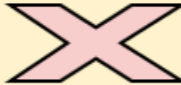



Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	
Hold and Wait	Request for all the resources initially	
No Preemption	Snatch all the resources	
Circular Wait	Assign priority to each resources and order resources numerically	

Among all the methods, violating Circular wait is the only approach that can be implemented practically.

Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.

A state of a system recorded at some random time is shown below.

Resources Assigned

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

Resources still needed

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

1. $E = (7 \ 6 \ 8 \ 4)$
2. $P = (6 \ 2 \ 8 \ 3)$
3. $A = (1 \ 4 \ 0 \ 1)$

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process.

Table 2 shows the instances of the resources, each process still needs. Vector E is the representation of total instances of each resource in the system.

Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use.

A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfill the request of all processes then the state of the system is called unsafe.

The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker's Algorithm:

Let '**n**' be the number of processes in the system and '**m**' be the number of resources types.

Available :

- It is a 1-d array of size '**m**' indicating the number of available resources of each type.
- $\text{Available}[j] = k$ means there are '**k**' instances of resource type **R_j**

Max :

- It is a 2-d array of size '**n*m**' that defines the maximum demand of each process in a system.
- $\text{Max}[i, j] = k$ means process **P_i** may request at most '**k**' instances of resource type **R_j**.

Allocation :

- It is a 2-d array of size '**n*m**' that defines the number of resources of each type currently allocated to each process.
- $\text{Allocation}[i, j] = k$ means process **P_i** is currently allocated '**k**' instances of resource type **R_j**

Need :

- It is a 2-d array of size '**n*m**' that indicates the remaining resource need of each process.
- $\text{Need}[i, j] = k$ means process **P_i** currently need '**k**' instances of resource type **R_j**

for its execution.

- $\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$
- Allocation_i specifies the resources currently allocated to process **P_i** and Need_i specifies the additional resources that process **P_i** may still request to complete its task.
- Banker's algorithm consists of Safety algorithm and Resource request algorithm
- **Safety Algorithm**
- The algorithm for finding out whether or not a system is in a safe state can be described as follows:

- 1) Let *Work* and *Finish* be vectors of length 'm' and 'n' respectively.
Initialize: *Work* = *Available*
Finish[i] = false; for i=1, 2, 3, 4....n
- 2) Find an i such that both
 - a) *Finish*[i] = false
 - b) $Need_i \leq Work$
 if no such i exists goto step (4)
- 3) *Work* = *Work* + *Allocation*[i]
Finish[i] = true
goto 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 array for process *P_i*. *Request_i* [j] = k means 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 $Request_i \leq Need_i$
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:
 $Available = Available - Request_i$
 $Allocation_i = Allocation_i + Request_i$
 $Need_i = Need_i - Request_i$
- Example:**
- Considering a system with five processes *P₀* through *P₄* and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time *t₀* following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
<i>P₀</i>	0 1 0	7 5 3	3 3 2
<i>P₁</i>	2 0 0	3 2 2	
<i>P₂</i>	3 0 2	9 0 2	
<i>P₃</i>	2 1 1	2 2 2	
<i>P₄</i>	0 0 2	4 3 3	

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

Need [i, j] = Max [i, j] – 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 a 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

7,4,3 3,3,2

Finish [0] is false and Need₀ > Work

So P₀ must wait But Need ≤ Work

For i=1 Step 2

Need₁ = 1, 2, 2

1,2,2 3,3,2

Finish [1] is false and Need₁ < Work

So P₁ must be kept in safe sequence

3, 3, 2 2, 0, 0 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

6,0,0 5,3,2

Finish [2] is false and Need₂ > Work

So P₂ must wait

For i=3 Step 2

Need₃ = 0, 1, 1

0,1,1 5,3,2

Finish [3] = false and Need₃ < Work

So P₃ must be kept in safe sequence

5, 3, 2 2, 1, 1 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

4,3,1 7,4,3

Finish [4] = false and Need₄ < Work

So P₄ must be kept in safe sequence

7, 4, 3 0, 0, 2 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

7,4,3 7,4,5

Finish [0] is false and Need < Work

So P₀ must be kept in safe sequence

7, 4, 5 0, 1, 0 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

6,0,0 7,5,5

Finish [2] is false and Need₂ < Work

So P₂ must be kept in safe sequence

7, 5, 5 3, 0, 2 Step 3

Work = Work + Allocation₂

Work =

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

0 1 2 3 4

Finish =

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

Finish [i] = true for 0 ≤ i ≤ n Step 4

Hence the system is in Safe state

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

Question3. What will happen if process P₁ 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

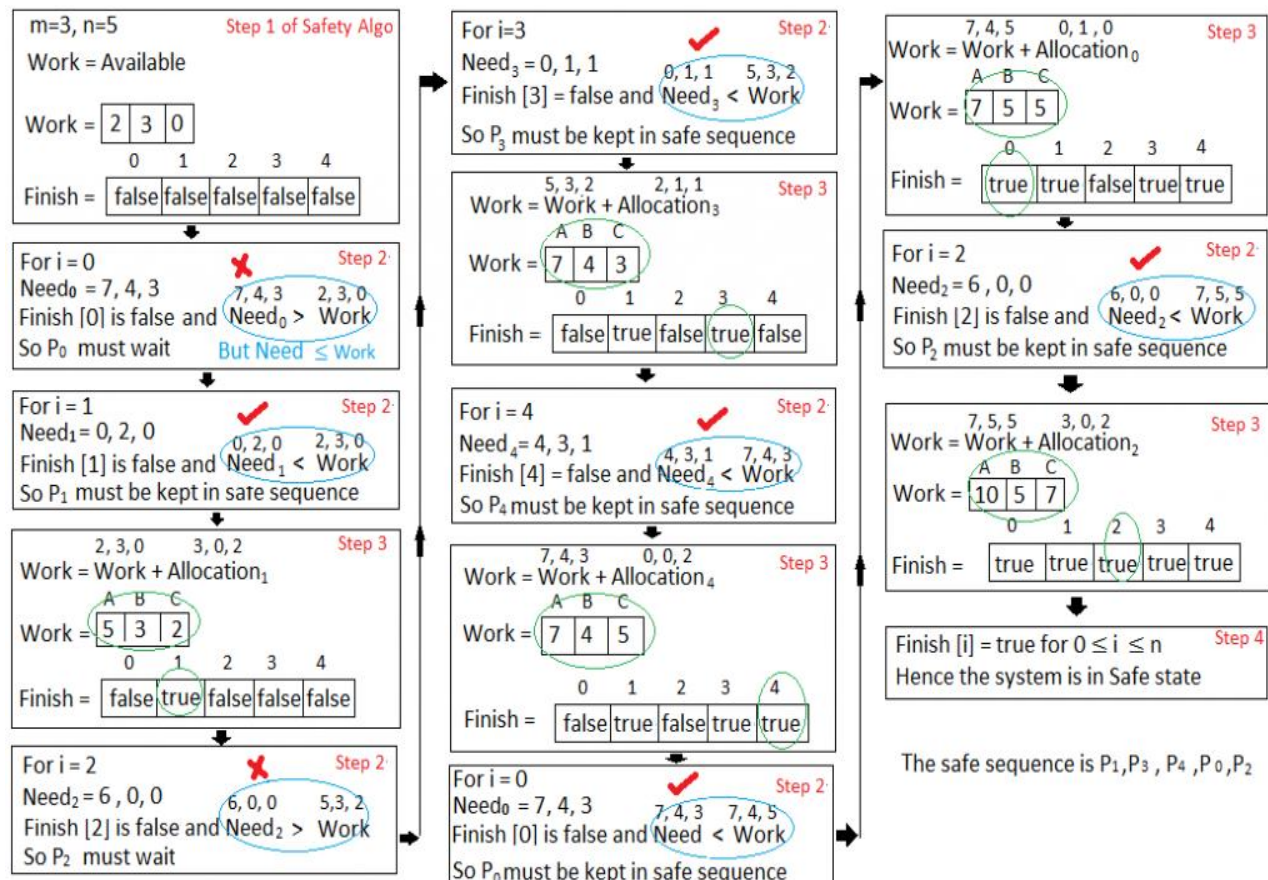
Step 2
1, 0, 2 3, 3, 2 ✓
Request₁ < Available

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	

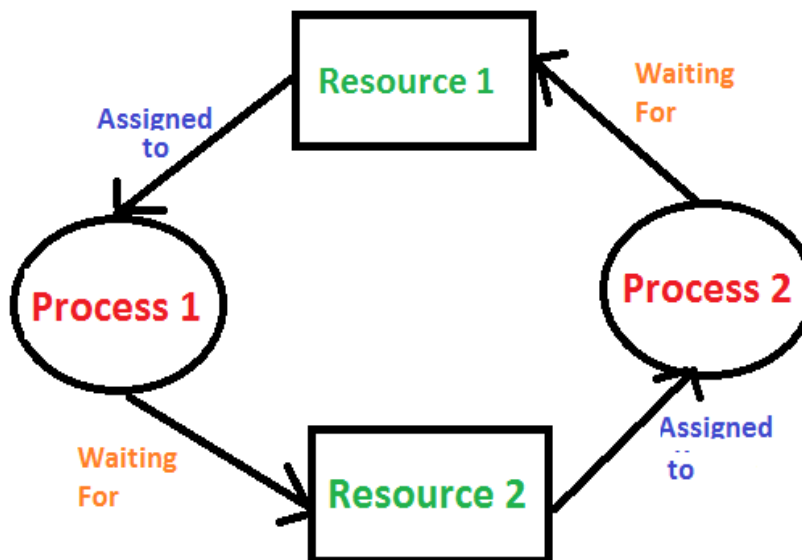
We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



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

Deadlock recovery

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



Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes. » 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 these partial computations must be discarded and probably will have to be recomputed later.

- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The

question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. 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 (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

2. Rollback. If we preempt a resource from a process, what should be done with that process? 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 that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? 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 (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.