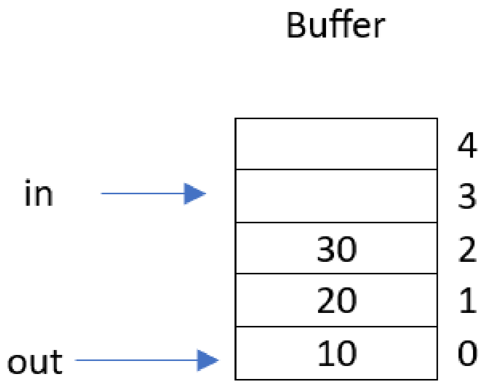
Module 3.1

PROCESS SYNCHRONIZATION

Producer – Consumer Problem

- There is a Buffer with some number of slots.
- One item can be stored in each slot of the buffer.
- *In* and *out* are pointers to the buffer.
- *Count* is a variable which indicates the number of items in the buffer.
- *Producer* is the process which inserts items into the buffer.
- *In* points to the next free slot of buffer into which the producer process inserts the next item.
- *In* pointer is moved to the next position after inserting an item into the buffer.
- Count is incremented by 1 after inserting an item into the buffer.
- *Consumer* is the process which removes items from the buffer.
- *Out* points to the slot from which the consumer process removes the item.
- *Out* pointer is moved to the next position after removing an item from the buffer.
- Count is decremented by 1 after removing an item from the buffer.



buffer_size = 5

count = 3

Producer Process

The code of producer process is:

```
while (true)
{
        while (count == buffer_size)
                ;
        Buffer[in] = item;
        in = (in + 1) % buffer_size;
        count = count + 1;
}
```

Consumer Process

The code of consumer process is:

```
while(true)
{
        while (count == 0)
                ;
        item = Buffer[out];
        out = (out+1) % buffer_size;
        count = count -1;
}
```

Producer Process

The code of producer process is:

```
while (true)
{
        P1:     while (count == buffer_size)
        P2:             ;
        P3:     Buffer[in] = item;
        P4:     in = (in + 1) % buffer_size;
        P5:     register$_1$ = count;
        P6:     register$_1$ = register$_1$ + 1;
        P7:     count = register$_1$;
}
```

Consumer Process

The code of consumer process is:

```
while(true)
{
        C1:     while (count == 0)
        C2:             ;
        C3:     item = Buffer[out];
        C4:     out = (out+1) % buffer_size;
        C5:     register$_2$ = count;
        C6:     register$_2$ = register$_2$ - 1;
        C7:     count = register$_2$;
}
```

Serial Execution of processes

Processes are executed one after another.

CPU is switched to next process only after completion of the currently running process.

N number of processes can be serially executed in N! number of ways.

For example, two processes p1 and p2 can be executed serially in two ways:

1) p1, p2 2) p2, p1

Three processes p1, p2 and p3 can be executed serially in six ways:

1) p1, p2, p3 2) p1, p3, p2 3) p2, p1, p3 4) p2, p3, p1 5) p3, p1, p2 6) p3, p2, p1

Concurrent Execution (or) Parallel Execution of processes

CPU can be switched to next process during execution of the current process.

The number of concurrent executions possible with n number of processes depends on the number of statements in each process.

Independent processes

Processes running in the computer system are said to be independent processes if they are not exchanging or sharing any data.

Cooperating processes

Processes running in the computer system are said to be cooperating processes if they are exchanging or sharing any data.

Correct output is generated when the independent processes are executed either serially or concurrently.

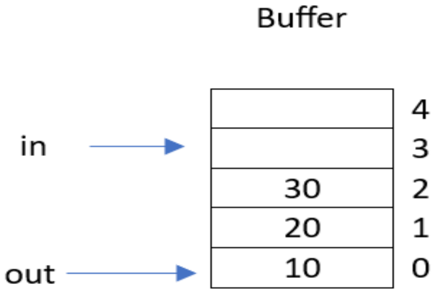
Correct output is generated when the cooperating processes are executed serially.

Concurrent execution of cooperating processes may generate wrong output.

One example concurrent execution of producer & consumer processes that leads to wrong result is:
P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, P7, C7.

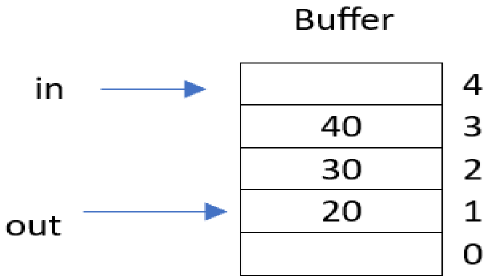
```
while (true)
{
        P1:     while (count == buffer_size)
        P2:             ;
        P3:     Buffer[in] = item;
        P4:     in = (in + 1) % buffer_size;
        P5:     $register_1$ = count;
        P6:     $register_1$ = $register_1$ + 1;
        P7:     count = $register_1$;
}
```

```
while(true)
{
        C1:     while (count == 0)
        C2:             ;
        C3:     item = Buffer[out];
        C4:     out = (out+1) % buffer_size;
        C5:     $register_2$ = count;
        C6:     $register_2$ = $register_2$ - 1;
        C7:     count = $register_2$;
}
```



buffer_size = 5

count = 3



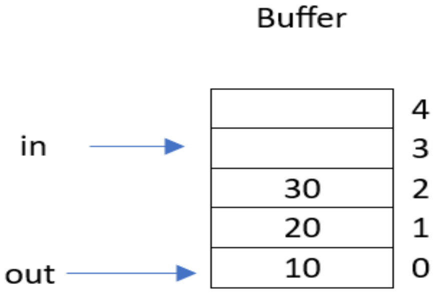
buffer_size = 5

count = 2

Another example concurrent execution of producer & consumer processes that leads to wrong result is:
P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, C7, P7.

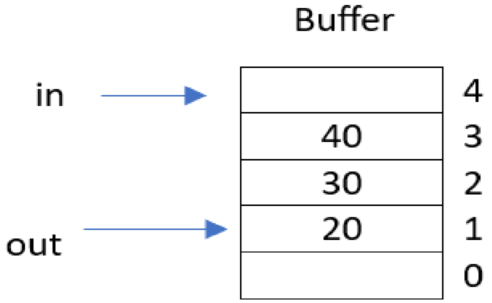
```
while (true)
{
        P1:     while (count == buffer_size)
        P2:             ;
        P3:     Buffer[in] = item;
        P4:     in = (in + 1) % buffer_size;
        P5:     $register_1$ = count;
        P6:     $register_1$ = $register_1$ + 1;
        P7:     count = $register_1$;
}
```

```
while(true)
{
        C1:     while (count == 0)
        C2:             ;
        C3:     item = Buffer[out];
        C4:     out = (out+1) % buffer_size;
        C5:     $register_2$ = count;
        C6:     $register_2$ = $register_2$ - 1;
        C7:     count = $register_2$;
}
```



buffer_size = 5

count = 3



buffer_size = 5

count = 4

Race condition

Race condition is getting different outputs for different concurrent executions of cooperating processes.

Ex: What are the different values that A and B get after the execution of the processes P1, P2 concurrently?

Initial values of A and B are 6 and 4 respectively.

Process P1:	Process P2:
I1: A=A-B;	I11: A=A+1;
I2: B=B+A;	I12: B=B-1;

Concurrent execution sequence1: I1, I2, I11, I12

A=6-4=2

B=4+2=6

A=2+1=3

B=6-1=5

A=3, B=5

Concurrent execution sequence2: I1, I11, I12, I2

A=6-4=2

A=2+1=3

B=4-1=3

B=3+3=6

A=3, B=6

Process P1:	**Process P2:**

I1: A=A-B;
I2: B=B+A;
	I11: A=A+1;
	I12: B=B-1;

Concurrent execution sequence3:   I1, I11, I2, I12

A=6-4=2

A=2+1=3

B=4+3=7

B=7-1=6

A=3, B=6

Concurrent execution sequence4:   I11, I12, I1, I2

A=6+1=7

B=4-1=3

A=7-3=4

B=3+4=7

A=4, B=7

Process P1:	**Process P2:**
I1: A=A-B;	**I11: A=A+1;**
I2: B=B+A;	**I12: B=B-1;**

Concurrent execution sequence5: I11, I1, I2, I12

A=6+1=7

A=7-4=3

B=4+3=7

B=7-1=6

A=3, B=6

Process P1:	**Process P2:**

I1: A=A-B;
I2: B=B+A;
	I11: A=A+1;
	I12: B=B-1;

Concurrent execution sequence6: I11, I1, I12, I2

A=6+1=7

A=7-4=3

B=4-1=3

B=3+3=6

A=3, B=6

The different possible values that A and B can take are:

A=3, B=5

A=3, B=6

A=4, B=7

Critical Section of a process

Critical section of a process is the set of statements in the process which access and modify the shared data.

The critical section of producer process is:

register_1 = count;
register_1 = register_1 + 1;
count = register_1;

The critical section of consumer process is:

register_2 = count;
register_2 = register_2 - 1;
count = register_2;

Critical Section Problem

When two or more cooperating processes are executing concurrently, then at a time only one process should execute the statements in its critical section.

(or)

When one process is already executing the statements in its critical section, no other process is allowed to execute the statements in its critical section.

(or)

No two processes are allowed to execute the statements in their critical section at the same time.

When the execution of cooperating processes is restricted in this way then the correct output is generated even if the cooperating processes are executed concurrently.

Each process must request permission to start the execution of statements in its critical section.

The section of code implementing this request is the **entry section**.

The critical section may be followed by an **exit section** which allows the other process to start the execution of statements in its critical section.

The general structure of a process is shown in the following figure.

```
while (true)
{
            .
            .
        Entry Section

            Critical Section

        Exit Section
            .
            .
}
```

Solutions for Critical Section Problem (or) Solutions for Implementing Mutual Exclusion

1. Peterson’s Solution or Software Solution
2. Hardware Solution
3. Semaphore Solution
4. Monitor Solution

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

1) Mutual exclusion
2) Progress
3) Bounded waiting

Mutual Exclusion

No two processes should execute the statements in their critical section at the same time.

Progress

At any time, at least one process should be in the running state or active. No deadlock should occur.

Bounded waiting

All processes should be given equal chance to execute the statements in their critical sections.

Peterson’s Solution

It is applicable to two processes only.

Two variables are used

- int turn;
- boolean flag[2];

turn variable indicates whose turn it is to execute the statements in its critical section.

turn=1 - indicates that the 1st process (producer process) can execute the statements in its critical section.

turn=2 - indicates that the 2nd process (consumer process) can execute the statements in its critical section.

flag indicates whether a process is ready to execute the statements in its critical section.

flag[1]=true - indicates that the 1st process (producer process) is ready to execute the statements in its critical section.

flag[2]=true - indicates that the 2nd process (consumer process) is ready to execute the statements in its critical section.

Initially, flag[1] and flag[2] are set to false.

The following procedure is used to solve the critical section problem:

When the 1st process (producer process) wants to execute the statements in its critical section then the 1st process (producer process) sets flag[1] to true.

If the 2nd process (consumer process) is also ready to execute the statements in its critical section, then the 1st process (producer process) allows the 2nd process (consumer process) to execute the statements in its critical section by setting the turn value to 2.

The 1st process (producer process) waits until the 2nd process (consumer process) completes the execution of statements in its critical section.

Then the 1st process (producer process) starts executing the statements in its critical section.

The 1st process (producer process) sets flag[1] to false after completing the execution of statements in its critical section.

Same procedure is followed by the 2nd process (consumer process) when it wants to execute the statements in its critical section.

Peterson’s solution to the critical section problem of producer-consumer is as follows:

Producer Process

```
while (true)
{
          while (count == buffer_size)
                    ;
          Buffer[in] = item;
          in = (in + 1) % buffer_size;

Entry     flag[1]=true;
Section   turn=2;
          while(flag[2]==true && turn==2)
                    ;

          $register_1$ = count;
          $register_1$ = $register_1$ + 1;   Critical Section
          count = $register_1$;

Exit      flag[1]=false;
Section
}
```

Consumer Process

```
while(true)
{
          while (count == 0)
                    ;
          item = Buffer[out];
          out = (out+1) % buffer_size;

Entry     flag[2]=true;
Section   turn=1;
          while(flag[1]==true && turn==1)
                    ;

          $register_2$ = count;
          $register_2$ = $register_2$ - 1;   Critical Section
          count = $register_2$;

Exit      flag[2]=false;
Section
}
```

Hardware Solution

A variable named ‘lock’ is used.

```
boolean lock;
```

When a process wants to execute the statements in its critical section, the process checks the value of lock variable.

If the value of lock variable is true, then the process is not allowed to execute the statements in its critical section.

If the value of lock variable is false, then the process is allowed to execute the statements in its critical section.

Before executing the statements in the critical section, the process must set the value of lock variable to true.

After executing the statements in the critical section, the process must set the value of lock variable to false.

The structure of process is:

```
while (true)
{
        .
        .

        Acquire lock;

        Critical Section

        Release lock;
        .
        .
}
```

Test_and_Set()

‘Test_and_Set’ is a hardware instruction which performs operations on the ‘lock’ variable to solve the critical section problem.

It takes ‘lock’ variable as input and returns a boolean value.

The definition of Test_and_Set() is

```
boolean Test_and_Set(boolean lock)
{
        boolean temp=lock;
        lock=true;
        return temp;
}
```

Test_and_Set() is an atomic instruction.

During the execution of Test_and_Set(), the CPU is not switched to any other process.

Initial value of ‘lock’ is false.

The solution to critical section problem of producer-consumer is

Producer Process

```
while (true)
{
          while (count == buffer_size)
                    ;
          Buffer[in] = item;
          in = (in + 1) % buffer_size;

Entry     while(Test_and_Set(lock))
Section             ;

          $register_1$ = count;
          $register_1$ = $register_1$ + 1;   Critical Section
          count = $register_1$;

Exit      lock=false;
Section
}
```

Consumer Process

```
while(true)
{
          while (count == 0)
                    ;
          item = Buffer[out];
          out = (out+1) % buffer_size;

Entry     while(Test_and_Set(lock))
Section             ;

          $register_2$ = count;
          $register_2$  = $register_2$ - 1;   Critical Section
          count = $register_2$;

Exit      lock=false;
Section
}
```

Swap()

‘Swap’ is another hardware instruction which is used to solve the critical section problem.

The definition of Swap() instruction is

```
void Swap(boolean lock, boolean key)
{
        boolean temp=lock;
        lock=key;
        key=temp;
}
```

‘lock’ is a global variable.

‘key’ is a local variable.

Initial value of lock variable is false.

The solution to critical section problem of producer-consumer is

Producer Process

```
while (true)
{
            while (count == buffer_size)
                        ;
            Buffer[in] = item;
            in = (in + 1) % buffer_size;

Entry       $key_1$ = true;
Section     while($key_1$==true)
                        swap(lock, $key_1$);

            $register_1$ = count;
            $register_1$ = $register_1$  + 1;   Critical Section
            count = $register_1$;

Exit        lock=false;
Section
}
```

Consumer Process

```
while(true)
{
            while (count == 0)
                        ;
            item = Buffer[out];
            out = (out+1) % buffer_size;

Entry       $key_2$ = true;
            while($key_2$==true)
Section                 swap(lock, $key_2$);

            $register_2$ = count;
            $register_2$  = $register_2$ - 1;   Critical Section
            count = $register_2$;

Exit        lock=false;
Section
}
```

Semaphore Solution

Semaphore is an integer variable.

There are two types of semaphore:

1) Binary semaphore
2) Counting semaphore

Binary semaphore is used to solve the critical section problem.

Binary semaphore is also called as ‘mutex’ variable as binary semaphore is used to implement mutual exclusion.

Initial value of binary semaphore is 1.

Counting semaphore is used to provide synchronous access to a resource when the resource is requested by a number of processes at a time.

Initial value of counting semaphore is equal to the number of instances of the resource.

A queue is associated with each semaphore variable.

Two atomic operations are defined on a semaphore variable:

1) wait()
2) signal()

The definition of wait() operation is:

```
wait(s)
{
        s=s-1;
        if(s<0)
        {
                Suspend the execution of the process which has invoked the wait() operation;
                Insert that process in the queue associated with the semaphore variable s;
        }
}
```

The definition of signal() operation is:

```
signal(s)
{
        s=s+1;
        if(s<=0)
        {
                remove a process from the queue associated with the semaphore variable s;
                restart the execution of that removed process;
        }
}
```

To solve the critical section problem, a process should invoke wait() operation on the semaphore variable before starting the execution of statements in its critical section and should invoke signal() operation after executing the statements in its critical section.

The structure of process is

```
                .
                .
                wait(S);
                 Critical Section
                signal(S);
                .
                .
```

The solution to critical section problem of producer-consumer is

Producer Process

```
while (true)
{
            while (count == buffer_size)
                        ;
            Buffer[in] = item;
            in = (in + 1) % buffer_size;

Entry       wait(s);
Section

            $register_1$ = count;
            $register_1$ = $register_1$ + 1;     Critical Section
            count = $register_1$;

Exit        signal(s);
Section
}
```

Consumer Process

```
while(true)
{
            while (count == 0)
                        ;
            item = Buffer[out];
            out = (out+1) % buffer_size;

Entry       wait(s);
Section

            $register_2$ = count;
            $register_2$ = $register_2$ - 1;     Critical Section
            count = $register_2$;

Exit        signal(s);
Section
}
```

Classic problems of synchronization

1) Bounded buffer or producer-consumer problem
2) Readers-writers problem
3) Dining philosopher’s problem

Readers-Writers Problem

There is a database that can be shared by a number of concurrent processes.

Some processes named ‘Readers’ only read data from the database.

Other processes named ‘Writers’ write data to the database.

At a time, any number of Readers can read data from the database without any conflict.

But, when a Writer is writing data to the database then other Writers and Readers are not allowed to access the database.

Semaphore solution to the Readers-Writers Problem

The following semaphore variables are used:

```
semphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

rw_mutex and mutex are binary semaphores.

read_count variable keeps track of how many Readers are currently reading from the database.

rw_mutex is used by both Readers and Writers to synchronously access the database.

mutex is used by Readers to synchronously access the read_count variable.

The code of a Writer process is:

```
while(true)
{
        wait(rw_mutex);

                Write data into the database;

        signal(rw_mutex);
}
```

The code of a Reader process is:

```
while(true)
{
    wait(mutex);
        read_count = read_count + 1;
        if(read_count == 1)
            wait(rw_mutex);
    signal(mutex);

        Read data from the database;

    wait(mutex);
        read_count = read_count - 1;
        if(read_count == 0)
            signal(rw_mutex);
    signal(mutex);
}
```

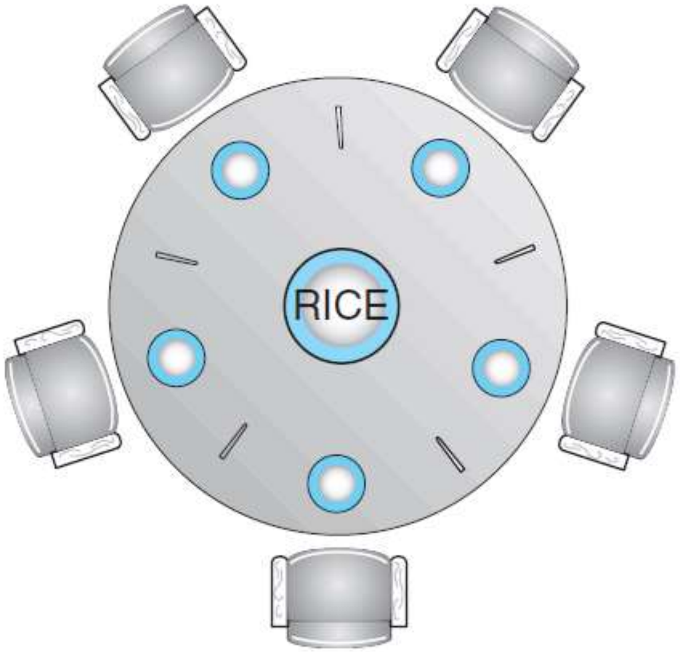
Dining Philosopher's Problem

There are 5 philosophers.

A philosopher at any time is in either thinking or eating state.

The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

In the center of the table is a bowl of rice, and the table is laid with five single chopsticks as shown in the diagram:



When a philosopher feels hungry then the philosopher tries to pick up the two chopsticks that are closest to him.

A philosopher may pick up only one chopstick at a time.

A philosopher cannot pick up a chopstick that is already in the hand of a neighbor.

When a hungry philosopher has both chopsticks at the same time, he eats without releasing the chopsticks.

When he is finished eating, he puts down both chopsticks and starts thinking again.

Semaphore solution to the Dining Philosopher's problem

One simple solution is to represent each chopstick with a semaphore.

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1.

A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.

A philosopher releases his chopsticks by executing the signal() operation on the appropriate semaphores.

The structure of philosopher ***i*** is:

```
while (true)
{
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);

                Eat for a while;

        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);

                Think for a while;
}
```

Module 3.2

Deadlock

A process is a program in execution state.

A resource is any hardware or software component present in the computer system.

Some example resources are: RAM, HD, CPU, compiler, file and so on.

There are two types of resources: sharable and non-sharable.

A sharable resource can be used by a number of processes at a time.

A non-sharable resource can be used by only one process at a time.

A computer system may have one or more instances of each resource type.

For example, a computer system may have 3 CPUs, 2 printers and so on.

A number of processes may be executing in the computer system at any particular time.

Each process requires some resources.

Operating system allocates the resources to the processes running in the computer system.

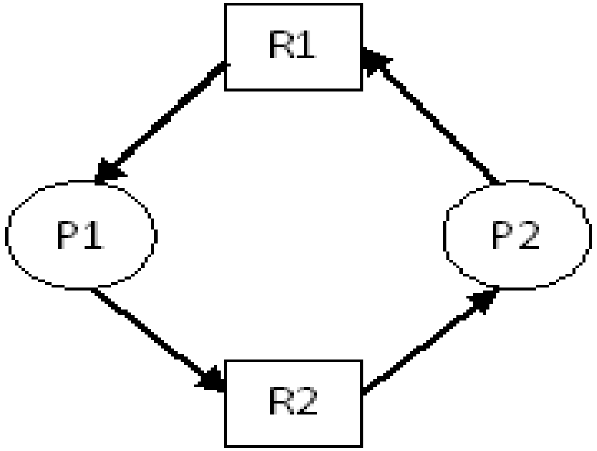
When a process requires a resource then the process makes a request to the operating system.

The operating system allocates the requested resource if that resource is free otherwise the process has to wait.

Deadlock

A set of processes is said to be in deadlock state if each process in the set is waiting for an event (releasing of resource) that can be caused by another process in the set.

Ex:



Necessary conditions for deadlock

The following four conditions must hold for the occurrence of deadlock state.

(or)

In a deadlock state, the following four conditions are satisfied.

Mutual Exclusion

Mutual exclusion condition indicates that a non-sharable resource should be used by only one process at a time.

Hold and Wait

Each process is holding some resources and waiting for other resources.

No-Preemption

A resource cannot be released from a process before completion of the process.

Circular Wait

There is a set of processes {P1, P2, P3, … , Pn} such that

P1 is waiting for P2
P2 is waiting for P3
.
.
.
.
Pn is waiting for P1

Resource Allocation Graph

Resource allocation graph is used to describe the deadlock state.

Nodes in the resource allocation graph represent both processes and resources.

Edges indicate both allocation of resources to processes and request to resources by the processes.

Processes are indicated using circles, resources are indicated using boxes.

A request edge is from a process to a resource.

An allocation edge is from a resource to a process.

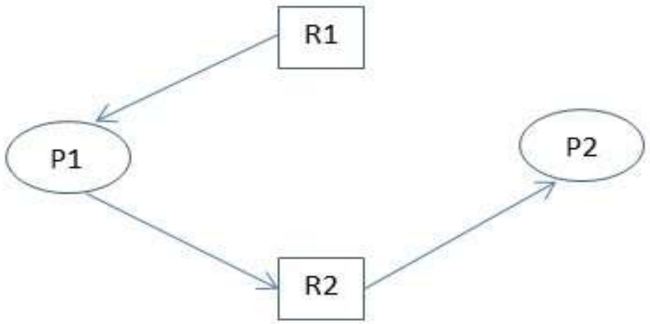
If a resource has number of instances then that number of instances is indicated using number of dot symbols inside the box representing that resource.

When a process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph.

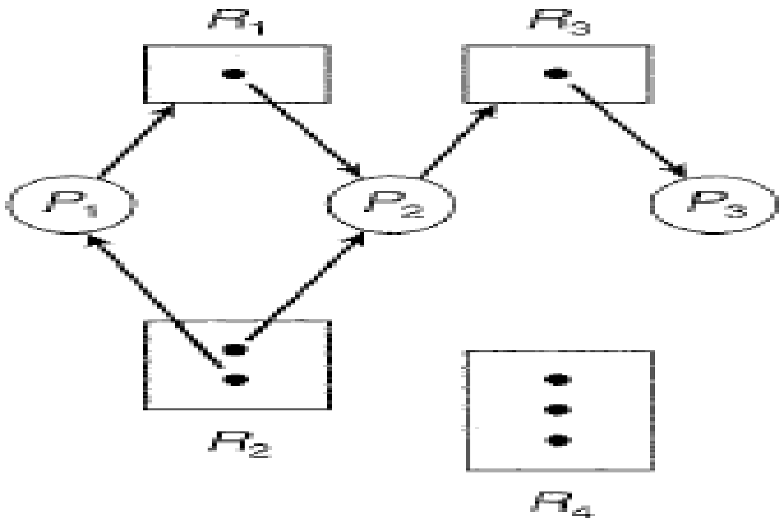
When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an allocation edge.

When the process no longer needs access to the resource, it releases the resource.

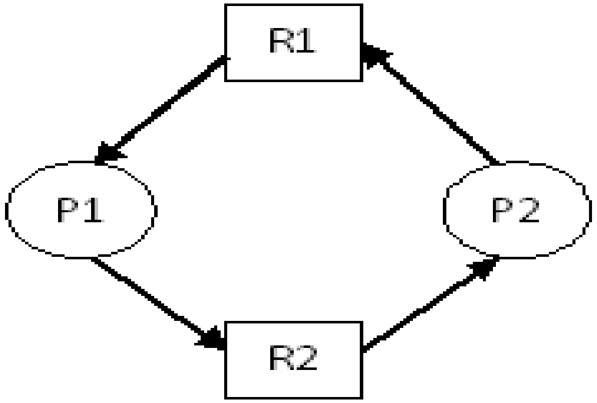
As a result, the allocation edge is deleted.



There is no cycle in above resource allocation graph. There is no deadlock.



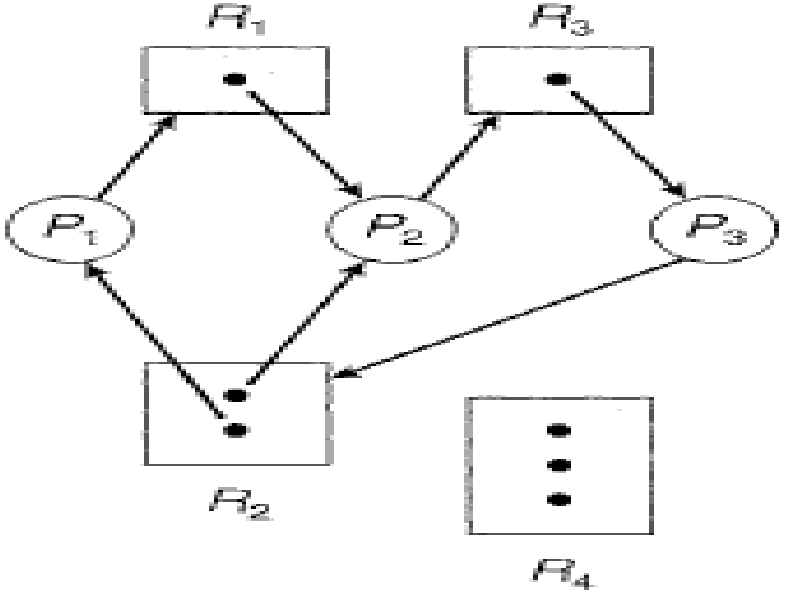
There is no cycle. There is no deadlock.



There is a cycle:

P1 -> R2 -> P2 -> R1 -> P1

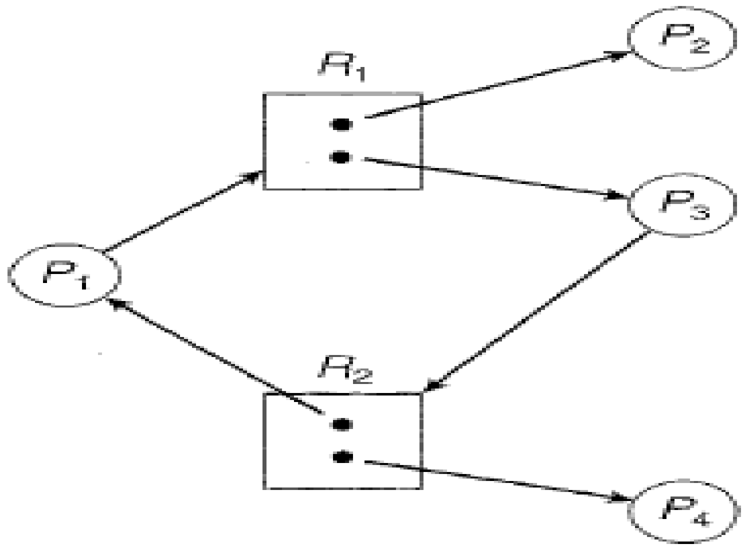
Processes P1 and P2 are in deadlock state.



There are two cycles:

P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1
P2 -> R3 -> P3 -> R2 -> P2

Processes *P1, P2,* and *P3* are in deadlock state.



There is a cycle:

P1 -> R1 -> P3 -> R2 -> P1

There is no deadlock.

Conclusions from the above examples are:

1) If there is no cycle in the Resource Allocation Graph then there is no deadlock state.

2) If each resource has only one instance then the presence of cycle in the resource allocation graph indicates the occurrence of deadlock state.

3) If the resources have multiple instances then the presence of cycles in the resource allocation graph may or may not indicate the occurrence of deadlock state.

Methods for handling deadlocks

1) Deadlock prevention
2) Deadlock avoidance
3) Deadlock detection
4) Deadlock recovery

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given additional information in advance concerning the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

With this additional knowledge, the operating system can decide for each request whether the request can be satisfied or must be delayed.

Deadlock detection detects the occurrence of deadlock state in the computer system.

Deadlock recovery recovers the computer system from the deadlock state when it occurs.

Deadlock prevention

For a deadlock to occur, each of the four necessary conditions must hold.

We can ***prevent*** the occurrence of a deadlock by ensuring that at least one of these conditions cannot hold.

Mutual Exclusion

To ensure that mutual exclusion condition cannot hold, a resource can be allocated to a number of processes at a time.

This is possible only with sharable resources.

For non-sharable resources, it is not possible to ensure that mutual exclusion condition cannot hold.

Hold and Wait

Any one of the following two mechanisms is used to ensure that hold and wait condition cannot hold.

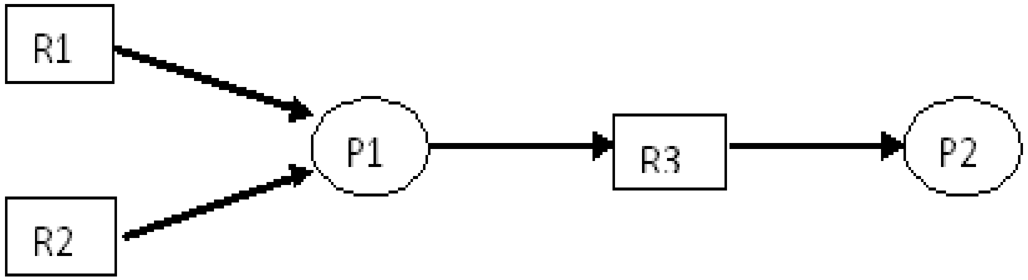
1) Allocate all required resources to the process before starting the execution of the process.

2) Allocate a subset of required resources to the process. Release this subset of resources from the process after using them and allocate another subset of resources. Repeat this procedure till the completion of process.

No preemption

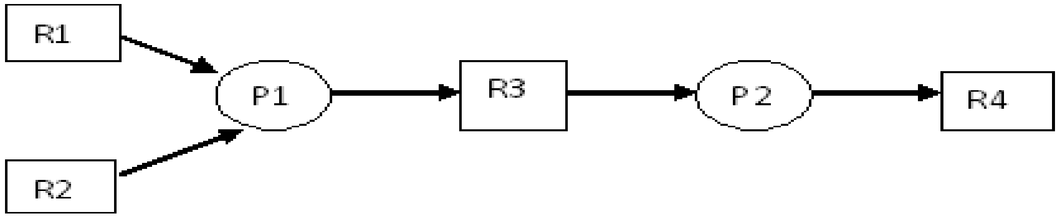
The following two mechanisms are used to ensure that no preemption condition cannot hold:

1) When a process requests for any resource which is currently allocated to a running process then the requesting process will go the waiting state and the resources that are currently allocated to the requesting process are released.



Resources R1 and R2 are released from P1

2) When a process request for a resource which is currently allocated to a waiting process then the resource from the waiting process is released and allocated to the requesting process.



Resources R3 is released from P2

Circular wait

The following mechanism is used to ensure that circular wait condition cannot hold:

An ordering (R_1, R_2, R_3, …..) is given to the resources and the processes are allowed to request the resources in an increasing order.

When a process request for a resource whose number is less than the number of any resource already allocated to the process then that request is rejected by the operating system.

For example, if a process P_1 is holding resources R_2 and R_5 and if it requests for resource R_7 then the request is accepted.

If P_1 request for R_3 then that request is rejected.

Deadlock Avoidance

To apply deadlock avoidance method, the following information must be known in advance:

1) the resources currently available
2) the resources currently allocated to each process
3) the future requests and releases of each process

The following algorithms are used for deadlock avoidance:

1) Resource-Allocation-Graph algorithm
2) Banker's algorithm

Resource-Allocation-Graph Algorithm

Resource-Allocation-Graph algorithm is used when each resource has only one instance.

In addition to the request and allocation edges, a **claim edge** is also included in the graph.

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future.

This edge resembles a request edge in direction but is represented in the graph by a dashed line.

When process P_i requests resource R_j, the claim edge P_i → R_j is converted to a request edge.

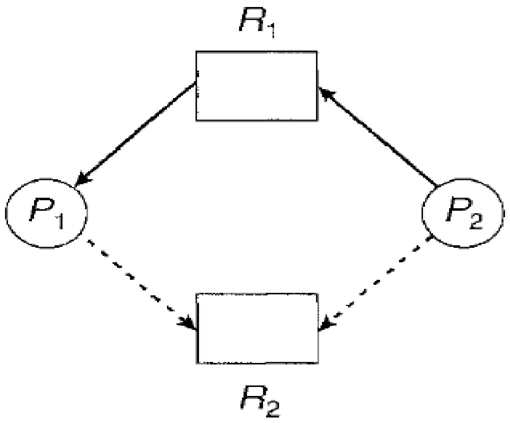
Similarly, when a resource R_j is released by P_i, the allocation edge R_j →P_i is reconverted to a claim edge P_i → R_j.

Before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph.

Now suppose that process P_i requests resource R_j.

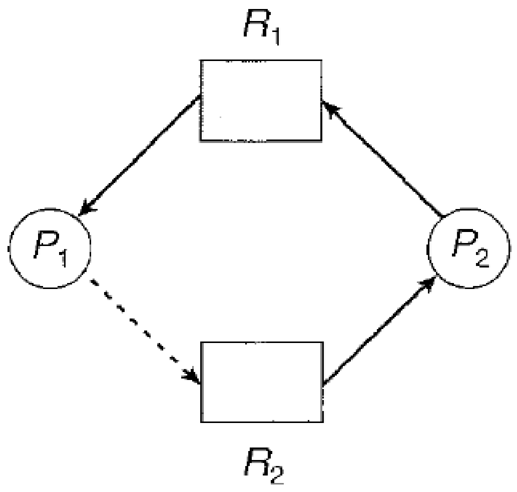
The request can be granted only if converting the request edge P_i → R_j to an allocation edge R_j → P_i does not result in the formation of a cycle in the resource-allocation graph.

To illustrate this algorithm, we consider the following 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 below



A cycle indicates that the system is in an unsafe state.

Safe state

If all processes can be completed in any order with available resources then the system is said to be in safe state. Otherwise, the system is said to be in unsafe state.

A safe state is not a deadlocked state.

An unsafe state may lead to a deadlock.

Safe Sequence

The order in which the processes can be completed is called the safe sequence.

Banker’s Algorithm for Deadlock Avoidance

Banker’s algorithm is used when resources have multiple instances or copies.

The following procedure is used to avoid the occurrence of deadlock state:

When any process request for resources then the algorithm checks whether the allocation of requested resources will lead to (or) result in the safe state.

If the allocation of requested resources to the process will lead to safe state, then the operating system accepts the request. Otherwise, rejects the request.

Four data structures are used in Banker’s algorithm for deadlock avoidance.

1) Available: is a vector of size m. Where m is the number of resource types.

Available vector indicates the number of copies of each resource type available in the system.

2) Max: is a matrix of size nxm. Where n is the number of processes.

Max matrix indicates the total number of copies of each resource type required by each process.

3) Allocation: is a matrix of size nxm.

Allocation matrix indicates the number of copies of each resource type allocated to each process.

4) Need: is a matrix of size nxm.

Need matrix indicates the number of copies of each resource type still required by each process.

Need matrix is calculated as

Need = Max - Allocation

Ex: n=4, m=2

Max

	A	B
P0	7	5
P1	3	2
P2	9	0
P3	2	2

Allocation

	A	B
P0	0	1
P1	2	0
P2	3	0
P3	2	1

Available

A	B
3	3

Need

	A	B
P0	7	4
P1	1	2
P2	6	0
P3	0	1

Banker’s algorithm for deadlock avoidance has two parts:

1) Resource request algorithm (main algorithm)
2) Safety algorithm (sub algorithm)

Safety algorithm

This algorithm is used to check whether the system is in safe state or not.

Steps in the algorithm are:

1. Find a process P_i which is not marked as finished and **$Need_i$ <= Available**. If no such process exists then go to step 3.

2. Mark the process P_i as finished and update the Available vector as

$$\textbf{Available = Available + Allocation}_i$$

Go to step 1.

3. If all processes are marked as finished, then indicate that the system is in safe state. Otherwise, indicate that the system is in unsafe state.

Ex: n=4, m=2

Max	A	B
P0	7	5
P1	3	2
P2	9	0
P3	2	2

Allocation	A	B
P0	0	1
P1	2	0
P2	3	0
P3	2	1

Available	
A	B
3	3

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

Above state is safe state as all processes can be completed in the order: P1, P3, P0, P2.

At first, process P1 is selected as it is not marked as finished and its **need$_1$<=Available** ((1,2)<=(3,3)).

Process P1 is marked as finished, the resources allocated to process P1 are released and then added to the available vector.

Available	
5	3

Next, process P3 is selected as it is not marked as finished and its **need$_3$<=Available** ((0,1)<=(5,3)).

Process P3 is marked as finished, the resources allocated to process P3 are released and then added to the available vector.

Available
7 4

Next, process P0 is selected as it is not marked as finished and its **need$_0$<=Available** ((7,4)<=(7,4)).

Process P0 is marked as finished, the resources allocated to process P0 are released and then added to the available vector.

Available
7 5

Next, process P2 is selected as it is not marked as finished and its **need$_2$<=Available** ((6,0)<=(7,5)).

Process P2 is marked as finished, the resources allocated to process P2 are released and then added to the available vector.

Available
10 5

Resource request algorithm

When a process P_i makes a $Request_i$ for resources then the **resource request algorithm** is invoked to decide whether to accept or reject the request.

The steps in the algorithm are:

1. If **$Request_i$ <= Available** then go to step 2. Otherwise, the process P_i has to wait.

2. Pretend that the requested resources are allocated to the process and update the values of matrices as

   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Need_i - Request_i$
   $Available = Available - Request_i$

3. Check whether the resulting state is safe or not by calling the safety algorithm. If the resulting state is safe, then accept the request. Otherwise, reject the request and restore the old state.

Ex: n=4, m=2

Max

	A	B
P0	7	5
P1	3	2
P2	9	0
P3	2	2

Allocation

	A	B
P0	0	1
P1	2	0
P2	3	0
P3	2	1

Available

A	B
3	3

Need

	A	B
P0	7	4
P1	1	2
P2	6	0
P3	0	1

The above state is safe as the processes can be completed in the order: P1, P3, P0, P2.

If the process P1 makes a request

Request$_1$ = (1, 0)

Then the operating system calls the **resource request algorithm** to decide whether the request can be accepted or not.

The **resource request algorithm** checks whether **Request$_1$<=Available** or not.

As **Request$_1$<=Available ((1,0)<=(3,3))**, the matrices are modified as

	Allocation		Need		Available	
	A	B	A	B	A	B
P0	0	1	7	4	2	3
P1	3	0	0	2		
P2	3	0	6	0		
P3	2	1	0	1		

Now, the safety algorithm is called to determine whether the resulting state is safe or not.

The safety algorithm returns that the state is safe as the processes can be completed in the order: P1, P3, P0, P2.

So, the operating system accepts the request.

Ex: n=4, m=2

Max

	A	B
P0	7	6
P1	3	2
P2	9	3
P3	2	2

Allocation

	A	B
P0	0	1
P1	3	0
P2	3	0
P3	2	1

Available

A	B
2	3

Need

	A	B
P0	7	5
P1	0	2
P2	6	3
P3	0	1

In the above state, if process P0 makes a request

Request$_0$ = (0, 2)

Then the operating system calls the resource request algorithm to decide whether the request can be accepted or not.

The resource request algorithm checks whether **Request$_0$<=Available** or not.

As **Request$_0$<=Available ((0,2)<=(2,3))**, the matrices are modified as

	Allocation	
	A	B
P0	0	3
P1	3	0
P2	3	0
P3	2	1

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

Available	
A	B
2	1

Now, the safety algorithm is called to determine whether the resulting state is safe or not.

The safety algorithm returns that the state is unsafe as it is not possible to complete all processes.

So, the operating system rejects the request.

Deadlock detection

Deadlock detection method is used to check the occurrence of deadlock state in the computer system.

Two methods are used for detecting the occurrence of deadlock state.

1) Wait-for-graph algorithm
2) Banker’s algorithm

Wait-for-graph algorithm

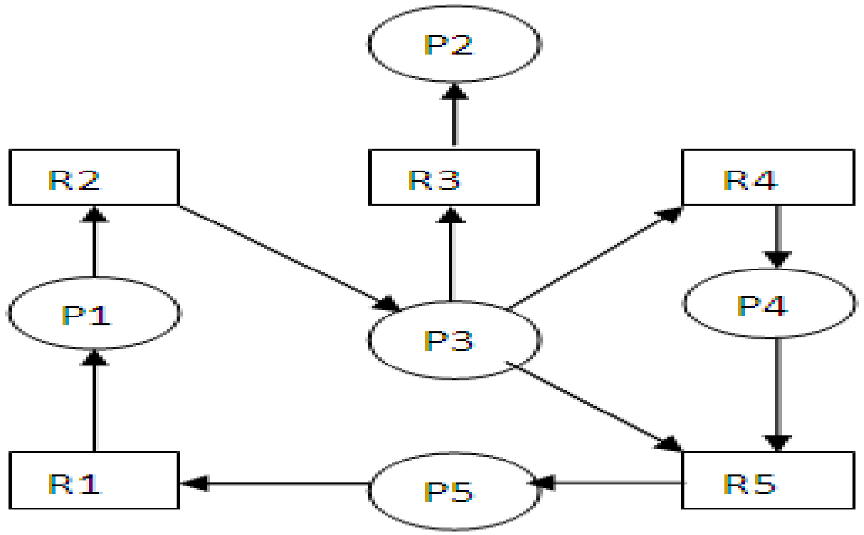
Wait-for-graph algorithm is used when each resource has only one instance.

In this algorithm, a wait-for-graph is constructed based on the resource allocation graph.

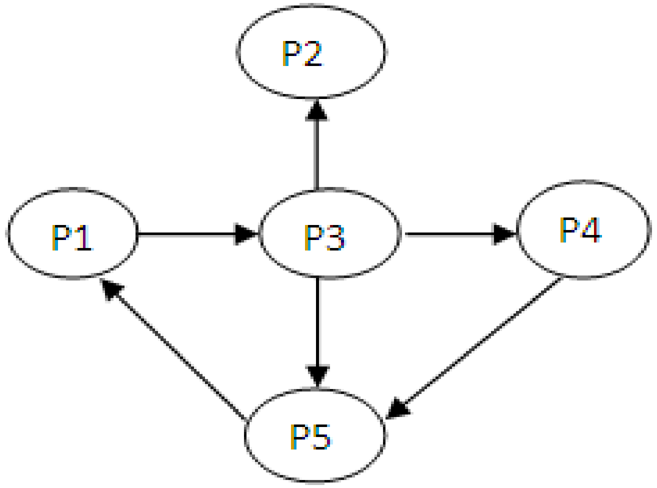
Wait-for-graph contains only the process nodes and they are indicated by circles.

If a process P_i is waiting for a resource held by process P_j, then an edge is drawn from process P_i to process P_j in the wait-for-graph.

Presence of cycles in the wait-for-graph indicates occurrence of the deadlock state.



Resource allocation graph



Wait-for-graph

There are 2 cycles in the wait-for-graph

Cycle 1: P1->P3->P5->P1
Cycle 2: P1->P3->P4->P5->P1

So, deadlock has occurred in the system.

The processes P1, P3, P4 and P5 are in deadlock state.

Process P2 is not involved in any cycle. P2 is not in deadlock state.

Banker’s algorithm for Deadlock Detection

Banker’s algorithm is used when the resources have number of instances or copies.

The following data structures are used in this algorithm.

Available: It is a vector of size m. Where m is number of resource types.

Available vector indicates the number of available copies of each resource type.

Allocation: It is a matrix of size nxm. Where n is number of processes.

Allocation matrix indicates the number of copies of each resource type that are currently allocated to each process.

Request: It is a matrix of size nxm.

Request matrix indicates the number of copies of each resource type that are required by each process.

Algorithm

1. Find a process P_i which is not marked as finished and **$Request_i$ <= Available**. If no such process exist then go to step 3.

2. Mark the process P_i as finished and update the Available vector as

$$\textbf{Available = Available + Allocation}_i$$

Go to step 1.

3. If all processes are marked as finished, then indicate that deadlock state has not occurred. If some of the processes are not marked as finished, then indicate that deadlock has occurred. The processes which are not marked as finished are in the deadlock sate.

Ex:

Allocation

	A	B
P0	0	1
P1	2	0
P2	3	0
P3	2	1

Request

	A	B
P0	0	0
P1	2	0
P2	0	0
P3	1	0

Available

A	B
0	0

At first, process P0 is selected as it is not marked as finished and its **Request_0 <= Available.**

Process P0 is marked as finished and the available vector is updated as

Available
0 1

Next, process P2 is selected as it is not marked as finished and its **Request_2 <= Available**.

Process P2 is marked as finished and the available vector is updated as

Available
3 1

Next, process P1 is selected as it is not marked as finished and its **Request$_1$ <= Available**.

Process P1 is marked as finished and the available vector is updated as

Available
5 1

Finally, process P3 is selected as it is not marked as finished and its **Request$_3$ <= Available**.

Process P3 is marked as finished and the available vector is updated as

Available
7 2

Deadlock state has not occurred in the system as all processes are marked as finished.

Ex:	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
P0	0	1	0	P0	0	0	0	0	0	0
P1	2	0	0	P1	2	0	2			
P2	3	0	3	P2	0	0	1			
P3	2	1	1	P3	1	0	0			
P4	0	0	2	P4	0	0	2			

First, process P0 is selected as it is not marked as finished and its **Request$_0$ <= Available**.

Process P0 is marked as finished and the available vector is updated as

Available
0 1 0

The request of other processes cannot be satisfied with available resources.

So, the processes P1, P2, P3 and P4 are not marked as finished and it indicates the occurrence of deadlock state in the system.

The processes P1, P2, P3 and P4 are in the deadlock state.

Deadlock Recovery

Any of the following two techniques is used to recover from the deadlock state:

1) Aborting the processes that are involved in the deadlock state.
2) Preempting the resources from the deadlocked processes.

Aborting the processes that are involved in the deadlock state

In this method, the processes which are involved in the deadlock state are terminated.

There are two ways for aborting the deadlocked processes.

1) Abort all processes at a time.
2) Abort the processes one by one until the deadlock cycle is eliminated.

In the first mechanism, all processes that are involved in the deadlock state are terminated or aborted.

<u>Advantages</u>

1. Deadlock can be recovered in less amount of time.
2. No need to apply the deadlock detection algorithm for number of times.

Disadvantage

1. Loosing of computations of all processes.

In the second mechanism, first one of the processes in the deadlock state is terminated.

After that the presence of deadlock state is checked.

If the deadlock state still exist, then one more process in the deadlock state is terminated.

This procedure is repeated until the system is recovered from the deadlock state.

Advantage

1. Less wastage in computation time compared to the first mechanism as only some processes in the deadlock state are terminated.

Disadvantage

1. Need of applying the deadlock detection algorithm for number of times.
2. Some criteria have to be used to select a process for termination at each step.

The general criteria used to select a process for termination are:

1) Execution time

The process that has been executed for less amount of time is selected for termination.

2) No of Resources

The process which is holding less number of resources is selected for termination.

3) Priority

The process with low priority is selected for termination.

Preempting the resources from the deadlocked processes

In this method, resources will be preempted successively from the deadlocked processes until the system is recovered to the normal state.

To use this mechanism for recovering from the deadlock state, the following factors are considered.

1. Selecting a victim (resource)
2. Rollback
3. Starvation

Selecting a victim: one of the deadlocked processes has to be selected and then one of the resources allocated to that process needs to be selected based on some criteria like execution time, number of resources allocated, priority and so on.

Rollback: When a resource is preempted from a deadlocked process then that process needs to be restated from the beginning.

Starvation: resources should not be preempted from the same process in each step.

Ex: n=5, m=4

Max

	A	B	C	D
P0	0	0	1	2
P1	1	7	5	0
P2	2	3	5	6
P3	0	6	5	2
P4	0	6	5	6

Allocation

	A	B	C	D
P0	0	0	1	2
P1	1	0	0	0
P2	1	3	5	4
P3	0	6	3	2
P4	0	0	1	4

Available

A	B	C	D
1	5	2	0

Need

	A	B	C	D
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Process P1 makes a request

Request$_1$ = (0,4,2,0)

Can the request be accepted or rejected?

Allocation

	A	B	C	D
P0	0	0	1	2
P1	1	4	2	0
P2	1	3	5	4
P3	0	6	3	2
P4	0	0	1	4

Need

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

Available

A	B	C	D
1	1	0	0