UNIT - II Process Synchronization & Dead locks

=) Process Synchronization:-

mutual understanding b/w 2 processes at the time of Execution.

=) Critical Section:-

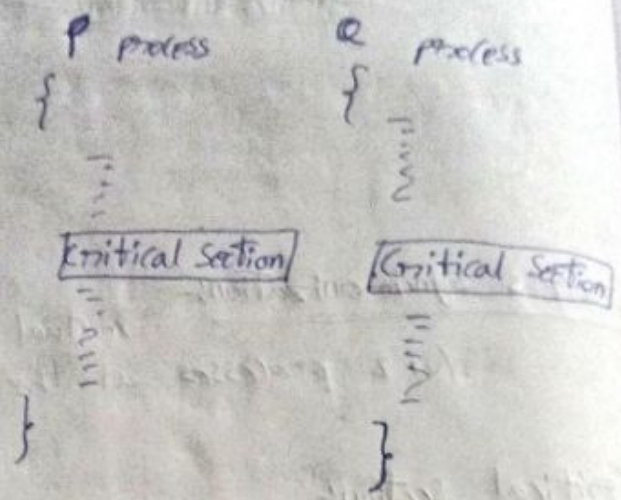
Each & every process will have a small segment of code in it, which is called Critical Section.

when 1 process executing it's Critical Section, the other processes are not allowed to execute their Critical Sections, because during the execution of Critical section, the following things may happen.

a) A variable value may be changed.
b) The table may be modified.
c) A file may be updated.

If we allow more than one process to execute their Critical Sections concurrently, then the Consistency of data will be affected.

Ex:-



=) Critical Section syntax:-

=)

```
do{
    Entry condition
        ↓
    Critical Section
        ↓
    Exit Condition
        Remainder Section;
}while();
```

using do while loop.

25/11/21

=) Solution for Critical Section problem:-

A Critical Section problem can be solved by the following issues.

a) mutual Exclusion
b) progess
c) Bounded time

=) peterson's

Section

2 process

is nothing but,

algorithm

The A

Algorithm:

=) Peterson's Solution for 2 process Critical Section problem:-

This peterson's Solution Solves 2 Process Critical Section problem. It's algorithm is nothing but the combination of the key points of algorithm ① & algorithm ②.

The Algorithm is as follows:-

Algorithm:-

$\begin{cases} i=0 \\ j=1-i \end{cases}$

```
Boolean flag[2];
int turn;
.
.
.
do
{
    flag[i] = true;
    turn = i;
    while(turn == j && flag[j])

    [Critical Section]

    flag[i] = false;

    Remainder Section;
} while();
```

26/11/21

=> Hardware Synchronization:-

2 processes critical section problem can be solved by using Hardware Synchronization concept.

2 types of Hardware Synchronization's available to Solve critical section problem.

a) Test() / Set() instruction.
b) Swap Instruction.

a) Test(), Set() instructions:-

Test() is a fn which checks the Status of a process. It returns a boolean value either true (or) false.

Set() is a fn which sets the function process so that one process is allowed to execute Critical Section part & the other process is made wait.

=> Syntax for Test() & Set() Instruction:-

```
do{
    while(Test and Set(lock))
        [Critical Section]
    lock = false;
        Remainder Section;
}while;
```

b) Swap Instru

be used for mutua
declared as
FALSE. An
as TRUE

=> Synta

=> Software

-zation, t
tools, Av
There
tools

b) Swap Instruction:-

It is a function which can be used for mutual exclusion. A variable (lock) is declared as boolean variable & initialized as FALSE. Another boolean variable 'key' is initialized as TRUE.

=> Syntax for Swap Instruction:-

```
do
{
    Key = TRUE;
    while(Key == TRUE)
        Swap(lock, key);
        [Critical Section]
    lock = false;
        Remainder Section;
}while();
```

=> Software Synchronization:-

Like, Hardware Synchronization, there are some Software Synchronization tools, Available to Solve Critical Section problem. There are 2 popular Software Synchronization tools available are:-

a) Semaphore
b) Monitor

a) Semaphore:-

It is defined as a 'binary variable' which is used to solve Critical Section problem. It's a Software Synchronization tool, denoted by s.
It uses 2 primitives:-

a) WAIT(s)
b) SIGNAL(s)

The WAIT(s) & SIGNAL(s) primitives can be defined as follows:

WAIT(s): / If s = 1, then make s=0, & allow process to Execute it's Critical Section part.
/else
place the process in a queue.

SIGNAL(s): / If queue ≠ Empty, remove the process from the queue & allow to Execute it's Critical Section part.
/else
make s = 1.

Note:-

If s = 0, Some process is Executing it's Critical Section part.

If s = 1, No process is executing it's Critical Section part.

=) Classical prob

a) produ
b) Dini
c) Si

a) producer

Produce

{

wait(

}

critica

(produci

{

Sign

Rem

}

Pro

{

P

=) Classical problems of Synchronization:-

a) producer – consumer
b) Dining – philosopher
c) Sleeping Barber

a) producer – consumer problem:-

S = 1(c), 0(p)

producer

```
{
  wait(s);
  {
  critical section;
  (producing an item)
  {
  signal(s);
  }
  Remainder section;
}
```

consumer

```
{
  wait(s);
  {
  critical section;
  (item is consumed);
  {
  signal(s);
  }
  Remainder section;
}
```

Producer is a process which produces an item, & consumer is a process which will consume the item called consumer process. In both processes, wait(s) is Entry & signal(s) is Exit conditions. First, consumer process is ideal.

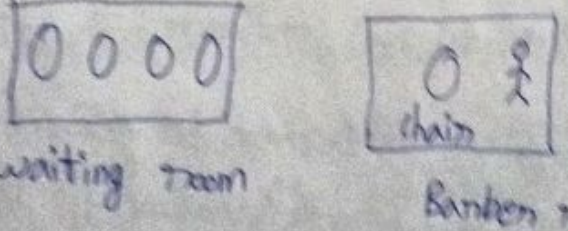
Solution for dining philosophers problem using Semaphore:-

```
Semaphore chopstick[5];

{

do
{
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5]);

    Eating

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

    Thinking

}while();
```

Structure of philosophers Eating process

~~One~~ 2 wait() & 2 signal() conditions are used. Since each philosophers needs 2 chopsticks.

2) Sleeping Barbers problem:-



In this problem, there will be a barbers room where (in room) only one chair is available. Barber serves

only one customer at a time. If there is no customer then he will sleep. The works done by the Barber are:-

a) Sleeping

b) working

If more than one customer comes at a time if the barber is busy with his work, then other want to wait in a room called waiting room. Here, the critical section problem is, if customer is there, he will serve, otherwise he will go sleep. If all chairs are occupied in the waiting room, if a new customer enters, he will leave

⇒ **Monitor:-**

It is a software synchronization tool, similar to Semaphore, which is used to solve critical section problem.

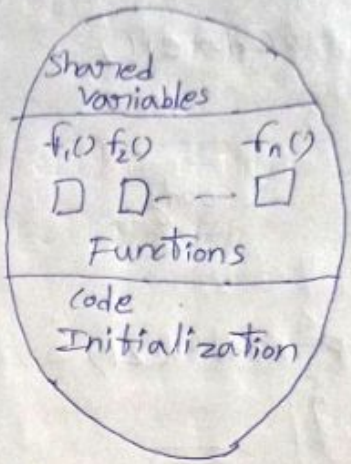
⇒ **Syntax of a monitor:-**

```
monitor monitor_name
{
  shared variables declarations
    procedure1()
    {
      ==
       body
      ==
    }
    procedure2()
    {
      ==
       body
      ==
    }
    procedure_3()
    {
      =
       body
      =
    }
    code initialization;
}
```

A monitor can be divided into 3 parts:-

a) Shared variables declarations.
b) Functions/procedures.
c) Code Initialization.

=> The schematic view of a monitor.



It uses a special data type called "condition" data type, & the monitor uses 2 primitives to solve the critical section problem.

a) wait()
b) signal()

The condition variable will be shared by multiple process.

```
P
{
    condition x;
    ---
    ---
    x.wait();
    ---
    ---
}
```

```
Q
{
    condition x;
    ---
    ---
    Critical Section
    ---
    ---
    x.signal();
}
```

If a process, for example "P" executes x.wait() then the process "P" will be suspended immediately & suspended process "P" can be involved when some other process executes x.signal()

=> Solution for Dining philosophers problem using monitors:-

```
monitor DP
{
    enum { thinking, hungry, eating };
    enum  State[3];
    Condition self[5];
    Void pick_up(int i)
     {
        State[i] = hungry;
        test(i);
        if(State[i] != eating)
           self[i].wait();
     }
    Void putdown(int i)
    {
       State[i] = thinking;
       test(i+4)%5;
       test(i+1)%5;
    }
```

```
void test(int i)
{
    if((state[(i+4)%5] != eating && state[i] ==
                    hungry && (state[(i+1)%5] != eat
    {
        state[i] = eating;
        self[i].signal();
    }
}
void init()
{
    for(i=0; i<5; i++)
    {
        state[i] = thinking;
    }
}
```

⇒ The purpose of above functions are;-

a) pick up();-

This function is used to pickup 2 chopstic to start(initiate) Eating process.

b) put down();-

This fn is used to put down the chopsticks after eating process.

c) Test();-

This fn is used to check the availability

1/12/21

=> Deadl

P- process

R- Resource

Each proc

But, R_2

To complete

R_1 is re

process

If 'p'

Q. So

But w

Resour

all.

=> Syste

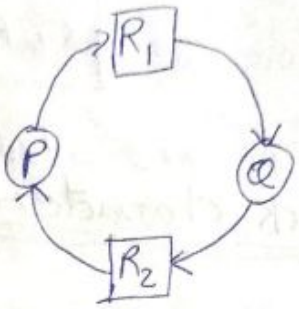
of the chopsticks.

1/12/21

⇒ **Deadlocks :-**

P- process - ○

R- Resource - □



There are 2 processes for Execution. Each process requires 2 resources R_1 & R_2. But, R_2 is given to P & R_1 is given to Q. To complete the Execution of first process P, R_1 is required (R_2 is with P). To Execute the process of Q, R_2 is required (R_1 is with Q). If 'P' completes the Execution R_2 can be given to Q. So, Q can completes its Execution (or) Vice versa. But w/o Execution no process will release the Resources. So, no process will be Executed at all. This situation is called 'Dead lock'.

⇒ **System models :-**

Under normal mode of operation, a process will get resources & use them in

the following way.

a. Request for Resource
b. Use of Resource
c. Release of Resource

=> Dead lock Characterization:-

Dead lock can be characterized based upon some conditions, Those conditions are called "necessary conditions of a dead lock."

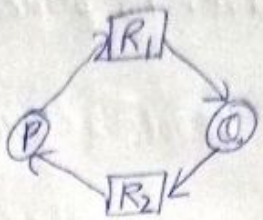
=> Necessary conditions of a deadlock:-

There are 4 conditions which are called the necessary conditions of a Dead lock.

a) Mutual Exclusion:-

At least 1 resource should be in non-sharable mode, then mutual understanding b/w 2 processes will not be available, i.e., a hardware resource can't be shared by more processes at a time.

b) Hold & wait:-



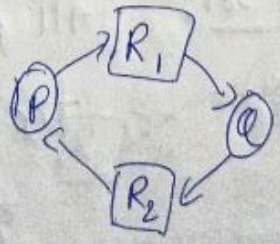
If a process is holding a resource & is waiting for another resource, this leads to dead lock.

Ex:- 'P' is holding the resource 'R2' & waiting for 'R1'.

c) No preemption:-

A resource can be released by a process only after completion of it's execution. At any cost, the process will not preempt the resource before execution.

d) Circular wait:-



$P \rightarrow R_1 \rightarrow Q \rightarrow R_2$

There Should not be any circular wait in the

system. i.e. no process should wait for any other process execution.

2/11/21

=) Resource allocation graph
(or)
System Resource allocation graph:-

It is a digraph which is used to find whether a deadlock is available in the system/not.

Resource allocation graph contains 2 sets

1. A vertex set
2. An Edge set

Again a vertex set contains 2 sets,

a) process set:- All processes available in the system are available here.

Ex:- process set: {P1, P2, P3, P4}

b) Resource set:- All resources available in the system are available here.

Ex:- Resource set : $\{R_1, R_2, R_3\}$

An Edge is made into 2 sets,

a) Assigned Edge:-

Resources allocated to a process is represented in an assigned Edge.

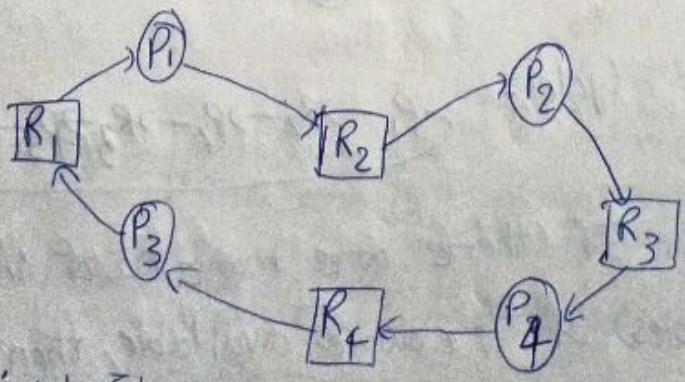
Ex:- $R_1 \rightarrow P_1$, $R_3 \rightarrow P_4$, i.e. Resource → process.

b) Request Edge:-

processes waiting for resources, are represented in a request Edge.

Ex:- $P \rightarrow R_1$, $Q \rightarrow R_2$, i.e. process → Resource

⇒ Example for System Resource Allocation Graph:-



Assinged Edges:- $R_1 \rightarrow P_1$, $R_2 \rightarrow P_2$, $R_3 \rightarrow P_4$, $R_4 \rightarrow P_3$

Request Edges:- $P_1 \rightarrow R_2$, $P_2 \rightarrow R_3$, $P_4 \rightarrow R_4$, $P_3 \rightarrow R_1$

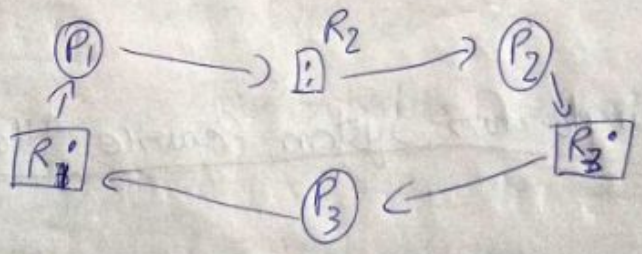
Note:-

Sometimes, even there is a cycle, deadlock may not occurs in the Syst...

=> Cycle:- $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_4 \rightarrow R_4 \rightarrow P_3 \rightarrow R_1$

If there is a cycle (combo of assign... & request Edges), then there is a possibi... of dead lock available in the System. There is a cycle, so deadlock occurs.

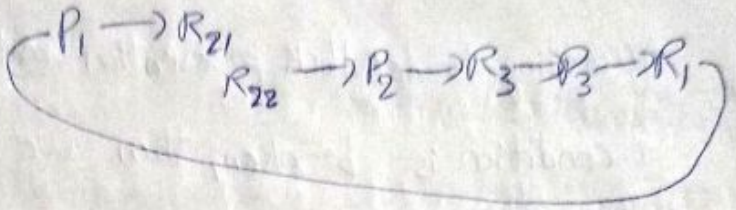
=> A resource allocation graph with cycle but without deadlock:-



Cycle:- $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_1$

If there are more no. of instances (copies) of resources available, then deadlock can be avoided. Here 2 copies of R_2 i.e. R_{21} & R_{22}, so, R_{21} can be given to P_1

& R_{22} can be given to P_2 so that cycle can be broken. So, deadlock can be avoided.



3/12/21

⇒ **Deadlock Handling methods:-**

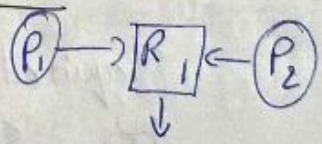
There are 3 popular methods available to handle dead locks.

The deadlock handling methods are;-

a) Deadlock prevention
b) Deadlock Avoidance
c) Deadlock detection and Recovery

a) **Deadlock prevention:-**

a) **mutual Exclusion:-**



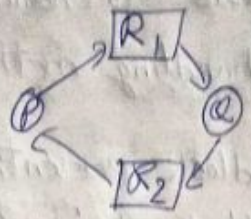
H/w resource / non-sharable resource.

If there are sharable resources like files, mutual understanding b/w 2 processes can exist. If only non-sharable resources are available like printer, then mutual Exclusion can't be taken.

Note:-

Deadlock will occur in a system, whenever all 4 conditions exist at a time. According to deadlock prevention method, if atleast 1 condition is broken, then we can prevent deadlock occurrence.

b) Hold & wait:-



Hold & wait() conditions can be broken by using 2 protocols.

a) Protocol 1:-

This protocol allocates all resources required by a process before it starts it's execution.

b) protocol 2:-

This protocol allocates all resources to a process, when the process is not holding any resource.

Scanned By Camera Scanner

c) No preemption:-

This no preemption condition can be broken by using one protocol.

protocol:-

This protocol allocates to a process, if the resource is available. If the resource is not available, then the resource already given to the process will be taken back.

d) Circular wait:-

To avoid circular wait, each resource is given a "unique number" & each process is instructed to request for a particular resource in either ascending/ descending order. A fn may be defined as,

function.

$$F: R_i \rightarrow k$$

R_i – resources

k – integers

If a process requests for a resource with unique number 2, then the next time, the process should request a resource greater than 3.

Deadlock avoidance:-

// Safe state:-

Deadlock prevention method never allow a deadlock in the System. This deadlock avoidance method will try to avoid the deadlock based on the State of System.

a) Safe State:-

System must allocate resources, whenever a process makes a request, then the System it is in Safe state. deadlock never occur.

b) unsafe state:-

System can't allocate resources, whenever a process makes a request, then the System is in unsafe state. It leads to deadlock, when the System is in unsafe state.

c) Safe sequence:-

The processes in the System can be executed in a particular order. So, deadlock won't occur. This sequence is called Safe sequence.

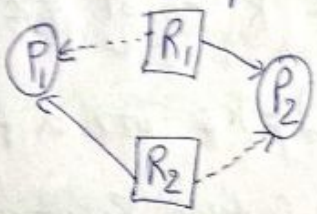
Ex:- P_1, P_2, P_3, P_4 are processes. If they're Executed in the order P_2, P_3, P_1, P_4 then

deadlock will not occur. The safe sequence can be represented as $<P_2, P_3, P_1, P_4>$

There are 2 Algorithms for deadlock Avoidance method;-

a) Resource Allocation graph Algorithm.
b) Banker's Algorithm

a) Resource Allocation graph Algorithm;-



Along the assigned edge & requested Edge, it also have an Edge called "claim edge". This claim edge is represented as

resource ----> process, i.e. R_1 ----> P_1.

meaning of claim edge is now a process is waiting for a resource & in the near future, it will be converted into assigned Edge.

b) Banker's Algorithm;-

It requires that a process must inform os that it requires a maximum no. of resources in advance;-

There are 4 different data structures used in Banker's Algorithm:-

a) Available[j] = K; There are K no. of resources of type R_j are available in the system.

b) max[i,j] = K; A process P_i requires a maximum of 'K' no. of resources of type R_j

c) Allocated[i,j] = K; For a process P_i, K no. of resources of type R_j are allocated.

d) Need[i,j] = K: A process P_i still needs 'K' no. of resources of type R_j.

The need matrix can be calculated as,

Need[i,j] = max[i,j] - allocated[i,j]

Banker's Algorithm contains 2 different algorithms:-

a) Safety Algorithm
b) Resource Allocation Algorithm

a) Safety Algorithm:-

Step 1:- Let work = Available &
Finish[i] = false, for i=0,1,2--

Step 2:- Find an 'i' such that

a) Finish[i] = false;

b) Need ≤ work;

If no such i exists goto Step4.

Step 3:- work = work + allocation;

Finish[i] = true;

goto Step 2;

Step 4:- if finish[i] = True, for all i, then System is in Safe State.

This safety Algorithm is used to find whether System is in Safe/unsafe State.

b) Resource Allocation Algorithm :-

Step 1:- If Request ≤ need goto 2, else print error, because process exceeds the limit.

Step 2:- If Request ≤ Available goto 3, else process has to wait because resource isn't Available.

Step 3:- Available = Available − Request

Allocation = Allocation + Request

Need = Need − Request

note:- By getting the information Safety algorithm, this resource allocation algorithm will be executed provided Safety Algorithm gives output as "Safe State".

problem:-

Apply Banker's Algorithm:-

a) Need matrix

b) $\langle P_1\ P_3\ P_4\ P_2\ P_0 \rangle$ is safe sequence/not

c) Request from P_1 for (1,0,2) is available immediately/not?

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			

i) Need-matrix:-

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

ii) Safe seque

P_1

ii) Safe sequence;-

$\langle P_1\ P_3\ P_4\ P_2\ P_0 \rangle$

P_1

	A	B	C
max	3	2	2
Initial	2	0	0
need	1	2	2

Need(P_1) => (1, 2 2) = request

Available(3, 3, 2)

Request ≤ Available

Satisfied.

P_3

	A	B	C
max	2	2	1
Initial	2	1	1
need	0	1	1

= request

Available(5 3 2)

Need (0 1 1) ≤ (5 3 2)

Satisfied

Available(7 4 3)

P_4

	A	B	C
max	4	3	3
initial	0	0	2
need	4	3	1

(request)

Request(4, 3, 1) ≤ Available (7 4 3)

Satisfied

Available (7 4 5).

8/12/21

P_2

	A	B	C
max :	9	0	2
initial :	3	0	2
need :	6	0	0 (Request)

Request (6 0 0) ≤ Available (7 4 5)

Satisfied. Available (10 4 7)

P_0

max :	7	5	3
initial :	0	1	0
need :	7	4	3 (Request)

Request (7 4 3) ≤ (10 4 7)

Satisfied.

<P_1 P_3 P_4 P_2 P_0> is safe resource.

iii)

P_1 ; need (1 2 2)

Request (1 0 2)

Request (1 0 2) ≤ Need (1 2 2)

Request (1 0 2) ≤ Available (3 3 2)

Satisfied. OS will allocate the resource immediately.

8/12/21

Dead lock detection & Recovery:-

Deadlock

=) Detection:-

2 Algorithms available for deadlock detection,

a) Single instance resource Algorithm

b) multiple instance resource Algorithm

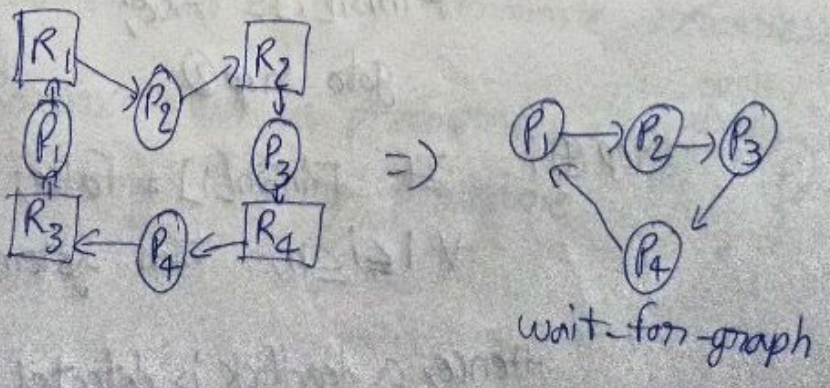
a) Single instance resource Algorithm:-

It works when there is only 1 instance of a resource available we take a resource Allocation graph & converted into wait for graph.

wait for graph:-

Resources in the graph once removed, then new graph contains only processes. A graph which contain only (processes) is called wait-for-graph.

If a wait-for-graph consists a cycle then deadlock is detected.



$\rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$

If contains cycle, then deadlock identified.

b) multiple instance Resource Algorithm:-

wait-for-graph Algorithm is applicable when only 1 instance of resource is available.

There are 4 data structures available,

a) Available[j] = k
b) max[i,j] = k
c) Allocated[i,j] = k
d) need[i,j] = k

Step ①: work = Available
Finish[i] = false; for all 'i'

Step ②: Find an 'i' such that
Finish[i] = false;
Request ≤ work;
If no such i exists then goto Step ④

Step ③: work = work + Allocation
Finish[i] = true;
goto Step ②

Step ④: If Finish[i] = false;
$\forall 1 \leq i \leq n$, then System is in unsafe state.

Hence, a deadlock is detected.

Deadlock

a)

Deadlock Recovery:-

2 ways to recover from deadlock, once deadlock is detected in the System.

a) process Termination

b) Resource preemption

a) process Termination:- a) Terminate all processes which cause dead lock occurrance. Disadvantage of this technique is the execution done so far is wasted.

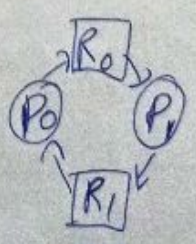
b) Terminate one process at a time till deadlock is removed from the System.

we start deleting 1 process & Checked that deadlock is removed. If deadlock is removed, we stop the process. otherwise, we start terminating another process and so on.

b) Resource preemption:-

Instead of terminating processes, Sometimes we may terminate resources.

Resources can be preempted using following procedure



a) Identifying a resource;-

out of all resources available 1 resource will be identified & preempted.

b) roll back;-

once the identified resource (by step i) is preempted then the process which was holding the resource will goto waiting

c) No starvation;-

If Each time, the same resource is terminated continuously then process will goto waiting state for long time. This causes starvation situation, so each time different resource should be terminated.