Operating System (5th semester)

Prepared by SANJIT KUMAR BARIK (ASST PROF, CSE)

MODULE-II

TEXT BOOK:

1. Operating System Concepts – Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, 8th edition, Wiley-India, 2009.
2. Mordern Operating Systems – Andrew S. Tanenbaum, 3rd Edition, PHI
3. Operating Systems: A Spiral Approach – Elmasri, Carrick, Levine, TMH Edition.

DISCLAIMER:

"THIS DOCUMENT DOES NOT CLAIM ANY ORIGINALITY AND CANNOT BE USED AS A SUBSTITUTE FOR PRESCRIBED TEXTBOOKS. THE INFORMATION PRESENTED HERE IS MERELY A COLLECTION FROM DIFFERENT REFERENCE BOOKS AND INTERNET CONTENTS. THE OWNERSHIP OF THE INFORMATION LIES WITH THE RESPECTIVE AUTHORS OR INSTITUTIONS."

Process Synchronization:

Process synchronization means sharing system resources by process in a such 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 co-operating processes.

Critical section problem

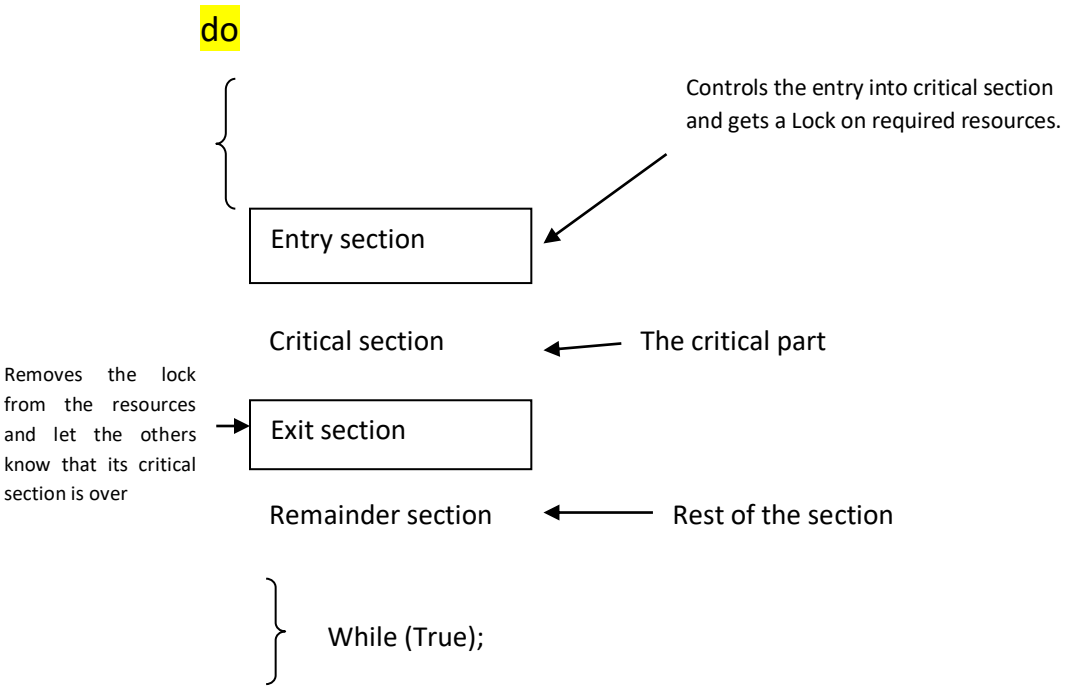
A critical section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of co-operating processes, at a given point of time, only one process must be executing its critical section. If other processes also want to execute its critical section,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

Critical section:

It is the part of the program where shared resources are accessed by various processes.

It is the place where shred variable, resources are placed.

General structure of a typical process P_i



Solution to Critical section Problem

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

1. Mutual exclusion
2. Progress
3. Bounded waiting
4. No assumption related to H/W speed

1. Mutual exclusion

Out of a group of co-operating processes, only one process can be in its critical section at a given point of time.

2. Progress:

If no process is in its critical section and if one or more process wants to execute in critical section than one of these process 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 time is reached, system must grant the process permission to get into its critical section.

4. No assumption related to H/W speed

Synchronization H/W:

Many systems provide H/W support for critical section code. The critical section problem could be solved easily in a single –processor environment if we could disallow interrupts to occur while a shred variable or resources is being modified.

In this manner, we could be sure that the current sequence of instruction would be allowed to execute in order without preemption.

- Unfortunately, this solution is not feasible in a multiprocessor Environment.
- Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

Mutex Locks

As the synchronization H/W solution is not easy to implement for everyone, a strict S/w approach called Mutex Locks was introduced. In this approach, in the ***entry section*** code, a Lock is required over critical resources modified and used inside ***critical section*** and in the ***exit section*** that Lock is released.

As the resources are locked while a process executes its ***critical section*** hence no other process can access it.

Two process solution:

This algorithm is restricted only for two (P0, P1) process. Processes may share some common variables to synchronize their actions.

The process are numbered P0 and P1 . In general if one process is P_i then other one is P_j where j=1-i.

Algorithm-1

Share variable :int *turn*

The value of turn either 0 or 1

Initially , *turn* is set to 0(*turn*=0)

If *turn* ==i, then P_i can enter it critical section.

The structure of process P_i

```
while (True)
{
while(turn != i );
critical section.
turn=j
remainder section
}
```

Explanation:

P0	P1

While(1)
{
While(turn!=0);
Critical section
Turn=1;
Remainder section
}
	while(1)
	{ while(turn!=1);
	critical section
	Turn=0;
	
	Remainder section
	}

Turn=0

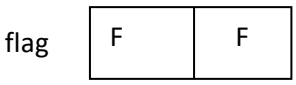
- It satisfies the mutual exclusion but not progress because it always depends on other process.
- Mutual exclusion is preserved
- The progress requirement is not satisfied.

Algorithm2

Shared variables
Boolean falg[2];
Initially flag[0]=flag[1]=false;
If flag[i]=true P_i is ready to enter its critical section .

while (true)
{
flag [i]=true.
while (flag[i]);
critical section.
flag[i]=false;
Remaider section.
}

Explanation:



P0

```
while(1)
{
flag[0]=T
while(flag[1]);
critical section
 flag[0]=F

}
```

P1

```
while(1)
{   flag[1]=T

        while(flag[0]);
        critical section
      flag[0]=F

        }
```

In this algorithm:

1. Mutual exclusion is preserved
2. The progress requirement is not satisfied
   (Since flag [0] =true and flag [1] =true; p0 and p1 are looping forever in their respective *while* statements

Algorithm3(Peterson's Solution)

Shared variables
by combining the key ideas of algorithm1 and 2 .
Boolean flag[2];
int turn

Structure of P_i

```
while (True)
{
flag[i]=true;
Turn =j;
while ((turn==j && flag[j]==T);
critical section;
flag[i]=false;
Remainder section;
}
```

Explanation:

P0

```
while(1)
{
flag[0]=T
turn=1;
while (turn==1 and
flag[1]==T);
critical section
flag[0]=F
}
```

P1

```
while(1)
{
flag[1]=T
turn=0;
while (turn==0 and
flag[0]==T);
critical section
flag[1]=F
}
```

Set	0	1
flag	F	F

Turn =0 / 1

1. Mutual exclusion is preserved
2. The progress requirement is satisfied
3. The bounded –waiting requirement is met.

Semaphore

Dijkstra proposed the concept of semaphore in 1965. Semaphore provides general purpose solution to impose mutual exclusion among concurrently executing processes, where many processes want to execute in their critical section but only one at a time is allowed and rest all other are excluded.

A semaphore basically consists of an integer variable S, shared by processes.

S is a protected variable that can only be accessed and manipulate by two operation:- wait() and signal() originally defined P(for wait) and V(for signal) by Dijkstra.

- Wait() and signal() are semaphore primitives
- The wait is sometimes called *down()* and signal is called *up()*.
- Each semaphore has a queue associated with it known as semaphore queue
- The wait and signal primitives ensures that one process at a time enters in its critical section and rest all other processes wanting to in their critical sections are kept waiting in the *semaphore queue.*

  A semaphore S is an integer variables that, apart from initialization, is accessed only through two standard atomic operations i.e wait () and signal ().
  Here wait means to test and signal means to increment
  The classical definition of *wait()* is :

```
wait()

{
while (S<=0);
//busy wait
S=S-1;
}
```

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.

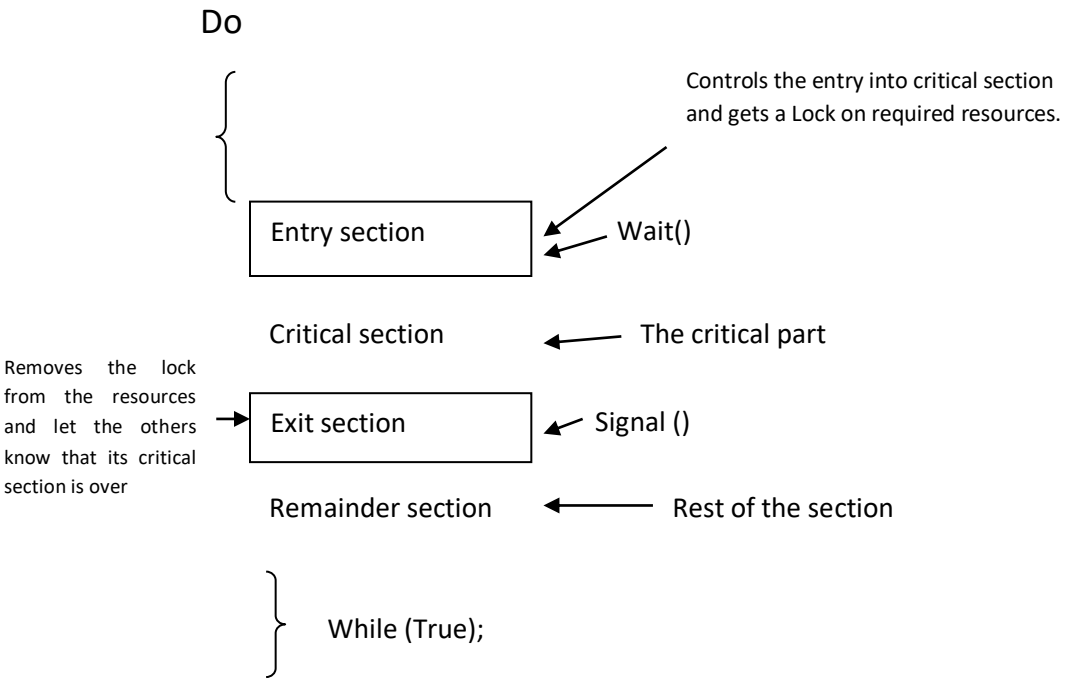
The classical definition of *signal()* is:

```
signal()
{
S=S+1;
}
```

All modification to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly: i.e when one process modifies the semaphore value no other process can simultaneously modify that same semaphore value.

In addition, in the case of wait(S), the testing of the integer value of S i.e S<=0, as well as its possible modification(S--), must be executed without interruption.

General structure of a typical process P_i



Properties of semaphore:

1. It is simple and always have a non-negative integer value
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into critical section at once, if desirable.
6. Solution to critical section
7. Act as resource management
8. It also decide the order of execution among the process(n-process)

Usage of semaphore

1. Counting semaphore
2. Binary semaphore

1. *Counting semaphore:*

- The value of counting semaphore can range over an unrestricted domain(-∞ to ∞)
- Counting semaphores can be used to control access to a given resource consisting of finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each processes that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count)
- When a process releases a resource, it performs a signal () operation (incrementing the count).
- When a count for the semaphore goes to all '0',all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than ' 0'.

2. *Binary semaphore:*

This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

Implementation semaphore(counting semaphore)

Semaphore is defined as:

```
typedef struct
{
int count;
struct  processQueue queue;
}semaphore;
```

The wait is defined as:

```
wait(semaphore S)
{
S.count--;
if(S.count<0)
{
/*perform block operation and move the process to the semaphore queue*
or put process(PCB) in suspended list/
 sleep() or block ();
}
}
```

The signal is defined as:

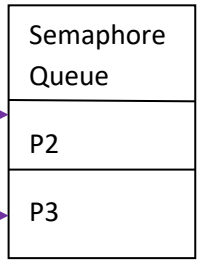
```
Signal(S)
{
S.count++;
If(S.count<=0)
{
/* semaphore queue is not empty, perform wakeup and move the first
process from semaphores queue to the ready queue or remove a process
(P) from suspended list*/
wakeup(P);
}
}
```

```
while (true)
{
//entry section
Wait(S);
<Critical section>
//exit section
Signal(S);
}
```

Initially the value of semaphore variables *S.count*=1;

Let P1,P2,P3

S.count	For P1			
	Wait(S)		Signal(S)	
	S.count--	If(s.count<0) move the process to the semaphore queue	S.count++	*If(S.count<=0)* *Move the first process from semaphore queue to the ready queue*
1	0	false	-	-
So P1 enters in its critical section and value of S.count=0 Mean while P2 also enters it its critical section				
For P2				
0	Wait(S)		Signal(S)	
	s.count--	If(s.count<0) move the process to the semaphore queue	S.count++	*If(S.count<=0)* *Move the first process from semaphore queue to the ready queue*
	-1	true		
Mean while P3 also enters it its critical section				
	For P1			
	Wait(S)		Signal(S)	
	s.count--	If(S.count<0) move the process to the semaphore queue	S.count++	*If(S.count<=0)* *Move the first process from semaphore queue to the ready queue*
-1	-2	true		
Mean while P1 finish its execution in its critical section and is in exit section				



	For P1			
	Wait(S)		Signal(S)	
	S.count--	If(s.count<0) move the process to the semaphore queue	S.count++	*If(S.count<=0)* *Move the first process from semaphore queue to the ready queue*
-2	-	-	-1	True, so the moves first process P2 from semaphore queue to the ready queue

S.count	Mean while P2 finish its execution in its critical section and is in exit section For P2			
	Wait(S)		Signal(S)	
	S.count--	If(S.count<0) move the process to the semaphore queue	S.count++	*If(S.count<=0)* *Move the first process from semaphore queue to the ready queue*
-1	-	-	0	True, so the moves first process P3 from semaphore queue to the ready queue
So P3 enters into its Critical section and value of S.count=0				
NOW P3 finish its execution in its critical section and is in exit section For P3				
S.count	For P3			
	Wait(S)		Signal(S)	
	S.count--	If(s.count<0) move the process to the semaphore queue	S.count++	*If(S.count<=0)* *Move the first process from semaphore queue to the ready queue*
0	-	-	1	-
	When all the process in the semaphore queue are finished, means semaphore queue are empty, the semaphore variable S.count again return to its initial value 1			

- The concept of semaphore queues ensures that no process go into busy waiting. Busy waiting is a condition in which if one process is executing in its critical section any other process wants to enter in its critical section then that process needs to check some condition in its entry section in continuous loop. This continuous looping is wastage of CPU cycle in a multiprogramming system where that CPU cycle can be used for some other productive work. The process waiting to execute in its critical section is moved to the *semaphore queue* till it get a chance to enter in its critical section without CPU engagement and this saves lot of CPU Time.

- Moving the process to the semaphore queue is called *block() operation* and changes the state of that process from running state to waiting state. Likewise removing the process from semaphore queue and placing it in the ready queue is called *wakeup().* The *wakeup() operation* resumes the process from waiting state to ready state so that process can enter in its critical section. Both *block* and *wakeup()* operations are performed by the operating system as a basic system call.

Q.*A Counting Semaphore was initialized to 12. then 10P (wait) and 4V (Signal) operations were computed on this semaphore. What is the result?*

Ans: S = 12 (initial)
10 p (wait) :
SS = S -10 = 12 - 10 = 2
then 4 V :
SS = S + 4 =2 + 4 = 6

Hence, the final value

Binary Semaphore Implementation:

```
Down ( semaphore S)
        {
        if(S.value==1)
        {
        S.value=0;
        }
        else
        {
                    /*perform block operation and move the process to the
                    semaphore queue* or put process(PCB) in suspended list/
                     Sleep() or block ();

            }}
```

Let S.vaue=1

P1	P2
Down(s)	Down(s)
CS	CS
Up(S)	Up(S)

```
Up(semaphore S)
        {
        if (semaphore queue is empty)
        {
        S.value=1;
        }
        else
            {
        /* semaphore queue is not empty, perform wakeup( ) and move the first
process from semaphores queue to the ready queue or remove a process (P) from
suspended list*/
              wakeup(P);
                        }
                              }
```

Classical problem of Synchronization:
OR
Classical problem in Concurrency

- Reader –Writers problem
- Dining Philosopher Problem(Assignment)
- Sleeping Barber Problem(Assignment)

Reader –Writers problem

Definition: There is a data containing some files ,records etc that is shared among the number of concurrent processes. The processes that reads the data from that common shared data area are called ***reader processes*** and processes that perform write operation(writing new data value or updating or modifying the data value) on the data stored in common shared data area are called ***writer processes***. The various conditions that need to take care in Reader-writer case are:

- Any number of reader processes can simultaneously read the data from common shared data area but only one writer at a time may write to that common shared data area.
- If any of the writer process is writing to common shared data area, then no reader processes are allowed to read it till the writer process has finished.
- If there is at least one reader reading the common data area,no writer processes are allowed to that common data area.

The reader-writer problem solution using semaphores consists of two binary-semaphores- ***mutex*** and ***rw_mutex*** and one integer variable ***NumberOfReaders(rc)***.The semaphore ***rw_mutex*** is shared by the all the processes and the semaphore ***mutex*** and the integer variable ***NumberOfReaders(rc)*** is shared by reader processes only. Here, variable ***NumberOfReaders(rc)*** keep track of how many reader processes are reading the common shared data at a time, and ***mutex*** provide mutual exclusion among reader processes when variable ***NumberOfReaders(rc)*** is incremented or decremented .The semaphore ***rw_mutex*** which is common to both readers and writers processes ensures that when one writer process is using the common data area, no other reader or writer processes can access that common data area.

```
int rc=0
Semaphore mutex =1;
Semaphore rw_mutex=1;
```

READER PROCESS

```
void Reader ( )
{
while (true)
{
Down (mutex)
rc=rc+1;
if(rc==1) Down (rw_mutex);
 UP(mutex);

 //Critical section
 //Database

Down(mutex);
rc=rc-1;
if (rc==0) then UP(rw_mutex);
UP(mutex);
Process_data;
}
}
```

WRITER PROCESS

```
void Writer()
{
while true()
{
Down (rw_mutex);

 //Critical section
 //Database

UP(rw_mutex);
}
}
```

Case 1: R-W->Problem

Case 2: W-R->Problem

Case 3: W-W->Problem

Case4: R-R->No Problem

Dead Lock

- In a multiprogramming system, a number process compete for limited no.of resources and if a resources is not available at that instance then process enters into waiting states
- If a process unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process, then system is said to be in deadlock

System Model

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: The process requests the resource .If the request cannot be granted immediately (Ex: if resources is being used by another process), then the requesting process must wait until it can acquire the resource.
2. Use: The process can operate on the resource (Ex: if the resource is a printer , the process can print on printer)
3. Release: The process releases the resource

The request and release of resources may be system calls

EX: request() and release() device

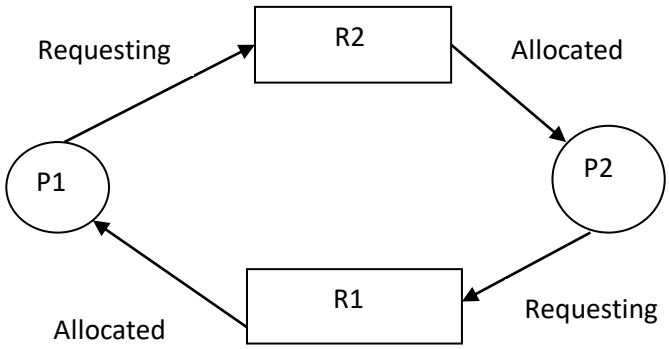
Open () and close () file

Allocate () and free () memory.

The request and release of semaphore (system resource) can be accomplished through wait () and signal () operations.

Dead lock:

A set of process is in a deadlocked state when every process in the set is waiting for an event can be caused only by another process in the set.



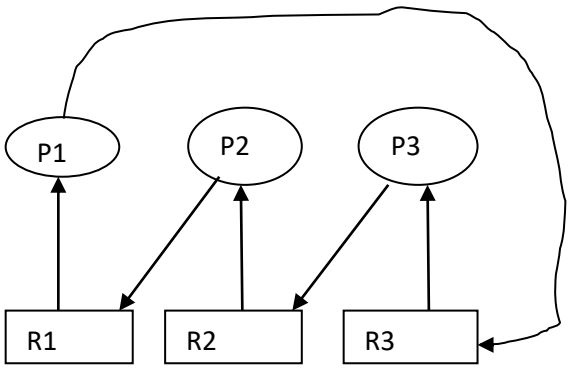
P1->R1->P2->R2->P3->R3->P4……..P_{n-1} ->R_{n-1} ->P_n ->R1

Necessary conditions for Deadlock:

A deadlock situation can arise if the following four conditions hold simultaneously in a system.

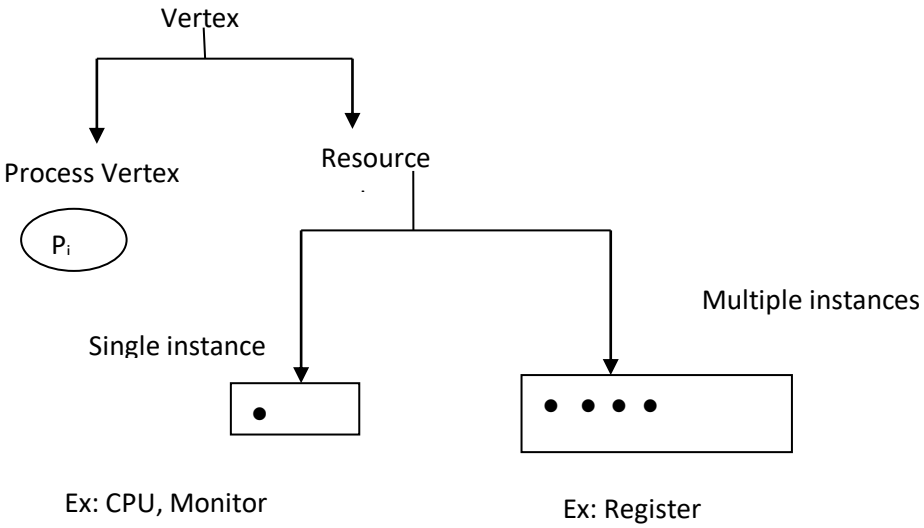
1. Mutual exclusion: At least one resource must be held in a non shareable mode i.e only one process at a time can use the resource .If another process request that resource ,the requesting process must be delayed until the resource has been released.
2. Hold and wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.
3. No preemption: Resources cannot be preempted i.e a resource can be released only voluntarily by process holding it after that process has completed its task.
4. Circular Wait: A set of process {p0……..Pn} of waiting processes must exist such that P0 is waiting for a resources held by P1, P1 is waiting for a resources held by P2……,Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by P0.

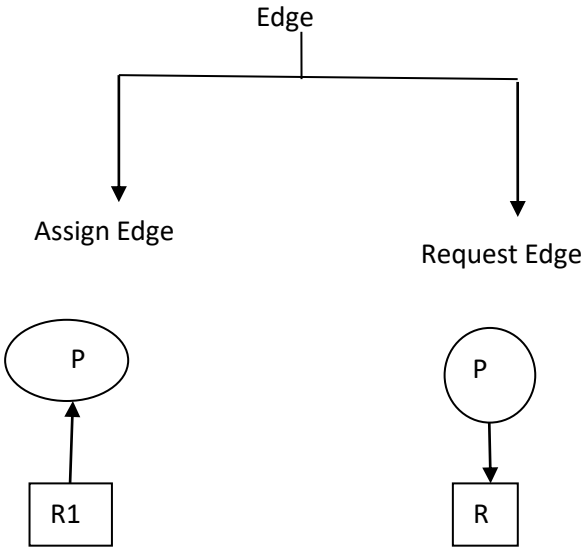
Circular Wait



Resource –allocation Graph (RAG) or System Resource Allocation Graph

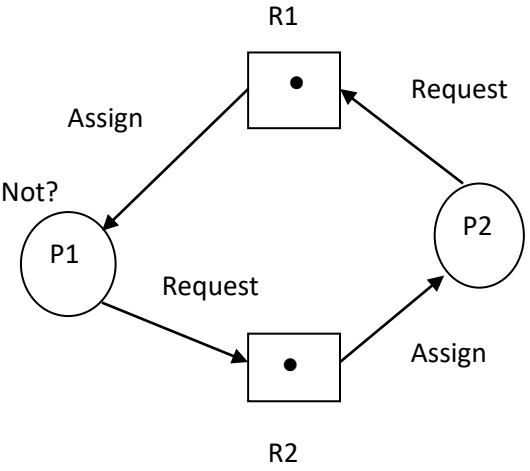
It describes the state of the system (dead lock or not) more precisely





Question:

Check whether the system,Dead lock or Not?



Ans:

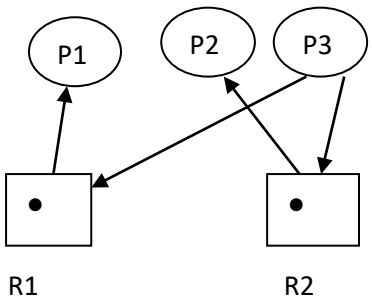
Process	Allocate		Request	
	R1	R2	R1	R2
p1	1	0	0	1
p2	0	1	1	0

Cyclic graph (circular wait)

Availbility=(0,0) can you fulfill the request of P1 and P2(NO), it is deadlock

Question:

Check dead lock or not!



Process	Allocate		Request	
	R1	R2	R1	R2
p1	1	0	0	0
p2	0	1	0	0
P3	0	0	1	1

Acyclic graph (no circular wait)

R1 R2

Availability (0 , 0)

1	0	p1
1	0	
0	1	p2
1	1	P3

No dead lock

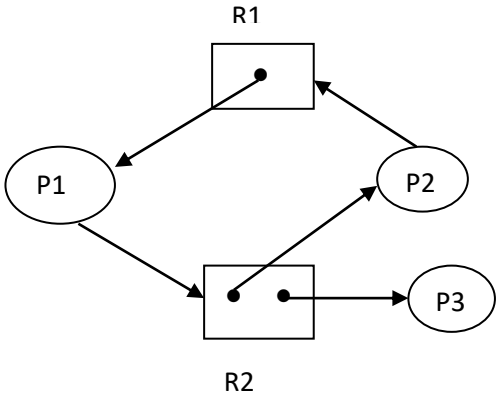
P1 p2 p3

Note: 1. If there is single instance and RAG has circular wait then there is deadlock (True)

2. If RAG has no cycle, then no dead lock occur (True)

3. If there is multiple instances and RAG has circular wait then there may or may not be deadlock

Multiple instances:



Solution:

Process	Allocate		Request	
	R1	**R2**	**R1**	**R2**
p1	**1**	**0**	**0**	**1**
p2	**0**	**1**	1	0
P3	0	1	0	0

Availability(0 , 0)

P3 0 1

0 1

P1 1 0

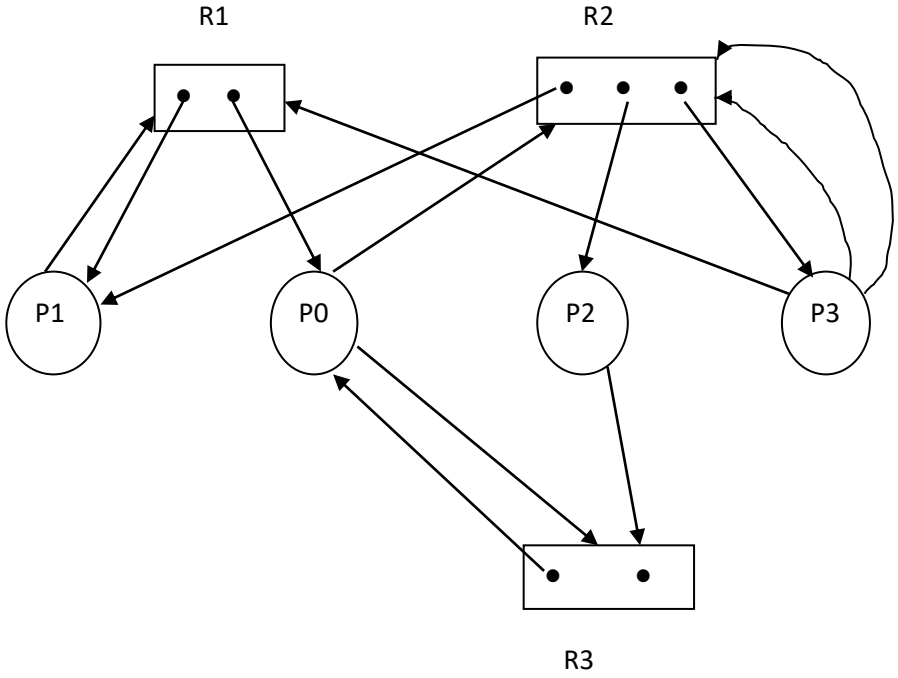
1 1

P2 0 1

1 2

No deadlock

Q.



Solution:

Process	Allocate			Request		
	R1	R2	R3	R1	R2	R3
P0	1	0	1	0	1	1
P1	1	1	0	1	0	0
P2	0	1	0	0	0	1
P3	0	1	0	1	2	0

Current Availability (0, 0, 1)

P2 0, 1, 0

0, 1, 1

P0 1, 0, 1

1 1 2

P1 1 1 0

2 2 2

P3 0 1 0

2 3 2

No dead lock

Banker's algorithm(Avoidance Algorithm or Safety 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.

Why Banker's algorithm is name do so?

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

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.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

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

Allocation :

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

Need :

- It is a 2-d array of size **'nxm'** that indicates the remaining resource need of each process.
- Need [i] [j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$ for its execution.
- Need [i][j] = Max [i][j] – 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

ALGORITHM:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
Initialize: Work = Available
Finish[i] = false; for i=0, 1, 2, 3....n-1;

2) Find an index i such that both
a) Finish[i] = false
b) $Need_i$ <= 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

Where m= number of resource types

n=number of process in the system

Resource-Request Algorithm

This algorithm describe whether requests can be safely granted or not!.

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 $Request_i$ <= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= 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 Pi by modifying the state as
follows:
Available = Available – Requesti
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$– $Request_i$

Example:

Considering a system with five processes P_0 through P_4 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 t_0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

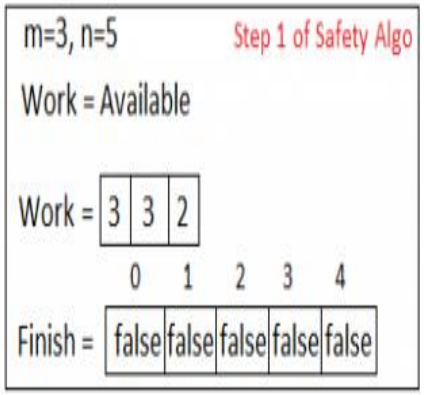
Question1. What will be the content of the Need matrix?

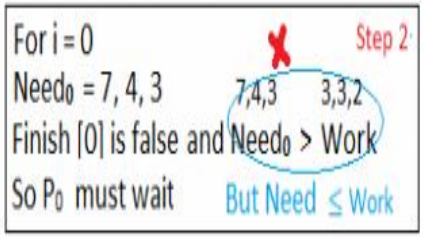
Need [i, j] = Max [i, j] – Allocation [i, j]

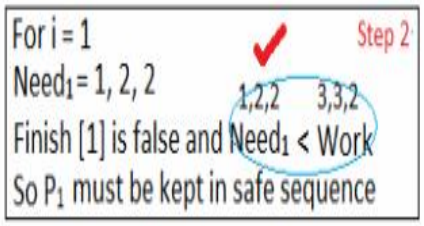
So, the content of Need Matrix is:

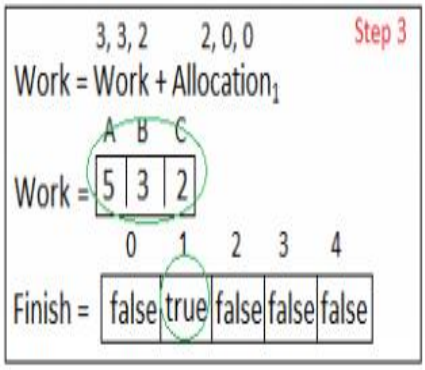
Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

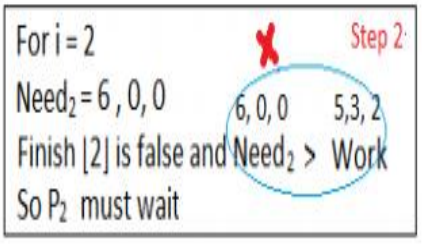
Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

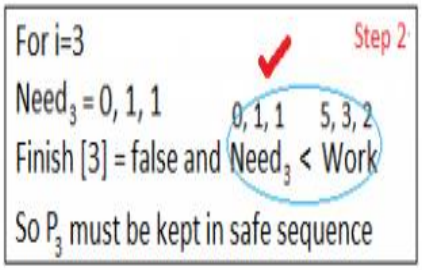


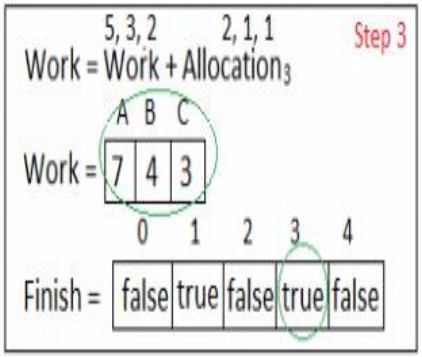


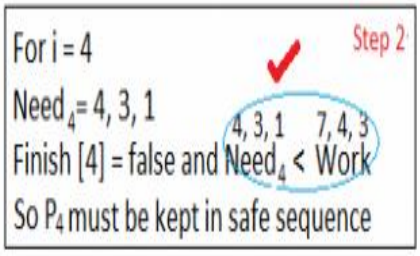


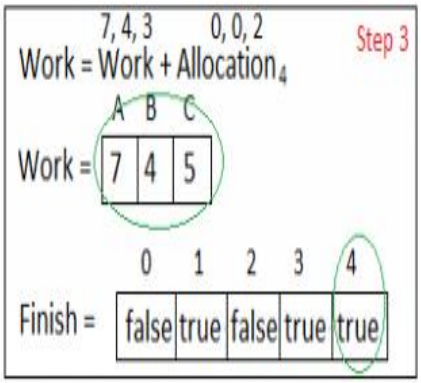


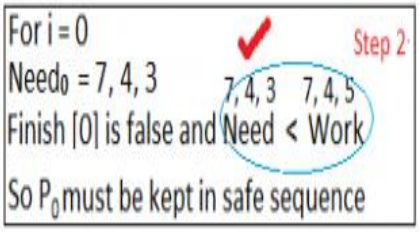


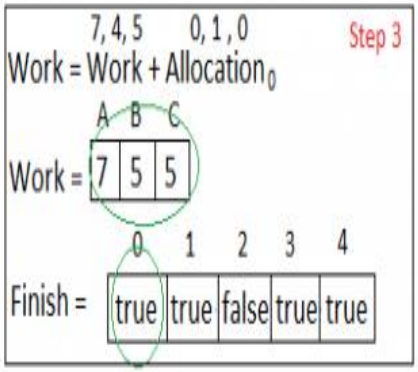


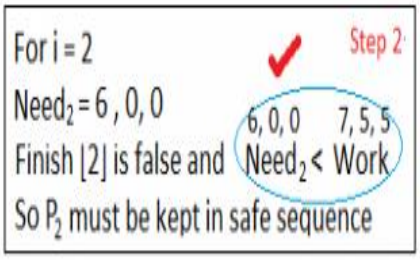


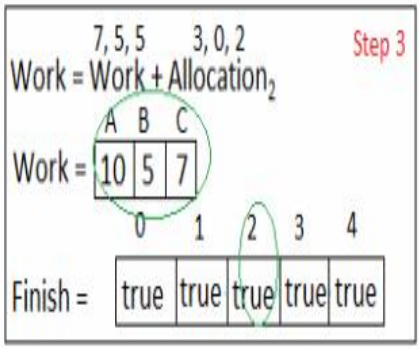


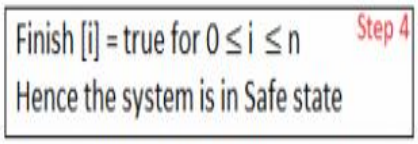












The safe sequence is P_1, P_3, P_4, P_0, P_2

Ex2:

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

- Calculate matrix *need?*
- *Is the system is in a safe state?*
- *If p1 arrives with request (0,3,2,0) can it be granted immediately ?*

Solution: Available[m] n=#process=5 = p0 to p4

Allocation[nxm] m=#resources=4= A,B,C,D

Max [nxm]

Need[nxm]= max-allocation

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

Work=available

A	B	C	D	
1	5	2	0	
0	0	1	2	p0
1	5	3	2	
1	3	5	4	p2
2	8	8	6	
0	6	3	2	p3
2	14	11	8	
0	0	1	4	p4
2	14	12	12	
1	0	0	0	p1
3	14	12	12	

index	Finish
0	~~F~~ T
1	~~F~~ T
2	~~F~~ T
3	~~F~~ T
4	~~F~~ T

safe sequence is<p0,p2,p3,p4,p1>

If p1 arrives with request (0,3,2,0) can it be granted immediately: YES