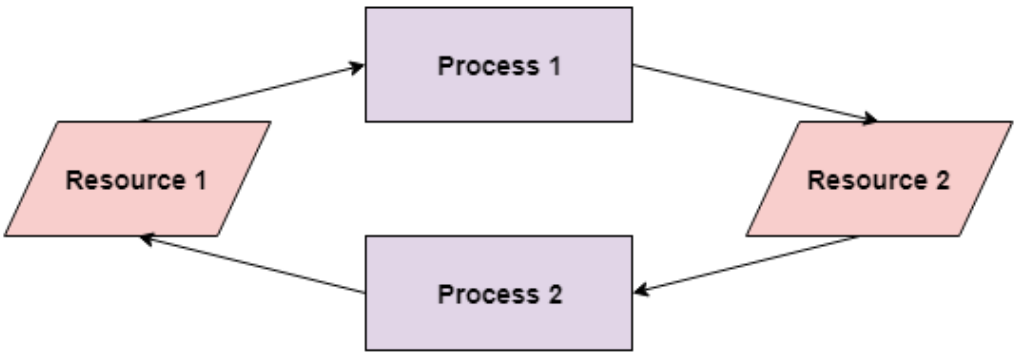
Student Name: NITESH PRASAD(57)
Student ID:11915597
Email Address: niteshprasad809gmail.com
GitHub Link: https://github.com/nitesh1016/os
Code: Question no.11
Subject: Operating System

1.Explain the problem in terms of operating system concept?
Description:

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



Deadlock in Operating System

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

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)

Hold and Wait: A process is holding at least one resource and waiting for resources.

No Pre-emption: A resource cannot be taken from a process unless the process releases the resource.

Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock

Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods:

- All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
- Resources can be preempted from some processes and given to others till the deadlock is resolved.

Deadlock detection and recovery:
Let deadlock occur, then do preemption to handle it once occurred.

Ignorance:
Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

2. Description:

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

Following Data structures are used to implement the Banker's Algorithm:
Let 'n' be the number of processes in the system and 'm' be the number of resources types.

Available:

• It is a 1-d array of size 'm' indicating the number of available resources of each type.
• Available[j] = k means there are 'k' instances of resource type Rj

Max:

• It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
• Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.

Allocation:

• It is a 2-d array of size 'n*m' that defines the number of resources of each type currently a allocated to each process.
• Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resource type Rj

Need:

• It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.
• Need [i, j] = k means process Pi currently allocated 'k' instances of resource type Rj
• Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

3.Algorithm:

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i[j] = k$ means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If Request <= Need
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 – Request
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i$– Request

4.Purpose of use:

Bankers algorithm is used to find out the safe sequence of execution of processes which when followed prevent system to be in deadlock condition.

It is a very sophisticated process and is very hard to implement so most of the OS like windows does not implement this and choose the way that the system will restart if ever in future deadlock occur because the chances of deadlock are assumed to be very few so instead of implementing this time consuming process we restart the system.

Banker's algorithm works in a similar way in computers. Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

5.Code Snippet

```
//code for the  given question..

#include<conio.h>
#include<stdio.h>


int current_s[5][5], maximum_claimed[5][5], available[5];
int allocation[5] = {0, 0, 0, 0, 0};
int max_res[5], running[5], safe = 0;
```

```
int i,j,counter=0,k=1,exec,resources,processes;
int main()
{
printf("\nEnter number of processes: ");
scanf("%d", &processes);
for (i = 0; i < processes; i++)
{
running[i] = 1;
counter++;
}
printf("\nEnter number of resources: ");
scanf("%d", &resources);
printf("\nEnter Claim Vector:");
for (i = 0; i < resources; i++)
{
scanf("%d", &max_res[i]);
}
 printf("\nEnter Allocated Resource Table:\n");
 for (i = 0; i < processes; i++)
{
 for(j = 0; j < resources; j++)
{
scanf("%d", &current_s[i][j]);
}
}
printf("\nEnter Maximum Claim Table:\n");
for (i = 0; i < processes; i++)
{
for(j = 0; j < resources; j++)
{
scanf("%d", &maximum_claimed[i][j]);
}
}
printf("\nThe Claim Vector is: ");
for (i = 0; i < resources; i++)
{
 printf("\t%d", max_res[i]);
}
printf("\nThe allocated resource table:\n");
 for (i = 0; i < processes; i++)
{
for (j = 0; j < resources; j++)
{
printf("\t%d", current_s[i][j]);
}
printf("\n");
```

```
}
printf("\nThe maximum claim table:\n");
for (i = 0; i < processes; i++)
{
for (j = 0; j < resources; j++)
{
printf("\t%d", maximum_claimed[i][j]);
}
printf("\n");
}
for (i = 0; i < processes; i++)
{
for (j = 0; j < resources; j++)
{
allocation[j] += current_s[i][j];
}
}
printf("\nAllocated resources:");
for (i = 0; i < resources; i++)
{
printf("\t%d", allocation[i]);
}
for (i = 0; i < resources; i++)
{
available[i] = max_res[i] - allocation[i];
}
printf("\nAvailable resources:");
for (i = 0; i < resources; i++)
{
printf("\t%d", available[i]);
}
printf("\n");
while (counter != 0)
{
safe = 0;
for (i = 0; i < processes; i++)
{
if (running[i])
{
exec = 1;
for (j = 0; j < resources; j++)
{
if (maximum_claimed[i][j] - current_s[i][j] > available[j])
{
exec = 0;
break;
```

```
}
}
if (exec)
{
printf("\nProcess%d is executing\n", i + 1);
running[i] = 0;
counter--;
safe = 1;
for (j = 0; j < resources; j++)
{
available[j] += current_s[i][j];
}
break;
}
}
}
if (!safe)
{
printf("\nThe processes are in unsafe state.\n");
break;
}
else
{
printf("\nThe process is in safe state");
printf("\nAvailable vector:");
for (i = 0; i < resources; i++)
{
printf("\t%d", available[i]);
}
printf("\n");
}
}
return 0;
}
```

Output:-

The processes are in unsafe state.

6. Description (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	

Time complexity = O(n*n*m) where n = number of processes and m = number of resources.

Yes ,I have made a total of 5 contribution on GitHub.

GitHub Link: https://github.com/nitesh1016/os