Name: Meet Brijwani
Roll no: 14
Batch: S11

EXP 7:

Aim : Process Management: Deadlock
a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
b. Write a program demonstrate the concept of Dining Philospher's Problem

Theory : The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the 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

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 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 needs ‘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. Banker’s algorithm consists of a Safety algorithm and a Resource request algorithm.

Banker’s Algorithm
1. Active:= Running U Blocked;

for k=1...r

New_ request[k]:= Requested_ resources[requesting_ process, k];

2. Simulated_ allocation:= Allocated_ resources; for k=1.....r

//Compute projected allocation state

Simulated_ allocation [requesting _process, k]:= Simulated_ allocation [requesting _process, k] + New_ request[k]; 3. feasible:= true; for

k=1....r // Check whether projected allocation state is feasible if Total_

resources[k]< Simulated_ total_ alloc [k] then feasible:= false;

4. if feasible= true
then // Check whether projected allocation state is a safe allocation state

while set Active contains a process P1 such that

For all k, Total _resources[k] – Simulated_ total_ alloc[k]>= Max_ need [l

,k]-Simulated_
allocation[l, k]

Delete Pl from Active;

for k=1.....r
Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];

5. If set Active is empty then // Projected allocation state is a safe

 allocation state for k=1....r // Delete the request from pending

 requests Requested_ resources[requesting_ process, k]:=0; for

 k=1....r // Grant the request

Allocated_ resources[requesting_ process, k]:= Allocated_ resources[requesting_ process, k] + New_ request[k];

Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];

Safety Algorithm
The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4....n
2) Find an i such that both a)
Finish[i] = false
b) Needi <= 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 Safety Algorithm

Resource-Request Algorithm

Let Requesti be the request array for process Pi. Requesti [j] = k means process Pi wants k

instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1) If Requesti <= Needi

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

2) If Requesti <= Available

Goto step (3); otherwise, Pi 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

Allocationi = Allocationi + Requesti

Needi = Needi– Requesti

Code :

```
// Banker's Algorithm
#include <iostream>

using namespace std;
```

```
int main()

{

int n, m, i, j, k;

n = 5;

m = 3;
```

```
int alloc[5][3] = { { 0, 1, 0 },




{ 2, 0, 0 },


{ 3, 0, 2 },

{ 2, 1, 1 },


{ 0, 0, 2 } };





int max[5][3] = { { 7, 5, 3 },




{ 3, 2, 2 },


{ 9, 0, 2 },

{ 2, 2, 2 },


{ 4, 3, 3 } };





int avail[3] = { 3, 3, 2 };





int f[n], ans[n], ind = 0;


for (k = 0; k < n; k++) {
f[k] = 0;
```

```
}

int need[n][m];

for (i = 0; i < n; i++) {
```

```
for (j = 0; j < m; j++)

need[i][j] = max[i][j] - alloc[i][j];

}

int y = 0;

for (k = 0; k < 5; k++) {

for (i = 0; i < n; i++) {

if (f[i] == 0) {


int flag = 0;

for (j = 0; j < m; j++) {

if (need[i][j] > avail[j]){

flag = 1;

break;

}

}


if (flag == 0) {
ans[ind++] = i;


for (y = 0; y < m; y++)

avail[y] += alloc[i][y];

f[i] = 1;
```

```
}

}

}

}



int flag = 1;



for(int i = 0;i<n;i++)

{



if(f[i]==0)
{


flag = 0;

cout << "The given sequence is not safe";

break;

}

}




if(flag==1)
{

cout << "Following is the SAFE Sequence" << endl;

for (i = 0; i < n - 1; i++)

cout << " P" << ans[i] << " ->";

cout << " P" << ans[n - 1] <<endl;

} return
(0);


}
```

Output :

```
#include <stdio.h>
```

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2


=== Code Execution Successful ===
```

Code :

```
#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM_PHILOSOPHERS 5

#define NUM_CHOPSTICKS 5



void dine(int n);

pthread_t philosopher[NUM_PHILOSOPHERS];

pthread_mutex_t chopstick[NUM_CHOPSTICKS];


int main()
```

```
{

int i, status_message;

void *msg;

for (i = 1; i <= NUM_CHOPSTICKS; i++)

{

status_message = pthread_mutex_init(&chopstick[i], NULL);

if (status_message == -1)

{

printf("\n Mutex initialization failed");
exit(1);

}

}

for (i = 1; i <= NUM_PHILOSOPHERS; i++)
```

```
{

status_message = pthread_create(&philosopher[i], NULL, (void *)dine,
(int *)i);
```

```
if (status_message != 0)

{

printf("\n Thread creation error \n");

exit(1);

}

}


for (i = 1; i <= NUM_PHILOSOPHERS; i++)

{

status_message = pthread_join(philosopher[i], &msg);


if (status_message != 0)

{

printf("\n Thread join failed \n");

exit(1);
}


}


for (i = 1; i <= NUM_CHOPSTICKS; i++)

{

status_message = pthread_mutex_destroy(&chopstick[i]);

if (status_message != 0)

{

printf("\n Mutex Destroyed \n");

exit(1);

}
```

```c
}

return 0;

}

void dine(int n)
{

printf("\nPhilosopher % d is thinking ", n);



pthread_mutex_lock(&chopstick[n]);



pthread_mutex_lock(&chopstick[(n + 1) % NUM_CHOPSTICKS]);



printf("\nPhilosopher % d is eating ", n);


sleep(3);



pthread_mutex_unlock(&chopstick[n]);



pthread_mutex_unlock(&chopstick[(n + 1) % NUM_CHOPSTICKS]);
```

```
printf("\nPhilosopher % d Finished eating ", n);

}
```

Output :

```
Philosopher  2 is thinking
Philosopher  2 is eating
Philosopher  3 is thinking
Philosopher  5 is thinking
Philosopher  5 is eating
Philosopher  1 is thinking
Philosopher  4 is thinking
Philosopher  4 is eating
Philosopher  2 Finished eating
Philosopher  5 Finished eating
Philosopher  1 is eating
Philosopher  4 Finished eating
Philosopher  3 is eating
Philosopher  1 Finished eating
Philosopher  3 Finished eating
```

Conclusion : Thus we have successfully implemented Banker's Algorithm and the concept of Dining Philospher's Problem using C++.