WEEK 7

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

1.1.1 Code:

```
#include <stdio.h>


int main()

{

        int n, m, i, j, k;

        printf("Enter the number of processes: ");

        scanf("%d", &n);

        printf("Enter the number of resources: ");

        scanf("%d", &m);


        int allocation[n][m];

        printf("Enter the Allocation Matrix:\n");

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

        {

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

        {

        scanf("%d", &allocation[i][j]);
```

```
}
}

int max[n][m];
printf("Enter the MAX Matrix:\n");
for (i = 0; i < n; i++)
{
for (j = 0; j < m; j++)
{
scanf("%d", &max[i][j]);
}
}

int available[m];
printf("Enter the Available Resources:\n");
for (i = 0; i < m; i++)
{
scanf("%d", &available[i]);
}

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] - allocation[i][j];
}
}

int y = 0;
for (k = 0; k < n; k++)
{
for (i = 0; i < n; i++)
{
if (f[i] == 0)
{
```

```
        int flag = 0;
        for (j = 0; j < m; j++)
        {
        if (need[i][j] > available[j])
        {
        flag = 1;
        break;
        }
        }

        if (flag == 0)
        {
        ans[ind++] = i;
        for (y = 0; y < m; y++)
        {
           available[y] += allocation[i][y];
        }
        f[i] = 1;
        }
}
}
```

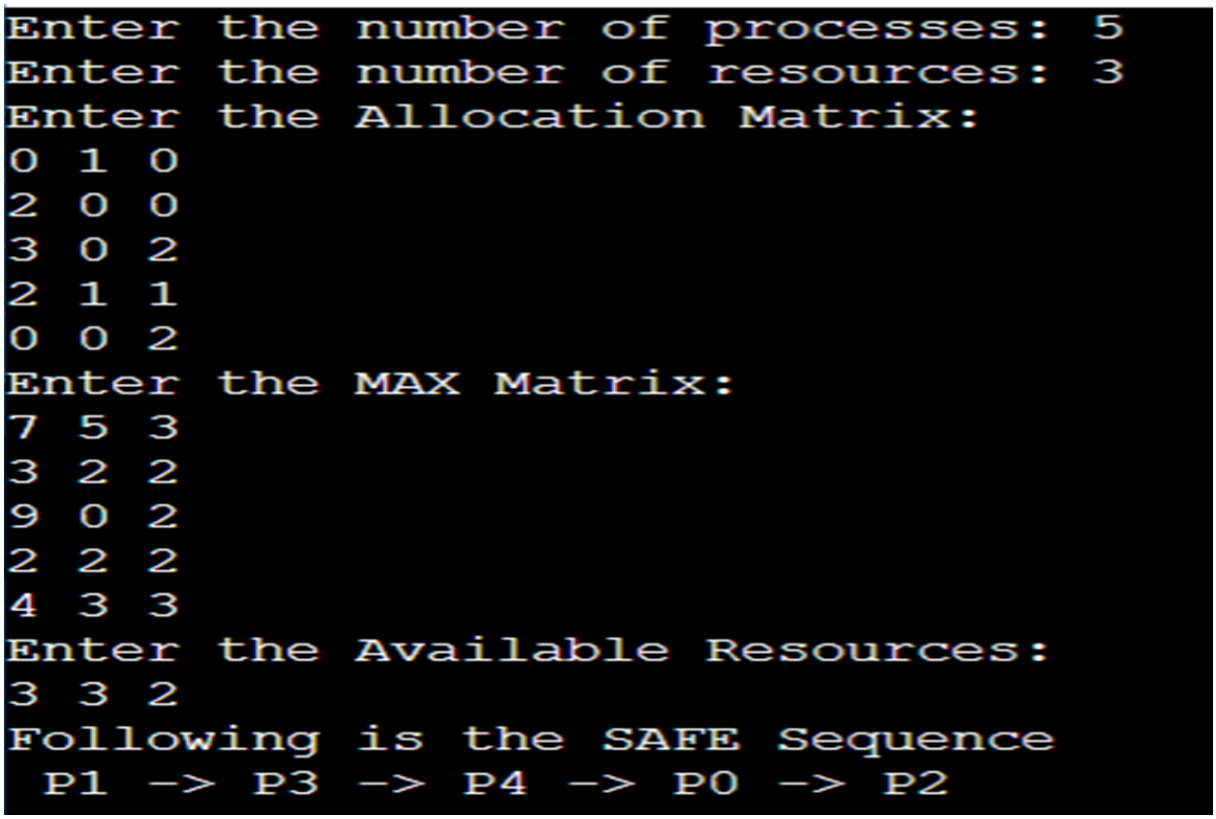
```
}

int flag = 1;
for (i = 0; i < n; i++)
{
if (f[i] == 0)
{
flag = 0;
printf("The following system is not safe\n");
break;
}
}

if (flag == 1)
{
printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
{
printf(" P%d ->", ans[i]);
}
printf(" P%d\n", ans[n - 1]);
```

```
    }
    return 0;
}
```

OUTPUT



```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the Allocation Matrix:
0 2 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the MAX Matrix:
8 4 6
3 5 7
3 6 7
9 5 3
2 5 7
Enter the Available Resources:
3 2 2
The following system is not safe
```

23/7/23 Dining Philosophers

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#define NUM_PHIL 5
#define NUM_CHOP 5

void *sine(int n);
pthread_t phil[NUM_PHIL];
pthread_mutex_t chop[NUM_CHOP];
int main() {
int i, status;
void msg;
for(i=1;i<NUM_CHOP;i++){
Status = pthread_mutex_init(&chop[i], NULL);
  if(Status==-1){
printf("Mutex failed!\n");
exit(1);
}
}
for(i=1;i<=NUMPHIL;i++){
Status = pthread_create(&phil[i], NULL, (void*)sine, (int*)i);}
if(status!=0){
printf("Thread creation error\n");
exit(1);
}
}
for(i=1;i<Num_PHIL;i++){
Status = pthread_create(&phil[i], NULL, (void*)sine(int*)i);
  if(Status!=0){
printf("Thread creation error\n");
  exit(1);
}
}
```

```
for (i = 1; i < NUM_PHIL; i++) {
  status = pthread_join (phil[i], &msg);
  if (status != 0) {
    Printf ("Thread join failed\n");
    exit (1);
  }
}
for (i = 1; i <= NUM_CHOP; i++) {
  status = pthread_mutex_destroy (&chop[i]);
  if (status != 0) {
    Printf ("Mutex destroyed\n");
    exit (1);
  }
}
return 0;
}
void *sub (int n) {
Printf ("\n Philosopher %d is thinking \n", n);
Pthread_mutex_lock (&chopstick[n]);
Pthread_mutex_lock (&chopstick[(n+1) % NUM_CHOP]);
Printf ("Philosopher %d is thinking", n);
Pthread_mutex_unlock (&chopstick[n]);
pthread_mutex_unlock (&chopstick[(n+1) % Num_CHOP]);
Printf ("\n Philosopher %d exits\n", n);
}
```

Output-

Philosopher 1 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 2 is thinking
Philosopher 5 is thinking
Philosopher 1 is eating
Philosopher 1 is finished eating
Philosopher 3 is finished eating
Philosopher 4 is eating
Philosopher 5 is eating
Philosopher 2 is eating
Philosopher 4 finished eating
Philosopher 2 finished eating
Philosopher 5 finished eating

10/10

2/8/23