U20CS093
NEHAL JHAJHARIA
OS PRACTICAL EXAMINATION

1)

Implement Banker's algorithm for Deadlock avoidance. Following data structures can be used:

Available :

It is a 1-d array of size 'm' indicating the number of available resources of each type.

Max :

It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system

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.

Need :

It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.

```
public class Bankers {
   int n = 5; // Number of processes
   int m = 3; // Number of resources
   int need[][] = new int[n][m];
   int [][]max;
   int [][]allocation;
   int []available;
   int safeSequence[] = new int[n];

   void initializeValues() {
       // Allocation Matrix
       allocation = new int[][] {
                { 0, 1, 0 }, //P0
                { 2, 0, 0 }, //P1
                { 3, 0, 2 }, //P2
                { 2, 1, 1 }, //P3
                { 0, 0, 0 } }; //P4

       // Max Matrix
       max = new int[][] {
                { 7, 5, 3 }, //P0
```

```
                { 3, 2, 2 }, //P1
                { 9, 0, 2 }, //P2
                { 2, 2, 2 }, //P3
                { 4, 4, 3 } }; //P4

        // Available Resources
        available = new int[] { 3, 3, 1 };
    }

    void isSafe() {
        // visited array to find the already allocated process
        boolean visited[] = new boolean[n];
        for (int i = 0;i < n; i++) {
            visited[i] = false;
        }

        // work array to store the copy of available resources
        int work[] = new int[m];
        for (int i = 0; i < m; i++) {
            work[i] = available[i];
        }

        int count = 0;
        while (count < n) {
            boolean flag = false;
            for (int i = 0; i < n; i++) {
                if (!visited[i]) {
                    int j = 0;
                    for (j = 0; j < m; j++) {
                        if (need[i][j] > work[j]) {
                            break;
                        }
                    }

                    if (j == m) {
                        safeSequence[count++] = i;
                        visited[i]=true;
                        flag=true;

                        for (j = 0; j < m; j++) {
                            work[j] += allocation[i][j];
                        }
                    }
                }
            }
            if (flag == false) {
                break;
            }
        }
```

```
        if (count < n) {
            System.out.println("UnSafe!!!");
        } else {
            System.out.println("Safe!");
            System.out.print("SAFE Sequence : ");
            for (int i = 0;i < n; i++) {
                System.out.print("P" + safeSequence[i]);
                if (i != n-1) {
                    System.out.print(", ");
                }
            }
        }
    }

    void calculateNeed() {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                need[i][j] = max[i][j] - allocation[i][j];
            }
        }
    }

    public static void main(String[] args) {
        Bankers m = new Bankers();

        m.initializeValues();
        m.calculateNeed();
        m.isSafe();
    }
}
```

```
Safe!
SAFE Sequence : P3, P1, P2, P4, P0
Process finished with exit code 0
```

2) Write a program to demonstrate the Producer-Consumer Process communication with bounded buffer. Also extend your program to demonstrate the Race Condition.

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

```
int m = 1; // Initialize a mutex to 1
int full_slots = 0; // Number of full slots
int empty_slots = 10; // Number of empty slots
int buffer_size = 0; // Size of buffer

void producer() {
   --m;
   ++full_slots;
   --empty_slots;
   buffer_size++;
   printf("\nProducer produces item %d", buffer_size);
   ++m;
}

void consumer() {
   --m;
   --full_slots;
   ++empty_slots;
   printf("\nConsumer consumes item %d", buffer_size);
   buffer_size--;
   ++m;
}

int main() {
   char ch = 0;
   printf("\nPress P to Producer"
          "\nPress C to Consumer"
          "\nPress 0 to Exit");

   #pragma omp critical
   while (1) {
       printf("\nEnter a choice: ");
       scanf("%c", &ch);

       switch (ch) {
       case 'P':
           if ((m == 1) && (empty_slots != 0)) {
               producer();
           } else {
               printf("Buffer is full!");
           }
```

```
            break;

        case 'C':
            if ((m == 1) && (full_slots != 0)) {
                consumer();
            } else {
                printf("Buffer is empty!");
            }
            break;

        // Exit Condition
        case '0':
            exit(0);
            break;

        default:
            break;
        }
    }
}
```

```
Press P to Producer
Press C to Consumer
Press 0 to Exit
Enter a choice: P

Producer produces item 1
Enter a choice:
Enter a choice: P

Producer produces item 2
Enter a choice:
Enter a choice: C

Consumer consumes item 2
Enter a choice:
Enter a choice: C

Consumer consumes item 1
Enter a choice:
Enter a choice: C
Buffer is empty!
Enter a choice:
Enter a choice: 0
```