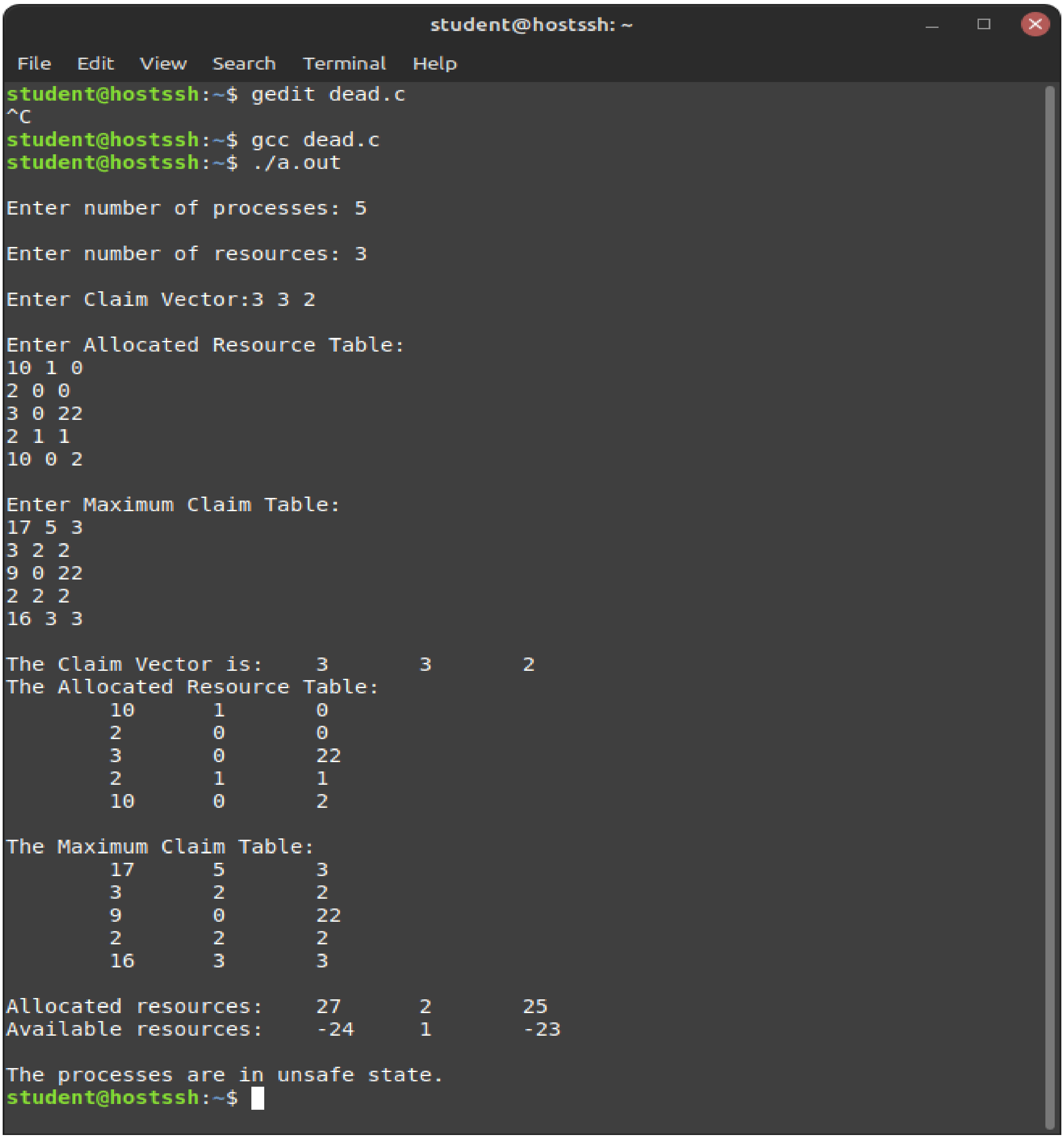
EXPERIMENT DEADLOCK

NAME: SPANDAN MUKHERJEE
REGISTRATION NUMBER: 21BCE1132
SUBJECT: OS LAB

QUESTION

1. Implement the banker's algorithm for n processes with m resources. Show the execution of your C program using suitable data set (a) with deadlock and (ii) without dead lock.

OUTPUT:



```
student@hostssh:~$ gedit dead.c
^C
student@hostssh:~$ gcc dead.c
student@hostssh:~$ ./a.out

Enter number of processes: 5

Enter number of resources: 3

Enter Claim Vector:3 3 2

Enter Allocated Resource Table:
10 1 0
2 0 0
3 0 22
2 1 1
10 0 2

Enter Maximum Claim Table:
17 5 3
3 2 2
9 0 22
2 2 2
16 3 3

The Claim Vector is:    3       3       2
The Allocated Resource Table:
        10      1       0
        2       0       0
        3       0       22
        2       1       1
        10      0       2

The Maximum Claim Table:
        17      5       3
        3       2       2
        9       0       22
        2       2       2
        16      3       3

Allocated resources:    27      2       25
Available resources:    -24     1       -23

The processes are in unsafe state.
student@hostssh:~$ 
```

CODE:

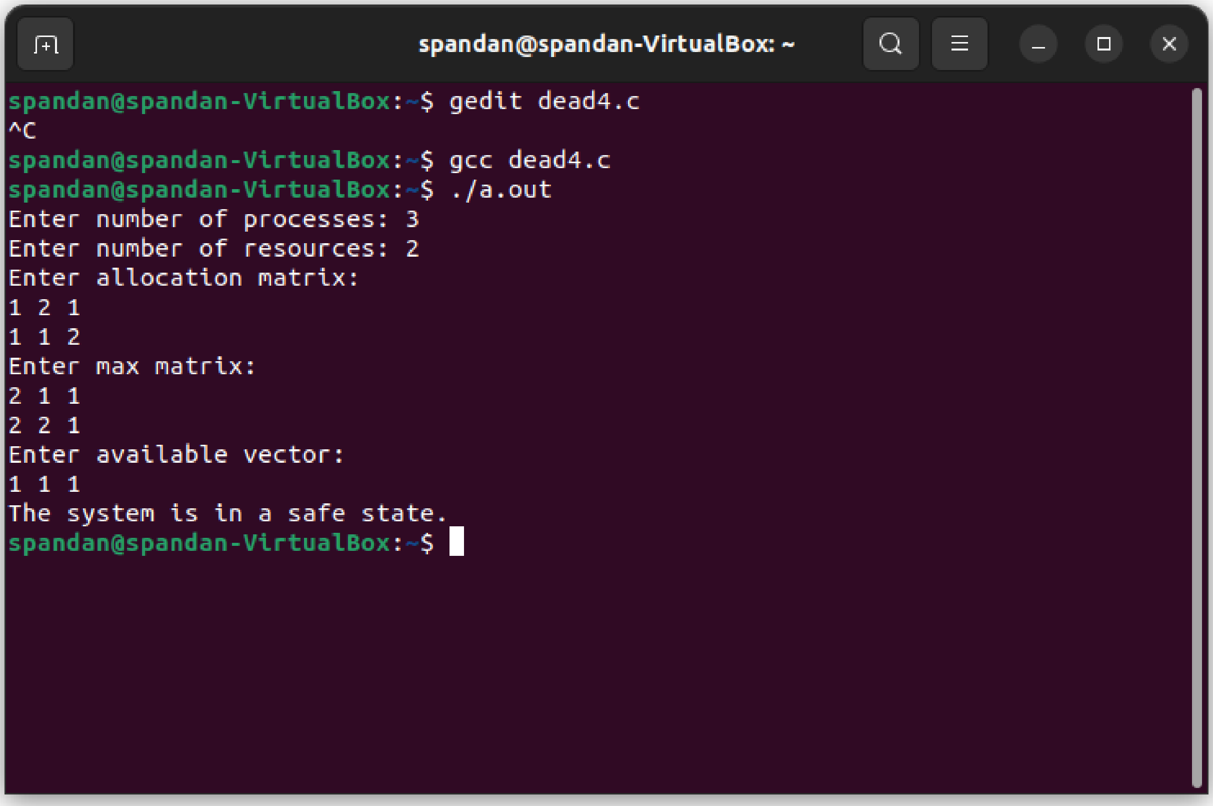
```
#include <stdio.h>

int current[5][5], maximum_claim[5][5], available[5];
int allocation[5] = {0, 0, 0, 0, 0};
int maxres[5], running[5], safe = 0;
int counter = 0, i, j, exec, resources, processes, k = 1;

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", &maxres[i]);
     }

   printf("\nEnter Allocated Resource Table:\n");
     for (i = 0; i < processes; i++)
{
        for(j = 0; j < resources; j++)
{
   scanf("%d", &current[i][j]);
         }
     }

     printf("\nEnter Maximum Claim Table:\n");
     for (i = 0; i < processes; i++)
{
         for(j = 0; j < resources; j++)
{
             scanf("%d", &maximum_claim[i][j]);
         }
     }

printf("\nThe Claim Vector is: ");
     for (i = 0; i < resources; i++)
{
        printf("\t%d", maxres[i]);
```

```
}

    printf("\nThe Allocated Resource Table:\n");
    for (i = 0; i < processes; i++)
{
      for (j = 0; j < resources; j++)
{
          printf("\t%d", current[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_claim[i][j]);
       }
       printf("\n");
    }

    for (i = 0; i < processes; i++)
{
       for (j = 0; j < resources; j++)
{
          allocation[j] += current[i][j];
       }
    }

    printf("\nAllocated resources:");
    for (i = 0; i < resources; i++)
{
       printf("\t%d", allocation[i]);
    }

    for (i = 0; i < resources; i++)
{
      available[i] = maxres[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_claim[i][j] - current[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[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;
}
```

2. Develop the C program to check whether there is a deadlock or not from Multiple Instance Resource Allocation Graph.

OUTPUT:



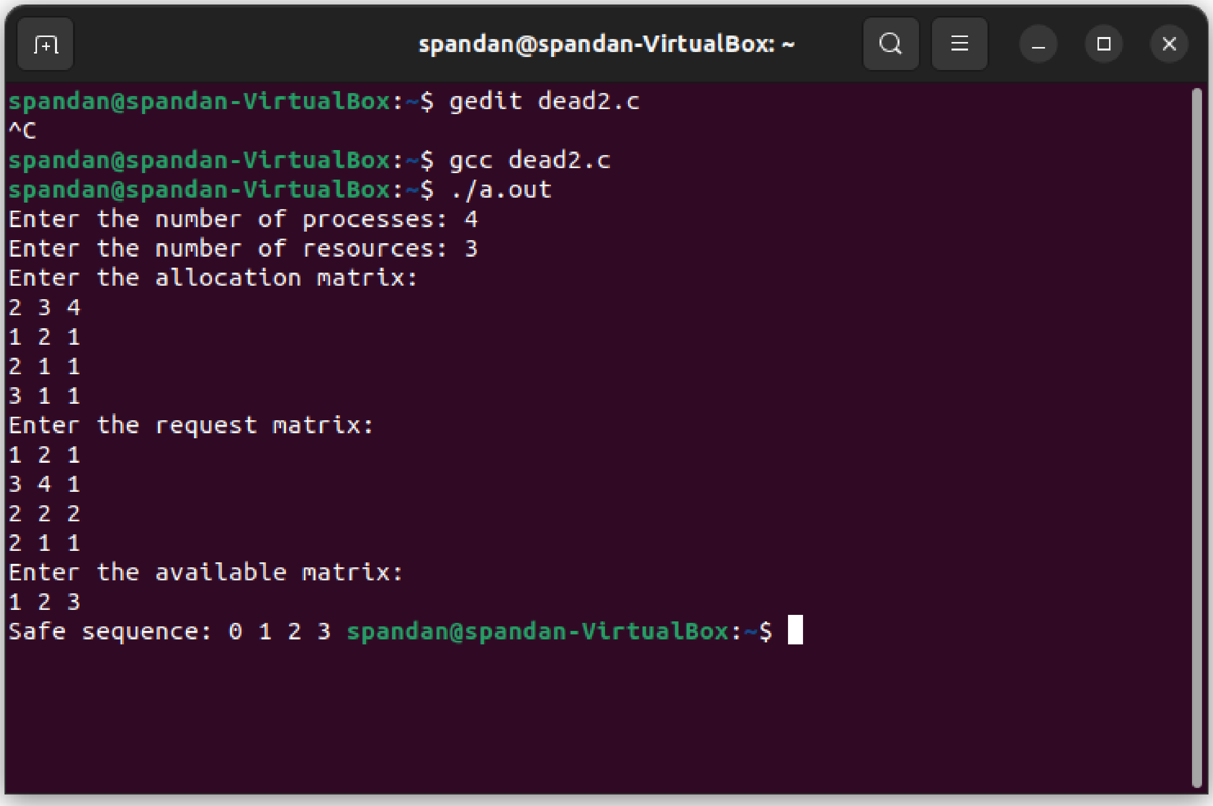
```
spandan@spandan-VirtualBox:~$ gedit dead4.c
^C
spandan@spandan-VirtualBox:~$ gcc dead4.c
spandan@spandan-VirtualBox:~$ ./a.out
Enter number of processes: 3
Enter number of resources: 2
Enter allocation matrix:
1 2 1
1 1 2
Enter max matrix:
2 1 1
2 2 1
Enter available vector:
1 1 1
The system is in a safe state.
spandan@spandan-VirtualBox:~$
```

CODE:

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int allocation[MAX_PROCESSES][MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int available[MAX_RESOURCES];
int work[MAX_RESOURCES];
int finish[MAX_PROCESSES];

int num_processes, num_resources;

void init() {
   // initialize finish array to false for all processes
   for (int i = 0; i < num_processes; i++) {
       finish[i] = 0;
   }
}

int is_safe_state() {
   // initialize work to available
   for (int i = 0; i < num_resources; i++) {
       work[i] = available[i];
   }

   // find an unfinished process with all resources less than or equal to work
   int count = 0;
   while (count < num_processes) {
       int found = 0;
       for (int i = 0; i < num_processes; i++) {
           if (finish[i] == 0) {
               int j;
               for (j = 0; j < num_resources; j++) {
                   if (need[i][j] > work[j]) {
                       break;
                   }
               }
               if (j == num_resources) {
                   // found a process that can be executed
                   finish[i] = 1;
                   for (int k = 0; k < num_resources; k++) {
                       work[k] += allocation[i][k];
                   }
                   found = 1;
                   count++;
               }
           }
       }
       if (!found) {
           // no process can be executed
           return 0;
       }
   }
   // all processes can be executed
   return 1;
}

int main() {
   printf("Enter number of processes: ");
   scanf("%d", &num_processes);

   printf("Enter number of resources: ");
   scanf("%d", &num_resources);

   // read in allocation matrix
   printf("Enter allocation matrix:\n");
   for (int i = 0; i < num_processes; i++) {
       for (int j = 0; j < num_resources; j++) {
           scanf("%d", &allocation[i][j]);
       }
   }

   // read in max matrix
   printf("Enter max matrix:\n");
   for (int i = 0; i < num_processes; i++) {
       for (int j = 0; j < num_resources; j++) {
           scanf("%d", &max[i][j]);
           need[i][j] = max[i][j] - allocation[i][j];
       }
   }

   // read in available vector
   printf("Enter available vector:\n");
   for (int i = 0; i < num_resources; i++) {
       scanf("%d", &available[i]);
   }

   init();

   if (is_safe_state()) {
       printf("The system is in a safe state.\n");
   } else {
       printf("The system is in an unsafe state.\n");
   }

   return 0;
}
```

3. Develop the C program to check whether there is a deadlock or not from Single Instance Resource Allocation Graph.

OUTPUT:



```
spandan@spandan-VirtualBox:~$ gedit dead2.c
^C
spandan@spandan-VirtualBox:~$ gcc dead2.c
spandan@spandan-VirtualBox:~$ ./a.out
Enter the number of processes: 4
Enter the number of resources: 3
Enter the allocation matrix:
2 3 4
1 2 1
2 1 1
3 1 1
Enter the request matrix:
1 2 1
3 4 1
2 2 2
2 1 1
Enter the available matrix:
1 2 3
Safe sequence: 0 1 2 3 spandan@spandan-VirtualBox:~$ 
```

CODE:

```
#include <stdio.h>
#define MAX_PROCESS 10
#define MAX_RESOURCE 10

int main(){
   int n, m; // number of processes and resources respectively
   int allocation[MAX_PROCESS][MAX_RESOURCE], // allocation matrix
      request[MAX_PROCESS][MAX_RESOURCE], // request matrix
      available[MAX_RESOURCE], // available resources
      work[MAX_RESOURCE]; // work array
   int finish[MAX_PROCESS] = { 0 }, // finish array
      safeSequence[MAX_PROCESS], // array to store safe sequence
      count = 0; // count of finished processes
   int i, j, k; // loop variables
```

```
printf("Enter the number of processes: ");
scanf("%d", &n);

printf("Enter the number of resources: ");
scanf("%d", &m);

// Input the allocation matrix
printf("Enter the allocation matrix:\n");
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        scanf("%d", &allocation[i][j]);
    }
}

// Input the request matrix
printf("Enter the request matrix:\n");
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        scanf("%d", &request[i][j]);
    }
}

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

// Initialize the work array
for (i = 0; i < m; i++) {
    work[i] = available[i];
}

// Check for deadlock
while (count < n) {
    int found = 0;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            int canFinish = 1;
            for (j = 0; j < m; j++) {
                if (request[i][j] > work[j]) {
                    canFinish = 0;
                    break;
                }
            }
            if (canFinish) {
                for (j = 0; j < m; j++) {
                    work[j] += allocation[i][j];
                }
                finish[i] = 1;
                found = 1;
```

```
                    safeSequence[count++] = i;
                }
            }
        }
        if (!found) {
            printf("Deadlock detected\n");
            return 0;
        }
    }

    // Print the safe sequence
    printf("Safe sequence: ");
    for (i = 0; i < n; i++) {
        printf("%d ", safeSequence[i]);
    }

    return 0;
}
```