Operating Systems Lab

Assessment – 7

Siddhi Singh
17BIT0028

QUESTION 1

Write a Program to implement banker's algorithm for Deadlock avoidance.

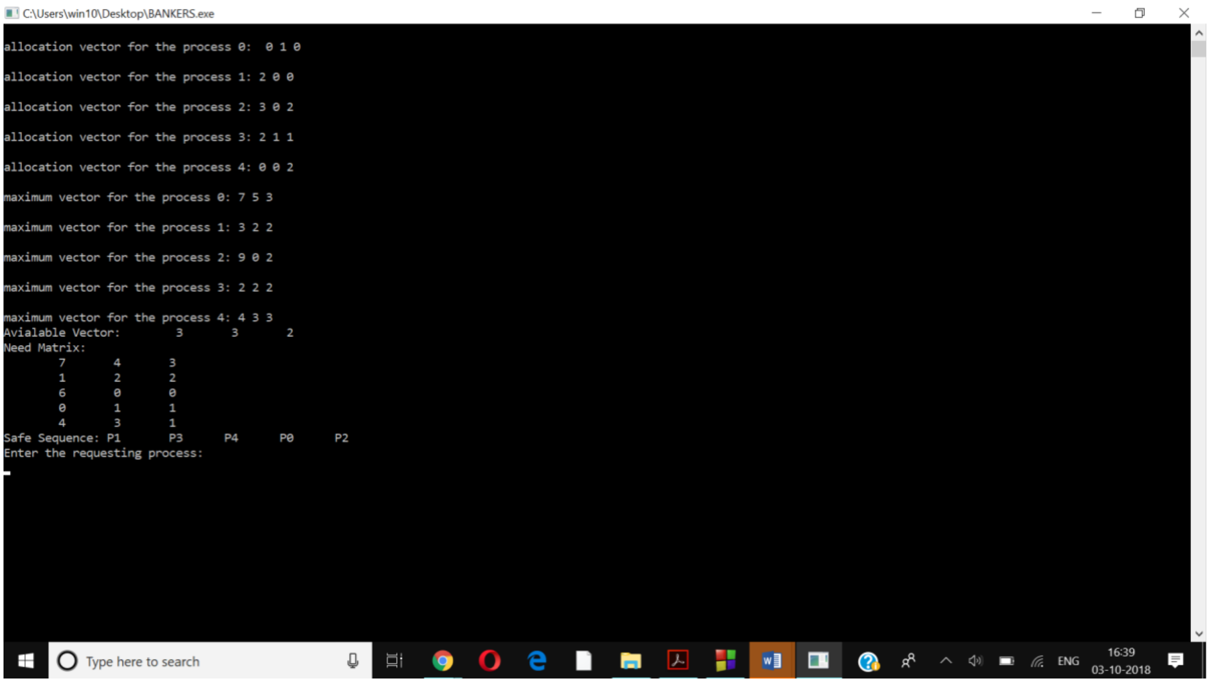
CODE

```
#include<iostream>
using namespace std;
int pt[10];
int * safety(int need[][10],int avlb[],int alloc[][10], int
p, int r)
{
int work[5],finish[5],i,j,k=0;
for(i=0;i<r;i++)
work[i]=avlb[i];
for(i=0;i<p;i++)
finish[i]=0;
i=0;
while(k<p)
{
if(finish[i]==0)
{
j=0;
while(j<r)
{
if(need[i][j]>work[j]) break;
j++;
}
if(j==r)
{
j=0;
while(j<r)
{
work[j]+=alloc[i][j];
j++;
}
pt[k]=i;
k++;
finish[i]=1;
}
}
i=(i+1)%p;
}
return pt;
}
int main()
{
int avlb[5],alloc[5][10],need[5][10],max[5]
[10],rq[5],i,j,r,t,p,rp,flag;
cout<<"Enter the number of processes: ";
cin>>p;
cout<<"Enter the number of resources: ";
cin>>r;
for(i=0;i<r;i++)
{
cout<<"Enter the instances of resource "<<i<<": ";
cin>>avlb[i];
}
for(i=0;i<p;i++)
{
```

```
cout<<"\nallocation vector for the process "<<i<<":
";
for(j=0;j<r;j++)
cin>>alloc[i][j];
}
for(i=0;i<p;i++)
{
cout<<"\nmaximum vector for the process "<<i<<":
";
for(j=0;j<r;j++)
cin>>max[i][j];
}
cout<<"Avialable Vector: ";
for(i=0;i<r;i++)
{
t=0;
for(j=0;j<p;j++)
{
t+=alloc[j][i];
need[j][i]=max[j][i]-alloc[j][i];
}
avlb[i]-=t;
cout<<"\t "<<avlb[i];
}
cout<<endl<<"Need Matrix:" <<endl;
for(i=0;i<p;i++)
{
for(j=0;j<r;j++)
cout<<"\t"<<need[i][j];
cout<<endl;
}
```

```
safety(need,avlb,alloc,p,r);
cout<<"Safe Sequence: ";
for(i=0;i<p;i++)
cout<<"P"<<pt[i]<<"\t";
cout<<endl;
```

OUTPUT



```
C:\Users\win10\Desktop\BANKERS.exe

allocation vector for the process 0:  0 1 0

allocation vector for the process 1: 2 0 0

allocation vector for the process 2: 3 0 2

allocation vector for the process 3: 2 1 1

allocation vector for the process 4: 0 0 2

maximum vector for the process 0: 7 5 3

maximum vector for the process 1: 3 2 2

maximum vector for the process 2: 9 0 2

maximum vector for the process 3: 2 2 2

maximum vector for the process 4: 4 3 3
Avialable Vector:        3       3       2
Need Matrix:
        7       4       3
        1       2       2
        6       0       0
        0       1       1
        4       3       1
Safe Sequence: P1       P3      P4      P0      P2
Enter the requesting process:
```

QUESTION 2

The Crisp Cafe in the strip mall near my house serves customers FIFO in the following way. Each customer entering the shop takes a single "ticket" with a number from a "sequencer" on the counter. The ticket numbers dispensed by the sequencer are guaranteed to be unique and sequentially increasing. When a barista (a person who makes and serves coffee) is ready to serve the next customer, it calls out the "event count", the lowest unserved number previously dispensed by the sequencer. Each customer waits until the event count reaches the number on its ticket. Each barista waits until the customer with the ticket it called places an order. Implement sequencers, tickets, and event counts using mutexes and condition variables. Your solution should also include code for the barista and customer threads.

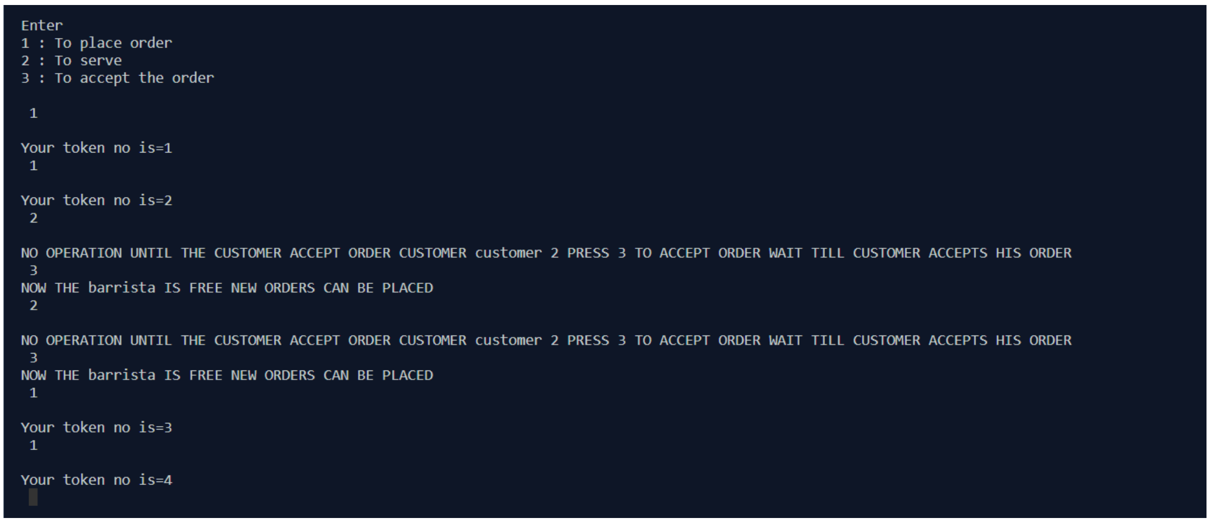
CODE

```
#include<stdio.h>
int k=0;
struct process
{
int token_no;
int order;
};

int mutex=0;

void wait()
{if(mutex<0){exit(0);}
else{mutex--;}
```

```
}
void signal()
{mutex++;
}
void serve(int i,struct process customer[])
{
if(customer[i].order==0){printf("\nCOSTOMER HAS NOT PLACED THE ODERR CANNOT SERVE UNTIL HE PLACE THE ORDER");}
else{printf("\nNO OPERATION UNTIL THE CUSTOMER ACCEPT ORDER CUSTOMER customer %d PRESS 3 TO ACCEPT ORDER ",k);wait(mutex);}
}

void accept()
{signal(mutex);}


void order(struct process p[],int j)
{if(mutex<0){printf("SHOPKEEPER IS SERVING A CUSTOMER PLEASE WAIT");}
else
{p[j].order=1;}
}
int main()
{int x;
struct process customer[100];
printf("\nEnter \n1=To place order     \n2=To serve \n3=To accept the order\n");
int n;
```

```
while(1)
{printf("\n");
scanf("%d",&n);

switch(n){
case 1:
   order(customer,++k);
   printf("\nYour token no is=%d",k);
   break;
case 2:
   serve(k,customer);
   printf("WAIT TILL CUSTOMER ACCEPTS HIS ORDER");
   break;
case 3:
   accept();
       printf("NOW THE barrista IS FREE NEW ORDERS
CAN BE PLACED");

}}
}
```

OUTPUT



```
Enter
1 : To place order
2 : To serve
3 : To accept the order

 1

Your token no is=1
 1

Your token no is=2
 2

NO OPERATION UNTIL THE CUSTOMER ACCEPT ORDER CUSTOMER customer 2 PRESS 3 TO ACCEPT ORDER WAIT TILL CUSTOMER ACCEPTS HIS ORDER
 3
NOW THE barrista IS FREE NEW ORDERS CAN BE PLACED
 2

NO OPERATION UNTIL THE CUSTOMER ACCEPT ORDER CUSTOMER customer 2 PRESS 3 TO ACCEPT ORDER WAIT TILL CUSTOMER ACCEPTS HIS ORDER
 3
NOW THE barrista IS FREE NEW ORDERS CAN BE PLACED
 1

Your token no is=3
 1

Your token no is=4
```

QUESTION 3

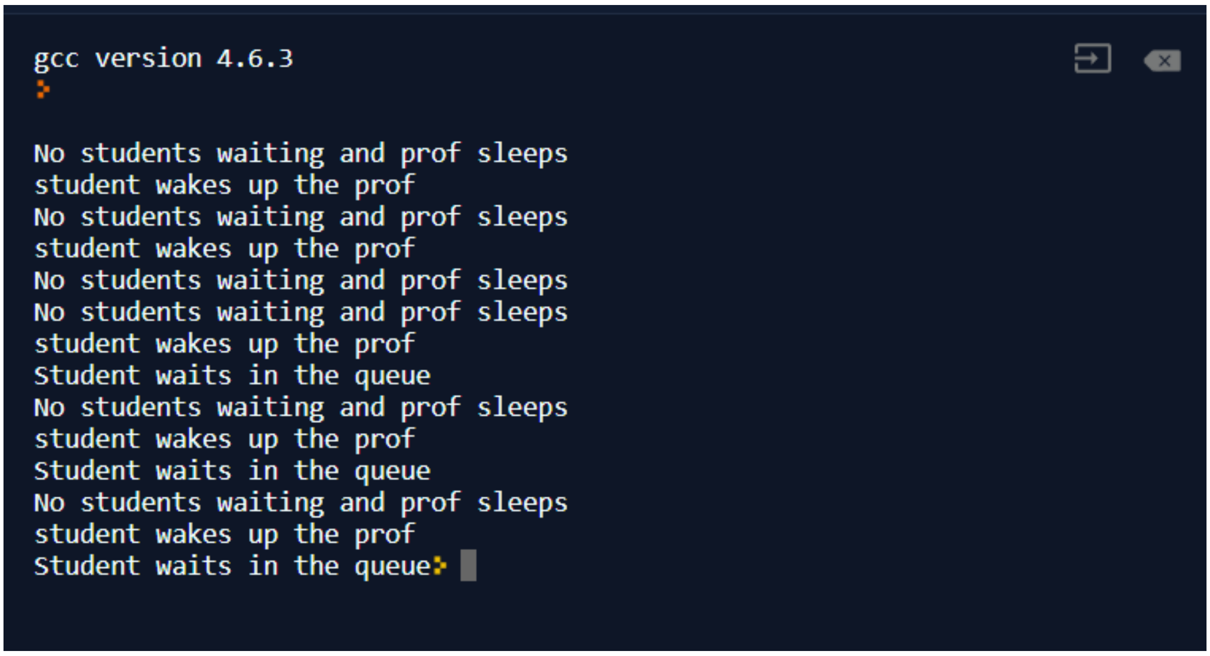
The Sleeping Professor Problem: Once class is over, professors like to sleep - except when students bother them to answer questions. You are to write procedures to synchronize threads representing one professor and an arbitrary number of students. A professor with nothing to do calls IdleProf(), which checks to see if a student is waiting outside the office to ask a question. IdleProf sleeps if there are no students waiting, otherwise it signals one student to enter the office, and returns. A student with a question to ask calls ArrivingStudent(), which joins the queue of students waiting outside the office for a signal from the professor; if no students are waiting, then the student wakes up the sleeping professor. The idea is that the professor and exactly one student will return from their respective functions "at the same time": after returning they discuss a topic of mutual interest, then the student goes back to studying and the professor calls IdleProf again. Implement IdleProf and ArrivingStudent using mutexes and condition variables.

CODE

```
#include<stdio.h>

#include<pthread.h>

#include<stdlib.h>

#include<stdbool.h>

pthread_mutex_t mutex;

pthread_cond_t prof,student;

bool profbusy=true;

int numstudents=0;

void *IdleProf(void *ptr);

void *ArrivingStudent(void *ptr);

void main(){
```

```
pthread_cond_init(&prof,NULL);
pthread_cond_init(&student,NULL);
 pthread_mutex_init(&mutex,NULL);
 pthread_t thread1, thread2;
for(int i=0;i<10;i++){
pthread_create( &thread1, NULL, IdleProf,NULL;
pthread_create(  &thread2,  NULL,  ArrivingStudent,
NULL);
}
exit(0);
}
void *IdleProf(void *ptr){
pthread_mutex_lock(&mutex);
profbusy=false;
if(numstudents==0){
printf("\nNo students waiting and prof sleeps");
pthread_cond_wait(&prof,&mutex);
}
else{
profbusy=true;
numstudents--;
pthread_cond_signal(&student);
}
pthread_mutex_unlock(&mutex);
}
void *ArrivingStudent(void *ptr){
pthread_mutex_lock(&mutex);
if(!profbusy && numstudents==0){
printf("\nstudent wakes up the prof");
profbusy=true;
pthread_cond_signal(&prof);
}
else{
printf("\nStudent waits in the queue");
numstudents++;
pthread_cond_wait(&student,&mutex);
}
pthread_mutex_unlock(&mutex);
}
```

OUTPUT



```
gcc version 4.6.3

No students waiting and prof sleeps
student wakes up the prof
No students waiting and prof sleeps
student wakes up the prof
No students waiting and prof sleeps
No students waiting and prof sleeps
student wakes up the prof
Student waits in the queue
No students waiting and prof sleeps
student wakes up the prof
Student waits in the queue
No students waiting and prof sleeps
student wakes up the prof
Student waits in the queue
```