Experiment No:01

Experiment Name: First-Come First-Served (FCFS) scheduling.

Objective:

1. To implement the FCFS scheduling algorithm.

2. To calculate waiting time, turnaround time, and completion time for each process.

3. To evaluate the performance of the FCFS algorithm using average waiting time and average turnaround time.

Algorithm:

The waiting time for the first process is 0 as it is executed first.

The waiting time for the upcoming process can be calculated by:

wt[i] = (at[i – 1] + bt[i – 1] + wt[i – 1]) – at[i]

where ,

wt[i] = waiting time of current process

at[i-1] = arrival time of previous process

bt[i-1] = burst time of previous process

wt[i-1] = waiting time of previous process

at[i] = arrival time of current process

The Average waiting time can be calculated by:

Average Waiting Time = (sum of all waiting time)/(Number of processes)

Code:

```
/*
 * FCFS Scheduling Program in C
*/
#include <stdio.h>
int main()
{
int pid[15];
int bt[15];
int n;
 printf("Enter the number of processes: ");
scanf("%d",&n);
printf("Enter process id of all the processes: ");
```

```
for(int i=0;i<n;i++)
{
scanf("%d",&pid[i]);

   }

printf("Enter burst time of all the processes: ");

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

   {

scanf("%d",&bt[i]);

       }
  int i, wt[n];
 wt[0]=0;
//for calculating waiting time of each process
for(i=1; i<n; i++)

   {
 wt[i]= bt[i-1]+ wt[i-1];

   }

   printf("Process ID     Burst Time     Waiting Time     TurnAround Time\n");

   float twt=0.0;

   float tat= 0.0;

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

 printf("%d\t\t", pid[i]);
printf("%d\t\t", bt[i]);
printf("%d\t\t", wt[i]);
  //calculating and printing turnaround time of each process
 printf("%d\t\t", bt[i]+wt[i]);
 printf("\n");
  //for calculating total waiting time

        twt += wt[i];
//for calculating total turnaround time
 tat += (wt[i]+bt[i]);

   }
float att,awt;
 //for calculating average waiting time
```

```
awt = twt/n;
//for calculating average turnaround time


    att = tat/n;


    printf("Avg. waiting time= %f\n",awt);
 printf("Avg. turnaround time= %f",att);
}
```

Input/Output:

```
ritu@Ritu:~/Desktop$ vim fcfs.c
ritu@Ritu:~/Desktop$ gcc fcfs.c -o fcfs
ritu@Ritu:~/Desktop$ ./fcfs
Enter the number of processes: 3
Enter process id of all the processes: 1 2 3
Enter burst time of all the processes: 5 11 11
Process ID     Burst Time     Waiting Time     TurnAround Time
1               5               0               5
2               11              5               16
3               11              16              27
Avg. waiting time= 7.000000
Avg. turnaround time= 16.000000ritu@Ritu:~/Desktop$
```

Experiment No:02

Experiment Name: Shortest –Job –First (SJF) scheduling.

Objective:

1. Implement the non-preemptive SJF scheduling algorithm.

2. Calculate waiting time, turnaround time, and completion time for each process.

3. Evaluate the performance of the SJF algorithm using average waiting time and average turnaround time.

Algorithm:

1. Sorting the processes based on arrival time.

2. Selecting the process with the smallest burst time among the arrived processes.

3. Calculating the waiting time, turnaround time, and completion time for each process.

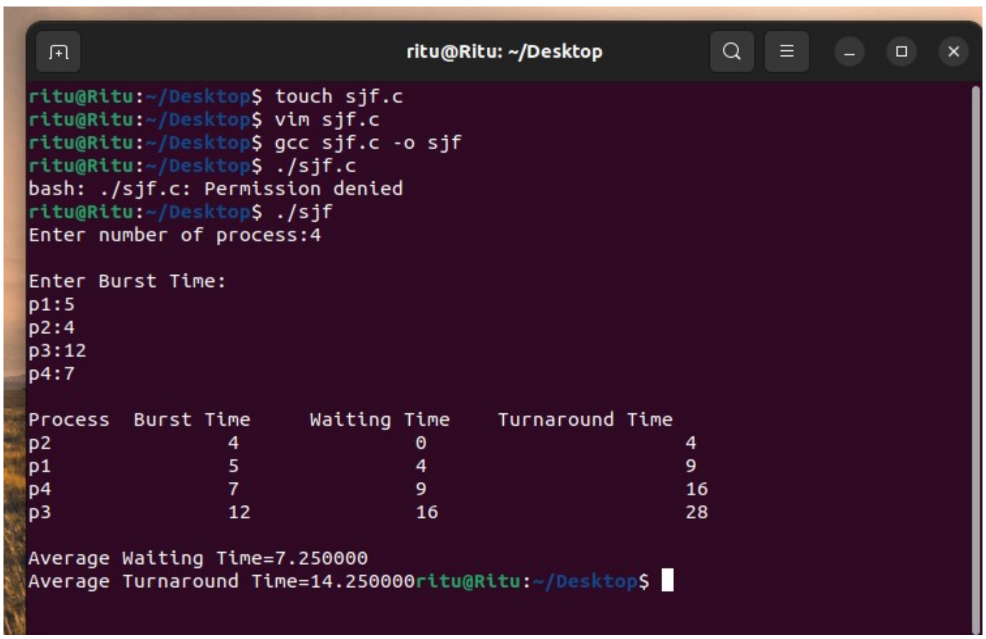
Code:

```
/*
 * C Program to Implement SJF Scheduling
 */
#include<stdio.h>
int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);
  printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;
    }
    //sorting of burst times
    for(i=0;i<n;i++)
    {
             pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }
```

```
    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;
    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}
wt[0]=0;
//finding the waiting time of all the processes
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
                    //individual WT by adding BT of all previous completed processes
        wt[i]+=bt[j];
    //total waiting time
    total+=wt[i];
}
//average waiting time
avg_wt=(float)total/n;
printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    //turnaround time of individual processes
    tat[i]=bt[i]+wt[i];
    //total turnaround time
    totalT+=tat[i];
```

```
        printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
 //average turnaround time
    avg_tat=(float)totalT/n;
    printf("\n\nAverage Waiting Time=%f",avg_wt);
           printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

Input/Output:



Experiment No:03

Experiment Name: Priority scheduling.

Objective:

1. Implement the non-preemptive priority scheduling algorithm.
2. Calculate waiting time, turnaround time, and completion time for each process.
3. Evaluate the performance of the priority scheduling algorithm using average waiting time and average turnaround time.

Algorithm:

1- First input the processes with their burst time and priority.

2- Sort the processes, burst time and priority according to the priority.

3- Now simply apply FCFS algorithm.

Code:

```
#include<stdio.h>
int main()
{
    int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
    printf("Enter Total Number of Process:");
    scanf("%d",&n);
    printf("\nEnter Burst Time and Priority\n");
    for(i=0;i<n;i++)
    {
        printf("\nP[%d]\n",i+1);
        printf("Burst Time:");
        scanf("%d",&bt[i]);
        printf("Priority:");
        scanf("%d",&pr[i]);
        p[i]=i+1;           //contains process number
    }
    //sorting burst time, priority and process number in ascending order using selection sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(pr[j]<pr[pos])
                pos=j;
```

```
    }
    temp=pr[i];
    pr[i]=pr[pos];
    pr[pos]=temp;
    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;
    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}
wt[0]=0; //waiting time for first process is zero
//calculate waiting time
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];
    total+=wt[i];
}
avg_wt=total/n;      //average waiting time
total=0;
printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];     //calculate turnaround time
    total+=tat[i];
```

```
        printf("\nP[%d]\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

    }

    avg_tat=total/n;      //average turnaround time

    printf("\n\nAverage Waiting Time=%d",avg_wt);

    printf("\nAverage Turnaround Time=%d\n",avg_tat);

return 0;

}
```

Input/Output:

```
ritu@Ritu:~/Desktop$ vim pri.c
ritu@Ritu:~/Desktop$ gcc pri.c -o pri
ritu@Ritu:~/Desktop$ ./pri
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:6
Priority:3

P[2]
Burst Time:2
Priority:2

P[3]
Burst Time:14
Priority:1

P[4]
Burst Time:6
Priority:4

Process     Burst Time          Waiting Time     Turnaround Time
P[3]            14                  0                  14
P[2]            2                   14                 16
```

```
P[1]            6                   16                 22
P[4]            6                   22                 28

Average Waiting Time=13
Average Turnaround Time=20
ritu@Ritu:~/Desktop$ 
```

Experiment No:04

Experiment Name: Round Robin scheduling.

Objective:

1. Implement the Round Robin scheduling algorithm in C.
2. Calculate waiting time, turnaround time, and completion time for each process.

3. Evaluate the performance of the Round Robin algorithm using average waiting time and average turnaround time.

Algorithm:

- Completion Time: Time at which process completes its execution.
- Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
- Waiting Time(W.T): Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time

Code:

```
/*
 * Round Robin Scheduling Program in C
 */
#include<stdio.h>
int main()
{
    //Input no of processed
   int  n;
    printf("Enter Total Number of Processes:");
    scanf("%d", &n);
    int wait_time = 0, ta_time = 0, arr_time[n], burst_time[n], temp_burst_time[n];
    int x = n;
    //Input details of processes
    for(int i = 0; i < n; i++)
    {
        printf("Enter Details of Process %d \n", i + 1);
        printf("Arrival Time:  ");
        scanf("%d", &arr_time[i]);
        printf("Burst Time:   ");
```

```
    scanf("%d", &burst_time[i]);
    temp_burst_time[i] = burst_time[i];
}
//Input time slot
int time_slot;
printf("Enter Time Slot:");
scanf("%d", &time_slot);
//Total indicates total time
//counter indicates which process is executed
int total = 0,  counter = 0,i;
printf("Process ID       Burst Time       Turnaround Time      Waiting Time\n");
for(total=0, i = 0; x!=0; )
{
    // define the conditions
    if(temp_burst_time[i] <= time_slot && temp_burst_time[i] > 0)
    {
        total = total + temp_burst_time[i];
        temp_burst_time[i] = 0;
        counter=1;
    }
    else if(temp_burst_time[i] > 0)
    {
        temp_burst_time[i] = temp_burst_time[i] - time_slot;
        total  += time_slot;
    }
    if(temp_burst_time[i]==0 && counter==1)
    {
```

```
            x--; //decrement the process no.
            printf("\nProcess No %d  \t\t %d\t\t\t\t %d\t\t\t %d", i+1, burst_time[i],
                  total-arr_time[i], total-arr_time[i]-burst_time[i]);
            wait_time = wait_time+total-arr_time[i]-burst_time[i];
            ta_time += total -arr_time[i];
            counter =0;
        }
        if(i==n-1)
        {
            i=0;
        }
        else if(arr_time[i+1]<=total)
        {
            i++;
        }
        else
        {
            i=0;
        }
    }
    float average_wait_time = wait_time * 1.0 / n;
    float average_turnaround_time = ta_time * 1.0 / n;
    printf("\nAverage Waiting Time:%f", average_wait_time);
    printf("\nAvg Turnaround Time:%f", average_turnaround_time);
return 0;
}
```

Input/Output:

```
ritu@Ritu: ~/Desktop
ritu@Ritu:~/Desktop$ vim round.c
ritu@Ritu:~/Desktop$ gcc round.c -o round
ritu@Ritu:~/Desktop$ ./round
Enter Total Number of Processes:3
Enter Details of Process 1
Arrival Time:  0
Burst Time:   10
Enter Details of Process 2
Arrival Time:  1
Burst Time:   8
Enter Details of Process 3
Arrival Time:  2
Burst Time:   7
Enter Time Slot:5
Process ID       Burst Time       Turnaround Time      Waiting Time

Process No 1             10                              20
10
Process No 2             8                               22
14
Process No 3             7                               23
16
Average Waiting Time:13.333333
Avg Turnaround Time:21.666666ritu@Ritu:~/Desktop$
```

Experiment No:05

Experiment Name: Producer/ Consumer problem.

Objective:

1. Implement the Producer-Consumer problem using a bounded buffer.
2. Use semaphores to ensure proper synchronization between producers and consumers.
3. Evaluate the performance and correctness of the implementation.

Algorithm:

initialize buffer as array of size BUFFER_SIZE

initialize in = 0, out = 0

initialize semaphores: empty = BUFFER_SIZE, full = 0

initialize mutex: mutex

Producer Process:

while true do

  item = produce_item() // Generate item

```
   sem_wait(empty)        // Wait if buffer is full
   pthread_mutex_lock(mutex)  // Enter critical section
buffer[in] = item      // Add item to buffer
   in = (in + 1) % BUFFER_SIZE
 pthread_mutex_unlock(mutex)  // Exit critical section
   sem_post(full)       // Signal that buffer is not empty
 // Repeat the process
```

Consumer Process:

```
while true do
   sem_wait(full)        // Wait if buffer is empty
   pthread_mutex_lock(mutex)  // Enter critical section
item = buffer[out]    // Remove item from buffer
   out = (out + 1) % BUFFER_SIZE
   pthread_mutex_unlock(mutex)  // Exit critical section
   sem_post(empty)       // Signal that buffer is not full
   consume_item(item)    // Process the item
   // Repeat the process
```

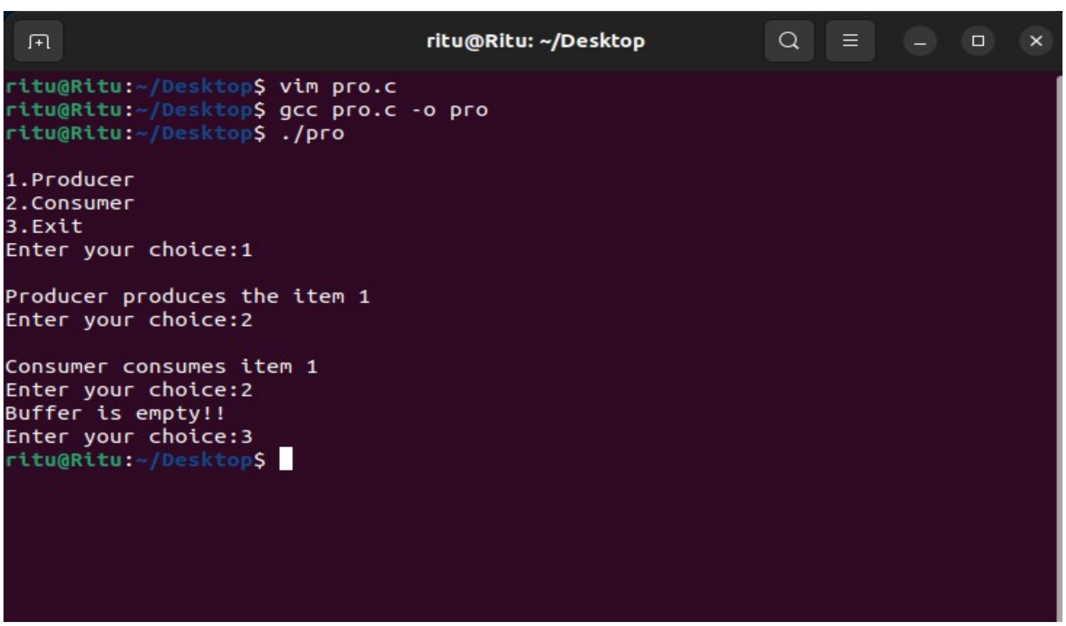
Code:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
```

```
printf("\n1.Producer\n2.Consumer\n3.Exit");
while(1)
{
printf("\nEnter your choice:");
scanf("%d",&n);
switch(n)
{
case 1: if((mutex==1)&&(empty!=0))
producer();
else
printf("Buffer is full!!");
break;
case 2: if((mutex==1)&&(full!=0))
consumer();
else
printf("Buffer is empty!!");
break;
case 3:
exit(0);
break;
}
}
return 0;
}
int wait(int s)
{
return (--s);}
```

```
 int signal(int s){
return(++s);}
void producer(){
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\nProducer produces the item %d",x);
mutex=signal(mutex);}
void consumer(){
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\nConsumer consumes item %d",x);
x--;
mutex=signal(mutex);}
```

Input/Output:



Experiment No:07

Experiment Name: Readers and writers problem.

Objective:

1. Implement the Readers-Writers problem using proper synchronization techniques.
2. Ensure that multiple readers can read concurrently while only one writer can write at a time.
3. Evaluate the performance and correctness of the implementation.

Algorithm:

```
initialize rw_mutex as semaphore with value 1

initialize mutex as mutex

initialize read_count = 0

Reader Process:

while true do

  pthread_mutex_lock(mutex) // Protect read_count

  read_count = read_count + 1

  if read_count == 1 then

     sem_wait(rw_mutex) // First reader locks the resource

  pthread_mutex_unlock(mutex) // Release protection of read_count

  // Read the resource

  print("Reader is reading")

  sleep(1) // Simulate reading time

  pthread_mutex_lock(mutex) // Protect read_count

  read_count = read_count - 1

  if read_count == 0 then

     sem_post(rw_mutex) // Last reader unlocks the resource

  pthread_mutex_unlock(mutex) // Release protection of read_count

  sleep(1) // Simulate time between reads

Writer Process:
```

```
while true do
    sem_wait(rw_mutex) // Lock the resource
    // Write the resource
    print("Writer is writing")
    sleep(1) // Simulate writing time
    sem_post(rw_mutex) // Unlock the resource
    sleep(1) // Simulate time between writes
```

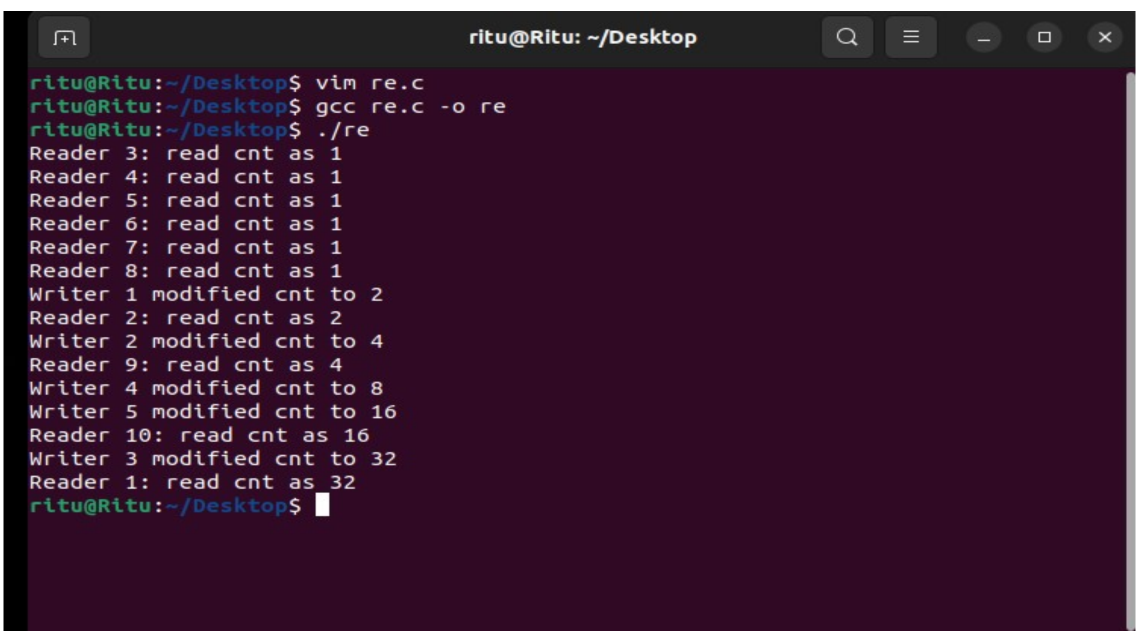
Code:

```c
#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

/*

This program provides a possible solution for first readers writers problem using mutex and semaphore.

I have used 10 readers and 5 producers to demonstrate the solution. You can always play with these values.

*/

sem_t wrt;

pthread_mutex_t mutex;

int cnt = 1;

int numreader = 0;

void *writer(void *wno)

{

    sem_wait(&wrt);

    cnt = cnt*2;

    printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);

    sem_post(&wrt);
```

```
}
void *reader(void *rno)
{
    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader++;
    if(numreader == 1) {
        sem_wait(&wrt); // If this id the first reader, then it will block the writer
    }
    pthread_mutex_unlock(&mutex);
    // Reading Section
    printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);

    // Reader acquire the lock before modifying numreader
    pthread_mutex_lock(&mutex);
    numreader--;
    if(numreader == 0) {
        sem_post(&wrt); // If this is the last reader, it will wake up the writer.
    }
    pthread_mutex_unlock(&mutex);
}
int main()
{
    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&wrt,0,1);
  int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and consumer
```

```
for(int i = 0; i < 10; i++) {
    pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
  }
  for(int i = 0; i < 5; i++) {
    pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
  }
  for(int i = 0; i < 10; i++) {
    pthread_join(read[i], NULL);
  }
  for(int i = 0; i < 5; i++) {
    pthread_join(write[i], NULL);
  }
  pthread_mutex_destroy(&mutex);
  sem_destroy(&wrt);
  return 0;
}
```

Input/Output:



Experiment No: 11

Experiment Name: Banker’s Algorithm.

Objective:

The primary objective of the Banker's Algorithm is to ensure that the system remains in a safe state by allocating resources only if it does not lead to a deadlock.

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];

Code:

```
#include<stdio.h>
int main() {
  /* array will store at most 5 process with 3 resoures if your process or
 resources is greater than 5 and 3 then increase the size of array */
int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3], done[5],
terminate = 0;

  printf("Enter the number of process and resources");

  scanf("%d %d", & p, & c);

  // p is process and c is diffrent resources

  printf("enter allocation of resource of all process %dx%d matrix", p, c);

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

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

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

    }

  }

  printf("enter the max resource process required %dx%d matrix", p, c);

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

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

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

    }

  }

  printf("enter the  available resource");

  for (i = 0; i < c; i++)
  scanf("%d", & available[i]);
```

```
printf("\n need resources matrix are\n");
 for (i = 0; i < p; i++) {
   for (j = 0; j < c; j++) {
     need[i][j] = max[i][j] - alc[i][j];
     printf("%d\t", need[i][j]);
   }
   printf("\n");
 }
 /* once process execute variable done will stop them for again execution */
 for (i = 0; i < p; i++) {
   done[i] = 0;
 }
 while (count < p) {
   for (i = 0; i < p; i++) {
     if (done[i] == 0) {
       for (j = 0; j < c; j++) {
         if (need[i][j] > available[j])
           break;
       }
       //when need matrix is not greater then available matrix then if j==c will true
       if (j == c) {
         safe[count] = i;
         done[i] = 1;
         /* now process get execute release the resources and add them in available resources */
         for (j = 0; j < c; j++) {
           available[j] += alc[i][j];
    }
```

```
          count++;
          terminate = 0;
        } else {
          terminate++;
        }
      }
    }
    if (terminate == (p - 1)) {
      printf("safe sequence does not exist");
      break;
    }
  }
  if (terminate != (p - 1)) {
    printf("\n available resource after completion\n");
    for (i = 0; i < c; i++) {
      printf("%d\t", available[i]);
    }
    printf("\n safe sequence are\n");
    for (i = 0; i < p; i++) {
      printf("p%d\t", safe[i]);
    }
  }
  return 0;
}
```

Input/Output:

ritu@Ritu: ~/Desktop

```
ritu@Ritu:~/Desktop$ vim banker.c
ritu@Ritu:~/Desktop$ gcc banker.c -o banker
ritu@Ritu:~/Desktop$ ./banker
Enter the number of process and resources5 3
enter allocation of resource of all process 5x3 matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
enter the max resource process required 5x3 matrix
7 5 3
3 2 2
9 0 2
4 2 2
5 3 3
enter the  available resource
3 3 2

 need resources matrix are
7       4       3
1       2       2
6       0       0
2       1       1
```

```
5       3       1

 available resource after completion
10      5       7
 safe sequence are
p1      p3      p4      p0      p2      ritu@Ritu:~/Desktop$
```