EXPERIMENT NO:-5

AIM: - Write a program to implement CPU Scheduling algorithms like FCFS & SJF.

THEORY:-

1. **FIRST-COME, FIRST-SERVE SCHEDULING (FCFS):**

In this, which process enter the ready queue first is served first. The OS maintains DS that is ready queue. It is the simplest CPU scheduling algorithm. If a process request the CPU then it is loaded into the ready queue, which process is the head of the ready queue, connect the CPU to that process.

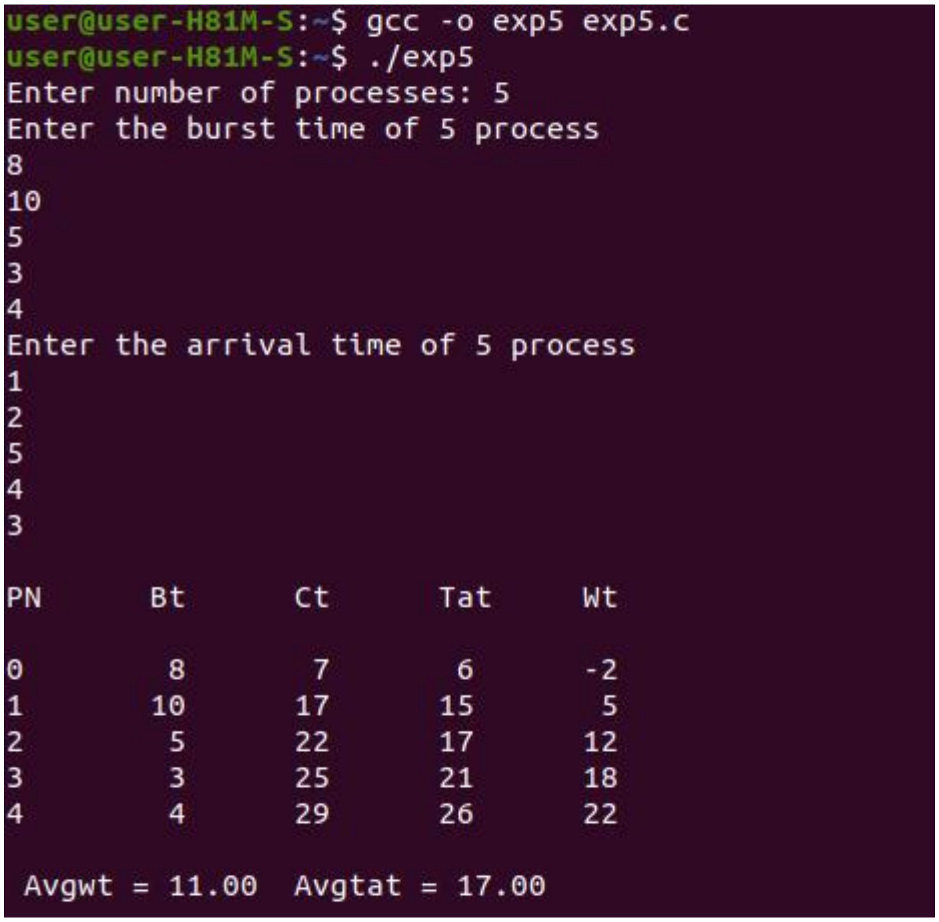
Algorithm for FCFS scheduling:

```
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Set the waiting of the first process as ‘0’ and its burst time as its turn around time
Step 5: for each process in the Ready Q calculate
    (c) Waiting time for process(n)=waiting time of process (n-1) + Bursttime of process(n-1)
    (d) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
Step 6: Calculate
    (e) Average waiting time = Total waiting Time / Number of process
    (f) Average Turnaround time = Total Turnaround Time / Number of process
Step 7: Stop the process
```

/* Program to Simulate First Come First Serve CPU Scheduling Algorithm */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main(){
int i,j,n,bt[10],compt[10],at[10], wt[10],tat[10];
float   sumwt=0.0,sumtat=0.0,avgwt,avgtat;
clrscr();
printf("Enter number of processes: ");
scanf("%d",&n);
printf("Enter the burst time of %d process\n", n);
for(i=0;i<n;i++){
scanf("%d",&bt[i]);}
printf("Enter the arrival time of %d process\n", n);
for(i=0;i<n;i++){
scanf("%d",&at[i]);
}
compt[0]=bt[0]-at[0];
for(i=1;i<n;i++)
compt[i]=bt[i]+compt[i-1];
for(i=0;i<n;i++){
tat[i]=compt[i]-at[i];
wt[i]=tat[i]-bt[i];
sumtat+=tat[i];
sumwt+=wt[i];
}
avgwt=sumwt/n;
avgtat=sumtat/n;
printf("..............................................\n");
printf("PN\tBt\tCt\tTat\tWt\n");
printf("..............................................\n");
for(i=0;i<n;i++){
printf("%d\t%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);}
printf("..............................................\n");
printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
printf("...............................................\n");
getch();
}
```

OUTPUT:



```
user@user-H81M-S:~$ gcc -o exp5 exp5.c
user@user-H81M-S:~$ ./exp5
Enter number of processes: 5
Enter the burst time of 5 process
8
10
5
3
4
Enter the arrival time of 5 process
1
2
5
4
3

PN      Bt      Ct      Tat     Wt

0        8       7       6      -2
1       10      17      15       5
2        5      22      17      12
3        3      25      21      18
4        4      29      26      22

 Avgwt = 11.00  Avgtat = 17.00
```

1. **SHORTEST JOB FIRST:**

The criteria of this algorithm are which process having the smallest CPU burst, CPU is assigned to that next process. If two process having the same CPU burst time FCFS is used to break the tie.

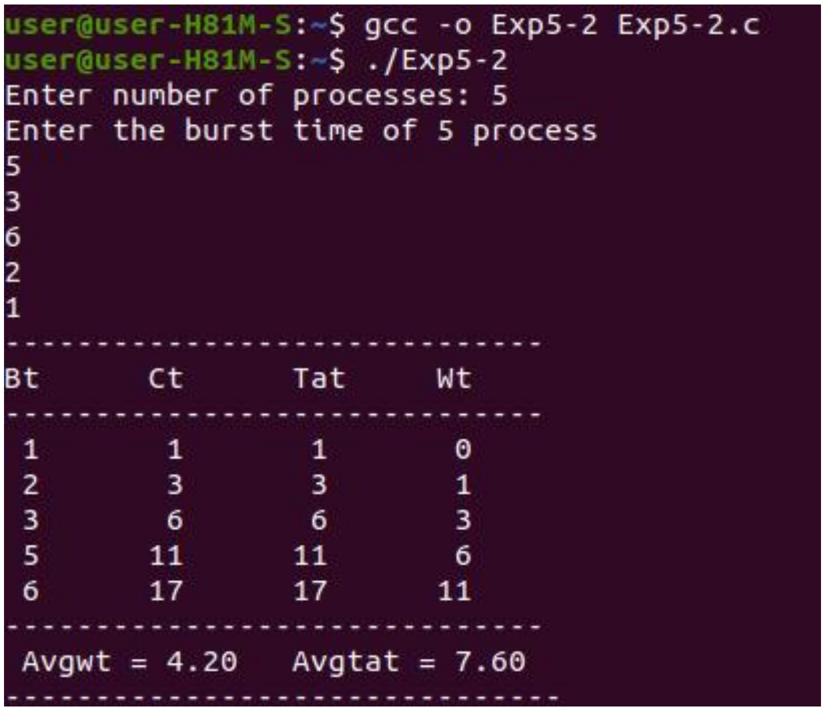
Algorithm for SJF:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as '0' and its turnaround time as its burst time.
Step 6: For each process in the ready queue, calculate
(a) Waiting time for process(n)=waiting time of process (n-1) + Bursttime of process(n-1)
b Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)
Step 7: Calculate
C Average waiting time = Total waiting Time / Number of process
D Average Turnaround time = Total Turnaround Time / Number of process
Step 8: Stop the process

/* Program to Simulate Shortest Job First CPU Scheduling Algorithm */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main(){
int i,j,n,bt[10],compt[10], wt[10],tat[10],temp;
float   sumwt=0.0,sumtat=0.0,avgwt,avgtat;
clrscr();
printf("Enter number of processes: ");
scanf("%d",&n);
printf("Enter the burst time of %d process\n", n);
for(i=0;i<n;i++){
scanf("%d",&bt[i]);}
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
if(bt[i]>bt[j]){
temp=bt[i];bt[i]=bt[j];
bt[j]=temp;
}
compt[0]=bt[0]; for(i=1;i<n;i++)
compt[i]=bt[i]+compt[i-1];
for(i=0;i<n;i++)
{
tat[i]=compt[i];
wt[i]=tat[i]-bt[i];
sumtat+=tat[i];
sumwt+=wt[i];
}
avgwt=sumwt/n;
avgtat=sumtat/n;
printf("..........................................\n");
printf("Bt\tCt\tTat\tWt\n");
printf("..........................................\n");
for(i=0;i<n;i++){
printf("%2d\t%2d\t%2d\t%2d\n",bt[i],compt[i],tat[i],wt[i]);
}
printf("..........................................\n");
printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
printf("...........................................\n");
getch();
}
```

OUTPUT:



```
user@user-H81M-S:~$ gcc -o Exp5-2 Exp5-2.c
user@user-H81M-S:~$ ./Exp5-2
Enter number of processes: 5
Enter the burst time of 5 process
5
3
6
2
1
------------------------------
Bt      Ct      Tat     Wt
------------------------------
 1       1       1       0
 2       3       3       1
 3       6       6       3
 5      11      11       6
 6      17      17      11
------------------------------
 Avgwt = 4.20   Avgtat = 7.60
-------------------------------
```

CONCLUSION:- Thus we have studied FCFS & SJF scheduling algorithm and its implementation.

SIGN AND REMARK

R1 (3 Marks)	R2 (3 Marks)	R3 (3 Marks)	R4 (3 Mark)	R5 (3 Mark)	Total (15 Marks)	Signature

EXPERIMENT NO:- 6

AIM:- Program to simulate producer and consumer problem using semaphores

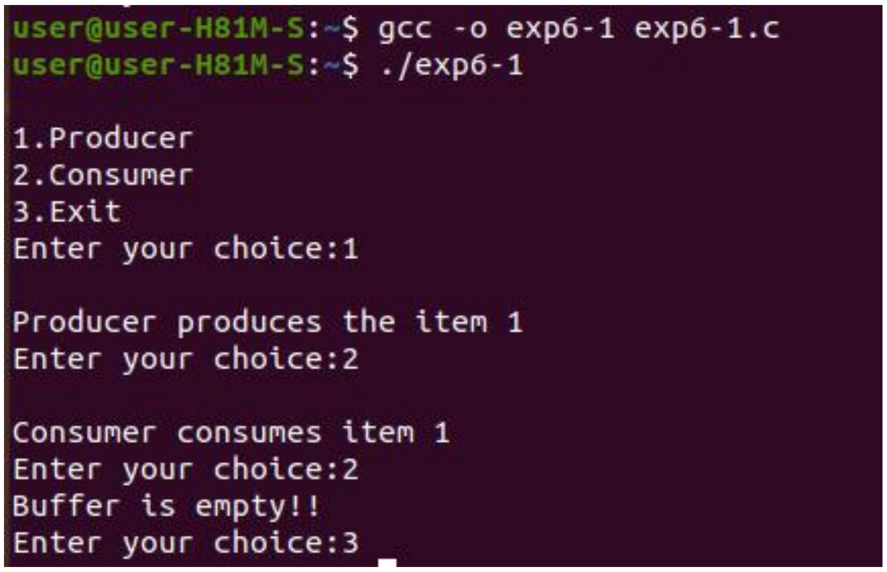
THEORY:-

The producer consumer problem is a synchronization problem. We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

```
#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);
}
```

`Output:`



```
user@user-H81M-S:~$ gcc -o exp6-1 exp6-1.c
user@user-H81M-S:~$ ./exp6-1

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3
```

CONCLUSION:- Hence we have studied and implemented semaphore to simulate producer and consumable problem.

SIGN AND REMARK

R1 **(3 Marks)**	**R2** **(3 Marks)**	**R3** **(3 Marks)**	**R4** **(3 Mark)**	**R5** **(3 Mark)**	**Total** **(15 Marks)**	**Signature**

EXPERIMENT NO:-7

AIM:- Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm

THEORY:-

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker's algorithm is named so?

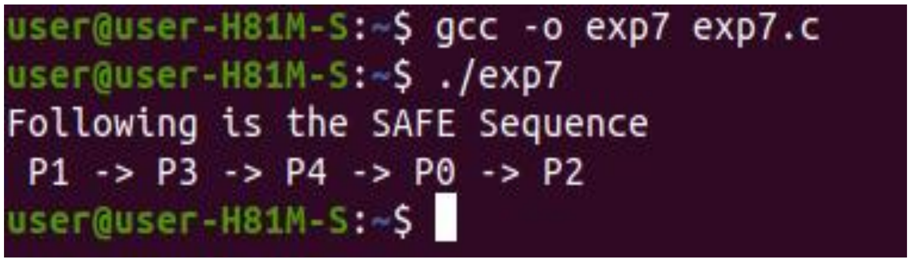
Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

```
// Banker's Algorithm
#include <stdio.h>
int main()
{// P0, P1, P2, P3, P4 are the Process names here
       int n, m, i, j, k;n = 5; // Number of
    processes m = 3; // Numberof resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4
    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4
    int avail[3] = { 3, 3, 2 }; // Available Resources
    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] - alloc[i][j];
       }
    int y = 0:
 for (k = 0; k < 5; k++) { for
 (i = 0; i < n; i++)
```

```
            { if (f[i] == 0) {
                int flag = 0;
                for (j = 0; j < m; j++){
                    if (need[i][j] > avail[j]){flag
                        = 1;
                         break;
                    }
                }
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++) avail[y]
                        += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n -
    1]); return (0);
}
```

Output:



```
user@user-H81M-S:~$ gcc -o exp7 exp7.c
user@user-H81M-S:~$ ./exp7
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
user@user-H81M-S:~$
```

CONCLUSION:- Hence we have studied and implemented the concept of deadlock avoidance through Banker's Algorithm.

SIGN AND REMARK

R1 (3 Marks)	R2 (3 Marks)	R3 (3 Marks)	R4 (3 Mark)	R5 (3 Mark)	Total (15 Marks)	Signature