Table of Contents

```
Enter the number of processes
5
Enter the process number
1
Enter the burst time of the process
6
Enter the priority of the process
3
Enter the process number
2
Enter the burst time of the process
8
Enter the priority of the process
1
Enter the process number
4
Enter the burst time of the process
7
Enter the priority of the process
2
Enter the process number
3
Enter the burst time of the process
7
Enter the priority of the process
4
Enter the process number
5
Enter the burst time of the process
9
Enter the priority of the process
5
Processid        Burst Time      Priority        Waiting Time     Turn Around Time
2                8               1                0               8
4                7               2                8               15
1                6               3                15              21
3                7               4                21              28
5                9               5                28              37
The average waiting time =  14.400000
The average turn aroud time = 21.799999
```

EXP. 6 BANKERS ALGORITHM

CODE:

```
#include<stdio.h>
#include<conio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
int i,j;
printf("********** Baner's Algo ***********\n");
input();
show();
cal();
getch();
return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
```

```
scanf("%d",&max[i][j]);
}}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
scanf("%d",&alloc[i][j]);
}}
printf("Enter the available Resources\n");
for(j=0;j<r;j++) {
scanf("%d",&avail[j]);
}}
void show() {
int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++) {
printf("\nP%d\t ",i+1);
for(j=0;j<r;j++) {
printf("%d ",alloc[i][j]); }
printf("\t");
for(j=0;j<r;j++) {
printf("%d ",max[i][j]); }
printf("\t");
if(i==0) {
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}}}
void cal()
{
int finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
int i,j;
for(i=0;i<n;i++) {
finish[i]=0; }
//find need matrix
for(i=0;i<n;i++) {
```

```
for(j=0;j<r;j++) {
need[i][j]=max[i][j]
-alloc[i][j];
}}
printf("\n");
while(flag) {
flag=0;
for(i=0;i<n;i++) {
int c=0;
for(j=0;j<r;j++) {
if((finish[i]==0)&&(need[i][j]<=avail[j])) {
c++;
if(c==r) {
for(k=0;k<r;k++) {
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1; }
printf("P%d->",i);
if(finish[i]==1) {
i=n;
}}}}}}
for(i=0;i<n;i++) {
if(finish[i]==1) {
c1++;
}
else
{printf("P%d->",i);
}}
if(c1==n)
{printf("\n The system is in safe state");
}
else
{
printf("\n Process are in dead lock");
printf("\n System is in unsafe state");
```

}}

```
********** Baner's Algo ************
Enter the no of Processes       5
Enter the no of resources instances     4
Enter the Max Matrix
0 0 1 2
1 7 5 0
2 3 5 6
0 6 5 2
0 6 5 2
Enter the Allocation Matrix
0 0 1 2
1 0 0 0
1 3 5 7
0 6 3 2
0 0 1 2
Enter the available Resources
1 5 2 0
Process  Allocation      Max     Available
P1       0 0 1 2        0 0 1 2         1 5 2 0
P2       1 0 0 0        1 7 5 0
P3       1 3 5 7        2 3 5 6
P4       0 6 3 2        0 6 5 2
P5       0 0 1 2        0 6 5 2
P0->P2->P1->P3->P4->
 The system is in safe state
```

EXP. 7 MEMORY ALLOCATION METHODS FOR FIXED PARTITION

BEST FIT

AIM:

To write a C program for implementation of FCFS and SJF scheduling algorithms.

ALGORITHM:

Step 1:Define the max as 25.

Step 2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0,

bf[max],ff[max]. Step 3: Get the number of blocks,files,size of the blocks using for loop.

Step 4: In for loop check bf[j]!=1, if so temp=b[j]-f[i]

Step 5: Check lowest>temp,if so assign ff[i]=j,highest=temp

Step 6: Assign frag[i]=lowest, bf[ff[i]]=1,lowest=10000

Step 7: Repeat step 4 to step 6.

Step 8: Print file no,size,block no,size and fragment.

Step 9: Stop the program.

CODE:

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
```

```
static int bf[max],ff[max];
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
if(lowest>temp)
{
ff[i]=j;
lowest=temp;
}
}
}
frag[i]=lowest;
bf[ff[i]]=1;
lowest=10000;
```

```
}
printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");
for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```

```
Enter the number of blocks:5
Enter the number of files:4

Enter the size of the blocks:-
Block 1:100
Block 2:500
Block 3:200
Block 4:300
Block 5:800
Enter the size of the files :-
File 1:112
File 2:317
File 3:221
File 4:436

File No File Size       Block No        Block Size      Fragment
1               112             3               200             88
2               317             2               500             183
3               221             4               300             79
4               436             5               800             364
```

EXP. 8 MEMORY ALLOCATION METHODS FOR FIXED PARTITION FIRST FIT

AIM:

To write a C program for implementation memory allocation methods for fixed partition using first fit.

ALGORITHM:

Step 1:Define the max as 25.

Step 2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0,

bf[max],ff[max]. Step 3: Get the number of blocks,files,size of the blocks using for loop.

Step 4: In for loop check bf[j]!=1, if so temp=b[j]-f[i]

Step 5: Check highest<temp,if so assign ff[i]=j,highest=temp

Step 6: Assign frag[i]=highest, bf[ff[i]]=1,highest=0

Step 7: Repeat step 4 to step 6.

Step 8: Print file no,size,block no,size and fragment.

Step 9: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
```

```
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
Compiled By : Rohit Kumar Bisht, M.E Computer 63
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
```

```
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch(); }
```

```
        Memory Management Scheme - First Fit
Enter the number of blocks:5
Enter the number of files:4

Enter the size of the blocks:-
Block 1:100
Block 2:500
Block 3:200
Block 4:300
Block 5:800
Enter the size of the files :-
File 1:112
File 2:317
File 3:221
File 4:436

File_no:        File_size :     Block_no:       Block_size:     Fragement
1               112             2               500             388
2               317             5               800             483
3               221             4               300             79
4               436             0               6487464         -236
```

EXP. 9 MEMORY ALLOCATION METHODS FOR FIXED PARTITION

WORST FIT

AIM:

To write a C program for implementation of FCFS and SJF scheduling algorithms.

ALGORITHM:

Step 1:Define the max as 25.

Step 2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0,

bf[max],ff[max]. Step 3: Get the number of blocks,files,size of the blocks using for loop.

Step 4: In for loop check bf[j]!=1, if so temp=b[j]-f[i]

Step 5: Check temp>=0,if so assign ff[i]=j break the for loop.

Step 6: Assign frag[i]=temp,bf[ff[i]]=1;

Step 7: Repeat step 4 to step 6.

Step 8: Print file no,size,block no,size and fragment.

Step 9: Stop the program.

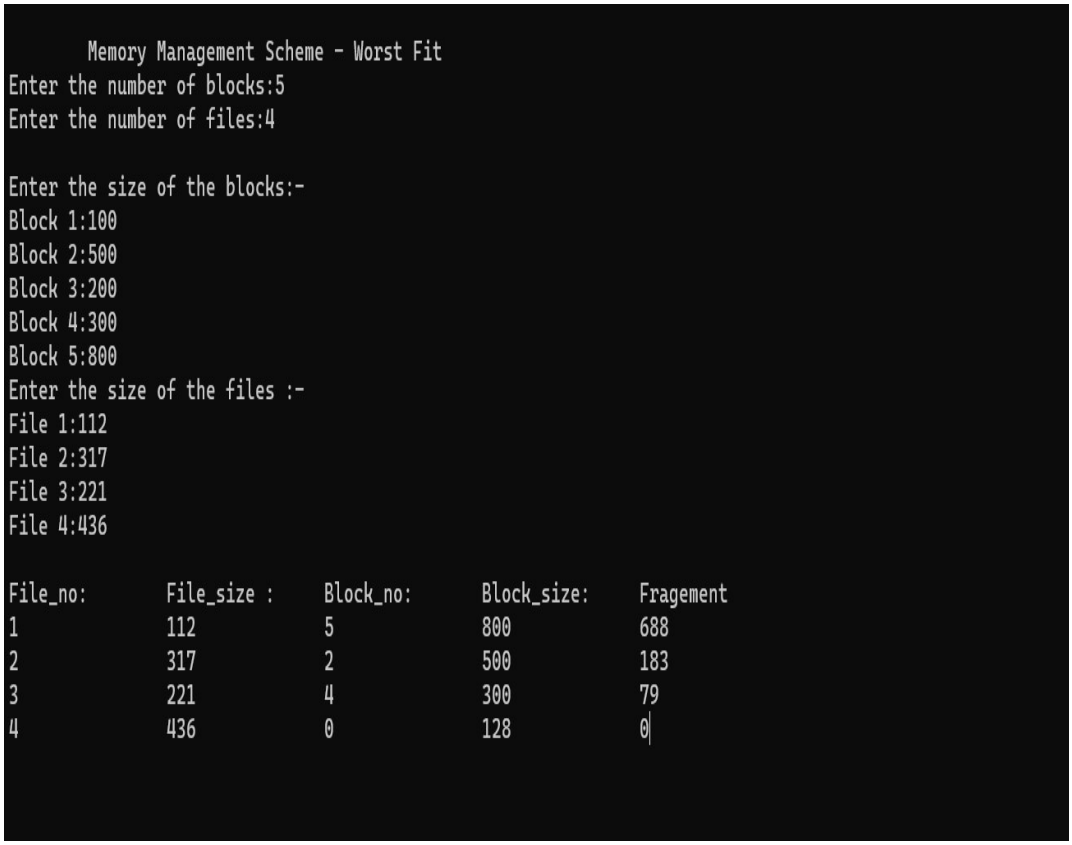
CODE:

```
#include<stdio.h>
#include<conio.h>
#define max 25
int main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
printf("\n\tMemory Management Scheme - Worst Fit");
```

```
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);

scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
```

```
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}
```



EXP. 10 CPU SCHEDULING ALGORITHMS | FCFS

AIM:

To write a C program for implementation of FCFS and SJF scheduling algorithms.

ALGORITHM:

Step 1: Inside the structure declare the variables.

Step 2: Declare the variable i,j as integer,totwtime and totttime is equal to zero.

Step 3: Get the value of „n" assign pid as I and get the value of p[i].btime.

Step 4: Assign p[0] wtime as zero and tot time as btime and inside the loop calculate wait timeand

turnaround time.

Step 5: Calculate total wait time and total turnaround time by dividing by total number ofprocess.

Step 6: Print total wait time and total turnaround time.

Step 7: Stop the program.

CODE:

```
#include<stdio.h>

int main()

{
int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
printf("Enter total number of processes(maximum 20):");
scanf("%d",&n);

printf("\nEnter Process Burst Time:\t");
```

```
for(i=0;i<n;i++)
{
printf("P[%d]:",i+1);
scanf("%d",&bt[i]);
}

wt[0]=0;

for(i=1;i<n;i++)
{
wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
}

printf("Process\t\tBurst Time\tWaiting Time\tTurnaround Time");

for(i=0;i<n;i++)
{
tat[i]=bt[i]+wt[i];
avwt+=wt[i];
avtat+=tat[i];
printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
}

avwt/=i;
avtat/=i;
printf("\nAverage Waiting Time:%d",avwt);
printf("\nAverage Turnaround Time:%d",avtat);

return 0;
}
```

```
Enter total number of processes(maximum 20):3

Enter Process Burst Time:       P[1]:24
P[2]:3
P[3]:3
Process         Burst Time      Waiting Time    Turnaround Time
P[1]            24              0               24
P[2]            3               24              27
P[3]            3               27              30
Average Waiting Time:17
Average Turnaround Time:27
--------------------------------
Process exited after 55.96 seconds with return value 0
Press any key to continue . . .
```

EXP.11 HRNN SCHEDULING

CODE:

```
#include <stdio.h>

// Defining process details
struct process {
char name;
int at, bt, ct, wt, tt;
int completed;
float ntt;
} p[10];

int n;

// Sorting Processes by Arrival Time
void sortByArrival()
{
struct process temp;
int i, j;

// Selection Sort applied
for (i = 0; i < n - 1; i++) {
for (j = i + 1; j < n; j++) {

// Check for lesser arrival time
if (p[i].at > p[j].at) {

// Swap earlier process to front
temp = p[i];
p[i] = p[j];
p[j] = temp;
}
```

```
}
}
}

int main()
{
int i, j, t, sum_bt = 0;
char c;
float avgwt = 0, avgtt = 0;
n = 5;

// predefined arrival times
int arriv[] = { 0, 2, 4, 6, 8 };

// predefined burst times
int burst[] = { 3, 6, 4, 5, 2 };

// Initializing the structure variables
for (i = 0, c = 'A'; i < n; i++, c++) {
p[i].name = c;
p[i].at = arriv[i];
p[i].bt = burst[i];

// Variable for Completion status
// Pending = 0
// Completed = 1
p[i].completed = 0;

// Variable for sum of all Burst Times
sum_bt += p[i].bt;
}

// Sorting the structure by arrival times
sortByArrival();
printf("\nName\tArrival Time\tBurst Time\tWaiting Time");
```

```
printf("\tTurnAround Time\t Normalized TT");
for (t = p[0].at; t < sum_bt;) {

// Set lower limit to response ratio
float hrr = -9999;

// Response Ratio Variable
float temp;

// Variable to store next process selected
int loc;
for (i = 0; i < n; i++) {

// Checking if process has arrived and is Incomplete
if (p[i].at <= t && p[i].completed != 1) {

// Calculating Response Ratio
temp = (p[i].bt + (t - p[i].at)) / p[i].bt;

// Checking for Highest Response Ratio
if (hrr < temp) {

// Storing Response Ratio
hrr = temp;

// Storing Location
loc = i;
}
}
}

// Updating time value
t += p[loc].bt;

// Calculation of waiting time
```

```
        p[loc].wt = t - p[loc].at - p[loc].bt;

        // Calculation of Turn Around Time
        p[loc].tt = t - p[loc].at;

        // Sum Turn Around Time for average
        avgtt += p[loc].tt;

        // Calculation of Normalized Turn Around Time
        p[loc].ntt = ((float)p[loc].tt / p[loc].bt);

        // Updating Completion Status
        p[loc].completed = 1;

        // Sum Waiting Time for average
        avgwt += p[loc].wt;
        printf("\n%c\t\t%d\t\t", p[loc].name, p[loc].at);
        printf("%d\t\t%d\t\t", p[loc].bt, p[loc].wt);
        printf("%d\t\t%f", p[loc].tt, p[loc].ntt);}
printf("\nAverage waiting time:%f\n", avgwt / n);
        printf("Average Turn Around time:%f\n", avgtt / n);}
```

```
Name    Arrival Time    Burst Time      Waiting Time    TurnAround Time  Normalized TT
A               0               3               0               3               1.000000
B               2               6               1               7               1.166667
C               4               4               5               9               2.250000
E               8               2               5               7               3.500000
D               6               5               9               14              2.800000
Average waiting time:4.000000
Average Turn Around time:8.000000

--------------------------------
Process exited after 0.05067 seconds with return value 0
Press any key to continue . . .
```

EXP. 12 CPU SCHEDULING ALGORITHM | ROUND ROBIN SCHEDULING

AIM:

To write a C program for implementation of Round Robin scheduling algorithms.

ALGORITHM:

Step 1: Inside the structure declare the variables.
Step 2: Declare the variable i,j as integer, totwtime and totttime is equal to zero.
Step 3: Get the value of „n" assign p and allocate the memory.
Step 4: Inside the for loop get the value of burst time and priority and read the time quantum.
Step 5: Assign wtime as zero.
Step 6: Check p[i].pri is greater than p[j].pri .
Step 7: Calculate the total of burst time and waiting time and assign as turnaround time.
Step 8: Stop the program.

CODE:

```
#include<stdio.h>

int main()
{
int i, limit, total = 0, x, counter = 0, time_quantum;
int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10],
temp[10];
float average_wait_time, average_turnaround_time;
printf("\nEnter Total Number of Processes:\t");
scanf("%d", &limit);
x = limit;
```

```
for(i = 0; i < limit; i++)
{
printf("\nEnter Details of Process[%d]\n", i + 1);

printf("Arrival Time:\t");

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

printf("Burst Time:\t");

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

temp[i] = burst_time[i];
}

printf("\nEnter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
for(total = 0, i = 0; x != 0;)
{
if(temp[i] <= time_quantum && temp[i] > 0)
{
total = total + temp[i];
temp[i] = 0;
counter = 1;
}
else if(temp[i] > 0)
{
temp[i] = temp[i] - time_quantum;
total = total + time_quantum;
}
if(temp[i] == 0 && counter == 1)
{
x--;
```

```
printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst_time[i], total - arrival_time[i], total - arrival_time[i] - burst_time[i]);
wait_time = wait_time + total - arrival_time[i] - burst_time[i];
turnaround_time = turnaround_time + total - arrival_time[i];
counter = 0;
}
if(i == limit - 1)
{
i = 0;
}
else if(arrival_time[i + 1] <= total)
{
i++;
}
else
{
i = 0;
}
}

average_wait_time = wait_time * 1.0 / limit;
average_turnaround_time = turnaround_time * 1.0 / limit;
printf("\n\nAverage Waiting Time:\t%f", average_wait_time);
printf("\nAvg Turnaround Time:\t%fn", average_turnaround_time);
return 0;
}
```

```
Enter Total Number of Processes:        3

Enter Details of Process[1]
Arrival Time:   0
Burst Time:     24

Enter Details of Process[2]
Arrival Time:   0
Burst Time:     3

Enter Details of Process[3]
Arrival Time:   0
Burst Time:     3

Enter Time Quantum:     4

Process ID              Burst Time       Turnaround Time         Waiting Time

Process[2]              3                7                       4
Process[3]              3                10                      7
Process[1]              24               30                      6

Average Waiting Time:   5.666667
Avg Turnaround Time:    15.666667n
--------------------------------
Process exited after 26.49 seconds with return value 0
Press any key to continue . . .
```

EXP. 13 SJF SCHEDULING

CODE:

```
#include <stdio.h>
int main()
{
// Matrix for storing Process Id, Burst
// Time, Average Waiting Time & Average
// Turn Around Time.
int A[100][4];
int i, j, n, total = 0, index, temp;
float avg_wt, avg_tat;
printf("Enter number of process: ");
scanf("%d", &n);
printf("Enter Burst Time:\n");
// User Input Burst Time and alloting Process Id.
for (i = 0; i < n; i++) {
printf("P%d: ", i + 1);
scanf("%d", &A[i][1]);
A[i][0] = i + 1;
}
// Sorting process according to their Burst Time.
for (i = 0; i < n; i++) {
index = i;
for (j = i + 1; j < n; j++)
if (A[j][1] < A[index][1])
index = j;
temp = A[i][1];
A[i][1] = A[index][1];
A[index][1] = temp;

temp = A[i][0];
A[i][0] = A[index][0];
```

```
A[index][0] = temp;
}
A[0][2] = 0;
// Calculation of Waiting Times
for (i = 1; i < n; i++) {
A[i][2] = 0;
for (j = 0; j < i; j++)
A[i][2] += A[j][1];
total += A[i][2];
}
avg_wt = (float)total / n;
total = 0;
printf("P     BT     WT     TAT\n");
// Calculation of Turn Around Time and printing the
// data.
for (i = 0; i < n; i++) {
A[i][3] = A[i][1] + A[i][2];
total += A[i][3];
printf("P%d     %d     %d      %d\n", A[i][0],
A[i][1], A[i][2], A[i][3]);
}
avg_tat = (float)total / n;
printf("Average Waiting Time= %f", avg_wt);
printf("\nAverage Turnaround Time= %f", avg_tat);
}
```

```
Enter number of process: 4
Enter Burst Time:
P1: 6
P2: 8
P3: 7
P4: 3
P     BT     WT     TAT
P4     3     0      3
P1     6     3      9
P3     7     9      16
P2     8     16      24
Average Waiting Time= 7.000000
Average Turnaround Time= 13.000000
--------------------------------
Process exited after 11.3 seconds with return value 0
Press any key to continue . . .
```

EXP. 14 DINING PHILOSOPHER

CODE:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
        {
// state that eating
state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",phnum + 1, LEFT + 1,
phnum + 1);
```

```
printf("Philosopher %d is Eating\n", phnum + 1);

// sem_post(&S[phnum]) has no effect
// during takefork
// used to wake up hungry philosophers
// during putfork
sem_post(&S[phnum]);
}
}

// take up chopsticks
void take_fork(int phnum)
{

sem_wait(&mutex);

// state that hungry
state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating
test(phnum);

sem_post(&mutex);

// if unable to eat wait to be signalled
sem_wait(&S[phnum]);

sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{
```

```
sem_wait(&mutex);

// state that thinking
state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT +
1, phnum + 1);
printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);
test(RIGHT);

sem_post(&mutex);
}

void* philosopher(void* num)
{

while (1) {

int* i = num;

sleep(1);

take_fork(*i);

sleep(0);

put_fork(*i);
}
}

int main()
{
```

```
int i;
pthread_t thread_id[N];

// initialize the semaphores
sem_init(&mutex, 0, 1);

for (i = 0; i < N; i++)

sem_init(&S[i], 0, 0);

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

// create philosopher processes
pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);
}

for (i = 0; i < N; i++)

pthread_join(thread_id[i], NULL);
}
```

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 2 is Hungry
Philosopher 4 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 3 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
```

EXP. 15 PRODUCER CONSUMER PROBLEM USING SEMAPHORES

AIM:

To write a C-program to implement the producer – consumer problem using semaphores.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the required variables.
Step 3: Initialize the buffer size and get maximum item you want to produce.
Step 4: Get the option, which you want to do either producer, consumer or exit from the
operation.
Step 5: If you select the producer, check the buffer size if it is full the producer should not
produce the item or otherwise produce the item and increase the value buffer size.
Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not
consume the item or otherwise consume the item and decrease the value of buffer size.
Step 7: If you select exit come out of the program.
Step 8: Stop the program.

CODE:

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

```
int signal(int);
printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
while(1) {
printf("\nENTER YOUR CHOICE\n");
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; }}}
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 producesthe item%d",x);
mutex=signal(mutex);
```

```
}
void consumer() {
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\n consumer consumes item%d",x);
x--;
mutex=signal(mutex);
}
```

```
1.PRODUCER
2.CONSUMER
3.EXIT

ENTER YOUR CHOICE
1

producer producesthe item1
ENTER YOUR CHOICE
2

 consumer consumes item1
ENTER YOUR CHOICE
1

producer producesthe item1
ENTER YOUR CHOICE
1

producer producesthe item2
ENTER YOUR CHOICE
1

producer producesthe item3
ENTER YOUR CHOICE
1
BUFFER IS FULL
ENTER YOUR CHOICE
```

EXP. 16 SLEEPING-BARBER PROBLEM

CODE:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>
#include <sys/types.h>
#include <sys/time.h>

void *barber_function(void *idp);
void *customer_function(void *idp);
void serve_customer();
void *make_customer_function();

/* Mutex */
pthread_mutex_t srvCust;

/* Semaphores */
sem_t barber_ready;
sem_t customer_ready;
sem_t modifySeats;

/* Inputs */
int chair_cnt;
int total_custs;

int available_seats;
int no_served_custs = 0;
time_t waiting_time_sum;
```

```
void *barber_function(void *idp)
{
int counter = 0;

while (1)
{
/* Lock semaphore "customer_ready" - try to get a customer or sleep if
there is none */
sem_wait(&customer_ready);

/* Lock semaphore "modifySeats" - try to get access to seats */
sem_wait(&modifySeats);

/* Increment by 1 the available seats */
available_seats++;

/* Unlock semaphore "modifySeats" */
sem_post(&modifySeats);

/* Unlock semaphore "barber_ready" - set barber ready to serve */
sem_post(&barber_ready);

/* Lock mutex "srvCust" - protect service by the same barber from other
threads */
pthread_mutex_lock(&srvCust);

/* Serve customer */
serve_customer();

/* Unlock mutex "srvCust" - finished service */
pthread_mutex_unlock(&srvCust);

printf("Customer was served.\n");

counter++;
```

```
if (counter == (total_custs - no_served_custs))
break;
}
pthread_exit(NULL);
}

void *customer_function(void *idp)
{
struct timeval start, stop;

/* Lock semaphore "modifySeats" */
sem_wait(&modifySeats);

/* If there is available seat */
if (available_seats >= 1)
{
/* Occupy a seat */
available_seats--;

printf("Customer[pid = %lu] is waiting.\n", pthread_self());
printf("Available seats: %d\n", available_seats);

/* Start waiting-time counter */
gettimeofday(&start, NULL);

/* Unlock semaphore "customer_ready" - set the customer ready to be
served */
sem_post(&customer_ready);

/* Unlock semaphore "modifySeats" */
sem_post(&modifySeats);

/* Lock semaphore "barber_ready" - wait for barber to get ready */
sem_wait(&barber_ready);
```

```
/* Stop waiting-time counter */
gettimeofday(&stop, NULL);

double sec = (double)(stop.tv_usec - start.tv_usec) / 1000000 +
(double)(stop.tv_sec - start.tv_sec);

/* Assign the time spent to global variable (ms) */
waiting_time_sum += 1000 * sec;
printf("Customer[pid = %lu] is being served. \n", pthread_self());
}
else
{
/* Unlock semaphore "modifySeats" */
sem_post(&modifySeats);
no_served_custs++;
printf("A Customer left.\n");
}

pthread_exit(NULL);
}

void serve_customer() {
/* Random number between 0 and 400 (miliseconds) */
int s = rand() % 401;

/* Convert miliseconds to microseconds */
s = s * 1000;
usleep(s);
}

void *make_customer_function() {
int tmp;
int counter = 0;

while (counter < total_custs)
```

```
{
/* Declare and create a customer thread */
pthread_t customer_thread;

tmp = pthread_create(&customer_thread, NULL, (void
*)customer_function, NULL);

if (tmp)
printf("Failed to create thread.");

/* Increment the counter */
counter++;

/* Sleep for 100ms before creating another customer */
usleep(100000);
}
}

int main() {
/* Initialization, should only be called once */
srand(time(NULL));

/* Barber 1 thread */
pthread_t barber_1;

/* Thread that creates customers */
pthread_t customer_maker;

int tmp;

/* Initialize mutex */
pthread_mutex_init(&srvCust, NULL);

/* Initialize semaphores */
sem_init(&customer_ready, 0, 0);
```

```
sem_init(&barber_ready, 0, 0);
sem_init(&modifySeats, 0, 1);

printf("Please enter the number of seats: \n");
scanf("%d", &chair_cnt);

printf("Please enter the total customers: \n");
scanf("%d", &total_custs);

available_seats = chair_cnt;

/* Create barber thread */
tmp = pthread_create(&barber_1, NULL, (void *)barber_function, NULL);

if (tmp)
printf("Failed to create thread.");

/* Create customer_maker thread */
tmp = pthread_create(&customer_maker, NULL, (void
*)make_customer_function, NULL);

if (tmp)
printf("Failed to create thread.");

/* Wait for threads to finish */
pthread_join(barber_1, NULL);
pthread_join(customer_maker, NULL);

printf("\n-----------------------------------------------\n");
printf("Average customers' waiting time: %f ms.\n", (waiting_time_sum /
(double) (total_custs - no_served_custs)));
printf("Number of customers that were forced to leave: %d\n",
no_served_custs);
}
```

```
Please enter the number of seats:
5
Please enter the total customers:
6
Customer[pid = 4] is waiting.
Available seats: 4
Customer[pid = 4] is being served.
Customer was served.
Customer[pid = 5] is waiting.
Available seats: 4
Customer[pid = 5] is being served.
Customer was served.
Customer[pid = 6] is waiting.
Available seats: 4
Customer[pid = 6] is being served.
Customer[pid = 7] is waiting.
Available seats: 4
Customer[pid = 8] is waiting.
Available seats: 3
Customer was served.
Customer[pid = 7] is being served.
Customer[pid = 9] is waiting.
Available seats: 3
Customer was served.
Customer[pid = 8] is being served.
Customer was served.
Customer[pid = 9] is being served.
Customer was served.

------------------------------------------------
Average customers' waiting time: 124.000000 ms.
Number of customers that were forced to leave: 0

--------------------------------
Process exited after 9.448 seconds with return value 49
Press any key to continue . . .
```

EXP. 17 PAGE REPLACEMENT ALGORITHMS | FIFO

AIM:

To write a C program for implementation of FIFO page replacement algorithm.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the necessary variables.
Step 3: Enter the number of frames.
Step 4: Enter the reference string end with zero.
Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen.
Step 6: When a page is brought into memory, it is inserted at the tail of the queue.
Step 7: Initially all the three frames are empty.
Step 8: The page fault range increases as the no of allocated frames also increases.
Step 9: Print the total number of page faults.
Step 10: Stop the program.

CODE:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int incomingStream[] = {4 , 1 , 2 , 4 , 5};
int pageFaults = 0;
int frames = 3;
int m, n, s, pages;
pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
printf(" Incoming \t\t Frame 1 \t\t Frame 2 \t\t Frame 3 ");
int temp[ frames ];
for(m = 0; m < frames; m++)
```

```
{
temp[m] = -1;
}
for(m = 0; m < pages; m++)
{
s = 0;
for(n = 0; n < frames; n++)
{
if(incomingStream[m] == temp[n])
{
s++;
pageFaults--;
}
}
pageFaults++;
if((pageFaults <= frames) && (s == 0))
{
temp[m] = incomingStream[m];
}
else if(s == 0)
{
temp[(pageFaults - 1) % frames] = incomingStream[m];
}
printf("\n");
printf("%d\t\t\t",incomingStream[m]);
for(n = 0; n < frames; n++)
{
if(temp[n] != -1)
printf(" %d\t\t\t", temp[n]);
else
printf(" - \t\t\t");
}
}
printf("\nTotal Page Faults:\t%d\n", pageFaults);
return 0;
}
```

```
 Incoming               Frame 1                Frame 2                Frame 3
4                       4                      -                      -
1                       4                      1                      -
2                       4                      1                      2
4                       4                      1                      2
5                       5                      1                      2
Total Page Faults:      4

--------------------------------
Process exited after 4.89 seconds with return value 0
Press any key to continue . . .
```