ALIGARH COLLEGE OF ENGINEERING AND TECHNOLOGY, ALIGARH

(APPROVED BY A.I.C.T.E. & AFFILIATED TO A.K.T.U. LUCKNOW)

SESSION: 2020-21

PRACTICAL FILE

OF

OPERATING SYSTEM(KCS-451)

BACHELOR OF TECHNOLOGY

BRANCH: COMPUTER SCIENCE AND ENGINEERING

YEAR: SECOND YEAR

SUBMISSION DATE: 21/08/2021

SUBMITTED TO:

MR.KAPIL ARORA

SUBMITTED BY:

SHIVAM SHARMA(1901090100062)

EXPERIMENT 1

Write a program for the following CPU scheduling:

A). FIRST COME FIRST SERVE:

AIM: To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

ALGORITHM:

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 name and the burst time Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time Step 5: for each process in the Ready Q calculate

a). Waiting time (n) = waiting time (n-1) + Burst time (n-1) b).

Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate

a) Average waiting time = Total waiting Time / Number of process

b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
```

```
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

INPUT

Enter the number of processes -- 3

Enter Burst Time for Process 0 -- 24

Enter Burst Time for Process 1 -- 3

Enter Burst Time for Process 2 -- 3

OUTPUT

PROCESS BURST TIME WAITING TIME TURNAROUND

TIME

P0 24 0 24

P1 3 24 27

P2 3 27 30

Average Waiting Time-- 17.000000

Average Turnaround Time -- 27.000000

B). SHORTEST JOB FIRST:

AIM: To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

DESCRIPTION:

To calculate the average waiting time in the shortest job first algorithm the sorting of

the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:

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: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue,

calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

c) Average waiting time = Total waiting Time / Number of process

d) Average Turnaround time = Total Turnaround Time / Number of process

Step 9: Stop the process

SOURCE CODE :

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

```
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
```

```
wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)

{

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

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

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

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n); getch();}
```


INPUT

Enter the number of processes -- 4

Enter Burst Time for Process 0 -- 6

Enter Burst Time for Process 1 -- 8

Enter Burst Time for Process 2 -- 7

Enter Burst Time for Process 3 -- 3

OUTPUT

PROCESS BURST

TIME

WAITING

TIME

TURNARO

UND TIME

P3 3 0 3

P0 6 3 9

P2 7 9 16

P1 8 16 24

Average Waiting Time -- 7.000000

Average Turnaround Time -- 13.000000

C). ROUND ROBIN:

AIM: To simulate the CPU scheduling algorithm round-robin.

DESCRIPTION:

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)

b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from 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

SOURCE CODE

```
#include<stdio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
```

```
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<(max/t)+1;j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t) {
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else {
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++){
wa[i]=tat[i]-
ct[i]; att+=tat[i];
awt+=wa[i];}
```

```
printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is -- %f ",awt/n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

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

printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);

getch();}
```

INPUT:

Enter the no of processes – 3

Enter Burst Time for process 1 – 24

Enter Burst Time for process 2 -- 3

Enter Burst Time for process 3 – 3

Enter the size of time slice – 3

OUTPUT:

PROCESS BURST TIME WAITING TIME TURNAROUNDTIME

1 24 6 30

2 3 4 7

3 3 7 10

The Average Turnaround time is – 15.666667 The

Average Waiting time is ---------- 5.666667

D). PRIORITY SCHEDULING:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION:

To calculate the average waiting time in the priority algorithm, sort the burst

times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

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: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround time

Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q calculate Step 8:

for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate

c) Average waiting time = Total waiting Time / Number of process

d) Average Turnaround time = Total Turnaround Time / Number of process Print the results in an order.

Step10: Stop

SOURCE CODE:

```
#include<stdio.h>
main()
```

```
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++){
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d
%d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k]){
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
```

```
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n); printf("\nAverage
Turnaround Time is --- %f",tatavg/n);
getch();}
```

EXPERIMENT 2

AIM: To Write a C program to simulate producer-consumer problem using semaphores.

DESCRIPTION

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the

producer produces items and that is consumed by a consumer process. One solution to the producerconsumer

problem uses shared memory. To allow producer and consumer processes to run

concurrently, there must be available a buffer of items that can be filled by the producer and emptied

by the consumer. This buffer will reside in a region of memory that is shared by the producer and

consumer processes. The producer and consumer must be synchronized, so that the consumer does

not try to consume an item that has not yet been produced.

PROGRAM

```
#include<stdio.>

void main()

{

int buffer[10], bufsize, in, out, produce, consume,

choice=0; in = 0;

out = 0;

bufsize = 10;

while(choice !=3)

{
```

```
printf(“\n1. Produce \t 2. Consume \t3. Exit”);
printf(“\nEnter your choice: ”);
scanf(“%d”,&choice);
switch(choice) {
case 1: if((in+1)%bufsize==out)
printf(“\nBuffer is Full”);
else
{
}
break;;;
printf(“\nEnter the value: “);
scanf(“%d”, &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
case 2: if(in == out)
printf(“\nBuffer is Empty”);
else
{
consume = buffer[out];
printf(“\nThe consumed value is %d”, consume);
out = (out+1)%bufsize;
}
break;
} } }
```

OUTPUT

1. Produce 2. Consume 3. Exit

Enter your choice: 2

Buffer is Empty

1. Produce 2. Consume 3. Exit

Enter your choice: 1

Enter the value: 100

1. Produce 2. Consume 3. Exit

Enter your choice: 2

The consumed value is 100

1. Produce 2. Consume 3. Exit

Enter your choice: 3

EXPERIMENT 3

AIM: To Write a C program to simulate the concept of Dining-Philosophers problem.

DESCRIPTION

The dining-philosophers problem is considered a classic synchronization problem because it is an example

of a large class of concurrency-control problems. It is a simple representation of the need to allocate several

resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers

who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs,

each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five

single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a

philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are

between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time.

Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry

philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is

finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers

problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of

deadlock.

PROGRAM

```
int tph, philname[20], status[20], howhung, hu[20], cho; main()
{
int i; clrscr();
printf("\n\nDINING PHILOSOPHER PROBLEM");
printf("\nEnter the total no. of philosophers: ");
scanf("%d",&tph);
for(i=0;i<tph;i++)
{
philname[i]=(i+1); status[i]=1;
}
printf("How many are hungry : ");
scanf("%d", &howhung);
if(howhung==tph)
{
printf(“\n All are hungry..\nDead lock stage will occur”);
printf(\n”Exiting\n”);
else{
for(i=0;i<howhung;i++){
printf(“Enterphilosopher%dposition:”,(i+1));
scanf(“%d”,&hu[i]);
status[hu[i]]=2;
}
```



```
do
```

```
{
printf("1.One can eat at a time\t2.Two can eat at a time
\t3.Exit\nEnter your choice:");
scanf("%d", &cho);
switch(cho)
{
case 1: one();
break;
case 2: two();
break;
case 3: exit(0);
default: printf("\nInvalid option..");
}
}while(1);
}
}
one()
{
int pos=0, x, i;
printf("\nAllow one philosopher to eat at any time\n");
for(i=0;i<howhung; i++, pos++)
{
printf("\nP %d is granted to eat", philname[hu[pos]]);
for(x=pos;x<howhung;x++)
printf("\nP %d is waiting", philname[hu[x]]);
```

```
}
}
two()
{
int i, j, s=0, t, r, x;
printf("\n Allow two philosophers to eat at same
time\n"); for(i=0;i<howhung;i++)
{
for(j=i+1;j<howhung;j++)
{
if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
{
printf("\n\ncombination %d \n", (s+1));
t=hu[i];
r=hu[j]; s++;
printf("\nP %d and P %d are granted to eat", philname[hu[i]],
philname[hu[j]]);
```

```
for(x=0;x<howhung;x++)
{
if((hu[x]!=t)&&(hu[x]!=r))
printf("\nP %d is waiting", philname[hu[x]]);
}
}
}
```

}

}

INPUT

DINING PHILOSOPHER PROBLEM

Enter the total no. of philosophers: 5

How many are hungry : 3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5

OUTPUT

1. One can eat at a time 2.Two can

eat at a time 3.Exit Enter your choice: 1

Allow one philosopher to eat at any time

P 3 is granted to eat

P 3 is waiting

P 5 is waiting

P 0 is waiting

P 5 is granted to eat

P 5 is waiting

P 0 is waiting

P 0 is granted to eat

P 0 is waiting

1.One can eat at a time 2.Two can eat at a time 3.Exit

Enter your choice: 2

Allow two philosophers to eat at same time

combination 1

P 3 and P 5 are granted to eat

P 0 is waiting

combination 2

P 3 and P 0 are granted to eat

P 5 is waiting

combination 3

P 5 and P 0 are granted to eat

P 3 is waiting

1.One can eat at a time 2.Two can

eat at a time 3.Exit Enter your choice: 3

EXPERIMENT 4

MEMORY ALLOCATION TECHNIQUES

AIM: To Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixed-sized

partitions. Each partition may contain exactly one process. In this multiple-partition method, when a

partition is free, a process is selected from the input queue and is loaded into the free partition. When the

process terminates, the partition becomes available for another process. The operating system keeps a

table indicating which parts of memory are available and which are occupied. Finally, when a process

arrives and needs memory, a memory section large enough for this process is provided. When it is time to

load or swap a process into main memory, and if there is more than one free block of memory of

sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy

chooses the block that is closest in size to the request. First-fit chooses the first available block that is

large enough. Worst-fit chooses the largest available block.

PROGRAM

WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int
frag[max],b[max],f[max],i,j,nb,nf,t
emp; static int bf[max],ff[max];
clrscr();
printf("\n\tMemory Management Scheme - First 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)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
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();
}
```

INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

OUTPUT

File No File Size Block No Block Size Fragment

1 1 1 5 4

2 4 3 7 3

BEST-FIT

```
#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];
clrscr();
```

```
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();
}
```

INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

OUTPUT

File No File Size

Block No

Block Size

Fragment

1 1 2 2 1

2 4 1 5 1

FIRST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int
frag[max],b[max],f[max],i,j,nb,nf,temp,highes
t=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);
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)
{
}
}
frag[i]=highest; bf[ff[i]]=1; highest=0;
}
ff[i]=j; highest=temp;
}
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();
}
```

INPUT

```
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7
Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
File No File Size Block No Block Size Fragment
1 1 3 7 6
2 4 1 5 1
```

EXPERIMENT 5

PAGE REPLACEMENT ALGORITHMS

AIM: To implement FIFO page replacement technique.

a) FIFO b) LRU c) OPTIMAL

DESCRIPTION:

Page replacement algorithms are an important part of virtual memory management and it helps the OS to

decide which memory page can be moved out making space for the currently needed page. However, the

ultimate objective of all page replacement algorithms is to reduce the number of page faults.

FIFO-This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track

of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be

replaced page in the front of the queue is selected for removal.

LRU-In this algorithm page will be replaced which is least recently used

OPTIMAL- In this algorithm, pages are replaced which would not be used for the longest duration of time

in the future. This algorithm will give us less page fault when compared to other page replacement

algorithms.

ALGORITHM:

1. Start the process

2. Read number of pages n

3. Read number of pages no

4. Read page numbers into an array a[i]

5. Initialize avail[i]=0 .to check page hit

6. Replace the page with circular queue, while re-placing check page availability in the frame

Place avail[i]=1 if page is placed in the frame Count page faults

7. Print the results.

8. Stop the process.

A) FIRST IN FIRST OUT

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h> int fr[3];
void main()
{
void display();
int i,j,page[12]={2,3,2,1,5,2,4,5,3,2,5,2};
int
flag1=0,flag2=0,pf=0,frsize=3,top=0;
clrscr();
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
```

```
flag1=0; flag2=0; for(i=0;i<12;i++)
{
if(fr[i]==page[j])
{
flag1=1; flag2=1; break;
}
}
if(flag1==0)
{
for(i=0;i<frsize;i++)
{
if(fr[i]==-1)
{
fr[i]=page[j]; flag2=1; break;
}
}
}
if(flag2==0)
{
fr[top]=page[j];
top++;
pf++;
if(top>=frsize)
top=0;
}
```

```
display();
}
```

```
printf("Number of page faults : %d ",pf+frsize);
getch();
}
void display()
{
int i; printf("\n");
for(i=0;i<3;i++)
printf("%d\t",fr[i]);
}
```
OUTPUT:

2 -1 -1

2 3 -1

2 3 -1

2 3 1

5 3 1

5 2 1

5 2 4

5 2 4

3 2 4

3 2 4

3 5 4

3 5 2

Number of page faults: 9

B) LEAST RECENTLY USED

AIM: To implement LRU page replacement technique.

ALGORITHM:

1. Start the process

2. Declare the size

3. Get the number of pages to be inserted

4. Get the value

5. Declare counter and stack

6. Select the least recently used page by counter value

7. Stack them according the selection.

8. Display the values

9. Stop the process

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
int fr[3];
void main()
{
void display();
int p[12]={2,3,2,1,5,2,4,5,3,2,5,2},i,j,fs[3];
```

```
int index,k,l,flag1=0,flag2=0,pf=0,frsize=3;
clrscr();
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
flag1=0,flag2=0;
for(i=0;i<3;i++)
{
if(fr[i]==p[j])
{
flag1=1;
flag2=1; break;
}
}
if(flag1==0)
```



```
{
for(i=0;i<3;i++)
{
if(fr[i]==-1)
{
fr[i]=p[j]; flag2=1;
```

```
break;
}
}
}
if(flag2==0)
{
for(i=0;i<3;i++)
fs[i]=0;
for(k=j-1,l=1;l<=frsize-1;l++,k--)
{
for(i=0;i<3;i++)
{
if(fr[i]==p[k]) fs[i]=1;
}}
for(i=0;i<3;i++)
{
if(fs[i]==0)
index=i;
}
fr[index]=p[j];
pf++;
}
display();
}
printf("\n no of page faults :%d",pf+frsize);
```

```
getch();
}
void display()
{
int i; printf("\n");
for(i=0;i<3;i++)
printf("\t%d",fr[i]);
}
```


OUTPUT:

2 -1 -1

2 3 -1

2 3 -1

2 3 1

2 5 1

2 5 1

2 5 4

2 5 4

3 5 4

3 5 2

3 5 2

3 5 2

No of page faults: 7

C) OPTIMAL

AIM: To implement optimal page replacement technique.

ALGORTHIM:

1. Start Program

2. Read Number Of Pages And Frames

3.Read Each Page Value

4. Search For Page In The Frames

5.If Not Available Allocate Free Frame

6. If No Frames Is Free Repalce The Page With The Page That Is Leastly Used

7.Print Page Number Of Page Faults

8.Stop process.

SOURCE CODE:

```
/* Program to simulate optimal page replacement */
#include<stdio.h>
#include<conio.h>
int fr[3], n, m;
void
display();
void main()
{
int i,j,page[20],fs[10];
int
max,found=0,lg[3],index,k,l,flag1=0,flag2=0,pf=0;
```

```
float pr;
clrscr();
printf("Enter length of the reference string: ");
scanf("%d",&n);
printf("Enter the reference string: ");
for(i=0;i<n;i++)
scanf("%d",&page[i]);
printf("Enter no of frames: ");
scanf("%d",&m);
for(i=0;i<m;i++)
fr[i]=-1; pf=m;
for(j=0;j<n;j++)
{
flag1=0; flag2=0;
for(i=0;i<m;i++)
{
if(fr[i]==page[j])
{
flag1=1; flag2=1;
break;
}
}
if(flag1==0)
{
for(i=0;i<m;i++)
```

```
{
if(fr[i]==-1)
{
fr[i]=page[j]; flag2=1;
break;
}
}
}
if(flag2==0)
{
for(i=0;i<m;i++)
lg[i]=0;
for(i=0;i<m;i++)
{
for(k=j+1;k<=n;k++)
{
if(fr[i]==page[k])
{
lg[i]=k-j;
break;
}
}
}
found=0;
for(i=0;i<m;i++)
```

```
{
if(lg[i]==0)
{
index=i;
found = 1;
break;
}
}
if(found==0)
{
max=lg[0]; index=0;
for(i=0;i<m;i++)
{
if(max<lg[i])
{
max=lg[i];
index=i;
}
}
}
fr[index]=page[j];
pf++;
}
display();
}
```

```
printf("Number of page faults : %d\n", pf);

pr=(float)pf/n*100;

printf("Page fault rate = %f \n", pr); getch();

}

void display()

{

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

printf("%d\t",fr[i]);

printf("\n");

}
```


OUTPUT:

```
Enter length of the reference string: 12

Enter the reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Enter no of frames: 3

1 -1 -1

1 2 -1

1 2 3

1 2 4

1 2 4

1 2 4

1 2 5

1 2 5

1 2 5

3 2 5
```

4 2 5

4 2 5

Number of page faults : 7

Page fault rate = 58.333332