# DELHI TECHNOLOGICAL UNIVERSITY

**OPERATING SYSTEMS LAB FILE**

**(CO - 204)**

****

**SUBMITTED BY: SUBMITTED TO:**

Ritik Singh Proff. Daya Gupta

2K19/CO/319

**DELHI TECHNOLOGICAL UNIVERSITY**

**(Formerly Delhi College of Engineering)**

**Bawana Road, Delhi – 110042**

**EXPERIMENT - 1**

**Aim:**

Write a C program to check the process ids of the program and parent of the program process

1. Use the system command to execute ps command to get all the processes under the current user.
2. Use getpid() and getppid() to get the program process and parent process ids.Check if the program process id and parent process ids are listed in the ps command.

Extend this to fork 5 child processes.parent must wait for all children to exit.

**Theory:**

A process is a program in execution.

A program is a file containing the information of a process and how to build it during run time. When you start execution of the program, it is loaded into RAM and starts executing.

Each process is identified with a unique positive integer called as process ID or simply PID (Process Identification number). The kernel usually limits the process ID to 32767, which is configurable. When the process ID reaches this limit, it is reset again, which is after the system processes range. The unused process IDs from that counter are then assigned to newly created processes.

The system call getpid() returns the process ID of the calling process.

pid\_t getpid(void);

This call returns the process ID of the calling process which is guaranteed to be unique. This call is always successful and thus no return value to indicate an error.

Each process has its unique ID called process ID that is fine but who created it? How to get information about its creator? Creator process is called the parent process. Parent ID or PPID can be obtained through getppid() call.

The system call getppid() returns the Parent PID of the calling process.

pid\_t getppid(void);

This call returns the parent process ID of the calling process. This call is always successful and thus no return value to indicate an error.

 The “C” library function system() executes a shell command. The arguments passed to system() are commands executed on shell. In the above program, command is “ps”, which gives process status.

The complete information about all running processes and other system related information are accessible from proc file system available at /proc location.

fork() is a system call used to create a new process. The new process is called a child process and the original process is called the parent process. The child process by default is a duplicate of the parent process. By duplicate we mean that the child process has the same code as the parent process but the memory space of both the processes is separate. The syntax of using fork() is :

pid\_t fork(void);

Working of fork() system call:

* After compiling the program with gcc it creates an output file “a.out”. The moment you run a.out using the command, ./a.out, a new process is created (parent).  This process gets the process id (PID) 27.
* The PID will differ from system to system and each time you run the program. The process starts to run and it prints before fork. Next it executes the fork() system call. If it gets executed a child process is created having a different PID.
* Now there are two process in the system both having the same code to run. But since the code has been run till this line the execution will continue from the next line in both the process. fork() on success returns either 0 or a non-zero value.
* Since, the same code is both the processes the variable ‘p’ will have some value in both the process. In the parent process it gets a non-zero positive value (which actually is the PID of the child). In the child process ‘p’ gets the value ‘0’.

**Implementation:**

**1.**

#include <stdio.h>

#include <unistd.h>

int main()

{

int p\_id,p\_pid;

p\_id=getpid(); /\*process id\*/

p\_pid=getppid(); /\*parent process id\*/

printf("Process ID: %d\n",p\_id);

printf("Parent Process ID: %d\n",p\_pid);

return 0;

}

**2.**

#include <stdio.h>

#include <unistd.h>

#include<stdlib.h>

int main()

{

int p\_id,p\_pid;

for (int id=0; id<5; id++) {

if (fork()==0) {

p\_id=getpid();

p\_pid=getppid();

printf("Son Process ID: %d and Parent Process ID : %d\n", p\_id,p\_pid);

exit(0);

}

}

for (int i = 0; i < 5; i++)

{

wait(NULL);

}

printf("Parent ID : %d ", p\_id);

return 0;

}

**Output:**

**EXPERIMENT - 1**

**Aim:**

To write a C program for simulation of the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time for a given problem a) FCFS b) SJF

**Introduction:**

CPU Scheduling:

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

There are mainly two types of CPU Scheduling:

Non-Preemptive Scheduling:

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.This scheduling method is used by Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

Preemptive Scheduling:

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

The following are two Non-Preemptive Scheduling which we will simulate in this lab experiment:

FCFS

First Come First Serve: Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-preemptive scheduling algorithm.

SJF

Shortest Job First: Process which have the shortest burst time are scheduled first.If two processes have the same bust time then FCFS is used to break the tie. It is the best approach to minimize waiting time. It can only be implemented where required CPU time and job execution time is known in advance.

**Algorithms:**

FCFS

1- We input the processes along with their burst time(bt)

and arrival time(at)

2- We find the waiting time for all other processes i.e. for

a given process i:

wt[i] = (bt[0] + bt[1] +...... bt[i-1]) - at[i]

3- We now find turn around time

= waiting\_time + burst\_time for all processes

4- We find the average waiting time: Average waiting time = total\_waiting\_time / no\_of\_processes

5- We also find the average turn around time: Average turn around time = total\_turn\_around\_time / no\_of\_processes

SJF

1- We input the processes along with their burst time(bt)

and arrival time(at).

2- We sort all the processes according to the arrival time.

3- We then select that process which has minimum arrival time and minimum Burst time.

4- After completion of process we make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

5- Completion Time: Time at which process completes its execution.

6- Turnaround Time = Completion Time – Arrival Time

7- Waiting Time = Turnaround Time – Burst Time

**Implementation:**

**A) FCFS**

#include<iostream>

using namespace std;

void finding\_wait\_time(int processes[], int n, int bt[],int wt[], int at[])

{

int service\_time[n];

service\_time[0] = 0;

wt[0] = 0;

for (int i = 1; i < n ; i++)

{

service\_time[i] = service\_time[i-1] + bt[i-1];

wt[i] = service\_time[i] - at[i];

if (wt[i] < 0)

wt[i] = 0;

}

}

void finding\_turnaround\_time(int processes[], int n, int bt[],int wt[], int tat[])

{

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

tat[i] = bt[i] + wt[i];

}

void finding\_average\_time(int processes[], int n, int bt[], int at[])

{

int wt[n], tat[n];

finding\_wait\_time(processes, n, bt, wt, at);

finding\_turnaround\_time(processes, n, bt, wt, tat);

cout << "Processes " << " Burst Time " << " Arrival Time "

<< " Waiting Time " << " Turn-Around Time "

<< " Completion Time \n";

int total\_wt = 0, total\_tat = 0;

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

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

int compl\_time = tat[i] + at[i];

cout << " " << i+1 << "\t\t" << bt[i] << "\t\t"

<< at[i] << "\t\t" << wt[i] << "\t\t "

<< tat[i] << "\t\t " << compl\_time << endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

int processes[] = {1, 2, 3};

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {5, 9, 6};

int arrival\_time[] = {0, 3, 6};

finding\_average\_time(processes, n, burst\_time, arrival\_time);

return 0;

}

B) SJF

#include<iostream>

using namespace std;

int process\_det[10][6];

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void arranging\_arrival\_time(int num, int process\_det[][6])

{

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

{

for(int j=0; j<num-i-1; j++)

{

if(process\_det[j][1] > process\_det[j+1][1])

{

for(int k=0; k<5; k++)

{

swap(process\_det[j][k], process\_det[j+1][k]);

}

}

}

}

}

void time\_completion(int num, int process\_det[][6])

{

int temp, val;

process\_det[0][3] = process\_det[0][1] + process\_det[0][2];

process\_det[0][5] = process\_det[0][3] - process\_det[0][1];

process\_det[0][4] = process\_det[0][5] - process\_det[0][2];

for(int i=1; i<num; i++)

{

temp = process\_det[i-1][3];

int low = process\_det[i][2];

for(int j=i; j<num; j++)

{

if(temp >= process\_det[j][1] && low >= process\_det[j][2])

{

low = process\_det[j][2];

val = j;

}

}

process\_det[val][3] = temp + process\_det[val][2];

process\_det[val][5] = process\_det[val][3] - process\_det[val][1];

process\_det[val][4] = process\_det[val][5] - process\_det[val][2];

for(int k=0; k<6; k++)

{

swap(process\_det[val][k], process\_det[i][k]);

}

}

}

int main()

{

int num, temp;

cout<<"Enter total number of Processes: ";

cin>>num;

cout<<"Enter the process ID:\n";

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

{

cout<<"Process "<<i+1<<"\n";

cout<<"Enter Process Id: ";

cin>>process\_det[i][0];

cout<<"Enter Arrival Time: ";

cin>>process\_det[i][1];

cout<<"Enter Burst Time: ";

cin>>process\_det[i][2];

}

arranging\_arrival\_time(num, process\_det);

time\_completion(num, process\_det);

cout<<"Final Result\n";

cout<<"Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";

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

{

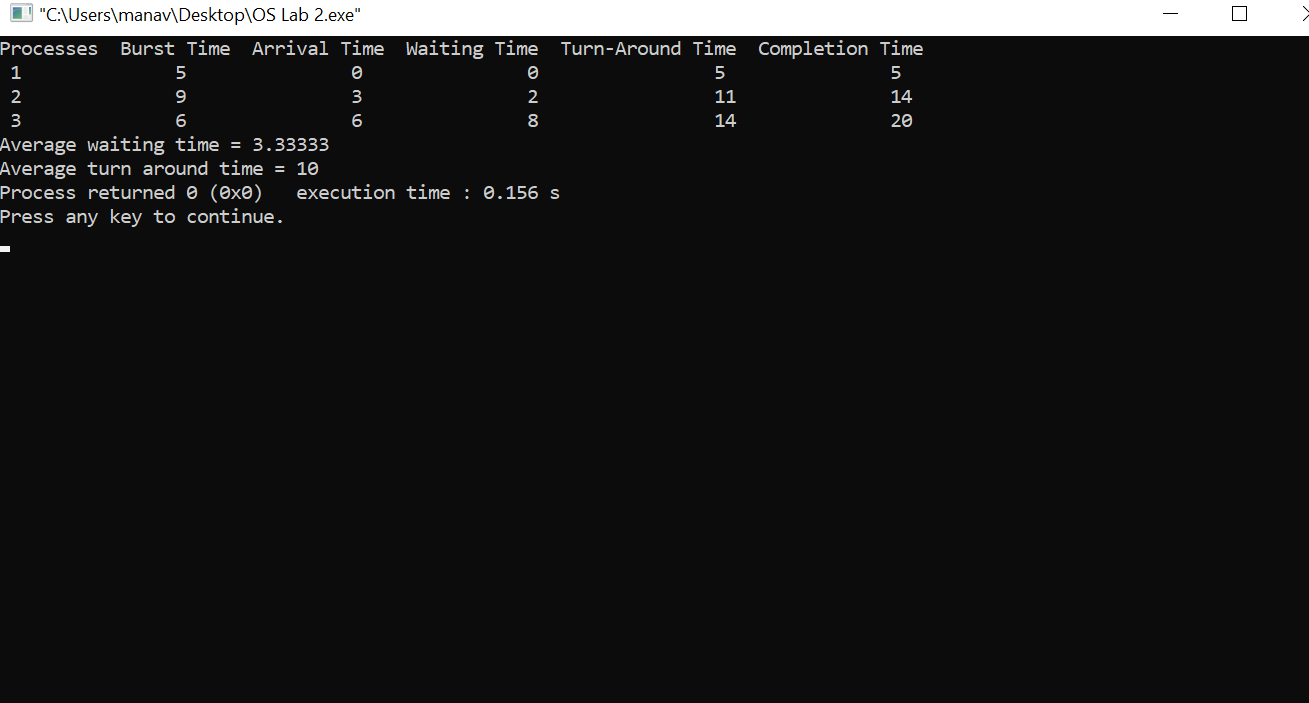
cout<<process\_det[i][0]<<"\t\t"<<process\_det[i][1]<<"\t\t"<<process\_det[i][2]<<"\t\t"<<process\_det[i][4]<<"\t\t"<<process\_det[i][5]<<"\n";

}

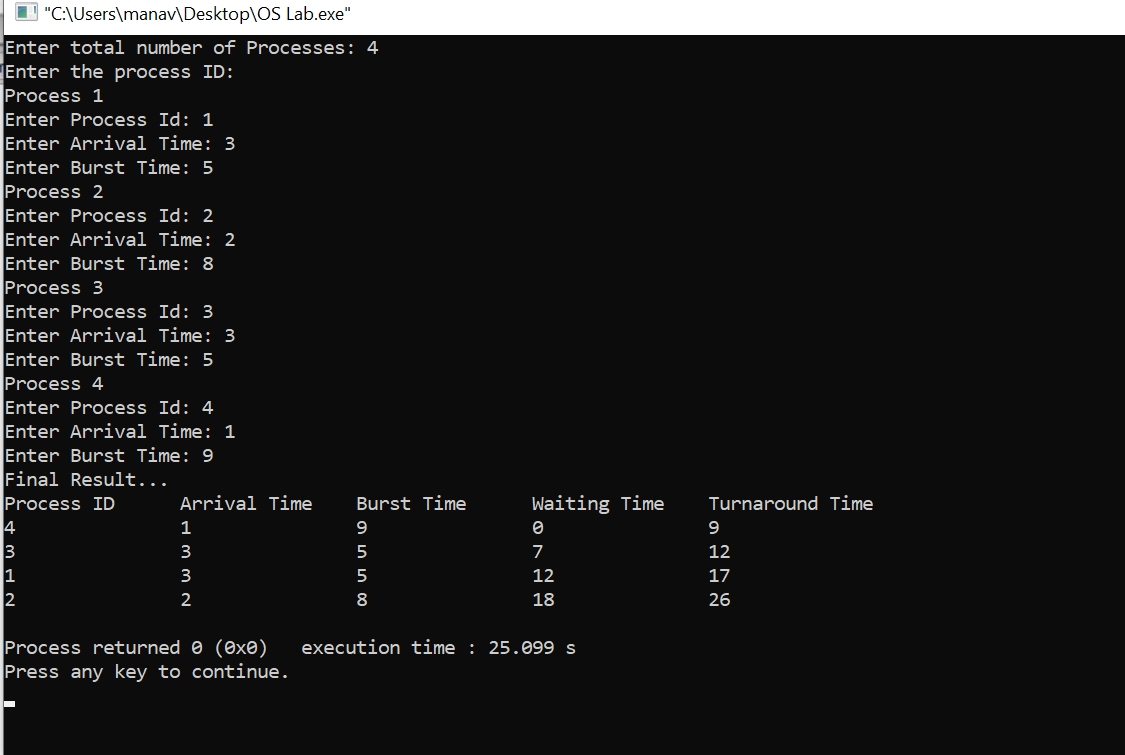
}

**Output:**

1. **FCFS**



**B) SJF**



**Learning From The Experiment:**

Both the algorithms are non-preemptive and both have their pros and cons.

The best thing about FCFS is that it’s the simplest scheduling algorithm. The worst disadvantage of this algorithm is that the average waiting time is quite long. It also leads to an issue called the convoy effect which is that it can cause short processes to wait for very long processes. This results in lower device or CPU utilization and lower efficiency. For SJF, it gives the minimum waiting time for a given set of processes and thus reduces the average waiting time. A loop hole of this algorithm is that long processes may never be processed by the system and may remain in the queue for a very long time.

**EXPERIMENT - 2**

**Aim:**

Write a C program to simulate the following preemptive CPU scheduling algorithms to find turnaround time and waiting time for a given problem a) Round Robin b) Priority.

**Introduction:**

### Preemptive Scheduling:

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

### Key Differences between Preemptive and Non-Preemptive Scheduling Algorithms:

1. In preemptive scheduling the CPU is allocated to the processes for the limited time whereas in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.
3. In Preemptive Scheduling, there is the overhead of switching the process from ready state to running state, vise-versa, and maintaining the ready queue. Whereas in case of non-preemptive scheduling it has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a high priority process frequently arrives in the ready queue then the process with low priority has to wait for a long time, and it may have to starve. On the other hand, in the non-preemptive scheduling, if CPU is allocated to the process having larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling attain flexibility by allowing the critical processes to access the CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.

The following are two Non-Preemptive Scheduling which we will simulate in this lab experiment:

1. **Round Robin:**

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is simple, easy to implement, and starvation-free as all processes get a fair share of CPU. One of the most commonly used techniques in CPU scheduling as a core. In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes.

1. **Priority Scheduling:**

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

**Algorithms:**

1. **Round Robin:**

1- Create an array rem\_bt[] to keep track of remaining

burst time of processes. This array is initially a

copy of bt[] (burst times array)

2- Create another array wt[] to store waiting times

of processes. Initialize this array as 0.

3- Initialize time : t = 0

4- Keep traversing the all processes while all processes

are not done. Do following for i'th process if it is

not done yet.

a- If rem\_bt[i] > quantum

(i) t = t + quantum

(ii) bt\_rem[i] -= quantum;

c- Else // Last cycle for this process

(i) t = t + bt\_rem[i];

(ii) wt[i] = t - bt[i]

(ii) bt\_rem[i] = 0; // This process is over

1. **Priority Scheduling:**

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

2- Sort the processes, according to arrival time if two process arrival time is same then sort according process priority if two process priority are same then sort according to process number.

3- Now simply apply the FCFS algorithm.

**Implementation:**

**A) Round Robin Scheduling:**

#include<iostream>

using namespace std;

void findWaitingTime(int processes[], int n,

int bt[], int wt[], int quantum)

{

int rem\_bt[n];

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

rem\_bt[i] = bt[i];

int t = 0;

while (1)

{

bool done = true;

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

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

t += quantum;

rem\_bt[i] -= quantum;

}

else

{

t = t + rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == true)

break;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])

{

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

tat[i] = bt[i] + wt[i];

}

void findavgTime(int processes[], int n, int bt[], int quantum)

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt, quantum);

findTurnAroundTime(processes, n, bt, wt, tat);

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

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

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {10, 5, 8};

int quantum = 2;

findavgTime(processes, n, burst\_time, quantum);

return 0;

}

**B) Priority Scheduling:**

#include <bits/stdc++.h>

using namespace std;

#define totalprocess 5

struct process

{

int at,bt,pr,pno;

};

process proc[50];

bool comp(process a,process b)

{

if(a.at == b.at)

{

return a.pr<b.pr;

}

else

{

return a.at<b.at;

}

}

void get\_wt\_time(int wt[])

{

int service[50];

service[0] = proc[0].at;

wt[0]=0;

for(int i=1;i<totalprocess;i++)

{

service[i]=proc[i-1].bt+service[i-1];

wt[i]=service[i]-proc[i].at;

if(wt[i]<0)

{

wt[i]=0;

}

}

}

void get\_tat\_time(int tat[],int wt[])

{

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

{

tat[i]=proc[i].bt+wt[i];

}

}

void findgc()

{

int wt[50],tat[50];

double wavg=0,tavg=0;

get\_wt\_time(wt);

get\_tat\_time(tat,wt);

int stime[50],ctime[50];

stime[0] = proc[0].at;

ctime[0]=stime[0]+tat[0];

for(int i=1;i<totalprocess;i++)

{

stime[i]=ctime[i-1];

ctime[i]=stime[i]+tat[i]-wt[i];

}

cout<<"Process\_no\tStart\_time\tComplete\_time\tTurn\_Around\_Time\tWaiting\_Time"<<endl;

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

{

wavg += wt[i];

tavg += tat[i];

cout<<proc[i].pno<<"\t\t"<<

stime[i]<<"\t\t"<<ctime[i]<<"\t\t"<<

tat[i]<<"\t\t\t"<<wt[i]<<endl;

}

cout<<"Average waiting time is : ";

cout<<wavg/(float)totalprocess<<endl;

cout<<"average turnaround time : ";

cout<<tavg/(float)totalprocess<<endl;

}

int main()

{

int arrivaltime[] = { 1, 2, 3, 4, 5 };

int bursttime[] = { 3, 5, 1, 7, 4 };

int priority[] = { 3, 4, 1, 7, 8 };

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

{

proc[i].at=arrivaltime[i];

proc[i].bt=bursttime[i];

proc[i].pr=priority[i];

proc[i].pno=i+1;

}

sort(proc,proc+totalprocess,comp);

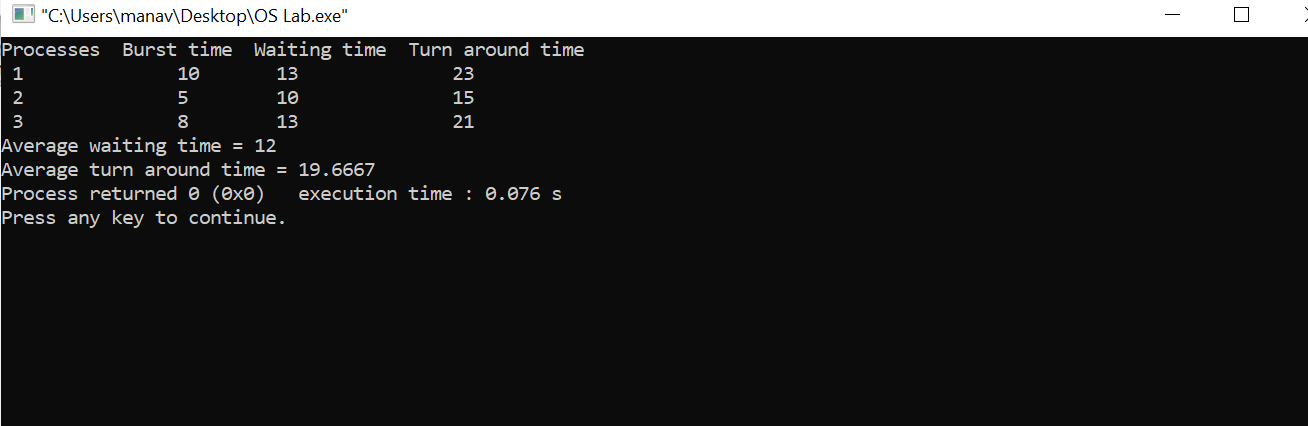
findgc();

return 0;

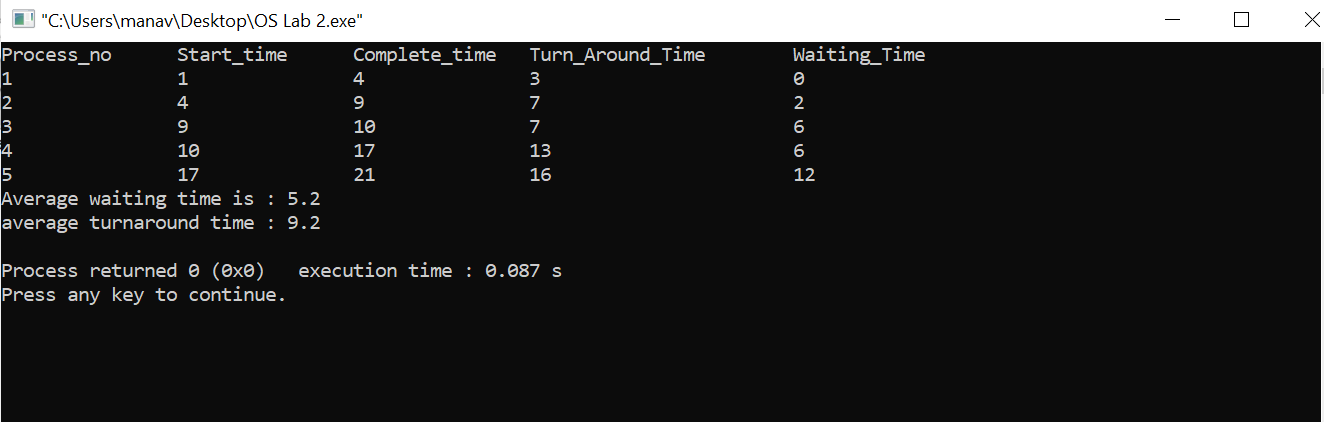
}

**Output:**

**A) Round Robin Scheduling:**



**B) Priority Scheduling:**



**Learning From The Experiment:**

The algorithms are full of pros and also cons for some cases (given below) which we learned from this experiment;

The biggest advantage of the round-robin scheduling method is that If you know the total number of processes on the run queue, then you can also assume the worst-case response time for the same process. However the disadvantage is that there are context switches which increases the time.

For Priority Scheduling Algorithm, Processes are executed on the basis of priority. So high priority does not need to wait for long which saves time also it is a user friendly algorithm. But of course it has its own disadvantages, a major problem is indefinite block, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely.

**EXPERIMENT - 3**

**Aim:**

Write a C program to simulate the following contiguous memory allocation Techniques a) Worst fit b) Best fit c) First fit.

**Introduction:**

### Contiguous Memory Allocation:

Contiguous memory allocation is basically a method in which a single contiguous section/part of memory is allocated to a process or file needing it. Because of this all the available memory space resides at the same place together, which means that the freely/unused available memory partitions are not distributed in a random fashion here and there across the whole memory space.The main memory is a combination of two main portions- one for the operating system and other for the user program. We can implement/achieve contiguous memory allocation by dividing the memory partitions into fixed size partitions.

### Non-Contiguous Memory Allocation:

Non-Contiguous memory allocation is basically a method on the contrary to a contiguous allocation method, allocating the memory space present in different locations to the process as per it’s requirements. As all the available memory space is in a distributed pattern so the freely available memory space is also scattered here and there. This technique of memory allocation helps to reduce the wastage of memory, which eventually gives rise to Internal and external fragmentation.

The following are three Contiguous Memory Allocation techniques which we will simulate in this lab experiment:

1. **Worst Fit:**

The worst fit approach is to locate the largest available free portion available in the main memory, so that the portion left will be big enough to be useful. If a large process comes at a later stage, then memory will not have space to accommodate it. It is the reverse of best fit.

1. **Best Fit:**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to the actual process size needed.

1. **First Fit:**

In the first fit, the partition is allocated which is the first sufficient block from the top of Main Memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough. It finishes after finding the first suitable free partition.

**Algorithms:**

1. **Worst Fit:**

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

3- Start by picking each process and find the

maximum block size that can be assigned to

current process i.e., find max(bockSize[1],

blockSize[2],.....blockSize[n]) >

processSize[current], if found then assign

it to the current process.

5- If not then leave that process and keep checking

the further processes.

1. **Best Fit:**

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

3- Start by picking each process and find the

minimum block size that can be assigned to

current process i.e., find min(bockSize[1],

blockSize[2],.....blockSize[n]) >

processSize[current], if found then assign

it to the current process.

5- If not then leave that process and keep checking

the further processes.

1. **First Fit:**

1- Input memory blocks with size and processes with size.

2- Initialize all memory blocks as free.

3- Start by picking each process and check if it can

be assigned to the current block.

4- If size-of-process <= size-of-block if yes then

assign and check for the next process.

1. If not then keep checking the further blocks.

**Implementation:**

**A) Worst Fit:**

#include<bits/stdc++.h>

using namespace std;

void worstFit(int blockSize[], int m, int processSize[],int n)

{

int allocation[n];

memset(allocation, -1, sizeof(allocation));

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

{

int wstIdx = -1;

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

{

if (blockSize[j] >= processSize[i])

{

if (wstIdx == -1)

wstIdx = j;

else if (blockSize[wstIdx] < blockSize[j])

wstIdx = j;

}

}

if (wstIdx != -1)

{

allocation[i] = wstIdx;

blockSize[wstIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

worstFit(blockSize, m, processSize, n);

return 0 ;

}

**B) Best Fit:**

#include<bits/stdc++.h>

using namespace std;

void bestFit(int blockSize[], int m, int processSize[], int n)

{

int allocation[n];

memset(allocation, -1, sizeof(allocation));

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

{

int bestIdx = -1;

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

{

if (blockSize[j] >= processSize[i])

{

if (bestIdx == -1)

bestIdx = j;

else if (blockSize[bestIdx] > blockSize[j])

bestIdx = j;

}

}

if (bestIdx != -1)

{

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

bestFit(blockSize, m, processSize, n);

return 0 ;

}

**C) First Fit:**

#include<bits/stdc++.h>

using namespace std;

void firstFit(int blockSize[], int m,

int processSize[], int n)

{

int allocation[n];

memset(allocation, -1, sizeof(allocation));

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

{

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

{

if (blockSize[j] >= processSize[i])

{

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t"

<< processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0 ;

}

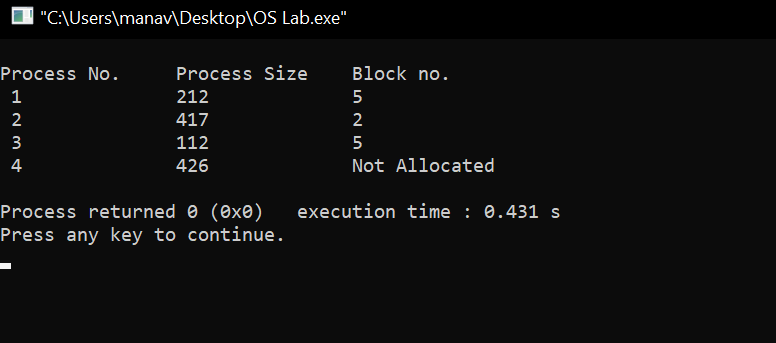
**Output:**

The Input given in the program for all three cases were:

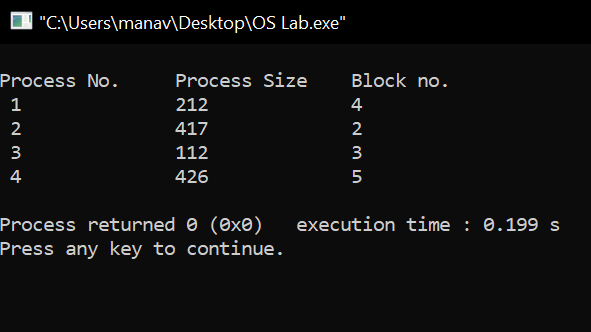
Block Size[] = {100, 500, 200, 300, 600};

Process Size[] = {212, 417, 112, 426};

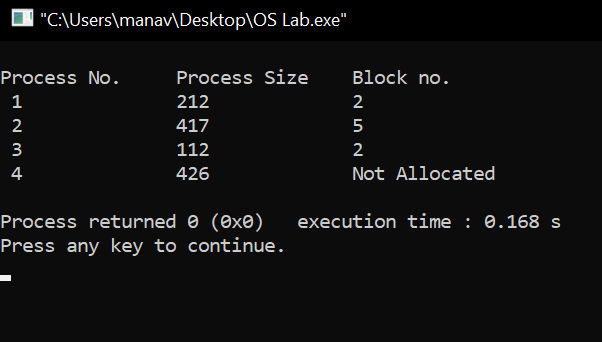
**A) Worst Fit:**

****

**B) Best Fit:**

****

**C) First Fit:**

****

**Learning From The Experiment:**

The algorithms are full of pros and also cons for some cases (given below) which we learned from this experiment;

Worst Fit chooses the largest hole/partition, therefore there will be large internal fragmentation. Now, this internal fragmentation will be quite big so that other small processes can also be placed in that leftover partition which is a great advantage of this algorithm. However, it is a slow process because it traverses all the partitions in the memory and then selects the largest partition among all the partitions, which is a time-consuming process.

Advantage of Best fit is that memory utilization in best fit is much better than first fit as it searches the smallest free partition first available.The operating system allocates the job minimum possible space in the memory, making memory management very efficient. To save memory from getting wasted, it is the best method. Still, it has a disadvantage that it is slower and may even tend to fill up memory with tiny useless holes.

First fit is the fastest algorithm because it searches as little as possible. Although the disadvantage is that the remaining unused memory areas left after allocation become waste if it is too small. Thus requests for larger memory requirements cannot be accomplished.

**EXPERIMENT - 4**

**Aim:**

Write a C program to simulate all file allocation strategies for a)Sequential, b) Indexed.

**Introduction:**

### File Allocation Methods:

The allocation methods define how the files are stored in the disk blocks. We must select the best method for the file allocation because it will directly affect the system performance and system efficiency. With the help of the allocation method, we can utilize the disk, and also files can be accessed.The main idea behind these methods is to provide efficient disk space utilization and fast access to the file blocks.

There are three main disk space or file allocation methods.

1. Sequential Allocation
2. Linked Allocation
3. Indexed Allocation
4. **Sequential Allocation:**

In Sequential Allocation, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n-1. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file. The directory entry for a file with contiguous allocation contains the address of the starting block and length of the allocated portion.

1. **Linked Allocation:**

In the linked list allocation method, it overcomes the drawbacks of the contiguous allocation method. In this file allocation method, each file is treated as a linked list of disks blocks. In the linked list allocation method, it is not required that disk blocks assigned to a specific file are in the contiguous order on the disk. The directory entry comprises a pointer for starting file block and also for the ending file block. Each disk block that is allocated or assigned to a file consists of a pointer, and that pointer points to the next block of the disk, which is allocated to the same file.

1. **Indexed Allocation:**

The Indexed allocation method is another method that is used for file allocation. In the index allocation method, we have an additional block, and that block is known as the index block. For each file, there is an individual index block. In the index block, the ith entry holds the disk address of the ith file block. We can see in the below figure that the directory entry comprises the address of the index block.

We will be simulating two of these algorithms which are Sequential Allocation and Indexed Allocation.

**Algorithms:**

1. **Sequential Allocation:**

1- Start the program.

2- Gather information about the number of files.

3- Gather the memory requirement of each file.

4- Allocate the memory to the file in a sequential manner.

5- Select any random location from the available location.

6- Check if the location that is selected is free or not.

7- If the location is allocated set the flag = 1.

8- Print the file number, length, and the block allocated.

9- Gather information if more files have to be stored.

10- If yes, then go to point 2.

11- If no, Stop the program.

1. **Indexed Allocation:**

1- Start the program.

2- Get information about the number of files.

3- Get the memory requirement of each file.

4- Allocate the memory to the file by selecting random locations.

5- Check if the location that is selected is free or not.

6- If the location is allocated set the flag = 1, and if free set flag = 0.

7- Print the file number, length, and the block allocated.

8- Gather information if more files have to be stored.

9- If yes, then go to point 2.

10- If no, Stop the program.

**Implementation:**

**A) Sequential Allocation:**

#include <iostream>

#include <conio.h>

#include <stdlib.h>

using namespace std;

void recurse(int files[]){

int flag = 0, startBlock, len, k;

cout << "Enter the starting block and the length of the files: ";

cin >> startBlock >> len;

for (int j=startBlock; j<(startBlock+len); j++){

if (files[j] == 0)

flag++;

}

if(len == flag){

for (int k=startBlock; k<(startBlock+len); k++){

if (files[k] == 0){

files[k] = 1;

cout << k <<"\t" << files[k] << endl;

}

}

if (k != (startBlock+len-1))

cout << "The file is allocated to the disk" << endl;

}

else

cout << "The file is not allocated to the disk" << endl;

cout << "Do you want to enter more files?" << endl;

int ch;

cout << "Press 1 for YES, 0 for NO: ";

cin >> ch;

if (ch == 1)

recurse(files);

else

exit(0);

return;

}

int main()

{

int files[50];

for(int i=0;i<50;i++)

files[i]=0;

recurse(files);

getch();

return 0;

}

**B) Indexed Allocation:**

#include <iostream>

#include <conio.h>

#include <stdlib.h>

using namespace std;

int files[50], indexBlock[50], indBlock, n;

void recurse1();

void recurse2();

void recurse1(){

cout << "Enter the index block: ";

cin >> indBlock;

if (files[indBlock] != 1){

cout << "Enter the number of blocks and the number of files needed for the index " << indBlock << " on the disk: ";

cin >> n;

}

else{

cout << indBlock << " is already allocated" << endl;

recurse1();

}

recurse2();

}

void recurse2(){

int flag = 0;

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

cin >> indexBlock[i];

if (files[indexBlock[i]] == 0)

flag++;

}

if (flag == n){

for (int j=0; j<n; j++){

files[indexBlock[j]] = 1;

}

cout << "Allocated" << endl;

cout << "File Indexed" << endl;

for (int k=0; k<n; k++){

cout << indBlock << " ------> " << indexBlock[k] << ": " << files[indexBlock[k]] << endl;

}

}

else{

cout << "File in the index is already allocated" << endl;

cout << "Enter another indexed file" << endl;

recurse2();

}

cout << "Do you want to enter more files?" << endl;

cout << "Enter 1 for Yes, Enter 0 for No: ";

int ch;

cin >> ch;

if (ch == 1)

recurse1();

else

exit(0);

return;

}

int main()

{

for(int i=0;i<50;i++)

files[i]=0;

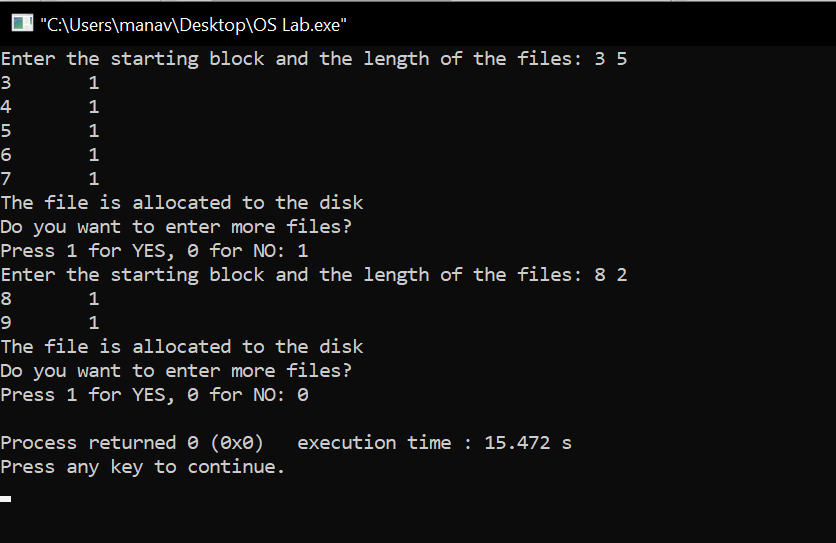
recurse1();

return 0;

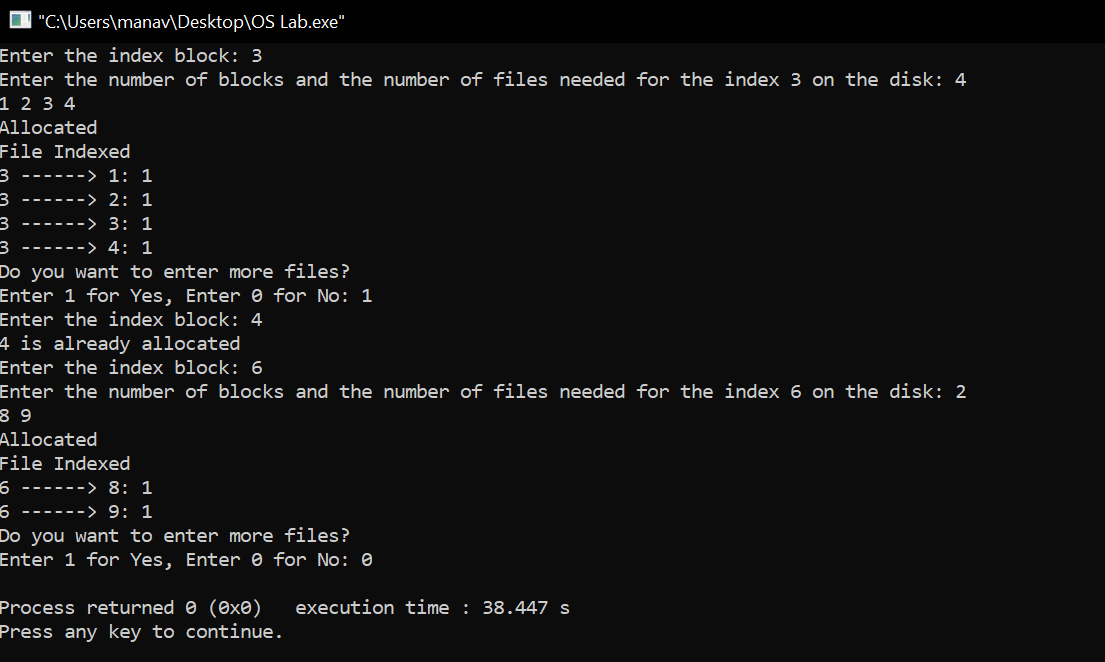
}

**Output:**

**A) Sequential Allocation:**

****

**B) Indexed Allocation:**

****

**Learning From The Experiment:**

The algorithms are full of pros and also cons for some cases (given below) which we learned from this experiment;

The Sequential Allocation supports both the Sequential and Direct Accesses. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks. However, this method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization. Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

The Indexed Allocation supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.It overcomes the problem of external fragmentation. Still, The pointer overhead for indexed allocation is greater than linked allocation.For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

**EXPERIMENT - 5**

**Aim:**

Write a C program to simulate Bankers Algorithm for DeadLock Avoidance.

**Introduction:**

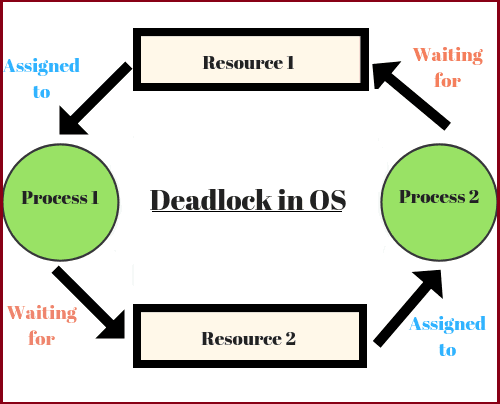
Deadlock:

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grants the resource if it is available otherwise let the process waits.
3. The process uses it and releases on the completion.

A Deadlock is a situation where each of the computer processes waits for a resource which is being assigned to some other process. In this situation, none of the process gets executed since the resource it needs is held by some other process which is also waiting for some other resource to be released.

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Necessary conditions for Deadlocks:

1. Mutual Exclusion: A resource can only be shared in a mutually exclusive manner. It implies, if two processes cannot use the same resource at the same time.
2. Hold and Wait: A process waits for some resources while holding another resource at the same time.
3. No preemption: The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.
4. Circular Wait: All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

**Deadlock Avoidance:**

Simplest and most useful model requires that each process declare

the maximum number of resources of each type that it may need.

The deadlock-avoidance algorithm dynamically examines the

resource-allocation state to ensure that there can never be a

circular-wait condition. Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

**Bankers Algorithm:**

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.

Data Structures required for Banker’s Algorithm

* Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available.
* Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
* Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
* Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.

Need [i,j] = Max[i,j] – Allocation [i,j]

**Algorithms:**

**Banker’s Algorithm:**

Banker’s algorithm consists of Safety algorithm and Resource request algorithm:

Safety Algorithm:

1- Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2- Find an i such that both

a) Finish[i] = false

b) Needi <= Work

if no such i exists goto step (4)

3- Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4- if Finish [i] = true for all i then the system is in a safe state.

Resource-Request Algorithm:

1- If Requesti <= Needi

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2- If Requesti <= Available

Goto step (3); otherwise, Pi must wait, since the resources are not available.

3- Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Requesti

Allocationi = Allocationi + Requesti

Needi = Needi– Requesti

**Implementation:**

**Banker’s Algorithm:**

The following is the input values and attributes for the processes with their requested resource.

----------------------------------------------------------------------------------------------------------------

Process | Allocation | Max Request | Available Resource |

| X Y Z | X Y Z | X Y Z |

----------------------------------------------------------------------------------------------------------------

Process0 | 0 1 0 | 6 6 4 | 3 2 4 |

Process1 | 3 0 0 | 3 2 2 | |

Process2 | 2 0 2 | 9 0 2 | |

Process3 | 1 1 1 | 2 2 2 | |

Process4 | 0 0 2 | 4 3 3 | |

---------------------------------------------------------------------------------------------------------------

**PROGRAM:**

#include <iostream>

using namespace std;

int main()

{

int n, m, i, j, k;

n = 5;

m = 3;

int max\_matrix[5][3] ={ { 6, 6, 4 },

{ 3, 2, 2 },

{ 9, 0, 2 },

{ 2, 2, 2 },

{ 4, 3, 3 } };

int available\_resources[3] = { 3, 2, 4 };

int allocation\_matrix[5][3] = { { 0, 1, 0 },

{ 3, 0, 0 },

{ 2, 0, 2 },

{ 1, 1, 1 },

{ 0, 0, 2 } };

int f[n], ans[n], ind = 0;

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

f[k] = 0;

}

int need\_matrix[n][m];

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

for (j = 0; j < m; j++)

need\_matrix[i][j] = max\_matrix[i][j] - allocation\_matrix[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\_matrix[i][j] > available\_resources[j]){

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

for (y = 0; y < m; y++)

available\_resources[y] += allocation\_matrix[i][y];

f[i] = 1;

}

}

}

}

cout << "Following is the SAFE Sequence" << endl;

for (i = 0; i < n - 1; i++)

cout << " P" << ans[i] << " ->";

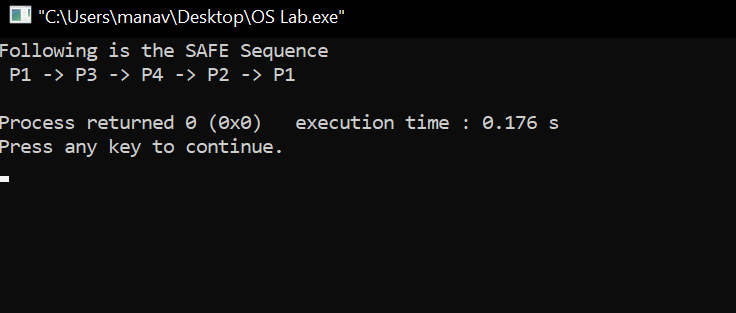
cout << " P" << ans[n - 1] <<endl;

return (0);

}

**Output:**

**Banker’s Algorithm**

****

**Learning From The Experiment:**

Banker’s algorithm avoids deadlock and it is less restrictive than deadlock prevention. It is not necessary to preempt and rollback processes, as in deadlock detection.

However, the restrictions are that it only works with a fixed number of resources and processes. It only guarantees finite time - not reasonable response time. It needs advanced knowledge of maximum needs. It is not suitable for multi-access systems. Maximum resource requirement for each process must be stated in advance. Processes under consideration must be independent and with no synchronization requirements. There must be a fixed number of resources to allocate. No process may exit while holding resources.

**EXPERIMENT - 6**

**Aim:**

Write a C program to simulate Bankers Algorithm for DeadLock Prevention.

**Introduction:**

### Deadlock:

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grants the resource if it is available otherwise let the process waits.
3. The process uses it and releases on the completion.

A Deadlock is a situation where each of the computer processes waits for a resource which is being assigned to some other process. In this situation, none of the process gets executed since the resource it needs is held by some other process which is also waiting for some other resource to be released.

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

Necessary conditions for Deadlocks:

1. Mutual Exclusion: A resource can only be shared in a mutually exclusive manner. It implies, if two processes cannot use the same resource at the same time.
2. Hold and Wait: A process waits for some resources while holding another resource at the same time.
3. No preemption: The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.
4. Circular Wait: All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

**Bankers Algorithm:**

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.

Data Structures required for Banker’s Algorithm

* Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available.
* Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
* Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
* Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.

Need [i,j] = Max[i,j] – Allocation [i,j]

**Algorithm:**

1- Start the Program.

2- Get the values of resources and processes.

3- Get the avail value.

4- After allocation, find the needed value.

5- Check whether it's possible to allocate.

6- If it is possible then the system is in a safe state.

7- Stop the execution.

**Implementation:**

#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("\*\*\*\*\*\*\*\*\*\* Deadlock Detection Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\n");

input();

show();

cal();

getch();

return 0;

}

void input()

{

int i,j;

printf("Enter the number of Processes\t");

scanf("%d",&n);

printf("Enter the number of resource 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\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 dead[100];

int safe[100];

int i,j;

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

{

finish[i]=0;

}

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

{

for(j=0;j<r;j++)

{

need[i][j]=max[i][j]-alloc[i][j];

}

}

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("\nP%d",i);

if(finish[i]==1)

{

i=n;

}}}}}}

j=0;

flag=0;

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

{

if(finish[i]==0)

{

dead[j]=i;

j++;

flag=1;

}

}

if(flag==1)

{

printf("\n\nSystem is in Deadlock and the Deadlock process are\n");

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

{

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

}

}

else

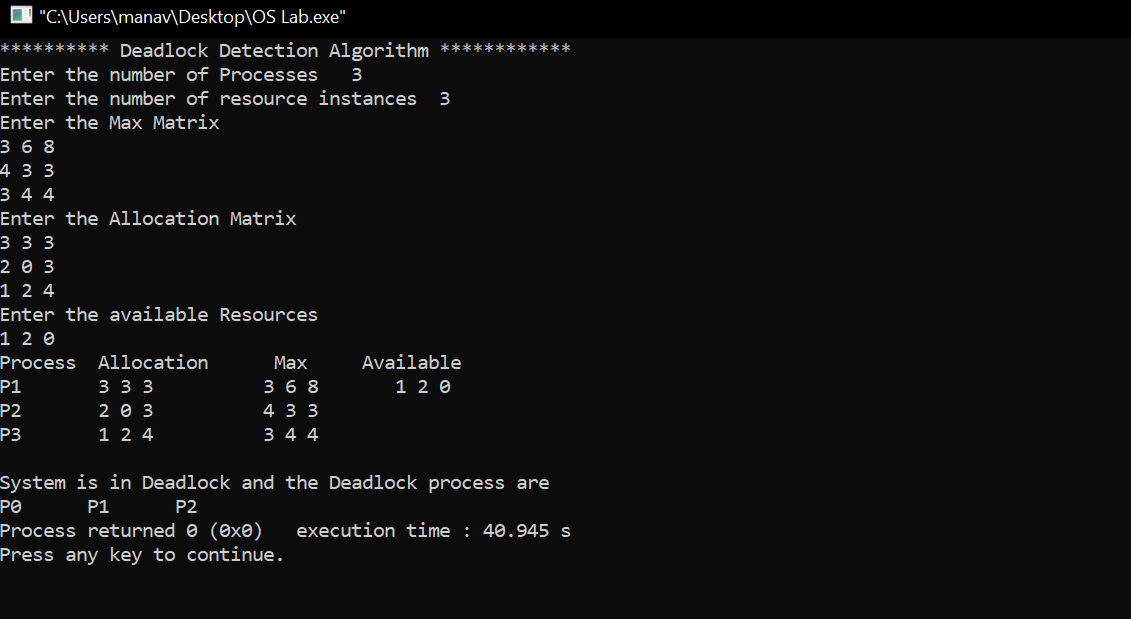
{

printf("\nNo Deadlock Occur");

}

}

**Output:**

****

**Learning From The Experiment:**

Banker’s algorithm avoids deadlock and it is less restrictive than deadlock prevention. It is not necessary to preempt and rollback processes, as in deadlock detection.

However, the restrictions are that it only works with a fixed number of resources and processes. It only guarantees finite time - not reasonable response time. It needs advanced knowledge of maximum needs. It is not suitable for multi-access systems. Maximum resource requirement for each process must be stated in advance. Processes under consideration must be independent and with no synchronization requirements. There must be a fixed number of resources to allocate. No process may exit while holding resources.

**EXPERIMENT - 7**

**Aim:**

Write a C program to simulate Producer-Consumer problem using semaphores.

**Introduction:**

### Producer-Consumer Problem:

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores.

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. Semaphores are of two types:

1. Binary Semaphore :

This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

1. Counting Semaphore :

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

A semaphore S is an integer variable that can be accessed only through two standard operations : *wait() and signal().*

The *wait()* operation reduces the value of semaphore by 1 and the *signal()* operation increases its value by 1.

*wait(S){*

*while(S<=0); // busy waiting*

*S--;*

*}*

*signal(S){*

*S++;*

*}*

Producer-Consumer 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.

Semaphores Initialization:

*mutex = 1*

*Full = 0 // Initially, all slots are empty. Thus full slots are 0*

*Empty = n // All slots are empty initially*

Producer Solution:

*do{*

*//produce an item*

*wait(empty);*

*wait(mutex);*

*//place in buffer*

*signal(mutex);*

*signal(full);*

*}while(true)*

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

Consumer solution:

*do{*

*wait(full);*

*wait(mutex);*

*// remove item from buffer*

*signal(mutex);*

*signal(empty);*

*// consumes item*

*}while(true)*

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

**Algorithm:**

Algorithm for Producer-

do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

Algorithm for Consumer-

do{

wait(full);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

**Implementation:**

#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);

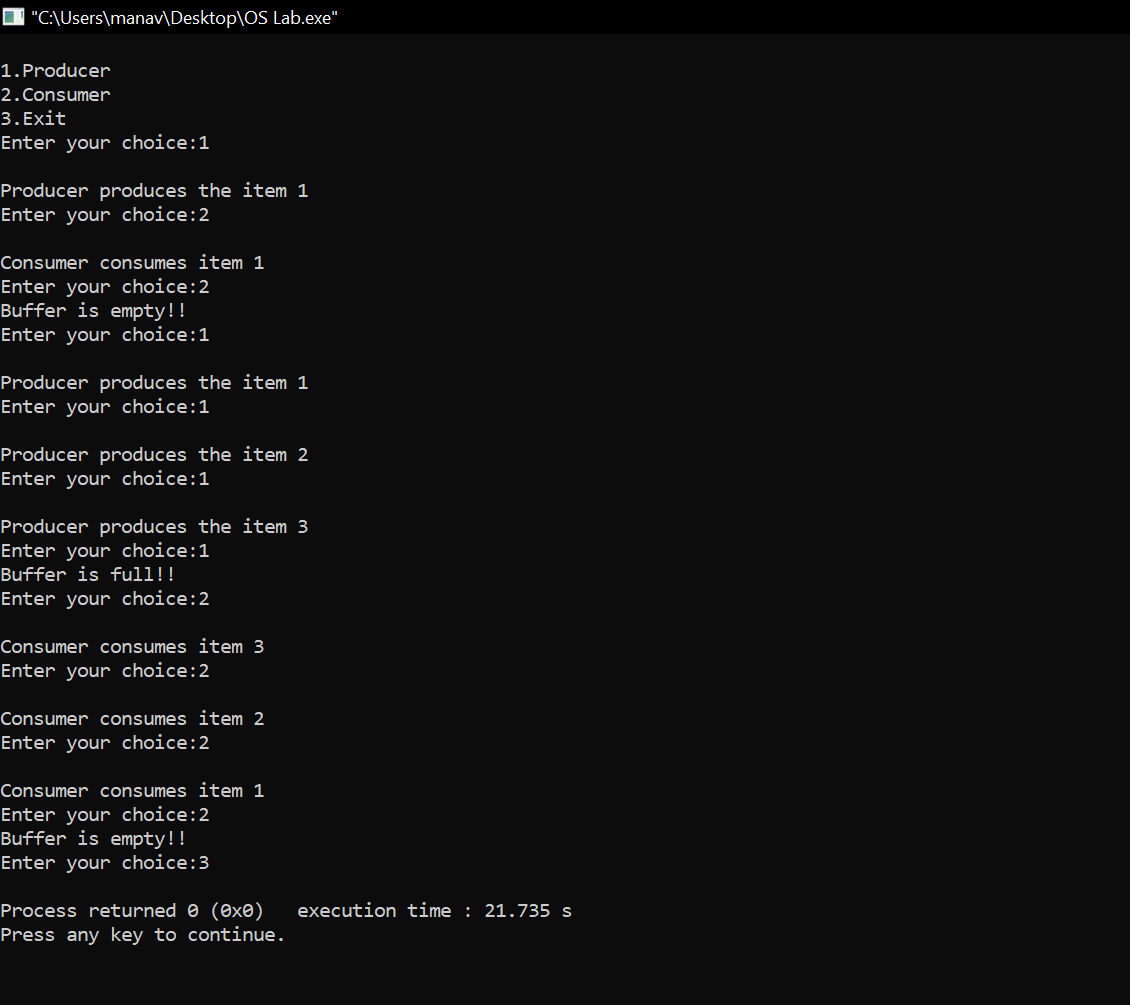
printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

**Output:**

****

**Learning From The Experiment:**

Semaphores has many advantages which are that they do not allow more than one process to enter the critical section. In this way, mutual exclusion is achieved and thus they are extremely efficient than other techniques for synchronization. Due to busy waiting in semaphore, there is no wastage of process time and resources. This is because the processes are only allowed to enter the critical section after satisfying a certain condition.

Having so many advantages of it, there are always some disadvantages which is priority inversion in semaphores. The operating system has to keep track of all calls to wait and to signal the semaphore.

**EXPERIMENT - 8**

**Aim:**

Write a C program to simulate page replacement algorithms a) FIFO b) LRU

**Introduction:**

Page Replacement Algorithms:

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

The following are the page replacement algorithms which we will be implementing:

### First In First Out (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.

*Example:* Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames.Find number of page faults. Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.

when 3 comes, it is already in memory so —> 0 Page Faults.

Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault.

6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault. Finally when 3 come it is not available so it replaces 0 1 page fault

*Belady’s anomaly:*  Belady’s anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

### Least Recently Used:

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

*Example-* Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults.Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already their so —> 0 Page fault.

when 3 came it will take the place of 7 because it is least recently used —>1 Page fault

0 is already in memory so —> 0 Page fault.

4 will takes place of 1 —> 1 Page Fault

Now for the further page reference string —> 0 Page fault because they are already available in the memory.

**Algorithms:**

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1. **First In First Out (FIFO)**

1- Start traversing the pages.

i) If set holds less pages than capacity.

1. Insert page into the set one by one until

the size of set reaches capacity or all

page requests are processed.

1. Simultaneously maintain the pages in the

queue to perform FIFO.

1. Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

1. Remove the first page from the queue as it was the first to be entered in

the memory

1. Replace the first page in the queue with the current page in the string.
2. Store current page in the queue.
3. Increment page faults.

2- Return page faults.

1. **Least Recently Used**

1- Start traversing the pages.

i) If set holds less pages than capacity.

1. Insert page into the set one by one until the size of set reaches capacity or all

page requests are processed.

1. Simultaneously maintain the recent occurred index of each page in a map called indexes.
2. Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

1. Find the page in the set that was least recently used. We find it using index array.We basically need to replace the page with minimum index.
2. Replace the found page with current page.
3. Increment page faults.
4. Update index of current page.
5. Return page faults.

**Implementation:**

**A) First In First Out (FIFO)**

#include<stdio.h>

#include<conio.h>

int main()

{

int a[5],b[20],n,p=0,q=0,m=0,h,k,i,q1=1;

char f='F';

printf("Enter the Number of Pages:"); scanf("%d",&n);

printf("Enter %d Page Numbers:",n); for(i=0;i<n;i++)

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

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

{if(p==0)

{

if(q>=3)

q=0;

a[q]=b[i];

q++;

if(q1<3)

{

q1=q;

}

}

printf("\n%d",b[i]);

printf("\t");

for(h=0;h<q1;h++)

printf("%d",a[h]);

if((p==0)&&(q<=3))

{

printf("-->%c",f);m++;

}

p=0;

for(k=0;k<q1;k++)

{

if(b[i+1]==a[k])

p=1;

}

}

printf("\nNo of faults:%d",m);

getch();

return 0;

}

**B) Least Recently Used**

#include<stdio.h>

#include<conio.h>

int main()

{

int g=0,a[5],b[20],p=0,q=0,m=0,h,k,i,q1=1,j,u,n;

char f='F';

printf("Enter the number of pages:"); scanf("%d",&n);

printf("Enter %d Page Numbers:",n); for(i=0;i<n;i++)

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

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

{if(p==0)

{

if(q>=3)

q=0;

a[q]=b[i];

q++;

if(q1<3)

{

q1=q;

}

}

printf("\n%d",b[i]);

printf("\t");

for(h=0;h<q1;h++)

printf("%d",a[h]);

if((p==0)&&(q<=3))

{

printf("-->%c",f);m++;

}

p=0;

g=0;

if(q1==3)

{

for(k=0;k<q1;k++)

{

if(b[i+1]==a[k])

p=1;

}

for(j=0;j<q1;j++)

{

u=0;

k=i;while(k>=(i-1)&&(k>=0))

{

if(b[k]==a[j])

u++;k--;

}

if(u==0)

q=j;

}

}

else

{

for(k=0;k<q;k++)

{

if(b[i+1]==a[k])

p=1;

}

}

}

printf("\nNo of faults:%d",m);

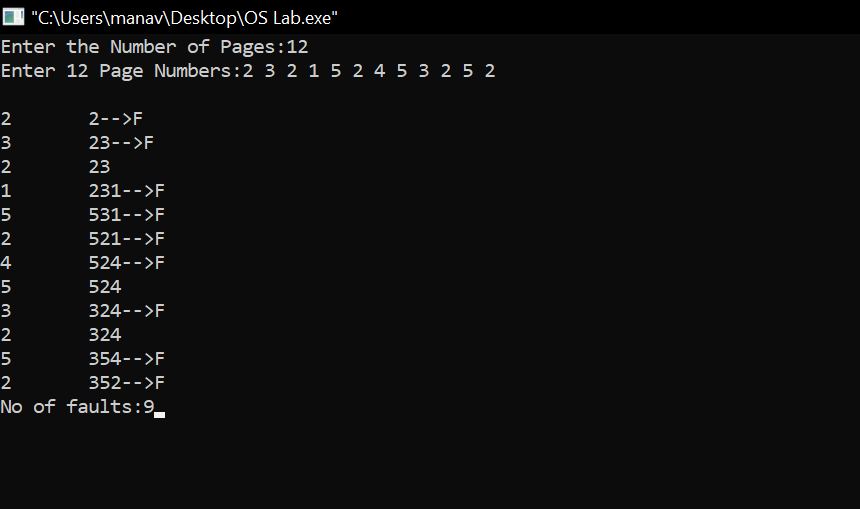
getch();

return 0;

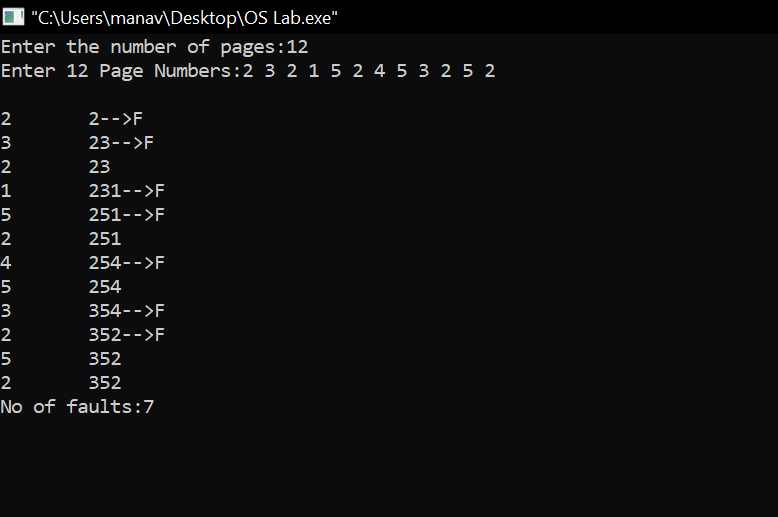
}

**Output:**

**A) First In First Out (FIFO)**



**B) Least Recently Used**



**Learning From The Experiment:**

Both the algorithms are have their pros and cons.

The only advantage of FIFO is that it is simple and easy to understand & implement. However, the disadvantage is that the process effectiveness is low and when we increase the number of frames while using FIFO, we are giving more memory to processes. So, page fault should decrease, but here the page faults are increasing. This problem is called as Belady’s Anomaly.

The advantage of LRU is that it is open for full analysis. We replace the page which is least recently used in LRU, thus free from Belady’s Anomaly. It is easy to choose page which has faulted and hasn’t been used for a long time. However, it requires additional Data Structure to be implemented.The hardware assistance is high.

**EXPERIMENT - 9**

**Aim:**

Write a C program to simulate page replacement algorithms a) LFU b) Optimal

**Introduction:**

Page Replacement Algorithms:

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

The following are the page replacement algorithms which we will be implementing:

### Least Frequently Used (LFU):

In LFU Page Replacement method, the page with the minimum count is selected for replacement with the page that needs to enter into the system. LFU is a cache algorithm which is used to manage computer’s memory. A counter is assigned to every block of memory that is loaded into the cache memory. However, the LFU technique is hardly implemented these days but this algorithm is normally combined with other algorithms which make it a hybrid algorithm, and then it is implemented. LFU algorithm is sometimes also combined with LRU replacement algorithm, and then implemented.

### Optimal:

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

*Example-* Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already there so —> 0 Page fault.

when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>1 Page fault.

0 is already there so —> 0 Page fault..

4 will takes place of 1 —> 1 Page Fault.

Now for the further page reference string —> 0 Page fault because they are already available in the memory.

**Algorithms:**

1. **Least Frequently Used (LFU):**

In LFU Page Replacement method, the page with the minimum count is selected for replacement with the page that needs to enter into the system.

LFU is a cache algorithm which is used to manage computer’s memory. A counter is assigned to every block of memory that is loaded into the cache memory.

ALGORITHM:

1 - In the current stack at any iteration choose that element

for replacement which has the smallest count in the incoming

page stream.

2 - Check the old page as well as the frequency of that Page.

1. If the frequency of the page is larger than the old page do

not remove it.

2. If all the old pages are having the same frequency then take

the last i.e FIFO method for that and remove that page.

1. **Optimal:**

In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

*Example :*

**Input :** Number of frames, fn = 3

Reference String, pg[] = {7, 0, 1, 2,

0, 3, 0, 4, 2, 3, 0, 3, 2, 1,

2, 0, 1, 7, 0, 1};

**Output :** No. of hits = 11

No. of misses = 9

The main idea is for every reference we do following :

1 - If the referred page is already present, increment hit

count.

2 - If not present, find a page that is never referenced in Future.

1. If such a page exists, replace this page with a new page.

2. If no such page exists, find a page that is referenced farthest

in future.

3 - Replace this page with a new page

Implementation:

**A) Least Frequently Used (LFU):**

#include<stdio.h>

int main()

{

int total\_frames, total\_pages, hit = 0;

int pages[25], frame[10], arr[25], time[25];

int m, n, page, flag, k, minimum\_time, temp;

printf("Enter Total Number of Pages:\t");

scanf("%d", &total\_pages);

printf("Enter Total Number of Frames:\t");

scanf("%d", &total\_frames);

for(m = 0; m < total\_frames; m++)

{

frame[m] = -1;

}

for(m = 0; m < 25; m++)

{

arr[m] = 0;

}

printf("Enter Values of Reference String\n");

for(m = 0; m < total\_pages; m++)

{

printf("Enter Value No.[%d]:\t", m + 1);

scanf("%d", &pages[m]);

}

printf("\n");

for(m = 0; m < total\_pages; m++)

{

arr[pages[m]]++;

time[pages[m]] = m;

flag = 1;

k = frame[0];

for(n = 0; n < total\_frames; n++)

{

if(frame[n] == -1 || frame[n] == pages[m])

{

if(frame[n] != -1)

{

hit++;

}

flag = 0;

frame[n] = pages[m];

break;

}

if(arr[k] > arr[frame[n]])

{

k = frame[n];

}

}

if(flag)

{

minimum\_time = 25;

for(n = 0; n < total\_frames; n++)

{

if(arr[frame[n]] == arr[k] && time[frame[n]] < minimum\_time)

{

temp = n;

minimum\_time = time[frame[n]];

}

}

arr[frame[temp]] = 0;

frame[temp] = pages[m];

}

for(n = 0; n < total\_frames; n++)

{

printf("%d\t", frame[n]);

}

printf("\n");

}

printf("Page Hit:\t%d\n", hit);

return 0;

}

**B) Optimal:**

#include <bits/stdc++.h>

using namespace std;

bool search(int key, vector<int>& fr)

{

for (int i = 0; i < fr.size(); i++)

if (fr[i] == key)

return true;

return false;

}

int predict(int pg[], vector<int>& fr, int pn, int index)

{

int res = -1, farthest = index;

for (int i = 0; i < fr.size(); i++) {

int j;

for (j = index; j < pn; j++) {

if (fr[i] == pg[j]) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

if (j == pn)

return i;

}

return (res == -1) ? 0 : res;

}

void optimalPage(int pg[], int pn, int fn)

{

vector<int> fr;

int hit = 0;

for (int i = 0; i < pn; i++) {

if (search(pg[i], fr)) {

hit++;

continue;

}

if (fr.size() < fn)

fr.push\_back(pg[i]);

else {

int j = predict(pg, fr, pn, i + 1);

fr[j] = pg[i];

}

}

cout << "No. of Hits = " << hit << endl;

cout << "No. of Misses = " << pn - hit << endl;

}

int main()

{

int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };

int pn = sizeof(pg) / sizeof(pg[0]);

int fn = 4;

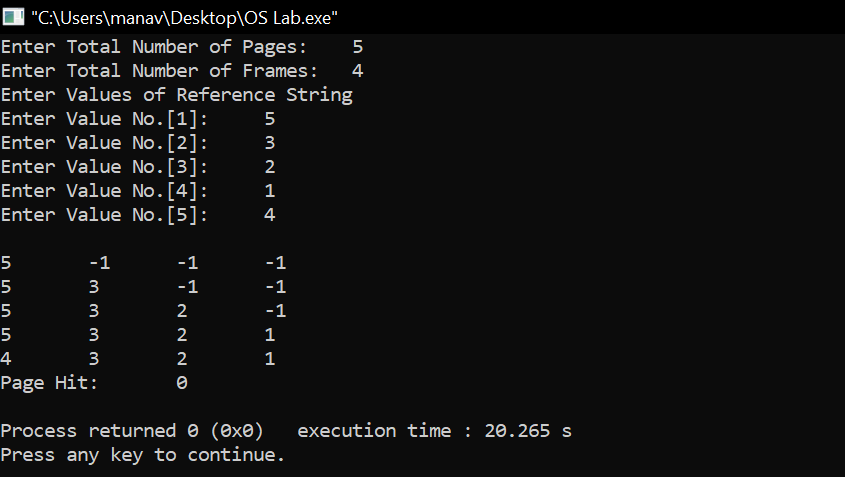
optimalPage(pg, pn, fn);

return 0;

}

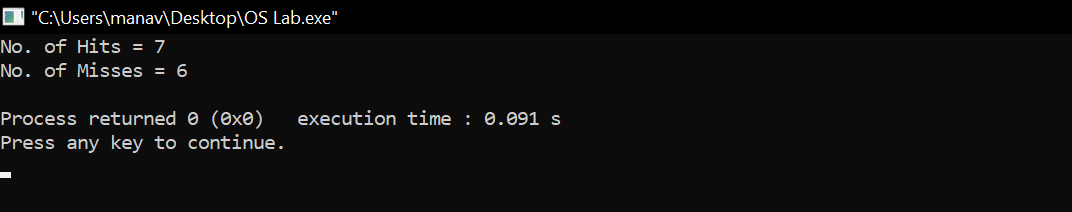
**Output:**

1. **Least Frequently Used (LFU):**

****

**B) Optimal:**

Input - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2

****

**Learning From The Experiment:**

While the LFU method may seem like an intuitive approach to memory management it is not without faults. Consider an item in memory which is referenced repeatedly for a short period of time and is not accessed again for an extended period of time. Due to how rapidly it was just accessed its counter has increased drastically even though it will not be used again for a decent amount of time. This leaves other blocks which may actually be used more frequently susceptible to purging simply because they were accessed through a different method.

In Optimal Page replacement, complexity is less and it is easy to implement. Assistance needed is low i.e Data Structure used are easy and light. Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

**EXPERIMENT - 10**

**Aim:**

Write a C program to simulate disk scheduling algorithms. a) First Come First Serve (FCFS) b) SCAN

**Introduction:**

Disk Scheduling Algorithms:

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

Disk scheduling is important because:

1. Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
2. Two or more request may be far from each other so can result in greater disk arm movement.
3. Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let’s have a quick look at some of the important terms:

**Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

**Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

**Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

**Disk Access Time:** Disk Access Time = Seek Time + Rotational Latency + Transfer Time

**Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. Average Response time is the response time of the all requests. Variance Response Time is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

The following are the disk scheduling algorithms which we will be implementing:

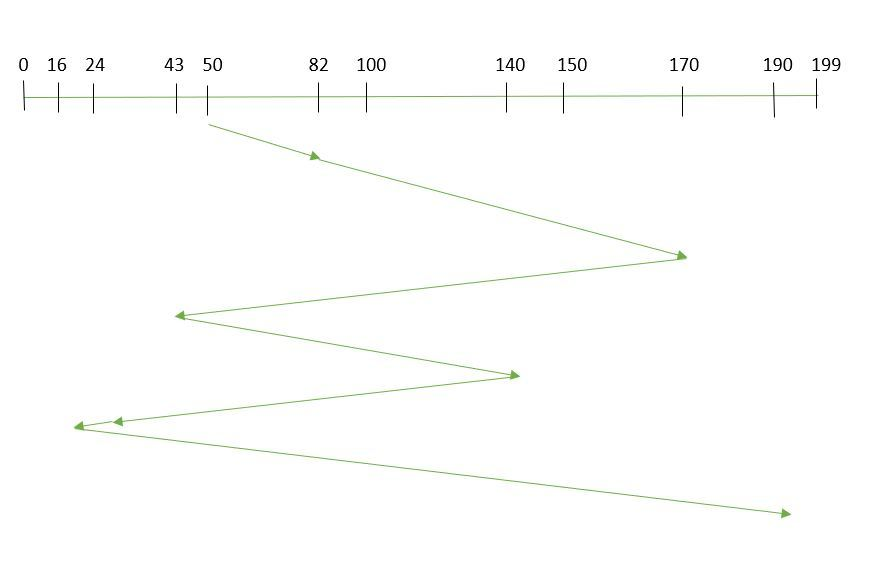
### First Come First Serve (FCFS):

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.Let us understand this with the help of an example.

*Example:*

Suppose the order of request is- (82,170,43,140,24,16,190)

And current position of Read/Write head is : 50



So, total seek time:

=(82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16)

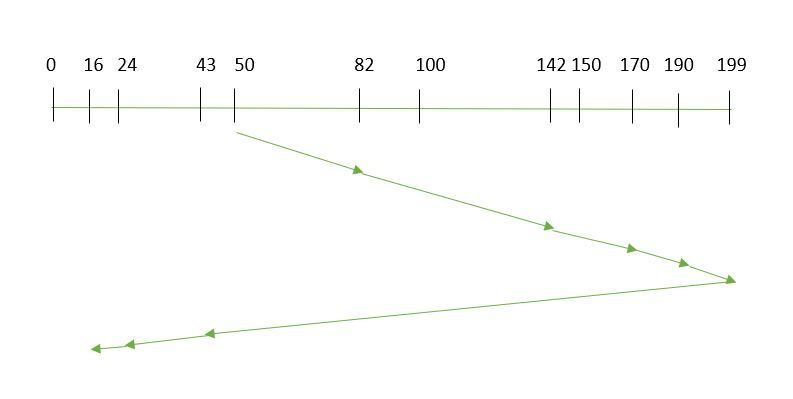
=642

### SCAN:

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

*Example:*

Suppose the requests to be addressed are- 82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “towards the larger value”.



Therefore, the seek time is calculated as:

=(199-50)+(199-16)

=332

Algorithms:

1. **First Come First Serve (FCFS):**

1- Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.

2- Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.

3- Increment the total seek count with this distance.

4- Currently serviced track position now becomes the new head position.

5- Go to step 2 until all tracks in request array have not been serviced.

1. **SCAN:**

1- Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.

2- Let direction represents whether the head is moving towards left or right.

3- In the direction in which head is moving service all tracks one by one.

4- Calculate the absolute distance of the track from the head.

5- Increment the total seek count with this distance.

6- Currently serviced track position now becomes the new head position.

7- Go to step 3 until we reach at one of the ends of the disk.

8- If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

**Implementation:**

**A) First Come First Serve (FCFS):**

#include <bits/stdc++.h>

using namespace std;

int size = 8;

void FCFS(int arr[], int head)

{

int seek\_count = 0;

int distance, cur\_track;

for (int i = 0; i < size; i++) {

cur\_track = arr[i];

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

cout << "Total number of seek operations = "

<< seek\_count << endl;

cout << "Seek Sequence is" << endl;

for (int i = 0; i < size; i++) {

cout << arr[i] << endl;

}

}

int main()

{

int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };

int head = 50;

FCFS(arr, head);

return 0;

}

**B) SCAN:**

#include <bits/stdc++.h>

using namespace std;

int size = 8;

int disk\_size = 200;

void SCAN(int arr[], int head, string direction)

{

int seek\_count = 0;

int distance, cur\_track;

vector<int> left, right;

vector<int> seek\_sequence;

if (direction == "left")

left.push\_back(0);

else if (direction == "right")

right.push\_back(disk\_size - 1);

for (int i = 0; i < size; i++) {

if (arr[i] < head)

left.push\_back(arr[i]);

if (arr[i] > head)

right.push\_back(arr[i]);

}

std::sort(left.begin(), left.end());

std::sort(right.begin(), right.end());

int run = 2;

while (run--) {

if (direction == "left") {

for (int i = left.size() - 1; i >= 0; i--) {

cur\_track = left[i];

seek\_sequence.push\_back(cur\_track);

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

direction = "right";

}

else if (direction == "right") {

for (int i = 0; i < right.size(); i++) {

cur\_track = right[i];

seek\_sequence.push\_back(cur\_track);

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

direction = "left";

}

}

cout << "Total number of seek operations = "

<< seek\_count << endl;

cout << "Seek Sequence is" << endl;

for (int i = 0; i < seek\_sequence.size(); i++) {

cout << seek\_sequence[i] << endl;

}

}

int main()

{

int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };

int head = 50;

string direction = "left";

SCAN(arr, head, direction);

return 0;

}

Output:

**A) First Come First Serve (FCFS):**

**Input:**

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

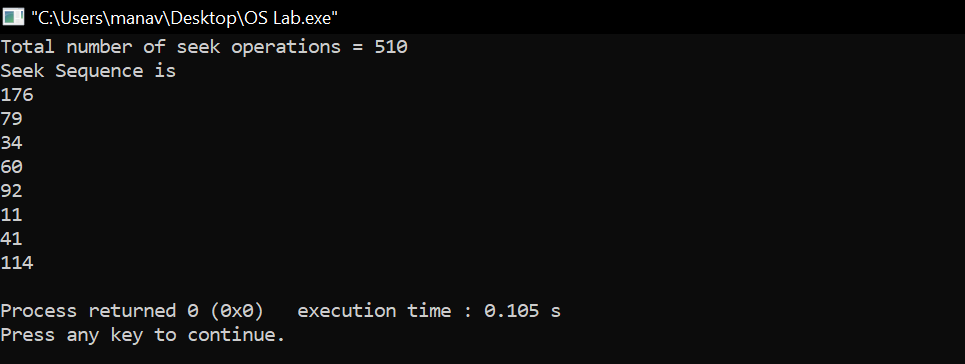
Initial head position = 50

**Output:**

The total seek count is calculated as:

= (176-50)+(176-79)+(79-34)+(60-34)+(92-60)+(92-11)+(41-11)+(114-41)

= 510



**B) SCAN:**

**Input:**

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

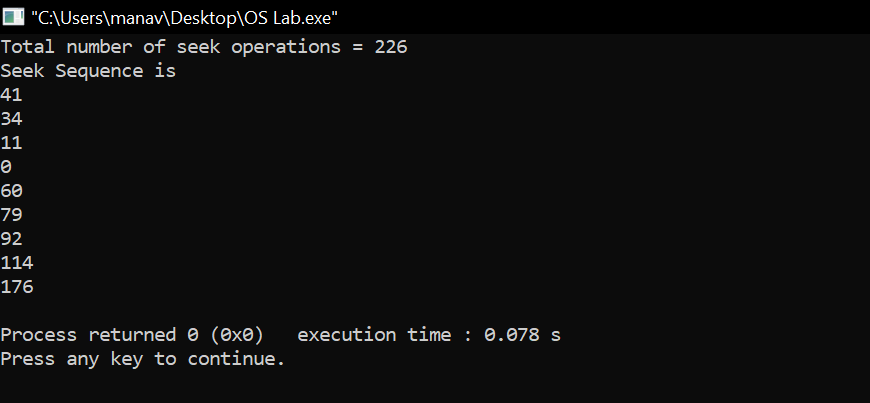
Direction = left (We are moving from right to left)

**Output:**

The total seek count is calculated as:

= (50-41)+(41-34)+(34-11)+(11-0)+(60-0)+(79-60)+(92-79)+(114-92)+(176-114)

= 226



**Learning From The Experiment:**

In FCFS algorithm every request gets a fair chance and there is no indefinite postponement. However, it does not try to optimize seek time. Also, FCFS may not provide the best possible service. FCFS being a nonpreemptive scheduling algorithm, the short processes which are at the back of the queue have to wait for the long process at the front to finish. The throughput of FCFS is not very efficient.

SCAN algorithm is simple and easy to understand. SCAN algorithm have no starvation. This algorithm is better than FCFS Scheduling algorithm. However, it is more complex algorithm to implement. This algorithm is not fair because it cause long waiting time for the cylinders just visited by the head.It causes the head to move till the end of the disk in this way the requests arriving ahead of the arm position would get immediate service but some other requests that arrive behind the arm position will have to wait for the request to complete.