# 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 program for bubble sort on linux Operating system.

**Theory:**

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

This simple algorithm performs poorly in real world use and is used primarily as an educational tool. More efficient algorithms such as quicksort, timesort, or merge sort are used by the sorting libraries built into popular programming.

**Algorithm :**

We assume a list is an array of n elements. We further assume that the swap function swaps the values of the given array elements.

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**Implementation :**

#include<iostream>

using namespace std;

void swap(int\* i, int\* j)

{

int temp = \*i;

\*i = \*j;

\*j = temp;

}

void bubbleSort(int\* arr, int size)

{

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

{

int swapped =0;

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

{

if(arr[j] > arr[j+1])

{

swap(&arr[j], &arr[j+1]);

swapped++;

}

}

if(swapped==0) break;

}

}

int main()

{

int size;

cout<<"Enter size of array"<<endl;

cin>>size;

int arr[size];

cout<<"Enter elements of array"<<endl;

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

{

cin>>arr[i];

}

bubbleSort(arr, size);

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

{

cout<<arr[i]<<" ";

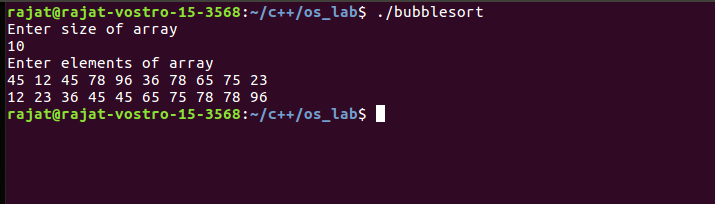
}

cout<<endl;

return 0;

}

**Output :**



**EXPERIMENT - 2**

**Aim :**

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

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

Extend this program 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. pid and ppid

#include<iostream>

#include<unistd.h>

using namespace std;

int main()

{

int processId, parentProcessId;

processId = getpid();

parentProcessId = getppid();

cout<<"Process ID: "<<processId<<endl;

cout<<"Parent Process ID: "<<parentProcessId<<endl;

return 0;

}

1. fork

#include<iostream>

#include<unistd.h>

#include<stdlib.h>

#include<sys/wait.h>

using namespace std;

int main()

{

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

{

if(fork()==0)

{

cout<<"Process ID: "<<getpid()<<", Parent Process ID: "<<getppid()<<endl;

exit(0);

}

}

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

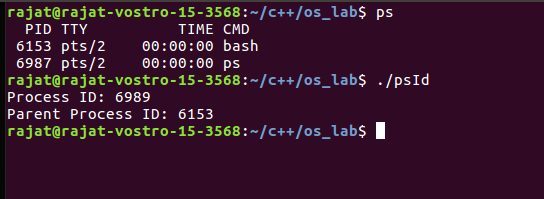
cout<<"Parent Process ID: "<<getpid()<<endl;

return 0;

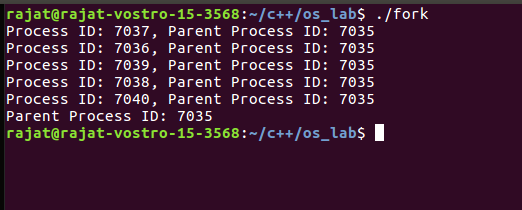
}

**Output :**

1. pid and ppid



1. fork



**EXPERIMENT - 3**

**Aim :**

Write a program to emulate the First Come First Serve Scheduling algorithm. Find the average waiting and turnaround time.

**Theory :**

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.

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.

**Algorithm :**

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 **turnaround 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 turnaround time: **Average turnaround time** = total\_turn\_around\_time / no\_of\_processes

**Implementation :**

#include<stdio.h>

int totalWaitingTime=0;

int totalTurnAroundTime=0;

struct Process

{

int processId;

int completionTime;

int arrivalTime;

int burstTime;

};

typedef struct Process process;

void swap(process\* i, process\* j)

{

process temp = \*i;

\*i = \*j;

\*j = temp;

}

void bubbleSort(process\* arr, int size)

{

int j,i,swapped;

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

{

swapped =0;

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

{

if(arr[j].arrivalTime > arr[j+1].arrivalTime)

{

swap(&arr[j], &arr[j+1]);

swapped++;

}

}

if(swapped==0) break;

}

}

int turnAroundTime(int arrivalTime, int completionTime)

{

return(completionTime - arrivalTime);

}

int waitingTime(int arrivalTime, int completionTime, int burstTime)

{

int temp = turnAroundTime(arrivalTime,completionTime);

totalTurnAroundTime+=temp;

totalWaitingTime+=(temp-burstTime);

return(temp-burstTime);

}

int main()

{

int size = 4,i;

float averageWaitingTime;

float averageTurnAroundTime;

process p[size];

printf("Enter processId, arrivalTime, completionTime, burstTime of process\n");

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

{

scanf("%d%d%d%d",&p[i].processId,&p[i].arrivalTime,&p[i].completionTime,&p[i].burstTime);

}

bubbleSort(p,size)

printf("processId\tarrivalTime\tburstTime\tcompletionTime\tturnAroundTime\twaitingTime\n");

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

{

printf("%-10d\t%-10d\t%-10d\t%-10d\t%-10d\t%-10d\n",p[i].processId, p[i].arrivalTime, p[i].burstTime, p[i].completionTime, turnAroundTime(p[i].arrivalTime,p[i].completionTime), waitingTime(p[i].arrivalTime,p[i].completionTime,p[i].burstTime));

}

averageWaitingTime=totalWaitingTime/(float)size;

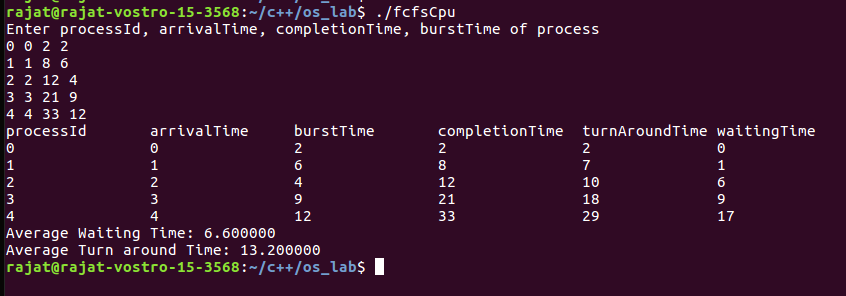
averageTurnAroundTime=totalTurnAroundTime/(float)size;

printf("Average Waiting Time: %f\n",averageWaitingTime);

printf("Average Turn around Time: %f\n",averageTurnAroundTime);

}

**Output :**



**Learning From The Experiment:**

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.

**EXPERIMENT - 4**

**Aim:**

Write a program to emulate Shortest Job First scheduling Algorithm. Find the average waiting time and turnaround time.

**Theory:**

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.

Shortest Job First: Processes 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:**

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 the process we make a pool of processes 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:**

#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;

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

{

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

process\_det[i][0] = i;

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";

}

int waiting = 0;

int turnaround = 0;

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

{

waiting+=process\_det[i][4];

turnaround+=process\_det[i][5];

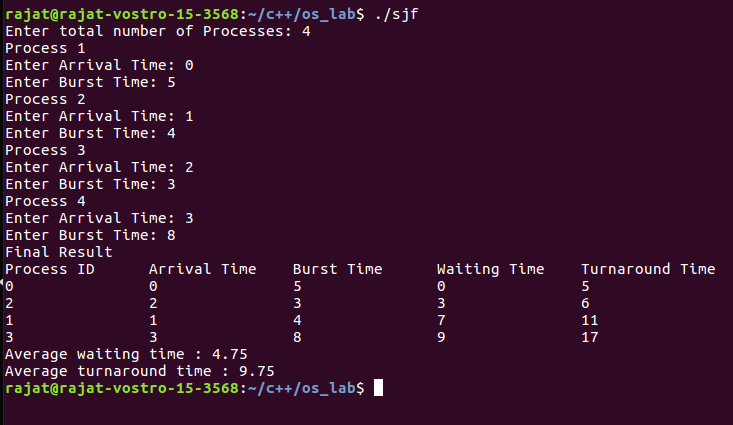
}

cout<<"Average waiting time : "<<waiting/float(num)<<endl;

cout<<"Average turnaround time : "<<turnaround/float(num)<<endl;

}

**Output:**



**Learning From The Experiment:**

For SJF, it gives the minimum waiting time for a given set of processes and thus reduces the average waiting time. A loophole 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 - 5**

**Aim:**

Write a program to emulate the Round Robin Scheduling algorithm. Find the average waiting time and turnaround time.

**Theory :**

### 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.

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.

**Implementation :**

#include<stdio.h>

void findWaitingTime(int\* arr, int\* res, int size, int quantum)

{

int remainingBurstTime[size];

int i,totalTime=0,complete;

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

{

remainingBurstTime[i]=arr[i];

}

while(1)

{

complete=1;

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

{

if(remainingBurstTime[i] > 0)

{

complete=0;

if(remainingBurstTime[i] > quantum)

{

totalTime+=quantum;

remainingBurstTime[i]-=quantum;

}

else

{

totalTime+=remainingBurstTime[i];

res[i]=totalTime-arr[i];

remainingBurstTime[i]=0;

}

}

}

if(complete)

break;

}

}

void findTurnAroundTime(int\* arr, int\* waitingTime, int\* res, int size)

{

int i;

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

{

res[i]=arr[i]+waitingTime[i];

}

}

float findAvg(int\* arr, int size)

{

int sum=0,i;

float avg;

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

{

sum+=arr[i];

}

avg=sum/(float)size;

return avg;

}

void compute(int\* arr, int size, int quantum)

{

int waitingTime[size],turnAroundTime[size],i;

findWaitingTime(arr,waitingTime,size,quantum);

findTurnAroundTime(arr,waitingTime,turnAroundTime,size);

printf("processId\tburstTime\tturnAroundTime\twaitingTime\n");

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

{

printf("%-10d\t%-10d\t%-10d\t%-10d\n",i+1,arr[i],turnAroundTime[i],waitingTime[i]);

}

printf("Average waitingTime: %f\n",findAvg(waitingTime,size));

printf("Average turnAroundTime: %f\n",findAvg(turnAroundTime,size));

}

int main()

{

int size=3,i,quantum;

int processBurstTime[size];

printf("Enter burst time\n");

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

{

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

}

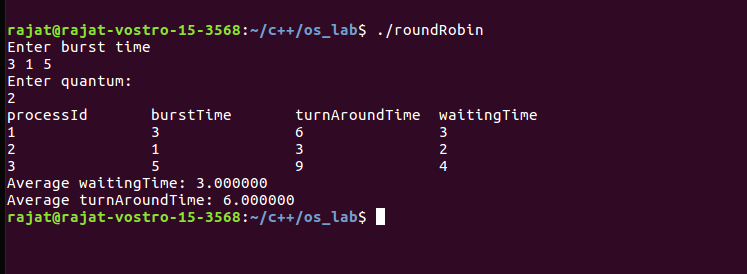
printf("Enter quantum: \n");

scanf("%d",&quantum);

compute(processBurstTime, size, quantum);

}

**Output :**



**Learning From The 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 increase the time.

**EXPERIMENT - 6**

**Aim:**

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

**Theory:**

### 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 attains 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.

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

2- Sort the processes according to arrival time if two process arrival times are the 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:**

#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;

}

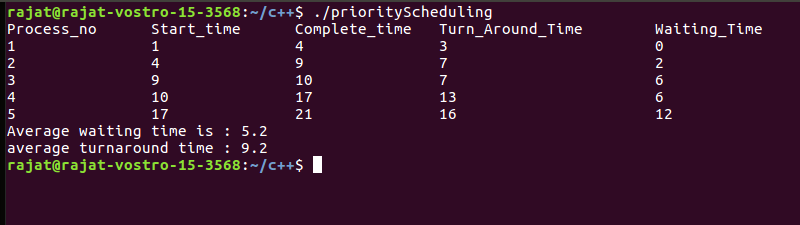
sort(proc,proc+totalprocess,comp);

findgc();

return 0;

}

**Output:**

****

**Learning From The Experiment:**

For Priority Scheduling Algorithms, 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 - 7**

**Aim:**

Write a program to implement all the placement algorithms in the context of memory management.

**Theory:**

### 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.

1. **Next Fit:**

Next fit is a modified version of the first fit. It begins as the first fit to find a free partition but when called next time it starts searching from where it left off, not from the beginning. This policy makes use of a roving pointer. The pointer moves along the memory chain to search for a next fit. This helps to avoid the usage of memory always from the head (beginning) of the free block chain.

**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.

5- If not then keep checking the further blocks.

1. **Next Fit:**

1- Input the number of memory blocks and their sizes and initializes all the blocks as free.

2- Input the number of processes and their sizes.

3- Start by picking each process and check if it can be assigned to the current block, if yes, allocate it the required memory and check for the next process but from the block where we left, not from starting.

4- If the current block size is smaller then keep checking the further blocks.

**Implementation:**

#include<stdio.h>

void bestFit(int\* M, int\* process, int p, int m)

{

int memory[m];

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

{

memory[i]=M[i];

}

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

{

int max=-1;

int index;

int allocated=0;

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

{

if(memory[j] > process[i])

{

allocated=1;

if(max ==-1 || max > memory[j])

{

max=memory[j];

index=j;

}

}

}

if(allocated)

{

memory[index]-=process[i];

printf("%d \t%d \t\t %d\n",i+1,process[i],index);

}

else

{

printf("%d \t%d \t\t None\n",i+1,process[i]);

}

}

}

void firstFit(int\* M, int\* process, int p, int m)

{

int memory[m];

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

{

memory[i]=M[i];

}

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

{

int allocated=0;

int index;

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

{

if(memory[j] > process[i])

{

allocated=1;

index=j;

break;

}

}

if(allocated)

{

memory[index]-=process[i];

printf("%d \t%d \t\t %d\n",i+1,process[i],index);

}

else

{

printf("%d \t%d \t\t None\n",i+1,process[i]);

}

}

}

void worstFit(int\* M, int\* process, int p, int m)

{

int memory[m];

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

{

memory[i]=M[i];

}

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

{

int max=-1;

int index;

int allocated=0;

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

{

if(memory[j] > process[i])

{

allocated=1;

if(max ==-1 || max < memory[j])

{

max=memory[j];

index=j;

}

}

}

if(allocated)

{

memory[index]-=process[i];

printf("%d \t%d \t\t %d\n",i+1,process[i],index);

}

else

{

printf("%d \t%d \t\t None\n",i+1,process[i]);

}

}

}

void nextFit(int\* M, int\* process, int p, int m)

{

int memory[m];

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

{

memory[i]=M[i];

}

int j=0;

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

{

int index;

int count=0;

int allocated=0;

int k = j;

while(j < m)

{

if(memory[j] >= process[i])

{

index=j;

allocated=1;

memory[j]-=process[i];

break;

}

j = (j+1)%m;

if(j == k) break;

}

if(allocated)

{

memory[index]-=process[i];

printf("%d \t%d \t\t %d\n",i+1,process[i],index);

}

else

{

printf("%d \t%d \t\t None\n",i+1,process[i]);

}

}

}

int main()

{

int noOfProcess, noOfMemoryBlock;

printf("Enter no of Memory Block\n");

scanf("%d",&noOfMemoryBlock);

printf("Enter size of Memory Blocks\n");

int memory[noOfMemoryBlock];

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

{

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

}

printf("Enter no of Process\n");

scanf("%d",&noOfProcess);

printf("Enter size of Processes\n");

int process[noOfProcess];

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

{

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

}

printf("-------------------------------Next Fit-------------------------------\n");

printf("Process\tprocess size\t memory block allocated\n");

nextFit(memory,process,noOfProcess,noOfMemoryBlock);

printf("-------------------------------First Fit-------------------------------\n");

printf("Process\tprocess size\t memory block allocated\n");

firstFit(memory,process,noOfProcess,noOfMemoryBlock);

printf("-------------------------------Best Fit--------------------------------\n");

printf("Process\tprocess size\t memory block allocated\n");

bestFit(memory,process,noOfProcess,noOfMemoryBlock);

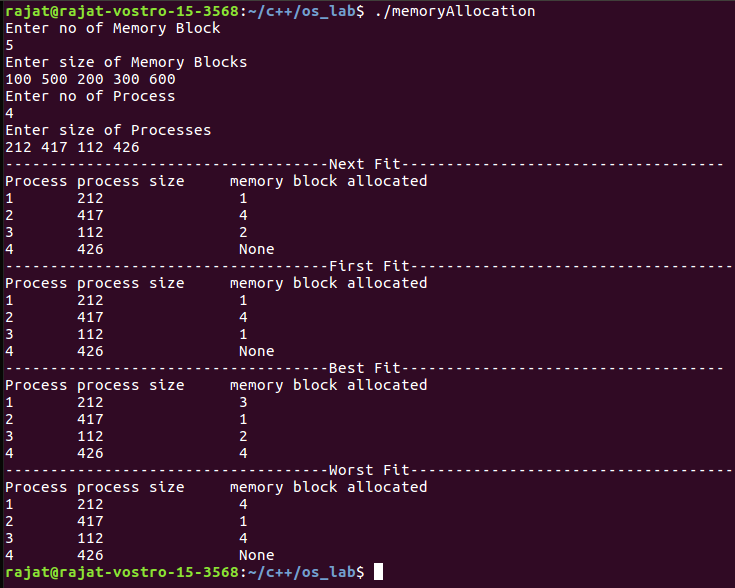
printf("--------------------------Worst Fit------------------------\n");

printf("Process\tprocess size\t memory block allocated\n");

worstFit(memory,process,noOfProcess,noOfMemoryBlock);

}

**Output:**



**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 for the smallest free partition first available.The operating system allocates the 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.

Next fit is a very fast searching algorithm and is also comparatively faster than First Fit and Best Fit Memory Management Algorithms.

**EXPERIMENT - 8**

**Aim:**

Write a program to implement Banker’s algorithm.

**Theory :**

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

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

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

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]

**Finish**: It is the vector of the order m. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

**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

**Implementation:**

#include<stdio.h>

int main()

{

int Process,Resource;

printf("Enter number of processes and resources:\n");

scanf("%d%d",&Process,&Resource);

int allocated[Process][Resource], max[Process][Resource], avail[Resource], need[Process][Resource], flag[Process], safe[Process];

int index=0;

// allocated resources

printf("Enter allocated Resources: \n");

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

{

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

{

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

}

}

//Max resource needed

printf("Enter max Resources needed: \n");

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

{

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

{

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

}

}

//Available resource

printf("Enter Available Resources: \n");

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

{

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

}

//Need of resources

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

{

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

{

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

}

}

printf("Remaining neeed of process: \n");

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

{

printf("P%d -> ",i);

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

{

printf("%d ",need[i][j]);

}

printf("\n");

}

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

{

flag[i] = 0;

}

for (int i = 0; i < Process; i++)//for flag

{

for (int j = 0; j < Process; j++)//for process

{

if(flag[j]==0)

{

int fullfill=1;

for (int k = 0; k < Resource; k++)//For resource

{

if(need[j][k] > avail[k])

{

fullfill=0;

break;

}

}

if(fullfill)

{

safe[index++]=j;

printf("Resource granted to P%d\n",j);

for (int l = 0; l < Resource; l++)

{

avail[l]+=allocated[j][l];

flag[j]=1;

}

}

else

{

printf("Request Denied for Process %d\n",j);

}

}

}

}

// printf("Safe state sequence: \n");

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

{

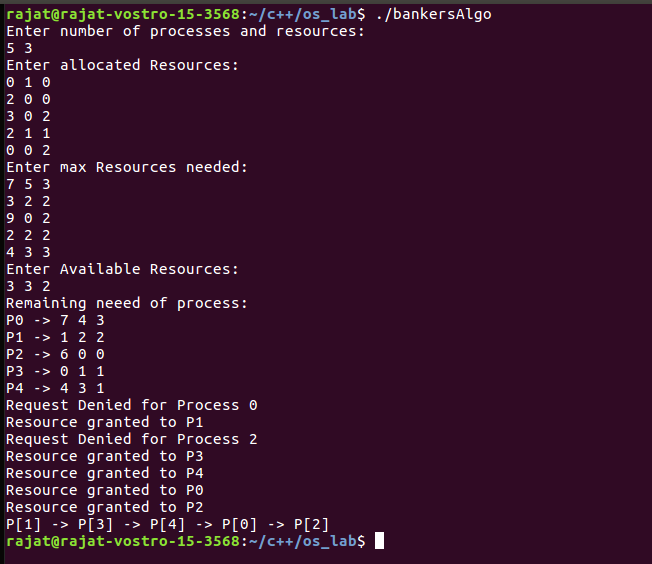
printf("P[%d] -> ",safe[i]);

}

printf("P[%d]\n",safe[Process-1]);

}

**Output:**



**Learning From The Experiment:**

* Banker's algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.
* Notations used in banker's algorithms are 1) Available 2) Max 3) Allocation 4) Need
* Resource request algorithm enables you to represent the system behavior when a specific process makes a resource request.
* Banker's algorithm keeps many resources that satisfy the requirement of at least one client
* The biggest drawback of banker's algorithm is that it does not allow the process to change its Maximum need while processing.

**EXPERIMENT - 9**

**Aim:**

Write a program to implement Page replacement algorithms.

**Theory:**

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 a 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, the 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, the page in the front of the queue is selected for removal.

*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 with which is least recently used.

### 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.

**Algorithms:**

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.

1. Store current page in the queue.
2. 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.

1. 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.

1. Replace the found page with current page.
2. Increment page faults.
3. Update index of current page.

2. Return page faults.

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

1. **Least Frequently Used:**

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.

**Implementation:**

/\*

page sequence size: 20

2 5 10 1 2 2 6 9 1 2 10 2 6 1 2 1 6 9 5 1

frame: 4

\*/

#include<stdio.h>

int n,numberOfFrames;

int inputSequence[100];

int p[50];

int hit=0;

int i,j,k;

int pgfaultcnt=0;

void getData()

{

printf("\nEnter length of page reference sequence:");

scanf("%d",&n);

printf("\nEnter the page reference sequence:");

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

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

printf("\nEnter no of frames:");

scanf("%d",&numberOfFrames);

}

void initialize()

{

pgfaultcnt=0;

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

p[i]=9999;

}

int isHit(int data)

{

hit=0;

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

{

if(p[j]==data)

{

hit=1;

break;

}

}

return hit;

}

int getHitIndex(int data)

{

int hitind;

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

{

if(p[k]==data)

{

hitind=k;

break;

}

}

return hitind;

}

void dispPages()

{

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

{

if(p[k]!=9999)

printf(" %d",p[k]);

}

}

void dispPgFaultCnt()

{

printf("\nTotal no of page faults:%d ",pgfaultcnt);

float percentage = pgfaultcnt/float(n);

printf("\nHit Ratio : %f \n",percentage\*100.0);

}

void fifo()

{

initialize();

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

{

printf("\n For %d :",inputSequence[i]);

if(isHit(inputSequence[i])==0)

{

for(k=0; k<numberOfFrames-1; k++)

p[k]=p[k+1];

p[k]=inputSequence[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault");

}

dispPgFaultCnt();

}

void optimal()

{

initialize();

int near[50];

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

{

printf("\n For %d :",inputSequence[i]);

if(isHit(inputSequence[i])==0)

{

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

{

int pg=p[j];

int found=0;

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

{

if(pg==inputSequence[k])

{

near[j]=k;

found=1;

break;

}

else

found=0;

}

if(!found)

near[j]=9999;

}

int max=-9999;

int repindex;

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

{

if(near[j]>max)

{

max=near[j];

repindex=j;

}

}

p[repindex]=inputSequence[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault");

}

dispPgFaultCnt();

}

void lru()

{

initialize();

int least[50];

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

{

printf("\n For %d :",inputSequence[i]);

if(isHit(inputSequence[i])==0)

{

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

{

int pg=p[j];

int found=0;

for(k=i-1; k>=0; k--)

{

if(pg==inputSequence[k])

{

least[j]=k;

found=1;

break;

}

else

found=0;

}

if(!found)

least[j]=-9999;

}

int min=9999;

int repindex;

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

{

if(least[j]<min)

{

min=least[j];

repindex=j;

}

}

p[repindex]=inputSequence[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault!");

}

dispPgFaultCnt();

}

void lfu()

{

int usedcnt[100];

int least,repin,sofarcnt=0,bn;

initialize();

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

usedcnt[i]=0;

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

{

printf("\n For %d :",inputSequence[i]);

if(isHit(inputSequence[i]))

{

int hitind=getHitIndex(inputSequence[i]);

usedcnt[hitind]++;

printf("No page fault!");

}

else

{

pgfaultcnt++;

if(bn<numberOfFrames)

{

p[bn]=inputSequence[i];

usedcnt[bn]=usedcnt[bn]+1;

bn++;

}

else

{

least=9999;

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

if(usedcnt[k]<least)

{

least=usedcnt[k];

repin=k;

}

p[repin]=inputSequence[i];

sofarcnt=0;

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

if(inputSequence[i]==inputSequence[k])

sofarcnt=sofarcnt+1;

usedcnt[repin]=sofarcnt;

}

dispPages();

}

}

dispPgFaultCnt();

}

int main()

{

getData();

printf("----------------------------------FIFO----------------------------------\n");

fifo();

printf("----------------------------------LRU----------------------------------\n");

lru();

printf("--------------------------------optimal---------------------------------\n");

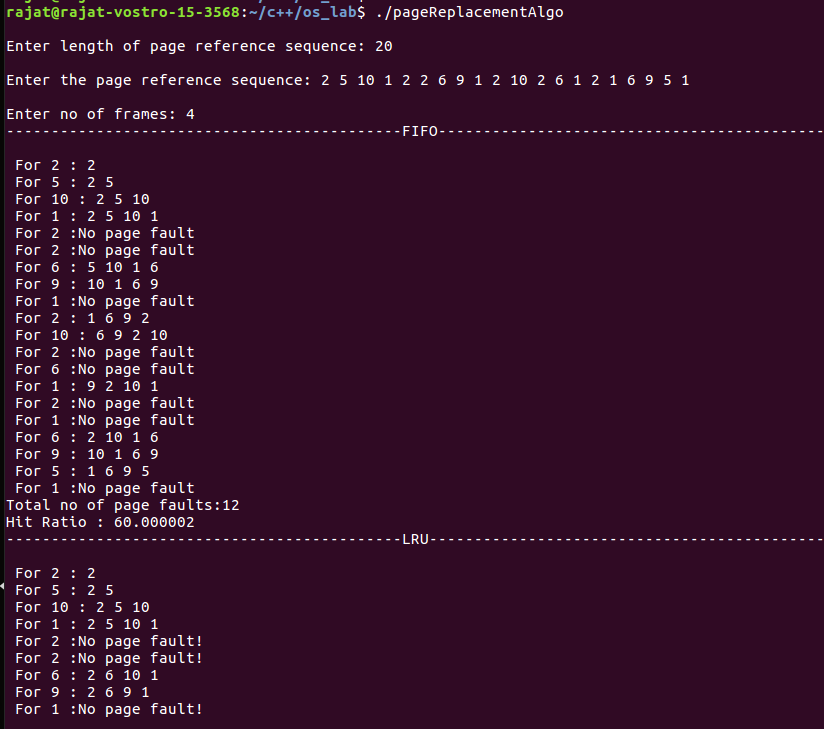
optimal();

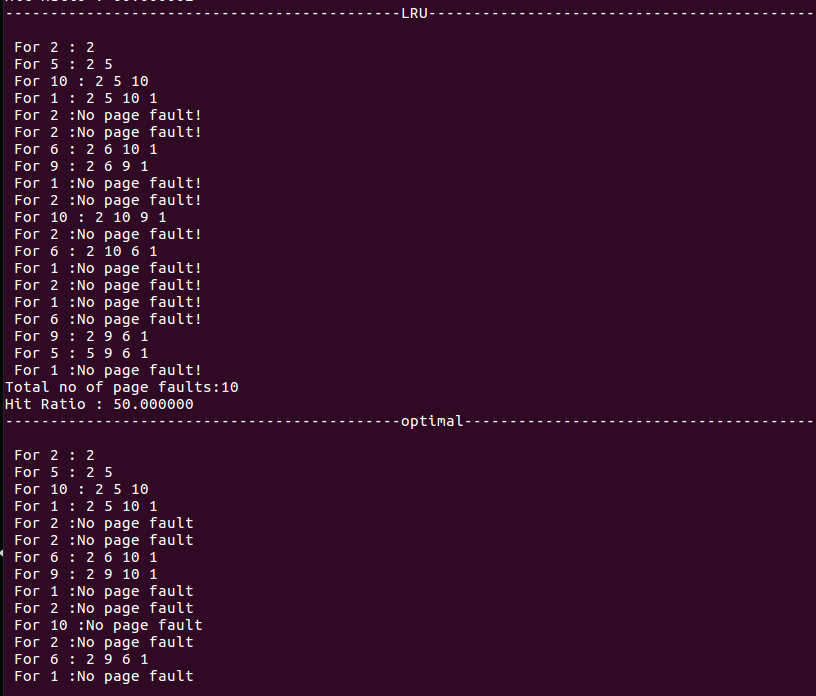
printf("----------------------------------LFU----------------------------------\n");

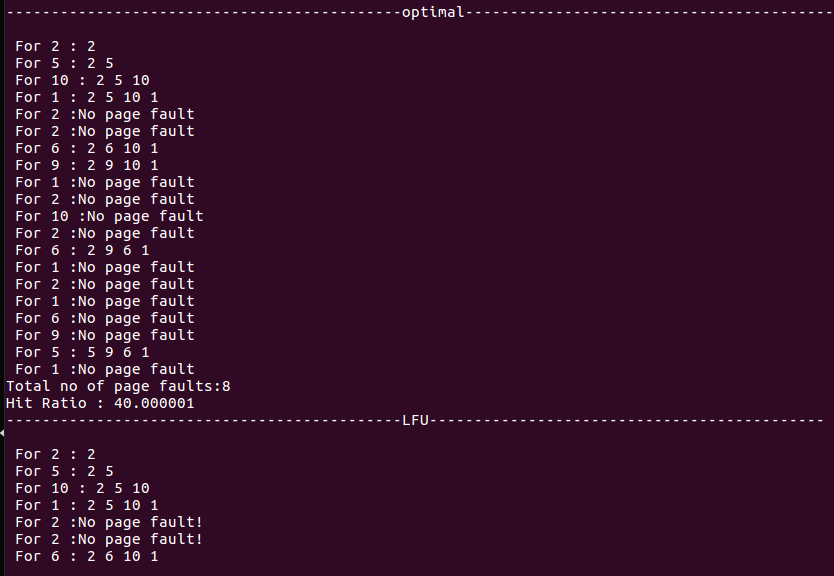
lfu();

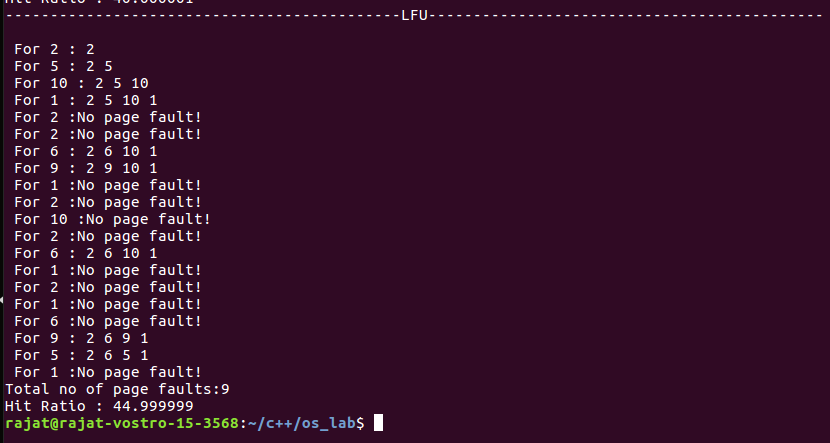
}

**Output:**









**Learning From The Experiment:**

Both the algorithms 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 faults should decrease, but here the page faults are increasing. This problem is called 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 a 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 - 10**

**Aim:**

Write a program to design Dining Philosophers Problem.

**Theory :**

The Dining Philosopher Problem states that K philosophers are seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

**Algorithm:**

do {

wait( chopstick[i] );

wait( chopstick[ (i+1) % 5] );

. .

. EATING THE RICE

.

signal( chopstick[i] );

signal( chopstick[ (i+1) % 5] );

.

. THINKING

.

} while(1);

**Implementation:**

#include<stdio.h>

#define n 4

int compltedPhilo = 0,i;

struct fork{

int taken;

}ForkAvil[n];

struct philosp{

int left;

int right;

}Philostatus[n];

void goForDinner(int philID)

{

if(Philostatus[philID].left==10 && Philostatus[philID].right==10)

printf("Philosopher %d completed his dinner\n",philID+1);

else if(Philostatus[philID].left==1 && Philostatus[philID].right==1)

{

printf("Philosopher %d completed his dinner\n",philID+1);

Philostatus[philID].left = Philostatus[philID].right = 10;

int otherFork = philID-1;

if(otherFork== -1)

otherFork=(n-1);

ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;

printf("Philosopher %d released fork %d and fork %d\n",philID+1,philID+1,otherFork+1);

compltedPhilo++;

}

else if(Philostatus[philID].left==1 && Philostatus[philID].right==0)

{

if(philID==(n-1))

{

if(ForkAvil[philID].taken==0)

{

ForkAvil[philID].taken = Philostatus[philID].right = 1;

printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);

}

else

{

printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);

}

}

else

{

int dupphilID = philID;

philID-=1;

if(philID== -1)

philID=(n-1);

if(ForkAvil[philID].taken == 0)

{

ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;

printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);

}

else

{

printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);

}

}

}

else if(Philostatus[philID].left==0)

{

if(philID==(n-1))

{

if(ForkAvil[philID-1].taken==0)

{

ForkAvil[philID-1].taken = Philostatus[philID].left = 1;

printf("Fork %d taken by philosopher %d\n",philID,philID+1);

}

else

{

printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);

}

}

else

{

if(ForkAvil[philID].taken == 0)

{

ForkAvil[philID].taken = Philostatus[philID].left = 1;

printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);

}

else

{

printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);

}

}

}

}

int main()

{

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

ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n)

{

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

goForDinner(i);

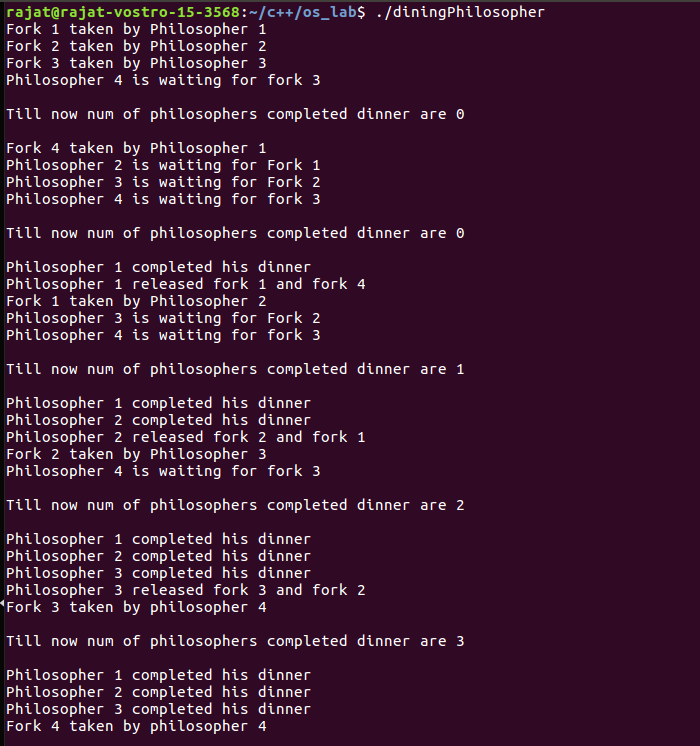
printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);

}

return 0;

}

**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 - 11**

**Aim:**

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

**Theory:**

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 requests 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 requests 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:

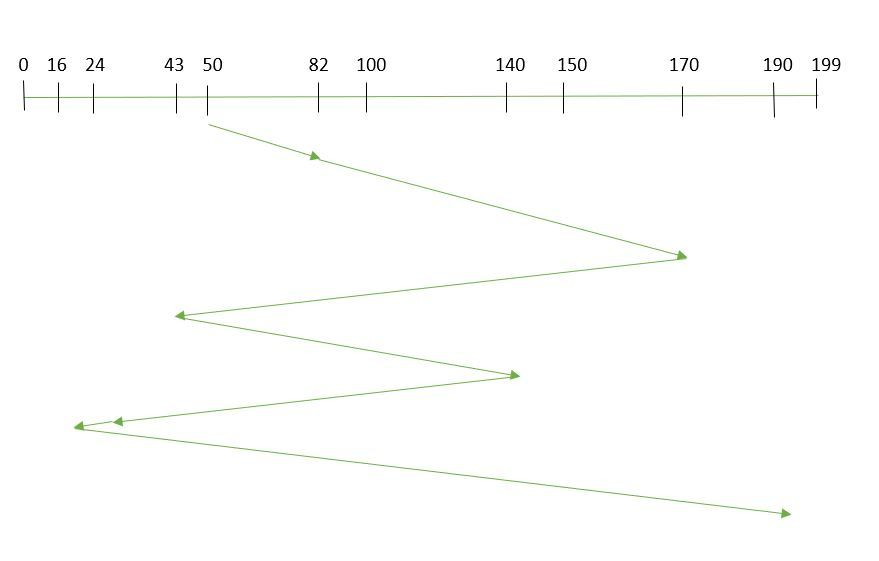
### First Come First Serve (FCFS):

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)

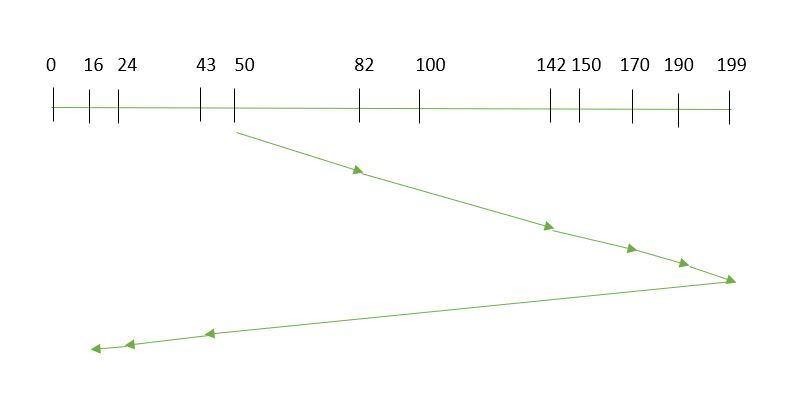
=642

### SCAN:

In the SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of the disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence is 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 the 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 represent whether the head is moving towards left or right.

3- In the direction in which the 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 one of the ends of the disk.

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

Implementation:

#include <bits/stdc++.h>

using namespace std;

int size = 8;

int disk\_size = 200;

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;

}

}

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";

printf("---------------------------------FCFS----------------------------------\n");

FCFS(arr,head);

printf("--------------------------------SCAN----------------------------------\n");

SCAN(arr, head, direction);

return 0;

}

**Output:**



**Learning From The Experiment:**

In the 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.

The SCAN algorithm is simple and easy to understand. SCAN algorithm have no starvation. This algorithm is better than the FCFS Scheduling algorithm. However, it is a 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.