DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

<u>OPERATING SYSTEM DESIGN</u>

CO 204
LAB FILE

SUBMITTED TO
Dr. Prashant Gridhar
Dept. of Computer Science & Eng.
Delhi Technological University

SUBMITTED BY:

Roll No.: 2K21/ CO / 197
Section : A3 Group - G2

INDEX

EXPERIMENT - 1

AIM: Understanding Operating Systems and its Type.

THEORY:

Introduction:

An operating system (OS) is a piece of software that manages and distributes system resources like memory, processors, and storage on behalf of computer hardware and applications. It is the application that controls a computer's resources and offers standard services to other programmes. Also, it controls the connections to add-on hardware like printers and networks. A typical OS service includes managing input and output processes, scheduling tasks, and providing a user interface. Linux is an open-source operating system, whereas Windows and macOS are proprietary ones. Android, iOS, and Windows Mobile are a few examples of operating systems for mobile devices like smartphones and tablets.

Types of Operating System

1. Batch OS
   An operating system for a computer that can process a set of data in batch mode, or without human interaction, is called a batch operating system. Without user input, this kind of technology is made to conduct batches of jobs sequentially. In a batch system, the jobs are often organised into queues and executed in the system's preset sequence. It is one of the oldest varieties of operating system and was popular when mainframes and minicomputers were first introduced. Today, it is still utilised in a few applications, including time-sharing and distributed computing systems.

2. MultiProgramming OS
   An operating system known as a multiprogramming operating system enables numerous programmes to execute simultaneously on a computer system. This is accomplished by allocating processor time among the programmes in a way that allows each programme to run for a specific amount of time before being preempted by another programme. Multiprogramming operating systems are made to use the system's resources as efficiently as possible while boosting

overall throughput. Operating systems including Windows, Linux, Mac OS X, and Solaris are examples of multiprogramming systems.

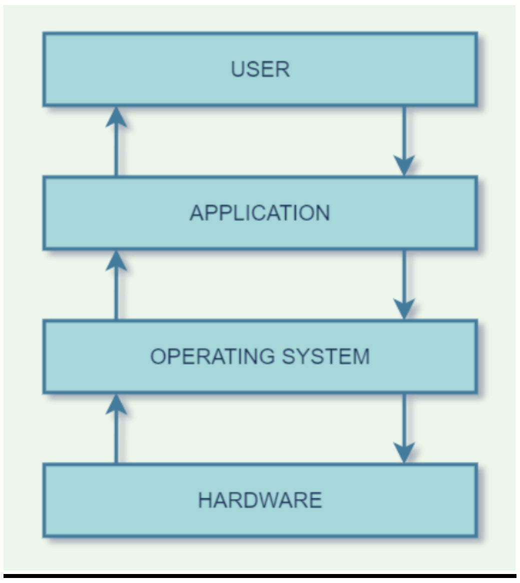
3. Multi Tasking OS
   An operating system that supports running many processes and/or applications simultaneously is known as a multitasking operating system. It enables the computer to move between tasks fast, enabling the user to handle several tasks at once. Operating systems that support many tasks include Linux, Apple's macOS, Microsoft Windows, and IBM's OS/2.

4. Multi Processing OS
   An operating system that supports multiple concurrent processing of numerous jobs, applications, or threads is known as a multiprocessing operating system. It is a sort of operating system that may support several CPUs or processing cores, making job processing more effective and quick. The operating system itself is responsible for managing the distribution of tasks among the different processors and managing the communication and coordination between them.

5. Real Time OS
   Operating systems called real-time operating systems (RTOS) are created expressly to support real-time applications that process data as it enters the system, usually without buffer delays. Depending on the application, it could be necessary to gather, process, and distribute data synchronously or asynchronously within a set time frame. The ability to ensure a specific set of performance parameters, such as a maximum execution time for a given task, is determinism, which is a feature that many RTOSs offer.

Working of an Operating System

The most crucial piece of software that runs on a computer is an operating system (OS). It controls the memory, operations, software, and hardware of the computer. You can converse with the computer using this method even if you don't understand its language.
By interacting with the computer's hardware and software, the operating system does all of these tasks. It



Advantages of Operating System:

1. **Resource Management:** Operating systems manage and allocate system resources such as memory, processor cycles, disk storage, and network bandwidth.

2. **Security**: Operating systems help to protect the system from unauthorized access and malicious activities by providing security features such as user authentication, access control, and firewalls.

3. **Device Management**: Operating systems provide device drivers that enable applications to access hardware devices such as printers, scanners, and network adapters.

4. **File Management**: Operating systems provide a file system that allows users to access and manage files stored on a computer.

5**. User Interface**: Operating systems provide a user interface that allows users to interact with their computer.

Disadvantages of Operating System:

1.**Complexity**: Operating systems can be complex and difficult to understand, especially for new users. This can lead to confusion and frustration when trying to use certain features.

2**.Compatibility Issues**: Different operating systems are not always compatible with each other, which can lead to difficulty in transferring data between them.
3.**Cost**: Operating systems can be expensive to purchase and maintain.
4.**System Updates**: Operating systems need to be regularly updated to keep them secure and running smoothly. This can be time-consuming and costly.

EXPERIMENT - 2

AIM : Write a program to implement FCFS/FIFO and find out the average Turn Around Time and Average Waiting time.

THEORY:

Introduction:

The adage "first come, first served" is frequently employed in a variety of settings, such as business, customer service, and queue management. It is a fundamental tenet that those who come first or are first in line ought to be the ones to receive a service or a good.

Body:

First come, first served can be applied in a professional context to control client interactions. For instance, a store might establish a rule that the first person to enter the store will receive assistance. Customers will be able to know that they will be serviced in the order in which they arrive, which can help with managing customer flow. First come, first serve can be applied to customer service to handle requests or complaints from clients. For instance, a customer service agent may have a rule that, regardless of the seriousness of the problem or the urgency of the request, they will attend to the first client that contacts them.
This can be useful for controlling consumer expectations and making sure that clients are treated fairly and consistently.
First come, first served can be applied to queue management to control the flow of consumers or clients. For instance, a store may have a rule stating that patrons will be assisted in the order of their arrival. This can be useful for controlling customer flow and making sure that clients are treated fairly and consistently.
It's crucial to understand that P2 has a waiting time of 5, as it begins running at time 5 after P1 has finished, and since P1's completion time is 25, waiting for P2 requires waiting for 20 - 15 = 5 units of time. Similar to P1 and P2, P3 begins running at time 8 and finishes at time 37, requiring it to wait for 29 - 12 = 17 units of time.

FCFS Algorithm with code:

FCFS (First Come First Serve) is a scheduling algorithm that processes tasks in the order they are received, without any priority or preference given to specific tasks.

CODE :

```
#include<iostream>
```

```
using namespace std;
void WaitingTime(int processes[], int n,
                 int bt[], int wt[])
{

   wt[0] = 0;
   // calculating waiting time
   for (int i = 1; i < n ; i++ )
      wt[i] = bt[i-1] + wt[i-1] ;
}

void TurnAroundTime( int processes[], int n,
            int bt[], int wt[], int tat[])
{
   // calculating turnaround time by adding
   // bt[i] + wt[i]
   for (int i = 0; i < n ; i++)
      tat[i] = bt[i] + wt[i];
}

//Function to calculate average time
void avgTime( int processes[], int n, int bt[])
{
   int wt[n], tat[n], total_wt = 0, total_tat = 0;

   //Function to find waiting time of all processes
   WaitingTime(processes, n, bt, wt);

   //Function to find turn around time for all processes
   TurnAroundTime(processes, n, bt, wt, tat);

   //Display processes along with all details
   cout << "Processes "<< " Burst time "
      << " Waiting time " << " Turn around time\n";

   // Calculate total waiting time and total turn
   // around time
   for (int i=0; i<n; i++)
   {
      total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
      cout << " " << i+1 << bt[i] <<"\t "
         << wt[i] << 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};
    avgTime(processes, n, burst_time);
    return 0;
}
```

OUTPUT :

```
Processes  Burst time  Waiting time  Turn around time
 110     010
 25      1015
 38      1523
Average waiting time = 8.33333
Average turn around time = 16
```

Advantage of FCFS :

The first come, first served guiding principle has some benefits and drawbacks. It has the benefit of being straightforward and simple to comprehend. Clients can schedule their visits accordingly because they are aware that they will be attended to in the sequence in which they arrive. Customers' misunderstanding and dissatisfaction may be lessened as a result. First come, first served can be utilised to control consumer flow, which is another benefit. For instance, a store can use first come, first serve to guarantee that customers are serviced promptly if it anticipates being busy at a particular time of day. Customers' wait times may be cut down as a result, and customer satisfaction levels may also increase.

Disadvantage of FCFS :

First come, first served has the drawback of perhaps being unfair to clients. For instance, if a client enters a store and is the first to be served, but they have a more urgent or complicated issue than the following customer, the subsequent customer may be served before the first customer is done. The first client may become frustrated as a result, and the following customer may experience longer wait times. First come, first served has the additional drawback of being challenging to handle in some circumstances. For example, if a store is very busy, it may be difficult to keep track of who arrived first and who should be served next. This can lead to confusion and frustration for customers and employees alike

EXPERIMENT – 3

AIM : Write a program to perform shortest job first (non-preemptive scheduling).

Theory:

Introduction:

A non-preemptive scheduling approach called Shortest Job First (SJF) gives the CPU the task with the smallest burst time. A new job is added to the ready queue when it comes in according to its burst time. The job with the smallest burst time is given the CPU when it is available. The Shortest Process Next algorithm is another name for it (SPN). You can take the following actions to implement SJF in a non-primitive scheduling environment:

1. Create a ready queue to hold the jobs that are waiting to be executed.
2. When a new job arrives, sort the ready queue in increasing order of burst time.
3. When the CPU is available, assign the job with the shortest burst time to the CPU.
4. Once a job is completed, remove it from the ready queue and repeat steps 2 and 3.

It is important to note that SJF is based on the assumption that the burst time of a job is known in advance. In reality, this is often not the case, and thus, SJF is not commonly used in practice.

SJF Algorithm with Code:

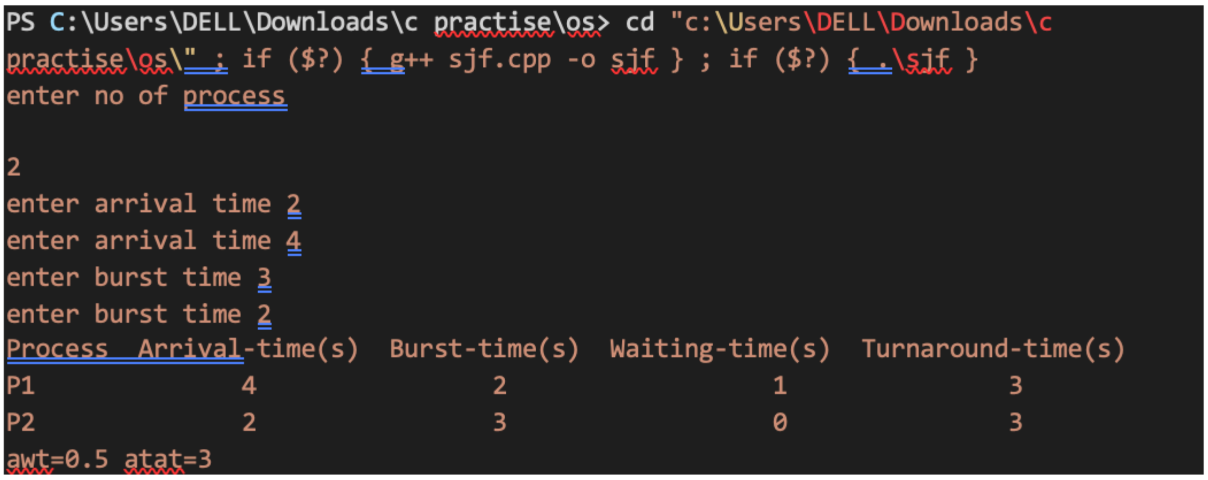
```
#include<iostream>
using namespace std;
int main()
{
    int n,temp,tt=0,min,d,i,j;
    float atat=0,awt=0,stat=0,swt=0;
    cout<<"enter no of process"<<endl;
    cin>>n;
    int a[n],b[n],e[n],tat[n],wt[n];

    for(i=0;i<n;i++)
    {
        cout<<"enter arival time ";       //input
        cin>>a[i];
    }
    for(i=0;i<n;i++)
    {
        cout<<"enter brust time ";      //input
        cin>>b[i];
    }
    for(i=0;i<n;i++)
    {
```

```
  for(j=i+1;j<n;j++)
   {
        if(b[i]>b[j])
        {
             temp=a[i];
             a[i]=a[j];
             a[j]=temp;

             temp=b[i];
             b[i]=b[j];
             b[j]=temp;
        }
   }
}
min=a[0];
for(i=0;i<n;i++)
{
     if(min>a[i])
     {
          min=a[i];
          d=i;
     }
}
tt=min;
e[d]=tt+b[d];
tt=e[d];

for(i=0;i<n;i++)
{
     if(a[i]!=min)
     {
          e[i]=b[i]+tt;
          tt=e[i];
     }
}
for(i=0;i<n;i++)
{

     tat[i]=e[i]-a[i];
     stat=stat+tat[i];
     wt[i]=tat[i]-b[i];
     swt=swt+wt[i];
}
atat=stat/n;
awt=swt/n;
```

```
    cout<<"Process  Arrival-time(s)  Burst-time(s)  Waiting-time(s)  Turnaround-time(s)\n";

   for(i=0;i<n;i++)
   {
   cout<<"P"<<i+1<<"           "<<a[i]<<"            "<<b[i]<<"             "<<wt[i]<<"            "<<tat[i]<<endl;
   }

   cout<<"awt="<<awt<<" atat="<<atat;  //average waiting time and turn around time
}
```

This programme accepts a list of jobs, each of which is expressed as a list including the job's arrival time and burst time. It iterates through the jobs, adding the burst time of the current job to the completion time, and figuring out the turnaround time for the current job as the completion time minus the arrival time. The jobs are sorted by their burst times in ascending order. The function gives the aggregate turnaround time for all jobs.

OUTPUT:



```
PS C:\Users\DELL\Downloads\c practise\os> cd "c:\Users\DELL\Downloads\c practise\os\" ; if ($?) { g++ sjf.cpp -o sjf } ; if ($?) { .\sjf }
enter no of process

2
enter arrival time 2
enter arrival time 4
enter burst time 3
enter burst time 2
Process  Arrival-time(s)  Burst-time(s)  Waiting-time(s)  Turnaround-time(s)
P1              4                2                  1                3
P2              2                3                  0                3
awt=0.5 atat=3
```

Applications of SJF (Shortest Job First)

Operating Systems:

Operating systems frequently employ the scheduling technique known as shortest job first (SJF) to choose the sequence in which tasks should be completed. System efficiency is improved and the overall completion time of all procedures is reduced.

SJF is used in job scheduling systems to rank tasks according to how long it will take them to complete. This guarantees that tasks with lower execution times are finished first, cutting down on the time it takes for all tasks to be finished in total. SJF is used in cloud computing environments to optimise the resource distribution to virtual machines. This lowers expenses while enhancing the performance of the cloud environment. Choosing the best route for data packets to take through the network is done with SJF in network routing.

Advantages of SJF:

1.High throughput: shortest job first (SJF) algorithm ensures that the CPU is always working on the shortest job, thus maximizing the throughput of the system.
2.Low waiting time: As the shortest job is executed first, the waiting time for other jobs is minimized. This results in a more efficient use of system resources.
3.High CPU utilization: As the CPU is always working on the shortest job, it is more likely to be in a busy state, resulting in high CPU utilization.

Disadvantages of SJF:

1.Difficult to predict job length: It is difficult to predict the length of a job, which can lead to inaccuracies in the scheduling algorithm.
2.Starvation of long jobs: longer jobs may be starved and may not be executed for a long time, resulting in poor performance.
3.No support for priority: SJF does not consider the priority of a job and only focuses on the job's length, which may lead to lower-priority jobs being executed before higher-priority jobs.

EXPERIMENT - 4

AIM : Write a program to perform longest job first (non-preemptive scheduling).

Theory:

Introduction:

In computer systems and operating systems, the scheduling method known as the Longest Job First (LJF) algorithm is employed. The fundamental idea behind LJF is to assign the longest job first so that shorter projects can be finished more rapidly. The LJF procedure operates as follows:

1. All jobs are first sorted by their length or burst time, with the longest job at the top of the list.
2. The longest job is assigned to the CPU, and it starts executing until it finishes or is pre-empted.
3. If a new job arrives while the current job is still executing, it is added to the list of jobs.
4. Once the current job finishes, the next job on the list is assigned to the CPU, and the process repeats until all jobs are completed.
5. The algorithm ensures that the longest job is always assigned first, ensuring that it has maximum CPU time and finishes first.

LJF is frequently used in batch processing systems, where the objective is to finish a lot of jobs in a short period of time. It can also be applied to real-time systems, when meeting deadlines is the main objective. In some systems, LJF can, however, result in increased wait times for shorter jobs, which can be a drawback. Also, a new job that has a burst time that is significantly longer than the one that is already running can cause a large delay for that job.

LJF Algorithm :

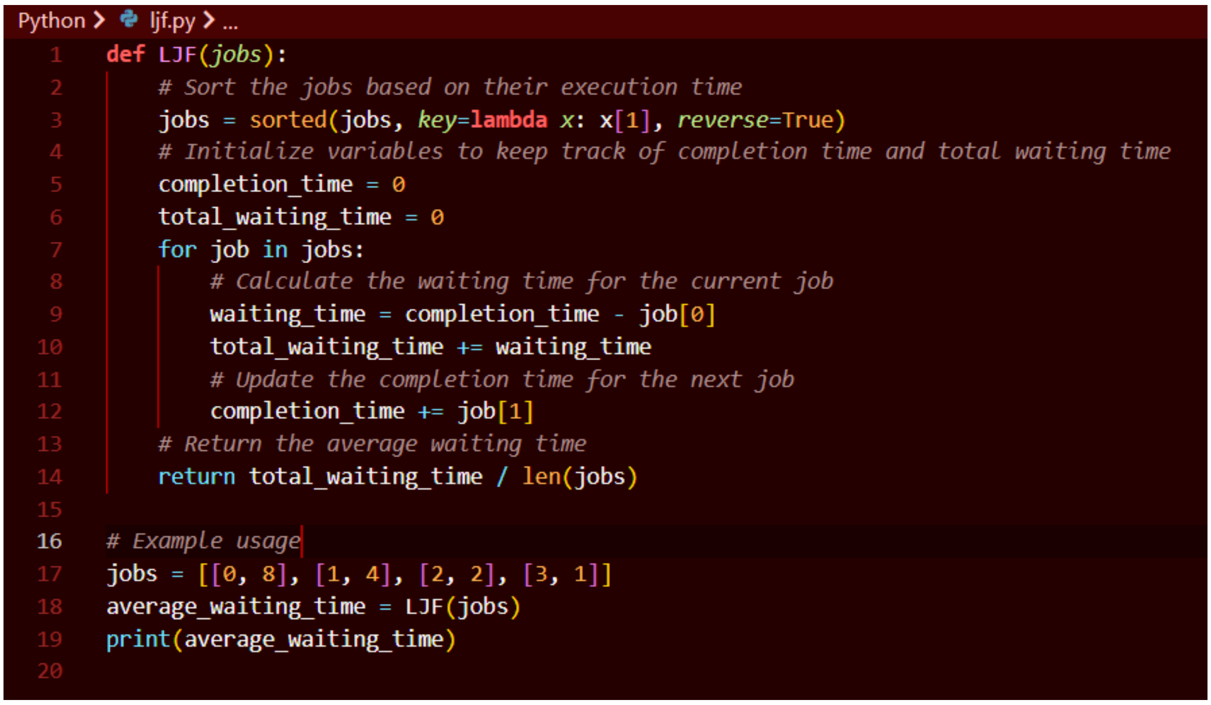
Step-1: First, sort the processes in increasing order of their Arrival Time.

Step 2: Choose the process having the highest Burst Time among all the processes that have arrived till that time.

Step 3: Then process it for its burst time. Check if any other process arrives until this process completes execution.

Step 4: Repeat the above three steps until all the processes are executed.

CODE:

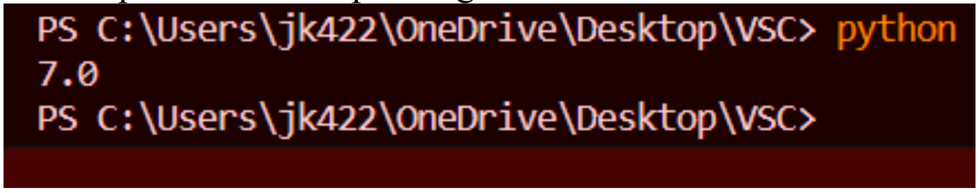


```
Python > ljf.py > ...
def LJF(jobs):
    # Sort the jobs based on their execution time
    jobs = sorted(jobs, key=lambda x: x[1], reverse=True)
    # Initialize variables to keep track of completion time and total waiting time
    completion_time = 0
    total_waiting_time = 0
    for job in jobs:
        # Calculate the waiting time for the current job
        waiting_time = completion_time - job[0]
        total_waiting_time += waiting_time
        # Update the completion time for the next job
        completion_time += job[1]
    # Return the average waiting time
    return total_waiting_time / len(jobs)

# Example usage
jobs = [[0, 8], [1, 4], [2, 2], [3, 1]]
average_waiting_time = LJF(jobs)
print(average_waiting_time)
```

In this example, the input to the LJF function is a list of jobs, where each job is represented as a tuple with two elements: the arrival time and the execution time. The function first sorts the jobs based on their execution time in descending order, and then calculates the average waiting time for all the jobs.

OUTPUT :

The output of the example usage is:



```
PS C:\Users\jk422\OneDrive\Desktop\VSC> python
7.0
PS C:\Users\jk422\OneDrive\Desktop\VSC>
```

This algorithm is simple to implement and understand, however it is not always the best choice for scheduling as it doesn’t consider the arrival time of the process.

Advantages of Longest Job First(LJF):

1. No other process can execute until the longest job or process executes completely.
2. All the jobs or processes finish at the same time approximately.

Disadvantages of Longest Job First CPU Scheduling Algorithm:

1. This algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
2. This may lead to a convoy effect.

3. It may happen that a short process may never get executed and the system keeps on executing the longer processes.
4. It reduces the processing speed and thus reduces the efficiency and utilization of the system.

EXPERIMENT - 5

AIM : Write a program to perform priority scheduling (non-preemptive scheduling).

Theory:

Introduction:
Priority scheduling is a scheduling algorithm used in computer operating systems to determine the order in which processes are executed. It is a non-pre-emptive scheduling algorithm, which means that once a process has started running, it continues to run until it finishes or until it voluntarily gives up control.

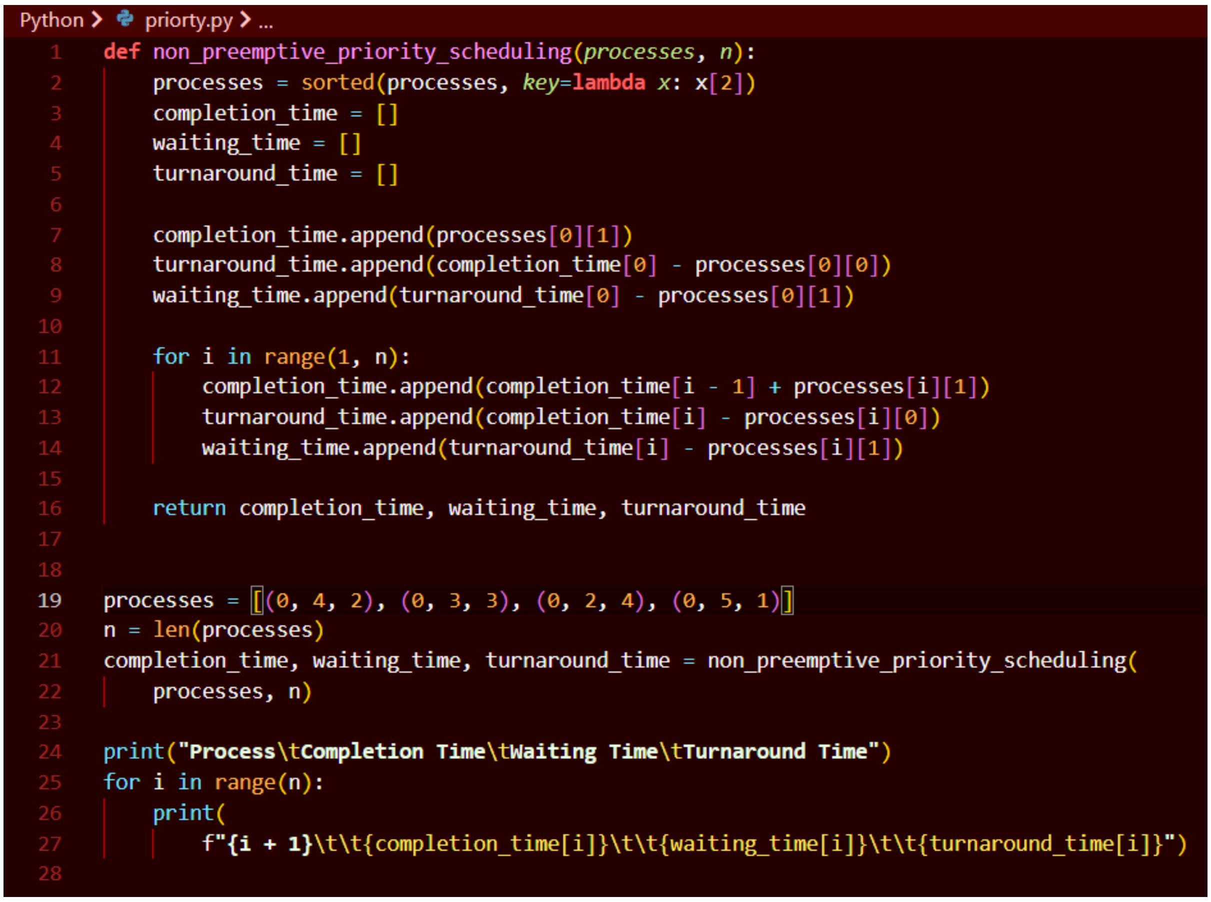
In priority scheduling, each process is assigned a priority number. Processes with higher priority numbers are executed first, while processes with lower priority numbers are executed later. If two or more processes have the same priority, they are executed in a round-robin fashion.

One advantage of priority scheduling is that it provides a means to control the execution order of processes, ensuring that important processes are executed first. However, it can also lead to starvation, where lower priority processes may never get a chance to run if higher priority processes are constantly executing. To avoid this, a aging technique can be used, where the priority of processes increases as they wait in the queue.

Overall, priority scheduling is an effective scheduling algorithm that balances the needs of different processes, while ensuring that important processes are executed first.

Priority (Non-Pre-emptive) Algorithm with Code

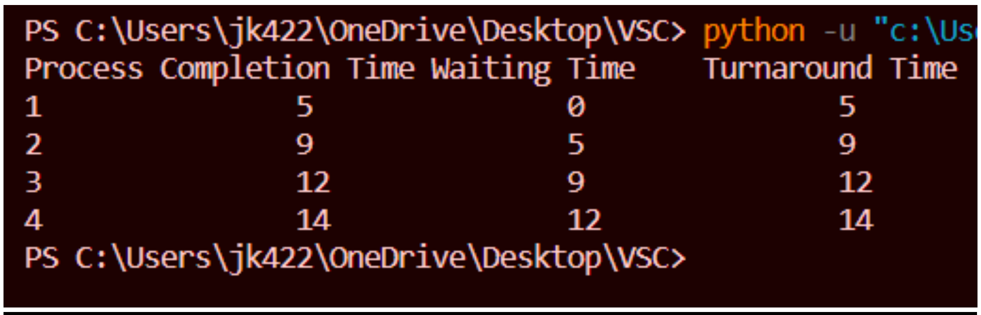
Here's an implementation of non-pre-emptive priority scheduling algorithm in Python, assuming all processes have 0 arrival time:


Python > priorty.py > ...

```python
def non_preemptive_priority_scheduling(processes, n):
    processes = sorted(processes, key=lambda x: x[2])
    completion_time = []
    waiting_time = []
    turnaround_time = []

    completion_time.append(processes[0][1])
    turnaround_time.append(completion_time[0] - processes[0][0])
    waiting_time.append(turnaround_time[0] - processes[0][1])

    for i in range(1, n):
        completion_time.append(completion_time[i - 1] + processes[i][1])
        turnaround_time.append(completion_time[i] - processes[i][0])
        waiting_time.append(turnaround_time[i] - processes[i][1])

    return completion_time, waiting_time, turnaround_time


processes = [(0, 4, 2), (0, 3, 3), (0, 2, 4), (0, 5, 1)]
n = len(processes)
completion_time, waiting_time, turnaround_time = non_preemptive_priority_scheduling(
    processes, n)

print("Process\tCompletion Time\tWaiting Time\tTurnaround Time")
for i in range(n):
    print(
        f"{i + 1}\t\t{completion_time[i]}\t\t{waiting_time[i]}\t\t{turnaround_time[i]}")
```

In this code, processes are a list of tuples, where each tuple represents a process with its burst time (process [1]) and priority (process [2]). The output is the completion time, waiting time, and turnaround time for each process, respectively.

Output



```
PS C:\Users\jk422\OneDrive\Desktop\VSC> python -u "c:\Us
Process Completion Time Waiting Time     Turnaround Time
1               5               0               5
2               9               5               9
3               12              9               12
4               14              12              14
PS C:\Users\jk422\OneDrive\Desktop\VSC>
```

Advantages:

- Easy to use.
- Processes with higher priority execute first which saves time.
- The importance of each process is precisely defined.
- A good algorithm for applications with fluctuating time and resource requirements.

Disadvantages:

- We can lose all the low-priority processes if the system crashes.
- This process can cause starvation if high-priority processes take too much CPU time. The lower priority process can also be postponed for an indefinite time.
- There is a chance that a process can't run even when it is ready as some other process is running currently.

EXPERIMENT - 6

AIM : Write a program to perform Highest Response Ratio Next (HRRN) CPU Scheduling (non-preemptive scheduling).

Theory:

Introduction:

Given N processes with their Arrival times and Burst times, the task is to find average waiting time and an average turn around time using HRRN scheduling algorithm.
The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

Characteristics of HRRN CPU Scheduling:

- Highest Response Ratio Next is a non-preemptive CPU Scheduling algorithm and it is considered as one of the most optimal scheduling algorithm.
- The criteria for HRRN is Response Ratio, and the mode is Non-Preemptive.
- HRRN is basically considered as the modification of Shortest Job First in order to reduce the problem of starvation.
- In comparison with SJF, during HRRN scheduling algorithm, the CPU is allotted to the next process which has the highest response ratio and not to the process having less burst time.

Implementation of HRRN Scheduling –

- Input the number of processes, their arrival times and burst times.
- Sort them according to their arrival times.
- At any given time calculate the response ratios and select the appropriate process to be scheduled.
- Calculate the turn around time as completion time – arrival time.
- Calculate the waiting time as turn around time – burst time.
- Turn around time divided by the burst time gives the normalized turn around time.
- Sum up the waiting and turn around times of all processes and divide by the number of processes to get the average waiting and turn around time.

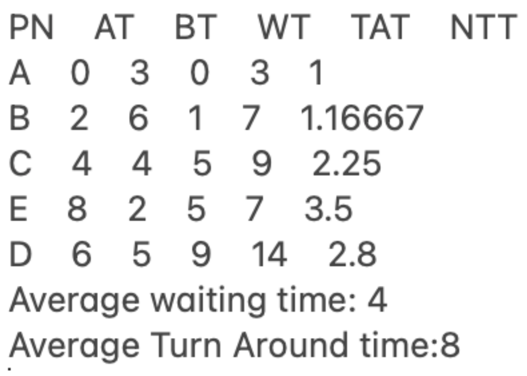
CODE:

```
// C++ program for Highest Response Ratio Next (HRRN)
// Scheduling
#include <bits/stdc++.h>
using namespace std;
// Defining process details
struct process {
        char name;
        int at, bt, ct, wt, tt;
        int completed;
        float ntt;
} p[10];

int n;

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

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

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

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

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

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

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

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

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

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

// Sorting the structure by arrival times
sortByArrival();
cout << "PN\tAT\tBT\tWT\tTAT\tNTT";
for (t = p[0].at; t < sum_bt;) {

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

        // Response Ratio Variable
        float temp;

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

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

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

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

                        // Storing Response Ratio
                        hrr = temp;

                        // Storing Location
                        loc = i;
                    }
                }
            }

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

            // Calculation of waiting time
            p[loc].wt = t - p[loc].at - p[loc].bt;

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

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

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

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

            // Sum Waiting Time for average
            avgwt += p[loc].wt;
            cout << "\n" << p[loc].name << "\t" << p[loc].at;
            cout << "\t" << p[loc].bt << "\t" << p[loc].wt;
            cout << "\t" << p[loc].tt << "\t" << p[loc].ntt;
        }
        cout << "\nAverage waiting time: " << avgwt / n << endl;
        cout << "Average Turn Around time:" << avgtt / n;
}
```

OUTPUT :



```
PN   AT   BT   WT   TAT   NTT
A   0   3   0   3   1
B   2   6   1   7   1.16667
C   4   4   5   9   2.25
E   8   2   5   7   3.5
D   6   5   9   14   2.8
Average waiting time: 4
Average Turn Around time:8
```

Advantages of HRRN CPU Scheduling

- HRRN Scheduling algorithm generally gives better performance than the shortest job first Scheduling.
- There is a reduction in waiting time for longer jobs and also it encourages shorter jobs.

Disadvantages of HRRN CPU Scheduling

- The on ground implementation of HRRN scheduling is not possible as it is not possible know the burst time of every job in advance.
- In this scheduling, there may occur overload on the CPU.

EXPERIMENT - 7

AIM : Write a program to perform Shortest Remaining Time First, SRTF (pre-emptive scheduling).

Theory:

Introduction:

In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Some of the key characteristics of SRTF

- **Preemptive:** SRTF is a preemptive algorithm, which means that the currently running process can be interrupted if a new process arrives with a shorter burst time. This helps in ensuring that the processes with the shortest burst times are executed first.
- **Dynamic:** SRTF is a dynamic algorithm, which means that it can adapt to changes in the arrival time and burst time of processes. It constantly re-evaluates the remaining burst time of each process and schedules the process with the shortest remaining time.
- **Low waiting time:** SRTF is known for its low waiting time. By selecting the process with the shortest remaining burst time, it ensures that the processes with the shortest burst times are executed first, which reduces the average waiting time of processes.

ALGORITHM:

```
  - Traverse until all process gets completely executed.
- Find process with minimum remaining time at every single time lap.
- Reduce its time by 1.
- Check if its remaining time becomes 0
- Increment the counter of process completion.
- Completion time of current process = current_time + 1;
```

- Calculate waiting time for each completed process.
  wt[i]= Completion time – arrival_time-burst_time
- Increment time lap by one.
  - Find turnaround time (waiting_time + burst_time).

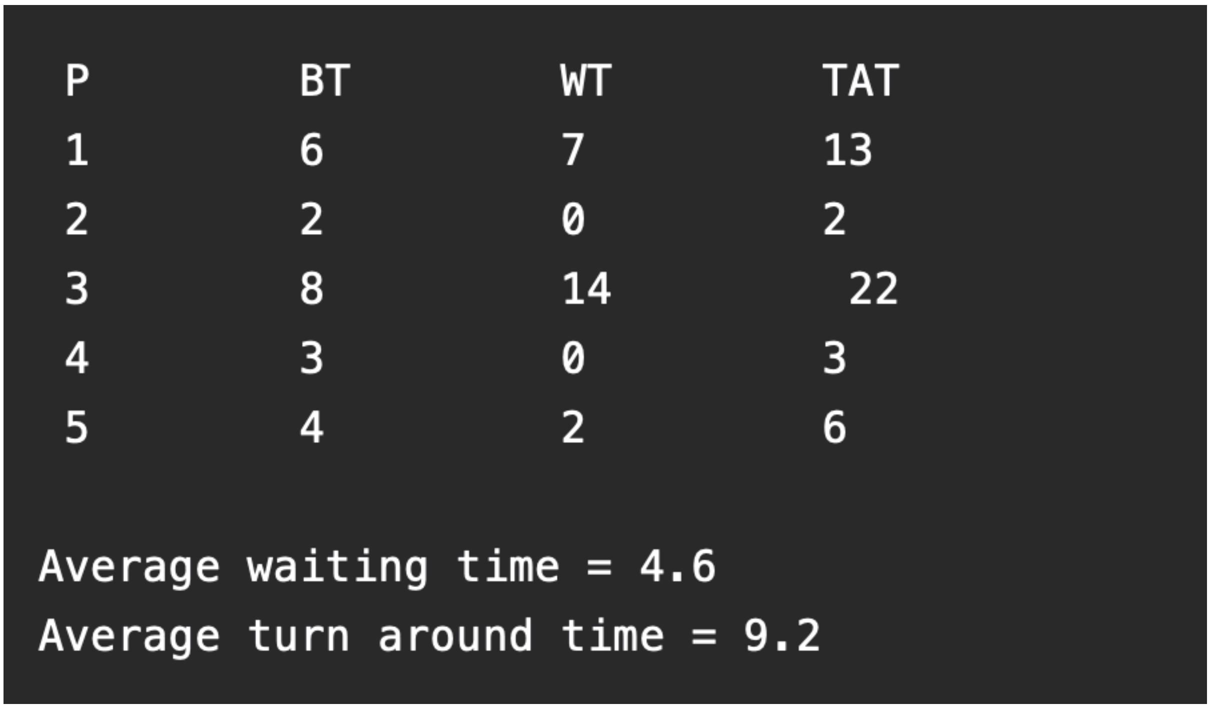
CODE :

```
// C++ program to implement Shortest Remaining Time First
// Shortest Remaining Time First (SRTF)

#include <bits/stdc++.h>
using namespace std;

struct Process {
	int pid; // Process ID
	int bt; // Burst Time
	int art; // Arrival Time
};

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n,
								int wt[])
{
	int rt[n];

	// Copy the burst time into rt[]
	for (int i = 0; i < n; i++)
		rt[i] = proc[i].bt;

	int complete = 0, t = 0, minm = INT_MAX;
	int shortest = 0, finish_time;
	bool check = false;

	// Process until all processes gets
	// completed
	while (complete != n) {

		// Find process with minimum
		// remaining time among the
		// processes that arrives till the
		// current time`
		for (int j = 0; j < n; j++) {
			if ((proc[j].art <= t) &&
			(rt[j] < minm) && rt[j] > 0) {
```

```
                    minm = rt[j];
                    shortest = j;
                    check = true;
               }
          }

          if (check == false) {
               t++;
               continue;
          }

          // Reduce remaining time by one
          rt[shortest]--;

          // Update minimum
          minm = rt[shortest];
          if (minm == 0)
               minm = INT_MAX;

          // If a process gets completely
          // executed
          if (rt[shortest] == 0) {

               // Increment complete
               complete++;
               check = false;

               // Find finish time of current
               // process
               finish_time = t + 1;

               // Calculate waiting time
               wt[shortest] = finish_time -
                              proc[shortest].bt -
                              proc[shortest].art;

               if (wt[shortest] < 0)
                    wt[shortest] = 0;
          }
          // Increment time
          t++;
     }
}

// Function to calculate turn around time
void findTurnAroundTime(Process proc[], int n,
                                                  int wt[], int tat[])
{
        // calculating turnaround time by adding
        // bt[i] + wt[i]
        for (int i = 0; i < n; i++)
                tat[i] = proc[i].bt + wt[i];
}

// Function to calculate average time
void findavgTime(Process proc[], int n)
{
        int wt[n], tat[n], total_wt = 0,
                                    total_tat = 0;

        // Function to find waiting time of all
        // processes
        findWaitingTime(proc, n, wt);

        // Function to find turn around time for
        // all processes
        findTurnAroundTime(proc, n, wt, tat);

        // Display processes along with all
        // details
        cout << " P\t\t"
                << "BT\t\t"
                << "WT\t\t"
                << "TAT\t\t\n";

        // Calculate total waiting time and
        // total turnaround time
        for (int i = 0; i < n; i++) {
                total_wt = total_wt + wt[i];
                total_tat = total_tat + tat[i];
                cout << " " << proc[i].pid << "\t\t"
                        << proc[i].bt << "\t\t " << wt[i]
                        << "\t\t " << tat[i] << endl;
        }

        cout << "\nAverage waiting time = "
                << (float)total_wt / (float)n;
        cout << "\nAverage turn around time = "
                << (float)total_tat / (float)n;
}

// Driver code
```

```
int main()
{
        Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },
                                        { 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
        int n = sizeof(proc) / sizeof(proc[0]);

        findavgTime(proc, n);
        return 0;
}
```

OUTPUT:



```
 P         BT         WT         TAT
 1         6          7          13
 2         2          0          2
 3         8          14           22
 4         3          0          3
 5         4          2          6

Average waiting time = 4.6
Average turn around time = 9.2
```

Advantages:

- Short processes are handled very quickly.
- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.
- When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

EXPERIMENT - 8

AIM : Write a program to perform Longest Remaining Time First, LRTF (pre-emptive scheduling).

Theory:

Introduction:

The Longest Remaining time First**(LRTF)** scheduling is the preemptive version of Longest Job First**(LJF)** scheduling. This scheduling algorithm is used by the operating system in order to schedule incoming processes so that they can be executed in a systematic way.
With this algorithm, the process having the maximum remaining time is processed first. In this, we will check for the maximum remaining time after an interval of time(say 1 unit) that is there another process having more Burst Time arrived up to that time.

Given below are some of the characteristics of LRTF:

- It is a CPU scheduling algorithm that is used to determine the process to be executed first among all incoming processes in a systematic way.
- This algorithm follows the preemptive approach because in this CPU is allocated to any process only for a fixed slice of time.
- In this algorithm, processes are selected on the basis of the highest burst time(the one with the highest burst time is processed first) and this process runs till the fixed slice of time. After that, the selection process takes place again.
- Due to the high value of the average waiting time, this algorithm is not optimal.

ALGORITHM:

- **Step-1:** Create a structure of process containing all necessary fields like AT (Arrival Time), BT(Burst Time), CT(Completion Time), TAT(Turn Around Time), WT(Waiting Time).
- **Step-2:** Sort according to the AT;
- **Step-3:** Find the process having Largest Burst Time and execute for each single unit. Increase the total time by 1 and reduce the Burst Time of that process with 1.

- **Step-4:** When any process have 0 BT left, then update the CT(Completion Time of that process CT will be Total Time at that time).
- **Step-2:** After calculating the CT for each process, find TAT and WT.

(TAT = CT - AT)
(WT = TAT - BT)

CODE:

```
#include <bits/stdc++.h>

using namespace std;

// creating a structure of a process
struct process {
	int processno;
	int AT;
	int BT;

	// for backup purpose to print in last
	int BTbackup;
	int WT;
	int TAT;
	int CT;
};

// creating a structure of 4 processes
struct process p[4];

// variable to find the total time
int totaltime = 0;
int prefinaltotal = 0;

// comparator function for sort()
bool compare(process p1, process p2)
{
	// compare the Arrival time of two processes
	return p1.AT < p2.AT;
}

// finding the largest Arrival Time among all the available
// process at that time
int findlargest(int at)
{
	int max = 0, i;
```

```
        for (i = 0; i < 4; i++) {
                if (p[i].AT <= at) {
                        if (p[i].BT > p[max].BT)
                                max = i;
                }
        }

        // returning the index of the process having the largest BT
        return max;
}

// function to find the completion time of each process
int findCT()
{

        int index;
        int flag = 0;
        int i = p[0].AT;
        while (1) {
                if (i <= 4) {
                        index = findlargest(i);
                }

                else
                        index = findlargest(4);
                cout << "Process executing at time " << totaltime
                        << " is: P" << index + 1 << "\t";

                p[index].BT -= 1;
                totaltime += 1;
                i++;

                if (p[index].BT == 0) {
                        p[index].CT = totaltime;
                        cout << " Process P" << p[index].processno
                                << " is completed at " << totaltime;
                }
                cout << endl;

                // loop termination condition
                if (totaltime == prefinaltotal)
                        break;
        }
}

int main()
```

```
{
        int i;

        // initializing the process number
        for (i = 0; i < 4; i++) {
                p[i].processno = i + 1;
        }

        // cout<<"arrival time of 4 processes : ";
        for (i = 0; i < 4; i++) // taking AT
        {
                p[i].AT = i + 1;
        }

        // cout<<" Burst time of 4 processes : ";
        for (i = 0; i < 4; i++) {

                // assigning {2, 4, 6, 8} as Burst Time to the processes
                // backup for displaying the output in last
                // calculating total required time for terminating
                // the function().
                p[i].BT = 2 * (i + 1);
                p[i].BTbackup = p[i].BT;
                prefinaltotal += p[i].BT;
        }

        // displaying the process before executing
        cout << "PNo\tAT\tBT\n";

        for (i = 0; i < 4; i++) {
                cout << p[i].processno << "\t";
                cout << p[i].AT << "\t";
                cout << p[i].BT << "\t";
                cout << endl;
        }
        cout << endl;

        // sorting process according to Arrival Time
        sort(p, p + 4, compare);

        // calculating initial time when execution starts
        totaltime += p[0].AT;

        // calculating to terminate loop
        prefinaltotal += p[0].AT;
        findCT();
        int totalWT = 0;
        int totalTAT = 0;
        for (i = 0; i < 4; i++) {
                // since, TAT = CT - AT
                p[i].TAT = p[i].CT - p[i].AT;
                p[i].WT = p[i].TAT - p[i].BTbackup;

                // finding total waiting time
                totalWT += p[i].WT;

                // finding total turn around time
                totalTAT += p[i].TAT;
        }

        cout << "After execution of all processes ... \n";

        // after all process executes
        cout << "PNo\tAT\tBT\tCT\tTAT\tWT\n";

        for (i = 0; i < 4; i++) {
                cout << p[i].processno << "\t";
                cout << p[i].AT << "\t";
                cout << p[i].BTbackup << "\t";
                cout << p[i].CT << "\t";
                cout << p[i].TAT << "\t";
                cout << p[i].WT << "\t";
                cout << endl;
        }

        cout << endl;
        cout << "Total TAT = " << totalTAT << endl;
        cout << "Average TAT = " << totalTAT / 4.0 << endl;
        cout << "Total WT = " << totalWT << endl;
        cout << "Average WT = " << totalWT / 4.0 << endl;
        return 0;
}
```

OUTPUT :

```
PNo   AT   BT
1   1   2
2   2   4
3   3   6
```

```
4    4    8

Process executing at time 1 is: P1
Process executing at time 2 is: P2
Process executing at time 3 is: P3
Process executing at time 4 is: P4
Process executing at time 5 is: P4
Process executing at time 6 is: P4
Process executing at time 7 is: P3
Process executing at time 8 is: P4
Process executing at time 9 is: P3
Process executing at time 10 is: P4
Process executing at time 11 is: P2
Process executing at time 12 is: P3
Process executing at time 13 is: P4
Process executing at time 14 is: P2
Process executing at time 15 is: P3
Process executing at time 16 is: P4
Process executing at time 17 is: P1    Process P1 is completed at 18
Process executing at time 18 is: P2    Process P2 is completed at 19
Process executing at time 19 is: P3    Process P3 is completed at 20
Process executing at time 20 is: P4    Process P4 is completed at 21
After execution of all processes ...
PNo    AT    BT    CT    TAT    WT
1    1    2    18    17    15
2    2    4    19    17    13
3    3    6    20    17    11
4    4    8    21    17    9

Total TAT = 68
Average TAT = 17
Total WT = 48
Average WT = 12
```

Advantages of LRTF

- This algorithm is easy to implement and simple
- This algorithm is starvation-free because all processes get a fair share of CPU.
- All the processes get completed by the time the longest job reaches its completion.

EXPERIMENT – 9

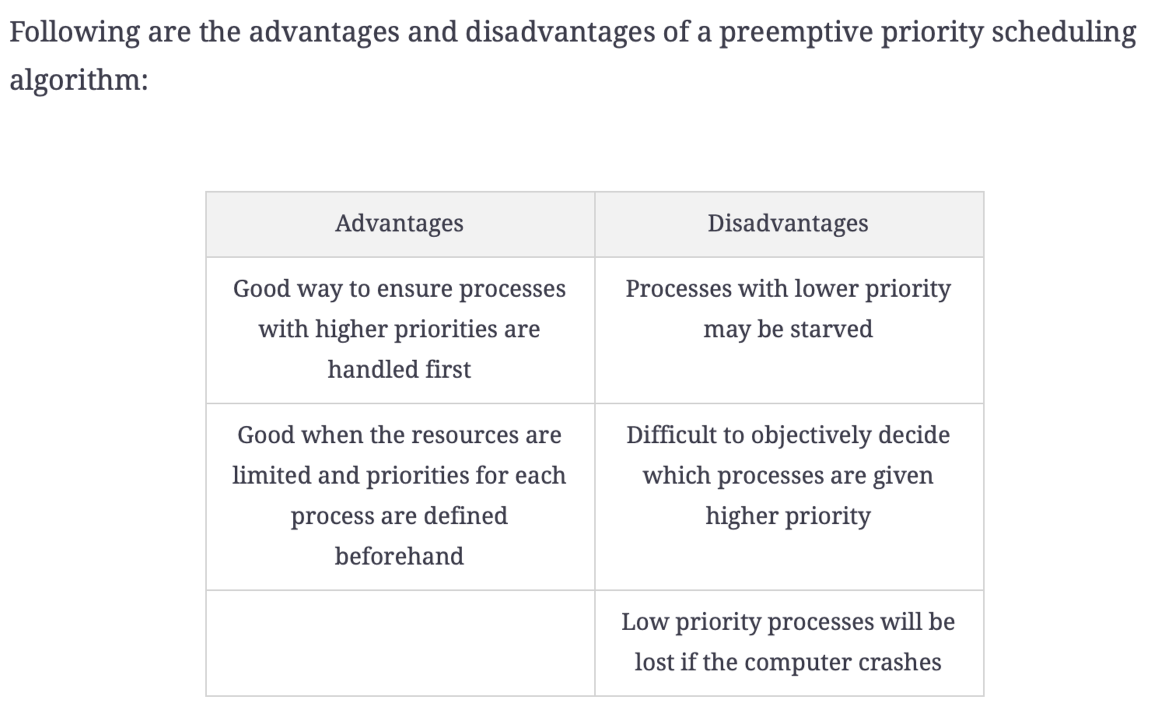
AIM : Write a program to perform Priority Scheduling (Pre-emptive scheduling).

Theory:

Introduction:

In **preemptive priority scheduling algorithm**, every time a process with higher priority arrives in the waiting queue, the CPU cycle is shifted to the process with the highest priority. This is preemptive because a process that’s already being executed can be stopped to execute a process with higher priority.

ADVANTAGES AND DISADVANTAGES



Following are the advantages and disadvantages of a preemptive priority scheduling algorithm:

Advantages	Disadvantages
Good way to ensure processes with higher priorities are handled first	Processes with lower priority may be starved
Good when the resources are limited and priorities for each process are defined beforehand	Difficult to objectively decide which processes are given higher priority
	Low priority processes will be lost if the computer crashes

CODE:

```
// CPP program to implement preemptive priority scheduling
#include <bits/stdc++.h>
using namespace std;

struct Process {
        int processID;
        int burstTime;
        int tempburstTime;
        int responsetime;
        int arrivalTime;
        int priority;
        int outtime;
        int intime;
};

// It is used to include all the valid and eligible
// processes in the heap for execution. heapsize defines
// the number of processes in execution depending on
// the current time currentTime keeps a record of
// the current CPU time.
void insert(Process Heap[], Process value, int* heapsize,
                        int* currentTime)
{
        int start = *heapsize, i;
        Heap[*heapsize] = value;
        if (Heap[*heapsize].intime == -1)
                Heap[*heapsize].intime = *currentTime;
        ++(*heapsize);

        // Ordering the Heap
        while (start != 0
                && Heap[(start - 1) / 2].priority
                        > Heap[start].priority) {
                Process temp = Heap[(start - 1) / 2];
                Heap[(start - 1) / 2] = Heap[start];
                Heap[start] = temp;
                start = (start - 1) / 2;
        }
}

// It is used to reorder the heap according to
// priority if the processes after insertion
// of new process.
void order(Process Heap[], int* heapsize, int start)
{
        int smallest = start;
        int left = 2 * start + 1;
        int right = 2 * start + 2;
        if (left < *heapsize
                && Heap[left].priority
                    < Heap[smallest].priority)
                smallest = left;
        if (right < *heapsize
                && Heap[right].priority
```

```
                        < Heap[smallest].priority)
                smallest = right;

        // Ordering the Heap
        if (smallest != start) {
                Process temp = Heap[smallest];
                Heap[smallest] = Heap[start];
                Heap[start] = temp;
                order(Heap, heapsize, smallest);
        }
}

// This function is used to find the process with
// highest priority from the heap. It also reorders
// the heap after extracting the highest priority process.
Process extractminimum(Process Heap[], int* heapsize,
                                        int* currentTime)
{
        Process min = Heap[0];
        if (min.responsetime == -1)
                min.responsetime
                        = *currentTime - min.arrivalTime;
        --(*heapsize);
        if (*heapsize >= 1) {
                Heap[0] = Heap[*heapsize];
                order(Heap, heapsize, 0);
        }
        return min;
}

// Compares two intervals
// according to starting times.
bool compare(Process p1, Process p2)
{
        return (p1.arrivalTime < p2.arrivalTime);
}

// This function is responsible for executing
// the highest priority extracted from Heap[].
void scheduling(Process Heap[], Process array[], int n,
                                int* heapsize, int* currentTime)
{
        if (heapsize == 0)
                return;

        Process min = extractminimum(
                Heap, heapsize, currentTime);
        min.outtime = *currentTime + 1;
        --min.burstTime;
        printf("process id = %d current time = %d\n",
                min.processID, *currentTime);

        // If the process is not yet finished
        // insert it back into the Heap*/
        if (min.burstTime > 0) {
                insert(Heap, min, heapsize, currentTime);
                return;
        }

        for (int i = 0; i < n; i++)
                if (array[i].processID == min.processID) {
                        array[i] = min;
                        break;
                }
}

// This function is responsible for
// managing the entire execution of the
// processes as they arrive in the CPU
// according to their arrival time.
void priority(Process array[], int n)
{
        sort(array, array + n, compare);

        int totalwaitingtime = 0, totalbursttime = 0,
                totalturnaroundtime = 0, i, insertedprocess = 0,
                heapsize = 0, currentTime = array[0].arrivalTime,
                totalresponsetime = 0;

        Process Heap[4 * n];

        // Calculating the total burst time
        // of the processes
        for (int i = 0; i < n; i++) {
                totalbursttime += array[i].burstTime;
                array[i].tempburstTime = array[i].burstTime;
        }

        // Inserting the processes in Heap
        // according to arrival time
        do {
                if (insertedprocess != n) {
```

```
            for (i = 0; i < n; i++) {
                if (array[i].arrivalTime == currentTime) {
                    ++insertedprocess;
                    array[i].intime = -1;
                    array[i].responsetime = -1;
                    insert(Heap, array[i],
                        &heapsize, ¤tTime);
                }
            }
        }
        scheduling(Heap, array, n,
                &heapsize, ¤tTime);
        ++currentTime;
        if (heapsize == 0
            && insertedprocess == n)
            break;
    } while (1);

    for (int i = 0; i < n; i++) {
        totalresponsetime
            += array[i].responsetime;
        totalwaitingtime
            += (array[i].outtime
                - array[i].intime
                - array[i].tempburstTime);
        totalbursttime += array[i].burstTime;
    }
    printf("Average waiting time = %f\n",
        ((float)totalwaitingtime / (float)n));
    printf("Average response time =%f\n",
        ((float)totalresponsetime / (float)n));
    printf("Average turn around time = %f\n",
        ((float)(totalwaitingtime
                    + totalbursttime)
            / (float)n));
}

// Driver code
int main()
{
    int n, i;
    Process a[5];
    a[0].processID = 1;
    a[0].arrivalTime = 4;
    a[0].priority = 2;
    a[0].burstTime = 6;
```

```
        a[1].processID = 4;
        a[1].arrivalTime = 5;
        a[1].priority = 1;
        a[1].burstTime = 3;
        a[2].processID = 2;
        a[2].arrivalTime = 5;
        a[2].priority = 3;
        a[2].burstTime = 1;
        a[3].processID = 3;
        a[3].arrivalTime = 1;
        a[3].priority = 4;
        a[3].burstTime = 2;
        a[4].processID = 5;
        a[4].arrivalTime = 3;
        a[4].priority = 5;
        a[4].burstTime = 4;
        priority(a, 5);
        return 0;
}
```

OUTPUT

```
process id = 3 current time = 1
process id = 3 current time = 2
process id = 5 current time = 3
process id = 1 current time = 4
process id = 4 current time = 5
process id = 4 current time = 6
process id = 4 current time = 7
process id = 1 current time = 8
process id = 1 current time = 9
process id = 1 current time = 10
process id = 1 current time = 11
process id = 1 current time = 12
process id = 2 current time = 13
process id = 5 current time = 14
process id = 5 current time = 15
process id = 5 current time = 16
Average waiting time = 4.200000
Average response time =1.600000
Average turn around time = 7.400000
```

EXPERIMENT – 10

AIM : Write a program to perform Round Robin scheduling. (Pre-emptive scheduling).

Theory:

Introduction:

The Round Robin scheduling algorithm is a preemptive scheduling algorithm. It uses a concept of time slice or time quantum. The process at the beginning of the ready queue gets the chance to be executed first but only for the span of one-time quantum.
As new and more processes get added to the ready queue, the ongoing process gets preempted and gets added to the end of the ready queue. The next process gets the chance, again for the span of one-time quantum. This algorithm is designed for time-sharing systems.

In Round Robin Scheduling Algorithm, when a process is already in the CPU being executed it has a limited time quantum which it gets to execute within but if it cannot complete itself then it gets preempted. So as the time quantum progresses the chances of the process being preempted increases.
Its priority decreases. Whereas, the process which is waiting in the ready queue is getting its chances increased in terms of getting the CPU next. So, its priority increases. So, we can say that Round Robin is a special kind of Preemptive Priority Scheduling Algorithm where a process in the ready queue gets its priority increased and a process in the CPU gets its priority decreased.

Advantages

- It doesn’t face the issues of starvation or convoy effect.
- All the jobs get a fair allocation of CPU.
- It deals with all process without any priority
- 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.

- This scheduling method does not depend upon burst time. That’s why it is easily implementable on the system.
- Once a process is executed for a specific set of the period, the process is preempted, and another process executes for that given time period.
- Allows OS to use the Context switching method to save states of preempted processes.
- It gives the best performance in terms of average response time.

Disadvantages

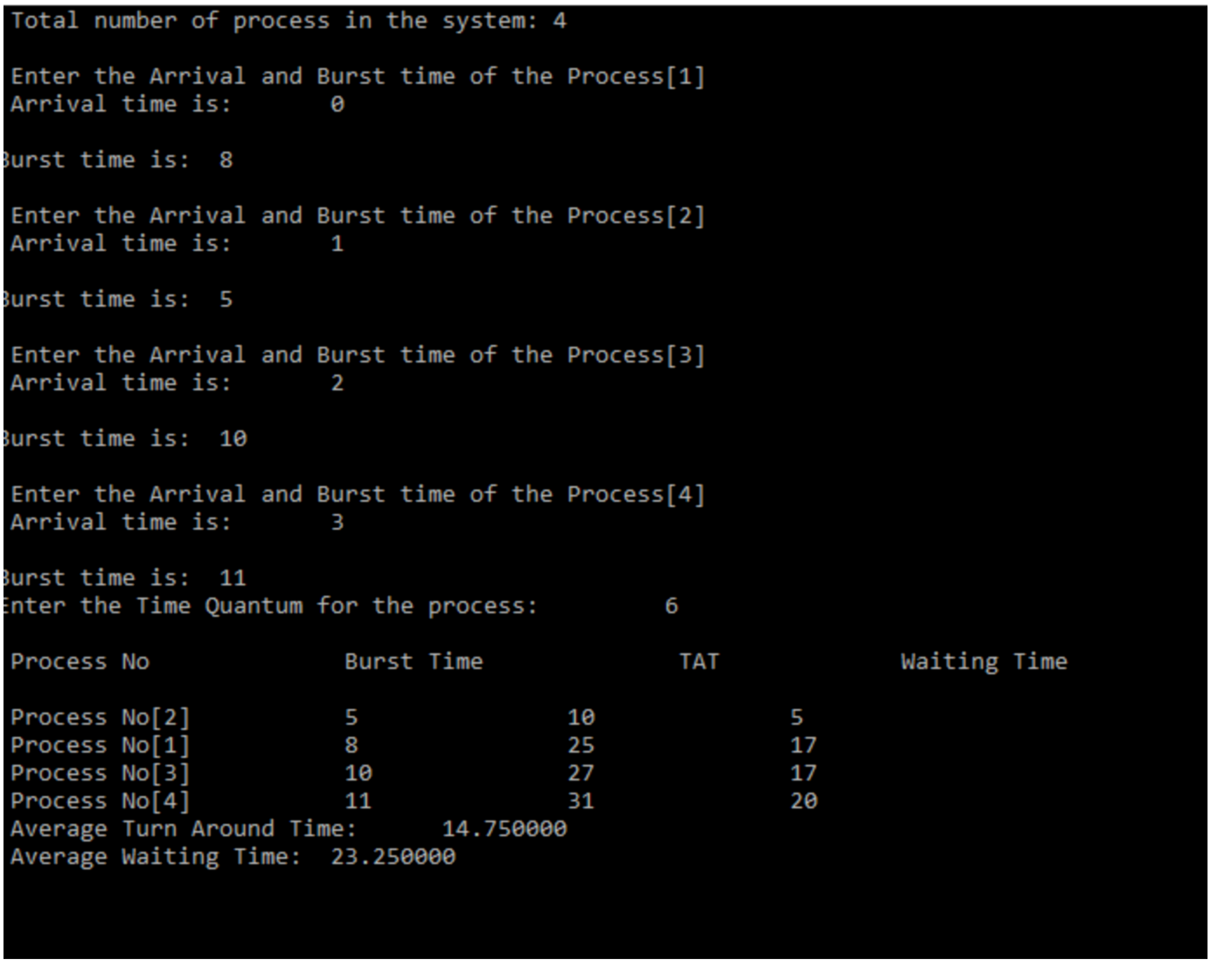
- If slicing time of OS is low, the processor output will be reduced.
- This method spends more time on context switching
- Its performance heavily depends on time quantum.
- Priorities cannot be set for the processes.
- Round-robin scheduling doesn’t give special priority to more important tasks.
- Decreases comprehension
- Lower time quantum results in higher the context switching overhead in the system.
- Finding a correct time quantum is a quite difficult task in this system.

CODE:

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

void main()
{
   // initlialize the variable name
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;
   printf(" Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");  // Accept arrival time
scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time
scanf("%d", &bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time qunat
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
   sum = sum + temp[i];
   temp[i] = 0;
   count=1;
   }
   else if(temp[i] > 0)
   {
      temp[i] = temp[i] - quant;
      sum = sum + quant;
   }
   if(temp[i]==0 && count==1)
   {
      y--; //decrement the process no.
      printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-
at[i]-bt[i]);
      wt = wt+sum-at[i]-bt[i];
      tat = tat+sum-at[i];
      count =0;
   }
   if(i==NOP-1)
   {
      i=0;
   }
   else if(at[i+1]<=sum)
   {
      i++;
   }
   else
   {
      i=0;
   }
}
// represents the average waiting time and Turn Around time
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
```

```
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}
```

OUTPUT:



```
Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]
Arrival time is:       0

Burst time is:  8

Enter the Arrival and Burst time of the Process[2]
Arrival time is:       1

Burst time is:  5

Enter the Arrival and Burst time of the Process[3]
Arrival time is:       2

Burst time is:  10

Enter the Arrival and Burst time of the Process[4]
Arrival time is:       3

Burst time is:  11
Enter the Time Quantum for the process:        6

Process No              Burst Time              TAT             Waiting Time

Process No[2]           5               10              5
Process No[1]           8               25              17
Process No[3]           10              27              17
Process No[4]           11              31              20
Average Turn Around Time:       14.750000
Average Waiting Time:   23.250000
```

EXPERIMENT - 11

AIM : To implement **Disk Management : FCFS**

Theory

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if First Come First Serve (FCFS) disk scheduling algorithm is used.

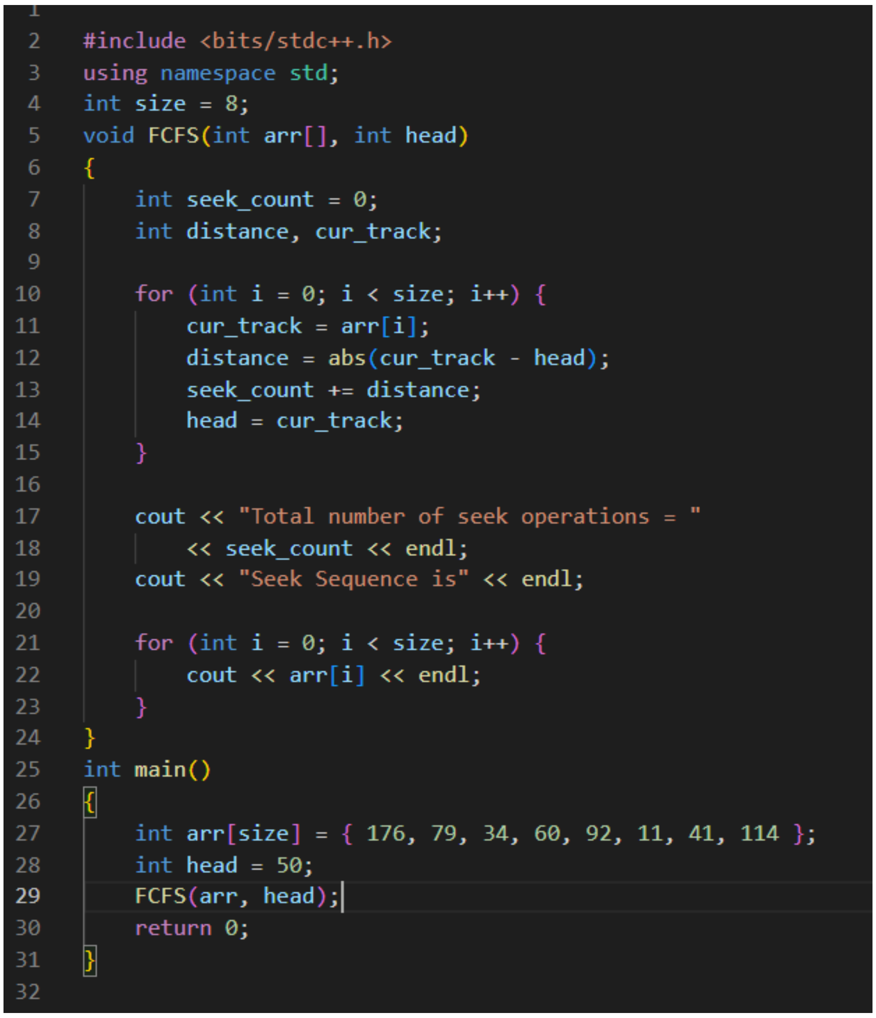
First Come First Serve (FCFS)

FCFS is the simplest disk scheduling algorithm. As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

Algorithm:

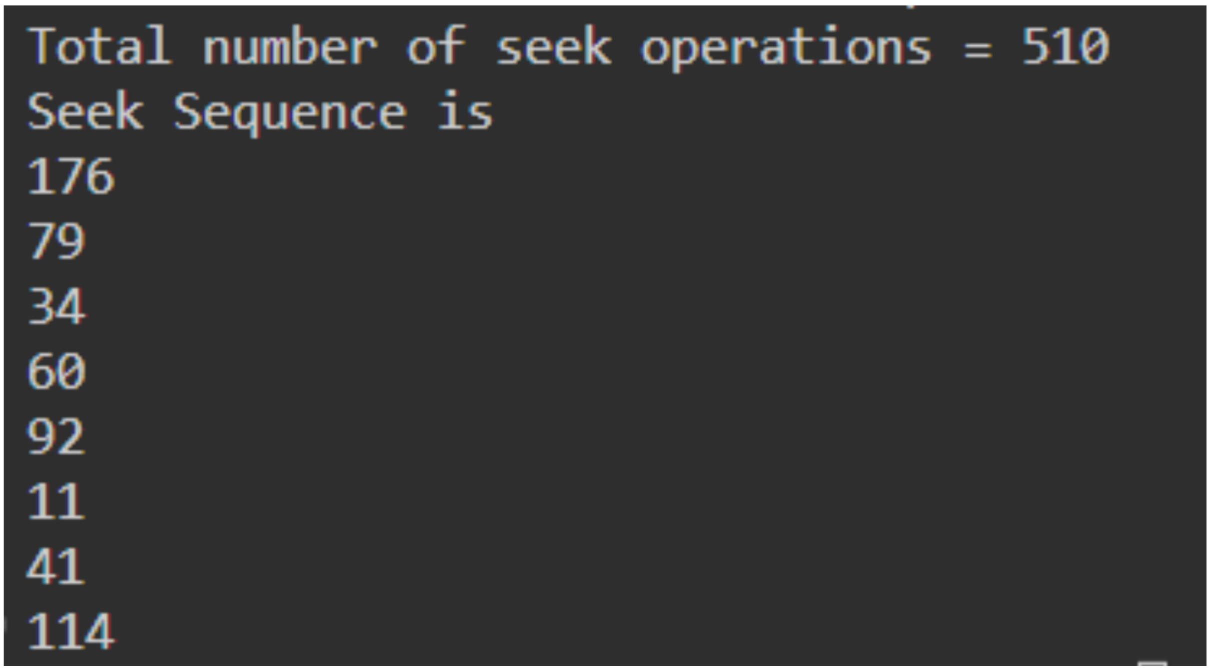
1. LetRequestarrayrepresentsanarraystoringindexesoftracksthathave been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Letusonebyonetakethetracksindefaultorderandcalculatethe absolute distance of the track from the head.
3. Incrementthetotalseekcountwiththisdistance.
4. Currentlyservicedtrackpositionnowbecomesthenewheadposition.
5. Gotostep2untilalltracksinrequestarrayhavenotbeenserviced.

CODE :



```

#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;
}

```

OUTPUT:



```
Total number of seek operations = 510
Seek Sequence is
176
79
34
60
92
11
41
114
```

NUMERICAL

Input:
Request sequence = {176, 79, 34, 60, 92, 11, 41, 114} Initial head position = 50

Output:
Total number of seek operations = 510
Seek Sequence is
176
79
34
60
92
11
41
114
Therefore, the total seek count is calculated as:

= (176-50)+(176-79)+(79-34)+(60-34)+(92-60)+(92-11)+(41-11)+(114-41) = 510

Applications:

1. FCFS is mostly used in systems that do not have many requests in the queue, as it provides a simple and efficient way to handle requests.

2. It can be used in embedded systems or devices with low processing power and limited resources, as it does not require complex calculations.

Advantages:

1. It is easy to understand and implement, as it requires minimal logic and data structures to implement.
2. It is fair, as the requests are processed in the order they arrive in the queue, ensuring that all requests are handled equally.
3. It is suitable for systems with low to moderate I/O request loads, as it provides a simple way to handle a small number of requests.

Disadvantages:

1. ItmaynotbeefficientforsystemswithahighnumberofI/Orequests,as it can result in long waiting times for some requests.
2. It can lead to poor disk utilization, as it may result in unnecessary head movement, leading to increased access time and reduced throughput.
3. Itmayresultinpoorperformancefortime-criticalsystems,asitdoesnot

   prioritize the most urgent or important requests.

In summary, FCFS is a simple and fair disk scheduling algorithm that works well for systems with low to moderate I/O request loads. However, it may not be the best choice for systems with a high number of requests or time-critical applications.

<u>EXPERIMENT - 12</u>

Aim : Disk management : **Shortest Seek Time First**

Theory

It appears reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the Shortest Seek Time First (SSTF) Algorithm.

The SSTF algorithm selects the request having the minimum distance from the current head position. Since distance increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

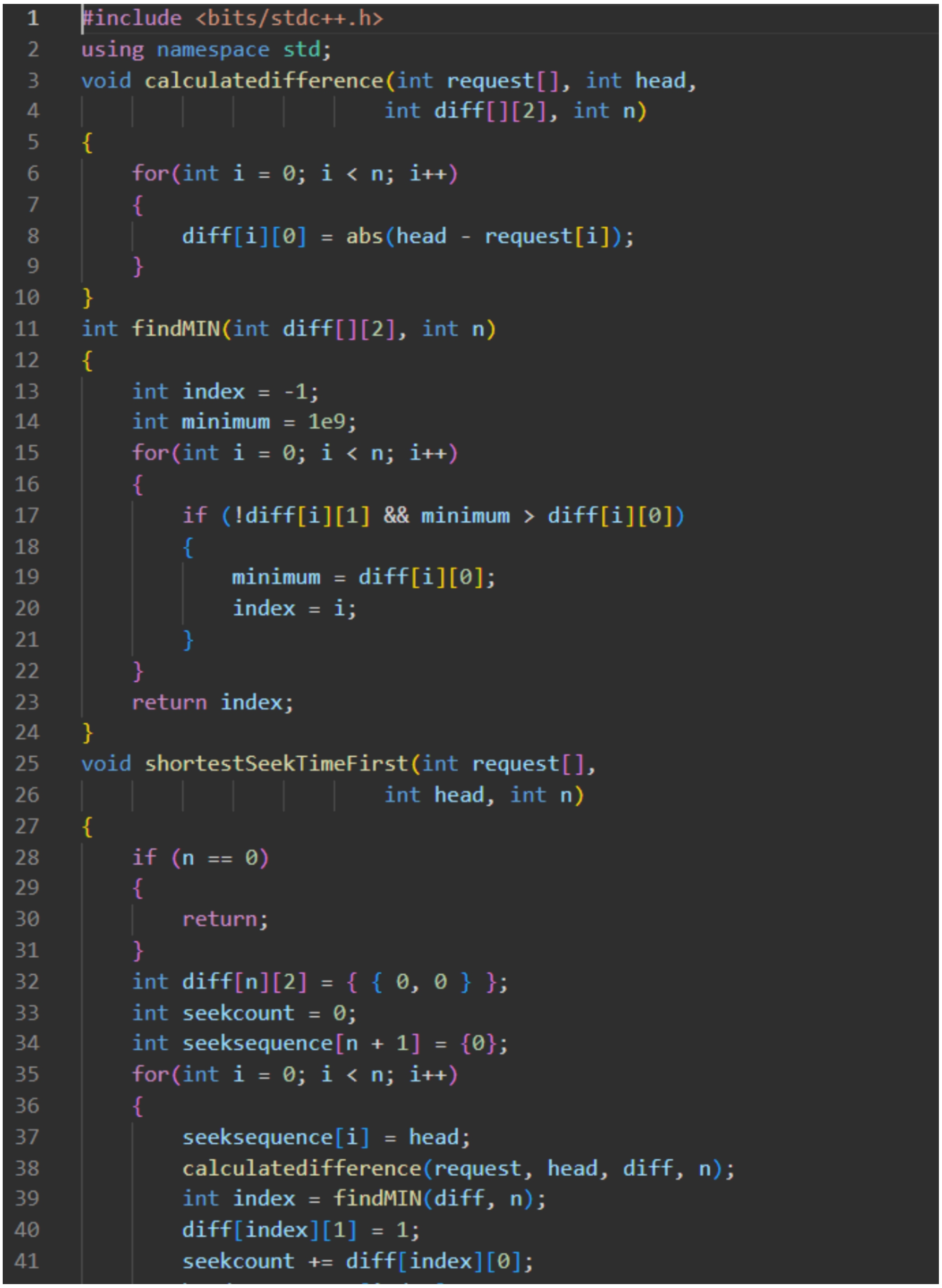
Algorithm

To understand the SSTF disk scheduling algorithm, let us assume a disc queue with requests for I/O. 'head' is the position of the disc head. We will now apply the SSTF algorithm-

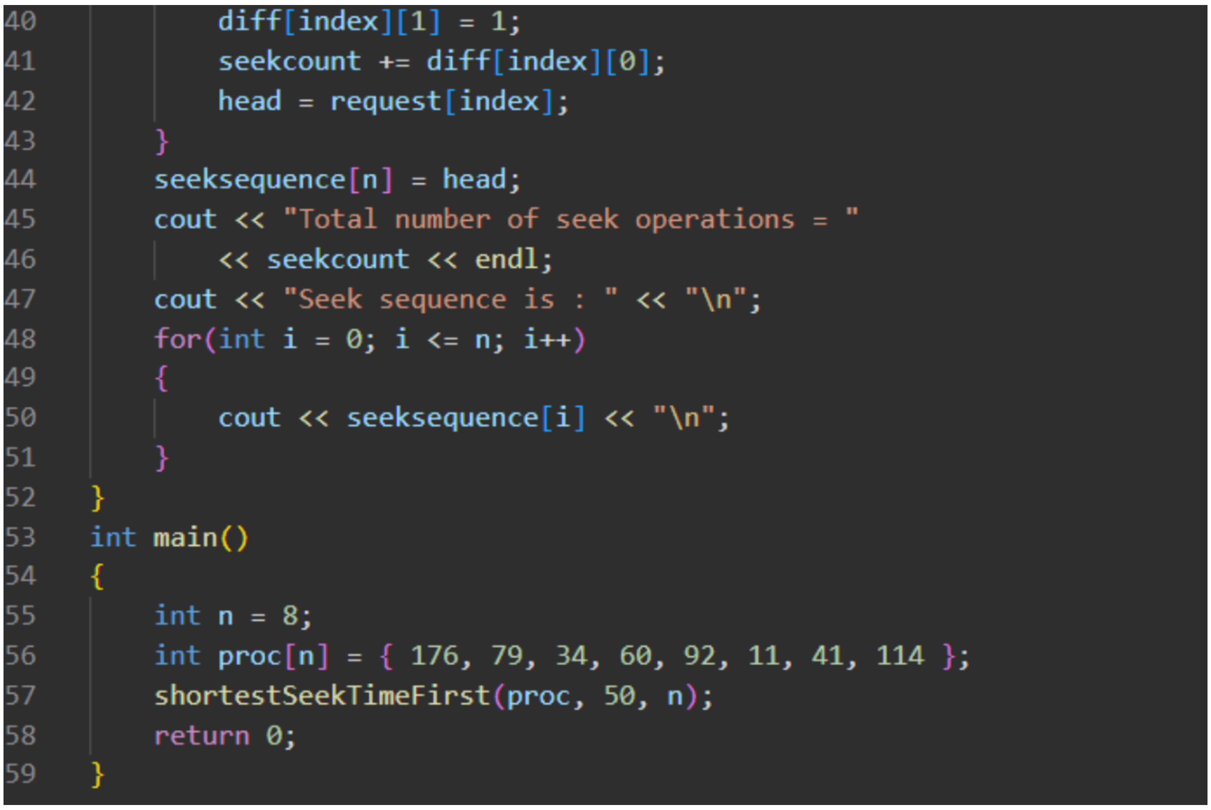
- Arrange all the I/O requests in ascending order.
- The head will find the nearest request (which has a minimum distance from the head) present in any direction (left or right) and will move to that request. Total head movement is calculated as
- Current request - previous request (if the current request is greater)
- Previous request - current request (if the previous request is greater)
- Then the head will move another nearest request which has not been

  serviced present in any direction.
- This process is repeated until all the requests are served and we get total head movement.

CODE :

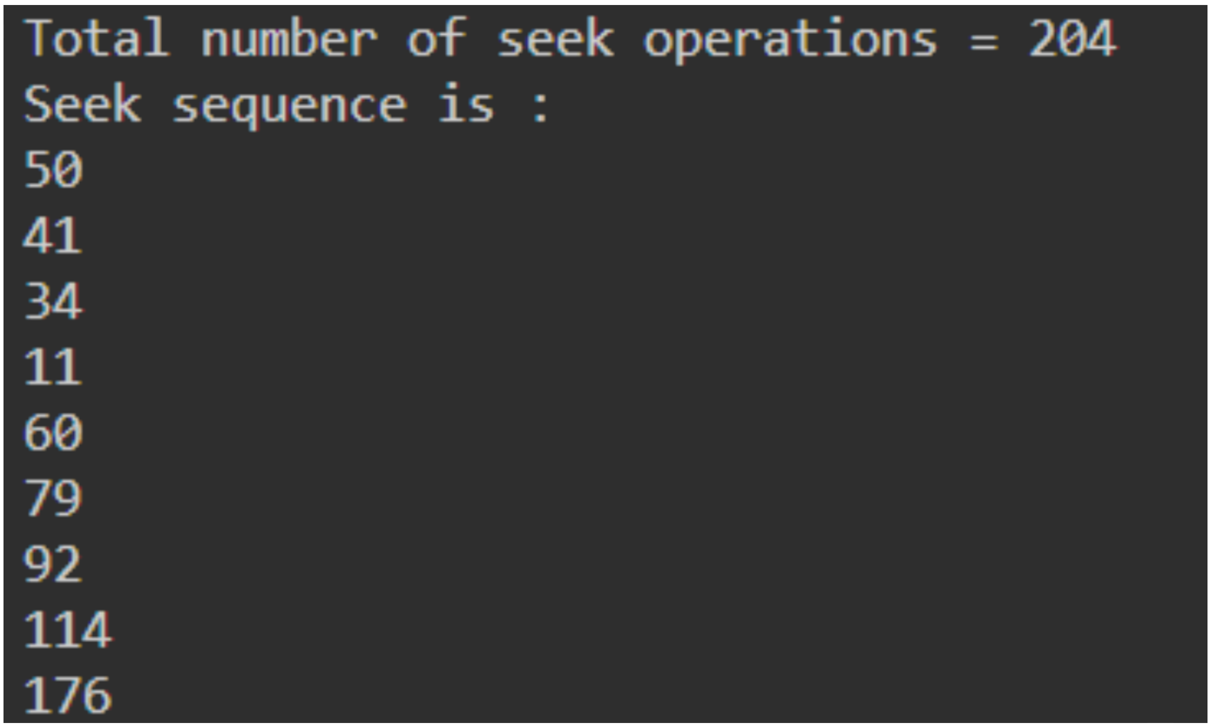


```cpp
#include <bits/stdc++.h>
using namespace std;
void calculatedifference(int request[], int head,
                                    int diff[][2], int n)
{
    for(int i = 0; i < n; i++)
    {
        diff[i][0] = abs(head - request[i]);
    }
}
int findMIN(int diff[][2], int n)
{
    int index = -1;
    int minimum = 1e9;
    for(int i = 0; i < n; i++)
    {
        if (!diff[i][1] && minimum > diff[i][0])
        {
            minimum = diff[i][0];
            index = i;
        }
    }
    return index;
}
void shortestSeekTimeFirst(int request[],
                                    int head, int n)
{
    if (n == 0)
    {
        return;
    }
    int diff[n][2] = { { 0, 0 } };
    int seekcount = 0;
    int seeksequence[n + 1] = {0};
    for(int i = 0; i < n; i++)
    {
        seeksequence[i] = head;
        calculatedifference(request, head, diff, n);
        int index = findMIN(diff, n);
        diff[index][1] = 1;
        seekcount += diff[index][0];
```



```cpp
        diff[index][1] = 1;
        seekcount += diff[index][0];
        head = request[index];
    }
    seeksequence[n] = head;
    cout << "Total number of seek operations = "
        << seekcount << endl;
    cout << "Seek sequence is : " << "\n";
    for(int i = 0; i <= n; i++)
    {
        cout << seeksequence[i] << "\n";
    }
}
int main()
{
    int n = 8;
    int proc[n] = { 176, 79, 34, 60, 92, 11, 41, 114 };
    shortestSeekTimeFirst(proc, 50, n);
    return 0;
}
```

OUTPUT



```
Total number of seek operations = 204
Seek sequence is :
50
41
34
11
60
79
92
114
176
```

Time Complexity: O(N^2)

Auxiliary Space: O(N)

Advantages of Shortest Seek Time First (SSTF) –

- Better performance than FCFS scheduling algorithm.
- It provides better throughput.
- This algorithm is used in Batch Processing system where throughput is more important.
- It has less average response and waiting time.

Disadvantages of Shortest Seek Time First (SSTF) –

- Starvation is possible for some requests as it favours easy to reach request and ignores the far away processes.
- There is lack of predictability because of high variance of response time.
- Switching direction slows things down.