DELHI TECHNOLOGICAL UNIVERSITY

OPERATING SYSTEMS PRACTICAL FILE

Submitted to:
Dr. Akshi Kumar
Assistant Professor

Submitted by:
Kunal Sinha
2K17/CO/A3/164

INDEX

S.No	Experiment	Done On	Checked on	Signature

PROGRAM 1

AIM:
WAP to write to a text file and then read it character wise while counting the number of alphabets, numeric characters, special characters, words, spaces and lines.

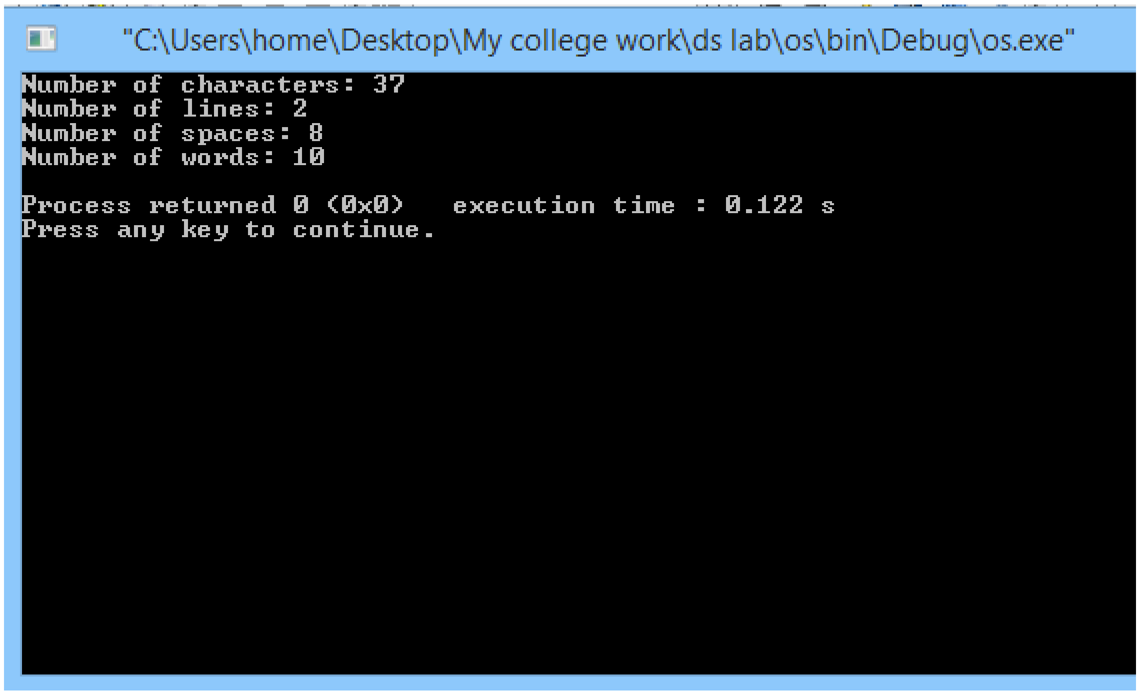
THEORY:
An fstream object is created and is attached to the file in out mode i.e. the write mode. The string to be written is taken as input from the user. This string is then written to the file. Then the file is opened in the in mode i.e. read mode and is read char by char till end of file is detected. And by looping the number of alphabets.the number of alphabets, numeric characters, special symbols, spaces, words are counted. The output is then displayed.

ALGORITHM:
```
Step 1: START
Step 2: fstream object created and attaches to the file in "out mode”.
Step 3: Input string taken from the user.
Step 4: String written to the file.
Step 5: File closed.
Step 6: File opened in ‘in' mode and is read character wise iteratively till eof is detected.
Step 7: Each character is tested for its ASCII value and then appropriate counters are
incremented.
Step 8: Final value of all the counters is displayed.
Step 9: END
```

CODE:
```
/*File content
Hello how are you
I am good, What about you?
*/
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int get_words(char s[])
{
int len = strlen(s), count = 0;
for(int i=0; i<len; i++)
    {
if (s[i] == ' ')
        {
count++;
while (s[++i] == ' ');
i--;
        }
    }
return count + 1;
}

int main()
{
ofstream fout;
fout.open("files.txt");
fout<<"Hello How are you\nI am good, What about you?";
fout.close();

ifstream fin;
fin.open("files.txt");
int num_char = 0, num_spaces = 0, num_lines = 0, num_words = 0;
char ch, s[100];
while (!fin.eof())
    {
fin.get(ch);
if (ch != 32)
num_char++;
    }
cout<< "Number of characters: " <<num_char<< "\n";

fin.clear();
fin.seekg(0,ios::beg);
while(!fin.eof())
    {
fin.getline(s, 100, '\n');
num_lines++;
    }
cout<< "Number of lines: " <<num_lines<< "\n";
fin.clear();
fin.seekg(0,ios::beg);
while (!fin.eof())
    {
fin.get(ch);
if (ch == 32)
num_spaces++;
    }
cout<< "Number of spaces: " <<num_spaces<< "\n";

fin.clear();
fin.seekg(0,ios::beg);
while (!fin.eof())
    {
fin.getline(s, 100, '\n');
```

```
num_words += get_words(s);
   }
cout<< "Number of words: " <<num_words<< "\n";
fin.close();
}
```

OUTPUT:



"C:\Users\home\Desktop\My college work\ds lab\os\bin\Debug\os.exe"

```
Number of characters: 37
Number of lines: 2
Number of spaces: 8
Number of words: 10

Process returned 0 (0x0)   execution time : 0.122 s
Press any key to continue.
```

LEARNING OUTCOMES

In this program we input the text from the user in the file and it gets stored in the file. Now we read from the file till end of file is reached.

1. Problem statement was executed.
2. Files are basically used to store data that may vanish if the program is run again.
3. ofstream and ifstream are objects of class fstream.
4. File is read in ifstream and is written in ofstream mode.

CONCLUSION:

We have successfully counted the required number of alphabets, numbers, characters, spaces in the file as input by the user using file manipulation functions

PROGRAM 2

AIM:
WAP to implement First Come First Serve (FCFS) algorithm for CPU scheduling.

THEORY:
First Come First Serve (FCFS) is a job scheduling algorithm used in CPU. In this the jobs are executed on the first come, first serve basis. It is a non-preemptive, pre-emptive scheduling algorithm. It is easy to understand and implement and its implementation is based on FIFO queue. It is poor in performance as average wait time is high.

ALGORITHM:

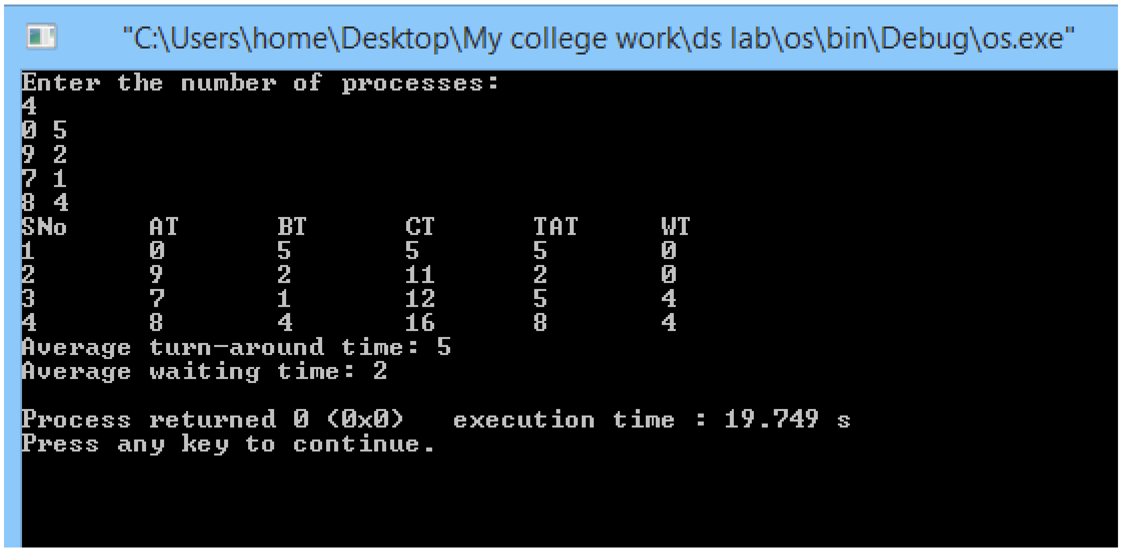
```
Step 1: START
Step 2: Input the number of processes (n).
Step 3: Input the Arrival time and Burst time of n processes.
Step 4: Set completion time of 0th process = arrival time of 0th + burst time of 0th.
Step 5: For i in range(1,n):
        Step 6: If (arrival time of ith< completion time of (i-1)th):
                        Set completion time of ith = completion time of (i-1)th+ burst time of ith
                Else
                        Set completion time of ith = arrival time of ith + burst time of ith.
Step 7:For i in range(0,n):
        Step 8: Set turnaround time of ith = completion time of ith – arrival time of ith
        Step 9: Set waiting time of ith = turnaround time of ith – burst time of ith
Step 10: Calculate sum of turnaround time for all processes and evaluate average.
Step 11: Calculate sum of waiting time for all processes and evaluate average.
Step 12: Print all times for all processes and Average of turnaround and waiting time.
Step 13: END
```

CODE:

```
#include <iostream>
using namespace std;

struct Process
{
int a_t, b_t, c_t, ta_t, w_t;
};
int n;
Process p[100];
void print_table()
{
int i;
cout<<"SNo\tAT\tBT\tCT\tTAT\tWT\n";
for (i = 0; i < n; i++)
   {
```

```
cout<<i + 1<<"\t"<<p[i].a_t<<"\t"<< p[i].b_t<<"\t"<<p[i].c_t<<"\t"<<p[i].ta_t<<"\t"<<
p[i].w_t<<"\n";
    }
}
int main()
{
int i, sum_tat = 0, sum_wt = 0;
float avg_tat, avg_wt;

cout<< "Enter the number of processes: \n";
cin>> n;
for (i = 0; i < n; i++)
    {
cin>> p[i].a_t;
cin>> p[i].b_t;
    }

p[0].c_t = p[0].a_t + p[0].b_t;
for (i = 1; i < n; i++)
    {
if (p[i].a_t< p[i - 1].c_t)
p[i].c_t = p[i - 1].c_t + p[i].b_t;
else
p[i].c_t = p[i].a_t + p[i].b_t;
    }

for (i = 0; i < n; i++)
    {
p[i].ta_t = p[i].c_t - p[i].a_t;
p[i].w_t = p[i].ta_t - p[i].b_t;
    }
for (i = 0; i < n; i++)
    {
sum_tat += p[i].ta_t;
sum_wt += p[i].w_t;
    }
avg_tat = (float)sum_tat / n;
avg_wt = (float)sum_wt / n;
print_table();
cout<< "Average turn-around time: " <<avg_tat<< "\n";
cout<< "Average waiting time: " <<avg_wt<< "\n";
return 0;
}
```

OUTPUT:



```
"C:\Users\home\Desktop\My college work\ds lab\os\bin\Debug\os.exe"
Enter the number of processes:
4
0 5
9 2
7 1
8 4
SNo     AT      BT      CT      TAT     WT
1       0       5       5       5       0
2       9       2       11      2       0
3       7       1       12      5       4
4       8       4       16      8       4
Average turn-around time: 5
Average waiting time: 2

Process returned 0 (0x0)   execution time : 19.749 s
Press any key to continue.
```

LEARNING OUTCOMES:

1. It is Non-Pre-emptive.
2. The Average Waiting Time is not optimal.
3. It can't utilize resources in parallel: Results in Convoy Effect.

CONCLUSION:
We have successfully implemented the FCFS principle in job scheduling and calculated the average waiting time and average turnaround time.

PROGRAM 3

AIM:
WAP to implement Shortest Job First (SJF) algorithm for CPU scheduling.

THEORY:
Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm. Shortest Job first has the advantage of having minimum average waiting time among all scheduling algorithms. It is a Greedy Algorithm.

ALGORITHM:

```
Step 1: START
Step 2: Input the number of processes (n).
Step 3: Input the Arrival time and Burst time of n processes.
Step 4: Initialize a priority queue and Set time to arrival time of first process.
Step 5: For all processes, if arrival time of i^th process is <= time, push process in a priority queue.
Step 6: While queue is not empty:
        Step 7: Remove process with minimum burst time.
        Step 8: Set completion time of popped process to time + burst time of the process.
        Step 9: Increment time by burst time of the process.
        Step 10: For all processes, if arrival time of ith process is <= time, push process in queue.
Step 11: For i in range (0, n):
        Step 12: Set turnaround time of i^th = completion time of i^th – arrival time of i^th
        Step 13: Set waiting time of i^th = turnaround time of i^th – burst time of i^th
Step 14: Calculate sum of turnaround time for all processes and evaluate average.
Step 15: Calculate sum of waiting time for all processes and evaluate average.
Step 16: Print all times for all processes and Average of turnaround and waiting time.
Step 17: END
```

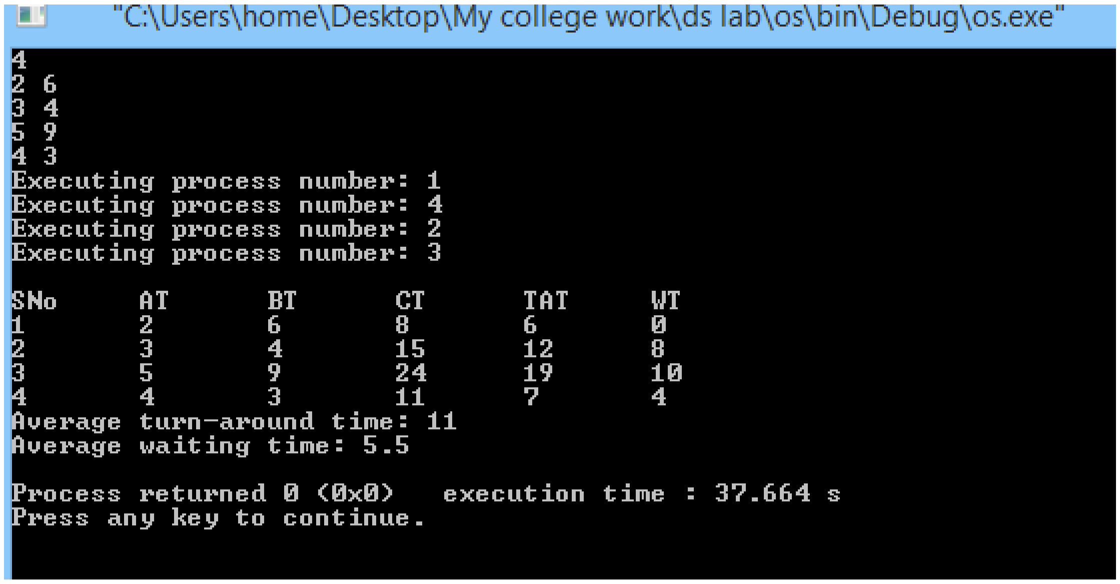
CODE:

```
#include <iostream>
#include <queue>
using namespace std;
// Class to encapsulate Process properties
class Process
{
public:
int a_t, b_t, c_t, ta_t, w_t, index;
};
// Function to overaload < operator
bool operator<(const Process &p1, const Process &p2)
{
return p2.b_t < p1.b_t;
}
int main()
```

```
{
int n, i, sum_tat = 0, sum_wt = 0;
float avg_tat, avg_wt;
Process p[100];
// Priority Q to get shortest burst time of processes added
priority_queue<Process> q;
cin >> n;
for (i = 0; i < n; i++)
{
cin >> p[i].a_t;
cin >> p[i].b_t;
p[i].index = i + 1;
}
int time = p[0].a_t, temp = 0;
for (i = 0; i < n; i++)
{
if (p[i].a_t <= time)
{
q.push(p[i]);
temp = i;
}
}
// Calculation of completion time for processes
while (!q.empty())
{
Process proc = q.top();
cout << "Executing process number: " << proc.index << "\n";
p[proc.index - 1].c_t = time + proc.b_t;
time = time + proc.b_t;
q.pop();
for (i = temp + 1; i < n; i++)
{
if (p[i].a_t <= time)
{
q.push(p[i]);
temp = i;
}}}
// Calculation of waiting and turn-around time
for (i = 0; i < n; i++)
{
p[i].ta_t = p[i].c_t - p[i].a_t;
p[i].w_t = p[i].ta_t - p[i].b_t;
}
// Calculation of average waiting and turn-around time
for (i = 0; i < n; i++)
{
```

```
sum_tat += p[i].ta_t;
sum_wt += p[i].w_t;
}
avg_tat = (float)sum_tat / n;
avg_wt = (float)sum_wt / n;
// Printing table
cout<<"\nSNo\tAT\tBT\tCT\tTAT\tWT\n";
for (i = 0; i < n; i++)
cout<<p[i].index<<"\t"<<p[i].a_t<<"\t"<<
p[i].b_t<<"\t"<<p[i].c_t<<"\t"<<p[i].ta_t<<"\t"<<p[i].w_t<<"\n";
cout << "Average turn-around time: " << avg_tat << "\n";
cout << "Average waiting time: " << avg_wt << "\n";
return 0;
}
```

OUTPUT:



```
"C:\Users\home\Desktop\My college work\ds lab\os\bin\Debug\os.exe"
4
2 6
3 4
5 9
4 3
Executing process number: 1
Executing process number: 4
Executing process number: 2
Executing process number: 3

SNo     AT      BT      CT      TAT     WT
1       2       6       8       6       0
2       3       4       15      12      8
3       5       9       24      19      10
4       4       3       11      7       4
Average turn-around time: 11
Average waiting time: 5.5

Process returned 0 (0x0)   execution time : 37.664 s
Press any key to continue.
```

LEARNING OUTCOMES:

1. It has minimum average waiting time.
2. It may cause starvation (in pre-emptive), if shorter processes keep coming.
3. It is Non-Pre-emptive.

CONCLUSION:

We have successfully implemented the SJF principle in job scheduling and calculated the average waiting time and average turn around time.

PROGRAM 4

AIM:

WAP to implement Shortest Remaining Time First (SRTF) algorithm for job scheduling.

THEORY:

This Algorithm is the preemptive version of SJF scheduling. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process. Once all the processes are available in the ready queue, No preemption will be done and the algorithm will work as SJF scheduling. The context of the process is saved in the Process Control Block when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the next execution of this process.

ALGORITHM:

```
Step 1: START
Step 2: Traverse until all process gets completely executed.
a) Find process with minimum remaining time at every single time lap.
b) Reduce its time by 1.
c) Check if its remaining time becomes 0
d) Increment the counter of process completion.
e) Completion time of current process = current_time +1;
e) Calculate waiting time for each completed process.
wt[i]= Completion time - arrival_time-burst_time
f) Increment time lap by one.
Step 3: Find turnaround time (waiting_time+burst_time).
Step 4: END
```

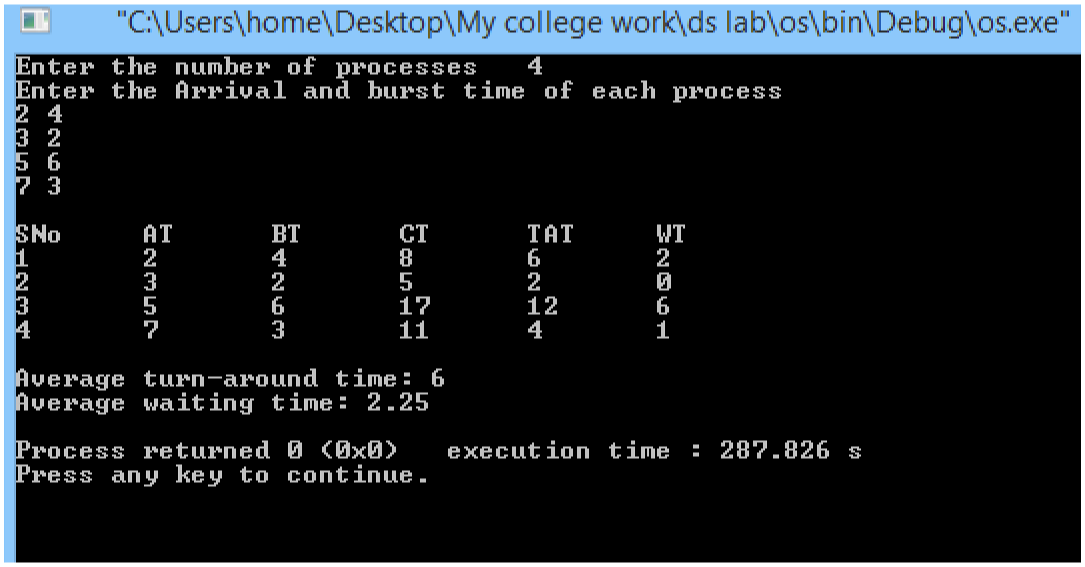
CODE:

```
#include <iostream>
#include <queue>
using namespace std;
class Process
{
public:
int index, arr_t, burst_t, comp_t, wait_t, ta_t, rem_t;
};
bool operator < (const Process &p1, const Process &p2)
{
return p2.rem_t < p1.rem_t;
}
Process p[100];
int n;
int main()
{
```

```
int i;
priority_queue <Process> q;
cout<<"Enter the number of processes   ";
cin >> n;
cout<<"Enter the Arrival and burst time of each process\n";
for(i=0; i<n; i++)
{
cin >> p[i].arr_t;
cin >> p[i].burst_t;
p[i].rem_t = p[i].burst_t;
p[i].index = i+1;
}
i = 0;
int pInQue=0;
int time = p[0].arr_t;
while(time == p[i].arr_t && i<n)
{
q.push(p[i]);
pInQue++;
i++;
}
Process shortestProcess;
while(pInQue!=n)
{
shortestProcess = q.top();
q.pop();
int index = shortestProcess.index-1;
int executionTime = p[pInQue].arr_t - time;
p[index].rem_t = p[index].rem_t - executionTime;
time = time + executionTime;
if(p[index].rem_t == 0)
p[index].comp_t = time;
else
q.push(p[index]);
int j= pInQue;
while(j<n && time >= p[j].arr_t)
{
q.push(p[j]);
pInQue++;
j++;
}
if(q.empty() && pInQue!=n)
{
time = p[j].arr_t;
while(time >= p[j].arr_t && j<n)
{
```

```
q.push(p[j]);
pInQue++;
j++;}
}}
while(!q.empty())
{
shortestProcess = q.top();
q.pop();
int index = shortestProcess.index-1;
p[index].comp_t = time + p[index].rem_t;
p[index].rem_t = 0;
time = p[index].comp_t;
}
// Calculation of waiting and turn-around time
int sum_tat = 0, sum_wt = 0;
float avg_tat, avg_wt;
for (i = 0; i < n; i++)
{
p[i].ta_t = p[i].comp_t - p[i].arr_t;
p[i].wait_t = p[i].ta_t - p[i].burst_t;
}
// Calculation of average waiting and turn-around time
for (i = 0; i < n; i++)
{
sum_tat += p[i].ta_t;
sum_wt += p[i].wait_t;
}
avg_tat = (float)sum_tat / n;
avg_wt = (float)sum_wt / n;
// Printing table
cout<<"\nSNo\tAT\tBT\tCT\tTAT\tWT\n";
for (i = 0; i < n; i++)
cout<<p[i].index<<"\t"<<p[i].arr_t<<"\t"<<p[i].burst_t<<"\t"<<p[i].comp_t<<"\t"<<p[i].ta_t<<"
\t"<<p[i].wait_t<<"\n";
cout << "\nAverage turn-around time: " << avg_tat << "\n";
cout << "Average waiting time: " << avg_wt << "\n";
return 0;
}
```

OUTPUT:



```
"C:\Users\home\Desktop\My college work\ds lab\os\bin\Debug\os.exe"
Enter the number of processes   4
Enter the Arrival and burst time of each process
2 4
3 2
5 6
7 3

SNo     AT      BT      CT      TAT     WT
1       2       4       8       6       2
2       3       2       5       2       0
3       5       6       17      12      6
4       7       3       11      4       1

Average turn-around time: 6
Average waiting time: 2.25

Process returned 0 (0x0)   execution time : 287.826 s
Press any key to continue.
```

LEARNING OUTCOMES:

1. Short processes are handled very quickly.
2. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.
3. 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.

CONCLUSION:
We have successfully implemented the SRTF principle in CPU scheduling and calculated the average waiting time and average turn around time.

PROGRAM 5

AIM:
WAP to implement Round robin process scheduling algorithm and calculate waiting time and average turn around time.

THEORY:
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 fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.
- Completion Time: Time at which process completes its execution.
- Turn Around Time: Time Difference between completion time and arrival time.
- Turn Around Time = Completion Time – Arrival Time
- Waiting Time(W.T): Time Difference between turn around time and burst time.
- Waiting Time = Turn Around Time – Burst Time

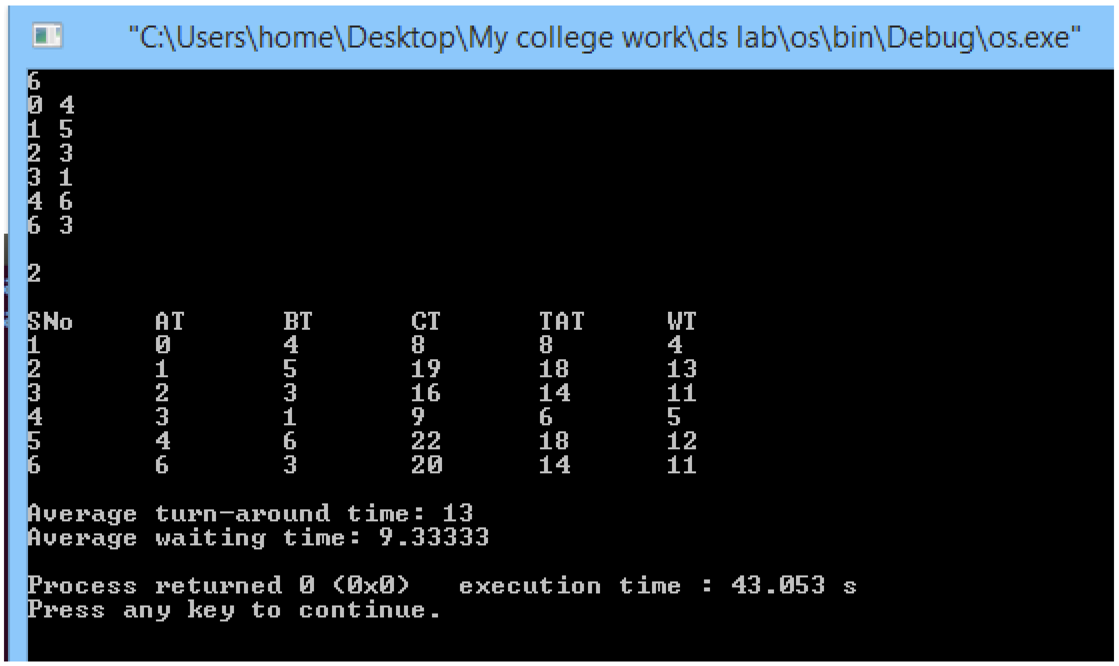
ALGORITHM:
```
Step 1: START
Step 2: Take necessary inputs.
Step 3: Using queue calculate the completion time.
Step 4: Using the completion time calculate the average waiting and turn around time.
Step 5: Print the output.
Step 6: END
```

CODE:
```
#include <iostream>
#include <queue>
using namespace std;

class Process
{
        public:
        int index, arr_t, burst_t, comp_t, wait_t, ta_t, rem_t;
};

Process p[100];
int n;

int main()
{
        int i;
```

```
                queue <Process> q;

                cin >> n;
                for(i=0; i<n; i++)
                {
                cin >> p[i].arr_t;
                cin >> p[i].burst_t;
                p[i].rem_t = p[i].burst_t;
                p[i].index = i+1;
                }

                int quant;
                cin>>quant;

i = 0;
int pInQue=0;
int time = p[0].arr_t;
while(time == p[i].arr_t && i<n)
{
   q.push(p[i]);
   pInQue++;
   i++;
}
Process processToBeScheduled;
while(!q.empty())
{
   processToBeScheduled = q.front();
   q.pop();
   int index = processToBeScheduled.index-1;
   int executionTime;
   int remainingTime = p[index].rem_t;
   if(remainingTime >= quant)
      executionTime = quant;
   else
      executionTime = remainingTime;

   p[index].rem_t = remainingTime - executionTime;
   time = time + executionTime;

   int j= pInQue;
   while(j<n && time >= p[j].arr_t)
   {
      q.push(p[j]);
      pInQue++;
      j++;
   }
```

```
        if(p[index].rem_t == 0)
           p[index].comp_t = time;
        else
           q.push(p[index]);

        if(q.empty() && pInQue!=n)
        {
           time = p[j].arr_t;
           while(time >= p[j].arr_t && j<n)
           {
              q.push(p[j]);
              pInQue++;
              j++;
           }
        }
     }

     // Calculation of waiting and turn-around time
     int sum_tat = 0, sum_wt = 0;
     float avg_tat, avg_wt;

     for (i = 0; i < n; i++)
     {
        p[i].ta_t = p[i].comp_t - p[i].arr_t;
        p[i].wait_t = p[i].ta_t - p[i].burst_t;
     }

     // Calculation of average waiting and turn-around time
     for (i = 0; i < n; i++)
     {
        sum_tat += p[i].ta_t;
        sum_wt += p[i].wait_t;
     }
     avg_tat = (float)sum_tat / n;
     avg_wt = (float)sum_wt / n;

     // Printing table
     cout<<"\nSNo\tAT\tBT\tCT\tTAT\tWT\n";
     for (i = 0; i < n; i++)
cout<<p[i].index<<"\t"<<p[i].arr_t<<"\t"<<p[i].burst_t<<"\t"<<p[i].comp_t<<"\t"<<p[i].ta_t<<"
\t"<<p[i].wait_t<<"\n";
     cout << "\nAverage turn-around time: " << avg_tat << "\n";
     cout << "Average waiting time: " << avg_wt << "\n";
return 0;
}
```

OUTPUT:



"C:\Users\home\Desktop\My college work\ds lab\os\bin\Debug\os.exe"

```
6
0 4
1 5
2 3
3 1
4 6
6 3

2

SNo     AT      BT      CT      TAT     WT
1       0       4       8       8       4
2       1       5       19      18      13
3       2       3       16      14      11
4       3       1       9       6       5
5       4       6       22      18      12
6       6       3       20      14      11

Average turn-around time: 13
Average waiting time: 9.33333

Process returned 0 (0x0)   execution time : 43.053 s
Press any key to continue.
```

LEARNING OUTCOMES:
It is easy to implement and starvation-free as all processes get good share of CPU.

CONCLUSION:
We have successfully implemented the Round Robin principle in CPU scheduling and calculated the average waiting time and average turn around time.

PROGRAM 6

AIM:
WAP to implement Non-preemptive Priority Scheduling process scheduling algorithm and calculate waiting time and average turn around time.

THEORY:

- **Priority Scheduling :**
  Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on. Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other requirement.

- **Turn Around Time :**

The time difference between completion and arrival time i.e.,

Turn Around Time = Completion time – Arrival time

- **Waiting Time :**

The time difference between turn around time and burst time i.e.,

Waiting Time = Turnaround Time - Burst Time

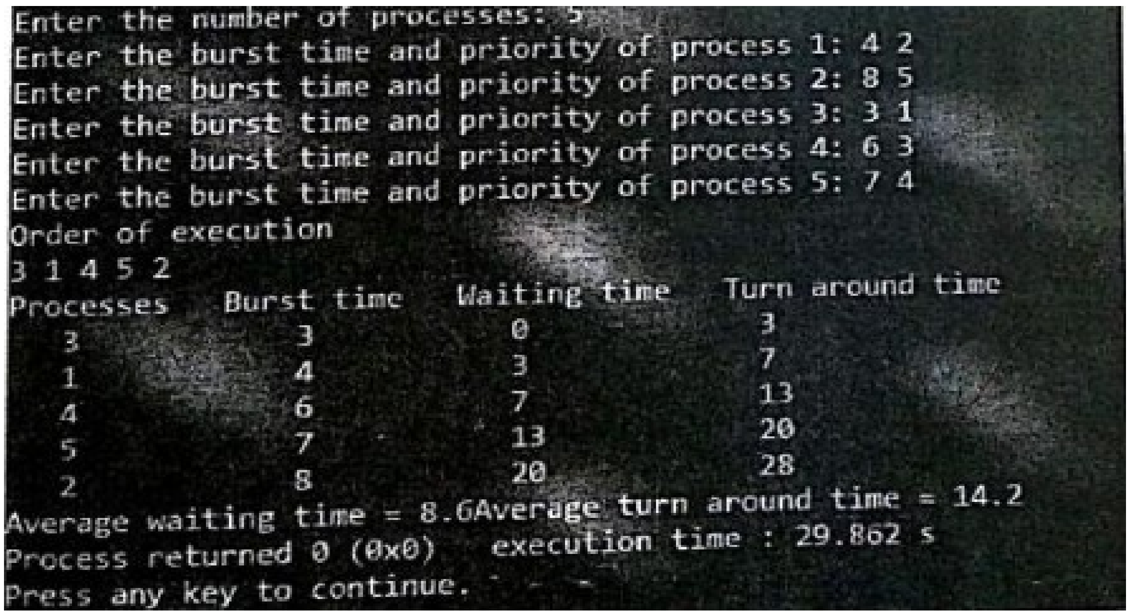
ALGORITHM :
1. Sort all the processes in increasing order according to burst time.
2. Then simply, apply FCFS.

CODE:
```
#include <iostream>
using namespace std;
struct Process
{
int pid;
int bt;
int priority;
};
void waitingTime(Process p[],int n,intwt[])
{
wt[0] = 0;
for(int i = 1; i< n; i++)
{
wt[i] = p[i-1].bt + wt[i-1];
}
}
void turnAroundTime(Process p[],int n,intwt[],int tat[])
{
for(int i = 0; i< n; i++)
```

```
{
tat[i] = p[i].bt + wt[i];
}
}
void avgTime(Process p[],int n)
{
int wt[20],tat[20],total_wt = 0,total_tat = 0;
waitingTime(p,n,wt);
turnAroundTime(p,n,wt,tat);
cout<<"\nProcesses Burst Time Waiting Time Turn Around Time\n";
for(int i = 0; i< n; i++)
{
total_wt += wt[i];
total_tat += tat[i];
cout<<p[i].pid<<" "<<p[i].bt<<" "<<wt[i]<<" "<<tat[i]<<endl;
}
cout<<"Average waiting time = "<<total_wt/(float)(n)<<endl;
cout<<"Average turn around time = "<<total_tat/(float)(n)<<endl;
}
void priorityScheduling(Process p[],int n)
{
for(int i = 0; i< n; i++)
{
for(int j = 0; j < n-1; j++)
{
if(p[j].priority> p[j+1].priority)
{
Process temp = p[j];
p[j] = p[j+1];
p[j+1] = temp;
}
}
}
cout<<"Order of execution : \n";
for(int i = 0; i< n; i++)
{
cout<<p[i].pid<<" ";
}
cout<<endl;
avgTime(p,n);
}
int main()
{
Process p[20];
int n;
cout<<"Enter the no. of processes : ";
```

```
cin>>n;
for(int i = 0; i< n; i++)
{
p[i].pid = i+1;
cout<<"Enter the burst time and priority pf process "<<i+1<<": ";
cin>>p[i].bt>>p[i].priority;
}
priorityScheduling(p,n);
return 0;
}
```

OUTPUT:



```
Enter the number of processes: 5
Enter the burst time and priority of process 1: 4 2
Enter the burst time and priority of process 2: 8 5
Enter the burst time and priority of process 3: 3 1
Enter the burst time and priority of process 4: 6 3
Enter the burst time and priority of process 5: 7 4
Order of execution
3 1 4 5 2
Processes   Burst time   Waiting time   Turn around time
   3            3             0              3
   1            4             3              7
   4            6             7              13
   5            7             13             20
   2            8             20             28
Average waiting time = 8.6Average turn around time = 14.2
Process returned 0 (0x0)   execution time : 29.862 s
Press any key to continue.
```

LEARNING OUTCOMES:

1. Can be preemptive and non-preemptive.
2. Process with the highest priority is to be executed first
3. Processes with the same priority follows FCFS method

CONCLUSION:

We have successfully implemented the Priority Scheduling (Non preemptive) principle in CPU scheduling and calculated the average waiting time and average turn around time.

PROGRAM 7

AIM:
Write a program to implement banker's algorithm for purpose of deadlock avoidance.

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.
Following **Data structures** are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :
• It is a 1-d array of size **'m'** indicating the number of available resources of each type.
• Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$**.**

Max :
• It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
• Max[i, j] = k means process **Pi** may request at most **'k'** instances of resource type **Rj.**

Allocation :
• It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
• Allocation[i, j]= k means process **Pi** is currently allocated **'k'** instances of resource type$\mathbf{R_j}$

Need :
• It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
• Need [i, j] = k means process **Pi** currently need **'k'** instances of resource type **Rj**
for its execution.
• Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation[i] specifies the resources currently allocated to process $\mathbf{P_i}$ and **Need[i]** specifies the additional resources that process Pi may still request to complete its task.
Banker's algorithm consist of Safety algorithm and Resource request algorithm .

ALGORITHMS :

Safety Algorithm:
The algorithm for finding out whether or not a system is in a safe state can be described as follows:
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) Need[i]<= Work if no such i exists goto step (4)
3) Work = Work + Allocation[i] Finish[i] = true goto step (2)

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

Resource-Request Algorithm:
Let Request[i] be the request array for process Pi. Requesti [j] = k means process Pi wants k instances of resource type R_j. When a request for resources is made by process Pi, thefollowing actions are taken:

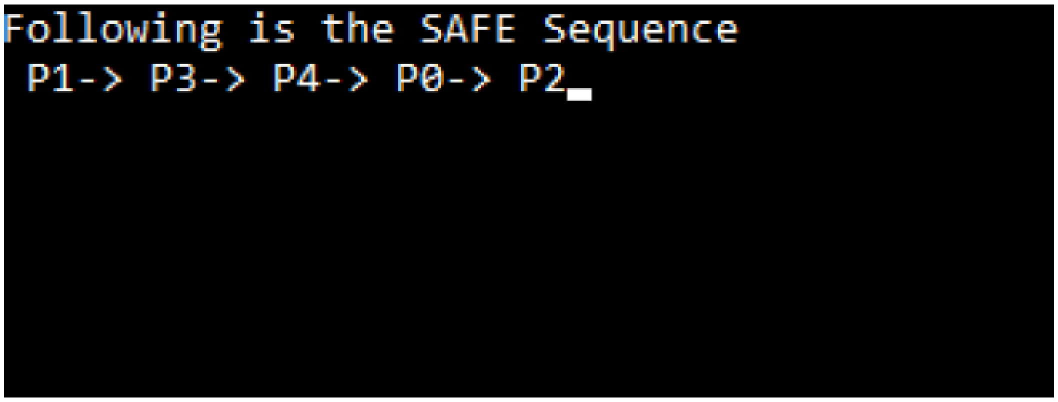
```
1)If Request[i]<= Need[i]
        Goto step (2) ;
Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2)If Request[i]<= Available Goto step (3);
otherwise, Pi must wait, since the resources are not available.

3)Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
Available = Available – Request[i]
Allocation[i] = Allocation[i] + Request[i]
Need[i] = Need[i]– Request[i]
```

CODE:

```
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{
// P0, P1, P2, P3, P4 are the Process names here
int n, m, i, j, k;
n = 5; // Number of processes
m = 3; // Number of resources
int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
{ 2, 0, 0 }, // P1
{ 3, 0, 2 }, // P2
{ 2, 1, 1 }, // P3
{ 0, 0, 2 } }; // P4
int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
{ 3, 2, 2 }, // P1
{ 9, 0, 2 }, // P2
{ 2, 2, 2 }, // P3
{ 4, 3, 3 } }; // P4
int avail[3] = { 3, 3, 2 }; // Available Resources
int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
f[k] = 0;
}
```

```
int need[n][m];
for (i = 0; i< n; i++) {
for (j = 0; j < m; j++)
need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
for (i = 0; i< n; i++) {
if (f[i] == 0) {
int flag = 0;
for (j = 0; j < m; j++) {
if (need[i][j] > avail[j])
flag = 1;
break;
}if (flag == 0) {
ans[ind++] = i;
for (y = 0; y < m; y++)
avail[y] += alloc[i][y];
f[i] = 1; }
}
}
cout<<"Following is the SAFE Sequence\n";
for (i = 0; i< n - 1; i++)
cout<<" P"<<ans[i]<<"->";
cout<<" P"<<ans[n - 1];
getch();
return 0;
}
```

OUTPUT:



```
Following is the SAFE Sequence
 P1-> P3-> P4-> P0-> P2
```

LEARNING OUTCOME:
We learnt that if we have a finite number of resources and processes then Banker’s Algorithm can be used to avoid deadlocks. This algorithm is less restrictive than deadlock prevention.

CONCLUSION:
We have successfully implemented the Banker’s Algorithm to detect a safe state from the list of processes.

PROGRAM 8

AIM:
WAP to implement Producer Consumer problem.

THEORY:
We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.
When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.
As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

ALGORITHM:

```
mutex = 1
Full = 0 // Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially
```

Solution for Producer –

```
do{
//produce an item
wait(empty);
wait(mutex);

//place in buffer
signal(mutex);
signal(full);
}(while true)
```

Solution for Consumer –

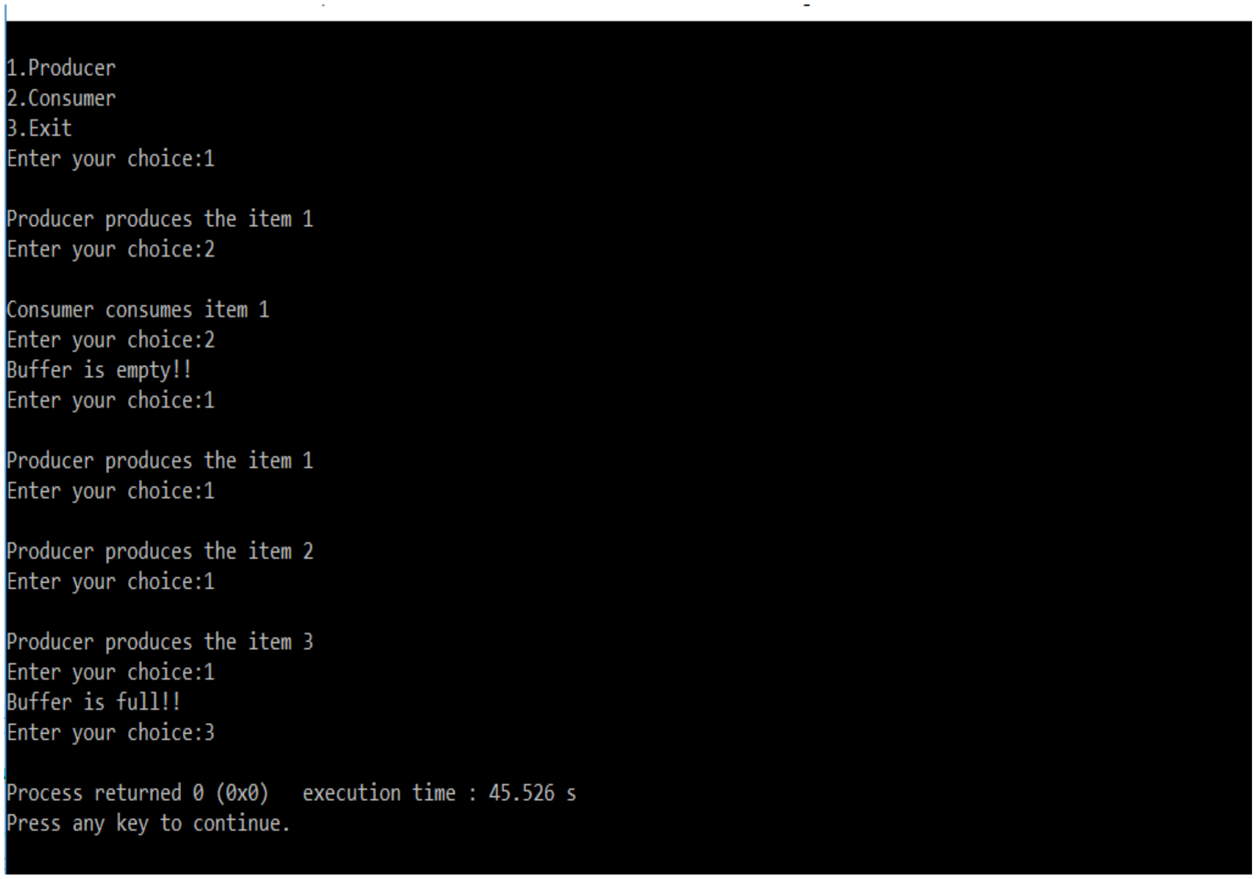
```
do{
//remove an item from buffer
wait(full);
wait(mutex);

//consumes item
signal(mutex);
signal(empty);

}(while true)
```

CODE:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;

int main()
{
        int n;
        void producer();
        void consumer();
        int wait(int);
        int signal(int);
        printf("\n1.Producer\n2.Consumer\n3.Exit");
        while(1)
        {       printf("\nEnter your choice:");
                scanf("%d",&n);
                switch(n)
                {
                        case 1: if((mutex==1)&&(empty!=0))
                                                producer();
                                else
                                                printf("Buffer is full!!");
                                break;
                        case 2: if((mutex==1)&&(full!=0))
                                                consumer();
                                else
                                                printf("Buffer is empty!!");
                                break;
                        case 3:
                                exit(0);
                                break;
                }}
        return 0;
}

int wait(int s)
{       return (--s);
}

int signal(int s)
```

```
{	return(++s);
}

void producer()
{	mutex=wait(mutex)
	full=signal(full);
	empty=wait(empty);
	x++;
	printf("\nProducer produces the item %d",x);
	mutex=signal(mutex);
}

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

OUTPUT:



```
1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:1
Buffer is full!!
Enter your choice:3

Process returned 0 (0x0)   execution time : 45.526 s
Press any key to continue.
```

LEARNING OUTCOMES

The order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock. It is important to note here that though mutex seems to work as a semaphore with value of 1 (binary semaphore), but there is difference in the fact that mutex has ownership concept. Ownership means that mutex can only be "incremented" back (set to 1) by the same process that "decremented" it (set to 0), and all other tasks wait until mutex is available for decrement (effectively meaning that resource is available), which ensures mutual exclusivity and avoids deadlock. Thus using mutexes improperly can stall many processes when exclusive access is not required, but mutex is used instead of semaphore.

CONCLUSION:

We have successfully implemented the Producer-Consumer problem.

PROGRAM 9

AIM:
WAP to implement First Come First Serve(FCFS) algorithm for disk scheduling.

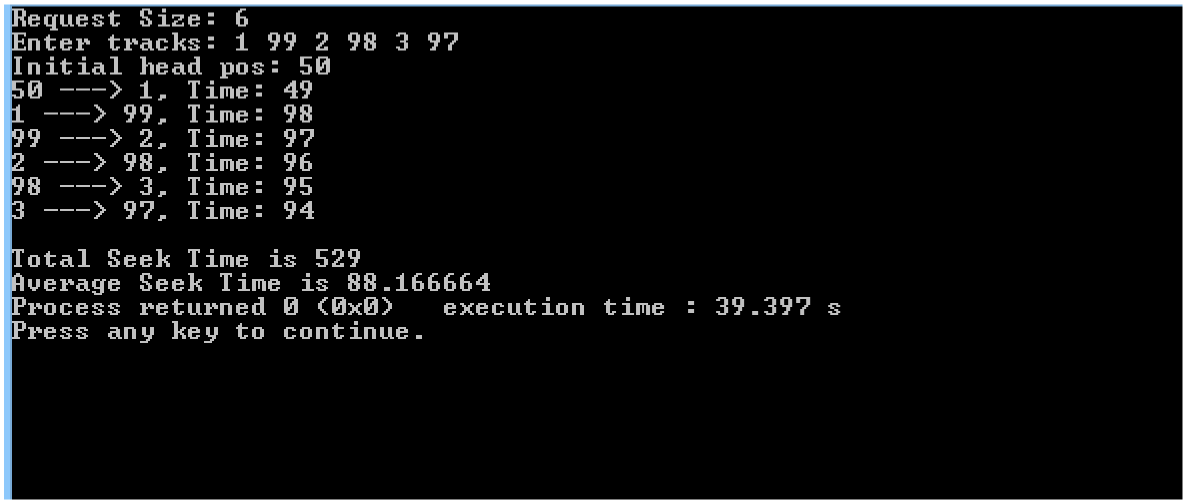
THEORY:
Disk scheduling is is done by operating systems to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling. Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
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.

ALGORITHM:

1. Input the tracks in queue[1 to n]
2. Input initial head in queue[0]
3. Traverse the requests in arrival order
4. Print seek time by calculating difference from next seek request
5. Add seek time to total seek time
6. Average seek time = total seek time / n
7. Print total seek and average seek time

CODE:
```
#include<stdio.h>
int main()
{
int queue[100], n, head, i, seek = 0, diff;
float avg;
printf("Request Size: ");
scanf("%d", &n);
printf("Enter tracks: ");
for(i = 1; i <= n; i++)
{
scanf("%d", &queue[i]);
}
printf("Initial head pos: ");
scanf("%d", &head);
queue[0] = head;
for(i = 0; i < n; i++)
{
diff = queue[i + 1] - queue[i];
if(diff < 0)
diff *= -1;
seek += diff;
printf("%d ---> %d, Time: %d\n", queue[i], queue[i + 1], diff);
}
printf("\nTotal Seek Time is %d\t", seek);
avg = seek / (float)n;
printf("\nAverage Seek Time is %f\t", avg);
return 0;
}
```

OUTPUT:



```
Request Size: 6
Enter tracks: 1 99 2 98 3 97
Initial head pos: 50
50 ---> 1, Time: 49
1 ---> 99, Time: 98
99 ---> 2, Time: 97
2 ---> 98, Time: 96
98 ---> 3, Time: 95
3 ---> 97, Time: 94

Total Seek Time is 529
Average Seek Time is 88.166664
Process returned 0 (0x0)   execution time : 39.397 s
Press any key to continue.
```

LEARNING OUTCOMES:

This algorithm is the simplest to implement but isn’t very efficient in most use cases.
Advantages of FCFS disk scheduling algorithm :-

- Every request gets a fair chance
- No starvation
- Easiest to implement

Disadvantages of FCFS disk scheduling algorithm :-

- Large waiting and response times possible
- Very inefficient

CONCLUSION:

We have successfully implemented the FCFS principle in Disk scheduling and calculated the total seek time and average seek time.

PROGRAM 10

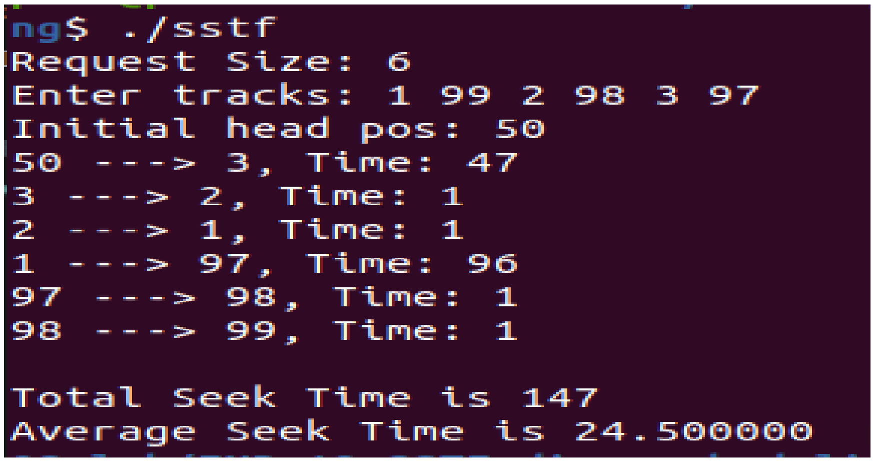
AIM:
WAP to implement Shortest Seek Time First (SSTF) algorithm for disk scheduling.

THEORY:
In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

ALGORITHM:
1. Let Request array represents an array storing indexes of tracks that have been requested.
2. 'head' is the position of disk head.
3. Find the positive distance of all tracks in the request array from head.
4. Find a track from requested array which has not been accessed/serviced yet and has
5. minimum distance from head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 2 until all tracks in request array have not been serviced.

CODE:
```
#include<stdio.h>
#include<math.h>
int queue[100], n, head, currentPos = 0;
void sort()
{
int i, j, temp;
for(i = currentPos; i < n + 1; i++)
{
        for(j = currentPos; j < n; j++)
        {
                if((abs(queue[j + 1] - head)) < (abs(queue[j] - head)))
                        {
                        temp = queue[j];
                        queue[j] = queue[j + 1];
                        queue[j + 1] = temp;
                        }
        }
}
}
int main()
{
int i, k, seek = 0, diff;
float avg;
51printf("Request Size: ");
scanf("%d", &n);
printf("Enter tracks: ");
for(i = 1; i <= n; i++)
{
scanf("%d", &queue[i]);
}
printf("Initial head pos: ");
scanf("%d", &head);
queue[0] = head;
currentPos = 1;
sort();
for(i = 0; i < n; i++)
{
diff = abs(queue[i + 1] - queue[i]);
seek += diff;
printf("%d ---> %d, Time: %d\n", queue[i], queue[i + 1], diff);
}
currentPos++;
head = queue[i + 1];
sort();
printf("\nTotal Seek Time is %d\t", seek);
avg = seek / (float)n;
printf("\nAverage Seek Time is %f\t", avg);
return 0;
}
```

OUTPUT:



```
ng$ ./sstf
Request Size: 6
Enter tracks: 1 99 2 98 3 97
Initial head pos: 50
50 ---> 3, Time: 47
3 ---> 2, Time: 1
2 ---> 1, Time: 1
1 ---> 97, Time: 96
97 ---> 98, Time: 1
98 ---> 99, Time: 1

Total Seek Time is 147
Average Seek Time is 24.500000
```

LEARNING OUTCOMES:

This algorithm is an improvement over FCFS
Advantages of SSTF:

- Average Response Time decreases
- Throughput increases

Disadvantages of SSTF:

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favours only some requests

CONCLUSION:

We have successfully implemented the SSTF principle in Disk scheduling and calculated the total seek time and average seek time.

PROGRAM 11

AIM:
WAP to implement SCAN Disk scheduling algorithm .

THEORY:

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

ALGORITHM:

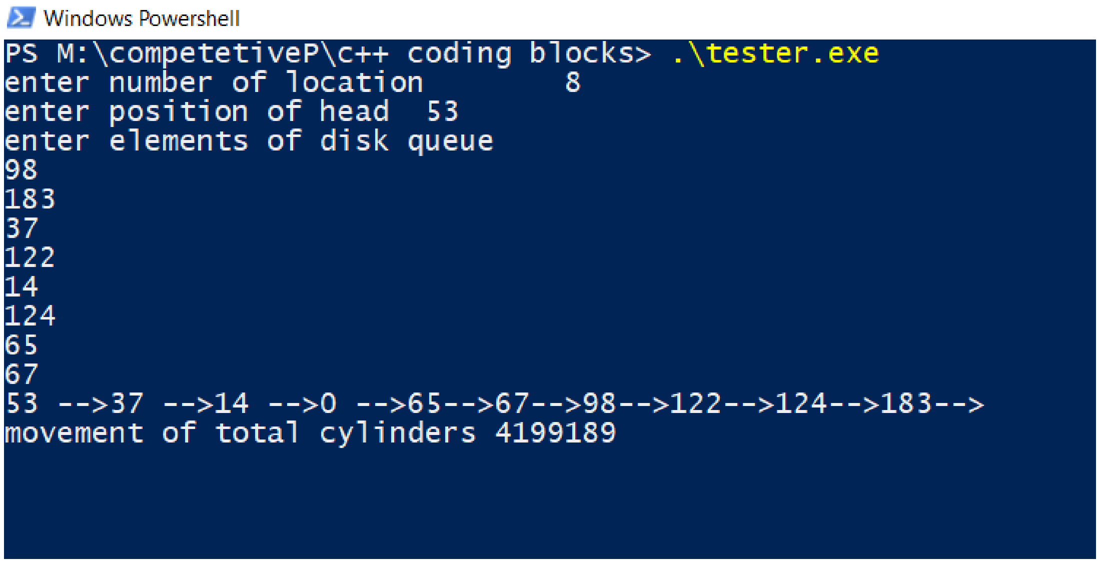
1. Head starts from one end of the disk and move towards the other end servicing all the requests in between.
2. After reaching the other end, head reverses its direction and move towards the starting end servicing all the requests in between.
3. The same process repeats.

CODE:

```
#include<conio.h>
#include<stdio.h>
int main()
{
 int i,j,sum=0,n;
 int d[20];
 int disk;   //loc of head
 int temp,max;
 int dloc;   //loc of disk in array
 clrscr();
 printf("enter number of location\t");
 scanf("%d",&n);
 printf("enter position of head\t");
 scanf("%d",&disk);
 printf("enter elements of disk queue\n");
 for(i=0;i<n;i++)
 {
 scanf("%d",&d[i]);
 }
 d[n]=disk;
 n=n+1;
 for(i=0;i<n;i++)    // sorting disk locations
 {
  for(j=i;j<n;j++)
  {
   if(d[i]>d[j])
```

```
    {
    temp=d[i];
    d[i]=d[j];
    d[j]=temp;
    }
  }
 }
 max=d[n];
 for(i=0;i<n;i++)   // to find loc of disc in array
 {
 if(disk==d[i]) { dloc=i; break;  }
 }
 for(i=dloc;i>=0;i--)
 {
 printf("%d -->",d[i]);
 }
 printf("0 -->");
 for(i=dloc+1;i<n;i++)
 {
 printf("%d-->",d[i]);
 }
 sum=disk+max;
      printf("\nmovement of total cylinders %d",sum);
 getch();
 return 0;
}
```

OUTPUT:



Windows Powershell

```
PS M:\competetiveP\c++ coding blocks> .\tester.exe
enter number of location        8
enter position of head  53
enter elements of disk queue
98
183
37
122
14
124
65
67
53 -->37 -->14 -->0 -->65-->67-->98-->122-->124-->183-->
movement of total cylinders 4199189
```

LEARNING OUTCOMES:

Advantages of SCAN scheduling algorithm:

- High throughput
- Low variance of response time
- Average response time

Disadvantages of SCAN scheduling algorithm:

- Long waiting time for requests for locations just visited by disk arm

CONCLUSION:

We have successfully implemented the SCAN Disk scheduling process and calculated the total seek time and average seek time.

PROGRAM 12

AIM:
WAP to implement C-SCAN Disk scheduling algorithm .

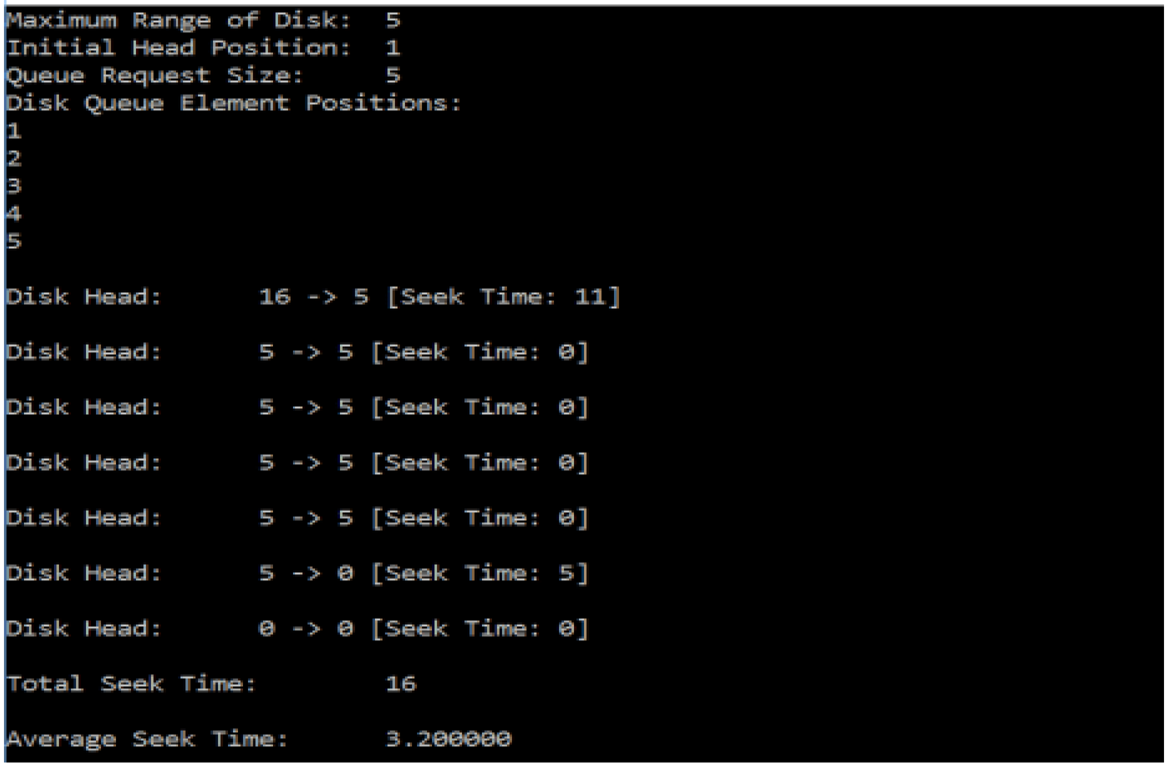
THEORY:
In C-SCAN algorithm, the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.

ALGORITHM:

- Circular-SCAN Algorithm is an improved version of the SCAN Algorithm.
- Head starts from one end of the disk and move towards the other end servicing all the requests in between.
- After reaching the other end, head reverses its direction.
- It then returns to the starting end without servicing any request in between.
- The same process repeats.

CODE:
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int queue1[30], queue2[30], queue3[30];
int limit, disk_head, count = 0, j, seek_time = 0, range, diff;
int t1, t2 = 0, t3 = 0;
float avg_seek_time;
printf("Maximum Range of Disk:\t");
scanf("%d", &range);
printf("Initial Head Position:\t");
scanf("%d", &disk_head);
printf("Queue Request Size:\t");
scanf("%d", &limit);
printf("Disk Queue Element Positions:\n");
while(count < limit)
{
scanf("%d", &t1);
if(t1 >= disk_head)
{
queue1[t2] = t1;
t2++;}
else
{queue2[t3] = t1;
t3++;}
count++;}
count = 0;
while(count < t2 - 1)
{j = count + 1;
while(j < t2)
{if(queue1[count] > queue1[j])
{t1 = queue1[count];
queue1[count] = queue1[j];
queue1[j] = t1;
}
j++;}
count++;}
count = 0;
while(count < t3 - 1)
{j = count + 1;
while(j < t3)
{if(queue2[count] > queue2[j])
{t1 = queue2[count];
queue2[count] = queue2[j];
queue2[j] = t1;}
j++;}
count++;}
count = 1;       j = 0;
while(j < t2)
{queue3[count] = queue1[j];
queue3[count] = range;
queue3[count + 1] = 0;
count++;
j++;}
count = t2 + 3;            j = 0;
while(j < t3)
{queue3[count] = queue2[j];
queue3[0] = disk_head;
count++;
j++;}
for(j = 0; j <= limit + 1; j++)
{diff = abs(queue3[j + 1] - queue3[j]);
seek_time = seek_time + diff;
printf("\nDisk Head:\t%d -> %d [Seek Time: %d]\n", queue3[j], queue3[j + 1], diff);}
printf("\nTotal Seek Time:\t%d\n", seek_time);
avg_seek_time = seek_time / (float)limit;
printf("\nAverage Seek Time:\t%f\n", avg_seek_time);
return 0;
}
```

OUTPUT:



```
Maximum Range of Disk:  5
Initial Head Position:  1
Queue Request Size:     5
Disk Queue Element Positions:
1
2
3
4
5

Disk Head:      16 -> 5 [Seek Time: 11]

Disk Head:      5 -> 5 [Seek Time: 0]

Disk Head:      5 -> 5 [Seek Time: 0]

Disk Head:      5 -> 5 [Seek Time: 0]

Disk Head:      5 -> 5 [Seek Time: 0]

Disk Head:      5 -> 0 [Seek Time: 5]

Disk Head:      0 -> 0 [Seek Time: 0]

Total Seek Time:        16

Average Seek Time:      3.200000
```

LEARNING OUTCOMES:

Advantages:

- Provides more uniform wait time compared to SCAN
- High throughput
- Low variance of response time
- Average response time

CONCLUSION:

We have successfully implemented the C-SCAN Disk scheduling process and calculated the total seek time and average seek time.

PROGRAM 13

AIM:
WAP to implement LOOK Disk scheduling algorithm .

THEORY:
It is similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

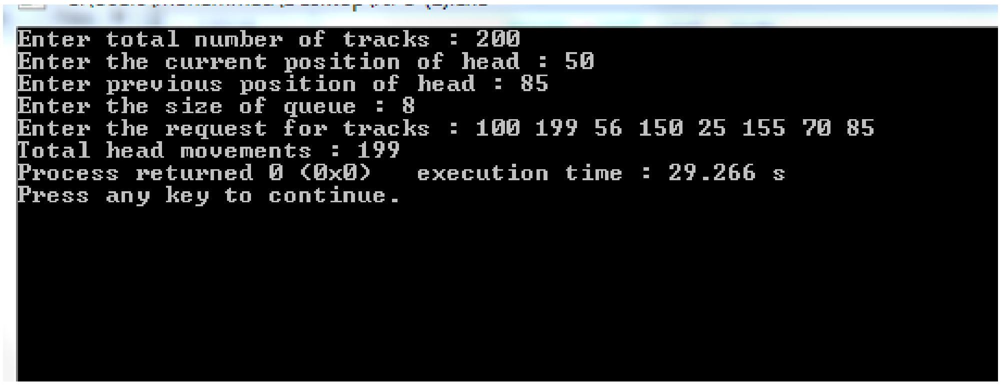
ALGORITHM:

1. Head starts from the first request at one end of the disk and moves towards the last request at the other end servicing all the requests in between.
2. After reaching the last request at the other end, head reverses its direction.
3. It then returns to the first request at the starting end servicing any request in between.
4. The same process repeats.

CODE:
```
#include<iostream>
#include<stdlib.h>
using namespace std;
class look_disk
{
int ref[100];
int ttrk,cur,size,prev;
int sort();
public:
void getdata();
void total_move();
};
int look_disk::sort()
{
int temp;
for(int i=0;i<size-1;i++)
for(int y=0;y<size-1;y++)
if(ref[y]>ref[y+1])
{
temp=ref[y+1];
ref[y+1]=ref[y];
ref[y]=temp;
}
for(int i=0;i<size;i++)
if(ref[i]>cur)
return i;
return size;}
void look_disk::getdata()
{cout<<"Enter total number of tracks : ";
cin>>ttrk;
ttrk--;
cout<<"Enter the current position of head : ";
cin>>cur;
cout<<"Enter previous position of head : ";
cin>>prev;
cout<<"Enter the size of queue : ";
cin>>size;
cout<<"Enter the request for tracks : ";
for(int i=0;i<size;i++)
cin>>ref[i];
}
void look_disk::total_move()
{int num=cur,move=0,ind,dir=cur-prev;
ind=sort();
if(dir>0)
{for(int i=ind;i<size;i++)
{move+=ref[i]-num;
num=ref[i];}
if(ind!=0)
{if(ind==size)
move+=num-ref[ind-1];
else
move+=ref[size-1]-ref[ind-1];
num=ref[ind-1];
for(int i=ind-2;i>=0;i--)
{move+=num-ref[i];
num=ref[i];
}}}
else
{for(int i=ind-1;i>=0;i--)
{move+=num-ref[i];
num=ref[i];}
if(ind==0)
move+=ref[ind]-num;
else if(ind!=size)
move+=ref[ind]-ref[0];
num=ref[ind];
for(int i=ind+1;i<size;i++)
{move+=ref[i]-num;
num=ref[i];
}}
cout<<"Total head movements : "<<move;
```

```
}
int main()
{look_disk look;
look.getdata();
look.total_move();
return 0;
}
```

OUTPUT:



```
Enter total number of tracks : 200
Enter the current position of head : 50
Enter previous position of head : 85
Enter the size of queue : 8
Enter the request for tracks : 100 199 56 150 25 155 70 85
Total head movements : 199
Process returned 0 (0x0)   execution time : 29.266 s
Press any key to continue.
```

LEARNING OUTCOMES:

<u>Advantages-</u>

1. It does not causes the head to move till the ends of the disk when there are no requests to be serviced.
2. It provides better performance as compared to SCAN Algorithm.
3. It does not lead to starvation.
4. It provides low variance in response time and waiting time.

<u>Disadvantages-</u>

- There is an overhead of finding the end requests.
- It causes long waiting time for the cylinders just visited by the head.

CONCLUSION:

We have successfully implemented the LOOK Disk scheduling process and calculated the total head movements.

PROGRAM 14

AIM:
WAP to implement C-LOOK Disk scheduling algorithm .

THEORY:
As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm inspite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

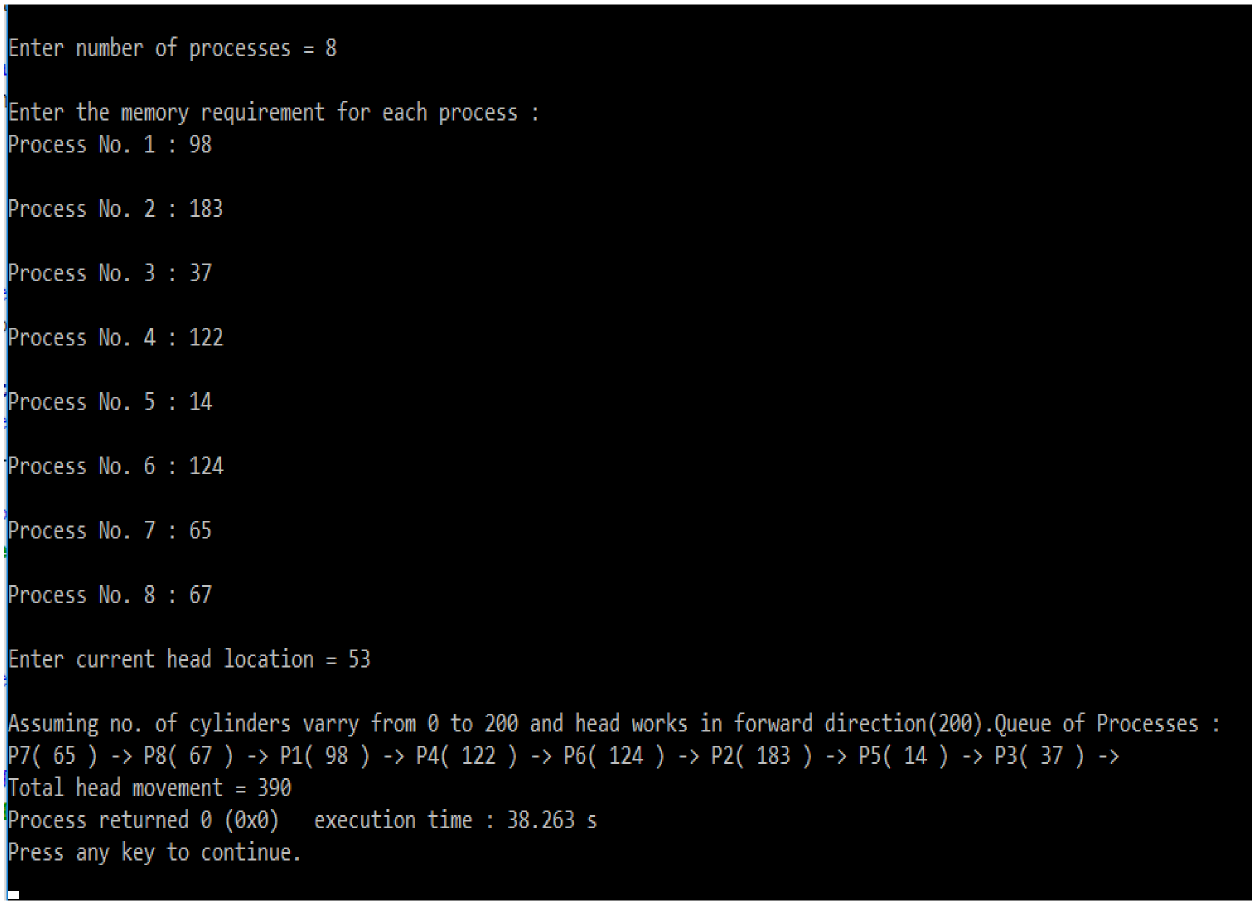
ALGORITHM:

- Head starts from the first request at one end of the disk and moves towards the last request at the other end servicing all the requests in between.
- After reaching the last request at the other end, head reverses its direction.
- It then returns to the first request at the starting end without servicing any request in between.
- The same process repeats

CODE:
```
#include<iostream>
using namespace std;
int mod(int a){
        if (a<0){
                return -a;
        }
        return a;}
int find_min(int *q, int n){
        int key=0;
        for (int i=0;i<n;i++){
                if (q[i] != -1 && q[key] > q[i]){
                        key = i;
                }
        }
        return q[key];
}
int check_prcss(int p, int h){
        if (p == -1){
                return 0;
        }
        else{
                if (p >= h)
                  return 1;
                return 0;
        }
}
int forward_movement(int *q, int ch,int n){
        int idx =-1;
        for(int i=0;i<n;i++){
                if(check_prcss(q[i],ch) == 1){
                        if (idx ==-1 || q[idx] > q[i]){
                          idx=i;}
                }}
        return idx;
}
int CLOOK(int *q, int n, int h){
        int head_movement = 0;
        int prev_head = h;
        int curr_head = h;
        int counter = 0;
        int idx = -1;
        cout<<"Queue of Processes : \n";
        while(counter < n){
for(int i=0;i<n;i++){
        idx = forward_movement(q,prev_head,n);
                   if(idx!= -1){
                   cout<<"P"<<idx+1<<"( "<<q[idx]<<" )"<<" ->";
             curr_head = q[idx];
                      int diff = mod(prev_head - curr_head);
                      head_movement = head_movement + diff;
                      prev_head = curr_head;
                      q[idx] =-1;
                      counter++;          }
}
                int pre = find_min(q,n);
                head_movement = head_movement + (prev_head - pre);
                prev_head = pre;
}
        cout<<"\nTotal head movement = ";
        return head_movement;
}
int main(){
        int prcss_no;
        cout<<"\nEnter number of processes = ";
        cin>>prcss_no;
        int *queue;
        queue=new int[prcss_no];
        cout<<"\nEnter the memory requirement for each process :";
        for(int i=0;i<prcss_no;i++)
```

```
            {
              cout<<"\nProcess No. "<<i+1<<" : ";
              cin>>queue[i];
            }

          int head;
          cout<<"\nEnter current head location = ";
          cin>>head;

cout<<"\nAssuming no. of cylinders varry from 0 to 200 and head works in forward
direction(200).";
          cout<<CLOOK(queue,prcss_no,head);
          delete queue;
          return 0;
}
```

OUTPUT:



```
Enter number of processes = 8

Enter the memory requirement for each process :
Process No. 1 : 98

Process No. 2 : 183

Process No. 3 : 37

Process No. 4 : 122

Process No. 5 : 14

Process No. 6 : 124

Process No. 7 : 65

Process No. 8 : 67

Enter current head location = 53

Assuming no. of cylinders varry from 0 to 200 and head works in forward direction(200).Queue of Processes :
P7( 65 ) -> P8( 67 ) -> P1( 98 ) -> P4( 122 ) -> P6( 124 ) -> P2( 183 ) -> P5( 14 ) -> P3( 37 ) ->
Total head movement = 390
Process returned 0 (0x0)   execution time : 38.263 s
Press any key to continue.
```

LEARNING OUTCOMES:

The huge jump from one end request to the other is not considered as a head movement as the cylinders are treated as a circular list. Its advantage is that it is better than Cscan algorithm in terms of head movement.

Advantages-

1. It does not causes the head to move till the ends of the disk when there are no requests to be serviced.
2. It reduces the waiting time for the cylinders just visited by the head.
3. It provides better performance as compared to LOOK Algorithm.
4. It does not lead to starvation.
5. It provides low variance in response time and waiting time.

Disadvantages-

- There is an overhead of finding the end requests.

CONCLUSION:

We have successfully implemented the C-LOOK Disk scheduling process and calculated the total seek time and average seek time.

PROGRAM 15

AIM:
WAP to implement Least Recently Used (LRU) page replacement algorithm .

THEORY:
In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page.
In Least Recently Used (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely

ALGORITHM:

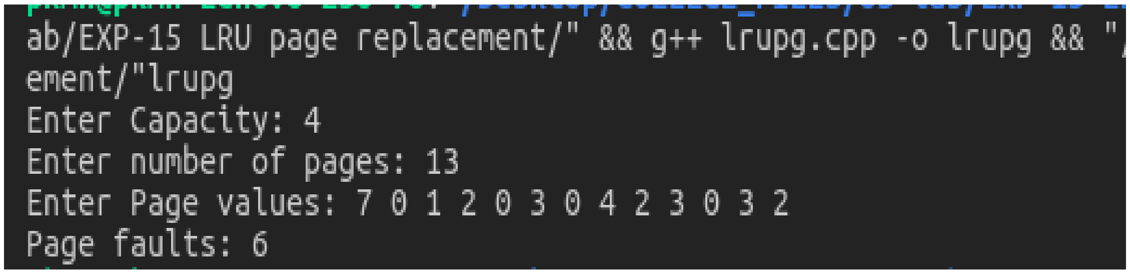
```
1. Start traversing the pages.
    i. If set holds less pages than capacity.
        a) Insert page into the set one by one until the size of set reaches
           capacity or all page requests are processed.
        b) Simultaneously maintain the recent occurred index of each page in
           a map called indexes.
        c) Increment page fault
    ii. Else
        a) If current page is present in set, do nothing.
    iii. Else
        a) 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.
        b) Replace the found page with current page.
        c) Increment page faults.
        d) Update index of current page
2. Return page faults.
```

CODE:

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

int pageFaults(int pages[], int n, int capacity)
{
        unordered_set<int> s;
   unordered_map<int, int> indexes;
   int page_faults = 0;
        for (int i=0; i<n; i++)
        {
           if (s.size() < capacity)
```

```
                {
                        if (s.find(pages[i])==s.end())
                        {
                                s.insert(pages[i]);
        page_faults++;
                        }
                        indexes[pages[i]] = i;
                }
      else
                {
                        if (s.find(pages[i]) == s.end())
                        {
                                int lru = INT_MAX, val;
                                for (auto it=s.begin(); it!=s.end(); it++)
                                {
                                        if (indexes[*it] < lru)
                                        {
                                                lru = indexes[*it];
                                                val = *it;                      }
                                }
           s.erase(val);
           s.insert(pages[i]);
           page_faults++;
                        }
           indexes[pages[i]] = i;
                }        }
        return page_faults;
}
int main()
{
int n, capacity;
cout << "Enter Capacity: ";
cin >> capacity;
cout << "Enter number of pages: ";
cin >> n;
int pages[n];
cout << "Enter Page values: ";
for (int i = 0; i < n; i++)
{
cin >> pages[i];
}
cout << "Page faults: " << pageFaults(pages, n, capacity) << endl;
return 0;
}
```

OUTPUT:



```
ab/EXP-15 LRU page replacement/" && g++ lrupg.cpp -o lrupg && "
ement/"lrupg
Enter Capacity: 4
Enter number of pages: 13
Enter Page values: 7 0 1 2 0 3 0 4 2 3 0 3 2
Page faults: 6
```

LEARNING OUTCOMES:

In this algorithm , the page that has not been used for the longest period of time has to be replaced.
Advantages of LRU Page Replacement Algorithm :-

- It is amenable to full statistical analysis.
- Never suffers from Belady’s anomaly

CONCLUSION:

We have successfully implemented the LRU page Replacement algorithm and calculated the number of page faults.

PROGRAM 16

AIM:
WAP to implement page replacement algorithm .

THEORY:
In operating systems that use paging for memory management, page replacement algorithm are

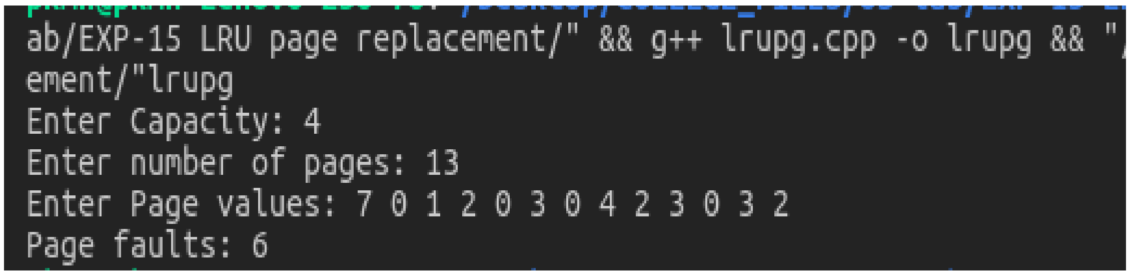
ALGORITHM:

```
3. Start traversing the pages.
    iv.   If set holds less pages than capacity.
          a) Insert page into the set one by one until the size of set reaches
             capacity or all page requests are processed.
          b) Simultaneously maintain the recent occurred index of each page in
             a map called indexes.
          c) Increment page fault
    v.    Else
          b) If current page is present in set, do nothing.
    vi.   Else
          e) 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.
          f) Replace the found page with current page.
          g) Increment page faults.
          h) Update index of current page
4. Return page faults.
```

CODE:

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

OUTPUT:



```
ab/EXP-15 LRU page replacement/" && g++ lrupg.cpp -o lrupg && "
ement/"lrupg
Enter Capacity: 4
Enter number of pages: 13
Enter Page values: 7 0 1 2 0 3 0 4 2 3 0 3 2
Page faults: 6
```

LEARNING OUTCOMES:
In this algorithm , the page that has not been used for the longest period of time has to be

CONCLUSION:

We have successfully implemented the LRU page Replacement algorithm and calculated the number of page faults.