Divya Gyan College

(Tribhuvan University)
Kamaladi Mod, Kathmandu

Lab Report of Operating System (CACS-251)

Faculty of Humanities and Social Sciences
Tribhuvan University
Kirtipur Nepal

Submitted By

Name: Manish Kumar Shrestha
Exam roll no: 75102116

Submitted To

Divya Gyan College
Department of Bachelor of Computer Application
Kamaladi Mod, Kathmandu, Nepal.

Table of Contents

<u>Title</u>: Installation of Linux in Virtual Machine

<u>Objectives:</u>

- To Install Linux in Virtual Machine and use some Linux commands

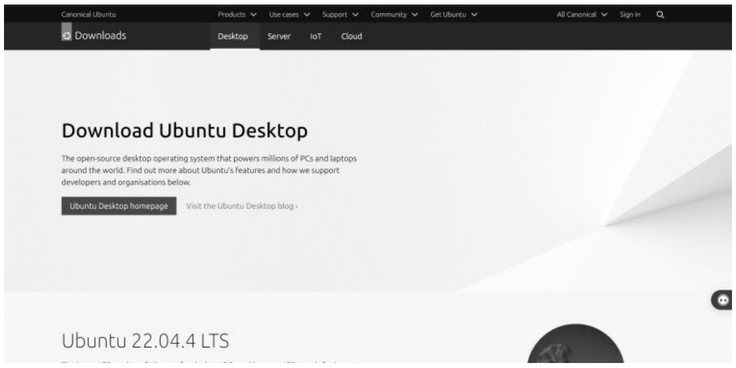
<u>Theory:</u>

A virtual machine (VM) is a software emulation of a computer system. It operates based on the architecture and functions of a real or physical computer. It allows you to run a guest operating system on top of your host operating system without making any changes to your primary operating system. This is particularly useful for testing, development, and running applications in a controlled environment separate from your main operating system.

Linux is an open-source operating system developed by Linus Torvalds in 1991. It is based on the Linux Kernel, which is the core of the operating system, managing how the computer interacts with its hardware and resources. The Linux Kernel is often referred to as the brain of the operating system because it ensures everything works smoothly and efficiently. To create a complete operating system, the Linux Kernel is combined with a collection of software packages and utilities, known as Linux distributions.

Steps to Install Linux in Virtual Machine:

Download a Linux Distribution: Choose a Linux distribution like Ubuntu, CentOS, or Debian. You can download the ISO image from their respective websites.

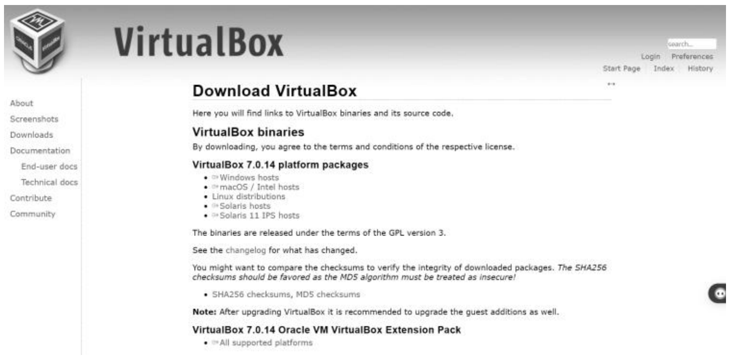


Download Ubuntu Desktop

The open-source desktop operating system that powers millions of PCs and laptops around the world. Find out more about Ubuntu's features and how we support developers and organisations below.

Ubuntu 22.04.4 LTS

Install a Virtual Machine Software: You can use software like VirtualBox, VMware, or Parallels Desktop to create a virtual machine on your computer.



Download VirtualBox

Here you will find links to VirtualBox binaries and its source code.

VirtualBox binaries

By downloading, you agree to the terms and conditions of the respective license.

VirtualBox 7.0.14 platform packages

- Windows hosts
- macOS / Intel hosts
- Linux distributions
- Solaris hosts
- Solaris 11 IPS hosts

The binaries are released under the terms of the GPL version 3.

See the changelog for what has changed.

You might want to compare the checksums to verify the integrity of downloaded packages. *The SHA256 checksums should be favored as the MD5 algorithm must be treated as insecure!*

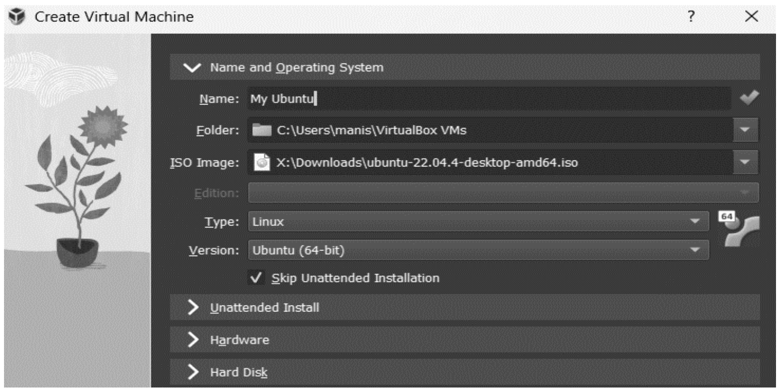
- SHA256 checksums, MD5 checksums

Note: After upgrading VirtualBox it is recommended to upgrade the guest additions as well.

VirtualBox 7.0.14 Oracle VM VirtualBox Extension Pack

- All supported platforms

Create a Virtual Machine: Open the virtual machine software and create a new virtual machine. During the setup process, you'll need to specify the Linux ISO image you downloaded as the installation media.


Create Virtual Machine

Name and Operating System

Name: My Ubuntu

Folder: C:\Users\manis\VirtualBox VMs

ISO Image: X:\Downloads\ubuntu-22.04.4-desktop-amd64.iso

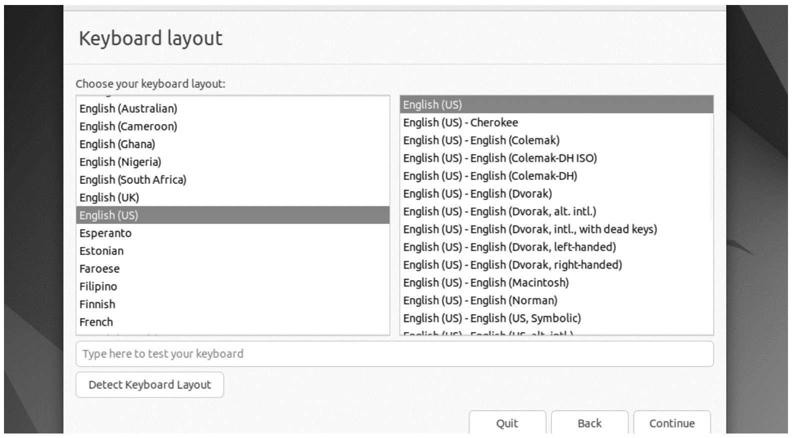
Edition:

Type: Linux

Version: Ubuntu (64-bit)

✓ Skip Unattended Installation

Install Linux: Start the virtual machine and follow the installation steps provided by the Linux distribution. This typically involves selecting language, keyboard layout, disk partitioning, creating a user account, etc.

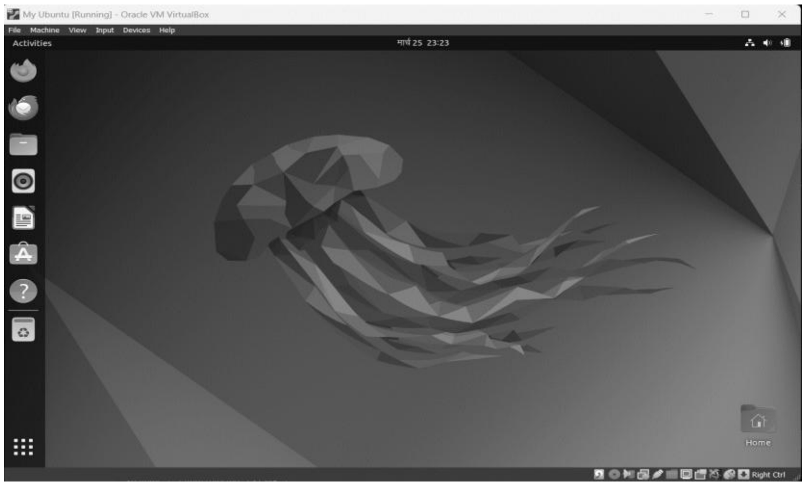

Keyboard layout

Choose your keyboard layout:

English (Australian)	English (US)
English (Cameroon)	English (US) - Cherokee
English (Ghana)	English (US) - English (Colemak)
English (Nigeria)	English (US) - English (Colemak-DH ISO)
English (South Africa)	English (US) - English (Colemak-DH)
English (UK)	English (US) - English (Dvorak)
English (US)	English (US) - English (Dvorak, alt. intl.)
Esperanto	English (US) - English (Dvorak, intl., with dead keys)
Estonian	English (US) - English (Dvorak, left-handed)
Faroese	English (US) - English (Dvorak, right-handed)
Filipino	English (US) - English (Macintosh)
Finnish	English (US) - English (Norman)
French	English (US) - English (US, Symbolic)

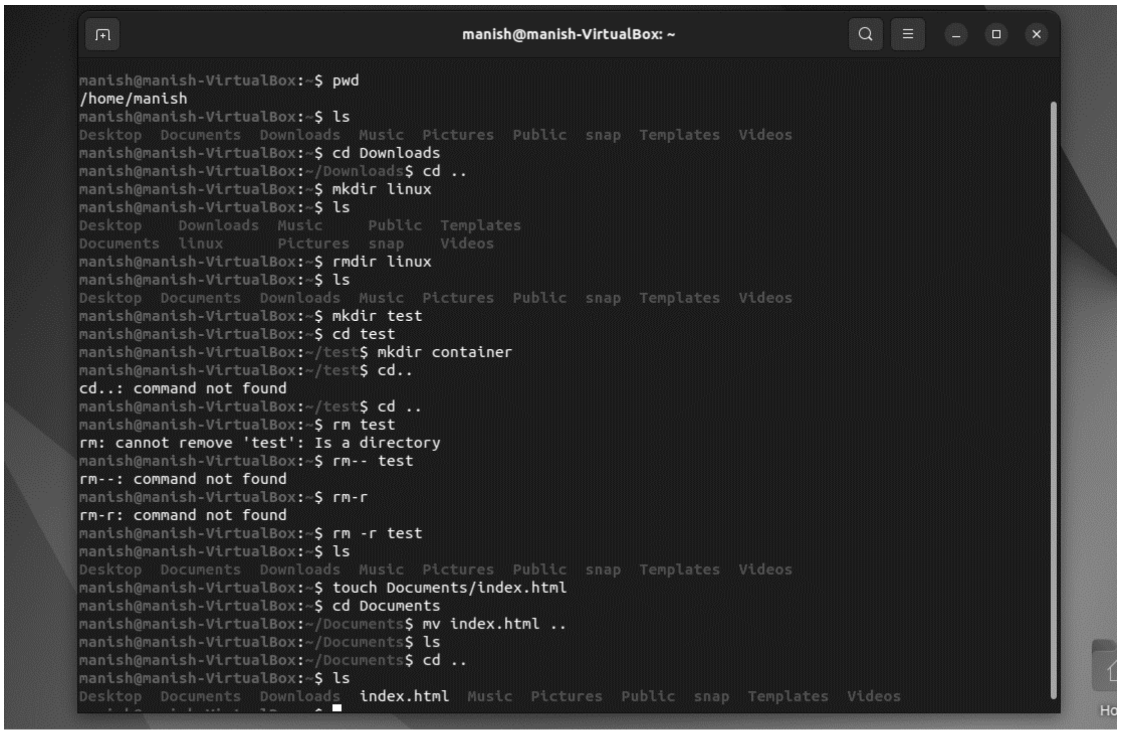
Type here to test your keyboard

Finish Installation: Once the installation is complete, you'll have a Linux system running inside your virtual machine.


My Ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
मार्च 25 23:23
Home
Right Ctrl

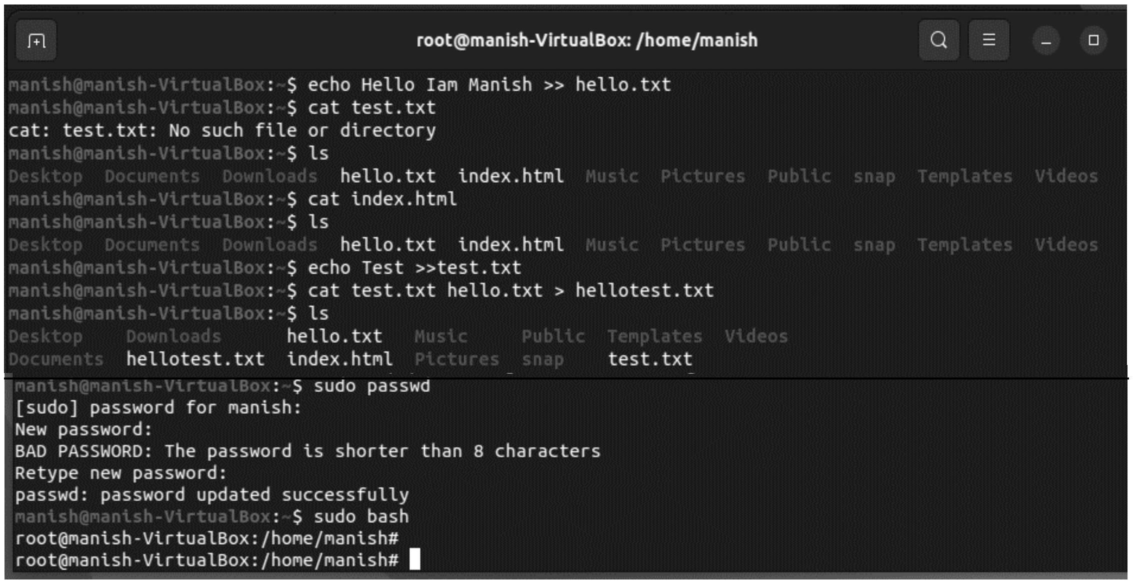
Running Linux Commands:

- Basic Commands:
  Commands Used: pwd, ls, cd, cd .., mkdir, rmdir, rm -r, touch and mv



```
manish@manish-VirtualBox:~$ pwd
/home/manish
manish@manish-VirtualBox:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
manish@manish-VirtualBox:~$ cd Downloads
manish@manish-VirtualBox:~/Downloads$ cd ..
manish@manish-VirtualBox:~$ mkdir linux
manish@manish-VirtualBox:~$ ls
Desktop    Downloads  Music     Public  Templates
Documents  linux      Pictures  snap    Videos
manish@manish-VirtualBox:~$ rmdir linux
manish@manish-VirtualBox:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
manish@manish-VirtualBox:~$ mkdir test
manish@manish-VirtualBox:~$ cd test
manish@manish-VirtualBox:~/test$ mkdir container
manish@manish-VirtualBox:~/test$ cd..
cd..: command not found
manish@manish-VirtualBox:~/test$ cd ..
manish@manish-VirtualBox:~$ rm test
rm: cannot remove 'test': Is a directory
manish@manish-VirtualBox:~$ rm-- test
rm--: command not found
manish@manish-VirtualBox:~$ rm-r
rm-r: command not found
manish@manish-VirtualBox:~$ rm -r test
manish@manish-VirtualBox:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
manish@manish-VirtualBox:~$ touch Documents/index.html
manish@manish-VirtualBox:~$ cd Documents
manish@manish-VirtualBox:~/Documents$ mv index.html ..
manish@manish-VirtualBox:~/Documents$ ls
manish@manish-VirtualBox:~/Documents$ cd ..
manish@manish-VirtualBox:~$ ls
Desktop  Documents  Downloads  index.html  Music  Pictures  Public  snap  Templates  Videos
```

- Intermediate Commands:
  Commands Used: echo, cat, sudo passwd, sudo bash



```
manish@manish-VirtualBox:~$ echo Hello Iam Manish >> hello.txt
manish@manish-VirtualBox:~$ cat test.txt
cat: test.txt: No such file or directory
manish@manish-VirtualBox:~$ ls
Desktop  Documents  Downloads  hello.txt  index.html  Music  Pictures  Public  snap  Templates  Videos
manish@manish-VirtualBox:~$ cat index.html
manish@manish-VirtualBox:~$ ls
Desktop  Documents  Downloads  hello.txt  index.html  Music  Pictures  Public  snap  Templates  Videos
manish@manish-VirtualBox:~$ echo Test >>test.txt
manish@manish-VirtualBox:~$ cat test.txt hello.txt > hellotest.txt
manish@manish-VirtualBox:~$ ls
Desktop    Downloads      hello.txt   Music     Public  Templates  Videos
Documents  hellotest.txt  index.html  Pictures  snap    test.txt
manish@manish-VirtualBox:~$ sudo passwd
[sudo] password for manish:
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: password updated successfully
manish@manish-VirtualBox:~$ sudo bash
root@manish-VirtualBox:/home/manish#
root@manish-VirtualBox:/home/manish#
```

Conclusion:

By this way Linux (Ubuntu) was installed in a virtual machine and tested with some basic and intermediate Linux Commands.

Title: Running C program and Process Creation in Linux

Objectives:

- To Run C program in Linux
- To Create Process in Linux using fork()

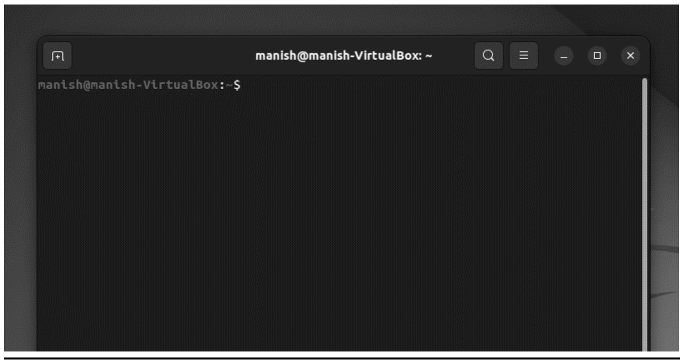
Theory:

Writing, compiling, and running a C program in Linux involves a straightforward process that can be executed through the terminal or using a code editor like Visual Studio Code.

The fork() system call is a fundamental function in Unix and Linux systems for creating a new process. When a process calls fork(), it creates a new process by duplicating the existing process. The new process, called the child, is an exact copy of the calling process, known as the parent, except for a few values that get changed, such as the process ID.

Steps for Running C Program in Linux:

1. Open the Terminal from the applications section



```
manish@manish-VirtualBox:~$
```

2. Use a text editor to create the C source code

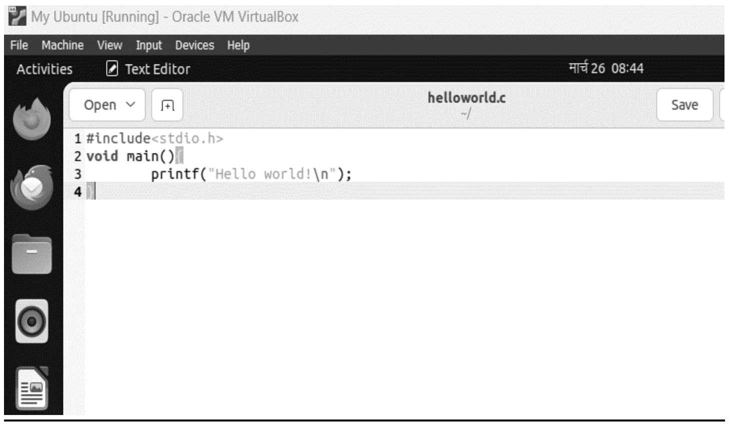
Type the command

```
gedit helloworld.c
```

and enter the C source code below:

```
#include <stdio.h>
void main() {
        printf("Hello World\n");
}
```

Save the code and close the editor

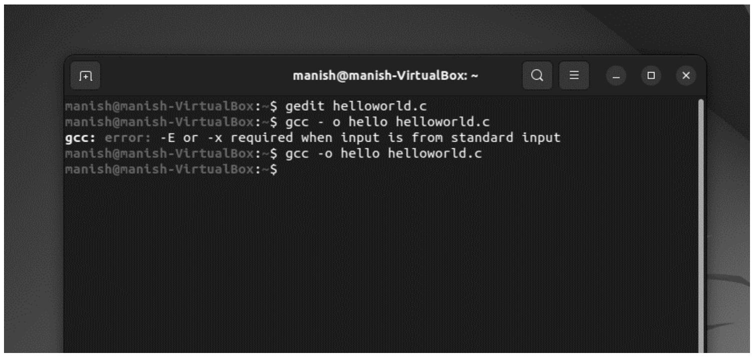

My Ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Text Editor
मार्च 26 08:44
Open
helloworld.c
~/
Save

```
#include<stdio.h>
void main(){
        printf("Hello world!\n");
}
```

3. Compile the Source Code:
   Type the command

   gcc -o hello helloworld.c

   This command will invoke the GNU C compiler to compile the file helloworld.c and output (-o) the result to an executable called hello



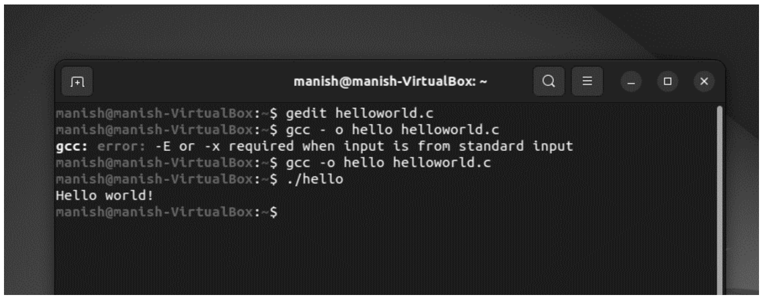
```
manish@manish-VirtualBox:~$ gedit helloworld.c
manish@manish-VirtualBox:~$ gcc - o hello helloworld.c
gcc: error: -E or -x required when input is from standard input
manish@manish-VirtualBox:~$ gcc -o hello helloworld.c
manish@manish-VirtualBox:~$
```

4. Execute the program:
   Type the command

   ./hello

   This should result in output:

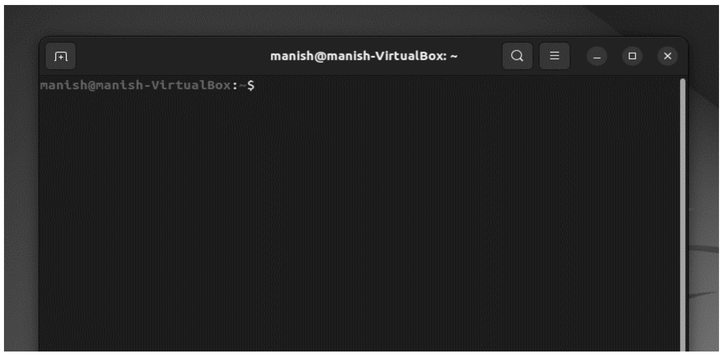
   Hello World



```
manish@manish-VirtualBox:~$ gedit helloworld.c
manish@manish-VirtualBox:~$ gcc - o hello helloworld.c
gcc: error: -E or -x required when input is from standard input
manish@manish-VirtualBox:~$ gcc -o hello helloworld.c
manish@manish-VirtualBox:~$ ./hello
Hello world!
manish@manish-VirtualBox:~$
```

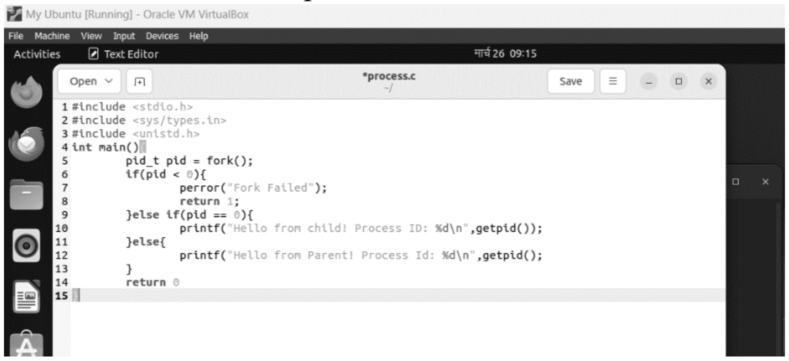
Steps for Running creating process in Linux:

1. Open the Terminal from the applications section



```
manish@manish-VirtualBox:~$
```

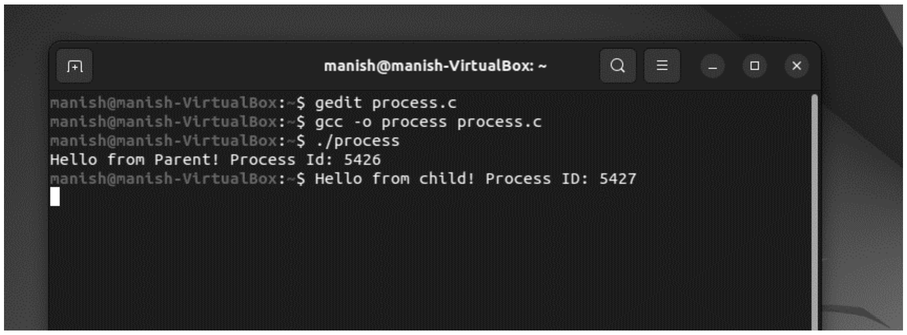
2. Use a text editor to create a new process file in c



My Ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Text Editor
मार्च 26 09:15
Open
*process.c
Save

```c
#include <stdio.h>
#include <sys/types.in>
#include <unistd.h>
int main()
        pid_t pid = fork();
        if(pid < 0){
                perror("Fork Failed");
                return 1;
        }else if(pid == 0){
                printf("Hello from child! Process ID: %d\n",getpid());
        }else{
                printf("Hello from Parent! Process Id: %d\n",getpid();
        }
        return 0

```

3. Compile and Execute the Source Code



```
manish@manish-VirtualBox:~$ gedit process.c
manish@manish-VirtualBox:~$ gcc -o process process.c
manish@manish-VirtualBox:~$ ./process
Hello from Parent! Process Id: 5426
manish@manish-VirtualBox:~$ Hello from child! Process ID: 5427
```

Conclusion:

By this way C program was run and a process was created using Fork() in Linux..

Title: Simulation of Producer-Consumer Problem using Semaphore.

Objectives:

- To Simulate Producer-Consumer problem using Semaphore

Theory:

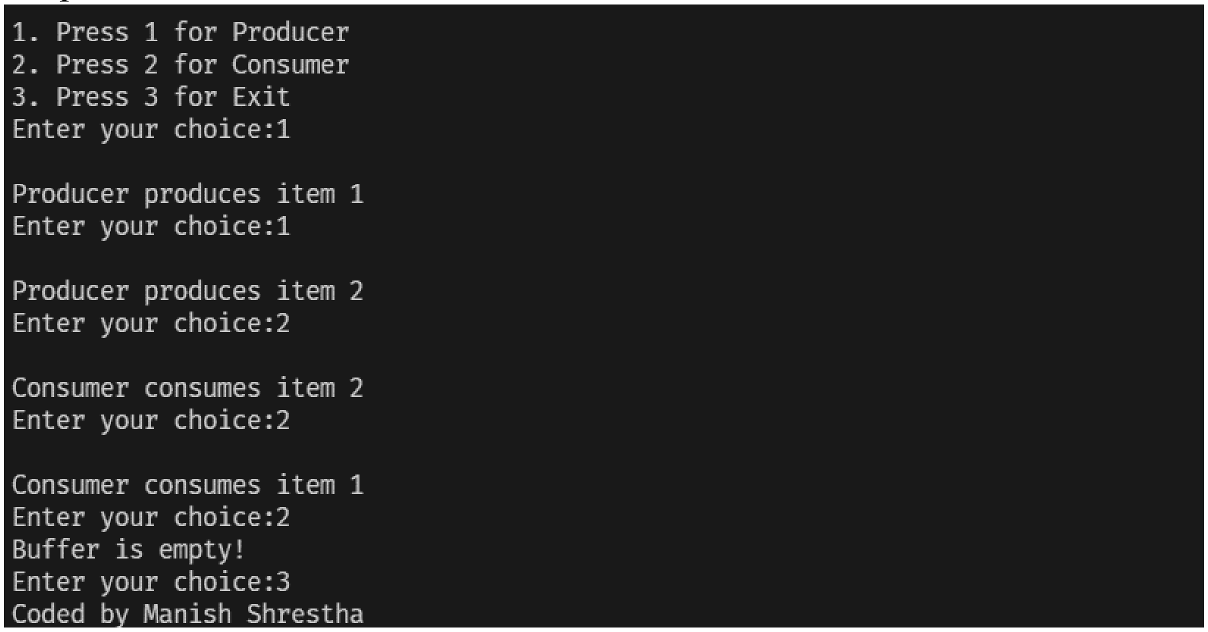
The Producer-Consumer Problem is a classic example of a multi-process synchronization problem in concurrent programming. It involves two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from the buffer. The challenge arises when the buffer is full and the producer tries to add more data, or when the buffer is empty and the consumer tries to consume data. This situation can lead to deadlock if not properly managed.

Program 1: Simulation of Producer-Consumer using Semaphore in C

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

int mutex = 1,full = 0, empty = 10, x = 0;
void producer(){
  --mutex;
  ++full;
  --empty;
  x++;
  printf("\nProducer produces item %d",x);
  ++mutex;
}
void consumer(){
  --mutex;
  --full;
  ++empty;
  printf("\nConsumer consumes item %d",x);
  x--;
  ++mutex;
}
void main(){
  int n, i;
  printf("1. Press 1 for Producer\n2. Press 2 for Consumer\n3. Press 3 for Exit");
  for (i = 1; i > 0; i++)
  {
    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:
               printf("Coded by Manish Shrestha");
               exit(0);
               break;
            }
         }
      }
```

Output 1:



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

Producer produces item 1
Enter your choice:1

Producer produces item 2
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:3
Coded by Manish Shrestha
```

Conclusion:

By this way the Producer-Consumer Problem was simulated using Semaphore in C.

.

Title: Simulation of CPU scheduling algorithms to find average turnaround time and average waiting time (I)

Objectives:

- To Simulate First Come First Serve (FCFS) scheduling algorithm in C
- To Simulate Shortest Job First (SJF) scheduling algorithm in C

Theory:

The First Come First Serve (FCFS) scheduling algorithm is a non-preemptive CPU scheduling method that operates on the principle of executing processes in the order they arrive. This means the process that requests the CPU first is allocated the CPU first. It is implemented using a FIFO (First In, First Out) queue, where each process's Process Control Block (PCB) is linked to the tail of the queue. When the CPU becomes free, it is allocated to the process at the head of the queue, and the running process is then removed from the queue.

The Shortest Job First (SJF) scheduling algorithm, also known as Shortest Job Next (SJN), is a CPU scheduling method that selects the process with the smallest execution time to execute next. It can be either preemptive or non-preemptive. SJF is considered a greedy algorithm due to its approach of always choosing the process with the shortest burst time available at any given time. This algorithm aims to minimize the average waiting time among all scheduling algorithms

Algorithm for FCFS Scheduling:

```
Step 1: Start the Process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process name and the burst time
Step 4: Set the waiting of the first process as 0 and its burst time as its turnaround time
Step 5: For each process in the Ready Q calculate
    a. Waiting time (n) = waiting time (n-1) + Burst time (n-1)
    b. Turnaround time (n) = waiting time (n) + Burst time (n)
Step 6: Calculate
    a. Average waiting time = Total waiting Time / Number of process
    b. Average Turnaround time = Total Turnaround Time/Number of process
Step 7: Stop the process
```

Algorithm for SJF Scheduling:

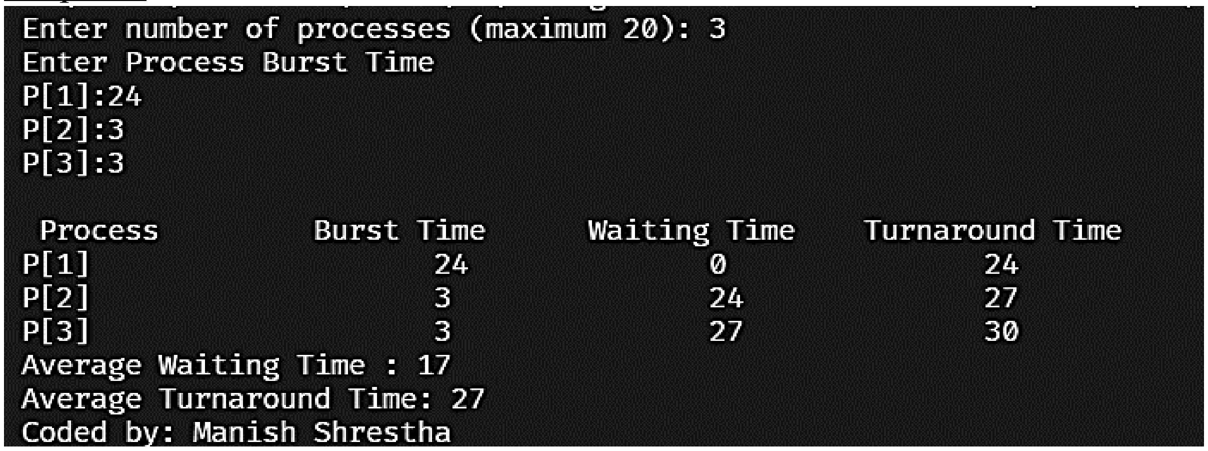
```
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according to the shortest Burst time by sorting according to lowest to highest burst time.
```

Step 5: Set the waiting time of the first process as 0 and its turnaround time as its burst time
Step 6: Sort the processes names based on their Burst time
Step 7: For each process in the ready queue, calculate
  a. Waiting time (n) = waiting time (n-1) + Burst time(n-1)
  b. Turnaround time (n) = waiting time(n) + Burst time(n)
Step 8: Calculate
  a. Average waiting time = Total waiting time / Number of process
  b. Average Turnaround time = Total Turnaround time / Number of process
Step 9: Stop the process

<u>Program 1</u>: Simulation of FCFS in C

```
#include <stdio.h>
void main()
{
   int n, bt[20], wt[20], tat[20], avwt = 0, avtat = 0, i, j;
   printf("Enter number of processes (maximum 20): ");
   scanf("%d", &n);
   printf("\n Enter Process Burst Time\n");
   for(i=0; i<n; i++){
      printf("P[%d]:",i+1);
      scanf("%d",&bt[i]);
   }
   wt[0] = 0;
   for(i=1;i<n;i++){
      wt[i]=0;
      for(j=0;j<i;j++){
         wt[i] += bt[j];
      }
   }
   printf("\n Process \t Burst Time \t Waiting Time\t Turnaround Time");
   for(i=0;i<n;i++){
      tat[i] = bt[i] + wt[i];
      avwt += wt[i];
      avtat += tat[i];
      printf("\nP[%d]\t\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
   }
   avwt/=i;
   avtat/=i;
   printf("\nAverage Waiting Time : %d",avwt);
   printf("\nAverage Turnaround Time: %d",avtat);
   printf("\nCoded by: Manish Shrestha");
}
```

Output 1:



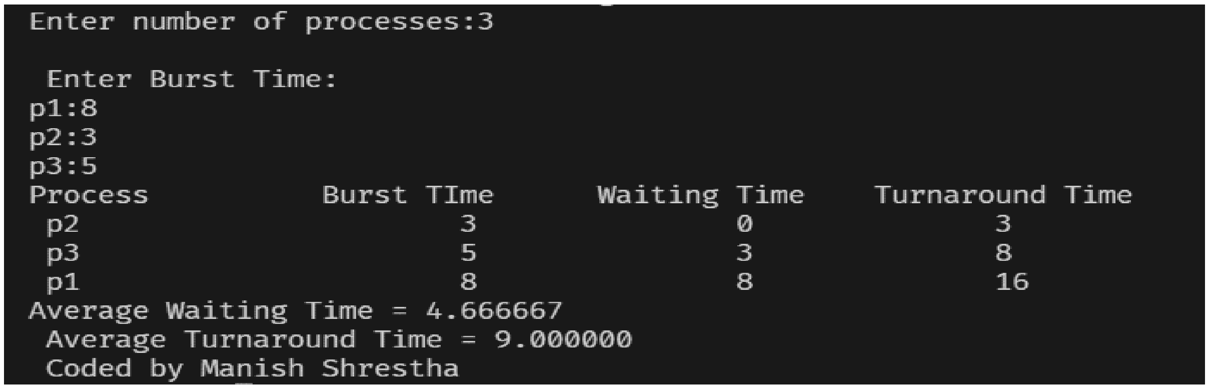
```
Enter number of processes (maximum 20): 3
Enter Process Burst Time
P[1]:24
P[2]:3
P[3]:3

 Process         Burst Time      Waiting Time    Turnaround Time
P[1]                    24              0               24
P[2]                    3               24              27
P[3]                    3               27              30
Average Waiting Time : 17
Average Turnaround Time: 27
Coded by: Manish Shrestha
```

Program 2: Simulation of SJF in C

```
#include <stdio.h>
void main()
{
   int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, pos, temp;
   float avg_wt, avg_tat;
   printf("Enter number of processes:");
   scanf("%d",&n);
   printf("\n Enter Burst Time:\n");
   for(i=0; i<n; ++i){
      printf("p%d:",i+1);
      scanf("%d", &bt[i]);
      p[i] = i+1;
   }
   for(i=0;i<n;i++){
      pos = i;
      for(j=i+1;j<n;j++){
         if(bt[j]<bt[pos]) pos = j;
      }
      temp = bt[i];
      bt[i] = bt[pos];
      bt[pos] = temp;
      temp = p[i];
      p[i] = p[pos];
      p[pos] = temp;
   }
   wt[0] = 0;
```

```
    for(i=1;i<n;i++){
        wt[i] = 0;
        for(j=0;j<i;j++){
            wt[i] += bt[j];
            total += wt[i];
        }
    }
    avg_wt = (float)total/n;
    total = 0;
    printf("Process \t Burst TIme\t Waiting Time \t Turnaround Time");
    for(i=0;i<n;i++){
        tat[i] = bt[i] + wt[i];
        total += tat[i];
        printf("\n p%d\t\t %d\t\t %d\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat = (float)total/n;
    printf("\nAverage Waiting Time = %f",avg_wt);
    printf("\n Average Turnaround Time = %f",avg_tat);
    printf("\nCoded by: Manish Shrestha");
}
```

Output :



```
Enter number of processes:3

 Enter Burst Time:
p1:8
p2:3
p3:5
Process          Burst TIme      Waiting Time    Turnaround Time
 p2                      3               0               3
 p3                      5               3               8
 p1                      8               8               16
Average Waiting Time = 4.666667
 Average Turnaround Time = 9.000000
 Coded by Manish Shrestha
```

Conclusion:

By this way the FCFS and SJF scheduling Algorithms were simulated in C to calculate average waiting time and average turnaround time of different processes.

Title: Simulation of CPU scheduling algorithms to find average turnaround time and average waiting time (II)

Objectives:

- To simulate Priority Scheduling Algorithm in C
- To simulate Round Robin Scheduling Algorithm in C

Theory:

Priority Scheduling is a method used by operating systems to manage the execution of processes based on their assigned priority levels. In this algorithm, each process is assigned a priority number, and the process with the highest priority (usually the lowest priority number) is executed first. If two processes have the same priority, they are executed on a first-come, first-served basis. This scheduling method is particularly useful in batch processing systems where certain tasks need to be executed before others due to their importance or resource requirements.

The Round Robin Scheduling Algorithm is a preemptive CPU scheduling algorithm that operates on a cyclical basis, assigning each process a fixed time slot, known as a time quantum, for execution. This algorithm is designed to ensure fairness by providing equal CPU time to each process in the ready queue, preventing starvation and promoting efficient use of the CPU.

Algorithm for Priority Scheduling:

```
Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according to the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as 0 and its turnaround time as its burst time
Step 6: Sort the processes names based on their process priority
Step 7: For each process in the ready queue, calculate
    a. Waiting time (n) = waiting time (n-1) + Burst time(n-1)
    b. Turnaround time (n) = waiting time(n) + Burst time(n)
Step 8: Calculate
    a. Average waiting time = Total waiting time / Number of process
    b. Average Turnaround time = Total Turnaround time / Number of process
Step 9: Stop the process
```

Algorithm for Round Robin Scheduling:

```
Step 1: Start the process
Step 2: Accept the number processes in the ready Queue and time quantum (or) time slice.
```

Step 3: For each process in the ready Q assign the process id and accept the CPU burst time.
Step 4: Calculate the no. of time slice for each process where No. of time slice for process (n) = burst time process (n) / time slice
Step 5: If the burst time is less than the time slice then the no. of time slice = 1.
Step 6: Consider the ready queue is a circular Q, calculate

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

Step 7: Calculate:

a. Average waiting time = Total waiting Time / Number of Process
b. Average Turnaround time = Total Turnaround time / Number of process

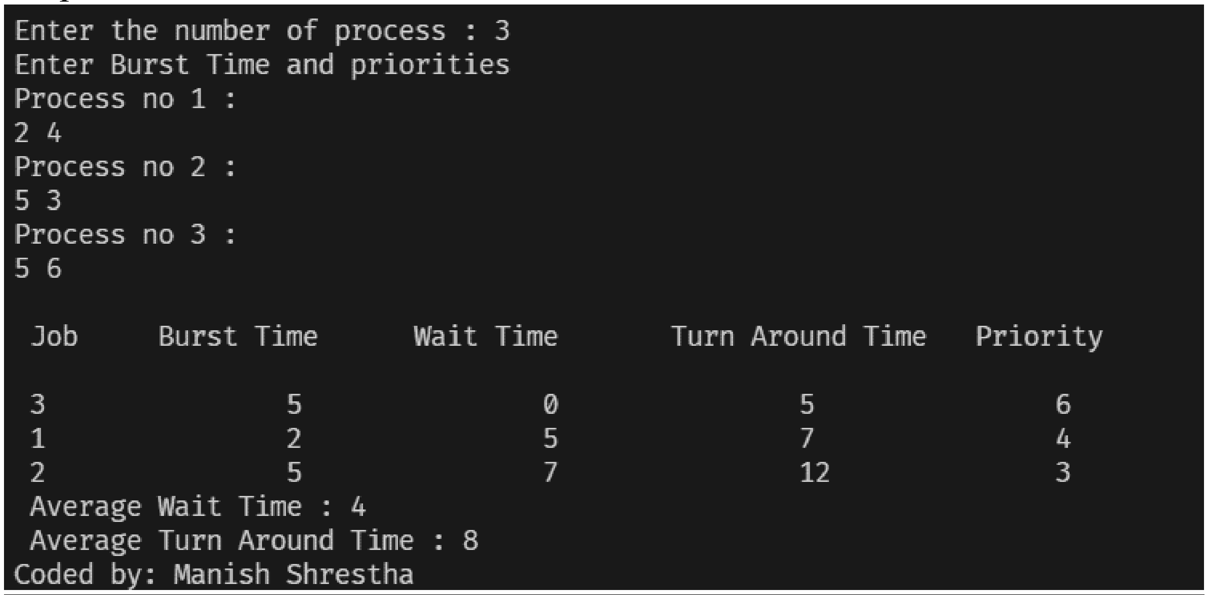
Step 8: Stop the process

Program 1: Simulation of Priority Scheduling in C

```
#include <stdio.h>
void main()
{
  int x, n, p[10], pp[10], pt[10], w[10], t[10], awt, atat, i;
  printf("Enter the number of process : ");
  scanf("%d", &n);
  printf("\n Enter Burst Time and priorities \n");
  for (i = 0; i < n; i++)
  {
    printf("Process no %d : \n", i + 1);
    scanf("%d  %d", &pt[i], &pp[i]);
    p[i] = i + 1;
  }
  for (i = 0; i < n - 1; i++)
  {
    for (int j = i + 1; j < n; j++)
    {
      if (pp[i] < pp[j])
      {
        x = pp[i];
        pp[i] = pp[j];
        pp[j] = x;
        x = pt[i];
        pt[i] = pt[j];
        pt[j] = x;
        x = p[i];
```

```
                p[i] = p[j];
                p[j] = x;
            }
        }
    }
    w[0] = 0;
    awt = 0;
    t[0] = pt[0];
    atat = t[0];
    for (i = 1; i < n; i++)
    {
        w[i] = t[i - 1];
        awt += w[i];
        t[i] = w[i] + pt[i];
        atat += t[i];
    }
    printf("\n Job \t Burst Time \t Wait Time \t Turn Around Time   Priority \n");
    for (i = 0; i < n; i++)
        printf("\n %d \t\t %d \t\t %d \t\t %d \t\t %d", p[i], pt[i], w[i], t[i], pp[i]);
    awt /= n;
    atat /= n;
    printf("\n Average Wait Time : %d", awt);
    printf("\n Average Turn Around Time : %d", atat);
    printf("\nCoded by: Manish Shrestha");
}
```

Output:



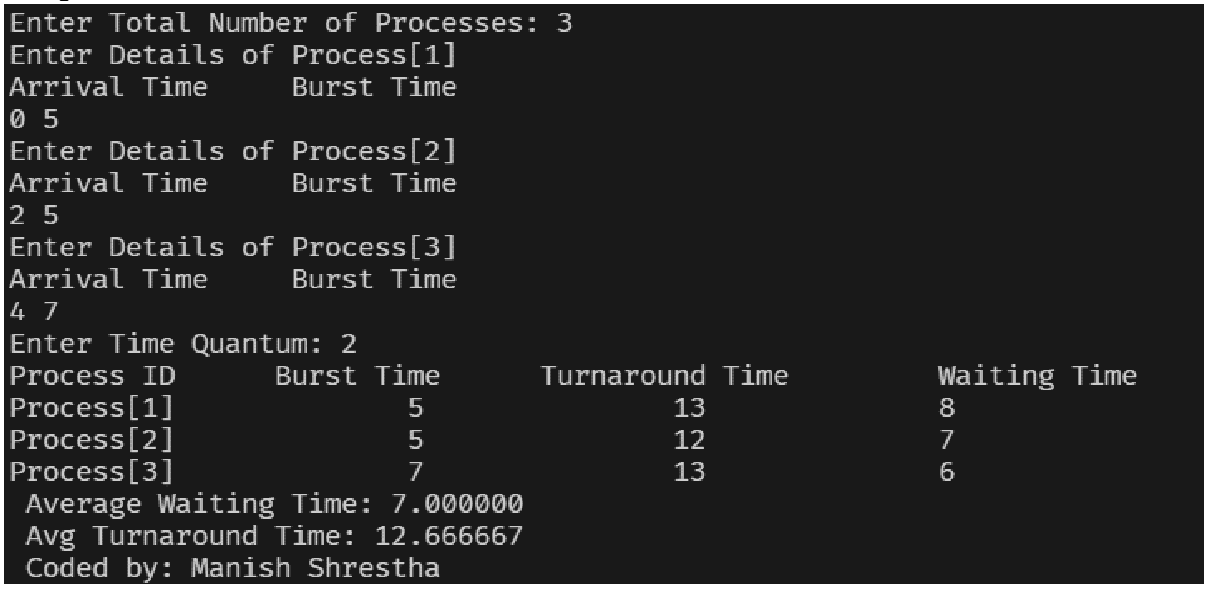
```
Enter the number of process : 3
Enter Burst Time and priorities
Process no 1 :
2 4
Process no 2 :
5 3
Process no 3 :
5 6

 Job     Burst Time      Wait Time       Turn Around Time   Priority

 3               5               0               5               6
 1               2               5               7               4
 2               5               7               12              3
 Average Wait Time : 4
 Average Turn Around Time : 8
Coded by: Manish Shrestha
```

Program 2: Simulation of Round Robin Scheduling in C

```
#include<stdio.h>
int main(){
    int i, limit, total = 0, x, counter = 0, time_quantum;
    int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
    float average_wait_time, average_turnaround_time;
    printf("Enter Total Number of Processes: ");
    scanf("%d", &limit);
    x = limit;
    for(i = 0; i < limit; i++)
    {
        printf("Enter Details of Process[%d]\n", i + 1);
        printf("Arrival Time\t Burst Time\n");
        scanf("%d %d", &arrival_time[i],&burst_time[i]);
        temp[i] = burst_time[i];
    }
    printf("Enter Time Quantum: ");
    scanf("%d", &time_quantum);
    printf("Process ID \tBurst Time \tTurnaround Time \tWaiting Time");
    for(total = 0, i = 0; x != 0;){
        if(temp[i] <= time_quantum && temp[i] > 0){
            total = total + temp[i];
            temp[i] = 0;
            counter = 1;
        }
        else if(temp[i] > 0){
            temp[i] = temp[i] - time_quantum;
            total = total + time_quantum;
        }
        if(temp[i] == 0 && counter == 1){
            x--;
            printf("\nProcess[%d]\t\t%d \t\t%d \t\t%d", i + 1, burst_time[i], total -
arrival_time[i], total - arrival_time[i] - burst_time[i]);
            wait_time = wait_time + total - arrival_time[i] - burst_time[i];
            turnaround_time = turnaround_time + total - arrival_time[i];
            counter = 0;
        }
        if(i == limit - 1) i = 0;
        else if(arrival_time[i + 1] <= total) i++;
        else i = 0;
    }
    average_wait_time = wait_time * 1.0 / limit;
    average_turnaround_time = turnaround_time * 1.0 / limit;
    printf("\n Average Waiting Time:t%f", average_wait_time);
```

```
        printf("\n Avg Turnaround Time:t%f", average_turnaround_time);
        printf("\n Coded by: Manish Shrestha");
    }
```

Output:



```
Enter Total Number of Processes: 3
Enter Details of Process[1]
Arrival Time     Burst Time
0 5
Enter Details of Process[2]
Arrival Time     Burst Time
2 5
Enter Details of Process[3]
Arrival Time     Burst Time
4 7
Enter Time Quantum: 2
Process ID      Burst Time      Turnaround Time         Waiting Time
Process[1]             5               13               8
Process[2]             5               12               7
Process[3]             7               13               6
 Average Waiting Time: 7.000000
 Avg Turnaround Time: 12.666667
 Coded by: Manish Shrestha
```

Conclusion:

By this way the Priority Scheduling and Round Robin scheduling Algorithms were simulated in C to calculate average waiting time and average turnaround time of different processes.

Title: Simulation of Contiguous Memory Allocation Algorithms.

Objectives:

- To simulate The First Fit Algorithm in C
- To simulate The Best Fit Algorithm in C
- To simulate The Worst Fit Algorithm in C

Theory:

The First Fit memory allocation technique is a method used by operating systems to allocate memory to processes. It operates by searching through the list of free memory blocks, starting from the beginning, until it finds a block that is large enough to accommodate the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

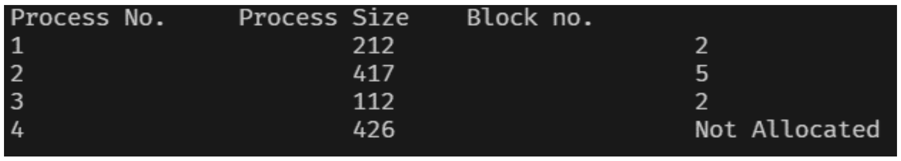
The Best Fit memory allocation technique is a method used by operating systems to allocate memory to processes. It operates by searching through the list of free memory blocks to find the smallest block that is large enough to accommodate the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

The Worst Fit memory allocation technique is a method used by operating systems to allocate memory to processes. It operates by searching through the list of free memory blocks to find the largest block that is large enough to accommodate the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

Program 1: Simulation of First Fit Algorithm in C

```
#include <stdio.h>
#include <stdlib.h>
void firstFit(int blockSize[], int m, int processSize[], int n) {
  int i, j;
  int* allocation = (int*)malloc(n * sizeof(int));
  if (allocation == NULL) {
     printf("Memory allocation failed.\n");
     return;
  }
 for (i = 0; i < n; i++) {
    allocation[i] = -1;
 }
  for (i = 0; i < n; i++) {
     for (j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (i = 0; i < n; i++) {
        printf("%i\t\t\t%i\t\t\t", i+1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%i", allocation[i] + 1);
        } else {
            printf("Not Allocated");
        }
        printf("\n");
    }
}
int main() {
    int m;
    int n;
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    m = sizeof(blockSize) / sizeof(blockSize[0]);
    n = sizeof(processSize) / sizeof(processSize[0]);
    firstFit(blockSize, m, processSize, n);
    return 0;
}
```

Output:

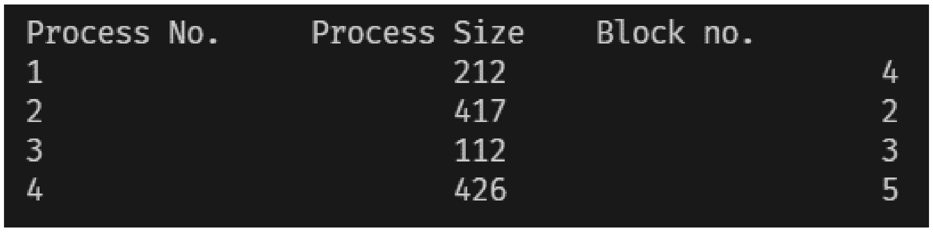


```
Process No.     Process Size    Block no.
1                       212                     2
2                       417                     5
3                       112                     2
4                       426                     Not Allocated
```

Program 2: Simulation of Best Fit Algorithm in C

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

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int i, j;
    int* allocation = (int*)malloc(n * sizeof(int));
    if (allocation == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

   for (i = 0; i < n; i++) {
       allocation[i] = -1;
   }

    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1) {
                    bestIdx = j;
                } else if (blockSize[bestIdx] > blockSize[j]) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t\t%d\t\t\t", i+1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d", allocation[i] + 1);
        } else {
            printf("Not Allocated");
        }
        printf("\n");
    }
}

void main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);
    bestFit(blockSize, m, processSize, n);
}
```

Output:



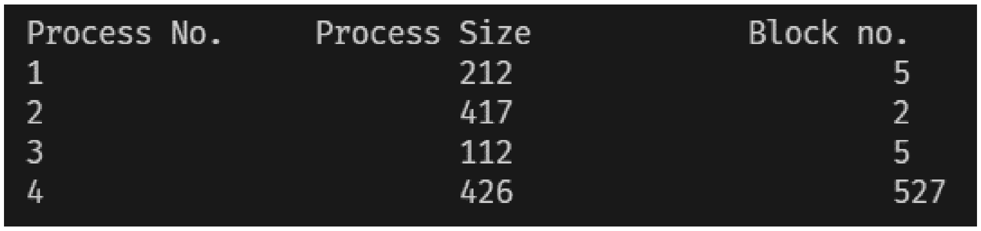
```
Process No.     Process Size    Block no.
1                       212                     4
2                       417                     2
3                       112                     3
4                       426                     5
```

Program 3: Simulation of Worst Fit Algorithm in C

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

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int* allocation = (int*)malloc(n * sizeof(int));
    memset(allocation, -1, sizeof(allocation));
    for (int i = 0; i < n; i++) {
        int wstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (wstIdx == -1) {
                    wstIdx = j;
                } else if (blockSize[wstIdx] < blockSize[j]) {
                    wstIdx = j;
                }
            }
        }
        if (wstIdx != -1) {
            allocation[i] = wstIdx;
            blockSize[wstIdx] -= processSize[i];
        }
    }
    printf("\nProcess No.\tProcess Size\t\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t\t%d\t\t\t", i+1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d", allocation[i] + 1);
        } else {
            printf("Not Allocated");
        }
        printf("\n");
    }
}
```

```
void main() {
   int blockSize[] = {100, 500, 200, 300, 600};
   int processSize[] = {212, 417, 112, 426};
   int m = sizeof(blockSize) / sizeof(blockSize[0]);
   int n = sizeof(processSize) / sizeof(processSize[0]);
   worstFit(blockSize, m, processSize, n);
}
```

Output:



```
Process No.     Process Size           Block no.
1                    212                    5
2                    417                    2
3                    112                    5
4                    426                    527
```

Conclusion:

By this way the First Fit, Best Fit and Worst Fit Algorithms were simulated in C to check the contiguous memory allocation in the system.

Title: Simulation of Page Replacement Algorithms.

Objectives:

- To simulate The FIFO Page Replacement Algorithm in C
- To simulate The LRU Page Replacement Algorithm in C
- To simulate The OPR Page Replacement Algorithm in C

Theory:

The FIFO (First In First Out) page replacement algorithm is a simple and straightforward method used in operating systems for managing memory. It operates on the principle that the oldest page in memory should be replaced when a page fault occurs and there is no free space left in the memory. This algorithm is based on the queue data structure, where the first page that enters the memory is the first one to be replaced when a new page needs to be loaded into memory.

The Least Recently Used (LRU) page replacement algorithm is a widely used method in operating systems for managing memory. It operates on the principle that the page that has not been used for the longest time should be replaced when a page fault occurs and there is no free space left in the memory. This algorithm is based on the principle of locality of reference, which suggests that pages that have been recently used are likely to be used again in the near future.

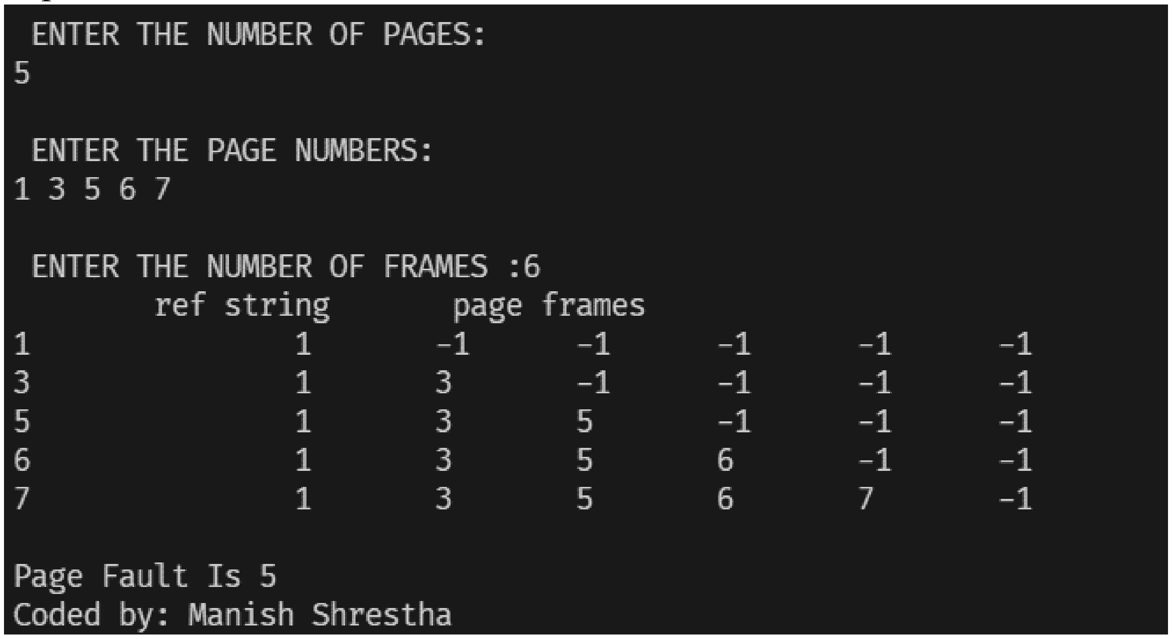
The Optimal Page Replacement (OPR) algorithm is a page replacement algorithm used in operating systems for managing memory. It aims to minimize the number of page faults by replacing the page that will not be used for the longest period of time in the future. This algorithm is based on the principle of future use, predicting which pages will not be needed in the near future and replacing them when a page fault occurs.

Program 1: Simulation of FIFO Page Replacement Algorithm in C

```
#include <stdio.h>

int main() {
   int i, j, n, a[50], frame[10], no, k, avail, count = 0;
   printf("\n ENTER THE NUMBER OF PAGES:\n");
   scanf("%d", &n);
   printf("\n ENTER THE PAGE NUMBERS:\n");
   for(i = 0; i < n; i++) {
      scanf("%d", &a[i]);
   }
   printf("\n ENTER THE NUMBER OF FRAMES :");
   scanf("%d", &no);
   for(i = 0; i < no; i++) {
      frame[i] = -1;
```

```
        }
        j = 0;
        printf("\tref string\t page frames\n");
        for(k = 0; k < n; k++) {
          avail = 0;
          for(i = 0; i < no; i++) {
            if(frame[i] == a[k]) {
              avail = 1;
              break;
            }
          }
          if(avail == 0) {
            frame[j] = a[k];
            j = (j + 1) % no;
            count++;
          }
          printf("%d\t\t", a[k]);
          for(i = 0; i < no; i++) {
            printf("%d\t", frame[i]);
          }
          printf("\n");
        }
        printf("\nPage Fault Is %d", count);
        printf("\nCoded by: Manish Shrestha");
        return 0;
    }
```

Output:



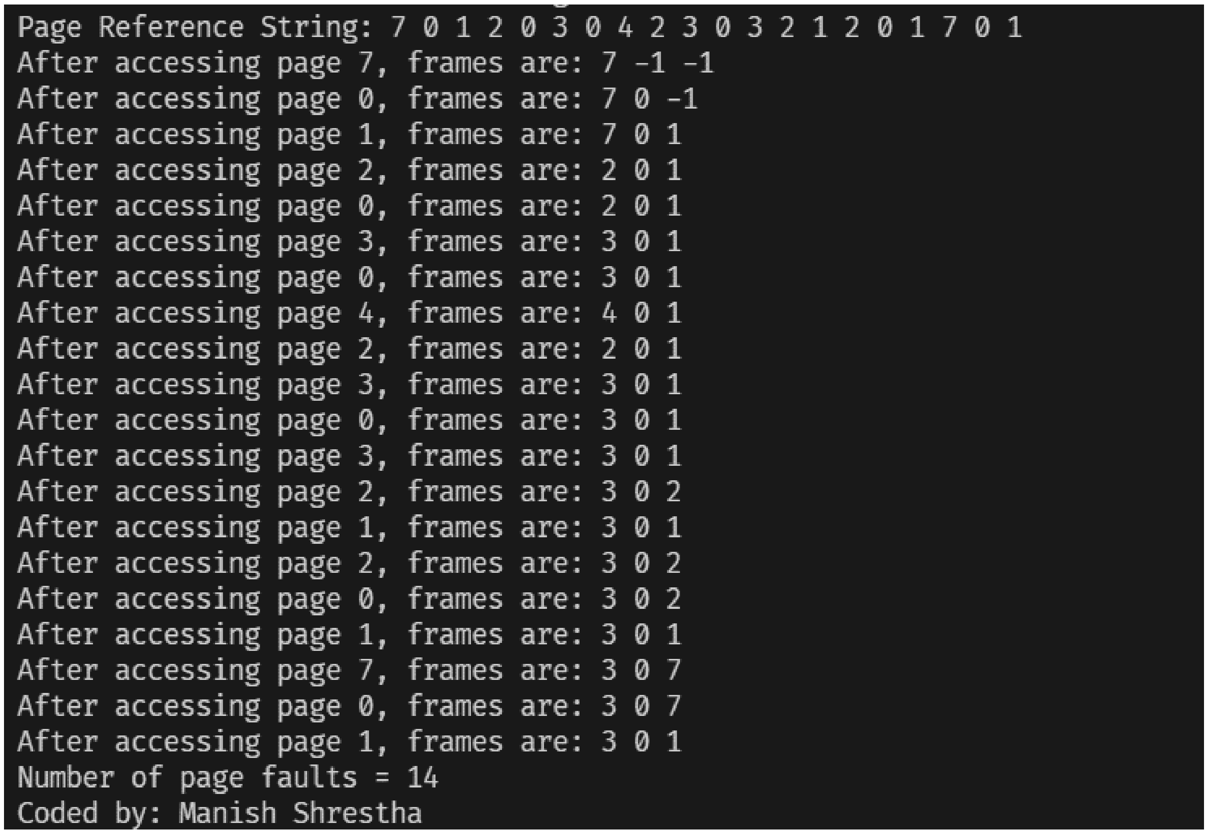
```
 ENTER THE NUMBER OF PAGES:
5

 ENTER THE PAGE NUMBERS:
1 3 5 6 7

 ENTER THE NUMBER OF FRAMES :6
        ref string       page frames
1               1       -1      -1      -1      -1      -1
3               1       3       -1      -1      -1      -1
5               1       3       5       -1      -1      -1
6               1       3       5       6       -1      -1
7               1       3       5       6       7       -1

Page Fault Is 5
Coded by: Manish Shrestha
```

Program 1: Simulation of LRU Page Replacement Algorithm in C

```
#include <stdio.h>
#include <stdlib.h>unction to find the LRU page
int findLRU(int time[], int n) {
   int i, minimum = time[0], pos = 0;
   for(i = 1; i < n; ++i) {
      if(time[i] < minimum) {
         minimum = time[i];
         pos = i;
      }
   }
   return pos;
}
void simulateLRU(int pages[], int n, int capacity) {
   int *frame = (int *)malloc(capacity * sizeof(int));
   int *time = (int *)malloc(capacity * sizeof(int));
   int frame_count = 0, page_fault = 0, i, j, flag;
   for(i = 0; i < capacity; ++i) {
      frame[i] = -1;
      time[i] = 0;
   }
   printf("Page Reference String: ");
   for(i = 0; i < n; ++i) {
      printf("%d ", pages[i]);
   }
   printf("\n");
   for(i = 0; i < n; ++i) {
      flag = 0;
      for(j = 0; j < capacity; ++j) {
         if(frame[j] == pages[i]) {
            flag = 1;
            time[j] = i + 1;
            break;
         }
      }
      if(flag == 0) {
         if(frame_count < capacity) {
            frame[frame_count++] = pages[i];
         } else {
            int pos = findLRU(time, capacity);
            frame[pos] = pages[i];
         }
         page_fault++;
      }
```

```
        printf("After accessing page %d, frames are: ", pages[i]);
        for(j = 0; j < capacity; ++j) {
            printf("%d ", frame[j]);
        }
        printf("\n");
    }
    printf("Number of page faults = %d\n", page_fault);
    printf("Coded by: Manish Shrestha");
    free(frame);
    free(time);
}
int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 3;
    simulateLRU(pages, n, capacity);
    return 0;
}
```

Output:

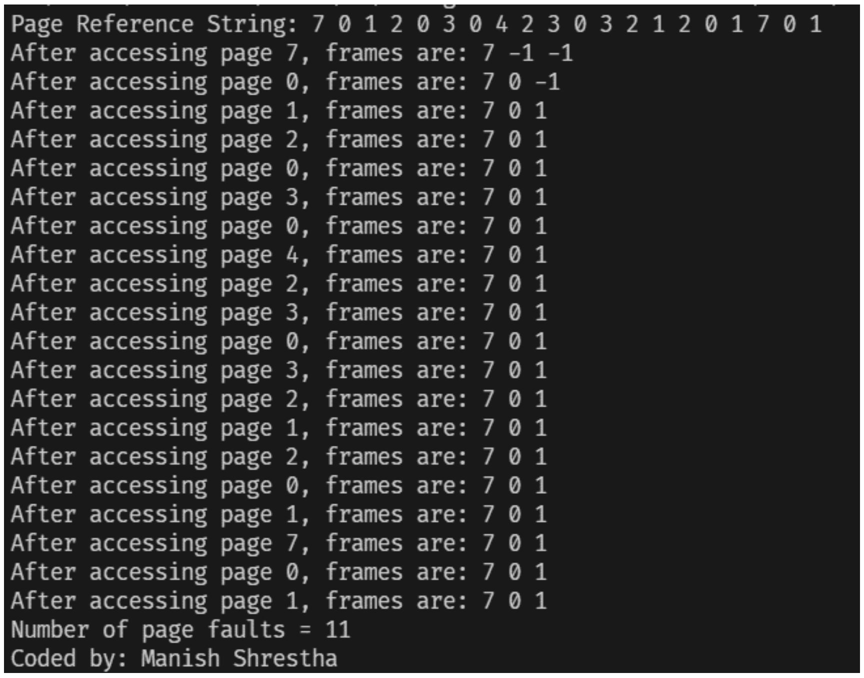


```
Page Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
After accessing page 7, frames are: 7 -1 -1
After accessing page 0, frames are: 7 0 -1
After accessing page 1, frames are: 7 0 1
After accessing page 2, frames are: 2 0 1
After accessing page 0, frames are: 2 0 1
After accessing page 3, frames are: 3 0 1
After accessing page 0, frames are: 3 0 1
After accessing page 4, frames are: 4 0 1
After accessing page 2, frames are: 2 0 1
After accessing page 3, frames are: 3 0 1
After accessing page 0, frames are: 3 0 1
After accessing page 3, frames are: 3 0 1
After accessing page 2, frames are: 3 0 2
After accessing page 1, frames are: 3 0 1
After accessing page 2, frames are: 3 0 2
After accessing page 0, frames are: 3 0 2
After accessing page 1, frames are: 3 0 1
After accessing page 7, frames are: 3 0 7
After accessing page 0, frames are: 3 0 7
After accessing page 1, frames are: 3 0 1
Number of page faults = 14
Coded by: Manish Shrestha
```

Program 1: Simulation of ORP Algorithm in C

```
#include <stdio.h>
#include <stdlib.h>
int findOPR(int pages[], int n, int frames[], int frame_count) {
   int i, j, max_future = 0, pos = 0;
   for(i = 0; i < frame_count; i++) {
      int future = 0;
      for(j = 0; j < n; j++) {
         if(pages[j] == frames[i]) {
            future = 1;
            break;
         }
      }
      if(future == 0) {
         return i;
      }
   }
   return -1;
}
void simulateOPR(int pages[], int n, int capacity) {
   int *frames = (int *)malloc(capacity * sizeof(int));
   int frame_count = 0, page_fault = 0, i, j;
   for(i = 0; i < capacity; ++i) {
      frames[i] = -1;
   }
   printf("Page Reference String: ");
   for(i = 0; i < n; ++i) {
      printf("%d ", pages[i]);
   }
   printf("\n");
   for(i = 0; i < n; ++i) {
      int flag = 0;
      for(j = 0; j < capacity; ++j) {
         if(frames[j] == pages[i]) {
            flag = 1;
            break;
         }
      }
      if(flag == 0) {
         if(frame_count < capacity) {
            frames[frame_count++] = pages[i];
         } else {
            int pos = findOPR(pages, n, frames, capacity);
            frames[pos] = pages[i];
            }
            page_fault++;
        }
        printf("After accessing page %d, frames are: ", pages[i]);
        for(j = 0; j < capacity; ++j) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }
    printf("Number of page faults = %d\n", page_fault);
    printf("Coded by: Manish Shrestha");
    free(frames);
}

int main() {
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 3;
    simulateOPR(pages, n, capacity);
    return 0;
}
```

Output:



```
Page Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
After accessing page 7, frames are: 7 -1 -1
After accessing page 0, frames are: 7 0 -1
After accessing page 1, frames are: 7 0 1
After accessing page 2, frames are: 7 0 1
After accessing page 0, frames are: 7 0 1
After accessing page 3, frames are: 7 0 1
After accessing page 0, frames are: 7 0 1
After accessing page 4, frames are: 7 0 1
After accessing page 2, frames are: 7 0 1
After accessing page 3, frames are: 7 0 1
After accessing page 0, frames are: 7 0 1
After accessing page 3, frames are: 7 0 1
After accessing page 2, frames are: 7 0 1
After accessing page 1, frames are: 7 0 1
After accessing page 2, frames are: 7 0 1
After accessing page 0, frames are: 7 0 1
After accessing page 1, frames are: 7 0 1
After accessing page 7, frames are: 7 0 1
After accessing page 0, frames are: 7 0 1
After accessing page 1, frames are: 7 0 1
Number of page faults = 11
Coded by: Manish Shrestha
```

Conclusion:

By this way the FIFO, LRU and OPR Page Replacement Algorithms were simulated in C.

Title: Simulation of Disk Scheduling Algorithms.

Objectives:

- To simulate The FCFS Disk Scheduling Algorithm in C
- To simulate The SCAN Disk Scheduling Algorithm in C

Theory:

The FCFS (First-Come, First-Served) disk scheduling algorithm is a straightforward and simple method used in operating systems to manage input/output (I/O) requests from processes to access disk blocks. This algorithm processes I/O requests in the order they arrive in the queue, without any reordering or prioritization. When a process generates an I/O request, it is added to the end of the queue, and the operating system services the requests in the same order. This approach ensures fairness, as all requests are serviced sequentially, and there is no starvation, meaning all requests are eventually processed.

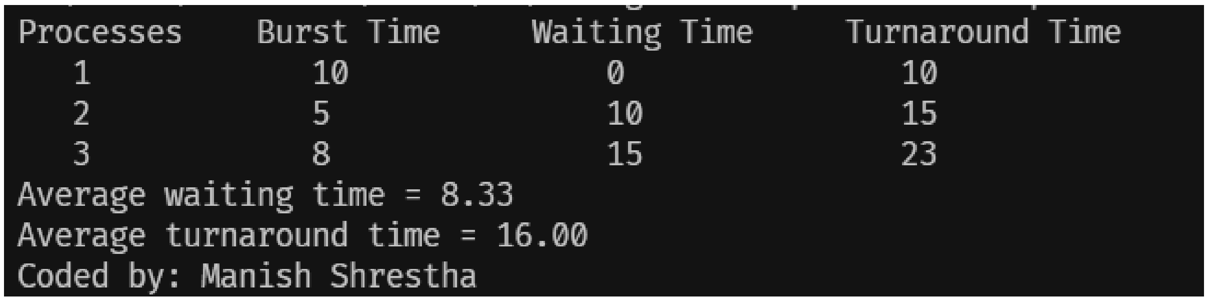
The SCAN disk scheduling algorithm, also known as the Elevator algorithm, is a method used in operating systems to manage input/output (I/O) requests from processes to access disk blocks. This algorithm is designed to improve disk system efficiency by reducing the seek time, which is the time it takes for the disk arm to move to the track where the data is stored.

Program 1: Simulation of FCFS Disk Scheduling Algorithm in C

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

void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   wt[0] = 0;
   for (int i = 1; i < n; i++)
      wt[i] = bt[i - 1] + wt[i - 1];
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])
{
   for (int i = 0; i < n; i++)
      tat[i] = bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[])
{
   int *wt = (int *)malloc(n * sizeof(int));
   int *tat = (int *)malloc(n * sizeof(int));
   int total_wt = 0, total_tat = 0;
```

```
    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);
    printf("Processes    Burst Time     Waiting Time     Turnaround Time\n");
    for (int i = 0; i < n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf("   %d", i + 1);
        printf("  \t\t%d", bt[i]);
        printf("     \t\t%d", wt[i]);
        printf("     \t\t%d\n", tat[i]);
    }
    printf("Average waiting time = %.2f", (float)total_wt / (float)n);
    printf("\nAverage turnaround time = %.2f", (float)total_tat / (float)n);
    printf("Coded by: Manish Shrestha");
    free(wt);
    free(tat);
}

int main()
{
    int processes[] = {1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
    int burst_time[] = {10, 5, 8};
    findavgTime(processes, n, burst_time);
    return 0;
}
```

Output:



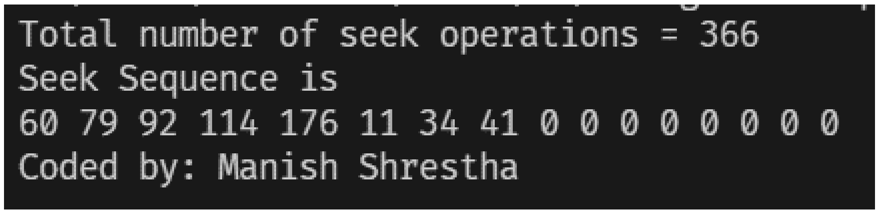
```
Processes    Burst Time     Waiting Time     Turnaround Time
   1            10             0                10
   2            5              10               15
   3            8              15               23
Average waiting time = 8.33
Average turnaround time = 16.00
Coded by: Manish Shrestha
```

<u>Program 2</u>: Simulation of SCAN Disk Scheduling Algorithm in C

```
#include <stdio.h>
#include <stdlib.h>
int compare(const void *a, const void *b) {
   return (*(int*)a - *(int*)b);
}

void SCAN(int arr[], int head, int size) {
   int seek_count = 0;
   int distance, cur_track;
   int *left = (int *)malloc(size * sizeof(int));
   int *right = (int *)malloc(size * sizeof(int));
   int *seek_sequence = (int *)malloc(2 * size * sizeof(int));
   int left_size = 0, right_size = 0;
   for (int i = 0; i < size; i++) {
      if (arr[i] < head)
         left[left_size++] = arr[i];
      else if (arr[i] > head)
         right[right_size++] = arr[i];
   }
   qsort(left, left_size, sizeof(int), compare);
   qsort(right, right_size, sizeof(int), compare);
   for (int i = 0; i < right_size; i++) {
      cur_track = right[i];
      seek_sequence[i] = cur_track;
      distance = abs(cur_track - head);
      seek_count += distance;
      head = cur_track;
   }
   head = 0;
   seek_count += (200 - 1);
   for (int i = 0; i < left_size; i++) {
      cur_track = left[i];
      seek_sequence[i + right_size] = cur_track;
      distance = abs(cur_track - head);
      seek_count += distance;
      head = cur_track;
   }
   printf("Total number of seek operations = %d\n", seek_count);
   printf("Seek Sequence is\n");
   for (int i = 0; i < 2 * size; i++) {
      printf("%d ", seek_sequence[i]);
   }
   printf("\n");
```

```
    free(left);
    free(right);
    free(seek_sequence);
}

int main() {
    int arr[] = {176, 79, 34, 60, 92, 11, 41, 114};
    int head = 50;
    int size = sizeof(arr) / sizeof(arr[0]);
    SCAN(arr, head, size);
    return 0;
}
```

Output:



```
Total number of seek operations = 366
Seek Sequence is
60 79 92 114 176 11 34 41 0 0 0 0 0 0 0 0
Coded by: Manish Shrestha
```

Conclusion:

By this way the FCFS and SCAN Disk Scheduling Algorithms were simulated in C.