Title: Installation of Linux in Virtual Machine

Objectives:

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

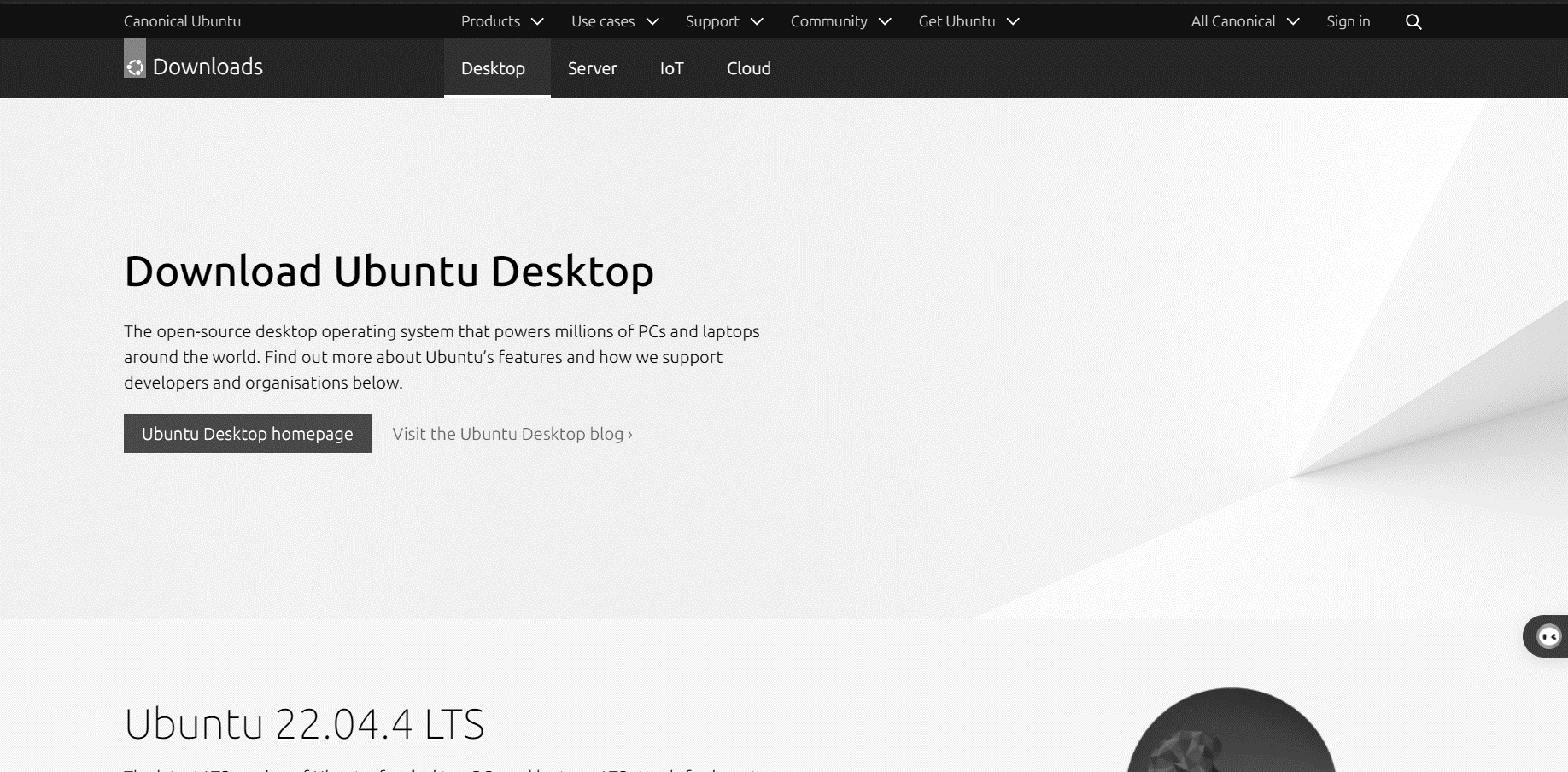
Theory:

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.

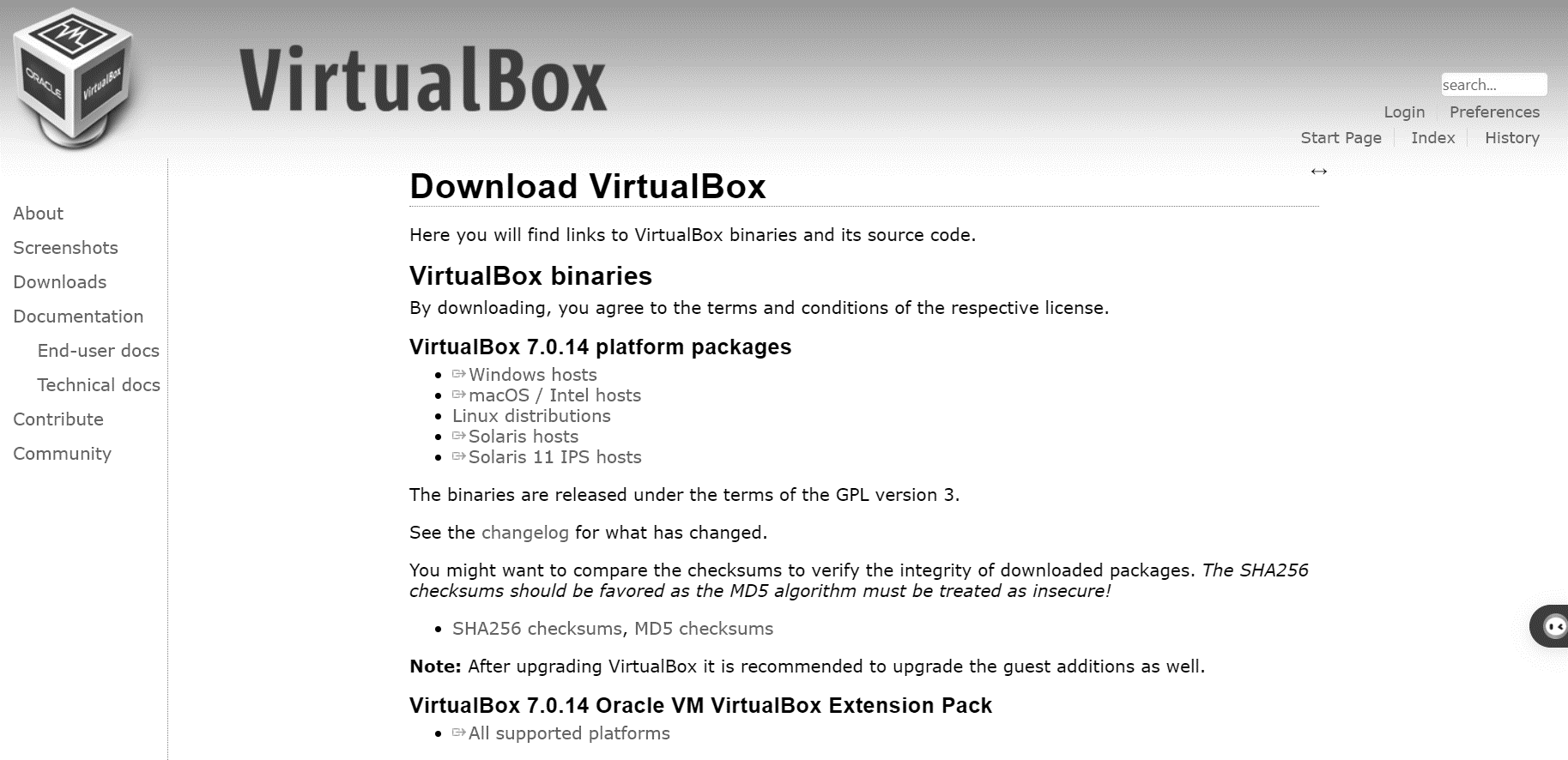
Linux is an open-source operating system developed by Linus Torvalds in 1991. It is based on the Linux Kernel, which is the core of the operating system, managing how the computer interacts with its hardware and resources. The Linux Kernel is often referred to as the brain of the operating system because 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:

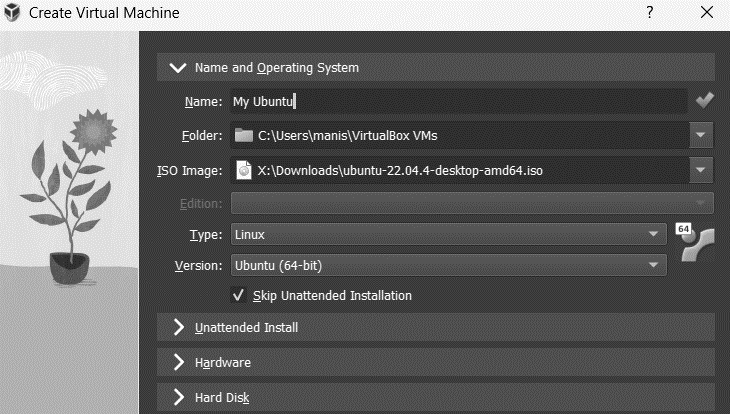
**Download a Linux Distribution:** Choose a Linux distribution like Ubuntu, CentOS, or Debian. You can download the ISO image from their respective websites.



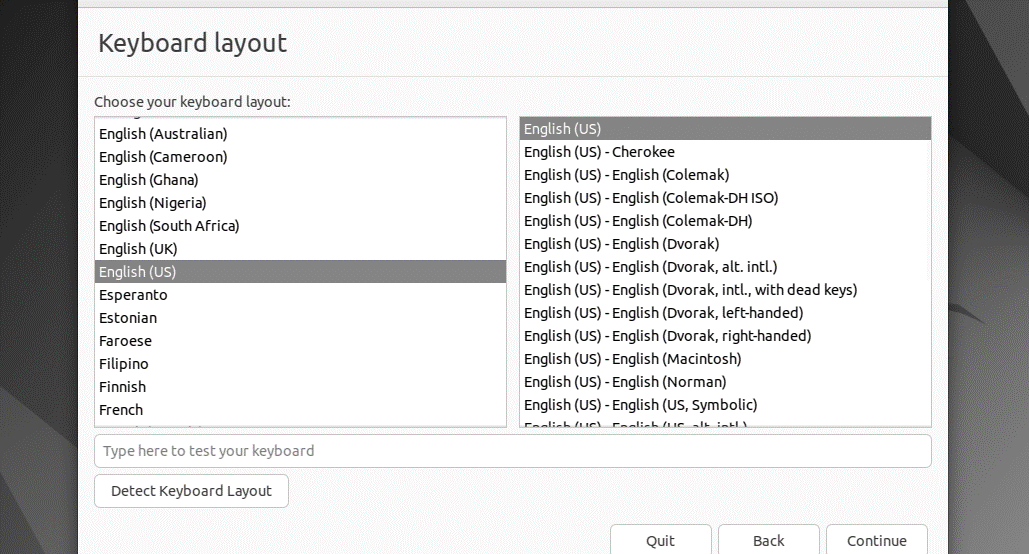
**Install a Virtual Machine Software:** You can use software like VirtualBox, VMware, or Parallels Desktop to create a virtual machine on your computer.



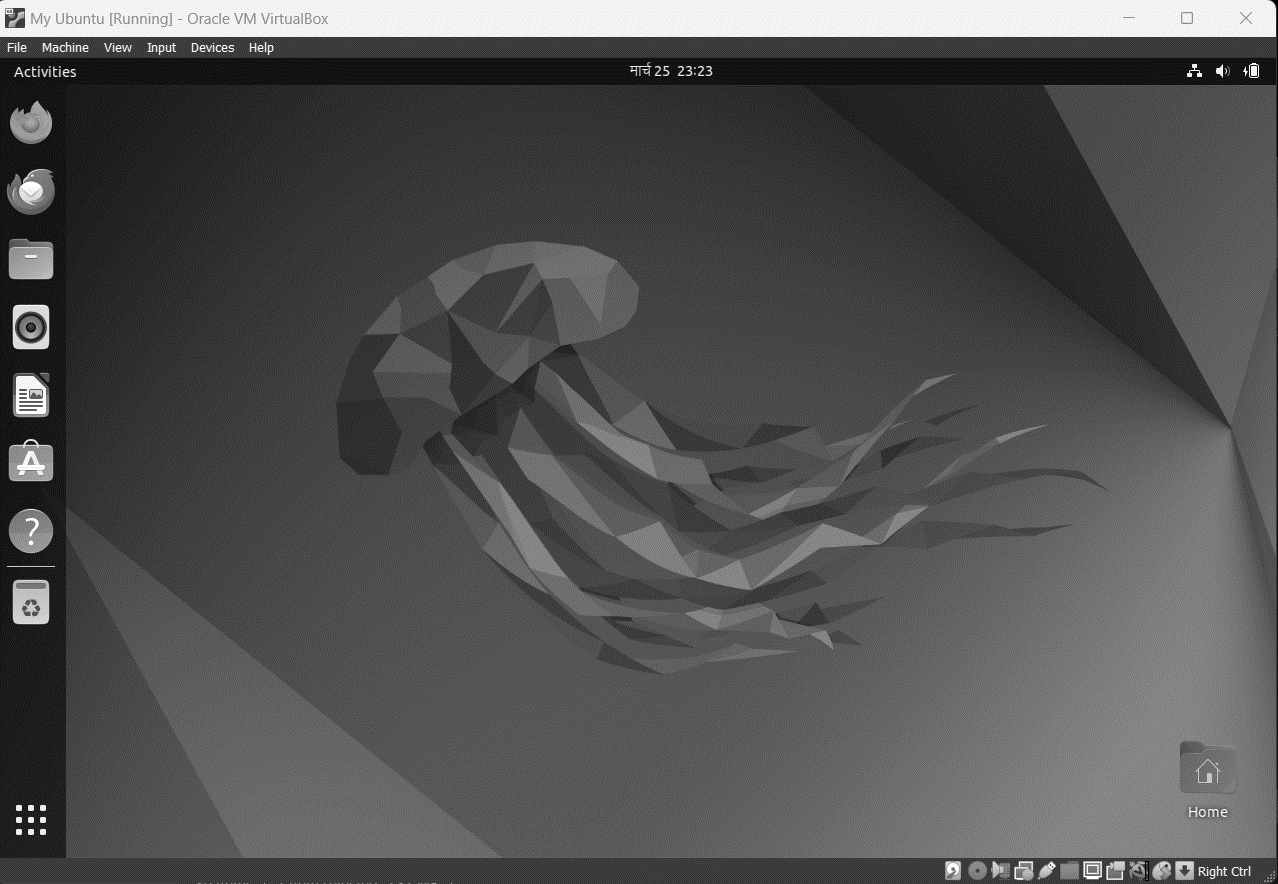
**Create a Virtual Machine:** Open the virtual machine software and create a new virtual machine. During the setup process, you'll need to specify the Linux ISO image you downloaded as the installation media.



**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.



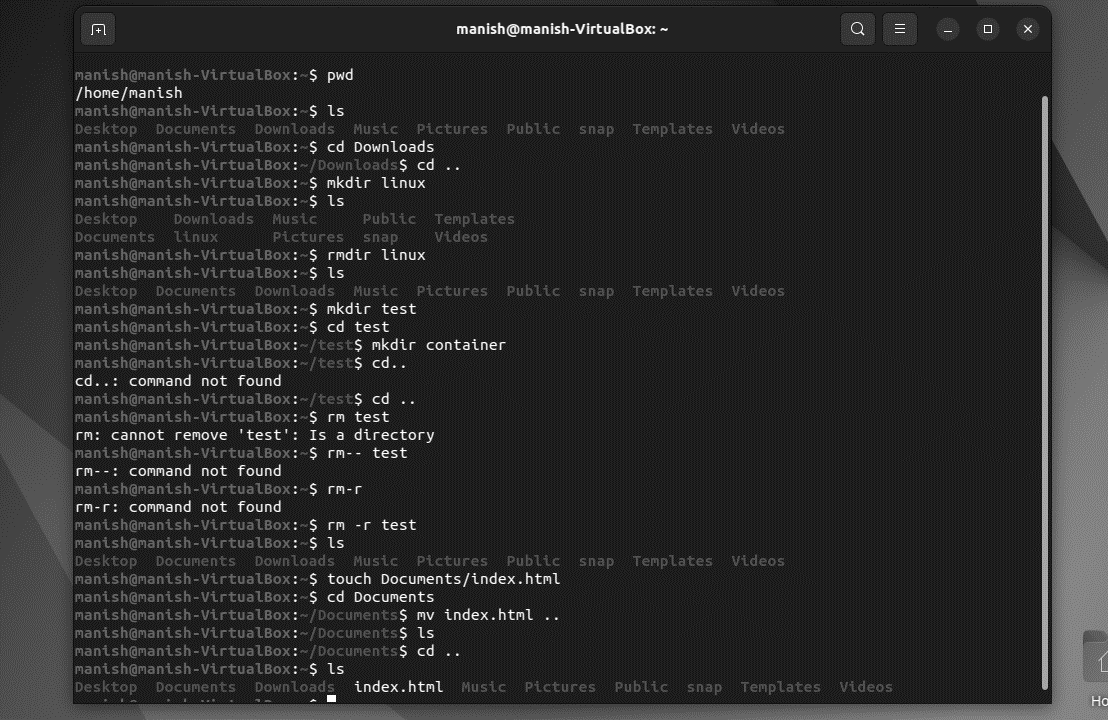
**Finish Installation:** Once the installation is complete, you'll have a Linux system running inside your virtual machine.



Running Linux Commands:

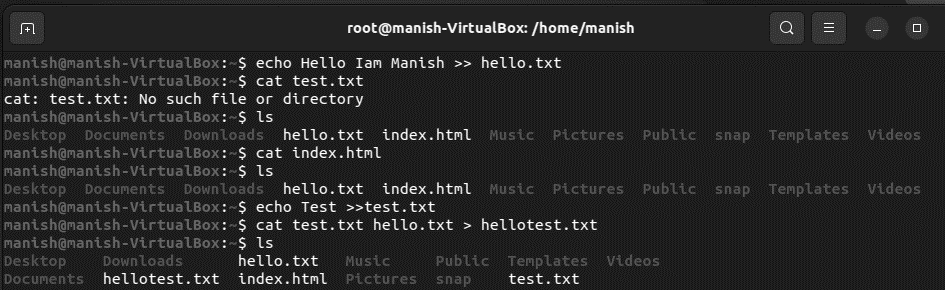
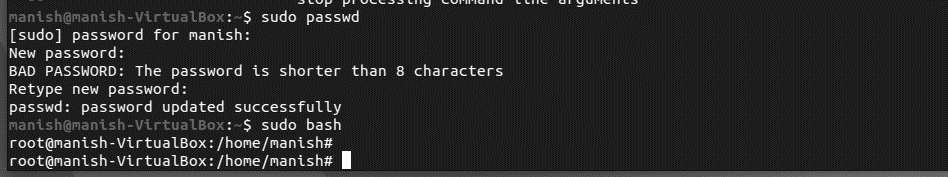
* Basic Commands:

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



* Intermediate Commands:

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



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()

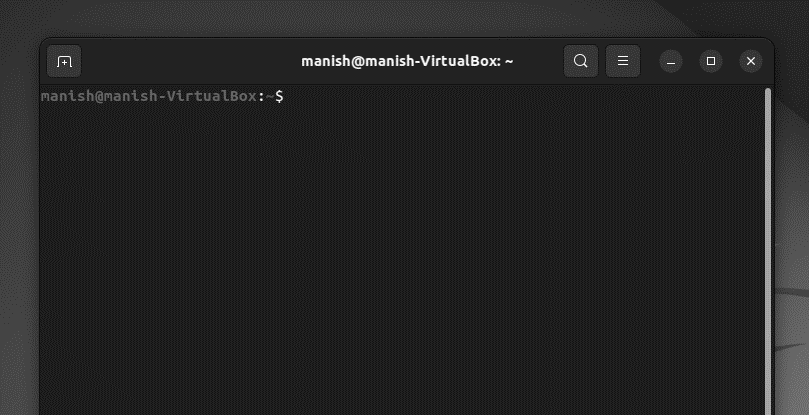
Theory:

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



1. 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

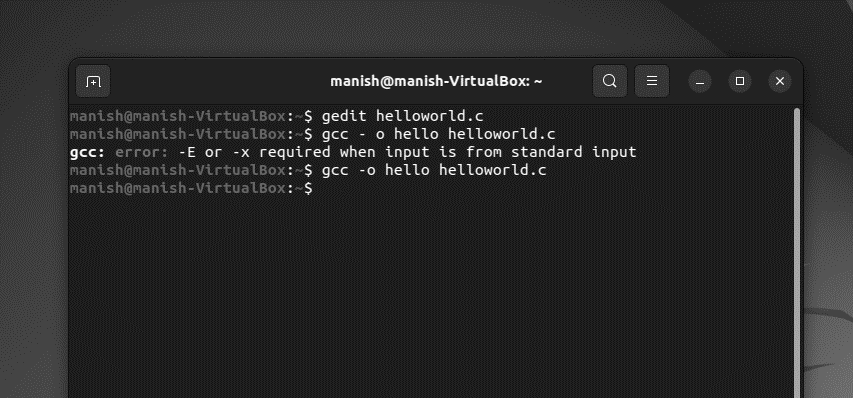


1. Compile the Source Code:

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



1. Execute the program:

Type the command

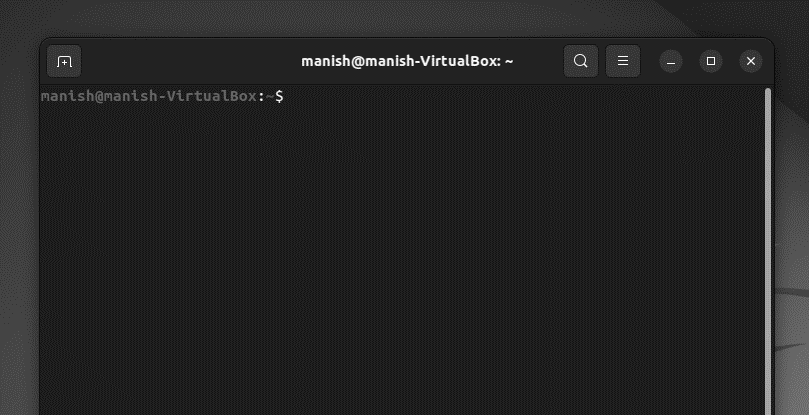
./hello

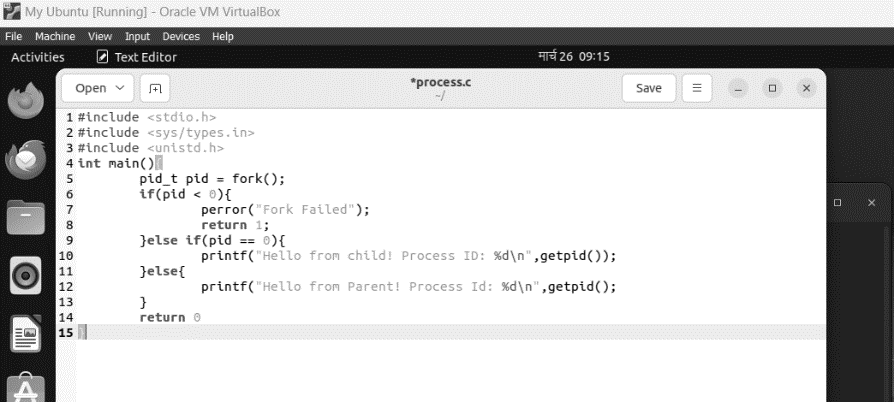
This should result in output:

Hello World

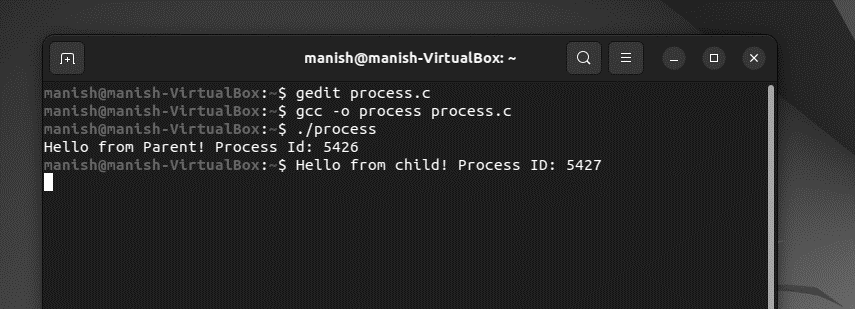


Steps for Running creating process in Linux:

1. Open the Terminal from the applications section
2. Use a text editor to create a new process file in c



1. Compile and Execute the Source Code



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);

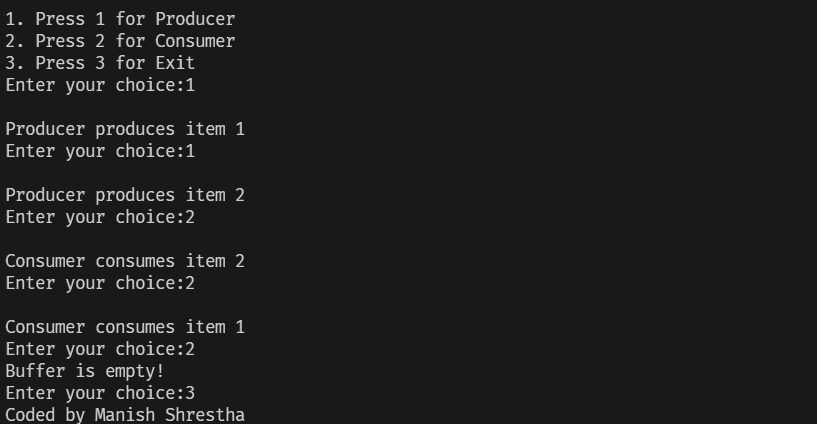
            break;

        }

    }

}

Output 1:



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

1. Waiting time (n) = waiting time (n-1) + Burst time (n-1)
2. Turnaround time (n) = waiting time (n) + Burst time (n)

Step 6: Calculate

1. Average waiting time = Total waiting Time / Number of process
2. 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

1. Waiting time (n) = waiting time (n-1) + Burst time(n-1)
2. Turnaround time (n) = waiting time(n) + Burst time(n)

Step 8: Calculate

1. Average waiting time = Total waiting time / Number of process
2. Average Turnaround time = Total Turnaround time / Number of process

Step 9: Stop the process

Program 1: 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;

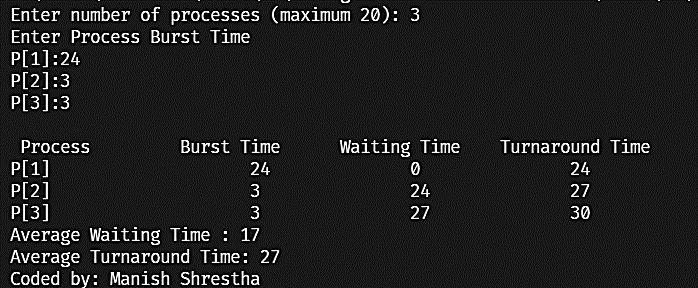
    printf("\nAverage Waiting Time : %d",avwt);

    printf("\nAverage Turnaround Time: %d",avtat);

printf("\nCoded by: Manish Shrestha");

}

Output 1:



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;

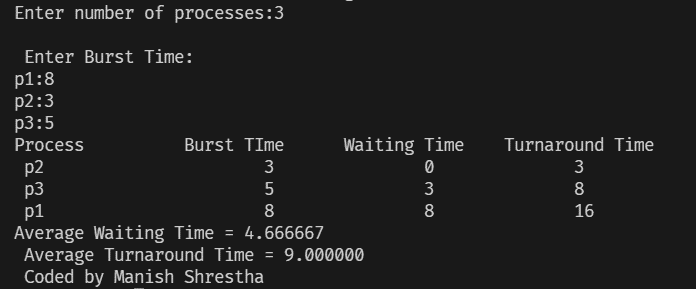
    printf("\nAverage Waiting Time = %f",avg\_wt);

    printf("\n Average Turnaround Time = %f",avg\_tat);

printf("\nCoded by: Manish Shrestha");

}

Output :



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

1. Waiting time (n) = waiting time (n-1) + Burst time(n-1)
2. Turnaround time (n) = waiting time(n) + Burst time(n)

Step 8: Calculate

1. Average waiting time = Total waiting time / Number of process
2. 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

1. 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)
2. 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:

1. Average waiting time = Total waiting Time / Number of Process
2. 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;

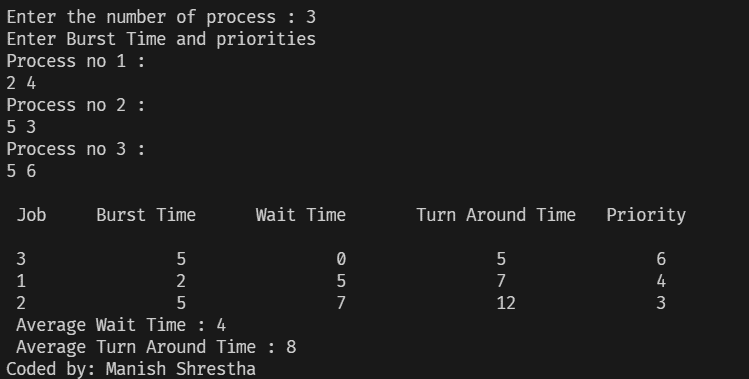
    printf("\n Average Wait Time : %d", awt);

    printf("\n Average Turn Around Time : %d", atat);

    printf("\nCoded by: Manish Shrestha");

}

Output:



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;

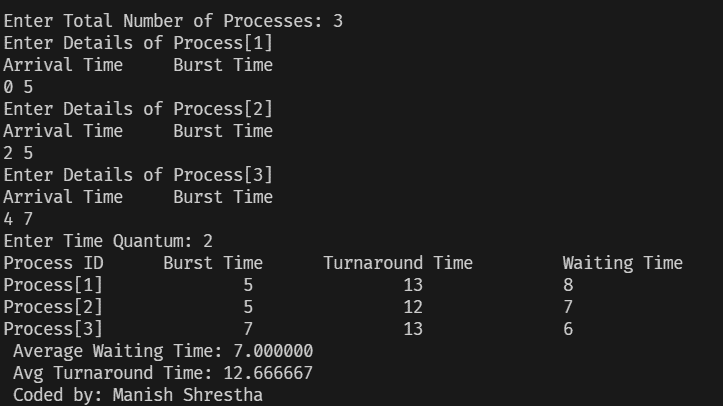
      printf("\n Average Waiting Time:t%f", average\_wait\_time);

      printf("\n Avg Turnaround Time:t%f", average\_turnaround\_time);

      printf("\n Coded by: Manish Shrestha");

}

Output:



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]);

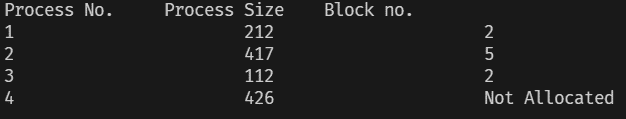
    n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0;

}

Output:



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};

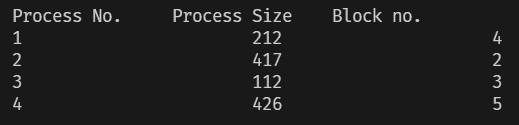
    int m = sizeof(blockSize) / sizeof(blockSize[0]);

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

    bestFit(blockSize, m, processSize, n);

}

Output:



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};

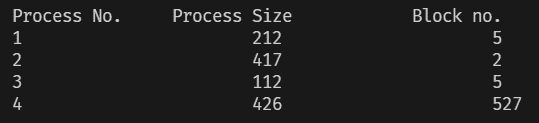
    int m = sizeof(blockSize) / sizeof(blockSize[0]);

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

    worstFit(blockSize, m, processSize, n);

}

Output:



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.