**Question 1:** Write a program to implement **fork()** system call:

**Answer:** Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

We can implement **fork()** system call in the Ubuntu Linux operating system as follows:

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main (){

    // make two process which run same program after this instruction

    fork();

// fork() return 0 to parent process and pid of child process to child process

    fork();

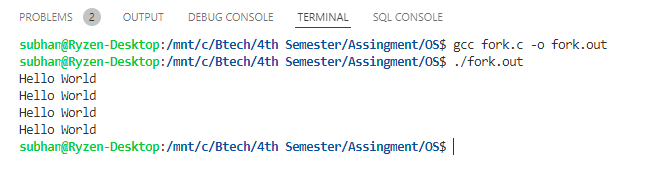
// fork() return 0 to parent process and 1 to child process

    printf ("Hello World\n");

    return 0;

}

**Output:**

****

**Question 2:** Write a program to implement **Critical Section Problem.**

**Answer:** The critical section problem is used to design a protocol followed by a group of processes, so that when one process has entered its critical section, no other process is allowed to execute in its critical section.

We can implement Critical Section problem in the Ubuntu Linux operating system as follows:

#include <stdio.h>

#include <pthread.h>

#include <stdint.h>

#include <stdlib.h>

#define TRUE    1

#define FALSE   0

int N;

int global = 10;

int entering[100];

int number[100];

int max(int number[100]) {

    int i = 0;

    int maximum = number[0];

    for (i = 0; i < N; i++) {

        if (maximum < number[i])

            maximum = number[i];

        }

    return maximum;

    }

void lock(int i) {

    int j = 0;

    entering[i] = TRUE;

    number[i] = 1 + max(number);

    entering[i] = FALSE;

    for (j = 0; j < N; j++) {

        while (entering[j]);

        while (number[j] != 0 && (number[j] < number[i] || (number[i] == number[j]) && j < i)) {}

    }

}

void unlock(int i) {

    number[i] = 0;

}

void \*fn(void \*integer) {

    int i = (intptr\_t)integer;

    lock(i);

    printf("\n\n-----------Process %d---------",i);

    printf("\nProcess %d is Entering Critical Section\n",i);

    global++;

    printf("%d is the value of global \n",global);

    printf("Process %d is leaving Critical Section\n",i);

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

    unlock(i);

}

int main()

{

    printf("Enter Number of Process\n");

    scanf("%d",&N);

    int th[N];

    void \*fn(void \*);

    pthread\_t thread[N];

    int i = 0;

    for (i = 0; i < N; i++) {

        th[i] = pthread\_create(&thread[i], NULL, fn, (void \*)(intptr\_t)i);

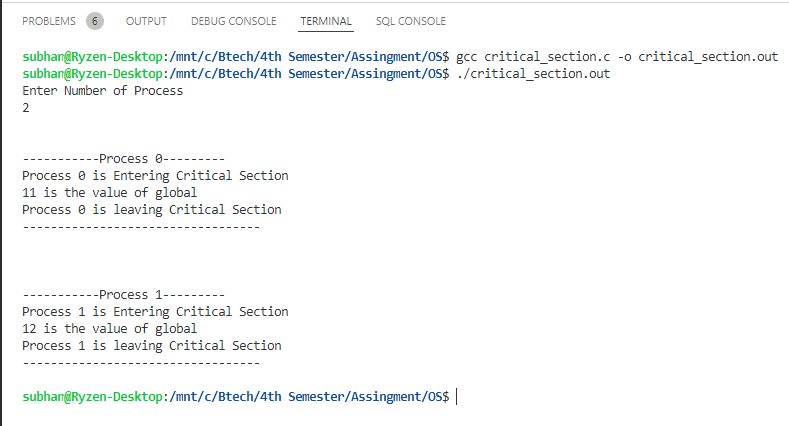
        pthread\_join(thread[i], NULL);

    }

    return EXIT\_SUCCESS;

}

**Output:**



**Question 3:** Write a program to implement Classical problems of Process Synchronization.

**Answer:** We can implement Classical problems of Process Synchronization in the Ubuntu Linux operating system as follows:

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

void \*thread(void \*arg)

{

    // wait

    sem\_wait(&mutex);

    printf("\nEntered..\n");

    // critical section

    sleep(4);

    // signal

    printf("\nJust Exiting...\n");

    sem\_post(&mutex);

}

int main()

{

    sem\_init(&mutex, 0, 1);

    pthread\_t t1, t2;

    pthread\_create(&t1, NULL, thread, NULL);

    sleep(2);

    pthread\_create(&t2, NULL, thread, NULL);

    pthread\_join(t1, NULL);

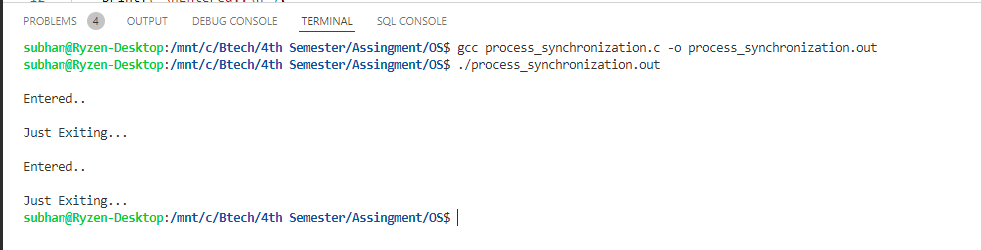
    pthread\_join(t2, NULL);

    sem\_destroy(&mutex);

    return 0;

}

**Output:**



**Question 4:** Write a program to implement **non-preemptive scheduling algorithm.**

**Answer:** Non-preemptive Scheduling is a CPU scheduling technique the process takes the resource (CPU time) and holds it till the process gets terminated or is pushed to the waiting state. No process is interrupted until it is completed, and after that processor switches to another process. Algorithms that are based on non-preemptive Scheduling are non-preemptive priority, and shortest Job first.  
We can implement non-preemptive scheduling algorithm in the Ubuntu Linux operating system as follows:

#include <stdio.h>

struct process

{

    int id, WT, AT, BT, TAT, PR;

};

struct process a[10];

// function for swapping

void swap(int \*b, int \*c)

{

    int tem;

    tem = \*c;

    \*c = \*b;

    \*b = tem;

}

// Driver function

int main()

{

    int n, check\_ar = 0;

    int Cmp\_time = 0;

    float Total\_WT = 0, Total\_TAT = 0, Avg\_WT, Avg\_TAT;

    printf("Enter the number of process \n");

    scanf("%d", &n);

    printf("Enter the Arrival time , Burst time and priority of the process\n");

    printf("AT BT PR\n");

    for (int i = 0; i < n; i++)

    {

        scanf("%d%d%d", &a[i].AT, &a[i].BT, &a[i].PR);

        a[i].id = i + 1;

        // here we are checking that arrival time

        // of the process are same or different

        if (i == 0)

            check\_ar = a[i].AT;

        if (check\_ar != a[i].AT)

            check\_ar = 1;

    }

    // if process are arrived at the different time

    // then sort the process on the basis of AT

    if (check\_ar != 0)

    {

        for (int i = 0; i < n; i++)

        {

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

            {

                if (a[j].AT > a[j + 1].AT)

                {

                    swap(&a[j].id, &a[j + 1].id);

                    swap(&a[j].AT, &a[j + 1].AT);

                    swap(&a[j].BT, &a[j + 1].BT);

                    swap(&a[j].PR, &a[j + 1].PR);

                }

            }

        }

    }

    // logic of Priority scheduling ( non preemptive) algo

    // if all the process are arrived at different time

    if (check\_ar != 0)

    {

        a[0].WT = a[0].AT;

        a[0].TAT = a[0].BT - a[0].AT;

        // cmp\_time for completion time

        Cmp\_time = a[0].TAT;

        Total\_WT = Total\_WT + a[0].WT;

        Total\_TAT = Total\_TAT + a[0].TAT;

        for (int i = 1; i < n; i++)

        {

            int min = a[i].PR;

            for (int j = i + 1; j < n; j++)

            {

                if (min > a[j].PR && a[j].AT <= Cmp\_time)

                {

                    min = a[j].PR;

                    swap(&a[i].id, &a[j].id);

                    swap(&a[i].AT, &a[j].AT);

                    swap(&a[i].BT, &a[j].BT);

                    swap(&a[i].PR, &a[j].PR);

                }

            }

            a[i].WT = Cmp\_time - a[i].AT;

            Total\_WT = Total\_WT + a[i].WT;

            // completion time of the process

            Cmp\_time = Cmp\_time + a[i].BT;

            // Turn Around Time of the process

            // compl-Arival

            a[i].TAT = Cmp\_time - a[i].AT;

            Total\_TAT = Total\_TAT + a[i].TAT;

        }

    }

    // if all the process are arrived at same time

    else

    {

        for (int i = 0; i < n; i++)

        {

            int min = a[i].PR;

            for (int j = i + 1; j < n; j++)

            {

                if (min > a[j].PR && a[j].AT <= Cmp\_time)

                {

                    min = a[j].PR;

                    swap(&a[i].id, &a[j].id);

                    swap(&a[i].AT, &a[j].AT);

                    swap(&a[i].BT, &a[j].BT);

                    swap(&a[i].PR, &a[j].PR);

                }

            }

            a[i].WT = Cmp\_time - a[i].AT;

            // completion time of the process

            Cmp\_time = Cmp\_time + a[i].BT;

            // Turn Around Time of the process

            // compl-Arrival

            a[i].TAT = Cmp\_time - a[i].AT;

            Total\_WT = Total\_WT + a[i].WT;

            Total\_TAT = Total\_TAT + a[i].TAT;

        }

    }

    Avg\_WT = Total\_WT / n;

    Avg\_TAT = Total\_TAT / n;

    // Printing of the results

    printf("The process are\n");

    printf("ID WT TAT\n");

    for (int i = 0; i < n; i++)

    {

        printf("%d\t%d\t%d\n", a[i].id, a[i].WT, a[i].TAT);

    }

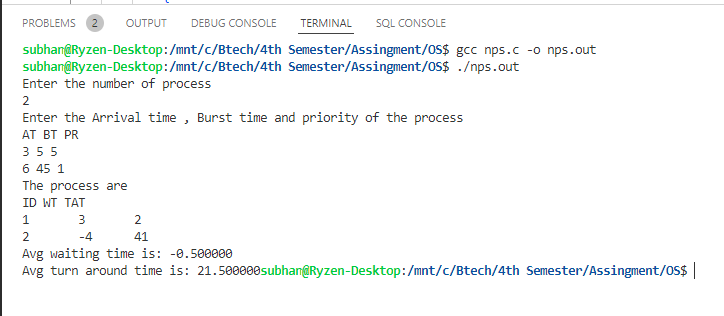
    printf("Avg waiting time is: %f\n", Avg\_WT);

    printf("Avg turn around time is: %f", Avg\_TAT);

    return 0;

}

**Output:**

****

**Question 5:** Write a program to implement **Preemptive scheduling algorithm.**

**Answer:** Preemptive Scheduling is a CPU scheduling technique that works by dividing time slots of CPU to a given process. The time slot given might be able to complete the whole process or might not be able to it. When the burst time of the process is greater than CPU cycle, it is placed back into the ready queue and will execute in the next chance. This scheduling is used when the process switch to ready state. Algorithms that are backed by preemptive Scheduling are round-robin (RR), priority, SRTF (shortest remaining time first).

We can implement Preemptive scheduling algorithm in the Ubuntu Linux operating system as follows:

#include <stdio.h>

struct process

{

    int WT, AT, BT, TAT, PT;

};

struct process a[10];

int main()

{

    int n, temp[10], t, count = 0, short\_p;

    float total\_WT = 0, total\_TAT = 0, Avg\_WT, Avg\_TAT;

    printf("Enter the number of the process\n");

    scanf("%d", &n);

    printf("Enter the arrival time , burst time and priority of the process\n");

    printf("AT BT PT\n");

    for (int i = 0; i < n; i++)

    {

        scanf("%d%d%d", &a[i].AT, &a[i].BT, &a[i].PT);

        // copying the burst time in

        // a temp array fot futher use

        temp[i] = a[i].BT;

    }

    // we initialize the burst time

    // of a process with maximum

    a[9].PT = 10000;

    for (t = 0; count != n; t++)

    {

        short\_p = 9;

        for (int i = 0; i < n; i++)

        {

            if (a[short\_p].PT > a[i].PT && a[i].AT <= t && a[i].BT > 0)

            {

                short\_p = i;

            }

        }

        a[short\_p].BT = a[short\_p].BT - 1;

        // if any process is completed

        if (a[short\_p].BT == 0)

        {

            // one process is completed

            // so count increases by 1

            count++;

            a[short\_p].WT = t + 1 - a[short\_p].AT - temp[short\_p];

            a[short\_p].TAT = t + 1 - a[short\_p].AT;

            // total calculation

            total\_WT = total\_WT + a[short\_p].WT;

            total\_TAT = total\_TAT + a[short\_p].TAT;

        }

    }

    Avg\_WT = total\_WT / n;

    Avg\_TAT = total\_TAT / n;

    // printing of the answer

    printf("ID WT TAT\n");

    for (int i = 0; i < n; i++)

    {

        printf("%d %d\t%d\n", i + 1, a[i].WT, a[i].TAT);

    }

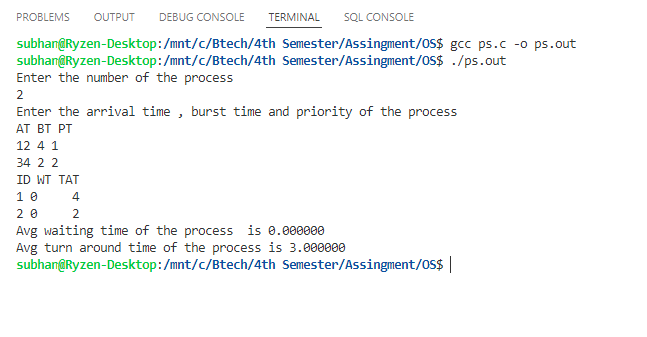
    printf("Avg waiting time of the process  is %f\n", Avg\_WT);

    printf("Avg turn around time of the process is %f\n", Avg\_TAT);

    return 0;

}

**Output:**



**Question 6:** Write a program to implement **Banker’s algorithm.**

**Answer:** 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.

We can implement Banker’s algorithm in the Ubuntu Linux operating system as follows:

#include <stdio.h>

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;

                }

            }

        }

    }

    int flag = 1;

    for (int i = 0; i < n; i++)

    {

        if (f[i] == 0)

        {

            flag = 0;

            printf("The following system is not safe");

            break;

        }

    }

    if (flag == 1)

    {

        printf("Following is the SAFE Sequence\n");

        for (i = 0; i < n - 1; i++)

        {

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

        }

        printf(" P%d", ans[n - 1]);

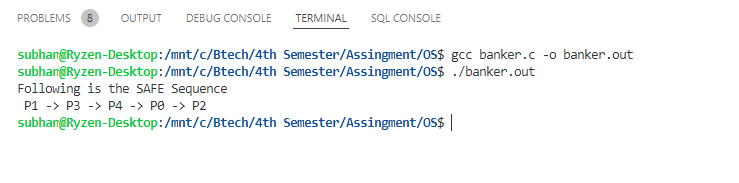
        printf("\n");

    }

    return (0);

}

**Output:**



**Question 7:** Write a program to implement **page replacement algorithm.**

**Answer:** The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault).

We can implement page replacement algorithm in the Ubuntu Linux operating system as follows:

// C program for FIFO page replacement algorithm

#include <stdio.h>

int main()

{

    int incomingStream[] = {4, 1, 2, 4, 5};

    int pageFaults = 0;

    int frames = 3;

    int m, n, s, pages;

    pages = sizeof(incomingStream) / sizeof(incomingStream[0]);

    printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

    int temp[frames];

    for (m = 0; m < frames; m++)

    {

        temp[m] = -1;

    }

    for (m = 0; m < pages; m++)

    {

        s = 0;

        for (n = 0; n < frames; n++)

        {

            if (incomingStream[m] == temp[n])

            {

                s++;

                pageFaults--;

            }

        }

        pageFaults++;

        if ((pageFaults <= frames) && (s == 0))

        {

            temp[m] = incomingStream[m];

        }

        else if (s == 0)

        {

            temp[(pageFaults - 1) % frames] = incomingStream[m];

        }

        printf("\n");

        printf("%d\t\t\t", incomingStream[m]);

        for (n = 0; n < frames; n++)

        {

            if (temp[n] != -1)

                printf(" %d\t\t\t", temp[n]);

            else

                printf(" - \t\t\t");

        }

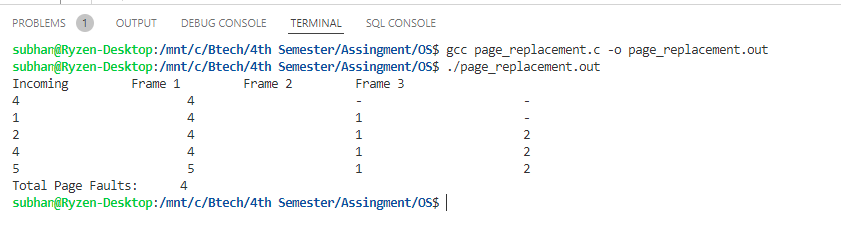
    }

    printf("\nTotal Page Faults:\t%d\n", pageFaults);

    return 0;

}

**Output:**



**Question 8:** Write a program to implement **Disk Scheduling algorithm.**

**Answer:** Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

We can implement Classical problems of Process Synchronization in the Ubuntu Linux operating system as follows:

#include <stdio.h>

#include <stdlib.h>

int main()

{

    int RQ[100], i, n, TotalHeadMoment = 0, initial;

    printf("Enter the number of Requests\n");

    scanf("%d", &n);

    printf("Enter the Requests sequence\n");

    for (i = 0; i < n; i++)

        scanf("%d", &RQ[i]);

    printf("Enter initial head position\n");

    scanf("%d", &initial);

    // logic for FCFS disk scheduling

    for (i = 0; i < n; i++)

    {

        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);

        initial = RQ[i];

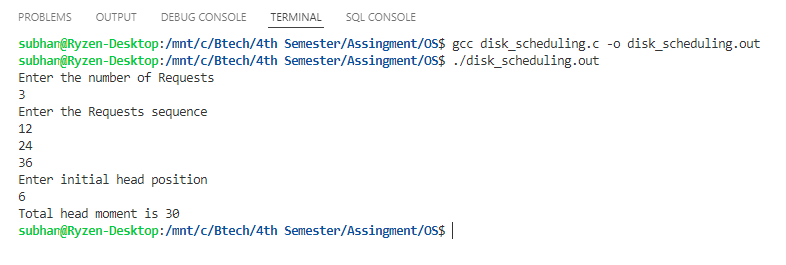
    }

    printf("Total head moment is %d \n", TotalHeadMoment);

    return 0;

}

**Output:**



**Question 9:** Write a program to implement **file allocation methods.**

**Answer:** The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation

We can implement file allocation methods in the Ubuntu Linux operating system as follows:

#include <stdio.h>

int main()

{

    char name[10][30];

    int start[10], length[10], num;

    printf("Enter the number of files to be allocated\n");

    scanf("%d", &num);

    int count = 0, k, j;

    for (int i = 0; i < num; i++)

    {

        printf("Enter the name of the file %d\n", i + 1);

        scanf("%s", &name[i][0]);

        printf("Enter the start block of the file %d\n", i + 1);

        scanf("%d", &start[i]);

        printf("Enter the length of the file %d\n", i + 1);

        scanf("%d", &length[i]);

        for (j = 0, k = 1; j < num && k < num; j++, k++)

        {

            if (start[j + 1] <= start[j] || start[j + 1] >= length[j])

            {

            }

            else

            {

                count++;

            }

        }

        if (count == 1)

        {

            printf("%s cannot be allocated disk space\n", name[i]);

        }

    }

    printf("File Allocation Table\n");

    printf("%s%40s%40s\n", "File Name", "Start Block", "Length");

    printf("%s%50d%50d\n", name[0], start[0], length[0]);

    for (int i = 0, j = 1; i < num && j < num; i++, j++)

    {

        if (start[i + 1] <= start[i] || start[i + 1] >= length[i])

        {

            printf("%s%50d%50d\n", name[j], start[j], length[j]);

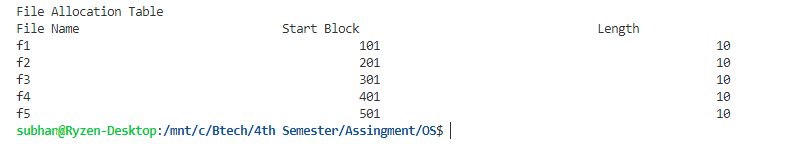
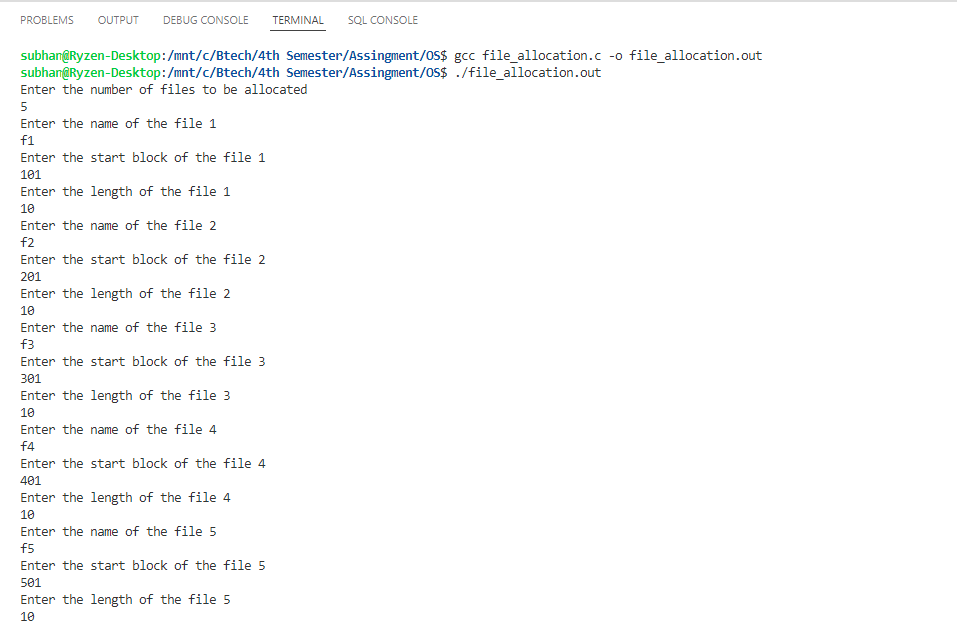
        }

    }

    return 0;

}

**Output:**



**Question 10:** Write a program to implement **MFT & MVT first fit and best fit.**

**Answer:** We can implement MFT & MVT first fit and best fit in the Ubuntu Linux operating system as follows:

// MFT

#include <stdio.h>

int main()

{

    int i, m, n, tot, s[20];

    printf("Enter total memory size: ");

    scanf("%d", &tot);

    printf("Enter no. of pages: ");

    scanf("%d", &n);

    printf("Enter memory for OS: ");

    scanf("%d", &m);

    for (i = 0; i < n; i++)

    {

        printf("Enter size of page %d:", i + 1);

        scanf("%d", &s[i]);

    }

    tot = tot - m;

    for (i = 0; i < n; i++)

    {

        if (tot >= s[i])

        {

            printf("Allocate page %d\n", i + 1);

            tot = tot - s[i];

        }

        else

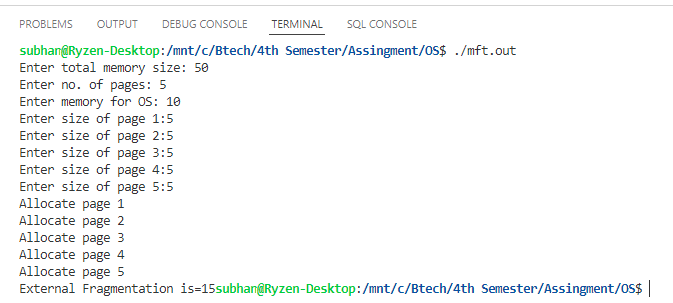
            printf("process p%d is blocked\n", i + 1);

    }

    printf("External Fragmentation is=%d", tot);

}

**Output:**

****

// MVT

#include <stdio.h>

int main()

{

    int ms, i, ps[20], n, size, p[20], s, intr = 0;

    printf("Enter size of memory: ");

    scanf("%d", &ms);

    printf("Enter memory for OS: ");

    scanf("%d", &s);

    ms -= s;

    printf("Enter no.of partitions to be divided: ");

    scanf("%d", &n);

    size = ms / n;

    for (i = 0; i < n; i++)

    {

        printf("Enter process and process size for partition %d: ", i + 1);

        scanf("%d%d", &p[i], &ps[i]);

        if (ps[i] <= size)

        {

            intr = intr + size - ps[i];

            printf("process %d is allocated\n", p[i]);

        }

        else

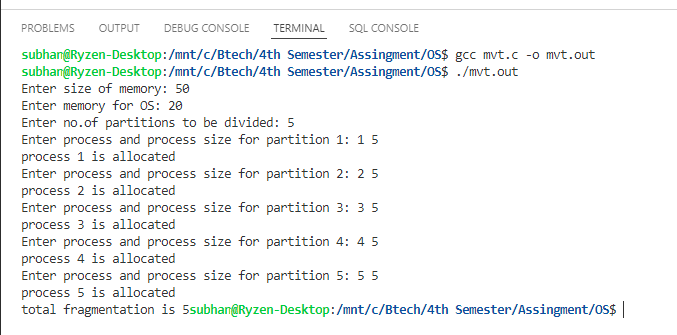
            printf("process %d is blocked", p[i]);

    }

    printf("total fragmentation is %d", intr);

}

**Output:**

****