**Week 1**

**Program 1) Write a program to create a child process using system call fork().**

**Source Code**

#include <stdio.h>

#include <unistd.h>

int main()

{

    pid\_t pid = fork();

    if (pid < 0)

    {

        printf("Fork failed!\n");

        return 1;

    }

    else if (pid == 0)

    {

        printf("Child Process PID: %d\n", getpid());

    }

    else

    {

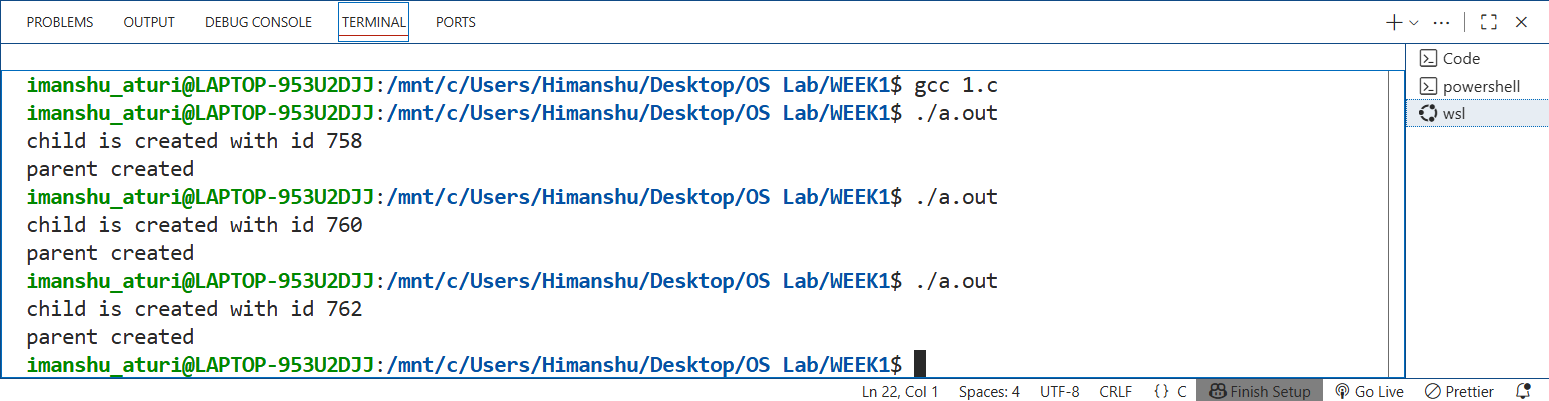
        printf("Parent Process PID: %d, Child PID: %d\n", getpid(), pid);

    }

    return 0;

}

**Output:**

****

**Program 2) Write a program to print process Id's of parent and child process i.e. parent should print its own and its child process id while child process should print its own and its parent process id.**

**Source Code**

#include <stdio.h>

#include <unistd.h>

int main()

{

    pid\_t id = fork();

    if (id < 0)

    {

        printf("Forked Failed\n");

        return 1;

    }

    else if (id == 0)

    {

        printf("Child Process\n");

        printf("Child PID: %d\n", getpid());

        printf("Parent PID: %d\n", getppid());

    }

    else

    {

        printf("Parent Process\n");

        printf("Child PID: %d\n", id);

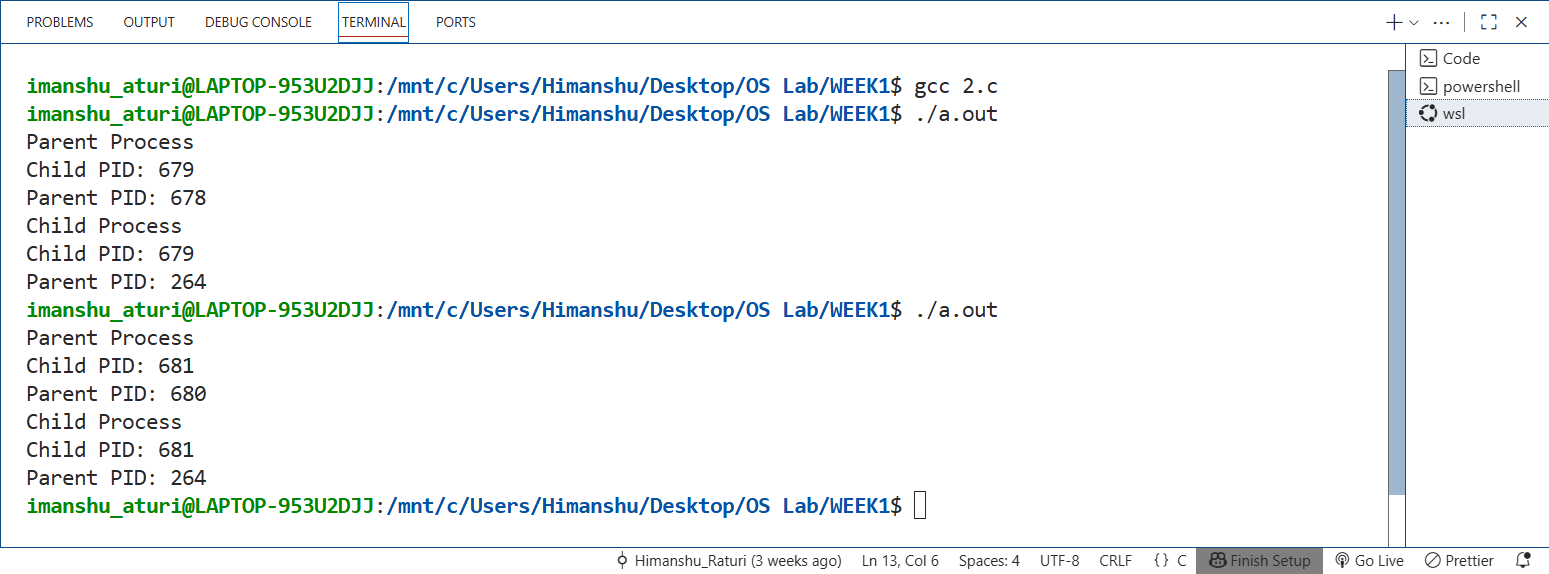
        printf("Parent PID: %d\n", getpid());

    }

    return 0;

}

**OUTPUT**



**Program 3) Write a program to create child process. Make sure that parent process waits until child has not completed its execution. (use wait(), exit()) What will happen if parent process dies before child process? Illustrate it by creating one more child of parent process.**

**Source Code**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int main()

{

    pid\_t id1 = fork();

    if (id1 == -1)

    {

        printf("Fork Failed.\n");

        return 1;

    }

    else if (id1 == 0)

    {

        printf("Child Process 1\n");

        printf("Child PId: %d\n", getpid());

        printf("Files are:\n");

        execlp("ls", "ls", "-l", (char \*)NULL);

        exit(0);

    }

    else if (id1 > 0)

    {

        wait(NULL);

        printf("Parent Process\n");

        printf("First Child Completed\n");

        printf("Parent PId: %d\n", getpid());

        pid\_t id2 = fork();

        if (id2 < 0)

        {

            printf("Fork Failed\n");

            return 1;

        }

        else if (id2 == 0)

        {

            printf("Child Process 2\n");

            sleep(5);

            printf("Child PID: %d\n", getpid());

            exit(0);

        }

        else

        {

            printf("Parent Process\n");

            printf("Parent Pid: %d\n", getpid());

            exit(0);

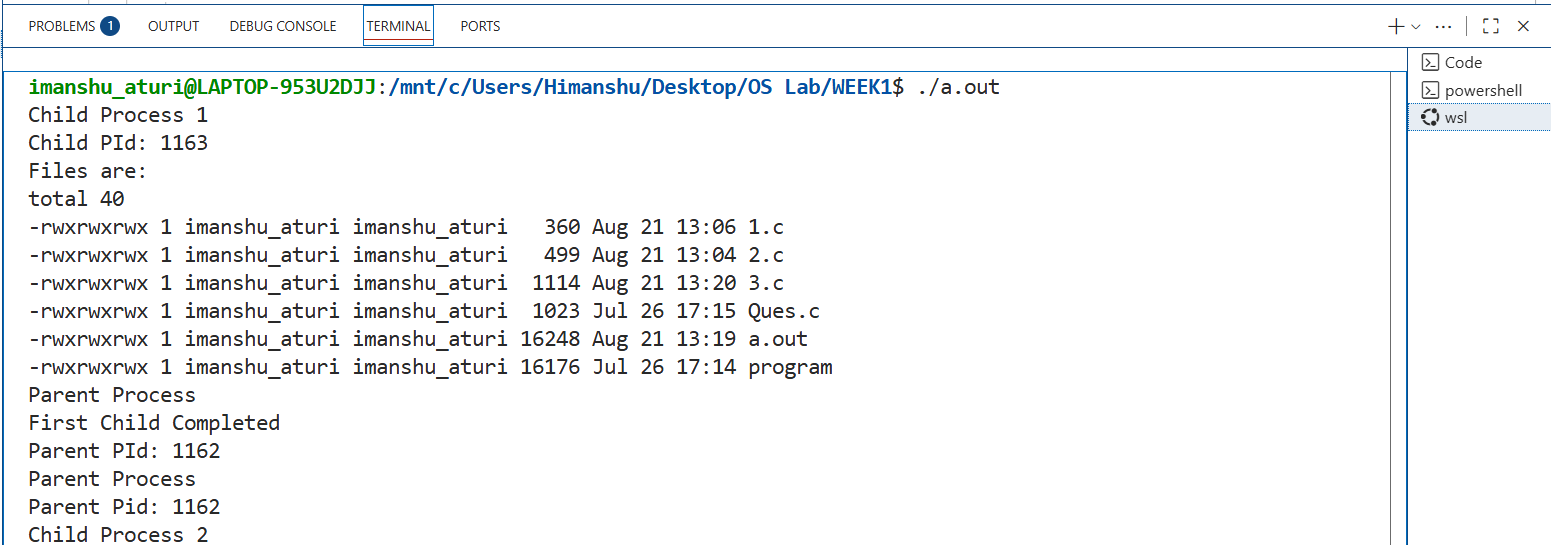
        }

    }

    return 0;

}

**OUTPUT**



**Program 4) Write a program to implement Orphan and Zombie Process.**

**Source Code**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main()

{

    int choice;

    printf("Enter Choice\n");

    printf("1. Orphan Process\n");

    printf("2. Zombie Process\n");

    scanf("%d", &choice);

    pid\_t pid;

    if (choice == 1)

    {

        printf("\nOrphan Process\n");

        pid = fork();

        if (pid < 0)

        {

            printf("Fork failed");

            exit(1);

        }

        else if (pid == 0)

        {

            printf("Orphan Child started. PID: %d, Parent PID: %d\n", getpid(), getppid());

            sleep(5);

            printf("Orphan Child still running. PID: %d, New Parent PID: %d \n",

                   getpid(), getppid());

            exit(0);

        }

        else

        {

            printf("Parent exiting, leaving child as orphan. PID: %d\n", getpid());

            exit(0);

        }

    }

    else if (choice == 2)

    {

        printf("\nZombie Process\n");

        pid = fork();

        if (pid < 0)

        {

            printf("Fork failed");

            exit(1);

        }

        else if (pid == 0)

        {

            printf("Zombie Child started. PID: %d\n", getpid());

            exit(0);

        }

        else

        {

            printf("Parent (PID: %d) sleeping, Child will become zombie.\n", getpid());

            sleep(20);

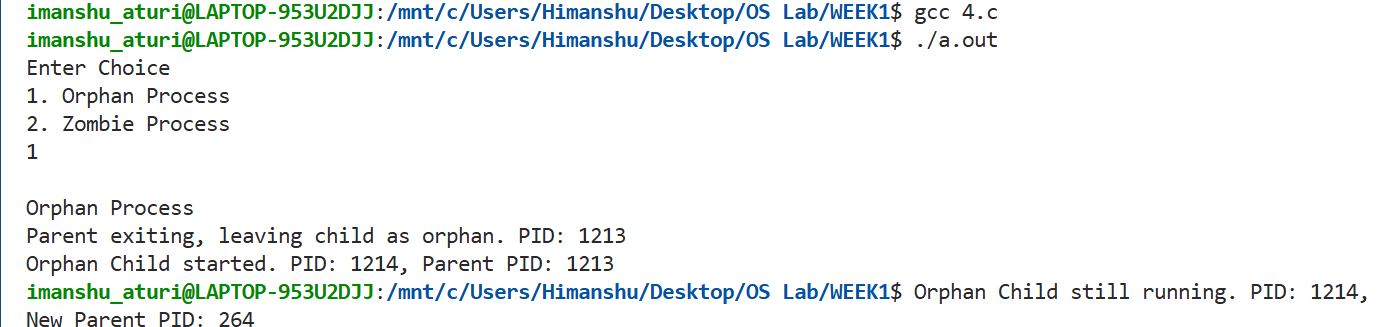
            printf("Parent exiting, zombie cleared.\n");

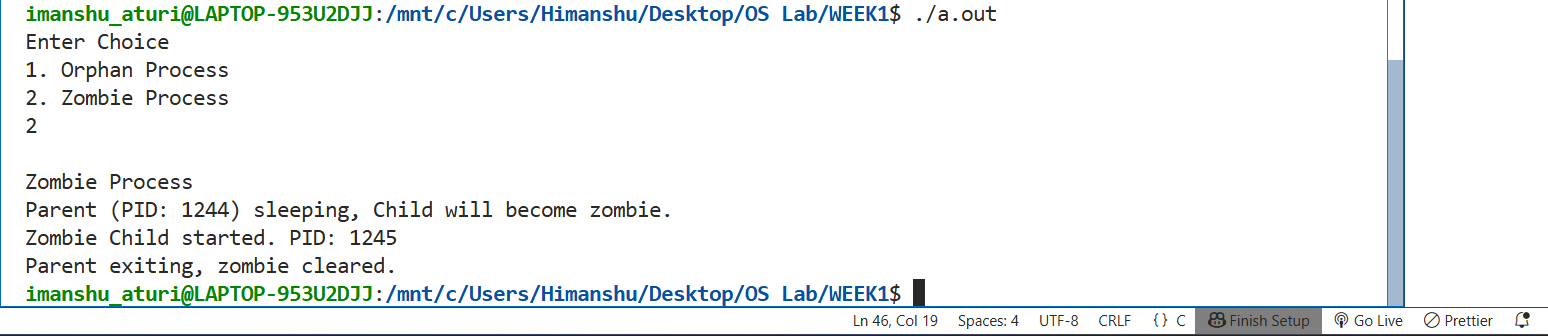
        }

    }

    return 0;

}

 **OUTPUT**



**WEEK 2**

**Program 1) Write a program to open a directory and list its contents.**

**Source Code**

#include <stdio.h>

#include <dirent.h>

int main()

{

    DIR \*dir;

    struct dirent \*entry;

    dir = opendir(".");

    if (dir == NULL)

    {

        printf("Error");

        return 1;

    }

    while ((entry = readdir(dir)) != NULL)

    {

        printf("%s\n", entry->d\_name);

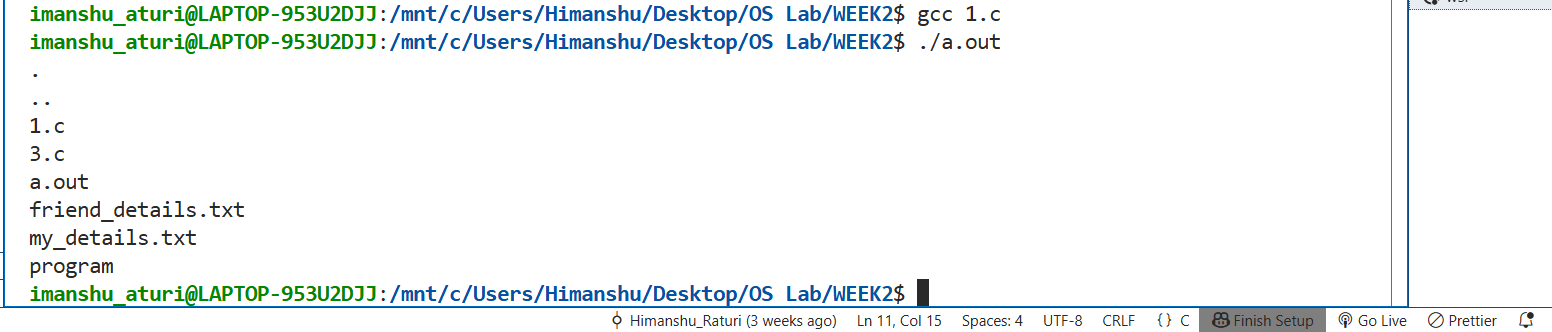
    }

    closedir(dir);

    return 0;

}

**OUTPUT**



**Program 2) Write a program to show working of execlp() system call by executing ls command.**

**Source Code**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

    if (execlp("ls", "ls", "-l", (char \*)NULL) == -1)

    {

        printf("execlp failed");

        exit(1);

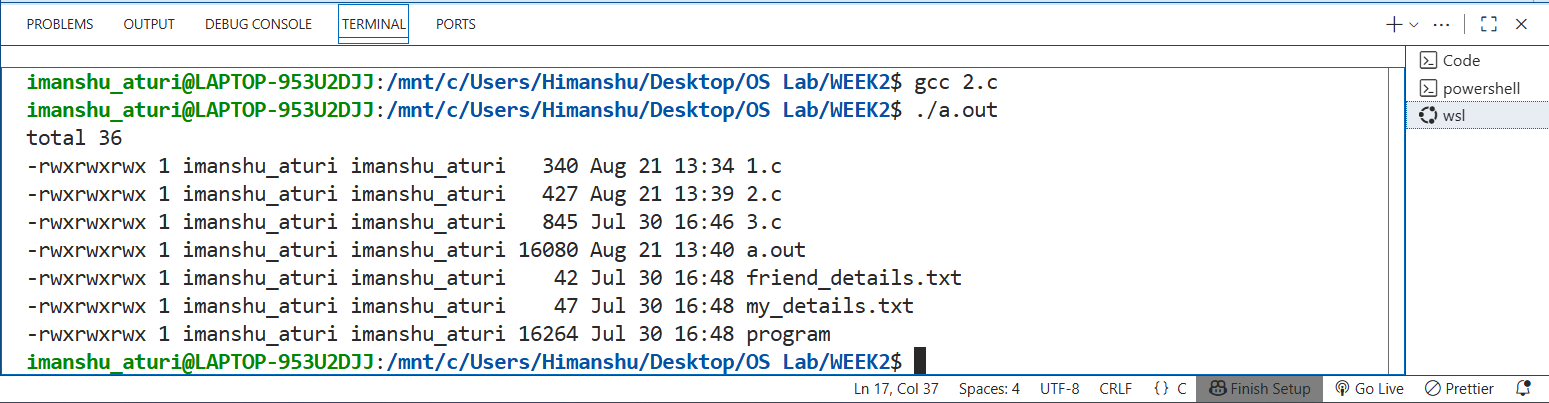
    }

    printf("After execlp() call\n");

    return 0;

}

**OUTPUT**



**Program 3) Write a program to read a file and store your details in that file. Your program should also create one more file and store your friends details in that file. Once both files are created, print lines which are matching in both files.**

**Source Code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void main()

{

    FILE \*file1, \*file2;

    char line1[100], line2[100];

    file1 = fopen("my\_details.txt", "w");

    fprintf(file1, "Name: Himanshu Raturi\nRoll: 31\nCity: Rishikesh\n");

    fclose(file1);

    file2 = fopen("friend\_details.txt", "w");

    fprintf(file2, "Name: Akhil Bhatt\nRoll: 30\nCity: Rishikesh\n");

    fclose(file2);

    file1 = fopen("my\_details.txt", "r");

    file2 = fopen("friend\_details.txt", "r");

    while (fgets(line1, sizeof(line1), file1))

    {

        rewind(file2);

        while (fgets(line2, sizeof(line2), file2))

        {

            if (strcmp(line1, line2) == 0)

                printf("Common line: %s", line1);

        }

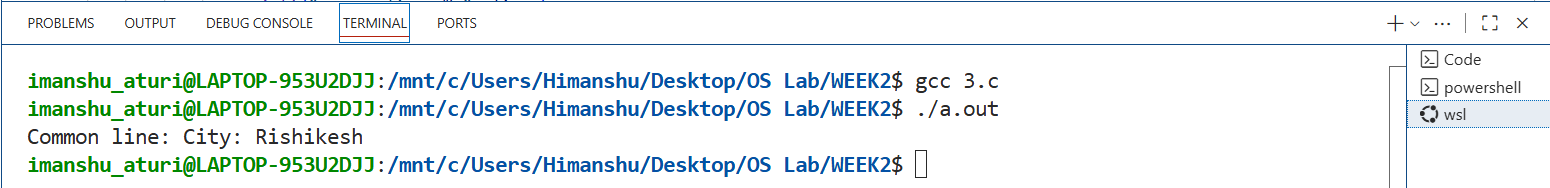
    }

    fclose(file1);

    fclose(file2);

}

**OUTPUT**







**WEEK 3**

**Program 1) Write a C program to implement FCFS algorithm.**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

typedef struct process

{

    int pid, arrival\_time, burst\_time, complete\_time, int start\_time, response\_time, turn\_around\_time , waiting\_time;

} process;

int compare(const void \*p1, const void \*p2)

{

    int a = ((process \*)p1)->arrival\_time;

    int b = ((process \*)p2)->arrival\_time;

    if (a < b)

        return -1;

    else if (a > b)

        return 1;

    else

        return 0;

}

int main()

{

    float swt = 0, stat = 0;

    int sbt = 0, n;

    float awt = 0, atat = 0, cu = 0 , throughput = 0;

    printf("Enter Number of Processes: ");

    scanf("%d", &n);

    process p[n];

    printf("Burst Time: ");

    for (int i = 0; i < n; i++)

    {

        scanf("%d", &p[i].burst\_time);

        p[i].pid = i + 1;

    }

    printf("Arrival Time: ");

    for (int i = 0; i < n; i++)

        scanf("%d", &p[i].arrival\_time);

    qsort(p, n, sizeof(process), compare);

    for (int i = 0; i < n; i++)

    {

        if (i == 0)

        {

            p[i].start\_time = p[i].arrival\_time;

        }

        else

        {

            if (p[i - 1].complete\_time > p[i].arrival\_time)

                p[i].start\_time = p[i - 1].complete\_time;

            else

                p[i].start\_time = p[i].arrival\_time;

        }

        p[i].complete\_time = p[i].burst\_time + p[i].start\_time;

        p[i].turn\_around\_time = p[i].complete\_time - p[i].arrival\_time;

        p[i].waiting\_time = p[i].turn\_around\_time - p[i].burst\_time;

        p[i].response\_time = p[i].start\_time - p[i].arrival\_time;

        swt += p[i].waiting\_time;

        sbt += p[i].burst\_time;

        stat += p[i].turn\_around\_time;

    }

    awt = swt / n;

    atat = stat / n;

    float max\_ct = p[0].complete\_time;

    for (int i = 1; i < n; i++)

    {

        if (p[i].complete\_time > max\_ct)

            max\_ct = p[i].complete\_time;

    }

    cu = ((float)sbt / max\_ct) \* 100;

    throughput = n / max\_ct;

    printf("\nPID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

    for (int i = 0; i < n; i++)

    {

        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

               p[i].pid, p[i].arrival\_time, p[i].burst\_time,

               p[i].start\_time, p[i].complete\_time,

               p[i].turn\_around\_time, p[i].waiting\_time, p[i].response\_time);

    }

    printf("\nGantt Chart : ");

    for (int i = 0; i < n; i++)

        printf("P%d ", p[i].pid - 1);

    printf("\n");

    printf("\nSum of Turn Around Time : %.2f\nAverage Turn Around Time : %.2f\n", stat, atat);

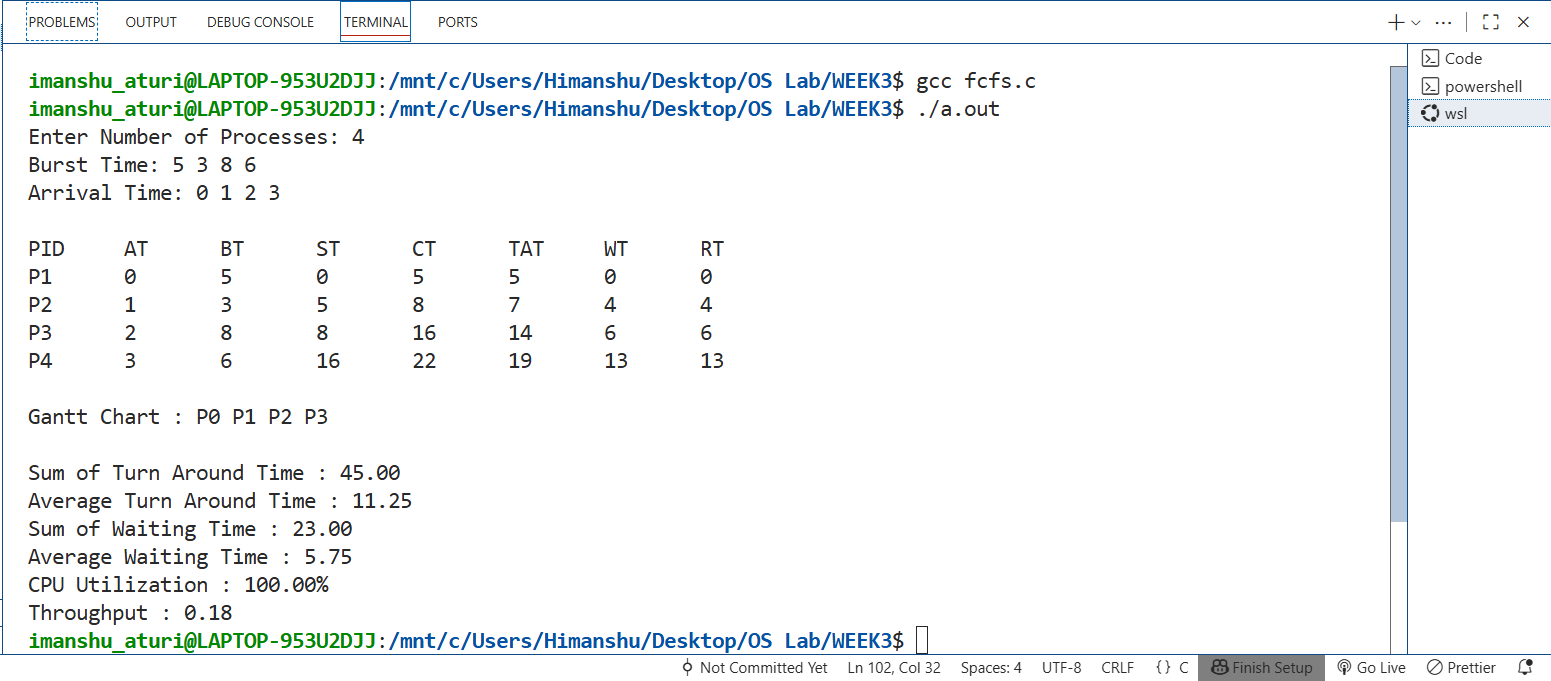
    printf("Sum of Waiting Time : %.2f\nAverage Waiting Time : %.2f\n", swt, awt);

    printf("CPU Utilization : %.2f%%\nThroughput : %.2f\n", cu, throughput);

    return 0;

}

**OUTPUT**

****

**Program 2) Write a C program to implement Shortest Job First Non-Preemptive algorithm.**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

typedef struct process

{

    int pid, arr-time, burst\_time, start\_time, complete\_time, turn\_ard\_time, wait\_time, response\_time, is\_completed;

} process;

void main()

{

    int n, sbt = 0;

    float swt = 0, stat = 0;

    float cu = 0, throughput = 0 , awt = 0 , atat = 0;

    printf("Number Of Process: ");

    scanf("%d", &n);

    process p[n];

    printf("\nBurst time: ");

    for (int i = 0; i < n; i++)

    {

        scanf("%d", &p[i].burst\_time);

        p[i].pid = i + 1;

        p[i].is\_completed = 0;

    }

    printf("\nArrival time: ");

    for (int i = 0; i < n; i++)

        scanf("%d", &p[i].arr\_time);

    int completed = 0, curr\_time = 0;

    float max\_completion\_time = 0;

    int gantt[n], g\_ind = 0;

    while (completed != n)

    {

        int min\_ind = -1;

        int min\_bt = INT\_MAX;

        for (int i = 0; i < n; i++)

        {

            if (p[i].arr\_time <= curr\_time && p[i].is\_completed == 0)

            {

                if (p[i].burst\_time < min\_bt)

                {

                    min\_bt = p[i].burst\_time;

                    min\_ind = i;

                }

                else if (p[i].burst\_time == min\_bt)

                {

                    if (p[i].arr\_time < p[min\_ind].arr\_time)

                        min\_ind = i;

                }

            }

        }

        if (min\_ind != -1)

        {

            p[min\_ind].start\_time = curr\_time;

            p[min\_ind].complete\_time = p[min\_ind].start\_time + p[min\_ind].burst\_time;

            p[min\_ind].turn\_ard\_time = p[min\_ind].complete\_time - p[min\_ind].arr\_time;

            p[min\_ind].wait\_time = p[min\_ind].turn\_ard\_time - p[min\_ind].burst\_time;

            p[min\_ind].response\_time = p[min\_ind].start\_time - p[min\_ind].arr\_time;

            swt += p[min\_ind].wait\_time;

            stat += p[min\_ind].turn\_ard\_time;

            sbt += p[min\_ind].burst\_time;

            if (p[min\_ind].complete\_time > max\_completion\_time)

                max\_completion\_time = p[min\_ind].complete\_time;

            gantt[g\_ind++] = p[min\_ind].pid;

            p[min\_ind].is\_completed = 1;

            completed++;

            curr\_time = p[min\_ind].complete\_time;

        }

        else

        {

            curr\_time++;

        }

    }

    awt = swt / n;

    atat = stat / n;

    cu = ((float)sbt / max\_completion\_time) \* 100;

    throughput = (float)n / max\_completion\_time;

    printf("\nPID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

    for (int i = 0; i < n; i++)

    {

        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

               p[i].pid, p[i].arr\_time, p[i].burst\_time,

               p[i].start\_time, p[i].complete\_time,

               p[i].turn\_ard\_time, p[i].wait\_time, p[i].response\_time);

    }

    printf("\nGantt Chart : ");

    for (int i = 0; i < n; i++)

        printf("P%d ", gantt[i]); // if you want P0, P1... instead of P1, P2...

    printf("\nTotal Turn Around Time : %.2f\nAverage Turn Around Time : %.2f\n", stat, atat);

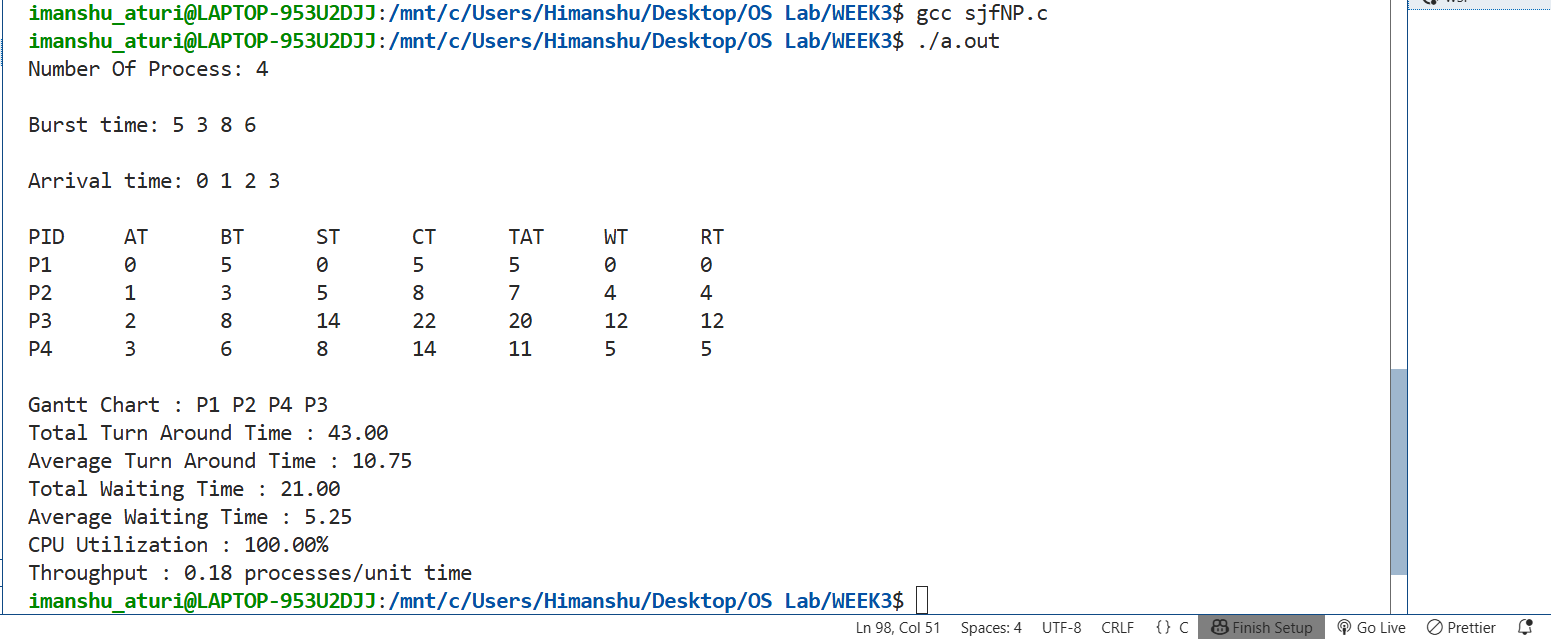
    printf("Total Waiting Time : %.2f\nAverage Waiting Time : %.2f\n", swt, awt);

    printf("CPU Utilization : %.2f%%\n", cu);

    printf("Throughput : %.2f processes/unit time\n", throughput);

}

**OUTPUT**



**Program 3) Write a C program to implement Shortest Job First Preemptive algorithm.**

**Source Code**

#include <stdio.h>

#include <limits.h>

typedef struct process

{

    int pid, arr\_time, burst\_time, rem\_time, start\_time, complete\_time, turn\_ard\_time,wait\_time, response\_time , is\_completed;

} process;

int main()

{

    int n;

    printf("Number of Processes: ");

    scanf("%d", &n);

    process p[n];

    int total\_bt =0 , completed = 0, curr\_time = 0;

    printf("\nBurst Time: ");

    for (int i = 0; i < n; i++)

    {

        scanf("%d", &p[i].burst\_time);

        p[i].pid = i + 1;

        p[i].rem\_time = p[i].burst\_time; // initially remaining = burst

        p[i].is\_completed = 0;

    }

    printf("\nArrival Time: ");

    for (int i = 0; i < n; i++)

        scanf("%d", &p[i].arr\_time);

    float total\_wt = 0, total\_tat = 0, total\_rt = 0;

    int gantt[100], g\_index = 0;

    while (completed != n)

    {

        int min\_ind = -1;

        int min\_rt = INT\_MAX;

        for (int i = 0; i < n; i++)

        {   if (p[i].arr\_time <= curr\_time && p[i].is\_completed == 0)

            {

                if (p[i].rem\_time < min\_rt)

                {

                    min\_rt = p[i].rem\_time;

                    min\_ind = i;

                }

                else if (p[i].rem\_time == min\_rt)

                {

                    if (p[i].arr\_time < p[min\_ind].arr\_time)

                        min\_ind = i;

                }

            }

        }

        if (min\_ind != -1)

        {

            if (p[min\_ind].rem\_time == p[min\_ind].burst\_time)

                p[min\_ind].start\_time = curr\_time;

            gantt[g\_index++] = p[min\_ind].pid;

            p[min\_ind].rem\_time--;

            curr\_time++;

            if (p[min\_ind].rem\_time == 0)

            {

                p[min\_ind].complete\_time = curr\_time;

                p[min\_ind].turn\_ard\_time = p[min\_ind].complete\_time - p[min\_ind].arr\_time;

                p[min\_ind].wait\_time = p[min\_ind].turn\_ard\_time - p[min\_ind].burst\_time;

                p[min\_ind].response\_time = p[min\_ind].start\_time - p[min\_ind].arr\_time;

                total\_wt += p[min\_ind].wait\_time;

                total\_tat += p[min\_ind].turn\_ard\_time;

                total\_rt += p[min\_ind].response\_time;

                p[min\_ind].is\_completed = 1;

                completed++;

            }

        }

        else

        {

            curr\_time++;

        }

    }

    printf("\nPID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

    for (int i = 0; i < n; i++)

    {

        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

               p[i].pid, p[i].arr\_time, p[i].burst\_time,

               p[i].start\_time, p[i].complete\_time,

               p[i].turn\_ard\_time, p[i].wait\_time, p[i].response\_time);

    }

    printf("\nGantt Chart : ");

    printf("P%d ", gantt[0]);

    for (int i = 1; i < g\_index; i++)

    {

        if (gantt[i] != gantt[i - 1])

            printf("P%d ", gantt[i]);

    }

    printf("\nAverage Turn Around Time: %.2f", total\_tat / n);

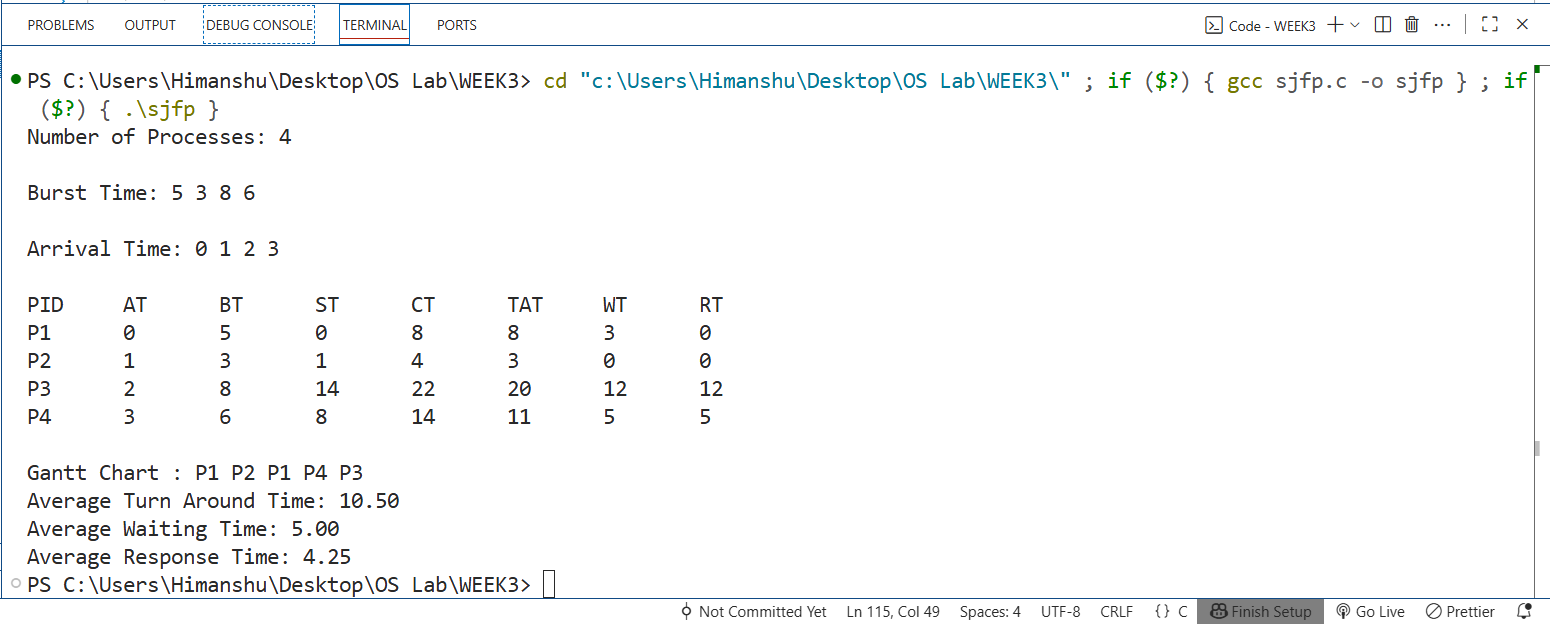
    printf("\nAverage Waiting Time: %.2f", total\_wt / n);

    printf("\nAverage Response Time: %.2f\n", total\_rt / n);

    return 0;

}

**OUTPUT**



**WEEK 4**

**Program 1) Write a C program to implement Priority Scheduling(higher the number higher its priority).**

**Source Code**

#include <stdio.h>

#include <limits.h>

typedef struct process

{

    int pid, arr\_time, burst\_time, rem\_time, priority, start\_time, complete\_time, turn\_ard\_time, wait\_time, response\_time, is\_completed;

} process;

void main()

{

    int n, completed = 0, curr\_time = 0 , gantt[1000], g\_index = 0;

    printf("Enter Number of Processes: ");

    scanf("%d", &n);

    process p[n];

    printf("\nBurst time: ");

    for (int i = 0; i < n; i++)

    {

        scanf("%d", &p[i].burst\_time);

        p[i].pid = i;

        p[i].rem\_time = p[i].burst\_time;

        p[i].is\_completed = 0;

    }

    printf("\nArrival time: ");

    for (int i = 0; i < n; i++)

        scanf("%d", &p[i].arr\_time);

    printf("\nPriority (higher number = higher priority): ");

    for (int i = 0; i < n; i++)

        scanf("%d", &p[i].priority);

    float total\_wt = 0, total\_tat = 0, total\_rt = 0;

    while (completed != n)

    {

        int idx = -1;

        int highest\_priority = INT\_MIN;

        for (int i = 0; i < n; i++)

        {

            if (p[i].arr\_time <= curr\_time && p[i].is\_completed == 0)

            {

                if (p[i].priority > highest\_priority)

                {

                    highest\_priority = p[i].priority;

                    idx = i;

                }

                else if (p[i].priority == highest\_priority)

                {

                    if (p[i].arr\_time < p[idx].arr\_time)

                        idx = i;

                }

            }

        }

        if (idx != -1)

        {

            if (p[idx].rem\_time == p[idx].burst\_time)

                p[idx].start\_time = curr\_time; // first execution

            gantt[g\_index++] = p[idx].pid;

            p[idx].rem\_time--;

            curr\_time++;

            if (p[idx].rem\_time == 0)

            {

                p[idx].complete\_time = curr\_time;

                p[idx].turn\_ard\_time = p[idx].complete\_time - p[idx].arr\_time;

                p[idx].wait\_time = p[idx].turn\_ard\_time - p[idx].burst\_time;

                p[idx].response\_time = p[idx].start\_time - p[idx].arr\_time;

                total\_wt += p[idx].wait\_time;

                total\_tat += p[idx].turn\_ard\_time;

                total\_rt += p[idx].response\_time;

                p[idx].is\_completed = 1;

                completed++;

            }

        }

        else

            curr\_time++;

    }

    printf("\nPID\tAT\tBT\tPRI\tST\tCT\tTAT\tWT\tRT\n");

    for (int i = 0; i < n; i++)

    {

        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

               p[i].pid, p[i].arr\_time, p[i].burst\_time, p[i].priority,

               p[i].start\_time, p[i].complete\_time,

               p[i].turn\_ard\_time, p[i].wait\_time, p[i].response\_time);

    }

    printf("\nGantt Chart : ");

    printf("P%d ", gantt[0]);

    for (int i = 1; i < g\_index; i++)

    {

        if (gantt[i] != gantt[i - 1])

            printf("P%d ", gantt[i]);

    }

    printf("\n\nAverage Waiting Time: %.2f", total\_wt / n);

    printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n);

   }

**OUTPUT**

