**OPERRATING SYSTEM ASSIGNMENT**

**Q1 :- HRRN (Highest Response Ratio Next) Scheduling**

* **HRRN is a non-preemptive CPU scheduling algorithm.**
* **It selects the process with the highest response ratio (HRR).**
* **Response Ratio Formula:Response**

**Ratio = Waiting Time+Burst Time/Burst Time**

**Task to be Done:**

* **Write a C/C++ program to simulate HRRN and compute AWT & ATAT.**
* **Input: Number of processes, arrival time and burst time.**
* **Output: Execution order, AWT, ATAT.**

**CODE:-**

#include <stdio.h>

typedef struct {

    int id, arrival\_time, burst\_time, waiting\_time, turnaround\_time, completed;

    float response\_ratio;

} Process;

void sortByArrival(Process p[], int n) {

    Process temp;

    for (int i = 0; i < n - 1; i++) {

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

            if (p[j].arrival\_time > p[j + 1].arrival\_time) {

                temp = p[j];

                p[j] = p[j + 1];

                p[j + 1] = temp;

            }

        }

    }

}

int main() {

    int n, time = 0, completed = 0;

    float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

    printf("Enter the number of processes: ");

    scanf("%d", &n);

    Process p[n];

    printf("\nEnter process details (Arrival Time & Burst Time):\n");

    for (int i = 0; i < n; i++) {

        printf("Process P%d Arrival & Burst Time: ", i);

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

        p[i].id = i;

        p[i].waiting\_time = 0;

        p[i].turnaround\_time = 0;

        p[i].completed = 0;

    }

    sortByArrival(p, n);

    printf("\nExecution Order: ");

    while (completed < n) {

        int highest\_response\_index = -1;

        float max\_response\_ratio = -1;

        for (int i = 0; i < n; i++) {

            if (p[i].arrival\_time <= time && !p[i].completed) {

                p[i].waiting\_time = time - p[i].arrival\_time;

                p[i].response\_ratio = (float)(p[i].waiting\_time + p[i].burst\_time) / p[i].burst\_time;

                if (p[i].response\_ratio > max\_response\_ratio) {

                    max\_response\_ratio = p[i].response\_ratio;

                    highest\_response\_index = i;

                }

            }

        }

        if (highest\_response\_index == -1) {

            time++;

            continue;

        }

        int idx = highest\_response\_index;

        printf("P%d -> ", p[idx].id);

        time += p[idx].burst\_time;

        p[idx].turnaround\_time = time - p[idx].arrival\_time;

        p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

        p[idx].completed = 1;

        avg\_waiting\_time += p[idx].waiting\_time;

        avg\_turnaround\_time += p[idx].turnaround\_time;

        completed++;

    }

    avg\_waiting\_time /= n;

    avg\_turnaround\_time /= n;

    printf("\n\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

    for (int i = 0; i < n; i++) {

        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].arrival\_time, p[i].burst\_time, p[i].waiting\_time, p[i].turnaround\_time);

    }

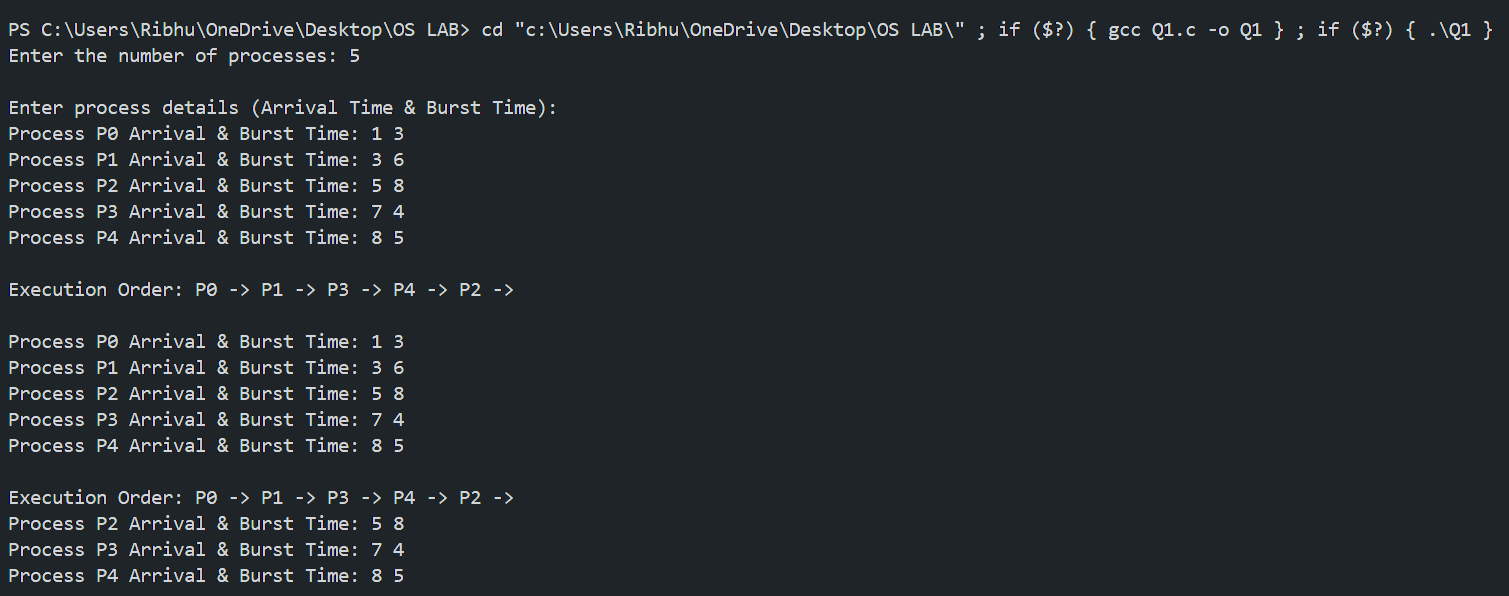
    printf("\nAverage Waiting Time: %.2f", avg\_waiting\_time);

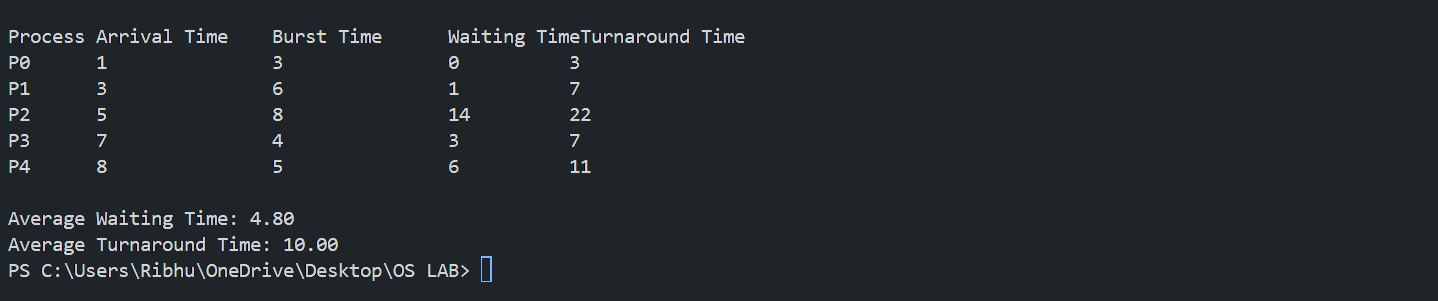
    printf("\nAverage Turnaround Time: %.2f\n", avg\_turnaround\_time);

    return 0;

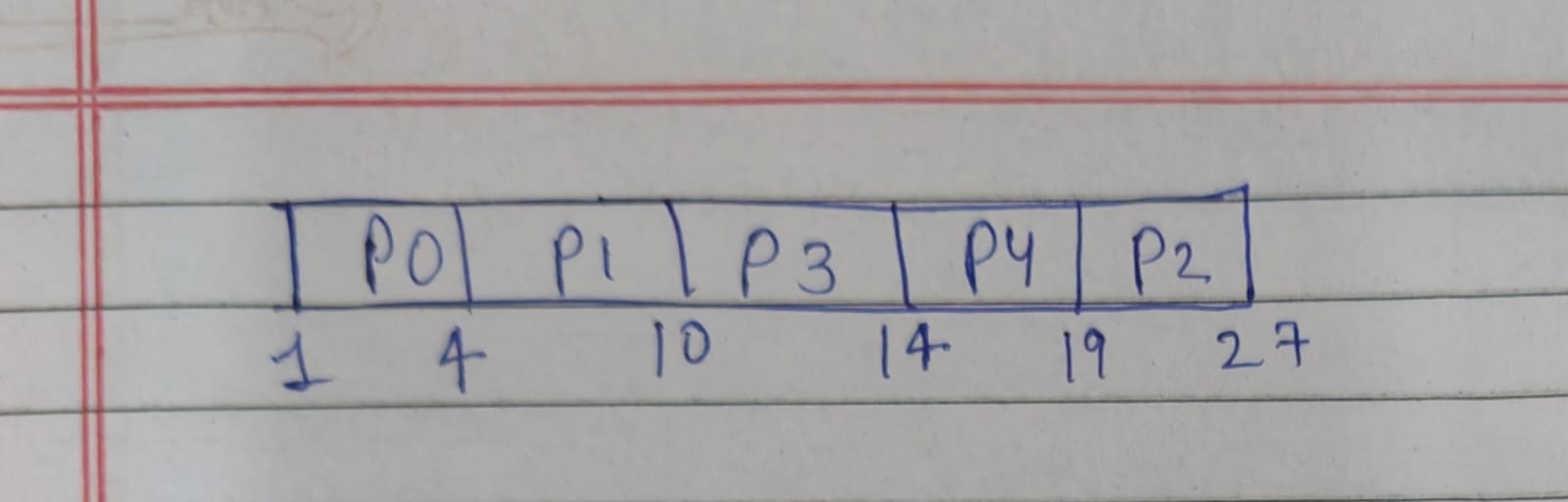
}

**OUTPUT:-**





**GHANTT CHART:-**



**Q2:- Multilevel Feedback Queue (MLFQ) Scheduling**

* **3 Queues with Different Scheduling Algorithms:  
  Queue 1 → Round Robin (Time Quantum = 4)  
  Queue 2 → Round Robin (Time Quantum = 8)  
  Queue 3 → First-Come, First-Serve (FCFS)**

**Task to be Done:**

* **Write a C/C++ program to simulate this MLFQ setup.**
* **Process moves down queues if not completed within a quantum.**
* **Input: Number of processes, arrival time, burst time.**
* **Output: Scheduling order, AWT, ATAT.**

**CODE:-** #include <stdio.h>

#include <stdlib.h>

#define MAX\_PROCESSES 10

#define TIME\_QUANTUM\_1 4

#define TIME\_QUANTUM\_2 8

typedef struct {

    int id, arrival\_time, burst\_time, remaining\_time, waiting\_time, turnaround\_time, queue\_level;

} Process;

void roundRobin(Process processes[], int n, int time\_quantum, int queue\_level) {

    int time = 0, completed = 0;

    while (completed < n) {

        int all\_done = 1;

        for (int i = 0; i < n; i++) {

            if (processes[i].queue\_level == queue\_level && processes[i].remaining\_time > 0) {

                all\_done = 0;

                if (processes[i].remaining\_time > time\_quantum) {

                    time += time\_quantum;

                    processes[i].remaining\_time -= time\_quantum;

                    printf("P%d -> ", processes[i].id);

                    processes[i].queue\_level++;

                } else {

                    time += processes[i].remaining\_time;

                    processes[i].waiting\_time = time - processes[i].arrival\_time - processes[i].burst\_time;

                    processes[i].turnaround\_time = time - processes[i].arrival\_time;

                    processes[i].remaining\_time = 0;

                    printf("P%d (Done) -> ", processes[i].id);

                    completed++;

                }

            }

        }

        if (all\_done) break;

    }

}

void fcfs(Process processes[], int n) {

    int time = 0;

    for (int i = 0; i < n; i++) {

        if (processes[i].queue\_level == 3 && processes[i].remaining\_time > 0) {

            time += processes[i].remaining\_time;

            processes[i].waiting\_time = time - processes[i].arrival\_time - processes[i].burst\_time;

            processes[i].turnaround\_time = time - processes[i].arrival\_time;

            processes[i].remaining\_time = 0;

            printf("P%d (Done) -> ", processes[i].id);

        }

    }

}

int main() {

    int n;

    printf("Enter the number of processes: ");

    scanf("%d", &n);

    Process processes[MAX\_PROCESSES];

    printf("\nEnter process details (Arrival Time & Burst Time):\n");

    for (int i = 0; i < n; i++) {

        printf("Process P%d Arrival & Burst Time: ", i);

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

        processes[i].id = i;

        processes[i].remaining\_time = processes[i].burst\_time;

        processes[i].waiting\_time = 0;

        processes[i].turnaround\_time = 0;

        processes[i].queue\_level = 1;

    }

    printf("\nExecution Order: ");

    roundRobin(processes, n, TIME\_QUANTUM\_1, 1);

    roundRobin(processes, n, TIME\_QUANTUM\_2, 2);

    fcfs(processes, n);

    printf("\n\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

    float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

    for (int i = 0; i < n; i++) {

        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival\_time, processes[i].burst\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

        avg\_waiting\_time += processes[i].waiting\_time;

        avg\_turnaround\_time += processes[i].turnaround\_time;

    }

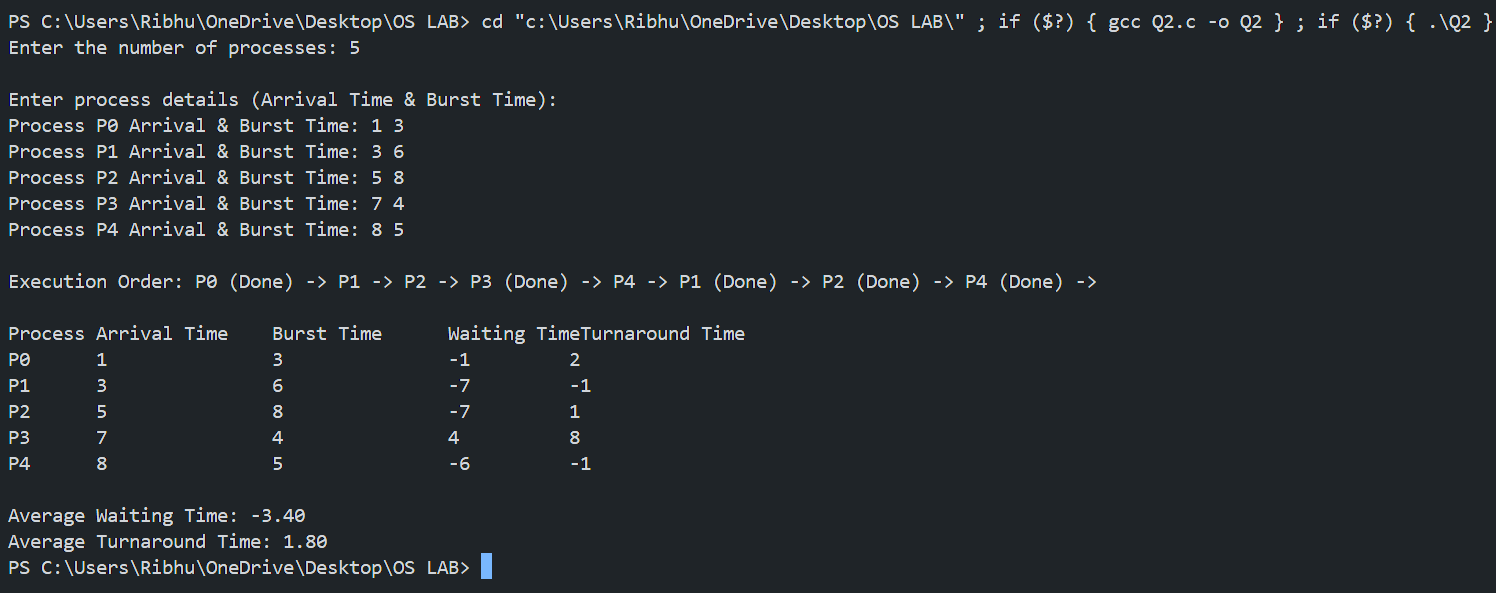
    printf("\nAverage Waiting Time: %.2f", avg\_waiting\_time / n);

    printf("\nAverage Turnaround Time: %.2f\n", avg\_turnaround\_time / n);

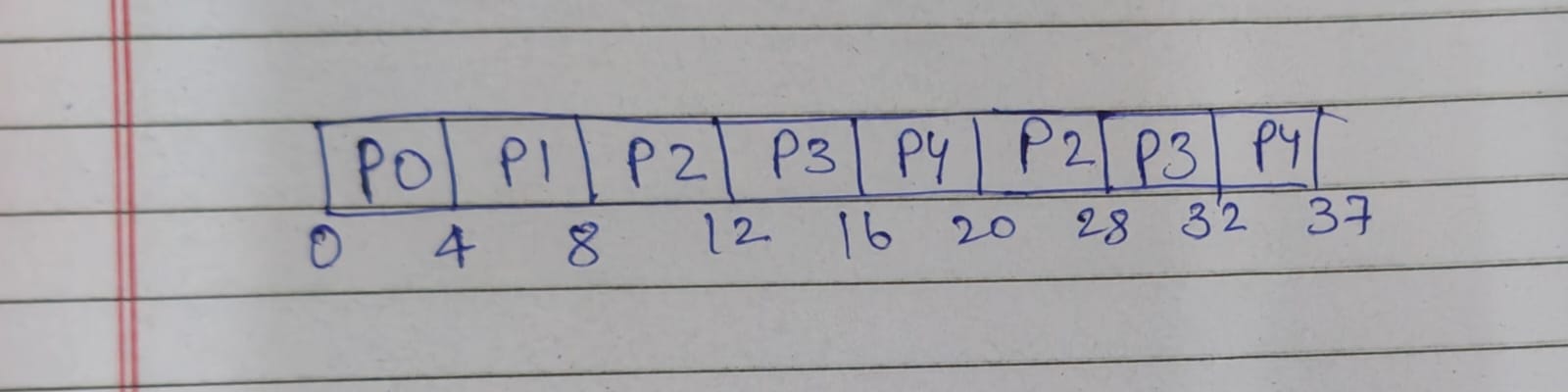
    return 0;

}

**OUTPUT:-**



**GHANTT CHART:-**



Q3:- Multilevel Queue (MLQ) Scheduling

* Predefined Queue Allocation  
  Queue 1 → P0, P1, P4  
  Queue 2 → P2, P3
* Each queue follows different scheduling:  
  Queue 1 → Higher Priority (Executes First)  
  Queue 2 → Lower Priority (Executes After Queue 1 is Empty)

Task to be Done:

* Write a C/C++ program to simulate MLQ scheduling.
* Input: Number of processes, arrival time, burst time.
* Output: Scheduling order, AWT, ATAT.

**CODE:-** #include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct {

    int id, arrival\_time, burst\_time, waiting\_time, turnaround\_time;

} Process;

void fcfs(Process processes[], int n, int \*current\_time) {

    for (int i = 0; i < n; i++) {

        if (\*current\_time < processes[i].arrival\_time) {

            \*current\_time = processes[i].arrival\_time;

        }

        processes[i].waiting\_time = \*current\_time - processes[i].arrival\_time;

        processes[i].turnaround\_time = processes[i].waiting\_time + processes[i].burst\_time;

        \*current\_time += processes[i].burst\_time;

        printf("P%d -> ", processes[i].id);

    }

}

int main() {

    int n;

    printf("Enter number of processes: ");

    scanf("%d", &n);

    Process queue1[MAX\_PROCESSES], queue2[MAX\_PROCESSES];

    int n1 = 0, n2 = 0;

    printf("\nEnter process details:\n");

    for (int i = 0; i < n; i++) {

        int id, arrival, burst;

        printf("Process P%d Arrival Time & Burst Time: ", i);

        scanf("%d %d", &arrival, &burst);

        if (i == 0 || i == 1 || i == 4) {

            queue1[n1++] = (Process){i, arrival, burst, 0, 0};

        } else {

            queue2[n2++] = (Process){i, arrival, burst, 0, 0};

        }

    }

    int current\_time = 0;

    printf("\nExecution Order: ");

    fcfs(queue1, n1, &current\_time);

    fcfs(queue2, n2, &current\_time);

    printf("Done\n");

    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

    float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

    for (int i = 0; i < n1; i++) {

        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", queue1[i].id, queue1[i].arrival\_time, queue1[i].burst\_time, queue1[i].waiting\_time, queue1[i].turnaround\_time);

        avg\_waiting\_time += queue1[i].waiting\_time;

        avg\_turnaround\_time += queue1[i].turnaround\_time;

    }

    for (int i = 0; i < n2; i++) {

        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", queue2[i].id, queue2[i].arrival\_time, queue2[i].burst\_time, queue2[i].waiting\_time, queue2[i].turnaround\_time);

        avg\_waiting\_time += queue2[i].waiting\_time;

        avg\_turnaround\_time += queue2[i].turnaround\_time;

    }

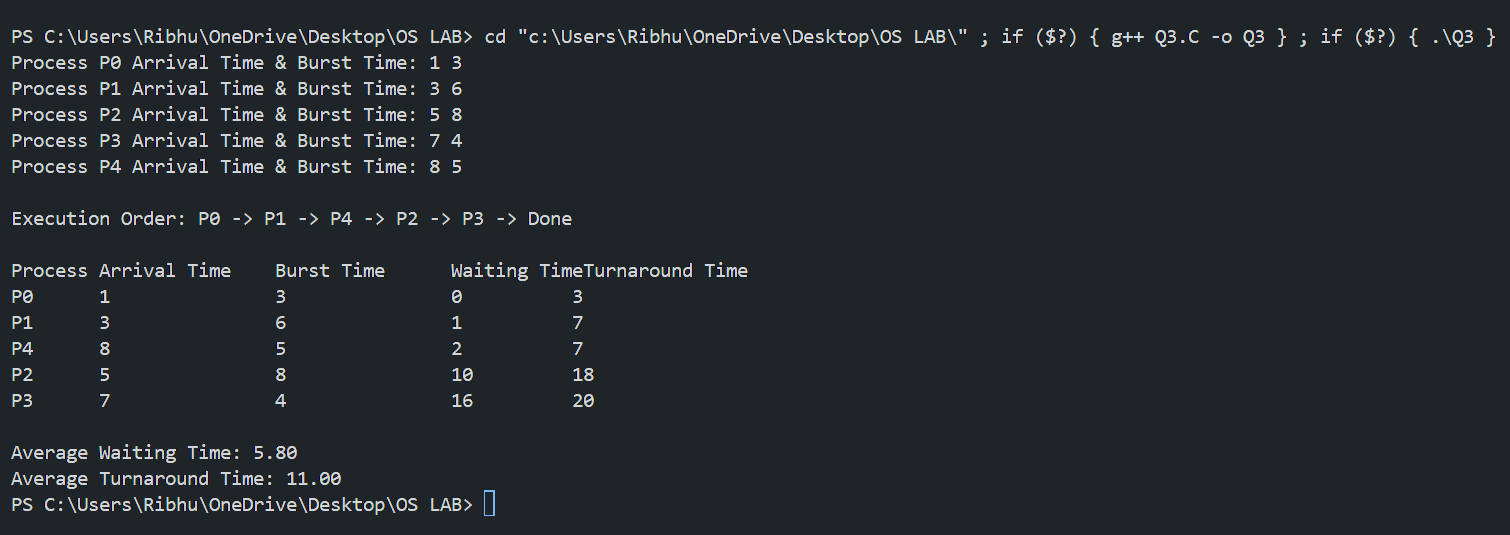
    printf("\nAverage Waiting Time: %.2f", avg\_waiting\_time / n);

    printf("\nAverage Turnaround Time: %.2f\n", avg\_turnaround\_time / n);

    return 0;

}

**OUTPUT:-**

****

**GHANTT CHART:-**

