Lab 3

Aim: Create simple programs implementing RR and Priority scheduling using same and different ATs.

Theory:

Round Robin (RR) and Priority scheduling are two commonly used scheduling algorithms in operating systems.

Round Robin Scheduling:

Round Robin scheduling is a pre-emptive scheduling algorithm that allocates CPU time slices to each process in a queue. The CPU is allocated to each process for a fixed time slice, also known as a time quantum. If a process does not complete within its allocated time, the CPU is preempted and given to the next process in the queue. This continues until each process has been given an equal share of the CPU time. RR scheduling is simple and easy to implement, but can lead to starvation (a situation where a low priority process may never get the CPU time it needs to complete).

Priority Scheduling:

Priority scheduling is a scheduling algorithm that allocates CPU time to processes based on their priority. Processes with a higher priority are given CPU time before processes with a lower priority. If two processes have the same priority, then RR scheduling is used to allocate the CPU time. Priority scheduling can prevent starvation, as long as all processes have a unique priority. If two processes have the same priority, then they may end up in a deadlock, where both are waiting for the other to release the CPU.

In summary, both RR and Priority scheduling algorithms have their own advantages and disadvantages. RR scheduling is easy to implement, but can lead to starvation. Priority scheduling can prevent starvation, but can result in deadlocks if two processes have the same priority. The choice of scheduling algorithm depends on the requirements of the operating system and the specific needs of the system.

Code for RR:

```
#include<stdio.h>

int main()
{
   int n,i,j,time_quantum,total=0,x,counter=0;
   int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
   printf("Enter Total Process:\t ");
   scanf("%d",&n);
   x=n;
   for(i=0;i<n;i++)
```

```
{
   printf("Enter Arrival Time and Burst Time for Process Process Number %d :",i+1);
   scanf("%d",&at[i]);
   scanf("%d",&bt[i]);
   rt[i]=bt[i];
}
printf("Enter Time Quantum:\t");
scanf("%d",&time_quantum);
printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
for(total=0,i=0;x!=0;)
{
   if(rt[i]<=time_quantum && rt[i]>0)
   {
      total=total+rt[i];
      rt[i]=0;
      counter=1;
   }
   else if(rt[i]>0)
   {
      rt[i]=rt[i]-time_quantum;
      total=total+time_quantum;
   }
   if(rt[i]==0 && counter==1)
   {
      x--;
      printf("P[%d]\t|\t%d\t|\t%d\n",i+1,total-at[i],total-at[i]-bt[i]);
      wait_time=wait_time+total-at[i]-bt[i];
      turnaround_time=turnaround_time+total-at[i];
      counter=0;
   }
   if(i==n-1)
      i=0;
   else if(at[i+1]<=total)
      i++;
   else
      i=0;
}
printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
printf("\nAvg Turnaround Time = %f\n",turnaround_time*1.0/n);
return 0;
```
 }

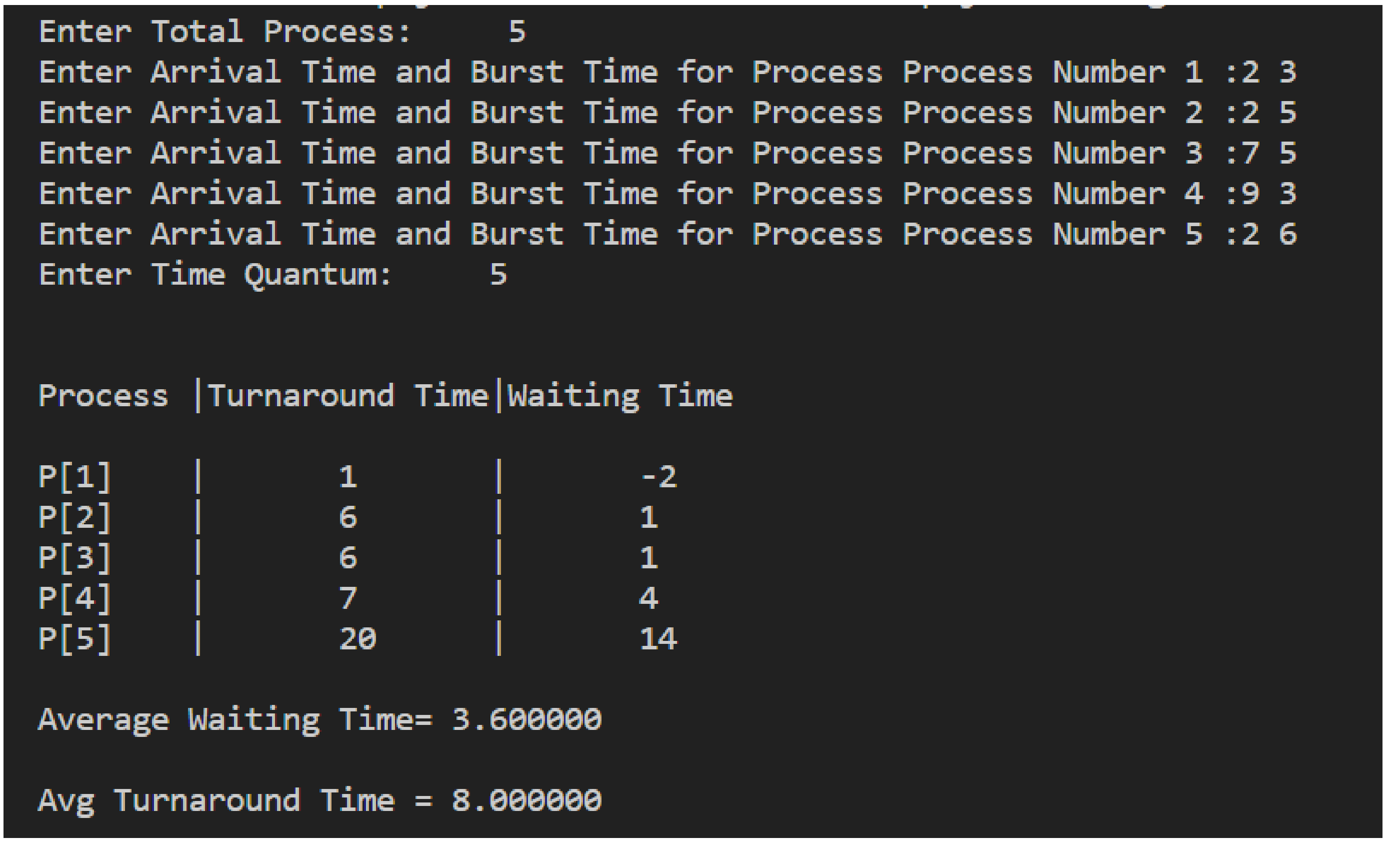
<u>ScreenShot</u>:-

INPUT:-

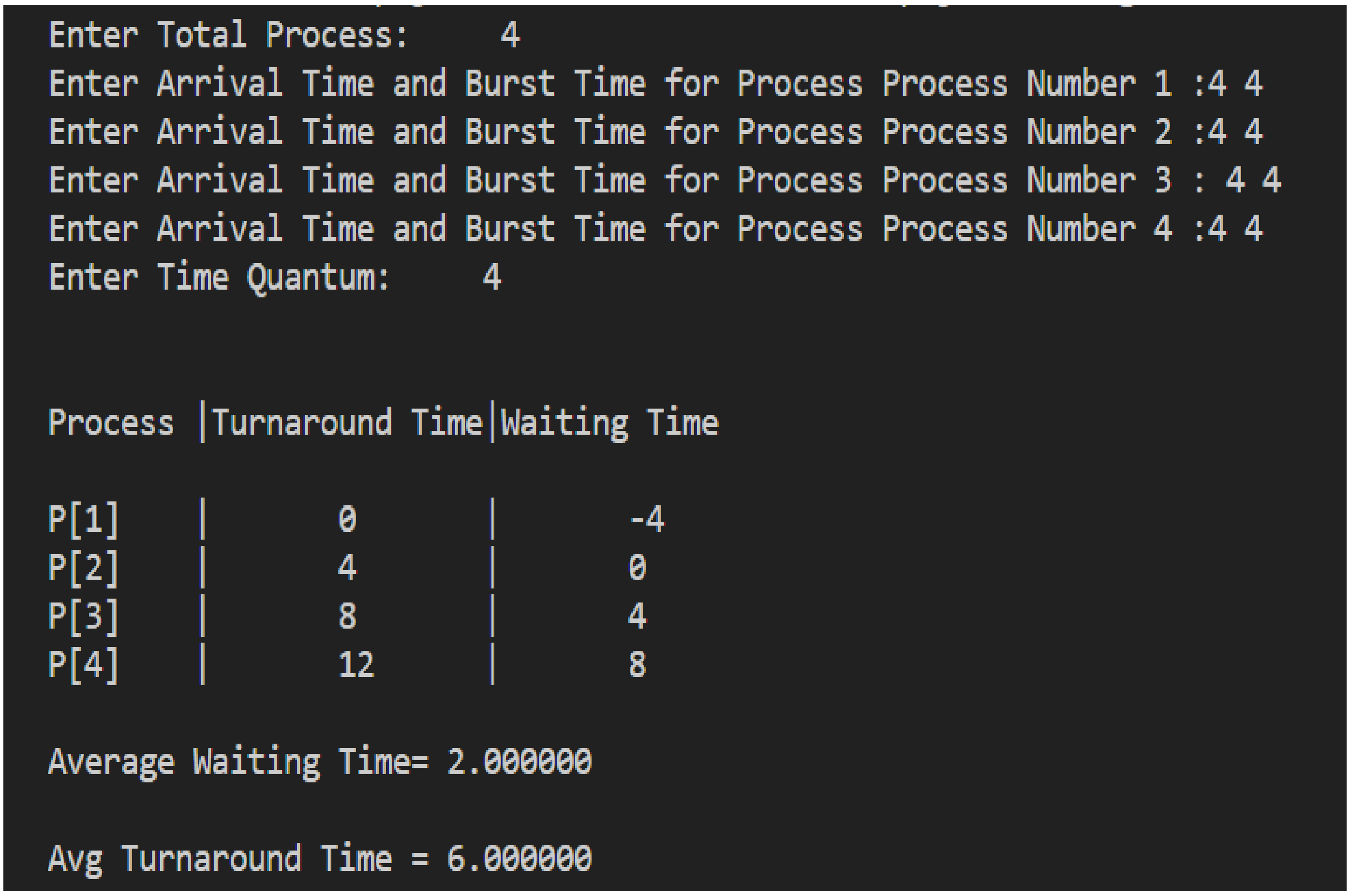
```
#include<stdio.h>
```

```c
int main()
{
    int n,i,j,time_quantum,total=0,x,counter=0;
    int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
    printf("Enter Total Process:\t ");
    scanf("%d",&n);
    x=n;
    for(i=0;i<n;i++)
    {
        printf("Enter Arrival Time and Burst Time for Process Process Number %d :",i+1);
        scanf("%d",&at[i]);
        scanf("%d",&bt[i]);
        rt[i]=bt[i];
    }
    printf("Enter Time Quantum:\t");
    scanf("%d",&time_quantum);
    printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
    for(total=0,i=0;x!=0;)
    {
        if(rt[i]<=time_quantum && rt[i]>0)
        {
            total=total+rt[i];
            rt[i]=0;
            counter=1;
        }
        else if(rt[i]>0)
        {
            rt[i]=rt[i]-time_quantum;
            total=total+time_quantum;
        }
        if(rt[i]==0 && counter==1)
        {
            x--;
            printf("P[%d]\t|\t%d\t|\t%d\n",i+1,total-at[i],total-at[i]-bt[i]);
            wait_time=wait_time+total-at[i]-bt[i];
            turnaround_time=turnaround_time+total-at[i];
            counter=0;
        }
        if(i==n-1)
            i=0;
        else if(at[i+1]<=total)
            i++;
        else
            i=0;
    }
    printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
    printf("\nAvg Turnaround Time = %f\n",turnaround_time*1.0/n);
    return 0;
}
```

OUTPUT:-



```
Enter Total Process:     5
Enter Arrival Time and Burst Time for Process Process Number 1 :2 3
Enter Arrival Time and Burst Time for Process Process Number 2 :2 5
Enter Arrival Time and Burst Time for Process Process Number 3 :7 5
Enter Arrival Time and Burst Time for Process Process Number 4 :9 3
Enter Arrival Time and Burst Time for Process Process Number 5 :2 6
Enter Time Quantum:     5


Process |Turnaround Time|Waiting Time

P[1]    |       1       |       -2
P[2]    |       6       |       1
P[3]    |       6       |       1
P[4]    |       7       |       4
P[5]    |       20      |       14

Average Waiting Time= 3.600000

Avg Turnaround Time = 8.000000
```



```
Enter Total Process:     4
Enter Arrival Time and Burst Time for Process Process Number 1 :4 4
Enter Arrival Time and Burst Time for Process Process Number 2 :4 4
Enter Arrival Time and Burst Time for Process Process Number 3 : 4 4
Enter Arrival Time and Burst Time for Process Process Number 4 :4 4
Enter Time Quantum:     4


Process |Turnaround Time|Waiting Time

P[1]    |       0       |       -4
P[2]    |       4       |       0
P[3]    |       8       |       4
P[4]    |       12      |       8

Average Waiting Time= 2.000000

Avg Turnaround Time = 6.000000
```

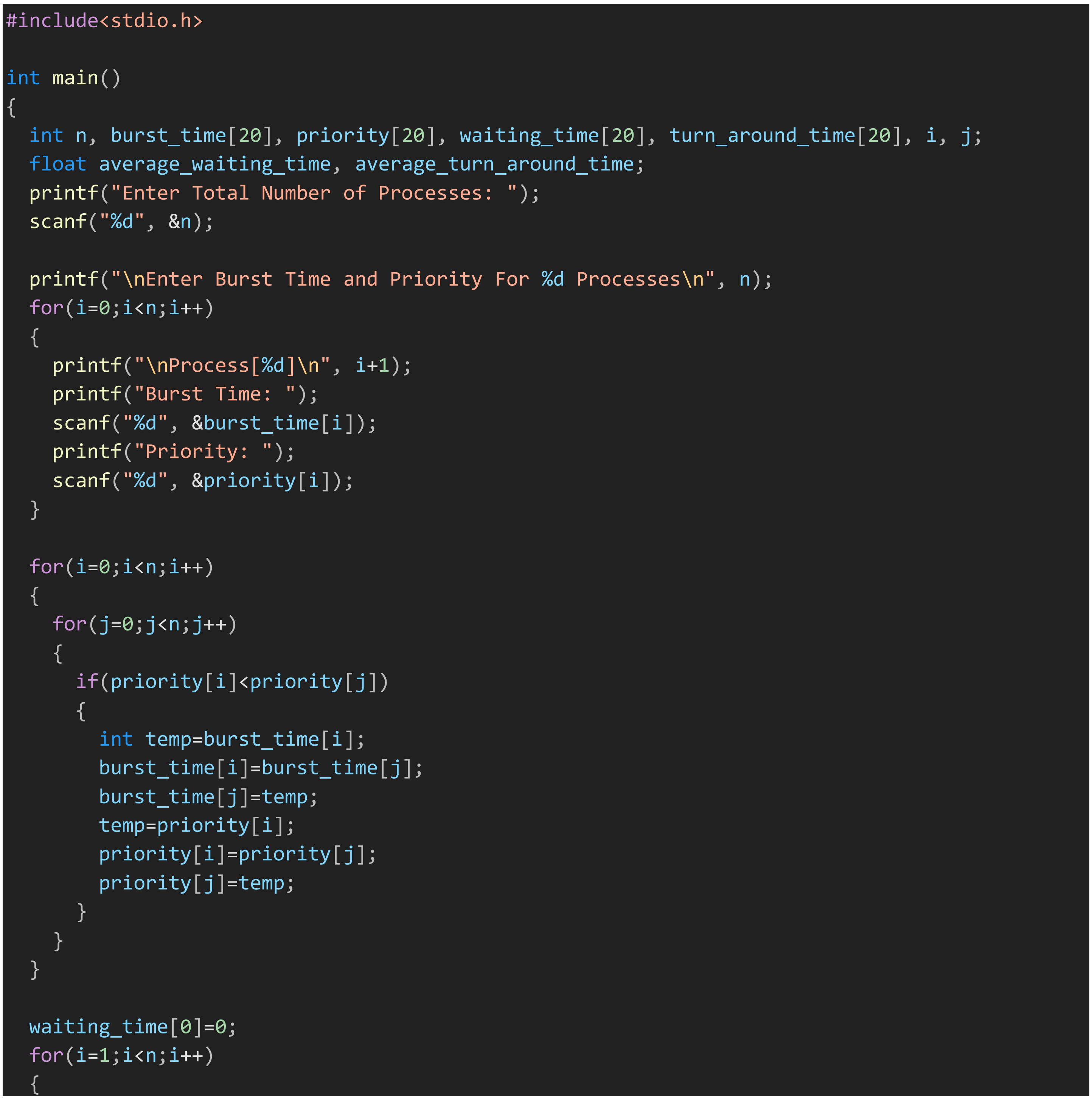
Code for Priority scheduling :

```
#include<stdio.h>

int main()
{
 int n, burst_time[20], priority[20], waiting_time[20], turn_around_time[20], i, j;
 float average_waiting_time, average_turn_around_time;
 printf("Enter Total Number of Processes: ");
 scanf("%d", &n);

 printf("\nEnter Burst Time and Priority For %d Processes\n", n);
 for(i=0;i<n;i++)
 {
  printf("\nProcess[%d]\n", i+1);
  printf("Burst Time: ");
  scanf("%d", &burst_time[i]);
  printf("Priority: ");
  scanf("%d", &priority[i]);
 }

 for(i=0;i<n;i++)
 {
  for(j=0;j<n;j++)
  {
   if(priority[i]<priority[j])
   {
    int temp=burst_time[i];
    burst_time[i]=burst_time[j];
    burst_time[j]=temp;
    temp=priority[i];
    priority[i]=priority[j];
    priority[j]=temp;
   }
  }
 }

 waiting_time[0]=0;
 for(i=1;i<n;i++)
 {
  waiting_time[i]=0;
  for(j=0;j<i;j++)
  {
   waiting_time[i]=waiting_time[i]+burst_time[j];
  }
 }
```

```
   printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
   for(i=0;i<n;i++)
   {
     turn_around_time[i]=burst_time[i]+waiting_time[i];
     average_waiting_time=average_waiting_time+waiting_time[i];
     average_turn_around_time=average_turn_around_time+turn_around_time[i];
     printf("\nProcess[%d]\t\t%d\t\t%d\t\t%d", i+1, burst_time[i], waiting_time[i],
 turn_around_time[i]);
   }
 }
```

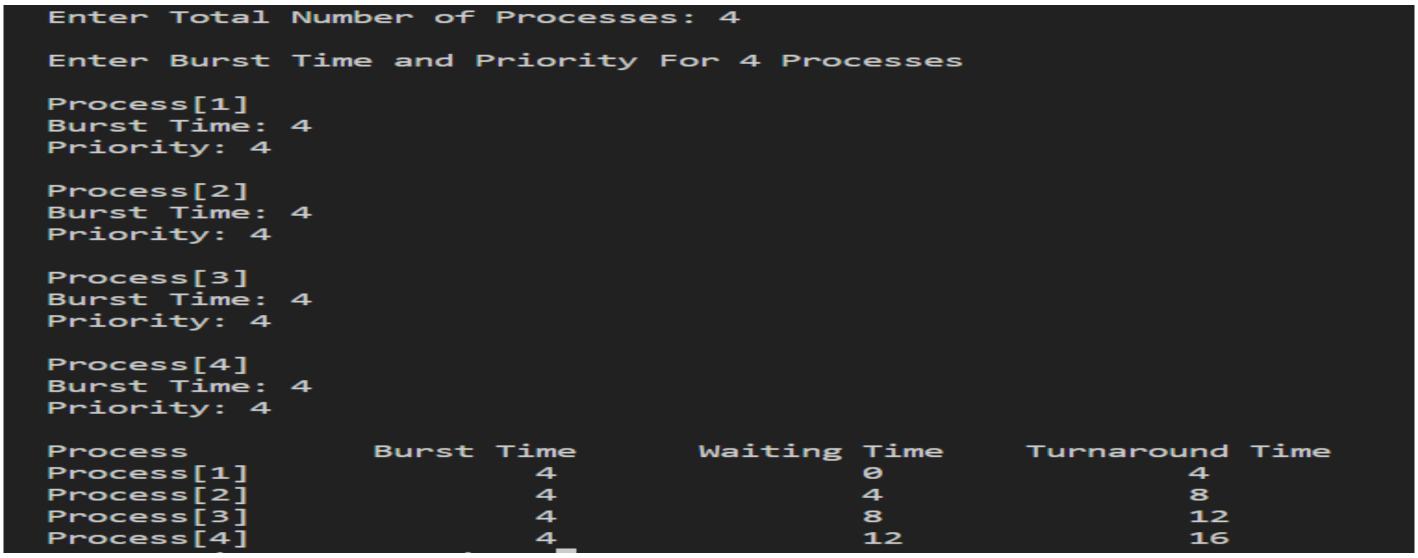
ScreenShot:-

INPUT:-

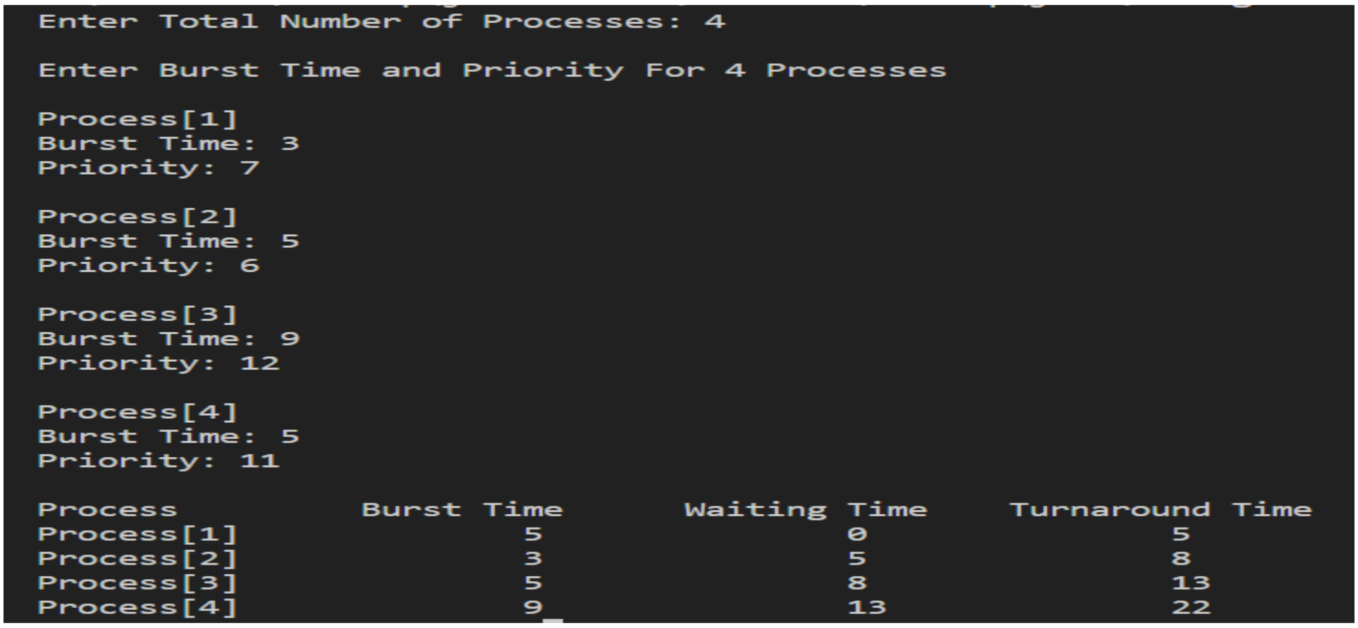


```
#include<stdio.h>

int main()
{
  int n, burst_time[20], priority[20], waiting_time[20], turn_around_time[20], i, j;
  float average_waiting_time, average_turn_around_time;
  printf("Enter Total Number of Processes: ");
  scanf("%d", &n);

  printf("\nEnter Burst Time and Priority For %d Processes\n", n);
  for(i=0;i<n;i++)
  {
    printf("\nProcess[%d]\n", i+1);
    printf("Burst Time: ");
    scanf("%d", &burst_time[i]);
    printf("Priority: ");
    scanf("%d", &priority[i]);
  }

  for(i=0;i<n;i++)
  {
    for(j=0;j<n;j++)
    {
      if(priority[i]<priority[j])
      {
        int temp=burst_time[i];
        burst_time[i]=burst_time[j];
        burst_time[j]=temp;
        temp=priority[i];
        priority[i]=priority[j];
        priority[j]=temp;
      }
    }
  }

  waiting_time[0]=0;
  for(i=1;i<n;i++)
  {
    waiting_time[i]=0;
    for(j=0;j<i;j++)
    {
      waiting_time[i]=waiting_time[i]+burst_time[j];
    }
  }

  printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
  for(i=0;i<n;i++)
  {
    turn_around_time[i]=burst_time[i]+waiting_time[i];
    average_waiting_time=average_waiting_time+waiting_time[i];
    average_turn_around_time=average_turn_around_time+turn_around_time[i];
    printf("\nProcess[%d]\t\t%d\t\t%d\t\t%d", i+1, burst_time[i], waiting_time[i],
turn_around_time[i]);
  }
}
```

OUTPUT:-



```
 Enter Total Number of Processes: 4

 Enter Burst Time and Priority For 4 Processes

 Process[1]
 Burst Time: 4
 Priority: 4

 Process[2]
 Burst Time: 4
 Priority: 4

 Process[3]
 Burst Time: 4
 Priority: 4

 Process[4]
 Burst Time: 4
 Priority: 4

 Process                Burst Time          Waiting Time        Turnaround Time
 Process[1]                 4                   0                    4
 Process[2]                 4                   4                    8
 Process[3]                 4                   8                    12
 Process[4]                 4                   12                   16
```



```
Enter Total Number of Processes: 4

Enter Burst Time and Priority For 4 Processes

Process[1]
Burst Time: 3
Priority: 7

Process[2]
Burst Time: 5
Priority: 6

Process[3]
Burst Time: 9
Priority: 12

Process[4]
Burst Time: 5
Priority: 11

Process                Burst Time          Waiting Time        Turnaround Time
Process[1]                 5                   0                    5
Process[2]                 3                   5                    8
Process[3]                 5                   8                    13
Process[4]                 9                   13                   22
```

Aim: Find CT, TAT, WT in RR.

Theory: CT (Completion Time), TAT (Turn Around Time), and WT (Waiting Time) are important metrics in the context of process scheduling and CPU scheduling algorithms. In a Round Robin (RR) scheduling algorithm with preemptive mode, the CPU is assigned to each process for a fixed time quantum, and if a process has not completed its execution within the given time quantum, it is moved to the back of the queue. The criteria for determining the arrival time of a process are used to calculate the CT, TAT, and WT for each process.

Completion Time (CT): The completion time of a process is the time when the process has finished executing and has released all its resources. In the context of RR scheduling with preemptive mode, the CT of a process can be calculated as the sum of the time quantum assigned to the process and the time that the process spent waiting in the queue.

Turn Around Time (TAT): The turn around time of a process is the difference between its completion time and its arrival time. It is a measure of the total time spent by a process from the moment it arrives to the moment it completes execution. The TAT of a process can be calculated as the difference between its CT and its arrival time.

Waiting Time (WT): The waiting time of a process is the time spent by the process waiting in the queue for its turn to be assigned to the CPU. In the context of RR scheduling with preemptive mode, the waiting time of a process can be calculated as the difference between its TAT and its burst time (the time required for the process to complete its execution).

Here's how to calculate CT, TAT, and WT when scheduling processes based on arrival time:

CT (Completion Time):

This is the time when a process has completed its execution and is no longer in the ready queue.
For a process with arrival time "a" and burst time "b", the completion time is calculated as follows:
CT = a + b

TAT (Turnaround Time):

This is the total time taken by a process from the moment it arrives in the ready queue to the moment it completes its execution.
For a process with arrival time "a" and completion time "ct", the turnaround time is calculated as follows:
TAT = ct – a

WT (Waiting Time):

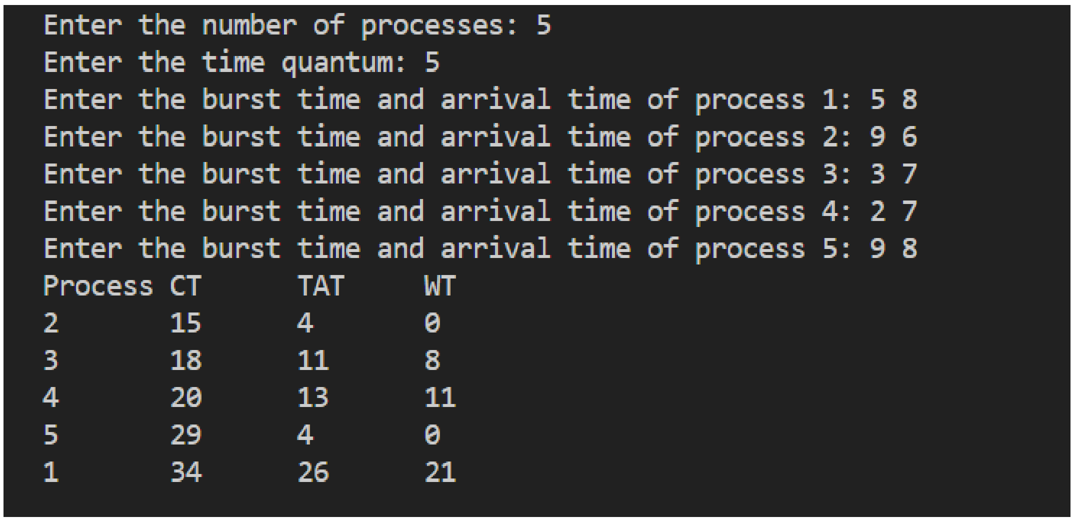
This is the amount of time a process spends waiting in the ready queue before it is executed. For a process with arrival time "a", burst time "b", and completion time "ct", the waiting time is calculated as follows:
WT = TAT - b
Note: These calculations assume that the CPU is available for execution at the time of arrival for each process.

Code :
```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#define MAX_PROCESSES 50
#define MAX_TIME 500

struct Process {
  int id;
  int burstTime;
  int arrivalTime;
  int completionTime;
  int turnAroundTime;
  int waitingTime;
};

int compare(const void *a, const void *b) {
  return ((struct Process *)a)->arrivalTime - ((struct Process *)b)->arrivalTime;
}

int main() {
  int n, quantum;
  printf("Enter the number of processes: ");
  scanf("%d", &n);

  printf("Enter the time quantum: ");
  scanf("%d", &quantum);

  struct Process processes[MAX_PROCESSES];

  for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;
    printf("Enter the burst time and arrival time of process %d: ", i + 1);
    scanf("%d%d", &processes[i].burstTime, &processes[i].arrivalTime);
  }
  qsort(processes, n, sizeof(struct Process), compare);

  int time = 0;
  int process = 0;
  int count = 0;

  while (count < n) {
    int nextProcess = -1;
    int shortestTime = MAX_TIME;

    for (int i = 0; i < n; i++) {
      if (processes[i].arrivalTime <= time && processes[i].burstTime > 0) {
        nextProcess = i;
        shortestTime = processes[i].burstTime;
        break;
      }
    }

    if (nextProcess == -1) {
      time++;
      continue;
    }

    if (processes[nextProcess].burstTime > quantum) {
      time += quantum;
      processes[nextProcess].burstTime -= quantum;
      processes[nextProcess].arrivalTime = time;
    } else {
      time += processes[nextProcess].burstTime;
      processes[nextProcess].completionTime = time;
      processes[nextProcess].turnAroundTime =
          processes[nextProcess].completionTime - processes[nextProcess].arrivalTime;
      processes[nextProcess].waitingTime = processes[nextProcess].turnAroundTime -
processes[nextProcess].burstTime;
      processes[nextProcess].burstTime = 0;
      count++;
    }
  }

  printf("Process\tCT\tTAT\tWT\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\n", processes[i].id, processes[i].completionTime,
         processes[i].turnAroundTime, processes[i].waitingTime);
  }

  return 0;}
```

INPUT:-

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

#define MAX_PROCESSES 50
#define MAX_TIME 500

struct Process {
  int id;
  int burstTime;
  int arrivalTime;
  int completionTime;
  int turnAroundTime;
  int waitingTime;
};

int compare(const void *a, const void *b) {
  return ((struct Process *)a)->arrivalTime - ((struct Process *)b)->arrivalTime;
}

int main() {
  int n, quantum;
  printf("Enter the number of processes: ");
  scanf("%d", &n);

  printf("Enter the time quantum: ");
  scanf("%d", &quantum);

  struct Process processes[MAX_PROCESSES];

  for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;
    printf("Enter the burst time and arrival time of process %d: ", i + 1);
    scanf("%d%d", &processes[i].burstTime, &processes[i].arrivalTime);
  }

  qsort(processes, n, sizeof(struct Process), compare);

  int time = 0;
  int process = 0;
  int count = 0;

  while (count < n) {
    int nextProcess = -1;
    int shortestTime = MAX_TIME;

    for (int i = 0; i < n; i++) {
      if (processes[i].arrivalTime <= time && processes[i].burstTime > 0) {
        nextProcess = i;
        shortestTime = processes[i].burstTime;
        break;
      }
    }
    if (nextProcess == -1) {
      time++;
      continue;
    }

    if (processes[nextProcess].burstTime > quantum) {
      time += quantum;
      processes[nextProcess].burstTime -= quantum;
      processes[nextProcess].arrivalTime = time;
    } else {
      time += processes[nextProcess].burstTime;
      processes[nextProcess].completionTime = time;
      processes[nextProcess].turnAroundTime =
          processes[nextProcess].completionTime - processes[nextProcess].arrivalTime;
      processes[nextProcess].waitingTime = processes[nextProcess].turnAroundTime -
processes[nextProcess].burstTime;
      processes[nextProcess].burstTime = 0;
      count++;
    }
  }

  printf("Process\tCT\tTAT\tWT\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\n", processes[i].id, processes[i].completionTime,
           processes[i].turnAroundTime, processes[i].waitingTime);
  }

  return 0;
}
```

OUTPUT:-



```
Enter the number of processes: 5
Enter the time quantum: 5
Enter the burst time and arrival time of process 1: 5 8
Enter the burst time and arrival time of process 2: 9 6
Enter the burst time and arrival time of process 3: 3 7
Enter the burst time and arrival time of process 4: 2 7
Enter the burst time and arrival time of process 5: 9 8
Process CT      TAT     WT
2       15      4       0
3       18      11      8
4       20      13      11
5       29      4       0
1       34      26      21
```

Aim: Try Preemptive scheduling with MinHeaps.

Theory: In preemptive scheduling, a running process can be interrupted by a higher priority process at any time. MinHeap is a data structure that can be used to implement a priority queue, where the process with the highest priority is always at the top of the queue.

In a preemptive scheduling algorithm using MinHeaps, the processes are stored in a MinHeap data structure, where the priority of each process is determined by its burst time, arrival time, or some other factor. The process at the top of the heap, i.e., with the lowest burst time or earliest arrival time, is executed first.

When a new process arrives or an existing process's burst time decreases, the MinHeap is updated accordingly to maintain the property of the MinHeap, where the process with the lowest burst time or earliest arrival time is always at the top.

When a process is executed, its burst time is decremented. If its burst time becomes zero, it is removed from the MinHeap. If its burst time is still greater than zero, it is reinserted into the MinHeap with an updated burst time.

This process of executing the process at the top of the MinHeap, updating the MinHeap, and executing the next process continues until all the processes have been executed.

In this way, MinHeaps can be used to implement a preemptive scheduling algorithm that ensures that the process with the highest priority is always executed first.

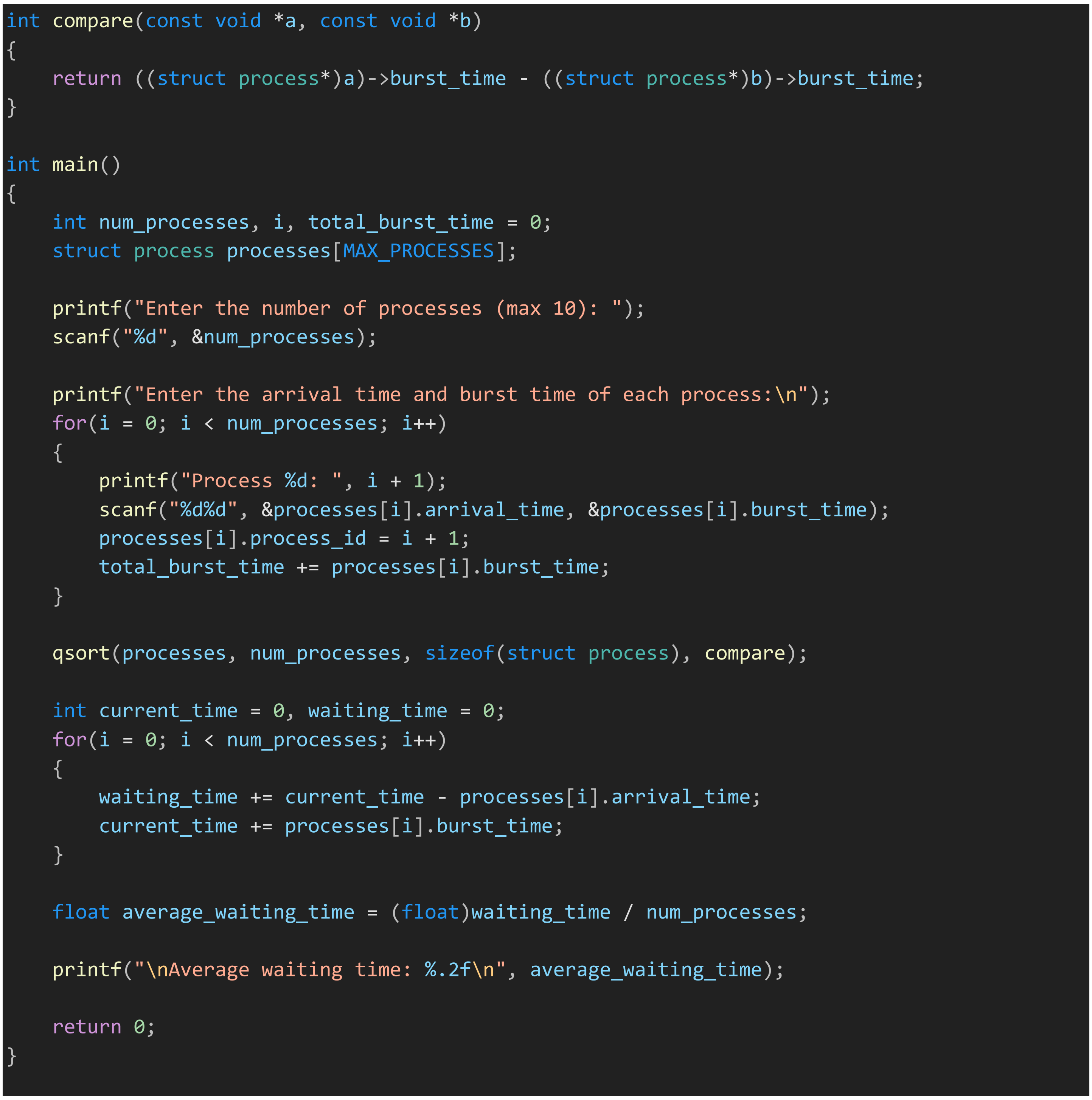
Code for Preemptive scheduling :

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

#define MAX_PROCESSES 10

// A process structure to store the process id, burst time and arrival time
struct process
{
   int process_id;
   int burst_time;
   int arrival_time;
};

// Function to compare the burst time of two processes
int compare(const void *a, const void *b)
{
   return ((struct process*)a)->burst_time - ((struct process*)b)->burst_time;
}
```

```
int main()
{
   int num_processes, i, total_burst_time = 0;
   struct process processes[MAX_PROCESSES];

   printf("Enter the number of processes (max 10): ");
   scanf("%d", &num_processes);

   printf("Enter the arrival time and burst time of each process:\n");
   for(i = 0; i < num_processes; i++)
   {
      printf("Process %d: ", i + 1);
      scanf("%d%d", &processes[i].arrival_time, &processes[i].burst_time);
      processes[i].process_id = i + 1;
      total_burst_time += processes[i].burst_time;
   }

   qsort(processes, num_processes, sizeof(struct process), compare);

   int current_time = 0, waiting_time = 0;
   for(i = 0; i < num_processes; i++)
   {
      waiting_time += current_time - processes[i].arrival_time;
      current_time += processes[i].burst_time;
   }

   float average_waiting_time = (float)waiting_time / num_processes;

   printf("\nAverage waiting time: %.2f\n", average_waiting_time);

   return 0;
}
```

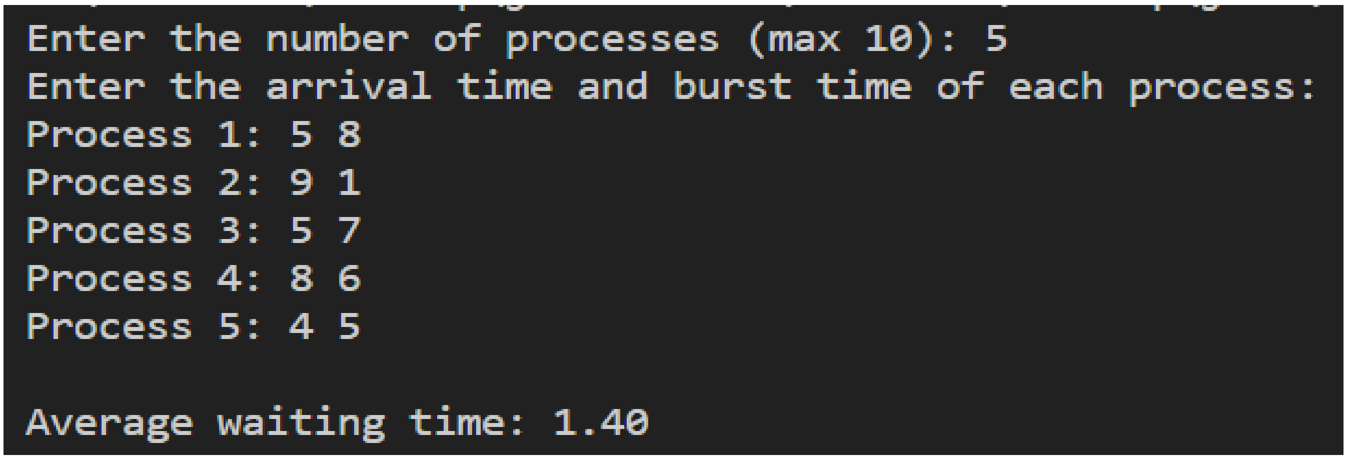
INPUT:-

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

#define MAX_PROCESSES 10

// A process structure to store the process id, burst time and arrival time
struct process
{
    int process_id;
    int burst_time;
    int arrival_time;
};

// Function to compare the burst time of two processes
```



```c
int compare(const void *a, const void *b)
{
    return ((struct process*)a)->burst_time - ((struct process*)b)->burst_time;
}

int main()
{
    int num_processes, i, total_burst_time = 0;
    struct process processes[MAX_PROCESSES];

    printf("Enter the number of processes (max 10): ");
    scanf("%d", &num_processes);

    printf("Enter the arrival time and burst time of each process:\n");
    for(i = 0; i < num_processes; i++)
    {
        printf("Process %d: ", i + 1);
        scanf("%d%d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].process_id = i + 1;
        total_burst_time += processes[i].burst_time;
    }

    qsort(processes, num_processes, sizeof(struct process), compare);

    int current_time = 0, waiting_time = 0;
    for(i = 0; i < num_processes; i++)
    {
        waiting_time += current_time - processes[i].arrival_time;
        current_time += processes[i].burst_time;
    }

    float average_waiting_time = (float)waiting_time / num_processes;

    printf("\nAverage waiting time: %.2f\n", average_waiting_time);

    return 0;
}
```

OUTPUT:-



```
Enter the number of processes (max 10): 5
Enter the arrival time and burst time of each process:
Process 1: 5 8
Process 2: 9 1
Process 3: 5 7
Process 4: 8 6
Process 5: 4 5

Average waiting time: 1.40
```