2. Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a) FCFS:

ALGORITHM:

1. Enter all the processes and their burst time.
2. Find waiting time, **WT** of all the processes.
3. For the 1st process, **WT = 0**.
4. For all the next processes i, **WT[i] = BT[i-1] + WT[i-1]**.
5. Calculate Turnaround **time = WT + BT** for all the processes.
6. Calculate **average waiting time** = total waiting time/no. of processes.
7. Calculate **average turnaround time** = total turnaround time/no. of processes.

CODE:

```
#include <stdio.h>
int main()


{
     int pid[15];
     int bt[15];
     int n;

    printf("Enter the number of processes: ");
    scanf("%d",&n);
    printf("Enter process id of all the processes: ");

    for(int i=0;i<n;i++)
   {
          scanf("%d",&pid[i]);
    }
    printf("Enter burst time of all the processes: ");

    for(int i=0;i<n;i++)
    {
       scanf("%d",&bt[i]);
    }


    int i, wt[n];
    wt[0]=0;

   for(i=1; i<n; i++)                          //for calculating waiting time of each process
   {
       wt[i]= bt[i-1]+ wt[i-1];
    }
    printf("Process ID     Burst Time     Waiting Time     TurnAround Time\n");

    float twt=0.0;
    float tat= 0.0;

    for(i=0; i<n; i++)
    {
        printf("%d\t\t", pid[i]);
        printf("%d\t\t", bt[i]);
        printf("%d\t\t", wt[i]);
        printf("%d\t\t", bt[i]+wt[i]);     //calculating and printing turnaround time of each process
        printf("\n");

        twt += wt[i];                                  //for calculating total waiting time
        tat += (wt[i]+bt[i]);                          //for calculating total turnaround time

    }
    float att,awt;
    awt = twt/n;                    //for calculating average waiting time
    att = tat/n;                    //for calculating average turnaround time

    printf("Avg. waiting time= %f\n",awt);
    printf("Avg. turnaround time= %f",att);
}
```

OUTPUT:

```
Enter the number of processes: 3
Enter process id of all the processes: 1 2 3
Enter burst time of all the processes: 5 11 11
Process ID     Burst Time     Waiting Time     TurnAround Time
1                     5                0                 5
2                    11                5                16
3                    11               16                27
Avg. waiting time= 7.000000
Avg. turnaround time= 16.000000
```

b) SJF

ALGORITHM:

1. Enter number of processes.
2. Enter the **burst time** of all the processes.
3. Sort all the processes according to their **burst time**.
4. Find waiting time, **WT** of all the processes.
5. For the smallest process, **WT = 0**.
6. For all the next processes **i**, find waiting time by adding burst time of all the previously completed process.
7. Calculate **Turnaround time = WT + BT** for all the processes.
8. Calculate **average waiting time = total waiting time / no. of processes**.
9. Calculate **average turnaround time= total turnaround time / no. of processes**.

CODE:

```
#include<stdio.h>

int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;
    float avg_wt,avg_tat;

    printf("Enter number of process:");
    scanf("%d",&n);
    printf("\nEnter 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++)                                    //sorting of burst times
    {
           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++)                              //finding the waiting time of all the processes
    {
         wt[i]=0;

         for(j=0;j<i;j++)
        {
           wt[i]+=bt[j];      //individual WT by adding BT of all previous completed processes
           total+=wt[i];
        }
    }

      avg_wt=(float)total/n;                              //average waiting time
      printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

      for(i=0;i<n;i++)
      {
           tat[i]=bt[i]+wt[i];                  //turnaround time of individual processes
           totalT+=tat[i];                      //total turnaround time

           printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
       }

        avg_tat=(float)totalT/n;                       //average turnaround time

        printf("\n\nAverage Waiting Time=%f",avg_wt);
        printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

OUTPUT:

```
Enter number of process:4

Enter Burst Time:
p1:5
p2:4
p3:12
```

```
p4:7

Process     Burst Time     Waiting Time   Turnaround Time
   p2           4               0               4
   p1           5               4               9
   p4           7               9              16
   p3          12              16              28
Average Waiting Time=7.250000
Average Turnaround Time=14.250000
```

c) ROUND ROBIN

ALGORITHM:

Step 1: Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

Step 2: Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

Step 3: If the process cannot complete their task within defined time interval or slots because it is stopped by another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

Step 4: Similarly, the scheduler selects another process from the ready queue to execute its tasks. When a process finishes its task within time slots, the process will not go for further execution because the process's burst time is finished.

Step 5: Similarly, we repeat all the steps to execute the process until the work has finished.

CODE:

```
#include<stdio.h>
#include<conio.h>
void main()

{
    int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
    float avg_wt, avg_tat;

    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP;                        // Assign the number of process to variable y

   // Use for loop to enter the details of the process like Arrival time and the Burst Time

    for (i=0; i<NOP; i++)

   {
            printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
            printf(" Arrival time is: \t");             // Accept arrival time
            scanf("%d", &at[i]);
            printf(" \nBurst time is: \t");             // Accept the Burst time
            scanf("%d", &bt[i]);

            temp[i] = bt[i];                                     // store the burst time in temp array
    }

    printf("Enter the Time Quantum for the process: \t");      // Accept the Time quantum
    scanf("%d", &quant);

   // Display the process No, burst time, Turn Around Time and the waiting time
    printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

    for (sum=0, i = 0; y!=0; )
   {
             if(temp[i] <= quant && temp[i] > 0)            // define the conditions
             {
                     sum = sum + temp[i];
                     temp[i] = 0;
                    count=1;
             }
             else if(temp[i] > 0)
             {
                     temp[i] = temp[i] - quant;
                     sum = sum + quant;
             }

     if(temp[i]==0 && count==1)
    {
       y--;                                                      //decrement the process no.
       printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
       wt = wt+sum-at[i]-bt[i];
       tat = tat+sum-at[i];
       count =0;
    }


     if(i==NOP-1)
    {
       i=0;
     }
    else if(at[i+1]<=sum)
    {
       i++;
    }
    else
    {
       i=0;
    }
  }

   avg_wt = wt * 1.0/NOP;     // represents the average waiting time and Turn Around time
   avg_tat = tat * 1.0/NOP;    // represents the average Turn Around time

   printf("\n Average Turn Around Time: \t%f", avg_wt);
  printf("\n Average Waiting Time: \t%f", avg_tat);
  getch();

 }
```

OUTPUT:

```
Total number of process in the system: 4


Enter the Arrival and Burst time of the Process[1]

Arrival time is: 0
```

```
Burst time is:  8

Enter the Arrival and Burst time of the
Process[2]Arrival time is: 1

Burst time is: 5


Enter the Arrival and Burst time of the
Process[3]Arrival time is: 2

Burst time is: 10


Enter the Arrival and Burst time of the
Process[4]Arrival time is: 3

Burst time is: 11


Enter the Time Quantum for the process: 6

  Process No          Burst Time        TAT        Waiting
                                                   Time
 Process No[2]           5               10           5
 Process No[1]           8               25          17
 Process No[3]          10               27          17
 Process No[4]          11               31          20


  Average Turn Around Time:
                            14.75000
  0Average Waiting Time:    23.25000
```

d) PRIORITY SCHEDULEING:

ALGORITHM:

The algorithms prioritize the processes that must be carried out and schedule them accordingly. A higher priority process will receive CPU priority first, and this will continue until all of the processes are finished. The algorithm that places the processes in the ready queue in the order determined by their priority and chooses which ones to go to the job queue for execution requires both the job queue and the ready queue. Due to the process' greater priority, the Priority Scheduling Algorithm is in charge of transferring it from the ready queue to the work queue.

CODE:

```
#include <stdio.h>

void swap(int *a,int *b)

{
    int temp=*a;
    *a=*b;
    *b=temp;
}

int main()
{
    int n;
    printf("Enter Number of Processes: ");
    scanf("%d",&n);

    int burst[n],priority[n],index[n];

    for(int i=0;i<n;i++)
    {
            printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
            scanf("%d %d",&burst[i],&priority[i]);
            index[i]=i+1;
    }

    for(int i=0;i<n;i++)
    {
           int temp=priority[i],
           int m=i;
```

```
     for(int j=i;j<n;j++)
     {
            if(priority[j] > temp)
            {
               temp=priority[j];
               m=j;
            }
        }
        swap(&priority[i], &priority[m]);
        swap(&burst[i], &burst[m]);
        swap(&index[i],&index[m]);
    }

    int t=0;
    printf("Order of process Execution is\n");

    for(int i=0;i<n;i++)
    {
        printf("P%d is executed from %d to %d\n",index[i],t,t+burst[i]);
        t+=burst[i];
    }

    printf("\n");
    printf("Process Id\tBurst Time\tWait Time\n");

    int wait_time=0;
    int total_wait_time = 0;

    for(int i=0;i<n;i++)
    {
        printf("P%d\t\t%d\t\t%d\n",index[i],burst[i],wait_time);
        total_wait_time += wait_time;
        wait_time += burst[i];
    }

    float avg_wait_time = (float) total_wait_time / n;
    printf("Average waiting time is %f\n", avg_wait_time);

    int total_Turn_Around = 0;

    for(int i=0; i < n; i++)
   {
        total_Turn_Around += burst[i];
    }

    float avg_Turn_Around = (float) total_Turn_Around / n;
    printf("Average TurnAround Time is %f",avg_Turn_Around);
    return 0;
}
```

OUTPUT:

```
Enter Number of Processes: 2
Enter Burst Time and Priority Value for Process 1: 5 3
Enter Burst Time and Priority Value for Process 2: 4 2
Order of process Execution is
P1 is executed from 0 to 5
P2 is executed from 5 to 9
Process Id          Burst Time          Wait Time
P1                  5                   0
P2                  4                   5
Average waiting time is 2.500000
Average TurnAround Time is 4.50
```

3. Develop a C program to simulate producer-consumer problem using semaphores

ALGORITHM:

1. Initialize the empty semaphore to the size of the buffer and the full semaphore to 0.
2. The producer acquires the empty semaphore to check if there are any empty slots in the buffer. If there are no empty slots, the producer blocks until a slot becomes available.
3. The producer acquires the mutex to access the buffer, inserts a data item into an empty slot in the buffer, and releases the mutex.
4. The producer releases the full semaphore to indicate that a slot in the buffer is now full.
5. The consumer acquires the full semaphore to check if there are any full slots in the buffer. If there are no full slots, the consumer blocks until a slot becomes available.
6. The consumer acquires the mutex to access the buffer, reads a data item from a full slot in the buffer, and releases the mutex.
7. The consumer releases the empty semaphore to indicate that a slot in the buffer is now empty.

CODE:

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

int mutex = 1;                                  // Initialize a mutex to 1
int full = 0;                                   // Number of full slots as 0
int empty = 10, x = 0;                          // Number of empty slots as size  of buffer

void producer()                 // Function to produce an item and add it to the buffer
{
   --mutex;                                     // Decrease mutex value by 1
   ++full;                                      // Increase the number of full slots by 1
   --empty;                                     // Decrease the number of empty slots by 1
    x++;                                             // Item produced
    printf("\nProducer produces"item %d", x);

   ++mutex;                                // Increase mutex value by 1
}

void consumer()                 // Function to consume an item and remove it from buffer
{
   --mutex;                         // Decrease mutex value by 1
   --full;                          // Decrease the number of full slots by 1
   ++empty;                         // Increase the number of empty slots by 1
   printf("\nConsumer consumes item %d", x);
   x--;
   ++mutex;                     // Increase mutex value by 1
}
int main()                          // Driver Code
{
   int n, i;

   printf("\n1. 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))
          // If mutex is 1 and empty  is non-zero, then it is possible to produce
        {
            producer();
         }
         else                                   // Otherwise, print buffer is full
         {
            printf("Buffer is full!");
         }
         break;

      case 2:

         // If mutex is 1 and full is non-zero, then it is possible to consume
         if ((mutex == 1) && (full != 0))
         {
            consumer();
         }
         else                                     // Otherwise, print Buffer  is empty
        {
            printf("Buffer is empty!");
         }
         break;

      case 3:                                   // Exit Condition
         exit(0);
         break;
      }
   }
}
```

OUTPUT:

```
1. Produce 2. Consume 3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 3
```

2nd one:

```
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 300

Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 300

Enter your choice: 3
```