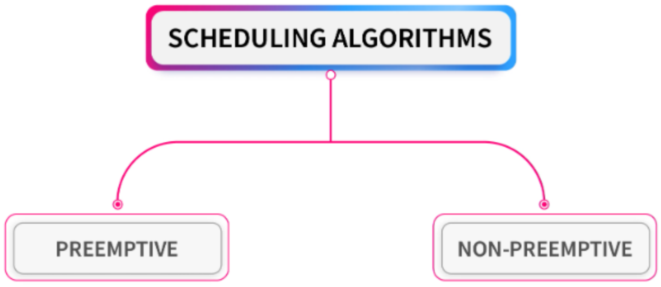
Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Operating System Laboratory
Course Code:	DJ19CEL403
Experiment No.:	04

AIM: **CPU SCHEDULING ALGORITHMS LIKE FCFS, SJF, ROUND ROBIN ETC.**

THEORY:

SCHEDULING ALGORITHMS:

- A CPU scheduling algorithm is used to determine which process will use CPU for execution and which processes to hold or remove from execution.
- The main goal or objective of CPU scheduling algorithms is to make sure that the CPU is never in an idle state, meaning that the OS has at least one of the processes ready for execution among the available processes in the ready queue.
- There are two types of scheduling algorithms:
  - **Preemptive**
    - In these algorithms, processes are assigned with a priority.
    - Whenever a high-priority process comes in, the lower-priority process which has occupied the CPU is pre-empted.
    - That is, it releases the CPU, and the high-priority process takes the CPU for its execution.
  - **Non-Preemptive**
    - In these algorithms, we cannot pre-empt the process.
    - That is, once a process is running in CPU, it will release it either by context switching or terminating.
    - Often, these are the types of algorithms that can be used because of the limitation of the hardware.



There are some important terminologies to know for understanding the scheduling algorithms:

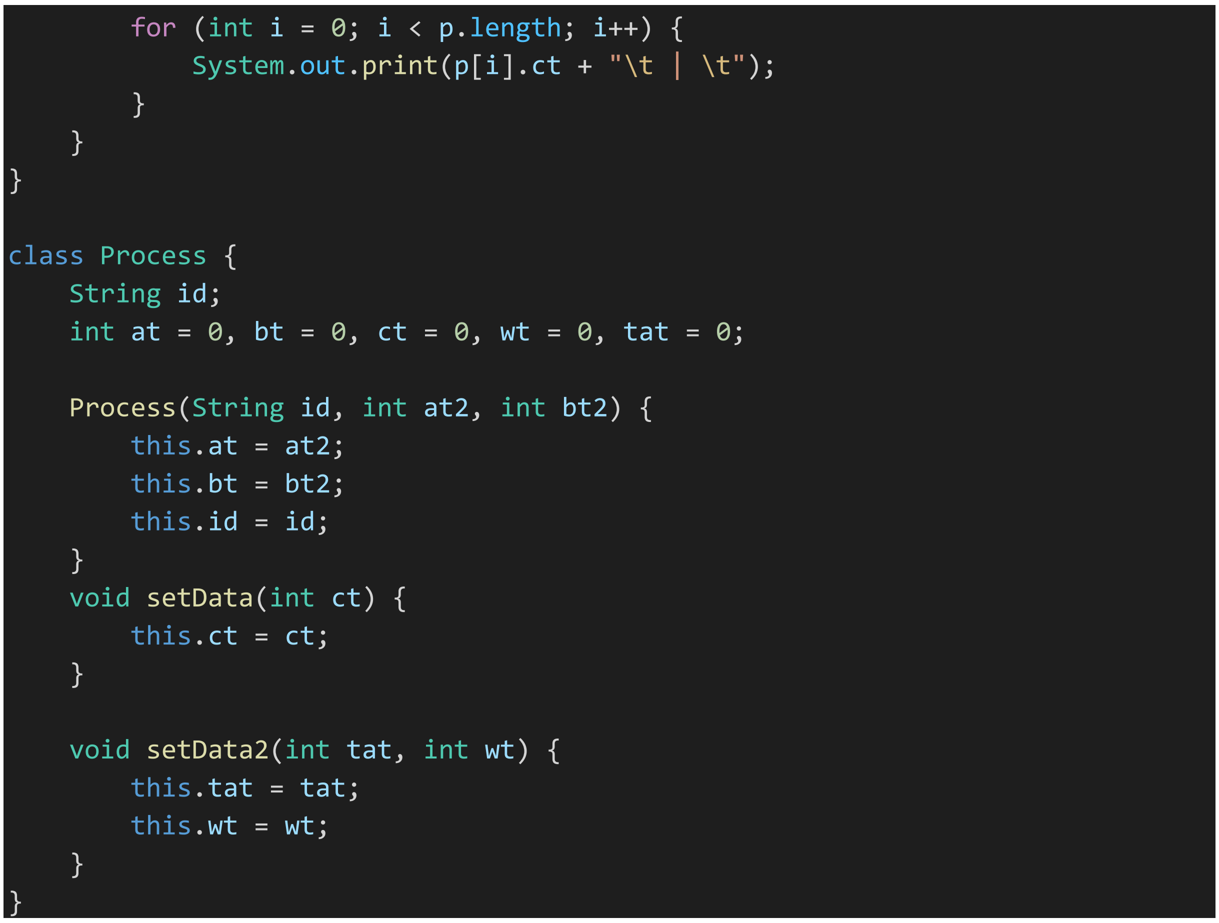
- **Arrival Time:** This is the time at which a process arrives in the ready queue.
- **Completion Time:** This is the time at which a process completes its execution.
- **Burst Time:** This is the time required by a process for CPU execution.
- **Turn-Around Time:** This is the difference in time between completion time and arrival time. This can be calculated as:
  - Turn Around Time = Completion Time – Arrival Time.
- **Waiting Time:** This is the difference in time between turnaround time and burst time. This can be calculated as:
  - Waiting Time = Turn Around Time – Burst Time.
- **Throughput:** It is the number of processes that are completing their execution per unit time.

FCFS – First Come First Serve

- In this type of scheduling algorithm, the CPU is first allocated to the process which requests the CPU first.
- That means the process with minimal arrival time will be executed first by the CPU.
- It is a non-preemptive scheduling algorithm as the priority of processes does not matter, and they are executed in the manner they arrive in front of the CPU.
- This scheduling algorithm is implemented with a FIFO(First In First Out) queue.
- As the process is ready to be executed, its Process Control Block (PCB) is linked with the tail of this FIFO queue.
- Now when the CPU becomes free, it is assigned to the process at the beginning of the queue.
- Advantages
    - Involves no complex logic and just picks processes from the ready queue one by one.
    - Easy to implement and understand.
    - Every process will eventually get a chance to run so no starvation occurs.
- Disadvantages
    - Waiting time for processes with less execution time is often very long.
    - It favours CPU-bound processes then I/O processes.
    - Leads to convoy effect.
    - Causes lower device and CPU utilization.
    - Poor in performance as the average wait time is high.

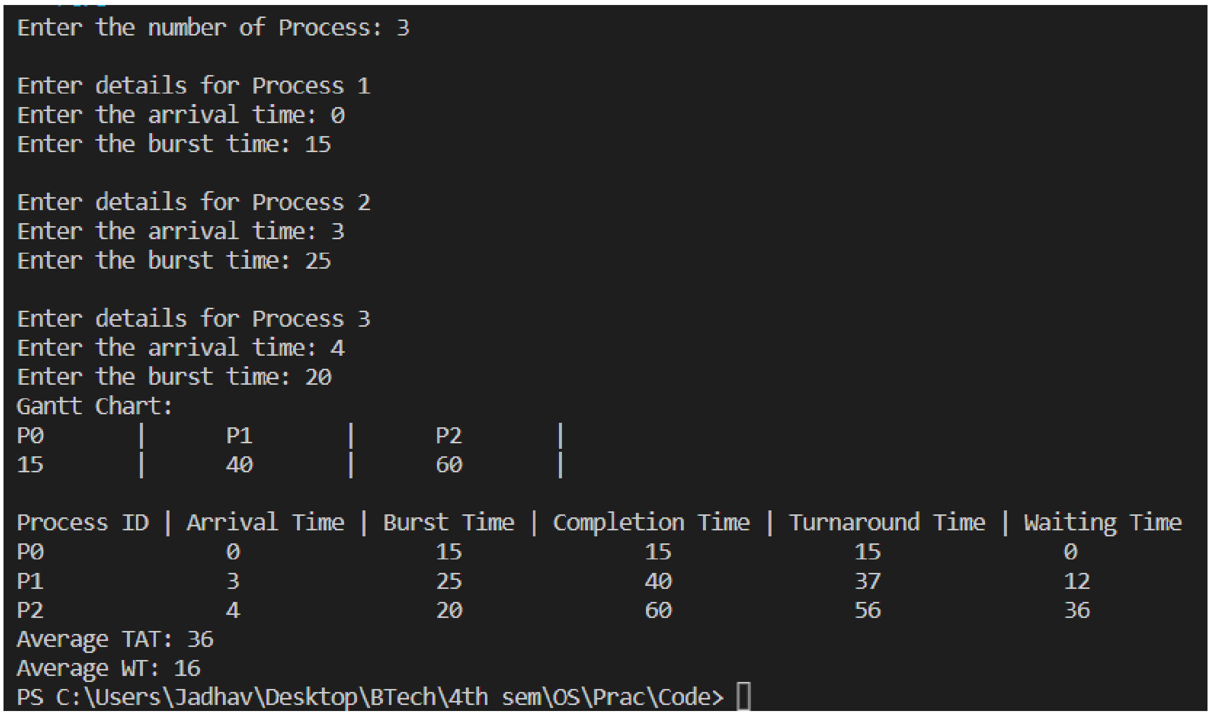
- **CODE:**

```java
import java.util.Scanner;

class FCFS {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of Process: ");
        int n = sc.nextInt();
        Process[] p = new Process[n];

        for (int i = 0; i < n; i++) {
            System.out.println("\nEnter details for Process " + (i + 1));

            System.out.print("Enter the arrival time: ");
            int at = sc.nextInt();
            System.out.print("Enter the burst time: ");
            int bt = sc.nextInt();

            p[i] = new Process("P" + i, at, bt);
        }
        scheduleFCFS(p);
        printTable(p);
        sc.close();
    }

    private static void printTable(Process[] p) {
        int totalWt = 0;
        int totalTAT = 0;
        System.out.println(
                "\n\nProcess ID | Arrival Time | Burst Time | Completion Time
| Turnaround Time | Waiting Time");
        for (int i = 0; i < p.length; i++) {
            System.out.println(p[i].id + "\t\t" + p[i].at + "\t\t" + p[i].bt +
"\t\t" + p[i].ct + "\t\t" + p[i].tat
                    + "\t\t" + p[i].wt + "\t\t");
            totalTAT += p[i].tat;
            totalWt += p[i].wt;
        }

        System.out.println("Average TAT: " + (totalTAT / p.length));
        System.out.println("Average WT: " + (totalWt / p.length));
    }

    static void scheduleFCFS(Process[] p) {
        String[] processesInCPU = new String[p.length];

        for (int i = 0; i < p.length; i++) {
            processesInCPU[i] = p[i].id;
            if (i == 0) {
                p[i].setData(p[i].bt);

            } else {
                p[i].setData(p[i - 1].ct + p[i].bt);
            }
            int tat = p[i].ct - p[i].at;
            int wt = tat - p[i].bt;
            p[i].setData2(tat, wt);
        }

        System.out.println("Gantt Chart:");
        for (int i = 0; i < p.length; i++) {
            System.out.print(p[i].id + "\t | \t");
        }
        System.out.println();
```



```
        for (int i = 0; i < p.length; i++) {
            System.out.print(p[i].ct + "\t | \t");
        }
    }
}

class Process {
    String id;
    int at = 0, bt = 0, ct = 0, wt = 0, tat = 0;

    Process(String id, int at2, int bt2) {
        this.at = at2;
        this.bt = bt2;
        this.id = id;
    }
    void setData(int ct) {
        this.ct = ct;
    }

    void setData2(int tat, int wt) {
        this.tat = tat;
        this.wt = wt;
    }
}
```

OUTPUT:



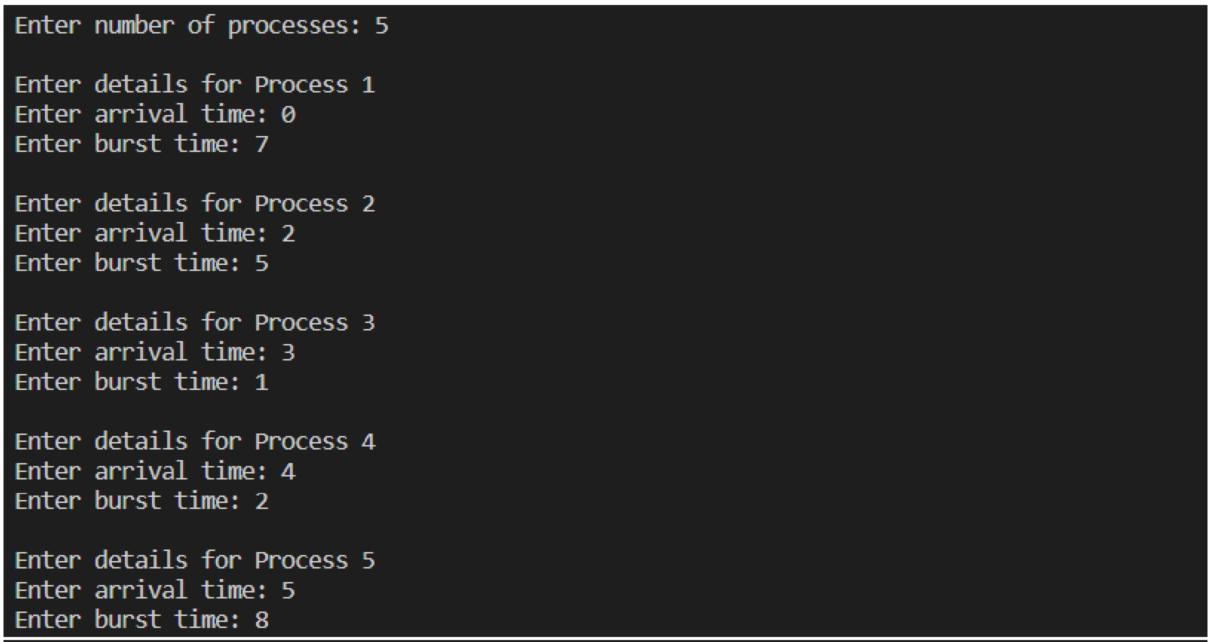
```
Enter the number of Process: 3

Enter details for Process 1
Enter the arrival time: 0
Enter the burst time: 15

Enter details for Process 2
Enter the arrival time: 3
Enter the burst time: 25

Enter details for Process 3
Enter the arrival time: 4
Enter the burst time: 20
Gantt Chart:
P0       |       P1       |       P2       |
15       |       40       |       60       |

Process ID | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time
P0              0              15              15              15              0
P1              3              25              40              37              12
P2              4              20              60              56              36
Average TAT: 36
Average WT: 16
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code> 
```

SJF – Shortest Job First (Non-Preemptive)

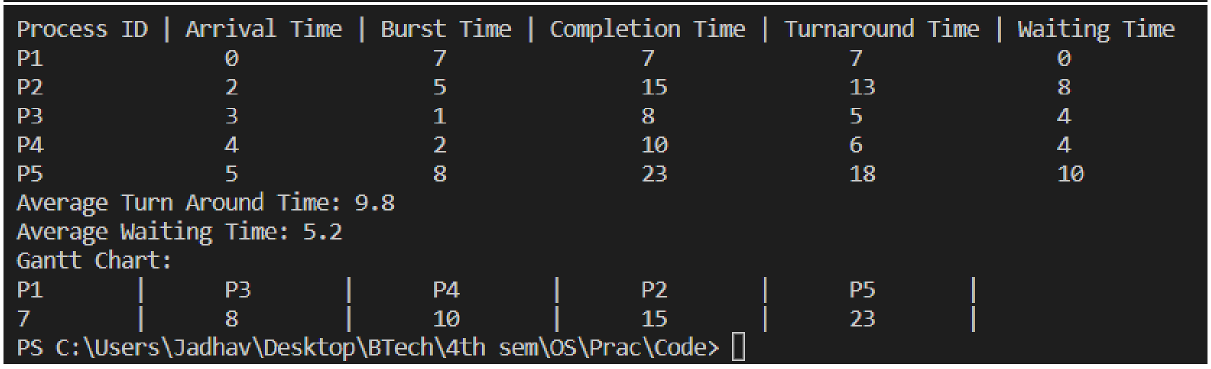
- Shortest Job First is a non-preemptive scheduling algorithm in which the process with the shortest burst or completion time is executed first by the CPU.
- That means the lesser the execution time, the sooner the process will get the CPU.
- In this scheduling algorithm, the arrival time of the processes must be the same, and the processor must be aware of the burst time of all the processes in advance.
- If two processes have the same burst time, then First Come First Serve (FCFS) scheduling is used to break the tie.
- The preemptive mode of SJF scheduling is known as the Shortest Remaining Time First scheduling algorithm.
- Advantages
    - Results in increased Throughput by executing shorter jobs first, which mostly have a shorter turnaround time.
    - Gives the minimum average waiting time for a given set of processes.
    - Best approach to minimize waiting time for other processes awaiting execution.
    - Useful for batch-type processing where CPU time is known in advance and waiting for jobs to complete is not critical.
- Disadvantages
    - May lead to starvation as if shorter processes keep on coming, then longer processes will never get a chance to run.
    - Time taken by a process must be known to the CPU beforehand, which is not always possible.
- **CODE:**

```java
import java.util.Scanner;

class SJF {
    public static void main(String args[]) {
        int n;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of processes: ");
        n = sc.nextInt();
        int p[] = new int[n]; // Process id array
        int at[] = new int[n]; // Arrival time array
        int bt[] = new int[n]; // Burst time array
        int flag[] = new int[n]; // To check if process is completed
        int i;
        for (i = 0; i < n; i++) {
            System.out.println("\nEnter details for Process " + (i + 1));

            p[i] = i + 1;
            System.out.print("Enter arrival time: ");
            at[i] = sc.nextInt();
            System.out.print("Enter burst time: ");
            bt[i] = sc.nextInt();
            flag[i] = 0;
        }
        sc.close();
        scheduler(p, at, bt, flag, n);
    }

    static void scheduler(int[] p, int[] at, int[] bt, int[] flag, int n){
        int i, j, min1, min2, min3;

        for (i = 0; i < n; i++) {
            min1 = at[i];
            min2 = bt[i];
            min3 = p[i];
            j = i - 1;
            while (j >= 0 && at[j] > min1) {
                at[j + 1] = at[j];
                bt[j + 1] = bt[j];
                p[j + 1] = p[j];
                j--;
            }
            at[j + 1] = min1;
            bt[j + 1] = min2;
            p[j + 1] = min3;
        }

        int cur_t = 0; // Current time
        int st[] = new int[n]; // Starting time of each process
        int ct[] = new int[n]; // completion time array
        int tot = 0; // To count number of processes completed
        int minbt = 1000; // To store the shortest bt
        int c = 0; // To track id of process to be scheduled next
        while (tot < n) {
            for (i = 0; i < n; i++) {
                if ((at[i] <= cur_t) && (flag[i] == 0) && (bt[i] <= minbt)) {
                    minbt = bt[i];
                    c = i;
                }
            }
            ct[c] = cur_t + minbt;
            st[c] = cur_t;
            flag[c] = 1;
            cur_t = ct[c];
            tot++;
            minbt = 1000; // reset so that bt values of remaining processes
are compared
        }
        int tat[] = new int[n]; // Turnaround Time array
        int wt[] = new int[n]; // Waiting Time array
        float sum1 = 0, sum2 = 0; // sum1 for tat and sum2 for wt
        System.out.println("\n\nProcess ID | Arrival Time | Burst Time |
Completion Time | Turnaround Time | Waiting Time");
        for (i = 0; i < n; i++) {
            tat[i] = ct[i] - at[i];
            wt[i] = tat[i] - bt[i];
            System.out.println("P" + p[i] + "\t\t" + at[i] + "\t\t" + bt[i] +
"\t\t" + ct[i] + "\t\t" + tat[i] + "\t\t" + wt[i]);
            sum1 += tat[i];
            sum2 += wt[i];
        }
        System.out.println("Average Turn Around Time: " + (sum1 / n));
        System.out.println("Average Waiting Time: " + (sum2 / n));

      for (i = 0; i < n; i++) {
            min1 = st[i];
            min2 = ct[i];
            min3 = p[i];
            j = i - 1;
            while (j >= 0 && st[j] > min1) {
                st[j + 1] = st[j];
                ct[j + 1] = ct[j];
                p[j + 1] = p[j];
                j--;
            }
            st[j + 1] = min1;
            ct[j + 1] = min2;
            p[j + 1] = min3;
        }
        System.out.println("Gantt Chart:");
        for (int z = 0; z < n; z++) {
            System.out.print("P"+p[z] + "\t | \t");
        }
        System.out.println();
        for (int z = 0; z < n; z++) {
            System.out.print(ct[z] + "\t | \t");
        }
    }
}
```

OUTPUT:



```
Enter number of processes: 5

Enter details for Process 1
Enter arrival time: 0
Enter burst time: 7

Enter details for Process 2
Enter arrival time: 2
Enter burst time: 5

Enter details for Process 3
Enter arrival time: 3
Enter burst time: 1

Enter details for Process 4
Enter arrival time: 4
Enter burst time: 2

Enter details for Process 5
Enter arrival time: 5
Enter burst time: 8
```



```
Process ID | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time
P1             0               7               7               7               0
P2             2               5               15              13              8
P3             3               1               8               5               4
P4             4               2               10              6               4
P5             5               8               23              18              10
Average Turn Around Time: 9.8
Average Waiting Time: 5.2
Gantt Chart:
P1        |     P3        |     P4        |     P2        |     P5        |
7         |     8         |     10        |     15        |     23        |
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code> 
```

SRTF - Shortest Remaining Time First (Preemptive)

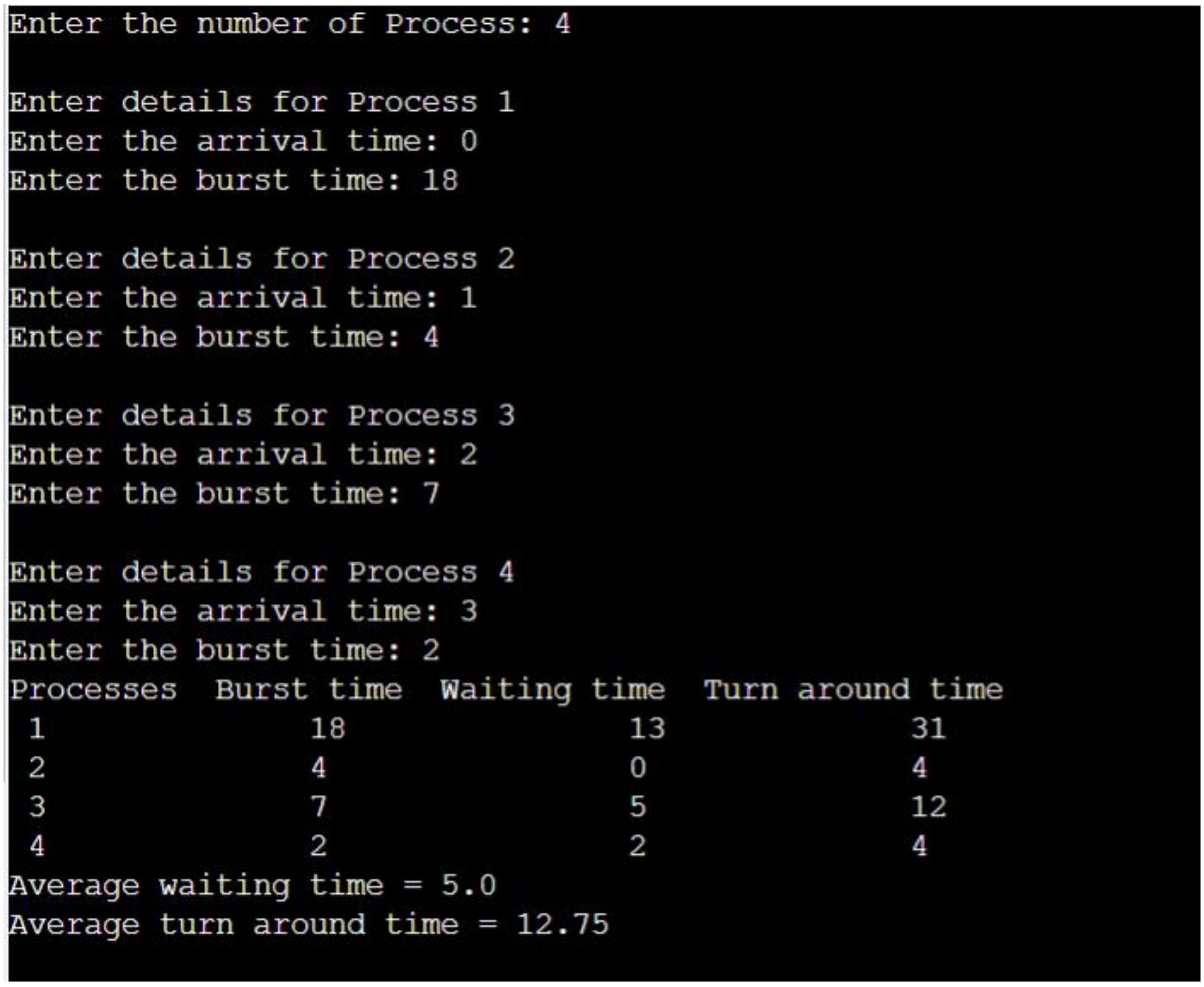
- Shortest Remaining Time First (SRTF) scheduling algorithm is basically a preemptive mode of the Shortest Job First (SJF) algorithm in which jobs are scheduled according to the shortest remaining time.
- In this scheduling technique, the process with the shortest burst time is executed first by the CPU, but the arrival time of all processes need not be the same.
- If another process with the shortest burst time arrives, then the current process will be preempted, and a newer ready job will be executed first.
- Advantages
    - Processes are executed faster than SJF, being the preemptive version of it.
- Disadvantages
    - Context switching is done a lot more times and adds to the more overhead time.
    - Like SJF, it may still lead to starvation and requires the knowledge of process time beforehand.
    - Impossible to implement in interactive systems where the required CPU time is unknown.

Code:

```
import java.util.*;

public class SRJF {
    // Method to find the waiting time for all
    // processes
    static void findWaitingTime(Process proc[], int n,
            int wt[]) {
        int rt[] = new int[n];

        // Copy the burst time into rt[]
        for (int i = 0; i < n; i++)
            rt[i] = proc[i].bt;

        int complete = 0, t = 0, minm = Integer.MAX_VALUE;
        int shortest = 0, finish_time;
        boolean check = false;

        // Process until all processes gets
        // completed
        while (complete != n) {

            // Find process with minimum
            // remaining time among the
            // processes that arrives till the
            // current time`
            for (int j = 0; j < n; j++) {
                if ((proc[j].art <= t) &&
                        (rt[j] < minm) && rt[j] > 0) {
                    minm = rt[j];
                    shortest = j;
                    check = true;
                }
            }

            if (check == false) {
                t++;
                continue;
            }

            // Reduce remaining time by one
            rt[shortest]--;

            // Update minimum
            minm = rt[shortest];
            if (minm == 0)
                minm = Integer.MAX_VALUE;

            // If a process gets completely
            // executed
            if (rt[shortest] == 0) {

                // Increment complete
                complete++;
                check = false;

                // Find finish time of current
                // process
                finish_time = t + 1;

                // Calculate waiting time
                wt[shortest] = finish_time -
                        proc[shortest].bt -
                        proc[shortest].art;

                if (wt[shortest] < 0)
                    wt[shortest] = 0;
            }
            // Increment time
            t++;
        }
    }

    // Method to calculate turn around time
    static void findTurnAroundTime(Process proc[], int n,
            int wt[], int tat[]) {
        // calculating turnaround time by adding
        // bt[i] + wt[i]
        for (int i = 0; i < n; i++)
            tat[i] = proc[i].bt + wt[i];
    }

    // Method to calculate average time
    static void findavgTime(Process proc[], int n) {
        int wt[] = new int[n], tat[] = new int[n];
        int total_wt = 0, total_tat = 0;

        // Function to find waiting time of all
```

```java
        // processes
        findWaitingTime(proc, n, wt);

        // Function to find turn around time for
        // all processes
        findTurnAroundTime(proc, n, wt, tat);

        // Display processes along with all
        // details
        System.out.println("Processes " +
                " Burst time " +
                " Waiting time " +
                " Turn around time");

        // Calculate total waiting time and
        // total turnaround time
        for (int i = 0; i < n; i++) {
            total_wt = total_wt + wt[i];
            total_tat = total_tat + tat[i];
            System.out.println(" " + proc[i].pid + "\t\t" + proc[i].bt + "\t\t
" + wt[i] + "\t\t" + tat[i]);
        }

        System.out.println("Average waiting time = " +
                (float) total_wt / (float) n);
        System.out.println("Average turn around time = " +
                (float) total_tat / (float) n);

    }

    // Driver Method
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of Process: ");
        int n = sc.nextInt();
        Process[] p = new Process[n];
        for (int i = 0; i < n; i++) {
            System.out.println("\nEnter details for Process " + (i + 1));
            System.out.print("Enter the arrival time: ");
            int at = sc.nextInt();
            System.out.print("Enter the burst time: ");
            int bt = sc.nextInt();
            p[i] = new Process(i + 1, bt, at);
        }
        findavgTime(p, p.length);
    }
}

class Process {
    int pid; // Process ID
    int bt; // Burst Time
    int art; // Arrival Time

    public Process(int pid, int bt, int art) {
        this.pid = pid;
        this.bt = bt;
        this.art = art;
    }
}
```

Output:

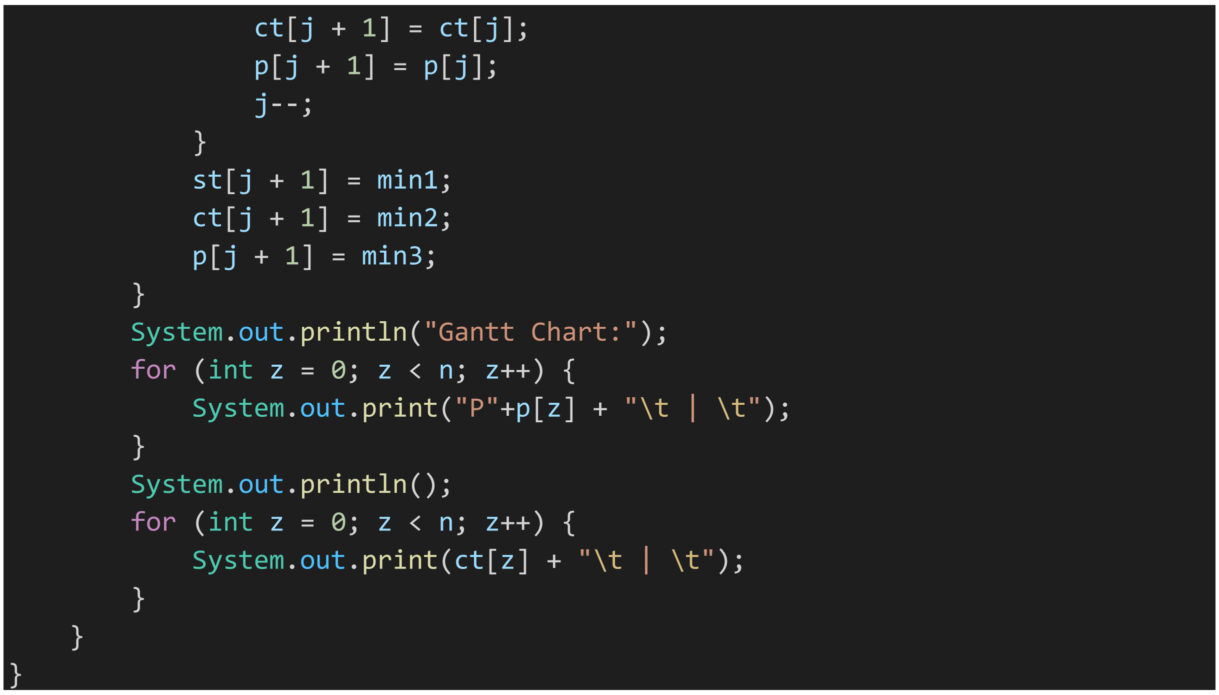


```
Enter the number of Process: 4

Enter details for Process 1
Enter the arrival time: 0
Enter the burst time: 18

Enter details for Process 2
Enter the arrival time: 1
Enter the burst time: 4

Enter details for Process 3
Enter the arrival time: 2
Enter the burst time: 7

Enter details for Process 4
Enter the arrival time: 3
Enter the burst time: 2
Processes  Burst time  Waiting time  Turn around time
 1              18             13             31
 2              4              0              4
 3              7              5              12
 4              2              2              4
Average waiting time = 5.0
Average turn around time = 12.75
```

RR

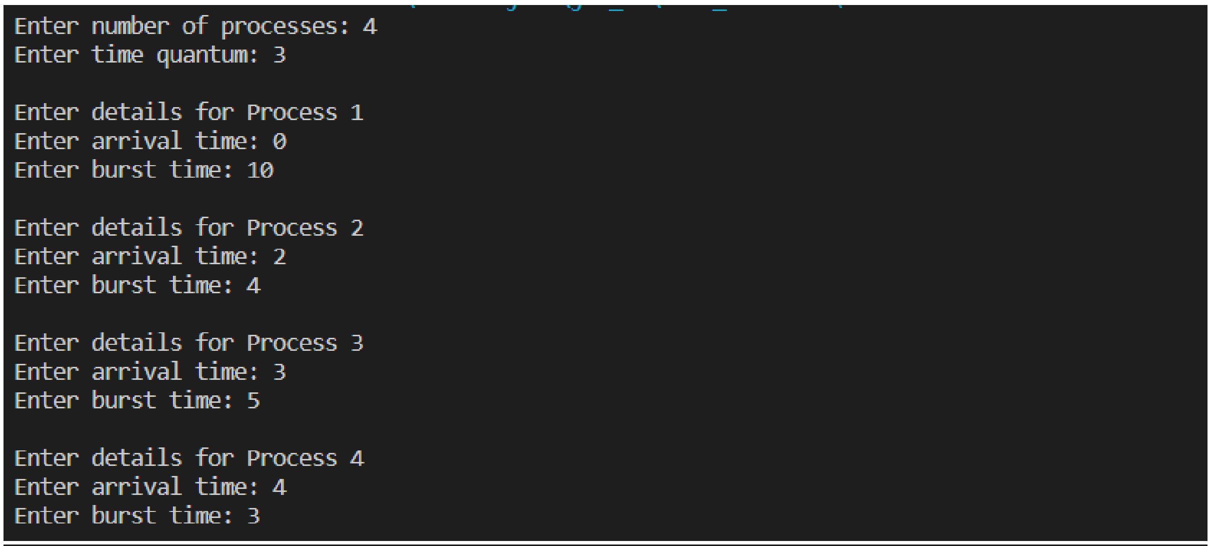
- The Round Robin algorithm is related to the First Come First Serve (FCFS) technique but implemented using a preemptive policy.
- In this scheduling algorithm, processes are executed cyclically, and each process is allocated a small amount of time called time slice or time quantum.
- The ready queue of the processes is implemented using the circular queue technique in which the CPU is allocated to each process for the given time quantum and then added back to the ready queue to wait for its next turn.
- If the process completes its execution within the given quantum of time, then it will be preempted, and other processes will execute for the given period of time.
- But if the process is not completely executed within the given time quantum, then it will again be added to the ready queue and will wait for its turn to complete its execution.
- The round-robin scheduling is the oldest and simplest scheduling algorithm that derives its name from the round-robin principle.
- In this principle, each person will take an equal share of something in turn.
- This algorithm is mostly used for multitasking in time-sharing systems and operating systems having multiple clients so that they can make efficient use of resources.
- Advantages
    - All processes are given the same priority; hence all processes get an equal share of the CPU.
    - Since it is cyclic in nature, no process is left behind, and starvation doesn't exist.
- Disadvantages
    - The performance of Throughput depends on the length of the time quantum. Setting it too short increases the overhead and lowers the CPU efficiency, but if we set it too long, it gives a poor response to short processes and tends to exhibit the same behaviour as FCFS.
    - Average waiting time of the Round Robin algorithm is often long.
    - Context switching is done a lot more times and adds to the more overhead time.
- **CODE:**

```
import java.util.Scanner;
class Queue {
    Node front, rear;
    int queueSize;
    class Node {
        int data;
        Node next;
    }
    Queue() {
        front = null;
        rear = null;
        queueSize = 0;
    }
    boolean isEmpty() {
        return (queueSize == 0);
    }
    int dequeue() {
        int data = front.data;
        front = front.next;
        if (isEmpty()) {
            rear = null;
        }
        queueSize--;
        return data;
    }
    void enqueue(int data) {
        Node oldRear = rear;
        rear = new Node();
        rear.data = data;
        rear.next = null;
        if (isEmpty()) {
            front = rear;
        } else {
            oldRear.next = rear;
        }
        queueSize++;
    }
}
class RoundRobin {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        int n, tq;
        System.out.print("Enter number of processes: ");
        n = sc.nextInt();
        System.out.print("Enter time quantum: ");
        tq = sc.nextInt();
        Queue q = new Queue();
        int p[] = new int[n]; // Process id array
        int at[] = new int[n]; // Arrival time array
        int bt[] = new int[n]; // Burst time array
        int flag[] = new int[n]; // To check if process is completed
        int f[] = new int[n]; // To check if process has arrived
        int i, j, min1, min2, min3;
        int cur_t = 0; // Current time
        int st[] = new int[n]; // Starting time of each process
        int ct[] = new int[n]; // completion time array
        int tot = 0; // To count number of processes completed
        int c = 0; // To track id of process to be scheduled next
        int b[] = new int[n]; // Copy of Burst time array
        for (i = 0; i < n; i++) {
            System.out.println("\nEnter details for Process " + (i + 1));
            p[i] = i + 1;
            System.out.print("Enter arrival time: ");
            at[i] = sc.nextInt();
            System.out.print("Enter burst time: ");
            bt[i] = sc.nextInt();
            flag[i] = 0;
            f[i] = 0;
            st[i] = -1;
        }
        for (i = 0; i < n; i++) {
            min1 = at[i];
            min2 = bt[i];
            min3 = p[i];
            j = i - 1;
            while (j >= 0 && at[j] > min1) {
                at[j + 1] = at[j];
                bt[j + 1] = bt[j];
                p[j + 1] = p[j];
                j--;
            }
            at[j + 1] = min1;
            bt[j + 1] = min2;
            p[j + 1] = min3;
        }
        for (i = 0; i < n; i++)
            b[i] = bt[i];
        while (tot < n) {
            for (i = 0; i < n; i++) {
                if ((f[i] == 0) && (flag[i] == 0) && (at[i] <= cur_t)) {
                    q.enqueue(i);
                    f[i] = 1;
                }
            }
            c = q.dequeue();
            if (st[c] == -1)
                st[c] = cur_t;
            ct[c] = cur_t;
            if (b[c] > tq) {
                ct[c] += tq;
                b[c] -= tq;
                cur_t += tq;
                for (i = 0; i < n; i++) {
                    if ((f[i] == 0) && (flag[i] == 0) && (at[i] <= cur_t)) {
                        q.enqueue(i);
                        f[i] = 1;
                    }
                }
                if (b[c] > 0)
                    q.enqueue(c);
            } else if (b[c] <= tq) {
                ct[c] += b[c];
                cur_t += b[c];
                b[c] = 0;
                for (i = 0; i < n; i++) {
                    if ((f[i] == 0) && (flag[i] == 0) && (at[i] <= cur_t)) {
                        q.enqueue(i);
                        f[i] = 1;
                    }
                }
            }
            if (b[c] == 0) {
                flag[c] = 1;
                tot++;
            }
        }
        int tat = 0; // Turnaround Time
        int wt = 0; // Waiting Time array
        float sum1 = 0, sum2 = 0; // sum1 for tat and sum2 for wt
        System.out.println("\nProcess No.\tA.T\tB.T.\tC.T.\tT.A.T.\tW.T.");
        for (i = 0; i < n; i++) {
            tat = ct[i] - at[i];
            wt = tat - bt[i];
            System.out.println("Process" + p[i] + "\t" + at[i] + "\t" + bt[i]
+ "\t" + ct[i] + "\t" + tat + "\t" + wt);
            sum1 += tat;
            sum2 += wt;
        }
        System.out.println("Average Turn Around Time: " + (sum1 / n));
        System.out.println("Average Waiting Time: " + (sum2 / n));
      for (i = 0; i < n; i++) {
            min1 = st[i];
            min2 = ct[i];
            min3 = p[i];
            j = i - 1;
            while (j >= 0 && st[j] > min1) {
                st[j + 1] = st[j];
```

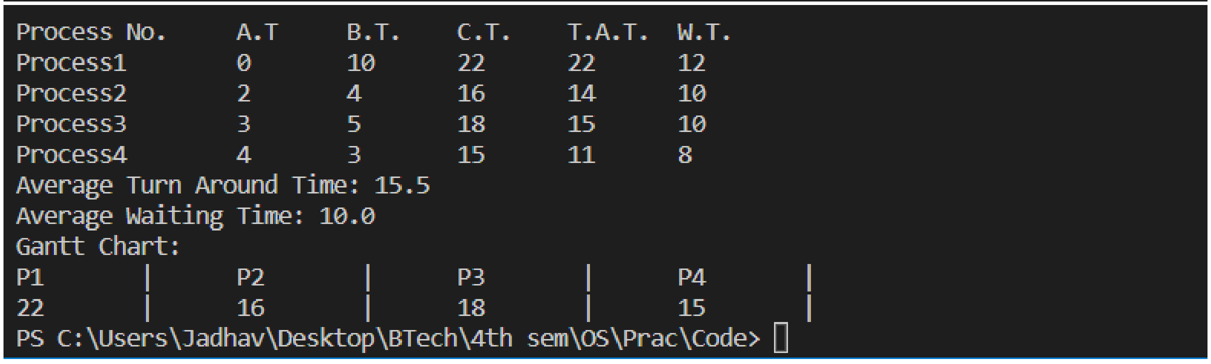


```
                ct[j + 1] = ct[j];
                p[j + 1] = p[j];
                j--;
            }
            st[j + 1] = min1;
            ct[j + 1] = min2;
            p[j + 1] = min3;
        }
        System.out.println("Gantt Chart:");
        for (int z = 0; z < n; z++) {
            System.out.print("P"+p[z] + "\t | \t");
        }
        System.out.println();
        for (int z = 0; z < n; z++) {
            System.out.print(ct[z] + "\t | \t");
        }
    }
}
```

OUTPUT:



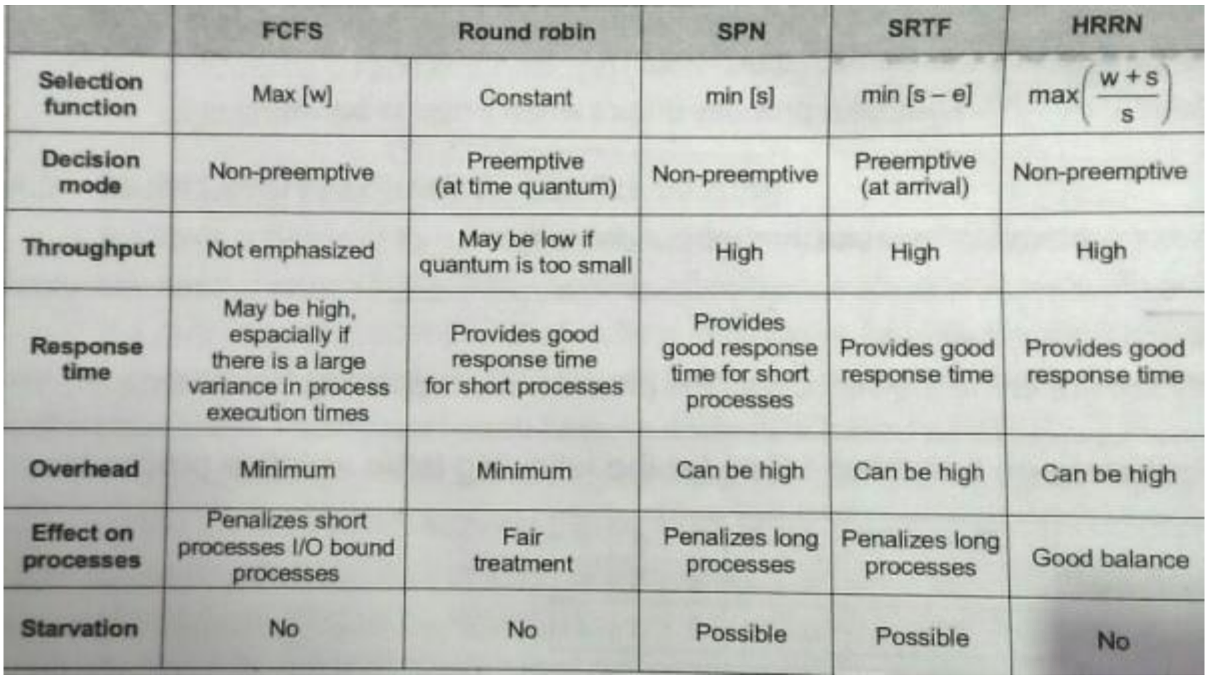
```
Enter number of processes: 4
Enter time quantum: 3

Enter details for Process 1
Enter arrival time: 0
Enter burst time: 10

Enter details for Process 2
Enter arrival time: 2
Enter burst time: 4

Enter details for Process 3
Enter arrival time: 3
Enter burst time: 5

Enter details for Process 4
Enter arrival time: 4
Enter burst time: 3
```



```
Process No.     A.T     B.T.    C.T.    T.A.T.  W.T.
Process1        0       10      22      22      12
Process2        2       4       16      14      10
Process3        3       5       18      15      10
Process4        4       3       15      11      8
Average Turn Around Time: 15.5
Average Waiting Time: 10.0
Gantt Chart:
P1       |      P2       |      P3       |      P4       |
22       |      16       |      18       |      15       |
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

CONCLUSION:

- The main goal of scheduling algorithms is to Maximize Throughput.
- The best scheduling algorithms depend on the situation, needs, and hardware and software capabilities.
- CPU scheduling is one of the basic factors for performance measure of multitasking operating system which makes a commuter system more productive by switching the CPU among the processes.
- The performance of the CPU scheduling algorithms depends on minimizing waiting time, response time, turnaround time and context switching, and maximizing CPU utilization.
- **Round Robin (RR) is the most widely used CPU scheduling algorithm in multitasking operating system**.
- The efficiency of a multitasking system comprising with Round Robin CPU scheduling relies on the selection of the optimal time quantum.
- **If the time quantum is longer, the response time of the processes becomes too high. On the other hand, the shorter time quantum raises the amount of context switch among the processes.**
- A modified CPU scheduling algorithm, called Round Robin with Dynamic Time Quantum (RRDTQ) is introduced for enhancing CPU performance using dynamic time quantum with RR. This time quantum is calculated from the burst time of the set of waiting processes in the ready queue. The experimental results show that the proposed algorithm solves the fixed time quantum problem and decreases the average waiting time and turnaround time compared to traditional RR algorithm.
- The following is a comparison of various scheduling algorithms in a tabular form:



	FCFS	Round robin	SPN	SRTF	HRRN
Selection function	Max [w]	Constant	min [s]	min [s – e]	$\max\left(\frac{w+s}{s}\right)$
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive
Throughput	Not emphasized	May be low if quantum is too small	High	High	High
Response time	May be high, espacially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance
Starvation	No	No	Possible	Possible	No