1. Theory :-

The aim is to calculate the average waiting time. Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm. First in, first out (FIFO), also known as first come, first serve (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.

Algorithm -

Step 1 : Start the process

Step 2 : ~~Find~~ Input the processes along with their burst time (bt).

Step 3 : Find waiting time (wt) for all processes.

Step 4 : As first process that comes need not to wait so waiting time for process 1 will be 0 i.e., wt[0] = 0.

Step 5 : Find waiting time for all other processes i.e., for process i → wt[i] = bt[i-1] + wt[i-1].

Step 6 : Find average waiting time = total waiting_time / no_of_processes.

Step 7 : Find turnaround time = waiting_time + burst time for all processes.

Step 8: Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

Input :-

Enter the total number of JOB: 3
Enter the Process Id of 1th Job: 1
Enter the Arrival Time of 1th Job: 0
Enter the Brust Time of 1th Job: 10
Enter the Process Id of 2th Job: 2
Enter the Arrival Time of 2th Job: 0
Enter the Brust Time of 2th Job: 5
Enter the Process Id of 3th Job: 3
Enter the Arrival Time of 3th Job: 0
Enter the Brust Time of 3th Job: 8

1. Code:-

```
#include <stdio.h>

struct job {
    int processId, arrivalTime, BrustTime, CompletionTime,
    TurnAroundTime, WaitingTime;
};

void MergeSort(struct job j[], int l, int h);
void Merge(struct job j[], int l, int mid, int h);
void compTime(struct job j[], int h);

int main(){
    int n, i;
    printf("Enter the total number of JOB: ");
    scanf("%d", &n);
    struct job j[n];
    for(i=0; i<n ; i++){
        printf("Enter the Process Id of %dth job: ",
            i+1);
        scanf("%d", &j[i].processId);
        printf("Enter the Arrival Time of %dth job: ",
            i+1);
        scanf("%d", &j[i].arrivalTime);
        printf("Enter the Brust Time of %dth job: "i+1);
        scanf("%d", &j[i].BrustTime);
    }
    printf("The Job table that you created was:\n");
    printf("Processid ArrivalTime BrustTime\n");
```

```
for(i=0; i<n; i++){
        printf("%d    %d    %d\n",
        j[i].processId, j[i].arrivalTime, j[i].BrustTime);

compTime(j, n-1);
printf("ProcessId  Arival Time  Brust Time
      Completion Time  Turn Around Time
      Waiting Time\n");
for(i=0; i<n; i++){
      printf("%d  %d  %d  %d  %d  %d\n",
            j[i].processId, j[i].arrivalTime, j[i].
            BrustTime, j[i].~~Turn~~CompletionTime,
            j[i].TurnAroundTime, j[i].WaitingTime};

return 0;

void compTime(struct job[], int k)
{
      int i; ~~totalWut=0,totalTat~~=0;
      for(i=0; i<=k; i++)
      {
            if(j[i-1].CompletionTime >=j[i].arrivalTime
              && i!=0){
                j[i].CompletionTime = j[i].BrustTime +
                                    j[i-1].CompletionTime};

              }
            else{
              j[i].CompletionTime = j[i].arrivalTime +
                                    j[i].BrustTime;

            }
```

```
for(i=0;i<=n;i++){
    j[i].TurnaroundTime = j[i].CompletionTime-
                          j[i].arrivalTime;
    j[i].TurnaroundTime=j[i].
    j[i].WaitingTime = j[i].TurnaroundTime -
                       j[i].BrustTime;
}
for (i=0; i<n; i++)
{
    total_wt = total_wt + j[i].waitingTime;
    total_tat = total_tat + j[i].TurnaroundTime;
    printf("Average waiting time=%f\n",
           (float)total_wt/(float)n;
    printf("Average turn around time= %f \n",
           (float) total_tat /(float)n;
}
}
}
```

Output :-

The Job table that you created was:

Process id	Arrival Time	Brust Time
1	0	10
2	0	5
3	0	8

Process Id	Arrival Time	Brust Time	completion Time
1	0	10	10
2	0	5	15
3	0	8	23

TurnAround Time	waiting Time
10	0
15	10
23	15

Average waiting time = 8.333 33

Average turn around time = 16

2. Theory :-

The aim is to calculate the average waiting time. Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

Algorithm :-

Step 1: Start the process

Step 2: Sort all the process according to the arrival time.

Step 3: Then select that process which has minimum arrival time and minimum Burst time.

Step 4: After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

Step 5: Completion time at which process completes its execution

Step 6 : Turn Around Time = Completion Time - Arrival Time

Step 7 : Waiting Time = Turn Around Time - Burst Time

Input:-

Enter the number of process: 3

Enter process name, arrival time & execution time:

2 5 7

Enter process name, arrival time & execution time:

3 6 14

Enter process name, arrival time & execution time:

4 7 12

Code:-

2.
```
#include <stdio.h>

struct util {
        int id;
        int at;
        int bt;
        int ct;
        int tat;
        int wt;
    }
ar[100];
struct utill {
        int p_id, bt1;
};
untill range;
utill tr[100];
int rp[100];

bool cmp(util a, util b)
{
        if(a.at == b.at)
                return a.id < b.id;
        return a.at < b.at;
}
```

```
void update(int node, int st, int end, int ind,
            int idl, int b_t)
{
    if(st == end){
        tr[node].p_id = idl;
        tr[node].bt1 = b_t;
        return;
    }
    int mid = (st + end)/2;
    if(ind <= mid
        update(2*node, st, mid, ind,
               id1, b_t);
    else
        update(2*node+1, mid+1, end, ind,
               id1, b_t);
    if(tr[2*node].bt1 < tr[2*node+1].bt1){
        tr[node].bt1 = tr[2*node].bt1;
        tr[node].p_id = tr[2*node].p_id;
    }
    else{
        tr[node].bt1 = tr[2*node+1].bt1;
        tr[node].p_id = tr[2*node+1].p_id;
    }
}
```

```
util query(int node, int st, int end, int lt, int rt)
{
    if (end < lt || st > rt)
        return range;
    if (st >= lt && end <= rt)
        return tr[node];
    int mid = (st + end)/2;
    util lm = query(2*node, st, mid, lt, rt);
    util rm = query(2*node+1, mid+1, end,
                    lt, rt);
    if (lm.st1 < rm.st1)
        return lm;
    return rm;
}

void non-preemptive-sjf(int n)
{
    int counter = n;
    int upper_range = 0;
    int tm = min(INT_MAX, ar[upper_range+1].at);
    while(counter){
        for(; upper_range <= n){
            upper_range++;
        if (ar[upper_range].at1 > tm || upper_range>n)
        {
            upper_range--;
            break;      }
```

```
        update(1, 1, n, upper_range, ar[upper_range].id,
            ar[upper_range].bt);
    }
    until res = query (1, 1, n, 1, upper_range);
    if(res.bt1 != INT_MAX){
        counter--;
        int index = mp[res.p_id];
        tm += (res.bt1);

        ar[index].ct = tm;
        ar[index].tat = ar[index].ct - ar[index].at;
        ar[index].wt = ar[index].tat - ar[index].bt;
        update(1, 1, n, index, INT_MAX, INT_MAX);

    }
    else{
        tm = ar[upper_range+1].at;
    }

}

void execute(int n)
{
    int i;
    sort(ar+1, ar+n+1, cmp);
    for(i=1; i<=n; i++)
        mp[ar[i].id] = i;

    nonpreemptive_sjf(n);
}
```

```
void print(int n)
{
    int i;
    printf("Process Id  Arrival Time  Burst Time
            Completion Time  Turn Around Time
            Waiting Time\n");
    for (i = 1; i<=n; i++){
    printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t
            %d\t\t%d\n", ar[i].id,
            ar[i].at, ar[i].bt, ar[i].ct, ar[i].tat,
            ar[i].wt);
    }
}
int main()
{
    int n=5,i;
    range.pid = INTMAX;
    range.bt1 = INTMAX;

    for(i=1; i<=100 ; i++){
        tr[i].pid = INT_MAX;
        tr[i].bt1 = INT_MAX;

    }

ar[1].at =1;
ar[1].bt =7;
ar[1].id = 1;
```

```
ar[2].at=2;
ar[2].bt=5;
ar[2].id=2;

ar[3].at=3;
ar[3].bt=1;
ar[3].id=3;

ar[4].at=4;
ar[4].bt=2;
ar[4].id=4;

ar[5].at=5;
ar[5].bt=8;
ar[5].id=5;

execute(n);
print(n);
return 0;
}
```

Output:-

Pname	arrivaltime	executiontime	waitingtime	tatime
2	5	7	0	7
4	7	12	5	17
3	6	14	18	32

Average waiting time is : 7.666667
Average turnaround time is : 18.666666

3. Theory :-

The aim is to find the average waiting time. The algorithm is the preemptive version of SJF scheduling. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Algorithm :-

Step 1 : Start the process

Step 2 : Input the process id, arrival time, Brust time.

Step 3 : Start from the the smallest arrival time and brust time.

Step 4 : If any job arrives later but with lesser burst time ~~of the~~ then preempt the previous job and start processing the current job.

Step 5 : Repeat the process till all the jobs are completed.

Code :-

3.
```
#include <stdio.h>

struct Process{
        int pid;
        int bt;
        int art;
};

void findWaitingTime(Process proc[], int n,
                     int wt)
{
        int rt[n],i;
        for(i=0; i<n; i++)
                rt[i] = proc[i].bt;

        int complete = 0, t = 0, minm = INT_MAX;
        int shortest = 0, finishtime;
        bool check = false;

        while(complete != n) {
                for(i = 0; i<n; i++){
                        if((proc[i].art <= t)&&
                          (rt[i] < minm)&& rt[i]>0){
                                minm = rt[i];
                                shortest = i;
                                check = true;
                        }
                }
```

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

rt[shortest]--;
minm = rt[shortest];
if(minm == 0)
        minm = INT_MAX;

if (rt[shortest] == 0){
        complete++;
        check = false;

        finish_time = t+1;
        wt[shortest] = finish_time -
                proc[shortest].bt -
                proc[shortest].art;

        if(wt[shortest] < 0)
                wt[shortest] = 0;

        }

t++;

}

}
```

```
void findTurnAroundTime(Process proc[], int n,
                        int wt[], int tat[])
{
    int i;
    for(i = 0; i<n ; i++)
        tat[i] = proc[i].bt + wt[i];
}

void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0,
        total_tat = 0, i;

    findWaitingTime(proc, n, wt);
    findTurnAroundTime(proc, n, wt, tat);
    printf("P\t\t BT\t\t WT\t\t TAT\t\t\n");
    for(i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf("%d\t\t %d\t\t %d\t\t %d",
               proc[i].pid, proc[i].bt,
               wt[i], tat[i]);
    }
```

```
    printf("\n Average waiting time = %f ",
            (float) totalwt / (float) n);

    printf("\n Average turn around time = %f",
            (float) total_tat / (float) n);

}

int main ()
{
    Process proc[] = { {1, 6, 2}, {2, 2, 5},
                       {3, 8, 1}, {4, 3, 0}, {5, 4, 4}};

    int n = sizeof(proc) / sizeof(proc[0]);

    findavgTime(proc, n);
    return 0;

}
```

4. Theory :-

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on FCFS basis.

Algorithm :-

Step 1 :- Start the process.

Step 2 :- first input the processes with their arrival time, burst time and priority.

Step 3 :- first process will schedule, which have the lowest arrival time. if two or more processes will have lowest arrival time, then whoever has higher priority will schedule first.

Step 4 :- Now further processes will be schedule according to the arrival time and priority of the process. If two process priority are same then sort according to process number.

Step 5 : Once all the processes have been arrived, we can schedule them based on their priority.

Input :-

Enter the number of process

3

Enter the Arrival time, Burst time and priority of the process

AT	BT	PR
0	2	1
1	3	3
2	4	2

Code:-

4.
```
#include<stdio.h>
struct process
{
    int id, WT, AT, BT, TAT, PR;
};
struct process a[10];
void swap(int *a, int *c)
{
    int tem;
    tem = *c;
    *c = *b;
    *b = temp;
}
int main()
{
    int n, check_ar = 0, i, j;
    int Cmp_time = 0;
    float Total_WT = 0, Total_TAT = 0, Avg_WT, Avg_TAT;
    printf("Enter the number of process \n");
    scanf("%d", &n);
    printf("Enter the Arrival time, Burst time and
            priority of the process \n");
    printf("AT  BT  PR \n");
```

```
for (i=0; i<n; i++)
{
    scanf ("%d%d%d", &a[i].AT, &a[i].BT, &a[i].PR);
    a[i].id = i+1;

    if (i==0)
        check_ar = a[i].AT;

    if (check_ar != a[i].AT)
        check_ar = 1;

}

if (check_ar != 0)
{
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {
            if (a[j].AT > a[j+1].AT)
            {
                swap(&a[j].id, &a[j+1].id);
                swap(&a[j].AT, &a[j+1].AT);
                swap(&a[j].BT, &a[j+1].BT);
                swap(&a[j].PR, &a[j+1].PR);
            }
        }
    }
}
```

```
if (check_ar != 0)
{
    a[0].WT = a[0].AT;
    a[0].TAT = a[0].BT - a[0].AT;

    Cmp_time = a[0].TAT;
    Total_WT = Total_WT + a[0].WT;
    Total_TAT = Total_TAT + a[0].TAT;

    for (i = 1; i<n; i++)
    {
        int min = a[i].PR;
        for (j = i+1;  j<n; j++)
        {
            if(min > a[j].PR && a[j].AT <= Cmp_time)
            {
                min = a[j].PR;
                swap(&a[i].id, &a[j].id);
                swap(&a[i].AT, &a[j].AT);
                swap(&a[i].BT, &a[j].BT);
                swap(&a[i].PR, &a[j].PR);
            }
        }
    a[i].WT = Cmp_time - a[i].AT;
    Total_WT = Total_WT + a[i].WT;

    Cmp_time = Cmp_time + a[i].BT;
```

```
        a[i].TAT = Cmp_time - a[i].AT;
        Total_TAT = Total_TAT + a[i].TAT;
    }
}
else
{
    for(i=0; i<n; i++)
    {
        int min = a[i].PR;
        for(j = i+1; j<n; j++)
        {
            if(min > a[j].PR && a[j].AT <= Cmp_time)
            {
                min = a[j].PR;
                swap(&a[i].id, &a[j].id);
                swap(&a[i].AT, &a[j].AT);
                swap(&a[i].BT, &a[j].BT);
                swap(&a[i].PR, &a[j].PR);
            }
        }
    }
    a[i].WT = Cmp_time - a[i].AT;
    Cmp_time = Cmp_time + a[i].BT;
    a[i].TAT = Cmp_time - a[i].AT;
    Total_WT = Total_WT + a[i].WT;
    Total_TAT = Total_TAT + a[i].TAT;
}
```

```
}

Avg_WT = Total_WT/n;
Avg_TAT = Total_TAT /n;

printf("The process are\n");
printf("ID  WT TAT\n");
for (i=0; i<n; i++)
{
    printf("%d\t%d\t%d\n",a[i].id,a[i].WT,
        a[i].TAT);
}

printf("Avg waiting time is : %f\n", Avg_WT);
printf("Avg turn around time is: %f", Avg_TAT);
return 0;
}
```

Output :-

The process are

ID	WT	TAT
1	0	2
3	1	4
2	3	7

Average waiting time is : 1.33333

Average turn around time is : 4.33333

5. Theory:-

In preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The one with the highest priority among all the available processes will be given the CPU next.

- Algorithm:-

Step1:- Start the process

Step2:- Enter the arrival time and burst time and priority of the process in a array.

Step3:- Arrang the arrays according to their arrival time

Step4:- Start the operation. As soon as a high priority job comes, take the CPU from the current job and give the high priority job.

Step5:- Repeat step 4 till all the job are processed.

Step6: Stop the process.

Input :-

Enter the number of the process

3

Enter the arrival time, burst time and priority of the process

AT	BT	PT
0	3	3
1	5	1
2	2	2

5. Code:-

```
#include <stdio.h>
struct process
{
    int WT, AT, BT, TAT, PT;
};
struct process a[10];
int main()
{
    int n, temp[10], t, count=0, shortp, i;
    float total_WT=0, total_TAT=0, Avg_WT, Avg_TAT;
    printf("Enter the number of the processes\n");
    scanf("%d", &n);
    printf("Enter the arrival time, burst time and
            priority of the process\n");
    printf("AT  BT  PT\n");
    for(i=0; i<n; i++)
    {
        scanf("%d %d %d", &a[i].AT, &a[i].BT,
                &a[i].PT);

        temp[i] = a[i].BT;

    }
a[9].PT=10000;
```

```
for(t=0; count!=n; t++)
{
    short_p=9;
    for(i=0; i<n; i++)
    {
        if(a[short_p].PT>a[i].PT && a[i].AT<=t
            && a[i].BT>0)
        {
            short_p=i;
        }
    }
    a[short_p].BT= a[short_p].BT-1;

    if(a[short_p].BT==0)
    {
        count++;
        a[short_p].WT= t+1- a[short_p].AT- temp[short_p];
        a[short_p].TAT=t+1- a[short_p].AT;

        total_WT = total_WT+a[short_p].WT;
        total_TAT = total_TAT+a[short_p].TAT;
    }
}

Avg_WT= total_WT/n;
Avg_TAT=total_TAT/n;
```

```
printf("ID WT TAT\n");
for (i=0; i<n; i++)
{
    printf("%d %d\t %d\n", i+1, a[i].WT, a[i].TAT);
}
printf("Avg waiting time of the process is %f\n",
        Avg_WT);
printf("Avg turn Around time of the process is
        %f\n", Avg_TAT);
return 0;
}
```

Output :-

ID	WT	TAT
1	7	10
2	0	5
3	4	6

Avg waiting time of the process is 3.666667
Avg turn around time of the process is 7.000000

6. Theory :-

The aim of is to calculate the average waiting time. There will be a time since, each process should be executed the time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual time and increment it by time-slot and the loop continues until all the processes are completed.

Algorithm :-

Step 1: Start the process

Step 2: Accept the number of processes in the ready Q, ~~assign the process id and accept the CPU burst time~~

~~Step 3:~~ ~~for~~ and time quantum (or) time slice.

Step 3: for each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process(n) = burst time process(n) / time slice.

Step 5: If the burst time is less than the time slice then the no. of time slices = 1.

Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process (n) = waiting time of process (n+1) + burst time of process (n-1) + the time difference in getting the CPU from process (n-1)

(b) Turnaround time for process (n) = waiting time of process (n) + burst time of process (n) + the difference in getting CPU from process (n).

Step 7: Calculate
(c) Average waiting time = Total waiting time / Number of process
(d) Average Turnaround time = Total Turnaround Time / Number of process step

Step 8: Stop the process.

Input:-

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

Code :-

6.
```
#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 awt, atat;
printf("Total number of process in the system: ");
scanf("%d", &NOP);
y = NOP;

for(i=0; i<NOP; i++)
{
    printf("\nEnter the Arrival and Burst time of the
        Processes [%d]\n", i+1);
    printf("Arrival time is: \t");
    scanf("%d", &at[i]);
    printf("\n Burst time is: \t");
    scanf("%d", &bt[i]);
    temp[i] = bt[i];
}

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

```
printf("\nProcess No[%d]\t\t %d\t\t\t\t %d\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;
  }
}
avgwt = wt * 1.0/NOP;
avgtat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}
```

Output:-

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 : 14.750000
Average Waiting Time : 23.250000

7. Theory :-

Semaphore in operating system, Inter Process Communication Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores.

A semaphore S is an integer variable that can be accessed only through two standard operations: wait() and signal(). The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

Algorithm :-

Producer -

```
do {
    // produce an item
    wait(empty);
    wait(mutex);
    //place in buffer
    signal(mutex);
    signal(full);
} while(true).
```

Consumer:-

```
do {
    wait(full);
    wait(mutex);
    // remove item from buffer
    signal(mutex);
    signal(full);
    // consumes item
} while(true)
```

Input:-

Enter 1 for producer
Enter 2 for consumer
Enter 3 to exit
Enter your choice : 1
The buffer produced by the producer is 1
Enter your choice : 1
The buffer produced by the producer is 2
Enter your choice : 2
The buffer ~~produced~~ consumed by the producer is 2
Enter your Choice : 2
The buffer consumed by the producer is 1
Enter your Choice : 2
The Buffer is empty
Enter your Choice : 3

Code:-

7.
```
#include <stdio.h>
int full = 0;
int mutex = 1;
int empty = 10, x=0;

void producer (){
    mutex--;
    full++;
    empty--;
    x++;
    printf("The buffer produced by the producer is
        %d\n", x);
    mutex++;
}

void consumer (){
    mutex--;
    full--;
    empty++;
    printf("The buffer consumed by the producer is
        %d\n", x);
    x--;
    mutex++;
}

int main (){
    int n, i;
```

```
printf("Enter 1 for producer\n"
       "Enter 2 for consumer\n"
       "Enter 3 to exit\n");

for (i=1; i>0; i++){
    printf("Enter your choice: ");
    scanf("%d",&n);
    switch(n){

        case (1):
            if(mutex==1 && empty!=0)
                producer();
            else printf("The Buffer is full\n");
            break;

        case (2):
            if(mutex==1 && full!=0)
                consumer();
            else printf("The Buffer is empty\n");
            break;

        case (3):
            exit(0);
            break;
    }

}
return 0;
}
```

Output :-

P	BT	WT	TAT
1	6	7	13
2	2	0	2
3	8	14	22
4	3	0	3
5	4	2	6

Average waiting time = 4.6
Average turn around time = 9.2