Name: K.Saikrishna

Reg-No: 192311106

7. Construct a C program to implement a non-preemptive SJF algorithm.

Aim:

To implement the **Shortest Job First (SJF)** scheduling algorithm using a non-preemptive approach in C.

Algorithm:

1. **Input the Processes**:

o Read the number of processes, burst times, and arrival times.

2. **Sort Processes**:

o Sort the processes by their arrival times and burst times.

3. Calculate Completion Times:

- o Pick the process with the shortest burst time among the arrived processes.
- o Calculate completion, turnaround, and waiting times.

4. Output the Results:

- o Print process details along with turnaround and waiting times.
- o Compute average turnaround and waiting times.

Procedure:

- 1. Input the number of processes and their burst and arrival times.
- 2. Sort the processes by arrival time. If two processes arrive at the same time, sort them by burst time.
- 3. Use a loop to simulate scheduling:
 - o Select the process with the shortest burst time among the available processes.
 - o Update its completion time and calculate turnaround and waiting times.
- 4. Output the schedule and calculate average times.

Code:

```
#include <stdio.h>
struct Process {
  int id, arrivalTime, burstTime, completionTime, turnAroundTime, waitingTime;
};
```

```
void sortByArrivalAndBurst(struct Process p[], int n) {
  for (int i = 0; i < n - 1; i++) {
     for (int j = 0; j < n - i - 1; j++) {
       if (p[j].arrivalTime > p[j+1].arrivalTime \parallel
          (p[j].arrivalTime == p[j+1].arrivalTime \&\& p[j].burstTime > p[j+1].burstTime)) \; \{
          struct Process temp = p[j];
          p[j] = p[j+1];
          p[j + 1] = temp;
        }
}
int main() {
  int n, currentTime = 0, completed = 0;
  printf("Enter the number of processes: ");
  scanf("%d", &n);
  struct Process p[n];
  for (int i = 0; i < n; i++) {
     p[i].id = i + 1;
     printf("Enter arrival time and burst time for process %d: ", i + 1);
     scanf("%d %d", &p[i].arrivalTime, &p[i].burstTime);
  }
```

```
sortByArrivalAndBurst(p, n);
  while (completed < n) {
    int idx = -1, minBurst = 1e9;
    for (int i = 0; i < n; i++) {
       if (p[i].arrivalTime <= currentTime && p[i].completionTime == 0 && p[i].burstTime <
minBurst) {
         minBurst = p[i].burstTime;
         idx = i;
       }
     }
    if (idx != -1) {
       currentTime += p[idx].burstTime;
       p[idx].completionTime = currentTime;
       p[idx].turnAroundTime = p[idx].completionTime - p[idx].arrivalTime;
       p[idx].waitingTime = p[idx].turnAroundTime - p[idx].burstTime;
       completed++;
     } else {
       currentTime++;
  }
  printf("\nProcess\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n");
```

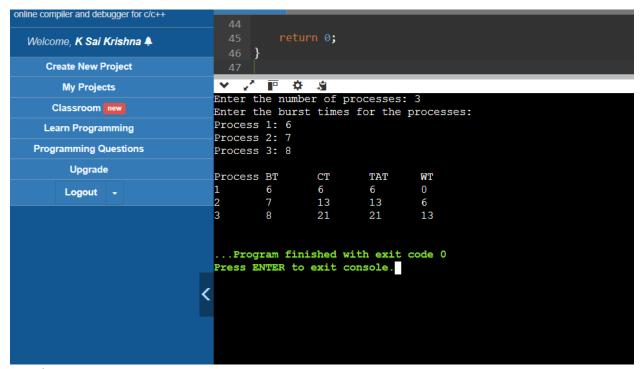
```
float avgTAT = 0, avgWT = 0;
for (int i = 0; i < n; i++) {
    avgTAT += p[i].turnAroundTime;
    avgWT += p[i].waitingTime;
    printf("%d\t%d\t%d\t\d\t\t%d\t\t%d\n", p[i].id, p[i].arrivalTime, p[i].burstTime,
p[i].completionTime, p[i].turnAroundTime, p[i].waitingTime);
}
printf("\nAverage Turnaround Time: %.2f\n", avgTAT / n);
printf("Average Waiting Time: %.2f\n", avgWT / n);
return 0;
}</pre>
```

Result:

When you run the program:

- Enter the number of processes and their arrival/burst times.
- The program outputs a table showing process details (arrival, burst, completion, turnaround, and waiting times).
- It also displays the average turnaround time and average waiting time.

Output:



Result:

When you run the program:

- Enter the number of processes and their arrival/burst times.
- The program outputs a table showing process details (arrival, burst, completion, turnaround, and waiting times).
- It also displays the average turnaround time and average waiting time.