Name : Sohag Ali

ID : 0242220005101426

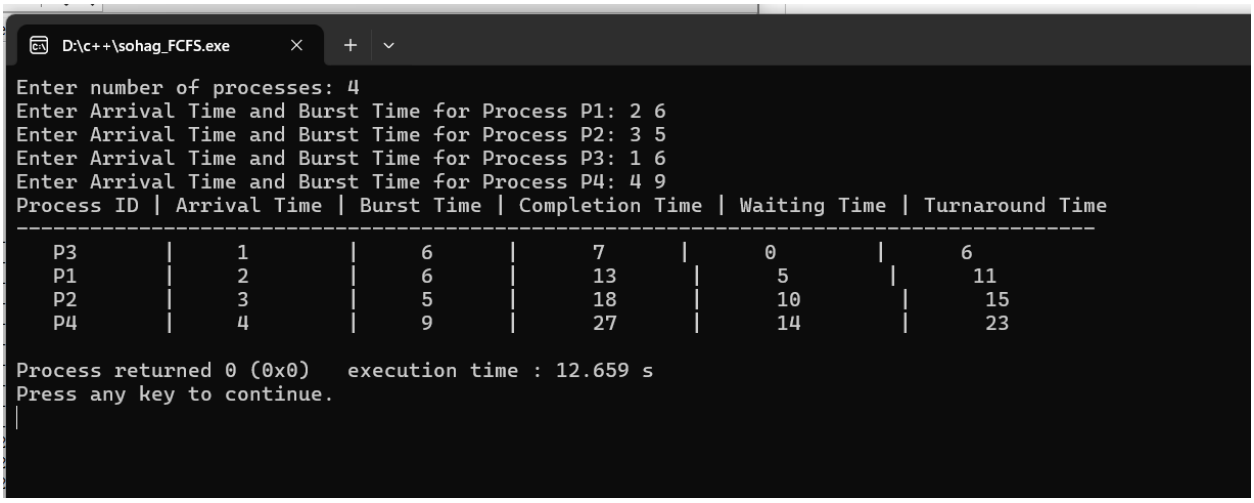
Section : 63_B

Q1. FCFS with idle time and arrival time

```
#include <iostream>
#include <algorithm>
using namespace std;
struct Process {
int id;
int arrivalTime;
int burstTime;
int completionTime;
int waitingTime;
int turnAroundTime;
};
void findCompletionTime(Process processes[], int n) {
sort(processes, processes + n, [](Process a, Process b) {
return a.arrivalTime < b.arrivalTime;
});
int currentTime = 0;
for (int i = 0; i < n; i++) {
if (currentTime < processes[i].arrivalTime) {
        currentTime = processes[i].arrivalTime;
    }
```

```
        processes[i].completionTime = currentTime + processes[i].burstTime;
        currentTime = processes[i].completionTime;

        processes[i].turnAroundTime = processes[i].completionTime - processes[i].arrivalTime;
        processes[i].waitingTime = processes[i].turnAroundTime - processes[i].burstTime;
    }
}

void displayResults(Process processes[], int n) {
    cout << "Process ID | Arrival Time | Burst Time | Completion Time | Waiting Time | Turnaround Time\n";
    cout << "-----------------------------------------------------------------------------------------------------\n";
    for (int i = 0; i < n; i++) {
        cout << "   P" << processes[i].id << "       |     "
             << processes[i].arrivalTime << "        |     "
             << processes[i].burstTime << "      |      "
             << processes[i].completionTime << "      |      "
             << processes[i].waitingTime << "        |      "
             << processes[i].turnAroundTime << "\n";
    }
}

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;
```

```
Process processes[n];

for (int i = 0; i < n; i++) {

processes[i].id = i + 1;

cout << "Enter Arrival Time and Burst Time for Process P" << i + 1 << ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime;

}

findCompletionTime(processes, n);

displayResults(processes, n);

return 0;

}
```



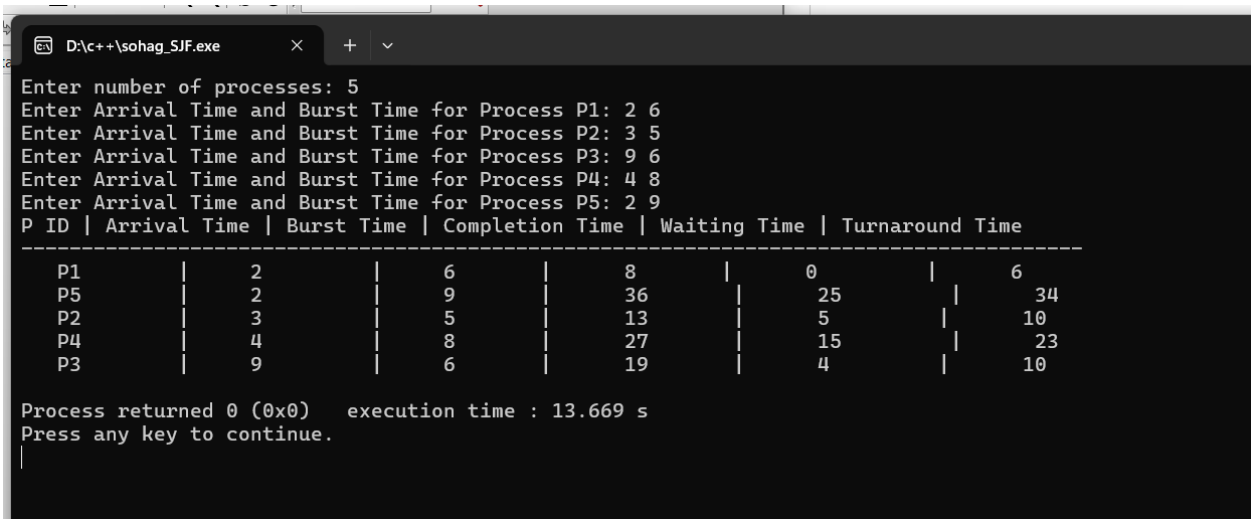
Q2. SJF with arrival time and handle idle issue

```
#include <iostream>

#include <climits>

#include <algorithm>


using namespace std;


struct Process {
```

```
    int id;
    int arrivalTime;
    int burstTime;
    int completionTime;
    int waitingTime;
    int turnAroundTime;
    bool isCompleted;
};

bool compareArrival(Process a, Process b) {
    return a.arrivalTime < b.arrivalTime;
}

void findCompletionTime(Process processes[], int n) {
    int currentTime = 0;
    int completed = 0;

    while (completed < n) {
        int idx = -1;
        int minBurstTime = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (!processes[i].isCompleted && processes[i].arrivalTime <= currentTime) {
                if (processes[i].burstTime < minBurstTime) {
                    minBurstTime = processes[i].burstTime;
                    idx = i;
```

```
            }
        }
    }


    if (idx != -1) {
        processes[idx].isCompleted = true;
        processes[idx].completionTime = currentTime + processes[idx].burstTime;
        currentTime = processes[idx].completionTime;
        processes[idx].turnAroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
        processes[idx].waitingTime = processes[idx].turnAroundTime - processes[idx].burstTime;
        completed++;
    } else {

        currentTime++;
    }
  }
}

void displayResults(Process processes[], int n) {
  cout << "P ID | Arrival Time | Burst Time | Completion Time | Waiting Time | Turnaround
Time\n";
  cout << "----------------------------------------------------------------------------------------------\n";
  for (int i = 0; i < n; i++) {
    cout << "   P" << processes[i].id << "        |     "
         << processes[i].arrivalTime << "         |     "
```

```
        << processes[i].burstTime << "       |      "
        << processes[i].completionTime << "       |      "
        << processes[i].waitingTime << "         |      "
<< processes[i].turnAroundTime << "\n";
}
}
int main() {
int n;
cout << "Enter number of processes: ";
cin >> n;
Process processes[n];
for (int i = 0; i < n; i++) {
processes[i].id = i + 1;
processes[i].isCompleted = false;
cout << "Enter Arrival Time and Burst Time for Process P" << i + 1 << ": ";
cin >> processes[i].arrivalTime >> processes[i].burstTime;
}
sort(processes, processes + n, compareArrival);
findCompletionTime(processes, n);
displayResults(processes, n);
return 0;
}
```



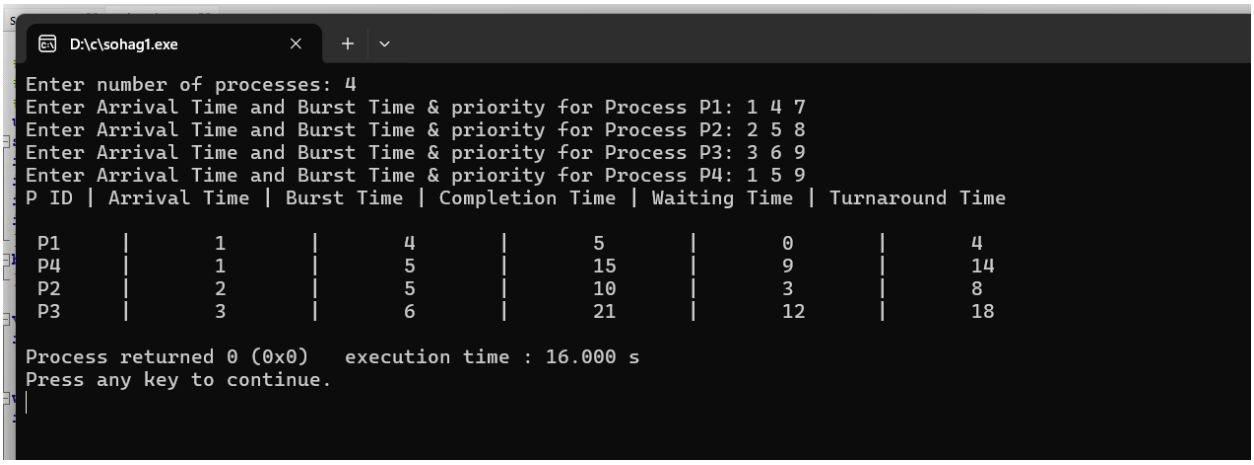
Q3. Priority Scheduling

```
#include <iostream>

#include <climits>

#include <algorithm>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int completionTime;

int waitingTime;

int turnAroundTime;

bool isCompleted;

int priority;

};

bool compareArrival(Process a, Process b) {

return a.arrivalTime < b.arrivalTime;

}
```

```
void findCompletionTime(Process processes[], int n) {
    int currentTime = 0;
    int completed = 0;

    while (completed < n) {
        int idx = -1;
        int highpriority = INT_MAX;


        for (int i = 0; i < n; i++) {
            if (!processes[i].isCompleted && processes[i].arrivalTime <= currentTime) {
                if (processes[i].priority < highpriority) {
                    highpriority = processes[i].priority;
                    idx = i;
                }
            }
        }


        if (idx != -1) {
            processes[idx].isCompleted = true;
            processes[idx].completionTime = currentTime + processes[idx].burstTime;
            currentTime = processes[idx].completionTime;
            processes[idx].turnAroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
```

```
            processes[idx].waitingTime = processes[idx].turnAroundTime - processes[idx].burstTime;
            completed++;
        } else {

            currentTime++;
        }
    }
}

void displayResults(Process processes[], int n) {
    cout << "P ID | Arrival Time | Burst Time | Completion Time | Waiting Time | Turnaround Time\n";
    cout << "-----------------------------------------------------------------------------------------------------\n";
    for (int i = 0; i < n; i++) {
        cout << "   P" << processes[i].id << "        |     "
             << processes[i].arrivalTime << "         |     "
             << processes[i].burstTime << "        |      "
             << processes[i].completionTime << "       |      "
             << processes[i].waitingTime << "         |      "
             << processes[i].turnAroundTime << "\n";
    }
}

int main() {
    int n;
    cout << "Enter number of processes: ";
```

```
    cin >> n;

    Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        processes[i].isCompleted = false;
        cout << "Enter Arrival Time and Burst Time & priority for Process P" << i + 1 << ": ";
cin >> processes[i].arrivalTime >> processes[i].burstTime>>processes[i].priority;
}
sort(processes, processes + n, compareArrival);
findCompletionTime(processes, n);
displayResults(processes, n);
return 0;
}
```

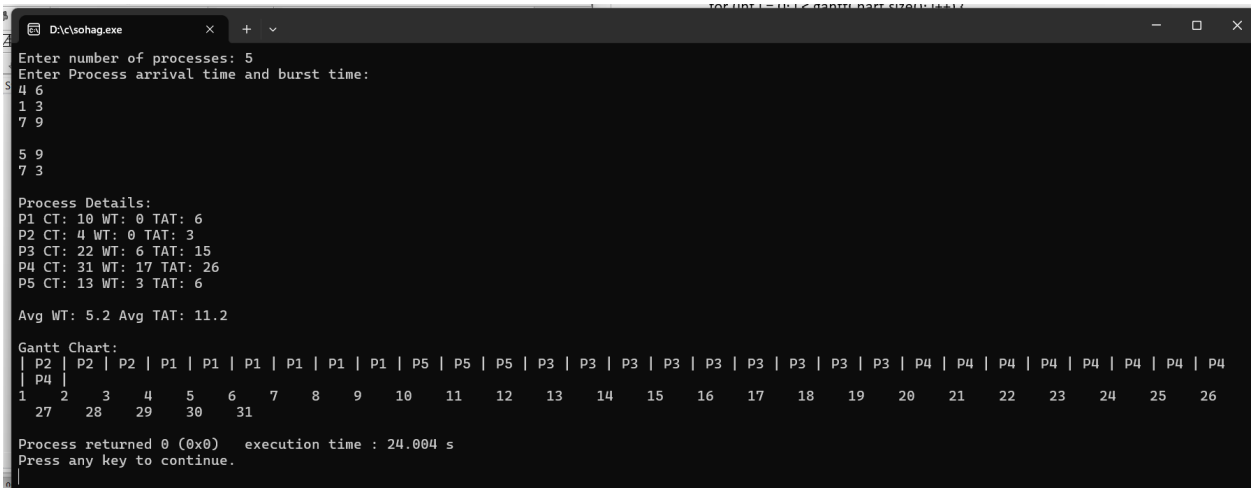


Q4. SRFT Algorithm With Gantt Chart :

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

struct Process {
```

```
    int id, arrivalTime, burstTime, remainingTime, waitingTime, turnaroundTime,
completionTime;
};

int main() {
    int n, currentTime = 0, completed = 0;
    cout << "Enter number of processes: ";
    cin >> n;
    vector<Process> p(n);

    cout << "Enter Process arrival time and burst time:" << endl;
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        cin >> p[i].arrivalTime >> p[i].burstTime;
        p[i].remainingTime = p[i].burstTime;
    }

    vector<pair<int, int>> ganttChart;
    while (completed < n) {
        int idx = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].arrivalTime <= currentTime && p[i].remainingTime > 0 && (idx == -1
|| p[i].remainingTime < p[idx].remainingTime)) {
                idx = i;
            }
        }

        if (idx != -1) {
            p[idx].remainingTime--;
            ganttChart.push_back({currentTime, p[idx].id});
            currentTime++;

            if (p[idx].remainingTime == 0) {
                p[idx].completionTime = currentTime;
                p[idx].turnaroundTime = currentTime - p[idx].arrivalTime;
                p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;
                completed++;
            }
        } else {
            currentTime++;
        }
```

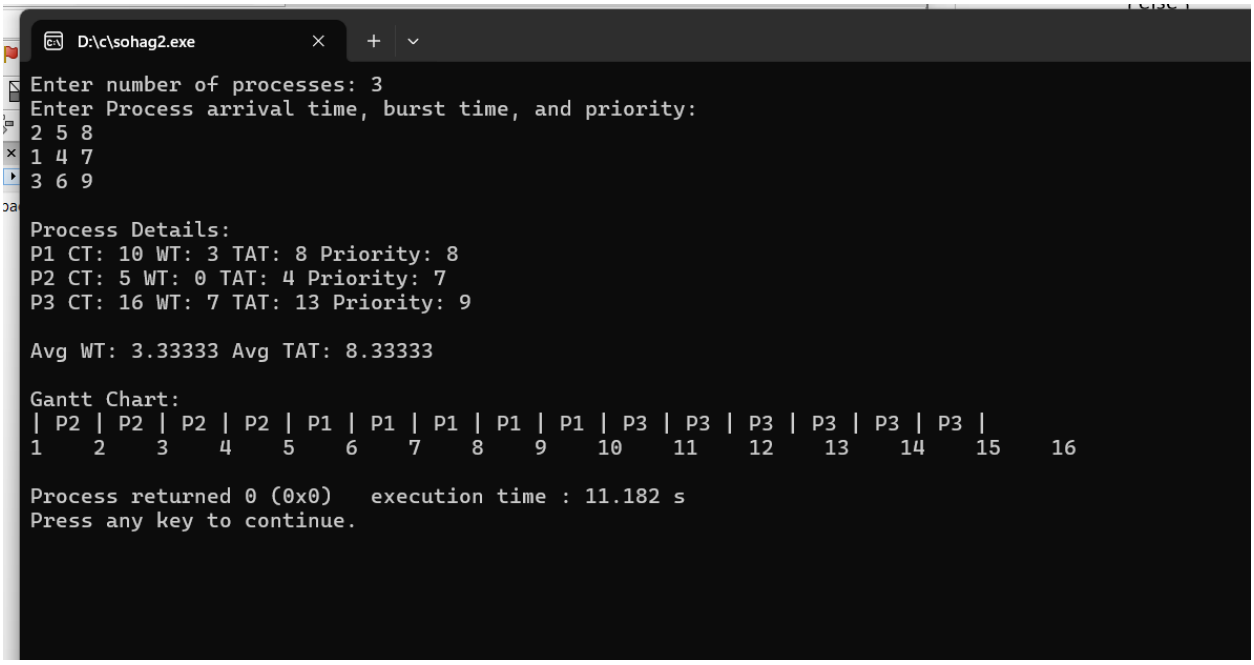
```
    }

    double totalWT = 0, totalTAT = 0;
    cout << "\nProcess Details:" << endl;
    for (auto &proc : p) {
        totalWT += proc.waitingTime;
        totalTAT += proc.turnaroundTime;
        cout << "P" << proc.id << " CT: " << proc.completionTime << " WT: " <<
proc.waitingTime << " TAT: " << proc.turnaroundTime << endl;
    }

    cout << "\nAvg WT: " << totalWT / n << " Avg TAT: " << totalTAT / n << endl;

    cout << "\nGantt Chart:\n";
    for (int i = 0; i < ganttChart.size(); i++) {
        cout << "| P" << ganttChart[i].second << " ";
    }
    cout << "|" << endl;

    for (int i = 0; i < ganttChart.size(); i++) {
        cout << ganttChart[i].first << "    ";
    }
    cout << currentTime << endl;

    return 0;
}
```



Q5. Preemptive Algorithm

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

struct Process {
   int id, arrivalTime, burstTime, remainingTime, waitingTime, turnaroundTime,
completionTime,
priority;
};

int main() {
   int n, currentTime = 0, completed = 0;
   cout << "Enter number of processes: ";
   cin >> n;
   vector<Process> p(n);

   cout << "Enter Process arrival time, burst time, and priority:" << endl;
   for (int i = 0; i < n; i++) {
      p[i].id = i + 1;
      cin >> p[i].arrivalTime >> p[i].burstTime >> p[i].priority;
      p[i].remainingTime = p[i].burstTime;
   }

   vector<pair<int, int>> ganttChart;
   while (completed < n) {
      int idx = -1;
      for (int i = 0; i < n; i++) {
         if (p[i].arrivalTime <= currentTime && p[i].remainingTime > 0 && (idx == -1
|| p[i].priority <
p[idx].priority)) {
            idx = i;
         }
      }

      if (idx != -1) {
         p[idx].remainingTime--;
         ganttChart.push_back({currentTime, p[idx].id});
         currentTime++;

         if (p[idx].remainingTime == 0) {
```

```
            p[idx].completionTime = currentTime;
            p[idx].turnaroundTime = currentTime - p[idx].arrivalTime;
            p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;
            completed++;
          }
        } else {
          currentTime++;
        }
    }

    double totalWT = 0, totalTAT = 0;
    cout << "\nProcess Details:" << endl;
    for (auto &proc : p) {
        totalWT += proc.waitingTime;
        totalTAT += proc.turnaroundTime;
        cout << "P" << proc.id << " CT: " << proc.completionTime << " WT: " <<
proc.waitingTime << " TAT: "
<< proc.turnaroundTime << " Priority: " << proc.priority << endl;
    }

    cout << "\nAvg WT: " << totalWT / n << " Avg TAT: " << totalTAT / n << endl;

    cout << "\nGantt Chart:\n";
    for (int i = 0; i < ganttChart.size(); i++) {
        cout << "| P" << ganttChart[i].second << " ";
    }
    cout << "|" << endl;

    for (int i = 0; i < ganttChart.size(); i++) {
        cout << ganttChart[i].first << "    ";
    }
    cout << currentTime << endl;

    return 0;
}
```



Q6. Round Robin Algorithm

```
#include <iostream>
using namespace std;


void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum, int
gantt[], int &ganttSize, int
timeMarks[])
{
   int rem_bt[n];
   for (int i = 0; i < n; i++)
      rem_bt[i] = bt[i];

   int t = 0;
   ganttSize = 0;

   while (1)
   {
      bool done = true;
      for (int i = 0; i < n; i++)
      {
         if (rem_bt[i] > 0)
         {
            done = false;
```

```
            gantt[ganttSize] = processes[i];
            timeMarks[ganttSize++] = t;

            if (rem_bt[i] > quantum)
            {
               t += quantum;
               rem_bt[i] -= quantum;
            }
            else
            {
               t += rem_bt[i];
               wt[i] = t - bt[i];
               rem_bt[i] = 0;
            }
         }
      }
      if (done)
      {
         timeMarks[ganttSize] = t;
         break;
      }
   }
}

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

void findavgTime(int processes[], int n, int bt[], int quantum)
{
   int wt[n], tat[n], total_wt = 0, total_tat = 0;
   int gantt[100], ganttSize = 0, timeMarks[100];

   findWaitingTime(processes, n, bt, wt, quantum, gantt, ganttSize, timeMarks);
   findTurnAroundTime(processes, n, bt, wt, tat);

   cout << "PN\t BT \tWT \tTAT\n";
   for (int i = 0; i < n; i++)
   {
```

```
        total_wt += wt[i];
        total_tat += tat[i];
        cout << " " << processes[i] << "\t " << bt[i] << "\t " << wt[i] << "\t " << tat[i] <<
endl;
    }

    cout << "Average waiting time = " << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = " << (float)total_tat / (float)n;


    cout << "\nGantt Chart:\n|";
    for (int i = 0; i < ganttSize; i++)
        cout << " P" << gantt[i] << " |";
    cout << "\n";

    for (int i = 0; i <= ganttSize; i++)
cout << timeMarks[i] << "\t";
cout << endl;
}
int main()
{
int processes[] = {4, 5, 6};
int n = sizeof processes / sizeof processes[0];
int burst_time[] = {9, 6, 7};
int quantum = 2;
findavgTime(processes, n, burst_time, quantum);
return 0;
}
```