**Assignment Number 02**

**Name:** Mihir Unmesh Patil

**Roll NO:** TYCOC213

**Batch:** C/C-3

**CODE:**

#include <iostream>

#include <cstdlib>

#include <queue>

#include <cstdio>

#include <algorithm>

using namespace std;

typedef struct process {

    int id, at, bt, remaining\_bt, completion\_time, pr;

    float wt, tat;

} process;

process p[10], p1[10], temp;

queue<int> q1;

int accept(int ch);

void turnwait(int n);

void display(int n);

void gantt\_fcfs(int n);

void sjf\_non\_preemptive(int n);

void priority\_non\_preemptive(int n);

void priority\_preemptive(int n, int\* gantt\_chart);

void ganttrr(int n);

void fcfs(int n);

void gantt\_sjf(int n);

void sjf\_preemptive(int n);

void gantt\_priority\_preemptive(int n, int\* gantt\_chart);

int main() {

int n, ch;

cout << "Choose Scheduling Algorithm: \n";

cout << "1. FCFS\n";

cout << "2. SJF (Non-Preemptive)\n";

cout << "3. SJF (Preemptive - SRTF)\n";

cout << "4. Preemptive Priority Scheduling\n";

cout << "5. Non-Preemptive Priority Scheduling\n";

cout << "6. Round Robin\n";

cin >> ch;

n = accept(ch);

switch (ch) {

case 1:

fcfs(n);

break;

case 2:

sjf\_non\_preemptive(n);

break;

case 3:

sjf\_preemptive(n);

break;

case 4: {

int gantt\_chart[100] = {0};

priority\_preemptive(n, gantt\_chart);

turnwait(n);

display(n);

gantt\_priority\_preemptive(n, gantt\_chart);

break;

}

case 5:

priority\_non\_preemptive(n);

break;

case 6:

ganttrr(n);

break;

default:

cout << "Invalid choice!" << endl;

exit(1);

}

if (ch != 4) {

turnwait(n);

display(n);

}

return 0;

}

int accept(int ch) {

    int i, n;

    cout << "Enter the Total Number of Processes: ";

    cin >> n;

    if (n == 0) {

        cout << "Invalid number of processes!" << endl;

        exit(1);

    }

    cout << endl;

    for (i = 1; i <= n; i++) {

        cout << "Enter the Arrival Time of Process P" << i << ": ";

        cin >> p[i].at;

        p[i].id = i;

    }

    cout << endl;

    for (i = 1; i <= n; i++) {

        cout << "Enter the Burst Time of Process P" << i << ": ";

        cin >> p[i].bt;

        p[i].remaining\_bt = p[i].bt;

    }

    if (ch == 3 || ch == 4) {

        for (i = 1; i <= n; i++) {

            cout << "Enter the Priority of Process P" << i << ": ";

            cin >> p[i].pr;

        }

    }

    for (i = 1; i <= n; i++) {

        p1[i] = p[i];

    }

    return n;

}

void turnwait(int n) {

    int i;

    for (i = 1; i <= n; i++) {

        p[i].tat = p[i].completion\_time - p[i].at;

        p[i].wt = p[i].tat - p[i].bt;

        p[0].tat = p[0].tat + p[i].tat;

        p[0].wt = p[0].wt + p[i].wt;

    }

    p[0].tat = p[0].tat / n;

    p[0].wt = p[0].wt / n;

}

void display(int n) {

    int i;

    cout << "\n=====================================================\n";

    cout << "\n\nHere AT = Arrival Time\nBT = Burst Time\nCT= Completion Time\nTAT = Turn Around Time\nWT = Waiting Time\n";

    cout << "\n===================TABLE==============================\n";

    cout << "Process\tAT\tBT\tCT\tTAT\t\tWT\n";

    for (i = 1; i <= n; i++) {

        printf("P%d\t%d\t%d\t%d\t%f\t%f\n", p[i].id, p[i].at, p[i].bt, p[i].completion\_time, p[i].tat, p[i].wt);

    }

    cout << "\n=====================================================\n";

    printf("\nAverage Turn Around Time: %f", p[0].tat);

    printf("\nAverage Waiting Time: %f\n", p[0].wt);

}

void fcfs(int n) {

    int i, current\_time = 0;

    // Sort processes based on arrival time

    sort(p + 1, p + n + 1, [](process a, process b) {

        return a.at < b.at;

    });

    for (i = 1; i <= n; i++) {

        // If the current time is less than the arrival time of the process,

        // update the current time to the arrival time

        if (current\_time < p[i].at) {

            current\_time = p[i].at;

        }

        // The completion time is the current time plus the burst time

        p[i].completion\_time = current\_time + p[i].bt;

        // Update the current time to the completion time

        current\_time = p[i].completion\_time;

    }

    gantt\_fcfs(n);

}

void gantt\_fcfs(int n) {

    cout << "\nGantt Chart for FCFS Scheduling\n";

    for (int i = 1; i <= n; i++) {

        cout << "P" << p[i].id << " ";

    }

    cout << endl;

}

void sjf\_preemptive(int n) {

int completed = 0, current\_time = 0;

int current\_process = -1;

bool is\_completed[10] = {false};

while (completed < n) {

int shortest\_time = 9999, next\_process = -1;

for (int i = 1; i <= n; i++) {

if (!is\_completed[i] && p[i].at <= current\_time

&& p[i].remaining\_bt < shortest\_time &&

p[i].remaining\_bt > 0) {

shortest\_time = p[i].remaining\_bt;

next\_process = i;

}

}

if (next\_process != -1) {

if (current\_process != next\_process) {

current\_process = next\_process;

}

p[current\_process].remaining\_bt--;

current\_time++;

if (p[current\_process].remaining\_bt == 0) {

p[current\_process].completion\_time =

current\_time;

is\_completed[current\_process] = true;

completed++;

current\_process = -1;

}

} else {

current\_time++;

}

}

}

void sjf\_non\_preemptive(int n) {

    int completed = 0, current\_time = 0, smallest;

    bool is\_completed[10] = {false};

    while (completed < n) {

        smallest = -1;

        int min\_burst\_time = 9999;

        for (int i = 1; i <= n; i++) {

            if (!is\_completed[i] && p[i].at <= current\_time && p[i].bt < min\_burst\_time) {

                min\_burst\_time = p[i].bt;

                smallest = i;

            }

        }

        if (smallest != -1) {

            current\_time += p[smallest].bt;

            p[smallest].completion\_time = current\_time;

            is\_completed[smallest] = true;

            completed++;

        } else {

            current\_time++;

        }

    }

    gantt\_sjf(n);

}

void gantt\_sjf(int n) {

    cout << "\nGantt Chart for SJF Scheduling\n";

    for (int i = 1; i <= n; i++) {

        cout << "P" << p[i].id << " ";

    }

    cout << endl;

}

void priority\_non\_preemptive(int n) {

    int completed = 0, current\_time = 0, smallest;

    bool is\_completed[10] = {false};

    while (completed < n) {

        smallest = -1;

        int min\_priority = 9999;

        for (int i = 1; i <= n; i++) {

            if (!is\_completed[i] && p[i].at <= current\_time && p[i].pr < min\_priority) {

                min\_priority = p[i].pr;

                smallest = i;

            }

        }

        if (smallest != -1) {

            current\_time += p[smallest].bt;

            p[smallest].completion\_time = current\_time;

            is\_completed[smallest] = true;

            completed++;

        } else {

            current\_time++;

        }

    }

}

void priority\_preemptive(int n, int\* gantt\_chart) {

    int completed = 0, current\_time = 0;

    bool is\_completed[10] = {false};

    int current\_process = -1;

    while (completed < n) {

        int highest\_priority = 9999, next\_process = -1;

        for (int i = 1; i <= n; i++) {

            if (!is\_completed[i] && p[i].at <= current\_time && p[i].pr < highest\_priority && p[i].remaining\_bt > 0) {

                highest\_priority = p[i].pr;

                next\_process = i;

            }

        }

        if (next\_process != -1) {

            if (current\_process != next\_process) {

                current\_process = next\_process;

            }

            p[current\_process].remaining\_bt--;

            gantt\_chart[current\_time] = current\_process;

            current\_time++;

            if (p[current\_process].remaining\_bt == 0) {

                p[current\_process].completion\_time = current\_time;

                is\_completed[current\_process] = true;

                completed++;

                current\_process = -1;

            }

        } else {

            gantt\_chart[current\_time] = 0;

            current\_time++;

        }

    }

}

void gantt\_priority\_preemptive(int n, int\* gantt\_chart) {

    cout << "\nGantt Chart for Preemptive Priority Scheduling\n";

    for (int i = 0; i < 20; i++) {

        if (gantt\_chart[i] != 0) {

            cout << "P" << gantt\_chart[i] << " ";

        } else {

            cout << "Idle ";

        }

    }

    cout << endl;

}

void ganttrr(int n) {

    int i, ts, m, nextval, nextarr;

    nextval = p1[1].at;

    i = 1;

    cout << "\nEnter the Time Slice or Quantum: ";

    cin >> ts;

    for (i = 1; i <= n && p1[i].at <= nextval; i++) {

        q1.push(p1[i].id);

    }

    while (!q1.empty()) {

        m = q1.front();

        q1.pop();

        if (p1[m].bt >= ts) {

            nextval = nextval + ts;

        } else {

            nextval = nextval + p1[m].bt;

        }

        if (p1[m].bt >= ts) {

            p1[m].bt = p1[m].bt - ts;

        } else {

            p1[m].bt = 0;

        }

        while (i <= n && p1[i].at <= nextval) {

            q1.push(p1[i].id);

            i++;

        }

        if (p1[m].bt > 0) {

            q1.push(m);

        }

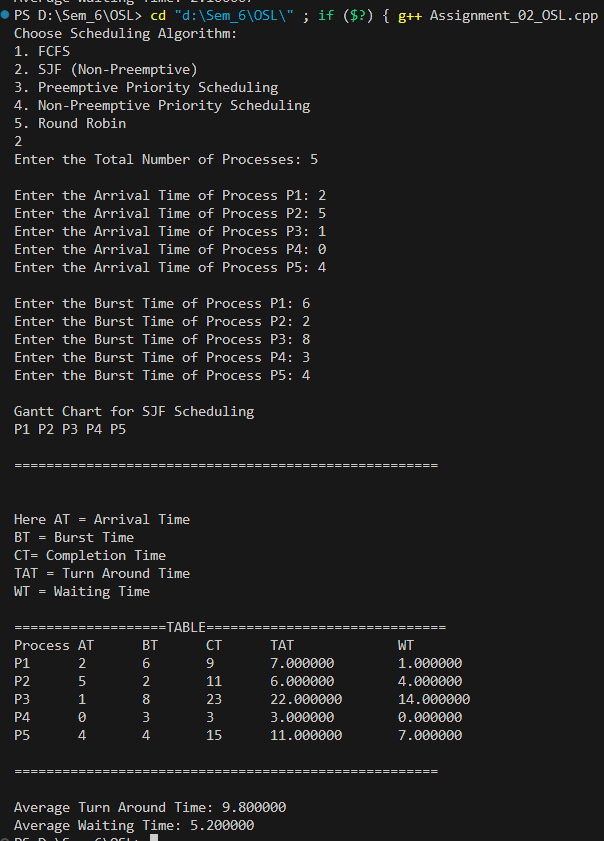
        if (p1[m].bt <= 0) {

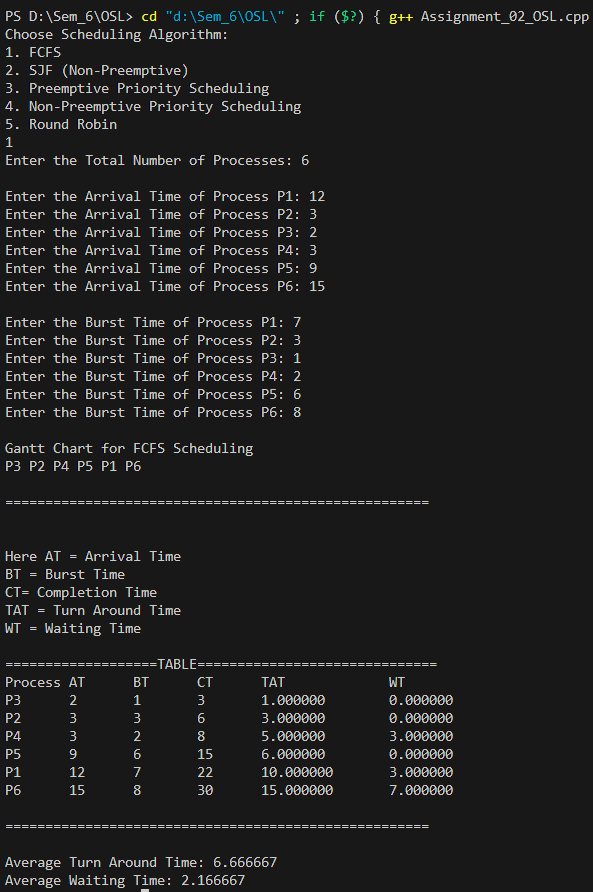
            p[m].completion\_time = nextval;

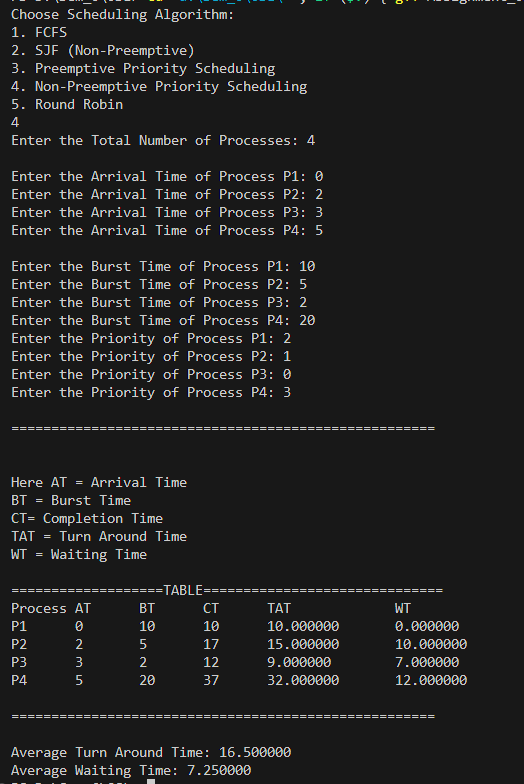
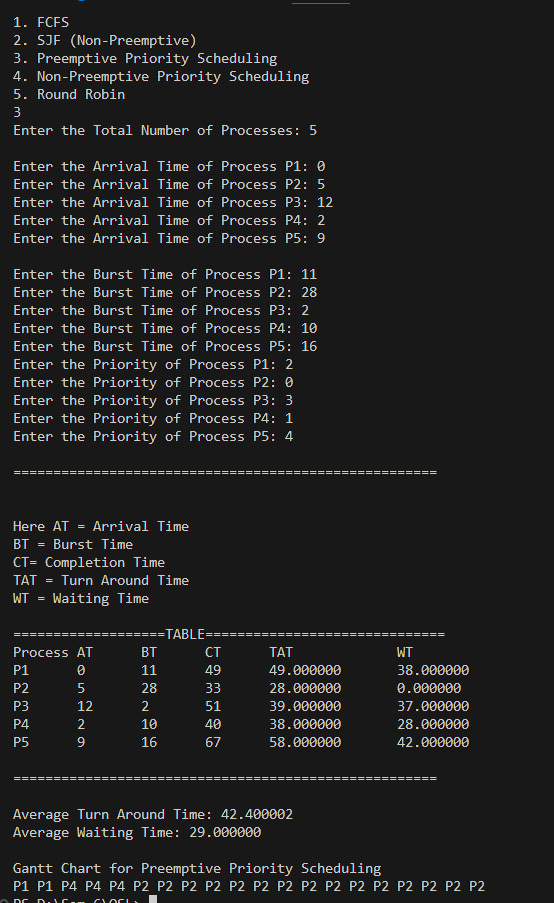
        }

    }

}

OUTPUT:



.

