

***The University of Azad Jammu and Kashmir Muzaffarabad***

|  |  |
| --- | --- |
| ***Name:*** | ***Asifullah*** |
| ***Roll No:*** | ***2023-SE-38*** |
| ***Course:*** | ***Operating Systems*** |
| ***Submitted To:*** | ***Mam Sidra*** |
| ***Lab Task no:*** | ***09*** |

Short Job first(sjf)

Non preemptive scheduling

Shortest Job First (SJF) is a CPU scheduling algorithm in which the process with the shortest burst time is executed first. In **Non-Preemptive SJF**, once a process starts execution, it runs until completion without being interrupted

Characteristics of Non-Preemptive SJF:

* Selection Criteria:
* The process with the shortest burst time is selected first.
* **No Interruption:** Once a process starts execution, it cannot be preempted (interrupted) until it completes.
* Efficient for Batch Processing:
* Since processes are completed one by one based on burst time, this method is ideal for batch systems.
* Sorting Required:
* Processes need to be sorted by burst time before execution

#include<iostream>

#include <algorithm>

using namespace std;

struct Process {

int id, arrival\_time, burst\_time, completion\_time, turnaround\_time, waiting\_time;

};

bool compare(Process p1, Process p2) {

if (p1.arrival\_time == p2.arrival\_time)

return p1.burst\_time < p2.burst\_time;

return p1.arrival\_time < p2.arrival\_time;

}

void calculateTimes(Process processes[], int n) {

sort(processes, processes + n, compare);

int current\_time = 0;

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

if (current\_time < processes[i].arrival\_time) {

current\_time = processes[i].arrival\_time;

}

processes[i].completion\_time = current\_time + processes[i].burst\_time;

processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

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

}

}

void printTable(Process processes[], int n) {

cout << "\nProcess Table:\n";

cout << "PID\tAT\tBT\tCT\tTAT\tWT\n";

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

cout << processes[i].id << "\t" << processes[i].arrival\_time << "\t"

<< processes[i].burst\_time << "\t" << processes[i].completion\_time << "\t"

<< processes[i].turnaround\_time << "\t" << processes[i].waiting\_time << "\n";

}

}

void printGanttChart(Process processes[], int n) {

cout << "\nGantt Chart:\n";

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

cout << "| P" << processes[i].id << " ";

}

cout << "|\n0";

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

cout << " " << processes[i].completion\_time;

}

cout << endl;

}

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

Process processes[n];

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

cout << "Enter arrival time and burst time for process " << i + 1 << ": ";

processes[i].id = i + 1;

cin >> processes[i].arrival\_time >> processes[i].burst\_time;

}

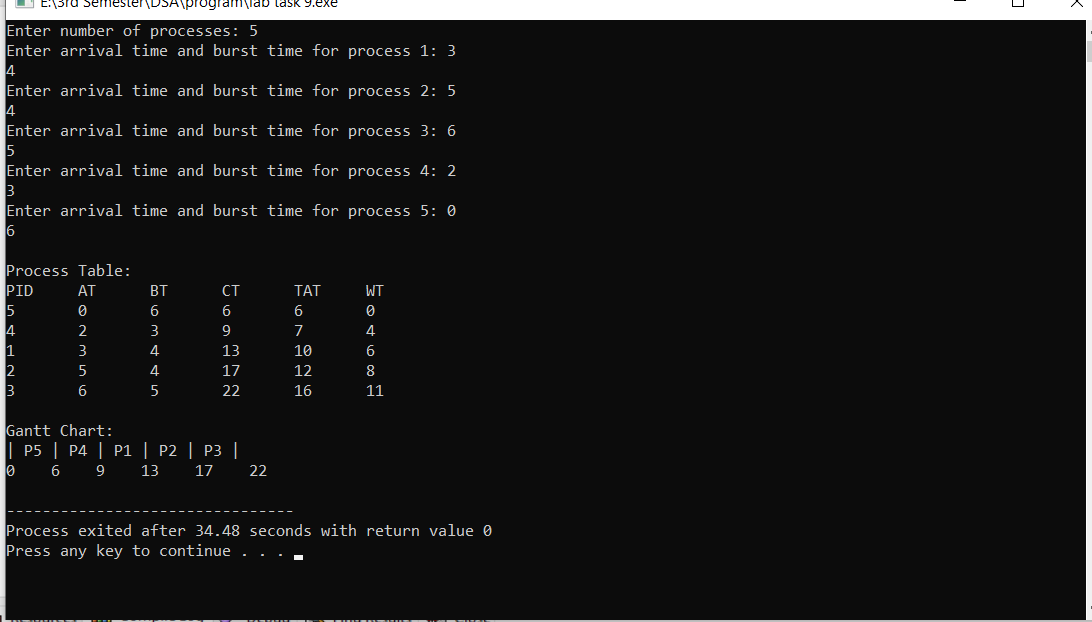
calculateTimes(processes, n);

printTable(processes, n);

printGanttChart(processes, n);

return 0;

}



**2) Shortest Job First (SJF) – Preemptive Scheduling:-**

Definition:

**Preemptive SJF,** also known as Shortest Remaining Time First (SRTF)**,** is a CPUscheduling algorithm where the process with the shortest remaining burst time is always executed first. If a new process arrives with a shorter burst time than the currently running process, the CPU switches to the new process**.**

**Characteristics of Preemptive SJF (SRTF):**

* **Preemptive Nature:** The CPU can switch to a new process if it has a shorter remaining time than the current one.
* **More Responsive:** Since new short processes can take over, it is better for time-sharing and real-time systems.
* **Dynamic Scheduling:** Decisions are made at every time unit based on the remaining burst times of processes.
* **Can Cause Higher Context Switching:** Due to frequent preemptions, there is an overhead in switching between processed.

#include <iostream>

#include <climits>

using namespace std;

struct Process {

int id, arrival\_time, burst\_time, remaining\_time, completion\_time, turnaround\_time, waiting\_time;

};

void calculateTimes(Process processes[], int n) {

int complete = 0, time = 0, min\_index = -1;

int min\_remaining\_time = INT\_MAX;

bool check = false;

while (complete != n) {

min\_index = -1;

min\_remaining\_time = INT\_MAX;

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

if (processes[i].arrival\_time <= time && processes[i].remaining\_time > 0 && processes[i].remaining\_time < min\_remaining\_time) {

min\_index = i;

min\_remaining\_time = processes[i].remaining\_time;

check = true;

}

}

if (!check) {

time++;

continue;

}

processes[min\_index].remaining\_time--;

if (processes[min\_index].remaining\_time == 0) {

complete++;

processes[min\_index].completion\_time = time + 1;

processes[min\_index].turnaround\_time = processes[min\_index].completion\_time - processes[min\_index].arrival\_time;

processes[min\_index].waiting\_time = processes[min\_index].turnaround\_time - processes[min\_index].burst\_time;

}

time++;

}

}

void printTable(Process processes[], int n) {

cout << "\nProcess Table:\n";

cout << "PID\tAT\tBT\tCT\tTAT\tWT\n";

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

cout << processes[i].id << "\t" << processes[i].arrival\_time << "\t"

<< processes[i].burst\_time << "\t" << processes[i].completion\_time << "\t"

<< processes[i].turnaround\_time << "\t" << processes[i].waiting\_time << "\n";

}

}

int main() {

int n;

cout << "Enter number of processes: ";

cin >> n;

Process processes[n];

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

cout << "Enter arrival time and burst time for process " << i + 1 << ": ";

processes[i].id = i + 1;

cin >> processes[i].arrival\_time >> processes[i].burst\_time;

processes[i].remaining\_time = processes[i].burst\_time;

}

calculateTimes(processes, n);

printTable(processes, n);

return 0;

}