**CS-EPC 411 – LABORATORY EXERCISE No.1 to No.3**

Joshua Razzi Garingo

Jan Anthony Geraldez

1. **Problem Description**

Create a program that will perform the ff:

* First-Come First-Serve (FCFS) scheduling.
* Short Job First (SJF) Non-preempted
* Short Job First (SJF) Preempted

1. **Source Code of the Program**

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <tuple>

using namespace std;

int fcfs(int processes){

int at[processes], bt[processes], wt[processes];

string processName[processes];

cout << "Input ARRIVAL TIME & BURST TIME in each processes: ";

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

for (int i=0; i<processes; i++)

{

processName[i] = "P" + to\_string(i);

cout << "Process" << "[" << i << "]" ; //process header

cout << "\nArrival Time" << "[" << i << "]" << ": "; cin >> at[i]; // Arrival Time

cout << "Burst Time" << "[" << i << "]" << ": "; cin >> bt[i]; // Burst Time

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

}

// Initialize storage for sorted processes

vector<tuple<int, int, string>> sortedATBT;

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

sortedATBT.push\_back(make\_tuple(at[i], bt[i], processName[i]));

}

// Sort Processes by Arrival Time

sort(sortedATBT.begin(), sortedATBT.end());

// Update AT[] and BT[] with sorted values

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

at[i] = get<0>(sortedATBT[i]);

bt[i] = get<1>(sortedATBT[i]);

processName[i] = get<2>(sortedATBT[i]);

}

// Computing for Waiting Time

float wtTotal = 0;

wt[0] = 0;

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

wt[i] = wt[i - 1] + bt[i - 1]; //Waiting Time = Constant Zero (0) + Burst Time

wtTotal += wt[i];

}

// Output Section

cout << "Process\tArrival Time\tBurst Time\tWaiting Time";

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

cout << "\n" << processName[i] << "\t" << at[i] << "\t\t" << bt[i] << "\t\t" << wt[i];

}

// Display Average Waiting Time

cout << "\n\nAverage Waiting Time: (";

for (int i = 0; i < processes - 1; i++){

cout << wt[i] << " + ";

}

cout << wt[processes - 1] << ") / "<< processes <<" = " << wtTotal / processes;

cout << "\nGantt Chart for Schedule: " << endl;

// Print top border

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

cout << "--------";

}

cout << "-" << endl;

// Print process names

cout << "|";

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

cout << " " << processName[i] << "\t|";

}

cout << endl;

// Print bottom border

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

cout << "|-------";

}

cout << "|" << endl;

// Print start times

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

cout << wt[i] << "\t";

}

cout << bt[processes - 1] + wt[processes - 1] << endl; // Print the last completion time

return 0;

}

int sjf\_non\_preemptive(int processes){

int at[processes], bt[processes], wt[processes], ct[processes];

string processName[processes];

cout << "Input ARRIVAL TIME & BURST TIME for each process:";

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

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

processName[i] = "P" + to\_string(i + 1);

cout << "Process [" << i + 1 << "]" << endl; // process header

cout << "Arrival Time [" << i + 1 << "] : "; cin >> at[i]; // Arrival Time

cout << "Burst Time [" << i + 1 << "] : "; cin >> bt[i]; // Burst Time

cout << "===================================" << endl;

}

// Initialize storage for sorted processes

vector<tuple<int, int, string>> sortedATBT;

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

sortedATBT.push\_back(make\_tuple(at[i], bt[i], processName[i]));

}

// Sort Processes by Arrival Time, then by Burst Time if Arrival Time is same

sort(sortedATBT.begin(), sortedATBT.end(), [](tuple<int, int, string> a, tuple<int, int, string> b) {

if (get<0>(a) == get<0>(b))

return get<1>(a) < get<1>(b); // Sort by Burst Time if Arrival Time is same

return get<0>(a) < get<0>(b); // Sort by Arrival Time

});

// Update AT[] and BT[] with sorted values

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

at[i] = get<0>(sortedATBT[i]);

bt[i] = get<1>(sortedATBT[i]);

processName[i] = get<2>(sortedATBT[i]);

}

int currentTime = 0;

float wtTotal = 0;

vector<int> completed(processes, 0);

vector<int> ganttChart;

for (int completedProcesses = 0; completedProcesses < processes;) {

int minIndex = -1;

int minBurst = INT\_MAX;

// Find the process with the shortest burst time that has arrived and is not completed

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

if (at[i] <= currentTime && !completed[i] && bt[i] < minBurst) {

minBurst = bt[i];

minIndex = i;

}

}

if (minIndex != -1) { // If a valid process is found

wt[minIndex] = currentTime - at[minIndex];

currentTime += bt[minIndex];

ct[minIndex] = currentTime;

completed[minIndex] = 1;

ganttChart.push\_back(currentTime);

wtTotal += wt[minIndex];

completedProcesses++;

} else {

currentTime++;

}

}

// Output Section

cout << "Process\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time";

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

cout << "\n" << processName[i] << "\t" << at[i] << "\t\t" << bt[i] << "\t\t" << ct[i] << "\t\t" << wt[i];

}

// Display Average Waiting Time

cout << "\n\nAverage Waiting Time: (";

for (int i = 0; i < processes - 1; i++) {

cout << wt[i] << " + ";

}

cout << wt[processes - 1] << ") / " << processes << " = " << wtTotal / processes;

// Print Gantt Chart

cout << "\nGantt Chart for Schedule: " << endl;

// Print top border

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

cout << "--------";

}

cout << "-" << endl;

// Print process names

cout << "|";

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

cout << " " << processName[i] << "\t|";

}

cout << endl;

// Print bottom border

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

cout << "|-------";

}

cout << "|" << endl;

// Print Gantt Chart start times

int startTime = 0;

cout << startTime << "\t";

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

startTime += bt[i];

cout << ganttChart[i] << "\t";

}

cout << endl;

return 0;

}

int sjf\_preemptive(int processes){

int at[processes], bt[processes], remaining\_bt[processes], wt[processes], ct[processes], tat[processes];

string processName[processes];

cout << "Input ARRIVAL TIME & BURST TIME for each process:";

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

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

processName[i] = "P" + to\_string(i + 1);

cout << "Process [" << i + 1 << "]" << endl; // process header

cout << "Arrival Time [" << i + 1 << "] : "; cin >> at[i]; // Arrival Time

cout << "Burst Time [" << i + 1 << "] : "; cin >> bt[i]; // Burst Time

cout << "===================================" << endl;

remaining\_bt[i] = bt[i]; // Remaining burst time initially set to burst time

}

int completed = 0, currentTime = 0, minBurstIndex;

float wtTotal = 0, tatTotal = 0;

bool foundProcess = false;

// While there are unfinished processes

while (completed != processes) {

int minBurst = INT\_MAX;

foundProcess = false;

// Find the process with the shortest remaining burst time that has arrived

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

if (at[i] <= currentTime && remaining\_bt[i] > 0 && remaining\_bt[i] < minBurst) {

minBurst = remaining\_bt[i];

minBurstIndex = i;

foundProcess = true;

}

}

if (!foundProcess) {

currentTime++; // No process is ready, increment time

continue;

}

// Execute the found process

remaining\_bt[minBurstIndex]--;

currentTime++;

// If the process is completed

if (remaining\_bt[minBurstIndex] == 0) {

completed++;

ct[minBurstIndex] = currentTime; // Completion time

tat[minBurstIndex] = ct[minBurstIndex] - at[minBurstIndex]; // Turnaround Time

wt[minBurstIndex] = tat[minBurstIndex] - bt[minBurstIndex]; // Waiting Time

wtTotal += wt[minBurstIndex];

tatTotal += tat[minBurstIndex];

}

}

// Output Section

cout << "Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time";

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

cout << "\n" << processName[i] << "\t" << at[i] << "\t\t" << bt[i] << "\t\t" << ct[i] << "\t\t" << tat[i] << "\t\t" << wt[i];

}

// Display Average Waiting Time

cout << "\n\nAverage Waiting Time: (";

for (int i = 0; i < processes - 1; i++) {

cout << wt[i] << " + ";

}

cout << wt[processes - 1] << ") / " << processes << " = " << wtTotal / processes;

// Display Average Turnaround Time

cout << "\nAverage Turnaround Time: (";

for (int i = 0; i < processes - 1; i++) {

cout << tat[i] << " + ";

}

cout << tat[processes - 1] << ") / " << processes << " = " << tatTotal / processes;

return 0;

}

int main()

{

int mode;

// Select Mode Section "FCFS, SJF(non-preemp.), SJF(preemp.)"

cout << "=====================================" <<endl;

cout << "[1] - First Come First Serve"<<"\n[2] - SJF (non-preemptive)" << "\n[3] - SJF (preemptive)"<<endl;

cout << "=====================================" <<endl;

cout << "Select Mode of Choice: "; cin >> mode;

cout << "=====================================" <<endl;

if (mode == 1){ //FIRST COME FIRST SERVE

int processes;

cout << "Input how many processes: "; cin >> processes;

fcfs(processes);

}else if (mode == 2){ //SHORT JOB FIRST (non-preemptive)

int processes;

cout << "Input how many processes: "; cin >> processes;

sjf\_non\_preemptive(processes);

}else if (mode == 3){ //SHORT JOB FIRST (preemptive)

int processes;

cout << "Input how many processes: "; cin >> processes;

sjf\_preemptive(processes);

}else

cout << "Please, Try Again!"; return mode;

}

1. **Description of the Program**

The provided program implements three different CPU scheduling algorithms: First-Come First-Serve (FCFS), Shortest Job First (SJF) Non-preemptive, and Shortest Job First (SJF) Preemptive. Upon execution, the program prompts the user to select one of these scheduling methods. Depending on the choice, it then calls the appropriate function: fcfs(), sjf\_non\_preemptive(), or sjf\_preemptive(). Each function is designed to handle a specific scheduling algorithm.

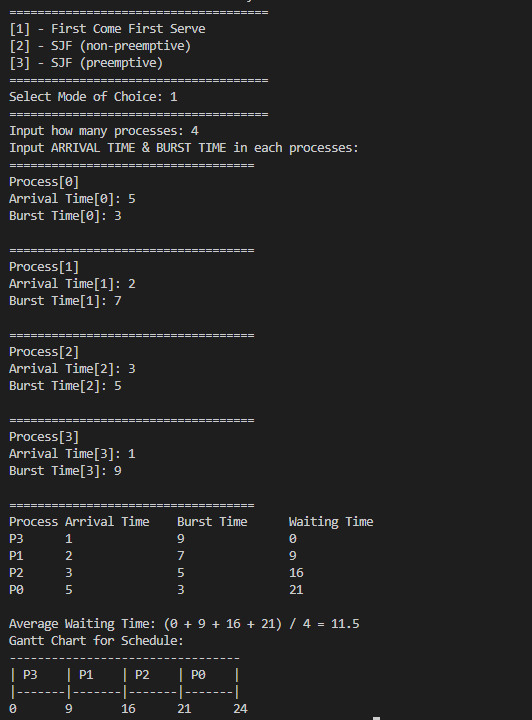
The fcfs() function follows the First-Come First-Serve approach, where processes are executed in the order they arrive. It prompts the user to input the Arrival Time and Burst Time for each process, sorts the processes based on their Arrival Time, and calculates the Waiting Time for each. The waiting time is computed as the sum of the burst times of all previous processes in the queue. The function then outputs the process details along with their waiting times, calculates the average waiting time, and displays a Gantt chart showing the execution sequence of the processes.

The sjf\_non\_preemptive() function implements the Shortest Job First Non-preemptive scheduling algorithm. This function begins by accepting the Arrival Time and Burst Time for each process and then sorts them first by Arrival Time and, in cases where processes have the same Arrival Time, by Burst Time. It selects the process with the shortest Burst Time among those that have arrived and executes it completely before moving to the next process. After executing all processes, it calculates and displays each process’s Waiting Time and Completion Time. The function also calculates the average waiting time and displays a Gantt chart to visualize the execution sequence.

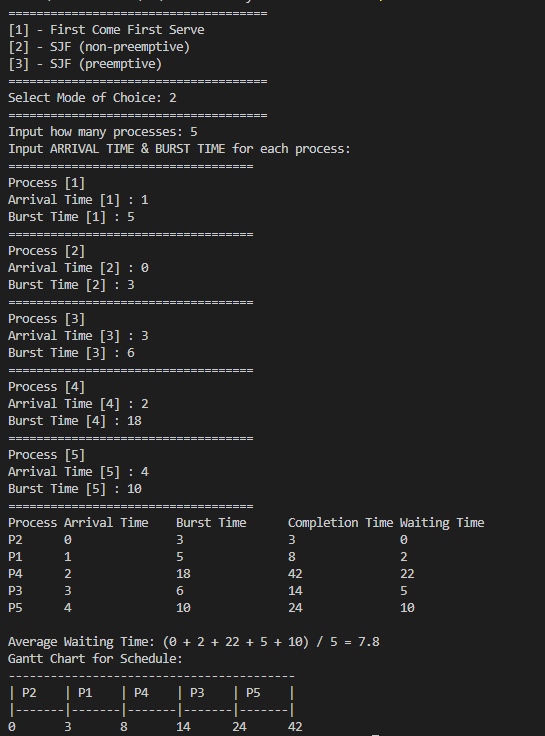
The sjf\_preemptive() function employs the Shortest Job First Preemptive scheduling strategy, also known as Shortest Remaining Time First (SRTF). In this approach, the process with the shortest remaining Burst Time is executed at each unit of time. If a new process with a shorter Burst Time arrives, the currently executing process is preempted, and the new process is executed. This function calculates the Waiting Time, Turnaround Time, and Completion Time for each process. After all processes have completed, it displays the process details along with the average waiting and turnaround times.

The program also generates Gantt charts for each scheduling algorithm to provide a visual representation of the process execution order and start/finish times. The Gantt charts help in understanding the scheduling behavior of each algorithm.

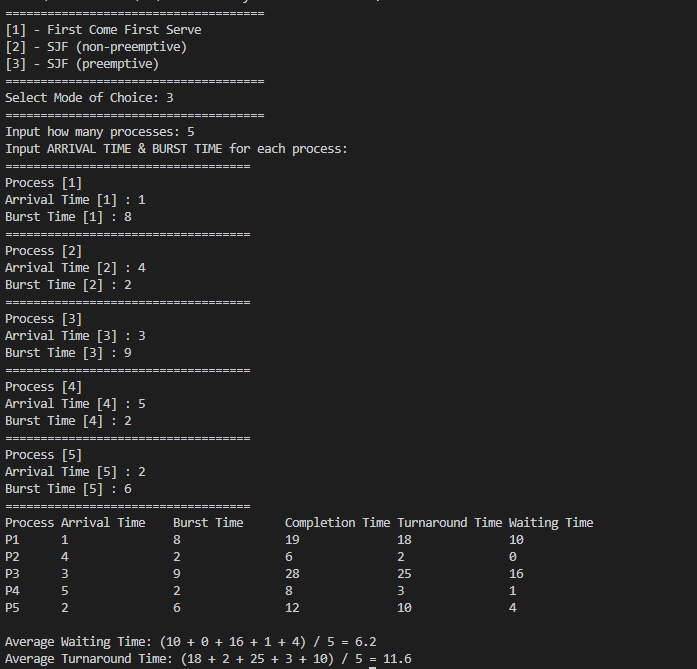
1. **Sample Output of the Program**



***Figure 1****. First Come First Serve sample output*



***Figure 2****. SJF non-preemptive sample output*



***Figure 3****. SFJ preemptive sample output*