#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {

int priority;

int burst\_time;

};

bool comparePriority(const Process& p1, const Process& p2) {

return p1.priority > p2.priority;

}

double calculateWaitingTime(const vector<int>& completion\_time) {

vector<int> waiting\_time(completion\_time.size());

waiting\_time[0] = 0;

for (int i = 1; i < completion\_time.size(); ++i) {

waiting\_time[i] = completion\_time[i - 1];

}

float sum=0;

for(int i=1; i<completion\_time.size();++i){

sum+=waiting\_time[i];

}

return sum / completion\_time.size();

}

void nonPreemptivePriorityScheduling(vector<Process>& processes) {

sort(processes.begin(), processes.end(), comparePriority);

vector<int> completion\_time(processes.size(), 0);

completion\_time[0] = processes[0].burst\_time;

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

completion\_time[i] = completion\_time[i - 1] + processes[i].burst\_time;

}

double average\_waiting\_time = calculateWaitingTime(completion\_time);

cout << "Non-preemptive Average Waiting Time: " << average\_waiting\_time << endl;

}

void preemptivePriorityScheduling(vector<Process>& processes) {

sort(processes.begin(), processes.end(), comparePriority);

int currentTime = 0;

vector<int> remaining\_time(processes.size(), 0);

while (any\_of(remaining\_time.begin(), remaining\_time.end(), [](int time) { return time > 0; })) {

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

if (remaining\_time[i] > 0) {

if (processes[i].burst\_time <= currentTime) {

currentTime += 1;

remaining\_time[i] -= 1;

} else

currentTime += 1;

}

}

}

double average\_waiting\_time = calculateWaitingTime(remaining\_time);

cout << "Preemptive Average Waiting Time: " << average\_waiting\_time << endl;

}

int main() {

int num\_processes;

cout << "Enter the number of processes: ";

cin >> num\_processes;

vector<Process> processes;

for (int i = 0; i < num\_processes; ++i) {

Process p;

cout << "Enter priority for process " << i + 1 << ": ";

cin >> p.priority;

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

cin >> p.burst\_time;

processes.push\_back(p);

}

// Non-preemptive Priority Scheduling

nonPreemptivePriorityScheduling(processes);

// Preemptive Priority Scheduling

preemptivePriorityScheduling(processes);

return 0;

}

Explanation:

1.Header and Namespace Declaration:

#include <iostream>

#include <vector>

#include <algorithm>

The code begins by including essential C++ standard libraries, such as iostream for input/output, vector for dynamic arrays, algorithm for sorting, and numeric for mathematical operations.

2. Process Structure:

struct Process {

int priority;

int burst\_time;

};

The Process structure is introduced to represent a process with attributes priority and burst\_time.

3. Comparator Function:

bool comparePriority(const Process& p1, const Process& p2) {

return p1.priority > p2.priority;

}

A comparator function named comparePriority is defined to facilitate the sorting of processes based on priority in descending order.

4. Calculate Waiting Time Function:

double calculateWaitingTime(const vector<int>& completion\_time) {

vector<int> waiting\_time(completion\_time.size());

waiting\_time[0] = 0;

for (int i = 1; i < completion\_time.size(); ++i) {

waiting\_time[i] = completion\_time[i - 1];

}

float sum=0;

for(int i=1; i<completion\_time.size();++i){

sum+=waiting\_time[i];

}

return sum / completion\_time.size();

}

The calculateWaitingTime function computes waiting times based on a vector of completion times. It uses the accumulate function to sum up waiting times.

5. Non-preemptive Priority Scheduling Function:

void nonPreemptivePriorityScheduling(vector<Process>& processes) {

sort(processes.begin(), processes.end(), comparePriority);

vector<int> completion\_time(processes.size(), 0);

completion\_time[0] = processes[0].burst\_time;

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

completion\_time[i] = completion\_time[i - 1] + processes[i].burst\_time;

}

double average\_waiting\_time = calculateWaitingTime(completion\_time);

cout << "Non-preemptive Average Waiting Time: " << average\_waiting\_time << endl;

}

The nonPreemptivePriorityScheduling function sorts processes by priority and calculates completion times. The average waiting time is then computed using the calculateWaitingTime function and displayed.

6. Preemptive Priority Scheduling Function:

void preemptivePriorityScheduling(vector<Process>& processes) {

sort(processes.begin(), processes.end(), comparePriority);

int currentTime = 0;

vector<int> remaining\_time(processes.size(), 0);

while (any\_of(remaining\_time.begin(), remaining\_time.end(), [](int time) { return time > 0; })) {

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

if (remaining\_time[i] > 0) {

if (processes[i].burst\_time <= currentTime) {

currentTime += 1;

remaining\_time[i] -= 1;

} else

currentTime += 1;

}

}

}

double average\_waiting\_time = calculateWaitingTime(remaining\_time);

cout << "Preemptive Average Waiting Time: " << average\_waiting\_time << endl;

}

The preemptivePriorityScheduling function simulates the execution of processes until completion in a preemptive manner. It updates remaining times during execution and calculates the average waiting time using the calculateWaitingTime function.

7. Main Function:

int main() {

int num\_processes;

cout << "Enter the number of processes: ";

cin >> num\_processes;

vector<Process> processes;

for (int i = 0; i < num\_processes; ++i) {

Process p;

cout << "Enter priority for process " << i + 1 << ": ";

cin >> p.priority;

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

cin >> p.burst\_time;

processes.push\_back(p);

}

// Non-preemptive Priority Scheduling

nonPreemptivePriorityScheduling(processes);

// Preemptive Priority Scheduling

preemptivePriorityScheduling(processes);

return 0;

}

In the main function, a vector of processes with priorities and burst times is created. Both non-preemptive and preemptive priority scheduling functions are called, and the results are printed.