#include <iostream>

#include <limits.h>

#include <vector>

#include <queue>

#include <iomanip>

using namespace std;

struct Process {

    int burst, arrival, id, completion, waiting, turnaround, response, remainingBurst;

    bool active;

};

class Scheduler {

public:

    int n;

    Process processes[30];

    virtual void inputProcesses() = 0; // Pure virtual function

    virtual void execute() = 0; // Pure virtual function

   void displayGanttChart(const vector<int>& ganttChart, const vector<int>& ganttTimes) {

    cout << "\nGantt Chart:\n";

    // Top border

    for (int i : ganttChart) {

        if (i == -1) { // Idle time

            cout << "+-----";

        } else {

            cout << "+-----";

        }

    }

    cout << "+\n";

    // Process rows with labels

    cout << "|";

    for (int i : ganttChart) {

        if (i == -1) {

            cout << " Idle |";

        } else {

            cout << " P" << setw(2) << i << " |";

        }

    }

    cout << "\n";

    // Bottom border

    for (int i : ganttChart) {

        cout << "+-----";

    }

    cout << "+\n";

    // Time labels

    for (size\_t i = 0; i < ganttTimes.size(); i++) {

        cout << setw(3) << ganttTimes[i] << "   ";

    }

    cout << "\n";

}

    void displayMetrics() {

        double totalWaiting = 0, totalTurnaround = 0, totalCompletion = 0;

        cout << "\n\n  | Completion time | Waiting time | Turnaround time | Response time\n";

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

            totalWaiting += processes[j].waiting;

            totalTurnaround += processes[j].turnaround;

            totalCompletion += processes[j].completion;

            cout << "P" << j << " | " << setw(15) << processes[j].completion

                 << " | " << setw(12) << processes[j].waiting

                 << " | " << setw(15) << processes[j].turnaround

                 << " | " << setw(12) << processes[j].response << "\n";

        }

        cout << "\nAverage completion time: " << totalCompletion / n;

        cout << "\nAverage waiting time: " << totalWaiting / n;

        cout << "\nAverage turnaround time: " << totalTurnaround / n;

    }

};

class FCFS : public Scheduler {

public:

    void inputProcesses() override {

        cout << "\nEnter number of processes: ";

        cin >> n;

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

            cout << "\nEnter arrival time of P" << i << ": ";

            cin >> processes[i].arrival;

            cout << "Enter burst time of P" << i << ": ";

            cin >> processes[i].burst;

            processes[i].id = i;

            processes[i].active = false;

        }

        cout << "\n  | Arrival | Burst\n";

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

            cout << "P" << j << " | " << processes[j].arrival << "       | " << processes[j].burst << "\n";

        }

    }

    void execute() override {

        cout << "\nSequence of processes is: ";

        int currentTime = 0;

        vector<int> ganttChart;

        vector<int> ganttTimes;

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

            if (currentTime < processes[i].arrival) {

                while (currentTime < processes[i].arrival) {

                    ganttChart.push\_back(-1); // Idle time

                    ganttTimes.push\_back(currentTime);

                    currentTime++;

                }

            }

            ganttChart.push\_back(processes[i].id);

            ganttTimes.push\_back(currentTime);

            processes[i].response = currentTime - processes[i].arrival;

            currentTime += processes[i].burst;

            processes[i].completion = currentTime;

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

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

        }

        ganttTimes.push\_back(currentTime);

        displayGanttChart(ganttChart, ganttTimes);

        displayMetrics();

    }

};

class SJF : public Scheduler {

public:

    void inputProcesses() override {

        cout << "\nEnter number of processes: ";

        cin >> n;

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

            cout << "\nEnter arrival time of P" << i << ": ";

            cin >> processes[i].arrival;

            cout << "Enter burst time of P" << i << ": ";

            cin >> processes[i].burst;

processes[i].remainingBurst = processes[i].burst;

            processes[i].id = i;

            processes[i].active = false;

        }

    }

void execute() override {

    int currentTime = 0, completed = 0;

    vector<int> ganttChart;

    vector<int> ganttTimes;

    while (completed < n) {

        int idx = -1;

        int minBurst = INT\_MAX;

        // Find the next process to execute

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

            if (processes[i].arrival <= currentTime && processes[i].burst > 0 && processes[i].burst < minBurst) {

                minBurst = processes[i].burst;

                idx = i;

            }

        }

        if (idx != -1) {

            // Only add to Gantt chart when the process starts or completes

            if (ganttChart.empty() || ganttChart.back() != processes[idx].id) {

                ganttChart.push\_back(processes[idx].id);

                ganttTimes.push\_back(currentTime);

            }

            processes[idx].response = currentTime - processes[idx].arrival;

            currentTime++;

            processes[idx].burst--;

            // Check if the process is completed

            if (processes[idx].burst == 0) {

                processes[idx].completion = currentTime;

                processes[idx].turnaround = processes[idx].completion - processes[idx].arrival;

                // Calculate waiting time

                processes[idx].waiting = processes[idx].turnaround - (processes[idx].remainingBurst); // Use original burst time

                completed++;

            }

        } else {

            currentTime++; // Idle time if no process is ready

        }

    }

    ganttTimes.push\_back(currentTime); // Final time point

    displayGanttChart(ganttChart, ganttTimes);

    displayMetrics();

}

};

class RoundRobin : public Scheduler {

public:

    int timeQuantum;

    void inputProcesses() override {

        cout << "\nEnter number of processes: ";

        cin >> n;

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

            cout << "\nEnter arrival time of P" << i << ": ";

            cin >> processes[i].arrival;

            cout << "Enter burst time of P" << i << ": ";

              cin >> processes[i].burst;

            processes[i].burst = processes[i].remainingBurst = processes[i].burst;

            processes[i].id = i;

            processes[i].waiting = 0; // Initialize waiting time

            processes[i].active = false;

        }

        cout << "\nEnter time quantum: ";

        cin >> timeQuantum;

    }

    void execute() override {

        int currentTime = 0;

        int completed = 0;

        vector<int> ganttChart;

        vector<int> ganttTimes;

        queue<int> readyQueue;

        while (completed < n) {

            // Add all processes that have arrived to the ready queue

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

                if (processes[i].arrival <= currentTime && processes[i].remainingBurst > 0 && !processes[i].active) {

                    readyQueue.push(i);

                    processes[i].active = true; // Mark as active

                }

            }

            if (readyQueue.empty()) {

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

                continue;

            }

            int currentProcess = readyQueue.front();

            readyQueue.pop();

            ganttChart.push\_back(processes[currentProcess].id);

            ganttTimes.push\_back(currentTime);

            // Process execution

            int timeSlice = min(timeQuantum, processes[currentProcess].remainingBurst);

            currentTime += timeSlice;

            processes[currentProcess].remainingBurst -= timeSlice;

            // If process is completed

            if (processes[currentProcess].remainingBurst == 0) {

                processes[currentProcess].completion = currentTime;

                processes[currentProcess].turnaround = processes[currentProcess].completion - processes[currentProcess].arrival;

                // Calculate waiting time

                processes[currentProcess].waiting = processes[currentProcess].turnaround - processes[currentProcess].burst;

                completed++;

            } else {

                readyQueue.push(currentProcess); // Re-add the process to the queue for further execution

            }

        }

        ganttTimes.push\_back(currentTime);

        displayGanttChart(ganttChart, ganttTimes);

        displayMetrics();

    }

};

class Priority : public Scheduler {

public:

    void inputProcesses() override {

        cout << "\nEnter number of processes: ";

        cin >> n;

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

            cout << "\nEnter arrival time of P" << i << ": ";

            cin >> processes[i].arrival;

            cout << "Enter burst time of P" << i << ": ";

            cin >> processes[i].burst;

            cout << "Enter priority of P" << i << ": ";

            cin >> processes[i].remainingBurst; // Using remainingBurst for priority

            processes[i].id = i;

            processes[i].active = false;

        }

    }

    void execute() override {

        int currentTime = 0, completed = 0;

        vector<int> ganttChart;

        vector<int> ganttTimes;

        while (completed < n) {

            int idx = -1;

            int highestPriority = INT\_MAX;

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

                if (processes[i].arrival <= currentTime && processes[i].remainingBurst > 0 && processes[i].remainingBurst < highestPriority) {

                    highestPriority = processes[i].remainingBurst;

                    idx = i;

                }

            }

            if (idx != -1) {

                ganttChart.push\_back(processes[idx].id);

                ganttTimes.push\_back(currentTime);

                processes[idx].response = currentTime - processes[idx].arrival;

                currentTime += processes[idx].burst;

                processes[idx].remainingBurst = 0; // Mark as completed

                processes[idx].completion = currentTime;

                processes[idx].turnaround = processes[idx].completion - processes[idx].arrival;

                processes[idx].waiting = processes[idx].turnaround - processes[idx].burst;

                completed++;

            } else {

                currentTime++;

            }

        }

        ganttTimes.push\_back(currentTime);

        displayGanttChart(ganttChart, ganttTimes);

        displayMetrics();

    }

};

int main() {

    int choice;

    cout << "Select scheduling algorithm:\n";

    cout << "1. FCFS\n";

    cout << "2. SJF\n";

    cout << "3. Round Robin\n";

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

    cin >> choice;

    Scheduler\* scheduler = nullptr;

    switch (choice) {

        case 1:

            scheduler = new FCFS();

            break;

        case 2:

            scheduler = new SJF();

            break;

        case 3:

            scheduler = new RoundRobin();

            break;

        case 4:

            scheduler = new Priority();

            break;

        default:

            cout << "Invalid choice.";

            return 1;

    }

    scheduler->inputProcesses();

    scheduler->execute();

    delete scheduler;

    return 0;

}