```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <queue>
#include <iomanip>
#include <climits>
#include <cstdlib>

using namespace std;


#define MAX_PROCESSES 100
#define MAX_GANTT_SEGMENTS 1000


struct Process {
    string pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int finish_time;
    int waiting_time;
    int turnaround_time;
};


struct GanttSegment {
    string pid;
    int start_time;
    int end_time;
};

int read_processes(const string& file_path, Process processes[], int&
process_count, int& quantum) {
    ifstream infile(file_path.c_str());
    if (!infile.is_open()) {
        cout << "لا يمكن فتح الملف: " << file_path << endl;
        exit(1);
    }

    string line;
    process_count = 0;
    quantum = 2;

    while (getline(infile, line)){
        if (line.empty() || line[0] == '#')
    continue;


string pid;
int arrival, burst;
string keyword;
int value;


int pos = line.find_first_of(" \t");
if (pos == string::npos)
    continue;

string first_part = line.substr(0, pos);
string remaining = line.substr(pos + 1);

if (first_part == "Quantum" || first_part == "quantum") {
    // قراءة قيمة الكوانتوم
    try {
        value = stoi(remaining);
        quantum = value;
    }
    catch (...) {
        cout << "قيمة كوانتوم غير صالحة في السطر: " << line << endl;
        exit(1);
    }
}
else {


    pid = first_part;


    int pos2 = remaining.find_first_of(" \t");
    if (pos2 == string::npos) {
        cout << "مدخلات عملية غير صحيحة في السطر: " << line << endl;
        exit(1);
    }

    string arrival_str = remaining.substr(0, pos2);
    string burst_str = remaining.substr(pos2 + 1);

    try {
        arrival = stoi(arrival_str);
        burst = stoi(burst_str);
    }
    catch (...) {
```

```
                cout << "أوقات وصول أو تنفيذ غير صالحة في السطر: " << line << endl;
                exit(1);
            }


            if (process_count < MAX_PROCESSES) {
                processes[process_count].pid = pid;
                processes[process_count].arrival_time = arrival;
                processes[process_count].burst_time = burst;
                processes[process_count].remaining_time = burst;
                processes[process_count].finish_time = 0;
                processes[process_count].waiting_time = 0;
                processes[process_count].turnaround_time = 0;
                process_count++;
            }
            else {
                cout << "تم تجاوز الحد الأقصى لعدد العمليات." << endl;
                break;
            }
        }
    }

    infile.close();
    return quantum;
}


void fcfs_scheduling(Process processes[], int process_count, GanttSegment
gantt_chart[], int& gantt_count, int& total_time) {

    for (int i = 0; i < process_count - 1; i++) {
        for (int j = i + 1; j < process_count; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time ||
                (processes[i].arrival_time == processes[j].arrival_time && processes[i].pid
> processes[j].pid)) {

                Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    gantt_count = 0;
    total_time = 0;
    int current_time = 0;
    for (int i = 0; i < process_count; i++) {

        if (current_time < processes[i].arrival_time) {
            if (gantt_count < MAX_GANTT_SEGMENTS) {
                gantt_chart[gantt_count].pid = "Idle";
                gantt_chart[gantt_count].start_time = current_time;
                gantt_chart[gantt_count].end_time = processes[i].arrival_time;
                gantt_count++;
                current_time = processes[i].arrival_time;
            }
        }


        if (gantt_count < MAX_GANTT_SEGMENTS) {
            gantt_chart[gantt_count].pid = processes[i].pid;
            gantt_chart[gantt_count].start_time = current_time;
            gantt_chart[gantt_count].end_time = current_time +
processes[i].burst_time;
            gantt_count++;
            current_time += processes[i].burst_time;
        }


        processes[i].finish_time = current_time;
        processes[i].turnaround_time = processes[i].finish_time -
processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
    }

    total_time = current_time;
}


void srt_scheduling(Process processes[], int process_count, GanttSegment
gantt_chart[], int& gantt_count, int& total_time) {

    for (int i = 0; i < process_count - 1; i++) {
        for (int j = i + 1; j < process_count; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time ||
                (processes[i].arrival_time == processes[j].arrival_time && processes[i].pid
> processes[j].pid)) {

                Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    gantt_count = 0;
    total_time = 0;
    int current_time = 0;
    int completed = 0;
    string last_pid = "";

    while (completed != process_count) {

        int idx = -1;
        int min_remaining = INT32_MAX;

        for (int i = 0; i < process_count; i++) {
            if (processes[i].arrival_time <= current_time && processes[i].remaining_time
> 0) {
                if (processes[i].remaining_time < min_remaining) {
                    min_remaining = processes[i].remaining_time;
                    idx = i;
                }
                else if (processes[i].remaining_time == min_remaining) {
                    if (processes[i].arrival_time < processes[idx].arrival_time) {
                        idx = i;
                    }
                }
            }
        }

        if (idx != -1) {
            if (last_pid != processes[idx].pid) {
                if (gantt_count < MAX_GANTT_SEGMENTS) {
                    gantt_chart[gantt_count].pid = processes[idx].pid;
                    gantt_chart[gantt_count].start_time = current_time;
                    gantt_chart[gantt_count].end_time = current_time + 1;
                    gantt_count++;
                }
            }
            else {

                gantt_chart[gantt_count - 1].end_time += 1;
            }

            processes[idx].remaining_time -= 1;
            current_time += 1;

                       if (processes[idx].remaining_time == 0) {
                processes[idx].finish_time = current_time;
                processes[idx].turnaround_time = processes[idx].finish_time -
processes[idx].arrival_time;
                processes[idx].waiting_time = processes[idx].turnaround_time -
processes[idx].burst_time;
                completed++;
            }

            last_pid = processes[idx].pid;
        }
        else {

            if (last_pid != "Idle") {
                if (gantt_count < MAX_GANTT_SEGMENTS) {
                    gantt_chart[gantt_count].pid = "Idle";
                    gantt_chart[gantt_count].start_time = current_time;
                    gantt_chart[gantt_count].end_time = current_time + 1;
                    gantt_count++;
                }
            }
            else {

                gantt_chart[gantt_count - 1].end_time += 1;
            }
            current_time += 1;
            last_pid = "Idle";
        }
    }

    total_time = current_time;
}


void rr_scheduling(Process processes[], int process_count, int quantum,
GanttSegment gantt_chart[], int& gantt_count, int& total_time) {

    for (int i = 0; i < process_count - 1; i++) {
        for (int j = i + 1; j < process_count; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time ||
                (processes[i].arrival_time == processes[j].arrival_time && processes[i].pid
> processes[j].pid)) {

                Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
```

```
gantt_count = 0;
total_time = 0;
int current_time = 0;
int completed = 0;
queue<int> ready_queue;
string last_pid = "";


for (int i = 0; i < process_count; i++) {
    if (processes[i].arrival_time <= current_time) {
        ready_queue.push(i);
    }
    else {
        break;
    }
}

int index = ready_queue.empty() ? 0 : ready_queue.back() + 1;

while (completed != process_count) {
    if (!ready_queue.empty()) {
        int idx = ready_queue.front();
        ready_queue.pop();


        int exec_time = min(quantum, processes[idx].remaining_time);


        if (last_pid != processes[idx].pid) {
            if (gantt_count < MAX_GANTT_SEGMENTS) {
                gantt_chart[gantt_count].pid = processes[idx].pid;
                gantt_chart[gantt_count].start_time = current_time;
                gantt_chart[gantt_count].end_time = current_time + exec_time;
                gantt_count++;
            }
        }
        else {

            gantt_chart[gantt_count - 1].end_time += exec_time;
        }

        current_time += exec_time;
        processes[idx].remaining_time -= exec_time;

        for (int i = index; i < process_count; i++) {
            if (processes[i].arrival_time <= current_time) {
                ready_queue.push(i);
```

```
                    index++;
                }
                else {
                    break;
                }
            }

            if (processes[idx].remaining_time > 0) {
                ready_queue.push(idx);
            }
            else {
                processes[idx].finish_time = current_time;
                processes[idx].turnaround_time = processes[idx].finish_time -
processes[idx].arrival_time;
                processes[idx].waiting_time = processes[idx].turnaround_time -
processes[idx].burst_time;
                completed++;
            }

            last_pid = processes[idx].pid;
        }
        else {

            if (index < process_count) {
                if (last_pid != "Idle") {
                    if (gantt_count < MAX_GANTT_SEGMENTS) {
                        gantt_chart[gantt_count].pid = "Idle";
                        gantt_chart[gantt_count].start_time = current_time;
                        gantt_chart[gantt_count].end_time = processes[index].arrival_time;
                        gantt_count++;
                    }
                }
                else {

                    gantt_chart[gantt_count - 1].end_time = processes[index].arrival_time;
                }
                current_time = processes[index].arrival_time;


                ready_queue.push(index);
                index++;
            }
            else {

                break;
            }
            last_pid = "Idle";
        }
    }

    total_time = current_time;
}


void print_gantt_chart(GanttSegment gantt_chart[], int gantt_count, const string&
title) {
    cout << "\n=== " << title << " Gantt Chart ===\n";
    // عرض PID
    for (int i = 0; i < gantt_count; i++) {
        cout << "| " << gantt_chart[i].pid << "  ";
    }
    cout << "|\n";


    if (gantt_count == 0) {
        cout << "0\n";
        return;
    }

    cout << gantt_chart[0].start_time;
    for (int i = 0; i < gantt_count; i++) {
        cout << "      " << gantt_chart[i].end_time;
    }
    cout << "\n";
}


void print_metrics(Process processes[], int process_count, const string&
algorithm_name, int total_time, GanttSegment gantt_chart[], int gantt_count) {
    cout << "\n=== " << algorithm_name << " Scheduling ===\n";
    cout << left << setw(10) << "Process"
        << setw(10) << "Arrival"
        << setw(10) << "Burst"
        << setw(10) << "Finish"
        << setw(10) << "Waiting"
        << setw(12) << "Turnaround" << "\n";

    double total_waiting = 0;
    double total_turnaround = 0;

    for (int i = 0; i < process_count; i++) {
        cout << left << setw(10) << processes[i].pid
            << setw(10) << processes[i].arrival_time
            << setw(10) << processes[i].burst_time
            << setw(10) << processes[i].finish_time
            << setw(10) << processes[i].waiting_time
            << setw(12) << processes[i].turnaround_time << "\n";

        total_waiting += processes[i].waiting_time;
        total_turnaround += processes[i].turnaround_time;
    }

    double avg_waiting = total_waiting / process_count;
    double avg_turnaround = total_turnaround / process_count;


    double cpu_busy_time = 0;
    for (int i = 0; i < gantt_count; i++) {
        if (gantt_chart[i].pid != "Idle") {
            cpu_busy_time += (gantt_chart[i].end_time - gantt_chart[i].start_time);
        }
    }
    double cpu_util = (cpu_busy_time / total_time) * 100.0;

    cout << fixed << setprecision(2);
    cout << "avg waiting: " << avg_waiting << "\n";
    cout << "avg turnaround " << avg_turnaround << "\n";
    cout << "utilization CPU: " << cpu_util << "%\n";
}

int main() {

    string file_path = "/Users/rawanshamali/Desktop/OSproject/OSproject/
processes.txt";




    Process processes[MAX_PROCESSES];
    int process_count = 0;
    int quantum = 2;


    quantum = read_processes(file_path, processes, process_count, quantum);


    GanttSegment fcfs_chart[MAX_GANTT_SEGMENTS];
    int fcfs_count = 0;
    int fcfs_total_time = 0;
    fcfs_scheduling(processes, process_count, fcfs_chart, fcfs_count,
fcfs_total_time);
```

```
    print_gantt_chart(fcfs_chart, fcfs_count, "FCFS");
    print_metrics(processes, process_count, "FCFS", fcfs_total_time, fcfs_chart,
fcfs_count);


    for (int i = 0; i < process_count; i++) {
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].finish_time = 0;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }


    GanttSegment srt_chart[MAX_GANTT_SEGMENTS];
    int srt_count = 0;
    int srt_total_time = 0;
    srt_scheduling(processes, process_count, srt_chart, srt_count, srt_total_time);
    print_gantt_chart(srt_chart, srt_count, "SRT");
    print_metrics(processes, process_count, "SRT", srt_total_time, srt_chart,
srt_count);


    for (int i = 0; i < process_count; i++) {
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].finish_time = 0;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }


    GanttSegment rr_chart[MAX_GANTT_SEGMENTS];
    int rr_count = 0;
    int rr_total_time = 0;
    rr_scheduling(processes, process_count, quantum, rr_chart, rr_count,
rr_total_time);
    print_gantt_chart(rr_chart, rr_count, "Round-Robin");
    print_metrics(processes, process_count, "Round-Robin", rr_total_time, rr_chart,
rr_count);

    return 0;
}
```