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

using namespace std;

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

    Process(string id, int arrival, int burst) :
        pid(id), arrival_time(arrival), burst_time(burst),
        remaining_time(burst), finish_time(0),
        waiting_time(0), turnaround_time(0) {
    }
};
struct GanttChart {
    string pid;
    int start_time;
    int end_time;

    GanttChart(string id, int start, int end) :
        pid(id), start_time(start), end_time(end) {
    }
};


pair<vector<Process>, int> read_processes(string file_path) {
    vector<Process> processes;
    int quantum = 2;
    ifstream infile(file_path);
    if (!infile.is_open()) {
        cerr << "can't open the file " << file_path << endl;
        exit(1);
    }
    string line;
    while (getline(infile, line)) {

        if (line.empty() || line[0] == '#') continue;
        vector<string> parts;
        string part;
        for (char c : line) {
            if (c == ' ' || c == '\t') {
                if (!part.empty()) {
                    parts.push_back(part);
                    part = "";
                }
            }
            else {
                part += c;
```

```
            }
        }
        if (!part.empty()) parts.push_back(part);
        if (parts.empty()) continue;
        if (parts[0] == "Quantum" || parts[0] == "quantum") {
            if (parts.size() >= 2) {
                quantum = stoi(parts[1]);
            }
        }
        else {
            if (parts.size() < 3) {
                cerr << "invalid process in file" << endl;
                exit(1);
            }
            string pid = parts[0];
            int arrival = stoi(parts[1]);
            int burst = stoi(parts[2]);
            processes.emplace_back(pid, arrival, burst);
        }
    }
    infile.close();
    return { processes, quantum };
}

pair<vector<GanttChart>, vector<Process>> fcfs_scheduling(vector<Process> processes) {
    sort(processes.begin(), processes.end(), [&](Process a, Process b) {
        if (a.arrival_time != b.arrival_time)
            return a.arrival_time < b.arrival_time;
        else
            return a.pid < b.pid;
        });

    vector<GanttChart> gantt_chart;
    int current_time = 0;

    for (auto& process : processes) {
        if (current_time < process.arrival_time) {
            // CPU في حالة Idle
            gantt_chart.emplace_back("Idle", current_time, process.arrival_time);
            current_time = process.arrival_time;
        }
        // بدء تنفيذ العملية
        gantt_chart.emplace_back(process.pid, current_time, current_time + process.burst_time);
        process.waiting_time = current_time - process.arrival_time;
        current_time += process.burst_time;
        process.finish_time = current_time;
        process.turnaround_time = process.finish_time - process.arrival_time;
    }

    return { gantt_chart, processes };
}

// خوارزمية Shortest Remaining Time (SRT)
pair<vector<GanttChart>, vector<Process>> srt_scheduling(vector<Process> processes) {
    // ترتيب العمليات حسب وقت الوصول
    sort(processes.begin(), processes.end(), [&](Process a, Process b) {
        if (a.arrival_time != b.arrival_time)
            return a.arrival_time < b.arrival_time;
        else
```

```
        return a.pid < b.pid;
    });

int n = processes.size();
int completed = 0;
int current_time = 0;
vector<GanttChart> gantt_chart;
vector<Process> proc = processes; // نسخة من العمليات
string last_pid = "";

while (completed != n) {
    // العثور على العملية التي وصلت وأقل وقت متبقي
    int idx = -1;
    int min_remaining = INT32_MAX;
    for (int i = 0; i < n; i++) {
        if (proc[i].arrival_time <= current_time && proc[i].remaining_time > 0) {
            if (proc[i].remaining_time < min_remaining) {
                min_remaining = proc[i].remaining_time;
                idx = i;
            }
            else if (proc[i].remaining_time == min_remaining) {
                if (proc[i].arrival_time < proc[idx].arrival_time) {
                    idx = i;
                }
            }
        }
    }

    if (idx != -1) {
        // إذا تغيرت العملية الحالية، نقوم بإضافة جزء جديد لـ Gantt Chart
        if (last_pid != proc[idx].pid) {
            gantt_chart.emplace_back(proc[idx].pid, current_time, current_time + 1);
            last_pid = proc[idx].pid;
        }
        else {
            gantt_chart.back().end_time += 1;
        }
        proc[idx].remaining_time -= 1;
        current_time += 1;
        if (proc[idx].remaining_time == 0) {
            proc[idx].finish_time = current_time;
            proc[idx].turnaround_time = proc[idx].finish_time - proc[idx].arrival_time;
            proc[idx].waiting_time = proc[idx].turnaround_time - proc[idx].burst_time;
            completed++;
        }
    }
    else {
        // CPU في حالة Idle
        if (last_pid != "Idle") {
            gantt_chart.emplace_back("Idle", current_time, current_time + 1);
            last_pid = "Idle";
        }
        else {
            gantt_chart.back().end_time += 1;
        }
        current_time += 1;
    }
}
```

```
    // إعادة ترتيب العمليات حسب PID
    sort(proc.begin(), proc.end(), [&](Process a, Process b) {
        return a.pid < b.pid;
        });

    return { gantt_chart, proc };
}

// خوارزمية Round-Robin (RR) المعدلة
pair<vector<GanttChart>, vector<Process>> rr_scheduling(vector<Process> processes, int
quantum) {
    // ترتيب العمليات حسب وقت الوصول
    sort(processes.begin(), processes.end(), [&](Process a, Process b) {
        if (a.arrival_time != b.arrival_time)
            return a.arrival_time < b.arrival_time;
        else
            return a.pid < b.pid;
        });

    int n = processes.size();
    int completed = 0;
    int current_time = 0;
    vector<GanttChart> gantt_chart;
    queue<int> ready_queue; // يحتوي على مؤشرات العمليات
    vector<Process> proc = processes; // نسخة من العمليات
    int index = 0; // لمتابعة العمليات التي وصلت

    string last_pid = "";

    while (completed != n) {
        // إضافة العمليات التي وصلت إلى ready_queue
        while (index < n && proc[index].arrival_time <= current_time) {
            ready_queue.push(index);
            index++;
        }

        if (!ready_queue.empty()) {
            int i = ready_queue.front();
            ready_queue.pop();

            // تحديد وقت التنفيذ
            int exec_time = min(quantum, proc[i].remaining_time);

            // تحديث Gantt Chart
            if (last_pid != proc[i].pid) {
                gantt_chart.emplace_back(proc[i].pid, current_time, current_time + exec_time);
                last_pid = proc[i].pid;
            }
            else {
                gantt_chart.back().end_time += exec_time;
            }

            // تحديث الوقت الحالي
            current_time += exec_time;
            proc[i].remaining_time -= exec_time;

            // إضافة العمليات التي وصلت خلال فترة التنفيذ
            while (index < n && proc[index].arrival_time <= current_time) {
```

```
            ready_queue.push(index);
            index++;
        }

        if (proc[i].remaining_time > 0) {
            ready_queue.push(i);
        }
        else {
            proc[i].finish_time = current_time;
            proc[i].turnaround_time = proc[i].finish_time - proc[i].arrival_time;
            proc[i].waiting_time = proc[i].turnaround_time - proc[i].burst_time;
            completed++;
        }
    }
    else {
        // إذا لم يكن هناك عمليات جاهزة، تخطي الزمن إلى وقت وصول العملية التالية
        if (index < n) {
            // إضافة فترة Idle واحدة تغطي الفارق الزمني
            if (last_pid != "Idle") {
                gantt_chart.emplace_back("Idle", current_time, proc[index].arrival_time);
                last_pid = "Idle";
            }
            else {
                gantt_chart.back().end_time = proc[index].arrival_time;
            }
            current_time = proc[index].arrival_time;
        }
    }
}

// إعادة ترتيب العمليات حسب PID
sort(proc.begin(), proc.end(), [&](Process a, Process b) {
    return a.pid < b.pid;
    });

return { gantt_chart, proc };
}

// كنص Gantt Chart دالة لعرض
void print_gantt_chart(vector<GanttChart> gantt_chart, string title) {
cout << "\n=== " << title << " Gantt Chart ===\n";
// عرض PID
for (auto& segment : gantt_chart) {
    cout << "| " << segment.pid << " ";
}
cout << "|\n";

// عرض الخط الزمني
if (gantt_chart.empty()) {
    cout << "0\n";
    return;
}

cout << gantt_chart[0].start_time;
for (auto& segment : gantt_chart) {
    cout << "      " << segment.end_time;
}
cout << "\n";
}
```

```
// دالة لعرض المؤشرات مع حساب CPU Utilization بشكل صحيح
void print_metrics(vector<Process> processes, string algorithm_name, int total_time,
vector<GanttChart> gantt_chart) {
    cout << "\n=== " << algorithm_name << " Scheduling ===\n";
    cout << left << setw(10) << "Process"
        << setw(12) << "Arrival"
        << setw(10) << "Burst"
        << setw(12) << "Finish"
        << setw(12) << "Waiting"
        << setw(15) << "Turnaround" << "\n";
    for (auto& process : processes) {
        cout << left << setw(10) << process.pid
            << setw(12) << process.arrival_time
            << setw(10) << process.burst_time
            << setw(12) << process.finish_time
            << setw(12) << process.waiting_time
            << setw(15) << process.turnaround_time << "\n";
    }
    // حساب المتوسطات
    double total_waiting = 0;
    double total_turnaround = 0;
    for (auto& p : processes) {
        total_waiting += p.waiting_time;
        total_turnaround += p.turnaround_time;
    }
    double avg_waiting = total_waiting / processes.size();
    double avg_turnaround = total_turnaround / processes.size();

    // حساب CPU Utilization بشكل صحيح
    double cpu_busy_time = 0;
    for (auto& segment : gantt_chart) {
        if (segment.pid != "Idle") {
            cpu_busy_time += (segment.end_time - segment.start_time);
        }
    }
    double cpu_util = (cpu_busy_time / total_time) * 100.0;

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

int main() {
    // Specify the file path
    // Use absolute path for Windows
    string file_path = "C:\\Users\\abed alrehman\\Desktop\\processes.txt";

    // Alternatively, use a relative path if the file is in the same directory
    // string file_path = "processes.txt";

    // Read processes and quantum from the file
    pair<vector<Process>, int> input = read_processes(file_path);
    vector<Process> processes = input.first;
    int quantum = input.second;

    // FCFS Scheduling
    pair<vector<GanttChart>, vector<Process>> fcfs = fcfs_scheduling(processes);
    // Calculate total time
```

```
    int total_time_fcfs = 0;
    if (!fcfs.first.empty())
        total_time_fcfs = fcfs.first.back().end_time;
    print_gantt_chart(fcfs.first, "FCFS");
    print_metrics(fcfs.second, "FCFS", total_time_fcfs, fcfs.first);

    // SRT Scheduling
    pair<vector<GanttChart>, vector<Process>> srt = srt_scheduling(processes);
    int total_time_srt = 0;
    if (!srt.first.empty())
        total_time_srt = srt.first.back().end_time;
    print_gantt_chart(srt.first, "SRT");
    print_metrics(srt.second, "SRT", total_time_srt, srt.first);

    // Round-Robin Scheduling
    pair<vector<GanttChart>, vector<Process>> rr = rr_scheduling(processes, quantum);
    int total_time_rr = 0;
    if (!rr.first.empty())
        total_time_rr = rr.first.back().end_time;
    print_gantt_chart(rr.first, "Round-Robin");
    print_metrics(rr.second, "Round-Robin", total_time_rr, rr.first);

    return 0;
}
```