fit@hcmus

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

ASSIGNMENT 02.01

IMPLEMENT SCHEDULING ALGORITHMS

Các thành viên trong nhóm:

1. Lê Trọng Anh Tú MSSV: 20127091
2. Phan Tuấn Khải MSSV: 20127524
3. Lê Đăng Khoa MSSV: 20127533

Giảng viên hướng dẫn: Ths. Thái Hùng Văn

Mục lục

1 Giới thiệu **3**

2 Kết quả đạt được **3**

3 Một số lưu ý **3**
3.1 Ký hiệu 3
3.2 Mã nguồn và cài đặt 3
3.3 Các giả định trong bài làm 3

4 Phần trả lời câu hỏi **4**
4.1 First−Come, First−Served (FCFS) Scheduling 4
4.2 Round−Robin (RR) Scheduling 5
4.3 Shortest−Job−First (SJF) Scheduling 8
4.4 Shortest−Remaining−Time−Next (SRTN) Scheduling 10
4.5 Priority Scheduling (Preemptive version) 11
4.6 Priority Scheduling (Nonpreemptive version) 13
4.7 Kết quả sau khi chạy chương trình 14

1 Giới thiệu

Bài tập thực hành môn Hệ điều hành lớp 20CLC07 được thực hiện bởi ba thành viên đề cập ở trang bìa. Nội dung của bài tập yêu cầu viết chương trình để triển khai các thuật toán điều phối tiến trình đã được học tại lớp.

Toàn bộ mã nguồn trong thư mục *SourceCode* ở file nộp và mã nguồn được trình bày trong các phần tiếp theo được nhóm thống nhất và sử dụng ngôn ngữ C/C++.

Dưới đây sẽ là phần phân công công việc, các phần đã làm được/chưa làm được và trình bày chi tiết câu trả lời của nhóm.

2 Kết quả đạt được

Trong thời gian thực hiện (từ 06/04/2022 đến 13/04/2022), nhóm đã hoàn thành 100% các yêu cầu được đưa ra trong bài tập. Dưới đây là bảng phân công công việc của nhóm.

STT	Sinh viên thực hiện	Nội dung	Mức độ hoàn thành
1	Phan Tuấn Khải	Thảo luận các thuật toán điều phối sẽ làm	100%
2		Triển khai thuật FCFS	100%
3		Triển khai thuật SRTN	100%
4	Lê Trọng Anh Tú	Thảo luận các thuật toán điều phối sẽ làm	100%
5		Triển khai thuật SJF	100%
6		Triển khai thuật Priority (độc quyền)	100%
7	Lê Đăng Khoa	Thảo luận các thuật toán điều phối sẽ làm	100%
8		Triển khai thuật RR	100%
9		Triển khai thuật Priority (không độc quyền)	100%

3 Một số lưu ý

3.1 Ký hiệu

- Một số ký hiệu nên nhớ
  - PQ: nếu không chú thích gì thêm, mặc định sẽ là Priority Queue chứa các process đang đợi để xử lý.
  - p: nếu không chú thích gì thêm, mặc định sẽ là process.
  - T: nếu không chú thích gì thêm, mặc định sẽ là thời gian hiện hành của quá trình xử lý.

3.2 Mã nguồn và cài đặt

- Câu lệnh biên dịch: **g++ −std=c++14 *.cpp −o *.exe && *.exe**
- Chạy với phiên bản c++14 hoặc mới hơn.

3.3 Các giả định trong bài làm

- Luôn tồn tại tiến trình có thời gian vào hệ thống (Arrival Time) là 0.
- Độ ưu tiên (Priority) của tiến trình có giá trị càng nhỏ → Độ ưu tiên càng lớn.

4 Phần trả lời câu hỏi

Sau khi thảo luận, nhóm quyết định làm 6 thuật toán cơ bản về điều phối. Từng thuật toán sẽ được mô tả chi tiết trong các mục sắp tới.

4.1 First−Come, First−Served (FCFS) Scheduling

- Ý tưởng thuật toán: Tới trước thì phục vụ trước, tiến trình nào tới đầu tiên trong hệ thống sẽ được thực thi đầu tiên.
- Là thuật toán điều phối độc quyền − Xử lý hết một tiến trình thì mới chuyển qua tiến trình khác. Nếu trong thời gian thực thi mà có tiến trình mới đi vào thì đưa tiến trình đó vào hàng đợi.
- Ưu điểm
  - Đơn giản, dễ cài đặt.
- Nhược điểm
  - Không quan tâm đến độ ưu tiên của tiến trình.
  - Hiệu năng thấp, không hiệu quả.
  - Có thể xảy ra tình trạng nếu tiến trình vào đầu tiên thực thi quá lâu → Tiến trình độc chiếm CPU.
- Triển khai thuật toán

  − Đầu tiên ta được cung cấp tham số đầu vào là mảng các Process trong hệ thống với mỗi Process có các thông tin như tên, thời gian vào hệ thống, thời gian xử lý và độ ưu tiên.

  − Sắp xếp các tiến trình theo thứ tự tăng dần của arrivalTime.

  − Bắt đầu xử lý với tham số T = 0, lặp cho đến khi nào mảng Process empty hay nói cách khác là các Process đã được xử lý hết.

  − Nếu tại thời điểm T mà có Process nào đó trong mảng Process đầu vào bằng với giá trị T thì tiến hành xử lý Process đó và xoá Process đó ra khỏi mảng Process đầu vào.
- Mã nguồn của thuật toán

```
void FCFS(vector<Process> p, int q){
    vector<string> chart;
    int cntP = p.size();
    sort(p.begin(), p.end(), [&](const Process& a, const Process& b){return a.
    arrivalTime < b.arrivalTime;});
    map<string, pair<int, int>> t;
    int time = 0;
    for(; p.size(); ++time){
        if(time == p[0].arrivalTime){
            chart.push_back(to_string(time));
            chart.push_back(p[0].name);
            if(p.size() > 1 && time + 1 + p[0].burst < p[1].arrivalTime){
                chart.push_back(to_string(time + p[0].burst));
                chart.push_back("IDLE");
            }
            time += p[0].burst;
            t[p[0].name].first = time - p[0].arrivalTime;
```

```
            t[p[0].name].second = t[p[0].name].first - p[0].burst;
            time--;
            p.erase(p.begin());
        }
        else if(time > p[0].arrivalTime){
            chart.push_back(to_string(time));
            chart.push_back(p[0].name);
            if(p.size() > 1 && time + 1 + p[0].arrivalTime < p[1].arrivalTime){
                chart.push_back(to_string(time + p[0].arrivalTime + p[0].burst));
                chart.push_back("IDLE");
            }
            time += p[0].burst;
            t[p[0].name].first = time  - p[0].arrivalTime;
            t[p[0].name].second = t[p[0].name].first - p[0].burst;
            time--;
            p.erase(p.begin());
        }
    }
    chart.push_back(to_string(time));
    freopen("FCFS.txt", "w", stdout);
    cout << "Scheduling chart: ";
    cout << chart[0];
    for(int i = 1; i < chart.size(); ++i) cout << "~" << chart[i];
    cout << '\n';
    int totalTT = 0;
    int totalWT = 0;
    for(auto m : t){
        totalTT += m.second.first;
        totalWT += m.second.second;
        cout << m.first << ": \t TT = " << m.second.first << ' ' << "WT = " << m.second.
    second << '\n';
    }
    cout << "Average:\t TT = " << 1.0 * totalTT / cntP << "\tWT = " << 1.0 * totalWT /
    cntP;
    fclose(stdout);
}
```

4.2 Round−Robin (RR) Scheduling

- Ý tưởng thuật toán: Lấy tiến trình đầu tiên trong hàng đợi với một khoảng thời gian quantum Q, nếu sau thời gian quantum tiến trình vẫn chưa thực hiện xong thì cũng kết thúc và thêm vào cuối hàng đợi để đợi lượt thực hiện tiếp theo.
- Là thuật toán không độc quyền. Thông thường được sử dụng trong hệ thống chia sẻ thời gian (Time−sharing system), hệ thống luân phiên CPU giữa các tiến trình → Có cảm giác như các tác vụ được xử lý song song đồng thời.
- Ưu điểm
  - Công bằng giữa các tiến trình, loại bỏ vấn đề độc chiếm CPU.
  - Giảm thời gian chờ đợi (Waiting Time) của các tiến trình có CPU Burst ngắn.
- Nhược điểm

- Vẫn chưa xét đến độ ưu tiên của tiến trình.
- Làm sao để chọn ra khoảng thời gian quantum phù hợp. Nếu thời gian quantum quá lớn → Có thể coi như thuật FCFS. Nếu thời gian quantum quá nhỏ → Chuyển đổi giữa các CPU quá nhiều lần.

- Triển khai thuật toán

  – Thuật toán nhận 2 tham số đầu vào là mảng các Process và thời gian quantum.

  – Khởi tạo vòng lập bắt đầu tại T = 0. Thêm tất cả các tiến trình từ thời điểm T đến thời điểm T + q vào một danh sách được xây dựng theo cấu trúc dữ liệu dạng hàng đợi (queue).

  – Lấy tiến trình trên cùng của hàng đợi, nếu CPU burst của tiến trình > q, tức tiến trình vẫn chưa hoàn thành, khi đó CPU burst = CPU burst − q, T = T + q. Tiến trình đó sẽ được thêm vào cuối hàng đợi. Trường hợp CPU burst ≤ q, khi đó tiến trình sẽ hoàn thành trong khoảng thời gian quantum, CPU burst = 0 và T = T + CPU Burst

  – Khi tiến trình hoàn thành, tiến hành tính TT và WT theo công thức: TT = T − p.arrivalTime; WT = TT − p.CPUBurst

  – Tiếp tục vòng lặp cho đến khi CPU burst của tất cả các tiến trình cần lập lịch bằng 0 và không còn tiến trình nào trong danh sách hàng đợi.

- Mã nguồn của thuật toán

```
void RR(vector<Process> p, int quantum) {
    stringstream ss;
    queue<string> process;
    vector<string> processName;
    vector<int> arrivalTime;
    vector<int> cpuBurst;
    for (int i = 0; i < (int)p.size(); i++) {
        processName.push_back(p[i].name);
        arrivalTime.push_back(p[i].arrivalTime);
        cpuBurst.push_back(p[i].burst);
    }
    vector<int> oldBurst = cpuBurst;
    vector<int> oldArrival = arrivalTime;
    int currentTime = 0;
    int* WT = new int[p.size()]();
    int* TT = new int[p.size()]();
    string oldProcess = "";
    int sumTime = 0;
    for (int i = 0; i < (int)p.size(); i++) sumTime += cpuBurst[i];
    while (true) {
        auto start = find(arrivalTime.begin(), arrivalTime.end(), 0);
        if (start == arrivalTime.end()) break;
        arrivalTime[distance(arrivalTime.begin(), start)] = -1;
        process.push(processName[distance(arrivalTime.begin(), start)]);
    }
    while (currentTime <= sumTime) {
        int indexProcess = -1;
        for (int i = currentTime + 1; i <= currentTime + quantum; i++) {
            bool flag = true;
            while (flag) {
                auto it = find(arrivalTime.begin(), arrivalTime.end(), i);
                if (it != arrivalTime.end()) {
```

```
                    int temp = distance(arrivalTime.begin(), it);
                    arrivalTime[temp] = -1;
                    process.push(processName[temp]);
                }
                else flag = false;
            }
        }
        if (process.size() == 0) {
            currentTime++;
            continue;
        }
        string ProcessInProcess = process.front();
        process.pop();
        auto it = find(processName.begin(), processName.end(), ProcessInProcess);
        if (it != processName.end()) indexProcess = distance(processName.begin(), it);
        if (oldProcess != ProcessInProcess) {
            ss << currentTime << " ~ " << ProcessInProcess << " ~ ";
            oldProcess = ProcessInProcess;
        }
        if (cpuBurst[indexProcess] > quantum) {
            currentTime += quantum;
            cpuBurst[indexProcess] -= quantum;
        }
        else {
            currentTime += cpuBurst[indexProcess];
            cpuBurst[indexProcess] = 0;
            TT[indexProcess] = currentTime - oldArrival[indexProcess];
            WT[indexProcess] = TT[indexProcess] - oldBurst[indexProcess];
        }
        if (cpuBurst[indexProcess] > 0) process.push(ProcessInProcess);
        if (currentTime == sumTime) ss << currentTime;
    }
    double AVG_WT = 0, AVG_TT = 0;
    for (int i = 0; i < (int)p.size(); i++) {
        AVG_WT += WT[i];
        AVG_TT += TT[i];
    }
    AVG_WT /= p.size();
    AVG_TT /= p.size();
    string chart = ss.str();
    ofstream out;
    out.open("RR.txt", ios::out);
    out << "Scheduling chart: " << endl;
    out << "\t" << chart << endl;
    for (int i = 0; i < (int)p.size(); i++) {
        out << processName[i] << ":\t" << "TT = " << TT[i] << " WT = " << WT[i] << endl;
    }
    out << "Average:  TT = " << AVG_TT << " WT = " << AVG_WT;
    out.close();
}
```

4.3 Shortest−Job−First (SJF) Scheduling

- Ý tưởng thuật toán: Tại thời điểm CPU vừa kết thúc xử lý, tiến trình nào trong hàng đợi có CPU Burst nhỏ nhất sẽ được lấy ra để chạy.
- Là thuật toán điều phối độc quyền (Nonpreemptive scheduling). Giả sử tại T = 2, P1 đang chạy (CPU Burst = 6) và P2 vừa đi vào (CPU Burst = 1) thì thuật toán vẫn tiếp tục chạy P1 cho đến khi kết thúc hoàn toàn thì mới đưa CPU cho P2 chạy (nếu trong hàng đợi P2 có CPU Burst nhỏ nhất).
- Ưu điểm
  - Đơn giản, dễ cài đặt.
  - Giảm thời gian TT (Turnaround Time − thời gian ở trong hệ thống) và WT (Waiting Time − thời gian chờ được thực thi) đối với các tiến trình có CPU Burst nhỏ.
  - Tối đa số lượng tiến trình cần hoàn tất trong một đơn vị thời gian.
- Nhược điểm
  - Một tiến trình quan trọng có CPU Burst cao có thể không bao giờ được chạy nếu hàng đợi có quá nhiều tiến trình ít quan trọng hơn với CPU Burst nhỏ.
  - Phải dự đoán tất cả CPU Burst của từng tiến trình → Đây cũng là một bài toán phức tạp không kém gì các thuật toán điều phối.
- Triển khai thuật toán
  - Đầu tiên ta được cung cấp tham số đầu vào là mảng các Process trong hệ thống với mỗi Process có các thông tin như tên, thời gian vào hệ thống, thời gian xử lý và độ ưu tiên.
  - Sắp xếp tiến trình theo thứ tự tăng dần của thời gian vào hệ thống.
  - Tại T = 0, ta duyệt mảng các Process đã được sắp xếp tại bước trên để thêm các Process có thời gian vào hệ thống = 0 vào hàng đợi.
    - Nếu thời gian đầu vào của một tiến trình khác 0 thì tất cả các tiến trình nằm sau nó đều sẽ khác 0 cho ta đã sắp xếp tăng dần → Thoát khỏi vòng lặp duyệt mảng.
  - Ta duy trì hai cấu trúc trong thuật toán. Thuật toán kết thúc khi số lượng phần tử trong 2 cấu trúc được nhắc tới đều bằng 0.
    - Hàng đợi cho các tiến trình đã đi vào hệ thống mà chưa được chạy hoặc chạy chưa hết thời gian của CPU Burst. Cấu trúc này sẽ sắp xếp các tiến trình theo thứ tự tăng dần về CPU Burst đáp ứng ý tưởng thuật toán SJF được nêu ban đầu.
    - Mảng các tiến trình chưa đi vào hệ thống.
  - Nếu hàng đợi tồn tại ít nhất 1 phần tử, lấy phần tử đầu tiên trong hàng đợi ra và cập nhật thời gian thực lên một khoảng CPU Burst của tiến trình ta vừa lấy ra. Đồng thời duyệt tiếp mảng các tiến trình chưa được vào hệ thống, nếu thời gian đầu vào của tiến trình đó nhỏ hơn thời gian thực (ví dụ P2 có thời gian đầu vào là 3 và thời gian thực là 4) thì xoá phần tử đó ra khỏi mảng chưa vào hệ thống và thêm vào hàng đợi.
  - Nếu hàng đợi không còn phần tử nào mà vẫn còn phần tử nằm trong mảng các tiến trình chưa được vào hệ thống thì ta cập nhật thời gian thực lên thời gian đầu vào của tiến trình đầu tiên trong mảng và làm tương tự bước trên.

- Mã nguồn của thuật toán

```
void SJF(vector<Process> p, int q) {
    ofstream out("SJF.txt");
    int totalTime = 0, totalTT = 0, totalWT = 0, count = p.size();
    sort(p.begin(), p.end(), [](const Process& a, const Process& b) { return a.
    arrivalTime < b.arrivalTime; });
    vector<Process> temp;
    while (p.size()) {
        if (p[0].arrivalTime == 0) {
            temp.push_back(p[0]);
            p.erase(p.begin());
        }
        else break;
    }
    map<string, int> TT, WT;
    out << "Scheduling chart: 0";
    while (p.size() != 0 || temp.size() != 0) {
        if (temp.size() == 0) {
            out << " - " << p[0].arrivalTime;
            int push = p[0].arrivalTime;
            totalTime = p[0].arrivalTime;
            while (p.size()) {
                if (p[0].arrivalTime == push) {
                    temp.push_back(p[0]);
                    p.erase(p.begin());
                }
                else break;
            }
        }
        sort(temp.begin(), temp.end(), [](const Process& a, const Process& b) { return a
    .burst < b.burst; });
        totalTime += temp[0].burst;
        TT.insert({ temp[0].name, totalTime - temp[0].arrivalTime });
        WT.insert({ temp[0].name, totalTime - temp[0].burst - temp[0].arrivalTime });
        out << " ~" << temp[0].name << "~ " << totalTime;
        temp.erase(temp.begin());
        for (int i = 0; i < p.size(); i++) {
            if (totalTime >= p[i].arrivalTime) {
                temp.push_back(p[i]);
                p.erase(p.begin() + i);
                i--;
            }
        }
    }
    out << endl;
    for (auto x : TT) {
        totalTT += x.second;
        totalWT += WT[x.first];
        out << x.first << ":\t\tTT = " << x.second << "\tWT = " << WT[x.first] << endl;
    }
    out << "Average:\t\t\tTT = " << 1.0 * totalTT / count << "\t\tWT = " << 1.0 *
    totalWT / count;
    out.close();
}
```

4.4 Shortest−Remaining−Time−Next (SRTN) Scheduling

- Là biến thể không độc quyền của thuật toán SJF hay nói cách khác, sắp xếp theo thời gian thực thi còn lại của tiến trình. Ví dụ P1 đang chạy và thời gian thực thi còn lại của P1 là 5, P2 mới vừa vào hệ thống và thời gian thực thi còn lại của P2 là 3 thì thuật toán sẽ thu hồi CPU của P1 lại và cung cấp cho P2 để chạy. Nếu cả hai tiến trình đều cùng thời gian thực thi còn lại thì có 2 cách giải quyết:
  - "*Chung thuỷ*": Tiếp tục chạy tiến trình trước đó đang chạy, tiến trình mới sẽ được thêm vào hàng đợi.
  - "*Không chung thuỷ*": Lấy tiến trình mới vào hệ thống để chạy, tiến trình đang chạy trước đó được thêm vào hàng đợi.
- Triển khai thuật toán
  - Đầu tiên ta được cung cấp tham số đầu vào là mảng các p trong hệ thống với mỗi p có các thông tin như tên, thời gian vào hệ thống, thời gian xử lý và độ ưu tiên.
  - Sắp xếp lại mảng các p theo chiều giảm dần các arrivalTime.
  - Thêm các p có arrivalTime = 0 vào PQ trước.
  - Bắt đầu với T = 0. Tiến hành lặp cho đến khi mảng P empty và PQ empty thì dừng.
    - Thêm các p có arrivalTime = T vào PQ.
    - Lấy p có burst time còn lại ít nhất ra xử lý.
    - Nếu p-burst = 0 sau khi xử lý thì p đã được hoàn thành. Nếu sau khi xử lý mà p-burst chưa = 0 thì add vào lại PQ đợi xử lý tiếp.
- Mã nguồn của thuật toán

```
void SRTN(vector<Process> p, int q) {
    vector<Process> pp = p;
    priority_queue<Process, vector<Process>, Process> pq;
    sort(p.begin(), p.end(), [&](const Process& a, const Process& b) {return a.
   arrivalTime > b.arrivalTime; });
    vector<string> chart;
    map<string, int> TT, WT;
    Process cur = p.back();
    bool done = false;
    p.pop_back();
    while (p.back().arrivalTime == 0) {
        pq.push(p.back());
        p.pop_back();
    }
    int time = 0;
    chart.push_back(to_string(time));
    while (!pq.empty() || p.size() != 0) {
        while (p.size() != 0 && time == p.back().arrivalTime) {
            pq.push(p.back());
            p.pop_back();
        }
        if (!pq.empty()) {
            if (cur.burst <= 0 && done == true) {
                cur = pq.top();
                pq.pop();
```

```
                done ^= 1;
            }
            else if (pq.top().burst < cur.burst) {
                if (cur.burst > 0) pq.push(cur);
                chart.push_back(cur.name);
                chart.push_back(to_string(time));
                cur = pq.top();
                pq.pop();
            }
        }
        cur.burst--;
        time++;
        if (cur.burst == 0) {
            done = true;
            TT[cur.name] = time - cur.arrivalTime;
            chart.push_back(cur.name);
            chart.push_back(to_string(time));
            if (p.size() > 1 && time + 1 != p[1].arrivalTime && pq.empty()) {
                chart.push_back("IDLE");
                chart.push_back(to_string(p[1].arrivalTime));
            }
            else if (p.size() == 1 && time + 1 != p[1].arrivalTime && pq.empty()) {
                chart.push_back("IDLE");
                chart.push_back(to_string(p[0].arrivalTime));
            }
        }
    }
    if (cur.burst > 0) {
        TT[cur.name] = time + cur.burst - cur.arrivalTime;
        chart.push_back(cur.name);
        chart.push_back(to_string(time + cur.burst));
    }
    int totalTT = 0, totalWT = 0;
    for (auto _p : pp) {
        WT[_p.name] = TT[_p.name] - _p.burst;
        totalTT += TT[_p.name];
        totalWT += WT[_p.name];
    }
    freopen("SRTN.txt", "w", stdout);
    cout << "Scheduling chart: 0";
    for (int i = 1; i < chart.size(); ++i) cout << "~" << chart[i];
    cout << '\n';
    for (auto m : TT) {
        cout << m.first << ": \t TT = " << m.second << " WT = " << WT[m.first] << '\n';
    }
    cout << "Average: \t TT = " << 1.0 * totalTT / pp.size() << "\t WT = " << 1.0 *
   totalWT / pp.size();
    fclose(stdout);
}
```

4.5 Priority Scheduling (Preemptive version)

- Là thuật toán điều phối không độc quyền dựa trên độ ưu tiên của từng tiến trình.
- Ý tưởng thuật toán và triển khai thuật toán giống tương tự với SRTN, chỉ là hàng đợi của các tiến trình

sẽ sắp xếp theo độ ưu tiên chứ không sắp xếp theo CPU Burst.

- Mã nguồn của thuật toán

```
void PreemptivePriority(vector<Process> p, int q) {
    priority_queue<tuple<int, int, string>> process;
    vector<string> processName;
    vector<int> arrivalTime;
    vector<int> cpuBurst;
    vector<int> priority;
    for (int i = 0; i < (int)p.size(); i++) {
        processName.push_back(p[i].name);
        arrivalTime.push_back(p[i].arrivalTime);
        cpuBurst.push_back(p[i].burst);
        priority.push_back(p[i].priority * -1);
    }
    vector<int> oldBurst = cpuBurst;
    vector<int> oldArrival = arrivalTime;
    int currentTime = 0;
    int* WT = new int[p.size()]();
    int* TT = new int[p.size()]();
    string oldProcess = "";
    stringstream ss;
    int sumTime = 0;
    for (int i = 0; i < (int)p.size(); i++) sumTime += cpuBurst[i];
    while (currentTime <= sumTime) {
        int indexProcess = -1;
        while (true) {
            auto push = find(arrivalTime.begin(), arrivalTime.end(), currentTime);
            if (push != arrivalTime.end()) {
                int temp = distance(arrivalTime.begin(), push);
                arrivalTime[temp] = -1;
                process.push(make_tuple(priority[temp], INT_MAX - oldArrival[temp],
   processName[temp]));
            }
            else break;
        }
        if (process.size() == 0) {
            currentTime++;
            continue;
        }
        tuple<int, int, string> tupp = process.top();
        string ProcessInProcess = get<2>(tupp);
        process.pop();
        auto it = find(processName.begin(), processName.end(), ProcessInProcess);
        indexProcess = distance(processName.begin(), it);
        string Name = processName[indexProcess];
        if (oldProcess != Name) {
            ss << currentTime << " ~ " << Name << " ~ ";
            oldProcess = Name;
        }
        currentTime += 1;
        cpuBurst[indexProcess] -= 1;
        if (cpuBurst[indexProcess] == 0) {
            priority[indexProcess] = INT_MAX;
            TT[indexProcess] = currentTime - oldArrival[indexProcess];
```

```
            WT[indexProcess] = TT[indexProcess] - oldBurst[indexProcess];
        }
        else {
            process.push(make_tuple(priority[indexProcess], INT_MAX - oldArrival[
    indexProcess], ProcessInProcess));
        }
        if (currentTime == sumTime) ss << currentTime;
    }
    double AVG_WT = 0, AVG_TT = 0;
    for (int i = 0; i < (int)p.size(); i++) {
        AVG_WT += WT[i];
        AVG_TT += TT[i];
    }
    AVG_WT /= p.size();
    AVG_TT /= p.size();
    string chart = ss.str();
    ofstream out;
    out.open("Priority (Preemptive).txt", ios::out);
    out << "Scheduling chart: " << endl;
    out << "\t" << chart << endl;
    for (int i = 0; i < (int)p.size(); i++) {
        out << processName[i] << ":\t" << "TT = " << TT[i] << " WT = " << WT[i] << endl;
    }
    out << "Average:  TT = " << AVG_TT << " WT = " << AVG_WT;
    out.close();
}
```

4.6 Priority Scheduling (Nonpreemptive version)

- Là thuật toán điều phối độc quyền dựa trên độ ưu tiên của từng tiến trình.
- Ý tưởng thuật toán và triển khai thuật toán giống tương tự với SJF, chỉ là hàng đợi của các tiến trình sẽ sắp xếp theo độ ưu tiên chứ không sắp xếp theo CPU Burst.
- Mã nguồn của thuật toán

```
void NonpreemptivePriority(vector<Process> p, int q) {
    ofstream out("Priority (Nonpreemptive).txt");
    int totalTime = 0, totalTT = 0, totalWT = 0, count = p.size();
    sort(p.begin(), p.end(), [](const Process& a, const Process& b) { return a.
    arrivalTime < b.arrivalTime; });
    vector<Process> temp;
    while (p.size()) {
        if (p[0].arrivalTime == 0) {
            temp.push_back(p[0]);
            p.erase(p.begin());
        }
        else break;
    }
    map<string, int> TT, WT;
    out << "Scheduling chart: 0";
    while (p.size() || temp.size()) {
        if (temp.size() == 0) {
            out << " - " << p[0].arrivalTime;
            int push = p[0].arrivalTime;
```

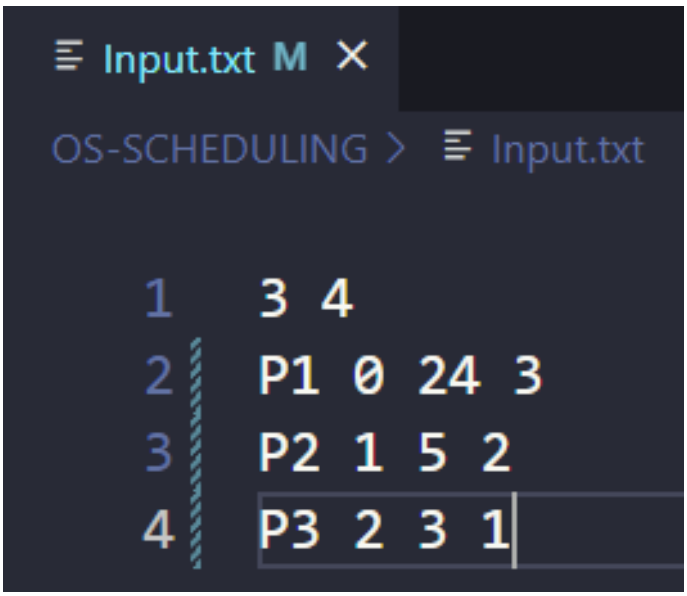
```
            totalTime = p[0].arrivalTime;
            while (p.size()) {
                if (p[0].arrivalTime == push) {
                    temp.push_back(p[0]);
                    p.erase(p.begin());
                }
                else break;
            }
        }
        sort(temp.begin(), temp.end(), [](const Process& a, const Process& b) { return a.priority < b.priority; });
        totalTime += temp[0].burst;
        TT.insert({ temp[0].name, totalTime - temp[0].arrivalTime });
        WT.insert({ temp[0].name, totalTime - temp[0].burst - temp[0].arrivalTime });
        out << " ~" << temp[0].name << "~ " << totalTime;
        temp.erase(temp.begin());
        for (int i = 0; i < p.size(); i++) {
            if (totalTime >= p[i].arrivalTime) {
                temp.push_back(p[i]);
                p.erase(p.begin() + i);
                i--;
            }
        }
    }
    out << endl;
    for (auto x : TT) {
        totalTT += x.second;
        totalWT += WT[x.first];
        out << x.first << ":\t\tTT = " << x.second << "\tWT = " << WT[x.first] << endl;
    }
    out << "Average:\t\t\tTT = " << 1.0 * totalTT / count << "\t\tWT = " << 1.0 * totalWT / count;
    out.close();
}
```

4.7 Kết quả sau khi chạy chương trình

- Với thông tin của từng tiến trình được thể hiện như bảng dưới đây (và thời gian Quantum = 4):

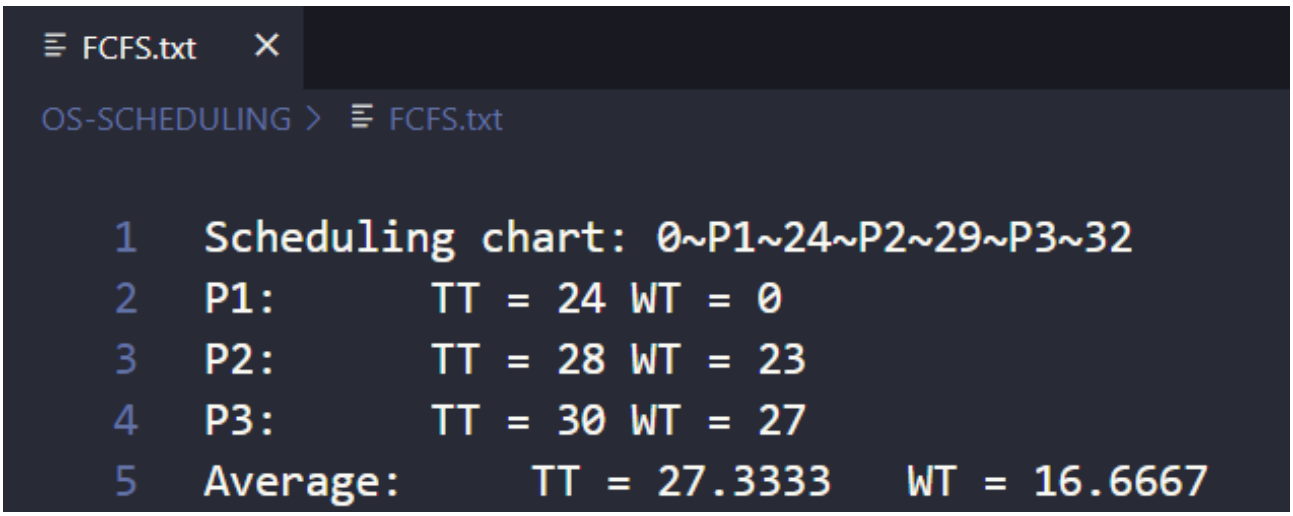
Process	Arrival Time	CPU Burst	Priority
P1	0	24	3
P2	1	5	2
P3	2	3	1

Thì ta có nội dung của tập tin đầu vào của chương trình như sau:



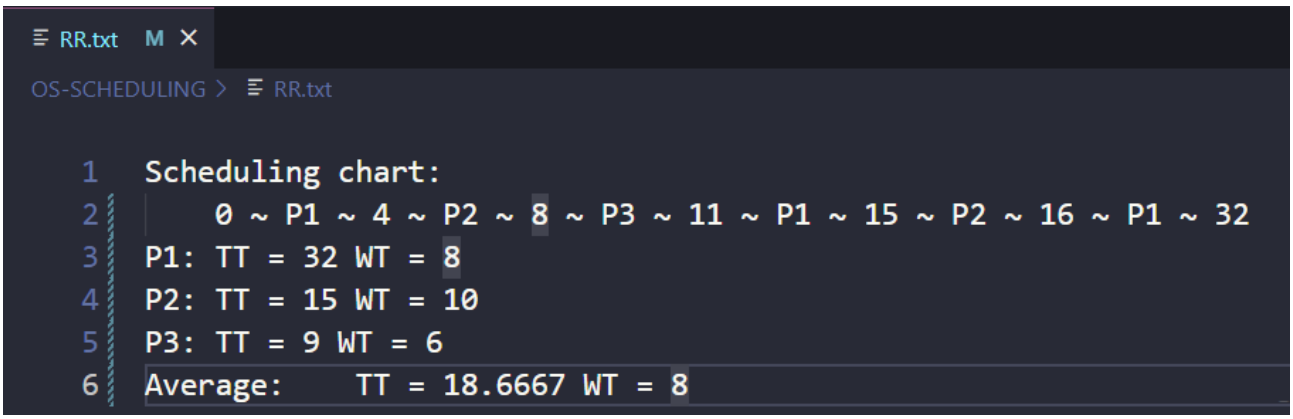
Hình 1: Nội dung tập tin *Input.txt*

- Kết quả của file *FCFS.txt* khi chạy thuật toán điều phối FCFS.



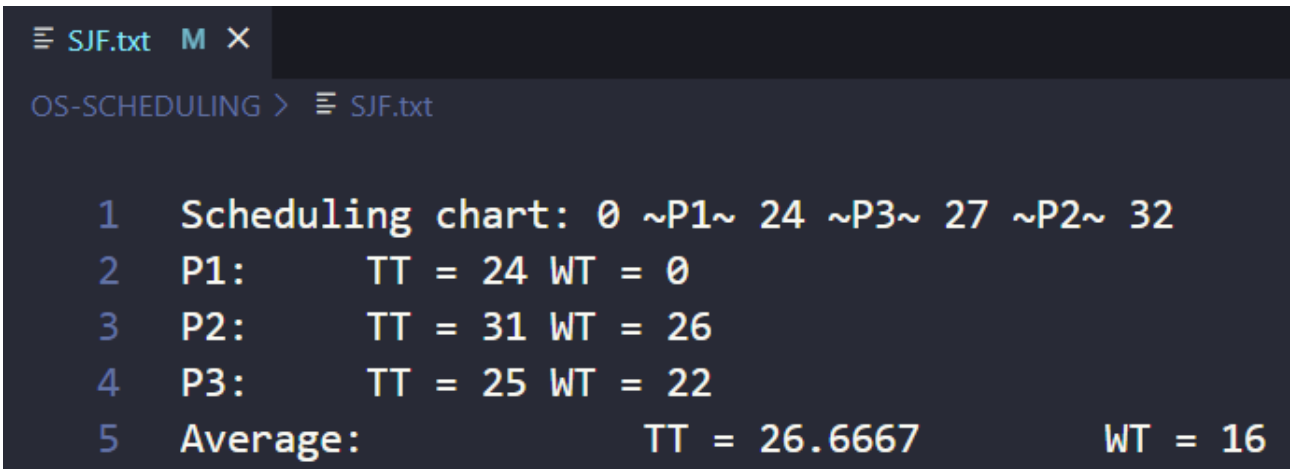
Hình 2: Nội dung tập tin *FCFS.txt*

- Kết quả của file *RR.txt* khi chạy thuật toán điều phối Round−Robin.



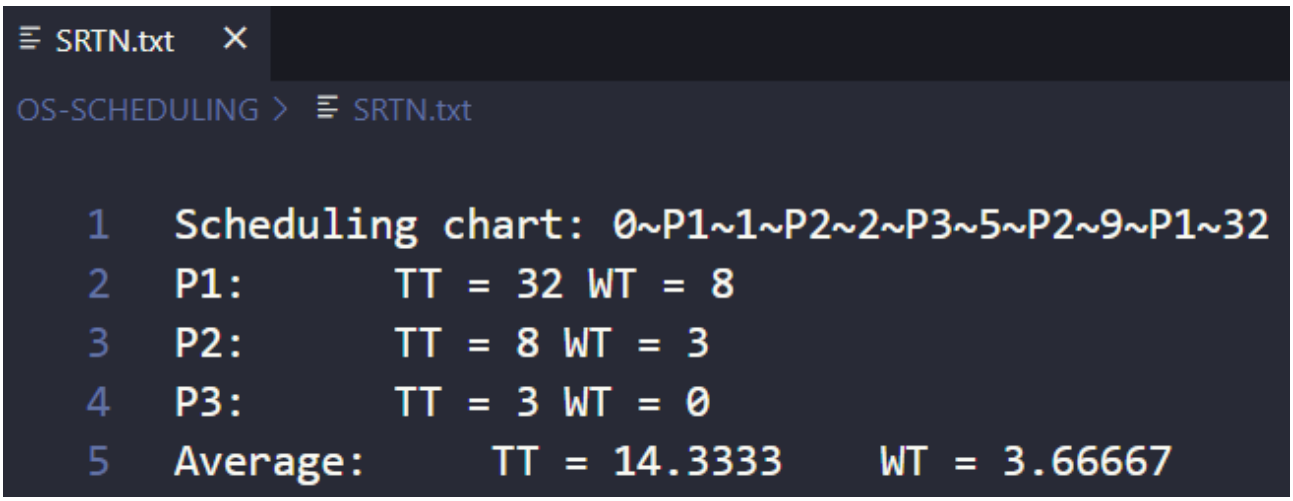
Hình 3: Nội dung tập tin *RR.txt*

- Kết quả của file *SJF.txt*



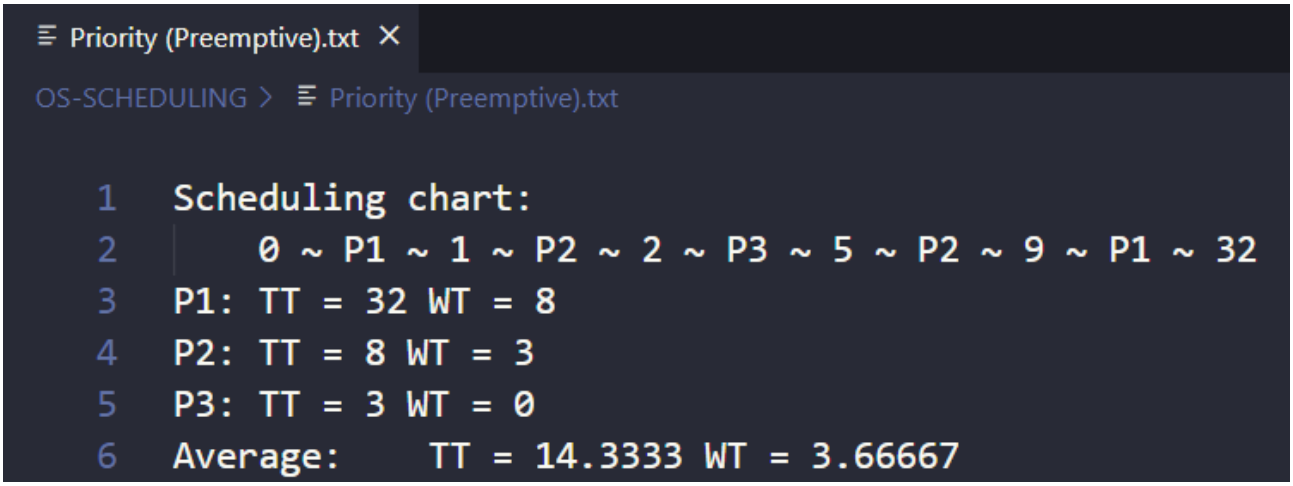
Hình 4: Nội dung tập tin *SJF.txt*

- Kết quả của file *SRTN.txt*



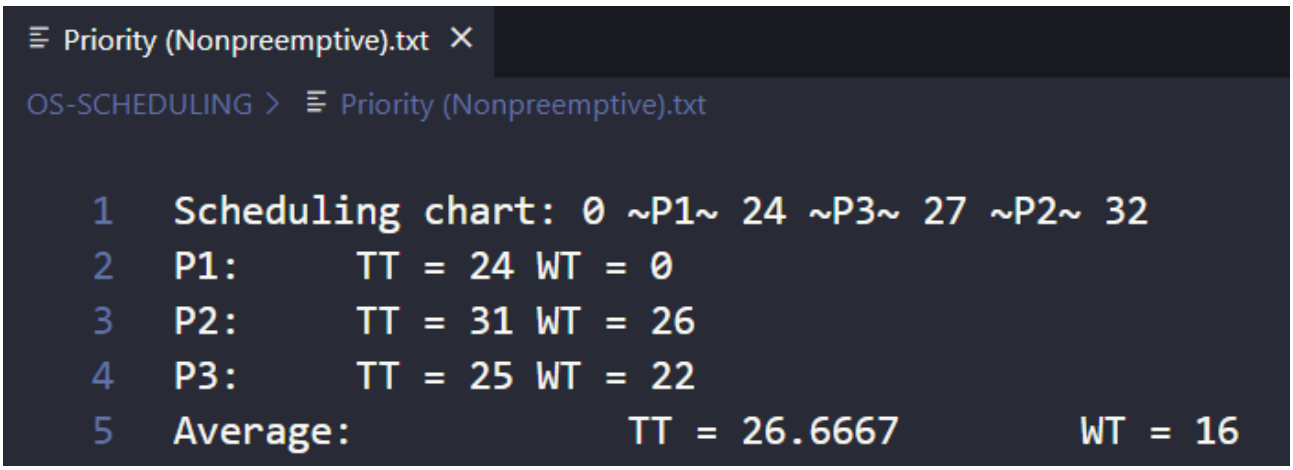
Hình 5: Nội dung tập tin *SRTN.txt*

- Kết quả của file *Prior (Preemptive).txt*



Hình 6: Nội dung tập tin *Prior (Preemptive).txt*

- Kết quả của file *Prior (Nonpreemptive).txt*



Hình 7: Nội dung tập tin *Prior (Nonpreemptive).txt*