CHƯƠNG TRÌNH MINH HỌA THUẬT TOÁN FIRST COME FIRST SERVE VÀ ELEVATOR ALGORITHM

OPERATING SYSTEMS(OSG202)

EDITED BY

NGUYỄN THIỆN PHÚC [HE186083]
NGUYỄN ĐỨC THẮNG [HE186090]

FPT University
Hoa Lac, Ha Noi, Viet Nam

JULY, 2023

Mục lục

Chương 1

Giới thiệu

1.1 Giới thiệu về nhóm

Nhóm chúng tôi gồm có 2 sinh viên đều thuộc chuyên ngành Software Engineer là Nguyễn Thiện Phúc và Nguyễn Đức Thắng. Nhóm được lập ra trên cơ sở hợp tác để minh họa lại 2 thuật toán FIRST COME FIRST SERVE và ELEVATOR nhằm phục vụ cho môn học OSG202 (Operations System) được chỉ dạy bởi thầy Ngô Hải Anh.

Báo cáo này tóm tắt những nội dung về phần thuyết trình trong môn học OSG202. OSG20 là một môn học được học trong học kỳ 2 của chuyên ngành SE. Trong báo cáo này, chúng e, trình bày các khái niệm, trình bày 2 thuật toán và phần chương trình.

1.2 Giới thiệu về thuật toán

1.2.1 Disk Scheduling Algorithm

Disk Scheduling được thực hiện bởi hệ điều hành để lên lịch các yêu cầu I/O đến đĩa. **Disk Scheduling** còn được gọi là **I/O Scheduling**. Disk Scheduling rất quan trọng vì:

- Nhiều yêu cầu I/O có thể đến bởi các quy trình khác nhau và chỉ một yêu cầu I/O có thể được bộ điều khiển đĩa phục vụ tại một thời điểm. Do đó, các yêu cầu I/O khác cần đợi trong hàng đợi và cần được lên lịch.
- Hai hoặc nhiều yêu cầu có thể cách xa nhau, do đó có thể dẫn đến

chuyển động của nhánh đĩa lớn hơn.

- Ổ đĩa cứng là một trong những phần chậm nhất của hệ thống máy tính và do đó cần phải được truy cập một cách hiệu quả.

Có nhiều thuật toán lập lịch đĩa nhưng trước khi thảo luận về chúng, chúng ta hãy xem nhanh một số thuật ngữ quan trọng:

Seek Time (Thời gian tìm kiếm) : Thời gian tìm kiếm là thời gian cần thiết để xác định vị trí của nhánh đĩa đến một rãnh cụ thể nơi dữ liệu sẽ được đọc hoặc ghi. Vì vậy, thuật toán lập lịch đĩa cung cấp thời gian tìm kiếm trung bình tối thiểu sẽ tốt hơn.

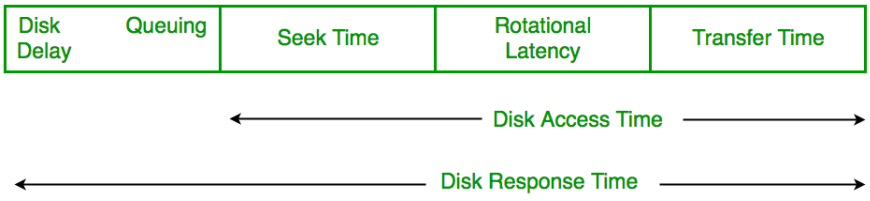
Rotation Latency (Độ trễ quay): Độ trễ quay là thời gian mà khu vực mong muốn của đĩa cần để xoay vào một vị trí sao cho nó có thể truy cập vào các đầu đọc/ghi. Vì vậy, thuật toán lập lịch đĩa cung cấp độ trễ quay tối thiểu sẽ tốt hơn.

Transfer Time (Thời gian truyền): Thời gian truyền là thời gian truyền dữ liệu. Nó phụ thuộc vào tốc độ quay của đĩa và số byte được truyền.

Disk Access Time (Thời gian truy cập đĩa): Thời gian truy cập đĩa là:

```
Disk Access Time = Seek Time +
                   Rotational Latency +
                   Transfer Time


Total Seek Time = Total head Movement * Seek Time
```

Disk Response Time (Thời gian phản hồi của đĩa): Thời gian phản hồi là thời gian trung bình mà một yêu cầu sử dụng để chờ thực hiện thao tác I/O của nó. Thời gian phản hồi trung bình là thời gian phản hồi của tất cả các yêu cầu. Thời gian phản hồi phương sai là thước đo cách yêu cầu riêng lẻ được phục vụ đối với thời gian phản hồi trung bình. Vì vậy, thuật toán lập lịch đĩa cung cấp thời gian phản hồi phương sai tối thiểu sẽ tốt hơn.



1.2.2 First Come First Serve (FCFS) Algorithm

FCFS là viết tắt của **First Come First Serve**. Trong thuật toán FCFS, công việc đến đầu tiên trong hàng đợi sẵn sàng được phân bổ cho CPU và sau đó là công việc đến thứ hai, v.v. Có thể nói rằng hàng đợi sẵn sàng hoạt động như một hàng đợi **FIFO** (Vào trước ra trước), do đó các công việc/quy trình đến được đặt ở cuối hàng đợi.

FCFS là một thuật toán lập lịch không ưu tiên vì một quá trình giữ CPU cho đến khi nó kết thúc hoặc thực hiện I/O. Như vậy, nếu một công việc dài hơn đã được gán cho CPU thì nhiều công việc ngắn hơn sau nó sẽ phải đợi. Thuật toán này được sử dụng trong hầu hết các hệ điều hành hàng loạt.

FCFS là thuật toán **Disk Scheduling** đơn giản nhất. Như tên gợi ý, thuật toán này xử lý các yêu cầu theo thứ tự chúng đến trong hàng đợi đĩa. Thuật toán trông rất công bằng và không có tình trạng chết (tất cả các yêu cầu đều được phục vụ tuần tự) nhưng nhìn chung, nó không cung cấp dịch vụ nhanh nhất.

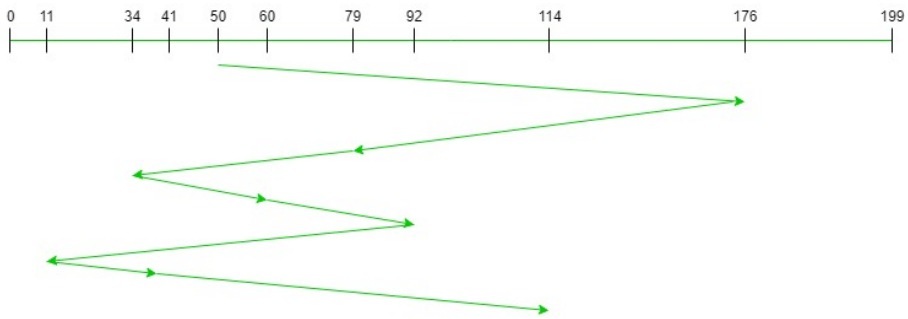
Thuật toán:

1. Đặt mảng **Request** là một mảng lưu trữ giá trị của tracks đã được yêu cầu theo thứ tự tăng dần về thời gian đến của chúng. '**head**' là vị trí của đầu đĩa.
2. Chúng ta hãy lần lượt theo dõi các **tracks** theo thứ tự mặc định và tính khoảng cách tuyệt đối của **tracks** từ **head**.
3. Tăng tổng số lần tìm kiếm với khoảng cách này.
4. Vị trí **track** hiện đang được bảo dưỡng giờ trở thành vị trí **head** mới.
5. Quay lại bước 2 cho đến khi tất cả các **tracks** trong mảng **Requests** được thực thi hết.

Ví dụ:

```
Input:
Request sequence = 176, 79, 34, 60, 92, 11, 41, 114
Initial head position = 50

Output:
Total number of seek operations = 510
Seek Sequence is 176 79 34 60 92 11 41 114
```

Biểu đồ sau đây cho thấy trình tự các **requested tracks** được tính bằng FCFS.



1.2.3 Elevator Algorithm

Thuật toán Elevator (Scan):

Trong thuật toán **Elevator Disk Scheduling**, đầu bắt đầu từ một đầu của disk và di chuyển về phía đầu kia, thực hiện các yêu cầu ở giữa từng cái một và đến đầu kia. Sau đó, hướng của **head** được đảo ngược và quá trình tiếp tục khi **head** liên tục quét qua lại để truy cập disk. Vì vậy, thuật toán này hoạt động như một thang máy và do đó được gọi là **Elevator Algoithm**. Do đó, các yêu cầu ở dải trung được phục vụ nhiều hơn và những yêu cầu đến sau nhánh đĩa sẽ phải đợi.

Thuật toán:

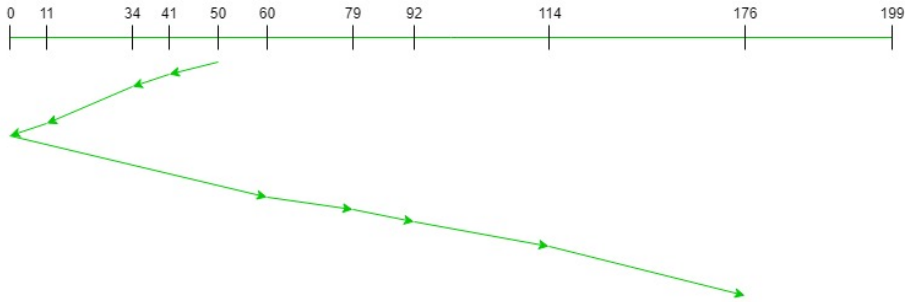
1. Đặt mảng **Request** là một mảng lưu trữ giá trị của các tracks đã được yêu cầu theo thứ tự tăng dần về thời gian đến của chúng. '**head**' là vị trí của đầu đĩa.
2. Đặt **direction** đại diện cho việc **head** đang di chuyển về phía trái hay phải.
3. Theo hướng mà **head** đang di chuyển, tính từng **track** một.
4. Tính khoảng cách tuyệt đối của **track** từ **head**.
5. Tăng tổng số lần tìm kiếm với khoảng cách này.
6. Vị trí **track** hiện tại bây giờ trở thành vị trí **head** mới.
7. Quay lại bước 3 cho đến khi chúng ta đến một trong các đầu của đĩa.
8. Nếu chúng ta đến cuối đĩa, hãy đảo ngược hướng và quay lại bước 2 cho đến khi tất cả các **track** trong mảng **Request** được thực thi hết.

Ví dụ:

Input:
Request sequence = 176, 79, 34, 60, 92, 11, 41, 114
Initial head position = 50
Direction = left (Chúng ta di chuyển từ phải sang trái)

Output:
Total number of seek operations = 226
Seek Sequence is 41 34 11 0 60 79 92 114 176

Biểu đồ sau đây cho thấy trình tự các **track** được thực hiện bằng **Elevator Algorithm**.



Chương 2

Chương trình

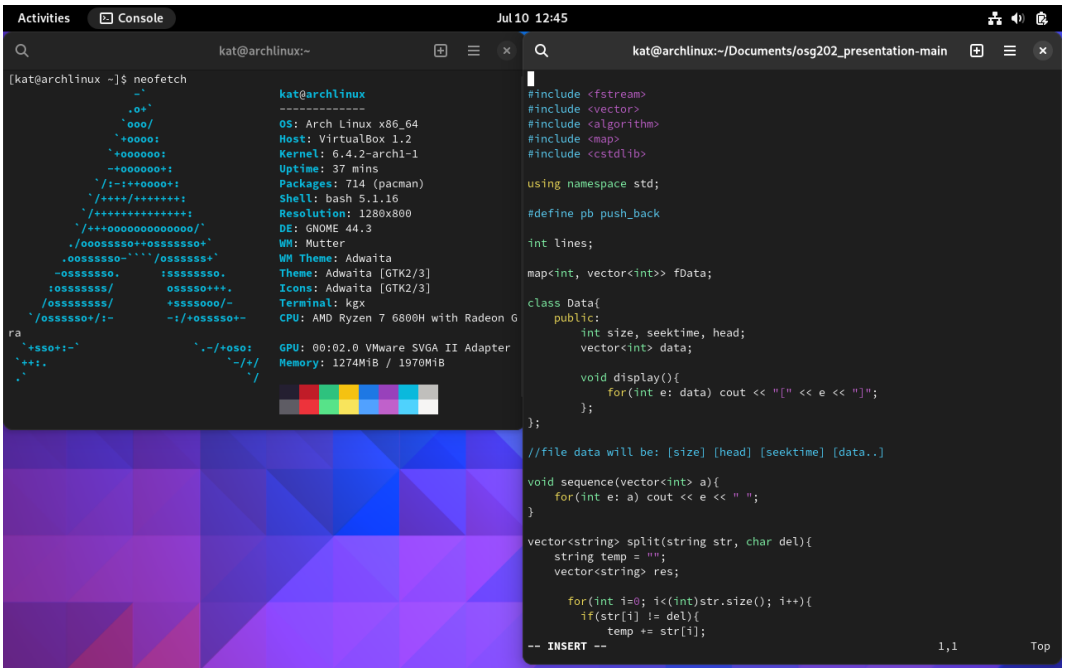
2.1 Cài đặt và chạy

2.1.1 Hệ điều hành

Chương trình được cài đặt trên hệ điều hành Linux.

Bởi vì hệ điều hành của thầy bọn em không có password của user nên không thể update được g++ compiler. Do đó, chúng em đã cài đặt hệ điều hành **ArchLinux** để có thể có được môi trường tốt để code và chạy chương trình này

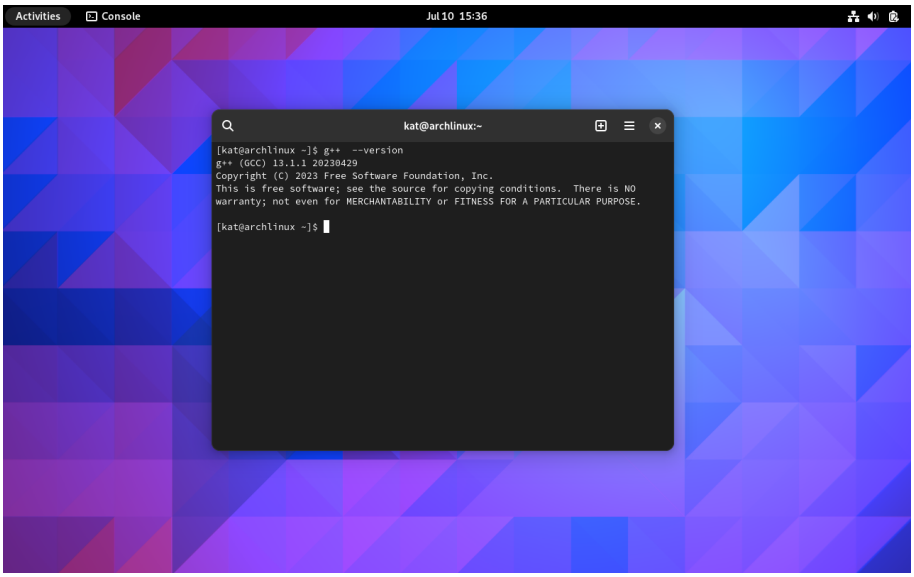
Bọn em đã cài đặt trên: ArchLinux Download và dựa theo clip How To Install Arch Linux in VirtualBox (2023) | Arch Linux Installation để có thể cài đặt **ArchLinux** một cách ổn nhất có thể.



2.1.2 Compiler

Chương trình được compile bằng g++ compiler Chúng em cài đặt GCC Compiler theo bài viết: How to Install GCC for Arch Linux.

G++ Version của bọn em đang sử dụng là: g++ (GCC) 13.1.1



Để code được chương trình thì bọn em còn sử dụng thêm Vim (một ứng dụng biên soạn văn bản). Chương trình này đã được cài đặt sẵn khi cài đặt ArchLinux nên bọn em không cần cài thêm.

Nếu không có Vim trong máy thì có thể cài đặt theo đường dẫn sau: How to Install Latest Vim Editor in Linux

2.1.3 Cài đặt chương trình

Chương trình có thể tải về từ repo: Chương trình mô phỏng 2 thuật toán FCFS và Elevator. Sau khi tải thì bước tiếp theo là giải nén và chạy file program.

2.2 Xử lý dữ liệu

Dữ liệu của chương trình sẽ gồm:

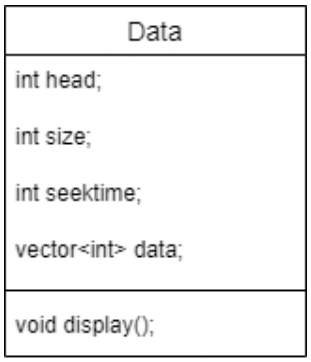
```
int lines;

class Data{
    public:
        int size, seektime, head;
        vector<int> data;

        void display(){
            for(int e: data) cout << "[" << e << "]";
        };
};
```

```
12
```

Listing 2.1: Data in program

Về phần data:

- lines: số dòng trong file input.txt
- Class Data:



  - head: giá trị của cylinder đầu tiên
  - size: số lượng cylinders trong data
  - seektime: thời gian 1 lần di chuyển cylinder (msec)
  - data: dãy vector<int> chứa giá trị của các cylinders
  - display(): Hàm đưa ra console giá trị của tất cả các cylinders trong data

Dữ liệu đầu vào và đầu ra sẽ được phân ra làm 2 phần:

- Thông qua console
- Thông qua file

2.2.1 Qua console

Người dùng sẽ nhập qua console những data cần thiết:

- head
- size
- seektime
- data

Dữ liệu đầu ra sẽ được xuất ra console theo format:

Console Output

Data: [data[0]] [data[1]] .. [data[n]]
FCFS seek time: kết quả
Seek sequence: Thứ tự đọc các cylinders
Elevator seek time: kết quả
Seek sequence: Thứ tự đọc các cylinders

2.2.2 Qua file I/O

Người dùng sẽ nhập các input vào trong file "input.txt"và chương trình sẽ chạy dựa theo các input đã được nhập trong "input.txt".

Các case trong file "input.txt"phải được nhập trên 1 dòng và theo format sau:

File Input Format

[size] [head] [seektime] [data]

Ví dụ:

input.txt

3 86 17 15 93 35
6 92 9 21 62 27 90 59 63
6 40 6 72 36 11 68 67 29
2 30 2 23 67

Sau khi lấy được dữ liệu từ "input.txt", chương trình sẽ tự động thực hiện tính toán cả 2 thuật toán FCFS và Elevator.

Sau khi tính toán xong kết quả sẽ tự động được lưu vào trong file "output.txt"theo format:

File Output Format

i:
FCFS: res, Seek sequence: [..]
Elevator: res, Seek sequence: [..]

Ví dụ:

output.txt

```
1:
FCFS: 3519ms, Seek sequence: [15] [93] [35]
Elevator: 3519ms, Seek sequence: [93] [15] [35]
2:
FCFS: 2205ms, Seek sequence: [21] [62] [27] [90] [59] [63]
Elevator: 2205, Seek sequence: [21] [27] [59] [62] [63] [90]
3:
FCFS: 1134ms, Seek sequence: [72] [36] [11] [68] [67] [29]
Elevator: 1134, Seek sequence: [67] [68] [72] [11] [29] [36]
4:
FCFS: 102ms, Seek sequence: [23] [67]
Elevator: 102ms, Seek sequence: [67] [23]
```

2.3 Chức năng chính

Chương trình mô phỏng 2 thuật toán sẽ có các chức năng:

- File Input.
- Console Input.
- Generate random cases.
- Exit

Người dùng sẽ chọn giữa 4 options bằng cách nhập input từ bàn phím.

2.3.1 Console Input

Option **Console Input**: Người dùng sẽ phải nhập input từ console và chương trình sẽ chạy để tính toán 2 thuật toán và in kết quả ra console.

```
void cCase(){
    Data uData;
    vector<int> seek_sequence;
    int fcfs, elevator;

    userInput(uData);

    cout << "\nData: "; uData.display();
    fcfs = FCFS(uData.head, uData, seek_sequence);
    cout << "FCFS seek time: " << fcfs << endl;
    cout << "Seek sequence: "; sequence(seek_sequence);
```

```
    cout << endl;
    seek_sequence.clear();

    cout << "\nData: "; uData.display();
    elevator = Elevator(uData.head, uData, seek_sequence);
    cout << "Elevator seek time: " << elevator << endl;
    cout << "Seek sequence: "; sequence(seek_sequence);
    cout << endl;
    seek_sequence.clear();
}

```

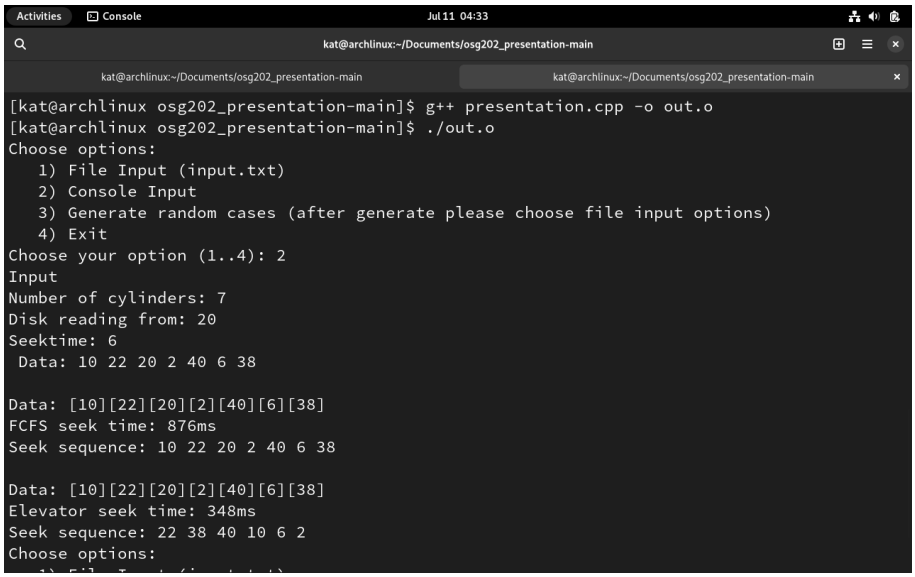
Listing 2.2: Console Input Function

Để có thể lưu trữ dữ liệu được nhập thì em đã viết function **User Input**:

```
void userInput(Data &data){
    cout << "Input\n";

    cout << "Number of cylinders: ";
    cin >> data.size;

    cout << "Disk reading from: ";
    cin >> data.head;

    cout << "Seektime: ";
    cin >> data.seektime;

    cout << " Data: ";
    for(int i = 0; i < data.size; ++i) {
        int tmp;
        cin >> tmp;
        data.data.pb(tmp);
    }
}

```

Listing 2.3: "User Input from Console"

Chương trình chạy thực tế:



2.3.2 File Input

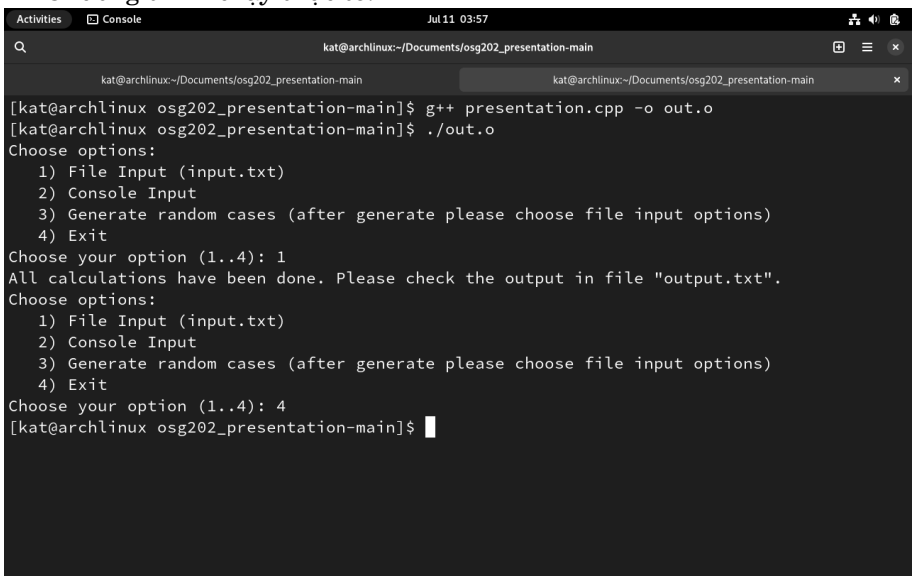
Option **File Input**: Người dùng sẽ không cần phải nhập input từ console mà các cases sẽ được tự động nhập từ file "input.txt".

```
void fCase(){
    map<int, Data> fData;
    vector<int> fcfs, elevator;
    vector<vector<int>> fcfs_se, elevator_se;


    fileInput("./input.txt", fData);

    for(int i = 0; i < lines; ++i){
        vector<int> fcfs_sequence, elevator_sequence;

        fcfs.pb(FCFS(fData[i].head, fData[i], fcfs_sequence));
        elevator.pb(Elevator(fData[i].head, fData[i],
    elevator_sequence));

        fcfs_se.pb(fcfs_sequence);
        elevator_se.pb(elevator_sequence);
    }

    cout << "All calculations have been done. Please check the
    output in file \"output.txt\"." << endl;

    fileOutput("./output.txt", fcfs, elevator, fcfs_se, elevator_se
    );
}
```

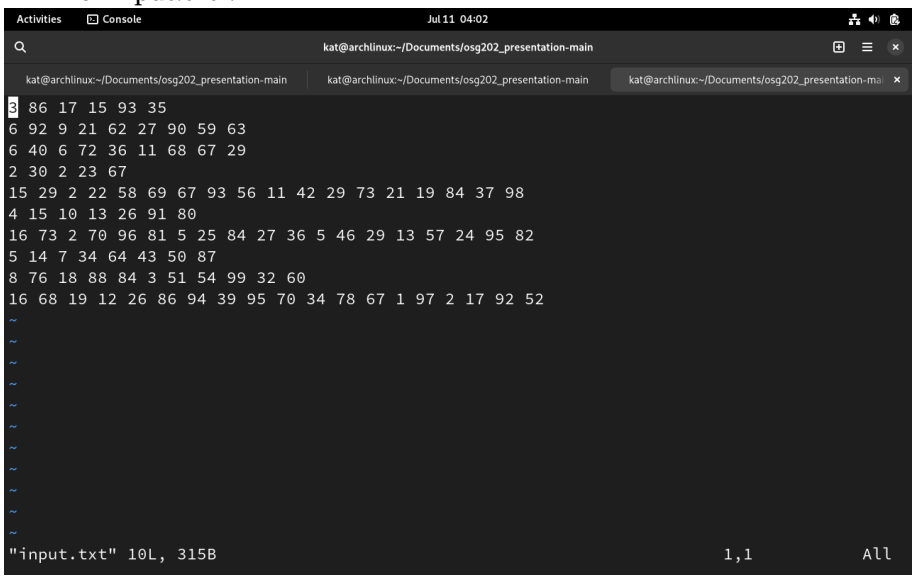
```
23
```

Listing 2.4: File Input Function

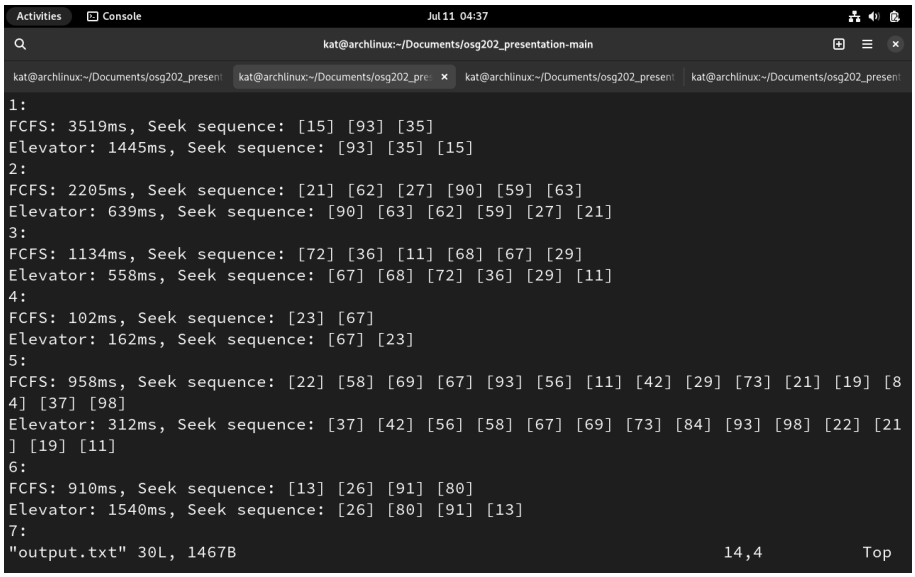
Chương trình chạy thực tế:



File "input.txt":



File "output.txt"sau khi chạy function:



2.3.3 Generate random cases

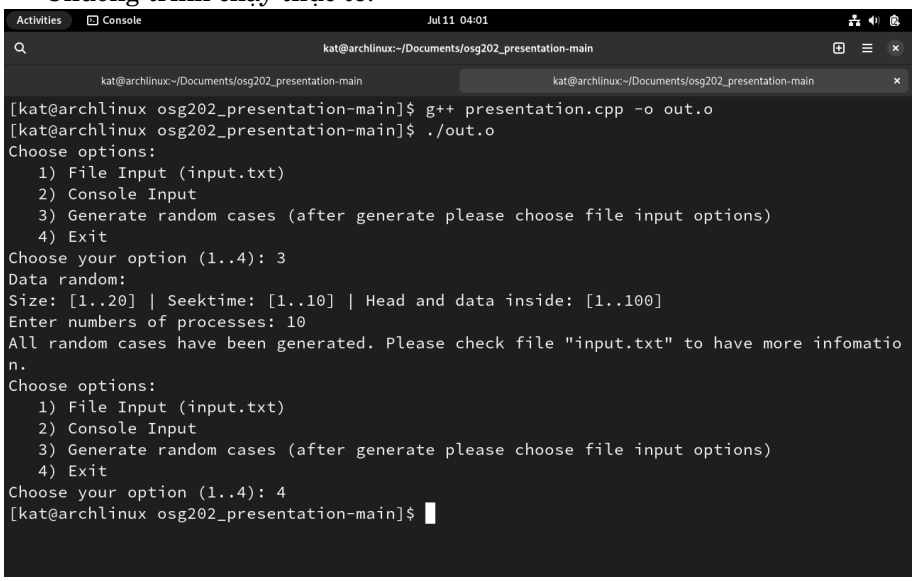
Option **Generate random cases**: Người dùng sẽ phải nhập input từ console và chương trình sẽ chạy để tính toán 2 thuật toán và in kết quả ra console.

```
void rCase(){
    int N = 100;
    int s = 20;
    int ss = 10;
    int n;
    vector<string> lines;
    cout << "Data random: " << endl;
    cout << "Size: [1..20] | Seektime: [1..10] | Head and data
    inside: [1..100]" << endl;;
    cout << "Enter numbers of processes: ";
    cin >> n;

    for(int i = 0; i < n; ++i){
        Data rData;
  int tmp;

  random(rData.size, s);
  random(rData.head, N);
  random(rData.seektime, s);

        for(int j = 0; j < rData.size; ++j) rData.data.pb(random(
    tmp, N));

  string line = to_string(rData.size) + " " + to_string(rData.head)
     + " " + to_string(rData.seektime);
    for(int e : rData.data){
```

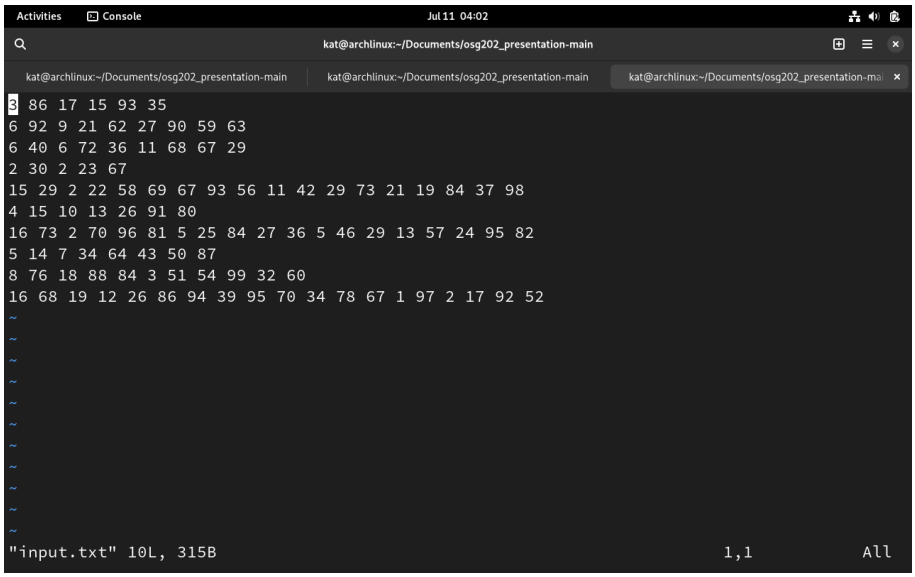
```
            line += " " + to_string(e);
        }
        line += "\n";

    lines.pb(line);
      }

      dataRandomOutput("./input.txt", lines);

      cout << "All random cases have been generated. Please check
      file \"input.txt\" to have more infomation." << endl;

}

```

Listing 2.5: Generate random cases Function

Chương trình chạy thực tế:



File "input.txt"sau khi chạy function:



Để có thể tạo 1 cases random em đã viết thêm 1 funtion **random()**:

```
int random(int &x, int max){
  x = rand() % max;
  while(x == 0) x = rand() % max;
  return x;
}

```

Listing 2.6: Random Function

2.4 Các function

Để có thể tính toán và có thể chạy được thì chúng em đã sử dụng các thư viện:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <map>
#include <cstdlib>
#include <string>

```

Listing 2.7: Libraries

Về các function để tính toán những dữ liệu:

- FCFS()

- Elevator()

Trong đó dữ liệu được đưa vào các function:

- head: giá trị của cylinder đầu tiên
- data: dữ liệu của data được đưa vào (tức case được đưa vào để sử dụng **FCFS**)
- seek_sequence: thứ tự đọc các cylinders

Và function sẽ trả về dữ liệu tốc độ truy xuất dữ liệu theo kiểu int.

2.4.1 FCFS

Function này dùng để tính toán dữ liệu về thuật toán First Come First Serve.

```
int FCFS(int head, Data data, vector<int>& seek_sequence){
    int res = 0;

    for(int i : data.data){
  seek_sequence.pb(i);
        res += abs(head - i);
        head = i;
    }

    return res *= data.seektime;
}

```

Listing 2.8: FCFS implementation

2.4.2 Elevator

Function này dùng để tính toán dữ liệu về thuật toán Elevator. Ở đây chúng em xin được tính theo chiều từ trái sang phải (**direction** = left).

```
int Elevator(int head, Data data, vector<int>& seek_sequence){
    int res = 0;
    int distance;
    vector<int> left, right;


    for(int o : data.data){
        if(o > head) right.pb(o);
        if(o < head) left.pb(o);
    }

    sort(left.begin(), left.end());
```

```
    sort(right.begin(), right.end());

    for(int o : right){
        seek_sequence.pb(o);
        res += abs(head - o);
        head = o;
    }

    for(int i = left.size() - 1; i >= 0; i--){
        int o = left[i];
  seek_sequence.pb(o);
        res += abs(head - o);
        head = o;
    }

    return res *= data.seektime;
}


```

Listing 2.9: Elevator Implementation

2.4.3 Đọc/ghi file

Để có thể sử dụng dữ liệu từ file chúng em sử dụng thư viện **<fstream>**.

Đọc file

```

void fileInput(string path, map<int, Data>& fData){
    ifstream input(path);

    string line;
    lines = 0;

    while(!input.eof()){
        Data tmpData;
        vector<string> element;

        input >> tmpData.size;
        input >> tmpData.head;
        input >> tmpData.seektime;

        for(int i = 0; i < tmpData.size; ++i){
            int tmp;
            input >> tmp;
            tmpData.data.pb(tmp);
        }

        fData.insert(pair<int, Data>(lines, tmpData));
        ++lines;
    }
    lines--;
```

```
    input.close();
}


```

Listing 2.10: File Reading

Dữ liệu được nhập vào function:

- path: Đường dẫn đến file input
- fData: Map chứa dữ liệu Data để có thể lưu

Ghi file

```
void fileOutput(string path, vector<int> fcfs, vector<int> elevator
    , vector<vector<int>> fcfs_sequence, vector<vector<int>>
    elevator_sequence){

    fstream output;
    output.open(path, std::ofstream::out | std::ofstream::trunc);

    for(int i = 0; i < lines; ++i){
        string line = "";
        line += to_string(i+1) + ": \n";
        line += "FCFS: " + to_string(fcfs[i]) + "ms, Seek sequence:
     ";
        for(int x : fcfs_sequence[i]) line += "[" + to_string(x) +
    "] ";
        line += "\n";

        line += "Elevator: " + to_string(elevator[i]) + "ms, Seek
    sequence: ";
        for(int x : elevator_sequence[i]) line += "[" + to_string(x
    ) + "] ";
        output << line << endl;
    }

    output.close();
}

```

Listing 2.11: File writing (Output)

Dữ liệu được nhập vào function:

- path: Đường dẫn đến file input
- fcfs: Mảng Vector chứa kết quả từ thuật toán **FCFS**
- elevator: Mảng Vector chứa kết quả từ thuật toán **Elevator**
- fcfs_sequence: Mảng Vector chứa các mảng thứ tự đọc cylinders của các cases của thuật toán **FCFS**

- elevator_sequence: Mảng Vector chứa các mảng thứ tự đọc cylinders của các cases của thuật toán **Elevator**

Lưu dữ liệu random vào file

```
void dataRandomOutput(string path, vector<string> lines){
    fstream output;
    output.open(path, std::ofstream::out | std::ofstream::trunc);

    for(string line : lines) output<<line;

    output.close();
}

```

Listing 2.12: Tạo dữ liệu random

Dữ liệu được nhập vào function:

- path: Đường dẫn đến file input
- lines: Mảng Vector chứa các dữ liệu input được tạo từ thuật toán random

Chương 3

Nhận xét

3.1 First-Come, First Serve

- Hoạt động: FCFS xử lý các yêu cầu I/O theo thứ tự chúng được gửi đến hệ thống. Điều này có nghĩa là yêu cầu nào đến trước sẽ được xử lý trước.
- Ưu điểm:
  . Đơn giản và dễ hiểu.
  . Không gây ra độ trễ cho các yêu cầu đến sau.
- Nhược điểm:
  . Không tối ưu hóa việc di chuyển đầu đọc/ghi đĩa.
  . Có thể gây ra hiện tượng "đói"đĩa (starvation) cho các yêu cầu ở vị trí xa.

3.2 Elevator (SCAN)

- Hoạt động: Elevator di chuyển đầu đọc/ghi đĩa từ một đầu đến đầu kia, xử lý các yêu cầu I/O trên đường đi và không quay lại trước khi đến đầu đĩa. Nếu không có yêu cầu nào trên đường đi, nó sẽ thay đổi hướng di chuyển và tiếp tục quét theo hướng ngược lại.
- Ưu điểm:
  . Tối ưu hóa việc di chuyển đầu đọc/ghi đĩa bằng cách di chuyển liên tục trong một hướng duy nhất.

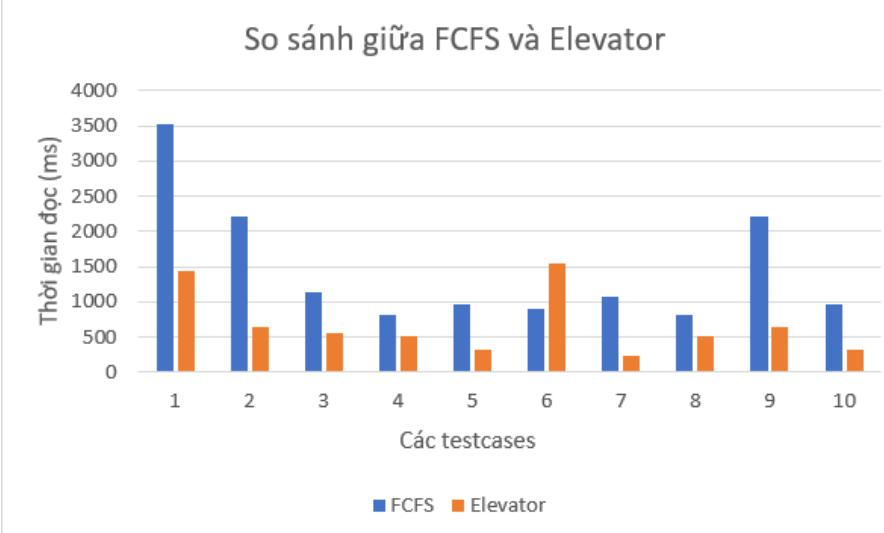
. Tránh hiện tượng "đói"đĩa bằng cách xử lý yêu cầu trên đường đi của đầu đọc/ghi đĩa.

- Nhược điểm:

. Có thể gây ra độ trễ lớn cho các yêu cầu mới đến nếu chúng không nằm trên đường đi của đầu đọc/ghi đĩa.

3.3 So sánh

Cả hai thuật toán tìm kiếm Cylinder đều có ưu điểm và nhược điểm khác nhau.



ĐỒ THỊ THỂ HIỆN THỜI GIAN GIỮA 2 THUẬT TOÁN TRONG 10 TESTCASES KHÁC NHAU

3.3.1 Độ phức tạp thời gian

- **FCFS**: Thuật toán này có độ phức tạp thời gian là O(n), trong đó n là số lượng yêu cầu tìm kiếm cylinder. Vì thuật toán chỉ đơn giản lựa chọn yêu cầu theo thứ tự đến, nên không có bước tính toán phức tạp.
- **Elevator**: Thuật toán Elevator có độ phức tạp thời gian trung bình là O(n), với n là số lượng yêu cầu tìm kiếm cylinder. Tuy nhiên, trong trường hợp xấu nhất, độ phức tạp có thể là O(n^2), nếu có yêu cầu tìm kiếm ở cả hai hướng và phải đi qua toàn bộ đĩa.

3.3.2 Hiệu suất tìm kiếm

- **FCFS**: Thời gian truy cập trong thuật toán FCFS phụ thuộc vào thứ tự các yêu cầu tìm kiếm. Nếu các yêu cầu được gửi đến theo thứ tự tương ứng với vị trí của cylinder trên đĩa cứng, thì thời gian truy cập sẽ là tốt nhất. Tuy nhiên, nếu các yêu cầu không được gửi đến theo thứ tự, thì thời gian truy cập có thể tăng lên do phải di chuyển đầu đĩa cứng qua nhiều vị trí không liên tiếp.
- **Elevator**: Thuật toán Elevator di chuyển theo một hướng duy nhất, do đó thời gian truy cập tương đối nhất định. Nếu các yêu cầu tìm kiếm được gửi đến theo hướng di chuyển hiện tại, thì thời gian truy cập sẽ là tốt nhất. Tuy nhiên, nếu có yêu cầu ở phía ngược lại hướng di chuyển, thì thời gian truy cập sẽ tăng lên do phải di chuyển đầu đĩa qua toàn bộ đĩa cứng.

3.3.3 Thích ứng với tải công việc

- **FCFS**: Thuật toán này đơn giản và không yêu cầu tính toán phức tạp, phù hợp trong các hệ thống có tải công việc nhẹ nhàng, không có quá nhiều yêu cầu đồng thời.
- **Elevator**: Thuật toán Elevator có khả năng xử lý tốt hơn trong các hệ thống có tải công việc nặng, có nhiều yêu cầu đồng thời. Việc di chuyển theo một hướng duy nhất giúp giảm thiểu tắc nghẽn và tăng tốc độ truy cập.

3.3.4 Tính linh hoạt

- **FCFS**: Thuật toán FCFS đơn giản và dễ hiểu, không đòi hỏi tính toán phức tạp. Nó có thể dễ dàng triển khai và áp dụng trong các hệ thống đơn giản.
- **Elevator**: Thuật toán Elevator phức tạp hơn so với FCFS, đòi hỏi tính toán hướng di chuyển và quản lý yêu cầu theo thứ tự. Tuy nhiên, nó cung cấp hiệu suất tốt hơn trong các tải công việc nặng và có khả năng giảm thiểu tắc nghẽn trên đĩa cứng.

3.3.5 Nâng cao

- **Tối ưu hóa di chuyển đầu đọc/ghi đĩa**: Elevator (SCAN) tốt hơn FCFS trong việc tối ưu hóa di chuyển đầu đọc/ghi đĩa bằng cách di chuyển theo một hướng duy nhất và xử lý các yêu cầu trên đường đi. Trong khi đó, FCFS không quan tâm đến vị trí yêu cầu I/O, điều này có thể dẫn đến các di chuyển không hiệu quả và lâu dài trên đĩa.

- **Hiện tượng "đói"đĩa**: Elevator (SCAN) giải quyết hiện tượng "đói"đĩa bằng cách xử lý các yêu cầu trên đường đi của đầu đọc/ghi đĩa, đảm bảo rằng không có yêu cầu nào bị chờ đợi quá lâu. Trong khi đó, FCFS không có cơ chế đảm bảo công bằng giữa các yêu cầu, dẫn đến khả năng một yêu cầu bị chờ đợi quá lâu.

3.4 TỔNG QUAN

Thuật toán FCFS đơn giản và phù hợp cho các tải công việc nhẹ, trong khi thuật toán Elevator có hiệu suất tốt hơn trong các tải công việc nặng. Tuy nhiên, cả hai thuật toán đều có nhược điểm riêng, và lựa chọn phụ thuộc vào bối cảnh sử dụng và yêu cầu cụ thể của hệ thống lưu trữ đĩa cứng.

Tùy thuộc vào bối cảnh và yêu cầu cụ thể, cả hai thuật toán FCFS và Elevator (SCAN) có ưu điểm và nhược điểm riêng. Lựa chọn giữa hai thuật toán này phụ thuộc vào yêu cầu và mục tiêu cụ thể của hệ thống quản lý ổ đĩa.