|  |
| --- |
| Name: Nguyễn Công Phi  ID: 19522006  Class: KTPM2019 |

OPERATING SYSTEM  
LAB 4’S REPORT

**SUMMARY**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task** | | | **Status** | **Page** |
| Section 4.5 | Task 1 | Code và giải thích | Done | 2 |
| Chạy trong ubuntu | Done | 8 |
| Task 2 | | Done | 9 |
| Task 3 | | Done |  |

**Self-scrores: 3/3**

# Section 4.5

## Task 1: Viết chương trình mô phỏng giải thuật SJF với các yêu cầu sau:

## ❖ Nhập số lượng process

## ❖ Nhập process name, arrival time, burst time

## ❖ In ra Process name, response time, waiting time, turnaround time, average waiting time, average turnaround time

* Code của giải thuật (Code được viết bằng ngôn ngữ C++) :
* #include <iostream>
* #include <string>
* using namespace std;
* class Process
* {
* public:
* int pname = 0;
* float paTime = 0.f, pbTime = 0.f;
* float wTime = 0.f, taTime = 0.f;
* float staTime = -1.f, fiTime = 0.f, resTime = 0.f;
* //Function
* void Nhap() {
* cout << "ProcessName, Process Arrival Time, Process Burst Time: ";
* cin >> pname >> paTime >> pbTime;
* }
* void Xuat(){
* printf("%d    %8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f",pname,paTime,pbTime,staTime, taTime,fiTime,wTime,resTime);
* cout<<"\n";
* }
* };
* Class Process chứa các thuộc tính và các phương thức nhập xuất các tiến trình

class Queue

{

public:

    Process\* p = NULL;

    int soLuong = 0;

    Queue(){

    }

    ~Queue(){

        if (soLuong != 0)

            delete p;

    }

    void Push(*Process* *temp*){

        Process\* t = **new** Process[soLuong];

        for (int i = 0; i < soLuong; i++)

        {

            t[i] = p[i];

        }

        soLuong += 1;

        p = **new** Process[soLuong];

        for (int i = 0; i < soLuong - 1; i++)

        {

            p[i] = t[i];

        }

        p[soLuong - 1] = temp;

        delete t;

    }

*Process* PopAtMin(){

        float bTimeMin = p[0].pbTime;

        for (int i = 1; i < soLuong; i++)

        {

            if (p[i].pbTime < bTimeMin)

            {

                bTimeMin = p[i].pbTime;

            }

        }

        Process res;

        int index = 0;

        for (int i = 0; i < soLuong; i++)

        {

            if (p[i].pbTime == bTimeMin)

            {

                res = p[i];

                index = i;

            break;

            }

        }

        Process\* t = **new** Process[soLuong - 1];

        for (int i = 0; i < index; i++)

        {

            t[i] = p[i];

        }

        for (int i = index + 1; i < soLuong; i++)

        {

            t[i - 1] = p[i];

        }

        soLuong -= 1;

        p = **new** Process[soLuong];

        for (int i = 0; i < soLuong; i++)

        {

            p[i] = t[i];

        }

        delete t;

        return res;

        }

    bool isEmpty(){

        if (soLuong == 0) return true;

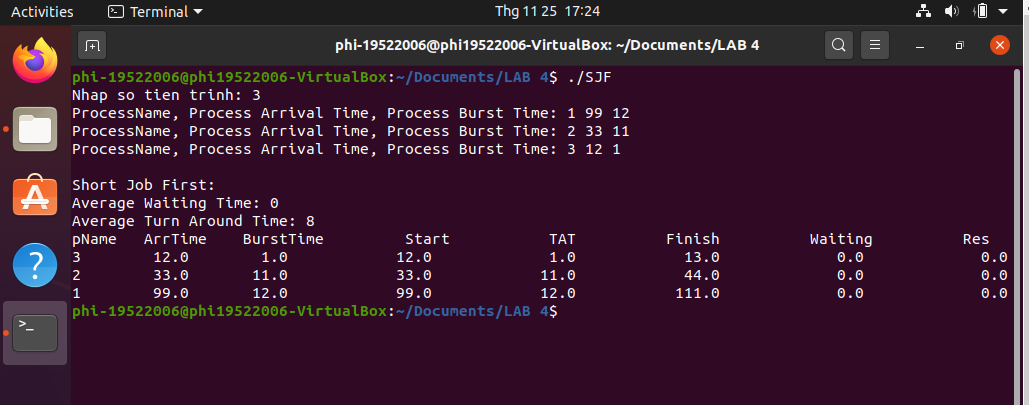
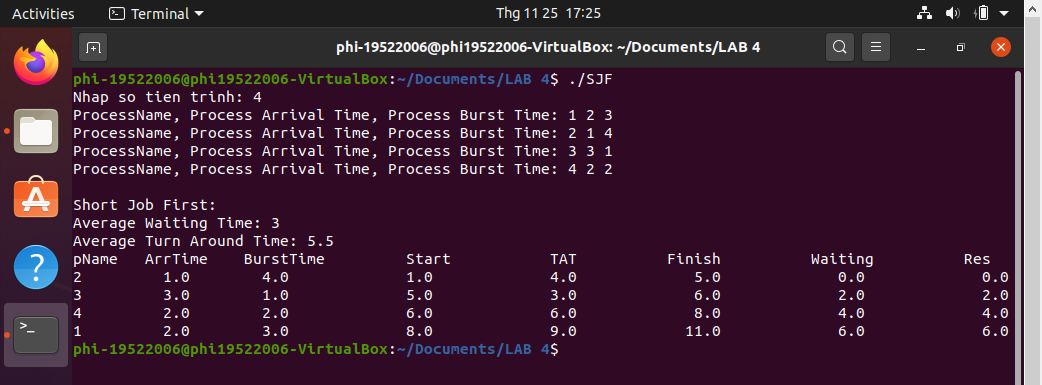
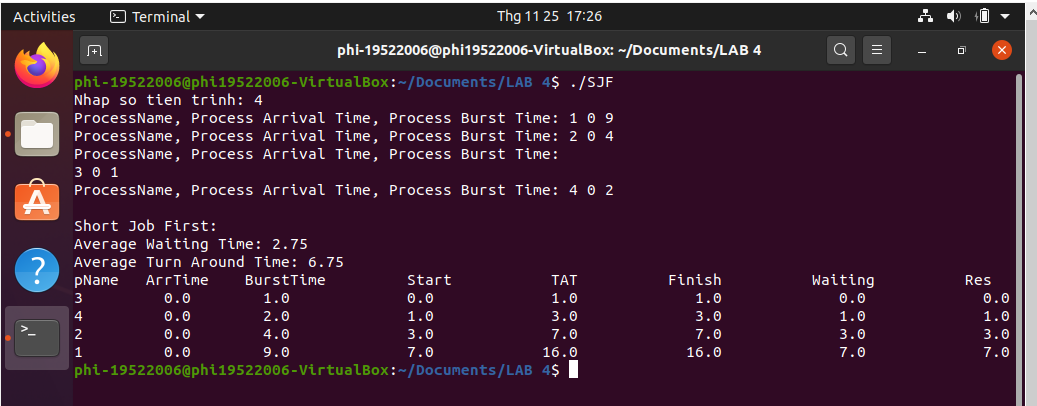
        else return false;

    }

};

* Class Queue là 1 queue chứa các Process với hàm:
  + Push: để thêm process vào Queue
  + PopAtMin: để lấy ra tiến trình có burst nhỏ nhất có trong queue và lấy ra khỏi Queue
  + isEmpty: Một bool để xét xem có process nào trong Queue không
* class Algorithm
* {
* public:
* Process pc[100];
* int soLuong = 0;
* float avgWTime = 0.f, avgTATime = 0.f;
* void Nhap() {
* cout << "Nhap so tien trinh: ";
* int n; cin >> n;
* soLuong = n;
* for (int i = 0; i < n; i++)
* {
* pc[i].Nhap();
* }
* }
* void Xuat() {
* cout << "pName   ArrTime    BurstTime         Start           TAT          Finish          Waiting          Res\n";
* for (int i = 0; i < soLuong; i++)
* {
* pc[i].Xuat();
* }
* }
* void ShortJobFirst(){
* //cout << "pName   ArrTime    BurstTime         Start           TAT          Finish\n";
* Queue pQueue;
* Process\* Ketqua = **new** Process[soLuong];
* //Sắp xếp các tiến trình theo arrival time
* for (int i = 0; i < soLuong - 1; i++)
* {
* for (int j = i + 1; j < soLuong; j++)
* {
* if (pc[i].paTime > pc[j].paTime)
* {
* Process temp = pc[i];
* pc[i] = pc[j];
* pc[j] = temp;
* }
* }
* }
* float time = pc[0].paTime;
* int so = soLuong;
* int index = 0;
* for (int i = 0; i < so; i++)
* {
* for (int j = index; j < soLuong; j++)
* {
* if (pc[j].paTime <= time)
* {
* pQueue.Push(pc[j]);
* }
* else
* {
* index = j;
* break;
* }
* if (j == soLuong - 1)
* {
* index = soLuong;
* }
* }
* Process k;
* if (pQueue.isEmpty()==true)
* {
* i--;
* time = pc[index].paTime;
* for (int j = index; j < soLuong; j++)
* {
* if (pc[j].paTime <= time)
* {
* pQueue.Push(pc[j]);
* }
* else
* {
* index = j;
* break;
* }
* if (j == soLuong - 1)
* {
* index = soLuong;
* }
* }
* continue;
* }
* else k = pQueue.PopAtMin();
* k.staTime = time;
* k.taTime = k.pbTime + time - k.paTime;
* time += k.pbTime;
* k.fiTime = time;
* k.wTime = k.staTime - k.paTime;
* k.resTime=k.staTime-k.paTime;
* Ketqua[i] = k;
* //Ketqua[i].Xuat();
* }
* for (int i = 0; i < soLuong; i++)
* {
* pc[i] = Ketqua[i];
* }
* delete Ketqua;
* //Tính avgWating và avgTAT
* for (int i = 0; i < soLuong; i++)
* {
* avgWTime += pc[i].wTime;
* avgTATime += pc[i].taTime;
* }
* avgWTime /= soLuong;
* avgTATime /= soLuong;
* cout << "\nShort Job First:\nAverage Waiting Time: " << avgWTime << "\nAverage Turn Around Time: " << avgTATime << "\n";
* }
* };
* Class Algorithm:
  + Gồm hai hàm nhập xuất các tiến trình
  + Hàm Short Job First là giải thuật Short Job First:
    - B1: Sắp xếp các tiến trình theo arrival time tăng dần
    - B2: Chọn thời gian hiện tại là thời gian của arrival time đầu tiên
    - B3: Xếp tất cả tiến trình có thời gian bằng với thời gian hiện tại vào Queue và lấy ra tiến trình có thời gian burst time bé nhỏ nhất với hàm PopAtMin đế thực hiện, nếu có nhiều hơn một thì lấy tiến trình nằm trước trong Queue để thực hiện
    - B4: Thực hiện xong tiến trình thì sẽ lấy thời gian hiện tại bằng giá trị lớn hơn thời gian lúc hoàn thành tiến trình và thời gian lúc tiến trình tiếp theo tới
    - B5: Nếu trong hàng đợi còn tiến trình hoặc index nhỏ hơn số tiến trình thì sẽ tiếp tục với bước 3 còn không sẽ in ra danh sách các tiến trình được thực hiện và kết thúc
* int main()
* {
* Algorithm a;
* a.Nhap();
* a.ShortJobFirst();
* a.Xuat();

    return 0;

* }
* Đoạn code chương trình chính:
  + Tạo 1 object algorithm, nhập các process, thực hiện short job first và in ra các thứ tự thực hiện các tiến trình bằng hàm xuất
* Kết quả chạy trên Ubuntu:
  + Testcase 1:
  + 
    - Với 3 tiến trình có thời gian arrival time chênh nhau lớn hơn burst time thì SJF sẽ thực hiện như First Come First Solve
  + Testcase 2:
  + 
    - Với 4 tiến trình trên thì tiến trình 2 với thời gian đến đầu tiên sẽ được thực hiện trước, sau khi thực hiện xong thì đã là giây thức 5, tất cả tiến trình khác đã vào hàng đợi nên nó sẽ thực hiện theo thứ tự tiến trình có burst time nhỏ nhất trước nến thứ tự thực hiện sẽ là 3 4 1
  + Testcase 3:
  + 
    - Các tiến trình có cùng arrival time là 0 nên sẽ được thực thi theo thứ tự tiến trình burst time nhỏ nhất trước, các tiến trình có burst time lớn nhất sau

## Task 2: Viết chương trình mô phỏng giải thuật SRT với các yêu cầu sau:

## ❖ Nhập số lượng process

## ❖ Nhập process name, arrival time, burst time 13

## ❖ In ra Process name, response time, waiting time, turnaround time, average waiting time, average turnaround time

* Code của thuật toán SRT(Được viết bằng C++)
* #include <iostream>
* #include <string>
* using namespace std;
* class Process
* {
* public:
* int pname=0;
* float paTime=0.f, pbTime=0.f;
* float wTime = 0.f, taTime = 0.f;
* float staTime = -1.f, fiTime=0.f;
* float timeLeft = 0.f, resTime = 0.f;// use for RR and SRJF
* //Function
* void Nhap()
* {
* cout << "ProcessName, Process Arrival Time, Process Burst Time: ";
* cin >> pname>> paTime>> pbTime;
* timeLeft = pbTime;
* }
* void Xuat()
* {
* printf("%d    %8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f\t%8.1f",pname,paTime,pbTime,staTime, taTime,fiTime,wTime,timeLeft,resTime);
* cout << "\n";
* }
* };
* Class Process chứa các thuộc tính và các phương thức nhập xuất các tiến trình
* class Queue
* {
* public:
* Process\* p = NULL;
* int soLuong = 0;
* Queue()
* {
* }
* ~Queue()
* {
* if(soLuong!=0)
* delete p;
* }
* void Push(*Process* *temp*)
* {
* Process\* t = **new** Process[soLuong];
* for (int i = 0; i < soLuong; i++)
* {
* t[i] = p[i];
* }
* soLuong += 1;
* p = **new** Process[soLuong];
* for (int i = 0; i < soLuong-1; i++)
* {
* p[i] = t[i];
* }
* p[soLuong - 1] = temp;
* delete t;
* }
* *Process* ViewAtMin()
* {
* float TimeMin = p[0].timeLeft;
* for (int i = 1; i < soLuong; i++)
* {
* if (p[i].timeLeft < TimeMin)
* {
* TimeMin = p[i].pbTime;
* }
* }
* Process res;
* int index = 0;
* for (int i = 0; i < soLuong; i++)
* {
* if (p[i].timeLeft == TimeMin)
* {
* res = p[i];
* index = i;
* break;
* }
* }
* return res;
* }
* *Process* PopAtMin2()
* {
* if (soLuong == 0)
* {
* Process error;
* error.pname = -1;
* return error;
* }
* else
* {
* float TimeMin = p[0].timeLeft;
* for (int i = 1; i < soLuong; i++)
* {
* if (p[i].timeLeft < TimeMin)
* {
* TimeMin = p[i].timeLeft;
* }
* }
* Process res;
* int index = 0;
* for (int i = 0; i < soLuong; i++)
* {
* if (p[i].timeLeft == TimeMin)
* {
* res = p[i];
* index = i;
* break;
* }
* }
* Process\* t = **new** Process[soLuong - 1];
* for (int i = 0; i < index; i++)
* {
* t[i] = p[i];
* }
* for (int i = index + 1; i < soLuong; i++)
* {
* t[i - 1] = p[i];
* }
* soLuong -= 1;
* p = **new** Process[soLuong];
* for (int i = 0; i < soLuong; i++)
* {
* p[i] = t[i];
* }
* delete t;
* return res;
* }
* }
* bool isEmpty()
* {
* if (soLuong == 0) return true;
* else return false;
* }
* void ViewQueue()
* {
* for (int i = 0; i < soLuong; i++)
* {
* printf("%d    %8.1f\t%8.1f\t%8.1f", p[i].pname, p[i].paTime, p[i].pbTime, p[i].staTime);
* cout << "\n";
* }
* }
* *Process* PopAndMerge()
* {
* Process first = p[0];
* int indexoffirst=0;
* for (int i = 1; i < soLuong; i++)
* {
* if (first.pname == p[i].pname)
* {
* first=p[i];
* for (int j = indexoffirst; j < soLuong - 1; j++)
* {
* p[j] = p[j + 1];
* }
* indexoffirst = i - 1;
* soLuong -= 1;
* }
* }
* for (int j = indexoffirst; j < soLuong - 1; j++)
* {
* p[j] = p[j + 1];
* }
* soLuong -= 1;
* return first;
* }
* };
* Class Queue là 1 hàng đợi có chứa các Process và các phương thức
  + Push: Thêm một process vào hàng đợi
  + PopAtMin: Lấy process có timeLeft nhỏ nhất ra khỏi hàng đợi
  + isEmpty: một bool xét thử Queue có đang trống hay không
  + ViewQueue: một void dùng để xem những process đang có trong queue(em dùng nó trong quá trình debug)
  + PopAndMerge: Nhóm tiến trình lại với nhau bởi khi push vào queue thì các tiến trình được push theo khoảng thời gian thực hiện, thời gian dừng thực hiện(có tiến trình khác chiếm chỗ hoặc do tiến trình đã được thực hiện xong)
* class Algorithm
* {
* public:
* Process pc[100];
* int soLuong=0;
* float avgWTime = 0.f, avgTATime = 0.f;
* void Nhap()
* {
* cout << "Nhap so tien trinh: ";
* int n; cin >> n;
* soLuong = n;
* for (int i = 0; i < n; i++)
* {
* pc[i].Nhap();
* }
* }
* void Xuat()
* {
* cout << "pName   ArrTime    BurstTime         Start           TAT          Finish          Waiting          TimeLeft          Res\n";
* for (int i = 0; i < soLuong; i++)
* {
* pc[i].Xuat();
* }
* }
* void ShortRemainingFirst()
* {
* cout << "pName   ArrTime    BurstTime         Start           TAT          Finish          Waiting          TimeLeft          Res\n";
* //cout << "pName   ArrTime    BurstTime         Start           TAT          Finish\n";
* Queue pQueue;

        Queue run;

* //Sắp xếp các tiến trình theo arrival time
* for (int i = 0; i < soLuong - 1; i++)
* {
* for (int j = i + 1; j < soLuong; j++)
* {
* if (pc[i].paTime > pc[j].paTime)
* {
* Process temp = pc[i];
* pc[i] = pc[j];
* pc[j] = temp;
* }
* }
* }
* int index = 0;
* float time = pc[0].paTime;
* Process currentProcess;
* currentProcess.timeLeft=-1;
* //Push những process cùng arrival time vào trong queue
* int j = index;
* while (j + 1 < soLuong && pc[j].paTime == pc[j + 1].paTime)
* {
* j++;
* }
* for (int k = index; k <= j; k++)
* {
* pQueue.Push(pc[k]);
* }
* index = j + 1;
* currentProcess = pQueue.PopAtMin2();
* while(true)
* {
* //tất cả các tiến trình đã vào trong queue
* if (index == soLuong)
* {
* //cout << "Index: " << index << " So luong: " << soLuong<<"\n";
* while (pQueue.isEmpty() != true)
* {
* if (currentProcess.timeLeft == -1)
* currentProcess = pQueue.PopAtMin2();
* if (currentProcess.staTime == -1)
* {
* currentProcess.staTime = time;
* }
* time += currentProcess.timeLeft;
* currentProcess.taTime = time - currentProcess.paTime;
* currentProcess.fiTime = time;
* currentProcess.timeLeft = 0;
* currentProcess.wTime = currentProcess.fiTime - currentProcess.paTime - currentProcess.pbTime;
* run.Push(currentProcess);
* currentProcess.Xuat();
* currentProcess.timeLeft = -1;
* }
* break;
* }
* else
* {
* //Xét trường hợp đang chạy thì không bị process mới nhảy vào
* while (pc[index].paTime - currentProcess.timeLeft > time)
* {
* if (currentProcess.timeLeft == -1)
* {
* if (pQueue.isEmpty() == true) break;
* else currentProcess = pQueue.PopAtMin2();
* }
* //if (currentProcess.staTime == -1)
* {
* currentProcess.staTime = time;
* }
* time += currentProcess.timeLeft;
* currentProcess.taTime = time - currentProcess.paTime;
* currentProcess.timeLeft = 0;
* currentProcess.fiTime = time;
* currentProcess.wTime = currentProcess.fiTime - currentProcess.paTime - currentProcess.pbTime;
* run.Push(currentProcess);
* currentProcess.Xuat();
* // cout << "In ra o while thu 2\n";
* if (pQueue.isEmpty() != true)
* {
* currentProcess = pQueue.PopAtMin2();
* }
* else
* {
* currentProcess.timeLeft = -1;
* }
* }
* if (pQueue.isEmpty() == true && currentProcess.timeLeft==-1)
* {
* time = pc[index].paTime;
* int j = index;
* while (j + 1 < soLuong && pc[j].paTime == pc[j + 1].paTime)
* {
* j++;
* }
* for (int k = index; k <= j; k++)
* {
* pQueue.Push(pc[k]);
* }
* index = j + 1;
* currentProcess = pQueue.PopAtMin2();
* }
* //Nếu như có 1 process mới nhảy vào hàng đợi lúc đang thực hiện
* else
* {
* if (currentProcess.staTime == -1)
* {
* currentProcess.staTime = time;
* }
* currentProcess.timeLeft -= (pc[index].paTime - time);
* //Push nh?ng process c�ng arrival time v�o trong queue
* int j = index;
* time = pc[j].paTime;
* while (j + 1 < soLuong && pc[j].paTime == pc[j + 1].paTime)
* {
* j++;
* }
* for (int k = index; k <= j; k++)
* {
* pQueue.Push(pc[k]);
* }
* index = j + 1;
* //Xem xét tiến trình đang thực hiện có thể bị đá ra hay không
* Process maybeCurrent = pQueue.ViewAtMin();
* if (currentProcess.timeLeft > maybeCurrent.timeLeft)
* {
* run.Push(currentProcess);
* currentProcess.Xuat();
* pQueue.Push(currentProcess);
* //cout << currentProcess.pname << " bi da boi " << maybeCurrent.pname<<"\n";
* currentProcess = maybeCurrent;
* pQueue.PopAtMin2();
* time = maybeCurrent.paTime;
* }
* else time = pc[j].paTime;
* }
* if (index == soLuong)
* {
* while (pQueue.isEmpty() != true)
* {
* if (currentProcess.timeLeft == -1)
* currentProcess = pQueue.PopAtMin2();
* if (currentProcess.staTime == -1)
* {
* currentProcess.staTime = time;
* }
* time += currentProcess.timeLeft;
* currentProcess.taTime = time - currentProcess.paTime;
* currentProcess.fiTime = time;
* currentProcess.timeLeft = 0;
* currentProcess.wTime = currentProcess.fiTime - currentProcess.paTime - currentProcess.pbTime;
* run.Push(currentProcess);
* currentProcess.Xuat();
* currentProcess.timeLeft = -1;
* }
* break;
* }
* }
* }
* if (currentProcess.timeLeft != -1)
* {
* if (currentProcess.staTime == -1)
* {
* currentProcess.staTime = time;
* }
* time += currentProcess.timeLeft;
* currentProcess.taTime = time - currentProcess.paTime;
* currentProcess.fiTime = time;
* currentProcess.timeLeft = 0;
* currentProcess.wTime = currentProcess.fiTime - currentProcess.paTime - currentProcess.pbTime;
* run.Push(currentProcess);
* currentProcess.Xuat();
* currentProcess.timeLeft = -1;
* }
* int i = 0;
* while (run.isEmpty() != true)
* {
* pc[i]=run.PopAndMerge();
* i++;
* }
* //Tính avgWating và avgTAT
* for (int i = 0; i < soLuong; i++)
* {
* avgWTime += pc[i].wTime;
* avgTATime += pc[i].taTime;
* pc[i].resTime = pc[i].staTime - pc[i].paTime;
* }
* avgWTime /= soLuong;
* avgTATime /= soLuong;
* cout << "\nShort Remaining Job First:\nAverage Waiting Time: " << avgWTime << "\nAverage Turn Around Time: " << avgTATime << "\n";
* }
* };
* Class Algorithm:
  + Chứa các process và Queue để có thể Push các Process vào
  + Và một hàm thực hiện thuật toán SRT
    - B1: Sắp xếp các process theo arrival time tăng dần
    - B2: Thêm các process có cùng
* int main()
* {
* Algorithm a;
* a.Nhap();
* a.ShortRemainingFirst();
* a.Xuat();
* }

Đây là chương trình chính, chương trình sẽ yêu cầu nhập các process sau đó thực hiện thuật toán SRT và xuất ra kết quả

* Kết quả chạy trên ubuntu

## Task 3: