*CT178 – Nguyên lý hệ đều*

*hành* **PROJECT – NGUYÊN LÝ HỆ ĐỀU HÀNH**

**Học kỳ 1 – 2022 -2023**

**HỌ VÀ TÊN: TRẦN HOÀNG KIM MSSV: B2007245**

**Tuyên bố: Project này là do chính tôi, *Trần Hoàng Kim* (MSSV: *B2007245*), tự thực hiện, không sao chép của bất kỳ ai.  Nếu có bất cứ sao chép nào, tôi hoàn toàn chịu trách nhiệm.**

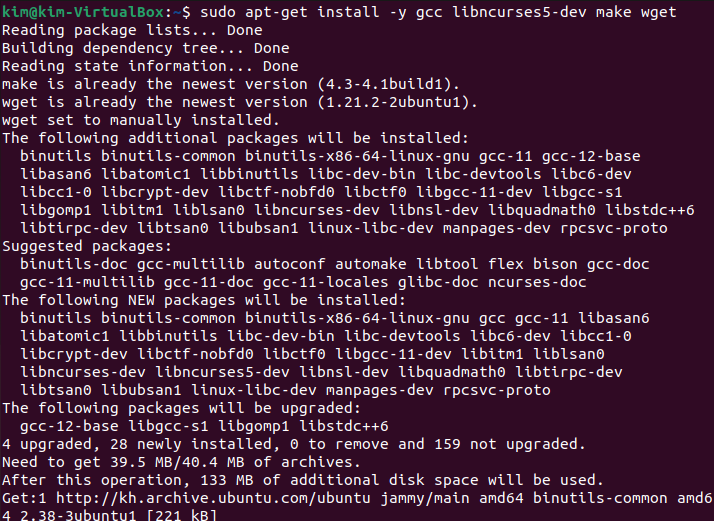
**PHẦN 1: XÂY DỰNG LINUX KERNEL**

Thực hiện ở chế độ người dùng root (root user)

**$su-**

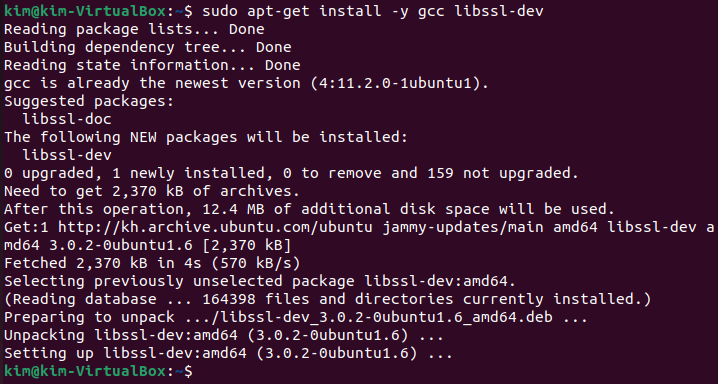
1. CHUẨN BỊ LINUX KERNEL CODE (Ubuntu)
2. Dowload và cài đặt tool cần thiết vào hệ thống

**$sudo apt-get install -y gcc libncurses5-dev make wget**

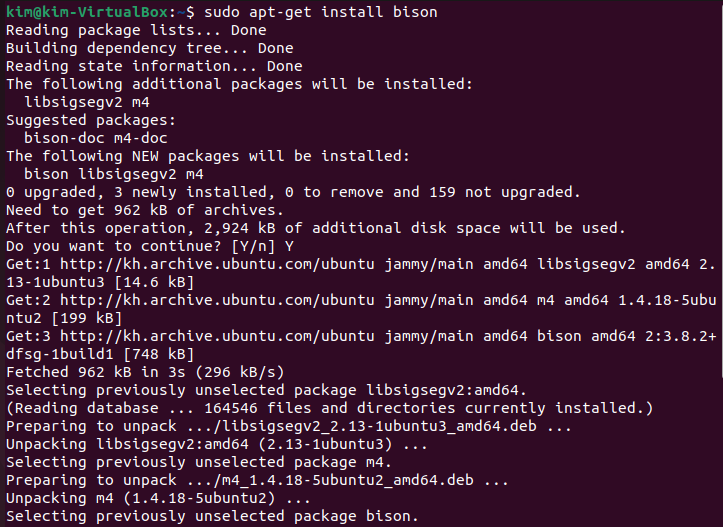


**$sudo apt-get install –y gcc libssl-dev**

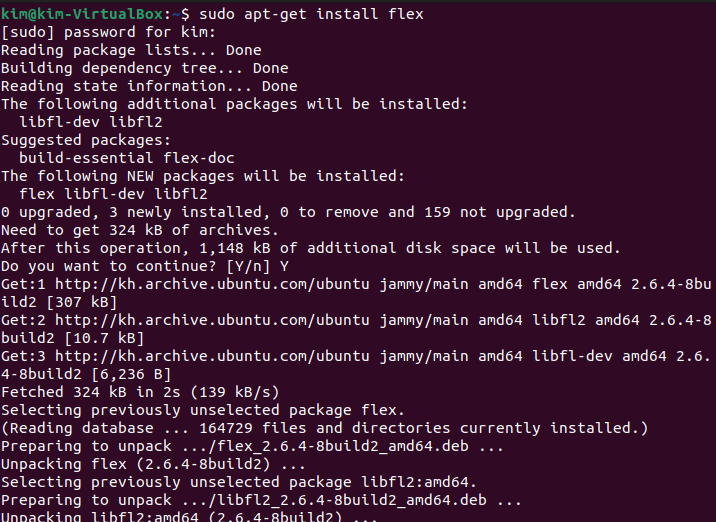
**$sudo apt-get install -y gcc libssl-dev**

****

**$sudo apt-get install bison**

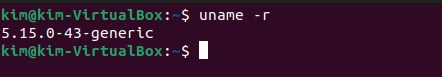
****

**$sudo apt-get install flex**

****

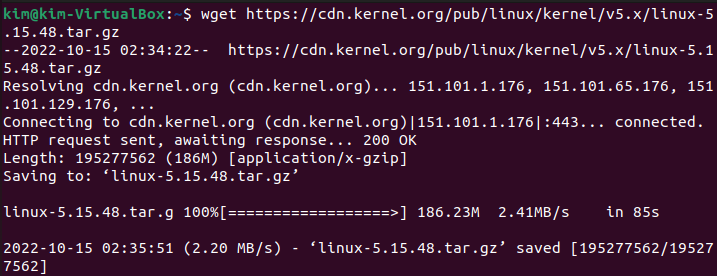
1. Xác định phiên bản hiện tại của kernel

**#uname -r**

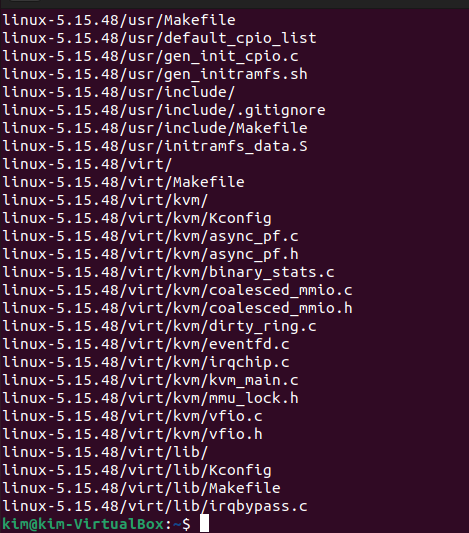


1. Dowload và giải nén

#**wget** [**https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.15.48.tar.gz**](https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.15.48.tar.gz)



**#tar xvzf linux-5.15.48.tar.gz**

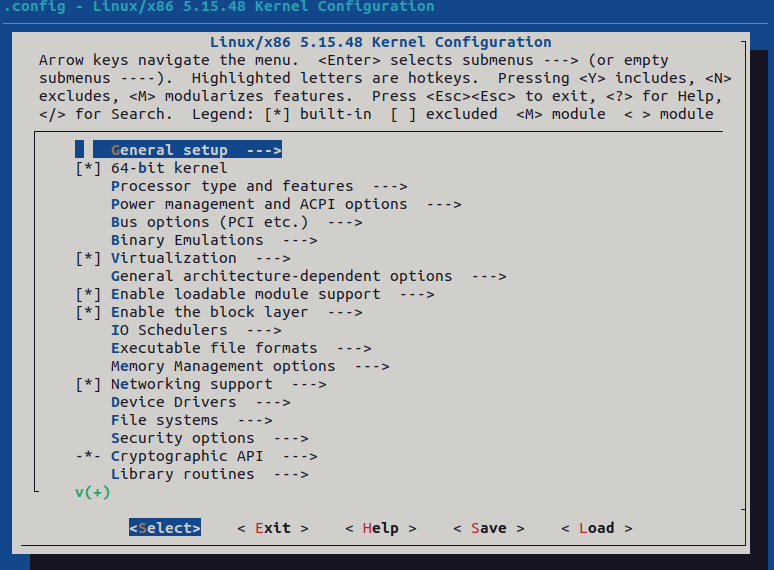


1. CẤU HÌNH KERNEL MỚI
2. Đảm bảo đường dẫn hiện tại ở ~/**linux-5.15.48** và “**linux-5.15.48**” nằm ở top directory của kernel source.



1. Tạo file cấu hình (config file)

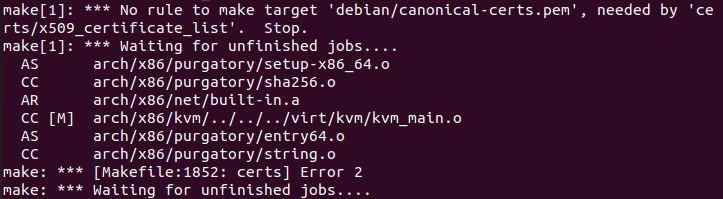
#**make menuconfig**

****

1. BIÊN DỊCH KERNEL
2. Tại **~/linux-5.15.48** tạo kernel giải nén (compressed kernel image)

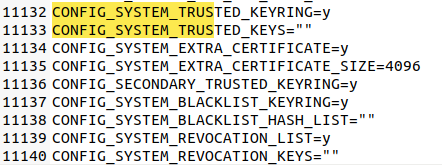
#**make –j4**

Lỗi

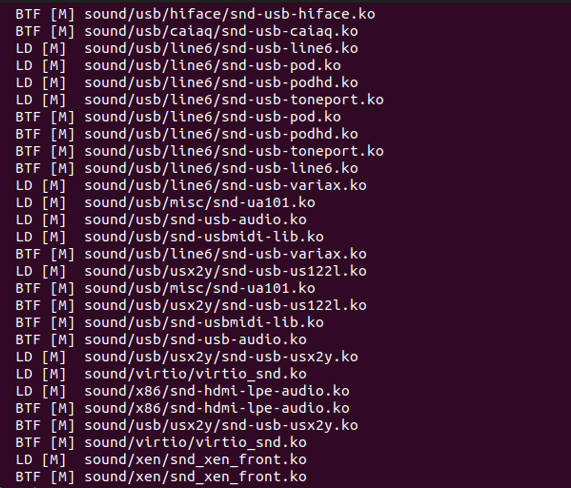


Xử lý

Vào **#gedit .config** chỉnh sửa thông tin như hình

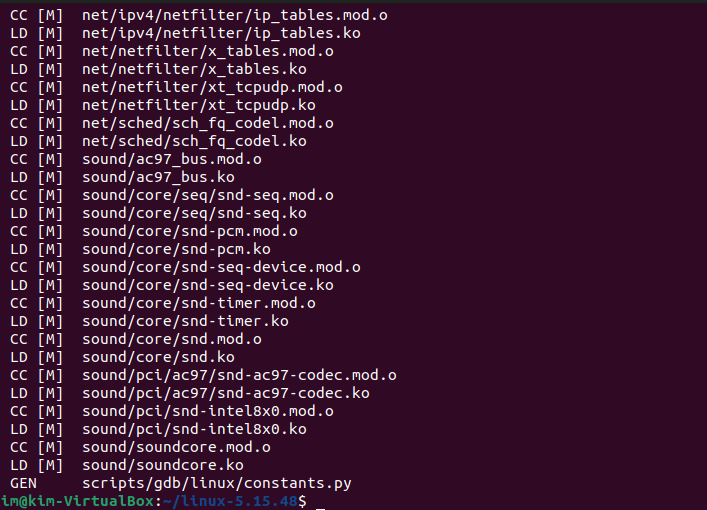


Sau khi thực hiện xong



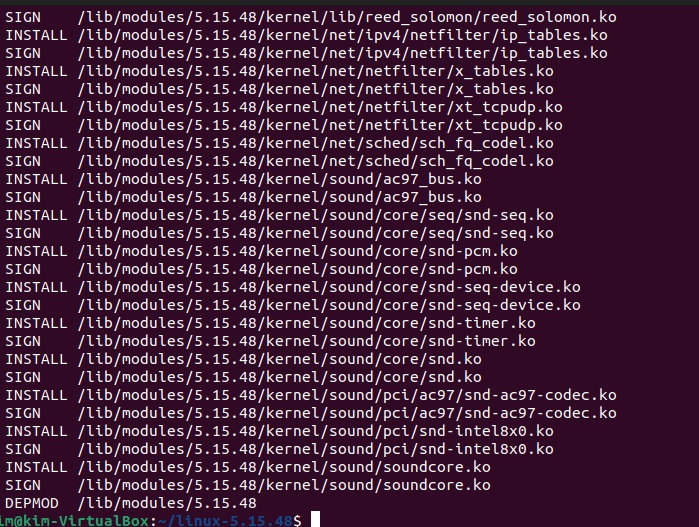
1. Biên dịch kernel modules

#**make modules**

****

1. CÀI ĐẶT KERNEL
2. Cài đặt kernel modules

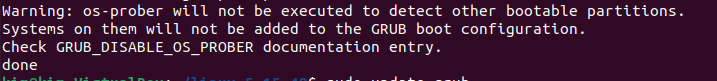
#**make modules\_install**



1. Cài đặt kernel

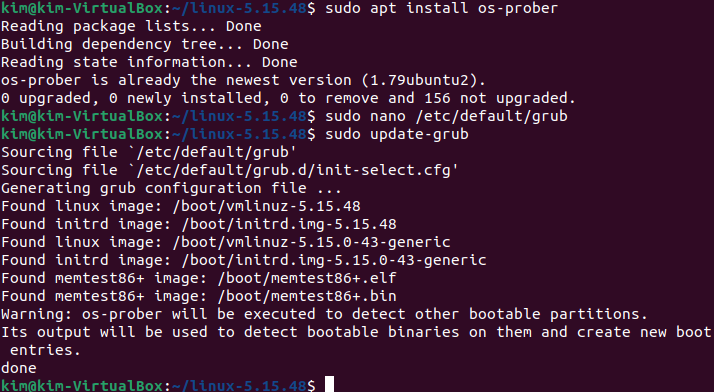
#**sudo make install**

Lỗi

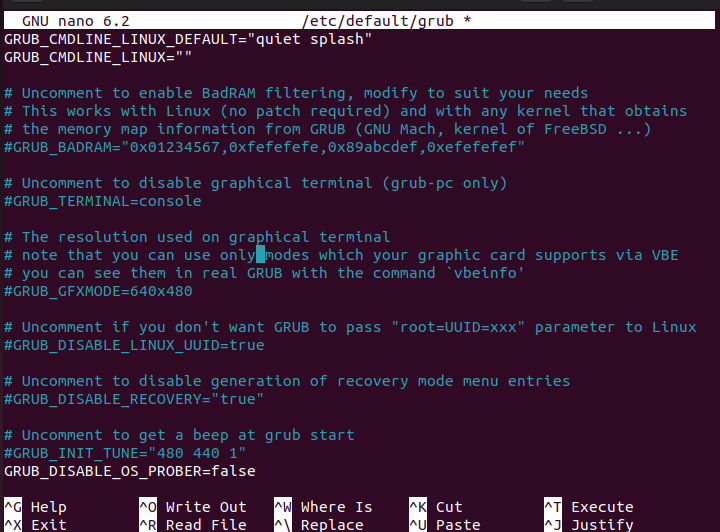


Xử lý

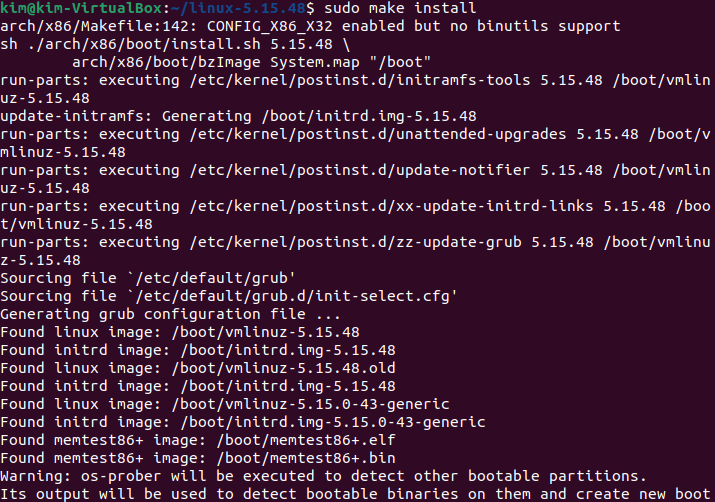
B1: #sudo apt install os-prober



B2: Gõ thêm dòng “GRUB\_DISABLE\_OS\_PROBER=false” save và thoát.

****

Kết quả

****

1. THAY ĐỔI CẤU HÌNH GRUB (GRUB CONFIGURATION FILE)

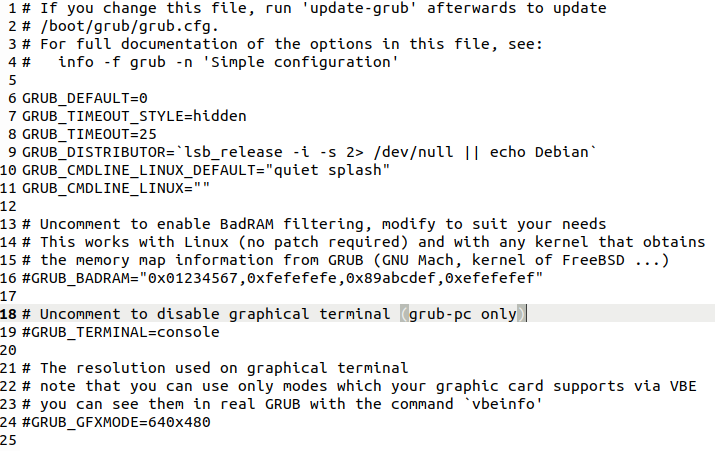
Thay đổi cấu hình grub

**#vim /etc/default/grub**

Thực hiên thay đổi sau

GRUB\_DEFAULT= 0

GRUB\_TIMEOUT=25



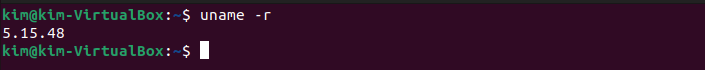
1. REBOOT VM
2. Reboot kernel mới

**#reboot**

Khởi động lại máy ảo

1. Sau khi reboot kiểm tra thông tin kernel mới có đúng chưa

#**uname -r**



**PHẦN 2: THÊM LỜI GỌI HỆ THỐNG VÀO LINUX KERNEL**

1. Download the kernel source code

**#wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.0.tar.gz**



1. Extract the kernel source code

**#tar xvzf linux-6.0.tar.gr**

****

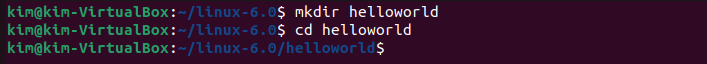
1. Define a new system call sys\_helloworld()

* Create a dicetory helloworld in the kernel source directory :-

**#mkdir helloworld**

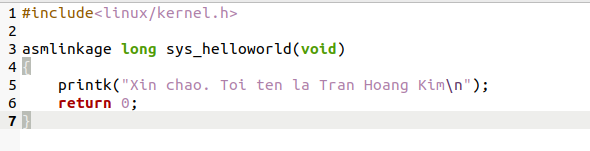
Change into this directory

**#cd helloworld**

****

1. Create a “helloworld.c” file in this folder and add the definition of the system call to it as given below (you can use any text editor).

**#gedit helloworld.c**

****

1. Create a “Makefile” in the helloworld folder and add the given line to it.

**#gedit Makefile**

Add the following line to it :-

**Obj-y := hello.o**

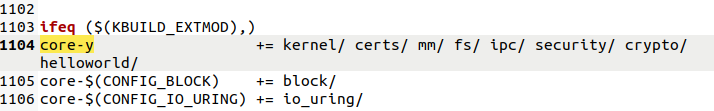
****

* Add the helloworld directory to the kernel’s Makefile

Change back into the linux-6.0 folder and open Makefile

and add the helloworld at the end of core-y

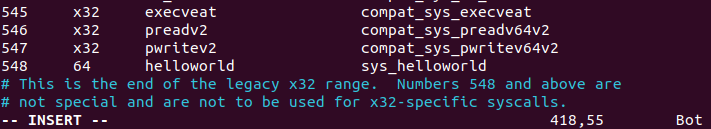
**#gedit Makefile**



Add the new system call (sys\_helloworld()) into the system call table (syscall\_64.tbl file) (for 64 bit system)

**#vi arch/x86/entry/syscalls/syscall\_64.tbl**

and add the end of line following

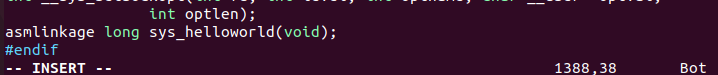


Add the new system call (sys\_helloworld()) in the system call header file.

**#vi include/linux/syscalls.h**

Add the following line to the end of the file just before #endif statement at the very bottom.

**#asmlinkage long sys\_helloworld(void);**

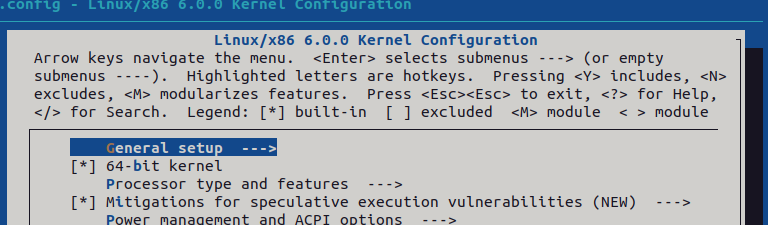


Compile kernel

1. Download necessary tools
2. Make menuconfig

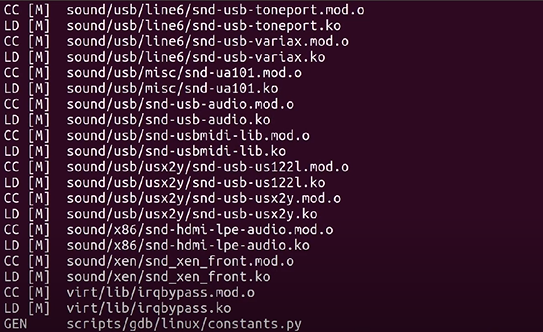
**#sudo make menuconfig**

Save and exit

****

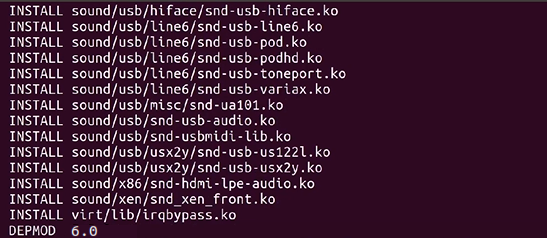
Now, compile the kernel (chỉnh sửa lỗi như phần trên)

**#make**

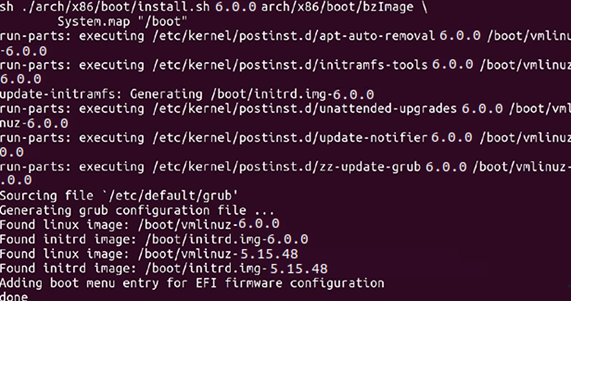


To install/ update the kernel

**#sudo make modules\_install**



**#sudo make install**

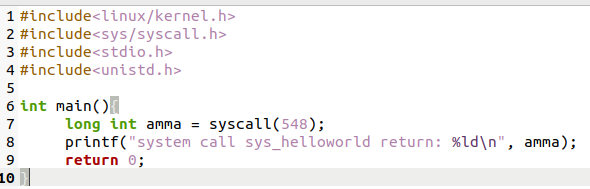


Reboot and test system call

**#uname –r**

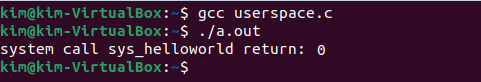


Create “userspace.c”



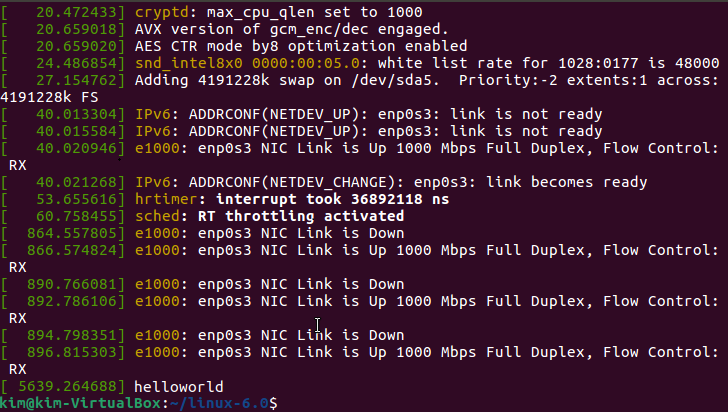
Following these step

If return 0 is right



**#dmesg**

This will display “Hello world” at the end of kernel’s message

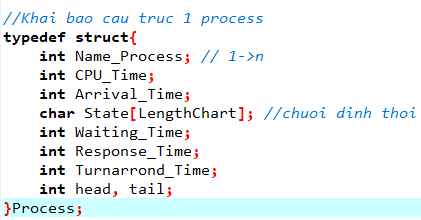


**PHẦN 3: ĐỊNH THỜI CPU**

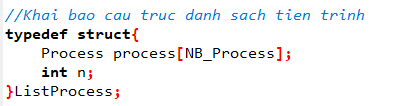
Thuật toán FCFS

* + Đúng như cái tên của thuật toán Frist-Come First-Serve tiến trình nào tới đầu tiên sẽ được thực hiện trước.
  + Ý tưởng: Ta sẽ dựa trên biểu đồ Grantt Chart và lưu trữu những thông tin trên biểu

đồ đó bằng 1 cấu trúc mảng với mỗi tiến trình là một phần tử.



* + - * head: lưu trữ thời gian lần đầu tiên bắt gặp và thực hiện tiến trình.
      * tail: lưu trữ thời gian hoàn thành tiến trình.
      * Tùy theo mỗi bài toán non-premtive hay premtive head, tail sẽ có cấu trúc khác nhau.
* Khai báo 1 mảng cấu trúc các tiến trình.



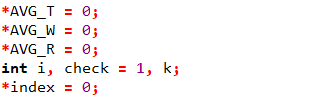
* Các hàm hỗ trợ cho danh sách tiến trình
* void initListProcess(ListProcess1 \*list): khởi tạo mảng tiến trình
* void appendListProcess(ListProcess1 \*list, Process1 x): thêm 1 tiến trình vào mảng
* void Input(ListProcess \*list): nhập dữ liệu
* void sort(ListProcess \*list): sắp xếp tiến trình theo thời gian đến hệ thống
* void output (int Chart[], int index): in biểu đồ grantt chart

* Hàm FCFS

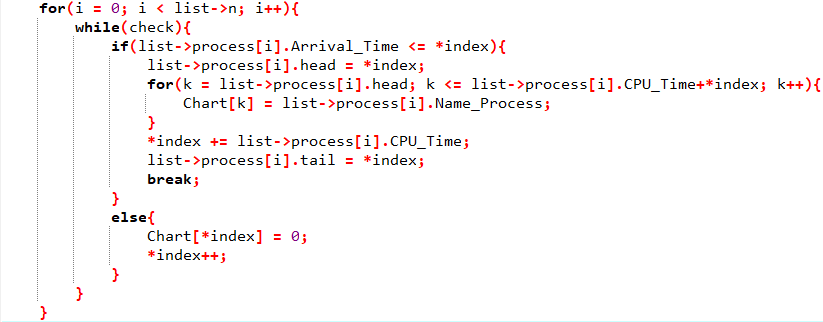


* Cần 1 mảng Chart và thật sự cần thiết 1 biến giữ tổng thời gian thực hiện hết tiến trình \*index

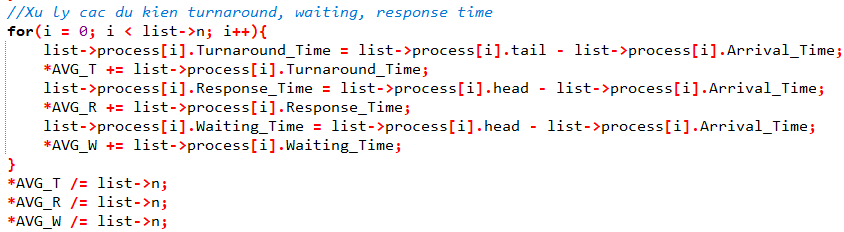
* Khởi tạo các giá trị cần tính



* Lặp trong n tiến trình và tính các giá trị head, tail của mỗi tiến trình.



* Kiểm tra tiến trình đó tới tại vị trí \*index = 0 hay chưa không thì tăng \*index++ và tại vị trí Chart[\*index] này không có tiến trình nào nên gán bằng 0.
* Ngược lại nếu có, sẽ lặp CPU Time lần của tiến trình này tức là từ \*index -> \*index + CPU Time của tiến trình đó gán vào mảng Chart[\*index] = tên tiến trình đại diện cho tại s đó.
* Tiến trình đó được thực hiện, head = \*index và tail = \*index + CPU Time, \*index += CPU Time của tiến trình đó.
* Xử lý các dự kiện cần tính.



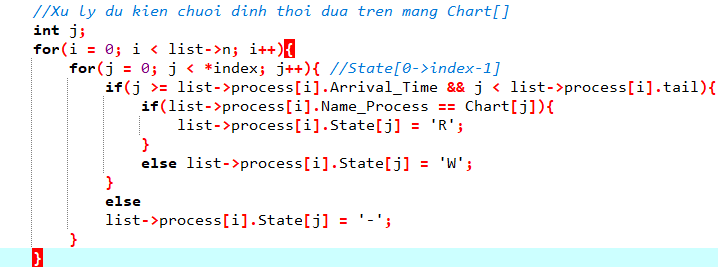
* + - Turnaround Time = Thời gian tiến trình đến hệ thống cho tời khi hoàn thành tiến

trình.

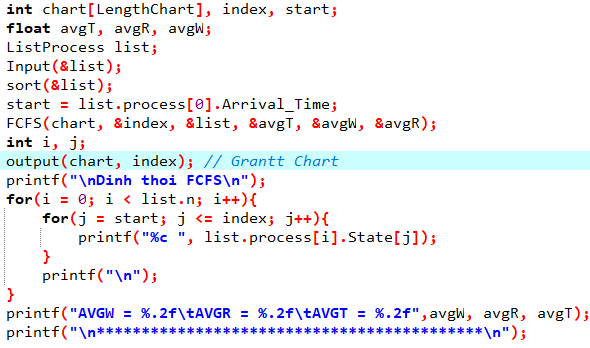
* + - Respone Time = Thời gian tiến trình đến hệ thống cho tới khi bắt gặp tiến trình

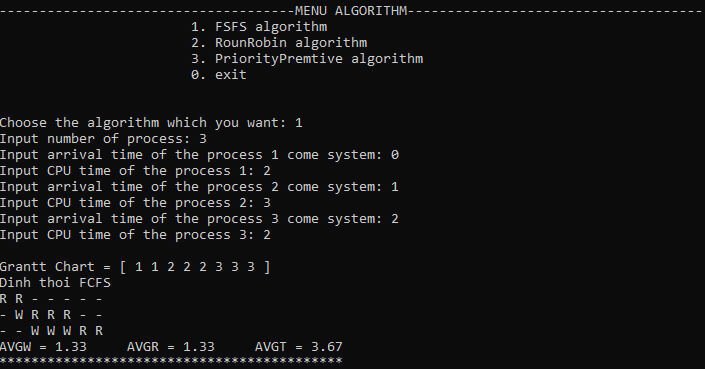
lần đầu tiên.

* + - Waiting Time = Thời gian tiến trình nằm trong hàng đọi Ready.
* Ta dễ dàng tính được các giá trị trung bình.
* Xử lý chuỗi định thời.
* “R” đại diện cho tiến trình đang được thực hiện.
* “W” đại diện cho tiến trình đang nằm trong hàng đợi Ready
* “-“ đại diện cho tiến trình đã thực hiện xong và không còn nằm trong hàng đợi Ready



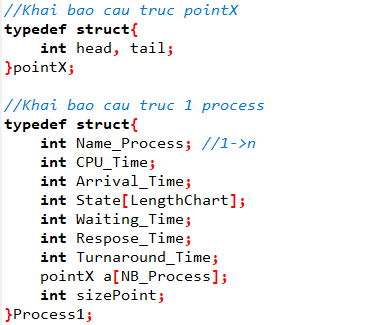
* Ta sẽ lặp trong tổng thời gian thực hiện các tiến trình tức là \*index.
* Xét trong khoảng thời gian từ head => tail của mỗi tiến trình.
* Nếu tại đó tên tiến trình bằng với Chart[j] tức là tiến trình đang được thực hiện ta gán = “R”
* Ngược lại tức là tiến trình đang đợi trong hàng đợi Ready.
* Nếu vượt khỏi miền giá trị của tiến trình đó thì sẽ gán = “-“ đã thực hiện xong.
* Hàm main.



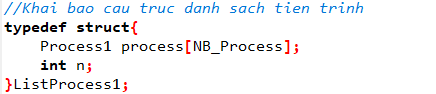
* Chạy chương trình

Thuật toán RoundRobin

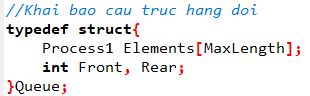
* Thực hiện trong q thời gian phải trả CPU cho hệ thống
* Ý tưởng giống với thuật toán trên nhưng sẽ khác 1 ít trong phần cài đặt



* + Chú ý rằng ở đây không đơn thuần giữ 1 điểm head với 1 điểm tail bởi vì tiến trình chỉ được phép thực hiện trong q thời gian phải trả CPU về cho hệ thống nên ta phải giữ 1 mảng các head, tail bằng struct pointX, cho nên trong Struct Process1 cần biến sizePoint để giữ số lượng các cặp head, tail này để tính các giá trị cần thiết.
  + Khai báo struct 1 mảng các tiến trình



* Các hàm hỗ trợ cho danh sách tiến trình
* void initListProcess1(ListProcess1 \*list): khởi tạo mảng tiến trình
* void appendListProcess1(ListProcess1 \*list, Process1 x): thêm 1 tiến trình vào mảng
* void Input1(ListProcess \*list): nhập dữ liệu
* void sort1(ListProcess \*list): sắp xếp tiến trình theo thời gian đến hệ thống
* void output1(int Chart[], int index): in biểu đồ grantt chart
* Ta sẽ khai báo thêm 1 struct hàng đợi để giữ các tiến trình thực hiện chưa xong CPU Time



* Các hàm hỗ trợ cho hàng đợi Queue
* void makeNullQueue(Queue \*Q): khởi tạo hàng đợi
* int isEmptyQueue(Queue Q): kiểm tra hàng đợi rỗng ?
* int isFullQueue(Queue Q): kiểm tra hàng đợi đầy ?
* void appendQueue(Queue \*Q, Process1 x): thêm 1 phần tử vào hàng đợi
* Process1 Front(Queue Q): trả về tiến trình đầu hàng đợi
* int headToPostion(Process1 p, ListProcess1 list): Xác định vị trí tiến trình trong

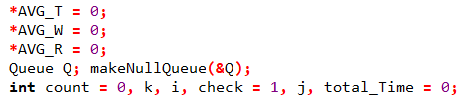
danh sách các tiến trình

* void deQueue(Queue \*Q): xóa phần tử trong hàng đợi
* int equals(Process1 x, Process1 y): trả về true nếu x.name == y.name else false
* int member(Queue Q, Process1 x): hàm thành viên kiểm tra xem 1 tiến trình nào đó
* có nằm trong hàng đợi hay không
* Hàm RR



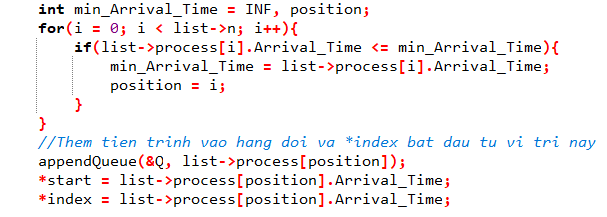
* + Với \*index sẽ giữ tổng thời gian của tiến trình và \*start sẽ xác định giây mà tiến trình tới đầu tiên nhất

* Khởi tạo các giá trị cần thiết



* Khai báo và khởi tạo thêm 1 hàng đợi Q

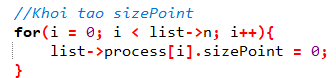
* Công việc đầu tiên là xác định tiến trình đầu tiên sẽ được thêm vào hàng đợi



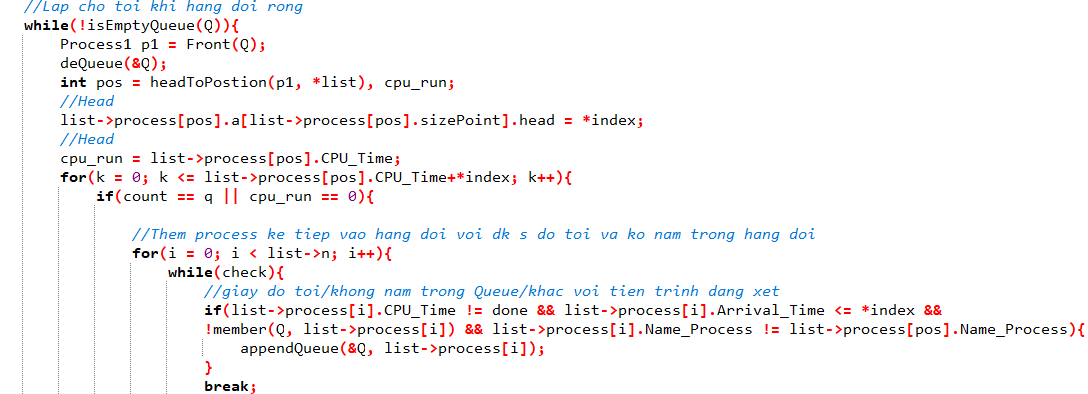
* Nếu tiến trình nào có Arrival\_Time nhỏ nhất sẽ được thêm vào hàng đợi

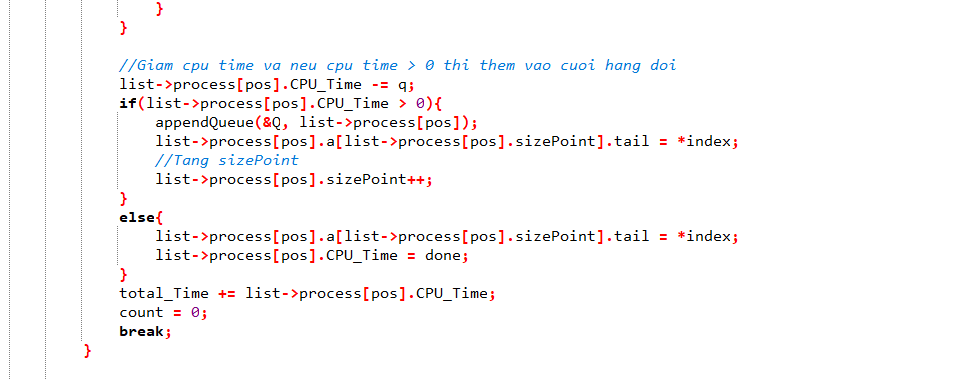
Và tính giá trị cho \*start, \*index

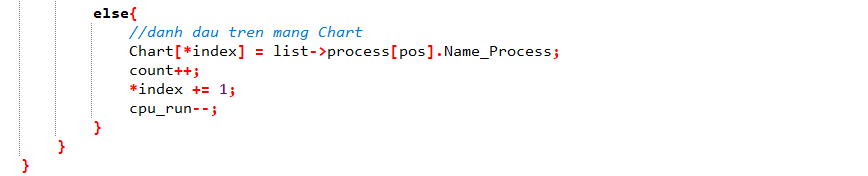
* Khởi tạo mảng head, tail cho từng tiến trình



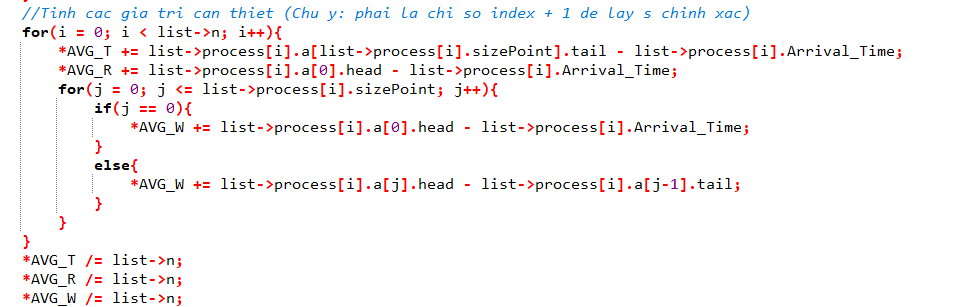
* Vòng lặp chính







* Đầu tiên phần tử đâu tiên được lấy ra và xóa khỏi trong hàng đợi
* Biến pos dùng để xác định xem tiến trình trong hàng đợi đang ở index nào trong ListProcess
* Biến cpu\_run dùng để đếm số lần q
* Tính giá trị head trong sizePoint đã được cấp phát ở trên
* Gán cpu\_run = CPU Time của tiến trình (CPU Time của tiến trình sẽ đc cập nhật sau mỗi lần lặp)
* Lặp trong CPU Time của tiến trình, ở đây bắt đầu từ k = 0 cũng không ảnh hưởng nhiều đến chương trình vì ngay ở bên dưới có 1 lệnh if để check điều kiện và break bất cứ lúc nào nếu điều kiện đúng
* Với lệnh if(count == q || cpu\_run == 0) nếu không đúng điều kiện thì sẽ chuyển xuống lệnh else bên dưới, mảng Chart[\*index] sẽ được gán tên tiến trình, count++ đến số q tăng lên và cpu\_run-- cập nhật lại cpu\_run để tiếp tục check điều kiện dừng
* Nếu đúng với điều kiện đó chúng ta sẽ tiến hành tìm các tiến trình đến trong giây đó với điều kiện “Tiến trình đó chưa xong tức là != done + Thời gian đến <= \*index + chưa nằm trong hàng đợi đang lặp + tên tiến trình đó phải khác tiến trình đang xét’
* Cập nhật lại các giá trị CPU Time
* if CPU Time > 0 tức là vẫn chưa thực hiện xong, ta sẽ tính giá trị tail cho nó và phải tăng sizePoint để giữ head, tail tiếp theo
* Ngược lại thì ta chỉ tính tail cho nó và gán CPU Time == done đánh dấu hoàn thành tiến trình
* Biến total\_time giữ số giây thực hiện các tiến trình sẽ được cập nhật sau mỗi lần lặp
* Reset biến count để đếm số lần q về = 0
* Break
* Tiếp tục lặp cho tới khi hoàn thành xong hết các tiến trình
* Tính các giá trị cần thiết



Ta thấy rằng tại vị trí sizePoint tiến trình mới thực hiện xong

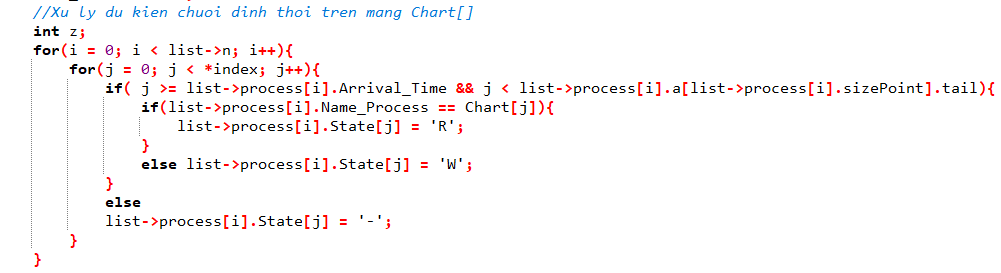
Tại vị trí sizePoint = 0 thì đó là lần đầu tiên đến hệ thống

Như đã nói ở trên thuật toán này sẽ giữ mảng head, tail để giữ đầu cuối của mỗi Running Time

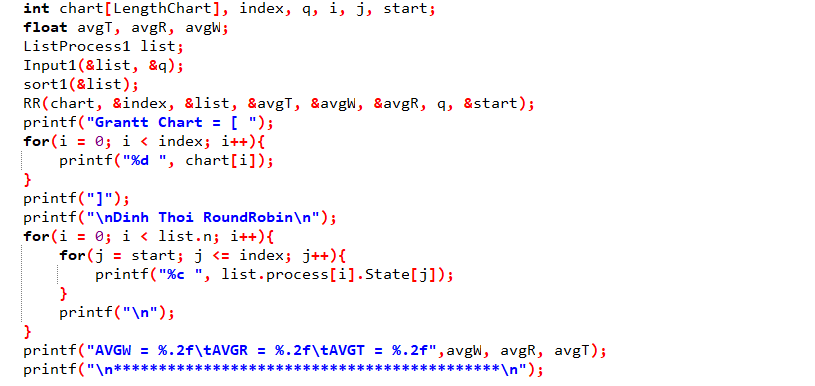
Nên ở đây ta cần tạo 1 vòng lặp để tính nó

Ta dễ dàng tính được các giá trị trên

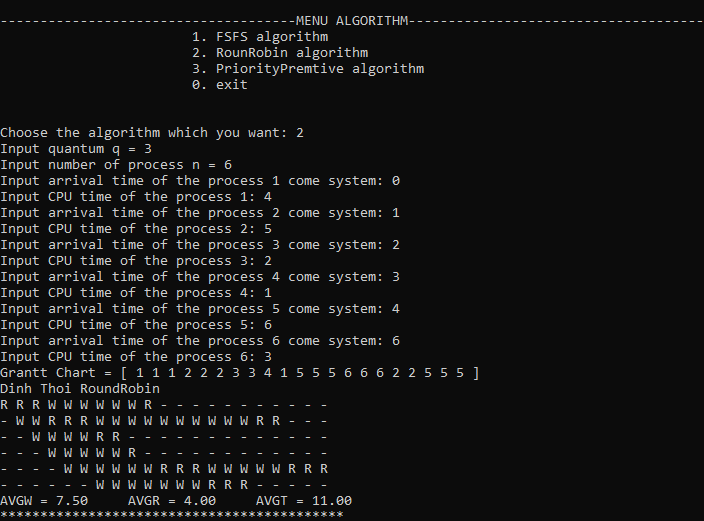
* Xử lý trên chuỗi định thời



* Cũng tương tự như bài toán FSFC nhưng ta phải cần vòng lặp lồng cho mảng head, tail để xét từng miền giá trị cho mỗi cặp giá trị đó
* Hàm main

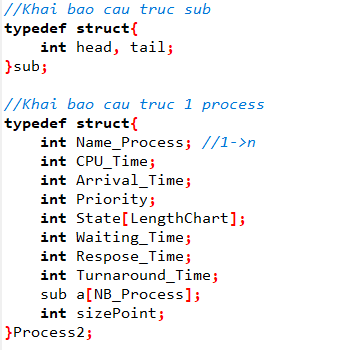


* Chạy chương trình

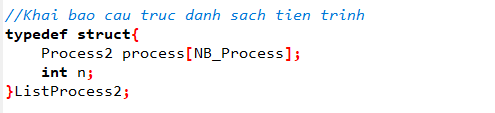


Thuật toán Priority Premtive

* Tiến trình nào được thực hiện trước phụ thuộc vào thời gian đến và piority của tiến trình đó
* Ý tưởng cũng tương tự như 2 thuật toán ở trên nhưng vẫn có 1 số phần khác trong cài đặt
* Khai báo cấu trúc 1 process sẽ có thêm thuộc tính priority



* Vẫn phải là mảng head, tail vì đây là Premtive
* Khai báo danh sách các mảng Process

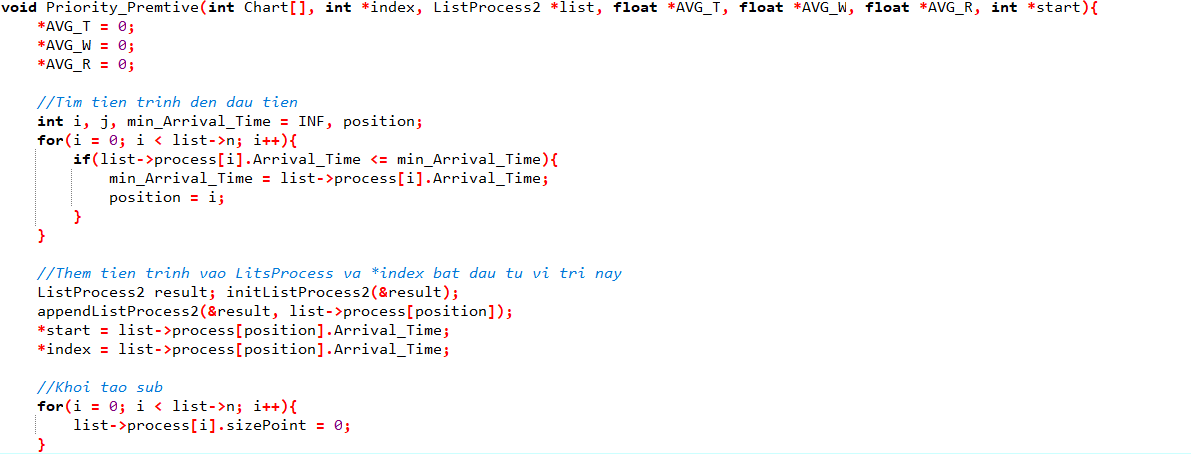


* Các hàm hỗ trợ cho danh sách process

* void initListProcess2(ListProcess2 \*list): khởi tạo danh sách
* int isEmpty(ListProcess2 list): kiểm tra danh sách rỗng ?
* int isFull(ListProcess2 list, int nbProcess): kiểm tra danh sách đầy?
* void appendListProcess2(ListProcess2 \*list, Process2 x): thêm một tiến trình vào

ListProcess

* void deleteListProcess2(ListProcess2 \*list, int index): xóa tiến trình tại index
* void Input2(ListProcess2 \*list): nhập dữ liệu
* Process2 getNextProcess(ListProcess2 list): chọn tiến trình kế tiếp để thực hiện
* int PtoIndex(Process2 p, ListProcess2 list): trả về vị trí của tiến trình trong ListProcess
* int member1(Process2 x, ListProcess2 result): kiểm tra thành viên
* int smallest\_Priority\_current(ListProcess2 list): trả về tên của tiến trình có priority nhỏ I
* Hàm PriorityPremtive

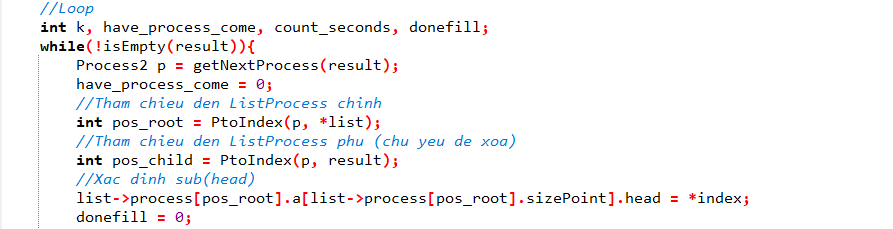


* Các việc như khởi tạo, tìm tiến trình đầu tiên vào danh sách ListProcess, gán các giá trị \*start

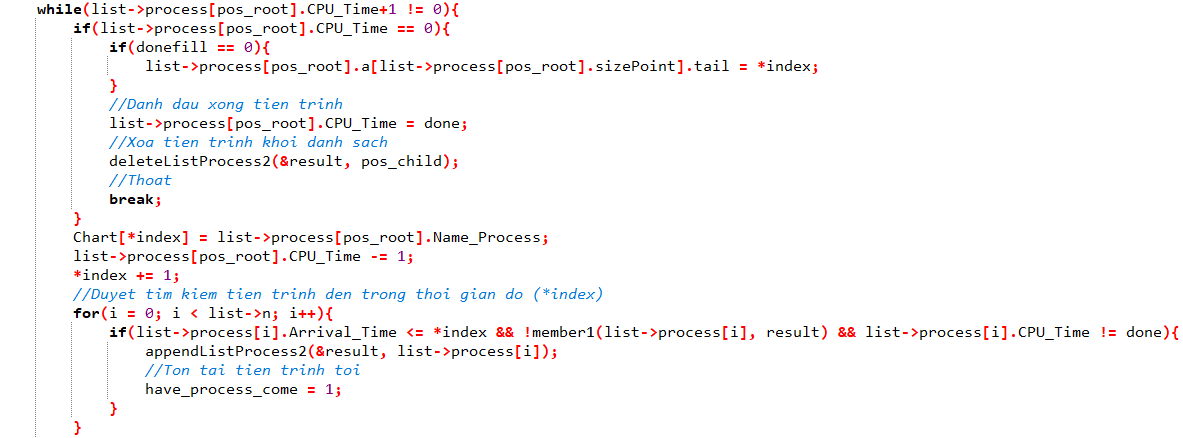
\*index, vào khởi tạo mảng các cặp head, tail thì tương tự như thuật toán RR bên trên nhưng ta

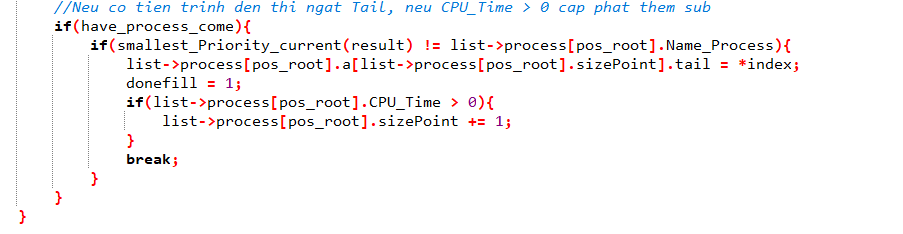
không sử dụng hàng đợi ta sẽ sự dụng luôn cấu trúc ListProcess.

* Vòng lặp chính

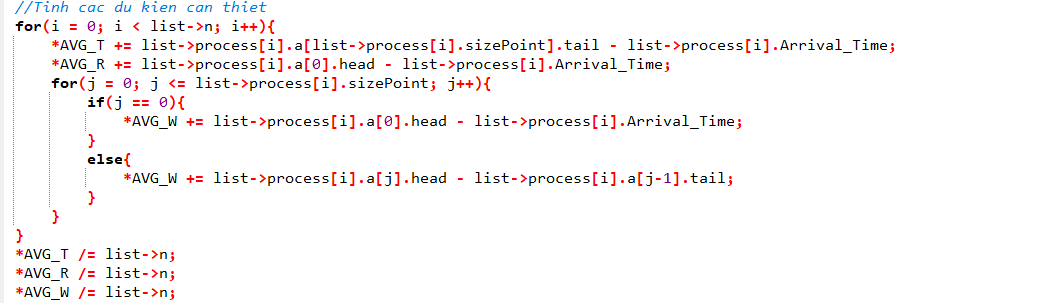


* Lặp cho đến khi mảng các tiến trình result rỗng
* Tiến trình đầu tiên được lấy ra, với hàm getNextProcess thì tiến trình nào có priority nhỏ nhất sẽ được lấy ra
* Biến have\_process\_come kiểm tra xem có tiến trình nào tới hay không
* Biến pos\_root có mục đích là tìm xem tiến trình đó nằm ở đâu trong ListProcess chính bởi vì ta cần chỉ số đó mới có thể cập nhật được thông số CPU Time ở root mà ở đây là ListProcess
* Biến pos\_child thì chỉ chủ yếu để xóa và kiểm tra điều kiện vòng lặp chính
* Xác định head, tail cho tiến trình tại vị trí sizePonit mà được cấp phát ở trên
* Biến donefill là biến xác định xem tiến trình đó có được thực hiện xong hay chưa
* Trong vòng lặp con của vòng lặp chính

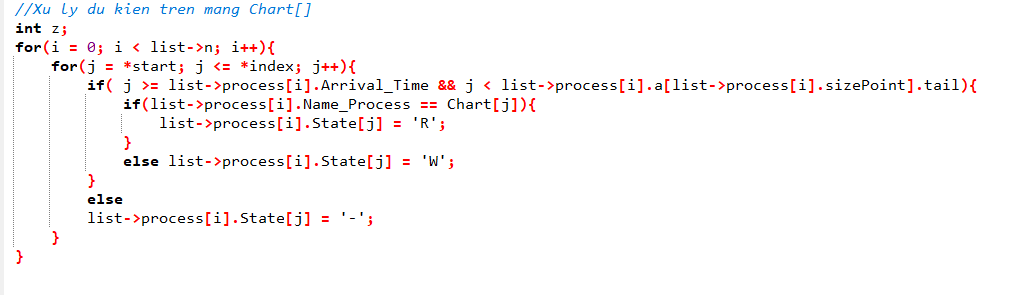




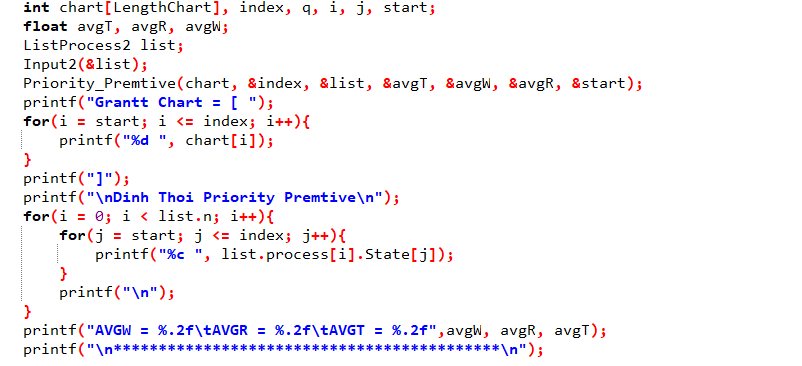
* Lặp trong CPU Time+1 của tiến trình
* Nếu CPU Time của tiến trình đó chưa = 0 thì thêm vào mảng Chart[\*index] = tên của tiến trình
* Cập nhật CPU Time của tiến trình
* Cập nhật \*index
* Vòng lặp for là tìm kiếm tiến trình coi trong giây đó có tiến trình nào tới không nếu có thì thêm vào mảng result với điều kiện “CPU Time <= \*index + Không phải thành viên của result và tiến trình đó thực hiện chưa xong khác done, và gán have\_process\_come = 1 báo hiệu là có tiến trình tới ta phải đến bước kế tiếp xét coi tiến trình nào có độ ưu tiên cao nhất
* Nếu mà các tiến trình vừa thêm ở bước trên khi gọi hàm smallest\_Priority\_curent(result) tức là tiến trình có priority nhỏ nhất hiện trong mảng tiến trình result không phải là tiến trình mình đang xét thì mình sẽ tính tail cho tiến trình đó và gán điểm chặn tail donefill = 1 và break khỏi vòng lặp con của vòng lặp chính và tìm xét tiếp tiến trình tiếp theo.
* Nếu mà CPU Time của tiến trình đã = 0 nếu điểm chặn donefill = 0 thì ta luôn tính luôn tail, đánh dấu tiến trình đó đã xong = done và xóa tiến trình đó ra khỏi ListProcess của result, quá trình cứ lặp đi lặp lại cho đến khi result rỗng
* Tính các giá trị cần thiết



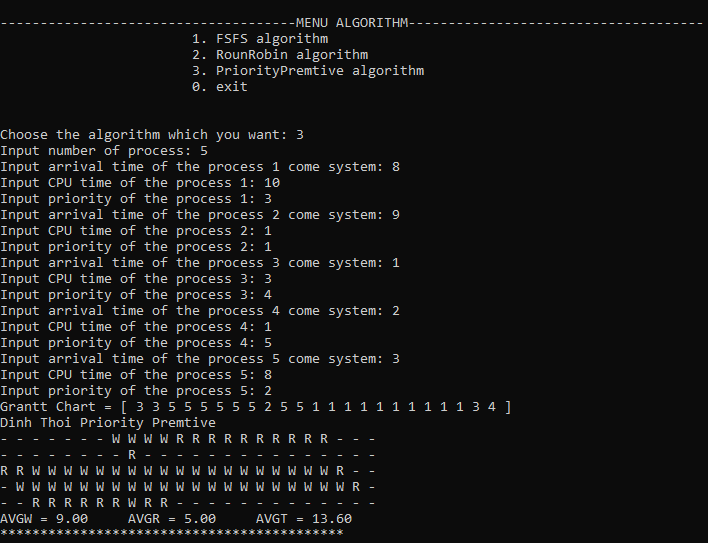
* Tương tự như là cách tính của thuật toán RoundRobin
* Xử lý chuỗi định thời



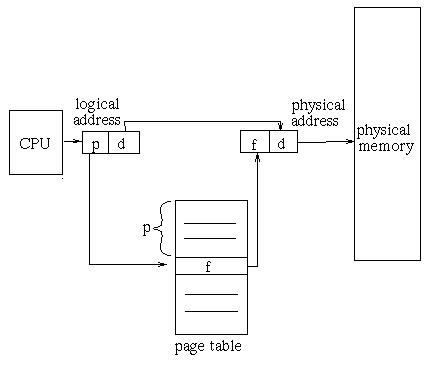
* Tương tự như là các tính của thuật toán RoundRobin
* Hàm main

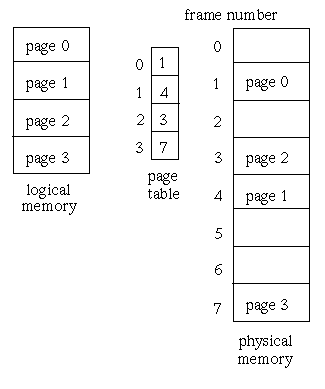


* Chạy chương trình

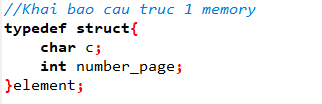


**PHẦN 4: QUẢN LÝ BỘ NHỚ**

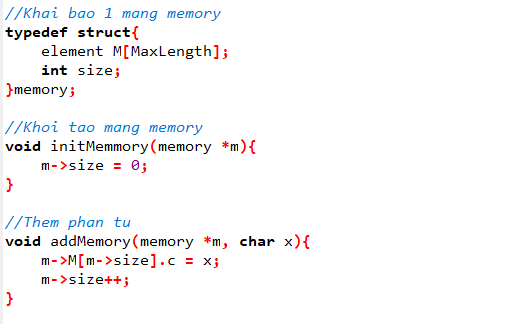
* Với sơ đồ



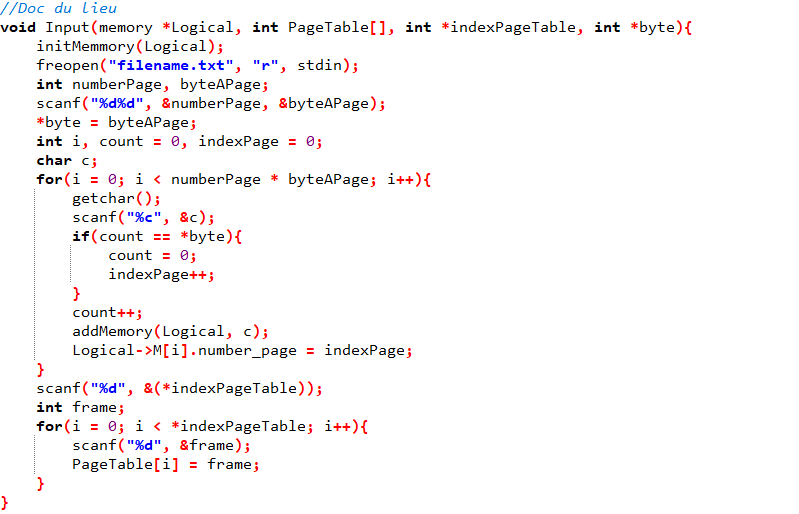
* Xét ví dụ
* Ta thấy rằng với page 0 đang ở vị trí 0 trong Logical memory, xét đến tại chỉ số 0 trong page table ta thấy rằng có frame = 1 nên tại Physical memory ta sẽ thêm vào vùng nhớ tại chỉ số bằng frame tức là bằng 1 page 0 này, cứ tiếp tục như vậy đối với page1, page2, page3, ta thấy rằng việc đó dễ dàng để tạo một chương trình như sau:
* Đầu tiên ta sẽ khai báo cấu trúc 1 vùng memory



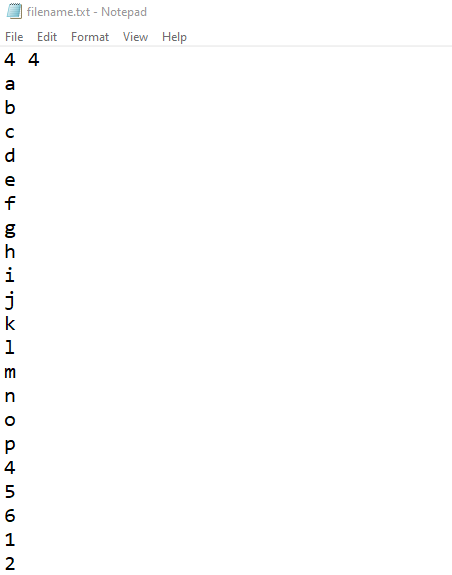
* Tiếp đến ta sẽ khai báo 1 mảng các vùng memory



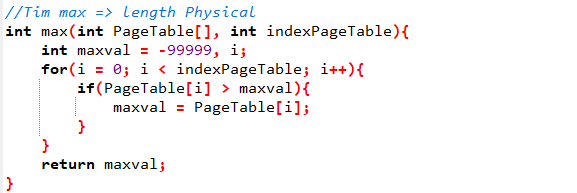
* Với hàm đọc dữ liệu đầu vào



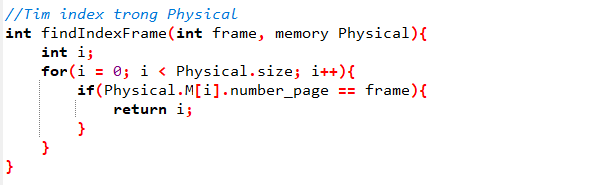
* Ta cần filename.txt có định dạng như sau



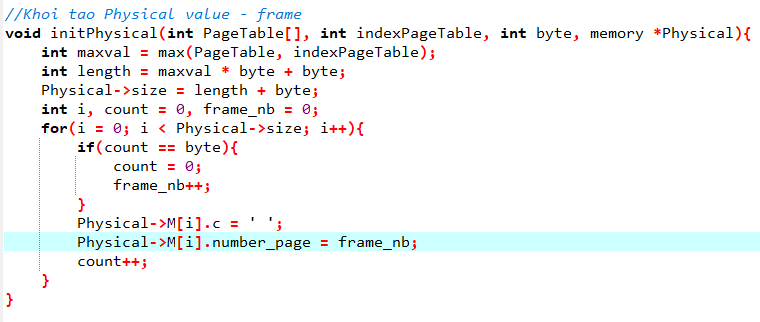
* Hàm tìm maxval mục đích là để tạo size cho mảng Physical memory



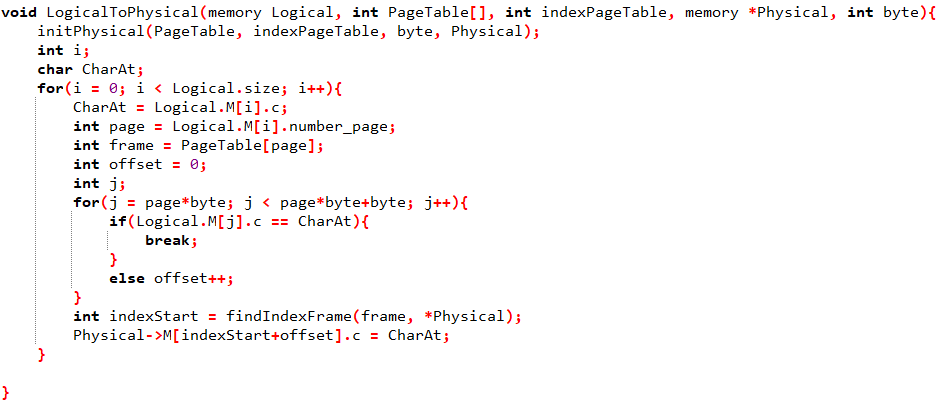
* Hàm findIndexFrame mục đích là tìm kiếm index của Physical memory mà tại đó có frame tức là vùng sẽ được thêm vào từ Logical memory



* Hàm khởi tạo initPhysical khởi tạo các vùng frame và gán char tại đó bằng rỗng



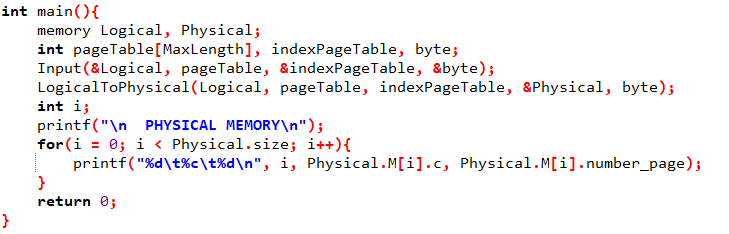
* Hàm LogicalToPhysical



* Việc đầu tiên ta sẽ khởi tạo Physical memory
* Duyệt từng phần tử trong Logical memory
* CharAt/ lấy phần tự tại vị trí i
* page/ Xét xem tại phần tử này là trang mấy ở trong Logical memory
* Khi có page/ ta sẽ tìm kiếm nó sẽ lưu ở frame mấy của Physical memory

frame = PageTable[page]

* Biến offset là biến độ dời
* Ta sẽ lặp từ đầu tới cuối của trang đó với số byte đề bài cho để tính offset
* biến indexStart sẽ thực hiện công việc là tìm index trong Physical memory
* Tại mảng Physical memory tại vin trí indexStart + offset sẽ là vị trí mà charAt được thêm vào
* Hàm main



* Chạy chương trình

