

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH



Hệ điều hành (TN)

Bài tập lớn số 1

GV: Vũ Văn Thống
SV: Võ Minh Long 1812951

Ho Chi Minh City, 05/2020



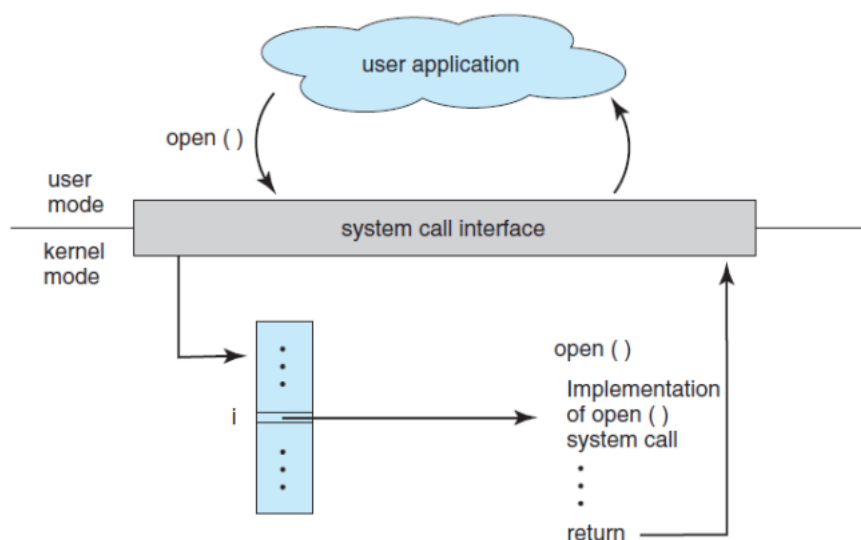
Mục lục

1	Thêm system call mới	2
2	Biên dịch lại Linux kernel	2
3	Hiện thực system call mới	4
4	Tạo wrapper cho system call	7
5	Validation	9
6	Question	10

1 Thêm system call mới

System call là một cơ chế mà các chương trình ứng dụng sử dụng yêu cầu các dịch vụ có sẵn của hệ điều hành. Hay nói cách khác, system call cung cấp giao diện cho các dịch vụ mà hệ điều hành hỗ trợ.

Ngoài ra, chúng ta còn có các API để chương trình thông thường sử dụng các system call. Ở hệ điều hành linux, API được hiện thực và cung cấp bằng ngôn ngữ C, tạo thành các thư viện như **glibc**. Thư viện này cung cấp các hàm đóng gói (wrapper functions) cho các system call. Các ứng dụng người dùng sẽ được cung cấp các API, thông qua các API này để sử dụng system call. Khi đó bộ xử lý sẽ chuyển từ **user mode** sang **kernel mode** để thực hiện phần hiện thực của system call, sau đó trả về kết quả cho ứng dụng.



Hình 1: Cơ chế gọi system call open của user.

2 Biên dịch lại Linux kernel

Nội dung của bài tập lớn này là thêm một system call mới giúp ứng dụng biết được các thông tin của vùng nhớ của một process (memory process), từ đó biết được cách tùy chỉnh và biên dịch kernel, cũng như hiểu về cơ chế của system call.

Bước đầu tiên, chúng ta chuẩn bị hệ điều hành, mã nguồn kernel và các công cụ liên quan. Trong bài báo cáo này, em dùng máy ảo VMware Workstation 15.5.2 và file ubuntu-16.04-desktop-amd64.iso tại ubuntu.com. Sau đó cài đặt các package gồm: *build-essential* (gồm các công cụ và thư viện cho việc biên dịch và cài đặt kernel như *g++*, *gcc*, *libc6-dev*, *make*, *dpkg-dev*,...), *openssl*, *libssl-dev* (giúp việc tải kernel bằng wget không bị lỗi SSL)¹

Trong hướng dẫn bài tập lớn còn có cài đặt gói **kernel-package**. Đây là gói công cụ tiện ích sử dụng cho việc build kernel Linux liên quan tới các package trên nhánh Debian (bao gồm cả Ubuntu). Công cụ này giúp chúng ta thuận tiện hơn, có thể tự động hóa quá trình build kernel,

¹<https://thuthuatmaytinh.vn/thu-thuat-windows/ssl-la-gi-cach-sua-loi-ssl-khi-vao-facebook-gmail/>

tạo ra file package cho Debian là .deb thông qua lệnh *make-kpkg*. Vì ở bài tập lớn này, sau khi chỉnh sửa và thêm system call mới, chúng ta build kernel bằng Makefile thông thường, do đó việc cài đặt gói này là không cần thiết, trừ trường hợp muốn tạo thành file package .deb để sử dụng về sau.

Tiếp tới ta tải mã nguồn kernel 4.4.21 theo địa chỉ trong hướng dẫn. Chúng ta phải sử dụng mã nguồn kernel khác từ server vì mã nguồn của kernel đang chạy không có sẵn, mà chỉ có các file nhị phân đã được biên dịch trong **/boot** để hoạt động, các file *headler.h* và một vài file liên quan để build **kernel_modules**. Do đó phải tải mã nguồn kernel từ nguồn khác để thực hiện chỉnh sửa và biên dịch lại từ đầu. Ở đây nên chọn server *kernel.org* vì đây là trang chủ chính thức chuyên cung cấp kernel cho hệ điều hành linux.

Tiếp theo ta cần tùy chỉnh lại phiên bản nội bộ của kernel theo mã số sinh viên. Bước này cần cài đặt gói **libncurses5-dev** để có giao diện cài đặt và tái sử dụng config của kernel hệ điều hành hiện tại.

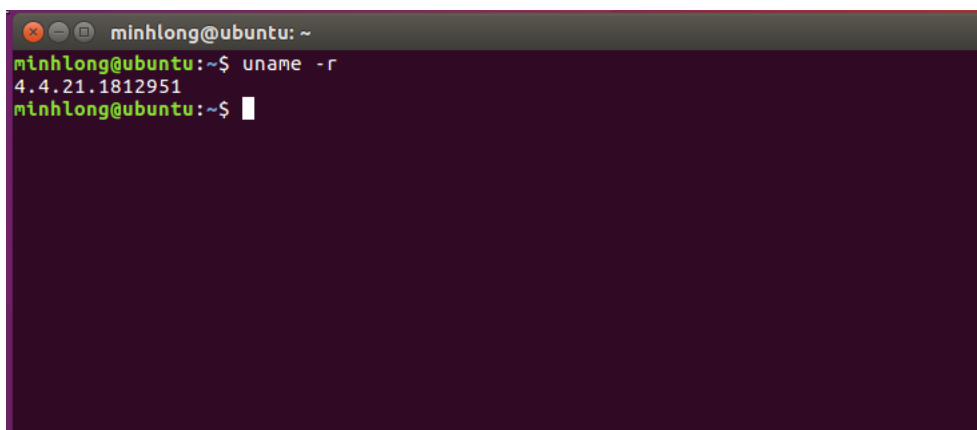
Sau khi copy file config tại **/boot/config-4.4.0-21-generic** tới **~/kernelbuild/.config** và tiến hành thay đổi version của kernel như hướng dẫn. Ta tiến hành biên dịch và cài đặt lại kernel. Để tăng tối đa tốc độ biên dịch và cài đặt, ta thực hiện lần lượt các lệnh sau:

```
$ make j 4  
$ make j 4 modules  
$ make j 4 modules_install  
$ make j 4 install
```

Trong đó:

- **make** để biên dịch và liên kết thành ảnh kernel, là tệp tin *vmlinux* duy nhất.
- **make modules** là lệnh để biên dịch modules, những tệp tin riêng lẻ tùy vào config, sau đó liên kết với kernel mới build.
- Với những CPU khác, ta thay số 4 bằng số luồng tối đa của CPU cho phép cùng lúc.

Sau khi hoàn tất, ta khởi động lại và kiểm tra phiên bản hệ điều hành bằng **uname -r**. Nếu phiên bản có chứa mssv tức là việc biên dịch và cài đặt đã thành công.



```
minhlong@ubuntu: ~  
minhlong@ubuntu:~$ uname -r  
4.4.21.1812951  
minhlong@ubuntu:~$
```

Hình 2: Biên dịch và cài đặt lại kernel.

3 Hiện thực system call mới

Như đã nói ở trước, mục tiêu của bài tập lớn này là hiện thực một system call mới để cho người sử dụng biết được trạng thái bộ nhớ của một process tại thời gian đang chạy trong hệ thống. Thông tin bộ nhớ của một process được trình bày theo cấu trúc:

```
struct prog_segs {  
    unsigned long mssv;  
    unsigned long start_code;  
    unsigned long end_code;  
    unsigned long start_data;  
    unsigned long end_data;  
    unsigned long start_heap;  
    unsigned long end_heap;  
    unsigned long start_stack;  
};
```

Hình 3: Cấu trúc thông tin bộ nhớ

Trong đó:

- **mssv** là mã số của sinh viên thực hiện (1819251)
- **start_code** và **end_code** là hai con trỏ trỏ đến byte đầu và cuối của phân vùng code segment.
- **start_data** và **end_data** là hai con trỏ trỏ đến byte đầu và cuối của phân vùng data segment.
- **start_heap** và **end_heap** là hai con trỏ trỏ đến byte đầu và cuối của phân vùng nhớ heap được cấp phát.
- **start_stack** là con trỏ trỏ đến byte đầu của phân vùng nhớ stack segment.

Sau khi chuẩn bị kernel và hệ điều hành xong, ta bắt đầu thêm system call mới hiện thực và kiểm tra system call. Thêm syscall vào bảng syscall bằng việc thêm dòng cuối sau vào file `arch/x86/entry/syscall_32.tbl`:

377	i386	procmem	sys_procmem	385,1	Bot
-----	------	---------	-------------	-------	-----

Hình 4: thêm vào syscall_32.tbl

Trong đó,

- **377** là index của system call mới, sau này sẽ sử dụng để gọi system call.
- **i386** chỉ ra rằng đây là system call 32bit.
- **procmem** là tên của syscall mới
- **sys_procmem** là entry point của syscall, tức điểm vào thực hiện của syscall hay tên của hàm hiện thực syscall.

Ta cũng thêm dòng sau vào cuối file **arch/x86/entry/syscall_64.tbl**:

546	x32	procmem	sys_procmem	373.1	Bot
-----	-----	---------	-------------	-------	-----

Hình 5: thêm vào syscall_64.tbl

Trong đó,

- **546** là index mới của file tùy thuộc số lượng system call.
- **x32** cho phép tận dụng lợi ích của tập lệnh x86-64 trong khi sử dụng con trỏ 32-bit, tránh chi phí hoạt động của con trỏ 64-bit.
- **procmem** là tên của syscall mới
- **sys_procmem** là entry point của syscall, tức điểm vào thực hiện của syscall hay tên của hàm hiện thực syscall.

Tiếp đó cần khai báo cấu trúc dữ liệu dùng để trả về kết quả của system call là struct **proc_segs** và hàm hiện thực **sys_procmem** ở cuối file **include/linux/syscalls.h** như sau:

```
struct proc_segs;  
asmlinkage long sys_procmem (int pid, struct proc_segs *info);
```

Hàm **sys_procmem** nhận vào 2 tham số : **pid** của process và con trỏ tới struct **proc_segs** để trả về dữ liệu.

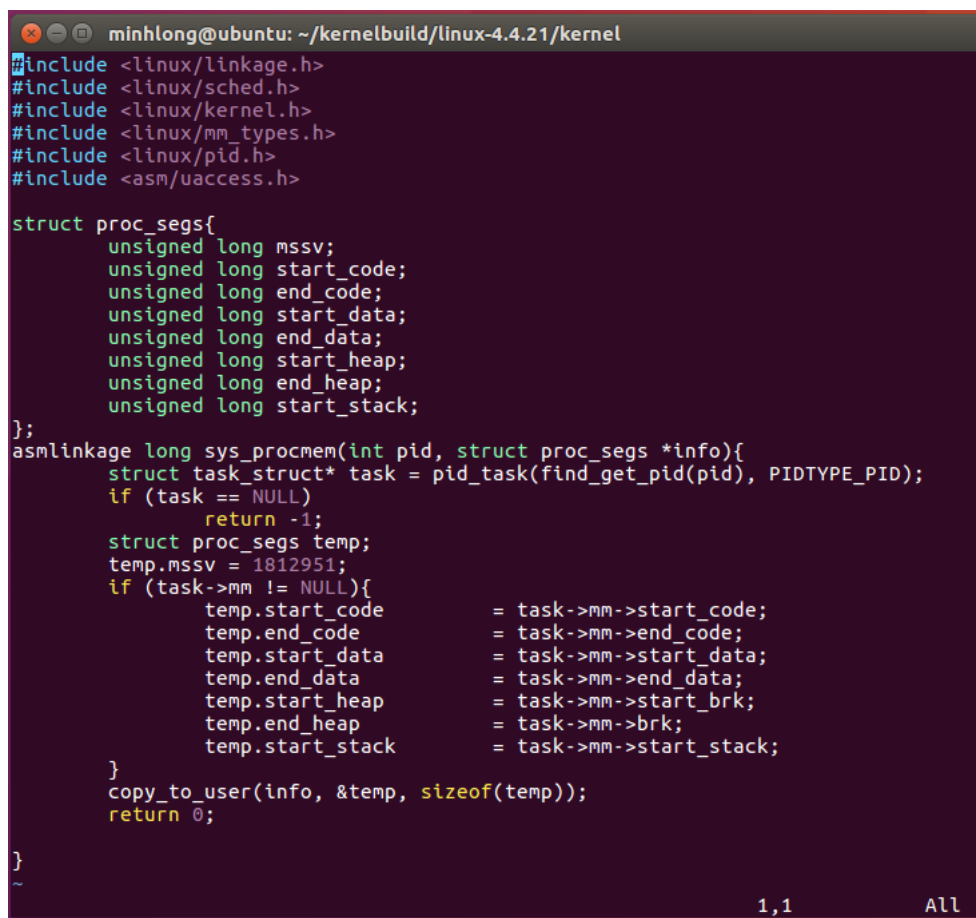
Hàm trả về giá trị kiểu long **asmlinkage** để báo hiệu với compiler rằng tham số của hàm lấy từ CPU Stack thay vì thanh ghi như thông thường.

Để kiểm tra lỗi và giảm thiểu số lần phải biên dịch kernel, ta sử dụng kernel module để chạy thử phần hiện thực system call. Ta sẽ lấy thông tin định thời của process cho trước bằng cách sử dụng thông tin trong 2 struct là **task_struct** và **sched_info**. Trước hết cần lấy được struct pid của một process với pid xác định bằng hàm **find_get_pid**:

```
struct pid *find_get_pid (int pid)  
struct task_struct *pid_task (struct pid *pid, enum pid_type)
```

Sau đó sử dụng hàm **pid_task** với **pid_type** bằng **PIDTYPE_PID** để có được **task_struct** của process theo pid đầu vào. Rồi lấy các thông tin như: **start_code**, **end_code**, **start_data**, **end_data**, **start_heap**, **end_heap**, **start_stack** thông qua **mm_struct** và trả về.

Ta đặt file hiện thực **sys_procmem.c** trong thư mục kernel. Kết quả ta được phần hiện thực hàm **sys_procmem.c** như sau:



Hình 6: Hiện thực file sys_procmem.c

Hàm `copy_to_user()`² dùng để copy dữ liệu từ kernel đến user để user có thể đọc, tránh trường hợp user không thể đọc những thông tin trong kernel

Cuối cùng ta cần chỉnh sửa file **Makefile** bằng cách thêm vào file **kernel/Makefile** dòng lệnh sau (ở đây tôi tìm đến vùng code có chứa các lệnh objy và thêm vào đó)



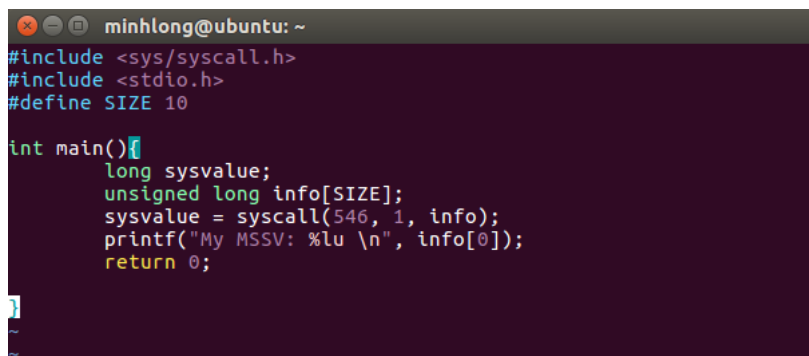
Hình 7: thêm dòng code vào /kernel/Makefile

Tiến hành biên dịch lại với các lệnh:

```
$ make j 4
$ make j 4 modules
$ make j 4 modules_install
$ make j 4 install
```

²<https://stackoverflow.com/questions/46302524/copy-to-user-undefined-in-linux-kernel-version-4-12-8>

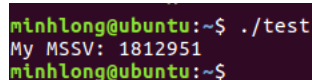
Sau khi chạy xong, ta khởi động lại và chạy đoạn code để kiểm tra thử:



```
minhlong@ubuntu: ~  
#include <sys/syscall.h>  
#include <stdio.h>  
#define SIZE 10  
  
int main(){  
    long sysvalue;  
    unsigned long info[SIZE];  
    sysvalue = syscall(546, 1, info);  
    printf("My MSSV: %lu \n", info[0]);  
    return 0;  
}
```

Hình 8: test.c

Nếu system call chạy đúng, mảng info sẽ chứa dữ liệu của struct **proc_segs**, trong đó phần tử đầu tiên là **unsigned long MSSV** sẽ được lưu vào **info[0]**. Vậy nếu chương trình in ra MSSV tương ứng với sinh viên hiện thực thì syscall mới đã hoạt động đúng.

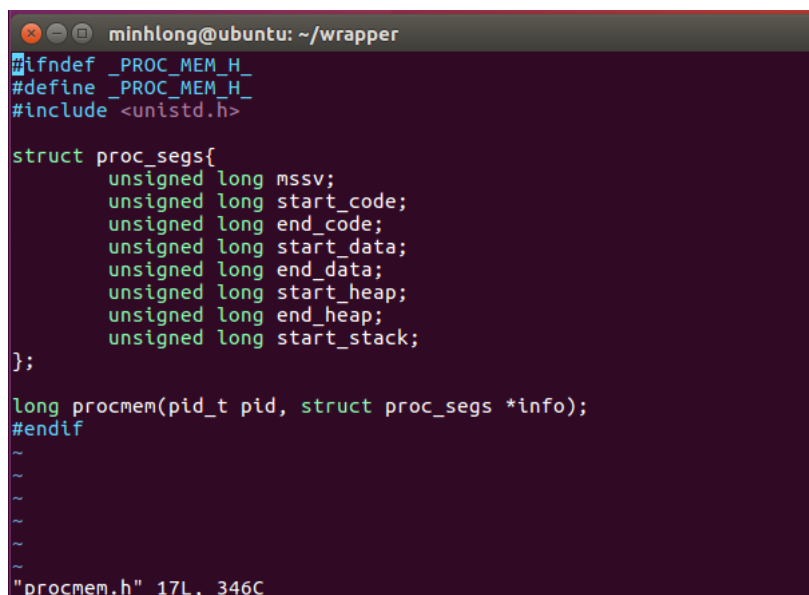


```
minhlong@ubuntu:~$ ./test  
My MSSV: 1812951  
minhlong@ubuntu:~$
```

Hình 9: Kết quả đoạn code kiểm tra

4 Tạo wrapper cho system call

Để việc sử dụng system call mới tạo thuận tiện hơn, ta cần tạo thêm **wrapper(API)** cho system call. Đầu tiên cần tạo file **procmem.h** và khai báo cấu trúc struct **proc_segs** và wrapper là hàm **procmem**. Sau đó copy file này vào **/usr/include**



```
minhlong@ubuntu: ~/wrapper
#ifndef _PROC_MEM_H_
#define _PROC_MEM_H_
#include <unistd.h>

struct proc_segs{
    unsigned long mssv;
    unsigned long start_code;
    unsigned long end_code;
    unsigned long start_data;
    unsigned long end_data;
    unsigned long start_heap;
    unsigned long end_heap;
    unsigned long start_stack;
};

long procmem(pid_t pid, struct proc_segs *info);
#endif
~
~
~
~
~
"procmem.h" 17L, 346C
```

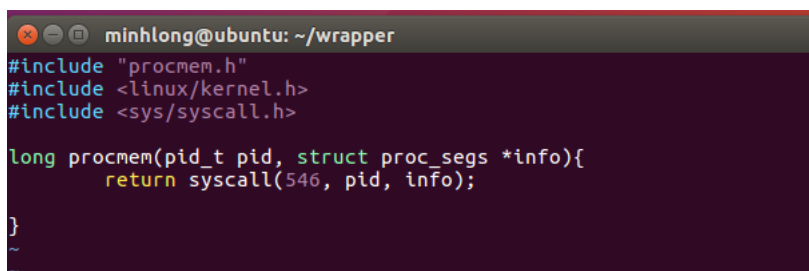
Hình 10: File procmem.h

Copy file vào thư mục **/usr/include** bằng lệnh:

```
$ sudo cp <path to procmem.h> /usr/include
```

Sở dĩ phải khai báo lại struct **proc_segs** là vì việc khai báo trong kernel là để biên dịch kernel. Sau khi biên dịch và cài đặt xong, cần phải khai báo lại và lưu tại **/usr/include** để khai báo thư viện **procmem.h** cho **gcc**. Khi đó mới có thể biên dịch **gcc** chương trình sử dụng **proc_segs**. Thư mục này cũng là nơi chứa nhiều header của các thư viện chuẩn C khác như **stdio.h**, **unistd.h**,... được sử dụng khi ta khai báo **include**.

khi copy file vào **/usr/include** cần sử dụng **sudo** vì chỉ người dùng root mới có quyền ghi trong thư mục này, người sử dụng nhóm root và người dùng khác đều không thể ghi mà chỉ có quyền đọc và thực thi. Do đó phải sử dụng kèm lệnh **sudo** để có quyền root khi copy file vào thư mục này. Tiếp theo hiện thực wrapper **procmem.c** như sau:



```
minhlong@ubuntu: ~/wrapper
#include "procmem.h"
#include <linux/kernel.h>
#include <sys/syscall.h>

long procmem(pid_t pid, struct proc_segs *info){
    return syscall(546, pid, info);
}
~
~
```

Hình 11: File procmem.c

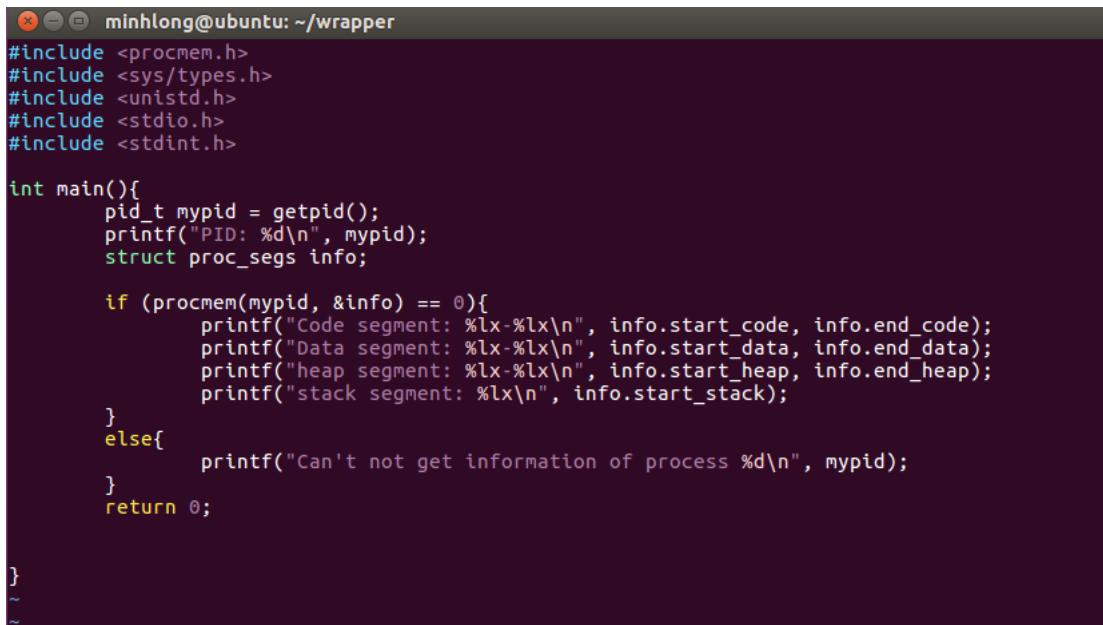
Biên dịch **procmem.c** để có được thư viện liên kết động bằng lệnh:

```
$ gcc -shared -fpic procmem.c -o libprocmem.so
```

Tùy chọn **-shared** giúp ta tạo ra mã đối tượng chia sẻ (**shared object**), mà có thể được liên kết với những object khác để thực thi dùng cho liên kết động. Từ đó nhiều chương trình có thể cùng sử dụng. Khi đó, lúc được load, vị trí code trên bộ nhớ phải độc lập (không phụ thuộc chương trình đang sử dụng), không phải chỉnh sửa địa chỉ nó được load vào bộ nhớ mà chương trình vẫn có thể tham chiếu và thực thi được. Do đó sử dụng **-fpic** hoặc **-fPIC** (PIC - Position Independence Code) là điều bắt buộc khi tạo ra shared object³

5 Validation

Tạo chương trình validation.c để kiểm tra thư viện và system call đang sử dụng.



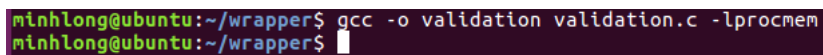
```
minhlong@ubuntu: ~/wrapper
#include <procmem.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>

int main(){
    pid_t mypid = getpid();
    printf("PID: %d\n", mypid);
    struct proc_segs info;

    if (procmem(mypid, &info) == 0){
        printf("Code segment: %lx-%lx\n", info.start_code, info.end_code);
        printf("Data segment: %lx-%lx\n", info.start_data, info.end_data);
        printf("heap segment: %lx-%lx\n", info.start_heap, info.end_heap);
        printf("stack segment: %lx\n", info.start_stack);
    }
    else{
        printf("Can't not get information of process %d\n", mypid);
    }
    return 0;
}
```

Hình 12: Chương trình kiểm tra validation.c

Compile với makefile thêm cờ **-lprocmem**



```
minhlong@ubuntu:~/wrapper$ gcc -o validation validation.c -lprocmem
minhlong@ubuntu:~/wrapper$
```

Hình 13: Compile chương trình validation.c

Và khi chạy chương trình, ta được kết quả

³<https://medium.com/meatandmachines/shared-dynamic-libraries-in-the-c-programming-language-8c2c03311756>

```
minhlong@ubuntu:~/wrapper$ ./validation
PID: 2721
Code segment: 400000-400ab4
Data segment: 600e00-601050
heap segment: 631000-652000
stack segment: 7ffe8d7340b0
minhlong@ubuntu:~/wrapper$
```

Hình 14: Kết quả khi chạy chương trình validation

6 Question

1. Why we need to install kernel-package?

Vì đây là gói công cụ tiện ích sử dụng cho việc build kernel linux liên quan tới các package nhánh Debian (bao gồm cả Ubuntu). Công cụ này giúp cho chúng ta thuận tiện hơn, có thể tự động hóa quá trình build kernel, tạo ra file package cho Debian là **.deb**

2. Why we have to use another kernel source from the server such as <http://www.kernel.org>, can we compile the original kernel (the local kernel on the running OS) directly?

Chúng ta phải tải mã nguồn kernel khác từ server vì mã nguồn của kernel đang chạy không có sẵn mà chỉ có các tệp tin nhị phân đã được biên dịch trong thư mục **/boot**, các tệp tin **header.h** và một số tệp tin cần thiết để build **kernel_modules**. Do đó phải tải mã nguồn kernel từ nguồn khác để chỉnh sửa và biên dịch lại từ đầu.

3. What is the meaning of these two stages, namely “make” and “make modules”?

- **make** Biên dịch và liên kết tạo thành ảnh kernel và tệp tin vmlinux duy nhất
- **make modules** Biên dịch modules, những tệp tin riêng lẻ tùy thuộc vào cấu hình, sau đó liên kết với kernel để build.

4. What is the meaning of other parts, i.e. i386, procmem, and sys procmem?

- **377** là index của system call mới, sau này sẽ sử dụng để gọi system call.
- **i386** chỉ ra rằng đây là system call 32bit.
- **procmem** là tên của syscall mới
- **sys_procmem** là entry point của syscall, tức điểm vào thực hiện của syscall hay tên của hàm hiện thực syscall.

5. What is the meaning of each line above? **struct** proc_segs: cấu trúc để lưu trữ thông tin vùng nhớ của quá trình.

Hàm **sys_procmem** nhận vào 2 tham số : **pid** của process và con trỏ tới struct **proc_segs** để trả về dữ liệu.

Hàm trả về giá trị kiểu long **asmlinkage** để báo hiệu với compiler rằng tham số của hàm lấy từ CPU Stack thay vì thanh ghi như thông thường.

6. Why this program could indicate whether our system works or not? Chương trình có thể kiểm tra hệ thống có hoạt động hay không bởi vì nếu lời gọi hệ thống thành công, thông tin của process sẽ được lưu vào info. Việc in ra info[0] giúp chúng ta kiểm tra xem chương trình đã chạy đúng không (nếu đúng sẽ in được mã số sinh viên).

7. Why we have to re-define `proc_segs` struct while we have already defined it inside the kernel? Mục đích định nghĩa lại cấu trúc `proc_segs` là vì việc khai báo trước đó trong kernel chỉ để biên dịch kernel. Muốn cho gcc biết và sử dụng khi biên dịch ta phải định nghĩa lại và lưu trong thư mục `/usr/include` là nơi chứa nhiều header của các thư viện chuẩn khác của C như: `stdio.h`, `stdlib.h`,...
8. Why root privilege (e.g. adding `sudo` before the `cp` command) is required to copy the header file to `/usr/include`? Vì chỉ người có quyền root mới có thể ghi trong thư mục này, người sử dụng nhóm root và người dùng khác đều không thể ghi mà chỉ có thể đọc và thực thi. Do đó cần phải kèm theo lệnh **sudo** để có quyền root khi sao chép tệp tin vào thư mục này.
9. Why we must put `-share` and `-fpic` option into gcc command?
 - Tùy chọn **-shared** giúp ta tạo ra mã đối tượng chia sẻ (**shared object**) có thể liên kết với những object khác để thực thi dùng cho liên kết động.
 - **-fpic** (PIC - Position Independence Code) là điều bắt buộc khi tạo ra shared object vì lúc được load vị trí code của thư viện phải trên bộ nhớ độc lập (không phụ thuộc vào chương trình đang sử dụng), không phải chỉnh sửa địa chỉ được load vào bộ nhớ mà chương trình vẫn có thể tham chiếu và thực thi được.