

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



## OPERATING SYSTEMS

---

### Assignment 1

# System Call

---

GVHD: Vu Van Thong  
SV: Nguyen Gia Huy - 1810173

TP. HỒ CHÍ MINH, THÁNG 5/2020



## Mục lục

<b>1 Chuẩn bị</b>	<b>2</b>
<b>2 Thêm system call mới</b>	<b>4</b>
<b>3 Hiện thực system call</b>	<b>6</b>
<b>4 Quá trình compile và cài đặt</b>	<b>8</b>
4.1 Compile . . . . .	8
4.2 Cài đặt . . . . .	8
<b>5 Tạo API cho system call</b>	<b>9</b>
<b>6 Kết luận</b>	<b>12</b>



Báo cáo này sẽ hướng dẫn cách thêm một system call mới vào kernel. Các bước hướng dẫn trong báo cáo được thực hiện trên máy ảo Ubuntu 14.04 **Trusty Tahr** và kernel được sử dụng là Linux kernel version 4.4.21.

## 1 Chuẩn bị

**Cài đặt các package cần thiết:** Ta chạy các lệnh sau trên terminal để cài đặt các package cần thiết :

```
$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install kernel-package
```

**QUESTION: Why we need to install kernel-package?**

**ANSWER:** Bởi vì **kernel-package** giúp tự động hóa các bước cần thiết để compile và cài đặt một kernel tùy chỉnh. Ngoài ra **kernel-package** còn đem lại nhiều lợi ích khác trong quá trình tạo kernel, ta có thể xem thêm bằng lệnh `man kernel-package`

**Tải kernel source:** Ta sẽ tạo một thư mục riêng cho kernel :

```
$ mkdir ~/kernelbuild
```

Tiếp theo ta sẽ tải kernel source từ <https://www.kernel.org>. Ở đây chúng ta sẽ chọn kernel version 4.4.21

```
$ cd ~/kernelbuild
$ wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.21.tar.xz
```

**QUESTION: 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?**

**ANSWER:** Chúng ta không nên thay đổi các file liên quan đến kernel gốc vì nếu kernel tùy chỉnh dẫn đến hệ thống gặp lỗi hoặc không thể khởi động thì ta có thể sử dụng kernel hiện hành để khởi động vào hệ thống. Thường thì chúng ta không thể compile trực tiếp kernel gốc của OS vì khi cài OS thì ta đã cài kernel đã được compile sẵn (binary executable), để có thể compile kernel ta cần có source code từ một nguồn khác.

**Giải nén kernel source:** ta chạy tiếp lệnh sau trong thư mục **kernelbuild**:

```
$ tar -xvJf linux-4.4.21.tar.xz
```

Sau khi giải nén ta sẽ có thư mục **linux-4.4.21** chứa mọi thứ của kernel, đây sẽ là nơi làm việc chính của ta

```
$ cd linux-4.4.21
```

**Configuration:** Việc tiếp theo ta cần làm là tinh chỉnh kernel để phản ánh đúng đặc điểm máy tính của mình. Các điều chỉnh của kernel được đặt trong file `.config` và ta sẽ mượn file này của kernel đang chạy trên máy ảo. File này thường được đặt ở `/boot/` nên ta chỉ cần chạy lệnh sau:

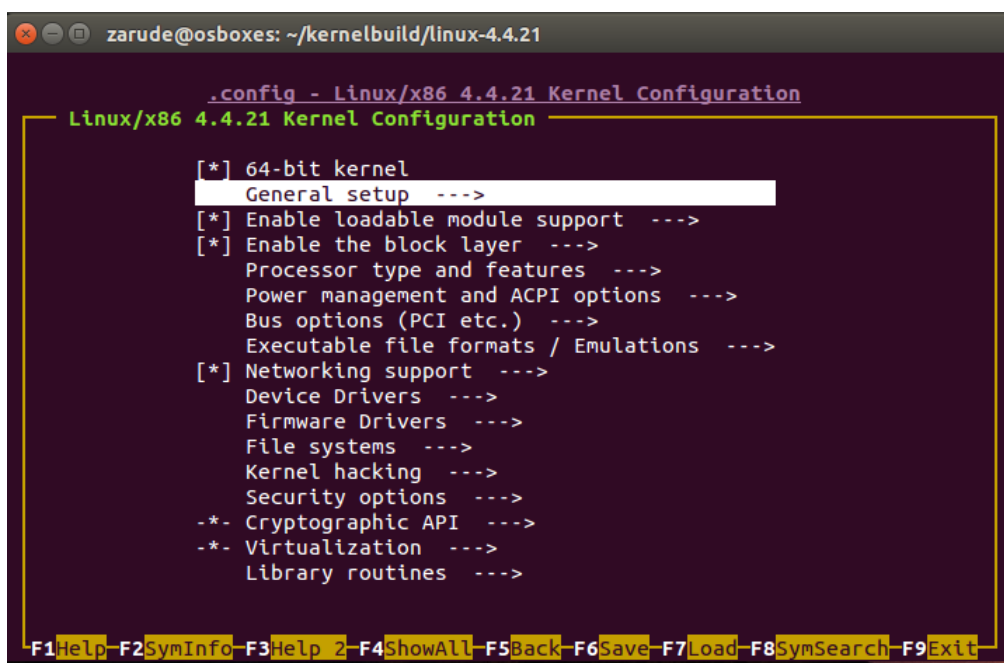
```
$ cp /boot/config-4.4.0-142-generic ~/kernelbuild/.config
```

*Lưu ý:* 4.4.0-142-generic là phiên bản kernel được cài sẵn trong máy ảo của người hướng dẫn. Để xem thông tin kernel trên máy của bạn, hãy chạy lệnh `uname -r`

**Quan trọng:** Đừng quên thay đổi tên phiên bản kernel trong mục General Setup. Bởi vì ta đang sử dụng lại file `.config` của kernel hiện tại, nên nếu bạn bỏ qua bước này thì có khả năng một trong các nhân hiện tại sẽ bị nhầm lẫn và ghi đè lên. Để thay đổi file `config`, ta có thể chỉnh sửa thông qua terminal interface với package `libncurses5-dev`:

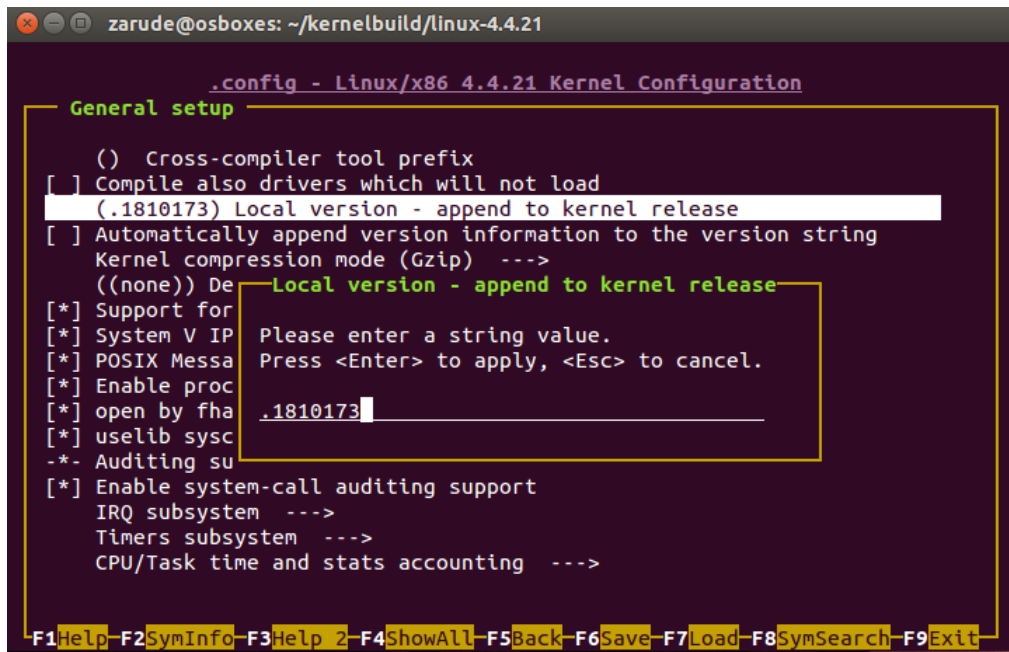
```
$ sudo apt-get install libncurses5-dev
```

Sau đó chạy lệnh `$ make menuconfig` hoặc `$ make nconfig` để mở cửa sổ Kernel Configuration.



Hình 1: Cửa sổ Kernel Configuration

Chọn mục General setup, truy cập đến dòng `"(-ARCH) Local version - append to kernel release"` và nhập vào MSSV. Sau đó bấm F6 để lưu và F9 để thoát ra.



Hình 2: Thêm MSSV vào "Local version"

*Lưu ý:* Trong quá trình compile, bạn có thể gặp lỗi gây ra vì thiếu package openssl. Bạn cần cài các package này bằng các chạy lệnh sau:

```
$ sudo apt-get install openssl libssl-dev
```

## 2 Thêm system call mới

Trước khi hiện thực system call mới, chúng ta phải thêm nó vào kernel trước. Danh sách các system call cho kiến trúc x86 được đặt ở thư mục `arch/x86/entry/syscalls`. Tại đây ta có 2 file gồm `syscall_32.tbl` và `syscall_64.tbl` lần lượt là danh sách system call cho bộ xử lý x86 32-bit và x86 64-bit. Để đảm bảo system call của ta hoạt động tốt với mọi bộ xử lý x86, ta sẽ thêm vào cả hai file.

Ta mở file `syscall_32.tbl` bằng vim:

```
$ vim arch/x86/entry/syscalls/syscall_32.tbl
```

và thêm dòng sau vào cuối file và lưu lại:

```
[number] i386 procmem sys_procmem
```

Ta sẽ chọn `[number]` là số lớn nhất trong danh sách cộng thêm 1 và các cột thông tin phải cách nhau bằng phím TAB.

```
368 i386 getpeername sys_getpeername
369 i386 sendto sys_sendto
370 i386 sendmsg sys_sendmsg compat_sys_sendmsg
371 i386 recvfrom sys_recvfrom compat_sys_recvfrom
372 i386 recvmsg sys_recvmsg compat_sys_recvmsg
373 i386 shutdown sys_shutdown
374 i386 userfaultfd sys_userfaultfd
375 i386 membarrier sys_membarrier
376 i386 mlock2 sys_mlock2
377 i386 procmem sys_procmem
```

Hình 3: Thêm system call trong file syscall\_32.tbl

**QUESTION: What is the meaning of other parts, i.e. i386, procmem, and sys\_procmem?**

**ANSWER:** i386 là ABI (Application Binary Interface) tương thích với kiến trúc x86 32-bit. procmem là tên của system call sẽ được sử dụng bởi người dùng ở "user-land". sys\_procmem là tên hàm mà kernel dùng để chạy system call - đây chính là tên hàm hiện thực system call của mình.

Tương tự như vậy đối với file syscall\_64.tbl:

[number] x32 procmem sys\_procmem

```
541 x32 setsockopt compat_sys_setsockopt
542 x32 getsockopt compat_sys_getsockopt
543 x32 io_setup compat_sys_io_setup
544 x32 io_submit compat_sys_io_submit
545 x32 execveat stub_x32_execveat
546 x32 procmem sys_procmem
```

Hình 4: Thêm system call trong file syscall\_64.tbl

Tiếp theo ta sẽ thêm định nghĩa system call của mình. Để làm như vậy ta sẽ thêm những thông tin cần thiết vào file header của kernel. Mở file include/linux/syscalls.h :

```
$ vim include/linux/syscalls.h
```

và thêm các dòng sau vào cuối file trước dòng #endif:

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

```
3
2 struct proc_segs;
1
894 asmlinkage long sys_procmem(int pid, struct proc_segs *info)
1
2 #endif
syscalls.h [+]
```

Hình 5: Thêm prototype vào /include/linux/syscalls.h

**QUESTION: What is the meaning of each line above?**

**ANSWER:** Mỗi dòng trên chính là phần khai báo prototype của struct `proc_segs` và hàm `sys_procmem()`. Tất cả các system call của hệ thống phải được đánh dấu bằng tag `asmlinkage`.

Chúng ta đã thêm những thông tin cần thiết về system call của mình vào kernel, bây giờ tiến tới phần hiện thực thôi.

### 3 Hiện thực system call

Trong directory `kernel/`, ta tạo file với tên `sys_procmem.c` :

```
$ vim kernel/sys_procmem.c
```

và thêm phần hiện thực system call như sau:

```
#include <linux/linkage.h>
#include <linux/sched.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;
};

asmlinkage long sys_procmem(int pid, struct proc_segs *info) {
    struct task_struct *proc;
    struct mm_struct *mmaps;

    read_lock(&tasklist_lock);
    proc = find_task_by_vpid(pid);
    if (proc)
        get_task_struct(proc);
    read_unlock(&tasklist_lock);
    if (!proc)
        return -1;

    mmaps = proc->active_mm;

    info->mssv      = 1810173;
    info->start_code = mmaps->start_code;
    info->end_code   = mmaps->end_code;
    info->start_data  = mmaps->start_data;
    info->end_data    = mmaps->end_data;
    info->start_heap  = mmaps->start_brk;
    info->end_heap    = mmaps->brk;
    info->start_stack = mmaps->start_stack;

    put_task_struct(proc);
    return 0;
}
```

Trước khi tiến hành bước compile ta cần phải thêm system call của mình vào Makefile trong thư mục kernel :

```
$ vim kernel/Makefile
```

và thêm dòng sau :

```
obj-y += sys_procmem.o
```



## 4 Quá trình compile và cài đặt

### 4.1 Compile

Đầu tiên ta sẽ chạy lệnh **make** để compile kernel và tạo ra **vmlinuz**. Ta có thể chạy song song bằng cách dùng tag "**-j np**" với **np** là số lượng bộ xử lý dùng để chạy. Ở đây tôi sử dụng **np = 4** để tận dụng 4 nhân đã cấp cho máy ảo :

```
$ make -j 4
```

Tiếp theo là build các modules của kernel :

```
$ make -j 4 modules
```

**QUESTION: What is the meaning of these two stages, namely “make” and “make modules”?**

**ANSWER:** Bước **make** để compile kernel và tạo ra **vmlinuz**. **vmlinuz** chính là nhân sẽ được giải nén và load vào bộ nhớ bởi GRUB hoặc boot loader của hệ thống. Bước **make modules** để build ra các module có thể gắn vào kernel.

Quá trình compile sẽ tốn một khoảng thời gian khá lâu. Sau khi máy đã compile thành công ta sẽ tiến hành cài đặt kernel.

### 4.2 Cài đặt

Đầu tiên là cài đặt các module:

```
$ sudo make -j 4 modules_install
```

Sau đó là cài đặt kernel:

```
$ sudo make -j 4 install
```

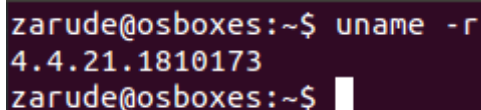
**Kiểm tra thành quả:** Sau khi cài đặt xong, chúng ta sẽ tiến hành khởi động lại

```
$ sudo reboot
```

Sau khi khởi động lại và đăng nhập vào Ubuntu, ta chạy câu lệnh sau:

```
$ uname -r
```

Nếu MSSV xuất hiện trong chuỗi được xuất ra nghĩa là ta đã compile và cài đặt thành công.



```
zarude@osboxes:~$ uname -r
4.4.21.1810173
zarude@osboxes:~$
```

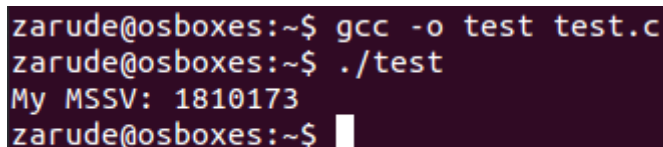
Hình 6: MSSV xuất hiện trong version của kernel

Tiếp theo ta có thể kiểm tra system call của mình bằng một chương trình C có tên `test.c` như sau :

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

Vì máy ảo của tôi đang chạy là Ubuntu 64-bit nên tôi sẽ lấy `SYS_NUM` là số của system call `procmem` trong file `syscall_64.tbl` (546). Hãy nhớ thay giá trị của `SYS_NUM` trong chương trình trên thành số trong file `syscall_32.tbl` hoặc `syscall_64.tbl` tùy thuộc vào Ubuntu 32 hoặc 64-bit trên máy của bạn.

Compile và chạy chương trình trên ta sẽ thấy được MSSV của mình.



```
zarude@osboxes:~$ gcc -o test test.c
zarude@osboxes:~$ ./test
My MSSV: 1810173
zarude@osboxes:~$
```

Hình 7: Compile và chạy chương trình test

**QUESTION:** Why this program could indicate whether our system works or not?

**ANSWER:** Vì system call `sys_procmem()` sẽ lưu MSSV vào `info[0]` (vị trí unsigned long đầu tiên của mảng `info`) nên nếu chương trình test có thể in ra đúng MSSV thì nghĩa là system call của ta đã được thêm vào kernel thành công và đã được sử dụng.

## 5 Tạo API cho system call

Tiếp theo ta sẽ tạo một hàm wrapper để tiện lợi trong việc sử dụng system call của ta. Đầu tiên ta sẽ tạo một thư mục mới để lưu trữ source code của hàm wrapper :

```
$ mkdir procmem  
$ cd procmem
```

Tiếp theo ta sẽ tạo file header trong thư mục này với tên `procmem.h` chứa prototype của hàm và struct `proc_segs`:

```
#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 // _PROC_MEM_H_
```

**QUESTION:** Why we have to re-define `proc_segs` struct while we have already defined it inside the kernel?

**ANSWER:** Ta phải định nghĩa lại struct `proc_segs` ở đây để người dùng có thể sử dụng khi include header này. Việc ta định nghĩa struct này trong kernel chỉ có hàm `sys_procmem` trong kernel sử dụng, hàm `procmem()` được sử dụng ở "user space" nên không thể hay biết.

Sau đó ta tạo file `procmem.c` có nội dung như sau :

```
#include "procmem.h"  
#include <linux/kernel.h>  
#include <sys/syscall.h>  
#define SYS_NUM 546  
  
long procmem(pid_t pid, struct proc_segs *info) {  
    return syscall(SYS_NUM, pid, info);  
}
```

Nhớ thay đổi giá trị `SYS_NUM` phù hợp trên máy bạn.

Tiếp theo ta sẽ copy header file của mình vào thư mục header của hệ thống :

```
$ sudo cp procmem.h /usr/include
```

**QUESTION: Why root privilege (e.g. adding sudo before the cp command) is required to copy the header file to /usr/include?**

**ANSWER:** Bởi vì chỉ có root user mới có quyền chỉnh sửa (write) trên thư mục directory /usr/, còn các user bình thường chỉ có quyền xem (read) nên ta cần quyền root (root privilege) để copy file vào /usr/include

Ta cần compile source code của mình thành một shared object để cho phép người dùng có thể tích hợp system call của ta vào các ứng dụng của họ. Để làm như vậy, ta sẽ chạy câu lệnh sau:

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

**QUESTION: Why we must put -shared and -fpic option into gcc command?**

**ANSWER:** Flag -shared giúp tạo shared object để có thể link với các object khác lúc tạo file thực thi. Flag -fpic tạo ra code không phụ thuộc vị trí (position-independent code - PIC) phù hợp cho sử dụng vào các thư viện dùng chung bởi vì vị trí của các thư viện trong bộ nhớ sẽ thay đổi theo các chương trình khác nhau.

Sau khi compile hoàn tất, ta copy file output vào /usr/lib:

```
$ sudo cp libprocmem.so /usr/lib
```

Bước cuối cùng của ta là kiểm tra lại thành quả toàn bộ công việc đã làm. Ta viết một chương trình test\_api.c như sau:

```
#include <procmem.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.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("Start stack: %lx\n", info.start_stack);
    } else {
        printf("Cannot get information from the process %d\n", mypid);
    }
    // If necessary, uncomment the following line to make this program run
    // long enough so that we could check out its maps file
    // sleep(100);
}
```



và compile nó kèm với tag "-lprocmem" :

```
$ gcc -o test_api test_api.c -lprocmem
```

và xem kết quả :

```
zarude@osboxes:~/os-study/assignment1$ gcc -o test_api test_api.c -lprocmem
zarude@osboxes:~/os-study/assignment1$ ./test_api
PID: 3058
Code segment: 400000-400a2c
Data segment: 600e00-601050
Heap segment: 1a06000-1a06000
Start stack: 7ffea77945d0
zarude@osboxes:~/os-study/assignment1$
```

Hình 8: Compile và chạy chương trình test\_api

## 6 Kết luận

Qua bài báo cáo này, ta đã thêm một system call mới vào kernel để giúp ứng dụng biết memory layout của một process nào đó. Đồng thời ta biết thêm được cách hiệu chỉnh, compile và cài đặt kernel Linux của riêng mình trên máy tính