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

HỆ ĐIỀU HÀNH

Báo cáo bài tập lớn 1
System call

Giảng viên hướng dẫn: Phạm Trung Kiên
Sinh viên thực hiện: Lê Công Linh
MSSV: 1711948

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 05/2019

Nội dung:

1. Quá trình thêm system call

1.1. Chuẩn bị

Cài đặt máy ảo VMWare với phiên bản Ubuntu 18.04.2. Sau đó unpdate và cài đặt các core package như hướng dẫn:

Metapackage:

$ sudo apt-get update

$ sudo apt-get install build-essential

Kernel package:

$ sudo apt-get install kernel-package

QUESTION: Tại sao phải cài đặt kernel-package?

ANSWER:

- Cài đặt kernel-package giúp cho việc cá nhân hóa dễ dàng hơn, nó cũng giúp cho việc biên dịch kernel thuận tiện hơn bằng việc một chuỗi các công việc theo trình tự các bước thực hiện.
- Kernel-package cho phép giữ nhiều phiên bản của kernel-image trên thiết bị mà không gây rối.
- Tự động di chuyển các thư mục tới vị trí thích hợp cũng như tự động lựa chọn các cài đặt phù hợp với từng kiến trúc
- Các kernel-module được liên kết với nhau, nên có thể biên dịch dễ dàng, đảm bảo tính tương thích.
- Công cụ dùm để build và quản lý Kernel Linux
- Đảm bảo các tệp cài đặt cùng với kernel-image luôn đi cùng nhau
- Cho phép việc biên dịch kernel trên nhiều kiến trúc con
- kmod: công cụ quản lí kernel module
- binutils: trình biên dịch GNU, trình liên kết và các tiện ích nhị phân.
- xz-utls: tiện ích nén định dạng xz
- cpio: một chương trình để quản lí tài liệu lưu trữ các tệp

Tiếp theo tạo thư mục có tên kernelbuild để build kernel. Tải kernel về thư mục kernelbuild:

$ mkdir ~/kernelbuild

$ cd ~/kernelbuild

$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux -5.0.5.tar.xz

Sau khi hoàn tất, trong thư mục kernelbuild, ta giải nén kernel tarball

$ tar –xvJf linux-5.0.5.tar.xz

QUESTION: Tại sao phải sử dụng kernel source khác tải từ server như là http://www.kernel.org, chúng ta có thể biên dịch trực tiếp kernel có trong hệ điều hành được không (the local kernel on the running OS)?

ANSWER: Khi sử dụng một kernel source từ server, có thể dễ dàng tủy chỉnh, xem xét các thư mục, cũng như cài đặt các bản vá cần thiết, dễ dàng phát hiện bug và sửa lỗi

Không thể biên dịch một kernel trực tiếp có trong OS. Bởi vì nó đã được biên dịch thành file thực thi, không phải là code C.

1.2. Cấu hình

Bởi vì quá trình thiết lập lại file configuration rất phức tạp, chúng ta sẽ mượn nội dung của file cấu hình có sẵn của kernel hiện đang được máy ảo sử dụng. File này thường nằm trong /boot/, vì vậy chúng ta chỉ việc sao chép nó vào thư mục mã nguồn linux-5.0.5 (vừa mới giải nén) với đuôi .config

$ cp /boot/config-$(uname –r) ~kernelbuild/linux-5.0.5/.config

Để chỉnh sửa tập cấu hình thông qua giao diện terminal, trước hết ta phải cài đặt gói *libncurses*5*-dev:*

$ sudo apt-get install fakeroot ncurses-dev xz-utils bc flex libelf-dev bison

Sau đó mở file configuration bằng lệnh:

$ make menuconfig

Để thay đổi phiên bản kernel, vào “General Setup”, truy cập dòng “(-ARCH) Local version – append to kernel release". Sau đó gõ dấu chấm “.” theo sau là MSSV:

.1711948

Nhấn F6 để lưu và F9 để thoát.

1.3. Thêm system call-sys_get_proc_info

Bước 1:

Thêm system call mới hiện thực vào danh sách system call của hệ thống. Trong thư mục *arch/x86/entry/syscalls*, tìm file *syscall_64.tbl*, thêm dòng sau vào cuối file:

"*549 common get_proc_info __x64_sys_get_proc_info*"

QUESTION: Ý nghĩa của các trường dữ liệu vừa mới thêm vào bảng system call (*549, common, get_proc_info, __x64_sys_get_proc_info*)

ANSWER:

- *549* là chỉ số định danh (index) của syscall. Tất cả các system call được xác định bởi một số duy nhất. Để gọi một syscall, ta gọi thông qua index chứ không gọi thông qua tên.
- *common*: là ABI (Application Binary Interface), môi trường tương tác giữa 2 chương trình thành phần nhị phân, cụ thể giữa người dùm với thư viện hoặc đệ điều hành, ở mức mã máy (phổ biến là x86 32 bit, 64x 64 bit)
- *get_proc_info*: là tên của system call
- *__x64_sys_get_proc_info*: là điểm nhập (entry point), là tên của hàm xử lý syscall.Đây là điểm bắt đầu chạy syscall (là cái lệnh đầu tiên được gọi khi chương trình được thực thi, SYSCALL_DEFINE2 yêu cầu phải có kiến trúc vào cú pháp gọi entry poin (vd: __x64_sys cú pháp bắt buộc)).

Bước 2:

Thêm syscall sắp hiện thực vào syscall header file:

$ cd ~/kernelbuild/linux-5.0.5/include/linux/

$ gedit syscall.h

Thêm vào trước *#endif* trong file *syscall.h* nội dung sau:

```
struct proc_info;
struct procinfos;
asmlinkage long sys_get_proc_info(pid_t pid, struct procinfos * info);
```

QUESTION: Ý nghĩa của các dòng trên?

ANSWER:

Các dòng trên khai báo prototype của struct *procinfos*, *proc_info* và syscall *get_proc_info* cùng kiểu trả về (*long*) và tham số truyền vào, những gì sẽ được hiện thực trong file *get_proc_info.c*.

struct procinfos, proc_info: Cấu trúc được sử dụng trong syscall

asmlinkage: Thẻ được gán cho system call, thông báo rằng chỉ nhận tham số từ stack

Bước 3:

Trong *kernelbuild/linux-5.0.5/*, ta tạo một thư mục *get_proc_info.*

Chạy các lệnh sau:

$ cd ~/kernelbuild/linux-5.0.5/

$ mkdir get_proc_info

$ cd get_proc_info/

Trong thư mục *get_proc_info*, tạo một file *sys_get_proc_info.c.*

$ gedit sys_get_proc_info.c

Đây sẽ là file mà ta cần hoàn thiện để hiện thực system call

```
#include <linux/kernel.h>
#include <unistd.h>

struct proc_info { //info about a single process
        pid_t pid; //pid of the process
        char name[16]; //file name of the program executed
};
struct procinfos { //info about processes we need
        long studentID;
        struct proc_info proc; //process with pid or current process
        struct proc_info parent_proc; //parent process
        struct proc_info oldest_child_proc; //oldest child process
};


SYSCALL_DEFINE2(get_proc_info, pid_t, pid, struct procinfos*, info)
{
        //Todo: implement system call
}
```

Bước 4:

Tạo Makefile cho source file trong thư mục *kernelbuild/linux-5.0.5/get_proc_info/*:

$ touch Makefile

$ echo "obj-y := get_proc_info.o"

Bước 5:

Thêm get_proc_info/ vào kernel Makefile:

$ cd ~/kernelbuild/linux-5.0.5/

$ gedit Makefile

Tìm dòng với nội dung:

```
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

Thêm *get_proc_info/* vào cuối dòng:

```
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ get_proc_info/
```

2. Hiện thực system call

Ta vào lại thư mục *kernelbuild/linux-5.0.5/ get_proc_info/*

$ cd ~/kernelbuild/linux-5.0.5/get_proc_info/

$ getdit sys_get_proc_info.c

Hiện thực system call với nội dung như sau:

```
#include <linux/kernel.h>
#include <unistd.h>
#include <linux/syscalls.h>
#include <asm/current.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <asm/uaccess.h>
#include <linux/errno.h>
#include <linux/string.h>

struct proc_info { //info about a single process
        pid_t pid; //pid of the process
        char name[16]; //file name of the program executed
};
struct procinfos { //info about processes we need
        long studentID;
        struct proc_info proc; //process with pid or current process
        struct proc_info parent_proc; //parent process
        struct proc_info oldest_child_proc; //oldest child process
};


SYSCALL_DEFINE2(get_proc_info, pid_t, pid, struct procinfos*, info)
{
        struct task_struct      *proc = NULL,
                                *parent_proc = NULL,
                                *oldest_child_proc = NULL;
        struct procinfos * kinfo;
        unsigned long cop;
        kinfo= kmalloc(sizeof(struct procinfos), GFP_KERNEL);
        if(kinfo ==NULL)
                return EINVAL;
        printk(KERN_INFO "1711948");
        kinfo->studentID=1711948;
        if (pid<-1 || info == NULL) {
                return EINVAL;
        }
```

```
if(pid==0)
{
        proc=find_task_by_vpid(1)->parent;
        kinfo->proc.pid=proc->pid;
        kinfo->parent_proc.pid=-1;
        strcpy(kinfo->proc.name,proc->comm);
        strcpy(kinfo->parent_proc.name,"No name");
        if(list_empty_careful(&proc->children))
        {
                kinfo->oldest_child_proc.pid=-1;
                strcpy(kinfo->oldest_child_proc.name,"No name");
        }
        else
        {
                oldest_child_proc=list_first_entry(&proc->children, struct task_struct, sibling); //Return the oldest child
                kinfo->oldest_child_proc.pid=oldest_child_proc->pid;
                strcpy(kinfo->oldest_child_proc.name,oldest_child_proc->comm);
        }
        cop= copy_to_user(info, kinfo, sizeof(struct procinfos));
        kfree(kinfo); // Deallocate kernel memory
        return 0;
}

if(pid == -1)
        proc=current;
else
        proc=find_task_by_vpid(pid);

if(proc ==NULL)
        return EINVAL;

parent_proc=proc->parent;

if(parent_proc ==NULL)
        return EINVAL;
kinfo->proc.pid=proc->pid;
kinfo->parent_proc.pid=parent_proc->pid;

strcpy(kinfo->proc.name,proc->comm);
strcpy(kinfo->parent_proc.name,parent_proc->comm);

if(list_empty_careful(&proc->children))
{
        kinfo->oldest_child_proc.pid=-1;
        strcpy(kinfo->oldest_child_proc.name,"No name");

}
else
{
        oldest_child_proc=list_first_entry(&proc->children, struct task_struct, sibling); //Return the oldest child
        kinfo->oldest_child_proc.pid=oldest_child_proc->pid;
        strcpy(kinfo->oldest_child_proc.name,oldest_child_proc->comm);
}

// Copy data from kernel space to user space//
cop= copy_to_user(info, kinfo, sizeof(struct procinfos));
if(cop!=0)
        return EINVAL;

kfree(kinfo); // Deallocate kernel memory
return 0;

}
```

Thẻ asmlinkage thông báo cho compiler biết rằng hàm sẽ truyền và tìm tham số trong ngăn xếp (stack method). Khi một syscall được gọi từ user space đến kernel space, tham số từ user space stack cũng được truyền theo. Nhưng 2 vùng này độc lập nên kernel không biết đến sự tồn tại của chúng, dẫn đến việc truyền tham số không đúng yêu cầu. Ta sẽ sử dụng:

- Macro SYSCALL_DEFINE2 có 2 tham số được truyền vào: *pid* có kiểu *pid_t*; *info* là con trỏ struct *proinfos**; thông số đầu tiên chính là tên của system call.

- Hàm copy_to_user(*to,*from,size): Copy dữ liệu có kích thước size bytes chứa trong *from ở kernel space đến con trỏ *to trong user space.

Trong bài tập lớn này, system call mà ta hiện thực cho phép người dùng xác định thông tin về quá trình cha và quá trình con già nhất của một quá trình.

Thông tin của quá trình được biểu diễn thông qua struct *procinfos* sau:

Với *proc_info* được định nghĩa:

```
struct proc_info { //info about a single process
    pid_t pid; //pid of the process
    char name[16]; //file name of the program executed
}
```

```
struct procinfos { //info about processes we need
    long studentID;
    struct proc_info proc; //process with pid or current process
    struct proc_info parent_proc; //parent process
    struct proc_info oldest_child_proc; //oldest child process
};
```

- *procinfos* chứa thông tin về 3 process:
  - *proc* : là process hiện tại hoặc process với ***pid***
  - *parent_proc* : cha của process *proc*
  - *oldest_child_proc* : là con già nhất của process *proc*.
- *proc_info* chứa thông tin của mỗi process, gồm:
  - *pid* : ID của process
  - *name* : tên của process đang thực thi

Để gọi syscall *get_proc_info*, người dùng phải cung cấp *pid* của process. Nếu pid truyền vào:

- Nhỏ hơn -1: trả về lỗi **EINVAL**
- -1: process hiện tại
- 0: process đặc biệt, là process duy nhất không có cha
- pid hợp lệ: sử dụng hàm find_task_by_vpid(pid): tìm process có id = pid
- Nếu không có cha: trả về thông tin process cha pid= -1; name = No name
- Nếu không có con: trả về thông tin process con pid= -1; name = No name

Nếu tìm thấy process có *pid*, nó sẽ lấy thông tin của process, gán vào tham số **info* và trả về 0. Nếu thất bại sẽ trả về **EINVAL**.

3. Quá trình biên dịch và cài đặt

3.1. Biên dịch

Đầu tiên, chạy lệnh "*make*" để biên dịch kernel và tạo *vmlinuz.* Sẽ tốn rất nhiều thời gian để thục thi lệnh "*$ make*". Thay vào đó, ta có thể chạy đồng thời bằng cách sử dụng "-j np", với *np* là số quá trình bạn chạy cho lệnh này.

$ make –j 8

Build loadable kernel module:

$ make –j 8 modules

QUESTION: Ý nghĩa của 2 bước***, "make"*** và ***"make modules"*** là gì? Chúng tạo ra các gì và mục đích được tạo ra?

ANSWER:

- ***make*** tạo ra một bản nén kernel-image, đây là một file đơn có tên *vmlinuz* được sử dụng bởi boot loader, chứa system call vừa được thêm vào.
- ***make modules*** biên dịch lại những driver, module toàn bộ (hoặc được chọn lọc) trong kernel sources.

3.2. Cài đặt

Trước tiên cài đặt module:

$ sudo make –j 8 modules_install

Sau đó, cài đặt kernel:

$ sudo make –j 8 install

Sau khi cài đặt hoàn tất, reboot lại máy ảo:

$ sudo reboot

Cuối cùng, đăng nhập vào máy ảo và chạy lệnh sau:

$ uname -r

Chuỗi xuất ra có chứa MSSV (5.0.5.1711948), quá trình compile và install thành công.

3.3. Kiểm tra

Sau khi boot vào kernel mới, ta tạo một file C để kiểm tra xem syscall mới đã được tích hợp vào kernel hay chưa.

Ta tạo file *testing.c :*

$ mkdir testing

$ cd testing/

$ gedit testing.c

Soạn nội dung như sau:

```c
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 200
int main(){
        long sys_return_value;
        unsigned long info[SIZE];
        sys_return_value = syscall(549, -1, &info);
        printf("My student ID: %lu\n", info[0]);
        return 0;
}
```

Trong đó:

- sys_return_value: giá trị trả về
- 549: index của system call ta vừa tạo
- -1: Lấy thông tin của process hiện tại (có thể lấy pid khác)

Biên dịch và chạy thử:

$ gcc testing.c –o testing

$ ./testing

Kết quả xuất ra màn hình:

```
conglinh@ubuntu ~/testing> ./testing
My student ID: 1711948
```

QUESTION: Tại sao chương trình trên có thể cho ta biết, syscall mới của mình có hoạt động hay không?

ANSWER:

Ta gọi trực tiếp thông qua chỉ số định danh của syscall. Syscall hoạt động mà không có lỗi thì việc gán MSSV vào biến **info* mới thành công và in ra được màn hình thông qua lệnh *printf("My student ID: %lu\n", info[0]).*

4. Tạo API cho system call

Mặc dù syscall *get_proc_info* hoạt động tốt, ta vẫn phải gọi nó thông qua chỉ số định danh. Điều này gây ra sự bất tiện cho các lập trình viên khác khi muốn sử dụng syscall của chúng ta. Vì vậy ta nên tạo một file C wrapper cho dễ sử dụng.

Chúng ta vào home, tạo một thư mục khác để chứa file wrapper.

$ mkdir wrapper

$ cd wrapper/

Tạo một header file có tên *get_proc_info.h* chứa prototype của *wrapper* và định nghĩa struct *proc_info* và *procinfos*.

```
#ifndef _GET_PROC_INFO_H_
#define _GET_PROC_INFO_H_
#include <unistd.h>
#include <unistd.h>

        struct proc_info {
                pid_t pid;
                char name[16];
        };
        struct procinfos {
                long studentID;
                struct proc_info proc;
                struct proc_info parent_proc;
                struct proc_info oldest_child_proc;
        };


        long get_proc_info(pid_t pid, struct procinfos * info);
#endif // _GET_PROC_INFO_H_
```

QUESTION: Tại sao phải định nghĩa lại 2 cấu trúc *procinfos* và *proc_info* trong khi đã định nghĩa bên trong kernel?

ANSWER:

- Ta chỉ mới định nghĩa 2 struct trên trong kernel space, còn hàm wrapper ta viết ở user space (2 vùng này tách biệt nhau) nên phải định nghĩa lại.
- Ta phải định nghĩa lại bởi vì ta mới chỉ khai báo prototype của struct *procinfos* và *proc_info* trong */arch/include/linux/syscalls.h*, ta chỉ có thể khai báo con trỏ đến struct mà không sử dụng được các members của nó, bởi vì size hay members của struct *procinfos* và *proc_info* chưa được biết bởi trình biên dịch.

Sau đó, ta tạo file *get_proc_info.c* chứa code wrapper với nội dung như sau:

```
#include "get_proc_info.h"
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/errno.h>

long get_proc_info(pid_t pid, struct procinfos * info) {
        if(pid<-1|| info==NULL)
                return  EINVAL;

        long sys_return_value;
        sys_return_value = syscall(549,pid,info);

        return sys_return_value;
}
```

Đầu tiên, copy file *get_proc_info.h* được tạo ở thư mục wapper vào thư mục */usr/include/* thông qua lệnh:

$ cd wrapper/

$ sudo cp get_proc_info.h /usr/include/

QUESTION: Tại sao đặc quyền của root (thêm sudo vào trước command) lại cần thiết cho việc copy file header vào */usr/include/*?

ANSWER:

Vì thư mục */usr/include/* thuộc quyền sở hữu của root nên muốn copy file vào thư mục này phải thêm sudo vào trước command để copy bằng đặc quyền của root (và phải nhập đúng mật khẩu của root).

Sau đó biên dịch file *get_proc_info.c* được tạo trong phần đóng gói:

$ gcc –shared –fpic get_proc_info.c –o libget_proc_info.so

QUESTION: Tại sao phải thêm *–shared* và *–fpic* vào gcc command?

ANSWER:

- *-shared* dung để tạo thư viện động chia sẻ (*.so), chia sẻ chúng để dùng chung giúp giảm kích thước các file thực thi và liên kết các file đối tượng khác tạo thành file thực thi.
- *-fpic* dung để tạo mã độc lập với vị trí (PIC) phù hợp để sử dụng thư viện dùng chung (tương thích với *–shared*). Mã này truy cập tất cả các vị trí không đổi thông qua bảng offset toàn cục.

Khi biên dịch thành công, copy file *libget_proc_info.so* vào thư mục */usr/lib/*

$ sudo cp libget_proc_info.so /usr/lib/

Bước cuối cùng, viết hàm *validation.c* để kiểm tra tất cả công việc.

$ cd validation/

$ gedit validation.c

```c
#include <get_proc_info.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <linux/errno.h>

int main(int argc, char **argv[]) {
        pid_t mypid;
        if(argc<2)
                mypid = -1;
        else
                mypid =atoi(argv[1]);
        printf("PID: %d\n", mypid);
        if (mypid<-1) {
                return EINVAL;
        }
        struct procinfos info;
        if(get_proc_info(mypid, &info) == 0) {
        // TODO: print all information in struct procinfos info
                printf("StudentID: %li\n",info.studentID);
                printf("ProcessID: %i\t\t\tname: %s\n", info.proc.pid,info.proc.name);

                printf("ParentID: %i\t\t\tname: %s\n", info.parent_proc.pid,info.parent_proc.name);

                printf("ChildID: %i\t\t\tname: %s\n", info.oldest_child_proc.pid,info.oldest_child_proc.name);
        }
        else {
                printf("Cannot get information from the process %d\n", mypid);
                sleep(100);
        }
        // If necessary , uncomment the following line to make this program run
        // long enough so that we could check out its dependence
        return 0;
}
```

Biên dịch với lựa chọn *–lget_proc_info*:

$ gcc validation.c –lget_proc_info –o validation

Xem kết quả :

- *$ ./validation 2275*

```
conglinh@ubuntu ~> ./validation 2275
PID: 2275
StudentID: 1711948
ProcessID: 2275                 name: fish
ParentID: 1805                  name: gnome-terminal-
ChildID: 2295                   name: htop
```

- *$ ./validation -1*

```
conglinh@ubuntu ~> ./validation -1
PID: -1
StudentID: 1711948
ProcessID: 2256                 name: validation
ParentID: 1815                  name: fish
ChildID: -1                     name: No name
```

- *$ ./validation 0*

```
conglinh@ubuntu ~> ./validation 0
PID: 0
StudentID: 1711948
ProcessID: 0                     name: swapper/0
ParentID: -1                     name: No name
ChildID: 1                       name: systemd
```

- *$ ./validation 1287*

```
conglinh@ubuntu ~/validation> ./validation 1287
PID: 1287
StudentID: 1711948
ProcessID: 1287                 name: colord
ParentID: 1                     name: systemd
ChildID: -1                     name: No name
```

5. Tài liệu tham khảo

- Implementing a system call in Linux Kernel 4.7.1
  https://medium.com/@ssreehari/implementing-a-system-call-in-linux-kernel-4-7-1-6f98250a8c38
  Last access: 30/04/2019
- Adding a New System Call
  https://www.kernel.org/doc/html/v4.12/process/adding-syscalls.html
  Last access: 30/04/2019
- Compile and run kernel modules
  http://www.tldp.org/LDP/lkmpg/2.6/html/x181.html
  Last access: 30/04/2019
- https://www.programering.com/a/MDN4kjNwATI.html
  Last access: 30/04/2019
- https://packages.ubuntu.com/bionic/kernel-package?fbclid=IwAR1X4X9J0cKVKNJkBRBrCCHkPT0_YXBLaOgte1_xZx5-ZNqPRZ5asNy-xuI
  Last access: 30/04/2019