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

HỆ ĐIỀU HÀNH - OPERATING SYSTEM (CO2017)

Bài tập lớn

Assignment 01: System Call

GVHD:	Nguyễn Quang Hùng
	La Hoàng Lộc
SV thực hiện:	Phạm Đức Duy Anh – 1810814

Tp. Hồ Chí Minh, Tháng 5/2020

Mục lục

1 Introduction

1.1 System call

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions. As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of the system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, include the parameters that are passed to each function and the return values the programmer can expect. The functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

1.2 Analyze Assignment Request

The assignment provides a progress of compiling Linux kernel. After that, the most important part is to implement a system call inside the kernel. Overview steps of doing assignment.

- Install package need to do assignment.
- Download kernel file tarball
- Extract kernel file
- Configure kernel
- Build and install the configured kernel.
- Trim the kernel
- Implement system call
- Test system call
- Wrap system call
- Validate system call

2 Compiling Linux Kernel

2.1 Preparation

Set up Ubuntu 18.04: Install Ubuntu 18.04 operating system parallel with Window operating system.
Install the core packages: Get Ubuntu's toolchain (gcc, make, and so forth) by installing the build-essential metapackage.

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

Install kernel-package

```
$ sudo apt-get install kernel-package
```

```
QUESTION: Why we need to install kernel-package
Answer: There are a lot of advantages when installing kernel-package. If we
compile kernel manually, there are many steps by steps that we need to follow
strictly. Kernel-package include almost everything we need. Therefore, time
spending on installing kernel will cut down a lot because it just need some
commands to set up kernel with kernel-package. Besides, it allow us to:
- Convenience: kernel-package was written to take all the required steps to
compile kernel manually. This is especially important to novices: make-kpkg
takes all the steps required to compile a kernel, and installation of kernels
is a snap. - Multiple images support: Keep multiple versions of kernel-image
on device without disturbing, make sure that installation files along with
kernel-image of its always goes together.
- Multiple flavors of same kernel version: Facilitate keeping multiple versions
of the same kernel on device.
- Automatically move folders to the appropriate location and select settings
suitable for each architecture.
- Kernel-models are linked, so it can be easily compiled and guaranteed
compatible.
- System can manage installed kernel versions.
- Create packages with header files, source files as well as .deb files that
are managed by the system manager (by Package Management System).
- Compile kernel on many child architectures.
- Compile kernel on other computers.
```

Create a kernel compilation directory: Create folder name "kernelbuild" in home directory to store our kernel.

```
$ mkdir ~/kernelbuild
```

Download the kernel compilation directory: First we move to kernelbuild directory which we had made. Download the kernel source from http://www.kernel.org. The file tarball (tar.xz) for chosen kernel version is added to kernelbuild directory. (I choose kernel version 5.0.5)

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

```
QUESTION: Why we have to use another kernel source form the server such as
http://www.kernel.org, can we compile the original kernel (the local kernel on
the running OS) directly
Answer: When using a kernel source, it is easy to edit, review directories and
set up necessary patch.
It is unable to compile a kernel directly from itself. Meanwhile, it is
possible to compile a kernel version similar to the current version, the
source code can be obtained by the following command: apt-get source
linux-image-$(uname -r)
```

Unpack the kernel sources: In kernelbuild directory, unpack the kernel tarball. It created linux-5.0.5 directory inside kernelbuild directory.

```
$ tar -xvJf linux-5.0.5.tar.xz
```

2.2 Configuration

Copy the content of configuration file of the existing kernel currently used by the ubuntu 18.04 located in /boot/ to the source code directory.

```
$ cp /boot/config-$(uname -r) ~/kernelbuild/.config
```

Then, we must install some essential packages to edit configure file through terminal interface.

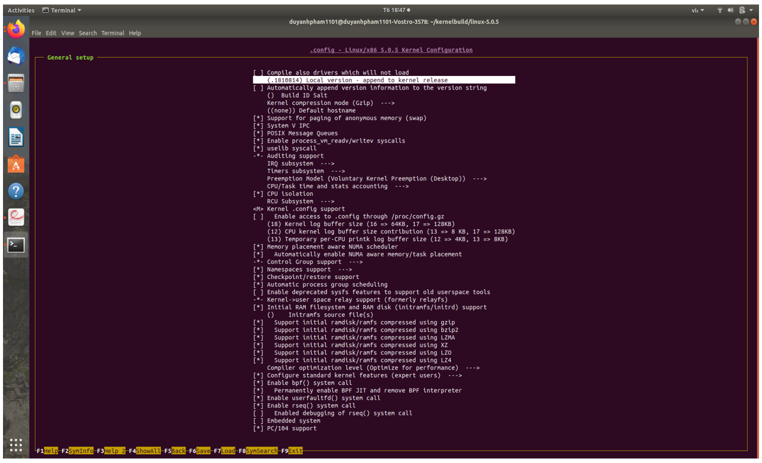
```
$ sudo apt-get install fakeroot ncurses-dev xz-utils bc flex libelf-dev bison
```

Then we open Kernel Configuration using command

```
$ make nconfig
```

We go to General Setup option, access to line "(-ARCH) Local version - append to kernel release, then add "." by student ID followed.

```
.1810814
```



Press F6 to save change and then press F9 to exit.
We install openssl package to avoid error caused by missing package by running the following command.

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

2.3 Building the configured kernel

We compile the kernel and create vmlinuz, we run following command. Using 8 processes help us run this stage in parallel to save time.

```
$ make -j 8
```

After that, we build the loadable kernel modules. We also use 8 processes to help running this stage in parallel.

```
$ make -j 8 modules
```

QUESTION: What is the meaning of these two stages, namely "make" and "make modules"? What are created and what for?
Answer:
- make -j 8: creates a compressed version of kernel-image, used by boot loader. The command make connect the libraries to the source then create linkage between necessary files, set it up for the final stage and create binary files. This stage is running parallel with 8 processes to help saving time.
- make -j 8 modules: compiles modules and leaves compiled binary files in the build directory. . This stage is running parallel with 8 processes to help saving time.

2.4 Installing the new kernel

First install the modules:

```
$ sudo make -j 8 modules_install
```

Then install kernel itself:

```
$ sudo make -j 8 install
```

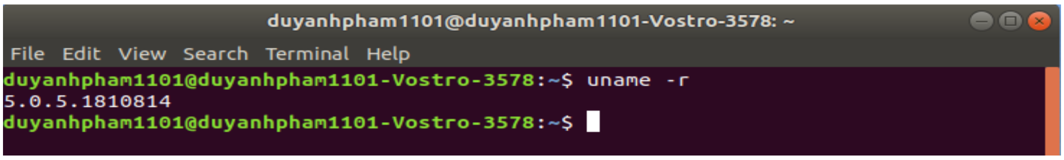
After installing the new kernel by steps described above. Reboot the computer by running command following:

```
$ sudo reboot
```

After logging into the computer again, run the following command.

```
$ uname -r
```

Check the output of the command.

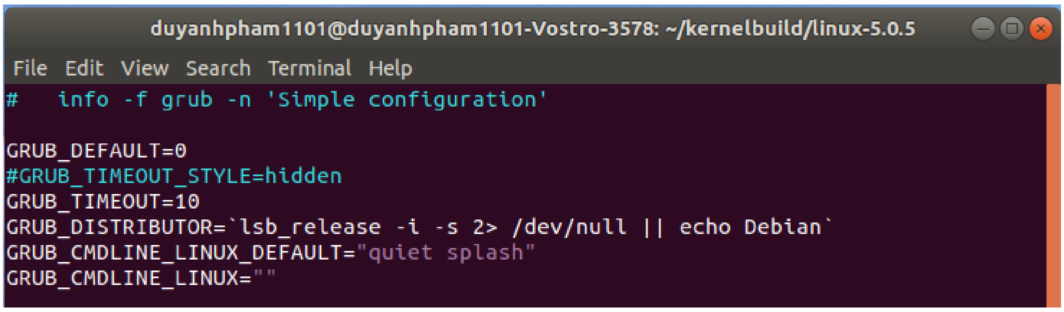


3 Trim the kernel

Display GRUB by running following command:

```
$ sudo vim /etc/default/grub
```

And command out GRUB_TIMEOUT_STYLE=hidden



After that to finish update GRUB by running following command.

```
$ sudo update-grub
```

Finally, trim the kernel by following command

```
$ make localmodconfig
```

Using the make target "localmodconfig" saves time and effort when creating a configuration for a custom Linux kernel by using the distribution kernel's configuration file as a starting point – these typically have most drivers enabled for compilation as modules, which can take a long time and is unnecessary, as often only a few of them are relevant for the specific system.

4 System call

4.1 Implementation

First we create a new directory name "get_proc_info" to store system call implementation by enter.

```
$ mkdir get_proc_info
```

Then we move to new directory which has been created and create sys_get_proc_info.c to implement system call.

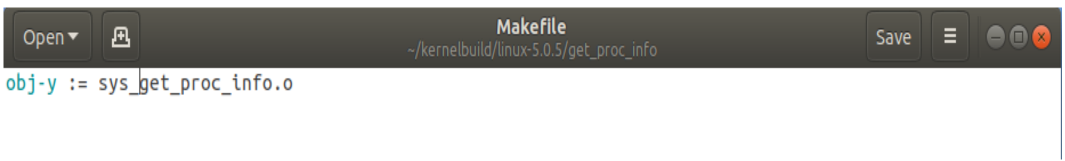
```
$ cd get_proc_info
$ touch sys_get_proc_info.c
```

We edit file sys_get_proc_info.c by entering.

```
$ gedit sys_get_proc_info.c
```

Code explaination:
We define 2 structs: struct proc_info to store information of a single process and struct procinfos to store information of processes we need. The get_proc_info function receive 2 parameters: pid_t pid is the pid of the process we need to get information and struct procinfos to store data. Specifically, each process has its own task_struct which help us to get process's name, pid, parent process, children process. In order to get the information of the process has given pid, we use the marco for_each_process(struct task_struct defined in linux/sched.h). When this marco is being run, it check every single running process pid. If the pid is matched with the pid passed to function, we will store data in struct procinfos structure.
One problem exists here is this function is using kernel space address, but struct procinfos we passed to function is allocated in user space address. If we access directly this space, it may cause segmentation fault which can make process is killed. To avoid this problem, we define a struct procinfos in kernel space address, store data in this and then use macro copy_to_user to copy data from kernel space to user space.
The specifical implementation was done in sys_get_proc_info.c.
After that, we create a Makefile for compiling source file.

```
$ touch Makefile
$ gedit Makefile
```



Then we add directory get_proc_info to the kernel Makefile.

```
$ cd ..
$ gedit Makefile
```

We find the following line (using Ctrl + F in gedit) and add directory get_proc_info/ to the end of this line:

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

Then we add the new system call to the system call table

```
$ cd arch/x86/entry/syscalls/
$ gedit syscall_64.tbl
```

```
330     common  pkey_alloc              __x64_sys_pkey_alloc
331     common  pkey_free               __x64_sys_pkey_free
332     common  statx                   __x64_sys_statx
333     common  io_pgetevents           __x64_sys_io_pgetevents
334     common  rseq                    __x64_sys_rseq
335     common  get_proc_info           __x64_sys_get_proc_info

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation. The __x32_compat_sys stubs are created
# on-the-fly for compat_sys_*() compatibility system calls if X86_X32
# is defined.
#
512     x32     rt_sigaction            __x32_compat_sys_rt_sigaction
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
```

```
QUESTION: What is the meaning of each line above?
Answer:
335: the number of the entry of the new system call.
common: system call has a common implementation for both of the ABIs for x86
and 64 bit instruction.
get_proc_info: using to declare as user.
__x64_sys_get_proc_info: name of function of the new system call inside kernel.
```

Then we add new system call to the system call header file.

```
$ cd
$ cd ~/kernelbuild/linux-5.0.5/include/linux
$ gedit syscalls.h
```

Finally recompile the kernel and restart the system to apply the new kernel.

4.2 Testing

We create "test" directory to check if the system call has been integrated into the kernel or not. We running following commands.

```
$ cd
$ cd ~/kernelbuild/linux-5.0.5
$ mkdir test
$ cd test
$ touch testing.c
$ gedit testing.c
```

testing.c
~/kernelbuild/linux-5.0.5/test

```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(335, -1, &info);
        printf("My student ID: %lu\n", info[0]);
        if(sys_return_value == 0) printf("System call successed!\n");
        else printf("System call failed!\n");
        return 0;
}
```

If the test program print the student ID. The system call is invoked successfully.

```
duyanhpham1101@duyanhpham1101-Vostro-3578: ~/kernelbuild/linux-5.0.5/test
File Edit View Search Terminal Help
duyanhpham1101@duyanhpham1101-Vostro-3578:~$ cd kernelbuild/linux-5.0.5/test
duyanhpham1101@duyanhpham1101-Vostro-3578:~/kernelbuild/linux-5.0.5/test$ gcc test
ing.c
duyanhpham1101@duyanhpham1101-Vostro-3578:~/kernelbuild/linux-5.0.5/test$ ls
a.out  testing.c
duyanhpham1101@duyanhpham1101-Vostro-3578:~/kernelbuild/linux-5.0.5/test$ ./a.out
My student ID: 1810814
System call successed!
duyanhpham1101@duyanhpham1101-Vostro-3578:~/kernelbuild/linux-5.0.5/test$
```

```
QUESTION: Why this program could indicate whether our system call works or not
Answer: In order to know if the new system call works or not, it depends on the
value returned of the function
sys_return_value = syscall([syscall_number_in_tbl], pid, info);
The function will invoke system call with specifier number, passing 2
parameters pid for finding process need to find information and info to
store information. If the system call implement succesfully, the value of
sys_return_value will be returned 0. Otherwise, it will be returned EINVAL.
If success, we can print the student ID exactly
```

4.3 Wrapper

When system call function is called by its number declared in system_64.tbl. The parameter get from the function is an array data type. All of these things make it difficult for user to call system call function. Thus, we need to make a wrapper for the system call function.
Firstly, we need to redefine proc_info and procinfos ordinarily as we define in kernel. Then, we implement wrapper for the new system call function. The goal of wrapper is to call system call in an easier way for users. Detail of implementation in get_proc_info.h and get_proc_info.c.
We running these following commands.

```
$ cd
$ cd ~/kernelbuild/linux-5.0.5
$ mkdir wrapper
$ cd wrapper
$ touch get_proc_info.h
$ touch get_proc_info.c
$ gedit get_proc_info.h
$ gedit get_proc_info.c
```

QUESTION: Why we have to redefine procinfos and proc_info struct while we have already defined it inside the kernel ?
Answer: When passing parameters to the system call function, we will get the results stored information of a process as format of the structures defined in kernel. Because of this, we have to redefine the structures for users to know and use the results exactly and effectively.

4.4 Validate

We will install the wrapper to our computer, make the header file visible to GCC by the following command:

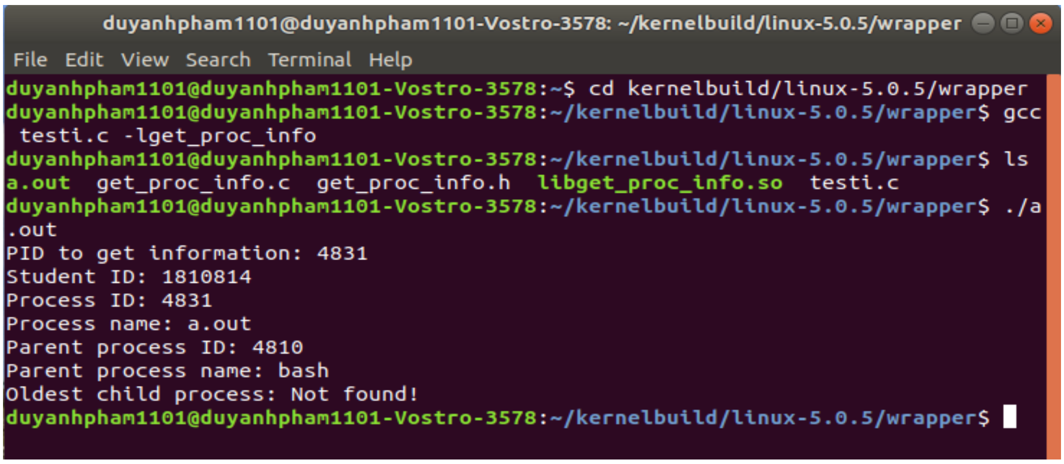
```
$ sudo cp /kernelbuild/linux-5.0.5/wrapper/get_proc_info.h /usr/include
$ gcc -shared -fpic get_proc_info.c -o libget_proc_info.so
```

Then we copy the output file to /usr/lib.

```
$ sudo cp /kernelbuild/linux-5.0.5/wrapper/libget_proc_info /usr/include
```

QUESTION: Why we must putshare and -fpic option into gcc command?.
Answer: We must put these things like this because:
fpic(Position Independent Code): it notifies compiler to create code snippets which is independent of location. These code snippets can be loaded into any virtual memory address.
shared: create objects which is used for shared libraries.

Finally, we have to check all of our work by writing a program and compile it with -lget_proc_info option. Detail of the program in testi.c. The result that we get is captured in the following picture.



Tài liệu

[1] https://www.geeksforgeeks.org/the-linux-kernel/

[2] https://en.wikipedia.org/wiki/System_call/

[3] https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h

[4] https://www.fsl.cs.sunysb.edu/kernel-api/re256.html