HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Assignment #1 - System Call

Pham Trung Kien

April 17, 2021

Goal: This assignment helps students to understand steps of modifying, compiling and installing Linux kernel.

Content: In detail, student will add a new system call which helps applications to know the parent and a child of a given process. This task requires student to understand system call invocation mechanism as well as steps of compiling and installing Linux kernel.

Result: After this assignment, student should know how to modify Linux kernel and deploy their own kernel on a given machine.

Contents

1 Introduction

1.1 System calls

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 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, including 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.

EXAMPLE OF STANDARD API As an example of a standard API, consider the `open()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page. A description of this API appears below:

```
$ man open

#include <unistd.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

A program that uses the `open()` function must include the unistd.h header file, as this file defines `int` data types (among other things). The parameters passed to `open()` are as follows.

- `const *path` - The relative or absolute path to the file that is to be opened.
- `int oflags` - A bitwise 'or' separated list of values that determine the method in which the file is to be opened.
- `mode_t mode` - A bitwise 'or' separated list of values that determine the permissions of the file if it is created.

The file descriptor returned is always the smallest integer greater than zero that is still available. If a negative value is returned, then there was an error opening the file.

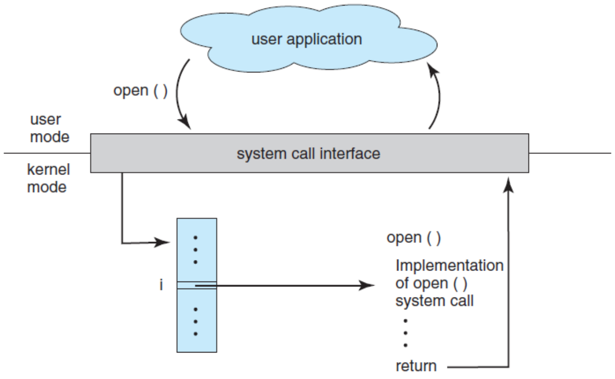


Figure 1: The handling of a user application invoking the open() system call.

The relationship between an API, the system-call interface, and the operating system is shown in Figure 1, which illustrates how the operating system handles a user application invoking the `open()` system call.

1.2 Requirement and Marking

As Figure 2 shown, this is the diagram of doing the assignment. Student will practice the progress of compiling Linux kernel. After that, the most important part is to implement a system call inside the kernel. The assignment is divided into multiple stages and score is marked by each stage as Figure 2.

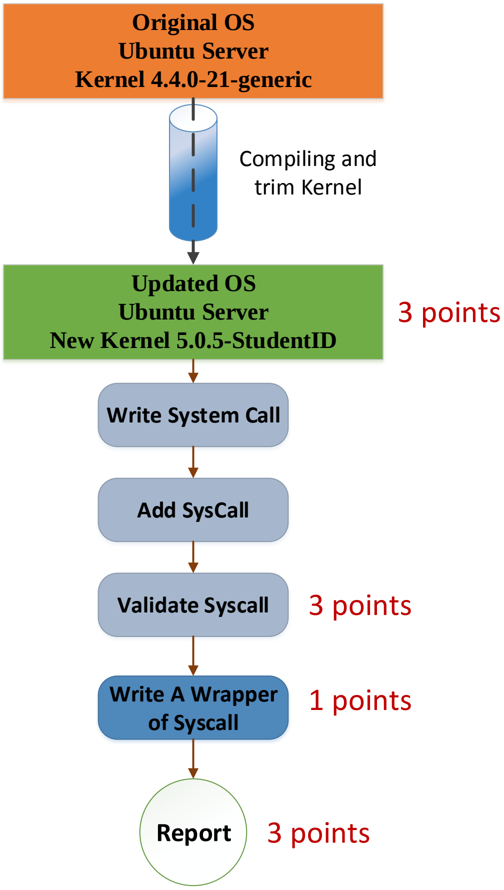


Figure 2: Diagram of implementing the assignment.

2 Compiling Linux Kernel

Compiling custom kernel has its own advantages and disadvantages. However, new Linux user/admin find it difficult to compile Linux kernel. Compiling kernel needs to understand few things and then just type couple of commands. This section guides you basic steps to compile the Linux kernel, but you need to consider the purposes of these commands for summarizing a short report.

2.1 Preparation

Set up Virtual machine Compiling and installing a new kernel is a risky task so you should work with the kernel inside a virtual machine. You **must** download and install **Ubuntu 18.04** on a virtual machine to do this assignment. You could use **VMWare** or **Virtual Box** to install the virtual machine. When create the virtual machine remember to allocate suitable resources to get the best performance of your system when doing this assignment (originally, number of (virtual) core equal to physical core, about 4GB ram depend on your system, and 40 GB hard disk space).
Important: Because making a mistake when compiling or installing a new kernel could cause the entire machine to crash, you must strictly follow instructions in this section. We also encourage you to frequently take snapshots to avoid repeating time consuming tasks and quickly restore the virtual machine.

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?

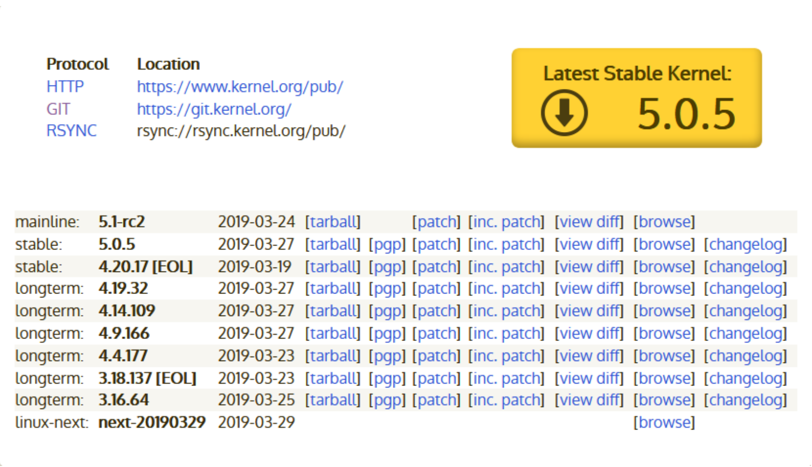
Create a kernel compilation directory: It is recommended to create a separate build directory for your kernel(s). In this example, the directory kernelbuild will be created in the home directory:

```
$ mkdir ~/kernelbuild
```

Download the kernel source: *Warning:* systemd requires kernel version 3.11 and above (4.2 and above for unified *cgroups* hierarchy support). See `/usr/share/systemd/README` for more information. In this assignment, you should choose the version 5.0.5 for consistency.
Download the kernel source from http://www.kernel.org. This should be the tarball (tar.xz) file for your chosen kernel. It can be downloaded by simply right-clicking the tar.xz link in your browser and selecting Save Link As..., or any other number of ways via alternative graphical or command-line tools that utilize HTTP, FTP, RSYNC, or Git.
In the following command-line example, wget has been installed and is used inside the `~/kernelbuild` directory to obtain kernel 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 from the server such as http://www.kernel.org, can we compile the original kernel (the local kernel on the running OS) directly?



Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Stable Kernel:
5.0.5

mainline:	5.1-rc2	2019-03-24	[tarball]		[patch]	[inc. patch]	[view diff]	[browse]	
stable:	5.0.5	2019-03-27	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
stable:	4.20.17 [EOL]	2019-03-19	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.19.32	2019-03-27	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.14.109	2019-03-27	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.9.166	2019-03-27	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.4.177	2019-03-23	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.18.137 [EOL]	2019-03-23	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.16.64	2019-03-25	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
linux-next:	next-20190329	2019-03-29						[browse]	

Figure 3: Kernel sources from www.kernel.org.

Unpack the kernel source:
Within the build directory, unpack the kernel tarball:

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

2.2 Configuration

This is the most crucial step in customizing the default kernel to reflect your computer's precise specifications. Kernel configuration is set in its .config file, which includes the use of Kernel modules. By setting the options in .config properly, your kernel and computer will function most efficiently. Since making our own configuration file is a complicated process, we could borrow the content of configuration file of an existing kernel currently used by the virtual machine. This file is typically located in `/boot/` so our job is simply copy it to the source code directory:

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

Important: Do not forget to rename your kernel version in the General Setup. Because we reuse the configure file of current kernel, if you skip this, there is the risk of overwriting one of your existing kernels by mistake. To edit configure file through terminal interface, we must install some essential packages first:

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

Then, run `$ make menuconfig` or `$ make nconfig` to open Kernel Configuration.

```
$ make nconfig
```

To change kernel version, go to General setup option, Access to the line "(-ARCH) Local version - append to kernel release". Then enter a dot "." followed by your MSSV. For example:

```
.1701234
```

Press F6 to save your change and then press F9 to exit.
Note: During compiling, you can encounter the error caused by missing `openssl` packages. You need to install these packages by running the following command:

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

2.3 Build the configured kernel

First run "make" to compile the kernel and create vmlinuz. It takes a long time to "$ make", we can run this stage in parallel by using tag "-j np", where np is the number of processes you run this command.

```
$ make
or
$ make -j 4
```

vmlinuz is "the kernel". Specifically, it is the kernel image that will be uncompressed and loaded into memory by GRUB or whatever other boot loader you use.
Then build the loadable kernel modules. Similarly, you can run this command in parallel.

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

QUESTION: What is the meaning of these two stages, namely "make" and "make modules"? What are created and what for?

2.4 Installing the new kernel

First install the modules:

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

Then install the kernel itself:

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

Check out your work: After installing the new kernel by steps described above. Reboot the virtual machine by typing

```
sudo reboot
```

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

```
uname -r
```

If the output string contains your MSSV then it means your custom kernel has been compiled and installed successfully. You could progress to the next part.

3 Trim the kernel

After install the kernel successfully, you have a new kernel with a default configuration. It will include support for nearly everything, so it is a general use and huge. With this configuration, you will need very long time to compile and long time to load. For use in this course, you need a different kernel configuration that create for your machine. In detail, you should reopen the "make nconfig" to choose suitable configuration for your machine. It will present you with a series of menus, from which you will choose the options you want to include. In most of cases, you have 3 choices: (blank) leave it out; (M) compile it as a module, which will be load if the feature needed; (*) compile it into the kernel, so it will be load in the first time kernel load. After

the configuration, you can rebuild the kernel (follow steps in section 2.3) and reboot the system to check the result.
REQUIREMENT:

- Boot the system with the new kernel successfully
- The new kernel must smaller than the default one (You should determine that which is the new kernel). This includes the boot image is smaller in size, less modules are compiled, etc. Consequently, the compile of the new kernel will be reduced.

Some helpful information:
You can look for information about devices you have and what drivers are running:

- dmesg to see the messages printed by device drivers or look at /var/log/syslog to see the system log file
- lspci or lsusb to see the devices that connect to the PCI bus or USB port respectively
- lsmod to see which modules are in use
- Look at /proc/devices to see devices the system has recognized.
- Look at /proc/cpuinfo or /proc/meminfo to see details about the processor and memory you have

When doing the assignment, booting the new kernel may not work. You can choose the previous kernel by enterring the GRUB menu by pressing *SHIFT* at the boot time. You also can display GRUB by default when booting the kernel.

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

And comment out GRUB_HIDDEN_TIMEOUT. To finish update GRUB by enter:

```
sudo update-grub
```

Configuring the kernel is a trial and error process so remember to back up the last configuration file by a sensible name. You also can use a version control system (github for example) for the management of changes.
In addition, since the version 2.6.32, **make localcongfig** is introduced. You can learn about it **(link)**

4 System call

Now you have known how to a kernel and install it to a system. In this section, you will learn how to add a new system call to the kernel.

4.1 The role of system call

The main part of this assignment is to implement a new system call that lets the user determine the information about the parent and the oldest child process. The information about the process's information is represented through the following struct:

```
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
};
```

Where the proc_info is defined as follows:

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

procinfos contain information of three processes: *proc* current process or process with pid; *parent_proc* parent of the first process; *oldest_child_proc* the oldest child process of the first process. The information of processes is store in *struct proc_info* and contains: *pid* process is of the process; *name* name of the program which is executed.

4.2 Prototype

The prototype of our system call is described as below:

```
long get_proc_info(pid_t pid, struct proinfos * info);
```

To invoke *get_proc_info* system call, user must provide the PID of the process or -1 in case of current process. If the system call find out the process having given PID it will get information of the process and put it in output parameter **info* and return 0. However, if the system call cannot find such process, it will return *EINVAL*

In addition, every time *get_proc_info* is invoked a message including your student ID must print to the kernel.

4.3 Implementation

Now you add the *get_proc_info* into the kernel and implement it. You look at appendix A to see the process of add a new system call into a kernel. The process of add *get_proc_info* can be described in brief:
In the kernel source directory enter the following command:

```
$ pwd
~/kernelbuild
$ mkdir get_proc_info
$ cd get_proc_info
$ touch sys_get_proc_info.c
```

Add the following lines to *sys_get_proc_info.c*

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

 struct procinfos { //info about processes we need
    long studentID; //for the assignment testing
    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
};

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

asmlinkage long sys_get_proc_info(pid_t pid, struct procinfos * info){
    // TODO: implement the system call
}
```

HINT:

- To find the current process: look at `arch/x86/include/asm/current.h` or for simple use macro `current` (`current -> pid`)
- To find info about each process: look at `include/linux/sched.h`
- To after the trimming process the time to build the kernel is reduced to about 10 minutes but it is till a long time to compile. To make to the development of system call as fast as possible, you can use kernel module to test the system call represented as a module in advance (Appd B).

Create a Makefile for the source file:

```
$ pwd
~/kernelbuild/get_proc_info
$ touch Makefile
$ echo "obj-y := get_proc_info.o"
```

Add get_proc_info/ to the kernel Makefile

```
$ pwd
~/kernelbuild/get_proc_info
$ cd ..
$ pwd
~/kernelbuild/
$ vim Makefile
```

Find the following line:

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

Add get_proc_info/ to the end of this line.

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

Add the new system call to the system call table:

```
$ pwd
~/kernelbuild/
$ cd arch/x86/entry/syscalls/
$ echo "548 64 get_proc_info sys_get_proc_info" >> syscall_64.tbl
```

QUESTION: What is the meaning of fields of the line that just add to the system call table (548, 64, get_proc_info, i.e.)
Add new system call to the system call header file:

```
$ ~/kernelbuild/include/linux/
```

Open *syscalls.h* and add the following line before *#endif* statement.

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

QUESTION: What is the meaning of each line above?
Finally recompile the kernel and restart the system to apply the new kernel:

```
$ make -j 8
$ make modules -j 8
$ sudo make modules_install
$ sudo make install
$ sudo reboot
```

4.4 Testing

After booting to the new kernel, create a small C program to check if the system call has been integrated into the kernel.

```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(548, -1, &info);
    printf("My student ID: %lu\n", info[0]);
    return 0;
}
```

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

4.5 Wrapper

Although get_proc_info system call works properly, we still have to invoke it through its number which is quite inconvenient for other programmers so we need to implement a C wrapper for it to make it easy to use. This can be done outside the kernel. Thus, to avoid recompile the kernel again, we leave out kernel source code directory and create another directory to store source code for our wrapper. We first create a header file which contains the prototype of the wrapper and declare procinfos and proc_info struct. Naming the header file with get_proc_info.h and put the following lines into its content:

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

#include <unistd.h>

 struct procinfos {
    long studentID;
    struct proc_info proc;
    struct proc_info parent_proc;
    struct proc_info oldest_child_proc;
};

struct proc_info {
    pid_t pid;
    char name[16];
};
```

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

Note: You must define fields in `procinfos` and `proc_info` struct in the same order as you did in the kernel.
QUESTION: Why we have to redefine `procinfos` and `proc_info` struct while we have already defined it inside the kernel?

We then create a file named `get_proc_info.c` to hold the source code file for wrapper. The content of this file should be as follows:

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

long get_proc_info(pid_t pid, struct procinfos * info) {
    // TODO: implement the wrapper here.
}
```

Hint: You could implement your wrapper based on the code of our test program above.

4.6 Validation

You could check your work by write an additional test module to call this functions but do not include the test part to your source file (`get_proc_info.c`). After making sure that the wrapper work properly, we then install it to our virtual machine. First, we must ensure everyone could access this function by making the header file visible to GCC. Run following command to copy our header file to header directory of our system:

```
$ sudo cp <path to get_proc_info.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`?
We then compile our source code as a shared object to allow user to integrate our system call to their applications. To do so, run the following command:

```
$ gcc -share -fpic get_proc_info.c -o libget_proc_info.so
```

If the compilation ends successfully, copy the output file to `/usr/lib`. (Remember to add `sudo` before `cp` command).
QUESTION: Why we must put `-share` and `-fpic` option into `gcc` command?.

We only have the last step: check all of your work. To do so, write following program, and compile it with `-lget_proc_info` option. The result should be consistent with the process information.

```
#include <get_proc_info.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 procinfos info;

     if (get_proc_info(mypid, &info) == 0) {
         // TODO: print all information in struct procinfos info
     }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 dependence
     // sleep(100);
}
```

You should write several tests using `get_process_time` to validate the correctness.

5 Submission

5.1 Source code

After you finish the assignment, you need to compress the following code files into a `zip` file:

- sys_get_process_time.c
- get_process_time.c
- all files you changed in the 5.0.5 Linux kernel
- report.pdf (Your report in PDF format)

Requirement: you have to code the system call followed by the coding style. Reference: https://www.gnu.org/prep/standards/html_node/Writing-C.html.

5.2 Report

As Figure 2 shown, the score for compiling kernel is 2 points, System call is 3 points, Wrapper of System call is 2 points and the report is 3 points. For the content of assignment report, describe steps of adding procmem system call to Linux kernel. The report layout is follows:

- Adding new system call
- System call Implementation
- Compilation and Installation process
- Making API for system call

In the report, just describe steps in short, succinct paragraphs. You have to add your answer to questions with the highlight word "**QUESTION**" throughout this instruction to related sections. Please do not answer questions as a list of items. One addition requirement for your report: describe in detail the process you study and implement the core (source code) of the system call. Your score are given based on the correctness of your answers and the clarify of the report content.

Appendices

A Add a new system call to kernel

A.1 Define a new system call sys_hello( )

To add a new system call to kernel, you first have to define the new system call. In this appendix, sys_hello() is used which print a message into the kernel log. In the kernel source directory, create a new directory named *hello* and change the directory to hello:

```
mkdir hello
cd ./hello
```

Create *hello.c* with the following lines:

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

asmlinkage long sys_hello(void)
{
        printk("Hello world\n");
        return 0;
}
```

printk() prints to the kernel log file.
Create a *Makefile* in the hello directory and add the following line:

```
obj-y := hello.o
```

A.2 Add hello/ to the kernel's Makefile

Change the directory and open the kernel Makefile:

```
cd ..
vim Makefile
```

Find the following line:

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

Add hello/ to the end of this line:

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

A.3 Add the new system call to the system call table

Modern processors support invoking system calls in many different ways depend on their architecture. Since our virtual machine runs on x86 processors, we only consider about Linux's system call implementation for this architecture.
In those file, each system call is declared in one row with following information: number, ABI, name, entry point and compat entry point separated by a TAB. System calls are call from user space through their numbers so our system call's number must be unique.
In the kernel source folder, run the following command to change the directory:

```
cd arch/x86/entry/syscalls/
```

Go to the last of 'syscall_64.tbl' and add the following line. (if you are on a 32-bit system you'll need to change 'syscall_32.tbl'. For 64-bit, change 'syscall_64.tbl'.)

```
548     64      hello       sys_hello
```

A.4 Add new system call to the system call header file

At this time, we have told the kernel that we have a new system call to be deployed but we still do not let it know the definition (e.g. its input parameters, return values, etc.) of this new system call. The next job is to explicitly define this system call. To do so, we must add necessary information to kernel's header files.
Go to the kernel source file and change the directory to include/linux/

```
cd include/linux/
```

Open *syscalls.h* and add the following line to near the end of file (before #endif statement)

```
asmlinkage long sys_hello(void);
```

A.5 Install/update Kernel

Recompile the kernel:

```
make -j 8
make modules -j 8
sudo make modules_install
sudo make install
```

And reboot to check the result.

A.6 Test system call

Create a C source file to call the system call.

```
//hello-syscall.c
#include <stdio.h>
#include <unistd.h>
int main()
{
        long int amma = syscall(548);
        printf("System call sys_hello returned %ld\n", amma);
        return 0;
}
```

Now compile *hello-syscall.c* and check the result.

```
$ gcc hello-syscall.c
$ ./a.out
```

In case of all the steps are done correctly, you will get the out put.

```
System call sys_hello returned 0
```

Run the following command to check the kernel message.

```
$ dmesg
```

This will display `Hello World` at the end of the kernel's message.
This is end process of adding sys_hello into you linux kernel.

B Kernel module

Treated the system call as a `Kernel Module` is a simple and fast way to save time compiling the kernel. To see how Kernel Module actually work, look at the `Hello World` example:

```
/*
 *  hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
```

```c
int init_module(void)
{
    printk(KERN_INFO "Hello world 1.\n");

    /*
     * A non 0 return means init_module failed; module can't be loaded.
     */
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye world 1.\n");
}
```

Kernel module must have at least two functions: a “start” (initialization) function called `init_module()` which is called when the module is insmoded into the kernel, and an “end” (clean up) function called `cleanup_module()` which is called just before it is rmmoded.

Compiling kernel modules and run it: please refer to the following **link**. For a deeper view of `Kernel Module`, look at **link**.

In this assignment, you can use the Kernel Module to test before writing the system call and compiling it.

```c
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");

module_param(PID, long, S_IRUSR);
MODULE_PARM_DESC(PID, "PID of the process");

int init_module(void)
{
    printk(KERN_INFO "Kernel module starts\n");
    printk(KERN_INFO "Input argument from insmod: %ld\n", PID);
    // TODO: Implement the kernel module
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Kernel module exits\n");
}
```

To check the kernel module:

```
$ sudo insmod <name of the module> PID=1
```

Reversion History

Revision	Date	Author	Description
1.0	2016	PD Nguyen, DH Nguyen, MT Chung	First Version
2.0	2019	TK Pham	Trim kernel, change the system call, add question