1 Purpose

The purpose of this assignment is to develop more knowledge of how to work with the Linux kernel. Additionally, you should start to learn how the source code is organized. Unfortunately, the online book you used for kernel modules does not explain how to create new system calls. This is inappropriate for the audience of the book. Fortunately, your OS book does illustrate how to do this. You can visit the book's website or read the end of chapter two for details. I will provide the important details in this document, as well. Note that all paths in this document are relative to /usr/src/linux.

2 Exercises

This assignment is broken down into two parts: modifying the kernel and accessing those modifications via a user application. You should turn in the code for both parts.

2.1 Creating a system call

To create the system call, you need to do three things: create an entry in the system call table, define a constant for that new entry location, and write the code. With these three items you've got a working system call.

2.1.1 Creating a system call table entry

You must edit the file in `arch/i386/kernel/syscall_table.S`. This file is an assembly file, so it is specific to a particular architecture. That is why it is located in the `arch` subdirectory. Since you are using the Intel 386 architecture, you choose the `i386` directory. The directory structure under `arch/i386` (or any other architecture) mirrors the directory structure in `/usr/src/linux` directly.

Don't worry if you don't understand assembly instructions. You won't have to write any special code for this assignment. This file only contains the data for various system calls. Note that the file looks something like:

```
.data
ENTRY(sys_call_table)
.long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting */
.long sys_exit
.long sys_fork
.long sys_read
.long sys_write
.long sys_open /* 5 */
```

`sys_call_table` is an array (table) of pointers (which fit into a long data value) to functions that implement system calls. Each line is the next system call in the table. When the OS gets a system call, the user program tells the

OS which system call it would like to run with an integer n. The OS goes to the nth element in the table and calls the function that is referenced in that element.

You need to add a new entry to the table, so you must go to the end of the file and add a line that looks like:

```
.long sys_mysyscall
```

where *sys_mysyscall* is the name of the function that will implement your system call. You may use whatever name you wish. It is just convention for it to start with `sys_` and then use a descriptive name.

2.1.2 Defining a constant

Note that there are comments in the system call table that help you figure out which element you are creating at the end of table. You need to let the rest of the kernel know where that system call is located. For this purpose, you must edit `include/asm-i386/unistd.h`. This file starts something like:

```
#define __NR_restart_syscall 0
#define __NR_ext 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
```

This gives a symbolic constant that points to the appropriate entry in the system call table. At the end of this list, add your system call to this list of definitions. Also, you need to increment the `NR_syscalls` definition to now include your new system call. The name of your constant should be `__NR_` followed by the name of the system call that is visible to user applications.

Note that there is some more code that follows these definitions. This code is included by applications to call the OS with the proper system call setting up the parameters properly. For instance, there is one `#define` called `_syscall0` that takes two parameters (`type`, and `name`). This creates a system call with zero parameters. It uses the defined constant to pass the right entry location to the OS.

2.1.3 Writing the code

Now, to add your system call. It is to have the following signature

```
int mysyscall(int flag, pid_t *pid)
```

where `flag` is a boolean value that tells the system call whether to print the process id into the system log using `printk()`. The process's id should be stored in the pointer given in `pid`. Any error condition should be signaled properly via the return value so that `errno` can be set by the system call handling code.

This seems pretty straightforward, but there are a couple of things you should be aware of. The biggest one is that `pid` is a pointer that points to some

logical address in the applications memory space. Technically, we could just use the pointer since Linux runs in the same address space as the user application that just made the system call. We must be aware of malicious users, however, that wish to break the system. They could provide us a valid pointer–not into their own process, but into the OS space–that would allow them to read or write memory that only the OS should know about. For this reason, Linux provides helper functions `copy_from_user()` and `copy_to_user()` to read from and write to user space respectively. These functions return true on failure and the system call should give the error message `-EFAULT` to the calling application.

Another thing to watch is where to add the code. You could add the function to `kernel/timer.c` since that is where the normal system call that returns the process id is located. To get practice adding files to the OS, I would like you to create a brand new file. This file should be in the `kernel` subdirectory since this is a "core" piece of functionality. You will have to modify the `Makefile` in that directory to add your new file to the build procedure. The best thing to do is add it to the `obj-y` set of files since you will want to unconditionally compile in this new system call.

The final thing to note is that this function needs to communicate with architecture specific code written in assembly language. The system call will be called by some assembly code and return a value to that same assembly code. The calling conventions for assembly code may be different than C calling conventions, so you need some way to indicate to the compiler to use the assembly calling conventions instead of the C calling conventions. gcc (which is the compiler for the Linux kernel) does this by adding `asmlinkage` before the function definition.

2.1.4 Building and installing your new kernel

I will not go into too much detail on this step since the installation is specific to the distribution you are using. If you type `make` in the top-level source directory, the build system should notice you've changed a few files and then recompile all the files that have changed, plus any files that depend on them to function correctly. Unfortunately, the first two files you modified are needed by almost every file in the kernel and the build system will have to recompile the entire kernel. Hopefully, you do the edits right on those two files. If you have to recompile again because you had to edit your new file, the kernel should only build it. Make sure the kernel is including your file in the build.

When the build completes, it should create a file (`arch/i386/boot/bzImage`) that contains the OS kernel. Copy this file to your boot partition and set up your boot loader to use this file. Make sure you do NOT copy over your current kernel in case there was a mistake. You want to have some way to get back to a known working kernel.

2.2 Using your new system call

Create an application that uses your new system call. It should be a tester program that passes known valid and invalid data. How will your program know about your new system call? You're going to have to create the proper prototype for it. This is where the `_syscall` macros from `unistd.h` come in. You need to use the correct one to inform your program how to send a system call with two parameters to the OS. The structure of these macros is that the first parameter is the type the system call returns, the second parameter is the name of the system call, the third parameter is the type of the first system call argument, the fourth parameter is the name of the first system call argument, the fifth parameter is the type of the second system call argument, the sixth parameter is the name of the second system call argument, etc. `_syscall0()` defines a system call with 0 arguments, `_syscall1()` defines a system call with 1 argument, etc.

3 Helpful hints

There are over 250 system calls in the kernel source code. Use them to help you write this function correctly.

4 Constraints

There are no new constraints from this assignment from the previous assignment. You must write this in C and have it work on my laptop, but you won't be doing anything kernel version specific so things should just work.