1 Introduction

This assignment has the following goals:

1. To introduce you to enhancing the Linux kernel
2. To show you concrete kernel data structures that we discussed in class
3. To introduce you to adding a system call

This is an *individual* assignment.

2 Files

The files you need for this assignment are located in the git repository on Github. You can use the following command to download the files:

```
% git clone https://github.com/jtcriswell/csc256-mp2.git
```

This repository will contain a script called getfiles.sh. Running the `getfiles.sh` script will download the OpenWRT disk image from the course web site.

Please read the `README.md` files provided with the Linux kernel source code. This file contains directions on compiling the Linux kernel, starting up the QEMU virtual machine, and importing files into and out of the virtual machine.

3 Add a New System Call

Write a new system call for the Linux kernel. The system call you write should take one argument (a pointer to a data structure) and return various information for the process identified by the pid in the data structure. All return information will be put into the data structure. For the following discussion, all relative paths refer to the top of your kernel source directory (linux-3.18.42).

The prototype for your system call will be:

```
int prinfo(struct prinfo *info);
```

As part of your solution, you should define `struct prinfo` as the following in the file include/linux/prinfo.h:

```
struct prinfo {
  long state;                     /* current state of process */
  pid_t pid;                      /* process id (input) */
  pid_t parent_pid;               /* process id of parent */
  pid_t youngest_child_pid;       /* process id of youngest child */
  pid_t younger_sibling_pid;      /* pid of the oldest among younger siblings */
  pid_t older_sibling_pid;        /* pid of the youngest among older siblings */
  unsigned long start_time;       /* process start time */
  unsigned long user_time;        /* CPU time spent in user mode */
  unsigned long sys_time;         /* CPU time spent in system mode */
  unsigned long cutime;           /* total user time of children */
  unsigned long cstime;           /* total system time of children */
  long uid;                       /* user id of process owner */
  char comm[16];                  /* name of program executed */
  unsigned long signal;           /* The set of pending signals */
  unsigned long num_open_fds;     /* Number of open file descriptors */
};
```

Sibling processes are those sharing the same parent. Young/old comparison between processes are made based on their start time.

For your kernel code, you should use kernel header files (those in include/linux and include/asm) to define the needed types. Note that pid_t is defined in include/linux/types.h. For your user-space test programs, you should use the standard header files such as sys/types.h to define the needed C types.

Sanity Checking Your Structure Definition: Because you will need to define struct prinfo twice (once in your kernel code and once in your test programs), it is important to ensure that the field types and offsets match. The __builtin_offsetof() function can print the offset of a type within a structure:

```
printf ("sys_time offset is %lx\n", __builtin_offsetof (struct prinfo, sys_time));
```

When writing your kernel code, you should have both your system call and your test program print the offsets of fields in struct prinfo to ensure that they define the structure identically. This debugging code should be removed before submitting the assignment but will be helpful in preventing and diagnosing difficult-to-debug errors.

Specifications: Your system call should meet the following requirements:

- Each system call must be assigned a number. You should use number 181 as it is currently unused by existing system calls.
- Your code should handle errors that can occur and return correct error values. Your system call should return 0 if no errors occur. Your system call should detect whether the input prinfo structure is NULL and set errno to EINVAL if so. It should also set errno to EINVAL if the process

ID queried does not exist and set errno to EFAULT if the pointer to prinfo is non-NULL but points to invalid memory.

Error codes are defined in include/asm/errno.h.

- Your system call should truncate the program name to 15 bytes and unconditionally add a string terminator byte to the `comm` field.

Hint: Linux maintains a list of all processes in a doubly linked list. Each entry in this list is a task_struct structure, which is defined in include/linux/sched.h. In include/asm/current.h, current is defined as an inline function which returns the address of the task_struct of the currently running process. All of the information to be returned in the prinfo structure can be determined by starting with `current`.

Another Hint: In order to learn about system calls, you may also find it helpful to search the Linux kernel for other system calls and see how they are defined. The file kernel/timer.c might provide some useful examples of this. The getpid system call might be a useful starting point. The system call sys_getpid defined in kernel/timer.c uses `current` and provides a good reference point for defining your system call.

4 Test Program

To test your system call, write a simple program that calls the prinfo system call with an input process ID. Your program should print all the process state information for the specified process.

Although system calls are generally accessed through a library (libc), your test program should access your system call directly. This is accomplished by utilizing the syscall macro in /usr/include/unistd.h. See the QEMU/Linux kernel lecture to see an example on how to proceed.

The output of the program should be easy to read. The ps command will provide valuable help in verifying the accuracy of information printed by your program. You can access detailed information on the ps command by entering `man ps`.

Your test program should be statically linked since the disk image used by the VM may not support the dynamic libraries needed by your program. To compile a program into a static binary, add the -static option to the GCC command line:

```
gcc -static -o test test.c
```

Note that the test program you are to write isn't the only set of tests you need to run on your kernel code. You should devise whatever tests are necessary to ensure that your code is implemented correctly.

5 Experiments

Using the test program you created for Section 4, run a few experiments. You will write up the results of your experiments in the README file that you hand in with your programming assignment:

- Write a program that prints out its process ID and then calls the `sleep()` system call to put itself to sleep for 100 seconds. Use your test program to find the state of the process. Report this state in your README file.
- Write a program that calls your system call with its own pid as input and prints out its own state. Run this program a few times. Report the state(s) that you see and describe what they mean.
- Write a program that prints out its process ID, uses `sigblock()` to block SIGUSR2, sends SIGUSR2 to itself two times, and then calls `prinfo()` to look up its pending signal set. Use your test program to print out the value of the list of pending signals. Report this value in the README and explain why signals cannot be queued.

6 Creating a Patch

The Linux kernel source code is large. In order to make submissions reasonable, you will need to provide a patch of your source code instead of submitting the entire kernel source code.

You should use the mkpatch.sh script within the source code distrubtion to create your patch:

```
% cd csc256-mp2
% sh mkpatch.sh > mypatch
```

This script finds all differences between the initial revision from the instructor's Github repository and your final revision and places them in the file named `mypatch`. If you clone the source code from the Github repository again, the following command will apply your changes to that source code:

```
% git clone https://github.com/jtcriswell/csc256-mp2.git
% cd csc256-mp2
% patch -p1 < mypatch
```

You should *test your patch on a clean clone of the instructor's Github repository*! If the grader cannot apply your patch, you will receive **zero points** for the assignment.

7 Administrative Policies

7.1 Turn-in

You are to create a directory named `mp2` and within this directory place the following items:

1. A patch file named `mypatch` containing your changes to the Linux kernel source code.
2. The source code for your test program described in Section 4 named test.c.
3. A README file, in text format, that contains:
   - a clear description of files you modified or added to the kernel source code to implement your system call
   - the results for the experiments in Section 5.

You should submit your files as a single GNU zipped tar archive named *netid*-mp2.tar.gz in which the text *netid* is replaced with your netid. When extracted, the TA should get a directory named `mp2` with your files located inside.

If you are unfamiliar with the tar and gzip/gunzip programs, please consult their man pages. You will submit your GNU zippped tar archive via Blackboard.

7.2 Code Comments

As you will learn when you start reading operating system kernel code, *too many* programmers write too few comments. It helped them code faster at the expense of your time and the time of all future kernel programmers to come. Therefore, to help end this cycle of misery, you are required to transcend current industry norms and comment your code.

Any non-trivial function should have a comment before the function describing its purpose, the meaning of its inputs, the meaning of its outputs, and the interpretation of its return value. Even simple functions that return `true` and `false` may need explanation of what those two values mean. Additional comments should exist throughout the code as necessary to explain what the code is doing and why.

In short, if the graders have difficulty understanding your code, then your commenting does not suffice.

7.3 Code Format

In addition to comments, your code should also be formatted well. While there are numerous coding standards, the most important factor in coding standards is *consistency*. This means that:

1. Your code should use the same formatting style as other code in the kernel. If the kernel puts the curly brace at the end of the line, put your curly braces at the end of the line. If the kernel indents using tabs, indent your code using tabs.

7.4 Grading

- *25 points:* your system call correctly reports the state, UID, parent process ID, and name of a process
- *20 points:* your system call correctly reports information for the oldest and youngest child and sibiling processes
- *10 points:* your system call correctly reports the information on pending signals
- *5 points:* your system call correctly reports the number of open file descriptors
- *10 points:* your README includes the results of the tests in Section 5, and the results are valid
- *10 points:* your system call does not hang the kernel, crash the kernel, or permit corruption of kernel memory
- *10 points:* your system call handles errors correctly
- *5 points:* your code is properly commented and formatted
- *5 points:* you turn in all requested files in the requested format