CSCI 401 Operating Systems
Instructor: Gedare Bloom
Project 2 (100 points)

You must do this assignment in collaboration with your assigned team members, if any. You may not use code downloaded from the Internet. Always watch the course website for updates on the assignments.

Read all instructions carefully. **Start early.** I recommend completing one Part each week.

Due Sunday, October 30

Part 0 (0 points): Due Wednesday, October 12

This project will be completed in teams of 2. Both team members will receive the same grade subject to participation. You are expected to work together, in person, at the same computer. You may choose your own team member. You and your team member must arrange to join the same Project2_Groups on Blackboard prior to submitting. Send the instructor an e-mail if you have trouble and need a manual adjustment of your group.

You will need a Linux development environment for this assignment, in particular you need to be able to boot and run the Linux kernel version 3.16.0. I recommend that you install Virtualbox and download the virtual machine (VM) available from the text book's companion site http://os-book.com/ -- You have to select the "Operating Systems Concepts" book and find the link to the Linux Virtual Machine. While you're on the site I highly recommend you browse around. The VM is a big file (almost 3 GB) so you will want to start the download immediately.

The LKD book and supplemental LDD book (available as PDF) will be helpful.

If you do use the VM from the text book web site, you need to do a little bit of system administration to add some additional hard drive space, because the hard disk it comes with is not large enough to recompile Linux. Without starting the VM, but after you download it and import it into Virtualbox, open Virtualbox, right-click on the OSC-2016 machine, and select "Settings". Go to the Storage menu, right-click on the SATA controller, and add another Hard Disk. You can select the VMDK format, and probably should use at least a 16 GiB drive. You should see "NewVirtualDisk1.vmdk" under the Controller: SATA item. Now boot the VM. The new disk should be available as /dev/sdb, but it needs to be partitioned and formatted to be usable. Run `sudo gdisk` and type device filename `/dev/sdb` to select the new disk. Type `p` to see the current disk information, it should show you on the first line the name "Disk /dev/sdb: … 16.0 GiB", and further down that the "Total free space … sectors". If not, you have the wrong disk and need to seek help elsewhere. Now you will create a partition by entering command `n`. Then just hit enter a few times to accept the defaults, and when you get the "Command" prompt again, type `p` and you should see a partition now. Write out the partition with the `w` command. Now you have a partition, and you need to create a filesystem on it. We will use the ext3 file system, so just run

```
sudo mkfs.ext3 /dev/sdb1
```

Note sdb1 this time, because we're making the file system on partition 1 that we just created. Congratulations, you made a file system! Now create a mount point for it, and mount it. Let's use `sudo mkdir /linux` and then `sudo mount /dev/sdb1 /linux` followed by `sudo chown -R oscreader /linux` and finally `cd /linux`. Eventually you will want to add this mount point to the `/etc/fstab` file, so open it and add the line:

```
/dev/sdb1      /linux   ext3
```

Download the kernel source for 3.16.0 from https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.16.tar.gz and move it into `/linux`, then uncompress and untar it to get `/linux/linux-3.16` directory.

Enter the linux-3.16 directory, open the Makefile in a text editor, and modify `EXTRAVERSION=` by adding a . and an extra version number or string, `"EXTRAVERSION=.project2"`.

Configure the kernel:

```
      cp /boot/config-* .config
      make oldconfig
```

You may need to answer a few questions, but the default should be OK so just hit "Enter" through all of the prompts. Now compile the kernel:

```
      make -j2
```

Coffee Break.

Install the kernel and modules:

```
      mkdir /linux/mykernels
      make INSTALL_PATH=/linux/mykernels install
      make INSTALL_MOD_PATH=/linux/mykernels modules_install
```

```
cd /linux/mykernels
sudo cp vmlinuz-3.16.0.project2 /boot/
sudo cp System.map-3.16.0.project2 /boot/
sudo cp -r ~/mykernels/lib/modules/3.16.0.project2 /lib/modules
sudo mkinitramfs -o /boot/initrd-3.16.0.project2.img 3.16.0.project2
sudo update-grub
```

Reboot the machine and enter the advanced options in the grub menu and then select the newly compiled kernel to boot it.

You can check the running kernel by executing the command `uname -a` in the shell.

You're ready to start hacking (this version of) Linux!

Part 1: Informative kernel modules (40 points)

(a) The `pid_info` module.

You will write a simple kernel module called `pid_info`. The module should cleanly load and unload an unlimited number of times. The module should take a parameter of the pid of a specific process. When loaded, the module will lookup the process `task_struct` of the specified process and print out, using the `printk()` function, the following information nicely formatted.

- pid
- uid, gid of process
- parent process id and name
- all of the children processes pid and name
- any flags that are set. Make sure to translate them to readable names.

You can find the `struct task_struct` defined in the file `linux/sched.h` The LKD chapters on processes, modules and appendix A on linked lists can be helpful. There is also an example of a module and Makefile already available on the virtual machine, located at `${HOME}/osc9e-src/ch2/`.

The c source file with your module should have two global variables (pid parameter and module name) and only two functions (the module init and cleanup). Use the MODULE_ macros (`linux/module.h`) to document the license (GPL), author, and description of your module. Your makefile should have a default target to build your modules and a clean target to remove all compiled files. Your modules should have the exact name given in the assignment.

(b) The `pid_vminfo` module.

Now you will write a module called `pid_vminfo` that creates a /proc entry file that when read, prints out the virtual memory state of a process. The virtual memory state of the process can be found by looking at the `mm` field of the `task` struct you used in the first module. You should print out at least enough information to identify the process (pid, name, ...) and all of the `vm_area` structs that are listed in the `mmap` list. Additional information that is displayed in a useful format will be a bonus.

The module should print out the state of whatever process is running when its proc file is read. So it does not need to lookup a pid like the first module. Creating and destroying /proc entries is easy; you can do it with a single function call. Documentation on how can be found on page 11-14 of Chapter 4 Debugging of the Linux Device Drivers 3rd Edition available online.

Part 2: Implementing System Calls (40 points)

You will create three new system calls and test them. You will need to implement the system call in an existing kernel source file, add the system call to the appropriate headers, and write or use a userspace test program to verify your system calls work correctly. Your system calls will set and retrieve the value of several variables stored in kernel space. The first variable is a variable length string, call it `my_string`. The second variable is an integer, call it `my_accumulator`.

Implementing system calls requires that you add the code for the actual call (prefixed with `sys_`) to an existing kernel source file, add the system call to the system call table, and add the header for the call to syscalls.h.

Make sure that your system calls have exactly the same signatures as below. Otherwise our test programs will fail to call your system calls and they will appear to fail. Function and variable names are case sensitive. All system calls should return an error code of `EINVAL` for inputs that are invalid. All error return values should be real Linux kernel error values (such as those found in `include/linux/errno.h`) that most closely match the error you are reporting. We specify a few specific errors that must be used, such as `EINVAL` and `EOVERFLOW`.

System call 1: sys_my_set_state

```
    long my_set_state(char *the_string, int accumulator)
```

This call sets the value of an integer and one string. The `accumulator` integer passed as input must always be non-negative. The string can be arbitrarily long, but is always null terminated, so you must dynamically allocate memory to store it when set. If a string is already stored in the kernel string variable, the memory allocated to it must be correctly freed. Strings stored in the kernel variable must be null terminated. The return value should be 0 for success, or a negative error value.

System call 2: sys my_get_and_sum

```
    long my_get_and_sum(int *my_value, int inc_value)
```

Take the value stored in the kernel variable `my_accumulator` and add the value passed in the `inc_value` parameter, which is always non-negative. Return the result in the `my_value` return value. The `my_accumulator` should store the updated value. If the resulting value is greater than `MAX INT`, then `EOVERFLOW` error should be returned. The return value should be 0 for success, or a negative error value.

System call 3: sys_my_get_string

```
    long my_get_string(char *buf, int bsize)
```

Take the kernel string variable stored in the kernel and copy it into the user provided buffer `buf` of size `bsize`. If the kernel string variable is empty, an error of `ENODATA` should be returned. The returned string will always be `NUL` terminated and therefore the following rule should be followed for selecting the string length to return:

> If `slen` is the length of the current string stored in the kernel variable, copy the leftmost `copylen = min( slen, bsize -1)` characters and append a `NUL` terminator to the end.

The system call should return the length of the string copied if successful, or a negative error value.

Because you will be modifying existing kernel source files, you will use the kernel build commands from Part 0 to re-compile your modified kernel.

Kernel Patches

It is very useful to be able to create kernel "patches" which show only the specific changes made and not all of the unchanged code. Since many modifications to the kernel are small (compared to the total kernel source size), such a patch can be many orders of magnitude smaller then the whole kernel. When you submit your changes to the kernel for assignments, you will submit a patch that we will apply to our copy of the kernel. So if you generate a bad patch, we may not get a working kernel even if you had it working yourself. So test applying your own patch before sending it. For this part of the assignment you need to generate a patch with only your changes to the kernel.

Testing (10 points)

You should write your own small (less than 50 line) programs that will let you test additional system call cases, such as calling your system calls in different sequences.

You should also write test programs for using your kernel modules.

Documentation (10 points)

Comment your code with /* */ style comments. Explain each function's purpose and the use of parameters. Include your name and a brief description at the top of each source file. If something doesn't work, you may get partial credit if your comments show that you were on the right track.

Write a README file that explains how to build and run your program, gives a brief description of your solutions, and contains any notes from discussions with others. Document any places where you received help from others.

Help

Some helpful functions/macros are `printk, kmalloc, kfree, copy from user, for_each_process, list_for_each, list_entry` (`man` or Google these if necessary. Pay attention to return values!)

Cross-linked Linux source: http://lxr.linux.no/linux/

Submission Instructions (read carefully)

Your submission must consist of your README, module source code and makefiles, and kernel patch(es); do not include compiled output!. Your source code must use UNIX style EOL characters \n not DOS style \n\r. You can write code on Windows, but you must convert it to Unix text files before submitting. The dos2unix command may be useful for this. Put all of your files in a single directory with both team members last names as Lastname1Lastname2/.

Make a tar and gzip file with both team members last names in the file name.
– tar -zcvf Lastname1Lastname2.tgz Lastname1Lastname2/.
Where all of your files are in the Lastname1Lastname2 directory. Upload the tgz file to Project 2 on Blackboard.

Grading Rubric

Part 1. pid_info – 20 points

- (5) Module parameter handling correct
- (5) Task structure lookup is efficient and correct (at most once through list)
- (10) Requested fields printed using printk
    - Formatting is clear
    - Children list lists all children

If module does not work correctly, up to 15 points can be received based on the code design and documentation.

Part 2: pid_vminfo – 20 points

- (5) Module finds currently executing process
- (5) Proc file entry created correctly
- (5) Read operation function handles short reads by returning partial information
    - (+3) Bonus if read op handles repeated reads for different byte ranges
- (5) Requested vm information printed using printk
    - Formatting is clear
    - All vm_area_structs are printed with address range and perms (RWXS)
    - (+5) Extra MM relevant information that is printed

If module does not work correctly, up to 15 points can be received based on the code design and documentation.

Part 3: System Calls – 40 points

- (5) Proper Modification of syscall_table.S - 5
- (10) my_set_state
    - (5) Use copy_from_user to move data from user space to kernel space.
    - (3) Use copy_from_user to move data from user space to kernel space (efficiency points deducted).
    - (1) Use strnlen_user to find the string length of user space string.
- (10) my_get_and_sum
    - (5) Use copy_to_user to move data from kernel space to user space.
- (10) my_get_string (using copylen)
    - (2): correct return length
    - (5): Use copy_to_user to move data from kernel space to user space.
    - (3): Use copy_to_user to move data from kernel space to user space.
    - Efficiency points may be lost if you forget to:
        - (3) Use copy_to_user or put_user to set null-term.
        - (3) Set null-term if copied amount is smaller than string stored.
        - (3) Calculate the correctly shortened buffer.

- (5) Proper Patch File
    - No extra files in patch, generated correctly from top level directory
    - Applies cleanly to vanilla 3.16.0 kernel.

Up to 25 points may be earned for the quality of algorithms and explanations in code if the system calls are not implemented correctly.

Testing and evaluation of modules and syscalls – 10 points

- (7) Basic operation tested (loads, unloads, calls execute).
    - (3) syscall test description.
    - (4) Module test description.
- (3) Checks for error conditions and displays useful output.

Documentation and Writeup – 10 points

- (5): Writing Matters.
- (5) Documentation in comments and README are clear and sufficient. Document all functions, parameters, and files. Discuss any problems, features, and testing.

Miscellaneous reasons points may be lost:

* Module load/unload does not catch errors or crashes - 5
* Internal kernel functions error return are not always checked and handled - 10
    * -5: Fails to check copy_to/from_user.
    * -5: Fails to check kmalloc.
* Calling System Calls out of Order Crashes - 5
    * -1: Get before Set string fails, but no crash.
    * -3: Get before Set string causes seg fault
    * -5: Get before Set fails for both string and accumulator.
* Error return values incorrect (EOVERFLOW, EINVAL, ENODATA, ...) - 10
    * -2: Didn’t use -ENOMEM for kmalloc failures.
    * -2: Didn’t use -EFAULT for general failures.
    * -2: Didn’t use negative of error conditions: e.g., return -ENOMEM;
        * -3: Overflow not properly detected.
* Misuse of KMalloc - 5
    * unnecessary free/realloc
    * incorrect size of buffer for string