CSE 451, Introduction to Operating Systems, Spring 2013

Course Home
Home

Administrivia
Overview
Using course email
Email archive
Discussion board

Materials
Lecture material
Sections
Midterm and final

Assignments
Projects
Homework and reading

Information
VM information
Linux information

Project 1: System Calls and Shells

Out: Friday October 4
Due: Friday October 18 electronically @ 11:59 p.m.
Submission: See the section on submission instructions.

Project Setup

There are two parts to setup: establishing your kernel build environment, and establishing your kernel test environment. The department labs have been set up to make this easy, if you work in them. It is also possible to work on, say, your home machine. (High speed connectivity is almost certainly a requirement, though.) Details for getting either environment set up are on the VM information page. See also the CSE451 git tutorial.

Obtaining the Linux Source

The version of the source we'll be using is on attu at:

```
/cse/courses/cse451/13au/linux-3.8.3-201.cse451custom.fc18.x86_64.tar.gz
```

This is the same as the one in the course VM under `~/rpmbuild/`. You must use this version of the kernel source no matter what -- no matter what platform it is you build on, no matter what year it is, not matter what the latest Linux distribution is, no matter what version you are running, no matter what Google says, etc.

Your Build Environment

This project involves writing both user-level code and kernel hacking. Exactly where this work *can* be done is pretty flexible. Because the situation is complicated, though, we describe only two:

1. Use the virtual machine we provide to write and compile your userspace code, and to boot your modified kernel.
2. Use remote access to a CSE machine, `forkbomb.cs.washington.edu`, to do everything other than booting your hacked kernel. Boot your kernel on a virtual machine we provide, either on your own personal computer or on a Windows lab computer. This is the preferred approach as compilation on `forkbomb` will be faster than inside the VM.
3. Use your own Linux machine instead of `forkbomb`
4. (DANGEROUS) Use any of the Linux lab machines for compiling the kernel by doing it in /tmp.

For both setup situations, you need to copy the source files you've created or modified if you want them to be backed up. Just assume neither `forkbomb` nor the VM is backed up. The number of source files you will be editing is small, so you can easily back them up to your CSE home directory. `forkbomb` mounts your home directory under `/homes/iws/`.

Using `forkbomb`

You can login to `forkbomb.cs.washington.edu` using SSH. Due to the usual disk space quotas, you will not have enough quota in your home directory to copy and compile a kernel, so we have instead provided each of you with local disk space under `/cse451/username`, where `username` is your unix account name. When you make your copy of the linux kernel and when you edit or compile it, you should do so in this local directory, not your network home directory. To build the kernel source on `forkbomb`, execute:

```
ssh username@forkbomb.cs.washington.edu
```

```
cd /cse451/username
cp /cse/courses/cse451/13au/linux-3.8.3-201.cse451custom.fc18.x86_64.tar.gz .
tar -xvf linux-3.8.3-201.cse451custom.fc18.x86_64.tar.gz
```

Using the Course VM

From the VM you must move files to the host you're running on and then to your home directory, as the VM can communicate only with the host, not the internet as a whole. A nice option is to install sshfs and "mount" your CSE home directory into the VM:

```
$ cd $HOME
$ mkdir cse_home
$ sshfs user@attu.cs.washington.edu:~/ cse_home
.... do some stuff ....
.... edit your files in ~/cse_home ....
.... profit, of course ....
$ fusermount -u ~/cse_home # unmount
```

Using lab Linux Machines

Done at your own risk. Your CSE home directory has a limit on how much space you can use. There is no such limit imposed on files in the `/tmp` directory. Also, it is a local drive so there is no network overhead in fetching files. However, `/tmp` is wiped upon reboot. Follow instructions for forkbomb but instead of unpacking the kernel source in your home directory, create a new directory in `/tmp` and change permissions so it is not world readable. Anyone on that machine could look at your code and we don't want that...

Compiling

The kernel should be compiled with sane defaults. Just run

```
make -j<number of cores> bzImage
```

The kernel will now be compiled. Once it finishes, you will have a `bzImage` file located under `arch/x86/boot/bzImage`. From a Windows lab machine, copy the `bzImage` to your home directory using the file transfer client under:

> Start->Programs->Internet and Remote Connections->SSH->Secure File Transfer Client.

If you are working from your own personal machine, you can use WinSCP, `scp`, etc. to copy the `bzImage` file to your machine. Once you have the file on the computer that you are using, copy it into the running VM image that you set up from the directions on the VM information page. Assuming that the `bzImage` file is now in the home directory of your VM, you can install it by running:

```
sudo mv ~/bzImage /boot/vmlinuz-3.8.3-201.cse451custom.fc18.x86_64
```

Now restart the VM and test out your changes. If the kernel fails to boot, you can restart the VM and pick another kernel from the Grub menu in order to get back to a working VM environment.

Notes

- A lot of time can be wasted in the overhead of copying files around (synchronizing state). Try to figure out a good workflow that minimizes this.
- Seriously consider using git to manage your source code. There is a CSE451 specific tutorial and a ton of others via google. Try starting your search with git workflow.
- Don't forget that some of the build options described above reset themselves. Remember to keep a permanent copy of your work somewhere.

The Assignment

Assignment Goals

- C, beautiful C, ...
- To understand the roles of and relationships among applications, OS command interpreters (shells), library calls, system calls, and the kernel.
- To become familiar with the tools and skills needed to understand, modify, compile, install, and debug the Linux

kernel.
- To design and implement a simple shell, a simple system call, and a simple library routine.

Background: The Shell

As we'll discuss in class, the OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is usually called *the shell*: it is a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. `sh, ksh, csh, tcsh, and bash` are all examples of UNIX shells.

Every shell is structured as a loop that includes the following:

1. Print a prompt.
2. Read a line of input from the user.
3. Parse the line into the program name, and an array of parameters.
4. Use the `fork()` system call to spawn a new child process.
   - The child process then uses the `exec()` system call to launch the specified program.
   - The parent process (the shell) uses the `wait()` system call to wait for the child to terminate.
5. Once the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most of the commands people type on the prompt are the name of other UNIX programs (such as `ls` or `cat`), shells recognize some special commands (called internal commands) that are not program names. For example, the `exit` command terminates the shell, and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking child processes to handle them.

Background: Library Routines

Library routines are just code that is made available to you. They differ from routines you write yourself in two ways, though:

- The code is provided in compiled form - `.o` files rather than `.c`.
- A library is actually a single file that (typically) contains many `.o` files within it (just as Winzip, `jar`, and `tar` files contain other files within them).

Many languages come with some kind of "standard library" that provides commonly used functionality. C does, and so does C++. The Java API might be thought of as the standard library for that language.

In C, many of the standard library functions serve as an interface between the code written for user applications and the operating system on which the application runs -- the library routines provide a standard interface to user code for IO (e.g, `printf()` and `scanf()`). This allows the applications to be somewhat portable (able to run on many different systems) - all you have to do is compile the application code on some system and link it with the standard library functions (written for and compiled on that system), and voila.

In Unix, libraries are manipulated with the `ar` command. In a typical installation, standard libraries are located in `/usr/lib`, and have file names beginning with "`lib`". The name of the standard C library is usually `/usr/lib/libc.a`, for instance.

Linkers, like `ld` (used by `gcc`), will often automatically look in standard libraries for code required at link time. If you have a non-standard library (as you will in this assignment), you have to tell the compiler/linker about it, e.g.,

```
$ gcc -c sub.c # creates sub.o
$ ar r mylib.a sub.o # creates mylib.a with sub.o inside it
$ gcc main.o mylib.a # creates a.out
```

See documentation on `gcc` for more complicated uses of libraries, and `man ar` for more information on creating and examining them.

The Assignment

This assignment has four parts:

1. Implement a simple shell.

2. Implement a new system call.
3. Implement a library routine capable of invoking that system call, and a simple driver program that exercises it.
4. Answer some questions about your implementation.

The implementation approach we suggest has a slightly different order than that, though, the problem being that to test the new syscall you need a working library routine, and to test the library routine you need a working app. The suggested methodology minimizes the number of pieces of code in play **not** known to be working, thus making debugging easy, or at least easier.

Step 1: Build a new shell (`fsh.c`)

Write a shell program (in C) `fsh.c` that has the following features:

- It should recognize three internal commands:
  - `exit [n]` terminates the shell, either by calling the `exit()` standard library routine or causing a return from the shell's `main()`. **If an argument (`n`) is given, it should be the exit value of the shell's execution**. Otherwise, the exit value should be the value returned by the last executed command (or 0 if no commands were executed.)
  - `cd [dir]` uses the `chdir()` standard library routine to change the shell's working directory to the argument directory. If no argument is given, the value of the `HOME` environment variable is used.
  - `. filename` causes commands to be read from the file. When end-of-file is reached, the shell returns to reading commands from the keyboard.
- If the command line doesn't invoke an internal command, the shell assumes it is of the form `<executable name> <arg0> <arg1> .... <argN>`
- Your shell uses the `fork()` standard library call, and some flavor of `exec()`, to invoke the executable, passing it any command line arguments.

Assume that executable names are specified as they are using "a real shell," i.e., name resolution involves the `PATH` environment variable (hint: is there a version of the `exec()` function that involves the `PATH` variable?). Try to use the same prompt as in the following:

```
CSE451Shell% date
Sat Mar 31 21:58:55 PDT 2012
CSE451Shell% /bin/cat /etc/motd /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
CSE451Shell% ./date
./date: No such file or directory
```

Notes:

- The words in **bold** in the sample session above were output by the shell, and those to the right of them were typed by the user. (Where did the non-bold, non-underlined words come from?)
- Please take a look at the manual pages for `execvp`, `fork`, `wait`, and `getenv`.
- To allow users to pass arguments to executables, you will have to parse the input line into words separated by whitespace (spaces and '\t' (the tab character), and then create an array of strings pointing at the words. You might try using `strtok()` for this (`man strtok` for a very good example of how to solve exactly this problem using it). See the `readline` library for a good method of taking input from the user.
- You'll need to pass the name of the command as well as the entire list of tokenized strings to one of the other variants of exec, such as `execvp()`. These tokenized strings will then end up as the `argv[]` argument to the `main()` function of the new program executed by the child process. Try `man execv` or `man execvp` for more details.
- Users tend to have a lot of bugs, and consequently dealing with them in a robust way can require a lot of code, especially if you're feeling like being helpful about reporting just what the mistake might be. Except as explicitly noted in the "specs" above, or when trivial to implement and of significant value, it's fine not to be very helpful toward the user in writing this shell.

Here is a selection of clarifying questions from last year's operating systems class and answers to them:

Question: Do we need to support quotes or multiple levels of quotes? For example, cd "/path/to/file with spaces".
Answer: No, don't worry about quotes. You also don't need to worry about escaped spaces, which should simplify tokenizing the arguments. Note that this "dumbs down" the capabilities of the shell, to an extent, so if you do

support either quotes or escaped spaces we will award bonus points.

Question: Can we specify a maximum limit on the length of a command to the shell? Answer: Yes, as long as the shell handles it gracefully if a command comes in that's longer than that limit (i.e. print an error message "error: command too long", but don't overflow the internal buffer and don't crash). As a hint, though, readline() (see man readline) could simplify this for you dramatically...

Question: Is the shell expected to handle pipes / redirection? Answer: No; just assume that the first argument is always the program name, and pass the rest of the stuff as arguments to that program. If the user passes | or > or etc. to the shell, it will just fail.

Step 2: Create a driver routine, and a dummy library routine (`getexeccounts.c`, `getexecounts.h`, `getcounts.a`, `getDriver.c`)

Our goal in this step is to debug the process of creating and linking with a library, as well as debugging a driver routine that will be used to test the library routine.

We start by writing `getexeccounts.c` and `getexeccounts.h`. The latter defines a single function:

```
int getExecCounts(int pid, int* pArray);
```

The former contains the implementation of that function. `getExecCounts()` returns 0 for success and non-zero for failure. The argument `pArray` is assumed to be an array of four `int`s. The implementation at this point is just a *dummy routine*: it assigns the value `k` to the `k`th element of the array, `k=0,1,2,3`, and always returns success.

Now write `getDriver.c`, an implementation of `main()` that invokes `getExecCounts` and prints the returned values like this:

```
pid 22114:
    0    fork
    1    vfork
    5    execve
    4    clone
```

It is up to you how your `getDriver.c` is implemented; ideally, it would take a process ID as a command-line parameter and invoke `getExecCounts()` using it.

Now create a library `getcounts.a` with a single member `getexeccounts.o` and make sure you can link `getDriver.o` against it. The model to have in mind is that getcount.a is a library wrapper around the `getExecCounts()` function, which is specified in `getexeccounts.c/h`, and `getDriver.c` is just a demo program that links against the library and exercises its functionality.

Step 3: Add a new system call, and modify the library routine to use it

There are four system calls in Linux related to creating new processes: `fork`, `vfork`, `execve`, and `clone`. (The man pages will describe for you the differences among them, although those details aren't important to the implementation portion of this assignment.) Implement a new system call that returns to the calling program how many invocations of each of those four process creation calls have been performed by **a specific process and all of its descendants**.

To do this requires four things:

1. Modify the process control block definition (`struct task_struct` in `include/linux/sched.h`) to allow you to record the required information on a per-process basis. `kernel/taskstats.c` and `kernel/pid.c` both define some functions that will help with this.
2. Instrument the kernel to keep track of this information.
3. Design and implement a new system call that will get this data back to the user application.
4. Modify your library routine to invoke your new syscall and return the results.

You'll use your dummy app to test the syscall (and potentially other techniques such as `printk()`).

Notes: Items 1-3

- I suggest you wade, rather than dive, into this. In particular:
    1. If you've never compiled the kernel, go back through the project kernel information above.
    2. Now put a "`printk()`" somewhere in the code, and figure out how to find its output. (Hints: `/var/log` and "`man dmesg`").

3. Now implement a parameterless system call, whose body is just a `printk()` call. See this excellent walkthrough, though be aware it was written for an older version of linux with fewer preexisting system calls. For the 3.8.3 kernel we're using, here is an *actual patch* to the Linux kernel from the chromium team that implements a new syscall. You can see exactly which files needed to be changed.
4. If you end up using the Qemu VM, you can put a breakpoint on your syscall handler (`sys_whatever`) from within GDB to make sure everything is as you expect it to be.

- Both the design and implementation aspects of steps 2 and 3 present several different ways to approach this problem. It is your job to analyze them from an engineering point-of-view and choose an appropriate set of trade-offs for your implementation.
- **Warning 1**: Remember that the Linux kernel should be allowed to access any memory location, while the calling application should be prevented from causing the kernel to unwittingly read/write addresses other than those in its own address space. Details about this are here.
- **Warning 2 (Hint 0)**: Remember that it's inconceivable that this problem (warning 1) has never before been confronted in the existing kernel.
- **Warning 3**: User programs tend to have a lot of bugs. Remember that the kernel must never, ever crash. This means that it cannot trust the application to know what it's talking about when it makes a request, particularly with respect to parameters passed in from the application to the kernel. (It also means the kernel cannot have a memory leak, as that would eventually cause a crash.)
- **Warning 4**: Similarly, remember that you must be sure not to create security holes in the kernel with your code.

Notes: Item 4

- You can't write C code that causes the compiler to generate a `syscall` instruction (meaning you can't directly invoke raw syscalls from C). Instead, you need to use the `syscall()` library routine. This code fragment show you how to do that:

  ```
  #define __NR_execcounts [someNumber]
  #include <sys/syscall.h>
  #include <unistd.h>
  ...
  int ret = syscall(__NR_execcounts, ...);
  ```

  `man syscall` will tell you more.
- In a more realistic situation, your new syscall number would be put in a system include files so that `#include <unistd.h>` would provide it to user programs. That's cumbersome in this classroom environment. As a workaround, we just `#define` the new value in the one file that needs it (the library routine).

Step 4: Implement a utility application (`execcnts.c`)

We want to implement a program, `execcnts` that is to process creation syscall statistics what `time` (for info, `man time`) is to seconds. `execcnts` takes a command invocation line as arguments, executes the command, and prints out the number of each of the four process creation syscalls made in executing the command line. For intance, `execcnts find . -name '*.c'` would print the numbers of each of the four syscalls involved in executing that `find` command. A caveat is that `execvp` actually issues multiple `execve`s, so your counts may be artificially inflated, but you don't need to worry about this issue.

What to Turn In

You should submit the following files in an archive named `netid1-netid2.tar.gz` under the directory format specified, omitting the files produced by compilation (such as .o files, executables, etc.). **One** group member should submit this file to the Catalyst DropBox.

1. The C source code of your shell (`fsh.c`) under `shell/`.
2. The C source code (`getexeccounts.c` and `.h` files) of your library routine implementation, the library itself (`getcounts.a`), and the driver program (`getDriver.c`) under `counts/`.
3. Your implementation of the utility program, `execcnts.c` under `utility/`.
4. The source code of the files you changed in or added to the kernel under `kernel/`.
5. A compiled kernel image (the `bzImage` image file that you installed under `/boot`) with your changes under `kernel/`.
6. The names of all of the Linux kernel source files that you modified, and a written description of what you did to them and why you needed to do it (i.e. why was it necessary to modify this particular file) in a file called `files.txt` under `kernel/`.

7. A transcript showing you using your new shell to invoke the `/bin/date` program, the `/bin/cat` program, and the `exit` and `cd` commands supported by your shell in a file called `sample_output.txt` in the top-level directory of the tar.gz file. (`/usr/bin/script` might come in handy to generate this printout. As always, do `man script` to find out how to use the command.)
8. A file called README at the top-level directory of the archive. This should contain the names and usernames of the members of your group as well as answers to the following questions:
   a. What is your syscall design? What arguments does it take? How does it return results? What restrictions are there, if any, on its use?
   b. Explain the calling sequence that makes your system call work. First, a user program calls <.....>. Then, <.....> calls <.....>. ... and so on. You can explain this using either text or a rough diagram (don't spend more than 15 minutes on a diagram).
   c. Why **must** some of the internal commands your shell supports be internal commands? (That is, why couldn't they be implemented as separate programs, like all other commands?) In the specific case of '.', explain how you chose to implement it and why (e.g. by forking or by interpreting as direct input), including the tradeoffs this entails.

Do not underestimate the importance of the write-up. Your project grade depends significantly on how well you understood what you were doing, and the write-up is a demonstration of that understanding. **Please ensure that your group members' names and usernames are at the top of your write-up file.**

While we ask only for the source files that you created for the shell, library, driver, and utility program, it would be nice if you included Makefiles for building those targets as well.

The grade on the project will be calculated as follows:

- Shell: 10 points
- Library routine: 5 points
- Utility program: 5 points
- System call: 20 points
- Write-up: 10 points

Additional notes/tips

Adding new files to the kernel source tree: If you need to add a new file to the Linux source tree, do it in the following manner:

- Determine which existing directory the file(s) will live in.
- Edit the Makefile in that directory, and add the name(s) of the .or file(s) that will be created to the definition of O_OBJS.