COSC 301: Operating Systems
Lab 4: Kernel Diving I: Understanding system calls
Due: 28 September 2011
Fall 2011

Overview

Work on this lab in groups of 2 or 3. You can submit one short writeup (described at the end of this document) for the entire group; just make sure the put all group member names on it.

We've been doing a bit of what is called systems programming, in which we write code that uses the facilities of the OS. In this project, we'll be delving into a real kernel, to see what it is like on the other side. Cover your eyes! It may get ugly.

The real kernel we will use is called `xv6` , which is a port of a classic version of Unix to a modern processor, Intel's x86. It is a clean and beautiful little kernel, and thus a perfect object for our study.

This first project with xv6 is a warmup to gain some familiarity with the system, and thus relatively light on work. The goal of the project is simple: to add a system call to xv6. Your system call, `getcount()`, should simply return the value of a counter in the system.

Specifically, you need to add one counter for every existing system call in the system (see `syscall.h` for the 21 current system calls; `getcount()` will be the 22nd). Each counter should be incremented every time a particular system call is issued; thus, if you repeatedly call `fork()` to create new processes, each call to `fork()` will increment the fork counter by one. When `getcount()` is called, the calling program can thus discover how often such a call is made.

Have fun!

The Code

Your new syscall should look like this: `int getcount(int syscall_number);`

The argument `syscall_number` is an integer indicating the system call for which we want the count. The return value is simply the system call count on success, and -1 on failure. The only way that this call should fail is if the system call number given as a parameter is invalid. System calls are numbered starting at 1, so "bad" system call numbers are anything less than 1 or greater than 22.

The count for a system call should only be incremented **before** the call is finished executing, not **after**.

The code and a pdf with cross-referenced kernel source can be found in `/projects/cosc301/xv6`. Everything you need to build and run and even debug the kernel is in there. To get started (assuming you are on one of the department Linux machines):

- Set up a working directory for the lab and `cd` there.
- Copy the xv6 files to your current directory: `cp /projects/cosc301/xv6/* .`
- Unpack the tar file: `tar xzvf xv6-rev5.tar.gz`
- Go into the xv6 source code directory: `cd xv6`
- Build the kernel: `make`

- Run it!: `make qemu`. This should pop up a new window with a console of the emulated OS[1].
  - Note: you can either run xv6 under the bochs emulator or under qemu, another emulation system. Typing `make qemu` or `make bochs` will invoke the respective emulator.
  - On your own linux system, you'll need to install qemu or bochs to do any of this. On Ubuntu, `sudo apt-get install qemu` or `sudo apt-get install bochs bochs-x` will do the trick.

Adding the system call

For adding your system call, you'll need to modify `syscall.h` and `syscall.c`. In `syscall.h`, just add one more `#define` line at the bottom for your new system call. In `syscall.c`, you'll need to do the following:

- Add a function prototype for your system call. You should add one more line after the last prototype (for `sys_uptime`, around line 100) to do that. You don't need to declare your system call as `extern` since we'll just write your function in the same C source file.
- Add a pointer to your function in the `syscalls` function table. Starting just after the last function prototype, there's an array of pointers to functions. You should add the function name of your system call as the last item in this array.
- Create a static array to hold your counters. Just after the system call function table, you can create an array of integers large enough to hold counters for all the system calls (22). You can just declare this array outside of any function (i.e., at global scope). Yes, we're doing the abhorrent thing of creating a global variable. Don't tell anybody.
- You'll need to write a bit of code to initialize all your counters (i.e., set them to zero). You can do this in `main.c`. You'll have to let `main.c` know that there's a global variable named `syscall_counts`. At the top of `main.c`, you can add something like the following to do that:

  ```
  extern int syscall_counts[];
  ```

  You can add your code to clear out the counters in the `mainc` function. I'd suggest doing that before the call to `userinit`.
- Modify the `syscall` function to add one to the respective counter for a system call that is called. It should be fairly clear where this increment should be done when looking at the source.
- Write the function! At the bottom of the `syscall.c` file, you can write the function body of your system call (which should be named `sys_getcount`). You'll need to use the `argint` function, defined at the top of `syscall.c` to obtain the function parameter that the user mode process passes (i.e., the system call number for which to get the current counter value). The first argument to `argint` should be 0, and the second can just be the address of an int variable to put the argument into.

  Remember that your system call should return -1 on error.
- We just wrote the kernel-space function, but we need to let user-space code know about the function. Edit `user.h` to add a function declaration for `int getcount(int)`. The other file we need to edit is usys.S (eek — an assembly file!). Don't worry, all you need to do is add a line at the end that says `SYSCALL(getcount)`.
- Write a couple tests to make sure your code works. A couple suggestions:
  - You can use the `cprintf` function (pretty much like the standard `printf`) when inside the kernel to print a message to the console.
  - You can modify the `usertests.c` source code to add some tests of your `getcount` system call. When you're running the emulator, you can type `usertests` at the shell prompt to run the program.

[1] Note: you can do this remotely, too, but you'll probably want to invoke ssh with the `-X` option so that you can tunnel any X-window system windows (i.e., `ssh -X cs.colgate.edu`). This will work on MacOS X and on any system on which you have X installed. (You can install X with cygwin on Windows.) You can also run qemu without X-windows by typing `run qemu-nox`. You'll need to type Ctrl+a x to get exit.

Further information about xv6

You may also find the following reading about xv6 useful and interesting: `http://pdos.csail.mit.edu/6.828/2011/xv6/book-rev6.pdf`. This book was written by the same team at MIT that ported xv6 to the x86. You shouldn't need to consult this reading; it's linked here for reference only.

The xv6 website has even more detail about xv6 and its history: `http://pdos.csail.mit.edu/6.828/xv6/`.

What to turn in

For this lab, submit a short writeup with answers to the following questions:

1. In the `syscall` function, there's a line `proc->tf->eax = syscalls[num]();`. On this line, a specific system call is being called. How is the return value of this function propagated back to a calling user-mode process?

2. The one line we added to `usys.S` made it possible for a user-space program to call our new system call. Briefly describe how the one line in that file makes calling our system call possible. (Note: you'll have to look at the preprocessor macro (in assembly!) at the top of the file. This macro works sort of like a function, except that for each use of SYSCALL in that file (after the macro definition) we'll get a copy of the assembly code, with the parameter substituted in place of `name` in the macro. Note also that `int` is the interrupt instruction.)

3. Briefly describe what's going on in the `argint` call. Why do we need a special set of functions to get arguments from user-space programs?

4. We didn't do anything special to "protect" updating our counters in the `syscall` function. Assume that we have multiple processors managed by the OS. What sorts of bad things might happen as a result of our present design?

5. Briefly describe how you tested your function. Try to provide some evidence (from the output of running the emulator) that your system call runs as expected.