

Computer Science 161: Operating Systems

Section Handout: Week 1

CS161 Course Staff / GWA
{cs161, werner}@fas.harvard.edu

February 12, 2007

1 Administrivia

1.1 Teaching Staff

Geoffrey Werner-Allen (gwa) / Head TF
werner@eecs.harvard.edu
Terminal Room: TBA, probably rotating

Andrew McCollum, 2nd Year TF
mccollum@fas.harvard.edu
Terminal Room: TBA, probably rotating

Stephen Dawson-Haggerty, 1st Year TF
sdawson@fas.harvard.edu
Terminal Room: TBA, probably rotating

No one ever calls us, but if you want our phone numbers/IM handles in addition to this info, please ask nicely. (dholland@eecs.harvard.edu) who wrote OS/161 and is the authority on the subject. He may occasionally provide assistance with grading or other things as needed. In addition, CS161 alums Kevin Bombino and Jon Hyman may fill in during the semester if we need extra help or Steven, Andrew or I is away.

1.2 Deadlines

A few deadlines to keep in mind:

- (02/13/2007) - Assignment 0 due by classtime (i.e. 1PM).
- (02/27/2007) - Assignment 1 due by 5PM.

Reminder: you have no late days for Assignments 0 and 1, so make sure to start them early.

2 Obtaining Assistance

- Sections - Sections are intended to bridge between the lectures and the homework. This year we will focus on illustrating examples from class using OS/161. We will probably set some time aside each time for assignment-specific questions, but addressing the assignments will not be our primary focus in section.

- Terminal Room Hours - there will be three, two-hour chunks of office hours held by the course staff each week. Be there early and often. They are in the basement of the Science Center, near the Linux workstations.
- Bulletin Board - The course class bulletin board is located at <http://motelab.eecs.harvard.edu/bb>. This is your primary way of asking questions and our primary way of answering them. Before posting take a look around to see if your question has already been asked or answered!
- Web Site - The course web site is located at <http://www.eecs.harvard.edu/~mdw/course/cs161/>. Please refer to it for lecture notes, handouts, and important announcements.

3 Software Overview

3.1 What is OS/161?

OS/161 is the operating system that you will be developing in this class. It is a stripped-down operating system mainly inspired by some of the BSD-derived systems like NetBSD and FreeBSD. You will get to play around with it in this course, including adding synchronization primitives in Assignment 1, process support and a number of system calls in Assignment 2, virtual memory in Assignment 3, and a file system in Assignment 4.

Though OS/161 does not run on any real hardware at the moment, it looks and feels like a traditional BSD-derived OS. Thus, after familiarizing yourself with OS/161, you should be able to hack on Mac OS X, NetBSD, FreeBSD, OpenBSD, BSDi, etc.

3.2 What is System/161?

System/161 is the synthetic hardware platform upon which you run your operating system. It is essentially simulating an earlier MIPS r2000/r3000 architecture, without floating point operations. We have already installed it on `ice`, and it's sitting in `~lib161/usr/bin/` as `sys161`. If you source our `.cshrc`, as we say in Assignment 0, this should take care of setting up your path.

If you want to run at home, you can download and install the simulator from the web site. More details there.

You can specify what you want your simulated hardware to look like with the `sys161.conf` configuration file. You can specify what configuration file to use by using the `-c` option to `sys161`.

Similarly, if you want to start debugging the kernel from the beginning of bootup, you can specify this using the `-w` flag and then attaching `cs161-gdb` to it.

3.3 What is the CS161 toolchain?

The CS161 toolchain is the set of utilities (other than `sys161`) that you need to develop for the MIPS r2000/r3000. Again, we've got it compiled and built by default for you. If you want to work at home, you'll need to download and build this yourself.

Notable tools:

- `cs161-gcc` - the compiler for MIPS r2000/r3000. You should probably never have to invoke this directly.

- `cs161-gdb` - the debugger, which you will have to invoke frequently.
- `cs161-addr2line` - converts an address in hex to a line number, so you can figure out where your program crashed and burned.
- `cs161-{ld, as, ar, ranlib, ...}` - all the normal build utilities you know and love.

4 Debugging

Your biggest friend in this course, without a doubt, will be `cs161-gdb`. Without GDB, you will be dead.

You will not be able to use `printf()` debugging in this course. It simply isn't feasible. You may be wed to the idea – it will not work. For instance, it's hard to use `printf()` before the terminal is initialized, if you've broken `printf()`, etc. Even better, it's possible that by adding a call to `printf()` into your code, you change the scheduling of your system and you hide bugs that you might otherwise see (and that we will see).

Using GDB in this course is a little different than what you've done before – specifically, you're trying to debug a program within a simulator, rather than just debugging a program. If you ran `cs161-gdb sys161`, you would be debugging the simulator, which you should not have to do. If you ran `cs161-gdb kernel`, then you would be trying to debug your kernel without a simulator, and this just wouldn't work.

Instead, you want to run your simulator/kernel in one window, open `cs161-gdb` in another window, and run them side-by-side. This can be accomplished as follows. In the first window, type:

```
is01:~ 251> cd ~/cs161/root
is01:~/cs161/root 252> sys161 -w kernel
```

In the second window (your debug window), you attach `cs161-gdb` to it as follows:

```
is01:~ 251> cd ~/cs161/root
is01:~/cs161/root 252> cs161-gdb kernel
(gdb) target remote unix:./sockets/gdb
```

Now that it's attached, you should be able to happily debug away. Two comments – first, make sure you're using the same ice box for running and debugging.

So there are those who hold the philosophy, “If you have not stepped through every line of code with a debugger, then your code has a bug.” I'm not sure if I believe that's strictly true, but it's certainly mostly true.

The best way to actually figure out what your code is doing is to watch it do it – stare at a variable, and watch it get clobbered. Figure out why you keep hitting that `panic()` when you **know** there's no way it can be getting there.

There are other, even more heinous situations. Consider the fact that there will be bugs in your assignment 2 and 3 code. And you will need to use that code for assignments 3 and 4, respectively. Thus, it's entirely possible that one function is not working because of what code you wrote six weeks ago. `printf()`'s will not help you discover this.

I highly recommend you sit down and check out the GDB guide on the web site. For instance, did you know you could `set` the value of a variable at any time in GDB? Or that you can set up watchpoints on a variable, having the program break and report to you every time a variable changes? You can see a list of all the commands you can use in GDB by typing `help` at the prompt.

Some common, useful, fun, and exciting commands:

- break - set breakpoint at specified line or function.
- clear - clear breakpoint at specified line or function.
- watch - set a watchpoint for an expression.
- call - call a function in the program.
- display - print value of expression EXP each time the program stops.
- print - print value of expression EXP.
- set - evaluate expression EXP and assign result to variable VAR.
- backtrace - print backtrace of all stack frames.
- down - select and print stack frame called by this one.
- up - select and print stack frame that called this one.
- attach - attach to a process or file outside of GDB.
- continue - continue program being debugged.
- detach - detach a process or file previously attached.
- finish - execute until selected stack frame returns.
- interrupt - interrupt the execution of the debugged program.
- next - step program.
- step - step program until it reaches a different source line.

There are lots of other commands as well, but these commands alone should significantly ease your time in CS161.

5 Version Control

Since there will be two people working on the same codebase, and the codebase is large and complex, we will be using CVS to keep track of the code and make it so that you and your partner don't kill each other. There is (surprise!) an excellent guide to CVS on the web site. I'd suggest everyone look through it.

Basically, after initialized, all the master code for your system lives in CVS. In order to actually work with any of the code, you checkout a copy from the master repository. When you're done making your changes, you commit it back to the repository. Much like my parents, CVS dutifully keeps a copy of everything you've ever given it.

CVS allows you to look at snapshots of your code and compare it against other snapshots and the current version by using commands like `diff` and `tag`. This will be very useful for this class, as you will tag the code with `asst1-begin` before you start Assignment 1, `asst1` when you finish Assignment 1, and to figure out what code you need to turn in, you use a diff between `asst1` and `asst1-begin`.

CVS also has a really nifty feature in that if both you and your partner are working on the same code, and then you both commit it to the master repository, it will try to merge the changes for you. If it cannot, it will flag it for a manual merge. This allows you and your partner to work together, even on the same file, without fear of stepping on each other's code.

Rather than repeat what Dave Holland has said in the guide, though, I refer you guys to the web site. The most essential commands, though, are:

- checkout - checkout a module. You generally need to do this to start.
- update - update your copies of the source files.
- add - add a file to the repository
- remove - remove a file from the repository
- commit - commit a file into the master repository

6 Making a System Call

To review what was covered in lecture this last week, let's step through an example of what happens when a process makes a system call.

- User process calls `read()`.
- Standard C library
- Kernel interrupt handler
- Kernel syscall handler
- Kernel code
- Return from exception
- Back to user process.

6.1 Kernel Stack v. User Stack

What is a stack? Why do we need one?

Kernel stack: used by the process while executing in the kernel. One page (i.e. 4K) per thread/process. Statically allocated.

User stack: used by the process while executing in user space. Dynamically allocated. Virtually addressed.

7 Fork and Exec

Why use fork?

- So we can `exec()`, i.e. to create new processes
- To achieve simple concurrency, i.e. without managing threads.
- Any other reasons?

Why split `fork()` and `exec()`?

8 Context Switching

What state do we need to save? How is this accomplished? Examples from OS/161.