

CS169 Programming Assignment 1: Kernel

General Information

Assignment Out: Monday, 22 Sept. 2008
Assignment Due: Monday, 29 Sept. 2008/Monday, 6 Oct. 2008 (11:59 pm)

1 Introduction

In this assignment you will write the basic building blocks for the Weenix operating system: threads, processes, synchronization primitives, and device drivers. At this point you are writing only kernel code. You will not be interacting with user level processes and threads until a later point. We have split the assignment into three handouts: one details the code for Kernel 1, involving device I/O and terminal drivers; one detailing the code for Kernel 2, involving the scheduling of processes, interrupts, and synchronization; and this one, which outlines some general information on the assignment.

Writing an operating system can be complicated, so we provide much of the code to do basic kernel operations such as memory allocation and page table management. These are either beyond the scope of the course or too time consuming for too little value. You are free to modify the code that we provide as you see fit, although we suggest that you do not wander too far from the interfaces that we have provided (to make things easier on you and your TAs!). We want you to be able to take ownership of the code. Though you might not write the entire system from scratch, you should, by the end of the project, have a basic understanding of all the code involved. This will certainly be true by virtual memory, which will involve the final touches on the kernel that you're beginning to write.

At the end of this assignment, you should have several threads and processes running, reading and writing to multiple terminals and to the disk. It is important to emphasize early on that **test code is critical**. Your mentor TA will make an appointment with you so that you can demo your basic kernel functionality. **Without test code you have no demo!** A substantial portion of the grade will be based on the quality of your test code, since the more that you can demo the more likely that your mentor TA will believe your operating system actually works.

Before you start this assignment, you should read and understand the chapters of the Brown Simulator documentation which cover booting the simulator, the simulated processor, devices,

and interrupts. Virtual memory will be dealt with in forthcoming assignments. Until you begin VM, all of the code which you run on the Brown Simulator will be operating in **kernel mode**, including your test code.

2 Coding practices

In addition to the practices outlined in the *Programming Guide* handed out at the beginning of the semester, there is some kernel-specific information that you should be aware of. Proper use of these coding practices is essential in order to avoid wasting time trying to track down bugs that could be more easily identified.

2.1 KASSERT and PANIC

The `KASSERT` macro is analogous to the normal `assert` macro, which hopefully you love and use as much as we do. If you don't, you'll grow to love it. Whenever you make any assumption about a piece of data or an argument, assert that your assumption is true. No matter how obvious it may seem, interactions within the kernel can make a simple bug cause failures in otherwise obvious sections of code. If you reach a section of code that you shouldn't be in, use the `PANIC` macro to print a message to the console and quit.

2.2 dbg

In addition to detecting fatal conditions, you should include as much debugging information as possible when running the kernel. We have provided a convenient library to conditionally output debugging information based on the environment. All of the debug messages are divided into categories, which you can turn on and off individually. To output a debug message, or to conditionally test a piece of code based on the environment, do the following:

```
dbg(DBG_XXX, ("Debugging %s\n", filename));
if (dbg_active(DBG_XXX)) {
    /* Do something... */
}
```

Note the use of extra parentheses around the printf arguments. The different modes are enumerated in `dbg_modes.h`. You will probably not use all of the modes, but feel free to add modes of your own as you see fit. To add a debug mode, you must add it to both `DBG_NAMETAB` and `DBG_COLORTAB`. We also support color-coded debug modes, which makes sifting through large

amount of debug output much easier. Using ASCII escape sequences defines in `colordefs.h`, each mode is associated with a color through `DBG_COLORTAB`. We have picked some arbitrary colors, feel free to change them if you prefer different colors. If you do not want color-coded modes, then set them all to be `_NORMAL_`.

Control of the debug modes is done through the `DBG` environment variable. Each mode has a human-readable name, defined in `DBG_NAMETAB`. Look through this definition to find out which name corresponds to which modes. Names can refer to groups of modes as well. To enable all debugging, `DBG` should be set to `all`.

```
% setenv DBG all
```

To enable certain debugging modes only, separate the values by commas.

```
% setenv DBG vnref,term,buf
```

The debug string also supports `-NAME` to remove groups from the environment. This allows setting of flags such as this:

```
% setenv DBG all,-term,-buf
```

Which will turn on everything except terminal code (`DBG_TERM`) and buffer cache routines (`DBG_BUF`).

To change debug modes for one run only, you can use a command like this:

```
% env DBG=buf,vfs,vm make run
```

3 Test Code

This section describes some guidelines for writing test code for your kernel. It is your responsibility to think of boundary conditions which could potentially cause problems in your kernel. Test code is an important part of software development. If you cannot demonstrate that your kernel works properly, then we will assume that it doesn't. It is a good idea to show your test suite to your mentor before handing in to make sure that it is acceptable. To be in the best shape possible before moving on, you should be able to test all of the following situations:

- Run several threads and processes concurrently. Devise a way to show that multiple threads are running and that they are working properly.

- Stress test your terminal code. Have two threads read from the same terminal, which will cause each thread to read every other line. If you're not sure why this is, ask a TA and re-read the simulator guide. Make sure that you can input and output more data than can be held in the internal terminal buffer. Also, make sure that you can have two threads simultaneously writing to the same terminal. It is **very** important that you get this code working flawlessly, since it can be a constant source of headaches later on if you don't do it properly.
- Stress test your disk code. This will not be needed until the S5FS assignment, but it is a good idea to make sure that it works now. Make sure that you can have multiple threads reading, writing, and verifying data from multiple disk blocks. Think of ways that you can write to a disk and display the data stored there. Good reading/writing drivers are crucial in the filesystem stage.
- Create several child processes and wait for them to terminate.
- Demonstrate that threads and processes exit cleanly, and that the synchronization primitives work.
- Note that kernels are hard to test so you might want to actually sit down and design your test code well so that it tests everything and in a relatively nice way (e.g. you might want to make it interactive so that you have an easier time testing everything). You may find writing test function after function in a central spot works well, but this can lead to ugly and long commented-out blocks when you're editing and testing your code. Writing some sort of limited shell is a good idea.

4 Installing, Building, and Running

4.1 Mercurial

To install the source for this assignment, simply clone the code repository:

```
% cd <your cs169 directory>
% hg clone /course/cs169/asgn/weenix
```

`hg` is the front-end command to Mercurial, a distributed revision control system. While you probably won't be sharing your changes with others, we will provide updates to the assignment (like the next chunk of support code, bugfixes, and comment changes) via Mercurial. We also recommend, but don't require, that you use it to keep track of your changes, in case you care to revert or look back on your version history.

- **hg pull**: bring new changes from the course directory's repository to yours. This won't update your working copy; it will suggest that you thereafter do:
- **hg update**: Sync changes from your repository to your working copy. It will never throw away or overwrite local changes, but if there are conflicts, it may tell you to do 'hg merge' instead.
- **hg commit**: save the changes to your working copy to a commit. If you do this frequently and write good log messages, you will be happy later when you are seeing a bug that you fixed earlier and you can look back through the history and find out what changed. You can also do:
- **hg diff**: Show differences between places in the repository. You can do this to find out what you changed since your last commit, or see what the TAs changed before you update, or any number of other potentially useful things.

If you find yourself with Mercurial problems, use `man hg`, see a Mercurial guide online or ask a TA.

4.2 Building

Once the assignment is installed, browse the supplied code carefully (paying close attention to comments) before attempting to write any code. Sections of code which you need to implement are marked with the macro `NOT_YET_IMPLEMENTED`. If you hit one of these sections, a warning message will be printed out. To find out where all of these functions are, simply run:

```
% make nyi
```

Note that this command will reveal all `NOT YET IMPLEMENTED` lines regardless of whether or not they have been commented out or not. In order to suppress this you should delete them from your source. But be sure to do so only when your functions are implementation-complete (if not bug-free).

While browsing and editing, you may wish to use Cscope. To construct the Cscope metadata files and launch Cscope, run `make cscope`. Cscope also has various editor integration facilities which make use of these metadata files; see the TAs for help with this.

Before you can build the project, you must set the `KERNROOT` variable in the file `Makefile.defines`. This should be set to your local installation directory, and is found at the top of the file. Once this is done, you can build and launch the kernel with:

```
% make all run
```

If you run the kernel immediately without adding any code, you will see something similar to the following:

```
Not yet implemented: proc_init, file proc.c, line 33
Not yet implemented: memdevs_init, file memdevs.c, line 47
Not yet implemented: tty_device_register, file tty.c, line 39
Not yet implemented: tty_device_register, file tty.c, line 39
Not yet implemented: tty_device_register, file tty.c, line 39
Not yet implemented: disk_init, file disk.c, line 50
Not yet implemented: bootstrap, file boot.c, line 125
Terminated
```

The lines reading ‘Not yet implemented’ are indications that there is a function that you need to fill in. It terminates because of the halt in bootstrap.

5 Debugging

Debugging your kernel is infinitely easier than debugging a real kernel, thanks to the simulator. Use `gdb` to debug your kernel code. Since the kernel code is loaded as a shared object by the simulator, one cannot simply run `gdb kernel`. To run the kernel under `gdb`, run:

```
% make gdb
```

This will start up the debugger appropriately. If you are interested in the mechanics, look at the Makefile. We provide a `.gdbinit` in the kernel root directory that provides many useful aliases and commands for going through source. Keep in mind that if you have your own customized `.gdbinit`, it will be overridden by the one we provide (if you are in its directory).

By default, the debugger will stop in `kern.boot` after your kernel object is loaded. We think this is useful so that you can control when the rest of the code will be executed if you’ve made some changes during the initialization steps. However, to customize the behavior after this point, edit the `sbt` definition in `.gdbinit` in the kernel directory. There can be breakpoints, conditionals, or simply the `c` command to automatically continue through the `kern.boot` function. Here is a brief summary of the most useful `gdb` commands:

```
p expression  prints the value of an expression
      c        continue execution
      s        step into next instruction
      n        step over next instruction
  b funcname   stop in given function
b filename:line stop at the given point in the file
      bt       current stack frame
      up       go up one stack frame
      down     go down one stack frame
```

6 Handing In

To hand in your kernel, run ‘make clean’, and then:

```
% /course/cs169/bin/cs169_handin kern1
```

Again, *Make sure that you have adequate test code!* Without test code, you have no demo. Consult your mentor TA to make sure that your test code is sufficient to test the functionality of your kernel.