Assignment 1

Written Answers Due: 1:15 pm, Thursday, September 7, 2006 (*in class*)
Code Due: 11:59 pm, Thursday, September 7, 2006

The purpose of this assignment is to familiarize you with OS/161, System/161 and the build environment you'll be using. Mostly you'll be reading code. It is intended as an *easy* assignment, but you should take care to do it well, and if possible go *beyond* what it asks of you. Time invested in understanding OS/161 now will pay off later.

Preliminaries

Read the AssignmentsSetup wiki page, you *must* have your path correctly set in order to undertake all CS 134 assignments. If you haven't done so already, put the appropriate line(s) in your `.login` or `.bashrc`. Do it now!

Once your path is correct, run the following commands

`svn checkout` will wrongly guess the username, just press return and it'll ask.

```
cd
mkdir cs134
chmod go-rx cs134
cd cs134
svn checkout svn://inuyasha.cs.hmc.edu/cs134/fall06/pairs/<group>/assign1
```

(i.e., *"<group>" is the name of your group)*

```
cd assign1
./setup
```

Running `./setup` can take several minutes depending on the speed of the machine you are using. Remember that you can skip ahead to some of the reading parts of the assignment while you are waiting for parts to compile.

When `./setup` finishes, it will have built all of the user-level commands (such as `/sbin/reboot`) and libraries for OS/161, but *not* the kernel itself. You can check that it has compiled one of these files by running:

```
file root/sbin/reboot
```

which should print something along the lines of

```
root/sbin/reboot: ELF 32-bit MSB mips-1 executable, MIPS R3000_BE,
version 1, statically linked, not stripped
```

Building the Kernel

The OS/161 kernel uses a fairly complex but powerful build system. It uses a configuration file to specify which parts of the kernel need to be built, and provides separate build areas for each configuration.

Our first step in building a kernel is to configure the build using a configuration file. Make sure your current directory is `~/cs134/assign1`, and then run

```
cd src/kern/conf
./config HW1
```

(There are other configuration files in this directory of the form `ASST`*n*. They are used by Harvard's OS course. We can safely ignore them.)

Now that the kernel is configured, we can build it. Type

```
cd ../compile/HW1
make install
```

Once it's built, change directory to your virtual root directory and test the kernel, as follows

```
cd ~/cs134/assign1/root
sys161 kernel
```

You can't do much with the kernel at this point. Two things you *can* do are initiate a kernel panic with the `panic` command, and run a user level program that performs the "reboot" system call by typing `p /sbin/reboot`. Try both of these now.

You can also use `gdb`. Open another terminal window, and ensure both are in the directory `~/cs134/assign1/root`. As follows

First Window

Run the command

```
sys161 -w kernel
```

The system should pause with the message that it is waiting for a message from the debugger.

⋮

OS/161 should now start up. Enter the command

```
panic
```

⋮

Execution should resume, and the kernel should exit with its panic message and shut down the machine.

Second Window

⋮

Run the command

```
cs161-gdb kernel
```

and then once `gdb` has started, type the following commands into `gdb`

```
connect
break panic
continue
```

⋮

GDB should have now stopped at the breakpoint you set for `panic`. Type the following commands

```
backtrace
continue
```

Quit `gdb`.

Code Reading

In this course, you may be getting your first taste of understanding and carefully modifying a large body of code that you did not write yourself. It is imperative that you take the time to read through the important parts of the code to get an understanding of the overall structure of the code, and what each part of the code does.

The code-reading component of this assignment aims to guide you through the code base to help you comprehend its contents, identify what functionality is implemented where, and be able to make intelligent decisions on how to modify the code base to achieve the goals of this and later assignments.

You don't need to understand every line of code, but you should have a rough idea of what each file does.

The Top-Level Directory

The `src` directory contains the top-level directory of the OS/161. It contains a few files along with subdirectories containing distinct parts of OS/161. The files are

`Makefile`
: Used to build the OS/161 base system, including all the provided utilities. It does not build the operating system kernel.

`configure`
: A configuration script, similar to the configuration scripts used in many open source programs (although it is not one of the impenetrable scripts generated by GNU `autoconf`). You shouldn't need to understand or tamper with it. It is run by our `setup` program.

`defs.mk`
: Definitions read in by the `Makefile`, and generated by the `configure` program.

`defs.mk.sample`
: A sample `defs.mk` file in case something goes wrong with `configure`. The idea is that to allow developers to fix `defs.mk` using the comments in this file. For us, `configure` is run by our `setup` script and should *never* fail.

In addition to these files, the `src` directory contains the following directories:

`bin/`
: Contains the source code for the user-level utilities available on OS/161. They are a subset of the typical unix `/bin` tools; for example, `cat`, `cp`, `ls`.

`include/`
: Contains include files used to build user-level programs on OS/161; they are not the kernel include files. Among other things, these files contain appropriate definitions for using the C library available on OS/161.

`kern/`
: Contains the sources to the OS/161 kernel itself. We will cover this code in more detail below.

`lib/`
: Contains the user-level library code for `libc`, the C library.

`sbin/`
: Contains the source code for the user-level system management utilities found in `/sbin` on a UNIX machine (e.g. `halt`, `reboot`, etc.).

`testbin/`
: these are pieces of test code. They are most relevant to the course given at Harvard, but are included here for your perusal and potential use.

Your focus during this code walkthrough should be on the kernel sources. You won't need a detailed understanding of the utilities in `bin` and `sbin`, but a general idea of how they work and where things are is useful. Likewise with the `lib` and `include` directories.

The `kern` Subdirectory

The margin notes provide hints on which directories are relevant for which written questions.

This directory and its subdirectories are where most (if not all) of the action takes place. The only file in this directory is a `Makefile`. This Makefile only installs various header files. It does not actually build anything.

We will now examine the various subdirectories in detail.

`kern/arch/`
: This directory contains architecture-dependent code, which means code that is dependent on the architecture OS/161 runs on. Different machine architectures have their own specific architecture-dependent directory. Currently, there is only one supported architecture, `mips`.

W1

`kern/arch/mips/conf/conf.arch`
: Tells the kernel config script where to find the machine-specific, low-level functions it needs (see `mips/mips`).

`kern/arch/mips/conf/Makefile.mips`
: Kernel Makefile; copied when you run the `config` script to configure a kernel.

W2–W6

`kern/arch/mips/include/`
: Contains include files for the machine-specific constants and functions.

`kern/arch/mips/mips/` W7–W9
: Contains the low-level machine-dependent functions.

`kern/asst1/`
: This is the directory that contains framework code for one of the assignments at Harvard, *not* this assignment. You can safely ignore it.

`kern/compile/`
: Where you built your kernel. Remember...?

`kern/conf/`
: Where you configured your kernel. Hopefully you remember that too...

`kern/include/` W10–W16
: Contains include files that the kernel needs.

`kern/include/kern` W17–W19
: Contains include files that the kernel and user-level code needs.

`kern/main` W20–W21
: Contains code to initialize the kernel. Guess which function you'll find defined here...

`kern/thread/` W22–W23
: Contains code implementing threads. Threads are the fundamental abstraction on which the kernel is built.

`kern/lib/` W24
: Contains library routines used throughout the kernel; for example, managing sleep queues, run queues, `kmalloc`, and so forth.

`kern/userprog/`
: Will contain code to create and manage user-level processes. As it stands now, OS/161 runs only kernel threads; there is essentially no support for user-level code.

`kern/vm/`
: Also fairly vacant. Virtual memory would be mostly implemented in here.

`kern/fs/vfs/` W25–W28
: Contains code implementing a file-system abstraction layer (vfs stands for "Virtual File System"). It establishes a framework into which you can add new file systems easily. You will want to review `vfs.h` and `vnode.h` before looking at this directory.

`kern/fs/sfs/`
: Contains code implementing the simple file system that OS/161 contains by default. You don't need to worry too much about this directory for now.

`kern/dev/`
: Where all the low-level device-management code resides. You can safely ignore most of this directory.

Written Component

Work *together* as a pair to find the answers to the questions below. You are not required to share a single machine pair-programming style, but that may be the easiest way to work on this part. You must make sure that you both know how each answer was found, have read through the relevant source file, and know why your answer is "the right answer".

Legibly write or print your pair's answers on letter-sized paper and bring them to Thursday's class. You do not need to use the submit system for your answers to this section. In class I may ask you *individually* about the answers you gave as a pair, *or*, ask you closely related questions whose answers you should know from answering the questions below.

W1. The kernel for this assignment is configured to use a particular VM system. What is this VM system called?

W2. Which register number is used for the stack pointer (`sp`) in OS/161?

W3. What bus/busses does OS/161 support?

W4. What is the difference between `splhigh` and `spl0`?

W5. Why do we use **`typedef`**s like *u_int32_t* instead of simply saying *int*?

W6. What must be the first thing in the process-control block?

W7. What does `splx` return?

W8. What is the highest interrupt level?

W9. What function is called when user-level code generates a fatal fault?

W10. How frequently are "hardclock" interrupts generated?

W11. What functions comprise the standard interface to a VFS device?

W12. How many characters are allowed in an SFS volume name?

W13. What is the standard interface to a file system (i.e., what functions must you implement to implement a new file system)?

W14. What function puts a thread to sleep?

W15. How large are OS/161 pids?

W16. What operations can you perform on a *vnode*?

W17. What is the maximum path length in OS/161?

W18. What is the system call number for a reboot?

W19. Where is `STDIN_FILENO` defined?

W20. What does `kmain()` do?

W21. Is it okay to initialise the thread system before the scheduler? Why (not)?

W22. What is a zombie?

W23. How large is the initial run queue?

W24. Can an array represented by a *struct array* be resized?

W25. What does a device name in OS/161 look like?

W26. What does a raw device name in OS/161 look like?

W27. What lock protects the vnode reference count?

W28. What device types are currently supported?

- Remember to bring your answers with you to class on Thursday.

Coding Component

In this portion of the assignment, we make a small modification to the OS/161 and debug it.

C1. Find the place in the code that prints `Put-your-group-name-here` and edit it to print your group's name.

C2. Recompile and test your kernel.

- Submit your code by running `svn commit`.
- In `src/kern/main` there is an extra file, `hello.c`, which is currently not compiled or used. It defines a function `complex_hello` that is intended to print "`Hello World`". (The code in this file is more complex than strictly necessary, but you should *not* replace it with a simple call to `kprintf`.)

C3. Edit one of the kernel source files to add a call to `complex_hello` so that "`Hello World`" will be printed once, just before the prompt appears.

C4. Edit `src/kern/conf/conf.kern` appropriately to include `hello.c`.

C5. Since you've changed a kernel configuration file, rerun the configure program in `kern/conf` (using the same steps you used on page 2).

C6. Change directory into `src/kern/compile/HW1` and rebuild the kernel (again, using the same steps as you used on page 2).

C7. Run the kernel. It will panic.

C8. Use `gdb` to find the bug.

C9. Fix the bug.

- Submit your code by running `svn commit`.