Assignment 0: An Introduction to OS/161

Due: In class on Tuesday, February 10, 2009.

Objectives

After this assignment, you should:

- Be familiar with Subversion (SVN) and GDB (the GNU Debugger).
- Understand the source code structure of OS/161, the software system we will be using this semester.
- Understand how System/161 provides the "hardware" environment on which OS/161 runs.
- Understand the source code structure of System/161.
- Be comfortable reading the OS/161 source code and figuring out where things are done and how they are done.
- Be prepared to undertake Assignment 1.

Preliminaries

NB: Throughout the rest of the assignment we will use <username> to refer to your FAS username; so please replace it with the appropriate string (no brackets).

NB: To shorten the Subversion URLs this document uses minitrue instead of minitrue.eecs.harvard.edu. If you are using cs161.seas, you are fine, since minitrue is set up in /etc/hosts correctly. Otherwise please use the full hostname.

In order to provide you access to `cs161.seas.harvard.edu` the cluster of machines on which we will be developing this term, and to grant you access to the Subversion repositories on `minitrue.eecs.harvard.edu`, we need two pieces of information from you.

First, please generate a RSA key pair. This can be done on FAS/ICE/NICE or any other UNIX system using a utility called `ssh-keygen`. The directions here may be helpful for some. You may have generated these keys already, and if so they are usually stored in `~/.ssh` or `~/.ssh2` and named `id_rsa.pub`. If you have not yet generated a key, it should be as easy as:

```
% ssh-keygen -t rsa
```

This will store the public key in the location described above. Please attach `id_rsa.pub` to an email along with the next piece of information we're about to collect.

Second, please generate a md5sum of a password you would like to use to access your Subversion repository. Again, on FAS/ICE/NICE you can simply use the following command:

```
% echo -n "<username>:minitrue:<password>" | md5sum
# Example: echo -n "gwa:minitrue:cs161" | md5sum
# 295c9c7d2b087975c789f86d46937c61 -
# You send us: gwa:minitrue:295c9c7d2b087975c789f86d46937c61
```

The hashed password is the first part of the output; ignore the "-". Also note that the password does not include the quotes. You may need backslashes to protect certain special characters used by the shell if your password contains them.

Now, please send an email to the course staff from your FAS account including, in the body, your FAS Unix username, your hashed password (step 2) and as an attachment, your `id_rsa.pub` file (step 1).

Once you are finished and have received confirmation from the course staff, please verify that:

1. You can log in to `cs161.seas.harvard.edu`:

   ```
   % ssh <username>@cs161.seas.harvard.edu
   % pwd
   /home/<username>
   ```

2. You can access your Subversion repository on `minitrue`:

   ```
   % svn list http://minitrue/repos/<username>
   branches/
   tags/
   trunk/
   ```

Once the two commands above seem to be working properly, continue with the rest of the assignment.

Introduction

In this assignment, we will introduce:

System/161

The machine simulator for which you are building an operating system this semester.

OS/161

The operating system you will be designing, extending, and running this semester.

Subversion (SVN)

Subversion is a source code revision control system. It manages the source files of a software package so that multiple programmers may work simultaneously. Each programmer has a private copy of the source tree and makes modifications independently. SVN attempts to intelligently merge multiple people's modifications, highlighting potential conflicts when it fails.

GDB (Gnu Debugger)

GDB allows you to examine what is happening inside a program while it is running. It lets you execute programs in a controlled manner and view and set the values of variables. In the case of OS/161, it allows you to debug the operating system you are building instead of the machine simulator on which that operating system is running.

The first part of this document briefly discusses the code on which you'll be working and the tools you'll be using. You can find more detailed information on SVN and GDB in separate handouts. The following sections provide precise instructions on exactly what you must do for the assignment. Each section with (hand me in) at the beginning indicates a section where there is something that you must do for the assignment.

What are OS/161 and System/161?

The code for this semester is divided into two main parts:

- **OS/161:** the operating system that you will augment in subsequent homework assignments.
- **System/161:** the machine simulator that emulates the physical hardware on which your operating system will run. This course is about writing operating systems, not designing or simulating hardware. Therefore, you may not change the machine simulator. If, however, you think you have found a bug in System/161, **please let the course staff know as soon as possible.** We have introduced some new support in the simulator for multiprocessing this year and bugs are at least possible.

The OS/161 distribution contains a full operating system source tree, including some utility programs and libraries. After you build the operating system you boot, run, and test it on the simulator.

We use a simulator in CS161 because debugging and testing an operating system on real hardware is extremely difficult. The System/161 machine simulator has been found to be an excellent platform for rapid development of operating system code, while still retaining a high degree of realism. Apart from floating point support and certain issues relating to RAM cache management, it provides an accurate emulation of a MIPS processor.

There will be an OS/161 programming assignment for each of the following topics:

- ASST1 : Synchronization and concurrent programming
- ASST2 : System calls and multiprogramming
- ASST3 : Virtual memory
- ASST4 : File systems
- ASST5 : Security

CS161 assignments are cumulative. Ideally you will build each assignment on top of your previous submission. If, however, at any point you wish to build on top of a solution set, contact your TF for a copy of the appropriate solution set, even if we have not yet released it.

Using the solution sets may seem like an attractive alternative, since they are guaranteed to work. Keep in mind, however, that using the solution set requires that you understand a code base that you did not write. We encourage you to consider these trade-offs carefully and select the solution that is likely to work best for you. We are happy to discuss the trade-offs with you and help you make a wise decision, but the final decision is yours.

About Subversion

Most programming you have probably done at Harvard has been in the form of 'one-off' assignments: you get an assignment, you complete it yourself, you turn it in, you get a grade, and then you never look at it again.

The commercial software world uses a very different paradigm: development continues on the same code base producing releases at regular intervals. This kind of development normally requires multiple people working simultaneously within the same code base, and necessitates a system for tracking and merging changes. Beginning with assignment 2 you will work in teams on OS/161 and will be developing a code base that will change over the couse of several assignments. Therefore, it is imperative that you start becoming comfortable with Subversion, an open source version control system.

Subversion is a powerful tool, but for CS161 you only need to know a subset of its functionality. The Introduction to Subversion handout contains all the information you need to know and should serve as a reference throughout the semester. If you'd like to learn more, there is comprehensive documentation available here .

About GDB

In some ways debugging a kernel is no different from debugging an ordinary program. On real hardware, however, a kernel crash will crash the whole machine, necessitating a time-consuming reboot. The use of a machine simulator such as System/161 provides several debugging benefits. First, a kernel crash will only crash the simulator, which only takes a few keystrokes to restart. Second, the simulator can sometimes provide useful information about what the kernel did to cause the crash, information that may or may not be easily available when running directly on top of real hardware.

To debug OS/161 you must use a version of GDB configured to understand OS/161 and MIPS. This is called `mips-harvard-os161-gdb`. It has been installed on `cs161.seas.harvard.edu` and can also be run using the shorter name `cs161-gdb`. This copy of GDB has also been patched to be able to communicate with your kernel through System/161.

An important difference between debugging a regular program and debugging an OS/161 kernel is that you need to make sure that you are debugging the operating system, not the machine simulator. Type

```
% cs161-gdb sys161
```

and you are debugging the simulator. Not good. The handout Debugging with GDB provides detailed instructions on how to debug your operating system and a brief introduction to GDB.

Setting up your account

We have created some scripts to help you set up your environment so that you can easily access the tools that you will need for this course. (A note to `[t,]csh` users: the equivalent set of scripts for `[t,]csh` are located in the same directory and entitled `cs161.cshrc`, `cs161.login`. Some of the commands that follow need to be modified to work on `[t,]csh`. We trust that you will know what to do.)

1. Add the following lines to your .bashrc (or equivalent) file:

```
. ~cs161/usr/etc/cs161.bashrc
```

2. Add the following line to your .bash_profile file:

```
. ~cs161/usr/etc/cs161.bash_login
```

3. You also probably want to tell Subversion which editor to use when it wants input from you. You do this by setting the `SVN_EDITOR` environment variable in your `.bashrc` file.
4. Log out and back in.

Getting the Distribution (hand me in)

First, download the OS161 source here.

In addition to the OS161, you will see distributions of System/161, the machine simulator, and the OS161 toolchain. If you are developing on the supported CS161 environment, you **do not** need these additional files, as they are already installed. If you wish to develop on your home machine at home, you will need to download, build, and install these packages as well.

1. First, check out a directory from our grading repository into which you'll place the files that you will submit, and create the `asst0` directory into which you'll put the files from **this** assignment.

```
% cd ~
% svn co http://minitrue/repos/grading/<username> submit
% svn mkdir submit/asst0
```

2. Script the rest of the following session:

```
% script ~/submit/asst0/setup.script
```

3. Make a directory in which you will do all your CS161 work. For the purposes of the remainder of this assignment, we'll assume that it will be called `cs161`.

```
% mkdir cs161
% cd cs161
% mv ../os161-current.tgz .
```

4. Unpack the OS/161 distribution by typing

```
% tar xzf os161-current.tgz
```

   This will create a directory named os161-version.

5. Rename your OS/161 source tree to just os161, and remove the tarball.

```
% mv os161-version os161
% rm *.tgz
```

6. End your script session by typing `exit` or by pressing Ctrl-D.

Setting up your Subversion repository (hand me in)

1. Script the following session:

```
% script ~/submit/asst0/svninit.script
```

2. Once you have completed the preliminaries we have created a Subversion repository for you on `minitrue`. Your Subversion repository URL is `http://minitrue/repos/<username>`.
3. Change directories into the OS/161 distribution that you unpacked in the previous section and import your source tree.

```
% cd ~/cs161/
% svn import os161 http://minitrue/repos/<username>/trunk/ -m "Initial import of os161"
% svn copy http://minitrue/repos/<username>/trunk http://minitrue/repos/<username>/tags/os161-1.99.01 -m "Tagging initial import."
```

   You can alter the arguments as you like; here's a quick explanation.

   `-m "Initial import of os161"` is the log message that Subversion records. (If you don't specify it on the command line, it will start up a text editor). `/trunk` is where Subversion will put the files within your repository. `http://minitrue/repos/<username>/trunk` is the Subversion URL you will specify when you check out your system. The second `svn copy` command creates a simple tag of the initial import of your system that you can later use with `svn diff` and other commands. More on that later (or see this tagging explanation).

4. Now, remove the source tree that you just imported.

```
% rm -rf os161
```

   Don't worry - now that you have imported the tree in your repository, there is a copy saved away. In the next step, you'll get a copy of the source tree that is yours to work on. You can safely remove the original tree.

5. Now, checkout a source tree in which you will work.

```
% svn co http://minitrue/repos/<username>/trunk src
```

   The `svn co` command creates a **working copy** of your tree that you can safely modify in `src` (feel free to put it wherever you like).

6. End your script session.

Code Reading (hand me in)

One of the challenges of CS161 is that you are going to be working with a large body of code that was written by someone else. When doing so, it is important that you grasp the overall organization of the entire code base, understand where different pieces of functionality are implemented, and learn how to augment it in a natural and correct fashion. As you and your partner develop code, although you needn't understand every detail of your partner's implementation, you still need to understand its overall structure, how it fits into the greater whole, and how it works.

In order to become familiar with a code base, there is no substitute for actually sitting down and reading the code. Admittedly, most code makes poor bedtime reading (except perhaps as a soporific), but it is essential that you read the code. It is all right if you don't understand most of the assembly code in the codebase; it is not important for this class that you know assembly.

You should use the code reading questions included below to help guide you through reviewing the existing code. While you needn't review every line of code in the system in order to answer all the questions, we strongly recommend that you look over at least every file in the kernel.

The key part of this exercise is understanding the base system. Your goal is to understand how it all fits together so that you can make intelligent design decisions when you approach future assignments. This may seem tedious, but if you understand how the system fits together now, you will have much less difficulty completing future assignments. Also, it may not be apparent yet, but you have much more time to do so now than you will at any other point in the semester.

The file system, I/O, and network sections may seem confusing since we have not discussed how these components work. However, it is still useful to review the code now and get a high-level idea of what is happening in each subsystem. If you do not understand the low-level details now, that is OK.

These questions are not meant to be tricky -- most of the answers can be found in comments in the OS/161 source, though you may have to look elsewhere (such as Tannenbaum) for some background information. Place the answers to the following questions in a file called `~/submit/asst0/code-reading.txt`.

Top Level Directory

The top level directory of many software packages is called `src` or `source`. The top of the OS/161 source tree is also called `src`. In this directory, you will find the following files:

`configure:` top-level configuration script; configures the OS/161 distribution, including all the provided utilities, but does not configure the operating system kernel.

`Makefile:` top-level makefile; builds the OS/161 distribution, including all the provided utilities, but does not build the operating system kernel.

You will also find the following directories:

`common:`code used both by the kernel and user-level programs, mostly standard C library functions.

`kern:`the kernel source code.

`man:`the OS/161 manual ("man pages") appear here. The man pages document (or specify) every program, every function in the C library, and every system call. You will use the system call man pages for reference in the course of assignment 2. The man pages are HTML and can be read with any browser.

`mk:`fragments of makesfiles used to build the system.

`tools:`programs used during compilation

`user:`user-level libraries and program code

In the `user` directory, you will find:

`bin:` all the utilities that are typically found in `/bin`, e.g., `cat, cp, ls`, etc. The things in bin are considered "fundamental" utilities that the system needs to run.

`include:` these are the include files that you would typically find in `/usr/include` (in our case, a subset of them). These are user level include files; not kernel include files.

`lib:` library code lives here. We have only two libraries: `libc`, the C standard library, and `hostcompat`, which is for recompiling OS/161 programs for the host UNIX system. There is also a `crt0` directory, which contains the startup code for user programs.

`sbin:` this is the source code for the utilities typically found in `/sbin` on a typical UNIX installation. In our case, there are some utilities that let you halt the machine, power it off and reboot it, among other things.

`testbin:` these are pieces of test code.

You needn't understand the files in bin, sbin, and testbin now, but you certainly will later on. Eventually, you will want to modify these and/or write your own utilities and these are good models. Similarly, you need not read and understand everything in lib and include, but you should know enough about what's there to be able to get around the source tree easily. The rest of this code walk-through is going to concern itself with the kern subtree.

The Kern Subdirectory

Once again, there is a Makefile. This Makefile installs header files but does not build anything.

In addition, we have more subdirectories for each component of the kernel as well as some utility directories. `kern/arch:` This is where architecture-specific code goes. By architecture-specific, we mean the code that differs depending on the hardware platform on which you're running. There are two directories here: `mips` which contains code specific to the MIPS processor and `sys161` which contains code specific to the System/161 simulator.

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

Question 0. What is the default compile option that we use for OS/161's virtual memory system?

`kern/arch/mips/include:` This folder and its subdirectories include files for the machine-specific constants and functions.

Question 1. In what file would you look to figure out how the various machine registers are labeled in OS/161?

Question 2. What are some of the details which would make a function "machine dependent"? Why might it be important to maintain this separation, instead of just putting all of the code in one function?

`kern/arch/mips/*:` The other directories contain source files for the machine-dependent code that the kernel needs to run. Most of this code is quite low-level.

Question 3. What will happen if you try to run on a machine with more than 512 MB of memory?

`kern/arch/sys161/conf/conf.arch:` Similar to `mips/conf/conf.arch`.

`kern/arch/sys161/include:` These files are include files for the System161-specific hardware, constants, and functions. machine-specific constants and functions.

Question 4. What bus/busses does OS/161 support?

`kern/compile:` This is where you build kernels. In the compile directory, you will find one subdirectory for each kernel you want to build. In a real installation, these will often correspond to things like a debug build, a profiling build, etc. In our world, each build directory will correspond to a programming assignment, e.g., ASST1, ASST2, etc. These directories are created when you configure a kernel (described in the next section). This directory and build organization is typical of UNIX installations and is not universal across all operating systems. `kern/conf: config` is the script that takes a config file, like ASST1, and creates the corresponding build directory. So, in order to build a kernel, you should:

```
% cd kern/conf
% ./config ASST0
% cd ../compile/ASST0
% bmake depend
% bmake
```

This will create the ASST0 build directory and then actually build a kernel in it. Note that you should specify the complete pathname `./config` when you configure OS/161. If you omit the `./`, you may end up running the configuration command for the system on which you are building OS/161, and that is almost guaranteed to produce rather strange results!

`kern/dev:` This is where all the low level device management code is stored. Unless you are really interested, you can safely ignore most of this directory.

`kern/include:` These are the include files that the kernel needs. The `kern` subdirectory contains include files that are visible not only to the operating system itself, but also to user-level programs. (Think about why it's named "kern" and where the files end up when installed.)

Question 5. What would splx(splhigh()) do?

Question 6. Why do you think types.h defines explicitly-sized types such as int32_t instead of using the shorter int?

Question 7. What about type names such as __time_t? What other purpose might these type definitions serve?

Question 8. What is the interface to a device driver (i.e., what functions must you implement to add a new device)?

Question 9. What is the easiest way to add debug messages to your operating system?

Question 10. What synchronization primitives are defined for OS/161?

Question 11. What is the difference between a thread_yield and a wchan_sleep?

Question 12. What version of OS/161 are you running? Why might this be important to know?

`kern/lib:` These are library routines used throughout the kernel, e.g., arrays, kernel printf, etc.

`kern/startup:` This is where the kernel is initialized and where the kernel main function is implemented.

`kern/thread:` Threads are the fundamental abstraction on which the kernel is built (do not forget to look back at header files!)

Question 13. What data structure do we use to keep track of the runnable threads in the system?

Question 14. Which synchronization primitives are completely provided for you? (Guess when the others will exist.)

Question 15. What is a zombie?

`kern/synchprobs:` This is the directory that contains the framework code that you will need to complete assignment 1. You can safely ignore it for now.

`kern/syscall:` This is where you will add code to create and manage user level processes. As it stands now, OS/161 runs only kernel threads; there is no support for user level code. In Assignment 2, you'll implement this support.

`kern/vm:` This directory is also fairly vacant. In Assignment 3, you'll implement virtual memory and most of your code will go in here.

Question 16. What is the purpose of functions like copyin and copyout in copyinout.c? What do they protect against? Where might you want to use these functions?

Question 17. Why are there two separate implementations of 'malloc' (kern/vm/kmalloc.c and user/lib/libc/stdlib/malloc.c)? Give one major difference between the two implementations.

`kern/vfs:` The file system implementation has two directories. We'll talk about each in turn. `kern/vfs` is the file-system independent layer (`vfs` stands for "Virtual File System"). It establishes a framework into which you can add new file systems easily. You will want to go look at `vfs.h` and `vnode.h` before looking at this directory.

Question 18. What happens when you do a read on /dev/null?

Question 19. What lock protects the current working directory?

`kern/fs:` This is where the actual file systems go. The subdirectory `sfs` contains a simple default file system. You will augment this file system as part of Assignment 4, so we'll ask you more questions about it then.

Question 20. The vnode layer is file system independent; why is there a file sfs_vnode.c in the sfs directory? What is the purpose of the routines in that file?

Building a kernel (hand me in)

Now it is time to build a kernel. As described above, you will need to configure a kernel and then build it.

1. Script the following steps:

   ```
   % script ~/submit/asst0/build.script
   ```

2. Configure your tree for the machine on which you are working. If you want to work in a directory that's not `$HOME/cs161` (which you will be doing when you test your later submissions) you might want to use the `--ostree option`. `./configure --help` explains the other options.

   ```
   % cd ~/cs161/src
   % ./configure
   ```

3. Now let's build and install the user level utilities.

```
% bmake
% bmake install
```

4. Now for the kernel. Configure a kernel named ASST0.

```
% cd ~/cs161/src/kern/conf
% ./config ASST0
```

5. Build and install the ASST0 kernel.

```
% cd ../compile/ASST0
% bmake depend
% bmake
% bmake install
```

6. End your script session.

Running your kernel (hand me in)

1. Download the file `sys161.conf` and place it in `~/cs161/root`. You should take a look at this file, as it describes how to configure the simulator you will be running your code in.
2. Script the following session:

```
% script ~/submit/asst0/run.script
```

3. Change into your root directory.

```
% cd ~/cs161/root
```

4. Run the machine simulator on your operating system.

```
% sys161 kernel
```

5. At the prompt, type `p /sbin/poweroff <return>`. This tells the kernel to run the "poweroff" program that shuts the system down.
6. End your script session.

Practice modifying your kernel (hand me in)

1. Create a file called `~/cs161/src/kern/startup/hello.c`.
2. In this file, write a function called `hello` that uses `kprintf()` to print "Hello World\n".
3. Edit `kern/startup/main.c` and add a call (in a suitable place) to `hello()`.
4. You must place a function prototype for `hello()` in some header file that both `hello.c` and `main.c` includes. For example, you could place it in `kernel/include/lib.h`. Or, you could create `kern/include/hello.h`, and have `hello.c` and `main.c` include `hello.h`.
5. You must also add `hello.c` to your `conf.kern` in `kernel/conf/`.
6. Reconfigure and rebuild your kernel.
7. Make sure that your new kernel runs and displays the new message.
8. Once your kernel builds, script a session demonstrating configuring, building and running your modified kernel. Put the output of this script session into `~/submit/asst0/newbuild.script`.

Using GDB (hand me in)

1. Script the following gdb session, that is, you needn't script the session in the run window, only the session in the debug window. Be sure both your run window and your debug window are on the same machine.
2. Run the kernel in gdb by first running the kernel and then attaching to it from gdb.

```
(In the run window:)
% cd ~/cs161/root
% sys161 -w kernel

(In the debug window:)
% script ~/submit/asst0/gdb.script
% cd ~/cs161/root
% cs161-gdb kernel
(gdb) target remote unix:.sockets/gdb
(gdb) break menu
(gdb) c
   [gdb will stop at menu() ...]
(gdb) where
   [displays a nice back trace...]
(gdb) detach
(gdb) quit
```

3. End your script session.

Practice with Subversion (hand me in)

In order to build your kernel above, you already checked out a source tree. Now we'll demonstrate some of the most common features of Subversion. Create a script of the following session in `~/submit/asst0/svn-use.script`. The script should contain everything except the editing sessions. Do those in a different window.

1. First, make sure that you have completed updating your working copy to include the necessary files from the "Hello World!" exercise above. `svn status` is your friend here. If you are concerned by all of the `.depend` files and `build` directories that show up when you run `svn status` in `~/cs161/src`, edit `~/.subversion/config`, uncomment the `global-ignores` line and add the `*.depend` pattern. This will eliminate the `.depend` files. To avoid seeing all of the build directories you may have to do a `bmake clean` from your `~/cs161/src` directory. There are better ways to avoid these sorts of irritations. Come see the

course staff. (A simple solution is to run `svn status` in `src/kern`, where you will be doing most of your work.)

2. Begin the script:

   ```
   % script ~/submit/asst0/svn-use.script
   ```

3. Change directory to `kern/startup/` and add a comment with your name to `main.c`.
4. First, from within your `~/cs161/src/kern` directory, execute:

   ```
   % svn status
   ```

   Note that `main.c` shows status 'M', indicating that the file has been modified. For the full list of `svn status` codes consult this cheat sheet.
5. Execute

   ```
   % svn diff -x -uw startup/main.c
   ```

   to display the differences in your version of this file.
6. Now commit your changes using `svn commit`.
7. Remove the first 100 lines of `main.c`.
8. Try to build your kernel (this ought to fail).
9. Realize the error of your ways and get back a good copy of the file.

   ```
   % svn revert startup/main.c
   ```

10. Try to build your tree again.
11. Now, examine the `DEBUG` macro in `lib.h`. Based on your earlier reading of the operating system, add ten useful debugging messages to your operating system.
12. Now, show us where you inserted these DEBUG statements by doing a diff.

    ```
    % cd ~/cs161/src/kern
    % svn diff -x -uw
    ```

13. End the script.

Create a Release

Finally, you should create a release. Refer to the document on Preparing your Assignments for Submission . **Once you are confident that your tree is in the condition you would like to leave it in**, i.e. `svn status` in `~/cs161/src` should be empty or full of things you know you didn't change or don't want to add, then follow the following steps.

1. Export a copy of your tree into a temporary directory:

   ```
   % cd ~/tmp/ # Create this directory if it doesn't exist
   % svn export http://minitrue/repos/<username>/trunk <username>-asst0
   ```

2. The step above creates a "clean" (i.e. free of subversion control files) copy of the tree currently in the repository in `~/tmp/<username>-asst0`. Testing this release helps you pick up files you may have forgotten to add or changes you haven't yet committed.
3. Enter your `~/tmp/<username>-asst0` directory and make sure that everything is as it should be, particularly that you can build and run your kernel! (You may want to pass a different `--ostree=<path>` argument to the top level `./configure` in order to have the built binaries placed somewhere other than `$HOME/cs161/root`) Once you are finished, clean up by destroying the contents of the `~/tmp/` directory, i.e. `rm -rf ~/tmp/*`.
4. Tag your repository for the end of asst0:

   ```
   % svn copy http://minitrue/repos/<username>/trunk http://minitrue/repos/<username>/tags/asst0-end
   ```

   Remember the tags directory we created earlier? This is versioned like any other part of your Subversion repository. The step above simply copies from `/trunk/` into `tags/asst0-end`. It's also repeatable, so if you realized later you have changes yet to commit you can repeat the copy using a new version of `/trunk/`.
5. Finally, export and `tar` up a copy of your tagged tree in your submission directory:

   ```
   % cd ~/submit/asst0
   % svn export http://minitrue/repos/<username>/tags/asst0-end <username>-asst0
   % tar -czf <username>-asst0.tgz <username>-asst0
   ```

6. After you tar your release, be sure to remove the `~/submit/asst0/<username>-asst0` your `svn export` created:

   ```
   % rm -rf <username>-asst0
   ```

What (and how) to hand in

In CS161 this semester, we are experimenting with using Subversion to allow you to submit your assignments. Like your codebase itself, this means that you can submit multiple times, version your submissions, and this will better allow us to collect and collaborate over grading your assignments.

Your `~/submit/asst0` directory should contain everything you need to submit, specifically:

- `setup.script`
- `svninit.script`
- `code-reading.txt`
- `build.script`
- `run.script`
- `newbuild.script`
- `gdb.script`
- `svn-use.script`

- `sys161.conf`: Copy this from `~/cs161/root`.
- `<username>-asst0.tgz`

Review the document Preparing your Assignments for Submission to learn how to properly submit your assignment. Specifically, for this assignment, and for a first-time submit:

```
% cd ~/submit/asst0/
% svn add *
% cd ..
% svn ci -m "ASST0 submission"
```

Once you have added `~/submit/asst0`, further work in this directory needs to be committed like anything else under version control.