Operating Systems

ECE344, WINTER 2014
UNIVERSITY OF TORONTO

ASSIGNMENT 0: AN INTRODUCTION TO OS161

Release date: Jan 17.
Due date (hard deadline, no extension): Jan 31, 11:59 pm.
TA responsible for this Lab: Yongle Zhang (zhangyongle dot kevin at gmail dot com).

Objectives

After this assignment, you should:

- Be familiar with Subversion (SVN) and GDB (the GNU Debugger).
- Understand the source code structure of OS161, the software system we will be using this term.
- Understand how System/161 emulates the MIPS hardware environment on which OS161 runs.
- Understand the source code structure of System/161.
- Be comfortable reading the OS161 source code and figuring out where things are done and how they are done.
- Write some initial testing code to familiarize yourself with making changes to the OS161 environment.
- Be prepared to undertake Assignment 1.

Introduction

In this assignment, we will introduce:

System/161
The machine simulator for which you are building an operating system this term.

OS161
The operating system you will be designing, extending, and running this term.

Subversion (SVN)
SVN is a source code revision control system. It manages the source files of a software package so that multiple progra
may work simultaneously. Each programmer has a private copy of the source tree and makes modifications independer
main copy of the source tree is stored in an area called the SVN repository. The private copy of the code is called the w
copy. Each programmer makes modifications to their working copy and then commits their modifications to the SVN rep
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 co
manner and view and set the values of variables. In the case of OS161, it allows you to debug the operating system you
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 finc
detailed information on SVN and GDB. The following sections provide instructions on what you must do for this assignment.

What are OS161 and System/161?

The code is divided into two main parts:

- **OS161:** 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
  is about writing operating systems, not designing or simulating hardware. Therefore, you may not change the machine
  simulator.

The OS161 distribution contains a barebones operating system source tree, including some utility programs and libraries. Aft
build the operating system you boot, run, and test it on the simulator.

We use a simulator in OS161 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, v
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.

Besides this initial assignment, there will be an OS161 programming assignment for each of the following topics:

- ASST1 : Synchronization and concurrent programming
- ASST2 : System calls and multiprogramming
- ASST3 : Virtual memory

OS161 assignments are cumulative. You will need to build each assignment on top of your previous submission. So you will h
make sure that each of your assignments work correctly!

About SVN

Most programming you have probably done has been in the form of 'one-off' assignments: you get an assignment, you compl
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 rel

regular intervals. This kind of development normally requires multiple people working simultaneously within the same code ba necessitates a system for tracking and merging changes. Your will be working in teams of 2 on OS161 and will be developing base that will change over the couse of several assignments. Therefore, it is imperative that you start becoming comfortable v SVN, an open source version control system.

SVN is a powerful tool, but for OS161 you only need to know a subset of its functionality. The SVN handout contains all the information you need to know and should serve as a reference throughout the term. 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 c crash the whole machine, necessitating a time-consuming reboot. The use of a machine simulator such as System/161 provic several debugging benefits. First, a kernel crash will only crash the simulator, which only takes a few keystrokes to restart. Se the simulator can sometimes provide useful information about what the kernel did to cause the crash, information that may or be easily available when running directly on top of real hardware.

To debug OS161 you must use a version of GDB configured to understand OS161 and MIPS. This is called `cs161-gdb`. This v of GDB has been patched to be able to communicate with your kernel through System/161.

An important difference between debugging a regular program and debugging an OS161 kernel is that you need to make sur 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 d your operating system and a brief introduction to GDB.

Setting up your account

Login to the lab machines. The OS161 tools are accessible from the workstation lab machines (`ug*.eecg.utoronto.ca`). The r machines that are accessible are roughly `ug51-ug100`, `ug132-ug180` and `ug201-250`. You will need to setup your path to access OS161 tools. To do so, you should check whether you use `csh` or `bash` as follows:

```
% echo $0
```

If the output shows `sh` or `bash`, then you are using `bash`. If the output shows `csh` or `tcsh`, then you are using `csh`.

1. For `csh`, add the following to the end of your `~/.cshrc`:

   ```
   set path=( /cad2/ece344s/cs161/bin $path)
   ```

2. For `bash`, add the following to the end of your `~/.bashrc`:

   ```
   export PATH=/cad2/ece344s/cs161/bin:$PATH
   ```

Then log out and log back in. Run `echo $PATH` and you should see the new path.

Getting the distribution and setting up your SVN repository

These instructions should be performed by one partner only. We will describe what the other partner needs to do below.

1. First, download the OS161 source into your home directory. If you are working remotely, you can use the `wget URL` com a workstation machine to download the OS161 sources directly to the workstation machine.

   In addition to OS161, you can also download the distributions for System/161, the machine simulator, and the OS161 to If you are developing on the lab machines, you **do not** need these additional files, as they are already installed. If you w develop on your home machine at home, you will need to download, build, and install this package as well. Note that we provide support for installing this package. Also, you must ensure that your assignment works on the lab machines.

2. Make a directory in which you will do all your work. For the purposes of the remainder of this assignment, we'll assume will be called `~/ece344`.

   ```
   % mkdir ~/ece344
   % cd ~/ece344
   % mv ../os161-1.11.tar.gz .
   ```

3. Unpack the OS161 distribution by typing

   ```
   % tar xzf os161-1.11.tar.gz
   ```

   This will create a directory named os161-1.11

4. Rename your OS161 source tree to just os161. You can also (optionally) remove the `os161-1.11.tar.gz` tarball from thi directory.

   ```
   % mv os161-1.11 os161
   ```

5. Each of you has been assigned a group number. You can find out group number by typing:

   ```
   % groups
   ```

   This command tells you all the Unix groups to which you belong. One of them should be in the range from `os-001` to `os-`

   This is your group number (or group id) for this course. Below, we use `GRP_NR` to denote this group number.

6. You will first need to setup your SVN repository. This repository will contain your OS161 code and you will use it to shar code with your partner. Your repository will be located on `ug250.eecg.utoronto.ca`, under the `/svn/GRP_NR/svn` directory

   To create your repository, first download the svn-setup.sh script in the `~/ece344` directory. Then run this script on the SV repository server machine:

   ```
   % cd ~/ece344
   % chmod +x ./svn-setup.sh
   % ssh ug250.eecg.utoronto.ca ~/ece344/svn-setup.sh GRP_NR
   ```

If this command returns an error, either you did not specify `GRP_NR` correctly, or your SVN repository exists already (e.g., partner has already created it). If you really need to remove your repository (generally, a really bad idea), you will need into the SVN repository machine and remove the repository directory.

7. Your SVN repository should now be located at `svn+ssh://ug250.eecg.utoronto.ca/svn/GRP_NR/svn`. Note that `GRP_NR` is group number and should lie between `os-001` to `os-040`. In the document below, this repository path is referred to as `$ECE344_SVN`. You can setup the `ECE344_SVN` environment variable as follows:
    1. For `csh`, add the following to the end of your `~/.cshrc`:

        ```
        setenv ECE344_SVN svn+ssh://ug250.eecg.utoronto.ca/svn/GRP_NR/svn
        ```

    2. For `bash`, add the following to the end of your `~/.bashrc`:

        ```
        export ECE344_SVN=svn+ssh://ug250.eecg.utoronto.ca/svn/GRP_NR/svn
        ```

8. Run

    ```
    % svn ls $ECE344_SVN
    ```

    and you should see

    ```
    tags/
    trunk/
    ```

9. Change directories into the OS161 distribution that you unpacked in the previous section and import your source tree.

    ```
    % cd ~/ece344/
    % svn import os161 $ECE344_SVN/trunk/ -m "Initial import of os161"
    ```

    You can alter the arguments as you like; here's a quick explanation. `-m "Initial import of os161"` is the log message SVN records. (If you don't specify it on the command line, it will start up a text editor). `/trunk` is where SVN will put the f within your repository. `$ECE344_SVN/trunk` is the SVN URL you will specify when you check out your system. Run `svn ls` URL and you will see that the os161 directories have been imported into your SVN repository.

10. 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 copy of the source tree that is yours to work on. You can safely remove the original tree.

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

    ```
    % cd ~/ece344/
    % svn co $ECE344_SVN/trunk os161
    % cd os161
    ```

    The `svn co` command creates a **working copy** of your tree that you can safely modify in `~/ece344/os161`.

12. Now, we will ignore various generated files in svn. To do so, run the `svn-ignore.sh` script as described here. This script

    to ignore files that will be generated when we compile the OS161 sources.

13. Now run the command `svn status` in the `~/ece344/os161` directory. The svn status command shows the status of files a directories.

    ```
    % cd ~/ece344/os161
    % svn status
    M       .
    M       kern/compile
    M       lib/hostcompat
    ...
    ```

    The `M` character in the first column shows that the corresponding file or directory has been modified. In the output above directories have been modified in the working copy because we have added the `svn:ignore` property to these directorie various files that will be generated in these directories.

14. Now commit the modifications made to the working copy by using `svn commit`. This command will commit your changes repository. The command invokes an editor so that you can add a log message. For short commit messages, you can u `m` option of commit.

    ```
    % cd ~/ece344/os161
    % svn commit -m "committing directories with ignored files"
    ```

    Running `svn status` again should show no output, indicating that the working copy is consistent with the repository.

15. Now use the `svn copy` command to tag the current version of the repository so that you can later use this version with `s` and other commands. We have provide more information about svn tags. Or you see the svn manual.

    ```
    % cd ~/ece344/os161
    % svn copy -m "starting assignment 0" $ECE344_SVN/trunk  $ECE344_SVN/tags/asst0-start -m "Tagging initial
    ```

16. If you ever have serious problems with your SVN repository (e.g., you have accidentally removed critical files from the repository), you will need to remove the repository directory and redo all the steps in this section.

Checking out code from SVN

After the SVN repository is setup, the second partner can checkout a copy of the OS161 code in their own directory as follow

1. You will first need to create the `~/ece344` directory.
2. Follow step 7 above.
3. Follow step 11 above.

Code reading

One of the challenges of OS161 is that you are going to be working with a large body of code that was written by someone el When doing so, it is important that you grasp the overall organization of the entire code base, understand where different pie functionality are implemented, and learn how to augment it in a natural and correct fashion. As you and your partner develop although you needn't understand every detail of your partner's implementation, you still need to understand its overall structu

although you needn't understand every detail of your partner's implementation, you still need to understand its overall structur 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, 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 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 review every line of code in the system in order to answer all the questions, we strongly recommend that you look over at leas 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 make intelligent design decisions when you approach future assignments. This may seem tedious, but if you understand how system fits together now, you will have much less difficulty completing future assignments. Also, it may not be apparent yet, b have much more time to do so now than you will at any other point in the term.

The file system, I/O, and network sections may seem confusing since we have not discussed how these components work. H 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 unders 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 OS161 source, though yo have to look elsewhere (such as Tanenbaum) for some background information. Make sure that you can answer these questi during evaluation.

Top level directory

In the top-level os161 directory (created by the checkout above), you will find the following files:

`configure`: top-level configuration script; configures the OS161 distribution, including all the provided utilities, but does not co the operating system kernel.

`Makefile`: top-level makefile; builds the OS161 distribution, including all the provided utilities, but does not build the operating kernel.

You will also find the following directories:

`kern`: the kernel source code.

`lib`: user-level library code lives here. We have only two libraries: `libc`, the C standard library, and `hostcompat`, which is for recompiling OS161 programs for the host UNIX system. There is also a `crt0` directory, which contains the startup code for us programs.

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

`testbin`: these are pieces of test code.

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

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

`man`: the OS161 manual ("man pages") appear here. The man pages document (or specify) every program, every function in library, and every system call. You will use the system call man pages for reference in the course of assignment 2. The man p are HTML and can be read with any browser.

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

You needn't understand all the files in bin, sbin, and testbin now, but you certainly will later on. In fact, you will be adding a di to testbin in this assignment. Eventually, you will want to modify other files in these directories and/or write your own utilities a these are good models. Similarly, you need not read and understand everything in lib and include, but you should know enou 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 t 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 where architecture-specific code goes. By architecture-specific, we mean the code that differs depending on the hardware pla on which you're running. There is one directory here: `mips` which contains code specific to the MIPS processor.

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

Question 0. What is the default compile option that we use for OS161'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 OS161?

Question 2. What are some of the details which would make a function "machine dependent"? Why might it be important to r 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. M 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?

Question 4. What bus/busses does OS161 support?

`kern/compile`: This is where you build kernels. In the compile directory, you will eventually find one subdirectory for each kerr want to build. In a real installation, these will often correspond to things like a debug build, a profiling build, etc. In our world, e build directory will correspond to a programming assignment, e.g., ASST1, ASST2, etc. These directories are created when y configure a kernel (described in the next section). This directory and build organization is typical of UNIX installations and is r universal across all operating systems. `kern/conf: config` is the script that takes a config file, like ASST1, and creates the corresponding build directory (shown later).

`kern/include`: These are the include files that the kernel needs. The `kern` subdirectory contains include files that are visible n to the operating system itself, but also to user-level programs. (Think about why it's named "kern" and where the files end up 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 OS161?

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

Question 12. What version of OS161 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/main`: 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/asst1`: This is the directory that contains the framework code that you will need to complete assignment 1. You can safe ignore it for now.

`kern/userprog`: This is where you will add code to create and manage user level processes. As it stands now, OS161 runs or 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 ir

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

`kern/dev`: This is where all the low level device management code is stored.

Question 17. Look at how `getch` is implemented. It is the function for reading a character from the terminal. Which function in kernel will the hardware call when a character is received from the terminal?

`kern/fs`: The file system implementation has two directories. We'll talk about each in turn. `kern/fs/vfs` is the file-system inde layer (`vfs` stands for "Virtual File System"). It establishes a framework into which you can add new file systems easily. You wi 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 augmer 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 pur the routines in that file?

Building a kernel

Now it is time to build a kernel. You will need to configure a kernel and then build it.

1. Configure your tree for the machine on which you are working.

   ```
   % cd ~/ece344/os161
   % ./configure --ostree=$HOME/ece344/root
   ```

   Note the use of `$HOME` instead of `~`. The `--ostree` option specifies the root of the OS tree. All programs will be installed th after the next step. Within the simulator, this root will be accessible as the `/` directory. For example, programs in the `bin` directory will be installed in `~/ece344/root/bin` and accessible as `/bin` within the simulator. `./configure --help` explains configure options.

2. Now let's build and install the user level utilities. If you have any compilation issues, make sure that your `$PATH` variable correctly as described above.

   ```
   % make
   ```

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

```
% cd ~/ece344/os161
% cd kern/conf
% ./config ASST0
```

This will create the ASST0 build directory in `kern/compile`. The next step will actually build a kernel in this directory. Not you should specify the complete pathname `./config` when you configure OS161. If you omit the `./`, you may end up rur configuration command for the system on which you are building OS161, and that is almost guaranteed to produce rath strange results!

4. Build and install the ASST0 kernel.

```
% cd ~/ece344/os161/kern
% cd compile/ASST0
% make depend
% make
% make install
```

Running your kernel

1. Change to your root directory. Copy the default simulator configuration file into the root directory.

```
% cd ~/ece344/root
% cp /cad2/ece344s/cs161/bin/sys161.conf.sample sys161.conf
```

You should take a look at this file, as it describes how to configure the simulator you will be running your code in.

2. Run the machine simulator on your operating system.

```
% sys161 kernel
```

3. At the OS161 command prompt, run the `poweroff` command that tells the system to shut down as follows.

```
OS/161 kernel [? for menu]: p /sbin/poweroff
```

Note that the `p` command in OS161 is used to run a program.

Practice modifying your kernel

1. Create a file called `~/ece344/os161/kern/main/hello.c`.
2. In this file, write a function called `hello` that uses `kprintf()` to print "Hello World\n".
3. Edit `kern/main/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` include. For example, could place it in `kern/include/lib.h`. If you do so, you may need to include some header files other than `lib.h` in `hello` 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 `kern/conf/`.
6. Reconfigure and rebuild your kernel. Note that you should only need to reconfigure the kernel (with the `./config` comma shown earlier) when you add or remove files from the kernel. Otherwise, running `make` and `make install` is sufficient for rebuilding the kernel.
7. Make sure that your new kernel runs and displays the new message.

Using GDB

1. For using gdb, you will need two windows (terminals). If you are logged in remotely, now is a good time to learn about th screen command, which will make it easier for you to work with multiple windows. Run the kernel in gdb by first running kernel in the run window, and then attach gdb to the kernel from the debug window.

```
(In the run window:)
% cd ~/ece344/root
% sys161 -w kernel

(In the debug window:)
% cd ~/ece344/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
```

Practice with SVN (you should do it only after Jan 22nd).

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 SVN.

1. First, make sure that you have completed updating your working copy to include the necessary files from the "Hello Wor exercise above. `svn status` is your friend here. Run this command in `~/ece344/os161`. You should only see files that you modified or added to your working copy.
2. Change directory to `kern/main/` and add a comment with your name to `main.c`.
3. From within your `~/ece344/os161/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 cheat sheet.

4. Execute

```
% cd ~/ece344/os161/kern
% svn diff main/main.c
```

to display the differences in your version of this file.

5. Now commit your changes using `svn commit` (from the `kern` directory). This command will commit your changes to your repository. The command invokes an editor so that you can add a log message. For short commit messages, you can u `m` option of commit.
6. Remove the first 100 lines of `main.c`.
7. Try to build your kernel (this ought to fail).
8. Realize the error of your ways and get back a good copy of the file.

```
% cd ~/ece344/os161/kern; svn revert main/main.c
```

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

```
% cd ~/ece344/os161/kern
% svn diff
```

Changing the OS menu

When you run your kernel under the simulator, typing `?` shows a kernel menu. This menu allows running various commands. example, `?t` shows various tests that you can run.

In this part of the lab, you will add some new options to the OS menu.

The `DEBUG` macro uses the `dbflags` variable in the kernel. Depending on the value of this variable, different types of debugging messages are printed. For example, if its value is 0x012, then `DB_SYSCALL` and `DB_THREADS` messages are printed. Why?

The problem is that if you want to see different types of messages, the `dbflags` variable has to be changed and the kernel has recompiled. The reason you may want to see different types of messages is that printing all types of messages may make it h debug a specific problem.

Your task is to allow changing the value of the `dbflags` variable from the OS menu.

First, find out the initial value of this variable. Then change the OS menu code (where is it located?) so that the menu output i follows:

```
OS/161 kernel [? for menu]: ?o

OS/161 operations menu
    [s]       Shell                     [pf]      Print a file
    [p]       Other program             [cd]      Change directory
    [dbflags] Debug flags               [pwd]     Print current directory
    [mount]   Mount a filesystem        [sync]    Sync filesystems
    [unmount] Unmount a filesystem      [panic]   Intentional panic
    [bootfs]  Set "boot" filesystem     [q]       Quit and shut down

Operation took 0.058339600 seconds
OS/161 kernel [? for menu]: dbflags

OS/161 Debug flags
    [df 1 on/off]        DB_LOCORE      [df 7 on/off]        DB_EXEC
    [df 2 on/off]        DB_SYSCALL     [df 8 on/off]        DB_VFS
    [df 3 on/off]        DB_INTERRUPT   [df 9 on/off]        DB_SFS
    [df 4 on/off]        DB_DEVICE      [df 10 on/off]       DB_NET
    [df 5 on/off]        DB_THREADS     [df 11 on/off]       DB_NETFS
    [df 6 on/off]        DB_VM          [df 12 on/off]       DB_KMALLOC

Current value of dbflags is 0x0
Operation took 0.058040000 seconds
```

Note that `?o` produces a new option `[dbflags]`. Typing `dbflags` produces the new menu. With this menu, typing `df 5 on` will tu `DB_THREADS` messages, and typing `df 5 off` will turn off these messages.

```
OS/161 kernel [? for menu]: df 5 on
Operation took 0.000024960 seconds
OS/161 kernel [? for menu]: dbflags

OS/161 Debug flags
    [df 1 on/off]     DB_LOCORE         [df 7 on/off]     DB_EXEC
    [df 2 on/off]     DB_SYSCALL        [df 8 on/off]     DB_VFS
    [df 3 on/off]     DB_INTERRUPT      [df 9 on/off]     DB_SFS
    [df 4 on/off]     DB_DEVICE         [df 10 on/off]    DB_NET
    [df 5 on/off]     DB_THREADS        [df 11 on/off]    DB_NETFS
    [df 6 on/off]     DB_VM             [df 12 on/off]    DB_KMALLOC

Current value of dbflags is 0x10
Operation took 0.062876880 seconds
```

If you type `df 3 on` after the code shown above, the dbflags value should then be 0x14. This part of the lab will help you exer manipulation skills, which will be useful in later labs.

Your code should also ensure that the arguments to `df` are passed correctly, or else your code should print the following in a separate line:

```
Usage: df nr on/off
```

Assignment submission

Finally, you need to submit your code for this assignment.

1. Once you are confident that you have completely done your assignment, run `make clean` from the `os161` directory. This v all generated files. Then use `svn status` in the `os161` directory to find out the status of all files. Any files that have a `?` be

them are not in the repository. If you have created these files by hand, then add them to the repository using svn add. If are generated files, use the instructions here to ignore these files.

2. Then run `svn commit` from the `os161` directory so that all modified files are checked in your repository. Use `svn status` in `os161` directory again to make sure that all your modified source files are properly committed. Make sure that your partn changes are also committed.
3. Tag your repository for the end of asst0:

   ```
   % svn copy -m "ending assignment 0" $ECE344_SVN/trunk $ECE344_SVN/tags/asst0-end
   ```

   Remember the tags directory we created earlier? This is versioned like any other part of your SVN repository. The step simply copies from `/trunk/` into `tags/asst0-end`.

Testing your assignment with the autotester

Please read the instructions for testing your code.