Advanced Operating Systems

Computer Science Department
Indiana University
Bloomington, IN 47405

Main Menu

- Home
- Contact
- About
- Schedule
- Assignments
- Resources
- Fun

The Tao

A novice asked the master: "I have a program that sometimes runs and sometimes aborts. I have followed the rules of programming, yet I am totally baffled. What is the reason for this?"

The master replied: "You are confused because you do not understand the Tao. Only a fool expects rational behavior from his fellow humans. Why do you expect it from a machine that humans have constructed? Computers simulate determinism; only the Tao is perfect.

The rules of programming are transitory; only the Tao is eternal. Therefore you must contemplate the Tao before you receive enlightenment."

"But how will I know when I have received enlightenment?" asked the novice.

"Your program will then run correctly," replied the master.

Programming Assignment 0

Assigned Aug 20, 2012

Due Aug 27, 2012

Note This problem set is to be done individually.

Introduction

This assignment will familiarize you with OS/161, the operating system with which you will be working this semester, and System/161, the machine simulator on which OS/161 runs. These tools were developed at Harvard University for their CS161 course (and are used here with permission). As a result, you will sometimes see reference to CS161 as all inclusive of the OS, VM, and associated tools.

We also introduce tools that will make your work this semester easier:

Git
: Git is a version control system intended as an alternative to Subversion and CVS. 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. Git attempts to intelligently merge multiple people's modifications, highlighting potential conflicts when it fails.

IU GitHub
: IU GitHub is a web-based interface for collaborating on software and documents that provides Git version control, issue tracking, and wiki space for hosted projects. You should be able to log in with your IU credentials and start sharing projects with other IU students.

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 Git 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.

Logging in to the CS Servers Remotely

— Geoffrey James, *The Tao of Programming*

You can do all of your work locally on a Linux CS machine. If you want to log in remotely, however, you can do so using the following command at a terminal on your home machine:

```
% ssh USERNAME@tank.cs.indiana.edu
```

where `USERNAME` is your IU user name. Your password will be the same one you use for OneStart and other IU services.

To send files from your local machine to your CS home directory, use the `scp` utility:

```
% scp myfile USERNAME@tank.cs.indiana.edu:
```

Windows Users

If you are on a Windows machine, you will need two pieces of software:

1. PuTTY - a free SSH client
2. WinSCP - a free SFTP client

For both pieces of Windows software above, use `tank.cs.indiana.edu` as the server and your IU username/password.

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 which emulates the physical hardware on which your operating system will run. This course is about writing operating systems, not designing or simulating hardware. Therefor 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.

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 P536 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 R3000 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 : Distributed shared memory

P536 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 AI for a copy of the appropriate solution set, even if we have not yet released it. We will publish the solution set when all groups have submitted, but depending on when late days are used some groups my finish before others.

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 groups to refrain from using the solution sets except in the most dire circumstances.

About Git

Most programming assignments you have done in college have been 'one-shots': 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 ASST2 you will work in teams on OS/161, and therefore it is imperative that you start becoming comfortable with Git and IU GitHub.

Git is very powerful, but for P536 you only need to know a subset of its functionality. The free online book Pro Git contains much more information than you will need for this class, but reading the first two chapters (and working through this assignment) should get you up to speed with the basics.

If you are familiar with Subversion already, Git will be similar with one major exception. When you commit using git, your changes are kept on your local machine until you explicitly run `git push origin master`.

This last point is very important! Running `git commit` will save changes only to your local machine. You **must** use `git push` to push changes to IU GitHub.

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 like 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.

You must use the CS161 version of GDB to debug OS/161. You can run on the UNIX systems used for the course as `cs161-gdb`. This copy of GDB has been configured for MIPS and 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 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. Detailed instructions on how to debug your operating system and a brief introduction to GDB are contained in the handout Introduction to GDB for CS161 .

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. In the department, the default shell is "bash" and following information is intended for that shell. (If you use a different shell, then we trust you'll know what to do).

1. Add the following lines to ~/.bashrc file:

```
if [ -f ~p536/cs161.bashrc ]; then
  source ~p536/cs161.bashrc
fi
```

   This will set some path variables for you and should occur after you set your path variables in your own .bashrc file.

2. Log out and back in.

Getting the Distribution

Download the OS161 source here or run the following command on your Linux CS machine:

```
% wget ftp://ftp.eecs.harvard.edu/pub/os161/os161-1.11.tar.gz
```

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 Linux environment in the CS Department, you do not need these files. If you wish to develop on your home machine at home, you will need to download, build, and install these packages as well. The instructional staff does not offer support for anything but the Linux environment on CS Department machines.

1. Script the following session using the `script` command.
2. Make a directory in which you will do all your OS/161 work. For the purposes of the remainder of this assignment, we'll assume that it will be called `cs161`.

```
% mkdir cs161
% cd cs161
% mv ../os161-1.11.tar.gz .
```

3. Unpack the OS/161 distribution by typing

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

   This will create a directory named os161-1.11.

4. Rename your OS/161 source tree to just `src`.

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

5. Delete the tar file

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

6. End your script session by typing `exit` or by pressing Ctrl-D. Rename your typescript file to be `setup.script`.

```
% mv typescript ~/cs161/setup.script
```

Connecting to IU GitHub

Before setting up your Git repository, we need to make sure you are connected to IU GitHub. Once you have everything set up, you will be able to easily share code with other IU users using Git.

Step 1 - Create an SSH Key

(Note: More detailed instructions are available on GitHub's help page.)

Log into a Linux CS machine and run the following commands in your home directory:

```
% ssh-keygen -t rsa -C "USERNAME@indiana.edu"
```

Accept the default location by pressing Return, and then press Return twice more to accept an empty passphrase:

```
% Generating public/private rsa key pair.
% Enter file in which to save the key (/u/USERNAME/.ssh/id_rsa): <RETURN>
% Enter passphrase (empty for no passphrase): <RETURN>
% Enter same passphrase again: <RETURN>
```

Now that your SSH key has been generated, you will need to copy and paste it into GitHub. Run the following command to display the key:

```
% cat ~/.ssh/id_rsa.pub
```

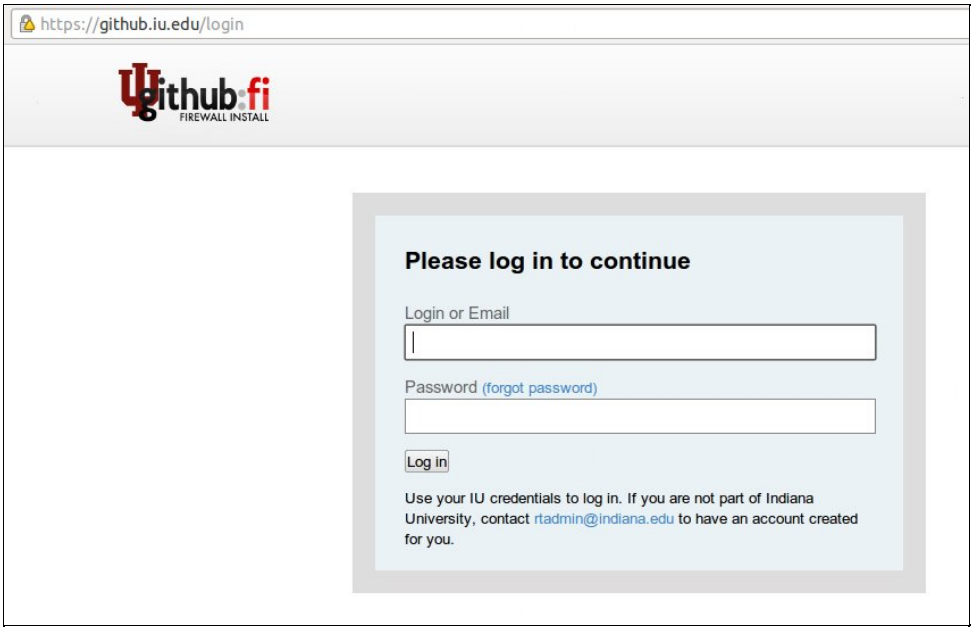
You should get a printout *similar* to this:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAqvEbXQY/KDU+Z1H+m1JOX6A9AzdnsWXJ4RUhpAyKDkFb
6F1nu33Nu9mcmgsJOyq+0cIxKH9yUxbWnQJg0dqq9Dk6DaCKDcS+W91Sx3PYJJt1b/td4/FIcoWGCZDU
f3LT31qm10tTel2N9R5NmH6FmoobtWbPQcScKHO9+x5Tg3Xpo6428tx8VI+lHngEQxCDxPIZr5JLpBhz
kLiDfppAgIZT4wiZCfZvu0YMQrYr8hiQWfu6R9gNbH5EiEOVFiIgkFj1/yt0iS353viaEVUXjocnl+g/
zqb6Bsd/V1Sv78DfNEz8S8J+/9iGZz93D5kL4vf3kbGHpZ/myoe3xpuBdw== mihansen@indiana.edu
```

Keep this printout close by. You will need to copy and paste this text **exactly** into GitHub in a moment.

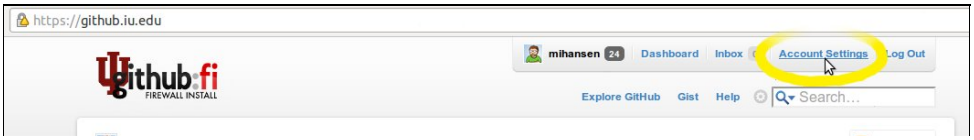
Step 2 - Log in to IU GitHub

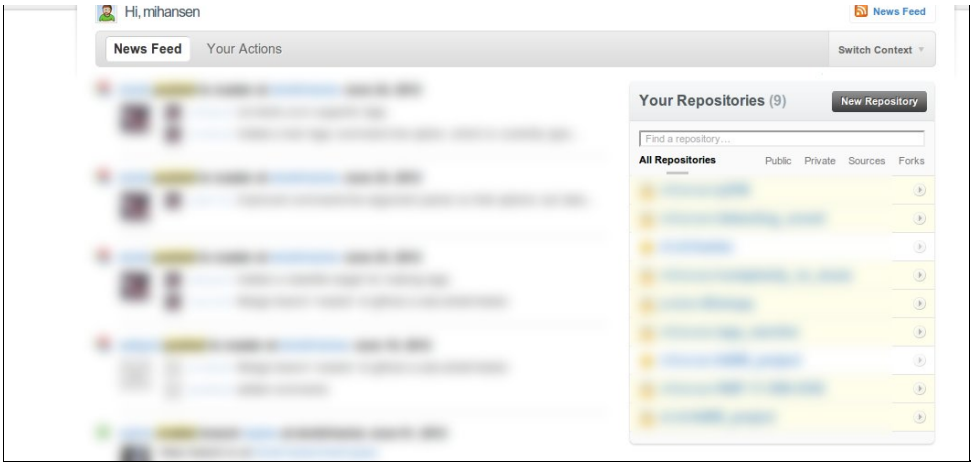
Visit http://github.iu.edu with your favorite web browser and log in using your IU user name and password (the same password you use for OneStart).



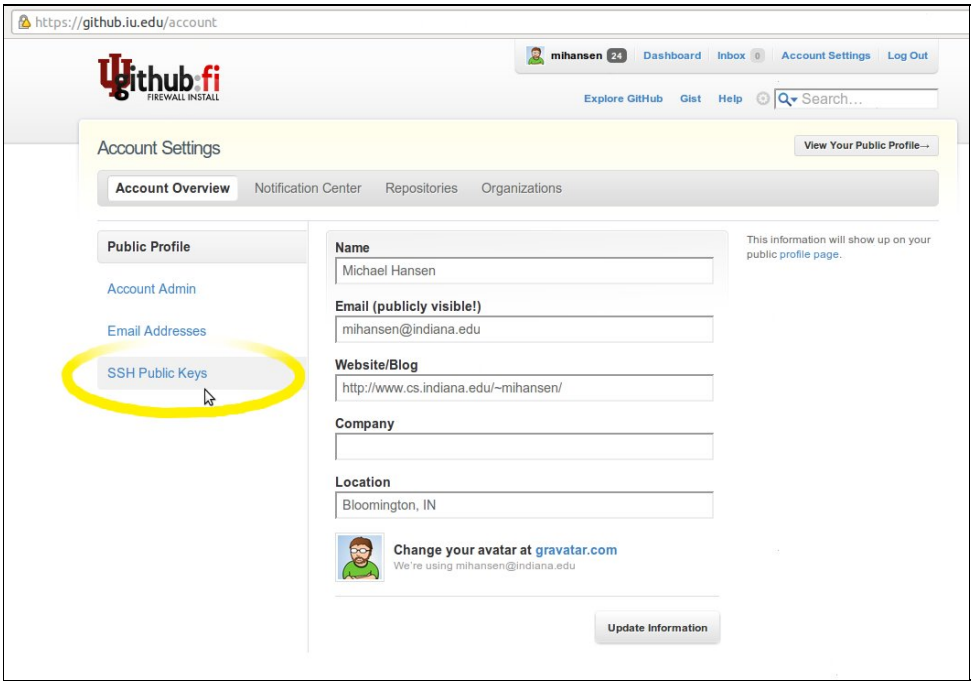
Step 3 - Add a new SSH Key to GitHub

Once you're logged in to GitHub, click on the `Account Settings` link.

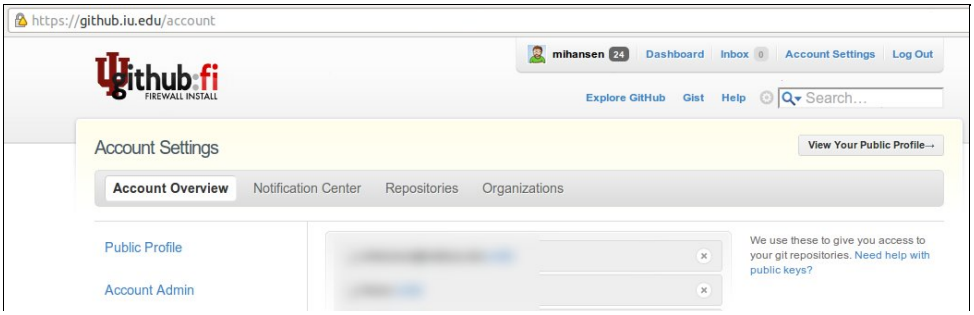


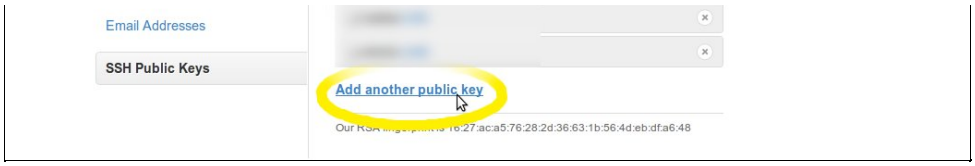


In Account Settings, click on the `SSH Public Keys` link.

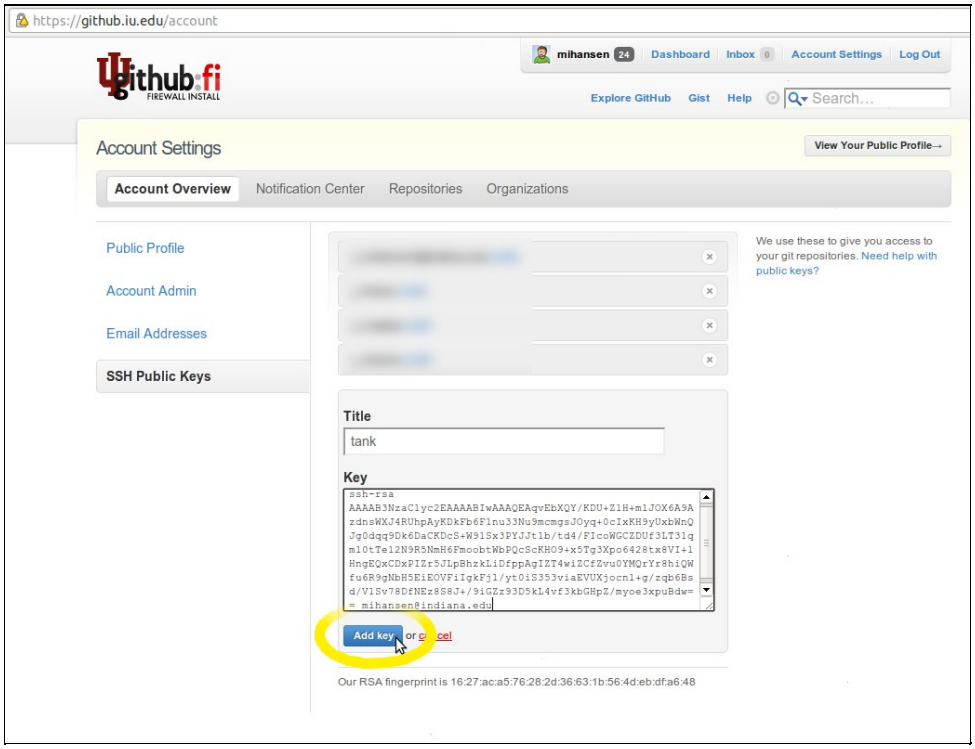


At the bottom of the SSH Public Keys page, click on the `Add another public key` link.





Enter a title (I called my key `tank`), paste in the printout from step 1, and click `Add key`.



You should now be able to use IU GitHub from a Linux CS machine (see next section). If you received any errors, please contact the associate instructor.

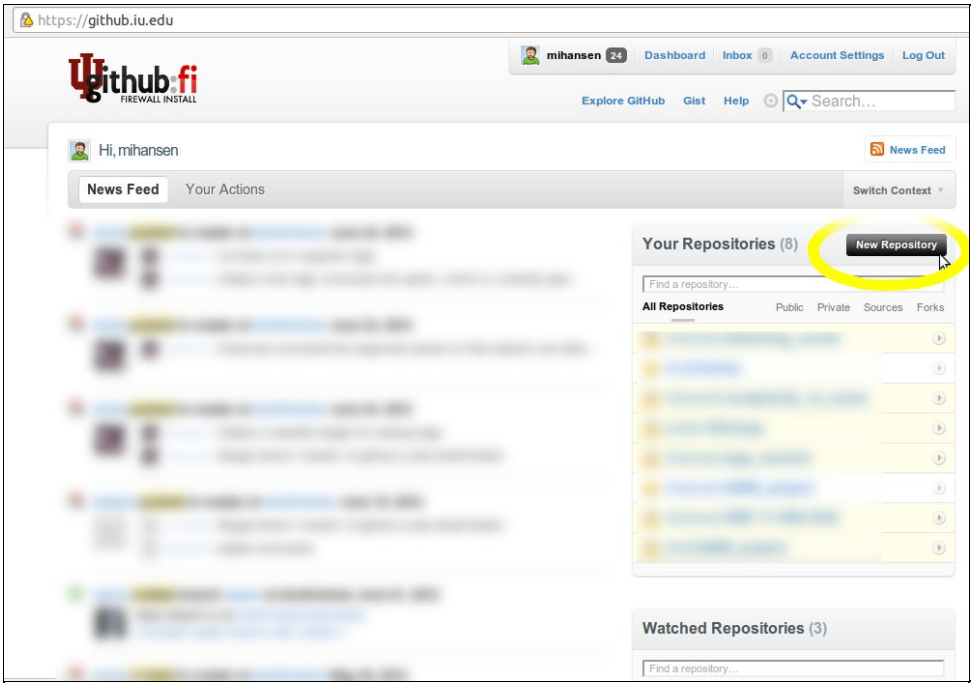
Setting up your Git repository

Step 1: Create a New Repository

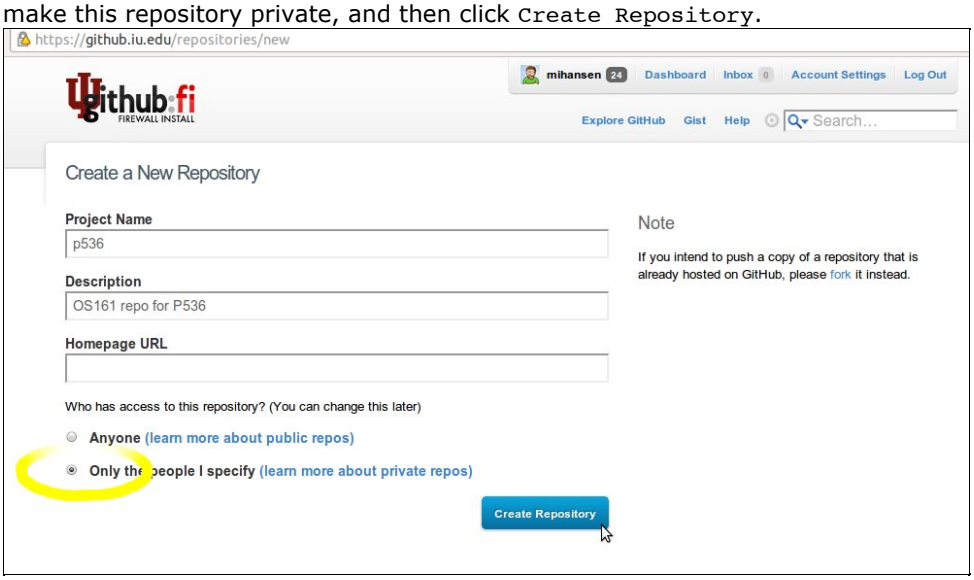
You will now create a new repository in IU GitHub.

Visit http://github.iu.edu with your favorite web browser and log in.

On your IU GitHub dashboard, click the `New Repository` button.



Name the project `p536` and give it a description. Choose "Only the people I specify" in order to make this repository private, and then click `Create Repository`.

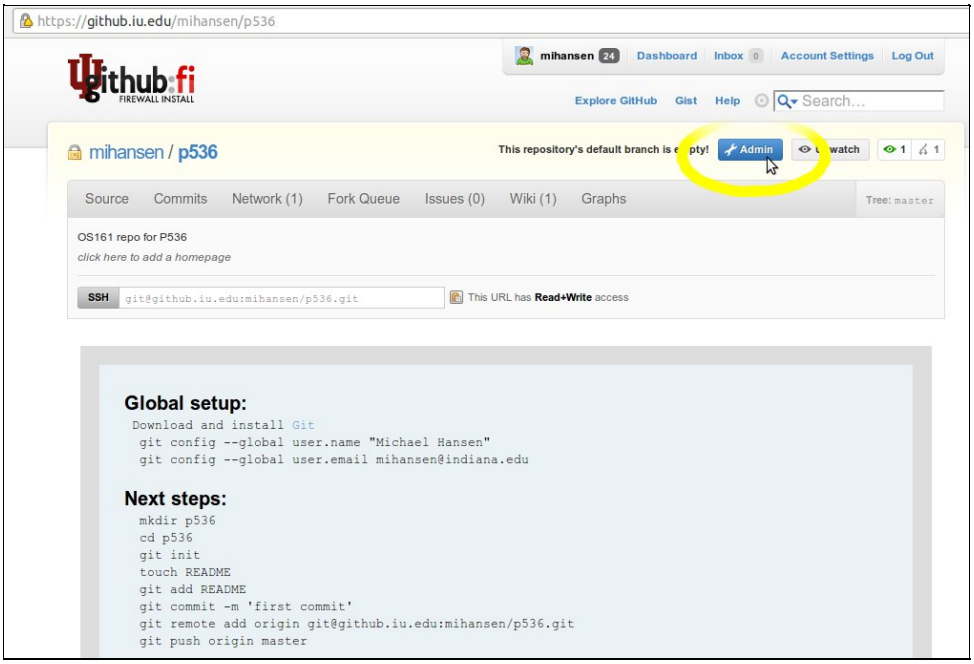


Step 3: Add a Collaborator

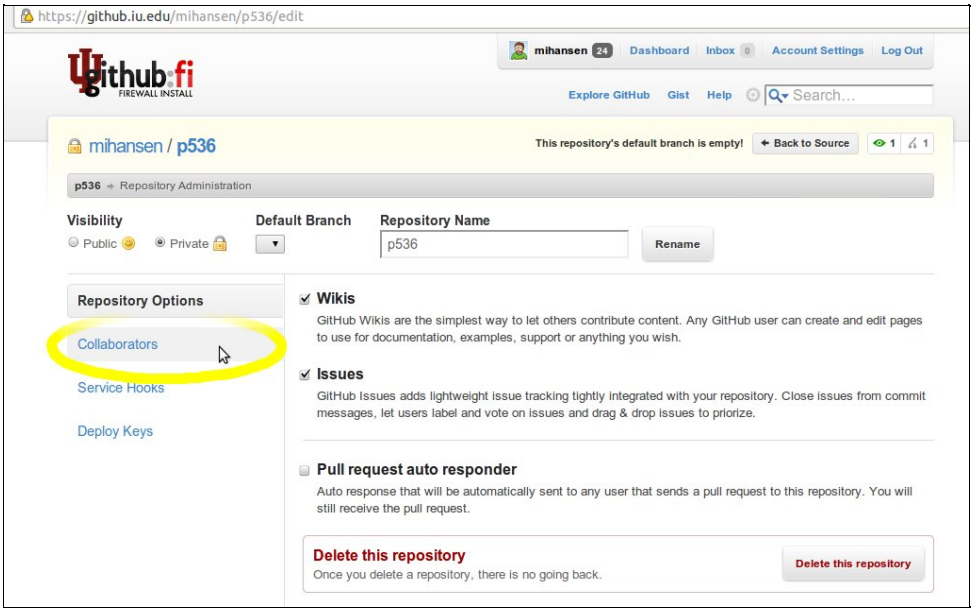
We will need access to your p536 repository in order to grade your assignments. You **must** add

the associate instructor (user `mihansen`) as a collaborator on your project.

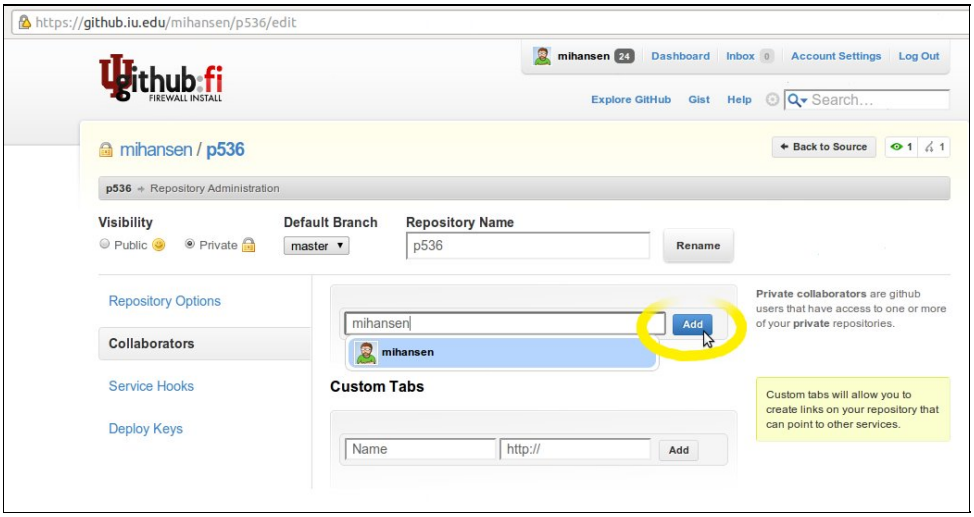
On your p536 project page, click on the `Admin` button



Click on the `Collaborators` link under Repository Options.



Enter the associate instructor's IU user name (`mihansen`) and click the `Add` button.



That's it! You're ready to start adding files to your repository.

Step 2: Your First Commit

1. Back on your Linux CS machine, script the following session using the `script` command.
2. Throughout the rest of this assignment, we will assume that you are using the repository that was created in step 1.
3. Change directories into the OS/161 distribution that you unpacked in the an earlier section and import your source tree (**change USERNAME to your IU user name**).

```
% cd ~/cs161
% git init
% git remote add origin git@github.iu.edu:USERNAME/p536.git
% git add src
% git commit -a -m "Base OS161 Distribution"
% git push origin master
```

   You can change the argument to `-m` to be any "comment" that you want attached to this version of the source code.

4. End your script session (type `exit` or hit `CTRL-D`). Create a directory called `asst0` and move your script output to `asst0/gitinit.script`.

```
% cd ~/cs161
% mkdir asst0
% mv typescript ~/cs161/asst0/gitinit.script
% git add asst0
% git commit -a -m "Git init"
% git push origin master
```

Code Reading

One of the challenges of P536 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 every file in the system.

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 `~/cs161/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:

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

`configure`: this is an autoconf-like script. It sets up things like `How to run the compiler.' You needn't understand this file, although we'll ask you to specify certain pathnames and options when you build your own tree.

`defs.mk`: this file is generated when you run `./configure`. You needn't do anything to this file.

`defs.mk.sample`: this is a sample `defs.mk` file. Ideally, you won't be needing it either, but if configure fails, use the comments in this file to fix `defs.mk`.

You will also fine the following directories:

`bin`: this is where the source code lives for 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.

`kern`: here is where the kernel source code lives.

`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.

`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`: this directory contains pieces of makefile that are used for building the system. You don't need to worry about these, although in the long run we do recommend that anyone working on large software systems learn to use `make` effectively.

`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 every line in every excutable in bin and sbin, but it is worth the time to peruse a couple to see how they work. 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.

For our purposes, you need only concern yourself with the `mips` subdirectory.

`kern/arch/mips/conf`:

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

`Makefile.mips`: Kernel Makefile; this is copied when you "config a kernel".

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

Question 1. Which register number is used for the stack pointer (sp) in OS/161?

Question 2. What bus/busses does OS/161 support?

Question 3. What is the difference between splhigh and spl0?

Question 4. Why do we use typedefs like `u_int32_t` instead of simply saying `"int"`?

`kern/arch/mips/mips`: These are the source files containing the machine-dependent code that the kernel needs to run. Most of this code is quite low-level.

Question 5. What does splx return?

Question 6. What is the highest interrupt level?

`kern/asst1`: 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/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
% make depend
% make
```

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 pick a particularly low level final project, You can probably 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 7. How frequently are hardclock interrupts generated?

Question 8. What functions comprise the standard interface to a VFS device?

Question 9. How many characters are allowed in a volume name?

Question 10. How many direct blocks does an SFS file have?

Question 11. What is the standard interface to a file system (i.e., what functions must you implement to implement a new file system)?

Question 12. What function puts a thread to sleep?

Question 13. How large are OS/161 pids?

Question 14. What operations can you do on a vnode?

Question 15. What is the maximum path length in OS/161?

Question 16. What is the system call number for a reboot?

Question 17. Where is STDIN_FILENO defined?

`kern/lib`: These are library routines used throughout the kernel, e.g., managing sleep queues, run queues, kernel malloc, 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.

Question 18 Is it OK to initialize the thread system before the scheduler? Why (not)?

Question 19. What is a zombie?

Question 20. How large is the initial run queue?

`kern/userprog`: 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.

`kern/fs`: The file system implementation has two subdirectories. We'll talk about each in turn. `kern/fs/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 21. What does a device pathname in OS/161 look like?

Question 22. What does a raw device name in OS/161 look like?

Question 23. What lock protects the vnode reference count?

Question 24. What device types are currently supported?

Question 24: What device types are currently supported?

`kern/fs/sfs`**:** This is the simple file system that OS/161 contains by default. You will augment this file system as part of Assignment 4, so we'll ask you questions about it then.

Add the answers to the questions above to your repository and commit:

```
% cd ~/cs161
% git add asst0/code-reading.txt
% git commit -a -m "Code reading for assignment 0"
% git push origin master
```

Building a Kernel

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 using the `script` command.
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. Configure a kernel named ASST0.

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

4. Build the ASST0 kernel.

   ```
   % cd ../compile/ASST0
   % make depend
   % make
   ```

5. Install the ASST0 kernel.

   ```
   % make install
   ```

6. Now also build the user level utilties.

   ```
   % cd ~/cs161/src
   % make
   ```

7. End your script session. Rename your script output to `build.script`.

   ```
   % mv typescript ~/cs161/asst0/build.script
   % cd ~/cs161/asst0/
   % git add build.script
   % git commit -a -m "Building kernel"
   % git push origin master
   ```

Running Your Kernel

1. Download the file `sys161.conf` from the course web site and place it in your OS/161 root directory (`~/cs161/root`).
2. Script the following session.
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. Rename your script output to `run.script`.

```
% mv typescript ~/cs161/asst0/run.script
% cd ~/cs161/asst0/
% git add run.script
% git commit -a -m "Running kernel"
% git push origin master
```

Practice Modifying Your Kernel

1. Create a file called `main/hello.c` in the `cs161/src/kern` directory.
2. In this file, write a function called `hello` that uses `kprintf()` to print "Hello World\n".
3. Edit `main/main.c` and add a call (in a suitable place) to `hello()`.
4. Make your kernel build again. You will need to edit `conf/conf.kern`, reconfig, and rebuild.
5. Make sure that your new kernel runs and displays the new message.
6. Once your kernel builds, script a session demonstrating a config and build of your modified kernel. Call the output of this script session `newbuild.script`.

```
% mv typescript ~/cs161/asst0/newbuild.script
% cd ~/cs161/asst0/
% git add newbuild.script
% git commit -a -m "Modify kernel"
% git push origin master
```

Using GDB

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
% 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. Rename your script output to `gdb.script`.

```
% mv typescript ~/cs161/asst0/gdb.script
% cd ~/cs161/asst0/
```

```
% git add gdb.script
% git commit -a -m "Running gdb"
% git push origin master
```

Note: Emacs has a gdb mode that allows you to run gdb from within emacs and view the source code you are debuggin within an emacs editor window. To invoke this, use the command M-x gdb from with emacs.

Practice with Git

In order to build your kernel above, you already have the source tree in a Git repository. Now we'll demonstrate some of the most common features of Git. Create a script of the following session (the script should contain everything except the editing sessions; do those in a different window). Call this file `git-use.script`.

1. Edit the file `kern/main/main.c`. Add a comment with your name in it.
2. Execute

   ```
   % cd ~/cs161/src

   % git diff kern/main/main.c
   ```

   to display the differences in your version of this file.
3. Now commit your changes using `git commit -a -m "Added name comment"` (where "Added name comment" can be any comment describing the changes in this commit).
4. Remove the first 100 lines of `main.c`.
5. Try to build your kernel (this ought to fail).
6. Realize the error of your ways and get back a good copy of the file.

   ```
   % git checkout main.c
   ```

   The checkout command discards your changes and uses the most recently committed version of the file.
7. Try to build your tree again.
8. 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.
9. Now, show us where you inserted these DEBUG statements by doing a diff.

   ```
   % cd ~/cs161/src
   % git diff
   ```

10. `exit` your scripting session and add the script to your repository.

    ```
    % mv typescript ~/cs161/asst0/git-use.script
    % cd ~/cs161
    % git add asst0/git-use.script
    ```

11. Then commit your changes again to save the debug statements and script.

    ```
    % cd ~/cs161/src
    % git commit -a -m "Added debugging statements"
    ```

12. Finally, you should create a tagged version of your repository.

    ```
    % cd ~/cs161
    % git tag -a asst0-end -m "Assignment 0 done"
    % git push origin master --tags
    ```

    Note that you need to add `--tags` to the end of `git push origin master` in order to push tags up to IU GitHub.

What (and How) to Turn in

What (and how) to turn in...

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

- `setup.script`
- `gitinit.script`
- `code-reading.txt`
- `build.script`
- `run.script`
- `newbuild.script`
- `gdb.script`
- `git-use.script`

Submission is accomplished by creating the tagged repository as indicated above. We will check out your tagged repository to grade your assignment and will check in a file there with your marks.

Tips: Use the `git status` command in your `cs161` directory to see if you have untracked files (files not in your repository) or changes that need to be committed.

Use `git log FILE` to see the history of FILE. You can use this command to find out when a file was last changed and by whom.

If you have comments or suggestions, email *p536@cs.indiana.edu*

Author: Andrew Lumsdaine
E-Mail: p536@cs.indiana.edu

Created: Sat Aug 25, 2001
Modified: Mon 24-Sep-2012