**Advanced Operating Systems**

**Computer Science Department**
**Indiana University**
**Bloomington, IN 47405**

**The Tao**

Something mysterious is formed, born in the silent void. Waiting alone and unmoving, it is at once still and yet in constant motion. It is the source of all programs. I do not know its name, so I will call it the Tao of Programming.

If the Tao is great, then the operating system is great. If the operating system is great, then the compiler is great. If the compiler is greater, then the applications is great. The user is pleased and there is harmony in the world.

The Tao of Programming flows far away and returns on the wind of morning.

— Geoffrey James, *The Tao of Programming*

# Programming Assignment 1

| | |
|---|---|
| **Assigned** | Aug 27, 2012 |
| **Design Due** | Sep 4, 2012 |
| **Assignment Due** | Sep 12, 2012 |
| **Note** | This problem set is to be done in groups of two. |

## Introduction

In this assignment you will implement a miniature UNIX shell. In doing so, you will become familiar with systems programming, as well as designing larger projects. In implementing this shell you will learn about system calls, pipes, and signals. In addition, you will continue to explore OS/161 and will add two system calls to it. Although the shell that you are writing in this problem set will run under Linux, in a subsequent problem set you will also use it under OS/161.

**Write readable code!**

In your programming assignments, you are expected to write well-documented, readable code. There are a variety of reasons to strive for clear and readable code. Since you will be working in pairs, it will be important for you to be able to read your partner’s code. Finally, clear, well-commented code makes your AIs happy!

There is no single right way to organize and document your code. It is not our intent to dictate a particular coding style for this class. The best way to learn about writing readable code is to read other people's code. Read the OS/161 code, read your partner's code, read the source code of some freely available projects. When you read someone else's code, note what you like and what you don't like. Pay close attention to the lines of comments which most clearly and efficiently explain what is going on. When you write code yourself, keep these observations in mind.

Here are some general tips for writing better code:

- Group related items together, whether they are variable declarations, lines of code, or functions.
- Use descriptive names for variables and procedures. Be consistent with this

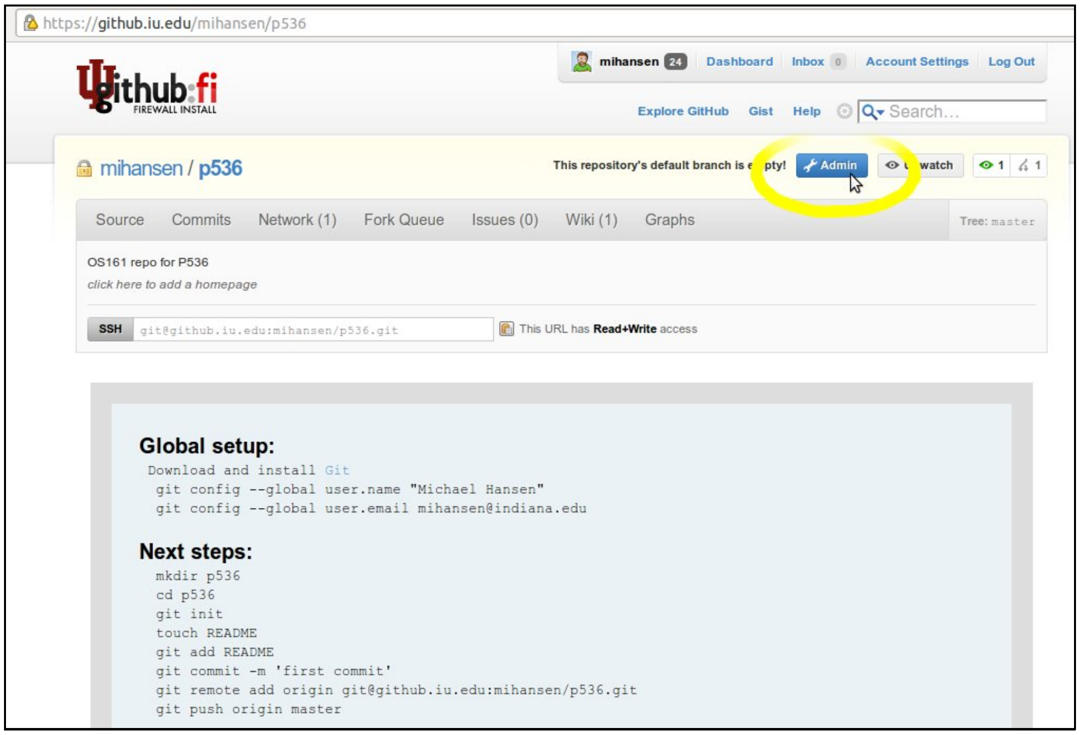
throughout the program.
- Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."

You and your partner will probably find it useful to agree on a coding style -- for instance, you might want to agree on how variables and functions will be named (my_function, myFunction, MyFunction, mYfUnCtIoN, ymayUnctionFay, etc.), since your code will have to interoperate.

## Begin Your Assignment

### Step 1 - Clone the Source Tree

Every student registered for P536 this semester should already have a Git repository. You and your partner should agree on which of your two repositories to use for your main source code repository. Once you have made this decision, the student with the main repository must add his or her partner as a collaborator. This done by visiting the project page for the repository on IU GitHub and clicking the `Admin` button:



OS161 repo for P536
click here to add a homepage

git@github.iu.edu:mihansen/p536.git

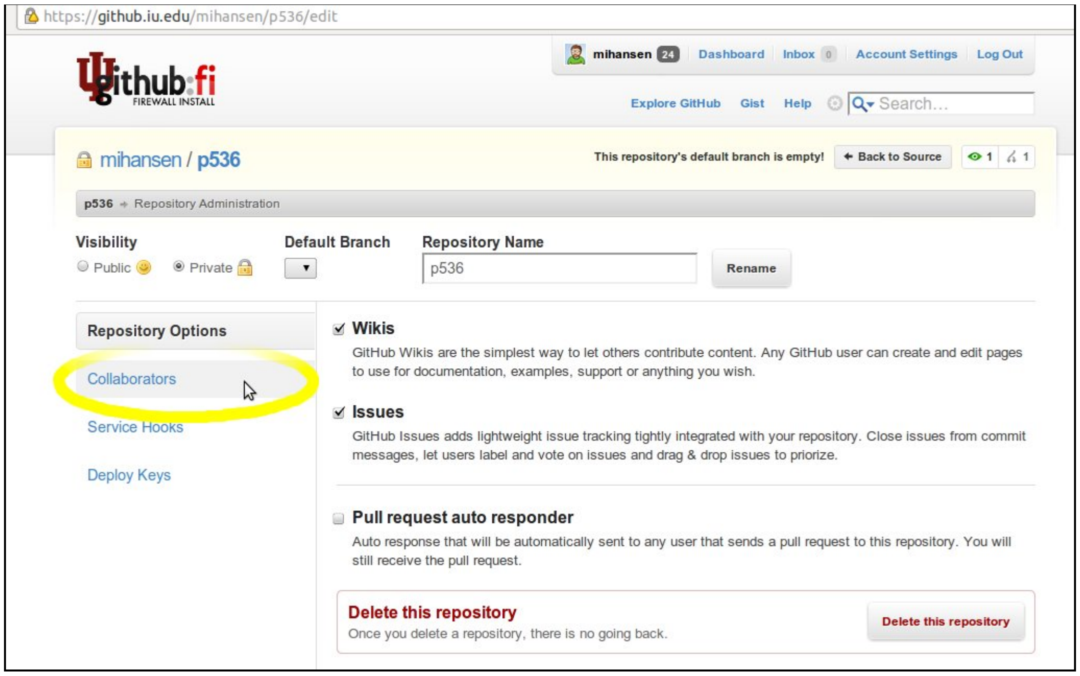
**Global setup:**

```
Download and install Git
  git config --global user.name "Michael Hansen"
  git config --global user.email mihansen@indiana.edu
```

**Next steps:**

```
  mkdir p536
  cd p536
  git init
  touch README
  git add README
  git commit -m 'first commit'
  git remote add origin git@github.iu.edu:mihansen/p536.git
  git push origin master
```

Next, click the the `Collaborators` link:


https://github.iu.edu/mihansen/p536/edit

mihansen 24 Dashboard Inbox 0 Account Settings Log Out

Explore GitHub Gist Help Search…

mihansen / p536

This repository's default branch is empty! Back to Source 1 1

p536 → Repository Administration

Visibility: Public, Private (selected)

Default Branch

Repository Name: p536 Rename

Repository Options

Collaborators

Service Hooks

Deploy Keys

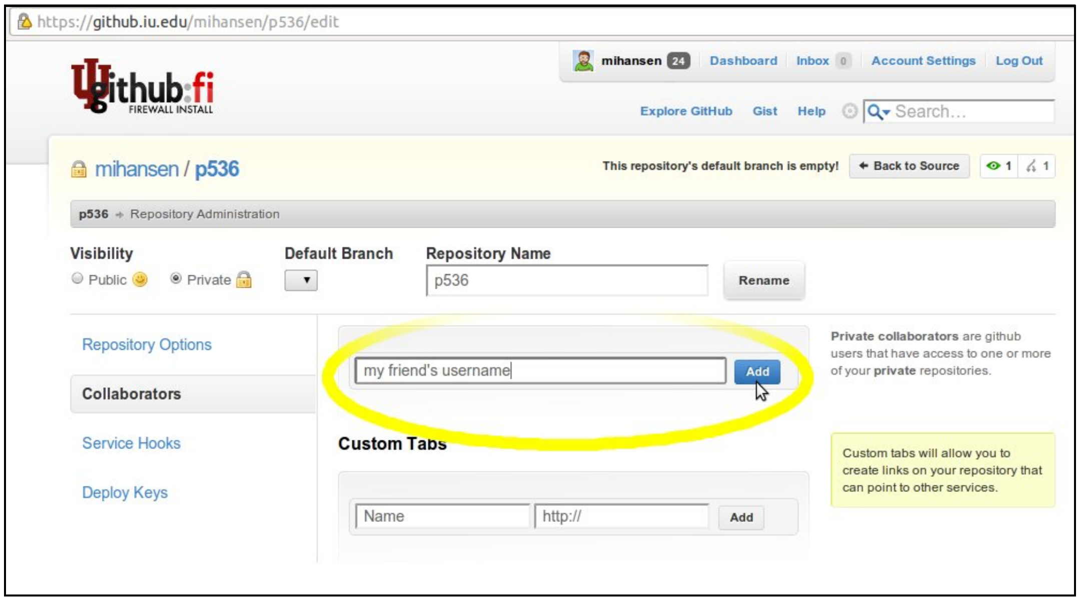
✓ **Wikis**
GitHub Wikis are the simplest way to let others contribute content. Any GitHub user can create and edit pages to use for documentation, examples, support or anything you wish.

✓ **Issues**
GitHub Issues adds lightweight issue tracking tightly integrated with your repository. Close issues from commit messages, let users label and vote on issues and drag & drop issues to priorize.

**Pull request auto responder**
Auto response that will be automatically sent to any user that sends a pull request to this repository. You will still receive the pull request.

**Delete this repository**
Once you delete a repository, there is no going back. Delete this repository

Finally, enter your partner's IU user name and click the `Add` button:


https://github.iu.edu/mihansen/p536/edit

mihansen 24 Dashboard Inbox 0 Account Settings Log Out

Explore GitHub Gist Help Search…

mihansen / p536

This repository's default branch is empty! Back to Source 1 1

p536 → Repository Administration

Visibility: Public, Private (selected)

Default Branch

Repository Name: p536 Rename

Repository Options

Collaborators

Service Hooks

Deploy Keys

my friend's username Add

**Private collaborators** are github users that have access to one or more of your **private** repositories.

Custom Tabs

Name http:// Add

Custom tabs will allow you to create links on your repository that can point to other services.

Your partner should now be able to clone your repository on their Linux CS machine using the following command (replace USERNAME with the IU user name of whoever has the main repository):

```
% git clone git@github.iu.edu:USERNAME/p536.git
```

This will create a directory called `p536` where you will be doing code edits and Git commits.

Before starting work, always grab the latest version of your source tree:

```
% cd ~/p536
% git pull origin master --tags
```

After making changes, you may commit locally at any time.

```
% cd ~/p536
% git commit -a -m "Comment describing my changes"
```

You may commit as many times as you like on your local machine. Your changes will **not** be visible to your partner until you push them up to IU GitHub.

```
% cd ~/p536
% git push origin master
```

Use the `git status` command to check if there are any pending changes to commit.

**Step 2 - Getting Started**

Before you start work on this assignment, tag your Git repository. The purpose of tagging your repository is to make sure that you have something against which to compare your final tree. Make sure that you do not have any outstanding updates in your tree.

Then, tag your repository exactly as shown below.

```
% git tag -a asst1-begin -m "Begin Assignment 1."
% git push origin master --tags
```

To get started with Assignment 1, create a directory called asst1. This directory should be at the same "level" as src (i.e., a subdirectory of p536).

```
% cd ~/p536
% mkdir asst1
```

**All work on the shell should occur in the asst1 directory.**

Acquire the scanner that is being provided by the instructional staff to ease the burden of taking input from users: scanner.tar.gz. Assuming the file is in your CS home directory, extract it as follows:

```
% mv scanner.tar.gz ~/p536/asst1/
% cd ~/p536/asst1
% tar -xzf scanner.tar.gz
% rm scanner.tar.gz
```

Your asst1 directory should now contain the following files:

- scanner.ll - A lex grammar file (you can ignore this if you are unfamiliar).
- scanner.c - A scanner to be compiled together with your shell.
- scanner.h - A header file for the scanner to be used with your shell.

It is expected that you create the following files:

- Makefile - The Makefile to build your shell (named "shell")
- shell.c - Your shell's code (you may split your code into multiple files)

## Shell Requirements

Your shell must support the following:

- Commands with an arbitrary number of flags (e.g., `ls -F -l -s -a`).
- Redirection in the form of "<", ">", and ">>" (i.e., read, write, append

from/to a file).
- Pipelines with an arbitrary number of sources and sinks (e.g., `ls | grep foo | grep bar | less`).
- Job control. Specifically, you must support:
    - Running jobs in the background with "&".
    - Stopping jobs with ctrl-Z.
    - Listing jobs with the built-in command `jobs`. The output of this this command should be, for each job, the job number (assigned by you), the command for that job, and the status of that job. For example, if there is one stopped and one job blocked on input, you might output:

      ```
      [1] ls | more  STOPPED
      [2] cat foo    BLOCKED
      ```

      The job number are assign as follows. The first job is assigned 1. When a job terminates, you can free its job number. Each successive job is assigned the lowest available job number.
    - Bringing a stopped or blocked job to the foreground with the built-in command `fg X`, where `X` is the job number as reported by the `jobs` command.
    - Restarting a stopped job in the background with the built-in command `bg X`, where `X` is the job number as reported by the `jobs` command.
    - Killing a stopped or blocked job with the built-in command `kill X`, where `X` is the job number as reported by the `jobs` command.
- Exiting the shell, with the command `bye`. This should not exit if any jobs are stopped; instead, it should print a message stating that there are stopped jobs.
- Error checking. This is limited to the following:
    1. stating that a command is invalid if it cannot be found,
    2. stating that a job number does not exist.

To assist you in parsing input, you will be provided with a scanner. The scanner reads in a token if yylex() is invoked. It takes no parameters and returns an integer code that tells you what the token is. Possible token return types are `AMPERSAND`, `LESS`, `GREATER`, `GRGR`, `PIPE`, `ENDLINE`, `INTEGER`, `STRING`. The header file `scanner.h` contains the definitions of these return types. Along with the return value, variable `yytext` will be set after each call to yylex(). Variable `yytext` is of type `char *`, and it holds the actual data scanned. For example, if the next token to scan is "ls", then the return value of `yylex()` is `STRING`, and `yytext` contains the string "ls". If the next token to scan is "4", then the return type of `yylex()` is `INTEGER`, and `yytext` contains the string "4". (You can use `atoi` to convert numeric strings to their integer values.)

**Notes:**

- In order to convert `STRING` tokens into numbers, be sure to use the `strtol` function so that you can catch invalid numbers.
- The shell must disable ctrl-Z for itself, but enable it for any of its children before they `exec` a command. This is because otherwise, a ctrl-Z will stop the shell itself, which we don't want.
- Make sure to handle pipes properly; in a string of pipe jobs, a "middle" job must read from the pipe of its left neighbor, and write to the pipe of its right neighbor.
- Make sure not to wait if there is an ampersand at the end of a command, or if you are going to fork a pipe job that is not the last one in the pipeline.
- To kill or restart stopped jobs, use the `kill` system call on each job in the

pipeline. That means each job will need to store its system-assigned process id.
- Note that `wait` will terminate if a child is stopped. You will need to use this property.
- You must clean up jobs that run in the background. The `wait` system call can also work for this by using the `WNOHANG` option. See the man pages.
- For pipeline jobs, you must make sure to use `dup2` and `close` appropriately to make sure that the pipe is written to instead of standard input/output.
- You may only use system calls for implementing your shell -- i.e., only functions found in manual section 2.

**Examples of valid commands:**

```
echo "foobar" > testfile
echo "foo" >> testfile
echo "bar" >> testfile

cat testfile
cat testfile > testfile2
cat < testfile > testfile2

cat testfile | grep "foo" > testfile2
cat testfile testfile2 | grep "foo"
cat testfile < testfile2 | grep "foo"
cat testfile | grep "foo" | grep "bar"
cat testfile | grep "foo" | grep "bar" > testfile2

ls -l -a -h | grep "testfile" >> testfile
wc -l < testfile > testfile2
```

## Shell Design and Implementation

Beginning with this assignment, please note that your *design documents* become an important part of the work you submit. The design documents should clearly reflect the development of your solution. They should not merely explain what you programmed. If you try to code first and design later, or even if you design hastily and rush into coding, you will most certainly end up in a software "tar pit". Don't do it! Work with your partner to plan everything you will do. Don't even think about coding until you can precisely explain to each other what problems you need to solve and how the pieces relate to each other.

Note that it can often be hard to write (or talk) about new software design -- you are facing problems that you have not seen before, and therefore even finding terminology to describe your ideas can be difficult. There is no magic solution to this problem; but it gets easier with practice. The important thing is to go ahead and try. Always try to describe your ideas and designs to someone else (we suggest your partner; roommates seem to have a low tolerance for this sort of thing). In order to reach an understanding, you may have to invent terminology and notation-this is fine (just be sure to explain it to your AI in your design document). If you do this, by the time you have completed your design, you will find that you have the ability to efficiently discuss problems that you have never seen before. Why do you think that CS is filled with so much jargon?

To help you get started with the shell project, we have provided the following questions to help you learn about some of the system calls you will need for building your shell. You will probably want to be familiar with the following system calls: `signal`, `kill`, `sigset`, `fork`, `waitpid`, `open`, `dup2`, `execvp`, `pipe` (these may not be the only ones you will need -- and you may not need all of these). The answers to some of the questions below can be found by reading the appropriate man pages.

**Questions**

1. What is the difference between section 2 and section 3 of the Unix manual ("man pages")?
2. How do you create new processes in Unix/Linux?
3. How can you cause a signal to a process to be ignored?
4. Is it possible to "chain" signal handlers together? I.e., suppose you wanted not to replace the default signal handler for a given signal but rather to invoke yours first and then invoke the default signal handler -- how would you do that?
5. Are there any signals that cannot be ignored?
6. How can a program tell if it is the child or the parent process after a call to `fork()`?
7. How do processes pass data through a pipe in Unix/Linux?
8. If a process has open files (and the associated file descriptors) and then calls `fork()`, will the child process also have access to those open files? Can the child process simply use the same file descriptors?

## OS/161

To help you get started with OS/161 (and, for this problem set, system calls in OS/161), we have provided the following questions as a guide for reading through the code. We recommend that you divide up the code and have each partner answer questions for different modules. After reading the code and answering questions, get together and exchange summations of the code you each reviewed. Once you have done this, you should be ready to discuss strategy for designing your code for this assignment.

## OS/161 Code Walk-Through

Include the answers to the code walk-through questions as the first part of your design document.

`kern/userprog`: This directory contains the files that are responsible for the loading and running of user-level programs. Currently, the only files in the directory are `loadelf.c`, `runprogram.c`, and `uio.c`. Understanding these files is the key to getting started with the implementation of multiprogramming. Note that to answer some of the questions, you will have to look in files outside this directory.

`loadelf.c`: This file contains the functions responsible for loading an ELF executable from the filesystem and into virtual memory space. (ELF is the name of the executable format produced by `cs161-gcc`.) Of course, at this point this virtual memory space does not provide what is normally meant by virtual memory -- although there is translation between the addresses that executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically. We recommend not stressing about this until a later assignment.

`runprogram.c`: This file contains only one function, `runprogram()`, which is responsible for running a program from the kernel menu.

`uio.c`: This file contains functions for moving data between kernel and user space. Knowing when and how to cross this boundary is critical to properly implementing userlevel programs, so this is a good file to read very carefully. You should also examine the code in `lib/copyinout.c`.

**Questions**

1. What are the ELF magic numbers?
2. What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?
3. Why can the `struct uio` that is used to read in a segment be allocated on the stack in `load_segment()` (i.e., where does the memory read actually go)?
4. In `runprogram()`, why is it important to call `vfs_close()` before going to usermode?
5. What function forces the processor to switch into usermode? Is this function machine dependent?
6. In what file are `copyin` and `copyout` defined? `memmove`? Why can't `copyin` and `copyout` be implemented as simply as `memmove`?
7. What (briefly) is the purpose of `userptr_t`?

**kern/arch/mips/mips: traps and syscalls**

Exceptions are the key to operating systems; they are the mechanism that enables the OS to regain control of execution and therefore do its job. You can think of exceptions as the interface between the processor and the operating system. When the OS boots, it installs an "exception handler" (carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this, which sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term in computer science, operating system lingo for an exception is a "trap" -- when the OS traps execution. Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, `syscall.c` handles traps that happen to be syscalls. Understanding at least the C code in this directory is key to being a real operating systems junkie, so we highly recommend reading through it carefully.

`trap.c`: `mips_trap()` is the key function for returning control to the operating system. This is the C function that gets called by the assembly exception handler. `md_usermode()` is the key function for returning control to user programs. `kill_curthread()` is the function for handling broken user programs; when the processor is in usermode and hits something it can't handle (say, a bad instruction), it raises an exception. There's no way to recover from this, so the OS needs to kill off the process.

`syscall.c`: `mips_syscall()` is the function that delegates the actual work of a system call to the kernel function that implements it. Notice that `reboot()` is the only case currently handled.

**Questions**

1. What is the numerical value of the exception code for a MIPS system call?
2. Why does `mips_trap()` set `curspl` to `SPL_HIGH` "manually", instead of

using `splhigh()`?
3. How many bytes is an instruction in MIPS? (Answer this by reading `mips_syscall()` carefully, not by looking somewhere else.)
4. What would be required to implement a system call that took more than 4 arguments?

`lib/libc`: This is the user-level C library. There's obviously a lot of code here. We don't expect you to read it all, although it may be instructive in the long run to do so. Job interviewers have an uncanny habit of asking people to implement standard C library functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls, which we detail below.

`errno.c`: This is where the global variable errno is defined.

`syscalls-mips.S`: This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls.

`syscalls.S`: This file is created from syscalls-mips.S at compile time and is the actual file assembled into the C library. The actual names of the system calls are placed in this file using a script called `callno-parse.sh` that reads them from the kernel's header files. This avoids having to make a second list of the system calls. In a real system, typically each system call stub is placed in its own source file, to allow selectively linking them in. OS/161 puts them all together to simplify the makefiles.

**Questions**

1. What is the purpose of the `SYSCALL` macro?
2. What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

## Implementation

### System calls and exceptions

Implement system calls and exception handling. The full range of system calls that we think you might want over the course of the semester is listed in `kern/include/kern/callno.h`. For this assignment you should implement only:

- `_time()`
- `_exit()`

The file `include/unistd.h` contains the user-level interface definition of the system calls that you will be writing for OS/161 (including ones you will implement in later assignments). This interface is different from that of the kernel functions that you will define to implement these calls. You need to think about a variety of issues associated with implementing system calls. Perhaps the most obvious one is: can two different user-level processes (or user-level threads, if you choose to implement them) find themselves running a system call at the same time? Be sure to argue for or against this, and explain your final decision in the design document.

A note on errors and error handling of system calls:

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from `kern/errno.h`. If none seem particularly appropriate, consider adding a new one. If you're adding an error code for a condition for which Unix has a standard error code symbol, use the same symbol if possible. If not, feel free to make up your own, but note that error codes should always begin with E, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes; on Linux systems "man errno" will do the trick.

Note that if you add an error code to `src/kern/include/kern/errno.h`, you need to add a corresponding error message to the file `src/lib/libc/strerror.c`.

Note that there is already a system call defined for `_time()` and then a library routine `time()`. Once you've written the system call, the library routine should function correctly. The same is true for `exit()`. For implementing `_time()`, the kernel provides the `gettime()` function internally.

For purposes of allowing our user-level processes/threads to terminate, we must provide a basic implementation of the `_exit`. In future assignemnts, you will need to extend this syscall to work with processes instead of threads, but for now, we will not get into that. Here's an implementation of the system call you may use in your kernel for now:

```
#include <thread.h>

int sys__exit(int code) {
        kprintf("\nExit code = %d\n", code);
        thread_exit();

        panic("I shouldn't be here!");
        return 0;
}
```

You will need to provide the appropriate code in `mips_syscall()` to call this function. Once you have done so, you should now be able to run *very* primitive user-level programs in your kernel.

## Testing

To test out system calls, we need to create a test program to be run as a user-level program. For simplicity, we are going to hijack the user-level shell (since it's in the menu and contains no code). You will find the shell along with all of the other user-level programs in OS/161 in the `src/bin` directory. We would like to change the `sh`'s `main()` to call the UNIX `time()` function and return the value as the program's exit code (don't forgot to include the appropriate header file):

```
time_t now;
time(&now);
return now;
```

At this point, our kernel only supports a limited set of operations from user-level programs. Calling `time()` and returning from the shell (implicitly calling `_exit()`) tests all of them.

To build the user-level programs and install them into our system, we execute the following commands:

```
% cd ~/p536/src
% make
```

Note that the Makefile automatically copies the binaries into `~/p536/root`.

Finally, be sure to rebuild your kernel:

```
% cd ~/p536/src/kern/compile/ASST0
% make
% make install
```

At this point, you should be able to test whether your "shell" works:

```
% cd ~/p536/root
% sys161 kernel
```

While inside your kernel, you should be able to select the menu item "s" &endash; intended to run a shell, to be added at a later date. If everything works, then you will see something like the following:

```
OS/161 kernel [? for menu]: s
Operation took 0.000149280 seconds
OS/161 kernel [? for menu]:
Exit code = 1221189556
```

If you were to try to run the shell prior to implementing `_exit()` you would see:

```
OS/161 kernel [? for menu]: Unknown syscall 0
...
```

If you were to try to run the shell prior to implementing `_time()` you would see:

```
OS/161 kernel [? for menu]: Unknown syscall 6
...
```

### Questions

1. Why are we unable to use `printf()` to report the time from our test program?
2. Why do we use `kprintf()` instead?

## What to Submit

### Design phase

For the design phase, we would like you to document the design of shell. Describe how you will go about implementing pipelines, redirection, and job control. Make sure your design document is in the asst1 directory. Additionally, we expect at this point for your shell to be able to handle simple commands (e.g., "cat testfile").

It is important that your shell can be built and run by executing:

```
% cd ~/p536/asst1
% make
% ./shell
```

Add the contents of asst1 to your Git repository and do a commit.

```
% cd ~/p536
% git add asst1
% git commit -a -m "Added assignent 1 directory"
```

To submit your assignment after the design phase, please tag your repository as asst1-design:

```
% git tag -a asst1-design -m "Assignment 1 design."
% git push origin master --tags
```

**All done!**

Finally, after finalizing all your commits tag your repository.

```
% git tag -a asst1-end -m "Assignment 1 end."
% git push origin master --tags
```

We will be checking out and grading this version of your repository.

You do not need to print out anything for this assignment.

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

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