Assignment 2 : System Calls and Processes

Design Due: Design document due 11:59PM on Wednesday, 25 Feb 2009
Assignment Due: Final submission due 11:59pm on Wednesday, 11 Mar 2009

Objectives

- Be able to write code that meets a specified interface definition.
- Understand how to represent processes in an operating system.
- Design and implement data structures to manage processes and scheduling in an operating system.
- Understand how to implement system calls.
- Understand the theory and practice of process scheduling.

Introduction

This is the first assignment that requires you to work in teams of two. This may be difficult if you are accustomed to working alone, but it is essential for the completion of the remaining assignments and is a worthwhile skill to develop in any case. These assignments are too complex to be done single-handedly, and you will gain valuable real-world experience from learning to work in a team effectively.

In order to begin working together, you and your partner need to decide on a code base and then need to set up a shared SVN repository. Choose your code base with care. The assignments are cumulative and you will have to live with the consequences of this decision for the remainder of the semester. We suggest that you and your partner resolve conflicts about things like programming style and naming conventions now in order to avoid confusion later. Working together on a program can be much more demanding and frustrating than doing lab work together. (Imagine writing a coherent term paper with someone else!)

The code base you select should be a working OS/161 system with clean, well-commented, bullet-proof synchronization primitives. You and your partner should share your solutions to the previous assignments (it's good practice to learn to read and understand someone else's code) and decide what your code base will be. You are free to choose either partner's code, to merge your solutions, or to use the solution set.

Once you have selected a code base, refer to Section 3 for directions on setting up a shared SVN repository.

Working in teams

There are a number of issues that you and your partner should work out now, when things are calm, so you needn't figure them out at 2:00 AM in the heat of the moment.

Naming Conventions

It's a good idea to select some rudimentary protocol for naming global variables and variables passed as arguments to functions. This way, you can just ask your partner to write some function and, while s/he's doing it, you can make calls to that function in your own code, confident that you have a common naming convention from which to work. Be consistent in the way you write the names of functions: given a function called "my function", one might write its name as my_function, myFunction,

MyFunction, mYfUnCtIoN, ymayUnctionFay, etc. Pick one model and stick to it (although we discourage the last two examples).

SVN Use

Since you and your partner will be using SVN to manage your work, you will need to decide when and how often to commit changes. (Advice: early and often.) Additionally, you should agree upon how much detail to log when committing files. Perhaps more importantly, you also need to think about how to maintain the integrity of the system- what procedures to follow to make sure you can always extract a working version of some sort from SVN, whether or not it's the latest version, what tests to run on a new version to make sure you haven't inadvertently broken something, etc.

Clear, explicit SVN logs are essential. If you are incommunicado for some reason, it is vital for your partner to be able to reconstruct your design, implementation and debugging process from your SVN logs. In general, leaving something uncommitted for a long period of time is dangerous: it means that you are undertaking too large a piece of work and have not broken it down effectively. Once some new code compiles and doesn't break the world, commit it. When you've got a small part working, commit it. When you've designed something and written a lot of comments, commit it. Commits are free. Hours spent hand-merging hundreds of lines of code wastes time you'll never get back. The combination of frequent commits and good, detailed comments will facilitate more efficient program development.

Use the features of SVN to help you. For example, you may want to use tags to identify when major features are added and when they're stable. The brave of heart might want to investigate SVN branches (using 'svn copy'), which provide completely separate threads of development (one significant caveat- although branches make life much easier while you're developing within a branch, merging branches together later is often a major headache).

Communication

Nothing replaces good, open communication between partners. The more you can direct that communication to issues of content ("How shall we design sys_execv()?") instead of procedural details ("What do you mean, you never checked in your version of foo.c?"), the more productive your group will be.

In your design documents for each assignment, you should identify, at least in general terms, who was responsible for the various parts of your solutions.

If at any time during the course of the semester, you and your partner realize that you are having difficulty working together, please come speak with Professor Seltzer or one of the teaching fellows. We will work with you to help your partnership work more effectively, or in extreme circumstances, we will help you find new partners. Do not suffer in silence; please come talk with us.

Setting up Your SVN Repositories

Before working on this assignment, you and your partner must set up a shared SVN repository where changes will be committed for the rest of the semester. We will aid you in this effort. Once you've emailed the course staff, or posted to the Bulletin Board, about your pairing (and team name), we will give you SVN access to each other's repositories. Decide which partner's synchronization code you will use and perform the following:

If you and your partner have decided to use your code:

```
sleep(); // Remember to tag your repository for the beginning of ASST2
```

If you and your partner have decided to use your partner's code:

```
% cd ~/cs161/
% svn ci -m "Asst1 Repo, transfering to partner's repo."
% rm -rf os161
$ svn co http://minitrue/repos/<partner:username>/truck src
```

Or you and your partner may decide to continue from the solution set. Please email the course staff to obtain a .tgz. Then you can recall how to use 'svn import' from Asst0 in order to set up your repositories.

Note: Remember which partner's repository you are creating your working copies from. Keep this straight, merging will be more difficult later if you accidentally checked out from your repository while your partner was working from his/her repository.

Assignment Organization

Your current OS/161 system has minimal support for running executables -- nothing that could be considered a true process. Assignment 2 starts the transformation of OS/161 into a true multi-tasking operating system. After the next assignment, it will be capable of running multiple processes at once from actual compiled programs stored in your account. These programs will be loaded into OS/161 and executed in user mode by System/161; this will occur under the control of your kernel and the command shell in `bin/sh`.

First, however, you must implement the interface between user-mode programs ("userland") and the kernel. As usual, we provide part of the code you will need. Your job is to design and build the missing pieces.

You will also be implementing the subsystem that keeps track of the multiple tasks you will have in the future. You must decide what data structures you will need to hold the data pertinent to a "process" (hint: look at kernel include files of your favorite operating system for suggestions, specifically the `proc` structure.)

The first step is to read and understand the parts of the system that we have written for you. Our code can run one user-level C program at a time as long as it doesn't want to do anything but shut the system down. We have provided sample user programs that do this (reboot, halt, poweroff), as well as others that make use of features you will be adding in this and future assignments.

So far, all the code you have written for OS/161 has only been run within, and only been used by, the operating system kernel. In a real operating system, the kernel's main function is to provide support for user-level programs. Most such support is accessed via "system calls." We give you one system call, `reboot()`, which is implemented in the function `sys_reboot()` in main.c. In GDB, if you put a breakpoint on `sys_reboot` and run the "reboot" program, you can use "backtrace" to see how it got there.

User level programs

Our System/161 simulator can run normal programs compiled from C. The programs are compiled with a cross-compiler, `cs161-gcc`. This compiler runs on the host machine and produces MIPS executables; it is the same compiler used to compile the OS/161 kernel. To create new user programs, you will need to edit the Makefile in `bin`, `sbin`, or `testbin` (depending on where you put your programs) and then create a directory similar to those that already exist. Use an existing program and its Makefile as a template.

Design

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 TF 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, 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.

Code walk-through (10 points)

Before beginning your code reading, please apply the latest patch to OS/161. You can get the patch here.

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

The key files that are responsible for the loading and running of user-level programs are `loadelf.c`, `runprogram.c`, and `uio.c`, although you may want to add more of your own during this assignment. 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 other files.

`kern/syscall/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 Assignment 3.

`kern/syscall/runprogram.c`: This file contains only one function, `runprogram()`, which is responsible for running a program from the kernel menu. It is a good base for writing the `execv()` system call, but only a base -- when writing your design doc, you should determine what more is required for `execv()` that `runprogram()` does not concern itself with. Additionally, once you have designed your process system, `runprogram()` should be altered to start processes properly within this framework; for example, a program started by `runprogram()` should have the standard file descriptors available while it's running.

`kern/lib/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 `kern/vm/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: 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/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.

`locore/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. `enter_new_process()` 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. Part of this assignment will be to write a useful version of this function.

`syscall/syscall.c`: `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. You will also find a function, `enter_forked_process()`, which is a stub where you will place your code to implement the `fork()` system call. It should get called from `sys_fork()`.

Questions

1. What is the numerical value of the exception code for a MIPS system call?
2. How many bytes is an instruction in MIPS? (Answer this by reading `syscall()` carefully, not by looking somewhere else.)
3. Why do you "probably want to change" the implementation of `kill_curthread()`?
4. What would be required to implement a system call that took more than 4 arguments?

`user/lib/crt0/mips`: This is the user program startup code. There's only one file in here, `mips-crt0.S`, which contains the MIPS assembly code that receives control first when a user-level program is started. It calls the user program's `main()`. This is the code that your `execv()` implementation will be interfacing to, so be sure to check what values it expects to appear in what registers and so forth.

`user/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.

`user/lib/libc/unix/errno.c`: This is where the global variable errno is defined.

`user/lib/libc/arch/mips/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 `syscalls/gensyscalls.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.)
3. After reading syscalls-mips.S and syscall.c, you should be prepared to answer the following question: Now that OS/161 supports 64-bit values, `lseek()` takes and returns a 64-bit offset value. Thus, lseek() takes a 32-bit file handle (arg0), a 64-bit offset (arg1), a 32-bit whence (arg3), and needs to return a 64-bit offset value. In `void syscall(struct trapframe *tf)` where will you find each of the three arguments (in which registers) and how will you return the 64-bit offset?

Design and Implementation
(90 points total; 30 points design, 30 points for functionality, 30 points for clarity and attention to detail)

Before you begin this assignment, tag your repository as `asst2-begin`.

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/syscall.h`. For this assignment you should implement:

- open, read, write, lseek, close, dup2, chdir, getcwd
- getpid
- fork, execv, waitpid, _exit

It's crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161.) You should consult the OS/161 man pages included in the distribution and understand fully the system calls that you must implement. You must return the error codes as decribed in the man pages.

Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages. Some of the grading scripts rely on the return of appropriate error codes; adherence to the guidelines is as important as the correctness of the implementation.

The file `user/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 design this interface and put it in `kern/include/syscall.h`. As you discovered (ideally) in Assignment 0, the integer codes for the calls are defined in `kern/include/kern/syscall.h`.

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.

open(), read(), write(), lseek(), close(), dup2(), chdir(), and __getcwd()

For any given process, the first file descriptors (0, 1, and 2) are considered to be standard input (stdin), standard output (stdout), and standard error (stderr). These file descriptors should start out attached to the console device ("con:"), but your implementation must allow programs to use `dup2()` to change them to point elsewhere.

Although these system calls may seem to be tied to the filesystem, which you you will build in Assignment 4, in fact, these system calls are really about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the current working directory) is specific only to the process, but others (such as file offset) is specific to the process and file descriptor. Don't rush this design. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

Note that there is a system call `__getcwd()` and then a library routine `getcwd()`. Once you've written the system call, the library routine should function correctly.

getpid()

A pid, or process ID, is a unique number that identifies a process. The implementation of `getpid()` is not terribly challenging, but pid allocation and reclamation are the important concepts that you must implement. It is not OK for your system to crash because over the lifetime of its execution you've used up all the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement `getpid()`.

fork(), execv(), waitpid(), _exit()

These system calls are probably the most difficult part of the assignment, but also the most rewarding. They enable multiprogramming and make OS/161 a much more useful entity.

`fork()` is the mechanism for creating new processes. It should make a copy of the invoking process and make sure that the parent and child processes each observe the correct return value (that is, 0 for the child and the newly created pid for the parent). You will want to think carefully through the design of `fork()` and consider it together with `execv()` to make sure that each system call is performing the correct functionality.

`execv()`, although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and make theme execute something useful (i.e., something different than what the parent is executing). Essentially, it must replace the existing address space with a brand new one for the new executable (created by calling `as_create` in the current `dumbvm` system) and then run it. While this is similar to starting a process straight out of the kernel (as `runprogram()` does), it's not quite that simple. Remember that this call is coming out of userspace, into the kernel, and then returning back to userspace. You must manage the memory that travels across these boundaries very carefully. (Also, notice that `runprogram()` doesn't take an argument vector -- but this must of course be handled correctly in `execv()`).

Although it may seem simple at first, `waitpid()` requires a fair bit of design. Read the specification carefully to understand the semantics, and consider these semantics from the ground up in your design. You may also wish to consult the UNIX man page, though keep in mind that you are not required to implement all the things UNIX `waitpid()` supports -- nor is the UNIX parent/child model of waiting the

only valid or viable possibility.

The implementation of `_exit()` is intimately connected to the implementation of waitpid(). They are essentially two halves of the same mechanism. Most of the time, the code for `_exit()` will be simple and the code for `waitpid()` relatively complicated -- but it's perfectly viable to design it the other way around as well. If you find both are becoming extremely complicated, it may be a sign that you should rethink your design.

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/include/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/kern/include/kern/errmsg.h`.

kill_curthread()

Feel free to write `kill_curthread()` in as simple a manner as possible. Just keep in mind that essentially nothing about the current thread's userspace state can be trusted if it has suffered a fatal exception -- it must be taken off the processor in as judicious a manner as possible, but without returning execution to the user level.

Testing using the shell

In `user/bin/sh` you will find a simple shell that will allow you to test your new system call interfaces. When executed, the shell prints a prompt, and allows you to type simple commands to run other programs. Each command and its argument list (an array of character pointers) is passed to the `execv()` system call, after calling `fork()` to get a new thread for tis execution. The shell also allows you to run a job in the background using `&`. You can exit the shell by typing "exit".

Under OS/161, once you have the system calls for this assignment, you should be able to use the shell to execute the following user programs from the bin directory: cat, cp, false, pwd, and true. You may also find some of the programs in the testbin directory helpful.

Scheduling

Right now, the OS/161 scheduler implements a simple round-robin queue. You will see in thread.c that the schedule() function is actually blank. The round-robin scheduling comes into effect when hardclock() calls thread_yield() and a subsequent call to thread_switch() pops the current thread on the cpu's run-que and takes the next thread off the current cpu's run-queue (FIFO order). As we will learn in class, this is probably not the best method for achieving optimal processor throughput. For this assignment, you should implement two more scheduling algorithms.

You should select two fairly different scheduling algorithms and in your design document, explain why you selected the ones you did, and under what conditions you expect each to perform better. We suggest selecting one extraordinarily simple scheduling algorithm (e.g., pick a random process to run) and one more complex algorithm. For each, think carefully about what information you will need to maintain in order to completely implement the algorithm.

For this portion of the assignment it is OK to ignore the multi-core nature of OS/161. Your schedulers can operate on the current cpu's run-queue (e.g. reshuffle the threads on the current cpu's run-queue based on some priority metric). The threads on the other core will be reshuffled when your scheduler runs on the other core.

You may want your schedulers to be configurable. For example, for a round robin scheduler, it should be possible to specify the time slice. For a multi-level feedback queuing system, you might want to specify the number of queues and/or the time slice for the highest priority queue.

It is OK to have to recompile to change these settings, as with the HZ parameter of the default scheduler. And it is OK to require a recompile to switch schedulers. But it shouldn't require editing more than a couple `#defines` or the kernel config file to make these changes.

In any event, OS/161 should display at bootup which scheduler is in use.

Test your scheduler by running several of the test programs from `testbin` (e.g., add.c, hog.c, farm.c, sink.c, kitchen.c, ps.c) using the default time slice and scheduling algorithm. Experiment with the different scheduling algorithms that you implemented. Write down what you expect to happen with each algorithm. Then compare what actually happened to what you predicted and explain the difference.

Thread Migration

In the current iteration of OS/161 thread_consider_migration() will periodically be called by each cpu. This function (dumbly) looks at the total number of threads, the number of threads on the current cpu, and moves a certain number of threads off the end of the runqueue to other cpus.

The final challenge of this assignment asks you to improve upon this thread migration system. Consider different load balancing metrics, choose one, and argue in your design documnet why your migration scheme - based on this metric - will improve your system's performance.

The teaching staff will release a slightly more intelligent, yet still dumb, migration system for you to consider as well. Feel free to build off this system or design your own.

Design Considerations

Here are some additional questions and thoughts to aid in writing your design document. They are not, by any means, meant to be a comprehensive list of all the issues you will want to consider. You do not need to explicit answer or discuss these questions in your design document, but you should at least think about them.

Your system must allow user programs to receive arguments from the command line. For example, you should be able to run the following program:

```
char  *filename = "/bin/cp";
char  *args[4];
pid_t  pid;

args[0] = "cp";
args[1] = "file1";
args[2] = "file2";
args[3] = NULL;

pid = fork();
if (pid == 0) execv(filename, argv);
```

which will load the executable file `cp`, install it as a new process, and execute it. The new process will then find `file1` on the disk and copy it to `file2`.

Some questions to think about:

Passing arguments from one user program, through the kernel, into another user program, is a bit of a chore. What form does this take in C? This is rather tricky, and there are many ways to be led astray. You will probably find that very detailed pictures and several walk-throughs will be most helpful.

What primitive operations exist to support the transfer of data to and from kernel space? Do you want to implement more on top of these?

How will you determine: (a) the stack pointer initial value; (b) the initial register contents; (c) the return value; (d) whether you can exec the program at all?

You will need to "bullet-proof" the OS/161 kernel from user program errors. There should be nothing a user program can do to crash the operating system (with the exception of explicitly asking the system to halt).

What new data structures will you need to manage multiple processes?

What relationships do these new structures have with the rest of the system?

How will you manage file accesses? When the shell invokes the `cat` command, and the `cat` command starts to read `file1`, what will happen if another program also tries to read file1? What would you like to happen?

Design Submission

Turn in your design document by email to your TF by 11:59pm on 2/25/2009.

Note: Your design document is worth 30% of the grade for this assignment. It should contain:

- Answers to the code walk-through questions.
- A high level description of how you are approaching the problem.
- A detailed description of the implementation (e.g., new structures, why they were created, what they are encapsulating, what problems they solve).
- A discussion of the pros and cons of your approach.
- Alternatives you considered and why you discarded them.

Final Online Submission

When you are finished with Assignment 2, create a directory called `asst2`. In this directory, you should place the following:

- `teamname-asst2.tar.gz` A copy of the complete current source code of your OS/161 version, as a .tar.gz file.
- `asst2.diff`: A diff file listing all the changes you made for asst2.
- Your design document.
- A script of the OS/161 shell running various basic commands (cat, rm, cp, and pwd) under OS/161.
- A script of OS/161 running the various tt* tests successfully.
- A script of the new test or tests you added for testing your wait/exit implementation.

- A script of OS/161 running different test programs under the different scheduling policies.
- Your `sys161.conf` file.

Then add and commit the `asst2` directory to `minitrue`.

How to proceed

Before the design doc is due:

1. Meet with your partner. Review the files described above. Separately answer the questions provided. Compare your solutions.
2. Discuss high level implementations of your solutions. Do detailed design in parallel with your partner.
3. Decide which functions you need to change and which structures you may need to create to implement the system calls. Discuss how you will pass arguments from user space to kernel space (and vice versa). Discuss how you will keep track of open files. For which system call is this useful? Discuss how you will keep track of running processes. For which system call is this useful?
4. Discuss how you will implement the `execv()` system call. How is the argument passing in this function different from that of other system calls?

Before the assignment is due:

1. Carefully divide up the work. `execv()` might be the single most demanding part of the assignment, but `waitpid()` is non-trivial as well. We suggest that one of you should do the basic system calls and the other focus on process support.
2. For the parts you're assigned, verify that the collaborated design will really work. If something needs to be redesigned, do it now, and run it by your partner.
3. Implement.
4. Test, test, test. Test your partner's code especially.
5. Fix. Perhaps you won't need this step. (We all need to dream, right?)

On the assignment due date:

1. Commit all your final changes to the system. Make sure your partner has committed everything as well. Make sure you have the most up-to-date version of everything. Re-run make on both the kernel and userland to make sure all the last-minute changes get incorporated.
2. Run the tests and generate scripts. If anything breaks, curse (politely of course) and repeat step 1.
3. Tag your SVN repository; prepare the diff and submission source tree.
4. Submit everything. It doesn't matter which partner submits, but please make sure only one does.