1 Administrivia

- **Wednesday, February 18, 2009**: Assignment 1 due at **11:59 p.m.**.
- **Wednesday, February 18, 2009**: Assignment 2 Released. Make sure you have a partner! Please use the class Bulletin Board if you are still in search of a partner. And once you’ve found a partner, do inform the cs161 staff by posting to the Bulletin Board. If you are at a total lost for a partner, please email cs161@fas.
- **Wednesday, February 25, 2009**: Email your assigned TF your design document by 11:59pm. Please send these as a plain text file (ASCII art is fine and can be helpful sometimes) or a pdf file. There is no guarantee that your TF will be able to comment on your design if it is in an unsupported format. As the saying goes, please no tex files with random packages, word documents, gnuplot scripts, etc.
- **Wednesday, March 11, 2009**: Assignment 2 due at **11:59 p.m.**: Do not leave Asst 2 to the last minute. It is more work than Asst 1. Also, if you are in CS124, please be aware that you will have a CS124 assignment due around the same time.

2 Quick Synchronization Review

2.1 Fast Food Drive-Thru

Finally, we’re going to teach you some skills that you will be able to put to use in your life after Harvard!

You’re in charge of running a fast food restaurant and you have a limited number of cooks making food. Customers can either make orders from the
counter or at the drive-thru. You would like to prioritize the drive-thru customers over the walkins because people waiting in cars are more impatient.

Also, there is some chance that a customer (either walkin or drive-thru) may have an error in their order, in which case they will loudly complain which prevent the placement of any new walkin (but not drive-thru) orders until they leave. Complainers will politely wait for other complainers to finish before they start complaining. Hard life, huh?

How can you solve this problem using the synchronization primitives we’ve learned in class?

2.1.1 First Try

Walkin Customer:

```c
acquire(drive_thru_mutex);
/* Check for cars at the drive-thru */
while (drive_thru_cars) {
    release(drivethru_mutex);
    sleep(1);
    acquire(drivethru_mutex);
}
release(drive_thru_mutex);

/* Try to order */
acquire(complaint_lock);
P(cooks_sem);
release(complaint_lock);

/* Order food */
if (not_what_i_ordered(food)) {
    acquire(complaint_lock);
    /* Complain */
    release(complaint_lock);
}
V(cooks_sem);
```

Drive-Thru Customer:

```c
/* Show that we have arrived at the drive-thru */
drive_thru_cars++;

/*
Since we’re in the car, we aren’t bothered by complaints

P(cooks_sem);

/* Order food */

if (not_what_i_ordered(food)) {
    acquire(complaint_lock);
    /* Complain */
    release(complaint_lock);
}

V(cooks_sem);

/* Show that we are leaving the drive-thru */
drive_thru_cars--;

For this problem, ironically, ignore the possibility of starvation. However, there are still at least three problems with the above code. What are they? How can we improve this code?

3 Solving Synchronization Problems

We want to avoid race conditions, deadlocks and starvation. How can we do this?

3.1 Avoiding race conditions and starvation

- Model your problem:
  - What does your bathroom look like?
  - What are its states?
- Write down rules for each component (thread)
  - How does your bathroom change its states?
  - How does a male/female know that he/she should enter the bathroom?
  - What should a male/female check for before entering the bathroom?

3.2 Avoiding deadlocks

These methods were discussed in class in relation to the Dining Philosopher’s problem.
All or nothing
- Either we get all the resources we need, or none of them
- Problem: This can become very inefficient

Break Conflicts
- Somehow revoke rights to a resource to resolve conflicts
- Problem: Difficult to handle failure states

Locking Order
- Acquire resources in a pre-specified order
- Problem: Sometimes it isn’t obvious how to do this when you have different access patterns
- However, this is still generally your best choice

4 General Synchronization Advice
Synchronization is hard. Here are some tips to make it a little easier.

- When to synchronize?
  - Modifying a global variable in different threads
  - Protecting state during forced sleep (i.e. I/O)
  - Obviously, this list is not exhaustive...

- Pick the right primitives
  - Locks - Used to make critical sections, especially modifying shared state
  - Semaphores - Used when two threads interact with each other cooperatively
  - CVs - Used when a thread needs to wait for a condition to be true before proceeding
  - These are just common examples – be ready when things don’t match any one category, or you need to mix primitives.

- Organize and limit conflicts
  - Try to modularize your code to minimize critical sections which require synchronization
  - Keep related synchronization close together, not spread across different files
  - Build things iteratively when possible to cut down on the possible causes of the problem
• When in doubt, draw pictures
  – Draw graphs of resources and their consumers
  – List the order in which things are acquired
  – Look for inconsistent orders of acquisition and circular dependencies

5 C Type Qualifiers

During your explorations through the OS/161 source you might have noticed various C-Type qualifiers used by the OS. For instance, in the semaphore struct we have declared volatile int sem_count. Here is the general information on type qualifiers in C declarations.

• volatile: A volatile variable can be changed by other processes or hardware. This tells the compiler not to create any optimizations around this variable. That is, every time we want to read from it, we should retrieve the actual value. Depending on some cached register value could get us into trouble. Why might the semaphore’s sem_count qualify as a good place to use a volatile variable?

• const: A const variable is one that you promise not to modify. This might allow the OS compiler to make certain optimizations, knowing this guarantee that the value will not be modified. Can you think of a few places this might come in handy? This differs, however, from a true C constant.

One example from OS/161 is in include/types.h, where we find:

typedef const struct _userptr *const _userptr_t;

Why might we want to declare a _userptr as const?

A short quiz: What’s the difference between:

• int const *i; and
• int * const i;?

6 Design Document

Assignment 2 will ask you to write and submit a design document to the TFs before you start coding. We realize that some of you might still be putting the finishing touches on Assignment 1, so this isn’t meant to scare you. We just wanted to get this information out to you as soon as possible, seeming that next week’s sections come right before the design document due date.
The CS 161 design documents are, in the TFs’ opinion, the single most important piece of work you will do in this class. (Corollary: They are the single most important piece of work you will do in your life.) Yes, other CS classes have required you to submit a “design” “document”. But let’s be honest, in such classes you tabled the design document and hacked away at the assignment until it worked, only to go back and write the design document at the end.

This method, however, will not work in CS161. The motto here is “measure twice, cut one”. We give you a week to complete the design document for a reason; we want you to spend a significant amount of time thinking about how the pieces will fit together (and what new pieces you might need), before you run off to hack them together. We give you 2 weeks after the design document due date to code, so there should be no need to rush this process.

To further motivate you, we remind you that the design is worth 30% of your final grade for the assignment. You do not want to spend four nights perfecting your system only to lose 10 points on the assignment because your design document wasn’t of right level of sophistication / completeness.

That brings us to an important question. What is a design doc and how do I write one? A good design document is as valuable as the code you will write. A design document should be a document which will allow a good programmer to write working code, even if that programmer doesn’t know the internals of os/161 too well. In other words, a design doc should reflect all the research and brainstorming you did before attempting the coding task. Do not shy away from details!

Below are guidelines for writing a good design doc. Again, don’t feel like you have to stick to them. But if you don’t even know where to start, these few bullet points are a good place to look.

- **Introduction** – briefly mention what problems you will encounter in completing the assignment. Of course, some problems you won’t be able to forecast, but the more you pick out now, the better off you will be. A brief description of your goals or overall technique you’ll employ is useful, but not required.

- **Overview** – give an overview of your design. Some things are always in fashion: data structures used (**very useful** for later assignments), pseudocode (**i.e.** syntax-sloppy C code), algorithms, a list of functions to write/change. Explain why each main variable you introduce is useful. Explain why your solution works (briefly).

- **Topics** – Break up your design into appropriate topics and discuss the details of each topic as explained below. For example, in assignment 2, a good topic breakdown is as follows:
  - Identifying processes
  - File descriptors
  - Scheduling
– fork
– execv
– Other system calls
– Synchronization issues

A useful thing to touch upon here is the interaction between different components: how will fork interact with the file descriptors? How do other system calls use process identification?

Sometimes it might be useful to write down which files you will need to modify for each part. This way you will get yourself thinking about file hierarchy, which may still seem a little confusing. Spend more time reading through the os/161 code to see where these pieces should be placed.

• Functions – describe each function you have to implement. Talk about the algorithms you are going to use and why. Rough pseudocode is probably not a bad idea. Identify subtleties and other problems. If there are helper functions you are going to need to write, identify, prototype, and describe them.

• Plan of Action – divide up the work between you and your partner, and set some milestones for when you are going to complete that work. While the milestones will be advisory, I am interested in making sure the workload is balanced between the two of you, so make sure that is accurate. This assignment breaks down into independent pieces pretty readily.

Students in the past have always had a good laugh when they looked back at their design documents, which outlined in detail what each person will do on which day and compared this with what actually happen. For example, one first design document featured the following “breakdown”:

<table>
<thead>
<tr>
<th>Friday</th>
<th>PIDs, getpid, descriptors,</th>
</tr>
</thead>
<tbody>
<tr>
<td>More Friday</td>
<td>chdir, getcwd, scheduler</td>
</tr>
<tr>
<td>Saturday</td>
<td>i/o, exit, waitpid, execv</td>
</tr>
<tr>
<td>Sunday</td>
<td>fork, scheduler</td>
</tr>
<tr>
<td>Monday</td>
<td>test</td>
</tr>
<tr>
<td>Tuesday</td>
<td>test</td>
</tr>
<tr>
<td>Wednesday</td>
<td>test</td>
</tr>
<tr>
<td>Thursday</td>
<td>test</td>
</tr>
<tr>
<td>Friday</td>
<td>drink</td>
</tr>
</tbody>
</table>

In fact, the actual schedule looked more like this:

| Friday       | nothing                     |
| Saturday     | PIDs, getpid                 |
| Sunday       | nothing                     |
| Monday       | my partner and I panic       |
After having written a design doc, you should look at it and *convince* yourselves that you explained every difficult detail. Usually students write things that translate into “I don’t really know how I’m going to solve it, but somehow I will” and you should avoid this as much as possible. If there is something you don’t know how to attack, talk to your TF, and in the worst case, write something like “I DON’T KNOW HOW TO DO THAT, HELP!!” in your design doc (don’t use caps lock, caps lock is evil. Exercise your pinky instead).

As a rule of thumb, the more code-looking text you include in your design doc (but please spare your TFs all the syntax issues), the better off you will be.

7 Process Basics

*Process states:*

- S_RUN
- S_READY
- S_SLEEP
- S_ZOMB

What do you think a zombie is? Where are these constants defined in OS/161?

*Where does the switching happen?* Look in *thread_switch* and *switchframe_switch*.

Here we have the machine-independent and the machine-dependent switch functions.
How to use processes:

```c
if (fork() == 0) {
    /* We're in the child */
} else {
    /* We're the parent */
}
```

What's a forkbomb? Why shouldn't you do it?

## 8 A Quick Look Into Assignment 2

You are tasked with implementing system calls that allow access to the file system. It should be noted that these file system calls are not implementation dependent – they should work with any file system, due to the higher-level VFS layer. There are a total of eight system calls you need to write: `open()`, `read()`, `write()`, `lseek()`, `close()`, `dup2()`, `chdir()`, and `getcwd()`.

The first thing that needs to be designed is the per-process file table. The file descriptor that you hand back to an application is just the index into the process's file table. You need to decide what goes into this file table. A familiarity with the VFS layer is a must before designing this.

The process management system calls are probably the harder part of this assignment. You will have to implement: `fork()`, `getpid()`, `waitpid()`, `execv()`, and `exit()`.

Next week’s section will cover these system calls in more detail as Assignment 2 ramps up.

## 9 Conclusion

### 9.1 Getting started :: Asst 2

- Begin by breaking down the code.
- Design stuff and have your partner poke holes in your design.
- Always think before coding.
- Don’t expect to get it all right in advance though.
- Start early. You should really start as soon as you hear back from your TF regarding your design.
- If you have a bug and are not making any progress, work on something else for a while. Sleep on it if possible.