

Computer Science 161: Operating Systems Processes, Scheduling, VM, Writing a Design Doc

CS161 Course Staff
updated by Stephen Dawson-Haggerty
{cs161,sdawson}@fas.harvard.edu

March 8, 2007

1 Administrivia

- *Wednesday, March 06, 2007*: 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.*
- *Friday, March 16, 2007*: Assignment 2 due at **5 p.m.**
- Today: All this stuff. Will we finish? Maybe, maybe not.

2 Scheduling

We could spend a lot of time on this, but it's just one topic of many, and (I think) other things are more interesting. Having said that, here we go.

CPU Utilization % of time that the CPU is running threads

CPU Throughput # jobs per second

Turnaround time {End Time} - {Start Time}

Response time total time jobs spend on ready queue

Waiting time total time jobs spend on wait queue

Scheduling algorithms: **FCFS**, **RR**, **SJF**, **SRTF**, **Priority**, **Lottery**, **MLFQ**.

2.1 Round Robin

Key terms: **quantum**.

Given three jobs with the following characteristics, determine the run order, turnaround time, waiting time, and response time of the CPU while it runs these jobs. Assume the schedule is preemptive and the CPU quantum is 10ms.

Job A	5ms CPU	35ms IO	5ms CPU
Job B	25ms CPU	15ms IO	10ms CPU
Job C	40ms CPU		

2.2 MLFQ

A little complicated. But not that complicated. . . Let's talk about it, but skip the example. Where do processes start? When do they get moved between queues? Which process runs next? This might be a good exam question; the trick of getting these is to have a good accounting system on your piece of paper to keep track of all the different things; it's easy to make mistakes.

You could work through the previous problem with a MLFQ scheduler, instead.

3 VM

VM is one of the most useful things you learn in this course, since probably for the first time you no longer need to be surprised by the "magic" the operating system provides. It's also one of Matt's favorite topics, and so will appear on the midterm. This wouldn't necessarily be a great reason to really learn it, but you also have to implement it so understanding what's happening sooner rather than later will save us all a lot of trouble.

Key terms: **page table**, **MMU**, **TLB**, **TLB miss**, **page fault**, **page frame**, **internal fragmentation**, **external fragmentation**

A sample problem: Suppose we are given a computer with a 16-bit virtual addresses, and a page size of 256 bytes. The system uses one-level page tables, which start at address 0x0400. (The first few pages are reserved for hardware flags, etc. Maybe you wanted to have DMA on your 16-bit system, who knows?) Assume page table entries have eight status bits: 1 valid bit, 1 modify bit, 1 reference bit, and 5 permissions bits (this is a very secure system).

- How many pages are there? How much memory do the page tables require?
- Translate the following *virtual* addresses to physical addresses: 0x0a32, 0x052f, 0x0fff.

0x0400	0x0410
0x2375	0x12e7
0x00ff	0xad54
0x2fee	0x5fcc
0x3512	0x02f9
0x43aa	0xdead
0xa42f	0xbeef
0xed65	0xcafe
0x8024	0x8410

4 Design Document

The CS 161 design documents are, in the TFs’ opinion, the single most important piece of work you will do in this class, and, some argue, ever do in life. As opposed to other CS classes you might have taken that required you to submit a “design” “document”, in this class doing them right is crucial. So crucial, in fact, that **gwa** gave you seven more hours to write your design doc (the deadline is 11:59pm on Wednesday).

To further motivate you, I 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?* In my opinion a good design doc is at valuable as the code you’ll write. A design doc should be a document which will allow a good programmer 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.

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, those 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.

- 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”:

```

Friday    PIDs, getpid, descriptors, chdir, getcwd, scheduler
Saturday  i/o, exit, waitpid, execv
Sunday    fork, scheduler
Monday    test
Tuesday   test
Wednesday test
Thursday  test
Friday    drink

```

In fact, the actual schedule looked more like this:

```

Friday    nothing
Saturday  PIDs, getpid

```

```
Sunday    nothing
Monday    my partner and I panic
Tuesday   descriptors, chdir, getcwd
Wednesday scheduler, we panic some more
Thursday  i/o, exit, waitpid (procrastinating before execv and fork, really)
Friday    execv, fork, test, test, test, test
Saturday  sleep
Sunday    sleep
Monday    drink
```

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. First, let’s remind ourselves of all that meaty system stuff.

5 The 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 at `mi_switch()`, `md_switch()`, and `mips_switch()`. (That stands for machine-independent and machine-dependent.)

How to use processes:

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

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

Let's now get into the details of the assignment.

6 File Descriptors and System Calls

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. There are a number of things to note here:

- How are open files represented in user space? File descriptors.
- There are three standard file descriptors, `STDIN`, `STDOUT`, and `STDERR`. To what device should these be initially attached?
- File descriptors are indexes into the file handles. How do you want to assign new file descriptors, and should there be a limit to the number of file descriptors? How do you assign file descriptors?
- Each file descriptor has an associated file offset. What happens with the offset if you open the same file twice?
- How should you convert between file descriptors and more meaningful information? How are open files represented in the kernel? `vnode`. Keep track of this correspondence using the file table.
- The file table helps associate open file descriptors with the corresponding `vnodes`, keeps track of the current file offset.
- What happens to a file table when you fork a process, or when the process exits? Each process has one file table, thus it is copied the a process forks. What happens to the seek position when you:

– `dup2()`?

- `fork()`?
- open the same file twice?
- When a process exits, its a good idea to close all its open files (be careful here. What if the process has forked).

The system calls you need to implement are:

- `open(const char *path, int oflag, mode_t mode)` - should be as simple as it seems. Hopefully it seems simple?
- `dup2(int oldfd, int newfd)` - if `newfd` is already opened, close it. Upon successful completion, both file descriptors refer to the same file table object and share all properties of the object.
- `close(int fd)` - might be a little bit more interesting. Why? Refcounting
- `lseek(int fd, off_t offset, int whence)` - can we always perform an `lseek()` beyond the end of the file? Is that legal? Use `VOP_TRYSEEK`
- `read(int fd, void *buf, size_t nbytes)` - you will want to use `struct uios`, so make sure you understand them. You may want to check out `VOP_READ`.
- `write(int fd, const void *buf, size_t nbytes)` - involves I/O to user-land. Again, `struct uios` will be helpful, as will `VOP_WRITE`.
- `getcwd(char *buf, size_t size)` - this should be very straightforward.
- `chdir(const char *dir)` - again, you should thank us for doing so much of the work for you.

7 Process Management

The process management system calls are probably the harder part of this assignment, so we'll just delve right in.

- What is a process and how does it relate to an `os161` thread?
- `fork()` -
 - As discussed in lecture, this system call creates an exact copy of the process that calls it. What does that entail?
 - How would you copy the file table? What happens to open descriptors?
 - You need to duplicate the address space. Look through the `dumbvm` code for a function that might be quite useful.

- Is there anything else you will need to replicate/copy? Current directory
 - The new process should get a PID. Here `fork()` needs to abide by the semantics of your PID management mechanism.
 - The trickiest part is making the child return 0 and behave exactly like the parent. It will be useful to understand the machine dependent system call mechanism for return values and errors in `kern/arch/mips/mips/syscall.c`. You will probably want to do something similar in `md_forkentry()`. This is very subtle – think about this and be sure to address it in your design document.
 - Trapframe handling – this can be tricky so be sure to discuss it in your design. When a process makes a system call, where how does it know where to return? - It saves return address on the trapframe. Therefore, the trapframe also needs to be copied. (Otherwise, child process would not know where to return.)
 - Return 0 to the child, process id of the child to the parent. How to return a different value to the child process? Look at how other system calls return and fiddle with trapframe appropriately. This is machine-dependent code. Place it into an appropriate directory. Look at `md_forkentry`. The skeleton is conveniently provided.
- `getpid()` - this should be the simplest function you write all semester. And it can't even fail!
 - `waitpid(pid_t wpid, int *status, int options)` and `_exit()`
 - These system calls are very closely tied to your PID management system. It's a synchronization problem.
 - How are you going to assign PIDs? You are not allowed to just run out. PID recycling?
 - Be very specific when you define the semantics of your `waitpid()` and `_exit()` interactions. The man pages give the minimum requirements. Note that when a process `_exits()` its PID may not be given to a new process right away since the parent might be interested in finding the exit code. What if the parent has already exited itself?
 - What does *interested* mean? Unix defines it strictly in terms of parent/child. The parent can get the child's exit status.
 - You may wish to implement `WNOHANG` for `waitpid()`, which allows `waitpid()` to be non-blocking.
 - What PID related data structures are you going to keep in the parent and in the child? Do you need to synchronize the access to those structures and if yes then how?
 - How can you make a parent wait for a child? What happens if a child tries to wait for its parent?

- How can you deadlock? (You shouldn't, of course.) Two processes waiting for each other
 - `_exit()` releases all resources used by the process. What are these? Do we always free *all* resources? What about the exit code. What happens if a child exits before its parent or before any other process that has "expressed interest" in the exit status?
 - Don't forget `kill_curthread()`
- `exec(const char *path, const *char argv[])` -
 - The idea is that we want to load a new executable in the address space of a process.
 - `execv()` is quite similar to `runprogram()` in `kern/userprog/runprogram.c`.
 - Load the executable. Let's see what `runprogram` does.
 - * It opens the file.
 - * Creates an address space into which the image would be loaded and activates it. (Don't worry too much about what this does until assignment 3, but remember to do it).
 - * Loads the executable into that address space.
 - * `load_elf` returns `entrypoint`. What is this?
 - * Define user stack.
 - * Return to user mode.

You can just follow all these steps but before 6, you need to `copyout` arguments to the user stack. Anything else? What about the old address space?

 - Coping with the argument vector is the hard part. This is hard and subtle.
 - Where do we get the arguments from the old program? They are user-level pointer that are passed as arguments to the system call. How can you get hold of them? `copyin` for pointers, `copyinstr` for strings. You need to `copyin` both the pointers and the strings. Where in the process's address space should we put the argument vector? On the stack. Where do we get the arguments from the old program? They are user-level pointer that are passed as arguments to the system call.
 - What is the stack? Its just a region of memory in the address space. Stack is used for temporary data during function calls. Why do we need it? To make efficient use of memory. Now getting the arguments into place is basically the opposite of getting them from user space. You will again need `copyin/copyout`. Place things wherever you want, but above the stackpointer. Stackpointer is where the process will start scratching on the stack.

- Note that the argument vector comes from user space. The functions in `kern/lib/copyinout.c` will be very useful in copying everything to/from the kernel address space. Why is this always important? Consider this example. Kai-Hua has a secret file. He's been using it, so its buffered in memory. Malicious dude Mal wants to get to it, and he has a pretty good guess where its buffered. Malicious Mal can pass a kernel address to open with `O_CREAT`. Viola, the filename appears with the contents of the file. This is **bad**, which is why we can't trust any thing coming from userland. Thus, `copyin/copyout`.
 - Each of the elements of the argument vector (`argv[i]`) is a string residing in userland and needs to be copied with care.
 - What happens to `argv[i]` in the new address space?
 - Make sure you null-terminate `argv` (i.e. `argv[argc] = NULL`).
 - Make sure that all of your pointers are word-aligned.
 - Don't forget to set up `stdin`, `stdout` and `stderr` (also in `runprogram`).
- Remember to handle all the corner cases. Can you think of some good ones for these system calls?

Since handling arguments in `execv` is so hard, lets do an example. We need to place `ls foo` on the user stack. This is what it should look like.

800	
799	<code>∅</code>
798	<code>o</code>
797	<code>o</code>
796	<code>f</code>
795	<code>[padding]</code>
794	<code>∅</code>
793	<code>s</code>
792	<code>l</code>
791	<code>∅</code>
790	<code>∅</code>
789	<code>∅</code>
788	<code>∅ [null-terminate]</code>
787	<code>argv[1]</code>
786	<code>argv[1]</code>
785	<code>argv[1]</code>
784	<code>argv[1] = 796</code>
783	<code>argv[0]</code>
782	<code>argv[0]</code>
781	<code>argv[0]</code>
780	<code>argv[0] = 792 = stackptr</code>

Why is there nothing in location 795? Because pointers have to be word-aligned (word = 4 bytes). Why? CPUs are designed this way. So make sure all your pointers are aligned on a 4-byte boundary. Otherwise your user program will crash, and you won't know why.

Why are 4 locations occupied by `argv[0]` and `argv[1]`? Because pointers are 4 bytes long.

Hopefully this has made things a little clearer.

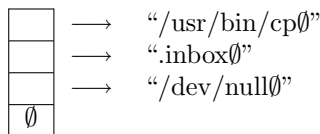


Figure 1: An illustration of an example `char **argv`, where `∅` represents the traditional C `NULL` value.

8 Schedulers

You must implement two schedulers to complement the one we have provided for you. You may choose whatever scheduling algorithms you'd like, as long as they're not substantially similar to each other. Choices include a round robin with priority, multi-level feedback queue scheduler, fair-share, a lottery scheduler, a priority scheduler, and a random scheduler. Keep the original round-robin scheduler, it can come in handy for testing etc (this of course does not count as one of the two you need to implement).

You should provide a mechanism for switching between the schedulers. You can have them chosen at compile time with some `#define` directives. You can use the config system to set these `#defines` (the assignment explains how to do this). Switching schedulers at runtime is neat but impractical and usually not worth the trouble.

9 Conclusion

9.1 Getting started

- Look at the survival guide.
- Being 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.
- If you have a bug and are not making any progress, work on something else for a while. Sleep on it if possible.

9.2 Debugging Hints

- Revisit debugging hints from previous section notes (especially the Intro section).
- Examining the sleep queue:

```
p sleepers.num
p *((struct thread *) sleepers.v[0])
p *(struct lock *) ((struct thread *) sleepers.v[0])->t_sleepaddr
```

- Break on `mips_trap` or on various points in `mips_trap`
- Set horizon breakpoints when stepping.
- If you want to monitor the value of something, get its address and then use `display` like this:

```
display *(type *) address
```

- Debugging user code: Sometimes, but not always, you can get at user memory from `gdb`. You can always go into `dumbvm` and find the physical address, then look there. You can also use `trace161 -tu` to see exactly what's going on.
- Remember that we are implementing standard system calls, so if you're not sure of some functionality, you should do one of the following:
 - Look at the `os/161` man pages
 - Look at the Unix man pages (there might be some differences, but the man pages have nice examples and explain things a little better)
 - Write your own C programs and test the functionality yourself!