Slide 1

Administrivia

- Reminder: Reading quiz 1 due Monday. Remember to turn by putting a PDF in the "turn-in" folder I set up for you on Google Drive. (I'm trying this instead of e-mail as in past semesters.)
- Reading quiz 2 posted. Due the following Monday.

Slide 2

Minute Essay From Previous Lectures

- I think everyone who reported on what they did during the summer had something interesting to report! many internships and research experiences, plus some non-CS projects (not a bad thing).
- Pretty much everyone liked the new textbook (and not only for the price). The main page has a (shameless?) plea for support. Consider contributing a few dollars! (I did, in addition to buying downloadable PDF and hardcopy.)

Slide 3

Processes and "Virtualizing the CPU"

- One thing we want an operating system to make possible is having multiple applications "open" at the same time, possibly more than we have processors to run them.

  Useful to think of each "open application" as an instance of "process abstraction".
- (Aside: Terminology can be confusing. Here, "processor" means anything that can execute a sequence of instructions, and can be a totally independent chip or one "core" on a multicore chip. An older term is "CPU". Possibly no standard term. A while back my research collaborators and I decided to use "processing element", but it hasn't caught on.)
- How? "Virtualize the CPU".

Slide 4

Virtualizing the CPU

- Big picture is that we want to have (conceptually) arbitrarily many processes, each with an "address space" (memory it can use — so, yes, we will have to "virtualize memory" as well, which is the next big topic). Need to "time-share" CPU, "space-share" memory.
- The "crux" of the problem is how to implement this, with reasonable efficiency. A lot about this is done should seem kind of like common sense, if you keep this in mind.

Slide 5

Process — Abstraction

- In CS terms, define an abstraction in terms of possible values, operations.
- Possible values here are a little complicated — some representation of the state of the process.
- Many operations possible; some basic ones include “create”, “destroy”, “wait”, and “signal”.

  Encapsulated as “process API” / set of system calls.

Slide 6

Process State — Implementation

- Key here is to include everything important-in-context, so includes:
- Machine state (e.g., contents of registers). (Might be a good time to reflect on what you remember from CSCI 2321.)
- Information about address space — e.g., where it is in physical memory. Curiously enough, its *contents* aren’t part of this state! (Which is okay, because we might want another process to change something.)
- List of open files.

Slide 7

Process Creation — Implementation

- Lots involved here. Interesting to reflect on this in the context of what you know about program startup from application perspective? Some things:
- Load program from disk. Details vary, but code has to be in memory to run, not on disk. Includes initializing data defined at startup.
- Initialize stack, so stack-based schemes for function calls work.
- Initialize heap, so we can do `malloc` and equivalents.
- Put any arguments on stack.
- Other stuff — e.g., for UNIX, set up some standard "open files".
- Start program (C `main()` or the equivalent).

Slide 8

Process Creation — Implementation, Continued

- Textbook shows semi-real-world example of data structure (figure 4.5).

  Note that this is for an architecture similar to x86, where the registers all have these I-think-ugly semi-symbolic names. Contrast to MIPS's simple r0, r1, etc.!

Slide 9

Some Specifics — UNIX

- Chapter on processes in UNIX is, I think, interesting. Not all operating systems work exactly like this of course but same basic ideas / principles.
- One noteworthy thing — process creation. Other systems *don't* do it the same way (e.g., Windows has a single "spawn process" system call), and it can be a topic for heated discussion!

Slide 10

Process Creation in UNIX

- Unusual in needing not one but two system calls:
- `fork()` to create a new process — which is an almost-exact copy of the calling process! including all its address space. Only difference is return code from function.
- `exec` functions (several options) load new code into process, replacing current code.
- Why oh why? can be useful to retain some things such as list of open files.
- Big potential performance hit; think a minute about what it is before going on.

Slide 11

Process Creation in UNIX, Continued

- Duplicating address space — probably fine when `fork()` was invented and memory was limited. But now?
- Mitigate by not actually making copy all at once — "copy on write" scheme.

Slide 12

Aside: "RTFM"

- Short for Read The Fine Manual. Typical busy local-expert response to questions that the asker could get by doing some homework before asking. (Other choices for "F" are possible.) In days past on UNIX this meant the `man` pages. Distinctly not easy reading, though generally accurate and complete. (Well, except when they reference the competing documentation, the `info` pages. UNIX is about choices! sometimes too many.)
- Why would you want to anyway now that answers to all questions are as close as your favorite search engine? (As if. Most questions, though.)
- As important as knowing how to do things, though, is knowing what things can be done. RTFM can help with that!

Slide 13

Minute Essay

- Questions? Is this material making sense?