Slide 1

Administrivia

- (None?)

Slide 2

Minute Essay From Last Lecture

- (Review question, my answer.)
- Many people at least on the right track!

Slide 3

Process Abstraction

- We want O/S to manage "things happening at the same time" — applications, hidden tasks such as managing a device, etc.
- Key abstraction for this — "process" — program plus associated data, including program counter.
- True concurrency ("at the same time") requires more than one CPU/processor/core. Can get apparent concurrency via interleaving — model one virtual CPU per process and have the real processor switch back and forth among them ("context switch").

  (Aside: In almost all respects, this turns out to be indistinguishable from true concurrency. "Hm!"?)

Slide 4

Process Abstraction, Continued

- Can also associate with process an "address space" — range of addresses the program can use. Simplifying a little, this is "virtual memory" (like the virtual CPU) that only this process can use. More (lots more) about this later. (Nitpick: Yes, we also want to be able to share memory among processes. More about that later too.)
- How to map this to the real hardware? Chapter 2 talks about how to share the real CPU(s) among processes; chapter 3 talks about how to share the real memory.

Slide 5

Context Switches

- What is it? switch from one process to another.
- When should this happen?

Slide 6

Context Switches, Continued

- Should happen
  - when a process's "time slice" is up.
  - when there's an unrecoverable error.
  - when there's something that needs to be done right away (e.g., deal with input/output).
  - maybe other times? (when a process has to wait for something, e.g.).

  All signalled by some kind of interrupt.
- Goal is to suspend work on a process such that we can later pick up exactly where we left off. How do we make that happen?

  (Think about what the hardware does when an interrupt happens, what's included in that "virtual CPU".)

Slide 7

Context Switches, Continued

- On interrupt, hardware saves program counter (at least — why?), transfers control to fixed location — which contains O/S code.
- That O/S code has to
  - Save CPU state (program counter, registers, etc.) for the current process.
  - Deal with interrupt (details depend on type — I/O versus timer versus . . .).
  - Restore CPU state for "next" process (previously saved), thereby restarting it.

    ("Next" process? yes, O/S might have to choose — more about that later.)

Slide 8

Process Creation and Termination

- When are processes created?
  - At system startup.
  - When another process makes a "create process" system call — e.g., to start a new application.
- When are processes destroyed?
  - At program exit.
  - After some kinds of errors.
  - When another process makes a "kill process" system call.

Slide 9

Process States

- Can think of processes as being in one of three states:
  - "Running" — being executed by a CPU.
  - "Blocked" — waiting for something to happen (I/O to complete, another process to do something, etc.) and unable to do anything useful until it does.
  - "Ready" — not blocked, but waiting because all CPUs are currently executing other processes.
- Possible transitions? Which ones require decision-making?

Slide 10

Process States, Continued

- Possible transitions (Figure 2-2):
  - Running to blocked — happens when, e.g., a process makes an I/O request and can't continue until it's complete.
  - Blocked to ready — happens when the event the blocked process is waiting for occurs.
  - Running to ready, ready to running — needed if we want some sort of time-sharing (give all non-blocked processes "a turn" frequently).
- Notice that moving to and from "blocked" state doesn't involve decision-making, but ready/running transitions do.
- The decision-maker — "scheduler" (to be discussed later). Often "running to ready" is triggered by an interrupt (I/O, timer, etc.), and "ready to running" involves this scheduler.

Slide 11

Implementing Processes

- Think about how you would implement this abstraction . . .
- First, you'd want a data structure to represent each process, to include — what?

Slide 12

Implementing Processes, Continued

- Data structure to represent each process would include some way to represent such things as:
  - Process ID.
  - Process state (running / ready / blocked).
  - Information needed for context switch — a place to save program counter, registers, etc.
  - Other stuff as needed — e.g., a list of data structures for open files.
- Then you'd collect these into a table (or some similar structure) — "process control table", with individual data structures being "entries in the process control table" or "process control blocks".

Slide 13

Implementing Processes, Example — Linux

- Each process (“task”) is represented by a C `struct` containing information similar to what we described.
- These `struct`s are chained as a doubly-linked list; there is also a hash table keyed by PID.
- (This is according to online information about the 2.4 kernel.)

Slide 14

Processes Versus Threads

- So far I’ve used “process” in an abstract/general way.
- In typical implementations, though, “process” is more specific — something that has its own address space, list of open files, etc. Often these are called “heavyweight processes”.
  - Advantages — such processes don’t interfere with each other.
  - Disadvantages — they can’t easily share data, switching between them is expensive (“a lot of state” to save/restore).
- For some applications, might be nice to have something that implements the abstract process idea but allows sharing data and faster context switching — “threads”.

Slide 15

Threads

- So, threads are another way to implement the process abstraction.
- Typically, a thread is "owned" by a (heavyweight) process, and all threads owned by a process share some of its state — address space, list of open files.
- However, each thread has its own "virtual CPU" (a distinct copy of registers, including program counter).
- Implementation involves data structures similar to process table.
- Advantages / disadvantages (compared to processes)?

Slide 16

Threads, Continued

- Advantages: threads can share data (same address space), switching from thread to thread is fairly fast.
- Disadvantages: sharing data has its hazards (more about this later).

Implementing Threads

Slide 17

- Two basic approaches — "in user space" and "in kernel space" Various hybrid schemes also possible.
- Basic idea of "in user space" — operating system thinks it's managing single-threaded processes, all the work of managing multiple threads happens via library calls within each process.
- Basic idea of "in kernel space" — operating system is involved in managing threads, the work of managing multiple threads happens via system calls (rather than user-level library calls).
- How do they compare?. . .

Implementing Threads, Continued

Slide 18

- Implementing in user space is likely more efficient — fewer system calls, so less overhead.
- Implementing in kernel space avoids some problems, though:
  - If a thread blocks, it may do so in a way that blocks the whole process.
  - Preemptive multitasking is difficult/impossible without help from the kernel, as is using multiple CPUs.

Slide 19

Adding Multithreading

- As you know if you've written multithreaded applications, moving from single-threaded to multithreaded not trivial:
  - Figure out how to split up computation among threads.
  - Coordinate threads' actions (including dealing properly with shared variables).
- Similar problems in adding multithreading to systems-level programs:
  - Deal properly with shared variables (including ones that may be hidden — e.g., in implementations of system calls).
  - Deal properly with signals/interrupts.

Slide 20

Sidebar: Signals

- Textbook mentions that one complication of adding support for threads is dealing with "signals". It may not be clear what those are.
- Signals are a mechanism used by UNIX-family operating systems for one form of interprocess communication, sort of a software equivalent of hardware interrupts.
- Signals can arise from hardware error interrupts (e.g., invalid memory address), from user input (e.g., control-C from console), or from another process (e.g., `kill` command).

Slide 21

Signals, Continued

- O/S delivers signal to process, which can choose to accept it or block it; if it accepts it, it can take a default action (e.g., ignore, or terminate process), or it can provide its own handler.
- If the process contains multiple threads, however . . . Implementation of threads must decide what happens then.

Slide 22

Implementing Threads, Example — Linux

- Early versions of Linux provided no support for kernel-space threading, but there were libraries for user-space threading (e.g., "green threads" for Java).
- More-recent kernels provide support, but in an interesting way — threads in some ways are just processes with some different flags allowing them to share memory, etc.

  Adding support for threads complicates process creation — the basic mechanism (`fork`) duplicates an existing process, and if that process is multithreaded, things can be interesting. Some details in chapter 10, or read the POSIX standard for `fork`.

Minute Essay

Slide 23

- In a system with 8 CPUs and 100 processes, what are the maximum and minimum number of processes that can be running? ready? blocked?
- How are you doing with regard to getting a copy of the textbook?

Minute Essay Answer

Slide 24

- Blocked: Maximum of 100 (unless you assume that there's an "idle" operating system process that runs when nothing else does and never blocks, and maybe one of these is needed for every CPU). Minimum of 0.
- Running: Maximum of 8, because there are 8 CPUs. Minimum of 0 (again unless you assume that there's an O/S process that runs when nothing else does).
- Ready: Maximum of 92, since all CPUs will be running processes if there are any that can be run. (Depending on details, you might have to add "except during context switches, when the scheduler is choosing the next process to run on a CPU".) Minimum of 0, since they could all be blocked or running.