Slide 1

Administrivia

- Reminder: No class Friday (campus-wide retreat).

Slide 2

Minute Essay From Last Lecture

- Difficult: Getting re-acclimated to C. (That was partly the point of the assignment — one more opportunity to practice with the language.) Also figuring out exactly what parameters to pass to the system-call library functions.
- Interesting: That it's not that hard to implement a simple shell. (Adding features could be an entertaining side project?)
- Interesting: How `fork` and `execve` work.
- Interesting: How many system calls are needed for even simple programs. (That also was partly the point of that problem.)
- Interesting(?): That `cd` apparently doesn't work. (Why?)

Slide 3

Process Abstraction — Review/Recap

- Processes are a key abstraction in “o/s as virtual machine”. Each can be thought of (at least to some extent) as a program running on its own CPU with its own memory (“address space”). (Nitpick: We probably also want some way to allow processes to share some memory, but — later.)
- How to map this to the real hardware? in this chapter we talk about how to share the real CPU(s) among processes; in the next chapter we talk about how to share the real memory.

Slide 4

Context Switches — Review/Recap

- Sharing real CPU(s) among processes probably means we need a way to “timeshare” among them. An obvious(?) way to do that involves executing code from one process for a while, then switching to another, with the idea that when we come back to the first process we pick up where we left off.
- Context switches normally (always?) triggered by various kinds of interrupts.
- Details of what happens in a context switch all pretty much flow from these two things.

Slide 5

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 6

Process States, Continued

- Possible transitions (figure in textbook, p. 90):
  - 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 7

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 8

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 9

Implementing Processes, Example — Linux

- Each process (“task”) is represented by a C `struct` containing information similar to what we described.
- These `structs` 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 10

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 11

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 a “virtual CPU” (a distinct copy of registers, including program counter).
- Implementation involves data structures similar to process table.
- Advantages / disadvantages (compared to processes)?

Slide 12

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).

Slide 13

Implementing Threads

- 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?. . .

Slide 14

Implementing Threads, Continued

- Implementing in user space is likely more efficient — fewer system calls.
- 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 15

Adding Multithreading

- 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).
  - Deal properly with signals/interrupts.

Slide 16

Implementing Threads, Example — Linux

- Early versions of Linux provided no support for kernel-space threading, but there were libraries for the user-space version.
- More-recent kernels provide support, but in an interesting way — threads in some ways are just processes with 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

- None — quiz.

Slide 17