Slide 1

Administrivia

- (None.)

Slide 2

Minute Essay From Last Lecture

- (Almost no one got it completely right, though admittedly many people didn't have time to try. Review my answer.)

Slide 3

Process Abstraction — Review/Recap

- 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 (more properly now, “more than one CPU/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”).
- Can also associate with process an “address space” — range of addresses the program can use.

Slide 4

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 5

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 6

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 7

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 8

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 9

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 10

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 11

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 12

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 13

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`.

Slide 14

Interprocess Communication

- Processes almost always need to interact with other processes:
  - "Ordering constraints" – e.g., process B uses as input some data produced by process A.
  - Use of shared resources — files, shared memory locations, etc.
- Use of shared resources can lead to "race conditions" — output depends on details of interleaving.
- Processes must communicate to avoid race conditions and otherwise synchronize.
- "Classical IPC problems" — simplified versions of things you often want to do.

Slide 15

Mutual Exclusion Problem

- In many situations, we want only one process at a time to have access to some shared resource.
- Generic/abstract version — multiple processes, each with a "critical region" ("critical section"):

```
    while (true) {
        do_cr();        // must be "finite"
        do_non_cr();    // need not be "finite"
    }
```

- Goal is to add something to this code such that:

1. No more than one process at a time can be "in its critical region".
2. No process not in its critical region can block another process.
3. No process waits forever to enter its critical region.
4. No assumptions are made about how many CPUs, their speeds.

Slide 16

Minute Essay

- None — quiz.