Slide 1

Administrivia

- (By e-mail.)

Slide 2

Concurrency and Threads — A Few Words First

- Discussion of concurrency traditional topic in O/S courses.
- Favorite topic of mine because my much-neglected research is parallel computing, and some of my most interesting courses in graduate school were about concurrent algorithms.
- I'm apt to spend more time on the topic; good news / bad news is that this isn't an option this year. But it *is* one of the more intellectually interesting parts of the course (as opposed to details).

Slide 3

Threads

- In discussion so far we've talked about a "process abstraction", as one of a set of concurrently-executing things.
- In most respects, however, this abstraction is implemented both as discussed so far ("heavy-weight process") and in the form of threads, are roughly the same *except* they share an address space and a few other data structures (e.g., list of open files).
- (Things get a little confused in that a system that supports threads has both processes and threads, and details of combining them not clear: Every thread has a containing(?) process, with an address space etc., okay. But does every process have to contain at least one thread? More later.)

Slide 4

Threads — Basic Elements

- Each thread has a "virtual CPU", with a place to store registers, a state (ready, running, or blocked), etc.
- Each thread does *not* have its own address space; instead each can be thought of as existing inside a process with an address space.
- Each thread *does* have its own stack (which rather breaks the tidy model of memory).

Why Threads? One Reason — Parallelism

Slide 5

- One reason — improved performance via “parallelism”.
- Idea here is to split work into pieces that can be done at the same time and divide among processors/cores.
- Sometimes easy (except maybe for a few details), sometimes more difficult, sometimes just plain impossible. This is what topic of “parallel computing” is about.

Why Threads? One Reason — Hiding Latency

Slide 6

- Way back in mainframe days, “multiprogramming” invented as a way let CPU make progress on one program if another was blocked.
- Similarly, if one part of a computation can’t make progress, but some other part can make progress while the first one is waiting, can set up two threads and put blocking part in one.
- Good way to think about many currently-common applications, such as GUIs: Useful to think of them as having a thread that waits for user input and a thread that manages display. (Java and Scala work this way!)

Slide 7

Threads — Example

- This may be familiar from short discussion of multithreading in CS2? Programs can launch threads, wait for them to finish.
- Note that exactly how program runs depends on scheduler (threads are subject to scheduling, as processes are), so may be different every time.

  And of course implications if multiple threads access shared variables . . .

Slide 8

Threads — Access to Shared Data

- Key point is that what threads execute is *sequences of machine instructions*; while the instructions from Thread A execute in normal order with regard to each other, there are no guarantees how they execute with regard to instructions from Thread B — can be interleaved in any arbitrary order, or even at the same time.
- If results can depend on details of scheduling — "race condition". Not invariably bad, but usually. (How could they be not bad . . . You may remember that floating-point addition is not associative? This means parallelizing some calculations effectively leads to race conditions — but if results are close that may be acceptable.)
- Textbook illustrations are all x86, but the same thing happens in MIPS assembler: If you think about two threads both adding to a variable, each has to first load, then add, then store. (I have a favorite bank-balance example.)

Slide 9

Mutual Exclusion

- Problem of avoiding race conditions referred to as “mutual exclusion problem”, and discussion goes back to early mainframe days, in the context of processes rather than threads but same ideas.
- Simplest of so-called “classical IPC problems” — simplified versions of things real programs (both applications and O/S) need to do. Edsger Dijkstra a key player in developing these ideas (though not the only person).
- Probably this historical context is one reason concurrency still discussed in O/S courses! O/S was first large-scale program to deal with multiple things happening in-effect-at-the-same-time.

Slide 10

Mutual Exclusion Problem — Classical Formulation

- 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 11

Mutual Exclusion Problem, Continued

- Various solutions possible:
  - Using only hardware features always present (some notion of shared variable).
  - Using optional hardware features.
  - Using "synchronization mechanisms" (abstractions that help solve this and other problems).
- Recall that a correct solution
  - Must work for more than one CPU (processor).
  - Must work even in the face of unpredictable context switches — whatever we're doing, another process can pull the rug out from under us between "atomic operations" (machine instructions).

Slide 12

"Atomic Operations"

- In many contexts, want to think about computation in terms of "atomic" operations. (Name goes back to when atoms were believed to be indivisible; idea is that an atomic operation executes as one indivisible thing, without interference from other process or thread.)
- At hardware level, instructions are atomic, but keep in mind that one program source line often translate to more than one instruction. Possibly worse for load/store architectures.
- Which of the following are atomic?
  - `x = 1;`
  - `x = x + 1;`
  - `++x;`
  - `if (x == 0) x = 1;`

  (Or does it depend? On what?)

Slide 13

Atomic Operations, Continued

- At application level, often success of an operation depends on what can be regarded as atomic. Examples:
- For multithreading, a key way to avoid race conditions — somehow package up more-than-single-instruction updates, possibly including a test.
- For disk I/O operations, "journaling filesystems" package multistep operations (e.g., assign an unused block to a file and take it out of the list of free blocks) so they can be considered atomic. (They aren't really, but the filesystem has ways to recover cleanly if one is interrupted.)

Slide 14

Other Classical Problems

- Other situations in which you want one one process/thread to wait for another, such as having one wait for another to do I/O.
- "Bounded buffer" posits collection of "producers" putting items into a limited-size shared buffer and "consumers" taking them out, with producers waiting if buffer full and consumers waiting if empty.
- "Dining philophers" posits — well, description is kind of silly and anthropopic, but it's representative of some resource sharing problems more complex than mututal exclusion.

Slide 15

Minute Essay

- What other exposure have you had to multithreaded programming? I hear that it's presented at least briefly in CS2. How much of it has stuck with you? Were you asked to write any programs?
- Have you had any other background in programming involving more than one thing at a time? possibly a networked game in CS2, or something in another context?