Slide 1

Administrivia

- Quiz 5 graded, and I'm very close to being done with Exam 1(!). I'll do another round of grade summaries when they're done so you know your scores. Returning your papers — I'm hoping to get help from the ASO with the scanning. May take a few days but on the to-do queue. As is grading your homeworks!
- Quiz 5 solution posted (on Web). Homework 6 solution posted (on Google Drive). Homework 7 solution coming soon, and Homework 8 by start of next week.
- Reminder: Homework 8 due this week (official due date Wednesday but AOK to turn in by Friday).
- Exam 2 time — Doodle poll in process. (Minute essay responses varied, though there were more votes for late in the day than early.) I'll plan to make a decision early tomorrow, so you have until end of today anyway to respond.

Slide 2

Minute Essay From Last Lecture

- Varied responses to question about how much students feel like they're learning.
- Sounds like most of you are at least trying to make the best of things (yay!), but it can be tough.
- Motivation is apparently a problem for many!
- At least one student mentioned that it helps a lot to have homework, as not all classes do. I can believe it!
- (Also refer to my mail-to-all.)

Slide 3

Virtual Machines — Executive-Level Summary

- Increasing interest lately in "virtual machines" / "virtualization". Some purely software (e.g., Java Virtual Machine); others involve or at least rely on hardware.
- Idea actually goes back a *long* time: IBM's VM/370 (1970s), a sort of stripped-down O/S that allowed running multiple "guest O/S"es side by side. Very useful in its time! Physical machines often needed to be shared among people with very different needs w.r.t. O/S. Successors still in use!
- Textbook has other examples; one I recognize is VMware ESX (this is what Trinity uses for its VDI?).

Slide 4

Sidebar: Dual-Mode Operation

- A key issue in designing an operating system is doing so in a way that can "defend itself" against buggy or malicious programs.
- One thing that can help — in fact may even be essential — is having (at least) two "modes":
  - "User mode", in which the currently executing program cannot execute certain "privileged" instructions.
  - "Supervisor mode" (a.k.a. "kernel mode"), in which it can.

  Which is in effect stored in bit of special-purpose register ("Program Status Word" or PSW).

  If this bit says user mode, control logic must raise exception if an attempt is made to execute a supervisor-only instruction. (You now probably know enough to kind of see how this might be done, no?)

Slide 5

Virtual Machines — Semi-Executive-Level Summary

- What the real hardware runs: "Virtual Machine Monitor", a.k.a. "hypervisor" (term analogous to "supervisor", a term for O/S). Interrupts and exceptions transfer control to this hypervisor, which then decides which guest O/S they're meant for and does the right thing.
- All works better with hardware support for dual-mode operation: Guest O/S's run in regular mode; when they execute privileged instructions (as they more or less have to), hypervisor gets control and then can simulate . . .
- Other than that, programs run as they do without this extra layer of abstraction — they're just executing instructions, after all?

Slide 6

Virtual Machines — Semi-Executive-Level Summary, Continued

- Some architectures make this easier than others — they're "virtualizable".
- Interestingly enough(?), IBM's rather old 370 had this, but for many newer architectures needed support has had to be added on, not always neatly. "Hm!"?
- (Textbook has a few more details, in section 5.8.)

Slide 7

Parallel Computing — Overview

- Support for "things happening at the same time" goes back to early mainframe days, in the sense of having more than one program loaded into memory and available to be worked on. If only one processor, "at the same time" actually means "interleaved in some way that's a good fake". (Why? To "hide latency".)
- Support for actual parallelism goes back almost as far, though mostly of interest to those needing maximum performance for large problems. Somewhat controversial, and for many years "wait for Moore's law to provide a faster processor" worked well enough. Now, however . . .

Slide 8

Parallel Computing — Overview, Continued

- Improvements in "processing elements" (processors, cores, etc.) seem to have stalled some years ago. Instead hardware designers are coming up with ways to provide more processing elements.
- One result is that multiple applications can execute really at the same time.
- Another result is that individual applications *could* run faster by using multiple processing elements.

  Non-technical analogy: If the job is too big for one person, you hire a team. But making this effective involves some challenges (how to split up the work, how to coordinate).
- In a perfect world, maybe compilers could be made smart enough to convert programs written for a single processing element to ones that can take advantage of multiple PEs. Some progress has been made, but goal is elusive.

Slide 9

Parallel Computing — Hardware Platforms (Overview)

- Clusters: Multiple processor/memory systems connected by some sort of interconnection (could be ordinary network or fast special-purpose hardware). Examples go back many years.
- Multiprocessor systems: Single system with multiple processors sharing access to a single memory. Examples also go back many years.
- Multicore processors: Single "processor" with multiple independent PEs sharing access to a single memory. Relatively new, but conceptually quite similar to multiprocessors.
- "SIMD" platforms: Hardware that executes a single stream of instructions but operates on multiple pieces of data at the same time. Popular early on (vector processors, early Connection Machines) and now being revived (GPUs used for general-purpose computing).

Slide 10

Parallel Programming — Software (Overview)

- Key idea is to split up application's work among multiple "units of execution" (processes or threads) and coordinate their actions as needed. Non-trivial in general, but not too difficult for some special cases ("embarrassingly parallel") that turn out to cover a lot of ground.
- Two basic models, shared-memory and distributed-memory. Shared-memory has two variants, SIMD ("single instruction, multiple data") and MIMD ("multiple instruction, multiple data"). SPMD ("single program, multiple data") can be used with either one, and often is, since it simplifies things.

Slide 11

Shared-Memory Model (MIMD)

- "Units of execution" are (typically) threads, all with access to common memory space, potentially executing different code.
- Convenient in a lot of ways, but sharing variables makes "race conditions" possible. (Now that you know more about how hardware works you may understand the issues better! A single line of HLL code may translate to multiple instructions . . .)
- Typical programming environments include ways to start threads, split up work, synchronize. OpenMP extensions (C/C++/Fortran) somewhat low-level standard.

Slide 12

Distributed-Memory Model

- "Units of execution" are processes, each with its own memory space, communicating using message passing, potentially executing different code.
- Less convenient, and performance may suffer if too much communication relative to amount of computation, but race conditions much less likely.
- Typical programming environments include ways to start processes, pass messages among them. MPI library (C/C++/Fortran) somewhat low-level standard.

Slide 13

SIMD Model

- "Units of execution" term may not make sense. Parallelism comes from all processing elements executing the same program in lockstep, but with different processing elements operating on different data elements.
- Excellent fit for some problems ("data-parallel"), not for others. Very convenient when it fits, pretty inconvenient when not.
- Typical programming environments feature ways to express data parallelism. OpenCL (C/C++) may emerge as somewhat low-level standard, especially suited for GPGPU.
- Parallel collections (as in Scala) probably fit here. Performance may not be great at this point but may well improve.

Slide 14

Distributed Programming

- All approaches mentioned so far rely to some extent on multiple UEs executing more or less synchronously. Works well for classic high-performance computing, where problems involve relatively frequent need for multiple threads of execution to exchange information. (Think simulation of large-scale physical system.)
- However, with some problems there's less need for thread of execution to communicate (think anything involving exploring multiple more or less independent possibilities).
- Various frameworks exist for this. Sadly, not something I know enough about.
- "Actors" model as used in Scala seems to fit best here.

Slide 15

Minute Essay

- Reminder — Quiz 6 available. High-level questions so may be quicker than some. Do now or before 5pm tomorrow.
- I plan to use Wednesday's class for some exam review and a general "what we did in the course" review, and allow time for you to do course evaluations. Anything else you'd like to do / hear about?
- (Curious about how evaluations are supposed to work? With help from the ASO I can put one "ticket" in each of your invidual Google Drive folders.)