Administrivia

Slide 1

- Reminder: Midterm next Tuesday. Review sheet on Web. Updated "practice problems" on Web.
- Reminder: If you're struggling with the homeworks, or with the course overall — it's okay to ask for help.

Why Parallel Computing

Slide 2

- It's been an article of faith for a long time that eventually we'd hit physical limits on speed of single CPUs, despite interpretation of Moore's law as "CPU speed doubles every 1.5 years."
- But — strictly speaking, Moore's law says that the number of transistors that can be placed on a die doubles every 1.5 years.
- Historically that has meant — more or less — doubling speed and memory size. That seems to be at an end (for now?) — tricks hardware designers use to get more speed require higher power density, generate more heat, etc.
- So, what to do with all those transistors? Provide hardware support for doing more than one thing at a time ("parallelism" or "concurrency").

Slide 3

One Approach — Multicore Chips

- Key idea here — chip includes several “cores”, all sharing one connection to memory.
- Each “core” is a processor in the sense we talk about in introductory and courses and Computer Design; each typically has its own first-level cache.
- To fully exploit this for a single application, need multiple threads (or processes).

Slide 4

Another Approach — Hyper Threading

- Key idea here — chip includes hardware support for having more than one thread at a time “active”, but strictly speaking only a single processing core. Replicated components include program counter, ALU.
- What this allows is very fine-grained concurrency (“more than one logical CPU”), which can hide latency. (Note, though, that performance improvements range from zero to about 30 percent.)
- To fully exploit this for a single application, need multiple threads (or processes).

Slide 5

Another Approach — Clusters

- In addition to hardware support for shared-memory parallelism —
  Ubiquity of networking makes almost any PC part of a “cluster”.

Slide 6

Concurrency Basics

- Textbooks on operating systems talk about “processes” — “threads of control” executing “concurrently”, i.e., at the same time (in fact or in effect).

  Each is a sequence of steps, like the (sequential) programs you’ve written.
- How does it work? Conceptually, all processes not waiting for something (such as I/O) run at the same time. Operating system basically simulates one CPU per thread, with real CPU(s) switching back and forth among them.
- This turns out to be a good mental model for managing applications, and activities of the O/S itself. It also means you could get better performance with more than one CPU/core — can potentially have more than one thing actually running at the same time.
- But there are some potential pitfalls, involving interaction among processes.

Slide 7

Processes Versus Threads

- Two basic ways to implement this idea of concurrent execution — “processes” and “threads”.
- “Processes” don’t (usually) share memory, and must communicate in some fairly restricted way.
- “Threads” do share memory, which is convenient but has potential pitfalls (“race conditions”).

Slide 8

Multithreading in Java

- Interestingly enough, Java has included support for multiple threads from the beginning — probably because it’s a good mental model for GUIs.
- Interaction among Java threads based on “monitors” (see textbooks on operating systems, parallel programming — idea goes back to 1975 papers by Hoare and Brinch Hansen). Java leaves out some aspects of full-fledged idea, but keeps enough to be useful.

Slide 9

Threads in Java

- `Thread` class provides basic functionality. To start a new thread, make a `Thread` object and call its `start` method. Two choices:
  - Create a `Thread` with an object that implements `Runnable` — `run` method has code to execute.
  - Define a subclass of `Thread` that has a `run` method with code to execute.
- Interthread interaction based on (implicit) locks:
  - Every object (and every class) has a lock.
  - `synchronized` methods must acquire lock — so only one at a time can run.
  - `wait` gives up the lock and sleeps; `notify` and `notifyAll` wake up one/all sleeping thread(s).

Slide 10

Threads in Java, Continued

- Other useful methods:
  - `Thread.sleep` makes current thread sleep for some interval.
  - `t.join` wait for `Thread t` to finish.
  - `t.interrupt` interrupts `Thread t` (which can check whether it has been interrupted with `isInterrupted` — safe/approved way for one thread to stop another.
- Can set thread priorities — sometimes useful, but not a substitute for proper synchronization.
- Lots of new threads-related stuff in Java 1.5 / 5.0 (`java.util.concurrent` package).

Slide 11

Uses for Threads in Java

- Formerly many uses for multithreading in GUIs (e.g., animation), but now most can be accomplished with new features of GUI classes (e.g., timers). Still useful, however, if you want something that might take a while to execute in the background.
- Multithreading also potentially useful for improving performance of computationally intensive code.
- Examples later . . . .

Slide 12

Java GUI Classes and Multithreading

- Currently Java GUI classes are implemented in terms of an “event dispatch thread” (EDT) — something that listens (to some part of the operating system/environment?) for “events” (from keyboard or mouse, e.g.) and “dispatches” them by calling appropriate methods associated with GUI components.
- Not all of what’s under the hood is thread-safe, so Sun recommends that all changes to GUI components be done in the EDT. This happens automatically with listener methods. Accesses from the “main” thread and from other threads should use `SwingUtilities.invokeLater`. (See example from last time again.)

Slide 13

Multithreading and the Game Framework

- Listener methods run in the EDT. Other methods run in a different thread.
- Problem? Maybe. Concurrent access to simple primitive types (`boolean`, `int`) is pretty safe — the worst that's likely to happen is that changes made by one thread aren't immediately visible to others. But anything involving more complicated data structures is probably a bad idea without explicit synchronization.

Slide 14

A Little About Homework 3

- In this homework you start writing code for your player, to replace the stick figure in the starter game.
- Key parts of this assignment are making the player
  - interact with different kinds of blocks.
  - move in response to keyboard or mouse input from human player.

  (If these don't apply to your game, talk to me about whether there are reasonable substitutes.)

  For design phase, you just need to describe this interaction.

Slide 15

Homework 3, Continued

- `Player` defines some constants you should use.
- You will implement `KeyListener` or one/both of the mouse-listener interfaces. When you do this, the framework will deliver key and/or mouse "events" to you.
- Most logic will go in `update`, `getUpdateTime`, and the listener methods.

Slide 16

Minute Essay

- None — sign in.