Lecture 1

January 11, 2010

Required background: computer architecture, programming languages, data structures, discrete math

Dictionary definitions:

Concurrent - 1. operating or occurring at the same time 2a. running parallel 3. acting in conjunction 4: exercised over the same matter or area by two different authorities

Concurrency - simultaneous occurence

Working definitions:

Sequential program - sequence of actions (statements) that produce a result (final value or sequence of outputs). Variously called a *process* or *task* or *thread (of control)*

Concurrent program - two or more sequential programs, cooperating to accomplish some goal, and running (at least conceptually) at the same time.

Concurrent programming - the art and science of creating concurrent programs

Hardware terms: *uniprocessor, shared-memory multiprocessor, multicomputer* (separate memories)

Parallel program - a program intended to exploit the capabilities of a parallel computer to achieve increased performance

What makes concurrent programming worthwhile? What situations give rise to concurrent programs?

- true parallelism for performance
- operating systems - device drivers, time sharing
- embedded systems - concurrency *plus* real-time
- distributed systems
- client-server systems
- user interface programs
- re-use existing applications (e.g. Unix pipes)

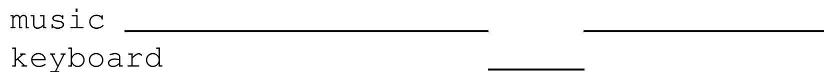
- true parallelism

We are going to ignore true parallelism as a motivation during much of this class. The techniques we study will be applicable to true parallelism but finding and exploiting parallelism in problems is different from what we will be doing. At the end we will perhaps revisit parallel programming.

The concurrency abstraction

We know about sequential programs which in machine-code take the form of sequentially executed instructions, with branching, calling, returning. The next instruction to execute is always determined by the previous instruction. This happens at the rate of *billions* of instructions *per second.* So how long does it take for my computer to process a keystroke – read the key, move the cursor, put up the character on the screen? How many instructions? 1,000? 10,000? 100,000? 1e6? Even 1e6 instructions is less than 0.1% of the available instructions in a second on a modern processor!

Maybe you want your computer to play music while you're writing your paper. It would be unreasonable to make the music-playing program aware of what needs to be done to process your keystrokes. Instead modern systems use *interrupt-driven I/O* to recognize availability of input, switch briefly to handling it, and then go back to what they were doing.



This generalizes to multiple independent activities going on (conceptually) all at once – the idea of processes (or threads, or tasks), with a *scheduler* that *multiplexes* the processor between the different tasks. The first experience people get with the problems of concurrency is often in operating systems where this multiplexing is managed. But the abstraction is more generally useful: e.g., media player that keeps playing music while you navigate its UI, managing the library while in the background it searches for music to add to the library.

What makes CP challenging? Why do we study it as an independent topic? First we need to understand the abstraction we call concurrency. We previously talked about sequential programs. We can model execution of a sequential program as a sequence of *states,* each state consisting of a program counter and value of all the variables of a program (including "hidden" variables like the stack, etc.).

```
<pc, x=1, y=3, ...>
```

Consider the program

```
x = 1; y = 2; z = 3
```

How many states does it have?

– discuss

Our model for concurrency extends the sequential model with a program counter for *each* process (and of course any other hidden variables needed for each process).

```
<pc1, pc2, x=1, y=3, ...>
```

Now consider (// means concurrently with, which implies arbitrary interleaving of steps of the components)

```
x = 1; y = 2; z = 3 // a = 1; b = 2; c = 3
```

How many states does it have?

– discuss

If there are n states in each of m concurrent processes, the program has n^m states based on the possible values of the program counters alone.

Now consider

```
x = x+1
```

How many states?

– discuss

To answer this we have to know something about how such a statement would be compiled, e.g. in a simple accumulator machine it might be

```
LD x,A
ADD 1
ST x,A
```

We adopt a concurrency abstraction that allows *arbitrary interleaving* of *atomic actions* occurring in the different processes.

Now consider

```
x = x+1 // y = x+1
```

How many states? If x is initially 0 what is the final value of y?

– discuss

Why *arbitrary* interleaving? It seems the best model for uniprocessors – interrupt can occur at any instruction; and also for multiprocessors – the multiple processors don't coordinate their actions but we do generally require that the results of truly parallel execution correspond to *some* sequential order.

Finally, consider

```
x = 1; y = 2; z = 3 {z==3} // a = 1; b = 2; z = 2
```

Interference; one thread changes a property that another thread expects to be true

As you can see, writing correct concurrent programs is going to be very hard if understanding what a tiny given program does is this difficult! So why isn't the whole notion of a concurrent programming a folly?

- exponential state growth - the blessing and the curse of concurrent programming. On the one hand CP provides compact and natural representation of real-world world problems with complicated state; the world is complicated – many states must be represented; if these states evolve along more-or-less independent trajectories, a representation as concurrent threads is more compact, and *easier* to understand and reason about than the alternative. (Which is not to say that the reasoning is *easy!*). yet on the other we have to reason about that complicated state to understand our concurrent programs
- common patterns and paradigms for dealing with the ensuing complexity
  - concurrency control mechanisms
  - programs
  - reasoning

State Explosion

What we ask then of our systems for reasoning about concurrent programs, our programming notations and mechanisms for concurrent programming, and our patterns and paradigms for concurrent programs is that they help us manage the complexity that arises due to arbitrary interleaving and interference.

Plan for the semester: we will begin by working with *shared-state concurrency* which will seem the most familiar to you but in which some of the thorniest problems arise. We will primarily use Java as the basis for this discussion but also discuss *pthreads* in C. We then move on to *message passing concurrency* in a *functional language* where the our goal is to see how these characteristics ensure that the complications of concurrency don't impact parts of programs where they are really unnecessary.

Reading and problems for next time:

- Read Chapters 1 and 7 in the Erlang book and Chapter 1 in the Java book.
- Submit a photo and contact info using the turn-in page of the class website: `www.eecs.wsu.edu/~hauser/cs580/turnin`.