Concurrent Programming Lecture 1

Required background: computer architecture, programming, 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 (value). 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 makes it challenging? Why do we study it as an independent topic?

- 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, yet on the other we have to reason about that complicated state to understand our concurrent programs
- interference
- common patterns and paradigms for dealing with the ensuing complexity
  - concurrency mechanisms
  - programs
  - reasoning
- true parallelism for performance

What situations give rise to concurrent programs?

- 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 revisit parallel programming.

State Explosion

Consider the program

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

How many states does it have?

– discuss

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.

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
```

Now consider

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

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

– discuss

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?

– discuss

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!*)

What we will be 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 developing our ability to reason about concurrent programs. We will use our new-found ability to develop more convenient notations and mechanisms that confine the complexity issues to small sections of the program. We will discover how these mechanism relate to each other by showing how they can be used to implement one another. We will look at implementing the mechanisms within the instruction set of common processors. And we will look at using the mechanisms to build programs with interesting behavior with a focus on recurring patterns of use. We will look at multi-threaded programming in Java and with the *pthreads* library for C/C++, as well as the language MPD used in the textbook.

We will also look at concurrency from communications-oriented viewpoint to contrast with our initial look using a shared-variable approach. The MPD language provides support for this viewpoint in an imperative language. If there is time we will also look at its use in a functional language.

Reading and problems for next time:

- Read Chapter 1; you may skim sections 1.4 through 1.8. Section 1.9 introduces notation we will be using so you need to understand it well.

- Work through Prof. Sheldon's web tutorial on logic and sets listed in the syllabus.
- Read sections 2.1 and 2.6, skimming for now 2.6.3.
- Retrieve the errata list for the textbook from http://www.cs.arizona.edu/~greg/mpdbook/errata.html and mark all the corrections in your book.
- Write a *sequential* program to solve problem 2.7. Annotate the program with a proof outline that the program is correct, using the techniques of Section 2.6. The theorem to be proved is `{}your-program {`$x = \sum_{i=1}^{n} a_i$`}`. Hand in your work at the beginning of class next time and be prepared to discuss.