

CS421 Lecture 21: Concurrency¹

Mark Hills
mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

July 28, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Concurrency Overview

Language Support

Shared Memory and Message Passing

Synchronization

Concurrency: Terminology

- ▶ A program where two or more execution contexts may be active (running) at once is **concurrent**.
- ▶ A concurrent program where execution is actually occurring in multiple contexts at once is **parallel**.
- ▶ A concurrent program with execution contexts running on multiple nodes is **distributed**.
- ▶ An execution context in a program is a **thread**.

Unfortunately, this terminology (especially thread) varies with different languages and systems.

Options in Concurrency

In a language with concurrency support, we have many design options. These are the big three, but they lead to others...

- ▶ Library or Language: Is support for concurrency baked directly into the language, or is it provided by system libraries?
- ▶ Shared Memory or Message Passing: Do threads communicate by reading and writing shared areas of memory, or by passing messages back and forth?
- ▶ Synchronization: How do threads make sure that operations occur in the desired order?

Using Libraries

Libraries are still the most common way to support concurrency.

- ▶ For shared memory systems, *pthread*s is the most common option
- ▶ Vendors often provide their own thread packages as well (Microsoft has similar support for Windows, for instance)
- ▶ For message passing, PVM and MPI are both very popular, with (my opinion) MPI probably moreso now

Language Support

Languages with direct support for concurrency have provided many different constructs to represent concurrent operations.

- ▶ `co-begin`
- ▶ parallel loops
- ▶ launch-at-elaboration
- ▶ fork/join
- ▶ others as well...

Co-begin

A co-begin construct allows a standard block to be executed either sequentially or in parallel, based on the chosen keywords. From Algol 68:

```

1 # from Algol 68 #
2 par begin
3   p(a, b, c),
4   begin
5     d := q(e, f);
6     r(d, g, h)
7   end,
8   s(i, j)
9 end
  
```

The comma-separated statements execute in parallel, but the internal begin is sequentialized.

Parallel Loops

Parallel loops are available in some languages to allow each iteration to be executed in parallel. Some optimizing compilers also transform loops automatically into forms that can be executed in parallel.

```

1 # from the SR language
2 co (i := 5 to 10) ->
3   p(a, b, i)      # six instances of p
4 oc
  
```

Launch-at-Elaboration

Here, the thread code is declared in syntax similar to that for subroutines. The thread is created to execute the contained code when the declaration is *elaborated* at runtime.

```

1 procedure P is
2   task T is
3     ...
4   end T;
5 begin
6   ...
7 end P;
  
```

Task T needs information in procedure P, so P will wait for T to finish. Many instances of T can run at once – think recursion...

Fork/Join

Instead of threads being worked in to the control flow, they can also be explicitly defined and created. In Ada:

```

1 task type T is
2   ...
3 begin
4   ...
5 end T;
6 ...
7 pt : access T := new T;
  
```

Fork/Join in Java

```

1 class CClass extends Thread {
2   ...
3   CClass(...) {
4     // constructor
5   }
6   ...
7   public void run() {
8     // code that runs in the thread
9   }
10 }
11 ...
12 CClass myclass = new CClass(...);
13 myclass.start();
14 ...
15 myclass.join();
  
```

Shared Memory

Shared memory systems allow threads to communicate by updating shared components of program memory.

- ▶ **Advantage:** easy, familiar programming model
- ▶ **Disadvantage:** leads to race conditions and a need for explicit synchronization
- ▶ **Disadvantage:** not scalable

Message Passing

Message passing systems all threads to communicate by sending messages to one another. Messages generally come in two forms:

- ▶ **Synchronous Messages:** One thread sends a message to another and waits for the message to be received; a variant (RPC) actually waits for a reply as well
- ▶ **Asynchronous Messages:** One thread sends a message to another and then continues computation

Synchronous is probably more common, and maybe (but not always) makes more sense for threads communicating within a process or on the same machine. Asynchronous makes more sense for many (but not all) distributed computations.

Synchronous Message Passing

Many languages use a synchronous message passing model – Concurrent ML, plus by default with most RPC and RMI mechanisms.

- ▶ **Advantage:** allows direct synchronization on events between threads
- ▶ **Advantage:** provides a sense of “shared state”; the threads know something about where they each are in a computation
- ▶ **Disadvantage:** does not handle network problems, unreliable or congested threads well

RMI and RPC

An RPC is a **Remote Procedure Call**.

- ▶ RPCs provide for communication between multiple threads or processes
- ▶ Can communicate either in process or across machine boundaries
- ▶ Object version called RMI, for **Remote Method Invocation**

These are used either implicitly or explicitly in many cases – programming with Java-RMI, COM/COM+/.Net Remoting, Enterprise Java Beans, CORBA, etc uses these techniques.

Asynchronous Message Passing

Asynchronous message passing is used in some languages and language extensions, and is fundamental to the Actor model of computation.

- ▶ **Advantage:** handles variable network and congestion problems well
- ▶ **Disadvantage:** often requires some special infrastructure (message buffers, etc) to work correctly
- ▶ **Disadvantage:** message receipt can be more involved (think out of order receipt) and doesn't convey as much “information” as synchronous – need special data structures (vector clocks, etc) to add this information

Synchronization

Synchronization is mainly a concern in shared-memory applications. Message passing provides implicit synchronization on message sends and receives. Several common options are:

- ▶ locks
- ▶ barriers
- ▶ semaphores
- ▶ monitors

Locks

Locks provide the simplest unit of synchronization.

- ▶ One thread acquires a lock
- ▶ Other threads block when they try to acquire the same lock
- ▶ When the first thread leaves the lock, other thread(s) awaken and attempt to acquire it

Some languages and libraries have specific lock objects, while others (like Java) allow locks on items – Java allows locking on objects, for instance.

Barriers

Barriers are often used in numerical computations.

- ▶ Provide for lock-step computation
- ▶ All threads do some work, and then signal when they have reached the barrier
- ▶ When all threads have reached the barrier, all threads are released again to continue

Semaphores

Semaphores are similar to locks, but can be used in more complex ways.

- ▶ A binary semaphore is basically a lock
- ▶ Semaphores can count beyond 1, though – allowing use to synchronize when counts are important (bounded buffer problem, etc)
- ▶ **P** operation lowers count by one, waits until count is non-negative
- ▶ **V** operation raises count by one

Monitors

Monitors provide a more direct language solution to concurrency.

- ▶ Monitors provide data and operations on that data
- ▶ Only one thread can be active inside a monitor at once
- ▶ Modern version: Java synchronized methods
- ▶ Thread can wait to suspend itself
- ▶ Other threads then signal to wake up waiting threads
- ▶ Related concept: condition variables

Problems

- ▶ Race conditions – multiple threads could access and/or change values of a variable if it is not protected correctly
- ▶ Deadlock – multiple threads could each acquire resources (like locks) needed by each other, leading to a situation where none can make progress
- ▶ Starvation – unfair scheduling practices could lead a thread to wait forever, never making progress

Anything Better?

Experience has shown that using threads properly is hard. Is there a better way to do this?

Anything Better?

Experience has shown that using threads properly is hard. Is there a better way to do this?

- ▶ Transactional memory – use transaction processing techniques from databases to provide protection while not requiring explicit locking/unlocking code