Introducing Shared-Memory Concurrency

Race Conditions and Atomic Blocks

Laura Effinger-Dean

November 19, 2007

Concurrency

Computation where "multiple things happen at the same time" is inherently more complicated than *sequential* computation.

- Entirely new kinds of bugs and obligations

Two forms of concurrency:

- *Time-slicing*: only one computation at a time but *pre-empt* to provide *responsiveness* or *mask I/O latency*.
- *True parallelism*: more than one CPU (e.g., the lab machines have two, the attu machines have 4, ...)

Within a program, each computaton becomes a separate *thread*.

Why do this?

- Convenient structure of code
  - Example: web browser. Each "tab" becomes a separate thread.
  - Example: Fairness – one slow computation only takes some of the CPU time without your own complicated timer code. Avoids *starvation*.
- Performance
  - Run other threads while one is reading/writing to disk (or other slow thing that can happen in parallel)
  - Use more than one CPU at the same time
    - The way computers will get faster over the next 10 years
    - So no parallelism means no faster.

Working in Parallel

Often you have a bunch of threads running at once and they *might* need the same mutable memory at the same time but *probably not*. Want to be *correct* without sacrificing parallelism.
Example: A bunch of threads processing bank transactions:

- withdraw, deposit, transfer, currentBalance, ...
- chance of two threads accessing the same account at the same time very low, but not zero.
- want *mutual exclusion* (a way to keep each other out of the way when there is *contention*)

Basics

C: The POSIX Threads (pthreads) *library*

- `#include <pthread.h>`
- `pthread_create` takes a function pointer and an argument for it; runs it as a separate thread.
- Many types, functions, and macros for threads, locks, etc.

Java: Built into the language

- Subclass `java.lang.Thread` overriding `run`
- Create a Thread object and call its `start` method
- Any object can “be synchronized on” (later)

Common bug: race conditions

There are several new types of bugs that occur in concurrent programs; *race conditions* are the most fundamental and the most common.

- A race condition is when the order of thread execution in a program affects the program's output.
- Difficult to identify and fix, because problematic thread interleavings may be unlikely.
- *Data races* (common type of race condition) - when multiple threads access the same location in memory "simultaneously," with at least one access being a write

Example: TwoThreads.java

- What is the *intended* output of this program?
- What *actual* outputs are possible?

What could go wrong?

Simultaneous updates lead to race conditions. Suppose two threads both execute `i++`. In machine code, this single statement becomes several operations:

Thread 1	Thread 2
`r1 = i`	`r2 = i`
`r1 += 1`	`r2 += 1`
`i = r1`	`i = r2`

If `i` starts at 0, what is the value of `i` after execution?

What could go wrong?

Simultaneous updates lead to race conditions.

```
Thread 1        Thread 2
r1 = i
r1 += 1
i = r1
                r2 = i
                r2 += 1
                i = r2
```

Final value of `i` is 2.

What could go wrong?

Simultaneous updates lead to race conditions.

```
Thread 1          Thread 2
r1 = i
r1 += 1
                  r2 = i
                  r2 += 1
i = r1
                  i = r2
```

Final value of `i` is 1. The first update to `i` is lost.

Detecting race conditions

- Without calls to `Thread.sleep()`, our code ran “so fast” that the race condition did not manifest.
- Forcing a thread to yield control is a good way to encourage “interesting” interleavings.
- BUT:
  - Calling `sleep` doesn't guarantee that the race condition will affect the output.
  - In general, programs are large and we don't know where to look for bugs or if bugs even exist.

Avoiding race conditions

- We will try to restrict the number of possible thread interleavings.
  - E.g., in TwoThreads, we got into trouble because the updates were interleaved.
- Simple limitation is to define *atomic blocks* in which a thread may assume that no other threads will execute.
  - Lots of variations on terminology: critical sections, synchronization, etc.

Fixing TwoThreads.java

(Demo.)

Fixing TwoThreads.java

Now the following troublesome interleaving is illegal!

Thread 1	Thread 2
`r1 = i`	
`r1 += 1`	
	`r2 = i`
	`r2 += 1`
`i = r1`	
	`i = r2`

Fixing TwoThreads.java

Instead, we get:

```
Thread 1                    Thread 2
atomic {
  r1 = i
  r1 += 1
  i = r1
}
                            atomic {
                              r2 = i
                              r2 += 1
                              i = r2
                            }
```

Atomic blocks

- Atomic blocks are a common way of fixing race conditions
- Allows us to think “single-threaded” even in a multi-threaded program
- How much code should be inside the block?
    - Atomic blocks that are “too long” could cause the program to slow down, as threads sleep waiting for other threads to finish executing atomically.
    - Atomic blocks that are “too short” might miss race conditions.

Familiar example: bank accounts

(Demo)

Producer and consumer threads

- One (or more) thread(s) produces values and leaves them in a buffer to be processed
- One (or more) thread(s) takes values from the buffer and consumes them
- Common application in operating systems, etc.

(Demo.)

Limitations of atomic blocks

In the producer-consumer example, we see:

- *Busy wait*: threads loop forever waiting for a state change.
- *Scheduling fairness*: no guarantee that the threads will get equal shares of processor time.

There are ways to fix both of these problems with atomic blocks, but we don't have good implementations for Java or C. Instead, we'll look briefly at what mechanisms Java and pthreads do provide (more on Wednesday).

Atomic blocks don't exist yet

- You can't (yet) use atomic blocks in true Java code (although we "faked it").
  - Active area of research—maybe it will be integrated into Java at some point.
- Instead programmers use *locks* (mutexes) or other mechanisms, usually to get the behavior of atomic blocks.
  - But misuse of locks will violate the "all-at-once" policy.
  - Or lead to other bugs we haven't seen yet.

Lock basics

A lock is *acquired* and *released* by a thread.

- At most one thread holds it at any moment.
- Acquiring it "blocks" until the holder releases it and the blocked thread acquires it.
  - Many threads might be waiting; one will "win."
  - The lock-implementor avoids race conditions on lock-acquire.
- So create an atomic block by surrounding with lock acquire and release.
- Problems:
  - Easy to mess up (e.g., use two different locks to protect the same location in memory).
  - *Deadlock*: threads might get stuck forever waiting for locks that will never be released.

Summary

- Concurrency introduces bugs that simply don't exist in sequential programming.
- Atomic blocks are a useful way of thinking about safety in concurrent programs.
- In real code, you'll use locks or other mechanisms, which eliminate race conditions if used properly but can lead to more bugs if misused.

Blatant plug(in)

- For the examples we used AtomEclipse, a plugin that I developed for Eclipse (an IDE for Java (and other languages)).
- AtomEclipse is designed for students like you to learn about atomic blocks more easily.
- Download and try out if you want to play around with the examples some more: `http://wasp.cs.washington.edu/atomeclipse` (linked from the 303 web page)
- Let me know what you think: `effinger@cs` or talk to me after class!