Comp 411
Principles of Programming Languages
Lecture 28
Confronting Concurrency

Corky Cartwright

November 28, 2012

A Concurrent Future

Generic processor chips now have 2-8 cores. Within a decade, that figure will be much larger, or the order of 16-64 cores. Most software applications will have to be significantly modified (drastically rewritten?) to achieve significant performance gains going forward.

Implications for computing research and technology

- Opportunity for new disruptive programming technologies that tame the perils of concurrency.
- Creating production software will become significantly more challenging. At a minimum, we will need much better analysis and testing tools.
- Software portability will be significantly more difficult to achieve. Thread scheduling and compiler optimization are very platform dependent. Among mainstream language platforms, only Java has addressed this issue.

A Glimpse of Concurrency

Operations on shared data (typically) break if atomicity is not enforced. The conventional approach to ensuring atomicity is to use locks (common variants: semaphores and monitors). Standard locks support two operations: lock() and unlock(). In the JVM, any object can serve as a lock; the lock() and unlock() operations are implemented by the monitorEnter and monitorExit byte codes. A lock that is already held by a thread can be re-entrantly acquired by the same thread (but the thread is obliged to a matching number of unlock operations before the lock is released). Java objects also support wait() and notify() methods; each such queue is called a "condition variable"). In Java each object/lock implicitly includes one condition variable.

Example: a shared counter incremented by multiple threads. In the absence of synchronization, what can go wrong?

See

http://concutest.org/download/sigcse2010-javaconcurrency.zip

Concurrency Complications

Twenty years ago, concurrent programming was generally simpler than it is today because nearly all multi-threaded computing platforms supported a simple memory model called “sequential consistency”. Operations on shared memory locations were interleaved in some order consistent with the sequential execution of each thread.

But weaker memory models can be implemented more efficiently and support far more code optimization by compilers. The Java Memory Model (JMM) is perhaps the first serious attempt to define a portable program level memory model that is weaker than sequential consistency.

In my experience, the JMM is difficult to use and facilitates programming errors because programmers tend to assume sequential consistency.

A Programmer's View of the JMM

The details of the JMM are rather arcane. If you are interested, you are encouraged to study the definitive technical reference by Pugh et al. The Java Memory Model.

It is available on the course web page under Other Readings

Alternatives

The sequential consistency model can be liberalized in some ways that are not detectable by the programmer. For example, if the next operations in two active threads access disjoint sets of variables, then their relative ordering does not matter (assuming that exceptions are asynchronous). This notion of "apparent sequential consistency" is the best compromise IMO. The new C++ memory model uses this approach.

A Glimpse of Concurrency

What is price of synchronization?

Significant overhead. Locking slows down fine-grained operations by as much as 10x in the absence of contention!

Does not scale. Contention increases dramatically with more threads.

Example of scaling failure: heap allocation in Java.

Efficient synchronization strategies avoid locking; CAS (compare-and-swap) and similar machine primitives are essential to building scalable approaches to synchronization.