COSC441 Concurrent Programming

Atomic Variables and Mutexes

Richard A. O'Keefe

August 1, 2017

Outline

- What does “atomic” mean?
- Atomic operations in C
- Atomic operations in Java
- How atomic variables work
- CAS and LL/SC
- What is a “mutex”?
- Mutexes in C.
- Mutexes in Java.
- Deadlock.
- Semaphores.

What does “atomic” mean?

An operation is *atomic* if and only if it is either completed with all its side effects visible in all threads, or not completed at all with no side effects. The key point is that an operation can temporarily break an invariant while updating a data structure and then restore it, while nothing else will ever see the object in a state that does not satisfy the invariant.

How memory used to work

In the days of ferrite core memory,

1. Acquire exclusive access to the memory bus.
2. Fetch the value (which destroys it).
3. Modify the value.
4. Store the value back (essential).
5. Release the memory bus.

Typical old-style instructions

- ADDM register, memory
  [memory] := [memory] + register.
- SWAP register, memory
  temp := [memory]
  [memory] := register
  register := temp

On a Load/Store architecture

Semiconductor memory doesn't destroy when reading.

- LD register, memory
- operate on register
- ST register, memory

These are *separate* instructions which can be interleaved with instructions in other threads.

On an x86 machine

- LOCK;*inst* executes *inst* with the LOCK# signal asserted. (Since P6 locks cache, relies on cache consistency to achieve global locking.)
- Can only be used with ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.
- That is add, subtract, and bitwise operations on 8-, 16-, 32-, or 64-bit integers, also swap and compare-and-swap.

In C11: Header and types

- Header is ⟨stdatomic.h⟩.
- Types are atomic_T where T is one of bool, char, schar, uchar, short, ushort, int, uint, long, ulong, llong, ullong, intmax, uintmax, intptr_t, uintptr_t, size_t, ptrdiff_t, char16_t, char_32_t, wchar_t, {,u}int_least,fast8,16,32,64_t.
- Operations are type-generic.

In C11: Operations

- *A* v = ATOMIC_VAR_INIT(value);
- **void** atomic_store(**volatile** *A* *target, *C* newval);
- *C* atomic_load(**volatile** *A* *source);
- *C* atomic_exchange(**volatile** *A* *target);
- bool atomic_compare_exchange_strong(**volatile** *A* *target, *C* oldval, *C* newval);
- *C* atomic_fetch_{add,sub,and,or,xor}(**volatile** *A* *target, *C* delta);

C11 *vs* x86

- essentially the same range of operations
- no **float** or **double** atomic types
- C11 actually has an extra parameter to specify what kind of memory barriers you get; stick to the simple ones as they are almost always what you want.
- stdatomic.h is supported in GCC 4.9 and later
- C11 threads are not in glibc or musl.
- clang on OSX has neither

Other atomic libraries for C

- Solaris: ⟨atomic.h⟩, offers compare-and-swap swap, add, inc, and dec for pointers and all integers and and, or for all integers.
- MacOSX: ⟨libkern/OSAtomic.h⟩, offers compare-and-swap, add, inc, dec, and, or, and xor for 32- and 64-bit integers, with and without barriers (use the Barrier version), and single bit test-and-set/clear.
- OpenBSD: ⟨sys/atomic.h⟩, offers compare-and-swap, swap on pointers, int, long, and add, sub, inc, dec on int, long.
- Linux: GCC intrinsic (see GCC manual) or C11.

Atomic operations in Java

- Package is `java.util.concurrent.atomic`
- AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray, AtomicReference$\langle V \rangle$, and AtomicReferenceArray$\langle E \rangle$.
- Each class has a private volatile field.
- Like C and x86 (surprise surprise), no atomic floats or doubles.
- Array classes like scalars but operations have an extra index parameter, because Java can't express "array of volatile int".
- No *built-in* bitwise or max/min update. You can use accumulate, or do it outside.

Java, core operations

- get(), current value, relies on volatile
- set(x), relies on volatile
- getAndSet(x), returns old value, is swap
- compareAndSet(old, new), returns boolean.
- accumulateAndGet(x, fn), call binary fn, return new
- getAndAccumulate(x, fn), call binary fn, return old

How does this work?

```
bool compare_and_swap(volatile T*var, T old, T new) {
    atomic {
        if (*var == old) {
            *var = new;
            return true;
        } else {
            return false;
        }
    }
}
```

CAS is a fundamental building block

```
volatile T var;
// to do var = fun(var) atomically.
// Note: fun must not have side effects.
T old, new;
do {
    old = var;
    new = fun(old);
} while (!compare_and_swap(&var, old, new);
```

Load Locked + Store Conditional

- Some machines don't have CAS.
- Notably ARM.
- They have LL and SC.
- CAS is vulnerable to the A-B-A problem.
- LL+SC is not.
- LL+SC has its own problems...

What they do

- LL register, address
  protected := address;
  register := [memory];
- ST register, address
  [address] := register;
  if address == protected, clear protected.
- SC register, address
  if address == protected, [address] := register
  otherwise indicate failure.

The A-B-A problem

- “has the old value” is not the same as “has not been changed”.
- Thread 1 loads variable, gets A.
- Thread 2 stores B into variable.
- Thread 3 stores A into variable.
- Thread 1 looks again, sees A.
- Thinks “no change” but it changed twice.
- Use CAS when this is OK.

Problems of LL+SC

- Spurious failure if interrupt, context switch, *etc.* between LL and SC.
- Processor may limit number of instructions between LL and SC.
- Processor has only one "protected" register so two LL+SC blocks at same time will hurt.
- But at least not fooled by A-B-A.

Atomic limitations

- Atomic variables/operations let you update *one* number.
- The MC680[234]0 had a double compare-and-swap instruction (see Wikipedia) but current machines don't.
- If we need to update more than one state variable, or we want to protect some resource other than a memory variable, we need something else.
- That something is a *mutex*.

What is a “mutex”?

- An instance of an abstract data type used for **mut**ual **ex**clusion.
- Five basic operations:
  - pthread_mutex_init (create)
  - pthread_mutex_lock (lock)
  - pthread_mutex_trylock (try to lock, return boolean)
  - pthread_mutex_unlock (lock)
  - pthread_mutex_destroy (destroy)

Typical use

```
pthread_mutex_lock(&mymutex);
for (i = 0; i < n; i++) {
    a[i] = b[i]+c[i];
}
pthread_mutex_unlock(&mymutex)
```

Four kinds of mutex in POSIX

- Plain: error to lock a mutex you already hold. (Deadlocks.)
- Checked: error to lock a mutex you already hold. (Returns error code.)
- Recursive: can lock mutex many times, must unlock it that many times.
- Robust: if thread crashes while holding it, goes into special state where next locker is told

Plain mutexes

- Contains (available?, queue of waiting threads)
- lock(&mutex) = atomic:
  if available then set available false otherwise join the queue of waiting threads.
- unlock(&mutex) = atomic:
  if available report error, otherwise if queue empty, set available true, otherwise remove and wake one waiting thread.
- lock(&m); lock(&m) deadlocks.

Recursive mutexes

- Contains (owner, counter, queue of waiting threads)
- lock(&mutex) = atomic:
  if counter = 0 set counter to 1 and owner to this thread, otherwise if owner is this thread, increment counter, otherwise join the queue of waiting threads.
- unlock(&mutex) = atomic:
  if owner is not this thread, report error, otherwise decrement the counter, and if it is now 0, clear the owner, and if the queue is not empty, remove and wake one waiting thread.
- lock(&m); lock(&m) just works.

Mutexes in Java

- Package `java.util.concurrent.locks`
- Lock interface
- ReentrantLock class.
- .lock(), .trylock(), and .unlock()
- .lockInterruptibly()

Mutexes *vs* **synchronized**

- Every object is a lock.
- **synchronized** combines lock and unlock.
- There is no try-synchronized.
- There is no synchronized-with-timeout.
- Locks allow Conditions (next week).
- **synchronized** is faster but more limited.

Deadlock

- at least two threads, at least two resources.
- Thread 1 claims A.
- Thread 2 claims B.
- Thread 1 claims B and waits.
- Thread 2 claims A and waits. . .
- Avoidance: place a total order on locks, acquire earlier before later.
- We'll revisit this.

Semaphores

- First general purpose synchronisation method.
- Invented by E. W. Dijkstra.
- Inspired by railway semaphores.
- (non-negative counter, queue of waiting threads)
- P(sem) = if counter positive, decrement it, otherwise join queue of waiting threads.
- V(sem) = if counter zero and threads waiting, remove and wake one thread, otherwise increment counter.

Semaphores 2

- See “The Little Book of Semaphores”.
- Let you keep track of discrete but indistinguishable “resources”.
- One thread can wait for an event and another thread can signal it. Mutexes do not allow this.

Next week

- Deadlock, livelock, and starvation.
- Monitors and conditions
- Some concurrent data structures