

Locks: Hardware Support

To go beyond Peterson's algorithm and to build a working lock, we will need some help from our old friend, the hardware. Over the years, a number of different hardware primitives have been added to the instruction sets of various computer architectures; while we won't study how these instructions are implemented (that, after all, is the topic of a computer architecture class), we will study how to use them in order to build a mutual exclusion primitive like a lock.

As we saw in previous notes, while it is possible to build crude locks with just loads and stores, modern systems need more in order to correctly build locks. And thus we arrive at the crux of the problem:

THE CRUX: HARDWARE LOCK SUPPORT

What hardware support is needed to build locks and provide mutual exclusion for critical sections? Given new hardware primitives, how can we use them to build locks that are efficient and meet our requirements?

20.1 Test-And-Set

Some systems provide a form of what is called **test-and-set** in a single atomically-executed instruction (one example is SPARC, which has a `ldstub` instruction [WG00]). You can think of this instruction as doing what this function does:

```
int TestAndSet(int *lock, int new) {
    int old = *lock;
    *lock = new;
    return old;
}
```

What the code does is as follows. It returns the old value pointed to by the `lock`, and simultaneously updates said value to `new`. The key, of course, is that this sequence of operations is performed atomically¹. We can use such a primitive to build a simple lock that works, as in Figure 20.1.

```
typedef struct __lock_t {
    int flag;
}

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held by a thread
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 20.1: A Simple Lock using Test-and-set

¹How does the hardware do this? Well, you'll have to take a hardware class to learn that. Sorry!

Let's make sure we understand why this works. Imagine first the case where a thread calls `lock()` and no other thread currently holds the lock; thus, `flag` should be 0. When the thread then calls `TestAndSet(flag, 1)`, the routine will return the old value of the `flag`, which is 0; thus, the calling thread, which is *testing* the value of `flag`, will not get caught spinning in the while loop and will acquire the lock. The thread will also atomically *set* the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it simply calls `unlock` to set the `flag` back to zero.

The second case we can imagine arises when one thread already has the lock held (i.e., `flag = 1`). In this case, this thread will call `lock()` and then call `TestAndSet(flag, 1)` as well. This time, however, `TestAndSet()` will return the old value at `flag`, which is 1 (because the lock is held), while simultaneously setting it to 1 again. While the lock is held by another thread, `TestAndSet()` will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released. When the `flag` is finally set to 0 by some other thread, this thread will call `TestAndSet()` again, which will finally return 0 while atomically setting the value to 1 and thus acquire the lock.

Thus, by making both the **test** (of the old value of the lock) and **set** (of new value) one atomic operation, we can ensure that only one thread acquires the lock, and thus we can build a successful mutual exclusion primitive.

20.2 Compare-And-Swap

Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction (as it is called on SPARC, for example), or **compare-and-exchange** (as it called on x86). The C pseudocode for this single instruction is found in Figure 20.2.

The basic idea is for compare-and-swap to test whether the value at the address specified by `ptr` is equal to `expected`; if so, update the memory location pointed to by `ptr` with the

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Figure 20.2: Compare-and-swap

new value. If not, do nothing. In either case, return the actual value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

With the compare-and-swap instruction, we can build a lock in a manner quite similar to that with test-and-set. For example, we could just replace the `lock()` routine above with the following:

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

The rest of the code is the same as the test-and-set example above. This code works quite similarly; it simply checks if the flag is 0 and if so, atomically swaps in a 1 thus acquiring the lock. Threads that try to acquire the lock while it is held will get stuck spinning until the lock is finally released.

If you want to see how to really make a C-callable x86-version of compare-and-swap, this code sequence might be useful (stolen from here [S05]):

```
char CompareAndSwap(int *ptr, int old, int new) {
    unsigned char ret;

    // Note that sete sets a 'byte' not the word
    __asm__ __volatile__ (
        " lock\n"
        " cmpxchgl %2,%1\n"
        " sete %0\n"
        : "=q" (ret), "=m" (*ptr)
        : "r" (new), "m" (*ptr), "a" (old)
        : "memory");
    return ret;
}
```

Finally, as you may have sensed, compare-and-swap is a more powerful instruction than test-and-set. We will make good use of this power in the future when we briefly delve into **wait-free synchronization**.

20.3 Load-Linked and Store-Conditional

Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture [H93], for example, the **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures. The C pseudocode for these instructions is as found in Figure 20.3. Alpha, PowerPC, and later versions of ARM provide similar instructions [W09].

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the load_linked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Figure 20.3: Load-linked and Store-conditional

The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intermittent store to the address has taken place. In the case of success, the store-conditional returns 1 and updates the value at `ptr` to `value`; if it fails, the value pointed to by `ptr` is *not* updated and 0 is returned.

As a challenge to yourself, try thinking about how to build a lock using load-linked and store-conditional. Then, when you are finished, look at the code below which provides one simple solution. Do it! The solution is in Figure 20.4.

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise try it again
        }
    }

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 20.4: Using LL/SC To Build A Lock

The `lock()` code is the only interesting piece. First, a thread spins waiting for the flag to be set to 0 (and thus indicate the lock is not held). Once so, the thread tries to acquire the lock via the store-conditional; if it succeeds, the thread has atomically changed the value of the flag to 1 and thus can proceed into the critical section.

Note how failure of the store-conditional might arise. One thread calls `lock()` and executes the load-linked, returning 0 as the lock is not held. Before it can attempt the store-conditional, it is interrupted and another thread enters the lock code, also executing the load-linked instruction, and also getting a 0 and continuing. At this point, two threads have each executed the load-linked and each are about to attempt the store-conditional. The key feature of these instructions is that only one of these threads will succeed in updating the flag to 1 and thus acquire the lock; the second thread to attempt the store-conditional will fail (because the other thread updated the value of flag between its load-linked and store-conditional) and thus have to try to acquire the lock again.

In class, David Capel² suggested a more concise form of the above, for those of you who enjoy short-circuiting boolean con-

²Now famous.

ditionals. See if you can figure out why it is equivalent:

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag) || StoreConditional(&lock->flag, 1))
        ; // spin
}
```

20.4 Fetch-And-Add

One final hardware primitive is the **fetch-and-add** instruction. In assembly, it might look like this:

```
fetch-and-add <address>, r0ld
```

Its behavior is very simple: it atomically increments whatever the value is pointed to by the address, and puts the old value (before the increment) into register `r0ld`. The C pseudocode looks like this:

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

In this example, we'll use `fetch-and-add` to build a more interesting **ticket lock**, as introduced by Mellor-Crummey and Scott [MS91]. The lock and unlock code looks something like what you see in Figure 20.5.

Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic `fetch-and-add` on the ticket value; that value is now considered this thread's "turn" (`myturn`). The globally shared `lock->turn` is then used to determine which thread's turn it is; when (`myturn == turn`) for a given thread, it is that thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section.

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = fetchandadd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
    fetchandadd(&lock->turn);
}
```

Figure 20.5: Ticket Locks

Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future (once those in front of it have passed through the critical section and released the lock). In our previous attempts, no such guarantee existed; a thread spinning on test-and-set (for example) could spin forever even as other threads acquire and release the lock.

20.5 Summary: So Much Spinning

Our simple hardware-based locks are simple (only a few lines of code) and they work (you could even prove that if you'd like to), which are two excellent properties of any system or code. However, in some cases, these solutions can be quite inefficient. Imagine you are running two threads on a single processor. Now imagine that one thread (thread 0) is in a critical

section and thus has a lock held, and unfortunately gets interrupted. The second thread (thread 1) now tries to acquire the lock, but finds that it is held. Thus, it begins to spin. And spin. Then it spins some more. And finally, a timer interrupt goes off, thread 0 is run again, which releases the lock, and finally (the next time it runs, say), thread 1 won't have to spin so much and will be able to acquire the lock. Thus, any time a thread gets caught spinning in a situation like this, it wastes an entire time slice doing nothing but checking a value that isn't going to change! The problem gets worse with N threads contending for a lock; $N - 1$ time slices may be wasted in a similar manner, simply spinning and waiting for a single thread to release the lock. And thus, our next problem:

THE CRUX: TOO MUCH SPINNING

How can we develop a lock that doesn't needlessly waste time spinning on the CPU?

Hardware support alone cannot solve the problem. We'll need OS support too, which is the topic of the next note.

References

[H93] "MIPS R4000 Microprocessor User's Manual".

Joe Heinrich, Prentice-Hall, June 1993

Available: http://cag.csail.mit.edu/raw/documents/R4400.Uman_book_Ed2.pdf

[MS91] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors"

John M. Mellor-Crummey and M. L. Scott

ACM TOCS, February 1991

[S05] "Guide to porting from Solaris to Linux on x86"

Ajay Sood, April 29, 2005

Available: <http://www.ibm.com/developerworks/linux/library/l-solar/>

[W09] "Load-Link, Store-Conditional"

Wikipedia entry on said topic, as of October 22, 2009

<http://en.wikipedia.org/wiki/Load-Link/Store-Conditional>

Can you believe I referenced wikipedia? Pretty shabby. But, I found the information there first, and it felt funny not to cite it. Further, they even listed the instructions for the different architectures: `ldl_l/stl_c` and `ldq_l/stq_c` (Alpha), `lwarx/stwax` (PowerPC), `ll/sc` (MIPS), and `ldrex/strex` (ARM version 6 and above).

[WG00] "The SPARC Architecture Manual: Version 9"

David L. Weaver and Tom Germond, September 2000

SPARC International, San Jose, California

Available: <http://www.sparc.org/standards/SPARCV9.pdf>

Also see: http://developers.sun.com/solaris/articles/atomic_sparc/ for some more details on Sparc atomic operations.