A Close Look at a Spinlock

The spinlock is the most basic mutual exclusion primitive provided by a multiprocessor operating system. Spinlocks need to protect against preemption on the current CPU (typically by disabling interrupts, but we'll ignore that aspect in this post) and also against attempts by other cores to concurrently access the critical section (by using atomic memory operations). As the name implies, attempts to acquire a locked spinlock simply spin: they burn CPU time. Thus, we don't want to hold spinlocks for long and we certainly don't want to be preempted while holding one.

In the old days, people came up with spinlock implementations that were based on standard memory operations: loads and stores. Eisenberg & McGuire and the Lamport Bakery are examples. Modern multicores with weak memory models (including x86 and x86-64) break naive implementations of these algorithms and, while they can be fixed by adding memory fences, the resulting code isn't as efficient as what can be accomplished using hardware-supported atomic memory operations that are more powerful than loads and stores, such as test-and-set, compare-and-swap, and load-linked / store-conditional.

A spinlock does not need to be complicated, we can write a decently portable user-mode one using GCC's atomic intrinsics:

```
struct spin_lock_t {
  int lock;
};
```

```
void spin_lock(struct spin_lock_t *s) {
  while (1) {
    int zero = 0;
    int one = 1;
    if (__atomic_compare_exchange(&s->lock, &zero,
                                  &one, 0,
                                  __ATOMIC_SEQ_CST,
                                  __ATOMIC_SEQ_CST))
      return;
  }
}

void spin_unlock(struct spin_lock_t *s) {
  int zero = 0;
  __atomic_store(&s->lock, &zero, __ATOMIC_SEQ_CST);
}
```

The idea here is that the lock field holds a zero when the lock is free and a one when it is held. To acquire the lock, we use a compare-and-swap operation to try to change a zero in the lock field into a one — the key is that the comparison and the swap execute atomically. How the architecture accomplishes this is out of scope for this post! The __ATOMIC_SEQ_CST specifies that we want the strongest possible synchronization behavior from these atomic operations. This, and other aspects of this lock, could be optimized, but this implementation works.

An issue with this spinlock is that it permits unfairness under contention: if multiple cores are trying to get into a critical section, it is the processor's memory subsystem that effectively chooses who gets in. On a random AMD machine that I have, it is easy to see some cores accessing the critical section several times more often than others when contention is heavy. To make access to the critical section more fair, we can create spinlocks that enforce FIFO access. Here's a simple way to do that:

```
struct ticket_lock_t {
  unsigned front;
  unsigned back;
};

void ticket_lock(struct ticket_lock_t *s) {
  unsigned ticket =
    __atomic_add_fetch(&s->back, 1, __ATOMIC_SEQ_CST) - 1;
  while (1) {
    unsigned front;
    __atomic_load(&s->front, &front, __ATOMIC_SEQ_CST);
    if (front == ticket)
```

```
      return;
  }
}

void ticket_unlock(struct ticket_lock_t *s) {
  __atomic_add_fetch(&s->front, 1, __ATOMIC_SEQ_CST);
}
```

Here the lock is free when front == back, and the number of threads waiting to enter the critical section is front – back – 1. The metaphor is the same as in the Lamport Bakery: threads receive increasing ticket numbers as they request access to the critical section, and they are granted access in order of increasing ticket number. Some experimentation on a largish multicore shows that this spinlock is highly fair, even when contention is high.

On most platforms, the Linux kernel currently uses a "queued spinlock" that has a pretty complicated implementation. It is spread across several files but the bulk of the logic is here. The rest of this post will ignore that code, however, and rather focus on Linux's spinlock for 32-bit ARM platforms, which has a lot going on that's fun to dissect. Here's the code for acquiring a lock; I'll refer to it by line number in the rest of this post. But first, here's the data structure for this spinlock, the code is a bit messy but basically it's just saying that we can look at the spinlock data either as a 32-bit int or else as a pair of 16-bit ints; these are going to work in the same way as front and back in the ticket spinlock above. Of course this spinlock will fail if anyone manages to create a machine containing enough 32-bit ARM cores that more than 65535 of them end up contending for the same spinlock. Seems safe enough.

The prefetchw() call at line 62 turns into a pldw instruction. I don't have any idea why it is advantageous to prefetch a value so close to where it is actually needed. [**UPDATE:** Luke Wren and Paul Khuong pointed out on Twitter that this is to reserve the cache line for writing right off the bat, instead of reserving it for reading and then a bit later reserving it for writing.]

The inline assembly block at lines 64-71 is for grabbing a ticket number; it is functionally equivalent to the __atomic_add_fetch() in the ticket lock above. GCC inline assembly is never super easy so let's dig in. Line 69 specifies the outputs of the inline assembly block: it is going to write values into three variables, lockval, newval, and tmp, and these are going to be referred to respectively as %0, %1,

and %2 in the assembly code. These are “virtual registers” that the compiler will map to physical registers as it sees fit. Line 70 specifies the inputs to the inline assembly block: the address of the lock struct will be virtual register %3 and the constant 1<<16 will be stored in virtual register %4. Finally, line 71 specifies the “clobbers” for this inline assembly block: machine state not mentioned anywhere else that is going to be overwritten by the assembly. “cc” tells the compiler that it may not assume that the condition code flags will have the same values after this block executes that they held on entry.

Next we have five ARM instructions to look at. Together, ldrex and strex are a load-linked / store-conditional pair. This is a quirky but powerful mechanism supporting optimistic concurrency control — “optimistic” because it doesn’t prevent interference, but rather allows code to detect interference and recover, usually by just trying again, which is what happens here. Ok, into the details. The ldrex from %3 accomplishes two things:

- the 32-bit contents of the lock struct (pointed to by %3) are loaded into virtual register %0, aka lockval
- the memory location containing the lock struct is tagged for exclusive use

Next, lockval is incremented by 1<<16 and the result is stored into newval (virtual register %1). It is easy to prove that this addition does not change the low 16 bits of this value, but rather increments the value stored in the high 16 bits by one. This gives us a new ticket value that will be used to determine when we get access to the critical section.

Next, the strex instruction either stores newval (%1) back into the lock struct, or doesn’t, depending on whether anyone has touched the lock struct since we marked it for exclusive use (in practice there are other conditions that can cause loss of exclusivity but they don’t matter for purposes of this spinlock). Additionally, the success or failure of the store is recorded in tmp (virtual register %2). We’re not used to seeing an error code for stores to RAM but that’s exactly how this works. Next, tmp is tested for equality against zero, and finally if tmp is non-zero we branch back to the ldrex, indicating that we lost exclusivity, the store failed, and we need to try this sequence of operations again.

So what have we seen so far? The five ARM assembly instructions implement a little spin loop that grabs a new ticket value, retrying until this can be done without interference. Since the race window is short, we can hope that the expected number of retries is very close to zero.

Lines 73-76 are comparatively easy: this is the actual spinlock where we wait for ourselves to be the customer who is currently being served. This happens when "next" and "owner" are the same value. In the expected (no-contention) case the loop body here never executes. Keep in mind that lockval holds the state of the spinlock struct before we incremented the next field, so a quiescent spinlock would have had those values being the same when we arrived.

wfe() is a macro for the wfe (wait for event) instruction, which puts our core into a low power state until someone tells us it's time to wake up — below we'll look at how the unlock code makes that happen. Once awakened, our core uses Linux's READ_ONCE() macro to load from the low half of the spinlock struct, which hopefully allows us to escape from the while loop. READ_ONCE() is shorthand for casting a pointer to "pointer to volatile" and then loading through that pointer — this lets the compiler know that it is not allowed to cache the value of "owner." If that happened, this would turn into an infinite loop.

Finally, the smp_mb() at line 78 creates a memory barrier instruction that stops memory accesses inside the critical section from being reordered with respect to memory accesses outside of the critical section, from the point of view of other cores in this system.

Releasing this spinlock has three parts. First, another memory barrier to stop unfriendly reorderings. Second, increment the "owner" part of the spinlock struct. This is going to require three instructions: a load, an add, and a store, and none of these instructions are special synchronization instructions. This is OK because at this point we hold the lock and nobody else is allowed to change the owner. The high half of the word (containing next) can be changed behind our back — but that doesn't matter because we're only touching owner using 16-bit load/store instructions. Finally, dsb_sev() turns into a data synchronization barrier (ensuring that everyone can see our update to the owner field) and a sev (signal event) instruction, which causes all cores sleeping on a wfe instruction to wake up and continue executing. And we're done!