# Lock-Free Bugs

Russ Cox
January 4, 2017

*research.swtch.com/lockfree*

[*I wrote this post in mid-2014 for debuggers.co, which seems to have gone at least partly defunct, so I am reproducing it here. That site collected answers from programmers to the prompt "What's the most interesting bug you've encountered?"*]

To me, the most interesting bugs are the ones that reveal fundamental, subtle misunderstandings about the way a program works. A good bug is like a good science experiment: through it, you learn something unexpected about the virtual world you are exploring.

About ten years ago I was working on a networked server that used threads, coordinating with locks and condition variables. This server was part of Plan 9 and was written in C. Occasionally it would crash inside `malloc`, which usually means some kind of memory corruption due to a write-after-free error. One day, while benchmarking with the bulk of the server disabled, I was lucky enough to have the crash happen reproducibly. The server being mostly disabled gave me a head start in isolating the bug, and the reproducibility made it possible to cut code out, piece by piece, until one section was very clearly implicated.

The code in question was cleaning up after a client that had recently disconnected. In the server, there is a per-client data structure shared by two threads: the thread R reads from the client connection, and the thread W writes to it. R notices the disconnect as an EOF from a read, notifies W, waits for an acknowledgement from W, and then frees the per-client structure.

To acknowledge the disconnect, W ran code like:

```
qlock(&conn->lk);
conn->writer_done = 1;
qsignal(&conn->writer_ack);
qunlock(&conn->lk);
thread_exit();
```

And to wait for the acknowledgement, R ran code like:

```
qlock(&conn->lk);
while(!conn->writer_done)
    qwait(&conn->writer_ack);

// The writer is done, and so are we:
// free the connection.
free(conn);
```

This is a standard locks and condition variables piece of code: `qwait` is defined to release the lock (here, `conn->lk`), wait, and then reacquire the lock before returning. Once R observes that `writer_done` is set, R knows that W is gone, so R can `free` the per-connection data structure.

R does not call `qunlock(&conn->lk)`. My reasoning was that calling `qunlock` before `free` sends mixed messages: `qunlock` suggests coordination with another thread using `conn`, but `free` is only safe if no other thread is using `conn`. W was the other thread, and W is gone. But somehow, when I added `qunlock(&conn->lk)` before `free(conn)`, the crashes stopped. Why?

To answer that, we have to look at how locks are implemented.

Conceptually, the core of a lock is a variable with two markings *unlocked* and *locked*. To acquire a lock, a thread checks that the core is marked *unlocked* and, if so, marks it *locked*, in one atomic operation. Because the operation is atomic, if two (or more) threads are attempting to acquire the lock, only one can succeed. That thread—let's call it thread A—now holds the lock. Another thread vying for the lock—thread B—sees the core is marked *locked* and must now decide what to do.

The first, simplest approach, is to try again, and again, and again. Eventually thread A will release the lock (by marking the core *unlocked*), at which point thread B's atomic operation will succeed. This approach is called spinning, and a lock using this approach is called a spin lock.

A simple spin lock implementation looks like:

```
struct SpinLock
{
    int bit;
};

void
spinlock(SpinLock *lk)
{
    for(;;) {
        if(atomic_cmp_and_set(&lk->bit, 0, 1))
            return;
    }
}

void
spinunlock(SpinLock *lk)
{
    atomic_set(&lk->bit, 0);
}
```

The spin lock's core is the `bit` field. It is 0 or 1 to indicate unlocked or locked. The `atomic_cmp_and_set` and `atomic_set` use special machine instructions to manipulate `lk->bit` atomically.

Spinning only makes sense if the lock is never held for very long, so that B's spin loop only executes a small number of times. If the lock can be held for longer periods of time, spinning while it is held wastes CPU and can interact badly with the operating system scheduler.

The second, more general approach is to maintain a queue of threads interested in acquiring the lock. In this approach, when thread B finds the lock already held, it adds itself to the queue and uses an operating system primitive to go to sleep. When thread A eventually releases the lock, it checks the queue, finds B, and uses an operating system primitive to wake B. This approach is called queueing, and a lock using this approach is called a queue lock. Queueing is more efficient than spinning when the lock may be held for a long time.

The queue lock's queue needs its own lock, almost always a spin lock. In the library I was using, `qlock` and `qunlock` were implemented as:

```
struct QLock
{
    SpinLock spin;
    Thread *owner;
    ThreadQueue queue;
};

void
```

```
qlock(QLock *lk)
{
    spinlock(&lk->spin);
    if(lk->owner == nil) {
        lk->owner = current_thread();
        spinunlock(&lk->spin);
        return;
    }
    push(&lk->queue, current_thread());
    spinunlock(&lk->spin);
    os_sleep();
}

void
qunlock(QLock *lk)
{
    Thread *t;

    spinlock(&lk->spin);
    t = pop(&lk->queue);
    lk->owner = t;
    if(t != nil)
        os_wakeup(t);
    spinunlock(&lk->spin);
}
```

The queue lock's core is the `owner` field. If `owner` is `nil`, the lock is unlocked; otherwise `owner` records the thread that holds the lock. The operations on `lk->owner` are made atomic by holding the spin lock `lk->spin`.

Back to the bug.

The locks in the crashing code were queue locks. The acknowledgement protocol between R and W sets up a race between W's call to `qunlock` and R's call to `qlock` (either the explicit call in the code or the implicit call inside `qwait`). Which call happens first?

If W's `qunlock` happens first, then R's `qlock` finds the lock unlocked, locks it, and everything proceeds uneventfully.

If R's `qlock` happens first, it finds the lock held by W, so it adds R to the queue and puts R to sleep. Then W's `qunlock` executes. It sets the owner to R, wakes up R, and unlocks the spin lock. By the time W unlocks the spin lock, R may have already started running, and R may have already called `free(conn)`. The `spinunlock`'s `atomic_set` writes a zero to `conn->lk.spin.bit`. That's the write-after-free, and if the memory allocator didn't want a zero there, the zero may cause a crash (or a memory leak, or some other behavior).

But is the server code wrong or is `qunlock` wrong?

The fundamental misunderstanding here is in the definition of the queue lock API. Is a queue lock required to be unlocked before being freed? Or is a queue lock required to support being freed while locked? I had written the queue lock routines as part of a cross-platform library mimicking Plan 9's, and this question had not occurred to me when I was writing `qunlock`.

One one hand, if the queue lock must be freed only when unlocked, then `qunlock`'s implementation is correct and the server must change. If R calls `qunlock` before `free`, then R's `qunlock`'s `spinlock` must wait for W's `qunlock`'s `spinunlock`, so that W will really be gone by the time R calls `free`.

On the other hand, if the queue lock can be freed while locked, then the server is correct and `qunlock` must change: the `os_wakeup` gives up control of `lk` and must be delayed until after the `spinunlock`.

The Plan 9 documentation for queue locks does not address the question directly, but the implementation was such that freeing locked queue locks was harmless, and since I was using my library to run unmodified Plan 9 software, I changed the lock implementation to call `os_wakeup` after `spinunlock`. Two years later, while fixing a different bug, I defensively changed the server implementation to call `qunlock` too, just in case. The definition of the POSIX pthread_mutex_destroy function gives a different answer to the same design question: "It is safe to destroy an initialised mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behaviour."

What did we learn?

The rationale I gave for not calling `qunlock` before `free` made an implicit assumption that the two were independent. After looking inside an implementation, we can see why the two are intertwined and why an API might specify, as POSIX does, that you must unlock a lock before destroying it. This is an example of implementation concerns influencing an API, creating a "leaky abstraction."

What makes this bug interesting is that it was caused by a complex interaction between manual memory management and concurrency. Obviously a program must stop using a resource before freeing it. But a concurrent program must stop all threads from using a resource before freeing it. On a good day, that can require bookkeeping or careful coordination to track which threads are still using the resource. On a bad day, that can require reading the lock implementation to understand the exact order of operations carried out in the different threads.

In the modern computing world of clients and servers and clouds, concurrency is a fundamental concern for most programs. In that world, choosing garbage collection instead of manual memory management eliminates a source of leaky abstractions and makes programs simpler and easier to reason about.

I started the post by saying that good bugs help you learn something unexpected about the virtual world you are exploring. This was especially true for Maurice Wilkes and his team, who built EDSAC, the first practical stored-program computer. The first program they ran on EDSAC (printing square numbers) ran correctly, but the second did not: the log for May 7, 1949 reads "Table of primes attempted - programme incorrect." That was a Saturday, making this the first weekend spent working on a buggy program.

What did they learn? Wilkes later recalled,

> "By June 1949, people had begun to realize that it was not so easy to get a program right as had at one time appeared. ... It was on one of my journeys between the EDSAC room and the punching equipment that the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs." (Wilkes, p. 145)

For more about this early history, see Brian Hayes's "The Discovery of Debugging" and Martin Campbell-Kelly's "The Airy Tape: An Early Chapter in the History of Debugging."