Mutex from Scratch

Concurrency primitives in C++

Arthur O’Dwyer
2017-10-18

Outline

- What is the C++ concurrency model?
- Let’s implement `std::atomic<T>` in the compiler.
- What is a mutex? how does it differ from a spinlock?
- Let's implement `std::mutex` using Linux futexes.
- Let's implement `std::condition_variable` using Linux futexes.
- What other things in the STL are “lockable”?
- Let’s implement `std::once_flag`.

The C++ concurrency model

- When we “write to memory,” what does that actually mean?
- When we write to memory, *who else* sees what we write?
- When do we see what *someone else* has written?
- Who are these “someone elses,” anyway?

The C++ concurrency model

- When we “write to memory,” what does that actually mean?
- When we write to memory, *who else* sees what we write?
- When do we see what *someone else* has written?
- Who are these “someone elses,” anyway?

C++03 had no concept of “someone else.” There was only the current program, which was single-threaded by definition.

- Memory is just kind of a “scratchpad” for the current thread. Writes may or may not propagate to other threads.
- There’s the `volatile` keyword, but compilers don’t really respect it.

The C++ concurrency model

In C++11, the “who else” is some other *thread*. C++11 still doesn’t really say what a “thread” is, but at least we have a name for it.

- Each thread can use uncontended memory as its own “scratchpad.”
- If two threads are both reading/writing the same memory address “at the same time,” bad things happen (undefined behavior).
- To force two operations to happen “at different times,” you use something like `std::atomic`. The C++11 memory model abstractly guarantees that if two threads are reading/writing a `std::atomic`, then one of the reads/writes definitely **“happens-before”** the other one.

Useful atomic operations

```
std::atomic<int> x(0);

x++;                                          // lock incl (%rsi)

int former = x.exchange(43);                  // xchgl %eax, (%rsi)
// former now holds the previous value of x

int expected = 43;                            // lock cmpxchgl %edx, (%rsi)
bool success = x.compare_exchange_weak(expected, 44);
// expected now holds the previous value of x
```

Compare-exchange (compare-and-swap)

```
bool atomic<T>::compare_exchange_weak(T& expected, T desired)
{
    if (this->value_ == expected && !spurious_failure()) {
        this->value_ = std::move(desired);
        return true;
    } else {
        expected = this->value_;
        return false;
    }
}
```

Compare-exchange (compare-and-swap)

```
bool atomic<T>::compare_exchange_strong(T& expected, T desired)
{
    if (this->value_ == expected && !spurious_failure()) {
        this->value_ = std::move(desired);
        return true;
    } else {
        expected = this->value_;
        return false;
    }
}
```

On x86 and x86-64, `cmpxchgl` does this whole thing atomically with no spurious failure.

Compare-exchange (compare-and-swap)

```
bool atomic<T>::compare_exchange_strong(T& expected, T desired)
{
    T unchanged = expected;
    do {
        if (this->compare_exchange_weak(expected, desired))
            return true;
    } while (expected == unchanged);
    return false;
}
```

On PowerPC, ARM, and MIPS, CAS-weak and CAS-strong are two different instruction sequences.

Useful atomic operations

```
std::atomic<int> x(0);

x++;                                        // lock incl (%rsi)

int y = x++;          // movl $1, %eax; lock xaddl %eax, (%rsi)

int z = ++x;          // ???
```

Useful atomic operations

```
std::atomic<int> x(0);

x++;                                    // lock incl (%rsi)

int y = x++;          // movl $1, %eax; lock xaddl %eax, (%rsi)

int z = ++x;          // movl $1, %eax; lock xaddl %eax, (%rsi);
                      // incl %eax
```

Short isn’t always sweet

What does the compiler generate for an atomic operation that has no direct equivalent in your processor’s instruction set?

```
std::atomic<int> x(0);

int y = (x &= 1);   // There’s no x86 instruction for this one!
```

Short isn’t always sweet

```
std::atomic<int> x(0);


int y = (x &= 1);   // There’s no x86 instruction for this one!


// The above one-liner compiles into the equivalent of this:
int old_x = x.load();
int new_x = old_x & 1;
while (x.compare_exchange_weak(old_x, new_x)) {
    new_x = old_x & 1;
}
```

Synchronization primitives

When “(A **happens-before** B) xor (B **happens-before** A)”, I’ll say that “A **synchronizes with** B.”

Synchronization primitives are simple software mechanisms provided by a platform to its users for the purpose of making it easy to express *synchronizes-with* relationships.

- mutex
- condition variable
- semaphore*
- monitor*
- barrier*

* not available in the STL

Mutex from scratch

A **mutex** (short for *mutual exclusion*) has one job — to make sure that only one thread at a time enters a given section of code.

```
// Thread A

int SafeStack::push(int value)
{
    m_mtx.lock();
    m_stack.push(value);
    m_mtx.unlock();
}
```

```
// Thread B

int SafeStack::pop()
{
    m_mtx.lock();
    int value = m_stack.top();
    m_stack.pop();
    m_mtx.unlock();
    return value;
}
```

Why are these sections “critical”?
Because if threads A and B are reading/writing the memory of `m_stack` “at the same time,” bad stuff (undefined behavior) will happen. We must ensure it doesn’t happen.

Mutex from scratch

A mutex is like a coffee-shop bathroom with one key.

- If Alice has the key, Barbara can't get into the bathroom. What does Barbara do instead? She must *wait*.
- If, while Barbara is waiting, Carol also shows up, then Carol should also wait.
- When Alice comes out and returns the key, one of Barbara and Carol should stop waiting, take the key, and go in. (Not necessarily Barbara, although that would be *fair*.)

Not a mutex (spinlock)

```
class spinlock {
    std::atomic<bool> m_held;

public:
    constexpr spinlock() noexcept : m_held(false) {}

    void lock() {
        while (m_held.exchange(true)) {}
    }
    void unlock() {
        m_held = false;
    }
};
```

This is like Barbara rattling the doorknob constantly.

Not only is it inelegant, it's actually counterproductive — Alice will probably take longer to finish, what with all this rattling going on!

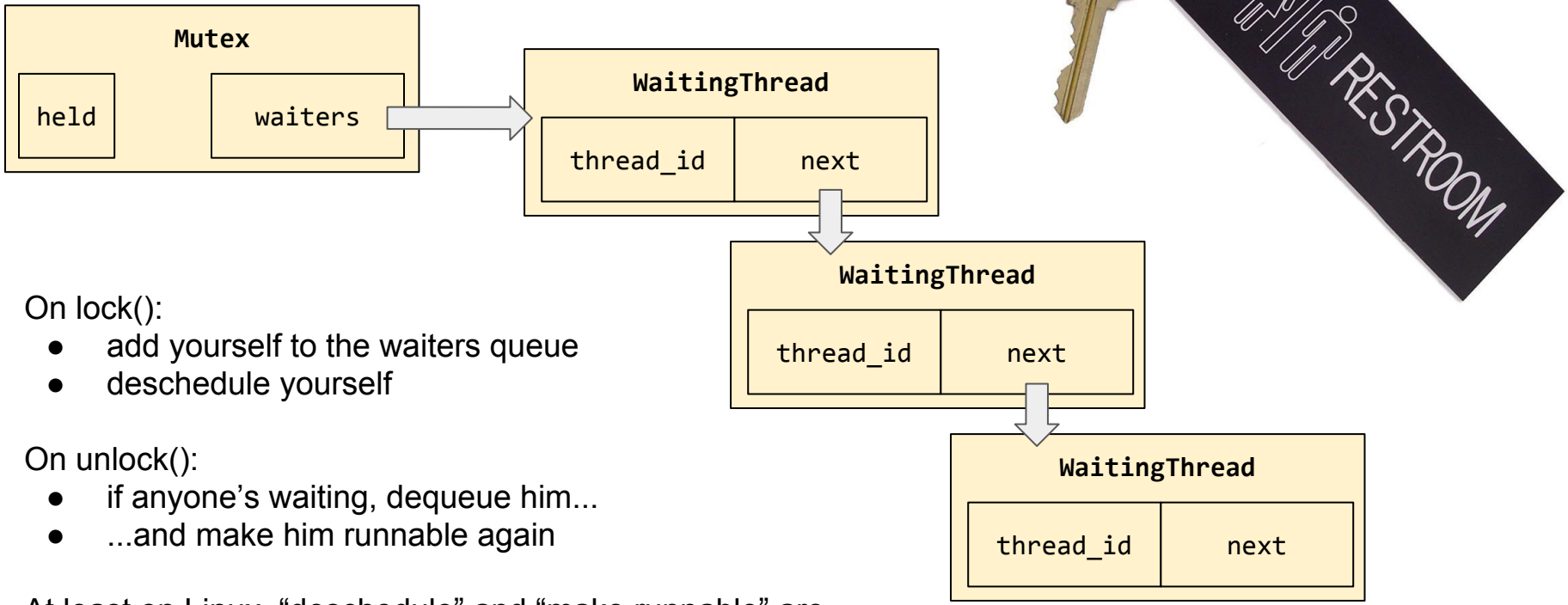
We need to teach Barbara how to *wait patiently*.

Mutex from scratch

A mutex may be **locked** or **unlocked**.

- Suppose thread B is trying to lock the mutex. If the mutex is already locked, then thread B must *wait*.
- In this context, *wait* means *sleep*, which means *tell the operating system to deschedule your thread until... later*.
- Therefore, in order to implement a mutex, we need to know something about the definition of *thread* and how to tell the OS to *deschedule* a thread “until later.”

Mutex from scratch



On lock():

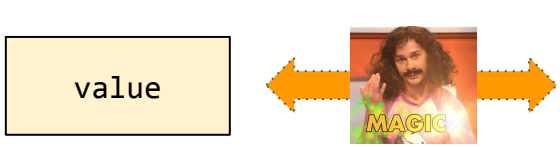
- add yourself to the waiters queue
- deschedule yourself

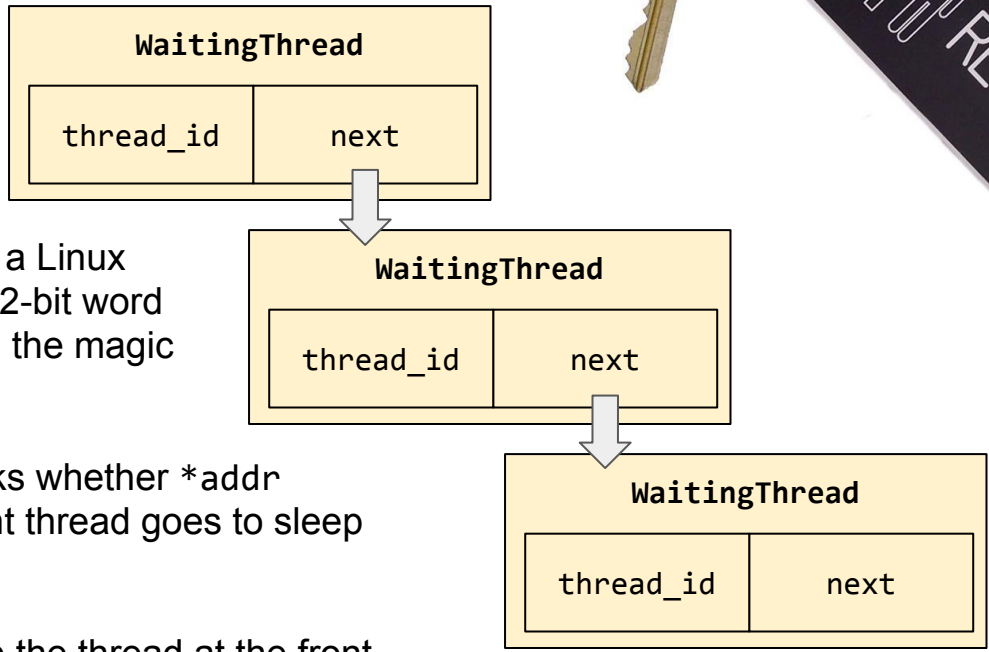
On unlock():

- if anyone's waiting, dequeue him...
- ...and make him runnable again

At least on Linux, "deschedule" and "make runnable" are things that happen inside the kernel, not in userland. The kernel exposes some nice primitives for our use.

Futexes in 30 seconds





A “futex” (fast userspace mutex) is a Linux concept. Actually a futex is just a 32-bit word anywhere in memory — what does the magic is a Linux system call.

`futex_wait(addr, value)` checks whether `*addr` is equal to `value`; if it is, the current thread goes to sleep in the queue associated with `addr`.

`futex_wake_one(addr)` wakes up the thread at the front of the queue associated with `addr`.

The dumbest mutex

```
class mutex {
    std::atomic<int> m_state;
    static constexpr int UNLOCKED = 0;
    static constexpr int LOCKED = 1;
public:
    constexpr mutex() noexcept : m_state(UNLOCKED) {}

    void lock() {
        while (m_state.exchange(LOCKED) != UNLOCKED) {
            futex_wait(&m_state, LOCKED);
        }
    }
    void unlock() {
        m_state = UNLOCKED;
        futex_wake_one(&m_state);
    }
};
```

Barbara (the waiter) is actually doing the right thing here.

The problem with this mutex is that *Alice* is doing something silly!

When Alice gets out, she goes looking for someone who wants the key (`futex_wake_one`).

Most of the time, she won't find anyone waiting.

Can we communicate the idea of "nobody is waiting" to Alice, so that she can skip that expensive call?

Ulrich Drepper's mutex

```
class mutex {
    std::atomic<int> m_state = 0;
    static constexpr int UNLOCKED = 0, LOCKED_WITHOUT_WAITERS = 1, LOCKED_WITH_WAITERS = 2;
public:
    void lock() {
        int x = UNLOCKED;
        if (m_state.compare_exchange_weak(x, LOCKED_WITHOUT_WAITERS)) {
            // the uncontended case
        } else {
            while (m_state.exchange(LOCKED_WITH_WAITERS) != UNLOCKED) {
                futex_wait(&m_state, LOCKED_WITH_WAITERS);
            }
        }
    }
    void unlock() {
        if (--m_state == UNLOCKED) {
            // the uncontended case
        } else {
            m_state = UNLOCKED;
            futex_wake_one(&m_state);
        }
    }
};
```

"Futexes Are Tricky"
Ulrich Drepper, 2003 (revised 2011)
https://www.akkadia.org/drepper/futex.pdf

Condition variable from scratch

A condition variable is **kind of** like a mutex, in that it has two operations, only one of which is blocking.

Mutex:

- Many threads can queue up on `lock`.
- Calling `unlock` unblocks exactly one waiter, who becomes the new “owner”.
- You’re not allowed to call `unlock` unless you “own” the mutex.
- If nobody “owns” the mutex then `lock` doesn’t block.

Condition variable:

- Many threads can queue up on `wait`.
- Calling `notify_one` unblocks exactly one waiter.
- Anybody can call `notify_one`.
- `wait` always blocks.

Condition variable

A condition variable is useful if you're one of the guys who refills the paper towels in the coffee-shop bathroom.

- You want to do a task (refill the towels) if and only if a condition is satisfied (the towel dispenser is empty). If the condition is not satisfied, you must *wait*.
- Also, if Alice is using the towel dispenser right now, you must wait for her to be done. When Alice comes out, if you're out of towels, she'll let you know.
- Still, there's the possibility of miscommunication; you should check for yourself. (You also need to check when you arrive at work in the morning.)

Notice that we *assume* mutual exclusion — that you shouldn't barge in unless you have the key.

We also assume cooperation — you will sleep, and Alice will not shirk her duty to wake you if she uses the last towel.

condition_variable

```
class condition_variable {
    std::atomic<int> m_value = 0;
public:
    void wait(unique_lock<mutex>& lk) {
        int sequence_number = m_value.load();
        if (sequence_number & 1) {
            if (!m_value.compare_exchange_strong(sequence_number, sequence_number + 1)) {
                if (sequence_number & 1) {
                    return;
                }
            }
        }
        lk.unlock();
        futex_wait(&m_value, sequence_number);
        lk.lock();
    }

    void notify_one() noexcept {
        m_value |= 1;
        futex_wake_one(&m_value);
    }
};
```

“Condition variable with futex”
Rémi Denis-Courmont, 2016
https://www.remlab.net/op/futex-misc.shtml

condition_variable_any

```
class condition_variable_any {
    std::atomic<int> m_value = 0;
public:
    template<class Lockable> void wait(Lockable& lk) {
        int sequence_number = m_value.load();
        if (sequence_number & 1) {
            if (!m_value.compare_exchange_strong(sequence_number, sequence_number + 1)) {
                if (sequence_number & 1) {
                    return;
                }
            }
        }
        lk.unlock();
        futex_wait(&m_value, sequence_number);
        lk.lock();
    }

    void notify_one() noexcept {
        m_value |= 1;
        futex_wake_one(&m_value);
    }
};
```

“Condition variable with futex”
Rémi Denis-Courmont, 2016
https://www.remlab.net/op/futex-misc.shtml

Lockable things in the STL

- `unique_lock<mutex>` — an exclusive lock (`pthread_mutex_t`)
- `unique_lock<shared_mutex>` — a write-lock (`pthread_rwlock_t`)
- `shared_lock<shared_mutex>` — a read-lock (`pthread_rwlock_t`)
  We haven't covered reader-writer locks, and we aren't going to.
- `unique_lock<recursive_mutex>` — an exclusive lock that can be "stacked"
  We won't cover this either.

All of these mutexes share a common notion of what it means to "sleep," so they all play nicely with `std::condition_variable_any`.

The notion of what it means to "sleep" is also shared by `std::thread`, by the free function `std::this_thread::sleep_for()`, ***and by thread-safe static initialization.***

once_flag from scratch

```
static std::once_flag flag;
std::call_once(flag, [&]() {
    // Do some complicated initialization.
});
```

Equivalent to this:

```
static int dummy = [&]() {
    // Do some complicated initialization.
    return 0;
}();
```

Calling the lambda; dummy is initialized with the lambda's result

once_flag from scratch

```
class LazyWidget {
    mutable std::once_flag mFlag;  // protects mCachedValue
    mutable int mCachedValue;
    std::vector<int> mInputs;
public:
    LazyWidget(std::vector<int> v) : mInputs(std::move(v)) {}

    int get() const {
        std::call_once(this->mFlag, [this]() {
            this->mCachedValue = expensiveComputation(this->mInputs);
        });
        return this->mCachedValue;
    }
};
```

`once_flag` from scratch

`mutex`:

- Many threads can queue up on `lock`.
- Calling `unlock` unblocks exactly one waiter: the new “owner”.
- `lock` blocks only if somebody “owns” the mutex.

`condition_variable`:

- Many threads can queue up on `wait`.
- Calling `notify_one` unblocks exactly one waiter.
- Calling `notify_all` unblocks all waiters.
- `wait` always blocks.

`once_flag`:

- Many threads can queue up on `call_once`.
- Failing at the callback unblocks exactly one waiter: the new “owner”.
- Succeeding at the callback unblocks all waiters and sets the “done” flag.
- `call_once` blocks only if the “done” flag hasn’t been set.

once_flag from scratch (mtx+cvar)

```
class once_flag {
    static constexpr int UNDONE = 0, IN_PROGRESS = 1, DONE = 2;
    int m_flag {UNDONE}; mutex m_mtx; condition_variable m_cv;
public:
    template<class F, class... Args>
    void call_once(F&& f, Args&&... args) {
        unique_lock<mutex> lk(m_mtx);
        while (m_flag == IN_PROGRESS)
            m_cv.wait(lk);
        if (m_flag == DONE) return;
        m_flag = IN_PROGRESS;
        lk.unlock();
        try {
            std::forward<F>(f)(std::forward<Args>(args)...);
        } catch (...) {
            lk.lock(); m_flag = UNDONE; m_cv.notify_one(); throw;
        }
        lk.lock(); m_flag = DONE; m_cv.notify_all();
    }
};
```

once_flag from scratch (futex)

```
class once_flag {
    static constexpr int UNDONE = 0, IN_PROGRESS = 1, DONE = 2;
    std::atomic<int> m_futex {UNDONE};
public:
    template<class F, class... Args>
    void call_once(F&& f, Args&&... args) {
        int x = UNDONE;
        while (!m_futex.compare_exchange_weak(x, IN_PROGRESS)) {
            if (x == DONE) return;
            futex_wait(&m_futex, IN_PROGRESS);
            x = UNDONE;
        }
        try {
            std::forward<F>(f)(std::forward<Args>(args)...);
        } catch (...) {
            m_futex = UNDONE; futex_wake_one(&m_futex); throw;
        }
        m_futex = DONE; futex_wake_all(&m_futex);
    }
};
```

...at_thread_exit()

```
namespace std {

    void notify_all_at_thread_exit(
        std::condition_variable&,
        std::unique_lock<std::mutex>
    );
} // namespace std
```

Given a lock and a condition_variable, take ownership of the lock from now until the current thread exits. *When the current thread exits,* after running any destructors associated with the thread's scope (including TLS), unlock the lock and notify the condition variable.
How on earth does the standard library implement this?

libc++ source code

```
typedef vector<pair<condition_variable*, mutex*>,
               __hidden_allocator<pair<condition_variable*, mutex*> > > _Notify;

__thread_struct_imp::~__thread_struct_imp()  // Think of this as std::thread::~thread().
{
   for (_Notify::iterator i = notify_.begin(), e = notify_.end();
           i != e; ++i)
   {
       i->second->unlock();
       i->first->notify_all();
   }
   // ...
}

void __thread_struct_imp::notify_all_at_thread_exit(condition_variable* cv, mutex* m)
{
   notify_.push_back(pair<condition_variable*, mutex*>(cv, m));
}
```

Questions?

Not all at once, please!