Lecture 8 — Lock Convoys, Atomics, Lock-Freedom

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

January 3, 2018

Lock Convoys

We'd like to avoid a situation called a lock convoy.

This happens when we have at least two threads that are contending for a lock of some sort.

And it's sort of like a lock traffic jam.

Lock Convoys

It occurs when 2+ threads at the same priority frequently (several times per quantum) acquire a synchronization object.

It’s bad even if they only hold that object for a very short amount of time.

We get into trouble if a thread’s time slice expires while it’s holding the lock.

Then we get all the threads jammed up...

Lock Convoys

Why is it called a convoy?

A convoy is when a grouping of vehicles, usually trucks or ships, travels all closely together.

This is also sometimes called the “boxcar” problem.

Ultimately far too much CPU time is spent handling context switches!

Weird Side Effects of Lock Convoys

Threads acquire the lock frequently and they are running for very short periods of time before blocking.

Other, unrelated threads of the same priority get to run for an unusually large percentage of the (wall-clock) time.

This can lead you to thinking that some other process is the real offender.

Unfairness is Fair?

In Windows Vista and later versions, the problem is solved because locks are unfair.

Windows XP: if a lock *l* is unlocked by *A* and there is a thread *B* waiting, then the lock is modified so that it looks like *B* owns it.

Then *B* is no longer blocked, and *B* already owns the lock when it wakes up.

The lock can never be “stolen”; hence “fair”.

Fair is Unfair

There is a period of time where the lock is held by *B*, but *B* is not running.

Would it be so bad to allow another thread *C* to sneak in there?

If it wants the lock why should it not get it and release it before *B* gets its turn?

Diagnosing Lock Convoys

Sign of a convoy: a lock that has some nonzero number of waiting threads but nobody appears to own it.

It just so happens that we're in the middle of a handover.

Some thread has signalled but the other thread has not yet woken up to run yet.

Starvation?

Changing the locks to be unfair does risk starvation.

Windows does give a thread priority boost, temporarily, after it gets unblocked, to see to it that the unblocked thread does actually get a chance to run.

Do Your Own Dirty Work

Although it can be nice to be able to give away such a problem to the OS developers, we might have to solve it for ourselves.

Options:

- Sleep
- Share
- Cache
- Trylock

Sleep, Data, Sleep

We could make the threads that are NOT in the lock convoy call a sleep() system call fairly regularly to give other threads a chance to run.

This solution is lame, though, because we're changing the threads that are not the offenders.

It just band-aids the situation so the convoy does not totally trash performance.

Still, we are doing a lot of thread switches, which themselves are expensive as outlined above.

Sharing is Caring

The next idea is sharing: can we use a reader-writer lock to allow much more concurrency than we would get if everything used exclusive locking?

If there will be a lot of writes then there's limited benefit to this speedup, but if reads are the majority of operations then it is worth doing.

We can also try to find a way to break a critical section up into two or more smaller ones, if that can be done correctly!

Nevermind, I wanted something else...

The next idea has to do with changing when (and how) you need the data.

Shrink the critical section to just pull a copy of the shared data and operate on the shared data.

But you saw the earlier discussion about critical section sizes, right? So you did that already...?

Try, Try Again

The last solution suggested is to use try-lock primitives:

```
int retries = 0;
while(pthread_mutex_trylock( &lock ) != 0 ) { /* 0 indicates lock acquired */
  if ( retries < SPIN_LIMIT ) {
    retries++;
    sleep(0);
    continue;
  }
  pthread_mutex_lock( &lock );
  break;
}
```

If we reach the limit then we just give up and enter the queue!

Polling?!

It looks like polling for the critical section.

The limit on the number of tries helps in case the critical section belongs to a low priority thread and we need the current thread to be blocked.

Under this scheme, if *A* is going to release the critical section, *B* does not immediately become the owner.

A may keep running and *A* might even get the critical section again before *B* tries again to acquire the lock (and may succeed).

Even if the spin limit is as low as 2, this means two threads can recover from contention without creating a convoy

You've been... THUNDERSTRUCK!

The lock convoy has some similarities with a different problem called the thundering herd problem.

In the thundering herd problem, some condition is fulfilled (e.g., broadcast on a condition variable) and it triggers a large number of threads to wake up.

It is likely they can't all proceed, so some will get blocked and then awoken again all at once in the future.

In this case it would be better to wake up one thread at a time instead of all of them.

Locks and Atomics

Atomics are a lower-overhead alternative to locks as long as you're doing suitable operations.

Remember that what we wanted sometimes with locks and mutexes and all that is that operations are indivisible.

Ex: an update to a variable doesn't get interfered with by another update.

Remember the key idea is: an atomic operation is indivisible.
Other threads see state before or after the operation; nothing in between.

About C++ atomics

You can use the default `std::memory_order`.
(= sequential consistency)

Don't use relaxed atomics unless you're an expert!

Different Memory Options

- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_consume
- memory_order_relaxed
- memory_order_seq_cst

Really, don’t use C++ relaxed atomics!

 [1]

[1] Photo Credit: Danielle Guerard

C++ atomics: Key Idea

An *atomic operation* is indivisible.

Other threads see state before or after the operation, nothing in between.

Simplest: `atomic_flag`

```
#include <atomic>

atomic_flag f = ATOMIC_FLAG_INIT;
```

Represents a boolean flag.

Operations on `atomic_flag`

Can clear, and can test-and-set:

```
#include <atomic>

atomic_flag f = ATOMIC_FLAG_INIT;
int foo() {
  f.clear();
  if (f.test_and_set()) {
    // was true
  }
}
```

`test_and_set`: atomically sets to true,
returns previous value.

No assignment (=) operator.

Although I guess in C++ you could define one if you wanted.

This is kind of a dangerous thing about C++.

If in C you see a line of code like `z = x + y;` you can have a pretty good idea about what it does and you can infer that there's some sort of natural meaning to the + operator there, like addition or concatenation.

In C++, however, this same line of code tells you nothing unless you know...
(1) the type of x,
(2) the type of y, and
(3) how the + operator is defined on those two operands *in that order*.

But I'm digressing.

Using more general C++ atomics

Declaring them:

```
#include <atomic>

atomic<int> x;
```

Libary's implementation:

- on small types, lock-free operations;
- on large types, mutexes.

What to do with Atomics

Kinds of operations:

- reads
- writes
- read-modify-write (RMW)

Reads and Writes

C++ has syntax to make these all transparent:

```
#include <atomic>
#include <iostream>

std::atomic<int> ai;
int i;

int main() {
    ai = 4;
    i = ai;
    ai = i;
    std::cout << i;
}
```

Can also use `i = ai.load()` and `ai.store(i)`.

Read-Modify-Write (RMW)

Consider `ai++`.

This is really

```
tmp = ai.read(); tmp++; ai.write(tmp);
```

Hardware can do that atomically.

Other RMWs: `+-`, `&=`, `etc`, `compare-and-swap`

more info:
`http://preshing.com/20130618/atomic-vs-non-atomic-operations/`

Lock Freedom

Suppose we'd like to operate in a world in which there are no locks.

Research has gone into the idea of lock-free data structures.

If you have a map and it will be shared between threads, the normal thing would be to protect access to the map with a mutex (lock).

But what if the data structure was written in such a way that we didn't have to do that?

That would be a lock-free data structure.

Use Locks Wisely!

For a great many situations, the normal locking and unlocking behaviour is sufficient.

We likely want to use it when we need to guarantee that progress is made.

Or: when we really can't use locks (e.g., signal handler), or where a thread dying while holding a lock results in the whole system hanging.

Non-Blocking Data Structure

A non-blocking data structure is one where none of the operations can result in being blocked.

In a language like Java there might be some concurrency-controlled data structures in which locking and unlocking is handled for you, but those can still be blocking.

Lock-free data structures are always inherently non-blocking.

A spin lock or busy-waiting approach is not lock free, because if the thread holding the lock is suspended then everyone else is stuck!

Lock-Free Data Structure

A lock-free data structure doesn't use any locks (duh) but there's also some implication that this is also thread-safe.

You can't make all your data structures lock-free ones by just deleting all the mutex code (sorry).

Lock free also doesn't mean it's a free-for-all; there can be restrictions.

For example, a queue that allows one thread to append to the end while another removes from the front, but not 2 removals at the same time.

Lock-Free Data Structure

The actual definition of lock-free is more formal.

If any thread performing an operation gets suspended during the operation, then other threads accessing the data structure are still able to complete their tasks.

This is distinct from the idea of waiting, though; an operation might still have to wait its turn or might get restarted.

Wait-Free Data Structures

You might need wait-free data structures.

This does not mean that nothing ever has to wait!

It does mean that each thread trying to perform some operation will complete it within a bounded number of steps regardless of what any other threads do.

This means that a compare-and-swap routine with infinite retries is not wait free, because a very unlucky thread could potentially take infinite tries...

The wait free data structures tend to be very complicated...

Example Lock-Free Algorithm

```
void stack_push(stack* s, node* n) {
    node* head;
    do
    {
        head = s->head;
        n->next = head;
    }
    while ( !atomic_compare_exchange(s->head, head, n) );
}
```

Example Wait-Free Algorithm

```
void increment_reference_counter(rc_base* obj) {
    atomic_increment(obj->rc);
}

void decrement_reference_counter(rc_base* obj) {
    if (0 == atomic_decrement(obj->rc))
        delete obj;
}
```

To Lock Free, or Not to Lock Free

Are lock-free programming techniques somehow better for performance? Maybe!

Lock free algorithms are about ensuring there is forward progress in the system and not really specifically about speed.

A particular algorithm implementation might be faster under lock-free algorithms.

But often they are not. In fact, the lock free algorithms could be slower, in which case you use them because you must, not because it is particularly speedy.