When Lock-Free Still Isn't Enough:

An Introduction to Wait-Free Programming and Concurrency Techniques

DANIEL ANDERSON

Cppcon
The C++ Conference
2024
September 15 - 20

Daniel Anderson
Assistant Teaching Professor @ CMU
danielanderson.net

Carnegie Mellon University

What we'll learn today

- Very quick review of concurrency and lock-free programming
- Review the "bread and butter" of lock-free design patterns
- Define ***wait-free algorithms***, understand the definition and practical implications
- An example of an elegant wait-free algorithm and wait-free design
- Some simple benchmarks

Some assumed knowledge

- You know what std::atomic does and what it is used for
- You've heard of lock-free programming and know what a *compare_exchange* is

Our motivating problem

- Sometimes referred to as a “sticky counter” (it gets stuck at zero)

```
struct Counter {
  // If the counter is greater than zero, add one and return true
  // otherwise do nothing and return false
  bool increment_if_not_zero();

  // Decrement the counter. If the counter now equals zero,
  // return true. Otherwise return false.
  // Precondition: The counter is not zero
  bool decrement();

  uint64_t read();  // Return the current value of the counter
};
```

- Required by std::weak_ptr<T>::lock
- Also, very useful for atomic memory management / concurrent data structures

Carnegie Mellon University

Our first implementation

```
struct Counter {

  bool increment_if_not_zero() {
    if (counter > 0) {
      counter++;
      return true;
    }
    return false;
  }

  bool decrement() {
    return (--counter == 0);
  }

  uint64_t read() { return counter; }

  uint64_t counter{1};
};
```

Thread 1:

```
increment_if_not_zero()
  if (counter > 0) {


    counter++;
    return true;
  }
```

Thread 2:

```
decrement() {

  return (--counter == 0);
}
```


Carnegie Mellon University

Making it thread safe

```
struct Counter {

  bool increment_if_not_zero() {
    std::lock_guard g_{m};
    if (counter > 0) {
      counter++;
      return true;
    }
    return false;
  }

  bool decrement() {
    std::lock_guard g_{m};
    return (--counter == 0);
  }

  std::mutex m;
  uint64_t counter{1};
};
```

Thread 1:

```
increment_if_not_zero()
  std::lock_guard g_{m};
  if (counter > 0) {
```

Thread 2:

```
decrement() {
  std::lock_guard g_{m};
```

Carnegie Mellon University

One Eternity Later

Making it thread safe

```
struct Counter {

  bool increment_if_not_zero() {
    std::lock_guard g_{m};
    if (counter > 0) {
      counter++;
      return true;
    }
    return false;
  }

  bool decrement() {
    std::lock_guard g_{m};
    return (--counter == 0);
  }

  std::mutex m;
  uint64_t counter{1};
};
```

Thread 1:

```
increment_if_not_zero()
  std::lock_guard g_{m};
  if (counter > 0) {



    counter++;
    return true;
  }
```

Thread 2:

```
decrement() {
  std::lock_guard g_{m};
```

Carnegie Mellon University

What have we learned so far?

Locks:

- Locks can eliminate ***most*** problems in concurrency!
- They do so by effectively eliminating concurrency…
- Locks often (but not always) have significant performance implications

Necessary disclaimer: Always measure performance. Never guess.

The rest of this talk:

- How to guess about performance
- We'll do some benchmarks too I promise

Carnegie Mellon University

Progress guarantees

- Progress guarantees are a way to ***theoretically*** categorize concurrent algorithms:

- **Blocking**: No guarantee
- **Obstruction free (progress in isolation)**: A single thread executed in isolation will complete the operation in a bounded number of steps.
  - Obstruction-free algorithms are immune to deadlock
- **Lock free (at least one thread makes progress)**: At any given time, ***at least one*** thread is making progress on its operation
  - Guarantees system-wide throughput. Some operations are always completing, but individual operations are never guaranteed to ever complete
- **Wait free (all threads make progress)**: Every operation completes in a bounded number of steps regardless of other concurrent operations
  - Guaranteed bounded completion time for every individual operation.

Carnegie Mellon University

A lock-free solution

```
struct Counter {

  bool increment_if_not_zero() {
    auto current = counter.load();
    while (current > 0 && !counter.compare_exchange_weak(current, current + 1)) { }
    return current > 0;
  }

  bool decrement() {
    return counter.fetch_sub(1) == 1;
  }

  uint64_t read() { return counter.load(); }

  std::atomic<uint64_t> counter{1};
};
```

A lock-free solution

```
struct Counter {

  bool increment_if_not_zero() {
    auto current = counter.load();
    while (current > 0 && !counter.compare_exchange_weak(current, current + 1)) { }
    return current > 0;
  }

  bool decrement() {
    return counter.fetch_sub(1) == 1;
  }

  uint64_t read() { return counter.load(); }

  std::atomic<uint64_t> counter{1};
};
```

```
compare_exchange(expected&, desired) {
  if (current_value == expected) {
    current_value = desired; return true; }
  else { expected = current_value; return false; }
}
```

A lock-free solution

```
struct Counter {

  bool increment_if_not_zero() {
    auto current = counter.load();
    while (current > 0 && !counter.compare_exchange_weak(current, current + 1)) { }
    return current > 0;
  }

  bool decrement() {
    return counter.fetch_sub(1) == 1;
  }

  uint64_t read() { return counter.load(); }

  std::atomic<uint64_t> counter{1};
};
```

```
compare_exchange(expected&, desired) {
  if (current_value == expected) {
    current_value = desired; return true; }
  else { expected = current_value; return false; }
}
```

A lock-free solution

```
struct Counter {

  bool increment_if_not_zero() {
    auto current = counter.load();
    while (current > 0 && !counter.compare_exchange_weak(current, current + 1)) { }
    return current > 0;
  }

  bool decrement() {
    return counter.fetch_sub(1) == 1;
  }

  uint64_t read() { return counter.load(); }

  std::atomic<uint64_t> counter{1};
};
```

```
compare_exchange(expected&, desired) {
  if (current_value == expected) {
    current_value = desired; return true; }
  else { expected = current_value; return false; }
}
```

Carnegie Mellon University

The “CAS loop”

- The so-called “CAS loop” (compare-and-swap loop) is the bread and butter of lock-free algorithms and data structures
  - Read the current state of the data structure
  - Compute the new desired state from the current state
  - Commit the change only if no one else has already changed it (compare-exchange)
  - If someone else changed it, try again
- Progress is ***lock free*** because if an operation fails to make progress (the compare-exchange returns false) it can only be because a different operation made progress
- Progress is ***not wait free*** because a particular operation can fail the CAS loop forever because of competing operations succeeding

Tools for wait freedom

- A wait-free algorithm can not contain an unbounded CAS loop
  - This does not mean you can not use compare-exchange, just not in an unbounded loop!

- Most wait-free algorithms will make use of ***atomic read-modify-write operations***:
  - **compare_exchange_weak/strong(expected, desired)**: Atomically replaces the current value with desired if current equals expected, otherwise loads the current value
  - **fetch_add(x)** / **fetch_sub(x)**: Atomically add/subtract x from the given variable and return the original value
  - **exchange(desired)**: Stores the value desired and returns the old value

Carnegie Mellon University

Towards wait-free algorithms

- The "bread and butter" of lock-free algorithms was the ***CAS loop***
- Threads ***clobber*** other threads that are trying to make progress, i.e., threads are competing to complete their operation first, at the expense of the others!
- To achieve wait freedom, threads should not be blocked by competing threads

Wait-free algorithm design

- Instead of being competitive, operations should be ***collaborative***
- The "bread and butter" of wait-free algorithm design is called ***helping***
- Operations that execute concurrently with another in-progress operation try to ***help them make progress*** instead of waiting on them or competing against them
- Requires a way for operations to detect others that are in progress, not always easy

Carnegie Mellon University

The wait-free counter design

- Operations need a way detect that other operations are in progress
- We want a way to signal to other threads that we are planning to or already have set the counter to zero.

- **Big idea**: Steal some high-order bits of the counter to use as flags

$$\underbrace{\mathbf{00}}_{\text{flags}}\underbrace{10101011101001110101010}_{\text{counter}}$$

- An operation wants to ***announce that the counter has been set to zero by setting the top flag***. (To any reading thread, top flag = 1 implies that the counter is zero).
- The second flag will be used for **helping**

Step 1: Wait-free counter without read

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }

  std::atomic<uint64_t> counter{1};
};
```

We must have counter == 0

Carnegie Mellon University

Step 1: Wait-free counter without read

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }


  std::atomic<uint64_t> counter{1};
};
```

Step 1: Wait-free counter without read

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }


  std::atomic<uint64_t> counter{1};
};
```

$\underbrace{10}_{\text{flags}}000000000000000000000000$

↓

10000000000000000000000001

10000000000000000000000010

10000000000000000000000011

⋮

Step 1: Wait-free counter without read

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }

  std::atomic<uint64_t> counter{1};
};
```

What if this compare_exchange returns false??

The decrement linearizes after the increment.

Step 2: Adding a read operation. How hard can it be?

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }

  uint64_t read() {
    auto val = counter.load();
    return (val & is_zero) ? 0 : val;
  }

  std::atomic<uint64_t> counter{1};
};
```

counter = 0

Thread 1:	Thread 2:	Thread 3:
decrement() counter.fetch_sub(1)	read() counter.load() // 0 **return 0**	increment_if_not_zero() counter.fetch_add(1)

If counter == 0, we can't affirm that the counter is *actually* zero! (It might go back to 1 because of an increment)

Carnegie Mellon University

Step 2: Wait-free counter with read?

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }

  uint64_t read() {
    auto val = counter.load();
    if (val == 0 then what?

  }

  std::atomic<uint64_t> counter{1};
```

Let's **help** them!!

There **must** be a thread about to do this!

Carnegie Mellon University

Step 2: Wait-free counter with read?

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;
  }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      return counter.compare_exchange_strong(e, is_zero);
    }
    return false;
  }

  uint64_t read() {
    auto val = counter.load();
    if (val == 0 && counter.compare_exchange_strong(val, is_zero)) return 0;    // helping!
    return (val & is_zero) ? 0 : val;
  }

  std::atomic<uint64_t> counter{1};
```

Did we fix the bug?

Yes! But we added another one…

If a read helps to set the **is_zero** flag, none of the decrements return true…

Carnegie Mellon University

Step 3: Almost there

```
struct Counter {
  static constexpr unsigned uint64_t is_zero = 1ull << 63;
  static constexpt unsigned uint64_t helped = 1ull << 62;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;   }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      if (counter.compare_exchange_strong(e, is_zero)) return true;
      else if ((e & helped) && (counter.exchange(is_zero) & helped)) return true;
    }
    return false;
  }

  uint64_t read() {
    auto val = counter.load();
    if (val == 0 && counter.compare_exchange_strong(val, is_zero | helped)) return 0;    // helping!
    return (val & is_zero) ? 0 : val;  }

  std::atomic<uint64_t> counter{1};
```

Carnegie Mellon University

Step 3: Almost there

```
struct Counter {
  static constexpr uint64_t is_zero = 1ull << 63;
  static constexpt uint64_t helped = 1ull << 62;

  bool increment_if_not_zero() {
    return  (counter.fetch_add(1) & is_zero) == 0;   }

  bool decrement() {
    if (counter.fetch_sub(1) == 1) {
      uint64_t e = 0;
      if (counter.compare_exchange_strong(e, is_zero)) return true;
      else if ((e & helped) && (counter.exchange(is_zero) & helped)) return true;
    }
    return false;
  }

  uint64_t read() {
    auto val = counter.load();
    if (val == 0 && counter.compare_exchange_strong(val, is_zero | helped)) return 0;    // helping!
    return (val & is_zero) ? 0 : val;  }

  std::atomic<uint64_t> counter{1};
```

Step 4: Got it!

```
struct Counter {
   static constexpr uint64_t is_zero = 1ull << 63;
   static constexpt uint64_t helped = 1ull << 62;

   bool increment_if_not_zero() {
      return  (counter.fetch_add(1) & is_zero) == 0;   }

   bool decrement() {
      if (counter.fetch_sub(1) == 1) {
         uint64_t e = 0;
         if (counter.compare_exchange_strong(e, is_zero)) return true;
         else if ((e & helped) && (counter.exchange(is_zero) & helped)) return true;
      }
      return false;
   }

   uint64_t read() {
      auto val = counter.load();
      if (val == 0 && counter.compare_exchange_strong(val, is_zero | helped)) return 0;    // helping!
      return (val & is_zero) ? 0 : val;  }

   std::atomic<uint64_t> counter{1};
```

Caref
multip
after

Solution: One and *only one* decrement must "take credit" for zeroing the counter

Carnegie Mellon University

Benchmarks

- My **atomic<shared_ptr>** implementation using the wait-free counter versus the lock-free counter. This affects the performance of the **load** operation.

- **Benchmark #1**: p threads loading from an **atomic<shared_ptr>**

Load Latency

	1%	50%	99%
$p = 1$	**21n**	**24.8n**	32.7n
$p = 28$	217n	1.75u	13.1u
$p = 56$	535n	5.36u	31.1u

Lock Free Counter

Load Latency

	1%	50%	99%
$p = 1$	21.7n	**24.8n**	**31.7n**
$p = 28$	**158n**	**1.31u**	**8.40u**
$p = 56$	**146n**	**2.43u**	**13.3u**

Wait Free Counter

Carnegie Mellon University

Benchmarks

- My **atomic<shared_ptr>** implementation using the wait-free counter versus the lock-free counter. This affects the performance of the **load** operation.

- **Benchmark #2**: p threads, **50% loading** from an **atomic<shared_ptr>**, the other **50% storing**

Load Latency

	1%	50%	99%
$p = 28$	212n	**1.19u**	8.67u
$p = 56$	215n	2.41u	26.7u

Lock Free Counter

Load Latency

	1%	50%	99%
$p = 28$	**164n**	1.21u	**7.65u**
$p = 56$	**153n**	**2.05u**	**26.4u**

Wait Free Counter

Benchmarks

- My **atomic<shared_ptr>** implementation using the wait-free counter versus the lock-free counter. This affects the performance of the **load** operation.

- **Benchmark #3**: p threads, **10% loading** from an **atomic<shared_ptr>**, the other **90% storing**

Load Latency

	1%	50%	99%
$p = 28$	217n	1.73u	14.1u
$p = 56$	**240n**	**6.40u**	**82.6u**

Lock Free Counter

Load Latency

	1%	50%	99%
$p = 28$	**188n**	**1.41u**	**13.3u**
$p = 56$	244n	7.46u	86.4u

Wait Free Counter

Carnegie Mellon University

Benchmarks

Remember: Always measure performance. Never guess.

- Which algorithm is best often depends on the ***workload***
  - How many reads vs how many writes
  - How many threads/cores
- Wait free was better for ***read-mostly***, but lock free looks better for ***write-mostly***

Carnegie Mellon University

Take-home messages

Progress guarantees

- Useful theoretical classification of concurrent algorithms that can inform algorithm design
- Lock-free algorithms guarantee that one thread is making progress, while ***wait-free algorithms guarantee that every thread is making progress***

Wait-free algorithm design

- The bread-and-butter technique is ***helping***. Operations help concurrent operations rather than waiting for them (blocking) or compete with them (lock-free)

Performance Implications

- Never guess about performance
- ***But do*** hypothesize about performance by analyzing an algorithm's progress guarantees, and use these progress guarantees to guide the design of your algorithm
- ***Then*** benchmark it