Lock-Free Programming

Julio Arroyo

Update on October 10, 2023

Contents

List of Figures

List of Theorems

List of Definitions

Part I

Theory

Chapter 1

Locks and Mutual Exclusion

Contents

gls example

- Greatest Common Divisor (GCD); Greatest Common Divisor; GCD; Greatest Common Divisor (GCD)

1.1 Critical Sections

Definition 1.1 (Mutual exclusion property)**.** When only one thread executes at the same time.

Definition 1.2 (Critical section)**.** A block of code that must be executed under *mutual exclusion* among threads in order to guarantee program correctness.

If multiple threads interleave when executing a critical section, the program may fail. The standard way of securing mutual exclusion for a critical section is through the use of a `Lock`.

```
template <typename T>
concept LockConcept = requires (T t)
{
    { t.lock() };
    { t.unlock() };
};
```

Code 1: The Lock interface, using C++ 20 concepts.

We can evaluate the "goodness" of a `Lock` algorithm using the following three properties.

1. **Mutual Exclusion**
2. **Deadlock-free:** "If one or more threads try to acquire the lock, then some thread will succeed". Equivalently we can say "if there is a thread trying to acquire the lock, then it is not the case that no one gets it."

3. **Starvation-free:** "If a given thread tries to acquire the lock, then that thread will get the lock eventually". Equivalently, we can say "every call to `lock()` eventually returns".

Corollary 1.1. *L is a starvation-free lock $\Rightarrow$ L is a deadlock-free lock.*

It is worth pointing out that a program that uses only deadlock-free locks may still deadlock. TODO: give example.

Lemma 1.1.

Claim 1.1.

Theorem 1.2.

Example 1.1.

Fact 1.1.

Remark 1.1.

Exercise 1.1. Prove A iff B

Solution. By induction: □

1.2 2-Thread Locks

Let us progressively build a starvation-free lock for two threads.

1.2.1 Mutually exclusive, but deadlocks in concurrent lock() calls.

As a first (unsuccessful) attempt, consider an algorithm where the lock maintains one flag for each of the two threads. Thus, a thread sets its flag to true whenever it wants to acquire the lock; so a thread blocks while the other thread has the lock.

```
class BadTwoThreadedLock
{
public:
    void lock() {
        int this_thread_id = ThreadID.get();  // either 0 or 1
        int other_thread_id = 1 - this_thread_id;

        interested[this_thread_id] = true;
        while (interested[other_thread_id]) {}
    }

    void unlock() {
        int this_thread_id = ThreadID.get();
        interested[this_thread_id] = false;
    }

private:
    bool interested[2] = {false, false};
};
static_assert(LockConcept<BadTwoThreadedLock>);
```

Code 2: Simple but incorrect two threaded lock.

Let us evaluate `BadTwoThreadedLock`:

1. Does `BadTwoThreadedLock` satisfy mutual exclusion? Yes, proof left as an exercise to the reader.
2. Is `BadTwoThreadedLock` deadlock-free? No. If two threads call `lock()` at the same time, it may deadlock:
   - Thread-0 sets its own flag to true.
   - Thread-1 sets its own flag to true.
   - Thread-0 reads Thread-1's flag to be true, so it spins in the while loop.
   - Thread-1 reads Thread-0's flag to be true, so it spins in the while loop.
   - `BadTwoThreadedLock` is in deadlock.
3. Is `BadTwoThreadedLock` starvation-free? No. Since it isn't deadlock-free, then it isn't starvation-free, by contrapositive of Corollary 1.1.

1.2.2 Mutually exclusive, but deadlocks in sequential lock() calls.

As a second (unsuccessful) attempt let's try to fix our first attempt; namely, let's make a lock that does not deadlock if two threads call `lock()` at the same time. This can be achieved via the following simple algorithm: when a thread wants to acquire the lock, it will first let the other thread acquire it.

```
class AnotherBadTwoThreadedLock
{
public:
    void lock() {
        int this_thread_id = ThreadID.get();
        waiter = this_thread_id;
        while (waiter == this_thread_id) {}
    }

    void unlock() {}

private:
    int waiter;
};
static_assert(LockConcept<BadTwoThreadedLock>);
```

Code 3: Another simple but incorrect two threaded lock.

Clearly, `AnotherBadTwoThreadedLock` satisfies mutual exclusion. However, it is not deadlock-free (so it also isn't starvation-free). Why? Think about what happens if only Thread-0 tries to acquire the lock and Thread-1 never does: it will wait forever to let Thread-1 go first. But this violates deadlock-freedom, since there is someone trying to acquire the lock but no one gets it.

1.2.3 Peterson Lock: a two-threaded starvation-free algorithm

Third time is the charm, so if we combine the ideas of our first two attempts we arrive at a starvation-free lock called the **Peterson Lock**.

```
class PetersonNaive {
public:
    void lock() {
        int thisID = ThreadID.get();  // either 0 or 1
        int otherID = 1 - thisID;

        interested[thisID] = true;
        waiter = thisID;
        while (interested[otherID] && waiter == thisID) {}
    }

    void unlock() {
        int thisID = std::this_thread::get_id() == firstThread.load();
        interested[thisID] = false;
    }

private:
    bool interested[2] = {false, false};
    int waiter{-1};
};
static_assert(LockConcept<PetersonNaive>);
```

Code 4: A simple, starvation-free lock algorithm. It also happens to be incorrect, since memory accesses are not sequentially consistent. See Exercise 1.4

Claim 1.2. *Peterson's Algorithm satisfies mutual exclusion.*

Proof. Left as an exercise to the reader. Hint: by contradiction. □

Claim 1.3. *Peterson's Lock is a starvation-free lock.*

Proof. Left as an exercise to the reader. Hint: by contradiction. □

Claim 1.4. *Peterson's Lock is a deadlock-free lock.*

Proof. Follows immediately from the fact that it is a starvation-free lock. □

1.3 Lamport's Bakery Algorithm: a fair, N-threaded lock

1.3.1 Fairness in locks

Informally speaking, a lock could be considered fair if it is first-come-first-served. Somewhat more precisely, if thread A calls `lock()` before thread B does, then thread A should enter the critical section before thread B. Let us formalize this notion.

First, some mathematical formalisms. Any call to `lock()` consists of two execution time intervals:

1. A doorway interval, whose execution consists of a bounded number of steps.
2. A waiting interval, whose execution may consist of an unbounded number of steps.

Now, we are ready to formally define fairness.

Definition 1.3 (First-come-first-served)**.** Consider two threads, A and B. A lock that is first-come-first-served guarantees that:

if A's doorway interval precedes B's doorway interval, then A's critical section interval precedes B's critical section interval.

1.3.2 Bakery Algorithm

Suppose there is a bakery. Whenever a client comes in, he takes a number. The next client to be served is the one with the lowest number that hasn't yet been served.

We can implement an n-threaded lock algorithm using the idea of the bakery. `interested[i]` is a flag indicating if thread i wants to acquire the lock, and `label[i]` is a number showing the relative order when threads first expressed interest in acquiring the lock. `label[i]` is assigned in two steps: scanning all previous labels, and incrementing the maximum by one. Note that, due to execution interleaving, ties are possible: multiple threads may end up with the same label. In such cases, ties are broken by deferring to the thread with lowest id.

```
class BakeryLock
{
public:
    BakeryLock(int num_threads) {
        n = num_threads;
        for (int i = 0; i < n; i++) {
            interested.push_back(false);
            label.push_back(0);
        }
    }

    void lock() {
        int this_thread = ThreadID.get();

        interested[this_thread] = true;
        label[this_thread] = max(label) + 1;

        // spin while there is another thread that is interested in
        // acquiring the lock AND arrived before this thread.
        while ((THERE EXISTS other_thread != this_thread) SUCH THAT
            (
                interested[other_thread] &&
                (
                    label[other_thread] < label[this_thread] ||
                    (
                        // break tie when two threads have same label
                        label[other_thread] == label[this_thread] && (other_thread < this_thread
                    )
```

```
                )
            )) {}

    void unlock() {
        interested[ThreadID.get()] = false;
    }

private:
    int n;
    std::vector<bool> interested;
    std::vector<bool> label;
};
static_assert(LockConcept<BakeryLock>);
```

Code 5: Pseudo-code for the simplest, best-known, starvation-free n-thread lock algorithm.

1.4 Theory digression

1.4.1 Bounded Timestamps

Note that the timestamps (labels) used in the Bakery Algorithm grow without a bound. This motivates the fundamental question: can we place a bound on the number of timestamps needed for a n-threaded deadlock-free lock algorithm?

Think about the two-threaded case. This can be achieved using three timestamps: 0,1,2 (as long as we agree that 2 comes before 0 and timestamps are consecutive). This notion can be generalized to the n-threaded case.

1.4.2 Lower Bounds on the Number of Read/Write Locations

Note that the Bakery algorithm requires reading and writing from $O(n)$ distinct locations, where n is the maximum number of concurrent threads. This motivates the following question: is there a clever lock algorithm, based solely on reading and writing memory, that avoids this overhead?

The answer is no. Any deadlock-free lock algorithm requires allocating and then reading or writing at least n distinct locations in the worst case. What does this mean for the design of modern multiprocessor machines? That we need synchronization operations stronger than read/write, and use them as the basis of our mutual exclusion algorithms.

The other important question is why this lower bound on the number of read/write locations an "immutable fact of nature"? The answer is based on the principle that any memory location written to by one thread may be overwritten without any other thread ever seeing it.

1.5 Algorithm

Algorithm 1.5.1: Primality testing - first attempt

input : Integer N and parameter 1^t
output: A decision as to whether N is prime or composite

```
for i = 1, 2, ..., t do
    a ← {1, ..., N_1};
    if a^{N-1} ≠ 1 mod N then
        return "composite"
return "prime"
```

1.6 Exercises

Exercise 1.2. (**Herlihy & Shavit Chapter 2 Exercise #12**)

Solution. TODO check this answer.
Achieves mutual exclusion, but is not deadlock-free or starvation-free. □

Exercise 1.3. (**Herlihy & Shavit Chapter 2 Exercise #15**)

Solution. The `FastPath` lock does NOT satisfy the mutual exclusion property.
Consider two threads contending for the lock (calling `lock()` at the same time). Then the following interleaving may happen:

- Thread-0 sets $x \leftarrow 0$
- Thread-1 sets $x \leftarrow 1$
- Thread-0 reads $y == -1$ so it does not spin on the first while loop.
- Thread-1 also reads $y == -1$ so it also does not spin on the first while loop.
- Thread-0 sets $y \leftarrow 0$
- Thread-0 reads $x == 1$, so it enters IF statement, calls `lock.lock()`, and enters its critical section.
- Thread-1 sets $y \leftarrow 1$, reads $x == 1$, so it does not go inside the IF statement, and enters its own critical section.
- Both threads are in the critical section at the same time.

□

Exercise 1.4. Fixing the Peterson Lock implementation, see Code 1.2.3.

1. Write a program that consists of two threads. Each increments a shared counter, say half-a-million times, using the Peterson Lock for mutual exclusion. Does the shared counter add up to a million, as expected?

2. Explain what went wrong with Part 1.

3. Fix Part 1, the counter should be exactly one million.

Solution.

```
#include "Peterson.hpp"

int main() {
    int counter = 0;
    PetersonGood mutex;

    auto f = [&mutex, &counter]() {
        for (int i = 0; i < 500000; i++) {
            mutex.lock();
            counter++;
            mutex.unlock();
        }
    };

    std::thread thread1(f);
    std::thread thread2(f);

    thread1.join();
    thread2.join();

    std::cout << "Final result: " << counter <<  std::endl;
    return 0;
}
```

The counter adds up to around 970,000 but does not reach a million.

The problem is that memory accesses are not sequentially consistent, which causes access to the shared counter not to be mutually exclusive. Suppose the lines `interested[thisID] = true;` and `waiter = thisID;` were flipped. Then imagine the following interleaving:

- Thread B writes `waiter = B`
- Thread A writes `waiter = A`
- Thread A writes `interested[A] = true`
- Thread A reads `interested[B] = false`
- Thread A enters critical section.
- Thread B writes `interested[B] = true`
- Thread B reads `interested[A] = true`, but then also reads `waiter = A`.
- Thread B enters critical section before A has left it.

To avoid the memory accesses from being reordered, we can implement the Peterson Lock in the following way, and the counter does add up to a million.

```
class PetersonGood {
public:
    void lock() {
        std::thread::id currThreadID = std::this_thread::get_id();

        // mark first thread to acquire lock
        std::thread::id defaultID = std::thread::id();
        if (firstThread.compare_exchange_strong(defaultID,
```

```
                                                currThreadID,
                                                std::memory_order_seq_cst,
                                                std::memory_order_seq_cst)) {
            std::cout << "FIRST THREAD: " << firstThread << std::endl;
        }

        int thisID = currThreadID == firstThread.load();
        int otherID = 1 - thisID;
        interested[thisID].store(true);
        waiter.store(thisID);
        while (interested[otherID].load() && waiter.load() == thisID) {}
    }

    void unlock() {
        int thisID = std::this_thread::get_id() == firstThread.load();
        interested[thisID].store(false);
    }

private:
    std::atomic_bool interested[2] = {false, false};
    std::atomic_int8_t waiter{-1};
    std::atomic<std::thread::id> firstThread;
};
static_assert(LockConcept<PetersonGood>);
```

□

Chapter 2

Concurrent Objects

Contents

2.1 Concurrency and Correctness

2.1.1 Quiescent Consistency

2.1.2 Sequential Consistency

Think about the intuitive notion for sequential consistency in a single-threaded application: if the program defines a sequence of instructions a, b, c (called program order), then the order in which the instructions should be executed is a, b, c. This notion can be extended to multi-threaded applications.

Definition 2.1 (Sequential consistency in multi-threaded programs)**.** A sequentially consistent multi-threaded program is one where the program order of each individual thread is preserved.

Informally speaking, method calls of sequentially consistent objects may be re-ordered as long as the program order of every thread is preserved.

Note the (perhaps counter-intuitive) fact that sequential consistency allows for two calls from different threads that do not overlap to be re-ordered. This means that sequential consistency does not necessarily respect real-time ordering. This can be one motivation to introduce the stronger notion of Linearizability.

Lemma 2.1. *Sequential consistency is a non-blocking correctness condition, because for every pending invocation of a total method, there exists a sequentially consistent response.*

Lemma 2.2. *In general, composing multiple sequentially consistent objects does not result in a sequentially consistent system.*

Memory accesses are NOT sequentially consistent in modern multiprocessor architectures.

The order in which your code reads and writes memory is not necessarily the exact order in which memory ends up being actually read from or written to. This is the result of optimization tricks done by both the

CPU and the compiler. While these reorderings are invisible in single-threaded programs, they may result in unexpected behavior in multi-threaded programs.

Consider a two-threaded program with two variables initialized to $x \leftarrow 0$ and $y \leftarrow 0$.

```
# Thread 1
x = 42
y = 1

# Thread 2
while y == 0:
    continue
print x
```

Code 6: Pseudo-code for a two-threaded program. If the x,y variables were accessed in sequentially consistent order, the printed output should be 42. However, if thread 1's instructions are re-ordered, then there is an interleaving where the printed output is 0.

Why are memory accesses re-ordered, in the first place? Think about it from the perspective of multi-processor architectural design. When a processor writes to a variable, that change is reflected in the cache but eventually it must be reflected on main memory as well. The first option is that any writes done in one processor's cache is immediately applied to main memory as well. A better option (used in practice) is to temporarily queue them up in a *store buffer* (or *write buffer*); this provides at least two key benefits: writes to different addresses can be batched together and applied at once in a single trip to main memory, and multiple writes to the same address can be absorbed into one.

When you need sequential consistency, one option is to use *memory barriers* (or *fences*), which are CPU instructions that flush out write buffers. They can be expensive, though, using at least 10^2 cycles.

2.1.3 Linearizability

Definition 2.2. A *linearizable* concurrent object is one in which every call appears to happen instantaneously sometime between the invocation and response.

Lemma 2.3. *Linearizability is compositional: the result of composing linearizable objects is linearizable.*

Informally speaking, method calls of different threads can only be re-ordered if they are concurrent.

Lemma 2.4. *Linearizability* $\Rightarrow$ *sequential consistency.*

2.2 Concurrency and Progress

Fact 2.1. Unexpected thread delays are common in multiprocessors:

- A cache miss delays a processor for 10^2 cycles.
- A page fault delays a processor for 10^6 cycles.
- Preemption delays a processor for 10^8 cycles.

Definition 2.3. A *wait-free* algorithm is one where every call finishes in a finite number of steps.

Definition 2.4. A *lock-free* algorithm is one that guarantees that infinitely often some thread finishes in a finite number of steps. Note that this allows for some threads to starve.

Definition 2.5. An *obstruction-free* algorithm guarantees that when a thread executes in isolation it finishes in a finite number of steps.

Lemma 2.5. *Wait-free* $\Rightarrow$ *lock-free* $\Rightarrow$ *obstruction-free.*

2.3 The C++ Memory Model

A memory model describes the interactions of threads through memory and their shared use of the data.

2.4 A Wait-Free, Sequentially Consistent, Single-Producer-Single-Consumer Ring Buffer

---

```
#include <memory>  // unique_ptr
#include <atomic>

template <typename T>
class SPSCqueue
{
public:
    SPSCqueue(int capacity)
    : capacity_{capacity},
      head_{0},
      tail_{0}
    {
        ringBuffer_ = std::make_unique<T[]>(capacity);
    }

    void enqueue(T v) {
        if (tail_.load(std::memory_order_seq_cst) - head.load(std::memory_order_seq_cst) == capa
            throw std::exception("FullException");
        }
        int tail = tail_.load(std::memory_order_seq_cst);
        ringBuffer_[tail] = v;
        tail_.store((tail+1) % capacity_, std::memory_order_seq_cst);
        return;
    }

    T dequeue() {
        if (tail_.load(std::memory_order_seq_cst) == head.load(std::memory_order_seq_cst)) {
            throw std::exception("EmptyException");
        }

        int head = head_.load(std::memory_order_seq_cst);
        T v = ringBuffer_[head];
```

```
        head_.store((head+1) % capacity_, std::memory_order_seq_cst);
        return v;
    }

private:
    int capacity_;
    std::atomic_int head_;
    std::atomic_int tail_;
    std::unique_ptr<T[]> ringBuffer_;
};
```

Code 7: False sharing prevention isn't implemented. Looser memory ordering semantics can be used.

Part II

Practice

Chapter 3

Spin Locks and Contention

Contents

3.1 Exercises

Exercise 3.1. Comparison of TAS and TTAS lock.

1. Produce performance plot, with x-axis for number of threads and y-axis for time.
2. Count number of cache misses for each.