Worksheet 3

Simon Grätz

May 20, 2024

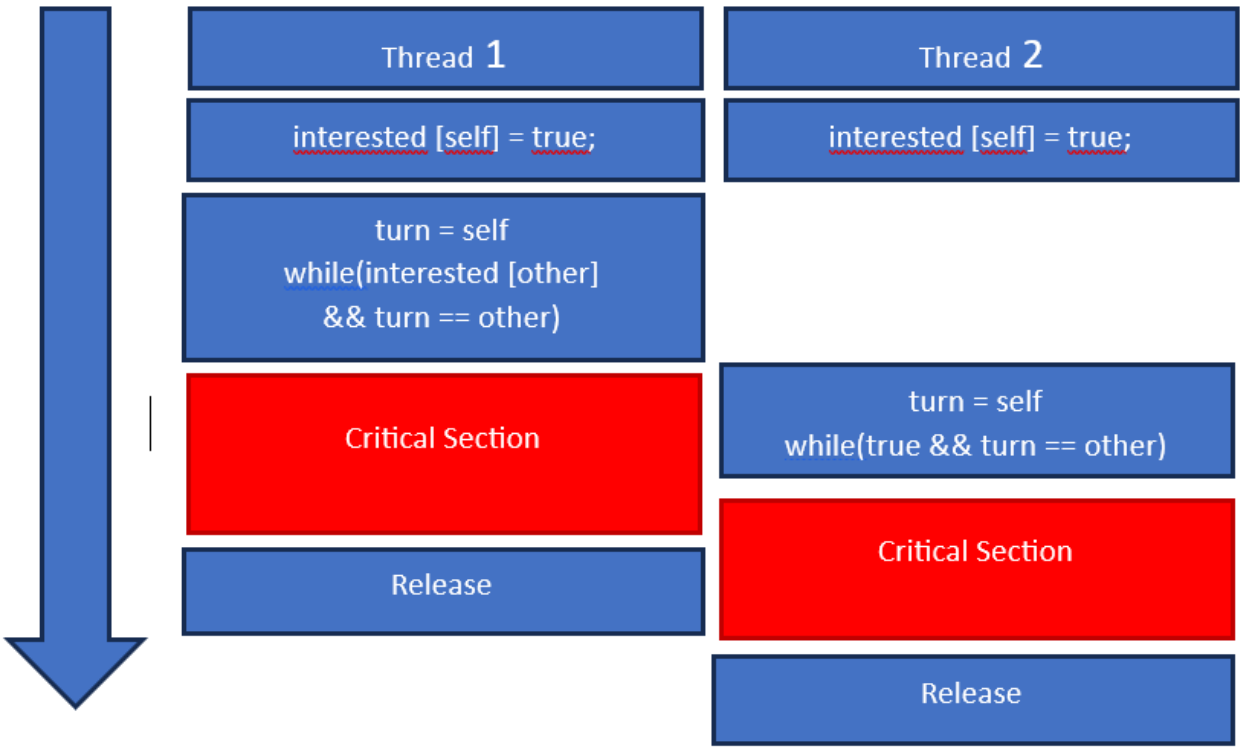
Contents

1 Peterson Lock

1.1 Peterson Lock in more selfish

1.1.1 Does the Peterson Lock still guarantee mutual exclusion?

Answer: No, because if each thread can set its own turn to self, there is a possibility of an overlap of the critical sections, if one thread happens its to set its turn to self after the first thread has entered but not left the critical section.



1.1.2 Is the lock still starvation free?

Answer: Yes, because either the mutual exclusion is not guaranteed but both threads can execute to completion - though be it with incorrect results. Or The release of the first thread will free the second from its limbo.

1.2 Atomic Peterson Lock

1.2.1 Implementation

Implement the lock using C++ atomics. Note that the pseudocode representation assumes a sequentially consistent memory model. Devise an approach for testing the correctness of your implementation.

```
#include <atomic>
#include <iomanip>
#include <iostream>
#include <thread>
#include <vector>

class AtomicPetersonLock {
private:
    std::atomic<int> turn;
    std::atomic<bool> interested[2];

public:
    AtomicPetersonLock() {
        turn.store(0, std::memory_order_release);
        interested[0].store(false, std::memory_order_release);
        interested[1].store(false, std::memory_order_release);
    }

    ~AtomicPetersonLock() {}

    void aquire(int thread_id) {
        int other = 1 - thread_id;
        interested[thread_id].store(true, std::memory_order_acquire);
        turn = other;

        int spins = 0;
        while (interested[other].load(std::memory_order_acquire) &&
            turn.load(std::memory_order_acquire) == other) {
            if (spins++ % 10000000 == 0) {
                std::cout << "thread" << thread_id << " spined for " << spins << "spins\n";
            }
        }
        std::cout << "aqurired by thread" << thread_id << "\n";
    }

    void release(int thread_id) {
        std::cout << "released by thread" << thread_id << "\n";
        interested[thread_id].store(false, std::memory_order_release);
    }
};
```

1.2.2 Test Implementation

Compare the 100000000 add 1 operations by each thread to a global variable. If no thread is used the result is different in each iteration. If the atomic Peterson lock implementation is used the result is correctly 200000000 each time – so it is not just coincidental. Also each iteration run without starvation and terminates.

```
void test_atomic_peterson_lock(void) {
    int global_int = 0;
    AtomicPetersonLock lock = AtomicPetersonLock();

    std::cout << "\n\nAtomic PetersonLock: \n";
    auto func = [&global_int, &lock](int thread_id) {
        lock.aquire(thread_id);
        for (auto i = 0; i < 100000000; ++i) {
            global_int += 1;
        }
        lock.release(thread_id);
    };
    auto func_without_lock = [&global_int](int thread_id) {
        for (auto i = 0; i < 100000000; ++i) {
            global_int += 1;
        }
    };

    std::cout << "Compare with lock vs without vs should be:\n";
    std::cout << "| Atomic Peterson Lock |  No Lock              | Expected "
        "value        |\n";
    for (auto i = 0; i < 10; ++i) {
        global_int = 0;
        {
            std::thread t1;
            std::thread t2;
            t1 = std::thread{func, 0};
            t2 = std::thread{func, 1};
            t1.join();
            t2.join();
        }
        int result = global_int;
        global_int = 0;
        {
            std::thread t1;
            std::thread t2;
            t1 = std::thread{func_without_lock, 0};
            t2 = std::thread{func_without_lock, 1};
            t1.join();
            t2.join();
        }
        std::cout << "| " << result << std::setw(14) << " | ";
        std::cout << global_int << std::setw(15) << " | ";
        std::cout << 200000000 << std::setw(15) << " |\n";
    }
}

auto main(int argc, char *argv[]) -> int {
    test_atomic_peterson_lock();
    return 0;
}
```

2 MCS Lock

2.1 Implement the [MCS] lock using C++ atomics.

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

struct QNode { std::atomic<bool> wait; std::atomic<QNode *> next; };

class AtomicMCSLock {
private:
    std::atomic<QNode *> tail;

    QNode *swap(std::atomic<QNode *> &_tail, QNode *_p) {
        return _tail.exchange(_p, std::memory_order_acq_rel);
    }

    bool cas(std::atomic<QNode *> &_tail, QNode *_expected, QNode *_desired) {
        return _tail.compare_exchange_weak(_expected, _desired,
            std::memory_order_acq_rel);
    }

public:
    void acquire(QNode *p) {
        p->next.store(nullptr);
        p->wait.store(true);

        QNode *prev = swap(tail, p);

        if (prev) {
            prev->next.store(p, std::memory_order_release);
            while (p->wait.load(std::memory_order_acquire)) {
                //
            }
        }
        std::cout << "acquired\n";
    }

    void release(QNode *p) {
        QNode *succ = p->next.load(std::memory_order_acquire);

        if (!succ) {
            auto desired = p;
            if (cas(tail, desired, nullptr)) {
                return;
            }
            do {
                succ = p->next.load(std::memory_order_acquire);
            } while (succ == nullptr);
        }
        succ->wait.store(false, std::memory_order_release);
        std::cout << "released\n";
    }
};
```

2.2 Test the implementation

Here I ran out of time...

```
auto main() -> int {
    std::thread t1;
    std::thread t2;

    AtomicMCSLock mcs = AtomicMCSLock();
    QNode node;
    int global = 0;

    auto func = [&mcs, &global, &node]() {
        mcs.acquire(&node);
        for (auto i = 0; i < 100000000; ++i) {
            global += 1;
        }
        mcs.release(&node);
    };

    std::cout << "global = " << global << "\n";
    t1 = std::thread{func};
    t2 = std::thread{func};
    t1.join();
    t2.join();
    std::cout << "global = " << global << "\n";

    return 0;
}
```