Mutual Exclusion

EECS 3/495 “Rust”

Spring 2017

Definitions: Time

- A thread A has *atomic* events $a_0, a_1, a_2, \ldots$

Definitions: Time

- A thread A has *atomic* events $a_0, a_1, a_2, \ldots$
- Event a_i^k is the kth occurrence of event a_i

Definitions: Time

- A thread A has *atomic* events $a_0, a_1, a_2, \ldots$
- Event a_i^k is the kth occurrence of event a_i
- $a \rightarrow b$ means event a *precedes* event b
  - $(\rightarrow)$ is a total order on events

Definitions: Time

- A thread A has *atomic* events $a_0, a_1, a_2, \ldots$
- Event a_i^k is the kth occurrence of event a_i
- $a \to b$ means event a *precedes* event b
  - $(\to)$ is a total order on events
- An *interval* $I_A = (a_0, a_1)$ is the duration between a_0 and a_1

Definitions: Time

- A thread A has *atomic* events $a_0, a_1, a_2, \ldots$
- Event a_i^k is the kth occurrence of event a_i
- $a \to b$ means event a *precedes* event b
  - $(\to)$ is a total order on events
- An *interval* $I_A = (a_0, a_1)$ is the duration between a_0 and a_1
- Interval I_A^k is the kth occurrence of interval I_A

Definitions: Time

- A thread A has *atomic* events $a_0, a_1, a_2, \ldots$
- Event a_i^k is the kth occurrence of event a_i
- $a \rightarrow b$ means event a *precedes* event b
  - $(\rightarrow)$ is a total order on events
- An *interval* $I_A = (a_0, a_1)$ is the duration between a_0 and a_1
- Interval I_A^k is the kth occurrence of interval I_A
- $(a_0, a_1) \rightarrow (b_0, b_1)$ means that $a_1 \rightarrow b_0$
  - $(\rightarrow)$ is a partial order on intervals

A counter class

```
class Counter
{
public:
    int get_and_inc()
    {
        int old = count_;
        count_ = old + 1;
        return old;
    }

private:
    int count_ = 0;
};
```

A problem

```
Counter c;

std::thread t1([&]() { c.get_and_inc(); });
std::thread t2([&]() { c.get_and_inc(); });

t1.join();
t2.join();

CHECK_EQUAL(2, c.get_and_inc());
```

Counter class has a critical section

```
class Counter
{
public:
    int get_and_inc()
    {
        int old = count_;       // danger begins
        count_ = old + 1;       // danger ends
        return old;
    }

private:
    int count_ = 0;
};
```

Mutual exclusion

We need *mutual exclusion*!

Mutual exclusion

We need *mutual exclusion*!

Definition: Critical sections can't overlap

More formally, for threads A, B, and integers j and k, either $CS_A^j \rightarrow CS_B^k$ or $CS_B^k \rightarrow CS_A^j$.

Solution: A lock (a/k/a mutex)

```
class ILock
{
    virtual void lock()   = 0;
    virtual void unlock() = 0;
};
```

Solution: A lock (a/k/a mutex)

```
class ILock
{
    virtual void lock()   = 0;
    virtual void unlock() = 0;
};

class Lock : public ILock { ··· };
```

Using a lock

```
class Counter
{
public:
    int get_and_inc()
    {
        lock_.lock();
        int old = count_;
        count_ = old + 1;
        lock_.unlock();
        return old;
    }

private:
    int count_ = 0;
    Lock lock_;
};
```

RAII: Resource Acquisition Is Initialization

```
class Lock_guard
{
public:
    Lock_guard(ILock & lock) : lock_(lock)
    {
        lock_.lock();
    }

    ~Lock_guard()
    {
        lock_.unlock();
    }

private:
    ILock & lock_;
};
```

Using a lock—RAII-style

```
class Counter
{
public:
    int get_and_inc()
    {
        Lock_guard guard(lock_);
        int old = count_;
        count_ = old + 1;
        return old;
    }

private:
    int count_ = 0;
    Lock lock_;
};
```

How to implement the lock?

Two-thread solutions first, then *n*-thread solutions

Base class for two-thread lock

```
class Lock_base : public ILock
{
    // i() is this thread:
    thread::id i() const
    {
        return this_thread::get_id();
    }

    // j() is the other thread:
    thread::id j() const
    {
        return i().other_thread();
    }
};
```

An attempt

```
class Lock_one : public Lock_base
{
    bool flag_[2] = {false, false};

public:
    virtual void lock() override
    {
        flag_[i()] = true;
        while (flag_[j()]) {}
    }

    virtual void unlock() override
    { flag_[i()] = false; }
};
```

Theorem

Lock_one satisfies mutual exclusion.

Theorem

Lock_one satisfies mutual exclusion. Proof by contradiction:

- Assume CS_A overlaps CS_B

Theorem

Lock_one satisfies mutual exclusion. Proof by contradiction:

- Assume CS_A overlaps CS_B
- Consider each thread's last read and write in *lock*() before entering its CS. For *A* to enter, it first writes true to its flag, and then needs to read false from the other's:
  - $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false}) \rightarrow CS_A$

Theorem

Lock_one satisfies mutual exclusion. Proof by contradiction:

- Assume CS_A overlaps CS_B
- Consider each thread's last read and write in *lock*() before entering its CS. For *A* to enter, it first writes true to its flag, and then needs to read false from the other's:
  - $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false}) \rightarrow CS_A$

  And by symmetry:
  - $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false}) \rightarrow CS_B$

Theorem

Lock_one satisfies mutual exclusion. Proof by contradiction:

- Assume CS_A overlaps CS_B
- Consider each thread's last read and write in *lock*() before entering its CS. For *A* to enter, it first writes true to its flag, and then needs to read false from the other's:
  - $write_A(flag[A] = true) \rightarrow read_A(flag[B] == false) \rightarrow CS_A$

  And by symmetry:
  - $write_B(flag[B] = true) \rightarrow read_B(flag[A] == false) \rightarrow CS_B$
- Note, also, that if *A* sees *B*'s flag as false, that must happen before *B* writes its flag, and by symmetry for *B* seeing *A*'s flag:
  - $read_A(flag[B] == false) \rightarrow write_B(flag[B] = true)$
  - $read_B(flag[A] == false) \rightarrow write_A(flag[A] = true)$

Theorem

Lock_one satisfies mutual exclusion. Proof by contradiction:

- Assume CS_A overlaps CS_B
- Consider each thread's last read and write in *lock*() before entering its CS. For A to enter, it first writes true to its flag, and then needs to read false from the other's:
  - $write_A(flag[A] = true) \rightarrow read_A(flag[B] == false) \rightarrow CS_A$

  And by symmetry:
  - $write_B(flag[B] = true) \rightarrow read_B(flag[A] == false) \rightarrow CS_B$
- Note, also, that if A sees B's flag as false, that must happen before B writes its flag, and by symmetry for B seeing A's flag:
  - $read_A(flag[B] == false) \rightarrow write_B(flag[B] = true)$
  - $read_B(flag[A] == false) \rightarrow write_A(flag[A] = true)$

  These events form a cycle, which is a contradiction. □

Two other properties

Deadlock-free:

- When threads try to acquire the lock, at least one succeeds
- System as a whole makes progress

Starvation-free

- Every locking thread eventually returns
- Every thread makes progress

Two other properties

Deadlock-free:

- When threads try to acquire the lock, at least one succeeds
- System as a whole makes progress
- Does Lock_one enjoy deadlock freedom?

Starvation-free

- Every locking thread eventually returns
- Every thread makes progress
- Does Lock_one enjoy starvation freedom?

Deadlock case for Lock_one

```
flag_[0] = true;
                              flag_[1] = true;
while (flag_[1]) {}
                              while (flag_[0]) {}
```

(But sequentially it’s fine.)

Another attempt

```
class Lock_two : public Lock_base
{
    int waiting_;

public:
    virtual void lock() override
    {
        waiting_ = i();
        while (waiting_ == i()) {}
    }

    virtual void unlock() override {}
};
```

Lock_two claims

- Satisfies mutual exclusion
- Non deadlock-free
  - Sequential execution deadlocks
  - Concurrent execution does not

Peterson's algorithm

```
class Peterson_lock : public Lock_base
{
    bool flag_[2] = {};
    int waiting_;

public:
    virtual void lock() override
    {
        flag_[i()] = true;
        waiting_ = i();
        while (flag_[j()] && waiting_ == i()) {}
    }

    virtual void unlock() override
    { flag_[i()] = false; }
};
```

Peterson's lock properties

- Mutual exclusion
  - By contradiction…
- Deadlock freedom
  - Only one thread at a time can be waiting
- Starvation freedom
  - If A finishes and tries to re-enter while B is waiting, B gets in first

Filter algorithm for *n* threads

```
template <int N>
class Filter_lock : public Lock_base
{
    int level_[N] = {};
    int waiting_[N];

    bool exists_competition(int level)
    {
        for (auto k : thread::all_ids())
            if (k != i() && level_[k] >= level)
                return true;
        return false;
    }
    ⋮
}
```

```
template <int N>
class Filter_lock : public Lock_base
{
⋮
public:
    virtual void lock() override
    {
        for (int level = 1; level < N; ++level) {
            level_[i()]       = level;
            waiting_[level] = i();
            while (exists_competition(level) &&
                   waiting_[level] == i()) {}
        }
    }

    virtual void unlock() override
    { level_[i()] = 0; }
}
```

Filter lock properties

- Mutual exclusion
  - By induction, one thread gets stuck in each level...
- Deadlock freedom
  - Like Peterson—only one thread can wait per level
- Starvation freedom
  - Like Peterson—every thread advances if any does