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

```
class BasicLock {
public:
    virtual void lock()   =0;
    virtual void unlock() =0;
};
```

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_;
};
```

Using a lockRAAI-style

```
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_;
};
```

```
class Counter {
public:
    int get_and_inc()
    {
        auto guard = lock_.acquire();
        int old = count_;
        count_ = old + 1;
        return old;
        // ~Guard() unlocks lock_ here
    }

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

Base class for RAII-style lock

```
class GuardedLockBase : public BasicLock {
public:
    Guard acquire() { return Guard{*this}; }

    class Guard {
        BasicLock& lock_;

    public:
        Guard(BasicLock& lock) : lock_{lock} { lock_.lock(); }
        virtual ~Guard() { lock_.unlock(); }
    };
    ⋮
};
```

How to implement the lock?

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

Base class for RAII-style lock

```
class GuardedLockBase : public BasicLock {
    ⋮
    // 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 LockOne : public GuardedLockBase {
    bool flag_[2] = {};
public:
    virtual void lock() override
    {
        flag_[i()] = true;
        while (flag_[j()]) {}
    }

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

Theorem

LockOne satisfies mutual exclusion.

Theorem

LockOne satisfies mutual exclusion. Proof by contradiction:

- Assume CS_A overlaps CS_B

Theorem

LockOne 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:
  - $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{CS}_A$

Theorem

LockOne 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:
  - $write_B(flag[B] = true) \rightarrow read_B(flag[A] == false) \rightarrow CS_B$

Theorem

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

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

Theorem

LockOne 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:
  - $\text{write}_A(\text{flag}[A] = \text{true}) \to \text{read}_A(\text{flag}[B] == \text{false}) \to CS_A$

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

  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:
  - $\text{read}_A(\text{flag}[B] == \text{false}) \to \text{write}_B(\text{flag}[B] = \text{true})$
  - $\text{read}_B(\text{flag}[A] == \text{false}) \to \text{write}_A(\text{flag}[A] = \text{true})$

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

Two other properties

Deadlock-free:

- One ill-behaved thread does not prevent other threads from locking other locks
- System as a whole makes progress

Two other properties

Deadlock-free:

- One ill-behaved thread does not prevent other threads from locking other locks
- System as a whole makes progress

Starvation-free

- Every locking thread eventually returns
- Every thread makes progress

Two other properties

Deadlock-free:

- One ill-behaved thread does not prevent other threads from locking other locks
- System as a whole makes progress
- Does LockOne enjoy deadlock freedom?

Starvation-free

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

Deadlock case for LockOne

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

(But sequentially it's fine.)

Another attempt

```
class LockTwo : public GuardedLockBase {
  int waiting_;
public:
  virtual void lock() override
  {
    waiting_ = i();
    while (waiting_ == i()) {}
  }

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

LockTwo claims

- Satisfies mutual exclusion
- Not deadlock-free
  - Sequential execution deadlocks
  - Concurrent execution does not (Why?)

Peterson's algorithm

```
class PetersonLock : public GuardedLockBase {
    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 FilterLock : public GuardedLockBase {
    int level_[N] = {0};
    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 FilterLock : public GuardedLockBase {
    ⋮
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