Uniprocessor Lock Implementation

```
class Lock {
    Lock() {}
    int locked = 0;
    ThreadQueue q;
};

void Lock::lock() {
    intrDisable();
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
    intrEnable();
}
```

```
void Lock::unlock() {
    intrDisable();
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    intrEnable();
}
```

Uniprocessor Lock Implementation??

```
class Lock {
    Lock() {}
    int locked = 0;
    ThreadQueue q;
};

void Lock::lock() {
    intrDisable();
    if (!locked) {
        locked = 1;
        intrEnable();
    } else {
        q.add(currentThread);
        intrEnable();
        blockThread();
    }
}
```

```
void Lock::unlock() {
    intrDisable();
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    intrEnable();
}
```

Locks for Multi-Core, v1

```
class Lock {
    Lock() {}
    std::atomic<int> locked(0);
};

void Lock::lock() {
    while (locked.exchange(1)) {
        /* Do nothing */
    }
}
```

```
void Lock::unlock() {
    locked = 0;
}
```

Locks for Multi-Core, v2

```
class Lock {
    Lock() {}
    std::atomic<int> locked(0);
    ThreadQueue q;
};

void Lock::lock() {
    if (locked.exchange(1)) {
        q.add(currentThread);
        blockThread();
    }
}
```

```
void Lock::unlock() {
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

Locks for Multi-Core, v3

```
class Lock {
    Lock() {}
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = 0;
    } else {
        q.add(currentThread);
        spinlock = 0;
        blockThread();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
}
```

Locks for Multi-Core, v4

```
class Lock {
    Lock() {}
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = 0;
        reschedule();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
}
```

Locks for Multi-Core, v5

```
class Lock {
    Lock() {}
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> spinlock;
};

void Lock::lock() {
    intrDisable();
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = 0;
        reschedule();
    }
    intrEnable();
}
```

```
void Lock::unlock() {
    intrDisable();
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
    intrEnable();
}
```