ECE 353: Systems Software
Lecture 20

Locks Implementation

1.0.0

Jon Eyolfson
March 1, 2023

You Can Implement Locks in Software with Minimal Hardware

Your hardware requirements just have to ensure:

- Loads and stores are atomic
- Instructions execute in order

There are 2 main algorithms you could use:

Peterson's algorithm and Lamport's bakery algorithm

The problem is that they don't scale well, and processors execute out-of-order

Let's Assume a Magical Atomic Function — `compare_and_swap`

`compare_and_swap(int *p, int old, int new)` is atomic

- It returns the original value pointed to
- It only swaps if the original value equals `old`, and changes it to `new`

Let's give it another shot:

```
void init(int *l) {
  *l = 0;
}
void lock(int *l) {
  while (compare_and_swap(l, 0, 1));
}
void unlock(int *l) {
  *l = 0;
}
```

What We Implement is Essentially a Spinlock

Compare and swap is a common atomic hardware instruction

On x86 this is the `cmpxchg` instruction (compare and exchange)

However, it still has this “busy wait” problem

Consider a uniprocessor system, if you can’t get the lock, you should yield

Let the kernel schedule another process, that may free the lock

On a multiprocessor machine, you could try again

Let's Add a Yield

```
void lock(int *l) {
  while (compare_and_swap(l, 0, 1)) {
    thread_yield();
  }
}
```

Now we have a thundering herd problem

- Multiple threads may be waiting on the same lock

We have no control over who gets the lock next

- We need to be able to reason about it (FIFO is okay)

We Can Add a Wait Queue to the Lock

```
void lock(int *l) {
  while (compare_and_swap(l, 0, 1)) {
    // add myself to the lock wait queue
    thread_sleep();
  }
}
void unlock(int *l) {
  *l = 0;
  if (/* threads in wait queue */) {
    // wake up one thread
  }
}
```

There are 2 issues with this: 1) lost wakeup, and 2) the wrong thread gets the lock

Lost Wakeup Example

```
1  void lock(int *l) {
2    while (compare_and_swap(l, 0, 1)) {
3      // add myself to the wait queue
4      thread_sleep();
5    }
6  }
7  void unlock(int *l) {
8    *l = 0;
9    if (/* threads in wait queue */) {
10     // wake up one thread
11   }
12 }
```

Assume we have thread 1 (T1) and thread 2 (T2), thread 2 holds the lock

T1 runs line 2 and fails, swap to T2 that runs lines 10-12, T1 runs lines 3 -4

T1 will never get woken up!

Wrong Thread Getting the Lock Example

```
1  void lock(int *l) {
2    while (compare_and_swap(l, 0, 1)) {
3      // add myself to the wait queue
4      thread_sleep();
5    }
6  }
7  void unlock(int *l) {
8    *l = 0;
9    if (/* threads in wait queue */) {
10     // wake up one thread
11   }
12 }
```

Assume we have T1, T2, and T3. T2 holds the lock, T3 is in queue.

T2 runs line 9, swap to T1 which runs line 2 and succeeds

T1 just stole the lock from T3!

To Fix These Problems, We Can Use Two Variables (One to Guard)

```
typedef struct {
  int lock;
  int guard;
  queue_t *q;
} mutex_t;

void lock(mutex_t *m) {
  while (
    compare_and_swap(m->guard, 0, 1)
  );
  if (m->lock == 0) {
    m->lock = 1; // acquire mutex
    m->guard = 0;
  } else {
    enqueue(m->q, self);
    m->guard = 0;
    thread_sleep();
    // wakeup transfers the lock here
  }
}
```

```
void unlock(mutex_t *m) {
  while (
    compare_and_swap(m->guard, 0, 1)
  );
  if (queue_empty(m->q)) {
    // release lock, no one needs it
    m->lock = 0;
  }
  else {
    // direct transfer mutex
    // to next thread
    thread_wakeup(dequeue(m->q));
  }
  m->guard = 0;
}
```

There's STILL A Data Race

After a thread calls `lock`, it could get interrupted right before the `thread_sleep`

However, it's been added to the wait queue, so `thread_wakeup`
would try to wake up a thread that's not sleeping yet (we know it's about to)

We could simply retry the call to `thread_wakeup`
until the thread finally calls `thread_sleep`

Remember What Causes a Data Race

A data race is when two concurrent actions access the same variable and at least one of them is a **write**

We could have any many readers as we want

We don’t need a mutex as long as nothing writes at the same time

We need different lock modes for reading and writing

Read-Write Locks

With mutexes/spinlocks, you have to lock the data,
even for a read since you don't know if a write could happen

Reads can happen in parallel, as long as there's no write

Multiple threads can hold a read lock (`pthread_rwlock_rdlock`),
but only one thread may hold a write lock (`pthread_rwlock_wrlock`)
and will wait until the current readers are done

We Can Use A Guard To Keep Track of Readers

```
typedef struct {
  int nreader;
  lock_t guard;
  lock_t lock;
} rwlock_t;

void write_lock(rwlock_t *l) {
  lock(&l->lock);
}

void write_unlock(rwlock_t *l) {
  unlock(&l->lock);
}
```

```
void read_lock(rwlock_t *l) {
  lock(&l->guard);
  ++nreader;
  if (nreader == 1) { // first reader
    lock(&l->lock);
  }
  unlock(&l->guard);
}
void read_unlock(rwlock_t *l) {
  lock(&l->guard);
  --nreader;
  if (nreader == 0) { // last reader
    unlock(&l->lock);
  }
  unlock(&l->guard);
}
```

We Want Critical Sections to Protect Against Data Races

We should know what data races are, and how to prevent them:

- Mutex or spinlocks are the most straightforward locks
- We need hardware support to implement locks
- We need some kernel support for wake up notifications
- If we know we have a lot of readers, we should use a read-write lock