Lecture 20, Mar 5, 2024

Locks Implementation

- Locks can be implemented with minimal hardware, assuming atomic loads and stores and in-order execution
  - However these do not scale well and real processors often execute out-of-order
- Assume there exists an atomic hardware instruction "compare and swap" (e.g. `cmpxchg` on x86), which checks if a value is equal to something, and assigns something else to it if it is, while returning the old value, then the lock can be implemented with:

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

- The above is a *spinlock*
- The first problem is the busy wait – if a thread cannot acquire the lock, it should yield and not keep taking CPU time
  - We can add a yield inside the `while` loop
- The second problem is the *thundering herd* problem – if multiple threads are waiting on the lock at the same time, we don't know which one will get the lock when it becomes available
  - We want threads to get the lock in FIFO order to ensure fairness
  - We can add a wait queue to the lock and wake up the next thread on unlock

```
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
    }
}
```

- This has 2 issues: lost wakeup (a thread goes to sleep and never wakes up) and wrong thread getting the lock
  - If a context switch happens after a thread calls `compare_and_swap` to lock (and fails), but before it inserts itself to the wait queue, the thread calling unlock will not wake it up and this thread will sleep forever
  - If a context switch happens after a thread unlocks but before it wakes up the next thread in queue, and the thread switched to acquires the lock immediately, that thread will have the lock before the threads in queue

```c
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;
}
```

- To fix this, we can combine both a spin lock and a wait queue, using a *lock* and *guard*
  - The guard is a spin lock for `lock()` and `unlock()` and ensures while one thread is trying to lock, other threads cannot try to unlock or vice versa
  - The lock itself still has a yield and wait queue to ensure efficiency and fairness
- One final data race remains – in `lock()`, we unlock the guard after putting the current thread in queue, but before going to sleep
  - If a context switch happens after guard unlock but before sleep, then the thread attempting to unlock will be trying to wake up a thread that never went to sleep
  - However, this is easily detected, and we can simply just try again (keep trying to wake up until the thread sleeps)
- Data races only occur if a write is happening while threads are trying to read, so if we have many threads reading at the same time, we won't get a data race
  - We can have different locking modes for read and write, so multiple reads can happen at the same time; these are known as *read-write locks*
  - Multiple threads can hold a read lock, but only one may hold a write lock at a time
    * Attempting to acquire a write lock will wait for all read locks to be unlocked first
    * Attempting to acquire a read lock while another thread has the write lock will wait for the write lock to be unlocked
  - `pthread_rwlock_rdlock` and `pthread_rwlock_wrlock` implements this functionality

  - Read-write locks can be implemented using two mutexes as shown below

```
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);
}
```

- Note the above simplified implementation allows for starvation of write, if reads keep coming in (i.e. it's not fair for write)