```
CS 202, Spring 2015
Handout 5 (Class 6)

Implementation of spinlocks and mutexes

1. Here is a BROKEN spinlock implementation:

        struct Lock {
          int locked;
        }

        void acquire(Lock *lock) {
          while (1) {
            if (lock->locked == 0) { // A
              lock->locked = 1;      // B
              break;
            }
          }
        }

        void release (Lock *lock) {
          lock->locked = 0;
        }

        What’s the problem? Two acquire()s on the same lock on different
        CPUs might both execute line A, and then both execute B. Then
        both will think they have acquired the lock. Both will proceed.
        That doesn’t provide mutual exclusion.

```



```

2. Correct spinlock implementation

    Relies on atomic hardware instruction. For example, on the x86,
        doing
                "xchg addr, %eax"
        does the following:

        (i)   freeze all CPUs’ memory activity for address addr
        (ii)  temp = *addr
        (iii) *addr = %eax
        (iv)  %eax = temp
        (v)   un-freeze memory activity

    /* pseudocode */
    int xchg_val(addr, value) {
        %eax = value;
        xchg (*addr), %eax
    }

    /* bare-bones version of acquire */
    void acquire (Lock *lock) {
      pushcli();    /* what does this do? */
      while (1) {
        if (xchg_val(&lock->locked, 1) == 0)
          break;
      }
    }

    void release(Lock *lock){
       xchg_val(&lock->locked, 0);
       popcli();    /* what does this do? */
    }


    /* optimization in acquire; call xchg_val() less frequently */
    void acquire(Lock* lock) {
        pushcli();
        while (xchg_val(&lock->locked, 1) == 1) {
            while (lock->locked) ;
        }
    }

    The above is called a *spinlock* because acquire() spins. The
    bare-bones version is called a "test-and-set (TAS) spinlock"; the
    other is called a "test-and-test-and-set spinlock".

    The spinlock above is great for some things, not so great for
    others. The main problem is that it *busy waits*: it spins,
    chewing up CPU cycles. Sometimes this is what we want (e.g., if
    the cost of going to sleep is greater than the cost of spinning
    for a few cycles waiting for another thread or process to
    relinquish the spinlock). But sometimes this is not at all what we
    want (e.g., if the lock would be held for a while: in those
    cases, the CPU waiting for the lock would waste cycles spinning
    instead of running some other thread or process).

    NOTE: the spinlocks presented here can introduce performance issues
    when there is a lot of contention. (This happens even if the
    programmer is using spinlocks correctly.) The performance issues
    result from cross-talk among CPUs (which undermines caching and
    generates traffic on the memory bus). If we have time later, we will
    study a remediation of this issue (search the Web for "MCS locks").

    ANOTHER NOTE: In everyday application-level programming, spinlocks
    will not be something you use (use mutexes instead). But you should
    know what these are for technical literacy, and to see where the
    mutual exclusion is truly enforced on modern hardware.

```

3. Mutex implementation

The intent of a mutex is to avoid busy waiting: if the lock is not available, the locking thread is put to sleep, and tracked by a queue in the mutex.

```
    struct Mutex {
        bool is_held;           /* true if mutex held */
        thread_id owner;        /* thread holding mutex, if locked */
        thread_list waiters;    /* queue of thread TCBs */
        Lock wait_lock;         /* as in item 2, above */
    }
```

The implementation of mutex_acquire() and mutex_release() would be something like:

```
    void mutex_acquire(Mutex *m) {

        acquire(&m->wait_lock);   /* we spin to acquire wait_lock */

        while (m->is_held) {     /* someone else has the mutex */

            m->waiters.insert(current_thread)
            release(&m->wait_lock);

            /*
             * NOTE! Right here, mutex_release() could execute. To
             * avoid "losing the wakeup", we check whether we are
             * on the scheduler's ready list. If we are, we
             * shouldn't yield().
             */

            yield_if_we_are_not_ready();

            acquire(&m->wait_lock);   /* we spin again */
            m->waiters.remove(current_thread)

        }

        m->is_held = true;      /* we now hold the mutex */
        m->owner = self;

        release(&m->wait_lock);
    }

    void mutex_release(Mutex *m) {

        acquire(&m->wait_lock);    /* we spin to acquire wait_lock */

        m->is_held = false;
        m->owner = 0;

        /* tell scheduler to run a waiter */
        place_a_waiter_on_ready_list(m->waiters);

        release(&m->wait_lock);

    }
```