Implementation of spinlocks and mutexes

1. Here is a BROKEN spinlock implementation:

```
        struct Spinlock {
          int locked;
        }

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

        void release (Spinlock *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 x86−64, doing

"xchg addr, %rax"

does the following:

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

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

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

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


    /* optimization in acquire; call xchg_val() less frequently */
    void acquire(Spinlock* 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. The next page has an implementation.



```c
1  #include <sys/queue.h>
2
3  typedef struct thread {
4      // ... Entries elided.
5      STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
6  } thread_t;
7
8  struct Mutex {
9      // Current owner, or 0 when mutex is not held.
10     thread_t *owner;
11
12     // List of threads waiting on mutex
13     STAILQ(thread_t) waiters;
14
15     // A lock protecting the internals of the mutex.
16     Spinlock splock;   // as in item 1, above
17 };
18
19 void mutex_acquire(struct Mutex *m) {
20
21     acquire(&m->splock);
22
23     // Check if the mutex is held; if not, current thread gets mutex and returns
24     if (m->owner == 0) {
25         m->owner = id_of_this_thread;
26         release(&m->splock);
27     } else {
28         // Add thread to waiters.
29         STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
30
31         // Tell the scheduler to add current thread to the list
32         // of blocked threads. The scheduler needs to be careful
33         // when a corresponding sched_wakeup call is executed to
34         // make sure that it treats running threads correctly.
35         sched_mark_blocked(&id_of_this_thread);
36
37         // Unlock spinlock.
38         release(&m->splock);
39
40         // Stop executing until woken.
41         sched_swtch();
42
43         // When we get to this line, we are guaranteed to hold the mutex. This
44         // is because we can get here only if context-switched-TO, which itself
45         // can happen only if this thread is removed from the waiting queue,
46         // marked "unblocked", and set to be the owner (in mutex_release()
47         // below). However, we might have held the mutex in lines 39-42
48         // (if we were context-switched out after the spinlock release(),
49         // followed by being run as a result of another thread's release of the
50         // mutex). But if that happens, it just means that we are
51         // context-switched out an "extra" time before proceeding.
52     }
53 }
54
55 void mutex_release(struct Mutex *m) {
56     // Acquire the spinlock in order to make changes.
57     acquire(&m->splock);
58
59     // Assert that the current thread actually owns the mutex
60     assert(m->owner == id_of_this_thread);
61
62     // Check if anyone is waiting.
63     m->owner = STAILQ_GET_HEAD(&m->waiters);
64
65     // If so, wake them up.
66     if (m->owner) {
67         sched_wakeone(&m->owner);
68         STAILQ_REMOVE_HEAD(&m->waiters, qlink);
69     }
70
71     // Release the internal spinlock
72     release(&m->splock);
73 }
```

```
Deadlock examples

1. Simple deadlock example

    T1:
        acquire(mutexA);
        acquire(mutexB);

        // do some stuff

        release(mutexB);
        release(mutexA);

    T2:
        acquire(mutexB);
        acquire(mutexA);

        // do some stuff

        release(mutexA);
        release(mutexB);
```



```
2. More subtle deadlock example

    Let M be a monitor (shared object with methods protected by mutex)
    Let N be another monitor

    class M {
        private:
            Mutex mutex_m;

            // instance of monitor N
            N another_monitor;

            // Assumption: no other objects in the system hold a pointer
            // to our "another_monitor"

        public:
            M();
            ~M();
            void methodA();
            void methodB();
    };

    class N {
        private:
            Mutex mutex_n;
            Cond cond_n;
            int navailable;

        public:
            N();
            ~N();
            void* alloc(int nwanted);
            void  free(void*);
    }

    int
    N::alloc(int nwanted) {
        acquire(&mutex_n);
        while (navailable < nwanted) {
            wait(&cond_n, &mutex_n);
        }

        // peel off the memory

        navailable -= nwanted;
        release(&mutex_n);
    }

    void
    N::free(void* returning_mem) {

        acquire(&mutex_n);

        // put the memory back

        navailable += returning_mem;

        broadcast(&cond_n, &mutex_n);

        release(&mutex_n);
    }
```

```
    void
    M::methodA() {

        acquire(&mutex_m);

        void* new_mem = another_monitor.alloc(int nbytes);

        // do a bunch of stuff using this nice
        // chunk of memory n allocated for us

        release(&mutex_m);
    }

    void
    M::methodB() {

        acquire(&mutex_m);

        // do a bunch of stuff

        another_monitor.free(some_pointer);

        release(&mutex_m);
    }

    QUESTION: What's the problem?
```