```
Handout for CS 439
Class 9
12 February 2013

1. Implementing threads

    Per−thread state in thread control block:

        typedef struct tcb {
            unsigned long esp;      /* Stack pointer of thread */
            char *t_stack;          /* Bottom of thread's stack */
            /* ... */
            };

    Machine−dependent thread−switch function:

        void swtch(tcb *current, tcb *next);

    Machine−dependent thread initialization function:

        void thread_init(tcb *t, void (*fn) (void *), void *arg);

    Implementation of swtch(current, next):

        pushl %ebp; movl %esp, %ebp             # Save frame pointer
        pushl %ebx; pushl %esi; pushl %edi      # Save callee−saved regs

        movl 8(%ebp),%edx                       # %edx = current
        movl 12(%ebp),%eax                      # %eax = next
        movl %esp,(%edx)                        # %edx−>esp = %esp
        movl (%eax),%esp                        # %esp = %eax−>esp

        popl %edi; popl %esi; popl %ebx # Restore callee saved regs
        popl %ebp                               # Restore frame pointer
        ret                                     # Resume execution


    [thanks to David Mazieres]
```



```

2. How can we implement locks, acquire(), and release()?

    2a. Here is A BADLY BROKEN implementation:

        struct Lock {
          int locked;
        }

        void [BROKEN] acquire(Lock *lock) {
          while (1) {
            if (lock−>locked == 0) { // C
              lock−>locked = 1;    // D
              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 C, and then both execute D.  Then
        both will think they have acquired the lock.  This is the same
        kind of race we were trying to eliminate to begin with. But we
        have made a little progress: now we only need a way to prevent
        interleaving in one place (acquire()), not for many arbitrary
        complex sequences of code.
```

```
    2b. Here's a way that is correct but that is appropriate only in
        some circumstances:

        Use an atomic instruction on the CPU. 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
        }

        struct Lock {
          int locked;
        }

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

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

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

        The above is called a *spinlock* because acquire() spins.

        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).
```



```
    2c. Here's an object that does not involve busy waiting. Note: the
    "threads" here can be user-level threads, kernel threads, or
    threads-inside-kernel.  The concept is the same in all cases.

        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 2b */
        }

        The implementation of acquire() and 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);
                schedule();   /* run a thread that is on the ready list */
                acquire(&m->wait_lock);   /* we spin again */
            }
            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;
            wake_up_a_waiter(m->waiters); /* select and run a waiter */
            release(&m->wait_lock);

        }
        [Please let me (MW) know if you see bugs in the above.]

3. NOTE: the above mutex does the right thing only if there are some
constraints on the order in which the CPU carries out memory reads and
writes. For example, if operations _after_ mutex_acquire() in program
order appear to another processor to happen in the opposite order, then
they would not be protected by the lock, and the program would be incorrect.

How do we get the required guarantee? By ensuring that neither the
compiler nor the processor reorders instructions with respect to the
acquire(). To prevent reordering by the compiler, the programmer can
mark the asm instructions as volatile. To prevent the processor from
reordering, one must use special assembly instructions. For instance,
fences (the "LFENCE", "SFENCE", and "MFENCE" instructions) tell the CPU
not to re-order memory operations past the fences. Also, the processor
will not reorder instructions with respect to xchg() (and a few others).

Moral of the above paragraphs: if you're implementing a concurrency
primitive, read the processor's documentation about how loads and stores
get sequenced, and how to enforce that the compiler *and* the processor
follow program order.

4. Terminology

    To avoid confusion, we will use the following terminology in this
    course (you will hear other terminology elsewhere):

    --A "lock" is an abstract object that provides mutual exclusion

    --A "spinlock" is a lock that works by busy waiting, as in 2b

    --A "mutex" is a lock that works by having a "waiting" queue and
    then protecting that waiting queue with atomic hardware
    instructions, as in 2c. The most natural way to "use the hardware"
    is with a spinlock, but there are others, such as turning off
    interrupts, which works if we're on a single CPU machine.
```