```
Handout for CS 372H
Class 9
15 February 2011

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. Example to illustrate interleavings: say that thread A executes f()
and thread B executes g(). (Here, we are using the term "thread"
abstractly. This example applies to any of the approaches that fall
under the word "thread".)

    a.

        int x;

        f() { x = 1; }

        g() { x = 2; }

        What are possible values of x after A has executed f() and B has
        executed g()?

    b.
        int y = 12;

        f() { x = y + 1; }
        g() { y = y * 2; }

        What are the possible values of x?

    c.
        int x = 0;
        f() { x = x + 1; }
        g() { x = x + 2; }

        What are the possible values of x?

3. Linked list example

    struct List_elem {
        int data;
        struct List_elem* next;
    };

    List_elem* head = 0;

    insert(int data) {
        List_elem* l = new List_elem;
        l−>data = data;
        l−>next = head;
        head = l;
    }

    What happens if two threads execute insert() at once and we get the
    following interleaving?

    thread 1: l−>next = head
    thread 2: l−>next = head
    thread 2: head = l;
    thread 1: head = l;
```

```


4. Producer/consumer example:

    /*
    "buffer" stores BUFFER_SIZE items
    "count" is number of used slots. a variable that lives in memory
    "out" is next empty buffer slot to fill (if any)
    "in" is oldest filled slot to consume (if any)
    */

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();
             while (count == BUFFER_SIZE)
                 ; // do nothing
             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
         }
     }

     void consumer (void *ignored) {
         for (;;) {
             while (count == 0)
                ; // do nothing
             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count−−;
             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

    /*
       what count++ probably compiles to:
        reg1 <−− count      # load
        reg1 <−− reg1 + 1   # increment register
        count <−− reg1      # store

       what count−− could compile to:
        reg2 <−− count      # load
        reg2 <−− reg2 − 1   # decrement register
        count <−− reg2      # store
    */

    What happens if we get the following interleaving?

        reg1 <−− count
        reg1 <−− reg1 + 1
        reg2 <−− count
        reg2 <−− reg2 − 1
        count <−− reg1
        count <−− reg2

```



```
5. Protecting the linked list......

        Lock list_lock;

        insert(int data) {
            List_elem* l = new List_elem;
            l−>data = data;

            acquire(&list_lock);

            l−>next = head;         // A
            head = l;               // B

            release(&list_lock);
        }

```

6. How can we implement list_lock, acquire(), and release()?

6a. 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 in insert(). 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.

6b. 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() waits in a busy loop.

Unfortunately, insert() with these locks is only correct if each CPU carries out memory reads and writes in program order. For example, if the CPU were to execute insert() out of order so that it did the read at A before the acquire(), then insert() would be incorrect even with locks. Many modern processors execute memory operations out of order to increase performance! So we may have to use special instructions ("lock", "LFENCE", "SFENCE", "MFENCE") to tell the CPU not to re-order memory operations past acquire()s and release()s. The compiler may also generate instructions in orders that don't correspond to the order of the source code lines, so we have to worry about that too. One way around this is to make the asm instructions volatile.

Moral of the above paragraph: 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.

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).

6c. Here's an object that does not involve busy waiting; it can work as the list_lock mentioned in #5, above. 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 6b */
    }
```

Now, instead of acquire(&list_lock) and release(&list_lock) as in #5, we'd write, mutex_acquire(&list_mutex) and mutex_release(&list_mutex). The implementation of the latter two would be something like this:

```
    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.]

7. 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 6b

−−A "mutex" is a lock that works by having a "waiting" queue and then protecting that waiting queue with atomic hardware instructions, as in 6c. 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.