```
Handout for CS 372H
Class 9
16 February 2010

1. 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".)

    1a.

        int x;

        f() { x = 1; }

        g() { x = 2; }

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

    1b.
        int y = 12;

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

        What are the possible values of x?

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

        What are the possible values of x?

2. 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;
```



```
3. 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
```

```
4. 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);
        }

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

    5a. 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.
```



```
    5b. 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;
        }

        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? */
        }

        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).
```

5c. Here’s a lock 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_locked;         /* true if locked */
        thread_id owner;        /* thread holding lock, if locked */
        thread_list waiters;    /* queue of thread TCBs */
        spinlock wait_lock;     /* exactly as in 5b */
    }
```

Now, mutex.acquire() looks something like this:

```
        wait_lock.acquire()
        while (is_locked) {
            waiters.insert(current_thread)
            wait_lock.release()
            schedule();   /* run a thread that is on the ready list */
            wait_lock.acquire();
        }
        is_locked = 1;
        owner = self;
        wait_lock.release();
```

And mutex.release() looks something like this:

```
        wait_lock.acquire()
        is_locked = 0;
        owner = 0;
        wake_up_a_waiter(); /* selects a waiter and runs it */
        wait_lock.release()
```

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

6. 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 5b

−−A "mutex" is a lock that works by having a "waiting" queue and then protecting that waiting queue with atomic hardware instructions, as in 5c. 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.