```
Handout for CS 372H
Class 10
17 February 2011

1. 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);
        }

2. How can we implement list_lock, 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 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.
```



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

        Unfortunately, insert() with these locks is correct only 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 (chapter 8 in current
        architecture manual).

        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; it can work as the list_lock mentioned in #1, 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 2b */
    }
```

Now, instead of acquire(&list_lock) and release(&list_lock) as in #1, 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.]

3. 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.

4. Producer/consumer example [also known as bounded buffer]

4a. buggy implementation (from last time)

```
 /*
 "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);
      }
  }
```

−−Review: what’s the problem?
−−Answer: count++ and count−− might compile to, respectively:

```
    reg1 <−− count      # load
    reg1 <−− reg1 + 1   # increment register
    count <−− reg1      # store

    reg2 <−− count      # load
    reg2 <−− reg2 − 1   # decrement register
    count <−− reg2      # store
```

−−Review: why not use instructions like "addl $0x1, _count"?
−−Answer: not atomic if there are multiple CPUs.

−−Review: so why not use "LOCK addl $0x1, _count"?
−−Answer: we could do that here, but LOCK won’t save us every time

−−Review: so use general−purpose approach to protecting critical sections: locks (or mutexes).

```
  4b. Producer/consumer [bounded buffer] using mutexes

    Mutex mutex;

    void producer (void *ignored) {
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            nextProduced = means_of_production();

            acquire(&mutex);
            while (count == BUFFER_SIZE) {
               release(&mutex);
               yield(); /* or schedule() */
               acquire(&mutex);
            }

            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
            release(&mutex);
        }
    }

    void consumer (void *ignored) {
        for (;;) {

            acquire(&mutex);
            while (count == 0) {
               release(&mutex);
               yield(); /* or schedule() */
               acquire(&mutex);
            }

            nextConsumed = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            count−−;
            release(&mutex);

            /* next line abstractly consumes the item */
            consume_item(nextConsumed);
        }
    }
```



```
    4c. Producer/consumer [bounded buffer] using mutexes and condition
    variables

         Mutex mutex;
         Cond nonempty;
         Cond nonfull;

         void producer (void *ignored) {
             for (;;) {
                 /* next line produces an item and puts it in nextProduced */
                 nextProduced = means_of_production();

                 acquire(&mutex);
                 while (count == BUFFER_SIZE)
                    cond_wait(&nonfull, &mutex);

                 buffer [in] = nextProduced;
                 in = (in + 1) % BUFFER_SIZE;
                 count++;
                 cond_signal(&nonempty);
                 release(&mutex);
             }
         }

         void consumer (void *ignored) {
             for (;;) {

                 acquire(&mutex);
                 while (count == 0)
                    cond_wait(&nonempty, &mutex);

                 nextConsumed = buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count−−;
                 cond_signal(&nonfull);
                 release(&mutex);

                 /* next line abstractly consumes the item */
                 consume_item(nextConsumed);
             }
         }


        Question: why does cond_wait need to both release the mutex and
        sleep? Why not:

            while (count == BUFFER_SIZE)  {
                release(&mutex);
                cond_wait(&nonfull);
                acquire(&mutex);
            }
```

4d. Producer/consumer [bounded buffer] with semaphores

```
    Semaphore mutex(1);           /* mutex initialized to 1 */
    Semaphore empty(BUFFER_SIZE);  /* start with BUFFER_SIZE empty slots */
    Semaphore full(0);            /* 0 full slots */

    void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();

            /*
             * next line diminishes the count of empty slots and
             * waits if there are no empty slots
             */
             sem_down(&empty);
             sem_down(&mutex);  /* get exclusive access */

             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;

             sem_up(&mutex);
             sem_up(&full);   /* we just increased the # of full slots */
         }
     }

     void consumer (void *ignored) {
         for (;;) {

             /*
              * next line diminishes the count of full slots and
              * waits if there are no full slots
              */
             sem_down(&full);
             sem_down(&mutex);

             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;

             sem_up(&mutex);
             sem_up(&empty);   /* one further empty slot */

             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }
```

Semaphores *can* (not always) lead to elegant solutions (notice that the code above is fewer lines than 1c) but they are much harder to use.

The fundamental issue is that semaphores make implicit (counts, conditions, etc.) what is probably best left explicit. Moreover, they *also* implement mutual exclusion.

For this reason, you should not use semaphores. This example is here mainly for completeness and so you know what a semaphore is. But do not code with them. Solutions that use semaphores in this course will receive no credit.

5. Example of a monitor: MyBuffer

```
    // This is pseudocode that is inspired by C++.
    // Don't take it literally.

    class MyBuffer {
      public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item = Dequeue();
      private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
    }

    void
    MyBuffer::MyBuffer()
    {
        in = out = count = 0;
        mutex = new Mutex;
        nonempty = new Cond;
        nonfull = new Cond;
    }

    void
    MyBuffer::Enqueue(Item item)
    {
        mutex.acquire();
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        ++count;
        cond_signal(&nonempty, &mutex);
        mutex.release();
    }

    Item
    MyBuffer::Dequeue()
    {
        mutex.acquire();
        while (count == 0)
            cond_wait(&nonempty, &mutex);

        Item ret = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        --count;
        cond_signal(&nonfull, &mutex);
        mutex.release();
        return ret;
    }
```

```
int main(int, char**)
{
    MyBuffer buf;
    int dummy;
    tid1 = thread_create(producer, &buf);
    tid2 = thread_create(consumer, &buf);
    thread_join(tid1);

    // never reach this point
    return −1;
}

void producer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        Item nextProduced = means_of_production();
        sharedbuf−>Enqueue(nextProduced);
    }
}

void consumer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextConsumed = sharedbuf−>Dequeue();

        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}

Key point: *Threads* (the producer and consumer) are separate from
*shared object* (MyBuffer). The synchronization happens in the
shared object.
```