```
Handout for CS 372H
Class 11
22 February 2011

1. 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.

```

```
2. Readers/writers

    state variables:
        AR = 0;  // # active readers
        AW = 0;  // # active writers
        WR = 0;  // # waiting readers
        WW = 0;  // # waiting writers

        Condition okToRead = NIL;
        Condition okToWrite = NIL;
        Mutex mutex = FREE;

    Database::read() {
        startRead();  // first, check self into the system
        Access Data
        doneRead();   // check self out of system
    }

    Database::startRead() {
        acquire(&mutex);
        while((AW + WW) > 0){
            WR++;
            wait(&okToRead, &mutex);
            WR−−;
        }
        AR++;
        release(&mutex);
    }

    Database::doneRead() {
        acquire(&mutex);
        AR−−;
        if (AR == 0 && WW > 0) { // if no other readers still
          signal(&okToWrite, &mutex);   // active, wake up writer
        }
        release(&mutex);
    }

    Database::write(){  // symmetrical
        startWrite();  // check in
        Access Data
        doneWrite();  // check out
    }

    Database::startWrite() {
        acquire(&mutex);
        while ((AW + AR) > 0) { // check if safe to write.
                                // if any readers or writers, wait
            WW++;
            wait(&okToWrite, &mutex);
            WW−−;
        }
        AW++;
        release(&mutex);
    }

    Database::doneWrite() {
        acquire(&mutex);
        AW−−;
        if (WW > 0) {
            signal(&okToWrite, &mutex); // give priority to writers
        } else if (WR > 0) {
            broadcast(&okToRead, &mutex);
        }
        release(&mutex);
    }

    NOTE: what is the starvation problem here?
```



```
3. Shared locks

    struct sharedlock {
      int i;
      Mutex mutex;
      Cond c;
    };

    void AcquireExclusive (sharedlock *sl) {
      acquire(&sl−>mutex);
      while (sl−>i) {
        wait (&sl−>c, &sl−>mutex);
      }
      sl−>i = −1;
      release(&sl−>mutex);
    }

    void AcquireShared (sharedlock *sl) {
      acquire(&sl−>mutex);
      while (sl−>i < 0) {
        wait (&sl−>c, &sl−>mutex);
      }
      sl−>i++;
      release(&sl−>mutex);
    }

    void ReleaseShared (sharedlock *sl) {
      acquire(&sl−>mutex);
      if (!−−sl−>i)
        signal (&sl−>c, &sl−>mutex);
      release(&sl−>mutex);
    }

    void ReleaseExclusive (sharedlock *sl) {
      acquire(&sl−>mutex);
      sl−>i = 0;
      broadcast (&sl−>c, &sl−>mutex);
      release(&sl−>mutex);
    }

    QUESTIONS:
    A. There is a starvation problem here. What is it? (Readers can keep
       writers out if there is a steady stream of readers.)
    B. How could you use these shared locks to write a cleaner version
       of the code in item 2., above? (Though note that the starvation
       properties would be different.)
```

```
4. 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);
```



```
5. 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);
```

```
310     }
311
312     void
313     M::methodB() {
314
315         acquire(&mutex_m);
316
317         // do a bunch of stuff
318
319         another_monitor.free(some_pointer);
320
321         release(&mutex_m);
322     }
323
324     QUESTION: What's the problem?
325
```