```
Handout for CS 439
Class 5
31 January 2013

1. Producer/consumer [bounded buffer]

    a. Producer/consumer with mutexes (last time)

    b. Producer/consumer [bounded buffer] with 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, &mutex);
                 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, &mutex);
                 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);
            }
```



```
    1c.  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.
```

```
2. 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);

        // never reach this point
        thread_join(tid1);
        thread_join(tid2);
        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.

3. 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?
```



```
4. 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 5., above? (Though note that the starvation
       properties would be different.)
```

```
5. 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);
```