```
Handout for CS 439
Class 4
29 January 2013

0. Recall race condition examples from last time:

    --the threads calling f()/g()

    --the threads simultaneously enqueuing to the head of a linked list

1. 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
```



```
2. Some other examples. What is the point of these?

    [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
    66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]

    a. Can both "critical sections" run?

        int flag1 = 0, flag2 = 0;

        int main () {
            tid id = thread_create (p1, NULL);
            p2 (); thread_join (id);
        }

        void p1 (void *ignored) {
            flag1 = 1;
            if (!flag2) {
                critical_section_1 ();
            }
        }

        void p2 (void *ignored) {
            flag2 = 1;
            if (!flag1) {
                critical_section_2 ();
            }
        }

    b. Can use() be called with value 0, if p2 and p1 run concurrently?

        int data = 0, ready = 0;

        void p1 () {
            data = 2000;
            ready = 1;
        }
        int p2 () {
            while (!ready) {}
            use(data);
        }

    c. Can use() be called with value 0?

        int a = 0, b = 0;

        void p1 (void *ignored) { a = 1; }

        void p2 (void *ignored) {
          if (a == 1)
            b = 1;
        }

        void p3 (void *ignored) {
          if (b == 1)
            use (a);
        }
```

```
3. Protecting the linked list......

        Mutex list_mutex;

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

            acquire(&list_mutex);

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

            release(&list_mutex);
        }

```



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

   4a. Producer/consumer [bounded buffer] with 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);
         }
     }

```

```
    4b. 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);
            }
```



```
    4c.  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);

        // 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.