```
Week 6.b

1. Producer/consumer example:

    /*
        "buffer" stores BUFFER_SIZE items
        "count" is number of used slots. a variable that lives in memory
        "in" is next empty buffer slot to fill (if any)
        "out" 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. Producer/consumer revisited [also known as bounded buffer]

2a. 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);
      }
    }

```

```
103    2b. Producer/consumer [bounded buffer] with mutexes and condition variables
104
105        Mutex mutex;
106        Cond nonempty;
107        Cond nonfull;
108
109        void producer (void *ignored) {
110          for (;;) {
111            /* next line produces an item and puts it in nextProduced */
112            nextProduced = means_of_production();
113
114            acquire(&mutex);
115            while (count == BUFFER_SIZE)
116              cond_wait(&nonfull, &mutex);
117
118            buffer [in] = nextProduced;
119            in = (in + 1) % BUFFER_SIZE;
120            count++;
121            cond_signal(&nonempty, &mutex);
122            release(&mutex);
123          }
124        }
125
126        void consumer (void *ignored) {
127          for (;;) {
128
129            acquire(&mutex);
130            while (count == 0)
131              cond_wait(&nonempty, &mutex);
132
133            nextConsumed = buffer[out];
134            out = (out + 1) % BUFFER_SIZE;
135            count--;
136            cond_signal(&nonfull, &mutex);
137            release(&mutex);
138
139            /* next line abstractly consumes the item */
140            consume_item(nextConsumed);
141          }
142        }
143
144
145    Question: why does cond_wait need to both release the mutex and
146    sleep? Why not:
147
148        while (count == BUFFER_SIZE) {
149          release(&mutex);
150          cond_wait(&nonfull);
151          acquire(&mutex);
152        }
153
154
```