Handout 3 gave examples of race conditions. The following panels demonstrate the use of concurrency primitives (mutexes, etc.). We are using concurrency primitives to eliminate race conditions (see items 1 and 2a) and improve scheduling (see item 2b).

1. 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;
   head = l;
   release(&list_mutex);
   }
   ```

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);
   }
   ```
2b. 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);
    }

2c. 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();
        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 (;;) {
        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 2b) 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.