1. Example to illustrate interleavings: say that thread A executes f() and thread B executes g(). (Here, we are using the term "thread" abstractly. This example applies to any of the approaches that fall under the word "thread".)

a. [this is pseudocode]

```c
int x;
int main(int argc, char** argv) {
    tid tid1 = thread_create(f, NULL);
    tid tid2 = thread_create(g, NULL);
    thread_join(tid1);
    thread_join(tid2);
    printf("%d\n", x);
}
void f() {
    x = 1;
    thread_exit();
}
void g() {
    x = 2;
    thread_exit();
}
```

What are possible values of x after A has executed f() and B has executed g()? In other words, what are possible outputs of the program above?

b. Same question as above, but f() and g() are now defined as follows:

```c
int y = 12;
void f() { x = y + 1; }
void g() { y = y * 2; }
```

What are the possible values of x?

c. Same question as above, but f() and g() are now defined as follows:

```c
int x = 0;
void f() { x = x + 1; }
void g() { x = x + 2; }
```

What are the possible values of x?

2. Linked list example

```c
struct List_elem {
    int data;
    struct List_elem* next;
};
List_elem* head = 0;
void insert(int data) {
    List_elem* l = new List_elem;
    l->data = data;
    l->next = head;
    head = l;
}
```

What happens if two threads execute insert() at once and we get the following interleaving?

```c
thread 1: l->next = head
thread 2: l->next = head
thread 2: head = l;
thread 1: head = l;
```
3. Producer/consumer example:

```c
/*
 * "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;
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

4. Some other examples. What is the point of these?

```

a. Can both "critical sections" run?
```
The previous handout gave examples of race conditions. This handout demonstrates 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......

```c
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

```c
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:

```c
while (count == BUFFER_SIZE) { cond_wait(&nonfull); }
```

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.

2c. Producer/consumer [bounded buffer] with semaphores

```c
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 */
    sem_down(&empty);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    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;
    count--;
    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.