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").

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

What are possible values of x after A has executed f() and B has executed g()?

b. int y = 12;
f() { x = y + 1; }
g() { y = y * 2; }
What are the possible values of x?

c. int x = 0;
f() { x = x + 1; }
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;
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
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++;
    }
}
```

```c
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?

```c
reg1 ←− count
reg1 ←− reg1 + 1
reg2 ←− count
reg2 ←− reg2 - 1
```
115 4. Some other examples. What is the point of these?
116
118
119 a. Can both *critical sections* run?
120
121 int flag1 = 0, flag2 = 0;
122
123 int main () {
124      tid id = thread_create (p1, NULL);
125      p2 (); thread_join (id);
126 }
127
128 void p1 (void *ignored) {
129      flag1 = 1;
130      if (!flag2) {
131          critical_section_1 ();
132      }  
133 }
134
135 void p2 (void *ignored) {
136      flag2 = 1;
137      if (!flag1) {
138          critical_section_2 ();
139      }
140    }
141
142 b. Can use() be called with value 0, if p2 and p1 run concurrently?
143
144 int data = 0, ready = 0;
145
146 void p1 () {
147      data = 2000;
148      ready = 1;
149    }
150
151 void p2 () {  
152      while (!ready) {}  
153      use(data);
154    }
155
156 c. Can use() be called with value 0?
157
158 int a = 0, b = 0;
159
160 void p1 (void *ignored) {  
161      a = 1;  
162    }
163
164 void p2 (void *ignored) {
165      if (a == 1)  
166          b = 1;
167    }
168
169 void p3 (void *ignored) {  
170      if (b == 1)  
171          use (a);
172    }
173
174 d. /* keyword *register* tells compiler to place the variable in a  
175     register, not on the stack. So f, g are local to each thread. */
176
177 int flag1 = 0, flag2 = 0;
178
179 int p1 (void *ignored) {  
180    register int f, g;  
181    flag1 = 1;  
182    f = flag1;
183    g = flag2;
184    return 2*f + g;  
185  }
186
187 int p2 (void *ignored) {  
188    register int f, g;  
189    flag2 = 1;  
190    f = flag2;
191    g = flag1;
192    return 2*f + g;  
193  }
194
195 5. Protecting the linked list......
196
197 Lock list_lock;
198
199 insert(int data) {  
200    List_elem* l = new List_elem;
201    l->data = data;
202    acquire(&list_lock);
203    l->next = head;  
204    head = l;
205    release(&list_lock);
206 }
207
208 6. How can we implement list_lock, acquire(), and release()?
209
210 6a. Here is A BADLY BROKEN implementation:
211
212 struct Lock {  
213    int locked;
214  }
215
216 void [BROKEN] acquire(Lock *lock) {  
217    while (1) {  
218      if (lock->locked == 0) {  
219        lock->locked = 1;  
220        break;
221      }
222    }
223  }
224
225 void release (Lock *lock) {  
226    lock->locked = 0;
227  }
228
229 What’s the problem? Two acquire()s on the same lock on different  
230 CPUs might both execute line C, and then both execute D. Then  
231 both will think they have acquired the lock. This is the same  
232 kind of race that we were trying to eliminate in insert(). But  
233 we have made a little progress: now we only need a way to  
234 prevent interleaving in one place (acquire()), not for many  
235 arbitrary complex sequences of code.
6b. Here’s a way that is correct but only sometimes appropriate:

```c
void mutex_acquire(Mutex *m) {
  acquire(&m->wait_lock);   /* we spin to acquire wait_lock */
  while (m->is_held) {     /* someone else has the mutex */
    release(&m->wait_lock);
    schedule();   /* run a thread that is on the ready list */
    acquire(&m->wait_lock);   /* we spin again */
  }
  m->is_held = true;     /* we now hold the mutex */
  m->owner = self;
  release(&m->wait_lock);
}
```

```c
void mutex_release(Mutex *m) {
  acquire(&m->wait_lock);    /* we spin to acquire wait_lock */
  m->is_held = false;
  m->owner = 0;
  wake_up_a_waiter(m->waiters); /* select and run a waiter */
  release(&m->wait_lock);
}
```

)xchg addr, %eax

```
7. NOTE: Unfortunately, insert() with these locks is correct only if there are some constraints on the order in which the CPU carries out memory reads and writes. For example, if insert() were executed so that the read at A appeared to another processor (and to memory) to be executed before the acquire(), then insert() would be incorrect even with locks.

How do we get the required guarantee? Answer: by ensuring that neither the programmer nor the processor reorders instructions with respect to the acquire().

8. Terminology

To avoid confusion, we will use the following terminology in this course (you will hear other terminology elsewhere):

- A "lock" is an abstract object that provides mutual exclusion
- A "spinlock" is a lock that works by busy waiting, as in 6b
- A "mutex" is a lock that works by having a "waiting" queue and then protecting that waiting queue with atomic hardware instructions, as in 6c. The most natural way to "use the hardware" is with a spinlock, but there are others, such as turning off interrupts, which works if we're on a single CPU machine.

6c. Here’s an object that does not involve busy waiting; it can work as the list_lock mentioned above. Note: the "threads" here can be user-level threads, kernel threads, or threads-inside-kernel. The concept is the same in all cases.

```c
struct Mutex {
  bool is_held;           /* true if mutex held */
  thread_id owner;     /* thread holding mutex, if locked */
  thread_list waiters;    /* queue of thread TCBs */
  Lock wait_lock;     /* as in 6b */
}
```

Now, instead of acquire(list_lock) and release(list_lock) as above, we'd write, mutex_acquire(list_mutex) and mutex_release(list_mutex). The implementation of the latter two would be something like this:
9. Producer/consumer example [also known as bounded buffer]

9a. buggy implementation

```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;
    count--;
    /* next line abstractly consumes the item */
    consume_item(nextConsumed);
  }
}

--Review: what's the problem?
--Answer: count++ and count-- might compile to, respectively:

reg1 <−− count      # load
reg1 <−− reg1 + 1   # increment register
count <−− reg1      # store
reg2 <−− count      # load
reg2 <−− reg2 − 1   # decrement register
count <−− reg2      # store

--Review: why not use instructions like "addl $0x1, _count"?
--Answer: not atomic if there are multiple CPUs.

--Review: so why not use "LOCK addl $0x1, _count"?
--Answer: we could do that here, but LOCK won't save us every time

--Review: so use general-purpose approach to protecting
critical sections: locks (or mutexes).
```

9b. Producer/consumer [bounded buffer] using mutexes

```c
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 (;;) {
    while (count == 0) {
      release(&mutex);
      yield(); /* or schedule() */
      acquire(&mutex);
    }
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* next line abstractly consumes the item */
    consume_item(nextConsumed);
    release(&mutex);
  }
}
```
9c. Producer/consumer [bounded buffer] using 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 (;;) {
        /* 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) {
    release(&mutex);
    cond_wait(&nonfull);
    acquire(&mutex);
}
```

9d. 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();

        /* 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;
        count++;
        sem_up(&mutex);
        sem_up(&full); /* we just increased the # of full slots */
    }
}

void consumer (void *ignored) {
    for (;;) {
        /* 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.