HW 5 Solutions

Question 1

```
A B C
B A C
```

ASSUMPTION: A does not wait, sleep, or block once it has acquired the mutex.

One way to answer this is to observe that the mutex is the only scheduling constraint that we are explicitly given. For this constraint, there are two cases: either A acquires the mutex first, or C does.

In the first case, A, being high priority, executes to completion. Then B, Then C. (Note that if A must go to sleep in the critical section, then we get the B A C output.)

In the second case, C acquires the mutex before A (probably because A's "do something" took a while). But C is low priority, so even though C got to the mutex first, B was scheduled ahead of C, i.e., B managed to run first. So B finishes. When C releases the mutex, A can acquire it, and since A is high priority, A runs to completion, printing A. Then C finally finishes.

[NOTE: you might wonder whether B C A is a possible output. The reason not is that even if C acquires the mutex first, at the instant that C releases it, A will run to completion before C's "do something" block.]

Question 2

```
void reader_release(sharedlock* lock) {
  atomic_decrement(&lock->value);
}

void writer_acquire(sharedlock* lock)
{
  /*
   * a common error here is not spinning, i.e., not including
   * the while() around the cmpxchg. note that spinning is what keeps
   * the writer from entering the critical section before it's
   * supposed to
   */

  while (cmpxchg(&lock->value, 0, -1) != 0) {}
}

void writer_release(sharedlock* lock) {
  xchg_val(&lock->value, 0); /* xchg_val is from class notes */
  /*
   * two other options:
   * 1.  cmpxchg(&lock->value, -1, 0);
```

```
   * 2.  lock->value = 0;
   * option 2. works because the question said to
   * assume sequential consistency. Without sequential
   * consistency, option 2. might not work because it might
   * not guarantee that all of the instructions before or after it
   * would appear in program order to the other CPUs in the machine.
   */
}
```

Question 3

3.1

The code is designed to place box2 inside of box1, and box3 inside of box2. The expected output is:

```
37
- 12
- - 19
```

3.2

The program's behavior is undefined, as `print_box()` will deference an undefined pointer. Possible outputs include an infinite loop, printing of garbage values, or a segmentation fault.

3.3

When a function is invoked with a call-by-value parameter, the parameter is copied and the function executes with a local copy. In this case, `insert_box()` makes a copy of inner; `insert_box()` then sets `outer->inner_box` to the address of this copy. But where does the copy live? It lives on the *stack* and will go out of scope when the function returns. Thus, when the function returns `outer->inner_box` points to invalid memory (concretely, it is pointing to a location on the stack that will be used for something else).

3.4

Change insert_box to use a pointer for inner:

```
void insert_box(struct box* outer, struct box* inner) {
    printf("insert box: placing id %d inside id %d\n", inner->id, outer->id);
    outer->inner_box = inner;
}
```

Also, change `main()` to correctly invoke the modified function:

```
int main() {
    ...
    insert_box(&box1, &box2);
    insert_box(&box2, &box3);
    ...
}
```