1. Multi-part:

a) Output:

```
$ ./prob1
Parent iteration #0.
Parent iteration #1.
Parent iteration #2.
Parent iteration #3.
Child iteration #0. buffer[0]: 0
Parent iteration #4.
Child iteration #1. buffer[1]: 1
Child iteration #2. buffer[2]: 2
Child iteration #3. buffer[3]: 3
Child iteration #4. buffer[4]: 4
Parent iteration #5.
Child iteration #5. buffer[0]: 5
Parent iteration #6.
Parent iteration #7.
Parent iteration #8.
Child iteration #6. buffer[1]: 6
Child iteration #7. buffer[2]: 7
Child iteration #8. buffer[3]: 8
Parent iteration #9.
Child iteration #9. buffer[4]: 9
```

b) Analysis:

- The parent / child threads take turns to an extent. It's not the same every time I run the program. They seem to run as long as they can while each of their semaphores allow them access to the critical section.

c) Analysis:

- Assuming the producer grabs the mutex first, it iterates for 4 iterations. Each iteration, it's posting to the not_empty semaphore, but that doesn't seem to have an effect because it keeps grabbing / releasing the mutex before the consumer can. Once we fill the buffer, the writer calls cwait, releasing the mutex, and and waiting on the write condition variable. In doing so, it has incremented it's condition variable count.

  The consumer then has the opportunity to read once, without getting stopped on cwait because the buffer has entries. It then uses cpost to alert the producer there's room in the buffer, and then gets locked by the next semaphore.

  Since the producer was stuck in cwait, we decrement our producer count, and then write. We're guaranteed mutual exclusion because the consumer is stuck on the next semaphore. We then post to the not_empty condition variable, which has no effect because we're not currently waiting on the not_empty semaphore. We are instead waiting on the next semaphore, so we release it, allowing the consumer to resume.

  The consumer seems to keep grabbing the mutex before the producer, allowing it to run 4 times, emptying the buffer to one unread entry. The cposts to the not full semaphore have no effect, since we aren't waiting on it.

After the consumer's 4 iterations, the producer has a shot, grabs the mutex, writes, and releases.

This continues, and seems to be whoever is able to grab at least one of the semaphores first. This is due to the fact that I didn't implement a mechanism to queue waiting processes and then enforce that ordering. So it's a mish mash.

d) Multi-part:

- History output:

```
$ ./prob1.history
Parent iteration #0.
Parent iteration #1.
Parent iteration #2.
Parent iteration #3.
Child iteration #0. buffer[0]: 0
Parent iteration #4.
Child iteration #1. buffer[1]: 1
Child iteration #2. buffer[2]: 2
Child iteration #3. buffer[3]: 3
Child iteration #4. buffer[4]: 4
Parent iteration #5.
Child iteration #5. buffer[0]: 5
Parent iteration #6.
Parent iteration #7.
Child iteration #6. buffer[1]: 6
Child iteration #7. buffer[2]: 7
Parent iteration #8.
Child iteration #8. buffer[3]: 8
Parent iteration #9.
Child iteration #9. buffer[4]: 9
History: 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 1, 1, 2, 2, 1, 2, 1, 2
```

- Analysis: The output seems to agree with my analysis above in the sense that we alternate for at least an iteration or two, depending on who can get to one of the semaphores first.

2. Output:

```
$ ./prob2
Reader #1: Roses are red.
Reader #2: Roses are red.
Reader #1: Violet's are blue.
Reader #1: Violet's are blue.
Reader #1: Threads honestly seem,
Reader #1: Threads honestly seem,
Reader #2: Rather cool.
```