- In the sample code above, comment out the `sleep(1);` line and run it. What happens? Why does it happen?

The output changes from the waiter sleeping until its count is equal to 5, and waking to see if it is equal to 5, to producing the last two lines of the output. I can only presume this is because the threads are unable to properly count up waiting for the printed lines, the value remains the same, 5, but it is unable to print each sleeping and waking moment before the execution is finished.

- What's the difference between a condition variable and the condition itself (the condition data)?

The condition variable is used by the threading function calls to signal between threads, and the threads use the condition data. In otherwords the condition variable is what we use to test the condition data we're looking to meet.

- Why should the signaling thread hold the mutex before doing so?

Because when the signaling thread holds the mutex, when a thread waits for the condition variable, it implicitly releases the mutex. And when that thread is signalled again through the condition variable, it implicitly acquires the mutex again. So waiting on the condition variable means the thread will automatically release the mutex, and will automatically reaquire it when it wakes up.

- Why is `while`-loop a good idea for threads waiting on a condition variable?

If we assume the thread is sleeping on the wait for condition variable line, another thread signals it so it is ready to run but hasn't yet. Meanwhile another thread runs acquires the mutex and finds the condition met and runs some code, resets the condition data and then releases the mutex. Then the thread that was waiting wakes and proceeds like the condition data hasn't been reset. We want to give the woken-up thread another chance at verifying the condition data has not been modified by another thread, so we use a while loop.

- If the waiting thread has acquired the mutex and is waiting on a call to `pthread_cond_wait()`, how can the signaling thread acquire the mutex seemingly at the same time?

By broadcasting instead of signalling. Broadcasting can wake up all waiting threads, so if one thread got woken up has its condition met than it can try running with its condition.

- Why are two mutexes (one for the producer threads and one for the consumer threads) required for a good producer/consumer implementation?

The producers add some data to a buffer or queue to be processed, the consumer takes data off the queue and processes it. When the queue is empty we want all consumers to be asleep and we want all producers to be asleep when the queue is full. By having two mutexes, then the producers can wake up consumers to take something from the queue and consumers can wake up producers so they can add more to it.