Graham Matthews

At program start, the buffer array is initialized to be empty (empty value represented by -1). The semaphores are initialized using sem_init on lines 34 and 35. The producer and consumer threads are created and stored in arrays in lines 37 through 57. From here, each thread runs its indicated function. Producer threads run the void *producer function and consumer threads run the void *consumer function.

The *producer function begins by sleeping for a random period of time. A random integer is used as the item to be produced and added to the buffer. Using the outputMutex lock on line 78, we ensure that no other outputs running concurrently in other threads output at the same time, causing outputs to possibly interleave. The producer announces it is waiting on the isEmpty semaphore before proceeding, then gives up the outputMutex lock. Once isEmpty has been signaled, the producer claims the input and console output locks on lines 89 and 91 to enter its critical section on lines 93-101. In its critical section, the producer inserts the item into the buffer. After outputting to console the status of adding the item, the locks are again given up and isFull is signaled.

The *consumer function begins similar to *producer, by sleeping for a random period of time. The consumer chooses an item to remove randomly, and gets the outputMutex lock in order to print to console its current status. Consumer waits for isFull to be signaled. When isFull is signaled, the consumer acquires the outputMutex and inputMutex locks on lines 130 and 132. In its critical section, from lines 134 to 142, the consumer removes an element from the buffer. In its remainder section, the consumer gives up the mutex locks.

1. The fact multiple threads can run the same code with the same data. For some reason when studying, I only comprehended that each thread gets one job, and assumed each thread's job was distinct and unique. Seeing that multiple threads are able to run the same code and manipulate the same data with synchronization implementation helped me to understand the implementation of thread manipulation better.
2. Creating threads was easy. After searching for the documentation for pthread_create the thread creation process was simple.
3. The combination of using both mutex locks and semaphores. When studying I only considered scenarios where one or the other were being used, rarely did I consider both could be used in the same program for different purposes (waiting on a signal about the status of the buffer versus locking the output entirely). Where synchronization statements should be placed also confused me. After testing out different places to put calls to sem_wait and sem_post, many appeared to have little change on the program.
4. The purpose of semaphores and mutexes. I believe I comprehended the differences between the two well when studying, but seeing the implementation and when a mutex is more applicable than a semaphore and vice versa helped to solidify my understanding of the subject. Implementing the producer function to wait on the buffer to be empty and the consumer function to wait on the buffer to be full helped my understanding of the subject as well. Then when used to compare to printing to the command line using mutexes, I understood which scenarios call for mutexes and which call for semaphores.
5. I am satisfied with my current design. I feel as though I abstracted away as much information as I could in the main.h and buffer.h header files, and wrote concise and efficient code.
6. The fact that I could move the calls for some synchronization and the program ran correctly. As long as the places I put the calls to synchronize the program make sense to those reviewing the code, that should be the first priority. As far as correctness goes, I was able to move the calls around from inside and outside of loops and conditionals without sacrificing correctness. So that aspect was the most interesting to me.