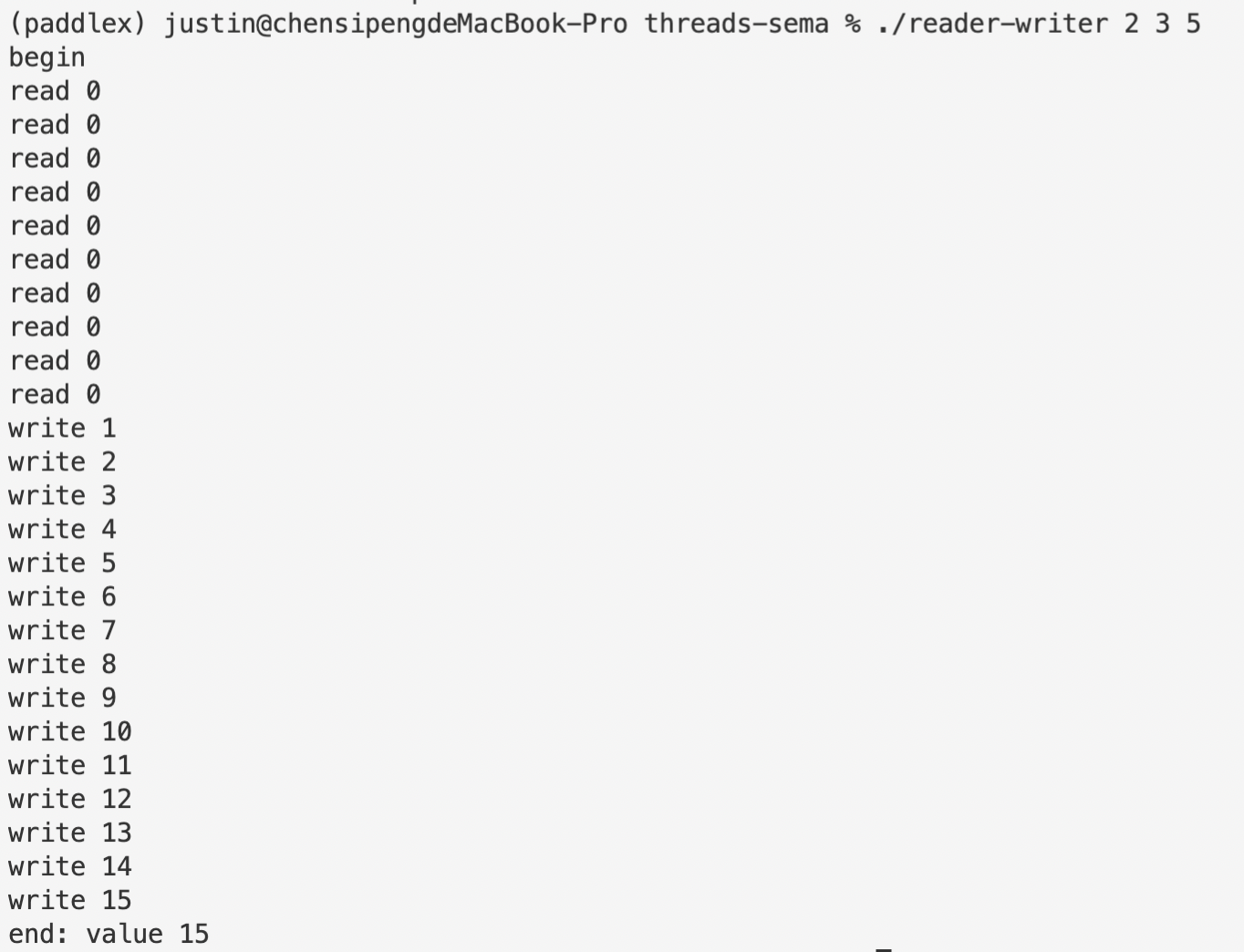
**Cpt 31**

**Homework (Code)**

**4. Now let’s solve the reader-writer problem, also as described in the text. In this first take, don’t worry about starvation. See the code in reader-writer.c for details. Add sleep() calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?**

The idea here is to build the writer lock that will spin until the thread that calls the lock is the only remaining runnable thread, and the reader lock that permits multiple reader threads to run simultaneously. Implementation of the writer lock will be quite simple. To acquire the lock, we simply let the thread spin until it receives the signal sent out by semaphore rootEmpty, which is going to signal once the thread queue is empty, metaphorically. As for the writing part, since we don’t allow any other executions to run with the writing thread, as soon as the writing execution ends, there will be no threads running, so we let the writing thread signal through the rootEmpty semaphore to wake all the threads waiting. Readlock is a bit more complex. To start with, the readlock should keep record of the number of reading threads that are currently running so that it could wake the sleeping writing thread once there are no reading threads running. To acquire the lock, we should add the number of reading threads by 1 and judge if the current thread is the first reading thread. If so, it will have to wait till rootEmpty signals, which is when the writing thread ends. Note that this whole process should be enclosed by a mutual excluding lock that makes the process atomic. Release part is largely the same, except that we should minus the number of reading threads by 1.

After coding, we may test the code with parameters num\_readers = 2, num\_writers = 3, and loop = 5. The output is shown in the screenshot below, from which we can see that writing threads are blocked until all reading threads end, which indicates that our implementation will indeed invoke the problem of starvation.



To show the code is working as expected, add a line of sleep() between the reader lock acquiring and releasing and test with the same parameters. The output is exactly the same as the screenshot above.

**5. Let’s look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See reader-writer-nostarve.c for details.**

To solve the starvation problem, we can add another semaphore named turnstile. Just as its naming suggests, this new semaphore is going to work as a turnstile, but just for readers. For writers, it will act as a mutual excluding lock. The writer waits for turnstile to signal, then operates as how we implemented in reader-writer.c, and signals through turnstile afterwards. When a reader enters, it should first wait for turnstile’s signal, and after that, the reader can immediately signal through turnstile to let in other readers. In this way, once the space has no readers, the writing thread that awaits can gain control at once, which is going to prevent the situation where writers have to wait till all readers have quit.

Tests with same parameters prove that writing threads no longer suffer from complete starvation.



Note that, however, after multiple trials, most of the time writing thread still start to come in after all reading threads have finished their job, which is because our solution here requires will introduce a race condition where reading threads that are trying to enter and writing threads that are trying to secure their place through signaling through turnstile competes with each other to gain control. 