Homework #1

Oregon State University

CS444 Operating System II

Spring 2018

Erich Kramer

April 15, 2018

1 Concurrency 1

1.1 Questions:

1.1.1 What do you think is the main point of the assignment?

Concurrency exposure, some early semaphore taste to get used to proper concurrency assignments.

1.1.2 How did we personally approach the problem?

There are three different conditions that need to be addressed with some form of locking. One, the buffer of objects cannot be both read and written to at the same time, likewise we only want one thread to be writing or reading at a given time. So we need a global mutex to lock the buffer as it is globally shared. Instead of having it be a global variable lets just set a struct that we pass as args to all of our threads. It is still effectively global but at least it looks less awful in the source code. We also need to address the condition of a full and empty buffer in the context of the consumer and producer. To do this we have two unique semaphores that limit readers and writers so that only so many may be functioning at the same time. Even with more consumers than we have buffer slots we will not accidentally over-read or access an empty buffer location. Similarly we don't want to overwrite an element with our producer and so we limit this with another semaphore.

1.1.3 How did we ensure the solution was correct?

One of my biggest concerns was that my code might segfault on edge conditions. To test this and later to debug I would run with a low number of one type of thread and a very high number of another type of thread.

1.1.4 What did we learn?

Using sem_getvalue to keep count of things is really bad. Threading is not as scary as it might seem.

1.2 Program Design

I originally approached this with the idea that I would use a semaphore to control the count of elements in the list for both the consumer and producer. One semaphore would observe the quantity remaining, the other would observe the distance from a full array. In doing this I could allow the consumer to block when the list was at 0 elements, as it would wait on the semaphore which was the quantity in the list. The producer also would wait on the semaphore for the quantity of spots left in the list, when it was 0 it too would wait. The consequence of this was that when running multiple threads there was a race condition in my design. I circumvented this issue by instead using a counter which was shared across threads for the size of the buffer and simply incrementing decrementing only inside the thread which had control of buffer access. The semaphore here working only to prevent multiple consumers that might attempt to consume when there are not enough elements to be consumed.