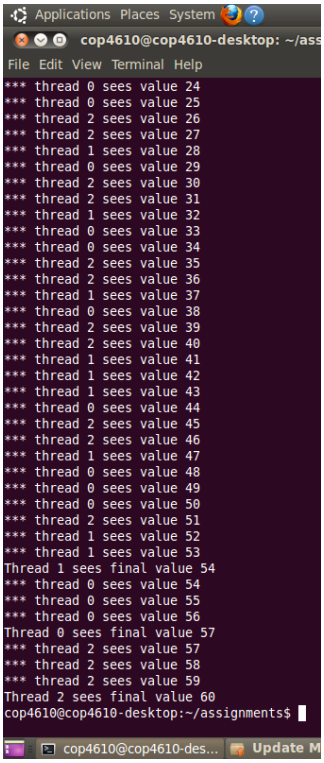
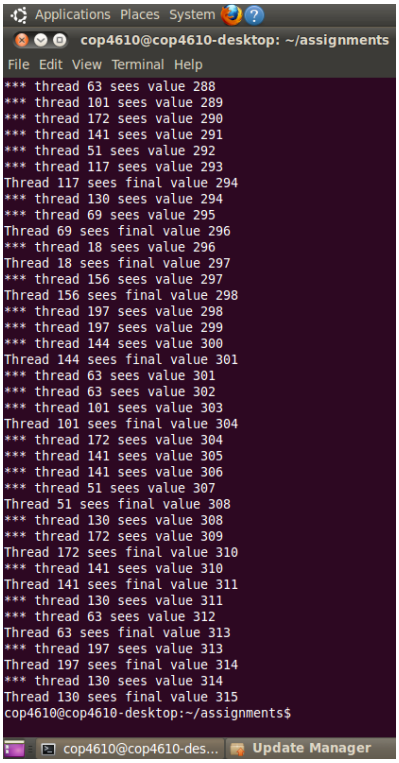
Alastair Paragas
Nagarajan Prabakar
January 28, 2017

Operating Systems: Lab 2

Report

Step 1.1: Simple Multi-threaded Programming without Synchronization





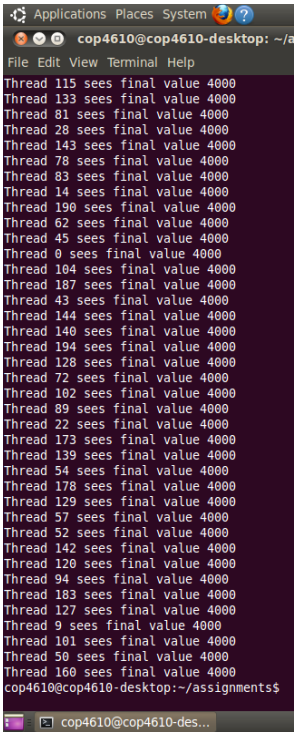
Unsynchronized run (3 threads)

Unsynchronized run (200 threads)

In both execution of the programs, it was noticeable that the final value emitted by all the threads were different values, which was different from the expectation that each thread would emit the same value. This is because each thread is allowed to indeterminately sleep and wakeup for an unknown amount of time each loop iteration, which compounds to each thread having a different ending time span size. It was also noticeable in the execution of the program, especially with larger thread amounts, that the value becomes corrupted. For example, considering that on the second case, 200 threads were ran, each with 20 loop iterations encased within the thread, it

was expected that the final value would be (thread_count * loop iterations) = 4000, but this was simply not the case. This was because the critical section of the program (which accessed the SharedVariable, incrementing it), spun across multiple threads, was not guarded against concurrent execution, nor was there any mechanism to ensure that the critical section was ran as an entire transactional unit.

Step 1.2: Simple Multi-threaded Programming with Proper Synchronization



Synchronized run (200 threads)

In this case, the final value was synchronized because of the usage of a pthread_barrier, allowing for all threads to cease operation until all threads arrived at that corresponding portion of the code on their individual executions. It was also noticeable that the expected value was not corrupted at all, because of the usage of a mutex lock that allowed us to ensure that a thread's entire critical section didn't run concurrently/interleaved with another thread's critical section code, allowing each critical section portions to run as an entire unit.