Luis Perez Concurrency Model Report

Locks (Mutex): Mutex and locks were relatively limited in their handling of the operations. In order to use them you have to adjust your approach to what the lock could accomplish. The lock could only block the part of code that required the synchronization, without any other conditions to check. For this the code was relatively easy to understand and implement, and the psychology of using a semaphore carried out to the different approaches long after we had already used Mutex. However, I think that its limitations changed the way I approached the problem and implicitly made the program worse. Additionally it was needed to manually guarantee that there would not be data races or deadlocks

Software Transactional Memory (STM): STM was used in a very similar way to a mutex. However, we could be sure of the safety of the data and transactions, there was no longer a need for checking for data races or deadlocks. STM had the complication of having to understand what represented a possible inconsistency in order to wrap the operation in a transaction. In order to achieve the results, every operation that shared information had to be made into a transaction, and because a great part of the data was obtained from counters and accumulators that were regularly updated, then every bit of information was prone to failure on write/read. The code required more updating but there was certainty of consistency, which was good.

Communicating Sequential Processes (CSP): CSP displayed more control over the process when making the flow of operations. No longer was the process dependent on the logic of the approach and more on the choice of the developer. Despite this, there was still certainty that there would be no data races or deadlocks. The locking was on purpose and it accomplished the purpose of making the operations wait on certain moments to synchronize. It was similar to the Mutex, but it allowed great control of how and when was the blocking occuring, not simply by virtue of the different processes trying to access a piece of code. With the passing of messages through the shared channel, the parts of the code where told when and when not they could proceed. It was a welcome addition and it worked very well. Additionally I felt it did not limit my code anymore and I had the freedom to make a more robust solution, knowing that it would keep consistency.

Actors: Being very similar to CSP, actors learned even more on the flexibility and control the programmer could display. Every action was mapped out to specific messages, no longer was it a nameless queue but a set of commands being waited on. Every operation knew what it wanted the other to do, and told it so. With this, the code felt a lot more intentional and clear. Actors helped me do what I wanted to instinctively do from the beginning, “tell” each process what to do at any given moment in the information flow. Now I had complete control of how and when my operations were working. Like CSP, we could be sure that there would be no data races or deadlocks, and it did not need to account for shared data problems like STM. The operations were done independently by each process and the actor was in a non blocking way being notified when to work and when to wait. In my opinion, despite being harder to understand than the other approaches, it was my favorite to get to work.