Assignment 4 Report

CMPT – 431

Joshua Campbell - jkcampbe@sfu.ca

Adam Penner – adpenner@sfu.ca

Settings: TRYRACE not defined, red rate: 1000000, blue rate: 1000000, 10 rounds

|  |  |  |
| --- | --- | --- |
|  | Red Success / Second | Blue Success / Second |
| RogueCourse | 7436.26 | 6674.86 |
| RogueCourse2 | 4488.54 | 4958.68 |
| RogueFine | 7551.24 | 6802.72 |
| RogueFine2 | 4187.02 | 5116.87 |
| RogueTM using RTM | 7441.02 | 7048.62 |
| RogueTM2 using RTM | 5668.26 | 6262.11 |
| RogueTM using HLE | 7662.08 | 8017.21 |
| RogueTM2 using HLE | 4196.49 | 5675.51 |

Settings: TRYRACE defined, red rate: 1000000, blue rate: 1000000, 10 rounds

|  |  |  |
| --- | --- | --- |
|  | Red Success / Second | Blue Success / Second |
| Rogue Unprotected | 184.684 | 5135.97 |
| RogueCourse | 173.211 | 321.722 |
| RogueCourse2 | 325.057 | 692.636 |
| RogueFine | 168.128 | 312.235 |
| RogueFine2 | 225.759 | 333.939 |
| RogueTM using RTM | 70.028 | 6343.85 |
| RogueTM2 using RTM | 62.7963 | 2970.64 |
| RogueTM using HLE | 153.105 | 198.822 |
| RogueTM2 using HLE | 314.089 | 481.71 |

Observations with TRYRACE not Defined

First observation is the difference between setting two lanes at once versus setting only one lane through various methods. Setting two lanes at once was always slower for one reason in which we had to search until we had two lanes that were both white and try to lock both of those lanes. If after locking the lanes aren't the same colour as before, we would have to try again to get two new white lanes. This act of needing two perfect lanes slows the success rate down by **20-30 percent**. By trying to get two new lanes instead of holding onto the one good one lane and finding another, we prevent deadlock situations. Such a situation would arise where there are only two lanes left, and each thread has one of those lanes and needs another to complete the call.

From the data above we also note the steady increase in cumulative success rate when dealing with single shot instances. These increases, while minor, follow the same path of going from a stricter locking mechanism (single lock) over to a more flexible and adaptable way of changing data (RTM and HLE).

|  |  |
| --- | --- |
|  | Cumulative Rate / Second |
| RogueCourse | 14,111.12 |
| RogueFine | 14,353.96 |
| RogueTM using RTM | 14,489.64 |
| RogueTM using HLE | 15,679.29 |

Compared to the course and fine grain locks, the RTM and HLE implementations were a lot easier to reason about without adding extra complexities. My extent of RTM and HLE implementations came down to the cleaner functions for shooters. The more verbose aspect of the RTM implementation made it easier to deal with failed attempts and how the overall flow should be like if there is a failure. The HLE implementation on the other-hand is extremely easy and straightforward, but lacks the option to deal with retries and failures.

**Analysis of RTM and HLE Implementations:**

When looking at the RTM and HLE implementations in this project’s code, one of the most striking differences is that RTM seems to look similar to the unprotected “Rogue” implementation (in that it is optimistic that there won’t be a race condition), whereas the HLE implementation looks very similar to that of a coarse grained lock. However, technically they both perform as transactions. RTM achieves this by being optimistic about there being a race condition or not, if there is a race condition, it resets and enters the fall back state (either retries the optimistic approach or utilizes the fallback locks). HLE is also a transaction even though it looks like a lock. HLE (in this case) works as a backup to the traditional lock. It works like a lock, but if there is some unforeseen conflict or race condition that occurs, it will reset to its initial locking state and reattempt the transaction. Both HLE and RTM are transactions; however, they utilize completely different methods to achieve this functionality.

**Difficulties in the Development Process:**

One of the main difficulties was trying to find a race condition once a lane was checked. This was difficult because it was rare that the threads would properly sync up to check at the exact same time. Ultimately, this was overcome in our race version by forcing a certain thread (in this case the red thread) to enter a for loop (a few thousand increments) right after the check and right before the attempted shot in order to simulate the thread not shooting fast enough. This debug/forced race condition mode allowed us to test our locking and transaction implementations to ensure that they performed their jobs accurately.