Peter Maurer – pdman13

Broulaye Doumbia – broulaye

Design and Problems

By understanding that the tasks form a binary tree we could understand that all the futures form a recursive sort, in post order. Thus, our thread logic was fairly easy to design. We did struggle during implementation, getting our locks to align properly. We used a very fine-grained lock system, with each lock protecting one item – either a future or a queue. Each future and each queue, including the global queue, had their own lock. This made ensuring no deadlocks slightly harder, and it took us some time to eliminate all our deadlocks and extreme waits. After we had the locks working, we moved on to helping and stealing.

First we implemented stealing, for which we followed the logic in the spec – steal in FIFO order. Stealing proved far easier to implement than the locks, and took very little time. We tested, and all tests ran and completed successfully. Then we moved on to helping. Helping cause us more issues than stealing, as when we tried to implement it we promptly got correctness errors. We struggled with what the proper behavior for helping should be, especially if the thread calling future\_get() was the main thread that created the threadpool. Once we determined the proper behavior, things started to fall into place. We changed future\_get() so that interior and exterior calls were being handled in completely different cases at the highest level, rather than trying to do so in each case, we also implemented leap frogging to optimize the wait we help a running task, it proved to be harder than we thought as it require perferct synchronization of the way we were acquiring and releasing locks. This resolved the remaining errors, and all we were back to all tests working. Since everything worked and we were getting no detectable race conditions, that turned into our final version.