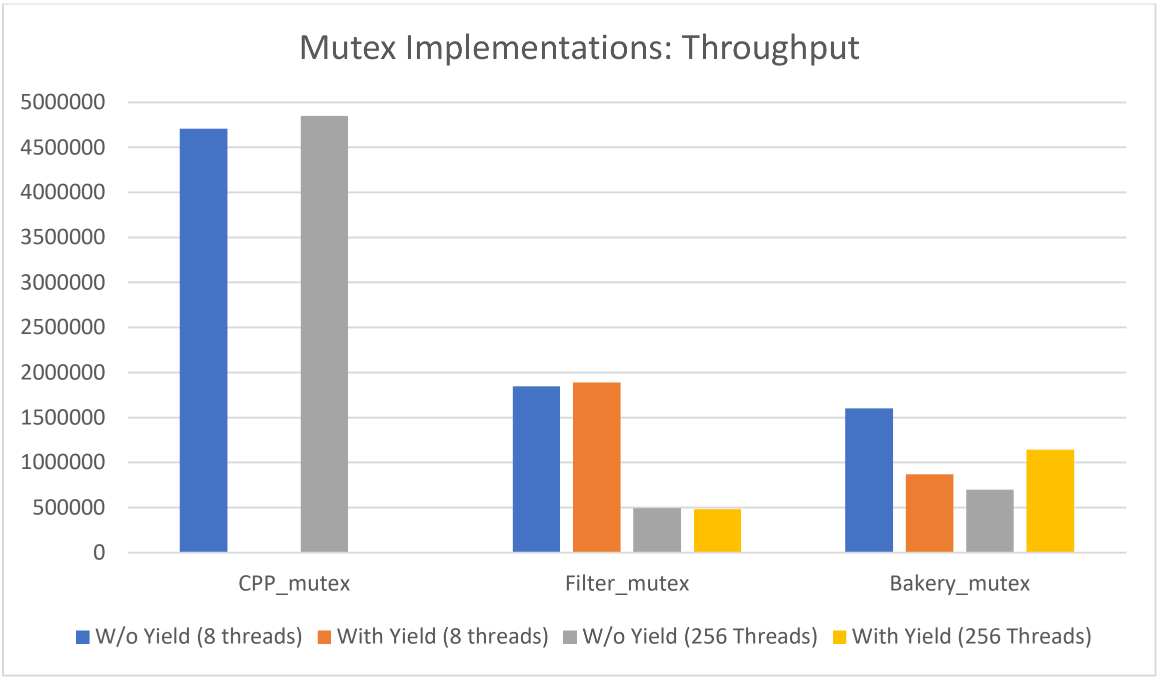
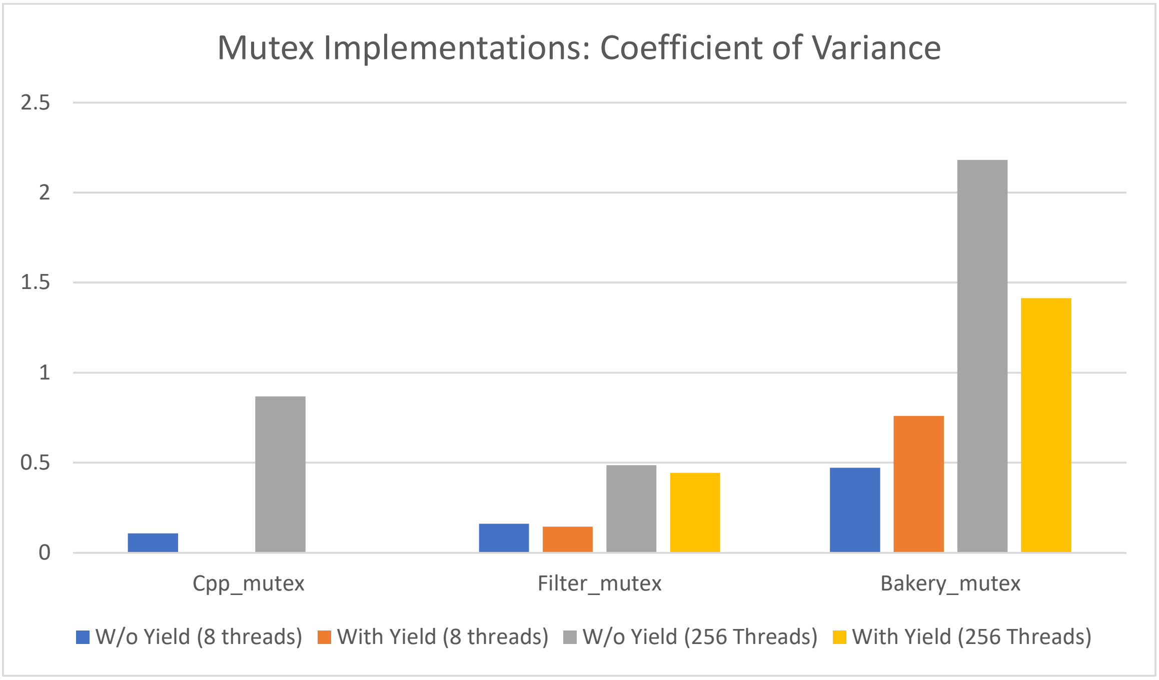
# Homework2 Write-Up

Galen Turoci

CSE 113

## Part 1





Presented here are the results – the total throughputs and the coefficients of variance – of 3 different mutex algorithms, 2 of which I implemented. For reference, I am running this on an 8-core 3700MHz processor and 16GB of RAM.

For the throughput, it does concern me somewhat that my algorithms produce so much less than the C++ mutex, however I have heard other students in office hours and Piazza express similar concerns and be told this was to be expected, so it is not an overlarge concern. Interestingly, my coefficients of variance seemed to fair a bit better, particularly with larger amounts of threads; why my bakery mutex spikes so sharply, being the only mutex to have a CofV >1, I suspect is due to the far smaller mean of its throughputs. One interesting result not shown on the charts is that without thread yielding, many of the threads for the bakery mutex obtained the mutex exactly 3 times; I have no idea why it is 3 in particular and not a more randomized assortment of numbers.

## Part 2

Default mutex average starvation factor (Reader:Writer ratio): ~60

Fair mutex average starvation factor: ~3.5

As is plain to see, my Fair Mutex decreased the relative writer starvation by a factor of nearly 20, causing far less discrepancy between the amount of readers and writers. To do this, I implemented a vector-based ticketing system, wherein as threads accessed either the reader or the writer, that thread's ticket would be placed at the front of its respective vector and compared to the ticket at the front of the other vector, forcing it to, conceptually speaking, wait for its number to be called before it could be served. Though I was under the impression my algorithm would bring the starvation factor down to a nice 1:1 ratio, I am still more than pleased with the result, and since the readers outnumber the writers by a factor similar to the starvation ratio (i.e. 3 readers for every 1 writer, similar to my 3.5 starvation factor), I suppose this outcome is more or less as good as I could hope for. As for why the code I posted on Piazza did not work, you'll notice that in that code, I have two internal mutexes per lock, with the first one being called just prior to the pushing of the ticket; however, via testing I discovered it was this mutex that was never being unlocked due to where I placed its unlock function, so I realized I simply had to move the unlock function to just underneath the vectors' push_back() functions.

For my results, I am a little bit concerned that the amount of times the readers and writers were respectively accessed suffered rather severely; for the default mutex, the readers were usually accessed ~3 million times while the writers ranged from ~35,000 to ~90,000 accesses, whereas with my mutex, the readers were accessed between 100,000 – 150,000 times and the writers were usually a bit above 30,000. I believe this is due to the sheer amount of overhead brought on by all of the vector operations; I'm beyond certain a more efficient solution exists, but for the time being I am content with this one.

## Part 3

Results:

Coarse lock results:

- Total operations: 976,076.6
- Pops: 647,712.5
- Pushes: 89,731.3
- Peeks: 238,632.8

RW lock results:

- Total ops: 801,244.8
- Pops: 26,929.5
- Pushes: 4,865.4
- Peeks: 769,450

Swaptop lock results:

- Total ops: 1,175,838.2
- Pops: 27,042
- Pushes: 4881.4
- Peeks: 1,117,472.5
- Swaptops: 26,442.3

My results in the RW lock seem to have underperformed, with the average number of operations being almost 20% less than that of the coarse lock, though this may simply be due to bad luck; during implementation and testing, it was not uncommon for my number of RW operations to number over a million, sometimes greatly so. In any case, the shared_mutex implementation works as intended, with the number of peek operations vastly outnumbering the number of all other operations.

For Swaptop, the implementation was rather trivial, though I was unable to use the shared lock, as every time I tried to use it the program would segfault for reasons I don't understand and have unfortunately run out of time to figure out. I would think using the shared lock to look at the start of the stack before switching to the exclusive lock for swapping would be a valid use of the mutex, but evidently not.