X 401031: Concurrency Multithreading

Coarse-grained locking

Håvard Skjærstein

VU id: hsk202

Github user: havask

September 30, 2022

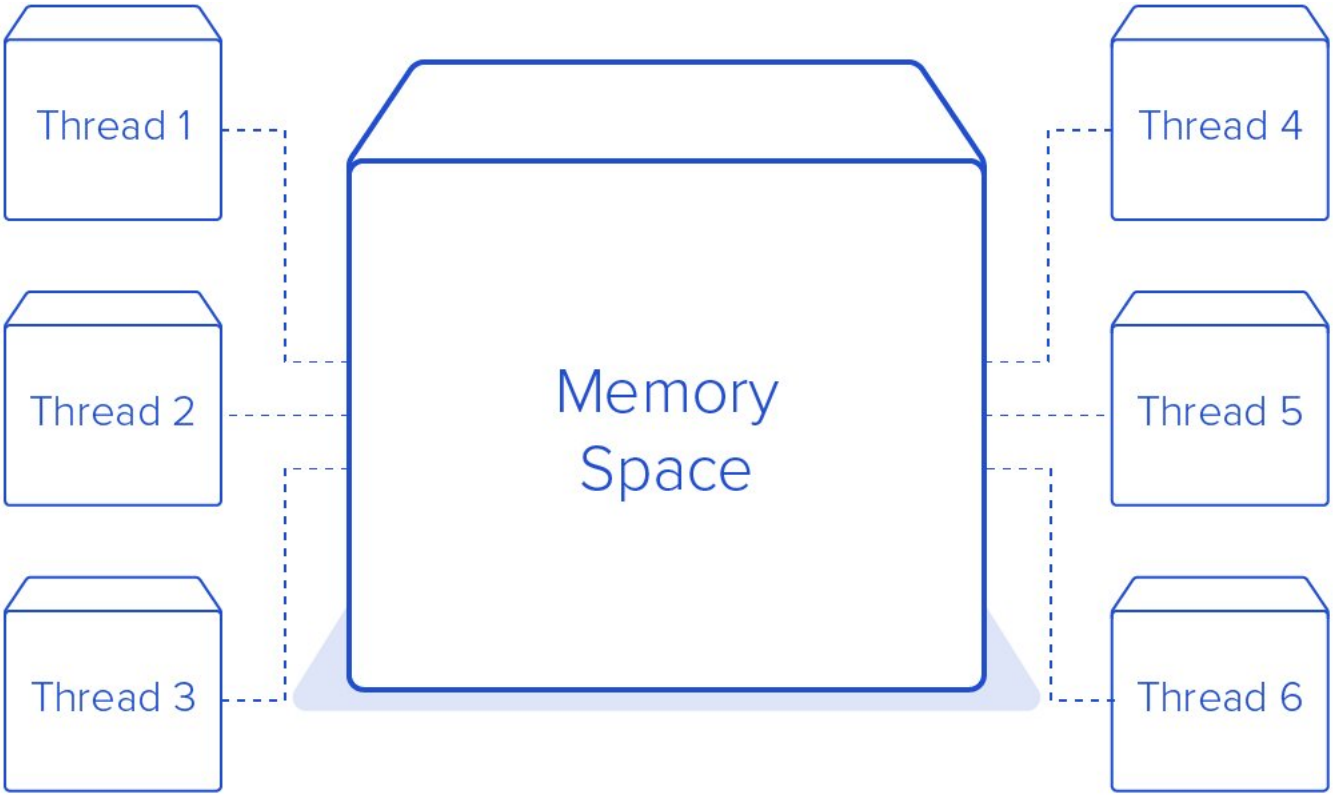


Figure 1: This illustrates a relation between memory and threads.

1 Introduction

Today most modern processors come with 4 cores or more, and each core usually has 4 threads each. Concurrency takes advantage of this computing power by implementing many subprocesses of a program to not waste CPU cycles. This assignment's goal is to implement two data structures using different concurrency mechanisms, examining the performance and writing a report.

2 Design

The program is implemented using coarse-grained synchronization. That involves taking a sequential implementation of a program, adding a lock to it and making sure that each method must acquire and release that lock. This design is to protect the critical section and prevent race conditions from occurring. For the data structures I used my own linked list and the tree is from the java.utils package due to time limitations. [1]

For the locking mechanism a reentrant mutual exclusion lock was implemented. The lock is owned by the thread that last locked it, and goes to the next thread once unlocked. The mutual exclusion ensures that only one thread can be in the critical section at the time and prevents race conditions and deadlocks, and ensures thread safety. [2]

The threads are treated as equals. It is designed to resemble a round-robin scheduling approach, meaning every thread is guaranteed to get a chance at execution, ensuring 100 percent thread fairness. Each method tries first to acquire the lock, and when it succeeds it locks the entire data structure and adds or removes an item from the structure. [2]

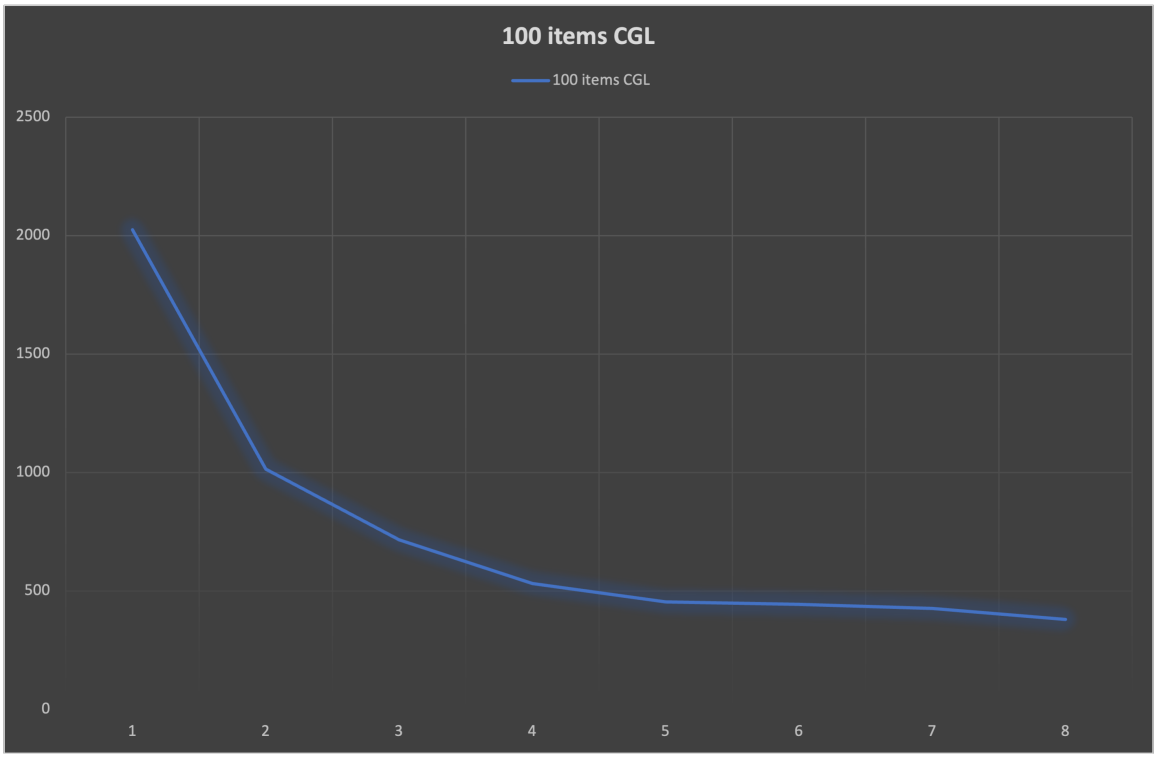
3 The hypotheses

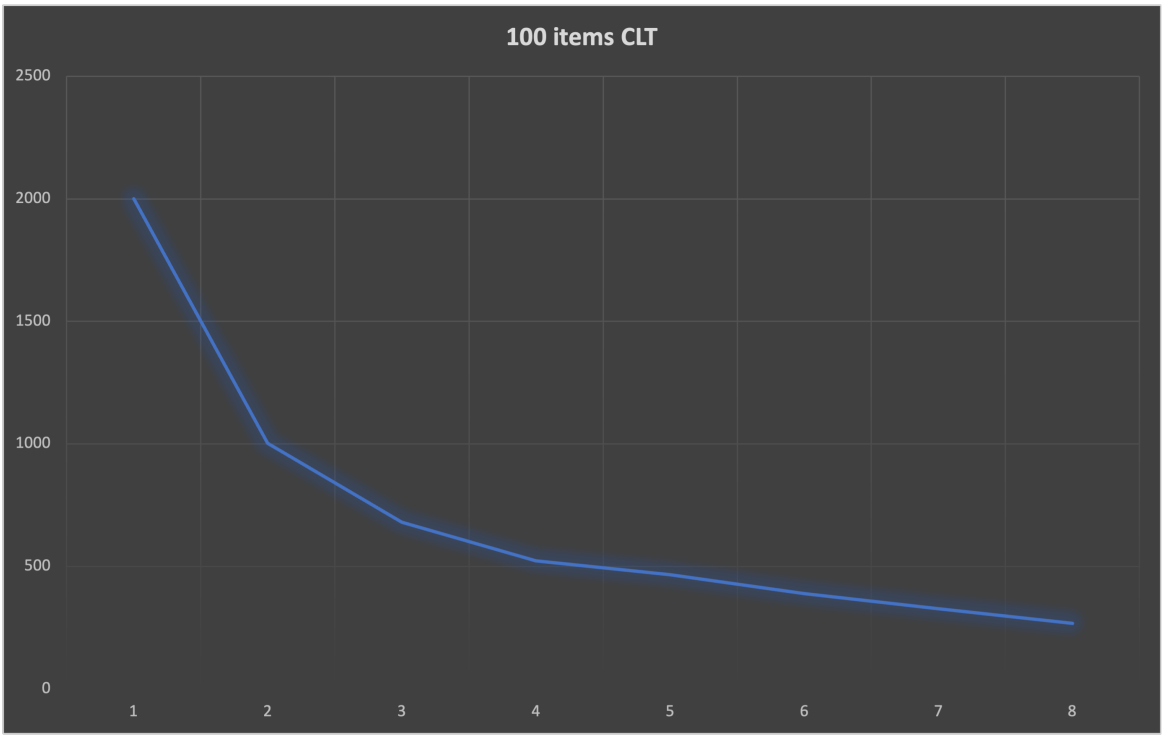
The hypothesis is that the performance of the program will increase with each thread, no matter the amount of workload. It will do so until the program eventually becomes a sequential bottleneck making threads wait in a long and slow queue before they can gain access to the critical section and execute.

The bottleneck is predicted to be the access to the data structure so adding new threads may at some point not increase the performance, and the concurrency will stagnate. The performance will be decided by how fast a thread can get in and out of the critical section. It doesn't matter if it's 100 items or a million, the speed will be the same so the scalability is predicted to be very bad with this concurrency scheme.

4 The evaluation

For evaluating the program two tests were done with 10 and 100 items, with a number of threads ranging from one to eight. The number of items had to be divisible by the number of threads so in the testing I added a few items for the program to run. The program was not run on the DAS cluster because I, an ignorant exchange student, was not able to find the computer labs in time. The tests are therefore run on a M1 macbook with it's limited cores and capabilities.


100 items CGL
100 items CGL
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8


100 items CLT
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8

As shown in Figures 1 and 2, the results for the linked list were that by going from one to two threads the performance increased by 50 percent. Going from two to three threads increases the program with another 40 percent. After three threads the performance stagnates when adding more threads to the program. The result from the tree structure ended up being the same. It had the most performance increase with three threads. After that the performance also stagnated. The program was also tested on different numbers of items, but the performance curve was still the same, it just took longer time to process.

The result from the measurements correspond with my hypotheses and expectations in that the program gets bottlenecked by the coarse grained lock because only one thread can access the data structure at a time. The performance is therefor decided by how fast a thread can operate on the given data structure. At some point it does not matter how many threads get added because to get executed they all must wait in the same, slow queue. The coarse grained implementation performs great with 2-3 threads, but it does little when adding more than that. It just adds to queue, not the throughput.

By switching the implementation from sequential to multi-threaded the CPU gets to be utilized more efficiently and gives a much higher throughput. The result reveals how much the CPU stands in idle mode when executing a program and how much multi-threading can improves performance. By just adding one extra thread of execution the program runs twice as fast, but at the same time we can't continually add threads and expect the program to run faster, the locking of the whole data structure bottlenecks the program after three threads. That is why the program will not scale well either because the output for adding and removing 100 items will be the same for a million ones.

5 Conclusion

This assignment is centered around concurrency and the implementation of two data structures using different concurrency mechanisms, and examining the performance. The report shows that with a coarse grained lock the program had the most performance increase with adding three threads and works well for small programs.

6 Sources

References

[1] Michael Spear, Maurice Herlihy. The Art of Multiprocessor Programming, 2nd edition. Chapters 9.4.

[2] Michael Spear, Maurice Herlihy. The Art of Multiprocessor Programming, 2nd edition. Chapters 2.