

Dalton Driscoll

Shem Louisy

Abumere Okhihan

Wednesday, February 25th, 2025

EECE 4811/5811 Operating Systems Spring 2025

Professor Tseng

Homework 3

Go Design Principle

[1] In Go, the principle "Don't communicate by sharing memory; share memory by communicating" encourages developers to use channels for passing data between goroutines rather than allowing multiple goroutines to access shared memory directly. This ensures that only one goroutine manipulates a particular piece of data at a time, reducing race conditions and making concurrent programming safer and better. By using channels, Go simplifies synchronization because data access is managed atomically, eliminating the need for many locks. [2] This enhances safety by preventing data corruption and concurrency bugs. However, this philosophy also has some drawbacks. Using channels can increase performance overhead. Message passing is often slower than directly accessing shared memory with proper synchronization mechanisms or locks. [3] Additionally, improper use of channels can lead to deadlocks, where goroutines get stuck waiting indefinitely for messages that never arrive. Regardless of these drawbacks, Go's approach to concurrency generally leads to more maintainable and overall better code.

Comparing Queue Algorithms

[5] The code consists of two kinds of mutex inclusive linked list. The first is a "Normal" (`list_t`) linked list that locks the entire list upon insertion, searching and destroying. [4] The second is a "Hand Over Hand" (`list_hh_t`) linked list that locks the entire list upon access but then releases the lock as it moves down the nodes of the list, locking each node upon access and releasing the lock when done. The same concepts are

kept for destroying the list, where the “*Normal*” linked list locks the entire list, whereas the “*Hand Over Hand*” linked list locks the entire list, along with each node before destroying it.

The benchmark here tests two different race conditions for each linked list. The time it takes each thread to insert into each list type at the same time (`thread_insert`), and to search each list at the same time (`thread_lookup`). The threads know which list to test using the `args` parameter (0 for Normal List, 1 for Hand Over Hand List). The user passes the `Node_Count` parameter through the command line, which sets the workload (the number of nodes to test). The `timespec` struct is used to keep track of and calculate the time it takes the thread to complete.

Results

While the “Hand Over Hand” linked list improves on security and memory protection compared to the “Normal” list, it lacks performance when it comes to speed. This rings true especially as nodes increase. Both linked list uses the same insert locking algorithm, so they have similar benchmark times. The main difference occurs when searching through the linked list, as this is done in a linear fashion. This causes threads to fight for individual nodes in order to proceed when accessing the “Hand Over Hand” list, slowing down the threads, while threads waiting for access to the “Normal” list simply acquire the entire list and proceeds to access all the nodes.

```
shem@Shems-MBP-2 HW3 % ./main 100000
Testing regular list insert times for 100000 nodes:
Thread Execution time = 0.006310 seconds
Thread Execution time = 0.007121 seconds
Testing hand over hand list insert times for 100000 nodes:
Thread Execution time = 0.007854 seconds
Thread Execution time = 0.008257 seconds
Testing regular list lookup times for 100000 nodes:
Thread Execution time = 16.049879 seconds
Thread Execution time = 16.499878 seconds
Testing hand over hand list lookup times for 100000 nodes:
Thread Execution time = 51.572754 seconds
Thread Execution time = 51.572811 seconds
```

In this example, when testing with 100 000 nodes, which results in 100 000 lookups, maximum insertion times are pretty much the same for both list types. However, a lookup takes significantly longer for the “Hand Over Hand” list, almost 35 seconds more.

Instructions


Compile using GCC in a Mac (*preferred*) , Linux terminal or Windows command prompt:

```
shem@Shems-MBP-2 HW3 % gcc -o main main.c list.c list_hh.c  
shem@Shems-MBP-2 HW3 %
```

Run the compiled file with the node count parameter.

```
shem@Shems-MBP-2 HW3 % ./main 5000
```

Node Count



References

[1] R. Pike, "Go at Google: Language design in the service of software engineering," Go.dev, 2012. <https://go.dev/talks/2012/splash.article>. [Accessed: Feb. 25, 2025].

Rob Pike is one of Go's creators, as well as this article is from the official Go website. Therefore, this article has credibility relating to Go's philosophy and documentation.

[2] A. Gerrand, "Share memory by communicating, " Go.dev Blog, Jun. 9, 2015. <https://go.dev/blog/codelab-share>. [Accessed: Feb. 25, 2025].

This article is also from Go's official website and outlines the positive aspects of sharing memory through communication. This is a blog posted by another creator of Go, Andrew Gerrand, furthermore, proving its credibility.

[3] W. Johnson, "Golang explanation: Share memory by communicating, " Medium, Nov. 21, 2023 https://medium.com/@relieved_goldmole_613/golang-explanation-share-memory-by-communicating-8be944cbf8f8. [Accessed: Feb. 25, 2025].

This article, however, is credible because the author is an ex-Google software engineer with over 10 years of experience and the team believes it to be a very well written and informative article about Go's philosophy.

[4] E. Demaine, "Lecture 21: Splay Trees," MIT CSAIL, 2008. <https://courses.csail.mit.edu/6.852/08/lectures/Lecture21.pdf>. [Accessed: 25-Feb-2025].

This document is a lecture from MIT's Advanced Data Structures course (6.852) by Erik Demaine, a well-respected professor in theoretical computer science. MIT CSAIL (Computer Science and Artificial Intelligence Laboratory) is a leading research institution, making the source highly credible.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1st ed. Arpaci-Dusseau Books, 2018.

The textbook for this course.