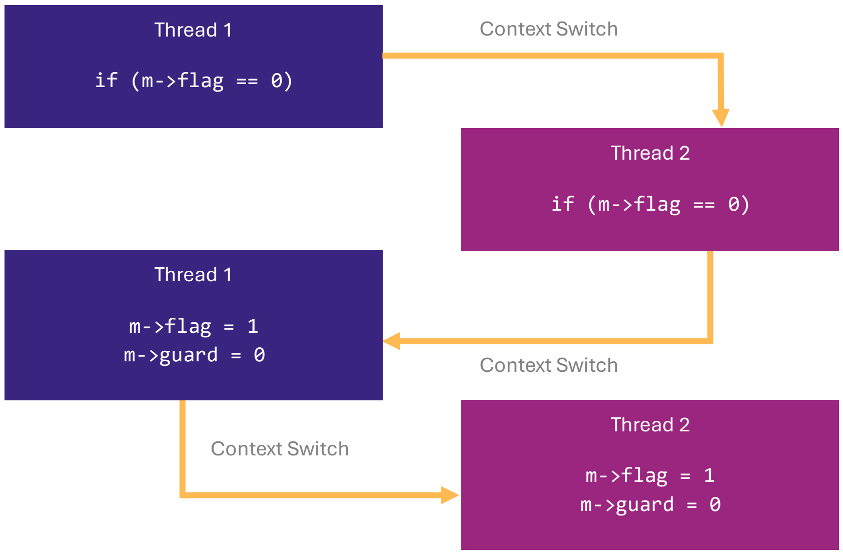
Dalton Driscoll
Shem Louisy
Abumere Okhihan
Wednesday, February 19th, 2025
EECE 4811/5811 Operating Systems Spring 2025
Professor Tseng

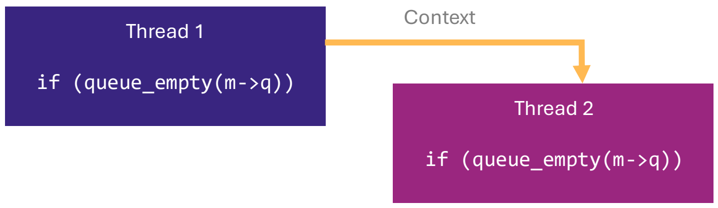
Homework 2

Lock Analysis

[1] Removing the guard code will increase race conditions as threads attempting to lock the critical area no longer have to spin/wait to acquire the guard. This means that multiple threads can adjust the flag between context switches. As a result, the queue might remain empty, meaning that they can then unlock the critical area without spinning either and adjust the flag and guard.



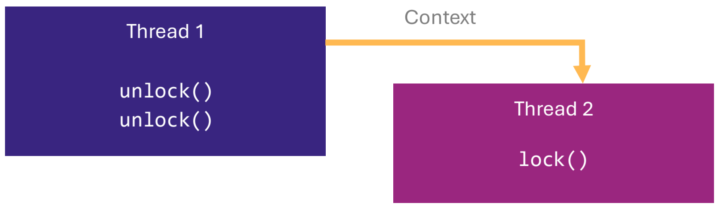
In the demonstration above, a malicious scheduler can switch between threads extremely fast to increase race conditions. [2] If done correctly, the if statements can run true before adjusting the flag, allowing both threads to then proceed to adjust the flag, acquire the lock and enter the critical section. If the malicious scheduler switches fast enough, it can accomplish this with several more threads.



Under these conditions the queue could remain empty, and threads can simply unlock the lock and set the guard.

Lock Analysis II

[3] Replacing the code `lock->flag` with `lock->flag – 1` creates potential issues with improper use:
No Conditional Check | there is no code (no if or case statement) to check if the flag is 1 or already has a value of zero before subtracting it by 1. This means that the value of flag can go negative if the `unlock()` function is called once or multiple times when the flag is already 0.
No Work Around | This doesn't work even if the flag variable was an unsigned variable, as an unsigned variable math will result in 0 – 1 = 4294967295.

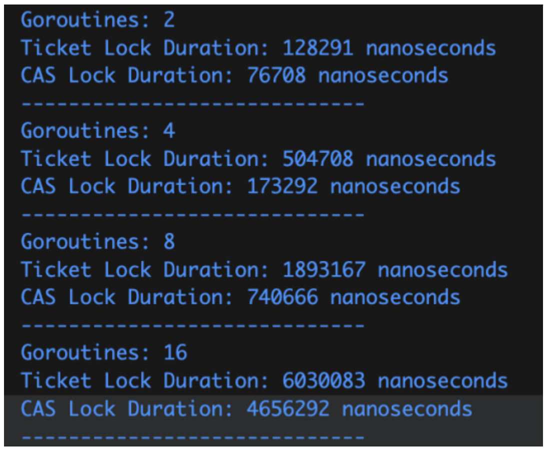


In the above example, a malicious scheduler can create unfairness allowing one thread to run longer than others to increase race conditions. If a thread calls `unlock()` more than once due to poor coding, it could set the value of the flag to negative, meaning that other thread(s) cannot acquire the lock, resulting in starvation of the other thread(s).

Comparing Lock Algorithms

[4] The lock types used in the go code consist of **`unit32`** variable types as the atomic library only supports those types. The Turn Ticket uses the atomic **`Add`** function, while the compare and swap lock uses the atomic **`compare and swap function`**. [5] The code is very similar to its original C structure excluding the need for a queue. Within the main function, the **`goroutines`** array holds the amount of go routines per lock to be tested, with the amount doubling each time as we go through the list. We start with 2 routines per test, then 4 and stop at 16. [6] The benchmark function is called for each lock type as we go through each number in the list, and the amount of go routines are executed. Each go routine competes for the lock 1000 times, with some arbitrary work (*a for loop*) being completed in between **`lock()`** and **`unlock()`**. The iteration variable controls that number of times/iterations each go routine calls the lock. We use a `WaitGroup` to

ensure that all goroutines are executed before the benchmark function is done executing. [7] The **`start`** variable is used to capture the time before the go routines start running, and the **`end`** variable captures the end time. The benchmark function returns the difference between those times (*the runtime*) in nanoseconds. The loop in the main function then prints out the runtime of each lock type.



```
Goroutines: 2
Ticket Lock Duration: 128291 nanoseconds
CAS Lock Duration: 76708 nanoseconds
-----------------------------
Goroutines: 4
Ticket Lock Duration: 504708 nanoseconds
CAS Lock Duration: 173292 nanoseconds
-----------------------------
Goroutines: 8
Ticket Lock Duration: 1893167 nanoseconds
CAS Lock Duration: 740666 nanoseconds
-----------------------------
Goroutines: 16
Ticket Lock Duration: 6030083 nanoseconds
CAS Lock Duration: 4656292 nanoseconds
-----------------------------
```

[8] Results show that the CAS Lock is faster when there are fewer goroutines (2, 4, 8), but as the number of goroutines increases to 16, the Ticket Lock starts to perform better than CAS Lock. CAS Lock is faster with fewer goroutines because it has lower overhead and doesn't involve waiting in a queue. However, as goroutines increase, it suffers from retry overhead due to contention. Ticket Lock performs better with more goroutines because it queues up requests in a fair manner, reducing retries and minimizing contention, but at the cost of slight queuing overhead. This becomes more efficient at higher contention levels.

Real World Example

A large group of people waiting in line to enter a door is more efficient than the same amount of people trying to cram or barge through a door, especially if more and more people join the line. The line is a metaphor for the Ticket Lock, while the angry crowd is a metaphor for the Compare & Swap Lock.

Instructions

The go code must be run in an IDE such as Visual Studio (*preferred*) or GoLand. The `go.mod` file must be in the directory for testing to work. Additionally, the code must be run locally on computer hardware rather than the Go Playground. This is important for the time functions to work. If ran in Go Playground, the times will all be zero (0).

References

[1] A. Shriram, "Locks and Mutual Exclusion." https://www.cs.sfu.ca/~ashriram/Courses/CS431/assets/lectures/Part5/084_31_Locks.pdf. [Accessed: Feb. 16, 2025]. Covers the importance of guarding critical sections in lock implementations.

[2] "Synchronization II - Cornell University." https://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/25-sync-i-g.pdf. [Accessed: Feb. 16, 2025]. Provides insights into locks, race conditions, and guard variables.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, "Locks." https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf. [Accessed: Feb. 16, 2025]. Detailed discussion on different lock implementations and their correctness.

[4] "Load-Link/Store-Conditional (LL/SC) - Wikipedia." https://en.wikipedia.org/wiki/Load-link/store-conditional. [Accessed: Feb. 16, 2025]. Explanation of LL/SC and its role in synchronization.

[5] "Hardware Synchronization Algorithms," GeeksforGeeks. https://www.geeksforgeeks.org/hardware-synchronization-algorithms-unlock-and-lock-test-and-set-swap/. [Accessed: Feb. 16, 2025]. Overview of various hardware-level synchronization primitives.

[6] "Synchronization - CS 341 (University of Illinois)." https://cs341.cs.illinois.edu/coursebook/Synchronization. [Accessed: Feb. 16, 2025]. Explains different synchronization methods, including spinlocks and ticket locks.

[7] "Thread Synchronization and Locking Mechanisms," Carnegie Mellon University. https://www.cs.cmu.edu/~410-s05/lectures/L18_Synchronization.pdf. [Accessed: Feb. 16, 2025]. Discusses real-world issues with locks, including deadlocks and starvation.

[8] "Ticket Locks and Performance Comparisons," MIT PDOS. https://pdos.csail.mit.edu/6.828/2021/lec/l-atomic.pdf. [Accessed: Feb. 16, 2025]. Explains ticket locks and compares their efficiency against spinlocks.