Operating Systems – HW2

Group Members: Katrina Wong and William Quinterogomez

Part 1: Lock Analysis

If you remove the code about guard, the new algorithm will not be correct. It will be incorrect since the malicious scheduler violates mutual exclusion, making the two threads hold the lock at the same time.

Part 2: Lock Analysis 2

I believe that the new lock algorithm is incorrect since the malicious scheduler, double unlock, can set the flag < 0, which allows multiple threads to get the lock simultaneously. It breaks mutual exclusion. This corrupts the lock state.

Part 3: Lock Analysis 3

The wake-up/waiting race problem is derived due to OS enforcing fairness in the CPU. There are timing interrupters that stop a thread from proceeding and yielding to another thread. We assume the worst-case scenario, malicious scheduler. The setpark() function allows the OS to know that this thread intends to sleep, so if a thread change happens then it will remember to call park() function for the thread that was expected to put it to sleep.

Part 4: Lock Analysis 4

In the GitHub repo: https://github.com/Katw-1/OS-HW2

Ticket lock:

When the program starts, it initializes a ticket lock and launches two goroutines that both attempt to acquire the same lock. The first goroutine acquires the lock immediately and holds it for two seconds using time.Sleep, simulating a long critical section. Shortly afterward, the second goroutine starts and records the current time before calling lock(&m) because the lock is already held, it spins in the ticket lock's loop until the first goroutine releases it. Once the lock becomes available, the second goroutine acquires it and computes how much time has passed since it first attempted to enter, which represents its waiting time. This duration is printed to the console, demonstrating how long the second thread had to wait for access to the lock. A WaitGroup ensures that the main function waits until both goroutines finish before exiting.

Test and Set:

This program demonstrates how a queue-based lock works and how to measure the time a second thread waits to acquire it. First, the program initializes a shared lock structure and then starts two goroutines that both attempt to use the same lock. The first goroutine acquires the lock immediately and deliberately holds it for two seconds by sleeping, which simulates a long critical section and ensures contention. Shortly after, the second goroutine records the current time and then calls the lock function; because the lock is already held, this goroutine is placed into the lock's waiting queue and blocked using the park mechanism. When the first goroutine releases the lock, it wakes the next waiting goroutine in the queue, allowing the second goroutine to proceed. At that moment, the second goroutine calculates how much time has passed since it first attempted to acquire the lock, which represents the waiting time. This measured duration is then printed, showing how long the second thread was blocked before it could enter the critical section.