Parallel and Distributed Computing
DD2443 - Pardis24
Exercises for Lecture 7

Name: Casper Kristiansson

September 19, 2024

Exercise 1

Question

Exercise 7.1. Fig. 7.33 shows an alternative implementation of CLHLock in which a thread reuses its own node instead of its predecessor node. Explain how this implementation can go wrong, and how the MCS lock avoids the problem even though it reuses thread-local nodes.

Answer

This implementation can go wrong due to the way it handles the re-usage of the thread-local nodes. This is because each thread resumes its QNode for every time it wants to acquire and use the lock. This means that if a thread unlocks and sets the locked=false it will also be used by the same node next time it wants to acquire the lock.

The reason why this is an issue is due to a race condition which can happen when a thread releases the lock and sets the locked=false and another thread tries to acquire it. This could lead to if a thread enters the lock method while another tries to acquire it a situation could be that the new thread could incorrectly think the lock is free, resulting in multiple threads entering the critical section simultaneously.

The reason why the usage of MCS can avoid this problem is because the locks avoid this specific problem by keeping track of each thread's state in separate nodes. This ensures proper hand-off and avoids the issue caused by node reuse.

Exercise 2

Question

Exercise 7.2. Imagine n threads, each of which executes method `foo()` followed by method `bar()`. Suppose we want to make sure that no thread starts `bar()` until all threads have finished `foo()`. For this kind of synchronization, we place a *barrier* between `foo()` and `bar()`.

First barrier implementation: We have a counter protected by a test-and-test-and-set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n.

Second barrier implementation: We have an n-element Boolean array $b[0..n-1]$, all initially `false`. Thread 0 sets $b[0]$ to `true`. Every thread i, for $0 < i < n$, spins until $b[i-1]$ is `true`, then sets $b[i]$ to `true` and then waits until $b[n-1]$ is true, after which it proceeds to leave the barrier.

Compare (in 10 lines) the behavior of these two implementations on a bus-based cache-coherent architecture. Explain which approach you expect will perform better under low load and high load.

Answer

The problem is regarding the implementation of a synchronization barrier in two different ways so that no thread can proceed to the `bar()` method until all threads have finished executing `foo()`.

First barrier implementation

For the implementation with the first barrier, each thread will have frequent reads and writes to the same counter. Because due to the cache of the counter, this implementation of a first barrier will work when the load is low but on higher loads it could lead to all threads needing to queue to access the same memory location of the counter.

Second barrier implementation

The second barrier implementation is based on that each thread interacts with a specific index of a boolean array. While this implementation still needs to synchronize with other threads. As the last one this one has good performance on low but on high load, it will be better due to the reduced caching issue that existed in the first barrier.

Because of this, the second implementation is expected to perform better under high load due to reduced cache contention, but under low load, both may perform equally.