Usage Fair Delegation Styled Lock

by

Hongtao Zhang

A thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science

(Computer Science)

at the

UNIVERSITY OF WISCONSIN-MADISON

Fall 2024

This thesis was completed under the direction of:
Remzi Arpaci-Dusseau, Professor, Computer Science

Signature of Professor: ______________________________ Date: ____________

to everyone I meet

Acknowledgments

Abstract

In multi-threaded enviroment, resources sharing and synchronization is an important topic. Locks are the most common synchronization technique for such problem. There are two fundamental properties a good lock want to achieve: 1) performance 2) fairness. Specifically, latest research emphasizes the need of usage fairness. Threads are not only allowed to enter the lock with different frequency, but also may use the lock for different amount of time.

On the first glance, it seems that these two properties are contradictory. Can we have a high performance lock while providing usage fairness?

To answer the above question, this dissertation proposes that lock should be designed as delegation: thread should not execute their critical section directly, but delegate the execution to a combiner. This thesis shows that under the setting of delegation styled lock, we can have a usage-fair lock without sacrificing performance. I introduce two types of fair delegation styled locks: 1) non-work-conserving `FC-Ban` and `CC-Ban` 2) work-conserving `FC-PQ`. My evaluation shows that all three of them can achieve high throughput and usage fairness simultaneously, while `FC-PQ` outperforms the other two when the lock is heavily contended while threads are having large non-critical section.

Contents

Lists of Figures

Lists of Tables

1

Introduction

In the current landscape of computational technology, the focus to enhance Central Processing Unit (CPU) performance has transitioned from increasing clock speeds to multiplying core counts. This evolution has given rise to multi-core architectures, which have become ubiquitous across computer systems. The scalability of applications on such multi-core infrastructures is predicated on Amdahl's Law, which postulates that the theoretical maximum improvement achievable through parallelization is limited by the code that must remain sequential.

A principal challenge in parallel computing is thread coordination via shared resources. Lock-based synchronization mechanisms are widely employed to ensure mutual exclusion and are critical for threads to communicate accurately and reliably [1, 5]. These synchronization points, however, are often a source of contention and can become performance bottlenecks in a concurrent execution environment [2]. Theoretically, the synchronization duration should be invariant with respect to the number of threads; yet, contention for locks often leads to a serious degradation in performance that is disproportionate to the increase in thread count [2, 4, 5].

1.1 Delegation-styled locks

Delegation-styled locks have emerged as a innovative solution aimed at boosting synchronization efficiency by minimizing contention and the associated overhead of data movement. Instead of each thread compete for a lock to execute their critical section, threads package their critical sections into requests and entrust them to a combiner, which processes these requests and returns the results. There are two predominant forms of delegation-styled locks: *combining* synchronization [2–4] and *client-server* synchronization [6, 10]. Combining locks allow for dynamic selection of the combiner role amongst the participants, whereas client-server locks dictates a consistent server thread to manage all requests. Empirical evidence suggests that this technique can outperform traditional locking mechanisms, even approaching the ideal of sequential execution efficiency regardless of number of threads.

1.2 Usage fairness

Newly conducted studies have introduced concerns regarding scheduler subversion when locks are implemented without a sophisticated fairness mechanism or are limited to fairness at the point of acquisition [8]. This is particularly problematic when threads exhibit imbalanced workloads within their critical sections, as the presence of a lock can disrupt the CPU's scheduling policy, which intends to allocate equitable processing time to concurrent threads. Envision a scenario where interactive threads engaging with users are in contention with batch threads performing background tasks, all synchronized by a lock. In the absense of principle of usage fairness, the interactive threads may suffer from inordinate delays in lock acquisition, thereby subverting the CPU scheduler's objective of ensuring prompt response times for interactive tasks. Moreover, the issue is magnified in the context of delegation-styled locks, where the elected combiner thread may be burdened with an unequal share of work. If an interactive thread is chosen as the combiner, it could lead to severe latency issues for the user, thus diminishing the attractiveness of combining locks in systems with disparate workloads.

1.3 Delegation-styled locks with Banning

TODO

1.4 Delegation-styled locks with a serialized scheduler

TODO

Background

2.1 Concurrency, Synchronization & Mutual Exclusion

In modern computing systems, concurrency is fundamental to achieving high performance and resource utilization. When multiple computations need to execute, they can be interleaved on a single processor to create the illusion of simultaneous execution, or run truly in parallel across multiple processors. This interleaving allows the processor to switch between different tasks when one is blocked or waiting, making efficient use of system resources.

These interleaved executions introduce challenges when multiple threads or processes need to access shared resources. People make assumptions about results of previous operations. However, the interleaving can cause violations of these assumptions, leading to incorrect program behavior.

Synchronization is the art of coordinating concurrent operations and maintaining program correctness. These mechanisms ensure that concurrent accesses to shared resources follow a proper order and maintain consistency. Without synchronization, concurrent access to shared data can lead to race conditions, where the program's outcome becomes unpredictable and depends on the precise timing of operations.

One of the most common synchronization mechanisms is to adopt *Mutual Exclusion*. *Mutual Exclusion* reconstructs the assumptions by disallowing concurrent access to the shared resource, and thus recovers the sequential behavior. The set of operations that needs to be protected by mutual exclusion is called a *critical section*.

There are various mechanisms to implement mutual exclusion, such as hardware provided atomic instructions, locks, semaphores, or transactional memory. Locks are one of the most commonly used mechanisms to implement achieve mutual exclusion. Only a single owner is allowed to own the lock at a time. By forcing threads to acquire the lock before entering the critical section, we can ensure that the critical section is executed sequentially without interference from other threads.

2.2 Lock

This section reviews the common lock primitives.

2.3 Common Lock Primitive Implementations

2.3.1 Naive Spinlock

2.3.2 Pthread Spinlock (test and test lock)

2.3.3 Exponential Backoff Spinlock

2.3.4 Pthread Mutex

2.3.5 Ticket Lock

2.3.6 MCS & K42 variant

2.4 Lock Usage

2.4.1 Scheduler Subversion

2.4.1.1 Imbalanced Scheduler Goals

2.4.1.2 Non-Preemptive Scheduling

2.5 Usage-Fair Lock

2.5.1 Scheduler-Cooperative Locks

2.5.1.1 u-SCL

2.5.1.2 k-SCL

2.5.1.3 RW-SCL

2.5.2 CFL

[7]

2.6 Delegation Styled Locks

2.6.1 Combine Style Locks

2.6.1.1 Flat Combining

[4]

2.6.1.2 CCSynch/DSM-Synch

[2]

2.6.2 Client-Server Styled Locks

TODO

2.6.2.1 RCL

[6]

Not Sure whether to include this

2.6.2.2 ffwd

[10]

Not Sure whether to include this

2.7 Lock-Free Data Structures

2.7.1 Linked List

[5]

2.7.2 Skip-List

[5, 11]

2.7.3 Priority Queue

[5]

2.7.4 MPSC Channel

[9]

Usage Fair Delegation Styled Locks

3.1 Banning Locks

3.1.1 Flat Combining (with Banning)

[4]

3.1.2 CCSynch with Banning

[2]

3.2 Naive Priority Locks

3.2.1 FC-Skiplist

[11]

3.3 Serialized Scheduling Locks

Charateristics & Experiments

4.1 Throughput

4.2 Latency of single acquire

Future Work

Bibliography

[1] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2018. *Operating systems: Three easy pieces.* Arpaci-Dusseau Books, LLC.

[2] Panagiota Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012. 257–266.

[3] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. 2023. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with {TCLOCKS}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023. 1–16.

[4] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, 2010. 355–364.

[5] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[6] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: Migrating {Critical-Section} execution to improve the performance of multithreaded applications. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012. 65–76.

[7] Jonggyu Park and Young Ik Eom. 2024. Locks as a Resource: Fairly Scheduling Lock Occupation with CFL. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24)*, 2024. Association for Computing Machinery, Edinburgh, United Kingdom, 17–29. https://doi.org/10.1145/3627535.3638477

[8] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. 2020. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. 1–17.

[9] Erik Rigtorp. 2021. Optimizing a ring buffer for throughput. Retrieved from https://rigtorp.se/ringbuffer/

[10] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017. 342–358.

[11] Nir Shavit and Itay Lotan. 2000. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000. 263–268.