A Comparative Study of Locking Techniques: Fairness and Throughput Performance Evaluation

Ching-Shen Lin
Department of Computer Science
National Chung Cheng University
Chiayi, Taiwan (R.O.C.)

Zhao-Ting Lin,
Department of Computer Science
National Chung Cheng University
Chiayi, Taiwan (R.O.C.)

Shiwu Lo
Department of Computer Science
National Chung Cheng University
Chiayi, Taiwan (R.O.C.)

***Abstract*— This paper examines the fairness and throughput of different locking methods in a shared resource access scenario. We compared popular spin lock algorithms, such as `GNU spinlock`, ticket lock, and MCS spin lock, and chose `GNU/Linux's mutex` and `GNU/Linux's binary_semaphore` for mutex and semaphore implementations, respectively. We evaluated the locks in a microbenchmark from three perspectives: critical section size, remainder section size, and oversubscription issue. We measured locks per second for various critical section sizes, evaluated the locks under different remainder section sizes, and assessed their performance in oversubscription scenarios. Each microbenchmark was executed twice in different perspectives, once without file I/O in the critical section and once with file I/O operations.**

***Index Terms*— Locking mechanisms, Multithreading, Synchronization, Fairness, Performance evaluation**

I. Introduction

Locking is a crucial technique used in parallel processing to synchronize access to shared resources. The GNU Linux uses two common locking strategies: spin locks and blocking locks. Spin locks use busy waiting to achieve synchronization and are efficient for short critical sections. In contrast, blocking locks put the requesting thread to sleep if the lock is not available and are more efficient for longer critical sections. As multi-core systems become more prevalent, the critical section problem arises, where only one process can access a code segment at a time. Atomic operations are used to manipulate locks, and busy waiting occurs when a processor is accessing a data structure. This article presents an implementation, analysis, and conclusion of the effectiveness of spin locks and blocking locks. Spin locks are implemented in C language, while blocking locks are implemented using `GNU/Linux's mutex` and `GNU/Linux's binary_semaphore` . The performance and fairness of the spin locks are evaluated, and the pros and cons of both locking mechanisms are discussed.

II. Implementation

In the spin lock phase, we used various spin lock algorithms, such as the raw spinlock, the `GNU/Linux's spinlock` provided by the GNU C library, ticket lock, and MCS spin lock. In contrast, for the mutex and semaphore implementation, we chose `GNU/Linux's mutex` and `GNU/Linux's binary_semaphore`, respectively. The `GNU/Linux's binary_semaphore` restricted the number of threads that could enter the critical section to one.

A. Testing Environment

The testing environment for this study includes hardware with an AMD Ryzen Threadripper 2990WX 32-Core Processor, with a total of 64 *virtual core* (*virtual core* per core is 2) and an x86_64 architecture. The compile environment used was g++ version 10.3.0, targeting the AMD Zen+ and x86_64-linux-gnu platform with a POSIX thread model.

B. Locking programs

We writed the following locks algorithm and compare them in Evaluation Section.

1) **Raw spinlock**: The Raw Spinlock algorithm uses *test and test-and-set* (TTAS) to repeatedly check and set the lock status. This ensures exclusive access to the critical section by a single thread at a time.
2) **Plock**: We used the GNU Pthreads spinlock function provided by C library, which also uses test and test-and-set.
3) **Ticket**: Threads wait until the "service number" equals their ticket number before entering the critical section. However, this method can consume significant Network-on-Chip (NoC) bandwidth as waiting threads continuously query the "grant" variable in a loop.
4) **MCS**: The MCS spinlock algorithm queues waiting threads in a linked list. Upon leaving the critical section, a thread sets the 'locked' variable of the next thread to false to signal that it can enter the critical section.
5) **Pthread Mutex**: We used the `GNU/Linux's mutex` function from the GNU pthread library. If the mutex is already locked, the calling thread will be blocked until it acquires the mutex. Upon completion of the function, the mutex object will be locked and owned by the calling thread.
6) **Binary Semaphore**: We used a binary semaphore with C library to restrict access to one thread at a time. `sem_wait()` decreases the semaphore value if greater than zero, blocking the thread otherwise. `sem_post()` reduces the semaphore by one and resumes waiting threads when the value increases.

C. Testing Programs - Microbenchmarks

We conducted a fair quantitative analysis of locking methods using a controlled microbenchmark, with each thread

assigned to a core. The benchmark included a critical section where the lock thread requested entry using a *while* loop, read and wrote to shared data, and simulated non-critical work in a remainder section. The benchmark ran in two ways, with and without file input/output operations, and ensured shared data accuracy by verifying the value of each element in sharedData. It should be noted that the file I/O operations in our study were used to illustrate I/O-related operations, and we did not delve into the implementation details of the `fprintf()` function in the code.

Algorithm 1 The Microbenchmark

```
while(1):
    spin_lock();
    // critical section start
    for(each element in sharedData):
        if(with file I/O):
            element += 1;
            fprintf(test_file);
        if(without file I/O):
            element += 1;
    // end of critcal section
    spin_unlock();
    // remainder section
    rs_start = clock_gettime();
    do{
        t = clock_gettime();
    }while(t-rs_start < RS_size*rand(0.85~1.15));
```

III. Throughput Evaluation

In the throughput evaluation, we evaluated the locks used in the microbenchmark from three different aspects:

1) **Critical Section Size**: We measured the amount of work done by a thread while holding the lock using the number of times the `for` loop was executed in the critical section.
2) **Remainder Section Size**: We measured the time it took for threads to complete non-critical work after accessing the shared resource using `clock_gettime()`. We tested different remainder section sizes to evaluate how the locks performed under varying loads of non-critical work.
3) **Oversubscription issue**: In oversubscription, there are more threads than available cores to execute them, causing threads to wait for their turn, leading to increased latency and decreased performance. We evaluated the locks for oversubscription by running the microbenchmark on a system with more threads than cores and measuring their performance.

We ran the microbenchmark twice for each aspect, with and without file I/O in the critical section, to evaluate the locks' performance under different workloads. We compared and analyzed the results to identify the best-performing locks in each scenario.

A. Critical Section Size

The critical section size refers to the number of loop iterations executed while holding the lock, and it impacts the lock's performance by determining how long a thread holds the lock and how many threads are blocked from accessing the shared resource. A small critical section size can lead to poor performance due to lock contention, while a large one can decrease program throughput by causing threads to wait for the lock for an extended period.

1) Without File I/O Operations: Figure 1 shows locks per second on the y-axis and critical section size on the x-axis. The graph displays different lines for each locking mechanism's performance under non-file I/O operations. Spinlocks have better performance for small critical section sizes but decrease when it exceeds 10,000, while semaphore and mutex locks perform slightly better for larger critical section sizes. Other locking mechanisms have stable performance regardless of the critical section size.

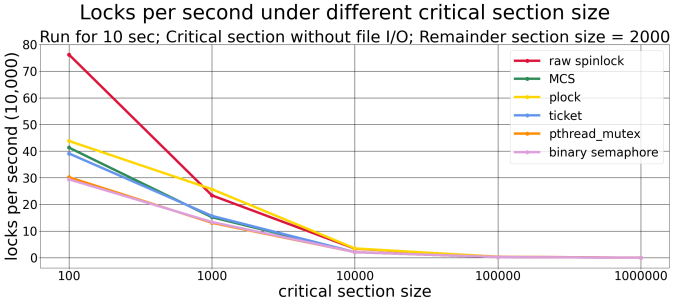


Figure 1. Locks per second under different critical section size

2) With File I/O Operations: Figure 2 shows that Semaphore and Mutex significantly outperform spinlocks when the critical section size exceeds 10,000.

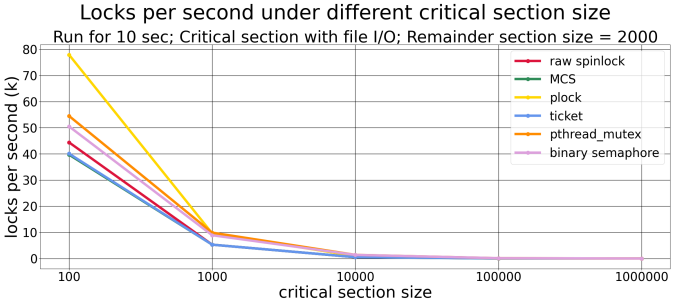


Figure 2. Locks per second under different critical section size

B. Remainder Section Size

The remainder section size refers to the waiting time after leaving the critical section. A higher size means a longer waiting time, indicating a low load condition, while a smaller size indicates a high load condition where the thread competes for resources.

1) Without File I/O Operations: In this evaluation, we fixed the critical section size at 100 and measured the number of locking times under different remainder section sizes in Figure 3. We can observe that semaphore and mutex locks are inefficient without file I/O operations.

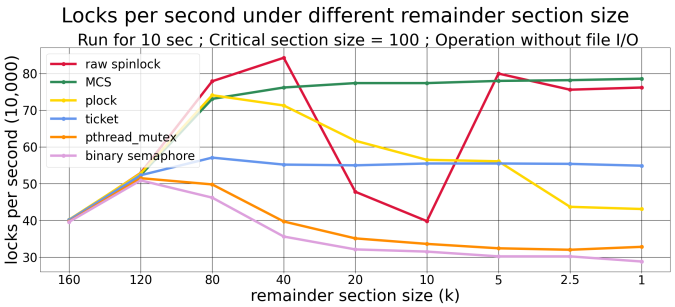


Figure 3. Locks per second under different remainder section size

2) With File I/O Operations: Figure 4 shows the performance of locking mechanisms with fixed critical section size at 100 and varying remainder section sizes. Semaphore and mutex locks gradually improve in performance under high-load conditions.

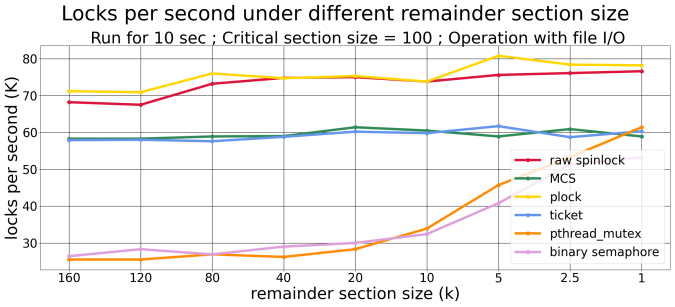


Figure 4. Locks per second under different remainder section size

C. Oversubscription issue

Oversubscription's performance depends on two factors: whether the lock-holding thread is scheduled and if the algorithm specifies the order of the next thread entering the critical section.

1) Without File I/O Operations: In Figure 5, the performance results are presented with a fixed remainder section size of 2000 and a critical section size of 100. The mutex and semaphore outperform most spinlocks with a high number of threads, which may be due to the optimization of the operating system scheduler for handling mutex locks. In contrast, the slow down of MCS and ticket spinlock as the number of threads increases can be attributed to the overhead of maintaining their respective queues. As the number of threads increases, the length of these queues also increases, resulting in additional overhead for the spinlocks. Raw spinlocks, on the other hand, do not have any queue-based overhead, making them better suited for scenarios with a large number of threads.

2) With File I/O Operations: Figure 6 indicates that spinlock's performance decreases as the number of threads increases, while mutex's performance slightly increases. The performance improvement of the GNU's pthread mutex implementation with increasing thread counts is due to the

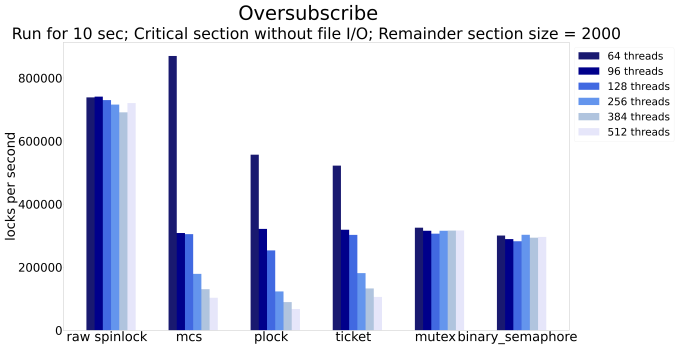


Figure 5. Locks per second under different threads number

architecture of the implementation, which uses several techniques to reduce contention and improve scalability. These techniques include a combination of spinlock and adaptive mutex algorithms, priority inheritance to prevent priority inversion, and advanced memory management to minimize cache invalidation. Overall, these techniques result in a well-balanced trade-off between scalability and fairness. This suggests that mutex may be a better locking mechanism for file operations under high thread loads.

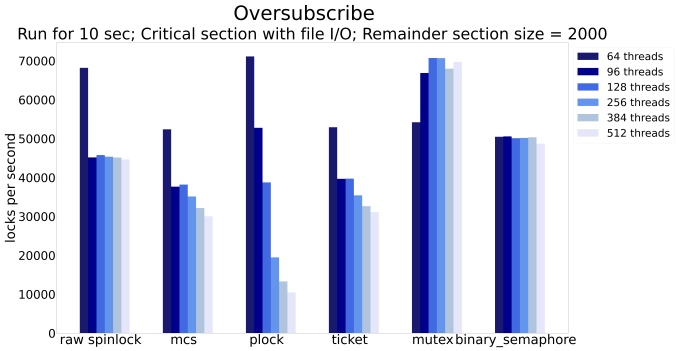


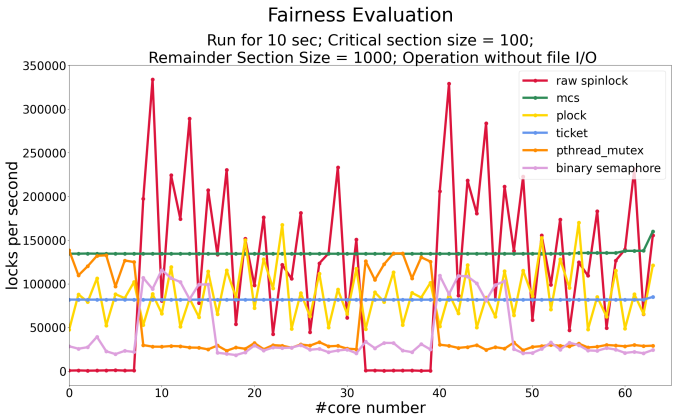
Figure 6. Locks per second under different threads number

IV. Fairness Evaluation

The fairness of a locking algorithm is crucial for preventing CPU starvation issues and maintaining overall system performance. Without fairness, some threads may acquire the lock more frequently, while others are blocked for a long time, leading to decreased performance and making it difficult to accurately measure performance between processor cores. Therefore, it is important to monitor and maintain the fairness of locking algorithms in order to ensure the stability and efficiency of the system.

A. Undersubscription Issue

Figure 7 monitored the critical section access of threads to evaluate lock fairness for non-file and file I/O operations. The ticket and mcs spinlocks were fair to each thread, while all locks except raw spinlock and `GNU/Linux's spinlock` were fair for file I/O. Raw spinlock and plock were extremely unfair, while mutex and semaphore were somewhat unfair but not as much as plock.



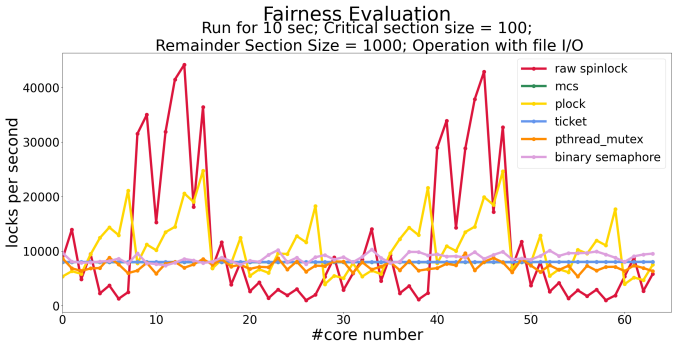
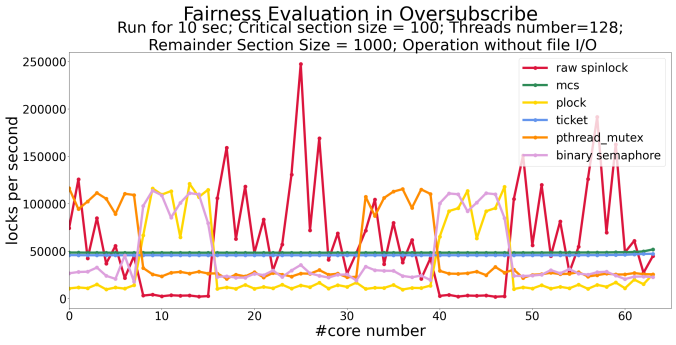


Figure 7. Fairness evaluation

B. Oversubscription issue

We evaluated lock fairness under oversubscription with 128 threads in Figures 8. The non-file operation results were similar to those in Figure 7, but for file operations, `GNU/Linux's mutex`'s fairness was extremely low in oversubscription, similar to `GNU/Linux's spinlock`. The unfairness of the pthread mutex under file I/O operation in oversubscription is due to its hybrid structure, which uses a spin lock and a semaphore. In low contention scenarios, the spin lock is used, but in high contention scenarios, it is switched to a semaphore, which can lead to additional overhead and unfairness.



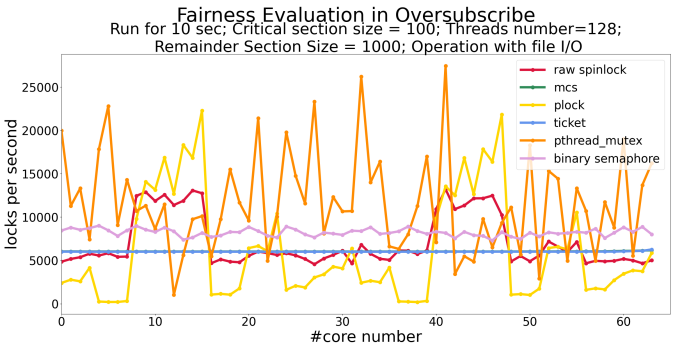


Figure 8. Fairness Evaluation in Oversubscribe

V. Conclusion

After conducting our experiments and analyzing the results, we can conclude that the choice of locking mechanism is crucial in determining the performance and fairness of shared resource access scenarios. The spinlock algorithms showed better performance than other locks in smaller critical section sizes. However, when the critical section size increases, semaphore and mutex locks tend to outperform ticket and MCS spinlocks. We also found that the type of operation being performed on the shared resource can greatly affect the performance of the locking mechanism. Specifically, when performing file I/O operations, mutex and semaphore locks outperform spinlocks. In addition, we observed that the oversubscription issue negatively affects the performance and fairness of all locking mechanisms. In particular, `GNU/Linux's mutex` under file I/O operation in oversubscription is found to be extremely unfair. Our findings suggest that in order to achieve optimal performance and fairness in shared resource access scenarios, the selection of locking mechanism should be carefully considered based on the critical section size, the type of operation being performed, and the potential for oversubscription.

References

[1] Shiwu Lo, Han-Ting Lin, Yaohong Xie, Lin Zhaoting, Yu-Hsueh Fang, Jingshen Lin, Ching-Chun Huang, Kam Yiu Lam, and Yuan-Hao Chang, "RON: One-Way Circular Shortest Routing to Achieve Efficient and Bounded-waiting Spinlocks," accepted by 17th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI), 2023.

[2] Love, R. (n.d.). Kernel Locking Techniques. Retrieved from https://www.kernel.org/doc/gorman/html/understand/understand005.html

[3] Singh, S. K., Kumar, K., & Kumar, P. (2014). A comparative study of locking techniques for multi-threaded applications. International Journal of Computer Applications, 105(9), 18-23.

[4] Umesh, R. N., & Venkatesh, R. (2013). Performance Evaluation of Mutexes, Spin Locks and Semaphore. International Journal of Computer Applications, 79(11), 7-13.

[5] Michael, M., & Scott, M. (1996). Performance Evaluation of Lock-Free Algorithms. Proceedings of the International Symposium on Parallel and Distributed Processing, 1996.

[6] Contavalli, C. L., Reinhold, P., & Soffa, M. L. (2012). Performance Comparison of Spinlocks and Mutexes in High-Contention Scenarios. In 2012 IEEE 26th International Parallel and Distributed Processing Symposium (pp. 553-564). IEEE.

[7] Mellor-Crummey, J. M., & Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS), 9(1), 21-65.

[8] Anderson, T. E. (1990). The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1(1), 6-16.