Generalised Hierarchical Fairness for Schedulable Entities Using Scheduler Cooperative Locks

Gunuru Manoj Taraka Ramarao, Prof. Abhishek Bichhawat
Indian Institute of Technology, Gandhinagar
manoj.gtr@iitgn.ac.in

Abstract

In the realm of system scheduling, an issue can arise from the imbalanced usage of locks, resulting in an unjust allocation of resources among processes or threads. This mainly occurs due to significant contention of locks and unequal size of critical sections of threads or processes [1]. Scheduler Cooperative Locks (SCLs) ensure fair lock usage among competing threads or processes. In this work, we design a hierarchical lock that can ensure fairness at different levels. We develop an algorithm that provides fairness to schedulable entities and implemented it on top of scheduler cooperative locks. Initial results show that we receive significant improvement in fairness when allocating resources.

2012 ACM Subject Classification Operating Systems, locks

Keywords and phrases Scheduler Cooperative Locks(SCLs), fair locks, scheduling

1 Motivation

In today's world, applications and services rely heavily on various resources such as memory, networks, storage, and computation. To maintain a fair distribution of these resources among applications, effective scheduling is crucial. Each application may require different levels of resource usage, and it is vital to ensure fairness at each level to achieve optimal overall performance. Therefore, it is imperative to implement fair allocation mechanisms to ensure that no application receives more than its equitable share.

Traditional locks, such as mutexes and spin-locks fail to provide fairness among processes or threads due to their limitations in handling unequal critical section sizes and higher contention of locks. Even though ticket locks ensure fairness in lock-acquisition orders, they also cannot guarantee fairness in holding the lock. Consequently, these locks can lead to potential starvation and overall lower performance. Scheduler Cooperative Locks (SCLs) provide fair allocation of locks for each thread or process, irrespective of their lock usage patterns, by providing a static time slice. For instance, a thread can acquire SCL multiple times when it is the owner of the lock-slice. However, if a thread uses the lock for more than the allotted time, it will be penalized. Nonetheless, thread-level fairness alone is insufficient. Different schedulable entities, such as virtual machines, containers, NUMA nodes, processes, threads, and their combinations, demand an arbitrary share of a common resource. The demands of these entities vary from individual to set level. Therefore it is crucial to ensure hierarchical fairness.

2 Problem Statement

SCLs ensure fairness by lock usage accounting at an entity level, penalising dominant users and dedicating a window opportunity time to each entity. We define a level where a particular set of threads can be categorized based on specific criteria. The criteria could be the execution type of threads, such as reading or writing, and the origin to which they belong, such as virtual machines, containers and processes. SCLs ensure fairness at one level.Our primary objective is to design a generalised lock that can guarantee fairness across all hierarchical levels, from the highest thread-distinguishing level down to the lowest possible thread-distinguishing level. In other words, we aim to create a lock that can ensure equity across all levels of the hierarchy.

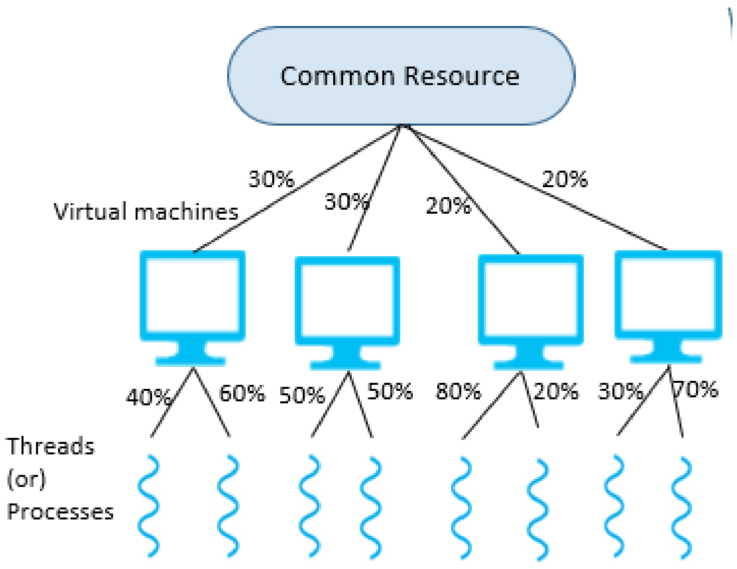

Figure 1 Hierarchical fairness problem

Figure 1 describes one of the scenarios where hierarchical fairness is needed. Virtual machines need an arbitrary share of a common resource. Threads or processes belonging to a particular virtual machine may demand an arbitrary share within the allocation that the virtual machine aims to acquire.

3 Background and Theory

SCLs and Fairness

To prevent scheduler subverting issues, SCLs have been introduced. These locks promote fairness by allotting a specific time window or lock opportunity time (LOT) for each competing entity to use the lock as many times as needed, possibly zero. This lock ensures that all processes have an equal chance to access the lock and prevents all processes from monopolizing it. [1].

$$LOT(i) = \sum Critical\_Section(i) + \sum Lock\_Idle\_Time(i)$$

SCLs are designed to allocate locks exclusively to a single thread within a particular time window, keep track of lock usage, and penalize them for excessive usage. These locks are aimed at aligning with scheduling goals rather than undermining them. There are three types of SCLs that are well-suited for different scenarios.

The first type is User-space Scheduler Cooperative Lock (u-SCL), which can serve as a replacement for the standard mutex in user space. The second type is Read and Write Scheduler Cooperative Lock (RW-SCL), which is used for implementing reader-writer locks.

The third type is kernel Scheduler Cooperative Lock (k-SCL), which is designed specifically for use at the OS-kernel level. In this context, we mainly focused on u-SCL to develop a generalized hierarchical fair lock.

User space - Scheduler Cooperative Lock (u-SCL)

u-SCL maintains a record of weights that determine the proportion of lock usage, the duration of lock usage, and the penalty period for excessive lock usage. To manage this information, u-SCL uses thread structures from the pthread library. This lock is designed to work efficiently with any number of threads that have arbitrary lock usage patterns.

To promote fairness, u-SCL stores information about the total weight of threads and updates it whenever a new thread acquires a lock or an existing thread terminates. This updating process ensures that CPU time is allocated equitably among all threads. The lock operates by giving a time slice to each thread, during which it can acquire the lock multiple times. Meanwhile, other threads must wait for their turn until the time slice ends. Once the time slice of the current thread expires, the next waiting thread becomes the new owner of the time slice. u-SCL checks If a thread overuses the lock, then it will be penalized when it tries to access the lock. u-SCL ensures fair lock usage and prevents all threads from dominating the lock.

4 The Hierarchical Fairness Problem Aspects

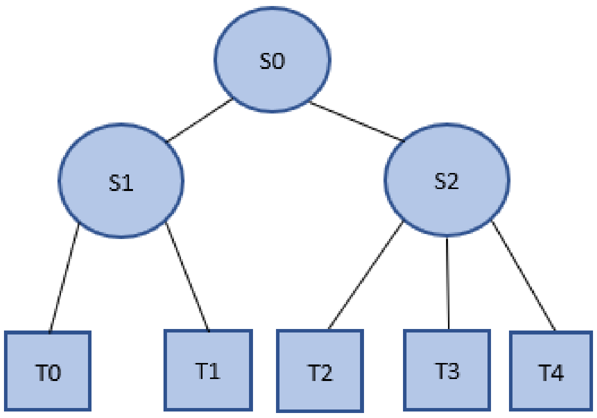

Figure 2 Example of generalised hierarchical fairness lock

We constructed a generalised hierarchal lock using u-SCLs which rely on non-preemptive scheduling. Figure 2 describes one of the simple examples where hierarchical fairness is ensured. S0, S1 and S2 are u-SCLs and T0, T1, T2, T3, T4 are threads. S1 ensures fairness between T0 and T1. S2 ensures fairness among T2, T3 and T4. S0 ensures fairness between the sets {T0, T1} and {T2, T3, T4}. We observed that it is not so trivial to make a generalised hierarchical lock by just making a hierarchy with u-SCLs. There are three main aspects we observed and tried to incorporate those aspects in our implementation. Firstly, S0 needs to schedule S1 and S2. It is necessary to modify the u-sCL lock mechanism to operate at the set level rather than the thread level. Secondly, we need to handle concurrency issues.

Correctly banning threads is crucial in maintaining accurate lock usage accounting and ensuring fairness across all hierarchical levels. Since locks are non-preemptive scheduling

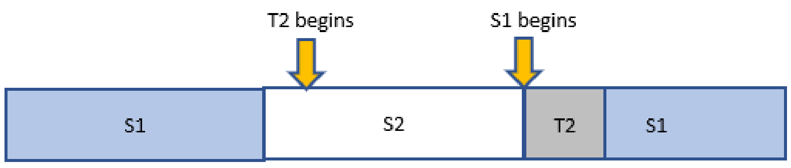

Figure 3 Banning issue in accounting of lock usage in S0

primitives, we cannot interrupt threads when their period as the owner of the slice in the path anywhere from root level to leaf level ends while holding a lock. Figure 3 depicts an example of possible incorrect lock usage accounting at S0, where thread T2 (any of T2, T3, T4) begins in slice S2 allotted by S0 and leaves the lock S2 in slice S1. In such cases, it becomes necessary for S0 to ban all threads that belong to S2 (T2, T3, and T4). It is also important to ensure that lock accounting must only be performed when the slice of the lock is active. Finally, to mitigate the banning issues mentioned above, we added a correction term in lock usage accounting while banning threads at every hierarchical level.

5 The Hierarchical Lock

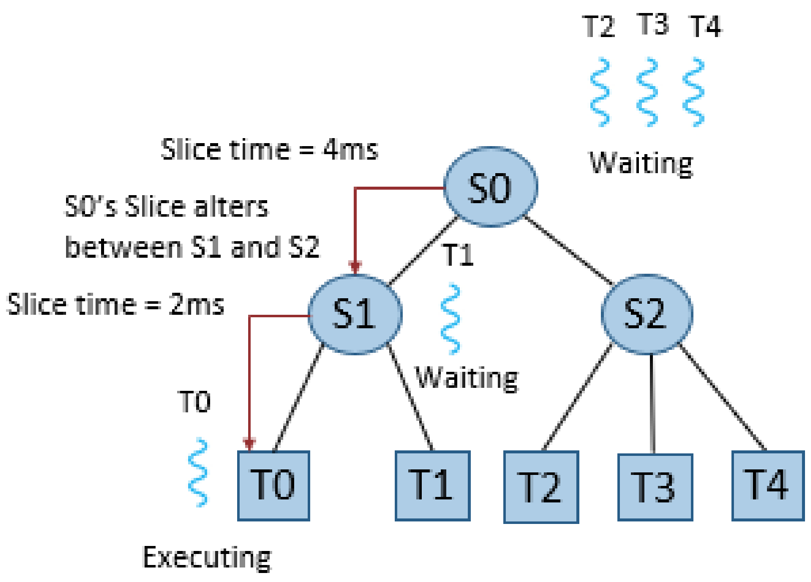

Figure 4 Hierarchical lock with u-SCLs in two levels

We implemented a generalised hierarchical fair lock. Figure 4 depicts one simple example of a generalised lock where we ensured hierarchical fairness by considering the hierarchical fairness problem aspects mentioned above. Every thread tries to move from the root to the leaf to execute its critical sections. Entities at the same level will have the same lock opportunity time. We referred to this lock opportunity time as slice times. Slice times, which are positive integral powers of two, increase from bottom to top in the hierarchical lock. Threads wait at different levels until their slices at those corresponding levels become active. S0 alters the slice between S1 and S2. When the slice of S1 is active, S0 allows T0 and T1 to acquire lock S1 and hinders all other threads. S1 will first allow the least dominant user to access the critical section and make others sleep until their ban is expired and they become the owner of the slice. When the slice of S1 expires, S0 makes S2 the owner of the slice, thereby allowing T2, T3, and T4 to acquire the lock S2. A correction term, which is equal to the number of children in the same level - 1 * time slice at that level is added for banning threads at every level. This term is added to ensure that the ban calculation happens when the slice of a particular node is active.

6 Observations and Results

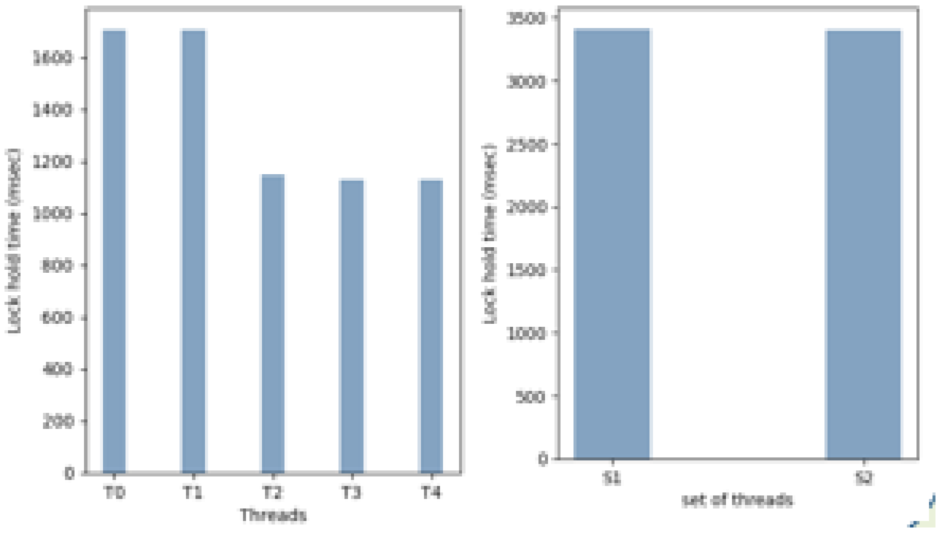

Figure 5 Lock hold times of threads and set of threads 4(b)

We constructed a hierarchy almost similar to the example in Figure 2, with all threads having the same priority and critical section sizes. S1 ensured that T0 and T1 get a fair share of the entire lock. S2 ensured that T2, T3 and T4 get a fair share of the entire lock. Sum of lock hold times of T0 and T1 is equal to the sum of the lock hold times of T2, T3 and T4. It indicates to us that S0 ensures fairness between S1 and S2. We got hierarchical fairness for this example.

7 Conclusion and Future Work

We made an algorithm that can ensure fairness at different hierarchical levels. We started our implementation of the lock and received some promising results when we tested our generalised lock with small benchmarks. As a continuation of this work, we would like to do the following:

- Test the performance of this generalised hierarchical lock on real-world applications like UpScaleDB and KyotoCabinet.
- Compare the performance of this lock with different types of locks, such as cohort locks and spin-locks.
- Currently, our lock has a lot of memory overhead. Minimize the memory overhead.
- We worked on the static time slices at different levels, which are positive integral powers of two. Need to conduct experiments with different static time slices for analysing the

overall performance.

References

1 Yuvraj Patel et al. Avoiding Scheduler Subversion using Scheduler–Cooperative Locks. *Commun. ACM*, 2020. doi:10.1145/3342195.3387521.