CS60038: Advances in Operating Systems Design

Term Project (Intermediate Report)

Group 2

Chappidi Yoga Satwik	19CS30013	Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLOCKS
Rajas Bhatt	19CS30037	Application-Informed Kernel Synchronization Primitives
Seemant G Achari	19CS30057	Using Trāṭr to tame Adversarial Synchronization

Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLOCKS

Introduction

Many programs leverage multi-core processors for higher performance. This is made possible by synchronization mechanisms like locks. Locks serve two purposes: ensuring mutual exclusion of shared data and transfer of shared data between the cores' caches. This data transfer between cache lines increases the time spent in the critical section. As we increase the core count, this latency becomes more prominent, impacting the performance.

Delegation-style locks move the computation rather than the shared data. Some designated server threads combine requests that act on the same memory location, thus bypassing the need for memory transfer. These locks show an improvement in application throughput. A drawback, however, is the API format. These locks accept a function pointer to the critical section, which needs to be transferred to the server thread. To make this change to all locks in the Linux Kernel (180K locks) will be very difficult. This paper presents a solution to achieve delegation-style behavior without needing to change the application logic by means of transparent delegation.

TC Locks: Transparent Delegation

This method captures variable-length critical sections as a set of registers and thread stack. Even though the shared data is accessible to all threads globally, the combiner needs access to the thread-local data of the waiting thread and the instructions required for executing its critical section. The paper addresses the challenge of handling thread-local data and the context of the critical section by relying on three key insights.

1. The execution context of a thread is clearly defined by the hardware, which includes thread-specific CPU registers and the stack containing all the necessary information for executing the critical section.
2. A waiting thread engages in busy-waiting without making any changes to its state once it has sent its request to the combiner. It only exits this state after receiving a response from the combiner.
3. When the lock API is called as a function, it ensures that the hardware pushes the next instruction onto the stack, thereby making the starting address of the critical section available to the combiner for executing the critical section.

Additionally, the waiter thread is given an ephemeral stack as interrupts in kernel space, signals in user space, and the waiter’s parking and wake-up mechanism can access the

waiter's stack during the execution of its delegated critical section. The ephemeral stack only serves as a copy of the stack to attend to immediate needs, while the combiner works on the true stack of the thread.

TC Locks: Workflow

Every lock has a corresponding queue of requests. When a thread acquires a lock, it captures its state, enters the queue, and switches to an ephemeral stack. The first waiting thread in the queue becomes the combiner thread. The next thread that acquires the lock, joins the queue, switches to an ephemeral stack, and waits for a notification from the combiner thread. If the first thread is done processing its critical section, it switches context to the next request in the queue. After execution, it notifies the requestor. This continues till the queue is empty.

TC Locks: Variants

The above workflow describes the working of SpinLock variant of TCLocks. The paper goes on to illustrate other variants such as Blocking and Read-Write. Many optimizations are also introduced to these locks like NUMA awareness, minimizing context switch overhead. There are a few more challenges when it comes to applying these TCLocks to more real-world applications. The fine-grained nature (nested) of locks and out-of-order unlocking in the Linux Kernel are addressed in the paper. Moreover, there are some special contexts (interrupt handlers, non-preemptable contexts) where per-CPU variables are used. As the combiner thread runs on a separate CPU, there is a problem of correctness. The framework simply resorts to using the normal locking mechanism in these special contexts rather than delegation style.

Importance in OS Research

The paper brings the performance of delegation-style locks to a simpler API. Now, large projects like the Linux Kernel can easily migrate to these locks with minimal modification. They can observe improved throughput and reduced time spent in critical sections. The paper provides a family of locks for various use cases and hardware setups.

Application-Informed Kernel Synchronization Primitives

Introduction

OS Developers have started to offer application-oriented kernel customization mechanisms in the context of **scheduling** (as noted in Part A of Assignment 1, in which I/O schedulers like Kyber, CFQ, and Deadline implemented as LKMs were observed), **networking**, **storage** and **accelerators**. However, such customizations are not offered while designing kernel locks which are usually thought of as vanilla kernel primitives offering common case functionality. As a result, they are not accessible and customizable by applications.
The following three points sum up the idea behind this paper:

1. Kernel locks impact system performance differently for different workloads
2. Developers should offer customizations while designing kernel locks
3. Since it would be difficult for developers themselves to maintain a huge amount of such customizations, this task must be safely delegated to the end-user (using eBPF)

Understanding: The SynCord Framework

The SynCord Framework is the authors' realization of the idea. It is split into two sections with a total of 10 API calls:
Safe (8): For people who do not have enough experience with synchronization primitives and can break critical section requirements, SynCord performs necessary validation. This is the standard way of using SynCord
Unsafe (2): For advanced users and engineers who can ensure critical section properties themselves. These API calls allow threads to bypass lock acquisition completely. The threads straightaway go to the critical section

The API calls of the safe section are further split into three groups along functional lines:

- **General (4)**: Consists of APIs called before/after the locks are acquired/released, these API calls are used for profiling (e.g., calculating the time taken to acquire/release a lock)
- **Fast path (2)**: These APIs intercept the fast path access to acquire the lock
- **Waiter reordering (2)**: These are the slow-path APIs which provide ways to re-order waiters to maximize performance

While the Unsafe and General Safe APIs are easy enough to understand, I provide an elaborate explanation of the Fast Path and Waiter Reordering APIs in the subsections below.

Fast Path APIs

Modern day locks usually have at least two acquiring paths, a fast path and a slow path. The **fast path** allows acquiring locks by trying to steal the lock if the lock is unlocked. This is the fast path because if the resource is available, this thread directly takes it instead of waiting for other threads who are waiting in a queue to acquire the lock (which is called the **slow path**). The fast path is useful when we want to give a new thread a chance to win the lock against other slow path threads. It can be seen that using the fast path often may compromise on fairness. There are two APIs in this group:
`lock_enable_fastpath(lock):` This checks if the system supports using the fast path. Systems which trade fairness for performance would generally set this option to be `true`. For other systems, this would be set to `false`.
`lock_to_enter_slowpath(lock,node):` If the thread is not able to get this lock using the fast path or if we don't allow it to use the fast path, this API call is called before entering the slow path. This API also allows forcing certain nodes (threads) to wait before enqueueing themselves into the wait queue based on user-specified logic.

Waiter Reordering APIs

These design policies control the order in which threads waiting in the lock acquiring queue get the lock. These APIs allow transforming the FIFO queue of waiter threads into a priority-queue-like data structure. This reordering is only done in the slow path and only while some thread tries to acquire the lock. SynCord gives two APIs for the same:
`should_reorder(lock,anchor,curr):` This call tries to compare the current thread (curr) with some reference thread (anchor) based on user-specified logic (such as number of important operations carried out by the node, load on the node, urgency of node completion, etc.), moves the current thread to the front of the queue.
`skip_reorder(lock,anchor):` This call skips the entire re-ordering procedure as described above and allows systems to enforce FIFO when some bottleneck arrives in the design logic and/or some threads start to starve and affect system performance.
Re-ordering of waiting threads, if not done intelligently enough, will impact fairness and can cause some waiter threads to starve, even though SynCord provides mechanisms like `skip_reorder`. To ensure bounded waiting times for every waiter, SynCord reverts to FIFO if the thread suffers from starvation after a statically set timeout (10ms).

Understanding: Use Cases

The authors provide a few use cases to justify their idea. Some of them are explained below:

1. **NUMA aware spinlocks**: Design to effectively use spinlocks on systems with Non Uniform Memory Access (NUMA) architectures. Such spinlocks batch threads running on the same socket together. A separate `socket_id` is stored in each thread for this purpose.
2. **Asymmetric multicore lock**: For systems with different speeds of their cores, SynCord checks if the thread is assigned to a fast core or a slow core. If assigned to a slow core, it decides a random backoff time before entering the queue, thereby boosting the throughput
3. **Scheduler-cooperative lock**: For scenarios when one thread holds the lock for a much larger time than the other thread, SynCord tracks the time a lock is held by a particular thread and decides waiter reordering accordingly.
4. **Dynamic lock profiling**: Since Kernel Locks decide how applications scale with larger data, profiling them is extremely important. The API calls in the General Section provide a good opportunity to measure the runtimes of default and customized kernel locks.

Importance in OS Research

While this is not the typical OS Research Paper exploring dozens of possible kernel optimizations or changes to kernel data structures, it explores a consumer oriented domain of Operating Systems. That being said, offering the end-user a chance to change lock policies is also a good way to make not-so-serious users/administrators think about how their application throughput can be improved by intelligent design of lock logic. Therefore, as commercial as it may sound, this paper allows users to think from an OS perspective in general and synchronization in particular. This leads to new insights in OS Research.

Using Trāṭr to tame Adversarial Synchronization

The paper introduces two novel attack methods exploiting the synchronization primitives used to access shared kernel resources between different users/tenants namely, Synchronisation and Framing attacks and Trāṭr, a Linux kernel framework to detect and recover from these attacks.

Synchronization and Framing Attacks

Setup: Infrastructure providers have been striving towards providing as much isolation as they can of shared resources like CPU time, memory etc. but for setups where each tenant has containers running atop a common operating system, isolating kernel data-structures, like the global file description table or list of task_structs is not possible. In such scenarios, it is imperative to use synchronization primitives like locks or RCU. The premise for the aforementioned attacks is that the attacker can intentionally add objects to these data-structures with weak time complexity guarantees (like a linked list perhaps), to increase the length of the critical section and thereby increase the stall/wait time for the synchronization, hence reducing the throughput drastically.

Synchronization Attack

The paper discusses three conditions for a data structure to be vulnerable to this attack.

C1 It needs to be protected by a non-preemptable synchronization primitive.
C2 Weak complexity guarantees, like in linked-list, the search is $O(n)$.
C3 Expand the data structure such that it now takes longer to perform the critical section.

It is an active attack wherein the attacker adds a large number of elements to the data structure, so that the victims now trying to access the data structure would have to wait long periods of time for the critical section to complete. They demonstrate the possibility of such an attack on the following data structures in the kernel:

`inode_cache`: The kernel maintains this cache in the form of a hash table with chaining used to handle collisions. The input to the hash being the inode value for the file (unique to each file) and the address to the filesystem superblock entry in the memory. They use the FUSE unprivileged file system to deterministically allocate inode values to the files from the user space. Hence the attacker can specifically generate files which give the same hash value, leading to long chains of inode entries.

`futex_hash_table`: Rather than have a separate wait queue for each `futex` variable, the kernel maintains a hash table indexed by hashing the address of the `futex` variable and the value being a chain of threads waiting for the `futex` with the same hash value. Whenever a futex is released and the next thread needs to be woken up, the whole chain is traversed to find the threads waiting for this particular `futex`. The attacker creates a `futex` variable and spawns thousands of threads to wait for it leading to expansion of the corresponding bucket. The attacker need not explicitly break the hash function here.

Framing Attack

This is a passive attack wherein the attacker simply expands the data structure as discussed above and does nothing thereafter. The victims which now try to access the data structure experience long critical sections thereby giving the impression that they are the cause for the long wait times. In a way, the attacker is now framing the victims, making it difficult to identify the attacker.

Trātṛ

Trātṛ is a linux kernel extension designed to detect and recover from synchronization and framing attacks. The high level design goals are listed below:

- Automatic response and recovery: The scale of the problem is huge considering there are hundreds of machines running thousands of containers in a data center. It is best if the system can self detect the attacks and initiate recovery protocols.
- Low false positives and false negatives: Since there is no definitive way of distinguishing between a heavy resource usage and a denial of service attack, they do not want to rely on only a few indicators which might lead to false positives or false negatives.
- Easy/flexible support for multiple data structures: Methods to recover from the attacks might differ from one data structure to another and hence it is important that the developers can specify their own recovery sequences.
- Minimal changes to kernel design and data structures: Drastically changing a data structure already in use by the kernel may hamper its compatibility with other data structures/routines in the kernel and one might end up rewriting a substantial part of the kernel which we can’t be sure is not prone to other attacks.

Indicators used to detect the attack:

- Condition **L**ong **C**ritical **S**ection(LCS): If all the tenants are consistently running long critical sections, it might be due to an ongoing attack or simply by coincidence that all the tenants operating on the machine happened to contribute largely yet equally to the expansion of the data structure leading to LCS.

- Condition **H**igh **S**ingle **U**ser **A**llocation(HUSA): To rule out the case of the coincidence mentioned in the above condition, we track the resource allocation made by each of the tenants.

Trātṛ detects that an attack is taking place only when both of the above mentioned conditions are satisfied as with only one of them happening we cannot be sure if an attack is taking place.

Detection of an attack