CS60038: Advances in Operating Systems Design

Term Project (Final 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: SpinLock

TCLocks augments test-and-set and MCS locks by employing the following steps.

Data Structure:
A queue represents every lock, and every requestor is a node in this queue.
The head of the queue is selected as the combiner by default.

Workflow:
When a thread acquires a lock, it switches to an ephemeral stack and joins the queue.

If it is the first node in the queue, it becomes the combiner. It can batch waiters' requests up to a set threshold. This reduces the chances of causing starvation and ensures long-term fairness.

The combiner iterates through the queue to find a waiter thread to process. It context-switches to the waiter's context and starts processing.

In the meantime, the waiter thread handles interrupts and signals on their ephemeral stacks and waits for a notification from the combiner.

When the combiner finishes processing the waiter's critical section, it notifies the waiter. The waiter changes back to the original context and continues with its non-critical section.

TC Locks: Blocking Lock

Blocking TCLock, following the design principles of SHFLlock (Scalable and Practical Locking With Shuffling), employs a spin-then-park strategy for waiters. Waiters spin locally until their time quota expires, at which point they schedule themselves out if the system is oversubscribed, otherwise yielding to the scheduler for eventual rescheduling. The lock queue manages both active and passive waiters. Blocking TCLock is designed by enhancing SpinLock TCLock to support a parking/wakeup policy. The combiner's role is extended to wake up sleeping waiters during critical section execution, with an ephemeral stack used to prevent concurrent accesses during parking. The stack switching protocol is consistent with SpinLock TCLock. The process involves atomic operations to prevent the lost wakeup problem, with specific phases for spinning, parking attempts, and critical section execution by the combiner.

TCLocks: Reader-Writer Lock

Reader-Writer TCLock is a combining-aware readers-writer lock that facilitates parallel execution for readers while combining writers. It employs a phase-based mechanism, alternating between readers and combined writers. The lock consists of a counter containing reader count (RCNT), a writer present byte (WP) indicating if a writer holds the lock, and a writer waiting mutex (wlock) for synchronizing phases between readers and the head of the writers queue. The algorithm involves using the Blocking SHFLlock algorithm for wlock, emphasizing the importance of decentralizing the count of readers to align with the combining design. The reader algorithm involves incrementing RCNT and executing the critical section if no writer is present, otherwise, entering the slow-path phase. Writers enter the critical section by switching WP from 0 to 1, with a slow-path phase if unsuccessful. The unlocking phase involves resetting the rwcounter to 0 for writers if the value is WP.

Optimizations: Direct Stack Switching

In the original SpinLock TCLock version, the combiner undergoes two stack switches before proceeding to the next waiter's critical section: initially to its own context and then to the

selected waiter's context. To eliminate the need for the switch to the combiner's context, the combining loop is divided. After transitioning the stack to a waiter's context, the combiner attempts to choose the subsequent waiter for combination and notifies the preceding waiter of the completion of its critical section. Subsequently, the combiner exits the lock function call and executes the current waiter's critical section. Following the critical section execution, during the unlock phase of the current waiter, the combiner assesses the combining loop conditions. If the conditions are met, the combiner directly switches to the next waiter's context. Conversely, if the conditions are not satisfied, it marks the conclusion of the combining phase, switches back to its context, notifies the previous waiter, and ultimately executes its own critical section.

Optimizations: Minimizing Context Switch Overhead

Using the function's caller-callee convention:
To minimize overhead, the authors utilize the function calling convention, designating the slow-path of lock acquisition as a function to prevent compiler inlining. This approach is triggered only in contention scenarios, similar to the Linux spinlock implementation. Additionally, the algorithm strategically saves, transfers, and restores only the necessary callee-saved registers, while the compiler handles caller-saved registers based on register liveness information. This optimization reduces the number of transferred cache lines to encapsulate critical sections, contributing to the algorithm's efficiency.

Prefetching thread-local data:
When accessing thread-local data within a critical section, the need to transfer the waiter's specific code and data from its CPU to the combiner's CPU can extend the critical section's duration. The authors observe that much of the local data typically resides on the stack, accessed or modified by a thread within the critical section due to aggressive compiler optimizations. To mitigate the impact on critical section latency, the algorithm prefetches contiguous cache lines from the top of the stack, reducing the time needed for data movement. This prefetching is executed by the combiner before entering the current waiter's critical section. This strategy is unique to the combining approach, as traditional lock designs face limitations since the lock holder already has access to its local data, and shared data cache lines can only move to the CPU holding the lock at the critical section's end.

Real-World Applications: Multi-level Locking

Multi-level locking in the context of combining introduces two distinct usage patterns that necessitate careful design and implementation. First, a combiner thread may act as a waiter while executing a nested lock, referred to as a "waiter-combiner." Second, locks can be released in an arbitrary order, resulting in out-of-order unlocking. While traditional locks are not impacted by out-of-order unlocking, TCLock demands additional attention to properly handle such situations. Mishandling could lead to data corruption and potential deadlocks. The authors now outline their approach to effectively support and address these usage patterns in the combining framework.

Nested Combining
TCLock addresses nested combining by adopting an interrupt processing mechanism similar to operating systems. In this approach, interrupt handlers push the current thread state onto the stack before processing the interrupt, and upon completion, they restore the thread's state from the stack. This mechanism allows TCLock to handle nested locks without affecting the interrupted thread's execution. Three cases arise when TCLocks interact with nested locking: firstly, when both locks are in their combining phase; secondly, when the outer-level lock is a combiner and the inner one is the fast-path TAS lock; and finally, the reverse scenario of the second case. To handle the first case, TCLock implements changes similar to the interrupt processing approach, allowing a combiner to acquire a nested lock within the critical section by pushing and restoring its state appropriately. TCLocks also seamlessly supports the latter two scenarios, where one of the locks is in the combining

phase, as each lock operates independently without interaction between their underlying mechanisms. This independence is consistent with traditional lock design, where each lock has its own lock word and interacts only with its specific lock word. Therefore, acquiring a lock in the fast path (TAS lock) does not interfere with the lock held by the combiner thread.

Out-of-order (OOO) unlocking
The discussed algorithms for TCLocks faced issues with out-of-order (OOO) unlocking, potentially leading to incorrect program execution. An example scenario was presented where multiple threads acquired locks L_A and L_B, entering the combining phase with C_A and C_B as combiners. When C_A, a waiter-combiner, attempted to acquire lock L_B, an out-of-order unlocking occurred when L_A was released before L_B. This caused combiner C_B to prematurely execute the next waiter's critical section while still holding lock L_B. To address this, TCLocks introduces changes to its lock and unlock functions. The algorithm now maintains a per-thread lock_addr array to record the order of lock acquisitions. A combiner stores its lock's address in this array before entering the combining loop and removes it after completion. In the unlock function, the combiner checks if the unlocked lock is the last entry in the lock_addr array. For non-OOO cases, the combiner follows the original algorithm; for OOO cases, the combiner marks the lock as OOO-unlocked and continues until the unlock function is called. The waiter-combiner for the OOO-unlocked lock waits for notification from the current combiner, effectively flattening the lock hierarchy for out-of-order unlocked locks. After receiving the notification, the waiter-combiner checks if its lock is OOO-unlocked, returning G_UNLOCKED_OOO if true. The combiner then switches to its previous state, returns to its combiner loop, notifies the current waiter, and continues combining the next waiter. The control returns to the waiter for the outermost lock once all locks are released, allowing it to execute its non-critical section. These changes address the OOO unlocking issue and ensure correct program semantics in the context of combining.

Evaluations
The authors conduct three main types of evaluations:
a) Kernel-based TCLock Implementation on microbenchmarks (to stress the lock)
b) Kernel-based TCLock Implementation on real applications (to stress kernel subsystems)
c) Userspace TCLock Implementation on real applications

The system they test is run on has an 8-socket, 224-core Intel Processor. The comparisons are made against Linux’s stock locks, CNA (Compact NUMA-aware Lock), ShflLock.

Test A:
TCLock demonstrates comparable performance to Stock when operating with two to eight cores, as the combiner struggles to function effectively at these lower core counts. The advantages of TCLock become apparent beyond eight cores, where the benefits of localizing shared data cache lines surpass the overhead of stack-switch operations. TCLock consistently sustains high throughput, even with 28 cores. In cross-socket comparisons, TCLock outperforms SHFLLock and CNA by 2-3×. The NUMA-aware policy of TCLock SP, based on combining, minimizes cache-line bouncing for both the lock word and shared data.

Test B:
In the context of the Metis in-memory map-reduce framework, which poses a page-fault-intensive workload, TCLock demonstrates superior performance compared to both SHFLLock and Stock by a factor of 1.3× This performance advantage is attributed to the phase-based design of Reader-writer TCLock , which optimizes efficiency by batching writers in one phase while concurrently executing readers in the next phase. Particularly across sockets, Reader-Writer TCLock exhibits performance improvements of 1.7× and 1.4× at 140 cores compared to ShflLock and Stock, respectively. The phase-based approach enhances overall throughput and responsiveness, making TCLock an efficient choice for handling the mmap_sem (rwsem) in the Linux kernel under the demanding workload characteristics of the Metis framework.

Test C:
The evaluation of TCLocks on the LevelDB benchmark, integrated into LiTL along with CNA and ShflLock, reveals notable performance improvements. Using the read random benchmark with 1 million key-value pairs contending on the global database lock illustrates the performance with spinlocks on an 8-socket machine. Within a socket, TCLock achieves throughput improvements of 1.9×–2.6× compared to other locks, highlighting the advantages of localizing shared data movement for enhanced performance. Across sockets, TCLock, with NUMA awareness and minimal shared data movement, outperforms other locks by up to 5.2×.

We observe similar performance trends on a 2-socket machine, where TCLock maintains comparable performance to the 8-socket configuration. In this scenario, TCLock achieves throughput improvements of 2.1×–3.6× compared to other locks. These results underscore the effectiveness of TCLock in optimizing performance for the LevelDB benchmark, showcasing its superiority in both intra-socket and inter-socket scenarios.

Limitations

TCLocks implement transparent delegation to facilitate the use of delegation-style locking without application rewriting. However, certain limitations exist in both algorithm design and kernel implementation. Notably, TCLocks exhibit overhead when a small number of threads (two to four) contend for a lock, as combining is only enabled when more than two waiters are present. To address this, the combining feature could be disabled when fewer than four threads are in the queue. However, efficiently identifying the queue size without additional memory usage or traversal poses a challenge. Additionally, TCLocks complicate resource accounting in the kernel, as accurate accounting for resources within the critical section is crucial for maintaining broader kernel semantics. The extension of accounting resources for combiner threads executing critical sections on behalf of waiter threads is left for future work.

Moreover, TCLocks present challenges when interacting with the Linux 'current' macro, which resolves to a per-CPU pointer variable pointing to the currently executing thread's task structure. While executing a waiter's critical section on the combiner's CPU, switching the 'current' pointer to the waiter's task structure could lead to subtle bugs, such as privilege escalation issues if the combiner thread has higher privileges. Modifying the 'current' macro's implementation to resolve the waiter's task structure during the waiter's critical section on the combiner CPU may introduce confusion within the scheduler, particularly when a thread sleeps within its critical section. Despite these challenges, the current macro remains unchanged in TCLocks, and users are advised to judiciously employ TCLock APIs in situations where a different thread identity within the critical section may result in unexpected behavior.

Importance in OS Research

The paper brings the performance of delegation-style locks to a simpler API. 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

Idea and 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 to be officially provided by Linux), **networking**, **storage** and **acceleration** (for example, efficient packet processing directly on the NIC using XDP). However, such customizations are not offered while designing kernel locks which are usually thought of as vanilla kernel primitives offering common case functionality and are generally considered to be 'efficient' enough for most user applications. Therefore, their accessibility and customizability is heavily restricted due to lack of necessity and the paramount importance of security of kernel memory.
Analysis reveals that the following three points sum up the idea behind this paper:

1. Several experiments have shown that kernel locks impact system performance differently for different workloads
2. Kernel Developers should offer customizations while designing kernel locks, if not directly, then they must offer abstractions for users to customize locks themselves.
3. Since it would be difficult for developers themselves to maintain a huge amount of possible customizations (imagine the effort of backporting these implementation changes to older stable kernels) and users know better about the workloads of their own application, this task must be safely delegated to the end-user.
   However, knowing that the user is not aware of the sensitivities in OS design and is only interested in the functionality, eBPF can be used for the same, since the eBPF verifier prevents unauthorized access to protected kernel memory (as seen in Assignment 2).

The SynCord Framework

The SynCord Framework is the authors' abstraction to customize kernel locks on the fly without recompiling or rebooting the kernel. Users write their lock algorithms in the user space and SynCord dynamically patches them into the kernel. The authors claim to satisfy the following design goals:

Correct lock patching: No correctness bugs inside the kernel Sandboxed user's code: Preventing user provided corrupted code to wreck the kernel, if they use SynCord provided APIs Usability and expressiveness: As much customization to kernel locks as possible

However, expressiveness and sandboxing are not independent of each other. More expressiveness means more API functions provided to the user. However, this leads to less sandboxing, i.e. there is a smaller chance that kernel routines are isolated and independent of each other. On the other hand, more sandboxing means more isolated kernel functions, which clearly lead to less customizability.
The authors try to balance sandboxing and expressiveness by using two different sections of API calls, Safe (sandboxing, focussed more on isolation) and Unsafe (expressiveness, full control of locks). These two sections contain a total of 10 API calls.

UNSAFE (2): For advanced users and engineers who can ensure critical section properties themselves. These API calls allow threads to bypass lock acquisition completely. This section consists of routines:

```
bool lock_bypass_acquire(lock)
bool lock_bypass_release(lock)
```

If these functions return true, the lock is given/released immediately and the `acquire()` and `release()` functions return. These functions allow users to write **new/customized locking algorithms** at the cost of mutual exclusion violation checking. Users will have to ensure mutual exclusion (correctness) themselves if they use these functions. This is why the authors call these APIs unsafe because there is no verifier to ensure correctness.

SAFE (8): This the standard sandboxed path to lock customization. Users who follow this path **cannot change the underlying implementation of locks**. However, these APIs allow users to define custom logic to **reorder waiters** waiting on a lock and data structures to **profile** the performance of the waiting algorithm (by offering APIs which measure acquire and release time). Basically, these APIs allow users to define 'priorities' among the competing threads. For people who do not have enough experience with synchronization primitives and can break critical section requirements, SynCord performs necessary validation.
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

General APIs

Most locks support functions such as `acquire()` and `release()`. SynCord provides four API calls which are called at the beginning and end of both of these API calls. The General APIs allow users to intercept the entry and exit points of the acquire and release phase. Their main use case is profiling. To measure the time taken to acquire a lock, we start a timer in the `lock_to_acquire(lock)` function and then end this timer in the `lock_acquired(lock)` function. Similar APIs are provided for releasing the lock. Similarly, time spent in the critical section can be profiled by starting a counter in the `lock_acquired(lock)` function and stopping it in the `lock_to_release(lock)`.

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. The fast path is implemented by using a `test_and_set` atomic instruction (studied in the OS course) and is used in low contention scenarios. Why queue yourself when it is certain that you will get the lock immediately?
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). It is important to observe that no other bugs except those due to starvation will happen since SynCord allows changes only to the ordering of the waiters.

SynCord Implementation

SynCord targets only non-blocking locks (such as spinlocks and reader-writer locks). I presume this is because for blocking locks, re-ordering will be very challenging, since this would involve tampering with kernel wait queues. For implementation purposes, SynCord uses eBPF and Livepatch. It requires a minimal one-time change of 143 lines in the Linux kernel to expose its APIs.
Usage of eBPF: SynCord guarantees code safety: memory safety (no access to illegal memory address), termination (no infinite loop), liveness (no deadlock) and mutual exclusion. The eBPF verifier is used by SynCord to enforce these security requirements given above. However, since the eBPF verifier does not support infinite while loops, SynCord provides a new helper eBPF function, called `backoff()` which can be used in the SynCord APIs.
Usage of Livepatch: SynCord uses Livepatch to dynamically deploy code into the kernel. Livepatch first generates a difference (diff) between the changed code and the original code. It then compiles the diff as a kernel module.This is not done using eBPF since code inserted using eBPF will affect all lock instances in the system. It is important to understand that the insertion of this module is done only when a thread leaves its kernel state or all the CPUs are idle.

Auxiliary Data Structures

SynCord supports three kinds of data structures, per-node, per-lock and global. These data structures will be used to store extra information which may be used for customized lock algorithms, such as the NUMA-aware lock given in Use Cases.
Per Node Data: A node is created when a thread is about to use the slow-path (i.e. go into the wait queue). It exists only as long as the thread is waiting in the queue and gets destroyed when the thread acquires the lock.
Per Lock Data: For per-lock data, shadow variables are used to allocate memory only for the targeted lock instances. In particular, per-lock auxiliary data is included inside an in-kernel key-value store created by Livepatch. The address of a target lock instance serves as the key, while the value is per-lock auxiliary data.
Global Data: Global data such as per-CPU data can be also stored in that key-value store. SynCord frees the extra memory allocated as shadow variables when it removes the corresponding policy. Again, it is important to understand that the underlying parent lock

structure is not modified to store auxiliary data, but it is stored in a separate location from the parent lock object. Therefore, extra space is only allocated when a policy is in force.

Use Cases

The authors provide five use cases to justify their idea. In fact about half of the paper focuses on use cases.

NUMA-aware Spinlocks

Designed to effectively use spinlocks on systems with Non Uniform Memory Access (NUMA) architectures in which accessing local memory is faster than accessing remote NUMA memory. Locks which are considered NUMA-aware batch acquisitions from the same socket together. This helps to improve performance, since changing between CPUs leads to cache line misses in accessing the lock's memory location. NUMA locks ensure long-term fairness by periodically passing the lock to another socket. SynCord handles this passing randomly using a sample random() call in the `skip_reorder()` method.
A per-node auxiliary integer is stored to record the socket ID of each waiting thread. The fast path (explained in Fast Path APIs) is enabled for lock stealing. However, if this fails, the waiter enters the slow path after storing the socket ID. Reordering of threads happens in the slow path. First, all the threads belonging to the same socket as the anchor thread are brought to the front of the queue by calling `should_reorder()` only on those sockets which have the same socket ID as the anchor thread. As mentioned in the paragraph above, to ensure long-term fairness and change of socket IDs, we randomly skip reordering, thereby making the queue return to normal FIFO operation. This makes the current waiter the anchor node and it repeats the grouping process until skip_reorder() again goes back to FIFO operation.

Asymmetric Multicore Lock

Asymmetric Multicore Processors (AMP) consist of different cores with variable computing powers. Most-commonly found in ARM architectures, these consist of energy-saving slow cores and energy-guzzling fast cores. Therefore, AMP machines can generalize across all usage patterns, using slow cores when power saving is needed, and using fast cores when in need of performance. Current lock designs are not suitable for the AMP architecture, since they assume homogeneity among the cores. However, it is easy to see why we need a separate locking routine for AMP systems. Slower cores will take time to execute critical sections.
SynCord handles locking on fast and slow cores using the following paradigm. During a low contention scenario where we expect locks to be acquired through the fast path (explained in Fast Path APIs) more often, it allows both slow cores and fast cores to acquire the lock to maximize performance, since there is not really a 'penalty' to acquisition (such as pushing to a queue). Meanwhile, during high contention (i.e. when `lock_to_enter_slowpath()` is called), it penalizes slow cores by forcing them to wait for a maximum fixed time of 10ms before entering the lock acquisition queue so that fast cores can acquire the lock more aggressively for better performance, since slow cores will anyway take huge times to process their critical sections. A very high wait time will ensure starvation of workloads on slow cores. A smaller wait time would reduce throughput. It has been found that an appropriate wait time (10ms) prevents starvation and ensures acceptable latency for workloads running on slow cores.
This wait time uses the `backoff()` helper function mentioned in SynCord Implementation.

Scheduler Co-operative Lock

Current CPU schedulers let each thread have an equal share of the CPU time. However, in the case when two threads spend most of their time executing a critical section protected by the same lock, and one thread holds the lock for a much longer time than the other thread

(these threads are called the bully and victim threads respectively). Now since critical sections are executed in an all-or-nothing fashion, the bigger thread ends up taking the CPU for a much larger time than the smaller thread. This directly goes against the principles of scheduling and can lead to lower performance. SynCord implements a new lock algorithm which strives to ensure that each thread holds the critical section for the same time. Basically, SynCord implements something similar to the CFS algorithm by storing something similar to the `vruntime` as discussed in the class. If one thread holds the lock longer than its share, it cannot acquire the lock until other threads have received an equal chance to acquire the lock.
The following variables are introduced:

- A per-thread hold-time variable
- A per-thread variable for recording the beginning of the critical section
- A per-lock integer for counting the number of contending threads and total lock holding time

Algorithm

1. Before entry into the wait queue (i.e. from `lock_to_enter_slowpath`), a thread (t) computes the lock quota based on the number of threads and the overall lock holding time (basically total lock holding time divided by the number of threads)
2. t waits until other threads get equal opportunity by backing off for the approximate time it spent in the critical section using the `backoff()` function defined in SynCord Implementation.
3. Per-lock hold time and per-thread hold time are updated by tracking when the thread starts in `lock_acquired()`(after acquiring) and when it ends in `lock_to_release()`(after releasing).

Biased Per-CPU Readers-Writers Lock

Readers-writer lock (`rwlock`) is one of the most widely used primitives in Linux. This primitive allows either multiple readers or one writer to acquire a lock. Most `rwlock` designs track active readers with a centralized readers indicator. However, the centralized readers indicator has poor scalability because frequent atomic instructions for readers result in frequent access to different cache-lines and coherence traffic. Therefore, `rwlock` is a good candidate only for read-intensive workloads since otherwise the complexity in implementing `rwlocks` tends to bottleneck performance.
SynCord provides a dynamic approach wherein a distributed readers-writer lock is only enabled when needed by the users and disabled in write-intensive workloads when a simple mutex-like lock provides better memory footprint and writer latency.
This lock is made entirely by the authors of the algorithm using the unsafe API methods (`lock_bypass_acquire` and `lock_bypass_release`) Moreover, another unsafe function (`backoff_unsafe`) that waits indefinitely until the condition is met is also used by the authors.
Two extra per-lock attributes are added to the lock

- Read Bias mode tracks whether the mode is biased towards reading
- A table of visible readers with cache-line aligned entries

Algorithm (Reader)

1. Before a reader (R) acquires a lock, it first checks the read-biased mode. If the read-biased mode is set, R marks itself as an active reader and checks the read-biased mode again due to a possible race from the writer's side.
2. If the read bias is still set, R bypasses the underlying lock, else it falls back to the underlying implementation
3. At the time of release, R checks whether it acquired the lock in the read-biased mode and bypasses the underlying lock release if so.
4. R is also responsible for setting read-biased mode once it acquires the underlying lock without bypassing it so that other readers can directly acquire the read lock by setting their respective indicators.

Algorithm (Writer)

1. A writer (W) first acquires the underlying writer lock.
2. W will only acquire the lock when there are no active readers that hold the underlying lock.
3. After that, W further checks for the read-biased mode. If it is active, W first disables it and waits for all readers to exit the critical section that used the per-CPU indicator.

Dynamic Lock Profiling

Kernel locks decide the scalability of applications. Therefore, lock profiling tools are critical to understanding a lock's performance. However, there are no majorly useful tools to measure lock-specific performance statistics, such as the time spent in a critical section of a particular lock instance in the system. Currently used tools can either profile all lock instances at once (consuming a large amount of memory) or can only profile CPU cycles (`linux perf`). Due to the exposure of API functions like `lock_to_acquire`, `lock_to_release`, `lock_acquired` and `lock_released`, users can customize which set of lock instances to profile with specific statistics (even the algorithm-specific ones, defined in `lock_bypass` APIs) using simple changes in the API functions and defining global and lock specific variables. Therefore, SynCord enables users to perform custom, fine-granularity lock profiling, which can greatly simplify the performance analysis of lock algorithms.

Limitations of SynCord

An important limitation of SynCord is its inability to generalize to blocking locks. While the authors of the paper have mentioned that they would eventually like to reach that goal, it does seem far-fetched to expose the functionalities of a large number of locks as APIs. Another limitation is that SynCord supports only C to write customization logic. Since eBPF code is now supported in Python and Rust which are more expressive memory safe languages, it is imperative that this support can also be added to SynCord.

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 extension to detect and recover from these attacks.

Algorithmic Complexity Attacks

Class of Denial of Service attacks where the attacker triggers worst case performance of the system by giving specific inputs thereby hogging up the resources for most of the time.

Shared infrastructure and how they are vulnerable?

Modern day infrastructure providers need to share their resources between tenants from different organizations thus requiring to isolate their usage of resources so that their performances are independent of each other. Previous works have focused on how to provide isolation of CPU usage, memory usage, file system directories, PIDs etc with scheduling policies, memory partitioning and namespace creation respectively. The sharing of the resources of a single machine is enabled by spawning multiple VMs or containers atop the hardware resources. A VM enables true isolation as the isolation is up to kernel level, but in the case of containers like say docker containers or linux containers, they share the same kernel underneath. Hence although the unprivileged users cannot meddle with the kernel resources directly they can use the system calls in a specific pattern to hog up the resources of the kernel. We will see in further sections in more detail how it is possible and the extent of damage that can be caused.

Vulnerable kernel data structures

The kernel uses several data-structures for book keeping and fast access like caches, hash tables etc. For example, the pid-struct pid mapping hash table, global open files table, the red-black tree for CPU scheduling etc. Many of these data structures internally provide isolation between users, for instance other users cannot access the files they are not permitted to access and since the processes have a direct pointer to the entries in the table, they need not traverse the whole list of open files.

However in the case of some of the data structures such isolation is not possible. For example, the pid-struct pid mapping hash table uses chaining to store all the processes with the same pid in different namespaces. Hence if a tenant wants to find its process in the table, it would have to traverse the whole doubly linked list housing process blocks from other tenants as well.

Synchronization primitives in the Kernel

Since multiple tasks the data structures in the kernel it is necessary that we use synchronization primitives to provide for mutual exclusion when modifying the data or to read the data when the data structure is being modified. The kernel uses the following synchronization primitives:

- Mutual exclusion locks: Only a task holding the lock can enter the critical section at a time. Once the task is holding a lock it is non-preemptable, that is another process requesting access to the critical section can only enter when the current process in the critical section gives up control voluntarily.
- Read-Copy-Update construct: In a data structure if we want to have reads concurrently when a write is happening we can use this construct. The write process takes place in two phases: removal and reclamation. Splitting the update into removal and reclamation phases permits the updater to perform the removal phase immediately, and to defer the reclamation phase until all readers active during the

removal phase have completed, either by blocking until they finish or by registering a callback that is invoked after they finish. Only readers that are active during the removal phase need be considered, because any reader starting after the removal phase will be unable to gain a reference to the removed data items, and therefore cannot be disrupted by the reclamation phase.

Synchronization attack and Framing attack

The premise for the aforementioned attacks is that the attacker can intentionally add objects to the shared 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.

Linked list example: Attacker increases the length of the critical section

In a linked list, while searching for an element it is imperative for the reader to hold a mutual exclusion lock. If an attacker somehow adds a lot of elements to the linked list, the reader threads searching for the elements would incur huge performance losses. In order to add millions of elements to the linked list, the attacker would be holding the lock for a long time hence stalling the other processes waiting for access to the list. This type of attack where the attacker actively holds the lock is called a Synchronization attack. In the other case where the attacker has already expanded the data structure and now does nothing but the since the other victims are in the critical section for so long, it looks like they are the culprits, such an attack is called a framing attack.

Conditions

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.
-

Framing attack:
Additional condition: The victims access the elongated portion of the data structure.

Synchronization attack makes the victims wait longer and framing attack additionally makes the victim use the critical section for a longer time too. Since existing isolation methods do not treat synchronization primitives as a resource and hence don't provide for fair distribution it is problematic.

Real World Examples

Inode_cache: Synchronization attack

An inode cache is a cache that contains copies of inodes for open files and some recently used files that are no longer open. The cache uses a hash table and inodes are indexed with a hash function that takes as parameters the superblock and the inode number associated with an inode. In case of collisions in the hash table, linear chaining is used. A global lock is used to provide synchronization protection. File System in Userspace is a user side file system in which the non-privileged users can specify the inode number they want to use for an inode. Thus if an attacker can figure out the superblock address for the file system, they can easily give inode numbers such that there is a large amount of collision. In the paper they mention that they have broken the hash function.

Thus to attack the inode cache, the attacker simply creates indodes to collide with a certain bucket in the table thus increasing the length of the critical section of the insert operation. This is a synchronization attack as the attacker cannot predict which buckets the user would access and hence cannot frame a user.

To demonstrate the damage caused by such an attack, they run an application Exim mail server over the machine and simultaneously attack the kernel. A reduction of 92%, i.e. 12x is seen in the throughput performance of the mail server application. In contrast to this running other competing applications like UpscaleDB would only reduce the throughput of the mail server by 15%.

Futex Table: Framing attack

A fast user-space mutex (futex) is a tool that allows a user-space thread to claim a mutex without requiring a context switch to kernel space, provided the mutex is not already held by another thread. Each futex variable has a wait queue of its own, so that the threads waiting on the futex are put to sleep in the waiting queue and woken up when it's their turn to enter the critical section. In the kernel, the wait queues are implemented using a hash table indexed by the memory address of the futex variable. It is possible that the hash values of multiple futexes collide and thus they share the wait queue. Each bucket in the hash table is protected by a mutex lock. Thus it satisfies conditions S1 and S2. Since each of the threads waiting for each futex are put in the same wait queue, an attacker simply needs to create a futex variable and spawn millions of threads to wait on it. It would rapidly expand the bucket corresponding to the futex. The attacker would not even have to break the hash function. They can create a few hundreds of the futex variables and sample the wait times for each of them. The one with the maximum wait time would have the maximum contention. The attacker hence spawns a lot of threads for that particular futex thus slowing down all the other processes as well. Since the other processes also have their threads residing in the same queue, they will undertake the read operation for a long time leading to a framing attack.

To demonstrate the damage caused by such an attack, they run an application UpscaleDB over the machine and simultaneously attack the kernel. A reduction of 5x is observed in the throughput while the latency increases by 45x-100x.

Directory Cache: Attack on RCU

In RCU, whenever an update is taking place the updater can collect the removed block as part of the update operation or add a callback whenever the reads on the removed block are finished. The time to wait for all reads to finish is called the grace period. If the attacker expands the data structure, the grace period would obviously increase hence increasing the update run time in case of synchronized reclamation and increase the load on the collector in case of asynchronized collection. The dcache is implemented as a hash table where each bucket stores a linked list of dentries with the same hash value. The hash function uses the parent dentry address and the filename to calculate the hash value. The attack exploits the dcache's support for negative entries. These entries record that no such file exists. By breaking the hash function, an attacker can create millions of negative entries mapping to a single hash bucket, thereby meeting condition S1 + S2 + S3. Walking an expanded hash bucket increases the read-side critical section, thereby increasing the grace period size too.

To demonstrate the damage caused by such an attack, they run an instance of the Exim Mail server and launch an attack on the kernel simultaneously. By the end of the experiment, the performance drops by up to 90% and the grace period increases from 20-30 ms to 2 s. The mail server generates hundreds of thousands of callbacks every second overwhelming the RCU background thread.

How to detect the attacks?

Trātṛ is a linux kernel extension(modifications in the kernel code and then compile) 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.

Attack Indications

- Condition LCS: Long critical section. Expansion of the data structure causes more work for the attacker or victims or both, making the critical section longer.
- Condition HSUA: High single-user allocations. A single user has created many entries associated with the data structure

It is argued that to declare that an attack is going on both the conditions are to be satisfied simultaneously. For instance the LCS condition could also happen in the case that the kernel has to handle a lot of interrupts while processing the critical section of a task in which case the task could be falsely flagged as the attacker. Hence, Tratr first checks critical section size and if it's too large, then checks if one user has a majority of the object allocations.

Tratr provides for detection of the attack, prevention from further damage once an attack is detected and recovery from the attack to prevent framing attacks.

Tracking

The slab allocator in the Linux kernel is a memory management mechanism intended for the efficient memory allocation of kernel objects. Each cache stores a different type of object. For example, one cache is for process descriptors, while another cache is for inode objects. Tratr modifies the slab allocator code in the kernel such that in the object a user ID is embedded while allocating specifying which user allocated the memory and a global hash table is maintained of all the users and the number of objects allocated by them. For the HUSA condition to identify an attack it involves traversing the data structure and finding out if the majority of objects are allocated by a single user who is consequently identified as an attacker. Furthermore in the prevention phase Tratr prevents the attacker from allocating more objects and hence preventing from expanding the data structure further.

The current implementation allows for tracking of objects allocated using `kmalloc()` and `kmem_cache_alloc()` and cannot track the allocations done via the `vmalloc()` and `get_free_pages()` APIs.

Detection

For detection of an attack on a data-structure Tratr runs a separate kernel thread per data-structure to run the detection routines. The detection routines involve sampling the wait

times for all the synchronization primitives in the data-structure as if an attack is taking place the wait times would have increased drastically. They check if the wait time exceeds a certain predetermined threshold. By running workloads that access the inode cache and the futex table, it is observed that worst-case lock hold times are around 2-3 µs and since they run it on a machine with 32 CPUs, the worst case wait time could be 64-96 µs. Hence they've set the threshold to be 100 µs. Tratr creates a sampling window within which a lock is repeatedly sampled; if the stall threshold is exceeded multiple times within this window, the CSS check finishes, and Tratr proceeds to the TCA check. Within a sampling window, Tratr introduces a random delay between two samples to incur low interference with normal user operations. The sampling delay begins between 5-20 ms. Tratr enters suspicious mode on measuring a stall above the threshold. When suspicious, Tratr increases the sampling window size and aggressively samples the lock every 1-5 ms to quickly detect an attack. If no attack is found in the elongated window, Tratr returns to normal mode.

As the sampling window size and the sampling delay is randomized, Tratr makes it difficult for a defense-aware attacker to launch an attack. A defense-aware attacker cannot know when to launch an attack and stop to remain undetected. To remain undetected they would have to stop before the threshold is reached.

The traverse and check allocations portion of the detection routine checks for the HUSA condition. If a particular user has allocated the majority of entries, Tratr tags that user as an attacker and passes their identity to the prevention and recovery mechanisms. Tagging the user holding the lock for a long time would lead to false positives in the cases that it is a framing attack and hence it is necessary to undertake this step.

Prevention

The prevention mechanism sits at the slab-allocator logic. Once an attacker is flagged, it is not allowed to allocate objects for a certain window of time known as the prevention window. This helps in prevention because if the user cannot allocate new objects, they cannot expand the data-structure.They do not kill the container identified as the attacker because in the case that the container was wrongly flagged, it could lead to application corruption. If an attacker is flagged again and again the prevention window for it is increased.

Recovery

Just detection and prevention of the attack would not be enough to restore the performance of the tenants as the objects allocated by the attacker would still be present in the data-structure. Moreover it would lead to the system falsely checking the CSS condition again and again. The recovery mechanism needs to be specific to the type of the data-structure. For instance, for a cache like data-structure it wouldn't affect the consistency of the applications if any objects are blindly removed from the data-structure as upon not finding the object in the cache it can be fetched from the memory. But in the case of data-structures like the futex table, removing the entries made by a user could lead to killing of threads responsible for something in the application leading to corruption. We now discuss the recovery schemes we can use in the latter case:
Partitioning is a traditional approach used to design data structures to ensure isolation. We partition the data-structure so that now the attacker and the victims are forced to use different data-structures thereby preventing any adverse effect on the performance of the victims. Tratr walks the data-structure to identify the entries created by the flagged attacker and moves them to *shadow data-structure.* The shadow data structure is dissolved once the prevention window is over. We note that partitioning may not work with cache-like data structures, as it could create multiple copies of entries allocated by both victim and attacker and lead to inconsistencies.

While Tratr is performing recovery, an attacker can continue to access the expanded data structure, thereby slowing recovery. Tratr solves this by blocking the attacker from acquiring the synchronization primitive until recovery completes.

How to generalize the recovery?
To add a new data structure, a developer must pass a flag to the memory management system when creating the associated slab-cache at boot time. Additionally, the developer must implement CSS checks, TCA checks, and recovery functions for the new data structure

Experiments

The experiments are run on two benchmark suites: the *IC benchmark and the FT benchmark.* The IC benchmark contains a victim which is sensitive to the inode cache because it creates an empty file every 100 microseconds. The IC attacker identifies the superblock pointer and then targets a hash bucket by creating files whose inode number maps to that hash bucket. The FT benchmark victim depends on the futex table because it contains 64 threads which continuously acquire a shared lock for 100 microseconds. The FT attacker targets the futex table by allocating thousands of futex variables, probing the hash buckets to identify a busy bucket, and then parking thousands of threads on that hash bucket; as a framing attack, the attacker turns passive after parking the threads

Detection

For the IC benchmark, Tratr detects the attack within 1 s after it starts. To provide context on how good that is, by 1 s after the attack started, the performance of the mail server only decreased by <5%, whereas in the case it hadn’t been detected it would have gone down to 90% within a few 10s of seconds. For the FT benchmark, while the attacker is still probing the buckets the LCS would not be triggered yet, but once the attacker parks some thousands of threads in the bucket, the attack is detected within 1s.

Prevention and Recovery

There is a slight dip in the throughput when an attack is detected and before recovery completes. As Tratr needs to access the synchronization primitives to perform recovery, the victim observes a slight dip in the performance. Tratr is run under two configurations, one where only detection and prevention are enabled and the other where all three mechanisms, detection, prevention and recovery are enabled. In the experiment where only prevention is enabled and no recovery, we see that the throughput performance continues to drop whereas in the case where recovery is enabled the throughput sees a dip for a brief moment and then again rises up. Prevention measures are necessary for faster recovery. Without the prevention measures, an attacker can continue to expand the data structure while holding the synchronization primitives. Since recovery must also acquire these synchronization primitives, longer lock hold times will make the recovery mechanism a victim.

Kernel Threads Overhead

In the detection phase, the kernel threads sample the locks and hence could be competing with the application for CPU. To measure the impact of the kernel threads they run three CPU heavy workloads: N-Queens, CP2k Molecular Dynamics, and Primesieve. They run the same application in 4 containers and allocate 8 CPUs to each one of them. For all three applications, the performance difference between the maximum runtime with Trātṛ and the minimum runtime with Vanilla kernel is around 1-1.5%. On average, the kernel threads spend 120 µs sampling the synchronization primitives every 12.5 ms (approximately 1%).