Contents

1 Introduction

Delegation locking adapts the request-response style of communication to minimize shared data movement. Specifically, the waiter delegates their critical section to a combiner (Flat-Combining/ CC-Synch) [1, 2]

, or a dedicated server thread [3, 4]

. We will describe some implementation details of Flat-Combining , CC-Synch, and RCL in Section 2.

It's important to highlight that the concept of the *lock slice*, as introduced in U-SCL, also intersects with the idea behinds delegation-style locks. *Lock slice* ensures that only a single thread executes the critical section within a given time slice. This design reduces the necessity for transferring shared data across threads/cores, leading to enhanced performance [5]. Notably, this approach aligns closely with the objectives of delegation-style locks, resulting in comparable performance improvements.

However, the *lock slice* strategy presents a significant limitation: during a valid *lock slice*, no other thread can hold the lock, even if the preceding thread has released it. Consequently, the non-critical section must be concise to prevent a substantial impact on throughput. In contrast, delegation-style locks offer similar performance benefits without the associated drawback of the *lock slice* strategy. This distinction underscores the trade-offs involved in different locking mechanisms and their impact on performance and concurrency.

The central obstacle associated with delegation-style locks revolves around the need to refactor pre-existing code. This arises from the fact that delegation-style locks do not provide a standard locking API. In response to this concern, RCL has been developed as a solution. RCL offers a code migration aid, facilitating the seamless transition of legacy code to one that employs RCL'

s locking mechanism [3]. Moreover, recent research efforts have demonstrated the viability of transparent delegation to provide delegation-style lock in standard locking API [6].

The majority of delegation-style locks inherently offer a degree of fairness guarantee. The rationale behind this lies in the notion that the combiner generally treat its critical sections similarly to those of other threads. As a result, the prevailing approach is to enumerate all ready jobs. For instance, in the case of Flat-Combining, the combiner (or the server of RCL) scans through all the thread lists to

determine if a thread is attempting to execute a critical section. This equitable treatment is not mirrored in spin-locks, where certain threads might dominate lock usage, causing others to be starved. Moreover, when threads repetitively acquire the lock, the thread releasing the lock gains an advantage in reacquiring it.

However, previous work has demonstrated that the acquisition fairness of lock is not enough to mitigate the problem of usage fairness and scheduler subversion [5]. For example, Figure 1 has demonstrated the unfairness of CC-Synch

even if it maintains a strictly FIFO order given varying critical section sizes. The 16 threads that are incrementing a shared counter are split into two groups: the first group will run for 10ns, and the second group will run for 30ns. We can easily see that threads in the two groups contribute to the shared counter differently — proportion to their critical section size.

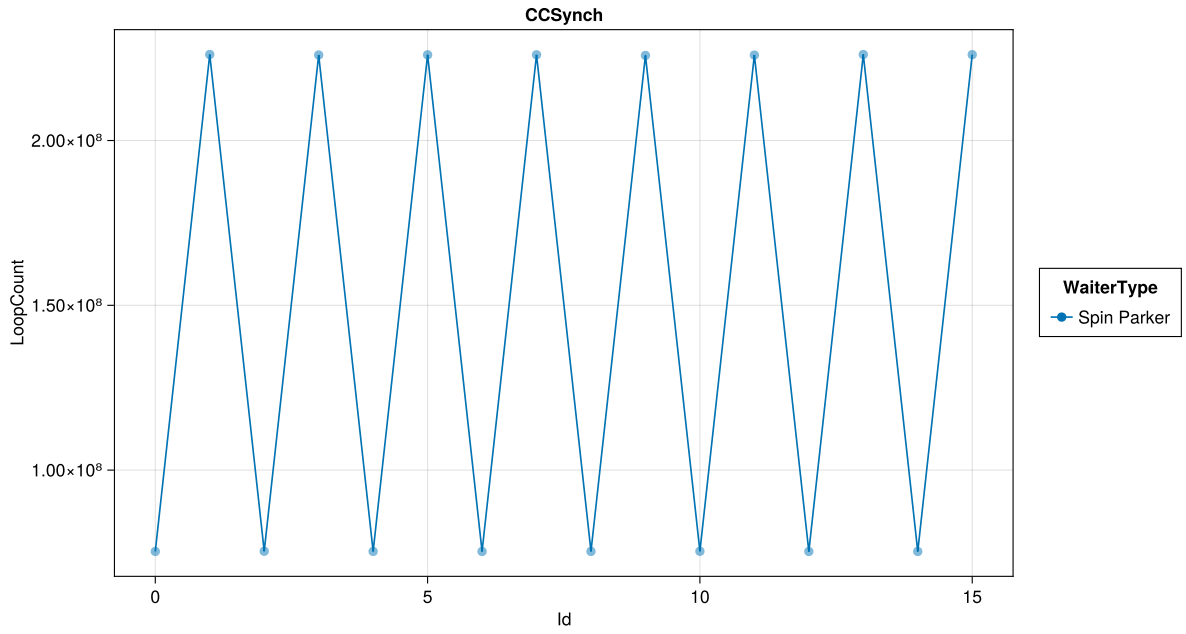


Figure 1: #ccsynch Loop Count per thread

To address these challenges, we have made modifications to the original algorithms of Flat-Combining and CC-Synch, focusing on achieving our objective of usage fairness. Our approach centers on two primary strategies aimed at enhancing fairness.

1.1 Banning

Similar to how U-SCL is implemented, we ban the thread that is trying to execute the critical sections for too long. Presently, we've developed two equitable variations of Flat-Combining

and CC-Synch, both stemming from this concept.

The banning time is calculated as below:

$$\max\left(0, \mathrm{cs} \times n_{\mathrm{thread}} - \mathrm{cs}_{\mathrm{avg}}\right)$$

where the average critical sections are calculated incrementally.

$$\mathrm{cs}_{\mathrm{avg}} \leftarrow \mathrm{cs}_{\mathrm{avg}} + \frac{\mathrm{cs} - \mathrm{cs}_{\mathrm{avg}}}{n_{\mathrm{exec}}}$$

The rationale behind maintaining an additional average critical section usage is to address situations where only a few threads share similar critical section lengths. This approach helps minimize the duration during which all threads are banned.

1.2 Priority Based Structure

On the other hand, we can replace the linear concurrent data structure with a prioritized one. For instance, in the context of Flat Combining, the current linear thread list could be substituted with a priority queue or a balance tree. This modification would allow us to prioritize threads based on their lock usage, similar to how Linux CFS prioritizes threads based on their execution time. This approach aligns well with delegation-style locks, given that the combiner naturally acts as a scheduler. A more detailed exploration of this approach will be presented in the section labeled Section 2.3.2.

2 Implementation Details

This work implement the following delegation-style locks and their variants in *rust* (so we assume some knowlege toward *rust*): Flat-Combining [1] , CC-Synch [2], RCL [3].

2.1 Job Type

Each lock implement a common trait (interface) as follows, where the generic type represents the shared data:

```rust
pub trait DLock<T> {
    fn lock<'a>(&self, f: impl DLockDelegate<T> + 'a);

    #[cfg(feature = "combiner_stat")]
```

```
    fn get_current_thread_combining_time(&self) -> Option<std::num::NonZeroI64>;
}
```

and the trait `DLockDelegate` refers to the critical sections defined as follows:

```
pub trait DLockDelegate<T>: Send + Sync {
    fn apply(&mut self, data: DLockGuard<T>);
}
```

The `DLockGuard<T>` is similar to the `MutexGuard<T>` in *rust*, which implements the `Deref` and `DerefMut`

traits. The `DLockGuard` is used to access the shared data. Unlike `MutexGuard` , which is also used to release the lock when dropped, the lock is automatically released when the delegate finishes.

```
pub struct DLockGuard<'a, T: ?Sized> {
    data: &'a SyncUnsafeCell<T>, // UnsafeCell is used to allow interior mutability
}
```

To make the lock easy to use, we implement `DLockDelegate<T>`

for lambda function type `FnMut(DLockGuard<T>) + Send + Sync`

. Thus, the user can use the lock as follows:

```
lock_type.lock(|mut guard: DLockGuard<u64>| {
    ...
});
```

2.2 Delegation-style Locks

To implement a delegation-style lock, two aspects need consideration:

1. How jobs are published.
2. How the combiner is selected.

There are two natural ways to publish jobs: per thread or per job.

For"per thread", each thread owns a dedicated memory location (node) that enables it to publish a job with its context. Then the combiner enumerate the threads' nodes, check whether the node is ready, and, if ready, execute the job. This method is adopted by Flat-Combining (Section 2.2.1), RCL (Section 2.2.2), and ffwd [4] .

On the other hand, in the"per job" approach, each thread publishes the job for execution to a dedicated job queue. The combiner then traverses the job queues to execute the jobs. This approach is embraced by CC-Synch and its variants (Section 2.2.3).

2.2.1 Flat-Combining

Flat-Combining upholds a list of nodes owned by each thread. Each node incorporates the thread's context and the job it intends to execute. The combiner iterates through this list of nodes, evaluating their readiness. If a node is deemed ready, the combiner proceeds to execute the job and subsequently updates the node to signal the job's completion.

The struct is defined as follows:

```
#[derive(Debug)]
pub struct FcLock<T, L, P>
where
    L: RawSimpleLock,
    P: Parker,
{
    pass: AtomicU32,
    combiner_lock: CachePadded<L>,
    data: SyncUnsafeCell<T>,
    head: AtomicPtr<Node<T, P>>,
    local_node: ThreadLocal<SyncUnsafeCell<Node<T, P>>>,
}
```

- The `pass` field serves to indicate the age of the lock, making it possible to remove threads that have not executed a job for an extended period.
- The `head` field is the head of the list of thread nodes.
- The `local_node` field is designed to store the current thread's local node, akin to `pthread_getspecific`.

- The `combiner_lock` is utilized for thread-based combiner election. The implementation idea is retrieved from [7] (code are avaliable in Section 4.2).

The node struct is defined as follows:

```
pub(super) struct Node<T, P>
where
    P: Parker,
{
    pub(super) age: u32,
    pub(super) active: AtomicBool,
    pub(super) f: CachePadded<Option<*mut (dyn DLockDelegate<T>)>>,
    pub(super) next: *mut Node<T, P>,
    pub(super) parker: P, // id: i32,
    #[cfg(feature = "combiner_stat")]
    pub(super) combiner_time_stat: i64,
}
```

- The `age` field synchronizes with the `pass` field of the lock, indicating the age of the node. If the `age` is significantly different from `pass`, the node will be removed from the linked list and marked as inactive (false in the `active` field).
- The `f` field represents the critical section.
- The `next` field is the pointer to the next node in the linkedlist.
- The `parker` field is the parker that is used to block the thread when job is published but not yet executed (refer to Section 2.4).
- The `combiner_time_stat` field is used to record the time that each thread becomes the combiner. It is not marked with `Atomic` because only the combiner (which is guarded by the `combiner_lock`) will overwrite this field (maybe volatile is needed?).

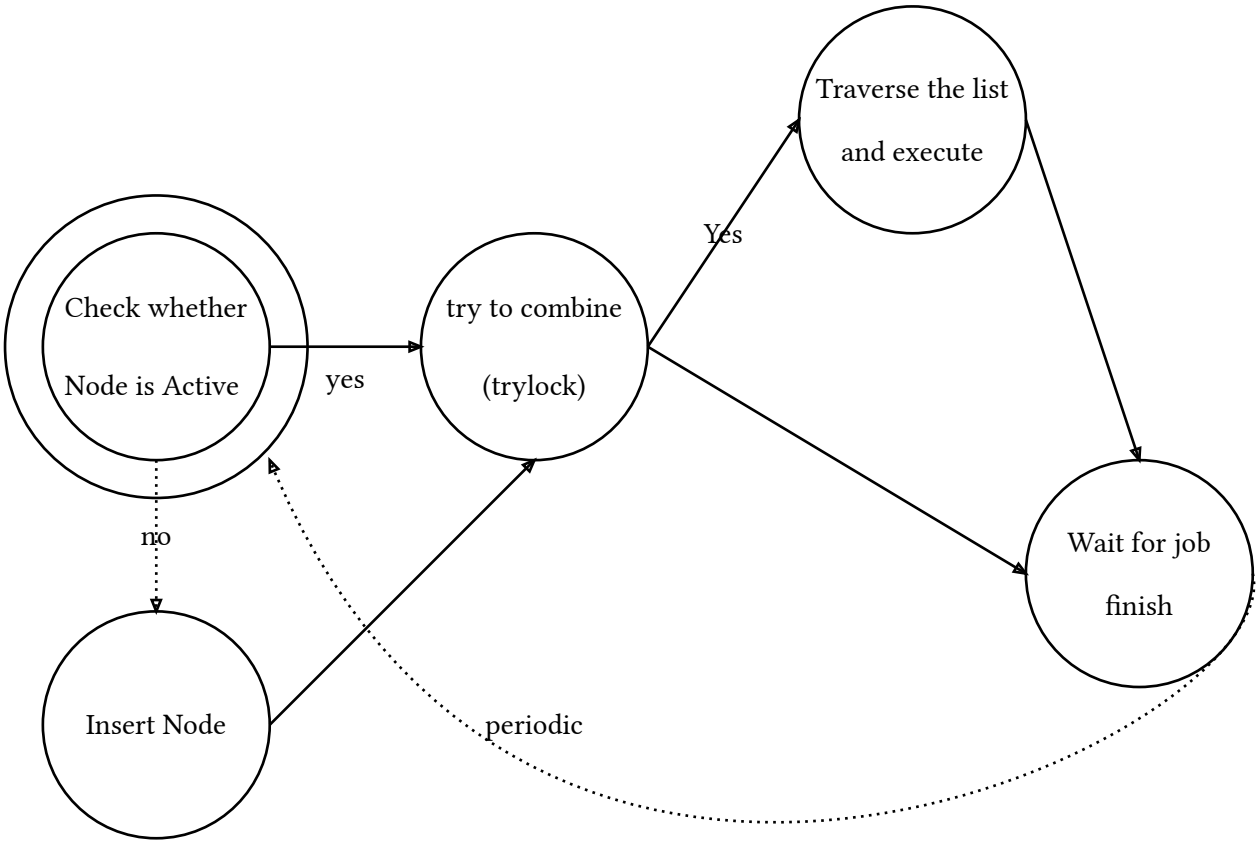
The original paper proposed that the ready state can be embedded into `f` field, thus saving a boolean field [1]. However, we choose not to do this in our implementation for two reason.

1. In *rust*, we encapsulate the job into a trait object. A pointer pointing to a trait object is a fat pointer (which contains two pointer). Atomic operation toward such double-sized field are not available universally. Introducing another layer of indirection that points to the fat pointer to make it accessible atomically will involve additional cost. With an additional bool field used as a

write-fence (`Release` Ordering), we ensure that the combiner views the entire job before the node is perceived as ready.

2. We aim to extract the waiting process into a `parker` field, enabling the sharing of blocking code across different locks. However, embedding the ready state into the `f` field would hinder this capability.

The overall algorithm can be roughly represented by the following graph:



The insertion is done via a simple CAS loop as a singly linked list:

```
fn push_node(&self, node: &mut Node<T, P>) {
    let mut head = self.head.load(Ordering::Acquire);
    loop {
        node.next = head;
        match self
            .head
            .compare_exchange_weak(head, node, Ordering::Release, Ordering::Acquire)
        {
            Ok(_) => {
                node.active.store(true, Ordering::Release);
```

```
                break;
            }
            Err(x) => head = x,
        }
    }
}
```

The combiner will regularly remove inactive nodes. Subtly, the head of the list won't be removed to ensure that removal can be safely performed even when new nodes are being added.

2.2.2 Remote Core Locking

Similar to Flat-Combining, RCL also maintains a list of nodes owned by each thread. The primary distinction is that RCL designates a specific thread to act as the server (combiner). Consequently, it's akin to a remote procedure call, as proposed in the original paper. Notably, RCL utilizes an array of nodes, whereas Flat-Combining uses a linked list of nodes. This design difference in RCL should yield improved performance when the number of inactive threads is relatively small.

One of the primary goals of RCL is to facilitate easy migration of legacy code. To achieve this, RCL employs a complex algorithm to ensure that the server thread won't be blocked or engaged in active spinning, which could negatively impact performance. Additionally, RCL's design enables the server to handle multiple locks, allowing for straightforward porting of legacy code. While our implementation doesn't cover the complete version of RCL and thus won't delve into these aspects, one core goal of enabling one server to manage various types of locks is retained. Achieving this presents a significant challenge within the Rust type system, as we aim to statically dispatch type information to prevent performance regression. Further implementation details can be found in the appendix labeled Section 4.3.

2.2.3 CC-Synch/H-Synch

CC-Synch maintains a first-in, first-out (FIFO) queue of jobs. Unlike Flat-Combining, where each node corresponds to a thread, in CC-Synch

each node signifies a job. Additionally, the selection of the combiner is achieved without involving a lock; rather, it's accomplished in a lock-free manner, relying on the position of the job queue.

Intuitively, a thread will be elected as a combiner if one of the following conditions is satisfied: 1. Its job is positioned at the end of the queue. 2. Previous Combiner has reached the execution limit.

In contrast to FlatCombining, it's important to note that an execution limit is necessary to ensure that the combiner does not execute an excessive number of critical sections. The job queue has the potential to extend infinitely, thereby causing the combiner to possibly prioritize jobs from the queue indefinitely, potentially neglecting its own task. In contrast, the maximum number of jobs performed by FlatCombining's combiner is constrained by the number of threads, which we can assume to be finite.

The algorithms can be outlined as follows:

1. Each thread maintains a mutable thread-local pointer to a node. Note that, unlike in FlatCombining where the node always belongs to the creating thread, in this scheme, nodes can be accessed and modified by different threads.
2. When a thread intends to execute a critical section:
    1. It swaps its pointer storage with the current head node.
    2. The task is then assigned to the old head node that was just swapped, and this node becomes connected to the previous node (which is now the new head).
3. To ensure lock-free operation, a somewhat intricate mechanism is employed:
    1. Each node has two properties: `wait` and `completed`
    2. Initially, `wait` and `completed` are set to false.
    3. Before a thread swaps its thread-local node to head, it sets the `wait` property to true. The old-head then becomes the thread-local node of the thread.
    4. Once the task assignment is done, the thread checks the `wait`

        property. If it' s true, the thread waits until it becomes false.
    5. Once wait becomes false, the thread examines the completed property. If the task has not been completed, it assumes the role of the combiner.
    6. The combiner will stop execution at `head` node, which doesn'

        t have children (there will always be an extra dummy node for swapping). Mark its wait as false, and exit.

H-Synch, although not currently implemented, represents a NUMA-aware variant of CC-Synch . The underlying concept is straightforward: it introduces multiple queues based on NUMA nodes, and each combiner competes for execution on the queue specific to its cluster. This approach takes into account the NUMA architecture to optimize task execution.

2.3 Fair delegation-style locks

We present two approaches for implementing fair delegation-style locks. One approach is founded on the concept of banning, akin to the implementation of SCLs (as detailed in [5]). The other approach relies on a priority-based structure.

2.3.1 Banning

The concept behind banning is to disallow a thread from executing its critical section if it has been attempting to do so for an extended duration. Implementing banning is not particularly unique for delegation-style locks. However, due to its inherent alignment with creating a straightforward fair lock, we have proceeded to implement a fair version of Flat-Combining

and CC-Synch based on this banning approach.

2.3.1.1 Flat-Combining (Banning)

Drawing inspiration from the banning concept, the combiner calculates the time taken to execute critical sections and determines when a thread should be banned, following the algorithm outlined in Section 1.1. Similar to the implementation of U-SCL, only the timestamp at which the thread can resume execution is recorded. This mechanism aims to ensure fairness by managing thread execution times.

When iterating through the thread list, the combiner will initially assess whether a thread is banned. Only after this evaluation will the combiner proceed with the standard Flat-Combining

algorithm. This additional step ensures that fairness is upheld by considering the ban status of each thread before applying the regular execution logic.

2.3.1.1.1 Drawback and Potential Resolution

It's important to acknowledge that this approach comes with an additional overhead. More nodes will be flagged as"

inactive" due to being banned. One possible solution is to eliminate banned threads from the list entirely, although this entails extra maintenance costs for the list itself. Employing heuristic algorithms could offer a strategy for deciding when nodes should be removed from the list, helping to manage these complexities.

Another nuanced consideration is to delegate the responsibility of determining whether a thread is banned to the waiter itself, instead of burdening the combiner with the task of verifying a thread' s ban status. This approach mirrors the implementation of CC-Synch (Banning)

, as detailed in Section 2.3.1.2, albeit introducing the cost of transmitting critical section lengths of jobs.

Considering that we've implemented a pre-wake mechanism, an alternative solution is to make the waiter record the timestamp when it is pre-woken. Subsequently, when the waiter is ultimately awakened, it can calculate the critical section duration. This technique aligns with the goal of maintaining fairness while minimizing the combiner' s involvement in ban checks. One issue regarding this is that waking from blocking might take times, which will make the recorded critical section slightly shorter than actual.

2.3.1.2 CC-Synch (Banning)

In contrast to the approach used in Flat-Combining (Banning), it would prove challenging for a combiner to skip a node when its corresponding thread is banned. Instead, each waiting thread will cease inserting its node into the job queue when it becomes banned. The combiner's role will involve recording the length of the critical section and communicating this information to the waiting threads. Subsequently, the waiters themselves will determine if they have been banned using the algorithm detailed in Section 1.1 to stop inserting job.

It's evident that the banning strategy employed here imposes a lesser performance penalty in comparison to Flat-Combining (Banning), primarily because combiners don't have to navigate through redundant nodes. Subtly, this trade-off still implies that a slightly larger number of threads might be necessary to achieve an equivalent number of combined jobs as in the case of CC-Synch.

2.3.2 Priority-based Structure

The concept of a priority-based structure involves substituting a linear data arrangement with a prioritized counterpart. As an instance, in the case of FlatCombining, a linear thread list is sustained. This linear arrangement can be supplanted with a Priority Queue, enabling the prioritization of threads according to their lock usage.

Concurrently, the job queue established in CC-Synch can be exchanged with a concurrent priority queue. A comparable approach is found in the Linux Completely Fair Scheduler (CFS), where a red-black tree is utilized to prioritize threads based on their execution duration. This approach aligns well with delegation-style locks, as it entails a scheduler-like entity: the combiner. In contrast to a straightforward prohibition mechanism, this method showcases a more refined nature and has the potential to produce a more intricate data structure.

Moreover, the response time demonstrates an inverse correlation with the usage of thread locks, a quality that holds significant benefits in various contexts. Conversely, banning lacks the ability to generate such promise.

2.3.2.1 Flat-Combining (Priority-Queue) (Not implemented)

We can substitute the linked-list structure in FlatCombining with a priority queue, which could be either a heap or a balanced tree. However, these options lack inherent concurrency like the linked list, necessitating either the enforcement of mutual exclusion mechanisms or a concurrent variant. Fortunately, there is a natural entity – the combiner – satisfying the second requirement.

Similar to the approach in Flat-Combining, we continue to employ a thread-local node to publish tasks. However, a key distinction emerges: the thread now efficiently pushes the node into a prepared queue (linked list). Subsequently, the combiner takes on the role of popping the node from this queue and transferring the thread's node into the priority queue.

At the outset of each iteration, the combiner undertakes a check to determine whether additional prepared threads are awaiting insertion into the priority queue. Should such threads exist, the combiner retrieves them from the prepared queue and integrates them into the priority queue.

Following this preparatory phase, job execution follows a pattern akin to Flat-Combining, utilizing the prioritized data structure to facilitate fairness.

2.3.2.2 Flat-Combining (Skip-List)

In the previous section, there's an alternative by utilizing a concurrent proritized data structure to implement a fair flat combining. This section present an slightly different implementation stemed from that idea.

We can maintain a job queue that is prioritized based on the critical section length with the help of a concurrent skip list, similar to *Linux CFS*. The combiner will be elected as Flat-Combining and then execute the job from the head of the priority queue.

This represents one of the simplest ideas when seeking to apply the priority-based structure concept to create a fair delegation-style lock. However, there exists a subtle distinction from Flat-Combining , as this approach chooses a job queue (similar to CC-Synch

) instead of a thread queue.

Drawing parallels with CC-Synch, the imposition of a combiner limit is crucial here, as the potential exists for an indefinite number of jobs awaiting execution. Our present implementation hinges on a" contribution" limit, analogous to the"H" limit in CC-Synch . However, in this case, the limit is calculated based on timestamps as opposed to the number of jobs.

2.3.2.2.1 Future work

The existing implementation is dependent on a third-party concurrently designed skip list, specifically [8]. This suggests the possibility of creating a more optimized rendition of the skip list to elect the combiner in a manner similar to CC-Synch.

2.3.3 Response Time

Delegation-style locks introduce an additional layer of unfairness beyond the conventional concern of unfairness arising from lock usage. The combiner, responsible for executing tasks on behalf of others, becomes subject to a significant impact on its own response time. To illustrate, consider the case of Flat-Combining , where the combiner for a lock must endure a wait time that can be twice the average response time of the threads it's assisting. This situation becomes particularly problematic in scenarios where the waiters are blocked, as evidenced in the left plot of the Figure 2. In this specific example, two out of the 32 threads are engaging in combining for over 80% of the

time. This issue escalates further as the thread count increases. (Note: When developing the initial version of Flat-Combining in Rust, it was observed that the blocking variant of Flat-Combining outperformed the spin-wait version, and an analysis of combining time offered insights into this phenomenon.) CC-Synch also suffers from this issue, which is only noticable with 32 threads.

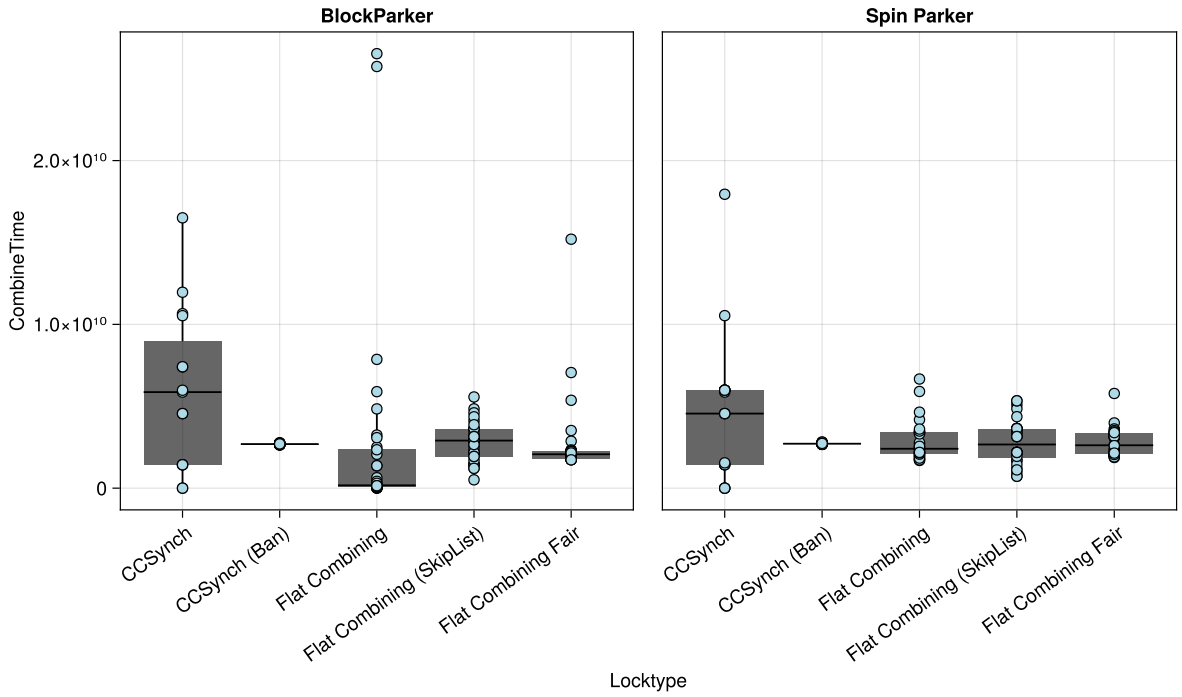


Figure 2: Combining Time Distribution of 32 threads execution (0ns critical section)

This insight underscores the significance of not only optimizing the scheduling of waiters but also addressing the scheduling of the combiner itself. Furthermore, it suggests that we can mitigate average response times by thoughtfully selecting which thread assumes the role of the combiner. A potential is the deliberate choice to designate the tail of the threads list as the combiner in Flat-Combining .

Due to the"flat" execution nature, the tail node is consistently executed last, implying that its response time equates to the cumulative sum of all critical sections within the current pass. Consequently, adopting the role of a combiner does not impose any penalty on its response time (if there's no combiner executing). Conversely, if the head node were to become the combiner, its response time would experience a significant regression, shifting from its own critical section length to the summation of all critical sections within the current pass.

Experiments regarding response time hasn't been added, and all these ideas also haven't been implemented.

2.3.3.1 Combiner Slice

The concept of the"Combiner slice" draws parallels with the notion of the"lock slice," albeit with slight variations. In this context, a thread is designated as the combiner within a designated time slice. While this approach could potentially yield performance enhancements by reducing combiner contention, it does face the same challenge as the"

lock slice": the non-critical section must be kept sufficiently brief.

TODO!

2.3.3.2 Flat-Combining (Banning, Combiner-Slice)

TODO! This is not a good implementation and is not an implementation of the combiner slice I describe above.

2.4 Parker

Irrespective of the specific variations in delegation-style locks, a common requirement emerges: the necessity for a waiter to wait after dispatch a job. Leveraging the benefits of zero-cost abstraction in Rust, it's possible to effectively implement a generic parker as a trait (interface). This versatile trait can then be seamlessly employed by all of our delegation-style locks.

The following are a few specifications that we want to achieve with our parker *trait* to make it usable for all kinds of delegation-style locks:

1. The parker is designed under the assumption that it will be used by a single waiter and a single waker, and thus it avoids incorporating complex mechanisms.
2. Unlike the park/park_timeout functions in the thread module of Rust, the parker relies on manual reset instead of automatic reset once a wait/wake pair is encountered.
3. The parker is engineered to provide visibility into whether a parked waiter is present.
4. Similar to the `park/unpark` implemented in the thread module of Rust, this `parker` should offer the capability to `wait` or `wait_timeout`. Waiters should be informed of the reason behind their wake-up.

5. A pre-wake mechanism is implemented, enabling a combiner to prompt the waiter to wake up before its job is completed. This is akin to the pre-fetching mechanism outlined in [5].

Thus, we design the trait `parker` as follows:

```
pub trait Parker: Debug + Default {
    fn wait(&self);
    fn wait_timeout(&self, timeout: Duration) -> Result<(), ()>;
    fn wake(&self);
    fn state(&self) -> State;
    fn reset(&self);
    fn prewake(&self);
    fn name() -> &'static str;
}
```

`State` is defined as follows:

```
#[derive(PartialEq, Eq, Debug)]
pub enum State {
    Empty,
    Parked,
    Prenotified,
    Notified,
}
```

Currently, two variants of parker are implemented: SpinParker and BlockParker. Naturally, we should have a 3rd SpinThenBlockParker, but given that our critical section is long, it will not show much performance benefits compared to BlockParker. Thus, we leave it in the future.

2.4.1 Spin Parker

The `SpinParker` is implemented as actively spinning with exponential backoff, and contains only an atomic integer to represent the state.

```
#[derive(Default, Debug)]
pub struct SpinParker {
```

```
    state: AtomicU32
}
```

The code implementation can be found in Section 4.4. The state changes are primarily achieved using the `compare_exchange`

operation. This is necessitated by the inclusion of an extra `Prenotified`

state. When threads are pre-woken, the behavior is altered to `spin_then_yield` . While our experiments didn't yield a substantial performance improvement with this modification, we chose to implement it as a prototype nonetheless.

2.4.2 Block Parker

Block Parker is essentially a straightforward abstraction built on top of an `AtomicU32` and is implemented using `Futex` synchronization mechanisms. The decision to avoid using the `park/unpark` functionality provided by *Rust/std* is driven by the following factors: + A field to record the `ThreadId` will be required. + It will involve an additional `AtomicU32` if we want to implement the pre-wake mechanism.

This cost are mainly trivial and probably should not be the reason of doing optimization. However, the `Futex` is flexible enough to allow us to implement the parker given we are only caring about *linux*.

The pre-wake mechanism initiates the wake-up of a waiting thread if it's currently parked. Subsequently, the thread engages in a spin wait with exponential backoff. Although the `spin_then_yield` approach might be more advantageous given the absence of contention on the state field, we presently continue to employ this strategy as no significant performance impact has been observed.

3 Experiments

We benchmark the performance by incrementing a shared counter for a given time slice.

The following locks are benchmarked in our example: + Flat-Combining

1. Flat-Combining (Banning)
2. Flat-Combining (Banning, Combiner-Slice) (Please ignore now)
3. Flat-Combining (Skip-List)

4. CC-Synch
5. CC-Synch (Banning)
6. RCL
7. Mutex (rust)
8. Spinlock (Section 4.2)
9. U-SCL ([5])

The experiment was conducted on an AMD EPYC 7302P machine, featuring 16 cores and 32 threads with hyperthreading enabled.

The worker threads were divided into two distinct groups. The first group executed for a duration of 10us, while the second group operated for 30us. Two sets of experiments were undertaken. The first set featured zero non-critical sections. In the second set, the non-critical sections were lengthy, where threads would be put to sleep for 10us.

Since all the delegation-style locks were implemented using a generic parker, the experiment' s results were segregated based on the two parker variants.

3.1 General Performance Comparison

The illustration labeled Figure 3 showcases the performance of the locks while attempting to increment a shared counter within non-critical sections lasting 10/30 nanoseconds.

Notably, the performance of U-SCL remains consistently favorable across varying thread counts. Conversely, other locks exhibit certain performance degradation when waiters resort to spinning. An exception to this trend is CC-Synch (Banning)

, which has peformance issue when 32 threads are running. This is likely due to the fact that banned threads are spining when being banned.

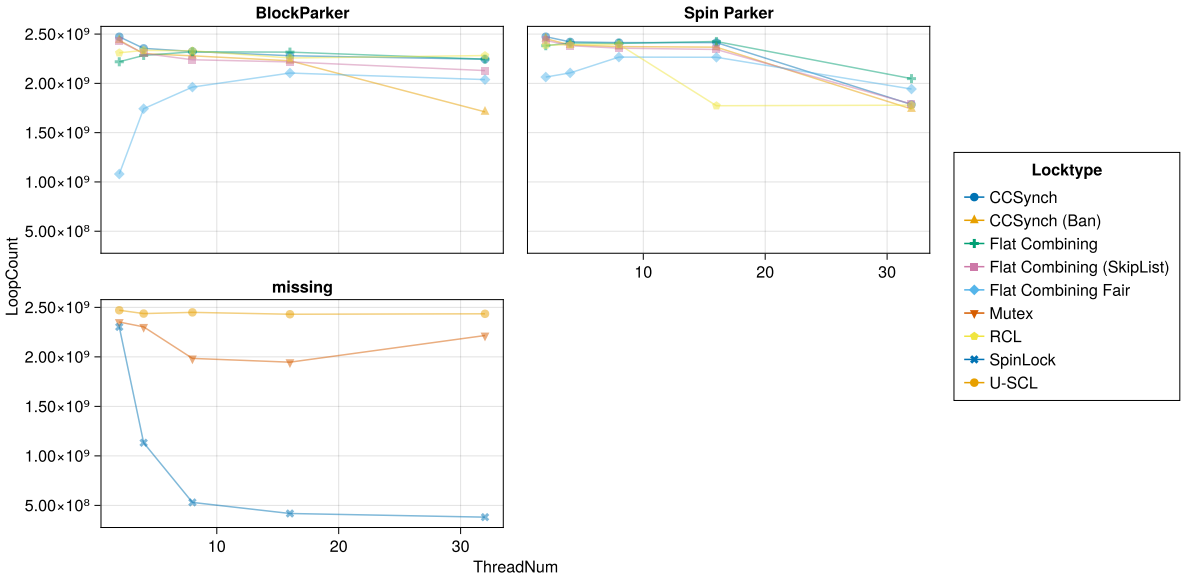


Figure 3: Performance Comparison (non-critical section 0ns)

Conversely, the performance of U-SCL experiences a significant decline when the size of the non-critical section becomes substantial. The illustration designated as Figure 4 demonstrates the lock performance when the non-critical section spans 10us. Remarkably, with a 10us sleep following the execution of the critical section, the throughput of U-SCL plunges by over 2 times.

In contrast, the performance of the other locks remains relatively stable, with no substantial drop observed. This discrepancy in performance behavior highlights the impact of varying non-critical section sizes on the effectiveness of U-SCL compared to the other lock implementations due to the use of *lock slice*.

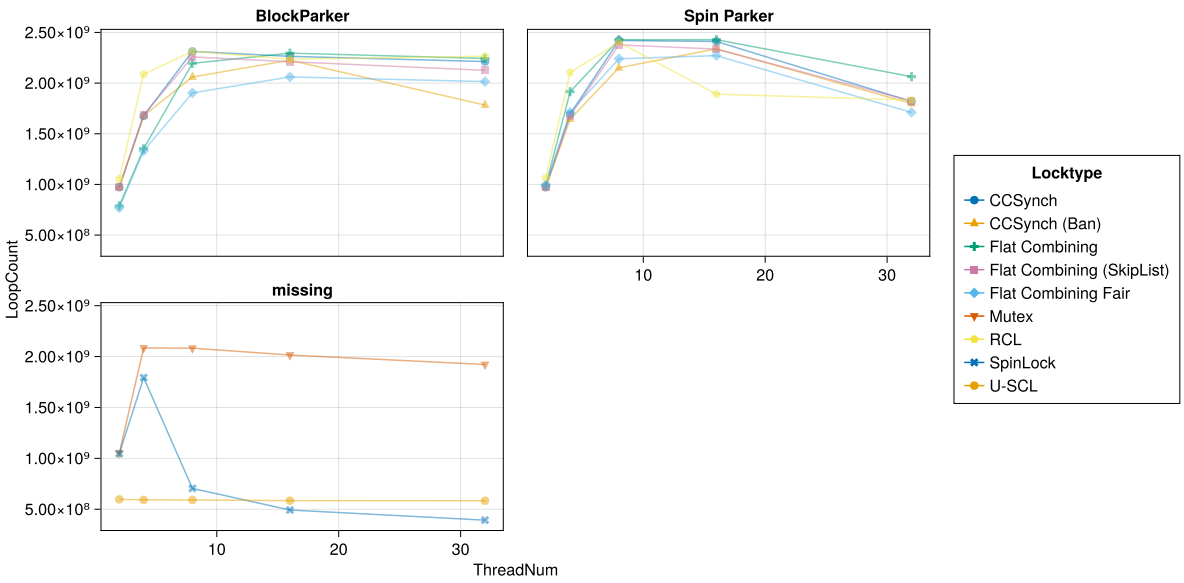


Figure 4: Performance Comparison (non-critical section 10ns)

3.2 Fairness Comparison

Furthermore, this work places significant emphasis on the aspect of fairness in lock utilization. Figure 5 provides an overview of how the locks are used by individual threads. It is evident that both the"banning" variants and FLAT-COMBINING (Skip-List) successfully achieve the intended goal of usage fairness, distributing the lock usage more evenly among threads.

A subtle degree of unfairness is still observable in our fair lock implementations, particularly when threads are blocking rather than spinning. Several factors could potentially contribute to this observed level of unfairness:

Threads are not instantly awakened once their jobs are completed, particularly in the case of blocking. This dynamic isn't factored into the lock usage calculations. The performance of a single 30ns critical section may naturally outperform that of three separate 10ns critical sections. This discrepancy in critical section lengths could play a role in the number of execution, thus enlarge the discrepency.

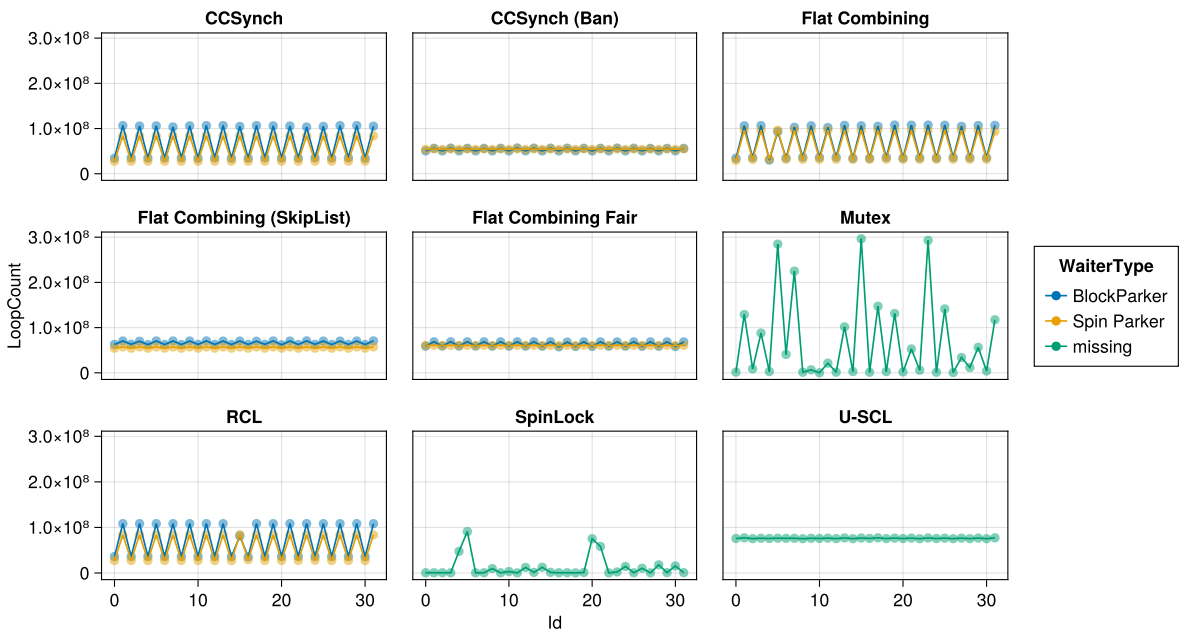


Figure 5: Fairness Comparison (non-critical section 0ns)

The scenario becomes intriguing when the non-critical section is not empty. Notably, U-SCL demonstrates behavior akin to acquisition-fair lock, while our fair variants of Flat-Combining and CC-Synch continue to maintain a higher level of usage-fairness. Within the realm of fair locks, the blocking versions display slightly greater degrees of unfairness compared to their spinning counterparts. This could potentially be attributed to the occurrence of double yielding when threads are put to sleep.

In this context, spin-lock now emulates a form of pseudo-acquisition fairness, as threads yield back once they conclude a critical section.

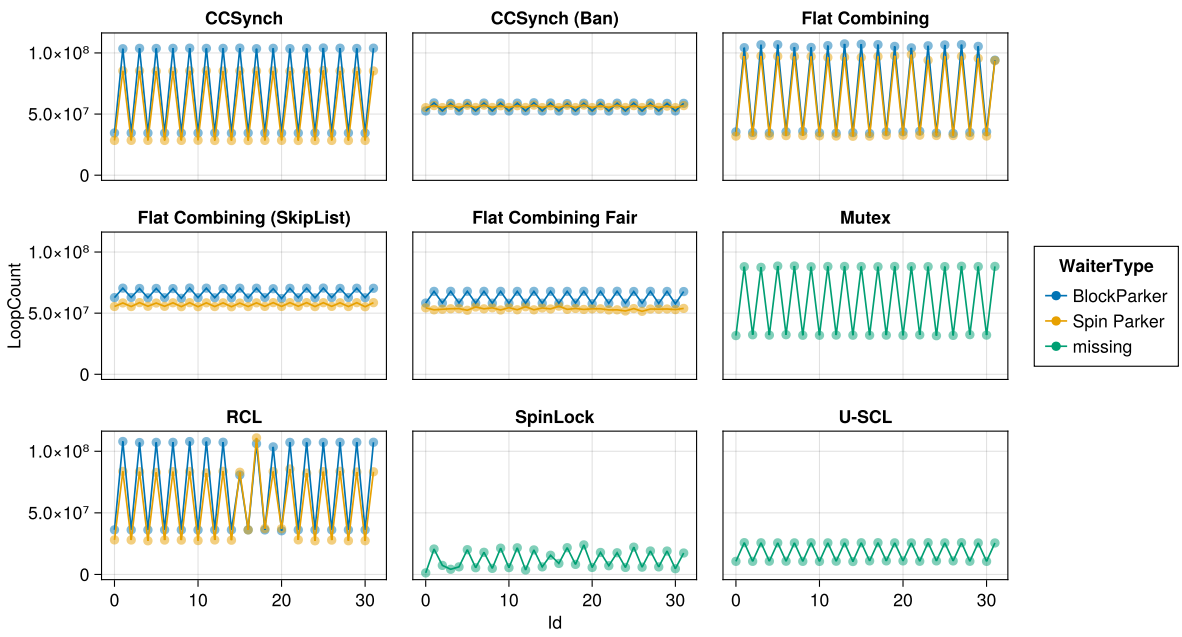


Figure 6: Fairness Comparison (non-critical section 10ns)

Bibliography

[1] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. Twenty-Second Annu. ACM Symp. Parallelism Algorithms Architectures*, 2010, pp. 355–364.

[2] P. Fatourou, and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 257–266.

[3] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: migrating $\{$critical-section$\}$ execution to improve the performance of multithreaded applications," in *2012 USENIX Annu. Tech. Conf. (USENIX ATC 12)*, 2012, pp. 65–76.

[4] S. Roghanchi, J. Eriksson, and N. Basu, "Ffwd: delegation is (much) faster than you think," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 342–358.

[5] Y. Patel, L. Yang, et al., "Avoiding scheduler subversion using scheduler-cooperative locks," in *Proc. Fifteenth Eur. Conf. Comput. Syst.*, 2020, pp. 1–17.

[6] V. Gupta, K. K. Dwivedi, et al., "Ship your critical section, not your data: enabling transparent delegation with $\{$tclocks$\}$," in *17th USENIX Symp. Operating Syst. Des. Implementation (OSDI 23)*, 2023, pp. 1–16.

[7] Taewoo An (codingskynet), *Concurrent Data Structure for Rust*, (2023). [Online]. Available: https://github.com/codingskynet/concurrent-data-structure

[8] Taiki Endo, Jules Bertholet, et al., *Crossbeam Skiplist*, (2023), crossbeam-rs. [Online]. Available: https://github.com/crossbeam-rs/crossbeam/tree/master/crossbeam-skiplist

[9] *Crossbeam*, (2023), crossbeam-rs. [Online]. Available: https://github.com/crossbeam-rs/crossbeam

[10] Mara Bos, *Linux-Futex*, (2023). [Online]. Available: https://github.com/m-ou-se/linux-futex

4 Appendix

4.1 Implementation difficulty in *Rust*

TODO!

4.2 RawSpinLock

The RawSpinLock is implemented as a test-and-test-and-set lock with exponential backoff:

```
#[derive(Debug)]
pub struct RawSpinLock {
    flag: AtomicBool,
}

unsafe impl RawSimpleLock for RawSpinLock {
    fn new() -> Self {
        Self {
            flag: AtomicBool::new(false),
        }
    }

    #[inline]
    fn try_lock(&self) -> bool {
        if !self.flag.load_consume() {
            self.flag
                .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
                .is_ok()
        } else {
            false
        }
    }

    #[inline]
    fn lock(&self) {
        let backoff = Backoff::new();

        while !self.try_lock() {
            backoff.snooze();
        }
    }

    #[inline]
    fn unlock(&self) {
        self.flag.store(false, Ordering::Release);
    }
}
```

4.2.1 Remark

The additional load doesn't seem to provide better performance to FlatCombining given that most of the time the thread will only `try_lock` once and then block for a while. However, it does provide better performance when used completely alone.

4.3 RCL Implementation Details

TODO!

4.4 Spin Parker Code

Spin Parker is implemented with expoentially backoff from CrossBeam [9]. Note that the `PreNotified` state greatly complex the code. Without it, a `swap` is enough to implement it.

```rust
use crate::parker::Parker;
use crossbeam::utils::Backoff;
use quanta::Clock;
use std::cell::SyncUnsafeCell;
use std::hint::spin_loop;

use std::sync::atomic::Ordering::{Acquire, Relaxed, Release};
use std::sync::atomic::{AtomicU32};
use std::thread::{current, yield_now, Thread};
use std::time::Duration;

use super::State;

#[derive(Default, Debug)]
pub struct SpinParker {
    state: AtomicU32
}

const PARKED: u32 = u32::MAX;
const EMPTY: u32 = 0;
const NOTIFIED: u32 = 1;
const PRENOTIFIED: u32 = 2;
const SPINLIMIT: u32 = 20;

impl Parker for SpinParker {
    fn wait(&self) {
        // change from EMPTY=>PARKED
        // it should not equal to PARKED as this is supposed to be called by one thread
        let backoff = Backoff::default();
        match self.state.compare_exchange(EMPTY, PARKED, Acquire, Relaxed) {
            Ok(_) | Err(PARKED) => loop {
                match self.state.load(Acquire) {
                    NOTIFIED => return,
                    PARKED => {
                        backoff.snooze();
                    }
                    PRENOTIFIED => {
                        self.wait_prenotified();
                    }
                    value => panic!("unexpected flag value {}", value),
                }
            },
            Err(NOTIFIED) => return,
            Err(PRENOTIFIED) => self.wait_prenotified(),
            Err(value) => panic!("unexpected flag value {}", value),
        };
    }

    fn wake(&self) {
        self.state.store(NOTIFIED, Release);
    }

    fn reset(&self) {
        self.state.store(EMPTY, Relaxed);
    }

    fn prewake(&self) {
        let _ = self
            .state
            .compare_exchange(EMPTY, PRENOTIFIED, Relaxed, Relaxed);
    }
```

```rust

    fn wait_timeout(&self, timeout: Duration) -> Result<(), ()> {
        let clock = Clock::new();
        let begin = clock.now();
        let backoff = Backoff::new();

        match self.state.compare_exchange(EMPTY, PARKED, Acquire, Relaxed) {
            Ok(_) | Err(PARKED) => loop {
                match self.state.load(Acquire) {
                    NOTIFIED => return Ok(()),
                    PRENOTIFIED => {
                        for _ in 0..SPINLIMIT {
                            if self.state.load(Acquire) == NOTIFIED {
                                return Ok(());
                            }
                            spin_loop();
                        }
                        if clock.now().duration_since(begin) >= timeout {
                            return Err(());
                        }
                        yield_now();
                    }
                    _ => {
                        if clock.now().duration_since(begin) >= timeout {
                            return Err(());
                        }
                        backoff.snooze();
                    }
                }
            },
            Err(NOTIFIED) => return Ok(()),
            Err(PRENOTIFIED) => {
                self.wait_prenotified();
                return Ok(());
            }
            Err(value) => panic!("unexpected flag value {}", value),
        }
    }

    fn state(&self) -> State {
        return match self.state.load(Acquire) {
            NOTIFIED => State::Notified,
            EMPTY => State::Empty,
            PRENOTIFIED => State::Prenotified,
            PARKED => State::Parked,
            value => panic!("unexpected flag value {}", value),
        };
    }

    fn name() -> &'static str {
        "Spin Parker"
    }
}

impl SpinParker {
    fn wait_prenotified(&self) {
        loop {
            let mut limit = SPINLIMIT - 1;
            while limit > 0 {
                if self.state.load(Acquire) == NOTIFIED {
                    return;
                }
```

```
                spin_loop();
                limit -= 1;
            }
            yield_now();
        }
    }
}
```

4.5 Block Parker Code

We take a Futex wrapper from Mara Bos to simplify our code [10]. The code is as follows:

TODO!