1. Prelimilary

I assume that we are familiar with the concept of asynchronous programming. Moreover, no part of the code should have specific thread affinity.

2. Abstract

I propose a method to utilize the mechanism behind async/await to create a user-level scheduler, taking into consideration lock and fairness aspects. By erasing thread identity, akin to a threadpool, we can run all critical sections in one thread (as in the concept of delegation-style lock), while running non-critical sections in other threads, without sacrificing the response time of the combiner.

3. Motivation

In recent weeks, I have been contemplating the type of system I could develop, with the ideas of usage-fair scheduling and delegation-style locking in mind. Given the current focus on fairness has gradually shifted to response time, I wondered if we could improve the combiner by allowing the waiters who are expected to wait longer to handle the combiner's non-critical sections, without sacrificing the benefits of combining.

That said, We aim to encapsulate both the critical and non-critical sections as "tasks." At first glance, this seems like a novel idea. Assuming the lock is "fair" or "usage-fair," a waiter will have a reasonable estimate of how long it might take for their critical section to be addressed. An example is the proportional backoff of a ticket lock.

However, quickly I realize like the disproportionate number of waiters compared to combiners interested in serving non-critical sections. As long as we have more than 4 waiters, where each one shares similar critical section size, both the 3rd and 4th might be able to serve the non-critical sections (assuming critical and non-critical size are similar) before their critical section being served.

Well, what if we can make the number of threads less than the "critical job", like a threadpool? In other words, if the threads are publishing job and their decendent jobs that depends on the results, then we are having a more freely job scheduling system.

I quickly recognized that this resembles a threadpool considering lock usage fairness, which may or may not be intriguing. Additionally, it could be challenging to use, potentially leading to a "callback hell"[1].

For example, I was thinking about how to turn the following code into something callable for both the critical part and non-critical part:

```
def foo():
    while True:
        if cond:
            with lock:
```

[1] I hear that when I learn about `promise` in *javascript*, but I learn *async/await* even earlier.

```
            # do something
    else:
        # do something else
```

It is certainly possible, but likely not as easy as wrapping a code block as a closure like what I have done with the critical sections earlier. Generally critical sections are easier, because the lock has to be released sometimes, and it is more like a continuous code block.

3.1. Resolution 1

Once I realize that this is not much different from a "callback hell" issue in asynchronous programming, a natural idea is to look how people solve it in most of the modern programming language. From my mere knowledge, coroutine is the name of the solution and there are two type of it:

1. stackful coroutine
2. stackless coroutine (*async/await*).

I am only familier with the latter, so I will only talk about *async/await*.

Essentially we want to have something callable for the non-critical section. This doesn't have to be a closure, but rather a state machine, as how *async/await* is generally implemented. Instead of passing a closure as a job, we pass something **resumable** such as a state machine. Thus, the job belongs to the combiner don't have to sacrifice the response time, while maintaining the benefits of combining. Essentially we are providing a "user-level" scheduler keeping in mind of the lock and fairness.

3.2. Resolution 2

Consider the scenario without a thread pool and we want to keep the standard locking api (I am not sure whether there're much reason when we don't like a threadpool for these kind of situation). The work of transparent delegation to provide delegation-style lock in standard locking API [1]. There they have domonstrated the potential of capturing the critical section by capturing value of stack register (**RSP**) and the program counter (**RIP**). By switching to the captured stack and program counter, we can execute the critical section in the context of the waiter.

That said, it is actually possible to have the context of non-critical section preserved once we released the lock. Then consider the situation where we don't have the threadpool, we can let the thread that anticipated to waited at the end to switch the non-critical section with the combiner. Once the combiner reaches its helper, we will have two choices:

1. The combiner can take over the critical section of the "volunteer" thread that helps him, and hand over the combining job to some next thread that we nominate to be the combiner.
2. The combiner can continue being the combiner, but some other thread (nominated in the same way as the previous volunteer thread) will take over the non-critical section job of the previous volunteer thread. Unless there're no more jobs, the combiner thread can always perform the

critical section (thus reducing the number of data move to the level of RCL). If there're no volunteer to take the non-critical section, the combiner can return back to its own job.

4. Potential Backfire

The idea of switching critical sections between threads may create some non-desired performance panelty. If the non-critical section strongly rely on the cache locality of its local variables, then switching the thread may be actually expensive. I am not very familier with the hardware architecture, so I am not able to tell whether this idea is generally better or not.

5. Implementation Difficulty (Async)

The whole idea is much more high level compared to any specific locks we built before. We are talking about scheduling of a threadpool like system, and thus the jobs must be distributed to the threadpool either sequencially or concurrently. Nonetheless, we still need to have the voting for combiner and scheudling around different locks.

`Rust` offers the nice *async/await* feature that we may want to utilize (that would be something we are not able to do with `C`). However, now since we are building the runtime rather than using them (I haven't used them as well), it would be much more complicated and much more things are needed to consider.

Some remarks for now:

1. Ordered Work Stealing for non-critical section
2. Combining for critical sections

6. Thoughts

Personally I've had the idea of using *async/await* to build a lock that enable faster switching for a long time. My initial motivation toward delegation-style locks comes from the UI-dispatcher of WPF that acts as a locks. Though, this type of usage of *async/await* is newly stemmed when I was thinking about solving the response time issue of the combiner. Even though in our earlier experiment the response time of the combiner is not worse too much, but my gut tells me that is likely due to the fact that the experiment is running in a short critical section, where the switching is expensive. The idea of using *async/await* is also inspired by the work of [1], where they has shown that what we really cares is the context, rather than the critical section itself. Thus *async/await* is a very nice fit (as a user-level thread) to modify the scheduling policy of the OS scheduler.

Further part of the idea stemed from Yuvraj's idea of switching the CPU when switching the owner of the lock. I propose that instead of switching the CPU, we can switch the thread, which might or might not be better? Nonetheless, I believe the spirit of delegation-style lock is that lock provides additional locality. Owning a locks mean we are sharing some resource, which might be expensive to move.

Bibliography

[1] V. Gupta, K. K. Dwivedi, Y. Kothari, Y. Pan, D. Zhou, and S. Kashyap, “Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with {TCLOCKS}”, 2023, pp. 1–16.