Making the Tokio scheduler 10x faster

tokio.rs/blog/2019-10-scheduler

We've been hard at work on the next major revision of Tokio, Rust's asynchronous runtime. Today, a complete rewrite of the scheduler has been submitted as a pull request. The result is huge performance and latency improvements. Some benchmarks saw a 10x speed up! It is always unclear how much these kinds of improvements impact "full stack" use cases, so we've also tested how these scheduler improvements impacted use cases like Hyper and Tonic (spoiler: it's really good).

In preparation for working on the new scheduler, I spent time searching for resources on scheduler implementations. Besides existing implementations, I did not find much. I also found the source of existing implementations difficult to navigate. To remedy this, I tried to keep Tokio's new scheduler implementation as clean as possible. I also am writing this detailed article on implementing the scheduler in hope that others in similar positions find it useful.

The article starts with a high level overview of scheduler design, including work-stealing schedulers. It then gets into the details of specific optimizations made in the new Tokio scheduler.

The optimizations covered are:

The major theme is "reduce." After all, there is no code faster than no code!

The article also covers testing the new scheduler. Writing correct, concurrent, lock-free code is really hard. It is better to be slow and correct than fast and buggy, especially if those bugs relate to memory safety. The best option, however, is to be fast *and* correct, so we wrote `loom`, a tool for testing concurrency.

Before jumping in, I want to extend some gratitude.

- **@withoutboats** and others who worked on Rust's `async / await` feature. You did a great job. It is a killer feature.
- **@cramertj** and others who designed `std::task`. It is a huge improvement compared to what we had before. Also, a great job there.
- **Buoyant**, the makers of Linkerd, and more importantly my employer. Thanks for letting me spend so much time on this work. Readers who are in need of a service mesh, check out Linkerd. It will soon include all the goodness discussed in this article.
- **Go** for having such a good scheduler implementation.

Grab a cup of coffee and get yourselves comfortable. This is going to be a long article.

Schedulers, how do they work?

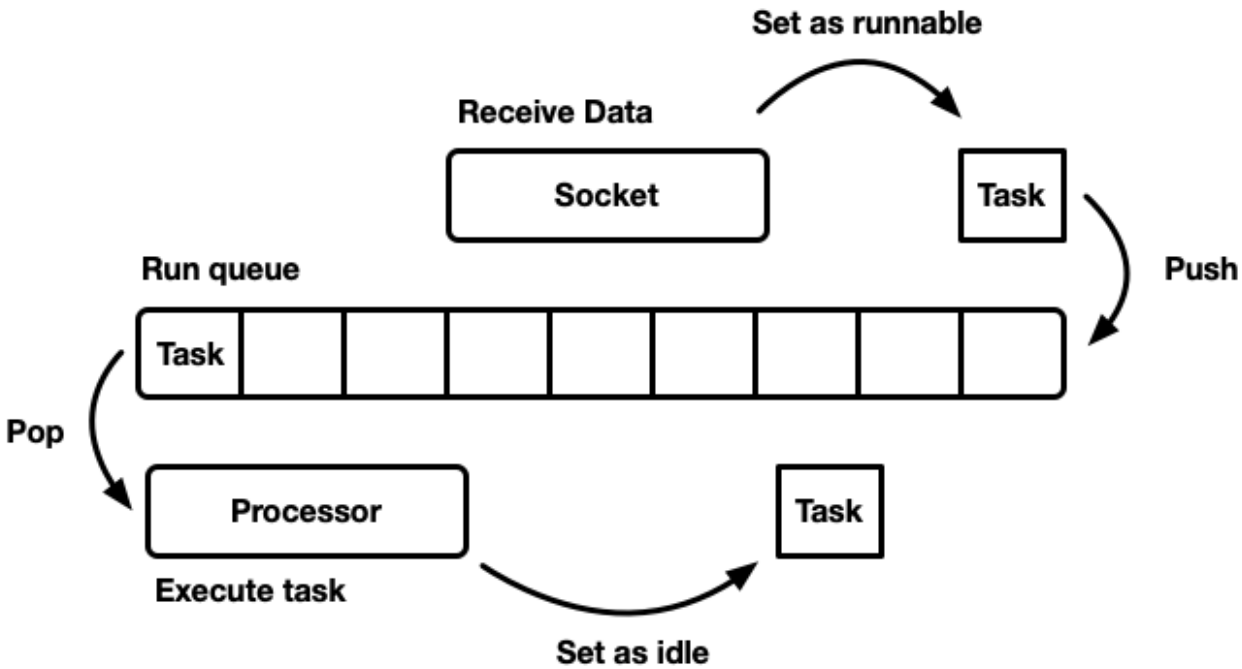
The role of a scheduler is to schedule work. An application is broken up into units of work, which we will call *tasks*. A task is *runnable* when it can make progress, and is no longer runnable (or *idle*) when it is blocked on an external resource. Tasks are independent in that any number of runnable tasks can execute concurrently. The scheduler is responsible for executing tasks in the running state until they transition back to idle. Executing a task implies assigning CPU time -- a global resource -- to the task.

The article discusses user-space schedulers, i.e., schedulers that run on top of operating system threads (which, in turn, are powered by a kernel land scheduler). The Tokio scheduler executes Rust futures, which can be though of as "asynchronous green threads". This is the M:N threading pattern where many user land tasks are multiplexed on a few operating system threads.

There are many different ways of modeling a scheduler, each with pros and cons. At the most basic level, the scheduler can be modeled as a *run queue* and a *processor* that drains the queue. A processor is a bit of code that runs on a thread. In pseudocode, it does:

```
while let Some(task) = self.queue.pop() {
    task.run();
}
```

When a task becomes runnable, it is inserted into the run queue.



While it is possible to design a system where resources, tasks, and the processor all exist on a single thread, Tokio chooses to use multiple threads. We live in a world where computers come with many CPUs. Designing a single-threaded scheduler would result in hardware underutilization. We want to use all of our CPUs. There are a few ways to do this:

- One global run queue, many processors.

- Many processors, each with their own run queue.

One queue, many processors

In this model, there is a single, global, run queue. When tasks become runnable, they are pushed onto the tail of the queue. There are multiple processors, each running on a separate thread. Each processor pops from the head of the queue, blocking the thread if no task is available.

The run queue must support both multiple producers and multiple consumers. The commonly used algorithm is an intrusive linked list. Intrusive implies that the task structure includes a pointer to the next task in the run queue instead of wrapping the task with a linked list node. This way, allocations are avoided for push and pop operations. It is possible to use a lock-free push operation but popping requires^1 a mutex to coordinate consumers.

^1: It is technically possible to implement a lock-free multi-consumer queue. However, in practice the overhead needed to correctly avoid locks is greater than just using a mutex.

This approach is commonly used when implementing a general-purpose thread pool, as it has several advantages:

- Tasks are scheduled fairly.
- The implementation is relatively simple. An off-the-shelf queue can be paired with the processor loop sketched above.

> A quick note on fairness. Scheduling fairness means that tasks are scheduled first-in, first-out. Tasks that are transitioned to running first are executed first. General purpose schedulers try to be fair, but there are use cases, like parallelizing a computation using fork-join, where the only important factor is how fast the end result is computed and not the fairness of each individual sub task.

This scheduler model has a downside. All processors contend on the head of the queue. For general-purpose thread pools, this usually is not a deal breaker. The amount of time processors spend executing the task far outweighs the amount of time spent popping the task from the run queue. When tasks execute for a long period of time, queue contention is reduced. However, Rust's asynchronous tasks are expected to take very little time executing when popped from the run queue. In this scenario, the overhead from contending on the queue becomes significant.

Concurrency and mechanical sympathy.

To get the best runtime performance out of a program, we must design it to take advantage of how the hardware operates. The term "mechanical sympathy" was applied to software by Martin Thompson (whose blog, while no longer updated, still contains much relevant knowledge).

A detailed discussion of how modern hardware handles concurrency is out of scope for this article. At a 10,000 foot view, hardware is gaining more performance not by going faster, but by making more CPU cores available to an application (my laptop has 6!). Each core can perform large amounts of computation in tiny periods of time. Relatively speaking, actions like cache and memory accesses takes much longer. Therefore, for applications to be fast, we must maximize the amount of CPU instructions per memory access. While the compiler can do a lot of work for us, as developers, we do need to think about things like struct layout and memory access patterns.

When it comes to concurrent threads, behavior is pretty similar to a single thread **until** concurrent mutations happen to the same cache line or sequential consistency is requested. Then, the CPU's cache coherence protocol will have to start working to ensure that each CPU's cache stays up to date.

This is all to say the obvious: avoid cross thread synchronization as much as possible because it is slow.

Many processors, each with their own run queue

Another way to model a scheduler is to use multiple single-threaded schedulers. Each processor gets its own run queue and tasks are pinned to a specific processor. This avoids the problem of synchronization entirely. As Rust's task model requires the ability to queue a task from any thread, there still needs to be a thread-safe way to inject tasks into the scheduler. Either each processor's run queue supports a thread-safe push operation (MPSC) or each processor has **two** run queues: an unsynchronized queue and a thread-safe queue.

This is the strategy used by Seastar. Because synchronization is almost entirely avoided, this strategy can be very fast. However, it's not a silver bullet. Unless the workload is entirely uniform, some processors will become idle while other processors are under load, resulting in resource underutilization. This happens because tasks are pinned to a specific processor. When a bunch of tasks on a single processor are scheduled in a batch, that single processor is responsible for working through the spike even if other procesors are idle.

Most "real world" workloads are not uniform. Because of this, general purpose schedulers tend to avoid this model.

Work-stealing scheduler

The work-stealing scheduler builds upon the sharded scheduler model and addresses the underutilization problem. Each processor maintains its own run queue. Tasks that become runnable are pushed onto the current processor's run queue and processors drain their local run queue. However, when a processor becomes idle, it checks sibling processor run queues and attempts to steal from them. A processor will go to sleep only once it fails to find work from sibling run queues.

At the model level, this is a "best of both worlds" approach. Under load, processors operate independently, avoiding synchronization overhead. In cases where the load is not evenly distributed across processors, the scheduler is able to redistribute. Because of this characteristic, the work-stealing scheduler is the choice of Go, Erlang, Java, and others.

The downside is this approach is far more complicated; the run queue algorithm must support the stealing operation and **some** cross processor synchronization is required to keep things running smoothly. If not done correctly, the overhead to implement the work-stealing model can be greater than the benefits gained.

Consider this case: Processor A is currently running a task and has an empty run queue. Processor B is idle; it tries to steal tasks and fails, so it goes to sleep. Then, the task being executed by processor A spawns 20 tasks. The goal would be for processor B to wake up and steal some of those newly spawned tasks. To achieve this, the work-stealing scheduler requires some heuristic where processors signal sleeping sibling processors when they observe new work in their queue. Of course, this introduces additional synchronization, so this operation must be minimized.

In summary:

- Minimizing synchronization is good.
- Work-stealing is the algorithm of choice for general purpose schedulers.
- Each processor is mostly independent, but some synchronization is required for stealing.

The Tokio 0.1 scheduler

Tokio first shipped its work-stealing scheduler in March 2018. This was a first attempt based on some assumptions that turned out to be incorrect.

First, the Tokio 0.1 scheduler assumed that processor threads should be shut down if idle for a certain period of time. The scheduler was originally intended to be a "general purpose" thread-pool executor for Rust futures. When the scheduler was first written, Tokio was still in the "tokio-core" days. At this point, the model was that I/O-based tasks would be executed on a single thread colocated with the I/O selector (epoll, kqueue, iocp, ...). More CPU bound work could be punted to a thread-pool. In this context, the number of active threads should be flexible and shutting down idle threads makes more sense. However, the model shifted to running *all* async tasks on the work-stealing scheduler, in which case it makes more sense to keep a small number of threads always active.

Second, it used the crossbeam deque implementation. This implementation is based on the Chase-Lev deque, which, for reasons described below, is not a good fit for the use case of scheduling independent asynchronous tasks.

Third, the implementation was over-complicated. This is due, in part, to this being my first scheduler implementation. Also, I was over eager in using atomics in code paths where a mutex would have done just fine. An important lesson learned is that there are many cases where mutexes are the best option.

Finally, the original implementation contained many small inefficiencies. The details of Rust's asynchronous model evolved significantly in the early years, yet libraries maintained API stability over those years. This resulted in some debt being accumulated that can now be payed back.

With tokio approaching its first major breaking release, we can pay all of that debt with the lessons learned over those years. It's an exciting time!

The next generation Tokio scheduler

Now it is time to dig into what changed in the new scheduler.

The new task system

First, its important to highlight something **not** part of Tokio, but is critical to some of the achieved gains: the new task system included in `std`, designed primarily by Taylor Cramer. This system provides the hooks that a scheduler must implement to execute Rust asynchronous tasks, and it is really well done. It is both much lighter and more flexible then the previous iteration.

The `Waker` struct is held by resources and is used to signal that a task is *runnable* and be pushed into the scheduler's run queue. In the new task system, the `Waker` struct is two pointers wide, when it used to be much bigger. Shrinking the size is important to minimize the overhead of copying the `Waker` value around, as well as taking less space in structs, allowing for more critical data to fit in the cache line. The custom vtable design enables a number of optimizations which will be discussed later.

A better run queue

The run queue is at the heart of the scheduler. As such, it is probably the most critical component to get right. The original Tokio scheduler used crossbeam's deque implementation, which is single-producer, multi-consumer deque. Tasks are pushed onto one end and values are popped from the other. Most of the time, the thread that pushes the value will pop it, however, other threads will occasionally "steal" by popping the value themselves. The deque is backed by an array and a set of indices tracking the head and tail. When the deque is full, pushing onto it will result in growing the storage. A new, larger, array is allocated and values are moved into the new store.

The ability for the deque to grow comes at a complexity and overhead cost. The push / pop operations must factor in this growth. Additionally, on growth, freeing the original array comes with additional difficulties. In a garbage collected language, the old array would go out of scope, and eventually the GC would free it. However, Rust does not come with a GC. This means we are responsible for freeing the array, but threads may be in the process of accessing the memory concurrently. Crossbeam's answer to this is to use an epoch based reclamation strategy. While not **terribly** expensive, it does add non-trivial overhead in the hot path. Every operation must now issue atomic RMW (read-modify-write) operations when entering and exiting the critical sections to signal to other threads that the memory is in use and to avoid freeing.

Because of the cost associated with growing the run queue, it's worth investigating if growing the queue is even a requirement. This question is what ended up spurring the rewrite of the scheduler. The new scheduler's strategy is to use a fixed size per-process queue. When the queue is full, instead of growing the local queue, the task is pushed into a global, multi-consumer, multi-producer, queue. Processors will need to occasionally check this global queue, but at a much less frequent rate than the local queue.

An early experiment replaced the crossbeam queue with a bounded mpmc queue. This did not result in much improvement due to the amount of synchronization performed by both push and pop. A key thing to remember about the work-stealing use case is that, under load, there is almost no contention on the queues since each processor only accesses its own queue.

At this point, I opted to read the Go source closer, and discovered they used a fixed size single-producer, multi-consumer queue. The impressive thing about this queue is how little synchronization is needed for it to function. I ended up adapting the algorithm for

use in the Tokio scheduler, making a few changes. Notably, the Go implementation uses sequential consistency for its atomic operations (based on my limited knowledge of Go). The version implemented as part of the Tokio scheduler also reduces some copying in less frequent code paths.

The queue implementation is a circular buffer, using an array to store values. Atomic integers are used to track the head and tail positions.

```
struct Queue {
    /// Concurrently updated by many threads.
    head: AtomicU32,

    /// Only updated by producer thread but read by many threads.
    tail: AtomicU32,

    /// Masks the head / tail position value to obtain the index in the buffer.
    mask: usize,

    /// Stores the tasks.
    buffer: Box<[MaybeUninit<Task>]>,
}
```

Pushing into the queue is done by a single thread:

```
loop {
    let head = self.head.load(Acquire);

    // safety: this is the **only** thread that updates this cell.
    let tail = self.tail.unsync_load();

    if tail.wrapping_sub(head) < self.buffer.len() as u32 {
        // Map the position to a slot index.
        let idx = tail as usize & self.mask;

        // Don't drop the previous value in `buffer[idx]` because
        // it is uninitialized memory.
        self.buffer[idx].as_mut_ptr().write(task);

        // Make the task available
        self.tail.store(tail.wrapping_add(1), Release);

        return;
    }

    // The local buffer is full. Push a batch of work to the global
    // queue.
    match self.push_overflow(task, head, tail, global) {
        Ok(_) => return,
        // Lost the race, try again
        Err(v) => task = v,
    }
}
```

Note that, in this `push` function, the only atomic operations are a load with `Acquire` ordering and a store with `Release` ordering. There are no read-modify-write operations (`compare_and_swap`, `fetch_and`, ...) or sequential consistency. This is important because, on x86 chips, all loads / stores are already "atomic". So, at the CPU level, this function has no synchronization. Using atomic operations will impact the compiler in that it prevents certain optimization, but that's it. The first `load` can most likely be done safely with `Relaxed` ordering, but there were no measurable gains in switching.

When the queue is full, `push_overflow` is called. This function moves half of the tasks in the local queue into the global queue. The global queue is an intrusive linked list guarded by a mutex. The tasks being moved to the global queue are first linked together, then the mutex is acquired and all the tasks are inserted by updating the global queue's tail pointer. This keeps the critical section small.

If you are familiar with the details of atomic memory orderings, you might notice a potential "issue" with the `push` function as shown above. An atomic `load` with `Acquire` ordering is pretty weak. It may return stale values, i.e. a concurrent steal operation may have already incremented the value of `self.head` but the thread performing the `push` had an old value in the cache so it didn't notice the steal operation. This is not a problem for the correctness of the algorithm. In the fast-path of `push`, we only care if the local run queue is full or not. Given that the current thread is the only thread that can push into the run queue, a stale `load` will result in seeing the run queue as more full than it actually is. It may incorrectly determine that the queue is full and enter the `push_overflow` function, but this function includes a stronger atomic operation. If `push_overflow` determines the queue is not actually full, it returns w/ `Err` and the `push` operation tries again. This is another reason why `push_overflow` will move half of the run queue to the global queue. By moving half of the queue, the "run queue is empty" false positive is hit a lot less often.

The local `pop` (from the processor that owns the queue) is also light:

```
loop {
    let head = self.head.load(Acquire);

    // safety: this is the **only** thread that updates this cell.
    let tail = self.tail.unsync_load();

    if head == tail {
        // queue is empty
        return None;
    }

    // Map the head position to a slot index.
    let idx = head as usize & self.mask;

    let task = self.buffer[idx].as_ptr().read();

    // Attempt to claim the task read above.
    let actual = self
        .head
        .compare_and_swap(head, head.wrapping_add(1), Release);

    if actual == head {
        return Some(task.assume_init());
    }
}
```

In this function, there is a single atomic load and one `compare_and_swap` with release. The primary overhead comes from the `compare_and_swap`.

The `steal` function is similar to the `pop` function but the load from `self.tail` must be atomic. Also, similar to `push_overflow`, the steal operation will attempt to claim half of the queue instead of a single task. This has some nice characteristics which we will cover later.

The last missing piece is consuming the global queue. This queue is used to handle overflow from processor local queues as well as to submit tasks to the scheduler from non-processor threads. If a processor is under load, i.e. the local run queue has tasks. The processor will attempt to pop from the global after executing ~60 tasks from the local queue. It also checks the global queue when in the "searching" state, described below.

Optimizing for message passing patterns

Applications written with Tokio are usually modeled with many small, independent tasks. These tasks will communicate with each other using message passing. This pattern is similar to other languages like Go and Erlang. Given how common this pattern is, it makes sense for the scheduler to try to optimize for it.

Given task A and task B. Task A is currently executing and sends a message to task B via a channel. The channel is the resource that task B is currently blocked on, so the action of sending the message will result in Task B transitioning to the runnable state and be

pushed into the current processor's run queue. The processor will then pop the next task off of the run queue, execute it, and repeat until it reaches task B.

The problem is that there can be significant latency between the time the message is sent and task B gets executed. Further, "hot" data, such as the message, is stored in the CPUs cache when it is sent but by the time task B gets scheduled, there is a high probability that the relevant caches have gotten purged.

To address this, the new Tokio scheduler implements an optimization (also found in Go's and Kotlin's schedulers). When a task transitions to the runnable state, instead of pushing it to the back of the run queue, it is stored in a special "next task" slot. The processor will always check this slot before checking the run queue. When inserting a task into this slot, if a task is already stored in it, the old task is removed from the slot and pushed to the back of the run queue. In the message passing case, this will result in the receiver of the message to be scheduled to run next.

Throttle stealing

In the work-stealing scheduler, when a processor's run queue is empty, the processor will then attempt to steal tasks from sibling processors. To do this, a random sibling is picked as a starting point and the processor performs a steal operation on that sibling. If no tasks are found, the next sibling is tried, and so on until tasks are found.

In practice, it is common for many processors to finish processing their run queue around the same time. This happens when a batch of work arrives (for example when `epoll` is polled for socket readiness). Processors are woken up, they acquire tasks, run them, and are finished. This results in all processors attempting to steal at the same time, which means many threads attempting to access the same queues. This creates contention. Randomly picking the starting point helps reduce contention, but it can still be pretty bad.

To work around this, the new scheduler limits the number of concurrent processors performing steal operations. We call the processor state in which processors attempt to steal as "searching for work", or the "searching" state for short (this will come up later). This optimization is performed by having an atomic int that the processor increments before starting to search and decrements when exiting the searching state. The max number of searchers is half the total number of processors. That said, the limit is sloppy, and that is OK. We don't need a hard limit on the number of searchers, just a throttle. We trade precision for algorithm efficiency.

Once in the searching state, the processor attempts to steal from sibling workers and checks the global queue.

Reducing cross thread synchronization

The other critical piece of a work-stealing scheduler is sibling notification. This is where a processor notifies a sibling when it observes new tasks. If the sibling is sleeping, it wakes up and steals tasks. The notification action has another critical responsibility. Recall the queue algorithm used weak atomic orderings (Acquire / Release). Because of how atomic memory ordering work, without additional synchronization, there is no guarantee that a sibling processor will ever see tasks in the queue to steal. The notification action also is responsible for establishing the necessary synchronization for the sibling processor to see the tasks in order to steal them. These requirements make the notification action expensive. The goal is to perform the notification action as little as possible without resulting in CPU under utilization, i.e. a processor has tasks and a sibling is unable to steal them. Overeager notification results in a thundering herd problem.

The original Tokio scheduler took a naive approach to notification. Whenever a new task was pushed on to the run queue, a processor was notified. Whenever a processor was notified and found a task upon waking, it would then notify yet another processor. This logic very quickly resulted in all processor getting woken up and searching for work (causing contention). Very often, most of these processors failed to find work and went back to sleep.

The new scheduler significantly improves on this by borrowing the same technique used in the Go scheduler. Notification is attempted at the same points as the previous scheduler, however, notification only happens if there are no workers in the searching state (see previous section). When a worker is notified, it is immediately transitioned to the searching state. When a processor in the searching state finds new tasks, it will first transition out of the searching state, then notify another processor.

This logic has the effect of throttling the rate at which processors wake up. If a batch of tasks is scheduled at once (for example, when `epoll` is polled for socket readiness), the first one will result in notifying a processor. That processor is now in the searching state. The rest of the scheduled tasks in the batch will not notify a processor as there is at least one in the searching state. That notified processor will steal half the tasks in the batch, and in turn notify another processor. This third processor will wake up, find tasks from one of the first two processors and steal half of those. This results in a smooth ramp up of processors as well as rapid load balancing of tasks.

Reducing allocations

The new Tokio scheduler requires only a single allocation per spawned task while the old one required two. Previously, the Task struct looked something like:

```
struct Task {
    /// All state needed to manage the task
    state: TaskState,

    /// The logic to run is represented as a future trait object.
    future: Box<dyn Future<Output = ()>>,
}
```

The `Task` struct would then be allocated in a `Box` as well. This has always been a wart that I have wanted to fix for a long time (I first attempted to fix this in 2014). Since the old Tokio scheduler, two things have changed. First, `std::alloc` stabilized. Second, the Future task system switched to an explicit vtable strategy. These were the two missing pieces needed to finally get rid of the double allocation per task inefficiency.

Now, the `Task` structure is represented as:

```
struct Task<T> {
    header: Header,
    future: T,
    trailer: Trailer,
}
```

Both `Header` and `Trailer` are state needed to power the task, however they are split between "hot" data (header) and "cold" data (trailer), i.e. roughly data that is accessed often and data that is rarely used. The hot data is placed at the head of the struct and is kept as small as possible. When the CPU dereferences the task pointer, it will load a cache line sized amount of data at once (between 64 and 128 bytes). We want that data to be as relevant as possible.

Reducing atomic reference counting

The final optimization we will discuss in this article is how the new scheduler reduces the amount of atomic reference counts needed. There are many outstanding references to the task structure: the scheduler and each waker hold a handle. A common way to manage this memory is to use atomic reference counting. This strategy requires an atomic operation each time a reference is cloned and an atomic operation each time a reference is dropped. When the final reference goes out of scope, the memory is freed.

In the old Tokio scheduler, each waker held a counted reference to the task handle, roughly:

```
struct Waker {
    task: Arc<Task>,
}

impl Waker {
    fn wake(&self) {
        let task = self.task.clone();
        task.scheduler.schedule(task);
    }
}
```

When the task is woken, the reference is cloned (atomic increment). The reference is then pushed into the run queue. When the processor receives the task and is done executing it, it drops the reference resulting in an atomic decrement. These atomic operations add up.

This problem has previously been identified by the designers of the `std::future` task system. It was observed that when Waker::wake is called, often times the original waker reference is no longer needed. This allows for reusing the atomic reference count when pushing the task into the run queue. The `std::future` task system now includes two "wake" APIs:

- `wake` which takes `self`
- `wake_by_ref` which takes `&self` .

This API design pushes the caller to use `wake` which avoids the atomic increment. The implementation now becomes:

```
impl Waker {
    fn wake(self) {
        task.scheduler.schedule(self.task);
    }

    fn wake_by_ref(&self) {
        let task = self.task.clone();
        task.scheduler.schedule(task);
    }
}
```

This avoids the overhead of additional reference counting **only if** it is possible to take ownership of the waker in order to wake. In my experience, it is almost always desirable to wake with `&self` instead. Waking with `self` prevents reusing the waker (useful in cases where the resource sends many values, i.e. channels, sockets, ...) it also is more difficult to implement thread-safe waking when `self` is required (the details of this will be left to another article).

The new scheduler side steps the entire "wake by `self` " issue by avoiding the atomic increment in `wake_by_ref` , making it as efficient as `wake(self)` . This is done by having the scheduler maintain a list of all tasks currently active (have not yet completed). This list represents the reference count needed to push a task into the run queue.

The difficulty with this optimization is to ensure that the scheduler will not drop any tasks from its list until it can be guaranteed that the task will not be pushed into the run queue again. The specifics of how this is managed are beyond the scope of this article, but I urge you to further investigate this in the source.

Fearless (unsafe) concurrency with Loom

Writing correct, concurrent, lock-free code is really hard. It is better to be slow and correct than fast and buggy, especially if those bugs relate to memory safety. The best option is to be fast and correct. The new scheduler makes some pretty aggressive optimizations and avoids most `std` types in order to write more specialized versions. There is quite a bit of `unsafe` code in the new scheduler.

There are a few ways to test concurrent code. One is to let your users do the testing and debugging for you (an attractive option, to be sure). Another is to write unit tests that run in a loop and hope it catches a bug. Maybe even throw in TSAN. Of course, if this does catch a bug, there is no way to easily reproduce it short of running the test in a loop again. Also, how long do you run that loop for? Ten seconds? Ten minutes? Ten days? This used to be the state of testing concurrent code with Rust.

We did not find the status quo acceptable. We want to feel confident (well, as confident as we can be) when we ship code -- especially concurrent, lock-free code. Reliability is something that Tokio users have come to expect.

To address our need, we wrote a new tool: Loom. Loom is a permutation testing tool for concurrent code. Tests are written as normal, but when they are executed with `loom`, `loom` will run the test many times, permuting all possible executions and behaviors that test is able to encounter in a threaded environment. It also validates correct memory access, freeing memory, etc...

As an example, here is a loom test for the new scheduler:

```
#[test]
fn multi_spawn() {
    loom::model(|| {
        let pool = ThreadPool::new();

        let c1 = Arc::new(AtomicUsize::new(0));

        let (tx, rx) = oneshot::channel();
        let tx1 = Arc::new(Mutex::new(Some(tx)));

        // Spawn a task
        let c2 = c1.clone();
        let tx2 = tx1.clone();
        pool.spawn(async move {
            spawn(async move {
                if 1 == c1.fetch_add(1, Relaxed) {
                    tx1.lock().unwrap().take().unwrap().send(());
                }
            });
        });

        // Spawn a second task
        pool.spawn(async move {
            spawn(async move {
                if 1 == c2.fetch_add(1, Relaxed) {
                    tx2.lock().unwrap().take().unwrap().send(());
                }
            });
        });

        rx.recv();
    });
}
```

It looks normal enough, but again, the bit of code within the `loom::model` block gets run many thousands of times (probably in the millions), each time with a slight variation in behavior. The exact ordering of threads changes each time. Also, for every atomic operation, `loom` will try all different behaviors permitted under the C++11 memory model. Recall how earlier, I mentioned that an atomic load with `Acquire` was fairly weak and could return stale values. A `loom` test will try every single possible value that can be loaded.

`loom` has been an invaluable tool to use while working on the new scheduler. It caught more than 10 bugs that were missed by the other unit tests, hand testing, and stress testing.

The astute reader might have questioned the claim that loom tests "all possible permutations", and would be right to do so. Naive permutation of behavior would result in combinatorial explosion at the factorial level. Any non trivial test would never complete. This problem has been researched for years and a number of algorithms exist to manage the combinatorial explosion. Loom's core algorithm is based on dynamic

partial order reduction. This algorithm is able to prune out permutations that result in identical executions. Even with this, it is possible for the state space to grow too large to complete in a reasonable time (few minutes). Loom also allows bounding the search space using a bounded variant of dynamic partial order reduction.

All in all, I am **much** more confident in the scheduler's correctness thanks to extensive testing with Loom.

The results

So, now that we are done going over what schedulers are and how the new Tokio scheduler achieved huge performance gains... what exactly were those gains? Given that the new scheduler is very new, there has not been extensive real-world testing yet. Here is what we do know.

First, the new scheduler is **much** faster in micro benchmarks. Here is how the new scheduler improved on some benchmarks.

Old scheduler

```
test chained_spawn ... bench:   2,019,796 ns/iter (+/- 302,168)
test ping_pong     ... bench:   1,279,948 ns/iter (+/- 154,365)
test spawn_many    ... bench:  10,283,608 ns/iter (+/- 1,284,275)
test yield_many    ... bench:  21,450,748 ns/iter (+/- 1,201,337)
```

New scheduler

```
test chained_spawn ... bench:     168,854 ns/iter (+/- 8,339)
test ping_pong     ... bench:     562,659 ns/iter (+/- 34,410)
test spawn_many    ... bench:   7,320,737 ns/iter (+/- 264,620)
test yield_many    ... bench:  14,638,563 ns/iter (+/- 1,573,678)
```

The benchmarks include:

- `chained_spawn` tests recursively spawning a new task, i.e. spawn a task, that spawns a task, that spawns a task, ...
- `ping_pong` allocates a `oneshot` channel, spawns a task that sends a message on that channel. The original task waits to receive the message. This is the closest "real world" benchmark.
- `spawn_many` tests injecting tasks into the scheduler, i.e. spawning tasks from outside of the scheduler context.
- `yield_many` tests a task waking itself.

The improvements from the old scheduler to the new scheduler are very impressive. However, how does that carry over to the "real world". It's hard to say exactly, but I did try running Hyper benchmarks using the new scheduler.

This is the "hello world" Hyper server being benchmarked using `wrk -t1 -c50 -d10` :

Old scheduler

```
Running 10s test @ http://127.0.0.1:3000
  1 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   371.53us   99.05us   1.97ms   60.53%
    Req/Sec   114.61k     8.45k  133.85k    67.00%
  1139307 requests in 10.00s, 95.61MB read
Requests/sec: 113923.19
Transfer/sec:      9.56MB
```

New scheduler

```
Running 10s test @ http://127.0.0.1:3000
  1 threads and 50 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   275.05us   69.81us   1.09ms   73.57%
    Req/Sec   153.17k    10.68k  171.51k    71.00%
  1522671 requests in 10.00s, 127.79MB read
Requests/sec: 152258.70
Transfer/sec:     12.78MB
```

That's an increase of 34% requests per second just from switching schedulers! When I first saw this, I was more than happy. I had originally expected an increase of ~5~10%. Then I was sad, because it also meant the old Tokio scheduler is not that good, but oh well. Then I remembered that Hyper already tops the TechEmpower benchmarks. I'm excited to see how the new scheduler will impact those rankings.

Tonic, a gRPC client & server, saw about a 10% speed up, which is pretty impressive given that Tonic is not yet highly optimized.

Conclusion

I'm really excited to finally get this work out. It has been in progress for months now and is going to be a big step forward in Rust's asynchronous I/O story. I'm very satisfied with the improvements the new scheduler work has shown already. There also are plenty of areas in the Tokio code base that could be sped up, so this new scheduler work is not the end on the performance front.

I also hope the amount of detail this article provides is useful for others who may attempt to write a scheduler.

—Carl Lerche