Published 1 Mar 2025

Last updated 13 Jun 2025

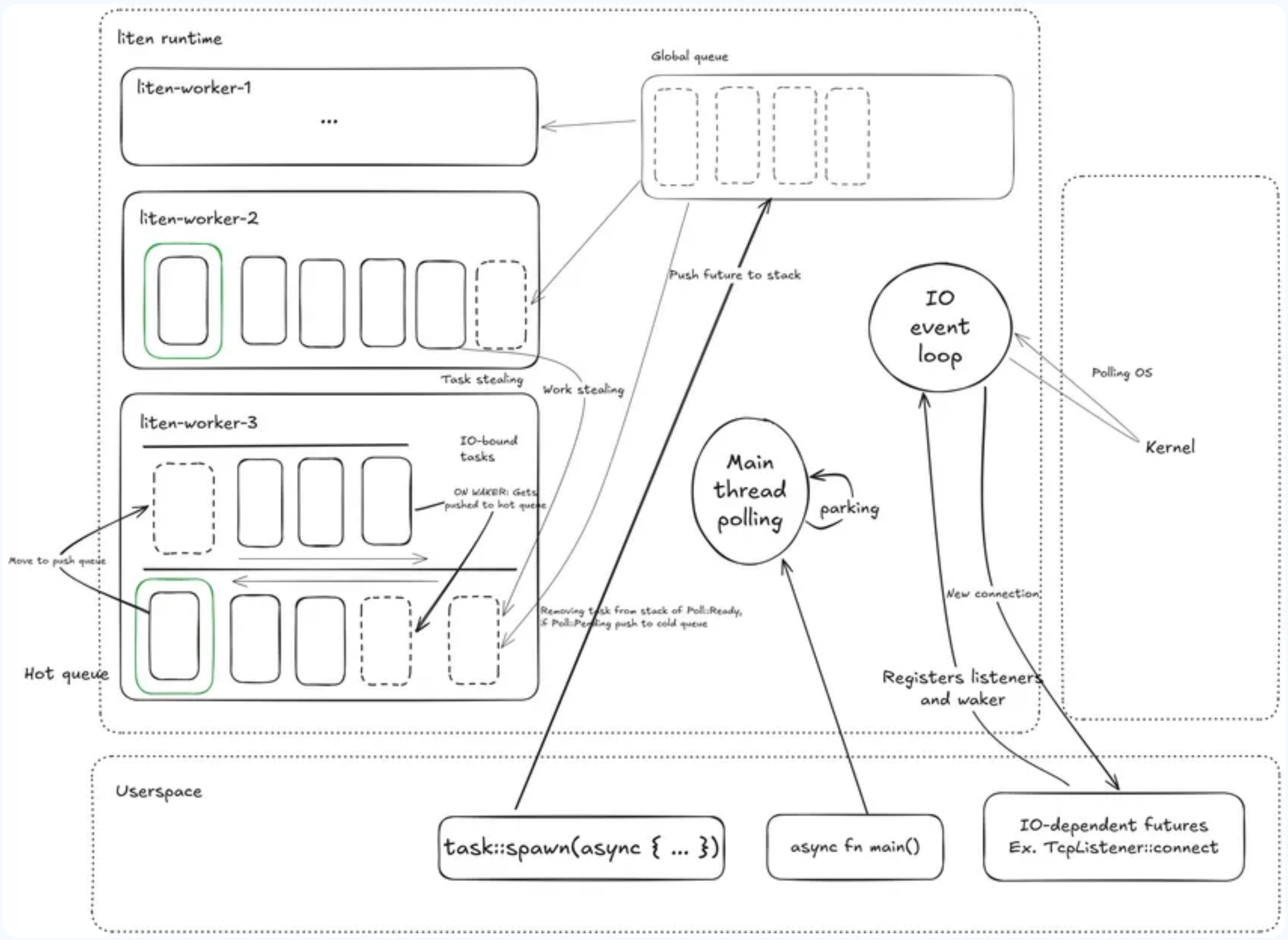


Diagram of the 'liten' rust async runtime.

Making a multithreaded rust async runtime

Many people don't know how async runtimes (for example tokio) works. The goal of this article is explaining how one works and what goes into making one efficient.

Background

Rust is very interesting to me. It allows for more control of the controlflow than any other language. It does this in many different ways. One of these ways is with the `Option<T>` and `Result<T, E>` **tagged unions**. `Option` prevents **the billion dollar mistake** and `Result` embraces the error-as-values instead of using Exceptions (thank god).

Rust takes the same idea and applies it at a much more complex area, it's asyncronous functionality, and does so very well. Rust has async syntax builtin, but not an async runtime builtin. What I'm refering to, is that a function can be colored `async`, but the `fn main()` *cannot* run an `async` function. This means that it's impossible (without correct implementation) to run a async function in the main function. This is where async runtimes come into play. If an application wishes to run Futures (async functions), one needs to implement a way to execute `Futures`.

What is a Future??

A future is just a trait, like any other, with syntax support from the rust compiler:

```
// Simplified version of std::future::Future trait (as of 1.83)
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

A `Future` can be thought as a lazy-evaluated javascript promise. A `Future` doesn't do anything unless `Future::poll`'ed. The future poll method returns either a `Poll::Ready(/* value */)` or `Poll::Pending`.

When a future returns a `Poll::Pending`, it tells the runtime that the future cannot make any more progress (in that moment). When the future returns Pending it's the implementor's responsibility to store the Waker contained in the Context struct and to call **Waker::wake()** when progress can be made. This tells the async runtime that this `Future` might be able to make progress by polling it again. Eventually the `Future` returns `Pending::Ready(/*value*/)`, where the async syntax automatically unwraps the value.

The point of this mental model is only allocating CPU-time to items that actually need it. This type of thinking isn't about performance. Asynchronous has never been about performance. It's about waiting as little as possible and only allocating CPU to tasks that has utility for CPU.

The only job for a async runtime is calling the poll method on `Future`'s as efficiently as possible.

Implementing an async runtime

There are essentially no limits on how the implementation looks like for an async runtime. As we know the only job an async runtime does is polling `Futures`. It is a very simple problem, with a potentially quite hard solution.

The point of async rust is *efficiency*. For example, if an application is listening on TCP, there is really no reason to block until a request comes. We only need to process the TCP connection, when one has arrived. In `std`, the TCP listener blocks until a request has been made, but with async we can make the TcpListener not do any work until it needs to, leaving precious CPU time to other async tasks/futures.

We can expand async rusts capabilities by adding green threads. These are threads in which it's lifecycle is not managed by the OS (brings overhead, such as startup times and more), but by the async runtime, making them very light and cheap. A prime example of this is `tokio::task`.

Requirements

Let's list some requirements before touching code or discussing any implementations:

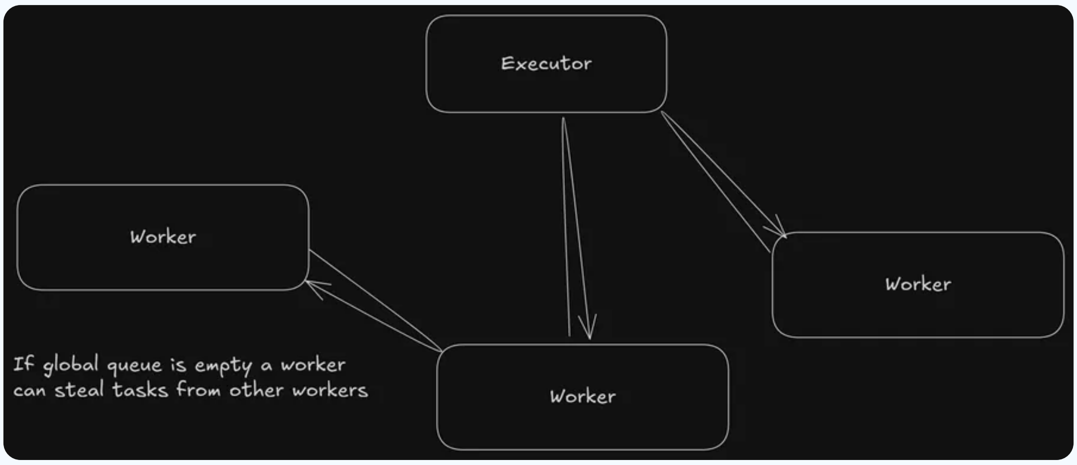
- Multithreaded
- Green-thread support
- As efficient as possible

Async is all about concurrency. The easiest way of implementing this is green threads, which can be implemented in various ways, but it's essentially all about mapping X amount of green threads onto X amount of CPUs. In our case, there isn't any restrictions put on our workload so we should have the same amount of OS threads as we have CPU cores. Any more than this and the CPU will just schedule our threads any way, which is not desired behavior.

For our runtime to be multithreaded we need a scheduler and workerthreads. The schedulers job is to poll the parent `Future` which in turn initiates any tasks or asynchronous functionality. The worker threads are spawned by the scheduler, and it's their job to execute any tasks. Worker threads can be implemented in two main ways. A pull model or push model.

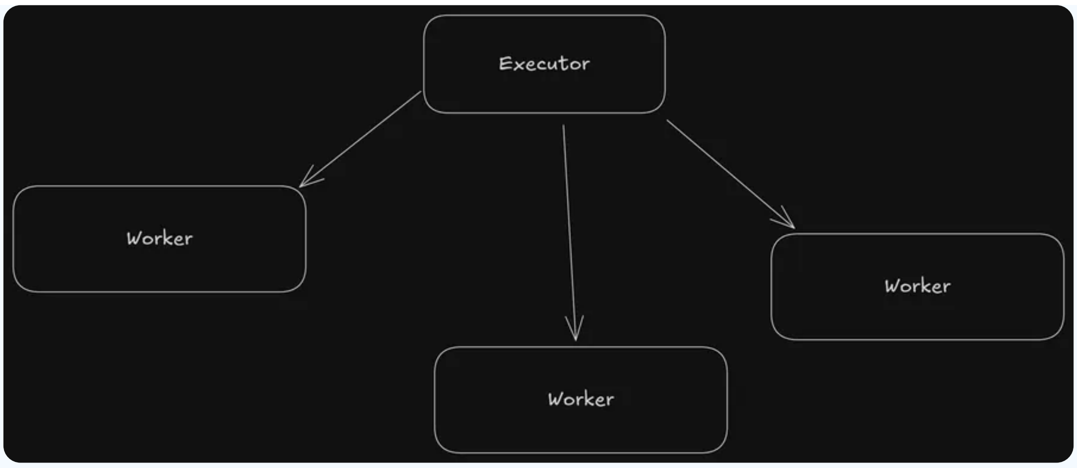
Approach 1: Pull-based model

In a pull model, the workers have total control over retreiving tasks and they prioritise themselves where to get them from. This allows for further optimisation. In this model the scheduler does very little since the workers have almost total control. Because the workers are very separate from the scheduler, we can allow the workers to communicate with each other. If the workers are allowed remote state of each other, we can implement work stealing. Work stealing allows a worker threads to steal tasks from other workers local queue. This levels pressure and makes sure that all workers are approximately an equal amount of workload.


Executor
Worker
Worker
Worker
If global queue is empty a worker can steal tasks from other workers

Approach 2: Push-based model

In a push model, the scheduler have total control over sending tasks to the workers, but the scheduler also needs to assure that all the workers get similar load and can do very little if a unbalance in load occurs. It is much more difficult to optimise and the scheduler has many more roles.



For these reasons we are going to use a pull-based runtime.

Designing our pull-based runtime

In this system, we will have one global FIFO queue and one local FIFO queue for every workerthread. The individual workers will prefer pulling from it's own queue and then take from the global queue, to prevent stale tasks. If the global queue is empty, then try to steal from other workers. We can implement this priority of fetching tasks with this function:

```rust
impl Worker {
  fn fetch_task(&self) -> Option<ArcTask> {
    // Get tasks from local queue
    if let Some(task) = self.local_queue.pop() {
      return Some(task);
    };

    // Try to steal tasks from the global queue
    loop {
      match self.steal_from_global_queue() {
        Steal::Retry => continue,
        Steal::Empty => break,
        Steal::Success(task) => return Some(task),
      };
    }

    // Global queue is empty: So we attempt to steal tasks from other workers.

    // Loop through all workers
    for remote_worker in self.handle.state().remotes.iter() {
      loop {
        // Steal workers and push to local queue, then pop local queue
        match remote_worker.stealer.steal_batch_and_pop(&self.local_queue) {
          // Try again with same remote
          Steal::Retry => continue,
          // Stop trying and move to next remote.
          Steal::Empty => break,
          // Break immediately and return task
          Steal::Success(task) => {
            return Some(task);
          }
        }
      }
    }

    // If none of these worked, return None
    None
  }
}
```

Snippet taken from [***my runtime 'liten's worker task fetching mechanism***]().

The queues contain green threads. In Rust's case it's a `Future` submitted by the user. The user supplies a Future and the runtime wraps that in another future and once the user-submitted Future is ready, the value gets sent in a oneshot channel to the JoinHandle. Once a task has been retreived, we allow little CPU time for that specific task. tasks Poll return decides what to do with it:

- Task returns `Poll::Pending`: Put it aside in a "cold" hashmap. This is to prevent the tasks in the "hot" queue to be polled unecessarily. Once the Future calls the waker, we put it back in the queue.
- Task returns `Poll::Ready`: Simply ignore, the future is a wrapper for the real future with the value, so nothing needs to be done here.

Once done with task, then try fetching another task. If no tasks can be found, the thread is parked for efficiency. The thread is later woken up when new work is available.

```
impl Worker {
  pub fn launch(&mut self) {
    let (sender, receiver) = mpsc::unbounded();
    loop {
      if self.receiver.try_recv().is_ok() {
        break;
      }

      // Add "pollable" tasks from "cold" queue to "hot" queue
      for now_active_task_id in receiver.try_iter() {
        let task = self
          .cold_queue
          .remove(&now_active_task_id)
          .expect("invalid waker called, TaskId doesn't exist");

        self.local_queue.push(task);
      }

      let Some(task) = self.fetch_task() else {
        self.parker.park();
        continue;
      };

      let id = task.id();
      let liten_waker = Arc::new(TaskWaker::new(id, sender.clone())).into();
      let mut context = std::task::Context::from_waker(&liten_waker);

      let unwind_task = task.clone();
      let poll_result = match std::panic::catch_unwind(move || unwind_task.poll(&mut context)) {
        Ok(value) => value,
        Err(_) => continue,
      };

      if Poll::Pending == poll_result {
        let old_value = self.cold_queue.insert(id, task);
        assert!(old_value.is_none(), "logic error of inserted cold_queue task");
      }
    }
  }
}
```

Here the worker puts tasks in and out of the local queue and into/out of a "cold queue". Why bother?

Well part of the reason is that it makes the implementation of the Waker very simple. The only thing the waker needs to do is to move the "parked" task into the hot queue, and the worker will pick it up and poll.

But the real reason is this. Theres two types of Futures, futures that will return Pending and Futures that are ready to be polled. If we know that a Future has returned Pending and wake has not been called, there should be no reason to call poll because the future hasn't made progress yet. So we can store it to the side until wake has been called. The whole point of the async mental model is to only give CPU to code that needs it. So as an optimisation, the tasks that has returned Poll::Pending are put to the side, because we know that they can't make progress, so why loop over the queue and poll futures that are pending?

Now we have a very basic multithreaded, work-stealing async runtime! It's missing features such as a IO-event loop and a timer but the core of a runtime is this. The interesting thing about this pull-based model is that the amount of worker threads can be scaled up or down entirely dynamically.

These snippets are taken from my own async runtime `liten` which already has these extra parts mentioned and even more features planned. Please star the repo if you find this sort of thing interesting!

Further reading

These articles allow for further reading of the same topic, and may be relevant:

- Making the Tokio scheduler 10x faster
- Async Rust in a Nutshell