Green Threads in Rust

KEVIN ROSENDAHL

Additional Key Words and Phrases: Rust, Green Threads, Concurrency

ACM Reference format:
Kevin Rosendahl. 2017. Green Threads in Rust. 1, 1, Article 1 (December 2017), 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUMMARY

For this project, I set out to implement a green threading library written in Rust that offered an interface as similar as possible to **std::thread**. The most challenging aspect of this was interfacing with Rust to accomplish this. Compared to writing a green threading library in C, Rust was very difficult to convince that the implementation is legal. However, once the low level implementation was complete, working in Rust is much better than working in straight C.

In the end, I was able to create a working green thread library in Rust that exposes a **std::thread** like interface. Along the way I learned about Rust's Foreign Function Interface (FFI), the underlying representation of closures and other data types, and gained a much stronger grasp on Rust's compile-time and runtime checking.

I was also able to investigate abstractions that can be built on top of green threads, and compare them to other concurrency abstractions using paradigms other than green threads that attempt to solve similar problems.

2 BACKGROUND

2.1 Threading

Among the different common concurrency paradigms, the most popular historically is threading. Threading allows the programmer to essentially write seperate programs that execute concurrently.

The two main types of threads are "native"/"OS" threads, and "green" threads. "Native" or "OS" threads are maintained by the Operating System the program is running on. These threads will usually be multiplexed over multiple CPU cores if they are available, and are typically preemptively scheduled by the Operating System.

Preemptive scheduling means that the program does not have to explicitly say that it yields control over to another thread. Instead, the Operating System itself will deschedule the thread from the core it is running on and schedule another thread on it automatically.

"Green" threads on the other hand are not scheduled by the Operating System, but instead are scheduled by a runtime running inside the program. This is sometimes referred to as "user level" threading (as opposed to "kernel level" threading). Commonly, these threads are scheduled "cooperatively."

Cooperative scheduling is the opposite of preemptive scheduling. In a cooperative scheduling paradigm, programs must explicitly yield control from the thread, usually back into the runtime which then decides which thread to run next.

2.2 Green Threads in the Wild

Green Threads can be found being used in a number of languages. The usage of green threads most in vogue right now is Go's "goroutine." Go implements a green threads runtime that multiplexes goroutines over multiple CPU cores. Go's runtime however is in charge of scheduling the goroutines; the program does not have to explicitly yield control.

Rust, in fact, has a history with green threads. A green threads runtime used to be the default paradigm for Rust code. Among other reasons (which will be addressed throughout the course of the rest of the paper) the Rust team decided that having a systems language use a green threads runtime did not quite align. Thus, they decided to abandon the green thread runtime and to switch to using native threads.

2.3 Event Loops

Another concurrency pattern that is becoming increasingly popular in mainstream programming languages is the event loop. Event loops have events registered with them, and loop through all of the events, handling any that have notified the loop that they are ready to be handled. Once all events that had triggered for that loop have been handled, the event loop begins again, handling newly triggered events.

One popular example of an eventloop is **libuv**, upon which **node.js** is built. Rust also has a popular event loop library **mio**, upon which the **tokio** components are built.

2.4 Async I/O, Callbacks, Futures, Promises, and Async/Await

Event loops are commonly employed to handle running many different tasks that interact with asynchronous I/O. One method for handling triggered events is providing a callback function. This results in what is commonly referred to as "callback hell," where a program devolves into a series of unparsable callback function redirections.

To combat "callback hell," a number of abstractions have been created. Futures provide a value which will eventually be resolved into the value it wraps. Promises and the async/await syntax are attempts at wrapping futures to make the code look more like typical procedural code, rather than evented callback-style code.

One problem with these styles of abstractions is that they all still leak. Figure 1 shows the proposed syntax for async/await in Rust. At a glance, this looks great. The return type is a normal return type, not a future or any other weird type. However, upon further inspection, we see that the function has to be tagged with **#[async]**. And indeed, all functions which end up calling *fetch_rust_lang* will have to mark themselves with **#[async]**. Soon, your whole program is littered with async notations, and every time you need to change

```
#[async]
 fn fetch_rust_lang(client: hyper::Client) -> io::Result<String> {
     let response = await!(client.get("https://www.rust-lang.org"))?;
     if !response.status().is_success() {
         return Err(io::Error::new(io::ErrorKind::Other, "request failed"))
     }
     let body = await!(response.body().concat())?;
     let string = String::from_utf8(body)?;
     Ok(string)
 }
```

Fig. 1. Proposed Rust async/await syntax

one function that previously did not call an asynchronous function, you have to annotate it and every function that ever calls it.

2.5 Relation to CS242

This project has strong ties to the lessons and themes of CS242. As I've just outlined, the concurrency paradigm that you choose to write your program with has drastic impacts on the syntax, control flow, and general feel of the code.

In addition to the more subjective (but nonetheless incredibly important) aspects described above, the paradigm you choose also can have an impact on the performance of your application. Later on we will discuss how the different design choices that can be made when writing a concurrency library can impact runtime performance.

3 APPROACH

3.1 Why a Rust Green Threads Library?

Writing a green threads library in Rust was something I was very interested in doing. Last year in CS240, I implemented a very basic green threads library in C (with some assembly) as part of a lab[12]. This got me interested in one day diving deeper into different concurrency techniques. I had been developing **node.js** applications for a while and had gotten quite fed up with callback passing, and had started to use a fibers library [10].

Rust has been an interest of mine for a few years now. I've found its static analysis and compiler guarantees, along with its type system, to be incredibly powerful. However, despite writing a few toy projects in Rust I hadn't yet had any real exposure to solving non-trivial problems in the language.

Writing a green threads library in Rust seemed to be a perfect marriage of these two interests.

3.2 Context Switching

A key aspect of a green threads library is the ability to **context switch**. That is, to be able to switch the executing control flow between one green thread to another.

In order to do this, you need to have the context you want to switch into, and a place to store the current context you're switching out of. Figure 2 shows the Rust struct created to represent this.

You can then use pointers to these structs as the arguments to our assembly function *grn_context_switch* (Fig. 3).

However, now we have to find a way to call this function from Rust. Enter Rust's Foreign Function Interface (FFI)[4]. Figure 4 shows the function we create in Rust in order to expose our assembly function.

```
#[repr(C)]
pub struct ThreadContext {
    pub rsp: u64,
    pub r15: u64,
    pub r14: u64,
    pub r13: u64,
    pub r12: u64,
    pub rbx: u64,
    pub rbp: u64,
}
```

Fig. 2. *ThreadContext*

```
grn_context_switch:
  # Move current state to old_context
  mov   %rsp, (%rdi)
  mov   %r15, 0x08(%rdi)
  mov   %r14, 0x10(%rdi)
  mov   %r13, 0x18(%rdi)
  mov   %r12, 0x20(%rdi)
  mov   %rbx, 0x28(%rdi)
  mov   %rbp, 0x30(%rdi)

  # Load new state from new_context
  mov   (%rsi), %rsp
  mov   0x08(%rsi), %r15
  mov   0x10(%rsi), %r14
  mov   0x18(%rsi), %r13
  mov   0x20(%rsi), %r12
  mov   0x28(%rsi), %rbx
  mov   0x30(%rsi), %rbp

  ret
```

Fig. 3. *grn_context_switch*

```
#[link(name = "context_switch", kind = "static")]
extern {
    fn grn_context_switch(old_context: *mut libc::c_void, new_context: *const libc::c_void);
}
```

Fig. 4. Rust *grn_context_switch* FFI function

This Rust wrapper, along with a build script using the *cc* library allows us to call out to the assembly function *grn_context_switch*. One interesting thing to note here is the annotation **#[repr(C)]** on the *ThreadContext* struct. This tells the Rust compiler that it should lay the struct out in memory as C would, which allows us to use the struct as we do in *grn_context_switch*.

3.3 Thread Stacks

Implementing the context switch required a decent amount of research into Rust's FFI, the compiler, how to get it to link, and the memory layout of structs. That said, it was a pretty straightforward implementation. No real design decisions had to be made.

Once you have context switching working, the next step is to create a stack for the eventual green thread to use. This sounds straightforward: just allocate some memory and call it a day. However, it turns out there's a large number of implications depending on the strategy you use for stack allocation and management.

The first style of stack is just like a normal operating thread stack. You simply allocate a chunk of memory on the heap, and use it as the stack. The main benefit of this technique is its simplicity. Additionally, if a thread ends up using a large portion of the stack, you get the benefit of only having to allocate memory once.

However, many times programs wish to use green threads as "lightweight tasks." Allocating a relatively large stack just to perform

```
#[repr(C)]
pub struct ThreadStack {
    pub inner: Box<[u8]>
}

impl ThreadStack {
    pub fn new () -> ThreadStack {
        let boxed : Box<[u8]> = vec![0; STACK_SIZE].into_boxed_slice();
        ThreadStack {
            inner: boxed,
        }
    }
}
```

Fig. 5. *ThreadStack*

a small task will not be very performant, and if many green threads are being spawned as is not unusual, could cause the program to use excessive amounts of memory.

A technique used to address this concern is called a "segmented stack." A segmented stack begins small (maybe around 8KB compared to a 2MB regular stack). When more stack space is required, the request is trapped, and a new stack segment is allocated, and used to "extend" the stack via a pointer. When this extra space is no longer required, the new stack segment is freed.

Besides the obvious added complexity of maintaining the different segments of the stack, there can be some pitfalls when using segmented stacks. Both Rust and Go began by writing runtimes using segmented stacks, and both eventually moved away from them. The common issue shared between both Rust's[9] and Go's[2] implementations was what they called "stack thrashing" or a "hot split" problem.

This problem arises when there is a tight loop in the program that is constantly hitting a stack segment boundary. In each iteration of the loop, the runtime will have to allocate a new segment, and at the end of the iteration the runtime will have to deallocate the new segment. If this occurs in a sufficiently critical section of the program, performance can be drastically diminish.

Go decided to take a pretty reasonable approach to the issue. The Go runtime will now start with a small stack size, similar to when using segmented stacks. However, now when the goroutine runs out of stack, the runtime will allocate a new, larger (2x as big) stack, and copy the original stack contents to the new stack. The runtime will also analyze the contents of the stack, and translate any any pointers that point at the old stack to be pointers onto the new stack at the proper location.

Go made a reasonable tradeoff, as can be seen in the Benchmarks section[2].

I believed that implementing such a scheme for my library was out of the scope of this project, so for simplicity's sake I decided to statically allocate a 2MB stack for each new green thread. Fig. 5 shows the *ThreadStack* struct and Builder used.

3.4 Bootstrapping the Thread

Now that we are able to switch contexts and we have chosen our stack management strategy, we are ready to try to actually run a green thread.

I thought that given my previous work on the C green threads library, porting the solution to Rust would be trivial. A number of factors made this assumption naive. The first factor is that Rust is, in fact, not C. While C will let you do just about anything with just about any data, you have to be much more mindful in Rust about how this is executed. The second is that I wanted to write an ergonomically sound library. The C library I had previously written accepted only static function pointers. This made the implementation very simple, but greatly reduces the utility of the library. I wanted my Rust library to be something that could actually be used. This meant at least trying to match the patterns provided in **std::thread**. *std::thread::spawn*'s signature is shown in Fig. 6.

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Fig. 6. *std::thread::spawn*

```
grn_bootstrap_thread:
  push    %rbp
  mov     %rsp, %rbp
  mov     0x8(%rbp), %r11
  callq   *%r11
  callq   _grn_exit
```

Fig. 7. C *grn_bootstrap_thread*

```
uint64_t *stack_head = (uint64_t *)&thread->stack[STACK_SIZE - 16];
*stack_head = (uint64_t)grn_bootstrap_thread;
*(stack_head + 1) = (uint64_t)fn;

thread->context.rsp = (uint64_t)stack_head;
```

Fig. 8. C stack initialization

In English, this signature basically says that it takes in an argument f, which is a closure that is safe to be sent across thread boundaries, which returns a value of type T which is also safe to be sent across thread boundaries.

The general technique for bootstrapping a green thread is to manipulate the stack pointer value once the context has been switched into to make control jump to a trampoline function. The initial stack of the green thread should also be manipulated such that the trampoline function can find the value of the function that should be called, and call it.

In C this is quite simple. Fig. 7 shows the trampoline function. It expects that when it is called, there will be the address of the function that should be called should be at the top of the stack. It then calls the function at that address, then calls the global variable *_grn_exit* to jump back into the C library and continue in the runtime.

Fig. 8 shows the C code necessary to set up the trampoline function. It simply puts the user function's address at the beginning of the stack, puts the address of *grn_bootstrap_thread* on the stack and points the stack pointer there.

However, we have not made it so easy on ourselves in the Rust library. The first issue that we run into is that the function we accept is not in fact a function pointer, but instead a closure trait

```
extern "C" fn call_fn(user_fn: *mut c_void, cb_fn: *mut c_void) {
    unsafe {
        let user_fn = transmute::<*mut c_void, Box<Box<FnBox()>>>(user_fn);
        (*user_fn)();

        let cb_fn = transmute::<*mut c_void, Box<Box<FnBox()>>>(cb_fn);
        (*cb_fn)();
    }
}
```

Fig. 9. *call_fn*

```
// expects:
// [ *callback            ] <- start of stack
// [ *wrapped_user_fn     ]
// [ *call_fn             ]
// [ *grn_bootstrap_thread ]
grn_bootstrap_thread:
  push    %rbp
  mov     %rsp, %rbp
  mov     0x8(%rbp), %r11

  # Call call_fn with wrapped_user_fn
  mov     0x10(%rbp), %rdi
  mov     0x18(%rbp), %rsi
  callq   *%r11
```

Fig. 10. Rust *grn_bootstrap_thread*

object[5]. The long and short of this is that we cannot simply take the address of this value, pass it around, and call it later. The trait object contains more information than just the function, it also contains the environment the function closes over.

The next issue is that we don't want to call some global *_grn_exit* function, we want to call a method on the *Runtime* struct that's in charge of keeping track of the set of active green threads. Eventually we arrive at the following. We create a new global function called *call_fn* (Fig. 9). This function takes in two void pointers as arguments. However, it then *transmutes* these pointers (essentially casting them back into Rust's typed, owned scheme) into pointers to pointers to closures. The first argument should point at the user's function, and the second function should point to the closure that is in charge of delegating back into the runtime after a thread exits. Fig. 10 shows the new *grn_bootstrap_thread* implementation, and Fig. 11 shows how some of the stack locations and the stack pointer were set.

The Rust code seems verbose, but makes sense. However, what I have included in Fig. 11 includes a bug that plagued me for longer than I care to admit. A program would run completely fine all the way through, but would crash while cleaning up resources at the end. For some reason, *Thread.stack.inner* was a null pointer. Fig. 12 shows the *Thread* struct, along with its memory layout. As can be seen, the *ThreadContext* is above directly below the *ThreadStack* in the address space[3]. *ThreadContext.rsp* in particular is 7 bytes before the *ThreadStack*'s pointer to where its slice is on the heap. When we copy pointers onto the stack, we use *std::ptr::copy(src, dst, 8)*. The 8 is necessary since we are copying into a *[u8]*, and since a pointer on our supported platforms (**x86_64**) is 8 bytes long. However, when we use the same call to copy over the *bootstrap_thread_ptr* into *ThreadContext.rsp*, we're now copying into a *u64*. This then copied the values into *ThreadContext.rsp* and the seven 8-byte memory locations after it. The last of these locations was where the pointer to the stack was held, overwriting it with *0x0*. Since none of our library actually touches the stack, the thread simply uses it as a

```
pub fn spawn<F, T>(&self, f: F) where
    F: FnOnce() -> T, F: Send + 'static, T: Send + 'static
{
    // ...
    let stack_ptr;
    let rsp_ptr;
    {
        let threads = self.threads.borrow();
        let thread = threads.get(&new_thread_id).unwrap();
        stack_ptr = &((*thread).stack.inner[0]) as *const u8 as *mut u8;
        rsp_ptr = &((*thread).context.rsp) as *const u64 as *mut u64;
    }

    let callback : Box<FnBox()> = Box::new(move || {
        self.exit();
    });
    let callback_fn_ptr = Box::new(callback);

    // ...

    unsafe {
        let stack_head = stack_ptr.offset(thread::STACK_SIZE as isize);

        let bootstrap_thread_ptr = Box::new(bootstrap_thread_ptr);
        let bootstrap_thread_stack_loc = stack_head.offset(-32isize);
        let bootstrap_thread_ptr = &*bootstrap_thread_ptr as *const _ as *mut _;
        std::ptr::copy(bootstrap_thread_ptr, bootstrap_thread_stack_loc, 8);

        //...

        let callback_fn_ptr = Box::new(callback_fn_ptr);
        let callback_fn_stack_loc = stack_head.offset(-8isize);
        let callback_fn_ptr = &*callback_fn_ptr as *const _ as *mut _;
        std::ptr::copy(callback_fn_ptr, callback_fn_stack_loc, 8);

        let bootstrap_thread_ptr = Box::new(bootstrap_thread_stack_loc);
        let bootstrap_thread_ptr = &*bootstrap_thread_ptr as *const _ as *mut _;
        std::ptr::copy(bootstrap_thread_ptr, rsp_ptr, 8);

        std::mem::forget(wrapped_user_fn_ptr);
        std::mem::forget(callback_fn_ptr);
    }
}
```

Fig. 11. Rust stack initialization

```
#[repr(C)]
pub struct Thread {
    pub id: u64,                 // 0x00
    pub status: ThreadStatus,    // 0x08
    pub context: ThreadContext, // 0x10-0x38
    pub stack: ThreadStack,      // 0x40-0x48
}
```

Fig. 12. *Thread* struct layout

stack, no issue comes up while the threads are running. However, when we go to deallocate the *ThreadStack*, its pointer is null, and the program crashes.

I think this bug really highlights what Rust is trying to accomplish. C would let the programmer make that mistake as much as they wanted to. But Rust made me really work to shoot myself in the foot. And while debugging it took a lot more time and a lot more *gdb* than I wish it had, I knew roughly where in the code the problem had to be. It had to be in one of the places that I had wrapped in *unsafe*.

3.5 Cross-thread Reference Passing

I wanted to have *Runtime* struct that was in charge of spawning, yielding, and cleaning up threads when they exited. Fig. 13 shows the basic outline of these methods.

As seen in Fig. 11, the callback we call *spawn* is simply *self.exit*. However, as we can see further down in *yield_now_to_status*, there

```
pub fn yield_now(&self) {
    self.yield_now_to_status(ThreadStatus::Ready)
}

fn exit(&self) {
    self.yield_now_to_status(ThreadStatus::Zombie);
}

fn yield_now_to_status(&self, status : ThreadStatus) {
    // pseudo code below
    let next_thread_idx = self.next_thread();

    let current thread = self.threads.get(self.current);
    *current_thread.status = status;
    let cur_ptr = &current_thread.context;
    let new_ptr = &self.threads.get(next_thread_idx);

    self.current = next_thread_idx;

    unsafe {
        let cur_ptr = transmute::<*mut ThreadContext, *mut c_void>(cur_ptr);
        let new_ptr = transmute::<*mut ThreadContext, *mut c_void>(new_ptr);
        grn_context_switch(cur_ptr, new_ptr);
    }
}
```

Fig. 13. *yield_now* and *exit*

```
pub struct Runtime {
    threads: RefCell<BTreeMap<u64, thread::Thread>>,
    current: RefCell<u64>,
    id_counter: RefCell<u64>,
}
```

Fig. 14. *Runtime*

is a decent amount of mutation that needs to be done when both yielding and exiting. Alas, when we actually consume the library, we want to be able to call *yield_now* from within any number of threads. This would require us to pass mutable references to the *Runtime* into each closure that is run in a green thread. However, Rust's borrow checker clearly won't allow this.

The answer to this problem is **std::cell::RefCell**. The *RefCell* allows you to push Rust's borrow checker requirements to runtime. Here we know that there will de facto only ever be one thread operating on these objects at a time, but Rust's compiler is not able to understand that. So we wrap all of the members of *Runtime* that we need to access in *RefCell*s, and call *borrow* and *borrow_mut* to access them. Fig. 14 shows what the *Runtime* struct ends being defined as. Fig. 15 shows the main body of *yield_now_to_status*.

One interesting thing to note here is the explicit blocks required. My original attempt at the rewritten *yield_now_to_status* did not include the explicit blocks, and would panic complaining that *self.threads* was borrowed mutably multiple times at ones. This was because *grn_context_switch* would switch threads before the destructor of the temporarily borrowed value was run. This means that the temporarily borrowed value never released its ownership, and when the next thread got to *yield_now_to_status* and tried to call *self.threads.borrow_mut*, it saw it as already borrowed mutably and panicked.

4 RESULTS

I accomplished the task I originally set out to: build a green threads library in Rust with a reasonably ergonomic API. Fig. 16 shows a sample program and its output.

I was also able to play around with implementing some higher level primitives. Notably, I implemented a *sync* function that runs a green thread, but does not require the invoking function to explicitly call *yield_now*. Fig. 17 shows a sample program using *sync*, and Fig. 18 shows *sync*'s implementation.

```
let cur_ptr;
{
    let mut threads = self.threads.borrow_mut();
    let mut current_thread = threads.get_mut(&self.current.borrow()).unwrap();
    current_thread.status = status;
    cur_ptr = &((*current_thread).context) as *const _ as *mut _;
}

let new_ptr;
{
    let mut threads = self.threads.borrow_mut();
    let mut new_thread = threads.get_mut(&next_thread_idx).unwrap();
    new_thread.status = ThreadStatus::Running;
    new_ptr = &((*new_thread).context) as *const _ as *mut _;
}

{
    let mut current = self.current.borrow_mut();
    *current = next_thread_idx;
}

unsafe {
    let cur_ptr = transmute::<*mut ThreadContext, *mut c_void>(cur_ptr);
    let new_ptr = transmute::<*mut ThreadContext, *mut c_void>(new_ptr);
    grn_context_switch(cur_ptr, new_context_ptr);
}
```

Fig. 15. *yield_now_to_status*

```
fn main() {
    thread::spawn(|| {
        for i in 0..5 {
            println!("in thread {}", i);
            thread::yield_now();
        }
    });

    for i in 0..5 {
        println!("in main {}", i);
        thread::yield_now();
    }

    println!("back out in main");
}

$ cargo run
in main 0
in thread 0
in main 1
in thread 1
in main 2
in thread 2
in main 3
in thread 3
in main 4
in thread 4
back out in main
```

Fig. 16. Sample program

I wished to also write a wrapper around an asynchronous I/O operation to be able to compare both the style and the speed of the green threads library vs something like **tokio**. However every async I/O library I found seemed to be coupled too tightly to either **mio** or some other paradigm to easily integrate with my library.

Barring being able to do that, I will qualitatively compare the library I have created and the possible abstractions that could be built on top with past, present, and future Rust options.

```
fn main() {
    thread::sync(|| {
        for i in 0..5 {
            println!("in thread {}", i);
            thread::yield_now();
        }
    });

    println!("back out in main");
}

$ cargo run
in thread 0
in thread 1
in thread 2
in thread 3
in thread 4
back out in main
```

Fig. 17. Sample program

```
pub fn sync<F, T>(f: F) where
    F: FnOnce() -> T, F: Send + 'static, T: Send + 'static
{
    let exited = Arc::new(Mutex::new(false));
    let exited_clone = exited.clone();

    spawn(move || {
        f();
        *exited_clone.lock().unwrap() = true;
    });

    while !*exited.lock().unwrap() {
        yield_now();
    }
}
```

Fig. 18. Sample program

4.1 libgreen

Originally Rust came with a green threads runtime, however it was later stripped in favor of solely using native threads and supporting blocking I/O in the standard library. There reasons were as follows[1]:

- **Forced co-evolution**: they were concerned that having both green and native threading models and supporting the same I/O API for them was both an undue burden and also yielded a lowest common denominator interface, not an ideal one.
- **Overhead**: Including the green thread runtime doubled the size of a basic "hello world" binary.
- **Problematic I/O interactions**: Not all of the asynchronous I/O functions used by the green thread runtime were *actually* asynchronous. Making a blocking call in a green thread can end up blocking the worker thread running the green scheduler.
- **Embedding**: Setting up the runtime when calling Rust code from C or other languages requires a lot of work
- **Maintenance burden**: It's hard to maintain two different implementations

I think the Rust team made the correct choice in stripping the runtime from Rust and putting it into a seperate library *libgreen*[7]. Rust is meant to be a highly performant systems language. It should not paint you into a corner where you cannot get maximal performance, and you absolutely need access to native threads and native APIs.

However, my library has one huge advantage over *libgreen*. You can actually use it today. *libgreen* has not been committed to in over 3 years, half a year before Rust hit v1. In examining the source you will not find Rust as you know it, but instead many different complicated now defunct syntax elements.

4.2 tokio

tokio is a bold attempt to implement a zero-cost abstraction over asynchronous I/O. The total number of characters of code you have to produce to get seriously impressive results is low, and it includes some nice combinators over the *Futures* that its functions return.

The technical work behind **tokio** is world class, yet the number one complaint they get is "it's confusing!"[8]. In fact the initial v0.1 release is blocked on explaining to the world how they're async abstraction isn't confusing. They're current hope is that Rust adding *async/await* syntax[13] will aid in that.

4.3 async/await

The hardships that **tokio** is facing is not something new. Recently, the exact same arguments were had in the **node.js** community. Four years ago, everyone was complaining about "callback hell," so people implemented *futures*, and wrapped them up in *promises*, and the world rejoiced. But then everyone realized they hated *promises*, so they decided instead that you should just have to write *async* before a function that is asynchronous, and *await* whenever you're calling another asynchronous function from within a function marked *async*.

The problem with this explicit *async/await* is that it spreads like the plague. When you need to add asynchronous I/O to a function, you then have to pop up the call stack everywhere that function is called, and annotate the call with *await*, mark the calling function as *async*, and repeat.

So while *async/await* may be syntactically better than *futures* and *promises*, there is still this underlying issue of function infection. This issue is nicely summed up in *What Color is Your Function?*[11].

4.4 A Happy Medium

Computer Science is all about tradeoffs. Go has a runtime built on green threads that automatically handles I/O asynchronously for you. Rust has an event-loop based asynchronous I/O framework that can produce an http server that can handle 33% more requests[6], but nobody can figure out how to use it [8].

Is there a single right answer? Of course not. **tokio** should absolutely exist, and should aim to provide the most efficient possible implementation. Hopefully *async/await* will make it palatable. And for many systems languages, maybe that's where we would want to stop.

But one day I hope to see Rust used not only as a high performance systems language, but also as an ergonomic, developer friendly general purpose programming language. And if with a green threads runtime that automatically handled asynchronous I/O you could get performance comparable to say Go, I could absolutely see it being used as such. But in that future, I know I for one will not be trying to write *async* **tokio**.

REFERENCES

[1] 0000-remove-runtime. (????). https://github.com/aturon/rfcs/blob/remove-runtime/active/0000-remove-runtime.md
[2] Contiguous stacks. (????). https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub
[3] Rust Container Cheat Sheet. (????). https://docs.google.com/presentation/d/1q-c7UAyrUlM-eZyTo1pd8SZ0qwA_wYxmPZVOQkoDmH4/edit#slide=id.p
[4] Trait Objects. (????). https://doc.rust-lang.org/book/first-edition/ffi.html
[5] Trait Objects. (????). https://doc.rust-lang.org/book/first-edition/trait-objects.html
[6] Zero-cost futures in Rust. (????). https://aturon.github.io/blog/2016/08/11/futures/
[7] alexcrichton. libgreen. (????). https://github.com/alexcrichton/green-rs
[8] alexcrichton. 2017. Restructure documentation from the ground up. (June 2017). https://github.com/tokio-rs/tokio/issues/13
[9] Brian Anderson. 2013. Abandoning segmented stacks in Rust. (Nov. 2013). https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html
[10] laverdet. node-fibers. (????). https://github.com/laverdet/node-fibers
[11] Bob Nystrom. 2015. Restructure documentation from the ground up. (Feb. 2015). http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/
[12] CS240 Staff. Lab 1: Cooperative Green Threads. (????). http://www.scs.stanford.edu/17sp-cs240/labs/lab1/
[13] steveklabnik. 2015. Async IO. (April 2015). https://github.com/rust-lang/rfcs/issues/1081