Writing an OS in Rust Philipp Oppermann's blog

« All Posts

A Freestanding Rust Binary

Feb 10, 2018

The first step in creating our own operating system kernel is to create a Rust executable that does not link the standard library. This makes it possible to run Rust code on the bare metal without an underlying operating system.

This blog is openly developed on GitHub. If you have any problems or questions, please open an issue there. You can also leave comments at the bottom. The complete source code for this post can be found in the post-01 branch.

▶ Table of Contents

Introduction

To write an operating system kernel, we need code that does not depend on any operating system features. This means that we can't use threads, files, heap memory, the network, random numbers, standard output, or any other features requiring OS abstractions or specific hardware. Which makes sense, since we're trying to write our own OS and our own drivers.

This means that we can't use most of the Rust standard library, but there are a lot of Rust features that we *can* use. For example, we can use iterators, closures, pattern matching, option and result, string formatting, and of course the ownership system. These features make it possible to write a kernel in a very expressive, high level way without worrying about undefined behavior or memory safety.

In order to create an OS kernel in Rust, we need to create an executable that can be run without an underlying operating system. Such an executable is often called a "freestanding" or "baremetal" executable.

This post describes the necessary steps to create a freestanding Rust binary and explains why the steps are needed. If you're just interested in a minimal example, you can **jump to the summary**.

Disabling the Standard Library

By default, all Rust crates link the standard library, which depends on the operating system for features such as threads, files, or networking. It also depends on the C standard library libc, which closely interacts with OS services. Since our plan is to write an operating system, we can

not use any OS-dependent libraries. So we have to disable the automatic inclusion of the standard library through the no std attribute.

We start by creating a new cargo application project. The easiest way to do this is through the command line:

```
cargo new blog os --bin --edition 2018
```

I named the project blog_os, but of course you can choose your own name. The --bin flag specifies that we want to create an executable binary (in contrast to a library) and the --edition 2018 flag specifies that we want to use the 2018 edition of Rust for our crate. When we run the command, cargo creates the following directory structure for us:

```
blog_os

├─ Cargo.toml

└─ src

└─ main.rs
```

The Cargo.toml contains the crate configuration, for example the crate name, the author, the semantic version number, and dependencies. The src/main.rs file contains the root module of our crate and our main function. You can compile your crate through cargo build and then run the compiled blog_os binary in the target/debug subfolder.

The no std Attribute

Right now our crate implicitly links the standard library. Let's try to disable this by adding the no_std_attribute:

```
// main.rs

#![no_std]

fn main() {
    println!("Hello, world!");
}
```

When we try to build it now (by running cargo build), the following error occurs:

The reason for this error is that the println macro is part of the standard library, which we no longer include. So we can no longer print things. This makes sense, since println writes to standard output, which is a special file descriptor provided by the operating system.

So let's remove the printing and try again with an empty main function:

```
// main.rs

#![no_std]

fn main() {}

> cargo build
error: `#[panic_handler]` function required, but not found
error: language item required, but not found: `eh_personality`
```

Now the compiler is missing a #[panic_handler] function and a language item.

Panic Implementation

The panic_handler attribute defines the function that the compiler should invoke when a panic occurs. The standard library provides its own panic handler function, but in a no_std environment we need to define it ourselves:

```
// in main.rs

use core::panic::PanicInfo;

/// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

The PanicInfo parameter contains the file and line where the panic happened and the optional panic message. The function should never return, so it is marked as a diverging function by returning the "never" type ! . There is not much we can do in this function for now, so we just loop indefinitely.

The eh personality Language Item

Language items are special functions and types that are required internally by the compiler. For example, the Copy trait is a language item that tells the compiler which types have *copy* semantics. When we look at the implementation, we see it has the special #[lang = "copy"] attribute that defines it as a language item.

While providing custom implementations of language items is possible, it should only be done as a last resort. The reason is that language items are highly unstable implementation details and not even type checked (so the compiler doesn't even check if a function has the right argument types). Fortunately, there is a more stable way to fix the above language item error.

The eh_personality language item marks a function that is used for implementing stack unwinding. By default, Rust uses unwinding to run the destructors of all live stack variables in case of a panic. This ensures that all used memory is freed and allows the parent thread to catch the panic and continue execution. Unwinding, however, is a complicated process and requires some OS specific libraries (e.g. libunwind on Linux or structured exception handling on Windows), so we don't want to use it for our operating system.

Disabling Unwinding

There are other use cases as well for which unwinding is undesirable, so Rust provides an option to abort on panic instead. This disables the generation of unwinding symbol information and thus considerably reduces binary size. There are multiple places where we can disable unwinding. The easiest way is to add the following lines to our Cargo.toml:

```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

This sets the panic strategy to abort for both the dev profile (used for cargo build) and the release profile (used for cargo build --release). Now the eh_personality language item should no longer be required.

Now we fixed both of the above errors. However, if we try to compile it now, another error occurs:

```
> cargo build
error: requires `start` lang_item
```

Our program is missing the start language item, which defines the entry point.

The start attribute

One might think that the main function is the first function called when you run a program. However, most languages have a runtime system, which is responsible for things such as garbage collection (e.g. in Java) or software threads (e.g. goroutines in Go). This runtime needs to be called before main, since it needs to initialize itself.

In a typical Rust binary that links the standard library, execution starts in a C runtime library called crt0 ("C runtime zero"), which sets up the environment for a C application. This includes creating a stack and placing the arguments in the right registers. The C runtime then invokes the entry point of the Rust runtime, which is marked by the start language item. Rust only has a very minimal runtime, which takes care of some small things such as setting up stack overflow guards or printing a backtrace on panic. The runtime then finally calls the main function.

Our freestanding executable does not have access to the Rust runtime and crt0, so we need to define our own entry point. Implementing the start language item wouldn't help, since it would still require crt0. Instead, we need to overwrite the crt0 entry point directly.

Overwriting the Entry Point

To tell the Rust compiler that we don't want to use the normal entry point chain, we add the #! [no main] attribute.

```
#![no_std]
#![no_main]

use core::panic::PanicInfo;

/// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

You might notice that we removed the main function. The reason is that a main doesn't make sense without an underlying runtime that calls it. Instead, we are now overwriting the operating system entry point with our own start function:

```
#[no_mangle]
pub extern "C" fn _start() -> ! {
    loop {}
}
```

By using the #[no_mangle] attribute we disable the name mangling to ensure that the Rust compiler really outputs a function with the name _start . Without the attribute, the compiler would generate some cryptic _ZN3blog_os4_start7hb173fedf945531caE symbol to give every function an unique name. The attribute is required because we need to tell the name of the entry point function to the linker in the next step.

We also have to mark the function as extern "C" to tell the compiler that it should use the C calling convention for this function (instead of the unspecified Rust calling convention). The

reason for naming the function _start is that this is the default entry point name for most systems.

The ! return type means that the function is diverging, i.e. not allowed to ever return. This is required because the entry point is not called by any function, but invoked directly by the operating system or bootloader. So instead of returning, the entry point should e.g. invoke the exit system call of the operating system. In our case, shutting down the machine could be a reasonable action, since there's nothing left to do if a freestanding binary returns. For now, we fulfill the requirement by looping endlessly.

When we run cargo build now, we get an ugly linker error.

Linker Errors

The linker is a program that combines the generated code into an executable. Since the executable format differs between Linux, Windows, and macOS, each system has its own linker that throws a different error. The fundamental cause of the errors is the same: the default configuration of the linker assumes that our program depends on the C runtime, which it does not.

To solve the errors, we need to tell the linker that it should not include the C runtime. We can do this either by passing a certain set of arguments to the linker or by building for a bare metal target.

Building for a Bare Metal Target

By default Rust tries to build an executable that is able to run in your current system environment. For example, if you're using Windows on $x86_64$, Rust tries to build a .exe Windows executable that uses $x86_64$ instructions. This environment is called your "host" system.

To describe different environments, Rust uses a string called *target triple*. You can see the target triple for your host system by running rustc --version --verbose:

```
rustc 1.35.0-nightly (474e7a648 2019-04-07) binary: rustc commit-hash: 474e7a6486758ea6fc761893b1a49cd9076fb0ab commit-date: 2019-04-07 host: x86_64-unknown-linux-gnu release: 1.35.0-nightly LLVM version: 8.0
```

The above output is from a $x86_64$ Linux system. We see that the host triple is $x86_64$ -unknown-linux-gnu, which includes the CPU architecture ($x86_64$), the vendor (unknown), the operating system (linux), and the ABI (gnu).

By compiling for our host triple, the Rust compiler and the linker assume that there is an underlying operating system such as Linux or Windows that use the C runtime by default, which causes the linker errors. So to avoid the linker errors, we can compile for a different environment with no underlying operating system.

An example for such a bare metal environment is the thumbv7em-none-eabihf target triple, which describes an embedded ARM system. The details are not important, all that matters is that the target triple has no underlying operating system, which is indicated by the none in the target triple. To be able to compile for this target, we need to add it in rustup:

```
rustup target add thumbv7em-none-eabihf
```

This downloads a copy of the standard (and core) library for the system. Now we can build our freestanding executable for this target:

```
cargo build --target thumbv7em-none-eabihf
```

By passing a --target argument we cross compile our executable for a bare metal target system. Since the target system has no operating system, the linker does not try to link the C runtime and our build succeeds without any linker errors.

This is the approach that we will use for building our OS kernel. Instead of thumbv7em-none-eabihf, we will use a custom target that describes a x86_64 bare metal environment. The details will be explained in the next post.

Linker Arguments

Instead of compiling for a bare metal system, it is also possible to resolve the linker errors by passing a certain set of arguments to the linker. This isn't the approach that we will use for our kernel, therefore this section is optional and only provided for completeness. Click on "Linker Arguments" below to show the optional content.

▶ Linker Arguments

Summary

A minimal freestanding Rust binary looks like this:

```
#![no_std] // don't link the Rust standard library
#![no_main] // disable all Rust-level entry points
use core::panic::PanicInfo;
```

```
#[no mangle] // don't mangle the name of this function
pub extern "C" fn start() -> ! {
    // this function is the entry point, since the linker looks for a function
    // named `_start` by default
    loop {}
}
/// This function is called on panic.
#[panic handler]
fn panic( info: &PanicInfo) -> ! {
    loop {}
}
Cargo.toml:
[package]
name = "crate_name"
version = "0.1.0"
authors = ["Author Name <author@example.com>"]
# the profile used for `cargo build`
[profile.dev]
panic = "abort" # disable stack unwinding on panic
# the profile used for `cargo build --release`
[profile.release]
panic = "abort" # disable stack unwinding on panic
```

To build this binary, we need to compile for a bare metal target such as thumbv7em-none-eabihf:

```
cargo build --target thumbv7em-none-eabihf
```

Alternatively, we can compile it for the host system by passing additional linker arguments:

```
# Linux
cargo rustc -- -C link-arg=-nostartfiles
# Windows
cargo rustc -- -C link-args="/ENTRY:_start /SUBSYSTEM:console"
# macOS
cargo rustc -- -C link-args="-e __start -static -nostartfiles"
```

Note that this is just a minimal example of a freestanding Rust binary. This binary expects various things, for example that a stack is initialized when the _start function is called. **So for any real use of such a binary, more steps are required**.

What's next?

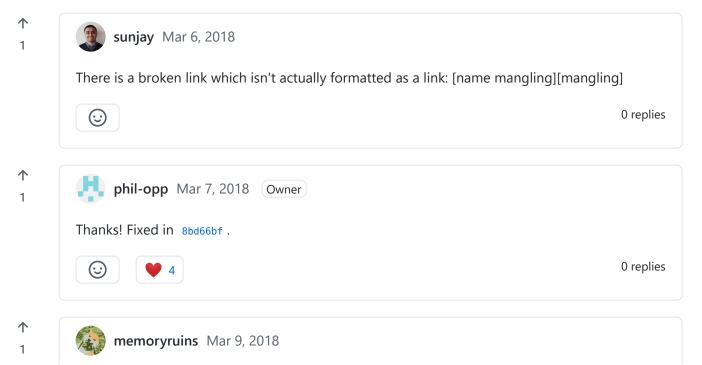
The next post explains the steps needed for turning our freestanding binary into a minimal operating system kernel. This includes creating a custom target, combining our executable with a bootloader, and learning how to print something to the screen.

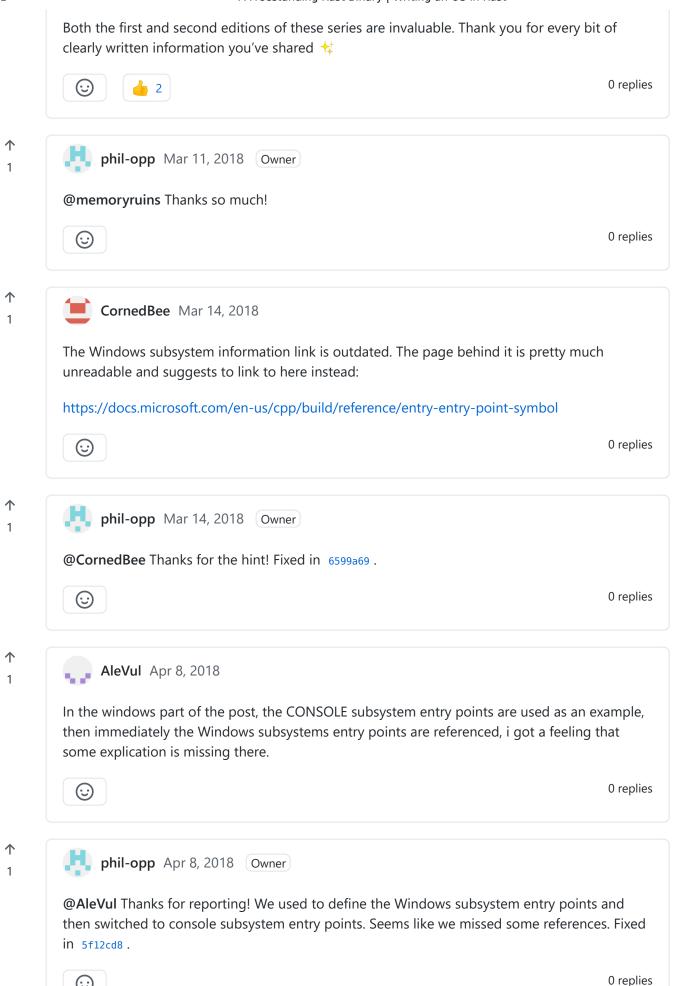
Support Me

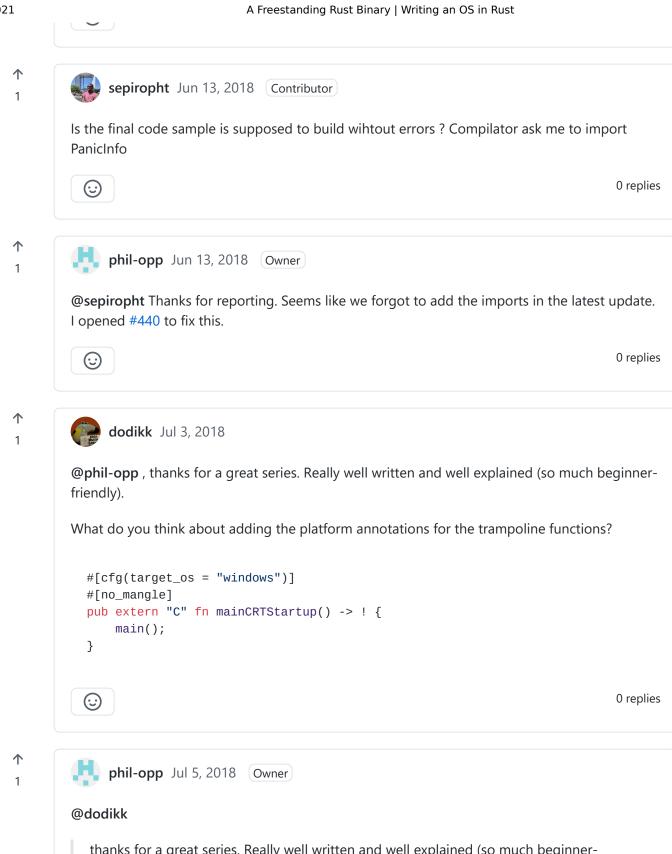
Creating and maintaining this blog and the associated libraries is a lot of work, but I really enjoy doing it. By supporting me, you allow me to invest more time in new content, new features, and continuous maintenance.

The best way to support me is to *sponsor me on GitHub*, since they don't charge any fees. If you prefer other platforms, I also have *Patreon* and *Donorbox* accounts. The latter is the most flexible as it supports multiple currencies and one-time contributions.

56 comments – powered by giscus







thanks for a great series. Really well written and well explained (so much beginner-friendly).

Thanks a lot!

What do you think about adding the platform annotations for the trampoline functions?

These functions are just used to get a compiling no_std executable on existing operating systems in the first post. It wouldn't make sense to keep them for the subsequent posts

because our OS kernel won't ever run on top of Windows or macOS. Only adding the conditional compilation for the first post would be possible, but I don't think that it's worth it because it makes things more complicated and the functions are removed in the second post anyway.





0 replies





AregevDev Aug 17, 2018

But you have two main() functions, one for Windows and one for MacOS. It errors on it, am I missing something?



0 replies





phil-opp Aug 17, 2018 Owner

@AregevDev You should only use one of the three variants, depending on your host OS. So if you're on Windows, only add the Windows-style entry point and not the Linux and macOS entry points.



0 replies





andre-richter Aug 17, 2018 (Contributor)

edited

@AregevDev An alternative approach to having os-specific entrypoints would be the introduction of a custom linker script. There, you can define the entry-point function via ENTRY().

I think the first edition of the blog talked about it.

One way or the other, you have a little piece of inconvenience. When writing a kernel, though, in the long run you will probably end up with a custom linker script at some point anyways.





0 replies

26 hidden items **Load more...**





Fibration Jan 4, 2020

@phil-opp Thanks for the tutorial. I have been enjoying following it so far. I am stuck at cargo

xbuild and have checked against your files in the post-02 branch to make sure that everything is identical.

I am receiving the following error at the end of cargo xbuild:

error: cannot produce bin for `os v0.1.0` as the target `x86_64-os` does not su

Cargo now uses a more informative error message. Issue here. I assume that xbuild is still using an older version of cargo.

I could not get any further than that. Any insight you have would be great.



0 replies





phil-opp Jan 5, 2020 Owner

@Fibration

Cargo now uses a more informative error message. Issue here. I assume that xbuild is still using an older version of cargo.

cargo-xbuild just calls out to the cargo you have installed. Maybe you did not add a nightly override for your directory (either though rustup override add or a rust-toolchain file) so that the default cargo is a stable version instead of a more recent nightly version?

Either way, the old error message contains the same number of information, it just uses a different formulation. The error indicates that your target does not support the bin crate type, which likely means that you don't have the "executables": true line in your target JSON:

blog_os/x86_64-blog_os.json Line 9 in fea4fa9

9 "executables": true,

Note that needs to be *executables* (plural), not *executable*. Unfortunately, cargo just ignores invalid target configuration keys instead of throwing an error for them.

I hope this solves your issue!



0 replies

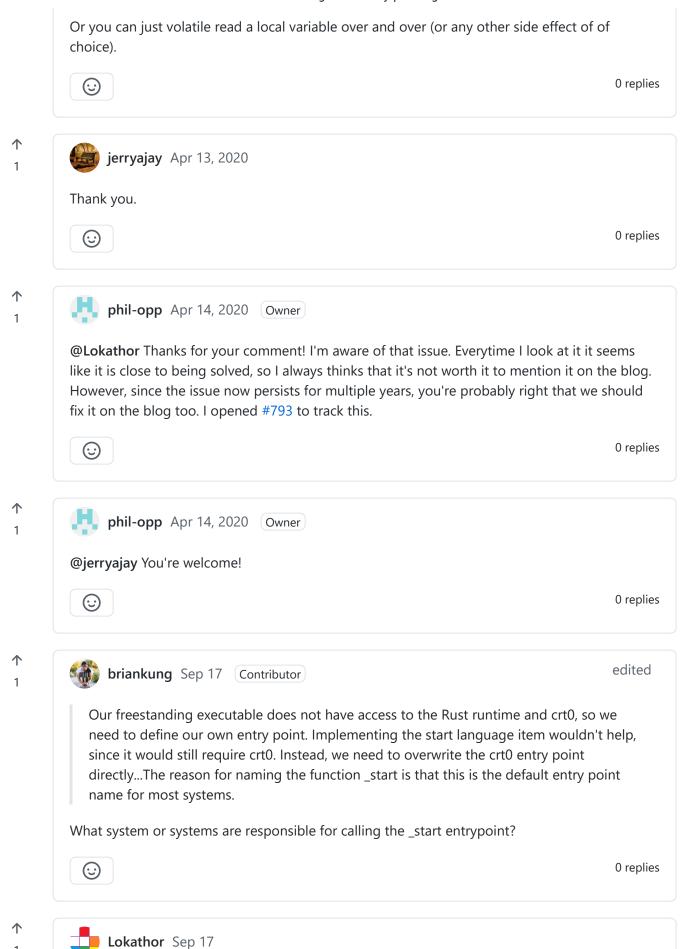




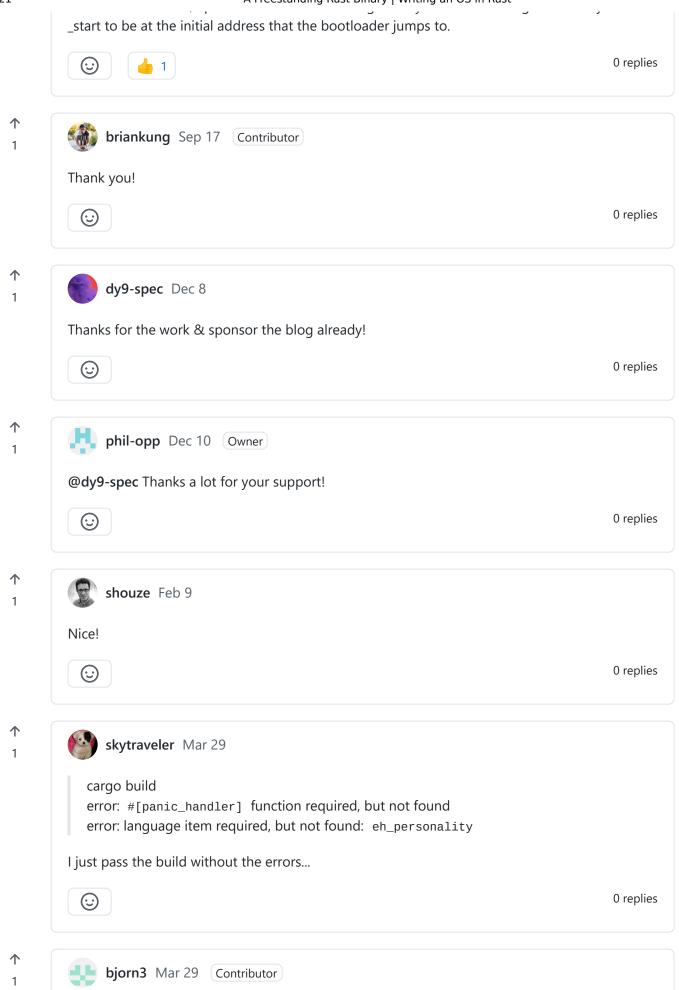
Lokathor Mar 27, 2020

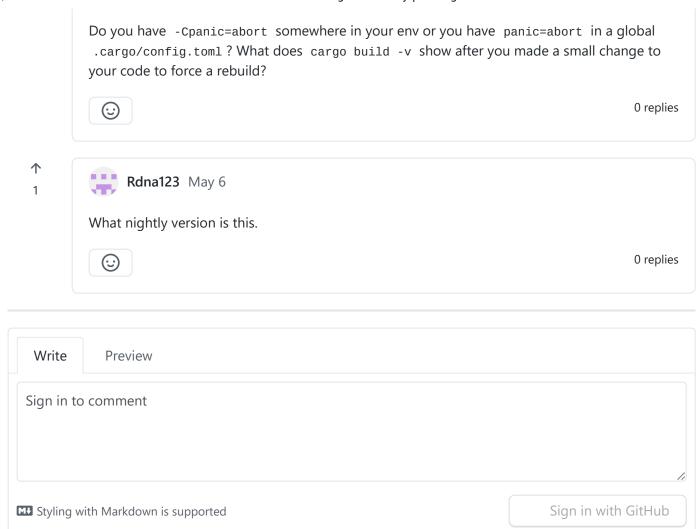
Infinite loop with no side effect is UB because of LLVM problems.

there is a -Z flag to fix this, rust-lang/rust#59546



since this booted to, i presume the linker is configured by default to arrange the binary for the





Instead of authenticating the giscus application, you can also comment directly on the on GitHub. Just click the "X comments" link at the top — or the date of any comment — to go to the GitHub discussion.

Other Languages

- Persian
- Japanese
- Chinese (simplified)
- Chinese (traditional)

About Me

I'm a Rust freelancer with a master's degree in computer science. I love systems programming, open source software, and new challenges.

If you want to work with me, reach out on LinkedIn or write me at job@phil-opp.com.

© 2021. All rights reserved. Contact