A Freestanding Rust Binary

Feb 10, 2018

Other Lang
- Chinese (s
- Chinese (t
- French
- Japanese
- Persian
- Russian
- Korean
- Arabic

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.

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 "bare-metal" 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't 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:

```
error: cannot find macro `println!` in this scope
 --> src/main.rs:4:5
  |
4 |     println!("Hello, world!");
  |     ^^^^^^^
```

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 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 a 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 an `.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 uses 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 of 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:

`src/main.rs`:

```
#![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*. Thank you!

A Minimal Rust Kernel »

Comments

Do you have a problem, want to share feedback, or discuss further ideas? Feel free to leave a comment here! Please stick to English and follow Rust's code of conduct. This comment thread directly maps to a *discussion on GitHub*, so you can also comment there if you prefer.

57 reactions

👍 23 🎉 11 ❤️ 16 🚀 6 👀 1

72 comments · 33+ replies – *powered by giscus*

Oldest Newest

sunjay Mar 6, 2018

There is a broken link which isn't actually formatted as a link: [name mangling][mangling]

↑ 1 1 reply

tifandotme Aug 5, 2023

test

phil-opp Mar 7, 2018 Owner

Thanks! Fixed in `8bd66bf` .

↑ 3 ❤️ 7 0 replies

memoryruins Mar 10, 2018

Both the first and second editions of these series are invaluable. Thank you for every bit of clearly written information you've shared ✨

↑ 1 👍 5 0 replies

phil-opp Mar 11, 2018 Owner

@memoryruins Thanks so much!

↑ 1 👍 1 0 replies

CornedBee Mar 14, 2018

The Windows subsystem information link is outdated. The page behind it is pretty much unreadable and suggests to link to here instead:

https://docs.microsoft.com/en-us/cpp/build/reference/entry-entry-point-symbol

↑ 1 0 replies

phil-opp Mar 14, 2018 Owner

@CornedBee Thanks for the hint! Fixed in `6599a69` .

↑ 1 0 replies

AleVul Apr 8, 2018

In the windows part of the post, the CONSOLE subsystem entry points are used as an example, then immediately the Windows subsystems entry points are referenced, i got a feeling that some explication is missing there.

↑ 1 0 replies

phil-opp Apr 8, 2018 Owner

@AleVul Thanks for reporting! We used to define the Windows subsystem entry points and then switched to console subsystem entry points. Seems like we missed some references. Fixed in `5f12cd8` .

↑ 1 0 replies

sepiropht Jun 13, 2018 Contributor

Is the final code sample is supposed to build wihtout errors ? Compilator ask me to import PanicInfo

↑ 1 0 replies

phil-opp Jun 14, 2018 Owner

@sepiropht Thanks for reporting. Seems like we forgot to add the imports in the latest update. I opened #440 to fix this.

↑ 1 0 replies

dodikk Jul 3, 2018

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

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

```
#[cfg(target_os = "windows")]
#[no_mangle]
pub extern "C" fn mainCRTStartup() -> ! {
    main();
}
```

↑ 1 0 replies

phil-opp Jul 5, 2018 Owner

@dodikk

> 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.

↑ 1 👍 2 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?

↑ 1 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.

↑ 1 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.

↑ 1 😄 1 0 replies

42 hidden items
Load more...

ghost Jul 26, 2021

عالی

↑ 1 0 replies

letto4135 Aug 28, 2021 edited

There is a linking issue with mac, not sure if it is M1 specific, but
`rustc --version --verbose`
returns x86_64-apple-darwin as the host
`rustup target add x86_64-apple-darwin && cargo build --target x86_64-apple-darwin`
ends in
`ld: entry point (_main) undefined. for architecture x86_64 clang: error: linker command failed with exit code 1 (use -v to see invocation)`

↑ 1 1 reply

bjorn3 Feb 8, 2022 Contributor

See https://os.phil-opp.com/freestanding-rust-binary/#linker-errors, this is expected.

snowmang1 May 22, 2022

This blog is amazing, thank you so much for the concise walk through and explanations!

↑ 0 1 reply

phil-opp Jun 7, 2022 Owner

Thanks!

Cerber-Ursi Jun 6, 2022

@phil-opp, is it intentional that under the original English post there's not this thread, but the thread for Russian translation instead?..

↑ 1 3 replies

phil-opp Jun 7, 2022 Owner

That's not intentional, thanks for reporting this! Let me try to fix it...

phil-opp Jun 7, 2022 Owner

Should be fixed now.

Cerber-Ursi Jun 16, 2022

Thanks. Btw, in that discussion I've added a question (before noticing the thread was wrong), did you see it?

UncleScientist Jul 21, 2022

I'm streaming my walkthrough of this blog series; would it be appropriate for me to share a link?

↑ 2 0 replies

XieGuochao Oct 24, 2022

Would it be interesting to create a version that can run on Mac M1's `qemu-system-aarch64` ?

↑ 1 4 replies

bjorn3 Oct 24, 2022 Contributor

While the most of the basic concepts are similar between x86_64 and aarch64, the actual implementation is significantly different between the two to the point that blog os would likely have to be rewritten entirely from scratch. Additionally while x86_64 is mostly standardized, the same is not the case for aarch64. There are significant differences between aarch64 processors. Even something as basic as the interrupt controller differs between processors. While most aarch64 processors nowadays use GIC (generic interrupt controller), apple uses their own AIC (generic interrupt controller), except in VM's where they have a partial GIC implementation in hardware and emulate the rest in the vm software. And even for GIC there are several versions: https://developer.arm.com/Architectures/Generic%20Interrupt%20Controller And that is not even talking about the boot process for which there is pretty much no standardization outside of the systems that use UEFI (mostly server aarch64 processors, but U-Boot also supports emulating a UEFI subset).

andre-richter Oct 24, 2022 Contributor

Shameless plug: For AArch64, you can take a look at https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials

👍 1

XieGuochao Oct 25, 2022

Thank you for the explanation and interesting resources! What about other architectures like RISC-V?

bjorn3 Oct 25, 2022 Contributor

RISC-V has less fragmentation with respect to booting. Basically everything with support for user mode uses either SBI or UEFI afaik. There are also microcontrollers which only support machine mode (equivalent to SMM on x86) and no supervisor mode or user mode.

https://osblog.stephenmarz.com/ seems to be a good tutorial for writing a RISC-V OS in Rust.

ochudi Jun 29, 2023

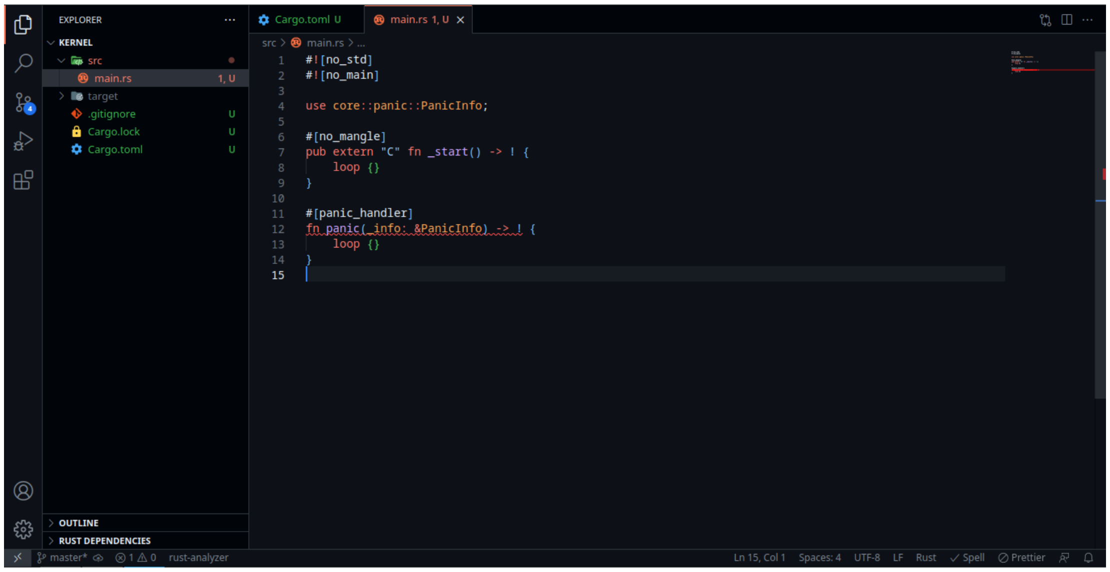
Thank you!

Just followed the first post to the end. You were recommended by my lecturer. Learning OS Dev.

↑ 1 ❤️ 1 0 replies

ghost Jul 28, 2023

I am getting an error is vscode



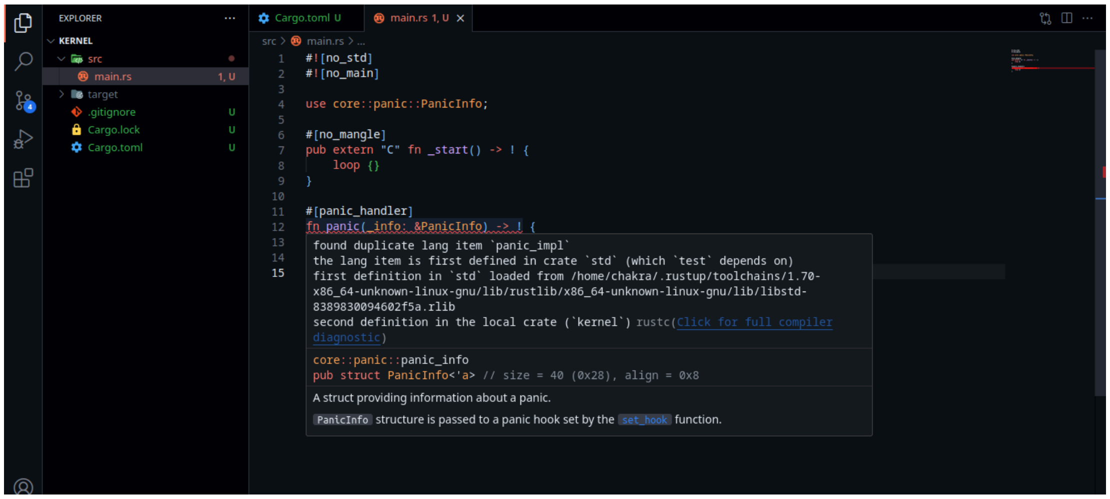
```rust
#![no_std]
#![no_main]

use core::panic::PanicInfo;

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

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}

```

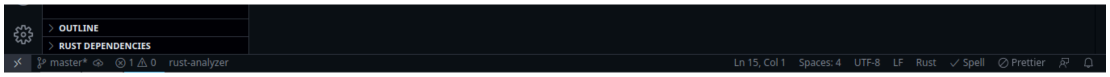


```
found duplicate lang item `panic_impl`
the lang item is first defined in crate `std` (which `test` depends on)
first definition in `std` loaded from /home/chakra/.rustup/toolchains/1.70-
x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-linux-gnu/lib/libstd-
8389830094602f5a.rlib
second definition in the local crate (`kernel`) rustc(Click for full compiler
diagnostic)

core::panic::panic_info
pub struct PanicInfo<'a> // size = 40 (0x28), align = 0x8
```

A struct providing information about a panic.

`PanicInfo` structure is passed to a panic hook set by the `set_hook` function.



```
found duplicate lang item `panic_impl`
the lang item is first defined in crate `std` (which `test` depends on)
first definition in `std` loaded from /home/chakra/.rustup/toolchains/1.70-x86_64-unkn
```

Any idea how to solve this ?

↑ 1 19 replies

Show 14 previous replies

testingapisname Dec 7, 2023

Just did the below and get a different error now :D

```
PS C:\RustOS\blog_os> rustup override set nightly
info: using existing install for 'nightly-x86_64-pc-windows-msvc'
info: override toolchain for 'C:\RustOS\blog_os' set to 'nightly-x86_64-pc-window

  nightly-x86_64-pc-windows-msvc unchanged - rustc 1.76.0-nightly (503e12932 2023

PS C:\RustOS\blog_os> cargo build
warning: unused manifest key: build
warning: unused manifest key: target.cfg(target_os = "none").runner
warning: unused manifest key: toolchain
warning: unused manifest key: unstable
   Compiling blog_os v0.1.0 (C:\RustOS\blog_os)
error: linking with `link.exe` failed: exit code: 1561
  |
  = note: "C:\\Program Files (x86)\\Microsoft Visual Studio\\2022\\BuildTools\\VC
  = note: LINK : fatal error LNK1561: entry point must be defined


error: could not compile `blog_os` (bin "blog_os") due to 1 previous error
PS C:\RustOS\blog_os>
```

testingapisname Dec 7, 2023

Tried the build as below and getting this error:

```
PS C:\RustOS\blog_os> cargo build --target thumbv7em-none-eabihf
warning: unused manifest key: build
warning: unused manifest key: target.cfg(target_os = "none").runner
warning: unused manifest key: toolchain
warning: unused manifest key: unstable
   Compiling blog_os v0.1.0 (C:\RustOS\blog_os)
error[E0463]: can't find crate for `core`
  |
  = note: the `thumbv7em-none-eabihf` target may not be installed
  = help: consider downloading the target with `rustup target add thumbv7em-none-
  = help: consider building the standard library from source with `cargo build -Z

error[E0463]: can't find crate for `compiler_builtins`
```

```
error[E0463]: can't find crate for `core`
 --> src\main.rs:6:5
  |
6 | use core::panic::PanicInfo;
  |     ^^^^ can't find crate
  |
  = note: the `thumbv7em-none-eabihf` target may not be installed
  = help: consider downloading the target with `rustup target add thumbv7em-none-
  = help: consider building the standard library from source with `cargo build -Z

error: requires `sized` lang_item

For more information about this error, try `rustc --explain E0463`.
error: could not compile `blog_os` (bin "blog_os") due to 4 previous errors
PS C:\RustOS\blog_os>
PS C:\RustOS\blog_os> rustup install core
error: invalid toolchain name: 'core'
PS C:\RustOS\blog_os>
```

testingapisname Dec 7, 2023

Success!

```
PS C:\RustOS\blog_os> cargo build --target thumbv7em-none-eabihf
warning: unused manifest key: build
warning: unused manifest key: target.cfg(target_os = "none").runner
warning: unused manifest key: toolchain
warning: unused manifest key: unstable
   Compiling blog_os v0.1.0 (C:\RustOS\blog_os)
    Finished dev [unoptimized + debuginfo] target(s) in 0.12s
PS C:\RustOS\blog_os>
```

testingapisname Dec 7, 2023

Needed to read the error and do the step with help:

```
PS C:\RustOS\blog_os> rustup target add thumbv7em-none-eabihf
```

ghost Dec 7, 2023

Nice

cleanwk Jan 26

excellent article, thanks to the author🤩

↑ 1 ❤️ 1 1 reply

phil-opp Feb 7 Owner

Thanks :)

pooriatgh Feb 17

Incase you ran into issue like below:

ound duplicate lang item `panic_impl`
he lang item is first defined in crate `std` (which `test` depends on)

try this from : rust-lang/rust-analyzer#4490 (comment)

works like a charm!

#[cfg(not(test))]
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
loop {}
}

what I understood is the test crate also depends on some standard library so better to disable it.

↑ 1 👍 1 1 reply

harshithnrao Jul 18

thank you !!

HarshMohanSason Mar 23

Excellent. This will boost up my rust skills as well along with OS knowledge. Thanks

↑ 1 ❤️ 1 0 replies

Blindspot22 May 11

Wow! Have had an idea of creating an OS with rust before and have had really tough times. Little did i know this post existed. Thanks to all who made this possible

↑ 1 ❤️ 1 🚀 1 0 replies

inabakumorioss Jun 26

Great Guide!!! Would you consider making an update showing how to interact with the screen aside from plain text(as in writing individual pixels and clearing the screen for a full display system)?

↑ 1 1 reply

phil-opp Jul 25 Owner

Yes, this is planned. I'm not sure when it will be ready yet though.

jwmurray 16 days ago

Has anyone installed this onto a Pi Pico or MircoBits or other arm device? I would like to give it a try and wonder which ones were successful.

↑ 1 1 reply

bjorn3 16 days ago Contributor

This is a tutorial for writing an OS for x86_64. While the high level things you need to implement for an arm OS are the same, all the low level details are different enough that this tutorial will likely not be of much help apart from introducing the basic concepts.

hgfernan 3 days ago

I shall read your text with lots of interest. But there's a question that I'm keeps disturbing my reading: any OS that I know about has parts in assembly language to cope with very specific hardware questions that higher level languages can't cope with.

How will you integrate that with the protected environment that Rust intends to create ?

↑ 1 0 replies

Write Preview Aa

Sign in to comment

Sign in with GitHub

Instead of authenticating the giscus application, you can also comment directly *on GitHub*.