REF:

<https://os.phil-opp.com/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](https://en.wikipedia.org/wiki/Bare_machine) without an underlying operating system.

This blog is openly developed on [GitHub](https://github.com/phil-opp/blog_os). If you have any problems or questions, please open an issue there. You can also leave comments [at the bottom](https://os.phil-opp.com/freestanding-rust-binary/#comments). The complete source code for this post can be found in the [post-01](https://github.com/phil-opp/blog_os/tree/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](https://doc.rust-lang.org/std/), but there are a lot of Rust features that we can use. For example, we can use [iterators](https://doc.rust-lang.org/book/ch13-02-iterators.html), [closures](https://doc.rust-lang.org/book/ch13-01-closures.html), [pattern matching](https://doc.rust-lang.org/book/ch06-00-enums.html), [option](https://doc.rust-lang.org/core/option/) and [result](https://doc.rust-lang.org/core/result/), [string formatting](https://doc.rust-lang.org/core/macro.write.html), and of course the [ownership system](https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html). These features make it possible to write a kernel in a very expressive, high level way without worrying about [undefined behavior](https://www.nayuki.io/page/undefined-behavior-in-c-and-cplusplus-programs) or [memory safety](https://tonyarcieri.com/it-s-time-for-a-memory-safety-intervention).

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**](https://os.phil-opp.com/freestanding-rust-binary/#summary).

## Disabling the Standard Library

By default, all Rust crates link the [standard library](https://doc.rust-lang.org/std/), 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](https://doc.rust-lang.org/1.30.0/book/first-edition/using-rust-without-the-standard-library.html).

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](https://doc.rust-lang.org/nightly/edition-guide/rust-2018/index.html) 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](https://semver.org/) 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](https://doc.rust-lang.org/1.30.0/book/first-edition/using-rust-without-the-standard-library.html):

// 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](https://doc.rust-lang.org/std/macro.println.html) 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](https://en.wikipedia.org/wiki/Standard_streams#Standard_output_.28stdout.29), 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](https://doc.rust-lang.org/stable/book/ch09-01-unrecoverable-errors-with-panic.html) 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](https://doc.rust-lang.org/nightly/core/panic/struct.PanicInfo.html) 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](https://doc.rust-lang.org/1.30.0/book/first-edition/functions.html#diverging-functions) by returning the [“never” type](https://doc.rust-lang.org/nightly/std/primitive.never.html) !. 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](https://doc.rust-lang.org/nightly/core/marker/trait.Copy.html) trait is a language item that tells the compiler which types have [copy semantics](https://doc.rust-lang.org/nightly/core/marker/trait.Copy.html). When we look at the [implementation](https://github.com/rust-lang/rust/blob/485397e49a02a3b7ff77c17e4a3f16c653925cb3/src/libcore/marker.rs#L296-L299), 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](https://github.com/rust-lang/rust/blob/edb368491551a77d77a48446d4ee88b35490c565/src/libpanic_unwind/gcc.rs#L11-L45) marks a function that is used for implementing [stack unwinding](https://www.bogotobogo.com/cplusplus/stackunwinding.php). By default, Rust uses unwinding to run the destructors of all live stack variables in case of a [panic](https://doc.rust-lang.org/stable/book/ch09-01-unrecoverable-errors-with-panic.html). 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](https://www.nongnu.org/libunwind/) on Linux or [structured exception handling](https://docs.microsoft.com/en-us/windows/win32/debug/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](https://github.com/rust-lang/rust/pull/32900) 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](https://en.wikipedia.org/wiki/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](https://github.com/rust-lang/rust/blob/bb4d1491466d8239a7a5fd68bd605e3276e97afb/src/libstd/rt.rs#L32-L73), 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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](https://clang.llvm.org/docs/CrossCompilation.html#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](https://en.wikipedia.org/wiki/Application_binary_interface) (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](https://en.wikipedia.org/wiki/Embedded_system) [ARM](https://en.wikipedia.org/wiki/ARM_architecture) 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](https://en.wikipedia.org/wiki/Cross_compiler) 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](https://doc.rust-lang.org/rustc/targets/custom.html) 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

In this section we discuss the linker errors that occur on Linux, Windows, and macOS, and explain how to solve them by passing additional arguments to the linker. Note that the executable format and the linker differ between operating systems, so that a different set of arguments is required for each system.

#### Linux

On Linux the following linker error occurs (shortened):

error: linking with `cc` failed: exit code: 1

|

= note: "cc" […]

= note: /usr/lib/gcc/../x86\_64-linux-gnu/Scrt1.o: In function `\_start':

(.text+0x12): undefined reference to `\_\_libc\_csu\_fini'

/usr/lib/gcc/../x86\_64-linux-gnu/Scrt1.o: In function `\_start':

(.text+0x19): undefined reference to `\_\_libc\_csu\_init'

/usr/lib/gcc/../x86\_64-linux-gnu/Scrt1.o: In function `\_start':

(.text+0x25): undefined reference to `\_\_libc\_start\_main'

collect2: error: ld returned 1 exit status

The problem is that the linker includes the startup routine of the C runtime by default, which is also called \_start. It requires some symbols of the C standard library libc that we don’t include due to the no\_std attribute, therefore the linker can’t resolve these references. To solve this, we can tell the linker that it should not link the C startup routine by passing the -nostartfiles flag.

One way to pass linker attributes via cargo is the cargo rustc command. The command behaves exactly like cargo build, but allows to pass options to rustc, the underlying Rust compiler. rustc has the -C link-arg flag, which passes an argument to the linker. Combined, our new build command looks like this:

cargo rustc -- -C link-arg=-nostartfiles

Now our crate builds as a freestanding executable on Linux!

We didn’t need to specify the name of our entry point function explicitly since the linker looks for a function with the name \_start by default.

#### Windows

On Windows, a different linker error occurs (shortened):

error: linking with `link.exe` failed: exit code: 1561

|

= note: "C:\\Program Files (x86)\\…\\link.exe" […]

= note: LINK : fatal error LNK1561: entry point must be defined

The “entry point must be defined” error means that the linker can’t find the entry point. On Windows, the default entry point name [depends on the used subsystem](https://docs.microsoft.com/en-us/cpp/build/reference/entry-entry-point-symbol). For the CONSOLE subsystem, the linker looks for a function named mainCRTStartup and for the WINDOWS subsystem, it looks for a function named WinMainCRTStartup. To override the default and tell the linker to look for our \_start function instead, we can pass an /ENTRY argument to the linker:

cargo rustc -- -C link-arg=/ENTRY:\_start

From the different argument format we clearly see that the Windows linker is a completely different program than the Linux linker.

Now a different linker error occurs

error: linking with `link.exe` failed: exit code: 1221

|

= note: "C:\\Program Files (x86)\\…\\link.exe" […]

= note: LINK : fatal error LNK1221: a subsystem can't be inferred and must be

defined

This error occurs because Windows executables can use different [subsystems](https://docs.microsoft.com/en-us/cpp/build/reference/entry-entry-point-symbol). For normal programs, they are inferred depending on the entry point name: If the entry point is named main, the CONSOLE subsystem is used, and if the entry point is named WinMain, the WINDOWS subsystem is used. Since our \_start function has a different name, we need to specify the subsystem explicitly:

cargo rustc -- -C link-args="/ENTRY:\_start /SUBSYSTEM:console"

We use the CONSOLE subsystem here, but the WINDOWS subsystem would work too. Instead of passing -C link-arg multiple times, we use -C link-args which takes a space separated list of arguments.

With this command, our executable should build successfully on Windows.

#### [🔗](https://os.phil-opp.com/freestanding-rust-binary/#macos)macOS

On macOS, the following linker error occurs (shortened):

error: linking with `cc` failed: exit code: 1

|

= note: "cc" […]

= note: ld: entry point (\_main) undefined. for architecture x86\_64

clang: error: linker command failed with exit code 1 […]

This error message tells us that the linker can’t find an entry point function with the default name main (for some reason, all functions are prefixed with a \_ on macOS). To set the entry point to our \_start function, we pass the -e linker argument:

cargo rustc -- -C link-args="-e \_\_start"

The -e flag specifies the name of the entry point function. Since all functions have an additional \_ prefix on macOS, we need to set the entry point to \_\_start instead of \_start.

Now the following linker error occurs:

error: linking with `cc` failed: exit code: 1

|

= note: "cc" […]

= note: ld: dynamic main executables must link with libSystem.dylib

for architecture x86\_64

clang: error: linker command failed with exit code 1 […]

macOS [does not officially support statically linked binaries](https://developer.apple.com/library/archive/qa/qa1118/_index.html) and requires programs to link the libSystem library by default. To override this and link a static binary, we pass the -static flag to the linker:

cargo rustc -- -C link-args="-e \_\_start -static"

This still does not suffice, as a third linker error occurs:

error: linking with `cc` failed: exit code: 1

|

= note: "cc" […]

= note: ld: library not found for -lcrt0.o

clang: error: linker command failed with exit code 1 […]

This error occurs because programs on macOS link to crt0 (“C runtime zero”) by default. This is similar to the error we had on Linux and can also be solved by adding the -nostartfiles linker argument:

cargo rustc -- -C link-args="-e \_\_start -static -nostartfiles"

Now our program should build successfully on macOS.

error: linking with `cc` failed: exit code: 1

|

= note: "cc" […]

= note: ld: library not found for -lcrt0.o

clang: error: linker command failed with exit code 1 […]

This error occurs because programs on macOS link to crt0 (“C runtime zero”) by default. This is similar to the error we had on Linux and can also be solved by adding the -nostartfiles linker argument:

cargo rustc -- -C link-args="-e \_\_start -static -nostartfiles"

Now our program should build successfully on macOS.

Right now we have different build commands depending on the host platform, which is not ideal. To avoid this, we can create a file named .cargo/config.toml that contains the platform-specific arguments:

# in .cargo/config.toml

[target.'cfg(target\_os = "linux")']

rustflags = ["-C", "link-arg=-nostartfiles"]

[target.'cfg(target\_os = "windows")']

rustflags = ["-C", "link-args=/ENTRY:\_start /SUBSYSTEM:console"]

[target.'cfg(target\_os = "macos")']

rustflags = ["-C", "link-args=-e \_\_start -static -nostartfiles"]

The rustflags key contains arguments that are automatically added to every invocation of rustc. For more information on the .cargo/config.toml file, check out the [official documentation](https://doc.rust-lang.org/cargo/reference/config.html).

Now our program should be buildable on all three platforms with a simple cargo build

#### Should You Do This?

While it’s possible to build a freestanding executable for Linux, Windows, and macOS, it’s probably not a good idea. The reason is that our executable still expects various things, for example that a stack is initialized when the \_start function is called. Without the C runtime, some of these requirements might not be fulfilled, which might cause our program to fail, e.g. through a segmentation fault.

If you want to create a minimal binary that runs on top of an existing operating system, including libc and setting the #[start] attribute as described [here](https://doc.rust-lang.org/1.16.0/book/no-stdlib.html) is probably a better idea.

## 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**.

## [🔗](https://os.phil-opp.com/freestanding-rust-binary/#what-s-next)What’s next?

The [next post](https://os.phil-opp.com/minimal-rust-kernel/) 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](https://os.phil-opp.com/status-update/) 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](https://github.com/sponsors/phil-opp), since they don't charge any fees. If you prefer other platforms, I also have [Patreon](https://www.patreon.com/phil_opp) and [Donorbox](https://donorbox.org/phil-opp) accounts. The latter is the most flexible as it supports multiple currencies and one-time contributions.

Thank you!

next post:

<https://os.phil-opp.com/minimal-rust-kernel/>

# A Minimal Rust Kernel

Feb 10, 2018

In this post, we create a minimal 64-bit Rust kernel for the x86 architecture. We build upon the [freestanding Rust binary](https://os.phil-opp.com/freestanding-rust-binary/) from the previous post to create a bootable disk image that prints something to the screen.

This blog is openly developed on [GitHub](https://github.com/phil-opp/blog_os). If you have any problems or questions, please open an issue there. You can also leave comments [at the bottom](https://os.phil-opp.com/minimal-rust-kernel/#comments). The complete source code for this post can be found in the [post-02](https://github.com/phil-opp/blog_os/tree/post-02) branch.

**Table of Contents**

## [🔗](https://os.phil-opp.com/minimal-rust-kernel/#the-boot-process)The Boot Process

When you turn on a computer, it begins executing firmware code that is stored in motherboard [ROM](https://en.wikipedia.org/wiki/Read-only_memory). This code performs a [power-on self-test](https://en.wikipedia.org/wiki/Power-on_self-test), detects available RAM, and pre-initializes the CPU and hardware. Afterwards, it looks for a bootable disk and starts booting the operating system kernel.

On x86, there are two firmware standards: the “Basic Input/Output System“ ([**BIOS**](https://en.wikipedia.org/wiki/BIOS)) and the newer “Unified Extensible Firmware Interface” ([**UEFI**](https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface)). The BIOS standard is old and outdated, but simple and well-supported on any x86 machine since the 1980s. UEFI, in contrast, is more modern and has much more features, but is more complex to set up (at least in my opinion).

Currently, we only provide BIOS support, but support for UEFI is planned, too. If you’d like to help us with this, check out the [Github issue](https://github.com/phil-opp/blog_os/issues/349).

### BIOS Boot

Almost all x86 systems have support for BIOS booting, including newer UEFI-based machines that use an emulated BIOS. This is great, because you can use the same boot logic across all machines from the last century. But this wide compatibility is at the same time the biggest disadvantage of BIOS booting, because it means that the CPU is put into a 16-bit compatibility mode called [real mode](https://en.wikipedia.org/wiki/Real_mode) before booting so that archaic bootloaders from the 1980s would still work.

But let’s start from the beginning:

When you turn on a computer, it loads the BIOS from some special flash memory located on the motherboard. The BIOS runs self-test and initialization routines of the hardware, then it looks for bootable disks. If it finds one, control is transferred to its bootloader, which is a 512-byte portion of executable code stored at the disk’s beginning. Most bootloaders are larger than 512 bytes, so bootloaders are commonly split into a small first stage, which fits into 512 bytes, and a second stage, which is subsequently loaded by the first stage.

The bootloader has to determine the location of the kernel image on the disk and load it into memory. It also needs to switch the CPU from the 16-bit [real mode](https://en.wikipedia.org/wiki/Real_mode) first to the 32-bit [protected mode](https://en.wikipedia.org/wiki/Protected_mode), and then to the 64-bit [long mode](https://en.wikipedia.org/wiki/Long_mode), where 64-bit registers and the complete main memory are available. Its third job is to query certain information (such as a memory map) from the BIOS and pass it to the OS kernel.

Writing a bootloader is a bit cumbersome as it requires assembly language and a lot of non insightful steps like “write this magic value to this processor register”. Therefore, we don’t cover bootloader creation in this post and instead provide a tool named [bootimage](https://github.com/rust-osdev/bootimage) that automatically prepends a bootloader to your kernel.

If you are interested in building your own bootloader: Stay tuned, a set of posts on this topic is already planned!

#### The Multiboot Standard

To avoid that every operating system implements its own bootloader, which is only compatible with a single OS, the [Free Software Foundation](https://en.wikipedia.org/wiki/Free_Software_Foundation) created an open bootloader standard called [Multiboot](https://wiki.osdev.org/Multiboot) in 1995. The standard defines an interface between the bootloader and the operating system, so that any Multiboot-compliant bootloader can load any Multiboot-compliant operating system. The reference implementation is [GNU GRUB](https://en.wikipedia.org/wiki/GNU_GRUB), which is the most popular bootloader for Linux systems.

To make a kernel Multiboot compliant, one just needs to insert a so-called [Multiboot header](https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#OS-image-format) at the beginning of the kernel file. This makes it very easy to boot an OS from GRUB. However, GRUB and the Multiboot standard have some problems too:

* They support only the 32-bit protected mode. This means that you still have to do the CPU configuration to switch to the 64-bit long mode.
* They are designed to make the bootloader simple instead of the kernel. For example, the kernel needs to be linked with an [adjusted default page size](https://wiki.osdev.org/Multiboot#Multiboot_2), because GRUB can’t find the Multiboot header otherwise. Another example is that the [boot information](https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-information-format), which is passed to the kernel, contains lots of architecture-dependent structures instead of providing clean abstractions.
* Both GRUB and the Multiboot standard are only sparsely documented.
* GRUB needs to be installed on the host system to create a bootable disk image from the kernel file. This makes development on Windows or Mac more difficult.

Because of these drawbacks, we decided to not use GRUB or the Multiboot standard. However, we plan to add Multiboot support to our [bootimage](https://github.com/rust-osdev/bootimage) tool, so that it’s possible to load your kernel on a GRUB system too. If you’re interested in writing a Multiboot compliant kernel, check out the [first edition](https://os.phil-opp.com/edition-1/) of this blog series.

### UEFI

(We don’t provide UEFI support at the moment, but we would love to! If you’d like to help, please tell us in the [Github issue](https://github.com/phil-opp/blog_os/issues/349).)

## A Minimal Kernel

Now that we roughly know how a computer boots, it’s time to create our own minimal kernel. Our goal is to create a disk image that prints a “Hello World!” to the screen when booted. We do this by extending the previous post’s [freestanding Rust binary](https://os.phil-opp.com/freestanding-rust-binary/).

As you may remember, we built the freestanding binary through cargo, but depending on the operating system, we needed different entry point names and compile flags. That’s because cargo builds for the host system by default, i.e., the system you’re running on. This isn’t something we want for our kernel, because a kernel that runs on top of, e.g., Windows, does not make much sense. Instead, we want to compile for a clearly defined target system.

### Installing Rust Nightly

Rust has three release channels: stable, beta, and nightly. The Rust Book explains the difference between these channels really well, so take a minute and [check it out](https://doc.rust-lang.org/book/appendix-07-nightly-rust.html#choo-choo-release-channels-and-riding-the-trains). For building an operating system, we will need some experimental features that are only available on the nightly channel, so we need to install a nightly version of Rust.

To manage Rust installations, I highly recommend [rustup](https://www.rustup.rs/). It allows you to install nightly, beta, and stable compilers side-by-side and makes it easy to update them. With rustup, you can use a nightly compiler for the current directory by running rustup override set nightly. Alternatively, you can add a file called rust-toolchain with the content nightly to the project’s root directory. You can check that you have a nightly version installed by running rustc --version: The version number should contain -nightly at the end.

The nightly compiler allows us to opt-in to various experimental features by using so-called feature flags at the top of our file. For example, we could enable the experimental [asm! macro](https://doc.rust-lang.org/stable/reference/inline-assembly.html) for inline assembly by adding #![feature(asm)] to the top of our main.rs. Note that such experimental features are completely unstable, which means that future Rust versions might change or remove them without prior warning. For this reason, we will only use them if absolutely necessary.

### Target Specification

Cargo supports different target systems through the --target parameter. The target is described by a so-called [*target triple*](https://clang.llvm.org/docs/CrossCompilation.html#target-triple), which describes the CPU architecture, the vendor, the operating system, and the [ABI](https://stackoverflow.com/a/2456882). For example, the x86\_64-unknown-linux-gnu target triple describes a system with an x86\_64 CPU, no clear vendor, and a Linux operating system with the GNU ABI. Rust supports [many different target triples](https://forge.rust-lang.org/release/platform-support.html), including arm-linux-androideabi for Android or [wasm32-unknown-unknown for WebAssembly](https://www.hellorust.com/setup/wasm-target/).

For our target system, however, we require some special configuration parameters (e.g. no underlying OS), so none of the [existing target triples](https://forge.rust-lang.org/release/platform-support.html) fits. Fortunately, Rust allows us to define [our own target](https://doc.rust-lang.org/nightly/rustc/targets/custom.html) through a JSON file. For example, a JSON file that describes the x86\_64-unknown-linux-gnu target looks like this:

{

"llvm-target": "x86\_64-unknown-linux-gnu",

"data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",

"arch": "x86\_64",

"target-endian": "little",

"target-pointer-width": "64",

"target-c-int-width": "32",

"os": "linux",

"executables": true,

"linker-flavor": "gcc",

"pre-link-args": ["-m64"],

"morestack": false

}

Most fields are required by LLVM to generate code for that platform. For example, the [data-layout](https://llvm.org/docs/LangRef.html#data-layout) field defines the size of various integer, floating point, and pointer types. Then there are fields that Rust uses for conditional compilation, such as target-pointer-width. The third kind of field defines how the crate should be built. For example, the pre-link-args field specifies arguments passed to the [linker](https://en.wikipedia.org/wiki/Linker_(computing)).

We also target x86\_64 systems with our kernel, so our target specification will look very similar to the one above. Let’s start by creating an x86\_64-blog\_os.json file (choose any name you like) with the common content:

{

"llvm-target": "x86\_64-unknown-none",

"data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",

"arch": "x86\_64",

"target-endian": "little",

"target-pointer-width": "64",

"target-c-int-width": "32",

"os": "none",

"executables": true

}

Note that we changed the OS in the llvm-target and the os field to none, because we will run on bare metal.

We add the following build-related entries:

"linker-flavor": "ld.lld",

"linker": "rust-lld"

Instead of using the platform’s default linker (which might not support Linux targets), we use the cross-platform [LLD](https://lld.llvm.org/) linker that is shipped with Rust for linking our kernel.

"panic-strategy": "abort",

This setting specifies that the target doesn’t support [stack unwinding](https://www.bogotobogo.com/cplusplus/stackunwinding.php) on panic, so instead the program should abort directly. This has the same effect as the panic = "abort" option in our Cargo.toml, so we can remove it from there. (Note that, in contrast to the Cargo.toml option, this target option also applies when we recompile the core library later in this post. So, even if you prefer to keep the Cargo.toml option, make sure to include this option.)

"disable-redzone": true,

We’re writing a kernel, so we’ll need to handle interrupts at some point. To do that safely, we have to disable a certain stack pointer optimization called the “red zone”, because it would cause stack corruption otherwise. For more information, see our separate post about [disabling the red zone](https://os.phil-opp.com/red-zone/).

"features": "-mmx,-sse,+soft-float",

The features field enables/disables target features. We disable the mmx and sse features by prefixing them with a minus and enable the soft-float feature by prefixing it with a plus. Note that there must be no spaces between different flags, otherwise LLVM fails to interpret the features string.

The mmx and sse features determine support for [Single Instruction Multiple Data (SIMD)](https://en.wikipedia.org/wiki/SIMD) instructions, which can often speed up programs significantly. However, using the large SIMD registers in OS kernels leads to performance problems. The reason is that the kernel needs to restore all registers to their original state before continuing an interrupted program. This means that the kernel has to save the complete SIMD state to main memory on each system call or hardware interrupt. Since the SIMD state is very large (512–1600 bytes) and interrupts can occur very often, these additional save/restore operations considerably harm performance. To avoid this, we disable SIMD for our kernel (not for applications running on top!).

A problem with disabling SIMD is that floating point operations on x86\_64 require SIMD registers by default. To solve this problem, we add the soft-float feature, which emulates all floating point operations through software functions based on normal integers.

For more information, see our post on [disabling SIMD](https://os.phil-opp.com/disable-simd/).

#### Putting it Together

Our target specification file now looks like this:

{

"llvm-target": "x86\_64-unknown-none",

"data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",

"arch": "x86\_64",

"target-endian": "little",

"target-pointer-width": "64",

"target-c-int-width": "32",

"os": "none",

"executables": true,

"linker-flavor": "ld.lld",

"linker": "rust-lld",

"panic-strategy": "abort",

"disable-redzone": true,

"features": "-mmx,-sse,+soft-float"

}

### Building our Kernel

Compiling for our new target will use Linux conventions (I’m not quite sure why; I assume it’s just LLVM’s default). This means that we need an entry point named \_start as described in the [previous post](https://os.phil-opp.com/freestanding-rust-binary/):

// src/main.rs

#![no\_std] // don't link the Rust standard library

#![no\_main] // disable all Rust-level entry points

use core::panic::PanicInfo;

/// This function is called on panic.

#[panic\_handler]

fn panic(\_info: &PanicInfo) -> ! {

loop {}

}

#[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 {}

}

Note that the entry point needs to be called \_start regardless of your host OS.

We can now build the kernel for our new target by passing the name of the JSON file as --target

> cargo build --target x86\_64-blog\_os.json

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

It fails! The error tells us that the Rust compiler no longer finds the [core library](https://doc.rust-lang.org/nightly/core/index.html). This library contains basic Rust types such as Result, Option, and iterators, and is implicitly linked to all no\_std crates.

The problem is that the core library is distributed together with the Rust compiler as a precompiled library. So it is only valid for supported host triples (e.g., x86\_64-unknown-linux-gnu) but not for our custom target. If we want to compile code for other targets, we need to recompile core for these targets first.

#### The build-std Option

That’s where the [build-std feature](https://doc.rust-lang.org/nightly/cargo/reference/unstable.html#build-std) of cargo comes in. It allows to recompile core and other standard library crates on demand, instead of using the precompiled versions shipped with the Rust installation. This feature is very new and still not finished, so it is marked as “unstable” and only available on [nightly Rust compilers](https://os.phil-opp.com/minimal-rust-kernel/#installing-rust-nightly).

To use the feature, we need to create a [cargo configuration](https://doc.rust-lang.org/cargo/reference/config.html) file at .cargo/config.toml with the following content:

# in .cargo/config.toml

[unstable]

build-std = ["core", "compiler\_builtins"]

This tells cargo that it should recompile the core and compiler\_builtins libraries. The latter is required because it is a dependency of core. In order to recompile these libraries, cargo needs access to the rust source code, which we can install with rustup component add rust-src

**Note:** The unstable.build-std configuration key requires at least the Rust nightly from 2020-07-15.

After setting the unstable.build-std configuration key and installing the rust-src component, we can rerun our build command:

> cargo build --target x86\_64-blog\_os.json

Compiling core v0.0.0 (/…/rust/src/libcore)

Compiling rustc-std-workspace-core v1.99.0 (/…/rust/src/tools/rustc-std-workspace-core)

Compiling compiler\_builtins v0.1.32

Compiling blog\_os v0.1.0 (/…/blog\_os)

Finished dev [unoptimized + debuginfo] target(s) in 0.29 secs

We see that cargo build now recompiles the core, rustc-std-workspace-core (a dependency of compiler\_builtins), and compiler\_builtins libraries for our custom target.

#### Memory-Related Intrinsics

The Rust compiler assumes that a certain set of built-in functions is available for all systems. Most of these functions are provided by the compiler\_builtins crate that we just recompiled. However, there are some memory-related functions in that crate that are not enabled by default because they are normally provided by the C library on the system. These functions include memset, which sets all bytes in a memory block to a given value, memcpy, which copies one memory block to another, and memcmp, which compares two memory blocks. While we didn’t need any of these functions to compile our kernel right now, they will be required as soon as we add some more code to it (e.g. when copying structs around).

Since we can’t link to the C library of the operating system, we need an alternative way to provide these functions to the compiler. One possible approach for this could be to implement our own memset etc. functions and apply the #[no\_mangle] attribute to them (to avoid the automatic renaming during compilation). However, this is dangerous since the slightest mistake in the implementation of these functions could lead to undefined behavior. For example, implementing memcpy with a for loop may result in an infinite recursion because for loops implicitly call the [IntoIterator::into\_iter](https://doc.rust-lang.org/stable/core/iter/trait.IntoIterator.html#tymethod.into_iter) trait method, which may call memcpy again. So it’s a good idea to reuse existing, well-tested implementations instead.

Fortunately, the compiler\_builtins crate already contains implementations for all the needed functions, they are just disabled by default to not collide with the implementations from the C library. We can enable them by setting cargo’s [build-std-features](https://doc.rust-lang.org/nightly/cargo/reference/unstable.html#build-std-features) flag to ["compiler-builtins-mem"]. Like the build-std flag, this flag can be either passed on the command line as a -Z flag or configured in the unstable table in the .cargo/config.toml file. Since we always want to build with this flag, the config file option makes more sense for us:

# in .cargo/config.toml

[unstable]

build-std-features = ["compiler-builtins-mem"]

build-std = ["core", "compiler\_builtins"]

(Support for the compiler-builtins-mem feature was only [added very recently](https://github.com/rust-lang/rust/pull/77284), so you need at least Rust nightly 2020-09-30 for it.)

Behind the scenes, this flag enables the [mem feature](https://github.com/rust-lang/compiler-builtins/blob/eff506cd49b637f1ab5931625a33cef7e91fbbf6/Cargo.toml#L54-L55) of the compiler\_builtins crate. The effect of this is that the #[no\_mangle] attribute is applied to the [memcpy etc. implementations](https://github.com/rust-lang/compiler-builtins/blob/eff506cd49b637f1ab5931625a33cef7e91fbbf6/src/mem.rs#L12-L69) of the crate, which makes them available to the linker.

With this change, our kernel has valid implementations for all compiler-required functions, so it will continue to compile even if our code gets more complex.

#### [🔗](https://os.phil-opp.com/minimal-rust-kernel/#set-a-default-target)Set a Default Target

To avoid passing the --target parameter on every invocation of cargo build, we can override the default target. To do this, we add the following to our [cargo configuration](https://doc.rust-lang.org/cargo/reference/config.html) file at .cargo/config.toml:

# in .cargo/config.toml

[build]

target = "x86\_64-blog\_os.json"

This tells cargo to use our x86\_64-blog\_os.json target when no explicit --target argument is passed. This means that we can now build our kernel with a simple cargo build. For more information on cargo configuration options, check out the [official documentation](https://doc.rust-lang.org/cargo/reference/config.html).

We are now able to build our kernel for a bare metal target with a simple cargo build. However, our \_start entry point, which will be called by the boot loader, is still empty. It’s time that we output something to screen from it.

### Printing to Screen

The easiest way to print text to the screen at this stage is the [VGA text buffer](https://en.wikipedia.org/wiki/VGA-compatible_text_mode). It is a special memory area mapped to the VGA hardware that contains the contents displayed on screen. It normally consists of 25 lines that each contain 80 character cells. Each character cell displays an ASCII character with some foreground and background colors. The screen output looks like this:



We will discuss the exact layout of the VGA buffer in the next post, where we write a first small driver for it. For printing “Hello World!”, we just need to know that the buffer is located at address 0xb8000 and that each character cell consists of an ASCII byte and a color byte.

The implementation looks like this:

static HELLO: &[u8] = b"Hello World!";

#[no\_mangle]

pub extern "C" fn \_start() -> ! {

let vga\_buffer = 0xb8000 as \*mut u8;

for (i, &byte) in HELLO.iter().enumerate() {

unsafe {

\*vga\_buffer.offset(i as isize \* 2) = byte;

\*vga\_buffer.offset(i as isize \* 2 + 1) = 0xb;

}

}

loop {}

}

First, we cast the integer 0xb8000 into a [raw pointer](https://doc.rust-lang.org/stable/book/ch19-01-unsafe-rust.html#dereferencing-a-raw-pointer). Then we [iterate](https://doc.rust-lang.org/stable/book/ch13-02-iterators.html) over the bytes of the [static](https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#the-static-lifetime) HELLO [byte string](https://doc.rust-lang.org/reference/tokens.html#byte-string-literals). We use the [enumerate](https://doc.rust-lang.org/core/iter/trait.Iterator.html#method.enumerate) method to additionally get a running variable i. In the body of the for loop, we use the [offset](https://doc.rust-lang.org/std/primitive.pointer.html#method.offset) method to write the string byte and the corresponding color byte (0xb is a light cyan).

Note that there’s an [unsafe](https://doc.rust-lang.org/stable/book/ch19-01-unsafe-rust.html) block around all memory writes. The reason is that the Rust compiler can’t prove that the raw pointers we create are valid. They could point anywhere and lead to data corruption. By putting them into an unsafe block, we’re basically telling the compiler that we are absolutely sure that the operations are valid. Note that an unsafe block does not turn off Rust’s safety checks. It only allows you to do [five additional things](https://doc.rust-lang.org/stable/book/ch19-01-unsafe-rust.html#unsafe-superpowers).

I want to emphasize that **this is not the way we want to do things in Rust!** It’s very easy to mess up when working with raw pointers inside unsafe blocks. For example, we could easily write beyond the buffer’s end if we’re not careful.

So we want to minimize the use of unsafe as much as possible. Rust gives us the ability to do this by creating safe abstractions. For example, we could create a VGA buffer type that encapsulates all unsafety and ensures that it is impossible to do anything wrong from the outside. This way, we would only need minimal amounts of unsafe code and can be sure that we don’t violate [memory safety](https://en.wikipedia.org/wiki/Memory_safety). We will create such a safe VGA buffer abstraction in the next post.

## Running our Kernel

Now that we have an executable that does something perceptible, it is time to run it. First, we need to turn our compiled kernel into a bootable disk image by linking it with a bootloader. Then we can run the disk image in the [QEMU](https://www.qemu.org/) virtual machine or boot it on real hardware using a USB stick.

### Creating a Bootimage

To turn our compiled kernel into a bootable disk image, we need to link it with a bootloader. As we learned in the [section about booting](https://os.phil-opp.com/minimal-rust-kernel/#the-boot-process), the bootloader is responsible for initializing the CPU and loading our kernel.

Instead of writing our own bootloader, which is a project on its own, we use the [bootloader](https://crates.io/crates/bootloader) crate. This crate implements a basic BIOS bootloader without any C dependencies, just Rust and inline assembly. To use it for booting our kernel, we need to add a dependency on it:

# in Cargo.toml

[dependencies]

bootloader = "0.9.23"

Adding the bootloader as a dependency is not enough to actually create a bootable disk image. The problem is that we need to link our kernel with the bootloader after compilation, but cargo has no support for [post-build scripts](https://github.com/rust-lang/cargo/issues/545).

To solve this problem, we created a tool named bootimage that first compiles the kernel and bootloader, and then links them together to create a bootable disk image. To install the tool, execute the following command in your terminal:

cargo install bootimage

For running bootimage and building the bootloader, you need to have the llvm-tools-preview rustup component installed. You can do so by executing rustup component add llvm-tools-preview.

After installing bootimage and adding the llvm-tools-preview component, we can create a bootable disk image by executing

> cargo bootimage

We see that the tool recompiles our kernel using cargo build, so it will automatically pick up any changes you make. Afterwards, it compiles the bootloader, which might take a while. Like all crate dependencies, it is only built once and then cached, so subsequent builds will be much faster. Finally, bootimage combines the bootloader and your kernel into a bootable disk image.

After executing the command, you should see a bootable disk image named bootimage-blog\_os.bin in your target/x86\_64-blog\_os/debug directory. You can boot it in a virtual machine or copy it to a USB drive to boot it on real hardware. (Note that this is not a CD image, which has a different format, so burning it to a CD doesn’t work).

#### How does it work?

The bootimage tool performs the following steps behind the scenes:

* It compiles our kernel to an [ELF](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) file.
* It compiles the bootloader dependency as a standalone executable.
* It links the bytes of the kernel ELF file to the bootloader.

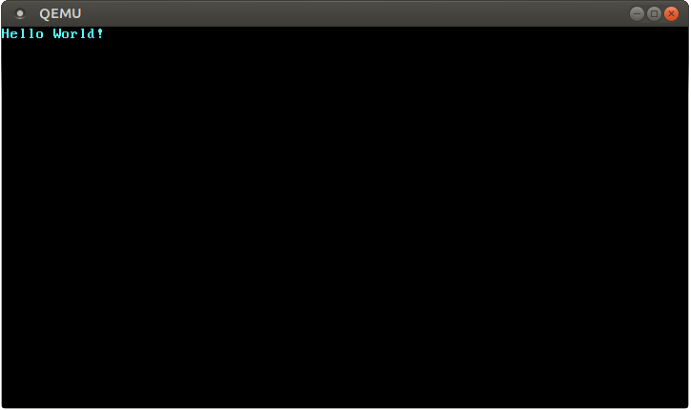
When booted, the bootloader reads and parses the appended ELF file. It then maps the program segments to virtual addresses in the page tables, zeroes the .bss section, and sets up a stack. Finally, it reads the entry point address (our \_start function) and jumps to it.

### Booting it in QEMU

We can now boot the disk image in a virtual machine. To boot it in [QEMU](https://www.qemu.org/), execute the following command:

> qemu-system-x86\_64 -drive format=raw,file=target/x86\_64-blog\_os/debug/bootimage-blog\_os.bin

This opens a separate window which should look similar to this:



We see that our “Hello World!” is visible on the screen.

### [🔗](https://os.phil-opp.com/minimal-rust-kernel/#real-machine)Real Machine

It is also possible to write it to a USB stick and boot it on a real machine, **but be careful** to choose the correct device name, because **everything on that device is overwritten**:

> dd if=target/x86\_64-blog\_os/debug/bootimage-blog\_os.bin of=/dev/sdX && sync

Where sdX is the device name of your USB stick.

After writing the image to the USB stick, you can run it on real hardware by booting from it. You probably need to use a special boot menu or change the boot order in your BIOS configuration to boot from the USB stick. Note that it currently doesn’t work for UEFI machines, since the bootloader crate has no UEFI support yet.

### Using cargo run

To make it easier to run our kernel in QEMU, we can set the runner configuration key for cargo:

# in .cargo/config.toml

[target.'cfg(target\_os = "none")']

runner = "bootimage runner"

The target.'cfg(target\_os = "none")' table applies to all targets whose target configuration file’s "os" field is set to "none". This includes our x86\_64-blog\_os.json target. The runner key specifies the command that should be invoked for cargo run. The command is run after a successful build with the executable path passed as the first argument. See the [cargo documentation](https://doc.rust-lang.org/cargo/reference/config.html) for more details.

The bootimage runner command is specifically designed to be usable as a runner executable. It links the given executable with the project’s bootloader dependency and then launches QEMU. See the [Readme of bootimage](https://github.com/rust-osdev/bootimage) for more details and possible configuration options.

Now we can use cargo run to compile our kernel and boot it in QEMU.

Jueves 8 de junio de 2023

