Writing an OS in Rust Philipp Oppermann's blog

Testing

Apr 27, 2019

Other Lang

- Chinese (s
- Japanese
- Persian
- Korean

This post explores unit and integration testing in `no_std` executables. We will use Rust's support for custom test frameworks to execute test functions inside our kernel. To report the results out of QEMU, we will use different features of QEMU and the `bootimage` tool.

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-04` branch.

Requirements

This post replaces the (now deprecated) *Unit Testing* and *Integration Tests* posts. It assumes that you have followed the *A Minimal Rust Kernel* post after 2019-04-27. Mainly, it requires that you have a `.cargo/config.toml` file that sets a default target and defines a runner executable.

Testing in Rust

Rust has a built-in test framework that is capable of running unit tests without the need to set anything up. Just create a function that checks some results through assertions and add the `#[test]` attribute to the function header. Then `cargo test` will automatically find and execute all test functions of your crate.

Unfortunately, it's a bit more complicated for `no_std` applications such as our kernel. The problem is that Rust's test framework implicitly uses the built-in `test` library, which depends on the standard library. This means that we can't use the default test framework for our `#[no_std]` kernel.

We can see this when we try to run `cargo test` in our project:

```
> cargo test
   Compiling blog_os v0.1.0 (/.../blog_os)
error[E0463]: can't find crate for `test`
```

Since the `test` crate depends on the standard library, it is not available for our bare metal target. While porting the `test` crate to a `#[no_std]` context is possible, it is highly unstable and requires some hacks, such as redefining the `panic` macro.

Custom Test Frameworks

Fortunately, Rust supports replacing the default test framework through the unstable `custom_test_frameworks` feature. This feature requires no external libraries and thus also works in `#[no_std]` environments. It works by collecting all functions annotated with a `#[test_case]` attribute and then invoking a user-specified runner function with the list of tests as an argument. Thus, it gives the implementation maximal control over the test process.

The disadvantage compared to the default test framework is that many advanced features, such as `should_panic` tests, are not available. Instead, it is up to the implementation to provide such features itself if needed. This is ideal for us since we have a very special execution environment where the default implementations of such advanced features probably wouldn't work anyway. For example, the `#[should_panic]` attribute relies on stack unwinding to catch the panics, which we disabled for our kernel.

To implement a custom test framework for our kernel, we add the following to our `main.rs` :

```
// in src/main.rs

#![feature(custom_test_frameworks)]
#![test_runner(crate::test_runner)]

#[cfg(test)]
pub fn test_runner(tests: &[&dyn Fn()]) {
    println!("Running {} tests", tests.len());
    for test in tests {
        test();
    }
}
```

Our runner just prints a short debug message and then calls each test function in the list. The argument type `&[&dyn Fn()]` is a *slice* of *trait object* references of the *Fn()* trait. It is basically a list of references to types that can be called like a function. Since the function is useless for non-test runs, we use the `#[cfg(test)]` attribute to include it only for tests.

When we run `cargo test` now, we see that it now succeeds (if it doesn't, see the note below). However, we still see our "Hello World" instead of the message from our `test_runner` . The reason is that our `_start` function is still used as entry point. The custom test frameworks feature generates a `main` function that calls `test_runner` , but this function is ignored because we use the `#[no_main]` attribute and provide our own entry point.

> **Note:** There is currently a bug in cargo that leads to "duplicate lang item" errors on `cargo test` in some cases. It occurs when you have set `panic = "abort"` for a profile in your `Cargo.toml` . Try removing it, then `cargo test` should work. Alternatively, if that doesn't work, then add `panic-abort-tests = true` to the `[unstable]` section of your `.cargo/config.toml` file. See the cargo issue for more information on this.

To fix this, we first need to change the name of the generated function to something different than `main` through the `reexport_test_harness_main` attribute. Then we can call the renamed function from our `_start` function:

```
// in src/main.rs

#![reexport_test_harness_main = "test_main"]

#[no_mangle]
pub extern "C" fn _start() -> ! {
    println!("Hello World{}", "!");

    #[cfg(test)]
    test_main();

    loop {}
}
```

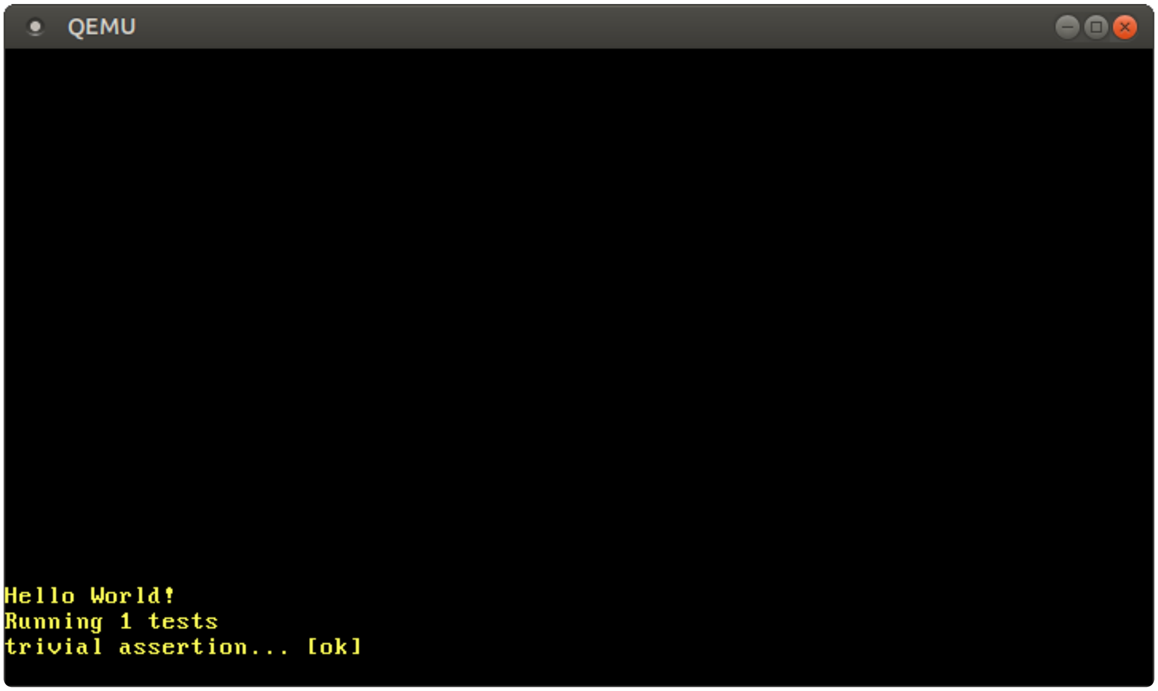
We set the name of the test framework entry function to `test_main` and call it from our `_start` entry point. We use conditional compilation to add the call to `test_main` only in test contexts because the function is not generated on a normal run.

When we now execute `cargo test`, we see the “Running 0 tests” message from our `test_runner` on the screen. We are now ready to create our first test function:

```
// in src/main.rs

#[test_case]
fn trivial_assertion() {
    print!("trivial assertion... ");
    assert_eq!(1, 1);
    println!("[ok]");
}
```

When we run `cargo test` now, we see the following output:



```
Hello World!
Running 1 tests
trivial assertion... [ok]
```

The `tests` slice passed to our `test_runner` function now contains a reference to the `trivial_assertion` function. From the `trivial assertion... [ok]` output on the screen, we see that the test was called and that it succeeded.

After executing the tests, our `test_runner` returns to the `test_main` function, which in turn returns to our `_start` entry point function. At the end of `_start`, we enter an endless loop because the entry point function is not allowed to return. This is a problem, because we want `cargo test` to exit after running all tests.

Exiting QEMU

Right now, we have an endless loop at the end of our `_start` function and need to close QEMU manually on each execution of `cargo test`. This is unfortunate because we also want to run `cargo test` in scripts without user interaction. The clean solution to this would be to implement a proper way to shutdown our OS. Unfortunately, this is relatively complex because it requires implementing support for either the APM or ACPI power management standard.

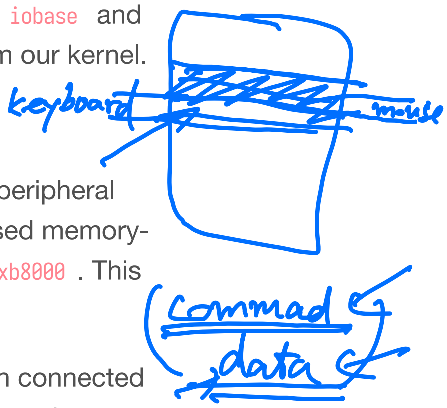
Luckily, there is an escape hatch: QEMU supports a special `isa-debug-exit` device, which provides an easy way to exit QEMU from the guest system. To enable it, we need to pass a `-device` argument to QEMU. We can do so by adding a `package.metadata.bootimage.test-args` configuration key in our `Cargo.toml`:

```toml
# in Cargo.toml

[package.metadata.bootimage]
test-args = ["-device", "isa-debug-exit,iobase=0xf4,iosize=0x04"]
```

The `bootimage runner` appends the `test-args` to the default QEMU command for all test executables. For a normal `cargo run`, the arguments are ignored.

Together with the device name (`isa-debug-exit`), we pass the two parameters `iobase` and `iosize` that specify the *I/O port* through which the device can be reached from our kernel.



I/O Ports

There are two different approaches for communicating between the CPU and peripheral hardware on x86, **memory-mapped I/O** and **port-mapped I/O**. We already used memory-mapped I/O for accessing the VGA text buffer through the memory address `0xb8000`. This address is not mapped to RAM but to some memory on the VGA device.

In contrast, port-mapped I/O uses a separate I/O bus for communication. Each connected peripheral has one or more port numbers. To communicate with such an I/O port, there are special CPU instructions called `in` and `out`, which take a port number and a data byte (there are also variations of these commands that allow sending a `u16` or `u32`).

The `isa-debug-exit` device uses port-mapped I/O. The `iobase` parameter specifies on which port address the device should live (`0xf4` is a generally unused port on the x86's IO bus) and the `iosize` specifies the port size (`0x04` means four bytes).

Using the Exit Device

The functionality of the `isa-debug-exit` device is very simple. When a `value` is written to the I/O port specified by `iobase`, it causes QEMU to exit with exit status `(value << 1) | 1`. So when we write `0` to the port, QEMU will exit with exit status `(0 << 1) | 1 = 1`, and when we write `1` to the port, it will exit with exit status `(1 << 1) | 1 = 3`.

Instead of manually invoking the `in` and `out` assembly instructions, we use the abstractions provided by the `x86_64` crate. To add a dependency on that crate, we add it to the `dependencies` section in our `Cargo.toml`:

```toml
# in Cargo.toml

[dependencies]
x86_64 = "0.14.2"
```

Now we can use the `Port` type provided by the crate to create an `exit_qemu` function:

```rust
// in src/main.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u32)]
```

```
pub enum QemuExitCode {
    Success = 0x10,
    Failed = 0x11,
}

pub fn exit_qemu(exit_code: QemuExitCode) {
    use x86_64::instructions::port::Port;

    unsafe {
        let mut port = Port::new(0xf4);
        port.write(exit_code as u32);
    }
}
```

The function creates a new `Port` at `0xf4`, which is the `iobase` of the `isa-debug-exit` device. Then it writes the passed exit code to the port. We use `u32` because we specified the `iosize` of the `isa-debug-exit` device as 4 bytes. Both operations are unsafe because writing to an I/O port can generally result in arbitrary behavior.

To specify the exit status, we create a `QemuExitCode` enum. The idea is to exit with the success exit code if all tests succeeded and with the failure exit code otherwise. The enum is marked as `#[repr(u32)]` to represent each variant by a `u32` integer. We use the exit code `0x10` for success and `0x11` for failure. The actual exit codes don't matter much, as long as they don't clash with the default exit codes of QEMU. For example, using exit code `0` for success is not a good idea because it becomes `(0 << 1) | 1 = 1` after the transformation, which is the default exit code when QEMU fails to run. So we could not differentiate a QEMU error from a successful test run.

We can now update our `test_runner` to exit QEMU after all tests have run:

```
// in src/main.rs

fn test_runner(tests: &[&dyn Fn()]) {
    println!("Running {} tests", tests.len());
    for test in tests {
        test();
    }
    /// new
    exit_qemu(QemuExitCode::Success);
}
```

When we run `cargo test` now, we see that QEMU immediately closes after executing the tests. The problem is that `cargo test` interprets the test as failed even though we passed our `Success` exit code:

```
> cargo test
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running target/x86_64-blog_os/debug/deps/blog_os-5804fc7d2dd4c9be
Building bootloader
   Compiling bootloader v0.5.3 (/home/philipp/Documents/bootloader)
    Finished release [optimized + debuginfo] target(s) in 1.07s
Running: `qemu-system-x86_64 -drive format=raw,file=/.../target/x86_64-blog_os/debug/
    deps/bootimage-blog_os-5804fc7d2dd4c9be.bin -device isa-debug-exit,iobase=0xf4,
    iosize=0x04`
```

```
error: test failed, to rerun pass '--bin blog_os'
```

The problem is that `cargo test` considers all error codes other than `0` as failure.

Success Exit Code

To work around this, `bootimage` provides a `test-success-exit-code` configuration key that maps a specified exit code to the exit code `0`:

```toml
# in Cargo.toml

[package.metadata.bootimage]
test-args = [...]
test-success-exit-code = 33         # (0x10 << 1) | 1
```

With this configuration, `bootimage` maps our success exit code to exit code 0, so that `cargo test` correctly recognizes the success case and does not count the test as failed.

Our test runner now automatically closes QEMU and correctly reports the test results. We still see the QEMU window open for a very short time, but it does not suffice to read the results. It would be nice if we could print the test results to the console instead, so we can still see them after QEMU exits.

Printing to the Console

To see the test output on the console, we need to send the data from our kernel to the host system somehow. There are various ways to achieve this, for example, by sending the data over a TCP network interface. However, setting up a networking stack is quite a complex task, so we will choose a simpler solution instead.

Serial Port

A simple way to send the data is to use the serial port, an old interface standard which is no longer found in modern computers. It is easy to program and QEMU can redirect the bytes sent over serial to the host's standard output or a file.

The chips implementing a serial interface are called UARTs. There are lots of UART models on x86, but fortunately the only differences between them are some advanced features we don't need. The common UARTs today are all compatible with the 16550 UART, so we will use that model for our testing framework.

We will use the `uart_16550` crate to initialize the UART and send data over the serial port. To add it as a dependency, we update our `Cargo.toml` and `main.rs`:

```toml
# in Cargo.toml

[dependencies]
uart_16550 = "0.2.0"
```

The `uart_16550` crate contains a `SerialPort` struct that represents the UART registers, but we still need to construct an instance of it ourselves. For that, we create a new `serial` module with the following content:

```rust
// in src/main.rs
```

```
mod serial;
```

```
// in src/serial.rs

use uart_16550::SerialPort;
use spin::Mutex;
use lazy_static::lazy_static;

lazy_static! {
    pub static ref SERIAL1: Mutex<SerialPort> = {
        let mut serial_port = unsafe { SerialPort::new(0x3F8) };
        serial_port.init();
        Mutex::new(serial_port)
    };
}
```

port address

Like with the VGA text buffer, we use `lazy_static` and a spinlock to create a `static` writer instance. By using `lazy_static` we can ensure that the `init` method is called exactly once on its first use.

Like the `isa-debug-exit` device, the UART is programmed using port I/O. Since the UART is more complex, it uses multiple I/O ports for programming different device registers. The unsafe `SerialPort::new` function expects the address of the first I/O port of the UART as an argument, from which it can calculate the addresses of all needed ports. We're passing the port address `0x3F8` which is the standard port number for the first serial interface.

To make the serial port easily usable, we add `serial_print!` and `serial_println!` macros:

```
// in src/serial.rs

#[doc(hidden)]
pub fn _print(args: ::core::fmt::Arguments) {
    use core::fmt::Write;
    SERIAL1.lock().write_fmt(args).expect("Printing to serial failed");
}

/// Prints to the host through the serial interface.
#[macro_export]
macro_rules! serial_print {
    ($($arg:tt)*) => {
        $crate::serial::_print(format_args!($($arg)*));
    };
}

/// Prints to the host through the serial interface, appending a newline.
#[macro_export]
macro_rules! serial_println {
    () => ($crate::serial_print!("\n"));
    ($fmt:expr) => ($crate::serial_print!(concat!($fmt, "\n")));
    ($fmt:expr, $($arg:tt)*) => ($crate::serial_print!(
        concat!($fmt, "\n"), $($arg)*));
}
```

The implementation is very similar to the implementation of our `print` and `println` macros.

Since the `SerialPort` type already implements the `fmt::Write` trait, we don't need to provide our own implementation.

Now we can print to the serial interface instead of the VGA text buffer in our test code:

```
// in src/main.rs

#[cfg(test)]
fn test_runner(tests: &[&dyn Fn()]) {
    serial_println!("Running {} tests", tests.len());
    […]
}

#[test_case]
fn trivial_assertion() {
    serial_print!("trivial assertion... ");
    assert_eq!(1, 1);
    serial_println!("[ok]");
}
```

Note that the `serial_println` macro lives directly under the root namespace because we used the `#[macro_export]` attribute, so importing it through `use crate::serial::serial_println` will not work.

QEMU Arguments

To see the serial output from QEMU, we need to use the `-serial` argument to redirect the output to stdout:

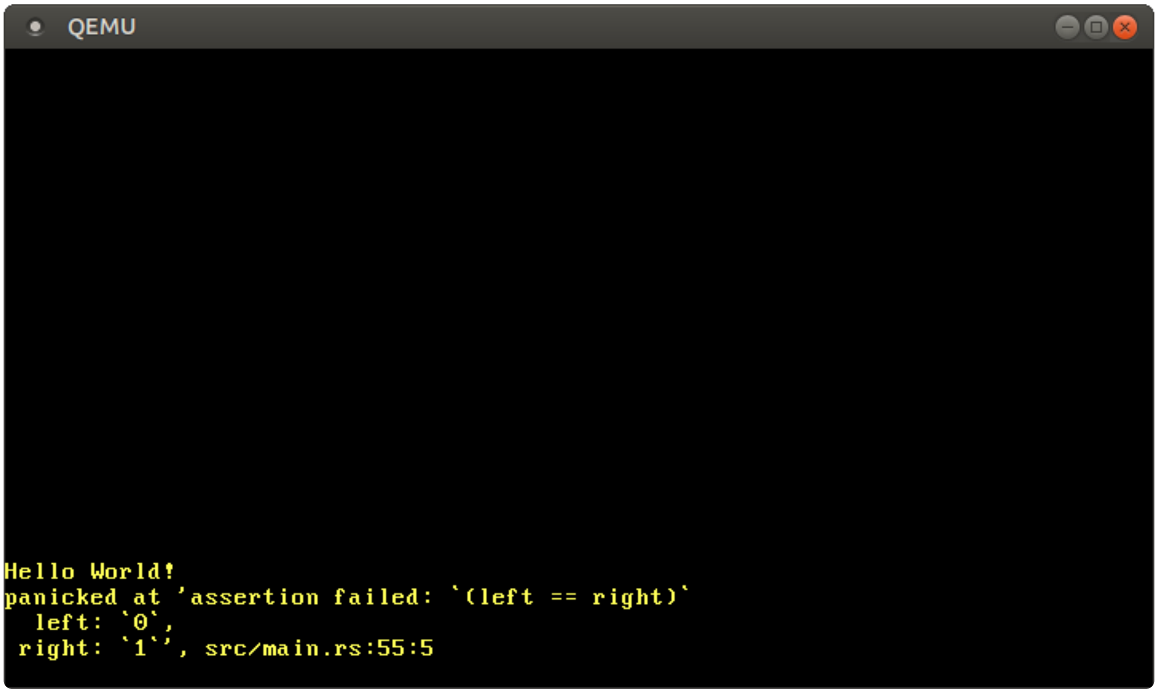
```
# in Cargo.toml

[package.metadata.bootimage]
test-args = [
    "-device", "isa-debug-exit,iobase=0xf4,iosize=0x04", "-serial", "stdio"
]
```

When we run `cargo test` now, we see the test output directly in the console:

```
> cargo test
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running target/x86_64-blog_os/debug/deps/blog_os-7b7c37b4ad62551a
Building bootloader
    Finished release [optimized + debuginfo] target(s) in 0.02s
Running: `qemu-system-x86_64 -drive format=raw,file=/…/target/x86_64-blog_os/debug/
    deps/bootimage-blog_os-7b7c37b4ad62551a.bin -device
    isa-debug-exit,iobase=0xf4,iosize=0x04 -serial stdio`
Running 1 tests
trivial assertion... [ok]
```

However, when a test fails, we still see the output inside QEMU because our panic handler still uses `println`. To simulate this, we can change the assertion in our `trivial_assertion` test to `assert_eq!(0, 1)`:



```
Hello World!
panicked at 'assertion failed: `(left == right)`
  left: `0`,
 right: `1`', src/main.rs:55:5
```

We see that the panic message is still printed to the VGA buffer, while the other test output is printed to the serial port. The panic message is quite useful, so it would be useful to see it in the console too.

Print an Error Message on Panic

To exit QEMU with an error message on a panic, we can use conditional compilation to use a different panic handler in testing mode:

```rust
// in src/main.rs

// our existing panic handler
#[cfg(not(test))] // new attribute
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    println!("{}", info);
    loop {}
}

// our panic handler in test mode
#[cfg(test)]
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    serial_println!("[failed]\n");
    serial_println!("Error: {}\n", info);
    exit_qemu(QemuExitCode::Failed);
    loop {}
}
```

For our test panic handler, we use `serial_println` instead of `println` and then exit QEMU with a failure exit code. Note that we still need an endless `loop` after the `exit_qemu` call because the compiler does not know that the `isa-debug-exit` device causes a program exit.

Now QEMU also exits for failed tests and prints a useful error message on the console:

```
> cargo test
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
     Running target/x86_64-blog_os/debug/deps/blog_os-7b7c37b4ad62551a
Building bootloader
    Finished release [optimized + debuginfo] target(s) in 0.02s
Running: `qemu-system-x86_64 -drive format=raw,file=/…/target/x86_64-blog_os/debug/
    deps/bootimage-blog_os-7b7c37b4ad62551a.bin -device
    isa-debug-exit,iobase=0xf4,iosize=0x04 -serial stdio`
Running 1 tests
trivial assertion... [failed]

Error: panicked at 'assertion failed: `(left == right)`
  left: `0`,
 right: `1`', src/main.rs:65:5
```

Since we see all test output on the console now, we no longer need the QEMU window that pops up for a short time. So we can hide it completely.

Hiding QEMU

Since we report out the complete test results using the `isa-debug-exit` device and the serial port, we don't need the QEMU window anymore. We can hide it by passing the `-display none` argument to QEMU:

```toml
# in Cargo.toml

[package.metadata.bootimage]
test-args = [
    "-device", "isa-debug-exit,iobase=0xf4,iosize=0x04", "-serial", "stdio",
    "-display", "none"
]
```

Now QEMU runs completely in the background and no window gets opened anymore. This is not only less annoying, but also allows our test framework to run in environments without a graphical user interface, such as CI services or SSH connections.

Timeouts

Since `cargo test` waits until the test runner exits, a test that never returns can block the test runner forever. That's unfortunate, but not a big problem in practice since it's usually easy to avoid endless loops. In our case, however, endless loops can occur in various situations:

- The bootloader fails to load our kernel, which causes the system to reboot endlessly.
- The BIOS/UEFI firmware fails to load the bootloader, which causes the same endless rebooting.
- The CPU enters a `loop {}` statement at the end of some of our functions, for example because the QEMU exit device doesn't work properly.
- The hardware causes a system reset, for example when a CPU exception is not caught (explained in a future post).

Since endless loops can occur in so many situations, the `bootimage` tool sets a timeout of 5 minutes for each test executable by default. If the test does not finish within this time, it is marked as failed and a "Timed Out" error is printed to the console. This feature ensures that tests that are stuck in an endless loop don't block `cargo test` forever.

You can try it yourself by adding a `loop {}` statement in the `trivial_assertion` test. When you run `cargo test`, you see that the test is marked as timed out after 5 minutes. The timeout duration is configurable through a `test-timeout` key in the Cargo.toml:

```
# in Cargo.toml

[package.metadata.bootimage]
test-timeout = 300          # (in seconds)
```

If you don't want to wait 5 minutes for the `trivial_assertion` test to time out, you can temporarily decrease the above value.

Insert Printing Automatically

Our `trivial_assertion` test currently needs to print its own status information using `serial_print!` / `serial_println!`:

```
#[test_case]
fn trivial_assertion() {
    serial_print!("trivial assertion... ");
    assert_eq!(1, 1);
    serial_println!("[ok]");
}
```

Manually adding these print statements for every test we write is cumbersome, so let's update our `test_runner` to print these messages automatically. To do that, we need to create a new `Testable` trait:

```
// in src/main.rs

pub trait Testable {
    fn run(&self) -> ();
}
```

The trick now is to implement this trait for all types `T` that implement the `Fn()` trait:

```
// in src/main.rs

impl<T> Testable for T
where
    T: Fn(),
{
    fn run(&self) {
        serial_print!("{}...\t", core::any::type_name::<T>());
        self();
        serial_println!("[ok]");
    }
}
```

We implement the `run` function by first printing the function name using the `any::type_name` function. This function is implemented directly in the compiler and returns a string description of every type. For functions, the type is their name, so this is exactly what we want in this case. The `\t` character is the tab character, which adds some alignment to the `[ok]` messages.

After printing the function name, we invoke the test function through `self()` . This only works because we require that `self` implements the `Fn()` trait. After the test function returns, we print `[ok]` to indicate that the function did not panic.

The last step is to update our `test_runner` to use the new `Testable` trait:

```
// in src/main.rs

#[cfg(test)]
pub fn test_runner(tests: &[&dyn Testable]) { // new
    serial_println!("Running {} tests", tests.len());
    for test in tests {
        test.run(); // new
    }
    exit_qemu(QemuExitCode::Success);
}
```

The only two changes are the type of the `tests` argument from `&[&dyn Fn()]` to `&[&dyn Testable]` and the fact that we now call `test.run()` instead of `test()` .

We can now remove the print statements from our `trivial_assertion` test since they're now printed automatically:

```
// in src/main.rs

#[test_case]
fn trivial_assertion() {
    assert_eq!(1, 1);
}
```

The `cargo test` output now looks like this:

```
Running 1 tests
blog_os::trivial_assertion...   [ok]
```

The function name now includes the full path to the function, which is useful when test functions in different modules have the same name. Otherwise, the output looks the same as before, but we no longer need to add print statements to our tests manually.

Testing the VGA Buffer

Now that we have a working test framework, we can create a few tests for our VGA buffer implementation. First, we create a very simple test to verify that `println` works without panicking:

```
// in src/vga_buffer.rs

#[test_case]
fn test_println_simple() {
    println!("test_println_simple output");
}
```

The test just prints something to the VGA buffer. If it finishes without panicking, it means that the `println` invocation did not panic either.

To ensure that no panic occurs even if many lines are printed and lines are shifted off the screen, we can create another test:

```
// in src/vga_buffer.rs

#[test_case]
fn test_println_many() {
    for _ in 0..200 {
        println!("test_println_many output");
    }
}
```

We can also create a test function to verify that the printed lines really appear on the screen:

```
// in src/vga_buffer.rs

#[test_case]
fn test_println_output() {
    let s = "Some test string that fits on a single line";
    println!("{}", s);
    for (i, c) in s.chars().enumerate() {
        let screen_char = WRITER.lock().buffer.chars[BUFFER_HEIGHT - 2][i].read();
        assert_eq!(char::from(screen_char.ascii_character), c);
    }
}
```

The function defines a test string, prints it using `println`, and then iterates over the screen characters of the static `WRITER`, which represents the VGA text buffer. Since `println` prints to the last screen line and then immediately appends a newline, the string should appear on line `BUFFER_HEIGHT - 2`.

By using `enumerate`, we count the number of iterations in the variable `i`, which we then use for loading the screen character corresponding to `c`. By comparing the `ascii_character` of the screen character with `c`, we ensure that each character of the string really appears in the VGA text buffer.

As you can imagine, we could create many more test functions. For example, a function that tests that no panic occurs when printing very long lines and that they're wrapped correctly, or a function for testing that newlines, non-printable characters, and non-unicode characters are handled correctly.

For the rest of this post, however, we will explain how to create *integration tests* to test the interaction of different components together.

Integration Tests

The convention for integration tests in Rust is to put them into a `tests` directory in the project root (i.e., next to the `src` directory). Both the default test framework and custom test frameworks will automatically pick up and execute all tests in that directory.

All integration tests are their own executables and completely separate from our `main.rs`. This means that each test needs to define its own entry point function. Let's create an example integration test named `basic_boot` to see how it works in detail:

```rust
// in tests/basic_boot.rs

#![no_std]
#![no_main]
#![feature(custom_test_frameworks)]
#![test_runner(crate::test_runner)]
#![reexport_test_harness_main = "test_main"]

use core::panic::PanicInfo;

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() -> ! {
    test_main();

    loop {}
}

fn test_runner(tests: &[&dyn Fn()]) {
    unimplemented!();
}

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

Since integration tests are separate executables, we need to provide all the crate attributes (`no_std` , `no_main` , `test_runner` , etc.) again. We also need to create a new entry point function `_start` , which calls the test entry point function `test_main` . We don't need any `cfg(test)` attributes because integration test executables are never built in non-test mode.

We use the `unimplemented` macro that always panics as a placeholder for the `test_runner` function and just `loop` in the `panic` handler for now. Ideally, we want to implement these functions exactly as we did in our `main.rs` using the `serial_println` macro and the `exit_qemu` function. The problem is that we don't have access to these functions since tests are built completely separately from our `main.rs` executable.

If you run `cargo test` at this stage, you will get an endless loop because the panic handler loops endlessly. You need to use the `ctrl+c` keyboard shortcut for exiting QEMU.

Create a Library

To make the required functions available to our integration test, we need to split off a library from our `main.rs` , which can be included by other crates and integration test executables. To do this, we create a new `src/lib.rs` file:

```rust
// src/lib.rs

#![no_std]
```

Like the `main.rs` , the `lib.rs` is a special file that is automatically recognized by cargo. The library is a separate compilation unit, so we need to specify the `#![no_std]` attribute again.

To make our library work with `cargo test`, we need to also move the test functions and attributes from `main.rs` to `lib.rs`:

```rust
// in src/lib.rs

#![cfg_attr(test, no_main)]
#![feature(custom_test_frameworks)]
#![test_runner(crate::test_runner)]
#![reexport_test_harness_main = "test_main"]

use core::panic::PanicInfo;

pub trait Testable {
    fn run(&self) -> ();
}

impl<T> Testable for T
where
    T: Fn(),
{
    fn run(&self) {
        serial_print!("{}...\t", core::any::type_name::<T>());
        self();
        serial_println!("[ok]");
    }
}

pub fn test_runner(tests: &[&dyn Testable]) {
    serial_println!("Running {} tests", tests.len());
    for test in tests {
        test.run();
    }
    exit_qemu(QemuExitCode::Success);
}

pub fn test_panic_handler(info: &PanicInfo) -> ! {
    serial_println!("[failed]\n");
    serial_println!("Error: {}\n", info);
    exit_qemu(QemuExitCode::Failed);
    loop {}
}

/// Entry point for `cargo test`
#[cfg(test)]
#[no_mangle]
pub extern "C" fn _start() -> ! {
    test_main();
    loop {}
}

#[cfg(test)]
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    test_panic_handler(info)
}
```

To make our `test_runner` available to executables and integration tests, we make it public and don't apply the `cfg(test)` attribute to it. We also factor out the implementation of our panic handler into a public `test_panic_handler` function, so that it is available for executables too.

Since our `lib.rs` is tested independently of our `main.rs` , we need to add a `_start` entry point and a panic handler when the library is compiled in test mode. By using the `cfg_attr` crate attribute, we conditionally enable the `no_main` attribute in this case.

We also move over the `QemuExitCode` enum and the `exit_qemu` function and make them public:

```
// in src/lib.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u32)]
pub enum QemuExitCode {
    Success = 0x10,
    Failed = 0x11,
}

pub fn exit_qemu(exit_code: QemuExitCode) {
    use x86_64::instructions::port::Port;

    unsafe {
        let mut port = Port::new(0xf4);
        port.write(exit_code as u32);
    }
}
```

Now executables and integration tests can import these functions from the library and don't need to define their own implementations. To also make `println` and `serial_println` available, we move the module declarations too:

```
// in src/lib.rs

pub mod serial;
pub mod vga_buffer;
```

We make the modules public to make them usable outside of our library. This is also required for making our `println` and `serial_println` macros usable since they use the `_print` functions of the modules.

Now we can update our `main.rs` to use the library:

```
// in src/main.rs

#![no_std]
#![no_main]
#![feature(custom_test_frameworks)]
#![test_runner(blog_os::test_runner)]
#![reexport_test_harness_main = "test_main"]

use core::panic::PanicInfo;
use blog_os::println;

#[no_mangle]
```

```
pub extern "C" fn _start() -> ! {
    println!("Hello World{}", "!");

    #[cfg(test)]
    test_main();

    loop {}
}

/// This function is called on panic.
#[cfg(not(test))]
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    println!("{}", info);
    loop {}
}

#[cfg(test)]
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    blog_os::test_panic_handler(info)
}
```

The library is usable like a normal external crate. It is called `blog_os` , like our crate. The above code uses the `blog_os::test_runner` function in the `test_runner` attribute and the `blog_os::test_panic_handler` function in our `cfg(test)` panic handler. It also imports the `println` macro to make it available to our `_start` and `panic` functions.

At this point, `cargo run` and `cargo test` should work again. Of course, `cargo test` still loops endlessly (you can exit with `ctrl+c`). Let's fix this by using the required library functions in our integration test.

Completing the Integration Test

Like our `src/main.rs` , our `tests/basic_boot.rs` executable can import types from our new library. This allows us to import the missing components to complete our test:

```
// in tests/basic_boot.rs

#![test_runner(blog_os::test_runner)]

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    blog_os::test_panic_handler(info)
}
```

Instead of reimplementing the test runner, we use the `test_runner` function from our library by changing the `#![test_runner(crate::test_runner)]` attribute to `#![test_runner(blog_os::test_runner)]` . We then don't need the `test_runner` stub function in `basic_boot.rs` anymore, so we can remove it. For our `panic` handler, we call the `blog_os::test_panic_handler` function like we did in our `main.rs` .

Now `cargo test` exits normally again. When you run it, you will see that it builds and runs the tests for our `lib.rs` , `main.rs` , and `basic_boot.rs` separately after each other. For the `main.rs` and the `basic_boot` integration tests, it reports “Running 0 tests” since these files don’t have any functions annotated with `#[test_case]`.

We can now add tests to our `basic_boot.rs`. For example, we can test that `println` works without panicking, like we did in the VGA buffer tests:

```rust
// in tests/basic_boot.rs

use blog_os::println;

#[test_case]
fn test_println() {
    println!("test_println output");
}
```

When we run `cargo test` now, we see that it finds and executes the test function.

The test might seem a bit useless right now since it’s almost identical to one of the VGA buffer tests. However, in the future, the `_start` functions of our `main.rs` and `lib.rs` might grow and call various initialization routines before running the `test_main` function, so that the two tests are executed in very different environments.

By testing `println` in a `basic_boot` environment without calling any initialization routines in `_start`, we can ensure that `println` works right after booting. This is important because we rely on it, e.g., for printing panic messages.

Future Tests

The power of integration tests is that they’re treated as completely separate executables. This gives them complete control over the environment, which makes it possible to test that the code interacts correctly with the CPU or hardware devices.

Our `basic_boot` test is a very simple example of an integration test. In the future, our kernel will become much more featureful and interact with the hardware in various ways. By adding integration tests, we can ensure that these interactions work (and keep working) as expected. Some ideas for possible future tests are:

- **CPU Exceptions**: When the code performs invalid operations (e.g., divides by zero), the CPU throws an exception. The kernel can register handler functions for such exceptions. An integration test could verify that the correct exception handler is called when a CPU exception occurs or that the execution continues correctly after a resolvable exception.
- **Page Tables**: Page tables define which memory regions are valid and accessible. By modifying the page tables, it is possible to allocate new memory regions, for example when launching programs. An integration test could modify the page tables in the `_start` function and verify that the modifications have the desired effects in `#[test_case]` functions.
- **Userspace Programs**: Userspace programs are programs with limited access to the system’s resources. For example, they don’t have access to kernel data structures or to the memory of other programs. An integration test could launch userspace programs that perform forbidden operations and verify that the kernel prevents them all.

As you can imagine, many more tests are possible. By adding such tests, we can ensure that we don’t break them accidentally when we add new features to our kernel or refactor our code.

This is especially important when our kernel becomes larger and more complex.

Tests that Should Panic

The test framework of the standard library supports a `#[should_panic]` attribute that allows constructing tests that should fail. This is useful, for example, to verify that a function fails when an invalid argument is passed. Unfortunately, this attribute isn't supported in `#[no_std]` crates since it requires support from the standard library.

While we can't use the `#[should_panic]` attribute in our kernel, we can get similar behavior by creating an integration test that exits with a success error code from the panic handler. Let's start creating such a test with the name `should_panic`:

```
// in tests/should_panic.rs

#![no_std]
#![no_main]

use core::panic::PanicInfo;
use blog_os::{QemuExitCode, exit_qemu, serial_println};

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    serial_println!("[ok]");
    exit_qemu(QemuExitCode::Success);
    loop {}
}
```

This test is still incomplete as it doesn't define a `_start` function or any of the custom test runner attributes yet. Let's add the missing parts:

```
// in tests/should_panic.rs

#![feature(custom_test_frameworks)]
#![test_runner(test_runner)]
#![reexport_test_harness_main = "test_main"]

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

    loop {}
}

pub fn test_runner(tests: &[&dyn Fn()]) {
    serial_println!("Running {} tests", tests.len());
    for test in tests {
        test();
        serial_println!("[test did not panic]");
        exit_qemu(QemuExitCode::Failed);
    }
    exit_qemu(QemuExitCode::Success);
}
```

Instead of reusing the `test_runner` from our `lib.rs`, the test defines its own `test_runner` function that exits with a failure exit code when a test returns without panicking (we want our tests to panic). If no test function is defined, the runner exits with a success error code. Since the runner always exits after running a single test, it does not make sense to define more than one `#[test_case]` function.

Now we can create a test that should fail:

```rust
// in tests/should_panic.rs

use blog_os::serial_print;

#[test_case]
fn should_fail() {
    serial_print!("should_panic::should_fail...\t");
    assert_eq!(0, 1);
}
```

The test uses `assert_eq` to assert that `0` and `1` are equal. Of course, this fails, so our test panics as desired. Note that we need to manually print the function name using `serial_print!` here because we don't use the `Testable` trait.

When we run the test through `cargo test --test should_panic` we see that it is successful because the test panicked as expected. When we comment out the assertion and run the test again, we see that it indeed fails with the *"test did not panic"* message.

A significant drawback of this approach is that it only works for a single test function. With multiple `#[test_case]` functions, only the first function is executed because the execution cannot continue after the panic handler has been called. I currently don't know of a good way to solve this problem, so let me know if you have an idea!

No Harness Tests

For integration tests that only have a single test function (like our `should_panic` test), the test runner isn't really needed. For cases like this, we can disable the test runner completely and run our test directly in the `_start` function.

The key to this is to disable the `harness` flag for the test in the `Cargo.toml`, which defines whether a test runner is used for an integration test. When it's set to `false`, both the default test runner and the custom test runner feature are disabled, so that the test is treated like a normal executable.

Let's disable the `harness` flag for our `should_panic` test:

```toml
# in Cargo.toml

[[test]]
name = "should_panic"
harness = false
```

Now we vastly simplify our `should_panic` test by removing the `test_runner`-related code. The result looks like this:

```rust
// in tests/should_panic.rs
```

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

use core::panic::PanicInfo;
use blog_os::{exit_qemu, serial_print, serial_println, QemuExitCode};

#[no_mangle]
pub extern "C" fn _start() -> ! {
    should_fail();
    serial_println!("[test did not panic]");
    exit_qemu(QemuExitCode::Failed);
    loop{}
}

fn should_fail() {
    serial_print!("should_panic::should_fail...\t");
    assert_eq!(0, 1);
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    serial_println!("[ok]");
    exit_qemu(QemuExitCode::Success);
    loop {}
}
```

We now call the `should_fail` function directly from our `_start` function and exit with a failure exit code if it returns. When we run `cargo test --test should_panic` now, we see that the test behaves exactly as before.

Apart from creating `should_panic` tests, disabling the `harness` attribute can also be useful for complex integration tests, for example, when the individual test functions have side effects and need to be run in a specified order.

Summary

Testing is a very useful technique to ensure that certain components have the desired behavior. Even if they cannot show the absence of bugs, they're still a useful tool for finding them and especially for avoiding regressions.

This post explained how to set up a test framework for our Rust kernel. We used Rust's custom test frameworks feature to implement support for a simple `#[test_case]` attribute in our bare-metal environment. Using the `isa-debug-exit` device of QEMU, our test runner can exit QEMU after running the tests and report the test status. To print error messages to the console instead of the VGA buffer, we created a basic driver for the serial port.

After creating some tests for our `println` macro, we explored integration tests in the second half of the post. We learned that they live in the `tests` directory and are treated as completely separate executables. To give them access to the `exit_qemu` function and the `serial_println` macro, we moved most of our code into a library that can be imported by all executables and integration tests. Since integration tests run in their own separate environment, they make it possible to test interactions with the hardware or to create tests that should panic.

We now have a test framework that runs in a realistic environment inside QEMU. By creating more tests in future posts, we can keep our kernel maintainable when it becomes more complex.

What’s next?

In the next post, we will explore *CPU exceptions*. These exceptions are thrown by the CPU when something illegal happens, such as a division by zero or an access to an unmapped memory page (a so-called “page fault”). Being able to catch and examine these exceptions is very important for debugging future errors. Exception handling is also very similar to the handling of hardware interrupts, which is required for keyboard support.

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!

« VGA Text Mode

CPU Exceptions »

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.

Loading comments...

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