Writing an OS in Rust Philipp Oppermann's blog

« All Posts

Double Faults

Jun 18, 2018

This post explores the double fault exception in detail, which occurs when the CPU fails to invoke an exception handler. By handling this exception we avoid fatal *triple faults* that cause a system reset. To prevent triple faults in all cases we also set up an *Interrupt Stack Table* to catch double faults on a separate kernel stack.

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-06` branch.

▶ **Table of Contents**

What is a Double Fault?

In simplified terms, a double fault is a special exception that occurs when the CPU fails to invoke an exception handler. For example, it occurs when a page fault is triggered but there is no page fault handler registered in the Interrupt Descriptor Table (IDT). So it's kind of similar to catch-all blocks in programming languages with exceptions, e.g. `catch(...)` in C++ or `catch(Exception e)` in Java or C#.

A double fault behaves like a normal exception. It has the vector number `8` and we can define a normal handler function for it in the IDT. It is really important to provide a double fault handler, because if a double fault is unhandled a fatal *triple fault* occurs. Triple faults can't be caught and most hardware reacts with a system reset.

Triggering a Double Fault

Let's provoke a double fault by triggering an exception for that we didn't define a handler function:

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

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

    blog_os::init();
    // trigger a page fault
    unsafe {
        *(0xdeadbeef as *mut u64) = 42;
    };

    // as before
    #[cfg(test)]
    test_main();

    println!("It did not crash!");
    loop {}
}
```

We use `unsafe` to write to the invalid address `0xdeadbeef`. The virtual address is not mapped to a physical address in the page tables, so a page fault occurs. We haven't registered a page fault handler in our IDT, so a double fault occurs.

When we start our kernel now, we see that it enters an endless boot loop. The reason for the boot loop is the following:

1. The CPU tries to write to `0xdeadbeef`, which causes a page fault.
2. The CPU looks at the corresponding entry in the IDT and sees that no handler function is specified. Thus, it can't call the page fault handler and a double fault occurs.
3. The CPU looks at the IDT entry of the double fault handler, but this entry does not specify a handler function either. Thus, a *triple* fault occurs.
4. A triple fault is fatal. QEMU reacts to it like most real hardware and issues a system reset.

So in order to prevent this triple fault, we need to either provide a handler function for page faults or a double fault handler. We want to avoid triple faults in all cases, so let's start with a double fault handler that is invoked for all unhandled exception types.

A Double Fault Handler

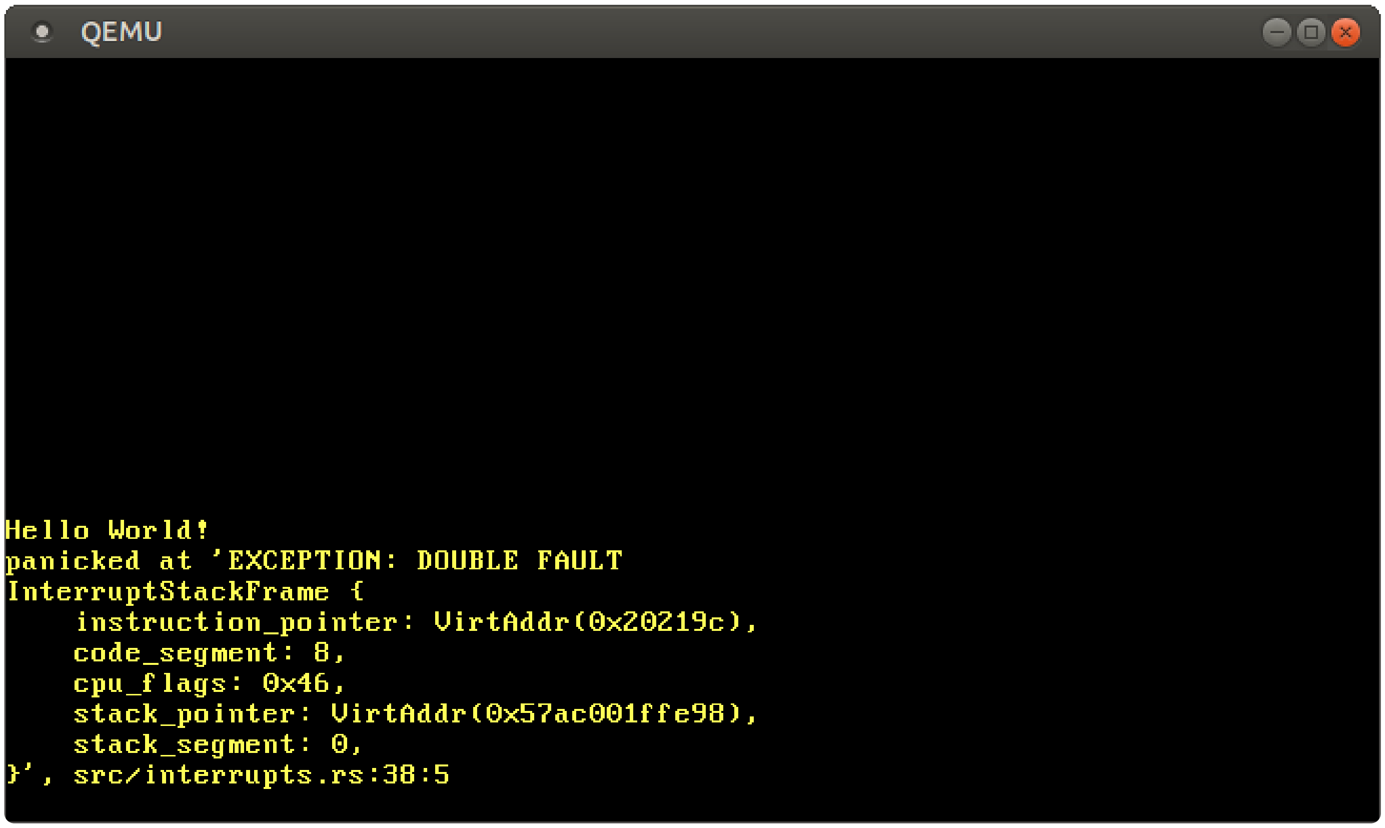
A double fault is a normal exception with an error code, so we can specify a handler function similar to our breakpoint handler:

```
// in src/interrupts.rs

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint.set_handler_fn(breakpoint_handler);
        idt.double_fault.set_handler_fn(double_fault_handler); // new
        idt
    };
}
```

```
// new
extern "x86-interrupt" fn double_fault_handler(
    stack_frame: InterruptStackFrame, _error_code: u64) -> !
{
    panic!("EXCEPTION: DOUBLE FAULT\n{:#?}", stack_frame);
}
```

Our handler prints a short error message and dumps the exception stack frame. The error code of the double fault handler is always zero, so there's no reason to print it. One difference to the breakpoint handler is that the double fault handler is *diverging*. The reason is that the `x86_64` architecture does not permit returning from a double fault exception.

When we start our kernel now, we should see that the double fault handler is invoked:



```
Hello World!
panicked at 'EXCEPTION: DOUBLE FAULT
InterruptStackFrame {
    instruction_pointer: VirtAddr(0x20219c),
    code_segment: 8,
    cpu_flags: 0x46,
    stack_pointer: VirtAddr(0x57ac001ffe98),
    stack_segment: 0,
}', src/interrupts.rs:38:5
```

It worked! Here is what happens this time:

1. The CPU tries to write to `0xdeadbeef` , which causes a page fault.
2. Like before, the CPU looks at the corresponding entry in the IDT and sees that no handler function is defined. Thus, a double fault occurs.
3. The CPU jumps to the – now present – double fault handler.

The triple fault (and the boot-loop) no longer occurs, since the CPU can now call the double fault handler.

That was quite straightforward! So why do we need a whole post for this topic? Well, we're now able to catch *most* double faults, but there are some cases where our current approach doesn't suffice.

Causes of Double Faults

Before we look at the special cases, we need to know the exact causes of double faults. Above, we used a pretty vague definition:

> A double fault is a special exception that occurs when the CPU fails to invoke an exception handler.

What does *"fails to invoke"* mean exactly? The handler is not present? The handler is swapped out? And what happens if a handler causes exceptions itself?

For example, what happens if:

1. a breakpoint exception occurs, but the corresponding handler function is swapped out?
2. a page fault occurs, but the page fault handler is swapped out?
3. a divide-by-zero handler causes a breakpoint exception, but the breakpoint handler is swapped out?
4. our kernel overflows its stack and the *guard page* is hit?

Fortunately, the AMD64 manual (PDF) has an exact definition (in Section 8.2.9). According to it, a "double fault exception *can* occur when a second exception occurs during the handling of a prior (first) exception handler". The *"can"* is important: Only very specific combinations of exceptions lead to a double fault. These combinations are:

First Exception	Second Exception
Divide-by-zero, Invalid TSS, Segment Not Present, Stack-Segment Fault, General Protection Fault	Invalid TSS, Segment Not Present, Stack-Segment Fault, General Protection Fault
Page Fault	Page Fault, Invalid TSS, Segment Not Present, Stack-Segment Fault, General Protection Fault

So for example a divide-by-zero fault followed by a page fault is fine (the page fault handler is invoked), but a divide-by-zero fault followed by a general-protection fault leads to a double fault.

With the help of this table, we can answer the first three of the above questions:

1. If a breakpoint exception occurs and the corresponding handler function is swapped out, a *page fault* occurs and the *page fault handler* is invoked.
2. If a page fault occurs and the page fault handler is swapped out, a *double fault* occurs and the *double fault handler* is invoked.
3. If a divide-by-zero handler causes a breakpoint exception, the CPU tries to invoke the breakpoint handler. If the breakpoint handler is swapped out, a *page fault* occurs and the *page fault handler* is invoked.

In fact, even the case of an exception without a handler function in the IDT follows this scheme: When the exception occurs, the CPU tries to read the corresponding IDT entry. Since the entry is 0, which is not a valid IDT entry, a *general protection fault* occurs. We did not define a handler function for the general protection fault either, so another general protection fault occurs. According to the table, this leads to a double fault.

Kernel Stack Overflow

Let's look at the fourth question:

> What happens if our kernel overflows its stack and the guard page is hit?

A guard page is a special memory page at the bottom of a stack that makes it possible to detect stack overflows. The page is not mapped to any physical frame, so accessing it causes a page fault instead of silently corrupting other memory. The bootloader sets up a guard page for our kernel stack, so a stack overflow causes a *page fault*.

When a page fault occurs the CPU looks up the page fault handler in the IDT and tries to push the interrupt stack frame onto the stack. However, the current stack pointer still points to the non-present guard page. Thus, a second page fault occurs, which causes a double fault (according to the above table).

So the CPU tries to call the *double fault handler* now. However, on a double fault the CPU tries to push the exception stack frame, too. The stack pointer still points to the guard page, so a *third* page fault occurs, which causes a *triple fault* and a system reboot. So our current double fault handler can't avoid a triple fault in this case.

Let's try it ourselves! We can easily provoke a kernel stack overflow by calling a function that recurses endlessly:

```
// in src/main.rs

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() -> ! {
    println!("Hello World{}", "!");
    blog_os::init();

    fn stack_overflow() {
        stack_overflow(); // for each recursion, the return address is pushed
    }

    // trigger a stack overflow
    stack_overflow();

    […] // test_main(), println(…), and loop {}
}
```

When we try this code in QEMU, we see that the system enters a boot-loop again.

So how can we avoid this problem? We can't omit the pushing of the exception stack frame, since the CPU itself does it. So we need to ensure somehow that the stack is always valid when a double fault exception occurs. Fortunately, the x86_64 architecture has a solution to this problem.

Switching Stacks

The x86_64 architecture is able to switch to a predefined, known-good stack when an exception occurs. This switch happens at hardware level, so it can be performed before the CPU pushes the exception stack frame.

The switching mechanism is implemented as an *Interrupt Stack Table* (IST). The IST is a table of 7 pointers to known-good stacks. In Rust-like pseudo code:

```
struct InterruptStackTable {
    stack_pointers: [Option<StackPointer>; 7],
}
```

For each exception handler, we can choose a stack from the IST through the `stack_pointers` field in the corresponding IDT entry. For example, we could use the first stack in the IST for our double fault handler. Then the CPU would automatically switch to this stack whenever a double fault occurs. This switch would happen before anything is pushed, so it would prevent the triple fault.

The IST and TSS

The Interrupt Stack Table (IST) is part of an old legacy structure called *Task State Segment* (TSS). The TSS used to hold various information (e.g. processor register state) about a task in 32-bit mode and was for example used for hardware context switching. However, hardware context switching is no longer supported in 64-bit mode and the format of the TSS changed completely.

On x86_64, the TSS no longer holds any task specific information at all. Instead, it holds two stack tables (the IST is one of them). The only common field between the 32-bit and 64-bit TSS is the pointer to the I/O port permissions bitmap.

The 64-bit TSS has the following format:

Field	Type
(reserved)	`u32`
Privilege Stack Table	`[u64; 3]`
(reserved)	`u64`
Interrupt Stack Table	`[u64; 7]`
(reserved)	`u64`
(reserved)	`u16`
I/O Map Base Address	`u16`

The *Privilege Stack Table* is used by the CPU when the privilege level changes. For example, if an exception occurs while the CPU is in user mode (privilege level 3), the CPU normally switches to kernel mode (privilege level 0) before invoking the exception handler. In that case, the CPU would switch to the 0th stack in the Privilege Stack Table (since 0 is the target privilege level). We don't have any user mode programs yet, so we ignore this table for now.

Creating a TSS

Let's create a new TSS that contains a separate double fault stack in its interrupt stack table. For that we need a TSS struct. Fortunately, the `x86_64` crate already contains a `TaskStateSegment` struct that we can use.

We create the TSS in a new `gdt` module (the name will make sense later):

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

pub mod gdt;

// in src/gdt.rs

use x86_64::VirtAddr;
use x86_64::structures::tss::TaskStateSegment;
use lazy_static::lazy_static;

pub const DOUBLE_FAULT_IST_INDEX: u16 = 0;

lazy_static! {
    static ref TSS: TaskStateSegment = {
        let mut tss = TaskStateSegment::new();
        tss.interrupt_stack_table[DOUBLE_FAULT_IST_INDEX as usize] = {
            const STACK_SIZE: usize = 4096 * 5;
            static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];

            let stack_start = VirtAddr::from_ptr(unsafe { &STACK });
            let stack_end = stack_start + STACK_SIZE;
            stack_end
        };
        tss
    };
}
```

We use `lazy_static` because Rust's const evaluator is not yet powerful enough to do this initialization at compile time. We define that the 0th IST entry is the double fault stack (any other IST index would work too). Then we write the top address of a double fault stack to the 0th entry. We write the top address because stacks on x86 grow downwards, i.e. from high addresses to low addresses.

We haven't implemented memory management yet, so we don't have a proper way to allocate a new stack. Instead, we use a `static mut` array as stack storage for now. The `unsafe` is required because the compiler can't guarantee race freedom when mutable statics are accessed. It is important that it is a `static mut` and not an immutable `static`, because otherwise the bootloader will map it to a read-only page. We will replace this with a proper stack allocation in a later post, then the `unsafe` will be no longer needed at this place.

Note that this double fault stack has no guard page that protects against stack overflow. This means that we should not do anything stack intensive in our double fault handler because a stack overflow might corrupt the memory below the stack.

Loading the TSS

Now that we created a new TSS, we need a way to tell the CPU that it should use it. Unfortunately this is a bit cumbersome, since the TSS uses the segmentation system (for historical reasons). Instead of loading the table directly, we need to add a new segment descriptor to the Global Descriptor Table (GDT). Then we can load our TSS invoking the `ltr` instruction with the respective GDT index. (This is the reason why we named our module `gdt`.)

The Global Descriptor Table

The Global Descriptor Table (GDT) is a relict that was used for memory segmentation before paging became the de facto standard. It is still needed in 64-bit mode for various things such as kernel/user mode configuration or TSS loading.

The GDT is a structure that contains the *segments* of the program. It was used on older architectures to isolate programs from each other, before paging became the standard. For more information about segmentation check out the equally named chapter of the free “Three Easy Pieces” book. While segmentation is no longer supported in 64-bit mode, the GDT still exists. It is mostly used for two things: Switching between kernel space and user space, and loading a TSS structure.

Creating a GDT

Let's create a static `GDT` that includes a segment for our `TSS` static:

```rust
// in src/gdt.rs

use x86_64::structures::gdt::{GlobalDescriptorTable, Descriptor};

lazy_static! {
    static ref GDT: GlobalDescriptorTable = {
        let mut gdt = GlobalDescriptorTable::new();
        gdt.add_entry(Descriptor::kernel_code_segment());
        gdt.add_entry(Descriptor::tss_segment(&TSS));
        gdt
    };
}
```

As before, we use `lazy_static` again. We create a new GDT with a code segment and a TSS segment.

Loading the GDT

To load our GDT we create a new `gdt::init` function, that we call from our `init` function:

```rust
// in src/gdt.rs

pub fn init() {
    GDT.load();
}

// in src/lib.rs

pub fn init() {
    gdt::init();
    interrupts::init_idt();
}
```

Now our GDT is loaded (since the `_start` function calls `init`), but we still see the boot loop on stack overflow.

The final Steps

The problem is that the GDT segments are not yet active because the segment and TSS registers still contain the values from the old GDT. We also need to modify the double fault IDT entry so that it uses the new stack.

In summary, we need to do the following:

1. **Reload code segment register**: We changed our GDT, so we should reload `cs`, the code segment register. This is required since the old segment selector could point to a different GDT descriptor now (e.g. a TSS descriptor).
2. **Load the TSS** : We loaded a GDT that contains a TSS selector, but we still need to tell the CPU that it should use that TSS.
3. **Update the IDT entry**: As soon as our TSS is loaded, the CPU has access to a valid interrupt stack table (IST). Then we can tell the CPU that it should use our new double fault stack by modifying our double fault IDT entry.

For the first two steps, we need access to the `code_selector` and `tss_selector` variables in our `gdt::init` function. We can achieve this by making them part of the static through a new `Selectors` struct:

```
// in src/gdt.rs

use x86_64::structures::gdt::SegmentSelector;

lazy_static! {
    static ref GDT: (GlobalDescriptorTable, Selectors) = {
        let mut gdt = GlobalDescriptorTable::new();
        let code_selector = gdt.add_entry(Descriptor::kernel_code_segment());
        let tss_selector = gdt.add_entry(Descriptor::tss_segment(&TSS));
        (gdt, Selectors { code_selector, tss_selector })
    };
}

struct Selectors {
    code_selector: SegmentSelector,
    tss_selector: SegmentSelector,
}
```

Now we can use the selectors to reload the `cs` segment register and load our `TSS` :

```
// in src/gdt.rs

pub fn init() {
    use x86_64::instructions::segmentation::set_cs;
    use x86_64::instructions::tables::load_tss;

    GDT.0.load();
    unsafe {
        set_cs(GDT.1.code_selector);
        load_tss(GDT.1.tss_selector);
    }
}
```

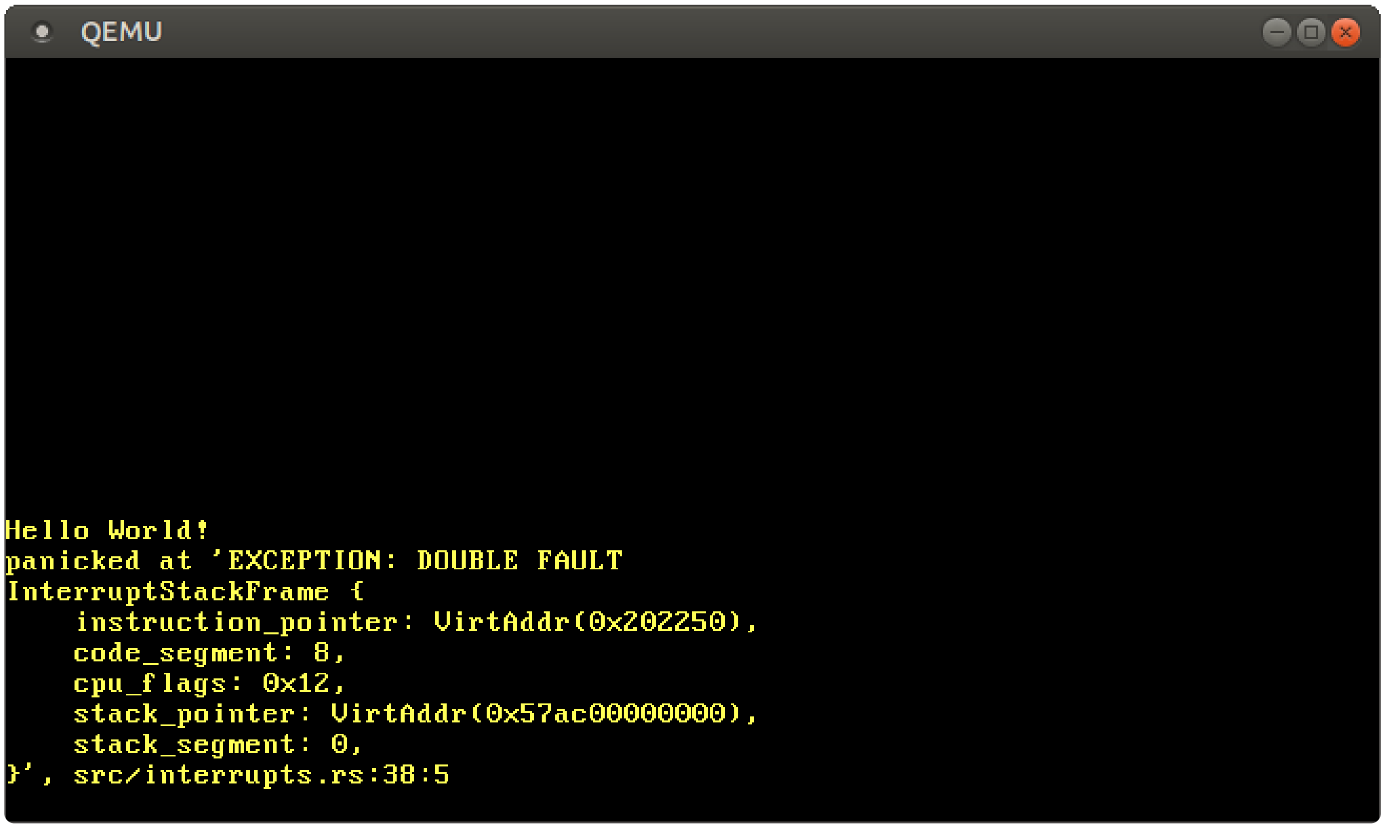
We reload the code segment register using `set_cs` and load the TSS using `load_tss`. The functions are marked as `unsafe`, so we need an `unsafe` block to invoke them. The reason is that it might be possible to break memory safety by loading invalid selectors.

Now that we loaded a valid TSS and interrupt stack table, we can set the stack index for our double fault handler in the IDT:

```
// in src/interrupts.rs

use crate::gdt;

lazy_static! {
    static ref IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        idt.breakpoint.set_handler_fn(breakpoint_handler);
        unsafe {
            idt.double_fault.set_handler_fn(double_fault_handler)
                .set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX); // new
        }

        idt
    };
}
```

The `set_stack_index` method is unsafe because the the caller must ensure that the used index is valid and not already used for another exception.

That's it! Now the CPU should switch to the double fault stack whenever a double fault occurs. Thus, we are able to catch *all* double faults, including kernel stack overflows:



```
Hello World!
panicked at 'EXCEPTION: DOUBLE FAULT
InterruptStackFrame {
    instruction_pointer: VirtAddr(0x202250),
    code_segment: 8,
    cpu_flags: 0x12,
    stack_pointer: VirtAddr(0x57ac00000000),
    stack_segment: 0,
}', src/interrupts.rs:38:5
```

From now on we should never see a triple fault again! To ensure that we don't accidentally break the above, we should add a test for this.

A Stack Overflow Test

To test our new `gdt` module and ensure that the double fault handler is correctly called on a stack overflow, we can add an integration test. The idea is to do provoke a double fault in the test function and verify that the double fault handler is called.

Let's start with a minimal skeleton:

```
// in tests/stack_overflow.rs

#![no_std]
#![no_main]

use core::panic::PanicInfo;

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

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

Like our `panic_handler` test, the test will run without a test harness. The reason is that we can't continue execution after a double fault, so more than one test doesn't make sense. To disable the test harness for the test, we add the following to our `Cargo.toml` :

```
# in Cargo.toml

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

Now `cargo test --test stack_overflow` should compile successfully. The test fails of course, since the `unimplemented` macro panics.

Implementing `_start`

The implementation of the `_start` function looks like this:

```
// in tests/stack_overflow.rs

use blog_os::serial_print;

#[no_mangle]
pub extern "C" fn _start() -> ! {
    serial_print!("stack_overflow::stack_overflow...\t");

    blog_os::gdt::init();
    init_test_idt();

    // trigger a stack overflow
    stack_overflow();

    panic!("Execution continued after stack overflow");
}

#[allow(unconditional_recursion)]
fn stack_overflow() {
    stack_overflow(); // for each recursion, the return address is pushed
    volatile::Volatile::new(0).read(); // prevent tail recursion optimizations
}
```

We call our `gdt::init` function to initialize a new GDT. Instead of calling our `interrupts::init_idt` function, we call a `init_test_idt` function that will be explained in a moment. The reason is that we want to register a custom double fault handler that does a `exit_qemu(QemuExitCode::Success)` instead of panicking.

The `stack_overflow` function is almost identical to the function in our `main.rs` . The only difference is that we do an additional volatile read at the end of the function using the `Volatile` type to prevent a compiler optimization called *tail call elimination*. Among other things, this optimization allows the compiler to transform a function whose last statement is a recursive function call into a normal loop. Thus, no additional stack frame is created for the function call, so that the stack usage does remain constant.

In our case, however, we want that the stack overflow happens, so we add a dummy volatile read statement at the end of the function, which the compiler is not allowed to remove. Thus, the function is no longer *tail recursive* and the transformation into a loop is prevented. We also add the `allow(unconditional_recursion)` attribute to silence the compiler warning that the function recurses endlessly.

The Test IDT

As noted above, the test needs its own IDT with a custom double fault handler. The implementation looks like this:

```
// in tests/stack_overflow.rs

use lazy_static::lazy_static;
use x86_64::structures::idt::InterruptDescriptorTable;

lazy_static! {
    static ref TEST_IDT: InterruptDescriptorTable = {
        let mut idt = InterruptDescriptorTable::new();
        unsafe {
            idt.double_fault
                .set_handler_fn(test_double_fault_handler)
                .set_stack_index(blog_os::gdt::DOUBLE_FAULT_IST_INDEX);
        }

        idt
    };
}

pub fn init_test_idt() {
    TEST_IDT.load();
}
```

The implementation is very similar to our normal IDT in `interrupts.rs`. Like in the normal IDT, we set a stack index into the IST for the double fault handler in order to switch to a separate stack. The `init_test_idt` function loads the IDT on the CPU through the `load` method.

The Double Fault Handler

The only missing piece is our double fault handler. It looks like this:

```
// in tests/stack_overflow.rs

use blog_os::{exit_qemu, QemuExitCode, serial_println};
use x86_64::structures::idt::InterruptStackFrame;

extern "x86-interrupt" fn test_double_fault_handler(
    _stack_frame: InterruptStackFrame,
    _error_code: u64,
) -> ! {
    serial_println!("[ok]");
    exit_qemu(QemuExitCode::Success);
    loop {}
}
```

When the double fault handler is called, we exit QEMU with a success exit code, which marks the test as passed. Since integration tests are completely separate executables, we need to set `#![feature(abi_x86_interrupt)]` attribute again at the top of our test file.

Now we can run our test through `cargo test --test stack_overflow` (or `cargo test` to run all tests). As expected, we see the `stack_overflow... [ok]` output in the console. Try to comment out the `set_stack_index` line: it should cause the test to fail.

Summary

In this post we learned what a double fault is and under which conditions it occurs. We added a basic double fault handler that prints an error message and added an integration test for it.

We also enabled the hardware supported stack switching on double fault exceptions so that it also works on stack overflow. While implementing it, we learned about the task state segment (TSS), the contained interrupt stack table (IST), and the global descriptor table (GDT), which was used for segmentation on older architectures.

What's next?

The next post explains how to handle interrupts from external devices such as timers, keyboards, or network controllers. These hardware interrupts are very similar to exceptions, e.g. they are also dispatched through the IDT. However, unlike exceptions, they don't arise directly on the CPU. Instead, an *interrupt controller* aggregates these interrupts and forwards them to CPU depending on their priority. In the next post we will explore the Intel 8259 (“PIC”) interrupt controller and learn how to implement 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*, since they don't charge any fees. If you prefer other platforms, I also have *Patreon* and *Donorbox* accounts. The latter is the most flexible as it supports multiple currencies and one-time contributions.

Thank you!

« CPU Exceptions Hardware Interrupts »

0 reactions

0 comments

Write Preview

Sign in to comment

Styling with Markdown is supported Sign in with GitHub

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

Other Languages

- Persian
- Japanese

About Me

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

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