CPU Exceptions

Jun 17, 2018

CPU exceptions occur in various erroneous situations, for example when accessing an invalid memory address or when dividing by zero. To react to them we have to set up an *interrupt descriptor table* that provides handler functions. At the end of this post, our kernel will be able to catch breakpoint exceptions and to resume normal execution afterwards.

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-05` branch.

▶ **Table of Contents**

Overview

An exception signals that something is wrong with the current instruction. For example, the CPU issues an exception if the current instruction tries to divide by 0. When an exception occurs, the CPU interrupts its current work and immediately calls a specific exception handler function, depending on the exception type.

On x86 there are about 20 different CPU exception types. The most important are:

- **Page Fault**: A page fault occurs on illegal memory accesses. For example, if the current instruction tries to read from an unmapped page or tries to write to a read-only page.
- **Invalid Opcode**: This exception occurs when the current instruction is invalid, for example when we try to use newer SSE instructions on an old CPU that does not support them.
- **General Protection Fault**: This is the exception with the broadest range of causes. It occurs on various kinds of access violations such as trying to execute a privileged instruction in user level code or writing reserved fields in configuration registers.
- **Double Fault**: When an exception occurs, the CPU tries to call the corresponding handler function. If another exception occurs *while calling the exception handler*, the CPU raises a double fault exception. This exception also occurs when there is no handler function registered for an exception.
- **Triple Fault**: If an exception occurs while the CPU tries to call the double fault handler function, it issues a fatal *triple fault*. We can't catch or handle a triple fault. Most processors react by resetting themselves and rebooting the operating system.

For the full list of exceptions check out the OSDev wiki.

The Interrupt Descriptor Table

In order to catch and handle exceptions, we have to set up a so-called *Interrupt Descriptor Table* (IDT). In this table we can specify a handler function for each CPU exception. The hardware uses this table directly, so we need to follow a predefined format. Each entry must have the following 16-byte structure:

Type	Name	Description
u16	Function Pointer [0:15]	The lower bits of the pointer to the handler function.
u16	GDT selector	Selector of a code segment in the global descriptor table.
u16	Options	(see below)
u16	Function Pointer [16:31]	The middle bits of the pointer to the handler function.
u32	Function Pointer [32:63]	The remaining bits of the pointer to the handler function.
u32	Reserved	

The options field has the following format:

Bits	Name	Description
0-2	Interrupt Stack Table Index	0: Don't switch stacks, 1-7: Switch to the n-th stack in the Interrupt Stack Table when this handler is called.
3-7	Reserved	
8	0: Interrupt Gate, 1: Trap Gate	If this bit is 0, interrupts are disabled when this handler is called.
9-11	must be one	
12	must be zero	
13-14	Descriptor Privilege Level (DPL)	The minimal privilege level required for calling this handler.
15	Present	

Each exception has a predefined IDT index. For example the invalid opcode exception has table index 6 and the page fault exception has table index 14. Thus, the hardware can automatically load the corresponding IDT entry for each exception. The Exception Table in the OSDev wiki shows the IDT indexes of all exceptions in the “Vector nr.” column.

When an exception occurs, the CPU roughly does the following:

1. Push some registers on the stack, including the instruction pointer and the RFLAGS register. (We will use these values later in this post.)
2. Read the corresponding entry from the Interrupt Descriptor Table (IDT). For example, the CPU reads the 14-th entry when a page fault occurs.
3. Check if the entry is present. Raise a double fault if not.
4. Disable hardware interrupts if the entry is an interrupt gate (bit 40 not set).
5. Load the specified GDT selector into the CS segment.
6. Jump to the specified handler function.

Don't worry about steps 4 and 5 for now, we will learn about the global descriptor table and hardware interrupts in future posts.

An IDT Type

Instead of creating our own IDT type, we will use the `InterruptDescriptorTable` struct of the `x86_64` crate, which looks like this:

```
#[repr(C)]
pub struct InterruptDescriptorTable {
    pub divide_by_zero: Entry<HandlerFunc>,
    pub debug: Entry<HandlerFunc>,
    pub non_maskable_interrupt: Entry<HandlerFunc>,
    pub breakpoint: Entry<HandlerFunc>,
    pub overflow: Entry<HandlerFunc>,
    pub bound_range_exceeded: Entry<HandlerFunc>,
    pub invalid_opcode: Entry<HandlerFunc>,
    pub device_not_available: Entry<HandlerFunc>,
    pub double_fault: Entry<HandlerFuncWithErrCode>,
    pub invalid_tss: Entry<HandlerFuncWithErrCode>,
    pub segment_not_present: Entry<HandlerFuncWithErrCode>,
    pub stack_segment_fault: Entry<HandlerFuncWithErrCode>,
    pub general_protection_fault: Entry<HandlerFuncWithErrCode>,
    pub page_fault: Entry<PageFaultHandlerFunc>,
    pub x87_floating_point: Entry<HandlerFunc>,
    pub alignment_check: Entry<HandlerFuncWithErrCode>,
    pub machine_check: Entry<HandlerFunc>,
    pub simd_floating_point: Entry<HandlerFunc>,
    pub virtualization: Entry<HandlerFunc>,
    pub security_exception: Entry<HandlerFuncWithErrCode>,
    // some fields omitted
}
```

The fields have the type `idt::Entry<F>`, which is a struct that represents the fields of an IDT entry (see the table above). The type parameter `F` defines the expected handler function type. We see that some entries require a `HandlerFunc` and some entries require a

`HandlerFuncWithErrCode` . The page fault even has its own special type: `PageFaultHandlerFunc` .

Let's look at the `HandlerFunc` type first:

```
type HandlerFunc = extern "x86-interrupt" fn(_: InterruptStackFrame);
```

It's a type alias for an `extern "x86-interrupt" fn` type. The `extern` keyword defines a function with a foreign calling convention and is often used to communicate with C code (`extern "C" fn`). But what is the `x86-interrupt` calling convention?

The Interrupt Calling Convention

Exceptions are quite similar to function calls: The CPU jumps to the first instruction of the called function and executes it. Afterwards the CPU jumps to the return address and continues the execution of the parent function.

However, there is a major difference between exceptions and function calls: A function call is invoked voluntary by a compiler inserted `call` instruction, while an exception might occur at *any* instruction. In order to understand the consequences of this difference, we need to examine function calls in more detail.

Calling conventions specify the details of a function call. For example, they specify where function parameters are placed (e.g. in registers or on the stack) and how results are returned. On x86_64 Linux, the following rules apply for C functions (specified in the System V ABI):

- the first six integer arguments are passed in registers `rdi` , `rsi` , `rdx` , `rcx` , `r8` , `r9`
- additional arguments are passed on the stack
- results are returned in `rax` and `rdx`

Note that Rust does not follow the C ABI (in fact, there isn't even a Rust ABI yet), so these rules apply only to functions declared as `extern "C" fn` .

Preserved and Scratch Registers

The calling convention divides the registers in two parts: *preserved* and *scratch* registers.

The values of *preserved* registers must remain unchanged across function calls. So a called function (the *"callee"*) is only allowed to overwrite these registers if it restores their original values before returning. Therefore these registers are called *"callee-saved"*. A common pattern is to save these registers to the stack at the function's beginning and restore them just before returning.

In contrast, a called function is allowed to overwrite *scratch* registers without restrictions. If the caller wants to preserve the value of a scratch register across a function call, it needs to backup and restore it before the function call (e.g. by pushing it to the stack). So the scratch registers are *caller-saved*.

On x86_64, the C calling convention specifies the following preserved and scratch registers:

preserved registers	scratch registers
`rbp` , `rbx` , `rsp` , `r12` , `r13` , `r14` , `r15`	`rax` , `rcx` , `rdx` , `rsi` , `rdi` , `r8` , `r9` , `r10` , `r11`
callee-saved	*caller-saved*

The compiler knows these rules, so it generates the code accordingly. For example, most functions begin with a `push rbp` , which backups `rbp` on the stack (because it's a callee-saved register).

Preserving all Registers

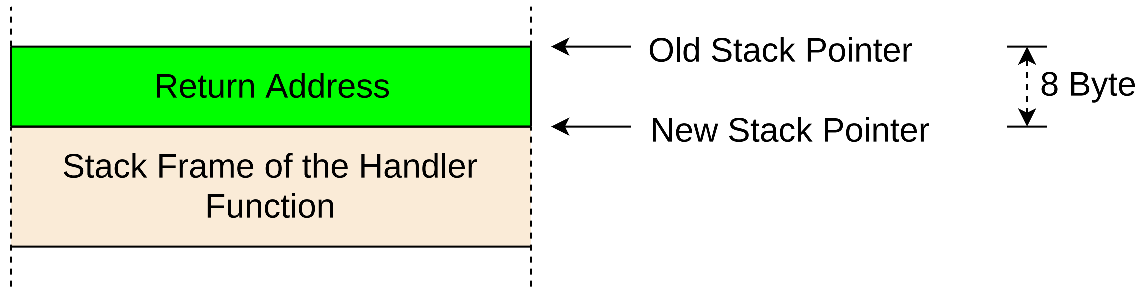
In contrast to function calls, exceptions can occur on *any* instruction. In most cases we don't even know at compile time if the generated code will cause an exception. For example, the compiler can't know if an instruction causes a stack overflow or a page fault.

Since we don't know when an exception occurs, we can't backup any registers before. This means that we can't use a calling convention that relies on caller-saved registers for exception handlers. Instead, we need a calling convention means that preserves *all registers*. The `x86-interrupt` calling convention is such a calling convention, so it guarantees that all register values are restored to their original values on function return.

Note that this does not mean that all registers are saved to the stack at function entry. Instead, the compiler only backs up the registers that are overwritten by the function. This way, very efficient code can be generated for short functions that only use a few registers.

The Interrupt Stack Frame

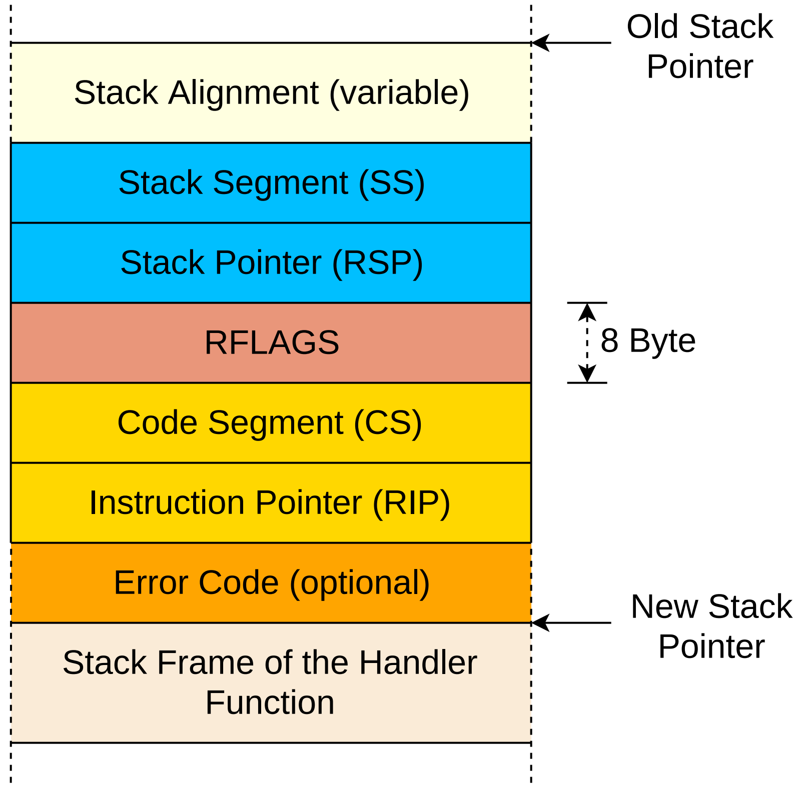
On a normal function call (using the `call` instruction), the CPU pushes the return address before jumping to the target function. On function return (using the `ret` instruction), the CPU pops this return address and jumps to it. So the stack frame of a normal function call looks like this:



For exception and interrupt handlers, however, pushing a return address would not suffice, since interrupt handlers often run in a different context (stack pointer, CPU flags, etc.). Instead, the CPU performs the following steps when an interrupt occurs:

1. **Aligning the stack pointer**: An interrupt can occur at any instructions, so the stack pointer can have any value, too. However, some CPU instructions (e.g. some SSE instructions) require that the stack pointer is aligned on a 16 byte boundary, therefore the CPU performs such an alignment right after the interrupt.
2. **Switching stacks** (in some cases): A stack switch occurs when the CPU privilege level changes, for example when a CPU exception occurs in an user mode program. It is also possible to configure stack switches for specific interrupts using the so-called *Interrupt Stack Table* (described in the next post).
3. **Pushing the old stack pointer**: The CPU pushes the values of the stack pointer (`rsp`) and the stack segment (`ss`) registers at the time when the interrupt occurred (before the alignment). This makes it possible to restore the original stack pointer when returning from an interrupt handler.
4. **Pushing and updating the `RFLAGS` register**: The `RFLAGS` register contains various control and status bits. On interrupt entry, the CPU changes some bits and pushes the old value.
5. **Pushing the instruction pointer**: Before jumping to the interrupt handler function, the CPU pushes the instruction pointer (`rip`) and the code segment (`cs`). This is comparable to the return address push of a normal function call.
6. **Pushing an error code** (for some exceptions): For some specific exceptions such as page faults, the CPU pushes an error code, which describes the cause of the exception.
7. **Invoking the interrupt handler**: The CPU reads the address and the segment descriptor of the interrupt handler function from the corresponding field in the IDT. It then invokes this handler by loading the values into the `rip` and `cs` registers.

So the *interrupt stack frame* looks like this:



In the `x86_64` crate, the interrupt stack frame is represented by the `InterruptStackFrame` struct. It is passed to interrupt handlers as `&mut` and can be used to retrieve additional information about the exception's cause. The struct contains no error code field, since only some few exceptions push an error code. These exceptions use the separate `HandlerFuncWithErrCode` function type, which has an additional `error_code` argument.

Behind the Scenes

The `x86-interrupt` calling convention is a powerful abstraction that hides almost all of the messy details of the exception handling process. However, sometimes it's useful to know what's happening behind the curtain. Here is a short overview of the things that the `x86-interrupt` calling convention takes care of:

- **Retrieving the arguments**: Most calling conventions expect that the arguments are passed in registers. This is not possible for exception handlers, since we must not overwrite any register values before backing them up on the stack. Instead, the `x86-interrupt` calling convention is aware that the arguments already lie on the stack at a specific offset.
- **Returning using `iretq`**: Since the interrupt stack frame completely differs from stack frames of normal function calls, we can't return from handlers functions through the normal `ret` instruction. Instead, the `iretq` instruction must be used.
- **Handling the error code**: The error code, which is pushed for some exceptions, makes things much more complex. It changes the stack alignment (see the next point) and needs to be popped off the stack before returning. The `x86-interrupt` calling convention handles all that complexity. However, it doesn't know which handler function is used for

which exception, so it needs to deduce that information from the number of function arguments. That means that the programmer is still responsible to use the correct function type for each exception. Luckily, the `InterruptDescriptorTable` type defined by the `x86_64` crate ensures that the correct function types are used.
- **Aligning the stack**: There are some instructions (especially SSE instructions) that require a 16-byte stack alignment. The CPU ensures this alignment whenever an exception occurs, but for some exceptions it destroys it again later when it pushes an error code. The `x86-interrupt` calling convention takes care of this by realigning the stack in this case.

If you are interested in more details: We also have a series of posts that explains exception handling using naked functions linked at the end of this post.

Implementation

Now that we've understood the theory, it's time to handle CPU exceptions in our kernel. We'll start by creating a new interrupts module in `src/interrupts.rs`, that first creates an `init_idt` function that creates a new `InterruptDescriptorTable`:

```
// in src/lib.rs

pub mod interrupts;

// in src/interrupts.rs

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

pub fn init_idt() {
    let mut idt = InterruptDescriptorTable::new();
}
```

Now we can add handler functions. We start by adding a handler for the breakpoint exception. The breakpoint exception is the perfect exception to test exception handling. Its only purpose is to temporarily pause a program when the breakpoint instruction `int3` is executed.

The breakpoint exception is commonly used in debuggers: When the user sets a breakpoint, the debugger overwrites the corresponding instruction with the `int3` instruction so that the CPU throws the breakpoint exception when it reaches that line. When the user wants to continue the program, the debugger replaces the `int3` instruction with the original instruction again and continues the program. For more details, see the "*How debuggers work*" series.

For our use case, we don't need to overwrite any instructions. Instead, we just want to print a message when the breakpoint instruction is executed and then continue the program. So let's create a simple `breakpoint_handler` function and add it to our IDT:

```
// in src/interrupts.rs

use x86_64::structures::idt::{InterruptDescriptorTable, InterruptStackFrame};
use crate::println;

pub fn init_idt() {
    let mut idt = InterruptDescriptorTable::new();
    idt.breakpoint.set_handler_fn(breakpoint_handler);
}

extern "x86-interrupt" fn breakpoint_handler(
    stack_frame: InterruptStackFrame)
{
    println!("EXCEPTION: BREAKPOINT\n{:#?}", stack_frame);
}
```

Our handler just outputs a message and pretty-prints the interrupt stack frame.

When we try to compile it, the following error occurs:

```
error[E0658]: x86-interrupt ABI is experimental and subject to change (see issue #4
  --> src/main.rs:53:1
   |
53 | / extern "x86-interrupt" fn breakpoint_handler(stack_frame: InterruptStackFram
54 | |     println!("EXCEPTION: BREAKPOINT\n{:#?}", stack_frame);
55 | | }
   | |_^
   |
   = help: add #![feature(abi_x86_interrupt)] to the crate attributes to enable
```

This error occurs because the `x86-interrupt` calling convention is still unstable. To use it anyway, we have to explicitly enable it by adding `#![feature(abi_x86_interrupt)]` on the top of our `lib.rs` .

Loading the IDT

In order that the CPU uses our new interrupt descriptor table, we need to load it using the `lidt` instruction. The `InterruptDescriptorTable` struct of the `x86_64` provides a `load` method function for that. Let's try to use it:

```
// in src/interrupts.rs

pub fn init_idt() {
    let mut idt = InterruptDescriptorTable::new();
    idt.breakpoint.set_handler_fn(breakpoint_handler);
    idt.load();
}
```

When we try to compile it now, the following error occurs:

```
error: `idt` does not live long enough
  --> src/interrupts/mod.rs:43:5
   |
43 |     idt.load();
   |     ^^^ does not live long enough
44 | }
   | - borrowed value only lives until here
   |
   = note: borrowed value must be valid for the static lifetime...
```

So the `load` methods expects a `&'static self`, that is a reference that is valid for the complete runtime of the program. The reason is that the CPU will access this table on every interrupt until we load a different IDT. So using a shorter lifetime than `'static` could lead to use-after-free bugs.

In fact, this is exactly what happens here. Our `idt` is created on the stack, so it is only valid inside the `init` function. Afterwards the stack memory is reused for other functions, so the CPU would interpret random stack memory as IDT. Luckily, the `InterruptDescriptorTable::load` method encodes this lifetime requirement in its function definition, so that the Rust compiler is able to prevent this possible bug at compile time.

In order to fix this problem, we need to store our `idt` at a place where it has a `'static` lifetime. To achieve this we could allocate our IDT on the heap using `Box` and then convert it to a `'static` reference, but we are writing an OS kernel and thus don't have a heap (yet).

As an alternative we could try to store the IDT as a `static`:

```rust
static IDT: InterruptDescriptorTable = InterruptDescriptorTable::new();

pub fn init_idt() {
    IDT.breakpoint.set_handler_fn(breakpoint_handler);
    IDT.load();
}
```

However, there is a problem: Statics are immutable, so we can't modify the breakpoint entry from our `init` function. We could solve this problem by using a `static mut`:

```rust
static mut IDT: InterruptDescriptorTable = InterruptDescriptorTable::new();

pub fn init_idt() {
    unsafe {
        IDT.breakpoint.set_handler_fn(breakpoint_handler);
        IDT.load();
    }
}
```

This variant compiles without errors but it's far from idiomatic. `static mut` s are very prone to data races, so we need an `unsafe` block on each access.

Lazy Statics to the Rescue

Fortunately the `lazy_static` macro exists. Instead of evaluating a `static` at compile time, the macro performs the initialization when the `static` is referenced the first time. Thus, we can do almost everything in the initialization block and are even able to read runtime values.

We already imported the `lazy_static` crate when we created an abstraction for the VGA text buffer. So we can directly use the `lazy_static!` macro to create our static IDT:

```
// in src/interrupts.rs

use lazy_static::lazy_static;

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

pub fn init_idt() {
    IDT.load();
}
```

Note how this solution requires no `unsafe` blocks. The `lazy_static!` macro does use `unsafe` behind the scenes, but it is abstracted away in a safe interface.

Running it

The last step for making exceptions work in our kernel is to call the `init_idt` function from our `main.rs` . Instead of calling it directly, we introduce a general `init` function in our `lib.rs` :

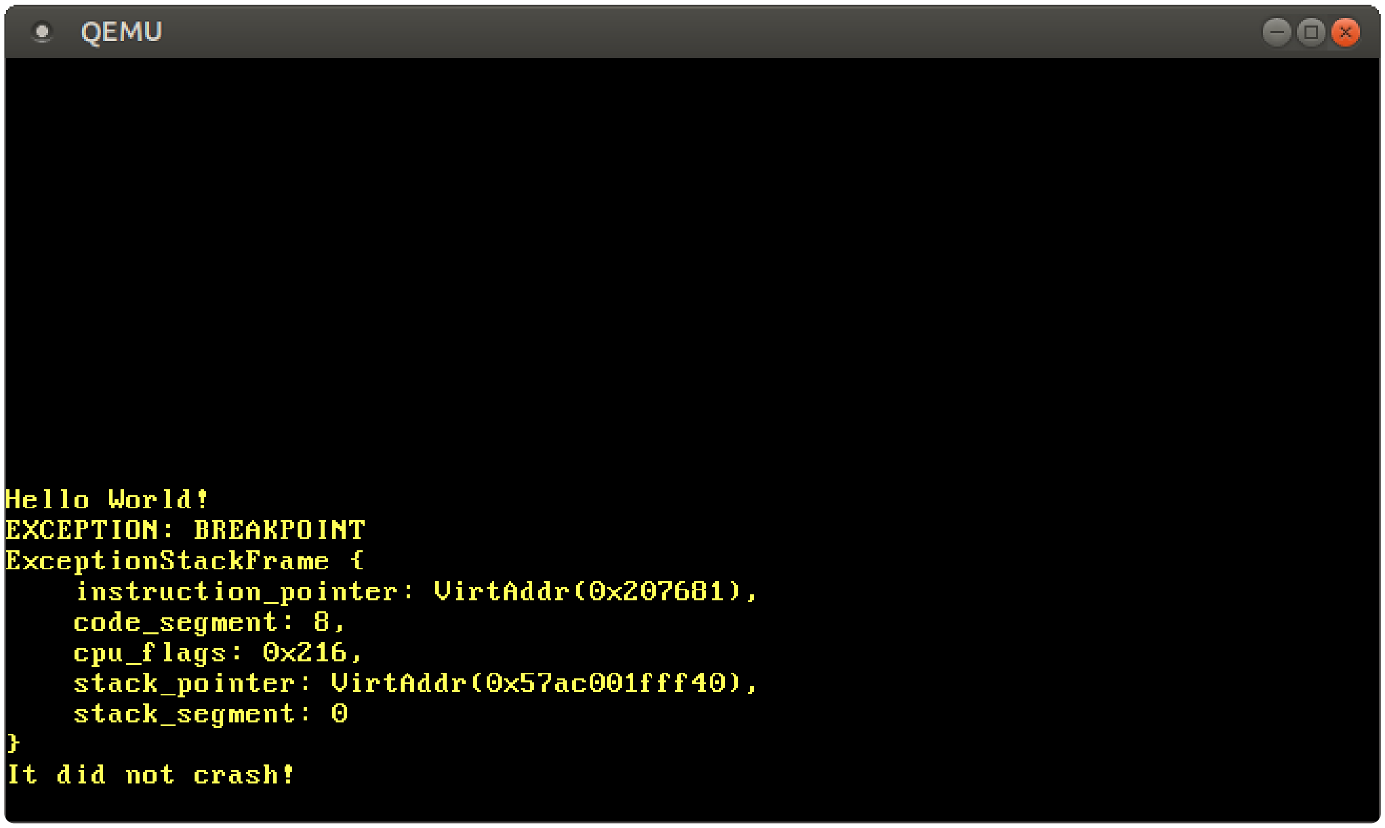
```
// in src/lib.rs

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

With this function we now have a central place for initialization routines that can be shared between the different `_start` functions in our `main.rs` , `lib.rs` , and integration tests.

Now we can update the `_start` function of our `main.rs` to call `init` and then trigger a breakpoint exception:

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

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

    blog_os::init(); // new

    // invoke a breakpoint exception
    x86_64::instructions::interrupts::int3(); // new

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

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

When we run it in QEMU now (using `cargo run`), we see the following:



```
Hello World!
EXCEPTION: BREAKPOINT
ExceptionStackFrame {
    instruction_pointer: VirtAddr(0x207681),
    code_segment: 8,
    cpu_flags: 0x216,
    stack_pointer: VirtAddr(0x57ac001fff40),
    stack_segment: 0
}
It did not crash!
```

It works! The CPU successfully invokes our breakpoint handler, which prints the message, and then returns back to the `_start` function, where the `It did not crash!` message is printed.

We see that the interrupt stack frame tells us the instruction and stack pointers at the time when the exception occurred. This information is very useful when debugging unexpected exceptions.

Adding a Test

Let's create a test that ensures that the above continues to work. First, we update the `_start` function to also call `init`:

```
// in src/lib.rs

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

Remember, this `_start` function is used when running `cargo test --lib`, since Rust's tests the `lib.rs` completely independent of the `main.rs`. We need to call `init` here to set up an IDT before running the tests.

Now we can create a `test_breakpoint_exception` test:

```
// in src/interrupts.rs

#[test_case]
fn test_breakpoint_exception() {
    // invoke a breakpoint exception
    x86_64::instructions::interrupts::int3();
}
```

The test invokes the `int3` function to trigger a breakpoint exception. By checking that the execution continues afterwards, we verify that our breakpoint handler is working correctly.

You can try this new test by running `cargo test` (all tests) or `cargo test --lib` (only tests of `lib.rs` and its modules). You should see the following in the output:

```
blog_os::interrupts::test_breakpoint_exception...       [ok]
```

Too much Magic?

The `x86-interrupt` calling convention and the `InterruptDescriptorTable` type made the exception handling process relatively straightforward and painless. If this was too much magic for you and you like to learn all the gory details of exception handling, we got you covered: Our

"Handling Exceptions with Naked Functions" series shows how to handle exceptions without the `x86-interrupt` calling convention and also creates its own IDT type. Historically, these posts were the main exception handling posts before the `x86-interrupt` calling convention and the `x86_64` crate existed. Note that these posts are based on the first edition of this blog and might be out of date.

What's next?

We've successfully caught our first exception and returned from it! The next step is to ensure that we catch all exceptions, because an uncaught exception causes a fatal triple fault, which leads to a system reset. The next post explains how we can avoid this by correctly catching double faults.

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!

« Testing Double Faults »

0 reactions

41 comments – *powered by giscus*

1

dodikk Jul 4, 2018 edited

Got a compiler error while trying out the code from listing.

```
error: use of deprecated item 'x86_64::structures::idt::Idt': type was renamed
  --> src/main.rs:56:31
   |
56 | use x86_64::structures::idt::{Idt, ExceptionStackFrame};
```

```
    |                                        ^^^
```

The error is obvious to fix.
Still, it would be a better experience to have correct listings out of box.

P.S. My code has the `x86_64 = "0.2.6"` dependency as recommended in the previous articles.

A dedicated github issue : #452

0 replies

1

phil-opp Jul 4, 2018 Owner

Yeah, we released a new x86_64 version with this renaming a few days ago. I thought it's better to wait a bit before updating the post to avoid breakage for the users on older versions. Normally the deprecation only causes a warning instead of an error, so I thought this wouldn't break new users.

The question is why it's an error for you. Are you explicitly using deny(warnings) or something like that?

0 replies

1

dodikk Jul 4, 2018

> The question is why it's an error for you. Are you explicitly using deny(warnings) or something like that?

Yes, that's the case. I've continued the discussion (a few more questions from me) in a dedicated ticket.

0 replies

1

yjhmelody Jul 19, 2018

I consider it strange to write these exception codes to main.rs or bin/*.rs. Why not write them to lib.rs which can pub a code.

0 replies

1

phil-opp Jul 19, 2018 Owner edited

Because we want to do different things on an exception (compare the implementations of the `breakpoint_handler` functions).

Edit: We moved the exception handlers to an `interrupts` module in #475.

0 replies

1

k0pernicus Nov 5, 2018

Thank you first for this awesome blog post!
There is an error in the part "Loading the IDT", where, just before the subpart "Lazy Statics to the Rescue", you are declaring your `IDT` variable as an `Option<...>`.
I think you have to remove the `Option` type here :)

Good luck, and thanks again!

0 replies

1

phil-opp Nov 5, 2018 Owner

@k0pernicus You're right, thank you! Fixed in `442da8c`.

0 replies

1

lilyball Nov 11, 2018

Why does `src/interrupts.rs` have `extern crate x86_64;`? That's already done in `src/lib.rs`.

0 replies

1

phil-opp Nov 13, 2018 Owner

Good catch! Removed in `d9f3b3d`.

0 replies

1

JoshMcguigan Jan 19, 2019

Hello, and thanks for your work on this blog series! I've been learning a lot by working through it.

One thing I was unsure about in this section, is why does the integration test not rely on our new `interrupts` module? It seems the test is actually checking whether the `x86_64` crate is correct. Am I missing something here?

0 replies

1

phil-opp Jan 20, 2019 Owner

@JoshMcguigan Good point! I think I already had the test for double faults in mind when writing this post. The double fault test makes sense because it checks that we have a correct GDT and TSS setup, but the test in this post requires no previous setup so it really tests only the x86_64 crate.

I don't think that we can reuse the `interrupts` module for the test, because we need to do different things in the handler functions. Therefore I propose to simply remove it. I opened #526 for this.

0 replies

1

AntoineSebert Feb 10, 2019 Contributor

Is there a way to define your own interrupts with dedicated handlers (without colliding with the hardware interrupts to implement later) ?

0 replies

1

phil-opp Feb 11, 2019 Owner

@AntoineSebert You mean software interrupts? Yes, you can just use any of the higher IDT entries (i.e. the entries not used for exceptions or hardware interrupt).

0 replies

1

AntoineSebert Feb 11, 2019 Contributor

@phil-opp
Since the OS is a x86-64 architecture and you use `usize` to access the elements, does it means that the theoretical maximal number of interrupts (and the IDT's max size) is 18.446.744.073.709.551.615 ?
So I can just assign a number higher than 47 (the last hardware interrupt's number) to an new interrupt ?

0 replies

1

phil-opp Feb 11, 2019 Owner

@AntoineSebert No, the `Idt` size is limited to 256 entries because the `int` instruction uses an `u8` for the interrupt number. Accessing `Idt[256]` or higher will result in a panic, similar to index operations on arrays.

> So I can just assign a number higher than 47 (the last hardware interrupt's number) to an new interrupt ?

Yes. For example, you could set `Idt[60]` to a handler function and then invoke `int 60` to run it.

0 replies

11 hidden items
Load more...

1

sunyulin728 Aug 25, 2019

@64 , Thank you.

The code is at https://github.com/sunyulin728/rustkvm. Sorry that it is just experiement code and the related path is hardcoded.

1, It need to be clone to /home/brad/rust/rustkvm
2. cd /home/brad/rust/rustkvm/
3. ./setup
3. cd qkernel
4. make
5. cd ../qvsior
6. Cargo run

0 replies

1

64 Aug 25, 2019

@sunyulin728 I haven't run the code myself (it's a little tricky to setup) but it sounds like it's an issue with either the GDT or paging - but it's pretty difficult for me to diagnose since you're going through a hypervisor.

If you're running in QEMU you can use `-d int` to print detailed information about the crash that you're getting which can help you diagnose the issue.

I understand that this isn't the best place for extended help so if you have further questions I'd be happy to answer them on the rust-osdev gitter.

👍 1

0 replies

1

sunyulin728 Aug 28, 2019

@matt Taylor, Thank you!

I agree with you. The issue is very likely from GDT setup.

I translate the KVM sample from https://github.com/david942j/kvm-kernel-example to Rust. But the orginal code didn't setup GDT at all.

I study the gdt setup code in https://os.phil-opp.com/double-fault-exceptions/. I find there is only cs and tss setup. There is no other *s such ds setup related code.

Do you know why?

0 replies

1

phil-opp Aug 29, 2019 Owner

@sunyulin728

> I study the gdt setup code in https://os.phil-opp.com/double-fault-exceptions/. I find there is only cs and tss setup. There is no other *s such ds setup related code.

According to the AMD/Intel manuals, the content of the other segment registers are ignored in 64-bit. See rust-osdev/x86_64#78 for some discussion about this.

0 replies

1

GuillaumeDIDIER Oct 6, 2019

The bug for error code apparently has been fixed on rust nightly, perhaps this reference should be changed to the past. (There used to be a bug with LLVM, but it is now fixed in the latest nightlies)

0 replies

1

phil-opp Jan 5, 2020 Owner

@GuillaumeDIDIER Good catch, thanks for reporting (and sorry for the late reply)! I removed the sentence about the bug in #711.

0 replies

1

midwinter1993 Feb 5, 2020

Hi **@phil-opp**, great post!

I have the same question about the claim `all registers are preserved` of the `x86-interrupt` convention as **@AtsukiTak**.

I tested a code snippet:

```
extern "x86-interrupt" fn timer_handler(stack_frame: &mut InterruptStackFrame) {
}
```

and I disassemble it:

```
; extern "x86-interrupt" fn timer_handler(stack_frame: &mut InterruptStackFrame)
  20f3a0: 50                                    pushq   %rax
  20f3a1: 50                                    pushq   %rax
  20f3a2: fc                                    cld
  20f3a3: 48 8d 44 24 10                        leaq    16(%rsp), %rax
  20f3a8: 48 89 04 24                           movq    %rax, (%rsp)
; }
  20f3ac: 48 83 c4 08                           addq    $8, %rsp
  20f3b0: 58                                    popq    %rax
  20f3b1: 48 cf                                 iretq
```

I didn't see pushing the general registers; so, where does the magic happens?
Thank you :-P

0 replies

1

GuillaumeDIDIER Feb 5, 2020

@midwinter1993 LLVM did the magic expected of it. It only saved registers that are clobbered by your function (hence rax). All other register are left unchanged hence there's no need of saving and restoring them.

0 replies

1

midwinter1993 Feb 5, 2020

@GuillaumeDIDIER
Thank you for clarifying the claim "preserves all registers". :-P

0 replies

1

phil-opp Feb 7, 2020 Owner

@midwinter1993 I just pushed `a3eeb1d` to make this more clear in the post. Thanks for asking!

0 replies

1

RoidoChan May 7, 2020

Hi Phil,
Love the series! With this post, when I use cargo xrun the executable seems to crash when it hits the "x86_64::instructions::interrupts::int3();" line and reboots over and over and qemu seems stuck in a loop. It never reaches the "it did not crash" println statement.

I've checked your branch for this post and verified there's no difference in code, cargo.toml nor the json file.

I've also commented the int3() line out and sprinkled some printlns here and there to see if the "right" functions are exectuted in my interrupts.rs and lib.rs files, so the functions to load the IDT are being hit.

Any ideas?

0 replies

1

GuillaumeDIDIER May 7, 2020

> Hi Phil,
> Love the series! With this post, when I use cargo xrun the executable seems to crash when it hits the "x86_64::instructions::interrupts::int3();" line and reboots over and over and qemu seems stuck in a loop. It never reaches the "it did not crash" println statement.
>
> I've checked your branch for this post and verified there's no difference in code, cargo.toml nor the json file.
>
> I've also commented the int3() line out and sprinkled some printlns here and there to see if the "right" functions are exectuted in my interrupts.rs and lib.rs files, so the functions to load the IDT are being hit.

> Any ideas?

The reboot looks like a triple fault (which is not surprising given that we don't have any double fault handler around).
I could suggest making use of an actual debugger (bochs with internal debugger is what I've been using), which could allow you to pin point what was the last instruction executed before things blew up (I now there are a few issues with stack alignment that have caused issue in the past if sse support is enabled)

0 replies

1

phil-opp May 7, 2020 Owner

Hi **@RoidoChan**, great to hear that you like the blog!

> I've checked your branch for this post and verified there's no difference in code, cargo.toml nor the json file.

Have you tried cloning the repo and running it directly? Sometimes it's a small typo or a file that one didn't think of that is causing the issue. Also, if you have your code online somewhere I can take a look and try to reproduce it.

> With this post, when I use cargo xrun the executable seems to crash when it hits the "x86_64::instructions::interrupts::int3();" line and reboots over and over and qemu seems stuck in a loop.

Sounds like something is wrong with your IDT setup. You can try passing `-d int` to QEMU (through `cargo xrun -- -d int`) to get a list of exceptions that occur. This trace contains the value of the instruction pointer for each exception, which you can then use to find out which instruction is causing it. A simple way for that is to grep the disassembly of your kernel (e.g. obtained via `objdump -d path_to_your_kernel_bin`).

Hope this helps! Please let me know what the issue was when you find it.

0 replies

1

RoidoChan May 7, 2020

Hi **@GuillaumeDIDIER**
Thanks for replying!

Hi **@phil-opp**,
I had a real nightmare trying to get cargo objdump -d to work, as It seemed the objdump didn't recognize the x86_64-unknown-none target for some reason, even i explicitly stated the .json file for the target.

I moved some "use" statements around in the .rs files and it started to work for some reason?

Worst thing is I didn't do a commit of the buggy version, so I can't diff and see exactly was causing the issue. Sorry!

Thanks so much for your time, and again the blog is great. Learning some comp sci fundamentals I never really understood, despite working professionally in games for some years.

signed up as a Patron! Keep it going!

0 replies

1

phil-opp May 11, 2020 Owner

@RoidoChan

> I had a real nightmare trying to get cargo objdump -d to work, as It seemed the objdump didn't recognize the x86_64-unknown-none target for some reason, even i explicitly stated the .json file for the target.

Yeah, I found `cargo objdump` difficult to use too. If you're on Linux or macOS, I recommend just using the system `objdump` on the ELF executable.

> I moved some "use" statements around in the .rs files and it started to work for some reason? Worst thing is I didn't do a commit of the buggy version, so I can't diff and see exactly was causing the issue. Sorry!

No worries, great to hear that it's working now!

> Thanks so much for your time, and again the blog is great. Learning some comp sci fundamentals I never really understood, despite working professionally in games for some years.
>
> signed up as a Patron! Keep it going!

It makes me happy to hear that. Thanks a lot for supporting me!

0 replies

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.