Module 1, Lesson 5, "Under-the-Hood: Exceptions and Interrupts"

Hello, and welcome to this lesson on Exceptions and Interrupts on the x86 architecture. This is one of several lessons where we look under the hood at how things work in detail on a real CPU, in this case the x86 family.

First we will focus on exceptions. One can think of exceptions as CPU-triggered interrupts, often in result to an error or to an unexpected situation. This will give us an opportunity to look at the interrupt vector table x86 style.

We will then look at how to build the software necessary to flexibly intercept and handle exceptions.

We will then do the same for interrupts.

Interrupts can be handled very similarly to exceptions, with minor differences, mostly due to the idiosyncrasies in the interrupt controller.

Before we start, a word of caution. This lesson will be quite lengthy and occasionally maybe a bit tedious. I encourage you to remain attentive as you go through the lesson, as it will prepare you for the machine problems to come. It describes the structure and operation of the exception and interrupt handling frameworks, which will be the base on which you will build your operating system.

Let's begin!

Interrup Descriptor Table

What we know as the interrupt vector table, is called the interrupt descriptor table on the x86, and it contains entries for exceptions, hardware interrupts, and software interrupts. An example of the latter is the Hex 80 system call interrupt.

The table has 256 entries, and each entry contains the address of the exception handler or the interrupt handler.

The x86 supports 32 different processor exceptions, which we will explore next. The first 32 entries in the interrupt descriptor table are thus reserved for exceptions.

The remaining entries can be freely used for handling hardware interrupts.

Processor Exceptions

We know already that the x86 supports 32 different exceptions. Each exception has a fixed number. Most of the exceptions are raised by the CPU when something unexpected happens that may need to be handled.

Exception 0, or example, it raised when a division by zero happens.

Another exception that you may encounter as part of the machine projects is Exception number 6, "invalid opcode". This exception is raised when the program counter points to undecipherable code, and the CPU does not know what to do.

Later in the course we will be dealing with Exception number 14, which is raised when the CPU is trying to access invalid memory.

When an exception happens, the CPU drops what it is doing, save the instruction pointer (also called program counter) and the CPU status register on the stack, and jumps to the location specified by the entry in the interrupt descriptor table. For some exceptions, the exception handler may need some additional information. For example, we will see later that a page fault handler for exception 14 needs access to the address that caused the invalid memory reference. For exceptions where this type of information is needed, the CPU also saves a so-called "error-code" on the stack.

In this table we see that for exceptions 8, and 10 to 14 the CPU saves an error code. For all the other exceptions, the CPU only saves the instruction pointer and status register.

Handling Exceptions

Let's explore how exceptions are handled.

For each of the 32 exceptions, we have a separate exception handler function, which we call _isr0 to _isr31, here are the declarations.

Here is the definition of the code for the divide-by-zero exception, with number zero. The role of this little piece of code is two-fold.

1. First, if the exception does not save an error number on the stack, this function pushes an dummy value zero. This is in order to simplify the code down the line.
2. Second, it pushes the exception number onto the stack, in this case number zero.

Once it has pushed these two values onto the stack, it continues execution at the label ISR_COMMON_STUB. We will get to this part of the code in a little while.

For an exception with an error-code, in this case the number 8 double-fault exception, the CPU has already pushed the error code onto the stack, so we can skip this part.

We just push the exception number, in this case the value 8, onto the stack, and continue execution at the common stub.

By the time we will be done with pushing all the context onto the stack, the top of the stack will look like the a structure, which we call REGS for registers.

Let's see how we are building this structure.

First, the processor pushes the instruction pointer, the code segment, and the status register EFLAGS onto the stack.

One of the several **__isr** functions gets called, which pushes either a dummy error code and the exception number onto the stack, or just the exception number in case of exceptions that push an error code in hardware.

The rest of the registers is pushed onto the stack by the piece of code starting at ISR_COMMON_STUB. In particular, the instruction PUSHA for "push all" saves all general purpose registers on the stack.

We explicitly save the four segment registers onto the stack.

Now we do a bit of gymnastics and call the function **__lowlevel__dispatch__exc**, which is implemented in C, with as argument the address of the top of the stack. By construction, the top-of-the-stack pointer can be interpreted as a pointer to the register structure depicted on the left. This makes it easy for C or C++ functions to access the register context when handling the exception. We will discuss the high-level part of the exception handling in a minute.

Now it also becomes clear why we do all this dummy-value pushing for exceptions without error code. By pushing dummies onto the stack, we only need one register structure. Otherwise we would need one for exceptions with error codes and one for exceptions without. This would make a mess of the exception dispatching down the line.

After returning from the high-level exception handler we need to restore the context so that we can return to where the exception was called. For this we explicitly pop the segment registers from the stack. We call the POPA instruction to pop all general-purpose registers from the stack, we pop the exception number and the error code off the stack and throw them away. We do this by simply adding twice the size of a 32-bit integer to the stack pointer.

Finally we return from the interrupt, which pops the instruction pointer, the code segment and the status register EFLAGS.

High-Level Exception Handler

Let's look at how we call the high-level exception handler.

In the assembly-level common stub we call the function **lowlevel_dispatch_exception**.

This function is defined in the exception handler as a C function. We do this because C++ has the habit of adding all kinds of pre-fixes to function names, and we don't know what these prefixes will be when we call the function from assembly code. The C language is much better behaved, and if we prefix the function definition with the key "extern C", then the C++ compiler understands that it should follow C-language convention and just add an underscore to the function name. This is why the same function in the Assembler code is called "underscore low level dispatch exception".

In this low-level dispatch exception we call the static function **dispatch_exception** of the class **ExceptionHandler**. Note that these functions expect a pointer to a register struct. This is the reason why we so carefully assembled the register context on the stack before branching out to the high level exception handler.

Let's have a closer look at the class **ExceptionHandler**.

Internally this class contains a static table of registered exception handler, one for each different exception. For the x86 CPU the size of this table (EXCEPTION_TABLE_SIZE) is 32.

Next there is a static function register handler, that allows us to register an exception handler for a given exception number. If we were to register a particular exception handler as a divide-by-zero handler, we would register the handler for number zero, and this code would store the pointer to the registered exception handler in the appropriate entry in the handler table defined above.

Next we have the function **dispatch_exception**, which gets called by the C function low-level dispatch exception every time an exception occurs.

Finally, we have the only non-static member of the exception handler, which is the handler for this particular exception.

Let's look at how the high-level exception dispatching works:

- first we figure out what exception was called. We find this information inside the register struct, which is passed to the function by the assembly layer code.

Using the exception number, we look up whether there is a handler registered for this exception. If not, we panic.

Otherwise, we call the handle exception function of the registered exception handler.

Example: Divide-by-Zero Exception Handler

With this framework it is very easy to write our own exception handler. Let's write one to handle divide-by-zero exceptions.

For this we derive class Divide-by-Zero Handler from class **ExceptionHandler**.

All we need to do is to override the **handle_exception** function.

We do that here.

All this exception handler does is to write an error message to the console and panic.

We install the exception handler in the kernel by defining a variable of type Divide-by-Zero handler, and by registering it for number zero exceptions with the exception dispatcher in class **ExceptionHandler**.

From now on, whenever the CPU encounters a division by zero operation and throws an exception, our exception handling framework gets exercised, and finally the **handle_exception** function of the Divide-by-Zero Handler object gets called, which displays an error message and then panics.

Initializing the IDT

One point that we have not explored yet is how the interrupt descriptor table gets initialized and populated. Here we just give a general idea.

Here we declare the functions **isr0** to **isr31** that we encountered earlier in the assembly code. We skip the underscore because – as we discussed a few minutes ago – it gets automatically added by the C++ compiler if we declare the function as extern "C".

In the dispatcher initialization function of the exception handler class we load the function pointer to the isr functions one by into the interrupt descriptor table, using the **idt_set_gate** function.

ISR 0 is loaded into entry 0, ISR 1 into entry 1, and so on.

We then proceed the initialize the high-level exception handler by marking all exception handler entries as unregistered.

The Exception handler framework is thus initialized by calling the function **init_dispatcher** at the very beginning of the kernel startup. Once it is set up, exception handler such as the Divide-by-zero exception handler that we just defined a minute ago can be defined and registered.

Hardware Interrupts: The Programmable Interrupt Controller (PIC)

Now let's talk about interrupts. Hardware interrupts are triggered by hardware devices raising a signal on a line that is fed into the interrupt controller, which then triggers an interrupt on the CPU.

In the case of older PCs the interrupt controller is a master/slave pair of 8259 programmable interrupt controllers, PICs for short. Today the 8259 is somewhat obsolete and has been replaced by the Advanced Programmable Interrupt Controller, APIC. One reason is that the older PIC does not support multiprocessors. Nevertheless we will use the venerable 8259 PIC in our machine problem because it is comparatively easy to use.

Interrupts, so called IRQs, are assigned entries in the interrupt descriptor table. These entries can be freely remapped.

Interrupt handlers work very similarly to exception handlers, with a few minor differences. Most prominently, at the end of the interrupt service routine, one has to send an END-OF-INTERRUPT command back to the interrupt controller to inform it that the interrupt has been handled. This gets complicated a bit if the signal came from the slave interrupt controller, in which case we need to send two commands, one to the master and the other to the slave.

Assigned Interrupt Lines in the PIC

Before we dive into the details of interrupt handling, let's have a brief look at what type of interrupts there are.

With the 8259, about eight interrupts IRQ 0 to IRQ 7 are handled by the master interrupt controller. These interrupts come from all kinds of devices,

- Interrupt 0 comes from the programmable system timer. This timer can be programmed to issue a timer interrupt at a fixed interval.
- Interrupt 1 comes from the keyboard.
- Interrupt 3 and 4 from serial lines, and so on.

The slave interrupt controller supports 8 interrupts as well. Of importance to us will be interrupt 14 and 15, which come from the two hard drive controllers.

Handling Hardware Interrupts

We handle interrupts largely in the same way we handle exceptions.

We have a list of low-level interrupt handling functions, IRQ 0 to IRQ 15. Each pushes a dummy error code and the interrupt descriptor table entry number onto the stack, and then calls the common stub.

Interrupt 0 uses entry 32 in the interrupt descriptor table, just after the exceptions. Interrupt 1 will use entry 33, and so on.

We note that all irq functions push a zero dummy error code onto the stack. This is because interrupts do not generate error codes as some exceptions do. Nevertheless we push dummy values because we want to re-use the same register struct that we used for exception handling.

High-Level Interrupt Handler

The high-level interrupt handler for interrupts is largely identical to the one for exceptions as well, with a few minor differences.

First, the low-level common stub does not call the low-level exception dispatcher, but the low-level interrupt dispatcher.

This function looks largely the same as the one for exceptions, except that it calls the **dispatch_interrupt** function of the Interrupt handler rather than the "dispatch exception" of the exception handler.

Let's have a closer look at the **dispatch_interrupt** function.

First we look at the interrupt number. Since the assembly level interrupt handlers pushed the idt entry number onto the stack, and the interrupts were mapped the entries just following the exceptions, we subtract the number of exceptions, in this case 32, from the entry number to get the actual interrupt number.

We look at the table of registered interrupts, just as we do for exceptions, and if there is not match we panic.

If there is a match we call the function **handle_interrupt** for the matching handler, similarly to what we do for exceptions.

Now comes the difference. Once we are done with handling the interrupt, we must send an **end_of_interrupt** command to the interrupt controller.

Moreover, if the interrupt came from the slave controller, we need to send a command to the slave controller as well.

We do that here, by sending a specially crafted command through an I/O port.

After this is taken care of, we send an END-OF-INTERRUPT command to the master controller.

Interrupt Handler Example: Periodic Timer

Let's put all this to practice by implementing a simple timer.

We do that by deriving class **SimpleTimer** from **InterruptHandler**.

The timer gets updated by a system timer interrupt, which fires periodically. Every time the system timer fires, we increment the number of ticks in the **SimpleTimer**.

Once we have collected enough ticks to sum up to a second, we increment the number of seconds.

For this to work correctly, we need to set the period of the system timer. How frequently do we update the ticks counter?

We set the period of the system timer in function **set_frequency**.

We initialize the **SimpleTimer** to be updated with a given frequency. This initializes the system timer to fire interrupts at the correct frequency.

Every time the system timer fires an interrupt, we handle it with the **handle_intertupt** function of the simple timer.

We construct a simple timer by initializing ticks and seconds to zero, and by setting the system timer frequency.

Setting the system timer frequency is done by sending a few carefully crafted commands to the I/O ports.

The core of the simpler timer is the function to handle the interrupts generated by the system timer.

First, we increment the number of ticks.

If the we have enough ticks to make for a second, we increment the number of seconds, set the ticks back to zero, and tell the world that a second has passed.

How do we install this timer in the kernel?

Let's look at how this can be done.

First the we initialize all kind of basic infrastructure related to exception, interrupt handling, and other. We initialize the global descriptor table, then the console.

Then we initialize the interrupt descriptor table and the exception dispatcher. We described earlier how this is done.

Then we initialize the part of the interrupt descriptor table that deals with interrupts. This is basically a bunch of remappings, with need to be communicated to the PIC. We then initialize the interrupt dispatcher, and everything is ready to go.

We define an object of type **SimpleTimer**, and set the system timer to fire every 10 milliseconds. And we register the timer to handle interrupt 0, which is the system timer interrupt. At this point nothing happens yet because the machine starts with all maskable interrupts masked.

We therefore enable the interrupts, by basically invoking the "STI" instruction to set the interrupt flag in the status register EFLAGS.

We let the world know that we are alive, and now the system will print a message to the screen whenever a second has passed.

Summary: Interrupts and Exceptions on the x86

We have now come to the conclusion of this rather lengthy under-the-hood look of exception and interrupt handling on the x86.

We have started by looking at processor exceptions and

how the addresses of exception handling and interrupt handling routines are stored in the interrupt descriptor table.

We have looked at a simple exception handler framework that you will be using in your machine problems.

And we have written a divide-by-zero exception handler.

We have learned about hardware interrupts, and how they are handled by the the 8259 programmable interrupt controller.

We looked at a simple interrupt handler framework, and we observed that it is not that different from the exception handling framework we looked at earlier.

We then practiced what we learned by building a simple timer.

I sincerely hope that you enjoyed this lecture on Exception and Interrupt handling on the x86. With the material that you have learned in this lecture you will be able to implement your own exception handling in the machine problems to come to implement paged memory management or implement high performance device drivers using interrupt handlers.

Thank you for watching!