8 Interrupts and the IDT

Notes about Adaptation

This is the unedited script of Chapter 8. The reference code for it can be found here:
`https://web.archive.org/web/20200117221638/http://www.osdever.net/bkerndev/Docs/idt.htm`

I know it's not the same without the animations, but if you read this while looking at the code from `osdever.net`, it might be a bit easier. For any questions or corrections, write me at:

`projectdaedalus.off@gmail.com`

Introduction

Today we're talking about interrupts. The implementation we're going to describe is not mine, and we're not going to write a lot of code for it, because while it is not very complex, it surely is quite a long implementation to explain step by step in a video. I'm going to give you a general idea of how interrupts work, but you will need to have a good read of the article you'll find in the description for a deeper understanding.

The properties and behaviour of each interrupt are described in the Interrupt Descriptor Table.

8.1 IDT

Just like the GDT, the IDT is a data structure, characterized by two values: the base, which is the starting address of the IDT, and the limit, which is the length of the IDT in bytes. Each entry of the IDT corresponds to an interrupt, and there should be up to 256 of them.

An IDT entry has the following properties:

- A 32 bit offset, which points to a thing called "Interrupt service routine", that we can think as the special function called during the interrupt
- A 16 bit selector, which describes a code segment (in our case *the* code segment). The selector for the code segment (which is the first entry of our GDT) is `8`. This value is calculated by setting the lowest two bits of the selector to the privilege level of our code segment (which is zero), the third to zero (it would be one if we were using a local descriptor table, but we are not) and the rest to the index of our code segment descriptor in the GDT. As it is the first entry, this index is one. So we end up with the 16 bit binary number:

  `0000000000001000`

  which is 8.
- Then we have eight "unused" bits, set to zero.
- And finally, a byte of type flags:
  - The first four bits represent what type of IDT gate we are defining, in this case we are defining a 32 bit interrupt, so these bits should be `1 1 1 0`, or `0xe`.
  - The fifth bit should be zero
  - The next two bits contain the descriptor privilege level, and should also be zero
  - The final bit contains 1 if the interrupt is "present", so it should be 1.

  So: `10001110`, which is `0x8E`.

In C we can define an IDT entry with a struct.
In this one we can notice how the offset, here called "base" is split in two, just like it was in the GDT.

To load the IDT, we're going to use the `lidt` instruction.

8.2 Interrupt Service Routines, Exceptions

For each interrupt we want to use, we need to define an interrupt service routine, pointed to by the offset of the corresponding IDT entry. As I said, these are special functions. We need to define them in assembly, and the main difference between ISRs and normal functions is in the return statement: instead of using the `ret` instruction, they use `iret`.

Some interrupts are used to communicate exceptions. When we, say, try to divide by zero, or encounter some kind of fault, one of 32 special interrupts is triggered. To catch these exceptions we need to have at least 32 IDT entries, each with a corresponding service routine. Some of these exceptions are considered errors. Whenever these interrupts happen, an error code is pushed onto the stack.

In the implementation we're using, each of these 32 functions has the following shape:

- First, if the exception does not have an error code, we push a dummy one to the stack.
- Then we push the interrupt number to the stack, to be used later.
- Finally, we jump to an "ISR common stub",

that calls a "fault handler", defined in C, that checks whether the interrupt is an exception, based on the interrupt number, and prints the corresponding message, stored in an array of strings, and halts the execution. Before calling the fault handler, you can notice how we push the current stack frame to the stack, so that the current state of the registers is saved in the actual parameter of the function, this struct. This way, as we had pushed the interrupt number, we can access it in our function.

In a more serious implementation we might want to treat different kinds of exceptions differently, but this is ok for now.
After calling the handler, we need to restore the state of the stack.

8.3 Interrupt Requests

Some interrupts might be triggered by hardware, like for example a keyboard. Hardware devices send an interrupt request to the CPU. These requests are managed by two chips, called PICs. One of these is called the master, and the other one the slave.

You can (and should) read more about them and how they work in the article in the description, but what you need to know to get an idea of how interrupt request works is that, combined, they can manage 16 different interrupt requests.

Of course, for each of these interrupts, we need to create an IDT entry that points to a service routine. These service routines are going to be called as soon as the interrupt is triggered. Once the interrupt has been handled, we need to inform the PICs, and we do so by writing the command byte `0x20` to the command register for the relevant PIC, located in one of two different I/O ports. We have never used I/O ports before, so I'm going to introduce them briefly before explaining what this means.

Our address space is divided in two distinct parts. There is memory, which we know very well at this point, and there are I/O ports. I/O, as you might imagine, stands for input and output. By reading and writing values to I/O ports, you can get information from your hardware, or, as we want to do here, set some register and change its state.

This is done by using the assembly instructions `inb` and `outb`.

One very useful thing you can use not to switch to your assembly files all the time is inline assembly. That's right, you can write inline assembly code in C. I'm leaving you an article on how to do that in the description, now back on topic.

As I was saying, in order to tell the PICs we are done, we need to write `0x20` to the command register of the PIC we're using. The master PIC manages the interrupts from 0 to 7, so if we've been handling one of those interrupt requests we need to set the master PIC's control register, located in I/O port `0x20` to `0x20`. If we've been handling an interrupt request between 8 and 15, we need to set the PIC's slave control register, located at `0xa0` and the master one to `0x20`.

If we have a look at the implementation, we will notice how these are treated very similarly to the other ISRs. Each IRQ, which stands for interrupt request, is pointed to by the offset of an IDT entry.

These functions are called whenever the corresponding interrupt request is executed. As you can see, they all jump to, this time, an "irq common stub", where the IRQ handler is called. The irq handler selects a handler function based on the irq number, that is retrieved in the very same way described earlier, and calls it if it exists. (An example of a handler function can be the one that decides what character to print when we press a key). Finally it sets the appropriate PIC registers to `0x20`. You might notice there's a 40 there, I'll explain it in a moment.

Once we return from the IRQ handler, we restore the state of the stack and use the `iret` instruction to return from the service routine.

What's with that 40? Well, we said that whether we should set the master or the slave's control register depends on whether the interrupt request number is less than 8 or not. Well, if we set interrupt request zero as the 32nd IDT entry, right after the exceptions, IRQ number 8 would very much be the 40th IDT entry. The thing is, how do we tell the PICs that we want IRQs to start from the 32 IDT entry. To do this, we need to execute this black magic I/O stuff, and we will have remapped the IRQs to where we want them to be.

8.4 Keyboard

A very simple example of IRQ handler is, as I mentioned, the keyboard handler. The keyboard is associated with interrupt request 1, so we should set the "keyboard handler" function as the IRQ1 handler. In our implementation, this is simply done by calling the "irq install handler" function, which simply sets the first element of an array of pointers to a pointer to our handler function. This way, when we execute the "irq handler" function, the handler will turn out to exists and will be executed.

What the keyboard handler does is it retrieves a scancode by reading I/O port `0x60` and uses this big array, which contains the characters corresponding to each scancode, to translate it to a character. Of course, some characters do not represent characters, but other keys like caps lock or shift, and these can be filtered by using this check.

The article goes on and describes how to set up a timer, which is a very similar procedure and you might want to try and do it yourself.

As I mentioned, this is the last episode of this series. I might eventually make other videos about OS development, but I also have other projects in mind, so I will probably make some videos about other topics.

Today's exercises are:

- Try and make today's code work. It's quite a challenge, and it's probably not going to work at the first try, but I suggest you to keep trying, and reading, and learning until you manage to make it work.
- Don't stop here! Try and write a memory manager, experiment with dynamic data structures, make some more drivers and make this operating system your operating system.

Good luck with your projects, see you next time!