Chapter 9: Filesystems

Introduction

Till this point, we have seen how a kernel of an operating system works as a resource manager. Given that both the processor and the main memory are resources in the system, 539kernel manages these resources [1] and provides them to the different processes in the system. Another role of a kernel is to provide a way to communicate with external devices, such as the keyboard and hard disk. *Device drivers* are the way of realizing this role of the kernel [2]. The details of the external devices and how to communicate with them are low-level and may be changed at any time. The goal of a device driver is to communicate with a given device by using the device's own language and the other goal of a device driver is to provide an interface for any other component of the system that wish to use the given device, most probably the low-level details of this given device will be hidden behind the interface that the device driver provides, that means the user of the device drivers doesn't need to know anything about how the device really work.

The matter of hiding the low-level details with something higher-level is too important and can be found in, basically, everywhere in computing and the kernels are not an exception of that. Of course, there is virtually no limit of providing higher-level concepts based a previous lower-level concept, also upon something that we consider as a high-level concept we can build something even higher-level. Beside the previous example of device drivers, one of obvious example where the kernels fulfill the role of hiding the low-level details and providing something higher-level, in other words, providing an *abstraction*, is a filesystem which provides the well-known abstraction, a file.

In this chapter we are going to cover these two topics, device drivers and filesystem by using 539kernel. As you may recall, it turned out that accessing to the hard disk is an important aspect for virtual memory, so, to be able to implement virtual memory, the kernel itself needs to access the hard disk which makes it an important component in the kernel, so, we are going to implement a device driver that communicate with the hard disk in this chapter. After getting the ability of reading from the hard disk or writing to it, we can explore the idea of providing abstractions by the kernel through writing a filesystem that uses the hard disk device driver and provides a higher-level view of the hard disk, that we all familiar with, instead of the physical view of the hard disk which has been described previously in chapter . The final result of this chapter is version `NE` of 539kernel which has as we mentioned a hard disk device driver and a filesystem.

[1] Incompletely of course, to keep 539kernel as simple as possible, only the basic parts of resources management were presented.

[2] At least a monolithic kernel.

ATA Device Driver

No need to say the hard disks are too common devices that are used as secondary storage devices. Also, there are a lot of manufacturers that manufacture hard disks and sell them. Imagine for a moment that each hard disk from a different manufacturer use its own way for the communication, that is, the method `X` should be used to be able to communicate with hard disks from manufacturer `A` while the method `Y` should be used with hard disks from manufacturer `B`. Given that there are too many manufacturers, this will be a nightmare. Each hard disk will need its own device driver which talks a different language from the other hard disk device drivers. Fortunately, this is not the case, at least for the hard disks, in this type of situations, standards are here to the rescue. A manufacturer may design the hard disk hardware in anyway, but when it comes to the part of the communication between the hard disk and the outside world, a standard can be used, so, any device driver that works with this given standard, will be able to communicate with this new hard disk. There are many well-known standards that are related to the hard disks, *small computer system interface* (SCSI) is one of them, another one is *advanced technology attachment* (ATA) [3]. While the latter one is more common in personal computers we are going to focus on it here and write a device driver for it, the former one is more common in servers.

As in PIC which is been discussed in chapter , ATA hard disks can be communicated with by using port-mapped I/O communication through the instructions `in` and `out` that we have covered previously. But before discussing the ATA commands that let us to issue a read or write request to the hard disk, let's write two routines in assembly that can be used in C code and perform the same functionality for the instructions `in` and `out`. If you didn't recall, the instruction `out` is used to write some bytes to a given port number, so, if we know the port number that a device receives that commands from, we can use the instruction `out` to write a valid command to that port, on the other hand, the instruction `in` reads data from a given port, for example, sometimes after we send a command to a device, it responds through writing something on a specific port, the instruction `in` can be used to read this value. The assembly code of the both routines that we are going to define next should reside in `starter.asm` anywhere between `bits 32` and the beginning of `start_kernel` routine. As we have said previously, the goal of these two routines is to make it possible for C code to use the instructions `in` and `out` through calling these routines. The following is the code of `dev_write` which can be used by C kernel code to write to a given port. In C, we can see that it has this prototype: `dev_write( int port, int cmd )`.

```
dev_write:
    ; Part 1
```

[3] Another well-known name for ATA is *Integrated Drive Electronics* (IDE). The older ATA standard is now known as Parallel ATA (PATA) while the newer version of ATA is known is Serial ATA (SATA).

```
    push edx
    push eax

    ; Part 2
    xor edx, edx
    xor eax, eax

    ; Part 3
    mov dx, [esp + 12]
    mov al, [esp + 16]

    ; Part 4
    out dx, al

    ; Part 5
    pop eax
    pop edx

    ret
```

The core part of this routine is part four which contains the instruction `out` that send the value which is stored in `al` to the port number which is stored in `dx`. Because we are using these two registers [4], we push their previous values into the stack as we did in the first part of the routine, pushing the previous values of these registers lets us restore them easily after the routine finishes its work, this restoration is performed in the fifth part of the routine right before returning from the routine, this is an important step to make sure that when the routine returns, the environment of the caller is same as the one before calling the routine. After storing the previous values of `eax` and `edx` we can use them freely, so, the first step after that is to clear their previous values by setting the value `0` to the both of them, as you can see, we have used `xor` and the both operands of it are the same register that we wish to clear, this is a well-known way in assembly programming to clear the value of a register [5]. After that, we can move the values that have been passed to the routine as parameters to the correct registers to be used with `out` instruction, this is performed in the third part of the routine [6]. The following is the code of the routine `dev_read` which uses the instruction `in` to read the data from a given port and return them to the caller, its prototype can be imagined as `char dev_read( int port )`.

```
dev_read:
    push edx
```

[4] Which are as you know parts of the registers `eax` and `edx` respectively.

[5] To my best knowledge its more performant than the normal way of using `mov`.

[6] The readers who have previous knowledge in x86 assembly programming may notice that I've omitted the epilogue of routines which creates a new stack frame, this decision has been made to make the matters simpler and you are absolutely free to use the calling convention.

```
xor edx, edx
xor eax, eax

mov dx, [esp + 8]

in al, dx

pop edx

ret
```

For the same reason of restoring the previous environment when returning to the caller, the routine pushes the value of `edx` into the stack, then both of `edx` and `eax` are cleared since they will be used by the instruction `in`. After that, the value of the passed parameter which represents the port number that caller wishes to read from, is stored in `dx`. Finally, `in` is called, the result is stored in `al`, the previous value of `edx` is restored and the routine returns. You may ask, why did we only stored and restored the previous value of `edx` while the register `eax` is also used also, why didn't we store and restore the previous value of `eax`? The reason is that `dev_read` is a function that returns a value, and according to `cdecl` convention the returned values should be stored in the register in `eax`, so, the value of `eax` is intended to be changed when return to the caller, therefore, it will not be correct, logically, to restore the the previous value of `eax` when `dev_read` returns. The ultimate goal of defining both `dev_write` and `dev_read` is to make them available to be used in C code, so, the lines `global dev_write` and `global dev_read` should be written in the beginning of `starter.asm`.

The Driver

One ATA bus in the computer's motherboard makes it possible to attach two hard disks into the system, one of them is called master drive which is the main one that the computer boots from, the other disk is known as slave drive. Usually, a computer comes with two ATA buses instead of just one, which means up two four hard disks can be attached into the computer. The first one of those buses is known as the primary bus while the second one is known as the secondary bus. The port numbers that are used to communicate with the devices that are attached into the primary bus start from `0x1F0` and ends in `0x1F7` each one of them has its own functionality while the ports number from `0x170` to `0x177` are used to communicate with devices that are attached into the secondary bus, so, there are eight ports for each ATA bus. The terms that combine a bus name and a device name are used to specify exactly which device is being discussed, for example, primary master means the master hard disk that is connected to the primary bus while secondary slave means the slave hard disk which is connected to the secondary bus. For the sake of simplicity, our device driver is going to assume that there is only a primary master and all read and write requests should be oriented to this primary master, therefore, our device driver uses the port number `0x1F0` as the base port to send the commands via PIC. You may ask, why are we calling this port number a base port? As you know that all the following port numbers are valid to communicate with the primary ATA bus: `0x1F0`, `0x1F1`, `0x1F2`, `0x1F3`, `0x1F4`, `0x1F5`, `0x1F6`, `0x1F7`, we can add any numbers from `0` through `7` to the base port number of the primary bus `0x1F0` to get a correct port number to communicate with the primary bus, the same is correct with the secondary ATA bus which its base port number is `0x170`. So, we can define the base port as a macro [7] as we will see in our device driver, and the we can use this macro by adding a specific value to it from `0` through `7` to get a specific port, the advantage of doing so is the easiness of changing the value of the base port to another port without the need of changing the code itself. Before starting in the implementation of the driver, let's create new two files: `ata.h` and `ata.c` which will contain the code of the device driver which communicates to ATA devices and provides an interface for the rest of the kernel to write and read data from the disk. The following is the content of `ata.h` and the details will be discussed in the next subsections.

```
void *read_disk( int );
void write_disk( int, void * );

void *read_disk_chs( int );
void write_disk_chs( int, void * );
```

[7] Or even variable