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). 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_out( int port, int cmd )`.

```
dev_write:
    ; Part 1
    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
```