

---

## I/O Devices

Before delving into the main content of this part of the notes (File Systems), we first introduce the concept of an **input/output (I/O) device** and show how the operating system might interact with such an entity. I/O is quite critical to computer systems, of course; imagine a program without any input (it produces the same result each time); now imagine a program with no output (what was the purpose of it running?). Clearly, for computer systems to be interesting, both input and output are required.

### 26.1 System Architecture

To begin our discussion, let's look at the structure of a typical system. Figure 26.1 presents a schematic.

The picture shows a single CPU attached to the main memory of the system via some kind of **memory bus** or interconnect. Some devices are connected to the system via a general **I/O bus**, which in many modern systems would be **PCI** (or one of its many derivatives); graphics and some other higher-performance I/O devices might be found here. Finally, even lower down are one or more of what we call a **peripheral bus**, such as **SCSI**, **ATA/IDE/etc.**, or **USB**. These connect the slowest devices to the system, including **disks**, **mice**, and other similar components.

One question you might ask is: why do we need a hierar-

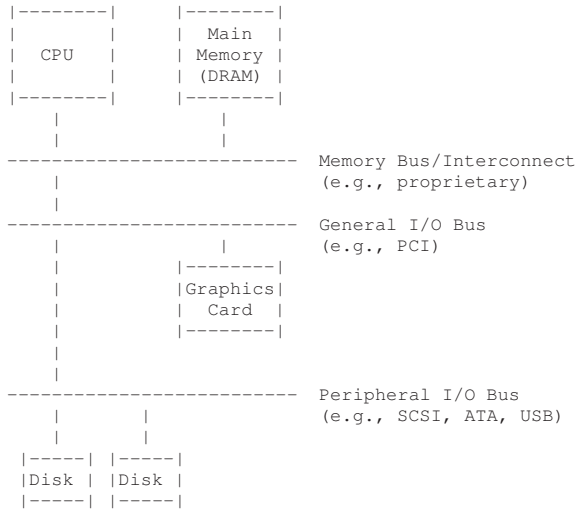


Figure 26.1: Prototypical System Architecture.

chical structure like this? Put simply: physics, and cost. The faster a bus is, the shorter it must be; thus, a high-performance memory bus does not have much room to plug devices and such into it. In addition, engineering a bus for high performance is quite costly. Thus, system designers have adopted this hierarchical approach, where components that demands high performance (such as the graphics card) are nearer the CPU. Lower performance components are further away. The benefit of placing disks and other slow devices on a peripheral bus are many; in particular, one can place many many devices on such a slow bus (PCI, for example, might limit one to only a few devices).

## 26.2 A Canonical Device

Let us now look at a canonical device (not a real one), and use this device to drive our understanding of some of the machinery required to make device interaction efficient. Figure 26.2 presents the device.

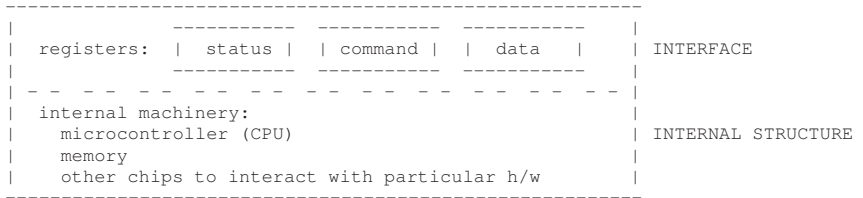


Figure 26.2: A Canonical Device.

From the figure, we can see that a device has two important components. The first is the hardware **interface** it presents to the rest of the system. Just like a piece of software, hardware must also present some kind of interface that allows the system software to control its operation. Thus, all devices have some specified interface and protocol for typical interaction.

The second part of any device is its **internal structure**. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Very simple devices will have one or a few hardware chips to implement their functionality; more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done. For example, modern RAID controllers might consist of hundreds of thousands of lines of **firmware** (i.e., software within a hardware device) to implement various RAID functionality (we'll learn more about RAID later).

### 26.3 The Canonical Protocol

In the picture above, the (simplified) device interface is com-

prised of three registers: a **status** register, which can be read to see the current status of the device; a **command** register, to tell the device to perform a certain task; and a **data** register to pass data to the device, or get data from the device. By reading and writing these registers, the operating system can control device behavior.

Let us now describe a typical interaction that the OS might have with the device in order to get the device to do something on its behalf. The protocol is as follows:

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

The protocol has four steps. In the first, the OS waits until the device is ready to receive a command by repeatedly reading the status register; we call this **polling** the device (basically, just asking it what is going on). Second, the OS sends some data down to the data register; one can imagine that if this were a disk, for example, that multiple writes would need to take place to transfer a disk block (say 4KB) to the device. When the main CPU is involved with the data movement (as in this example protocol), we refer to it as **programmed I/O (PIO)**. Third, the OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command. Finally, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished (it may then get an error code to indicate success or failure).

This basic protocol has the positive aspect of being simple and working. However, there are some inefficiencies and inconveniences involved. The first problem you might notice in the protocol is that polling seems inefficient; specifically, it wastes a great deal of CPU time just waiting for the (potentially slow)

device to complete its activity, instead of switching to another ready process and thus better utilizing the CPU.

#### THE CRUX OF THE PROBLEM: POLLING

How can the OS check device status without frequent polling, and thus lower the CPU overhead required to manage the device?

## 26.4 Lowering CPU Overhead with Interrupts

The invention that many engineers came upon years ago to improve this interaction is something we've seen already: the **interrupt**. Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a pre-determined **interrupt service routine (ISR)** or more simply an **interrupt handler**. The handler is just a piece of operating system code that will finish the request (for example, by reading data and perhaps an error code from the device) and wake the process waiting for the I/O, which can then proceed as desired.

Interrupts thus allow for **overlap** of computation and I/O, which is key for improved utilization. This timeline shows the problem:

```
CPU  1111111111pppppppppp1111111111
Disk -----1111111111-----
```

In the diagram, process 1 runs on the CPU for some time (indicated by repeated a 1 on the CPU line), and then issues an I/O request to the disk to read some data. Without interrupts, the system simply spins, polling the status of the device repeatedly until the I/O is complete (indicated by a p). The disk then services the request and finally process 1 can run again.

If instead we utilize interrupts and allow for overlap, the OS can do something else while waiting for the disk:

```
CPU  11111111112222222222211111111111
Disk -----1111111111-----
```

In this example, the OS runs process 2 on the CPU while the disk services 1's request. When the disk request is finished, an interrupt occurs, and the OS wakes up 1 and runs it again. Thus, *both* the CPU and the disk are properly utilized during the middle stretch of time.

Note that using interrupts is not *always* the best solution. For example, imagine a device that performs its tasks very quickly: the first poll usually finds the device to be done with task. Using an interrupt in this case will actually *slow down* the system: switching to another process, handling the interrupt, and switching back to the issuing process is expensive. Thus, if a device is fast, it may be best to poll; if it is slow, interrupts, which allow overlap, are best. If the speed of the device is not known, or sometimes fast and sometimes slow, it may be best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts. This **two-phased** approach may achieve the best of both worlds.

#### WARNING: INTERRUPTS NOT ALWAYS BETTER THAN PIO

Although interrupts allow for overlap of computation and I/O, they only really make sense for slow devices. Otherwise, the cost of interrupt handling and context switching may outweigh the benefits interrupts provide. There are also cases where a flood of interrupts may overload a system and lead it to livelock [MR96]; in such cases, polling provides more control to the OS in its scheduling and thus is again useful.

Another reason not to use interrupts arises in networks [MR96]. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**, that is, find itself

only processing interrupts and never allowing a user-level process to run and actually service the requests. For example, imagine a web server that suddenly experiences a high load due to the “slashdot effect”. In this case, it is better to occasionally use polling to better control what is happening in the system and allow the web server to service some requests before going back to the device to check for more packet arrivals.

## 26.5 More Efficient Data Movement with DMA

Unfortunately, there is one other aspect of our canonical protocol that requires our attention. In particular, when using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU is once again overburdened with a rather trivial task, and thus wastes a lot of time and effort that could better be spent running other processes. This timeline illustrates the problem:

```
CPU    1111111111cccccc2222222222111111111111
Disk   -----1111111111-----
```

In the timeline, process 1 is running and then wishes to write some data to the disk. It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked *c* in the diagram). When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else.

### THE CRUX OF THE PROBLEM: PROGRAMMED I/O OVERHEADS

With PIO, the CPU spends too much time moving data to and from devices by hand. How can we offload this work and thus allow the CPU to be more effectively utilized?

The solution to this problem is something we refer to as **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate trans-

fers between devices and main memory without much CPU intervention.

The process is as follows. To transfer data to the device, for example, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to. At that point, the OS is done with the transfer and can proceed with other work. When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete. The revised timeline:

```

CPU      1111111111222222222222222222222211111111111111
DMA      -----cccccc-----
Disk     -----11111111111-----

```

From the timeline, you can see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run process 2. Process 2 thus gets to use more CPU before process 1 runs again.

## 26.6 Methods of Device Interaction

Now that we have some sense of the efficiency issues involved with performing I/O, there are a few other problems we need to handle to incorporate devices into modern systems. One problem you may have noticed thus far: we have not really said anything about how the OS actually communicates with the device! Thus, our problem:

THE CRUX OF THE PROBLEM:  
 HOW TO COMMUNICATE WITH DEVICES  
 How should the hardware communicate with a device?  
 Should there be explicit instructions? Or are there other ways to do it?

Over time, two primary methods of device communication have developed. The first, oldest method (used by IBM main-

frames for many years) is to have explicit **I/O instructions**. These instructions specify a way for the OS send data to specific device registers and thus allow the construction of the protocols described above.

For example, on x86, the `in` and `out` instructions can be used to communicate with devices. For example, to send data to a device, the caller specifies a register with the data in it, and a specific *port* which names the device in question. Executing the instruction leads to the desired behavior.

Such instructions are usually **privileged**. The OS controls devices, and the OS thus is the only entity allowed to directly communicate with them. Imagine if any program could read or write the disk, for example: total chaos (as always), as any user program could use such a loophole to gain complete control over the machine.

The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, then, the OS would issue a load (to read) or a store (to write) to that address; the hardware then routes the load/store to the device instead of main memory.

There is not some great advantage to one approach or the other. The memory-mapped approach is nice in that no new instructions are needed to support it, but both approaches are still in use today.

## 26.7 Fitting into the OS: The Device Driver

One final problem we will discuss: how to fit devices, each of which have very specific interfaces, into the OS, which we would like to keep as general as possible. For example, consider a file system. We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drives, and so forth, and we'd like the file system to be relatively oblivious to all of the details of how to issue a read or write request to these different types of drives. Thus, our problem:



Note that such encapsulation can have its downside as well. For example, if there is a device that has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused. This situation arises, for example, in Linux with SCSI devices, which have very rich error reporting; because other block devices (e.g., ATA/IDE) have much simpler error handling, all that higher levels of software ever receive is a generic `EIO` (generic IO error) error code; any extra detail that SCSI may have provided is thus lost to the file system [G08].

Interestingly, because device drivers are needed for any device you might plug into your system, over time they have come to represent a huge percentage of kernel code. Studies of the Linux kernel reveal that over 70% of OS code is found in device drivers [C01]; for Windows-based systems, it is undoubtedly a much higher percentage as Windows-based operating systems support many more devices. Thus, when people tell you that the OS has millions of lines of code, what they are really saying is that the OS has millions of lines of device-driver code. Of course, for any given installation, most of that code may not be active (i.e., only a few devices are connected to the system at a time). Perhaps more depressingly, as drivers are often written by “amateurs” (instead of full-time kernel developers), they tend to have many more bugs and thus are a primary contributor to kernel crashes [S03].

## 26.8 Case Study: A Simple IDE Disk Driver

I should do a case study of a simple IDE disk driver [L94]. Do you think this would be useful? Let me know.

## 26.9 Summary

You should now have a very basic understanding of how an OS interacts with a device. Two techniques, the interrupt and DMA, have been introduced to help with device efficiency, and

two approaches to accessing device registers, explicit I/O instructions and memory-mapped I/O, have been described. Finally, the notion of a device driver has been presented, showing how the OS itself can encapsulate low-level details and thus make it easier to build the rest of the OS in a device-neutral fashion.

## References

- [C01] "An Empirical Study of Operating System Errors"  
Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler  
SOSP '01
- [G08] "EIO: Error-handling is Occasionally Correct"  
Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Ben Liblit  
FAST '08
- [L94] "AT Attachment Interface for Disk Drives"  
Lawrence J. Lamers, X3T10 Technical Editor  
Available at: <ftp://ftp.t10.org/t13/project/d0791r4c-ATA-1.pdf>  
Reference number: ANSI X3.221 - 1994
- [MR96] "Eliminating Receive Livelock in an Interrupt-driven Kernel"  
Jeffrey Mogul and K. K. Ramakrishnan  
USENIX '96
- [S03] "Improving the Reliability of Commodity Operating Systems"  
Michael M. Swift, Brian N. Bershad, and Henry M. Levy  
SOSP '03