## Lecture 30 — Input/Output Devices & Drivers

Jeff Zarnett

Input/Output Devices

Though the computer's name and typical use kind of suggests the purpose of the machine is computation, input/output is just as important to the usefulness of a computer. A computer that takes no inputs and produces no outputs is, presumably, not very useful[1]. I/O is, unfortunately, a messy business. Whereas there are only a small number of CPU types your typical desktop OS might run (and the AMD and Intel processors, for example, can execute the same binary code), there are uncountably many I/O devices in the world. And they're all different. Keyboards, hard drives, printers, and headsets are all I/O devices, but they serve very different purposes and work in different ways. This diagram shows the dramatic differences in speed between various devices of a typical PC:
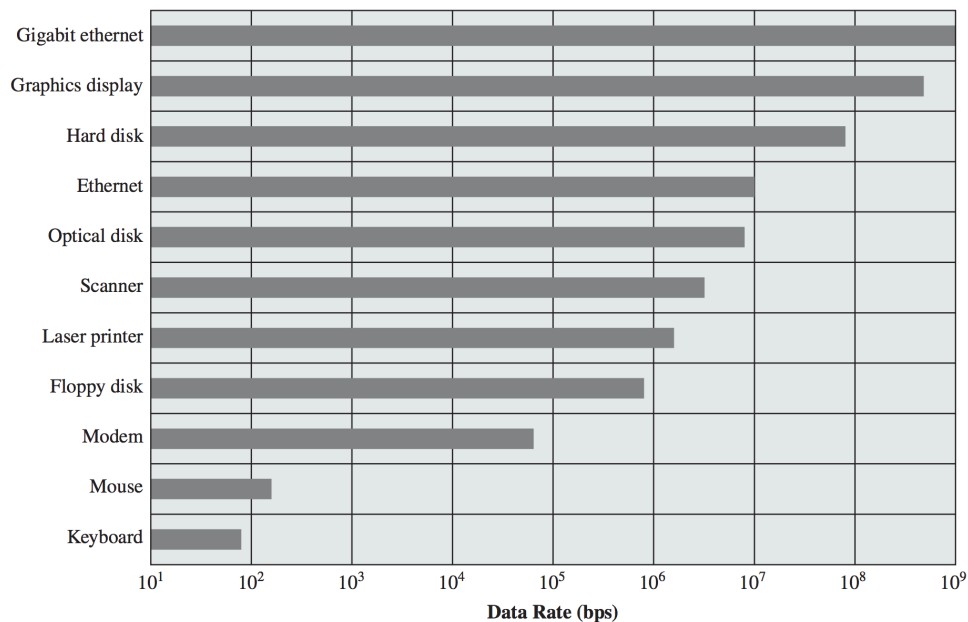
Typical I/O Device Data Rates [Sta14].

We might like to think that USB has taken away some of the complexity, but that's just the way a device connects to the computer; managing the device itself is still as complicated as ever. All of the examples just listed in the previous paragraph can be connected via USB (Universal Serial Bus).

Disk I/O, when discussing magnetic hard disks, has a huge impact on system performance that it will receive its own discussion. But first, a little discussion about I/O in general.

We will not be spending any time talking bout the specifics of the hardware (though it might be helpful to review that information on your own). For I/O the key parts are the bus (and a modern PC has several, including the PCIe bus, USB, etc.) and a controller (electronics to operate the hardware port, bus, device...). After that there will be a system or protocol for how the devices will communicate, whether it is polling, interrupts, or DMA (Direct Memory Access) as discussed earlier. With that out of the way, let us examine devices at a higher level.

---

[1]Schrödinger's computer: did the program actually run if there was no output to observe?

Application I/O Interface

Ideally, when a general-purpose operating system is written, it will accept new devices being added to the system without editing/reinstalling/recompiling the code. Your experience with object oriented programming gives you some familiarity with the solution of how we should accomplish this goal. We want to abstract away the details of the hardware, to the extent we can, and provide a uniform *interface* to interact with.

In the very early days of operating systems, the hardware the computer shipped with was all the hardware it ever supported. If the vendor came out with a new module, they would have a new operating system update to introduce support for that device. This got to be unmanageable in the era of the IBM PC because anybody could create hardware and attach it via a standard interface. Relying on IBM or Microsoft or whoever your OS vendor was to implement support for a random piece of hardware was not realistic.

Operating system developers thought they were very clever. They realized that they could shift the work to the hardware developers through a concept called *device drivers*. The device driver plugs in to the operating system through a standard interface and tells the operating system a bit about the hardware and translates commands from the operating system to hardware instructions. Where this fell apart was the fact that hardware developers often made extremely poor drivers (whether this was due to inability or lack of caring is not clear).

The problem was exacerbated by a Windows design decision that device drivers would run in the system at the same protection level as the kernel. Some other operating systems have user-mode drivers, where possible (in Mac OS X and Linux systems, for example, there is the ability to have file system drivers in user space through a program called FUSE) or at an intermediate level between that of user space and the kernel. In any event, in Windows, if a device driver is unhappy with the current state of the system, contains a programming error, or encounters a situation it cannot handle, the driver can invoke a system call that brings up everyone's favourite feature: the Blue Screen of Death (BSOD).

Microsoft, rightly or wrongly, was blamed for a lot of those blue screens of death. They took two approaches to remedy this problem. One was to write and include in Windows a lot more device drivers. The other was to introduce the static driver verifier, a complex piece of static analysis software used to test, at compile/build time, whether the driver will behave badly. Passing this test is required to get a sticker of approval from Microsoft to put on the box. So, the battle rages on about who is responsible for writing the drivers, unfortunately.

Device drivers, when they exist, connect into the kernel's I/O subsystem to mediate between the kernel's I/O subsystem and the hardware device controller. See the diagram:
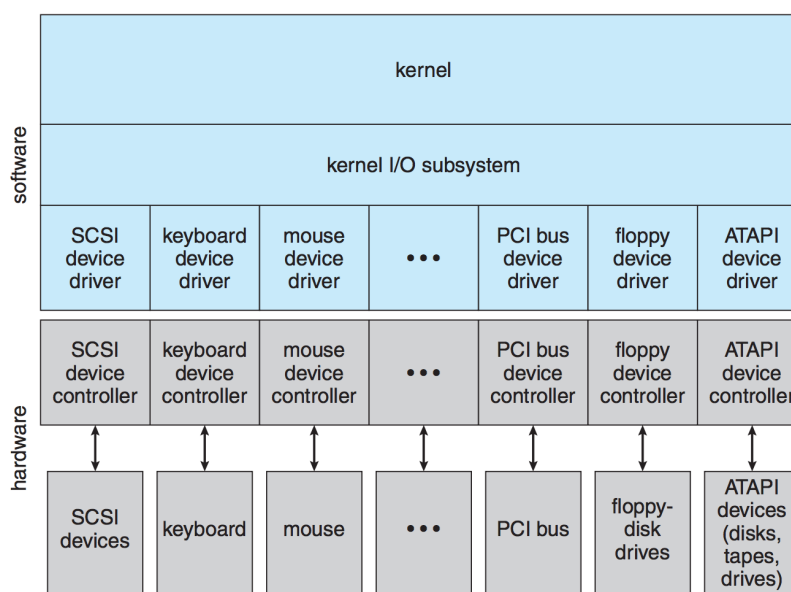
Kernel I/O structure, showing the placement of device drivers [SGG13].

Abstracting away details of the hardware makes the job of the operating system developer easier: the interface provides a standard way of interacting with the hardware device. Part of the difficulty is that devices can vary on numerous dimensions, as we can see through a few examples from [SGG13]:

- **Data transfer mode**. A device may operate on a character level (transfer one character at a time) like a keyboard, or may operate on a block (group of bytes) all at once, such as a disk operating on a 4 KB block.

- **Access method**. A device may allow only sequential access (that is, reading only the next element or piece of data or writing to the next location) such as a modem, or it could allow random access (reading from or writing to any point) as in a CD (compact disc).

- **Transfer schedule**. The schedule may be synchronous (transferring data with predictable response times) as in the case of reading from tape, or asynchronous (with unpredictable response times), such as awaiting a response that will come over the network.

- **Dedication**. A device may be shareable (allows multiple concurrent threads) such as the speakers, or dedicated (allowing only one thread to use it at a time), such as a printer.

- **Device Speed**. Recall the diagram from earlier showing the dramatic differences in transfer speeds.

- **Transfer Direction**. Some devices are input only (such as a temperature sensor), other devices are output only (speakers), and some devices are capable of both (hard disk).

We would like to, as much as possible, keep the details above from the operating system, though devices will typically be grouped into a few categories so appropriate system calls can be issued. If a device is block-oriented, the OS should be issuing block read and write commands, not trying to do it one character at a time.

Operating systems also usually have an *escape* system call that allows passing of a command directly from an application or the kernel to a device driver. This allows us to issue commands to a device that the OS designers have not thought of and created system calls for. The UNIX system call for this is `ioctl` ("I/O Control") and it takes three parameters: a file descriptor indicating the hardware device (everything in UNIX is a file, remember), the command number (an integer), and a pointer to an arbitrary data structure (presumably `void*`) that has any necessary control information and data [SGG13].

Block and Character I/O

The block device interface is used for block devices such as hard disk drives. Any device will support `read` and `write` commands, and if it is a random-access device, it will have a `seek` command to jump to a specific block. An application usually accesses the hard disk through the file system, but going even one level lower, the OS can work on the hard drive using these two/three commands without being concerned with how that command is actually transmitted to the physical hardware.

We can abstract things a little bit further, from the perspective of the application developer, by having a memory-mapped file. Then, rather than using the block oriented operations directly, the application just writes to and reads from "memory" and the OS handles the behind-the-scene coordination to make writes go out to the correct block and read from the correct block.

A character-oriented device is something like the keyboard; the system calls are `get` and `put`. Libraries and other structures may exist to work on a whole line at a time (compare to, for example, the Java constructs for reading and writing files: the default streams are one character at a time, but if you want anything other than terrible performance, use a buffered reader which can read a whole line). This is a good match for input devices that produce data in small amounts and ant unpredictable times, and perhaps for printers and sound output which operate naturally on a linear stream of bytes [SGG13].

Buffering

Regardless of whether a device is block- or character-oriented, the operating system can improve its performance through the use of buffering. As you may have experienced with Java, the use of a buffer speeds things up quite a lot. A buffer is nothing more than an area of memory that stores data being transferred, whether it is from memory to a device, device to memory, or device to device.

A buffer is a good way to deal with a speed mismatch between devices. Users type very slowly, from the perspective of the computer, and it would be awfully inefficient to ask the disk, a block oriented device, to update itself on every single character. It is much better if we wait until we have some certain amount of data (e.g., a whole line or whole block or full buffer, whichever it is) and then write this out to disk all at once.

The write is, however, not instantaneous and in the meantime, the user can still keep typing. Thus, to solve this, the typical solution is *double buffering*, that is, two buffers. While buffer one is being emptied, buffer two receives any incoming keystrokes. Double buffering decouples the producer and consumer of data, helping to overcome speed differences between the two [SGG13].

Network Devices

Network devices are fundamentally different from those that are directly attached to the system. Thus, the `read`, `write`, and `seek` routines are not really appropriate. The model we are most familiar with from UNIX and Windows is that of *sockets*. As the name suggests, the analogy to consider is electrical sockets (the ones you plug laptop chargers into, for example). We end up with a client-server model of communication: the server "opens a socket" to which the client "plugs in" when it wants to connect. This is exactly what happens when the ssh daemon is started; it opens a socket on the machine using port 22, and when a user wants to connect to that server via ssh, his or her ssh client connects to that socket. Then, packets of data can be exchanged between client and server.

To support servers with multiple clients, the socket interface has a function `select`, that manages a set of sockets. Invoking this function returns information about what sockets have a packet waiting to be received and which are available for sending a packet. Proper use of `select` eliminates polling and busy-waiting in a situation where delays are unpredictable (which is always the case with the network).

Spooling and Reservations

A *spool* is a buffer for a device, like a printer, that can serve only one job at a time. Unlike disk, where it might read for P_1 now and then write for P_2 immediately afterwards, a printer needs to finish a whole print job before it starts the next. Printing is the most obvious example, but it is by no means the only one. The operating system centralizes all communication to the printer and runs it through the spooling system.

I/O Protection

Recall from much earlier the idea of kernel mode and user mode instructions; some processor instructions are restricted such that only the OS can issue them (in kernel mode), otherwise it is an error (illegal instruction). We want all user accesses to I/O to be mediated through the operating system, so the OS can check to see if the request is valid. If it is, allow it to proceed; otherwise terminate the process with an error. This helps minimize errors and problems where people do bad things like cancel another process's request so theirs can go first. Our typical tradeoff: increased safety in exchange for reduced performance.

In certain circumstances, for performance reasons, we might want to allow direct access. In the case of a game, for example, we want to allow the game to work directly on the graphics card's memory (despite the fact that it is an I/O device). Mediating every access through the kernel would result in unacceptably poor performance (which is to say, the other team would totally call us noobs). So a section of graphics memory, corresponding to a particular window on the screen, will be locked by the kernel to be accessible by the game's process [SGG13].

Kernel I/O Data Structures
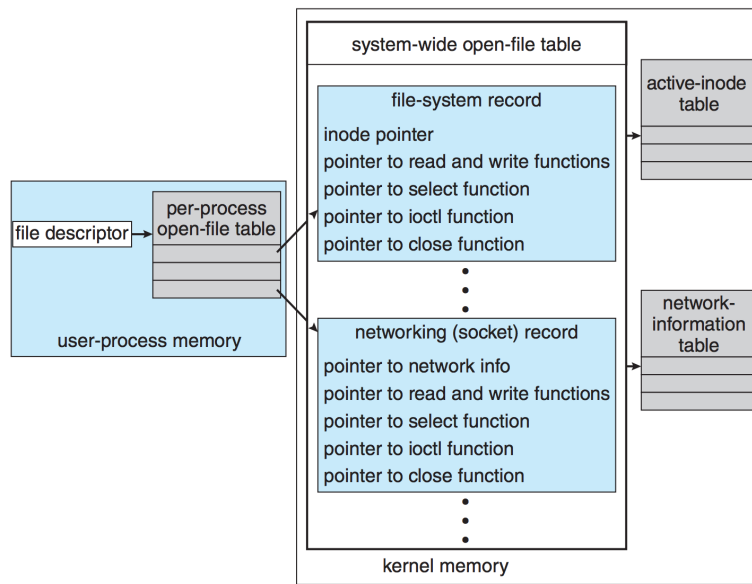
The UNIX structures to manage I/O [SGG13].

I/O Scheduling

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.