Lecture 30 — Input/Output Devices & Drivers

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 1, 2015

The Importance of I/O

Though the computer's name and typical use suggests the purpose of the machine is computation, input/output is just as important to the usefulness.

A computer that takes no inputs and produces no outputs is not very useful.

Schrödinger's computer.

I/O is Messy

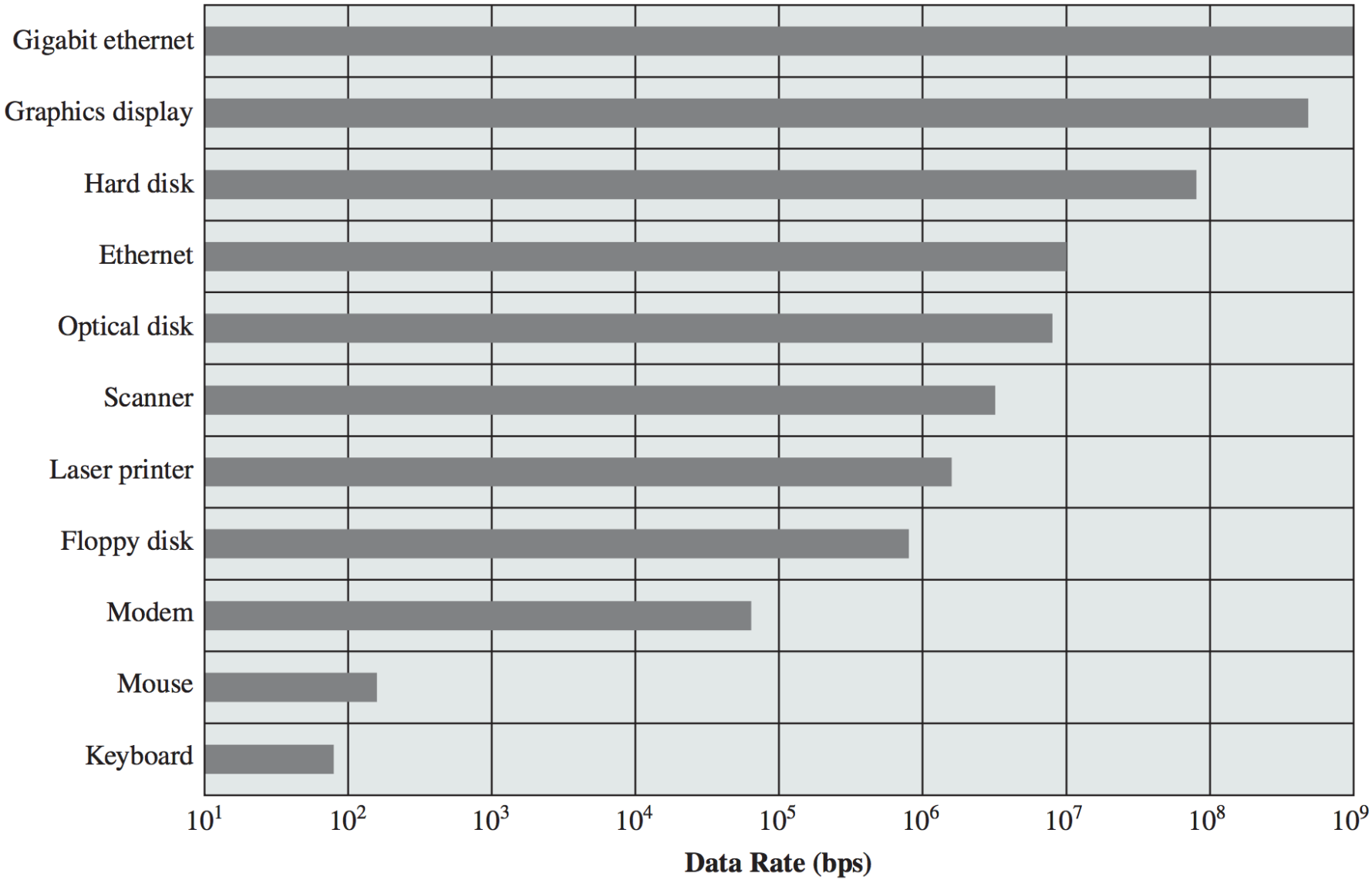
I/O is, unfortunately, a messy business.

There are only a small number of CPU types your typical desktop OS might run (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.

I/O Device Rates



I/O is Messy

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 can be connected via USB (Universal Serial Bus).

Disk vs General I/O

Disk I/O has a huge impact on performance; it will receive its own discussion.

But first, a little discussion about I/O in general.

We will not be spending any time talking about 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 controller.

After that there will be a protocol for how the devices will communicate
e.g., polling, interrupts, or DMA.

Application I/O Interface

Ideally a general-purpose operating system 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.

Provide a uniform interface to interact with.

I/O Interfaces

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.

I/O Interfaces

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.

It tells the operating system a bit about the hardware and translates commands from the operating system to hardware instructions.

Hardware developers often made extremely poor drivers.

Device Drivers

The problem was exacerbated by a Windows design decision.

Device drivers run in the system at the same protection level as the kernel.

Some other operating systems have user-mode drivers, where possible or at an intermediate level between that of user space and the kernel.

In Windows, a driver can invoke a system call that brings up everyone's favourite feature: the Blue Screen of Death (BSOD).

Blue Screen of Death

Microsoft, rightly or wrongly, was blamed for a lot of those BSODs.

They took two approaches to remedy this problem.

One was to write and include in Windows a lot more device drivers.

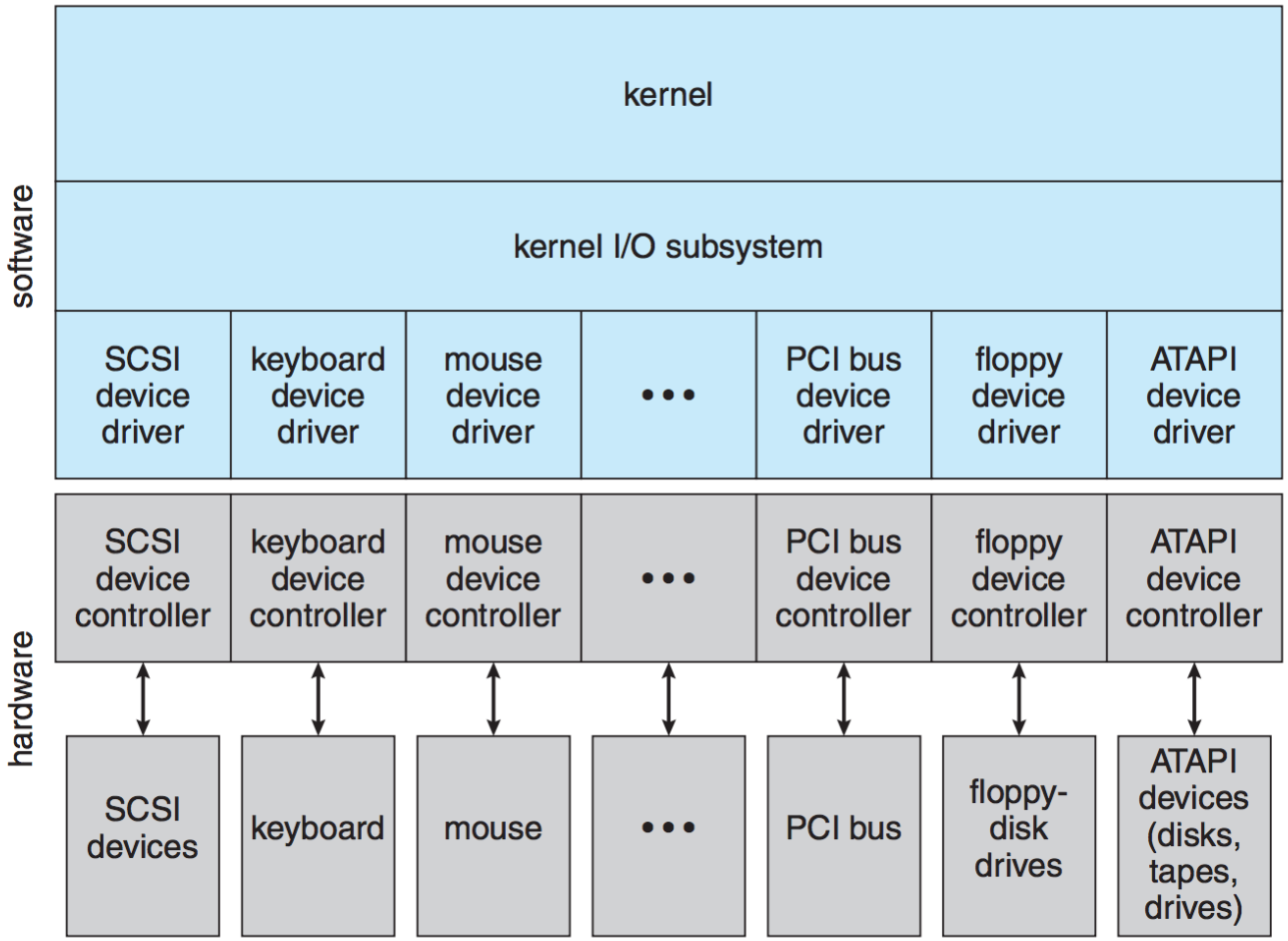
The other was to introduce the static driver verifier; software used to test, at compile/build time, whether the driver will behave badly.

Passing this test is required to get a sticker of approval from Microsoft.

The battle rages on about who is responsible for writing the drivers.

Device Drivers

Device drivers connect into the kernel's I/O subsystem to mediate between the kernel's I/O subsystem and the hardware device controller.



Abstraction

Abstracting away details of the hardware makes the job of the OS dev easier.

Part of the difficulty is that devices can vary on numerous dimensions:

- **Data transfer mode**
- **Access method**
- **Transfer schedule**
- **Dedication**
- **Device Speed**
- **Transfer Direction**

Abstraction

We would like to, as much as possible, keep the details above from the OS.

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.

Escape

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").

It takes three parameters:

- A file descriptor indicating the hardware device.
- The command number
- A pointer to an arbitrary data structure that has control info & data.

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.

The OS can work on the hard drive using these 2/3 commands without being concerned with how that command is actually transmitted.

Block and Character I/O

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”.

The OS handles the behind-the-scene coordination to make writes go out to the correct block and read from the correct block.

Block and Character I/O

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 (Java).

This is a good match for input devices that produce data in small amounts and at unpredictable times.

Perhaps also for printers and sound output which operate naturally on a linear stream of bytes.

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.

A buffer is nothing more than an area of memory that stores data being transferred, from memory to a device, device to memory, or device to device.

Buffering

A buffer is a good way to deal with a speed mismatch between devices.

Users type very slowly, from the perspective of the computer.

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 and then write this out to disk all at once.

Buffering

The write is not instantaneous and in the meantime, a user can still keep typing.

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.

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.

Socket To Me

Analogy to consider is electrical sockets.

We end up with a client-server model of communication.
    The server “opens a socket” to which a client “plugs in” to connect.

This is exactly what happens when the ssh daemon is started.
    The daemon opens a socket on the machine using port 22.

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.

Socket To Me

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 & Reservations

A spool is a buffer for a device, like a printer, that can serve only 1 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.

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.

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.

I/O Protection and Direct Access

For performance reasons, we might want to allow direct access.

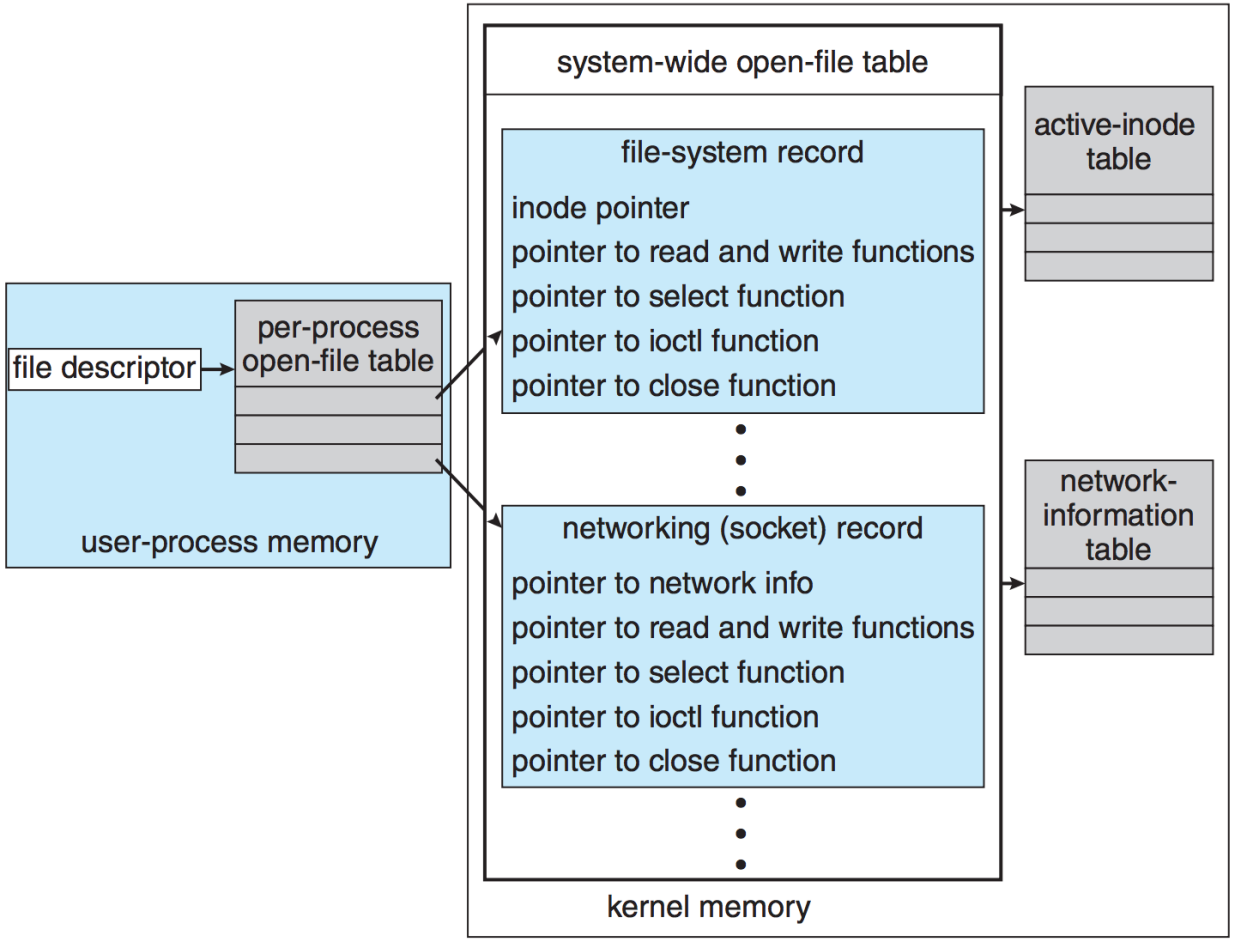
In the case of a game, for example, we want to allow the game to work directly on the graphics card's memory (despite the fact that it is an I/O device).

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.

Kernel I/O Data Structures

The kernel must, as is rather obvious, keep track of what I/O devices are in use by which processes and the general state of the I/O device.



I/O Scheduling

Sometimes we want to schedule I/O requests in some order other than First-Come, First-Served.

Analogy: you need to go to the grocery store, the dry cleaners, and the bank.

The bank is 1 km to the west of your current location; the grocery store is 3 km west; and dry cleaners is in the same plaza as the grocery store.

It would be fine to go to the bank, then the dry cleaners, then the grocery store, but not to go to the dry cleaners, then the bank, then the grocery store.

The unnecessary back-and-forth wastes time and energy.

I/O Scheduling

The operating system will want to do something similar with I/O requests.

Maintain requests structure; re-arrange them to be accomplished efficiently.

This will, naturally, have some limits: requests should presumably get scheduled before too much time has elapsed even if it would be “inconvenient”.

It might also take priority into account.

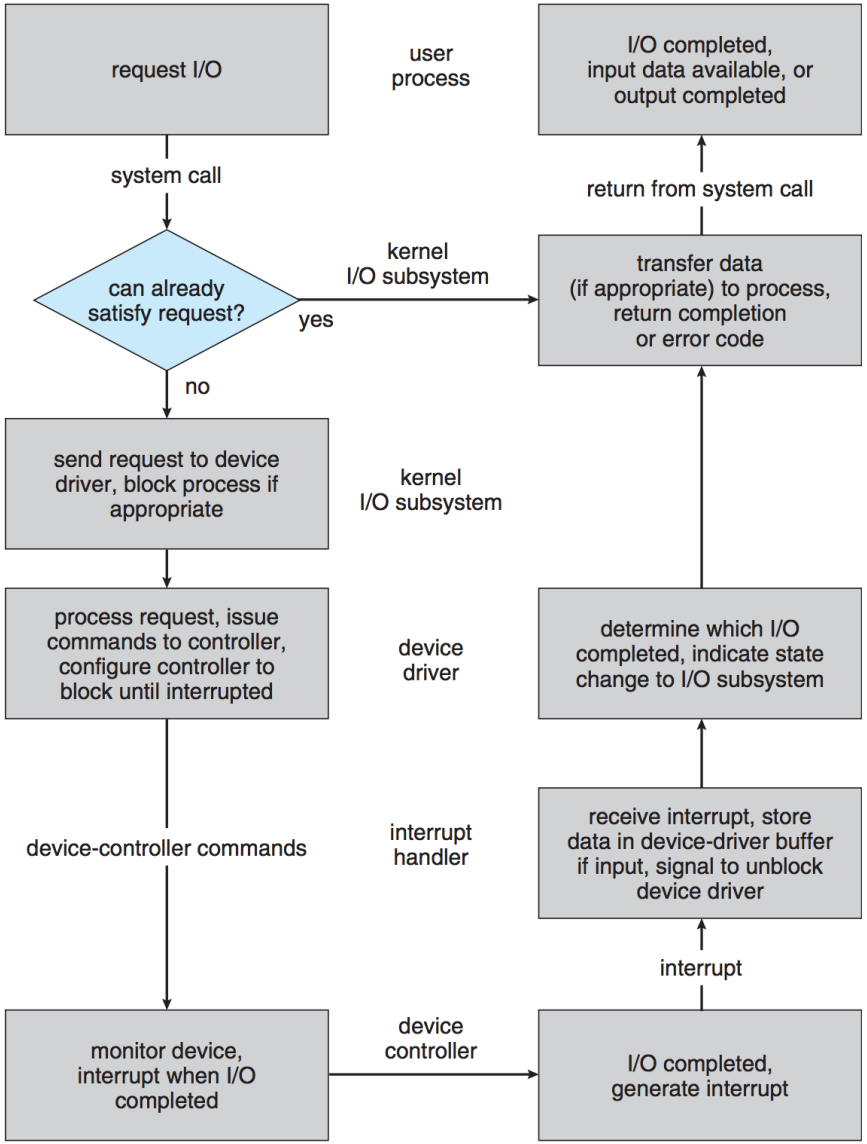
Transforming I/O Requests to Hardware Operations

Reading from the file system on disk, for example, requires a few steps.

If I want to open a file like `example.txt`, the file system will associate this file with some information about where on the disk this is.

Then, to read the file, `read` commands can be issued to get those blocks into memory so I can edit it with `vi`.

I/O Request Life Cycle



I/O Request Life Cycle

1. A process issues a `read` command (assume the file is already open).
2. The system call routine checks the parameters for correctness. If the data is in a cache or buffer, return it straight away.
3. Otherwise, the process is blocked waiting for the device and the I/O request is scheduled. When the operation is to take place, the I/O subsystem tells the device driver.
4. The device driver allocates a buffer to receive the data. The device is signalled to perform the I/O (usually by writing into device-control registers or sending a signal on a bus).
5. The device controller operates the hardware to do the work.

I/O Request Life Cycle

6. The driver may poll for status, await the interrupt when finished, or for the DMA controller to signal it is finished.
7. The interrupt handler receives the interrupt and stores the data; it then signals the device driver to indicate the operation is done.
8. The device driver identifies what operation has just finished, determines the status, and tells the I/O subsystem it is done.
9. The kernel transfers the data (or error code or whatever) to the address space of the requesting process, and unblocks that process.
10. When the scheduler chooses that process, it resumes execution.