I/O and Interrupts

CISC 3320 — CUNY Brooklyn College
Lecture Notes

I/O and Interrupts

In this chapter, we will:

1. Define what interrupts are and explain how they are implemented.
2. Provide details on how I/O devices connect to other hardware devices.
3. Elaborate on I/O handling methods (those we saw in the "Overview of Operating Systems" topic.)
4. Explain how user applications interact with I/O devices.
5. Describe the I/O services provided by the kernel.
6. Illustrate how I/O service requests are translated into hardware actions.

Interrupts

An **interrupt** is a signal that a hardware device issues to the CPU to request the OS to pause the current execution of code.

A device could issue an interrupt in case of an error (such as dividing by 0), but also to indicate that an I/O event is happening.

When the CPU is interrupted, it pauses the execution of code and instead executes a function called **service routine** whose purpose is to handle the interrupt. After the service routine returns, the CPU resumes the previous execution of code from the point where the interrupt occurred.

Since each interrupt must be handled differently (depending on the interrupt type,) the service routine uses an array called the **interrupt vector** that is stored in memory in order to find and call a **handler**, which is a program that handles a specific type of an interrupt. Each array index contains an address to a handler program.

Interrupts

The interrupt mechanism is implemented as follows:

1. A **device controller**, which is a piece of hardware in the device whose purpose is to control the device's I/O signals, sends a signal to the CPU via a wire called an **interrupt-request line**.
2. In-between the execution of some program's instructions, the CPU notices that the controller issued a signal, and reads the interrupt number (which indicates what the interrupt type is.)
3. The CPU uses this number as an index into the interrupt vector to call the appropriate interrupt handler.
4. The handler first saves the current execution state (so that the CPU could return executing code after the interrupt is handled,) finds the reason for the interrupt, performs all the actions needed to resolve the interrupt, restores the saved execution state, and returns.
5. The CPU now continues to execute the instructions of the program that was running before the interrupt occurred.

Interrupts

Here are various Intel interrupt types and their corresponding numbers in the interrupt vector:

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Intel interrupt types and the interrupt vector. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

Interrupts

Most CPUs have two types of interrupt request lines: a nonmaskable one and a maskable one, each of which is used to transfer signals for two types of interrupts.

A **nonmaskable** interrupt is one that a CPU cannot ignore, and must take measures in order to handle the interrupt. Events that could issue a nonmaskable interrupt are usually unrecoverable memory errors, such as division by zero.

A **maskable** interrupt might be ignored by the CPU when critical OS code that must not be interrupted is executed.

While executing such critical code, the CPU would turn off the maskable interrupt request lines, so that no interrupt signals are registered while such code is running.

In the diagram on the previous slide, all I/O-related interrupts have numbers ranging from 32 to 255. This is because all the described interrupts in numbers 0 to 31 are nonmaskable, while I/O interrupts are maskable and can be ignored by the CPU when needed.

Interrupts

Any questions?

I/O Hardware

Since handling I/O is a fundamental goal for an operating system, OS kernels are frequently developed with I/O subsystems within them. An **I/O subsystem** is, on the one hand, the collection of kernel services provided for I/O operations, and, on the other hand, the hardware used for I/O (busses, controllers, drives, and I/O devices.)

An I/O subsystem needs to take two apparently conflicting trends in technology into account:

1. The strive towards a unified interface for all the connected devices, regardless of how differently they operate. When using a standardized interface, an operating system facilitates the integration of new devices into the system.
2. The great variety in I/O devices, and the integration of devices with new technologies.

To achieve a uniform interface between the OS and I/O devices, the OS uses **device drivers**, each of which is software that provides access to and manages I/O on these devices.

I/O Hardware

An I/O device, which is a type of peripheral device, communicates with the OS with the use of signals that are sent via wires or through the air (in wireless communication.)

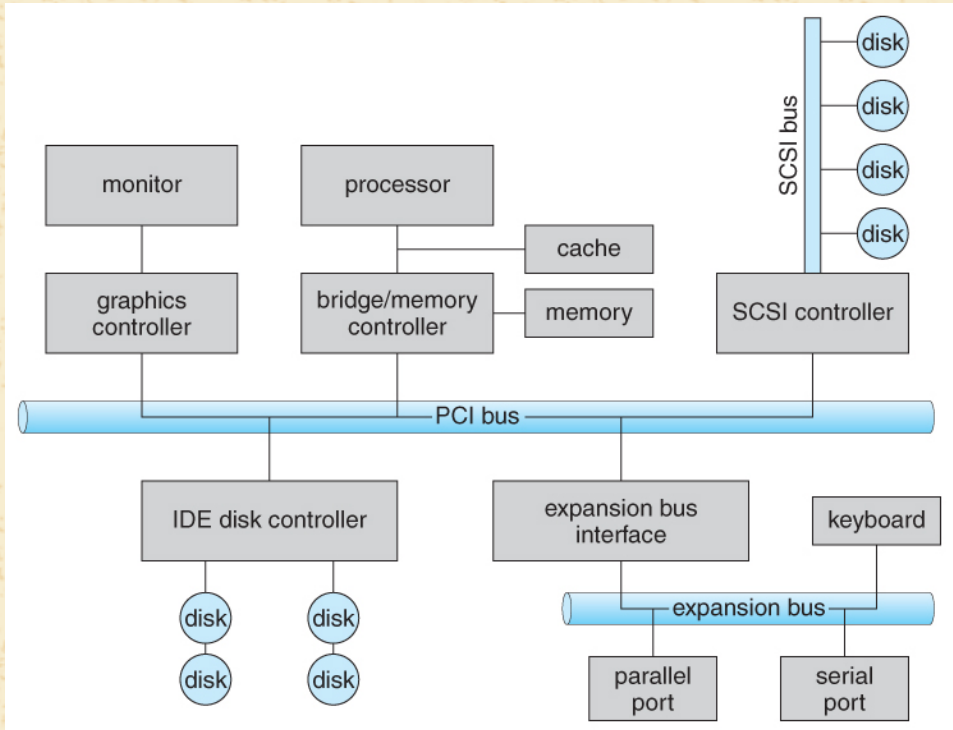
The physical point at which the device connects to a computer, in case of a wired connection, is called a **port**.

A **bus** consists of a collection of wires and a set of rules and procedures, called a **protocol**, that dictate how signals should be sent via these wires. Signals come in a form of electrical voltages transmitted through the wires at a defined timing.

A computer uses a variety of buses, which differ by speed, voltage, throughput (workload), and connection methods.

The following illustration shows a typical bus structure in an OS.

I/O Hardware



A typical PC bus structure. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

I/O Hardware

The buses that appear in the figure in the previous slide are:

1. A **PCIe** (Peripheral Component Interconnect Express) bus, whose purpose is to connect CPUs, cache, and memory to other devices, including disks and I/O devices. The devices connected to this bus are relatively fast.
2. An **expansion bus**, which connects I/O devices, like the keyboard or mouse. These devices are a bit slower than those connected directly with the PCIe bus.
3. A **serial-attached SCSI** (Small Computer System Interface) bus, or **SAS** bus, which moves data from the computer to secondary storage devices.

In that figure, disks are attached either via SAS or via IDE (Integrated Drive Electronics) technologies. SAS is faster than IDE, but IDE is a cheaper tech that supports a greater storage capacity. More on the differences between the two is described in the article at https://getprostorage.com/blog/scsi-vs-sata-vs-ide/.

I/O Hardware

As we defined it previously, a **device controller** is a piece of hardware in the device whose purpose is to control the device's transfer of I/O data.

To be able to communicate with the device, a CPU would use the **registers** located on the controller of the device.

A device's controller has minimally 4 registers:

1. The CPU reads data from the **`data-in` register** (has the size of 1 to 4 bytes.)
2. The CPU sends data to the device via the **`data-out` register** (has the size of 1 to 4 bytes.)
3. Each bit in the **`status` register** represents an aspect of the device's status (a bitmap!) These include whether a transaction was completed already or not, whether the device is sleeping, whether the device is ready, whether it is busy, and whether any errors occurred.
4. The CPU changes the current settings of the device (e.g., the length of a word) via the bits of the **`control` register**.

I/O Handling Methods

How can the CPU access an I/O device?

The CPU can use these registers in one of two ways:

1. **Port-mapped I/O** (or **Isolated I/O**): A CPU would issue special I/O instructions that are transferred via busses to the registers of the device controller of the device that is specified by the instructions.
2. **Memory-mapped I/O**: The contents of the device controller registers are copied (mapped) regularly to the main memory, so the CPU simply reads and writes to the memory addresses associated with the registers, which is faster than issuing I/O instructions. The controller, on the other hand, reads these memory locations to update its registers and writes data to memory if, for example, the device's status changes or when new I/O data is ready to be transferred.

Today, most I/O actions are performed by using memory-mapped I/O.

I/O Handling Methods

Professor Marilyn Wolf demonstrates how memory-mapped I/O works:

https://www.youtube.com/watch?v=NrYmieuWLy4

I/O Handling Methods

Any questions?

I/O Handling Methods

How does the CPU get notified of an I/O event?

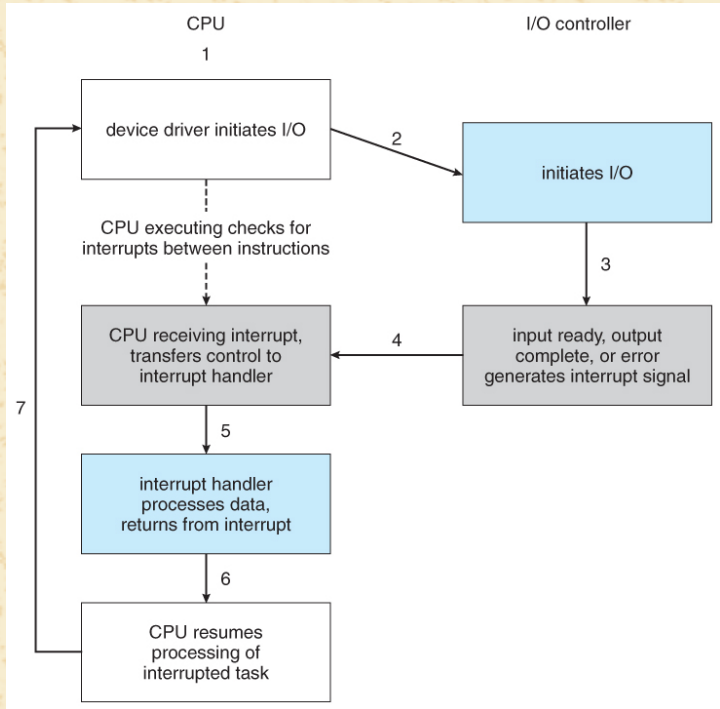
1. An OS is said to implement **Polling I/O** when the CPU continuously polls (= checks) whether an I/O device is ready to transfer data. Two main bits are involved in this procedure: the **`busy` bit** in the `status` register, and the **`command-ready` bit** in the `control` register. A typical output communication cycle (handshake) goes as follows:
   a. The CPU keeps looking into the `busy` bit until it turns to 0 (1 means busy and 0 means not busy anymore.)
   b. The CPU changes the `write` bit in the `control` register to 1, and writes the data to the `data-out` register.
   c. The CPU also changes the `command-ready` bit to 1 to indicate that it finished preparing data for the device.
   d. The device controller notices that `write` is on, so it reads the `data-out` register and performs the I/O action with the device (e.g., if the device is the screen, data is printed to the screen.)
   e. Finally, the controller puts 0 in the `command-ready` bit, puts 0 in the `error` bit to indicate that no errors happened, and puts 0 in the `busy` bit to indicate that the device is ready for new commands from the CPU.

I/O Handling Methods

1. Polling I/O is efficient when the device works fast. If the device is slow, however, the CPU will spend valuable time just waiting for the device to become ready, without being able to execute user programs. This phenomenon is called **busy waiting** or **polling**. The solution is to use interrupt-driven I/O.
2. With an **interrupt-driven I/O** approach, the CPU can perform other activities while waiting for the I/O device. After every instruction that the CPU executes, it will check if a signal came via one of its interrupt-request lines. If not, the CPU will continue to execute the next instruction. If yes, the service routine, which the CPU executes, will run an interrupt handler which will handle the I/O operation. We say that:
    - A device's controller ***raises*** an interrupt.
    - The CPU ***catches*** the interrupt and ***dispatches*** the interrupt handler.
    - The interrupt handler ***clears*** the interrupt by servicing the device.

    The flow chart on the next slide illustrates what occurs during an interrupt.

I/O Handling Methods



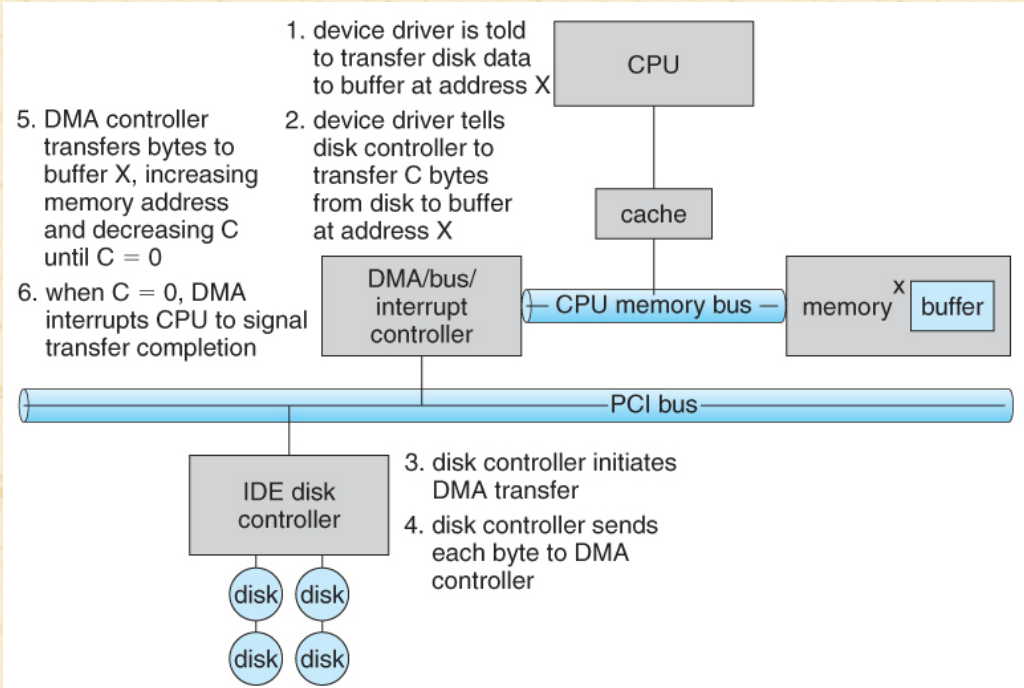
Interrupt-driven I/O cycle. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

I/O Handling Methods

How does the CPU read data from an I/O device?

1. When a CPU reads data one byte at a time, we say that the OS uses **Programmed I/O**. Here, the CPU gets direct access to each piece of the transferred data.
2. In order to 'free' the CPU from reading all the data, an OS might introduce an additional processor called a **direct-memory access controller** or **DMA controller** whose sole purpose is to manage I/O data. The process goes as follows:
   a. The main CPU issues a command to the DMA controller with the following info: (1) the device to which data should be transferred, (2) the location to the beginning of the data within memory, and (3) the total number of bytes to be transferred. Note that (2) and (3) tell precisely what chunk of data should be transferred.
   b. The DMA controller proceeds to handle the I/O, while the main CPU continues executing other instructions.
   c. When the DMA finishes its work, it raises an interrupt to inform the CPU that the data transfer is complete.

I/O Handling Methods



Steps in a DMA transfer. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

Application I/O Interface

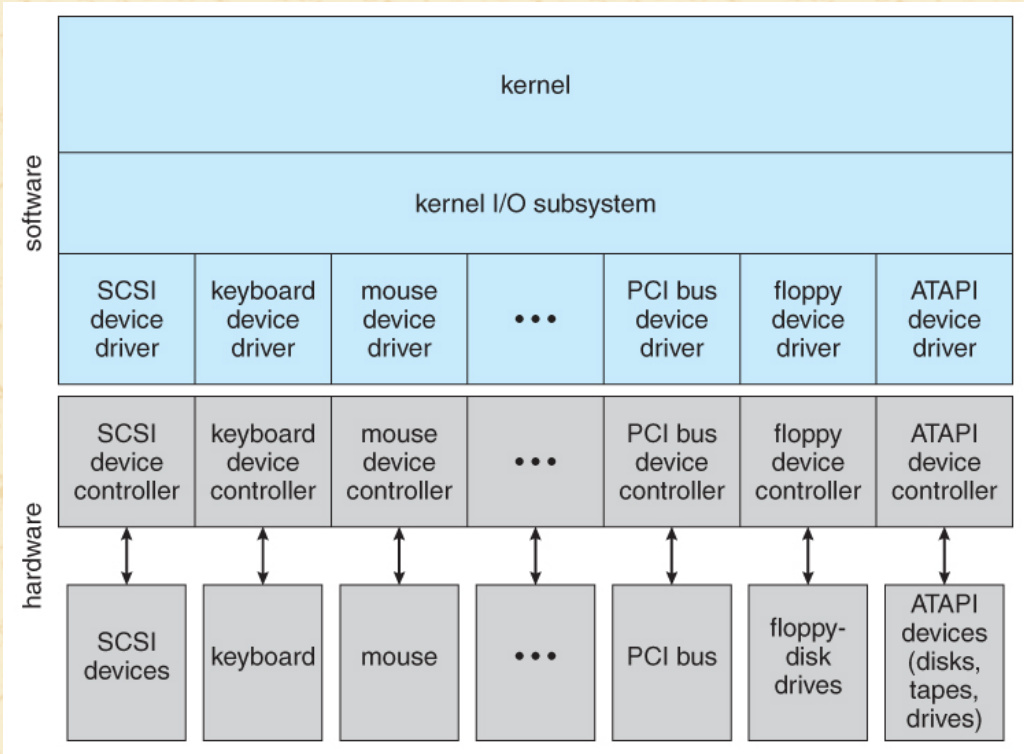
Since, as we emphasized before, devices can differ in various aspects (which we also learn in a few slides from now,) the OS strives to create a unified interface for all the devices, and hide the device-specific aspects from user applications.

To achieve that, the operating system implements layers of software and hardware, each of which layers has specific roles and capabilities that differ from those of other layers.

The device drivers layer, which is a part of the OS, deals with device-specific code, while the application layer 'sees' all the devices via a common interface.

The principle of using a unified interface and hiding device-specific details from applications or users is called **abstraction**, **standardization**, or **transparency**.

Application I/O Interface



A kernel I/O structure and layers. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

Application I/O Interface

I/O devices, besides being labeled as either 'input' or 'output', can subsequently be divided into a few additional categories (which we review in the next few slides) based on the device's performance and capabilities.

Many operating systems have a function or system call that a user application can use in order to interact with a device driver. On Linux, for example, this is the `ioctl()` ("I/O control") system call. It takes 3 arguments: (1) the device identifier number, (2) the command number: each command tells the driver to perform a certain action, and (3) a pointer to (address in) memory with the data structure that is needed for the performance of the action.

The following diagram lists various device traits (by which devices differ) and brings several examples.

Application I/O Interface

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O devices. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

Application I/O Interface

Any questions?

Application I/O Interface

We can further divide devices into the following categories:

1. **Block** vs. **Character** devices.
    - This is about the size of data that the OS can access within the device at once.
    - **Block** devices can be accessed one block at a time. A **block** is a fixed size of data, usually standing at a multiple of 512 bytes. Block sizes of 4k (= 4096) and 8k (= 8192) bytes are common.
    - Examples of block devices include hard disk drives, solid-state drives, and USB flash drives.
    - **Character** devices can be accessed one byte at a time (remember that 1 byte can store exactly one character or a small integer between 0-255.)
    - Examples of character devices include the mouse, the keyboard, and the monitor.
    - As a general rule, every peripheral I/O device is a character device.

Application I/O Interface

2. **Network** devices.
    - Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
    - One common and popular interface is the **socket interface**, which acts like a cable or pipeline connecting two entities over a network.
    - Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
    - Modern OSs have several network interfaces for various communication types. For instance, Windows provides one interface to the network interface card and a second interface to the network protocols.
3. **Clock** and **Timer** devices.
    - Several time-related services are needed in an OS: knowing the current time, checking how much time elapsed after a previous point in time, and counting the difference (in seconds or milliseconds) between two given timestamps.

Application I/O Interface

3.
   - A device that can be used to trigger operations and to measure elapsed time is called the **programmable interrupt timer** or **PIT**.
   - It could, for example, be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.
   - PIT has several uses in an OS. For instance, the CPU scheduler uses a PIT to trigger interrupts when the time that was given for a program to run on the CPU ends (in order to select another program to run on the CPU.)
   - An OS could implement the system clock by using services from the PIT. However, this could result in time drift. As such, a better solution would be to use high-frequency hardware counters, which provide much higher resolution and accuracy, but do not support interrupts.

Application I/O Interface

4. **Blocking** vs. **Nonblocking devices**.
   - This is about whether a program should wait for I/O to be ready, or let the program continue executing other instructions, without waiting for the I/O data transfer to finish.
   - With **blocking I/O**, a program is moved to the "wait" queue when an I/O request is made, and moved back to the ready queue (to continue running) when the request completes, allowing other processes to run in the meantime while the current program waits.
   - With **nonblocking I/O**, the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the program to check for available data without getting hung completely if it is not there.
   - A nonblocking I/O call returns quickly, with a return value that indicates how many bytes were transferred.

Application I/O Interface

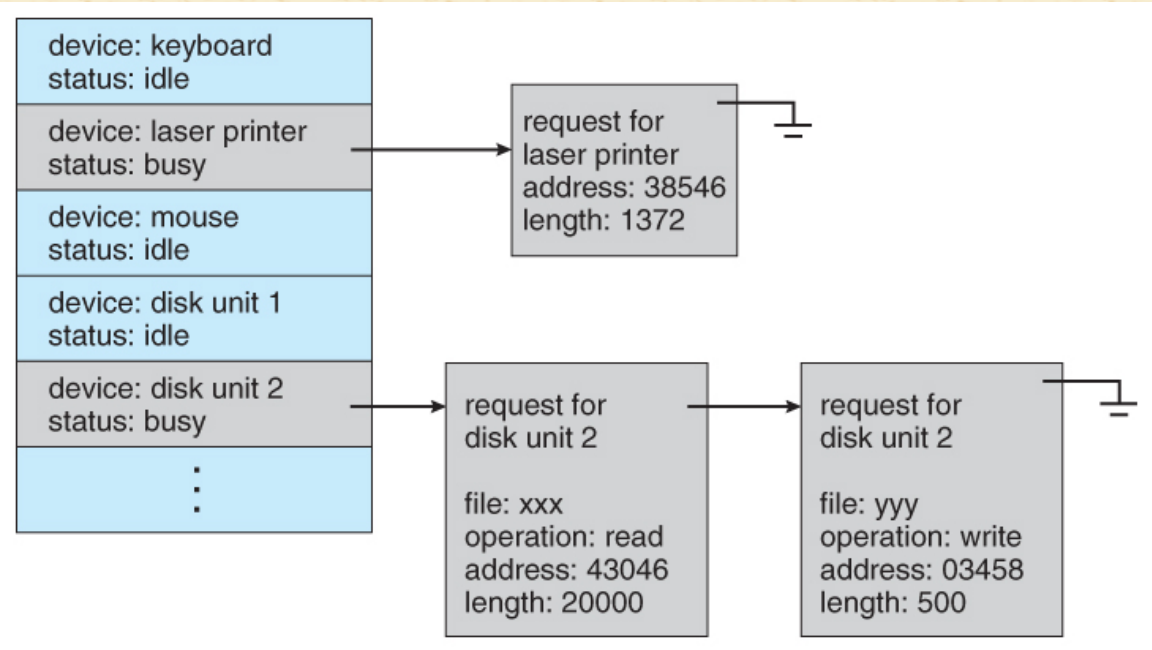
5. **Vectored** I/O.
   - So far, we touched on I/O operations that are done 1 at a time. Each I/O-related system call will carry the data needed for a single I/O operation.
   - Recall that every system call will incur some overhead, since we need to switch activity from the user mode to the kernel mode, and since we might need to switch several programs on the CPU.
   - With **vectored I/O**, several I/O requests can be issued at once.
   - A good analogy is the route of some school bus in the morning. Having the school bus pick up the kids from their homes, and, only after all the kids are on the bus, driving to the school is going to be considerably faster than picking up one kid, driving to the school and letting the kid off at the school, picking up the next kid, and then driving to the school again, etc.
   - The data of all the I/O requests will be provided within some form of array (= vector), and will be handled by the kernel sequentially.

Services of the Kernel I/O Subsystem

In Topic 2, we discussed 10 different service types that an OS provides. One of these was handling I/O requests. Some particular I/O services that an OS provides are:

1. **I/O Scheduling**: It is critical, in terms of efficiency, for the OS to decide in which order I/O requests will be handled.
    - Various CPU scheduling algorithms could be used to determine the most efficient order.
    - The choice of an algorithm depends on what we are trying to achieve: increased throughput, faster multitasking, shorter response time, etc.
    - Every algorithm achieves one of these goals but might undermine another.
    - A **device-status table** stores all the I/O operations that will occur on an I/O device.
    - We will cover more on CPU Scheduling in Topic 6.

Services of the Kernel I/O Subsystem



Device-status table. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

Services of the Kernel I/O Subsystem

2. **I/O Buffering**: this is about storing temporary data that is to be transferred in an I/O operation.
    - A **buffer**, in general, is a memory section (array) that stores data that will be transferred between 2 devices or between a device and an application.
    - **Buffering** is the usage of a data buffer to store data that hasn't been yet transferred.
    - The OS performs buffering for 3 reasons:
        1. The device that receives the data might be slower than the one that provides it. Hence, to prevent the data from being lost, the data is stored in a buffer until the recipient device is ready to accept and process it.
        2. Some devices can't receive data that is too large, meaning that the device can receive data only in smaller chunks. To let the recipient device process data, we store the arriving data in a buffer.
        3. To prevent the data from being mistakenly erased. When we store data in a separate buffer, instead of storing it in an array that a user program created, we ensure that the correct data that was supposed to be transferred is indeed transferred, regardless of whether the program mistakenly overwrote the array's contents before the data was processed.

Services of the Kernel I/O Subsystem

3. **Caching**: Remember **cache**? To reduce the time it takes for a CPU to access data in memory, the OS stores some frequently used data within the cache hardware, which lies closer to the CPU than memory does, and therefore is accessed quicker than memory.
4. **Spooling and Device Reservation**: A **spool** (Simultaneous Peripheral Operations On-Line) buffer stores data for peripheral devices like printers that cannot support interleaved data streams. When two programs want to print files simultaneously, the spool stores the file of one of the programs until the 1st finishes the printing.
5. **Error Handling**: I/O requests can fail for many reasons, either transient like buffer overflow, or permanent like disk crash. In any case, the OS is responsible for detecting when an I/O error happens and properly addressing it.
6. **I/O Protection**: The OS must ensure that no data is being accidentally overwritten by another program. This also includes the prevention of accidental I/O operations. In addition, the OS must prevent access of user programs to sections in memory that are dedicated to memory-mapped I/O and other OS-only memory section.

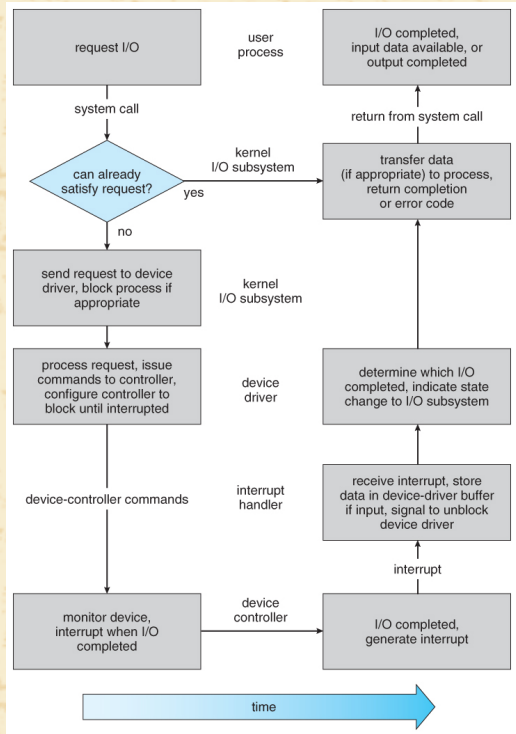
Transforming I/O Requests to Hardware Operations

We now explain how the operating system translates a service request from a user program to actions performed by hardware.

- A user application can specify what data to access by mentioning the name of the file that contains the data.
- Windows apps will use a string and a colon, such as "`C:/`", to indicate the name of the device.
- In Linux and Mac, devices could be accessed using their absolute paths, such as "`/dev/hda`". The leftmost "`/`" forward slash stands for the root folder.
- Linux and Mac use special device files, usually located in the "`/dev/`" directory, to represent and access physical devices directly.
- The operating system could also use lookup tables, which will facilitate the search for connected devices.

The following flow chart shows the life cycle of an I/O request that involves reading data from a device.

Transforming I/O Requests to Hardware Operations



The life cycle of an I/O request. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

Transforming I/O Requests to Hardware Operations

Any questions?

I/O and Interrupts: Sources

Bell, John T. "I/O Systems." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html