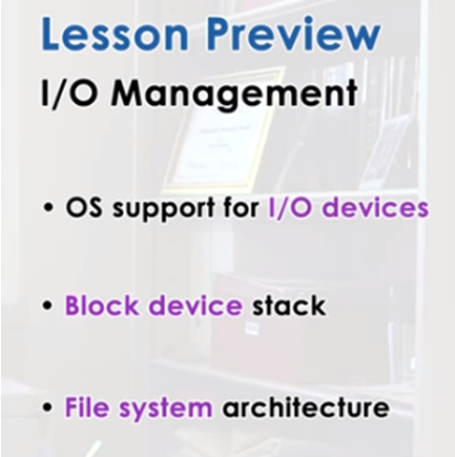
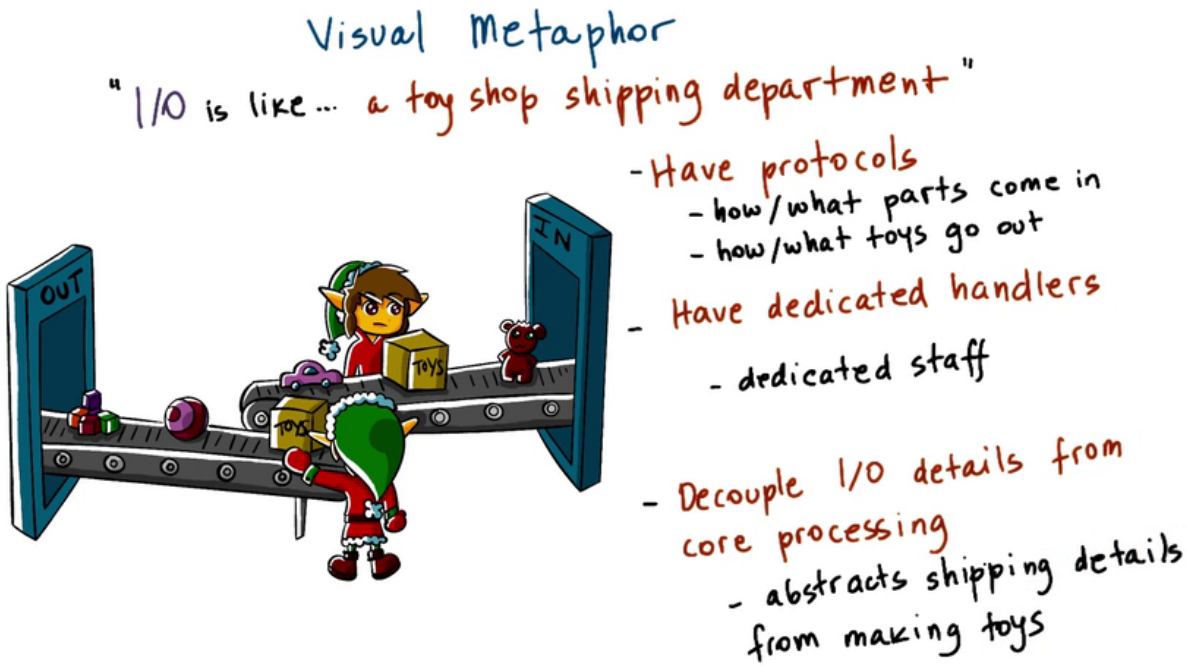
P3L5 - I/O Management

01 - Lesson Preview

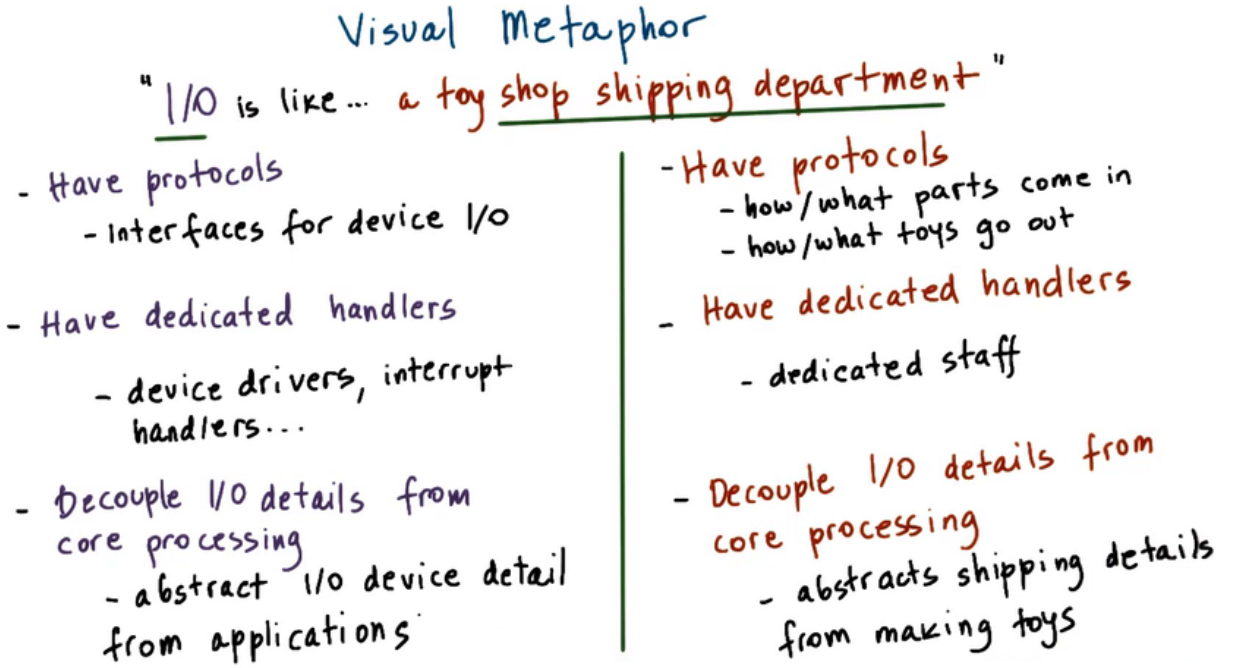


We said that a main role of operating systems is to manage the underlying hardware, but so far we've talked primarily about CPUs and memory. In this lesson we will focus on the mechanisms that operating systems use to represent and manage I/O devices. In particular we will look at the operating system stack for block devices using storage as an example. So in this context, we will also discuss file systems since files are the key operating system abstraction that's used by processes to interact with storage devices. We will describe the linux file system architecture as a concrete example of this.

02 - Visual Metaphor



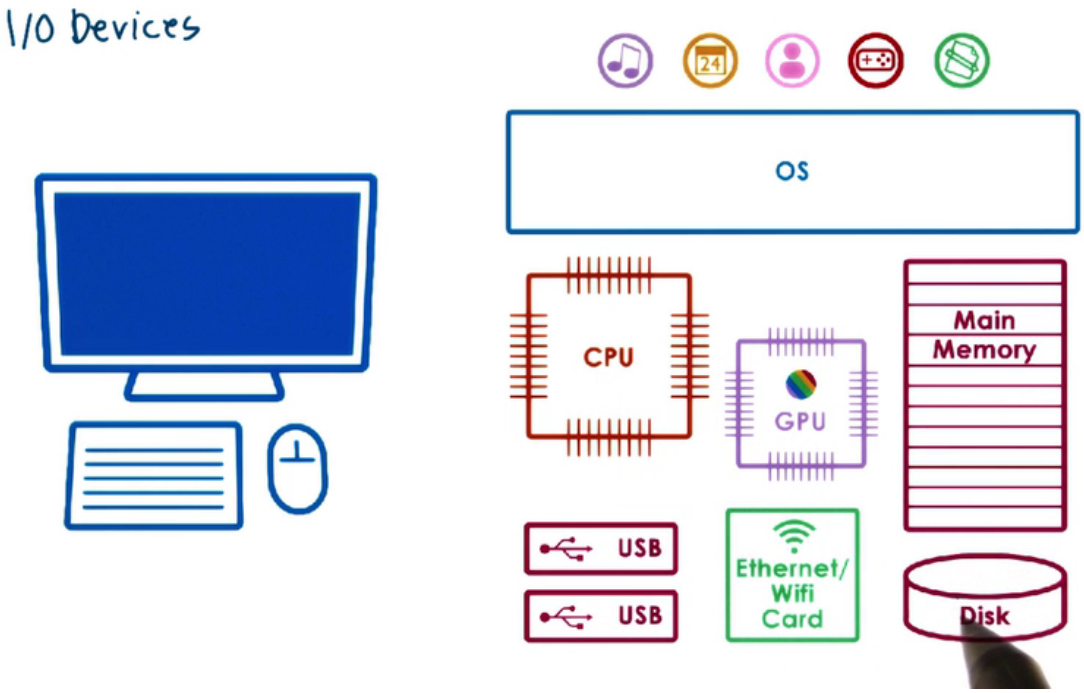
I/O management is all about managing the inputs and the outputs of a system. To illustrate some of the tasks involved in I/O management we will draw a parallel between I/O in computer systems and a shipping department in a toy shop. Let's see what are the similarities we can find between how operating system manages I/O and how the shipping department is managed in a toy shop. For instance, in both the toy shop and in the operating system, there'll be protocols in terms of how the I/O needs to be managed. In both cases, there will be dedicated handlers or dedicated staff that will oversee these protocols and in both environments, the details about the I/O operations are decoupled from the core processing actions. In a toy shop, the I/O protocols determine how and what parts exactly come from a storage facility into the toy shop and which toys and toy orders are being shipped and which order and how. To make sure that these protocols are enforced, in the toy shop will have dedicated staff that handle all aspects related to shipping and in the toy shop the process of making toys is completely separate from any details regarding the shipping process, what are the carriers being used, what are their protocols, are there any sleighs being used, it doesn't matter.



This is analogous in the context of the I/O management in operating systems in multiple ways. For instance, operating systems incorporate interfaces for different types of I/O devices and how these interfaces are used determines the protocols that are used for accessing those types of devices. Similarly, operating systems have dedicated handlers, dedicated operating system components that are responsible for the I/O management. There are device drivers, there are interrupt handlers, these are used in order to access and interact with devices. And finally, by specifying these interfaces and using this device driver model, operating systems are able to achieve this last bullet. They are able to abstract the details of the I/O device and hide them

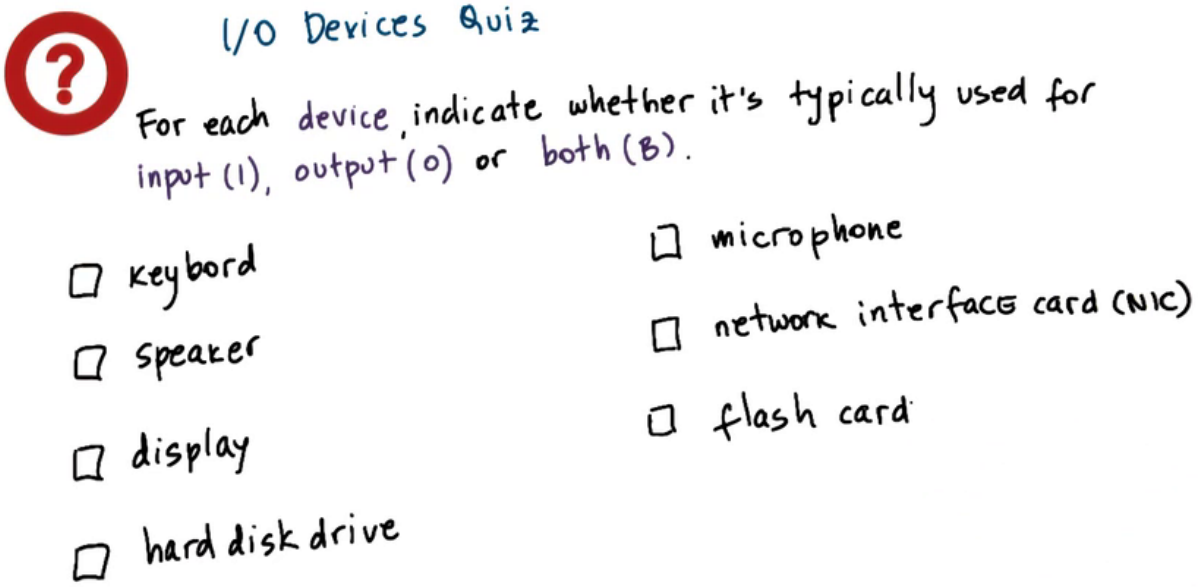
from applications or upper levels of the system software stack, from other system software components.

03 - IO Devices

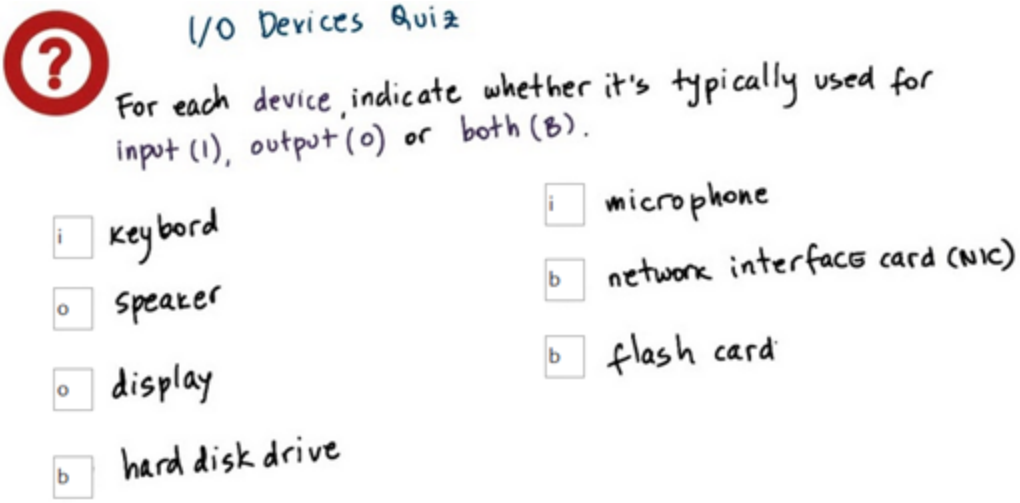


Here is the illustration of a computer system we used in our introductory lesson in operating systems. As you can see, the execution of applications doesn't rely only on the CPU and memory, but relies on many other different types of hardware components. Some of these components are specifically tied to providing inputs or directing outputs and these are referred to as I/O devices. Examples include keyboards and microphones, displays, speakers, mice, also network interface cards, disks. So there are number of different types of input output devices that operating systems integrate into the overall computing system.

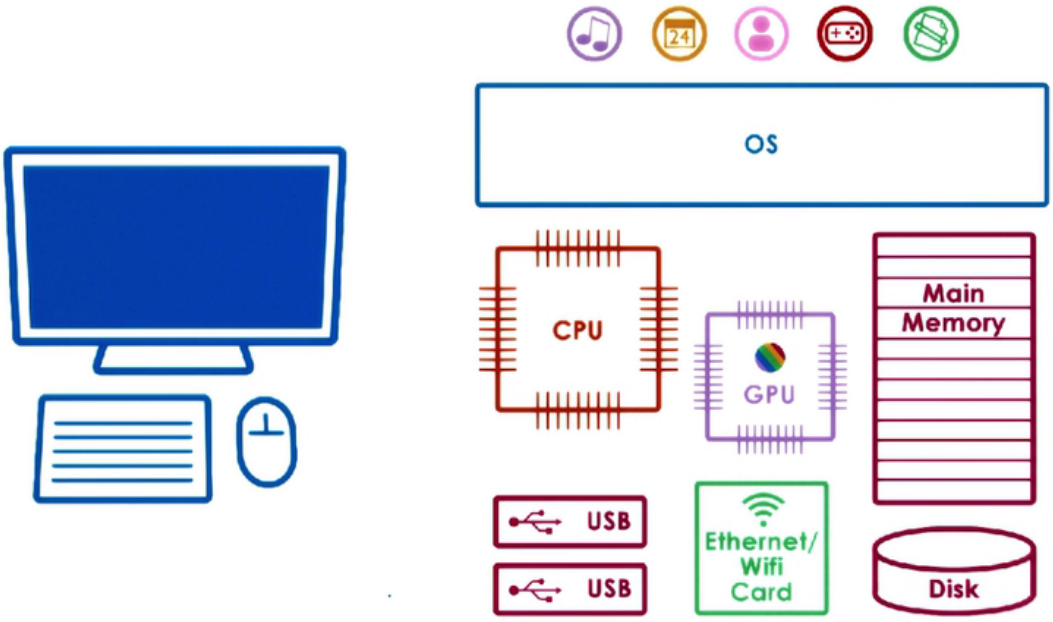
04 - IO Devices Quiz



05 - IO Devices Quiz



06 - IO Device Features



As this figure suggests, the device space is extremely diverse. Device spaces come in all sorts of shapes and sizes with a lot of variability in their hardware architecture, in the type of functionality that they provide, in the interfaces that applications use to interact with them. So, in order to simplify our discussion in this lesson, we'll point out the key features of a device that enable the integration of devices into a system.

Basic I/O Device Features

Control registers
- command
- data transfers
- status

Microcontroller == device's CPU

On device memory

Other logic
- e.g., analog to digital converters

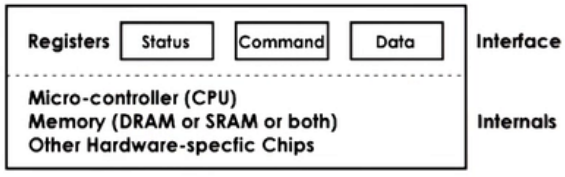


Figure 36.2: A Canonical Device

Any device can be abstracted to have the following set of features. Any device will have a set of control registers that can be accessed by the CPU and that permit the CPU device interactions.

These are typically divided into command registers that the CPU uses to control what exactly the device will be doing. Data registers that are in some way used for the CPU to control the data transfers in and out of the device, and then also some status registers that are used by the CPU to find out exactly what's happening on the device. Internally, the device will incorporate all other device specific logic. This will include a microcontroller and that's like the device CPU, this is what controls all of the operations that actually take place on the device. It may be influenced by the CPU but the microcontroller will make sure what things happen versus not. And then there's some amount of on-device memory, any other types of processing logic, special chips, special hardware that's needed on the device, like for instance to convert analog to digital signals. To actually interact with the network medium with the physical medium whether it's optics or copper or whatever else. So all of those chips will be part of the device.

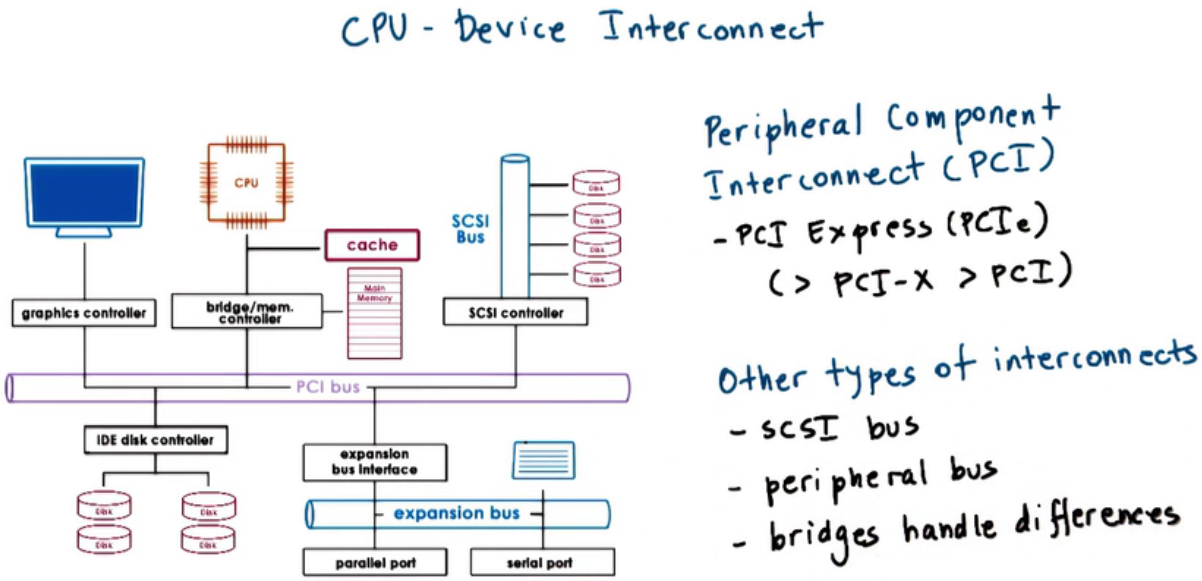
Instructor Notes

Figure 36.2 Reference

Figure 36.2 is referenced from the "I/O Devices" section in Three Easy Pieces.

- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces.
- Arpaci-Dusseau Books, Version 0.80, May 2014.

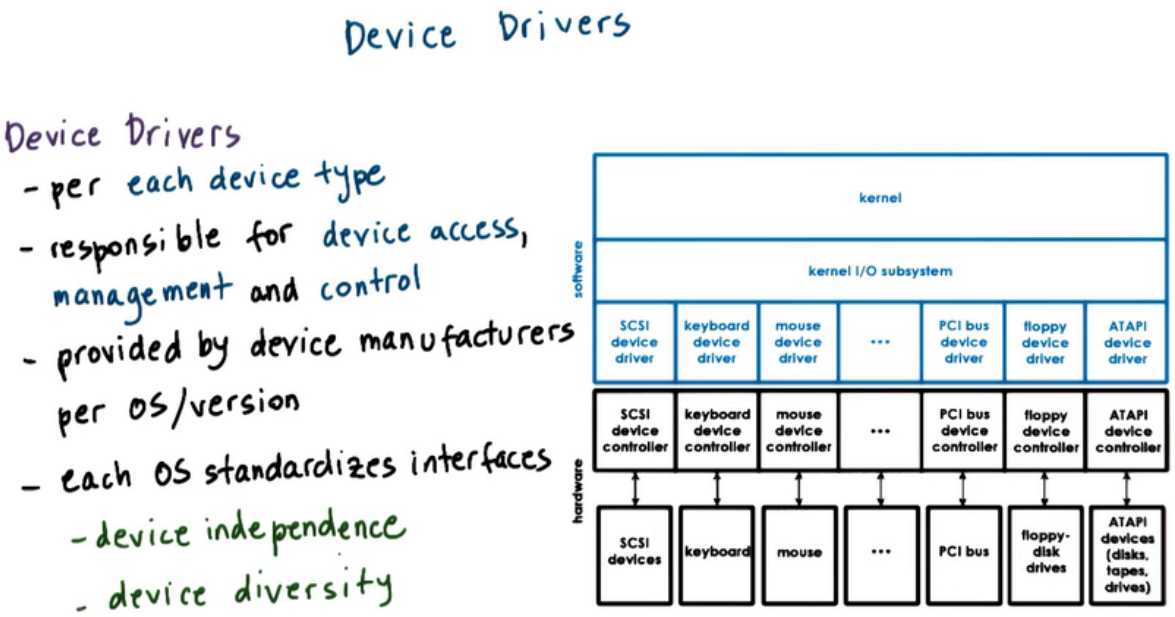
07 - CPU Device Interconnect



Devices interface with the rest of the system via controller that's typically integrated as part of the device packaging that's used to connect a device with the rest of the CPU complex via some CPU device interconnect. Whatever off-chip interconnect is supported by the CPU that different devices can connect to. This figure that's adapted from the Silberschatz book shows a number of different devices that are interconnected to the CPU complex via PCI bus. PCI stands for Peripheral Component Interconnect and it's one of the standards for connecting devices to the

CPU. Today's platforms typically support PCI Express interconnects which are technologically more advanced than the original PCI or PCI-X bus. So PCI Express has more bandwidth, and it's faster, has lower access latency, supports more devices, and in all aspects it is better than PCI-X which was a follow on the original PCI standard. For compatibility reasons though, today's platforms will also include some of these older technologies, typically PCI-X because it's compatible with PCI. PCI-X stands for PCI Extended. Note that the PCI bus is not the only interconnect that's show on this figure alone. There's also a SCSI bus that connects SCSI discs, there's a peripheral bus here shown that connects devices like keyboards and there may be other types of buses. The controllers that are part of the device hardware, they determine what type of interconnect can a device directly attach to. And there are also bridging controllers that can handle any differences between different types of interconnects.

08 - Device Drivers



Instructor Notes

Here is a link to HP Device Drivers (like the drivers mentioned @1:13).

Operating systems support devices via device drivers. Here's a chart that shows where device drivers sit with respect to the rest of the operating system and the actual hardware they manage. Device drivers are device specific software components and so the operating system has to include a device driver for every type of different hardware devices that are incorporated in the system. Device drivers are responsible for all aspects of device access management and control. This includes logic that determines how can requests be passed from the higher levels

of the system software or applications to the actual device. How can the system respond to device level events like errors or response notifications or other status change information or in general any device specific details regarding the device configuration or operation. The manufacturers or the designers of the actual hardware devices are the ones that are responsible for making sure that their device drivers are available for the different operating systems or versions of operating systems where that particular type of device needs to be available. For instance you may have had to download drivers for printers from a manufacturer like HP. Operating systems in turn standardize their interfaces to device drivers. Typically this is done by providing some device driver framework so that device manufacturers can develop the specific device driver within that framework given specific interfaces that the operating system supports. In this way, both device driver developers know exactly what is expected from the ways the rest of the operating system will interact with their device. And also the operating system does not depend on one particular device, one specific device if there are multiple options for devices that provide the same functionality. For instance, for storage you may have different types of hard disks devices from different manufacturers or from networks, you may have different types of network interconnect cards and switching the hardware components will require switching the device driver and the rest of the operating system, the rest of the applications will not be affected. So there are standardized interfaces both in terms of the interaction with those devices as well as in terms of the development and the integration of the device drivers. In this way, we achieve both device independence, the operating system does not have to be specialized or to integrate this particular type of functionality for each specific type of device, and also device diversity so there's an easy way for the operating system to support arbitrarily different types of devices. We just need a device driver.

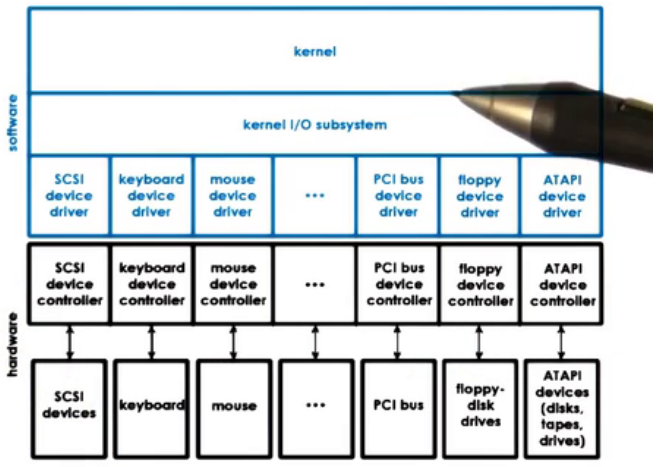
09 - Types of Devices

Types of Devices

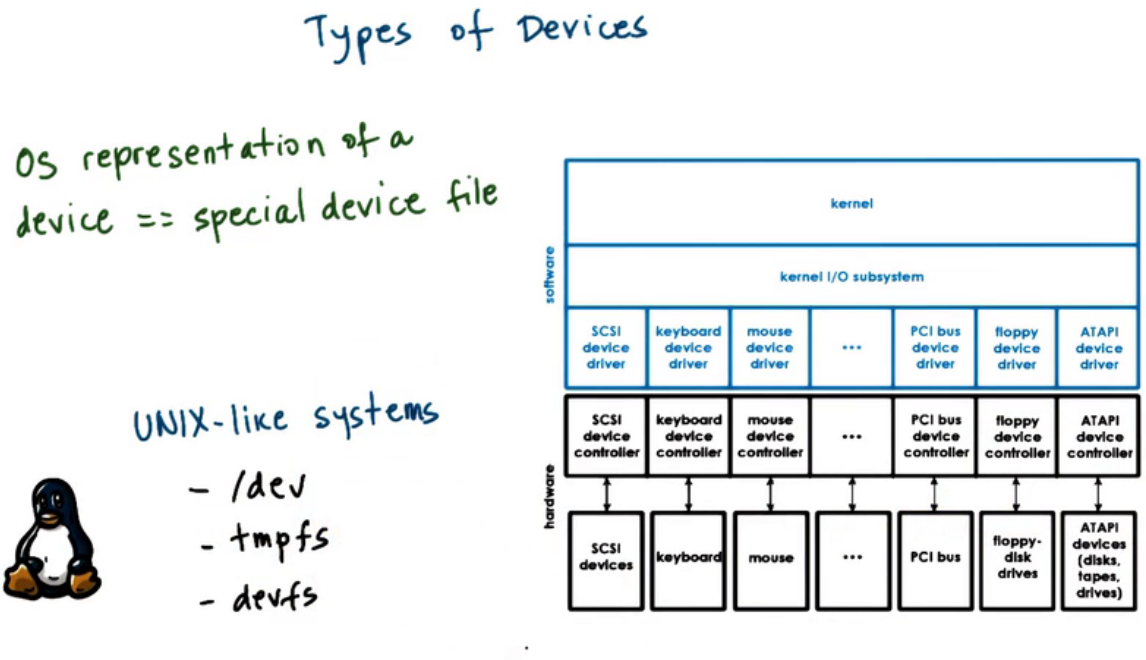
Block: disk
- read/write blocks of data
- direct access to arbitrary block

Character: keyboard
- get/put character

Network devices



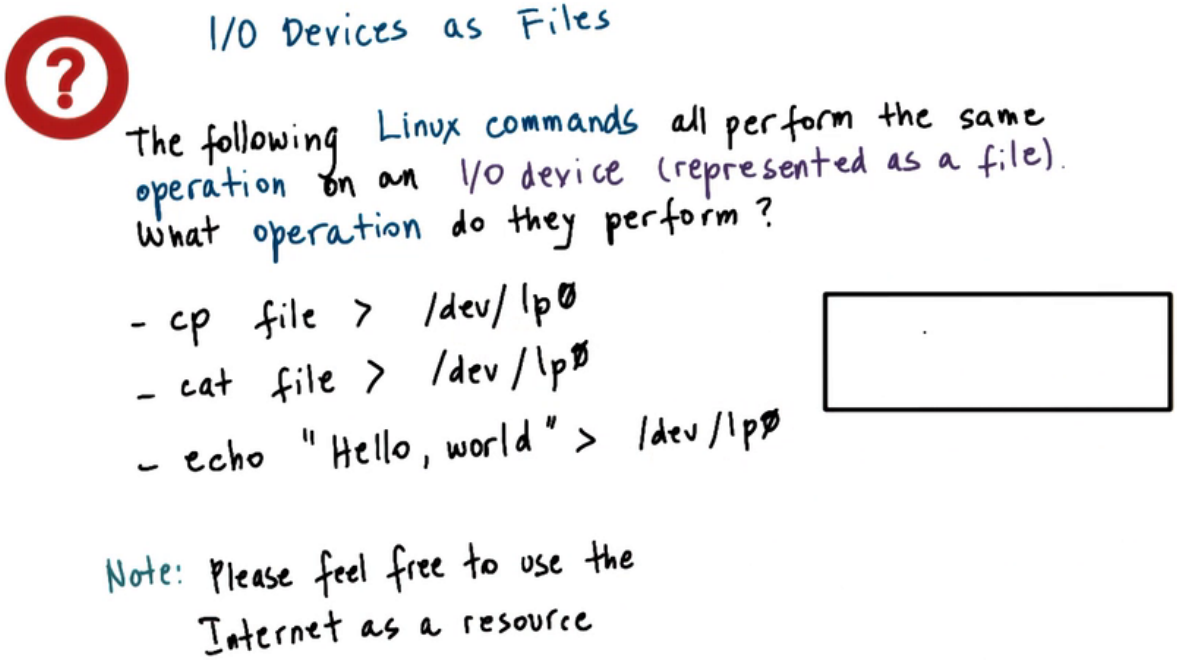
To deal with this great device diversity, devices are roughly grouped in several categories. This includes block devices like disks. These are devices that operate the granularity of entire blocks of data that's delivered in and out of the device, in and out of the CPU complex on the other end. A key property is that individual blocks can be directly accessed. For instance, if you have 10 blocks of data on a disk, you can request to directly access the 9th. That's what makes it a block device. Then there are character devices like keyboards that work with a serial sequence of characters and support something like a get-a-character, put-a-character type of interface. And then there are network devices that are somewhat of a special case, somewhere in between, since they deliver more than a character at a time, but their granularity is not necessarily a fixed block size, it can be more flexible. So this looks more like a stream of data chunks of potentially different sizes. In this manner, the interfaces from the operating systems to the devices are standardized based on the type of device. For instance, for block devices, the drivers should support operations to read/write block of data. For character devices, the drivers should support operations to put/get a character to a device. So in that sense standardized.



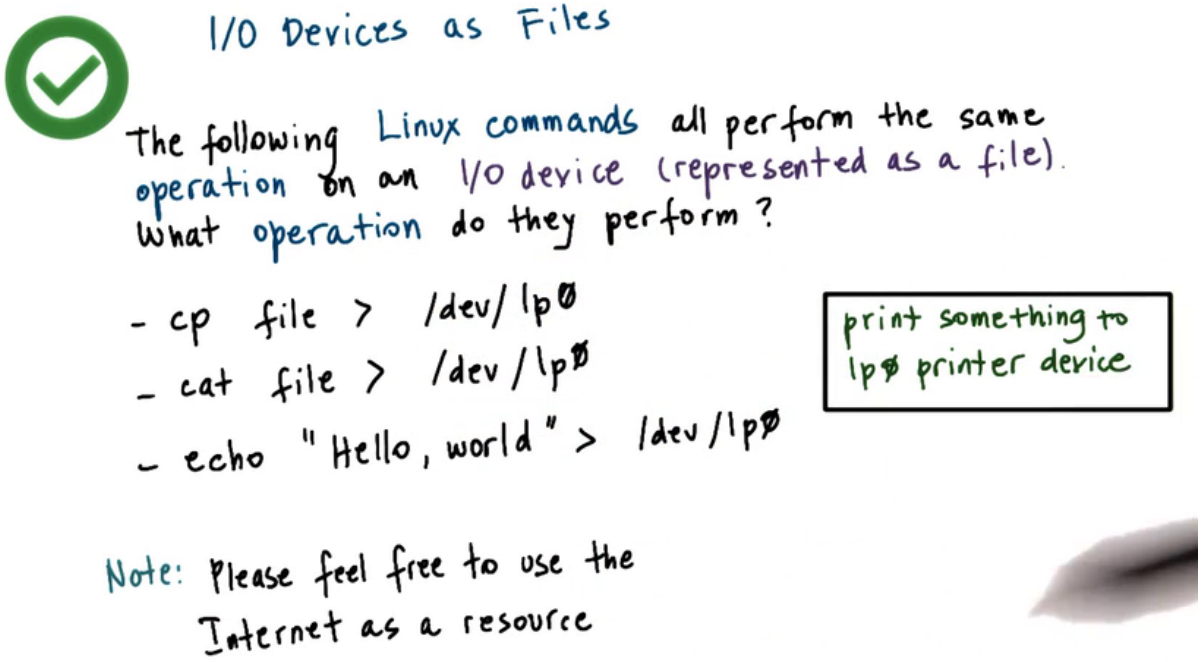
Internally, the operating system maintains some representation for each of the devices that are available on the platform. This is typically done by using a file abstraction to represent the different devices and in that way what really the operating system achieves is that it can use any of the other mechanisms that are already part of the operating system to otherwise manipulate files to now think name referred to access different types of devices. Given ultimately that these files will correspond to devices and not real files then things like read and write operations or actual operations that manipulate this file will be handled in some device specific manner. On Unix like systems, all devices appear as files underneath the /dev directory. As special files

they're also treated by special files system. They're not really stored on the real file system. In Linux versions, these are tmpfs and devfs.

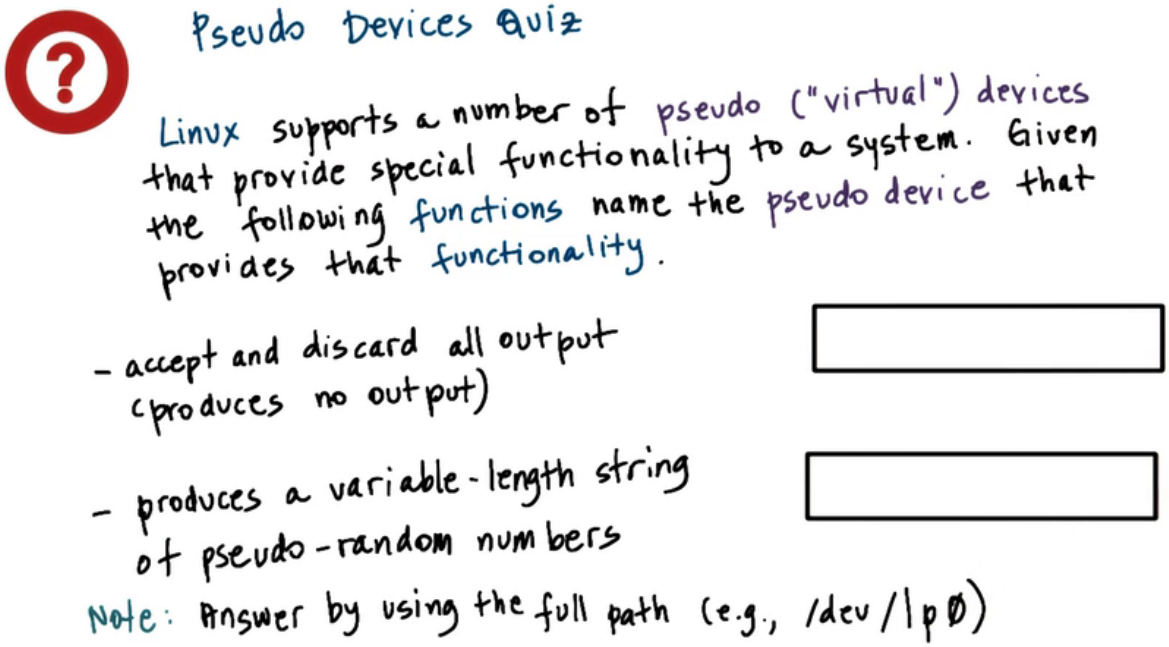
10 - IO Devices as Files Quiz



11 - IO Devices as Files Quiz

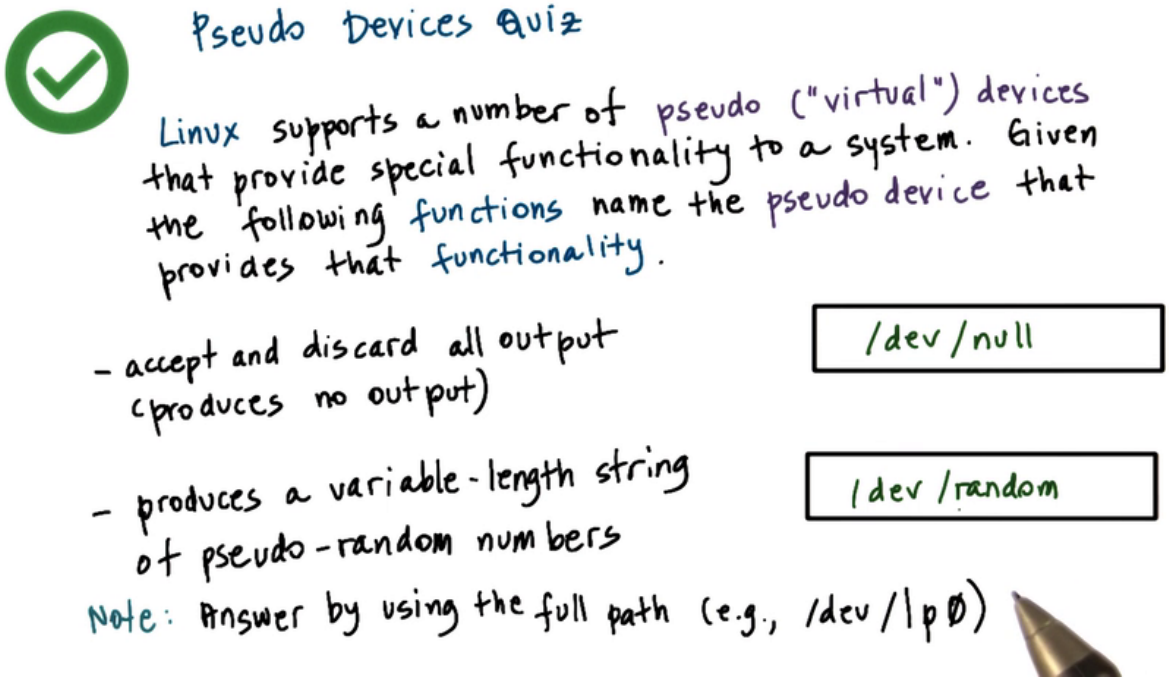


12 - Pseudo Devices Quiz



To go in a little deeper into the special device files, linux also supports what are called pseudo or virtual devices. These devices don't represent actual hardware are not critical in our understanding of I/O management but are useful nonetheless. As a quiz...

13 - Pseudo Devices Quiz



14 - Looking at Dev Quiz

Looking at /Dev quiz

Run the command 'ls -la /dev' in a Linux environment. What are some of the device names you see? Enter at least five device names in a comma-separated list in the text-box below.

Note: You may use the Ubuntu VM provided for this course. See the Instructor Notes for a download link.

Instructor Notes

[Ubuntu VM Setup Instructions]

15 - Looking at Dev Quiz

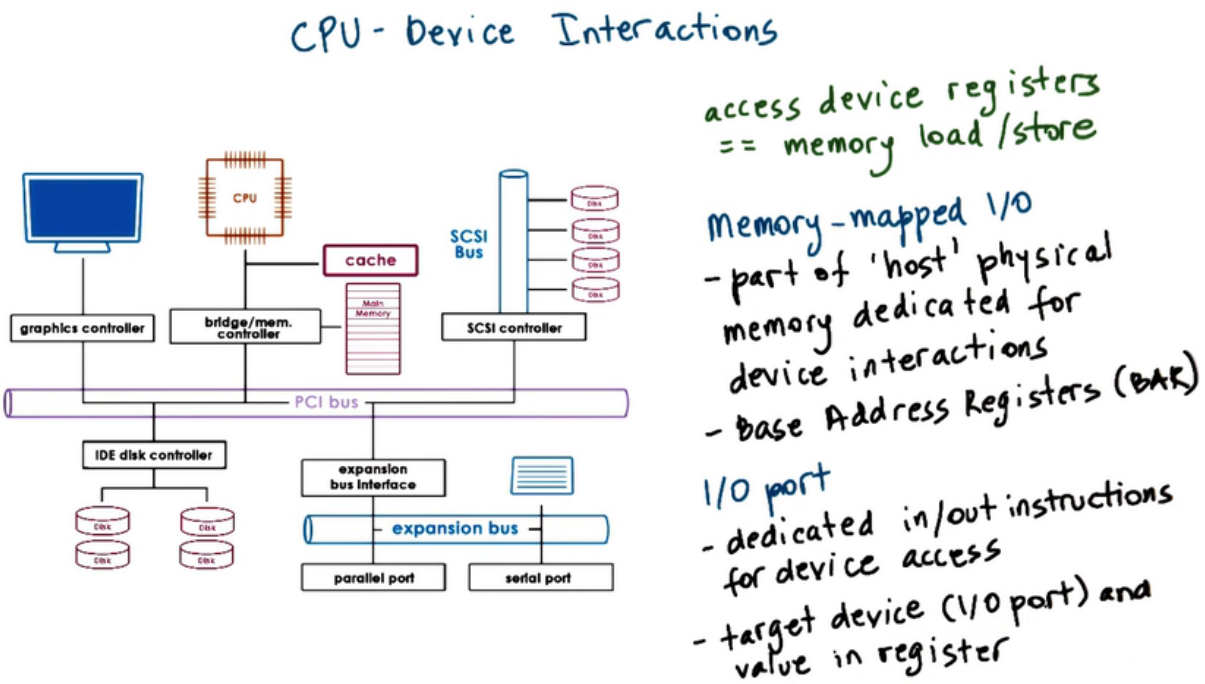
Looking at /Dev quiz

Run the command 'ls -la /dev' in a Linux environment. What are some of the device names you see? Enter at least five device names in a comma-separated list in the text-box below.

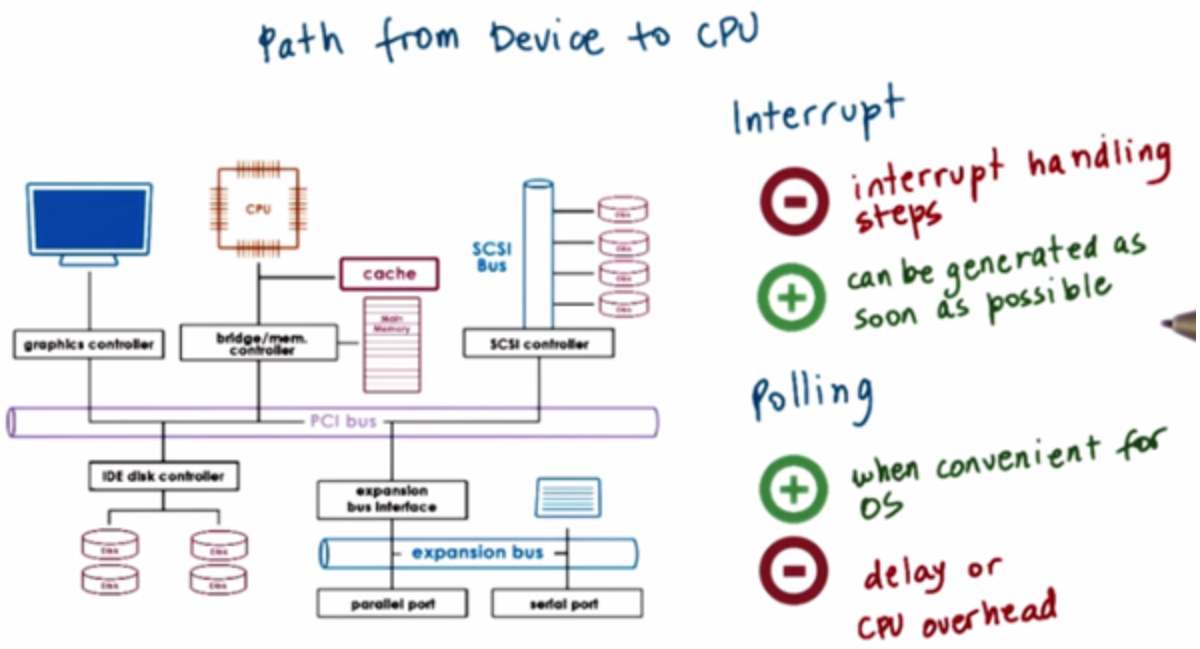
hda, sda, tty, null, zero, ppp, lp, mem, console, autoconf, ...

Note: You may use the Ubuntu VM provided for this course. See the Instructor Notes for a download link.

16 - CPU Device Interactions



The main way in which PCI interconnects devices to the CPUs is by making devices accessible in a manner that's similar to how CPUs access memory. The device registers appear to the CPU as memory locations at a specific physical address. So then when the CPU writes to these locations the integrated memory PCI controller realizes that this access, the access to this physical location, should be routed to the appropriate device. What this means is that a portion of the physical memory on that computing system is dedicated for interactions with the device. We call this memory mapped I/O and the portion of the memory that is reserved for these interactions is controlled by a specific set of registers, the base address registers. So how much memory and starting up which address which will be used by a particular device. This gets configured during the boot process and it's determined how exactly it's done by the PCI configuration protocol. In addition, the CPU can access devices via special instructions. For instance, on x86 platforms there's special in/out instructions that are used for accessing devices. Each of the instructions has to specify the target device, so the I/O port. And some value that's gonna be stored in registers that's the value that's that needs to be written out to device or the value that will be read out of the device. This model is called the I/O port model.



The path from the device to the CPU's complex can take two routes. A device can generate interrupts to the CPU or the option is for the CPU to poll the device by reading its status register to determine if the device has some data for the CPU, does the device have a response to a request that was sent to the CPU, or some other information. There are overheads that are associated with both of these methods. As always, there are tradeoffs between the two. With interrupts, the problem is due to the handling of the interrupt routine, the interrupt handler. The actual steps involved in the handling of the interrupt routine. There's certain operations like setting and resetting the interrupt mask depending on what kinds of interrupts are allowed to interrupt the interrupt handling routine or not. And also some indirect effects due to cache pollution that's related to the execution of this handler. So all of these are overheads, but on the flip side, it is possible to trigger an interrupt as soon as the device has something to do, some kind of notification, some kind of data for the CPU. For polling, the operating system has a possibility to choose when it will poll. At some convenient times when at least some of the cache pollution effects won't be too bad. If that is the case then that's great, some of those overheads will be removed but potentially this will introduce some delays in the way that event is observed, in the way the event is handled. The opposite of just continuously polling will clearly introduce some CPU overheads that may simply not be affordable if there aren't enough CPU's in the system or the system is busy otherwise doing other things. Whether an interrupt or polling mechanism should be selected will really depend on the kind of device that we're dealing with on the objectives, whether we want to maximize things like throughput or latency on the complexity of the interrupt handling routine and the characteristics of the load of the device, so what is input data rate, what is the output data rate that needs to be met and a number of other factors.

17 - Device Access PIO

Programmed I/O (PIO)

no additional hardware support

CPU "programs" the device
- via command registers
- data movement

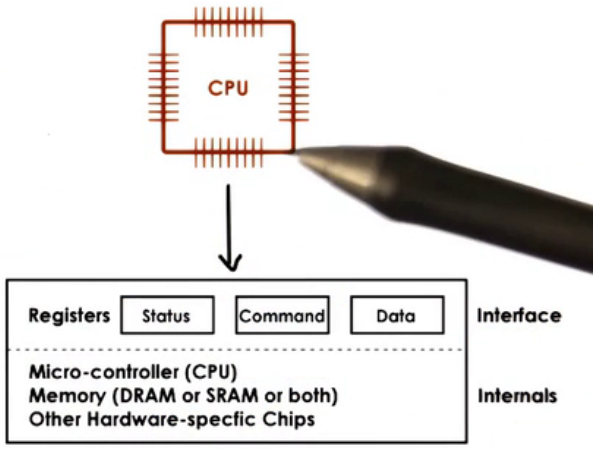


Figure 36.2: A Canonical Device

With just basic support from an interconnect like PCI and the corresponding PCI controllers on the device, a system can access a request an operation from a device using a method called programmed I/O. This method requires no additional hardware support. It involves the CPU issuing instructions by writing into the command registers of the device and also controlling the data movement by accessing the data registers of the device. So either the data will be written to these data registers or read from them.

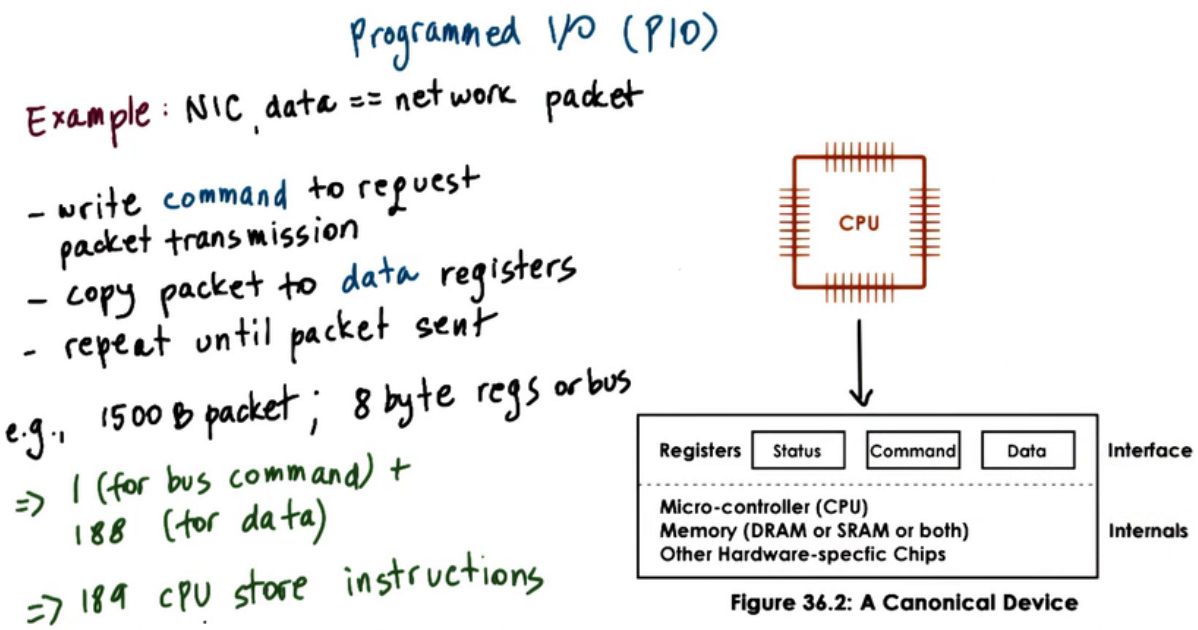


Figure 36.2: A Canonical Device

lp0

For instance, let's look at a network interface card or a NIC as a sample device that's accessed via programmed I/O or PIO. Here in a NIC case, when a process that's running on the CPU wants to transmit some data that has been formatted into a network packet the following steps need to take place. First, the CPU needs to write to a command register on the device and this command needs to instruct the device that it needs to perform a transmission of the data, that the CPU will provide. The CPU also needs to copy this packet into the data registers and then the whole thing will be repeated as many times as it's necessary for the entire packet to be sent out. Given that the size of this register space that's available on the device may be smaller than the network packet. For instance, if we have a 1500 byte packet and the device data registers or the bus that connects the device with PCU are 8 bytes long, then the whole operation of performing programmed I/O will require 1 CPU access to the device registers to write out the command and then another 188 accesses (1500/8). In total there will be 189 CPU accesses to the device specific registers and we said that these look like CPU store instructions. This gives us some idea about the costs associated with programmed I/O.

18 - Device Access DMA

Direct Memory Access (DMA)

relies on DMA controller

CPU "programs" the device
- via command registers
- via DMA controls

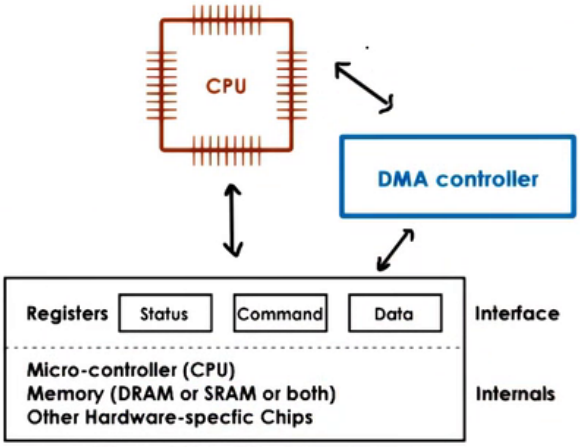


Figure 36.2: A Canonical Device

An alternative to programmed I/O is to use DMA supported devices. DMA stands for direct memory access and it is a technique that relies on a special hardware support in the form a DMA controller. For devices that have DMA support, the way the CPU interacts with them is that it would still write commands in the command registers on the device however the data movement will be controlled by configuring the DMA controller which data to be moved from CPU memory into the device. This requires interactions between the CPU and the DMA controller and in fact the same method can be used to move data in the opposite direction from the device to the CPU. So the device would have a DMA controller that it interacts with to enable that interaction.

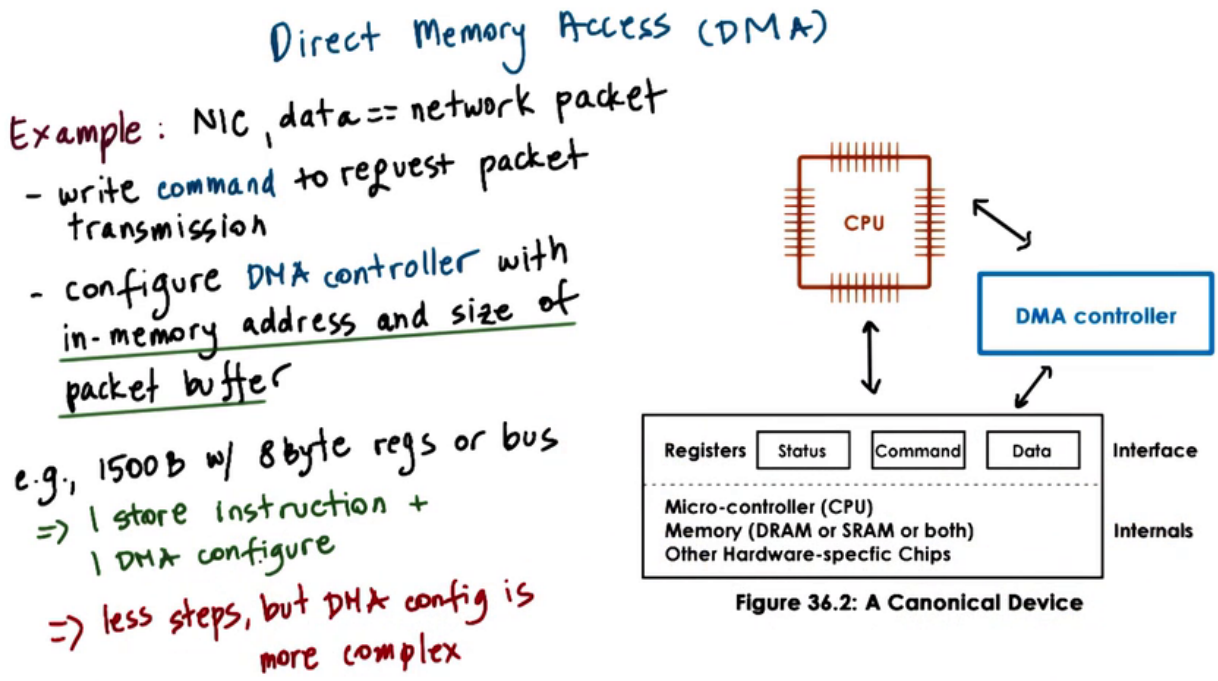


Figure 36.2: A Canonical Device

Let's look again at the NIC example to illustrate how exactly DMA based CPU device interactions are carried out. Again here, the data that needs to be moved from the CPU to the device is already formed as a network packet. And the very first step requires that the CPU writes a command into the device command register to request the packet transmission. This however needs to be accompanied with an operation that configures the DMA controller with the information about the in-memory address and size of the packet buffer that's holding this packet that we want transmitted. That's basically the location of the data we need to move and the total amount of data to be moved. Once this is done, the device can perform the desired operation. From the CPU perspective, performing this transmission requires that we perform one store instruction in order to write out the contents in the command register and one operation to configure the DMA controller. Although this looks much better than the alternative of performing 189 store operations that we saw with programmed I/O, the second step..configuring the DMA controller.. is not a trivial operation. It takes many more cycles than a memory store. Therefore for smaller transfers, programmed I/O is still better than DMA because the DMA step itself is more complex.

Direct Memory Access (DMA)

For DMAs:
- data buffer must be in physical memory until transfer completes
=> pinning regions (non-swappable)

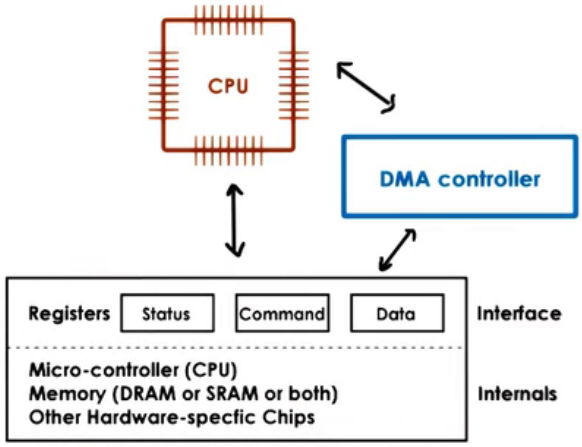


Figure 36.2: A Canonical Device

One important thing is that in order to make DMA work is we have to make sure that the data buffer that we want moved or where we want data to be written must be present in physical memory until the transfer completes. It cannot be swapped out to disk, since this DMA controller can only read and write data to and from physical memory. So the contents have to be there. This means that the memory regions involved in DMA's are pinned. They are not swappable. They have to remain in physical memory.

19 - DMA vs PIO Quiz

DMA vs. PIO Quiz

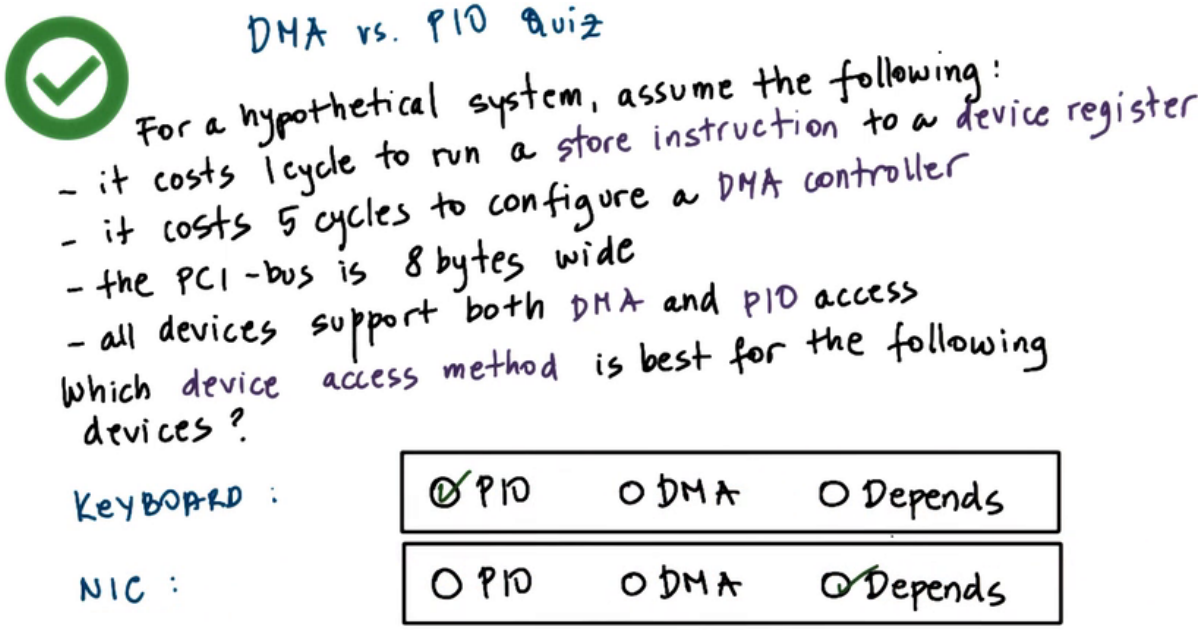
For a hypothetical system, assume the following:
- it costs 1 cycle to run a store instruction to a device register
- it costs 5 cycles to configure a DMA controller
- the PCI-bus is 8 bytes wide
- all devices support both DMA and PIO access

Which device access method is best for the following devices?

KEYBOARD : ○ PIO ○ DMA ○ Depends

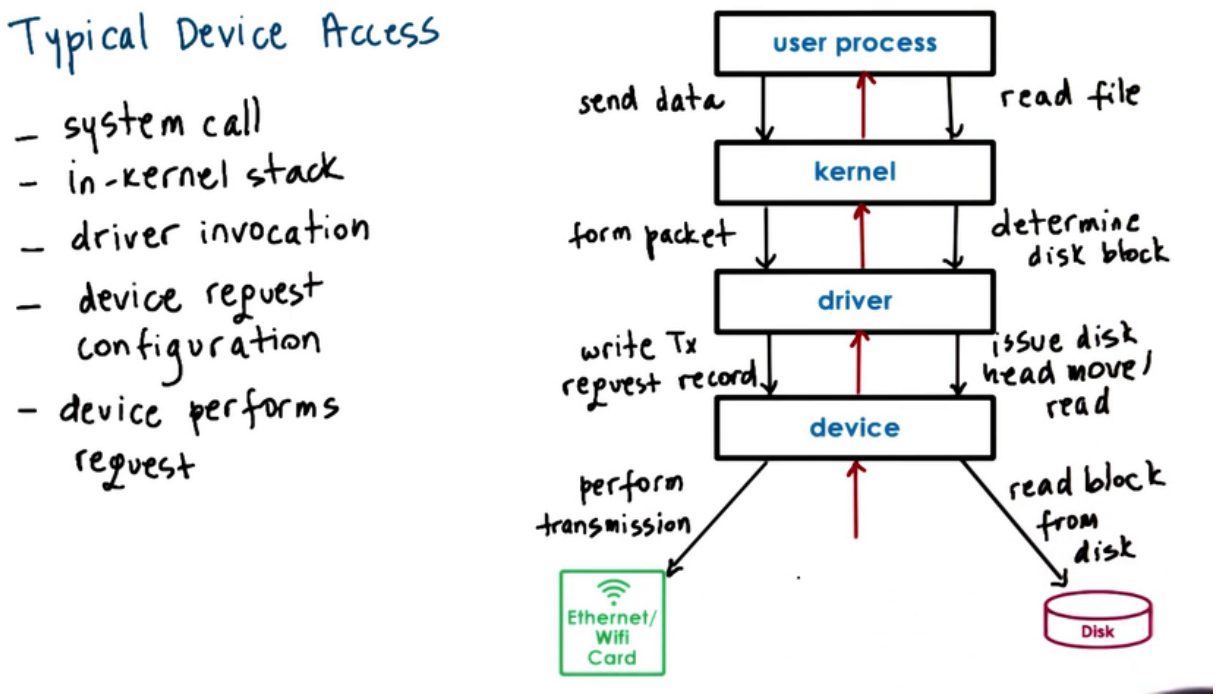
NIC : ○ PIO ○ DMA ○ Depends

20 - DMA vs PIO Quiz



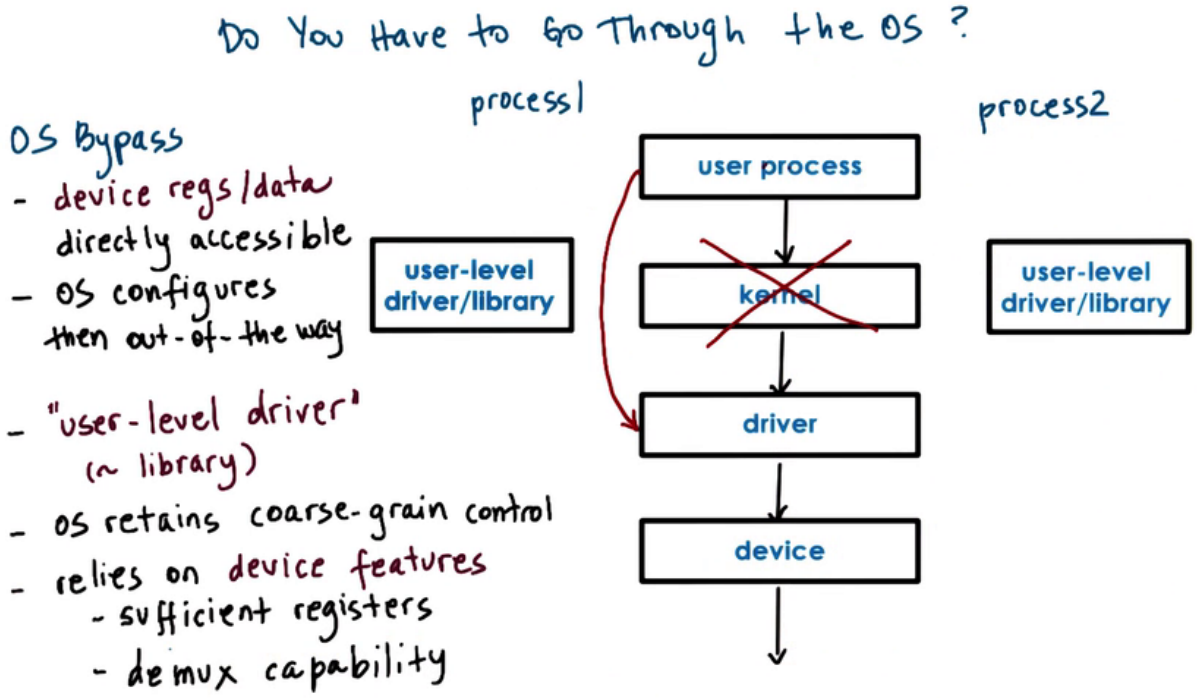
The answer to each question will depend heavily on the size of the data transfers. For a keyboard which likely will not transfer much data for each keystroke, a programmed I/O approach is better since configuring the DMA may be more complex than to perform one or two extra store instructions. For the network card, the answer is the popular it depends answer. If we're sending out small packets that required that we perform less than 5 store instructions to device data registers, given that the difference between the store instruction and the DMA controller in this hypothetical example is 1 to 5, then it's better to perform programmed I/O. If we need to perform larger data transfers, then the DMA option will be the better one since we just need to configure the DMA controller and then issue the request.

21 - Typical Device Access



Typical ways in which user processes interact with a device is as follows. A process needs to perform an operation that requires access from a hardware device for instance to perform a send operation to transmit a network packet or to read a file from disk. The process will perform a system call in which it will specify the appropriate operation. With this, the process is requesting this operation from the operating system kernel. The operating system will run the in kernel stack related to this particular device, like for instance the TCP/IP stack to form the appropriate packet before the data is sent out of the network or the file system that's needed to determine the particular disk block that stores the file data. Then the operating system will invoke the appropriate device driver for the network or for the block device for the disk and then the device driver will actually perform the configuration of the request to the device. On the network side, this means that the device driver will write out a record that configures the device to perform a transmission of the appropriate packet data or on the disk side this means that the device driver will issue certain commands to the disk that the configured the disk head movement or where the data should be read from, etc. The device drivers will issue these commands using the appropriate programmed I/O or DMA operations and this will be done in a device specific manner. So the drivers are the ones that understand the available memory, register on the device. They understand the other pending requests. So the drivers are the ones that will need to make sure that the requests aren't somehow overwritten or undelivered to the device. So all of this configuration and control will be performed at this level, at the driver level. And finally, once the device is configured, it will perform the actual request. In the case of the network card, this device will perform the transmission of the data. In the case of the disk device, the device will perform the block read operation from disk. Finally, any results from the request or in general any events that are originating on the devices will traverse this call chain in a reverse manner.

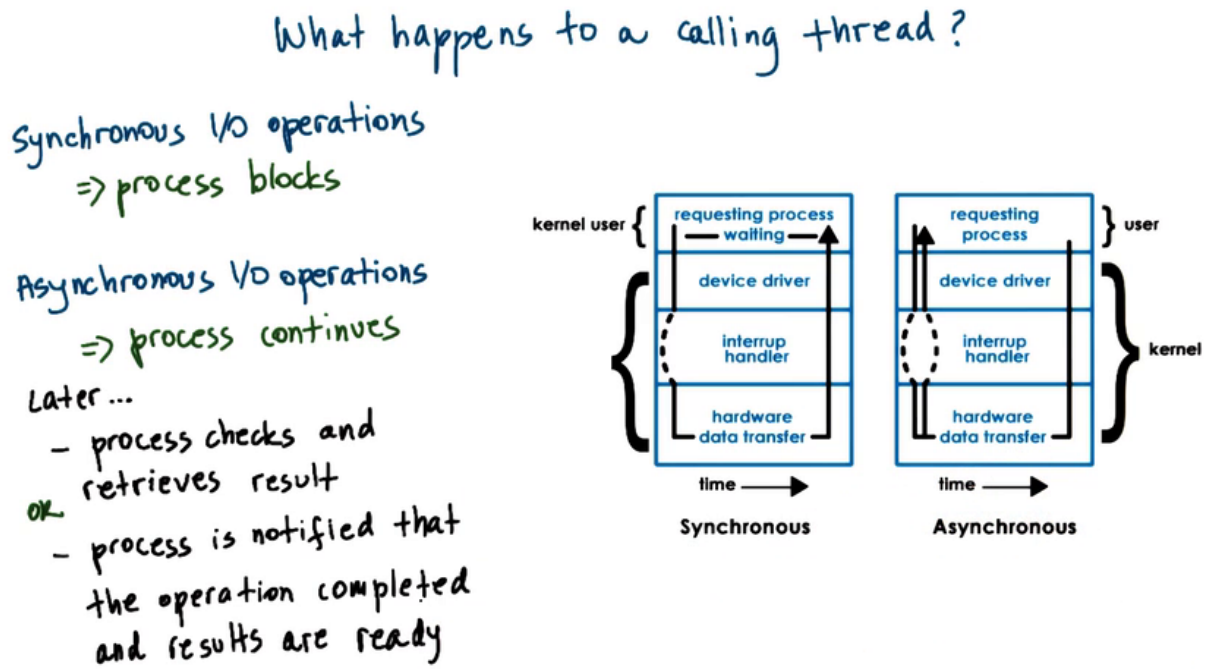
22 - OS Bypass



It is not actually necessary to go through the kernel to get to a device. For certain devices, it is possible to configure them to be directly accessed from user level. This method is called operating system bypass. We're bypassing the operating system and from user level directly accessing the device. That means that the device registers or any memory that is assigned for use for this device is directly accessible to the user process. The operating system is involved in configuring this so making sure that any memory for registers or data corresponding to the device are mapped to the user process but then after that's performed the operating system is out of the way, it is not involved in it and we go from the user process all the way down to the device. Since we don't want the user process to go into the operating system, this driver has to be some user level driver. It's like a library that the user process has to be linked with in order to perform the device specific operations regarding the access to registers or device configurations that are typically performed by the kernel level drivers. Like the kernel drivers, this code, this user level drivers will typically be provided by the actual manufacturers of the devices. When bypassing the operating system the operating system has to make sure that it still has some kind of coarse grain control. Like for instance enabling or disabling a device, adding permissions to add more processes to use the device, etc. This means that the operating system relies on some type of device features like for instance the device has to have sufficient registers so that the operating system can map some registers to the user process so the user process can perform the default device functionality like send receive if it's a network device or read write if it's a disk device but still retain access to whatever registers are used for configuring and controlling the device that are needed for these coarse grain control operations. If the device has too few registers and it's reusing the same registers for both the core data movement or the core functionality as well as these control operations that are needed to be

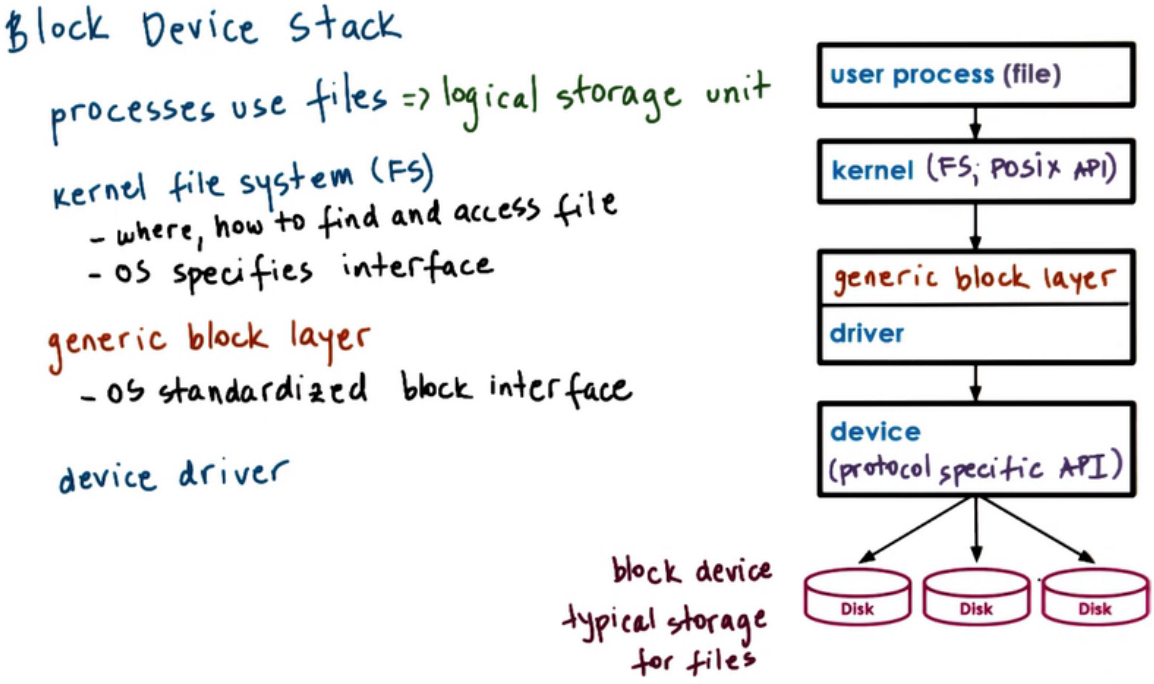
performed by the operating system, we can't do this. We need to be able to share the same device across potentially multiple user processes so assign some subset of the registers to different user level processes to be controlled by different user level drivers and libraries and still make sure that the operating system has some coarse grain control over exactly how the device is used and whether something needs to be changed. Another thing that happens when multiple processes use the device at the same time is that when the device needs to be pass some data to one of the processes, it is now the device that needs to be able to figure out how exactly how to pass data to the address space of process 1 vs process 2. For instance when receiving network packets the device itself has to determine which process is the target of the packet. If you think about what that means with respect to the networking protocols that means the device has to peek inside of the packet in order to see what is the port number that this packet is intended to and then also it has to know which are the socket port numbers that these processes are using in their communications. So what that means is that the device has to perform some protocol functionality in order to be able to demultiplex the different packets that belong to these different processes. In general it needs to have demultiplexing capabilities so that data that's arriving at this device can be correctly passed to the appropriate process. In a regular device stack where the operating system kernel is involved, it is the kernel that performs these operations, the kernel is aware of the resources that are allocated to each process and the mappings that each process has with respect to the physical resources in the system. When the operating system is bypassed, those types of checks have to be performed by the device itself.

23 - Sync vs Async Access



When an I/O request is made, the user process typically requires some type of a response from the device. Even if it's just a status that yes I got it, I'll do this for you. So what happens when the user process or the user thread once the I/O call is made. What will happen to the thread will depend on whether the I/O operations are synchronous or asynchronous. For synchronous operations the process or the calling thread at least will be blocked. The OS kernel will place that thread on the wait queue that's associated with the corresponding device and then the thread will eventually become runnable when the response from this request becomes available. So in the meantime, it will be blocked, it will be waiting, it will not be able to perform anything else. With asynchronous operations, the user process is allowed to continue as soon as it issues the I/O call. At some later time, the user process can be allowed to come in and check are the results ready for this operation and at that point it will retrieve the results, or perhaps at a later point the process itself will be notified by the device or by the operating system that the operation has completed and that any results are ready and are available at the particular location. The benefit of this is that the process doesn't have to go and periodically check whether the results are available. This is somewhat analogous to the polling vs interrupt based interface that we talked about earlier. Remember that we talked about the asynchronous I/O operations when we talked about the Flash paper in the lesson on thread performance consideration. There, the solution was for the kernel to avoid blocking the user process by creating separate threads that will perform I/O operations, in that case synchronous operations that will block. Here we're really talking about asynchronous I/O operations truly being supported within the operating system.

24 - Block Device Stack

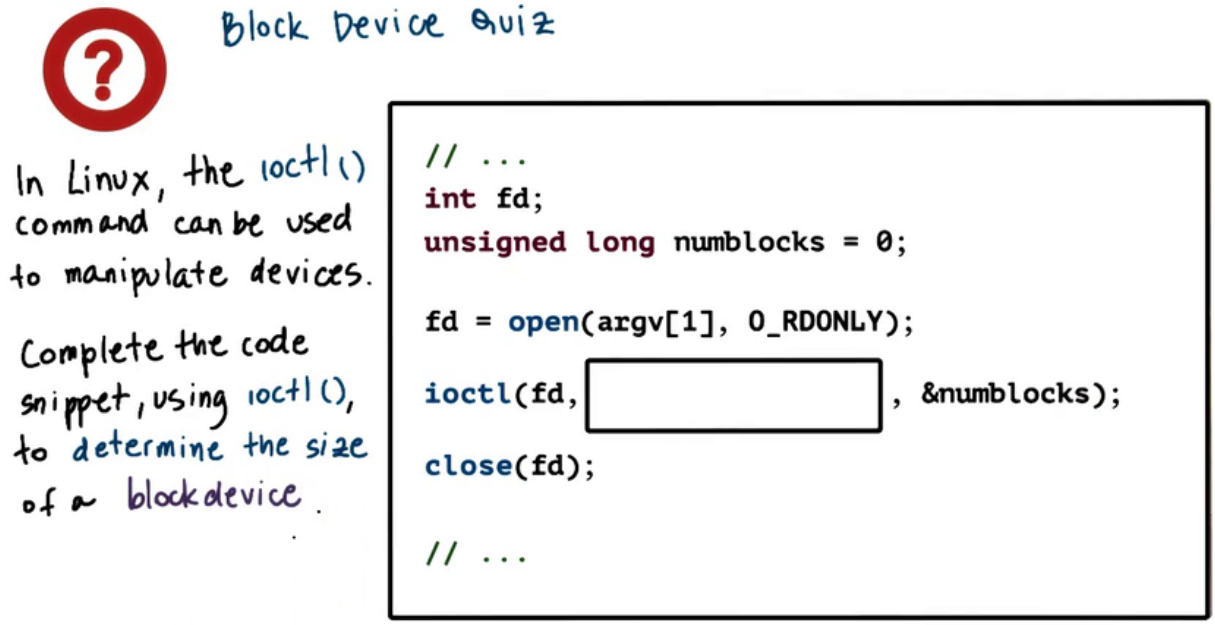


Blank Video >@4:02

Let's look closer at how block devices are used using a similar diagram. Block devices like disks are typically used for storage and the typical storage related abstraction used by applications is a file. The file is a logical storage unit and it is mapped to some underlying physical storage location. What that means is that at the topmost level, applications don't think about the disks, they don't issue operations on disks for seeking blocks and sectors, etc, instead they think about files and they request operations to be performed on files. Below this file based interface used by applications will be the file system. The file system will have the information how to take these reads and writes that are coming from the application and to then determine where exactly is the file, how to access it, what is the particular portion of that file that needs to be accessed, what are any permission checks that need to be performed and to ultimately initiate the actual access. One thing that's useful to have is for operating systems to provide some flexibility in the actual details that a file system has in terms of how it lays out files on disk, how it performs these access operations. In order to do that, operating systems allow for a file system to be modified or completely replaced with a different file system. To make this easy, operating systems specify something about the file system interfaces. This includes both the interfaces that are used by applications to interact with a file system and there the norm is the POSIX API that includes the read and write and open file system calls that we've mentioned so far. Also standardizing the file system interfaces means that there will be certain standard API's in terms of how these file systems interact with the underlying storage devices as well as with operating system components that need to interact with. If the files are indeed stored on block devices,

clearly at the lowest level, the file system will need to interact with these block devices via their device drivers. We can have different types of block devices where the files could be stored, SCSI, IDE disk, hard disk, usb drives, solid state disks, and the actual interaction with them will require certain protocol specific API's. Now in spite of the fact that all of these may be block devices, there still may be certain differences among them. For instance, what are the types of errors they report or how they report the errors. So in order to mask all of that, the block device stack introduces another layer and that is the generic block layer. The intent of this layer is provide a standard for a particular operating system to all types of block devices. The full device features are still available and accessible through the device driver however if used by the file system stack, some of these will be abstracted underneath this generic block device interface. So then what happens when the user process wants to perform an access a read or write operation on a file, it invokes that read write operation per the POSIX API and the kernel level file system will then based on the information that it maintains will determine what is the exact device that needs to be accessed and what is the exact block on that device that supports that particular region of the file. That in turn will result in some generic read block write block operations that are passed to the generic block layer and this layer will know how exactly to interact with the particular driver, so how to pass those operations to the particular driver and how to interpret the error messages or notifications or responses that are generated by the driver. Any lower level differences among the devices like the protocols that they use, etc will be handled by the driver, it will speak the specific protocol that's necessary for the particular device.

25 - Block Device Quiz

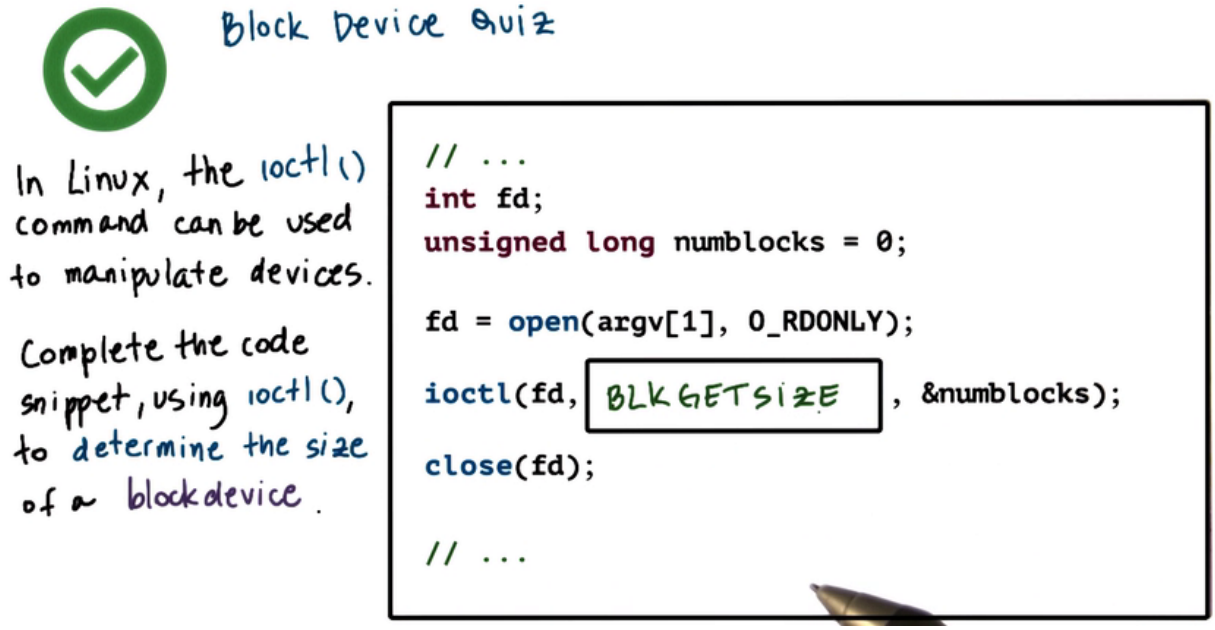


Instructor Notes

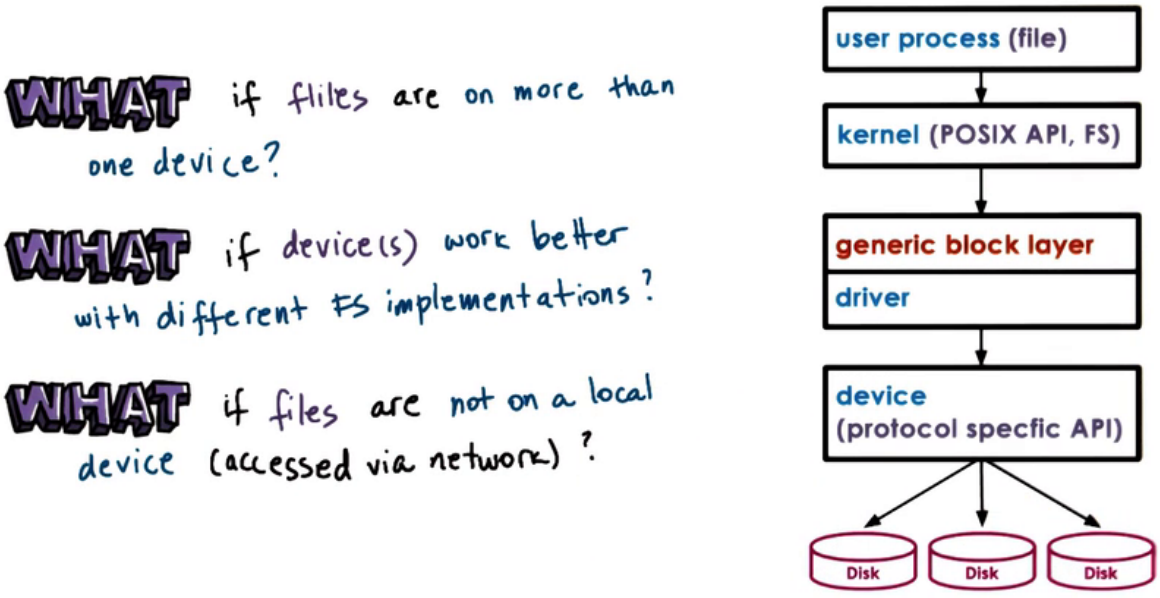
Quiz Help

- List of ioctl calls

26 - Block Device Quiz

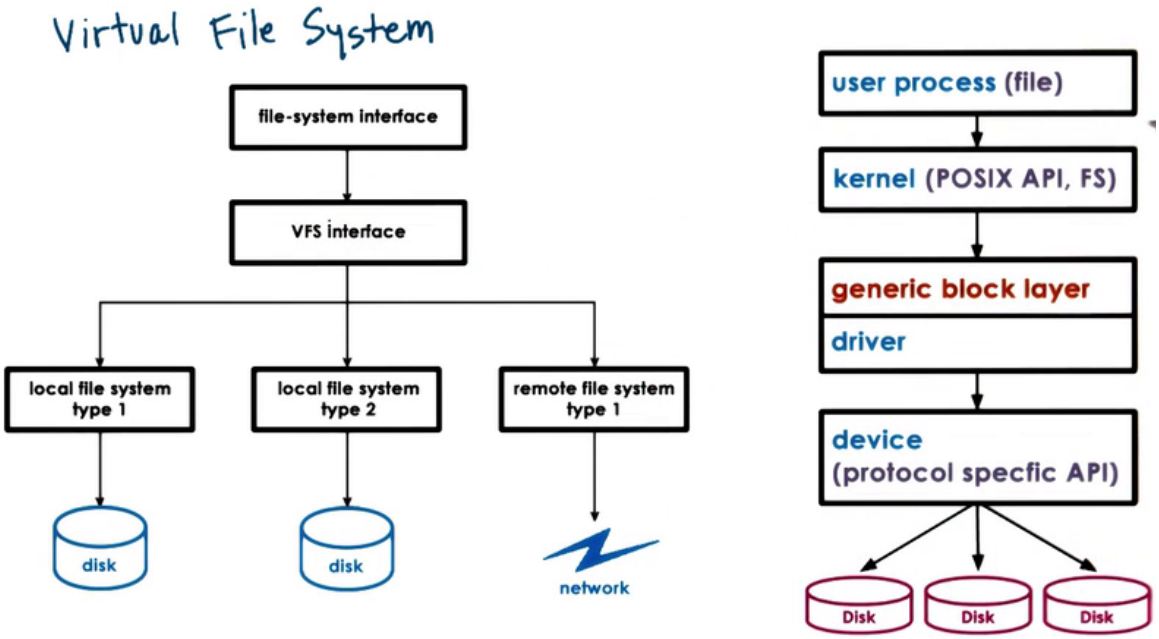


27 - Virtual File System



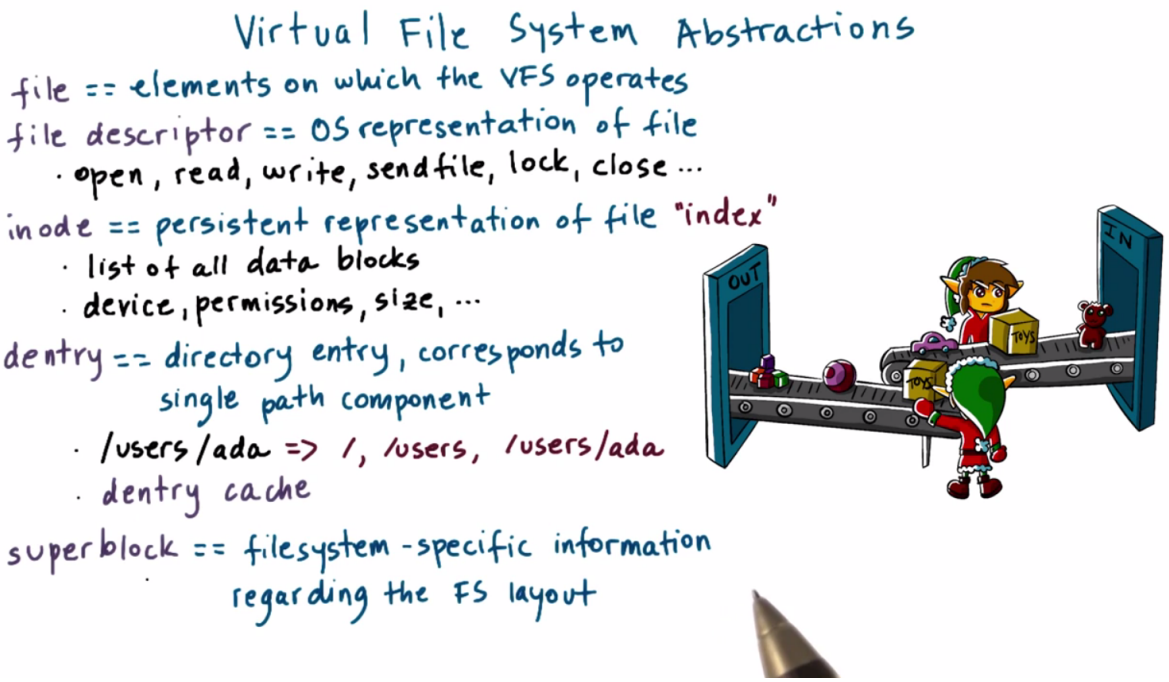
Here is the block device stack figure from the previous lecture. As we said it has well defined interfaces among the applications and the kernel level file system and this is what makes it easy to change the implementation of the file system that we want to use without making any changes to the applications. In the same way, we can also change what is the particular type of a device that a file system uses without really making changes further up the stack. However,

what if we want to make sure that the user applications can seamlessly as a single file system see files that are distributed across multiple devices and if on top of that different types of devices work better with different file system implementations. Or what if the files aren't even local on that particular machine on some of the devices, what if they need to be accessed via the network.



To deal with these differences, operating systems like linux use a virtual file system layer. The virtual file system will hide from applications all details regarding the underlying file system whether there is one or more, whether they use one or more local devices, or a file systems that reference remote servers and use remote storage. Applications continue to interact with this virtual file system using this same type of API for instance the POSIX API and the virtual file system will specify a more detailed set of file system related abstractions that every single one of the underlying file systems must implement so that it can perform the necessary translations.

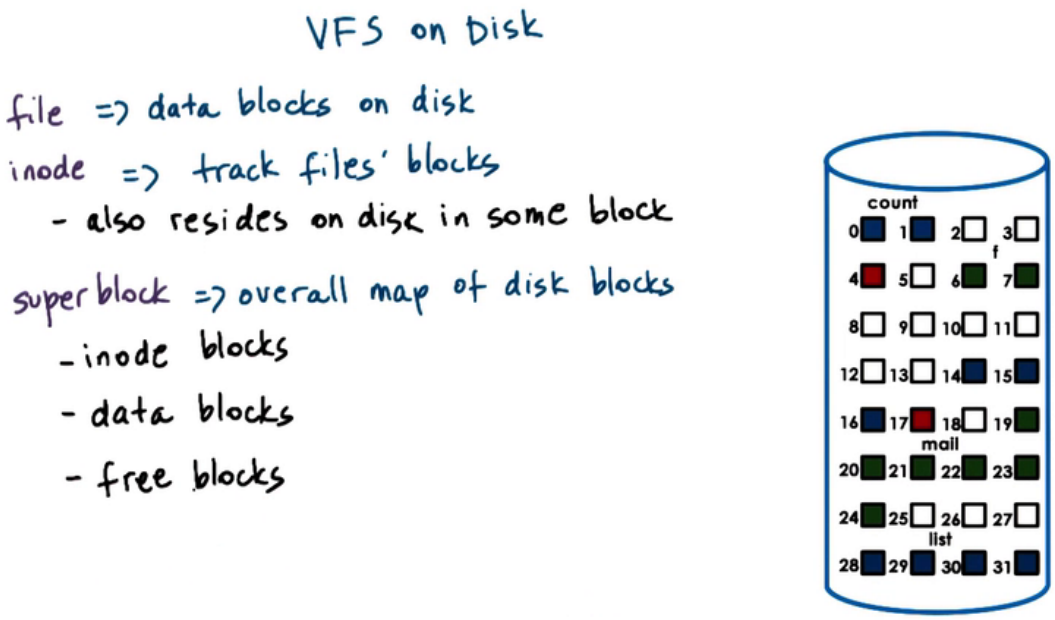
28 - Virtual File System Abstractions



The virtual file system supports several key abstractions. First, there is obviously the file abstraction. These are the elements on which the virtual file system operates. The OS represents files via file descriptors. A file descriptor is created when the file is first opened and there are a number of operations that can be supported on files by using the file descriptor to identify the specific file. These include read, write, lock a file, send a file, also close a file ultimately. For each file, VFS maintains a persistent data structure called an inode. One of the information that is maintained in this inode is the list of all the data blocks that correspond to this file. This is how the inode derives it's name. It's like an index node for that file. The inode also contains other pieces of information for that file like permissions, the size of the file, whether the file is locked, etc. The inode is a standard data structure in unix based systems and again it's important because the file does not need to be stored contiguously on disk. Its blocks may be all over the storage media and therefore it's important to maintain this index. Now we know that files are organized in directories but from the virtual file system's perspective and in general from unix based system's perspective, a directory is really just a file except its contents include information about files and their nodes so that we can find where are the data blocks for these files. So, the virtual file system will interpret the contents of the directory a little bit differently. To help with certain operations on directories, linux maintains a data structure called dentry, which stands for directory entry and each dentry object corresponds to a single path component that's being traversed as we're trying to reach a particular file. For instance, if we're trying to reach a file that's in my directory, the directory named ada we would have to traverse this path /users/ada. In the process, the virtual file system will create a dentry element for every path component for / for /users and then finally for the 3rd directory in this path, /users/ada. Note this first / that corresponds to the root directory in the file system structure. The reason that this is useful is that when we need to find another file that's also stored in this directory ada, we don't have to go through the entire path and try to reread the files that correspond to all of these

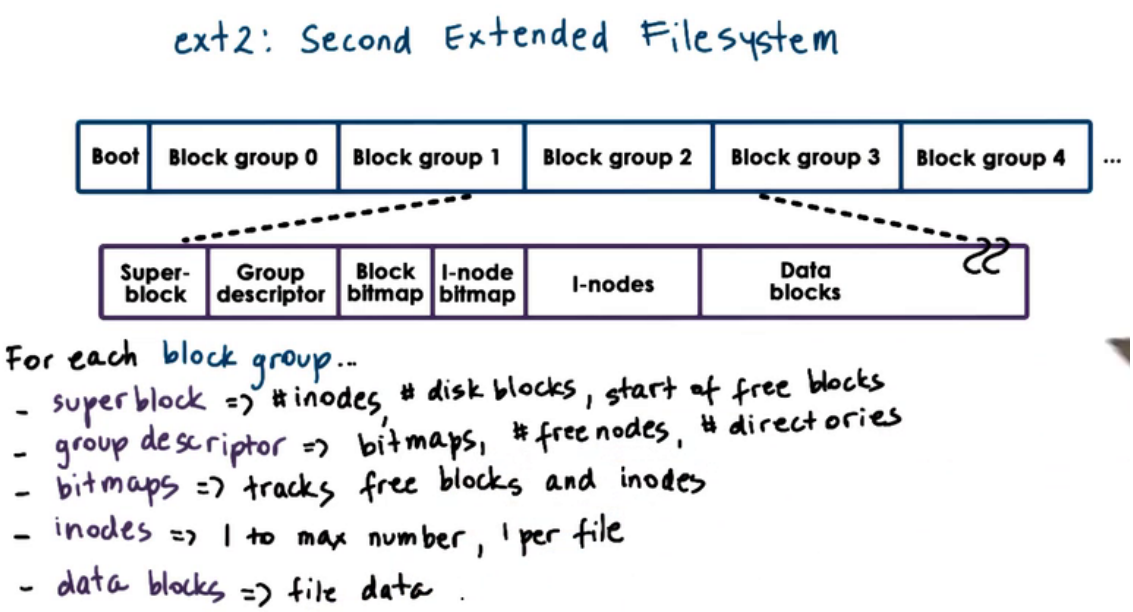
directories in order to get to the directory /ada and then ultimately to find the file the next file that we're searching. The file system will maintain a cache of all the directory entries that have been visited and we call that the dentry cache. Note that this is soft state, there isn't some persistent on-disk representation of the dentry objects. This is only in memory, maintained by the operating system. Finally, there is the superblock abstraction that's required by the virtual file system so that it can provide some information about how a particular file system is laid out on some storage device. This is like a map that the file system maintains so that it can figure out how it has organized on disk the various persistent data elements like the inodes or the data blocks that belong to different files. Each file system also maintains some additional metadata in this superblock structure that helps during its operation. Exactly what information will be stored in a superblock and how it will be stored differs among file systems so that's why we say it's file system specific information.

29 - VFS on Disk



The virtual file system data structures are software entities and they're created and maintained by the operating system file system component. But other than the dentries, the remaining components will actually correspond to blocks that are present on disk. The files of course are written out to disk and they will occupy some blocks. And here we have two files, the green file and the blue file and they occupy multiple blocks that don't have to be contiguous. The inodes we said will track all of the blocks that correspond to a file and they do have to be persistent data structures so they will live somewhere on disk so let's say these two blocks here correspond to the inodes for these two different files for simplicity. To make sense of all of this and to be able to tell what is an inode, what is a data block, what is a free block the superblock maintains an overall map of all of the disks on a particular device. This is used for allocation when we need to find some free blocks to allocate to a new file creation request or a file write request. And it is also used for lookup when we need to find a particular portion of a particular file.

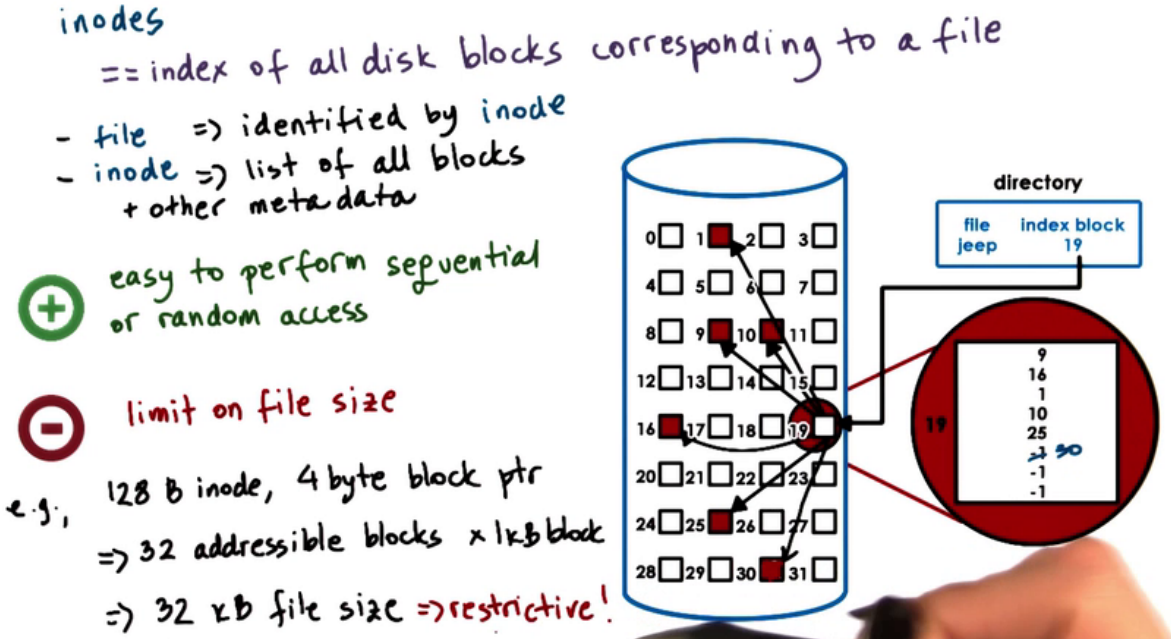
30 - ext2 Second Extended Filesystem



are the blocks that hold the file data. Again a reminder, a directory will really be just a file except that in the upper To make things concrete, let's look at a file system that's supported on disk devices. We will look at ext2 which stands for extended file system version 2. It was a default file system in several versions of linux until it was replaced by ext3 and then ext4 more recently that are the default versions in more current versions of linux. It is also available for other operating systems. It's not just linux specific. A disk partition that is used as a ext2 linux file system will be organized in the following way. The first block, block 0, is not used by linux and it often contains the code to boot the computer. The rest of this partition is divided into groups and exactly what are the sizes of the groups, that has nothing to do with the physics of the disks. What are the disk cylinders or sectors or anything like that. Each of the block groups in this partition will be organized as followed. The first block in a block group is the superblock and this contains information about the overall block group. It will have information about the number of inodes, about the number of disk blocks in this block and it will also have information about the start of the free blocks. The overall state of the block group is further described in the group descriptor and this will have information about the location of the bitmaps we'll describe what they mean next about the total number of free nodes, the total number of directories in the system. This information is useful when files are being allocated because ext2 tries to balance the overall allocation of directories and files across the different block groups. The bitmaps are used to quickly find a free block or a free inode, so for every single inode in this particular group and every single data block, the bitmap will be able to tell the upper layer allocators whether that inode component or that data block are free or they're used by some other file or directory. Then come the inodes. They're number from 1 up to some maximum number. And every one of the inodes is in ext2 a 128 byte long data structure that describes exactly 1 file. It will have information like what is the owner of the file, some accounting information that system calls like

stat would return and also some information how to locate the actual data blocks. So these are the blocks that hold the file data. Again, a reminder, a directory will really be just the file except that in the upper levels of the file system software stack there will be these dentry data structures created for each particular component for the particular directory.

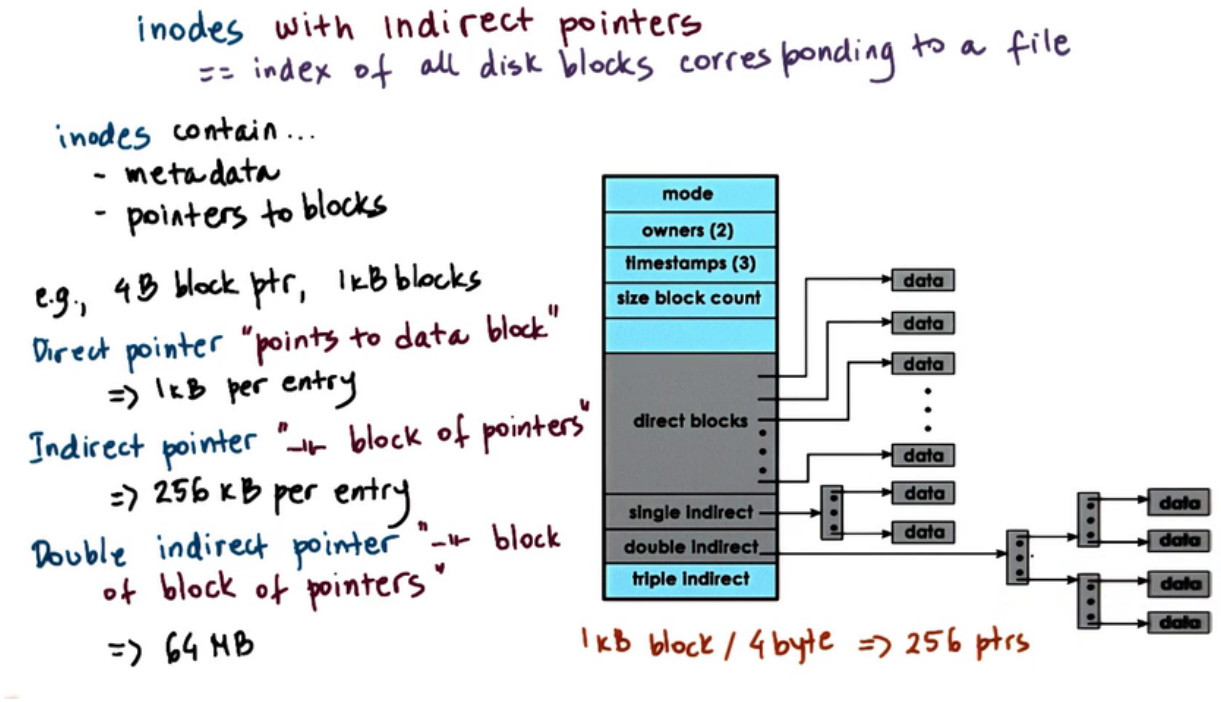
31 - inodes



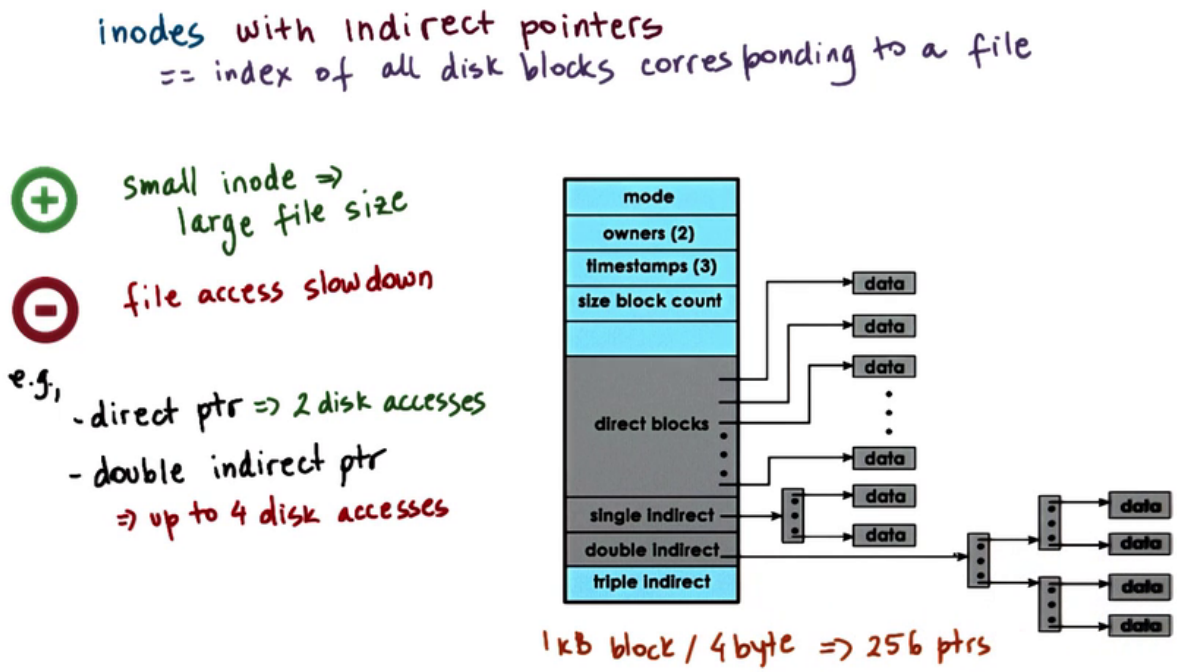
We said that inodes play a key role in keeping track how file is organized on disk because they essentially integrate an index of all of the disk blocks that correspond to a particular file. First, the file is uniquely identified by its inode. In the virtual file system, inodes are uniquely numbered so to identify a file we use the number identifier of the corresponding inode. This inode itself will contain a list of all of the blocks that correspond to the actual file. So it will be like an index into the blocks of the actual storage device that when stitched together give us the actual file. In addition to this list of blocks, the inode also has some other metadata information and thus useful to keep track whether or not certain file accesses are legal or to correctly update the status of the file, if it's locked or not locked or other things. One simple way in which this can be used is as follows. A filename is mapped to a single inode and an inode let's say here it corresponds to a block so the file jeep points to the block 19 that is the inode for this particular file. The contents of this inode block are all of the other blocks that constitute the contents of the file. If we look here, we see that this file has 5 blocks allocated to it and if it turns out that we need more storage for this file as we're appending, as we're writing more data into the file, the file system will allocate a free block, let's say in this case this block 30 it will correctly update the inode data structure, the list of blocks, to say that the next node in the system is this node 30 and so the actual representation of the file on disk will now look as follows. The benefits of this approach is that it's easy to perform both sequential or random accesses to the file. So for sequential, we just need to get the next index into this list of blocks. For random access, we just based on the block size need to compute which particular block reference do we need to find and so it's fairly straightforward to do this. The downside is that this limits the size of the file to the total # of blocks that can be indexed using this linear data structure. Let's take a look at this example. Let's say we have 128 byte inodes and let's say they only contain these indexes to the blocks on disk. Supposedly we have 4 bytes to address individual block on disk. So that

means that the maximum number of block pointers, of block addresses that can be included in this inode is 32 of those. That's if we don't have any metadata in the inode. If we assume that a single block is 1 KB, that means that the maximum number of a file that can be addressed using this inode data structure that's represented in this way is 32 KB. That clearly is very restrictive.

32 - inodes with Indirect Pointers

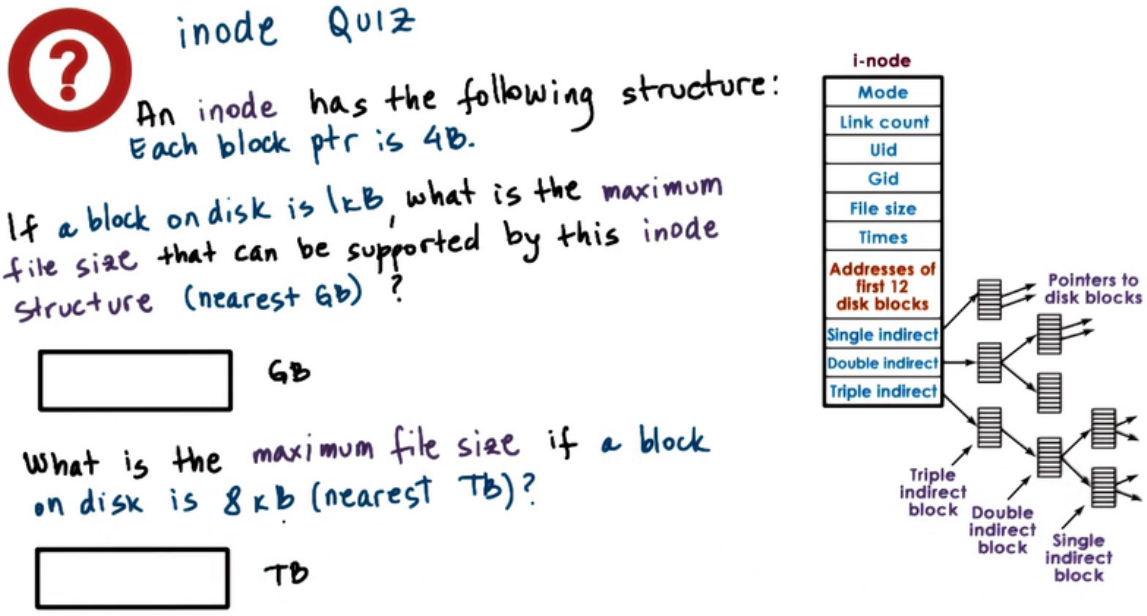


One way to solve this is to use so called indirect pointers. Here is an inode structure that uses these indirect pointers. Just like before, the inode contains metadata and pointers to blocks of data. The inode is organized in a way that first it has all the metadata, owner, when is the file last access, then it starts with the pointers to blocks. The first part is a list of pointers that directly point to a block on disk that stores the file data. Using the same example like before where we had blocks that are 1KB large and we use 4 bytes to address an individual block. These direct pointers will point to 1KB of data per entry. To extend the number of disk blocks that can be addressed via single inode element while also keeping the size of the inode small, we use so called indirect pointers. An indirect pointer as opposed to pointing to an actual data block will point to a block that's full of pointers. Given that our blocks are 1KB large, and our pointers are 4B large, that means that a single indirect pointer can point to 256KB of file content. So just using a single element of the inode data structure as this indirect pointer can significantly increase the overall size of the files that can be supported in this file system. Now if we need even larger files we can use double indirect pointers. A double indirect pointer points to a block which contains pointers to blocks of data. If every block has 256 pointers then this double indirect pointer can help us address 256*256*1KB blocks for a total of 64 MB of data. We can apply the same idea to triple indirect addressing and so forth.



The benefits of using indirect pointers is that it allows us to continue using relatively small inodes while at the same time addressing really large files. The downside is that this has an implication on slowing down the file access. With direct pointers, when we need to perform an access to a portion of the file we need to first access the inode itself and that may be stored somewhere on disk and then we will find out what is the pointer to a particular data block and will access the data block. So, we may end up performing 2 disk accesses per file access and that's at most. With double indirect pointers, the situation is very different. We need 1 disk access to get to the block that contains the inode. Then we may need another disk access to get to the first addressing block, then from there a second disk access to get to the second level addressing block and then finally we get to the block that contains the data. So for a single file operation we may end up performing up to 4 disk accesses. That's a 2x performance degradation.

33 - inode Quiz



Instructor Notes

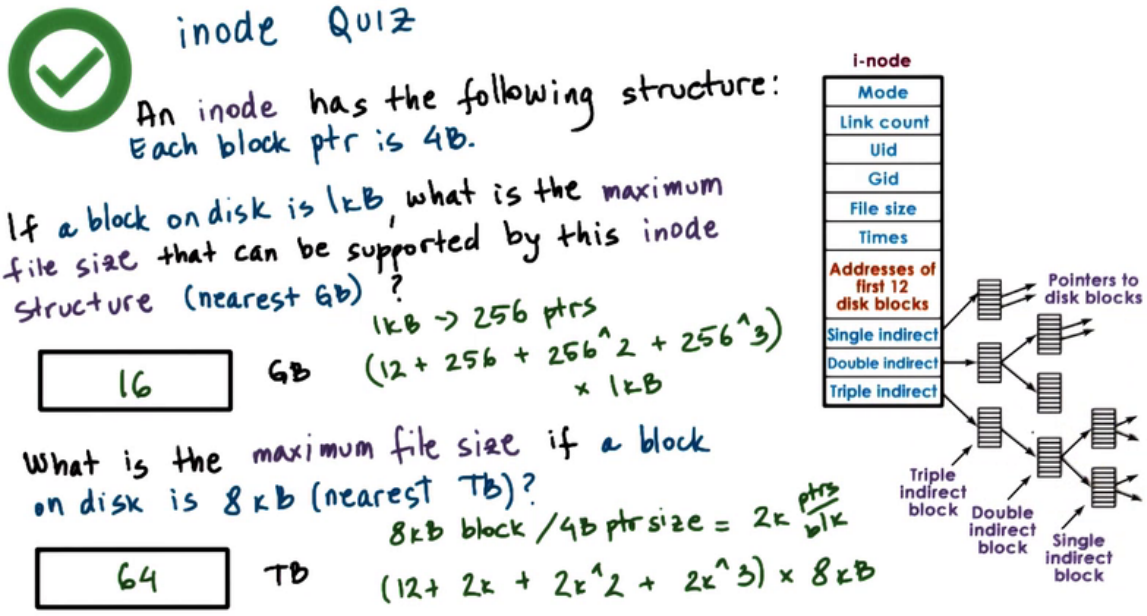
Quiz Help

Maximum File Size:

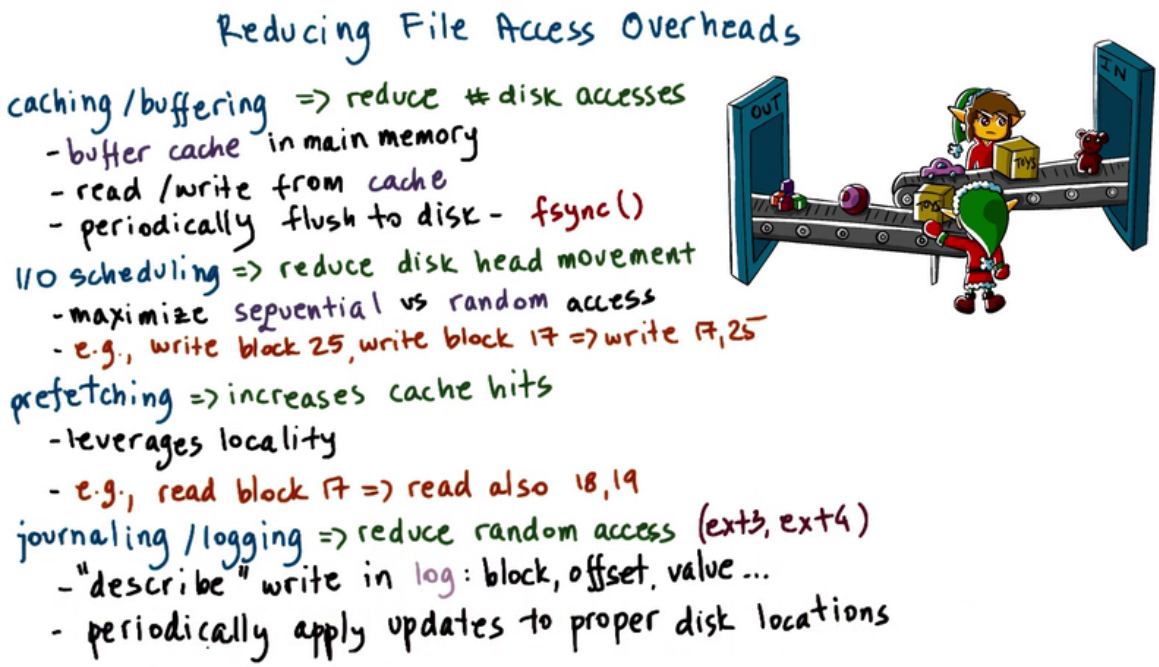
- number_of_addressable_blocks * block_size
- number_of_addressable_blocks = 12 + blocks_addressable_by_single_indirect + blocks_addressable_by_double_indirect + blocks_addressable_by_triple_indirect

You do not have to include units in your answers.

34 - inode Quiz



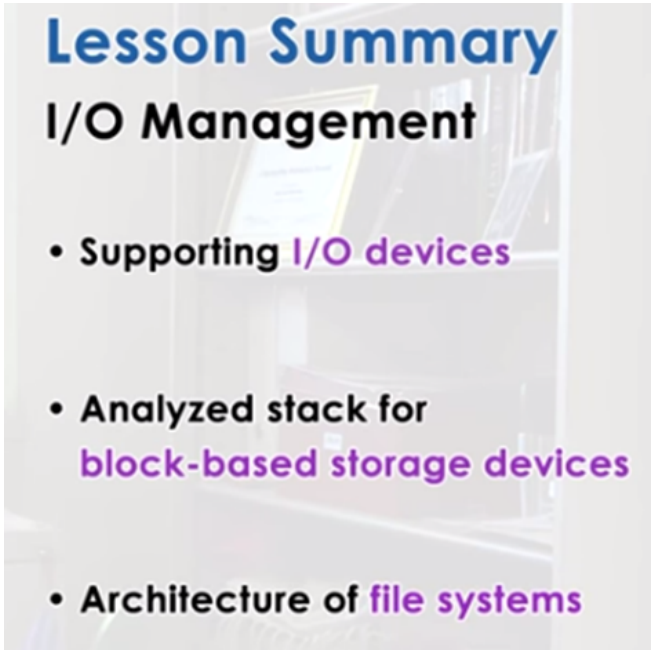
35 - Disk Access Optimizations



File systems use several techniques to try to minimize the accesses to disk and to improve the file access overheads. Much like hardware caches are used to temporarily hold recently used data and avoid accessing main memory, file systems rely on buffer caches except these buffer caches are present in main memory and they're used in order to reduce the number of necessary disk accesses. With caching, the file will be read or written to from the cache and periodically any changes to the file that have not been backed up on permanent storage will be

flushed from memory to disk. By forcing these flushes to happen only periodically, we can amortize the cost of performing disk access over multiple intermittent writes that will hit the cache. File systems support this operation using the fsync system call. Another component that helps reduce the file access overheads is what we call I/O scheduling. This is the component that orders how disk access operations will be scheduled. The intent is to reduce the disk head movement that's a slow operation and we can do that by maximizing the number of sequential accesses which are needed, so for sequential accesses the disk head movement is not expensive. It's expensive for these random access so that's what we want to avoid. What that means is let's say we have two operations that have been issued, write block 25 followed by write block 17 and if the disk head is at position 15 these operations will be re-ordered by the I/O scheduler so that they're issued as write first block 17 and then block 25. And this will achieve this objective of maximizing sequential and minimizing random accesses. Another useful technique is prefetching. Since for many workloads there is a lot of locality in how the file is accessed, it is likely that if one data block is accessed, the subsequent data blocks will be accessed as well. File systems can take advantage of this feature by prefetching more than 1 block of a file whenever a single block is accessed. This does use up more disk bandwidth to move larger in this case 3x worth of data from disk into main memory but it can significantly impact or reduce the access latency by increasing the cache hit rate as more of the accesses will be served out of cache potentially. Finally, another useful technique is journaling. I/O scheduling reduces the random access but it still keeps the data in memory so these blocks 17 and 25 are still in memory waiting for the I/O scheduler to interleave them in the right way. That means that if something happens and the system crashes, these data blocks will be lost. So we want to make sure the data ends up on disk, but we still want to make sure that we reduce the level of random access that's required. This is where journaling helps. As opposed to writing out the data to the proper disk location which will require a lot of random access, we write updates in a log. So, the log will contain some description of the write that's supposed to take place. If we specify the block, the offset, and the value that essentially describes an individual write. Now, I'm over trivializing this, there is a little bit more that goes into it but this is the overall nature of the journaling or the logging based systems. The follow on to ext2 called ext3 and ext4 that are also part of current linux systems they use journaling as well as many other file systems. Note that a journal has to be periodically updated into a proper disk location otherwise it will just grow indefinitely and it will be really hard to find anything. So if we look at these 4 techniques in summary, every single one of them contributes to reducing the file system overheads and latencies. This is done by increasing the likelihood of accessing data from memory, by not having to wait on slow disk head movements, by reducing the overall number of accesses to disk, and definitely the number of random accesses to disk. These techniques are commonly used in file system solutions.

36 - Lesson Summary



In this lesson we learned how operating systems manage I/O devices. In particular we talked about the operating system stack for block based storage devices. And in this context, we also talked about file systems and how they manage how files are stored on such block devices. For this, I gave as an example some details of the linux file system architecture.