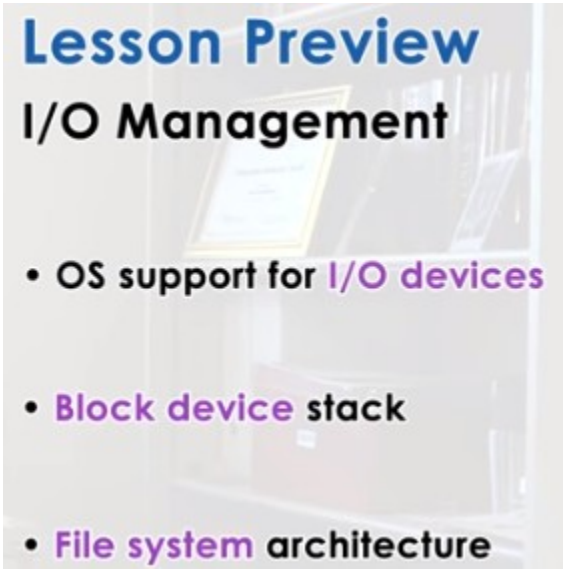
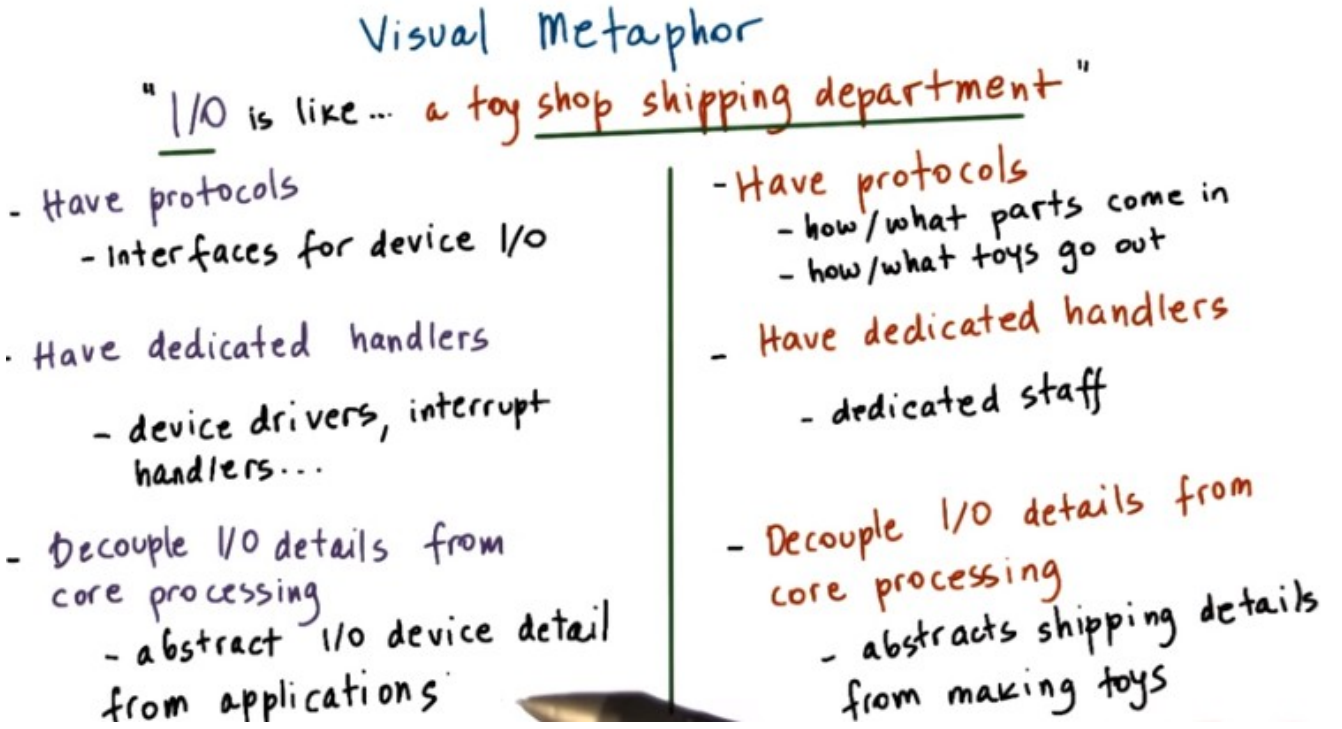
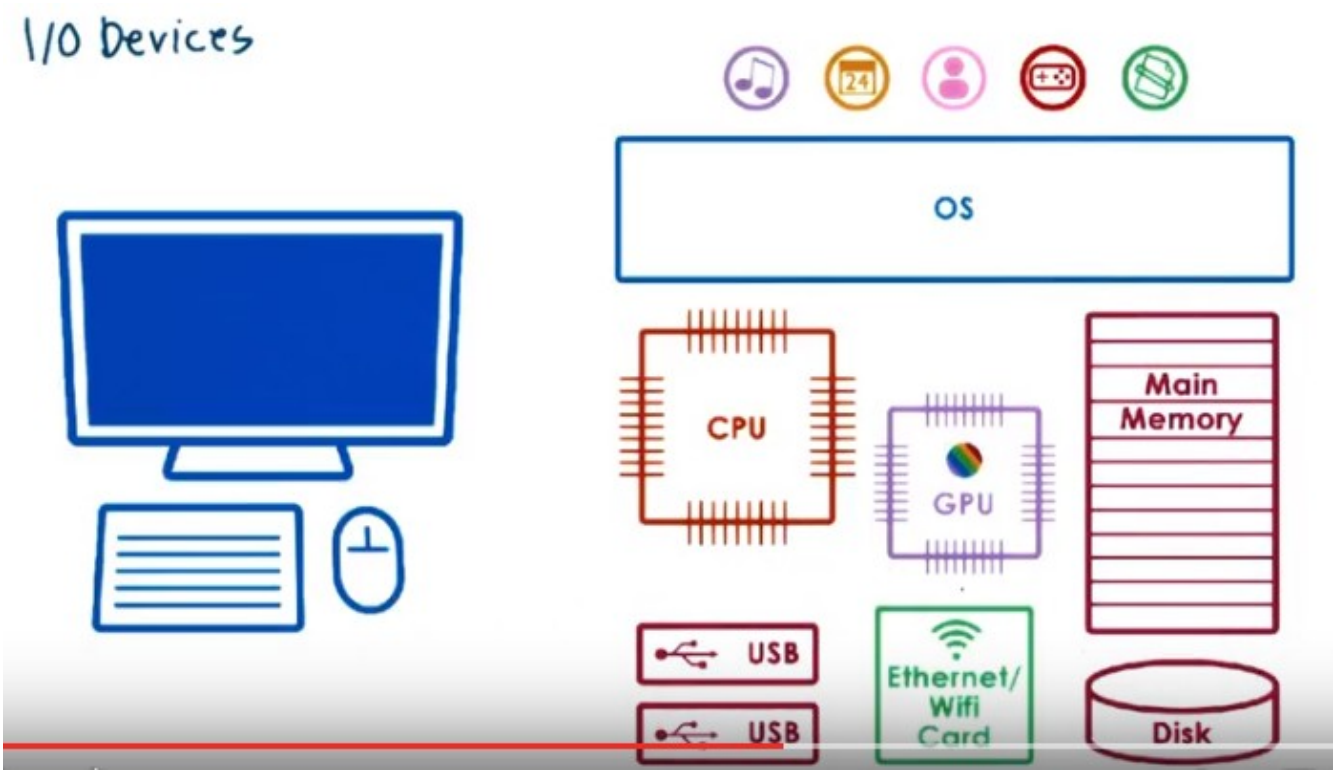


1. We said that a main role of operating systems is to manage the underlying hardware. But, so far, we talked primarily about CPUs and memory. In this lesson, we will focus on the mechanisms that operating systems use to represent and manage I/O devices. In particular, we will look at the operating system 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.



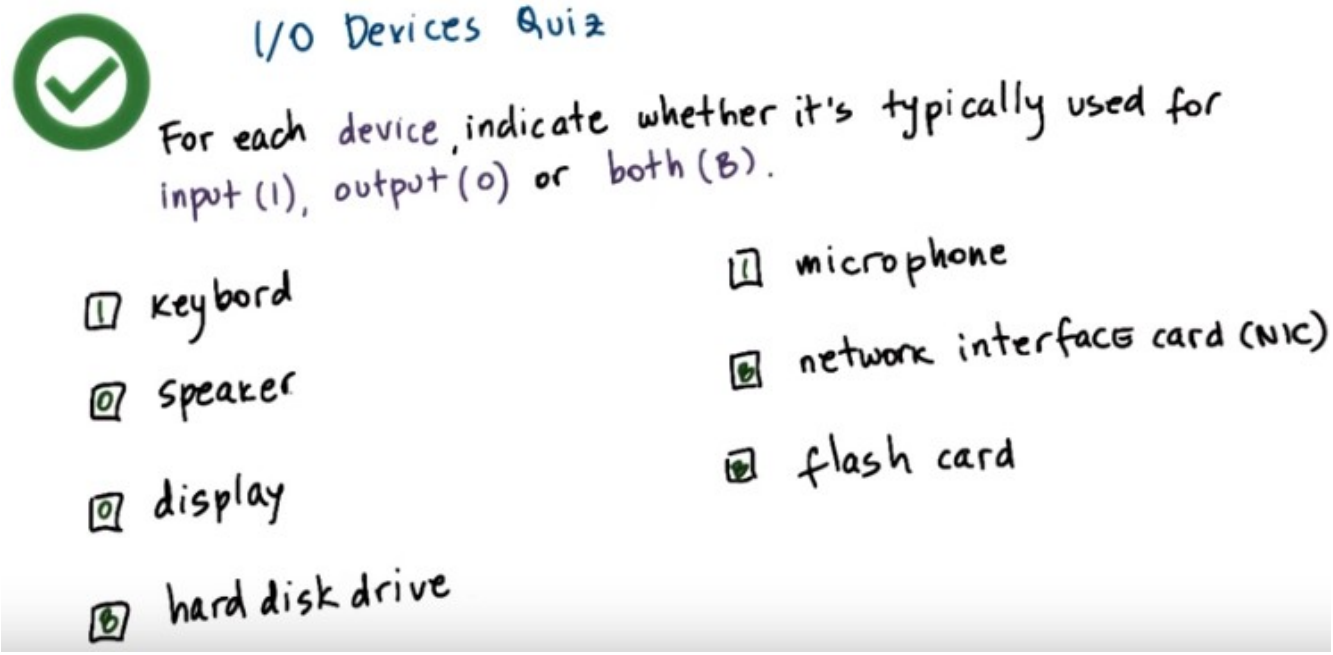
2. I/O management is all about managing the inputs and the outputs of a system, to illustrate some of the tasks involved in I/O management, we will draw a parallel between I/O in computer systems and a shipping department in a toy shop. Let's see what are some of the similarities we can find between how

operating system manages I/O and how the shipping department is managed in a toy shop. For instance, both in the toy shop and in the operating system there will be protocols, in terms of how the needs to be managed, in both cases there will be dedicated handlers or some dedicated staff that will oversee these protocols. And in both environments the details about the I operations are the cop out 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 out in 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 that are being used? What are their protocols? Are there any slaves that are being used? It doesn't matter. This has analogies 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 their device drivers, their 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're 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.

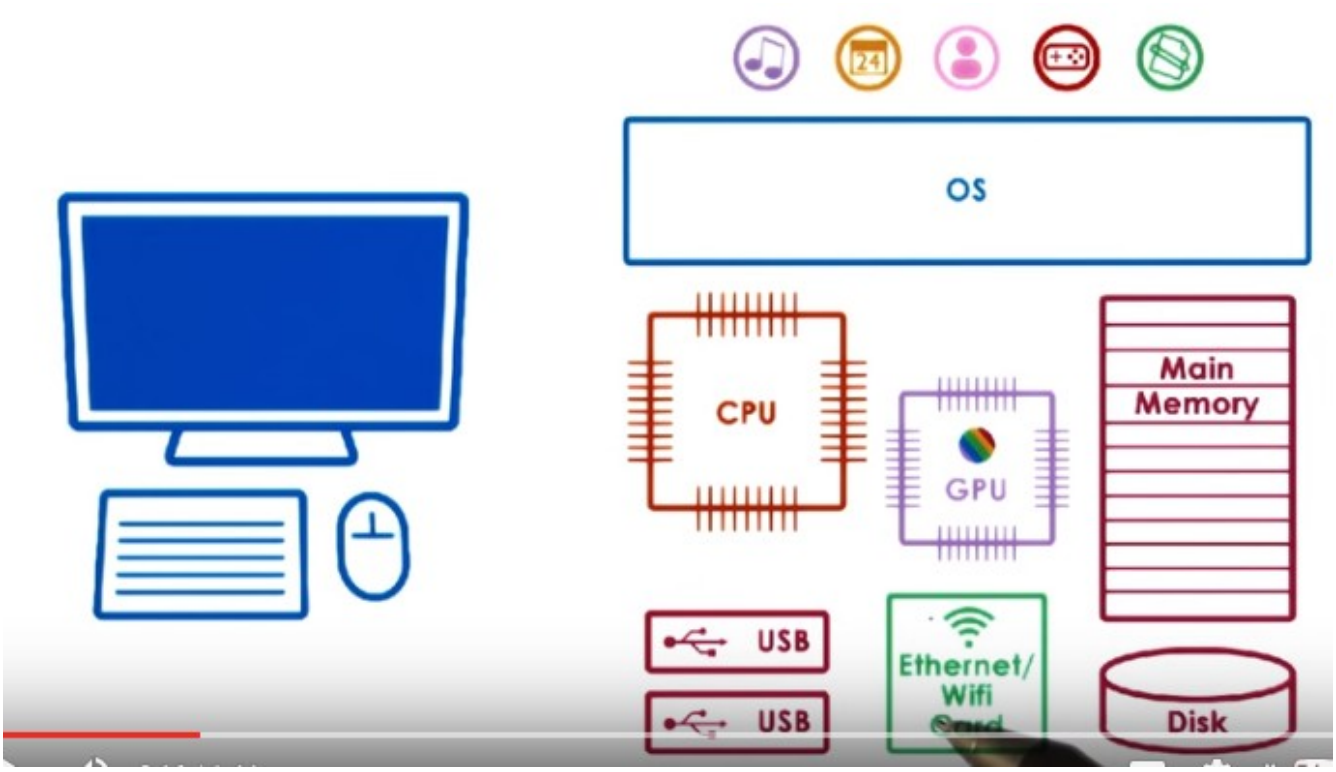


3. Here is the illustration of a computer system we used in our introductory lesson in operating systems. As you can see, the execution of applications doesn't rely only on the CPU and memory, but relies on many other different types of hardware components. Some of these components are specifically tied 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 hard disk. So there are a number of different types of input/output devices that operating systems integrate into the overall computing system.

4. Here's a quick quiz, for each device indicate whether it is typically used for input, for output, or for both? The devices mentioned are keyboard, speaker, display, hard disk drive, microphone, network interface card, or NIC, flash cards.



5. Your answer should be as follows. Keyboards and microphones are input devices. Speakers and displays are both output devices. And all of the remaining devices can be used both for input and output. In addition to these types of devices, there are many other types of devices. And then within each of these categories there are many concrete examples. Many different types of microphones. Many different types of speakers or network interface cards. Operating systems must be designed in a way that can handle all of these different types of devices efficiently and in a straightforward manner.



6. 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 simply our discussion in this lesson, we'll point out the key features of a device that enable the integration of devices into a system.

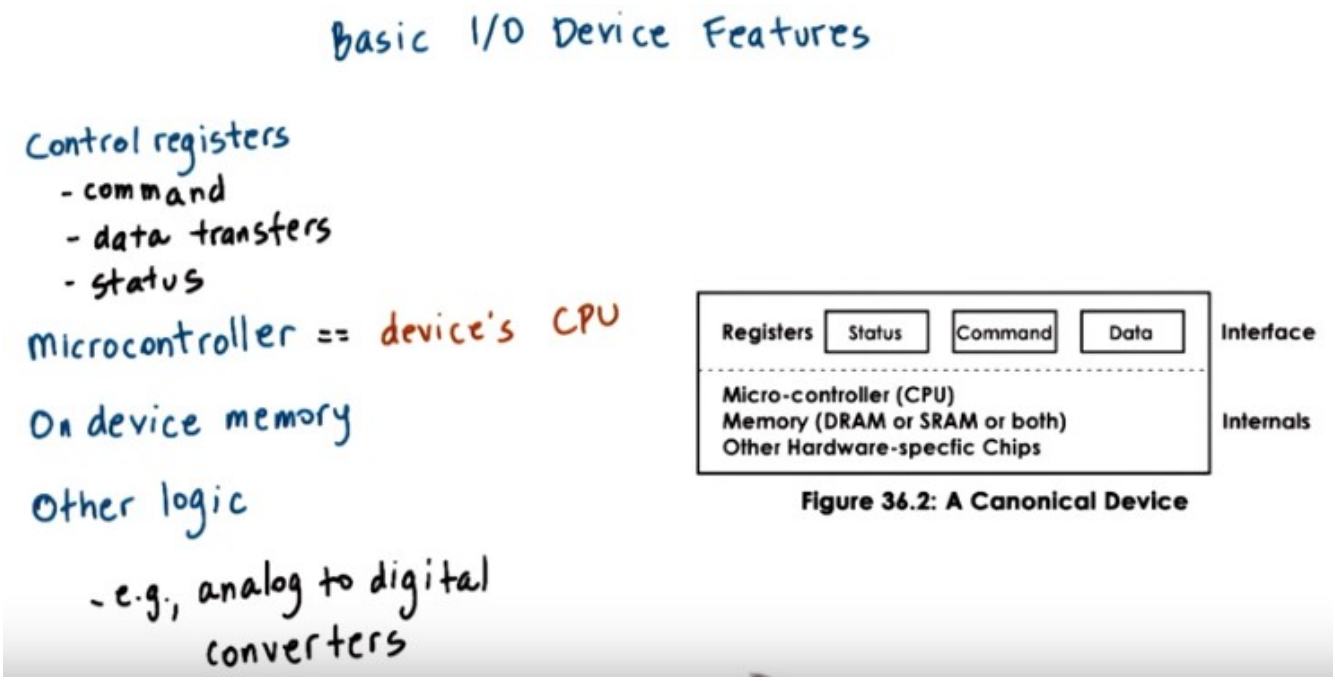
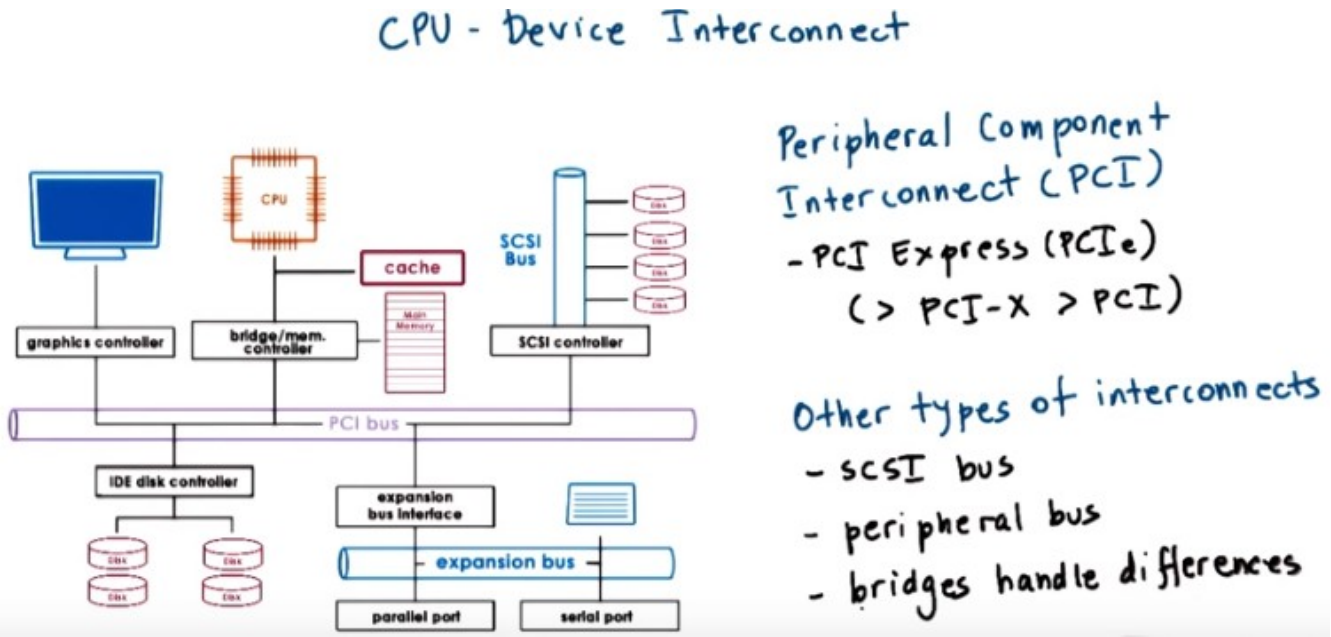
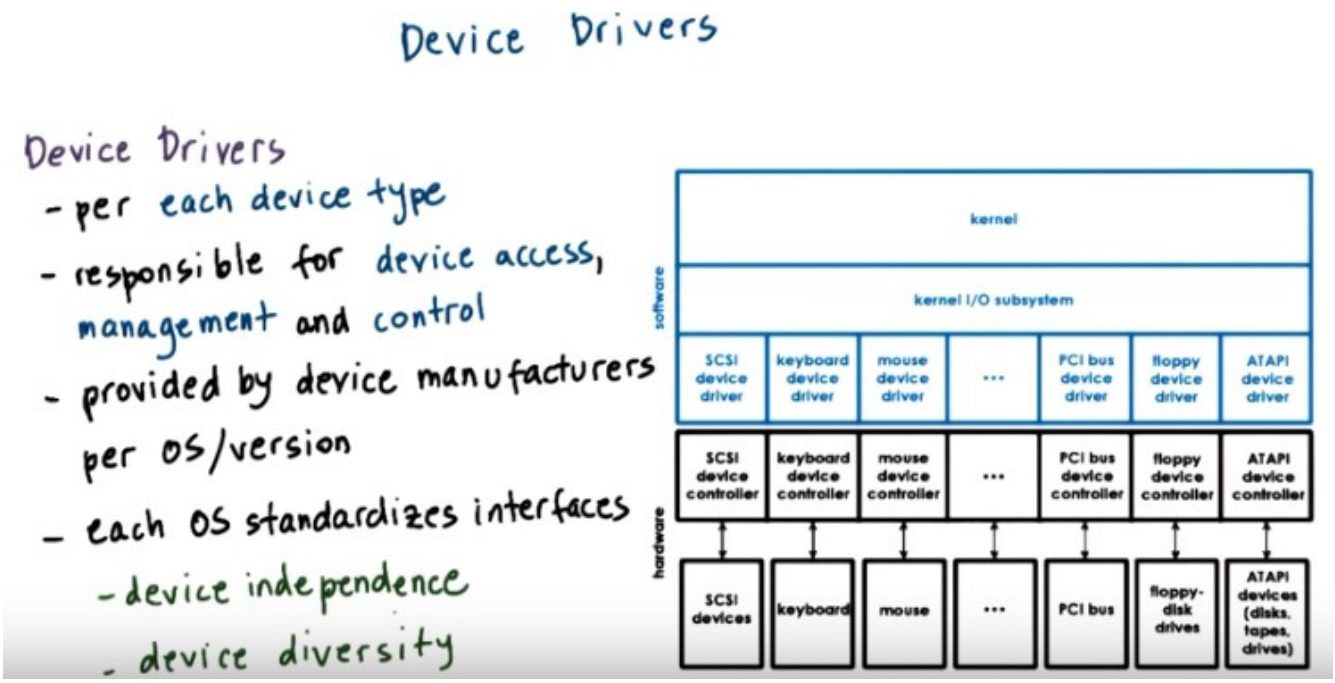


Figure 36.2: A Canonical Device

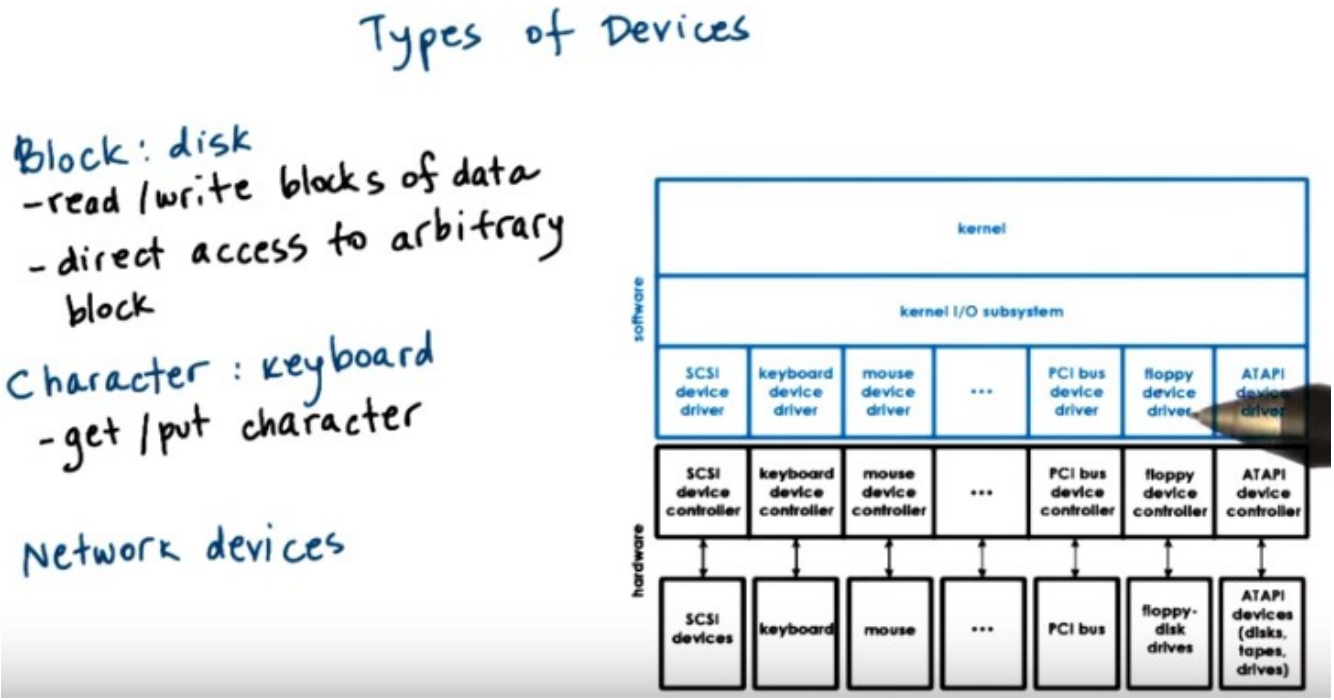
Any device can be abstracted to have the following set of features. Any device will have a set of control registers that can be accessed by the CPU and that permit the CPU device interactions. These are typically divided into command registers that the CPU uses to control what exactly the device will 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. Then there's some amount of on device memory, any other types of processing clauses, special chips, special hardware that's needed on the device. Like for instance, you convert analog to digital signals to actually interact with the network medium, with the physical medium. Whether it's optics or, or copper, whatever else, so all of those chips will be part of the device.



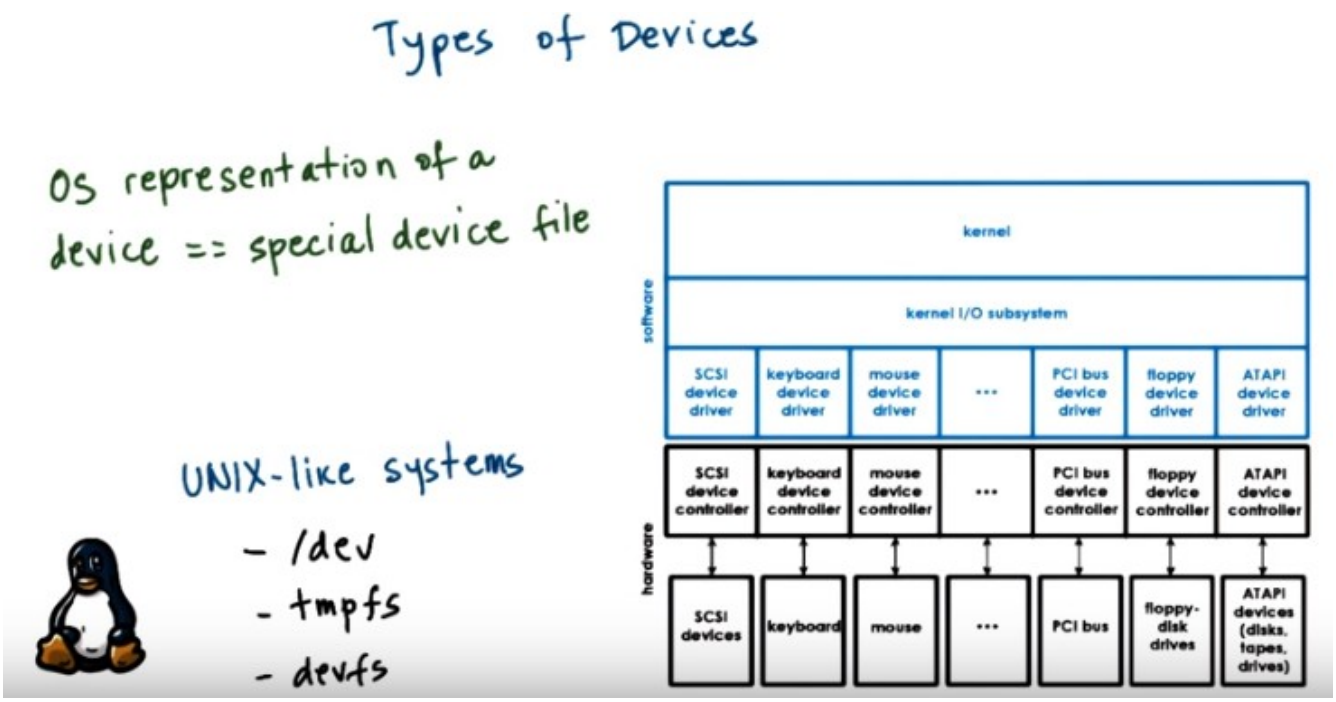
7. Devices interface with the rest of the system via controller that's typically integrated as part of the device packaging, that's used to connect the device with the rest of the CPU complex via some CPU device interconnect. Whatever off-chip interconnect is supported by the CPU that different devices can connect to. This figure that's adapted from the Silver Sheds 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 standard methods for connecting devices to the CPU. Today's platforms typically support PCI Express interconnect, 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. In all aspects, it is better than PCI-X which was the follow on on the original PCI standard. For compatibility reasons, though, today's platform 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 shown in this figure alone. There is also a SCSI bus that connect SCSI disks, 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 bridge and controllers that can handle any differences between different types of interconnects



8. 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 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 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 a device driver's. Typically this is done by providing some device driver framework so that device manufacturers can develop the specific device drivers within that framework, given specific interfaces that that 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 disk 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 but the rest of the operating system, the rest of the applications will not be affected. So there's standardized interfaces both in terms of the interaction with those devices as well as in terms of the development and 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 device. And also device diversity. It's an easy way for the operating system to support arbitrarily different types of devices. We just need another device driver.

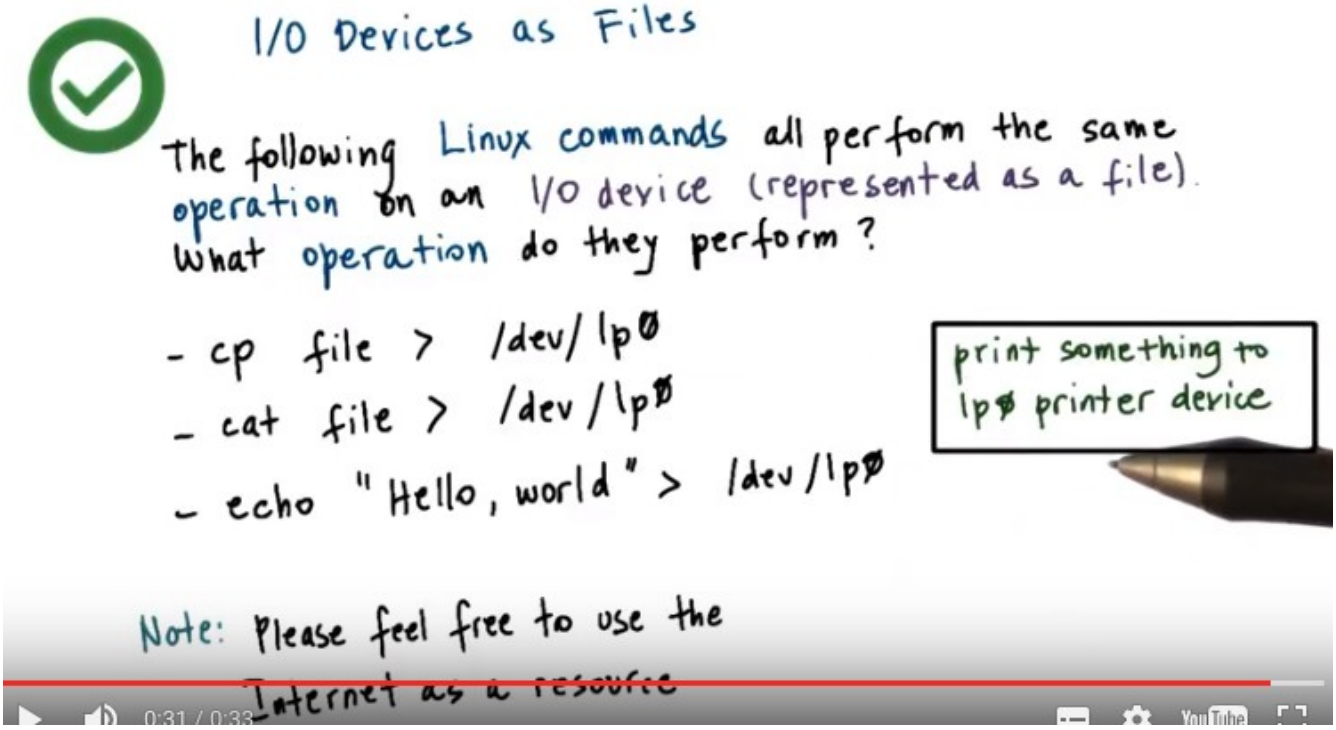


9. To deal with this great device diversity, devices are roughly grouped in several categories. This includes block devices like disks (此處的 block 不是堵住的意思, 而是一块數据的意思). These are devices that operate at granularity of entire blocks of data that's delivered in and out of the device and in and out of the CPU complex in the other end. A key property is that individual blocks can be directly accessed. For instance if you have ten blocks of data on the disk, you can request to directly access the ninth one. 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 system to the devices are standardized based on the type of device. For instance for a block devices, the driver should support operations to read/write block of data, for character devices, a driver should support operations to put/get a character from, into the device. So, in that sense standard.



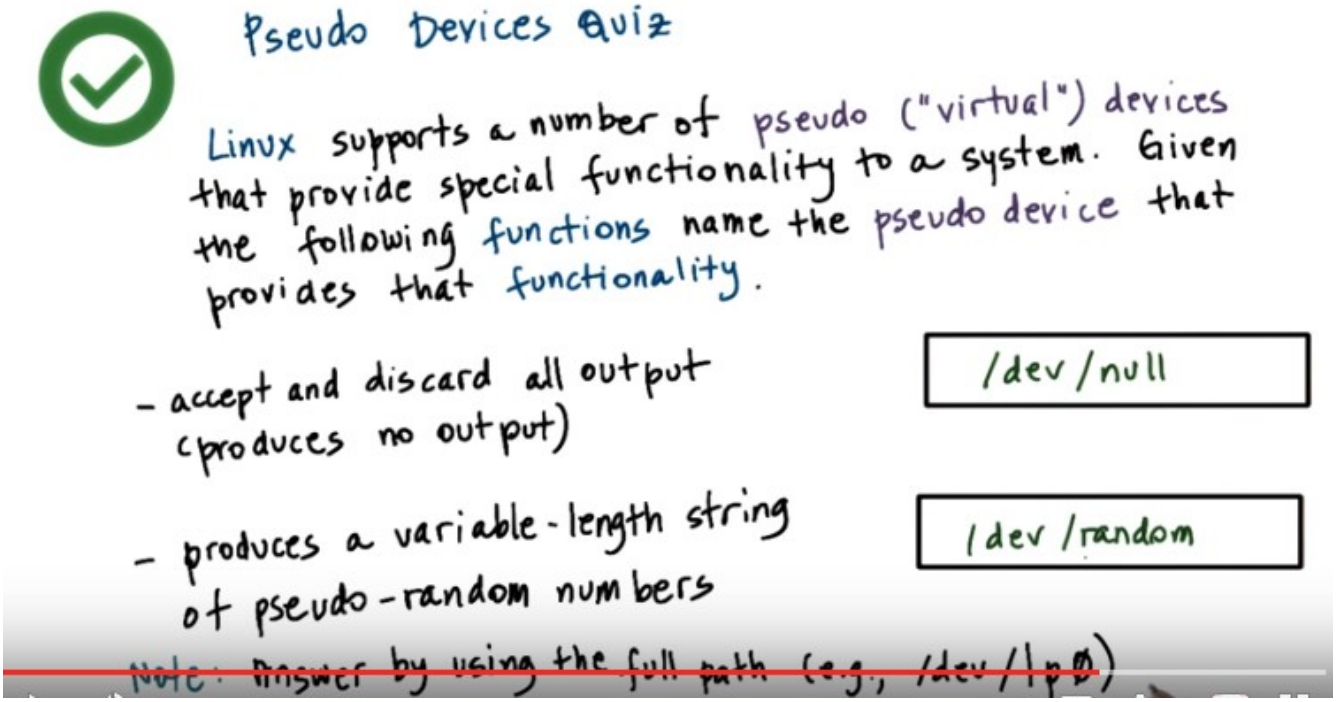
Internally, the operating system maintains some representation for each of the devices that are available on the platform. This is typically done by using a file abstraction to represent the different devices. And in that way what really the operating system achieves is that it can use any of the other mechanisms that are already part of the operating system to otherwise manipulate files. Too, now think mean to refer 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 this /dev directory. As special files, they're also treated by a special file system. They're not really stored on the real file system. In Linux versions, these are tmpfs and devfs.

10. As we just stated, Linux represents devices as special files and the operations on those files have some meaning that's device specific. As a quiz, look at the following three Linux commands which perform the same operation with an I/O device. What operation do they perform? Type your answer in the text box and feel free to use the internet as a resource.



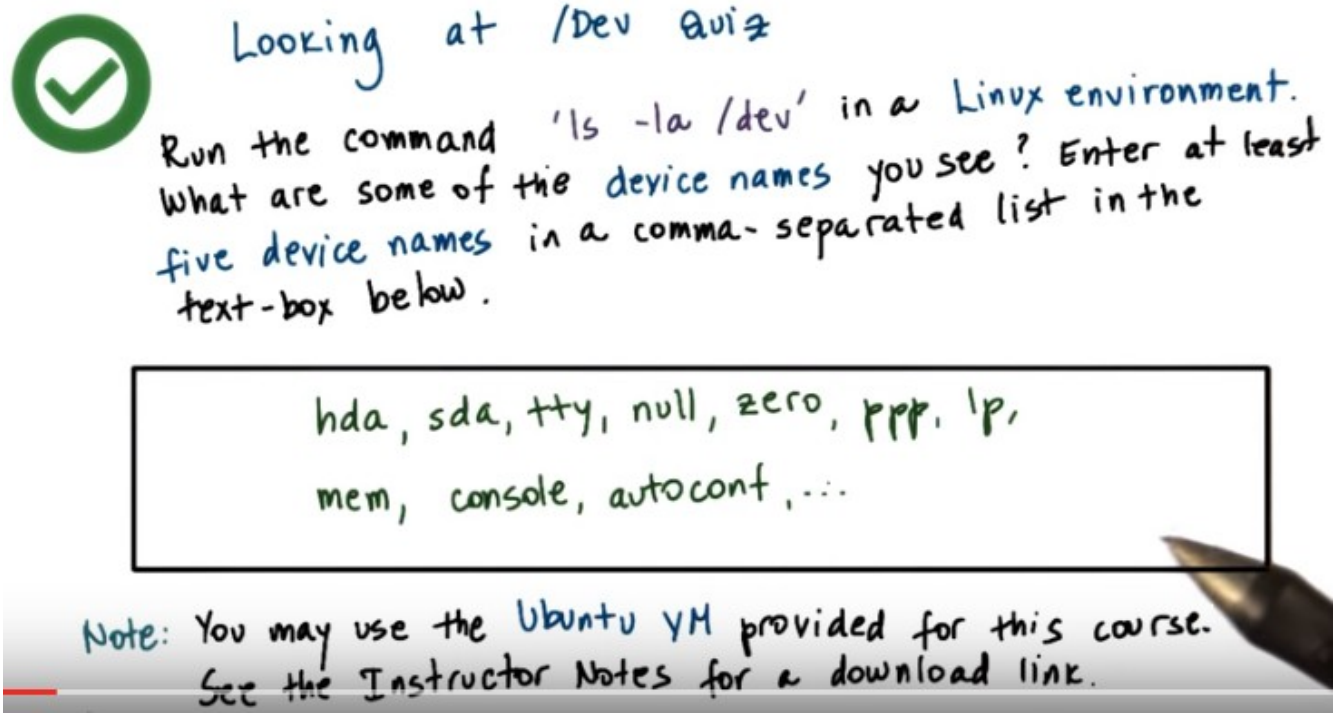
11. If we take a look at these commands, we see that in all of these cases, there's some content, whether the content of file, or the string "Hello, world" that's being written into a file, /dev/lp0. 'lp' stands for line printer and 0 stands for the first line printer that's identified by the Linux system. What these commands mean that we're printing something. So the correct answer is that each of the commands are printing something to the lp0 printer device.

12. To go in a little bit deeper into the special device files, Linux also supports what are called pseudo or virtual devices. These devices do not represent an actual hardware and are not critical in our understanding of I/O management, but are useful nonetheless. As a quiz, I will ask you to look at the following function descriptions and name the pseudo device that provides that function. First, what is a device that accepts and then discards all output without actually producing anything? Second, what is a device that produces a variable-length string of pseudo-random numbers? For your answers, please fill them in, in these boxes, and also use the full path to the device to specify what is the correct answer. For instance to answer that the devices line printer is here up, use path /dev, /lp0.

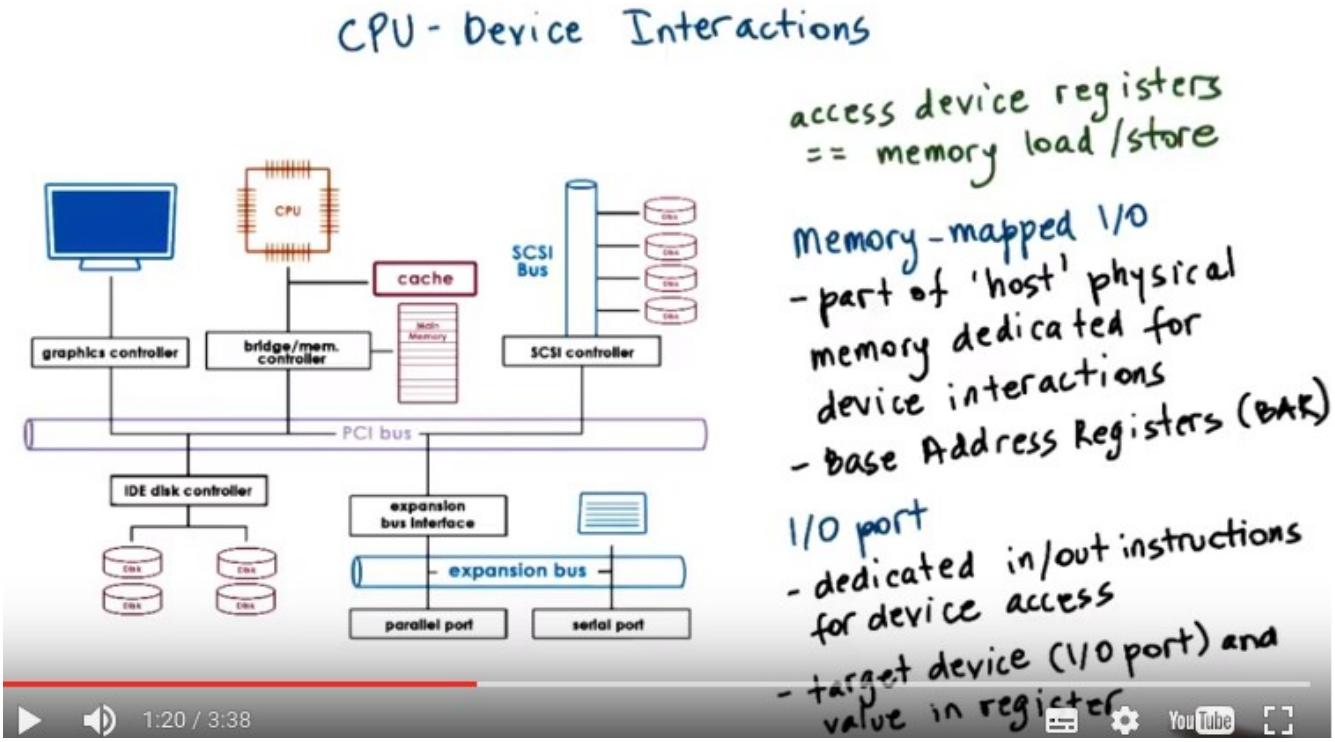


13. The answer to the first question is /dev/null. For instance, if you just want to discard the output of a process you can just send it to /dev/null. The answer to the second question is the device /dev/random. It also has a more robust counterpart, /dev/urandom, and one use of this device is to create files that contain pseudo-random bytes.

14. As an exploratory quiz, try running the following command in a Linux environment. This command lists the contents off a directory, in this case, the /dev directory. The question to you, then, is what are some of the device names that you see when you run this command? Enter at least five of those device names in a comma separated list in this text box. And, if you don't have access to some other Linux environment, you may use the Ubuntu VMs that are provided this course. The instructor notes have a download link.

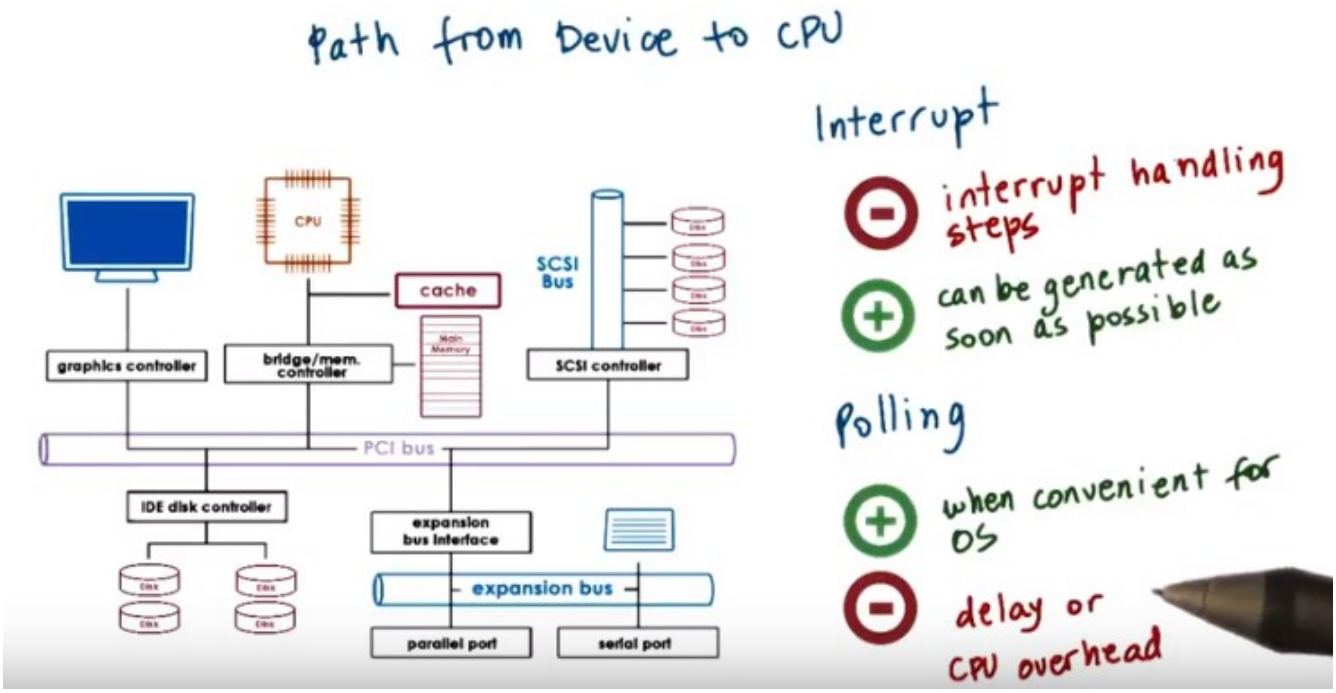


15. Here are some of the possible answers to this question. You may have noticed hda or sda devices, these are drives, like hard drives, SSDs, or even CD-ROMs. Also, you may have noticed some tty devices. These are special devices representing terminal stations. They can actually be used to pipe output to and from terminals. There are many other devices, the null device that we already mentioned, lp for printer, and many more.



Device register 見第 6 段

16. 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's reserved for these interactions is controlled by a specific set of registers, the Base Address Registers. So how much memory in starting it, which address, 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, in x86 platforms, there are 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 going to be stored in registers. That's the value 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 CPUs complex can take two routes. Devices can generate interrupts to the CPU, or the other option is for the CPU to poll the device by reading its status register in order to determine, does the device have 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 associated with both of these methods, and as always, there are trade-offs between the two. With interrupts, the problem is due to the handling of the interrupt routine, the interrupt handler. There are the actual steps involved in the handling of the interrupt routine. There's several 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 effect 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 and in the way that 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 CPUs in the system or the system is busy otherwise doing other things. But our 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 the input data rate, the output data rate that needs to met, and number of other factors.

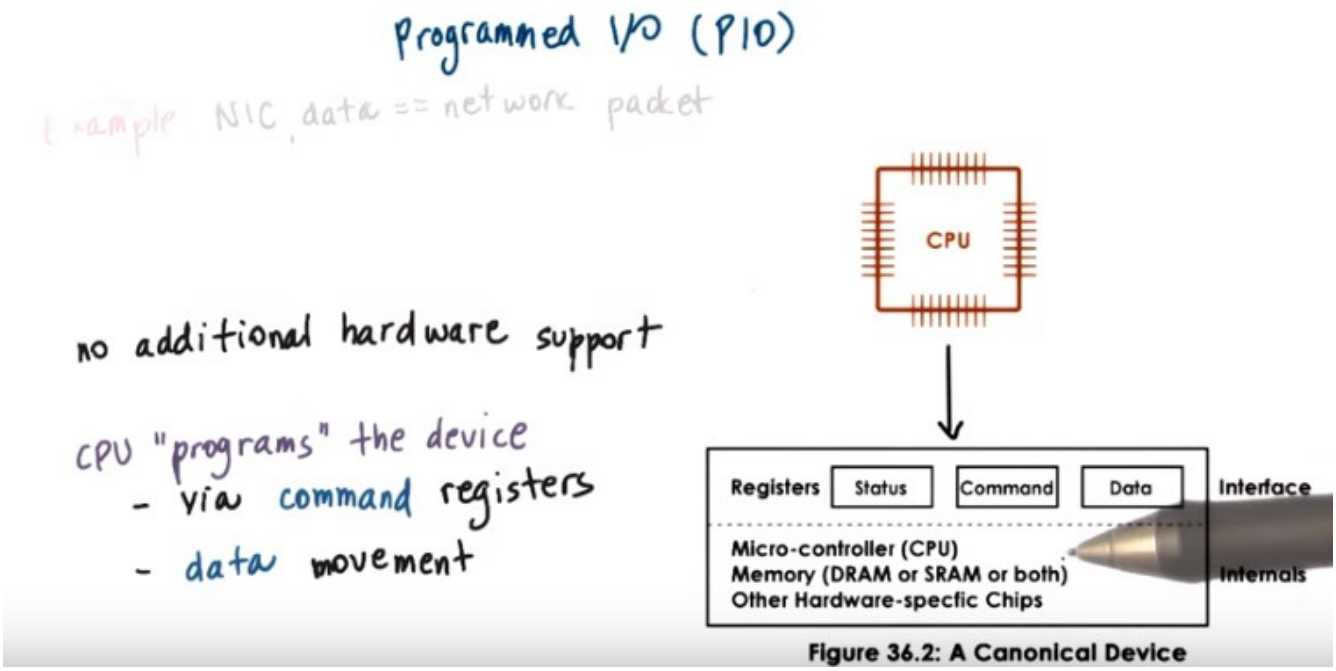
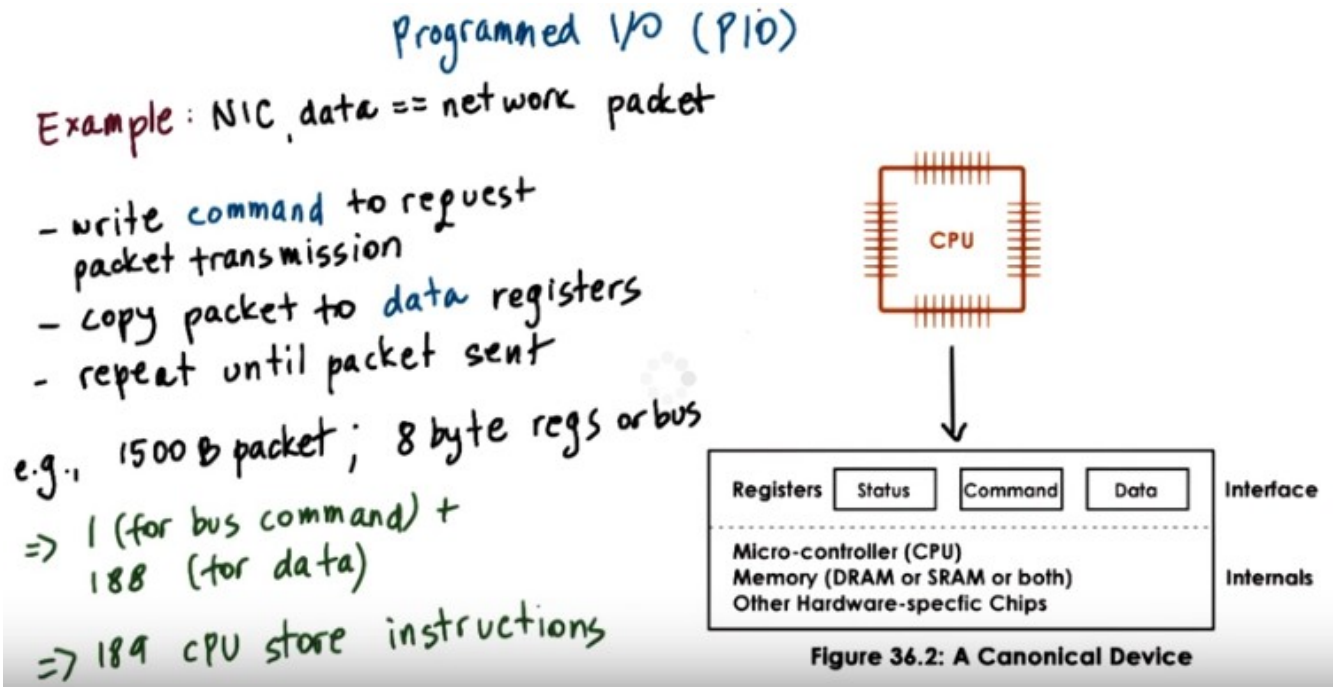


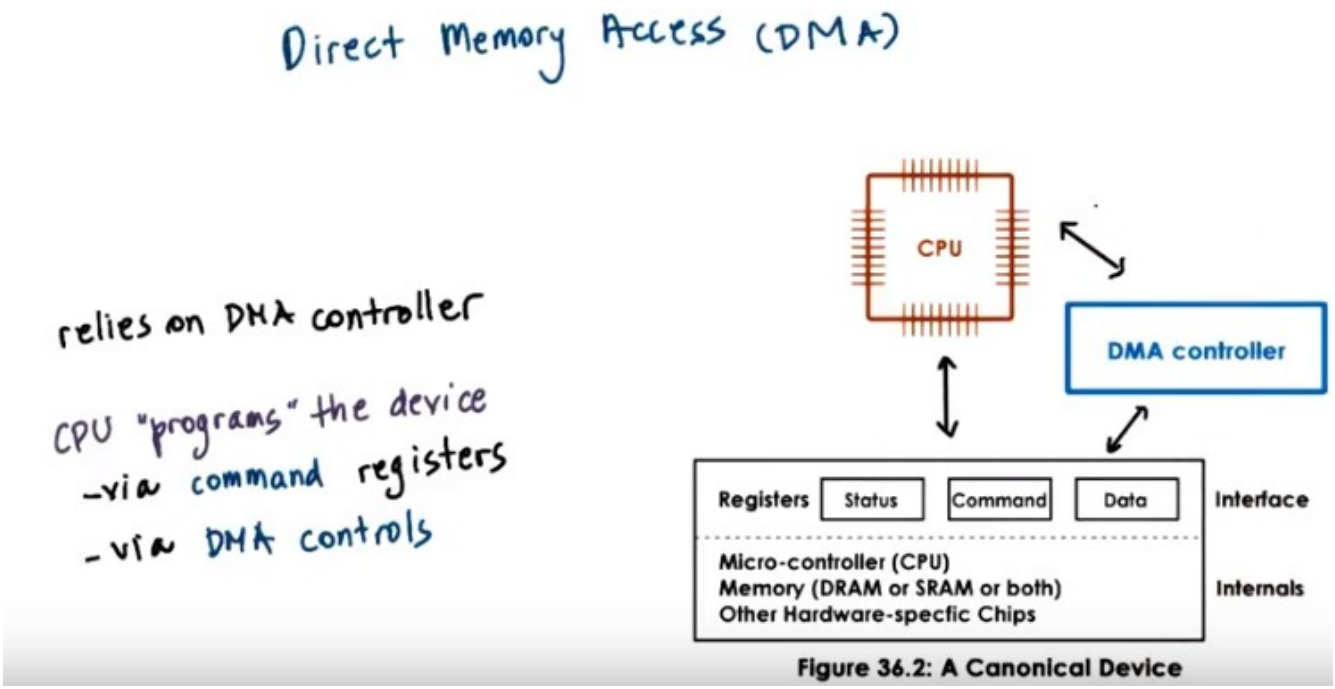
Figure 36.2: A Canonical Device

17. With just basic support from an interconnect like PCI and the corresponding PCI controllers on the device, a system can access or 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.

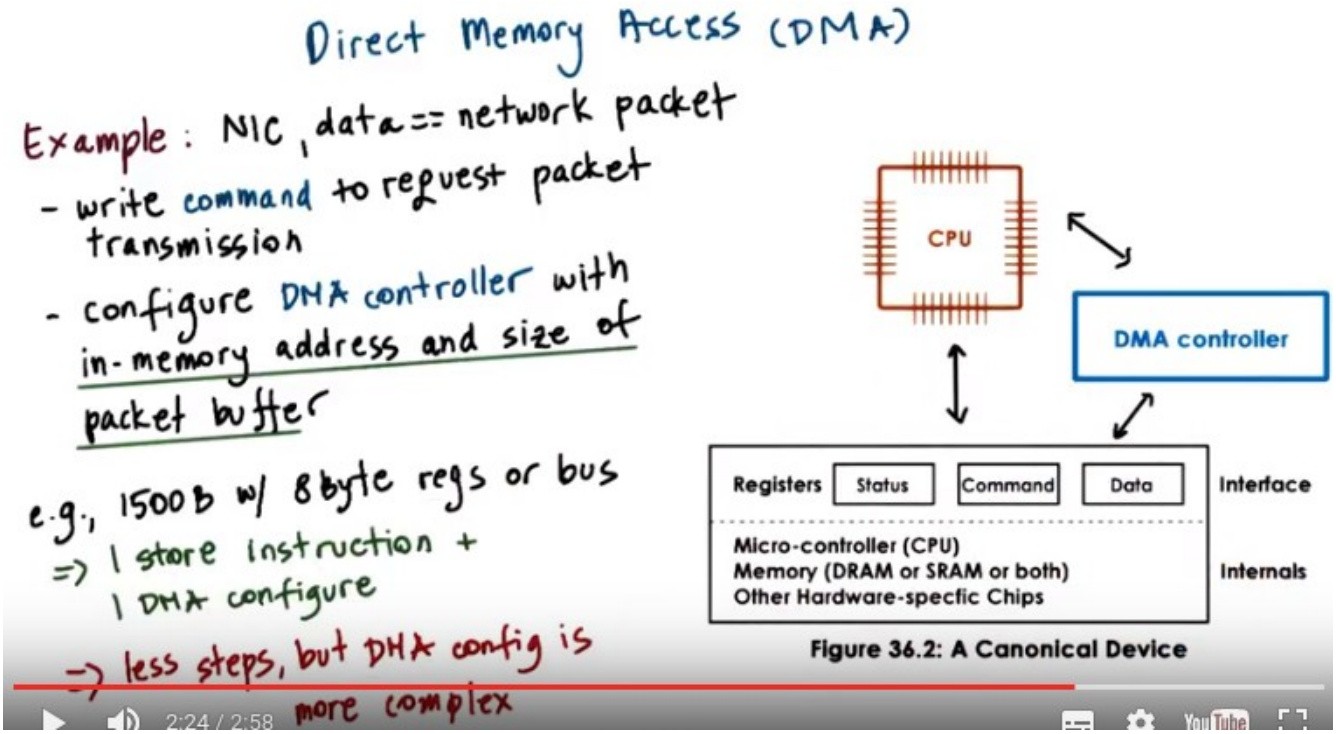


NIC: network interface card

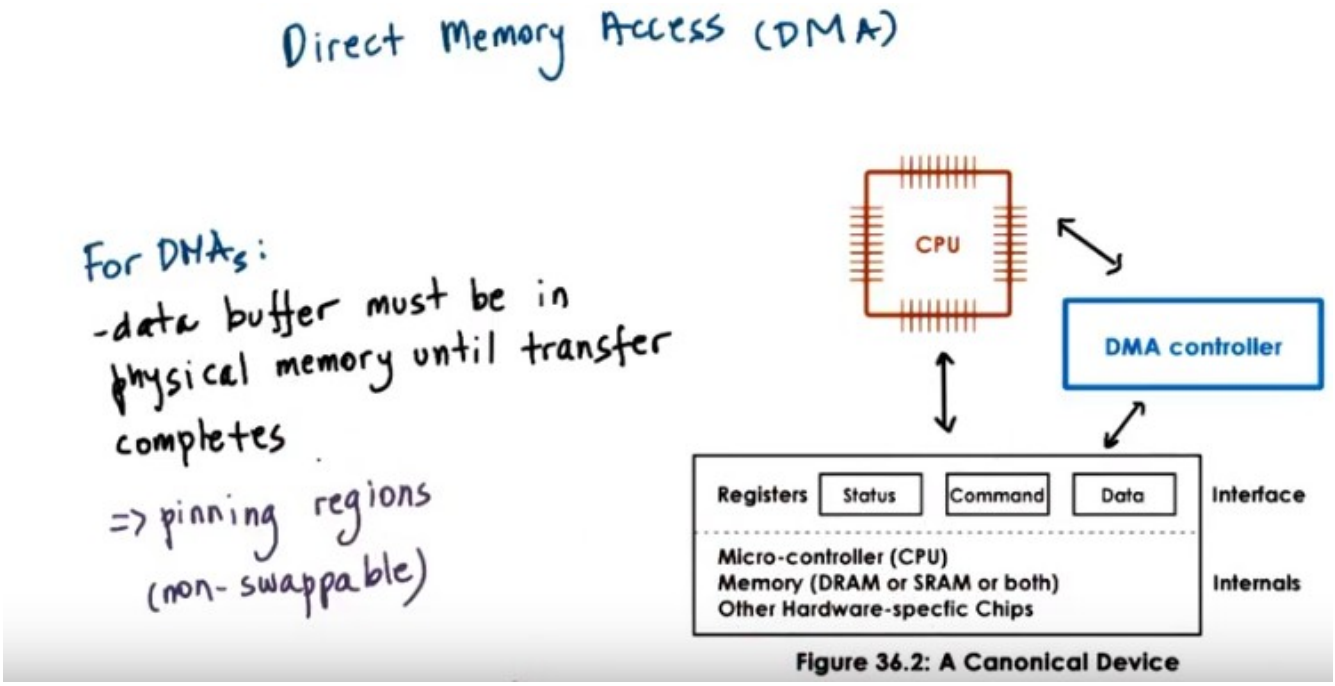
For instance, let's look at a network interface card or a NIC as a simple device that's accessed via programmed I/O or PIO. Here in the NIC case, when a process that's running from the CPU wants to transmit some data (即 CPU 傳數据給 device) that has been formatted into an network packet (即要傳的數据是以 packet 的形式存在的), 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 packets 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 they have a 1500-byte packet and the device data registers (由此可知 data register 只是起中間傳送的作用, 不是數据最終要傳到的目的地, 可以將 data register 理解為 project 3 中的 shared memory segment, 一個 segment 只用來傳文件) or the bus that connects the device with the PCU are 8 bytes long, then the whole operation of performing programmed I/O will require one CPU access to the device registers to write out the command, and then another 188 accesses, so 1500 divided by 8, approximately 188. In total, there will be 189 CPU accesses to the device-specific registers. And, we said that these look like CPU store instruction. This gives us some idea about the costs associated with programmed I/O. Let's now look at the alternative.



18. An alternative to programmed IO is to use DMA-supported devices (注意此種情況還是 PIO). DMA stands for direct memory access, and it is a technique that relies on special hardware support in the form of 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 as DMA controller that it interacts with to enable that interaction.

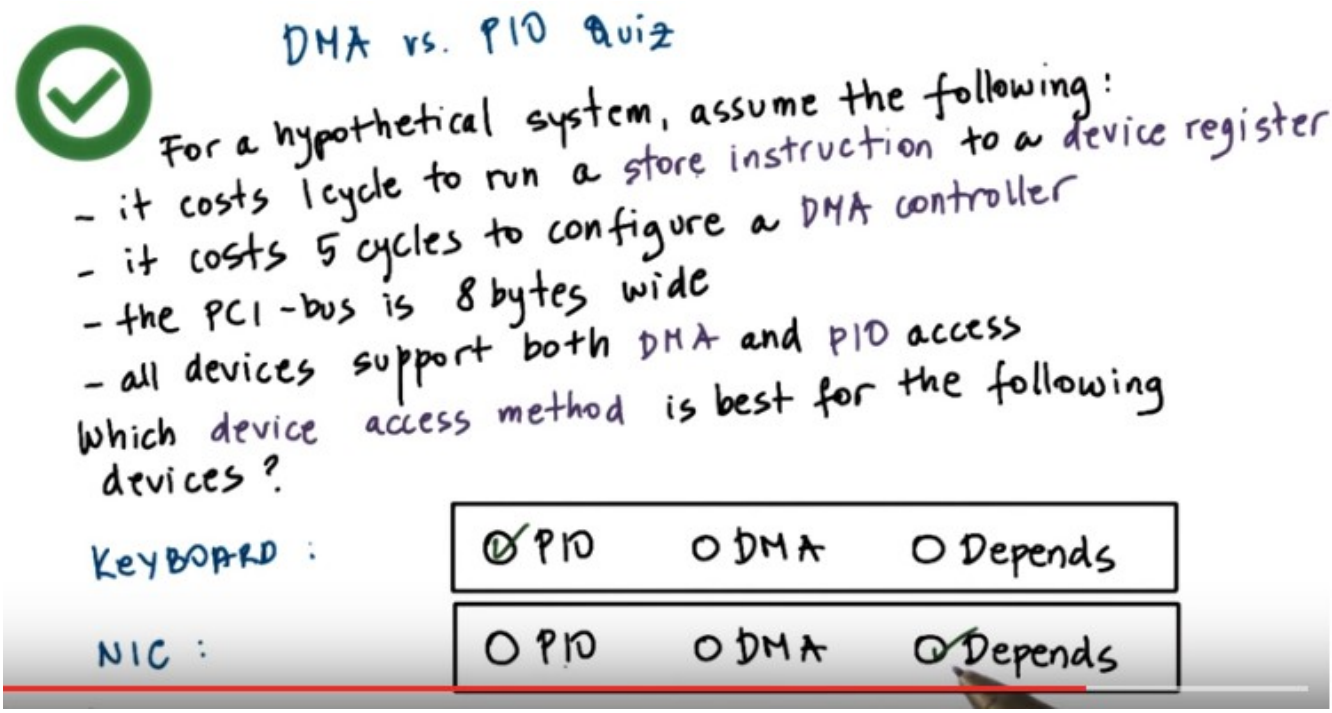


Let's look again at the network interface card, the NIC example, to illustrate how exactly DMA-based CPU-device interactions are carried out. Again, here, the data that needs to be moved from the CPU to the device is already formed as a network packet. And the very first step requires that the CPU write 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 buffer that's holding this packet that we want transmitted. So 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. All of this looks much better than the alternative of performing 189 store operations that we saw with programmed IO. The second step, configuring the DMA controller, is not a trivial operation, and it takes many more cycles then a memory store. Therefore, for smaller transfers, programmed IO will still be better than DMA, because the DMA step itself is more complex.

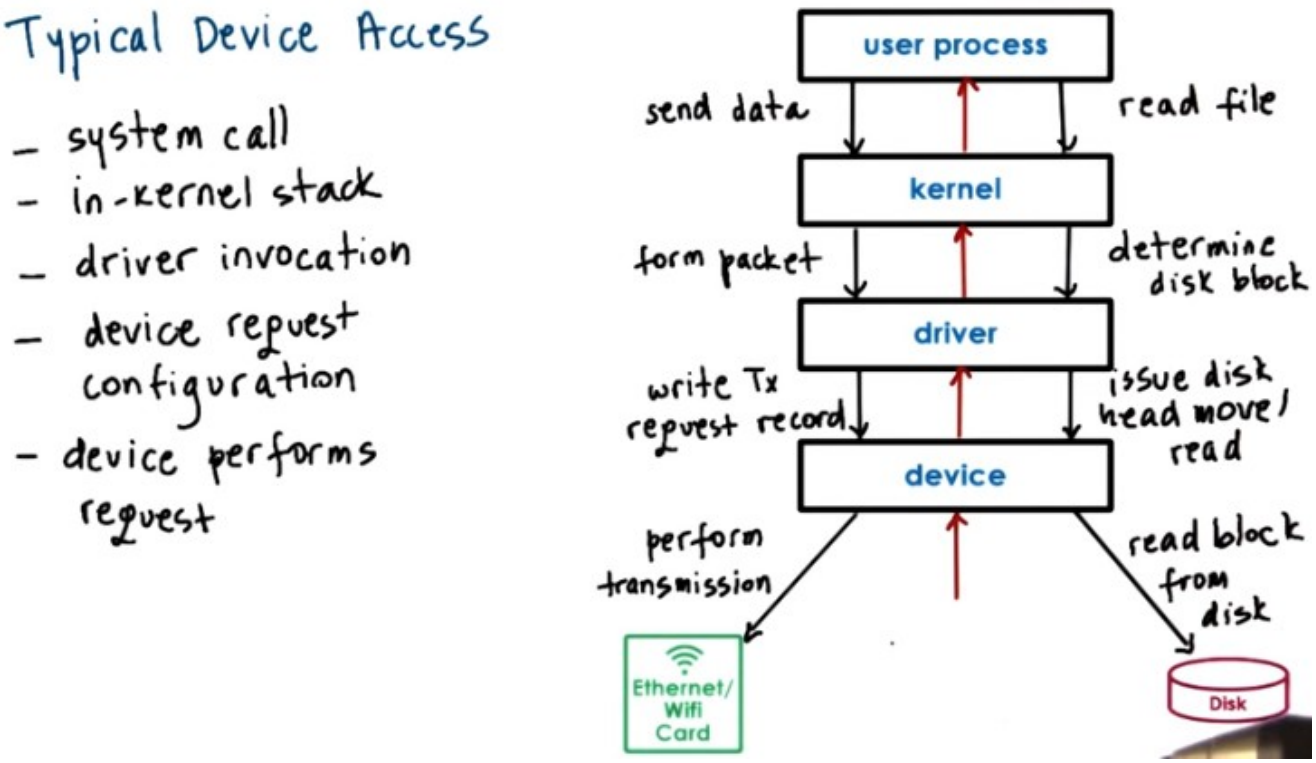


One important thing is that in order to make DMA work is, we have to make sure that the data buffer that we want moved or where we want data to be written must be present in physical memory until the transfer completes. It cannot be swapped out to disk, since this DMA controller can only read and write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

19. As a quiz, I would like for you to consider DMA and programmed IO in a hypothetical system. For a hypothetical system, assume the following, it costs 1 cycle to perform a store instruction to a device register, and it costs 5 cycles to configure a DMA controller. Also, the PCI-bus is 8 bytes wide. And also assume that all devices in the system support both DMA and programmed IO based operations. With this in mind, which device access method is best for the following devices? Keyboards or NICs? The options to consider are programmed IO, DMA, or depends. These are the three answers you need to provide for each of these devices.



20. The answer to each question will be 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 require that we perform a less than five store instructions to the device data registers, given that the difference between the store instruction and the DMA controller in this hypothetical example is one is to five. Then it's better to perform program PIO. If we need to perform larger data transfers, then the DMA option will be a better one since we just need to configure the DMA controller and then issue the request.



21. 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 (user 想傳東西出去) or to read a file from disk (user 想從 disk 中讀文件).

User process:
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, from the kernel.

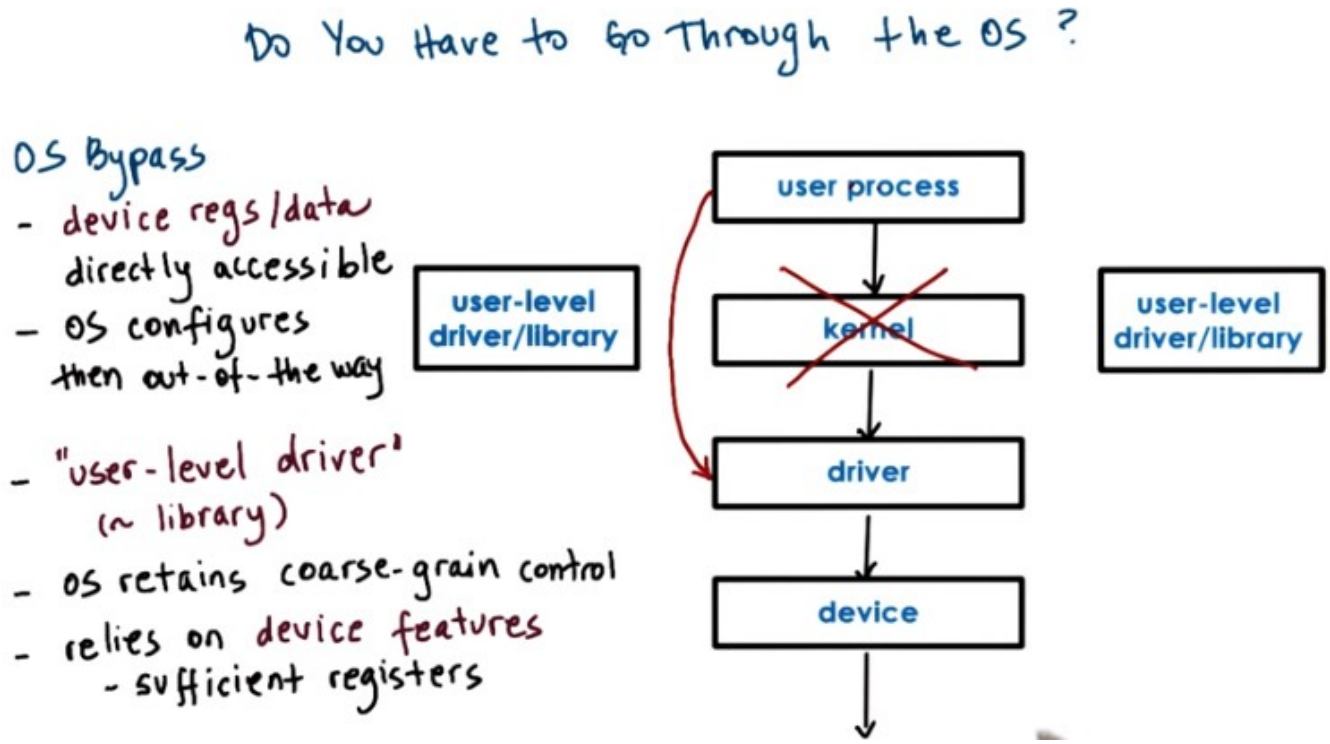
Kernel:
The operating system then will run the in-kernel stack related to this particular device. Like, for instance, the TCPIP stack to form the appropriate packet before the data is sent out to the network or the file system that's needed to determine the particular disk block that stores the file data. And the operating system will invoke the appropriate device driver for the network or for the block device for the disk,

Driver:
and then the device driver will actually perform the configuration (注意上圖左邊寫的 device request configuration, 不要看成了 device perform 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 pack of data. Or on the disk side, this means that the device driver will issue certain commands to the disk that configure the disk head movement or where the data should be read from, et cetera. The device drivers will issue these commands using the appropriate programmed IO or DMA operations, and this will be that in a device-specific manner. So that driver sort of wants that are understand the available memory registers on the device. They understand the other pending requests. So the driver is the one 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.

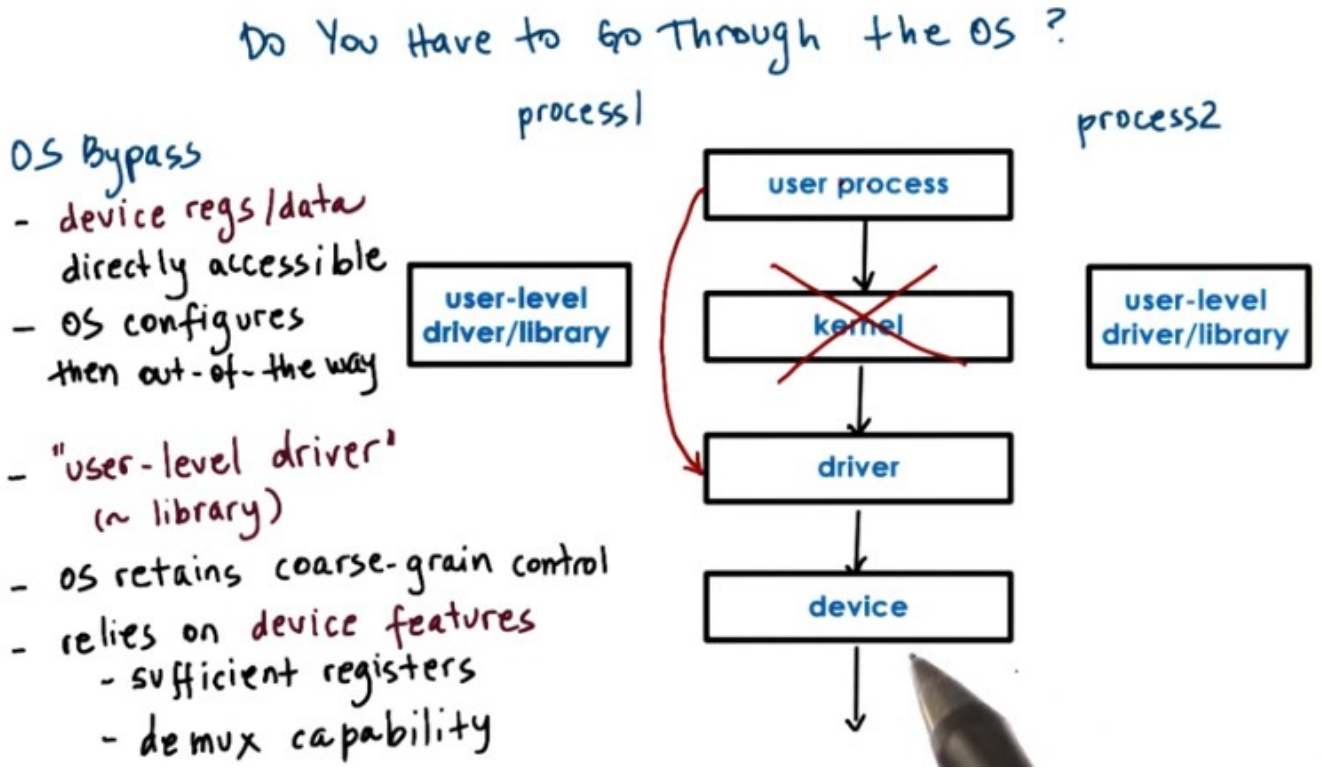
Device:
And finally, once the device is configured, it will perform the actual request. In the case of the network card, this device will perform the transmission of the data. In the case of the disk device, the device will perform the block read operation from disk.

Send response back:
And finally, any results from the request or in general any events that are originating on the devices will traverse this coal chain in a reverse manner.



22. It is not actually necessary to go through the kernel to get to a device. For some 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 register share 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's 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 level 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

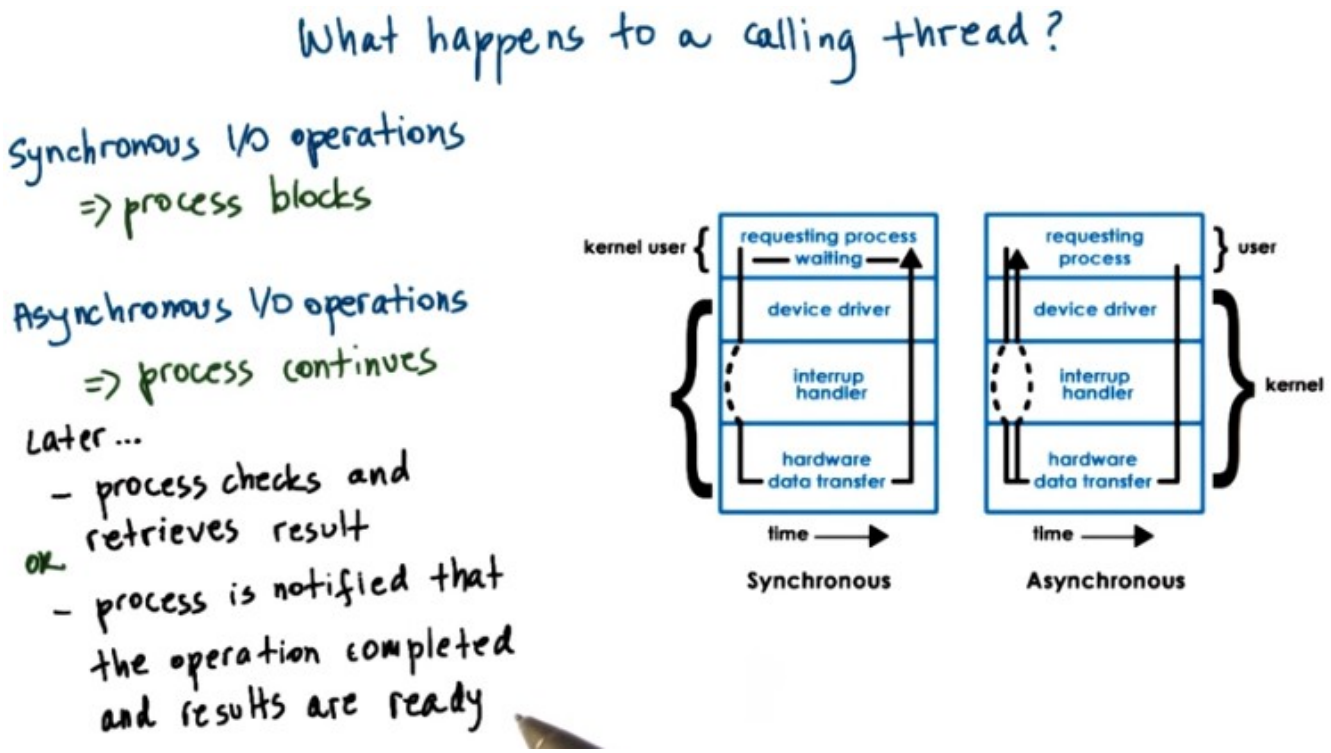
registers so that the operating system can map some registers to the user process. So the user process can perform the default device functionality like send/receive if it's a network device or read/write if it's a disk device. But still retain access to whatever registers are used for configuring and 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.



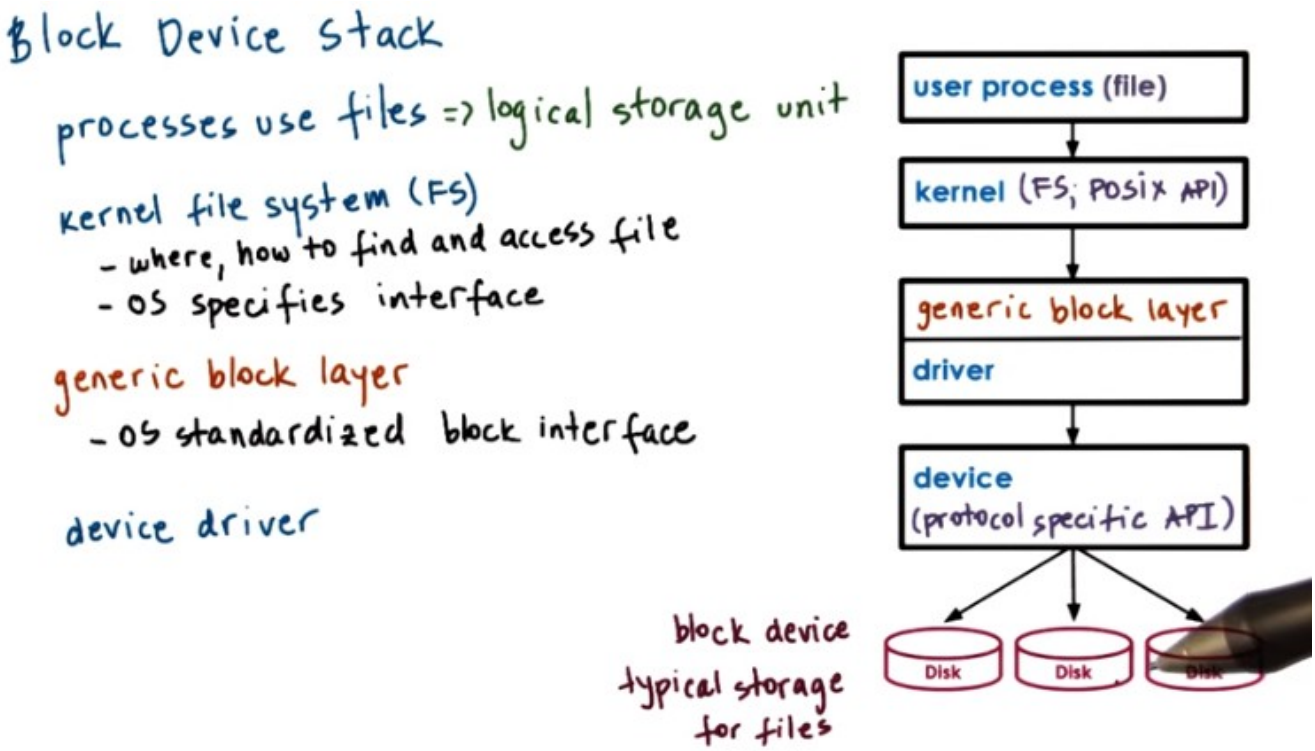
demux 即 demultiplex: 多路分配; 信號分離

Another thing that happens when multiple processes use the device at the same time is that when the device needs to pass some data to one of the processes. It is now the device that needs to be able to figure out how exactly to pass data to the address space of process one versus process two. 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 do. And then also it has to know which are the socket port numbers that these processes are using in their communication. 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 in this device can be correctly passed through the appropriate process. In the 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. 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 with 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 I/O 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 runable when the response from this request becomes available so in the mean time it will be blocked, it will be waiting, it will not be able to perform anything else. But 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 to see whether the results are available, this is somewhat analogous to the polling versus interrupt base interface that we talked about earlier. Remember that we talked about the asynchronous I operations when we talked about the flash paper in the lesson on thread performance consideration. In 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.

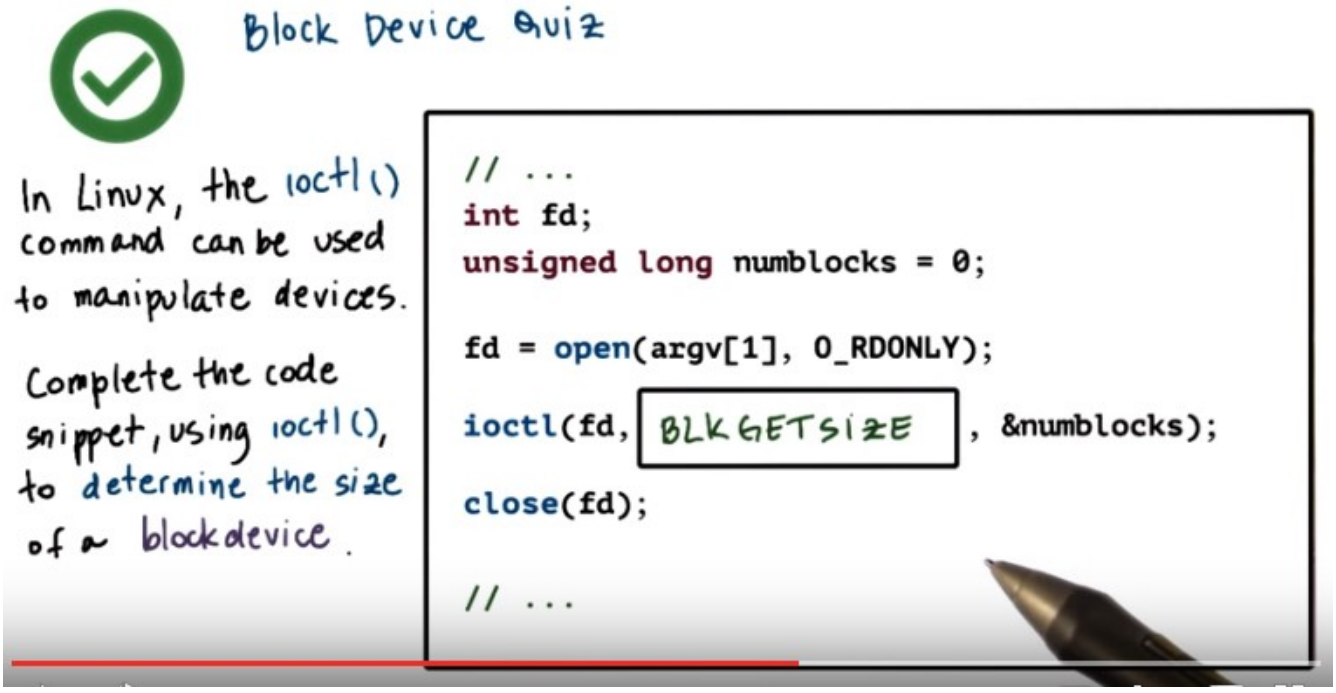


FS: file system

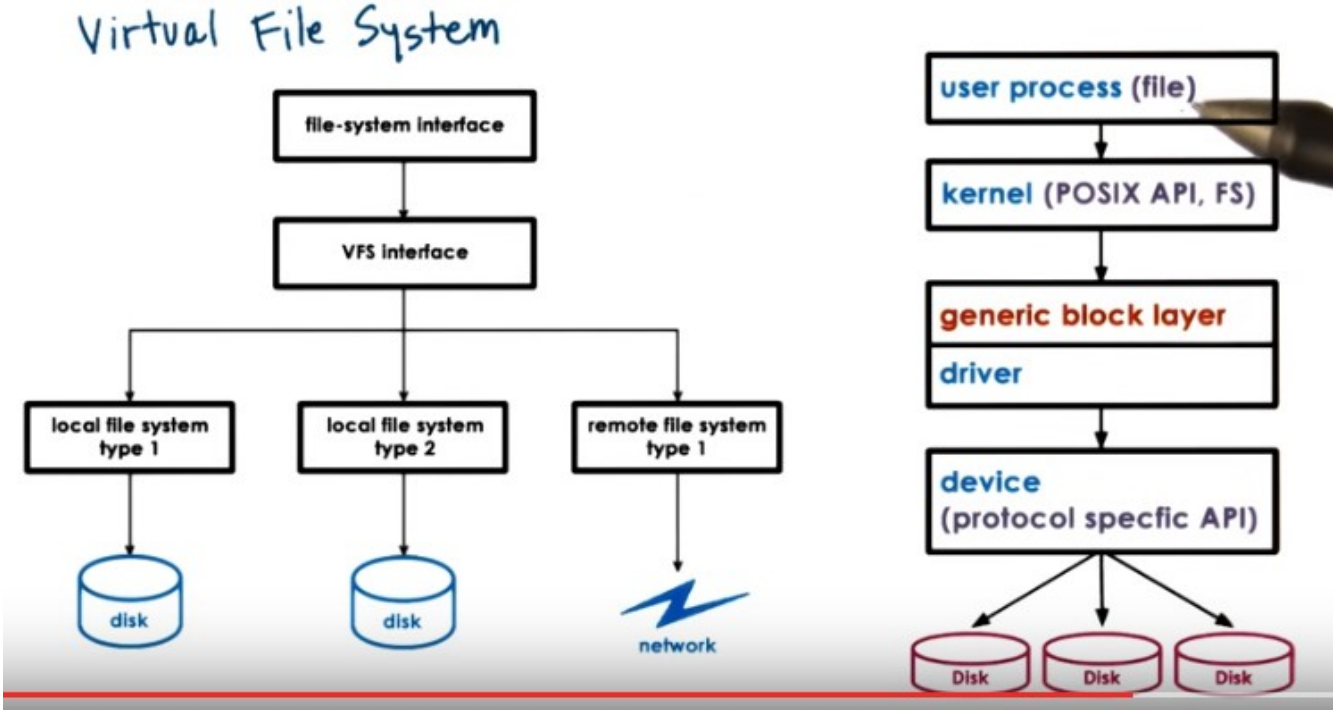
24. Let's look closer at how block devices (此處的 block 意思不是堵住, 而是块) are used, using the similar diagram. Block devices, like disks, are typically used for storage. And the typical storage-related abstraction, used by applications, is a file. The file is a logical storage unit, and it is mapped to some underlying physical storage location. What that means is that at the top-most 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 provide some flexibility in the actual details that a file system has in terms of how it lays out files in 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 the 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 would be certain standard APIs in terms of how these file systems interact with the underlying storage devices, as well as with other operating system components they 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 near their device drivers. We can have different types of block devices where the files could be stored, SCSI ID disk, hard disk, USB drive, solid state disks. And the actual interaction with them will require certain protocol specific APIs. 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 (difference), the block device stack introduces another layer, and that is the generic block layer. The intent of this layer is to provide a standard for a particular operating system to all types of block devices. The full device features are still available and unaccessible 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 in the file, it invokes that read/write operation for 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 a 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 that driver. Any lower-level differences among the devices, like the protocols that they use, et cetera, will be handled by the driver. It will speak the specific protocol that's necessary for the particular device.

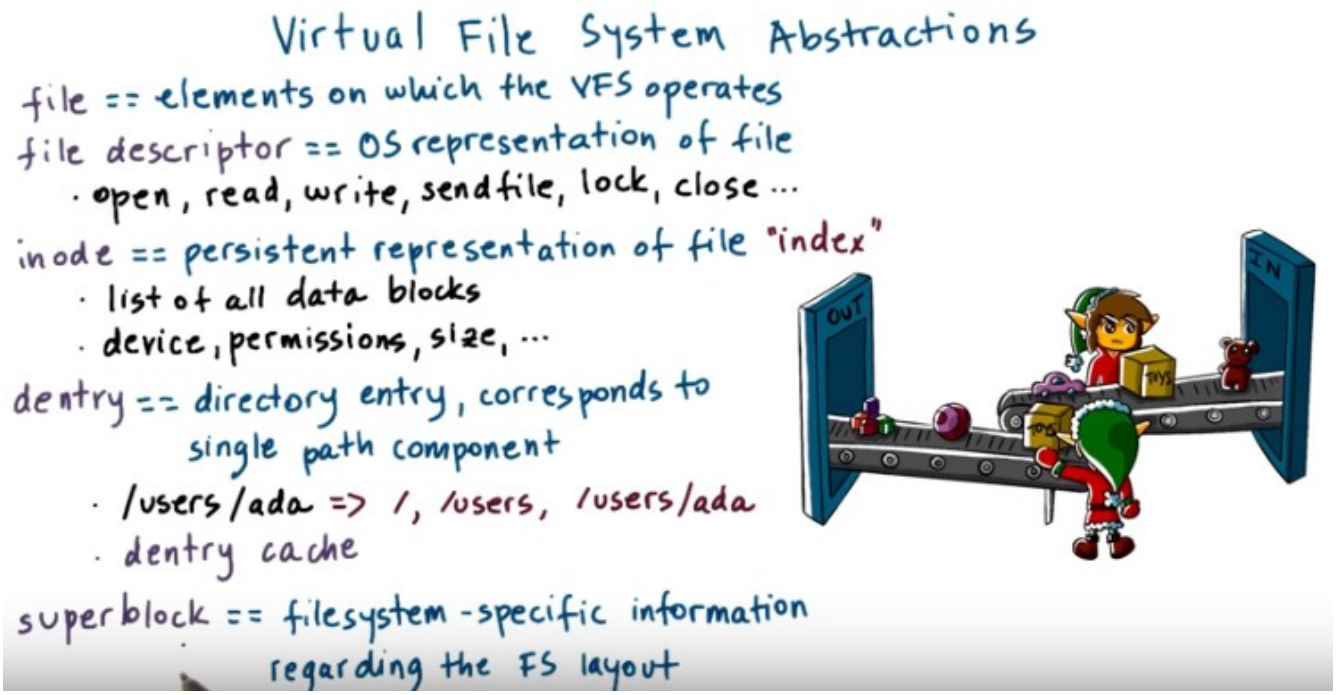
25. As we mentioned in a previous morsel, system software can access devices directly. And in Linux, the command ioctl is used to directly access and manipulate devices. It stands for io control, and this is essentially a way that the operating system can access the device's control registers. For this quiz you need to do the following. Again, in Linux the ioctl command can be used to manipulate devices. What you need to do is complete the code snippet using ioctl so as the determine the size of a block device.



26. The correct argument for the ioctl operation will be BLKGETSIZE. B-L-K GETSIZE. And this is specified in the Linux fs.h header file. When this function is executed the memory location that's pointed by the numblocks variable will be filled out with the returned value (即返回值是存到 numblocks 中的).



27. Here is the block device stack figure from the previous morsel, 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 they reference remote servers and use remote storage. Applications continue to interact with this virtual filesystem using the same type of API, for instance the POSIX API, and the virtual file system will specify a more detailed set of all system related abstractions that every single one of the underlined file systems must implement, so that it can perform the necessary translations.



28. The virtual file system supports several key abstractions.

First, there's obviously the file abstraction. These are the elements on which the virtual operating system operates.

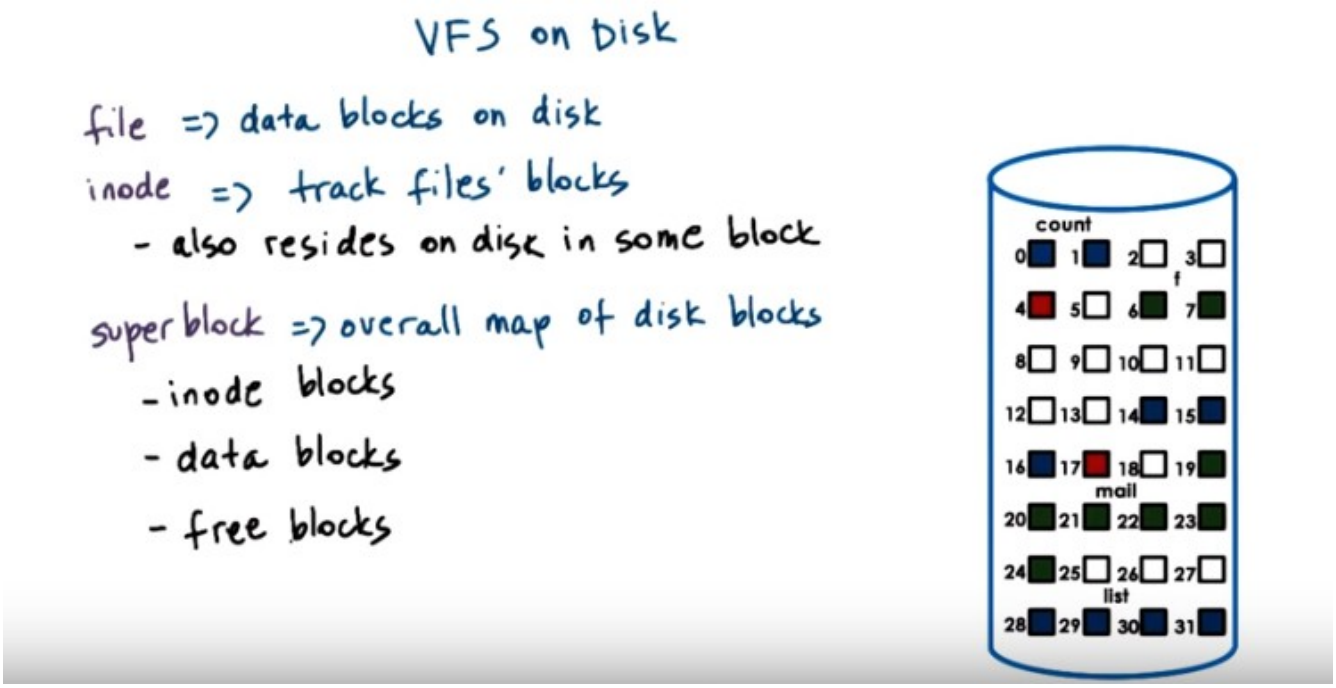
The OS represents files via file descriptors. A file descriptor is created when the file is first opened, and there are a number of operation that can be supported on files by using the file descriptor to identify the specific file. These include read, write, lock a file, send file, also close a file, ultimately.

For each file VFS maintains a persistent data structure called an inode. One of the information that's maintaining those 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, et cetera. 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. 下一段(第 29 段)會細講 inode 和 superblock, 看了那段後, 就很清楚甚麼是 inode 和 superblock 了.

Now, we know that files are organized in directories. But, from the virtual file systems perspective and in general from unique space systems perspective, a directory is really just a file, except, its contents include information about files and their inodes. So that we can find where the data blocks for these files and 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, stands for directory entry. And each dentry object corresponds to a single path component that's being traversed as we are trying to reach a particular file. For instance, if we're trying to reach a file that's in my directory, in 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 (即 dentry 可以是 /,

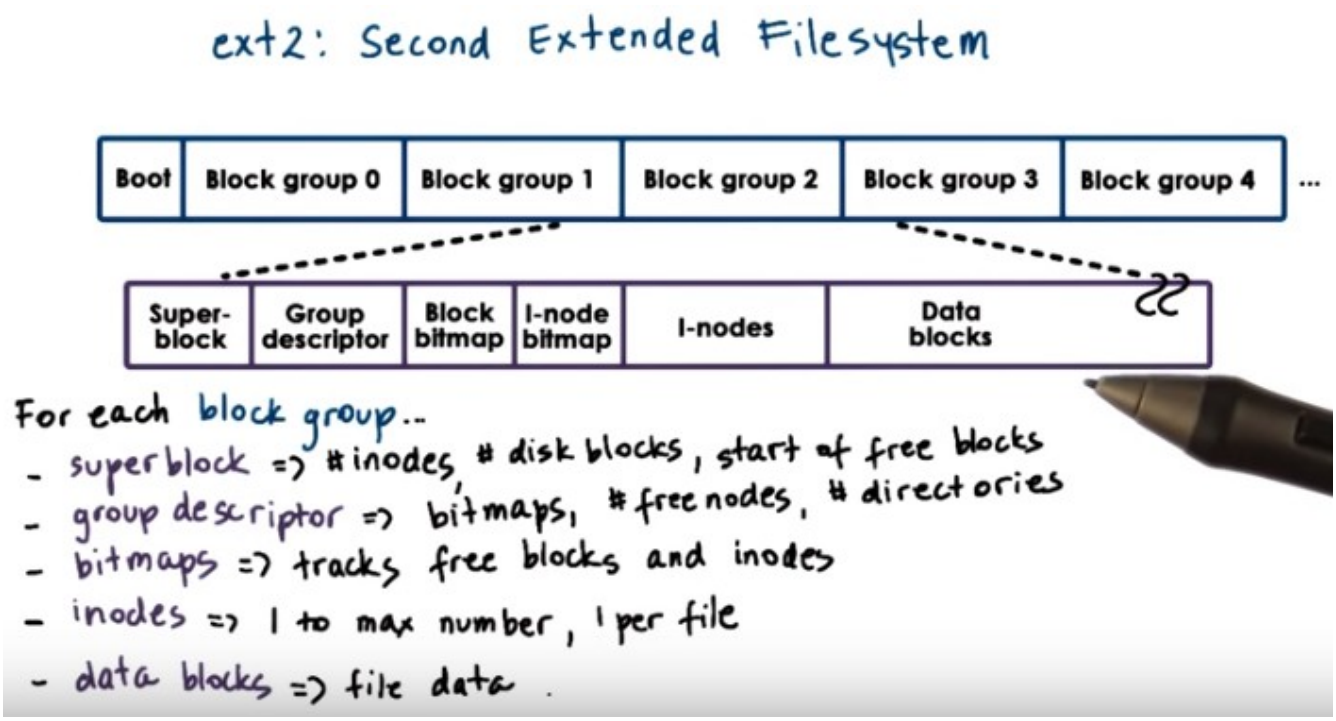
/users, /users/ada 這些東西). Forward slash for /users. That's the second directory in this path. And then finally, for the third directory in this path, /users/ada. Now this first slash, that corresponds to the root directory in the file system structure. The reason that this is useful is that when we need 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 (意思是: 若/代表的是/home/pt/wokao/blabla, 則不用每次都走一遍/home/pt/wokao/blabla 這個路徑, 而是由/就知道是當前文件夾) 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 are searching. So the file system will maintain a cache of all of 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 has it organized on disk the various persistent data elements, like the inodes or the data blocks that belong to the print files. Each file system also maintains some additional metadata in the superblock structure that helps to 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.



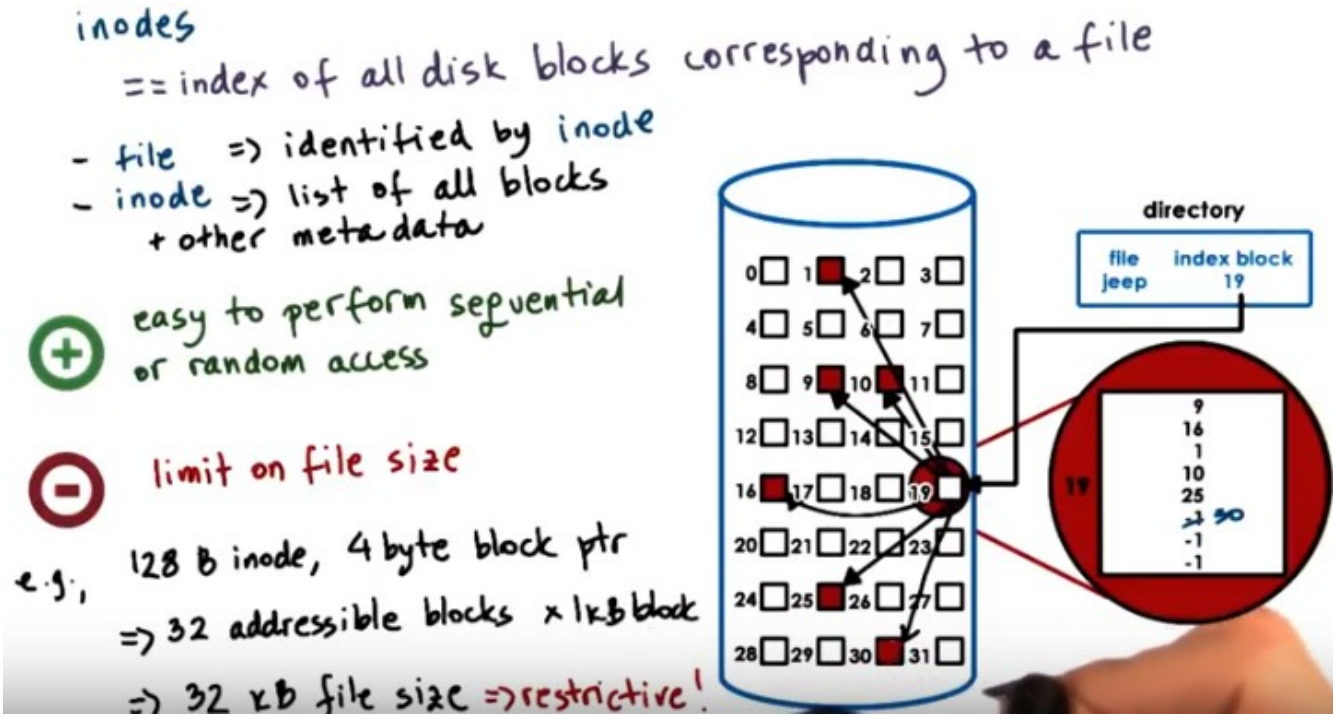
29. The virtual file system data structures are software entities. And they're created and maintained by the operating system file system component. But other than the dentries, the remaining components will actually correspond to blocks that are present on disk. The files, of course, are written out to disk, and they will occupy some blocks. And here we have two files, the green file and the blue file (紅的是甚麼? 下面會說). 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 in disk (即 inodes 也是保存在 disk 中的, 這就是為甚麼上圖中會有 inode blocks 這一項). So let's say these two blocks (那兩個紅的) here correspond to the inodes for these two different files (green file & blue file), 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 file write request. And it is also used for lookup when we need to find the particular portion of a particular file.



30. 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 Filesystem 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 an 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 disk: what are the disk cylinders or sectors or anything like that. Each of the Block Groups in this partition will be organized as follows. The first Block in a Block Group is the superlock. 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. About the total number of directories in the system. This information is useful when files are being allocated because the ext 2 tries to balance the over all allocation of directories and files across the determined Block Groups. The bitmaps are used to quickly find a free Block or a free inodes. And 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 the Data Block are free or they're used by some other file or directory. Then come the inodes. They're numbered from one up to some maximum number, and every one of the inodes is in ext2 128 byte long data structure that describes **exactly one 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 like 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

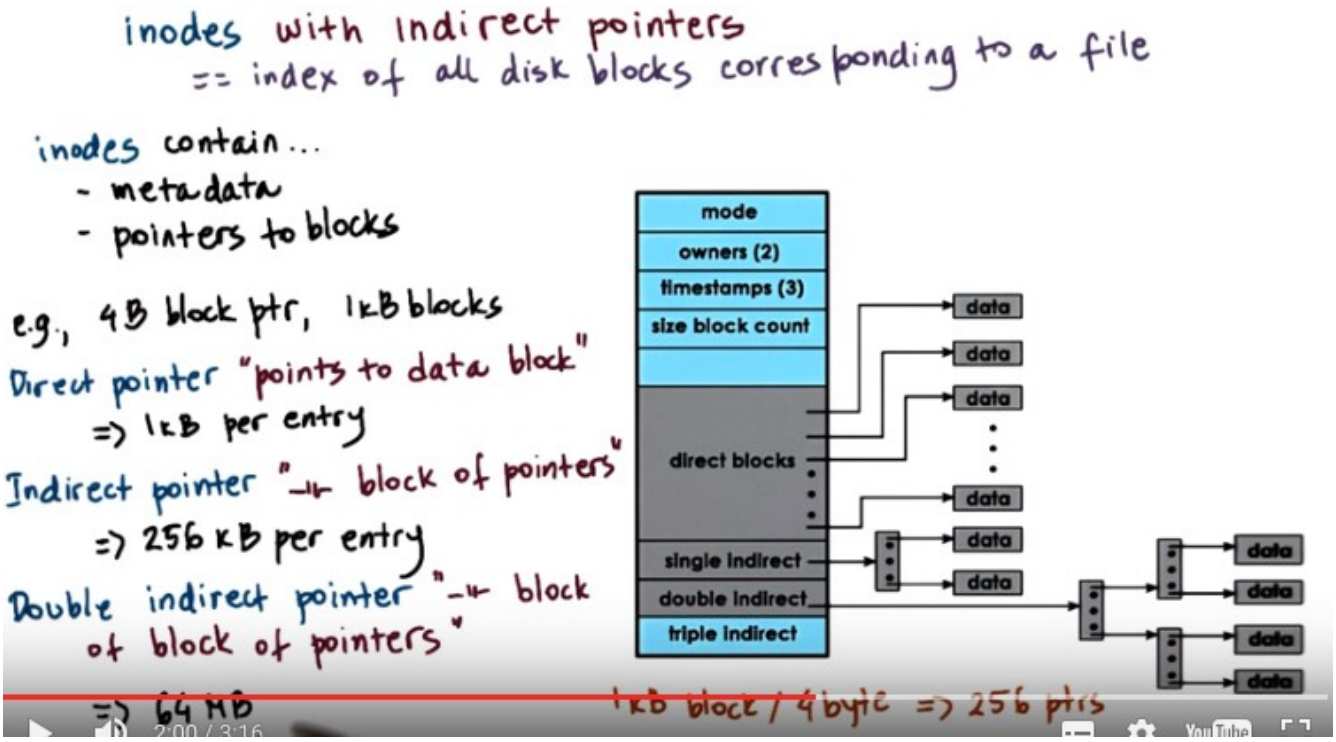
the upper levels of the file system software stack there will be these entry data structures created for each particular component for the particular directory.



注意最右邊手上的那個大紅圈的號碼是 19.
上圖左下角的 4 byte block ptr 意田是指 inode 中一個 index (如 9)的佔的空間為 4 byte

31. 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 a 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 this is useful to keep track of whether or not certain certain file accesses are legal, or to correctly updated the status of the file if it's locked or not locked. Or other things. One simple way in which this can be used this as follows. A file name (右上方框中的 jeep) is mapped to a single inode (右上方框中的 19) . And an inode, let's say here that 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 you look here we see that this file has five blocks allocated to it, and if we turn aside that we need more storage for these files, as per, pending per writing more data into the file. The file system will allocate a free block, lets 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 benefit of this approach is that it's easy to perform both sequential or random accesses to the file. That 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 number 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 this. 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 kilobyte, 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 kilobytes. That clearly is very restrictive.

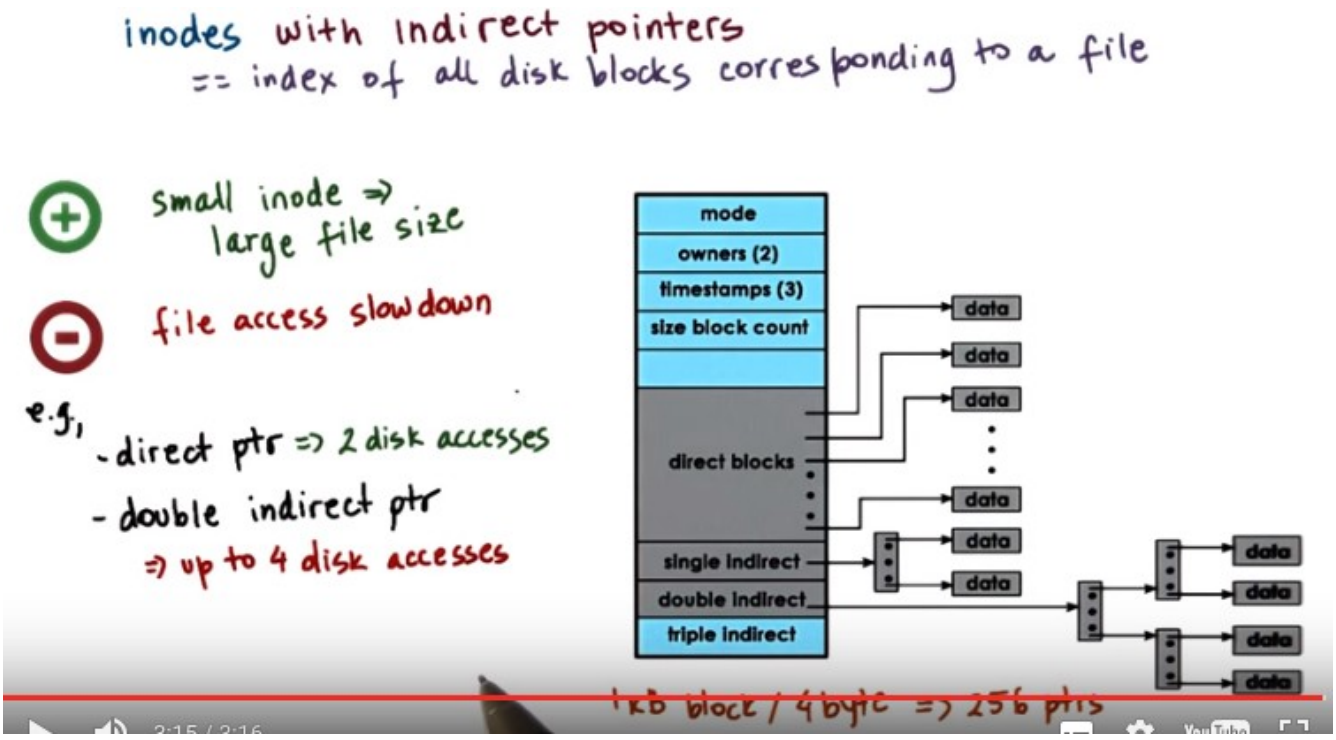


上圖中的那個大豎框即一個 inode.

256 kB per entry 是因為: 圖中 single indirect 所指的那個 block 大小為 1 kB, 而此 block 中放的全是 pointers, 每個 pointer 大小為 4 B, 故此 block 可放 1 kB / 4 B = 256 個 pointer, 而每個 pointer 又可以指向一個 1 kB 的 block, 故總共指向 256 kB 的空間.
64 MB 是因為: 256 * 256 * 1 kB = 65538 kB ≈ 64 MB.

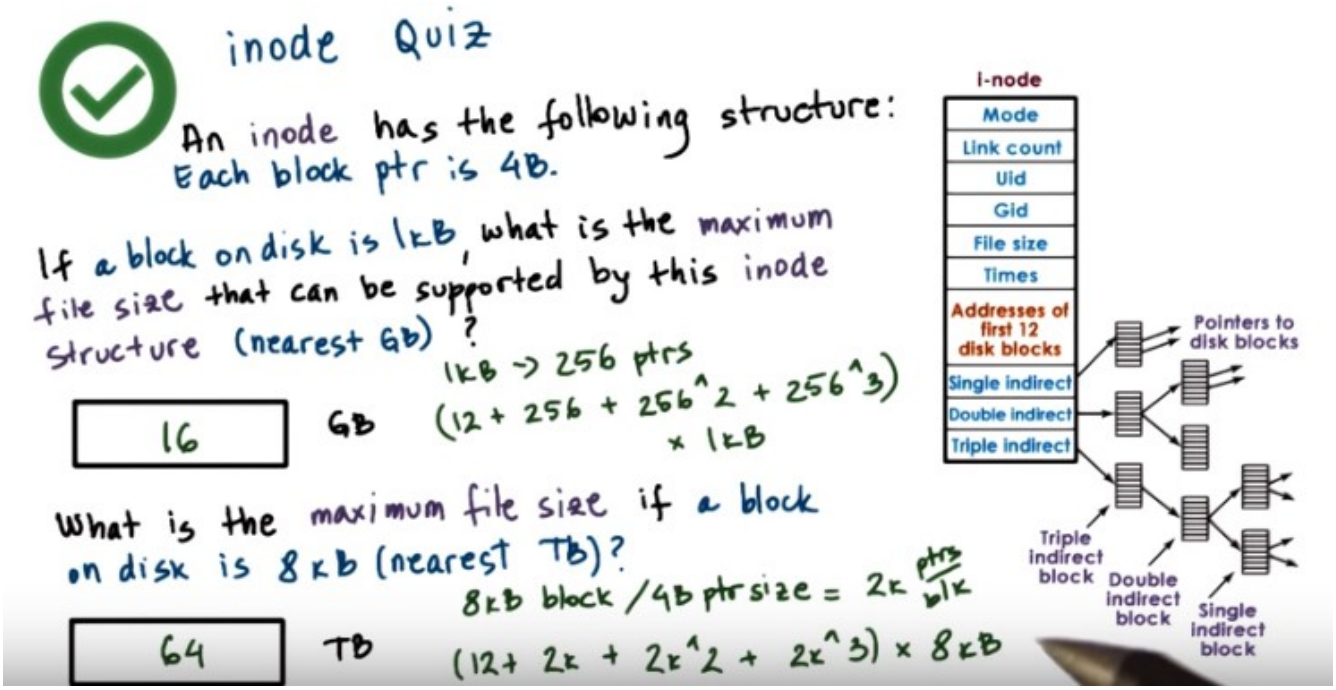
32. One way to solve this is to use so called indirect pointers. Here is an inode structure that uses these indirect pointers. Just like before, the inode contains metadata and pointers to blocks of data. The inode is organized in a way that first, it has all of the metadata, owner, when is the file last accessed. 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 1 kilobyte large, and we used 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 keeping the size of the inode small, we use so-called indirect pointers. And 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 1 kilobyte large and our pointers are 4 byte large, that means that a single indirect pointer can point to 256 kilobytes 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 times 256 times 1 kilobyte blocks for a total of 64 megabytes 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 pointer 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 we will access the data block. So we mean the performing 2 disk accesses per file access and that's at most. With the double indirect pointers, the situation is very different. We need one 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. And inode has the following structure and every single one of the block pointers, both the direct and indirect ones, is 4 bytes long. If in the system a block on disk is 1 kilobyte, what is the maximum file size that can be supported with this inode structure? You should write your answer, rounded up to the nearest gigabyte. Also answer, what is the maximum file size if a block on disk is now 8 kilobytes. In this case, provide the answer to the nearest terabyte.



Instructor Notes, Quiz Help

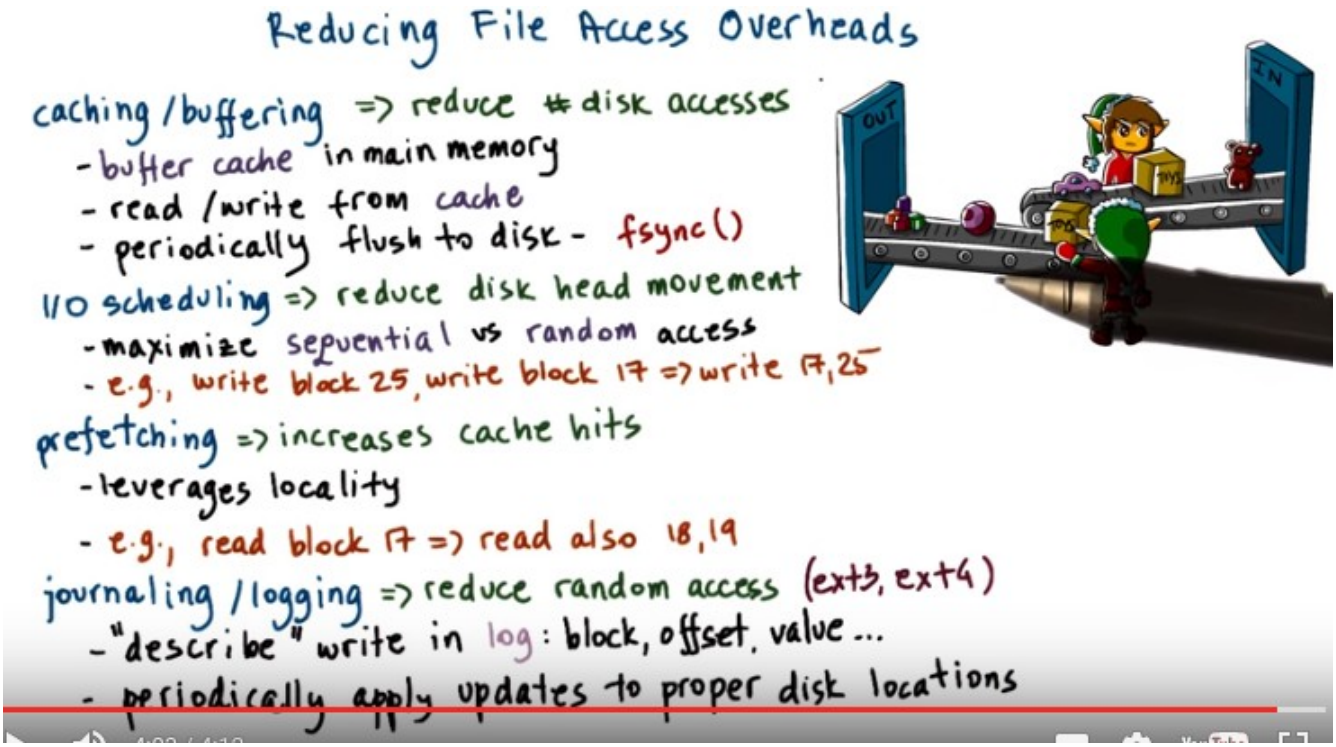
Maximum File Size:

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

(12 + 256 + (256 * 256) + (256 * 256 * 256)) * 1 kB ≈ 16 GB.
8 * 256 = 2048
(12 + 2048 + (2048 * 2048) + (2048 * 2048 * 2048)) * 8 kB ≈ 64 GB.

34. To answer this question we will need to add up the sizes that can be addressed with every single type of the different pointers that are included in the inode data structure. This includes the 12th direct disk block pointers, and then the one single indirect pointer, the one double indirect pointer, and the one triple indirect pointer. The answer to the first question is 16 gigabytes. Remember, we have 1 kilobyte blocks, and every single block pointer is 4 bytes. So, with a single block, again, we address 256 pointers. Now, if we think about what is the total number of file blocks that are addressed, their direct pointers will address 12 file blocks. Then the single indirect will address another 256 of those. The double indirect will address 256 square. Triple indirect 256 cube. And all of that needs to be multiplied by the actual size of the blocks. So that's 1 kilobyte. That will produce a maximum file size of 16 gigabytes. The answer to the second question is 64 terabytes. Remember here we have 8 kilobyte block sizes, and given that the pointer size again is 4 bytes, each one of the blocks can contain 2k pointers. Now if we do the math to compute what is the total number of blocks that can be addressed with this inode, we will come up with this calculation and multiply that by 8 kilobytes again, that is the block size will produce the answer of 64 terabytes. So, although we've only increased the

block size from 1 kilobyte to 8 kilobyte, because of this non-linear data structure that's used to address the blocks, we're able to achieve much larger file sizes that can be supported by these classes.



35. File systems use several techniques to try to minimize the access to disk and to improve the file access overheads.

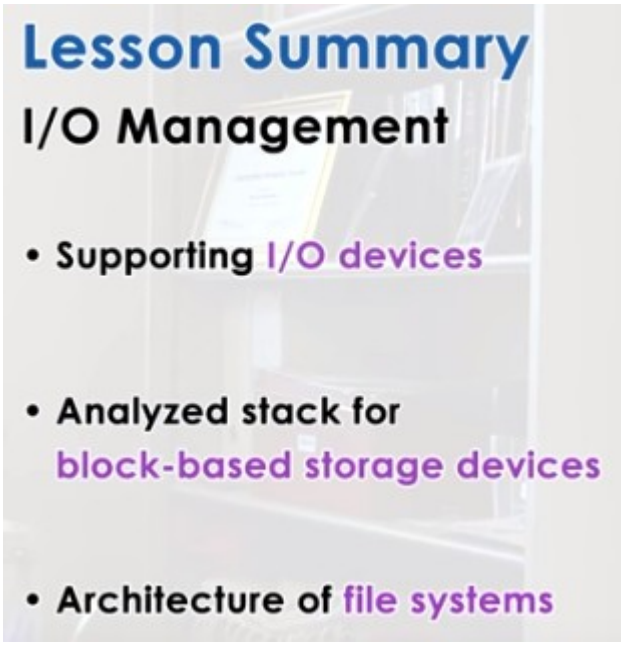
Much like hardware caches are used to temporarily hold recently used data and avoid accessing main memory, file systems rely on buffer caches, except these buffer caches are present in main memory and they're used in order to reduce the number of necessary disk accesses. With caching the file will be 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 disc, by forcing these flushes to happen only periodically, we can amortize the cost of performing a disc access over multiple intermittent rights 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 diskette movement that's a slow operation and we can do that by maximizing the number of sequential accesses, which are needed, so, for sequential access, this kind of movement is not expensive It's expensive for these random accesses, 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, let's say 15, these operations will be reordered by the I/O scheduler so that they're issues 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 blocks will be accessed as well. File systems can take advantage of this feature by prefetching more than one block of a file whenever a single block is accessed, this does use up more disk bandwidth to move larger, in this case,

three x worth of data from disk into main memory. But it can significantly impact, or reduce, the access latency, by increasing the cache hit rate because more of the accesses will be served out of cache, potentially.

Finally, another useful technique is journaling (上圖中最後一行對理解很有幫助). 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 that 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 in 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 overall, the nature of the journal link or the logging base systems. The following file systems to execute called the ext3 and the ext4 they're also part of current Linux versions, 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 four techniques and 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 current file system solutions.



36. 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.

37. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.