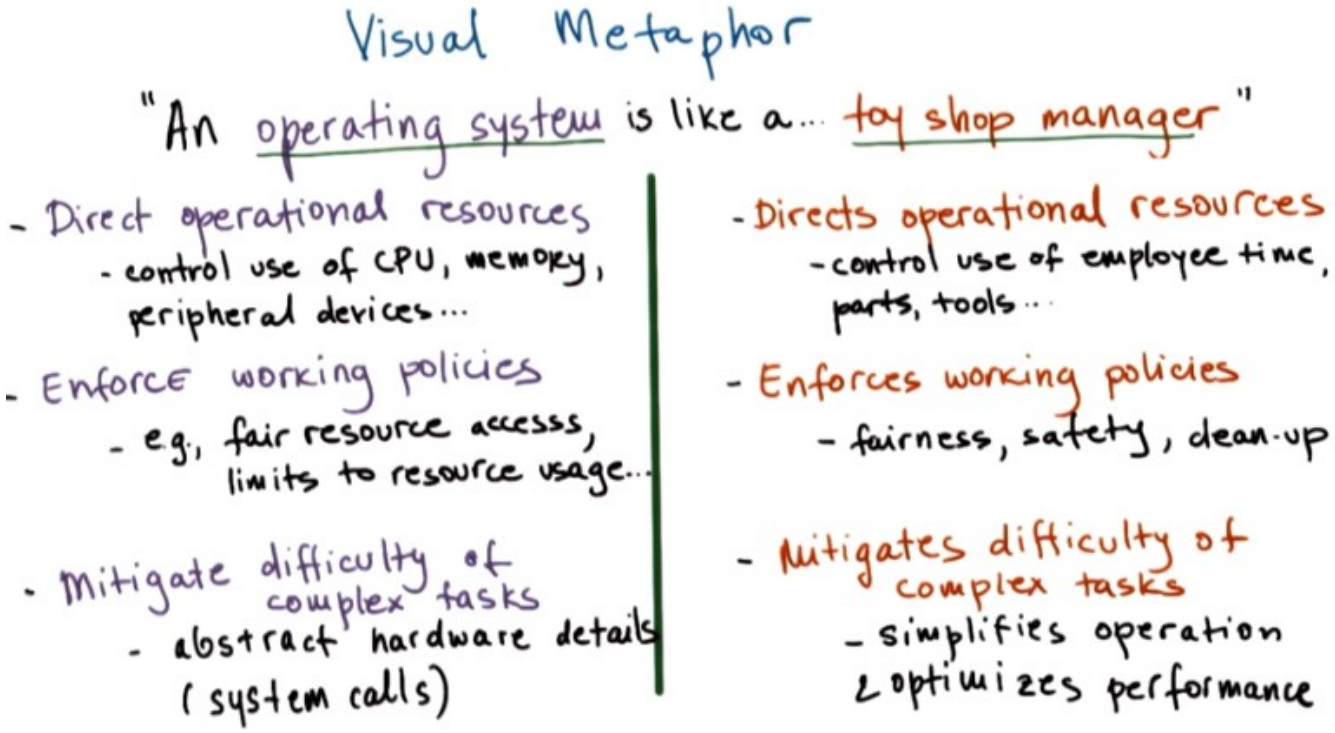
Communication between applications and OS via system calls: 第 23 段

arbitrate: 仲裁

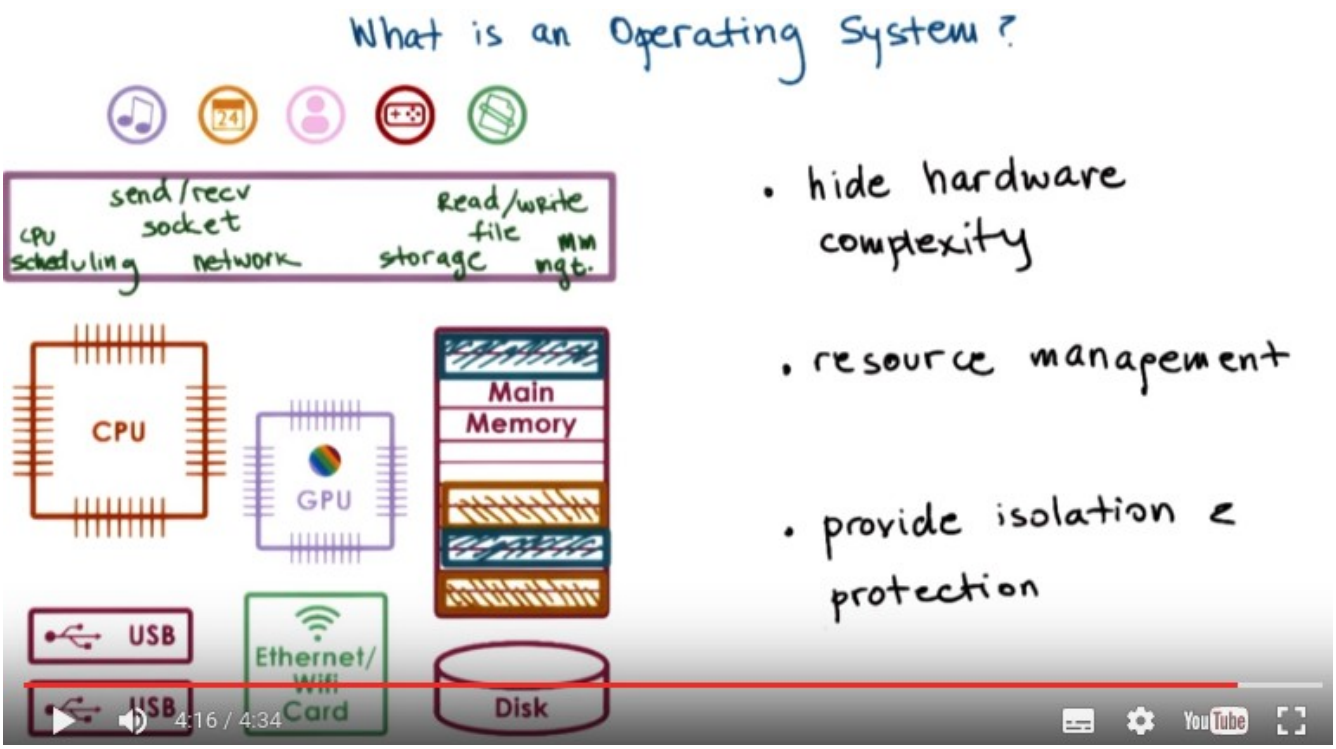
1. In this introductory lecture, we will provide a very high-level view of operating systems. We will explain what is an operating system and what is the role that it plays in computer systems. Then we'll look at some of the key components of operating systems, design principles, and organizational structures. The following lectures in this class will provide you with a much more in-depth understanding of what any of the terms and mechanisms that come up during this lesson actually are. In simplest terms, an operating system is a piece of software that abstracts and arbitrates the underlying hardware system. And in this context, abstract means to, to simplify what the hardware actually looks like. And arbitrate, it means to manage, to oversee, to control the hardware use. We will see throughout this course that operating system support a number of abstractions and arbitration mechanisms for the various types of hardware components in computer systems.



mitigate: (減輕).
由以上可知 system call 之作用

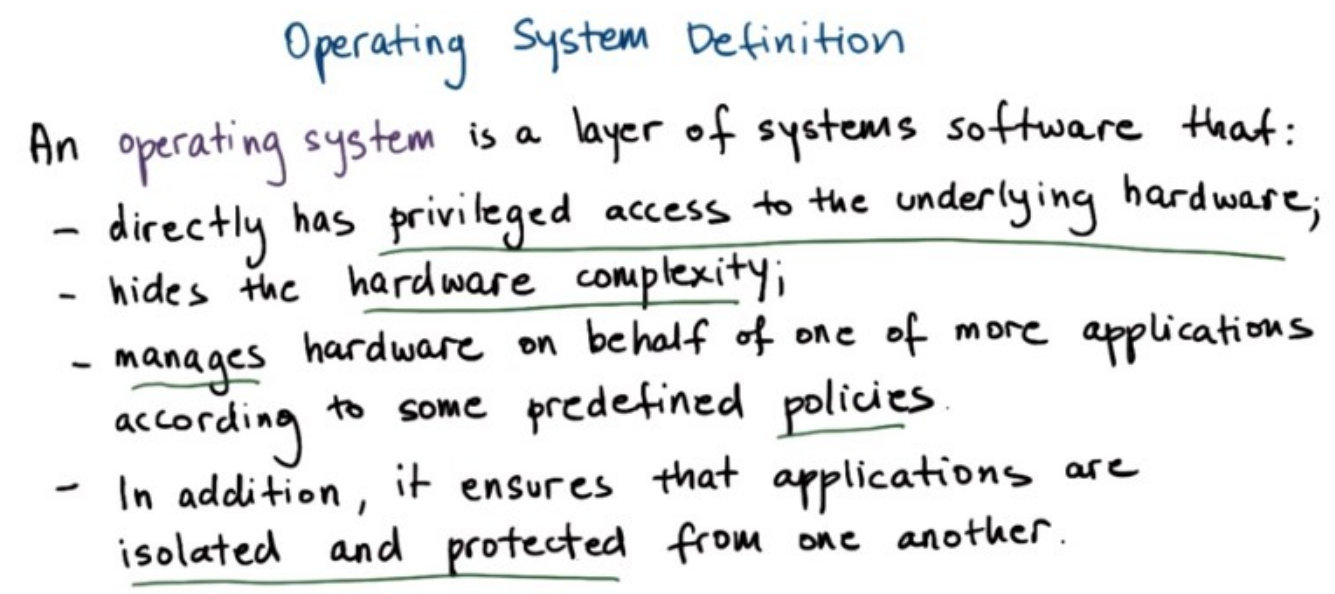
2. While this entire course will cover many elements of an operating system, it would be nice to have a simple illustration of what an operating system is like. So, to give a visual metaphor using our toy theme, we can say that an operating system is like a toy shop manager. At a high level, how is a toy shop manager like an operating system? First, a toy shop manager directs operational resources in the shop. Second, a toy shop manager enforces working policies. And finally, a toy shop manager mitigates the difficulty of complex tasks. For instance, for any toy shop manager, they must be in charge of their employees and direct their working efforts towards completing the work. Additionally, a toy shop manager is responsible for scheduling, for allocating, for distributing the resources in the shop, the parts and the tools, and for determining which of the employees will work on which orders and use which tools. Toy shop managers are also the ones that enforce working policies in the toy shop. This can be policies regarding fairness, safety, clean-up. For instance, how workers working on two different orders have to share one of the shared resources, shared parts and tools for instance. Another example is policy regarding how to clean up after yourselves once a worker is done with processing a toy order. Finally, a toy shop manager helps mitigate the effects of complex tasks. It does that by trying to make the overall operation of the toy shop more optimized and more simple. Without a manger, for instance, employers may not know how to establish order among themselves, how to decide, how to handle the workload. So the toy shop manager has to deal with this situation so as to avoid dealing with a much more complex task. So, the question then is, what parallels are there between a toy shop manager with an operating system? Operating systems do direct operating

resources. In particular, they control the use of the hardware, CPU, memory, peripheral devices such as disk, network cards, etc. And they decide how these resources with be allocated to applications. Along those lines, operating systems actually enforce some policies. This can include polices regarding how these resources are used, for instance, to control fair access to the shared resources. Or it can be even to impose certain limits to allocate maximum amount of a certain resource that a particular application or process can use. Examples of such limits are, for instance, the number of open files that can be open per process, or some thresholds that need to be passed in order for some memory management daemons to kick in. There are numerous other examples of limiting resource usage. And finally, the operating system helps with the difficulty of complex tasks. And particularly important is that it simplifies the view of the underlying hardware that's observed by the applications that are running on that particular platform. Instead, applications interact via system calls with the operating system, and the operating system takes care of managing those difficult tasks for them.

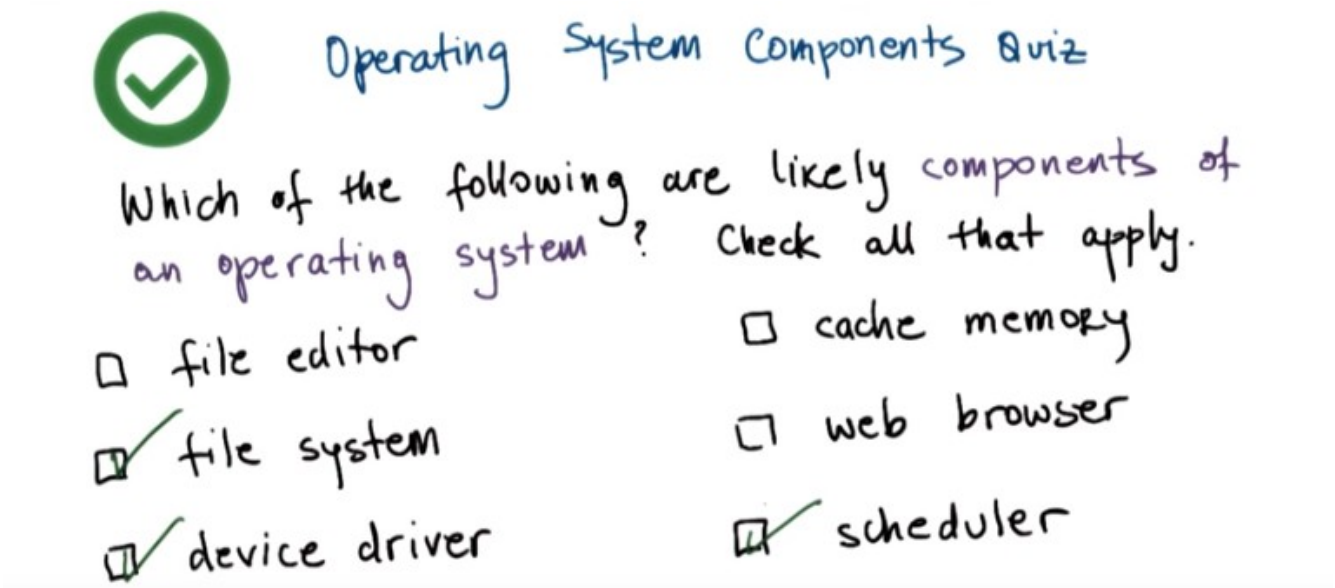


3. So, what is an operating system and what is the operating system's role in computing systems? Let's start taking a look at what a computing system looks like and how it's used. Computing systems consist of a number of hardware components. This includes one or more processing element. So, processors or CPUs. Today, the CPUs further consist of multiple cores. So, a single CPU will have more than one processing element. And there's also memory, and there are also devices like network interconnects for connecting to the Internet, like your Ethernet port or your Wifi card. Other components include graphics processing cards, like GPUs for instance. And also storage devices like hard disks, SSDs, flash devices like USB drives. Except for in some very specific environments like certain embedded platforms or sensors, all of these hardware components will be used by multiple applications. For instance, on your laptops or desktops, you may currently be running a browser, a text editor, perhaps Skype. You may have a number of other applications. In data centers, server machines also run also multiple applications, maybe a web server, a database, or some computation intensive simulation, many other options. An operating system is the layer of systems software that sits between

the complex hardware and all of these applications. There isn't one formal definition of what an operating system is. So instead, we will look at the role that the operating system serves and the functionality that it must provide. And in that way, we will try to build our understanding of what an operating system is. First, the operating system hides the complexities of the underlying hardware from both the applications, as well as the application developers. You don't want to worry about disk sectors or blocks when you're saving the output of a computation in your program. For instance, the operating system hides the complexities of the various types of storage devices, so hard disks, SSDs, USB flash. And it manages a higher level abstraction of file, and it provides some number of operations, like reads and writes, that you can perform on that file. Similarly, for a web server application, when you're accepting and responding to end user requests, you don't want to think about the bits and the packets that you need to compose when performing that communication. Instead, the operating system abstracts the networking resource. It provides a higher level abstraction that is typically called a socket. And then it provides some services to send and receive packets or data from that socket. Furthermore, the operating system not only hides the hardware complexity, but it actually also manages the underlying hardware on behalf of the executing applications. For instance, the operating system decides how many and which one of these resources will the application use. For instance, the operating system allocates memory for these applications, and also is the one that schedules these applications on to the CPU that can execute them. It also controls access of these applications to the various devices in the system. So overall, it's responsible for all types of resource allocations and resource management tasks on behalf of these applications. Finally, when multiple applications are co-running on the same hardware, the operating system must ensure that each of them can make adequate progress and that they don't hurt each other. So, it will provide isolation and protection. For instance, the operating system will allocate to different applications different parts of the underlying physical memory. And it will also make sure that unless explicitly intended, applications don't access each other's memory. This is important both from a protection perspective so they don't read each other data, and it's also important from an isolation perspective so that they don't end up overwriting each other's memory. Note that these types of mechanisms are important even in environments that were traditionally considered embedded platforms like mobile phones, where hardware was typically running just one application. Think about it. How many applications do you currently have running on your smartphone? On my phone, I have a browser, I have Skype, I have Facebook. And I still have that one key application, and that is the actual phone, making phone calls and receiving phone calls.

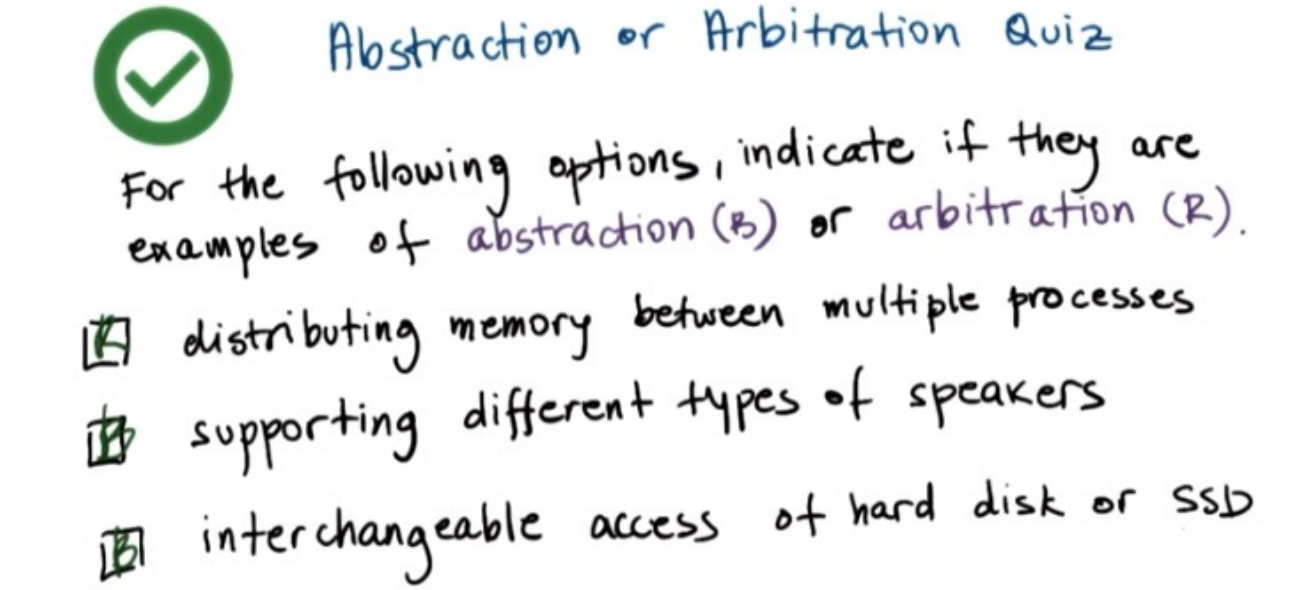


4. In summary, an operating system is a layer of systems software that resides between applications and hardware. It corresponds to the software that has privileges to access the underlying hardware and manipulates its state. In contrast, software that corresponds to the applications is not allowed to do that. Its role is to hide hardware complexity and to manage the hardware on behalf of one or more applications. And it has to do that according to some predefined policies. And finally, it has to ensure that applications are isolated and protected from one another.



5. Now that we have been given a definition, let's take a quiz to see what types of components do we expect an operating system to have. The question is, which of the following are likely components of an operating system? The options are a file editor, a file system, a device driver, cache memory, a web browser, and a scheduler. You should check all that apply.

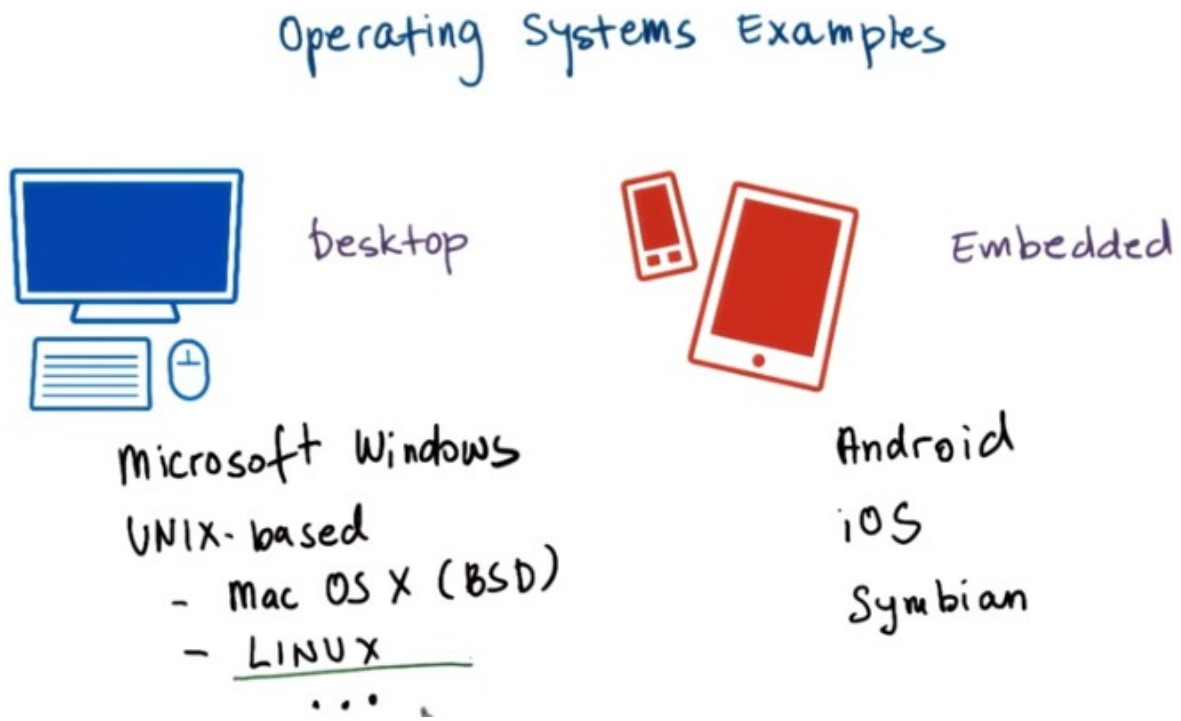
6. Starting from the top, a file editor is likely not a part of an operating system because the users interact with it directly, and it's not involved directly in managing hardware. Next, the file system is likely a part of an operating system. It's directly responsible for hiding hardware complexity and for exporting a simpler, more intuitive abstractions. A file, as opposed to block of disk storage. Device drivers are also likely part of an operating system. A device driver is directly responsible for making decisions regarding the usage of the hardware devices. Cache memory is a little bit tricky. Although the operating system and the application software utilize cache memory for performance, the OS doesn't directly manage the cache. It's really the hardware that manages it itself. Web browsers are also not part of an operating system. Again, just like in the file editor case, it's an application that users interact with and does not have direct control over underlying hardware. And finally, the scheduler. This is indeed a part of the operating system because it's responsible for distributing the access to the processing element, the CPU, among all of the applications that share that platform.



7. In the previous morsel, it was stated that an operating system abstracts and arbitrates the use of the computer system. For the following options, indicate if they're examples of an abstraction, where the operating system simplifies something about the underlying hardware, or arbitration, where the operating system manages the underlying hardware. Here are the options. Distributing memory between multiple processes. Supporting different types of speakers. Or, providing interchangeable access of hard disk or SSD.

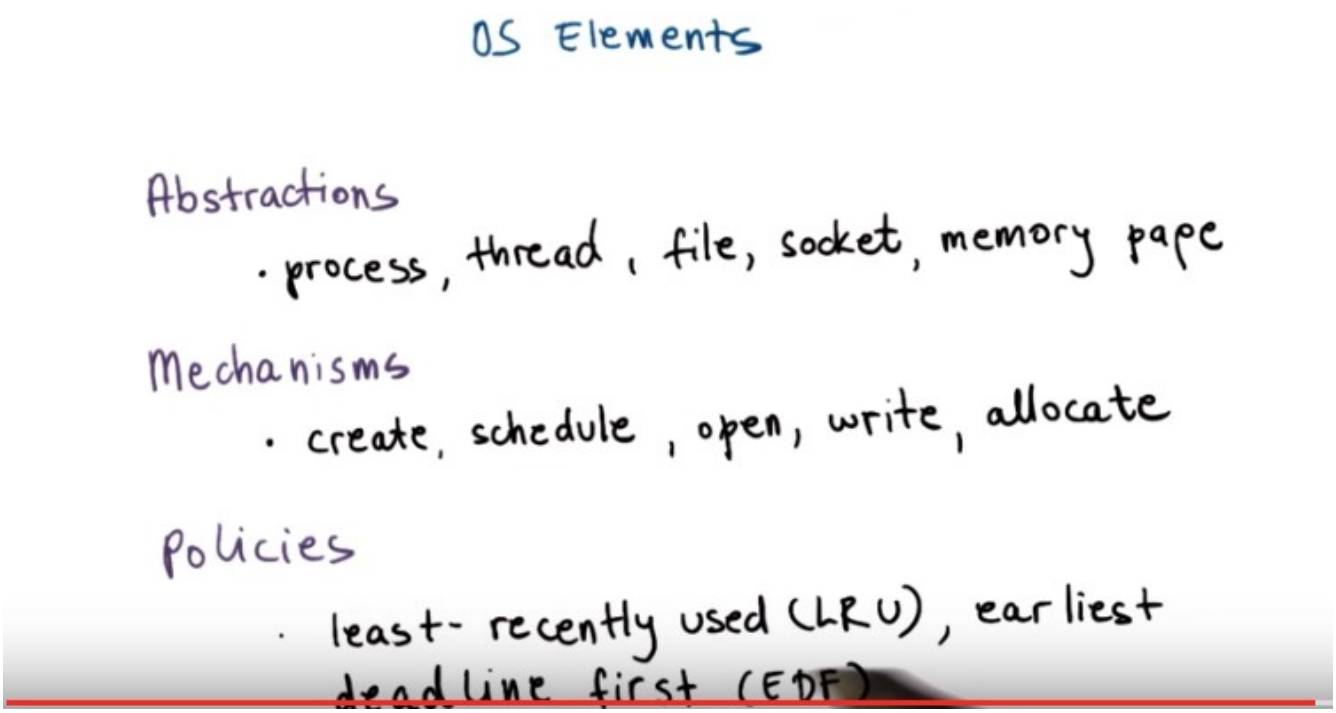
8. For the first option, distributing memory between processes, that's an arbitration. This is something an operating system does as a result of its effort to manage the memory and determine how multiple processes will share. The second option, supporting different types of speakers, that's an abstraction. It is because the operating system provides abstractions such as this one that you can plug in one set of speakers, and if they don't work, exchange them with something else. In some cases, drivers are required, which enables an operating system to control the hardware device without knowing details about that specific hardware, so the device driver will have the knowledge of the specific actual

hardware element, like the specific speaker. And along similar lines, the ability to interchangeably access different types of storage devices like hard disks or SSDs is again an example of an abstraction just like the example above. Again, because of the use of the storage abstraction that operating systems support, they can underneath feel the different types of devices and hide that from the applications.

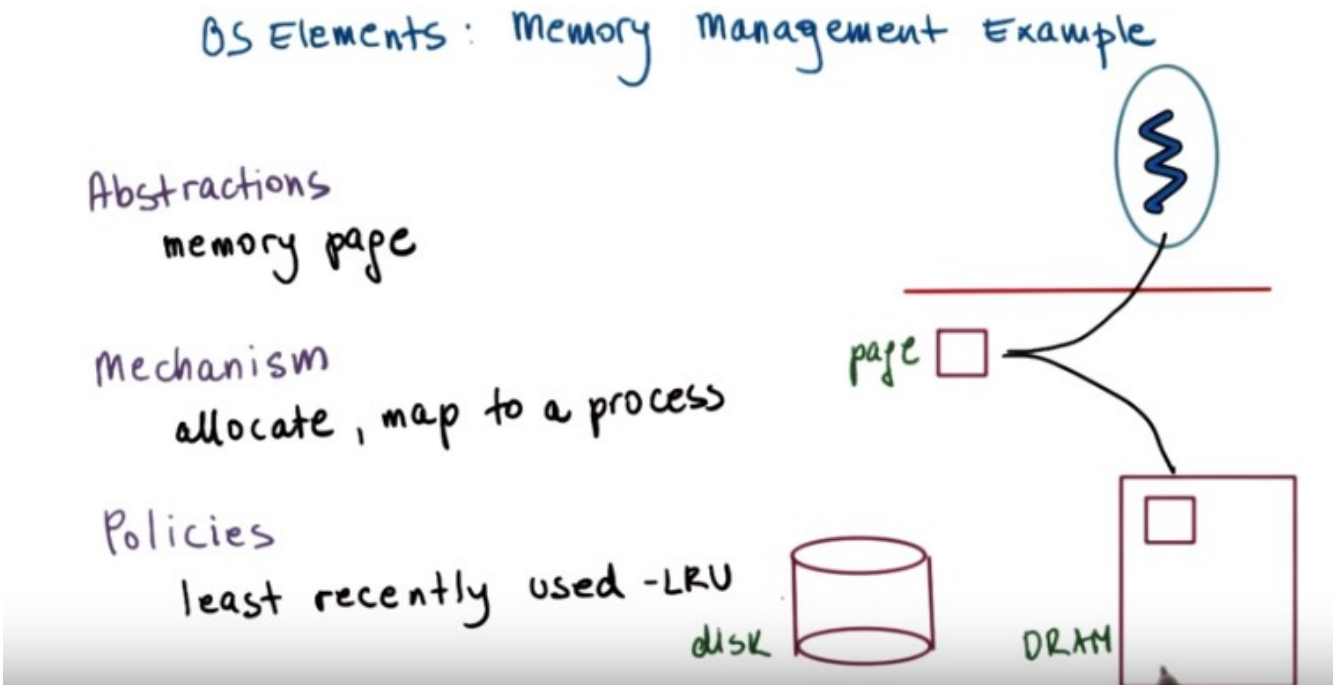


9. Now that we understand what an operating system is, we can ask ourselves, what are some examples of actual operating systems? The examples of real systems out there differ based on the environment that they target. For instance, certain operating systems target more of the desktop environment o.r the server environment. Others target more of the embedded environment. Yet another set of operating systems target ultra high end machines like mainframes. But we'll focus on these environments, the desktop and embedded in our discussions just because they're most common. And also with these examples we'll really focus on more recent, more current really operating systems, as opposed to those that have been around or that have evolved over the last 60 plus years of computer science. For desktop operating systems one of the very popular ones is Microsoft Windows. It is been a well-known operating system since the early 1980s. Next there's a family of operating systems that extended from the operating system that originated at Bell Labs in the late 1960s, and these are all referred to as UNIX-based operating systems. This involves the Mac OS X operating system for Apple devices, and this extends the UNIX BSD kernel, and BSD here is really Berkley System Distribution of Unix. And Linux is another very popular UNIX like system, and it is open sourced, and it comes bundled with many popular software libraries. There are in fact many different versions of Linux. Ubuntu, CentOS, etc. On the embedded side, we've recently seen bigger proliferation of different operating systems, just because of the rising number of smartphones and user devices, like tablets, and now even smaller form factor devices. First you're probably very familiar with Android. It's an embedded form of Linux that runs on many of these types of devices. And its versions come with funny names like Ice Cream Sandwich and KitKat. Next we have iOS and that's the, Apple proprietary operating system for devices

like iPhones and iPads. Then there is Symbian, and then there are other less population options. In each of these operating systems there are a number of unique choices in their design and implementation, and in this class we will particularly focus on Linux. So the majority of more illustrative, more in-depth examples will be given based on the Linux operating system.



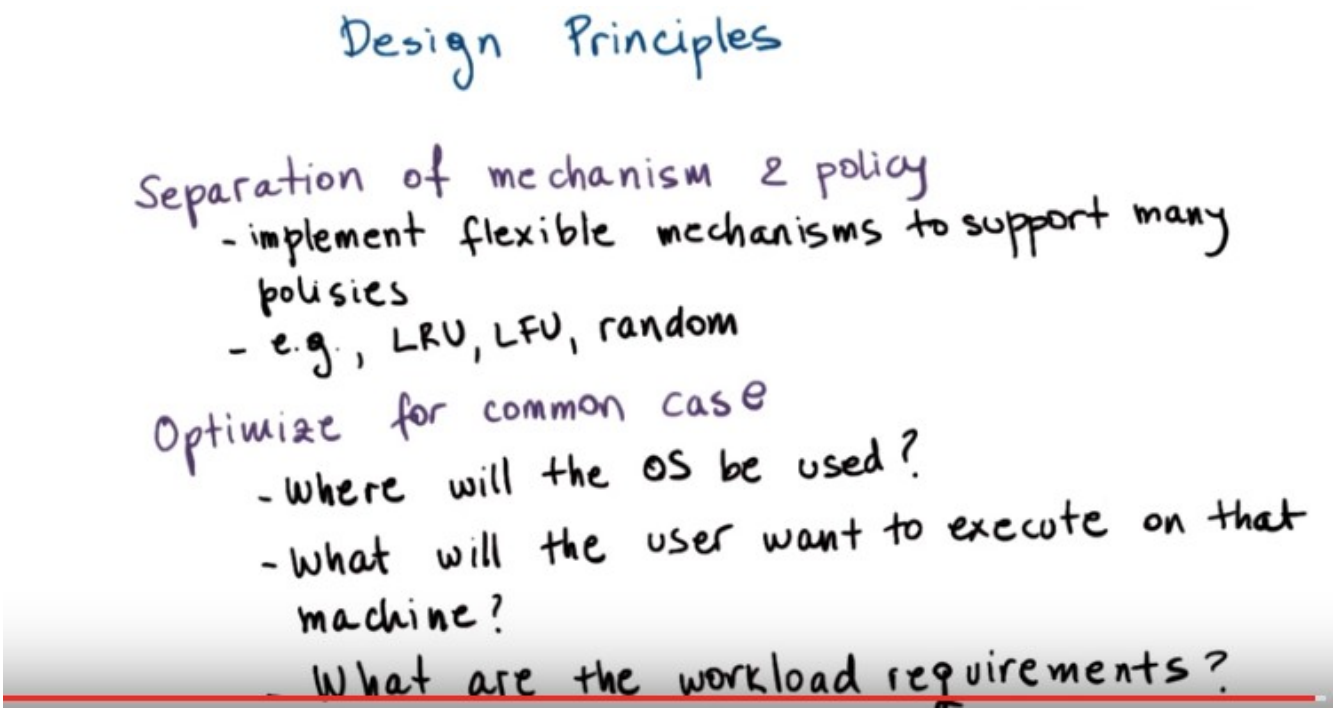
10. To achieve its goals, an operating system supports a number of higher-level abstractions, and then a number of key mechanisms that operate on top of these instructions. For instance, some of these abstractions, like process and thread, correspond to the applications that the operating system executes. Some corresponding mechanisms would be mechanisms to create, to launch an application to start executing, or to schedule it to actually run on the CPU. Other OS abstractions like file or socket that we've mentioned before or memory page, they may more closely correspond to the hardware resources that the operating systems need to manage. Storage device like disk, or a network card for the socket, or the actual memories, so, memory pages in abstraction abstract memory as a resource. To operate on these abstractions, the operating system may incorporate mechanisms to open gain access to a particular device or hardware component, to write to it, to update its state, to allocate to make sure that a particular application has access to that resource. These are some examples of mechanisms. Operating systems may also integrate specific policies that determine exactly how these mechanisms will be used to manage the underlying hardware. For instance, a policy can control what is the maximum number of sockets that a process can actually have access to. Or they may control which data will be removed from physical memory, for instance, based on some algorithm like least-recently used.



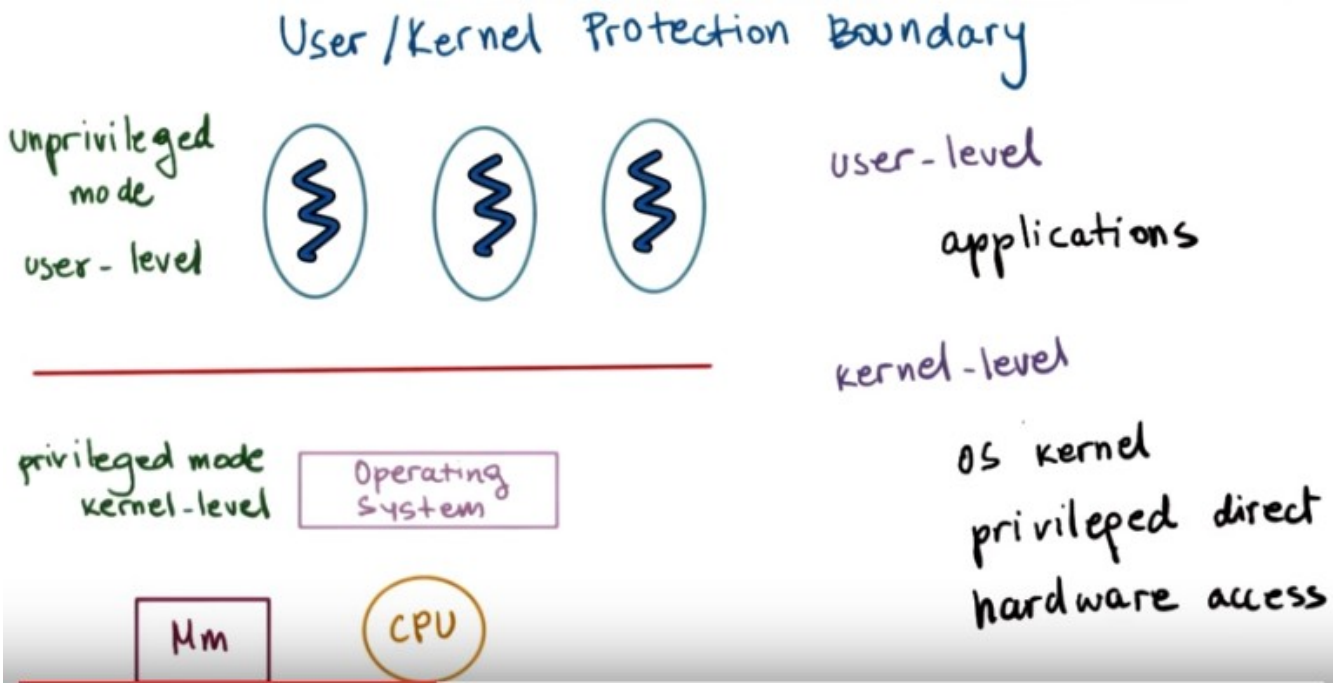
Online: 内存即 RAM

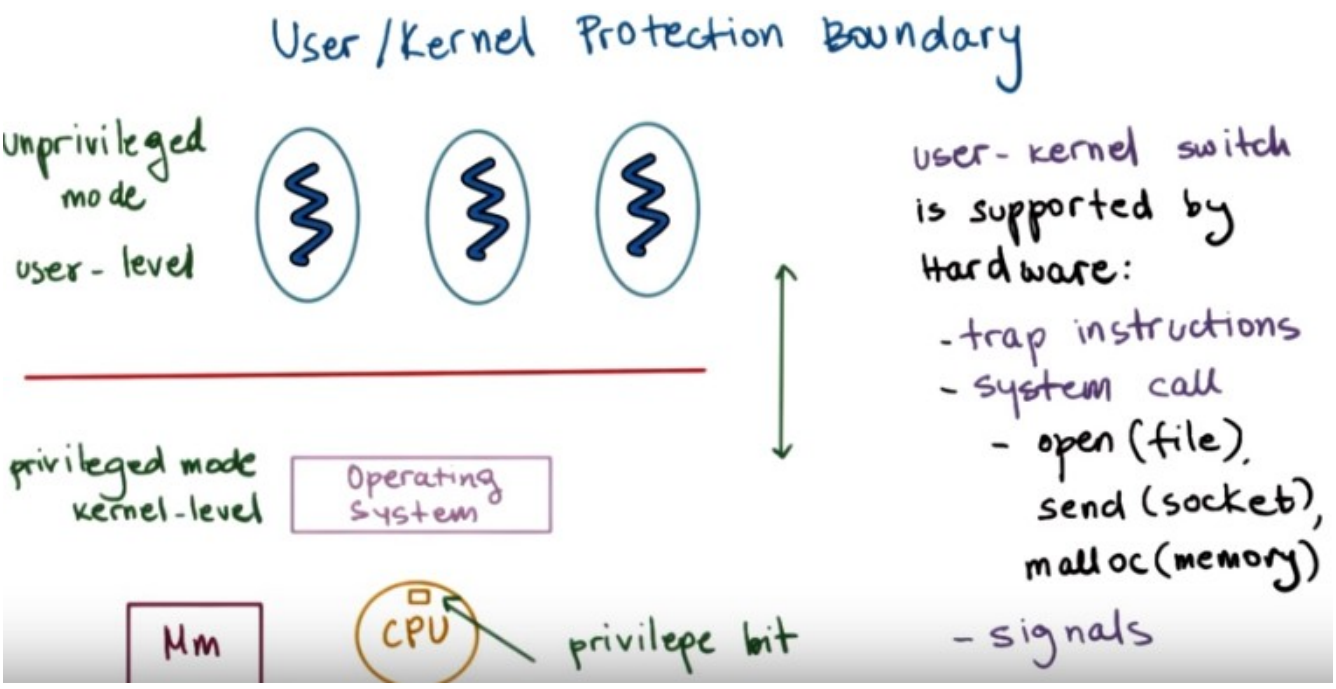
11. Let's look at an example. And, for instance, we said one of the responsibilities of the operating system is to manage resources like memory. So, we'll look at a memory management based example. To do that, the operating system uses a memory page as an abstraction. And this abstraction corresponds to some addressable region of memory of some fixed size, for instance, four k. The operating system also integrates a number of mechanisms to operate on that page. It can allocate that page in DRAM, and it can map that page into the address piece of the process. By doing that it, allows the process to access the actual physical memory that corresponds to the contents of that page. In fact, over time, this page may be moved across different locations in physical memory. Or, it sometimes may even be stored on disk, if we need to make room for some other content in physical memory. This last one brings us to the third element, policies. Since it is faster to access data from memory then on disk, the operating system must have some policies to decide whether the contents of this page will be stored in physical memory or copied on disk. And, a common policy that operating systems incorporate is one that decides that the pages that have been least recently used over a period of time are the ones that will no longer be in physical memory, and instead will be copied on this. We refer to this also as swappings. So, we swap the pages. It's no longer in physical memory, it's in disk. The rational for that is that pages that have not been accessed in a while, so the least recently used ones, are likely not to be as important, or likely will not even be used any time in the near future. And, that's why we can afford to copy them on disk. The ones that have been accessed more frequently are likely more important, or likely recurrently working on that particular part of the content, so we will continue

accessing them, and that's why we maintain them in memory.



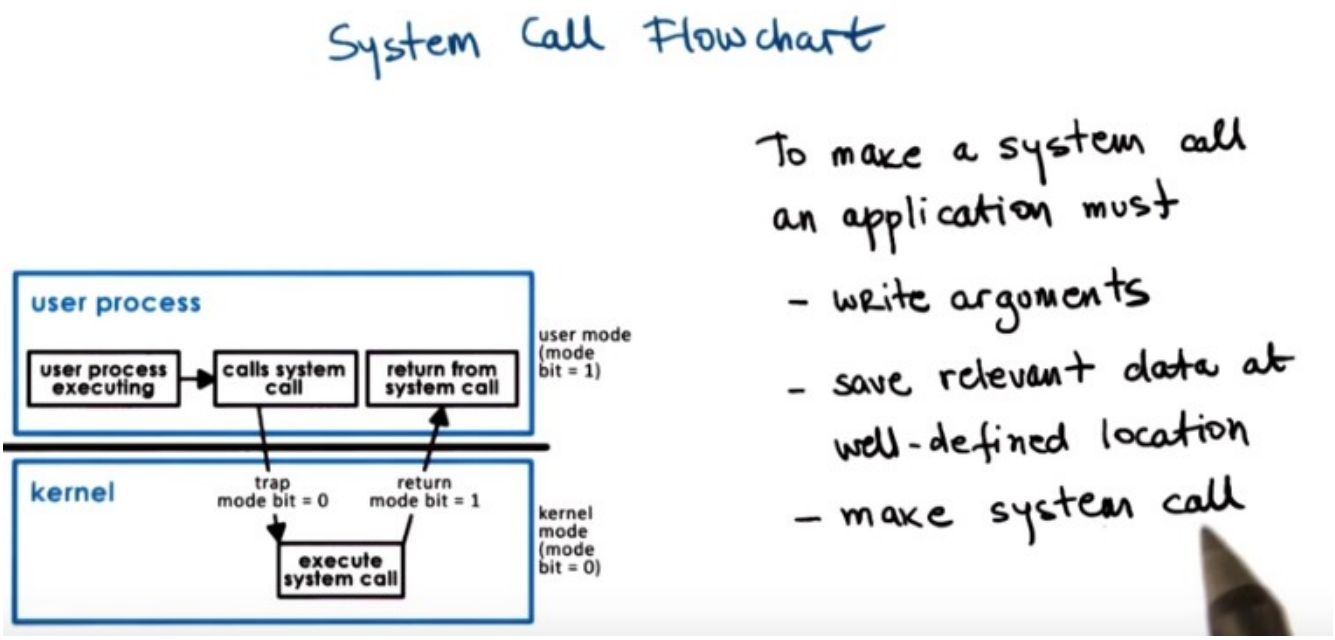
12. Let's look at some good guiding policies when thinking about how to design an operating system. The first one we call separation between mechanisms and policy. What this means is that we want to incorporate into the operating system a number of flexible mechanisms that can support a range of policies. For memory management, some useful policies would include least recently used, or least frequently used, or completely random. So what that means is that in the operating system, we'd like to have some mechanism to track the frequency or the, the time when memory locations have been accessed. This will help us keep track of when a page was last used or when a page was least frequently used. Or we can completely ignore that information. But the bottom line is we can implement any one of these policies in terms of how that memory management is going to operate. And the reason is that in different settings, different policies make more sense. This leads us to the second principle, which is optimize for the common case. What this means is that we need to understand a number of questions, how the operating system will, will be used, the, what it will need to provide in order to understand what the common case is. This includes understanding where will it be used, what kind of machine it will run on, how many processing elements does it have, how many CPUs, how much memory, what kinds of devices. And we also need to understand what are the common things that the end users will do on that machine. What are the applications that they will execute, and also what are the requirements of that workload? So how does that workload behave? We need to understand the common case, and then based on that common case, pick a specific policy that makes sense and that can be supported given the underlying mechanisms and abstractions that the operating system supports.

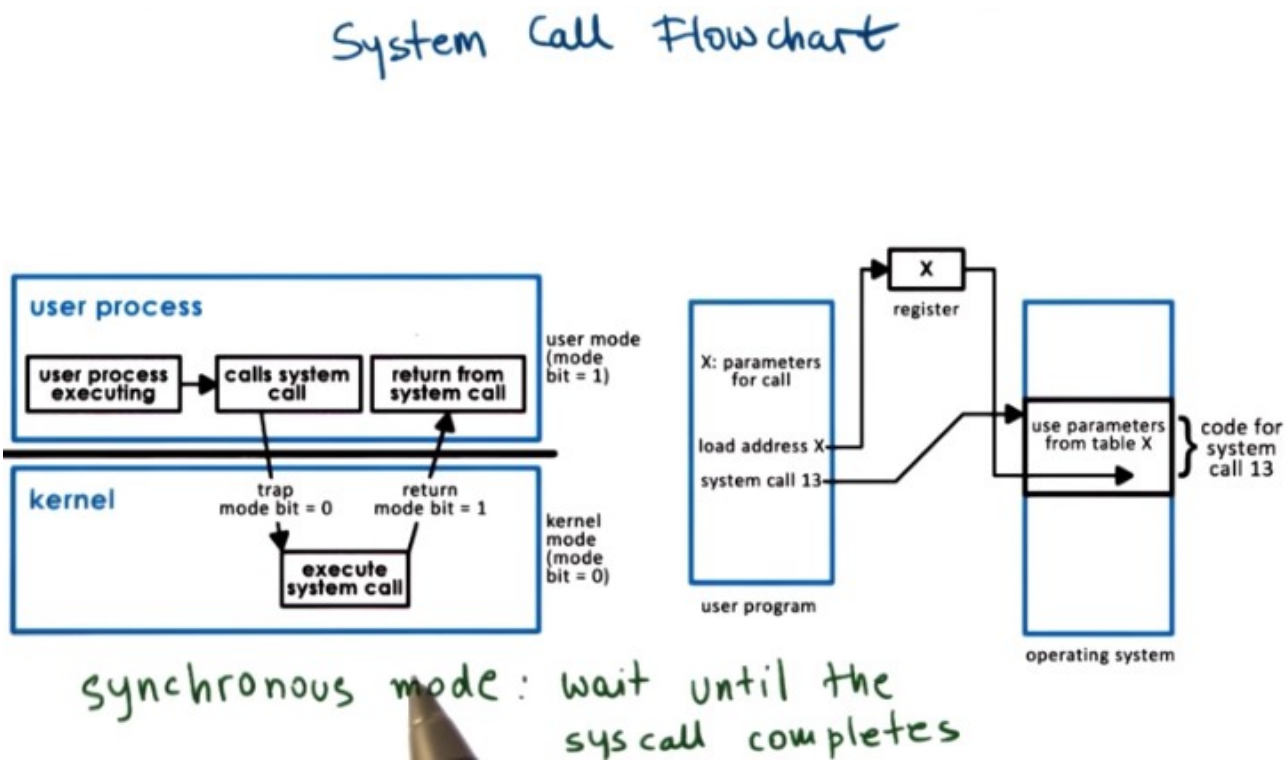




注意上圖上半部分是 unprivileged, 下半部分是 privileged.

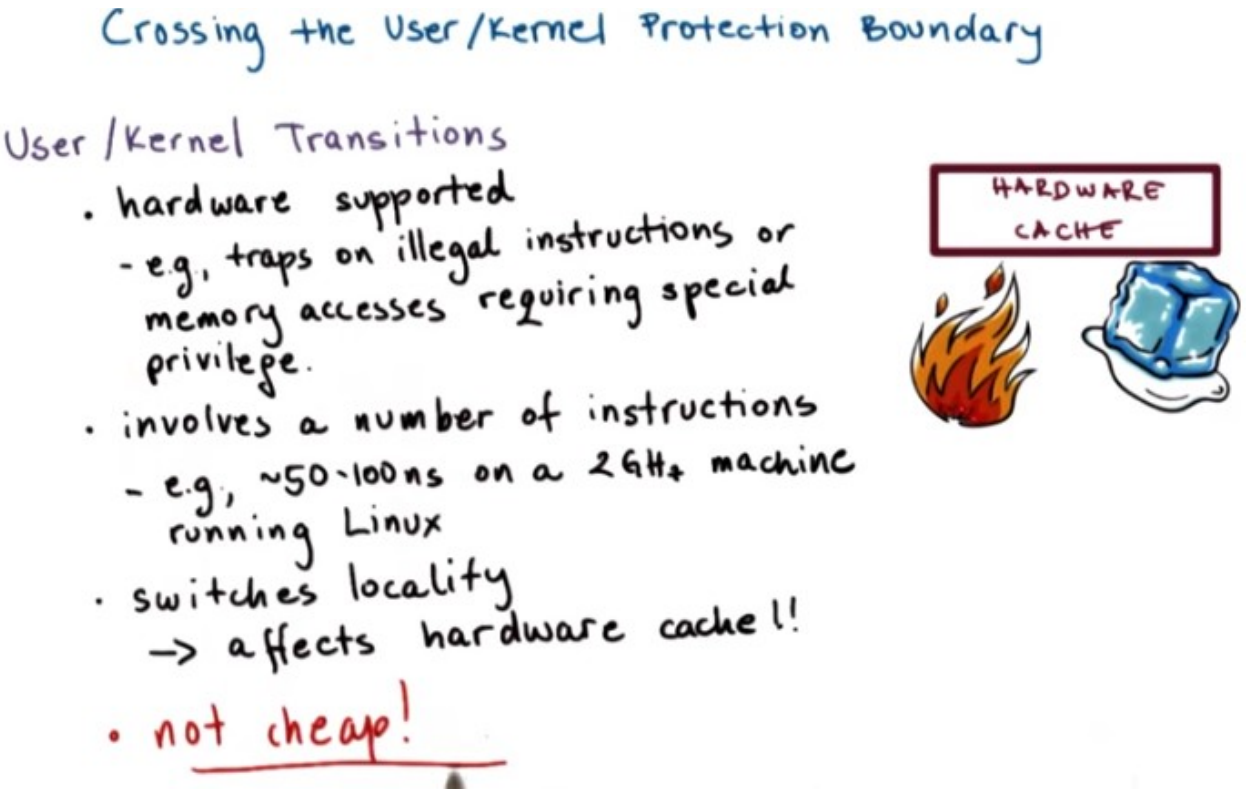
13. To achieve its role of controlling and managing hardware resources on behalf of applications, the operating system must have special privileges, as the definition pointed out, to have direct access to the hardware. Computer platforms distinguish between at least two modes, privileged kernel mode, and unprivileged or user mode. Because an operating system must have direct hardware access, the operating system must operate in privileged kernel mode. Note the rectangle labeled Mm, this means main memory, and I will use this simplified drawing of memory and CPU throughout this course. The applications in turn operate in unprivileged or user mode. Hardware access can be performed only from kernel mode by the operating system kernel. Crossing from user level into kernel level and the other way around, or in general, distinguishing between the two is supported by the hardware on most modern platforms. For instance, when in kernel mode, a special bit is set in the CPU, and if this bit is set, any instruction that directly manipulates hardware is permitted to execute. When in user mode, this bit is not set, and such instructions that attempt to perform privileged operations will be forbidden. In fact, such attempts to perform a privileged operation when in user mode will cause a trap. The application will be interrupted, and the hardware will switch the control to the operating system at a specific location. At that point, the operating system will have a chance to check what caused that trap to occur, and then to verify if it should grant that access or if it should perhaps terminate the process if it was trying to perform something illegal. In addition to this trap method, the interaction between the applications and the operating system can be via these system call interface. The operating systems export a system call interface. So, the set of operations that the applications can explicitly invoke if they want the operating system to perform certain service and to perform certain privileged access on their behalf. Examples would be open to perform access to a file, or send to perform access to a socket, or malloc to allocate memory, many others. And operating systems also support signals, which is a mechanism for the operating system to pass notifications into the applications. And I will talk about these in a later lesson.





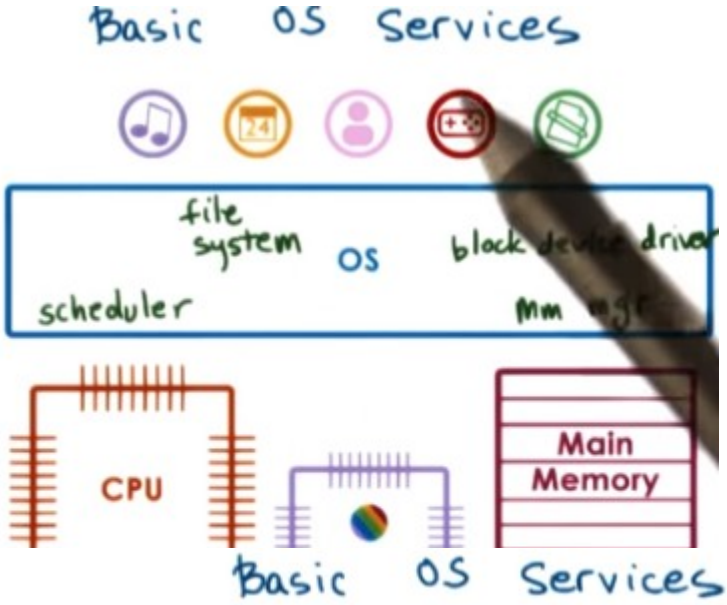
14. Let's talk a little bit more about system calls. Using this diagram I'm going to trace through where control and data are exchanged during a system call. I will use this icon to denote where I am in the diagram. We will start by assuming we are currently in an executing user process. Then because the user process needs some hardware access, it makes a system call. On a system call, control is passed to the operating system, in privileged mode, and the operating system will perform the operation and then it will return the results to the process. Executing a system call involves changing the execution context from the user process to that of the OS kernel, also passing arguments, whatever necessary for the system cooperation. And then jumping somewhere in the memory of the kernel so that you can go through the instruction sequence that corresponds to that system call. With the system call, control is passed to the operating system. The operating system operates in privileged mode. And it's allowed to perform whatever operation was specified in the system call. Once the system call completes, it returns the result and the control back to calling process. And this again will involve changing the execution context from, now from kernel mode into the user mode, passing any arguments back into the user address space, and then, jumping to the exact same location in the execution of the user process where the system call was being made from. But the entire process involved changing the execution context from user to kernel mode and back, passing arguments, jumping around in memory to locations where the code to be executed is. So, it's not necessarily a cheap operation. To make a system call, an

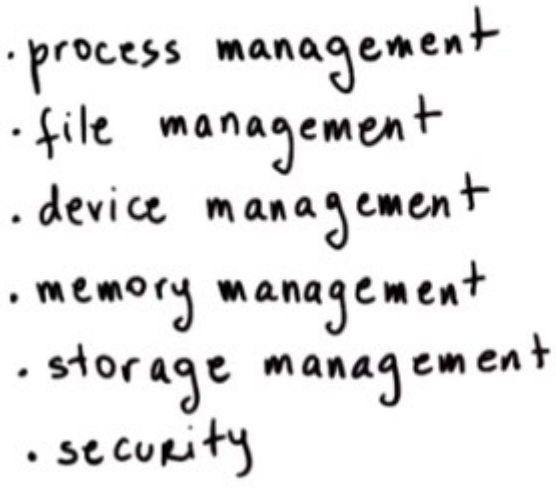
application must write arguments, save all relevant data at a well-defined location, make the actual system call using this specific system call number. The well-defined location is necessary so that the operating system kernel, based on the system call number, can determine which, how many arguments it should retrieve and where are they, at this well-defined location? The arguments can either be passed directly between the user program and the operating system, or they can be passed indirectly by specifying their address. In synchronous mode, the process will wait until the system call completes. I will talk about an alternative where we can issue asynchronous system calls, but that we will leave for a later discussion in this course. For now, you must understand that there are some basic steps (即應該不用在乎細節) involved in calling an operating system service and obtaining the results.

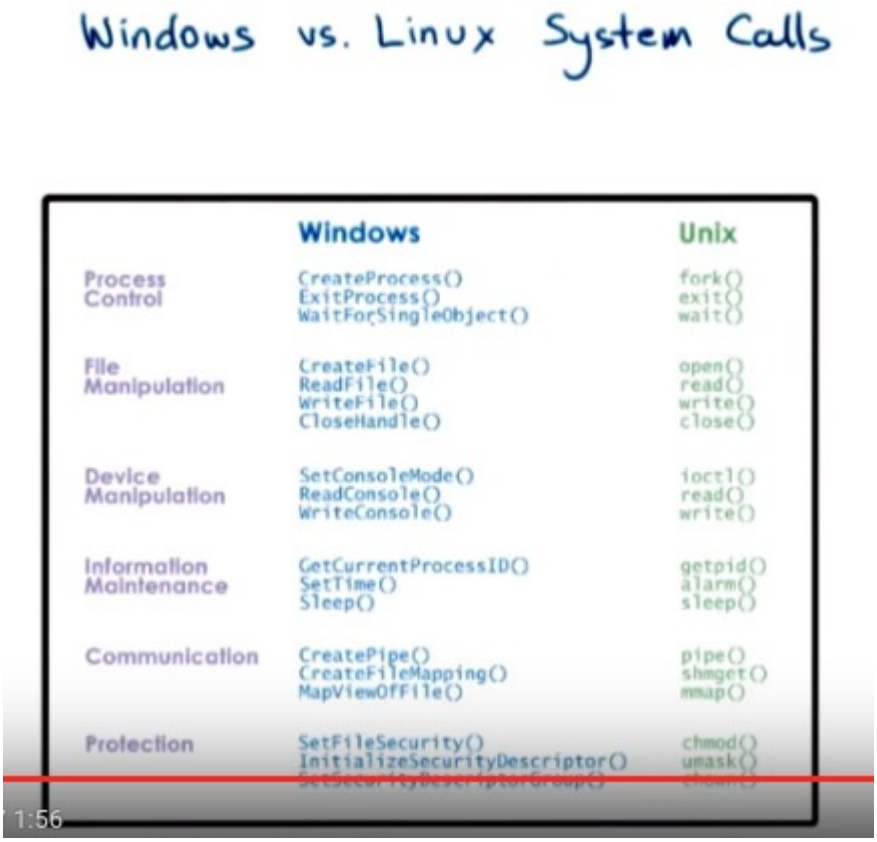


15. In summary, user/kernel transitions are a necessary step during application execution. Applications may need to perform access to certain types of hardware. Or, may need to request change in the allocations of hardware resources that have been made to them. Only the operating system, the kernel, are allowed to perform those types of operations. The hardware provides support for performing user/kernel transitions. We explain that the hardware will cause a trap if the application from unprivileged mode tries to perform some instruction, or a memory access for which it doesn't have permissions. For instance, the application cannot change the contents of certain registers, and give itself more CPU, or more memory. Only the operating system can do that. The result of this trap is that the hardware initiates transfer of the control to the operating system, to the kernel, and marks this by that special privilege bit that we mentioned. At that point, once control is passed over to the operating system, the operating system can check what caused the trap, and determine what's the appropriate thing to do. Whether to grant or deny the specific request that caused the trap to occur in the first place. This will, of course, depend on the policies that are supported by the operating system. Performing all of this, despite of the fact that hardware provides support, still takes a number of

instructions. For instance, on a two gigahertz machine running Linux,it can take 50 to 100 nanoseconds to perform all the operations that are necessary around a user/kernel transition. This is real time, real overhead for the system. The other problem with these transitions is they affect the hardware cache usage. The application performance is very dependent on the ability to use the hardware cache. If accessing cache is order a few cycles, accessing memory can be order of hundreds of cycles. When we perform a system call, or in general when we cross into the operating system, the operating system, while executing, will likely bring content that it needs in the cache. This will replace some of the application content that was in the hardware cache before that transition was performed. And, so this will have some impact on the application performance, because it will no longer be able to access its data in cache, it will have to go to memory. In summary, these user/kernel transitions, they're not cheap.





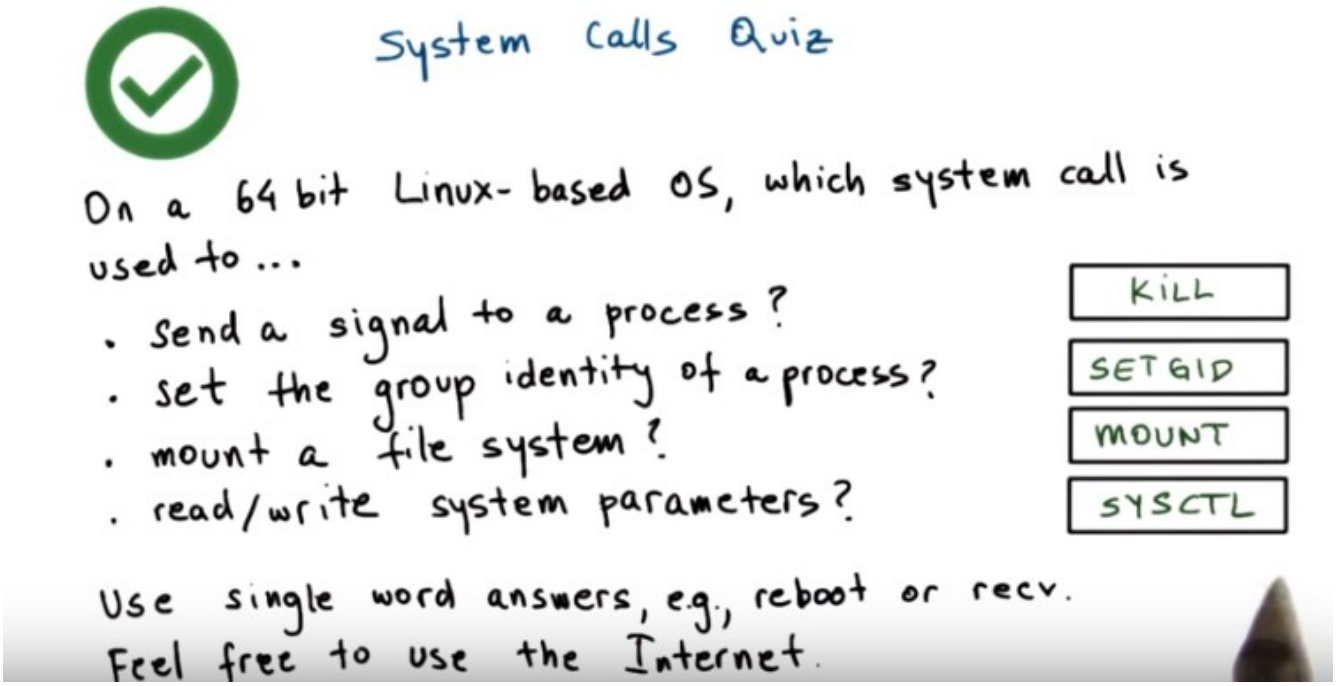


由上圖可知, 一些典型的 system calls (Unix)為: 對文件的 open(), read(), write(), close(), 對 process 的 fork(), 對 bash 等的 chmod().

16. An operating system provides applications with access to the underlying hardware. It does so by exporting a number of services. At the most basic level, these services are directly linked to some of the components of the hardware. For instance, there is a scheduling component that's responsible for

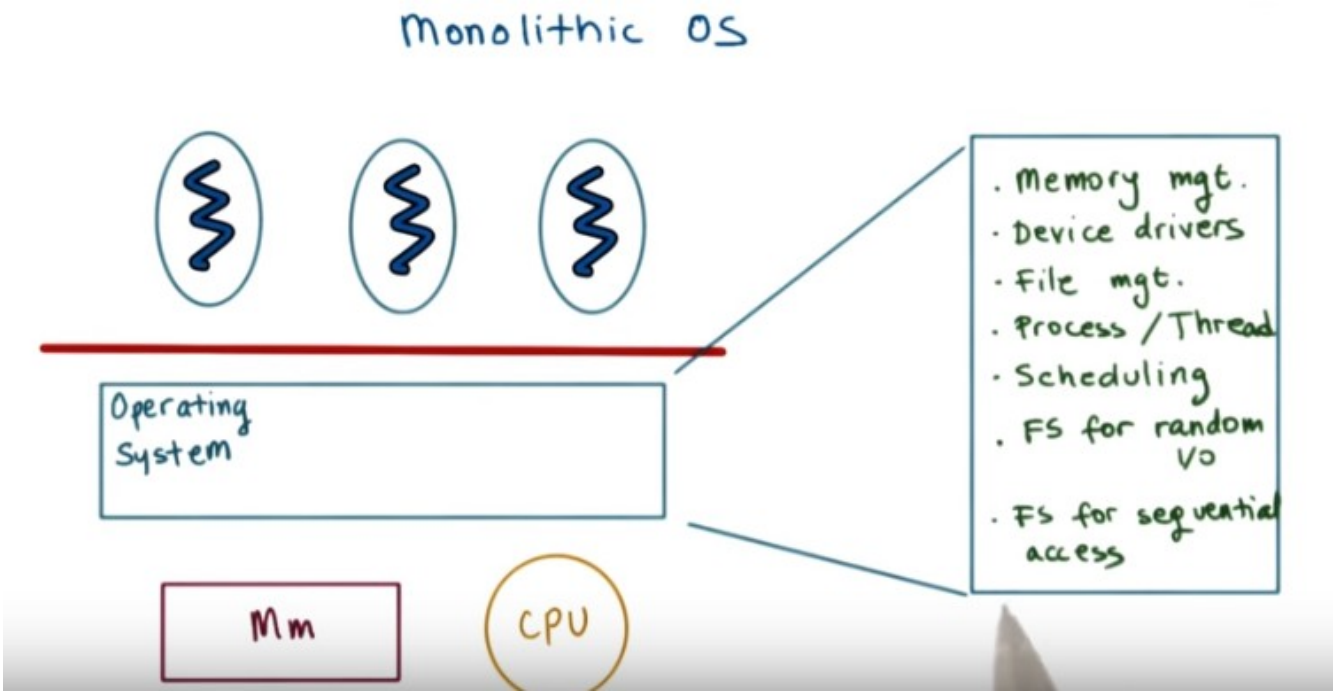
controlling the access to the CPU, or maybe there are even multiple CPUs. The memory manager is responsible for allocating the underlying physical memory to one or more co-running applications. And it also has to make sure that multiple applications don't overwrite each other's accesses to memory. A block device driver is responsible for access to a block device like disk. In addition, the operating system also exports higher-level services that are linked with higher-level abstractions, as opposed to those that are linked with abstractions that really map to the hardware. For instance, the file is a useful abstraction that's supported by virtually all operating systems. And in principle, operating systems integrate file system as a service. In summary, the operating system will have to incorporate a number of services in order to provide applications and application developers with a number of useful types of functionality. This includes process management, file management, device management, and so forth. Operating systems make all of these services available via system calls. For example, here are some system calls in two popular operating systems, Windows and Unix. I will not read through this list, but notice although these are two very different operating systems, the types of system calls and the abstractions around those systems calls these two OSes provide are very similar. But process control, creating a process, exiting a process, waiting for object, creating files, etc.

17. Because we have been discussing system calls, I would like for you to take a quiz. You will need to fill in the following statement. On a 64 bit Linux-based operating system, which system call is used to send a signal to a process? To set the group identity of a process? To mount a file system? Or to read/write system parameters? Please use only single word answers. For instance, just reboot, and also feel free to use the Internet.

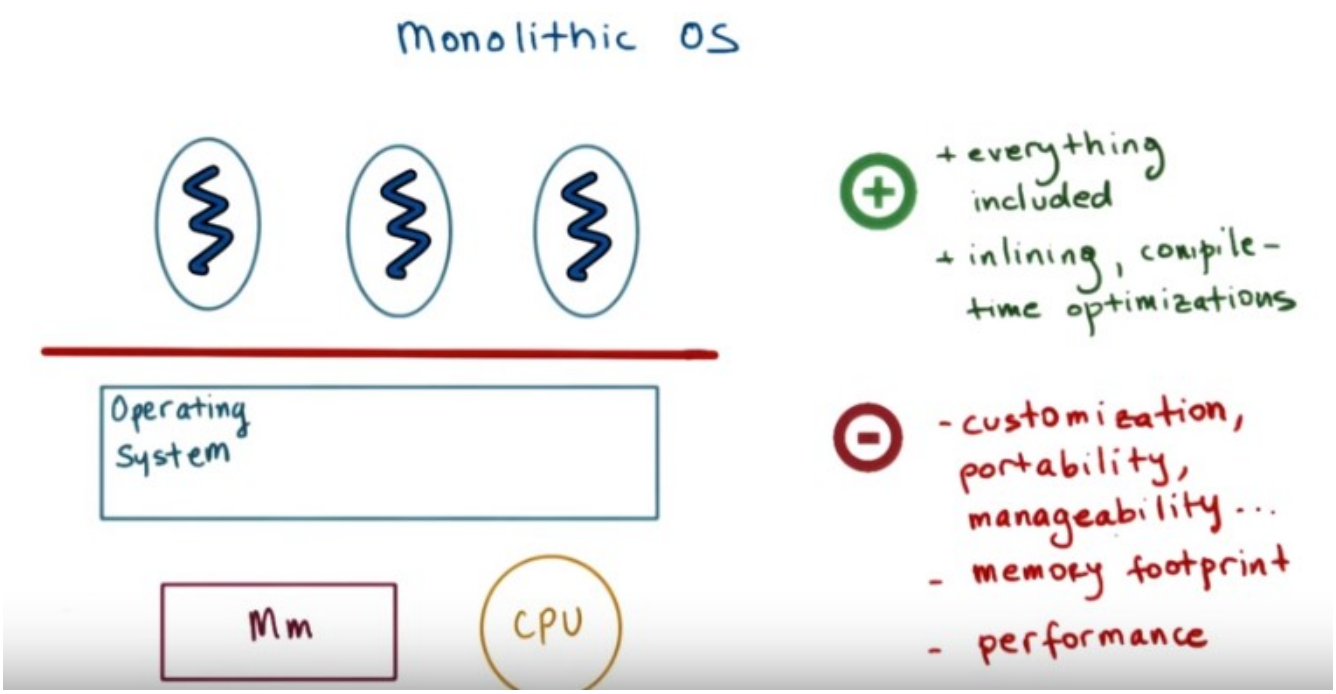


18. The answers to this quiz are as follows. To send a signal to a process, there is a system called kill. To set the group identity of a process, there is a system called SETGID. This is valid on 64 bit operating systems, on 16 or 32 bit systems, there is a variant SETGID 16, or SETGID 32. Mounting a

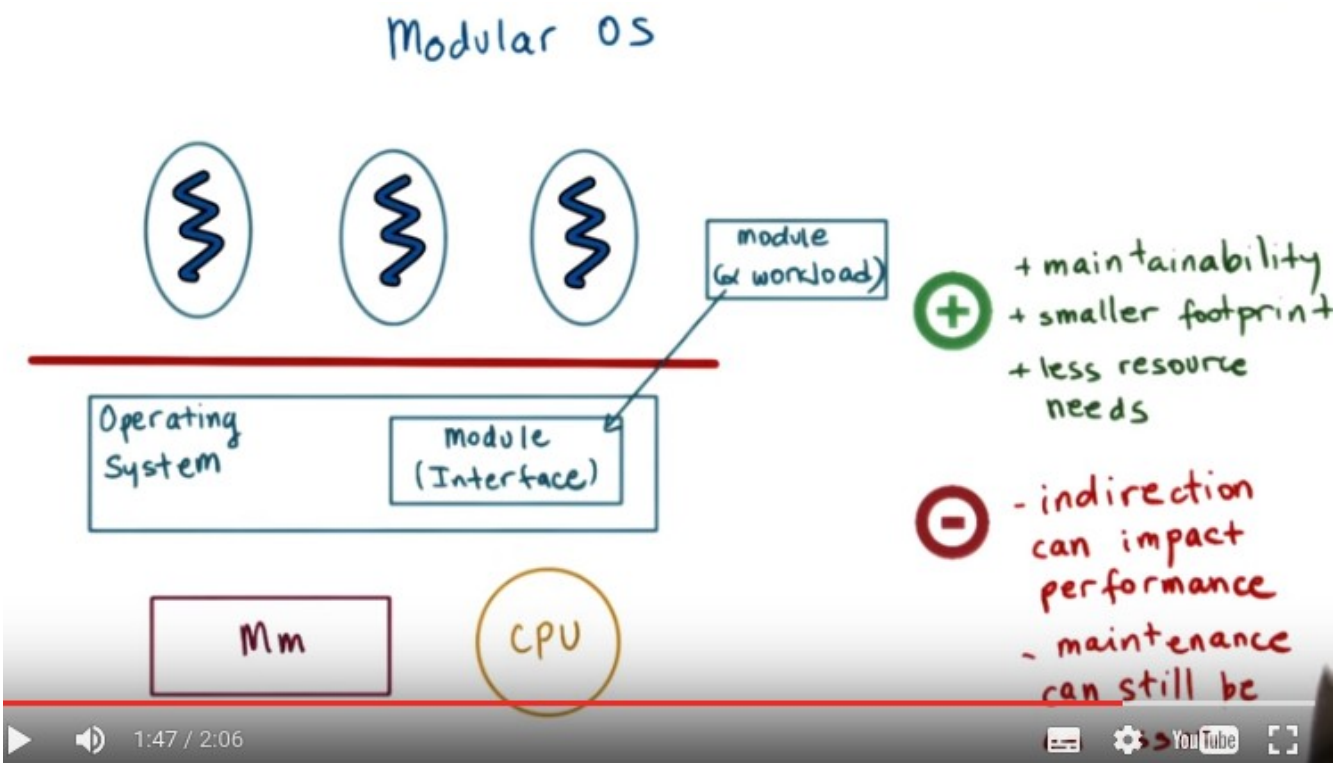
file system is done via the mount system call. And finally reading or writing system parameters is done via the system control system call, SYSCTL.



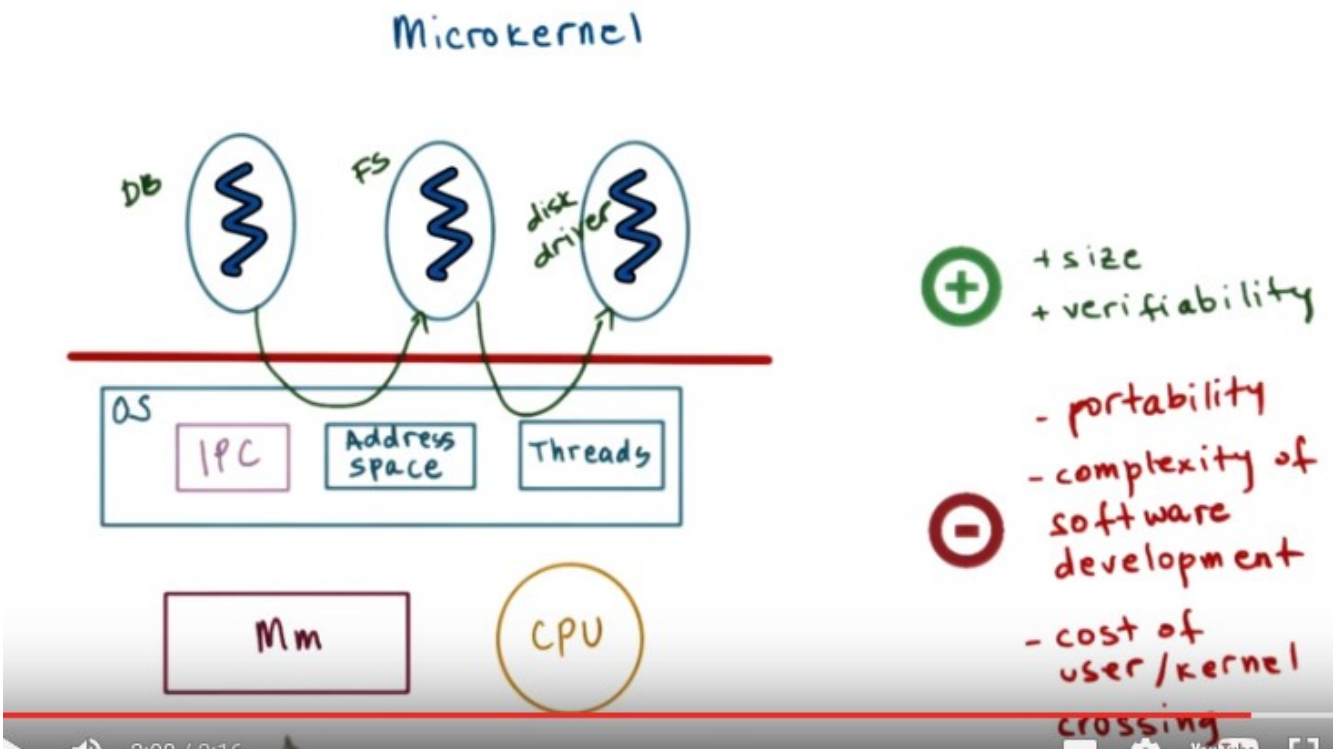
Monolithic: 單片的, 整體的. 上圖中的 FS 即 file system.



19. We saw so far some rough indications how an operating system is laid out. But let's now more explicitly look at different types of operating system organizations, and we will start with what we call a monolithic operating system. Historically, the operating system had a monolithic design. That's when every possible service that any one of the applications can require or that any type of hardware will demand is already part of the operating system. For instance, such a monolithic operating system may include several possible file systems, where one is specialized for, of sequential workloads where the workload is sequentially accessing files when reading and writing them. And then maybe other file system that's optimized for random I/O. For instance, this is common with databases. There isn't necessarily a sequential access there. Rather, each database operation can randomly access any portion of the backing file. This would clearly make the operating system potentially really, really large. The benefit of this approach is that everything is included in the operating system. The abstractions, all the services, and everything is packaged at the same time. And because of that, there's some possibilities for some compile-time optimizations. The downside is that there is too much state, too much code that's hard to maintain, debug, upgrade. And then its large size also poses large memory requirements, and that can always impact the performance that the applications are able to observe.

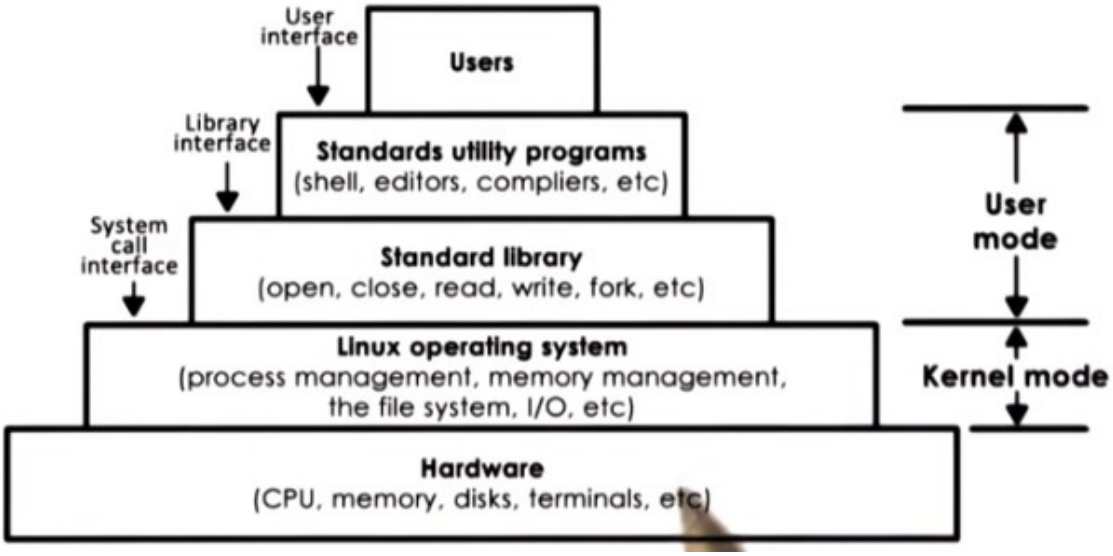


20. A more common approach today is the modular approach, as with the Linux operating system. This kind of operating system has a number of basic services and APIs already part of it, but more importantly, as the name suggests, everything can be added as a module. With this approach, you can easily customize which particular file system or scheduler the operating system uses. This is possible because the operating system specifies certain interfaces that any module must implement in order to be part of the operating system. And then dynamically, depending on the workload, we can install a module (如裝驅動) that implements this interface in a way that makes sense for this workload. Like, if these are database applications, we may run the file system that's optimized for random file access. And if these are some other types of computations, we may run the file system that's optimized for sequential access. And most importantly, we can dynamically install new modules in the operating system. The benefits of this approach is that it's easier to maintain an upgrade. It also has a smaller code base and it's less resource intensive, which means that it will leave more resources more memory for the applications. This can lead to better performance as well. The downside of this approach is that all the modularity may be good for maintainability. The level of interaction that it requires, because we have to go through this interface specification before we actually go into the implementation of a particular service. This can reduce some opportunities for optimizations. Ultimately, this can have some impact on performance, though, typically, not very significant. Maintenance, however, can still be an issue given that these modules may come from completely disparate code bases and can be a source of bugs. But overall, this approach delivers significant improvements over the monolithic approach, and it's the one that's more commonly used today.



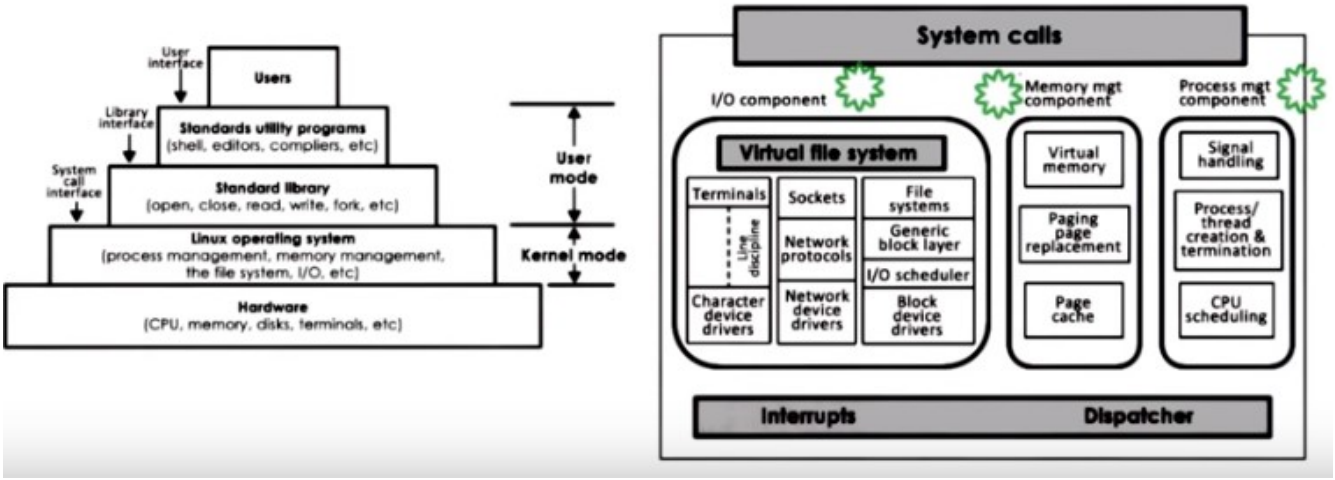
21. Another example of OS design is what we call a microkernel. Microkernels only require the most basic primitives at the operating system level. For instance, at the OS level, the microkernel can support some basic services such as to represent an executing application, its address space, and its context, so a thread. Everything else, all other software components, applications like databases, as well as software that we typically think of as an operating system component, like file systems, device drivers, everything else will run outside of the operating system kernel at user level, at unprivileged level. For this reason, this microkernel-based organization of operating systems requires lots of inter-process interactions. So typically, the microkernel itself will support inter-process communications as one of its core abstractions and mechanisms, along with address spaces and threads. The benefits of a microkernel is that it's, it's very small. This can not only lead to lower overheads and better performance, but it may be very easy to, to verify, to test that the code exactly behaves as it should. And this makes microkernels valuable in some environments where it's particularly critical for the operating systems to behave properly, like some embedded devices or certain control systems (後面的 Mac OS architecture 圖後的文字也出現了 microkernel 和 primitives). These are some examples where microkernels are common. The downsides of the microkernel design are that although it is small, its portability is sort of questionable because it is typically very specialized, very customized to the underlying hardware. The fact that there may be more one-off versions of a microkernel specialized for different platforms makes it maybe harder to find common components of software, and that leads to software complexity as well. And finally, the fact that we have these very frequent interactions between different processes, these different user-level applications, means that there is a need for frequent user/kernel crossings. And we said already that these can get costly.

Linux Architecture

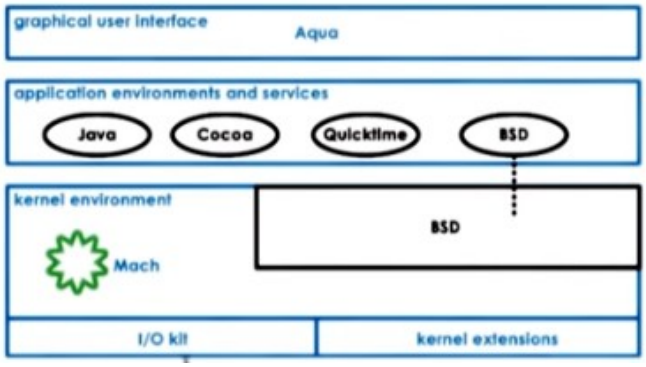


注意上圖中的 system call. 上圖從上到下, 可以簡記為: User → programs → library (做 system call) → OS → hardware

Linux Architecture

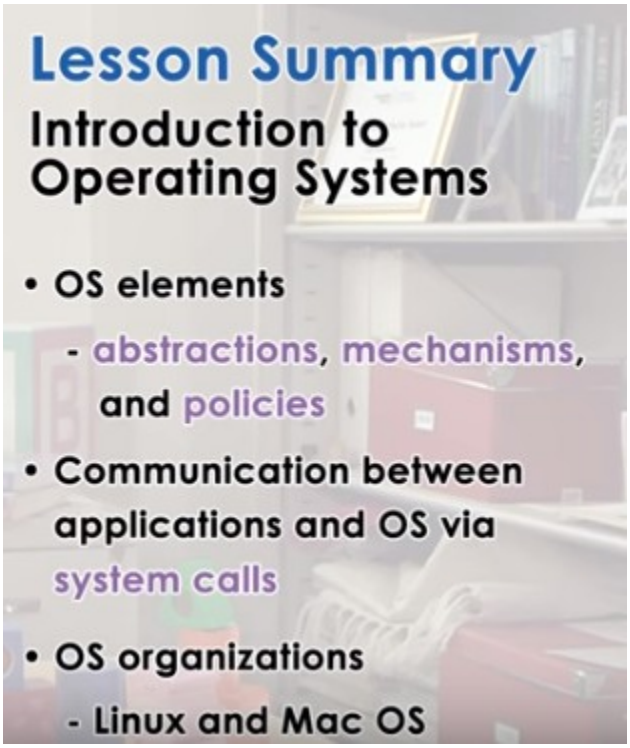


Mac OS Architecture



22. Let's look at some popular operating systems, starting with the Linux architecture. This is what the Linux environment looks like. Starting at the bottom, we have the hardware, and the Linux kernel abstracts and manages that hardware by supporting a number of abstractions and the associated mechanisms. Then come a number of standard libraries, such as those that implement the system call interfaces, followed by a number of utility programs that make it easier for users and developers to interact with the operating system. And, finally, at the very top, you have the user developed applications. The kernel, itself, consists of several logical components, like all of the the I/O

management, memory management, process management. And, these have well defined functionality, as well as interfaces. Each of these separate components can be independently modified or replaced. And, this makes the modular approach in Linux possible. The Mac OSX operating system, from Apple, uses a different organization. At the core is the Mac micro kernel and this implements key primitives like memory management, thread scheduling and interprocess communication mechanisms including for, what we call RPC. The BSD component provides Unix interoperability via a BSD command line interface, POSIX API support as well as network I/O. All application environments sit above this layer. The bottom two modules are environments for development of drivers, and also for kernel modules that can be dynamically loaded into the kernel.



再次注意上圖中說的: Communication between applications and OS via system calls

23. In this lesson, we answered the big question, what is an operating system? And we saw that it's important because it helps abstract and arbitrate the use of the underlying hardware system. We explained that to achieve this, an operating system relies on a number of abstractions, like processes and threads, a number of mechanisms that allow it to manipulate those abstractions, and a number of policies that specify how those abstractions can be modified. We saw that operating systems support a system call interface that allows applications to interact with them. We looked at several alternatives in organizational structures for operating systems. Then very briefly, we looked at some specific examples of operating systems, Windows, Linux, and Mac OS to see some examples of their system call interfaces or their organization.

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