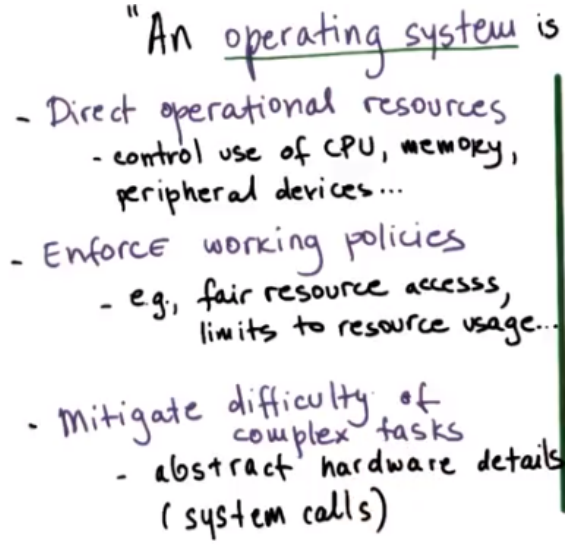
P1L2 - Introduction to Operating Systems

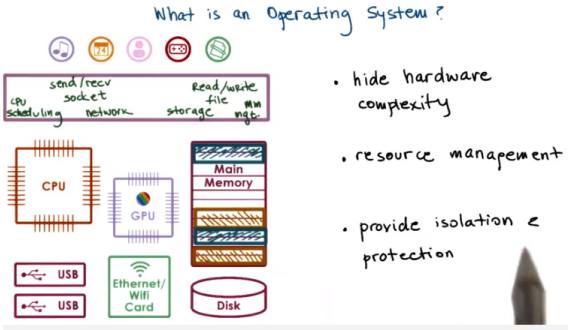
01 - Lesson Preview

In simplest terms, an operating system(OS) is a special piece of software that abstracts and arbitrates the underlying hardware system. Abstract means to simplify what the hardware actually looks like. Arbitrate means to manage, to oversee, to control the hardware use. OS supports a number of abstractions and arbitration mechanisms for the various types of hardware components in computer systems.

02 - Visual Metaphor



03 - What is an Operating System



04 - Operating System Definition

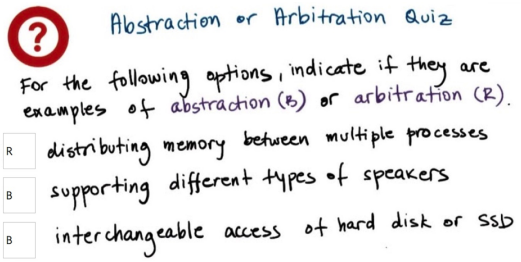
An operating system is a layer of systems software that resides between hardware and applications:
1) it directly has privileged access to the underlying hardware and manipulate its state;
2)its role is to hide the hardware complexity and manage hardware on behalf of one or more applications according to some predefined policies;
3)it ensures that applications are isolated and protected from one another

05 - Operating System Components Quiz

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

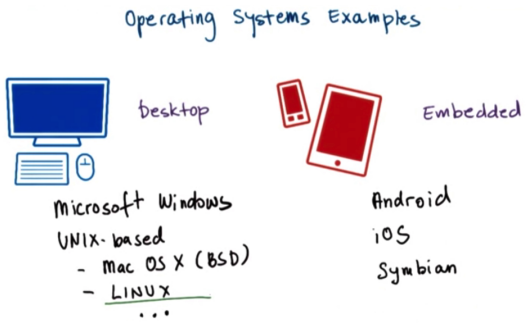
because it's responsible for distributing the access to the processing element, the CPU, among all of the applications that share that platform.

06 - Abstraction or Arbitration Quiz



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.

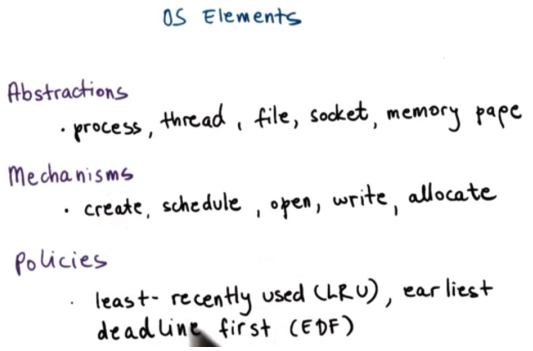
07 - Operating System Examples



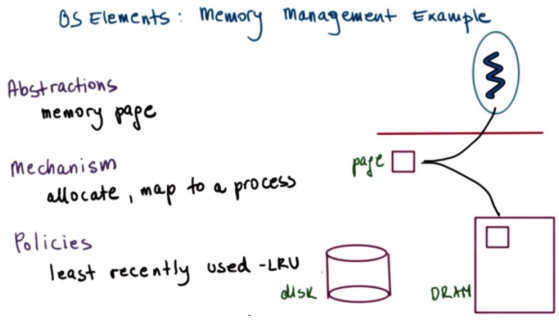
Now that we understand what an operating system is, we can ask ourselves, what are some examples of actual operating systems? The examples of real systems out there differ based on the environment that they target. For instance, certain operating systems target more of the desktop environment or the 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 Berkeley 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 popular 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.

08 - OS Elements

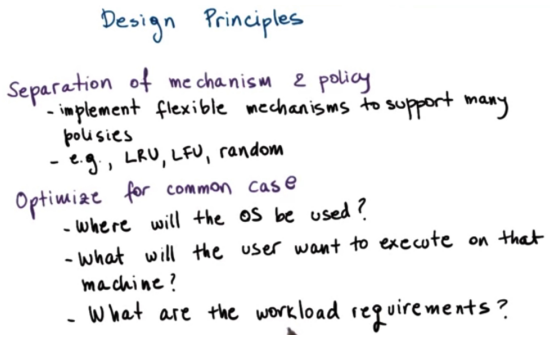


w209 - OS Elements Example



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, 4k.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 space 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 than 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 disk. We refer to this also as swapping. So, we swap the

pages. It's no longer in physical memory, it's in disk. The rationale 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 we're currently wor



king on that particular part of the content, so, we will continue accessing them, and that's why we maintain them in memory.

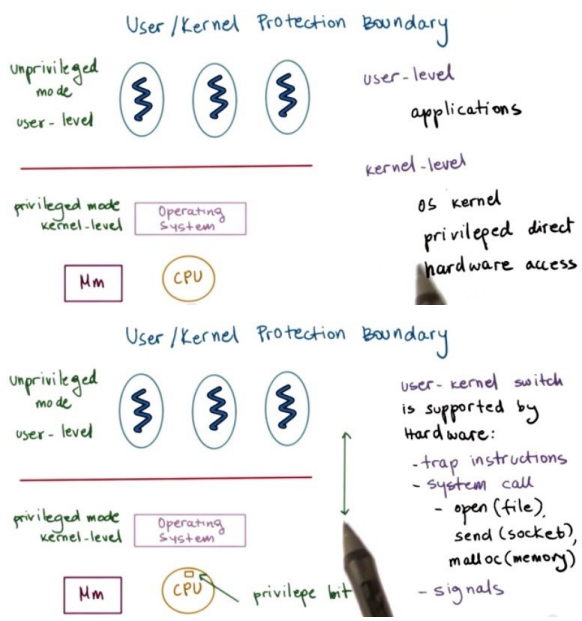
10 - OS Design Principles

Let's look at some good guiding policies when thinking about how to design a good operating system. The first one we call 'separation of 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', 'least frequently used' or completely random. So what this means is that in the operating system we'd like to have some mechanisms to track the frequency or the time when memory location has been accessed. This will help us keep track of when a page was last used or when a page was last 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 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, well, will be used. What it will need to provide in order to understand what the common case is. This includes understanding where it will be used, what kind of machine it will run on, how many processing elements does it have, how many cpu's, how much memory, what kinds of devices… And we also need to understand what are the common things the end-users will do on that machine? What are the applications they will execute and also what are the requirements of that workload and 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 the operating system supports.

11 - OS Protection Boundary

Modes:

user-level is unprivileged
kernel-level is privileged



user -kernel switch is supported by hardware via trap instructions, system calls, signals and etc.

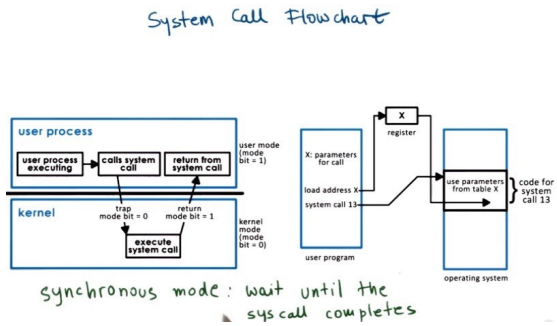
Instructor Notes - Errata ^

@2:14 in the video, it should read **mmap (memory)** instead of malloc (memory).

- **mmap** is a POSIX-compliant Unix system call
- malloc is a standard C library function

To achieve its role of controlling and managing hardware resources on behalf of applications, the operating system must have a set of special privileges, as the definition have 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 labelled 'MM' - this means '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 in 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 instruction such as 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 this system call interface . The operating systems export a system call interface so the set up 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 to perform access to a file, or a 'send' to perform access to a socket or 'mmap' 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

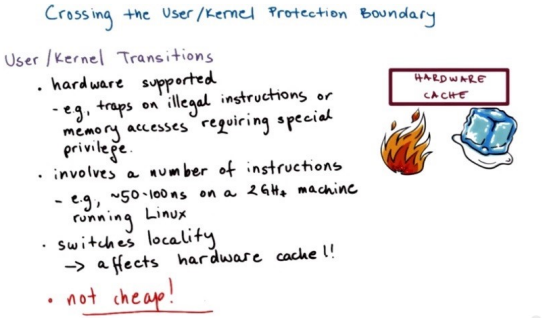
12 - System Call Flowchart



To make a system call an application must

- Write arguments (arguments can be passed directly betweeSave relevant data at n user program and OS or indirectly by specifying their address i.e. above diagram on right)
- well-defined location (necessary so that OS kernel based on system call number can determine how many arguments it should retrieve and where are they at the well defined location)
- Make system call(using the specific system call number)

13 - Crossing the OS Boundary



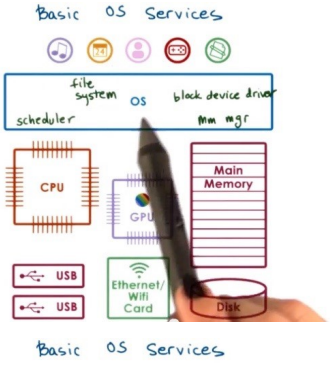
Instructor Notes

Errata

At 1:46 an **image of the hardware cache, fire, and ice cube** appear. This image is not well explained in the video, so here is a better explanation:

Because context switches will swap the data/addresses currently in **cache**, the performance of applications can benefit or suffer based on how a context switch changes what is in **cache** at the time they are accessing it.
A **cache** would be considered **hot** (fire) if an application is accessing the **cache** when it contains the data/addresses it needs.
Likewise, a **cache** would be considered **cold** (ice) if an application is accessing the **cache** when it does not contain the data/addresses it needs -- forcing it to retrieve data/addresses from main memory.

š14 - OS Services



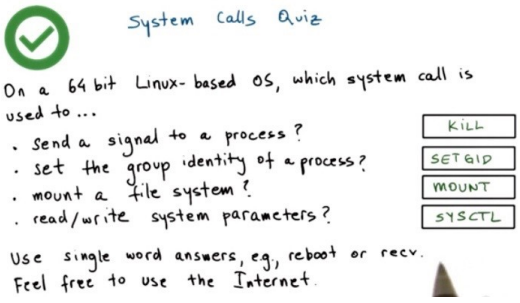
An operating system provides applications with access to the underlying hardware. It does so by exporting a number of services. At the most basic level, these services are directly linked to some of the components of the hardware. For instance, there is a scheduling component that's responsible for

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.

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTime() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

15 - System Calls Quiz

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.

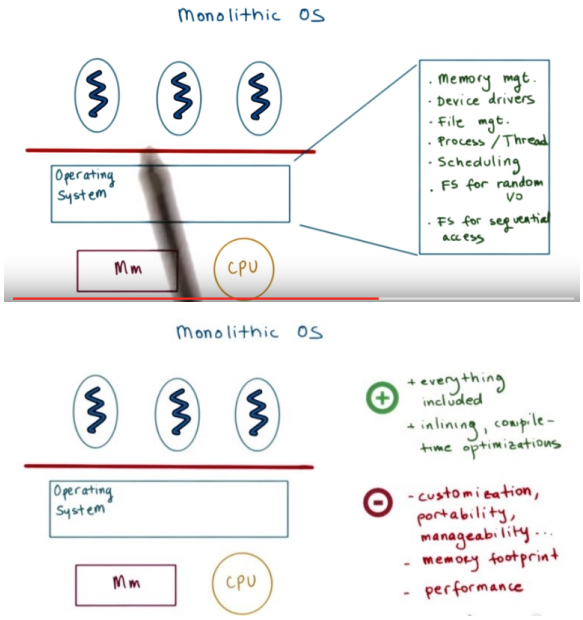


setpgid() vs. setgid()

The gid (group ID) is a security attribute of the process. The "process group ID" is an association of that process with other processes in the same group. Not a security concept.
See Section 10.2 at
http://www.win.tue.nl/~aeb/linux/lk/lk-10.html

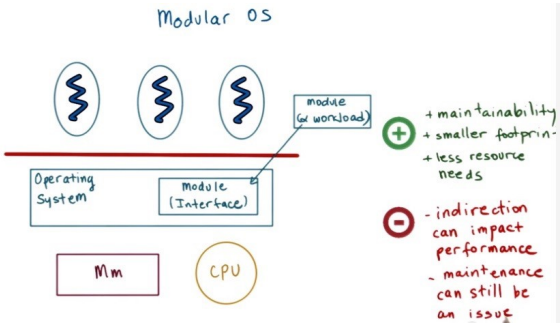
Every process is a member of a unique process group, identified by its process group ID. (When the process is created, it becomes a member of the process group of its parent.) By convention, the process group ID of a process group equals the process ID of the first member of the process group, called the process group leader. A process finds the ID of its process group using the system call getpgrp(), or, equivalently, getpgid(0). One finds the process group ID of process p using getpgid(p).

16 - Monolithic OS : contain every service/abstractions



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.

17 - Modular OS : contain basic services & EPI. everything could be added as modules

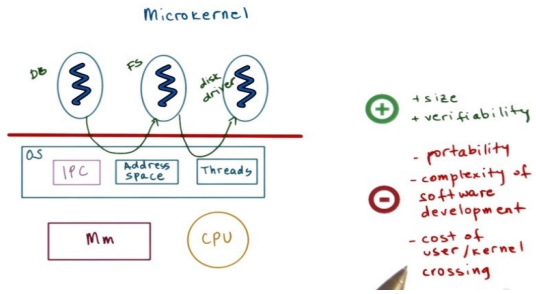


Overall Modular OS delivers significant improvements over Monolithic OS and is the one more commonly used today (linux).

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 EPIs 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 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 although 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.

18 - Microkernel



User level

DB=database FS=file system disk driver

System Level

IPC=InterProcess Communications => one of core abstractions/mechanisms (along with Address space and threads)

Microkernel is small but it's generally very customized, and makes software that uses it more complex.

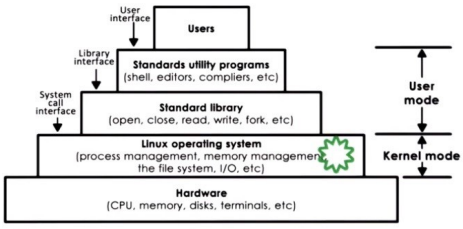
Examples of Microkernel: Systems where it is critical for the OS to behave properly. ie embedded devices and control systems

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

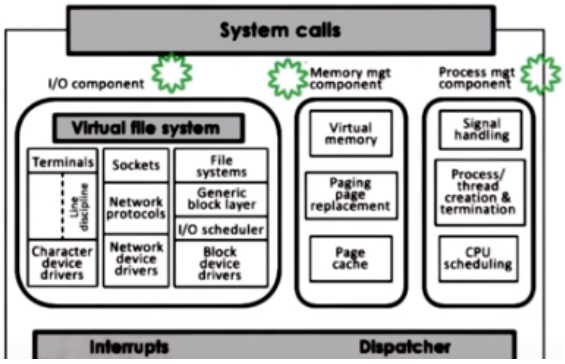
address spaces and threads. The benefits of a microkernel is that it's, it's very small. This can not only lead to lower overheads and better performance, but it may be very easy to, to verify, to test that the code exactly behaves as it should. And this makes microkernels valuable in some environments where it's particularly critical for the operating systems to behave properly, like some embedded devices or certain control systems. 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.

22 - Linux and Mac OS Architectures

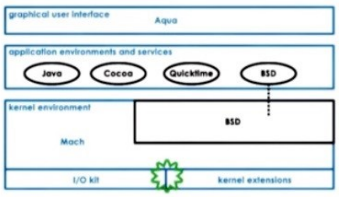
Linux Architecture



a **shell** is a user interface for access to an operating system's services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation.



Mac OS Architecture



Bsd component: provide unix interoperability via CLI, POSIX API support as well as network I/O. (The **Portable Operating System Interface** (**POSIX**)[1] is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems)

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.