The Structure of Operating Systems

Hello, and welcome to this lesson on the structure of operating systems. Until now we have mentioned the terms operating systems, monitors, kernels repeatedly, but everything may sound mysterious and appear quite amorphous.

So we will first look at a structure of an operating system from the perspective of an external observer.

We will see that the operating system has two interfaces, one for the user and the other for hardware, typically devices.

The user accesses operating system services through the system call interface, while devices interact with the system through device drivers. We will not spend too much time on both, as we will explore them in much more detail in separate lessons.

We will then look into the operating system and explore how it may be architected. We will look at layered architectures, which give rise to what some people call monolithic kernels.

We will look at two alternative architectures, the microkernel architecture and the combination of exokernels and library operating systems.

External Structure of an Operating System

Let's look at the operating system from far away. As we discussed in earlier lessons, the operating system finds itself between the hardware at the bottom and user programs, or user processes, at the top.

Between these two, here depicted in blue, is the core of the operating system, the so-called operating system kernel, which contains the various components that make for the operating system services, such as the CPU scheduler, the file system, the memory manager, and many others.

The kernel interacts with the underlying hardware using so-called device drivers, and with the user programs through the so-called system-call interface. Whenever a user programs requests a service from the operating system, it does so by making a so-called system call.

This is the operating system kernel in its entirety. We notice that there are no utility programs, no command-line interpreters or shells, no productivity software, no whatever else is typically packaged together with operating systems. When we talk about operating systems in this course, we talk strictly about the operating system kernel, here framed in blue.

Example: Windows

This very simple picture generally translates well to real systems. Let's look at the example of a recent version of Microsoft Windows.

At the bottom we have the hardware, and at the top different types of user programs, which can be native Windows applications, or POSIX compliant applications, which is a more UNIX/Linux like system call interface, or, finally, legacy applications that were developed for the nowadays defunct OS/2 operating system.

The applications are linked with libraries that translate any non-native system calls to appropriate Windows system calls.

These system calls request services provided by the kernel.

On the hardware side, Windows has a bit of a specialty, the Hardware Abstraction Layer, HAL in short, which is a thin layer that abstracts away hardware idiosyncrasies and in theory makes it easy to port windows to different hardware.

A number of device drivers interact with the HAL or with the hardware directly.

And finally we have all the components that provide the services, such as I/O, security, interprocess communication (IPC), virtual memory, and all the rest.

Internal Structure of an Operating System

Let's now peek into the kernel.

How are the components and services of the operating system kernel structured?

Internal Structure: Layered Services

A traditional way to think about operating system implementations is to use the onion as a layering abstraction. The operating system is an onion, - with the center being the hardware, and the - outermost layer the user programs.

We add layer from inside-out. - Layered directly on the hardware is the memory manager. All other components of the system rely on a memory manager. - Once we have a memory manager and allocator in place, in this particular system we can think about processes. This is mostly about the address spaces of the process. - Once we have processes, we need to be able to coordinate them. Hence, we need to add scheduling and synchronization. - With synchronization we can add interprocess communication facilities, such as pipes, shared memory, and so on. - Now may be a good time to add support for some sort of timers and clocks. - Once we have that, we can add devices through device drivers. - With the availability of devices, in particular networking devices, we can think about actual network protocols like TCP/IP and others. - We now can intelligently talk to other machines, but we don't have a file system yet. Let's add it.

And voila, we have a rather full-fledged operating system.

This particular layering represents the design of the XINU operating system, an instructional operating system from the Eighties. Other systems may be layered differently.

Internal Structure: Monolithic versus Microkernels

Let's draw the onion a bit differently, as a stack in fact, with the hardware at the bottom and the user programs at the top. The various components of the OS are now layered on top of each other, and as a result we have one big fat kernel, separated from the user programs in user space. We call this a layered kernel, or a hierarchically organized kernel.

We clearly see that this layering business leads to a kernel that is very large. Some people like to call these type of kernels monolithic kernels.

Let's look at a different way to structure kernels. At the bottom we still have the hardware, and at the top the user processes. Between hardware and user programs we have a very thin kernel, which has a very minimal functionality. Most of the components of the operating system are not in the kernel, but rather in separate user processes, in user space. Kernels that are architected in this way are called microkernels, and they typically have kernels that only supports core operating system functions, such as interrupt and exception handling, protection, and maybe naming. All other functionality of the kernel, such as device drivers, file system, process management, virtual memory, and so on, resides in separate server processes in user space, just like web servers or email servers.

Let's have a closer look at the architecture of a micro-kernel operating system. And let's replace the picture with a slightly simpler one

Operations in a Micro-Kernel

In this case we have a microkernel with a single service, the file server, and a single user process.

Like all the other non-core services, the file service is implemented as a server process in user space.

The interaction between user process and server uses messages, which are routed through the kernel.

In this example, the user process issues a file open system call.

The kernel creates an *open_file* request and routes it to the file system server. Presumably, the file system server, upon startup registered with the kernel to receive *open_file* requests, so the kernel now knows where to send them.

Once the file system server receives the *open_file* requests, it does whatever magic file system components do, and as result generates a file handle. This is returned to the kernel, which routes it back to the user process.

Some like to call this "client-server computing on a single computer".

Examples of microkernel operating systems are the venerable Mach research operating system, which started this all, and Chorus, or L4, or even the Windows NT family. As we will see in a minute, Windows does not operate as a microkernel, but uses the design principles of separating a microkernel from server components.

Client-Server Computing with Micro-Kernel

Let's investigate this "client server computing on a single computer" a bit further.

Let's split the computer into two, each with its own microkernel, and let the two be connected by a network, the Internet, say.

Since the user process and the server communicate using what amounts to messages, the user-side microkernel can forward the message to the server-side microkernel. The server-side microkernel does the same with the return message, which contains the file handle. Both user and server are unaware that the request was forwarded across a network. In fact, they are unaware even that they run on different computers.

Benefit of Micro-Kernels

What are the benefits of microkernels?

First extensibility. If you want the operating system to provide additional services, all you have to do is write another server program that provides this service. These servers are very easy to write because you do that in user space, without having to "hack the kernel" as some people call it.

Similarly, existing servers are easy to debug and customize or improve.

Microkernels are generally very easy to port. The L4 kernel, for example, has been ported to many different platforms. It is so easy to port that some system designers port Linux to new platforms by first porting it to run as a server on L4, and then they port L4 to the new platform. By moving L4, Linux is moved for free.

One other appreciated benefit is that the servers are isolated from each other. If the file system server crashes, for example, the rest of the system survives and can simply re-start the file system server. In this way it can continue as if nothing had happened.

Finally, as we illustrated just a minute ago, microkernels are ideally suited for distributing services across a network.

Micro-Kernels: Performance is a Problem

So, what is the problem with microkernels? Why is Linux, for example, not a microkernel?

The main problem of microkernels is their cost of system calls. As we learn in a separate lesson, system calls are very expensive because they are typically invoked using software interrupts or similar mechanisms that cause context switches.

In traditional kernels, the system incurs one context switch when the user process makes a system call, and another when the system call returns.

On microkernels, there is another pair of context switches, this time on the server side (on the right), one context switch when the server gets invoked, and another when it returns.

In microkernels, system calls therefore have twice the overhead than in traditional systems.

This is enough of a problem that people typically use one of two approaches to address this.

One approach is the move performance critical services back into the kernel. We call this the "make the kernel bigger" approach.

The other approach is to make the kernel even smaller, give give rise to nano kernels or exokernels. We will briefly describe these two approaches with an example of each.

Bigger: Windows

Microsoft is decidedly in the "bigger is better" camp.

Let's focus on the important components of the Windows architecture.

If we look closely, we see that the Windows architecture was clearly designed with microkernels in mind. There is even a component called so. In addition to the microkernel we have the object manager, which knows how to direct requests to the correct objects. If we combine the microkernel and the object manager, we get something like a traditional microkernel: Core operating system services plus messaging.

On top of this microkernel we have all the services that one would expect in a full fledged operating system, and they look conceptually like servers, except that we call them "managers". One could easily imagine that these managers could be user-level servers that run besides user programs.

Largely because of performance considerations, Microsoft decided to not put the managers into user space. Instead, they are all inside the kernel. Interaction between the microkernel layer and managers therefore is very light-weight. The Microsoft team has actually been quite aggressive in moving managers into the kernel.

One rather extreme example is the Window Manager, which is inside the kernel as well. This is very different from most other operating systems, where the window manager is run as a user-level server, such as X Server or Wayland or Mir for various Linux systems.

Having the windows server inside the kernel clearly improves performance. One important reasons against having it in the kernel is the need to have the overall system survive if the graphics system crashes. This argument does not work for typical windows users: Once the graphics system crashed, the user would make sure that the rest of the system crashed as well, by pushing the power button and so power-cycling the system. One could as well just have the graphics system be part of the kernel.

Smaller: Exokernel

One example of the "smaller is better" camp is Exokernel, a system designed in the mid Nineties. The idea of Exokernel was to largely eliminate the context switch overhead of what remained of system calls.

The premise of Exokernel was that traditional operating systems had it all wrong. It was wrong to try to have the operating system provide easy-to-use abstractions for hardware resources.

Instead, the kernel should export the hardware directly, without any abstraction layer. All a kernel had to do is to provide protection of the exported hardware resources.

Everything about resources is exported.

This includes their physical names such as addresses.

And applications can request and allocate specific resources. The management of the resources is therefore not done by the kernel, but by the application.

Exokernel: Secure Binding

Let's look at an example.

- As usual, at the bottom we have the hardware resources.
- At the top we have applications, for example a web application.

Let's focus on the web application first. We could run this application directly on the hardware if we linked all the necessary functionality to the application. In this case this would be some sort of an HTTP library, which traditionally is implemented as a user-level library.

Then we could have two more libraries, one the implements the entire functionality of a POSIX system, and the other that takes care of the networking. Because the last two libraries provide what amounts to operating system services, we call them operating system libraries. If we had such libraries, we could link all this together and run the application very efficiently, super optimized, on the bare machine, without an operating system at all. This works great as long as we have only one application to run.

Let's assume that we have a second application, this one does numerical computation.

Similarly to the first application, this may have some application libraries, in this example some user level distributed shared memory library.

The library operating system for this application is maybe specifically designed for numerical applications, and it supports inter-process-communication, exceptions, and virtual memory. Again, we could in principle link all this together and run it on the bare hardware, without need for an operating system.

But how would we run these two applications together? This is where the exokernel comes in. It provides so-called secure bindings.

For an application through its library OS of course, to get access to a hardware resource, it needs to request a binding. Once it has the binding, it can access the resource directly.

The principle of Exokernel is simple: The applications, through their library OSs, manage the hardware resources, and the exokernel protects the library OSs from each other.

Summary: Operating System Structure

We come to the conclusion of this lesson on operating systems structure.

We have learned how the operating system kernel sits between hardware resources and user processes. And how it provides two interfaces, the system call interface to users, and the device driver interf ace to hardware.

We then peered into the operating system. Because of their complexity, traditionally, operating systems have been designed using a layered approach, which led to what some people call “monolithic kernels”.

We explored alternative kernel architectures. A popular one is the microkernel architecture, where OS services are moved to user space where they are implemented as server programs. We discussed the pros and cons of microkernels, their main disadvantage being the cost of system calls.

One solution to reduce this cost is to grow the kernel again, and the other is to shrink it further. One example of the “small is better” school of thought is the exokernel, which protects resources and exports them directly to library operating systems, which manage them.

We sincerely hope that you enjoyed this lesson on operating system structures, and that you are in a good position to explore the various operating system services, while remaining aware that they can be implemented inside or outside of the kernel, or even in library operating systems.

Thank you for watching.