



## Operating Systems: Crash Course Computer Science #18

Crash Course: Computer Science

<https://youtube.com/watch?v=26QPDBe-NB8>

<https://nerdfighteria.info/v/26QPDBe-NB8>

This episode is supported by Hover.

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science! Computers in the 1940s and early 50s ran one program at a time.

A programmer would write one at their desk, for example, on punch cards. Then, they'd carry it to a room containing a room-sized computer, and hand it to a dedicated computer operator. That person would then feed the program into the computer when it was next available.

The computer would run it, spit out some output, and halt. This very manual process worked OK back when computers were slow, and running a program often took hours, days or even weeks. But, as we discussed last episode, computers became faster... and faster... and faster – exponentially so!

Pretty soon, having humans run around and inserting programs into readers was taking longer than running the actual programs themselves. We needed a way for computers to operate themselves, and so, operating systems were born. INTRO Operating systems, or OS'es for short, are just programs.

But, special privileges on the hardware let them run and manage other programs. They're typically the first one to start when a computer is turned on, and all subsequent programs are launched by the OS. They got their start in the 1950s, as computers became more widespread and more powerful.

The very first OSES augmented the mundane, manual task of loading programs by hand. Instead of being given one program at a time, computers could be given batches. When the computer was done with one, it would automatically and near-instantly start the next.

There was no downtime while someone scurried around an office to find the next program to run. This was called batch processing. While computers got faster, they also got cheaper.

So, they were popping up all over the world, especially in universities and government offices. Soon, people started sharing software. But there was a problem... In the era of one-off computers, like the Harvard Mark 1 or ENIAC, programmers only had to write code for that one single machine.

The processor, punch card readers, and printers were known and unchanging. But as computers became more widespread, their configurations were not always identical, like computers might have the same CPU, but not the same printer. This was a huge pain for programmers.

Not only did they have to worry about writing their program, but also how to interface with each and every model of printer, and all devices connected to a computer, what are called peripherals. Interfacing with early peripherals was very low level, requiring programmers to know intimate hardware details about each device. On top of that, programmers rarely had access to every model of a peripheral to test their code on.

So, they had to write code as best they could, often just by reading manuals, and hope it worked when shared. Things weren't exactly plug-and-play back then... more plug-and-pray. This was clearly terrible, so to make it easier for programmers, Operating Systems stepped in as intermediaries between software programs and hardware peripherals.

More specifically, they provided a software abstraction, through APIs, called device drivers. These allow programmers to talk to

common input and output hardware, or I/O for short, using standardized mechanisms. For example, programmers could call a function like "print highscore", and the OS would do the heavy lifting to get it onto paper.

By the end of the 1950s, computers had gotten so fast, they were often idle waiting for slow mechanical things, like printers and punch card readers. While programs were blocked on I/O, the expensive processor was just chillin'... not like a villain... you know, just relaxing. In the late 50's, the University of Manchester, in the UK, started work on a supercomputer called Atlas, one of the first in the world.

They knew it was going to be wicked fast, so they needed a way to make maximal use of the expensive machine. Their solution was a program called the Atlas Supervisor, finished in 1962. This operating system not only loaded programs automatically, like earlier batch systems, but could also run several at the same time on its single CPU.

It did this through clever scheduling. Let's say we have a game program running on Atlas, and we call the function "print highscore" which instructs Atlas to print the value of a variable named "highscore" onto paper to show our friends that we're the ultimate champion of virtual tiddlywinks. That function call is going to take a while, the equivalent of thousands of clock cycles, because mechanical printers are slow in comparison to electronic CPUs.

So instead of waiting for the I/O to finish, Atlas instead puts our program to sleep, then selects and runs another program that's waiting and ready to run. Eventually, the printer will report back to Atlas that it finished printing the value of "highscore". Atlas then marks our program as ready to go, and at some point, it will be scheduled to run again on the CPU, and continue onto the next line of code following the print statement.

In this way, Atlas could have one program running calculations on the CPU, while another was printing out data, and yet another reading in data from a punch tape. Atlas' engineers doubled down on this idea, and outfitted their computer with 4 paper tape readers, 4 paper tape punches, and up to 8 magnetic tape drives. This allowed many programs to be in progress all at once, sharing time on a single CPU.

This ability, enabled by the Operating System, is called multitasking. There's one big catch to having many programs running simultaneously on a single computer, though. Each one is going to need some memory, and we can't lose that program's data when we switch to another program.

The solution is to allocate each program its own block of memory. So, for example, let's say a computer has 10,000 memory locations in total. Program A might get allocated memory addresses 0 through 999, and Program B might get 1000 through 1999, and so on.

If a program asks for more memory, the operating system decides if it can grant that request, and if so, what memory block to allocate next. This flexibility is great, but introduces a quirk. It means that Program A could end up being allocated non-sequential blocks of memory, in say addresses 0 through 999, and 2000 through 2999.

And this is just a simple example - a real program might be allocated dozens of blocks scattered all over memory. As you might imagine, this would get really confusing for programmers to keep track of. Maybe there's a long list of sales data in memory that a program has to total up at the end of the day, but this list is stored across a bunch of different blocks of memory.



## Operating Systems: Crash Course Computer Science #18

Crash Course: Computer Science

<https://youtube.com/watch?v=26QPDBe-NB8>

<https://nerdfighteria.info/v/26QPDBe-NB8>

To hide this complexity, Operating Systems virtualize memory locations. With Virtual Memory, programs can assume their memory always starts at address 0, keeping things simple and consistent. However, the actual, physical location in computer memory is hidden and abstracted by the operating system.

Just a new level of abstraction. Let's take our example Program B, which has been allocated a block of memory from address 1000 to 1999. As far as Program B can tell, this appears to be a block from 0 to 999.

The OS and CPU handle the virtual-to-physical memory remapping automatically. So, if Program B requests memory location 42, it really ends up reading address 1042. This virtualization of memory addresses is even more useful for Program A, which in our example, has been allocated two blocks of memory that are separated from one another.

This too is invisible to Program A. As far as it can tell, it's been allocated a continuous block of 2000 addresses. When Program A reads memory address 999, that does coincidentally map to physical memory address 999.

But if Program A reads the very next value in memory, at address 1000, that gets mapped behind the scenes to physical memory address 2000. This mechanism allows programs to have flexible memory sizes, called dynamic memory allocation, that appear to be continuous to them. It simplifies everything and offers tremendous flexibility to the Operating System in running multiple programs simultaneously.

Another upside of allocating each program its own memory, is that they're better isolated from one another. So, if a buggy program goes awry, and starts writing gobbledygook, it can only trash its own memory, not that of other programs. This feature is called Memory Protection.

This is also really useful in protecting against malicious software, like viruses. For example, we generally don't want other programs to have the ability to read or modify the memory of, let say, our email, with that kind of access, malware could send emails on your behalf and maybe steal personal information. Not good!

Atlas had both virtual and protected memory. It was the first computer and OS to support these features! By the 1970s, computers were sufficiently fast and cheap.

Institutions like a university could buy a computer and let students use it. It was not only fast enough to run several programs at once, but also give several users simultaneous, interactive access. This was done through a terminal, which is a keyboard and screen that connects to a big computer, but doesn't contain any processing power itself.

A refrigerator-sized computer might have 50 terminals connected to it, allowing up to 50 users. Now operating systems had to handle not just multiple programs, but also multiple users. So that no one person could gobble up all of a computer's resources, operating systems were developed that offered time-sharing.

With time-sharing each individual user was only allowed to utilize a small fraction of the computer's processor, memory, and so on. Because computers are so fast, even getting just 1/50th of its resources was enough for individuals to complete many tasks. The most influential of early time-sharing Operating Systems was Multics, or Multiplexed Information and Computing Service, released in 1969.

Multics was the first major operating system designed to be secure

from the outset. Developers didn't want mischievous users accessing data they shouldn't, like students attempting to access the final exam on their professor's account. Features like this meant Multics was really complicated for its time, using around 1 Megabit of memory, which was a lot back then!

That might be half of a computer's memory, just to run the OS! Dennis Ritchie, one of the researchers working on Multics, once said: 9:18 "One of the obvious things that went wrong with Multics as a commercial success was just that it was sort of over-engineered in a sense. There was just too much in it." T his lead Dennis, and another Multics researcher, Ken Thompson, to strike out on their own and build a new, lean operating system... called Unix.

They wanted to separate the OS into two parts: First was the core functionality of the OS, things like memory management, multitasking, and dealing with I/O, which is called the kernel. The second part was a wide array of useful tools that came bundled with, but not part of the kernel, things like programs and libraries. Building a compact, lean kernel meant intentionally leaving some functionality out.

Tom Van Vleck, another Multics developer, recalled: "I remarked to Dennis that easily half the code I was writing in Multics was error recovery code." He said, "We left all that stuff out of Unix. If there's an error, we have this routine called panic, and when it is called, the machine crashes, and you holler down the hall, 'Hey, reboot it.'"" You might have heard of kernel panics, This is where the term came from. It's literally when the kernel crashes, has no recourse to recover, and so calls a function called "panic".

Originally, all it did was print the word "panic" and then enter an infinite loop. This simplicity meant that Unix could be run on cheaper and more diverse hardware, making it popular inside Bell Labs, where Dennis and Ken worked. As more developers started using Unix to build and run their own programs, the number of contributed tools grew.

Soon after its release in 1971, it gained compilers for different programming languages and even a word processor, quickly making it one of the most popular OSes of the 1970s and 80s. At the same time, by the early 1980s, the cost of a basic computer had fallen to the point where individual people could afford one, called a personal or home computer. These were much simpler than the big mainframes found at universities, corporations, and governments.

So, their operating systems had to be equally simple. For example, Microsoft's Disk Operating System, or MS-DOS, was just 160 kilobytes, allowing it to fit, as the name suggests, onto a single disk. First released in 1981, it became the most popular OS for early home computers, even though it lacked multitasking and protected memory.

This meant that programs could, and would, regularly crash the system. While annoying, it was an acceptable trade-off, as users could just turn their own computers off and on again! Even early versions of Windows, first released by Microsoft in 1985 and which dominated the OS scene throughout the 1990s, lacked strong memory protection.

When programs misbehaved, you could get the blue screen of death, a sign that a program had crashed so badly that it took down the whole operating system. Luckily, newer versions of Windows have better protections and usually don't crash that often. Today, computers run modern operating systems, like Mac OS X, Windows 10, Linux, iOS and Android.

Even though the computers we own are most often used by just a



## Operating Systems: Crash Course Computer Science #18

Crash Course: Computer Science

<https://youtube.com/watch?v=26QPDBe-NB8>

<https://nerdfighteria.info/v/26QPDBe-NB8>

---

single person, you! their OSes all have multitasking and virtual and protected memory. So, they can run many programs at once: you can watch YouTube in your web browser, edit a photo in Photoshop, play music in Spotify and sync Dropbox all at the same time. This wouldn't be possible without those decades of research and development on Operating Systems, and of course the proper memory to store those programs.

Which we'll get to next week. I'd like to thank Hover for sponsoring this episode. Hover is a service that helps you buy and manage domain names.

Hover has over 400 domain extensions to end your domain with - including .com and .net. You can also get unique domains that are more professional than a generic address. Here at Crash Course, we'd get the domain name "mongols.fans" but I think you know that already.

Once you have your domain, you can set up your custom email to forward to your existing email address -- including Outlook or Gmail or whatever you already use. With Hover, you can get a custom domain and email address for 10% off. Go to [Hover.com/crashcourse](https://Hover.com/crashcourse) today to create your custom domain and help support our show!