CS162 Note 4: A Timeline of OS

Kevin Moy

Abstract

In the previous note, we covered what characteristics we want to see in an ideal operating system. We recognized there were tradeoffs. This note will discuss the origins of operating systems as well as where their future.

1 The Beginnings

It is clear, through Moore's Law and just pure observation over time, that exponential improvment has occurred in areas such as processor speed, memory, bandwidth, etc. These relative changes have radically altered both the use of computers and the tradeoffs faced by operating system designers.

Things used to be pretty shit back in the early days. Computers were more expensive than most people's salaries. Programs took *days* to execute. But over time, improvements and mass production made computers much more cheaper and affordable. We can also have one for every person now.

However, the operating system still faces the same responsibilities as it did 50 years ago, the responsibilities we hopefully are very familiar with now: resource allocation, fault isolation, standard services, hardware abstraction, etc. The world has made unbelievably large improvements in the OS over these fifty years. But we still have work to do.

2 Early Operating Systems

The very first operating systems were just as you expect: warehouse-sized, cost millions, and only a single user at a time. Their only purpose was to serve as *runtime libraries*, used to simplify programming. To use it, one had to first reset their computer, load the program *one bit at a time* (god damn), and execute. This was a day-long process and so the user better have made sure no bugs were in their program!

Computers were incredibly expensive back then, so it was essential to reduce programmer bugs as much as possible. That's exactly the purpose these primitive operating systems served: providing these common services/routines that were error-proofed. These services made it more likely that a user's program would produce useful output.

The problem was these initial operating systems were extremely inefficient-basically slow as shit.

3 Multi-User

The next huge evolution came with **sharing**. Remember the first operating systems could only support one user at a time. This was wasteful. For example, the loading of a program (BIT BY BIT) took a lot of time- meaning a lot of time the processor sat around doing *nothing* while other people were waiting.

A **batch operating system** allows for a queue of tasks. It'll execute each in sequence. While one job runs, the OS "prepares" the next job via **direct memory access** (DMA). The big idea with DMA is that certain **I/O devices** transfers their data directly into some memory location specified by the OS. Then after this transfer completes, the processor is interrupted so it can grab the data. Then the OS starts the next transfer and resumes execution of the current task. More on how this is relevant later.

Operating systems then extended to implement **multitasking**. Multiple programs are loaded at once, and the processor simply has other stuff to do in the case of a stall on one program. If the task queue is long enough and the number of IO devices are numerous enough, there shouldn't be *any* idle time for the processor.

There was an issue: debugging. Batch operating systems assumed direct control of hardware. So we could only debug if we stopped all other applications and rebooted- basically we had to revert to single-user. Too expensive!

The solution came with **virtual machines**: treating operating systems as (debuggable) applications. VMs are now widely used for operating system development, backwards compatibility, and cross-platform support. The virtual machine monitor runs two virtual machines — one for new OS apps, and another for legacy apps. For example, Windows can run Mac programs inside VMs.

4 Time Sharing

Time-sharing OS (Windows, MacOS, Linux) gave computers the interactive design we are familiar with today instead of the one-at-a-time design of batch OS. In time-sharing systems, user input comes through some input device (keyboard, mouse) which would *interrupt* the processor and proceed with subsequent interrupt handling. There is still a queue of events, but they can certainly be done concurrently, with up to thousands of events being processable per second.

Our web server is similar! Web server waits for GET request packet (in memory). Network uses DMA to copy packet into memory. Interrupt processor, handle packet. Like a time-sharing system, the server has to handle many short actions per second.

5 Modern Operating Systems

Let's look at some of the various kinds of modern operating systems today.

5.1 Desktop/Laptop OS

These are the operating systems you are (probably) most familiar with: Windows, MacOS, Linux. Single user, multi-program, with many IO devices (screen,keyboard,mouse,etc.). Not really much else to say about this- y'all already know what's up.

5.2 Smartphone OS

Next up are **smartphones**: phones but with an embedded computer chip that can run apps. This is also something we are incredibly familiar with, probably even more so than desktops and laptops; examples we know are iOS and Android and Windows Phone (there are others but only if you're a weirdo). Again, these are single-user systems, but must support many third-party apps.

One thing to note is because smartphones are smaller in general, aspects such as battery usage and user privacy are emphasized more.

5.3 Server OS

Now we reach into the realm of Google and Gmail. These servers are hosted on *data center* computers, where each server computer has its own OS which must be stronger and faster than normal operating systems. Each machine only serves one purpose (thus runs one application), but its OS has to deal with thousands, maybe millions of simultaneous requests. So the OS gotta be tough: maximizing throughput (req handled per second) is key.

It's important to note that servers also are attacked a lot more than other OS (EX: DDOS), so it goes without saying that resistance to hackers and bugs is *crucial*, especially since so many people rely on a server's functionality.

5.4 Virtual Machines

Virtual Machine Monitors are the operating systems that can run application-operating-systems (*guest* OS). Common examples you might know are VMWare and VirtualBox. These VMMs have to not only handle standard OS duties but also coordinating it to the guest OS, who behaves as if it had complete control of resource (even though it's really sharing it with the host OS).

Why use VMs? One purpose is to allow a single servers to run a set of independent services. So we can configure a bunch of (unrelated) web servers to share the same physical hardware (but are still independent). Thus we want to maximize efficiency and minimize overhead with virtual machine design.

5.5 Server Clusters

Server clusters are basically a group of server computers clustered in one or more geographically distributed data centers. This setup provides fault tolerance: if one computer fucks up, another takes over. It also helps with responsiveness: requests can be distributed among multiple machines.

Just like with normal operating systems, server cluster applications are abstracted from hardware, fault-isolated, and can share resources.

6 Future Operating Systems

So operating systems have made tremendous progress since they were invented, there's no question.

But there's also no question that there's much room for improvement. We rely on computer systems for tasks that involve coordination and control for things like telephone networks and emergency response systems. We'd see massive benefits in life itself if we could get operating systems up to the task of replacing human operators.

Evolution in hardware will also trigger changes in OS design standards.