1 Introduction: What is an Operating System?

This is an Operating Systems course, but what actually is an operating system? I’m sure you’ve had spirited religious debates over the advantages of Mac OS X vs. Windows vs. Linux, etc. but what are these various Operating Systems actually doing? This is a tricky question because you’ve all grown up with PCs running fairly sophisticated OSs, chock full of all kinds of graphical user-friendly widgetry that conceals the true purpose of the OS. But, the real purpose of an OS is fairly simple: an OS provides a consistent VIRTUAL machine for programmers, so that application developers don’t have to worry about specific hardware details. That’s it, the main function of an OS is to HIDE things from the programmer. At first, that might sound a little insulting, ‘why is it that the OS guys think that I can’t handle what they’re hiding from me?’, you might say. But think about it this way, when have you ever written a program that needed to know which hard drive contained the files you were working on? When did you ever write some code that needed to know exactly which sound hardware your computer had? ¹

Operating Systems spare application developers from worrying about hardware specifics. Filesystems hide details about the size and number of hard drives, DVD-ROM drives, etc. Processes and threads hide details about the size and layout of system memory, and APIs like OpenGL, Quartz, DirectX, Xlib and ALSA hide details about graphics and sound hardware. As a programmer, you can then code to the OS interface and trust that it will handle all the fiddly details for you (and usually better than you could do yourself). In this course, you’re going to learn about all the complex machinery and engineering that goes into maintaining this virtual machine on top of real hardware.

A common theme in Operating Systems is resource management. The hardware of a computer system represents a known, finite, and usually fixed set of hardware resources. It is then the job of the OS to virtualize and multiplex these resources among many running programs. As an example, consider memory. My laptop has 1GB of physical memory in it, but as a 32-bit machine it’s capable of addressing 4GB. Using a combination of hardware (virtual memory translation) and software (the memory manager) the OS virtualizes this 1GB of physical memory into 4GB of address space. This memory is also a resource, the OS has to share this memory around between many running programs and orchestrate it so that each program ‘thinks’ that it has full, unfettered access to the machine. As another example, consider hard disks and filesystems. The filesystem is the OSs way of virtualizing disk storage. Users and programs don’t need to concern themselves with exactly where the files are stored on disk, or on exactly which disk they are store (except for windows, which still presents users with volume information (this, by the way, is a bad idea and a throwback to DOS)). On the resource management end, each hard disk can really only be reading or writing to one thing at a time, so the OS has to deal with the case where two programs are accessing files that happen to be stored on the same disk while still preserving the illusion that each program has complete and total access to the filesystem.

¹This last example is actually from the dark days of MS-DOS, when sound wasn’t supported in the OS, so application developers had to write different code for different sound hardware (usually soundblaster or adlib)! This meant that the user not only had to know which sound card was installed in the machine, but specific hardware details (such as IRQ and DMA channels).
An operating system is an engineered system, and as such has multiple requirements, namely:

1. **Convenience:** an OS must make the machine easier to use
2. **Efficiency:** the OS must efficiently utilize the hardware resources of the machine
3. **Evolvability:** an OS must be able to change to meet the diverse requirements of the user base, and the OS must be able to be upgraded and patched

Of course, as in most engineering, trade-offs have to be made between these three.

# 2 OS Features: A Historical Perspective

*Chap. 2.2.*

In the embryonic days of Computer Science, there were no Operating Systems. Computers were giant expensive machines, filling rooms and possessing far less computing power than a garden-variety digital watch. Each computer was totally unique, and programmers weren’t likely to encounter more than one with any regularity, so the idea of writing a program for a specific set of hardware wasn’t odd. As people started writing more and more code, impromptu libraries of useful code were assembled. This code later evolved into the first style of OSes, the monitor. Although primitive, the monitor established the fundamental function of an OS, which is a tool to aid in the development and running of programs.

## 2.1 Serial Batch Processing

In the late 50s and early 60s, computers were large and finicky, overseen by a priesthood of system operators who tended the giant machines. Anyone who actually wanted to use the computer had to write their program onto some kind of media (paper cards or tape were popular choices), and then hand their code to one of the administrators to run. In this scenario, an Operating System was just a pile of code that ran before the user’s code. It got everything set up to some kind of default initial state and then jumped into the user’s code. A conscientious user would then jump back into the monitor code when their code was done. This actually made very poor use of the machine, since only one program could run at once, unused parts of the machine couldn’t be used by another program concurrently. Even in this limited context, some OS principles were used. These batch systems needed some way to protect parts of memory, so that the user’s program couldn’t overwrite/corrupt the OS code. It was also helpful to make a distinction between user and OS (usually called supervisor) instructions. If the user’s code could access the hardware directly, then the OS couldn’t ensure that I/O devices (such as a disk or tape drive) weren’t corrupted.

**Virtualized Resources:** Memory and I/O
2.2 Multiprogrammed Batch Processing

Early on, people realized that a computer could run several programs at once. Ideally, this means that if one program is waiting for some data to arrive from disk, the processor could be executing code from another program. Which would mean less real time is spent running the two jobs than if you just ran them back-to-back. This is important if you’re charging people for computer time. Economically speaking, the more programs you can run (and bill people for) in a given amount of time, the more money you make. Of course, running two programs at the same time is way more than twice as hard as running one program. Now, the OS has to protect each program from the others, which required more sophisticated memory management than just protecting the OS. And because each hardware device can really only do one thing at a time, the OS has to be able to swap running programs. The OS developer has to solve a tricky problem, namely: how do you safely ‘stop’ a running program and safely start it back up again (when the processor becomes free). This, by the way, is what processes are all about. Instead of programs, an OS thinks in terms of processes that it can stop, swap out with another process and restart at some later time. This whole can of worms is called context switching (and it can be very complicated). Of course, the fun doesn’t stop there. Even if you’ve solved the context switching problem, now you have to worry about choosing between several suspended processes. Which one do you run next? For how long? This problem is known as process (or thread) scheduling, and it’s an active area of research. Note that all of this jiggery-pokery under the hood of the OS is to preserve the illusion that each program has the entire machine to itself. Giving each process a separate virtual machine simplifies programming greatly. Coders don’t have to worry about what other programs are doing, or if there’s enough memory on the machine. The OS takes care of it all (most of the time, that is).

Virtualized Resources: Memory, I/O, and processor time
Figure 2: Multiprogrammed Batch Processing Example Schedule 1

Figure 3: Multiprogrammed Batch Processing Example Schedule 2
2.3 Time Sharing

Multiprogrammed batch machines were still batch machines. That is, programmers didn’t have direct access. They had to hand their shoebox full of punch cards to some guy to run through the machine. By the late 60’s, programmers really wanted to have their own machine, but computers were still far too expensive for each person to actually have a separate machine. Time sharing was invented to solve this problem. Time sharing is a way of partitioning time between multiple users (also called ‘time-slicing’). Because processors are so much faster than humans, a computer can actually service multiple human requests in turn and return to a person without them ever noticing that the computer wasn’t actually running their process at that moment. In this model, users would interact with the computer through a shell (a.k.a. the command line). The user would use shell commands to start and stop processes (things like editors and compilers). From the OS’s point of view, the shell was just another process to multiplex onto the CPU and memory. Most modern PC OS’s are time shared systems. The OS switches many dozens to hundreds of times a second, so that humans using the system won’t notice any lag. Modern OS’s also utilize paged virtual memory which is a way of finely partitioning memory so that many programs, large and small, can be packed tightly into a fixed-size physical memory.

**Virtualized Resources:** Memory, I/O, Proc. time

![Time Sharing Example Schedule](image)

Note that the history of OSes on the desktop follows a similar trend. Early systems (MS-DOS, CP/M, MacOS 1) were all serial systems with no memory protection. As PCs became more sophisticated, features from time-sharing systems were added. Memory protection was usually first, because it let the user have multiple programs in memory (they usually weren’t executing...
concurrently, though), and then ‘multi-tasking’ (which is actually time-slicing), until now where modern OSes are chock full of all the features found in multi-user networked OSes from the late 70s and early 80s.