What is this Book on Operating Systems?

Peter J. Denning

5/25/19

Operating systems are among the largest systems engineered by humans; they rival modern aircraft and ships in complexity. They are a gigantic market – over $37B annual sales worldwide, not counting open source OS related projects. Commercial OSs approach 100 million lines of code, assembled and maintained by enormous teams of 10,000 or more programmers. OSs have to reliably control events of enormous time-variability, from user interactions (minutes, days, years) all the way down to clock ticks and hardware instructions (picoseconds). Amid all this complexity, OS vulnerabilities are common – they enable more than $600B a year of cybercrimes such as espionage, data exploitation, ransomware, malware invasion, and identity theft.

And yet, despite all the complexity and vulnerabilities, there is a reason that OSs work as well as they do: they all are built around a relatively small *kernel*, a set of software routines implementing the core functions that all applications use. Over the years, we have learned how to build reliable kernels and as a result our operating systems are remarkably reliable despite their vulnerabilities.

Although OS kernels are complex, they are based on a well understood set of basic principles. If you really understand those principles, you can design remarkably small, highly secure kernels. If you don't understand those principles, your kernel can turn into an ugly mass of spaghetti.

This set of slides is designed to help you understand the basic principles used by OS kernels. What functions does the kernel perform? Why are those functions there? How can you use them?

My approach is to present a single model for each OS kernel function and show you the kernel as a set of levels layered on one another. By keeping the individual levels simple, I can show you how the whole thing fits together into a working kernel. Many of the existing textbooks go into great detail about many possible ways to implement each function, leaving you in a state of confusion about which one is best and how the whole mass fits together. Instead, I am aiming to show you how the fundamental principles lead to a compact, simple design of a kernel. You can then go on to read one of the encyclopedic OS books and have a much better chance to understand what is going on.

Why do it this way? OSs are complex systems with low level events happening at intervals of 10^{-9} or shorter seconds, and user level events happening at intervals of 10^{3} seconds or more. That is a span of over 10^{12} orders of magnitude in the pace of events. Few human-designed systems encompass such a span. Because the OS is all software resting on top of silicon chips, it fits into a small space – a few chips a few centimeters on a side. Nonetheless, getting all those levels of software to work together, perform correctly, without freezing or hanging up or succumbing to

vulnerabilities, is a monumental task. If you are a newcomer to the field, the complexity of OSs produced by thousands of programmers over several years will stagger you. You do not have to be staggered. I have written this for beginners.

You can think of this as a map of the principles of operating systems. It is not a map of any particular operating system. It is an idealized kernel composed of simple models.

As we move through the kernel, the separate file *OS_Levels.pdf* will be your map. Consult it frequently to keep the big picture in mind.

History of Operating Systems

The first operating systems were built in the 1950s. By 1960 the research labs building them (mostly at universities) had demonstrated time-sharing (automatic process multiplexing), semaphores (synchronization of processes), virtual memory (automatic management of main memory), multiprogramming, hierarchical file systems, access control lists for files, and hardware support for interrupts, address mapping, memory protection, shell, schedulers, and sharing of data between memory partitions. These ideas have stood the test of time and pervade modern operating systems. I have included the essay "Fifty years of operating systems" to give you a fuller picture of the origins and significance of these innovations.

Many concepts in operating systems are used throughout computer science, but are not part of ordinary programming. One example is the distinction between "program" and "process". A program is a segment of code that, when executed on a CPU, transforms input data to output data in a specified way. A process is a program in execution on a virtual machine. (A virtual machine is a simulated replica of a real machine.) Operating systems were intended to manage the dynamics of many programs operating on behalf of many simultaneous users; the process (not the program) is the entity the OS manages.

Another example is the idea that the operating system is built from a collection on ongoing, nonterminating processes. It is not simply a library of system programs that you call when need. Most of the processes are designed to render a specific service that you can request at any time. If you open up a "process control panel" on your computer, you will typically see 100-500 processes listed, even before you have started any applications. That is quite a society.

With such a population of processes all vying for limited resources such as CPU time, disk access, and memory space, some rules of order are needed to maintain harmony and prevent chaos. For example, processes request service from other processes by sending them messages and then stopping to wait for an answer. A service process waits at its "homing position" until a request message arrives, then it serves the request, returns results to the requestor, and returns to its homing position. A message passing system is an essential element of maintaining order in the society of processes.

Another example is that processes cannot see the data in the private memories of other processes. Data are stored in memory partitions accessible only to their owners. The OS hardware and software prevents any other process from accessing that region of memory. This rule of order guarantees that no process can be suddenly surprised by some else changing its data.

In general, the kernel embodies all the basic rules to enable a society of processes to work effectively.

Our Scope

The following questions and their answers define the scope of our conversations.

What is an OS?

- Control program to safely allocate resources among competing users
- Environment for program construction and execution
- Environment for communication and coordination with Internet
- Interface to many services (built in and remote)
- Maintenance of personal workspaces and data for many users

What are the main goals of an OS?

- Allocating resource units to users and switching them quickly [Multiplexing]
- Isolating users -- actions of one cannot interfere with others [Partitioning]
- Controlling access to and sharing of digital objects [Protection]
- Facilitating work and programming [Working Environment]
- Maintaining speed, small response time, low energy, stability [Performance]

How does OS differ from other parts of CS?

- Complex system of many subsystems and networks
- Society of cooperating processes
- Many limited resources serving many more users
- Concurrency always a central issue
- Ongoing computations rather than terminating algorithms
- Protection of information
- Constantly under attack

What we **do** study here:

- Kernel only (innermost core used by every user and app)
- Shell (command interface to kernel)
- Principles of kernel operation
- Various problems which drive kernel design and implementation choices
- Trade-offs between design choices

What we **do not** study here:

- Services outside the kernel (built in or Internet; the vast majority of OS bulk)
- How specific operating systems work (Unix, Windows, etc)
- Systems programming
- Systems configurations
- System performance evaluation