Chapter 3

Classical Operating Systems and the Kernel

This course is about Computer Systems, and specifically systems software. It includes material that would traditionally be split between a course on operating systems, and a different one on distributed systems.

Treating these as a single topic is, we think, a much more accurate picture of reality. A good illustration of this is a basic definition of what an operating system actually is.

3.1 The role of the OS

Anderson and Dahlin [AD14], roughly speaking, define an OS functionally:

\[ \text{Definition 3.1 (Operating System).} \quad \text{The operating system for a unit of computing hardware is that part of the software running on the machine which fulfils three particular roles: Referee, Illusionist, and Glue.} \]

- As a Referee, the OS multiplexes the hardware of the machine (CPU cores, memory, devices, etc.) among different principals (users, programs, or something else), and protects these principals from each other: preventing them from reading or writing each others’ data (memory, files, etc.) and using each others’ resources (CPU time, space, memory, etc.). The OS is basically a resource manager whose goal is to provide some kind of fairness, efficiency, and/or predictability.

- As an Illusionist, the OS provides the illusion of “real” hardware resources to resource principals through virtualization. These virtual resources may resemble the underlying physical resources, but frequently look very different.

- Finally, as Glue, the OS provides abstractions to tie different resources together, communications functionality to apply principals to exchange data with each other and synchronise, and hide details of the hardware to allow programs portability across different platforms.

Remarks:

- Most older OS textbooks (which are not recommended for this course) do claim that the OS essentially provides multiplexing, protection, communication, and abstraction, but then go on to define it ontologically as a kernel, libraries, and daemons – we look at these components below, but only in the context of an OS for a single, simplified machine.

- Note, however, that this definition is pretty broad in what the hardware actually is. It might be a small Raspberry Pi, or a robot, or a phone, laptop, a tightly coupled cluster of PC-based servers in a rack, or an entire datacenter – the definition still makes sense in all these cases. That’s why we’re not separating operating systems and distributed systems in this course.

- In this course, we’ll cover many of these scales. They have different properties; for example, whether part of the machine can fail when the rest keeps going (as in a datacenter or rackscale machine) or it all fails together (as in a phone or PC). A good way to make sense of this is the concept of a domain.

3.2 Domains

Domains are useful concepts for thinking about computing systems at different scales.

Definition 3.2 (Domain). A domain is a collection of resources or principals which can be treated as a single uniform unit with respect to some particular property (depending on the domain). Some examples:

- NUMA domain: cores sharing a group of memory controllers in a NUMA system.
- Coherence domain or coherence island: caches which maintain coherence between themselves.
- Failure domain or unit of failure: the collection of computing resources which are assumed to fail together.
- Shared memory domain: cores which share the same physical address space (not necessarily cache coherent)
- Administrative domain: resources which are all managed by a single central authority or organization.
- Trust domain: a collection of resources which mutually trust each other.
- Protection domain: the set of objects which are all accessible to a particular security principal.
- Scheduling domain: set of processes or threads which are scheduled as a single unit.
Remarks:

- Each topic we will cover is going to be tied to some combination of domain boundaries. Sometimes we’ll mention which ones, but it’s always good to figure out which ones are in play.
- Domains often correspond to the assumptions the software can make inside a domain, or the functionality the software must provide between domains. Domains also often can be expressed as invariants or uniform properties of the units inside each domain.

3.3 OS components

A classical mainstream OS (like UNIX or Windows) operates on a single machine. That’s a single coherence domain, a single unit of failure, a single administrative domain (hence hierarchical authority), and a single resource allocation domain (centralized resource control).

Such an OS consists of the kernel, libraries, and daemons.

Definition 3.3 (Kernel). The kernel is that part of an OS which executes in privileged mode. Historically, it has also been called the nucleus, the nub, the supervisor, and a variety of other names.

Remarks:

- The kernel is a large part of OS in UNIX and Windows, less so in L4, Barrelfish, etc.
- While most computer systems have a kernel, very small embedded systems do not.
- The kernel is just a (special) computer program, typically an event-driven server. It responds to multiple entry points: system calls, hardware interrupts, program traps. It can also have its own long-running threads, but not always.

Definition 3.4 (System libraries). System Libraries are libraries which are there to support all programs running on the system, performing either common low-level functions or providing a clean interface to the kernel and daemons.

Remarks:

- The standard C library is a good example: it provides convenience functions like `strncpy()`, essential facilities like `malloc()`, and interfaces to the kernel (system calls like `open()` and `sbrk()`).

Definition 3.5 (Daemon). A daemon is a user-space process running as part of the operating system.

Remarks:

- Daemons are different from in-kernel threads. They execute OS functionality that can’t be in a library (since it encapsulates some privilege), but is better off outside the kernel (for reasons of modularity, fault tolerance, and ease of scheduling).

3.4 Operating System models

There are a number of different architectural models for operating systems. They all “ideals”: any real OS is at best an approximation. Nevertheless, they are useful for classifying different OS designs.

Definition 3.6 (Monolithic kernel). A monolithic kernel-based OS implements most of the operating systems functionality inside the kernel.

Remarks:

- This the best-known model, and UNIX adheres closely to it.
- It can be efficient, since almost all the runs in a single, privileged address space.
- Containing faults (software or hardware bugs) in a monolithic kernel is hard. Recent evidence suggests that this does result in reduced reliability.
- Monolithic kernels tend to have many threads which execute entirely in kernel mode, in addition to user-space application processes and daemons.

Definition 3.7 (Microkernel). A microkernel-based OS implements minimal functionality in the kernel, typically only memory protection, context switching, and inter-process communication. All other OS functionality is moved out into user-space server processes.

Remarks:

- Microkernels use process boundaries to modularize OS functionality. Device drivers, file systems, pages, all run as separate processes which communicate with each other and applications.
- The motivation for microkernels is to make the OS more robust to bugs and failures, and make it easier to structure and evolve the OS over time since dependencies between components are in theory more controlled.
- Microkernels can be slower since more kernel-mode transitions are needed to achieve any particular result, increasing overhead. However, the very small size of microkernels can actually improve performance due to much better cache locality. The debate remains controversial.
- Examples of contemporary microkernels include Minix and the L4 family. There is a myth that microkernels have never been successful. If you use a phone with a Qualcomm processor, you’re running L4. If you use an Intel machine with a management engine, you’re running Minix. That’s well over a billion deployed units right there.

Definition 3.8 (Exokernel). In contrast to microkernels, exokernel-based systems move as much functionality as possible out of the kernel into system libraries linked into each application.
3.5 Bootstrap

Remarks:

• Moving OS functionality into application-linked libraries is, at first sight, an odd idea, but greatly simplifies reasoning about security in the system and providing performance guarantees to applications which also need to invoke OS functionality.

• Exokernel designs came out of academic research (MIT and Cambridge), but found their biggest impact in virtual machine monitors: VMware ESX Server and Xen are both examples of exokernels.

Definition 3.9 (Multikernel). A multikernel-based system targets multiprocessor machines, and runs different kernels (or different copies of the same kernel) on different cores in the system. Each kernel can be structured as a monolithic, micro- or exo-kernel.

Remarks:

• Multikernels are a relatively new idea (although, if you look back in history, several old systems look like this as well).

• The individual kernels can be anything (or a mixture): people have built multikernels using exokernels (e.g. Barrenfish) or complete monolithic systems (e.g. Popcorn Linux). The key characteristic is the kernels themselves do not share memory or state, but communicate via messages.

• They reflect modern trends in hardware: heterogeneous processors (making a single kernel program impossible), increasingly networked interconnects (meaning programmers already worry about inter-core communication explicitly), and increasingly low-latency inter-machine networks (in other words, we increasingly need to consider a rack of servers or an entire datacenter as a single machine). Indeed, a theme of this course is that the boundary between "operating systems" and "distributed systems", always problematic, is now basically meaningless. They are the same topic.

3.5 Bootstrap

Definition 3.10 (Bootstrap). Bootstrapping, or more commonly these days simply booting, is the process of starting the operating system when the machine is powered on or reset up to the point where it is running regular processes.

Remarks:

• When a processor is reset or first powered on, it starts executing instructions at a fixed address in memory. A computer design must ensure that the memory at this address holds the right code to start the machine up in a controlled manner.

• While this code could be the initializing code for the kernel, in practice it is another program which sets up the hardware. This can be highly complex process for two reasons: firstly, modern processors and memory systems are incredibly complicated these days, and there is a lot to do (potentially hundreds of thousands of lines of code) to execute before the OS itself can be loaded. This even includes starting memory controllers: this initialization code often runs only out of cache and initializes the DRAM controllers itself. This code is sometimes called the Basic Input/Output System (BIOS), or its functionality is standardized as the Unified Extensible Firmware Interface (UEFI). The BIOS is generally built into the machine as Read-Only Memory (ROM).

• The BIOS also sets up a standard execution environment for the next program to run, so that it doesn’t need to know at compile time the precise devices that the computer has (storage devices, network interfaces, how much memory, etc.). This next program can therefore be the same across a wide range of computers.

• The next program is typically the boot loader, and its job is to find the operating system kernel itself, load it into memory, and start it executing.

• The OS kernel itself, once it is entered, initializes its own data structures and creates the first process (known as init in traditional Unix). Finally, it starts this new process executing, and the system is now in a regular steady state.

3.6 Entering and leaving the kernel

Definition 3.11 (Mode transfer). Mode transfer is the process of software execution transitioning between different hardware processor modes.

Remarks:

• This typically involves switching between user mode (running regular programs and daemons) and kernel mode (running the kernel). However, other modes are possible (such as with virtual machine support).

• The key goal of user to kernel mode transfer is to protect the kernel from a malicious or buggy user process.

• The kernel is entered from userspace as a result of a processor exception: either a synchronous trap or an asynchronous fault (as we saw in the Systems Programming course).

• Mode transfer to the kernel cannot be done via any kind of jump instruction: we cannot allow a user program to start executing at an arbitrary point in the kernel. Instead, as with faults, when a user program executes a system call the processor starts executing in the kernel at a fixed address, and then kernel code has to figure out the reason for the system call based on the call’s arguments.
• In contrast, transferring from kernel to user mode happens under different circumstances: returning from a system call, creating a new process, or switching to different process after the current process has been interrupted.

Definition 3.12 (Execution state). The execution state of a process consists of the current values of user mode processor registers, stack pointer, program counter, and other state such as a page table base address.

Remarks:
• When entering the kernel from user mode, the kernel must save the execution state of the currently running process so it can resume it later.
• The most common way of creating a new process is to create this “saved” execution state, and then “resume” it as if it had previously entered the kernel.
• There is one exception to this “resume” model of process control, which we will say later when we look at communication primitives: the upcall.

Definition 3.13 (System call). A system call is a trap (synchronous exception) deliberately invoked by a user program to request a service from the kernel. A system call is defined as taking arguments and returning values.

Example 3.14 (Unix write()). Consider the write() system call in Unix (type “man 2 write” for more documentation).

write() has the following functional prototype:

\[
\text{ssize_t write(int fd, const void *buf, size_t count);} \\
\]

Such a call would be implemented as follows:

Algorithm 3.15 System call for write( fd, buf, count )

Procedure in user space
1. Load fd, buf, and count into processor registers
2. Load system call number for write into a register
3. Trap
4. Read result from a register
5. return Result

Execution in the kernel (the trap handler)
6. Set up execution environment (stack, etc.)
7. Read system call number from register
8. Jump to write code based on this
9. Read fd, buf, and count from processor registers
10. Check buf and count for validity
11. Copy count bytes from user memory into kernel buffer
12. Do the rest of the code for write
13. Load the return value into a register
14. Resume the calling process, transfer to user mode

Remarks:
• This is very similar to a Remote Procedure Call, which we will see later.
• The code sequences in user space and kernel that marshal the arguments and return values are called stubs.
• In traditional operating systems, this code is mostly written by hand. However, it requires care to get right, and the consequences of a bug here can result in corruption of the kernel, or worse.

Bibliography