The OS Kernel and System Calls

Processes

One of the key concepts in operating systems is the notion of a *process*. A process is a computer program that has been started by someone or something and is active on the system. It may or may not be currently using the CPU.

Processes have instructions or code that they run and they have memory, both in CPU registers and in RAM. The OS maintains a data structure for each process that has info about it such as its process ID, what files it is using, its current working directory, what state it is in, and more. Depending on the OS, there can be a number of possible states for the process. Many of the states are variations on these three:

- Running — The process is currently using the CPU
- Ready — The process is ready to use the CPU, but it isn't using it, most likely because something else is using the CPU.
- Blocked — The process doesn't want to use the CPU right now because it is waiting for something. That something might be user input, it might be waiting for a disk right to finish, or it might be waiting for another process to release a resource it needs.

In summary, a process consists of code, its memory, its state on the CPU (register values), and the metadata held in the OS data structure.

The kernel

The *kernel* of an operating system is the part of the OS that is responsible for the core features of running the system. These include things like scheduling processes, managing memory, and managing devices. Non-kernel features include things like the OS's screenshot tool, its text editor, and its terminal emulator.

Types of kernels Kernels are divided into two types: *monolithic kernels* and *microkernels*. A monolithic kernel is essentially in one big piece. It is like a single process with a bunch of different functions to handle different tasks. In a microkernel, all of the tasks are broken into their own processes, called *servers*. A benefit of this approach is that if something goes wrong with one of the servers and it crashes, it will just take down that server, not the whole system. In a monolithic kernel, a crash would take down the entire server.

The trend, however, has been to use monolithic kernels because they are much faster. It is more efficient for the different parts of an operating system to communicate if they are part of the same process than if they are in separate processes. Also, it takes time to switch from one process to another, which adds additional overhead to the microkernel approach. Linux and most other Unix distributions are monolithic kernels. Windows and MacOS are *hybrid kernels*. That is, they are mostly monolithic, but some parts are broken into separate processes.

System calls

Most modern computers have both a *kernel mode* and a *user mode*. Kernel mode is a privileged state, while user mode is a restricted state. Anything operating in kernel mode has full, unrestricted access to the system. It's specifically for the kernel. Anything in user mode has much more limited access to the system. It is for most ordinary processes running on the system. Kernel mode is indicated by a flag in a particular CPU register. Anytime a process attempts to do a privileged operation, a check is first made of that CPU register to make sure the system is in kernel mode.

The reason for two modes is that it's a bad idea to give a process full access to everything on the computer. For instance, you wouldn't want some random program you installed to have access to the memory used by your banking app. Also, a program with full access could accidentally do things that would crash or even destroy the operating system.

Writing to the hard drive is one example of an operation that can only be done in kernel mode. This is a bit of a problem as many ordinary user-mode processes would need to write to the hard drive. The solution is for the user-mode process to ask the kernel to do the writing for it. The mechanism for this is a *system call*. Some common system calls include creating a new process, killing a process, reading/writing files, and deleting files. Modern systems typically have a few hundred different types of systems calls.

On Unix systems, systems calls often look a lot like ordinary programming language library calls. For instance, the `write` system call for writing to a file has the signature `write(int fd, const void *buf, size_t count)`. It takes a file descriptor, a buffer to write, and an amount of bytes to write. However, under the hood, things are a little more complicated. When a process makes a system call, the following sequence of events happens:

1. A special CPU instruction called a *trap* is run. It sets things from user mode to kernel mode by changing the value of a flag (a single bit) in a CPU register. The trap instruction then copies some of the process's register values into a part of the process's memory called the *kernel stack*.
2. Next, each system call has a number associated with it which is an index into a table called the *trap table* that contains the locations of the OS code to run for each system call. That table is set up by the OS when it first boots up. Once the table is consulted, the corresponding system call code is run, which performs the desired operation (like writing to disk).
3. Then a CPU instruction copies the process's register values back from memory into the CPU registers, and it switches things back into user mode. The process can then pick up with the next thing it was going to do.

The reason for saving the process's register values is that when the system call does its thing (like write to disk), it needs to use the CPU to do that. In particular, it will need to store things in the CPU registers, and that would destroy whatever was currently there. When the system call is done, the process that called it needs to pick up where it left off, and that wouldn't work very well if the values it was using are destroyed.