Lecture 2

Gidon Rosalki

2025-03-30

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Kernel mode and interrupts

There are 2 classes of instruction in the OS. Privileged, and non-privileged instructions. The non-privileged instructions are all the usual instructions that you would expect such as add, multiply, jump, etc. The privileged instructions are special instructions for the OS, other user applications cannot use them. For example, an instruction to activate the disk to get a block of data. The OS uses this to support the abstraction of files, and prevent access to other users' data. We may thus split the computer world in 2. The applications run in user mode, and only have access to system calls. The operating system and hardware run in kernel mode, and have access to the privileged instructions, and system interrupts (interrupts are from the hardware to the OS). Kernel mode is also called privileged mode. Kernel mode has access to both the privileged, **and** non-privileged instructions, where user mode **only** has access to the non-privileged instructions.

1.1 System calls

User processes get OS services through **system calls**. These are functions that can be called from user programs. It is important to note that these are **not** like regular function calls. User programs can access and alter internal OS data structures **only** through system calls. System calls also check parameter validity, unlike other internal OS functions. This is because they are exposed to external users, who will make mistakes. System calls enable a portable interface, to do things on different sets of hardware without worrying about the underlying implementation. They also provide protection. So how does this work in a program? The program executes, until it performs a system call. When it makes a system call, then the *mode bit* is set to 0, indicating kernel mode, and the computer executes the system call in kernel mode. Once the system call is finished, the mode bit is set back to 1, indicating user mode, and the program continues after the system call. It is important to note that the changing of the bit is also a privileged instruction, since otherwise one would be able to arbitrarily start executing in kernel mode, which is not secure. Modern OSs generally have more than 1 mode bit, resulting in more than 2 modes, generally called `rings`. The instruction in question for changing modes is the *trap* instruction. In detail, the user app calls a function in a system library, such as *open*. This then calls the *trap* instruction, which brings us into kernel mode, to the system calls entry point. It extracts the system call number, and uses it in a switch to call the actual function, passing it the stored parameters. The system call then executes (in our example, *sys_open*). When it completes, it returns to the entry point function, which will then store the return value in a register than can be accessed by the user mode, and returns to the library function that then retrieves the stored value. Finally, this returns to the user app. This entire process causes significant overhead. There is nothing to be done about this, this is the trade off we need to have portable and secure code, provided by the OS.

Types of system calls include process control, file management, device management, information maintenance (CPU usage, RAM usage, etc.), communication (such as between processes), protection (permissions, what can be run/read by which user, etc.).

Sometimes things go wrong, and exceptions are thrown (such as, when a program divides by 0). These happen when the hardware cannot execute the required instruction. Thus there is a transition to kernel mode in these instances, such that the OS can handle the exception.

In the top level of the OS stack, we have the shell and windows. There are 2 types of programs that are run on an OS. There are the system programs / system utilities, where they partially run in user space, and partially in kernel space. Modern OSs ship with system programs. They are often an interface to a system call (runs in user mode), are sometimes more complex, and are designed to provide services to other software. They are not to be confused with application programs, which are sometimes provided by the OS, but are not part of the OS. They are run by users, and use system programs to complete their operations. An example of this is a web browser. An example of a system program is *echo*. It writes the argument to standard output.
Beneath this in the OS stack, we have libraries. They are usually library specific, and often have higher level abstractions.

The next level down is the OS Abstractions, and is the level in which we will work for most of this course. These provide lower level abstractions and mechanisms, such as storage with file systems, computation with processes, and communication with sockets.
Next we have hardware drivers, which provide usable interface to the hardware. These make up the lions share of the code volume in an OS. They are often written by the device vendors.
Below this is the hardware. Consider other courses (such as computer architecture) for that.

There is a problem here. The drivers are part of the OS, and so run inside kernel mode. They thus have access to all OS data structures, and can do things like change process priority. This is not a good idea. The solution to this problem is to use multiple privilege levels, not just two. For this we use the `ring` systems. However, since this is a complicated an expensive process, most OSs do not use this process.

1.2 Interrupts

We connect I/O devices through what is called a **bus**. This has the advantage of being versatile, new devices can be added easily. Peripherals can be moved between computer system that use the same bus standard. Additionally, it is low cost. A single set of wires is shared in multiple ways. However, it also has its disadvantages. For example, it creates a communication bottleneck between the devices. The bandwidth of that bus can limit the maximum I/O throughput. The maximum speed of the bus is largely limited by the length of the bus, the number of devices of the bus, and the need to support a range of devices with widely varying latencies and data transfer rates. In reality there are often many buses, the PCI bus, the expansion bus, and so on.

In order to communicate with I/O devices we use interrupts. Devices have controllers with firmware, and writing to certain bus addresses actives the controllers and transfers data to them. The driver for a device knows the specific details. Controllers operate in the background, using DMA, and use interrupts to signal completion. Examples of interrupts are mouse clicks, and keyboard presses. Interrupts are generally either external input (be it user or network), or to signal to the computer that some process has finished on the device. They cause the CPU to stop what it is doing, and start running an OS function to handle the interrupt, just like exceptions and system calls. Upon an interrupt the CPU needs to run the correct interrupt handler. This is part of the OS. However, how does the hardware know where the OS function is located? After all, hardware is unchangeable, and may have been created before a given OS was created. The solution to this is indirection via the interrupt vector. The hardware defines an area in memory where it expects to find pointers to interrupt handlers. This is part of the architecture. The OS is responsible for installing pointers to the correct interrupt handlers in cells of the interrupt vector. This is done during system boot, when new devices are installed, and are defined as OS memory that is inaccessible to users.
Upon an interrupt the hardware blindly loads the address from the specified vector to the PC, and sets the mode bit to kernel mode. Identifying the device is part of the interrupt mechanism. Additionally IDs are assigned to the devices, and this is part of what is called plug-and-play, meaning that when you connect a new device such as a mouse, it starts working immediately, instead of needing to be connected on start-up. On x86 there are 256 interrupt entries, and the first 32 are reserved, for example 0x00 is reserved for division by 0, and 0x80 for system calls traps.
There are a few classifications of interrupts. For example, software vs hardware. Software interrupts are caused by a piece of software, which includes traps and exceptions. Hardware is caused by the hardware, which includes interrupts from external devices. There is also an internal vs external division, generally internal is caused by software, and external is cause by hardware. There is also synchronous vs asynchronous (also known as periodic vs aperiodic). Synchronous interrupts happen with the internal computer clock tick. This is useful in time sharing. Additionally, there are maskable vs non-maskable. Maskable interrupts are hardware interrupts that can be delayed if more important interrupts are currently being handled.
If we return to consider the trap instruction we discussed earlier, we now know it is in fact an interrupt, and is the interrupt 0x80, which is the syscall interrupt.

We mentioned earlier that making system calls has significant overhead. To deal with this we could make a smaller kernel, this would then have fewer traps, less overhead, and therefore higher throughput. More of the CPU time would be spent running our programs, rather than handing off between the user programs and the OS. There would also be a lower memory footprint, and since most functions would be outside of the kernel, changing these would not require recompiling the kernel. However, there would also be less protection provided by this kernel. Such is the trade off.

There are a few types of kernels. There are monolithic kernels such as MS-DOS, and Unix, and modular monolithic, whose primary difference is the existence of loadable kernel modules. Mac OS X, Windows, Linux, Solaris, and modern Unix are all modular monolithic.