Tutorial 2

Gidon Rosalki

2025-04-02

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

1 Interrupts and system calls

1.1 User mode and Kernel mode

A **program** is an executable file (exe on windows, binary on Linux). A **process** is an executing instance of a program. An **active process** is a process that is currently advancing in the CPU, while other processes are waiting in memory for their turns to use the CPU. **Input/Output** is the collection of all programs, operations, or devices, that transfer data to or from a peripheral device. For example, disk drives, keyboards, printers, screens, etc.

The **kernel** is the core of the operating system, and it has complete control over everything that happens in the system. There are instructions that are too dangerous for everybody to be able to run. In an operating system we view the kernel as trusted software, and everything else as untrusted software. The kernel can run whatever instruction it wants, where other programs (even programs being run as sudo), cannot. To enable this separation, the CPU contains permission levels: the kernel mode and user mode. User mode is a non privileged mode, the program may only execute user commands. For example, it may not directly execute commands like halt, access all the memory (it may only access its allocated memory), and it has no direct access to hardware. When the CPU is in kernel mode it is assumed to be executing trusted software, and will thus execute any command, and reference any memory address. The entire kernel, which is a controller of processes, executes *only* in kernel mode. Some privileged instructions are

- HLT (tells the CPU to halt until an external interrupt arrives)
- Read/Write from the control registers (these contain important flags and values, for example a flag to determine whether or not to disable memory values)
- LIDT (changes values of the vector interrupt table)

In practice, in most architectures there are **protection rings**. These are 4 levels of permissions, 0 through 3, where 0 is kernel mode, and 3 has the fewest privileges (user mode). However, we will act like there are only the 2, user, and kernel. The CPU is aware of the current privilege level (CPL). In x86, the CPL is the 2 least significant bits of the code segment (CS) register. When an instruction needs to be executed, the CPU validates that the instruction's requested privilege level is fulfilled by the CPL.

The goal of the OS is to let processes run. However, processes may need services provided by the kernel (such as I/O interaction). When a user mode process wants to use a service that is provided by the kernel, the system must temporarily switch into kernel mode. The OS provides services via **system calls**. These are requests to the OS by an active process for a service performed by the kernel. Each system call has a unique identifying number, the process of performing a system call is as follows:

1. Save the current state (the register values)
2. Change the RAX register to be the identifying number of the system call
3. Set the other registers as arguments (much like with a normal function, they generally call RDI as the first argument, and RSI as the second, and so on)
4. **Trap**: switch to kernel mode, and change PC (to do with permissions, related to CS), with the SYSCALL opcode.
5. Restore the old callee registers, and return the permission to user space.

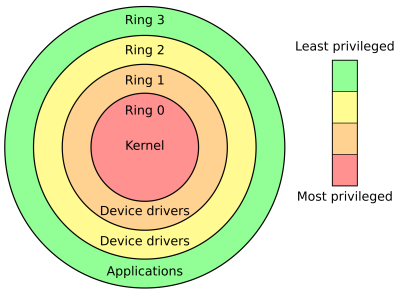

Figure 1: Privilege rings

It is important to note that Libc's functions are not system calls. The C standard library wraps some of the system calls, for example all of `write`, `read`, `printf`, `scanf`, `fopen`, open are all a part of the C standard library, not direct system calls, they simply wrap relevant system calls.
Trap has significant overhead, and is incredibly slow. Switching from kernel mode to user mode is easy, however the opposite is hard and consumes a lot of time. Since not all the kernel functions really need kernel mode, the OS decides what runs in user mode and what runs in kernel mode.

1.1.1 Example: open system call

1. The program calls `fopen`
2. C std functions are called
3. Sets the RAX to the number of `open`
4. Calls a trap (interrupt 0x80, or `syscall`) **MOVES TO KERNEL MODE**
5. Trap handler: checks which `syscall` was called (RAX)
6. Finds the memory of `sys_open`
7. Runs `sys_open`. Additional kernel functions are also called here.
8. Eventually the return value is stored in the AX register **RETURNS TO USER MODE**
9. The C std uses the return value, and eventually returns a `*FILE` object.

This is why we always check the return values when opening files / generally calling these functions. Since a positive integer is just a file descriptor, the negative values represent failures with the file.

Error name	Error code (number)	Message
ENOENT	2	No such file or directory
EINTR	4	Interrupted system call
EIO	5	I/O error
EACCESS	13	Permission denied
EBUSY	16	Device or resource busy

Table 1: Possible return values

We may use the `man` command to read manuals on other commands and functions. `man` takes 1 or more arguments. Using `man 1 <>` shows the manual for executable programs, or shell commands, `man 2 <>` for system calls (functions provided by the kernel), and `man 3 <>` for C library calls (functions within the program libraries). Consider the program `mkdir`, it has entries under 1, 2, and 3, since it is in all of these subcategories.

1.2 Interrupts

Much of the functionality embedded inside a computer is implemented by hardware devices other than the processor. Since each device operates at its own pace (different clock speed to the CPU), a method is needed for synchronising the operation of the processor with these devices. We may also use **polling**. The processor runs a loop asking each device about its current state. If new data is available, then it processes this, and if there is not yet new information, then it waits a bit and tries again. This is wasteful in terms of processing power, and when the rate of data transfer is extremely high, the processor might lose data arriving form the hardware devices. However, we often do not need things to occur at the precise nanosecond that they happened (for example with keyboards / mice), and having it regularly checked by the OS at times it specified, rather than interrupting every process to achieve it, can be wasteful, and we will instead use polling, to check at regularly scheduled and planned for intervals.
If we return to interrupts, here each hardware device is responsible for notifying the processor about its current state. When a hardware device needs the processor's attention, it simply sends an electrical signal through a dedicated pin on the interrupt controller chip. Interrupts are thus not part of the running program's pre-planned code. An interrupt can also be characterised as an "asynchronous procedure call" (APC). The interrupt is asynchronous, the program cannot control it. This all occurs at the hardware level, and is thus effectively invisible to the interrupted program.
The interrupt controller serves as an intermediate between the hardware devices and the processor. The interrupt controller has several input lines that take requests from the different devices. Its responsibility is to alert the processor when one of the hardware devices needs its immediate attention. It passes the request to the CPU, telling it which device issued the request, (which interrupt number triggered the request, based off the interrupt vector).

We can categorise interrupts in several ways. They may be categorised by source (caused by software events, or caused by hardware events). Traps are a special kind of internal interrupt, but not every internal interrupt is a trap, for example, division by 0 is an internal interrupt, but not a trap. External interrupts are caused by external hardware events, for example pressing a key on the keyboard, the disk drive having data that was requested, the timer (used by the OS as an alarm clock) expired (for example consider time sharing, each program runs for an amount of time).

For external interrupts, let us consider the following program:

```
add r1, r2
sub r4, r4
and r5, r3
```

As execution reaches the code above, a user moves the mouse, and an interrupt is triggered. Based on the time of the movement (in the middle of sub), the hardware completes `add`, and `sub`, but delays `and` for now. The handler starts, and moves the mouse on the screen. The handler finishes, and then execution resumes with `and`.
An internal interrupt (also called exception) occurs when the processor detects an error condition while executing an instruction. This is unlike external interrupts which occur at random times during the execution of the program. For example, we may get internal interrupts from division by zero, attempts to execute privileged instructions, invalid instructions, segmentation faults / page faults (will be discussed later in the course). When an exception occurs, the kernel interrupt routine runs:

1. The address of the instruction which generated the exception is saved
2. Control is handed over to the OS
3. Depending on the exception, the OS may notify the user space process, and let the application handle it. If the program can handle it, the program is restarted at the address of the fault, and continues normally. Otherwise, the program is aborted.

Sometimes not all the memory a program uses is currently on the main memory. The data might be on the disk, and should be fetched to main memory so it can be used. Suppose a program requests data that is not in main memory. An exception is triggers the OS to fetch the data from the disk, and load it into main memory. This occurs invisibly to the program, the program gets the data without knowing that an exception has occurred, and the program will continue with the same instruction.
A trap is a type of exception, which occurs in the usual running of the program, but unlike it, it is not the product of some error. Traps are instead generated upon system calls, and cause switching to OS code, and to kernel mode. Once in kernel mode, a trap handler is executed to service the request. They then restart at the address following the address causing the trap.
Interrupt handling is like dealing with a function call, with the hardware calling a function to deal with it. Hence we need to save the state as it was when the interruption happened, handle the interruption, and then return to the state as it was. In external interrupts, both hardware and software participate in handling the interrupt. As stated earlier, handling interrupts is a process that is extremely similar to function calls, and the order of operations is virtually identical:

1. Interrupt is triggered
2. Save the state (partially)
3. Transfer control, and service the request
4. Previous state is restored
5. Return control

In more detail:

1. The signal interrupting the processor is only recognised at the end of the instruction cycle loop. In other words, after the current instruction is processed, but before the next is fetched from memory.
2. Before it can service the interrupt, it must first save its current status. It must save the program counter, to know to where to return, and the contents of several registers, as they might be changed by the handling of the interrupt.
3. The CPU checks which device sent the interrupt request, and determines where to find the necessary instructions needed to service that specific request. An **interrupt vector** contains the interrupt device numbers, and the addresses for subroutines for each interrupt handler. The interrupt vector is stored at a predefined memory location by the CPU. It also switches to kernel mode at this point.
4. All register values are restored to their previous values, as they were before the interrupt began, the CPU is returned to user mode, the next instruction is pointed to by the program counter

5. Control is returned.

There is an asm instruction `INT3`, which is a one byte instruction defined for use by debuggers to temporarily replace an instruction on a running program in order to set a code breakpoint. Once the process executes the `INT3` instruction, the OS stops it. A simple usage in C would be `__asm__("int3");`:

```
int i;
for (i = 0; i < 3; i++)
    {
        __asm__("int3");
    }
```

The OS notifies the process that it has to stop, using a signal (more in tutorial 3).