

Lecture 3 — Traps and OS Services

Jeff Zarnett

It's a Trap!

Operating systems run, as previously discussed, on interrupts. In addition to the interrupts that will be generated by hardware and devices (e.g., a keyboard signalling that the F1 key has been pressed), there are also interrupts generated in software. These are often referred to as a *trap* (or, sometimes, an exception). The trap is usually generated either by an error like an invalid instruction or from a user program request.

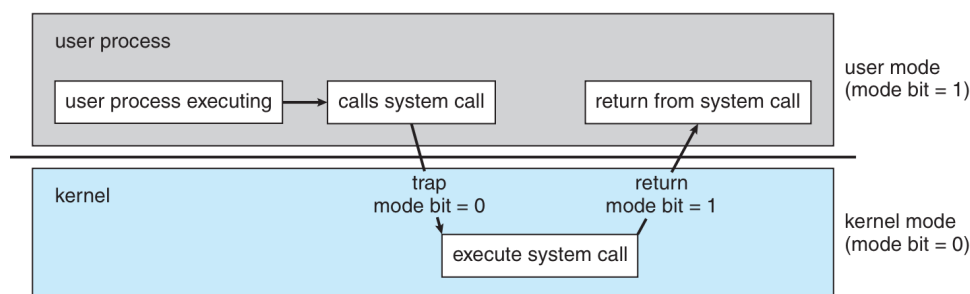
If it is simply an error the operating system will decide how to deal with it, and in desktop/laptop OSes, the usual strategy is sending the exception to the program that caused it, and this is usually fatal to the offending program. Your programming experience will tell you that you can sometimes deal with an exception (perhaps through the language equivalent of the Java/C# `try-catch-finally` syntax), but often an exception is unhandled and terminates the program.

The more interesting case is the intentional use of the trap: this is how a user program gets the operating system's attention. When a user program is running, the operating system is not; we might even say it is "sleeping". If the program running needs the operating system to do something, it needs to wake up the OS: interrupt its sleep. When the trap occurs, the interrupt handler (part of the OS) is going to run to deal with the request.

Already we saw the concept of user mode vs. supervisor mode instructions: some instructions are not available in user mode. Supervisor mode, also called kernel mode, allows all instructions and operations. Even something seemingly simple like reading from disk or writing to console output requires privileged instructions. These are common operations, but they involve the operating system every time.

Modern processors keep track of what mode they are in with the mode bit. This was not the case for some older processors and some current processors have more than two modes, but we will restrict ourselves to dual-mode operation with a mode bit. Thus we can see at a glance which mode the system is in. At boot up, the computer starts up in kernel mode as the operating system is started and loaded. User programs are always started in user mode. When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode; when it is finished the system goes back to user mode before the user program resumes [SGG13].

Suppose a text editor wants to output data to a printer. Management of I/O devices like printers is the job of the OS, so to send the data, the text editor must ask the OS to step in, as in the diagram below:



Transition from user to supervisor (kernel) mode [SGG13].

So to print out the data, the program will prepare the data for printing. Then it calls the system call. You may think of this as being just like a normal function call, except it involves the operating system. This triggers the

operating system (with a trap). The operating system responds and executes the system call and dispatches that data to the printer. When this job is done, operation goes back to user mode and the program returns from the system call.

Motivation for Dual Mode Operation

Why do we have user and supervisor modes, anyway? As Uncle Ben told Spiderman, “with great power comes great responsibility”. Many of the reasons are the same as why we have user accounts and administrator accounts: we want to protect the system and its integrity against errant and malicious users.

An example: multiple programs might be trying to use the same I/O device at once. If Program 1 tries to read from disk, it will take time for that request to be serviced. During that time, if Program 2 wants to read from the same disk, the operating system will force Program 2 to wait its turn. Without the OS to enforce this, it would be up to the author(s) of Program 2 to check if the disk is currently in use and to wait patiently for it to become available. That may work if everybody plays nicely, but without someone to enforce the rules, sooner or later there will be a program that does something nasty, like cancel another program’s read request and perform its read first.

This doesn’t come for free, of course: there is a definite performance trade-off. Switching from user mode to kernel mode requires some instructions and some time. It would be faster if everything ran in kernel mode because we would spend no time switching. Despite this, the performance hit for the mode switch is judged worthwhile for the security and integrity benefits it provides.

Example: Reading from Disk

Let us examine in some more detail what is actually happening in a system call. This example is from [Tan08] and will use C code to perform a read on a UNIX system. The specification says the function `read` takes three parameters: (1) the file name; (2) where to read the data to; and (3) how many bytes to read. Here is an example:

```
int bytesRead = read( filename, buffer, numBytes );
```

(The `read` function returns the number of bytes successfully read; it is normally equal to `numBytes` but might be smaller if the end of the file is reached. Here we are storing it in a variable `bytesRead`.)

In preparation for invocation of `read` the parameters are pushed on the stack. This is the normal way in which a procedure is called in C(++). Then the `read` procedure is called and this is just the normal instruction to enter another function. The `read` function will put its identifier (the system call number) in a predefined location (typically a register). Then it executes the trap instruction, activating the OS.

When the trap occurs, the OS takes over and control switches from user mode to kernel mode. Control transfers to a predefined memory location within the kernel (the trap handler). The trap handler then runs and examines the request: it checks the identifier that was put in the register earlier. Based on that, it knows what system call request handler should execute: the one to read from a file. That routine executes. When it is finished, control will be returned to the `read` function; we exit the kernel and return to user mode.

Back in user mode, the `read` call finishes and returns, and control goes back to the user program.

Summary: Invoking a System Call

To summarize, the steps, arranged chronologically, when invoking a system call are:

1. The user program pushes arguments onto the stack.
2. The user program invokes the system call.

3. The system call puts its identifier in the designated location.
4. The system call issues the `trap` instruction.
5. The OS responds to the interrupt and examines the identifier in the designated location.
6. The OS runs the system call handler that matches the identifier.
7. When the handler is finished, control exits the kernel and goes back to the system call (in user mode).
8. The system call returns control to the user program.

Operating System Structures

References

- [SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.
- [Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.