## **User Space and Kernel Space**

A module runs in *kernel space*, whereas applications run in *user space*. This concept is at the base of operating systems theory.

The role of the operating system, in practice, is to provide programs with a consistent view of the computer’s hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This nontrivial task is possible only if the CPU enforces protection of system software from the applications.

Every modern processor is able to enforce this behavior. The chosen approach is to implement different operating modalities (or levels) in the CPU itself. The levels have different roles, and some operations are disallowed at the lower levels; program code can switch from one level to another only through a limited number of gates. Unix systems are designed to take advantage of this hardware feature, using two such levels. All current processors have at least two protection levels, and some, like the x86 family, have more levels; when several levels exist, the highest and lowest levels are used. Under Unix, the kernel executes in the highest level (also called *supervisor mode)*, where everything is allowed, whereas applications execute in the lowest level (the so-called *user mode)*, where the processor regulates direct access to hardware and unauthorized access to memory.

We usually refer to the execution modes as *kernel space* and *user space*. These terms encompass not only the different privilege levels inherent in the two modes, but also the fact that each mode can have its own memory mapping—its own address space—as well.

Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt. Kernel code executing a system call is working in the context of a process—it operates on behalf of the calling process and is able to access data in the process’s address space. Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process.

The role of a module is to extend kernel functionality; modularized code runs in kernel space. Usually, a driver performs both the tasks outlined previously: some functions in the module are executed as part of system calls, and some are in charge of interrupt handling.

## **Concurrency in the Kernel**

One way in which kernel programming differs greatly from conventional application programming is the issue of concurrency. Most applications, with the notable exception of multithreading applications, typically run sequentially, from the beginning to the end, without any need to worry about what else might be happening to change their environment. Kernel code does not run in such a simple world, and even the simplest kernel modules must be written with the idea that many things can be happening at once.

There are a few sources of concurrency in kernel programming. Naturally, Linux systems run multiple processes, more than one of which can be trying to use your driver at the same time. Most devices are capable of interrupting the processor; interrupt handlers run asynchronously and can be invoked at the same time that your driver is trying to do something else. Several software abstractions run asynchronously as well. Moreover, of course, Linux can run on symmetric multiprocessor (SMP) systems, with the result that your driver could be executing concurrently on more than one CPU. Kernel code has been made preemptible; this change causes even uniprocessor systems to have many of the same concurrency issues as multiprocessor systems.

As a result, Linux kernel code, including driver code, must be *reentrant* —it must be capable of running in more than one context at the same time. Data structures must be carefully designed to keep multiple threads of execution separate, and the code must take care to access shared data in ways that prevent corruption of the data. Writing code that handles concurrency and avoids race conditions (situations in which an unfortunate order of execution causes undesirable behavior) requires thought and can be tricky. Proper management of concurrency is required to write correct kernel code; for that reason, every sample driver in this book has been written with concurrency in mind. The techniques used are explained as we come to them.

A common mistake made by driver programmers is to assume that concurrency is not a problem as long as a particular segment of code does not go to sleep (or “block”). Even in previous kernels (which were not preemptive), this assumption was not valid on multiprocessor systems. Kernel code can (almost) never assume that it can hold the processor over a given stretch of code. If you do not write your code with concurrency in mind, it will be subject to catastrophic failures that can be exceedingly difficult to debug.

Defined on line 31 - int proc\_init(void)

Prints Kernel info created

Defined on line 41 – proc\_exit(void)

Prints Kernel info removed

Defined on line 49 - sszie Proc\_read

Checking if the process has completed or not and assigning the variables accordingly.

Copying all the values from the user to buffer