For example, let’s say you’re driving a car and the ride is rough. You can quickly assess the three basic car-related abstractions just mentioned to determine the source of the problem. It should be fairly easy to eliminate the first two abstractions (your car or the way you’re driving) if neither is the issue, so you can narrow the problem down to the road itself. You’ll probably find that the road is bumpy. Now, if you want, you can dig deeper into your abstraction of the road and find out why the road has deteriorated or, if the road is new, why the construction workers did a lousy job.

Software developers use abstraction as a tool when building an operating system and its applications. There are many terms for an abstracted subdivision in computer software—including *subsystem*, *module*, and *package*—but we’ll use the term *component* in this chapter because it’s simple. When building a software component, developers typically don’t think much about the internal structure of other components, but they do consider other components they can use (so that they don’t have to write any additional unnecessary software) and how to use them.

This chapter provides a high-level overview of the components that make up a Linux system. Although each one has a tremendous number of technical details in its internal makeup, we’re going to ignore these details and concentrate on what the components do in relation to the whole system. We’ll look at the details in subsequent chapters.

1.1 Levels and Layers of Abstraction in a Linux System

Using abstraction to split computing systems into components makes things easier to understand, but it doesn’t work without organization. We arrange components into *layers* or *levels*, classifications (or groupings) of components according to where the components sit between the user and the hardware. Web browsers, games, and such sit at the top layer; at the bottom layer we have the memory in the computer hardware—the 0s and 1s. The operating system occupies many of the layers in between.

A Linux system has three main levels. *Figure 1-1* shows these levels and some of the components inside each level. The *hardware* is at the base. Hardware includes the memory as well as one or more central processing units (CPUs) to perform computation and to read from and write to memory. Devices such as

disks and network interfaces are also part of the hardware.

The next level up is the *kernel*, which is the core of the operating system. The kernel is software residing in memory that tells the CPU where to look for its next task. Acting as a mediator, the kernel manages the hardware (especially main memory) and is the primary interface between the hardware and any running program.

Processes—the running programs that the kernel manages—collectively make up the system's upper level, called *user space*. (A more specific term for process is *user process*, regardless of whether a user directly interacts with the process. For example, all web servers run as user processes.)

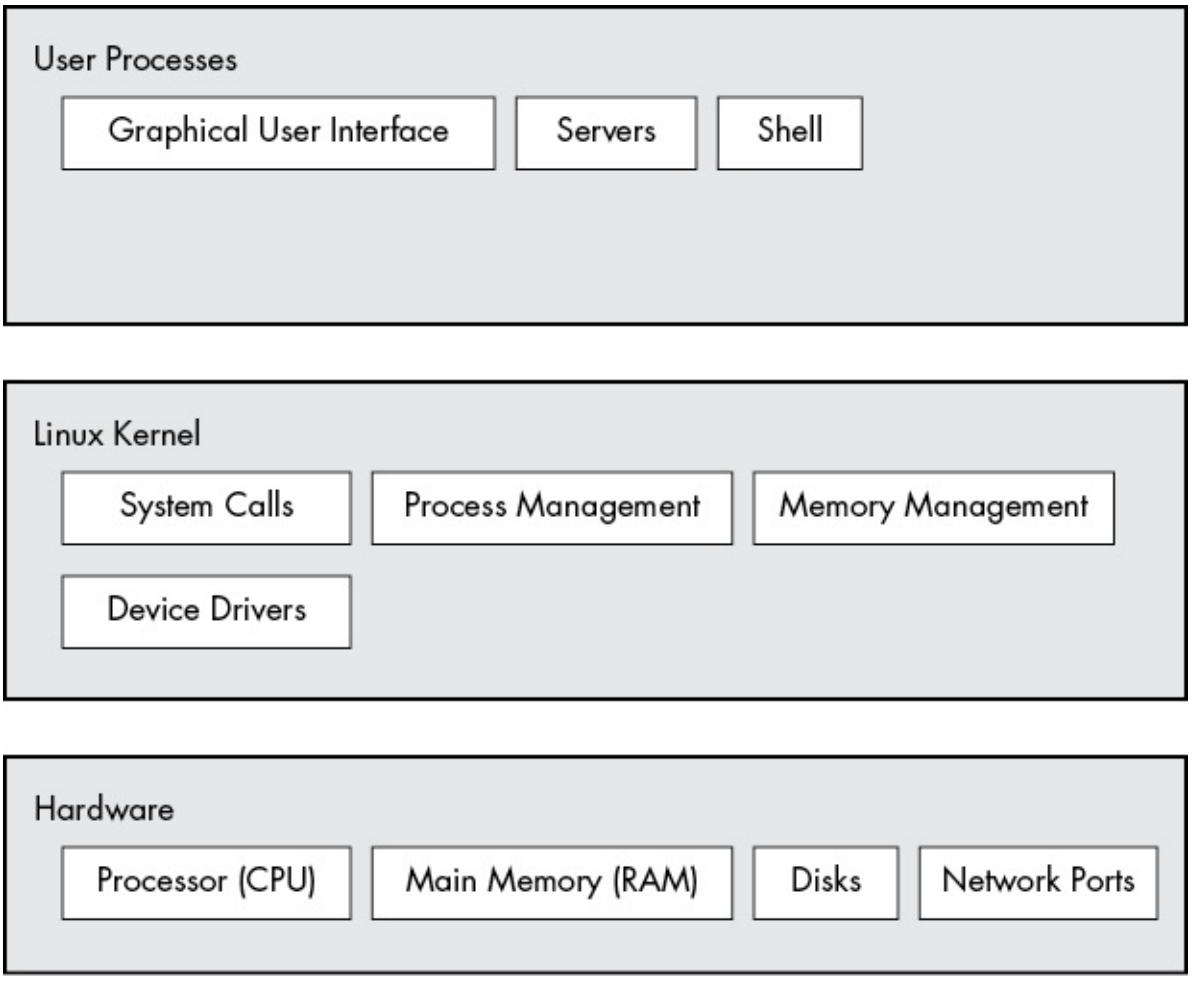

Figure 1-1: General Linux system organization

There is a critical difference between how the kernel and the user processes run: the kernel runs in *kernel mode*, and the user processes run in *user mode*. Code running in kernel mode has unrestricted access to the processor and main

memory. This is a powerful but dangerous privilege that allows the kernel to easily corrupt and crash the entire system. The memory area that only the kernel can access is called *kernel space*.

User mode, in comparison, restricts access to a (usually quite small) subset of memory and safe CPU operations. *User space* refers to the parts of main memory that the user processes can access. If a process makes a mistake and crashes, the consequences are limited and can be cleaned up by the kernel. This means that if your web browser crashes, it probably won't take down the scientific computation that has been running in the background for days.

In theory, a user process gone haywire can't cause serious damage to the rest of the system. In reality, it depends on what you consider "serious damage," as well as the particular privileges of the process, because some processes are allowed to do more than others. For example, can a user process completely wreck the data on a disk? With the correct permissions, yes—and you might consider this to be fairly dangerous. There are safeguards to prevent this, however, and most processes simply aren't allowed to wreak havoc in this manner.

NOTE

The Linux kernel can run kernel threads, which look much like processes but have access to kernel space. Some examples are `kthreadd` *and* `kblockd`.

1.2 Hardware: Understanding Main Memory

Of all of the hardware on a computer system, *main memory* is perhaps the most important. In its rawest form, main memory is just a big storage area for a bunch of 0s and 1s. Each slot for a 0 or 1 is called a *bit*. This is where the running kernel and processes reside—they're just big collections of bits. All input and output from peripheral devices flows through main memory, also as a bunch of bits. A CPU is just an operator on memory; it reads its instructions and data from the memory and writes data back out to the memory.

You'll often hear the term *state* in reference to memory, processes, the kernel, and other parts of a computer system. Strictly speaking, a state is a particular arrangement of bits. For example, if you have four bits in your memory, 0110,

0001, and 1011 represent three different states.

When you consider that a single process can easily consist of millions of bits in memory, it's often easier to use abstract terms when talking about states. Instead of describing a state using bits, you describe what something has done or is doing at the moment. For example, you might say, "The process is waiting for input" or, "The process is performing Stage 2 of its startup."

NOTE

Because it's common to refer to the state in abstract terms rather than to the actual bits, the term image *refers to a particular physical arrangement of bits.*

1.3 The Kernel

Why are we talking about main memory and states? Nearly everything that the kernel does revolves around main memory. One of the kernel's tasks is to split memory into many subdivisions, and it must maintain certain state information about those subdivisions at all times. Each process gets its own share of memory, and the kernel must ensure that each process keeps to its share.

The kernel is in charge of managing tasks in four general system areas:

Processes The kernel is responsible for determining which processes are allowed to use the CPU.

Memory The kernel needs to keep track of all memory—what is currently allocated to a particular process, what might be shared between processes, and what is free.

Device drivers The kernel acts as an interface between hardware (such as a disk) and processes. It's usually the kernel's job to operate the hardware.

System calls and support Processes normally use system calls to communicate with the kernel.

We'll now briefly explore each of these areas.

NOTE