This chapter looks at virtual memory from two angles. The first half of the chapter describes how virtual memory works. The second half describes how virtual memory is used and managed by applications. There is no avoiding the fact that VM is complicated, and the discussion reflects this in places. The good news is that if you work through the details, you will be able to simulate the virtual memory mechanism of a small system by hand, and the virtual memory idea will be forever demystified.

The second half builds on this understanding, showing you how to use and manage virtual memory in your programs. You will learn how to manage virtual memory via explicit memory mapping and calls to dynamic storage allocators such as the `malloc` package. You will also learn about a host of common memory-related errors in C programs and how to avoid them.

9.1 Physical and Virtual Addressing

The main memory of a computer system is organized as an array of M contiguous byte-size cells. Each byte has a unique *physical address (PA)*. The first byte has an address of 0, the next byte an address of 1, the next byte an address of 2, and so on. Given this simple organization, the most natural way for a CPU to access memory would be to use physical addresses. We call this approach *physical addressing*. Figure 9.1 shows an example of physical addressing in the context of a load instruction that reads the 4-byte word starting at physical address 4. When the CPU executes the load instruction, it generates an effective physical address and passes it to main memory over the memory bus. The main memory fetches the 4-byte word starting at physical address 4 and returns it to the CPU, which stores it in a register.

Early PCs used physical addressing, and systems such as digital signal processors, embedded microcontrollers, and Cray supercomputers continue to do so. However, modern processors use a form of addressing known as *virtual addressing*, as shown in Figure 9.2.

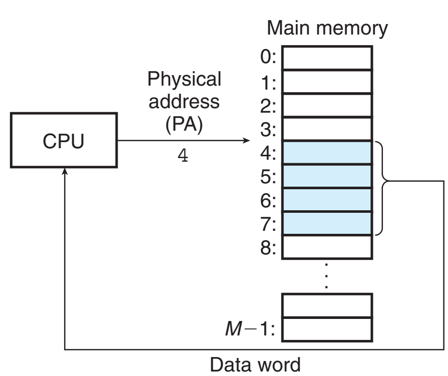

Figure 9.1
A system that uses physical addressing.

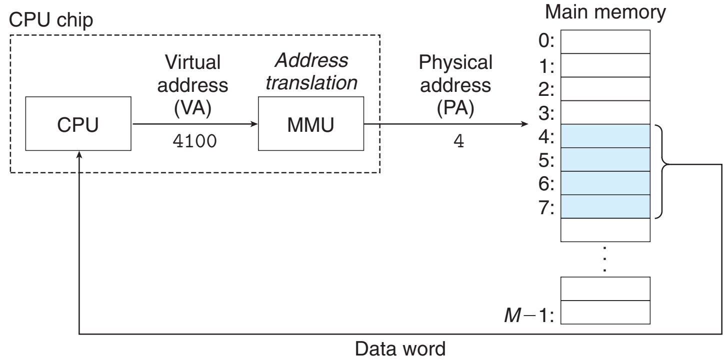

Figure 9.2
A system that uses virtual addressing.

With virtual addressing, the CPU accesses main memory by generating a *virtual address (VA)*, which is converted to the appropriate physical address before being sent to main memory. The task of converting a virtual address to a physical one is known as *address translation*. Like exception handling, address translation requires close cooperation between the CPU hardware and the operating system. Dedicated hardware on the CPU chip called the *memory management unit (MMU)* translates virtual addresses on the fly, using a lookup table stored in main memory whose contents are managed by the operating system.

9.2 Address Spaces

An *address space* is an ordered set of nonnegative integer addresses

$$\{0, 1, 2, \ldots\}$$

If the integers in the address space are consecutive, then we say that it is a *linear address space*. To simplify our discussion, we will always assume linear address spaces. In a system with virtual memory, the CPU generates virtual addresses from an address space of $N = 2^n$ addresses called the *virtual address space*:

$$\{0, 1, 2, \ldots, N - 1\}$$

The size of an address space is characterized by the number of bits that are needed to represent the largest address. For example, a virtual address space with $N = 2^n$ addresses is called an n-bit address space. Modern systems typically support either 32-bit or 64-bit virtual address spaces.

A system also has a *physical address space* that corresponds to the M bytes of physical memory in the system:

$$\{0, 1, 2, \ldots, M - 1\}$$

M is not required to be a power of 2, but to simplify the discussion, we will assume that $M = 2^m$.