Processes in a system share the CPU and main memory with other processes. However, sharing the main memory poses some special challenges. As demand on the CPU increases, processes slow down in some reasonably smooth way. But if too many processes need too much memory, then some of them will simply not be able to run. When a program is out of space, it is out of luck. Memory is also vulnerable to corruption. If some process inadvertently writes to the memory used by another process, that process might fail in some bewildering fashion totally unrelated to the program logic.

In order to manage memory more efficiently and with fewer errors, modern systems provide an abstraction of main memory known as virtual memory (VM). Virtual memory is an elegant interaction of hardware exceptions, hardware address translation, main memory, disk files, and kernel software that provides each process with a large, uniform, and private address space. With one clean mechanism, virtual memory provides three important capabilities. (1) It uses main memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active areas in main memory, and transferring data back and forth between disk and memory as needed. (2) It simplifies memory management by providing each process with a uniform address space. (3) It protects the address space of each process from corruption by other processes.

Virtual memory is one of the great ideas in computer systems. A major reason for its success is that it works silently and automatically, without any intervention from the application programmer. Since virtual memory works so well behind the scenes, why would a programmer need to understand it? There are several reasons.

- **Virtual memory is central.** Virtual memory pervades all levels of computer systems, playing key roles in the design of hardware exceptions, assemblers, linkers, loaders, shared objects, files, and processes. Understanding virtual memory will help you better understand how systems work in general.

- **Virtual memory is powerful.** Virtual memory gives applications powerful capabilities to create and destroy chunks of memory, map chunks of memory to portions of disk files, and share memory with other processes. For example, did you know that you can read or modify the contents of a disk file by reading and writing memory locations? Or that you can load the contents of a file into memory without doing any explicit copying? Understanding virtual memory will help you harness its powerful capabilities in your applications.

- **Virtual memory is dangerous.** Applications interact with virtual memory every time they reference a variable, dereference a pointer, or make a call to a dynamic allocation package such as malloc. If virtual memory is used improperly, applications can suffer from perplexing and insidious memory-related bugs. For example, a program with a bad pointer can crash immediately with a “Segmentation fault” or a “Protection fault,” run silently for hours before crashing, or scariest of all, run to completion with incorrect results. Understanding virtual memory, and the allocation packages such as malloc that manage it, can help you avoid these errors.

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-sized 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 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.

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 the memory. The task of converting a virtual address to a physical one is known as address translation. Like exception handling, address translation

![Figure 9.1](image.jpg)

**A system that uses physical addressing.**
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 look-up 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 two, but to simplify the discussion we will assume that \( M = 2^m \).

The concept of an address space is important because it makes a clean distinction between data objects (bytes) and their attributes (addresses). Once we recognize this distinction, then we can generalize and allow each data object to have multiple independent addresses, each chosen from a different address space.
This is the basic idea of virtual memory. Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.

Practice Problem 9.1

Complete the following table, filling in the missing entries and replacing each question mark with the appropriate integer. Use the following units: $K = 2^{10}$ (Kilo), $M = 2^{20}$ (Mega), $G = 2^{30}$ (Giga), $T = 2^{40}$ (Tera), $P = 2^{50}$ (Peta), or $E = 2^{60}$ (Exa).

<table>
<thead>
<tr>
<th>No. virtual address bits ($n$)</th>
<th>No. virtual addresses ($N$)</th>
<th>Largest possible virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$2^8$</td>
<td>$64K$</td>
</tr>
<tr>
<td>$2^7$</td>
<td>$2^7$</td>
<td>$2^{32} - 1 = ?G - 1$</td>
</tr>
<tr>
<td>64</td>
<td>$2^8$</td>
<td>$256T$</td>
</tr>
</tbody>
</table>

9.3 VM as a Tool for Caching

Conceptually, a virtual memory is organized as an array of $N$ contiguous byte-sized cells stored on disk. Each byte has a unique virtual address that serves as an index into the array. The contents of the array on disk are cached in main memory. As with any other cache in the memory hierarchy, the data on disk (the lower level) is partitioned into blocks that serve as the transfer units between the disk and the main memory (the upper level). VM systems handle this by partitioning the virtual memory into fixed-sized blocks called virtual pages (VPs). Each virtual page is $P = 2^p$ bytes in size. Similarly, physical memory is partitioned into physical pages (PPs), also $P$ bytes in size. (Physical pages are also referred to as page frames.)

At any point in time, the set of virtual pages is partitioned into three disjoint subsets:

- **Unallocated**: Pages that have not yet been allocated (or created) by the VM system. Unallocated blocks do not have any data associated with them, and thus do not occupy any space on disk.