CHAPTER 6

The Memory Hierarchy

To this point in our study of systems, we have relied on a simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU. In our simple model, the memory system is a linear array of bytes, and the CPU can access each memory location in a constant amount of time. While this is an effective model as far as it goes, it does not reflect the way that modern systems really work.

In practice, a *memory system* is a hierarchy of storage devices with different capacities, costs, and access times. CPU registers hold the most frequently used data. Small, fast *cache memories* nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because well-written programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data your program needs are stored in a CPU register, then they can be accessed in zero cycles during the execution of the instruction. If stored in a cache, 1 to 30 cycles. If stored in main memory, 50 to 200 cycles. And if stored in disk tens of millions of cycles!

Here, then, is a fundamental and enduring idea in computer systems: if you understand how the system moves data up and down the memory hierarchy, then you can write your application programs so that their data items are stored higher in the hierarchy, where the CPU can access them more quickly.

This idea centers around a fundamental property of computer programs known as *locality*. Programs with good locality tend to access the same set of data items over and over again, or they tend to access sets of nearby data items. Programs with good locality tend to access more data items from the upper levels of the memory hierarchy than programs with poor locality, and thus run faster. For example, the running times of different matrix multiplication kernels that perform the same number of arithmetic operations, but have different degrees of locality, can vary by a factor of 20!

In this chapter, we will look at the basic storage technologies—SRAM memory, DRAM memory, ROM memory, and rotating and solid state disks—and describe how they are organized into hierarchies. In particular, we focus on the cache memories that act as staging areas between the CPU and main memory, because they have the most impact on application program performance. We show you how to analyze your C programs for locality and we introduce techniques for improving the locality in your programs. You will also learn an interesting way to characterize the performance of the memory hierarchy on a particular machine as a "memory mountain" that shows read access times as a function of locality.

6.1 Storage Technologies

Much of the success of computer technology stems from the tremendous progress in storage technology. Early computers had a few kilobytes of random-access memory. The earliest IBM PCs didn't even have a hard disk. That changed with the introduction of the IBM PC-XT in 1982, with its 10-megabyte disk. By the year 2010, typical machines had 150,000 times as much disk storage, and the amount of storage was increasing by a factor of 2 every couple of years.

6.1.1 Random-Access Memory

Random-access memory (RAM) comes in two varieties—*static* and *dynamic*. *Static RAM* (SRAM) is faster and significantly more expensive than *Dynamic RAM* (DRAM). SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAM stores each bit in a *bistable* memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or *states*. Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable states. Such a memory cell is analogous to the inverted pendulum illustrated in Figure 6.1.

The pendulum is stable when it is tilted either all the way to the left or all the way to the right. From any other position, the pendulum will fall to one side or the other. In principle, the pendulum could also remain balanced in a vertical position indefinitely, but this state is *metastable*—the smallest disturbance would make it start to fall, and once it fell it would never return to the vertical position.

Due to its bistable nature, an SRAM memory cell will retain its value indefinitely, as long as it is kept powered. Even when a disturbance, such as electrical noise, perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.

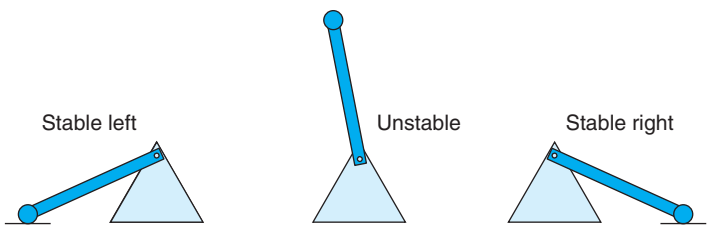


Figure 6.1
Inverted pendulum. Like an SRAM cell, the pendulum has only two stable configurations, or *states*.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1×	Yes	No	100×	Cache memory
DRAM	1	10×	No	Yes	1×	Main mem, frame buffers

Figure 6.2 **Characteristics of DRAM and SRAM memory.**

Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads, that is, 30×10^{-15} farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense—each cell consists of a capacitor and a single access transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

Various sources of leakage current cause a DRAM cell to lose its charge within a time period of around 10 to 100 milliseconds. Fortunately, for computers operating with clock cycle times measured in nanoseconds, this retention time is quite long. The memory system must periodically refresh every bit of memory by reading it out and then rewriting it. Some systems also use error-correcting codes, where the computer words are encoded a few more bits (e.g., a 32-bit word might be encoded using 38 bits), such that circuitry can detect and correct any single erroneous bit within a word.

Figure 6.2 summarizes the characteristics of SRAM and DRAM memory. SRAM is persistent as long as power is applied. Unlike DRAM, no refresh is necessary. SRAM can be accessed faster than DRAM. SRAM is not sensitive to disturbances such as light and electrical noise. The trade-off is that SRAM cells use more transistors than DRAM cells, and thus have lower densities, are more expensive, and consume more power.

Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into d *supercells*, each consisting of w DRAM cells. A $d \times w$ DRAM stores a total of dw bits of information. The supercells are organized as a rectangular array with r rows and c columns, where $rc = d$. Each supercell has an address of the form (i, j), where i denotes the row, and j denotes the column.

For example, Figure 6.3 shows the organization of a 16×8 DRAM chip with $d = 16$ supercells, $w = 8$ bits per supercell, $r = 4$ rows, and $c = 4$ columns. The shaded box denotes the supercell at address (2, 1). Information flows in and out of the chip via external connectors called *pins*. Each pin carries a 1-bit signal. Figure 6.3 shows two of these sets of pins: eight `data` pins that can transfer 1 byte

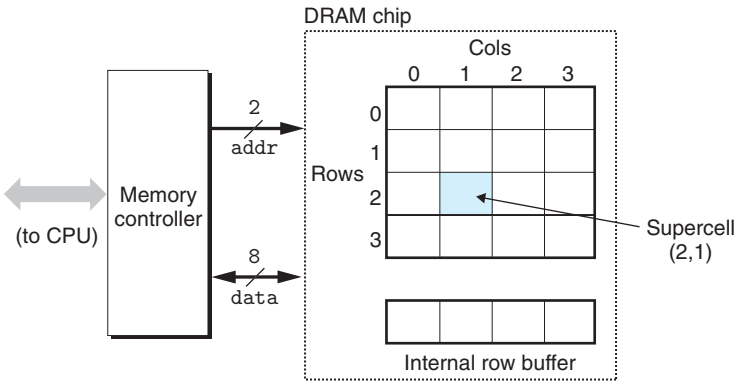


Figure 6.3
High-level view of a 128-bit 16 × 8 DRAM chip.

in or out of the chip, and two `addr` pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

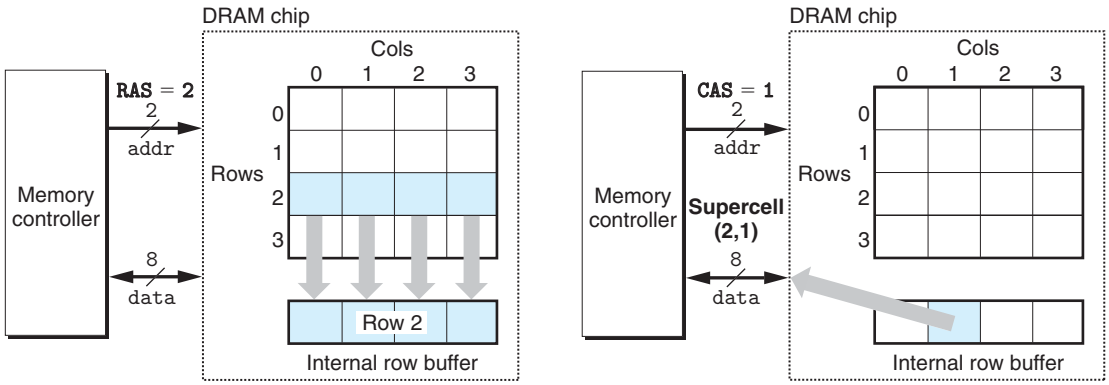
Aside A note on terminology

The storage community has never settled on a standard name for a DRAM array element. Computer architects tend to refer to it as a "cell," overloading the term with the DRAM storage cell. Circuit designers tend to refer to it as a "word," overloading the term with a word of main memory. To avoid confusion, we have adopted the unambiguous term "supercell."

Each DRAM chip is connected to some circuitry, known as the *memory controller*, that can transfer w bits at a time to and from each DRAM chip. To read the contents of supercell (i, j), the memory controller sends the row address i to the DRAM, followed by the column address j. The DRAM responds by sending the contents of supercell (i, j) back to the controller. The row address i is called a *RAS (Row Access Strobe) request*. The column address j is called a *CAS (Column Access Strobe) request*. Notice that the RAS and CAS requests share the same DRAM address pins.

For example, to read supercell (2, 1) from the 16 × 8 DRAM in Figure 6.3, the memory controller sends row address 2, as shown in Figure 6.4(a). The DRAM responds by copying the entire contents of row 2 into an internal row buffer. Next, the memory controller sends column address 1, as shown in Figure 6.4(b). The DRAM responds by copying the 8 bits in supercell (2, 1) from the row buffer and sending them to the memory controller.

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to reduce the number of address pins on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address pins instead of two. The disadvantage of the two-dimensional array organization is that addresses must be sent in two distinct steps, which increases the access time.



(a) Select row 2 (RAS request).

(b) Select column 1 (CAS request).

Figure 6.4 **Reading the contents of a DRAM supercell.**

Memory Modules

DRAM chips are packaged in *memory modules* that plug into expansion slots on the main system board (motherboard). Common packages include the 168-pin *dual inline memory module* (DIMM), which transfers data to and from the memory controller in 64-bit chunks, and the 72-pin *single inline memory module* (SIMM), which transfers data in 32-bit chunks.

Figure 6.5 shows the basic idea of a memory module. The example module stores a total of 64 MB (megabytes) using eight 64-Mbit $8M \times 8$ DRAM chips, numbered 0 to 7. Each supercell stores 1 byte of *main memory*, and each 64-bit doubleword[1] at byte address A in main memory is represented by the eight supercells whose corresponding supercell address is (i, j). In the example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

To retrieve a 64-bit doubleword at memory address A, the memory controller converts A to a supercell address (i, j) and sends it to the memory module, which then broadcasts i and j to each DRAM. In response, each DRAM outputs the 8-bit contents of its (i, j) supercell. Circuitry in the module collects these outputs and forms them into a 64-bit doubleword, which it returns to the memory controller.

Main memory can be aggregated by connecting multiple memory modules to the memory controller. In this case, when the controller receives an address A, the controller selects the module k that contains A, converts A to its (i, j) form, and sends (i, j) to module k.

1. IA32 would call this 64-bit quantity a "quadword."

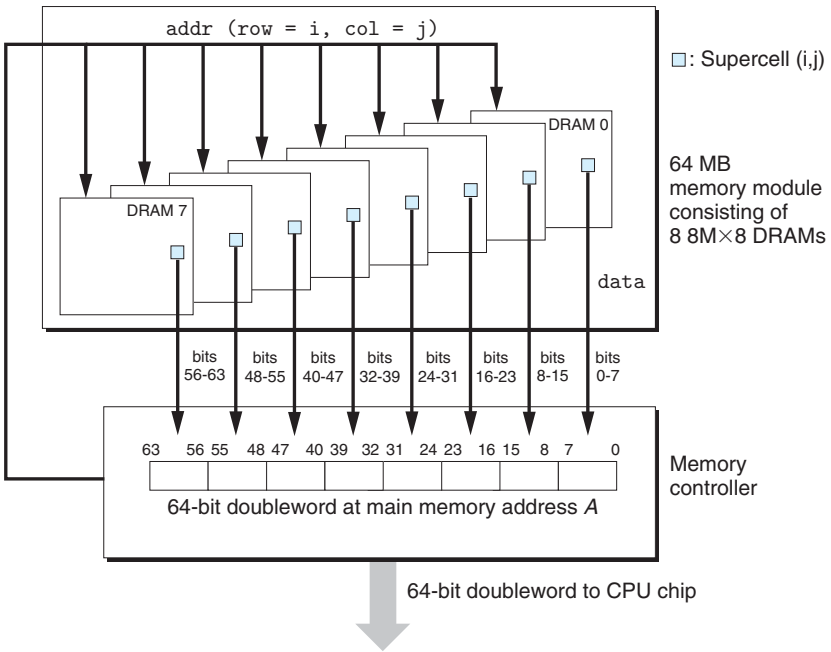


Figure 6.5 **Reading the contents of a memory module.**

Practice Problem 6.1

In the following, let r be the number of rows in a DRAM array, c the number of columns, b_r the number of bits needed to address the rows, and b_c the number of bits needed to address the columns. For each of the following DRAMs, determine the power-of-two array dimensions that minimize $\max(b_r, b_c)$, the maximum number of bits needed to address the rows or columns of the array.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1					
16×4					
128×8					
512×4					
1024×4					

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing

processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

- *Fast page mode DRAM (FPM DRAM).* A conventional DRAM copies an entire row of supercells into its internal row buffer, uses one, and then discards the rest. FPM DRAM improves on this by allowing consecutive accesses to the same row to be served directly from the row buffer. For example, to read four supercells from row i of a conventional DRAM, the memory controller must send four RAS/CAS requests, even though the row address i is identical in each case. To read supercells from the same row of an FPM DRAM, the memory controller sends an initial RAS/CAS request, followed by three CAS requests. The initial RAS/CAS request copies row i into the row buffer and returns the supercell addressed by the CAS. The next three supercells are served directly from the row buffer, and thus more quickly than the initial supercell.
- *Extended data out DRAM (EDO DRAM).* An enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time.
- *Synchronous DRAM (SDRAM).* Conventional, FPM, and EDO DRAMs are asynchronous in the sense that they communicate with the memory controller using a set of explicit control signals. SDRAM replaces many of these control signals with the rising edges of the same external clock signal that drives the memory controller. Without going into detail, the net effect is that an SDRAM can output the contents of its supercells at a faster rate than its asynchronous counterparts.
- *Double Data-Rate Synchronous DRAM (DDR SDRAM).* DDR SDRAM is an enhancement of SDRAM that doubles the speed of the DRAM by using both clock edges as control signals. Different types of DDR SDRAMs are characterized by the size of a small prefetch buffer that increases the effective bandwidth: DDR (2 bits), DDR2 (4 bits), and DDR3 (8 bits).
- *Rambus DRAM (RDRAM).* This is an alternative proprietary technology with a higher maximum bandwidth than DDR SDRAM.
- *Video RAM (VRAM).* Used in the frame buffers of graphics systems. VRAM is similar in spirit to FPM DRAM. Two major differences are that (1) VRAM output is produced by shifting the entire contents of the internal buffer in sequence, and (2) VRAM allows concurrent reads and writes to the memory. Thus, the system can be painting the screen with the pixels in the frame buffer (reads) while concurrently writing new values for the next update (writes).

Aside Historical popularity of DRAM technologies

Until 1995, most PCs were built with FPM DRAMs. From 1996 to 1999, EDO DRAMs dominated the market, while FPM DRAMs all but disappeared. SDRAMs first appeared in 1995 in high-end systems, and by 2002 most PCs were built with SDRAMs and DDR SDRAMs. By 2010, most server and desktop systems were built with DDR3 SDRAMs. In fact, the Intel Core i7 supports only DDR3 SDRAM.

Nonvolatile Memory

DRAMs and SRAMs are *volatile* in the sense that they lose their information if the supply voltage is turned off. *Nonvolatile memories*, on the other hand, retain their information even when they are powered off. There are a variety of nonvolatile memories. For historical reasons, they are referred to collectively as *read-only memories* (ROMs), even though some types of ROMs can be written to as well as read. ROMs are distinguished by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them.

A *programmable ROM* (PROM) can be programmed exactly once. PROMs include a sort of fuse with each memory cell that can be blown once by zapping it with a high current.

An *erasable programmable ROM* (EPROM) has a transparent quartz window that permits light to reach the storage cells. The EPROM cells are cleared to zeros by shining ultraviolet light through the window. Programming an EPROM is done by using a special device to write ones into the EPROM. An EPROM can be erased and reprogrammed on the order of 1000 times. An *electrically erasable PROM* (EEPROM) is akin to an EPROM, but does not require a physically separate programming device, and thus can be reprogrammed in-place on printed circuit cards. An EEPROM can be reprogrammed on the order of 10^5 times before it wears out.

Flash memory is a type of nonvolatile memory, based on EEPROMs, that has become an important storage technology. Flash memories are everywhere, providing fast and durable nonvolatile storage for a slew of electronic devices, including digital cameras, cell phones, music players, PDAs, and laptop, desktop, and server computer systems. In Section 6.1.3, we will look in detail at a new form of flash-based disk drive, known as a *solid state disk* (SSD), that provides a faster, sturdier, and less power-hungry alternative to conventional rotating disks.

Programs stored in ROM devices are often referred to as *firmware*. When a computer system is powered up, it runs firmware stored in a ROM. Some systems provide a small set of primitive input and output functions in firmware, for example, a PC's BIOS (basic input/output system) routines. Complicated devices such as graphics cards and disk drive controllers also rely on firmware to translate I/O (input/output) requests from the CPU.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example,

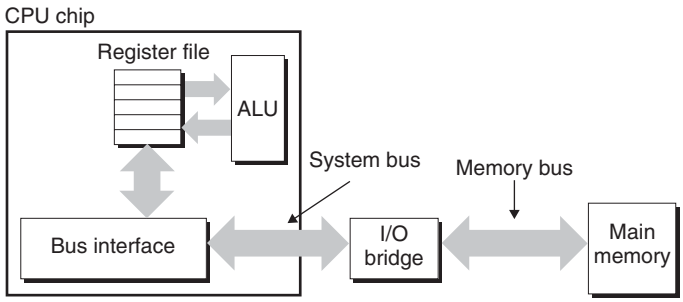


Figure 6.6
Example bus structure that connects the CPU and main memory.

is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of an example computer system. The main components are the CPU chip, a chipset that we will call an *I/O bridge* (which includes the memory controller), and the DRAM memory modules that make up main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O bridge to the main memory.

The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

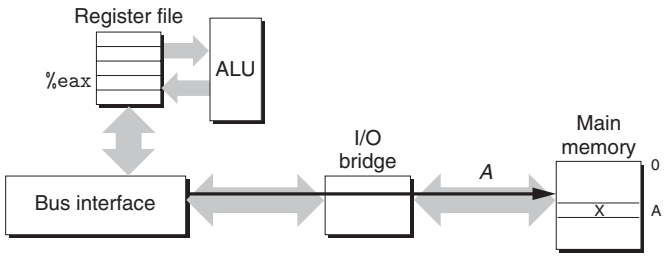
Aside A note on bus designs

Bus design is a complex and rapidly changing aspect of computer systems. Different vendors develop different bus architectures as a way to differentiate their products. For example, Intel systems use chipsets known as the *northbridge* and the *southbridge* to connect the CPU to memory and I/O devices, respectively. In older Pentium and Core 2 systems, a *front side bus* (FSB) connects the CPU to the northbridge. Systems from AMD replace the FSB with the *HyperTransport* interconnect, while newer Intel Core i7 systems use the *QuickPath* interconnect. The details of these different bus architectures are beyond the scope of this text. Instead, we will use the high-level bus architecture from Figure 6.6 as a running example throughout the text. It is a simple but useful abstraction that allows us to be concrete, and captures the main ideas without being tied too closely to the detail of any proprietary designs.

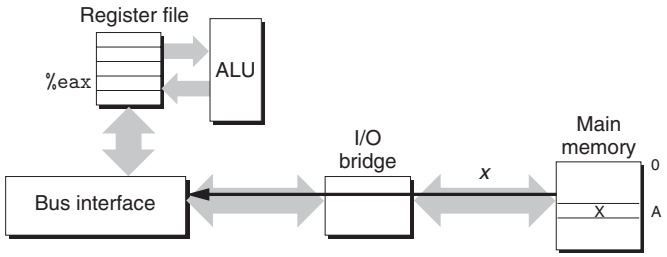
Consider what happens when the CPU performs a load operation such as

```
movl A,%eax
```

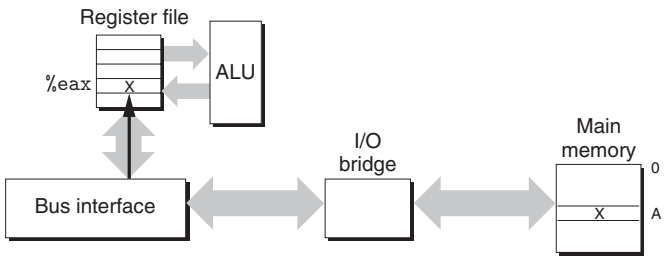
where the contents of address *A* are loaded into register `%eax`. Circuitry on the CPU chip called the *bus interface* initiates a read transaction on the bus. The read transaction consists of three steps. First, the CPU places the address *A* on the system bus. The I/O bridge passes the signal along to the memory bus (Figure 6.7(a)). Next, the main memory senses the address signal on the memory



(a) CPU places address *A* on the memory bus.



(b) Main memory reads *A* from the bus, retrieves word *x*, and places it on the bus.



(c) CPU reads word *x* from the bus, and copies it into register `%eax`.

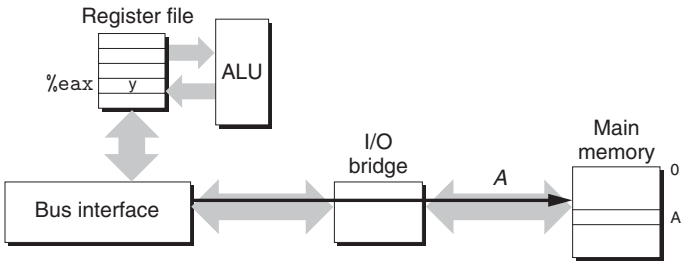
Figure 6.7 **Memory read transaction for a load operation:** `movl A,%eax`.

bus, reads the address from the memory bus, fetches the data word from the DRAM, and writes the data to the memory bus. The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads it from the bus, and copies it to register `%eax` (Figure 6.7(c)).

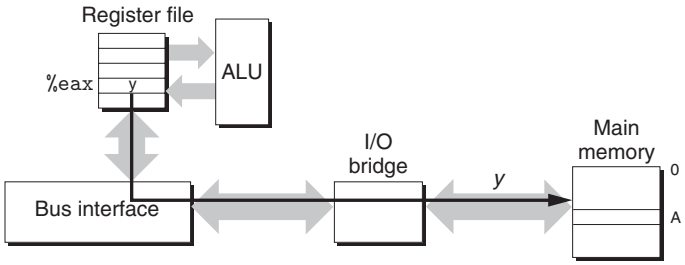
Conversely, when the CPU performs a store instruction such as

```
movl %eax,A
```

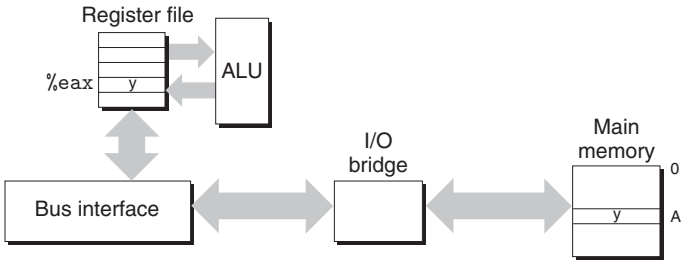
where the contents of register `%eax` are written to address *A*, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data word in `%eax` to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).



(a) CPU places address *A* on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word *y* on the bus.



(c) Main memory reads data word *y* from the bus and stores it at address *A*.

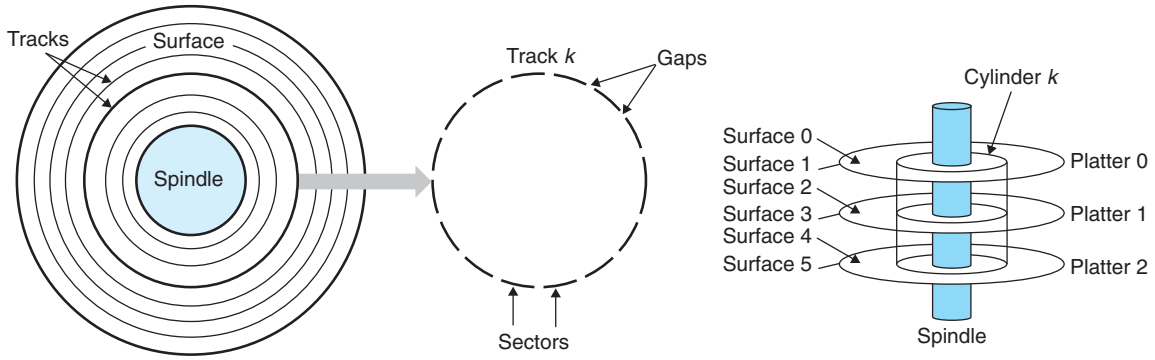
Figure 6.8 **Memory write transaction for a store operation:** `movl %eax,A`.

6.1.2 Disk Storage

Disks are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.

Disk Geometry

Disks are constructed from *platters*. Each platter consists of two sides, or *surfaces*, that are coated with magnetic recording material. A rotating *spindle* in the center of the platter spins the platter at a fixed *rotational rate*, typically between 5400 and



(a) Single-platter view

(b) Multiple-platter view

Figure 6.9 **Disk geometry.**

15,000 *revolutions per minute* (RPM). A disk will typically contain one or more of these platters encased in a sealed container.

Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called *tracks*. Each track is partitioned into a collection of *sectors*. Each sector contains an equal number of data bits (typically 512 bytes) encoded in the magnetic material on the sector. Sectors are separated by *gaps* where no data bits are stored. Gaps store formatting bits that identify sectors.

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a *disk drive*, although we will usually refer to it as simply a *disk*. We will sometime refer to disks as *rotating disks* to distinguish them from flash-based *solid state disks* (SSDs), which have no moving parts.

Disk manufacturers describe the geometry of multiple-platter drives in terms of *cylinders*, where a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. For example, if a drive has three platters and six surfaces, and the tracks on each surface are numbered consistently, then cylinder k is the collection of the six instances of track k.

Disk Capacity

The maximum number of bits that can be recorded by a disk is known as its *maximum capacity*, or simply *capacity*. Disk capacity is determined by the following technology factors:

- *Recording density* (*bits*/*in*): The number of bits that can be squeezed into a 1-inch segment of a track.
- *Track density* (*tracks*/*in*): The number of tracks that can be squeezed into a 1-inch segment of the radius extending from the center of the platter.

- *Areal density* (*bits*/*in*^{2}): The product of the recording density and the track density.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every few years. The original disks, designed in an age of low areal density, partitioned every track into the same number of sectors, which was determined by the number of sectors that could be recorded on the innermost track. To maintain a fixed number of sectors per track, the sectors were spaced farther apart on the outer tracks. This was a reasonable approach when areal densities were relatively low. However, as areal densities increased, the gaps between sectors (where no data bits were stored) became unacceptably large. Thus, modern high-capacity disks use a technique known as *multiple zone recording*, where the set of cylinders is partitioned into disjoint subsets known as *recording zones*. Each zone consists of a contiguous collection of cylinders. Each track in each cylinder in a zone has the same number of sectors, which is determined by the number of sectors that can be packed into the innermost track of the zone. Note that diskettes (floppy disks) still use the old-fashioned approach, with a constant number of sectors per track.

The capacity of a disk is given by the following formula:

$$\text{Disk capacity} = \frac{\text{\# bytes}}{\text{sector}} \times \frac{\text{average \# sectors}}{\text{track}} \times \frac{\text{\# tracks}}{\text{surface}} \times \frac{\text{\# surfaces}}{\text{platter}} \times \frac{\text{\# platters}}{\text{disk}}$$

For example, suppose we have a disk with 5 platters, 512 bytes per sector, 20,000 tracks per surface, and an average of 300 sectors per track. Then the capacity of the disk is:

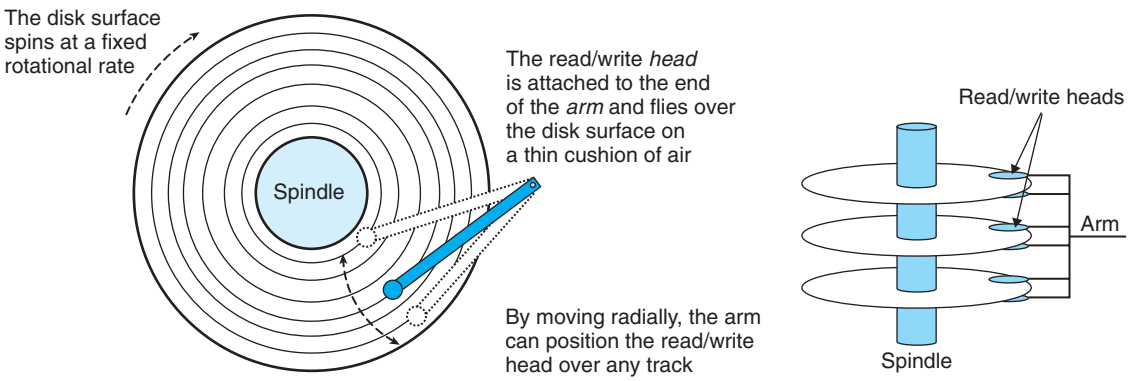
$$\begin{aligned}\text{Disk capacity} &= \frac{512 \text{ bytes}}{\text{sector}} \times \frac{300 \text{ sectors}}{\text{track}} \times \frac{20{,}000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{5 \text{ platters}}{\text{disk}} \\ &= 30{,}720{,}000{,}000 \text{ bytes} \\ &= 30.72 \text{ GB}.\end{aligned}$$

Notice that manufacturers express disk capacity in units of gigabytes (GB), where $1\,\text{GB} = 10^9$ bytes.

Aside How much is a gigabyte?

Unfortunately, the meanings of prefixes such as kilo (K), mega (M), giga (G), and tera (T) depend on the context. For measures that relate to the capacity of DRAMs and SRAMs, typically $K = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$, and $T = 2^{40}$. For measures related to the capacity of I/O devices such as disks and networks, typically $K = 10^3$, $M = 10^6$, $G = 10^9$, and $T = 10^{12}$. Rates and throughputs usually use these prefix values as well.

Fortunately, for the back-of-the-envelope estimates that we typically rely on, either assumption works fine in practice. For example, the relative difference between $2^{20} = 1{,}048{,}576$ and $10^6 = 1{,}000{,}000$ is small: $(2^{20} - 10^6)/10^6 \approx 5\%$. Similarly for $2^{30} = 1{,}073{,}741{,}824$ and $10^9 = 1{,}000{,}000{,}000$: $(2^{30} - 10^9)/10^9 \approx 7\%$.



(a) Single-platter view

(b) Multiple-platter view

Figure 6.10 **Disk dynamics.**

Practice Problem 6.2

What is the capacity of a disk with two platters, 10,000 cylinders, an average of 400 sectors per track, and 512 bytes per sector?

Disk Operation

Disks read and write bits stored on the magnetic surface using a *read/write head* connected to the end of an *actuator arm*, as shown in Figure 6.10(a). By moving the arm back and forth along its radial axis, the drive can position the head over any track on the surface. This mechanical motion is known as a *seek*. Once the head is positioned over the desired track, then as each bit on the track passes underneath, the head can either sense the value of the bit (read the bit) or alter the value of the bit (write the bit). Disks with multiple platters have a separate read/write head for each surface, as shown in Figure 6.10(b). The heads are lined up vertically and move in unison. At any point in time, all heads are positioned on the same cylinder.

The read/write head at the end of the arm flies (literally) on a thin cushion of air over the disk surface at a height of about 0.1 microns and a speed of about 80 km/h. This is analogous to placing the Sears Tower on its side and flying it around the world at a height of 2.5 cm (1 inch) above the ground, with each orbit of the earth taking only 8 seconds! At these tolerances, a tiny piece of dust on the surface is like a huge boulder. If the head were to strike one of these boulders, the head would cease flying and crash into the surface (a so-called *head crash*). For this reason, disks are always sealed in airtight packages.

Disks read and write data in sector-sized blocks. The *access time* for a sector has three main components: *seek time*, *rotational latency*, and *transfer time*:

- **Seek time:** To read the contents of some target sector, the arm first positions the head over the track that contains the target sector. The time required to move the arm is called the *seek time*. The seek time, T_{seek}, depends on the previous position of the head and the speed that the arm moves across the surface. The average seek time in modern drives, $T_{avg\ seek}$, measured by taking the mean of several thousand seeks to random sectors, is typically on the order of 3 to 9 ms. The maximum time for a single seek, $T_{max\ seek}$, can be as high as 20 ms.
- **Rotational latency:** Once the head is in position over the track, the drive waits for the first bit of the target sector to pass under the head. The performance of this step depends on both the position of the surface when the head arrives at the target sector and the rotational speed of the disk. In the worst case, the head just misses the target sector and waits for the disk to make a full rotation. Thus, the maximum rotational latency, in seconds, is given by

$$T_{max\ rotation} = \frac{1}{\text{RPM}} \times \frac{60\ \text{secs}}{1\ \text{min}}$$

  The average rotational latency, $T_{avg\ rotation}$, is simply half of $T_{max\ rotation}$.
- **Transfer time:** When the first bit of the target sector is under the head, the drive can begin to read or write the contents of the sector. The transfer time for one sector depends on the rotational speed and the number of sectors per track. Thus, we can roughly estimate the average transfer time for one sector in seconds as

$$T_{avg\ transfer} = \frac{1}{\text{RPM}} \times \frac{1}{\text{(average \# sectors/track)}} \times \frac{60\ \text{secs}}{1\ \text{min}}$$

We can estimate the average time to access the contents of a disk sector as the sum of the average seek time, the average rotational latency, and the average transfer time. For example, consider a disk with the following parameters:

Parameter	Value
Rotational rate	7200 RPM
$T_{avg\ seek}$	9 ms
Average # sectors/track	400

For this disk, the average rotational latency (in ms) is

$$\begin{aligned} T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60\ \text{secs} / 7200\ \text{RPM}) \times 1000\ \text{ms/sec} \\ &\approx 4\ \text{ms} \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\ transfer} &= 60 / 7200\ \text{RPM} \times 1 / 400\ \text{sectors/track} \times 1000\ \text{ms/sec} \\ &\approx 0.02\ \text{ms} \end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned} T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\ &= 9\ \text{ms} + 4\ \text{ms} + 0.02\ \text{ms} \\ &= 13.02\ \text{ms} \end{aligned}$$

This example illustrates some important points:

- The time to access the 512 bytes in a disk sector is dominated by the seek time and the rotational latency. Accessing the first byte in the sector takes a long time, but the remaining bytes are essentially free.
- Since the seek time and rotational latency are roughly the same, twice the seek time is a simple and reasonable rule for estimating disk access time.
- The access time for a doubleword stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-sized block from memory is roughly 256 ns for SRAM and 4000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2500 times greater than DRAM. The difference in access times is even more dramatic if we compare the times to access a single word.

Practice Problem 6.3

Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational rate	15,000 RPM
$T_{avg\ seek}$	8 ms
Average # sectors/track	500

Logical Disk Blocks

As we have seen, modern disks have complex geometries, with multiple surfaces and different recording zones on those surfaces. To hide this complexity from the operating system, modern disks present a simpler view of their geometry as a sequence of B sector-sized *logical blocks*, numbered $0, 1, \ldots, B-1$. A small hardware/firmware device in the disk package, called the *disk controller*, maintains the mapping between logical block numbers and actual (physical) disk sectors.

When the operating system wants to perform an I/O operation such as reading a disk sector into main memory, it sends a command to the disk controller asking it to read a particular logical block number. Firmware on the controller performs a fast table lookup that translates the logical block number into a *(surface, track, sector)* triple that uniquely identifies the corresponding physical sector. Hardware on the controller interprets this triple to move the heads to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed

by the head into a small memory buffer on the controller, and copies them into main memory.

Aside Formatted disk capacity

Before a disk can be used to store data, it must be *formatted* by the disk controller. This involves filling in the gaps between sectors with information that identifies the sectors, identifying any cylinders with surface defects and taking them out of action, and setting aside a set of cylinders in each zone as spares that can be called into action if one or more cylinders in the zone goes bad during the lifetime of the disk. The *formatted capacity* quoted by disk manufacturers is less than the maximum capacity because of the existence of these spare cylinders.

Practice Problem 6.4

Suppose that a 1 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Parameter	Value
Rotational rate	10,000 RPM
$T_{avg\ seek}$	5 ms
Average # sectors/track	1000
Surfaces	4
Sector size	512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{avg\ seek} + T_{avg\ rotation}$.

A. *Best case:* Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential).

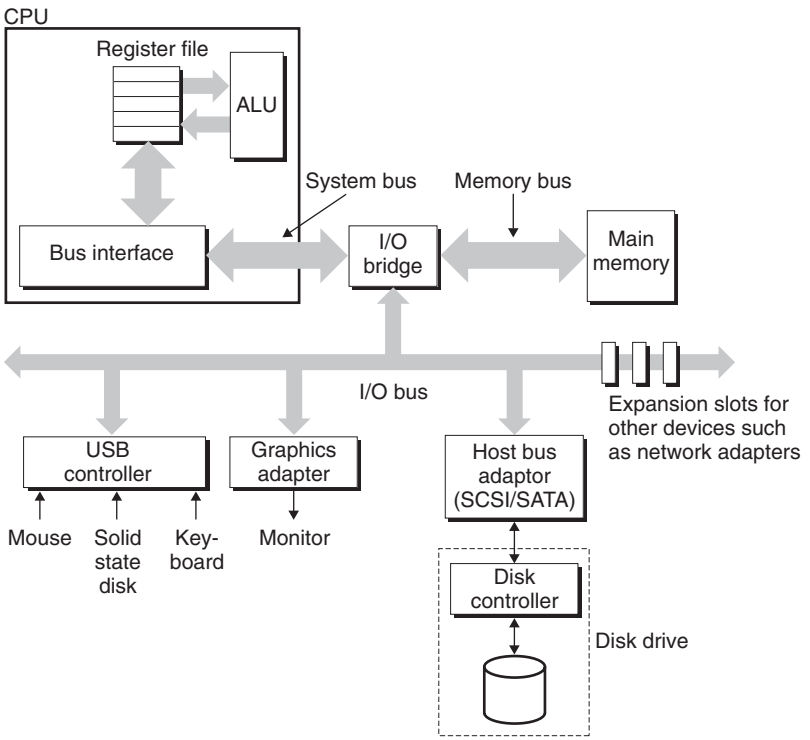
B. *Random case:* Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

Connecting I/O Devices

Input/output (I/O) devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an *I/O bus* such as Intel's *Peripheral Component Interconnect* (PCI) bus. Unlike the system bus and memory buses, which are CPU-specific, I/O buses such as PCI are designed to be independent of the underlying CPU. For example, PCs and Macs both incorporate the PCI bus. Figure 6.11 shows a typical I/O bus structure (modeled on PCI) that connects the CPU, main memory, and I/O devices.

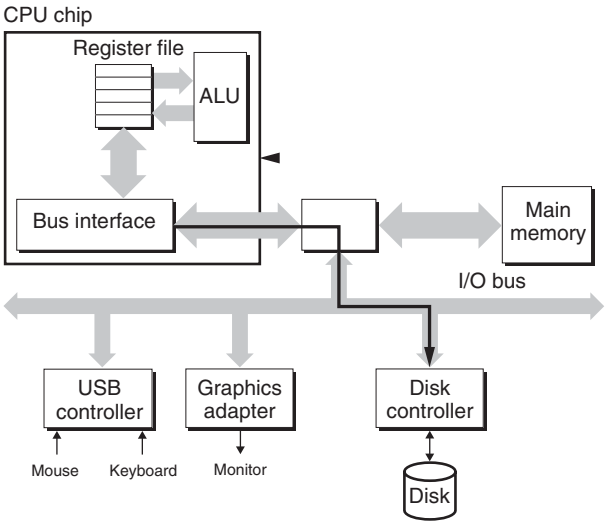
Although the I/O bus is slower than the system and memory buses, it can accommodate a wide variety of third-party I/O devices. For example, the bus in Figure 6.11 has three different types of devices attached to it.

Figure 6.11
Example bus structure that connects the CPU, main memory, and I/O devices.

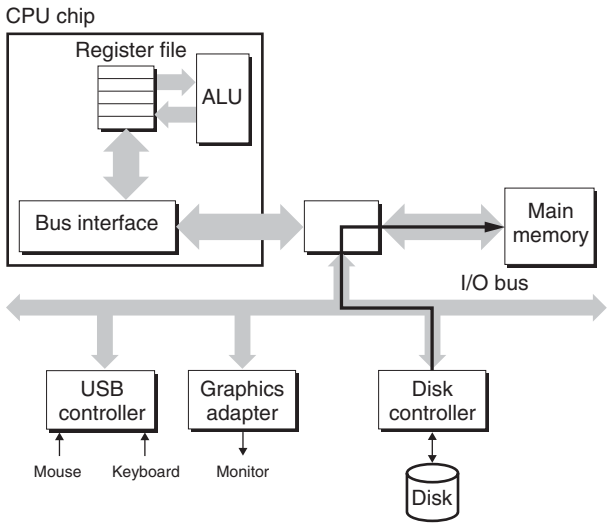


- A *Universal Serial Bus* (USB) controller is a conduit for devices attached to a USB bus, which is a wildly popular standard for connecting a variety of peripheral I/O devices, including keyboards, mice, modems, digital cameras, game controllers, printers, external disk drives, and solid state disks. USB 2.0 buses have a maximum bandwidth of 60 MB/s. USB 3.0 buses have a maximum bandwidth of 600 MB/s.
- A *graphics card* (or *adapter*) contains hardware and software logic that is responsible for painting the pixels on the display monitor on behalf of the CPU.
- A *host bus adapter* that connects one or more disks to the I/O bus using a communication protocol defined by a particular *host bus interface*. The two most popular such interfaces for disks are *SCSI* (pronounced "scuzzy") and *SATA* (pronounced "sat-uh"). SCSI disks are typically faster and more expensive than SATA drives. A SCSI host bus adapter (often called a *SCSI controller*) can support multiple disk drives, as opposed to SATA adapters, which can only support one drive.

Additional devices such as *network adapters* can be attached to the I/O bus by plugging the adapter into empty *expansion slots* on the motherboard that provide a direct electrical connection to the bus.



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



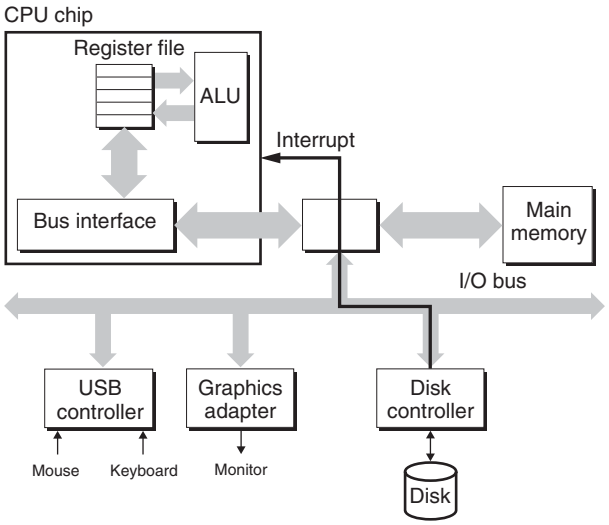
(b) The disk controller reads the sector and performs a DMA transfer into main memory.

Figure 6.12 **Reading a disk sector.**

Accessing Disks

While a detailed description of how I/O devices work and how they are programmed is outside our scope here, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.

The CPU issues commands to I/O devices using a technique called *memory-mapped I/O* (Figure 6.12(a)). In a system with memory-mapped I/O, a block of



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

Figure 6.12 *(continued)* **Reading a disk sector.**

addresses in the address space is reserved for communicating with I/O devices. Each of these addresses is known as an *I/O port*. Each device is associated with (or mapped to) one or more ports when it is attached to the bus.

As a simple example, suppose that the disk controller is mapped to port `0xa0`. Then the CPU might initiate a disk read by executing three store instructions to address `0xa0`: The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished. (We will discuss interrupts in Section 8.1.) The second instruction indicates the logical block number that should be read. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is taking place would be enormously wasteful.

After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU (Figure 6.12(b)). This process, whereby a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as *direct memory access* (DMA). The transfer of data is known as a *DMA transfer*.

After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is

Geometry attribute	Value
Platters	4
Surfaces (read/write heads)	8
Surface diameter	3.5 in.
Sector size	512 bytes
Zones	15
Cylinders	50,864
Recording density (max)	628,000 bits/in.
Track density	85,000 tracks/in.
Areal density (max)	53.4 Gbits/sq. in.
Formatted capacity	146.8 GB

Performance attribute	Value
Rotational rate	15,000 RPM
Avg. rotational latency	2 ms
Avg. seek time	4 ms
Sustained transfer rate	58–96 MB/s

Figure 6.13 **Seagate Cheetah 15K.4 geometry and performance.** Source: `www.seagate.com`.

currently working on and jump to an operating system routine. The routine records the fact that the I/O has finished and then returns control to the point where the CPU was interrupted.

Anatomy of a Commercial Disk

Disk manufacturers publish a lot of useful high-level technical information on their Web pages. For example, the Cheetah 15K.4 is a SCSI disk first manufactured by Seagate in 2005. If we consult the online product manual on the Seagate Web page, we can glean the geometry and performance information shown in Figure 6.13.

Disk manufacturers rarely publish detailed technical information about the geometry of the individual recording zones. However, storage researchers at Carnegie Mellon University have developed a useful tool, called DIXtrac, that automatically discovers a wealth of low-level information about the geometry and performance of SCSI disks [92]. For example, DIXtrac is able to discover the detailed zone geometry of our example Seagate disk, which we've shown in Figure 6.14. Each row in the table characterizes one of the 15 zones. The first column gives the zone number, with zone 0 being the outermost and zone 14 the innermost. The second column gives the number of sectors contained in each track in that zone. The third column shows the number of cylinders assigned to that zone, where each cylinder consists of eight tracks, one from each surface. Similarly, the fourth column gives the total number of logical blocks assigned to each zone, across all eight surfaces. (The tool was not able to extract valid data for the innermost zone, so these are omitted.)

The zone map reveals some interesting facts about the Seagate disk. First, more sectors are packed into the outer zones (which have a larger circumference) than the inner zones. Second, each zone has more sectors than logical blocks

Zone number	Sectors per track	Cylinders per zone	Logical blocks per zone
(outer) 0	864	3201	22,076,928
1	844	3200	21,559,136
2	816	3400	22,149,504
3	806	3100	19,943,664
4	795	3100	19,671,480
5	768	3400	20,852,736
6	768	3450	21,159,936
7	725	3650	21,135,200
8	704	3700	20,804,608
9	672	3700	19,858,944
10	640	3700	18,913,280
11	603	3700	17,819,856
12	576	3707	17,054,208
13	528	3060	12,900,096
(inner) 14	—	—	—

Figure 6.14 **Seagate Cheetah 15K.4 zone map.** Source: DIXtrac automatic disk drive characterization tool [92]. Data for zone 14 not available.

(check this yourself). These *spare sectors* form a pool of *spare cylinders*. If the recording material on a sector goes bad, the disk controller will automatically remap the logical blocks on that cylinder to an available spare. So we see that the notion of a logical block not only provides a simpler interface to the operating system, it also provides a level of indirection that enables the disk to be more robust. This general idea of indirection is very powerful, as we will see when we study virtual memory in Chapter 9.

Practice Problem 6.5

Use the zone map in Figure 6.14 to determine the number of spare cylinders in the following zones:

A. Zone 0

B. Zone 8

6.1.3 Solid State Disks

A *solid state disk* (SSD) is a storage technology, based on flash memory (Section 6.1.1), that in some situations is an attractive alternative to the conventional rotating disk. Figure 6.15 shows the basic idea. An SSD package plugs into a standard disk slot on the I/O bus (typically USB or SATA) and behaves like any other

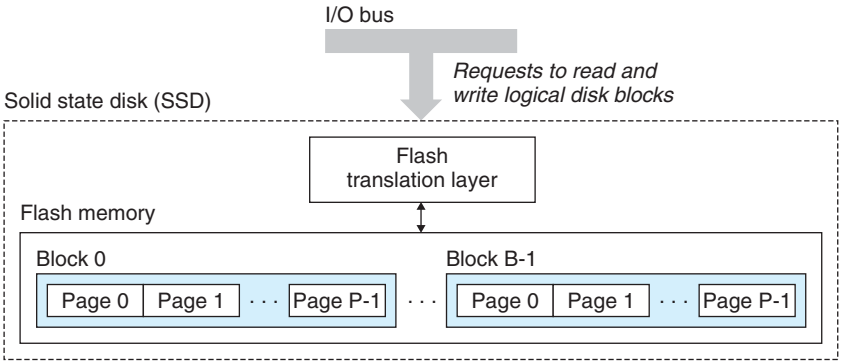


Figure 6.15 **Solid state disk (SSD).**

Reads		Writes	
Sequential read throughput	250 MB/s	Sequential write throughput	170 MB/s
Random read throughput	140 MB/s	Random write throughput	14 MB/s
Random read access time	30 μs	Random write access time	300 μs

Figure 6.16 **Performance characteristics of a typical solid state disk.** Source: Intel X25-E SATA solid state drive product manual.

disk, processing requests from the CPU to read and write logical disk blocks. An SSD package consists of one or more flash memory chips, which replace the mechanical drive in a conventional rotating disk, and a *flash translation layer*, which is a hardware/firmware device that plays the same role as a disk controller, translating requests for logical blocks into accesses of the underlying physical device.

SSDs have different performance characteristics than rotating disks. As shown in Figure 6.16, sequential reads and writes (where the CPU accesses logical disk blocks in sequential order) have comparable performance, with sequential reading somewhat faster than sequential writing. However, when logical blocks are accessed in random order, writing is an order of magnitude slower than reading.

The difference between random reading and writing performance is caused by a fundamental property of the underlying flash memory. As shown in Figure 6.15, a flash memory consists of a sequence of B *blocks*, where each block consists of P pages. Typically, pages are 512–4KB in size, and a block consists of 32–128 pages, with total block sizes ranging from 16 KB to 512 KB. Data is read and written in units of pages. A page can be written only after the entire block to which it belongs has been *erased* (typically this means that all bits in the block are set to 1). However, once a block is erased, each page in the block can be written once with no further erasing. A blocks wears out after roughly 100,000 repeated writes. Once a block wears out it can no longer be used.

Random writes are slow for two reasons. First, erasing a block takes a relatively long time, on the order of 1 ms, which is more than an order of magnitude longer than it takes to access a page. Second, if a write operation attempts to modify a page p that contains existing data (i.e., not all ones), then any pages in the same block with useful data must be copied to a new (erased) block before the write to page p can occur. Manufacturers have developed sophisticated logic in the flash translation layer that attempts to amortize the high cost of erasing blocks and to minimize the number of internal copies on writes, but it is unlikely that random writing will ever perform as well as reading.

SSDs have a number of advantages over rotating disks. They are built of semiconductor memory, with no moving parts, and thus have much faster random access times than rotating disks, use less power, and are more rugged. However, there are some disadvantages. First, because flash blocks wear out after repeated writes, SSDs have the potential to wear out as well. *Wear leveling* logic in the flash translation layer attempts to maximize the lifetime of each block by spreading erasures evenly across all blocks, but the fundamental limit remains. Second, SSDs are about 100 times more expensive per byte than rotating disks, and thus the typical storage capacities are 100 times less than rotating disks. However, SSD prices are decreasing rapidly as they become more popular, and the gap between the two appears to be decreasing.

SSDs have completely replaced rotating disks in portable music devices, are popular as disk replacements in laptops, and have even begun to appear in desktops and servers. While rotating disks are here to stay, it is clear that SSDs are an important new storage technology.

Practice Problem 6.6

As we have seen, a potential drawback of SSDs is that the underlying flash memory can wear out. For example, one major manufacturer guarantees 1 petabyte (10^{15} bytes) of random writes for their SSDs before they wear out. Given this assumption, estimate the lifetime (in years) of the SSD in Figure 6.16 for the following workloads:

A. *Worst case for sequential writes:* The SSD is written to continuously at a rate of 170 MB/s (the average sequential write throughput of the device).

B. *Worst case for random writes:* The SSD is written to continuously at a rate of 14 MB/s (the average random write throughput of the device).

C. *Average case:* The SSD is written to at a rate of 20 GB/day (the average daily write rate assumed by some computer manufacturers in their mobile computer workload simulations).

6.1.4 Storage Technology Trends

There are several important concepts to take away from our discussion of storage technologies.

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
$/MB	19,200	2900	320	256	100	75	60	320
Access (ns)	300	150	35	15	3	2	1.5	200

(a) SRAM trends

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
$/MB	8000	880	100	30	1	.1	0.06	130,000
Access (ns)	375	200	100	70	60	50	40	9
Typical size (MB)	0.064	0.256	4	16	64	2000	8,000	125,000

(b) DRAM trends

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
$/MB	500	100	8	0.30	0.01	0.005	0.0003	1,600,000
Seek time (ms)	87	75	28	10	8	5	3	29
Typical size (MB)	1	10	160	1000	20,000	160,000	1,500,000	1,500,000

(c) Rotating disk trends

Metric	1980	1985	1990	1995	2000	2003	2005	2010	2010:1980
Intel CPU	8080	80286	80386	Pent.	P-III	Pent. 4	Core 2	Core i7	—
Clock rate (MHz)	1	6	20	150	600	3300	2000	2500	2500
Cycle time (ns)	1000	166	50	6	1.6	0.30	0.50	0.4	2500
Cores	1	1	1	1	1	1	2	4	4
Eff. cycle time (ns)	1000	166	50	6	1.6	0.30	0.25	0.10	10,000

(d) CPU trends

Figure 6.17 **Storage and processing technology trends.**

Different storage technologies have different price and performance trade-offs. SRAM is somewhat faster than DRAM, and DRAM is much faster than disk. On the other hand, fast storage is always more expensive than slower storage. SRAM costs more per byte than DRAM. DRAM costs much more than disk. SSDs split the difference between DRAM and rotating disk.

The price and performance properties of different storage technologies are changing at dramatically different rates. Figure 6.17 summarizes the price and performance properties of storage technologies since 1980, when the first PCs were introduced. The numbers were culled from back issues of trade magazines and the Web. Although they were collected in an informal survey, the numbers reveal some interesting trends.

Since 1980, both the cost and performance of SRAM technology have improved at roughly the same rate. Access times have decreased by a factor of about 200 and cost per megabyte by a factor of 300 (Figure 6.17(a)). However, the trends

for DRAM and disk are much more dramatic and divergent. While the cost per megabyte of DRAM has decreased by a factor of 130,000 (more than five orders of magnitude!), DRAM access times have decreased by only a factor of 10 or so (Figure 6.17(b)). Disk technology has followed the same trend as DRAM and in even more dramatic fashion. While the cost of a megabyte of disk storage has plummeted by a factor of more than 1,000,000 (more than six orders of magnitude!) since 1980, access times have improved much more slowly, by only a factor of 30 or so (Figure 6.17(c)). These startling long-term trends highlight a basic truth of memory and disk technology: it is easier to increase density (and thereby reduce cost) than to decrease access time.

DRAM and disk performance are lagging behind CPU performance. As we see in Figure 6.17(d), CPU cycle times improved by a factor of 2500 between 1980 and 2010. If we look at the *effective cycle time*—which we define to be the cycle time of an individual CPU (processor) divided by the number of its processor cores—then the improvement between 1980 and 2010 is even greater, a factor of 10,000. The split in the CPU performance curve around 2003 reflects the introduction of multi-core processors (see aside on next page). After this split, cycle times of individual cores actually increased a bit before starting to decrease again, albeit at a slower rate than before.

Note that while SRAM performance lags, it is roughly keeping up. However, the gap between DRAM and disk performance and CPU performance is actually widening. Until the advent of multi-core processors around 2003, this performance gap was a function of latency, with DRAM and disk access times increasing more slowly than the cycle time of an individual processor. However, with the introduction of multiple cores, this performance gap is increasingly a function of throughput, with multiple processor cores issuing requests to the DRAM and disk in parallel.

The various trends are shown quite clearly in Figure 6.18, which plots the access and cycle times from Figure 6.17 on a semi-log scale.

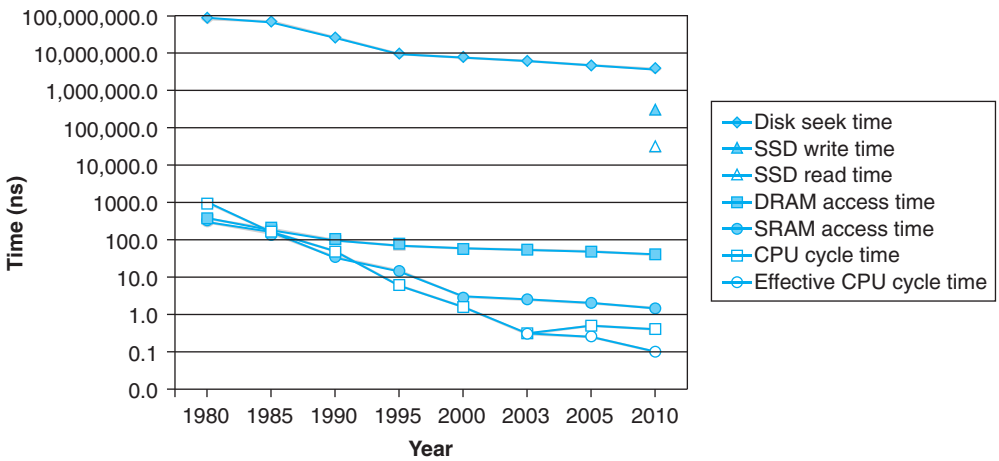


Figure 6.18 **The increasing gap between disk, DRAM, and CPU speeds.**

As we will see in Section 6.4, modern computers make heavy use of SRAM-based caches to try to bridge the processor-memory gap. This approach works because of a fundamental property of application programs known as *locality*, which we discuss next.

Aside When cycle time stood still: the advent of multi-core processors

The history of computers is marked by some singular events that caused profound changes in the industry and the world. Interestingly, these inflection points tend to occur about once per decade: the development of Fortran in the 1950s, the introduction of the IBM 360 in the early 1960s, the dawn of the Internet (then called ARPANET) in the early 1970s, the introduction of the IBM PC in the early 1980s, and the creation of the World Wide Web in the early 1990s.

The most recent such event occurred early in the 21st century, when computer manufacturers ran headlong into the so-called "power wall," discovering that they could no longer increase CPU clock frequencies as quickly because the chips would then consume too much power. The solution was to improve performance by replacing a single large processor with multiple smaller processor *cores*, each a complete processor capable of executing programs independently and in parallel with the other cores. This *multi-core* approach works in part because the power consumed by a processor is proportional to $P = fCv^2$, where f is the clock frequency, C is the capacitance, and v is the voltage. The capacitance C is roughly proportional to the area, so the power drawn by multiple cores can be held constant as long as the total area of the cores is constant. As long as feature sizes continue to shrink at the exponential Moore's law rate, the number of cores in each processor, and thus its effective performance, will continue to increase.

From this point forward, computers will get faster not because the clock frequency increases, but because the number of cores in each processor increases, and because architectural innovations increase the efficiency of programs running on those cores. We can see this trend clearly in Figure 6.18. CPU cycle time reached its lowest point in 2003 and then actually started to rise before leveling off and starting to decline again at a slower rate than before. However, because of the advent of multi-core processors (dual-core in 2004 and quad-core in 2007), the effective cycle time continues to decrease at close to its previous rate.

Practice Problem 6.7

Using the data from the years 2000 to 2010 in Figure 6.17(c), estimate the year when you will be able to buy a petabyte (10^{15} bytes) of rotating disk storage for $500. Assume constant dollars (no inflation).

6.2 Locality

Well-written computer programs tend to exhibit good *locality*. That is, they tend to reference data items that are near other recently referenced data items, or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

Locality is typically described as having two distinct forms: *temporal locality* and *spatial locality*. In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Programmers should understand the principle of locality because, in general, *programs with good locality run faster than programs with poor locality*. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality. At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as *cache memories* that hold blocks of the most recently referenced instructions and data items. At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system. The principle of locality also plays a crucial role in the design of application programs. For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk. High-volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

6.2.1 Locality of References to Program Data

Consider the simple function in Figure 6.19(a) that sums the elements of a vector. Does this function have good locality? To answer this question, we look at the reference pattern for each variable. In this example, the `sum` variable is referenced once in each loop iteration, and thus there is good temporal locality with respect to `sum`. On the other hand, since `sum` is a scalar, there is no spatial locality with respect to `sum`.

As we see in Figure 6.19(b), the elements of vector `v` are read sequentially, one after the other, in the order they are stored in memory (we assume for convenience that the array starts at address 0). Thus, with respect to variable `v`, the function has good spatial locality but poor temporal locality since each vector element

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

(a)

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

(b)

Figure 6.19 **(a) A function with good locality. (b) Reference pattern for vector `v` ($N = 8$).** Notice how the vector elements are accessed in the same order that they are stored in memory.

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9   }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

(b)

Figure 6.20 **(a) Another function with good locality. (b) Reference pattern for array** `a` **(*M* = 2, *N* = 3).** There is good spatial locality because the array is accessed in the same row-major order in which it is stored in memory.

is accessed exactly once. Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the `sumvec` function enjoys good locality.

A function such as `sumvec` that visits each element of a vector sequentially is said to have a *stride-1 reference pattern* (with respect to the element size). We will sometimes refer to stride-1 reference patterns as *sequential reference patterns*. Visiting every kth element of a contiguous vector is called a *stride-k reference pattern*. Stride-1 reference patterns are a common and important source of spatial locality in programs. In general, as the stride increases, the spatial locality decreases.

Stride is also an important issue for programs that reference multidimensional arrays. For example, consider the `sumarrayrows` function in Figure 6.20(a) that sums the elements of a two-dimensional array. The doubly nested loop reads the elements of the array in *row-major order*. That is, the inner loop reads the elements of the first row, then the second row, and so on. The `sumarrayrows` function enjoys good spatial locality because it references the array in the same row-major order that the array is stored (Figure 6.20(b)). The result is a nice stride-1 reference pattern with excellent spatial locality.

Seemingly trivial changes to a program can have a big impact on its locality. For example, the `sumarraycols` function in Figure 6.21(a) computes the same result as the `sumarrayrows` function in Figure 6.20(a). The only difference is that we have interchanged the i and j loops. What impact does interchanging the loops have on its locality? The `sumarraycols` function suffers from poor spatial locality because it scans the array column-wise instead of row-wise. Since C arrays are laid out in memory row-wise, the result is a stride-N reference pattern, as shown in Figure 6.21(b).

6.2.2 Locality of Instruction Fetches

Since program instructions are stored in memory and must be fetched (read) by the CPU, we can also evaluate the locality of a program with respect to its instruction fetches. For example, in Figure 6.19 the instructions in the body of the

```
1   int sumarraycols(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (j = 0; j < N; j++)
6           for (i = 0; i < M; i++)
7               sum += a[i][j];
8       return sum;
9   }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

(b)

Figure 6.21 **(a) A function with poor spatial locality. (b) Reference pattern for array** `a` **($M = 2$, $N = 3$).** The function has poor spatial locality because it scans memory with a stride-N reference pattern.

`for` loop are executed in sequential memory order, and thus the loop enjoys good spatial locality. Since the loop body is executed multiple times, it also enjoys good temporal locality.

An important property of code that distinguishes it from program data is that it is rarely modified at run time. While a program is executing, the CPU reads its instructions from memory. The CPU rarely overwrites or modifies these instructions.

6.2.3 Summary of Locality

In this section, we have introduced the fundamental idea of locality and have identified some simple rules for qualitatively evaluating the locality in a program:

- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-k reference patterns, the smaller the stride the better the spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.
- Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.

Later in this chapter, after we have learned about cache memories and how they work, we will show you how to quantify the idea of locality in terms of cache hits and misses. It will also become clear to you why programs with good locality typically run faster than programs with poor locality. Nonetheless, knowing how to

glance at a source code and getting a high-level feel for the locality in the program is a useful and important skill for a programmer to master.

Practice Problem 6.8

Permute the loops in the following function so that it scans the three-dimensional array *a* with a stride-1 reference pattern.

```
1   int sumarray3d(int a[N][N][N])
2   {
3       int i, j, k, sum = 0;
4
5       for (i = 0; i < N; i++) {
6           for (j = 0; j < N; j++) {
7               for (k = 0; k < N; k++) {
8                   sum += a[k][i][j];
9               }
10          }
11      }
12      return sum;
13  }
```

Practice Problem 6.9

The three functions in Figure 6.22 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

(a) An array of `structs`

```
1   #define N 1000
2
3   typedef struct {
4       int vel[3];
5       int acc[3];
6   } point;
7
8   point p[N];
```

(b) The `clear1` function

```
1   void clear1(point *p, int n)
2   {
3       int i, j;
4
5       for (i = 0; i < n; i++) {
6           for (j = 0; j < 3; j++)
7               p[i].vel[j] = 0;
8           for (j = 0; j < 3; j++)
9               p[i].acc[j] = 0;
10      }
11  }
```

Figure 6.22 **Code examples for Practice Problem 6.9.**

(c) The `clear2` function

```
1   void clear2(point *p, int n)
2   {
3       int i, j;
4
5       for (i = 0; i < n; i++) {
6           for (j = 0; j < 3; j++) {
7               p[i].vel[j] = 0;
8               p[i].acc[j] = 0;
9           }
10      }
11  }
```

(d) The `clear3` function

```
1   void clear3(point *p, int n)
2   {
3       int i, j;
4
5       for (j = 0; j < 3; j++) {
6           for (i = 0; i < n; i++)
7               p[i].vel[j] = 0;
8           for (i = 0; i < n; i++)
9               p[i].acc[j] = 0;
10      }
11  }
```

Figure 6.22 *(continued)* **Code examples for Practice Problem 6.9.**

6.3 The Memory Hierarchy

Sections 6.1 and 6.2 described some fundamental and enduring properties of storage technology and computer software:

- *Storage technology:* Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.
- *Computer software:* Well-written programs tend to exhibit good locality.

In one of the happier coincidences of computing, these fundamental properties of hardware and software complement each other beautifully. Their complementary nature suggests an approach for organizing memory systems, known as the *memory hierarchy*, that is used in all modern computer systems. Figure 6.23 shows a typical memory hierarchy. In general, the storage devices get slower, cheaper, and larger as we move from higher to lower *levels*. At the highest level (L0) are a small number of fast CPU registers that the CPU can access in a single clock cycle. Next are one or more small to moderate-sized SRAM-based cache memories that can be accessed in a few CPU clock cycles. These are followed by a large DRAM-based main memory that can be accessed in tens to hundreds of clock cycles. Next are slow but enormous local disks. Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network. For example, distributed file systems such as the Andrew File System (AFS) or the Network File System (NFS) allow a program to access files that are stored on remote network-connected servers. Similarly, the World Wide Web allows programs to access remote files stored on Web servers anywhere in the world.

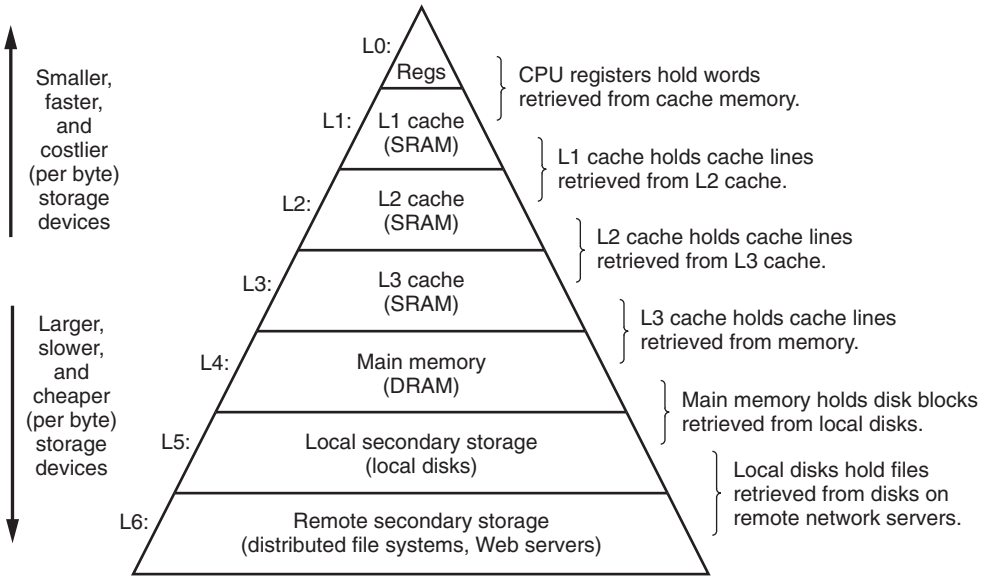


Figure 6.23 **The memory hierarchy.**

Aside Other memory hierarchies

We have shown you one example of a memory hierarchy, but other combinations are possible, and indeed common. For example, many sites back up local disks onto archival magnetic tapes. At some of these sites, human operators manually mount the tapes onto tape drives as needed. At other sites, tape robots handle this task automatically. In either case, the collection of tapes represents a level in the memory hierarchy, below the local disk level, and the same general principles apply. Tapes are cheaper per byte than disks, which allows sites to archive multiple snapshots of their local disks. The trade-off is that tapes take longer to access than disks. As another example, solid state disks are playing an increasingly important role in the memory hierarchy, bridging the gulf between DRAM and rotating disk.

6.3.1 Caching in the Memory Hierarchy

In general, a *cache* (pronounced "cash") is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device. The process of using a cache is known as *caching* (pronounced "cashing").

The central idea of a memory hierarchy is that for each k, the faster and smaller storage device at level k serves as a cache for the larger and slower storage device at level $k+1$. In other words, each level in the hierarchy caches data objects from the next lower level. For example, the local disk serves as a cache for files (such as Web pages) retrieved from remote disks over the network, the main memory serves as a cache for data on the local disks, and so on, until we get to the smallest cache of all, the set of CPU registers.

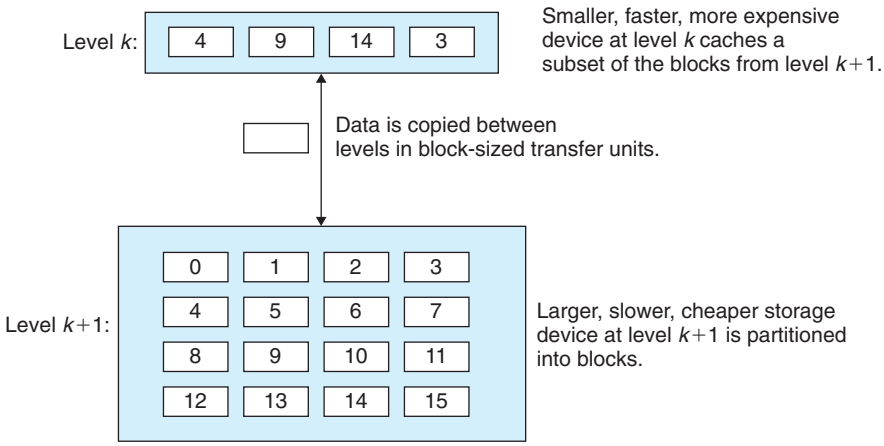


Figure 6.24 **The basic principle of caching in a memory hierarchy.**

Figure 6.24 shows the general concept of caching in a memory hierarchy. The storage at level $k+1$ is partitioned into contiguous chunks of data objects called *blocks*. Each block has a unique address or name that distinguishes it from other blocks. Blocks can be either fixed-sized (the usual case) or variable-sized (e.g., the remote HTML files stored on Web servers). For example, the level $k+1$ storage in Figure 6.24 is partitioned into 16 fixed-sized blocks, numbered 0 to 15.

Similarly, the storage at level k is partitioned into a smaller set of blocks that are the same size as the blocks at level $k+1$. At any point in time, the cache at level k contains copies of a subset of the blocks from level $k+1$. For example, in Figure 6.24, the cache at level k has room for four blocks and currently contains copies of blocks 4, 9, 14, and 3.

Data is always copied back and forth between level k and level $k+1$ in block-sized *transfer units*. It is important to realize that while the block size is fixed between any particular pair of adjacent levels in the hierarchy, other pairs of levels can have different block sizes. For example, in Figure 6.23, transfers between L1 and L0 typically use one-word blocks. Transfers between L2 and L1 (and L3 and L2, and L4 and L3) typically use blocks of 8 to 16 words. And transfers between L5 and L4 use blocks with hundreds or thousands of bytes. In general, devices lower in the hierarchy (further from the CPU) have longer access times, and thus tend to use larger block sizes in order to amortize these longer access times.

Cache Hits

When a program needs a particular data object d from level $k+1$, it first looks for d in one of the blocks currently stored at level k. If d happens to be cached at level k, then we have what is called a *cache hit*. The program reads d directly from level k, which by the nature of the memory hierarchy is faster than reading d from level $k+1$. For example, a program with good temporal locality might read a data object from block 14, resulting in a cache hit from level k.

Cache Misses

If, on the other hand, the data object d is not cached at level k, then we have what is called a *cache miss*. When there is a miss, the cache at level k fetches the block containing d from the cache at level $k+1$, possibly overwriting an existing block if the level k cache is already full.

This process of overwriting an existing block is known as *replacing* or *evicting* the block. The block that is evicted is sometimes referred to as a *victim block*. The decision about which block to replace is governed by the cache's *replacement policy*. For example, a cache with a *random replacement policy* would choose a random victim block. A cache with a least-recently used (LRU) replacement policy would choose the block that was last accessed the furthest in the past.

After the cache at level k has fetched the block from level $k+1$, the program can read d from level k as before. For example, in Figure 6.24, reading a data object from block 12 in the level k cache would result in a cache miss because block 12 is not currently stored in the level k cache. Once it has been copied from level $k+1$ to level k, block 12 will remain there in expectation of later accesses.

Kinds of Cache Misses

It is sometimes helpful to distinguish between different kinds of cache misses. If the cache at level k is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a *cold cache*, and misses of this kind are called *compulsory misses* or *cold misses*. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been *warmed up* by repeated memory accesses.

Whenever there is a miss, the cache at level k must implement some *placement policy* that determines where to place the block it has retrieved from level $k+1$. The most flexible placement policy is to allow any block from level $k+1$ to be stored in any block at level k. For caches high in the memory hierarchy (close to the CPU) that are implemented in hardware and where speed is at a premium, this policy is usually too expensive to implement because randomly placed blocks are expensive to locate.

Thus, hardware caches typically implement a more restricted placement policy that restricts a particular block at level $k+1$ to a small subset (sometimes a singleton) of the blocks at level k. For example, in Figure 6.24, we might decide that a block i at level $k+1$ must be placed in block (i mod 4) at level k. For example, blocks 0, 4, 8, and 12 at level $k+1$ would map to block 0 at level k; blocks 1, 5, 9, and 13 would map to block 1; and so on. Notice that our example cache in Figure 6.24 uses this policy.

Restrictive placement policies of this kind lead to a type of miss known as a *conflict miss*, in which the cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing. For example, in Figure 6.24, if the program requests block 0, then block 8, then block 0, then block 8, and so on, each of the references to these two blocks would miss in the cache at level k, even though this cache can hold a total of four blocks.

Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the *working set* of the phase. When the size of the working set exceeds the size of the cache, the cache will experience what are known as *capacity misses*. In other words, the cache is just too small to handle this particular working set.

Cache Management

As we have noted, the essence of the memory hierarchy is that the storage device at each level is a cache for the next lower level. At each level, some form of logic must *manage* the cache. By this we mean that something has to partition the cache storage into blocks, transfer blocks between different levels, decide when there are hits and misses, and then deal with them. The logic that manages the cache can be hardware, software, or a combination of the two.

For example, the compiler manages the register file, the highest level of the cache hierarchy. It decides when to issue loads when there are misses, and determines which register to store the data in. The caches at levels L1, L2, and L3 are managed entirely by hardware logic built into the caches. In a system with virtual memory, the DRAM main memory serves as a cache for data blocks stored on disk, and is managed by a combination of operating system software and address translation hardware on the CPU. For a machine with a distributed file system such as AFS, the local disk serves as a cache that is managed by the AFS client process running on the local machine. In most cases, caches operate automatically and do not require any specific or explicit actions from the program.

6.3.2 Summary of Memory Hierarchy Concepts

To summarize, memory hierarchies based on caching work because slower storage is cheaper than faster storage and because programs tend to exhibit locality:

- *Exploiting temporal locality.* Because of temporal locality, the same data objects are likely to be reused multiple times. Once a data object has been copied into the cache on the first miss, we can expect a number of subsequent hits on that object. Since the cache is faster than the storage at the next lower level, these subsequent hits can be served much faster than the original miss.
- *Exploiting spatial locality.* Blocks usually contain multiple data objects. Because of spatial locality, we can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

Caches are used everywhere in modern systems. As you can see from Figure 6.25, caches are used in CPU chips, operating systems, distributed file systems, and on the World Wide Web. They are built from and managed by various combinations of hardware and software. Note that there are a number of terms and acronyms in Figure 6.25 that we haven't covered yet. We include them here to demonstrate how common caches are.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte block	On-chip L1 cache	1	Hardware
L2 cache	64-byte block	On/off-chip L2 cache	10	Hardware
L3 cache	64-byte block	On/off-chip L3 cache	30	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

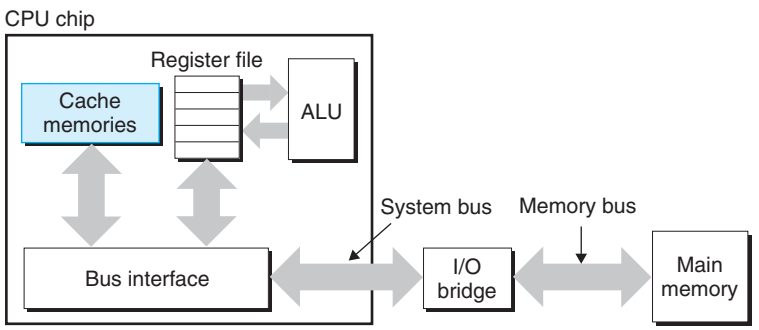
Figure 6.25 **The ubiquity of caching in modern computer systems.** Acronyms: TLB: translation lookaside buffer, MMU: memory management unit, OS: operating system, AFS: Andrew File System, NFS: Network File System.

6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main DRAM memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert a small SRAM *cache memory*, called an *L1 cache* (Level 1 cache) between the CPU register file and main memory, as shown in Figure 6.26. The L1 cache can be accessed nearly as fast as the registers, typically in 2 to 4 clock cycles.

As the performance gap between the CPU and main memory continued to increase, system designers responded by inserting an additional larger cache, called an *L2 cache*, between the L1 cache and main memory, that can be accessed in about 10 clock cycles. Some modern systems include an additional even larger cache, called an *L3 cache*, which sits between the L2 cache and main memory

Figure 6.26
Typical bus structure for cache memories.



in the memory hierarchy and can be accessed in 30 or 40 cycles. While there is considerable variety in the arrangements, the general principles are the same. For our discussion in the next section, we will assume a simple memory hierarchy with a single L1 cache between the CPU and main memory.

6.4.1 Generic Cache Memory Organization

Consider a computer system where each memory address has m bits that form $M = 2^m$ unique addresses. As illustrated in Figure 6.27(a), a cache for such a machine is organized as an array of $S = 2^s$ *cache sets*. Each set consists of E *cache lines*. Each line consists of a data *block* of $B = 2^b$ bytes, a *valid bit* that indicates whether or not the line contains meaningful information, and $t = m - (b + s)$ *tag bits* (a subset of the bits from the current block's memory address) that uniquely identify the block stored in the cache line.

In general, a cache's organization can be characterized by the tuple (S, E, B, m). The size (or capacity) of a cache, C, is stated in terms of the aggregate size of all the blocks. The tag bits and valid bit are not included. Thus, $C = S \times E \times B$.

When the CPU is instructed by a load instruction to read a word from address A of main memory, it sends the address A to the cache. If the cache is holding a copy of the word at address A, it sends the word immediately back to the CPU.

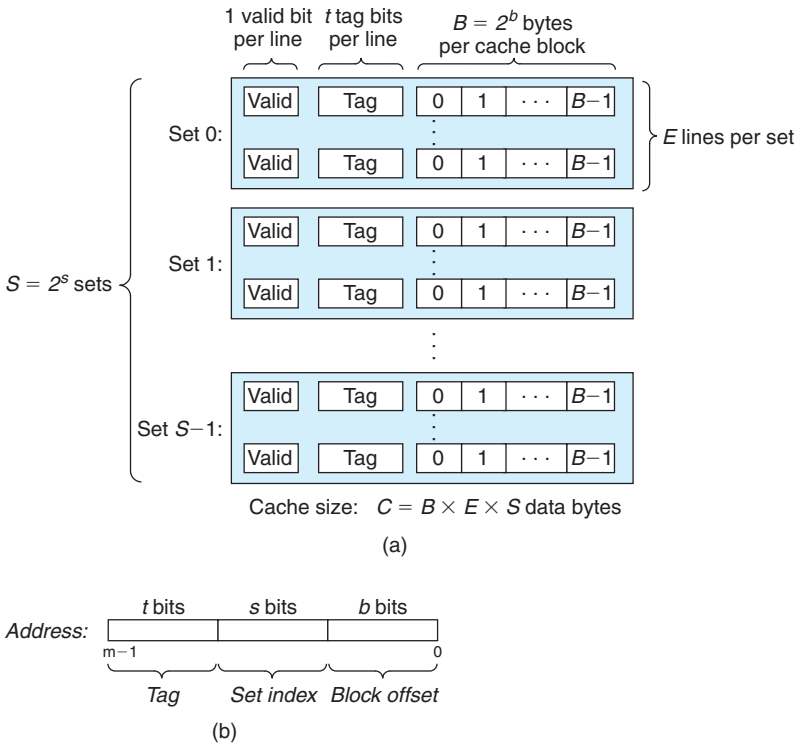


Figure 6.27 **General organization of cache** (S, E, B, m). (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.

Fundamental parameters	
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

Derived quantities	
Parameter	Description
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

Figure 6.28 **Summary of cache parameters.**

So how does the cache know whether it contains a copy of the word at address A? The cache is organized so that it can find the requested word by simply inspecting the bits of the address, similar to a hash table with an extremely simple hash function. Here is how it works:

The parameters S and B induce a partitioning of the m address bits into the three fields shown in Figure 6.27(b). The s *set index bits* in A form an index into the array of S sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the set index bits tell us which set the word must be stored in. Once we know which set the word must be contained in, the t tag bits in A tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A. Once we have located the line identified by the tag in the set identified by the set index, then the b *block offset bits* give us the offset of the word in the B-byte data block.

As you may have noticed, descriptions of caches use a lot of symbols. Figure 6.28 summarizes these symbols for your reference.

Practice Problem 6.10

The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1				
2.	32	1024	8	4				
3.	32	1024	32	32				

6.4.2 Direct-Mapped Caches

Caches are grouped into different classes based on E, the number of cache lines per set. A cache with exactly one line per set ($E = 1$) is known as a *direct-mapped* cache (see Figure 6.29). Direct-mapped caches are the simplest both to implement and to understand, so we will use them to illustrate some general concepts about how caches work.

Suppose we have a system with a CPU, a register file, an L1 cache, and a main memory. When the CPU executes an instruction that reads a memory word w, it requests the word from the L1 cache. If the L1 cache has a cached copy of w, then we have an L1 cache hit, and the cache quickly extracts w and returns it to the CPU. Otherwise, we have a cache miss, and the CPU must wait while the L1 cache requests a copy of the block containing w from the main memory. When the requested block finally arrives from memory, the L1 cache stores the block in one of its cache lines, extracts word w from the stored block, and returns it to the CPU. The process that a cache goes through of determining whether a request is a hit or a miss and then extracting the requested word consists of three steps: (1) *set selection*, (2) *line matching*, and (3) *word extraction*.

Set Selection in Direct-Mapped Caches

In this step, the cache extracts the s set index bits from the middle of the address for w. These bits are interpreted as an unsigned integer that corresponds to a set number. In other words, if we think of the cache as a one-dimensional array of sets, then the set index bits form an index into this array. Figure 6.30 shows how set selection works for a direct-mapped cache. In this example, the set index bits 00001_2 are interpreted as an integer index that selects set 1.

Line Matching in Direct-Mapped Caches

Now that we have selected some set i in the previous step, the next step is to determine if a copy of the word w is stored in one of the cache lines contained in

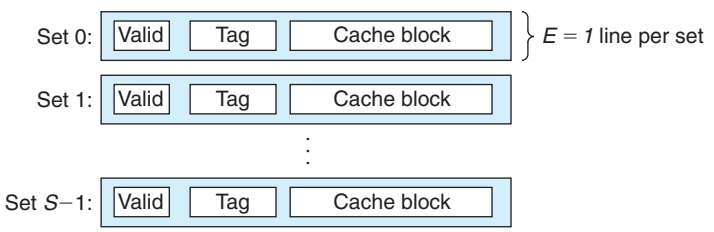


Figure 6.29
Direct-mapped cache (*E* = 1). There is exactly one line per set.

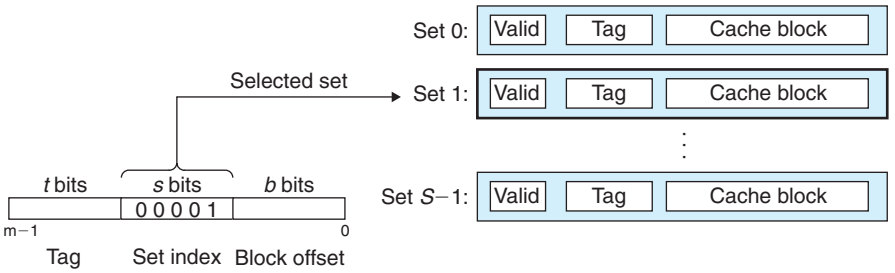


Figure 6.30 **Set selection in a direct-mapped cache.**

set i. In a direct-mapped cache, this is easy and fast because there is exactly one line per set. A copy of w is contained in the line if and only if the valid bit is set and the tag in the cache line matches the tag in the address of w.

Figure 6.31 shows how line matching works in a direct-mapped cache. In this example, there is exactly one cache line in the selected set. The valid bit for this line is set, so we know that the bits in the tag and block are meaningful. Since the tag bits in the cache line match the tag bits in the address, we know that a copy of the word we want is indeed stored in the line. In other words, we have a cache hit. On the other hand, if either the valid bit were not set or the tags did not match, then we would have had a cache miss.

Word Selection in Direct-Mapped Caches

Once we have a hit, we know that w is somewhere in the block. This last step determines where the desired word starts in the block. As shown in Figure 6.31, the block offset bits provide us with the offset of the first byte in the desired word. Similar to our view of a cache as an array of lines, we can think of a block as an array of bytes, and the byte offset as an index into that array. In the example, the block offset bits of 100_2 indicate that the copy of w starts at byte 4 in the block. (We are assuming that words are 4 bytes long.)

Line Replacement on Misses in Direct-Mapped Caches

If the cache misses, then it needs to retrieve the requested block from the next level in the memory hierarchy and store the new block in one of the cache lines of

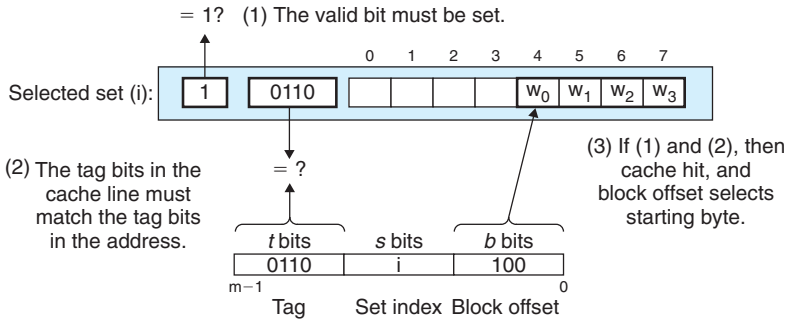


Figure 6.31
Line matching and word selection in a direct-mapped cache. Within the cache block, w_0 denotes the low-order byte of the word w, w_1 the next byte, and so on.

Address (decimal)	Address bits: Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.32 **4-bit address space for example direct-mapped cache.**

the set indicated by the set index bits. In general, if the set is full of valid cache lines, then one of the existing lines must be evicted. For a direct-mapped cache, where each set contains exactly one line, the replacement policy is trivial: the current line is replaced by the newly fetched line.

Putting It Together: A Direct-Mapped Cache in Action

The mechanisms that a cache uses to select sets and identify lines are extremely simple. They have to be, because the hardware must perform them in a few nanoseconds. However, manipulating bits in this way can be confusing to us humans. A concrete example will help clarify the process. Suppose we have a direct-mapped cache described by

$$(S, E, B, m) = (4, 1, 2, 4)$$

In other words, the cache has four sets, one line per set, 2 bytes per block, and 4-bit addresses. We will also assume that each word is a single byte. Of course, these assumptions are totally unrealistic, but they will help us keep the example simple.

When you are first learning about caches, it can be very instructive to enumerate the entire address space and partition the bits, as we've done in Figure 6.32 for our 4-bit example. There are some interesting things to notice about this enumerated space:

- The concatenation of the tag and index bits uniquely identifies each block in memory. For example, block 0 consists of addresses 0 and 1, block 1 consists of addresses 2 and 3, block 2 consists of addresses 4 and 5, and so on.
- Since there are eight memory blocks but only four cache sets, multiple blocks map to the same cache set (i.e., they have the same set index). For example, blocks 0 and 4 both map to set 0, blocks 1 and 5 both map to set 1, and so on.
- Blocks that map to the same cache set are uniquely identified by the tag. For example, block 0 has a tag bit of 0 while block 4 has a tag bit of 1, block 1 has a tag bit of 0 while block 5 has a tag bit of 1, and so on.

Let us simulate the cache in action as the CPU performs a sequence of reads. Remember that for this example, we are assuming that the CPU reads 1-byte words. While this kind of manual simulation is tedious and you may be tempted to skip it, in our experience students do not really understand how caches work until they work their way through a few of them.

Initially, the cache is empty (i.e., each valid bit is zero):

Set	Valid	Tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

Each row in the table represents a cache line. The first column indicates the set that the line belongs to, but keep in mind that this is provided for convenience and is not really part of the cache. The next three columns represent the actual bits in each cache line. Now, let us see what happens when the CPU performs a sequence of reads:

1. **Read word at address 0.** Since the valid bit for set 0 is zero, this is a cache miss. The cache fetches block 0 from memory (or a lower-level cache) and stores the block in set 0. Then the cache returns m[0] (the contents of memory location 0) from block[0] of the newly fetched cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

2. **Read word at address 1.** This is a cache hit. The cache immediately returns m[1] from block[1] of the cache line. The state of the cache does not change.
3. **Read word at address 13.** Since the cache line in set 2 is not valid, this is a cache miss. The cache loads block 6 into set 2 and returns m[13] from block[1] of the new cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

4. **Read word at address 8.** This is a miss. The cache line in set 0 is indeed valid, but the tags do not match. The cache loads block 4 into set 0 (replacing the line that was there from the read of address 0) and returns m[8] from block[0] of the new cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. **Read word at address 0.** This is another miss, due to the unfortunate fact that we just replaced block 0 during the previous reference to address 8. This kind of miss, where we have plenty of room in the cache but keep alternating references to blocks that map to the same set, is an example of a conflict miss.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Conflict Misses in Direct-Mapped Caches

Conflict misses are common in real programs and can cause baffling performance problems. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of 2. For example, consider a function that computes the dot product of two vectors:

```
1  float dotprod(float x[8], float y[8])
2  {
3      float sum = 0.0;
4      int i;
5
6      for (i = 0; i < 8; i++)
7          sum += x[i] * y[i];
8      return sum;
9  }
```

This function has good spatial locality with respect to `x` and `y`, and so we might expect it to enjoy a good number of cache hits. Unfortunately, this is not always true.

Suppose that floats are 4 bytes, that `x` is loaded into the 32 bytes of contiguous memory starting at address 0, and that `y` starts immediately after `x` at address 32. For simplicity, suppose that a block is 16 bytes (big enough to hold four floats) and that the cache consists of two sets, for a total cache size of 32 bytes. We will assume that the variable `sum` is actually stored in a CPU register and thus does not require a memory reference. Given these assumptions, each `x[i]` and `y[i]` will map to the identical cache set:

Element	Address	Set index	Element	Address	Set index
`x[0]`	0	0	`y[0]`	32	0
`x[1]`	4	0	`y[1]`	36	0
`x[2]`	8	0	`y[2]`	40	0
`x[3]`	12	0	`y[3]`	44	0
`x[4]`	16	1	`y[4]`	48	1
`x[5]`	20	1	`y[5]`	52	1
`x[6]`	24	1	`y[6]`	56	1
`x[7]`	28	1	`y[7]`	60	1

At run time, the first iteration of the loop references `x[0]`, a miss that causes the block containing `x[0]`–`x[3]` to be loaded into set 0. The next reference is to `y[0]`, another miss that causes the block containing `y[0]`–`y[3]` to be copied into set 0, overwriting the values of `x` that were copied in by the previous reference. During the next iteration, the reference to `x[1]` misses, which causes the `x[0]`–`x[3]` block to be loaded back into set 0, overwriting the `y[0]`–`y[3]` block. So now we have a conflict miss, and in fact each subsequent reference to `x` and `y` will result in a conflict miss as we *thrash* back and forth between blocks of `x` and `y`. The term *thrashing* describes any situation where a cache is repeatedly loading and evicting the same sets of cache blocks.

The bottom line is that even though the program has good spatial locality and we have room in the cache to hold the blocks for both `x[i]` and `y[i]`, each reference results in a conflict miss because the blocks map to the same cache set. It is not unusual for this kind of thrashing to result in a slowdown by a factor of 2 or 3. Also, be aware that even though our example is extremely simple, the problem is real for larger and more realistic direct-mapped caches.

Luckily, thrashing is easy for programmers to fix once they recognize what is going on. One easy solution is to put B bytes of padding at the end of each array. For example, instead of defining `x` to be `float x[8]`, we define it to be `float x[12]`. Assuming `y` starts immediately after `x` in memory, we have the following mapping of array elements to sets:

Element	Address	Set index	Element	Address	Set index
x[0]	0	0	y[0]	48	1
x[1]	4	0	y[1]	52	1
x[2]	8	0	y[2]	56	1
x[3]	12	0	y[3]	60	1
x[4]	16	1	y[4]	64	0
x[5]	20	1	y[5]	68	0
x[6]	24	1	y[6]	72	0
x[7]	28	1	y[7]	76	0

With the padding at the end of `x`, `x[i]` and `y[i]` now map to different sets, which eliminates the thrashing conflict misses.

Practice Problem 6.11

In the previous `dotprod` example, what fraction of the total references to `x` and `y` will be hits once we have padded array `x`?

Practice Problem 6.12

In general, if the high-order s bits of an address are used as the set index, contiguous chunks of memory blocks are mapped to the same cache set.

A. How many blocks are in each of these contiguous array chunks?

B. Consider the following code that runs on a system with a cache of the form $(S, E, B, m) = (512, 1, 32, 32)$:

```
int array[4096];

for (i = 0; i < 4096; i++)
    sum += array[i];
```

What is the maximum number of array blocks that are stored in the cache at any point in time?

Aside Why index with the middle bits?

You may be wondering why caches use the middle bits for the set index instead of the high-order bits. There is a good reason why the middle bits are better. Figure 6.33 shows why. If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set. For example, in the figure, the first four blocks map to the first cache set, the second four blocks map to the second set, and so on. If a program has good spatial locality and scans the elements of an array sequentially, then the cache can only hold a block-sized chunk of the array at any point in time. This is an inefficient use of the cache. Contrast this with middle-bit indexing, where adjacent blocks always map to different cache lines. In this case, the cache can hold an entire C-sized chunk of the array, where C is the cache size.

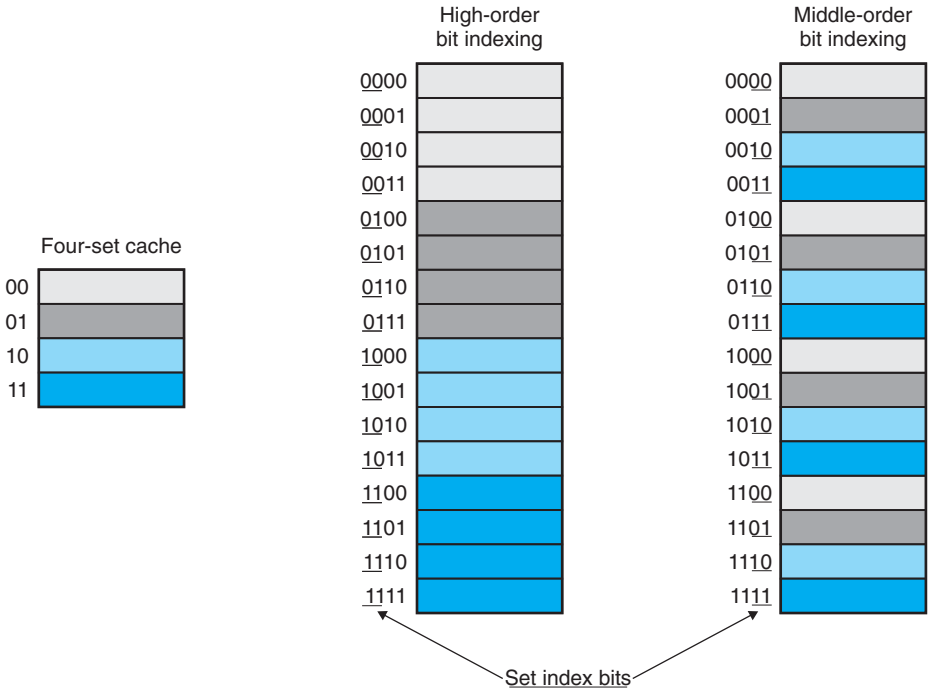


Figure 6.33 **Why caches index with the middle bits.**

6.4.3 Set Associative Caches

The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$). A *set associative cache* relaxes this constraint so each set holds more than one cache line. A cache with $1 < E < C/B$ is often called an E-way set associative cache. We will discuss the special case, where $E = C/B$, in the next section. Figure 6.34 shows the organization of a two-way set associative cache.

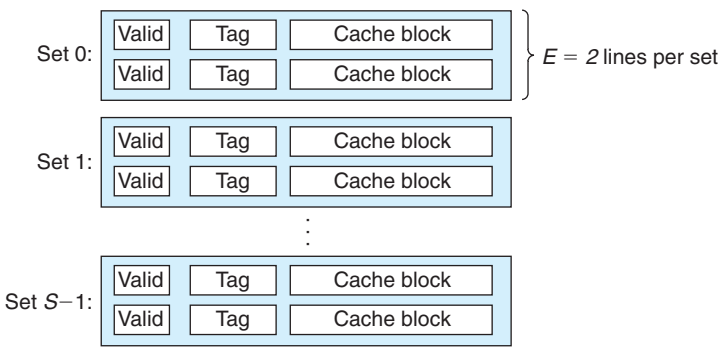


Figure 6.34
Set associative cache $(1 < E < C/B)$**.** In a set associative cache, each set contains more than one line. This particular example shows a two-way set associative cache.

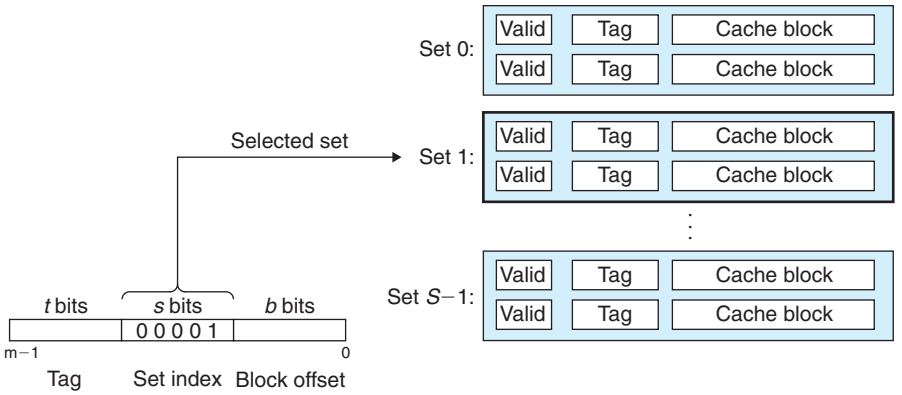


Figure 6.35 **Set selection in a set associative cache.**

Set Selection in Set Associative Caches

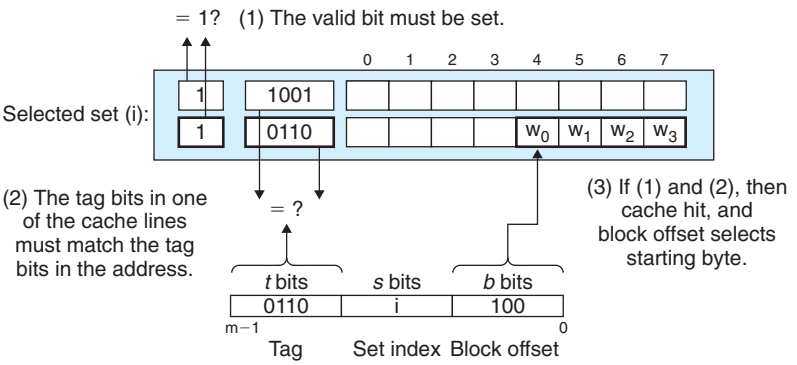
Set selection is identical to a direct-mapped cache, with the set index bits identifying the set. Figure 6.35 summarizes this principle.

Line Matching and Word Selection in Set Associative Caches

Line matching is more involved in a set associative cache than in a direct-mapped cache because it must check the tags and valid bits of multiple lines in order to determine if the requested word is in the set. A conventional memory is an array of values that takes an address as input and returns the value stored at that address. An *associative memory*, on the other hand, is an array of (key, value) pairs that takes as input the key and returns a value from one of the (key, value) pairs that matches the input key. Thus, we can think of each set in a set associative cache as a small associative memory where the keys are the concatenation of the tag and valid bits, and the values are the contents of a block.

Figure 6.36 shows the basic idea of line matching in an associative cache. An important idea here is that any line in the set can contain any of the memory blocks

Figure 6.36
Line matching and word selection in a set associative cache.



that map to that set. So the cache must search each line in the set, searching for a valid line whose tag matches the tag in the address. If the cache finds such a line, then we have a hit and the block offset selects a word from the block, as before.

Line Replacement on Misses in Set Associative Caches

If the word requested by the CPU is not stored in any of the lines in the set, then we have a cache miss, and the cache must fetch the block that contains the word from memory. However, once the cache has retrieved the block, which line should it replace? Of course, if there is an empty line, then it would be a good candidate. But if there are no empty lines in the set, then we must choose one of the nonempty lines and hope that the CPU does not reference the replaced line anytime soon.

It is very difficult for programmers to exploit knowledge of the cache replacement policy in their codes, so we will not go into much detail about it here. The simplest replacement policy is to choose the line to replace at random. Other more sophisticated policies draw on the principle of locality to try to minimize the probability that the replaced line will be referenced in the near future. For example, a *least-frequently-used* (LFU) policy will replace the line that has been referenced the fewest times over some past time window. A *least-recently-used* (LRU) policy will replace the line that was last accessed the furthest in the past. All of these policies require additional time and hardware. But as we move further down the memory hierarchy, away from the CPU, the cost of a miss becomes more expensive and it becomes more worthwhile to minimize misses with good replacement policies.

6.4.4 Fully Associative Caches

A *fully associative cache* consists of a single set (i.e., $E = C/B$) that contains all of the cache lines. Figure 6.37 shows the basic organization.

Set Selection in Fully Associative Caches

Set selection in a fully associative cache is trivial because there is only one set, summarized in Figure 6.38. Notice that there are no set index bits in the address, which is partitioned into only a tag and a block offset.

Line Matching and Word Selection in Fully Associative Caches

Line matching and word selection in a fully associative cache work the same as with a set associative cache, as we show in Figure 6.39. The difference is mainly a question of scale. Because the cache circuitry must search for many matching

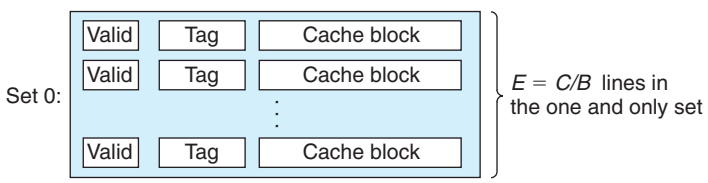


Figure 6.37
Fully associative cache ($E = C/B$). In a fully associative cache, a single set contains all of the lines.

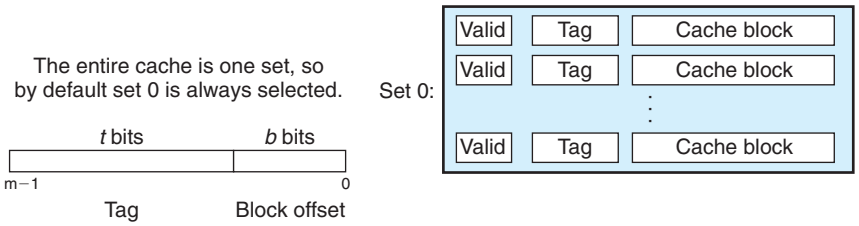


Figure 6.38 **Set selection in a fully associative cache.** Notice that there are no set index bits.

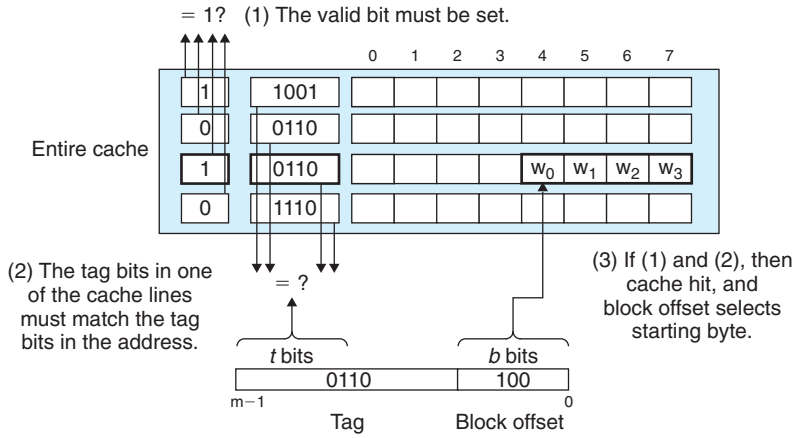


Figure 6.39 **Line matching and word selection in a fully associative cache.**

tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast. As a result, fully associative caches are only appropriate for small caches, such as the translation lookaside buffers (TLBs) in virtual memory systems that cache page table entries (Section 9.6.2).

Practice Problem 6.13

The problems that follow will help reinforce your understanding of how caches work. Assume the following:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and eight sets ($S = 8$).

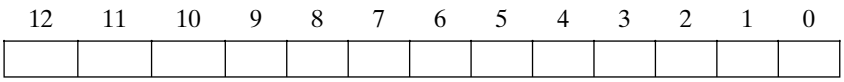
The contents of the cache are as follows, with all numbers given in hexadecimal notation.

2-way set associative cache

	Line 0						Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	—	—	—	—
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	—	—	—	—	0B	0	—	—	—	—
3	06	0	—	—	—	—	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	—	—	—	—
6	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

The following figure shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

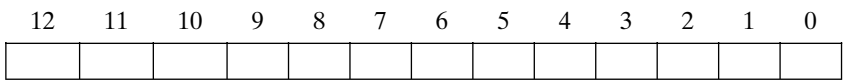
CO The cache block offset
CI The cache set index
CT The cache tag



Practice Problem 6.14

Suppose a program running on the machine in Problem 6.13 references the 1-byte word at address `0x0E34`. Indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned."

A. Address format (one bit per box):



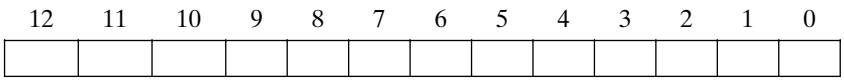
B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Practice Problem 6.15

Repeat Problem 6.14 for memory address `0x0DD5`.

A. Address format (one bit per box):



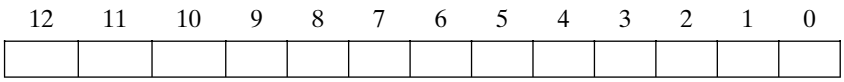
B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Practice Problem 6.16

Repeat Problem 6.14 for memory address `0x1FE4`.

A. Address format (one bit per box):



B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Practice Problem 6.17

For the cache in Problem 6.13, list all of the hex memory addresses that will hit in set 3.

6.4.5 Issues with Writes

As we have seen, the operation of a cache with respect to reads is straightforward. First, look for a copy of the desired word w in the cache. If there is a hit, return

w immediately. If there is a miss, fetch the block that contains w from the next lower level of the memory hierarchy, store the block in some cache line (possibly evicting a valid line), and then return w.

The situation for writes is a little more complicated. Suppose we write a word w that is already cached (a *write hit*). After the cache updates its copy of w, what does it do about updating the copy of w in the next lower level of the hierarchy? The simplest approach, known as *write-through*, is to immediately write w's cache block to the next lower level. While simple, write-through has the disadvantage of causing bus traffic with every write. Another approach, known as *write-back*, defers the update as long as possible by writing the updated block to the next lower level only when it is evicted from the cache by the replacement algorithm. Because of locality, write-back can significantly reduce the amount of bus traffic, but it has the disadvantage of additional complexity. The cache must maintain an additional *dirty bit* for each cache line that indicates whether or not the cache block has been modified.

Another issue is how to deal with write misses. One approach, known as *write-allocate*, loads the corresponding block from the next lower level into the cache and then updates the cache block. Write-allocate tries to exploit spatial locality of writes, but it has the disadvantage that every miss results in a block transfer from the next lower level to cache. The alternative, known as *no-write-allocate*, bypasses the cache and writes the word directly to the next lower level. Write-through caches are typically no-write-allocate. Write-back caches are typically write-allocate.

Optimizing caches for writes is a subtle and difficult issue, and we are only scratching the surface here. The details vary from system to system and are often proprietary and poorly documented. To the programmer trying to write reasonably cache-friendly programs, we suggest adopting a mental model that assumes write-back write-allocate caches. There are several reasons for this suggestion.

As a rule, caches at lower levels of the memory hierarchy are more likely to use write-back instead of write-through because of the larger transfer times. For example, virtual memory systems (which use main memory as a cache for the blocks stored on disk) use write-back exclusively. But as logic densities increase, the increased complexity of write-back is becoming less of an impediment and we are seeing write-back caches at all levels of modern systems. So this assumption matches current trends. Another reason for assuming a write-back write-allocate approach is that it is symmetric to the way reads are handled, in that write-back write-allocate tries to exploit locality. Thus, we can develop our programs at a high level to exhibit good spatial and temporal locality rather than trying to optimize for a particular memory system.

6.4.6 Anatomy of a Real Cache Hierarchy

So far, we have assumed that caches hold only program data. But in fact, caches can hold instructions as well as data. A cache that holds instructions only is called an *i-cache*. A cache that holds program data only is called a *d-cache*. A cache that holds both instructions and data is known as a *unified cache*. Modern processors

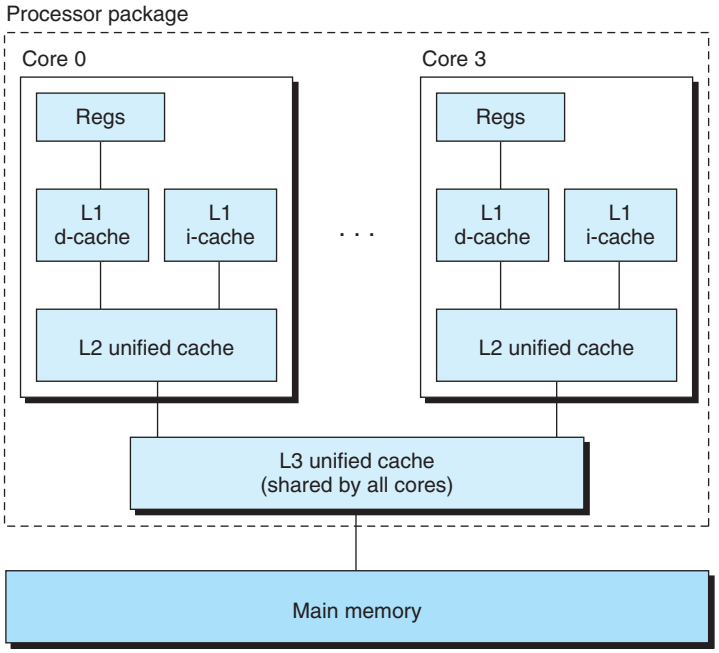


Figure 6.40
Intel Core i7 cache hierarchy.

include separate i-caches and d-caches. There are a number of reasons for this. With two separate caches, the processor can read an instruction word and a data word at the same time. I-caches are typically read-only, and thus simpler. The two caches are often optimized to different access patterns and can have different block sizes, associativities, and capacities. Also, having separate caches ensures that data accesses do not create conflict misses with instruction accesses, and vice versa, at the cost of a potential increase in capacity misses.

Figure 6.40 shows the cache hierarchy for the Intel Core i7 processor. Each CPU chip has four cores. Each core has its own private L1 i-cache, L1 d-cache, and L2 unified cache. All of the cores share an on-chip L3 unified cache. An interesting feature of this hierarchy is that all of the SRAM cache memories are contained in the CPU chip.

Figure 6.41 summarizes the basic characteristics of the Core i7 caches.

Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	11	256 KB	8	64 B	512
L3 unified cache	30–40	8 MB	16	64 B	8192

Figure 6.41 **Characteristics of the Intel Core i7 cache hierarchy.**

6.4.7 Performance Impact of Cache Parameters

Cache performance is evaluated with a number of metrics:

- *Miss rate.* The fraction of memory references during the execution of a program, or a part of a program, that miss. It is computed as *#misses/#references*.
- *Hit rate.* The fraction of memory references that hit. It is computed as 1 – *miss rate*.
- *Hit time.* The time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is on the order of several clock cycles for L1 caches.
- *Miss penalty.* Any additional time required because of a miss. The penalty for L1 misses served from L2 is on the order of 10 cycles; from L3, 40 cycles; and from main memory, 100 cycles.

Optimizing the cost and performance trade-offs of cache memories is a subtle exercise that requires extensive simulation on realistic benchmark codes and thus is beyond our scope. However, it is possible to identify some of the qualitative trade-offs.

Impact of Cache Size

On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make large memories run faster. As a result, larger caches tend to increase the hit time. This is especially important for on-chip L1 caches that must have a short hit time.

Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems usually compromise with cache blocks that contain 32 to 64 bytes.

Impact of Associativity

The issue here is the impact of the choice of the parameter E, the number of cache lines per set. The advantage of higher associativity (i.e., larger values of E) is that it decreases the vulnerability of the cache to thrashing due to conflict misses. However, higher associativity comes at a significant cost. Higher associativity is expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. Higher associativity can increase hit time, because of the increased complexity, and it can also increase the miss penalty because of the increased complexity of choosing a victim line.

The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for smaller associativity for L1 caches (where the miss penalty is only a few cycles) and a higher degree of associativity for the lower levels, where the miss penalty is higher. For example, in Intel Core i7 systems, the L1 and L2 caches are 8-way associative, and the L3 cache is 16-way.

Impact of Write Strategy

Write-through caches are simpler to implement and can use a *write buffer* that works independently of the cache to update memory. Furthermore, read misses are less expensive because they do not trigger a memory write. On the other hand, write-back caches result in fewer transfers, which allows more bandwidth to memory for I/O devices that perform DMA. Further, reducing the number of transfers becomes increasingly important as we move down the hierarchy and the transfer times increase. In general, caches further down the hierarchy are more likely to use write-back than write-through.

Aside Cache lines, sets, and blocks: What's the difference?

It is easy to confuse the distinction between cache lines, sets, and blocks. Let's review these ideas and make sure they are clear:

- A *block* is a fixed-sized packet of information that moves back and forth between a cache and main memory (or a lower-level cache).
- A *line* is a container in a cache that stores a block, as well as other information such as the valid bit and the tag bits.
- A *set* is a collection of one or more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.

In direct-mapped caches, sets and lines are indeed equivalent. However, in associative caches, sets and lines are very different things and the terms cannot be used interchangeably.

Since a line always stores a single block, the terms "line" and "block" are often used interchangeably. For example, systems professionals usually refer to the "line size" of a cache, when what they really mean is the block size. This usage is very common, and shouldn't cause any confusion, so long as you understand the distinction between blocks and lines.

6.5 Writing Cache-friendly Code

In Section 6.2, we introduced the idea of locality and talked in qualitative terms about what constitutes good locality. Now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to

write code that is *cache friendly*, in the sense that it has good locality. Here is the basic approach we use to try to ensure that our code is cache friendly.

1. *Make the common case go fast.* Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core functions and ignore the rest.
2. *Minimize the number of cache misses in each inner loop.* All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

To see how this works in practice, consider the `sumvec` function from Section 6.2:

```
1   int sumvec(int v[N])
2   {
3       int i, sum = 0;
4
5       for (i = 0; i < N; i++)
6           sum += v[i];
7       return sum;
8   }
```

Is this function cache friendly? First, notice that there is good temporal locality in the loop body with respect to the local variables `i` and `sum`. In fact, because these are local variables, any reasonable optimizing compiler will cache them in the register file, the highest level of the memory hierarchy. Now consider the stride-1 references to vector `v`. In general, if a cache has a block size of B bytes, then a stride-k reference pattern (where k is expressed in words) results in an average of min (1, $(wordsize \times k)/B$) misses per loop iteration. This is minimized for $k = 1$, so the stride-1 references to `v` are indeed cache friendly. For example, suppose that `v` is block aligned, words are 4 bytes, cache blocks are 4 words, and the cache is initially empty (a cold cache). Then, regardless of the cache organization, the references to `v` will result in the following pattern of hits and misses:

`v[i]`	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
Access order, [h]it or [m]iss	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

In this example, the reference to `v[0]` misses and the corresponding block, which contains `v[0]`–`v[3]`, is loaded into the cache from memory. Thus, the next three references are all hits. The reference to `v[4]` causes another miss as a new block is loaded into the cache, the next three references are hits, and so on. In general, three out of four references will hit, which is the best we can do in this case with a cold cache.

To summarize, our simple `sumvec` example illustrates two important points about writing cache-friendly code:

- Repeated references to local variables are good because the compiler can cache them in the register file (temporal locality).

- Stride-1 reference patterns are good because caches at all levels of the memory hierarchy store data as contiguous blocks (spatial locality).

Spatial locality is especially important in programs that operate on multi-dimensional arrays. For example, consider the `sumarrayrows` function from Section 6.2, which sums the elements of a two-dimensional array in row-major order:

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9   }
```

Since C stores arrays in row-major order, the inner loop of this function has the same desirable stride-1 access pattern as `sumvec`. For example, suppose we make the same assumptions about the cache as for `sumvec`. Then the references to the array `a` will result in the following pattern of hits and misses:

a[i][j]	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
i = 0	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
i = 1	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
i = 2	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
i = 3	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

But consider what happens if we make the seemingly innocuous change of permuting the loops:

```
1   int sumarraycols(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (j = 0; j < N; j++)
6           for (i = 0; i < M; i++)
7               sum += a[i][j];
8       return sum;
9   }
```

In this case, we are scanning the array column by column instead of row by row. If we are lucky and the entire array fits in the cache, then we will enjoy the same miss rate of 1/4. However, if the array is larger than the cache (the more likely case), then each and every access of `a[i][j]` will miss!

a[i][j]	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
i = 0	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
i = 1	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
i = 2	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
i = 3	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

Higher miss rates can have a significant impact on running time. For example, on our desktop machine, `sumarrayrows` runs twice as fast as `sumarraycols`. To summarize, programmers should be aware of locality in their programs and try to write programs that exploit it.

Practice Problem 6.18

Transposing the rows and columns of a matrix is an important problem in signal processing and scientific computing applications. It is also interesting from a locality point of view because its reference pattern is both row-wise and column-wise. For example, consider the following transpose routine:

```
1   typedef int array[2][2];
2
3   void transpose1(array dst, array src)
4   {
5       int i, j;
6
7       for (i = 0; i < 2; i++) {
8           for (j = 0; j < 2; j++) {
9               dst[j][i] = src[i][j];
10          }
11      }
12  }
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with a block size of 8 bytes.
- The cache has a total size of 16 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

dst array

	Col 0	Col 1
Row 0	m	
Row 1		

src array

	Col 0	Col 1
Row 0	m	
Row 1		

B. Repeat the problem for a cache with 32 data bytes.

Practice Problem 6.19

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1024-byte direct-mapped data cache with 16-byte blocks ($B = 16$). You are given the following definitions:

```
1  struct algae_position {
2      int x;
3      int y;
4  };
5
6  struct algae_position grid[16][16];
7  int total_x = 0, total_y = 0;
8  int i, j;
```

You should also assume the following:

- `sizeof(int) == 4`.
- `grid` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `grid`. Variables `i`, `j`, `total_x`, and `total_y` are stored in registers.

Determine the cache performance for the following code:

```
1       for (i = 0; i < 16; i++) {
2           for (j = 0; j < 16; j++) {
3               total_x += grid[i][j].x;
4           }
5       }
6
7       for (i = 0; i < 16; i++) {
8           for (j = 0; j < 16; j++) {
9               total_y += grid[i][j].y;
10          }
11      }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

Practice Problem 6.20

Given the assumptions of Problem 6.19, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[j][i].x;
4               total_y += grid[j][i].y;
5           }
6       }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

D. What would the miss rate be if the cache were twice as big?

Practice Problem 6.21

Given the assumptions of Problem 6.19, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[i][j].x;
4               total_y += grid[i][j].y;
5           }
6       }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

D. What would the miss rate be if the cache were twice as big?

6.6 Putting It Together: The Impact of Caches on Program Performance

This section wraps up our discussion of the memory hierarchy by studying the impact that caches have on the performance of programs running on real machines.

6.6.1 The Memory Mountain

The rate that a program reads data from the memory system is called the *read throughput*, or sometimes the *read bandwidth*. If a program reads n bytes over a period of s seconds, then the read throughput over that period is n/s, typically expressed in units of megabytes per second (MB/s).

If we were to write a program that issued a sequence of read requests from a tight program loop, then the measured read throughput would give us some insight into the performance of the memory system for that particular sequence of reads. Figure 6.42 shows a pair of functions that measure the read throughput for a particular read sequence.

The `test` function generates the read sequence by scanning the first `elems` elements of an array with a stride of `stride`. The `run` function is a wrapper that calls the `test` function and returns the measured read throughput. The call to the `test` function in line 29 warms the cache. The `fcyc2` function in line 30 calls the `test` function with arguments `elems` and estimates the running time of the `test` function in CPU cycles. Notice that the `size` argument to the `run` function is in units of bytes, while the corresponding `elems` argument to the `test` function is in units of array elements. Also, notice that line 31 computes MB/s as 10^6 bytes/s, as opposed to 2^{20} bytes/s.

The `size` and `stride` arguments to the `run` function allow us to control the degree of temporal and spatial locality in the resulting read sequence. Smaller values of `size` result in a smaller working set size, and thus better temporal locality. Smaller values of `stride` result in better spatial locality. If we call the `run` function repeatedly with different values of `size` and `stride`, then we can recover a fascinating two-dimensional function of read throughput versus temporal and spatial locality. This function is called a *memory mountain*.

Every computer has a unique memory mountain that characterizes the capabilities of its memory system. For example, Figure 6.43 shows the memory mountain for an Intel Core i7 system. In this example, the `size` varies from 2 KB to 64 MB, and the `stride` varies from 1 to 64 elements, where each element is an 8-byte `double`.

The geography of the Core i7 mountain reveals a rich structure. Perpendicular to the `size` axis are four *ridges* that correspond to the regions of temporal locality where the working set fits entirely in the L1 cache, the L2 cache, the L3 cache, and main memory, respectively. Notice that there is an order of magnitude difference between the highest peak of the L1 ridge, where the CPU reads at a rate of over 6 GB/s, and the lowest point of the main memory ridge, where the CPU reads at a rate of 600 MB/s.

There is a feature of the L1 ridge that should be pointed out. For very large strides, notice how the read throughput drops as the working set size approaches 2 KB (falling off the back side of the ridge). Since the L1 cache holds the entire working set, this feature does not reflect the true L1 cache performance. It is an artifact of overheads of calling the `test` function and setting up to execute the loop. For large strides in small working set sizes, these overheads are not amortized, as they are with the larger sizes.

code/mem/mountain/mountain.c

```
1   double data[MAXELEMS];      /* The global array we'll be traversing */
2
3   /*
4    * test - Iterate over first "elems" elements of array "data"
5    *        with stride of "stride".
6    */
7   void test(int elems, int stride) /* The test function */
8   {
9       int i;
10      double result = 0.0;
11      volatile double sink;
12
13      for (i = 0; i < elems; i += stride) {
14          result += data[i];
15      }
16      sink = result; /* So compiler doesn't optimize away the loop */
17  }
18
19  /*
20   * run - Run test(elems, stride) and return read throughput (MB/s).
21   *       "size" is in bytes, "stride" is in array elements, and
22   *       Mhz is CPU clock frequency in Mhz.
23   */
24  double run(int size, int stride, double Mhz)
25  {
26      double cycles;
27      int elems = size / sizeof(double);
28
29      test(elems, stride);                    /* warm up the cache */
30      cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
31      return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
32  }
```

code/mem/mountain/mountain.c

Figure 6.42 **Functions that measure and compute read throughput.** We can generate a memory mountain for a particular computer by calling the `run` function with different values of `size` (which corresponds to temporal locality) and `stride` (which corresponds to spatial locality).

On each of the L2, L3, and main memory ridges, there is a slope of spatial locality that falls downhill as the stride increases, and spatial locality decreases. Notice that even when the working set is too large to fit in any of the caches, the highest point on the main memory ridge is a factor of 7 higher than its lowest point. So even when a program has poor temporal locality, spatial locality can still come to the rescue and make a significant difference.

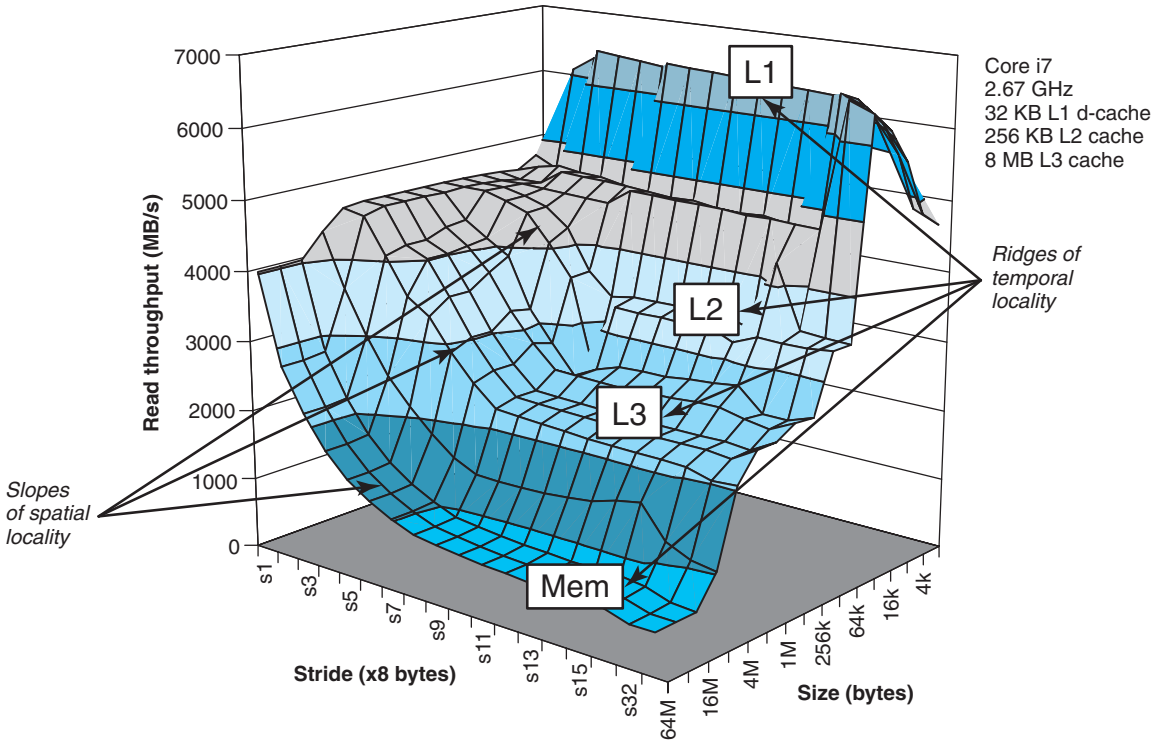


Figure 6.43 **The memory mountain.**

There is a particularly interesting flat ridge line that extends perpendicular to the stride axis for strides of 1 and 2, where the read throughput is a relatively constant 4.5 GB/s. This is apparently due to a hardware *prefetching* mechanism in the Core i7 memory system that automatically identifies memory referencing patterns and attempts to fetch those blocks into cache before they are accessed. While the details of the particular prefetching algorithm are not documented, it is clear from the memory mountain that the algorithm works best for small strides—yet another reason to favor sequential accesses in your code.

If we take a slice through the mountain, holding the stride constant as in Figure 6.44, we can see the impact of cache size and temporal locality on performance. For sizes up to 32 KB, the working set fits entirely in the L1 d-cache, and thus reads are served from L1 at the peak throughput of about 6 GB/s. For sizes up to 256 KB, the working set fits entirely in the unified L2 cache, and for sizes up to 8M, the working set fits entirely in the unified L3 cache. Larger working set sizes are served primarily from main memory.

The dips in read throughputs at the leftmost edges of the L1, L2, and L3 cache regions—where the working set sizes of 32 KB, 256 KB, and 8 MB are equal to their respective cache sizes—are interesting. It is not entirely clear why these dips occur. The only way to be sure is to perform a detailed cache simulation, but it

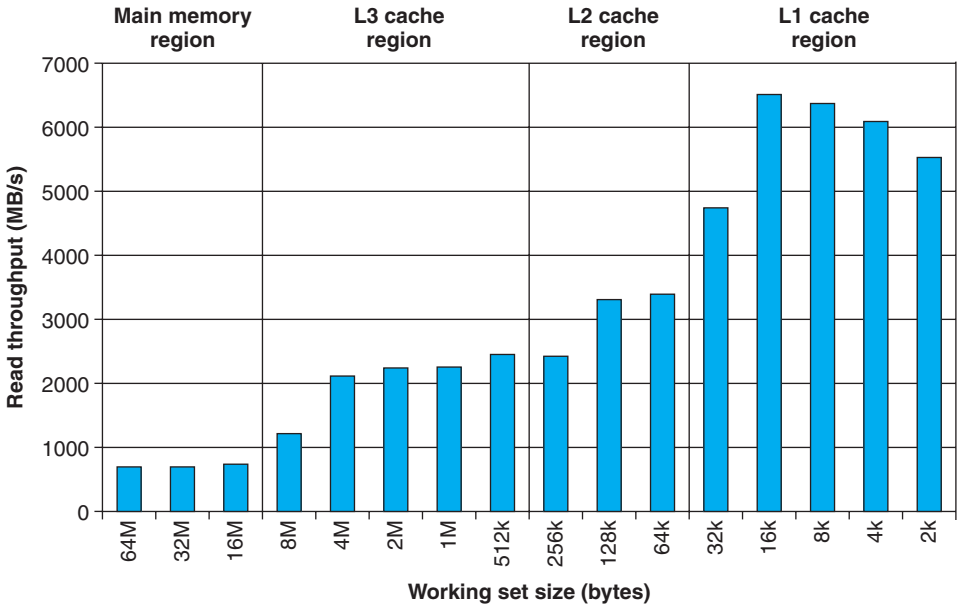


Figure 6.44 **Ridges of temporal locality in the memory mountain.** The graph shows a slice through Figure 6.43 with `stride=16`.

is likely that the drops are caused by other data and code blocks that make it impossible to fit the entire array in the respective cache.

Slicing through the memory mountain in the opposite direction, holding the working set size constant, gives us some insight into the impact of spatial locality on the read throughput. For example, Figure 6.45 shows the slice for a fixed working set size of 4 MB. This slice cuts along the L3 ridge in Figure 6.43, where the working set fits entirely in the L3 cache, but is too large for the L2 cache.

Notice how the read throughput decreases steadily as the stride increases from one to eight doublewords. In this region of the mountain, a read miss in L2 causes a block to be transferred from L3 to L2. This is followed by some number of hits on the block in L2, depending on the stride. As the stride increases, the ratio of L2 misses to L2 hits increases. Since misses are served more slowly than hits, the read throughput decreases. Once the stride reaches eight doublewords, which on this system equals the block size of 64 bytes, every read request misses in L2 and must be served from L3. Thus, the read throughput for strides of at least eight doublewords is a constant rate determined by the rate that cache blocks can be transferred from L3 into L2.

To summarize our discussion of the memory mountain, the performance of the memory system is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations can vary by over an order of magnitude. Wise programmers try to structure their programs so that they run in the peaks instead of the valleys. The aim is to exploit temporal locality so that

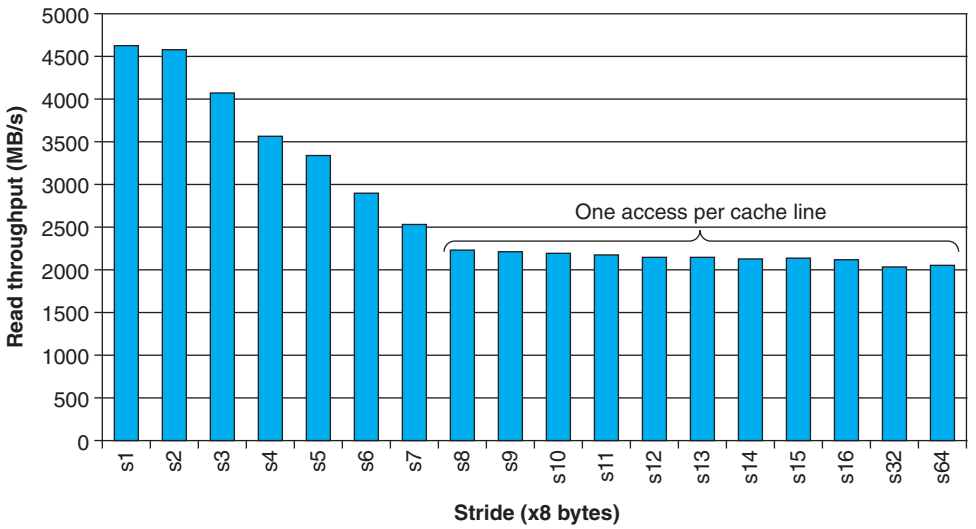


Figure 6.45 **A slope of spatial locality.** The graph shows a slice through Figure 6.43 with `size=4 MB`.

heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.

Practice Problem 6.22

Use the memory mountain in Figure 6.43 to estimate the time, in CPU cycles, to read an 8-byte word from the L1 d-cache.

6.6.2 Rearranging Loops to Increase Spatial Locality

Consider the problem of multiplying a pair of $n \times n$ matrices: $C = AB$. For example, if $n = 2$, then

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

where

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

A matrix multiply function is usually implemented using three nested loops, which are identified by their indexes i, j, and k. If we permute the loops and make some other minor code changes, we can create the six functionally equivalent versions

(a) Version ijk

code/mem/matmult/mm.c

```
1  for (i = 0; i < n; i++)
2      for (j = 0; j < n; j++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

code/mem/matmult/mm.c

(b) Version jik

code/mem/matmult/mm.c

```
1  for (j = 0; j < n; j++)
2      for (i = 0; i < n; i++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

code/mem/matmult/mm.c

(c) Version jki

code/mem/matmult/mm.c

```
1  for (j = 0; j < n; j++)
2      for (k = 0; k < n; k++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }
```

code/mem/matmult/mm.c

(d) Version kji

code/mem/matmult/mm.c

```
1  for (k = 0; k < n; k++)
2      for (j = 0; j < n; j++) {
3          r = B[k][j];
4          for (i = 0; i < n; i++)
5              C[i][j] += A[i][k]*r;
6      }
```

code/mem/matmult/mm.c

(e) Version kij

code/mem/matmult/mm.c

```
1  for (k = 0; k < n; k++)
2      for (i = 0; i < n; i++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }
```

code/mem/matmult/mm.c

(f) Version ikj

code/mem/matmult/mm.c

```
1  for (i = 0; i < n; i++)
2      for (k = 0; k < n; k++) {
3          r = A[i][k];
4          for (j = 0; j < n; j++)
5              C[i][j] += r*B[k][j];
6      }
```

code/mem/matmult/mm.c

Figure 6.46 **Six versions of matrix multiply.** Each version is uniquely identified by the ordering of its loops.

of matrix multiply shown in Figure 6.46. Each version is uniquely identified by the ordering of its loops.

At a high level, the six versions are quite similar. If addition is associative, then each version computes an identical result.[2] Each version performs $O(n^3)$ total

2. As we learned in Chapter 2, floating-point addition is commutative, but in general not associative. In practice, if the matrices do not mix extremely large values with extremely small ones, as often is true when the matrices store physical properties, then the assumption of associativity is reasonable.

Matrix multiply version (class)	Loads per iter.	Stores per iter.	A misses per iter.	B misses per iter.	C misses per iter.	Total misses per iter.
ijk & jik (AB)	2	0	0.25	1.00	0.00	1.25
jki & kji (AC)	2	1	1.00	0.00	1.00	2.00
kij & ikj (BC)	2	1	0.00	0.25	0.25	0.50

Figure 6.47 **Analysis of matrix multiply inner loops.** The six versions partition into three equivalence classes, denoted by the pair of arrays that are accessed in the inner loop.

operations and an identical number of adds and multiplies. Each of the n^2 elements of A and B is read n times. Each of the n^2 elements of C is computed by summing n values. However, if we analyze the behavior of the innermost loop iterations, we find that there are differences in the number of accesses and the locality. For the purposes of this analysis, we make the following assumptions:

- Each array is an $n \times n$ array of `double`, with `sizeof(double) == 8`.
- There is a single cache with a 32-byte block size ($B = 32$).
- The array size n is so large that a single matrix row does not fit in the L1 cache.
- The compiler stores local variables in registers, and thus references to local variables inside loops do not require any load or store instructions.

Figure 6.47 summarizes the results of our inner loop analysis. Notice that the six versions pair up into three equivalence classes, which we denote by the pair of matrices that are accessed in the inner loop. For example, versions ijk and jik are members of Class AB because they reference arrays A and B (but not C) in their innermost loop. For each class, we have counted the number of loads (reads) and stores (writes) in each inner loop iteration, the number of references to A, B, and C that will miss in the cache in each loop iteration, and the total number of cache misses per iteration.

The inner loops of the Class AB routines (Figure 6.46(a) and (b)) scan a row of array A with a stride of 1. Since each cache block holds four doublewords, the miss rate for A is 0.25 misses per iteration. On the other hand, the inner loop scans a column of B with a stride of n. Since n is large, each access of array B results in a miss, for a total of 1.25 misses per iteration.

The inner loops in the Class AC routines (Figure 6.46(c) and (d)) have some problems. Each iteration performs two loads and a store (as opposed to the Class AB routines, which perform two loads and no stores). Second, the inner loop scans the columns of A and C with a stride of n. The result is a miss on each load, for a total of two misses per iteration. Notice that interchanging the loops has decreased the amount of spatial locality compared to the Class AB routines.

The BC routines (Figure 6.46(e) and (f)) present an interesting trade-off: With two loads and a store, they require one more memory operation than the AB routines. On the other hand, since the inner loop scans both B and C row-wise

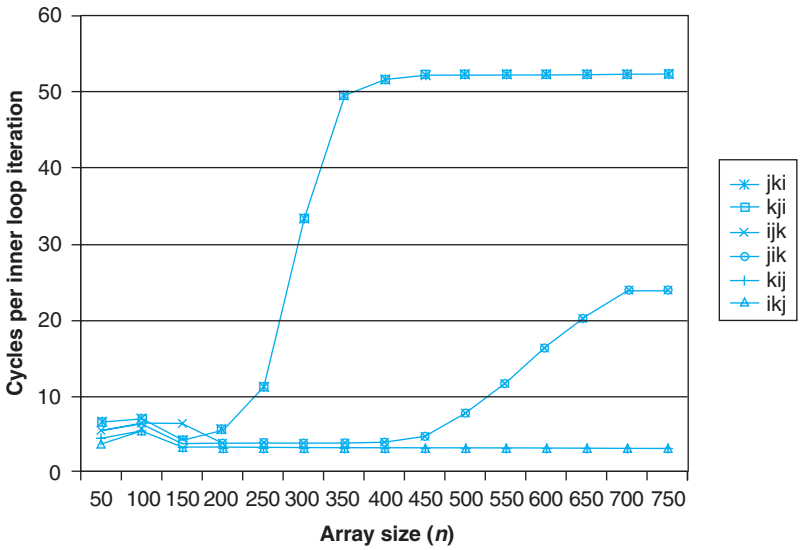


Figure 6.48
Core i7 matrix multiply performance. Legend: jki and kji: Class AC; ijk and jik: Class AB; kij and ikj: Class BC.

with a stride-1 access pattern, the miss rate on each array is only 0.25 misses per iteration, for a total of 0.50 misses per iteration.

Figure 6.48 summarizes the performance of different versions of matrix multiply on a Core i7 system. The graph plots the measured number of CPU cycles per inner loop iteration as a function of array size (n).

There are a number of interesting points to notice about this graph:

- For large values of n, the fastest version runs almost 20 times faster than the slowest version, even though each performs the same number of floating-point arithmetic operations.
- Pairs of versions with the same number of memory references and misses per iteration have almost identical measured performance.
- The two versions with the worst memory behavior, in terms of the number of accesses and misses per iteration, run significantly slower than the other four versions, which have fewer misses or fewer accesses, or both.
- Miss rate, in this case, is a better predictor of performance than the total number of memory accesses. For example, the Class BC routines, with 0.5 misses per iteration, perform much better than the Class AB routines, with 1.25 misses per iteration, even though the Class BC routines perform more memory references in the inner loop (two loads and one store) than the Class AB routines (two loads).
- For large values of n, the performance of the fastest pair of versions (kij and ikj) is constant. Even though the array is much larger than any of the SRAM cache memories, the prefetching hardware is smart enough to recognize the stride-1 access pattern, and fast enough to keep up with memory accesses in the tight inner loop. This is a stunning accomplishment by the Intel engi-

neers who designed this memory system, providing even more incentive for programmers to develop programs with good spatial locality.

Web Aside MEM:BLOCKING Using blocking to increase temporal locality

There is an interesting technique called *blocking* that can improve the temporal locality of inner loops. The general idea of blocking is to organize the data structures in a program into large chunks called *blocks*. (In this context, "block" refers to an application-level chunk of data, *not* to a cache block.) The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on.

Unlike the simple loop transformations for improving spatial locality, blocking makes the code harder to read and understand. For this reason, it is best suited for optimizing compilers or frequently executed library routines. Still, the technique is interesting to study and understand because it is a general concept that can produce big performance gains on some systems.

6.6.3 Exploiting Locality in Your Programs

As we have seen, the memory system is organized as a hierarchy of storage devices, with smaller, faster devices toward the top and larger, slower devices toward the bottom. Because of this hierarchy, the effective rate that a program can access memory locations is not characterized by a single number. Rather, it is a wildly varying function of program locality (what we have dubbed the memory mountain) that can vary by orders of magnitude. Programs with good locality access most of their data from fast cache memories. Programs with poor locality access most of their data from the relatively slow DRAM main memory.

Programmers who understand the nature of the memory hierarchy can exploit this understanding to write more efficient programs, regardless of the specific memory system organization. In particular, we recommend the following techniques:

- Focus your attention on the inner loops, where the bulk of the computations and memory accesses occur.
- Try to maximize the spatial locality in your programs by reading data objects sequentially, with stride 1, in the order they are stored in memory.
- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.

6.7 Summary

The basic storage technologies are random-access memories (RAMs), nonvolatile memories (ROMs), and disks. RAM comes in two basic forms. Static RAM (SRAM) is faster and more expensive, and is used for cache memories both on and off the CPU chip. Dynamic RAM (DRAM) is slower and less expensive, and is used for the main memory and graphics frame buffers. Nonvolatile memories, also called read-only memories (ROMs), retain their information even if the supply voltage is turned off, and they are used to store firmware. Rotating disks are

mechanical nonvolatile storage devices that hold enormous amounts of data at a low cost per bit, but with much longer access times than DRAM. Solid state disks (SSDs) based on nonvolatile flash memory are becoming increasingly attractive alternatives to rotating disks for some applications.

In general, faster storage technologies are more expensive per bit and have smaller capacities. The price and performance properties of these technologies are changing at dramatically different rates. In particular, DRAM and disk access times are much larger than CPU cycle times. Systems bridge these gaps by organizing memory as a hierarchy of storage devices, with smaller, faster devices at the top and larger, slower devices at the bottom. Because well-written programs have good locality, most data are served from the higher levels, and the effect is a memory system that runs at the rate of the higher levels, but at the cost and capacity of the lower levels.

Programmers can dramatically improve the running times of their programs by writing programs with good spatial and temporal locality. Exploiting SRAM-based cache memories is especially important. Programs that fetch data primarily from cache memories can run much faster than programs that fetch data primarily from memory.

Bibliographic Notes

Memory and disk technologies change rapidly. In our experience, the best sources of technical information are the Web pages maintained by the manufacturers. Companies such as Micron, Toshiba, and Samsung provide a wealth of current technical information on memory devices. The pages for Seagate, Maxtor, and Western Digital provide similarly useful information about disks.

Textbooks on circuit and logic design provide detailed information about memory technology [56, 85]. IEEE Spectrum published a series of survey articles on DRAM [53]. The International Symposium on Computer Architecture (ISCA) is a common forum for characterizations of DRAM memory performance [34, 35].

Wilkes wrote the first paper on cache memories [116]. Smith wrote a classic survey [101]. Przybylski wrote an authoritative book on cache design [82]. Hennessy and Patterson provide a comprehensive discussion of cache design issues [49].

Stricker introduced the idea of the memory mountain as a comprehensive characterization of the memory system in [111], and suggested the term "memory mountain" informally in later presentations of the work. Compiler researchers work to increase locality by automatically performing the kinds of manual code transformations we discussed in Section 6.6 [22, 38, 63, 68, 75, 83, 118]. Carter and colleagues have proposed a cache-aware memory controller [18]. Seward developed an open-source cache profiler, called `cacheprof`, that characterizes the miss behavior of C programs on an arbitrary simulated cache (`www.cacheprof.org`). Other researchers have developed *cache oblivious* algorithms that are designed to run well without any explicit knowledge of the structure of the underlying cache memory [36, 42, 43].

There is a large body of literature on building and using disk storage. Many storage researchers look for ways to aggregate individual disks into larger, more robust, and more secure storage pools [20, 44, 45, 79, 119]. Others look for ways to use caches and locality to improve the performance of disk accesses [12, 21]. Systems such as Exokernel provide increased user-level control of disk and memory resources [55]. Systems such as the Andrew File System [74] and Coda [91] extend the memory hierarchy across computer networks and mobile notebook computers. Schindler and Ganger developed an interesting tool that automatically characterizes the geometry and performance of SCSI disk drives [92]. Researchers are investigating techniques for building and using Flash-based SSDs [8, 77].

Homework Problems

6.23 ◆◆

Suppose you are asked to design a rotating disk where the number of bits per track is constant. You know that the number of bits per track is determined by the circumference of the innermost track, which you can assume is also the circumference of the hole. Thus, if you make the hole in the center of the disk larger, the number of bits per track increases, but the total number of tracks decreases. If you let r denote the radius of the platter, and $x \cdot r$ the radius of the hole, what value of x maximizes the capacity of the disk?

6.24 ◆

Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational rate	15,000 RPM
$T_{avg\ seek}$	4 ms
Average # sectors/track	800

6.25 ◆◆

Suppose that a 2 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Parameter	Value
Rotational rate	15,000 RPM
$T_{avg\ seek}$	4 ms
Average # sectors/track	1000
Surfaces	8
Sector size	512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{avg\ seek} + T_{avg\ rotation}$.

A. *Best case:* Estimate the optimal time (in ms) required to read the file over all possible mappings of logical blocks to disk sectors.

B. *Random case:* Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

6.26 ◆

The following table gives the parameters for a number of different caches. For each cache, fill in the missing fields in the table. Recall that m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, E is the associativity, S is the number of cache sets, t is the number of tag bits, s is the number of set index bits, and b is the number of block offset bits.

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

6.27 ◆

The following table gives the parameters for a number of different caches. Your task is to fill in the missing fields in the table. Recall that m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, E is the associativity, S is the number of cache sets, t is the number of tag bits, s is the number of set index bits, and b is the number of block offset bits.

Cache	m	C	B	E	S	t	s	b
1.	32		8	1		21	8	3
2.	32	2048			128	23	7	2
3.	32	1024	2	8	64			1
4.	32	1024		2	16	23	4	

6.28 ◆

This problem concerns the cache in Problem 6.13.

A. List all of the hex memory addresses that will hit in set 1.

B. List all of the hex memory addresses that will hit in set 6.

6.29 ◆◆

This problem concerns the cache in Problem 6.13.

A. List all of the hex memory addresses that will hit in set 2.

B. List all of the hex memory addresses that will hit in set 4.

C. List all of the hex memory addresses that will hit in set 5.

D. List all of the hex memory addresses that will hit in set 7.

6.30 ◆◆

Suppose we have a system with the following properties:

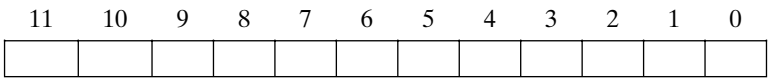
- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not to 4-byte words).
- Addresses are 12 bits wide.
- The cache is two-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and four sets ($S = 4$).

The contents of the cache are as follows, with all addresses, tags, and values given in hexadecimal notation:

Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	00	1	40	41	42	43
	83	1	FE	97	CC	D0
1	00	1	44	45	46	47
	83	0	—	—	—	—
2	00	1	48	49	4A	4B
	40	0	—	—	—	—
3	FF	1	9A	C0	03	FF
	00	0	—	—	—	—

A. The following diagram shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO The cache block offset
CI The cache set index
CT The cache tag



B. For each of the following memory accesses indicate if it will be a cache hit or miss when **carried out in sequence** as listed. Also give the value of a read if it can be inferred from the information in the cache.

Operation	Address	Hit?	Read value (or unknown)
Read	0x834	_____	_____
Write	0x836	_____	_____
Read	0xFFD	_____	_____

6.31 ◆

Suppose we have a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is four-way set associative ($E = 4$), with a 4-byte block size ($B = 4$) and eight sets ($S = 8$).

Consider the following cache state. All addresses, tags, and values are given in hexadecimal format. The *Index* column contains the set index for each set of four lines. The *Tag* columns contain the tag value for each line. The *V* columns contain the valid bit for each line. The *Bytes 0–3* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

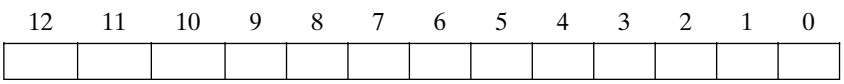
4-way set associative cache

Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3
0	F0	1	ED 32 0A A2	8A	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	0	25 44 6F 1A
1	BC	0	03 3E CD 38	A0	0	16 7B ED 5A	BC	1	8E 4C DF 18	E4	1	FB B7 12 02
2	BC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E
3	BE	0	2F 7E 3D A8	C0	1	27 95 A4 74	C4	0	07 11 6B D8	BC	0	C7 B7 AF C2
4	7E	1	32 21 1C 2C	8A	1	22 C2 DC 34	BC	1	BA DD 37 D8	DC	0	E7 A2 39 BA
5	98	0	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	BC	1	48 16 81 0A
6	38	0	5D 4D F7 DA	BC	1	69 C2 8C 74	8A	1	A8 CE 7F DA	38	1	FA 93 EB 48
7	8A	1	04 2A 32 6A	9E	0	B1 86 56 0E	CC	1	96 30 47 F2	BC	1	F8 1D 42 30

A. What is size (C) of this cache in bytes?

B. The box that follows shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

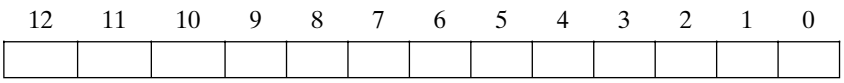
CO The cache block offset
CI The cache set index
CT The cache tag



6.32 ◆◆

Supppose that a program using the cache in Problem 6.31 references the 1-byte word at address `0x071A`. Indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned". *Hint: Pay attention to those valid bits!*

A. Address format (one bit per box):



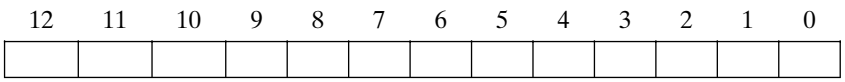
B. Memory reference:

Parameter	Value
Block offset (CO)	0x_____
Index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

6.33 ◆◆

Repeat Problem 6.32 for memory address 0x16E8.

A. Address format (one bit per box):



B. Memory reference:

Parameter	Value
Cache offset (CO)	0x_____
Cache index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

6.34 ◆◆

For the cache in Problem 6.31, list the eight memory addresses (in hex) that will hit in set 2.

6.35 ◆◆

Consider the following matrix transpose routine:

```
1   typedef int array[4][4];
2
3   void transpose2(array dst, array src)
4   {
5       int i, j;
6
7       for (i = 0; i < 4; i++) {
8           for (j = 0; j < 4; j++) {
9               dst[j][i] = src[i][j];
10          }
11      }
12  }
```

Assume this code runs on a machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate, with a block size of 16 bytes.
- The cache has a total size of 32 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`dst` array

	Col 0	Col 1	Col 2	Col 3
Row 0	m			
Row 1				
Row 2				
Row 3				

`src` array

	Col 0	Col 1	Col 2	Col 3
Row 0	m			
Row 1				
Row 2				
Row 3				

6.36 ◆◆

Repeat Problem 6.35 for a cache with a total size of 128 data bytes.

`dst` array

	Col 0	Col 1	Col 2	Col 3
Row 0				
Row 1				
Row 2				
Row 3				

`src` array

	Col 0	Col 1	Col 2	Col 3
Row 0				
Row 1				
Row 2				
Row 3				

6.37 ◆◆

This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```
1       int x[2][128];
2       int i;
3       int sum = 0;
4
5       for (i = 0; i < 128; i++) {
6           sum += x[0][i] * x[1][i];
7       }
```

Assume we execute this under the following conditions:

- `sizeof(int) = 4`.
- Array `x` begins at memory address `0x0` and is stored in row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:

A. Case 1: Assume the cache is 512 bytes, direct-mapped, with 16-byte cache blocks. What is the miss rate?

B. Case 2: What is the miss rate if we double the cache size to 1024 bytes?

C. Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate?

D. For Case 3, will a larger cache size help to reduce the miss rate? Why or why not?

E. For Case 3, will a larger block size help to reduce the miss rate? Why or why not?

6.38 ◆◆

This is another problem that tests your ability to analyze the cache behavior of C code. Assume we execute the three summation functions in Figure 6.49 under the following conditions:

- `sizeof(int) == 4`.
- The machine has a 4KB direct-mapped cache with a 16-byte block size.
- Within the two loops, the code uses memory accesses only for the array data. The loop indices and the value `sum` are held in registers.
- Array `a` is stored starting at memory address `0x08000000`.

Fill in the table for the approximate cache miss rate for the two cases $N = 64$ and $N = 60$.

Function	`N = 64`	`N = 60`
`sumA`	_____	_____
`sumB`	_____	_____
`sumC`	_____	_____

6.39 ◆

3M™ decides to make Post-It® notes by printing yellow squares on white pieces of paper. As part of the printing process, they need to set the CMYK (cyan, magenta, yellow, black) value for every point in the square. 3M hires you to determine

```
typedef int array_t[N][N];

int sumA(array_t a)
{
    int i, j;
    int sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    return sum;
}

int sumB(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++) {
            sum += a[i][j];
        }
    return sum;
}

int sumC(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j+=2)
        for (i = 0; i < N; i+=2) {
            sum += (a[i][j] + a[i+1][j]
                    + a[i][j+1] + a[i+1][j+1]);
        }
    return sum;
}
```

Figure 6.49 **Functions referenced in Problem 6.38.**

the efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks. You are given the following definitions:

```
struct point_color {
    int c;
    int m;
    int y;
    int k;
};
```

```
7
8   struct point_color square[16][16];
9   int i, j;
```

Assume the following:

- `sizeof(int) == 4`.
- `square` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `square`. Variables `i` and `j` are stored in registers.

Determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               square[i][j].c = 0;
4               square[i][j].m = 0;
5               square[i][j].y = 1;
6               square[i][j].k = 0;
7           }
8       }
```

A. What is the total number of writes?

B. What is the total number of writes that miss in the cache?

C. What is the miss rate?

6.40 ◆

Given the assumptions in Problem 6.39, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               square[j][i].c = 0;
4               square[j][i].m = 0;
5               square[j][i].y = 1;
6               square[j][i].k = 0;
7           }
8       }
```

A. What is the total number of writes?

B. What is the total number of writes that miss in the cache?

C. What is the miss rate?

6.41 ◆

Given the assumptions in Problem 6.39, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++) {
2           for (j = 0; j < 16; j++) {
3               square[i][j].y = 1;
4           }
5       }
6       for (i = 0; i < 16; i++) {
7           for (j = 0; j < 16; j++) {
8               square[i][j].c = 0;
9               square[i][j].m = 0;
10              square[i][j].k = 0;
11          }
12      }
```

A. What is the total number of writes?

B. What is the total number of writes that miss in the cache?

C. What is the miss rate?

6.42 ◆◆

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640 × 480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines. The C structures you are using are as follows:

```
1   struct pixel {
2       char r;
3       char g;
4       char b;
5       char a;
6   };
7
8   struct pixel buffer[480][640];
9   int i, j;
10  char *cptr;
11  int *iptr;
```

Assume the following:

- `sizeof(char)` == 1 and `sizeof(int)` == 4.
- `buffer` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

What percentage of writes in the following code will miss in the cache?

```
1       for (j = 0; j < 640; j++) {
2           for (i = 0; i < 480; i++){
3               buffer[i][j].r = 0;
4               buffer[i][j].g = 0;
5               buffer[i][j].b = 0;
6               buffer[i][j].a = 0;
7           }
8       }
```

6.43 ◆◆

Given the assumptions in Problem 6.42, what percentage of writes in the following code will miss in the cache?

```
1       char *cptr = (char *) buffer;
2       for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
3           *cptr = 0;
```

6.44 ◆◆

Given the assumptions in Problem 6.42, what percentage of writes in the following code will miss in the cache?

```
1       int *iptr = (int *)buffer;
2       for (; iptr < ((int *)buffer + 640*480); iptr++)
3           *iptr = 0;
```

6.45 ◆◆◆

Download the `mountain` program from the CS:APP2 Web site and run it on your favorite PC/Linux system. Use the results to estimate the sizes of the caches on your system.

6.46 ◆◆◆◆

In this assignment, you will apply the concepts you learned in Chapters 5 and 6 to the problem of optimizing code for a memory-intensive application. Consider a procedure to copy and transpose the elements of an $N \times N$ matrix of type `int`. That is, for source matrix S and destination matrix D, we want to copy each element $s_{i,j}$ to $d_{j,i}$. This code can be written with a simple loop,

```
1   void transpose(int *dst, int *src, int dim)
2   {
3       int i, j;
4
5       for (i = 0; i < dim; i++)
6           for (j = 0; j < dim; j++)
7               dst[j*dim + i] = src[i*dim + j];
8   }
```

where the arguments to the procedure are pointers to the destination (`dst`) and source (`src`) matrices, as well as the matrix size N (`dim`). Your job is to devise a transpose routine that runs as fast as possible.

6.47 ◆◆◆◆

This assignment is an intriguing variation of Problem 6.46. Consider the problem of converting a directed graph g into its undirected counterpart g'. The graph g' has an edge from vertex u to vertex v if and only if there is an edge from u to v or from v to u in the original graph g. The graph g is represented by its *adjacency matrix* G as follows. If N is the number of vertices in g, then G is an $N \times N$ matrix and its entries are all either 0 or 1. Suppose the vertices of g are named $v_0, v_1, v_2, \ldots, v_{N-1}$. Then $G[i][j]$ is 1 if there is an edge from v_i to v_j and is 0 otherwise. Observe that the elements on the diagonal of an adjacency matrix are always 1 and that the adjacency matrix of an undirected graph is symmetric. This code can be written with a simple loop:

```
1  void col_convert(int *G, int dim) {
2      int i, j;
3
4      for (i = 0; i < dim; i++)
5          for (j = 0; j < dim; j++)
6              G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
7  }
```

Your job is to devise a conversion routine that runs as fast as possible. As before, you will need to apply concepts you learned in Chapters 5 and 6 to come up with a good solution.

Solutions to Practice Problems

Solution to Problem 6.1 (page 565)

The idea here is to minimize the number of address bits by minimizing the aspect ratio $\max(r, c)/\min(r, c)$. In other words, the squarer the array, the fewer the address bits.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1	4	4	2	2	2
16×4	4	4	2	2	2
128×8	16	8	4	3	4
512×4	32	16	5	4	5
1024×4	32	32	5	5	5

Solution to Problem 6.2 (page 573)

The point of this little drill is to make sure you understand the relationship between cylinders and tracks. Once you have that straight, just plug and chug:

$$\begin{aligned}\text{Disk capacity} &= \frac{512 \text{ bytes}}{\text{sector}} \times \frac{400 \text{ sectors}}{\text{track}} \times \frac{10{,}000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{2 \text{ platters}}{\text{disk}} \\ &= 8{,}192{,}000{,}000 \text{ bytes} \\ &= 8.192 \text{ GB}\end{aligned}$$

Solution to Problem 6.3 (page 575)

This solution to this problem is a straightforward application of the formula for disk access time. The average rotational latency (in ms) is

$$\begin{aligned}T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60 \text{ secs} / 15{,}000 \text{ RPM}) \times 1000 \text{ ms/sec} \\ &\approx 2 \text{ ms}\end{aligned}$$

The average transfer time is

$$\begin{aligned}T_{avg\ transfer} &= (60 \text{ secs} / 15{,}000 \text{ RPM}) \times 1/500 \text{ sectors/track} \times 1000 \text{ ms/sec} \\ &\approx 0.008 \text{ ms}\end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned}T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\ &= 8 \text{ ms} + 2 \text{ ms} + 0.008 \text{ ms} \\ &\approx 10 \text{ ms}\end{aligned}$$

Solution to Problem 6.4 (page 576)

This is a good check of your understanding of the factors that affect disk performance. First we need to determine a few basic properties of the file and the disk. The file consists of 2000, 512-byte logical blocks. For the disk, $T_{avg\ seek}$ = 5 ms, $T_{max\ rotation}$ = 6 ms, and $T_{avg\ rotation}$ = 3 ms.

A. *Best case:* In the optimal case, the blocks are mapped to contiguous sectors, on the same cylinder, that can be read one after the other without moving the head. Once the head is positioned over the first sector it takes two full rotations (1000 sectors per rotation) of the disk to read all 2000 blocks. So the total time to read the file is $T_{avg\ seek} + T_{avg\ rotation} + 2 * T_{max\ rotation} = 5 + 3 + 12 = 20$ ms.

B. *Random case:* In this case, where blocks are mapped randomly to sectors, reading each of the 2000 blocks requires $T_{avg\ seek} + T_{avg\ rotation}$ ms, so the total time to read the file is $(T_{avg\ seek} + T_{avg\ rotation}) * 2000 = 16{,}000$ ms (16 seconds!).

You can see now why it's often a good idea to defragment your disk drive!

Solution to Problem 6.5 (page 581)

This problem, based on the zone map in Figure 6.14, is a good test of your understanding of disk geometry, and it also enables you to derive an interesting characteristic of a real disk drive.

A. Zone 0. There are a total of $864 \times 8 \times 3201 = 22{,}125{,}312$ sectors and 22,076,928 logical blocks assigned to zone 0, for a total of $22{,}125{,}312 - 22{,}076{,}928 = 48{,}384$ spare sectors. Given that there are $864 \times 8 = 6912$ sectors per cylinder, there are $48{,}384/6912 = 7$ spare cylinders in zone 0.

B. Zone 8. A similar analysis reveals there are $((3700 \times 5632) - 20{,}804{,}608)/5632 = 6$ spare cylinders in zone 8.

Solution to Problem 6.6 (page 583)

This is a simple problem that will give you some interesting insights into feasibility of SSDs. Recall that for disks, 1 PB = 10^9 MB. Then the following straightforward translation of units yields the following predicted times for each case:

A. Worst case sequential writes (170 MB/s): $10^9 \times (1/170) \times (1/(86{,}400 \times 365)) \approx 0.2$ years.

B. Worst case random writes (14 MB/s): $10^9 \times (1/14) \times (1/(86{,}400 \times 365)) \approx 2.25$ years.

C. Average case (20 GB/day): $10^9 \times (1/20{,}000) \times (1/365) \approx 140$ years.

Solution to Problem 6.7 (page 586)

In the 10-year period between 2000 and 2010, the unit price of rotating disk dropped by a factor of about 30, which means the price is dropping by roughly a factor of 2 every 2 years. Assuming this trend continues, a petabyte of storage, which costs about $300,000 in 2010, will drop below $500 after about ten of these factor-of-2 reductions. Since these are occurring every 2 years, we can expect a petabyte of storage to be available for $500 around the year 2030.

Solution to Problem 6.8 (page 590)

To create a stride-1 reference pattern, the loops must be permuted so that the rightmost indices change most rapidly.

```
1   int sumarray3d(int a[N][N][N])
2   {
3       int i, j, k, sum = 0;
4
5       for (k = 0; k < N; k++) {
6           for (i = 0; i < N; i++) {
7               for (j = 0; j < N; j++) {
8                   sum += a[k][i][j];
9               }
10          }
11      }
12      return sum;
13  }
```

This is an important idea. Make sure you understand why this particular loop permutation results in a stride-1 access pattern.

Solution to Problem 6.9 (page 590)

The key to solving this problem is to visualize how the array is laid out in memory and then analyze the reference patterns. Function `clear1` accesses the array using a stride-1 reference pattern and thus clearly has the best spatial locality. Function `clear2` scans each of the N structs in order, which is good, but within each struct it hops around in a non-stride-1 pattern at the following offsets from the beginning of the struct: 0, 12, 4, 16, 8, 20. So `clear2` has worse spatial locality than `clear1`. Function `clear3` not only hops around within each struct, but it also hops from struct to struct. So `clear3` exhibits worse spatial locality than `clear2` and `clear1`.

Solution to Problem 6.10 (page 598)

The solution is a straightforward application of the definitions of the various cache parameters in Figure 6.28. Not very exciting, but you need to understand how the cache organization induces these partitions in the address bits before you can really understand how caches work.

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1	256	22	8	2
2.	32	1024	8	4	32	24	5	3
3.	32	1024	32	32	1	27	0	5

Solution to Problem 6.11 (page 605)

The padding eliminates the conflict misses. Thus, three-fourths of the references are hits.

Solution to Problem 6.12 (page 605)

Sometimes, understanding why something is a bad idea helps you understand why the alternative is a good idea. Here, the bad idea we are looking at is indexing the cache with the high-order bits instead of the middle bits.

A. With high-order bit indexing, each contiguous array chunk consists of 2^t blocks, where t is the number of tag bits. Thus, the first 2^t contiguous blocks of the array would map to set 0, the next 2^t blocks would map to set 1, and so on.

B. For a direct-mapped cache where $(S, E, B, m) = (512, 1, 32, 32)$, the cache capacity is 512 32-byte blocks, and there are $t = 18$ tag bits in each cache line. Thus, the first 2^{18} blocks in the array would map to set 0, the next 2^{18} blocks to set 1. Since our array consists of only $(4096 * 4)/32 = 512$ blocks, all of the blocks in the array map to set 0. Thus, the cache will hold at most one array block at any point in time, even though the array is small enough to fit

entirely in the cache. Clearly, using high-order bit indexing makes poor use of the cache.

Solution to Problem 6.13 (page 609)

The 2 low-order bits are the block offset (CO), followed by 3 bits of set index (CI), with the remaining bits serving as the tag (CT):

12	11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

Solution to Problem 6.14 (page 610)

Address: `0x0E34`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	1	1	0	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x0`
Cache set index (CI)	`0x5`
Cache tag (CT)	`0x71`
Cache hit? (Y/N)	Y
Cache byte returned	`0xB`

Solution to Problem 6.15 (page 611)

Address: `0x0DD5`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	0	1	0	1
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Cache block offset (CO)	`0x1`
Cache set index (CI)	`0x5`
Cache tag (CT)	`0x6E`
Cache hit? (Y/N)	N
Cache byte returned	—

Solution to Problem 6.16 (page 611)

Address: `0x1FE4`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Cache block offset	`0x0`
Cache set index	`0x1`
Cache tag	`0xFF`
Cache hit? (Y/N)	N
Cache byte returned	—

Solution to Problem 6.17 (page 611)

This problem is a sort of inverse version of Problems 6.13–6.16 that requires you to work backward from the contents of the cache to derive the addresses that will hit in a particular set. In this case, set 3 contains one valid line with a tag of `0x32`. Since there is only one valid line in the set, four addresses will hit. These addresses have the binary form `0 0110 0100 11xx`. Thus, the four hex addresses that hit in set 3 are

`0x064C`, `0x064D`, `0x064E`, and `0x064F`

Solution to Problem 6.18 (page 618)

A. The key to solving this problem is to visualize the picture in Figure 6.50. Notice that each cache line holds exactly one row of the array, that the cache is exactly large enough to hold one array, and that for all i, row i of `src` and `dst` maps to the same cache line. Because the cache is too small to hold both arrays, references to one array keep evicting useful lines from the other array. For example, the write to `dst[0][0]` evicts the line that was loaded when we read `src[0][0]`. So when we next read `src[0][1]`, we have a miss.

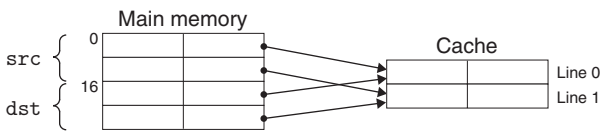
`dst` array

	Col 0	Col 1
Row 0	m	m
Row 1	m	m

`src` array

	Col 0	Col 1
Row 0	m	m
Row 1	m	h

Figure 6.50
Figure for Problem 6.18.



B. When the cache is 32 bytes, it is large enough to hold both arrays. Thus, the only misses are the initial cold misses.

`dst` array

	Col 0	Col 1
Row 0	m	h
Row 1	m	h

`src` array

	Col 0	Col 1
Row 0	m	h
Row 1	m	h

Solution to Problem 6.19 (page 619)

Each 16-byte cache line holds two contiguous `algae_position` structures. Each loop visits these structures in memory order, reading one integer element each time. So the pattern for each loop is miss, hit, miss, hit, and so on. Notice that for this problem we could have predicted the miss rate without actually enumerating the total number of reads and misses.

A. What is the total number of read accesses? 512 reads.

B. What is the total number of read accesses that miss in the cache? 256 misses.

C. What is the miss rate? $256/512 = 50\%$.

Solution to Problem 6.20 (page 620)

The key to this problem is noticing that the cache can only hold 1/2 of the array. So the column-wise scan of the second half of the array evicts the lines that were loaded during the scan of the first half. For example, reading the first element of `grid[8][0]` evicts the line that was loaded when we read elements from `grid[0][0]`. This line also contained `grid[0][1]`. So when we begin scanning the next column, the reference to the first element of `grid[0][1]` misses.

A. What is the total number of read accesses? 512 reads.

B. What is the total number of read accesses that miss in the cache? 256 misses.

C. What is the miss rate? $256/512 = 50\%$.

D. What would the miss rate be if the cache were twice as big? If the cache were twice as big, it could hold the entire `grid` array. The only misses would be the initial cold misses, and the miss rate would be $1/4 = 25\%$.

Solution to Problem 6.21 (page 620)

This loop has a nice stride-1 reference pattern, and thus the only misses are the initial cold misses.

A. What is the total number of read accesses? 512 reads.

B. What is the total number of read accesses that miss in the cache? 128 misses.

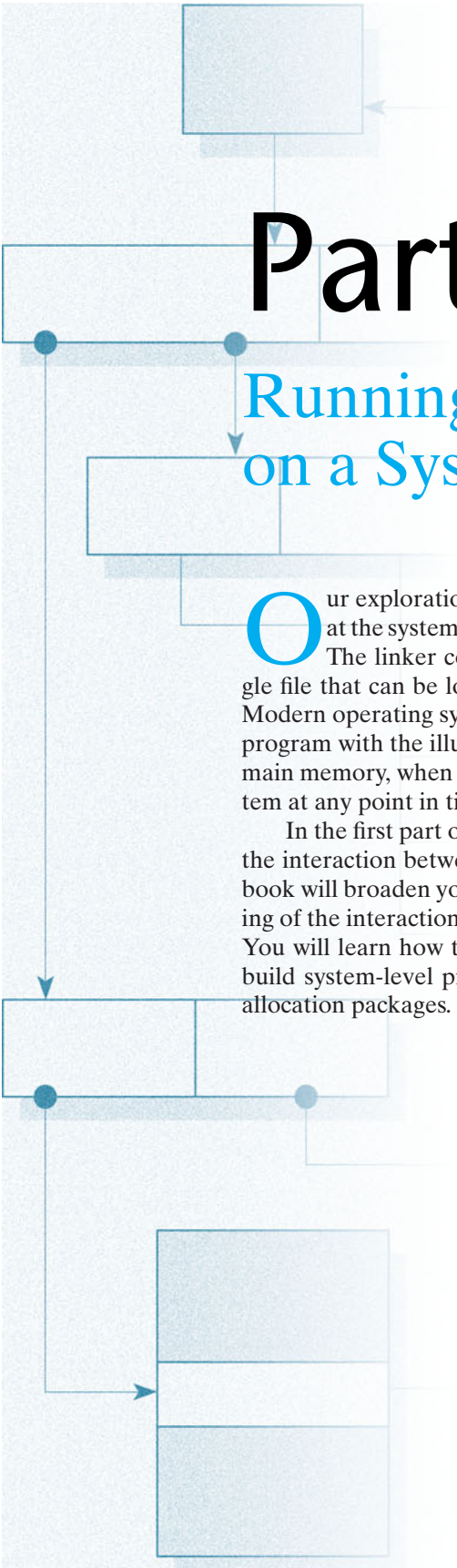
C. What is the miss rate? $128/512 = 25\%$.

D. What would the miss rate be if the cache were twice as big? Increasing the cache size by any amount would not change the miss rate, since cold misses are unavoidable.

Solution to Problem 6.22 (page 625)

The peak throughput from L1 is about 6500 MB/s, the clock frequency is 2670 MHz, and the individual read accesses are in units of 8-byte `double`s. Thus, from this graph we can estimate that it takes roughly $2670/6500 \times 8 = 3.2 \approx 4$ cycles to access a word from L1 on this machine.

This page intentionally left blank



Part II

Running Programs on a System

Our exploration of computer systems continues with a closer look at the systems software that builds and runs application programs. The linker combines different parts of our programs into a single file that can be loaded into memory and executed by the processor. Modern operating systems cooperate with the hardware to provide each program with the illusion that it has exclusive use of a processor and the main memory, when in reality, multiple programs are running on the system at any point in time.

In the first part of this book, you developed a good understanding of the interaction between your programs and the hardware. Part II of the book will broaden your view of systems by giving you a solid understanding of the interactions between your programs and the operating system. You will learn how to use services provided by the operating system to build system-level programs such as Unix shells and dynamic memory allocation packages.

This page intentionally left blank

CHAPTER 7

Linking

Linking is the process of collecting and combining various pieces of code and data into a single file that can be *loaded* (copied) into memory and executed. Linking can be performed at *compile time*, when the source code is translated into machine code; at *load time*, when the program is loaded into memory and executed by the *loader*; and even at *run time*, by application programs. On early computer systems, linking was performed manually. On modern systems, linking is performed automatically by programs called *linkers*.

Linkers play a crucial role in software development because they enable *separate compilation*. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files.

Linking is usually handled quietly by the linker, and is not an important issue for students who are building small programs in introductory programming classes. So why bother learning about linking?

- *Understanding linkers will help you build large programs.* Programmers who build large programs often encounter linker errors caused by missing modules, missing libraries, or incompatible library versions. Unless you understand how a linker resolves references, what a library is, and how a linker uses a library to resolve references, these kinds of errors will be baffling and frustrating.
- *Understanding linkers will help you avoid dangerous programming errors.* The decisions that Unix linkers make when they resolve symbol references can silently affect the correctness of your programs. Programs that incorrectly define multiple global variables pass through the linker without any warnings in the default case. The resulting programs can exhibit baffling run-time behavior and are extremely difficult to debug. We will show you how this happens and how to avoid it.
- *Understanding linking will help you understand how language scoping rules are implemented.* For example, what is the difference between global and local variables? What does it really mean when you define a variable or function with the `static` attribute?
- *Understanding linking will help you understand other important systems concepts.* The executable object files produced by linkers play key roles in important systems functions such as loading and running programs, virtual memory, paging, and memory mapping.
- *Understanding linking will enable you to exploit shared libraries.* For many years, linking was considered to be fairly straightforward and uninteresting. However, with the increased importance of shared libraries and dynamic linking in modern operating systems, linking is a sophisticated process that provides the knowledgeable programmer with significant power. For example, many software products use shared libraries to upgrade shrink-wrapped binaries at run time. Also, most Web servers rely on dynamic linking of shared libraries to serve dynamic content.

This chapter provides a thorough discussion of all aspects of linking, from traditional static linking, to dynamic linking of shared libraries at load time, to dynamic linking of shared libraries at run time. We will describe the basic mechanisms using real examples, and we will identify situations in which linking issues can affect the performance and correctness of your programs.

To keep things concrete and understandable, we will couch our discussion in the context of an x86 system running Linux and using the standard ELF object file format. For clarity, we will focus our discussion on linking 32-bit code, which is easier to understand than linking 64-bit code.[1] However, it is important to realize that the basic concepts of linking are universal, regardless of the operating system, the ISA, or the object file format. Details may vary, but the concepts are the same.

7.1 Compiler Drivers

Consider the C program in Figure 7.1. It consists of two source files, `main.c` and `swap.c`. Function `main()` calls `swap`, which swaps the two elements in the external global array `buf`. Granted, this is a strange way to swap two numbers, but it will serve as a small running example throughout this chapter that will allow us to make some important points about how linking works.

Most compilation systems provide a *compiler driver* that invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user. For example, to build the example program using the GNU compilation system, we might invoke the GCC driver by typing the following command to the shell:

```
unix> gcc -O2 -g -o p main.c swap.c
```

Figure 7.2 summarizes the activities of the driver as it translates the example program from an ASCII source file into an executable object file. (If you want to see these steps for yourself, run GCC with the `-v` option.) The driver first runs the C preprocessor (`cpp`), which translates the C source file `main.c` into an ASCII intermediate file `main.i`:

```
cpp [other arguments] main.c /tmp/main.i
```

Next, the driver runs the C compiler (`cc1`), which translates `main.i` into an ASCII assembly language file `main.s`.

```
cc1 /tmp/main.i main.c -O2 [other arguments] -o /tmp/main.s
```

Then, the driver runs the assembler (`as`), which translates `main.s` into a *relocatable object file* `main.o`:

```
as [other arguments] -o /tmp/main.o /tmp/main.s
```

1. You can generate 32-bit code on an x86-64 system using `gcc -m32`.

(a) `main.c`

code/link/main.c

```
1   /* main.c */
2   void swap();
3
4   int buf[2] = {1, 2};
5
6   int main()
7   {
8       swap();
9       return 0;
10  }
```

code/link/main.c

(b) `swap.c`

code/link/swap.c

```
1   /* swap.c */
2   extern int buf[];
3
4   int *bufp0 = &buf[0];
5   int *bufp1;
6
7   void swap()
8   {
9       int temp;
10
11      bufp1 = &buf[1];
12      temp = *bufp0;
13      *bufp0 = *bufp1;
14      *bufp1 = temp;
15  }
```

code/link/swap.c

Figure 7.1 **Example program 1:** The example program consists of two source files, `main.c` and `swap.c`. The `main` function initializes a two-element array of ints, and then calls the `swap` function to swap the pair.

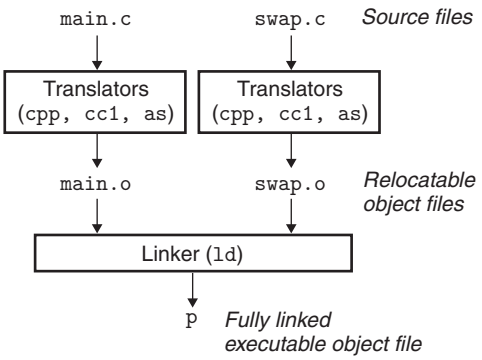


Figure 7.2 **Static linking.** The linker combines relocatable object files to form an executable object file `p`.

The driver goes through the same process to generate `swap.o`. Finally, it runs the linker program `ld`, which combines `main.o` and `swap.o`, along with the necessary system object files, to create the *executable object file* `p`:

```
ld -o p [system object files and args] /tmp/main.o /tmp/swap.o
```

To run the executable `p`, we type its name on the Unix shell's command line:

```
unix> ./p
```

The shell invokes a function in the operating system called the *loader*, which copies the code and data in the executable file `p` into memory, and then transfers control to the beginning of the program.

7.2 Static Linking

Static linkers such as the Unix `ld` program take as input a collection of relocatable object files and command-line arguments and generate as output a fully linked executable object file that can be loaded and run. The input relocatable object files consist of various code and data sections. Instructions are in one section, initialized global variables are in another section, and uninitialized variables are in yet another section.

To build the executable, the linker must perform two main tasks:

- *Symbol resolution.* Object files define and reference *symbols*. The purpose of symbol resolution is to associate each symbol reference with exactly one symbol definition.
- *Relocation.* Compilers and assemblers generate code and data sections that start at address 0. The linker *relocates* these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location.

The sections that follow describe these tasks in more detail. As you read, keep in mind some basic facts about linkers: Object files are merely collections of blocks of bytes. Some of these blocks contain program code, others contain program data, and others contain data structures that guide the linker and loader. A linker concatenates blocks together, decides on run-time locations for the concatenated blocks, and modifies various locations within the code and data blocks. Linkers have minimal understanding of the target machine. The compilers and assemblers that generate the object files have already done most of the work.

7.3 Object Files

Object files come in three forms:

- *Relocatable object file.* Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.
- *Executable object file.* Contains binary code and data in a form that can be copied directly into memory and executed.
- *Shared object file.* A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time.

Compilers and assemblers generate relocatable object files (including shared object files). Linkers generate executable object files. Technically, an *object module*

is a sequence of bytes, and an *object file* is an object module stored on disk in a file. However, we will use these terms interchangeably.

Object file formats vary from system to system. The first Unix systems from Bell Labs used the `a.out` format. (To this day, executables are still referred to as `a.out` files.) Early versions of System V Unix used the Common Object File format (COFF). Windows NT uses a variant of COFF called the Portable Executable (PE) format. Modern Unix systems—such as Linux, later versions of System V Unix, BSD Unix variants, and Sun Solaris—use the Unix *Executable and Linkable Format* (ELF). Although our discussion will focus on ELF, the basic concepts are similar, regardless of the particular format.

7.4 Relocatable Object Files

Figure 7.3 shows the format of a typical ELF relocatable object file. The *ELF header* begins with a 16-byte sequence that describes the word size and byte ordering of the system that generated the file. The rest of the ELF header contains information that allows a linker to parse and interpret the object file. This includes the size of the ELF header, the object file type (e.g., relocatable, executable, or shared), the machine type (e.g., IA32), the file offset of the section header table, and the size and number of entries in the section header table. The locations and sizes of the various sections are described by the *section header table*, which contains a fixed sized entry for each section in the object file.

Sandwiched between the ELF header and the section header table are the sections themselves. A typical ELF relocatable object file contains the following sections:

`.text`: The machine code of the compiled program.

`.rodata`: Read-only data such as the format strings in `printf` statements, and jump tables for switch statements (see Problem 7.14).

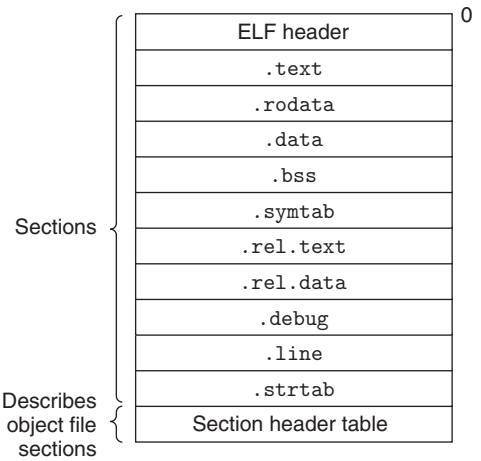


Figure 7.3
Typical ELF relocatable object file.

`.data`: *Initialized* global C variables. Local C variables are maintained at run time on the stack, and do `not` appear in either the `.data` or `.bss` sections.

`.bss`: *Uninitialized* global C variables. This section occupies no actual space in the object file; it is merely a place holder. Object file formats distinguish between initialized and uninitialized variables for space efficiency: uninitialized variables do not have to occupy any actual disk space in the object file.

`.symtab`: A *symbol table* with information about functions and global variables that are defined and referenced in the program. Some programmers mistakenly believe that a program must be compiled with the `-g` option to get symbol table information. In fact, every relocatable object file has a symbol table in `.symtab`. However, unlike the symbol table inside a compiler, the `.symtab` symbol table does not contain entries for local variables.

`.rel.text`: A list of locations in the `.text` section that will need to be modified when the linker combines this object file with others. In general, any instruction that calls an external function or references a global variable will need to be modified. On the other hand, instructions that call local functions do not need to be modified. Note that relocation information is not needed in executable object files, and is usually omitted unless the user explicitly instructs the linker to include it.

`.rel.data`: Relocation information for any global variables that are referenced or defined by the module. In general, any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.

`.debug`: A debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the `-g` option.

`.line`: A mapping between line numbers in the original C source program and machine code instructions in the .text section. It is only present if the compiler driver is invoked with the `-g` option.

`.strtab`: A string table for the symbol tables in the `.symtab` and `.debug` sections, and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

Aside Why is uninitialized data called `.bss`?

The use of the term `.bss` to denote uninitialized data is universal. It was originally an acronym for the "Block Storage Start" instruction from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the `.data` and `.bss` sections is to think of "bss" as an abbreviation for "Better Save Space!"

7.5 Symbols and Symbol Tables

Each relocatable object module, *m*, has a symbol table that contains information about the symbols that are defined and referenced by *m*. In the context of a linker, there are three different kinds of symbols:

- *Global symbols* that are defined by module *m* and that can be referenced by other modules. Global linker symbols correspond to *nonstatic* C functions and global variables that are defined *without* the C `static` attribute.
- Global symbols that are referenced by module *m* but defined by some other module. Such symbols are called *externals* and correspond to C functions and variables that are defined in other modules.
- *Local symbols* that are defined and referenced exclusively by module *m*. Some local linker symbols correspond to C functions and global variables that are defined with the `static` attribute. These symbols are visible anywhere within module *m*, but cannot be referenced by other modules. The sections in an object file and the name of the source file that corresponds to module *m* also get local symbols.

It is important to realize that local linker symbols are not the same as local program variables. The symbol table in `.symtab` does not contain any symbols that correspond to local nonstatic program variables. These are managed at run time on the stack and are not of interest to the linker.

Interestingly, local procedure variables that are defined with the C `static` attribute are not managed on the stack. Instead, the compiler allocates space in `.data` or `.bss` for each definition and creates a local linker symbol in the symbol table with a unique name. For example, suppose a pair of functions in the same module define a static local variable `x`:

```
1   int f()
2   {
3       static int x = 0;
4       return x;
5   }
6
7   int g()
8   {
9       static int x = 1;
10      return x;
11  }
```

In this case, the compiler allocates space for two integers in `.data` and exports a pair of unique local linker symbols to the assembler. For example, it might use `x.1` for the definition in function `f` and `x.2` for the definition in function `g`.

New to C? Hiding variable and function names with `static`

C programmers use the `static` attribute to hide variable and function declarations inside modules, much as you would use *public* and *private* declarations in Java and C++. C source files play the role of modules. Any global variable or function declared with the `static` attribute is private to that module. Similarly, any global variable or function declared without the `static` attribute is public and can be accessed by any other module. It is good programming practice to protect your variables and functions with the `static` attribute wherever possible.

Symbol tables are built by assemblers, using symbols exported by the compiler into the assembly-language `.s` file. An ELF symbol table is contained in the `.symtab` section. It contains an array of entries. Figure 7.4 shows the format of each entry.

The `name` is a byte offset into the string table that points to the null-terminated string name of the symbol. The `value` is the symbol's address. For relocatable modules, the `value` is an offset from the beginning of the section where the object is defined. For executable object files, the `value` is an absolute run-time address. The `size` is the size (in bytes) of the object. The `type` is usually either data or function. The symbol table can also contain entries for the individual sections and for the path name of the original source file. So there are distinct types for these objects as well. The `binding` field indicates whether the symbol is local or global.

Each symbol is associated with some section of the object file, denoted by the `section` field, which is an index into the section header table. There are three special pseudo sections that don't have entries in the section header table: ABS is for symbols that should not be relocated. UNDEF is for undefined symbols, that is, symbols that are referenced in this object module but defined elsewhere. COMMON is for uninitialized data objects that are not yet allocated. For COMMON symbols, the `value` field gives the alignment requirement, and `size` gives the minimum size.

code/link/elfstructs.c

```
1   typedef struct {
2       int name;        /* String table offset */
3       int value;       /* Section offset, or VM address */
4       int size;        /* Object size in bytes */
5       char type:4,     /* Data, func, section, or src file name (4 bits) */
6            binding:4;  /* Local or global (4 bits) */
7       char reserved;   /* Unused */
8       char section;    /* Section header index, ABS, UNDEF, */
9                        /* Or COMMON  */
10  } Elf_Symbol;
```

code/link/elfstructs.c

Figure 7.4 **ELF symbol table entry.** `type` and `binding` are four bits each.

For example, here are the last three entries in the symbol table for `main.o`, as displayed by the GNU READELF tool. The first eight entries, which are not shown, are local symbols that the linker uses internally.

```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     8 OBJECT  GLOBAL  0    3 buf
  9:        0    17 FUNC    GLOBAL  0    1 main
 10:        0     0 NOTYPE  GLOBAL  0  UND swap
```

In this example, we see an entry for the definition of global symbol `buf`, an 8-byte object located at an offset (i.e., `value`) of zero in the `.data` section. This is followed by the definition of the global symbol `main`, a 17-byte function located at an offset of zero in the `.text` section. The last entry comes from the reference for the external symbol `swap`. READELF identifies each section by an integer index. `Ndx=1` denotes the `.text` section, and `Ndx=3` denotes the `.data` section.

Similarly, here are the symbol table entries for `swap.o`:

```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     4 OBJECT  GLOBAL  0    3 bufp0
  9:        0     0 NOTYPE  GLOBAL  0  UND buf
 10:        0    39 FUNC    GLOBAL  0    1 swap
 11:        4     4 OBJECT  GLOBAL  0  COM bufp1
```

First, we see an entry for the definition of the global symbol `bufp0`, which is a 4-byte initialized object starting at offset 0 in `.data`. The next symbol comes from the reference to the external `buf` symbol in the initialization code for `bufp0`. This is followed by the global symbol `swap`, a 39-byte function at an offset of zero in `.text`. The last entry is the global symbol `bufp1`, a 4-byte uninitialized data object (with a 4-byte alignment requirement) that will eventually be allocated as a `.bss` object when this module is linked.

Practice Problem 7.1

This problem concerns the `swap.o` module from Figure 7.1(b). For each symbol that is defined or referenced in `swap.o`, indicate whether or not it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `main.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, or `.bss`) it occupies in that module.

Symbol	`swap.o` `.symtab` entry?	Symbol type	Module where defined	Section
`buf`	_____	_____	_____	_____
`bufp0`	_____	_____	_____	_____
`bufp1`	_____	_____	_____	_____
`swap`	_____	_____	_____	_____
`temp`	_____	_____	_____	_____

7.6 Symbol Resolution

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Symbol resolution is straightforward for references to local symbols that are defined in the same module as the reference. The compiler allows only one definition of each local symbol per module. The compiler also ensures that static local variables, which get local linker symbols, have unique names.

Resolving references to global symbols, however, is trickier. When the compiler encounters a symbol (either a variable or function name) that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle. If the linker is unable to find a definition for the referenced symbol in any of its input modules, it prints an (often cryptic) error message and terminates. For example, if we try to compile and link the following source file on a Linux machine,

```
1   void foo(void);
2
3   int main() {
4       foo();
5       return 0;
6   }
```

then the compiler runs without a hitch, but the linker terminates when it cannot resolve the reference to `foo`:

```
unix> gcc -Wall -O2 -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function ‘main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to ‘foo'
collect2: ld returned 1 exit status
```

Symbol resolution for global symbols is also tricky because the same symbol might be defined by multiple object files. In this case, the linker must either flag an error or somehow choose one of the definitions and discard the rest. The approach adopted by Unix systems involves cooperation between the compiler, assembler, and linker, and can introduce some baffling bugs to the unwary programmer.

Aside Mangling of linker symbols in C++ and Java

Both C++ and Java allow overloaded methods that have the same name in the source code but different parameter lists. So how does the linker tell the difference between these different overloaded functions? Overloaded functions in C++ and Java work because the compiler encodes each unique method and parameter list combination into a unique name for the linker. This encoding process is called *mangling*, and the inverse process *demangling*.

Happily, C++ and Java use compatible mangling schemes. A mangled class name consists of the integer number of characters in the name followed by the original name. For example, the class `Foo` is encoded as `3Foo`. A method is encoded as the original method name, followed by `__`, followed

by the mangled class name, followed by single letter encodings of each argument. For example, `Foo::bar(int, long)` is encoded as `bar__3Fooil`. Similar schemes are used to mangle global variable and template names.

7.6.1 How Linkers Resolve Multiply Defined Global Symbols

At compile time, the compiler exports each global symbol to the assembler as either *strong* or *weak*, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols. For the example program in Figure 7.1, `buf`, `bufp0`, `main`, and `swap` are strong symbols; `bufp1` is a weak symbol.

Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiply defined symbols:

- Rule 1: Multiple strong symbols are not allowed.
- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.
- Rule 3: Given multiple weak symbols, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

```
1  /* foo1.c */
2  int main()
3  {
4      return 0;
5  }
```

```
1  /* bar1.c */
2  int main()
3  {
4      return 0;
5  }
```

In this case, the linker will generate an error message because the strong symbol `main` is defined multiple times (rule 1):

```
unix> gcc foo1.c bar1.c
/tmp/cca015022.o: In function `main':
/tmp/cca015022.o(.text+0x0): multiple definition of `main'
/tmp/cca015021.o(.text+0x0): first defined here
```

Similarly, the linker will generate an error message for the following modules because the strong symbol `x` is defined twice (rule 1):

```
1  /* foo2.c */
2  int x = 15213;
3
4  int main()
5  {
6      return 0;
7  }
```

```
1  /* bar2.c */
2  int x = 15213;
3
4  void f()
5  {
6  }
```

However, if x is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (rule 2):

```
1   /* foo3.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x = 15213;
6
7   int main()
8   {
9       f();
10      printf("x = %d\n", x);
11      return 0;
12  }
```

```
1   /* bar3.c */
2   int x;
3
4   void f()
5   {
6       x = 15212;
7   }
```

At run time, function f changes the value of x from 15213 to 15212, which might come as an unwelcome surprise to the author of function main! Notice that the linker normally gives no indication that it has detected multiple definitions of x:

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./foobar3
x = 15212
```

The same thing can happen if there are two weak definitions of x (rule 3):

```
1   /* foo4.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x;
6
7   int main()
8   {
9       x = 15213;
10      f();
11      printf("x = %d\n", x);
12      return 0;
13  }
```

```
1   /* bar4.c */
2   int x;
3
4   void f()
5   {
6       x = 15212;
7   }
```

The application of rules 2 and 3 can introduce some insidious run-time bugs that are incomprehensible to the unwary programmer, especially if the duplicate symbol definitions have different types. Consider the following example, in which x is defined as an int in one module and a double in another:

```
1   /* foo5.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x = 15213;
6   int y = 15212;
7
8   int main()
9   {
10      f();
11      printf("x = 0x%x y = 0x%x \n",
12             x, y);
13      return 0;
14  }
```

```
1   /* bar5.c */
2   double x;
3
4   void f()
5   {
6       x = -0.0;
7   }
```

On an IA32/Linux machine, `double`s are 8 bytes and `int`s are 4 bytes. Thus, the assignment `x = -0.0` in line 6 of `bar5.c` will overwrite the memory locations for `x` and `y` (lines 5 and 6 in `foo5.c`) with the double-precision floating-point representation of negative zero!

```
linux> gcc -o foobar5 foo5.c bar5.c
linux> ./foobar5
x = 0x0 y = 0x80000000
```

This is a subtle and nasty bug, especially because it occurs silently, with no warning from the compilation system, and because it typically manifests itself much later in the execution of the program, far away from where the error occurred. In a large system with hundreds of modules, a bug of this kind is extremely hard to fix, especially because many programmers are not aware of how linkers work. When in doubt, invoke the linker with a flag such as the GCC `-fno-common` flag, which triggers an error if it encounters multiply defined global symbols.

Practice Problem 7.2

In this problem, let `REF(x.i) --> DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example that follows, use this notation to indicate how the linker would resolve references to the multiply defined symbol in each module. If there is a link-time error (rule 1), write "ERROR." If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN."

A.
```
/* Module 1 */
int main()
{
}
```

```
/* Module 2 */
int main;
int p2()
{
}
```

(a) REF(main.1) --> DEF(_____.___)

(b) REF(main.2) --> DEF(_____.___)

B.
```
/* Module 1 */          /* Module 2 */
void main()             int main=1;
{                       int p2()
}                       {
                        }
```

(a) REF(main.1) --> DEF(_____.___)

(b) REF(main.2) --> DEF(_____.___)

C.
```
/* Module 1 */          /* Module 2 */
int x;                  double x=1.0;
void main()             int p2()
{                       {
}                       }
```

(a) REF(x.1) --> DEF(_____.___)

(b) REF(x.2) --> DEF(_____.___)

7.6.2 Linking with Static Libraries

So far, we have assumed that the linker reads a collection of relocatable object files and links them together into an output executable file. In practice, all compilation systems provide a mechanism for packaging related object modules into a single file called a *static library*, which can then be supplied as input to the linker. When it builds the output executable, the linker copies only the object modules in the library that are referenced by the application program.

Why do systems support the notion of libraries? Consider ANSI C, which defines an extensive collection of standard I/O, string manipulation, and integer math functions such as `atoi`, `printf`, `scanf`, `strcpy`, and `rand`. They are available to every C program in the `libc.a` library. ANSI C also defines an extensive collection of floating-point math functions such as `sin`, `cos`, and `sqrt` in the `libm.a` library.

Consider the different approaches that compiler developers might use to provide these functions to users without the benefit of static libraries. One approach would be to have the compiler recognize calls to the standard functions and to generate the appropriate code directly. Pascal, which provides a small set of standard functions, takes this approach, but it is not feasible for C, because of the large number of standard functions defined by the C standard. It would add significant complexity to the compiler and would require a new compiler version each time a function was added, deleted, or modified. To application programmers, however, this approach would be quite convenient because the standard functions would always be available.

Another approach would be to put all of the standard C functions in a single relocatable object module, say, `libc.o`, that application programmers could link into their executables:

```
unix> gcc main.c /usr/lib/libc.o
```

This approach has the advantage that it would decouple the implementation of the standard functions from the implementation of the compiler, and would still be reasonably convenient for programmers. However, a big disadvantage is that every executable file in a system would now contain a complete copy of the collection of standard functions, which would be extremely wasteful of disk space. (On a typical system, `libc.a` is about 8 MB and `libm.a` is about 1 MB.) Worse, each running program would now contain its own copy of these functions in memory, which would be extremely wasteful of memory. Another big disadvantage is that any change to any standard function, no matter how small, would require the library developer to recompile the entire source file, a time-consuming operation that would complicate the development and maintenance of the standard functions.

We could address some of these problems by creating a separate relocatable file for each standard function and storing them in a well-known directory. However, this approach would require application programmers to explicitly link the appropriate object modules into their executables, a process that would be error prone and time consuming:

```
unix> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

The notion of a static library was developed to resolve the disadvantages of these various approaches. Related functions can be compiled into separate object modules and then packaged in a single static library file. Application programs can then use any of the functions defined in the library by specifying a single file name on the command line. For example, a program that uses functions from the standard C library and the math library could be compiled and linked with a command of the form

```
unix> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

At link time, the linker will only copy the object modules that are referenced by the program, which reduces the size of the executable on disk and in memory. On the other hand, the application programmer only needs to include the names of a few library files. (In fact, C compiler drivers always pass `libc.a` to the linker, so the reference to `libc.a` mentioned previously is unnecessary.)

On Unix systems, static libraries are stored on disk in a particular file format known as an *archive*. An archive is a collection of concatenated relocatable object files, with a header that describes the size and location of each member object file. Archive filenames are denoted with the `.a` suffix. To make our discussion of libraries concrete, suppose that we want to provide the vector routines in Figure 7.5 in a static library called `libvector.a`.

(a) `addvec.o`

code/link/addvec.c

```
1  void addvec(int *x, int *y,
2              int *z, int n)
3  {
4      int i;
5
6      for (i = 0; i < n; i++)
7          z[i] = x[i] + y[i];
8  }
```

code/link/addvec.c

(b) `multvec.o`

code/link/multvec.c

```
1  void multvec(int *x, int *y,
2               int *z, int n)
3  {
4      int i;
5
6      for (i = 0; i < n; i++)
7          z[i] = x[i] * y[i];
8  }
```

code/link/multvec.c

Figure 7.5 **Member object files in** `libvector.a`.

To create the library, we would use the AR tool as follows:

```
unix> gcc -c addvec.c multvec.c
unix> ar rcs libvector.a addvec.o multvec.o
```

To use the library, we might write an application such as `main2.c` in Figure 7.6, which invokes the `addvec` library routine. (The include (header) file `vector.h` defines the function prototypes for the routines in `libvector.a`.)

code/link/main2.c

```
1  /* main2.c */
2  #include <stdio.h>
3  #include "vector.h"
4
5  int x[2] = {1, 2};
6  int y[2] = {3, 4};
7  int z[2];
8
9  int main()
10 {
11     addvec(x, y, z, 2);
12     printf("z = [%d %d]\n", z[0], z[1]);
13     return 0;
14 }
```

code/link/main2.c

Figure 7.6 **Example program 2:** This program calls member functions in the static `libvector.a` library.

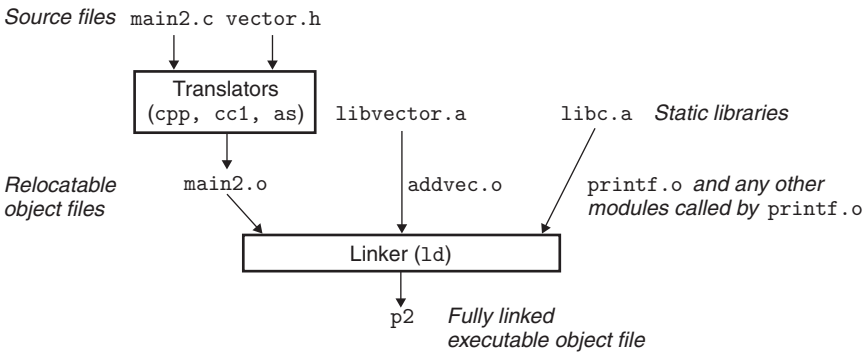


Figure 7.7 **Linking with static libraries.**

To build the executable, we would compile and link the input files `main.o` and `libvector.a`:

```
unix> gcc -O2 -c main2.c
unix> gcc -static -o p2 main2.o ./libvector.a
```

Figure 7.7 summarizes the activity of the linker. The `-static` argument tells the compiler driver that the linker should build a fully linked executable object file that can be loaded into memory and run without any further linking at load time. When the linker runs, it determines that the `addvec` symbol defined by `addvec.o` is referenced by `main.o`, so it copies `addvec.o` into the executable. Since the program doesn't reference any symbols defined by `multvec.o`, the linker does *not* copy this module into the executable. The linker also copies the `printf.o` module from `libc.a`, along with a number of other modules from the C run-time system.

7.6.3 How Linkers Use Static Libraries to Resolve References

While static libraries are useful and essential tools, they are also a source of confusion to programmers because of the way the Unix linker uses them to resolve external references. During the symbol resolution phase, the linker scans the relocatable object files and archives left to right in the same sequential order that they appear on the compiler driver's command line. (The driver automatically translates any `.c` files on the command line into `.o` files.) During this scan, the linker maintains a set E of relocatable object files that will be merged to form the executable, a set U of unresolved symbols (i.e., symbols referred to, but not yet defined), and a set D of symbols that have been defined in previous input files. Initially, E, U, and D are empty.

- For each input file f on the command line, the linker determines if f is an object file or an archive. If f is an object file, the linker adds f to E, updates U and D to reflect the symbol definitions and references in f, and proceeds to the next input file.

- If f is an archive, the linker attempts to match the unresolved symbols in U against the symbols defined by the members of the archive. If some archive member, m, defines a symbol that resolves a reference in U, then m is added to E, and the linker updates U and D to reflect the symbol definitions and references in m. This process iterates over the member object files in the archive until a fixed point is reached where U and D no longer change. At this point, any member object files not contained in E are simply discarded and the linker proceeds to the next input file.
- If U is nonempty when the linker finishes scanning the input files on the command line, it prints an error and terminates. Otherwise, it merges and relocates the object files in E to build the output executable file.

Unfortunately, this algorithm can result in some baffling link-time errors because the ordering of libraries and object files on the command line is significant. If the library that defines a symbol appears on the command line before the object file that references that symbol, then the reference will not be resolved and linking will fail. For example, consider the following:

```
unix> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function ‘main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to ‘addvec'
```

What happened? When `libvector.a` is processed, U is empty, so no member object files from `libvector.a` are added to E. Thus, the reference to `addvec` is never resolved and the linker emits an error message and terminates.

The general rule for libraries is to place them at the end of the command line. If the members of the different libraries are independent, in that no member references a symbol defined by another member, then the libraries can be placed at the end of the command line in any order.

If, on the other hand, the libraries are not independent, then they must be ordered so that for each symbol s that is referenced externally by a member of an archive, at least one definition of s follows a reference to s on the command line. For example, suppose `foo.c` calls functions in `libx.a` and `libz.a` that call functions in `liby.a`. Then `libx.a` and `libz.a` must precede `liby.a` on the command line:

```
unix> gcc foo.c libx.a libz.a liby.a
```

Libraries can be repeated on the command line if necessary to satisfy the dependence requirements. For example, suppose `foo.c` calls a function in `libx.a` that calls a function in `liby.a` that calls a function in `libx.a`. Then `libx.a` must be repeated on the command line:

```
unix> gcc foo.c libx.a liby.a libx.a
```

Alternatively, we could combine `libx.a` and `liby.a` into a single archive.

Practice Problem 7.3

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references.

A. `p.o` → `libx.a`.

B. `p.o` → `libx.a` → `liby.a`.

C. `p.o` → `libx.a` → `liby.a` **and** `liby.a` → `libx.a` →`p.o`.

7.7 Relocation

Once the linker has completed the symbol resolution step, it has associated each symbol reference in the code with exactly one symbol definition (i.e., a symbol table entry in one of its input object modules). At this point, the linker knows the exact sizes of the code and data sections in its input object modules. It is now ready to begin the relocation step, where it merges the input modules and assigns run-time addresses to each symbol. Relocation consists of two steps:

- *Relocating sections and symbol definitions.* In this step, the linker merges all sections of the same type into a new aggregate section of the same type. For example, the `.data` sections from the input modules are all merged into one section that will become the `.data` section for the output executable object file. The linker then assigns run-time memory addresses to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules. When this step is complete, every instruction and global variable in the program has a unique run-time memory address.
- *Relocating symbol references within sections.* In this step, the linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses. To perform this step, the linker relies on data structures in the relocatable object modules known as relocation entries, which we describe next.

7.7.1 Relocation Entries

When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate

code/link/elfstructs.c

```
1  typedef struct {
2      int offset;     /* Offset of the reference to relocate */
3      int symbol:24,  /* Symbol the reference should point to */
4          type:8;     /* Relocation type */
5  } Elf32_Rel;
```

code/link/elfstructs.c

Figure 7.8 **ELF relocation entry.** Each entry identifies a reference that must be relocated.

location is unknown, it generates a *relocation entry* that tells the linker how to modify the reference when it merges the object file into an executable. Relocation entries for code are placed in `.rel.text`. Relocation entries for initialized data are placed in `.rel.data`.

Figure 7.8 shows the format of an ELF relocation entry. The `offset` is the section offset of the reference that will need to be modified. The `symbol` identifies the symbol that the modified reference should point to. The `type` tells the linker how to modify the new reference.

ELF defines 11 different relocation types, some quite arcane. We are concerned with only the two most basic relocation types:

- `R_386_PC32`: Relocate a reference that uses a 32-bit PC-relative address. Recall from Section 3.6.3 that a PC-relative address is an offset from the current run-time value of the program counter (PC). When the CPU executes an instruction using PC-relative addressing, it forms the *effective address* (e.g., the target of the `call` instruction) by adding the 32-bit value encoded in the instruction to the current run-time value of the PC, which is always the address of the next instruction in memory.
- `R_386_32`: Relocate a reference that uses a 32-bit absolute address. With absolute addressing, the CPU directly uses the 32-bit value encoded in the instruction as the effective address, without further modifications.

7.7.2 Relocating Symbol References

Figure 7.9 shows the pseudo code for the linker's relocation algorithm. Lines 1 and 2 iterate over each section `s` and each relocation entry `r` associated with each section. For concreteness, assume that each section `s` is an array of bytes and that each relocation entry `r` is a `struct` of type `Elf32_Rel`, as defined in Figure 7.8. Also, assume that when the algorithm runs, the linker has already chosen run-time addresses for each section (denoted `ADDR(s)`) and each symbol (denoted `ADDR(r.symbol)`). Line 3 computes the address in the `s` array of the 4-byte reference that needs to be relocated. If this reference uses PC-relative

```
1   foreach section s {
2       foreach relocation entry r {
3           refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5           /* Relocate a PC-relative reference */
6           if (r.type == R_386_PC32) {
7               refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8               *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9           }
10
11          /* Relocate an absolute reference */
12          if (r.type == R_386_32)
13              *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14      }
15  }
```

Figure 7.9 **Relocation algorithm.**

addressing, then it is relocated by lines 5–9. If the reference uses absolute addressing, then it is relocated by lines 11–13.

Relocating PC-Relative References

Recall from our running example in Figure 7.1(a) that the `main` routine in the `.text` section of `main.o` calls the `swap` routine, which is defined in `swap.o`. Here is the disassembled listing for the `call` instruction, as generated by the GNU OBJDUMP tool:

```
6:   e8 fc ff ff ff          call   7 <main+0x7>   swap();
                             7: R_386_PC32 swap    relocation entry
```

From this listing, we see that the `call` instruction begins at section offset `0x6` and consists of the 1-byte opcode `0xe8`, followed by the 32-bit reference `0xfffffffc` (−4 decimal), which is stored in little-endian byte order. We also see a relocation entry for this reference displayed on the following line. (Recall that relocation entries and instructions are actually stored in different sections of the object file. The OBJDUMP tool displays them together for convenience.) The relocation entry `r` consists of three fields:

```
r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

These fields tell the linker to modify the 32-bit PC-relative reference starting at offset `0x7` so that it will point to the `swap` routine at run time. Now, suppose that the linker has determined that

```
ADDR(s) = ADDR(.text) = 0x80483b4
```

and

```
ADDR(r.symbol) = ADDR(swap) = 0x80483c8
```

Using the algorithm in Figure 7.9, the linker first computes the run-time address of the reference (line 7):

```
refaddr = ADDR(s)   + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```

It then updates the reference from its current value (−4) to 0x9 so that it will point to the swap routine at run time (line 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
        = (unsigned) (0x80483c8      + (-4)    - 0x80483bb)
        = (unsigned) (0x9)
```

In the resulting executable object file, the call instruction has the following relocated form:

```
 80483ba:  e8 09 00 00 00          call   80483c8 <swap>          swap();
```

At run time, the call instruction will be stored at address 0x80483ba. When the CPU executes the call instruction, the PC has a value of 0x80483bf, which is the address of the instruction immediately following the call instruction. To execute the instruction, the CPU performs the following steps:

```
1. push PC onto stack
2. PC <- PC + 0x9 = 0x80483bf + 0x9 = 0x80483c8
```

Thus, the next instruction to execute is the first instruction of the swap routine, which of course is what we want!

You may wonder why the assembler created the reference in the call instruction with an initial value of −4. The assembler uses this value as a bias to account for the fact that the PC always points to the instruction following the current instruction. On a different machine with different instruction sizes and encodings, the assembler for that machine would use a different bias. This is a powerful trick that allows the linker to blindly relocate references, blissfully unaware of the instruction encodings for a particular machine.

Relocating Absolute References

Recall that in our example program in Figure 7.1, the swap.o module initializes the global pointer bufp0 to the address of the first element of the global buf array:

```
int *bufp0 = &buf[0];
```

Since bufp0 is an initialized data object, it will be stored in the .data section of the swap.o relocatable object module. Since it is initialized to the address of a global array, it will need to be relocated. Here is the disassembled listing of the .data section from swap.o:

```
00000000 <bufp0>:
   0:   00 00 00 00                                    int *bufp0 = &buf[0];
                        0: R_386_32 buf                Relocation entry
```

We see that the .data section contains a single 32-bit reference, the bufp0 pointer, which has a value of 0x0. The relocation entry tells the linker that this is a 32-bit absolute reference, beginning at offset 0, which must be relocated so that it points to the symbol buf. Now, suppose that the linker has determined that

```
ADDR(r.symbol) = ADDR(buf) = 0x8049454
```

The linker updates the reference using line 13 of the algorithm in Figure 7.9:

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr)
        = (unsigned) (0x8049454      + 0)
        = (unsigned) (0x8049454)
```

In the resulting executable object file, the reference has the following relocated form:

```
0804945c <bufp0>:
 804945c:  54 94 04 08                          Relocated!
```

In words, the linker has decided that at run time the variable bufp0 will be located at memory address 0x804945c and will be initialized to 0x8049454, which is the run-time address of the buf array.

The .text section in the swap.o module contains five absolute references that are relocated in a similar way (see Problem 7.12). Figure 7.10 shows the relocated .text and .data sections in the final executable object file.

Practice Problem 7.4

This problem concerns the relocated program in Figure 7.10.

A. What is the hex address of the relocated reference to swap in line 5?

B. What is the hex value of the relocated reference to swap in line 5?

C. Suppose the linker had decided for some reason to locate the .text section at 0x80483b8 instead of 0x80483b4. What would the hex value of the relocated reference in line 5 be in this case?

(a) Relocated `.text` section

code/link/p-exe.d

```
080483b4 <main>:
 80483b4:  55                      push   %ebp
 80483b5:  89 e5                   mov    %esp,%ebp
 80483b7:  83 ec 08                sub    $0x8,%esp
 80483ba:  e8 09 00 00 00          call   80483c8 <swap>          swap();
 80483bf:  31 c0                   xor    %eax,%eax
 80483c1:  89 ec                   mov    %ebp,%esp
 80483c3:  5d                      pop    %ebp
 80483c4:  c3                      ret
 80483c5:  90                      nop
 80483c6:  90                      nop
 80483c7:  90                      nop

080483c8 <swap>:
 80483c8:  55                      push   %ebp
 80483c9:  8b 15 5c 94 04 08       mov    0x804945c,%edx          Get *bufp0
 80483cf:  a1 58 94 04 08          mov    0x8049458,%eax          Get buf[1]
 80483d4:  89 e5                   mov    %esp,%ebp
 80483d6:  c7 05 48 95 04 08 58    movl   $0x8049458,0x8049548    bufp1 = &buf[1]
 80483dd:  94 04 08
 80483e0:  89 ec                   mov    %ebp,%esp
 80483e2:  8b 0a                   mov    (%edx),%ecx
 80483e4:  89 02                   mov    %eax,(%edx)
 80483e6:  a1 48 95 04 08          mov    0x8049548,%eax          Get *bufp1
 80483eb:  89 08                   mov    %ecx,(%eax)
 80483ed:  5d                      pop    %ebp
 80483ee:  c3                      ret
```

code/link/p-exe.d

(b) Relocated `.data` section

code/link/pdata-exe.d

```
08049454 <buf>:
 8049454:  01 00 00 00 02 00 00 00

0804945c <bufp0>:
 804945c:  54 94 04 08                    Relocated!
```

code/link/pdata-exe.d

Figure 7.10 **Relocated `.text` and `.data` sections for executable file `p`.** The original C code is in Figure 7.1.

7.8 Executable Object Files

We have seen how the linker merges multiple object modules into a single executable object file. Our C program, which began life as a collection of ASCII text files, has been transformed into a single binary file that contains all of the information needed to load the program into memory and run it. Figure 7.11 summarizes the kinds of information in a typical ELF executable file.

The format of an executable object file is similar to that of a relocatable object file. The ELF header describes the overall format of the file. It also includes the program's *entry point*, which is the address of the first instruction to execute when the program runs. The `.text`, `.rodata`, and `.data` sections are similar to those in a relocatable object file, except that these sections have been relocated to their eventual run-time memory addresses. The `.init` section defines a small function, called `_init`, that will be called by the program's initialization code. Since the executable is *fully linked* (relocated), it needs no `.rel` sections.

ELF executables are designed to be easy to load into memory, with contiguous chunks of the executable file mapped to contiguous memory segments. This mapping is described by the *segment header table*. Figure 7.12 shows the segment header table for our example executable `p`, as displayed by OBJDUMP.

From the segment header table, we see that two memory segments will be initialized with the contents of the executable object file. Lines 1 and 2 tell us that the first segment (the *code segment*) is aligned to a 4 KB (2^{12}) boundary, has read/execute permissions, starts at memory address `0x08048000`, has a total memory size of `0x448` bytes, and is initialized with the first `0x448` bytes of the executable object file, which includes the ELF header, the segment header table, and the `.init`, `.text`, and `.rodata` sections.

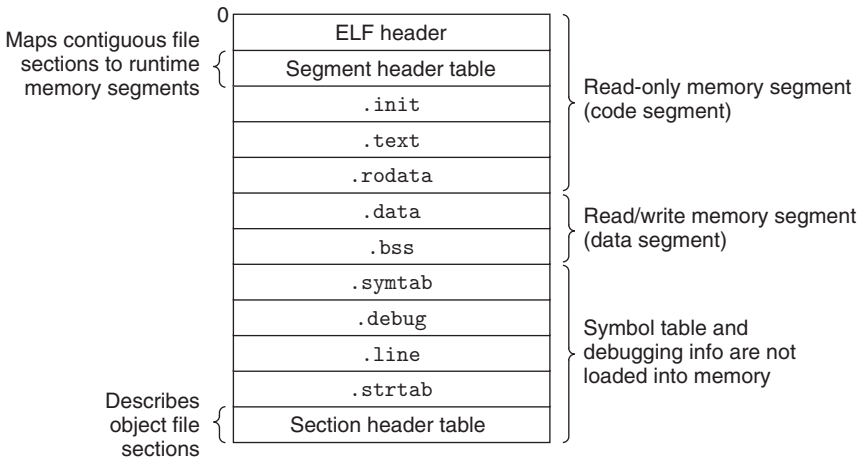


Figure 7.11 **Typical ELF executable object file.**

code/link/p-exe.d

```
   Read-only code segment
1    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
2         filesz 0x00000448 memsz 0x00000448 flags r-x

   Read/write data segment
3    LOAD off    0x00000448 vaddr 0x08049448 paddr 0x08049448 align 2**12
4         filesz 0x000000e8 memsz 0x00000104 flags rw-
```

code/link/p-exe.d

Figure 7.12 **Segment header table for the example executable** p. Legend: `off`: file offset, `vaddr/paddr`: virtual/physical address, `align`: segment alignment, `filesz`: segment size in the object file, `memsz`: segment size in memory, `flags`: run-time permissions.

Lines 3 and 4 tell us that the second segment (the *data segment*) is aligned to a 4 KB boundary, has read/write permissions, starts at memory address `0x08049448`, has a total memory size of `0x104` bytes, and is initialized with the `0xe8` bytes starting at file offset `0x448`, which in this case is the beginning of the `.data` section. The remaining bytes in the segment correspond to `.bss` data that will be initialized to zero at run time.

7.9 Loading Executable Object Files

To run an executable object file `p`, we can type its name to the Unix shell's command line:

```
unix> ./p
```

Since `p` does not correspond to a built-in shell command, the shell assumes that `p` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the `loader`. Any Unix program can invoke the loader by calling the `execve` function, which we will describe in detail in Section 8.4.5. The loader copies the code and data in the executable object file from disk into memory, and then runs the program by jumping to its first instruction, or *entry point*. This process of copying the program into memory and then running it is known as *loading*.

Every Unix program has a run-time memory image similar to the one in Figure 7.13. On 32-bit Linux systems, the code segment starts at address `0x08048000`. The data segment follows at the next 4 KB aligned address. The run-time *heap* follows on the first 4 KB aligned address past the read/write segment and grows up via calls to the `malloc` library. (We will describe `malloc` and the heap in detail in Section 9.9.) There is also a segment that is reserved for shared libraries. The user stack always starts at the largest legal user address and grows down (toward lower memory addresses). The segment starting above the stack is reserved for

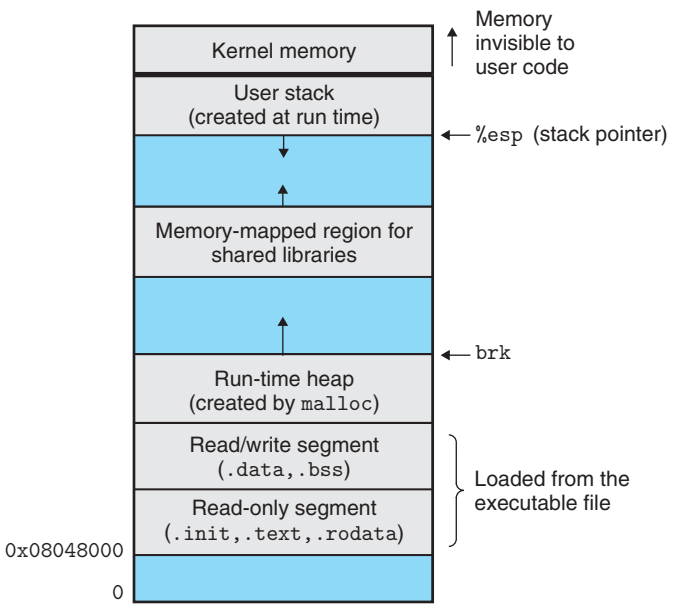


Figure 7.13
Linux run-time memory image.

the code and data in the memory-resident part of the operating system known as the *kernel*.

When the loader runs, it creates the memory image shown in Figure 7.13. Guided by the segment header table in the executable, it copies chunks of the executable into the code and data segments. Next, the loader jumps to the program's entry point, which is always the address of the `_start` symbol. The *startup code* at the `_start` address is defined in the object file `crt1.o` and is the same for all C programs. Figure 7.14 shows the specific sequence of calls in the startup code. After calling initialization routines from the `.text` and `.init` sections, the startup code calls the `atexit` routine, which appends a list of routines that should be called when the application terminates normally. The `exit` function runs the functions registered by `atexit`, and then returns control to the operating system

```
1   0x080480c0 <_start>:          /* Entry point in .text */
2       call __libc_init_first    /* Startup code in .text */
3       call _init                /* Startup code in .init */
4       call atexit               /* Startup code in .text */
5       call main                 /* Application main routine */
6       call _exit                /* Returns control to OS */
7   /* Control never reaches here */
```

Figure 7.14 **Pseudo-code for the `crt1.o` startup routine in every C program.** Note: The code that pushes the arguments for each function is not shown.

by calling `_exit`. Next, the startup code calls the application's `main` routine, which begins executing our C code. After the application returns, the startup code calls the `_exit` routine, which returns control to the operating system.

Aside How do loaders really work?

Our description of loading is conceptually correct, but intentionally not entirely accurate. To understand how loading really works, you must understand the concepts of *processes*, *virtual memory*, and *memory mapping*, which we haven't discussed yet. As we encounter these concepts later in Chapters 8 and 9, we will revisit loading and gradually reveal the mystery to you.

For the impatient reader, here is a preview of how loading really works: Each program in a Unix system runs in the context of a process with its own virtual address space. When the shell runs a program, the parent shell process forks a child process that is a duplicate of the parent. The child process invokes the loader via the `execve` system call. The loader deletes the child's existing virtual memory segments, and creates a new set of code, data, heap, and stack segments. The new stack and heap segments are initialized to zero. The new code and data segments are initialized to the contents of the executable file by mapping pages in the virtual address space to page-sized chunks of the executable file. Finally, the loader jumps to the `_start` address, which eventually calls the application's `main` routine. Aside from some header information, there is no copying of data from disk to memory during loading. The copying is deferred until the CPU references a mapped virtual page, at which point the operating system automatically transfers the page from disk to memory using its paging mechanism.

Practice Problem 7.5

A. Why does every C program need a routine called `main`?

B. Have you ever wondered why a C `main` routine can end with a call to `exit`, a `return` statement, or neither, and yet the program still terminates properly? Explain.

7.10 Dynamic Linking with Shared Libraries

The static libraries that we studied in Section 7.6.2 address many of the issues associated with making large collections of related functions available to application programs. However, static libraries still have some significant disadvantages. Static libraries, like all software, need to be maintained and updated periodically. If application programmers want to use the most recent version of a library, they must somehow become aware that the library has changed, and then explicitly relink their programs against the updated library.

Another issue is that almost every C program uses standard I/O functions such as `printf` and `scanf`. At run time, the code for these functions is duplicated in the text segment of each running process. On a typical system that is running 50–100

processes, this can be a significant waste of scarce memory system resources. (An interesting property of memory is that it is *always* a scarce resource, regardless of how much there is in a system. Disk space and kitchen trash cans share this same property.)

Shared libraries are modern innovations that address the disadvantages of static libraries. A shared library is an object module that, *at run time*, can be loaded at an arbitrary memory address and linked with a program in memory. This process is known as *dynamic linking* and is performed by a program called a *dynamic linker*.

Shared libraries are also referred to as *shared objects*, and on Unix systems are typically denoted by the `.so` suffix. Microsoft operating systems make heavy use of shared libraries, which they refer to as DLLs (dynamic link libraries).

Shared libraries are "shared" in two different ways. First, in any given file system, there is exactly one `.so` file for a particular library. The code and data in this `.so` file are shared by all of the executable object files that reference the library, as opposed to the contents of static libraries, which are copied and embedded in the executables that reference them. Second, a single copy of the `.text` section of a shared library in memory can be shared by different running processes. We will explore this in more detail when we study virtual memory in Chapter 9.

Figure 7.15 summarizes the dynamic linking process for the example program in Figure 7.6. To build a shared library `libvector.so` of our example vector arithmetic routines in Figure 7.5, we would invoke the compiler driver with the following special directive to the linker:

```
unix> gcc -shared -fPIC -o libvector.so addvec.c multvec.c
```

The `-fPIC` flag directs the compiler to generate position-independent code (more on this in the next section). The `-shared` flag directs the linker to create a shared object file.

Once we have created the library, we would then link it into our example program in Figure 7.6:

```
unix> gcc -o p2 main2.c ./libvector.so
```

This creates an executable object file `p2` in a form that can be linked with `libvector.so` at run time. The basic idea is to do some of the linking statically when the executable file is created, and then complete the linking process dynamically when the program is loaded.

It is important to realize that none of the code or data sections from `libvector.so` are actually copied into the executable `p2` at this point. Instead, the linker copies some relocation and symbol table information that will allow references to code and data in `libvector.so` to be resolved at run time.

When the loader loads and runs the executable `p2`, it loads the partially linked executable `p2`, using the techniques discussed in Section 7.9. Next, it notices that `p2`

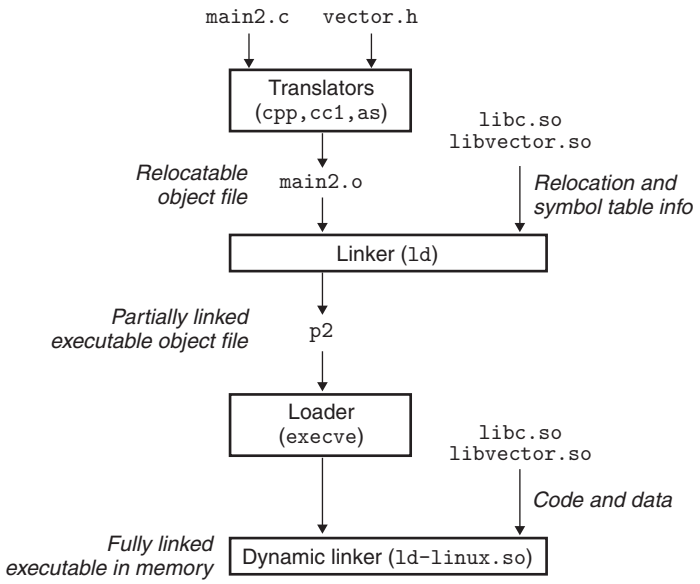


Figure 7.15
Dynamic linking with shared libraries.

contains a `.interp` section, which contains the path name of the dynamic linker, which is itself a shared object (e.g., LD-LINUX.SO on Linux systems). Instead of passing control to the application, as it would normally do, the loader loads and runs the dynamic linker.

The dynamic linker then finishes the linking task by performing the following relocations:

- Relocating the text and data of `libc.so` into some memory segment.
- Relocating the text and data of `libvector.so` into another memory segment.
- Relocating any references in `p2` to symbols defined by `libc.so` and `libvector.so`.

Finally, the dynamic linker passes control to the application. From this point on, the locations of the shared libraries are fixed and do not change during execution of the program.

7.11 Loading and Linking Shared Libraries from Applications

Up to this point, we have discussed the scenario in which the dynamic linker loads and links shared libraries when an application is loaded, just before it executes. However, it is also possible for an application to request the dynamic linker to load and link arbitrary shared libraries while the application is running, without having to link in the applications against those libraries at compile time.

Dynamic linking is a powerful and useful technique. Here are some examples in the real world:

- *Distributing software.* Developers of Microsoft Windows applications frequently use shared libraries to distribute software updates. They generate a new copy of a shared library, which users can then download and use as a replacement for the current version. The next time they run their application, it will automatically link and load the new shared library.
- *Building high-performance Web servers.* Many Web servers generate *dynamic content*, such as personalized Web pages, account balances, and banner ads. Early Web servers generated dynamic content by using `fork` and `execve` to create a child process and run a "CGI program" in the context of the child. However, modern high-performance Web servers can generate dynamic content using a more efficient and sophisticated approach based on dynamic linking.

  The idea is to package each function that generates dynamic content in a shared library. When a request arrives from a Web browser, the server dynamically loads and links the appropriate function and then calls it directly, as opposed to using `fork` and `execve` to run the function in the context of a child process. The function remains cached in the server's address space, so subsequent requests can be handled at the cost of a simple function call. This can have a significant impact on the throughput of a busy site. Further, existing functions can be updated and new functions can be added at run time, without stopping the server.

Linux systems provide a simple interface to the dynamic linker that allows application programs to load and link shared libraries at run time.

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
                                    Returns: ptr to handle if OK, NULL on error
```

The `dlopen` function loads and links the shared library `filename`. The external symbols in `filename` are resolved using libraries previously opened with the `RTLD_GLOBAL` flag. If the current executable was compiled with the `-rdynamic` flag, then its global symbols are also available for symbol resolution. The `flag` argument must include either `RTLD_NOW`, which tells the linker to resolve references to external symbols immediately, or the `RTLD_LAZY` flag, which instructs the linker to defer symbol resolution until code from the library is executed. Either of these values can be or'd with the `RTLD_GLOBAL` flag.

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
                                    Returns: ptr to symbol if OK, NULL on error
```

The `dlsym` function takes a `handle` to a previously opened shared library and a `symbol` name, and returns the address of the symbol, if it exists, or NULL otherwise.

```
#include <dlfcn.h>

int dlclose (void *handle);
                                                Returns: 0 if OK, −1 on error
```

The `dlclose` function unloads the shared library if no other shared libraries are still using it.

```
#include <dlfcn.h>

const char *dlerror(void);
                                Returns: error msg if previous call to dlopen, dlsym,
                                    or dlclose failed, NULL if previous call was OK
```

The `dlerror` function returns a string describing the most recent error that occurred as a result of calling `dlopen`, `dlsym`, or `dlclose`, or NULL if no error occurred.

Figure 7.16 shows how we would use this interface to dynamically link our `libvector.so` shared library (Figure 7.5), and then invoke its `addvec` routine. To compile the program, we would invoke GCC in the following way:

```
unix> gcc -rdynamic -O2 -o p3 dll.c -ldl
```

Aside Shared libraries and the Java Native Interface

Java defines a standard calling convention called *Java Native Interface* (JNI) that allows "native" C and C++ functions to be called from Java programs. The basic idea of JNI is to compile the native C function, say, `foo`, into a shared library, say `foo.so`. When a running Java program attempts to invoke function `foo`, the Java interpreter uses the `dlopen` interface (or something like it) to dynamically link and load `foo.so`, and then call `foo`.

code/link/dll.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

code/link/dll.c

Figure 7.16 **An application program that dynamically loads and links the shared library** `libvector.so`.

7.12 Position-Independent Code (PIC)

A key purpose of shared libraries is to allow multiple running processes to share the same library code in memory and thus save precious memory resources. So how can multiple processes share a single copy of a program? One approach would be to assign *a priori* a dedicated chunk of the address space to each shared library, and then require the loader to always load the shared library at that address. While straightforward, this approach creates some serious problems. It would be an inefficient use of the address space because portions of the space would be allocated even if a process didn't use the library. Second, it would be difficult to manage. We would have to ensure that none of the chunks overlapped. Every time a library were modified, we would have to make sure that it still fit in its assigned chunk. If not, then we would have to find a new chunk. And if we created a new library, we would have to find room for it. Over time, given the hundreds of libraries and versions of libraries in a system, it would be difficult to keep the address space from fragmenting into lots of small unused but unusable holes. Even worse, the assignment of libraries to memory would be different for each system, thus creating even more management headaches.

A better approach is to compile library code so that it can be loaded and executed at any address without being modified by the linker. Such code is known as *position-independent code* (PIC). Users direct GNU compilation systems to generate PIC code with the `-fPIC` option to GCC.

On IA32 systems, calls to procedures in the same object module require no special treatment, since the references are PC-relative, with known offsets, and thus are already PIC (see Problem 7.4). However, calls to externally defined procedures and references to global variables are not normally PIC, since they require relocation at link time.

PIC Data References

Compilers generate PIC references to global variables by exploiting the following interesting fact: No matter where we load an object module (including shared object modules) in memory, the data segment is always allocated immediately after the code segment. Thus, the *distance* between any instruction in the code segment and any variable in the data segment is a run-time constant, independent of the absolute memory locations of the code and data segments.

To exploit this fact, the compiler creates a table called the *global offset table* (GOT) at the beginning of the data segment. The GOT contains an entry for each global data object that is referenced by the object module. The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each entry in the GOT so that it contains the appropriate absolute address. Each object module that references global data has its own GOT.

At run time, each global variable is referenced indirectly through the GOT using code of the form

```
        call L1
L1:     popl %ebx              ebx contains the current PC
        addl $VAROFF, %ebx     ebx points to the GOT entry for var
        movl (%ebx), %eax      reference indirect through the GOT
        movl (%eax), %eax
```

In this fascinating piece of code, the call to `L1` pushes the return address (which happens to be the address of the `popl` instruction) on the stack. The `popl` instruction then pops this address into `%ebx`. The net effect of these two instructions is to move the value of the PC into register `%ebx`.

The `addl` instruction adds a constant offset to `%ebx` so that it points to the appropriate entry in the GOT, which contains the absolute address of the data item. At this point, the global variable can be referenced indirectly through the GOT entry contained in `%ebx`. In this example, the two `movl` instructions load the contents of the global variable (indirectly through the GOT) into register `%eax`.

PIC code has performance disadvantages. Each global variable reference now requires five instructions instead of one, with an additional memory reference to the GOT. Also, PIC code uses an additional register to hold the address of the GOT entry. On machines with large register files, this is not a major issue. On register-starved IA32 systems, however, losing even one register can trigger spilling of the registers onto the stack.

PIC Function Calls

It would certainly be possible for PIC code to use the same approach for resolving external procedure calls:

```
        call L1
L1:     popl %ebx              ebx contains the current PC
        addl $PROCOFF, %ebx    ebx points to GOT entry for proc
        call *(%ebx)           call indirect through the GOT
```

However, this approach would require three additional instructions for each run-time procedure call. Instead, ELF compilation systems use an interesting technique, called *lazy binding*, that defers the binding of procedure addresses until the first time the procedure is called. There is a nontrivial run-time overhead the first time the procedure is called, but each call thereafter only costs a single instruction and a memory reference for the indirection.

Lazy binding is implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the *procedure linkage table* (PLT). If an object module calls any functions that are defined in shared libraries, then it has its own GOT and PLT. The GOT is part of the `.data` section. The PLT is part of the `.text` section.

Figure 7.17 shows the format of the GOT for the example program `main2.o` from Figure 7.6. The first three GOT entries are special: GOT[0] contains the address of the `.dynamic` segment, which contains information that the dynamic linker uses to bind procedure addresses, such as the location of the symbol table

Address	Entry	Contents	Description
08049674	GOT[0]	0804969c	address of `.dynamic` section
08049678	GOT[1]	4000a9f8	identifying info for the linker
0804967c	GOT[2]	4000596f	entry point in dynamic linker
08049680	GOT[3]	0804845a	address of `pushl` in PLT[1] (`printf`)
08049684	GOT[4]	0804846a	address of `pushl` in PLT[2] (`addvec`)

Figure 7.17 **The global offset table (GOT) for executable** p2. The original code is in Figures 7.5 and 7.6.

and relocation information. GOT[1] contains some information that defines this module. GOT[2] contains an entry point into the lazy binding code of the dynamic linker.

Each procedure that is defined in a shared object and called by `main2.o` gets an entry in the GOT, starting with entry GOT[3]. For the example program, we have shown the GOT entries for `printf`, which is defined in `libc.so`, and `addvec`, which is defined in `libvector.so`.

Figure 7.18 shows the PLT for our example program p2. The PLT is an array of 16-byte entries. The first entry, PLT[0], is a special entry that jumps into the dynamic linker. Each called procedure has an entry in the PLT, starting at PLT[1]. In the figure, PLT[1] corresponds to `printf` and PLT[2] corresponds to `addvec`.

```
PLT[0]
08048444:  ff 35 78 96 04 08  pushl  0x8049678    push &GOT[1]
 804844a:  ff 25 7c 96 04 08  jmp  *0x804967c     jmp to *GOT[2](linker)
 8048450:  00 00                                  padding
 8048452:  00 00                                  padding

PLT[1] <printf>
 8048454:  ff 25 80 96 04 08  jmp    *0x8049680  jmp to *GOT[3]
 804845a:  68 00 00 00 00     pushl  $0x0        ID for printf
 804845f:  e9 e0 ff ff ff     jmp    8048444     jmp to PLT[0]

PLT[2] <addvec>
 8048464:  ff 25 84 96 04 08  jmp    *0x8049684  jump to *GOT[4]
 804846a:  68 08 00 00 00     pushl  $0x8        ID for addvec
 804846f:  e9 d0 ff ff ff     jmp    8048444     jmp to PLT[0]

<other PLT entries>
```

Figure 7.18 **The procedure linkage table (PLT) for executable** p2. The original code is in Figures 7.5 and 7.6.

Initially, after the program has been dynamically linked and begins executing, procedures `printf` and `addvec` are bound to the first instruction in their respective PLT entries. For example, the call to `addvec` has the form

```
80485bb:   e8 a4 fe ff ff   call 8048464 <addvec>
```

When `addvec` is called the first time, control passes to the first instruction in PLT[2], which does an indirect jump through GOT[4]. Initially, each GOT entry contains the address of the `pushl` entry in the corresponding PLT entry. So the indirect jump in the PLT simply transfers control back to the next instruction in PLT[2]. This instruction pushes an ID for the `addvec` symbol onto the stack. The last instruction jumps to PLT[0], which pushes another word of identifying information on the stack from GOT[1], and then jumps into the dynamic linker indirectly through GOT[2]. The dynamic linker uses the two stack entries to determine the location of `addvec`, overwrites GOT[4] with this address, and passes control to `addvec`.

The next time `addvec` is called in the program, control passes to PLT[2] as before. However, this time the indirect jump through GOT[4] transfers control to `addvec`. The only additional overhead from this point on is the memory reference for the indirect jump.

7.13 Tools for Manipulating Object Files

There are a number of tools available on Unix systems to help you understand and manipulate object files. In particular, the GNU *binutils* package is especially helpful and runs on every Unix platform.

AR: Creates static libraries, and inserts, deletes, lists, and extracts members.

STRINGS: Lists all of the printable strings contained in an object file.

STRIP: Deletes symbol table information from an object file.

NM: Lists the symbols defined in the symbol table of an object file.

SIZE: Lists the names and sizes of the sections in an object file.

READELF: Displays the complete structure of an object file, including all of the information encoded in the ELF header; subsumes the functionality of SIZE and NM.

OBJDUMP: The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the `.text` section.

Unix systems also provide the LDD program for manipulating shared libraries:

LDD: Lists the shared libraries that an executable needs at run time.

7.14 Summary

Linking can be performed at compile time by static linkers, and at load time and run time by dynamic linkers. Linkers manipulate binary files called object files, which come in three different forms: relocatable, executable, and shared. Relocatable object files are combined by static linkers into an executable object file that can be loaded into memory and executed. Shared object files (shared libraries) are linked and loaded by dynamic linkers at run time, either implicitly when the calling program is loaded and begins executing, or on demand, when the program calls functions from the `dlopen` library.

The two main tasks of linkers are symbol resolution, where each global symbol in an object file is bound to a unique definition, and relocation, where the ultimate memory address for each symbol is determined and where references to those objects are modified.

Static linkers are invoked by compiler drivers such as GCC. They combine multiple relocatable object files into a single executable object file. Multiple object files can define the same symbol, and the rules that linkers use for silently resolving these multiple definitions can introduce subtle bugs in user programs.

Multiple object files can be concatenated in a single static library. Linkers use libraries to resolve symbol references in other object modules. The left-to-right sequential scan that many linkers use to resolve symbol references is another source of confusing link-time errors.

Loaders map the contents of executable files into memory and run the program. Linkers can also produce partially linked executable object files with unresolved references to the routines and data defined in a shared library. At load time, the loader maps the partially linked executable into memory and then calls a dynamic linker, which completes the linking task by loading the shared library and relocating the references in the program.

Shared libraries that are compiled as position-independent code can be loaded anywhere and shared at run time by multiple processes. Applications can also use the dynamic linker at run time in order to load, link, and access the functions and data in shared libraries.

Bibliographic Notes

Linking is not well documented in the computer systems literature. Since it lies at the intersection of compilers, computer architecture, and operating systems, linking requires understanding of code generation, machine-language programming, program instantiation, and virtual memory. It does not fit neatly into any of the usual computer systems specialties and thus is not well covered by the classic texts in these areas. However, Levine's monograph provides a good general reference on the subject [66]. The original specifications for ELF and DWARF (a specification for the contents of the `.debug` and `.line` sections) are described in [52].

Some interesting research and commercial activity centers around the notion of *binary translation*, where the contents of an object file are parsed, analyzed,

and modified. Binary translation can be used for three different purposes [64]: to emulate one system on another system, to observe program behavior, or to perform system-dependent optimizations that are not possible at compile time. Commercial products such as VTune, Purify, and BoundsChecker use binary translation to provide programmers with detailed observations of their programs. Valgrind is a popular open-source alternative.

The Atom system provides a flexible mechanism for instrumenting Alpha executable object files and shared libraries with arbitrary C functions [103]. Atom has been used to build a myriad of analysis tools that trace procedure calls, profile instruction counts and memory referencing patterns, simulate memory system behavior, and isolate memory referencing errors. Etch [90] and EEL [64] provide roughly similar capabilities on different platforms. The Shade system uses binary translation for instruction profiling [23]. Dynamo [5] and Dyninst [15] provide mechanisms for instrumenting and optimizing executables in memory at run time. Smith and his colleagues have investigated binary translation for program profiling and optimization [121].

Homework Problems

7.6 ◆

Consider the following version of the `swap.c` function that counts the number of times it has been called:

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   static int *bufp1;
5
6   static void incr()
7   {
8       static int count=0;
9
10      count++;
11  }
12
13  void swap()
14  {
15      int temp;
16
17      incr();
18      bufp1 = &buf[1];
19      temp = *bufp0;
20      *bufp0 = *bufp1;
21      *bufp1 = temp;
22  }
```

For each symbol that is defined and referenced in `swap.o`, indicate if it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `main.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, or `.bss`) it occupies in that module.

Symbol	`swap.o .symtab` entry?	Symbol type	Module where defined	Section
`buf`				
`bufp0`				
`bufp1`				
`swap`				
`temp`				
`incr`				
`count`				

7.7 ◆

Without changing any variable names, modify `bar5.c` on page 666 so that `foo5.c` prints the correct values of `x` and `y` (i.e., the hex representations of integers 15213 and 15212).

7.8 ◆

In this problem, let `REF(x.i) --> DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example below, use this notation to indicate how the linker would resolve references to the multiply defined symbol in each module. If there is a link-time error (rule 1), write "ERROR." If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN."

A.
```
/* Module 1 */
int main()
{
}
```
```
/* Module 2 */
static int main=1;
int p2()
{
}
```

(a) `REF(main.1) --> DEF(_____.___)`

(b) `REF(main.2) --> DEF(_____.___)`

B.
```
/* Module 1 */
int x;
void main()
{
}
```
```
/* Module 2 */
double x;
int p2()
{
}
```

(a) `REF(x.1) --> DEF(_____.___)`

(b) `REF(x.2) --> DEF(_____.___)`

C.
```
/* Module 1 */
int x=1;
void main()
{
}
```

```
/* Module 2 */
double x=1.0;
int p2()
{
}
```

(a) REF(x.1) --> DEF(_____.___)

(b) REF(x.2) --> DEF(_____.___)

7.9 ◆

Consider the following program, which consists of two object modules:

```
1  /* foo6.c */
2  void p2(void);
3
4  int main()
5  {
6      p2();
7      return 0;
8  }
```

```
1  /* bar6.c */
2  #include <stdio.h>
3
4  char main;
5
6  void p2()
7  {
8      printf("0x%x\n", main);
9  }
```

When this program is compiled and executed on a Linux system, it prints the string "0x55\n" and terminates normally, even though p2 never initializes variable main. Can you explain this?

7.10 ◆

Let a and b denote object modules or static libraries in the current directory, and let a→b denote that a depends on b, in the sense that b defines a symbol that is referenced by a. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references:

A. p.o → libx.a → p.o

B. p.o → libx.a → liby.a **and** liby.a → libx.a

C. p.o → libx.a → liby.a → libz.a **and** liby.a → libx.a → libz.a

7.11 ◆

The segment header in Figure 7.12 indicates that the data segment occupies 0x104 bytes in memory. However, only the first 0xe8 bytes of these come from the sections of the executable file. What causes this discrepancy?

7.12 ◆◆

The swap routine in Figure 7.10 contains five relocated references. For each relocated reference, give its line number in Figure 7.10, its run-time memory address, and its value. The original code and relocation entries in the swap.o module are shown in Figure 7.19.

```
1   00000000 <swap>:
2      0:   55                      push   %ebp
3      1:   8b 15 00 00 00 00       mov    0x0,%edx        Get *bufp0=&buf[0]
4                                   3: R_386_32     bufp0  Relocation entry
5      7:   a1 04 00 00 00          mov    0x4,%eax        Get buf[1]
6                                   8: R_386_32     buf    Relocation entry
7      c:   89 e5                   mov    %esp,%ebp
8      e:   c7 05 00 00 00 00 04    movl   $0x4,0x0        bufp1 = &buf[1];
9     15:   00 00 00
10                                  10: R_386_32    bufp1  Relocation entry
11                                  14: R_386_32    buf    Relocation entry
12     18:  89 ec                   mov    %ebp,%esp
13     1a:  8b 0a                   mov    (%edx),%ecx     temp = buf[0];
14     1c:  89 02                   mov    %eax,(%edx)     buf[0]=buf[1];
15     1e:  a1 00 00 00 00          mov    0x0,%eax        Get *bufp1=&buf[1]
16                                  1f: R_386_32    bufp1  Relocation entry
17     23:  89 08                   mov    %ecx,(%eax)     buf[1]=temp;
18     25:  5d                      pop    %ebp
19     26:  c3                      ret
```

Figure 7.19 **Code and relocation entries for Problem 7.12.**

Line # in Fig. 7.10	Address	Value

7.13 ◆◆◆

Consider the C code and corresponding relocatable object module in Figure 7.20.

A. Determine which instructions in `.text` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

B. Determine which data objects in `.data` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

Feel free to use tools such as OBJDUMP to help you solve this problem.

7.14 ◆◆◆

Consider the C code and corresponding relocatable object module in Figure 7.21.

A. Determine which instructions in `.text` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

(a) C code

```
1   extern int p3(void);
2   int x = 1;
3   int *xp = &x;
4
5   void p2(int y) {
6   }
7
8   void p1() {
9       p2(*xp + p3());
10  }
```

(b) `.text` section of relocatable object file

```
1   00000000 <p2>:
2      0:   55                      push   %ebp
3      1:   89 e5                   mov    %esp,%ebp
4      3:   89 ec                   mov    %ebp,%esp
5      5:   5d                      pop    %ebp
6      6:   c3                      ret

7   00000008 <p1>:
8      8:   55                      push   %ebp
9      9:   89 e5                   mov    %esp,%ebp
10     b:   83 ec 08                sub    $0x8,%esp
11     e:   83 c4 f4                add    $0xfffffff4,%esp
12    11:   e8 fc ff ff ff          call   12 <p1+0xa>
13    16:   89 c2                   mov    %eax,%edx
14    18:   a1 00 00 00 00          mov    0x0,%eax
15    1d:   03 10                   add    (%eax),%edx
16    1f:   52                      push   %edx
17    20:   e8 fc ff ff ff          call   21 <p1+0x19>
18    25:   89 ec                   mov    %ebp,%esp
19    27:   5d                      pop    %ebp
20    28:   c3                      ret
```

(c) `.data` section of relocatable object file

```
1   00000000 <x>:
2      0:   01 00 00 00
3   00000004 <xp>:
4      4:   00 00 00 00
```

Figure 7.20 **Example code for Problem 7.13.**

(a) C code

```
1   int relo3(int val)  {
2       switch (val) {
3       case 100:
4           return(val);
5       case 101:
6           return(val+1);
7       case 103: case 104:
8           return(val+3);
9       case 105:
10          return(val+5);
11      default:
12          return(val+6);
13      }
14  }
```

(b) `.text` section of relocatable object file

```
1   00000000 <relo3>:
2      0:   55                      push   %ebp
3      1:   89 e5                   mov    %esp,%ebp
4      3:   8b 45 08                mov    0x8(%ebp),%eax
5      6:   8d 50 9c                lea    0xffffff9c(%eax),%edx
6      9:   83 fa 05                cmp    $0x5,%edx
7      c:   77 17                   ja     25 <relo3+0x25>
8      e:   ff 24 95 00 00 00 00    jmp    *0x0(,%edx,4)
9     15:   40                      inc    %eax
10    16:   eb 10                   jmp    28 <relo3+0x28>
11    18:   83 c0 03                add    $0x3,%eax
12    1b:   eb 0b                   jmp    28 <relo3+0x28>
13    1d:   8d 76 00                lea    0x0(%esi),%esi
14    20:   83 c0 05                add    $0x5,%eax
15    23:   eb 03                   jmp    28 <relo3+0x28>
16    25:   83 c0 06                add    $0x6,%eax
17    28:   89 ec                   mov    %ebp,%esp
18    2a:   5d                      pop    %ebp
19    2b:   c3                      ret
```

(c) `.rodata` section of relocatable object file

```
      This is the jump table for the switch statement
1   0000 28000000 15000000 25000000 18000000 4 words at offsets 0x0,0x4,0x8, and 0xc
2   0010 18000000 20000000                   2 words at offsets 0x10 and 0x14
```

Figure 7.21 **Example code for Problem 7.14.**

B. Determine which data objects in `.rodata` will need to be modified by the linker when the module is relocated. For each such instruction, list the information in its relocation entry: section offset, relocation type, and symbol name.

Feel free to use tools such as OBJDUMP to help you solve this problem.

7.15 ◆◆◆

Performing the following tasks will help you become more familiar with the various tools for manipulating object files.

A. How many object files are contained in the versions of `libc.a` and `libm.a` on your system?

B. Does `gcc -O2` produce different executable code than `gcc -O2 -g`?

C. What shared libraries does the GCC driver on your system use?

Solutions to Practice Problems

Solution to Problem 7.1 (page 662)

The purpose of this problem is to help you understand the relationship between linker symbols and C variables and functions. Notice that the C local variable `temp` does *not* have a symbol table entry.

Symbol	`swap.o` `.symtab` entry?	Symbol type	Module where defined	Section
`buf`	yes	extern	`main.o`	`.data`
`bufp0`	yes	global	`swap.o`	`.data`
`bufp1`	yes	global	`swap.o`	`.bss`
`swap`	yes	global	`swap.o`	`.text`
`temp`	no	—	—	—

Solution to Problem 7.2 (page 666)

This is a simple drill that checks your understanding of the rules that a Unix linker uses when it resolves global symbols that are defined in more than one module. Understanding these rules can help you avoid some nasty programming bugs.

A. The linker chooses the strong symbol defined in module 1 over the weak symbol defined in module 2 (rule 2):

(a) `REF(main.1) --> DEF(main.1)`

(b) `REF(main.2) --> DEF(main.1)`

B. This is an ERROR, because each module defines a strong symbol `main` (rule 1).

C. The linker chooses the strong symbol defined in module 2 over the weak symbol defined in module 1 (rule 2):

(a) `REF(x.1) --> DEF(x.2)`

(b) `REF(x.2) --> DEF(x.2)`

Solution to Problem 7.3 (page 672)

Placing static libraries in the wrong order on the command line is a common source of linker errors that confuses many programmers. However, once you understand how linkers use static libraries to resolve references, it's pretty straightforward. This little drill checks your understanding of this idea:

A. `gcc p.o libx.a`

B. `gcc p.o libx.a liby.a`

C. `gcc p.o libx.a liby.a libx.a`

Solution to Problem 7.4 (page 676)

This problem concerns the disassembly listing in Figure 7.10. Our purpose here is to give you some practice reading disassembly listings and to check your understanding of PC-relative addressing.

A. The hex address of the relocated reference in line 5 is `0x80483bb`.

B. The hex value of the relocated reference in line 5 is `0x9`. Remember that the disassembly listing shows the value of the reference in little-endian byte order.

C. The key observation here is that no matter where the linker locates the `.text` section, the distance between the reference and the `swap` function is always the same. Thus, because the reference is a PC-relative address, its value will be `0x9`, regardless of where the linker locates the `.text` section.

Solution to Problem 7.5 (page 681)

How C programs actually start up is a mystery to most programmers. These questions check your understanding of this startup process. You can answer them by referring to the C startup code in Figure 7.14.

A. Every program needs a `main` function, because the C startup code, which is common to every C program, jumps to a function called `main`.

B. If `main` terminates with a `return` statement, then control passes back to the startup routine, which returns control to the operating system by calling `_exit`. The same behavior occurs if the user omits the `return` statement. If `main` terminates with a call to `exit`, then `exit` eventually returns control to the operating system by calling `_exit`. The net effect is the same in all three cases: when `main` has finished, control passes back to the operating system.

This page intentionally left blank

CHAPTER 8

Exceptional Control Flow

From the time you first apply power to a processor until the time you shut it off, the program counter assumes a sequence of values

$$a_0, a_1, \ldots, a_{n-1}$$

where each a_k is the address of some corresponding instruction I_k. Each transition from a_k to a_{k+1} is called a *control transfer*. A sequence of such control transfers is called the *flow of control*, or *control flow* of the processor.

The simplest kind of control flow is a "smooth" sequence where each I_k and I_{k+1} are adjacent in memory. Typically, abrupt changes to this smooth flow, where I_{k+1} is not adjacent to I_k, are caused by familiar program instructions such as jumps, calls, and returns. Such instructions are necessary mechanisms that allow programs to react to changes in internal program state represented by program variables.

But systems must also be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program. For example, a hardware timer goes off at regular intervals and must be dealt with. Packets arrive at the network adapter and must be stored in memory. Programs request data from a disk and then sleep until they are notified that the data are ready. Parent processes that create child processes must be notified when their children terminate.

Modern systems react to these situations by making abrupt changes in the control flow. In general, we refer to these abrupt changes as *exceptional control flow* (ECF). Exceptional control flow occurs at all levels of a computer system. For example, at the hardware level, events detected by the hardware trigger abrupt control transfers to exception handlers. At the operating systems level, the kernel transfers control from one user process to another via context switches. At the application level, a process can send a *signal* to another process that abruptly transfers control to a signal handler in the recipient. An individual program can react to errors by sidestepping the usual stack discipline and making nonlocal jumps to arbitrary locations in other functions.

As programmers, there are a number of reasons why it is important for you to understand ECF:

- *Understanding ECF will help you understand important systems concepts.* ECF is the basic mechanism that operating systems use to implement I/O, processes, and virtual memory. Before you can really understand these important ideas, you need to understand ECF.
- *Understanding ECF will help you understand how applications interact with the operating system.* Applications request services from the operating system by using a form of ECF known as a *trap* or *system call*. For example, writing data to a disk, reading data from a network, creating a new process, and terminating the current process are all accomplished by application programs invoking system calls. Understanding the basic system call mechanism will help you understand how these services are provided to applications.
- *Understanding ECF will help you write interesting new application programs.* The operating system provides application programs with powerful ECF

mechanisms for creating new processes, waiting for processes to terminate, notifying other processes of exceptional events in the system, and detecting and responding to these events. If you understand these ECF mechanisms, then you can use them to write interesting programs such as Unix shells and Web servers.

- *Understanding ECF will help you understand concurrency.* ECF is a basic mechanism for implementing concurrency in computer systems. An exception handler that interrupts the execution of an application program, processes and threads whose execution overlap in time, and a signal handler that interrupts the execution of an application program are all examples of concurrency in action. Understanding ECF is a first step to understanding concurrency. We will return to study it in more detail in Chapter 12.
- *Understanding ECF will help you understand how software exceptions work.* Languages such as C++ and Java provide software exception mechanisms via `try`, `catch`, and `throw` statements. Software exceptions allow the program to make *nonlocal* jumps (i.e., jumps that violate the usual call/return stack discipline) in response to error conditions. Nonlocal jumps are a form of application-level ECF, and are provided in C via the `setjmp` and `longjmp` functions. Understanding these low-level functions will help you understand how higher-level software exceptions can be implemented.

Up to this point in your study of systems, you have learned how applications interact with the hardware. This chapter is pivotal in the sense that you will begin to learn how your applications interact with the operating system. Interestingly, these interactions all revolve around ECF. We describe the various forms of ECF that exist at all levels of a computer system. We start with exceptions, which lie at the intersection of the hardware and the operating system. We also discuss system calls, which are exceptions that provide applications with entry points into the operating system. We then move up a level of abstraction and describe processes and signals, which lie at the intersection of applications and the operating system. Finally, we discuss nonlocal jumps, which are an application-level form of ECF.

8.1 Exceptions

Exceptions are a form of exceptional control flow that are implemented partly by the hardware and partly by the operating system. Because they are partly implemented in hardware, the details vary from system to system. However, the basic ideas are the same for every system. Our aim in this section is to give you a general understanding of exceptions and exception handling, and to help demystify what is often a confusing aspect of modern computer systems.

An *exception* is an abrupt change in the control flow in response to some change in the processor's state. Figure 8.1 shows the basic idea. In the figure, the processor is executing some current instruction I_{curr} when a significant change in the processor's *state* occurs. The state is encoded in various bits and signals inside the processor. The change in state is known as an *event*. The event might be directly

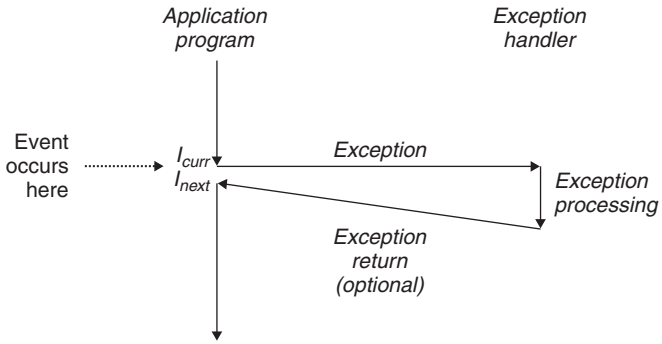


Figure 8.1
Anatomy of an exception. A change in the processor's state (event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

related to the execution of the current instruction. For example, a virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. On the other hand, the event might be unrelated to the execution of the current instruction. For example, a system timer goes off or an I/O request completes.

In any case, when the processor detects that the event has occurred, it makes an indirect procedure call (the exception), through a jump table called an *exception table*, to an operating system subroutine (the *exception handler*) that is specifically designed to process this particular kind of event.

When the exception handler finishes processing, one of three things happens, depending on the type of event that caused the exception:

1. The handler returns control to the current instruction I_{curr}, the instruction that was executing when the event occurred.
2. The handler returns control to I_{next}, the instruction that would have executed next had the exception not occurred.
3. The handler aborts the interrupted program.

Section 8.1.2 says more about these possibilities.

Aside Hardware vs. software exceptions

C++ and Java programmers will have noticed that the term "exception" is also used to describe the application-level ECF mechanism provided by C++ and Java in the form of `catch`, `throw`, and `try` statements. If we wanted to be perfectly clear, we might distinguish between "hardware" and "software" exceptions, but this is usually unnecessary because the meaning is clear from the context.

8.1.1 Exception Handling

Exceptions can be difficult to understand because handling them involves close cooperation between hardware and software. It is easy to get confused about which component performs which task. Let's look at the division of labor between hardware and software in more detail.

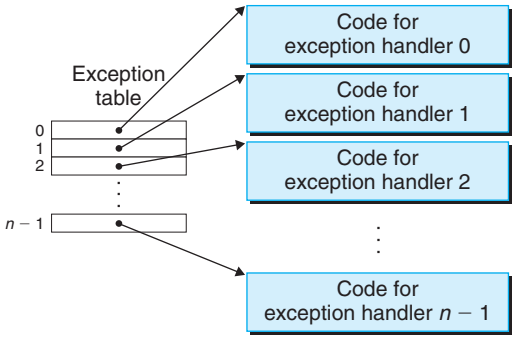


Figure 8.2
Exception table. The exception table is a jump table where entry k contains the address of the handler code for exception k.

Each type of possible exception in a system is assigned a unique nonnegative integer *exception number*. Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system *kernel* (the memory-resident part of the operating system). Examples of the former include divide by zero, page faults, memory access violations, breakpoints, and arithmetic overflows. Examples of the latter include system calls and signals from external I/O devices.

At system boot time (when the computer is reset or powered on), the operating system allocates and initializes a jump table called an *exception table*, so that entry k contains the address of the handler for exception k. Figure 8.2 shows the format of an exception table.

At run time (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an indirect procedure call, through entry k of the exception table, to the corresponding handler. Figure 8.3 shows how the processor uses the exception table to form the address of the appropriate exception handler. The exception number is an index into the exception table, whose starting address is contained in a special CPU register called the *exception table base register*.

An exception is akin to a procedure call, but with some important differences.

- As with a procedure call, the processor pushes a return address on the stack before branching to the handler. However, depending on the class of exception, the return address is either the current instruction (the instruction that

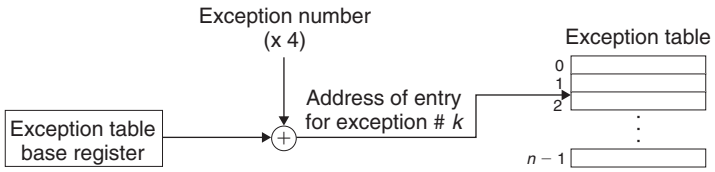


Figure 8.3
Generating the address of an exception handler. The exception number is an index into the exception table.

was executing when the event occurred) or the next instruction (the instruction that would have executed after the current instruction had the event not occurred).

- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns. For example, an IA32 system pushes the EFLAGS register containing, among other things, the current condition codes, onto the stack.
- If control is being transferred from a user program to the kernel, all of these items are pushed onto the kernel's stack rather than onto the user's stack.
- Exception handlers run in *kernel mode* (Section 8.2.4), which means they have complete access to all system resources.

Once the hardware triggers the exception, the rest of the work is done in software by the exception handler. After the handler has processed the event, it optionally returns to the interrupted program by executing a special "return from interrupt" instruction, which pops the appropriate state back into the processor's control and data registers, restores the state to *user mode* (Section 8.2.4) if the exception interrupted a user program, and then returns control to the interrupted program.

8.1.2 Classes of Exceptions

Exceptions can be divided into four classes: *interrupts*, *traps*, *faults*, and *aborts*. The table in Figure 8.4 summarizes the attributes of these classes.

Interrupts

Interrupts occur *asynchronously* as a result of signals from I/O devices that are external to the processor. Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction. Exception handlers for hardware interrupts are often called *interrupt handlers*.

Figure 8.5 summarizes the processing for an interrupt. I/O devices such as network adapters, disk controllers, and timer chips trigger interrupts by signaling a pin on the processor chip and placing onto the system bus the exception number that identifies the device that caused the interrupt.

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4 **Classes of exceptions.** Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

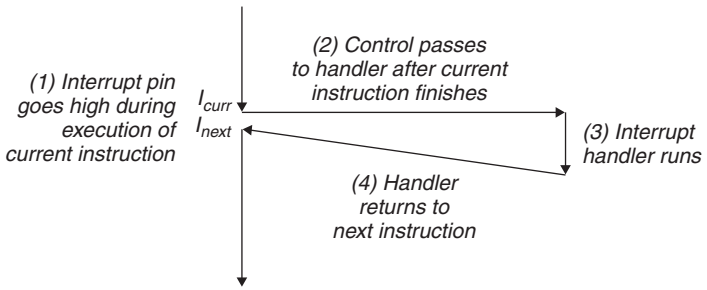


Figure 8.5
Interrupt handling. The interrupt handler returns control to the next instruction in the application program's control flow.

After the current instruction finishes executing, the processor notices that the interrupt pin has gone high, reads the exception number from the system bus, and then calls the appropriate interrupt handler. When the handler returns, it returns control to the next instruction (i.e., the instruction that would have followed the current instruction in the control flow had the interrupt not occurred). The effect is that the program continues executing as though the interrupt had never happened.

The remaining classes of exceptions (traps, faults, and aborts) occur *synchronously* as a result of executing the current instruction. We refer to this instruction as the *faulting instruction*.

Traps and System Calls

Traps are *intentional* exceptions that occur as a result of executing an instruction. Like interrupt handlers, trap handlers return control to the next instruction. The most important use of traps is to provide a procedure-like interface between user programs and the kernel known as a *system call*.

User programs often need to request services from the kernel such as reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), or terminating the current process (`exit`). To allow controlled access to such kernel services, processors provide a special "`syscall` *n*" instruction that user programs can execute when they want to request service *n*. Executing the `syscall` instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine. Figure 8.6 summarizes the processing for a system call. From a programmer's perspective, a system call is identical to a

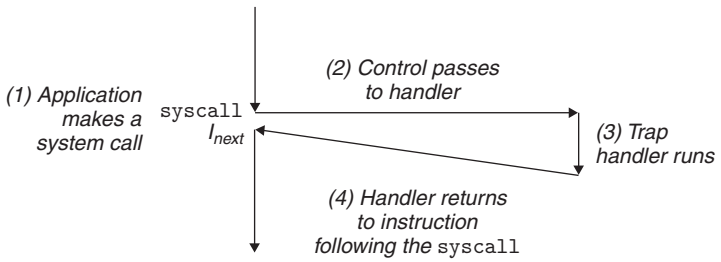


Figure 8.6
Trap handling. The trap handler returns control to the next instruction in the application program's control flow.

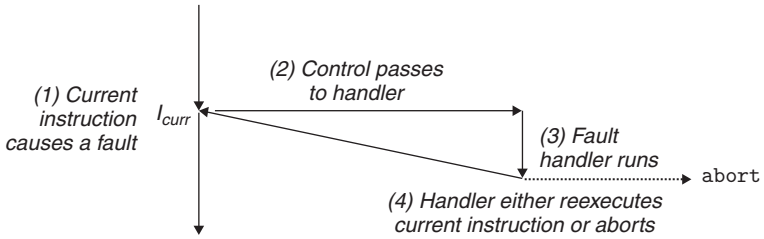


Figure 8.7
Fault handling. Depending on whether the fault can be repaired or not, the fault handler either reexecutes the faulting instruction or aborts.

regular function call. However, their implementations are quite different. Regular functions run in *user mode*, which restricts the types of instructions they can execute, and they access the same stack as the calling function. A system call runs in *kernel mode*, which allows it to execute instructions, and accesses a stack defined in the kernel. Section 8.2.4 discusses user and kernel modes in more detail.

Faults

Faults result from error conditions that a handler might be able to correct. When a fault occurs, the processor transfers control to the fault handler. If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby reexecuting it. Otherwise, the handler returns to an `abort` routine in the kernel that terminates the application program that caused the fault. Figure 8.7 summarizes the processing for a fault.

A classic example of a fault is the page fault exception, which occurs when an instruction references a virtual address whose corresponding physical page is not resident in memory and must therefore be retrieved from disk. As we will see in Chapter 9, a page is a contiguous block (typically 4 KB) of virtual memory. The page fault handler loads the appropriate page from disk and then returns control to the instruction that caused the fault. When the instruction executes again, the appropriate physical page is resident in memory and the instruction is able to run to completion without faulting.

Aborts

Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program. As shown in Figure 8.8, the handler returns control to an `abort` routine that terminates the application program.

8.1.3 Exceptions in Linux/IA32 Systems

To help make things more concrete, let's look at some of the exceptions defined for IA32 systems. There are up to 256 different exception types [27]. Numbers in the range from 0 to 31 correspond to exceptions that are defined by the Intel architects, and thus are identical for any IA32 system. Numbers in the range from

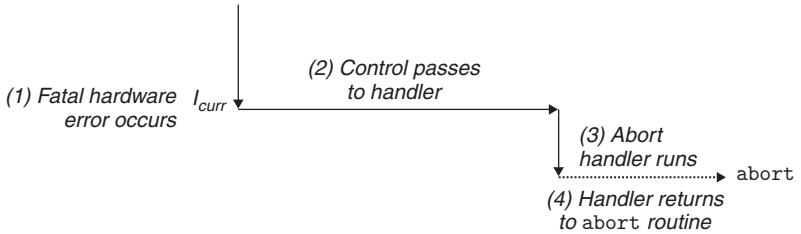


Figure 8.8
Abort handling. The abort handler passes control to a kernel `abort` routine that terminates the application program.

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

Figure 8.9 **Examples of exceptions in IA32 systems.**

32 to 255 correspond to interrupts and traps that are defined by the operating system. Figure 8.9 shows a few examples.

Linux/IA32 Faults and Aborts

Divide error. A divide error (exception 0) occurs when an application attempts to divide by zero, or when the result of a divide instruction is too big for the destination operand. Unix does not attempt to recover from divide errors, opting instead to abort the program. Linux shells typically report divide errors as "Floating exceptions."

General protection fault. The infamous general protection fault (exception 13) occurs for many reasons, usually because a program references an undefined area of virtual memory, or because the program attempts to write to a read-only text segment. Linux does not attempt to recover from this fault. Linux shells typically report general protection faults as "Segmentation faults."

Page fault. A page fault (exception 14) is an example of an exception where the faulting instruction is restarted. The handler maps the appropriate page of physical memory on disk into a page of virtual memory, and then restarts the faulting instruction. We will see how page faults work in detail in Chapter 9.

Machine check. A machine check (exception 18) occurs as a result of a fatal hardware error that is detected during the execution of the faulting instruction. Machine check handlers never return control to the application program.

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

Figure 8.10 **Examples of popular system calls in Linux/IA32 systems.** Linux provides hundreds of system calls. Source: `/usr/include/sys/syscall.h`.

Linux/IA32 System Calls

Linux provides hundreds of system calls that application programs use when they want to request services from the kernel, such as reading a file, writing a file, or creating a new process. Figure 8.10 shows some of the more popular Linux system calls. Each system call has a unique integer number that corresponds to an offset in a jump table in the kernel.

System calls are provided on IA32 systems via a trapping instruction called `int` *n*, where *n* can be the index of any of the 256 entries in the IA32 exception table. Historically, system calls are provided through exception 128 (`0x80`).

C programs can invoke any system call directly by using the `syscall` function. However, this is rarely necessary in practice. The standard C library provides a set of convenient wrapper functions for most system calls. The wrapper functions package up the arguments, trap to the kernel with the appropriate system call number, and then pass the return status of the system call back to the calling program. Throughout this text, we will refer to system calls and their associated wrapper functions interchangeably as *system-level functions*.

It is quite interesting to study how programs can use the `int` instruction to invoke Linux system calls directly. All parameters to Linux system calls are passed through general purpose registers rather than the stack. By convention, register `%eax` contains the syscall number, and registers `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, and `%ebp` contain up to six arbitrary arguments. The stack pointer `%esp` cannot be used because it is overwritten by the kernel when it enters kernel mode.

For example, consider the following version of the familiar `hello` program, written using the `write` system-level function:

```
1  int main()
2  {
3      write(1, "hello, world\n", 13);
4      exit(0);
5  }
```

code/ecf/hello-asm.sa

```
1   .section .data
2   string:
3     .ascii "hello, world\n"
4   string_end:
5     .equ len, string_end - string

6   .section .text
7   .globl main
8   main:
     First, call write(1, "hello, world\n", 13)
9     movl $4, %eax          System call number 4
10    movl $1, %ebx          stdout has descriptor 1
11    movl $string, %ecx     Hello world string
12    movl $len, %edx        String length
13    int $0x80              System call code

     Next, call exit(0)
14    movl $1, %eax          System call number 0
15    movl $0, %ebx          Argument is 0
16    int $0x80              System call code
```

code/ecf/hello-asm.sa

Figure 8.11 **Implementing the `hello` program directly with Linux system calls.**

The first argument to `write` sends the output to `stdout`. The second argument is the sequence of bytes to write, and the third argument gives the number of bytes to write.

Figure 8.11 shows an assembly language version of `hello` that uses the `int` instruction to invoke the `write` and `exit` system calls directly. Lines 9–13 invoke the `write` function. First, line 9 stores the number for the `write` system call in `%eax`, and lines 10–12 set up the argument list. Then line 13 uses the `int` instruction to invoke the system call. Similarly, lines 14–16 invoke the `exit` system call.

Aside A note on terminology

The terminology for the various classes of exceptions varies from system to system. Processor macroarchitecture specifications often distinguish between asynchronous "interrupts" and synchronous "exceptions," yet provide no umbrella term to refer to these very similar concepts. To avoid having to constantly refer to "exceptions and interrupts" and "exceptions or interrupts," we use the word "exception" as the general term and distinguish between asynchronous exceptions (interrupts) and synchronous exceptions (traps, faults, and aborts) only when it is appropriate. As we have noted, the basic ideas are the same for every system, but you should be aware that some manufacturers' manuals use the word "exception" to refer only to those changes in control flow caused by synchronous events.

8.2 Processes

Exceptions are the basic building blocks that allow the operating system to provide the notion of a *process*, one of the most profound and successful ideas in computer science.

When we run a program on a modern system, we are presented with the illusion that our program is the only one currently running in the system. Our program appears to have exclusive use of both the processor and the memory. The processor appears to execute the instructions in our program, one after the other, without interruption. Finally, the code and data of our program appear to be the only objects in the system's memory. These illusions are provided to us by the notion of a process.

The classic definition of a process is *an instance of a program in execution*. Each program in the system runs in the *context* of some process. The context consists of the state that the program needs to run correctly. This state includes the program's code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors.

Each time a user runs a program by typing the name of an executable object file to the shell, the shell creates a new process and then runs the executable object file in the context of this new process. Application programs can also create new processes and run either their own code or other applications in the context of the new process.

A detailed discussion of how operating systems implement processes is beyond our scope. Instead, we will focus on the key abstractions that a process provides to the application:

- An independent *logical control flow* that provides the illusion that our program has exclusive use of the processor.
- A private address space that provides the illusion that our program has exclusive use of the memory system.

Let's look more closely at these abstractions.

8.2.1 Logical Control Flow

A process provides each program with the illusion that it has exclusive use of the processor, even though many other programs are typically running concurrently on the system. If we were to use a debugger to single step the execution of our program, we would observe a series of program counter (PC) values that corresponded exclusively to instructions contained in our program's executable object file or in shared objects linked into our program dynamically at run time. This sequence of PC values is known as a *logical control flow*, or simply *logical flow*.

Consider a system that runs three processes, as shown in Figure 8.12. The single physical control flow of the processor is partitioned into three logical flows, one for each process. Each vertical line represents a portion of the logical flow for

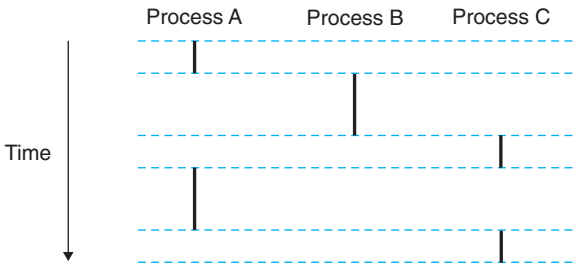


Figure 8.12
Logical control flows. Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.

a process. In the example, the execution of the three logical flows is interleaved. Process A runs for a while, followed by B, which runs to completion. Process C then runs for awhile, followed by A, which runs to completion. Finally, C is able to run to completion.

The key point in Figure 8.12 is that processes take turns using the processor. Each process executes a portion of its flow and then is *preempted* (temporarily suspended) while other processes take their turns. To a program running in the context of one of these processes, it appears to have exclusive use of the processor. The only evidence to the contrary is that if we were to precisely measure the elapsed time of each instruction, we would notice that the CPU appears to periodically stall between the execution of some of the instructions in our program. However, each time the processor stalls, it subsequently resumes execution of our program without any change to the contents of the program's memory locations or registers.

8.2.2 Concurrent Flows

Logical flows take many different forms in computer systems. Exception handlers, processes, signal handlers, threads, and Java processes are all examples of logical flows.

A logical flow whose execution overlaps in time with another flow is called a *concurrent flow*, and the two flows are said to *run concurrently*. More precisely, flows X and Y are concurrent with respect to each other if and only if X begins after Y begins and before Y finishes, or Y begins after X begins and before X finishes. For example, in Figure 8.12, processes A and B run concurrently, as do A and C. On the other hand, B and C do not run concurrently, because the last instruction of B executes before the first instruction of C.

The general phenomenon of multiple flows executing concurrently is known as *concurrency*. The notion of a process taking turns with other processes is also known as *multitasking*. Each time period that a process executes a portion of its flow is called a *time slice*. Thus, multitasking is also referred to as *time slicing*. For example, in Figure 8.12, the flow for Process A consists of two time slices.

Notice that the idea of concurrent flows is independent of the number of processor cores or computers that the flows are running on. If two flows overlap in time, then they are concurrent, even if they are running on the same processor. However, we will sometimes find it useful to identify a proper subset of concurrent

flows known as *parallel flows*. If two flows are running concurrently on different processor cores or computers, then we say that they are *parallel flows*, that they are *running in parallel*, and have *parallel execution*.

Practice Problem 8.1

Consider three processes with the following starting and ending times:

Process	Start time	End time
A	0	2
B	1	4
C	3	5

For each pair of processes, indicate whether they run concurrently (y) or not (n):

Process pair	Concurrent?
AB	_____
AC	_____
BC	_____

8.2.3 Private Address Space

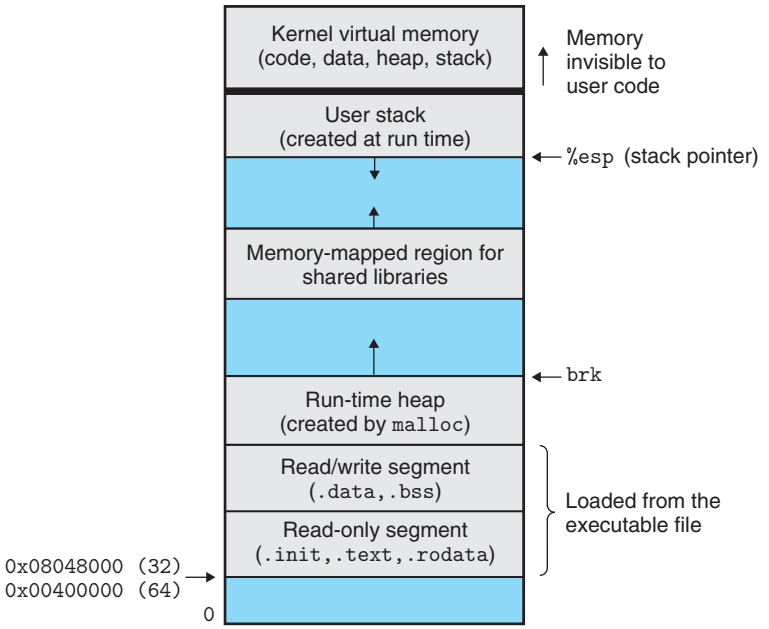
A process provides each program with the illusion that it has exclusive use of the system's address space. On a machine with n-bit addresses, the *address space* is the set of 2^n possible addresses, $0, 1, \ldots, 2^n - 1$. A process provides each program with its own *private address space*. This space is private in the sense that a byte of memory associated with a particular address in the space cannot in general be read or written by any other process.

Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.13 shows the organization of the address space for an x86 Linux process. The bottom portion of the address space is reserved for the user program, with the usual text, data, heap, and stack segments. Code segments begin at address `0x08048000` for 32-bit processes, and at address `0x00400000` for 64-bit processes. The top portion of the address space is reserved for the kernel. This part of the address space contains the code, data, and stack that the kernel uses when it executes instructions on behalf of the process (e.g., when the application program executes a system call).

8.2.4 User and Kernel Modes

In order for the operating system kernel to provide an airtight process abstraction, the processor must provide a mechanism that restricts the instructions that an application can execute, as well as the portions of the address space that it can access.

Figure 8.13
Process address space.



Processors typically provide this capability with a *mode bit* in some control register that characterizes the privileges that the process currently enjoys. When the mode bit is set, the process is running in *kernel mode* (sometimes called *supervisor mode*). A process running in kernel mode can execute any instruction in the instruction set and access any memory location in the system.

When the mode bit is not set, the process is running in *user mode*. A process in user mode is not allowed to execute *privileged instructions* that do things such as halt the processor, change the mode bit, or initiate an I/O operation. Nor is it allowed to directly reference code or data in the kernel area of the address space. Any such attempt results in a fatal protection fault. User programs must instead access kernel code and data indirectly via the system call interface.

A process running application code is initially in user mode. The only way for the process to change from user mode to kernel mode is via an exception such as an interrupt, a fault, or a trapping system call. When the exception occurs, and control passes to the exception handler, the processor changes the mode from user mode to kernel mode. The handler runs in kernel mode. When it returns to the application code, the processor changes the mode from kernel mode back to user mode.

Linux provides a clever mechanism, called the `/proc` filesystem, that allows user mode processes to access the contents of kernel data structures. The `/proc` filesystem exports the contents of many kernel data structures as a hierarchy of text files that can be read by user programs. For example, you can use the `/proc` filesystem to find out general system attributes such as CPU type (`/proc/cpuinfo`), or the memory segments used by a particular process (`/proc/<process id>/maps`).

The 2.6 version of the Linux kernel introduced a `/sys` filesystem, which exports additional low-level information about system buses and devices.

8.2.5 Context Switches

The operating system kernel implements multitasking using a higher-level form of exceptional control flow known as a *context switch*. The context switch mechanism is built on top of the lower-level exception mechanism that we discussed in Section 8.1.

The kernel maintains a *context* for each process. The context is the state that the kernel needs to restart a preempted process. It consists of the values of objects such as the general purpose registers, the floating-point registers, the program counter, user's stack, status registers, kernel's stack, and various kernel data structures such as a *page table* that characterizes the address space, a *process table* that contains information about the current process, and a *file table* that contains information about the files that the process has opened.

At certain points during the execution of a process, the kernel can decide to preempt the current process and restart a previously preempted process. This decision is known as *scheduling*, and is handled by code in the kernel called the *scheduler*. When the kernel selects a new process to run, we say that the kernel has *scheduled* that process. After the kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a mechanism called a *context switch* that (1) saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.

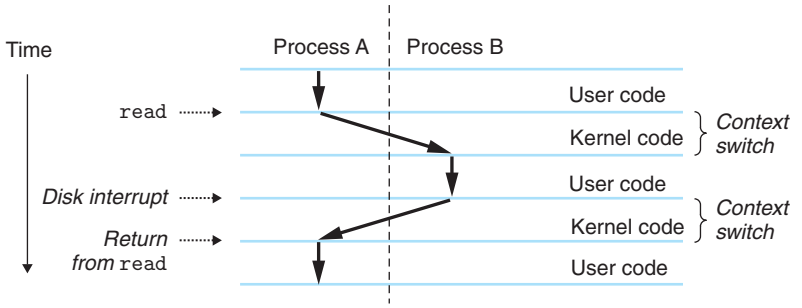
A context switch can occur while the kernel is executing a system call on behalf of the user. If the system call blocks because it is waiting for some event to occur, then the kernel can put the current process to sleep and switch to another process. For example, if a `read` system call requires a disk access, the kernel can opt to perform a context switch and run another process instead of waiting for the data to arrive from the disk. Another example is the `sleep` system call, which is an explicit request to put the calling process to sleep. In general, even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process.

A context switch can also occur as a result of an interrupt. For example, all systems have some mechanism for generating periodic timer interrupts, typically every 1 ms or 10 ms. Each time a timer interrupt occurs, the kernel can decide that the current process has run long enough and switch to a new process.

Figure 8.14 shows an example of context switching between a pair of processes A and B. In this example, initially process A is running in user mode until it traps to the kernel by executing a `read` system call. The trap handler in the kernel requests a DMA transfer from the disk controller and arranges for the disk to interrupt the processor after the disk controller has finished transferring the data from disk to memory.

The disk will take a relatively long time to fetch the data (on the order of tens of milliseconds), so instead of waiting and doing nothing in the interim, the kernel performs a context switch from process A to B. Note that before the switch,

Figure 8.14
Anatomy of a process context switch.



the kernel is executing instructions in user mode on behalf of process A. During the first part of the switch, the kernel is executing instructions in kernel mode on behalf of process A. Then at some point it begins executing instructions (still in kernel mode) on behalf of process B. And after the switch, the kernel is executing instructions in user mode on behalf of process B.

Process B then runs for a while in user mode until the disk sends an interrupt to signal that data has been transferred from disk to memory. The kernel decides that process B has run long enough and performs a context switch from process B to A, returning control in process A to the instruction immediately following the `read` system call. Process A continues to run until the next exception occurs, and so on.

Aside Cache pollution and exceptional control flow

In general, hardware cache memories do not interact well with exceptional control flows such as interrupts and context switches. If the current process is interrupted briefly by an interrupt, then the cache is cold for the interrupt handler. If the handler accesses enough items from main memory, then the cache will also be cold for the interrupted process when it resumes. In this case, we say that the handler has *polluted* the cache. A similar phenomenon occurs with context switches. When a process resumes after a context switch, the cache is cold for the application program and must be warmed up again.

8.3 System Call Error Handling

When Unix system-level functions encounter an error, they typically return -1 and set the global integer variable `errno` to indicate what went wrong. Programmers should *always* check for errors, but unfortunately, many skip error checking because it bloats the code and makes it harder to read. For example, here is how we might check for errors when we call the Linux `fork` function:

```
1       if ((pid = fork()) < 0) {
2           fprintf(stderr, "fork error: %s\n", strerror(errno));
3           exit(0);
4       }
```