(a) An array of structs

```
#define N 1000

typedef struct {
    int vel[3];
    int acc[3];
} point;

point p[N];
```

(b) The clear1 function

```
void clear1(point *p, int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++)
            p[i].vel[j] = 0;
        for (j = 0; j < 3; j++)
            p[i].acc[j] = 0;
    }
}
```

(c) The clear2 function

```
void clear2(point *p, int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++) {
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}
```

(d) The clear3 function

```
void clear3(point *p, int n)
{
    int i, j;

    for (j = 0; j < 3; j++) {
        for (i = 0; i < n; i++)
            p[i].vel[j] = 0;
        for (i = 0; i < n; i++)
            p[i].acc[j] = 0;
    }
}
```

Figure 6.20 Code examples for Practice Problem 6.8.

Practice Problem 6.8 (solution page 699)

The three functions in Figure 6.20 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.

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.

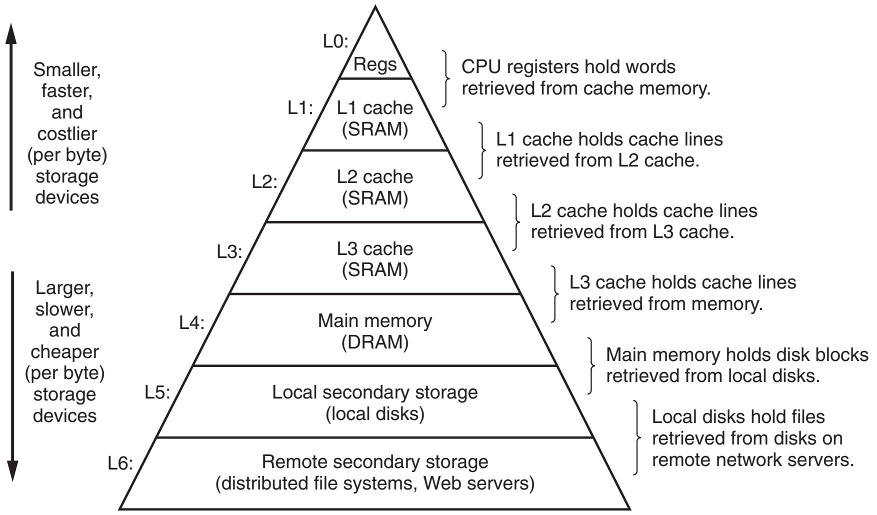
Smaller,
faster,
and
costlier
(per byte)
storage
devices
Larger,
slower,
and
cheaper
(per byte)
storage
devices
L0:
Regs
L1:
L1 cache
(SRAM)
L2:
L2 cache
(SRAM)
L3:
L3 cache
(SRAM)
L4:
Main memory
(DRAM)
L5:
Local secondary storage
(local disks)
L6:
Remote secondary storage
(distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Figure 6.21 **The memory hierarchy.**

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 6.21 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-size 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.

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

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, including Google datacenters, 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.

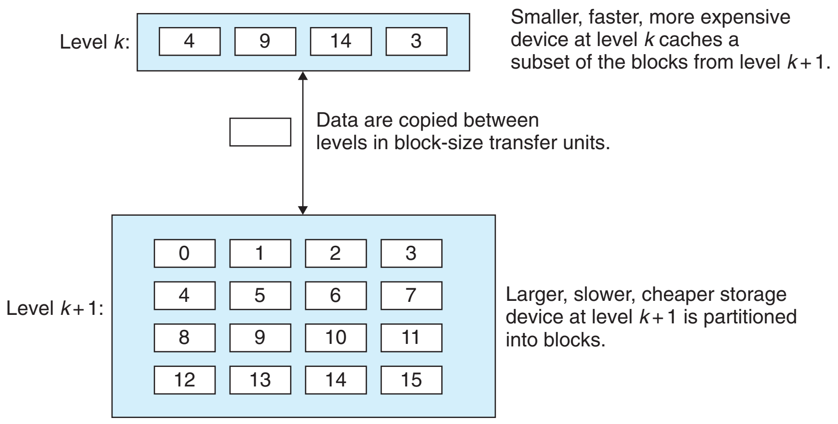
Level k:
4
9
14
3
Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1.
Data are copied between levels in block-size transfer units.
Level k+1:
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.

Figure 6.22 **The basic principle of caching in a memory hierarchy.**

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 6.22 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 size (the usual case) or variable size (e.g., the remote HTML files stored on Web servers). For example, the level $k + 1$ storage in Figure 6.22 is partitioned into 16 fixed-size 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 6.22, the cache at level k has room for four blocks and currently contains copies of blocks 4, 9, 14, and 3.

Data are always copied back and forth between level k and level $k+1$ in block-size *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 6.21, transfers between L1 and L0 typically use word-size blocks. Transfers between L2 and L1 (and L3 and L2, and L4 and L3) typically use blocks of tens of bytes. 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 6.22, 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 simpler 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 6.22, 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 6.22 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 6.22, 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.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte words	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-chip L1 cache	4	Hardware
L2 cache	64-byte blocks	On-chip L2 cache	10	Hardware
L3 cache	64-byte blocks	On-chip L3 cache	50	Hardware
Virtual memory	4-KB pages	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	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 6.23 **The ubiquity of caching in modern computer systems.** Acronyms: TLB: translation lookaside buffer; MMU: memory management unit; OS: operating system; NFS: network file system.

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 6.23, 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 6.23 that we haven't covered yet. We include them here to demonstrate how common caches are.

6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert