Memory Architecture — Chapter Six

6.1 Chapter Overview

This chapter discusses the memory hierarchy – the different types and performance levels of memory found on a typical 80x86 computer system. Many programmers tend to view memory as this big nebulous block of storage that holds values for future use. From a semantic point of view, this is a reasonable view. However, from a performance point of view there are many different kinds of memory and using the wrong one or using one form improperly can have a dramatically negative impact on the performance of a program. This chapter discusses the memory hierarchy and how to best use it within your programs.

6.2 The Memory Hierarchy

Most modern programs can benefit greatly from a large amount of very fast memory. A physical reality, however, is that as a memory device gets larger, it tends to get slower. For example, cache memories (see “Cache Memory” on page 153) are very fast but are also small and expensive. Main memory is inexpensive and large, but is slow (requiring wait states, see “Wait States” on page 151). The memory hierarchy is a mechanism of comparing the cost and performance of the various places we can store data and instructions. Figure 6.1 provides a look at one possible form of the memory hierarchy.

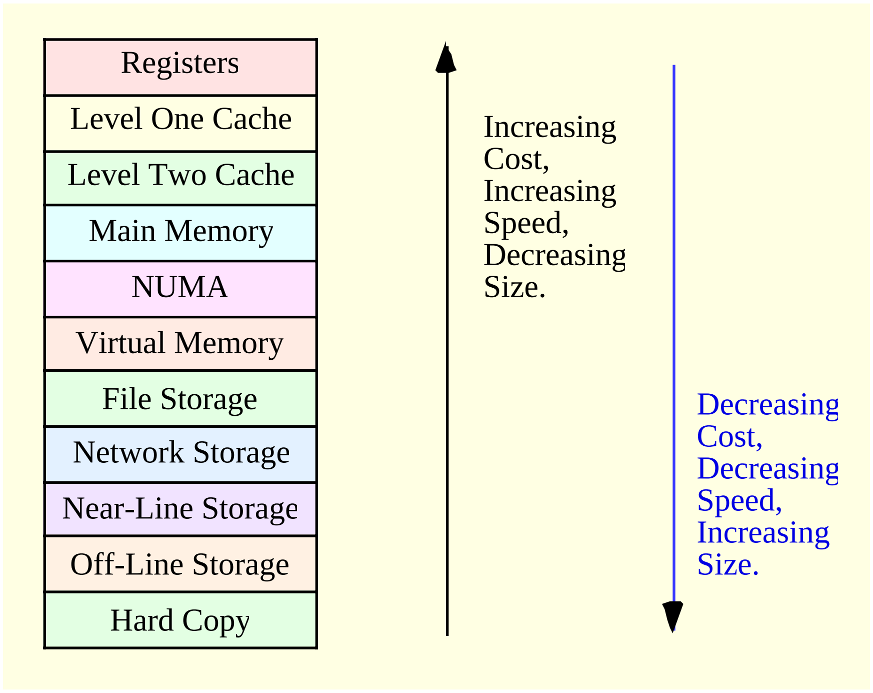

Figure 6.1 The Memory Hierarchy

At the top level of the memory hierarchy are the CPU’s general purpose registers. The registers provide the fastest access to data possible on the 80x86 CPU. The register file is also the smallest memory object in the memory hierarchy (with just eight general purpose registers available). By virtue of the fact that it is virtually impossible to add more registers to the 80x86, registers are also the most expensive memory locations.

Note that we can include FPU, MMX, SIMD, and other CPU registers in this class as well. These additional registers do not change the fact that there are a very limited number of registers and the cost per byte is quite high (figuring the cost of the CPU divided by the number of bytes of register available).

Working our way down, the Level One Cache system is the next highest performance subsystem in the memory hierarchy. On the 80x86 CPUs, the Level One Cache is provided on-chip by Intel and cannot be expanded. The size is usually quite small (typically between 4Kbytes and 32Kbytes), though much larger than the registers available on the CPU chip. Although the Level One Cache size is fixed on the CPU and you cannot expand it, the cost per byte of cache memory is much lower than that of the registers because the cache contains far more storage than is available in all the combined registers.

The Level Two Cache is present on some CPUs, on other CPUs it is the system designer's task to incorporate this cache (if it is present at all). For example, most Pentium II, III, and IV CPUs have a level two cache as part of the CPU package, but many of Intel's Celeron chips do not[1]. The Level Two Cache is generally much larger than the level one cache (e.g., 256 or 512KBytes versus 16 Kilobytes). On CPUs where Intel includes the Level Two Cache as part of the CPU package, the cache is not expandable. It is still lower cost than the Level One Cache because we amortize the cost of the CPU across all the bytes in the Level Two Cache. On systems where the Level Two Cache is external, many system designers let the end user select the cache size and upgrade the size. For economic reasons, external caches are actually more expensive than caches that are part of the CPU package, but the cost per bit at the transistor level is still equivalent to the in-package caches.

Below the Level Two Cache system in the memory hierarchy falls the main memory subsystem. This is the general-purpose, relatively low-cost memory found in most computer systems. Typically, this is DRAM or some similar inexpensive memory technology.

Below main memory is the NUMA category. NUMA, which stands for <u>N</u>on<u>U</u>niform <u>M</u>emory <u>A</u>ccess is a bit of a misnomer here. NUMA means that different types of memory have different access times. Therefore, the term NUMA is fairly descriptive of the entire memory hierarchy. In Figure 6.1a, however, we'll use the term NUMA to describe blocks of memory that are electronically similar to main memory but for one reason or another operate significantly slower than main memory. A good example is the memory on a video display card. Access to memory on video display cards is often much slower than access to main memory. Other peripheral devices that provide a block of shared memory between the CPU and the peripheral probably have similar access times as this video card example. Another example of NUMA includes certain slower memory technologies like Flash Memory that have significant slower access and transfers times than standard semiconductor RAM. We'll use the term NUMA in this chapter to describe these blocks of memory that look like main memory but run at slower speeds.

Most modern computer systems implement a Virtual Memory scheme that lets them simulate main memory using storage on a disk drive. While disks are significantly slower than main memory, the cost per bit is also significantly lower. Therefore, it is far less expensive (by three orders of magnitude) to keep some data on magnetic storage rather than in main memory. A Virtual Memory subsystem is responsible for transparently copying data between the disk and main memory as needed by a program.

File Storage also uses disk media to store program data. However, it is the program's responsibility to store and retrieve file data. In many instances, this is a bit slower than using Virtual Memory, hence the lower position in the memory hierarchy[2].

Below File Storage in the memory hierarchy comes Network Storage. At this level a program is keeping data on a different system that connects the program's system via a network. With Network Storage you can implement Virtual Memory, File Storage, and a system known as Distributed Shared Memory (where processes running on different computer systems share data in a common block of memory and communicate changes to that block across the network).

Virtual Memory, File Storage, and Network Storage are examples of so-called *on-line memory subsystems*. Memory access via these mechanism is slower than main memory access, but when a program

1. Note, by the way, that the level two cache on the Pentium CPUs is typically not on the same chip as the CPU. Instead, Intel packages a separate chip inside the box housing the Pentium CPU and wires this second chip (containing the level two cache) directly to the Pentium CPU inside the package.
2. Note, however, that in some degenerate cases Virtual Memory can be much slower than file access.

requests data from one of these memory devices, the device is ready and able to respond to the request as quickly as is physically possible. This is not true for the remaining levels in the memory hierarchy.

The Near-Line and Off-Line Storage subsystems are not immediately ready to respond to a program's request for data. An Off-Line Storage system keeps its data in electronic form (usually magnetic or optical) but on media that is not (necessarily) connected to the computer system while the program that needs the data is running. Examples of Off-Line Storage include magnetic tapes, disk cartridges, optical disks, and floppy diskettes. When a program needs data from an off-line medium, the program must stop and wait for a someone or something to mount the appropriate media on the computer system. This delay can be quite long (perhaps the computer operator decided to take a coffee break?). Near-Line Storage uses the same media as Off-Line Storage, the difference is that the system holds the media in a special robotic jukebox device that can automatically mount the desired media when some program requests it. Tapes and removable media are among the most inexpensive electronic data storage formats available. Hence, these media are great for storing large amounts of data for long time periods.

Hard Copy storage is simply a print-out (in one form or another) of some data. If a program requests some data and that data is present only in hard copy form, someone will have to manually enter the data into the computer. Paper (or other hard copy media) is probably the least expensive form of memory, at least for certain data types.

6.3 How the Memory Hierarchy Operates

The whole point of the memory hierarchy is to allow reasonably fast access to a large amount of memory. If only a little memory was necessary, we'd use fast static RAM (i.e., the stuff they make cache memory out of) for everything. If speed wasn't necessary, we'd just use low-cost dynamic RAM for everything. The whole idea of the memory hierarchy is that we can take advantage of the principle of locality of reference (see "Cache Memory" on page 153) to move often-referenced data into fast memory and leave less-used data in slower memory. Unfortunately, the selection of often-used versus lesser-used data varies over the execution of any given program. Therefore, we cannot simply place our data at various levels in the memory hierarchy and leave the data alone throughout the execution of the program. Instead, the memory subsystems need to be able to move data between themselves dynamically to adjust for changes in locality of reference during the program's execution.

Moving data between the registers and the rest of the memory hierarchy is strictly a program function. The program, of course, loads data into registers and stores register data into memory using instructions like MOV. It is strictly the programmer's or compiler's responsibility to select an instruction sequence that keeps heavily referenced data in the registers as long as possible.

The program is largely unaware of the memory hierarchy. In fact, the program only explicitly controls access to main memory and those components of the memory hierarchy at the file storage level and below (since manipulating files is a program-specific operation). In particular, cache access and virtual memory operation are generally transparent to the program. That is, access to these levels of the memory hierarchy usually take place without any intervention on the program's part. The program just accesses main memory and the hardware (and operating system) take care of the rest.

Of course, if the program really accessed main memory on each access, the program would run quite slowly since modern DRAM main memory subsystems are much slower than the CPU. The job of the cache memory subsystems (and the cache controller) is to move data between main memory and the cache so that the CPU can quickly access data in the cache. Likewise, if data is not available in main memory, but is available in slower virtual memory, the virtual memory subsystem is responsible for moving the data from hard disk to main memory (and then the caching subsystem may move the data from main memory to cache for even faster access by the CPU).

With few exceptions, most transparent memory subsystem accesses always take place between one level of the memory hierarchy and the level immediately below or above it. For example, the CPU rarely accesses main memory directly. Instead, when the CPU requests data from memory, the Level One Cache subsystem takes over. If the requested data is in the cache, then the Level One Cache subsystem returns the data and that's the end of the memory access. On the other hand if the data is not present in the level one cache, then it passes the request on down to the Level Two Cache subsystem. If the Level Two Cache subsystem has the data, it returns this data to the Level One Cache, which then returns the data to the CPU. Note that requests for this same data in the near future will come from the Level One Cache rather than the Level Two Cache since the Level One Cache now has a copy of the data.

If neither the Level One nor Level Two Cache subsystems have a copy of the data, then the memory subsystem goes to main memory to get the data. If found in main memory, then the memory subsystems copy this data to the Level Two Cache which passes it to the Level One Cache which gives it to the CPU. Once again, the data is now in the Level One Cache, so any references to this data in the near future will come from the Level One Cache.

If the data is not present in main memory, but is present in Virtual Memory on some storage device, the operating system takes over, reads the data from disk (or other devices, such as a network storage server) and places this data in main memory. Main memory then passes this data through the caches to the CPU.

Because of locality of reference, the largest percentage of memory accesses take place in the Level One Cache system. The next largest percentage of accesses occur in the Level Two Cache subsystems. The most infrequent accesses take place in Virtual Memory.

6.4 Relative Performance of Memory Subsystems

If you take another look at Figure 6.1 you'll notice that the speed of the various levels increases at the higher levels of the memory hierarchy. A good question to ask, and one we'll hope to answer in this section, is "how much faster is each successive level in the memory hierarchy?" It actually ranges from "almost no difference" to "four orders of magnitude" as you'll seem momentarily.

Registers are, unquestionably, the best place to store data you need to access quickly. Accessing a register never requires any extra time[3]. Further, instructions that access data can almost always access that data in a register. Such instructions already encode the register "address" as part of the MOD-REG-R/M byte (see "Encoding Instruction Operands" on page 290). Therefore, it never takes any extra bits in an instruction to use a register. Instructions that access memory often require extra bytes (i.e., displacement bytes) as part of the instruction encoding. This makes the instruction longer which means fewer of them can sit in the cache or in a prefetch queue. Hence, the program may run slower if it uses memory operands more often than register operands simply due to the instruction size difference.

If you read Intel's instruction timing tables, you'll see that they claim that an instruction like "mov(someVar, ecx);" is supposed to run as fast as an instruction of the form "mov(ebx, ecx);" However, if you read the fine print, you'll find that they make several assumptions about the former instruction. First, they assume that *someVar's* value is present in the level one cache memory. If it is not, then the cache controller needs to look in the level two cache, in main memory, or worse, on disk in the virtual memory subsystem. All of a sudden, this instruction that should execute in one cycle (e.g., one nanosecond on a one gigahertz processor) requires several milliseconds to execution. That's over six orders of magnitude difference, if you're counting. Now granted, locality of reference suggests that future accesses to this variable will take place in one cycle. However, if you access *someVar's* value one million times immediately thereafter, the average access time of each instruction will be two cycles because of the large amount of time needed to access *someVar* the very first time (when it was on a disk in the virtual memory system). Now granted, the likelihood that some variable will be on disk in the virtual memory subsystem is quite low. But there is a three orders of magnitude difference in performance between the level one cache subsystem and the main memory subsystem. So if the program has to bring in the data from main memory, 999 accesses later you're still paying an average cost of two cycles for the instruction that Intel's documentation claims should execute in one cycle. Note that register accesses never suffer from this problem. Hence, register accesses are much faster.

3. Okay, strictly speaking this is not true. However, we'll ignore data hazards in this discussion and assume that the programmer or compiler has scheduled their instructions properly to avoid pipeline stalls due to data hazards with register data.

The difference between the level one and level two cache systems is not so dramatic. Usually, a level two caching subsystem introduces between one and eight wait states (see “Wait States” on page 151). The difference is usually much greater, though, if the secondary cache is not packaged together with the CPU.

On a one gigahertz processor the level one cache must respond within one nanosecond if the cache operates with zero wait states (note that some processors actually introduce wait states in accesses to the level one cache, but system designers try not to do this). Accessing data in the level two cache is always slower than in the level one cache and there is always the equivalent of at least one wait state, perhaps more, when accessing data in the level two cache. The reason is quite simple – it takes the CPU time to determine that the data it is seeking is not in the L1 (level one) cache; by the time it determines that the data is not present, the memory access cycle is nearly complete and there is no time to access the data in the L2 (level two) cache.

It may also be that the L2 cache is slower than the L1 cache. This is usually done in order to make the L2 cache less expensive. Also, larger memory subsystems tend to be slower than smaller ones, and L2 caches are usually 16 to 64 times larger than the L1 cache, hence they are usually slower as well. Finally, because L2 caches are not usually on the same silicon chip as the CPU, there are some delays associated with getting data in and out of the cache. All this adds up to additional wait states when accessing data in the L2 cache. As noted above, the L2 cache can be as much as an order of magnitude slower than the L1 cache.

Another difference between the L1 and L2 caches is the amount of data the system fetches when there is an L1 cache miss. When the CPU fetches data from the L1 cache, it generally fetches (or writes) only the data requested. If you execute a "mov(al, memory);" instruction, the CPU writes only a single byte to the cache. Likewise, if you execute "mov(mem32, eax);" then the CPU reads 32 bits from the L1 cache. Access to memory subsystems below the L1 cache, however, do not work in small chucks like this. Usually, memory subsystems read blocks (or *cache lines*) of data whenever accessing lower levels of the memory hierarchy. For example, if you execute the "mov(mem32, eax);" instruction and *mem32’s* value is not in the L1 cache, the cache controller doesn’t simply read *mem32’s* value from the L2 cache (assuming it’s present there). Instead, the cache controller will actually read a block of bytes (generally 16, 32, or 64 bytes, this depends on the particular processor) from the lower memory levels. The hope is that spatial locality exists and reading a block of bytes will speed up accesses to adjacent objects in memory[4]. The bad news, however, is that the "mov(mem32, eax);" instruction doesn’t complete until the L1 cache reads the entire cache line (of 16, 32, 64, etc., bytes) from the L2 cache. Although the program may amortize the cost of reading this block of bytes over future accesses to adjacent memory locations, there is a large passage of time between the request for *mem32* and the actual completion of the "mov(mem32, eax);" instruction. This excess time is known as latency. As noted, the hope is that extra time will be worth the cost when future accesses to adjacent memory locations occur; however, if the program does not access memory objects adjacent to *mem32*, this latency is lost time.

A similar performance gulf separates the L2 cache and main memory. Main memory is typically an order of magnitude slower than the L2 cache. Again the L2 cache reads data from main memory in blocks (cache lines) to speed up access to adjacent memory elements.

There is a three to four order of magnitude difference in performance between standard DRAM and disk storage. To overcome this difference, there is usually a two to three orders of magnitude difference in size between the L2 cache and the main memory. In other words, the idea is "if the access time difference between main memory and virtual memory is two orders of magnitude greater than the difference between the L2 cache and main memory, then we’d better make sure we have two orders of magnitude more main memory than we have L2 cache." This keeps the performance loss to a reasonable level since we access virtual memory on disk two orders of magnitude less often.

We will not consider the performance of the other memory hierarchy subsystems since they are more or less under programmer control (their access is not automatic by the CPU or operating system). Hence, very little can be said about how frequently a program will access them.

4. Note that reading a block of n bytes is much faster than n reads of one byte. So this scheme is many times faster if spatial locality does occur in the program. For information about spatial locality, see “Cache Memory” on page 153.

6.5 Cache Architecture

Up to this point, cache has been this magical place that automatically stores data when we need it, perhaps fetching new data as the CPU requires it. However, a good question is "how exactly does the cache do this?" Another might be "what happens when the cache is full and the CPU is requesting additional data not in the cache?" In this section, we'll take a look at the internal cache organization and try to answer these questions along with a few others.

The basic idea behind a cache is that a program only access a small amount of data at a given time. If the cache is the same size as the typical amount of data the program access at any one given time, then we can put that data into the cache and access most of the data at a very high speed. Unfortunately, the data rarely sits in contiguous memory locations; usually, there's a few bytes here, a few bytes there, and some bytes somewhere else. In general, the data is spread out all over the address space. Therefore, the cache design has got to accommodate the fact that it must map data objects at widely varying addresses in memory.

As noted in the previous section, cache memory is not organized as a group of bytes. Instead, cache organization is usually in blocks of cache lines with each line containing some number of bytes (typically a small number that is a power of two like 16, 32, or 64), see Figure 6.2.

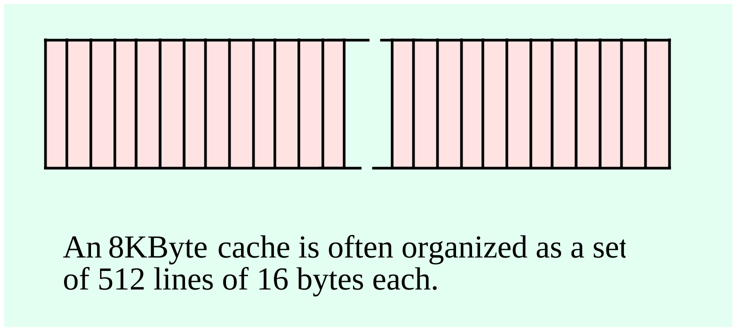
An 8KByte cache is often organized as a set of 512 lines of 16 bytes each.

Figure 6.2 Possible Organization of an 8 Kilobyte Cache

The idea of a cache system is that we can attach a different (non-contiguous) address to each of the cache lines. So cache line #0 might correspond to addresses $10000..$1000F and cache line #1 might correspond to addresses $21400..$2140F. Generally, if a cache line is n bytes long (n is usually some power of two) then that cache line will hold n bytes from main memory that fall on an n-byte boundary. In this example, the cache lines are 16 bytes long, so a cache line holds blocks of 16 bytes whose addresses fall on 16-byte boundaries in main memory (i.e., the L.O. four bits of the address of the first byte in the cache line are always zero).

When the cache controller reads a cache line from a lower level in the memory hierarchy, a good question is "where does the data go in the cache?" The most flexible cache system is the *fully associative cache*. In a fully associative cache subsystem, the caching controller can place a block of bytes in any one of the cache lines present in the cache memory. While this is a very flexible system, the flexibility is not without cost. The extra circuitry to achieve full associativity is expensive and, worse, can slow down the memory subsystem. Most L1 and L2 caches are not fully associative for this reason.

At the other extreme is the *direct mapped cache* (also known as the *one-way set associative cache*). In a direct mapped cache, a block of main memory is always loaded into the same cache line in the cache. Generally, some number of bits in the main memory address select the cache line. For example, Figure 6.3 shows how the cache controller could select a cache line for an 8 Kilobyte cache with 16-byte cache lines and a 32-bit main memory address. Since there are 512 cache lines, this example uses bits four through twelve to select one of the cache lines (bits zero through three select a particular byte within the 16-byte cache line). The direct-mapped cache scheme is very easy to implement. Extracting nine (or some other number of) bits from the address and using this as an index into the array of cache lines is trivial and fast. However, direct-mapped caches to suffer from some other problems.

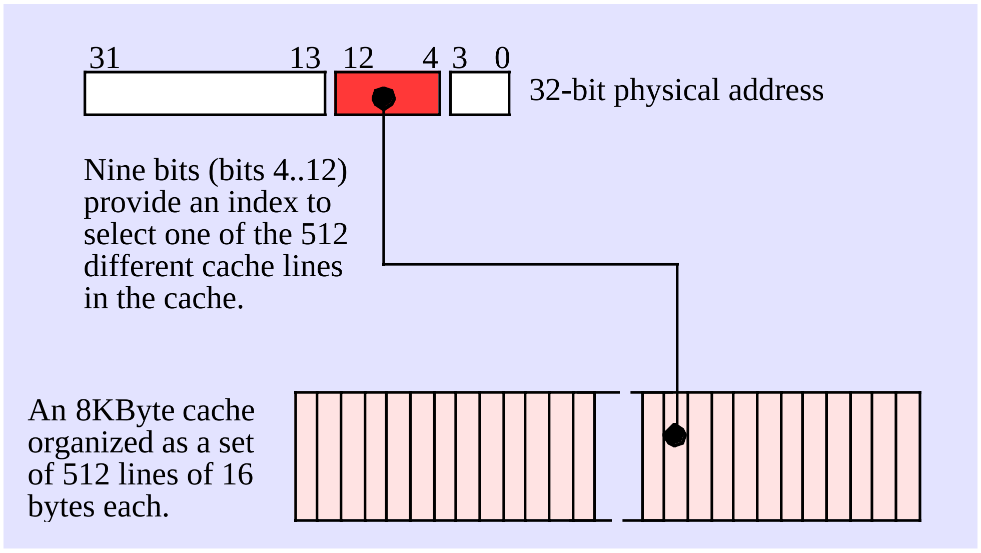
31
13
12
4
3
0
32-bit physical address
Nine bits (bits 4..12) provide an index to select one of the 512 different cache lines in the cache.
An 8KByte cache organized as a set of 512 lines of 16 bytes each.

Figure 6.3 Selecting a Cache Line in a Direct-mapped Cache

Perhaps the biggest problem with a direct-mapped cache is that it may not make effective use of all the cache memory. For example, the cache scheme in Figure 6.3 maps address zero to cache line #0. It also maps address $2000 (8K), $4000 (16K), $6000 (24K), $8000 (32K), and, in fact, it maps every address that is an even multiple of eight kilobytes to cache line #0. This means that if a program is constantly accessing data at addresses that are even multiples of 8K and not accessing any other locations, the system will only use cache line #0, leaving all the other cache lines unused. Each time the CPU requests data at an address that is not at an address within cache line #0, the CPU will have to go down to a lower level in the memory hierarchy to access the data. In this pathological case, the cache is effectively limited to the size of one cache line. Had we used a fully associative cache organization, each access (up to 512 cache lines' worth) could have their own cache line, thus improving performance.

If a fully associative cache organization is too complex, expensive, and slow to implement, but a direct-mapped cache organization isn't as good as we'd like, one might ask if there is a compromise that gives us more capability that a direct-mapped approach without all the complexity of a fully associative cache. The answer is yes, we can create an *n-way set associative cache* which is a compromise between these two extremes. The idea here is to break up the cache into sets of cache lines. The CPU selects a particular set using some subset of the address bits, just as for direct-mapping. Within each set there are n cache lines. The caching controller uses a fully associative mapping algorithm to select one of the n cache lines within the set.

As an example, an 8 kilobyte two-way set associative cache subsystem with 16-byte cache lines organizes the cache as a set of 256 sets with each set containing two cache lines ("two-way" means each set contains two cache lines). Eight bits from the memory address select one of these 256 different sets. Then the cache controller can map the block of bytes to either cache line within the set (see Figure 6.4). The advantage of a two-way set associative cache over a direct mapped cache is that you can have two accesses on 8 Kilobyte boundaries (using the current example) and still get different cache lines for both accesses. However, once you attempt to access a third memory location at an address that is an even multiple of eight kilobytes you will have a conflict.

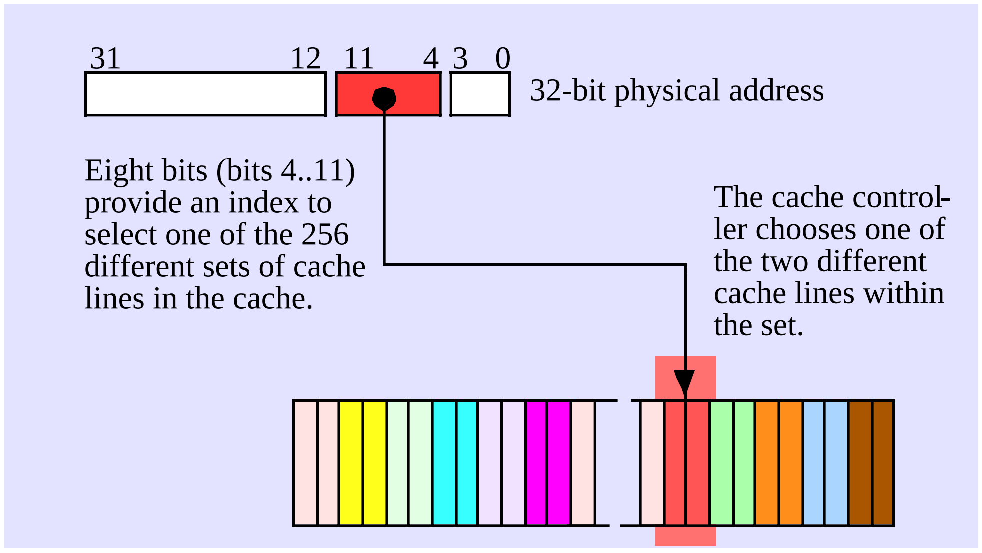
31
12
11
4
3
0
32-bit physical address
Eight bits (bits 4..11) provide an index to select one of the 256 different sets of cache lines in the cache.
The cache controller chooses one of the two different cache lines within the set.

Figure 6.4 A Two-Way Set Associative Cache

A two-way set associative cache is much better than a direct-mapped cache and considerably less complex than a fully associative cache. However, if you're still getting too many conflicts, you might consider using a four-way set associative cache. A four-way set associative cache puts four associative cache lines in each block. In the current 8K cache example, a four-way set associative example would have 128 sets with each set containing four cache lines. This would allow up to four accesses to an address that is an even multiple of eight kilobytes before a conflict would occur.

Obviously, we can create an arbitrary m-way set associative cache (well, m does have to be a power of two). However, if m is equal to n, where n is the number of cache lines, then you've got a fully associative cache with all the attendant problems (complexity and speed). Most cache designs are direct-mapped, two-way set associative, or four-way set associative. The 80x86 family CPUs use all three (depending on the CPU and cache).

Although this section has made direct-mapped cache look bad, they are, in fact, very effective for many types of data. In particular, they are very good for data that you access in a sequential rather than random fashion. Since the CPU typically executes instructions in a sequential fashion, instructions are a good thing to put into a direct-mapped cache. Data access is probably a bit more random access, so a two-way or four-way set associative cache probably makes a better choice.

Because access to data and instructions is different, many CPU designers will use separate caches for instructions and data. For example, the CPU designer could choose to implement an 8K instruction cache and an 8K data cache rather than a 16K unified cache. The advantage is that the CPU designer could choose a more appropriate caching scheme for instructions versus data. The drawback is that the two caches are now each half the size of a unified cache and you may get fewer cache misses from a unified cache. The choice of an appropriate cache organization is a difficult one and can only be made after analyzing lots of running programs on the target processor. How to choose an appropriate cache format is beyond the scope of this text, just be aware that it's not an easy choice you can make by reading some textbook.

Thus far, we've answered the question "where do we put a block of data when we read it into the cache?" An equally important question we ignored until now is "what happens if a cache line isn't available when we need to read data from memory?" Clearly, if all the lines in a set of cache lines contain data, we're going to have to replace one of these lines with the new data. The question is, "how do we choose the cache line to replace?"

For a direct-mapped (one-way set associative) cache architecture, the answer is trivial. We replace exactly the block that the memory data maps to in the cache. The cache controller replaces whatever data was formerly in the cache line with the new data. Any reference to the old data will result in a cache miss and the cache controller will have to bring that data into the cache replacing whatever data is in that block at that time.

For a two-way set associative cache, the replacement algorithm is a bit more complex. Whenever the CPU references a memory location, the cache controller uses some number of the address bits to select the set that should contain the cache line. Using some fancy circuity, the caching controller determines if the data is already present in one of the two cache lines in the set. If not, then the CPU has to bring the data in from memory. Since the main memory data can go into either cache line, somehow the controller has to pick one or the other. If either (or both) cache lines are currently unused, the selection is trivial: pick an unused cache line. If both cache lines are currently in use, then the cache controller must pick one of the cache lines and replace its data with the new data. Ideally, we'd like to keep the cache line that will be referenced first (that is, we want to replace the one whose next reference is later in time). Unfortunately, neither the cache controller nor the CPU is omniscient, they cannot predict which is the best one to replace. However, remember the principle of temporal locality (see "Cache Memory" on page 153): if a memory location has been referenced recently, it is likely to be referenced again in the very near future. A corollary to this is "if a memory location has not been accessed in a while, it is likely to be a long time before the CPU accesses it again." Therefore, a good replacement policy that many caching controllers use is the "least recently used" or LRU algorithm. The idea is to pick the cache line that was not most frequently accessed and replace that cache line with the new data. An LRU policy is fairly easy to implement in a two-way set associative cache system. All you need is a bit that is set to zero whenever the CPU accessing one cache line and set it to one when you access the other cache line. This bit will indicate which cache line to replace when a replacement is necessary. For four-way (and greater) set associative caches, maintaining the LRU information is a bit more difficult, which is one of the reasons the circuitry for such caches is more complex. Other possible replacement policies include First-in, First-out[5] (FIFO) and random. These are easier to implement than LRU, but they have their own problems.

The replacement policies for four-way and n-way set associative caches are roughly the same as for two-way set associative caches. The major difference is in the complexity of the circuit needed to implement the replacement policy (see the comments on LRU in the previous paragraph).

Another problem we've overlooked in this discussion on caches is "what happens when the CPU writes data to memory?" The simple answer is trivial, the CPU writes the data to the cache. However, what happens when the cache line containing this data is replaced by incoming data? If the contents of the cache line is not written back to main memory, then the data that was written will be lost. The next time the CPU reads that data, it will fetch the original data values from main memory and the value written is lost.

Clearly any data written to the cache must ultimately be written to main memory as well. There are two common write policies that caches use: write-back and write-through. Interestingly enough, it is sometimes possible to set the write policy under software control; these aren't hardwired into the cache controller like most of the rest of the cache design. However, don't get your hopes up. Generally the CPU only allows the BIOS or operating system to set the cache write policy, your applications don't get to mess with this. However, if you're the one writing the operating system...

The write-through policy states that any time data is written to the cache, the cache immediately turns around and writes a copy of that cache line to main memory. Note that the CPU does not have to halt while the cache controller writes the data to memory. So unless the CPU needs to access main memory shortly after the write occurs, this writing takes place in parallel with the execution of the program. Still, writing a cache line to memory takes some time and it is likely that the CPU (or some CPU in a multiprocessor system) will want to access main memory during this time, so the write-through policy may not be a high performance solution to the problem. Worse, suppose the CPU reads and writes the value in a memory location several times in succession. With a write-through policy in place the CPU will saturate the bus with cache line writes and this will have a very negative impact on the program's performance. On the positive side, the write-through policy does update main memory with the new value as rapidly as possible. So if two different CPUs are communicating through the use of shared memory, the write-through policy is probably better because the second CPU will see the change to memory as rapidly as possible when using this policy.

5. This policy does exhibit some anomalies. These problems are beyond the scope of this chapter, but a good text on architecture or operating systems will discuss the problems with the FIFO replacement policy.

The second common cache write policy is the write-back policy. In this mode, writes to the cache are not immediately written to main memory; instead, the cache controller updates memory at a later time. This scheme tends to be higher performance because several writes to the same variable (or cache line) only update the cache line, they do not generate multiple writes to main memory.

Of course, at some point the cache controller must write the data in cache to memory. To determine which cache lines must be written back to main memory, the cache controller usually maintains a *dirty bit* with each cache line. The cache system sets this bit whenever it writes data to the cache. At some later time the cache controller checks this dirty bit to determine if it must write the cache line to memory. Of course, whenever the cache controller replaces a cache line with other data from memory, it must first write that cache line to memory if the dirty bit is set. Note that this increases the latency time when replacing a cache line. If the cache controller were able to write dirty cache lines to main memory while no other bus access was occurring, the system could reduce this latency during cache line replacement.

A cache subsystem is not a panacea for slow memory access. In order for a cache system to be effective the software must exhibit locality of reference. If a program accesses memory in a random fashion (or in a fashion guaranteed to exploit the caching controller's weaknesses) then the caching subsystem will actually cause a big performance drop. Fortunately, real-world programs do exhibit locality of reference, so most programs will benefit from the presence of a cache in the memory subsystem.

Another feature to the cache subsystem on modern 80x86 CPUs is that the cache automatically handles many misaligned data references. As you may recall from an earlier chapter, there is a penalty for accesses larger data objects (words or dwords) at an address that is not an even multiple of that object's size. As it turns out, by providing some fancy logic, Intel's designers have eliminated this penalty as long as the data access is completely within a cache line. Therefore, accessing a word or double word at an odd address does not incur a performance penalty as long as the entire object lies within the same cache line. However, if the object crosses a cache line, then there will be a performance penalty for the memory access.

6.6 Virtual Memory, Protection, and Paging

In a modern operating system such as Linux or Windows, it is very common to have several different programs running concurrently in memory. This presents several problems. First, how do you keep the programs from interfering with one another? Second, if one program expects to load into memory at address $1000 and a second program also expects to load into memory at address $1000, how can you load and execute both programs at the same time? One last question we might ask is what happens if our computer has 64 megabytes of memory and we decide to load and execute three different applications, two of which require 32 megabytes and one that requires 16 megabytes (not to mention the memory the operating system requires for its own purposes)? The answer to all these questions lies in the virtual memory subsystem the 80x86 processors support[6].

Virtual memory on the 80x86 gives each process its own 32-bit address space[7]. This means that address $1000 in one program is physically different than address $1000 in a separate program. The 80x86 achieves this sleight of hand by using *paging* to remap *virtual addresses* within one program to different *physical addresses* in memory. A virtual address in the memory address that the program uses. A physical address is the bit pattern than actually appears on the CPU's address bus. The two don't have to be the same (and usually, they aren't). For example, program #1's virtual address $1000 might actually correspond to physical address $215000 while program #2's virtual address $1000 might correspond to physical memory address $300000. How can the CPU do this? Easy, by using *paging*.

6. Actually, virtual memory is really only supported by the 80386 and later processors. We'll ignore this issue here since most people have an 80386 or later processor.
7. Strictly speaking, you actually get a 36-bit address space on Pentium Pro and later processors, but Windows and Linux limits you to 32-bits so we'll use that limitation here.

The concept behind paging is quite simple. First, you break up memory into blocks of bytes called pages. A page in main memory is comparable to a cache line in a cache subsystem, although pages are usually much larger than cache lines. For example, the 80x86 CPUs use a page size of 4,096 bytes.

After breaking up memory into pages, you use a lookup table to translate the H.O. bits of a virtual address to select a page; you use the L.O. bits of the virtual address as an index into the page. For example, with a 4,096-byte page, you'd use the L.O. 12 bits of the virtual address as the offset within the page in physical memory. The upper 20 bits of the address you would use as an index into a lookup table that returns the actual upper 20 bits of the physical address (see Figure 6.5).

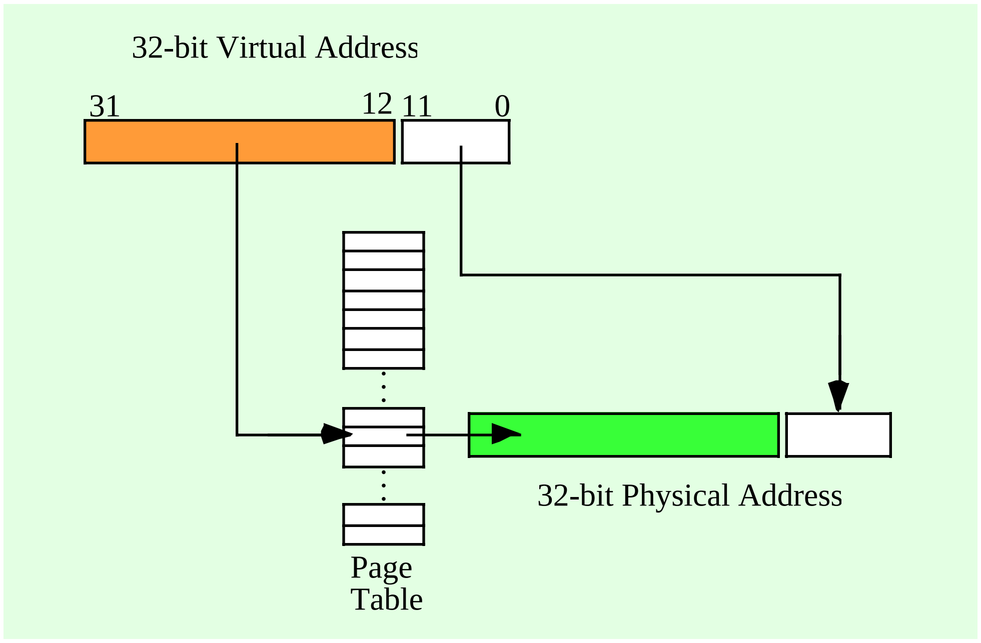

Figure 6.5 Translating a Virtual Address to a Physical Address

Of course, a 20-bit index into the page table would require over one million entries in the page table. If each entry is 32 bits (20 bits for the offset plus 12 bits for other purposes), then the page table would be four megabytes long. This would be larger than most of the programs that would run in memory! However, using what is known as a multi-level page table, it is very easy to create a page table that is only 8 kilobytes long for most small programs. The details are unimportant here, just rest assured that you don't need a four megabyte page table unless your program consumes the entire four gigabyte address space.

If you study Figure 6.5 for a few moments, you'll probably discover one problem with using a page table – it requires two memory accesses in order to access an address in memory: one access to fetch a value from the page table and one access to read or write the desired memory location. To prevent cluttering the data (or instruction) cache with page table entries (thus increasing the number of cache misses), the page table uses its own cache known as the *Translation Lookaside Buffer*, or TLB. This cache typically has 32 entries on a Pentium family processor. This provides a sufficient lookup capability to handle 128 kilobytes of memory (32 pages) without a miss. Since a program typically works with less data than this at any given time, most page table accesses come from the cache rather than main memory.

As noted, each entry in the page table is 32 bits even though the system really only needs 20 bits to remap the addresses. Intel uses some of the remaining 12 bits to provide some memory protection information. For example, one bit marks whether a page is read/write or read-only. Another bit determines if you can execute code on that page. Some bits determine if the application can access that page or if only the operating system can do so. Some bits determine if the page is "dirty" (that is, if the CPU has written to the page) and whether the CPU has accessed the page recently (these bits have the same meaning as for cache lines). Another bit determines whether the page is actually present in physical memory or if it's stored on secondary storage somewhere. Note that your applications do not have access to the page table, and therefore they cannot modify these bits. However, Windows does provide some functions you can call if you want to change certain bits in the page table (e.g., Windows will allow you to set a page to read-only if you want to do so). Linux users also have some memory mapping functions they can call to play around with the access bits.

Beyond remapping memory so multiple programs can coexist in memory even though they access the same virtual addresses, paging also provides a mechanism whereby the operating system can move infrequently used pages to secondary storage (i.e., a disk drive). Just as locality of reference applies to cache lines, it applies to pages in memory as well. At any one given time a program will only access a small percentage of the pages in memory that contain data and code (this set of pages is known as the *working set*). While this working set of pages varies (slowly) over time, for a reasonable time period the working set remains constant. Therefore, there is little need to have the remainder of the program in memory consuming valuable physical memory that some other process could be using. If the operating system can save those (currently unused) pages to disk, the physical memory they consume would be available for other programs that need it.

Of course, the problem with moving data out of physical memory is that sooner or later the program might actually need it. If you attempt to access a page of memory and the page table bit tells the MMU (memory management unit) that this page is not present in physical memory, then the CPU interrupts the program and passes control to the operating system. The operating system analyzes the memory access request and reads the corresponding page of data from the disk drive to some available page in memory. The process is nearly identical to that used by a fully associative cache subsystem except, of course, accessing the disk is much slower than main memory. In fact, you can think of main memory as a fully associative write-back cache with 4,096 byte cache lines that caches the data on the disk drive. Placement and replacement policies and other issues are very similar to those we've discussed for caches. Discussing how the virtual memory subsystem works beyond equating it to a cache is will beyond the scope of this text. If you're interested, any decent text on operating system design will explain how a virtual memory subsystem swaps pages between main memory and the disk. Our main goal here is to realize that this process takes place in operating systems like Linux or Windows and that accessing the disk is very slow.

One important issue resulting from the fact that each program as a separate page table and the programs themselves don't have access to the page table is that programs cannot interfere with the operation of other programs by overwriting those other program's data (assuming, of course, that the operating system is properly written). Further, if your program crashes by overwriting itself, it cannot crash other programs at the same time. This is a big benefit of a paging memory system.

Note that if two programs want to cooperate and share data, they can do so. All they've got to do is to tell the operating system that they want to share some blocks of memory. The operating system will map their corresponding virtual addresses (of the shared memory area) to the same physical addresses in memory. Under Windows, you can achieve this use *memory mapped files*; see the operating system documentation for more details. Linux also supports memory mapped files as well as some special shared memory operations; again, see the OS documentation for more details.

6.7 Thrashing

Thrashing is a degenerate case that occurs when there is insufficient memory at one level in the memory hierarchy to properly contain the working set required by the upper levels of the memory hierarchy. This can result in the overall performance of the system dropping to the speed of a lower level in the memory hierarchy. Therefore, thrashing can quickly reduce the performance of the system to the speed of main memory or, worse yet, the speed of the disk drive.

There are two primary causes of thrashing: (1) insufficient memory at a given level in the memory hierarchy, and (2) the program does not exhibit locality of reference. If there is insufficient memory to hold a working set of pages or cache lines, then the memory system is constantly replacing one block (cache line or page) with another. As a result, the system winds up operating at the speed of the slower memory in the hierarchy. A common example occurs with virtual memory. A user may have several applications running at the same time and the sum total of these programs' working sets is greater than all of physical memory available to the program. As a result, as the operating system switches between the applications it has to copy each application's data to and from disk and it may also have to copy the code from disk to memory. Since a context switch between programs is often much faster than retrieving data from the disk, this slows the programs down by a tremendous factor since thrashing slows the context switch down to the speed of swapping the applications to and from disk.

If the program does not exhibit locality of reference and the lower memory subsystems are not fully associative, then thrashing can occur even if there is free memory at the current level in the memory hierarchy. For example, suppose an eight kilobyte L1 caching system uses a direct-mapped cache with 16-byte cache lines (i.e., 512 cache lines). If a program references data objects 8K apart on each access then the system will have to replace the same line in the cache over and over again with each access. This occurs even though the other 511 cache lines are currently unused.

If insufficient memory is the cause of thrashing, an easy solution is to add more memory (if possible, it is rather hard to add more L1 cache when the cache is on the same chip as the processor). Another alternative is to run fewer processes concurrently or modify the program so that it references less memory over a given time period. If lack of locality of reference is causing the problem, then you should restructure your program and its data structures to make references local to one another.

6.8 NUMA and Peripheral Devices

Although most of the RAM memory in a system is based on high-speed DRAM interfaced directly to the processor's bus, not all memory is connected to the CPU in this manner. Sometimes a large block of RAM is part of a peripheral device and you communicate with that device by writing data to the RAM on the peripheral. Video display cards are probably the most common example, but some network interface cards and USB controllers also work this way (as well as other peripherals). Unfortunately, the access time to the RAM on these peripheral devices is often much slower than access to normal memory. We'll call such access NUMA[8] access to indicate that access to such memory isn't uniform (that is, not all memory locations have the same access times). In this section we'll use the video card as an example, although NUMA performance applies to other devices and memory technologies as well.

A typical video card interfaces to the CPU via the AGP or PCI (or much worse, ISA) bus inside the computer system. The PCI bus nominally runs at 33 MHz and is capable of transferring four bytes per bus cycle. In burst mode, a video controller card, therefore, is capable of transferring 132 megabytes per second (though few would ever come close to achieving this for technical reasons). Now compare this with main memory access. Main memory usually connects directly to the CPU's bus and modern CPUs have a 400 MHz 64-bit wide bus. Technically (if memory were fast enough), the CPU's bus could transfer 800 MBytes/sec. between memory and the CPU. This is six times faster than transferring data across the PCI bus. Game programmers long ago discovered that it's much faster to manipulate a copy of the screen data in main memory and only copy that data to the video display memory when a vertical retrace occurs (about 60 times/sec.). This mechanism is much faster than writing directly to the video memory every time you want to make a change.

Unlike caches and the virtual memory subsystem that operate in a transparent fashion, programs that write to NUMA devices must be aware of this and minimize the accesses whenever possible (e.g., by using an off-screen bitmap to hold temporary results). If you're actually storing and retrieving data on a NUMA device, like a Flash memory card, then you must explicitly cache the data yourself. Later in this text you'll learn about hash tables and searching. Those techniques will help you create your own caching system for NUMA devices.

8. Remember, NUMA stands for NonUniform Memory Access.

6.9 Segmentation

Segmentation is another memory management scheme, like paging, that provides memory protection and virtual memory capabilities. Linux and Windows do not support the use of segments, nor does HLA provide any instructions that let you manipulate segment registers or use segment override prefixes on an instruction[9]. These 32-bit operating system employ the flat memory model that, essentially, ignore segments on the 80x86. Furthermore, the remainder of this text also ignores segmentation. What this means is that you don't really need to know anything about segmentation in order to write assembly language programs that run under modern OSes. However, it's unthinkable to write a book on 80x86 assembly language programming that doesn't at least mention segmentation. Hence this section.

The basic idea behind the segmentation model is that memory is managed using a set of segments. Each segment is, essentially, its own address space. A segment consists of two components: a base address that contains the address of some physical memory location and a length value that specifies the length of the segment. A segmented address also consists of two components: a segment selector and an offset into the segment. The segment selector specifies the segment to use (that is, the base address and length values) while the offset component specifies the offset from the base address for the actual memory access. The physical address of the actual memory location is the sum of the offset and the base address values. If the offset exceeds the length of the segment, the system generates a protection violation.

Segmentation on the 80x86 got a (deservedly) bad name back in the days of the 8086, 8088, and 80286 processors. The problem back then is that the offset into the segment was only a 16-bit value, effectively limiting segments to 64 kilobytes in length. By creating multiple segments in memory it was possible to address more than 64K within a single program; however, it was a major pain to do so, especially if a single data object exceeded 64 kilobytes in length. With the advent of the 80386, Intel solved this problem (and others) with their segmentation model. By then, however, the damage had been done; segmentation had developed a really bad name that it still bears to this day.

Segments are an especially powerful memory management system when a program needs to manipulate different variable sized objects and the program cannot determine the size of the objects before run time. For example, suppose you want to manipulate several different files using the memory mapped file scheme. Under Windows or Linux, which don't support segmentation, you have to specify the maximum size of the file before you map it into memory. If you don't do this, then the operating system can't leave sufficient space at the end of the first file in memory before the second file starts. On the other hand, if the operating system supported segmentation, it could easily return segmented pointers to these two memory mapped files, each in their own logical address space. This would allow the files to grow to the size of the maximum offset within a segment (or the maximum file size, whichever is smaller). Likewise, if two programs wanted to share some common data, a segmented system could allow the two programs to put the shared data in a segment. This would allow both programs to reference objects in the shared area using like-valued pointer (offset) values. This makes is easier to pass pointer data (within the shared segment) between the two programs, a very difficult thing to do when using a flat memory model without segmentation as Linux and Windows currently do.

One of the more interesting features of the 80386 and later processors is the fact that Intel combined both segmentation and paging in the same memory management unit. Prior to the 80386 most real-world CPUs used paging or segmentation but not both. The 80386 processor merged both of these memory management mechanisms into the same chip, offering the advantages of both systems on a single chip. Unfortunately, most 32-bit operating systems (e.g., Linux and Windows) fail to take advantage of segmentation so this feature goes wasted on the chip.

6.10 Segments and HLA

Although HLA creates programs use the flat memory model under Windows and Linux[10], HLA does provide limited support for segments in your code. However, HLA's (and the operating system's) segments

9. Though you could easily create macros to do this.

are not the same thing as 80x86 segments; HLA segments are a logical organization of memory that has very little to do with segmentation on the 80x86. HLA's segments provide a simple way to organize variables and other objects in memory.

Logically, a segment is a block of memory where you place related objects. By default, HLA supports five different segments: a segment that holds machine instructions, a read-only segment that holds constant objects that HLA creates, a readonly segment that holds values you declare in the READONLY section, a data segment that holds variables and other objects you declare in the STATIC section, and a "BSS" section that holds uninitialized variables you declare in the STORAGE section[11].

Normally you are completely unaware of the fact that HLA creates these segments in memory. The use of these segments is automatic and generally transparent to your HLA programs. In a few cases, however, you may need access to this segment information. For example, when linking your HLA programs with high level languages like C/C++ or Delphi you may need to tell HLA to use different names for the five segments it create (as imposed by the high level language). By default, HLA uses the following segment names for its five segments under Windows:

- _TEXT for the code segment (corresponds to the ".code" segment).
- _DATA for the STATIC section (corresponds to the ".data" segment).
- _BSS for the STORAGE section (corresponds to the ".bss" segment).
- "CONST" for the HLA constant segment (corresponds to the ".edata" segment).
- "readonly" for the HLA READONLY segment (this is not a standardized segment name).

The "_TEXT", "_DATA", "_BSS", and "CONST" segment names are quite standard under Windows. Most common compilers that generate Win32 code use these segment names for the code, data, uninitialized, and constant data sections. There does not seem to be a common segment that high level language compilers use for read-only data (other than CONST), so HLA creates a separate specifically for this purpose: the "readonly" segment where HLA puts the objects you declare in the READONLY section.

Here's the typical names under Linux:

- .text for the code segment.
- .data for the STATIC section.
- .bss for the STORAGE section.
- .rodata for the HLA constant segment.
- .rodata for the HLA READONLY segment.

Examples of objects HLA puts in the "CONST" segment include string literal constants for string variables, constants HLA emits for extended syntax forms of the MUL, IMUL, DIV, IDIV, BOUNDS, and other instructions, floating point constants that HLA automatically emits (e.g., for the "fld(1.234);" instruction) and so on. Generally, you do not explicitly declare values that wind up in this section (other than through the use of one of the aforementioned instructions).

6.10.1 Renaming Segments Under Windows

Under Windows, HLA provides special directives that let you change the default names for the default segments. Although "_TEXT", "_DATA", "_BSS" and "CONST" are very standard names, some compilers may use different names and expect HLA to put its code and data in those different segments. The "readonly" segment is definitely non-standard, some compilers may not allow you to use it (indeed, some compilers may not even allow read-only segments in memory). Should you encounter a language that wants different segment names or doesn't allow readonly segments, you can tell HLA to use a different segment name or to map the read-only segments to the static data segment. Here are the directives to achieve this:

```
#code( "codeSegmentName", "alignment", "class" )
```

10. When this was first written, segments were not yet functional under Linux. This may have changed by the time you read this. They are, however, fully functional under Windows.
11. In theory, there is also a stack and a heap segment. However, the linker, not HLA, defines and allocates these two segments. You cannot explicitly declare static objects in these two segments during compilation.

```
#static( "dataSegmentName", "alignment", "class" )
#storage( "bssSegmentName", "alignment", "class" )
#readonly( "readOnlySegmentName", "alignment", "class" )
#const( "constSegmentName", "alignment", "class" )
```

The #code directive tells HLA to rename the code segment ("_TEXT") or use different alignment or classification options. The #static directive renames the data segment ("_DATA", the segment the STATIC section uses). The #storage directive renames the uninitialized data segment ("_BSS", the segment the STORAGE section uses). The #readonly directive renames the "readonly" segment (where HLA places data you declare in the READONLY section). Finally, the #const directive renames HLA's "CONST" segments (where HLA places constants that it emits internally).

Each of these directives contains three string expression operands. The first string operand specifies the name of the segment. The second string specifies the segment alignment; we'll return to a discussion of this operand in a moment. The third operand is the segment class; the linker uses this name to combine segments that have different names into a single memory block. Generally, the class name is some variant of the segment name, but this is not necessarily the case (e.g., the standard class name for the "_TEXT" segment is "CODE").

The alignment operand must be a string that contains one of the following identifiers: "byte", "word", "dword", "para", or "page". HLA will only allow a string constant containing one of these five strings. The alignment option specifies the boundary on which the linker will start a segment. This option is only meaningful if you combine two different segments by using the same string for the class parameter. The linker combines two segments by concatenating them in memory. When the linker combines the segments, it makes sure that the concatenated segments start on the boundary the alignment operand specifies. A "byte" alignment means that the segment can start at an arbitrary byte boundary. The "word" and "dword" alignment options tell the linker that the segment must start on a word or double word boundary (respectively). The "para" alignment option tells the linker to start the segment on a paragraph (16-byte) boundary. The "page" option tells the linker to align the segment on a 256-byte page boundary (this has nothing to do with 4K pages). Most systems expect paragraph alignment, so the most common option here is "para"[12].

By default, the linker will start each segment in memory on a 4K MMU page boundary. Therefore, if each segment in an HLA program uses only one byte, that program will consume at least 20K because each segment in memory will start on a different 4K boundary. This is why a simple "Hello World" application consumes so much memory – the five default HLA segments each consume 4K of the memory space whether or not the segments actually have 4K of data. The program isn't really 20K long, it's just spread out over the 20K. As you add more code to the "Hello World" program, you'll notice that the executable file doesn't grow in size until you reach some magic point. Then the program jumps in size by increments of 4K (each time a segment's length crosses a 4K boundary, the program grows in length by 4K). If you want the shortest possible executable file, you can tell HLA to combine all the segments into a single segment. However, saving 8K, 12K, or even 16K of data is hardly useful on modern computer systems. Combining segments only saves a significant percentage of the program's size on very tiny programs, so it's not worth the effort for most real applications.

To combine two segments you use the same name for the third parameter in the #code, #data, #static, #readonly, and #const directives. For example, if you want to combine the "CONST" and "readonly" segments into a single memory segment, you can do so with the following two statements (this is actually the default definition):

```
#readonly( "readonly", "para", "CONST" )
#const( "CONST", "para", "CONST" )
```

By using the same class names but different segment names you tell the linker to combine these two segments in memory. Of course, you can also combine the two segments by giving them the same segment name, e.g.,

```
#readonly( "readonly", "para", "readonly" )
```

12. In fact, MASM requires PARA alignment for the standard segment names. You may only change the alignment if you specify different segment names.

```
#const( "readonly", "para", "readonly" )     // This is a bad idea, see below.
```

If the particular language you are using doesn't support read-only segments, you should map the "readonly" and "CONST" segments to the "_TEXT" (or equivalent) segment using the "CODE" combine class parameter.

The segment renaming directives do not check the syntax of the strings you specify for the segment name and class fields. These should be legal MASM identifiers and should not be MASM keywords. Generally, legal HLA identifiers work just fine here (unless, of course, you just happen to pick a MASM reserved word). If you specify a syntactically incorrect segment name or class name, HLA will not complain until it attempts to assemble its output file with MASM.

You may only rename the HLA segments once and these directives must appear before the UNIT or PROGRAM statements in an HLA source file. HLA does not allow you to change the name of one of these segments after it has emitted any code for a specific segment. Since HLA emits segment declarations in response to a UNIT or PROGRAM statement, you must do any segment renaming prior to these statements in an HLA source file; i.e., these directives will typically be the very first statements in a source file.

Here are the default segment names, alignments, and class values that HLA uses:

```
#code( "_TEXT", "para", "CODE" )
#static( "_DATA", "para", "DATA" )
#storage( "_BSS", "para", "BSS" )
#const( "CONST", "para", "CONST" )
#readonly( "readonly", "para", "CONST" )
```

If you use the MASM-defined names "_TEXT", "_DATA", "_BSS", or "CONST" you must provide the alignment and class parameters given above or MASM will complain when it compile's HLA's output.

6.11 User Defined Segments in HLA (Windows Only)

In addition to the five standard segments, HLA lets you declare your own segments in your assembly programs. Like the five standard segments, you should not confuse HLA segments with 80x86 segments. You do not use the 80x86 segment registers to access data in user-defined segments. Instead, user segments exist as a logical entity to group a set of related objects into the same physical block of memory. In this section we'll take a look at why you would want to use segments and how you declare them in HLA.

It should come as no surprise that when you declare two variables in adjacent statements in a declaration section (e.g., STATIC) that HLA allocates those objects in adjacent memory locations. What may be surprising is that HLA will probably not allocate two variables adjacently in memory if you declare those variables in two adjacent declaration selections. E.g., HLA will allocate *i* and *j* below in adjacent memory locations, but it probably will not allocate *j* and *k* in adjacent memory locations:

```
static
    i:uns32;
    j:int32;

storage
    k:dword;
```

The reason *k* does not immediately follow *j* in memory is because k is in the "_BSS" segment while i and j are in the "_DATA" segment. Since segments typically start on 4K boundaries, there may be a huge gap between *j* and *k*, assuming that the "_BSS" segment follows the "_DATA" segment in memory (and it may not).

Another somewhat surprising result is that HLA (and MASM and the linker) will combine declarations from declaration sections with the same segment name, even if those declarations are not adjacent. Consider the following code:

```
static
    i:uns32;
    j:int32;

storage
    k:dword;

static
    m:real32;
```

Although *j* and *k* probably won't occupy adjacent memory locations, nor will *k* and *m*, it is quite possible for *j* and *m* to occupy adjacent memory locations since HLA places both declarations in the "_DATA" segment. There is no requirement for HLA to allocate *m* immediately after *j*, but HLA will allocate both objects in the same block of physical memory. If you need allocate two variables in adjacent memory locations, or one variable must appear at a lower address than another in memory, you must allocate both objects in the same (physical) declaration sequence. I.e., *i* and *j* (in the declarations above) will be allocated in adjacent memory locations with *i* at the lower address. HLA allocates *m* in the same segment as *i* and *j*, but there's no guarantee that m will appear at a higher or lower address than *i* and *j*.

In addition to the five standard segments, HLA lets you define your own memory segments. You use the SEGMENT declaration statement to accomplish this. The SEGMENT statement takes the following form:

```
segment segName( "alignment", "class" );
    << Declarations >>
```

You would use this declaration anywhere a STATIC, READONLY, or STORAGE, declaration section is legal[13]. Anything legal after a STATIC keyword is legal after the SEGMENT declaration.

The *segName* field in the declaration above is the name you're going to give this segment. You should choose a unique name and it probably shouldn't be *_TEXT*, *_BSS*, *_DATA*, *readonly*, or *CONST* (HLA doesn't prevent the use of these segment names; however, there is little purpose to using most of them since you can create objects in most of these segments using the standard declaration sections). This segment name will automatically be a public name, so you should use an identifier that doesn't conflict with any MASM keywords or other global symbols.

The alignment field must be one of the following strings: *"byte"*, *"word"*, *"dword"*, *"para"*, or *"page"*. This alignment directive has the same meaning as the corresponding string in the segment renaming directives.

The "class" operand specifies the combine class. This field has the same meaning as the combine class operand in the segment renaming directives. Note that, like those directives, this operand must be a string constant. HLA does not check the syntax of this string. It should be a legal identifier that doesn't conflict with any MASM reserved words (just like the segment renaming directives' class field).

Segment names are global, even if you define a segment within a procedure. You may use the same segment name in several different segment declaration sections throughout your program; if you do, HLA (and MASM and the linker) will physically combine all the objects you declare in such a segment.

One nice thing about using different segments for variable declarations is that you physically separate the objects in memory. This reduces the impact of errant programs on data unrelated to the task at hand. For example, if you put each procedure's static variables in their own separate segment, this will reduce the likelihood that one procedure will accidentally overwrite another procedure's data if it oversteps an array bounds by a few bytes. Of course, the procedure can still wipe out its own variables by doing this, however, keeping the values in their own segment localizes the impact and makes it easier to track down this defect in your code. One bad thing about using separate segments for each procedure is that each segment consumes a minimum of 4K of memory; so you're program's executable will contain a lot of empty data if you have a large number of these segments and you don't declare 4K of data in each procedure.

13. Well, not really. segment declarations may not appear in classes or namespaces. See the appropriate sections later in this text for a discussion of classes and namespaces.

6.12 Controlling the Placement and Attributes of Segments in Memory (Windows Only)

Whenever you compile and HLA program, HLA produces several output files: in particular, HLA produces an ".ASM" file that HLA assembles via MASM, and a ".LINK" file that contains information for the linker. The ".LINK" file controls the placement of segments within memory (when the program actually executes) and it also controls other attributes of segments (such as whether they may contain executable code, whether the segment is writable, etc.). When HLA compiles a program to an executable, it first calls a program named "HLAPARSE.EXE" which is actually responsible for translating the HLA source code to a MASM-compatible ".ASM" file. Then HLA calls the LINK program to link the OBJ files that MASM produces with various library files to produce an executable file[14]. In addition to passing in the list of OBJ and LIB filenames, HLA also provides the linker with other useful information about segment placement. In this section we'll explore some of the linker options so you can run the linker separately should you want to exercise explicit control over the placement of segments in memory.

To get a (partial) list of the linker options, run the link.exe program with the "/?" command line option. The linker will respond with a list that looks something like the following:

```
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

usage: LINK [options] [files] [@commandfile]

   options:

      /ALIGN:#
      /BASE:{address|@filename,key}
      /COMMENT:comment
      /DEBUG
      /DEBUGTYPE:{CV|COFF}
      /DEF:filename
      /DEFAULTLIB:library
      /DELAY:{NOBIND|UNLOAD}
      /DELAYLOAD:dll
      /DLL
      /DRIVER[:{UPONLY|WDM}]
      /ENTRY:symbol
      /EXETYPE:DYNAMIC
      /EXPORT:symbol
      /FIXED[:NO]
      /FORCE[:{MULTIPLE|UNRESOLVED}]
      /GPSIZE:#
      /HEAP:reserve[,commit]
      /IMPLIB:filename
      /INCLUDE:symbol
      /INCREMENTAL:{YES|NO}
      /LARGEADDRESSAWARE[:NO]
      /LIBPATH:dir
      /LINK50COMPAT
      /MACHINE:{ALPHA|ARM|IX86|MIPS|MIPS16|MIPSR41XX|PPC|SH3|SH4}
      /MAP[:filename]
      /MAPINFO:{EXPORTS|FIXUPS|LINES}
      /MERGE:from=to
      /NODEFAULTLIB[:library]
      /NOENTRY
      /NOLOGO
```

14. If you've got any resource files, HLA will also call the resource compiler, rc.exe, to compile these files. Resource files are beyond the scope of this chapter, so we will ignore them here.

```
/OPT:{ICF[,iterations]|NOICF|NOREF|NOWIN98|REF|WIN98}
/ORDER:@filename
/OUT:filename
/PDB:{filename|NONE}
/PDBTYPE:{CON[SOLIDATE]|SEPT[YPES]}
/PROFILE
/RELEASE
/SECTION:name,[E][R][W][S][D][K][L][P][X]
/STACK:reserve[,commit]
/STUB:filename
/SUBSYSTEM:{NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX}[,#[.##]]
/SWAPRUN:{CD|NET}
/VERBOSE[:LIB]
/VERSION:#[.#]
/VXD
/WARN[:warninglevel]
/WINDOWSCE:{CONVERT|EMULATION}
/WS:AGGRESSIVE
```

Most of these options are very advanced, or of little use to us right now. However, a good number of them are useful on occasion so we'll discuss them here.

/ALIGN:*number* The number value must be a decimal number and it must be a power of two[15]. The default (which HLA uses) is 4096. This specifies the default alignment for each segment in the program. You should normally leave this at 4K, but if you write a lot of very short assembly programs you can shrink the size of the executable image by setting this to a smaller value. Note that this number should be at least as large as the largest alignment option (byte, word, dword, para, or page) that you specify for you segments.

The /BASE:*address* option lets you specify the starting address of the code segment ("_TEXT"). The linker defaults this address to 0x4000000 (i.e., $400_0000). HLA typically uses a default value of 0x3000000 ($300_0000). This leaves room for a 16 Mbyte unused block, a 16 Mbyte stack segment, and a 16 Mbyte heap segment below the code segment in memory (which is where the linker normally puts the stack and heap). If you want a larger heap or stack segment, you should specify a higher starting address with the /BASE linker option.

The /ENTRY:*name* options specifies the name of the main program. This is the location where program execution begins when Windows first executes the program. For HLA console window applications, the name of the main program is "?HLAMain". Unless you're linking HLA code with a main program written in another language, or you completely understand the HLA start up sequence, you should always use this identifier to specify the entry point of an HLA main program. Note that if you circumvent this entry point, HLA does not properly set up the exception handling facilities and other features of the language. So change this name at your own risk.

/HEAP:*reserve,commit* This option specifies the amount of memory that the system reserves for the heap. The first numeric value indicates the amount of heap space to reserve, the second parameter specifies the amount of that heap space to actual map into the address space. By default, HLA supplies 0x1000000 ($100_0000, or 16 Mbytes) for both values. This sets aside room for a 16 Mbyte heap and makes all of it available to your program. This is a rather large value for the heap, especially if you write short programs that don't allocate much memory dynamically. For most small applications you may want to set this to a more reasonable (smaller) value. The Windows default is one megabyte (0x100000 or $10_0000). If you don't do much dynamic memory allocation, your code will probably coexist better with other applications if you set this value to 128K (0x20000 or $2_0000). As a general rule, you should set both operands to the same value.

The /MACHINE:IX86 option tells the linker that you're creating code for an 80x86 CPU. You should not attempt to specify a different CPU when using HLA.

/MAP and /MAP:*filename*. These options tell the linker to produce a map *file*. The first form, without the optional filename, causes the linker to produce a map file with the same base name as the output file and

15. You can use a hexadecimal value if you specify the number using C/C++ syntax, e.g., "0x123ABC".

a suffix of ".map". The second form lets you specify the name of the map file. The map file is a text file that contains several bits of information about the object file. You should produce a map file something and view this information with a text editor to see the kind of information the linker produces. None of this information is usually essential, but it is handy to have now and then. By default, HLA does not produce a map file.

/MERGE:*from=to*. This option merges the segment (section) named *from* to *to*. This will cause the linker to concatenate the two segments in memory. This is roughly equivalent to using the same combine class string in the segment declaration. For example, "/MERGE:readonly=.edata" merges the readonly segment with the CONST segment by concatenating the two.

/OUT:*filename*. This option specifies the output (executable) filename. By default, HLA appends ".EXE" to the base name of your program and uses that as the executable name. If you would prefer a different name, then use this option to specify the executable file name that LINK produces.

/SECTION:*name,options*. This option lets you specify the ordering of segments in memory as well as apply attributes to those segments. The ".LINK" file that HLA produces contains a list of /SECTION commands to feed to the linker that specifies the ordering of the segments (by their appearance in the ".LINK" file) and the attributes of those segments. The *name* field is the segment name. This is a case sensitive field, so the case of *name* must exactly match the original segment declaration. The *options* field is a string of one or more characters that specifies the characteristics of that segment in memory. Here are some of the more common options:

•	E	Allows the execution of code in this segment
•	R	Allows the program to read data in this segment
•	W	Allows the program to write data in this segment
•	S	Shared. Allows multiple copies of this program to share this data.
•	K	Marks the page as non-cachable (generally for multiprocessing applications).
•	P	Marks the page as non-pageable (i.e., it must always be in real memory).

Most of the other options are either very advanced, uninteresting. or not applicable to HLA programs. Most segments will have at least one of the E, R, or W options. HLA's default segments generally use the following section options:

```
/SECTION:.text,ER     -- Note: .text = _TEXT
/SECTION:.edata,R     -- Note: .edata = CONST
/SECTION:readonly,R
/SECTION:.data,RW     -- Note: .data = _DATA
/SECTION:.bss,RW      -- Note: .bss = _BSS
```

/STACK:*reserve,commit*. This option is similar to the /HEAP option except it reserves space for the program's stack segment rather than the heap segment. Like the HEAP segment, HLA defaults the stack size to 16 Mbytes (0x4000000 or $400_0000). If you write shorter applications that don't use a lot of local variable space or heavy recursion, you may want to consider setting this value to one megabyte or less, e.g., /STACK:0x100000,0x100000.

/SUBSYSTEM:*system*. You must supply a subsystem option when you create an executable program. For HLA programs you would normally use "/SUBSYSTEM:CONSOLE" when writing a standard console application. You can use HLA to write GUI applications, if you do this, then you will need to use the "/SUBSYSTEM:WINDOWS" linker option. By default, HLA links your code with the "/SUBSYSTEM:CONSOLE" option. If you use the HLA "-w" command line option, then HLA will invoke the linker with the "/SUBSYSTEM:WINDOWS" option. Of course, if you explicitly run the linker yourself, you will have to supply one of these two options.

The preceding paragraphs explain most of the command line options you'll use when linking programs written in HLA. For more information about the linker, see the Microsoft on-line documentation that accompanies the linker.

If you get tired of typing really long linker command lines every time you compile and link an HLA program, you can gather all the (non-changing) command line options into a linker command file and tell the linker to grab those options and filenames from the command file rather than from the command line. The ".LINK" file that the HLA compiler produces is a good example of a linker command file. The ".LINK" file contains the /SECTION options for the default and user-defined segments found in an HLA program.

Rather than manually supplying these options on each call to the linker, you can use a command line like the following:

```
link @filename.link  other_options file_names
```

The at-sign ("@") tells the linker to read a list of commands from the specified command file. Note that you can have several different command files, so if you're compiling and linking several different HLA source files, you can specify the ".link" file for each compilation on the command line.

The filenames you specify on the linker command line should be the names of OBJ and LIB files that you wish to link together. In addition to the OBJ files you've created with HLA, you'll probably want to specify the following library files:

•	kernel32.lib	Contains definitions for the base Windows API (e.g., console stuff)
•	user32.lib	Contains the definition of the MessageBox dialog (used for exceptions).
•	hlalib.lib	The HLA Standard Library

If you don't call any HLA Standard Library routines (unlikely, but possible) then you obviously don't need to specify the hlalib.lib file. Note that it doesn't hurt to specify the name of a library whose members you don't use. The linker will not include any object code from a library unless the program actually uses code or data from that library.

If you're manually linking code that you compile with HLA, you will probably want to create one linker command file containing all the static commands and include that and any appropriate HLA ".LINK" files on the linker command line. Here's a typical example of a static link file (i.e., a file that doesn't get rewritten each time you compile the HLA program):

```
/heap:0x20000, 0x20000
/stack:0x2000, 0x20000
/base:0x1000000
/machine:IX86
/entry:?HLAMain
/out:mypgm.exe
kernel32.lib
user32.lib
hlalib.lib
```

Generally, you'd use the /SECTION commands from the HLA ".LINK" file unless you wanted to explicitly set the segment ordering or change the attributes of the memory segments.

To run the linker manually, you'd normally tell HLA to perform a compile (and assemble) only operation. This is done using the HLA "-c" command line option. That is, a command like "hla -c myfile.hla" will compile "myfile.hla" to "myfile.asm" and then run MASM to assemble this to "myfile.obj". HLA will not run the linker when you specify the "-c" option. If you prefer, you can run MASM separately by using the "-s" command line option as follows:

```
hla -s myfile.hla
ml -c -Cp -COFF myfile.asm
```

However, there is very little benefit to running the assembler yourself (run "MASM /?" to see the available MASM command line options).

Once you've compiled all necessary source files, you can link them by using the Microsoft LINK.EXE program with the command line (or command file) options this section discusses. Note that this section discusses options specific to the LINK.EXE v6.0 product. These features may change in a future version of the linker. Please see the Microsoft documentation if you have any questions about how the linker operates or if you're using a different version of the linker.

6.13 Putting it All Together

CPU architects divide memory into several different types depending on cost, capacity, and speed. They call this the memory hierarchy. Many of the levels in the memory hierarchy are transparent to the programmer. That is, the system automatically moves data between levels in the memory hierarchy without intervention on the programmer's part. However, if you are aware of the effects of the memory hierarchy on program performance, you can write faster programs by organizing your data and code so that it conforms to the expectations of the caching and virtual memory subsystems in the memory hierarchy.