soon. We can keep track of the last time each block was accessed (assign a time-stamp to the block), and select as the victim block the block that has been used least recently. This is the *least recently used* (*LRU*) algorithm. Unfortunately, LRU requires the system to keep a history of accesses for every cache block, which requires significant space and slows down the operation of the cache. There are ways to approximate LRU, but that is beyond the scope of this book. (Refer to the references at the end of the chapter for more information.)

First in, first out (*FIFO*) is another popular approach. With this algorithm, the block that has been in cache the longest (regardless of how recently it has been used) would be selected as the victim to be removed from cache memory.

Another approach is to select a victim at *random*. The problem with LRU and FIFO is that there are degenerate referencing situations in which they can be made to *thrash* (constantly throw out a block, then bring it back, then throw it out, then bring it back, repeatedly). Some people argue that random replacement, although it sometimes throws out data that will be needed soon, never thrashes. Unfortunately, it is difficult to have truly random replacement, and it can decrease average performance.

The algorithm selected often depends on how the system will be used. No single (practical) algorithm is best for all scenarios. For that reason, designers use algorithms that perform well under a wide variety of circumstances.

6.4.3 Effective Access Time and Hit Ratio

The performance of a hierarchical memory is measured by its *effective access time* (*EAT*), or the average time per access. EAT is a weighted average that uses the hit ratio and the relative access times of the successive levels of the hierarchy. For example, suppose the cache access time is 10ns, main memory access time is 200ns, and the cache hit rate is 99%. The average time for the processor to access an item in this two-level memory would then be:

$$\text{EAT} = \underbrace{0.99(10\text{ns})}_{\text{cache hit}} + \underbrace{0.01(200\text{ns})}_{\text{cache miss}} = 9.9\text{ns} + 2\text{ns} = 11\text{ns}$$

What, exactly, does this mean? If we look at the access times over a long period of time, this system performs as if it had a single large memory with an 11ns access time. A 99% cache hit rate allows the system to perform very well, even though most of the memory is built using slower technology with an access time of 200ns.

The formula for calculating effective access time for a two-level memory is given by:

$$\text{EAT} = H \times \text{Access}_C + (1 - H) \times \text{Access}_{MM}$$

where H = cache hit rate, Access_C = cache access time, and Access_{MM} = main memory access time.

This formula can be extended to apply to three- or even four-level memories, as we will see shortly.

6.4.4 When Does Caching Break Down?

When programs exhibit locality, caching works quite well. However, if programs exhibit bad locality, caching breaks down and the performance of the memory hierarchy is poor. In particular, object-oriented programming can cause programs to exhibit less than optimal locality. Another example of bad locality can be seen in two-dimensional array access. Arrays are typically stored in row-major order. Suppose, for purposes of this example, that one row fits exactly in one cache block and cache can hold all but one row of the array. If a program accesses the array one row at a time, the first row access produces a miss, but once the block is transferred into cache, all subsequent accesses to that row are hits. So a 5×4 array would produce 5 misses and 15 hits over 20 accesses (assuming we are accessing each element of the array). If a program accesses the array in column-major order, the first access to the column results in a miss, after which an entire row is transferred in. However, the second access to the column results in another miss. The data being transferred in for each row is not being used because the array is being accessed by column. Because cache is not large enough, this would produce 20 misses on 20 accesses. A third example would be a program that loops through a linear array that does not fit in cache. There would be a significant reduction in the locality when memory is used in this fashion.

6.4.5 Cache Write Policies

In addition to determining which victim to select for replacement, designers must also decide what to do with so-called *dirty blocks* of cache, or blocks that have been modified. When the processor writes to main memory, the data may be written to the cache instead under the assumption that the processor will probably read it again soon. If a cache block is modified, the cache *write policy* determines when the actual main memory block is updated to match the cache block. There are two basic write policies:

- *Write-through*—A write-through policy updates both the cache and the main memory simultaneously on every write. This is slower than write-back, but ensures that the cache is consistent with the main system memory. The obvious disadvantage here is that every write now requires a main memory access. Using a write-through policy means every write to the cache necessitates a main memory write, thus slowing the system (if all accesses are write, this essentially slows the system down to main memory speed). However, in real applications, the majority of accesses are reads so this slow-down is negligible.
- *Write-back*—A write-back policy (also called *copyback*) only updates blocks in main memory when the cache block is selected as a victim and must be removed from cache. This is normally faster than write-through because time is not wasted writing information to memory on each write to cache. Memory traffic is also reduced. The disadvantage is that main memory and cache may not contain the same value at a given instant of time, and if a process terminates (crashes) before the write to main memory is done, the data in cache may be lost.

To improve the performance of cache, one must increase the hit ratio by using a better mapping algorithm (up to roughly a 20% increase), better strategies for

write operations (potentially a 15% increase), better replacement algorithms (up to a 10% increase), and better coding practices, as we saw in the earlier example of row versus column-major access (up to a 30% increase in hit ratio). Simply increasing the size of cache may improve the hit ratio by roughly 1–4%, but is not guaranteed to do so.

6.5 VIRTUAL MEMORY

You now know that caching allows a computer to access frequently used data from a smaller but faster cache memory. Cache is found near the top of our memory hierarchy. Another important concept inherent in the hierarchy is *virtual memory*. The purpose of virtual memory is to use the hard disk as an extension of RAM, thus increasing the available address space a process can use. Most personal computers have a relatively small amount (typically less than 512MB) of main memory. This is usually not enough memory to hold multiple applications concurrently, such as a word processing application, an e-mail program, and a graphics program, in addition to the operating system itself. Using virtual memory, your computer addresses more main memory than it actually has, and it uses the hard drive to hold the excess. This area on the hard drive is called a *page file*, because it holds chunks of main memory on the hard drive. The easiest way to think about virtual memory is to conceptualize it as an imaginary memory location in which all addressing issues are handled by the operating system.

The most common way to implement virtual memory is by using *paging*, a method in which main memory is divided into fixed-size blocks and programs are divided into the same size blocks. Typically, chunks of the program are brought into memory as needed. It is not necessary to store contiguous chunks of the program in contiguous chunks of main memory. Because pieces of the program can be stored out of order, program addresses, once generated by the CPU, must be translated to main memory addresses. Remember, in caching, a main memory address had to be transformed into a cache location. The same is true when using virtual memory; every virtual address must be translated into a physical address. How is this done? Before delving further into an explanation of virtual memory, let's define some frequently used terms for virtual memory implemented through paging:

- *Virtual address*—The logical or program address that the process uses. Whenever the CPU generates an address, it is always in terms of virtual address space.
- *Physical address*—The real address in physical memory.
- *Mapping*—The mechanism by which virtual addresses are translated into physical ones (very similar to cache mapping)
- *Page frames*—The equal-size chunks or blocks into which main memory (physical memory) is divided.
- *Pages*—The chunks or blocks into which virtual memory (the logical address space) is divided, each equal in size to a page frame. Virtual pages are stored on disk until needed.

- *Paging*—The process of copying a virtual page from disk to a page frame in main memory.
- *Fragmentation*—Memory that becomes unusable.
- *Page fault*—An event that occurs when a requested page is not in main memory and must be copied into memory from disk.

Because main memory and virtual memory are divided into equal size pages, pieces of the process address space can be moved into main memory but need not be stored contiguously. As previously stated, we need not have all of the process in main memory at once; virtual memory allows a program to run when only specific pieces are present in memory. The parts not currently being used are stored in the page file on disk.

Virtual memory can be implemented with different techniques, including paging, segmentation, or a combination of both, but paging is the most popular. (This topic is covered in great detail within the study of operating systems.) The success of paging, like that of cache, is very dependent on the locality principle. When data is needed that does not reside in main memory, the entire block in which it resides is copied from disk to main memory, in hopes that other data on the same page will be useful as the program continues to execute.

6.5.1 Paging

The basic idea behind paging is quite simple: Allocate physical memory to processes in fixed size chunks (page frames) and keep track of where the various pages of the process reside by recording information in a *page table*. Every process has its own page table that typically resides in main memory, and the page table stores the physical location of each virtual page of the process. The page table has N rows, where N is the number of virtual pages in the process. If there are pages of the process currently not in main memory, the page table indicates this by setting a *valid bit* to 0; if the page is in main memory, the valid bit is set to 1. Therefore, each entry of the page table has two fields: a valid bit and a frame number.

Additional fields are often added to relay more information. For example, a *dirty bit* (or a *modify bit*) could be added to indicate whether the page has been changed. This makes returning the page to disk more efficient, because if it is not modified, it does not need to be rewritten to disk. Another bit (the *usage bit*) can be added to indicate the page usage. This bit is set to 1 whenever the page is accessed. After a certain time period, the usage bit is set to 0. If the page is referenced again, the usage bit is set to 1. However, if the bit remains 0, this indicates that the page has not been used for a period of time, and the system might benefit by sending this page out to disk. By doing so, the system frees up this page's location for another page that the process eventually needs (we discuss this in more detail when we introduce replacement algorithms).

Virtual memory pages are the same size as physical memory page frames. Process memory is divided into these fixed size pages, resulting in potential *internal fragmentation* when the last page is copied into memory. The process may not

actually need the entire page frame, but no other process may use it. Therefore, the unused memory in this last frame is effectively wasted. It might happen that the process itself requires less than one page in its entirety, but it must occupy an entire page frame when copied to memory. Internal fragmentation is unusable space *within* a given partition (in this case, a page) of memory.

Now that you understand what paging is, we will discuss how it works. When a process generates a virtual address, the operating system must dynamically translate this virtual address into the physical address in memory at which the data actually resides. (For purposes of simplicity, let's assume we have no cache memory for the moment.) For example, from a program viewpoint, we see the final byte of a 10-byte program as address 9, assuming 1-byte instructions and 1-byte addresses, and a starting address of 0. However, when actually loaded into memory, the logical address 9 (perhaps a reference to the label *X* in an assembly language program) may actually reside in physical memory location 1239, implying the program was loaded starting at physical address 1230. There must be an easy way to convert the logical, or virtual, address 9 to the physical address 1230.

To accomplish this address translation, a virtual address is divided into two fields: a *page field* and an *offset field*, to represent the offset within that page where the requested data is located. This address translation process is similar to the process we used when we divided main memory addresses into fields for the cache mapping algorithms. And similar to cache blocks, page sizes are usually powers of 2; this simplifies the extraction of page numbers and offsets from virtual addresses.

To access data at a given virtual address, the system performs the following steps:

1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Translate the page number into a physical page frame number by accessing the page table.
   - **A.** Look up the page number in the page table (using the virtual page number as an index).
   - **B.** Check the valid bit for that page.
     1. If the valid bit = 0, the system generates a page fault and the operating system must intervene to
        - **a.** Locate the desired page on disk.
        - **b.** Find a free page frame (this may necessitate removing a "victim" page from memory and copying it back to disk if memory is full).
        - **c.** Copy the desired page into the free page frame in main memory.
        - **d.** Update the page table. (The virtual page just brought in must have its frame number and valid bit in the page table modified. If there was a "victim" page, its valid bit must be set to zero.)
        - **e.** Resume execution of the process causing the page fault, continuing to Step B2.

2. If the valid bit = 1, the page is in memory.
   - **a.** Replace the virtual page number with the actual frame number.
   - **b.** Access data at offset in physical page frame by adding the offset to the frame number for the given virtual page.

Please note that if a process has free frames in main memory when a page fault occurs, the newly retrieved page can be placed in any of those free frames. However, if the memory allocated to the process is full, a victim page must be selected. The replacement algorithms used to select a victim are quite similar to those used in cache. FIFO, Random, and LRU are all potential replacement algorithms for selecting a victim page. (For more information on replacement algorithms, see the references at the end of this chapter.)

Let's look at an example. Suppose that we have a virtual address space of 2^8 words for a given process (this means the program generates addresses in the range 0 to 255_{10} which is 00 to FF_{16}), and physical memory of 4 page frames (no cache). Assume also that pages are 32 words in length. Virtual addresses contain 8 bits, and physical addresses contain 7 bits (4 frames of 32 words each is 128 words, or 2^7). Suppose, also, that some pages from the process have been brought into main memory. Figure 6.11 illustrates the current state of the system.

Each virtual address has 8 bits and is divided into 2 fields: the page field has 3 bits, indicating there are 2^3 pages of virtual memory $\left(\frac{2^8}{2^5}\right) = 2^3$. Each page is $2^5 = 32$ words in length, so we need 5 bits for the page offset. Therefore, an 8-bit virtual address has the format shown in Figure 6.12.

Suppose the system now generates the virtual address $13_{10} = 0D_{16} = 00001101_2$. Dividing the binary address into the page and offset fields (see Figure

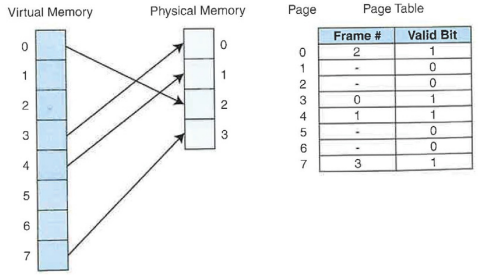


FIGURE 6.11 Current State Using Paging and the Associated Page Table

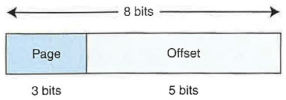


FIGURE 6.12 Format for an 8-Bit Virtual Address with 2^5 = 32 Word Page Size

6.13), we see the page field P = 000_2 and the offset field equals 01101_2. To continue the translation process, we use the 000 value of the page field as an index into the page table. Going to the 0th entry in the page table, we see that virtual page 0 maps to physical page frame $2 = 10_2$. Thus the translated physical address becomes page frame 2, offset 13. Note that a physical address has only 7 bits (2 for the frame, because there are 4 frames, and 5 for the offset). Written in binary, using the two fields, this becomes 1001101_2, or address $4D_{16} = 77_{10}$ and is shown in Figure 6.14. We can also find this address another way. Each page has 32 words. We know the virtual address we want is on virtual page 0, which maps to physical page frame 2. Frame 2 begins with address 64. An offset of 13 results in address 77.

Let's look at a complete example in a real (but small) system (again, with no cache). Suppose a program is 16 bytes long, has access to an 8-byte memory that uses byte addressing (this means each byte, or word, has its own address), and a page is 2 words (bytes) in length. As the program executes, it generates the following address reference string (addresses are given in decimal values): 0, 1, 2, 3, 6, 7, 10, 11. (This address reference string indicates that address 0 is referenced first, then address 1, then address 2, and so on.) Originally, memory contains no pages for this program. When address 0 is needed, both address 0 and address 1 (in page 0) are copied to page frame 2 in main memory (it could be that frames 0

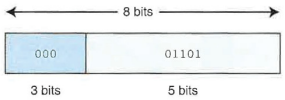


FIGURE 6.13 Format for Virtual Address $00001101_2 = 13_{10}$

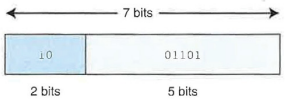


FIGURE 6.14 Format for Physical Address $101101_2 = 77_{10}$

and 1 of memory are occupied by another process and thus unavailable). This is an example of a page fault, because the desired page of the program had to be fetched from disk. When address 1 is referenced, the data already exists in memory (so we have a page hit). When address 2 is referenced, this causes another page fault, and page 1 of the program is copied to frame 0 in memory. This continues, and after these addresses are referenced and pages are copied from disk to main memory, the state of the system is as shown in Figure 6.15a. We see that address 0 of the program, which contains the data value "A", currently resides in

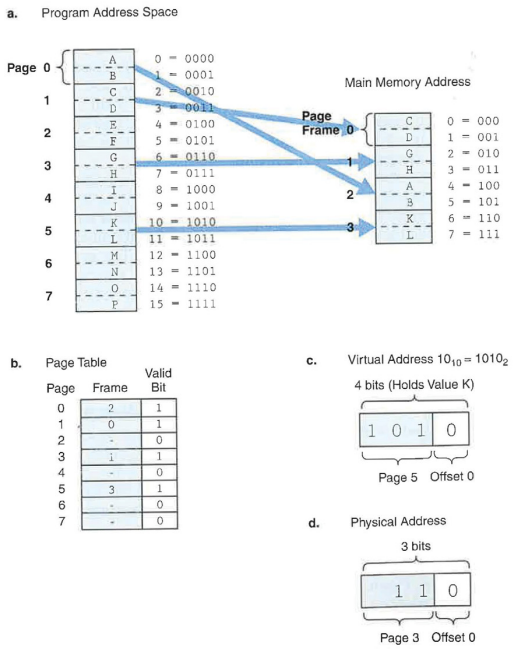


FIGURE 6.15 A Small Memory Example

memory location 4 = 100_2. Therefore, the CPU must translate from virtual address 0 to physical address 4, and uses the translation scheme described above to do this. Note that main memory addresses contain 3 bits (there are 8 bytes in memory), but virtual addresses (from the program) must have 4 bits (because there are 16 bytes in the virtual address). Therefore, the translation must also convert a 4-bit address into a 3-bit address.

Figure 6.15b depicts the page table for this process after the given pages have been accessed. We can see that pages 0, 1, 3, and 5 of the process are valid, and thus reside in memory. Pages 2, 6, and 7 are not valid and would each cause page faults if referenced.

Let's take a closer look at the translation process. Suppose the CPU now generates program, or virtual, address 10 = 1010_2 for a second time. We see in Figure 6.15a that the data at this location, "K", resides in main memory address 6 = 0110_2. However, the computer must perform a specific translation process to find the data. To accomplish this, the virtual address, 1010_2, is divided into a page field and an offset field. The page field is 3 bits long because there are 8 pages in the program. This leaves 1 bit for the offset, which is correct because there are only 2 words on each page. This field division is illustrated in Figure 6.15c.

Once the computer sees these fields, it is a simple matter to convert to the physical address. The page field value of 101_2 is used as an index into the page table. Because $101_2 = 5$, we use 5 as the offset into the page table (Figure 6.15b) and see that virtual page 5 maps to physical frame 3. We now replace the $5 = 101_2$ with $3 = 11_2$, but keep the same offset. The new physical address is 110_2, as shown in Figure 6.15d. This process successfully translates from virtual addresses to physical addresses and reduces the number of bits from four to three as required.

Now that we have worked with a small example, we are ready for a larger, more realistic example. Suppose we have a virtual address space of 8K words, a physical memory size of 4K words that uses byte addressing, and a page size of 1K words (there is no cache on this system either, but we are getting closer to understanding how memory works and eventually will use both paging and cache in our examples), and a word size of one byte. A virtual address has a total of 13 bits ($8K = 2^{13}$), with 3 bits used for the page field (there are $\frac{2^{13}}{2^{10}} = 2^3$ virtual pages), and 10 used for the offset (each page has 2^{10} bytes). A physical memory address has only 12 bits ($4K = 2^{12}$), with the first 2 bits as the page field (there are 2^2 page frames in main memory) and the remaining 10 bits as the offset within the page. The formats for the virtual address and physical address are shown in Figure 6.16a.

For purposes of this example, let's assume we have the page table indicated in Figure 6.16b. Figure 6.16c shows a table indicating the various main memory addresses (in base 10) that is useful for illustrating the translation.

Suppose the CPU now generates virtual address $5459_{10} = 1010101010011_2$. Figure 6.16d illustrates how this address is divided into the page and offset fields and how it is converted to the physical address $1363_{10} = 010101010011_2$. Essentially, the

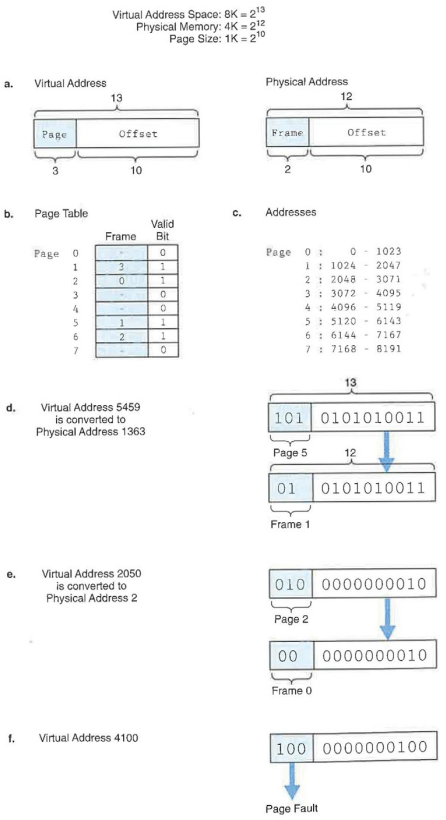


FIGURE 6.16 A Larger Memory Example

page field 101 of the virtual address is replaced by the frame number 01, since page 5 maps to frame 1 (as indicated in the page table). Figure 6.16e illustrates how virtual address 2050_{10} is translated to physical address 2. Figure 6.16f shows virtual address 4100_{10} generating a page fault; page $4 = 100_2$ is not valid in the page table.

It is worth mentioning that selecting an appropriate page size is very difficult. The larger the page size is, the smaller the page table is, thus saving space in main memory. However, if the page is too large, the internal fragmentation becomes worse. Larger page sizes also mean fewer actual transfers from disk to main memory as the chunks being transferred are larger. However, if they are too large, the principle of locality begins to break down and we are wasting resources by transferring data that may not be necessary.

6.5.2 Effective Access Time Using Paging

When we studied cache, we introduced the notion of effective access time. We also need to address EAT while using virtual memory. There is a time penalty associated with virtual memory: For each memory access that the processor generates, there must now be *two* physical memory accesses—one to reference the page table and one to reference the actual data we wish to access. It is easy to see how this affects the effective access time. Suppose a main memory access requires 200ns and that the page fault rate is 1% (99% of the time we find the page we need in memory). Assume it costs us 10ms to access a page not in memory (this time of 10ms includes the time necessary to transfer the page into memory, update the page table, and access the data). The effective access time for a memory access is now:

$$EAT = .99(200ns + 200ns) + .01(10ms) = 100{,}396ns$$

Even if 100% of the pages were in main memory, the effective access time would be:

$$EAT = 1.00(200ns + 200ns) = 400ns,$$

which is double the access time of memory. Accessing the page table costs us an additional memory access because the page table itself is stored in main memory.

We can speed up the page table lookup by storing the most recent page lookup values in a page table cache called a *translation look-aside buffer* (*TLB*). Each TLB entry consists of a virtual page number and its corresponding frame

Virtual Page Number	Physical Page Number
-	-
5	1
2	0
-	-
-	-
1	3
6	2

TABLE 6.2 Current State of the TLB for Figure 6.16

number. A possible state of the TLB for the previous page table example is indicated in Table 6.2.

Typically, the TLB is implemented as associative cache, and the virtual page/frame pairs can be mapped anywhere. Here are the steps necessary for an address lookup, when using a TLB (see Figure 6.17):

1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Search for the virtual page number in the TLB.
4. If the (virtual page #,page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
5. If there is a TLB miss, go to the page table to get the necessary frame number. If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.
6. If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.

6.5.3 Putting It All Together: Using Cache, TLBs, and Paging

Because the TLB is essentially a cache, putting all of these concepts together can be confusing. A walkthrough of the entire process will help you to grasp the overall idea. When the CPU generates an address, it is an address relative to the program itself, or a virtual address. This virtual address must be converted into a physical address before the data retrieval can proceed. There are two ways this is accomplished: (1) use the TLB to find the frame by locating a recently cached (page, frame) pair; or (2) in the event of a TLB miss, use the page table to find the corresponding frame in main memory (typically the TLB is updated at this point as well). This frame number is then combined with the offset given in the virtual address to create the physical address.

At this point, the virtual address has been converted into a physical address but the data at that address has not yet been retrieved. There are two possibilities for retrieving the data: (1) search cache to see if the data resides there; or (2) on a cache miss, go to the actual main memory location to retrieve the data (typically cache is updated at this point as well).

Figure 6.18 illustrates the process of using a TLB, paging, and cache memory.

6.5.4 Advantages and Disadvantages of Paging and Virtual Memory

In Section 6.5.2, we discussed how virtual memory implemented through paging adds an extra memory reference when accessing data. This time penalty is partially alleviated by using a TLB to cache page table entries. However, even with a high hit ratio in the TLB, this process still incurs translation overhead. Another disadvantage of virtual memory and paging is the extra resource consumption (the memory overhead for storing page tables). In extreme cases (very large programs), the page tables may take up a significant portion of physical memory. One solution offered for this latter problem is to page the page tables, which can

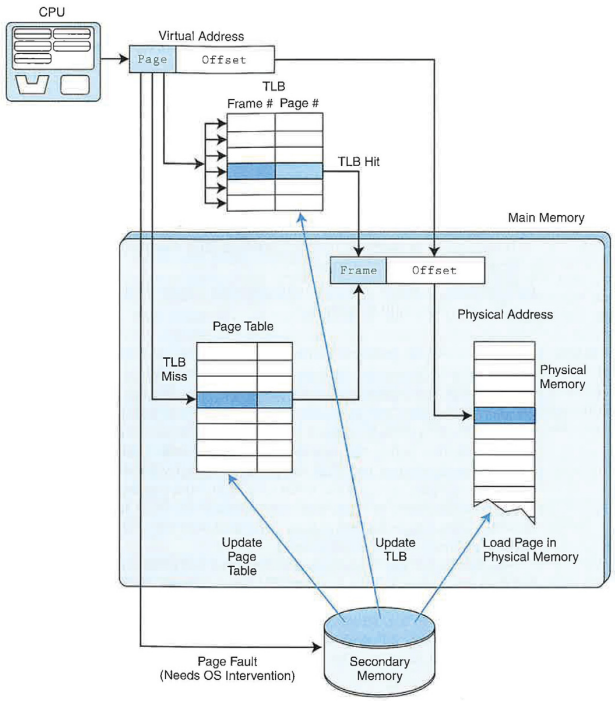


FIGURE 6.17 Using the TLB

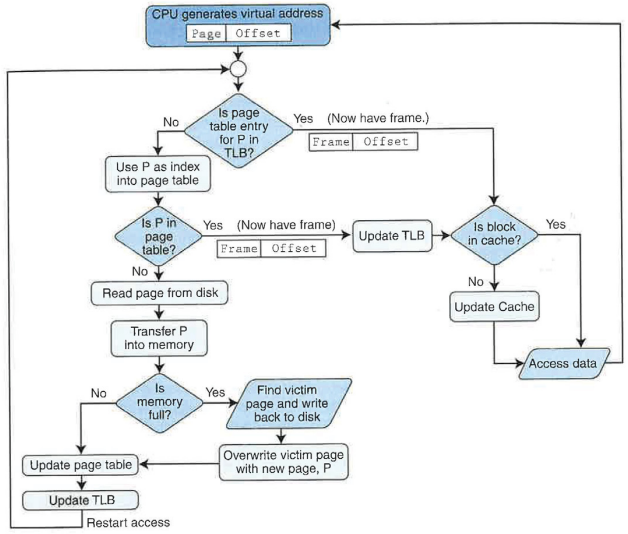


FIGURE 6.18 Putting It All Together: The TLB, Page Table, Cache and Main Memory

get very confusing indeed! Virtual memory and paging also require special hardware and operating system support.

The benefits of using virtual memory must outweigh these disadvantages to make it useful in computer systems. But what are the advantages of virtual memory and paging? It is quite simple: Programs are no longer restricted by the amount of physical memory that is available. Virtual memory permits us to run individual programs whose virtual address space is larger than physical memory. (In effect, this allows one process to share physical memory with itself.) This makes it much easier to write programs because the programmer no longer has to worry about the physical address space limitations. Because each program requires less physical memory, virtual memory also permits us to run more programs at the same time. This allows us to share the machine among processes whose total address space sizes exceed the physical memory size, resulting in an increase in CPU utilization and system throughput.

The fixed size of frames and pages simplifies both allocation and placement from the perspective of the operating system. Paging also allows the operating system to specify protection ("this page belongs to User X and you can't access it") and sharing ("this page belongs to User X but you can read it") on a per page basis.

6.5.5 Segmentation

Although it is the most common method, paging is not the only way to implement virtual memory. A second method employed by some systems is *segmentation*. Instead of dividing the virtual address space into equal, fixed-size pages, and the physical address space into equal-size page frames, the virtual address space is divided into logical, variable-length units, or *segments*. Physical memory isn't really divided or partitioned into anything. When a segment needs to be copied into physical memory, the operating system looks for a chunk of free memory large enough to store the entire segment. Each segment has a base address, indicating were it is located in memory, and a bounds limit, indicating its size. Each program, consisting of multiple segments, now has an associated *segment table* instead of a page table. This segment table is simply a collection of the base/bounds pairs for each segment.

Memory accesses are translated by providing a segment number and an offset within the segment. Error checking is performed to make sure the offset is within the allowable bound. If it is, then the base value for that segment (found in the segment table) is added to the offset, yielding the actual physical address. Because paging is based on a fixed-size block and segmentation is based on a logical block, protection and sharing are easier using segmentation. For example, the virtual address space might be divided into a code segment, a data segment, a stack segment, and a symbol table segment, each of a different size. It is much easier to say "I want to share all of my data, so make my data segment accessible to everyone" than it is to say "OK, in which pages does my data reside, and now that I have found those four pages, let's make three of the pages accessible, but only half of that fourth page accessible."

As with paging, segmentation suffers from fragmentation. Paging creates internal fragmentation because a frame can be allocated to a process that doesn't need the entire frame. Segmentation, on the other hand, suffers from *external fragmentation*. As segments are allocated and deallocated, the free chunks that reside in memory become broken up. Eventually, there are many small chunks, but none large enough to store an entire segment. The difference between external and internal fragmentation is that, with external fragmentation, enough total memory space may exist to allocate to a process, but this space is not contiguous—it exists as a large number of small, unusable holes. With internal fragmentation, the memory simply isn't available because the system has over-allocated memory to a process that doesn't need it. To combat external fragmentation, systems use some sort of *garbage collection*. This process simply shuffles occupied chunks of memory to coalesce the smaller, fragmented chunks into larger, usable chunks. If you have ever defragmented a disk drive, you have witnessed a similar

process, collecting the many small free spaces on the disk and creating fewer, larger ones.

6.5.6 Paging Combined with Segmentation

Paging is not the same as segmentation. Paging is based on a purely physical value: The program and main memory are divided up into the same physical size chunks. Segmentation, on the other hand, allows for logical portions of the program to be divided into variable-sized partitions. With segmentation, the user is aware of the segment sizes and boundaries; with paging, the user is unaware of the partitioning. Paging is easier to manage: allocation, freeing, swapping, and relocating are easy when everything's the same size. However, pages are typically smaller than segments, which means more overhead (in terms of resources to both track and transfer pages). Paging eliminates external fragmentation, whereas segmentation eliminates internal fragmentation. Segmentation has the ability to support sharing and protection, both of which are very difficult to do with paging.

Paging and segmentation both have their advantages; however, a system does not have to use one or the other—these two approaches can be combined, in an effort to get the best of both worlds. In a combined approach, the virtual address space is divided into segments of variable length, and the segments are divided into fixed-size pages. Main memory is divided into the same size frames.

Each segment has a page table, which means every program has multiple page tables. The physical address is divided into three fields. The first field is the segment field, which points the system to the appropriate page table. The second field is the page number, which is used as an offset into this page table. The third field is the offset within the page.

Combined segmentation and paging is very advantageous because it allows for segmentation from the user's point of view and paging from the system's point of view.

6.6 A REAL-WORLD EXAMPLE OF MEMORY MANAGEMENT

Because the Pentium exhibits fairly characteristic traits of modern memory management, we present a short overview of how this processor deals with memory.

The Pentium architecture allows for 32-bit virtual addresses and 32-bit physical addresses. It uses either 4KB or 4MB page sizes, when using paging. Paging and segmentation can be applied in different combinations, including unsegmented, unpaged memory; unsegmented, paged memory; segmented, unpaged memory; and segmented, paged memory.

The Pentium has two caches, L1 and L2, both utilizing a 32-byte block size. L1 is next to the processor, whereas L2 is between the processor and memory. The L1 cache is actually two caches; the Pentium (like many other machines) separates L1 cache into cache used to hold instructions (called the *I-cache*) and cache used to hold data (called the *D-cache*). Both L1 caches utilize an LRU bit for dealing with

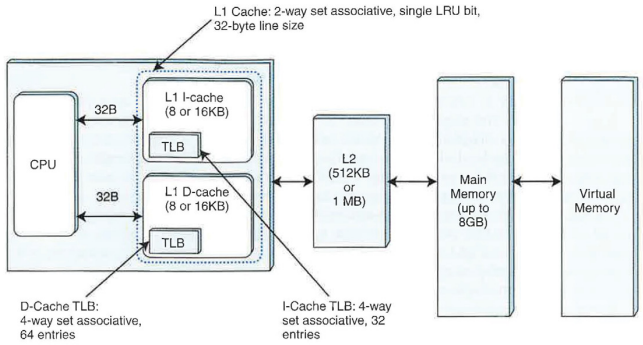


FIGURE 6.19 Pentium Memory Hierarchy

block replacement. Each L1 cache has a TLB: the D-cache TLB has 64 entries and the I-cache has only 32 entries. Both TLBs are 4-way set associative and use a pseudo-LRU replacement. The L1 D-cache and I-cache both use 2-way set associative mapping. The L2 cache can be from 512KB (for earlier models) up to 1MB (in later models). The L2 cache, like both L1 caches, uses 2-way set associative mapping.

To manage access to memory, the Pentium I-cache and the L2 cache use the MESI cache coherency protocol. Each cache line has two bits that store one of the following MESI states: (1) M: modified (cache is different than main memory); (2) E: exclusive (cache has not been modified and is the same as memory); (3) S: shared (this line/block may be shared with another cache line/block); and (4) I: invalid (the line/block is not in cache). Figure 6.19 presents an overview of the Pentium memory hierarchy.

We have given only a brief and basic overview of the Pentium and its approach to memory management. If you are interested in more details, please check the "Further Reading" section.

CHAPTER SUMMARY

Memory is organized as a hierarchy, with larger memories being cheaper but slower, and smaller memories being faster but more expensive. In a typical memory hierarchy, we find a cache, main memory, and secondary memory (usually a disk drive). The principle of locality helps bridge the gap between successive layers of this hierarchy, and the programmer gets the impression of a very fast and very large memory without being concerned about the details of transfers among the various levels of this hierarchy.

Cache acts as a buffer to hold the most frequently used blocks of main memory and is close to the CPU. One goal of the memory hierarchy is for the processor to see an effective access time very close to the access time of the cache. Achieving this goal depends on the behavioral properties of the programs being executed, the size and organization of the cache, and the cache replacement policy. Processor references that are found in cache are called cache hits; if not found, they are cache misses. On a miss, the missing data is fetched from main memory, and the entire block containing the data is loaded into cache.

The organization of cache determines the method the CPU uses to search cache for different memory addresses. Cache can be organized in different ways: direct mapped, fully associative, or set associative. Direct mapped cache needs no replacement algorithm; however, fully associative and set associative must use FIFO, LRU, or some other placement policy to determine the block to remove from cache to make room for a new block, if cache is full. LRU gives very good performance but is very difficult to implement.

Another goal of the memory hierarchy is to extend main memory by using the hard disk itself, also called virtual memory. Virtual memory allows us to run programs whose virtual address space is larger than physical memory. It also allows more processes to run concurrently. The disadvantages of virtual memory implemented with paging include extra resource consumption (storing the page table) and extra memory accesses (to access the page table), unless a TLB is used to cache the most recently used virtual/physical address pairs. Virtual memory also incurs a translation penalty to convert the virtual address to a physical one as well as a penalty for processing a page fault should the requested page currently reside on disk instead of main memory. The relationship between virtual memory and main memory is very similar to the relationship between main memory and cache. Owing to this similarity, the concepts of cache memory and the TLB are often confused. In reality the TLB *is* a cache. It is important to realize that virtual addresses must be translated to physical ones first, before anything else can be done, and this is what the TLB does. Although cache and paged memory appear to be very similar, the objectives are different: Cache improves the effective access time to main memory whereas paging extends the size of main memory.

FURTHER READING

Mano (1991) has a nice explanation of RAM. Stallings (2000) also gives a very good explanation of RAM. Hamacher, Vranesic, and Zaky (2002) contains an extensive discussion of cache. For good coverage of virtual memory, see Stallings (2001), Tanenbaum (1999), or Tanenbaum and Woodhull (1997). For more information on memory management in general, check out Flynn and McHoes (1991),

Stallings (2001), Tanenbaum and Woodhull (1997), or Silberschatz, Galvin, and Gagne (2001) books. Hennessy and Patterson (1996) discuss issues involved with determining cache performance. For an online tutorial on memory technologies, see www.kingston.com/tools/umg. George Mason University also has a set of workbenches on various computer topics. The workbench for virtual memory is located at cne.gmu.edu/workbenches/vmsim/vmsim.html.

REFERENCES

Davis, W. *Operating Systems, A Systematic View*, 4th ed., Redwood City, CA: Benjamin/Cummings, 1992.

Flynn, I. M., & McHoes, A. M. *Understanding Operating Systems.* Pacific Grove, CA: Brooks/Cole, 1991.

Hamacher, V. C., Vranesic, Z. G., & Zaky, S. G. *Computer Organization*, 5th ed., New York: McGraw-Hill, 2002.

Hennessy, J. L., & Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed., San Francisco: Morgan Kaufmann, 1996.

Mano, Morris. *Digital Design*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1991.

Silberschatz, A., Galvin, P., & Gagne, G. *Operating System Concepts*, 6th ed., Reading, MA: Addison-Wesley, 2001.

Stallings, W. *Computer Organization and Architecture*, 5th ed., New York: Macmillan Publishing Company, 2000.

Stallings, W. *Operating Systems*, 4th ed., New York: Macmillan Publishing Company, 2001.

Tanenbaum, A. *Structured Computer Organization*, 4th ed., Englewood Cliffs, NJ: Prentice Hall, 1999.

Tanenbaum, A., & Woodhull, S. *Operating Systems, Design and Implementation*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1997.

REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. Which is faster, SRAM or DRAM?
2. What are the advantages of using DRAM for main memory?
3. Name three different applications where ROMs are often used.
4. Explain the concept of a memory hierarchy. Why did your authors choose to represent it as a pyramid?
5. Explain the concept of locality of reference and state its importance to memory systems.
6. What are the three forms of locality?
7. Give two noncomputer examples of the concept of cache.
8. Which of L1 or L2 cache is faster? Which is smaller? Why is it smaller?
9. Cache is accessed by its _____, whereas main memory is accessed by its _____.
10. What are the three fields in a direct mapped cache address? How are they used to access a word located in cache?

11. How does associative memory differ from regular memory? Which is more expensive and why?
12. Explain how fully associative cache is different from direct mapped cache.
13. Explain how set associative cache combines the ideas of direct and fully associative cache.
14. Direct mapped cache is a special case of set associative cache where the set size is 1. So fully associative cache is a special case of set associative cache where the set size is ___.
15. What are the three fields in a set associative cache address and how are they used to access a location in cache?
16. Explain the four cache replacement policies presented in this chapter.
17. Why is the optimal cache replacement policy important?
18. What is the worst-case cache behavior that can develop using LRU and FIFO cache replacement policies?
19. What, exactly, is effective access time (EAT)?
20. Explain how to derive an effective access time formula.
21. When does caching behave badly?
22. What is a dirty block?
23. Describe the advantages and disadvantages of the two cache write policies.
24. What is the difference between a virtual memory address and a physical memory address? Which is larger? Why?
25. What is the objective of paging?
26. Discuss the pros and cons of paging.
27. What is a page fault?
28. What causes internal fragmentation?
29. What are the components (fields) of a virtual address?
30. What is a TLB and how does it improve EAT?
31. What are the advantages and disadvantages of virtual memory?
32. When would a system ever need to page its page table?
33. What causes external fragmentation and how can it be fixed?

EXERCISES

◆ 1. Suppose a computer using direct mapped cache has 2^{20} words of main memory and a cache of 32 blocks, where each cache block contains 16 words.

- ◆ a) How many blocks of main memory are there?
- ◆ b) What is the format of a memory address as seen by the cache, that is, what are the sizes of the tag, block, and word fields?
- ◆ c) To which cache block will the memory reference $0DB63_{16}$ map?

2. Suppose a computer using direct mapped cache has 2^{32} words of main memory and a cache of 1024 blocks, where each cache block contains 32 words.
   a) How many blocks of main memory are there?
   b) What is the format of a memory address as seen by the cache, that is, what are the sizes of the tag, block, and word fields?
   c) To which cache block will the memory reference $000063FA_{16}$ map?

◆ 3. Suppose a computer using fully associative cache has 2^{16} words of main memory and a cache of 64 blocks, where each cache block contains 32 words.
   ◆ a) How many blocks of main memory are there?
   ◆ b) What is the format of a memory address as seen by the cache, that is, what are the sizes of the tag and word fields?
   ◆ c) To which cache block will the memory reference $F8C9_{16}$ map?

4. Suppose a computer using fully associative cache has 2^{24} words of main memory and a cache of 128 blocks, where each cache block contains 64 words.
   a) How many blocks of main memory are there?
   b) What is the format of a memory address as seen by the cache, that is, what are the sizes of the tag and word fields?
   c) To which cache block will the memory reference $01D872_{16}$ map?

◆ 5. Assume a system's memory has 128M words. Blocks are 64 words in length and the cache consists of 32K blocks. Show the format for a main memory address assuming a 2-way set associative cache mapping scheme. Be sure to include the fields as well as their sizes.

6. A 2-way set associative cache consists of four sets. Main memory contains 2K blocks of eight words each.
   a) Show the main memory address format that allows us to map addresses from main memory to cache. Be sure to include the fields as well as their sizes.
   b) Compute the hit ratio for a program that loops 3 times from locations 8_{10} to 51_{10} in main memory. You may leave the hit ratio in terms of a fraction.

7. Suppose a computer using set associative cache has 2^{16} words of main memory and a cache of 32 blocks, and each cache block contains 8 words.
   a) If this cache is 2-way set associative, what is the format of a memory address as seen by the cache, that is, what are the sizes of the tag, set, and word fields?
   b) If this cache is 4-way set associative, what is the format of a memory address as seen by the cache?

8. Suppose a computer using set associative cache has 2^{21} words of main memory and a cache of 64 blocks, where each cache block contains 4 words.
   a) If this cache is 2-way set associative, what is the format of a memory address as seen by the cache, that is, what are the sizes of the tag, set, and word fields?
   b) If this cache is 4-way set associative, what is the format of a memory address as seen by the cache?

*9. Suppose we have a computer that uses a memory address word size of 8 bits. This computer has a 16-byte cache with 4 bytes per block. The computer accesses a number of memory locations throughout the course of running a program.

Suppose this computer uses direct-mapped cache. The format of a memory address as seen by the cache is shown here:



The system accesses memory addresses (in hex) in this exact order: 6E, B9, 17, E0, 4E, 4F, 50, 91, A8, A9, AB, AD, 93, and 94. The memory addresses of the first four accesses have been loaded into the cache blocks as shown below. (The contents of the tag are shown in binary and the cache "contents" are simply the address stored at that cache location.)

	Tag Contents	Cache Contents (represented by address)
Block 0	1110	E0
		E1
		E2
		E3

	Tag Contents	Cache Contents (represented by address)
Block 1	0001	14
		15
		16
		17

	Tag Contents	Cache Contents (represented by address)
Block 2	1011	B8
		B9
		BA
		BB

	Tag Contents	Cache Contents (represented by address)
Block 3	0110	6C
		6D
		6E
		6F

   a) What is the hit ratio for the entire memory reference sequence given above?
   b) What memory blocks will be in the cache after the last address has been accessed?

10. A direct-mapped cache consists of eight blocks. Main memory contains 4K blocks of eight words each. Access time for the cache is 22ns and the time required to fill a cache slot from main memory is 300ns. (This time allows us to determine the block is missing and bring it into cache.) Assume a request is always started in parallel to both cache and to main memory (so if it is not found in cache, we do not have to add this cache search time to the memory access). If a block is missing from cache, the entire block is brought into the cache and the access is restarted. Initially, the cache is empty.
   a) Show the main memory address format that allows us to map addresses from main memory to cache. Be sure to include the fields as well as their sizes.
   b) Compute the hit ratio for a program that loops 4 times from locations 0 to 67_{10} in memory.
   c) Compute the effective access time for this program.

11. Consider a byte-addressable computer with 24-bit addresses, a cache capable of storing a total of 64KB of data, and blocks of 32 bytes. Show the format of a 24-bit memory address for:
   a) direct mapped
   b) associative

c) 4-way set associative

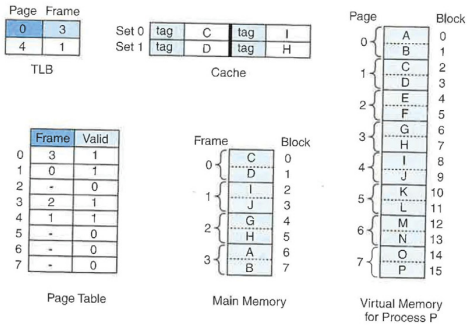
12. Suppose a process page table contains the entries shown below. Using the format shown in Figure 6.15a, indicate where the process pages are located in memory.

Frame	Valid Bit
1	1
-	0
0	1
3	1
-	0
-	0
2	1
-	0

• **13.** Suppose a process page table contains the entries shown below. Using the format shown in Figure 6.15a, indicate where the process pages are located in memory.

Frame	Valid Bit
-	0
3	1
-	0
-	0
2	1
0	1
-	0
1	1

*14. You have a virtual memory system with a two-entry TLB, a 2-way set associative cache, and a page table for a process P. Assume cache blocks of 8 words and page size of 16 words. In the system below, main memory is divided into blocks, where each block is represented by a letter. Two blocks equal one frame.



Given the system state as depicted above, answer the following questions:

a) How many bits are in a virtual address for process P? Explain.

b) How many bits are in a physical address? Explain.

c) Show the address format for virtual address 18_{10} (specify field name and size) that would be used by the system to translate to a physical address and then translate this virtual address into the corresponding physical address. (Hint: convert 18 to its binary equivalent and divide it into the appropriate fields.) Explain how these fields are used to translate to the corresponding physical address.

d) Given virtual address 6_{10} converts to physical address 54_{10}. Show the format for a physical address (specify the field names and sizes) that is used to determine the cache location for this address. Explain how to use this format to determine where physical address 54 would be located in cache. (Hint: Convert 54 to binary and divide it into the appropriate fields.)

e) Given virtual address 25_{10} is located on virtual page 1, offset 9. Indicate exactly how this address would be translated to its corresponding physical address and how the data would be accessed. Include in your explanation how the TLB, the page table, cache, and memory are used.

15. Given a virtual memory system with a TLB, a cache, and a page table, assume the following:

- A TLB hit requires 5ns.
- A cache hit requires 12ns.
- A memory reference requires 25ns.
- A disk reference requires 200ms (this includes updating the page table, cache, and TLB).
- The TLB hit ratio is 90%.
- The cache hit rate is 98%.
- The page fault rate is .001%.
- On a TLB or cache miss, the time required for access includes a TLB and/or cache update, but the access is *not* restarted.
- On a page fault, the page is fetched from disk, all updates are performed, but the access *is* restarted.
- All references are sequential (no overlap, nothing done in parallel).

For each of the following, indicate whether or not it is possible. If it is possible, specify the time required for accessing the requested data.

a) TLB hit, cache hit

b) TLB miss, page table hit, cache hit

c) TLB miss, page table hit, cache miss

d) TLB miss, page table miss, cache hit

e) TLB miss, page table miss

Write down the equation to calculate the effective access time.

16. A system implements a paged virtual address space for each process using a one-level page table. The maximum size of virtual address space is 16MB. The page table for the running process includes the following valid entries (the → notation indicates that a virtual page maps to the given page frame, that is, it is located in that frame):

Virtual page 2 → Page frame 4 Virtual page 4 → Page frame 9

Virtual page 1 → Page frame 2 Virtual page 3 → Page frame 16

Virtual page 0 → Page frame 1

The page size is 1024 bytes and the maximum physical memory size of the machine is 2MB.

a) How many bits are required for each virtual address?

b) How many bits are required for each physical address?

c) What is the maximum number of entries in a page table?

d) To which physical address will the virtual address 1524_{10} translate?

e) Which virtual address will translate to physical address 1024_{10}?

17. a) If you are a computer builder trying to make your system as price-competitive as possible, what features and organization would you select for its memory hierarchy?

b) If you are a computer buyer trying to get the best performance from a system, what features would you look for in its memory hierarchy?

*18. Consider a system that has multiple processors where each processor has its own cache, but main memory is shared among all processors.

a) Which cache write policy would you use?

b) **The Cache Coherency Problem.** With regard to the system just described, what problems are caused if a processor has a copy of memory block *A* in its cache and a second processor, also having a copy of *A* in its cache, then updates main memory block *A*? Can you think of a way (perhaps more than one) of preventing this situation, or lessening its effects?

*19. Pick a specific architecture (other than the one covered in this chapter). Do research to find out how your architecture approaches the concepts introduced in this chapter, as was done for Intel's Pentium.

83

"Who is General Failure and why is he reading my disk?"

—Anonymous

CHAPTER 7 Input/Output and Storage Systems

7.1 INTRODUCTION

A computer is of no use without some means of getting data into it and information out of it. Having a computer that does not do this effectively or efficiently is little better than having no computer at all. When processing time exceeds user "think time," users will complain that the computer is "slow." Sometimes this slowness can have a substantial productivity impact, measured in hard currency. More often than not, the root cause of the problem is not in the processor or the memory but in how the system processes its input and output (I/O).

I/O is more than just file storage and retrieval. A poorly functioning I/O system can have a ripple effect, dragging down the entire computer system. In the preceding chapter, we described virtual memory, that is, how systems page blocks of memory to disk to make room for more user processes in main memory. If the disk system is sluggish, process execution slows down, causing backlogs in CPU as well as disk queues. The easy solution to the problem is to simply throw more resources at the system. Buy more main storage. Buy a faster processor. If we're in a particularly Draconian frame of mind, we could simply limit the number of concurrent processes!

Such measures are wasteful, if not plain irresponsible. If we really understand what's happening in a computer system we can make the best use of the resources available, adding costly resources only when absolutely necessary. The goal of this chapter is to present you with a survey of ways in which I/O and storage capacities can be optimized, allowing you to make informed storage choices. Our highest hope is that you might be able to use this information as a springboard for further study—and perhaps, even innovation.

7.2 AMDAHL'S LAW

Each time a (particular) microprocessor company announces its latest and greatest CPU, headlines sprout across the globe heralding this latest leap forward in technology. Cyberphiles the world over would agree that such advances are laudable and deserving of fanfare. However, when similar advances are made in I/O technology, the story is apt to appear on page 67 of some obscure trade magazine. Under the blare of media hype, it is easy to lose sight of the integrated nature of computer systems. A 40% speedup for one component certainly will not make the entire system 40% faster, despite media implications to the contrary.

In 1967, George Amdahl recognized the interrelationship of all components with the overall efficiency of a computer system. He quantified his observations in a formula, which is now known as Amdahl's Law. In essence, Amdahl's Law states that the overall *speedup* of a computer system depends on both the speedup in a particular component and how much that component is used by the system. In symbols:

$$S = \frac{1}{(1-f) + f/k}$$

where

S is the speedup;

f is the fraction of work performed by the faster component; and

k is the speedup of a new component.

Let's say that most of your daytime processes spend 70% of their time running in the CPU and 30% waiting for service from the disk. Suppose also that someone is trying to sell you a processor array upgrade that is 50% faster than what you have and costs $10,000. The day before, someone had called you on the phone offering you a set of disk drives for $7,000. These new disks promise two and a half times the throughput of your existing disks. You know that the system performance is starting to degrade, so you need to do something. Which would you choose to yield the best performance improvement for the least amount of money?

For the processor option we have:

$$f = .70, k = 1.5, \text{ so } \quad S = \frac{1}{(1-0.7) + 0.7/1.5} = 1.30.$$

We therefore appreciate a total speedup of 130% with the new processor for $10,000.

For the disk option we have:

$$f = .30, k = 2.5, \text{ so } \quad S = \frac{1}{(1-0.3) + 0.3/2.5} \approx 1.22.$$

The disk upgrade gives us a speedup of 122% for $7,000.

All things being equal, it is a close decision. Each 1% of performance improvement resulting from the processor upgrade costs about $333. Each 1% with the disk upgrade costs about $318. This makes the disk upgrade a slightly better choice, based solely upon dollars spent per performance improvement percentage point. Certainly, other factors would influence your decision. For example, if your disks are nearing the end of their expected life, or if you're running out of disk space, you might consider the disk upgrade even if it were to cost more than the processor upgrade.

Before you make that disk decision, however, you need to know your options. The sections that follow will help you to gain an understanding of general I/O architecture, with special emphasis on disk I/O. Disk I/O follows closely behind the CPU and memory in determining the overall effectiveness of a computer system.

7.3 I/O ARCHITECTURES

We will define input/output as a subsystem of components that moves coded data between external devices and a host system, consisting of a CPU and main memory. I/O subsystems include, but are not limited to:

- Blocks of main memory that are devoted to I/O functions
- Buses that provide the means of moving data into and out of the system
- Control modules in the host and in peripheral devices
- Interfaces to external components such as keyboards and disks
- Cabling or communications links between the host system and its peripherals

Figure 7.1 shows how all of these components can fit together to form an integrated I/O subsystem. The I/O modules take care of moving data between main memory and a particular device interface. Interfaces are designed specifically to communicate with certain types of devices, such as keyboards, disks, or printers. Interfaces handle the details of making sure that devices are ready for the next batch of data, or that the host is ready to receive the next batch of data coming in from the peripheral device.

The exact form and meaning of the signals exchanged between a sender and a receiver is called a *protocol*. Protocols comprise command signals, such as "Printer reset"; status signals, such as "Tape ready"; or data-passing signals, such as "Here are the bytes you requested." In most data-exchanging protocols, the receiver must acknowledge the commands and data sent to it or indicate that it is ready to receive data. This type of protocol exchange is called a *handshake*.

External devices that handle large blocks of data (such as printers, and disk and tape drives) are often equipped with buffer memory. Buffers allow the host system to send large quantities of data to peripheral devices in the fastest manner possible, without having to wait until slow mechanical devices have actually written the data. Dedicated memory on disk drives is usually of the fast cache variety, whereas printers are usually provided with slower RAM.

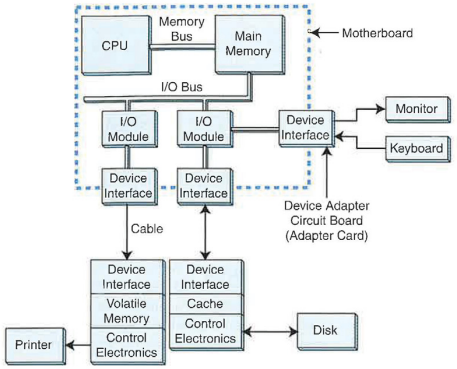


FIGURE 7.1 A Model I/O Configuration

Device control circuits take data to or from on-board buffers and assure that it gets where it's going. In the case of writing to disks, this involves making certain that the disk is positioned properly so that the data is written to a particular location. For printers, these circuits move the print head or laser beam to the next character position, fire the head, eject the paper, and so forth.

Disk and tape are forms of *durable storage*, so-called because data recorded on them lasts longer than it would in volatile main memory. However, no storage method is permanent. The expected life of data on these media is approximately five years for magnetic media and as much as 100 years for optical media.

7.3.1 I/O Control Methods

Computer systems employ any of four general I/O control methods. These methods are programmed I/O, interrupt-driven I/O, direct memory access, and channel-attached I/O. Although one method isn't necessarily better than another, the manner in which a computer controls its I/O greatly influences overall system design and performance. The objective is to know when the I/O method employed by a particular computer architecture is appropriate to how the system will be used.

Programmed I/O

Systems using *programmed I/O* devote at least one register for the exclusive use of each I/O device. The CPU continually monitors each register, waiting for data to arrive. This is called *polling*. Thus, programmed I/O is sometimes referred to as *polled I/O*. Once the CPU detects a "data ready" condition, it acts according to instructions programmed for that particular register.

The benefit of using this approach is that we have programmatic control over the behavior of each device. Program changes can make adjustments to the number and types of devices in the system as well as their polling priorities and intervals. Constant register polling, however, is a problem. The CPU is in a continual "busy wait" loop until it starts servicing an I/O request. It doesn't do any useful work until there is I/O to process. Owing to these limitations, programmed I/O is best suited for special-purpose systems such as automated teller machines and systems that control or monitor environmental events.

Interrupt-Driven I/O

Interrupt-driven I/O can be thought of as the converse of programmed I/O. Instead of the CPU continually asking its attached devices whether they have any input, the devices tell the CPU when they have data to send. The CPU proceeds with other tasks until a device requesting service interrupts it. Interrupts are usually signaled with a bit in the CPU flags register called an *interrupt flag*.

Once the interrupt flag is set, the operating system interrupts whatever program is currently executing, saving that program's state and variable information. The system then fetches the *address vector* that points to the address of the I/O service routine. After the CPU has completed servicing the I/O, it restores the information it saved from the program that was running when the interrupt occurred, and the program execution resumes.

Interrupt-driven I/O is similar to programmed I/O in that the service routines can be modified to accommodate hardware changes. Because vectors for the various types of hardware are usually kept in the same locations in systems running the same type and level of operating system, these vectors are easily changed to point to vendor-specific code. For example, if someone comes up with a new type of disk drive that is not yet supported by a popular operating system, the manufacturer of that disk drive may update the disk I/O vector to point to code particular to that disk drive. Unfortunately, some of the early DOS-based virus writers also used this idea. They would replace the DOS I/O vectors with pointers to their own nefarious code, eradicating many systems in the process. Many of today's popular operating systems employ interrupt-driven I/O. Fortunately, these operating systems have mechanisms in place to safeguard against this kind of vector manipulation.

Direct Memory Access

With both programmed I/O and interrupt-driven I/O, the CPU moves data to and from the I/O device. During I/O, the CPU runs instructions similar to the following pseudocode:

```
WHILE More-input AND NOT Error
      ADD 1 TO Byte-count
      IF Byte-count > Total-bytes-to-be-transferred THEN
        EXIT
```

```
            ENDIF
            Place byte in destination buffer
            Raise byte-ready signal
            Initialize timer
            REPEAT
              WAIT
            UNTIL Byte-acknowledged, Timeout, OR Error
    ENDWHILE
```

Clearly, these instructions are simple enough to be programmed in a dedicated chip. This is the idea behind *direct memory access* (*DMA*). When a system uses DMA, the CPU offloads execution of tedious I/O instructions. To effect the transfer, the CPU provides the DMA controller with the location of the bytes to be transferred, the number of bytes to be transferred, and the destination device or memory address. This communication usually takes place through special I/O registers on the CPU. A sample DMA configuration is shown in Figure 7.2.

Once the proper values are placed in memory, the CPU signals the DMA subsystem and proceeds with its next task, while the DMA takes care of the details of the I/O. After the I/O is complete (or ends in error), the DMA subsystem signals the CPU by sending it another interrupt.

As you can see by Figure 7.2, the DMA controller and the CPU share the memory bus. Only one of them at a time can have control of the bus, that is, be the *bus master*. Generally, I/O takes priority over CPU memory fetches for program instructions and data because many I/O devices operate within tight timing parameters. If they detect no activity within a specified period, they *timeout* and

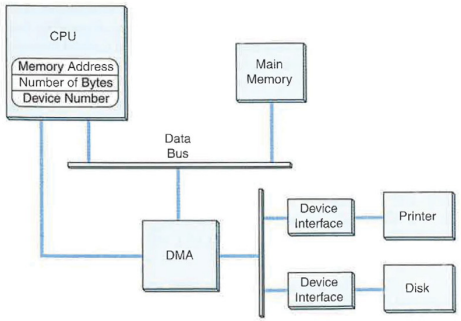


FIGURE 7.2 An Example DMA Configuration

abort the I/O process. To avoid device timeouts, the DMA uses memory cycles that would otherwise be used by the CPU. This is called *cycle stealing*. Fortunately, I/O tends to create *bursty* traffic on the bus: data is sent in blocks, or clusters. The CPU should be granted access to the bus between bursts, though this access may not be of long enough duration to spare the system from accusations of "crawling during I/O."

Channel I/O

Programmed I/O transfers data one byte at a time. Interrupt-driven I/O can handle data one byte at a time or in small blocks, depending on the type of device participating in the I/O. Slower devices such as keyboards generate more interrupts per number of bytes transferred than disks or printers. DMA methods are all block-oriented, interrupting the CPU only after completion (or failure) of transferring a group of bytes. After the DMA signals the I/O completion, the CPU may give it the address of the next block of memory to be read from or written to. In the event of failure, the CPU is solely responsible for taking appropriate action. Thus, DMA I/O requires only a little less CPU participation than does interrupt-driven I/O. Such overhead is fine for small, single-user systems; however, it does not scale well to large, multi-user systems such as mainframe computers. Most mainframes use an intelligent type of DMA interface known as an *I/O channel*.

With *channel I/O*, one or more I/O processors control various I/O pathways called *channel paths*. Channel paths for "slow" devices such as terminals and printers can be combined (*multiplexed*), allowing management of several of these devices through only one controller. On IBM mainframes, a multiplexed channel path is called a *multiplexor channel*. Channels for disk drives and other "fast" devices are called *selector channels*.

I/O channels are driven by small CPUs called *I/O processors* (*IOPs*), which are optimized for I/O. Unlike DMA circuits, IOPs have the ability to execute programs that include arithmetic-logic and branching instructions. Figure 7.3 shows a simplified channel I/O configuration.

IOPs execute programs that are placed in main system memory by the host processor. These programs, consisting of a series of *channel command words* (*CCWs*), include not only the actual transfer instructions, but also commands that control the I/O devices. These commands include such things as various kinds of device initializations, printer page ejects, and tape rewind commands, to name a few. Once the I/O program has been placed in memory, the host issues a *start subchannel* command (SSCH), informing the IOP of the location in memory where the program can be found. After the IOP has completed its work, it places completion information in memory and sends an interrupt to the CPU. The CPU then obtains the completion information and takes action appropriate to the return codes.

The principal distinction between standalone DMA and channel I/O lies in the intelligence of the IOP. The IOP negotiates protocols, issues device commands, translates storage coding to memory coding, and can transfer entire files or groups of files independent of the host CPU. The host has only to create the program instructions for the I/O operation and tell the IOP where to find them.

98