

entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory (Figure 7.14). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

$$\text{effective access time} = 0.80 \times 100 + 0.20 \times 200 = 120 \text{ nanoseconds}$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have effective access time = $0.99 \times 100 + 0.01 \times 200 = 101$ nanoseconds. This increased hit rate produces only a 1 percent slowdown in access time. As we noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated

frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work. A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 9. TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

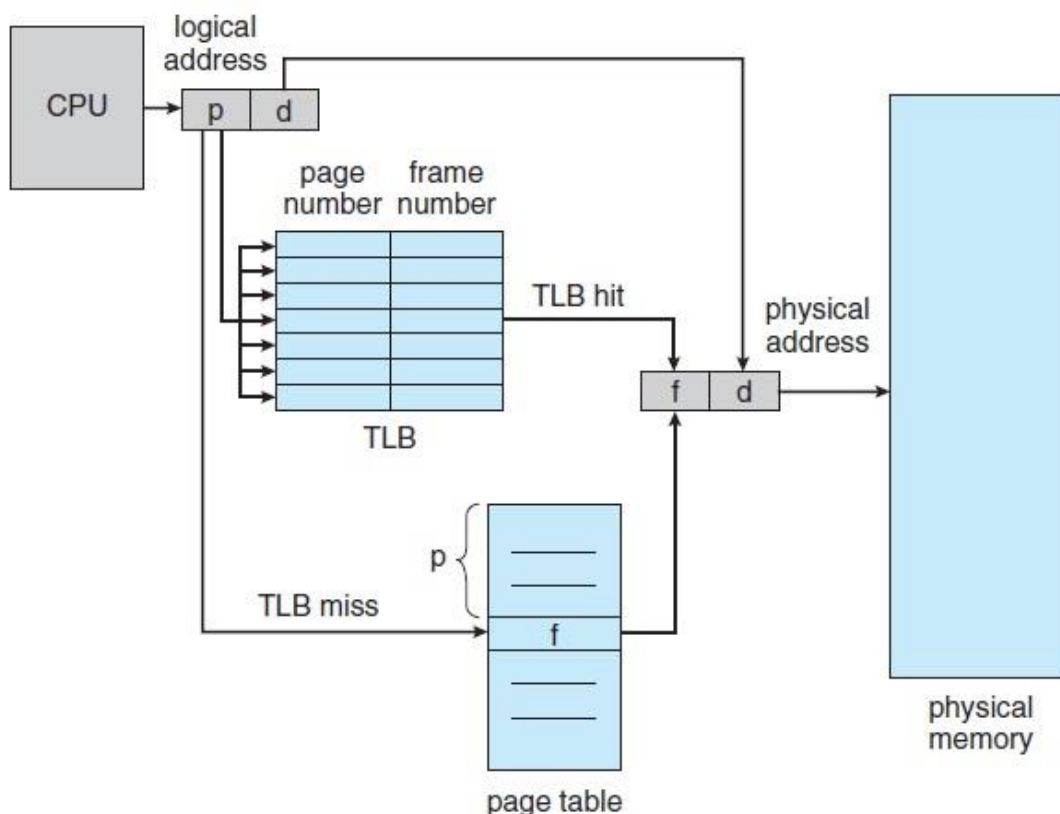


Figure 7-14 Paging hardware with TLB

7.5.3. Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid*, the associated page is in the process’s logical address space and is thus a legal (or valid) page. When the bit is set to *invalid*, the page is not in the process’s logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 7.15. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.