**Replacement strategies**. Sooner or later, cache will become full. When this occurs, a mechanism should be there to replace a word with the newly accessed data from memory. In general, there are three main strategies for determining which word should be swapped out from the cache; they are called *random, least frequently used*, and *least recently used* replacement policies.

***Random Replacement***. This method picks a word at random and replaces that word with the newly accessed data. This method is easy to implement in hardware, and it is faster than most other algorithms.
The disadvantage is that the words most likely to be used again have as much of a chance of being swapped out as a word that is likely not to be used again. This disadvantage diminishes as the cache size increases.

***Least Frequently Used***. This method replaces the data that are used the least. It assumes that data that are not referenced frequently are not needed as much. For each word, a counter is kept for the total number of
times the word has been used since it was brought into the cache. The word with the lowest count is the word to be swapped out. The advantage of this method is that a frequently used word is more likely to remain in cache than a word that has not been used often. One disadvantage is that words that have recently been brought into the cache have a low count total, despite the fact that they are likely to be used again. Another disadvantage is that this method is more difficult to implement in terms of hardware and is thus more expensive.

***Least Recently Used***. This method has the best performance per cost compared with the other techniques and is often implemented in real-world systems. The idea behind this replacement method is that a word
that has not been used for a long period of time has a lesser chance of being needed in the near future according to the property of temporal locality. Thus, this method retains words in the cache that are more likely to be used again. To do this, a mechanism is used to keep track of which words have been accessed most recently. The word that will be swapped out is the word that has not been used for the longest period of time. One way to implement such a mechanism is to assign a counter to each word in the cache. Each time the cache is accessed, each word's counter is incremented, and the word's counter that was accessed is reset to zero. In this manner, the word with the highest count is the one that was least recently used.

## Virtual Memory

Another technique used to improve system performance is called *virtual memory*. As the name implies, virtual memory is the illusion of a much larger main memory size (logical view) than what actually exists (physical view). Prior to the advent of virtual memory, if a program's address space exceeded the actual available memory, the programmer was responsible for breaking up the program into smaller pieces called
*overlays*. Each overlay then could fit in main memory. The basic process was to store all these overlays in secondary memory, such as on a disk, and to load individual overlays into main memory as they were needed.
This process required knowledge of where the overlays were to be stored on disk, knowledge of input/output operations involved with accessing the overlays, and keeping track of the entire overlay process. This was a very complex and tedious process that made the complexity of programming a computer even more difficult.
The concept of virtual memory was created to relieve the programmer of this burden and to let the computer manage this process. Virtual memory allows the user to write programs that grow beyond the bounds of physical memory and still execute properly. It also allows for multiprogramming, by which main memory is shared among many users on a dynamic basis. With multiprogramming, portions of several programs are placed in the main memory at the same time, and the processor switches its time back and forth among these programs. The processor executes one program for a brief period of time (called a *quantum* or *time-slice*) and then switches to another program; this process continues until each program is completed. When virtual memory is used, the addresses used by the programmer are seen by the system as *virtual addresses*, which are so called because they are mapped onto the addresses of physical memory and therefore do not access the same physical memory address from one execution of an instruction to the next. Virtual addresses, also called *logical addresses*, are generated by the processor during the compile time and are translated into physical addresses at run time. The two main methods for achieving a virtual memory environment are *paging* and *segmentation*. Each is explained next.

**Paging.** Paging is the technique of breaking a program (referred to in the following as process) into smaller blocks of identical size and storing these blocks in secondary storage in the form of *pages*. By taking advantage of the locality of reference, these pages can then be loaded into main memory, a few at a time, into blocks of the same size called *frames* and executed just as if the entire process were in memory.
For this method to work properly, each process must maintain a *page table* in main memory. Figure 2.53 shows how a paging scheme works. The base register, which each process has, points to the beginning of the process's page table. Page tables have an entry for each page that the process contains. These entries usually contain a *load* field of one bit, an *address* field, and an *access* field. The load field specifies

whether the page has been brought into main memory. The address field specifies the frame number of the frame into which the page is loaded. The address of the page within main memory is evaluated by multiplying the frame number and the frame size. (Since frame size is usually a power of 2, shifting is often used for multiplying frame number by frame size.) If a page has not been loaded, the address of the page within secondary memory is held in this field. The access field specifies the type of operation that can be performed on a block. It determines whether a block is read only, read/write, or executable.

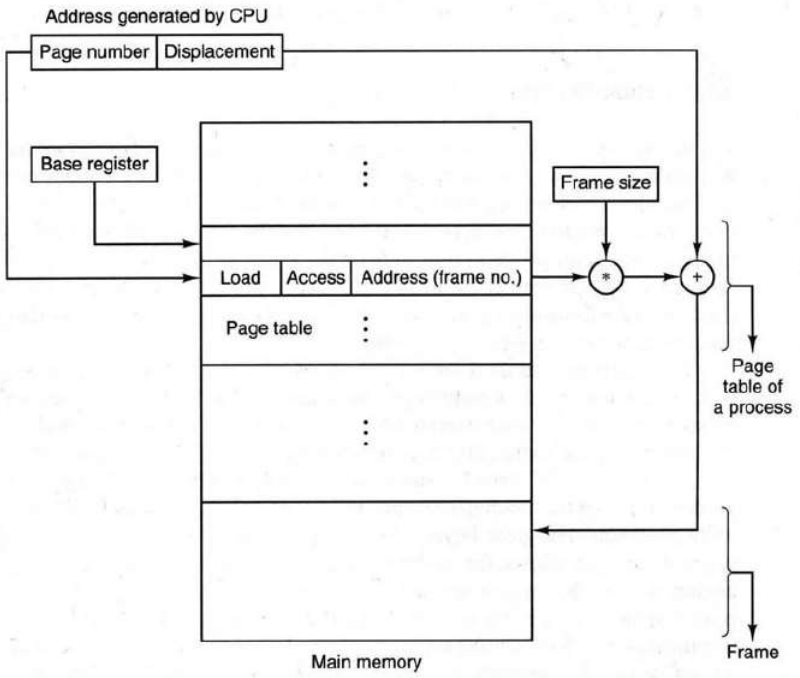


Figure 2.53 Using page table to convert a virtual address to a physical address.

When an access to a variable or an instruction that is not currently loaded into the memory is encountered, a *page fault* occurs, and the page that contains the necessary variable or instruction is brought into the memory. The page is stored in a free frame, if one exists. If a free frame does not exist, one of the process's own frames must be given up, and the new page will be stored in its place. Which frame is given up and whether the old page is written back to secondary storage depend on which of several page replacement algorithms (discussed later in this section) is used.

As an example, Figure 2.54 shows the contents of page tables for two processes, process 1 and process 2. Process 1 consists of three pages, $P0$, $P1$, and $P2$, whereas process 2 has only two pages, $P0$ and $P1$. Assume

that all the pages of process 1 have read access only and the pages of process 2 have read/write access; this is denoted by *R* and *W* in each page table. Because each frame has 4096 (4K) bytes, the physical address of
the beginning of each frame is computed by the product of the frame number and 4096. Therefore, given that *P*0 and *P*2 of process 1 are loaded into frames 1 and 3, their beginning address in main memory will be 4K = (1 * 4096) and 12K = (3 * 4096), respectively.

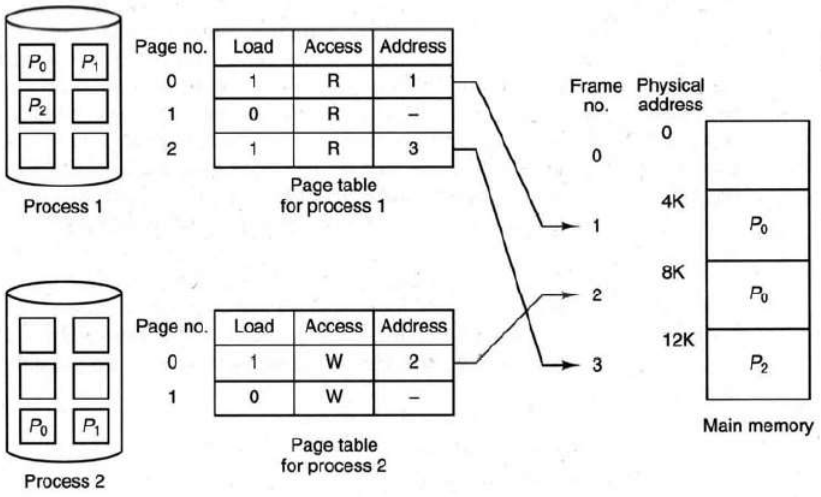


Figure 2.54 Page tables for two different processes, Process 1 and Process 2.

The process of converting a virtual address to a physical address can be sped up by using a high-speed lookup table called a *translation lookaside buffer* (TLB). The page number of the virtual address is fed to the TLB where it is translated to a frame number of the physical address. The TLB can be implemented as an associative memory that has an entry for each of the most recently or likely referenced pages. Each entry contains a page number and other relevant information similar to the page table entry, essentially the frame number and access type. For a given page number, an entry of the TLB that matches (a hit) this page number is used to provide the corresponding frame number. If a match cannot be found in the TLB (a miss), the page table of the corresponding process will be located and used to produce the frame number.
The efficiency of the virtual memory system depends on minimizing the number of page faults. Because the access time of secondary memory is much higher than the access time of main memory, an excessive number of page faults can slow the system dramatically. When a page fault occurs, a page in the main memory must be located and identified as one not needed at the present time so that it can be written back to

the secondary memory. Then the requested page can be loaded into this newly freed frame of main memory.
Obviously, paging increases the processing time of a process substantially, because two disk accesses would be required along with the execution of a replacement algorithm. There is an alternative, however,
which at times can reduce the number of the disk accesses to just one. This reduction is achieved by adding to the hardware an extra bit to each frame, called a *dirty* bit (also called an *inconsistent* bit). If some modification has taken place to a particular frame, the corresponding dirty bit is set to 1. If the dirty bit for frame *f* is 1, for example, and in order to create an available frame, *f* has been chosen as the frame to swap
out, then two disk accesses would be required. If the dirty bit is 0 (meaning that there were no modifications on *f* since it was last loaded), there would be no need to write *f* back to disk. Because the original state of *f* is still on disk (remember that the frames in main memory contain copies of the pages in secondary memory) and no modifications have been made to *f* while in main memory, the page frame containing *f* can simply be overwritten by the newly requested page.
Most replacement algorithms consider the principle of locality when selecting a frame to replace. The principle of locality states that over a given amount of time the addresses generated will fall within a small portion of the virtual address space and that these generated addresses will change slowly with time. Two possible replacement algorithms are
1. First in, first out (FIFO)
2. Least recently used (LRU)

Before the discussion of replacement algorithms, you should note that the efficiency of the algorithm is based on the page size (*Z*) and the number of pages (*N*) the main memory (*M*1) can contain. If $Z = 100$ bytes and $N = 3$, then $M1 = N * Z = 3 * 100 = 300$ bytes.
Another concern with replacement algorithms is the page fault frequency (*PF*). The PF is determined by the number of page faults (*F*) that occurs in an entire execution of a process divided by *F* plus the number of no-fault references (*S*): $PF = F / (S + F)$. The *PF* should be as low a percentage as possible in order to minimize disk accesses. The *PF* is affected by page size and the number of page frames.

**First In, First Out**. First in, first out (FIFO) is one of the simplest algorithms to employ. As the name implies, the first page loaded will be the first page to be removed from main memory. Figure 2.55a demonstrates how FIFO works as well as how the page fault frequency (*PF*) is determined by using a table.

The number of rows in the table represents the number of available frames, and the columns represent each reference to a page. These references come from a given reference sequence 0, 1, 2, 0, 3, 2, 0, 1, 2, 4, 0, where the first reference is to page 0, the second is to page 1, the third is to page 2, and so on. When a page fault occurs, the corresponding page number is put in the top row, representing its precedence, and marked with an asterisk. The previous pages in the table are moved down. Once the page frames are filled and a page fault occurs, the page in the bottom page frame is removed or swapped out. (Keep in mind that this
table is used only to visualize a page's current precedence. The movement of these pages does not imply that they are actually being shifted around in *M*1. A counter can be associated with each page to determine the oldest page.)
Once the last reference in the reference sequence is loaded, the *PF* can be calculated. The number of asterisks appearing in the top row equals page faults (*F*) and the items in the top row that do not contain an asterisk equals success (*S*). Considering Figure 2.55, when *M*1 contains three frames, *PF* is 81% for the above reference sequence. When *M*1 contains four frames, *PF* reduces to 54%. That is, *PF* is improved by increasing the number of page frames to 4.

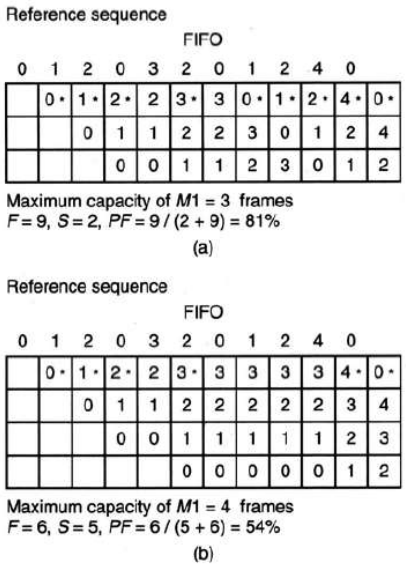


Figure 2.55 Performance of the FIFO replacement technique on two different memory configurations.

The disadvantage of FIFO is that it may significantly increase the time it takes for a process to execute because it does not take into consideration the principle of locality and consequently may replace heavily used frames as well as rarely used frames with equal probability. For example, if an early frame contains a global variable that is in constant use, this frame will be one of the first to be replaced. During the next access

to the variable, another page fault will occur, and the frame will have to be reloaded, replacing yet another page.

***Least Recently Used***. The least recently used (LRU) method will replace the frame that has not been used for the longest time. In this method, when a page is referenced that is already in *M*1, it is placed in the top row and the other pages are shifted down. In other words, the most used pages are kept at the top. See Figure 2.56a. An improvement of *PF* is made using LRU by adding a page frame to *M*1. See Figure 2.56b.
In general, LRU is more efficient than FIFO, but it requires more hardware (usually a counter or a stack) to keep track of the least and most recently used pages.

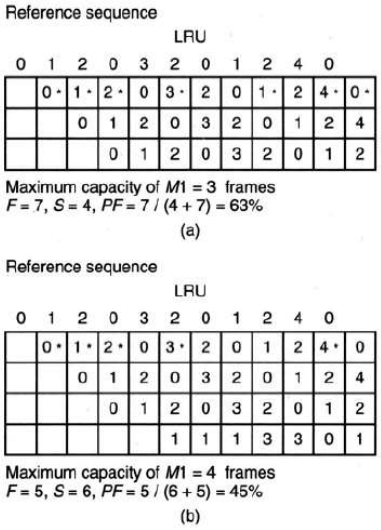


Figure 2.56 Performance of LRU replacement technique on two different memory configurations.

**Segmentation**. Another method of swapping between secondary and main memory is called *segmentation*.
In segmentation, a program is broken into variable-length sections known as *segments*. For example, a segment can be a data set or a function within the program. Each process keeps a segment table within main memory that contains basically the same information as the page table. However, unlike pages, segments have variable lengths, and they can start anywhere in the memory; therefore, removing one segment from main memory may not provide enough space for another segment.
There are several strategies for placing a given segment into the main memory. Among the most well known strategies are *first fit*, *best fit*, and *worst fit*. Each of these strategies maintains a list that represents the size and position of the free

storage blocks in the main memory. This list is used for finding a suitable block size for the given segment. The following is an explanation.

***First Fit***. This strategy puts the given segment into the first suitable free storage. It searches through the free storage list until it finds a block of free storage that is large enough for the segment; then it allocates a block of memory for the segment.
The main advantage of this strategy is that it encourages free storage areas to become available at high memory addresses by assigning segments to the low-memory addresses whenever possible. However, this
strategy will produce free areas that may be too small to hold a segment. This phenomenon is known as *fragmentation*. When fragmentation occurs, eventually some sort of compaction algorithm will have to be
run to collect all the small free areas into one large one. This causes some overhead, which degrades the performance.

***Best Fit***. This strategy allocates the smallest available free storage block that is large enough to hold the segment. It searches through the free storage list until it finds the smallest block of storage that is large enough for the segment. To prevent searching the entire list, the free storage list is usually sorted according to the increasing block size. Unfortunately, like first fit, this strategy also causes fragmentation. In fact, it may create many small blocks that are almost useless.

***Worst Fit***. This strategy allocates the largest available free storage block for the segment. It searches the free storage list for the largest block. The list is usually sorted according to the decreasing block size.
Again, the worst fit, like the other two strategies, causes fragmentation. However, in contrast to first fit and best fit, worst fit reduces the number of small blocks by always allocating the largest block for the segment.