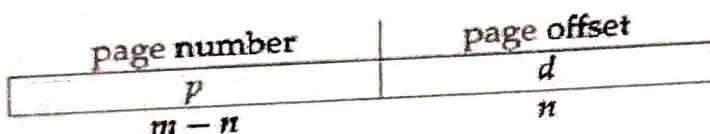


Figure 9.7 Paging model of logical and physical memory.



Where p is an index into the page table and d is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user view of memory can be mapped into physical memory.

Logical address 0 is page 0, offset 0. indexing into the page table; we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 ($= (5 \times 4) + 0$).

Logical address 3 (page 0, offset 3) maps to physical address 23 ($= (5 \times 4) + 3$). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.

Thus, logical address 4 maps to physical address 24 ($= (6 \times 4) + 0$). Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements

of a process do not happen to fall on page boundaries, the *last* frame allocated may not be completely full. For example, if pages are 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of $2048 - 1086 = 962$ bytes. In the worst case, a process would need n pages plus one byte. It would be allocated $n + 1$ frames, resulting in an internal fragmentation of almost an entire frame.

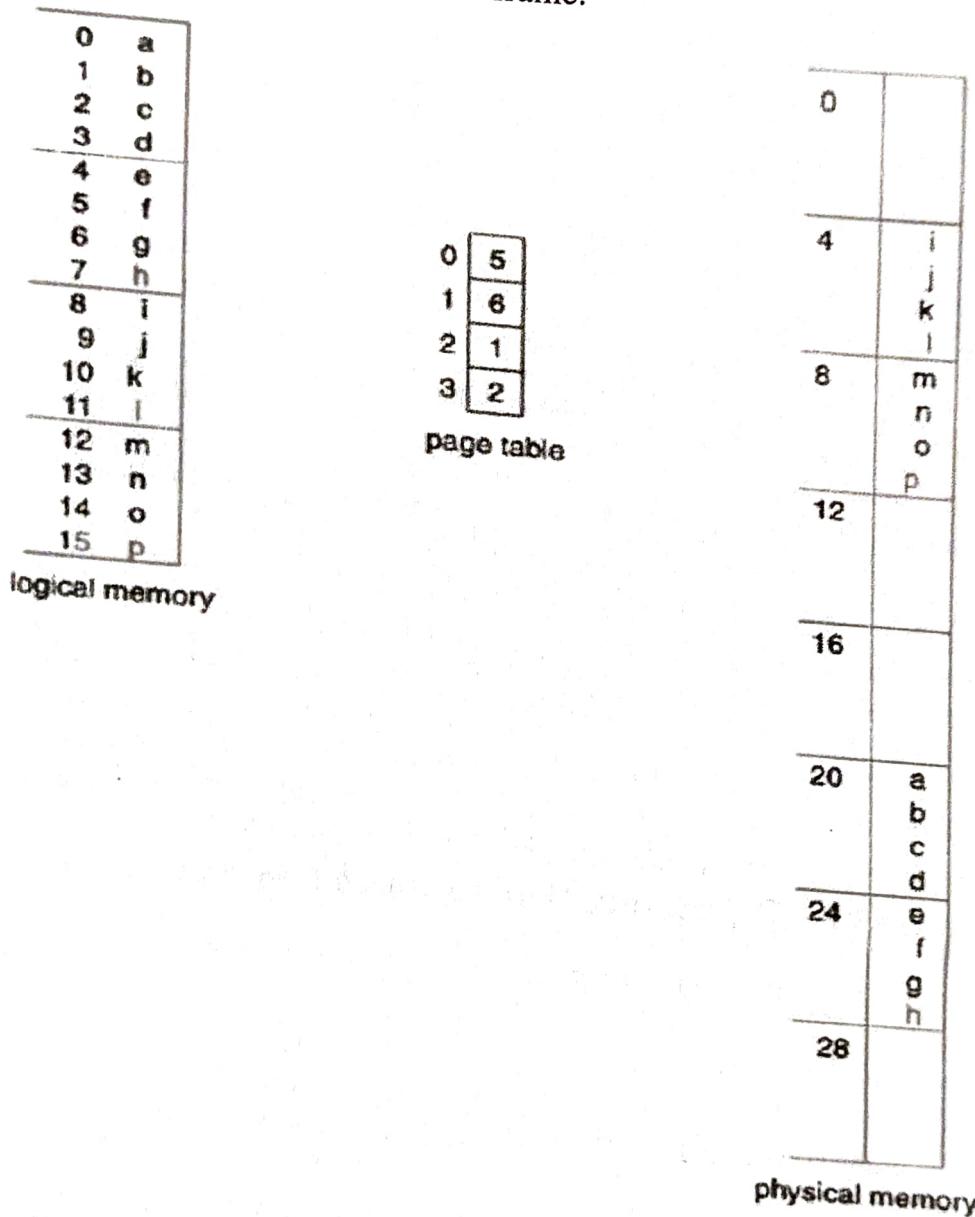


Figure 9.8 Paging example for a 32-byte memory with 4-byte pages.

Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. The mapping allows differentiation between **logical memory and physical memory**.

Basic Method

We think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. We talk about "the symbol table," "function *Sqrt*," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the symbol table is stored before or after the *Sqrt* function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment:

Segmentation is a memory-management scheme that supports this user view of memory. A logical-address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple*:

<Segment-number, offset>.

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A Pascal compiler might create separate segments for the following:

1. The global variables;
2. The procedure call stack, to store parameters and return addresses;
3. The code portion of each procedure or function;
4. The local variables of each procedure and function.

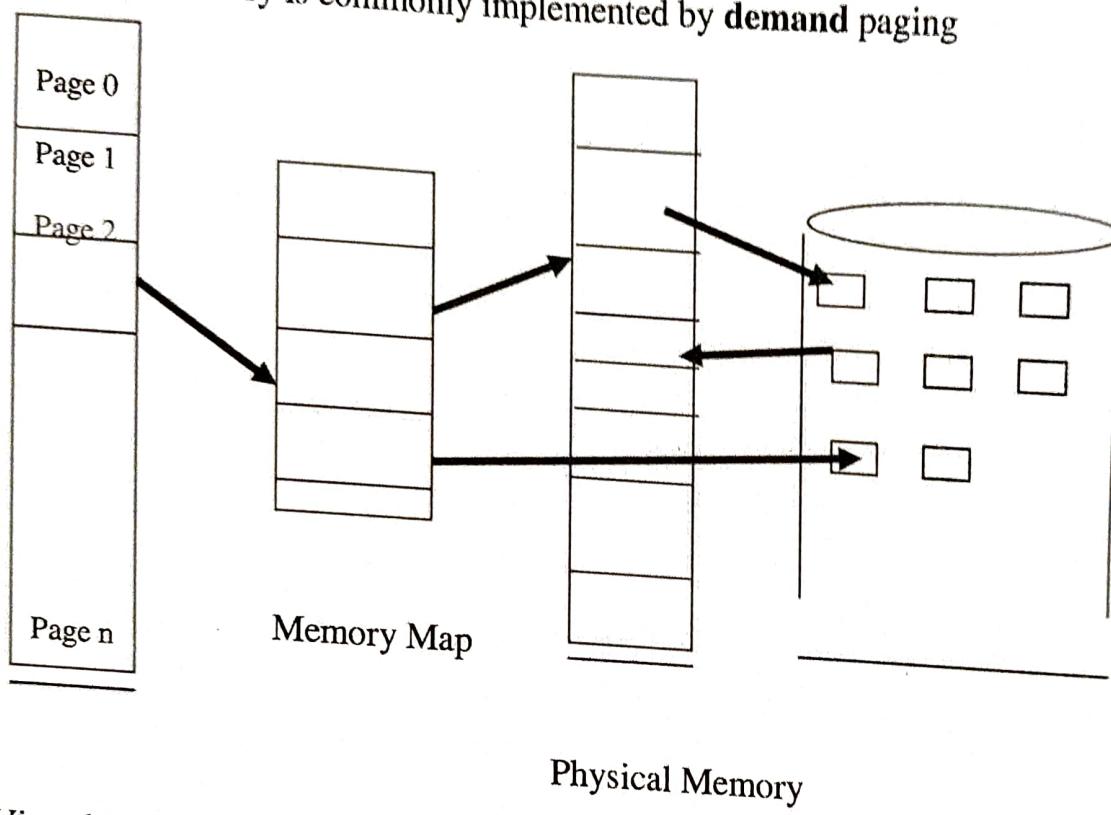
VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations.

Background

Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available, or about what code can be placed in overlays.

Virtual memory is commonly implemented by **demand** paging



Virtual Memory

Physical Memory

Figure Diagram showing virtual memory that is larger than physical memory.

Demand Paging

A demand-paging system is similar to a paging system with swapping. Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed.

With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid-invalid

bit scheme described in this Section can be used for this purpose. This time, however, when this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid or currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page-fault trap. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk-transfer overhead and memory requirements), rather than an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript). We must therefore correct this oversight. The procedure for handling this page fault is straightforward (Figure):

1. We check page table for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, and we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

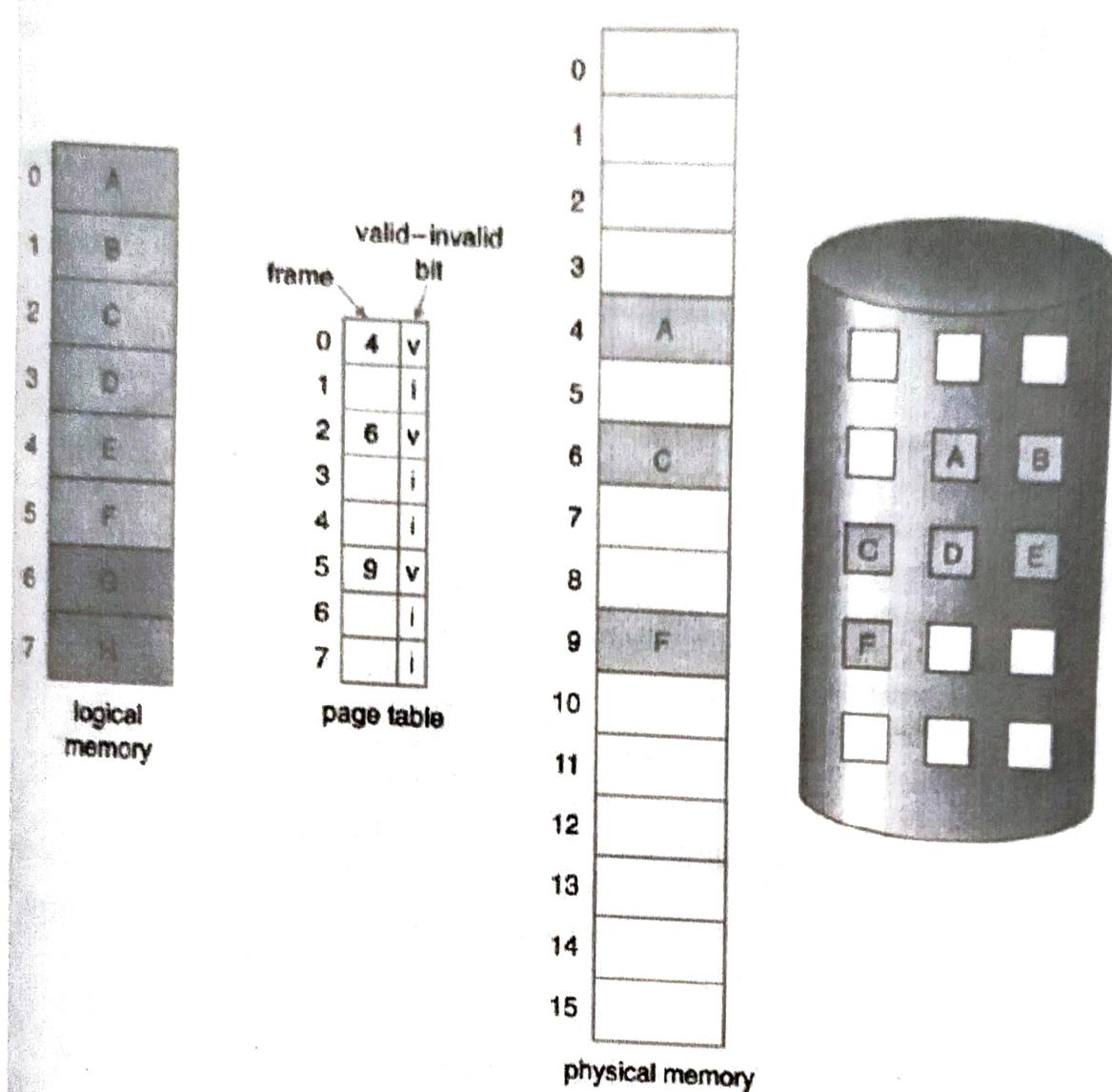


Figure 10.3 Page table when some pages are not in main memory.

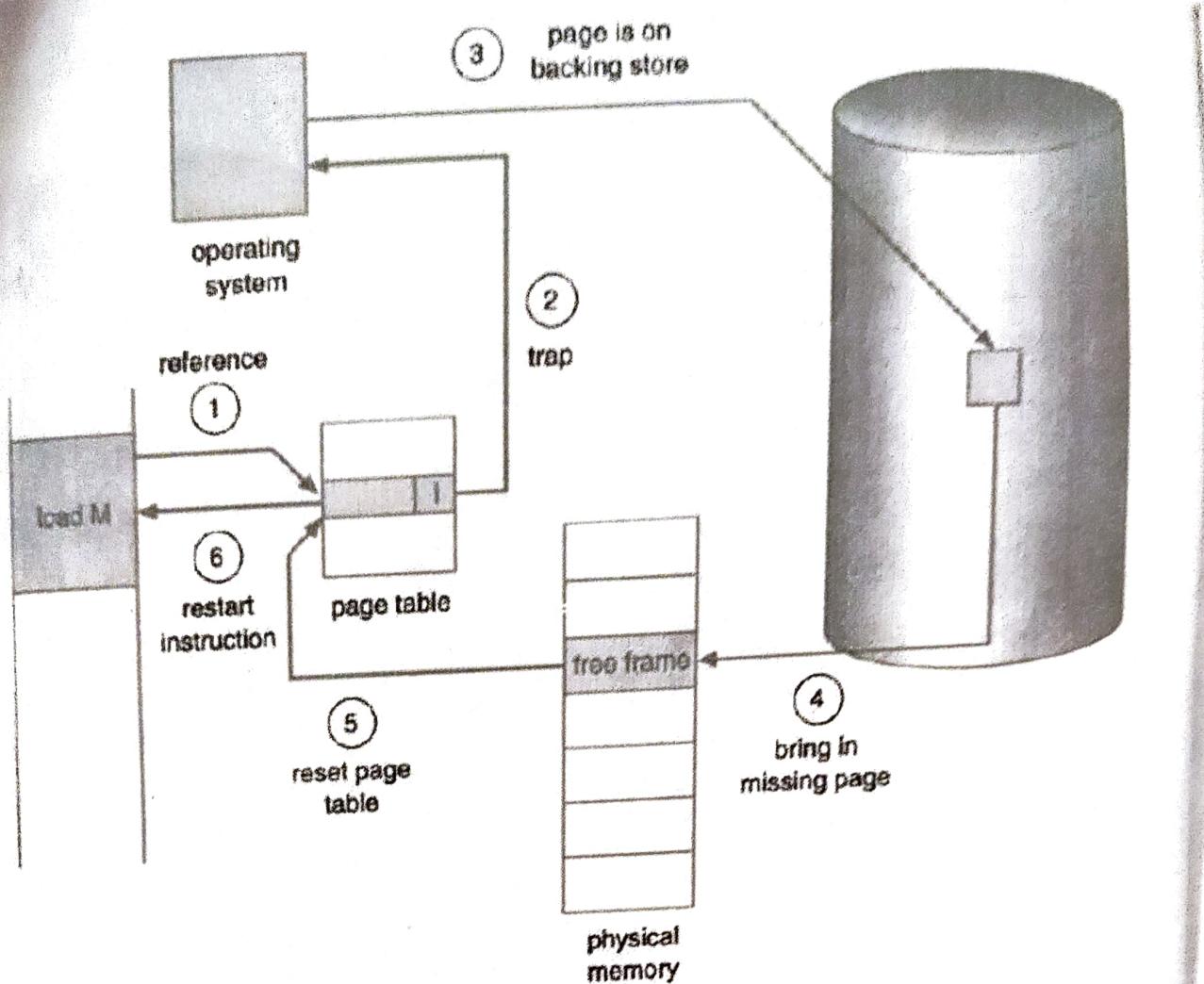


Figure 10.4 Steps in handling a page fault.

In the extreme case, we could start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is pure demand paging: Never bring a page into memory until it is required.

Page Replacement

While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access. The operating system determines

where the desired page is residing on the disk, but then finds that there are no free frames on the free-frame list: All memory is in use (Figure.)

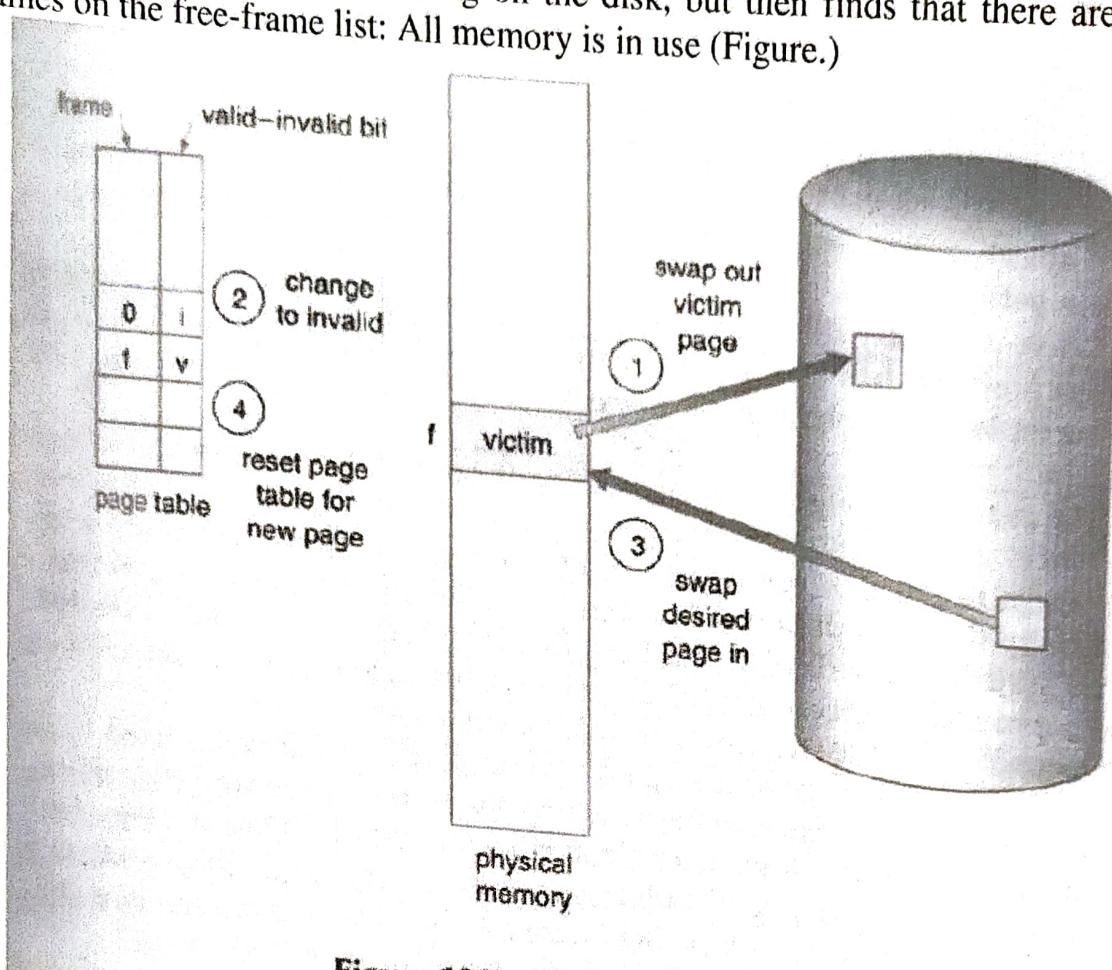


Figure 10.7 Page replacement.

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. So this option is not the best choice.

The operating system could swap out a process, freeing all its frames, and reducing the level of multiprogramming.

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk
2. Find a free frame:
 - a. If there is a free frame, use it.
 - b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
 - c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

Page Replacement Algorithm

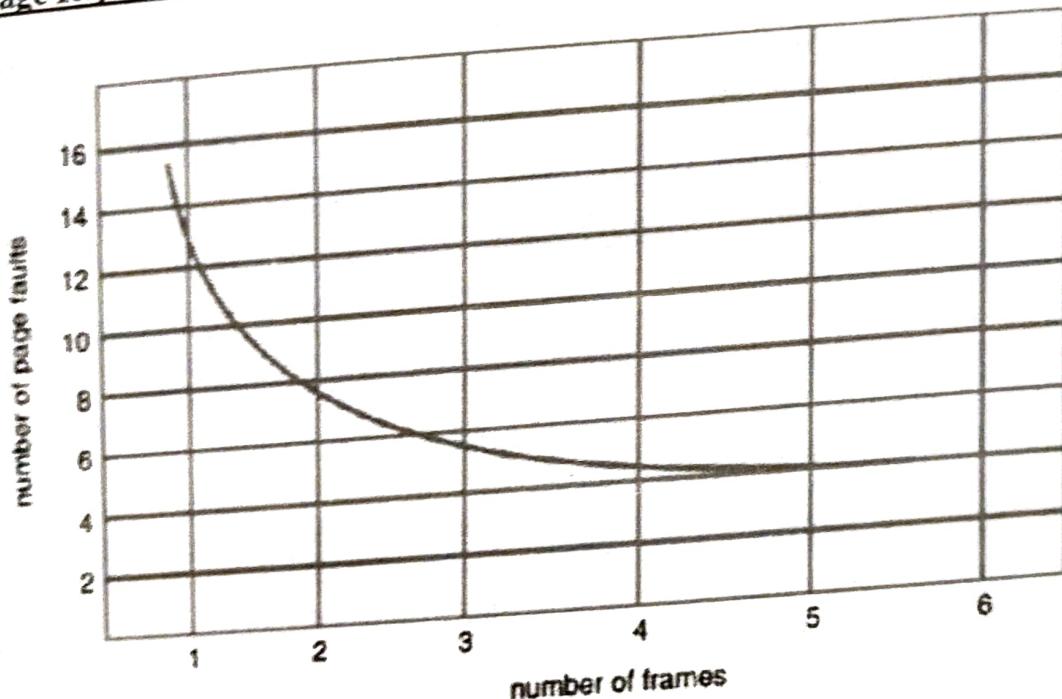


Figure 10.8 Graph of page faults versus the number of frames.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than entire address. Second, if we have a reference to a page p , then any *immediate* following references to page p will never cause a page fault. Page p will be memory after the first reference; the immediately following references will r fault. .

For example, if we trace a particular process, we might record the following address sequence:

0100,0432,0101,0612,0102,0103,0104,0101,0611,0102,0103,

Which, at 100 bytes per page, is reduced to the following reference string
1,4, 1, 6, 1, 6, 1, 6, 1.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

7,0,1,2,0,3,0,4,2,3,0,3, 2,1, 2, 0, 1, 7, 0,1

For a memory with three frames.

1. FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A **FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.** We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0,1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure. Every time a fault occurs, we show which pages are in our three

frames. There are 15 faults altogether.

- The FIFO page-replacement algorithm is easy to understand and program.
- However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

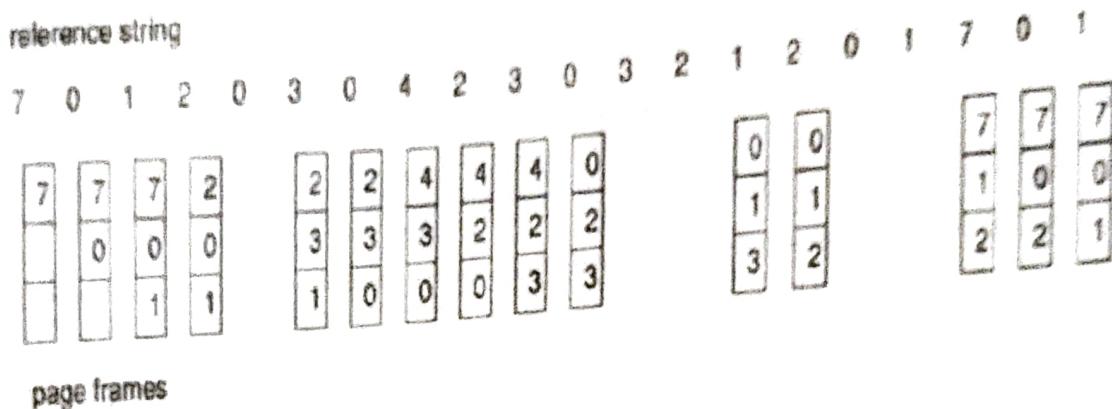


Figure 10.9 FIFO page-replacement algorithm.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Figure shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: For some page-replacement algorithms, the page fault rate may *increase* as the number of allocated frames increases.

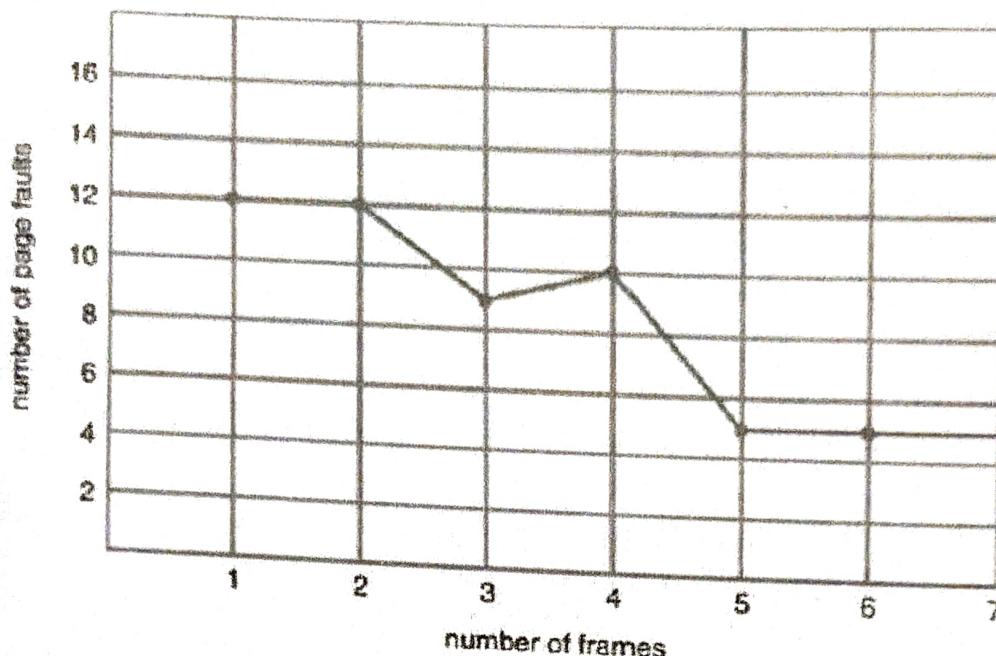


Figure 10.10 Page-fault curve for FIFO replacement on a reference string.

2. Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. It is simply

Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure . The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

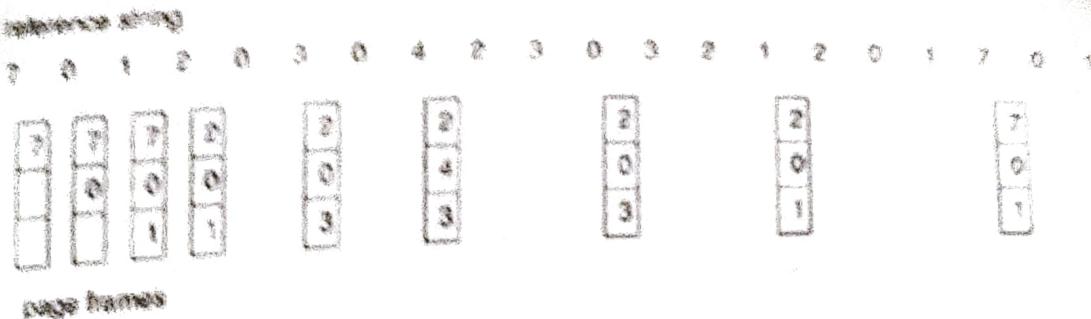


Figure 10.11 Optimal page-replacement algorithm.

3. LRU Page Replacement

If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used* for the longest period of time (Figure). This approach is the least-recently-used (LRU) algorithm.

LRU replacement associates with each page the time of that pages last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward.

The result of applying LRU replacement to our example reference string is shown in Figure. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0 and just before that page 3 was used. Thus the LRU algorithm replaces page 2 not knowing that page 2 is about to be used. When it then faults for page 2 the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

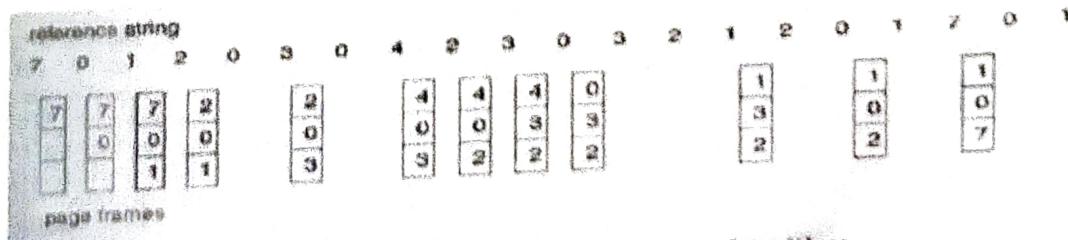


Figure 10.12 LRU page-replacement algorithm.

Two implementations are:

Counters: In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a **logical clock or counter**. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page, and a write-to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

- Stack:** Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page.

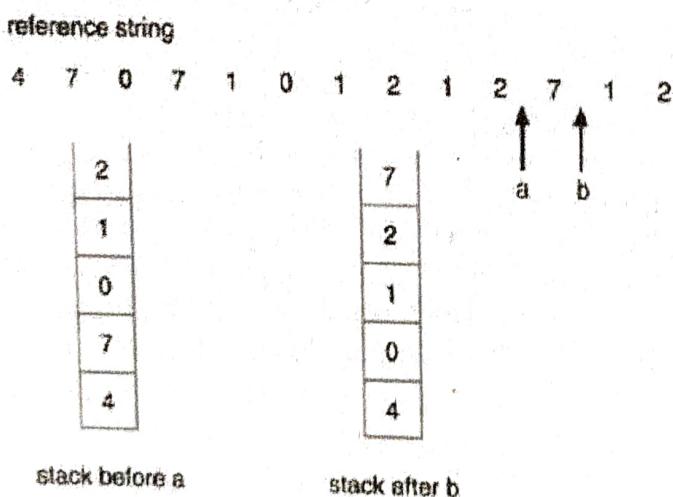


Figure 10.13 Use of a stack to record the most recent page references.