Virtual Memory

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need.
- The ability to execute a program that is only partially in memory would confer many benefits:

1. A program would no longer be constrained by the amount of physical memory that is available.
2. Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
3. Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

- Virtual memory involves the separation of logical memory as perceived by users from physical memory.
- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
- Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory.
- We allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls.
- The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.
- Virtual address spaces that include holes are known as sparse address spaces.
- Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries during program execution.

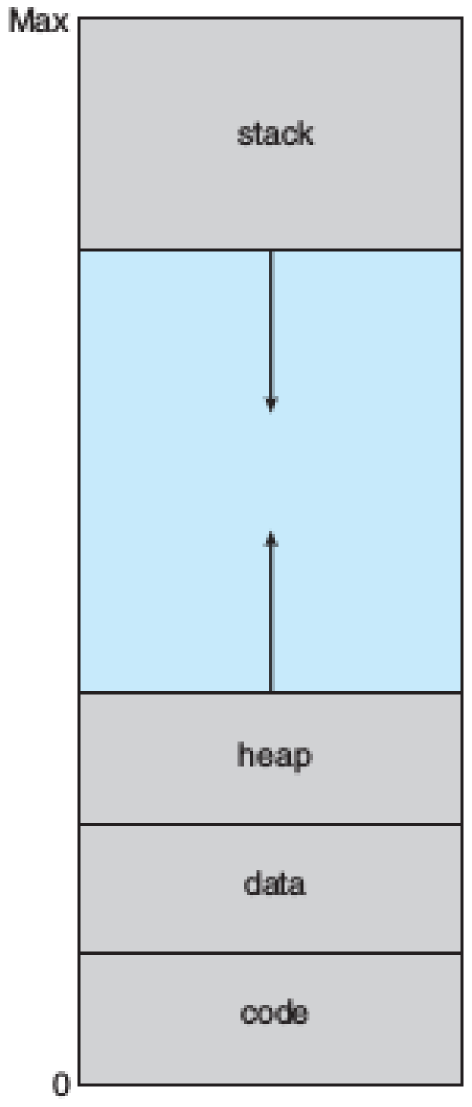

Figure 9.2 Virtual address space.

- Virtual memory allows files and memory to be shared by two or more processes through page sharing.

  1. **System libraries** can be shared by several processes through mapping of the shared object into a virtual address space.

- Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes
- Typically, a library is mapped read-only into the space of each process that is linked with it.

  2. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared

  3. Pages can be shared during process creation with the fork() system call, thus speeding up process creation.

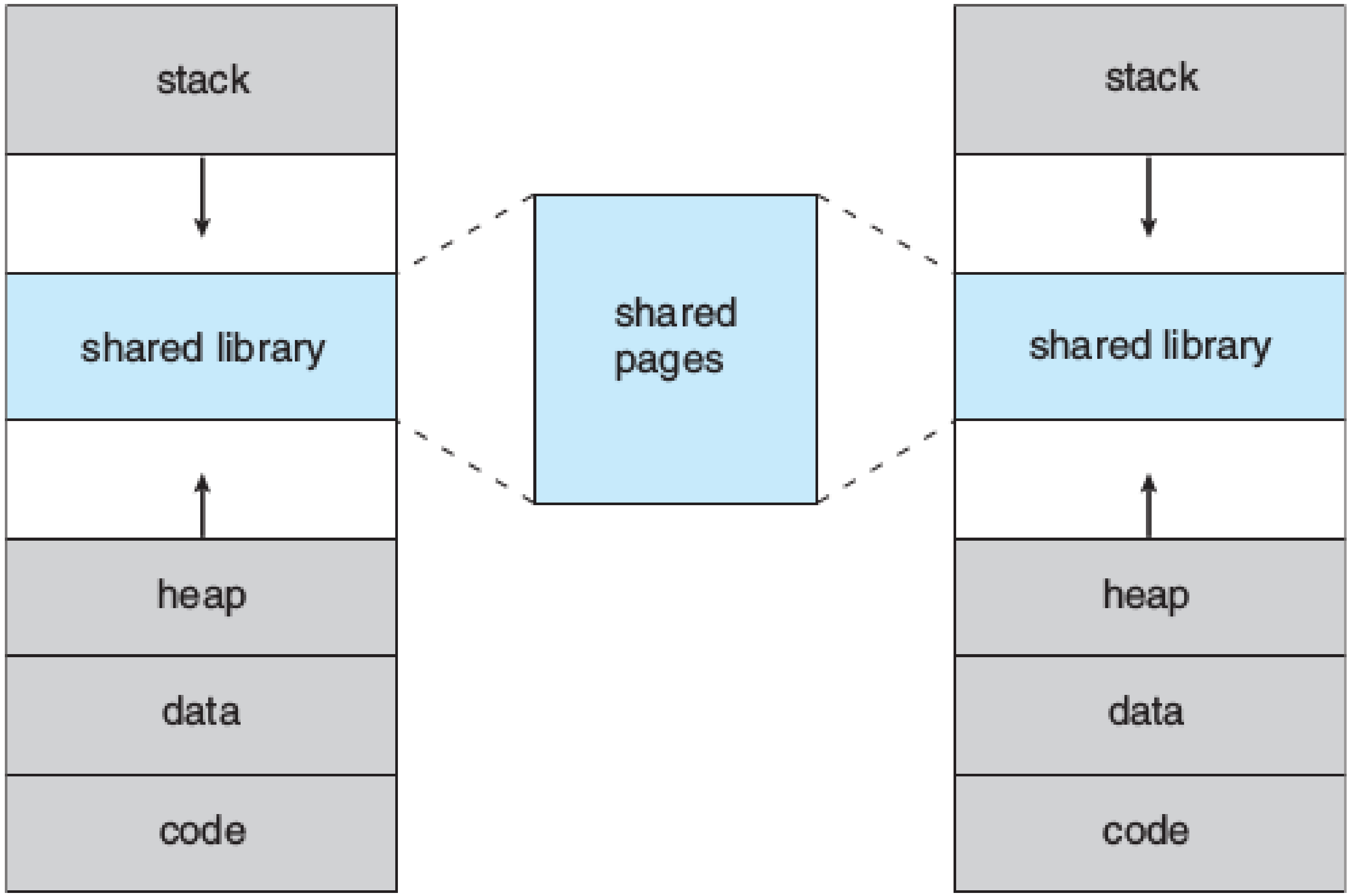

Figure 9.3 Shared library using virtual memory.

Demand Paging

- It involves loading pages only as they are needed and is commonly used in virtual memory systems.
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
- Pages that are never accessed are thus never loaded into physical memory.
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).
- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper** or **pager**.
- A lazy swapper never swaps a page into memory unless that page will be needed.

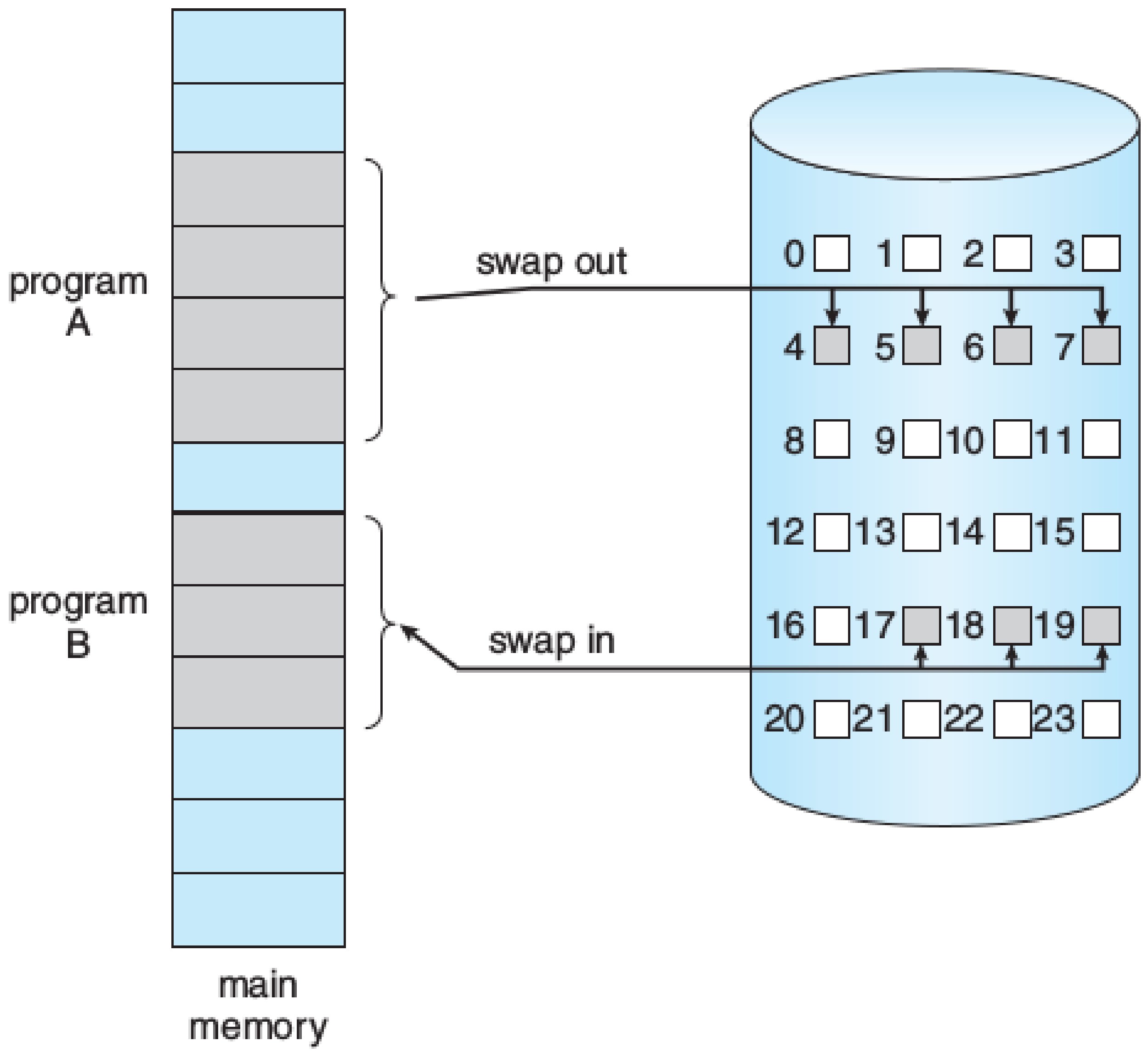

Figure 9.4 Transfer of a paged memory to contiguous disk space.

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- To distinguish between the pages that are in memory and the pages that are on the disk, the valid –invalid bit scheme is used.
- When this bit is set to “valid,” the associated page is both legal and in memory.
- If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

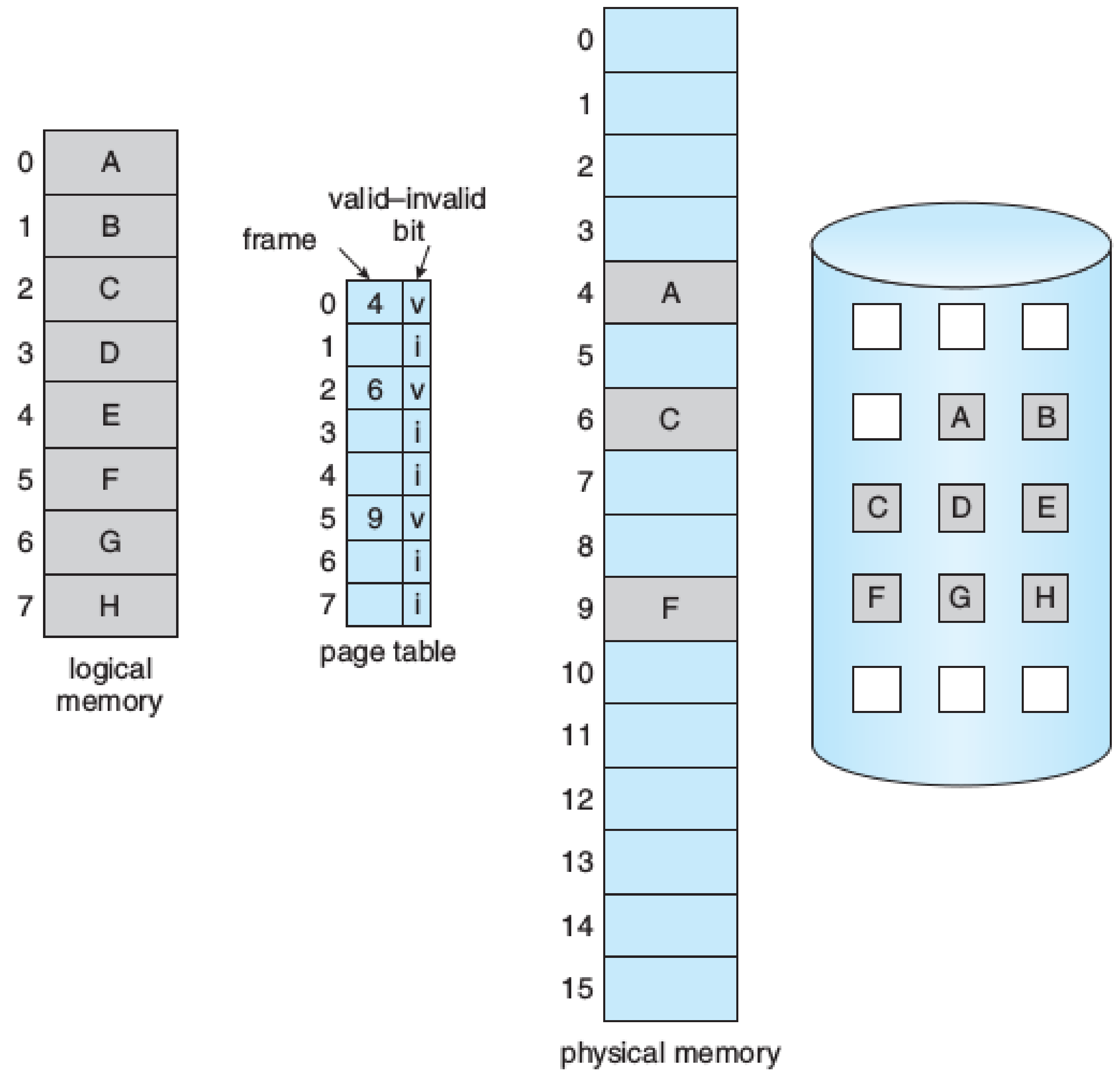

Figure 9.5 Page table when some pages are not in main memory.

- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- Access to a page marked invalid causes a **page fault**. The procedure for handling this page fault is:

  1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

  2. If the reference was invalid, we terminate the process. If it was valid but 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 trap. The process can now access the page as though it had always been in memory.

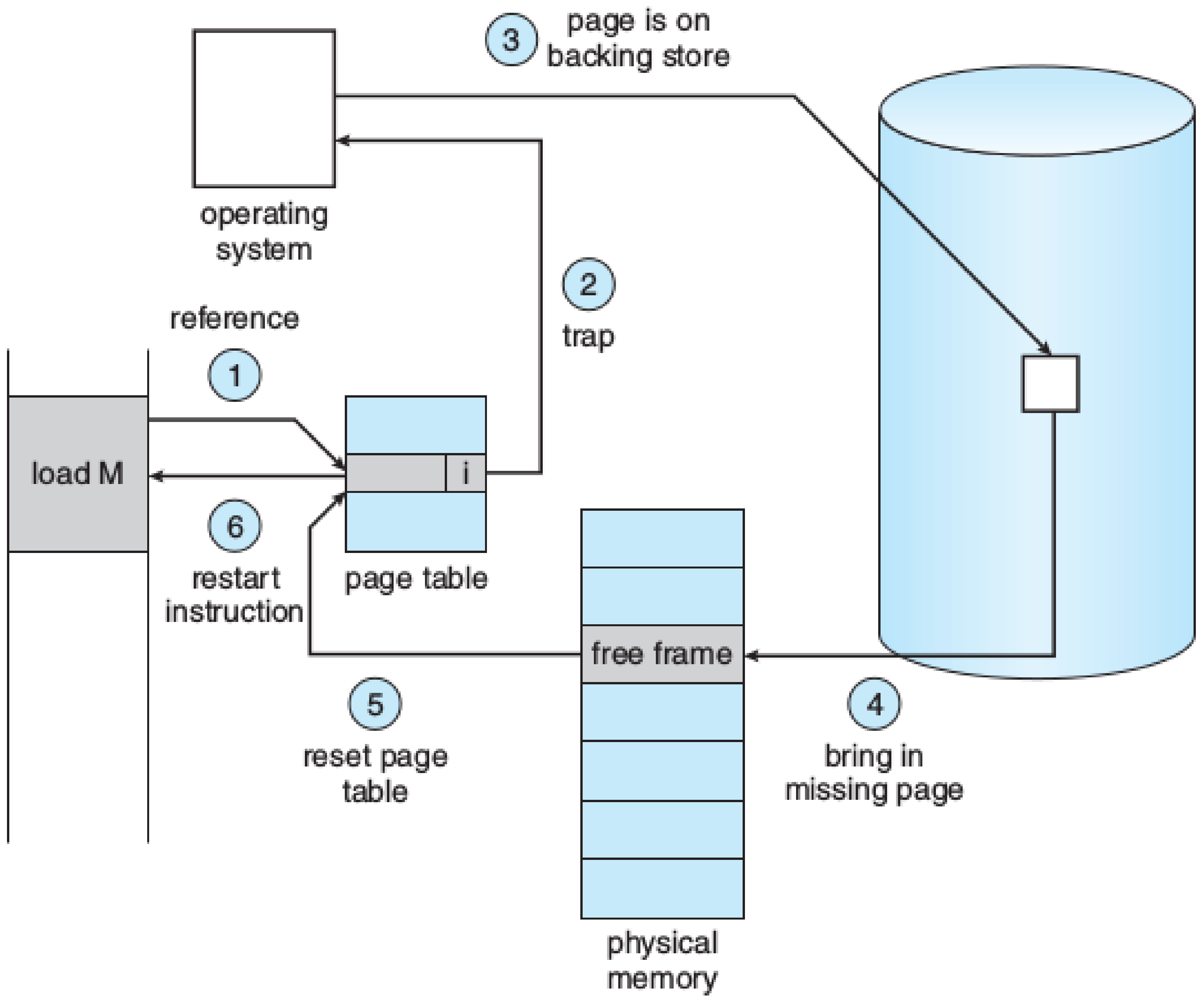

Figure 9.6 Steps in handling a page fault.

- In the extreme case, we can 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.

- Programs tend to have **locality of reference**, which results in reasonable performance from demand paging.
- If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again.
- If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.
- Consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

  1. Fetch and decode the instruction (ADD).
  2. Fetch A.
  3. Fetch B.
  4. Add A and B.
  5. Store the sum in C.
- If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction.
- The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again.

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system.
- For most computer systems, the memory-access time, denoted ma, ranges from 10 to 200 nanoseconds. As long as we have no page faults, theeffective access time for a demand-paged Memory is equal to the memory access time.
- If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults.
- The effective access time is then

  effective access time = $(1 - p) \times ma + p \times$ page fault time.
- To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

  1. Trap to the operating system.

  2. Save the user registers and process state.

3. Determine that the interrupt was a page fault.

4. Check that the page reference was legal and determine the location of the page on the disk.

5. Issue a read from the disk to a free frame:

a. Wait in a queue for this device until the read request is serviced.

b. Wait for the device seek and/or latency time.

c. Begin the transfer of the page to a free frame.

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).

7. Receive an interrupt from the disk I/O subsystem (I/O completed).

8. Save the registers and process state for the other user (if step 6 is executed).

9. Determine that the interrupt was from the disk.

10. Correct the page table and other tables to show that the desired page is now in memory.

11. Wait for the CPU to be allocated to this process again.

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

- With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

  effective access time = $(1 - p) \times (200) + p$ (8 milliseconds)

  $= (1 - p) \times 200 + p \times 8{,}000{,}000$

  $= 200 + 7{,}999{,}800 \times p$.

- If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

  $220 > 200 + 7{,}999{,}800 \times p$,

  $20 > 7{,}999{,}800 \times p$,

  $p < 0.0000025$.

Copy-on-Write

- The fork() system call creates a child process that is a duplicate of its parent. Traditionally, fork() worked by creating a copy of the parent’s address space for the child, duplicating the pages belonging to the parent.
- However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent’s address space may be unnecessary.
- Instead, we can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages.
- These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

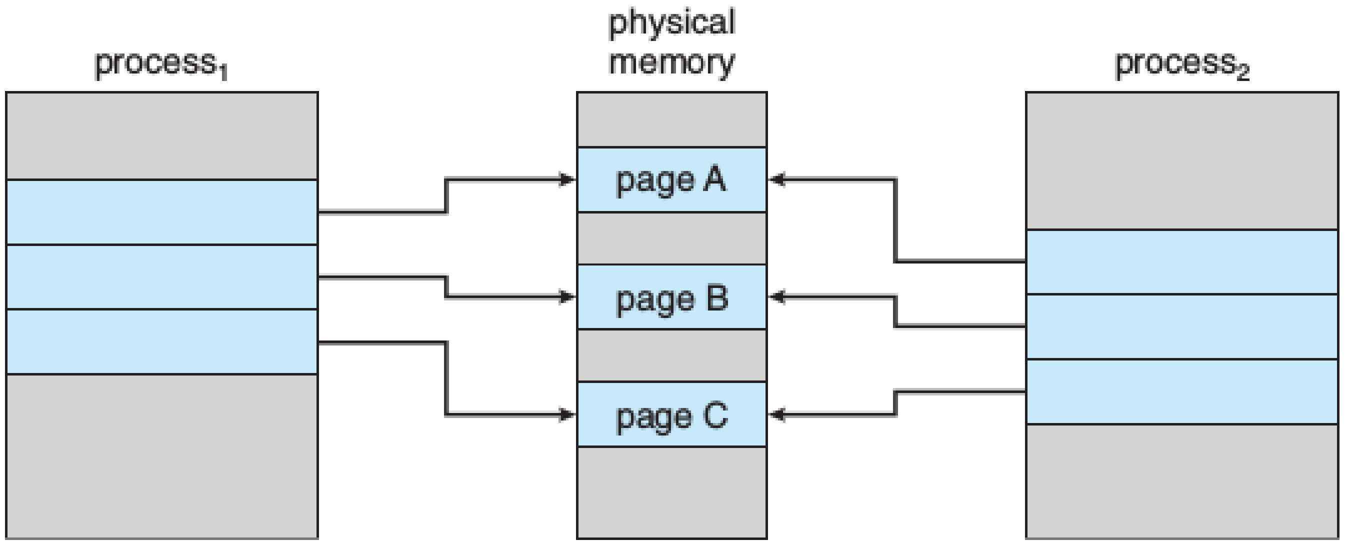

Figure 9.7 Before process 1 modifies page C.

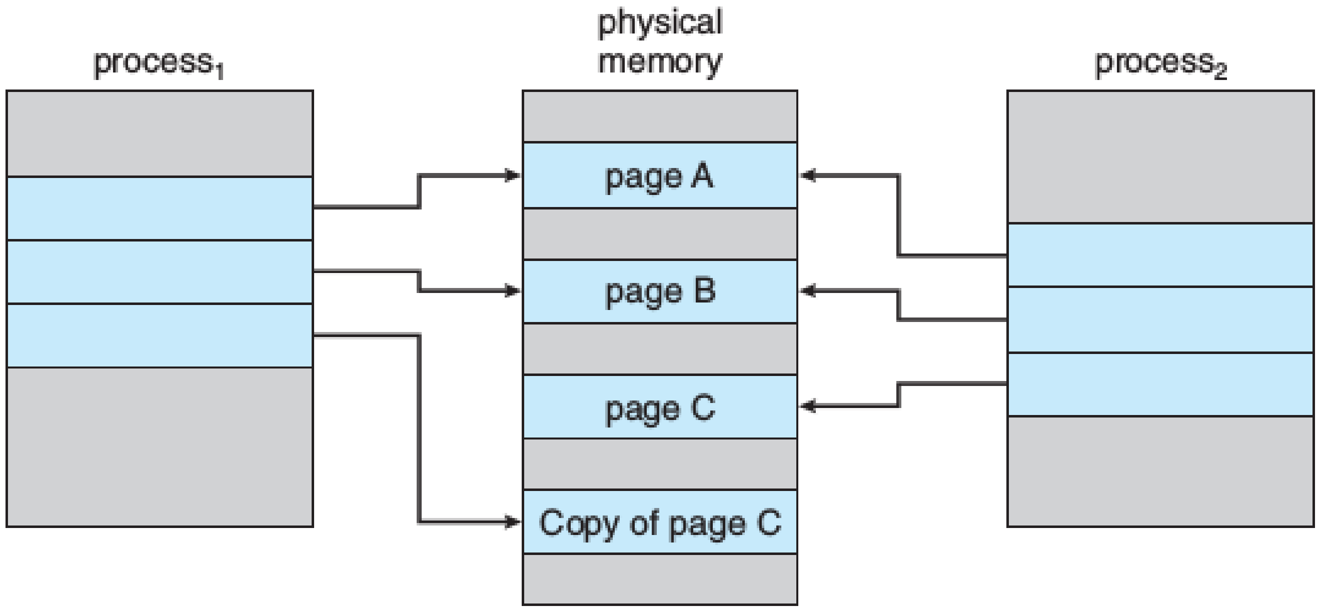

Figure 9.8 After process 1 modifies page C.

- For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
- The operating system will create a copy of this page, mapping it to the address space of the child process.
- The child process will then modify its copied page and not the page belonging to the parent process.
- Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes.
- When it is determined that a page is going to be duplicated using copy-on-write, it is important to note the location from which the free page will be allocated.
- Operating systems typically allocate these pages using a technique known as **zero-fill-on-demand**. Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

Page Replacement

- If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used.
- Thus, with demand paging, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).
- If we increase our degree of multiprogramming, we are **over-allocating memory**.
- If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare.
- It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

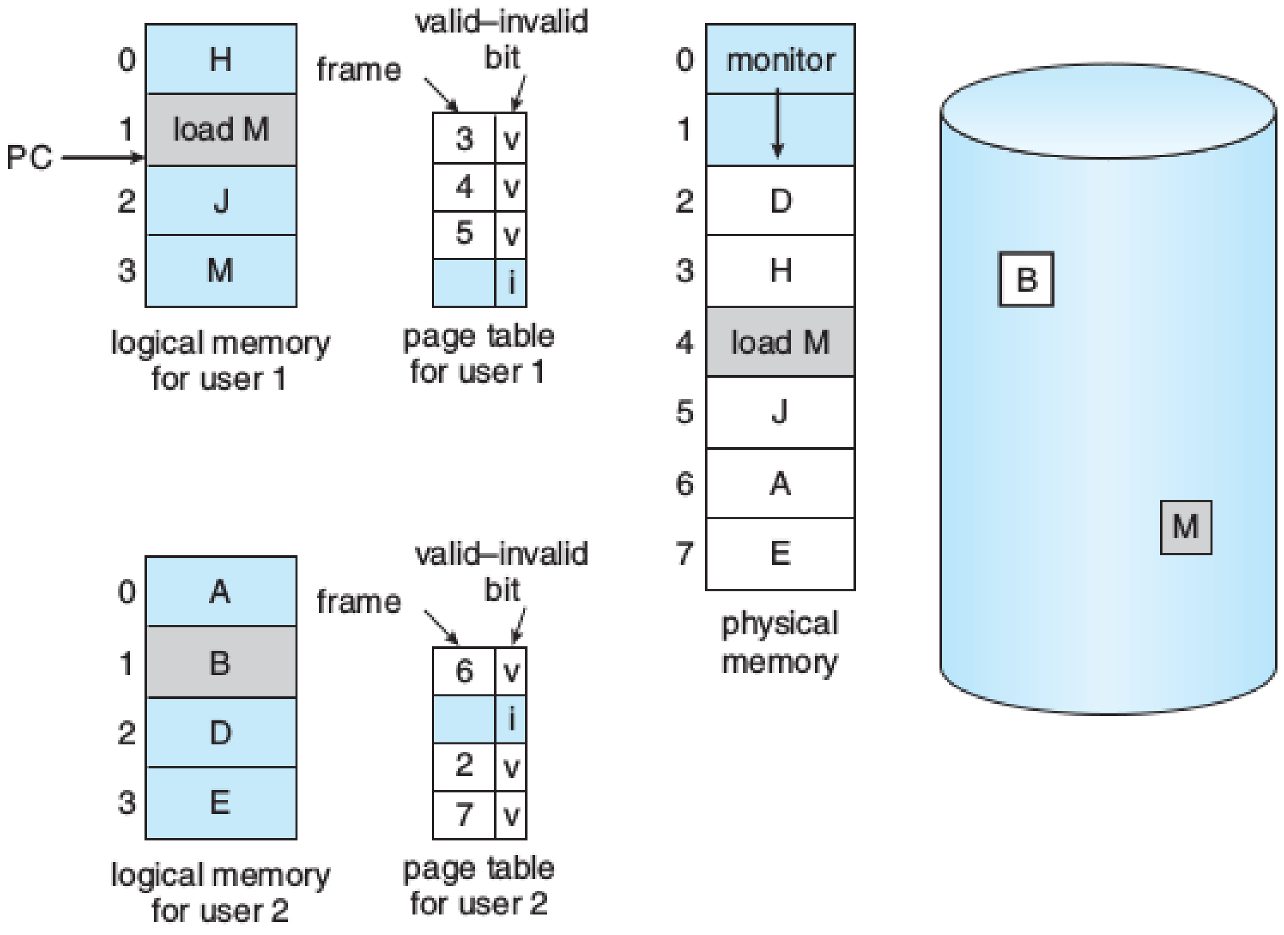

Figure 9.9 Need for page replacement.

- While a user process is executing, a page fault occurs. 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.

- The operating system 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.
- The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.
- A third solution is **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 frame to the disk; change the page and frame tables accordingly.

  3. Read the desired page into the newly freed frame; change the page and frame tables.

  4. Continue the user process from where the page fault occurred.

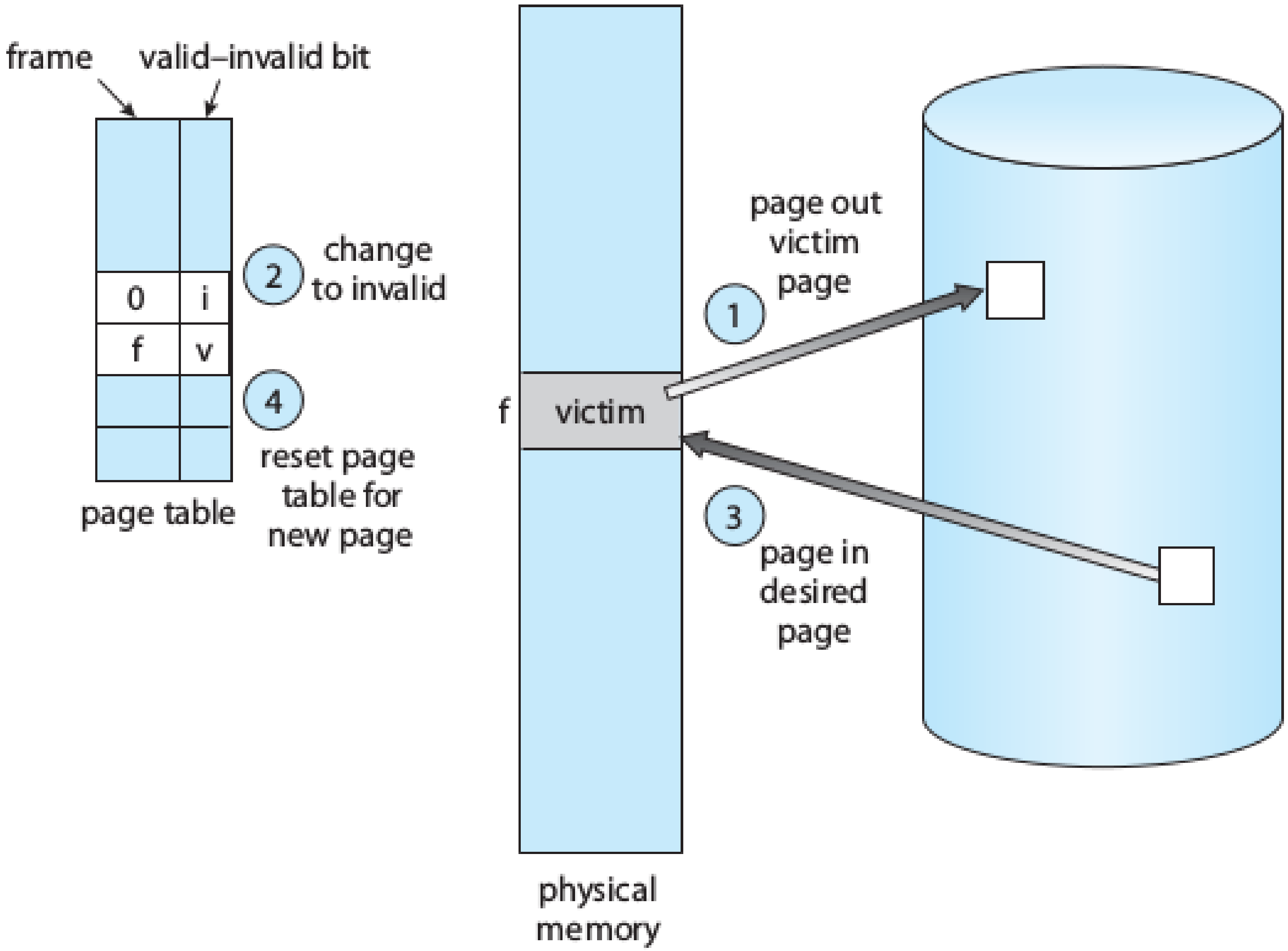

Figure 9.10 Page replacement.

- If no frames are free, two page transfers (one out and one in) are required.
- This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
- A **modify bit (or dirty bit)** can be used. Each page or frame has a modify bit associated with it in the hardware.
- The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.

- Two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**.
- That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.
- 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**.
- 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.

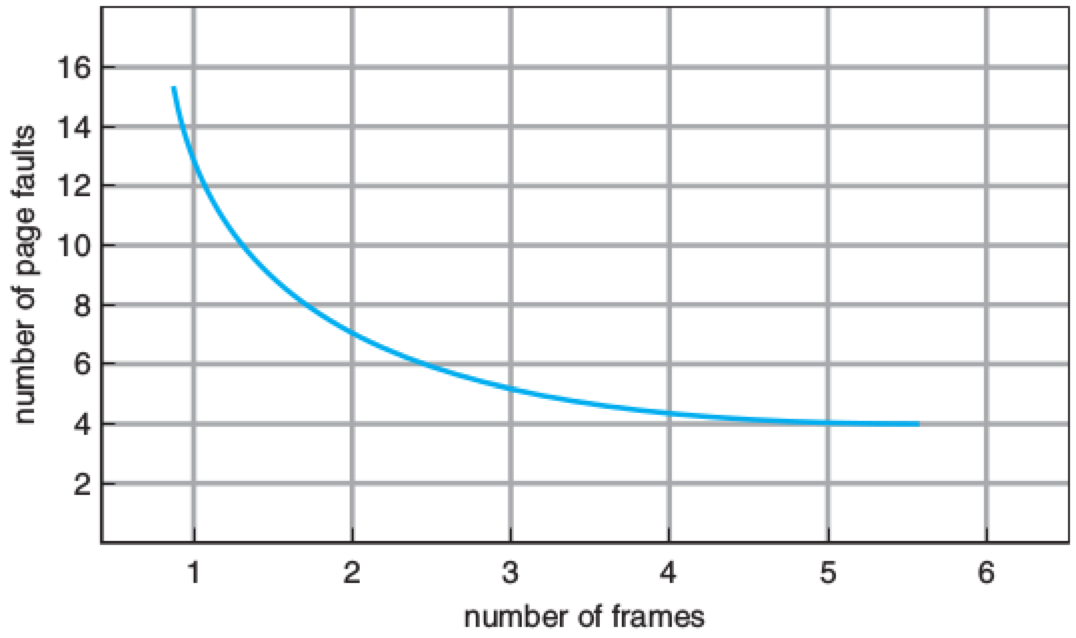

Figure 9.11 Graph of page faults versus number of frames.

1. **FIFO Page Replacement**

- It associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.
- Consider the reference string

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

and 3 available frames.

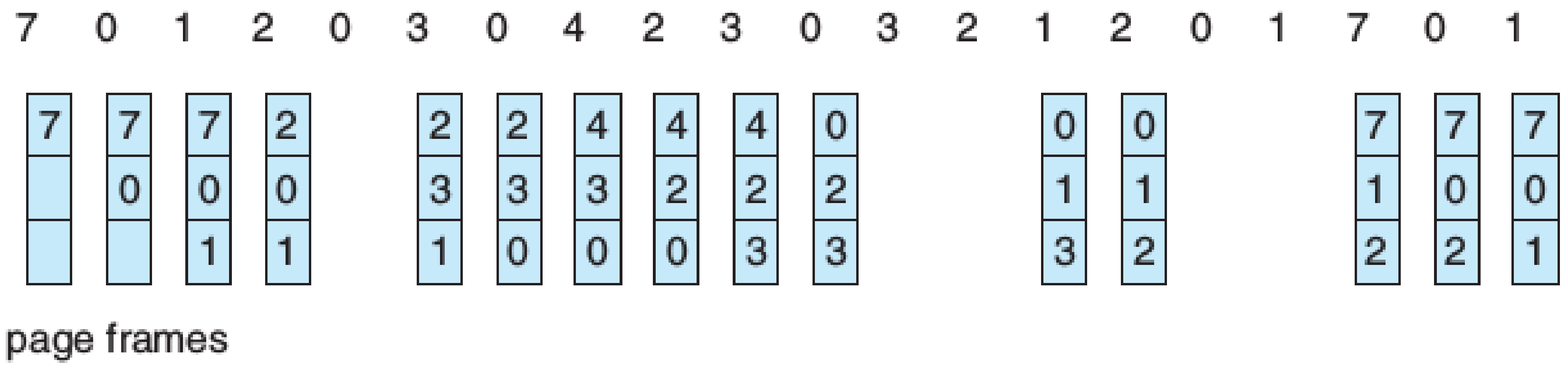

Figure 9.12 FIFO page-replacement algorithm.

- Solve for 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 with 3 frames then 4 frames.
- The number of faults for four frames (ten) 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.

2. Optimal Page Replacement

- This algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Replace the page that will not be used for the longest period of time.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

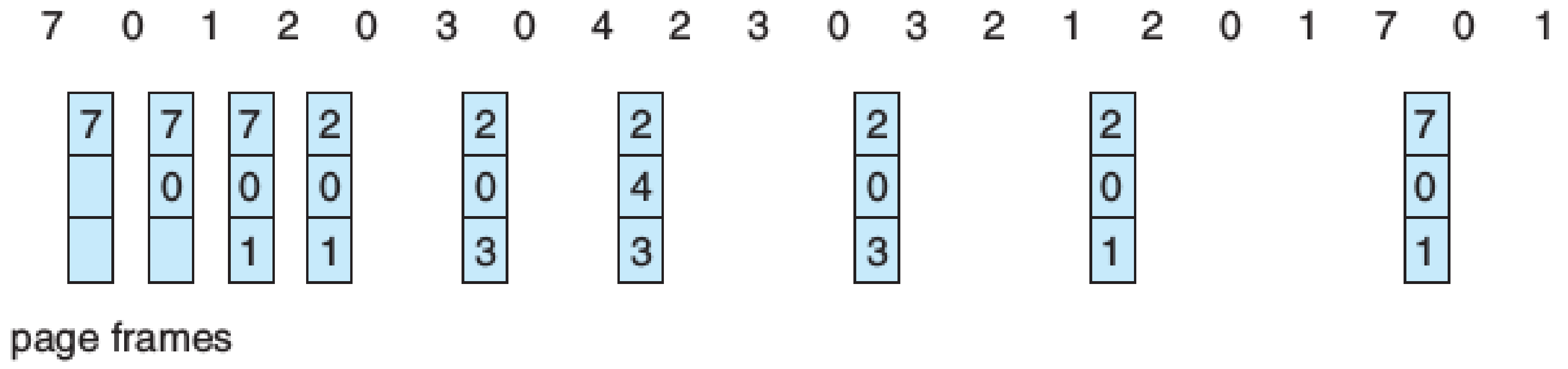

Figure 9.14 Optimal page-replacement algorithm.