Chapter 8. Virtual Memory

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018). *Operating System Concepts* (10th ed.). Wiley.

Contents

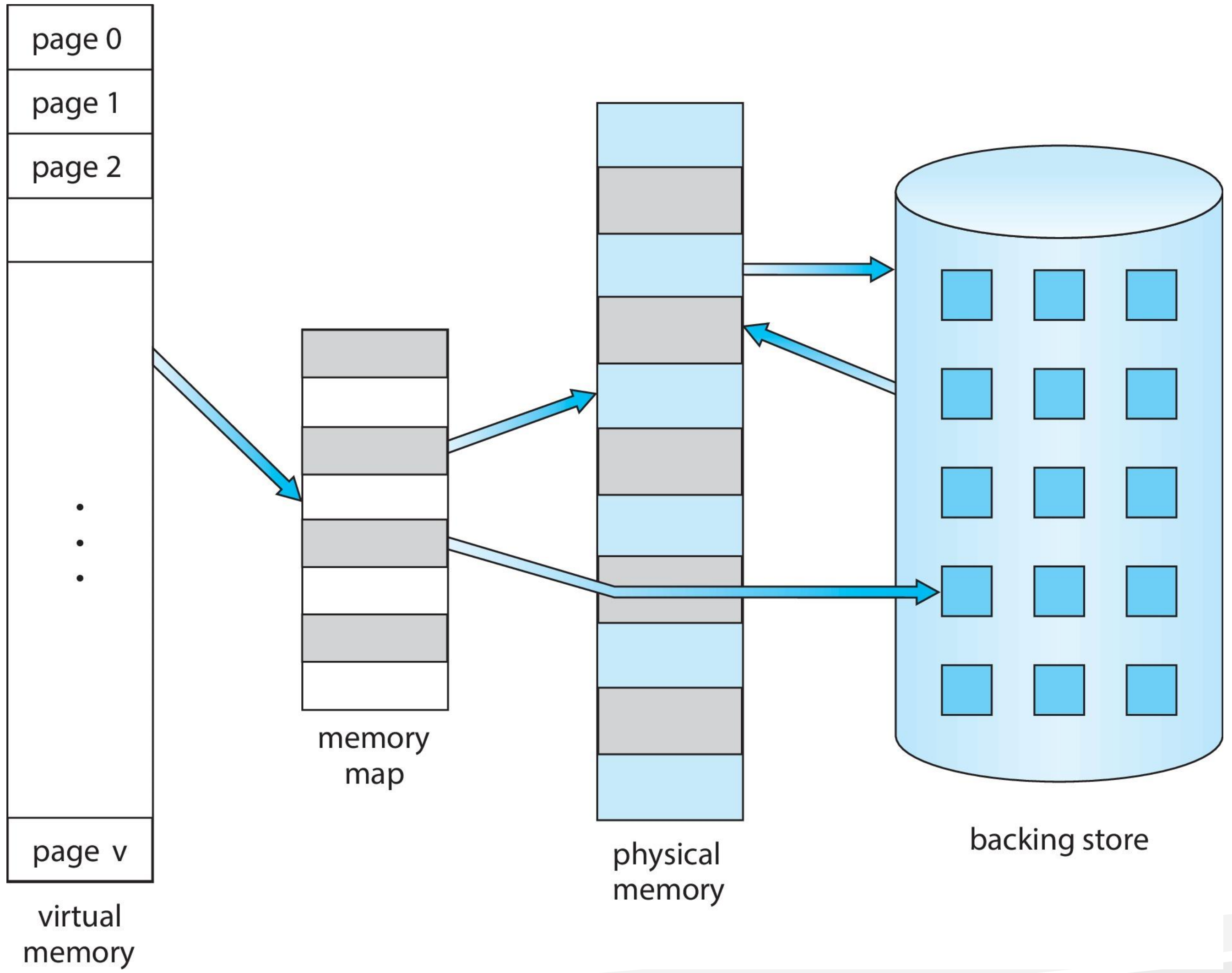
1. Background

- Instructions being executed must be in physical memory.
  - The first approach to meeting this requirement is to place the entire logical address space in physical memory.
    - In fact, an examination of real programs shows us that, in many cases, the entire program is not needed.
    - Even in those cases where the entire program is needed, it may not all be needed at the same time.
  - The ability to execute a program that is only partially in memory would confer many benefits:
    - A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large **virtual** address space, simplifying the programming task.
    - Because each 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.
    - Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

→ Thus, running a program that is not entirely in memory would benefit both the system and its users.

1. Background
Virtual Memory

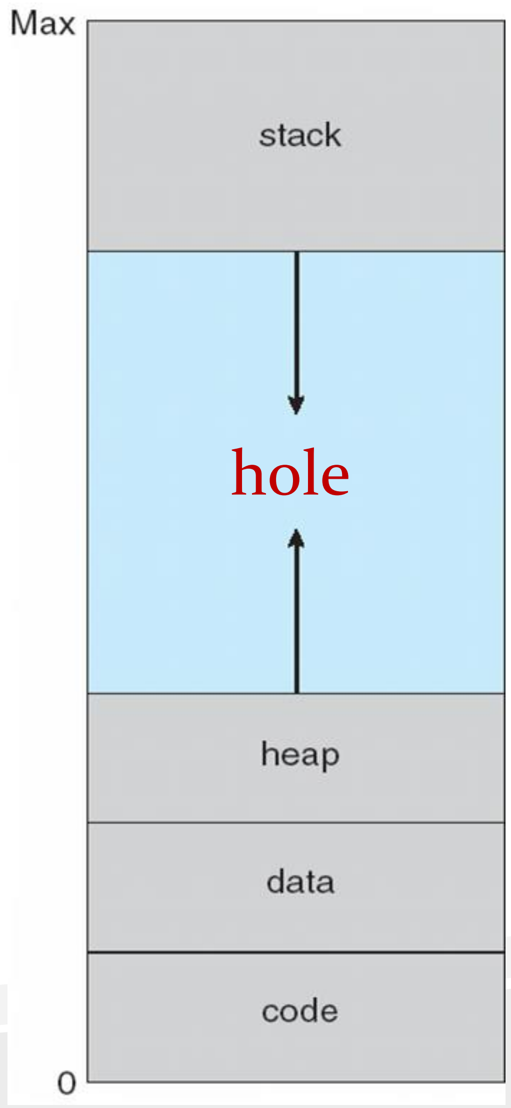
- **Virtual memory** involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
  - Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available.



1. Background

Virtual Address Space

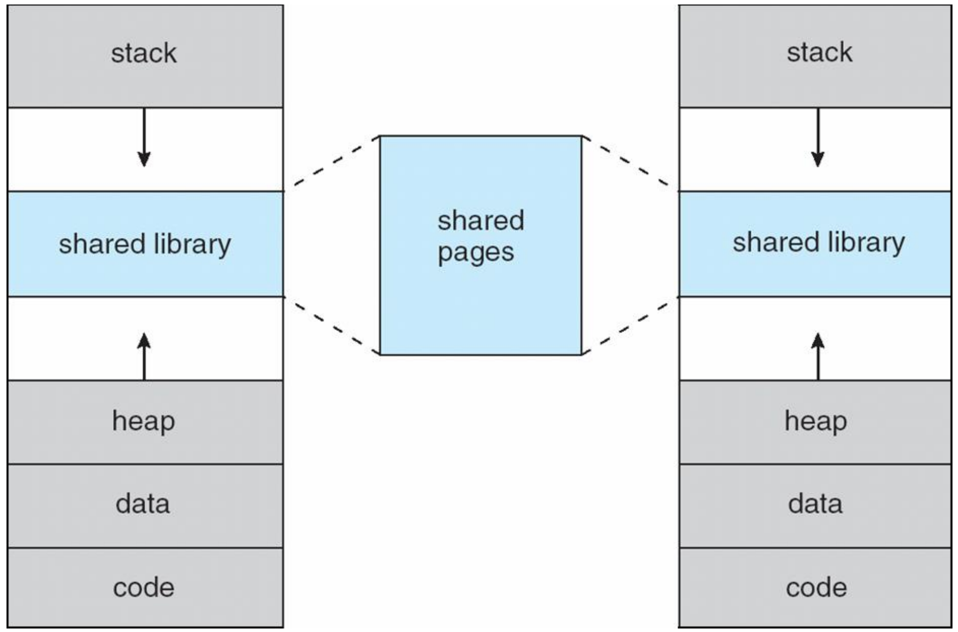
- The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address -say, address 0 - and exists in contiguous memory.
  - Recall that physical memory is organized in frames and that the physical frames assigned to a process may not be contiguous
- The heap grows upward in memory as it is used for dynamic memory allocation. The stack grows 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.



1. Background

Virtual Address Space

- In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:
  - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space.
  - Similarly, processes can share memory. Recall that two or more processes can communicate through the use of shared memory.
  - Pages can be shared during process creation with the `fork()` system call, thus speeding up process creation.



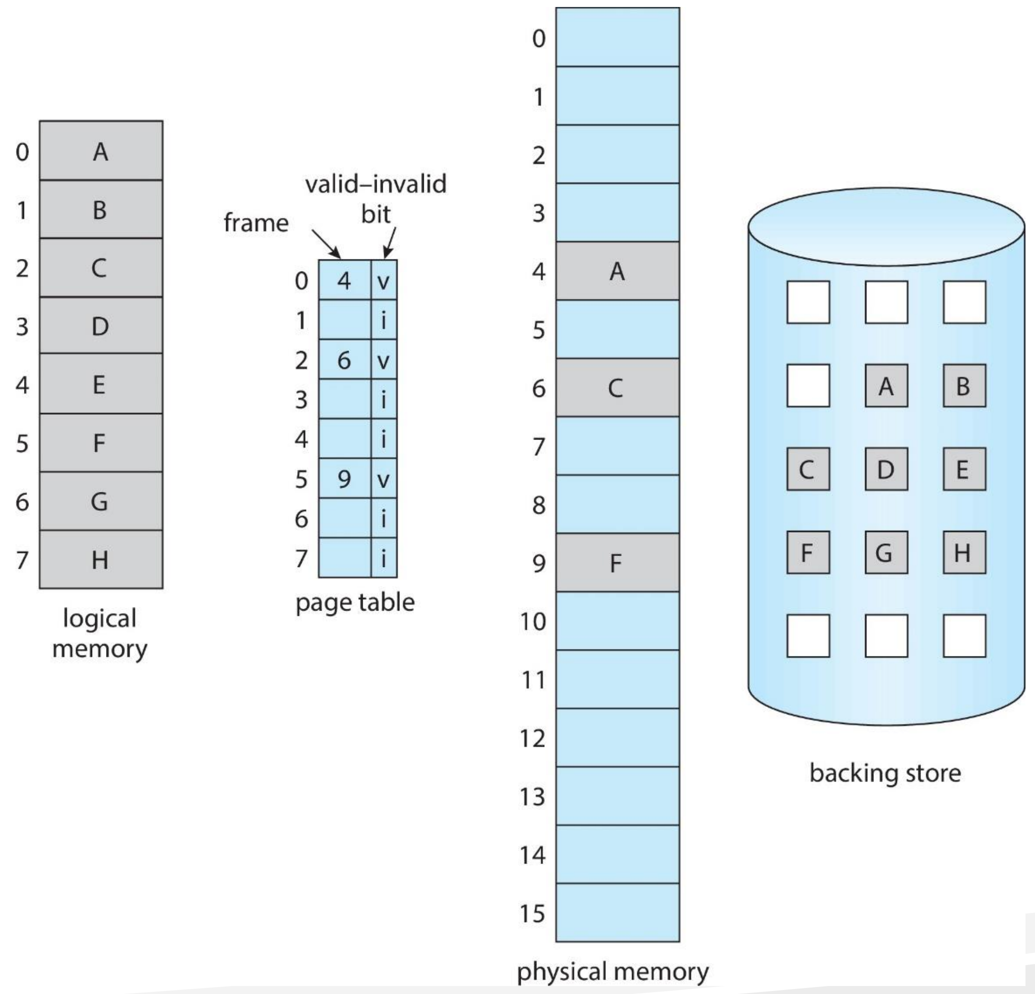
2. Demand Paging

- Consider how an executable program might be loaded from secondary storage into memory.
  - One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory.
  - An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.
    - Pages are loaded only when they are ***demanded*** during program execution. Pages that are never accessed are thus never loaded into physical memory.
    - Demand paging explains one of the primary benefits of virtual memory—by loading only the portions of programs that are needed, memory is used more efficiently.

2. Demand Paging

Basic Concepts

- The general concept behind demand paging is to load a page in memory only when it is needed. As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage.
  - To distinguish between the two an invalid bit scheme can be used.
    - Valid: In memory
    - Invalid: Not in memory
  - Access to a page marked invalid causes a **page fault**. 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.



2. Demand Paging

Basic Concepts

- Steps in handling a page fault

  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.

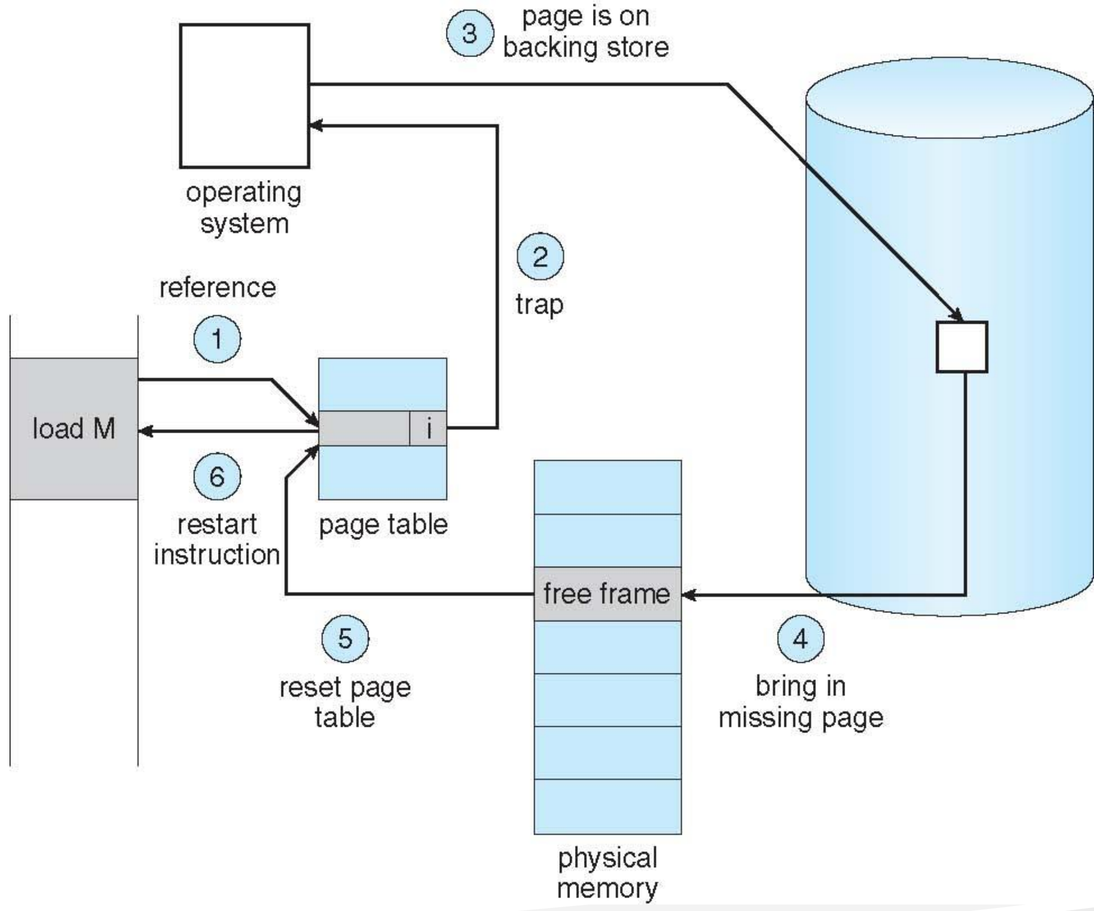
  2. If the reference was invalid, we terminate the process.

  3. We find a free frame (by taking one from the free-frame list, for example)

  4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.

  5. When the storage 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



2. Demand Paging

Basic Concepts

- Aspects of demand paging
  - 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.
  - Actually, a given instruction could access multiple pages → multiple page faults.
    - Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, which results in reasonable performance from demand paging.
  - The hardware to support demand paging is the same as the hardware for paging and swapping:
    - Page table. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
    - Secondary memory. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk or NVM device. It is known as the swap device, and the section of storage used for this purpose is known as swap space.

3. Copy-on-Write

- **Copy-on-write** (COW) allows both parent and child processes to initially share the same pages in memory.
  - 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.
- COW allows more efficient process creation as only modified pages are copied

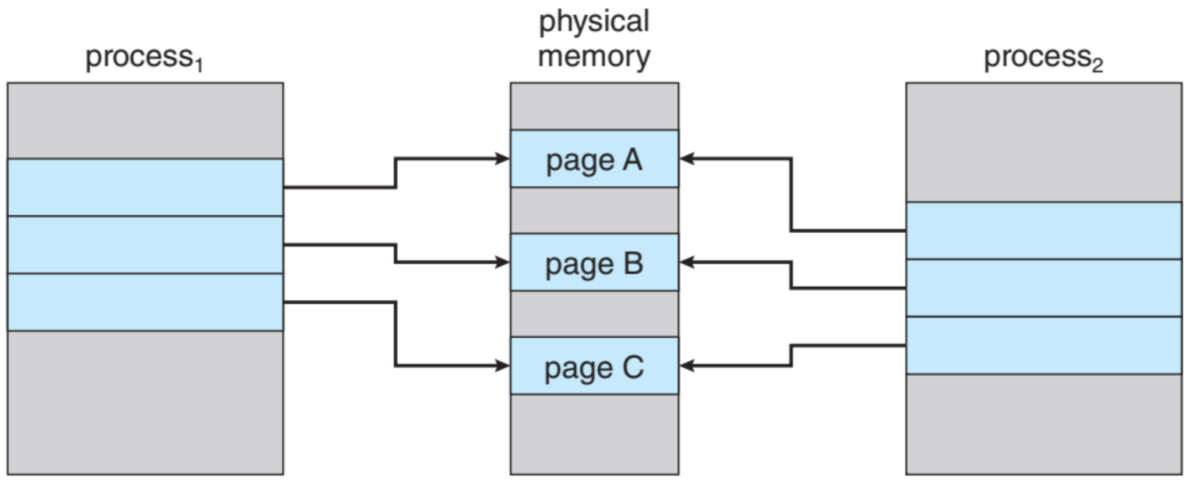


Figure 10.7 Before process 1 modifies page C.

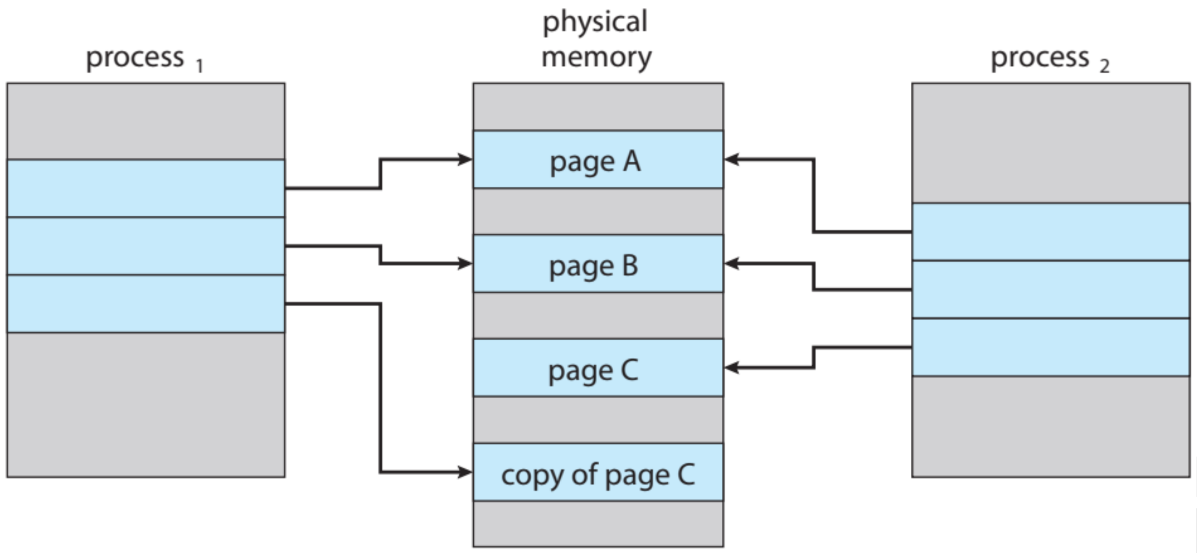
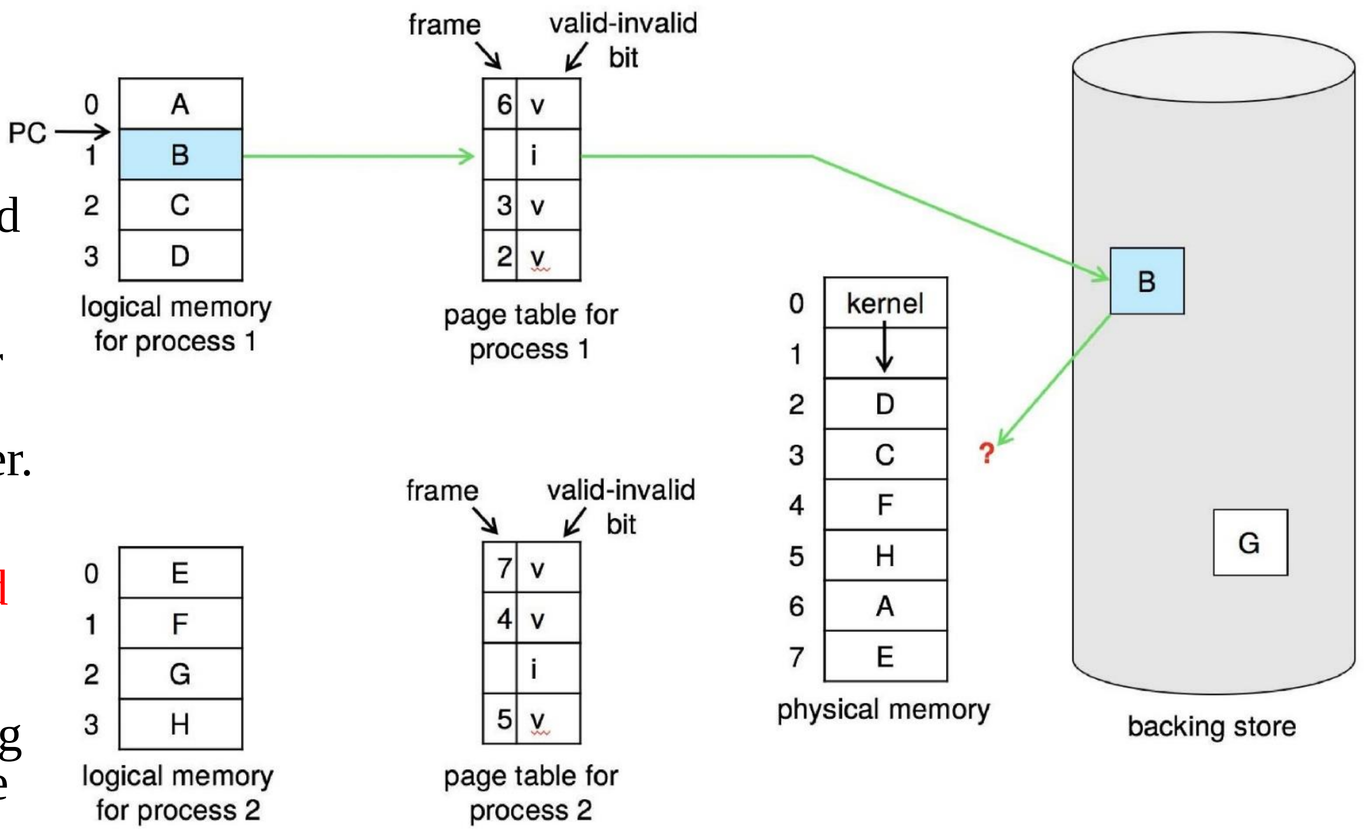


Figure 10.8 After process 1 modifies page C.

4. Page Replacement

- What happens if there is no free frame? The operating system has several options at this point.
  - It could terminate the 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 instead use standard swapping and swap out a process, freeing all its frames and reducing the level of multiprogramming. However, standard swapping is no longer used by most operating systems due to **the overhead of copying entire processes between memory and swap space**.
  - Most operating systems now combine swapping pages with **page replacement**.



4. Page Replacement

Basic Page Replacement

- 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.
  - 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 secondary storage.
  - 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 secondary storage (if necessary); 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 process from where the page fault occurred.

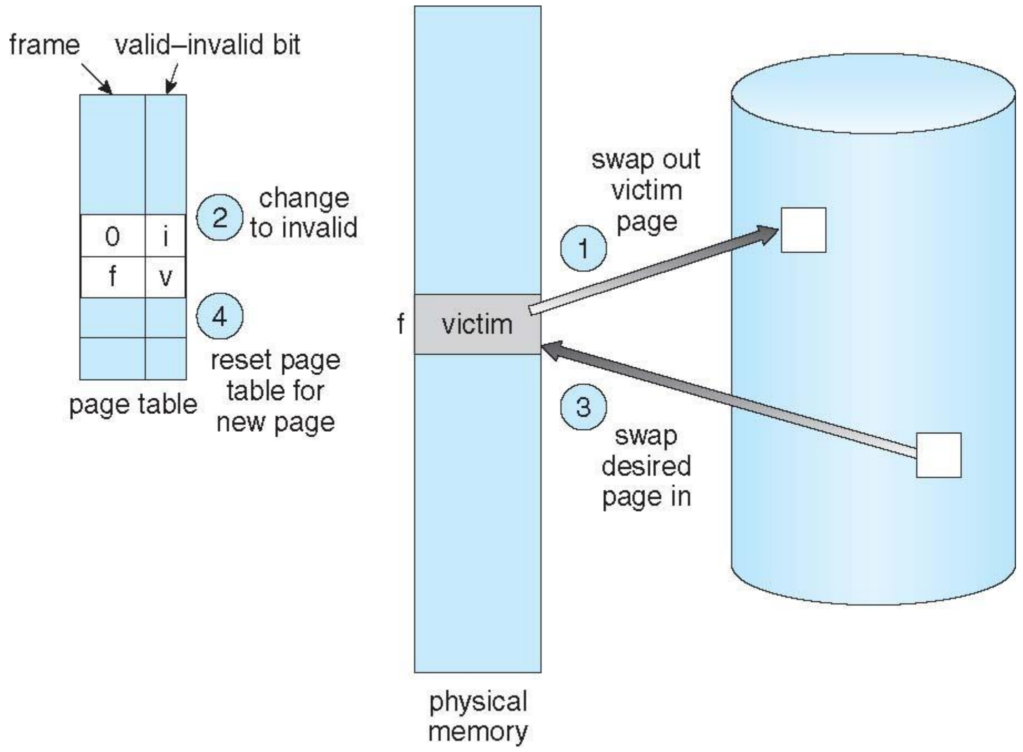


Figure 10.10 Page replacement

4. Page Replacement
Basic Page Replacement

- Notice that, if no frames are free, *two page transfers* (one for the page-out and one for the page-in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
- We can reduce this overhead by using a **modify bit** (or **dirty bit**).
  - 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 secondary storage. In this case, we must write the page to storage.
    - 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 storage: it is already there.

4. Page Replacement

Basic Page Replacement

- We must solve 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.
- How do we select a particular replacement algorithm?
  - In general, we want the one with the **lowest page-fault rate**.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

4. Page Replacement

Basic Page Replacement

- In general, we expect a curve such as that in Figure 10.11.
  - 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.
- We next illustrate several page-replacement algorithms. In doing so, we 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.

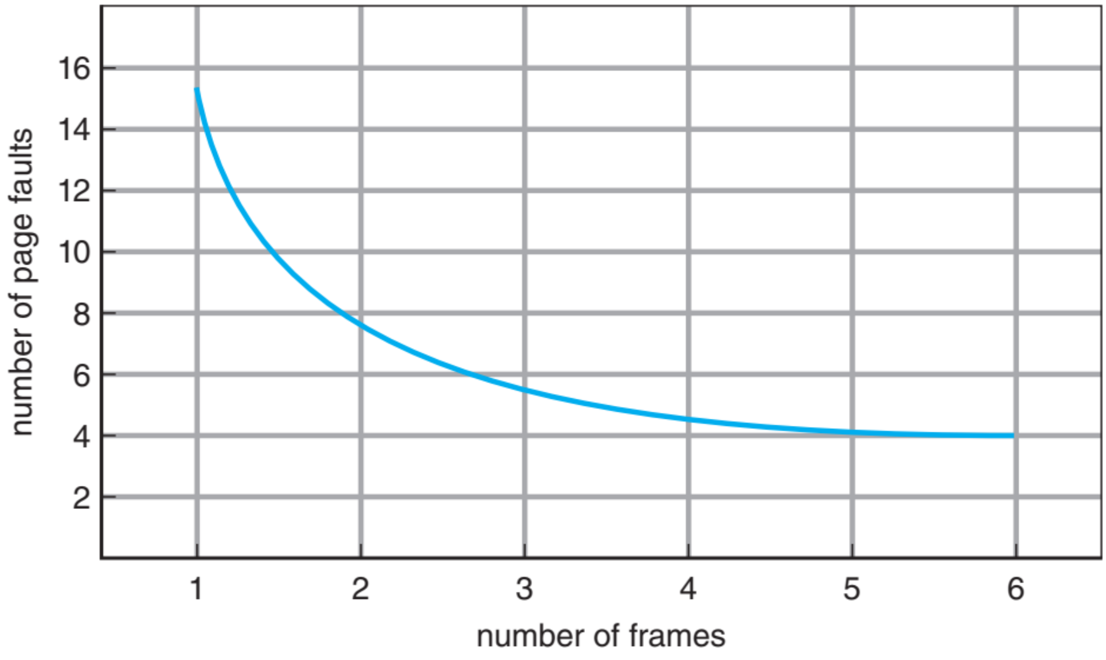
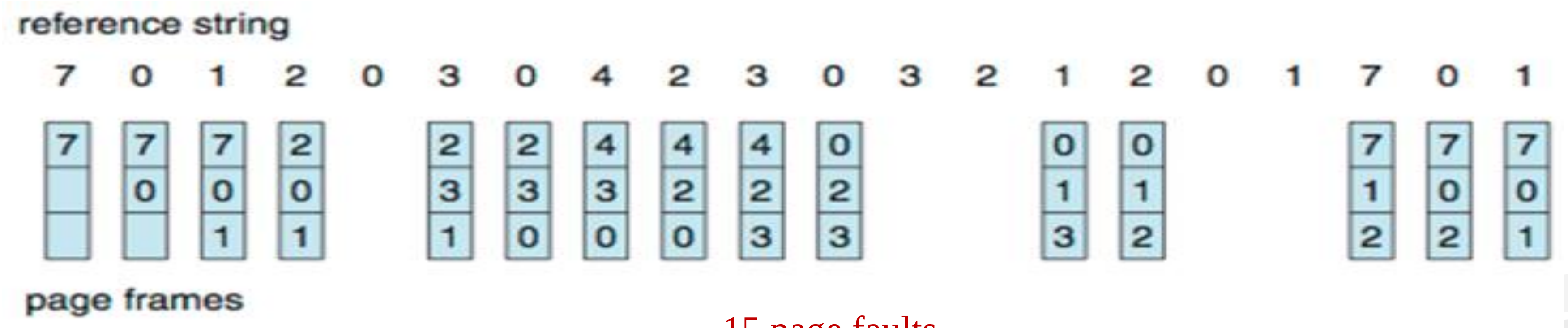


Figure 10.11 Graph of page faults versus number of frames.

4. Page Replacement
FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (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.
  - How to track ages of pages?
    - Just use a FIFO queue
- Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.

4. Page Replacement
FIFO Page Replacement

- Belady’s anomaly
  - Consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Figure 10.13 shows the curve of page faults for this reference string versus the number of available frames.
  - Notice that 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.

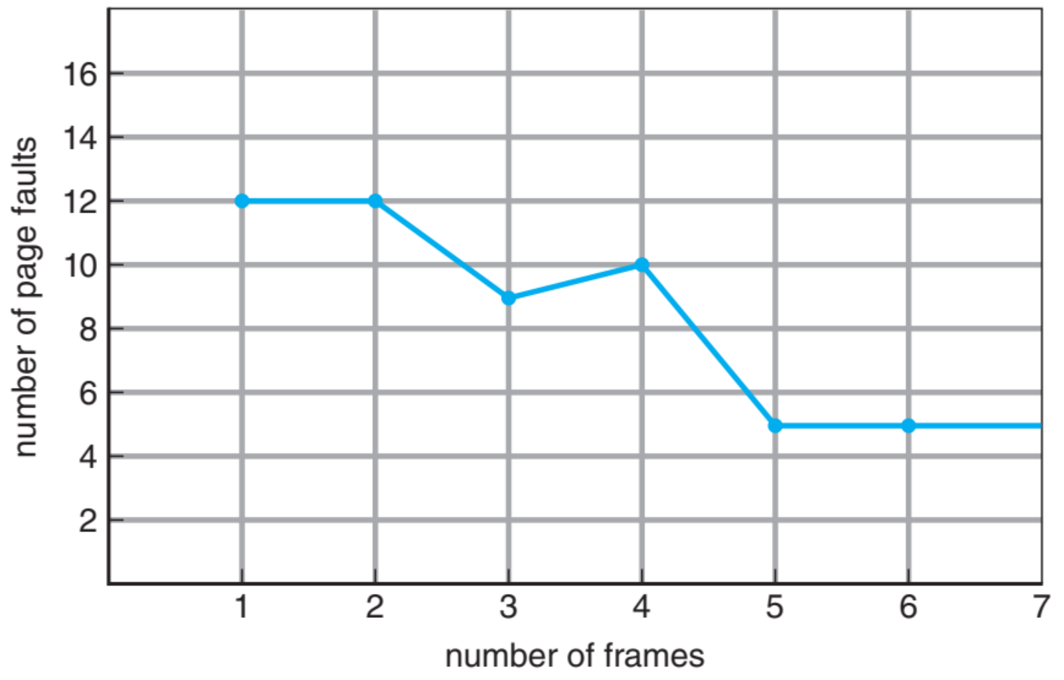
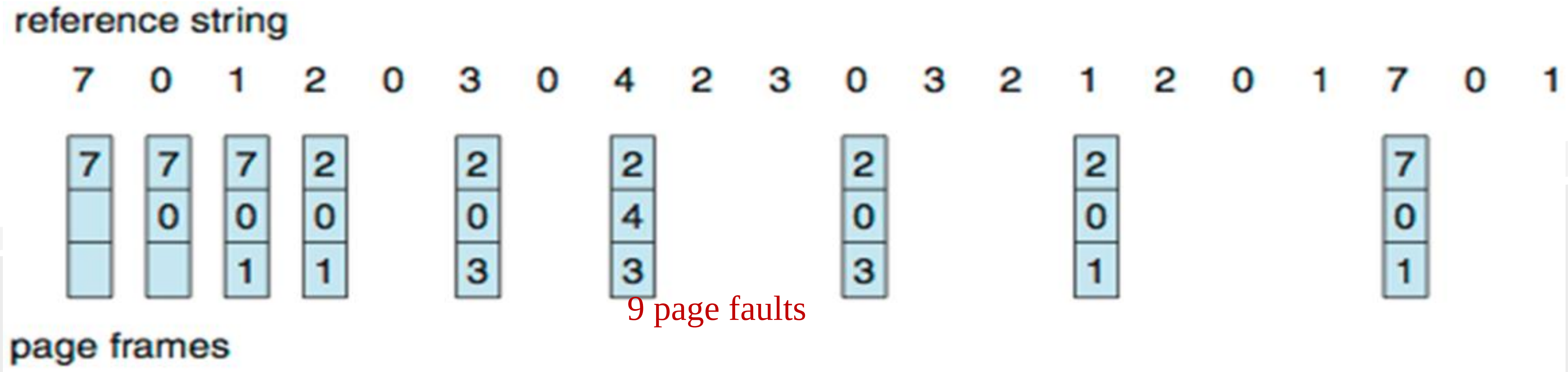


Figure 10.13 Page-fault curve for FIFO replacement on a reference string.

4. Page Replacement

Optimal Page Replacement

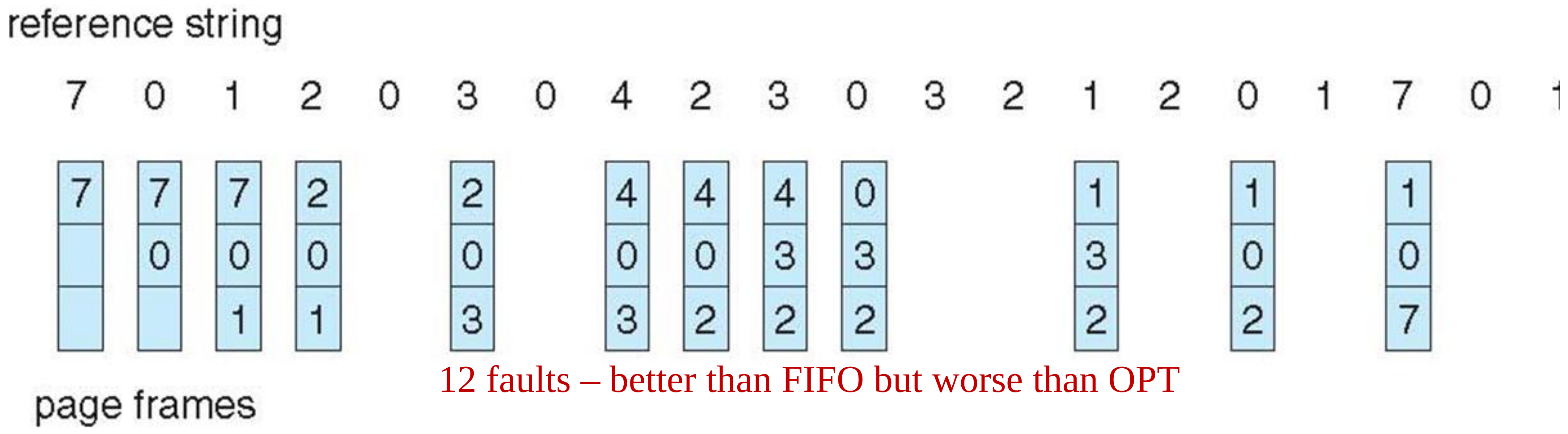
- Replace the page that will **not be used for the longest period of time**.
  - This algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN.
  - **How do you know this?**
    - **Can't read the future → is not feasible**



9 page faults

4. Page Replacement
Least Recently Used (LRU) Page Replacement

- Replace page that **has not been used in the most amount of time**.
  - Use past knowledge rather than future
  - Associate time of last use with each page



12 faults – better than FIFO but worse than OPT

- Generally good algorithm and frequently used
- But how to implement?

4. Page Replacement
Least Recently Used (LRU) Page Replacement

- Counter implementation
  - Every page entry has a time-of-use field.
  - CPU has a logical clock (counter). The clock is incremented for every memory reference.
  - Every time page is referenced through this entry, copy the content of the clock into the time-of-use field in the page entry. In this way, we always have the “time” of the last reference to each page.
  - **We replace the page with the smallest time-of-use value.**
    - **Search through table needed**

4. Page Replacement
Least Recently Used (LRU) Page Replacement

- Stack implementation
  - 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 most recently used page is always at the top of the stack, and the least recently used page is always at the bottom.
  - Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a 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
  - Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called **stack algorithms**, that can never exhibit Belady's anomaly.

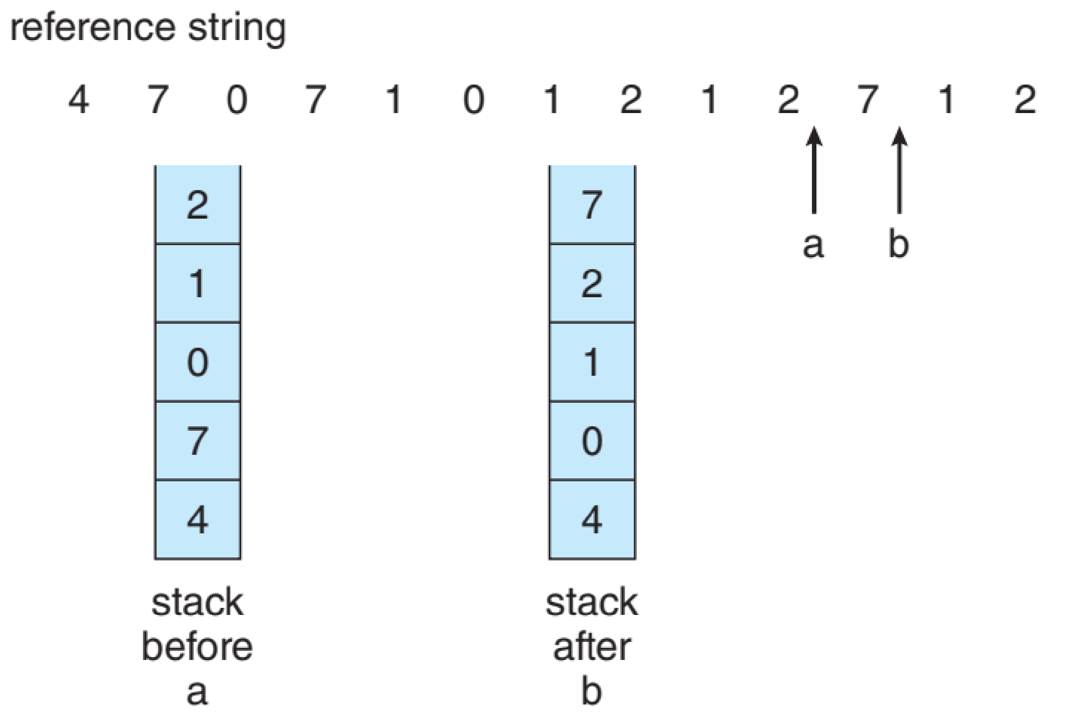


Figure 10.16 Use of a stack to record the most recent page references.

4. Page Replacement

LRU-Approximation Page Replacement

- Not many computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used.

- Many systems provide some help, however, in the form of a **reference bit**. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
  - This information is the basis for many page-replacement algorithms that approximate LRU replacement.

4. Page Replacement

LRU-Approximation Page Replacement

Additional-Reference-Bits Algorithm

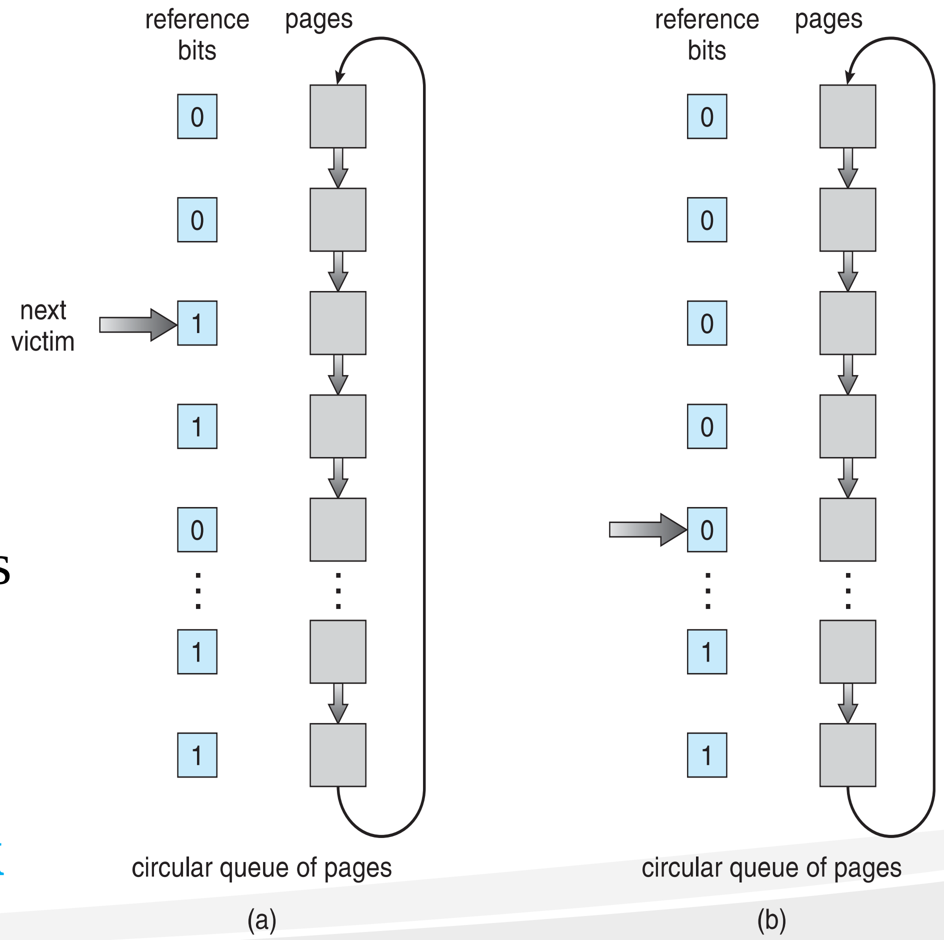
- We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the **high-order bit** of its 8-bit byte, shifting the other bits right by 1 bit and **discarding the low-order bit**. These 8-bit shift registers contain the history of page use for the last eight time periods.
  - If the shift register contains 00000000, for example, then the page has not been used for eight time periods.
  - A page that is used at least once in each period has a shift register value of 11111111.
  - A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
  - If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced.

4. Page Replacement

LRU-Approximation Page Replacement

Second-Chance Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit.
  - If the value is 0, we proceed to replace this page;
  - If the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- One way to implement the second-chance algorithm (sometimes referred to as the **clock** algorithm) is as a circular queue.



4. Page Replacement
LRU-Approximation Page Replacement
Enhance Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:
  - (0, 0) neither recently used nor modified—best page to replace
  - (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
  - (1, 0) recently used but clean—probably will be used again soon
  - (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced

4. Page Replacement

Counting-Based Page Replacement

- There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.
  - Lease Frequently Used (LFU) Algorithm:
    - Replaces page with smallest count
      - A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
  - Most Frequently Used (MFU) Algorithm:
    - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
- Neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.