# Virtual Memory

## Background

- In multiprogramming, we need to keep many processes in memory simultaneously to execute them.
- However, they tend to require that an entire process be in memory before it can execute.
- But entire program rarely used at same time, eg. Error code, unusual routines, large data structures.
- Arrays, lists, and tables are often allocated more memory than they actually need.
- An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. Consider ability to execute partially-loaded program.

## Background

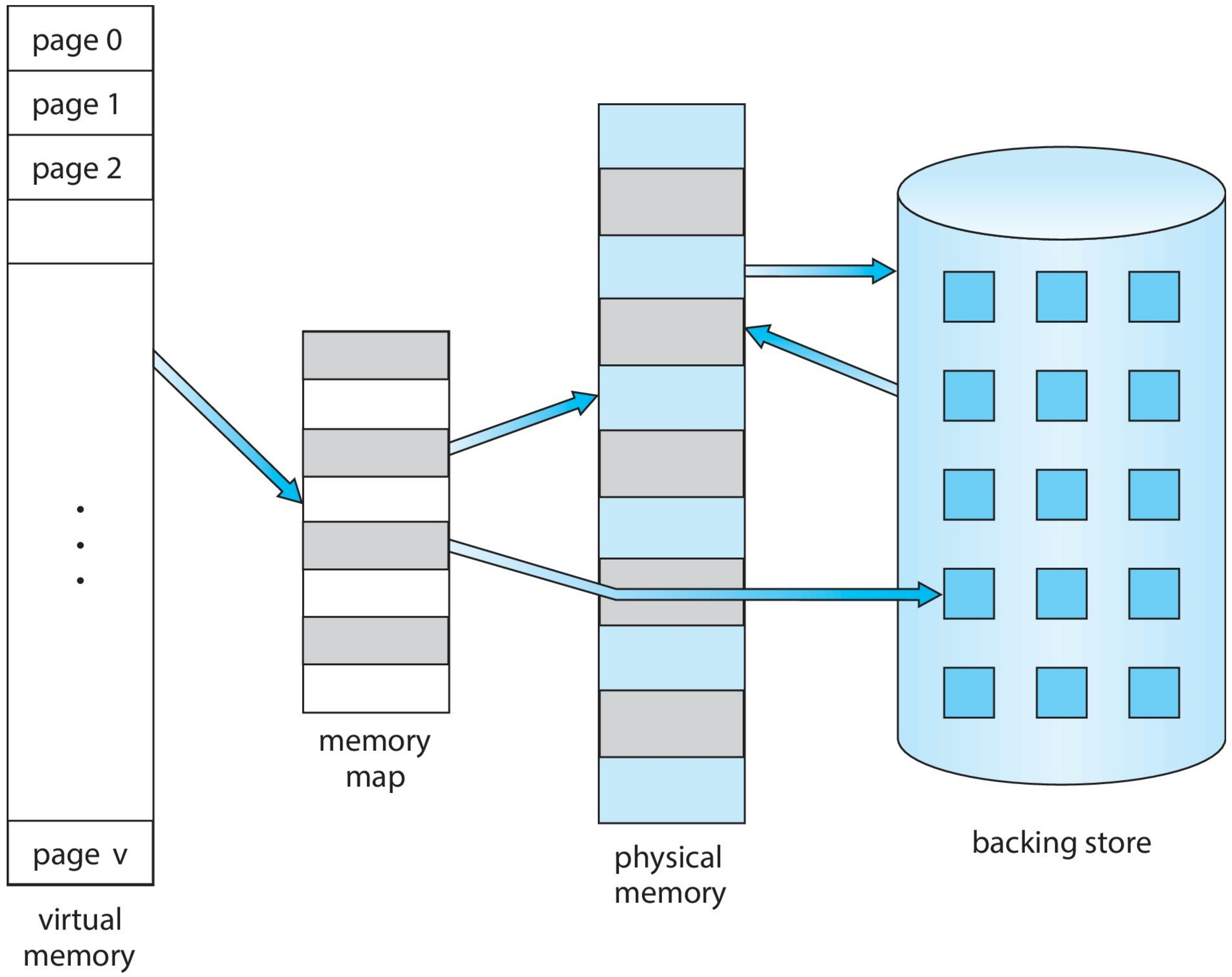
The ability to execute a program that is only partially in memory would confer many benefits:

- 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.

## Virtual memory

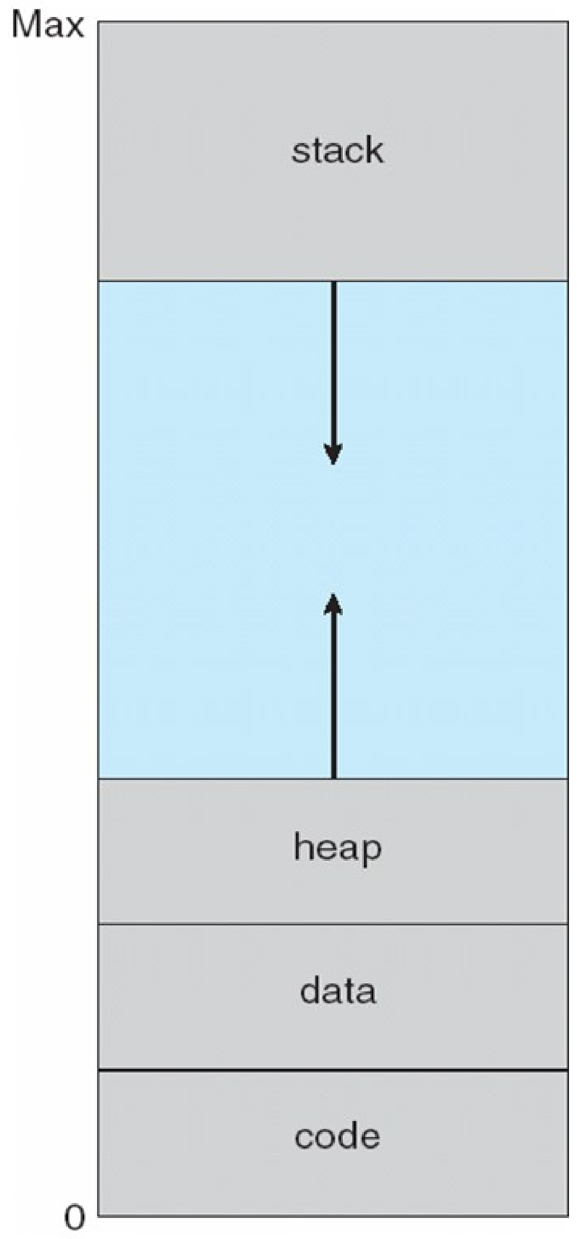
- Virtual memory is a technique that allows the execution of processes that are **not completely in memory**.
- One major advantage of this scheme is that **programs can be larger than physical memory**.
- Only part of the program needs to be in memory for execution.
- Logical address space can therefore be much larger than physical address space.
- Allows address spaces to be shared by several processes.
- More programs running concurrently.
- This technique frees programmers from the concerns of memory-storage limitations.

# Virtual Memory that is Larger than Physical Memory



# Virtual-address Space

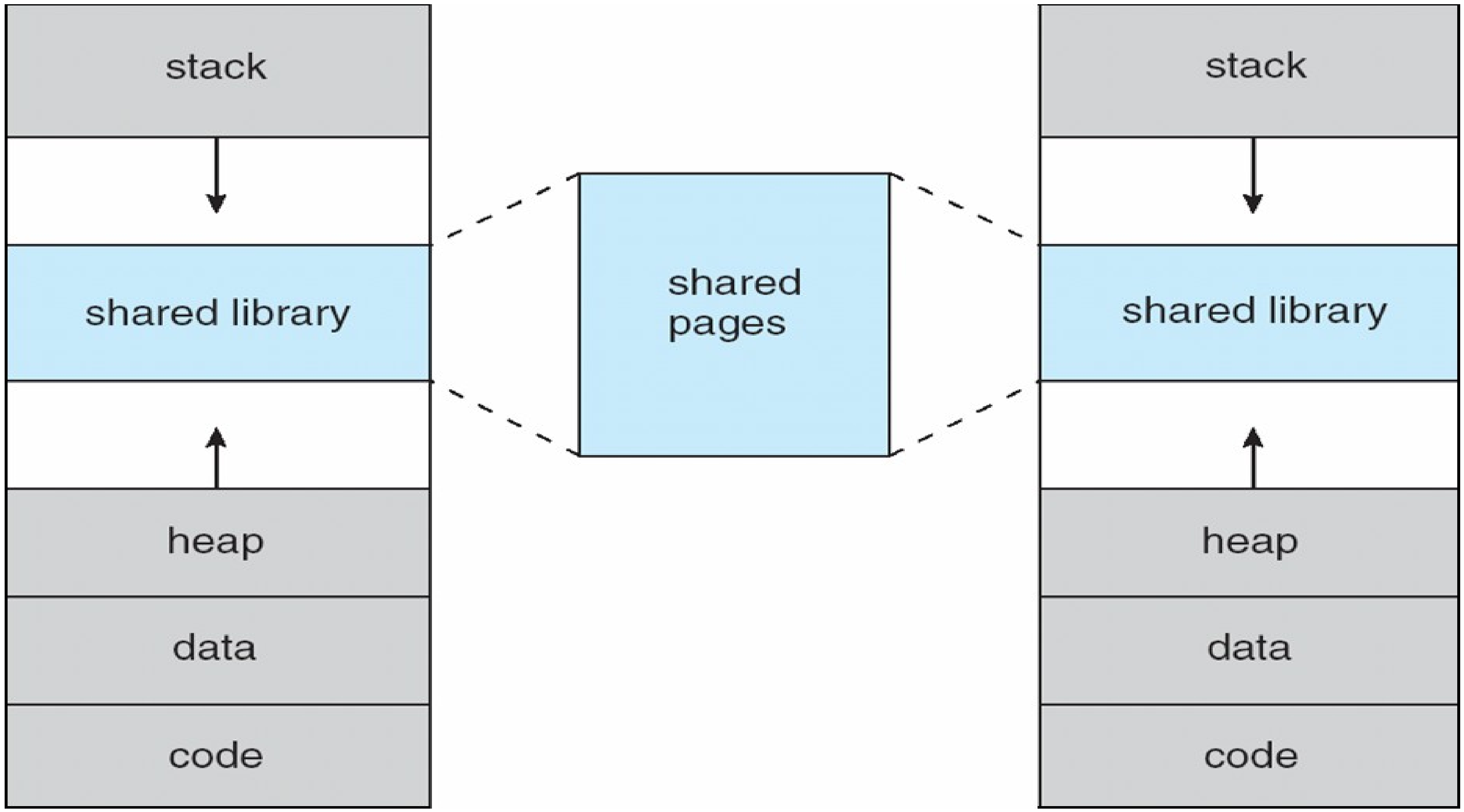
- We allow the heap to grow upward in memory as it is used for dynamic memory allocation.
- Similarly, we allow for the stack to grow downward in memory through successive function calls.
- The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.
- Virtual address spaces that include holes are known as sparse address spaces.



## Page sharing using Virtual Memory

- 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.
- Benefits of shared pages:
- 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.
- 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.

## Shared library using Virtual Memory



## Page sharing using Virtual Memory

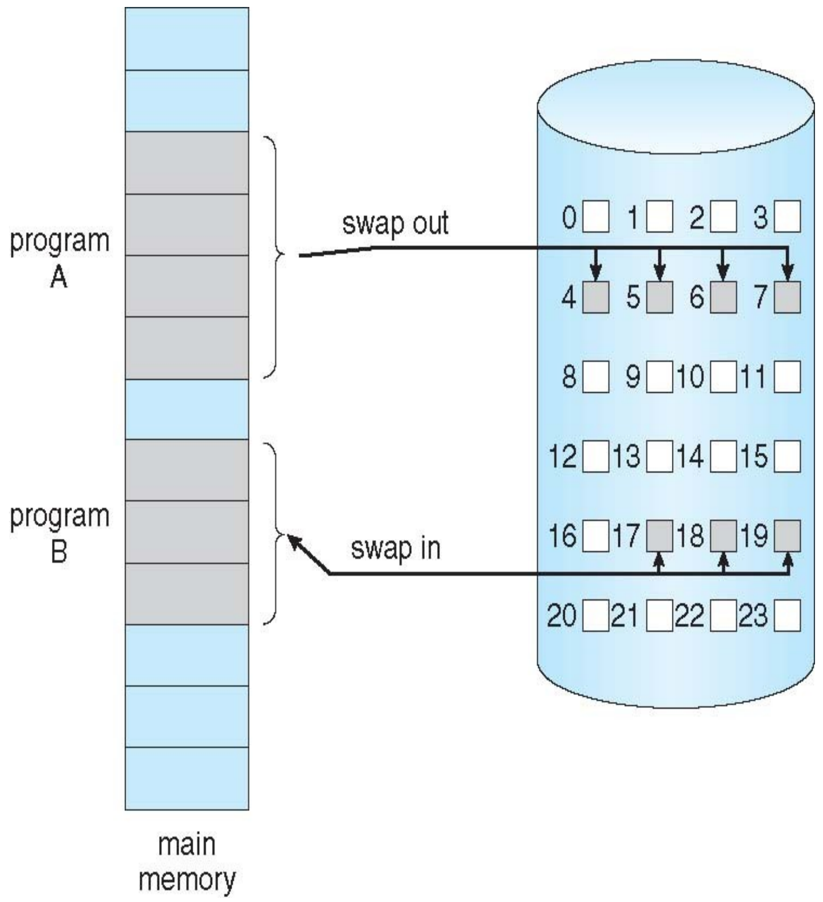
- Two or more processes can communicate through the use of shared memory.
- Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes sharing this region consider it part of their virtual address space.
- Pages can be shared during process creation with the fork() system call, thus speeding up process creation.

## Demand Paging

- Demand paging technique is commonly used to implement virtual memory concept. 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 an HDD device).
- 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.

## Demand Paging

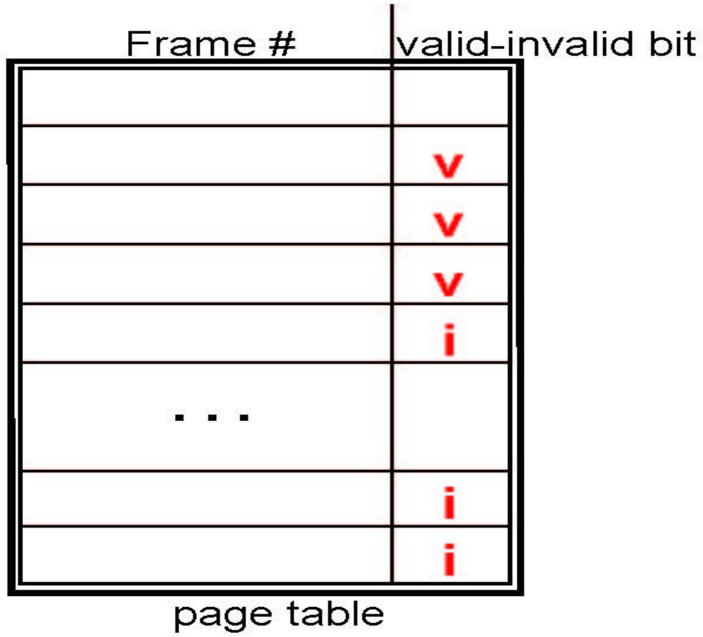
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Faster response
  - More users
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



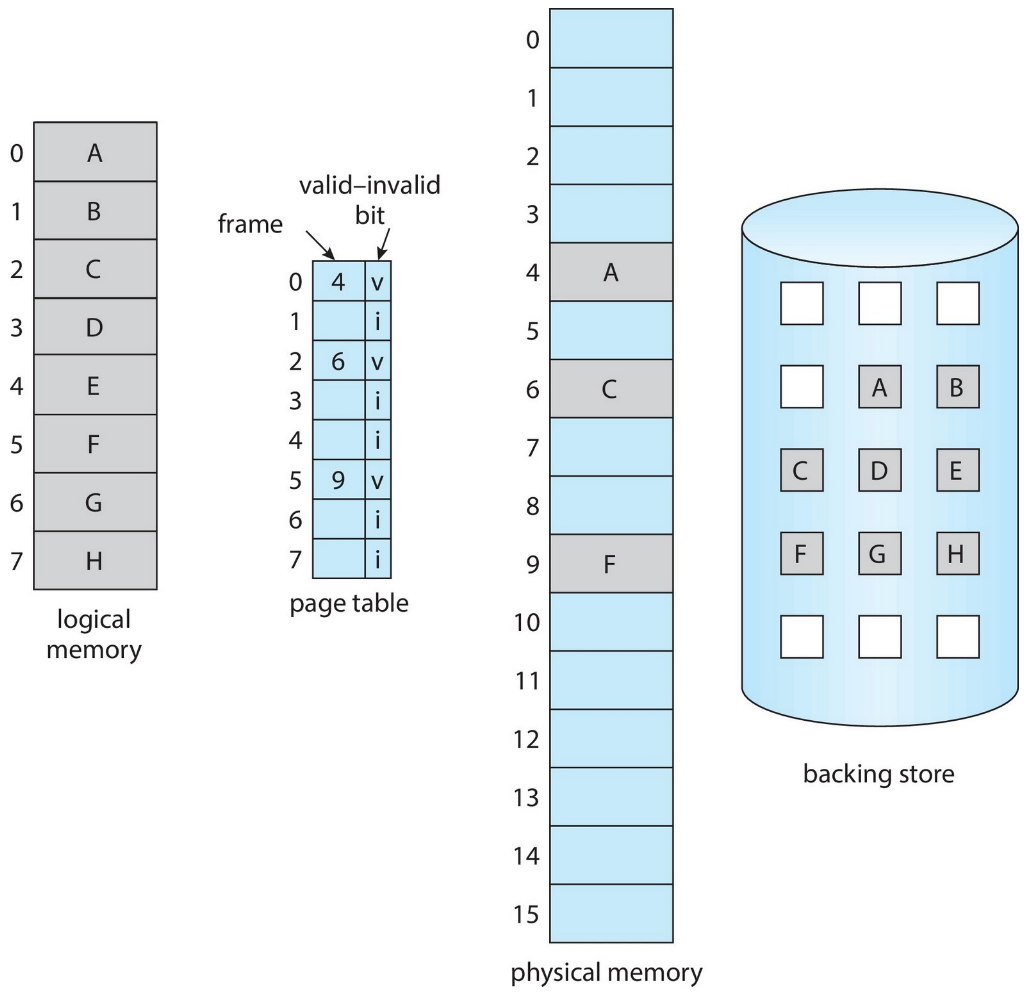
## Page table with Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
- A 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 or is valid but is currently in secondary storage.

(**v** $\Rightarrow$ in-memory, **i** $\Rightarrow$ not-in-memory)



# Page table with Valid-Invalid Bit



# Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system

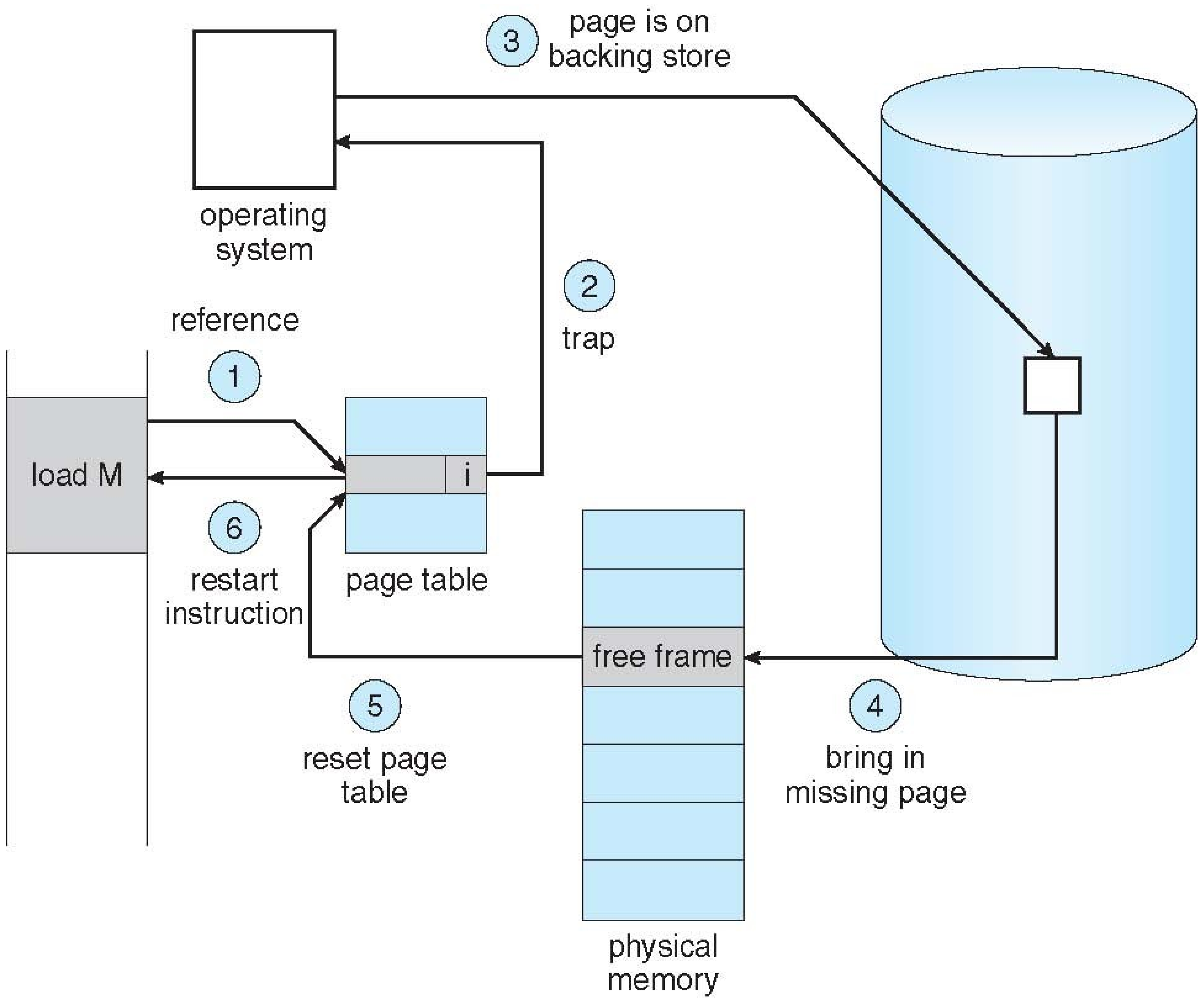
- Page fault

2. Operating system looks at another table to decide:

- Invalid reference $\Rightarrow$ abort
- Just not in memory (go to step 3)

3. Find free frame (what if there is none?)
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

# Steps in Handling a Page Fault



# Aspects of Demand Paging

- **Pure demand paging:** start process with *no* pages in memory
  - ✓ OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - ✓ And for every other process pages on first access
- Hardware support needed for demand paging
  - ✓ Page table with valid / invalid bit
  - ✓ Secondary memory (swap device with swap space)
  - ✓ Instruction restart

## Stages in Demand Paging – Worse Case

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

## Stages in Demand Paging – Worse Case

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show 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.

# Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed.
  - Input the page from disk – lots of time.
  - Restart the process – again just a small amount of time.
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = (1 - p) &\text{ x memory access} \\ &+ p \text{ x (page fault overhead} \\ &+ \text{swap page out} \\ &+ \text{swap page in )} \end{aligned}$$

# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

  EAT = $(1 - p) \times 200 + p$ (8 milliseconds)

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

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

- The effective access time is directly proportional to the page-fault rate. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds.
- This is a slowdown by a factor of 40
- If we want performance degradation < 10 percent,
- Then, we need to keep the probability of page faults :

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

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

  $p < .0000025$

  - one page fault in every 400,000 memory accesses

## Demand Paging Optimizations

- It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.
- One option for the system to gain better paging throughput is by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space.
- During page replacement, swap out only modified pages. The unmodified pages are simply removed from the physical memory.

## Demand Paging Optimizations

- 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).
- When this scheme is 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,

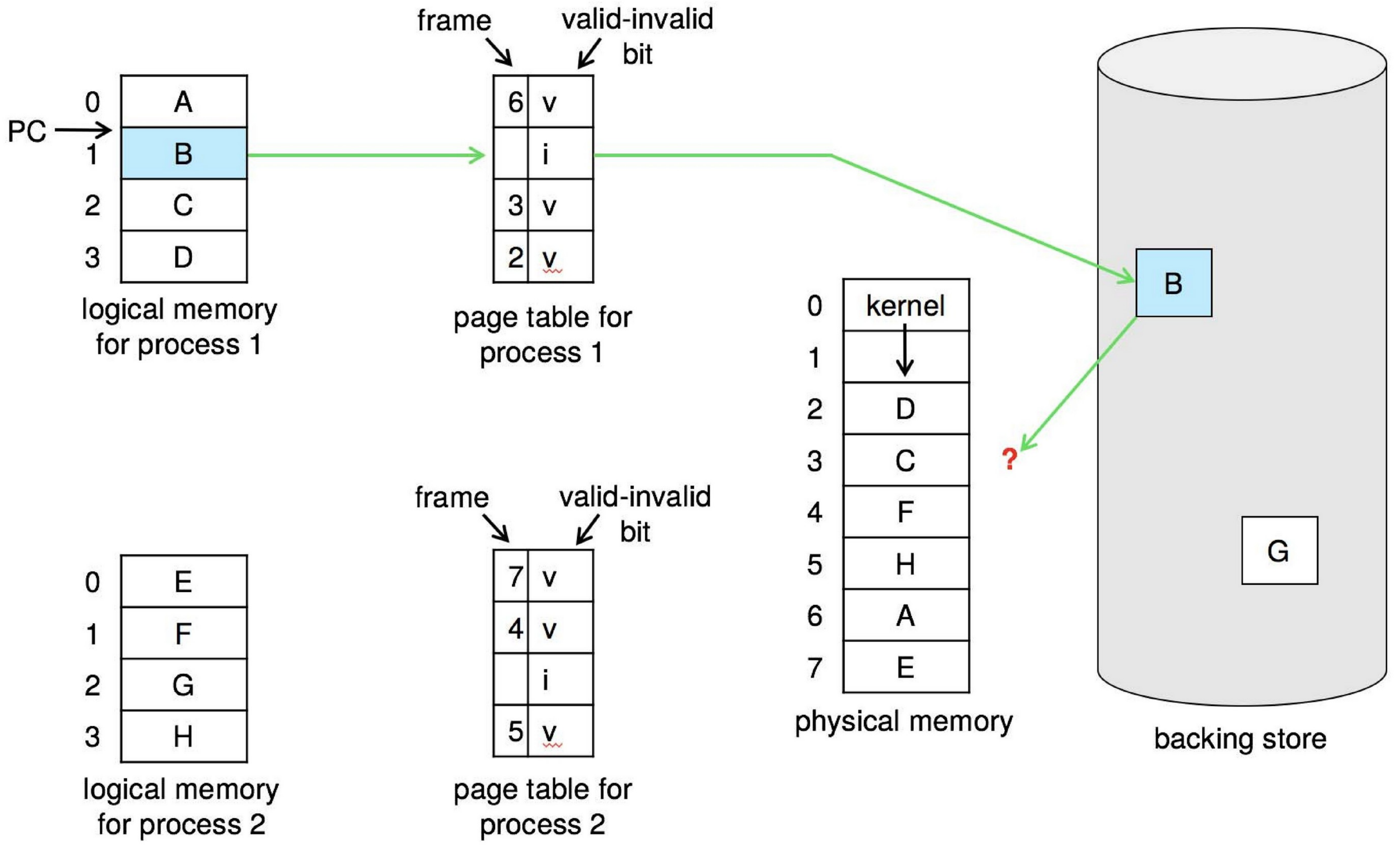
# Demand Paging Optimizations

- 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.
- This technique also applies to read-only pages. Such pages cannot be modified; thus, they may be discarded when desired.
- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by **one-half** if the **page has not been modified**.

## Page Replacement

- While a process is executing, a page fault occurs. The operating system determines where the desired page is residing on secondary storage but then finds that there are **no free frames** on the free-frame list; all memory is in use.
- 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.

# Need For Page Replacement

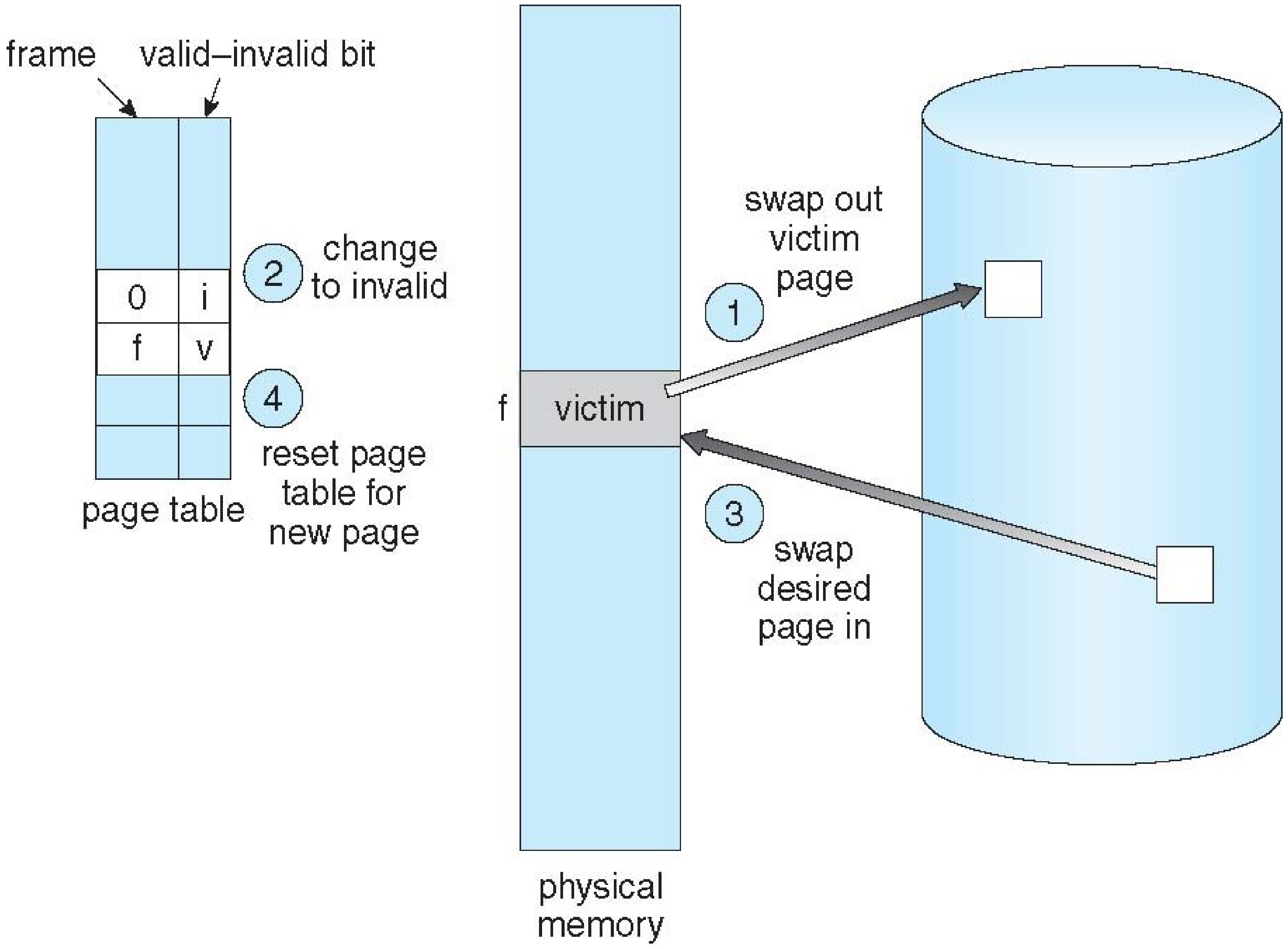


## Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if **dirty**
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

# Page Replacement



## Page Replacement

- 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**).
- When this scheme is 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.

## Page Replacement

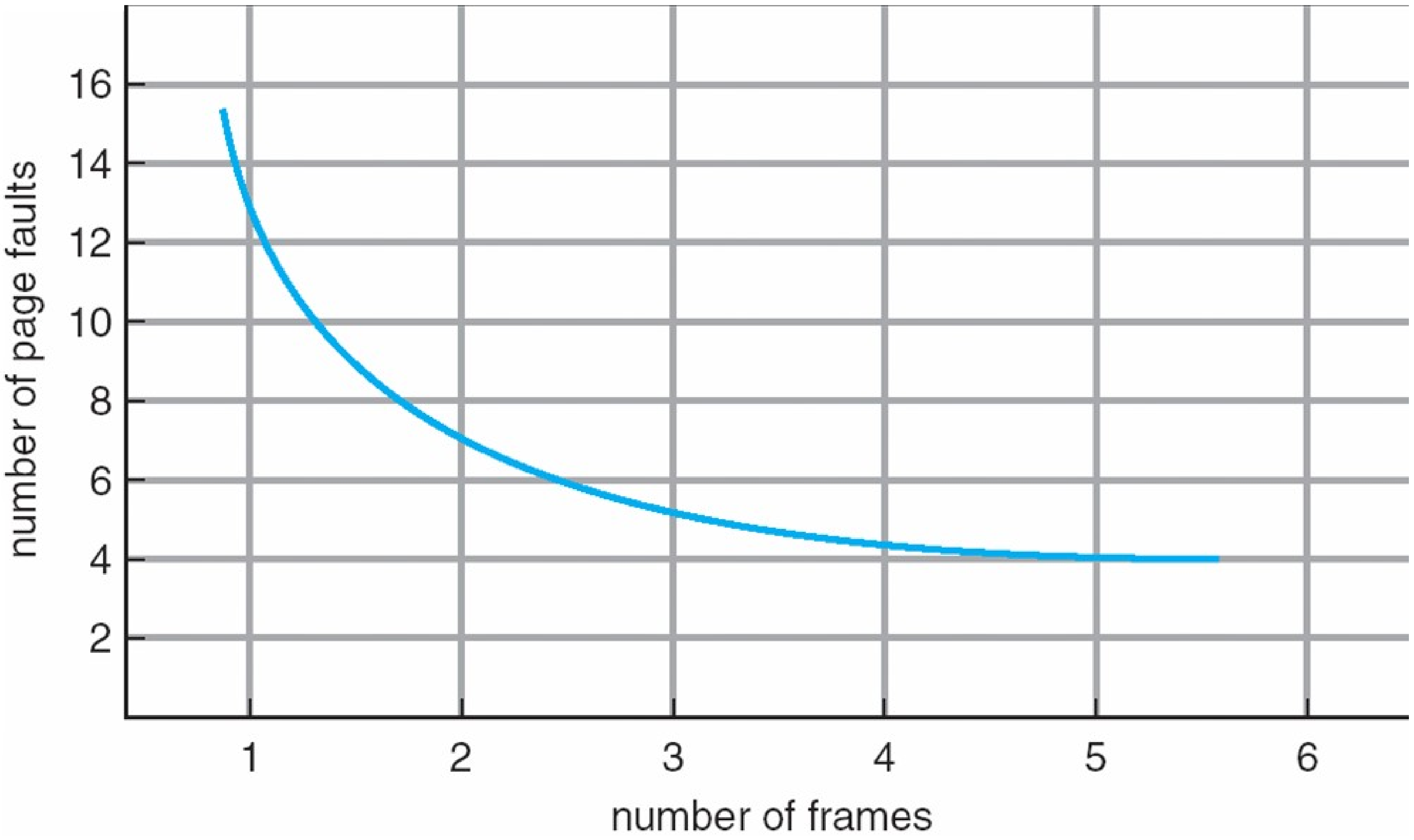
- If the **bit is set**, we know that the **page has been modified** since it was read in from secondary storage. In this case, we must **write the page to storage**. If the modify bit is **not set**, however, the page **has not been modified** since it was read into memory. In this case, we need not write the memory page to storage: it is already there. This technique also applies to read-only pages. Such pages cannot be modified; thus, **they may be discarded when desired**. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by **one-half if the page has not been modified**.

## Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - **Want lowest page-fault rate** on both first access and re-access
- 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
- For examples, the **reference string** of referenced page numbers:

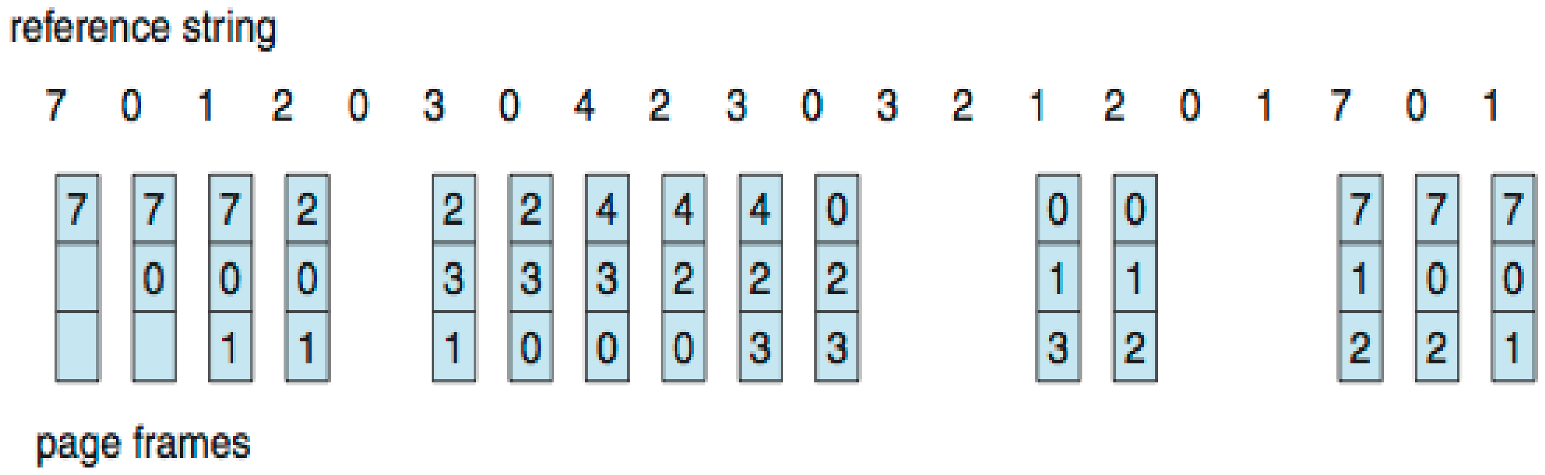
**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

# Graph of Page Faults Versus the Number of Frames



# First-In-First-Out (FIFO) Algorithm

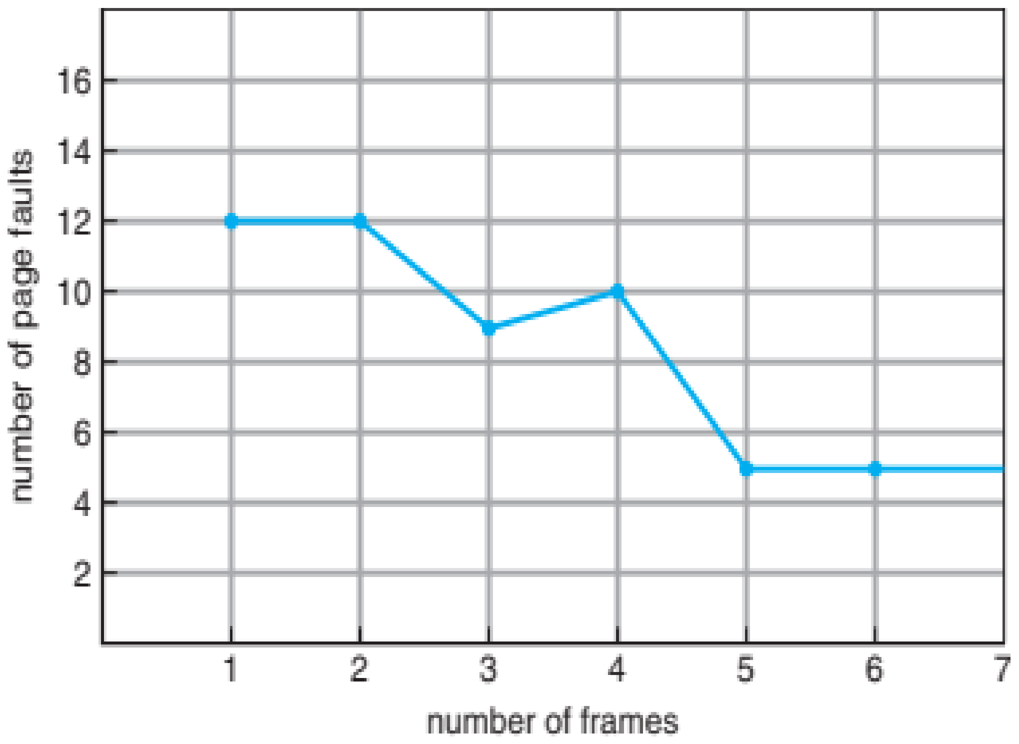
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- How to track ages of pages?
    - Just use a FIFO queue

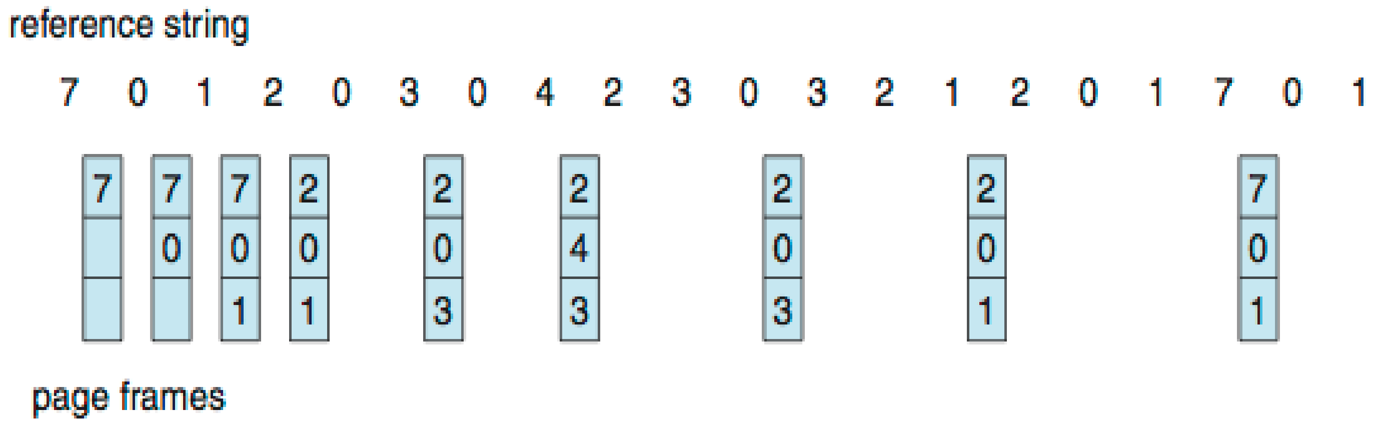
# Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
- Graph illustrating Belady's Anomaly



# Optimal Algorithm

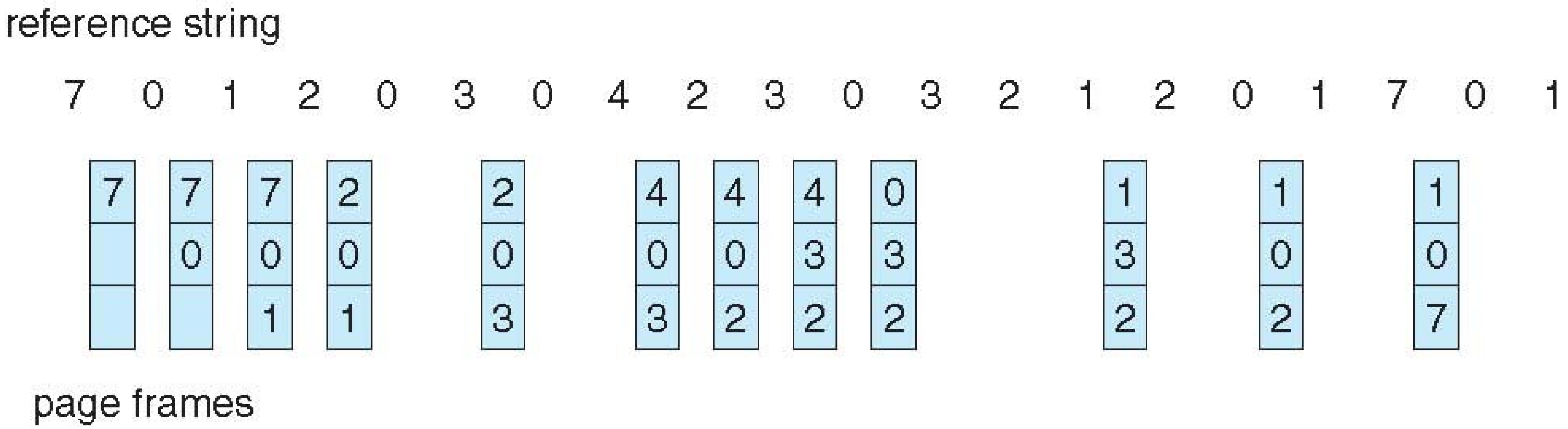
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



- Optimal algorithm don't suffer from Belady's Anomaly

# Least Recently Used (LRU) Algorithm

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



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- LRU is another example of stack algorithms; thus it does not suffer from Belady's Anomaly

## LRU Algorithm Implementation

**Time-counter implementation:**

- Every page entry has a time-counter variable; every time a page is referenced through this entry, copy the value of the clock into the time-counter.
- When a page needs to be changed, look at the time-counters to find smallest value.
- The **clock is incremented for every memory reference**.

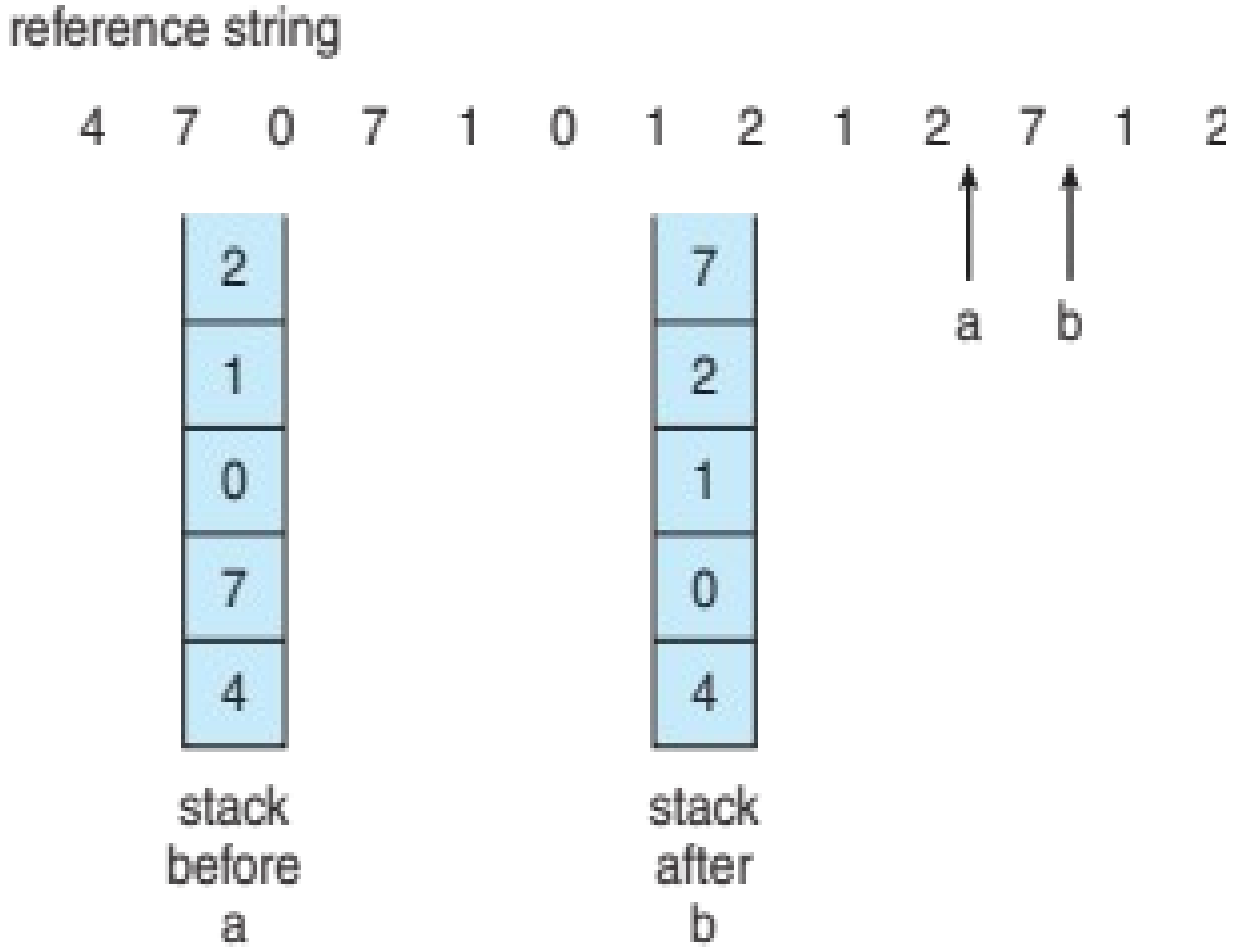
# LRU Algorithm Implementation

**Stack implementation:**

- Keep a stack of page numbers in a double link form
- Whenever a page is referenced, it is removed from the stack and **put on the top**.
- So, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom.
- The best implement of 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**.
- But each update more expensive
- **No search for replacement**.

## Stack Implementation

- Use of a stack to record most recent page references



## LRU Approximation Page Replacement

- Needs special hardware

**One bit solution:**

- Reference bit
- 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).
- After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use.

# LRU Approximation Page Replacement

**One byte solution:**

- We can keep an 8-bit byte for each page in a table in memory.
- The OS 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.
- 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.

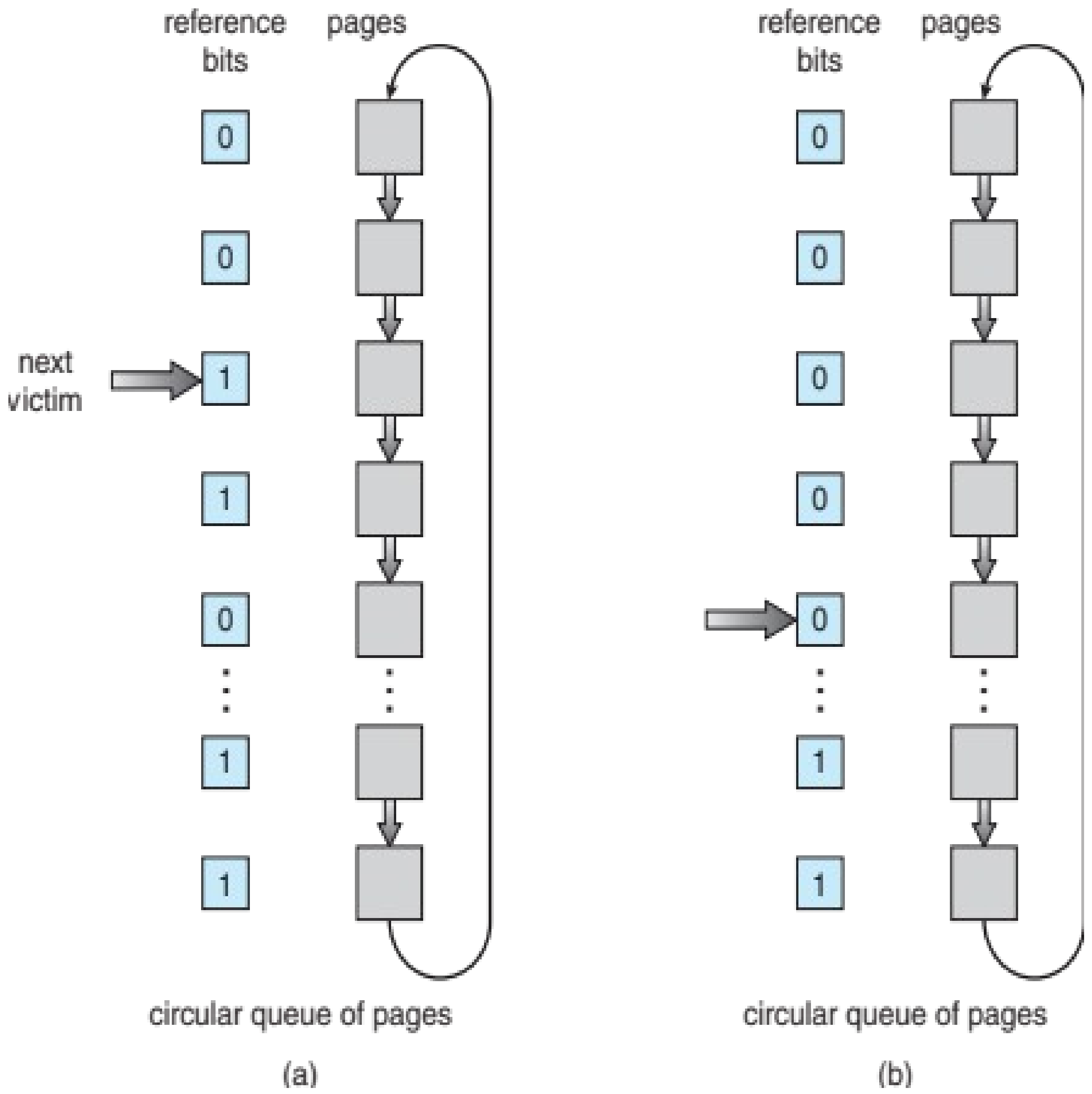
# LRU Approximation Page Replacement

**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.
- But 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.

# LRU Approximation Algorithms

- A pointer indicates which page is to be replaced next.
- When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.
- As it advances, it clears the reference bits. Once a victim page is found, the page is replaced.



## LRU Approximation Page Replacement

**Enhanced Second-Chance algorithm:**

- We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair.

1) (0, 0) neither recently used nor modified—best page to replace
2) (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3) (1, 0) recently used but clean—probably will be used again soon
4) (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

# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- Lease Frequently Used (LFU) Algorithm:
  - Replaces page with smallest count
- 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

## Allocation Algorithms

- **Equal allocation–**
- The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m/n* frames (ignoring frames needed by the operating system for the moment).
- For instance, if there are 93 frames and 5 processes, each process will get 18 frames.
- The 3 leftover frames can be used as a free-frame buffer pool.

# Allocation Algorithms

- Proportional allocation – Allocate according to the size of process.
- Let the size of the virtual memory for process $p_i$ be $s_i$, and define

$$S = \sum s_i.$$

- Then, if the total number of available frames is $m$, we allocate $a_i$ frames to process $p_i$, where $a_i$ is approximately

$$a_i = s_i / S \times m.$$

# Thrashing

- Consider what occurs if a process does not have "enough" frames.
- It does not have the minimum number of frames it needs to support pages in the working set.
- The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away.
- Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.