Operating Systems 2017/2018 | Part IX – Virtual Memory

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

Chapter 9

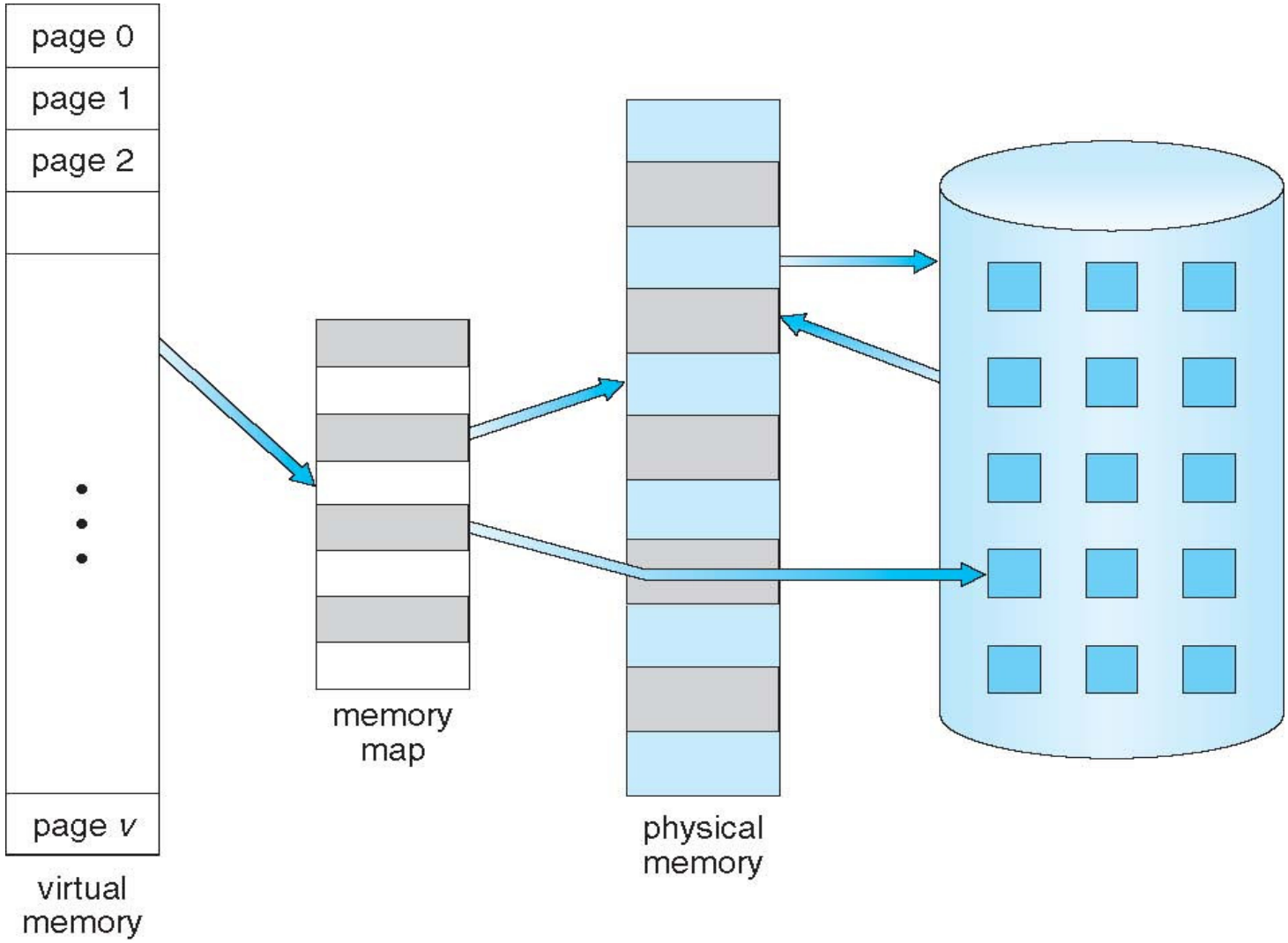
Background

- Code needs to be in memory to execute, but entire program rarely used
  - Code to handle unusual situations is almost never executed
  - Large data structures often allocate more memory than they actually need
  - Even if the entire program is used, it is not all needed at same time
- **Virtual memory** is a technique that allows the **execution of processes that are not entirely in memory**
  - Abstracts main memory into an extremely large uniform array of storage
  - Frees programmers from the concerns of memory storage limitations

Background

- Executing a process that is not entirely in memory **benefits not only the users but also the operating system**
  - Allows for less memory usage
  - Allows for more efficient process creation
  - Less I/O needed to load or swap processes into memory
  - More programs could run concurrently
  - Logical address space can be much larger than physical address space

From Virtual To Physical Memory

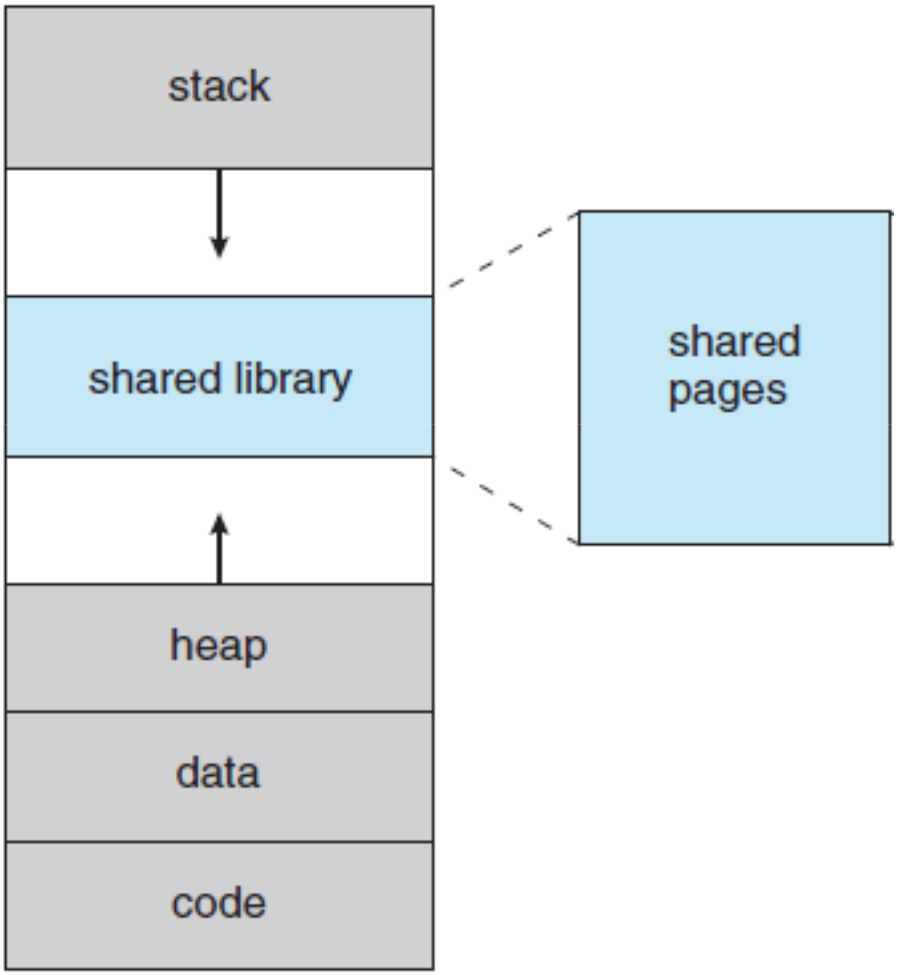


Demand Paging

- **90-10 rule**: programs spend 90% of their time in 10% of their code
- Demand paging is a technique that **loads a page into memory only when it is needed**
  - Avoids unnecessary I/O
  - Less memory and less I/O needed
  - Faster response and more users allowed
  - Pages that are never accessed are thus never loaded into physical memory (pages not in memory reside in secondary memory, usually a disk)

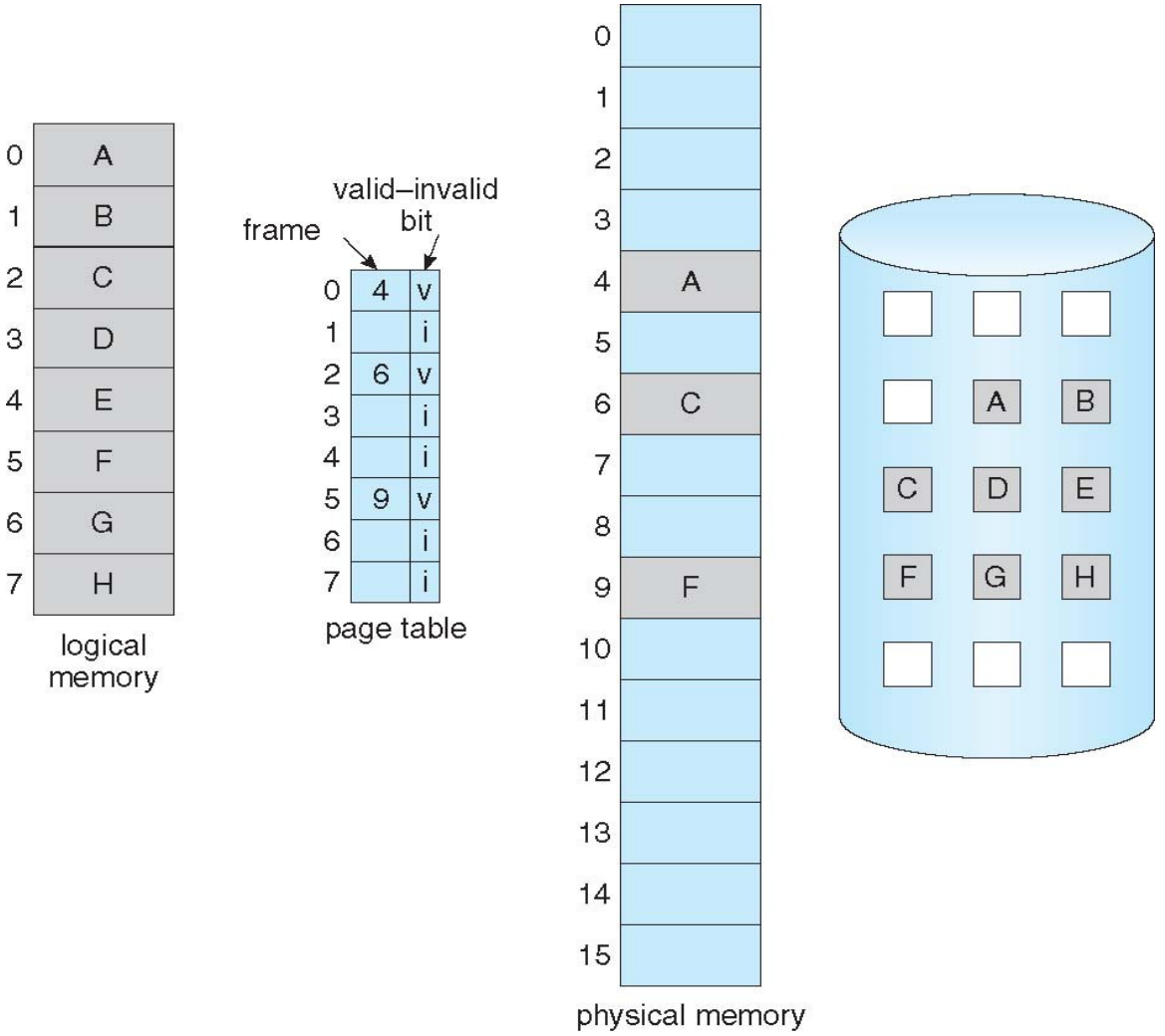
Virtual Address Space

- 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** until the highest logical address allowed
- Enables **sparse address spaces** with holes left for growth or to dynamically link shared objects (e.g., system libraries or shared pages) during program execution



Valid-Invalid Bit Scheme

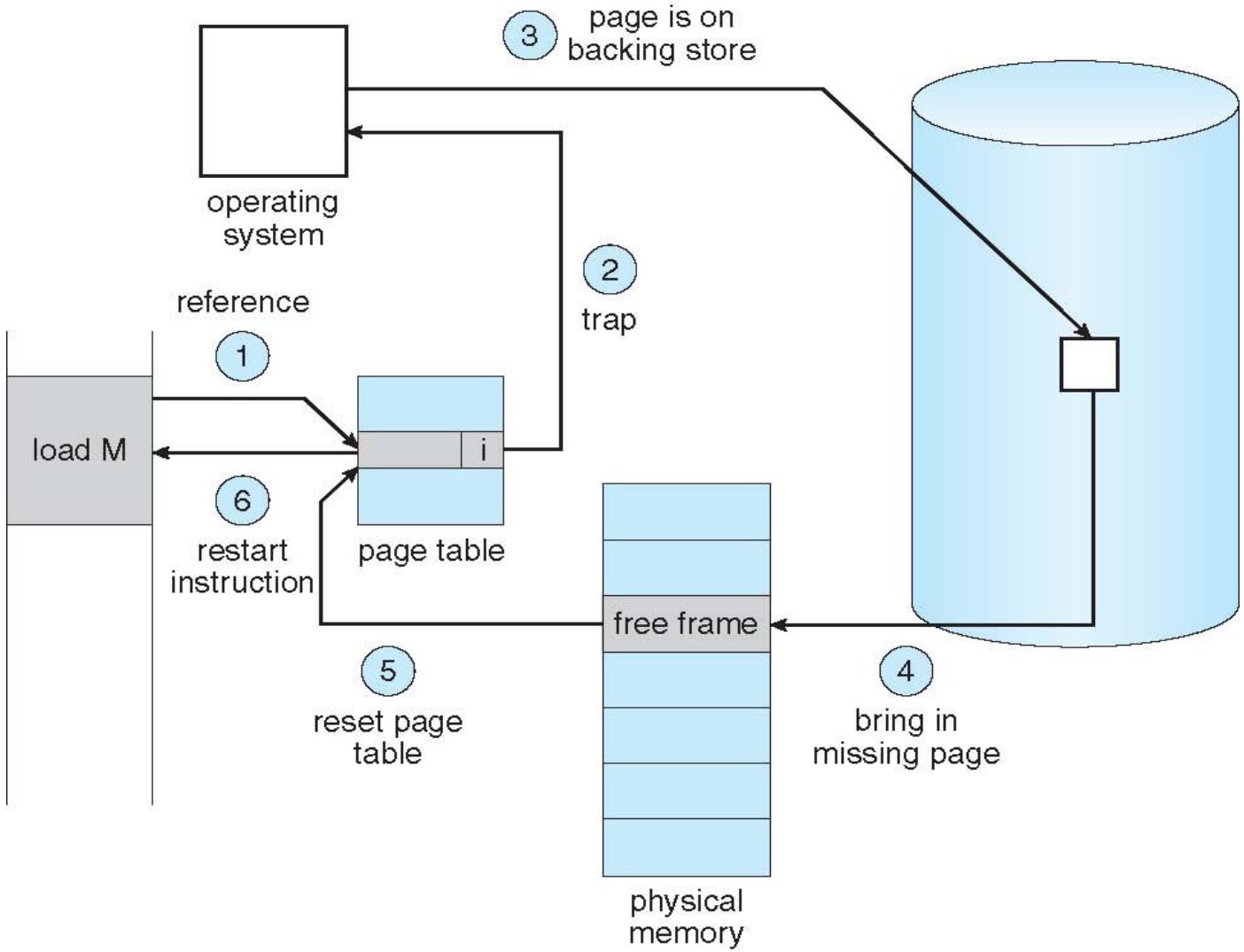
- Associates a valid-invalid bit with each page table entry
  - Initially, the valid-invalid bit is set to invalid on all entries
  - When the bit is set to **valid (v)**, the associated page **is in memory and is legal** (i.e., it belongs to the logical address space of the process)
  - When the bit is set to **invalid (i)**, the page **is either on the disk or is not legal** (i.e., it not belongs to the logical address space of the process)



Page Fault

- A **page fault** occurs when the paging hardware **accesses to a page marked as invalid**
  - In translating the address through the page table, the paging hardware will notice that the invalid bit is set, causing a trap to the operating system
- The procedure for handling a page fault is straightforward:
  - Check PCB to determine whether the reference was a legal or illegal memory access (if illegal reference, terminate the process)
  - Find a free frame and bring the missing page into it by loading page from backing store
  - Reset page table to indicate that page is now in memory
  - Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Effective Access Time (EAT)

- EAT = (Hit Ratio x Memory Access Time) + (Miss Ratio x Page Fault Time)
- To compute the EAT, we must know how much time is needed to service a page fault. There are three major tasks of the page-fault service time:
  - Service the page-fault interrupt
  - Read in the page
  - Restart the process
- The first and third tasks can take from 1 to 100 microseconds each
- The second task however, will probably be close to 5 to 10 milliseconds, including hardware and software time

Effective Access Time (EAT)

- Consider the following scenario:
  - Memory access time = 200 ns
  - Average page-fault service time = 8 ms = 8,000,000 ns
  - EAT = (1 - p) * 200 + p * 8,000,000 = 200 + p * 7,999,800 ns
- If one access out of 1,000 (p = 0.001) causes a page fault, then:
  - EAT = 200 + 7,999.8 = 8,199.8 = ±8.2 microseconds (41 times slower)
- If we want a performance degradation less than 10% (220 ns):
  - 200 + p * 7,999,800 < 220 ↔ p < 0.0000025
  - Less than one page fault out of 400,000 memory accesses
- In sum, it is important to keep the page-fault rate very low in a demand paging system, otherwise process execution is dramatically slowed down

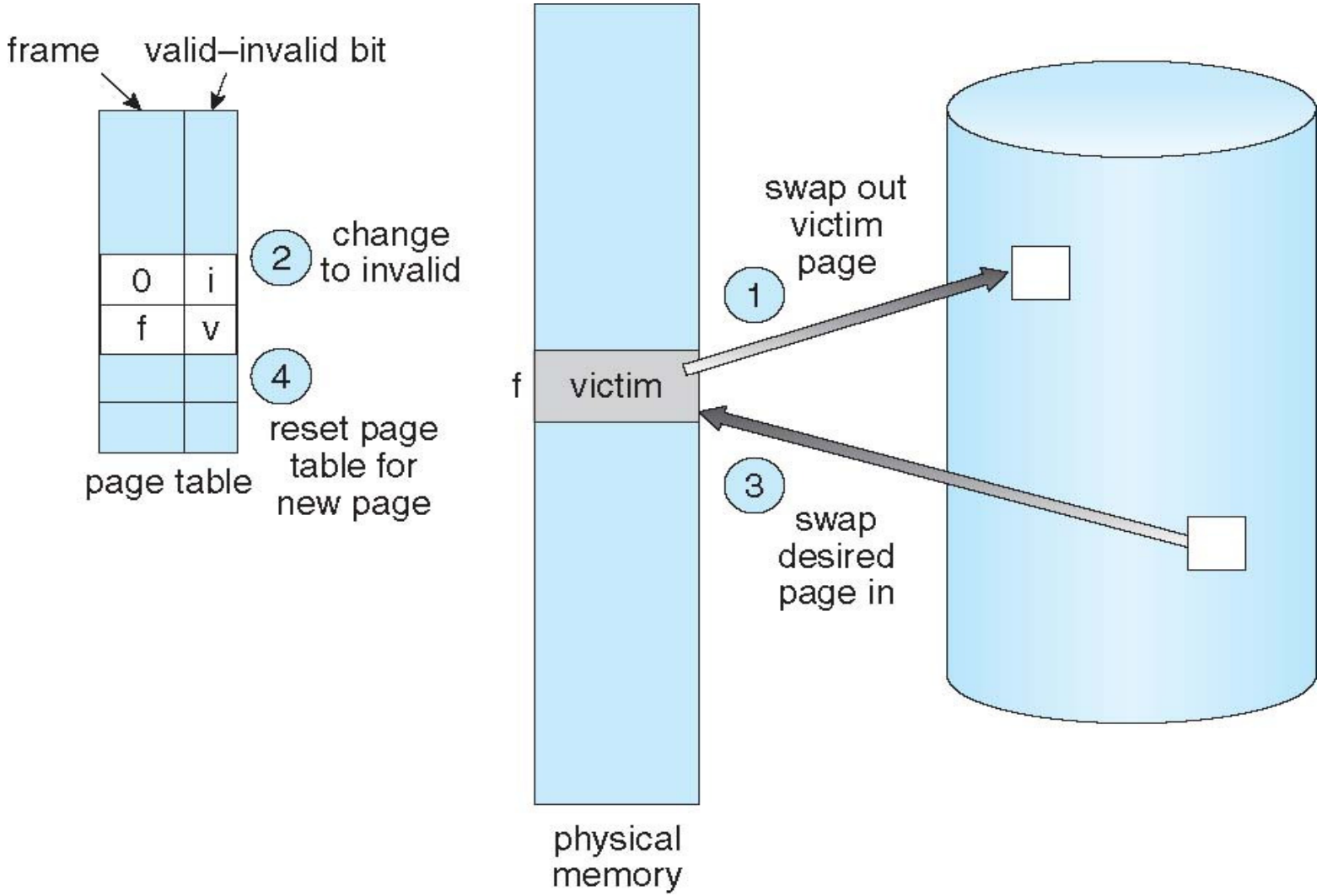
What If No Free Frame Exists?

- After a page fault occurs, it might happen that all physical memory is in use and there are no free frames on the free-frame list
  - The operating system could swap out a process, freeing all its frames
  - The most common solution is **page replacement**, i.e., **find some page not currently being used and free it**
- Page replacement completes separation between logical memory and physical memory
  - Large virtual memory can be provided on a smaller physical memory
  - Same page may be brought into memory several times
- Want a page replacement algorithm which will result in **minimum number of page faults**

Basic Page Replacement

- If no free frame exists, the page replacement algorithm:
  - Selects a **victim page**
  - Writes the victim page to disk
  - Brings the desired page into the newly free frame
  - Updates the page table accordingly
- Notice that, in fact, two page transfers (one out and one in) are required
  - Doubles the page-fault service time and increases the EAT accordingly
- This overhead can be reduced by using a **modify (or dirty) bit** per page
  - The dirty bit is set whenever a page is modified, i.e., a byte was written into it
  - When we select a page for replacement, if its dirty bit is unset, the page has not been modified since it was read from disk, and we can avoid writing it back to disk since it is already there

Basic Page Replacement

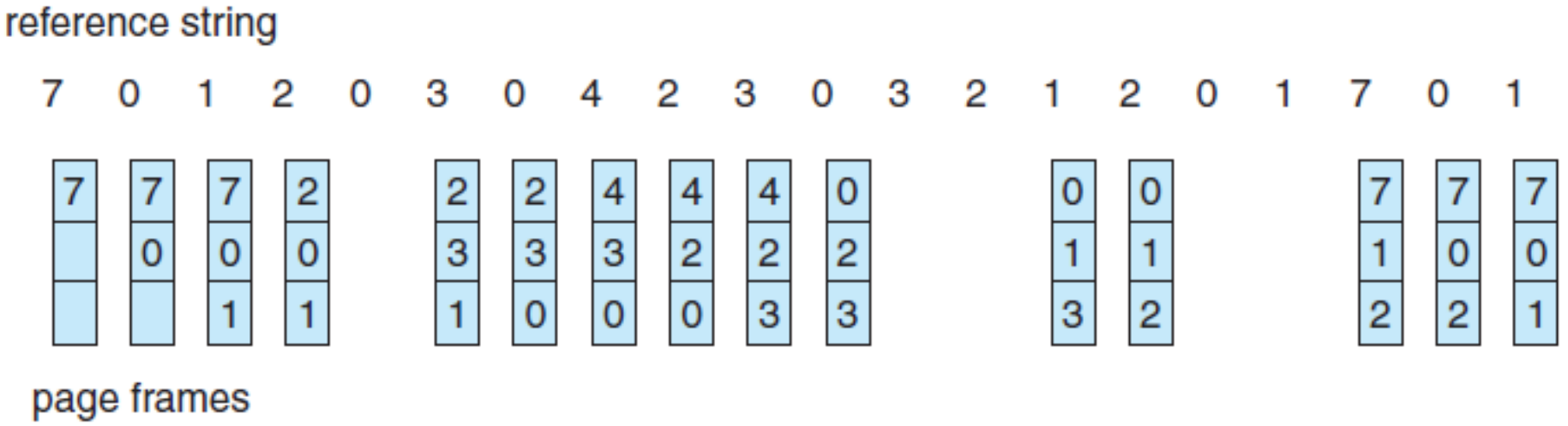


Demand Paging Algorithms

- To implement demand paging we must develop a **frame allocation algorithm** and a **page replacement algorithm**
  - The frame allocation algorithm determines how many frames to give to each process
  - The page replacement algorithm determines which frames to replace
  - Because disk I/O is so expensive, even slight improvements in those algorithms yield large gains in system performance
- We evaluate an algorithm by running it on a particular string of memory references (reference string) and by 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

FIFO Page Replacement

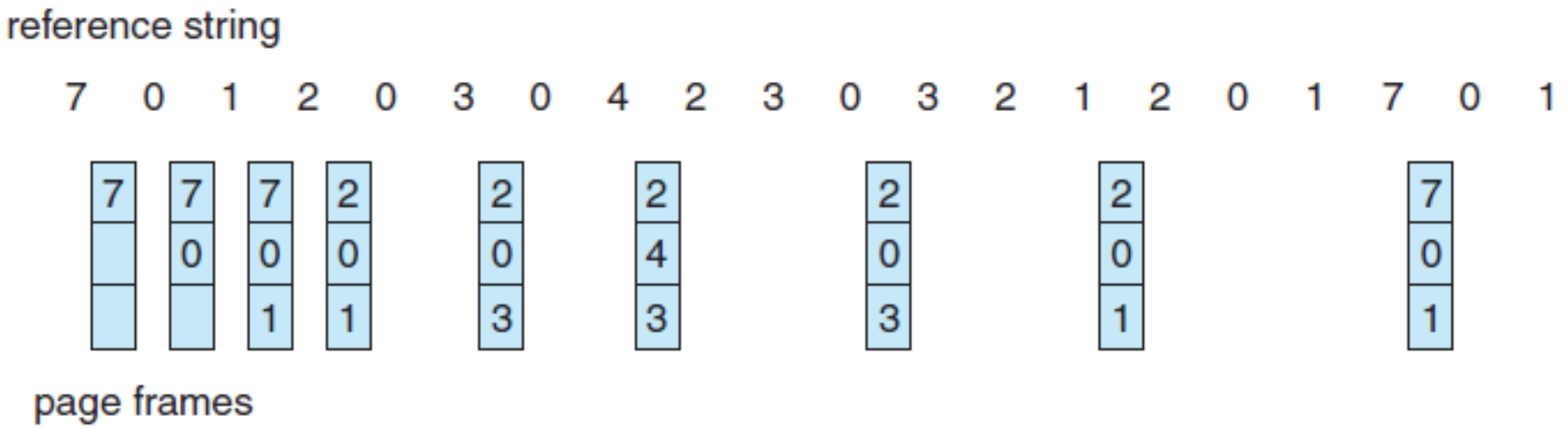
- The simplest page replacement algorithm is the **FIFO (first-in first-out)** algorithm where **the page selected to be replaced is the oldest page**
- A FIFO queue is usually used to track the ages of pages
  - When a page is brought into memory, it is inserted at the tail of the queue
  - The page at the head of the queue is the next to be replaced



OPT Page Replacement

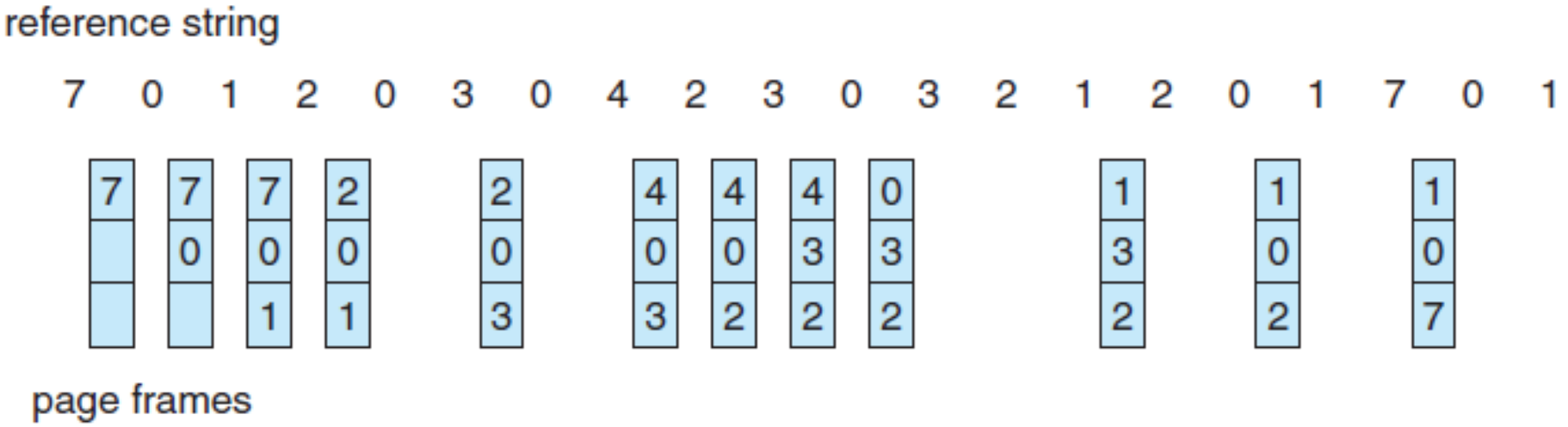
- The **OPT (optimal)** page replacement algorithm has the lowest possible page-fault rate since it **replaces the page that will not be used for the longest period of time**

- How do we know this?
  - Impossible to implement – can't read the future
  - Used mainly for comparison studies with other algorithms



LRU Page Replacement

- The **LRU (least recently used)** algorithm uses the recent past as an approximation of the near future ant it **replaces the page that has not been used for the longest period of time**
  - LRU associates with each page the time of its last use
  - We can think of LRU as the OPT algorithm looking backward in time, rather than forward
  - Generally good algorithm (better than FIFO but worse than OPT)



LRU Page Replacement

- How to implement LRU?
- Counters implementation
  - Add to the CPU a **logical clock** and increment it for every memory reference
  - Associate with each page entry a **time-of-use field** and, whenever a page is referenced, copy the clock register to the time-of-use field of that page
  - To replace a page, **search the entire page table** to find the one with the smallest time value
- Stack implementation
  - Keep a **stack of page numbers in a doubly linked list** with head and tail pointers and, whenever a page is referenced, move it to the top of the stack
  - To replace a page, the **LRU page is always at the bottom**
  - Comparing to counters implementation, each update is more expensive, but there is no search for replacement

LRU Approximation Page Replacement

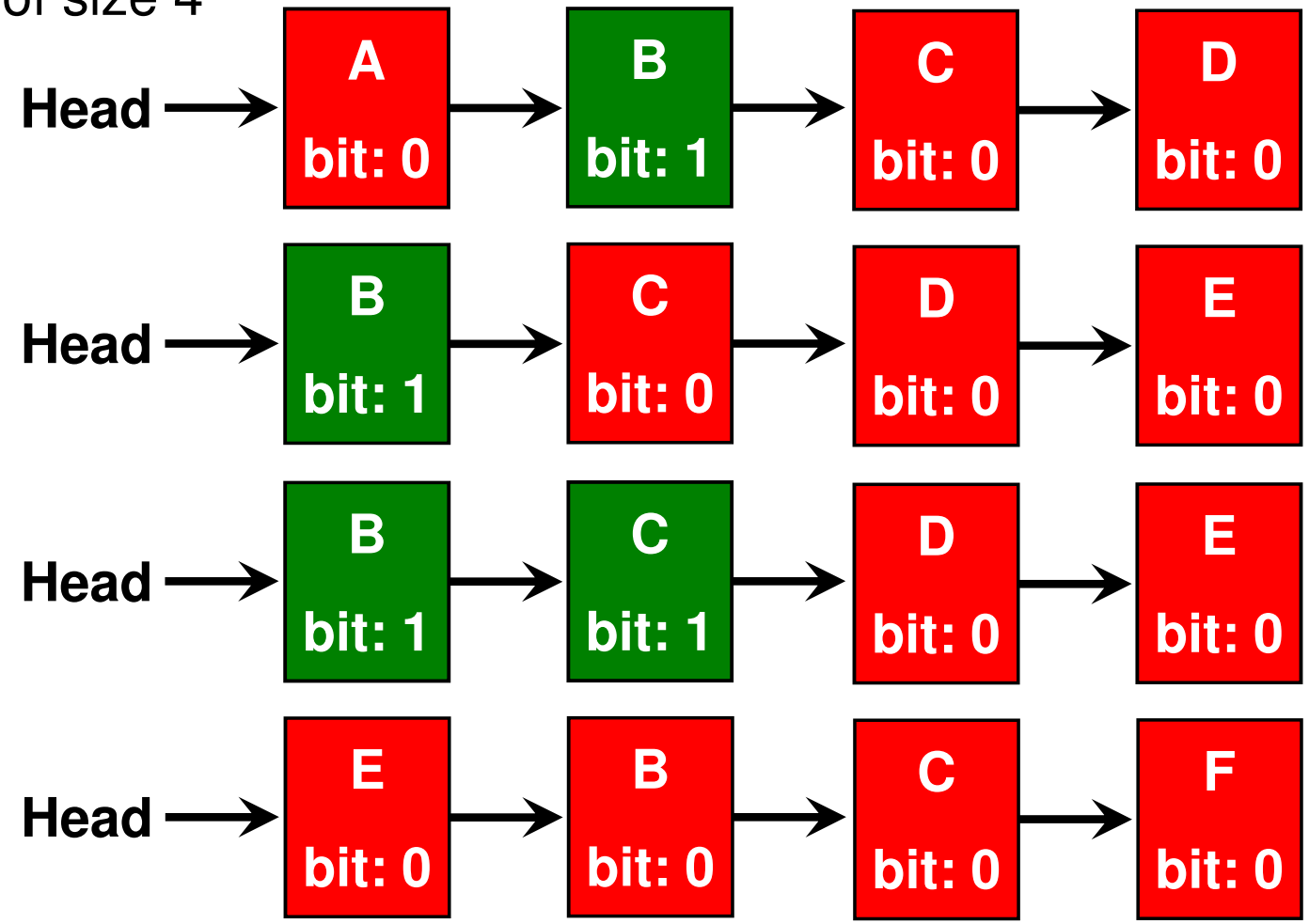
- Note that neither LRU implementation would be conceivable without hardware assistance
  - Updating the clock fields or stack must be done for every memory reference
  - Using an interrupt to allow updating such data structures, would slowdown every memory reference by a factor of at least ten, which is not tolerable
- An approximation is to **associate a reference bit (R) with each page entry**
  - Initially all bits are set to 0
  - When page is referenced (either a read or write to a byte in page) set bit to 1
  - At any point, 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

Second Chance Page Replacement

- Second chance **replaces an old page but not necessarily the oldest**
  - The second chance algorithm works like a **FIFO plus a reference bit**
- When checking the page at head of the queue, also inspect reference bit
  - If bit is 0, proceed to replace page
  - If bit is 1, clear bit and move page to tail (give page a second chance)
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced or given second chances
  - If a page is used often enough to keep its reference bit set, it will never be replaced
- Problem
  - Moving pages to tail is a complex operation

Second Chance Page Replacement

- Consider a page table of size 4
  - Page A arrives
  - Page B arrives
  - Access page B
  - Page C arrives
  - Page D arrives
  - Page E arrives
  - Access page C
  - Page F arrives



Clock Page Replacement

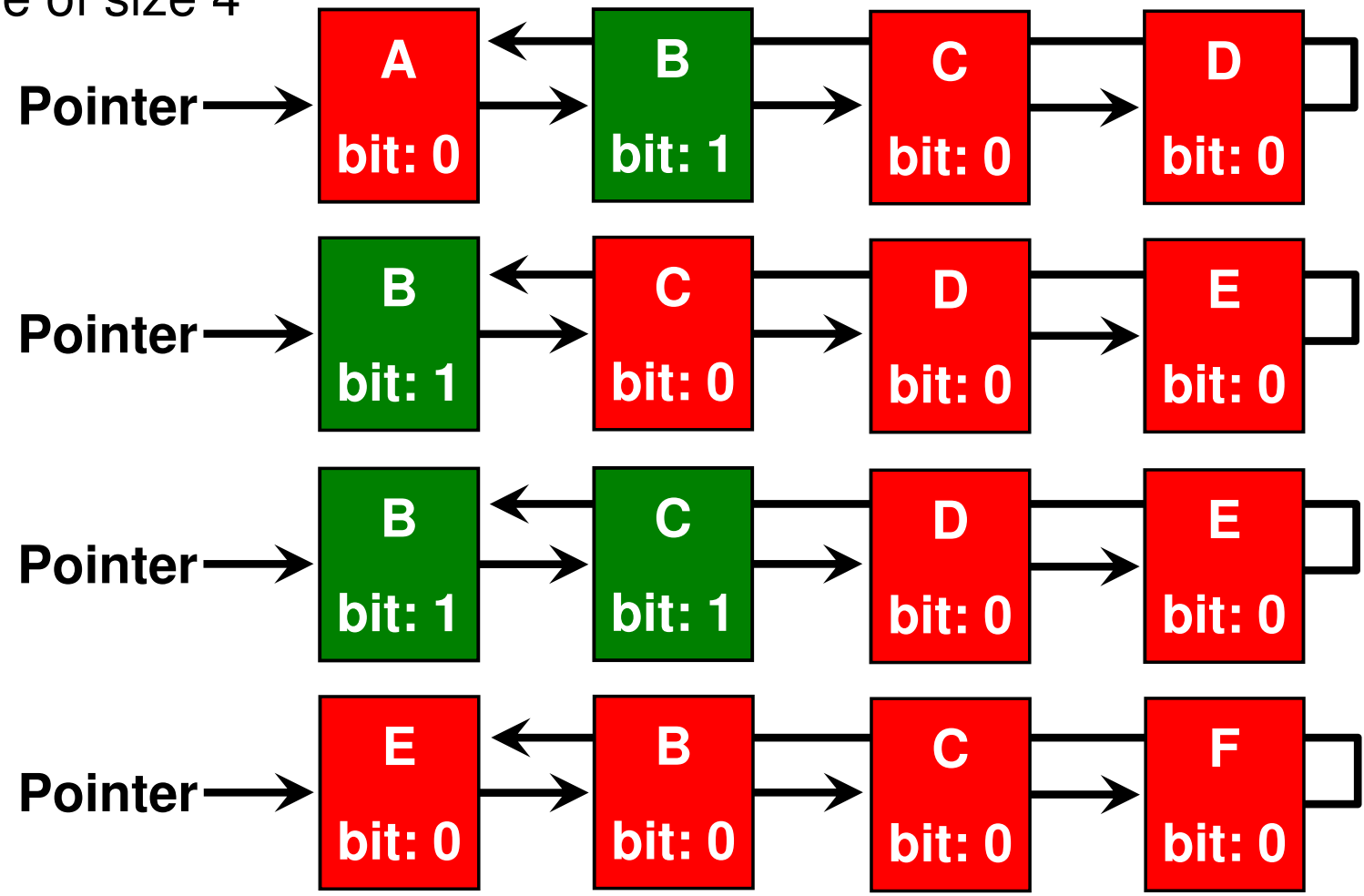
- The clock algorithm is a more efficient implementation of the second chance algorithm, which arranges pages in a **circular queue** and a **pointer indicates which page is to be checked next**

- When a victim page 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 and the new page is inserted in the circular queue in that position

- Will always find a page or loop forever?
  - In the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance and then replaces the initial page

Clock Page Replacement

- Consider a page table of size 4
  - Page A arrives
  - Page B arrives
  - Access page B
  - Page C arrives
  - Page D arrives
  - Page E arrives
  - Access page C
  - Page F arrives



NRU Page Replacement

- The **NRU (not recently used)** algorithm **favors keeping pages in memory that have been recently used**
- It considers a **reference bit (R)** and a **modify bit (M)** (set when the page is modified or written to) and the following **four possible classes (R,M)**:
  - **Class 0 = (0,0)** neither recently used nor modified – best page to replace
  - **Class 1 = (0,1)** not recently used but modified – not quite as good, because the page will need to be written out to disk before replacement
  - **Class 2 = (1,0)** recently used but clean – probably will be used again soon
  - **Class 3 = (1,1)** recently used and modified – probably will be used again soon, and the page will be need to be written out to disk before replacement
- The setting of the bits is usually done by the hardware

NRU Page Replacement

- How to implement NRU?
- Can use the same scheme as in the clock algorithm and **replace the first page encountered in the lowest nonempty class**
  - As it advances, it clears the reference bits
  - May have to scan the circular queue more than once before reaching a page to be replaced
- Additionally, **at a certain fixed time interval**, use the clock interrupt to **clear the reference bit of all pages**
  - Only pages referenced within the current clock interval are marked with a referenced bit

LFU/NFU Page Replacement

- The **LFU (least frequently used)** or **NFU (not frequently used)** algorithm uses counters to keep track of how frequently a page has been used and it **replaces the page with the lowest counter**
  - Requires a counter per page, initially set to 0
  - At each clock interval, each reference bit is added to the corresponding page counter
- A problem arises 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
  - One solution is to use **aging**

Aging Page Replacement

- The aging algorithm is a descendant of the LFU/NFU algorithm with **modifications to make it aware of the time span of use**
- Instead of just incrementing the counters, **each counter is first shifted right** (divided by 2) before adding the referenced bit to the left of that binary number
  - The idea is to ensure that pages referenced more recently, though less frequently referenced, will have higher priority over pages more frequently referenced in the past
- For instance, if a page has referenced bits 1,0,0,1,1,0 in the past 6 clock ticks, its referenced counter will look like this:
  - 10000000, 01000000, 00100000, 10010000, 11001000, 01100100

Recap: FIFO

- Frames: F1, F2 and F3
- Pages: A, B, C and D

Frame	A	B	C	A	B	D	A	D	B	C	B
F1	A					D				C	
F2		B					A				
F3			C						B		

7 page faults

Recap: LRU

- Frames: F1, F2 and F3
- Pages: A, B, C and D

Frame	A	B	C	A	B	D	A	D	B	C	B
F1	A			X			X			C	
F2		B			X				X		X
F3			C			D		X			

5 page faults

Recap: Second Chance

- Frames: F1, F2 and F3
- Pages: A, B, C and D

Frame	A	B	C	A	B	D	A	D	B	C	B
F1	A			1		0	1			C	
F2		B			1	0			1	0	1
F3			C			D		1		0	

5 page faults

Recap: Aging

- Frames: F1, F2 and F3
- Pages: A, B, C and D (4 bits counter)

Frame	A	B	C	A	B	D	A	D	B	C	B
F1	A	4	2	9	4	2	9	4	2	C	4
F2		B	4	2	9	4	2	1	8	4	10
F3			C	4	2	D	4	10	5	2	1

5 page faults

Allocation of Frames

- How to allocate the fixed amount of physical memory among processes?
  - Cannot allocate more than the total number of available frames
  - Must allocate a minimum number of frames that keeps a fair page-fault rate
- **Equal allocation** – allocate an equal share to everyone
  - If there are 100 frames and 2 processes, give each process 50 frames
- **Proportional allocation** – allocate proportionally to processes' size
  - If there are 100 frames and 2 processes, one of 20 pages and one of 180 pages, give each 10 (20/200*100) and 90 (180/200*100) frames, respectively
- **Priority allocation** – use a proportional allocation scheme using priorities rather than size or a combination of priority and size
  - On a page-fault, select one of its frames or a frame from a lower priority process

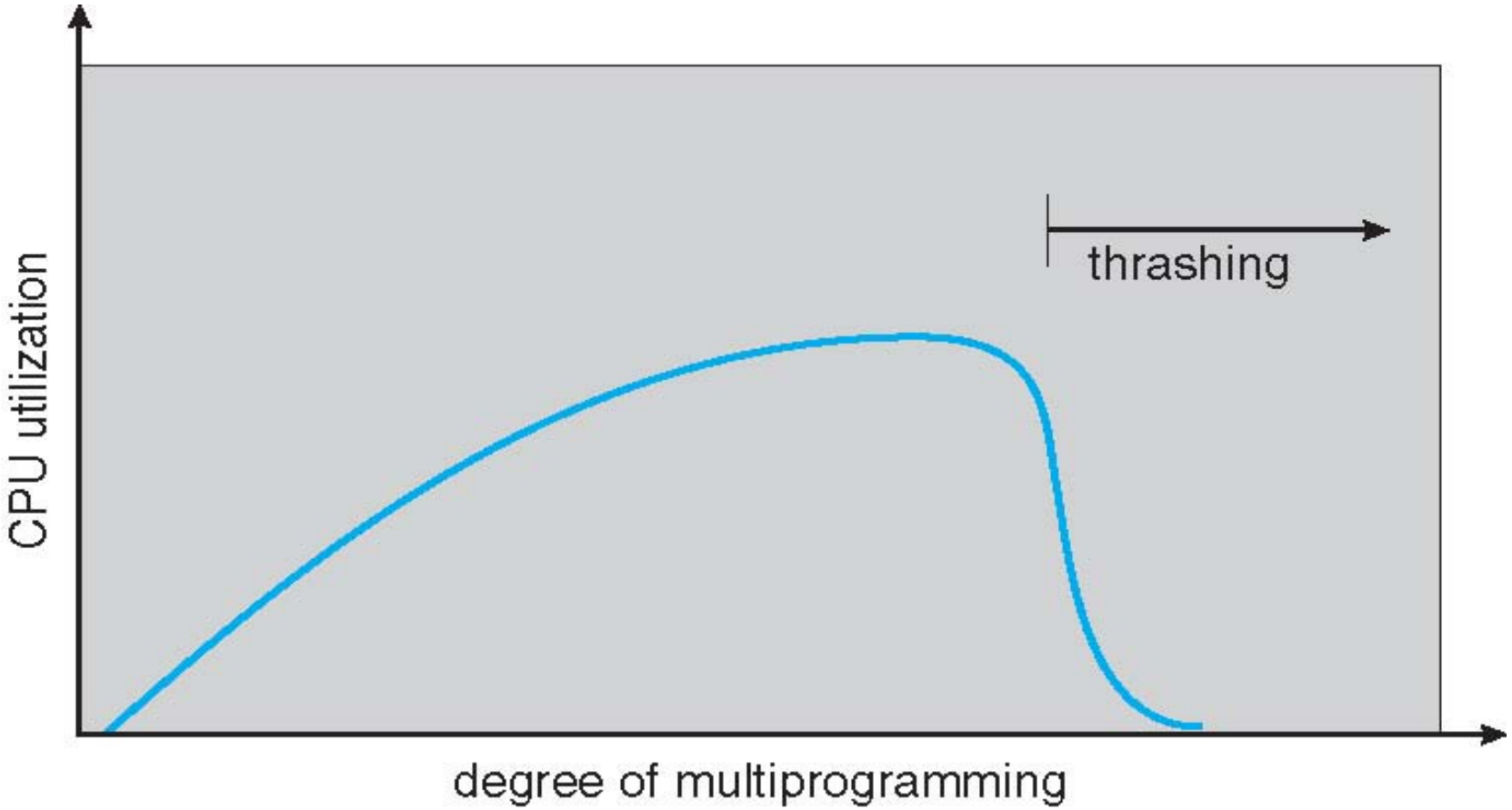
Allocation of Frames

- **Global replacement** – allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process, i.e., one process can take a frame from another
  - Process execution time can vary greatly since its paging behavior depends on the other processes running simultaneously
  - Generally results in greater system throughput and is therefore more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - Might underutilized memory, by not making less used pages available to others

Thrashing

- Thrashing occurs when a **process is busy swapping pages in and out** and **spends more time paging than executing**
- If a process does not have enough frames, its page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - Since all its pages are in active use, quickly must need replaced page back
  - Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately
- This leads to low CPU utilization
  - Operating system spends most of its time swapping to disk

Thrashing

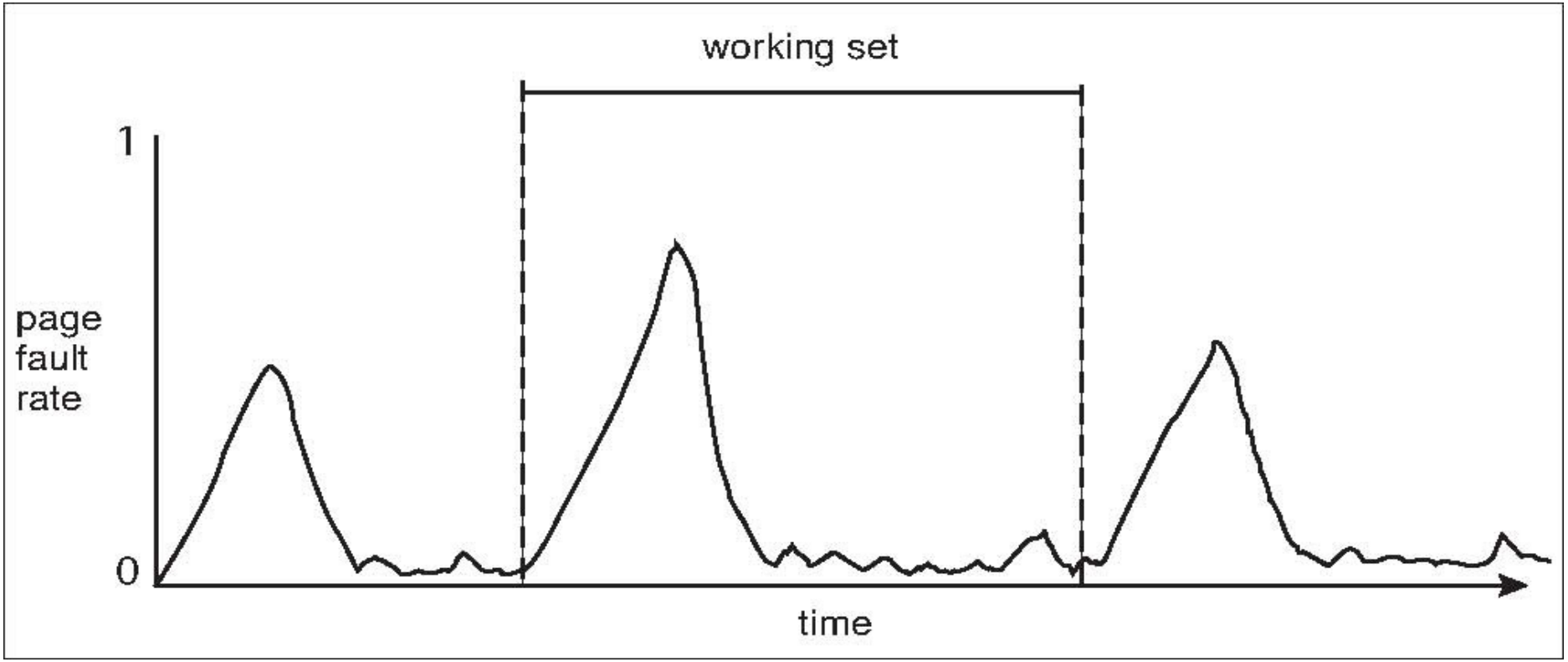


Working Set Model

- Program memory access patterns have **temporal and spatial locality** and, as a process executes, it **moves from locality to locality**
  - For example, when a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality. We may return to this locality later.
- The set of pages that are accessed together along a given time slice (or locality) is called the **working set**
  - The working set defines the **minimum number of pages needed for a process to behave well**
  - If we do not have enough frames for the current locality, the process will thrash

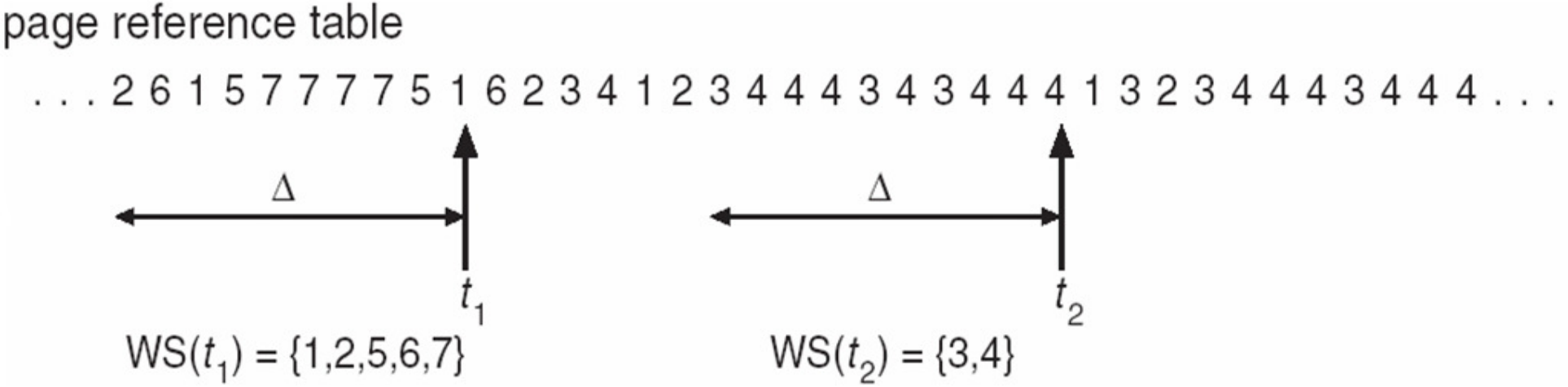
Working Set Model

- Initially, the process will fault for the pages in its locality until all these pages are in memory but then, it will not fault again until it changes locality



Working Set Model

- The working set model uses a **parameter Δ to define the working set window**, i.e., the set of pages in the current window Δ is the working set
  - If a page is in active use, it will be in the working set
  - If a page is no longer being used, it will drop from the working set
  - The working set is thus an approximation of the program's locality



Working Set Model

- If we know the working set size for each process Pi (WSi), we can then compute the total demand D for frames:

$$D = \Sigma WSi$$

- If the total demand is greater than the total number of available frames, some processes will not have enough frames and thrashing will occur

- A possible solution is to select a process to suspend, swap out its pages and reallocated its frames to other processes (the suspended process can be restarted later)
  - This strategy prevents thrashing while keeping the degree of multiprogramming as high as possible

Page-Fault Frequency Model

- The difficulty with the working set model is defining the working set window Δ
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities or the entire program
- A more direct approach establishes **upper and lower bounds on the acceptable page-fault frequency rate**
  - If the actual page-fault rate is too high, the process gains more frames
  - If the actual page-fault rate is too low, the process may lose frames
- If no free frames are available, we must select a process to suspend, swap out its pages and distributed its frames to processes with high page-fault rates

Page-Fault Frequency Model