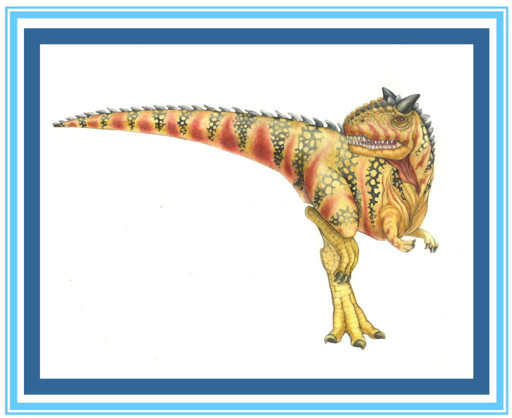
Chapter 10: Virtual Memory



Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

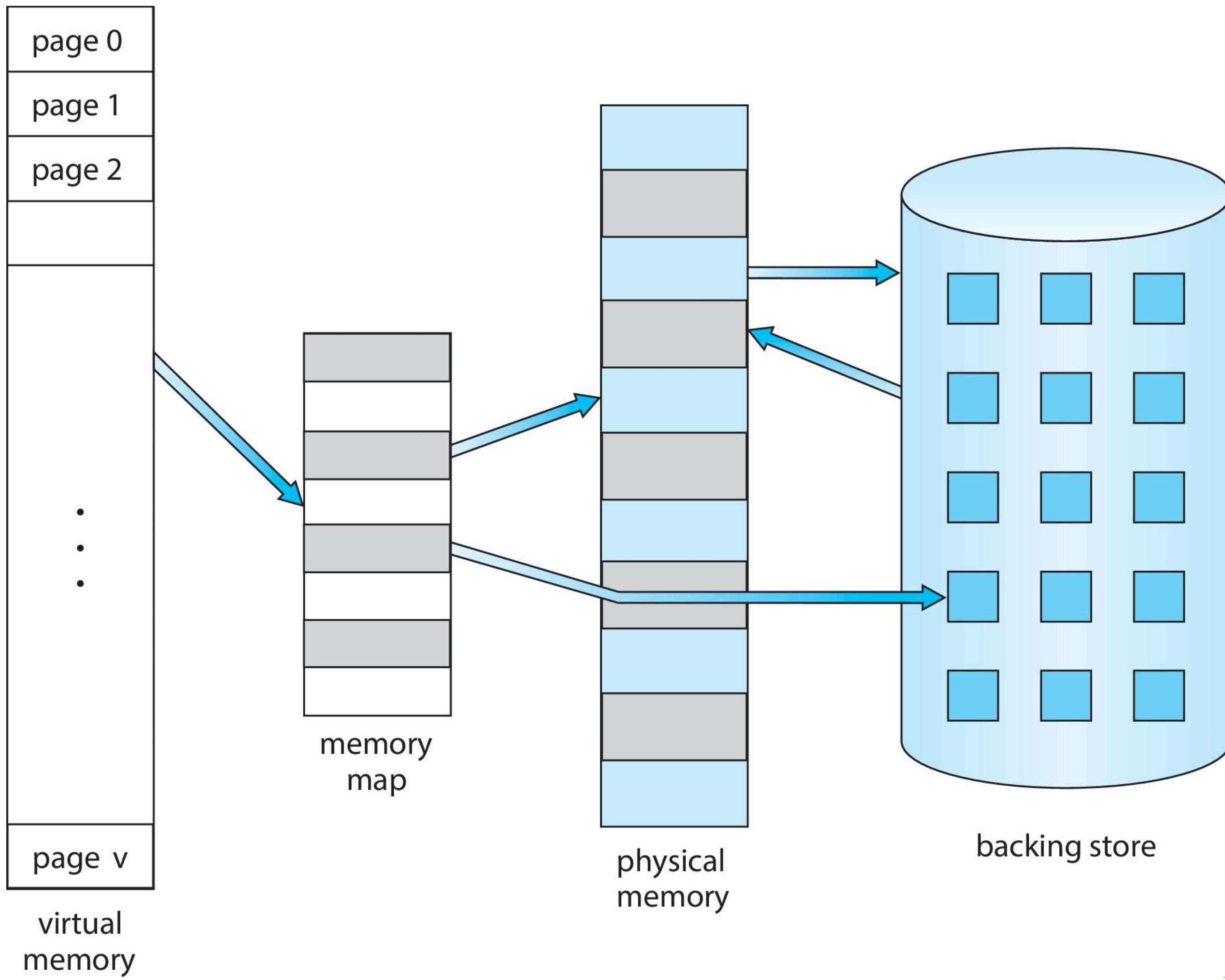
Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Even in those cases where the entire program is needed, it may not all be needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program and programs could be larger than physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time

Virtual memory

- **Virtual memory** – separation of user logical memory (virtual address space) from 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
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

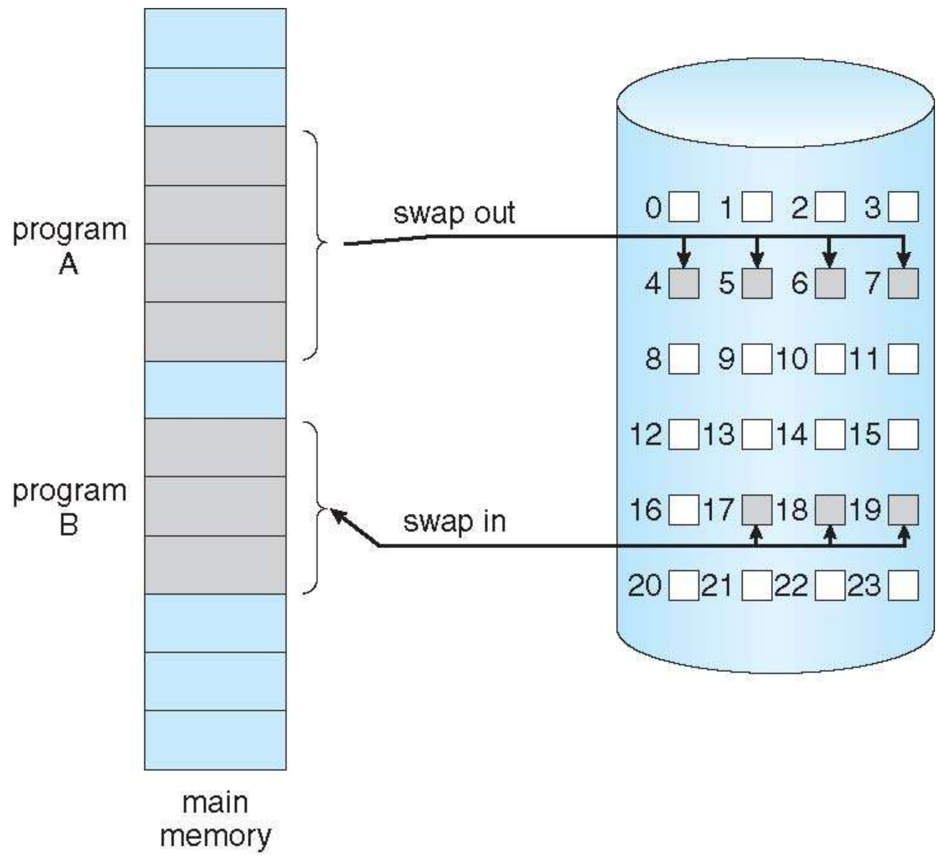


Demand Paging

- Pages are only loaded 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
  - Rather than swapping the entire process into memory, however, we use a **lazy swapper**
  - Lazy swapper: never swaps a page into memory unless that page will be needed
    - Swapper that deals with pages is a **pager**

Demand Paging

- Similar to paging system with swapping (diagram on right)



Basic Concepts

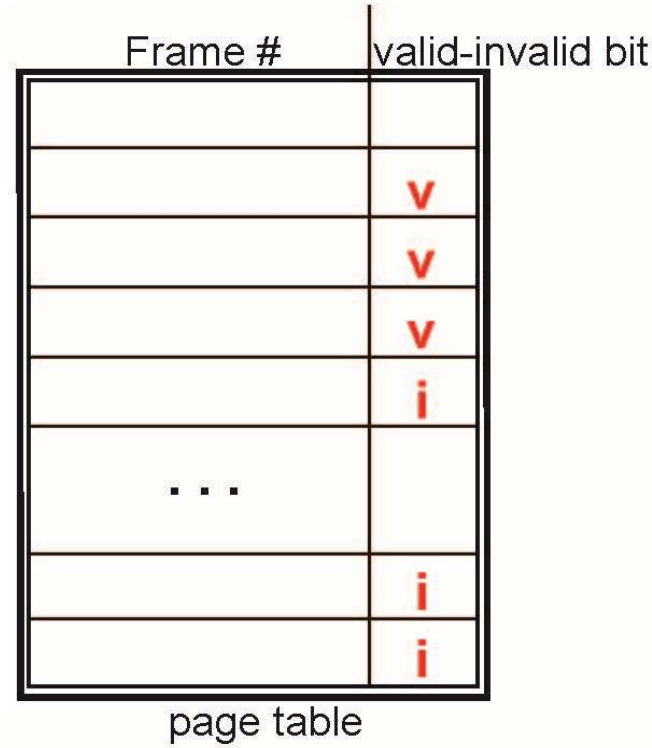
- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - **v** ⇒ in-memory – **memory resident**
  - **i** ⇒ not-in-memory
- Initially valid–invalid bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault trap

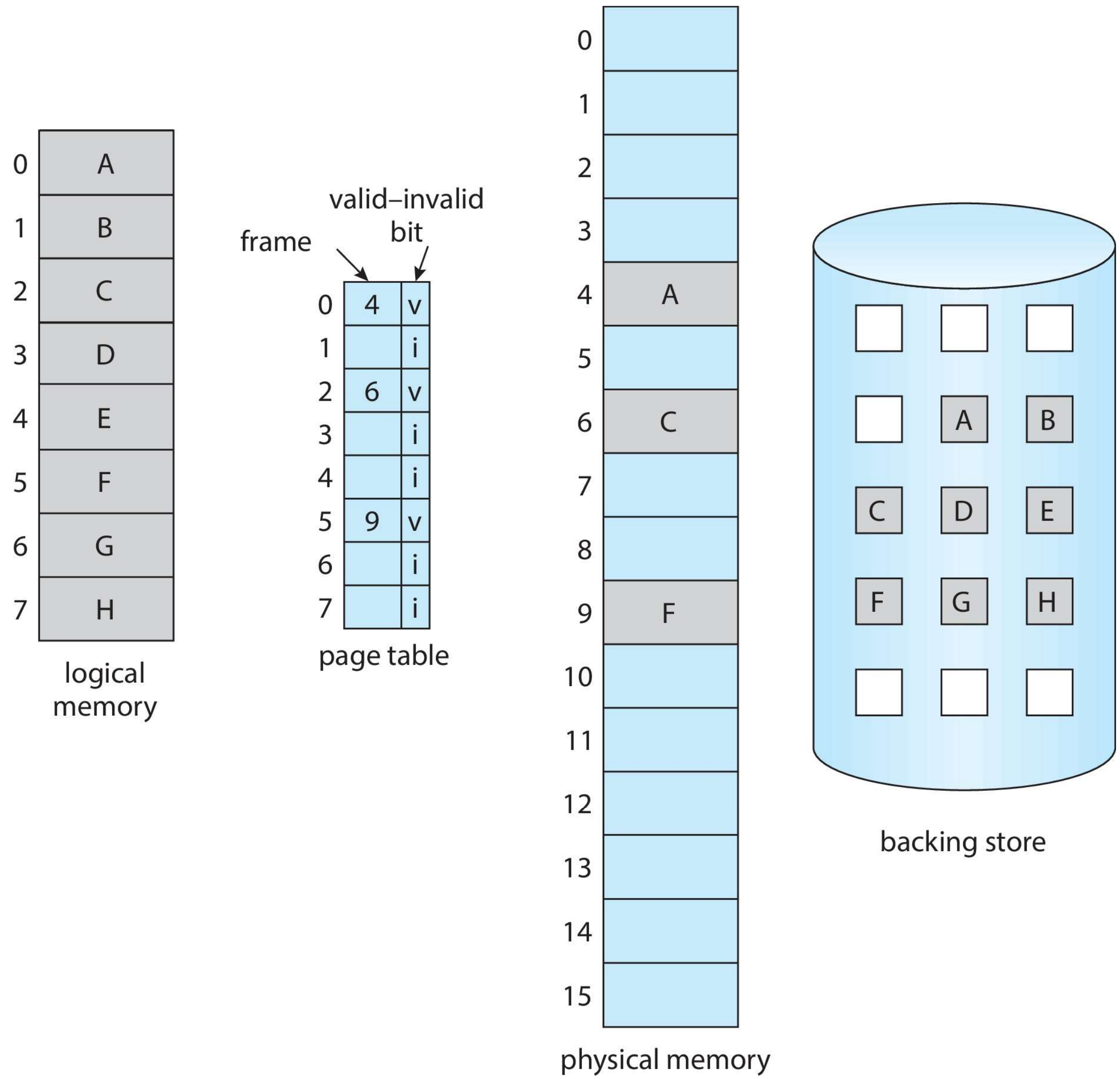
Valid-Invalid Bit

- Example of a page table snapshot:

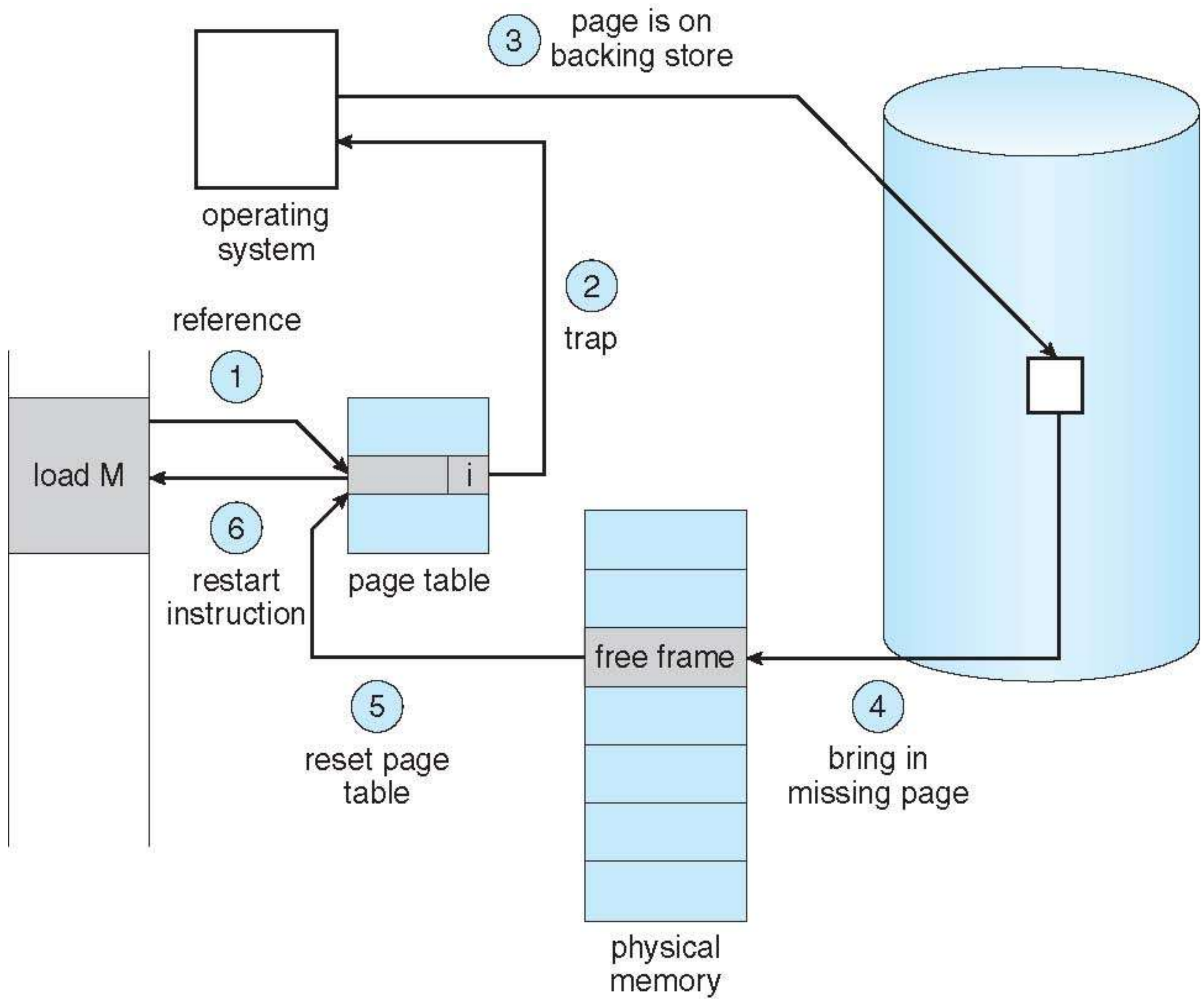


- During MMU address translation, if valid–invalid bit in page table entry is **i** $\Rightarrow$ page fault

Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault (Cont.)



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
3. Find free frame
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

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 (Cont.)

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

Pure Demand Paging

- Start process with ***no*** pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident $\Rightarrow$ page fault occurs
  - And for every other process pages, page fault occurs on first access
  - Never bring a page into memory until it is required
- Theoretically, a given instruction could access multiple pages $\Rightarrow$ multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1$
- Effective Access Time (EAT)

  EAT = $(1 - p) \times$ memory access + $p \times$ page fault overhead

- page-fault overhead
  1. Service the page-fault interrupt
  2. Read in the page (read the page from disk) – lots of time
  3. Restart the process
- Example
  - Memory access time = 200 nanoseconds
  - Average page fault service time = 8 milliseconds

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = $(1 – p) \times 200 + p$ (8 milliseconds)

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

  $= 200 + p \times 7{,}999{,}800$
- If one access out of 1,000 causes a page fault, then

  $P = 1/1000$

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!! (the access time gets 1/40!!)
- If want performance degradation < 10 percent, then EAT = 200 + 20 220
  - $220 > 200 + 7{,}999{,}800 \times p$
    $20 > 7{,}999{,}800 \times p$
  - $p < .0000025$ => (25 page faults in any 10,000,000 memory accesses)
  - < one page fault in every 400,000 memory accesses

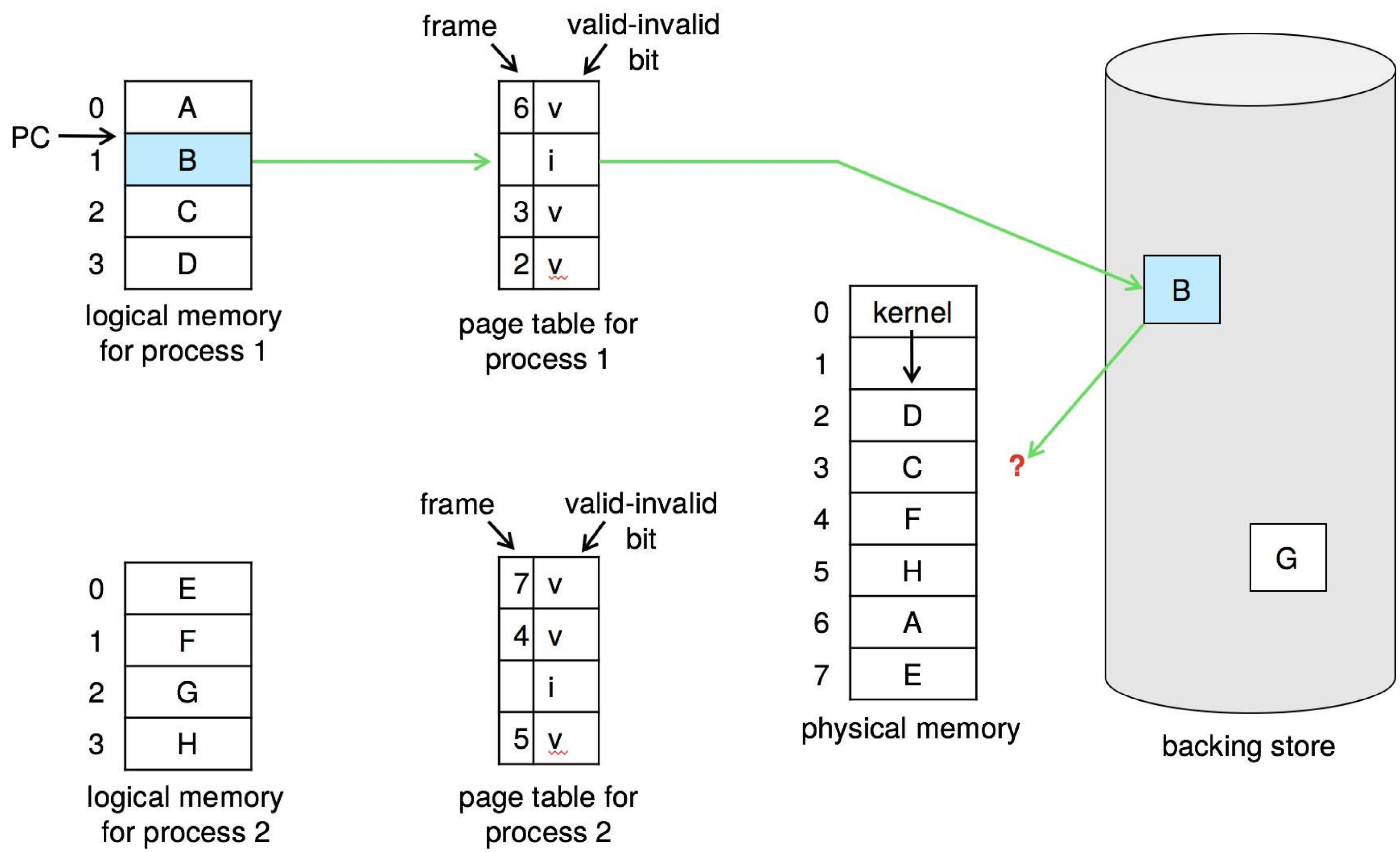
Copy-on-Write

- Recall that the fork() system call creates a child process as a duplicate of its parent
  - It creates a copy of the parent's address space for the child
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied

Page Replacement

- When virtual memory management over-allocates memory, it is possible that all available memory is used by active processes
  - In this situation, if a page fault occurs, there is no free frame to allocate it to the requested page
- Solution
  - Find some page in memory that is not currently being used and page it out
  - We can free a frame by writing its contents to swap space and changing the page table
  - Same page may be brought into memory several times

Need For Page Replacement



Page Replacement

- Page replacement increases the effective access time
- Use **modify (dirty) bit** to reduce overhead of page transfers
  - Each page has a modify bit associated with it
  - The modify bit for a page is set by the hardware whenever any word or byte in the page is written
  - Only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory
  - large virtual memory can be provided on a smaller physical memory

Demand Paging Algorithms

- Frame-allocation algorithm
  - Determines how many frames to allocate to each process
- Page-replacement algorithm
  - Selects the frames that are to be replaced
- Designing efficient algorithms is so important, because disk I/O is so expensive
  - In general, we want the algorithm with the lowest page-fault rate

Page Replacement Algorithms

- We 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
- In all our examples, the reference string is

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

and there are 3 frames

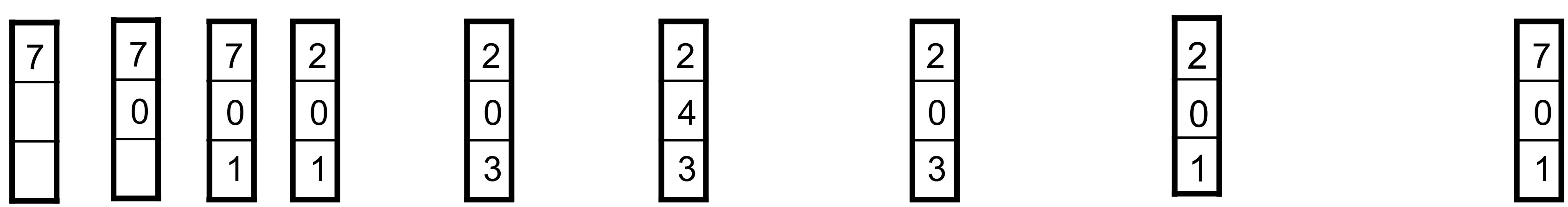
Page Replacement Algorithms

- Optimal
- FIFO
- LRU (Least Recently Used)
- LRU Approximation
  - **Additional-Reference-Bits Algorithm**
  - Second-Chance Algorithm
- Counting-Based Page Replacement
  - LFU (Least Frequently Used)
  - MFU (Most Frequently Used)

Optimal Page Replacement

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

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

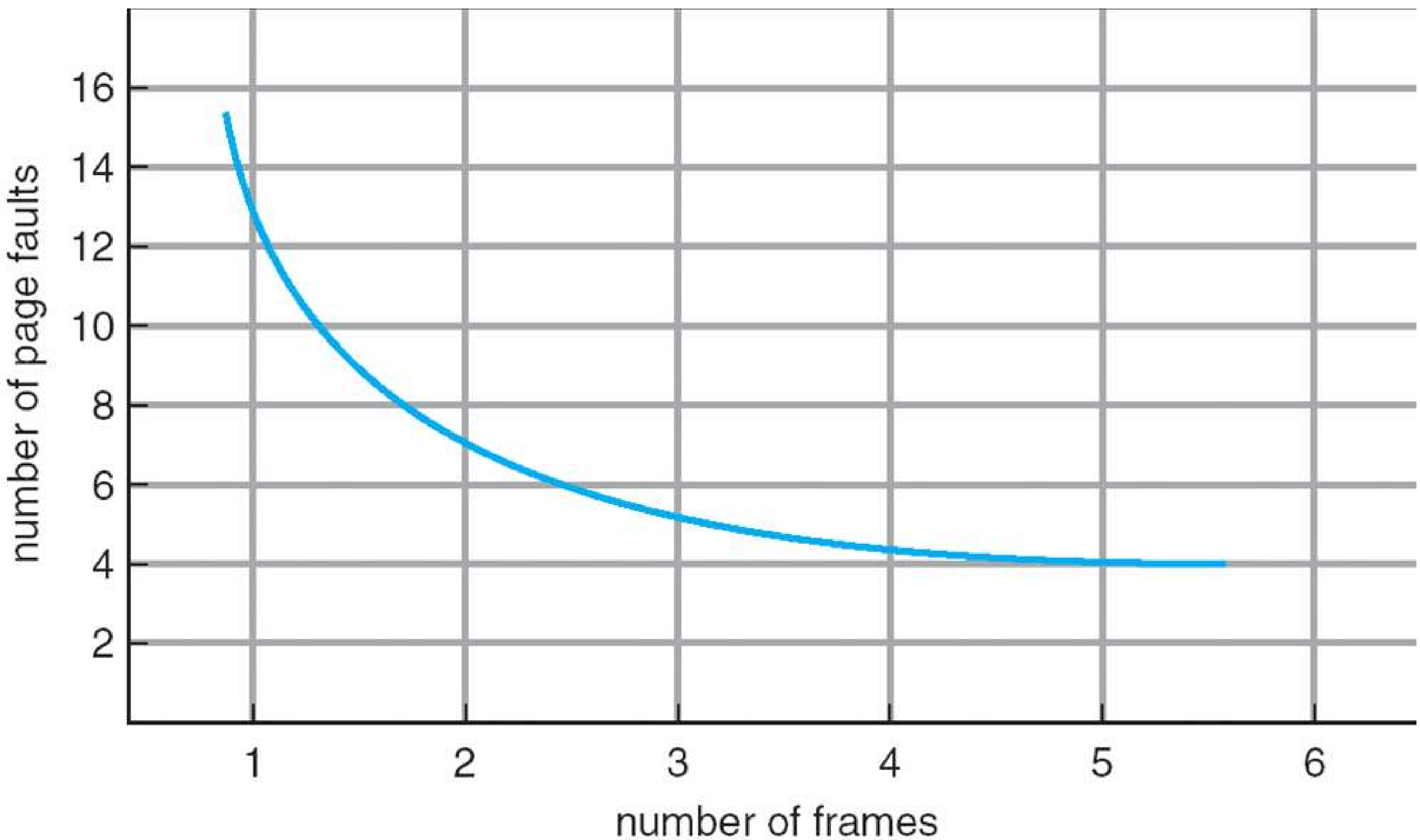


First-In-First-Out (FIFO) Algorithm

- When a page must be replaced, the oldest page is chosen
  - Can be implemented using a FIFO queue
- Example: 15 page faults

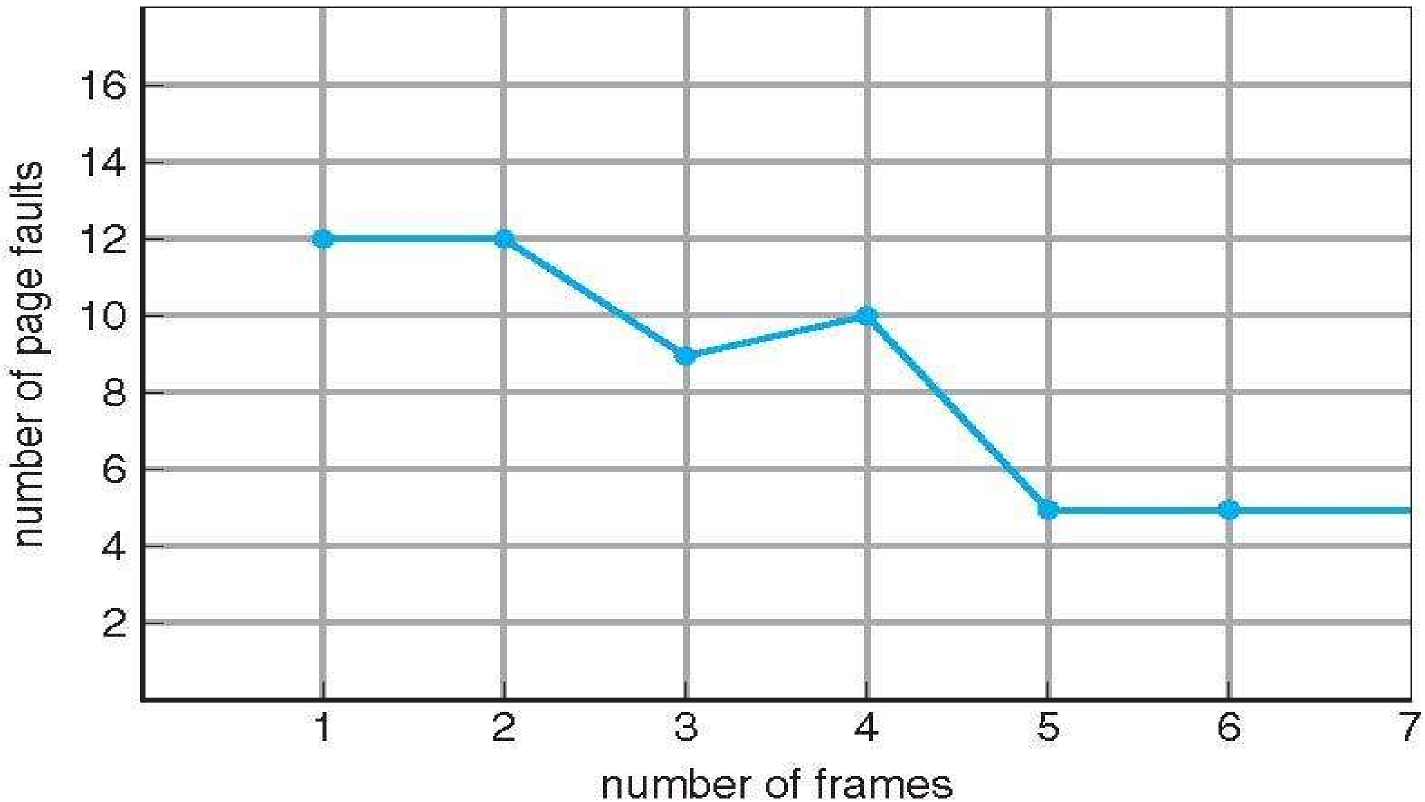
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	0		1	2	3	0	4	2			3	0			1	2	7
	0	0	1		2	3	0	4	2	3			0	1			2	7	0
		1	2		3	0	4	2	3	0			1	2			7	0	1

Graph of Page Faults Versus the Number of Frames



Belady's Anomaly

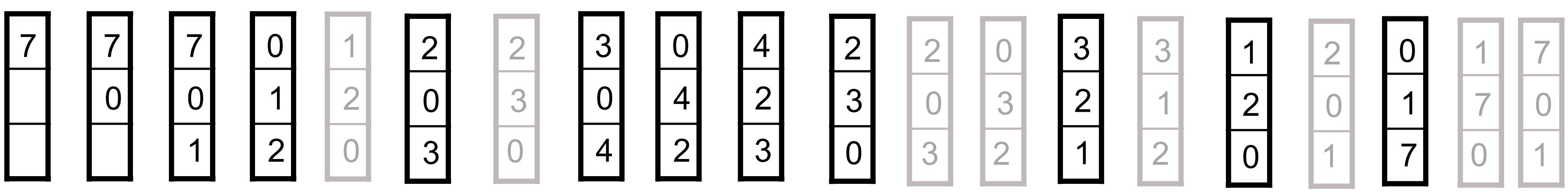
- For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases
- Example: reference string = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Least Recently Used (LRU)

- LRU is an approximation of the optimal algorithm
  - It uses the recent past as an approximation of the near future
- LRU replaces the page that has not been used for the longest period of time

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



Least Recently Used (LRU)

- Stack implementation
  - Keep a stack of page numbers in a double link form
  - Page referenced: move it to the top
  - But each update more expensive
  - No search for replacement
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed

Least Recently Used (LRU)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
  - An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames
- Implementation of LRU needs hardware assistance
  - The updating of the clock fields or stack must be done for every memory reference

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- Few computer systems provide sufficient hardware support for true LRU
- Many systems provide some help, in the form of a reference bit
  - The reference bit for a page is set by the hardware whenever that page is referenced
  - Reference bits are associated with each entry in the page table

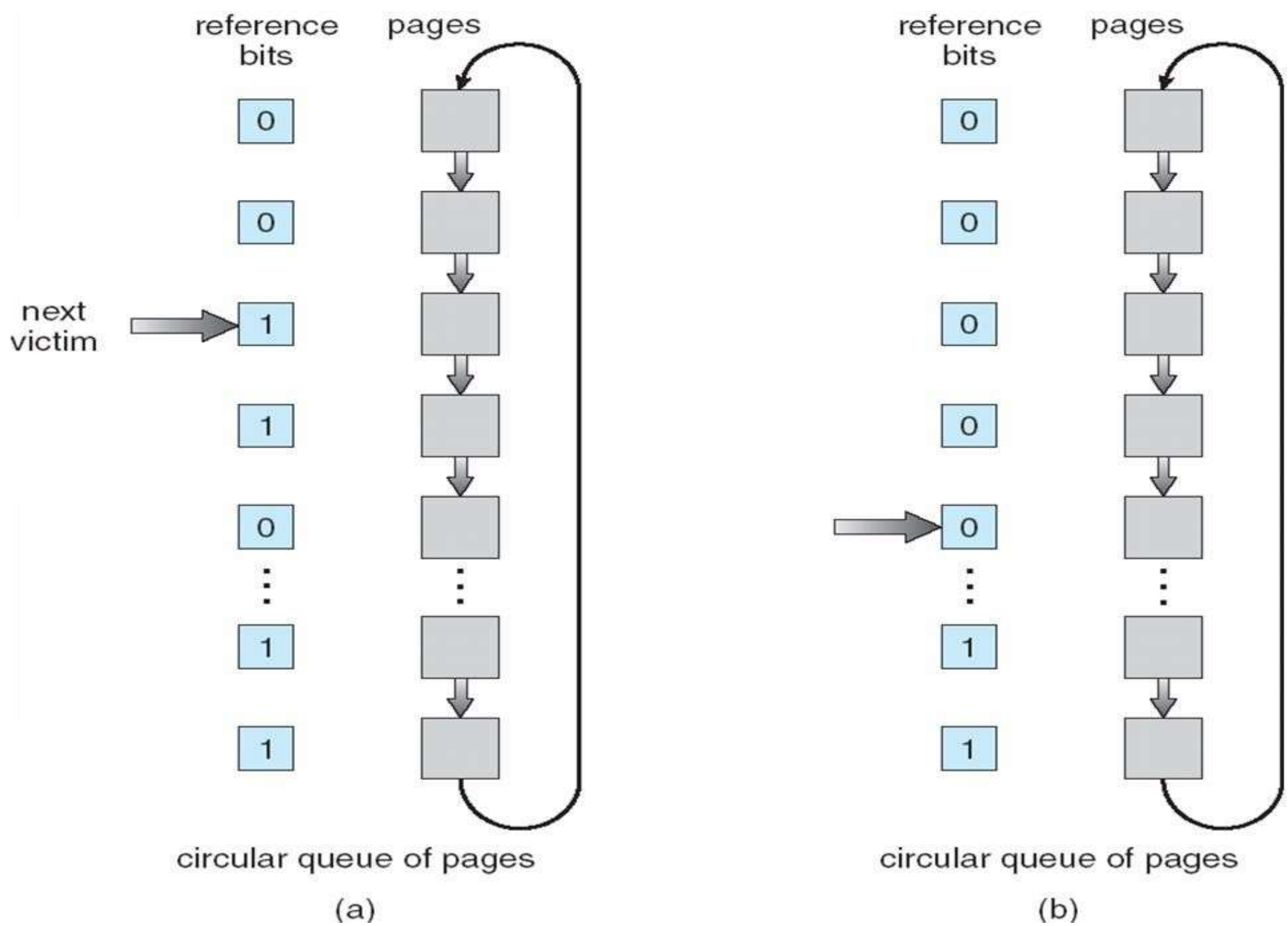
Additional-Reference-Bits Algorithm

- Keep an 8-bit byte for each page
- At regular intervals (say, 100 ms), 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
- These 8-bit shift registers contain the history of page use for the last eight time periods
  - Example: 00000000, 11111111, 11000100, 01110111
- The page with the lowest number is the LRU page, and it can be replaced.

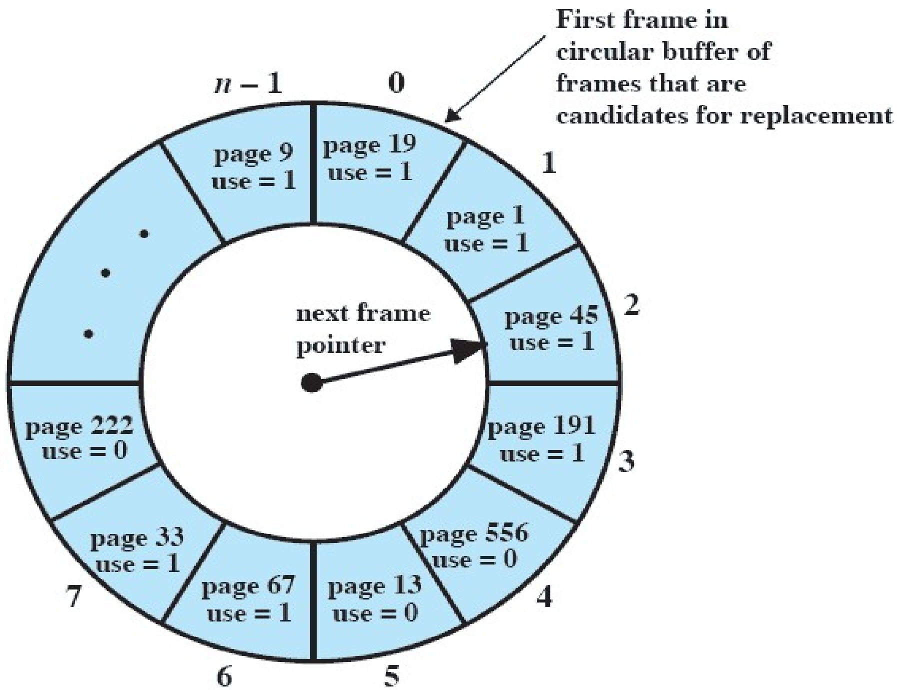
Second Chance Algorithm

- The basic algorithm is FIFO
- When a page has been selected, we inspect its reference bit
  - If the value is 0, we proceed to replace this page
  - If the value is 1, we give the page a second chance and move on to select the next FIFO page, its reference bit is cleared, and its arrival time is reset to the current
- Clock algorithm
  - An implementation of the second-chance algorithm using a circular queue

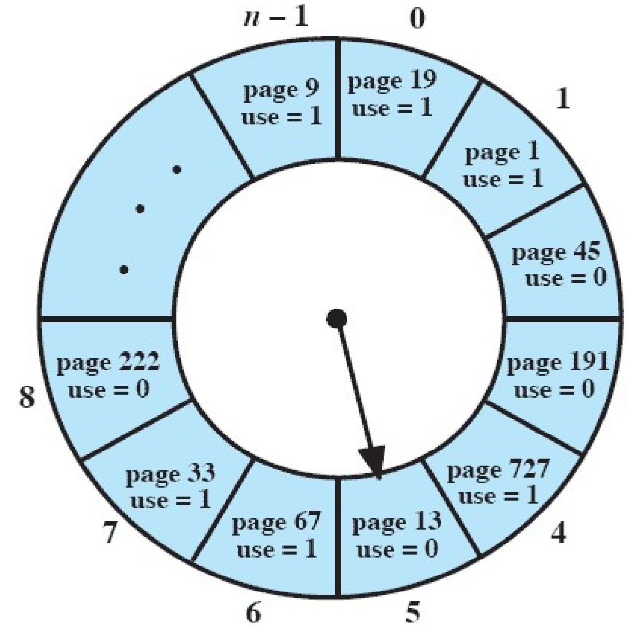
Second Chance Algorithm



Second Chance Algorithm



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

Enhanced Second-Chance Algorithm

- We can use reference bit and modify bit together
- Then we have four cases
  - (0, 0) neither recently used nor modified - best page to replace
  - (0, 1) not recently used but modified- not quite as good, need to be written out
  - (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 need to be written out
- We replace the first page encountered in the lowest nonempty class

Enhanced Second-Chance Algorithm

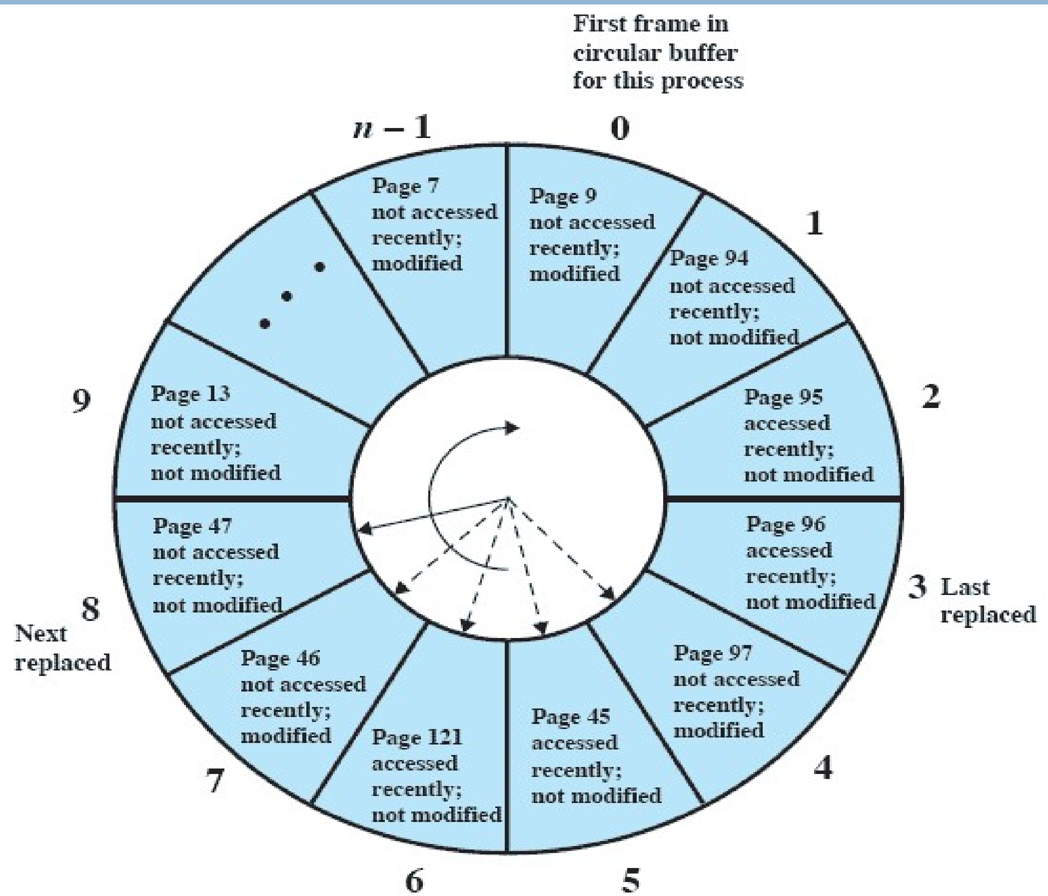
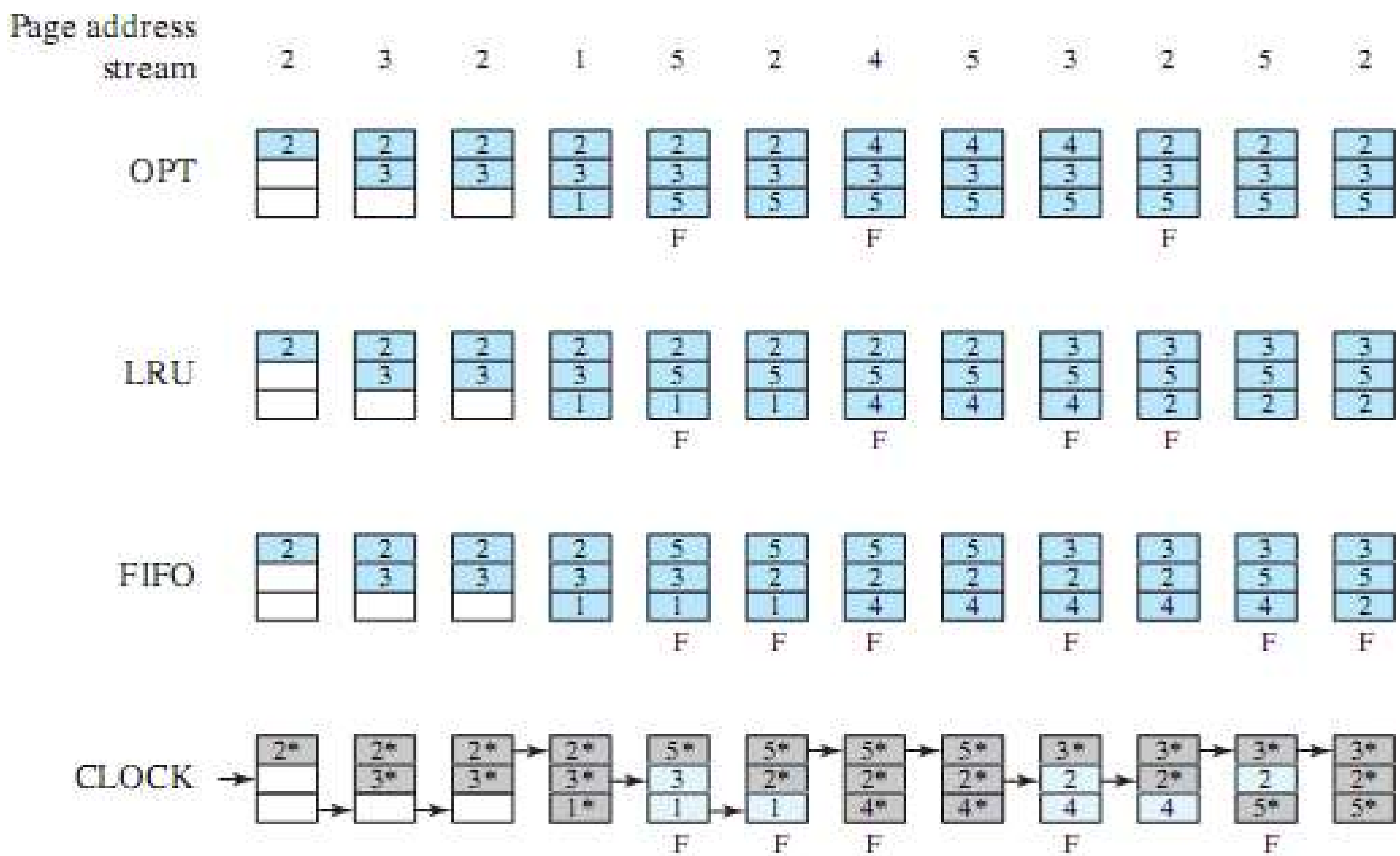


Figure 8.18 The Clock Page-Replacement Algorithm [GOLD89]

Combined Examples



F= page fault occurring after the frame allocation is initially filled

Figure 8.15 **Behavior of Four Page Replacement Algorithms**

Comparison

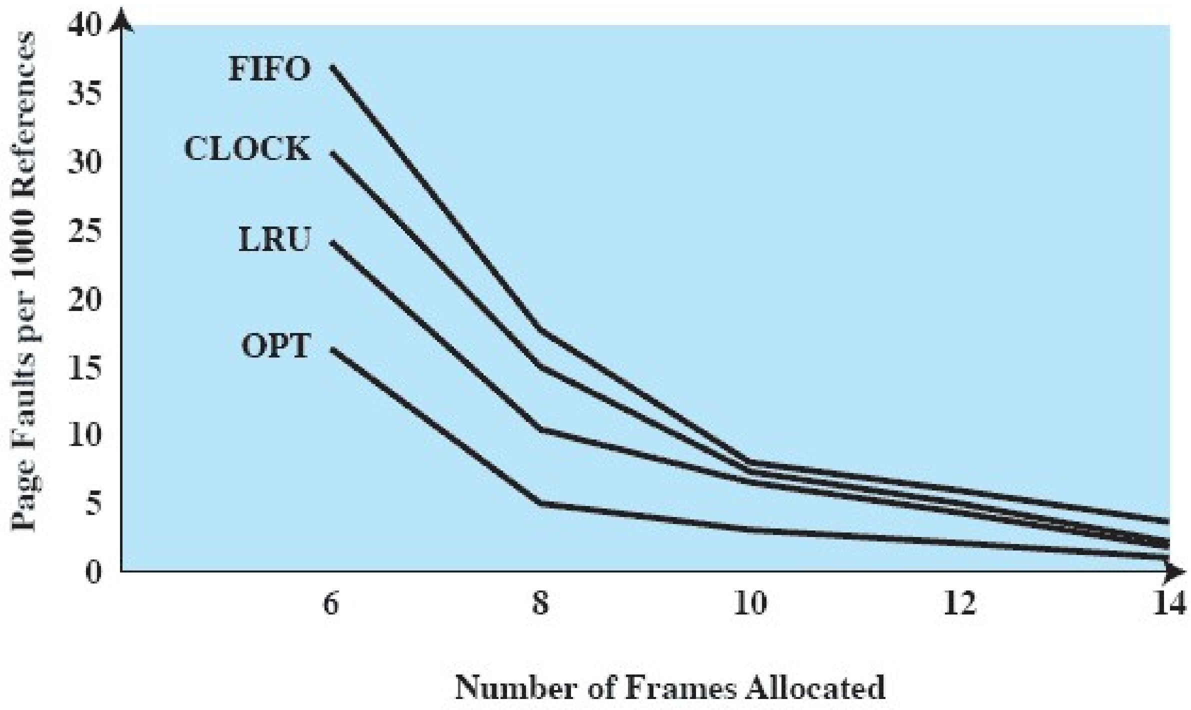


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- Least Frequently Used (LFU)
  - Requires that the page with the smallest count be replaced
  - Problem: when a page is used heavily during the initial phase of a process but then is never used again
  - Solution: shift the counts right by 1 bit at regular intervals

Counting Algorithms

- Most Frequently Used (MFU)
  - is 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
  - They do not approximate OPT replacement well

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - When a page fault occurs, a victim frame is chosen as before
  - However, the desired page is read into a free frame from the pool before the victim is written out
  - When the victim is later written, its frame is added to the free-frame pool
- Keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty

Resident Set Management

- The OS must decide how many pages to bring into main memory
  - The smaller the amount of memory allocated to each process, the more processes that can reside in memory.
  - Small number of pages loaded increases page faults.
  - Beyond a certain size, further allocations of pages will not affect the page fault rate.

Resident Set Size

- Fixed-allocation
  - Gives a process a fixed number of pages within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
  - Number of pages allocated to a process varies over the lifetime of the process

Fixed Allocation

- Equal allocation
  - If there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
- Proportional allocation
  - Allocate according to the size of process

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Replacement Scope

- The scope of a replacement strategy can be categorized as *global* or *local*.
  - Both types are activated by a page fault when there are no free page frames.
  - A local replacement policy chooses only among the resident pages of the process that generated the page fault
  - A global replacement policy considers all unlocked pages in main memory

Fixed Allocation, Local Scope

- Decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate
- If allocation is too large there will be too few programs in main memory
  - Increased processor idle time or
  - Increased swapping

Variable Allocation, Global Scope

- Easiest to implement
  - Adopted by many operating systems
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces one from another process
  - Therein lies the difficulty … which to replace.

Variable Allocation, Local Scope

- When new process added, allocate number of page frames based on application type, program request, or other criteria
- When page fault occurs, select page from among the resident set of the process that suffers the fault
- Reevaluate allocation from time to time

Resident Set Management Summary

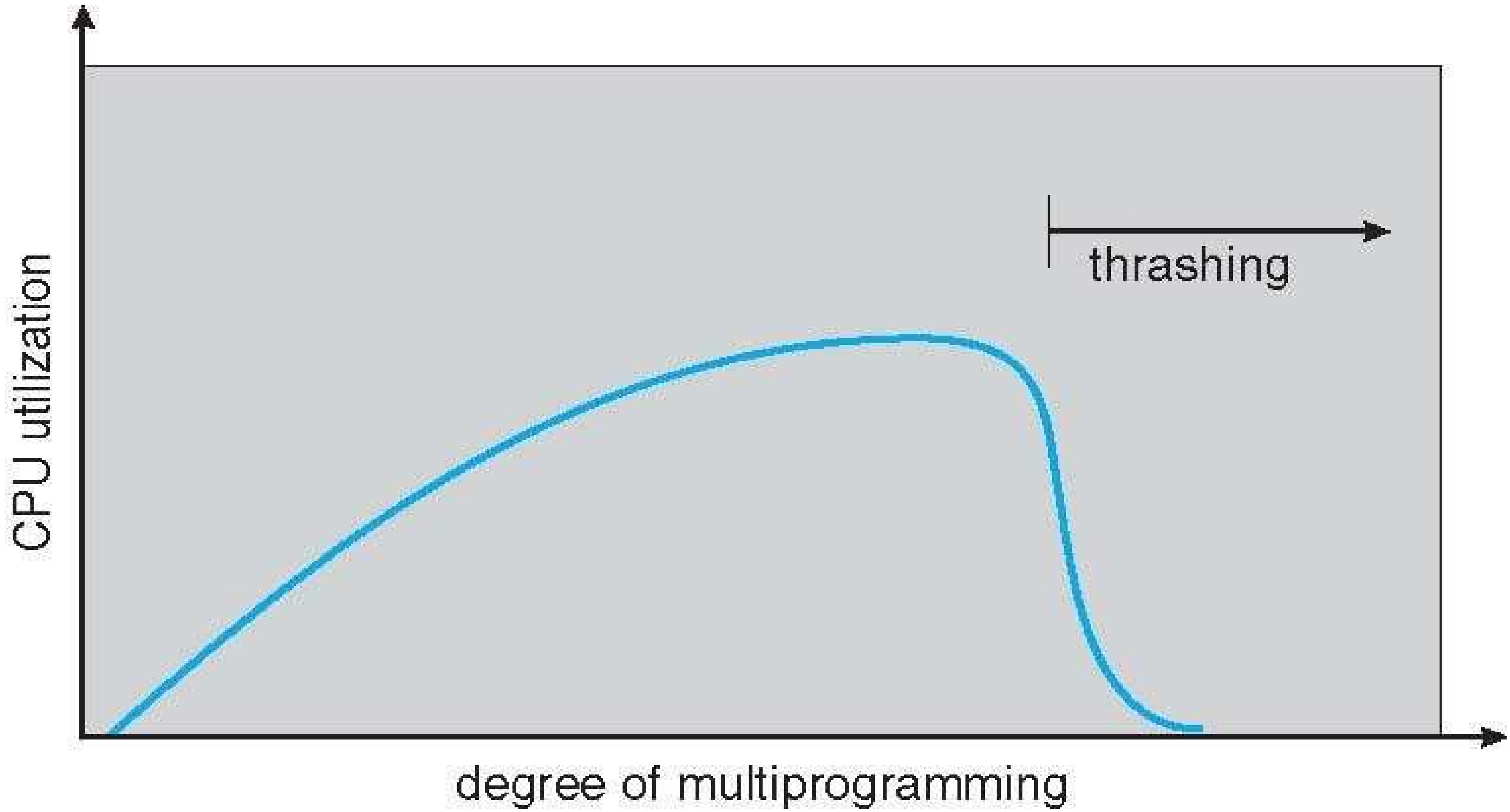
Table 8.5 Resident Set Management

	Local Replacement	Global Replacement
Fixed Allocation	• Number of frames allocated to process is fixed. • Page to be replaced is chosen from among the frames allocated to that process.	• Not possible.
Variable Allocation	• The number of frames allocated to a process may be changed from time to time, to maintain the working set of the process. • Page to be replaced is chosen from among the frames allocated to that process.	• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Thrashing

- **Thrashing** ≡ a process is busy swapping pages in and out
- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

Thrashing



End of Chapter 10