CECS 326
Operating Systems

Virtual Memory

(Reference: Chapter 10. Operating system Concepts by Silberschatz, Galvin and Gagne)

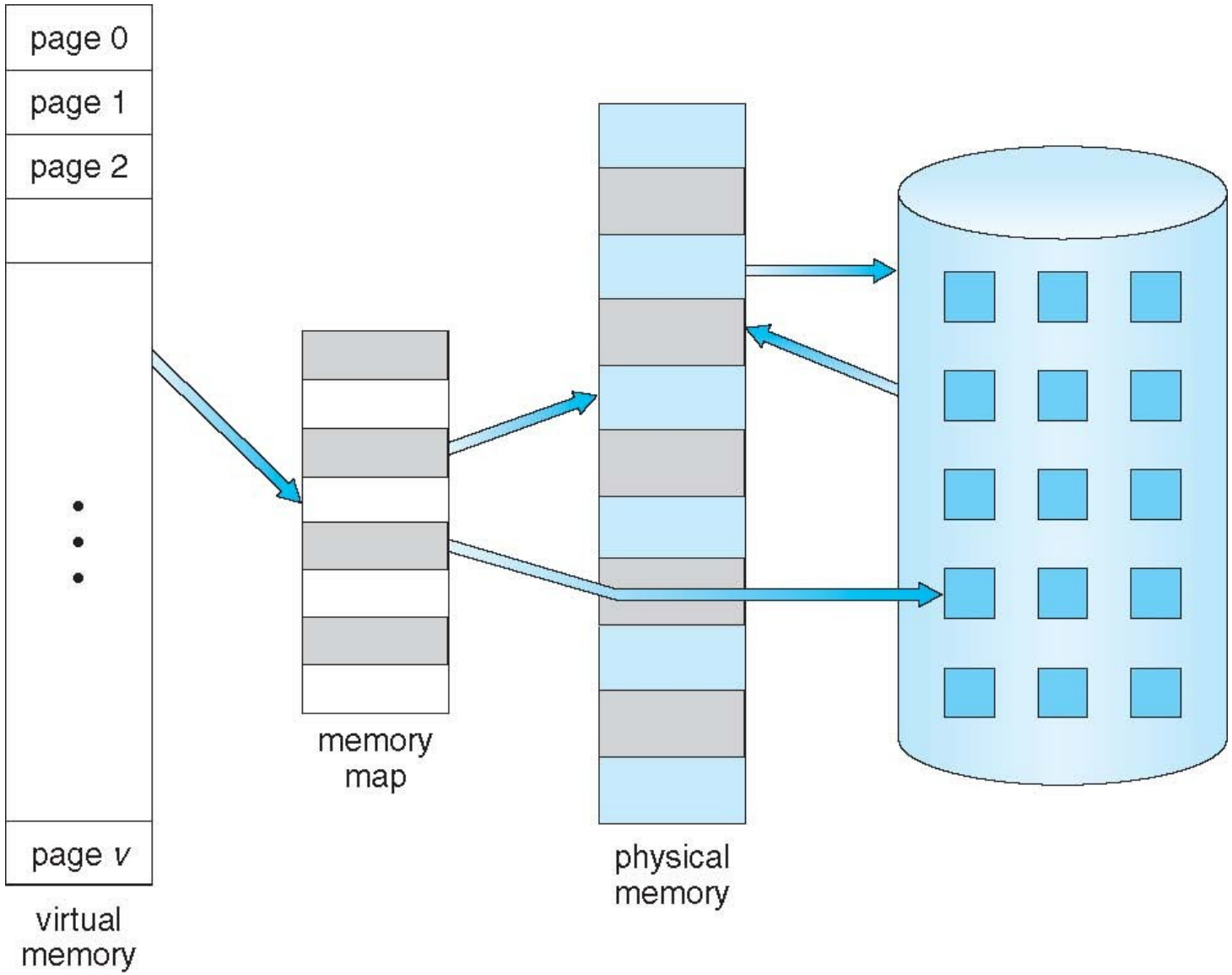
Background of Virtual Memory

- Virtual memory – a technique that allows a process to execute without its entire address space in memory
- Code needs to be in memory to execute, but entire program rarely used
  - E.g., error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Benefits of being able to execute partially-loaded program
  - Program no longer constrained by limits of 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
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Related Concepts

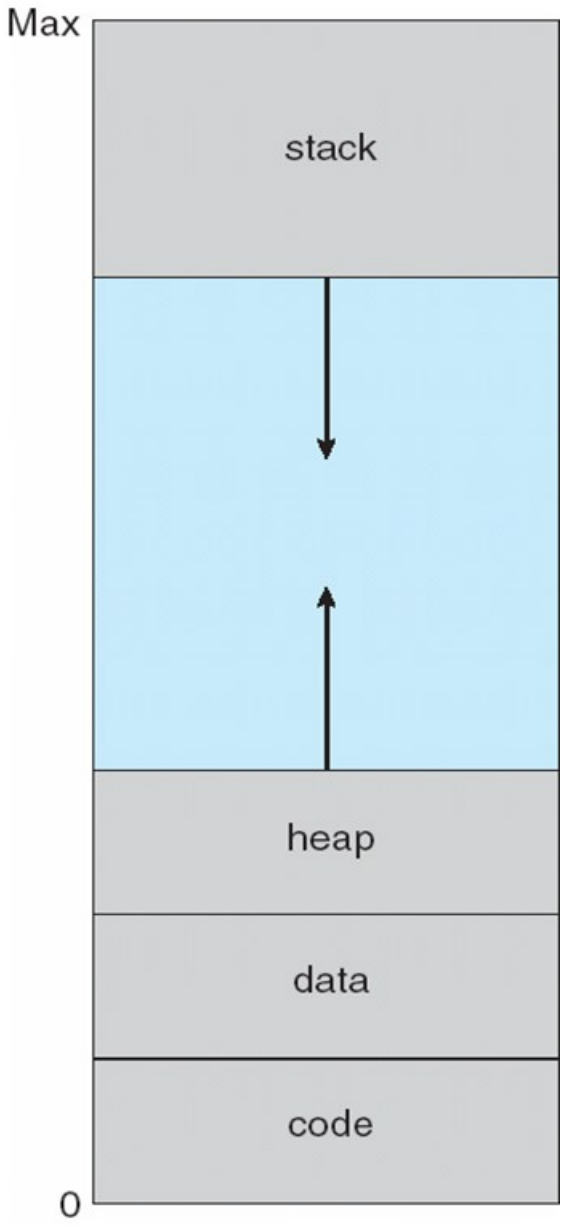
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- **Logical address space** – logical view of how a process is organized in memory
  - Usually start at address 0, contiguous addresses until end of space, space divided into pages (or segments)
  - Meanwhile, physical memory organized in page frames (aka memory blocks)
  - Up to memory management unit (MMU) to map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Logical Memory vs Physical Memory

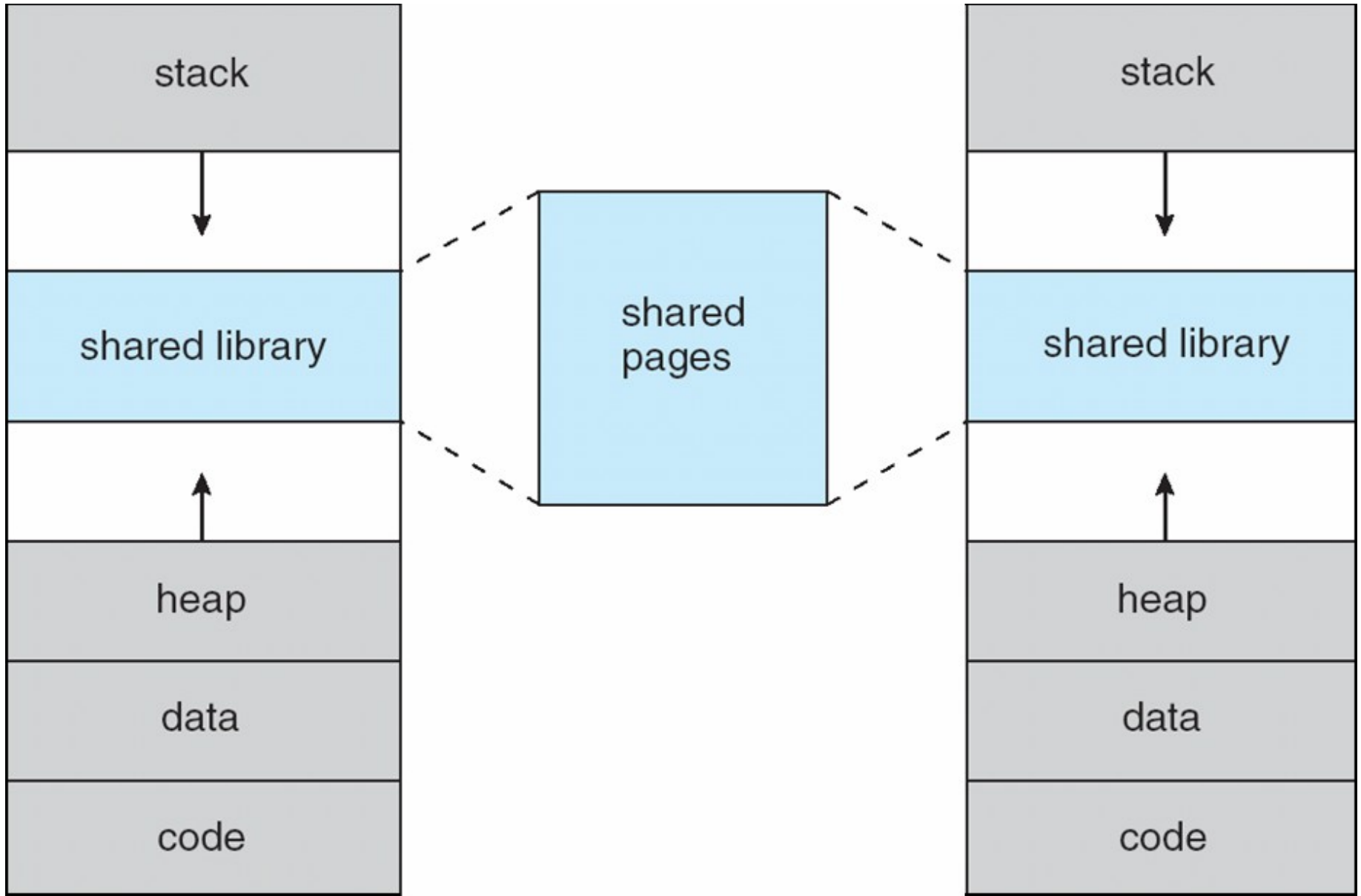


Logical Address Space

- A process' logical address space consists of code and data segments. Data segment includes the static partition, stack and heap.
- Usually design logical address space for stack to start at maximum logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Using sparse address spaces with holes allows growth, e.g. dynamically linked libraries
- System libraries can be shared among processes via mapping into logical address space
- Shared memory can be mapped into individual processes' logical address space
- Pages can be shared during `fork()`, thus speeding process creation



Shared Library Using Virtual Memory



Memory Management Policies

- Fetch policy
  - Decision on when to load
  - In non-virtual memory system, entire address space loaded before execution begins
  - In virtual memory system implemented with demand paging, load only the part being referenced
- Placement policy
  - Decision on where to load
  - With contiguous memory allocation, need memory space large enough to hold the loaded address space
  - With paged memory organization, simply find enough page frames to hold the loaded address space
- Replacement policy
  - Decision on which part to replace
  - Only relevant in virtual memory implementation where process can execute with partial address space loaded

Demand Paging

- Instead of bringing a process' entire address space into memory, bring a page into it only when it is needed
  - No unnecessary I/O, hence less I/O performed
  - Less memory needed
  - Faster response
  - More users
- Page is needed means a reference is made to it
  - If reference is invalid, then abort
  - If the referenced page is not-in-memory, bring to memory
- Component in OS that brings a page into memory is a pager

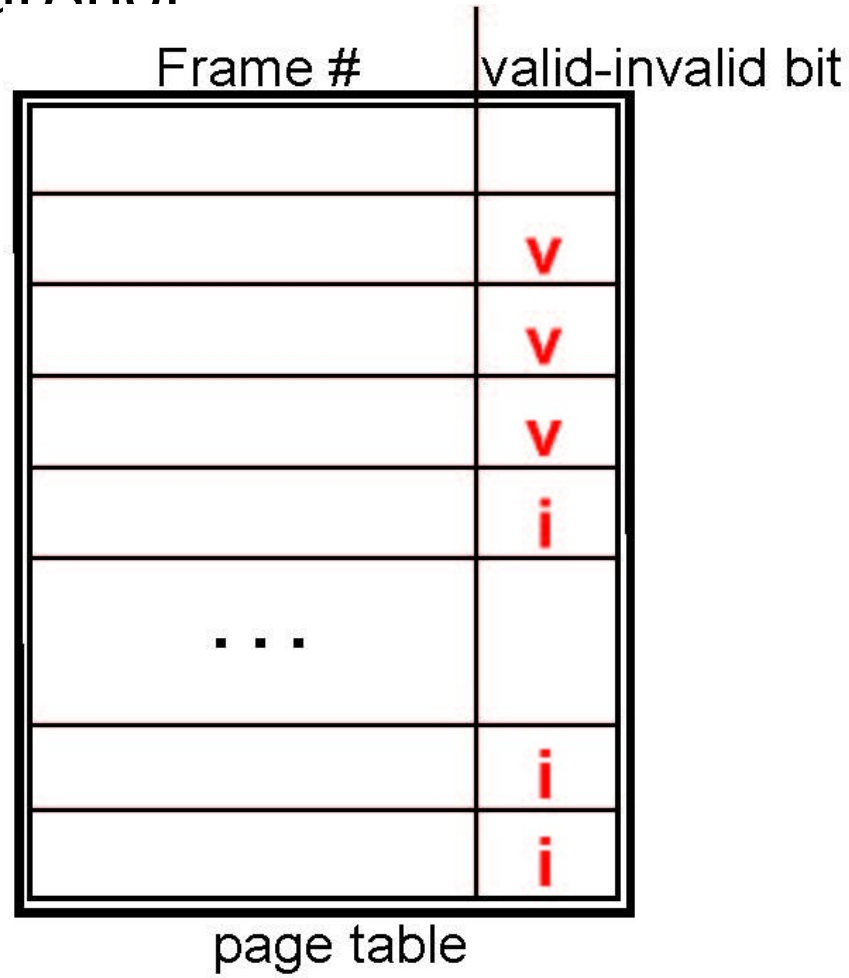
Basic Concepts of Demand Paging

- How to determine the set of pages to bring in?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non demand-paging
- If page needed is not memory resident
  - Need to locate and load the page into memory from secondary storage
    - Without changing program behavior
    - Without programmer needing to change code

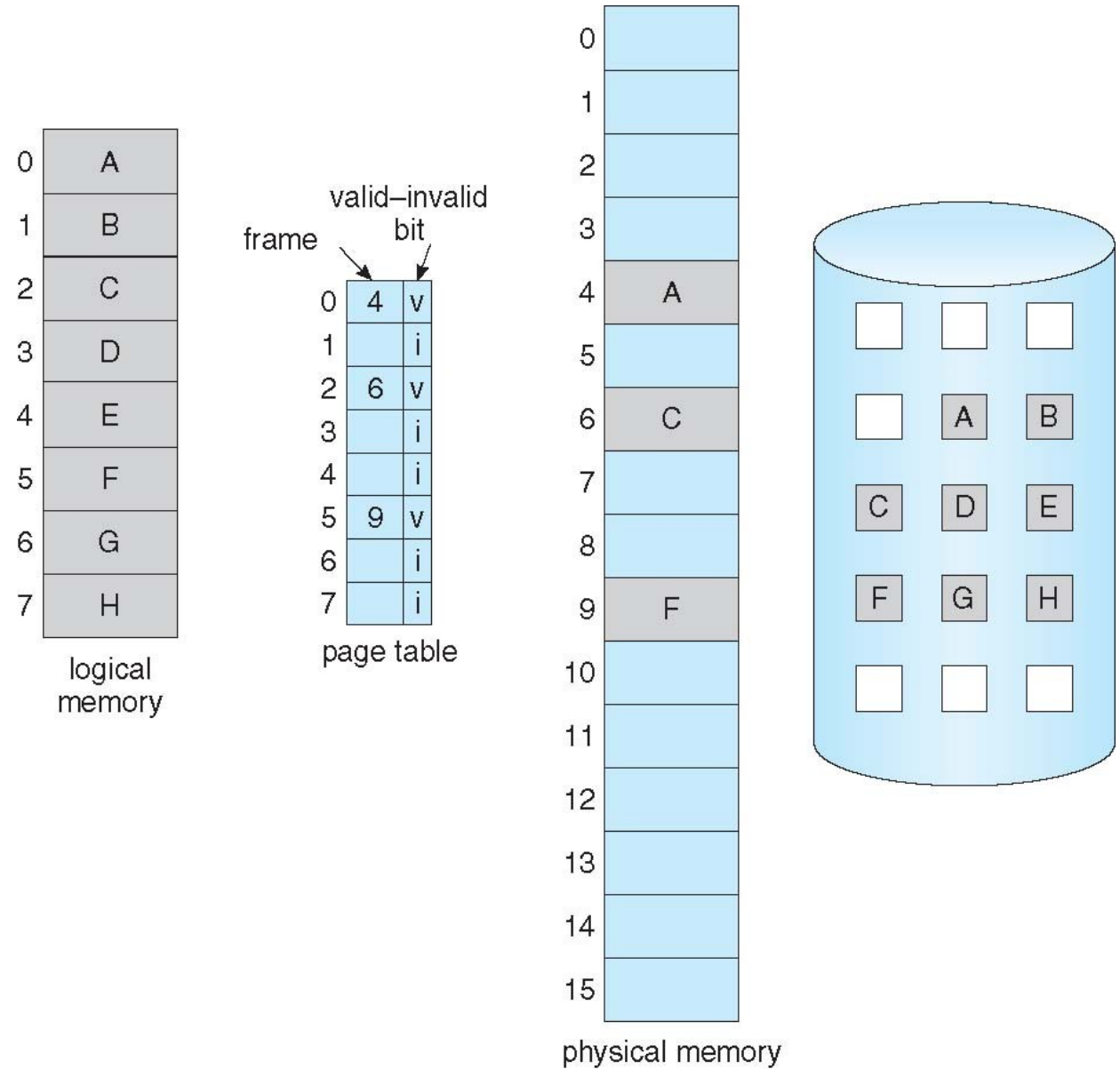
Page Table and Valid-Invalid Bit

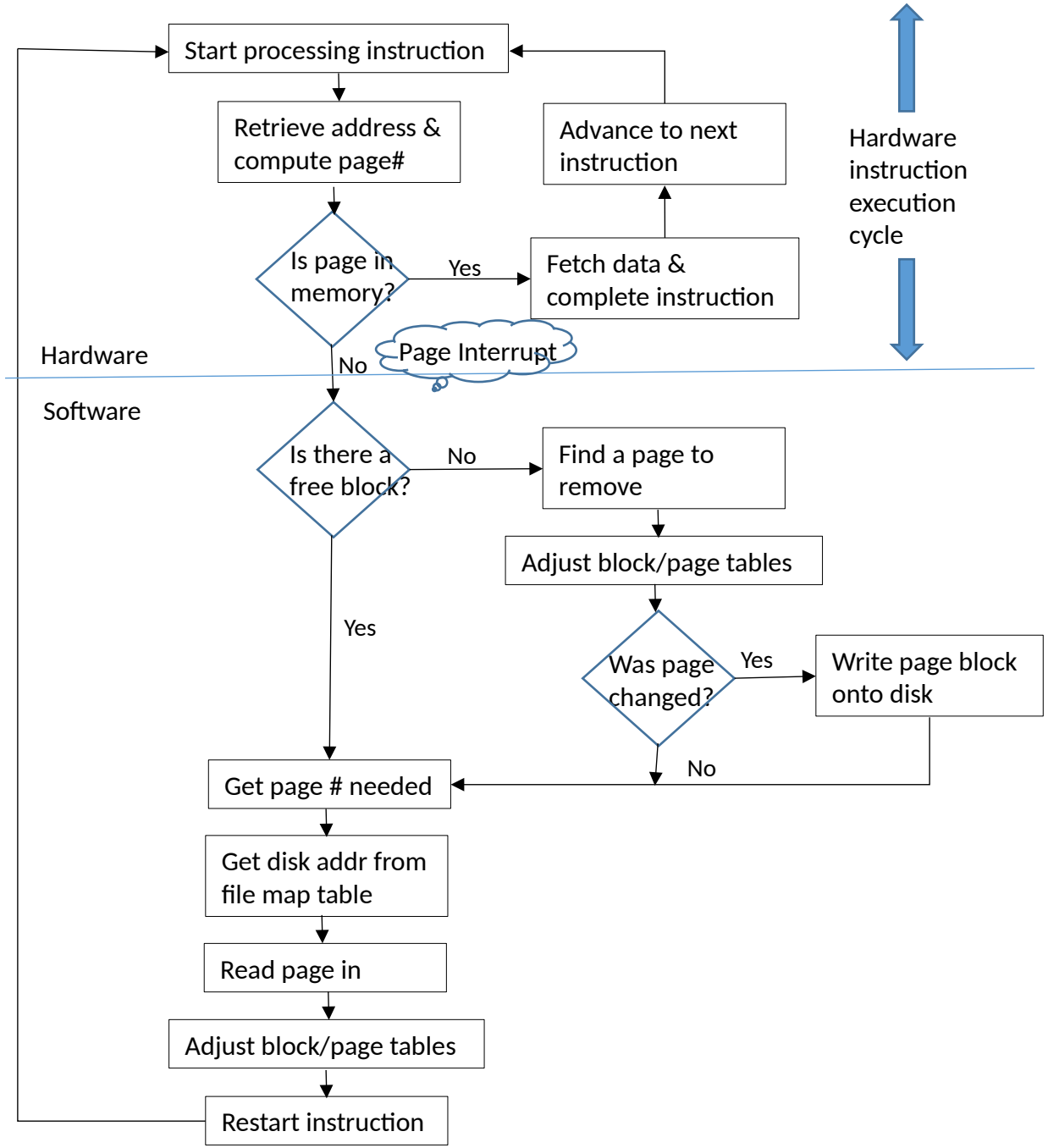
- Need a page table to keep track of memory allocation of each page
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory (memory resident), **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- A hardware device called Memory Management Unit (MMU) is used to translate from logical address to physical address
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

- Example of a page table snapshot:



Page Table When Some Pages Are Not in Memory

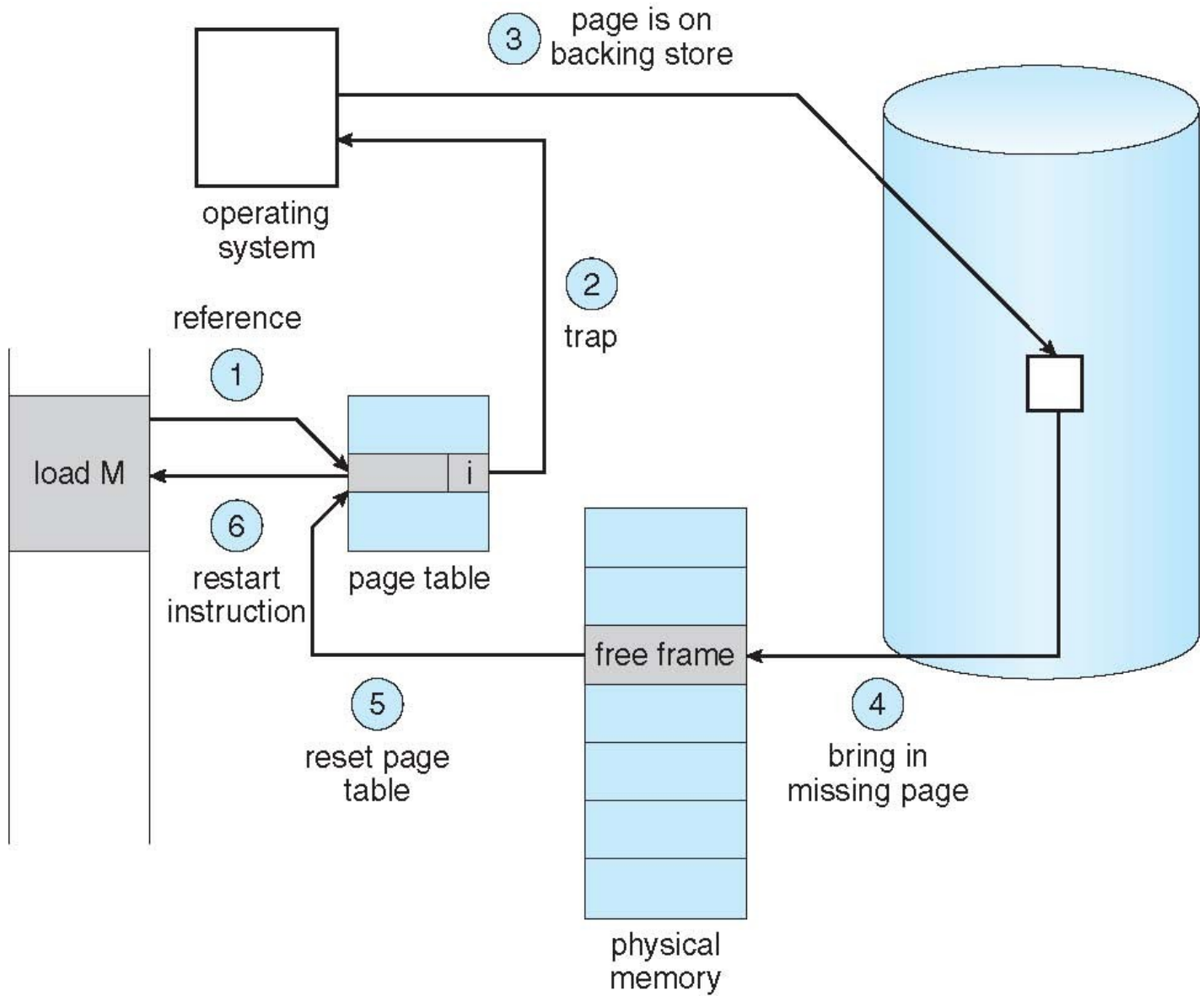




Instruction Execution in Demand Paging

- When page being referenced is not in memory, page fault occurs & need to bring page in
  - Find a free frame
  - Swap page into frame via scheduled disk operation
  - Reset tables to indicate page is now in memory & set valid-invalid bit to “v”
  - Restart instruction that caused the page fault
  - If no free frame, need to replace

Steps in Handling Page Fault



Steps at page fault in Demand Paging

1. Trap to kernel
2. Save the user registers and program counter
3. Determine that the interrupt was a page fault
4. Find a free page frame
    1. Allocate if one is available
    2. Select a page to remove if not, and schedule a write-back if page had been modified
5. Determine location of the page on disk
6. Issue a read from the disk to the free frame
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
7. While waiting, allocate the CPU to some other user process
8. Receive an interrupt from the disk I/O subsystem (I/O completed)
9. Save the registers and program counter for the other user process
10. Determine that the interrupt was from the disk
11. Correct the page table and other tables to show page is now in memory
12. Wait for the CPU to be allocated to this process again
13. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance Effect of Demand Paging

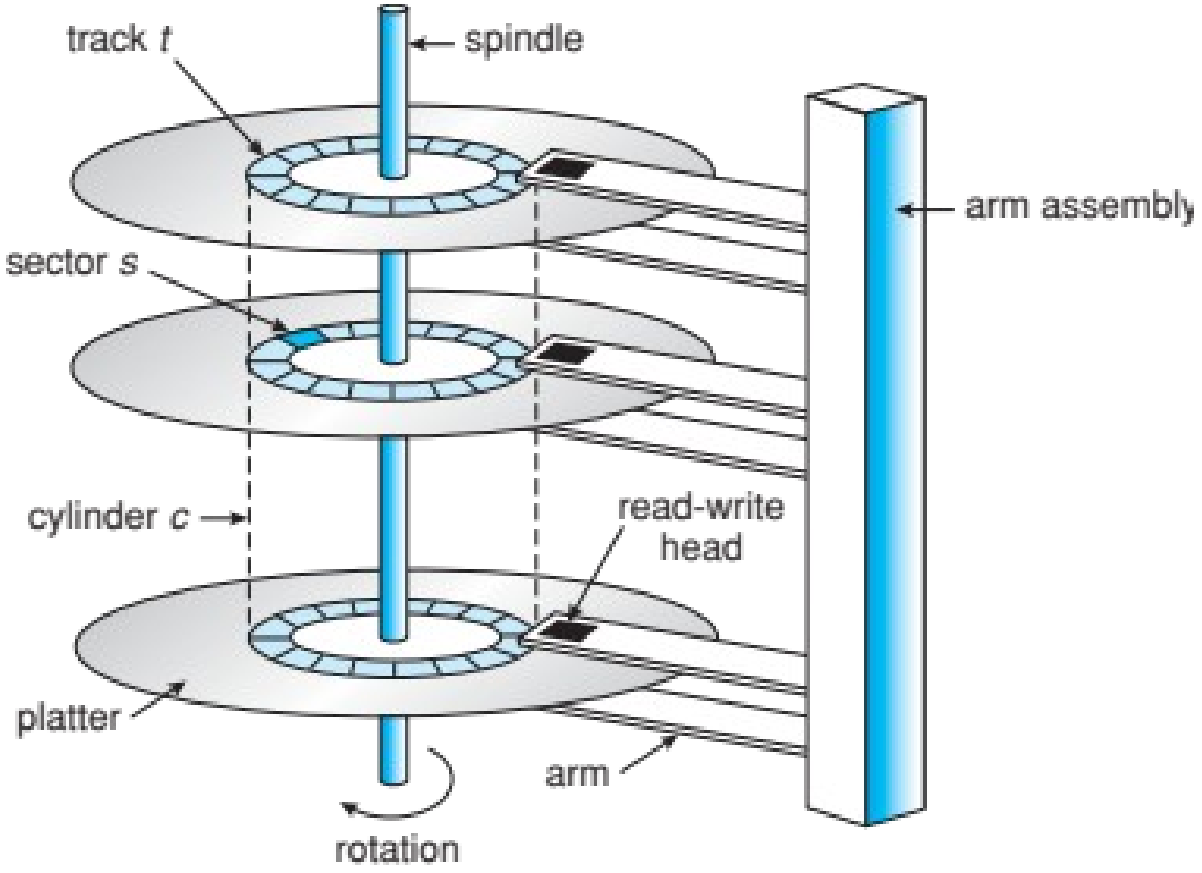
- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time needed to access disk & transfer page in
  - Restart the process – again just a small amount of time
  - All three together called page fault service overhead
- 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)

  EAT = (1 – p) x memory access

  + p (page fault service overhead

  + memory access)

Moving-Head Disk Mechanism



Hard Disk Performance

Access Latency = **Average access time** = average seek time + average latency

For fastest disk 3ms + 2ms = 5ms

For slow disk 9ms + 5.56ms = 14.56ms

Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead

Performance Effect of Demand Paging

- Assume
  - Memory access time = 200 nanoseconds (nsec)
  - Average page-fault service time = 8 milliseconds (msec)
- Effective Access Time (EAT)

  EAT = (1 – p) x 200 nsec + p (8 msec+200 nsec)

  = 200 – p x 200 + p x 200 + p x 8,000,000

  = 200 + p x 8,000,000
- If one access out of 1,000 causes a page fault, i.e., p = 1/1000, then

  EAT = 8,200 nsec = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent

  220 > 200 + 8,000,000 x p
  20 > 8,000,000 x p

  p < .0000025

  i.e., can only afford < one page fault in every 400,000 memory accesses

What Happens If There Is No Free Frame

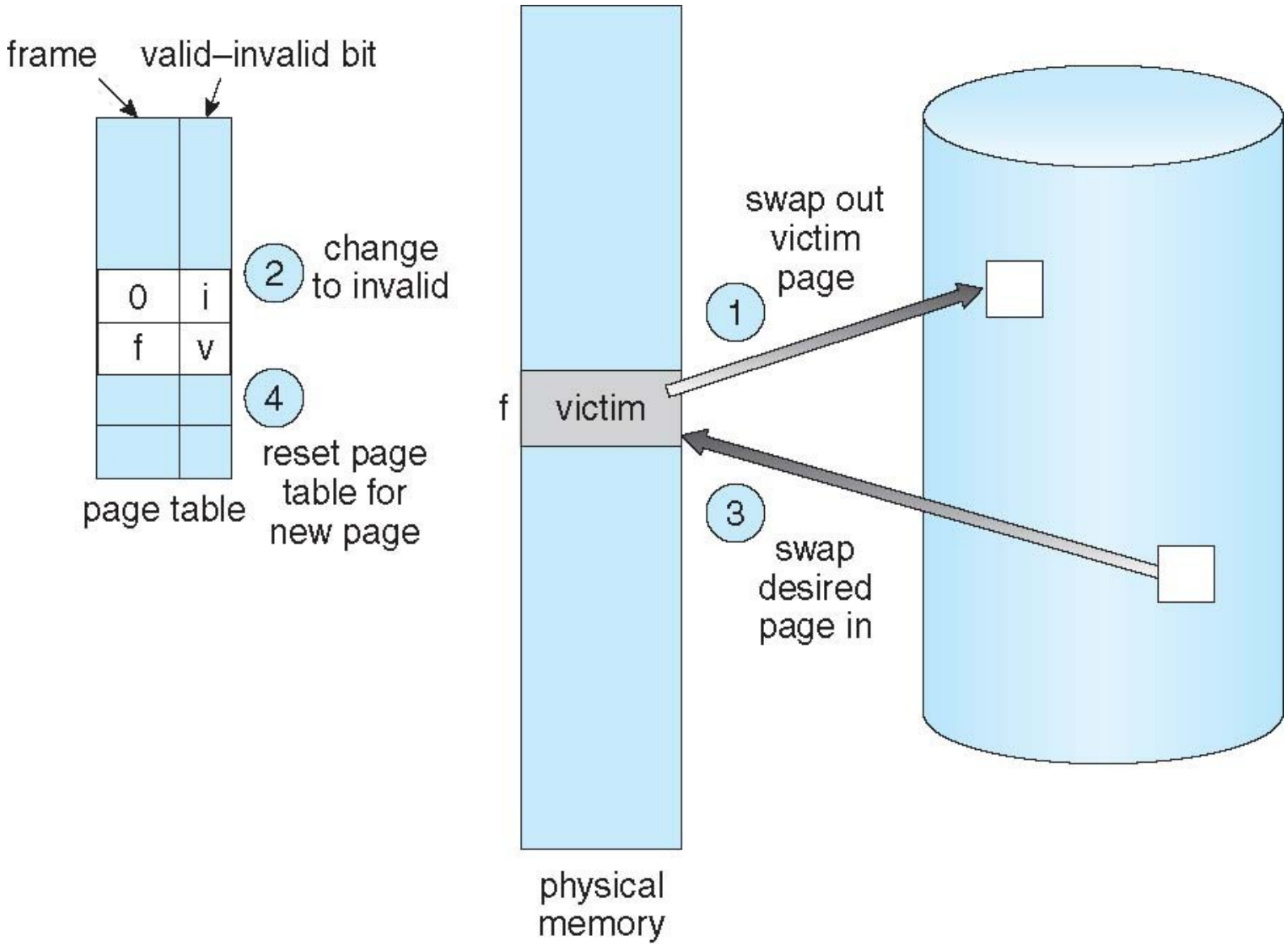
- Page frames may be used up by process pages
- Page frames are also in demand from the kernel, I/O buffers, etc.
- How much to allocate to each process?
- Page replacement – find some page in memory, but not really in use, page it out
  - Reason for finding page replacement, as opposed to swapping out process
  - Performance consideration – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

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 (need dirty bit in page table)
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

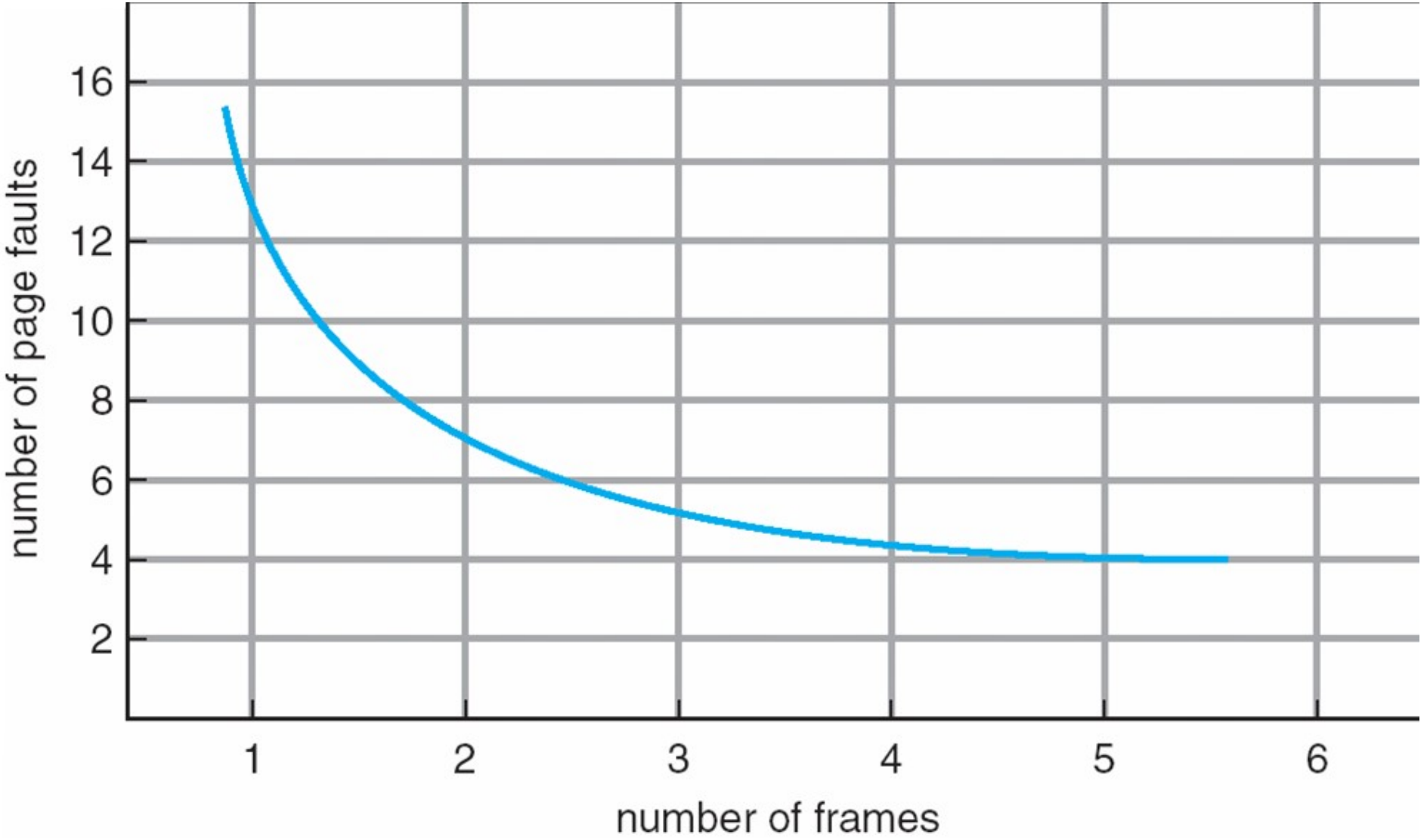


Frame Allocation & Page Replacement

- **Frame-allocation algorithm** determines
  - How many frames to give each process
- **Page-replacement algorithm** selects
  - Which frames to replace
- **Objective**
  - 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
- In all our examples, the **reference string** of referenced page numbers is

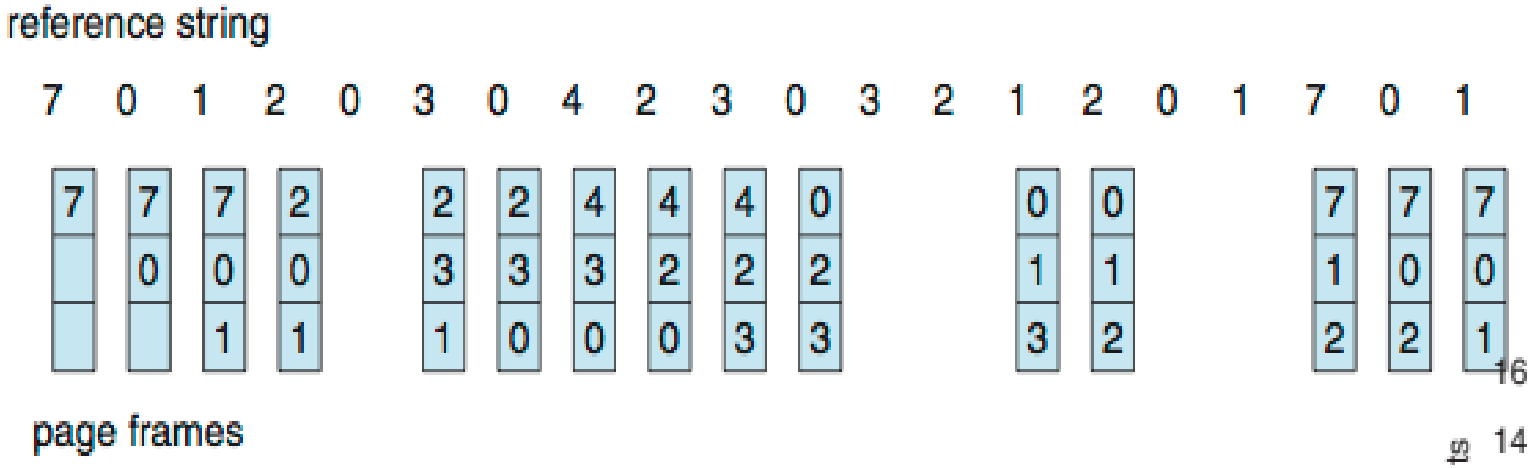
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Expected Graph of Page Faults vs. Number of Frames



First-In-First-Out (FIFO) Algo

- 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
- Consider another string: 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

2 frames:12 page faults

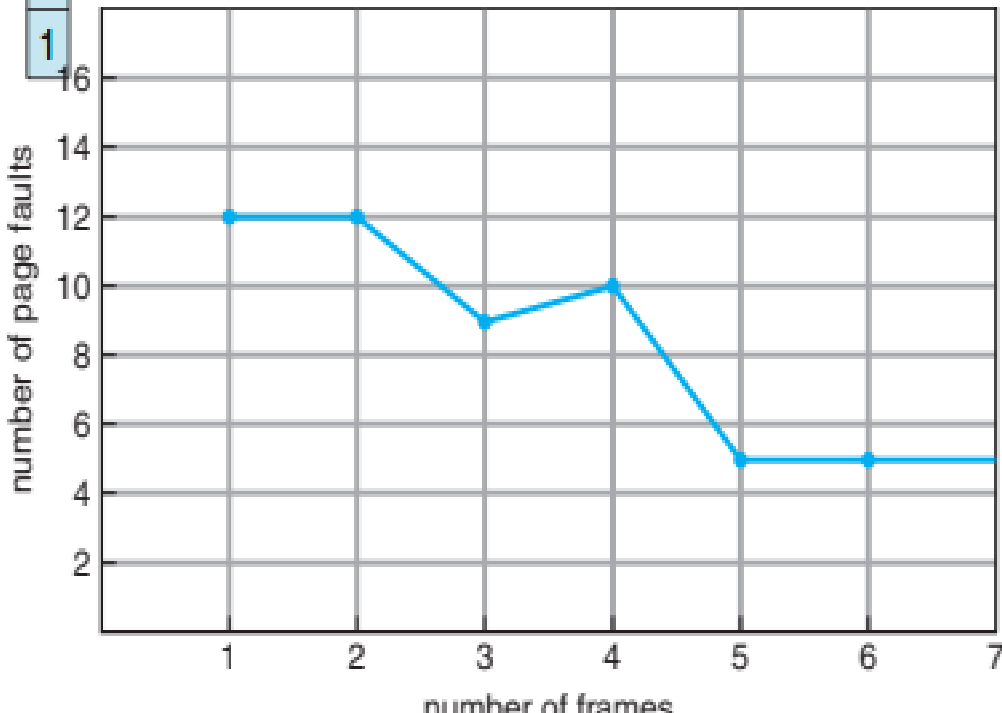
	1	2	3	4	1	2	5	1	2	3	4	5
	*	*	*	*	*	*	*	*	*	*	*	*
	1	1	3	4	4	2	2	5	2	3	3	5
		2	2	3	1	1	5	1	1	2	4	4

3 frames: 9 page faults

	1	2	3	4	1	2	5	1	2	3	4	5
	*	*	*	*	*	*	*			*	*	
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4

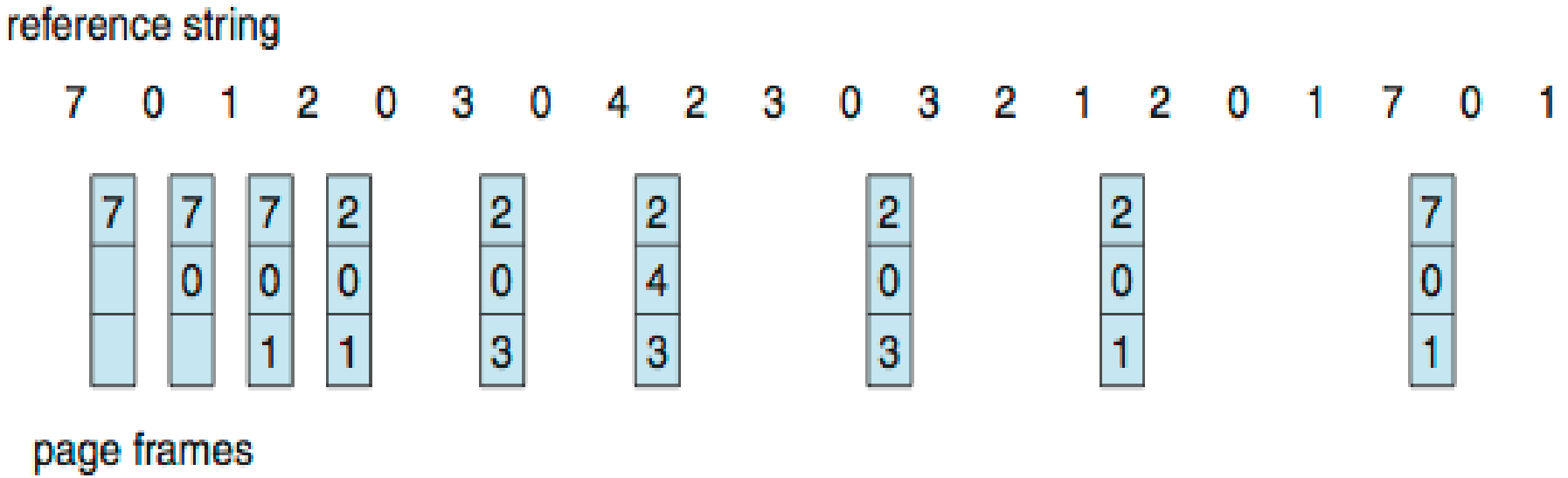
FIFO queue

Tail	1	2	3	4	1	2	5			3	4	
		1	2	3	4	1	2			5	3	
Head			1	2	3	4	1			2	5	



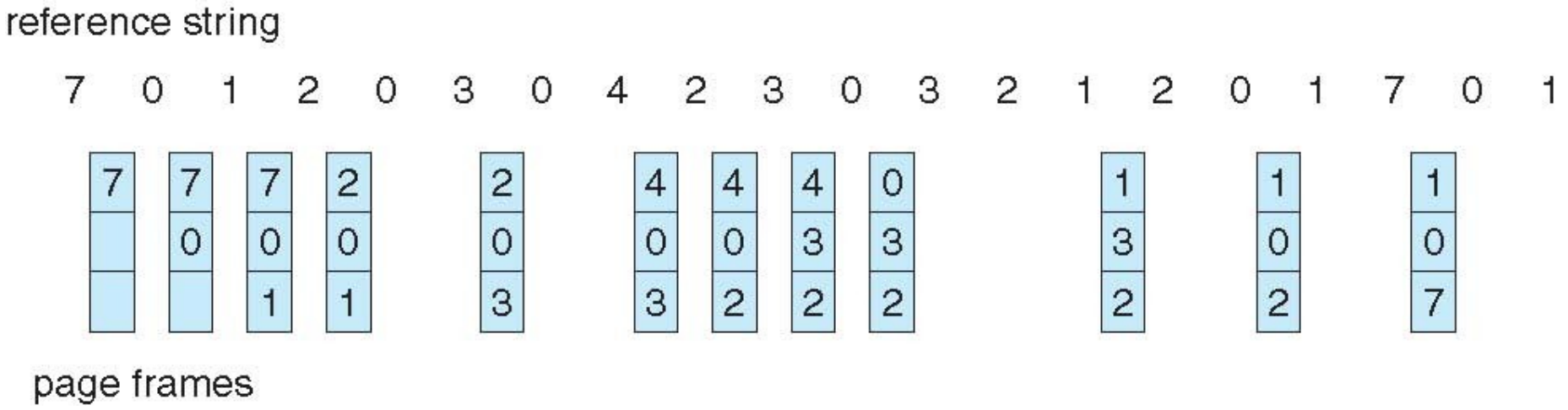
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- Difficult to implement since we can't read the future
- Useful for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

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



- 12 faults – better than FIFO but worse than OPT
- Reflects observed behavior (locality) in program execution
- Generally good algorithm and frequently used
- But how to implement?
  - Use timestamps
  - Use a stack
  - Second chance algorithm: a simplified and approximate implementation

Implementation of LRU Algorithm

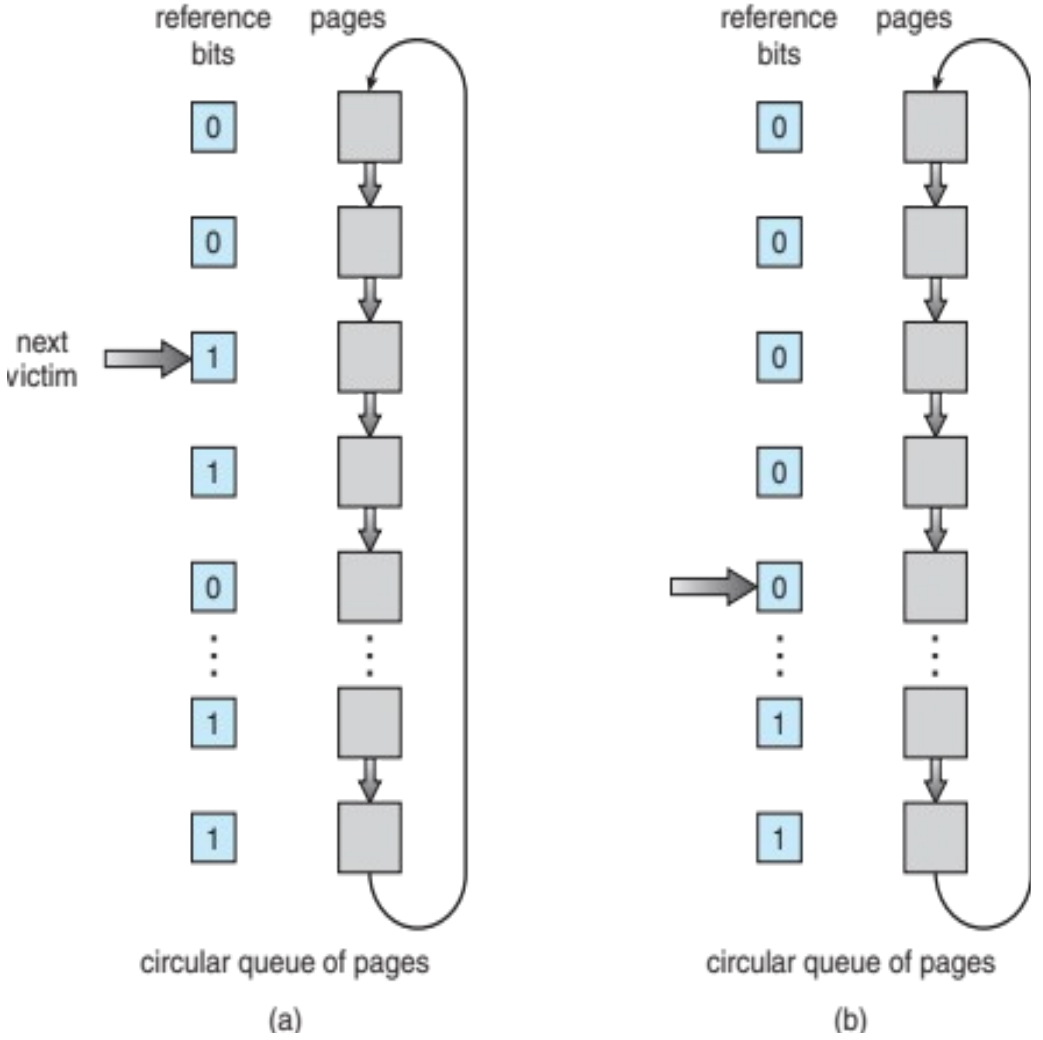
- Counter (to hold timestamp) implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - Each update more expensive
  - No search for replacement

w	4	7	0	7	1	0	1	2	1	2	7	1	2
Stack										2	7		
										1	2		
										0	1		
										7	0		
										4	4		

	Old stack				
	← 2 →	← 1 →	← 0 →	← 7 →	← 4 →
New stack - 6 pointers need to be changed					
← 7 →	← 2 →	← 1 →	← 0 →		← 4 →

LRU Approximation - Second Chance Algorithm

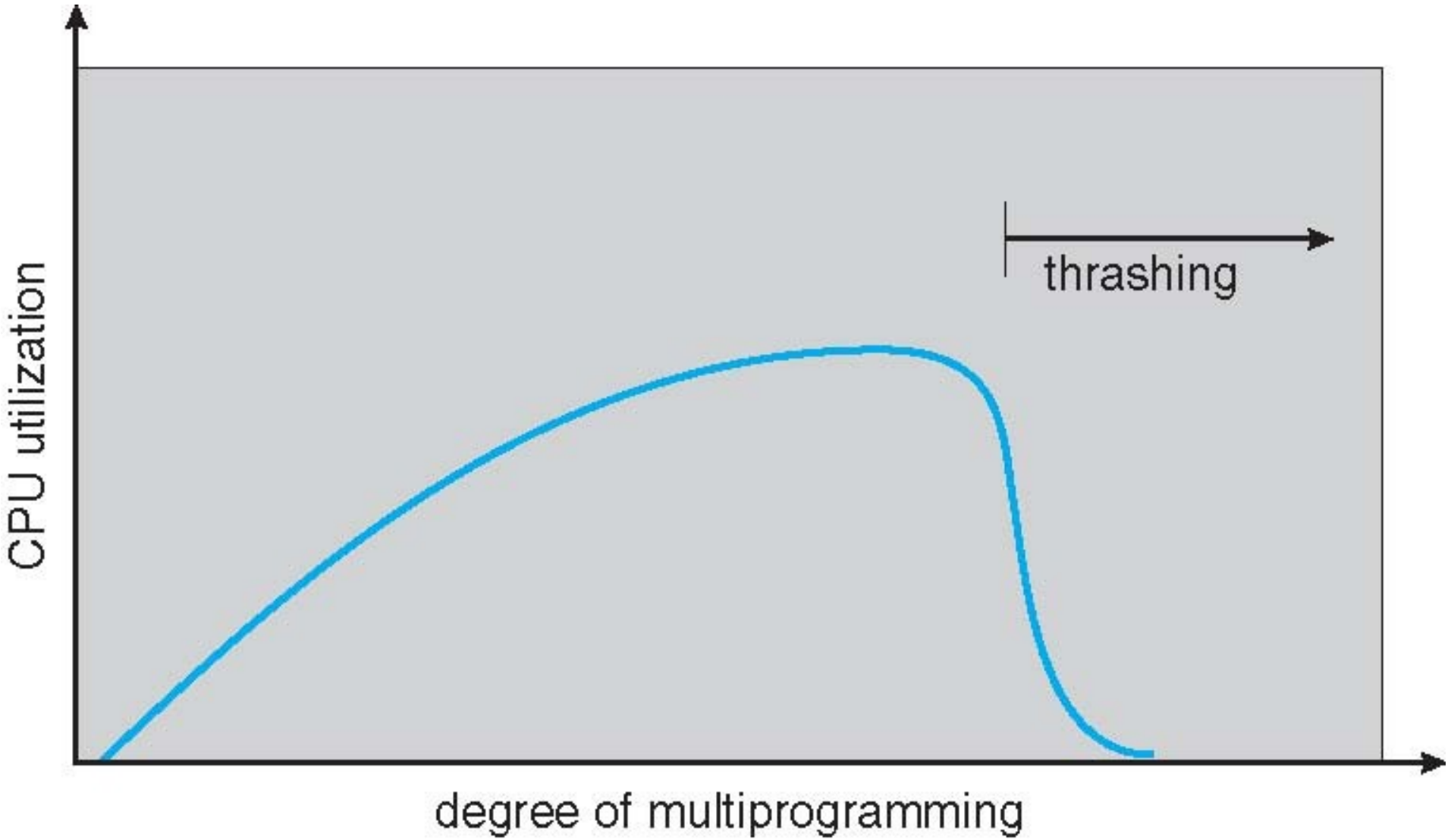
- LRU needs special hardware to support
- **Reference bit – hardware provided**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Maintains a pointer to the reference bits one after another (Think of it as the hand of a clock circulating through every element in the circular linked list of reference bits. Hence this algorithm is also called the clock algorithm.)
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules
  - A page that is used often tends to stay in memory.



Thrashing

- If a process does not have enough frames to hold pages in active use, the page-fault rate can be very high
  - Page fault to get page
  - Replace a resident page to free a frame
  - But the replaced page is needed again soon
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** $\equiv$ a phenomenon where a process is spending more time swapping pages in and out than running
- Solution: working set model

Thrashing



Working Sets

- A working set can be loosely defined as the set of pages likely to be referenced again in the immediate future.
- *Working set principle*:
  - A process may execute only if its working set is resident in main memory. A page may not be removed from main memory if it is in the working set of an executing process.
- *Motivation*:
  - If we limit the number of active processes too much, throughput suffers; if we do not control them enough, then each process has inadequate real memory for its execution and thrashing can result. Working sets help determine the optimum balance between the policies for process and memory management.
- One method for defining the working set: With a preset parameter Δ for a process at time t, working set W(t, Δ) is the set of pages that have been referenced in the last Δ time units. Δ is the window size and is a tunable parameter.

Execution Using Working Set with Window Size $\Delta = 5$

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Reference String	a	e	h	a	e	h	b	d	g	b	d	g	a	e	h	b	f	i	c	f	i	b	e	h	a	d	g	a	d	g
Page Fault	*	*	*				*	*	*				*	*	*	*	*	*	*			*	*	*	*	*	*			
W(t,5)	a	e	h	a	e	h	b	d	g	b	d	g	a	e	h	b	f	i	c	f	i	b	e	h	a	d	g	a	d	g
		a	e	h	a	e	h	b	d	g	b	d	g	a	e	h	b	f	i	c	f	i	b	e	h	a	d	g	a	d
			a	e	h	a	e	h	b	d	g	b	d	g	a	e	h	b	f	i	c	f	i	b	e	h	a	d	g	a
							a	e	h	h			b	d	g	a	e	h	b	b		c	f	i	b	e	h	h		
								a	e					b	d	g	a	e	h				c	f	i	b	e			