Operating System Concepts

Lecture 27: Virtual Memory Management

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Demand paged virtual memory
- Page replacement policies
    - FIFO
    - Optimal
    - LRU and its approximations

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out

free-frame list: a pool of free frames

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out
- while I/O is being done, OS context switches to another process
- OS gets interrupt when I/O completes (i.e., the page is read from disk and loaded into memory)
  - saves the registers and state of the other process

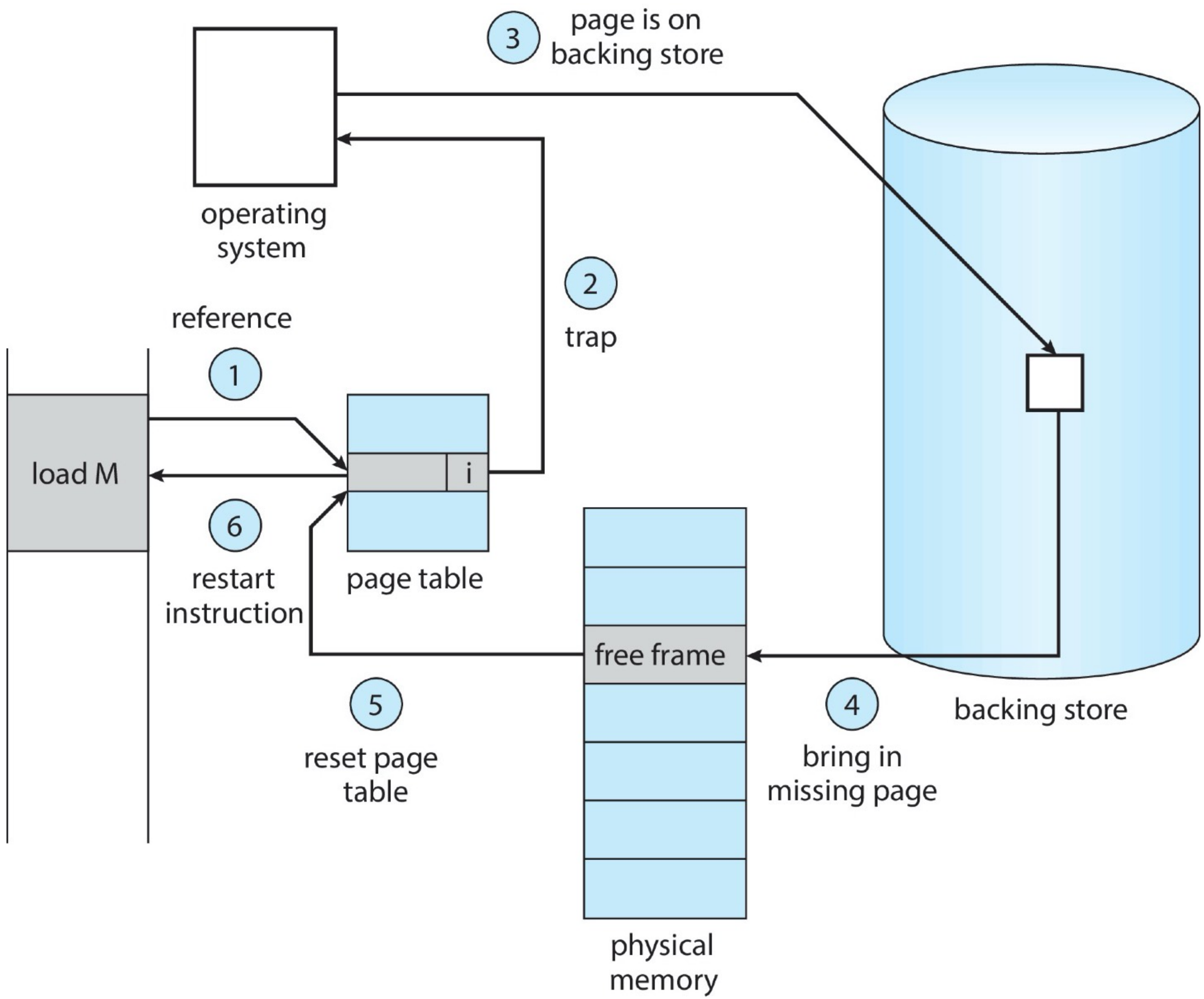
What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out
- while I/O is being done, OS context switches to another process
- OS gets interrupt when I/O completes (i.e., the page is read from disk and loaded into memory)
  - saves the registers and state of the other process
- OS updates the page table entry

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out
- while I/O is being done, OS context switches to another process
- OS gets interrupt when I/O completes (i.e., the page is read from disk and loaded into memory)
  - saves the registers and state of the other process
- OS updates the page table entry
- OS puts the faulting process back on the ready queue
  - resumes the interrupted instruction when CPU is allocated to this process again

Handling a page fault



Understanding the disk read latency

- the delay is the sum of the following
  - time spent waiting in a queue for this device until the read request is serviced
  - time spent waiting for the device seek and rotational latency
  - time that it takes to transfer the page to a free frame

Swap space

- what happens when a page is removed from memory?
  - if the page contained code, we could simply remove it since it can be reloaded from the disk
  - if the page contained data, we need to save the data so that it can be reloaded if the process it belongs to refers to it again
    - it is stored in the **swap space**, i.e., a portion of the disk is reserved for storing pages that are evicted from memory
    - **why?** swap space I/O is faster than file system I/O because swap is allocated in larger chunks and requires less management than file system

Swap space

- what happens when a page is removed from memory?
  - if the page contained code, we could simply remove it since it can be reloaded from the disk
  - if the page contained data, we need to save the data so that it can be reloaded if the process it belongs to refers to it again
    - it is stored in the **swap space**, i.e., a portion of the disk is reserved for storing pages that are evicted from memory
    - **why?** swap space I/O is faster than file system I/O because swap is allocated in larger chunks and requires less management than file system
- so at any given time, a page of virtual memory might exist in one or more of the following locations
  - file system, physical memory, swap space

Swap space

- what happens when a page is removed from memory?
  - if the page contained code, we could simply remove it since it can be reloaded from the disk
  - if the page contained data, we need to save the data so that it can be reloaded if the process it belongs to refers to it again
    - it is stored in the **swap space**, i.e., a portion of the disk is reserved for storing pages that are evicted from memory
    - **why?** swap space I/O is faster than file system I/O because swap is allocated in larger chunks and requires less management than file system
- so at any given time, a page of virtual memory might exist in one or more of the following locations
  - file system, physical memory, swap space
- need additional bits in the page table to find out where to find a page

Why demand paging works?

- in theory, a process can access a new page with each instruction
- in practice, a process typically exhibits locality of reference
  - **temporal locality**: if a process accesses an item in memory, it will tend to reference the same item again soon
  - **spatial locality**: if a process accesses an item in memory, it will tend to reference an adjacent item soon

Performance of demand paging

- let ρ_{fault} be the probability of a page fault ($0 \le \rho_{fault} \le 1$)
  - the **effective access time** (EAT) is

$$(1 - \rho_{fault})C_{ma} + \rho_{fault}C_{pagefault}$$

where C_{ma} is the memory access time and $C_{pagefault}$ is the page fault service time (= servicing the interrupt + reading the page + restarting the process)

    - disk I/O is quite expensive (~4 orders of magnitude more expensive than memory access)

Performance of demand paging

- let ρ_{fault} be the probability of a page fault ($0 \leq \rho_{fault} \leq 1$)
  - the **effective access time** (EAT) is

    $$(1 - \rho_{fault})C_{ma} + \rho_{fault}C_{pagefault}$$

    where C_{ma} is the memory access time and $C_{pagefault}$ is the page fault service time (= servicing the interrupt + reading the page + restarting the process)

    - disk I/O is quite expensive (~4 orders of magnitude more expensive than memory access)

- so if memory access time is 200*ns* and the average page-fault service time is 25*ms*
  - EAT = (1-ρ_{fault}) x 200*ns* + ρ_{fault} x 25,000,000*ns*

    = 200*ns* + ρ_{fault} x 24,999,800*ns*

Performance of demand paging

- if we want the EAT to be only 10% slower than memory access time, what value ρ_{fault} must have?
  - $(1-\rho_{fault})$ x 200 + ρ_{fault} x 25,000,000 = 220 -> ρ_{fault} **~ 10^{-6}**
- if page fault ratio becomes larger, the effective access time approaches the disk access time

How does the OS transparently and safely restart a faulting instruction?

- need hardware support to save
  - the faulting instruction
  - the CPU state
- what about instructions with side effects?
  - page fault may happen in the middle of running an instruction
  - for example CISC instruction `mov a, (r10)+` moves a into the address contained in `register 10` and increments `register 10`
  - **solution**:
    - undo all side effects
    - delay side effects until after all memory references

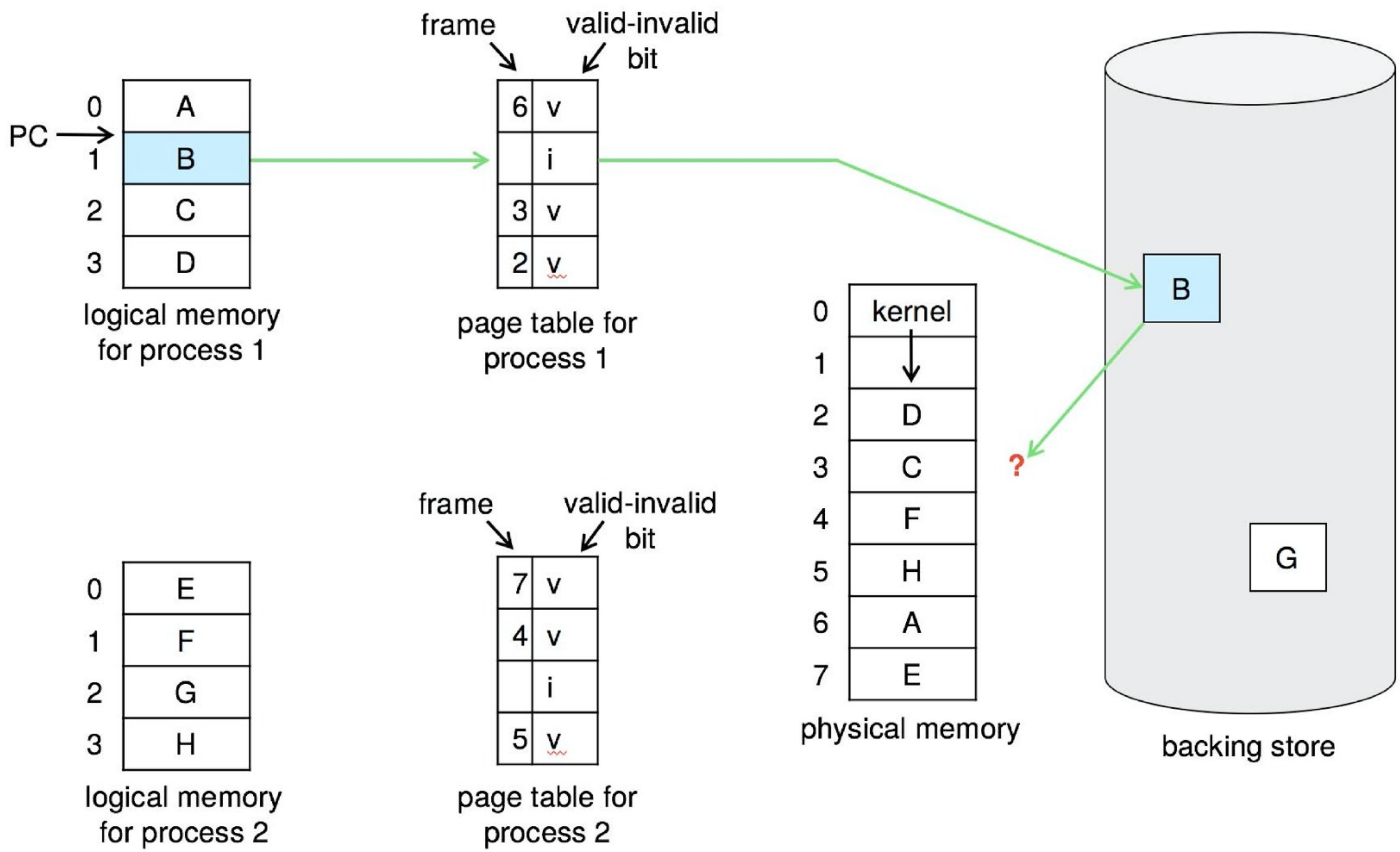
Implementation issues

- placement strategies
  - where to place pages in memory? which frame should be selected?
- replacement strategies
  - what to do when there are more pages than memory frames?
  - a good page replacement algorithm can reduce the number of page faults and improve performance
- load control strategies
  - how many processes can be loaded into memory at once? determines the degree of multiprogramming
  - **tradeoff?**

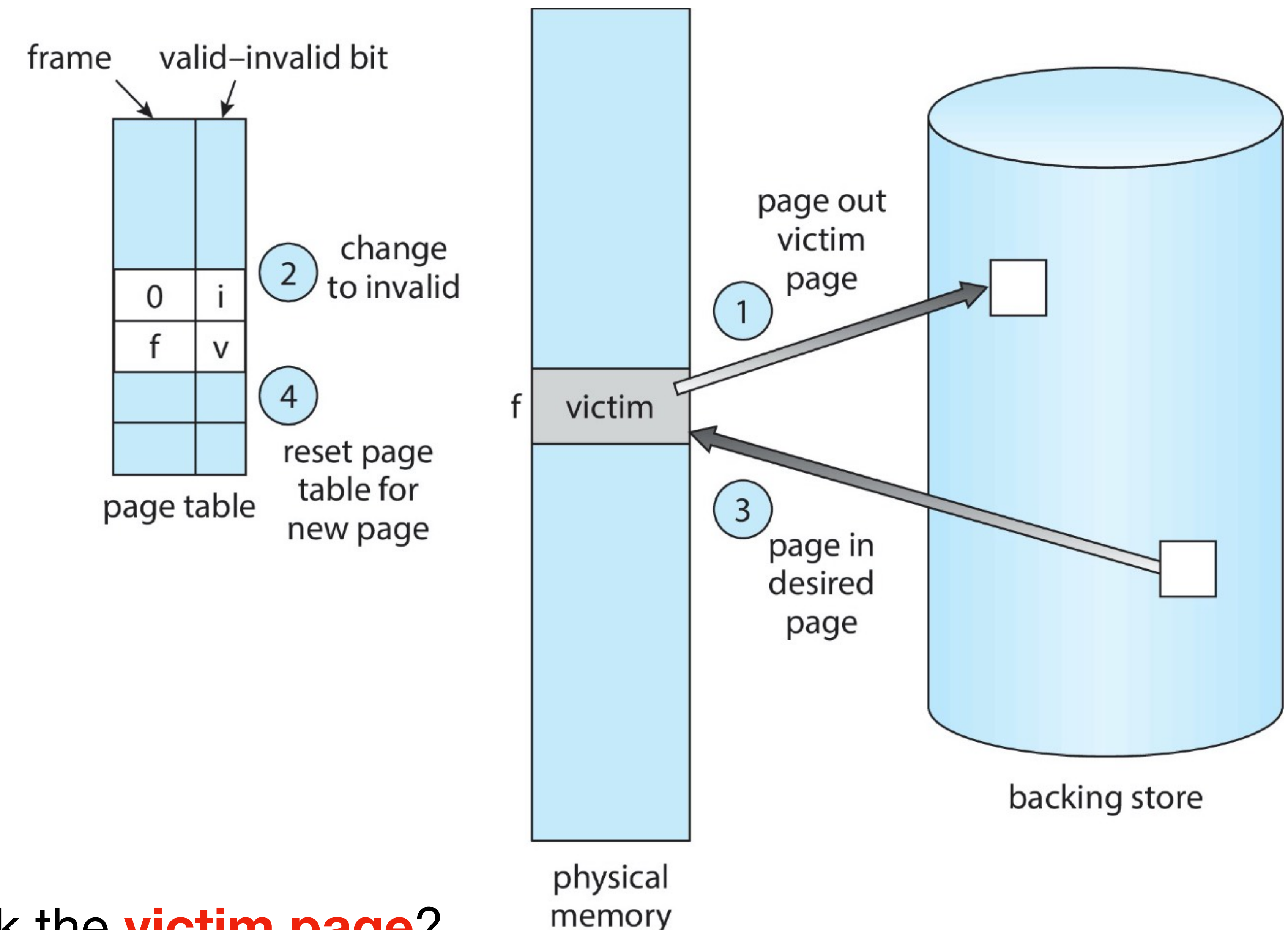
The need for page replacement

when a process faults and memory is full, some page must be swapped out to free a frame

handling a page fault now requires 2 disk accesses



How does it work?



how to pick the **victim page**?

1. local replacement: replace a page of the faulting process
2. global replacement: replace the page of another process

Designing a page replacement algorithm

- record all the pages being accessed
  - for example, if the following virtual addresses are being referenced
    `(3, 0), (1, 9), (4, 1), (2, 1), (5, 3), (2, 0), (1, 9), (2, 4), (3, 1), (4, 8)`
    where each virtual address is (page#, offset)
  - the following stream shows pages being accessed
    `3, 1, 4, 2, 5, 2, 1, 2, 3, 4`

Designing a page replacement algorithm

- record all the pages being accessed
  - for example, if the following virtual addresses are being referenced
    `(3, 0), (1, 9), (4, 1), (2, 1), (5, 3), (2, 0), (1, 9), (2, 4), (3, 1), (4, 8)`
    where each virtual address is (page#, offset)
  - the following stream shows pages being accessed
    `3, 1, 4, 2, 5, 2, 1, 2, 3, 4`
- on a page fault, we need to choose a page to evict
  - how? randomly?
    - amazingly, this algorithm works pretty well
  - better not choose a page that will probably need to be brought back in soon

Page replacement algorithms

- first-in first-out (FIFO): throw out the oldest page
  - simple idea, but the OS can easily throw out a page that is being accessed frequently

reference string

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
frame 1		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
frame 2			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
page fault	●	●	●	●		●	●	●	●	●	●			●	●			●	●	●

no pages in memory at first

15 page faults in total

Does adding memory helps with FIFO?

reference string	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame 0	7	7	7	7	7	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2
frame 1		0	0	0	0	0	0	4	4	4	4	4	4	4	4	4	4	7	7	7
frame 2			1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
frame 3				2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1
page fault	●	●	●	●		●		●			●			●	●			●		

10 page faults in total

FIFO makes bad replacement choices

consider a different reference stream

	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5	5	5	
frame 1		2	2	2	1	1	1	1	1	3	3	
frame 2			3	3	3	2	2	2	2	2	4	
page fault	●	●	●	●	●	●	●			●	●	

9 page faults in total

FIFO makes bad replacement choices

consider a different reference stream

	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	1	1	4	4	4	5	5	5	5	5	
frame 1		2	2	2	1	1	1	1	1	3	3	
frame 2			3	3	3	2	2	2	2	2	4	
page fault	●	●	●	●	●	●	●			●	●	

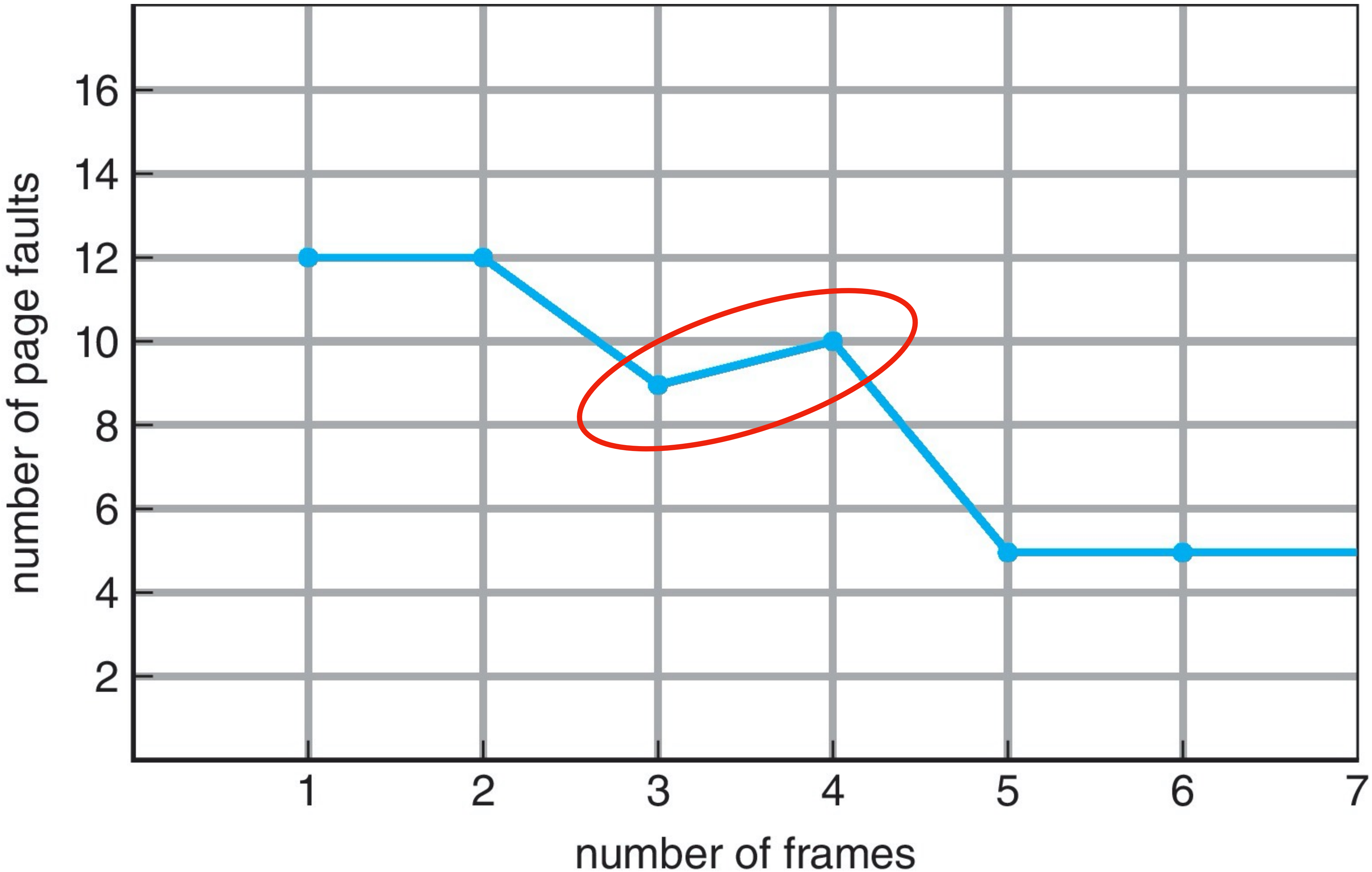
9 page faults in total

frame 0	1	1	1	1	1	1	5	5	5	5	4	4
frame 1		2	2	2	2	2	2	1	1	1	1	5
frame 2			3	3	3	3	3	3	2	2	2	2
frame 3				4	4	4	4	4	4	3	3	3
page fault	●	●	●	●			●	●	●	●	●	●

10 page faults in total

Does adding memory always reduce the number of page faults?

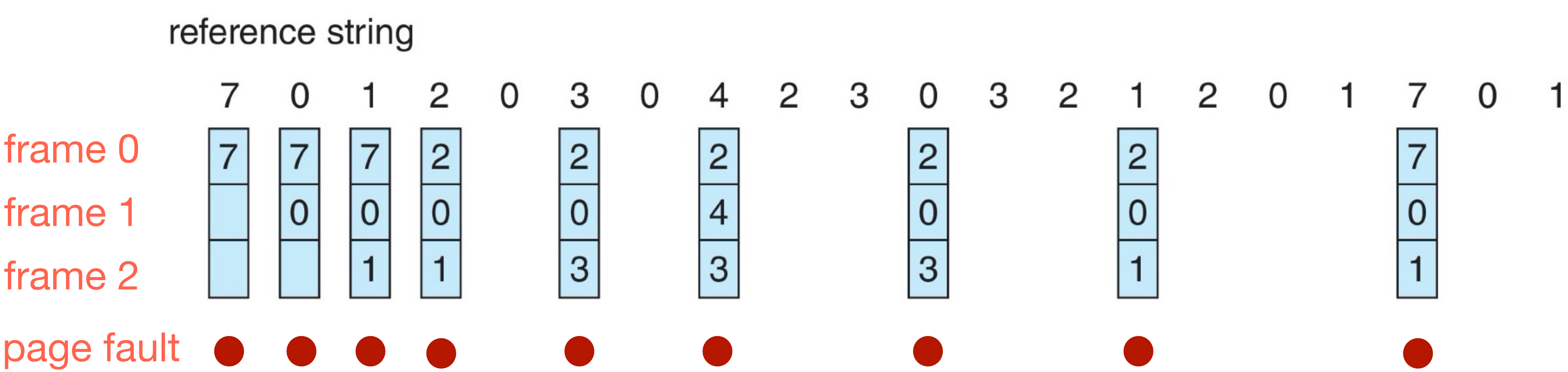
Belady's anomaly: the page fault rate may increase as the number of allocated frames increases



L. A. Belady, R. A. Nelson, and G. S. Shedler. 1969. An anomaly in space-time characteristics of certain programs running in a paging machine. Commun. ACM 12, 6 (June 1969), 349-353.

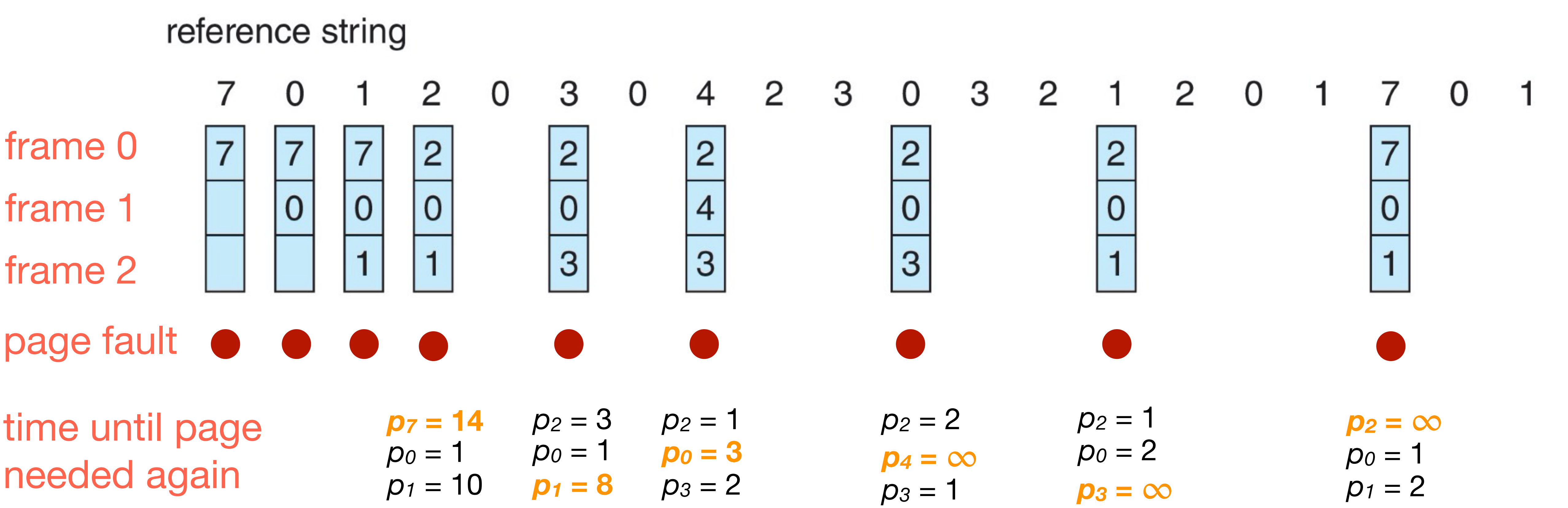
Page replacement algorithms

- optimal (OPT): look into the future and throw out the page that will be accessed farthest in the future
- it is provably optimal but unrealizable
  - how do we know future access patterns?



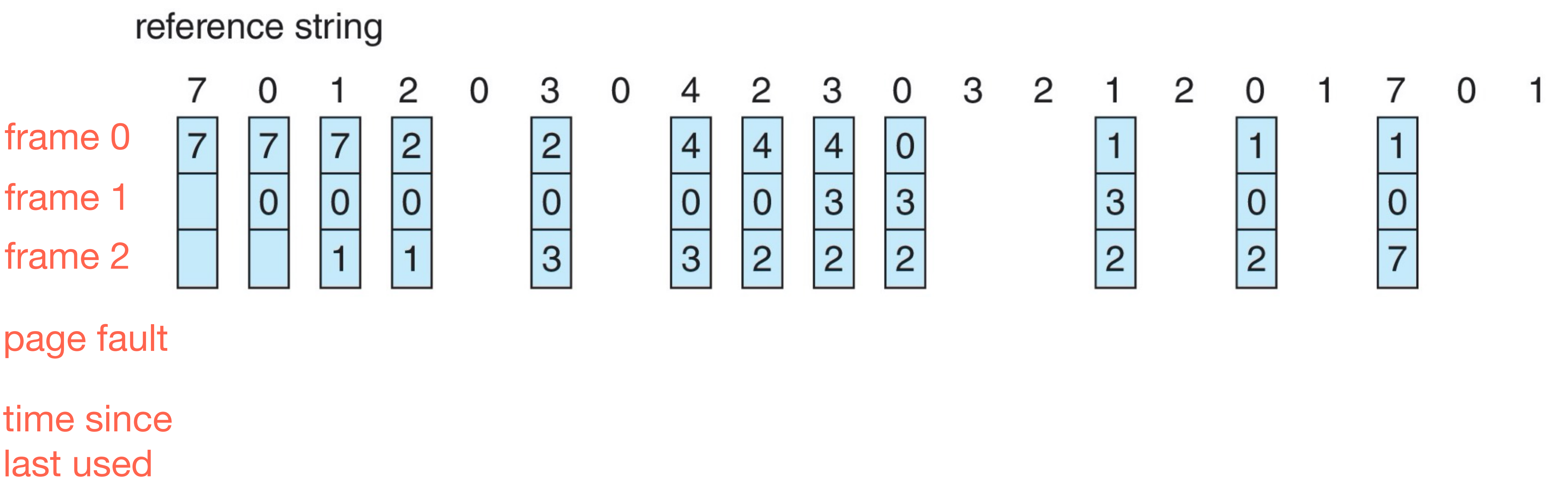
Page replacement algorithms

- optimal (OPT): look into the future and throw out the page that will be accessed farthest in the future
- it is provably optimal but unrealizable
  - how do we know future access patterns?



Page replacement algorithms

- least recently used (LRU): throw out the page that has not been used in the longest time
- it is an approximation of OPT
  - works well if the recent past is a good predictor of the future



Implementing LRU

all implementations and approximations of LRU require hardware support

Implementing LRU

all implementations and approximations of LRU require hardware support

- approach 1: keep a time stamp for each page with the time of the last access and throw out the LRU page
  - **problem**: it too expensive! the OS must log the timestamp for each memory access, and to throw out a page the OS has to look at all pages

Implementing LRU

all implementations and approximations of LRU require hardware support

- approach 1: keep a time stamp for each page with the time of the last access and throw out the LRU page
  - **problem**: it too expensive! the OS must log the timestamp for each memory access, and to throw out a page the OS has to look at all pages
- approach 2: keep a list of pages, where the front of the list is the most recently used page, and the end is the least recently used
  - on a page access, move the page to the front of the list. Doubly link the list
  - **problem**: it is still too expensive because the OS must modify 6 pointers on each memory access (in the worst case)

Approximating LRU

- maintain a reference bit for each page
  - on each access to the page, the hardware sets the reference bit to ‘1’
  - set it to ‘0’ at varying times depending on the page replacement algorithm

Approximating LRU

- maintain a reference bit for each page
  - on each access to the page, the hardware sets the reference bit to ‘1’
  - set it to ‘0’ at varying times depending on the page replacement algorithm
- additional reference bits: maintain more than 1 bit, say for example 8 bits
  - at regular intervals (e.g., every 100 milliseconds) or on each memory access, right shift the byte (**an aging mechanism**) by one bit, placing a 0 in the high order bit;
    on a page fault, the lowest numbered page is kicked out
    - it is approximate, since it does not guarantee a total order on the pages; but it is faster, since it requires setting a single bit on each memory access
  - page fault still requires a search through all the pages