Lecture 9: Virtual Memory
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF
GOTHENBURG

Reading instructions

- Chapter 3.3 to 3.6

  (extra facultative reading: 9.1-9.4.5, 9.5 to 9.7.1 from Silberschatz Operating System Concepts)

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

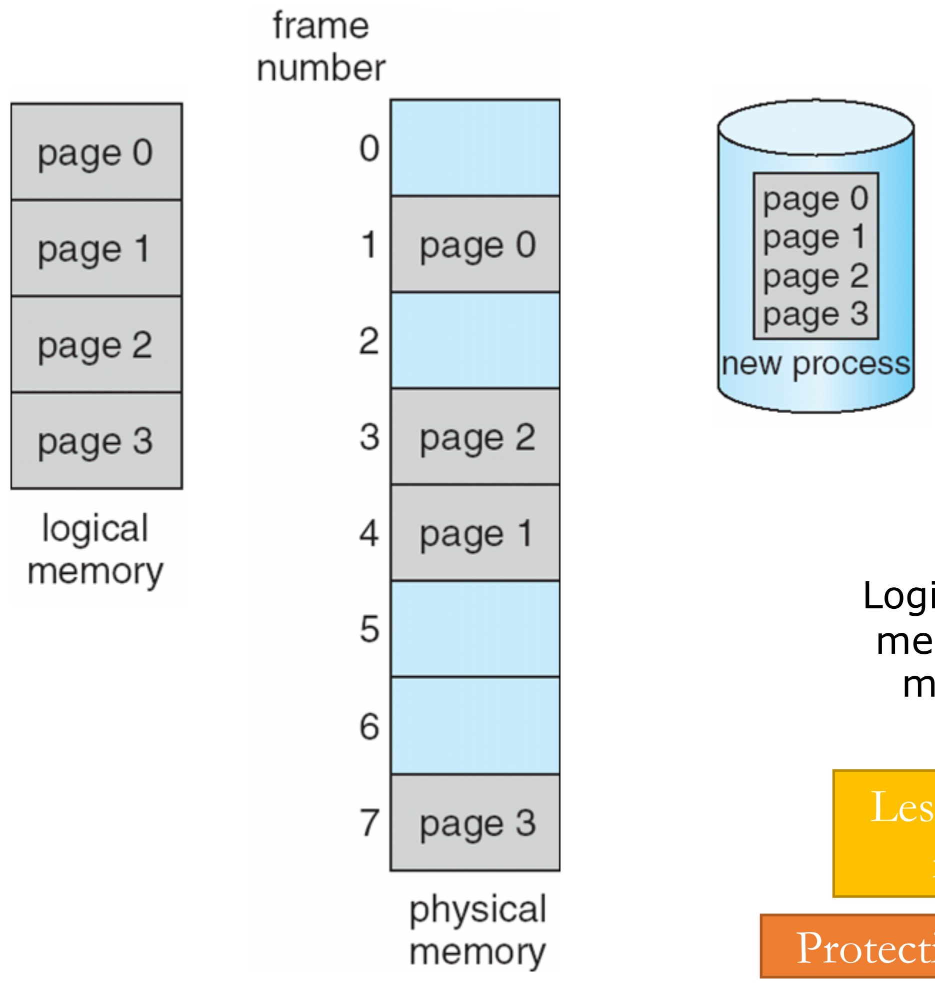
Agenda

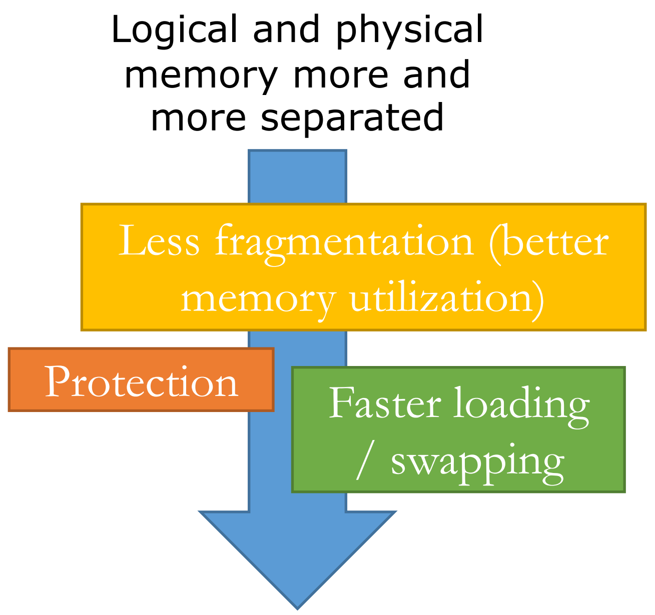
- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

Agenda

- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

… we discussed paging!



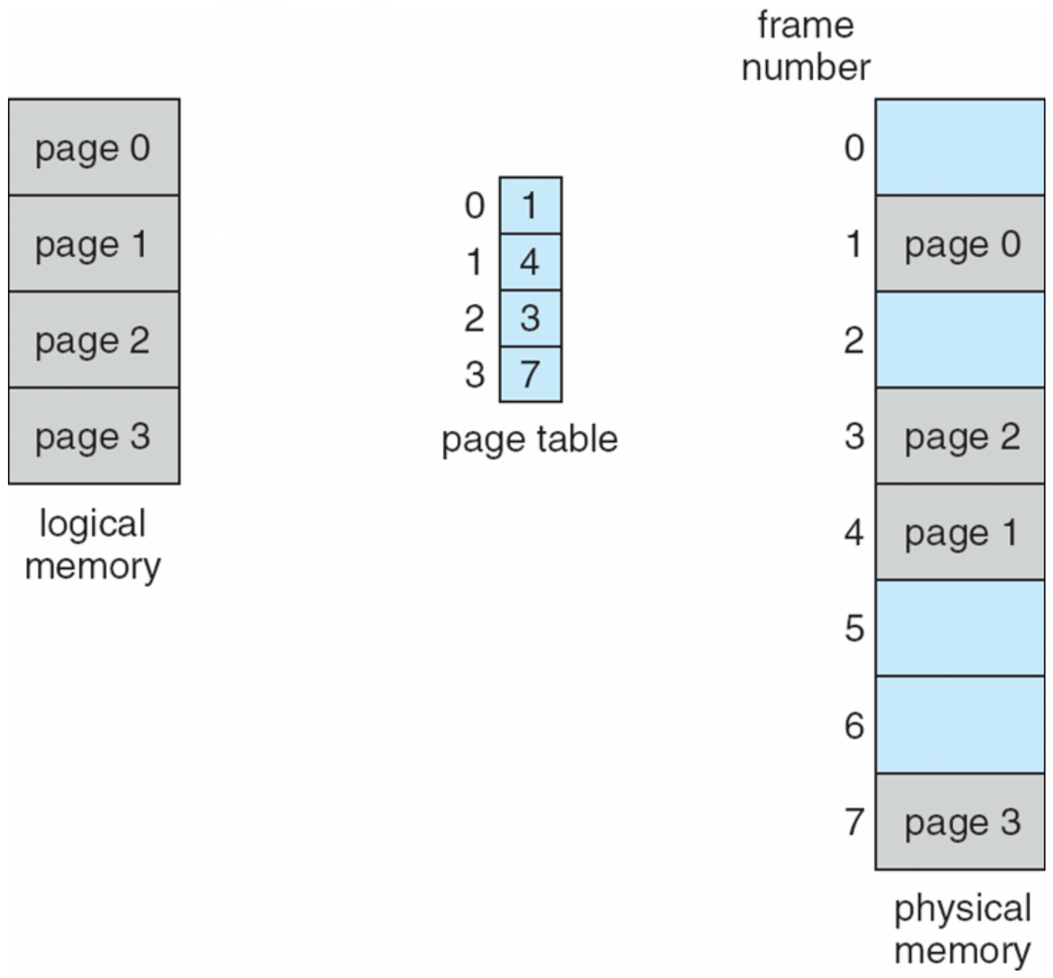


Logical and physical memory more and more separated
Less fragmentation (better memory utilization)
Protection
Faster loading / swapping

- Split logical memory in pages
- Split physical memory in frames
- Keep a program (disk) organized in pages

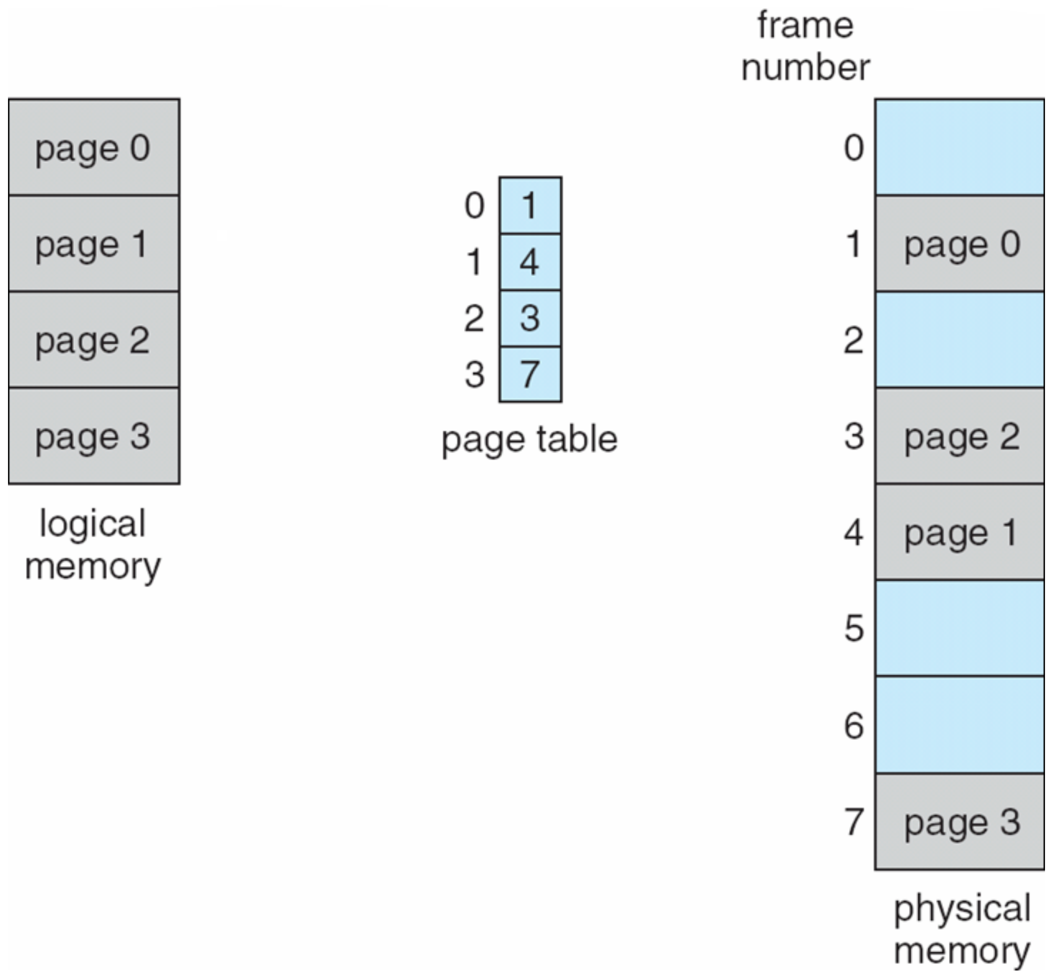
<u>**Page size = Frame size!**</u>

The intuition behind Virtual Memory



What would happen if the OS does not load page 3 in frame 7?
(does not load at all, not in a different frame)

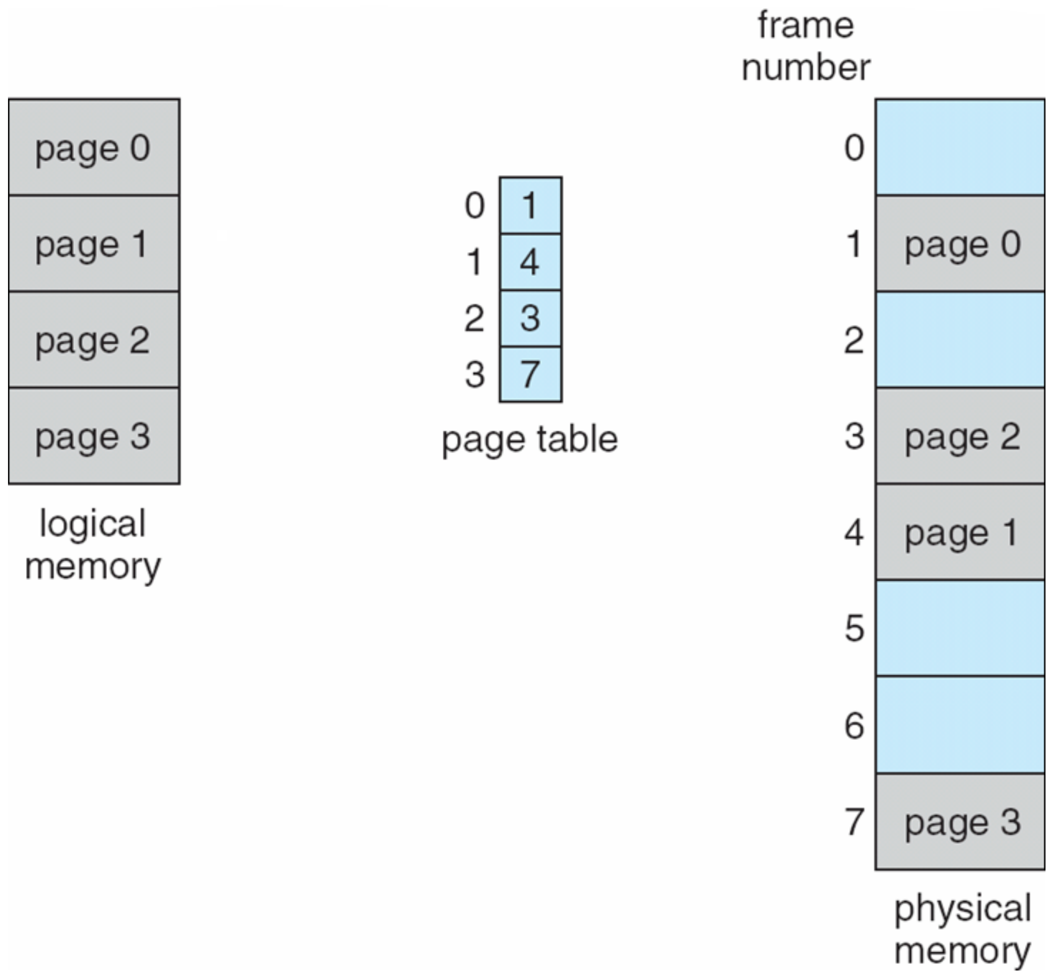
The intuition behind Virtual Memory



What would happen if the OS does not load page 3 in frame 7?
(does not load at all, not in a different frame)

When would that be a problem?

The intuition behind Virtual Memory

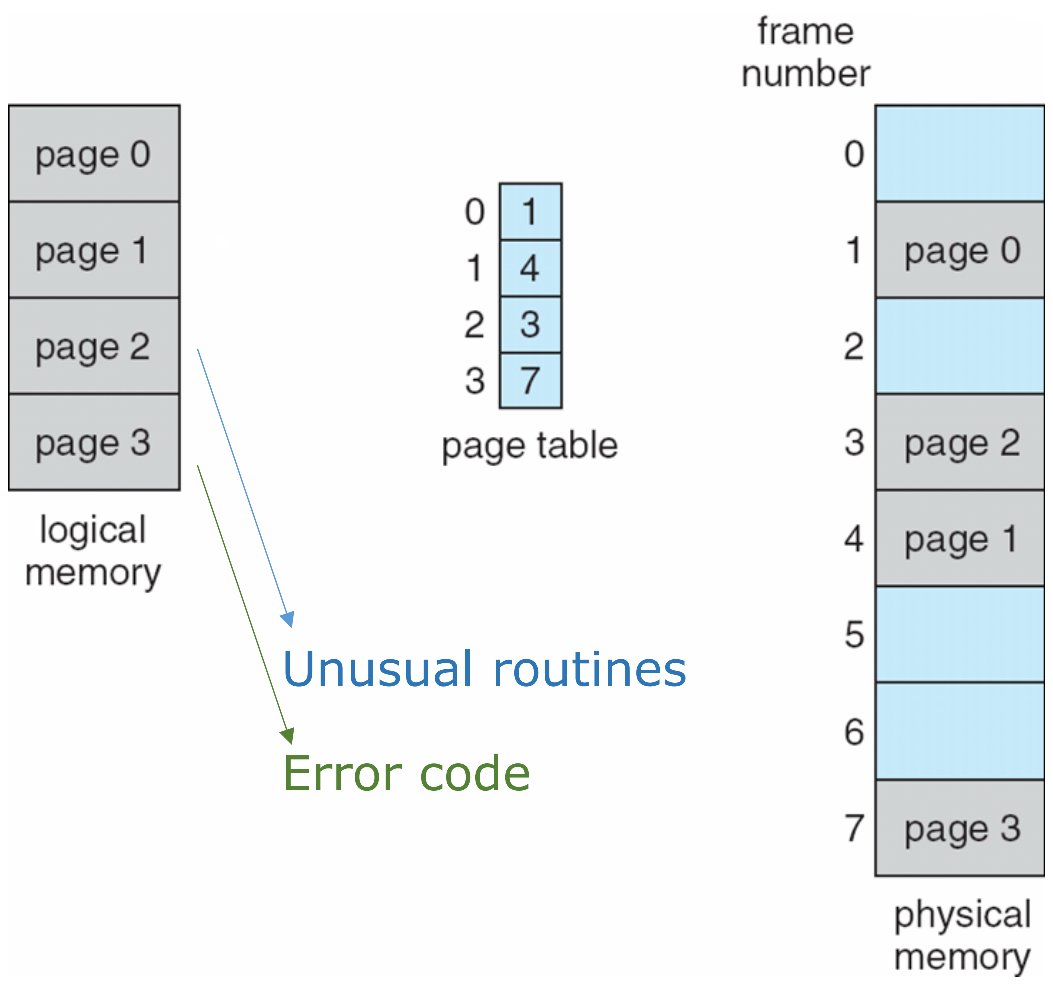


What would happen if the OS does not load page 3 in frame 7?
(does not load at all, not in a different frame)

When would that be a problem?

Let's wait until the process tries to access page 3… If it does, then we will load it into frame 7!

Virtual Memory in a Nutshell (i)



- Code needs to be in memory to execute, but entire program rarely used
- Entire program code not needed at same time
- If we manage to execute partially-loaded programs:
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual Memory in a Nutshell (ii)

- What are the costs / trade-offs / complications when executing partially-loaded programs:

  - Need a mechanism to check if a page is actually in the frame or not
  - Because of this mechanism, sometimes accessing a page might require more time than expected
  - … and more …

Agenda

- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

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
  - Less memory needed
  - More users

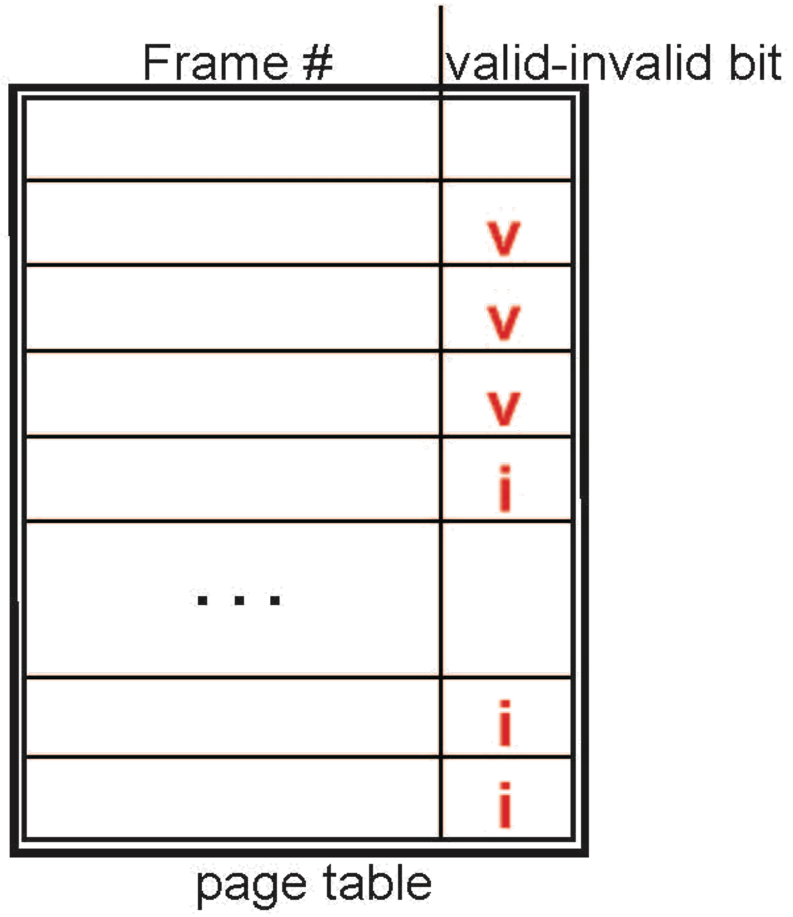
- Page is needed ⇒ reference to it
  - not-in-memory ⇒ bring to memory
- Lazy swapper / pager – never swaps a page into memory unless page will be needed

Basic Concepts

- Pager brings in only those pages into memory
- 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

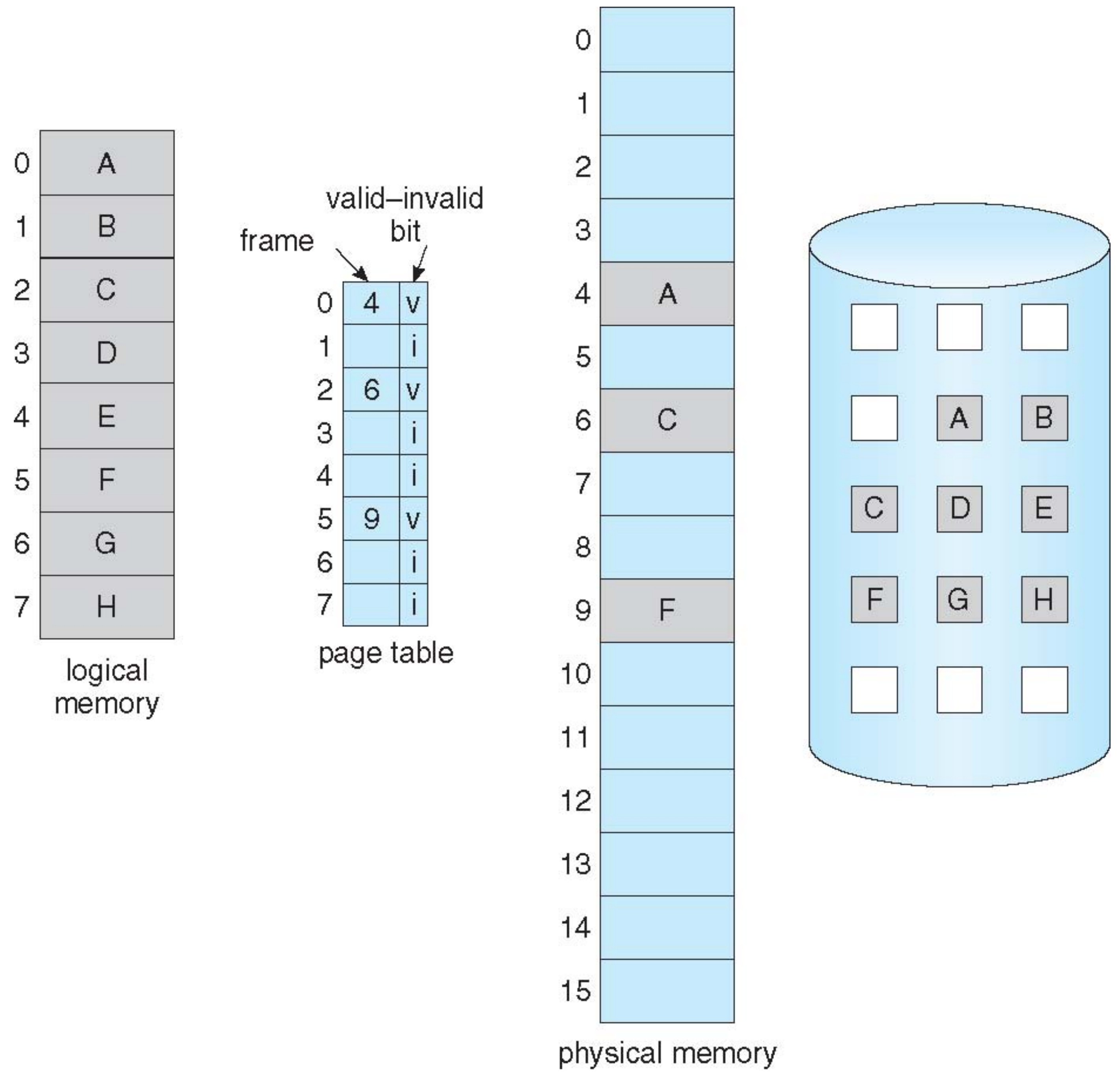
Valid-Invalid Bit

- 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
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is i ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



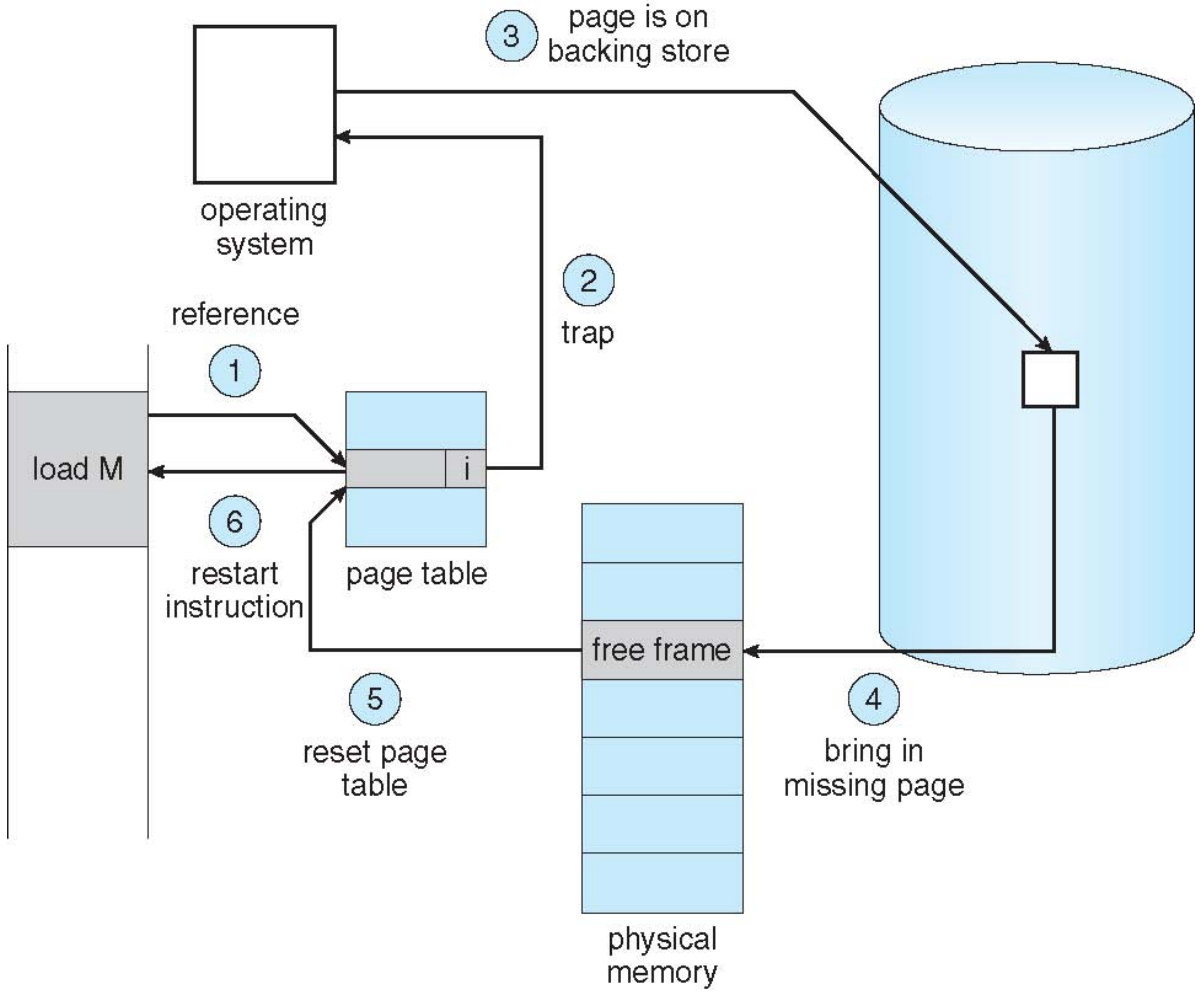
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. 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

Performance of Demand Paging

- 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:
    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
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 (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – 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)

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

Agenda

- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

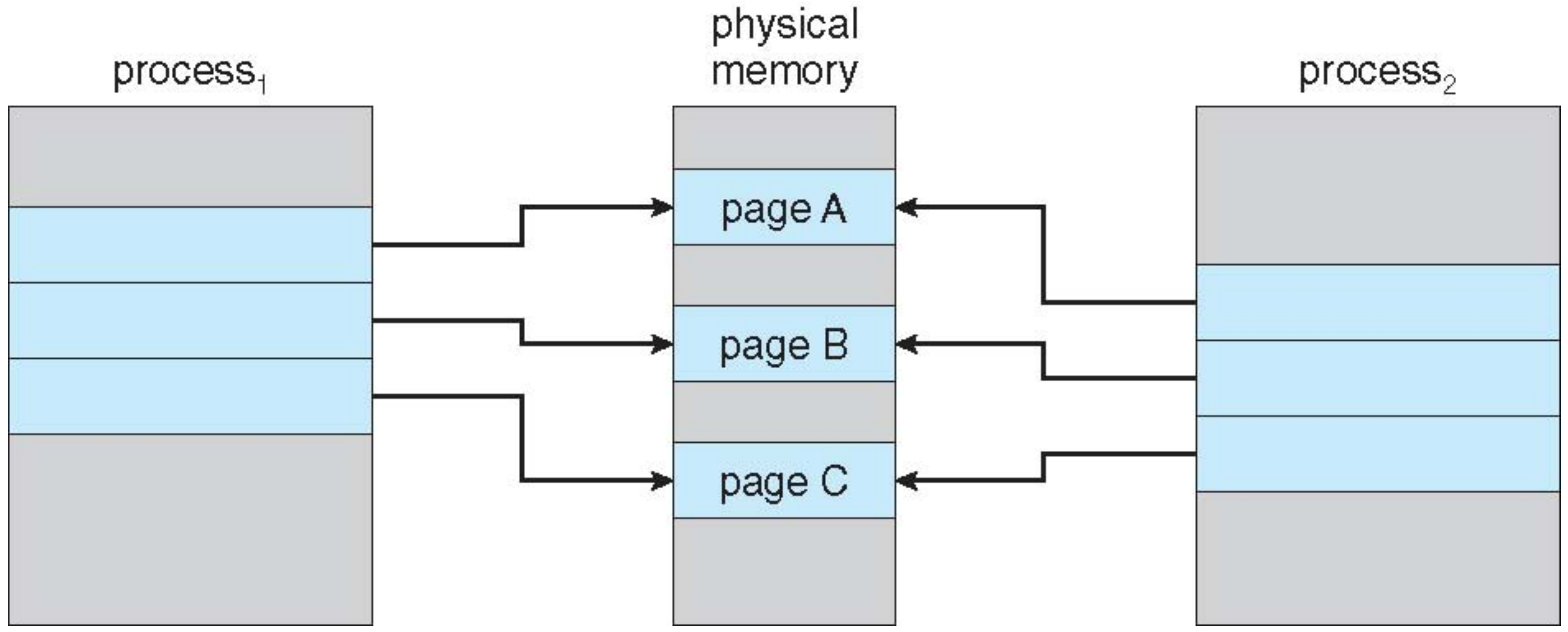
Copy-on-Write

- **Example of optimization enabled by separation of Logical from Physical memory and by demand paging**

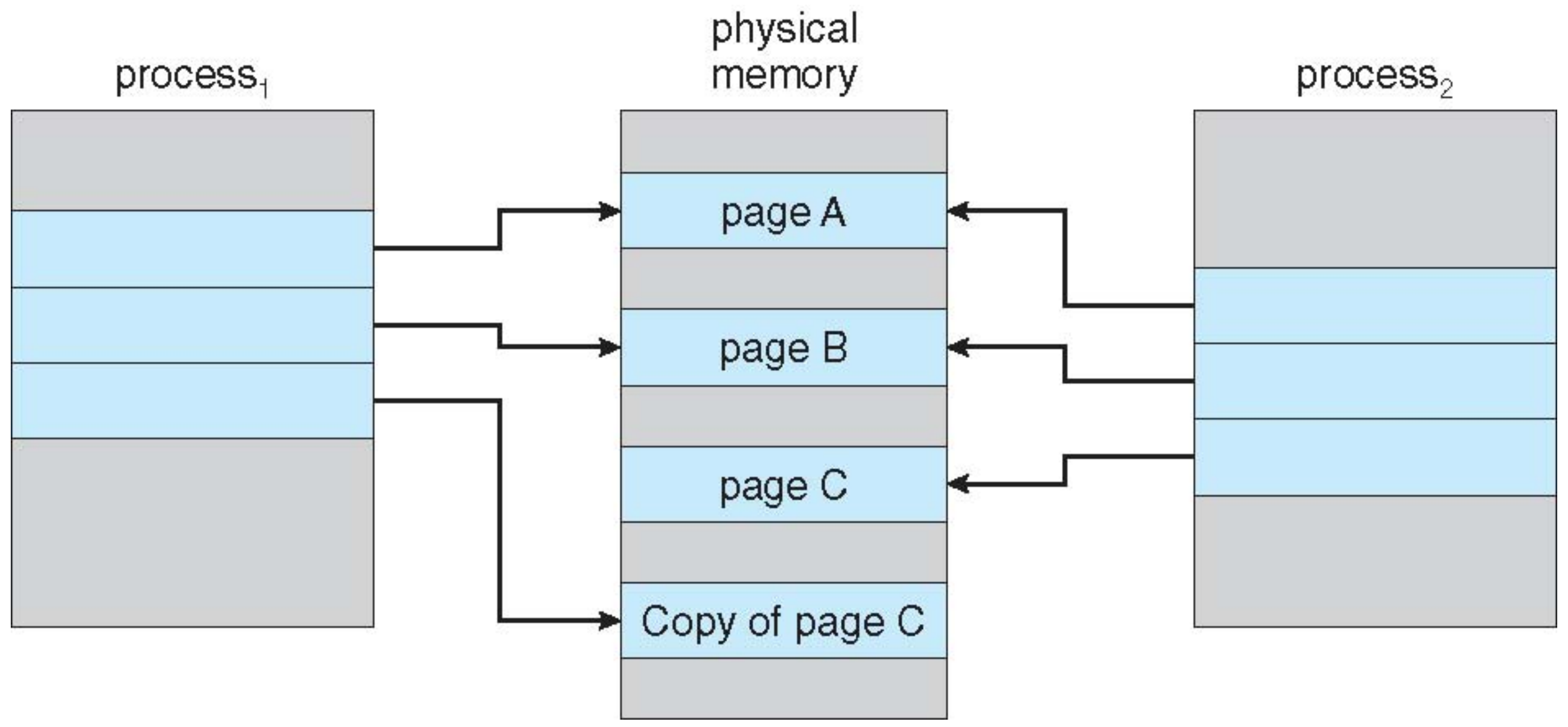
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
    - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied

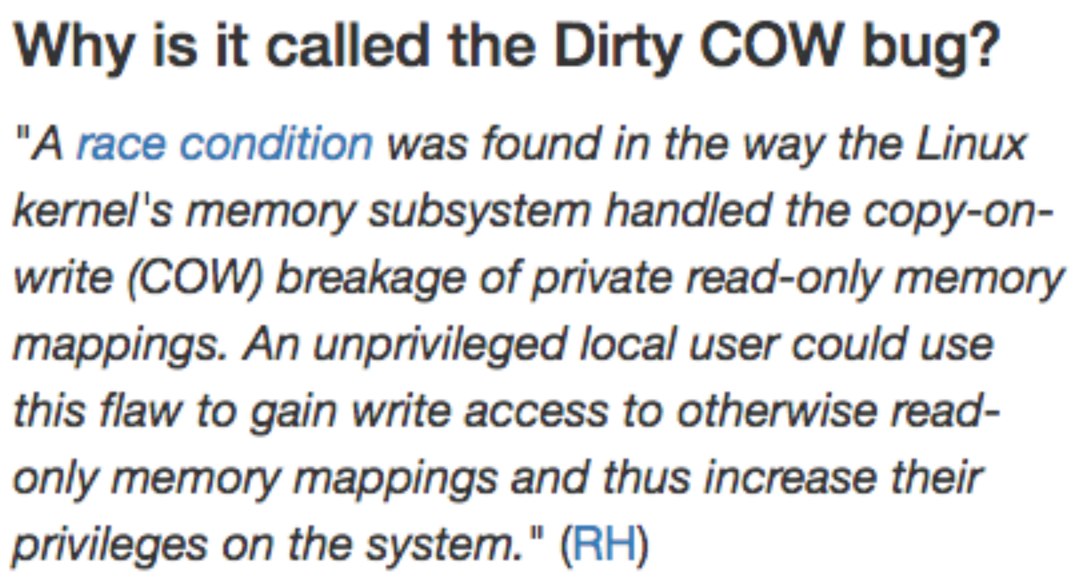
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
    - Designed to have child call `exec()`
    - Very efficient

Before Process 1 Modifies Page C



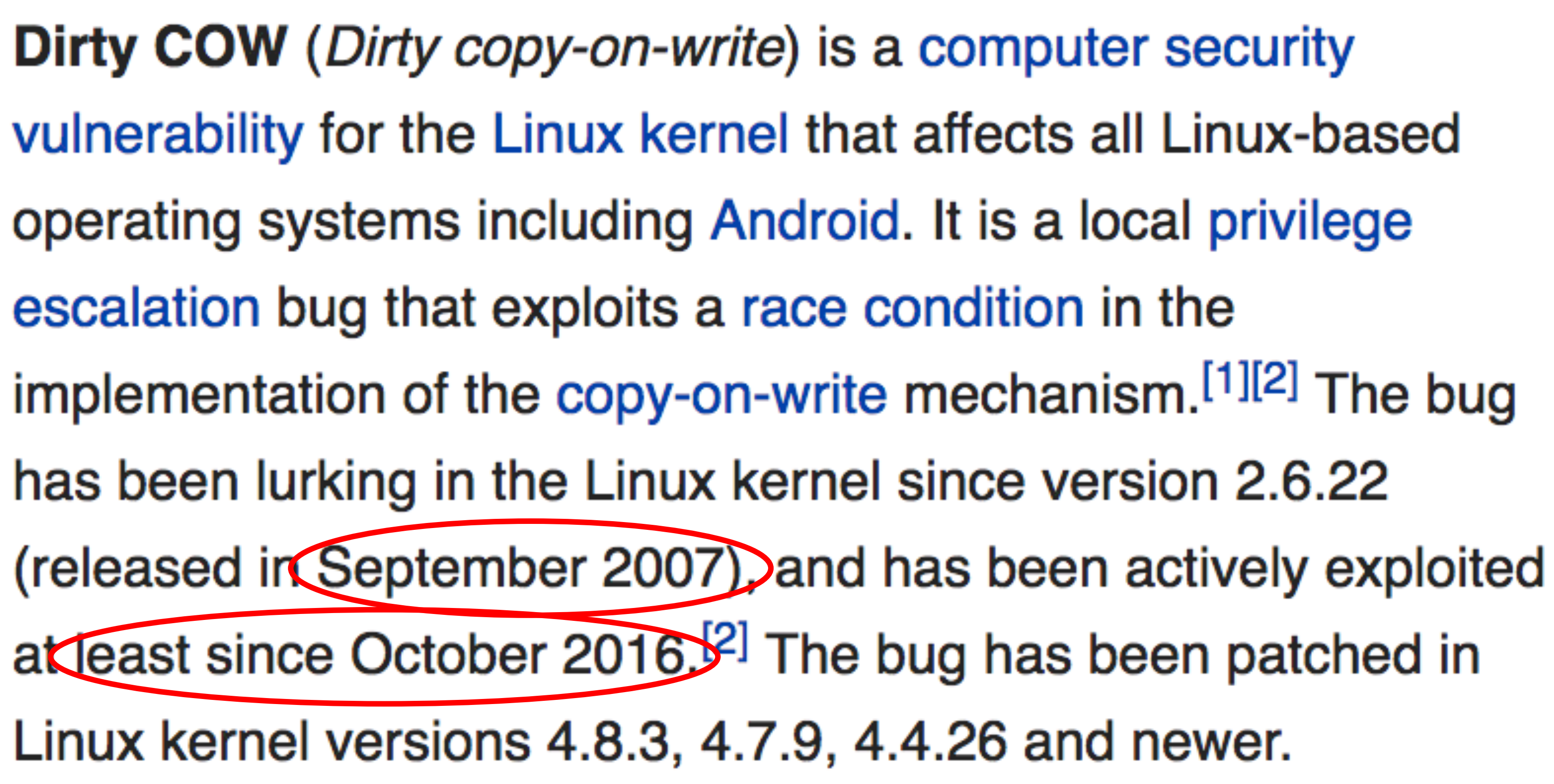
After Process 1 Modifies Page C





Why is it called the Dirty COW bug?

"A race condition was found in the way the Linux kernel's memory subsystem handled the copy-on-write (COW) breakage of private read-only memory mappings. An unprivileged local user could use this flaw to gain write access to otherwise read-only memory mappings and thus increase their privileges on the system." (RH)

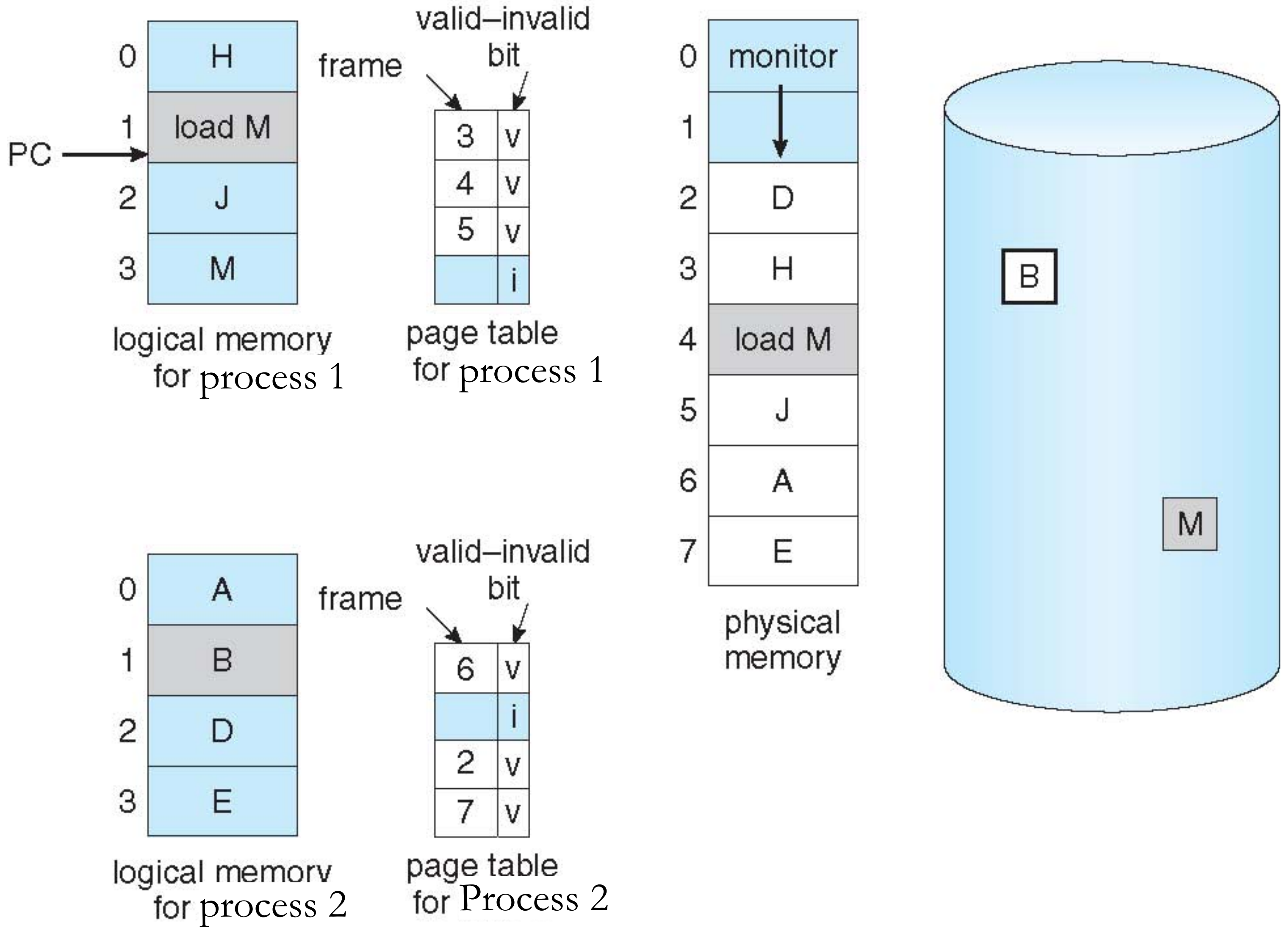


Dirty COW (*Dirty copy-on-write*) is a computer security vulnerability for the Linux kernel that affects all Linux-based operating systems including Android. It is a local privilege escalation bug that exploits a race condition in the implementation of the copy-on-write mechanism.[1][2] The bug has been lurking in the Linux kernel since version 2.6.22 (released in September 2007), and has been actively exploited at least since October 2016.[2] The bug has been patched in Linux kernel versions 4.8.3, 4.7.9, 4.4.26 and newer.

Agenda

- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

Suppose we end up in a situation like the following one



Process 1 needs a free frame (but 0 available!)

- *What to do now?*
- *Can we replace an existing page?*
- *Should it be from process 1 or can it also be from process 2?*
- *Is there any page that is faster to swap out than others?*
- *Is there any page that has lower probability of being needed later on?*
- *Could have we prevented this situation?*
- *How many page should we have allocate to the OS and to processes 1 and 2 to prevent this?*
- …

Page and Frame Replacement Algorithms

- *What to do now?*
- *Can we replace an existing page?*
- *Should it be from process 1 or can it also be from process 2?*
- *Is there any page that is faster to swap out than others?*
- *Is there any page that has lower probability of being needed later on?*
- *Could have we prevented this situation?*
- *How many page should we have allocate to the OS and to processes 1 and 2 to prevent this?*
- …

Frame-allocation algorithm
determines how many frames to give each process

Page-replacement algorithm
Which page to replace? Want lowest page-fault rate on both first access and re-access

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – 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

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

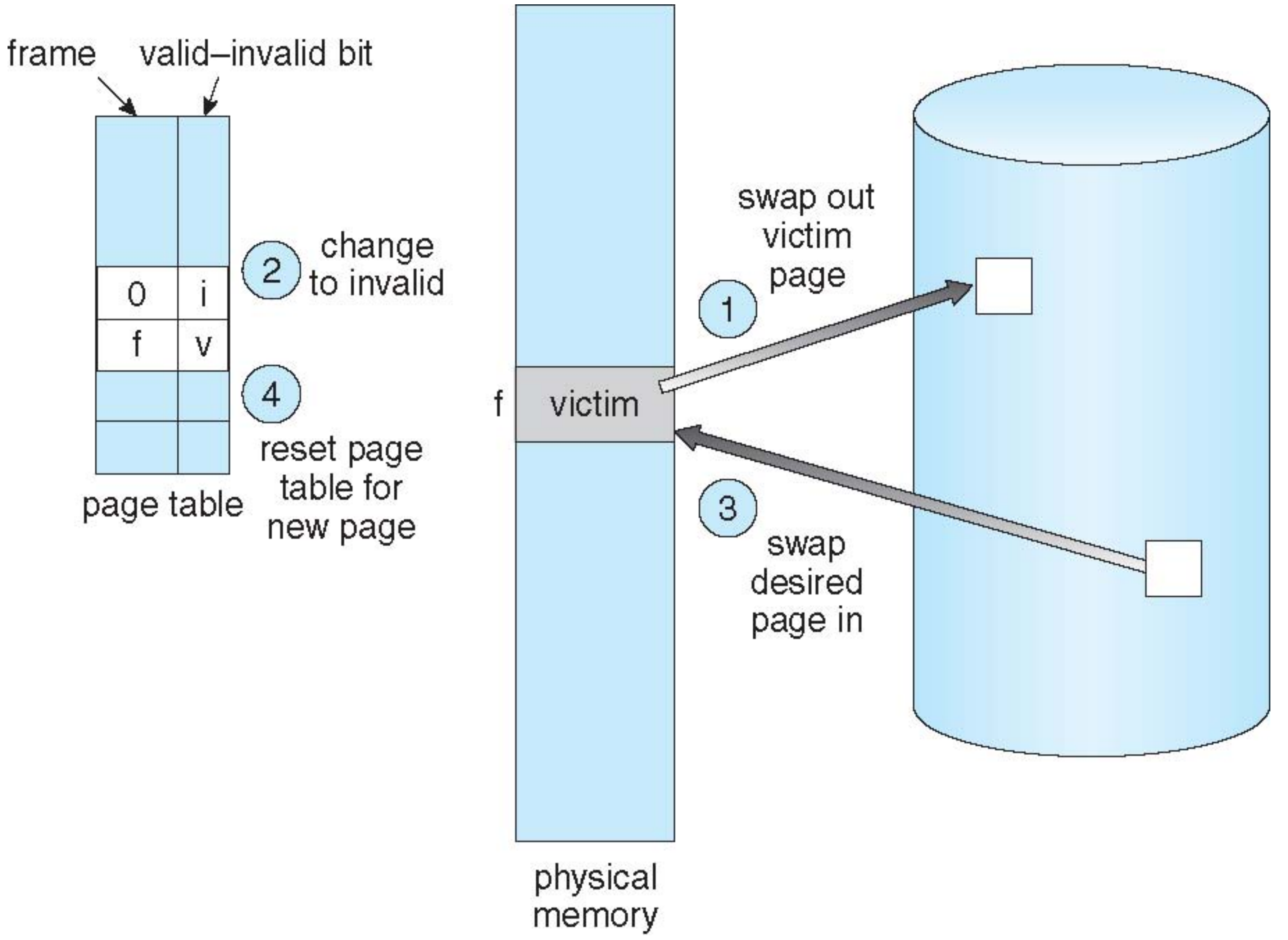
Note now potentially 2 page transfers for page fault – increasing EAT

Page replacement algorithms

- We are going to take into account:

- FIFO (First-In-First-Out)
- Optimal algorithm
- Least Recently Used (LRU)
- Approximated LRU

Page Replacement



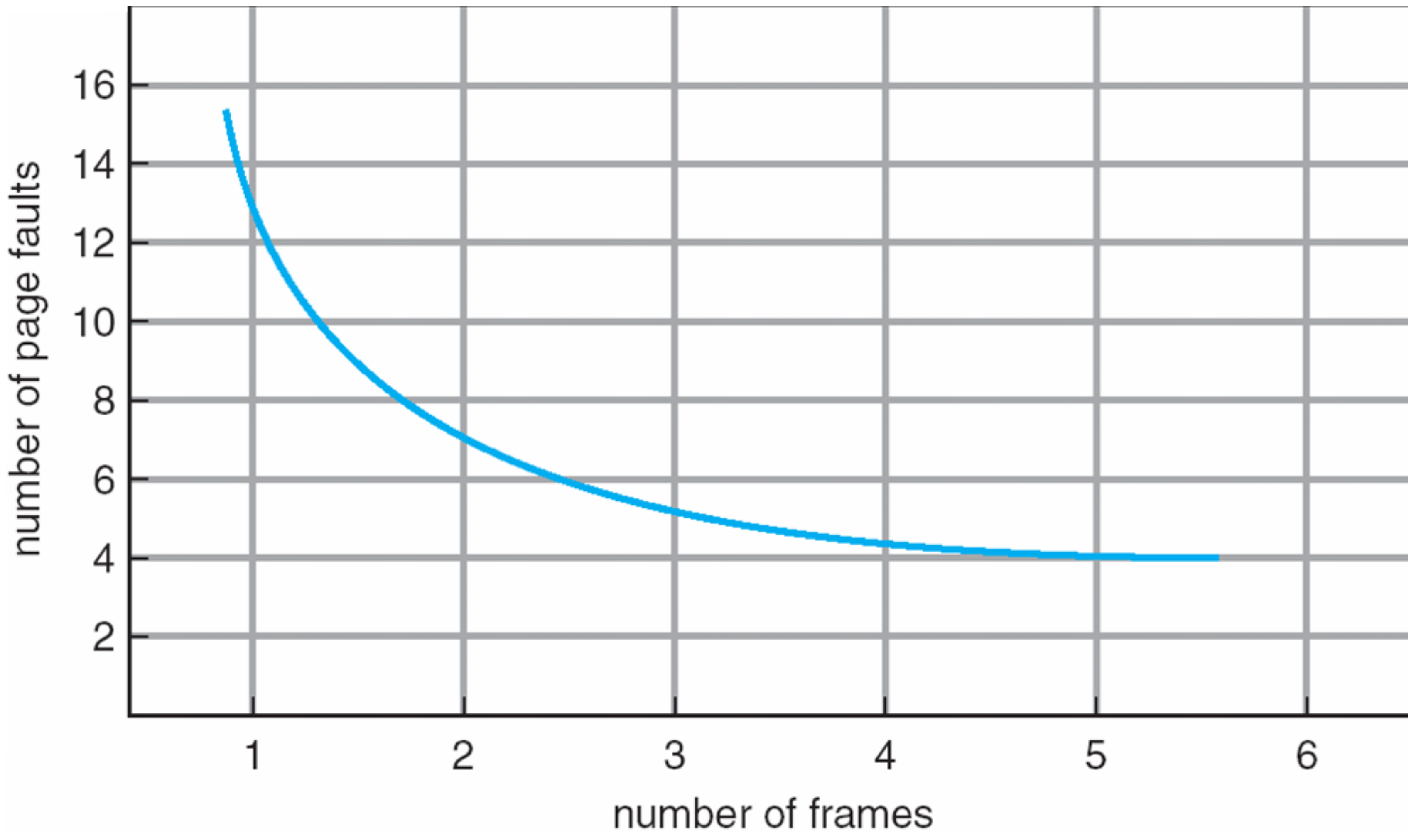
How to evaluate a page replacement algorithm?

- **Run** 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

What do we expect?
(Graph of Page Faults Versus The Number of Frames)



First-In-First-Out (FIFO) Algorithm

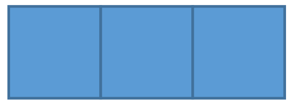
- If all frames are used, free the one containing the page loaded more time ago
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

reference string

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

Page needed: 7

Page faults: 0

Status of frames: 

Is it there? NO

Do we have a free frame? YES

If YES, load the page

+1 page fault

First-In-First-Out (FIFO) Algorithm

- If all frames are used, free the one containing the page loaded more time ago
- 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)

reference string

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

Page needed: 0

Page faults: 1

Status of frames:

7		

Is it there? NO

Do we have a free frame? YES

If YES, load the page

+1 page fault

First-In-First-Out (FIFO) Algorithm

- If all frames are used, free the one containing the page loaded more time ago
- 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)

reference string

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

Page needed: 1

Page faults: 2

Status of frames:

7	0	

Is it there? NO

Do we have a free frame? YES

If YES, load the page

+1 page fault

First-In-First-Out (FIFO) Algorithm

- If all frames are used, free the one containing the page loaded more time ago
- 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)

reference string

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

Page needed: 2

Page faults: 3

Status of frames:

7	0	1

Is it there? NO

Do we have a free frame? NO

If NO, substitute the page loaded more time ago (7)

+1 page fault

First-In-First-Out (FIFO) Algorithm

- If all frames are used, free the one containing the page loaded more time ago
- 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)

reference string

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

Page needed: 0

Page faults: 4

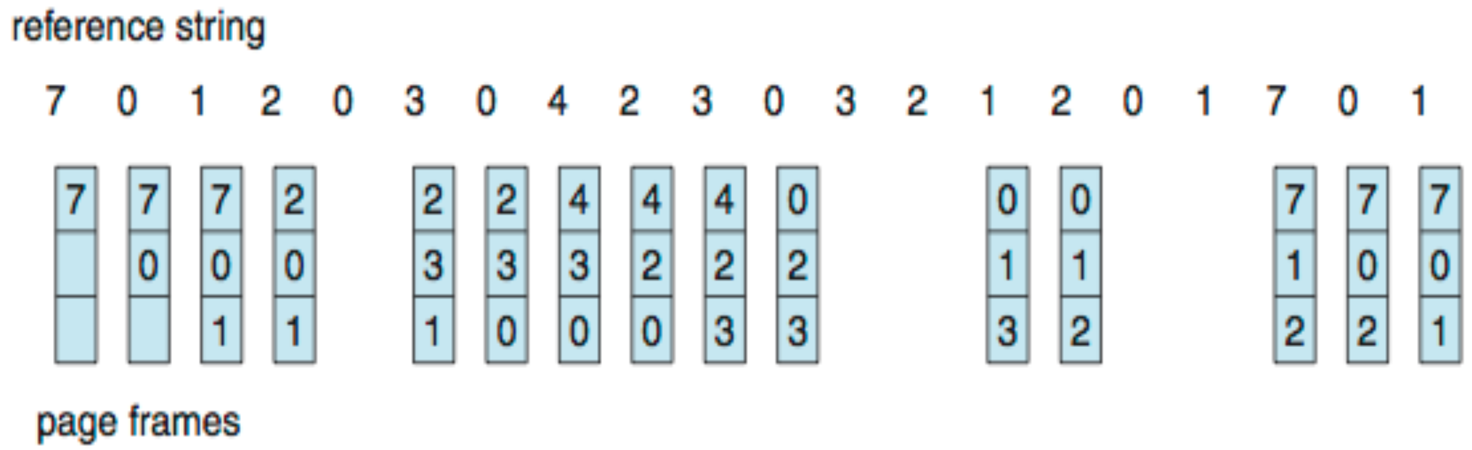
Status of frames:

2	0	1

Is it there? YES

First-In-First-Out (FIFO) Algorithm

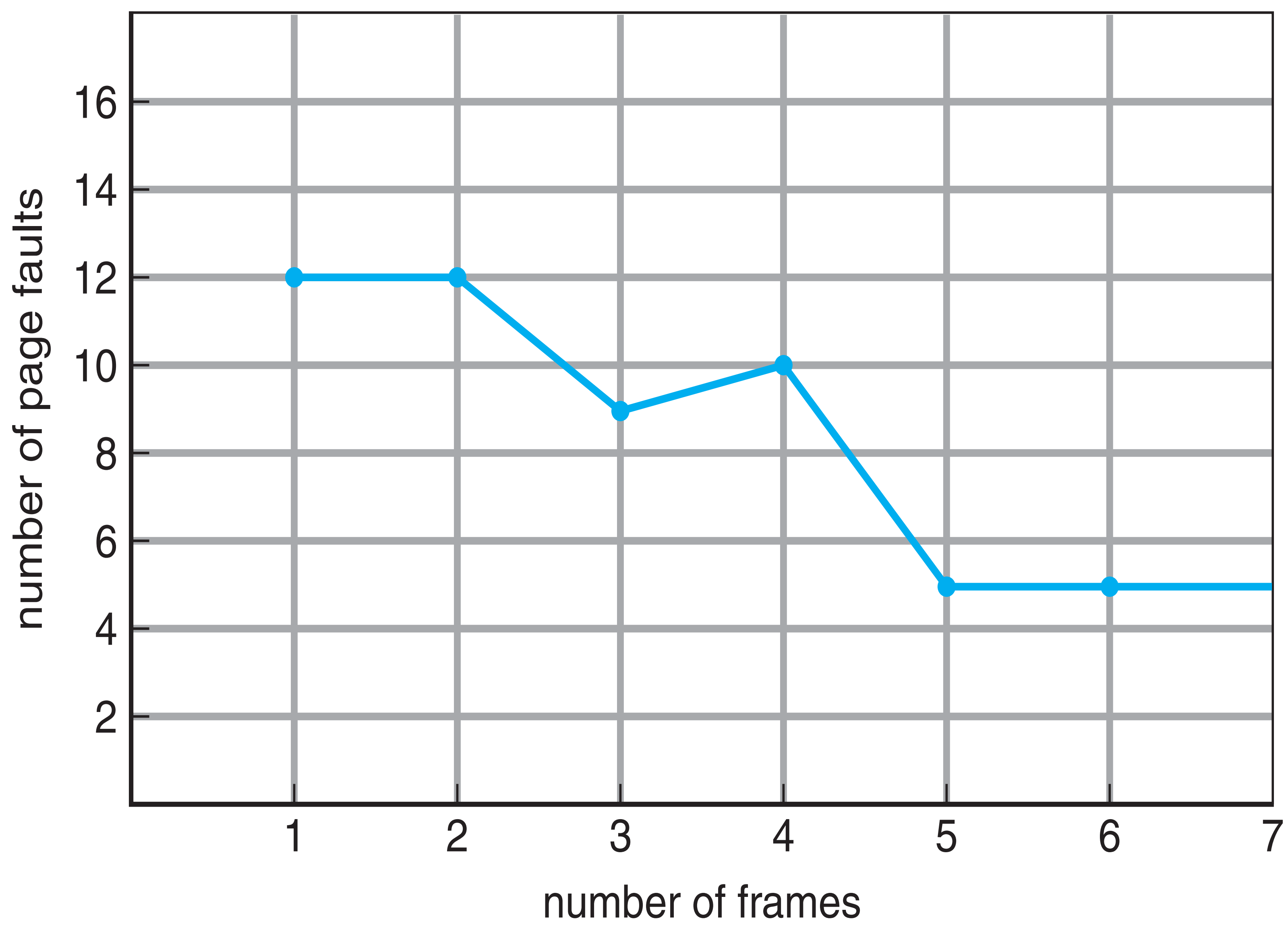
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

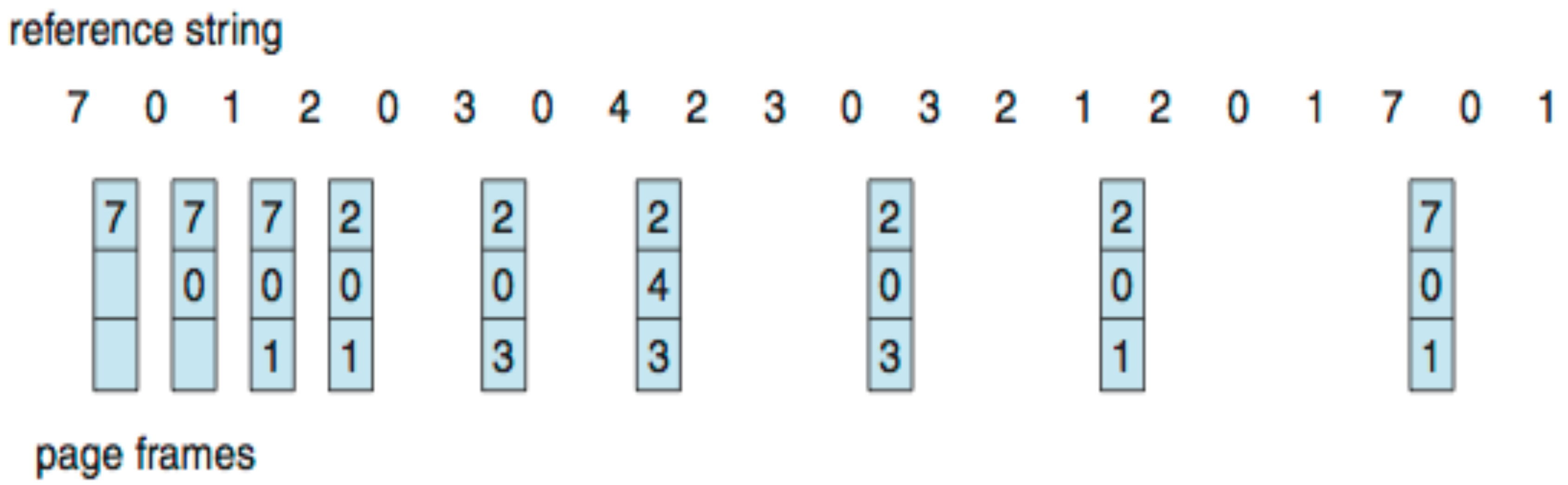
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
- Adding more frames can cause more page faults!
  - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



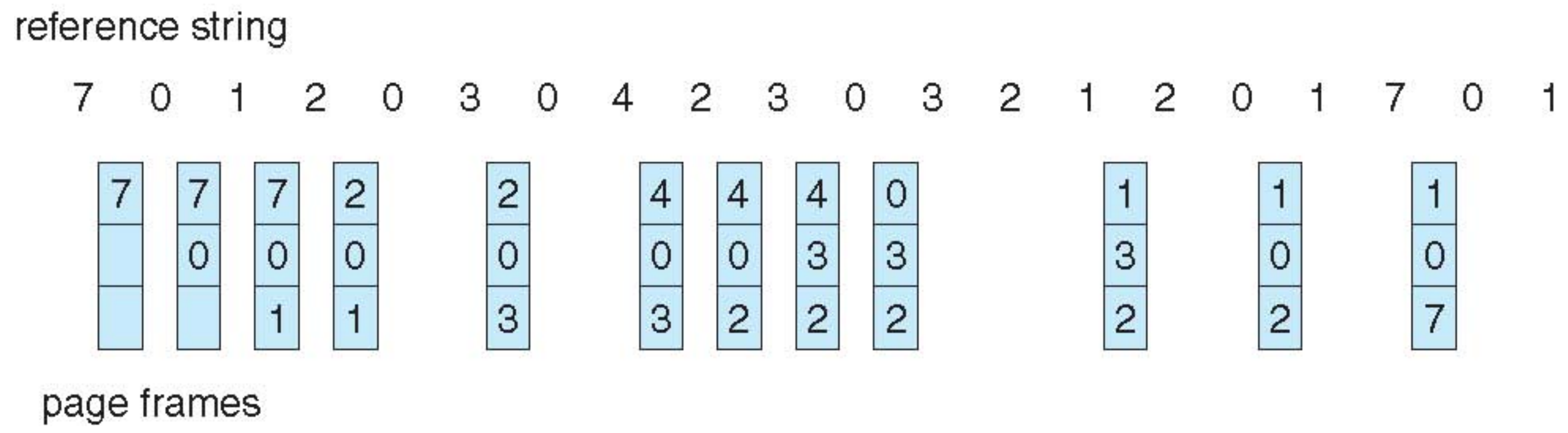
Optimal Algorithm

- 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



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



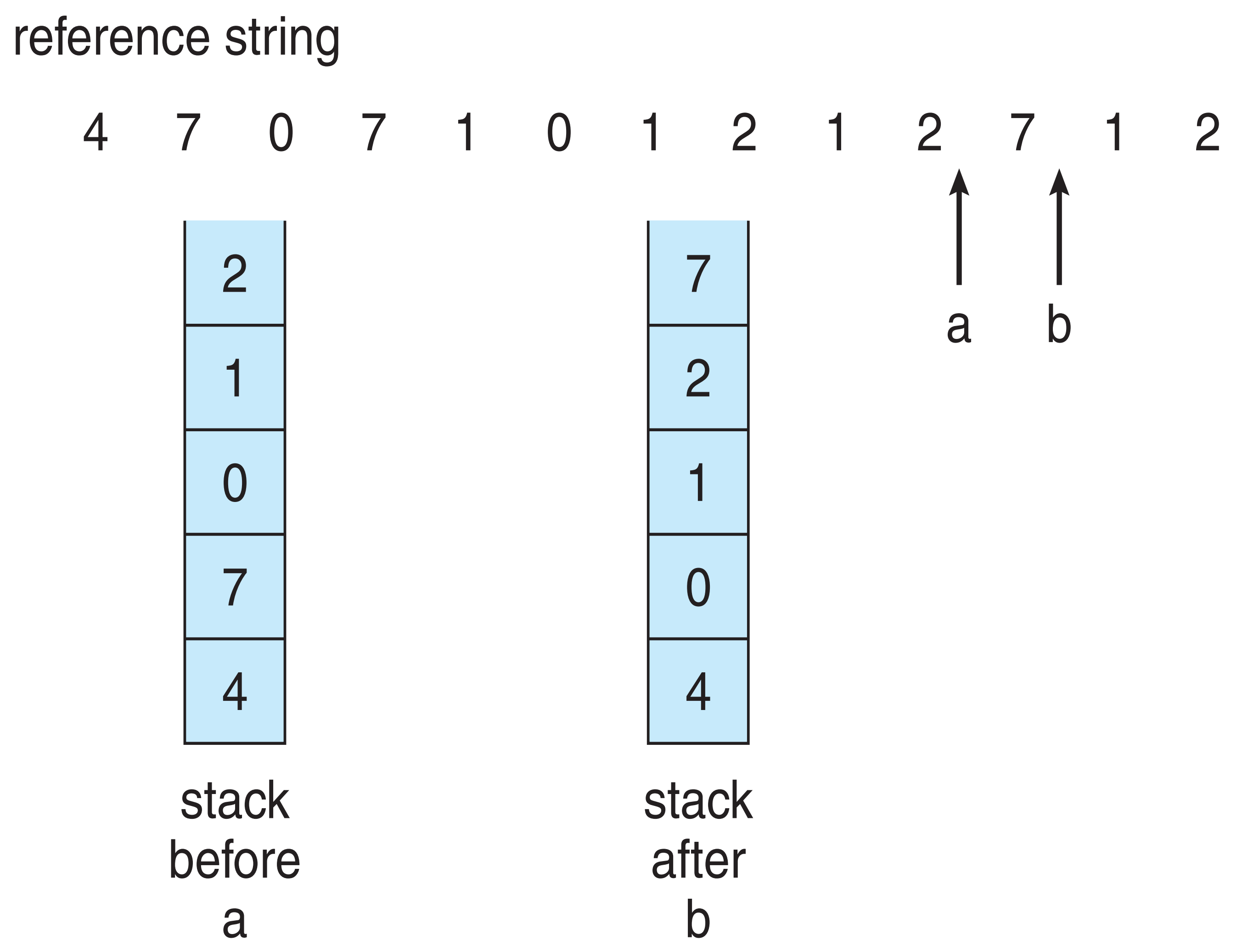
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- 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

- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced → move it to the top
  - Each update more expensive
  - No search for replacement (page to replace at the bottom of the stack)

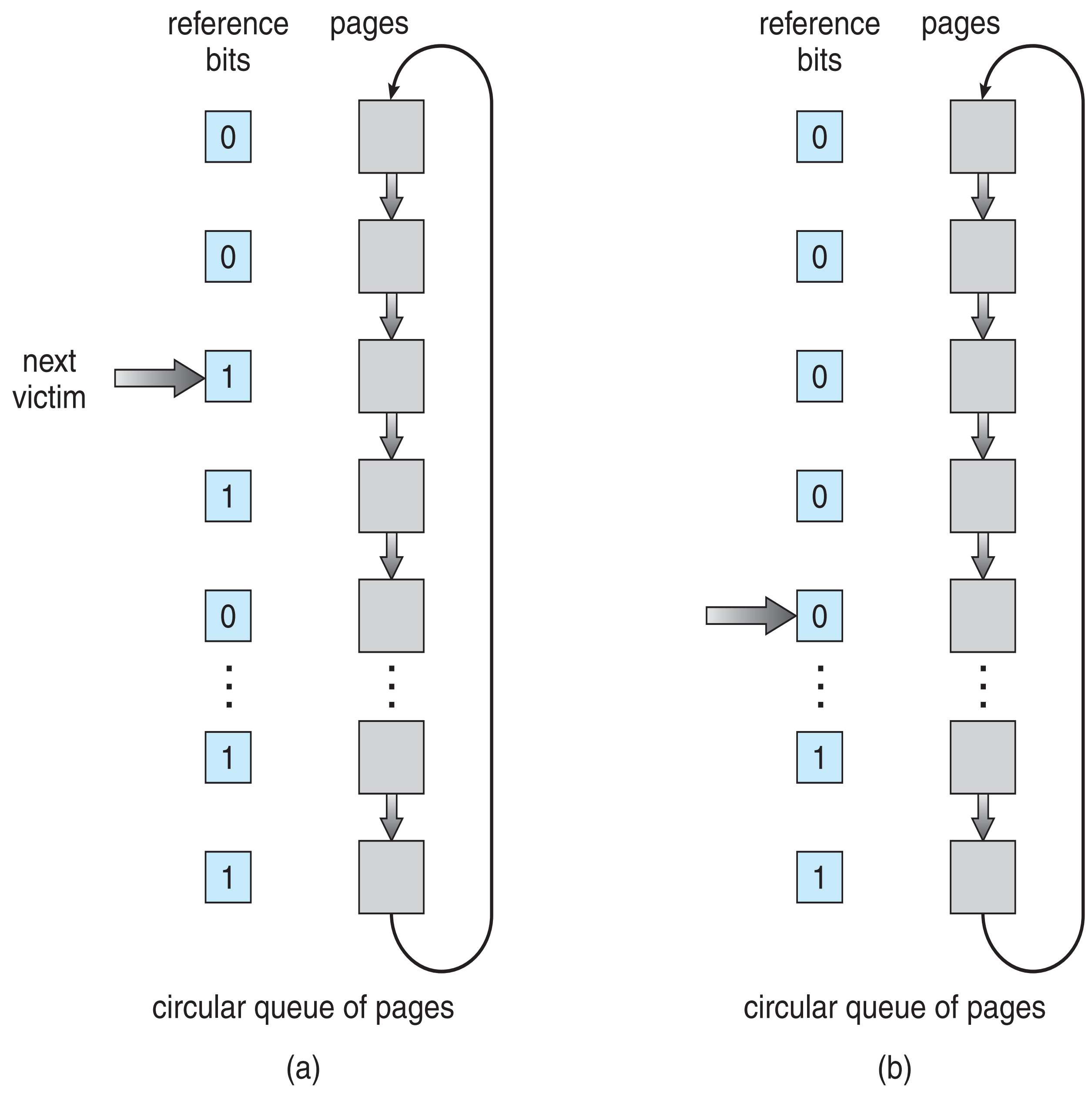
Use Of A Stack to Record Most Recent Page References



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **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)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - 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

Second-Chance (clock) Page-Replacement Algorithm



Agenda

- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

Page and Frame Replacement Algorithms

- *What to do now?*
- *Can we replace an existing page?*
- *Should it be from process 1 or can it also be from process 2?*
- *Is there any page that is faster to swap out than others?*
- *Is there any page that has lower probability of being needed later on?*
- *Could have we prevented this situation?*
- *How many page should we have allocate to the OS and to processes 1 and 2 to prevent this?*
- …

Frame-allocation algorithm
determines how many frames to give each process

Page-replacement algorithm
Which page to replace? Want lowest page-fault rate on both first access and re-access

Allocation of Frames

- Each process needs ***minimum*** number of frames
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

– s_i = size of process p_i

– $S = \sum s_i$

– m = total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process $\boldsymbol{P_i}$ generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Agenda

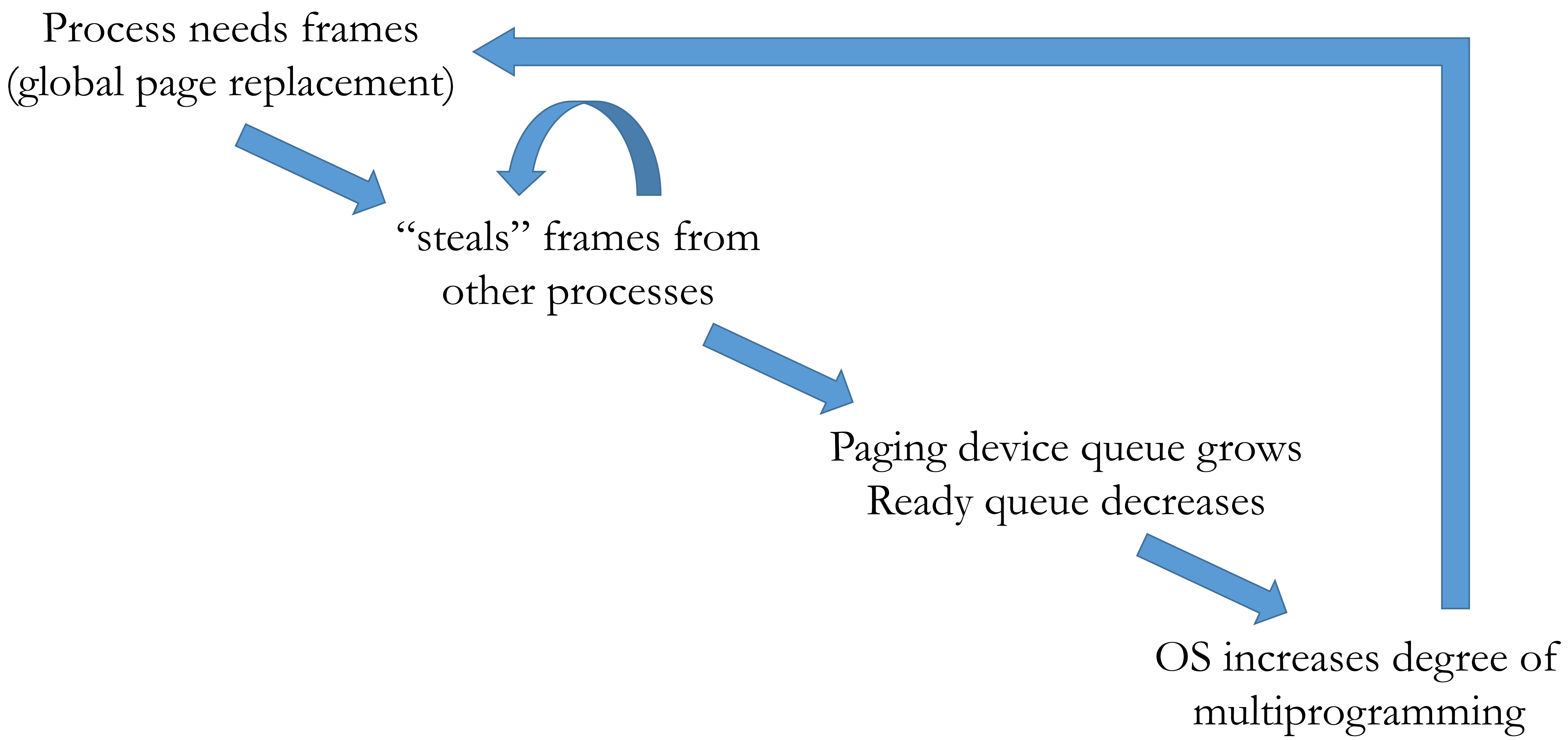
- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

Thrashing

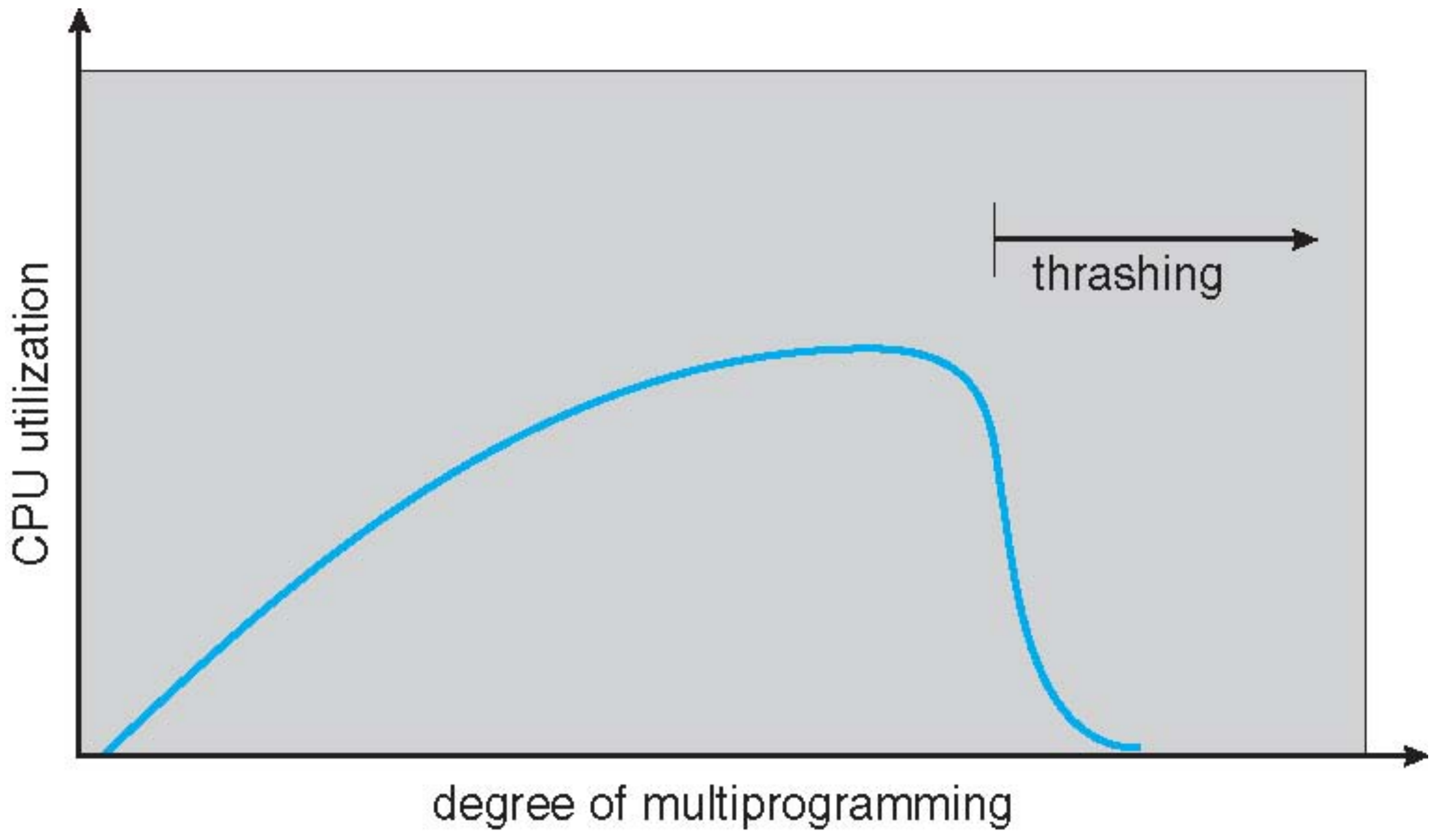
- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (cont.)


Process needs frames (global page replacement)
"steals" frames from other processes
Paging device queue grows Ready queue decreases
OS increases degree of multiprogramming

Thrashing (Cont.)



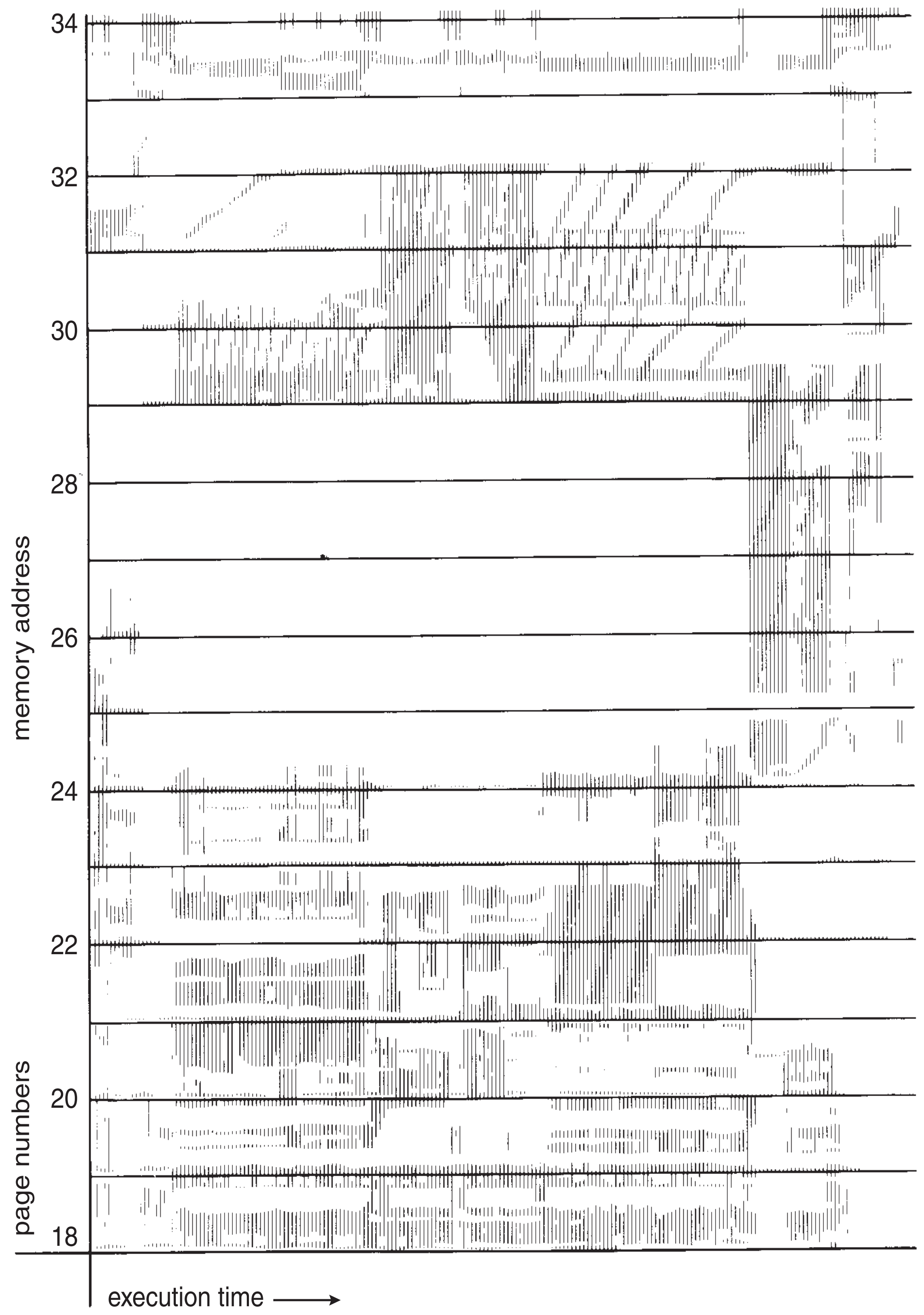
Sometimes the OS needs to reduce degree of parallelism to increase CPU utilization

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

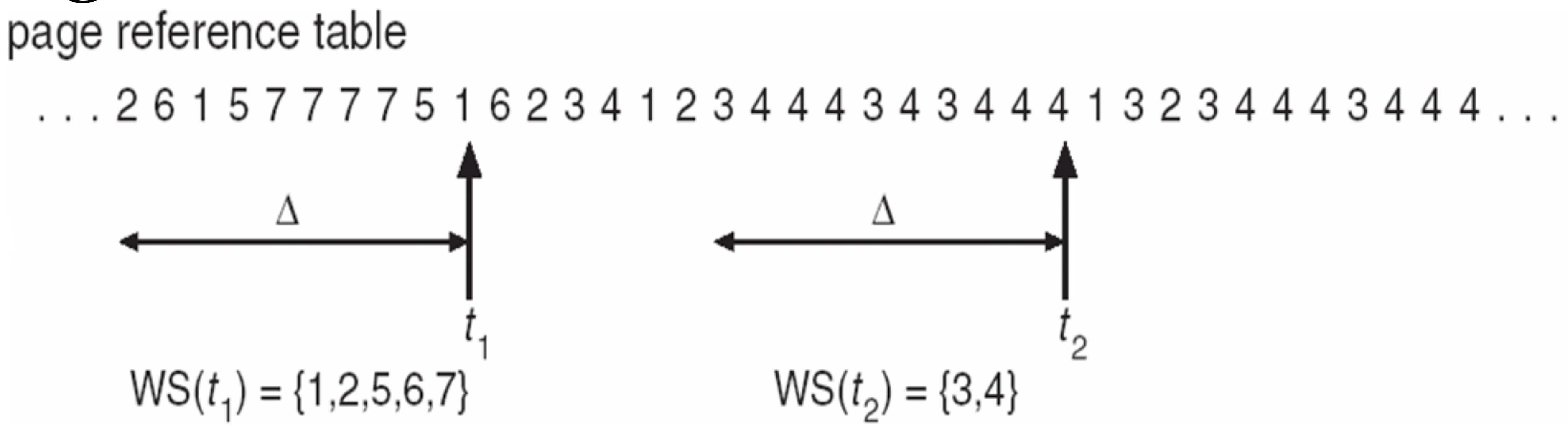
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



How could the OS avoid trashing?

Working-Set Model

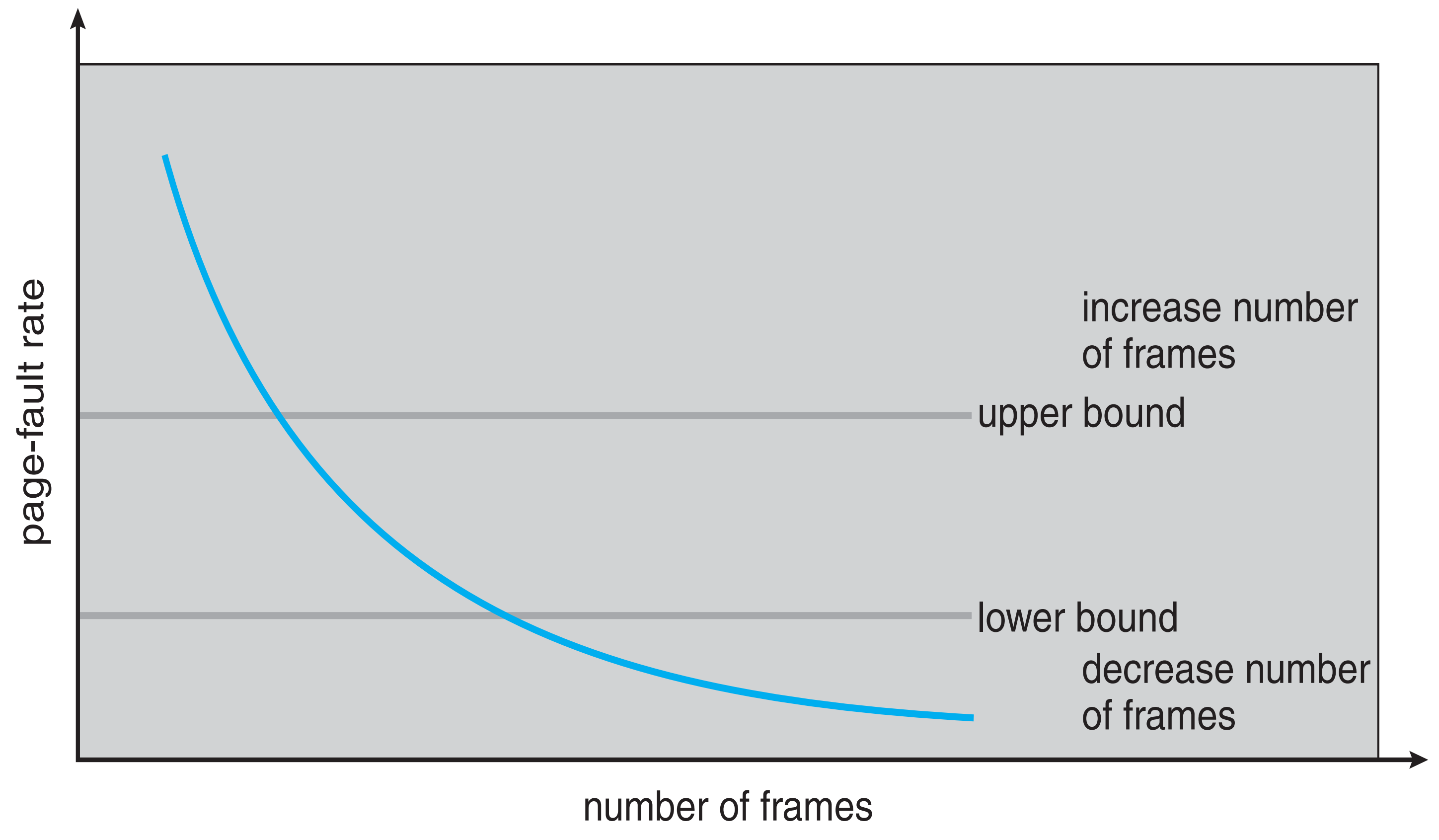


- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (Working Set Size of Process i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- D (Total demand for frames) = Σ WSSi
  - Approximation of locality
- if D > available frames m $\Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes

How to keep track of the Working Set?

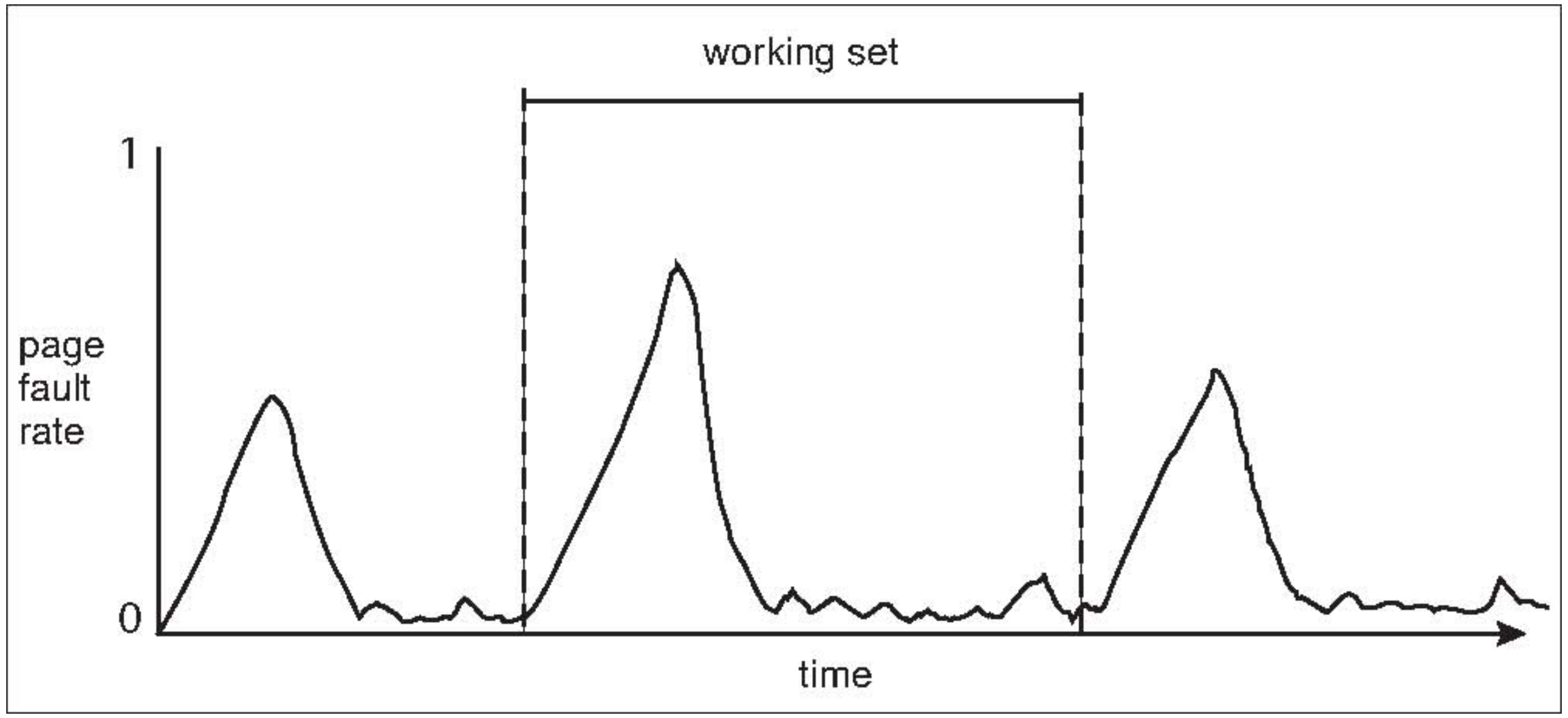
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



By looking at the Page Fault Rate you can observe the working set variations over time

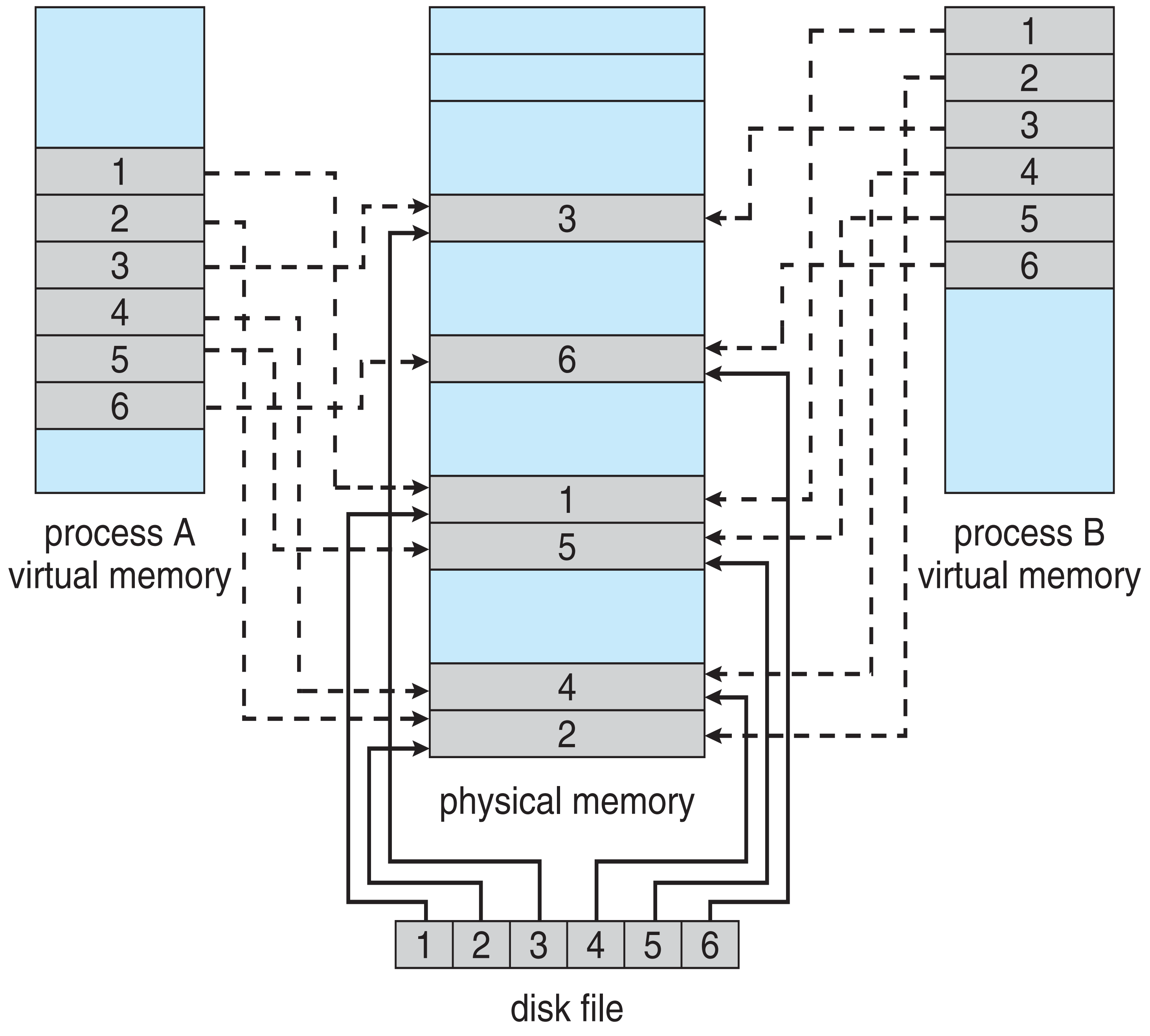
Agenda

- Recap / Introduction
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped files [Self-reading]

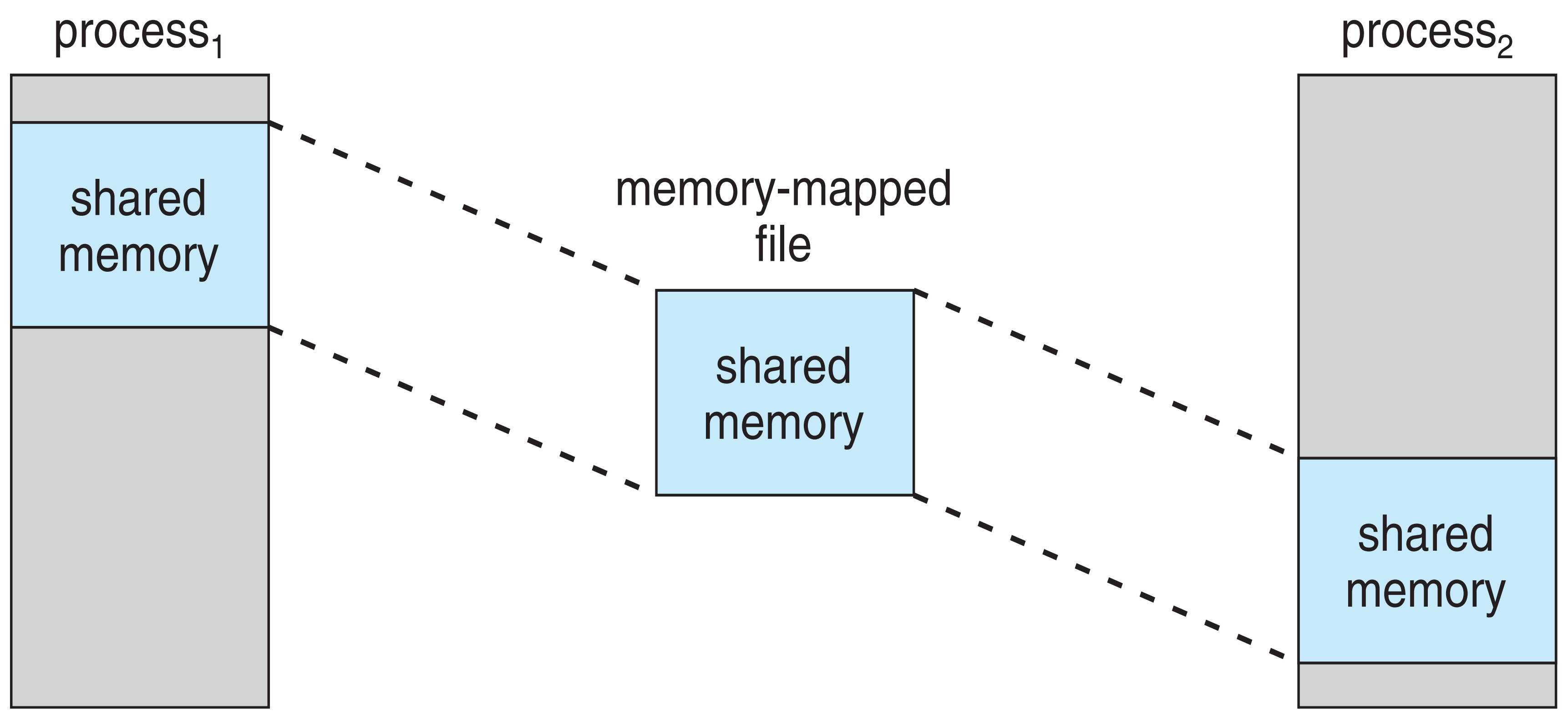
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

Memory Mapped Files



Shared Memory via Memory-Mapped I/O



Thank you for your attention!

Please evaluate the lecture!

https://forms.gle/EKVCNbxYsCf16eiGA