Virtual Memory

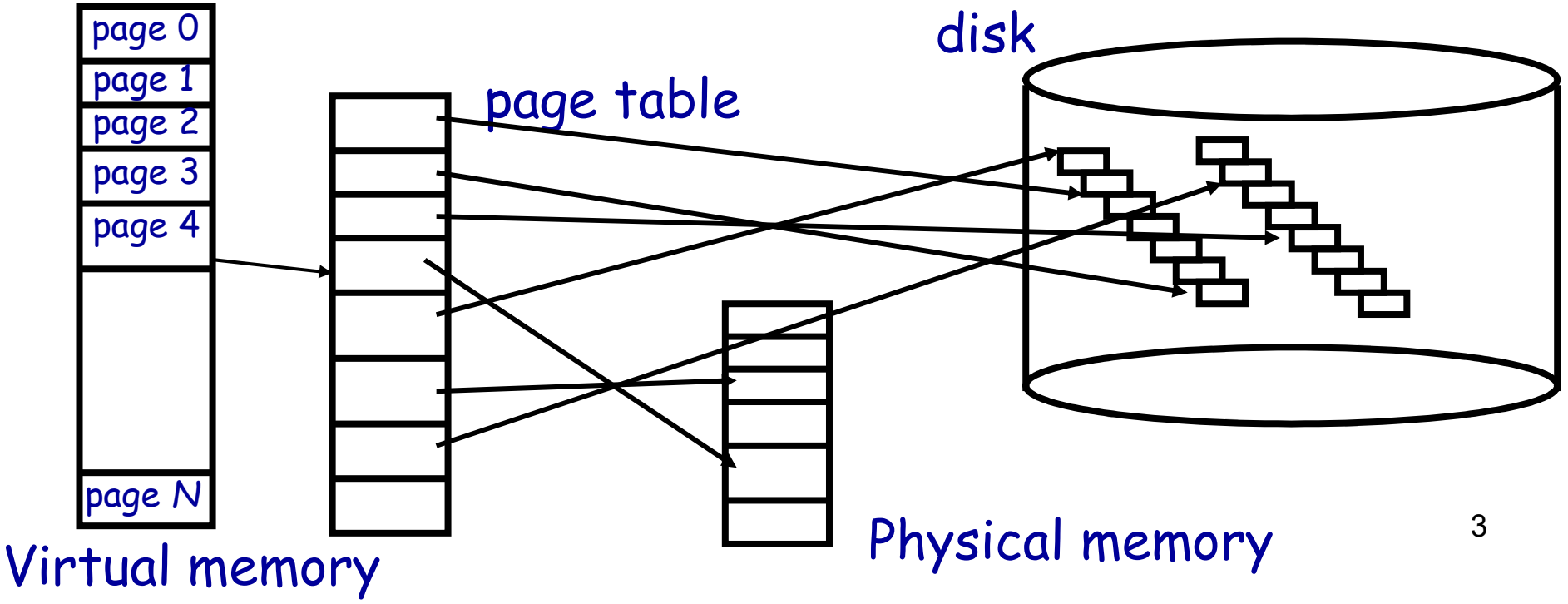
Yi Shi
Fall 2018
Xi'an Jiaotong University

Goals for Today

- Virtual memory
- How does it work?
  - Page faults
  - Resuming after page faults
- When to fetch?
- What to replace?
  - Page replacement algorithms
    - FIFO, OPT, LRU (Clock)
  - Page Buffering
  - Allocating Pages to processes
- What is thrashing?

What is virtual memory?

- Each process has illusion of large address space
  - 2^{32} for 32-bit addressing
- However, physical memory is much smaller
- How do we give this illusion to multiple processes?
  - Virtual Memory: some addresses reside in disk



Virtual Memory

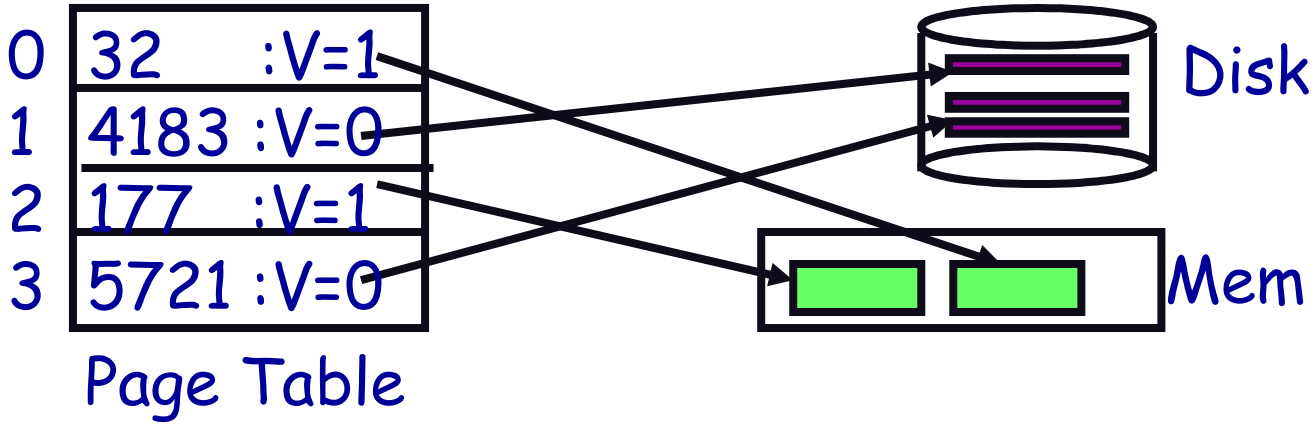
- Separates users logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation

Virtual Memory

- Load entire process in memory (swapping), run it, exit
  - Is slow (for big processes)
  - Wasteful (might not require everything)
- Solutions: partial residency
  - Paging: only bring in pages, not all pages of process
  - Demand paging: bring only pages that are required
- Where to fetch page from?
  - Have a contiguous space in disk: swap file (pagefile.sys)

How does VM work?

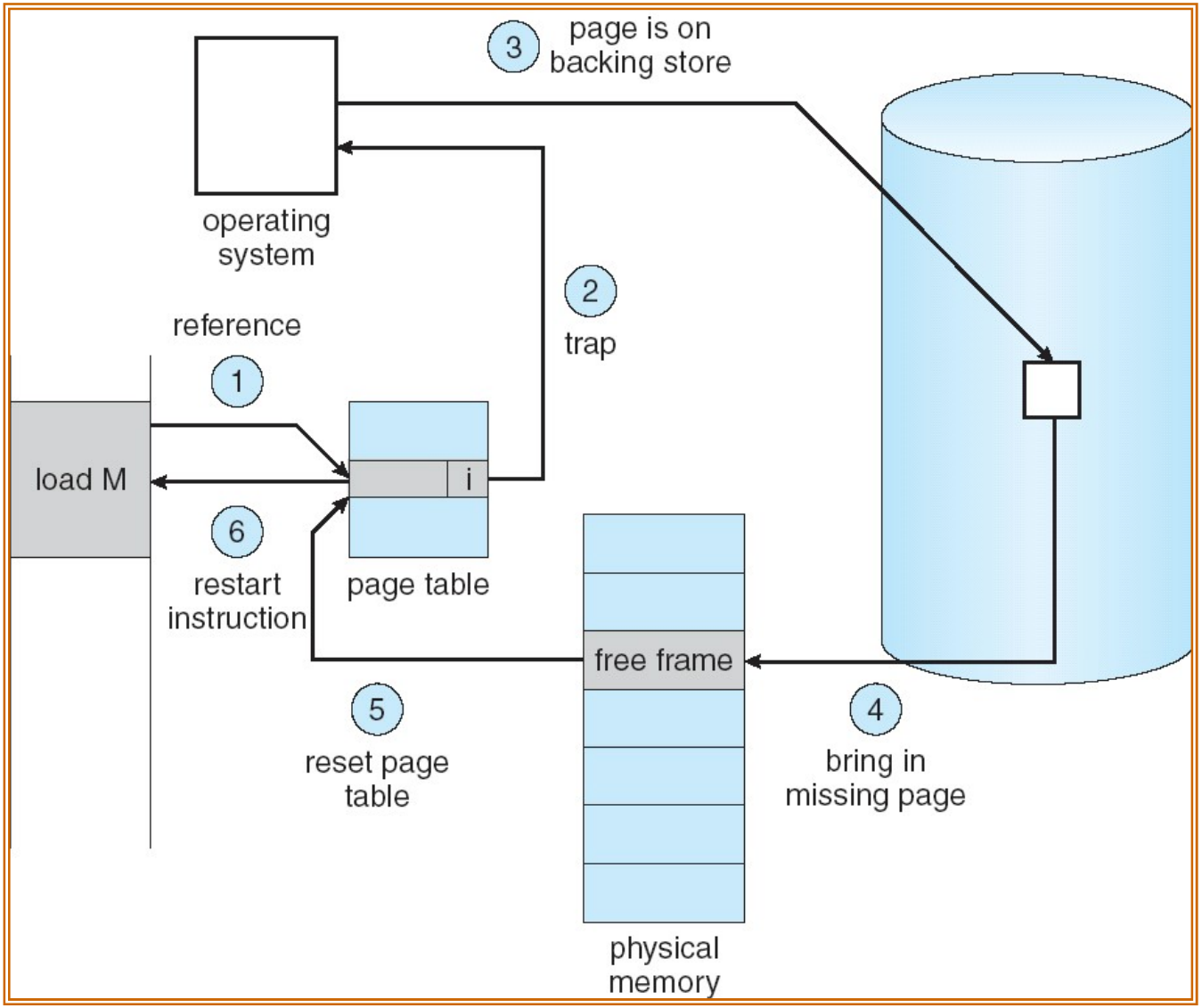
- Modify Page Tables with another bit (“valid”)
  - If page in memory, *valid = 1*, else *valid = 0*
  - If page is in memory, translation works as before
  - If page is not in memory, translation causes a **page fault**



Page Faults

- On a page fault:
  - OS finds a free frame, or evicts one from memory (which one?)
    - Want knowledge of the future?
  - Issues disk request to fetch data for page (what to fetch?)
    - Just the requested page, or more?
  - Block current process, context switch to new process (how?)
    - Process might be executing an instruction
  - When disk completes, set valid bit to 1, and current process in ready queue

Steps in Handling a Page Fault



Resuming after a page fault

- Should be able to restart the instruction
- For RISC processors this is simple:
  - Instructions are idempotent until references are done
- More complicated for CISC:
  - E.g. move 256 bytes from one location to another
  - Possible Solutions:
    - Ensure pages are in memory before the instruction executes

When to fetch?

- Just before the page is used!
  - Need to know the future
- Demand paging:
  - Fetch a page when it faults
- Prepaging:
  - Get the page on fault + some of its neighbors

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

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

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

What to replace?

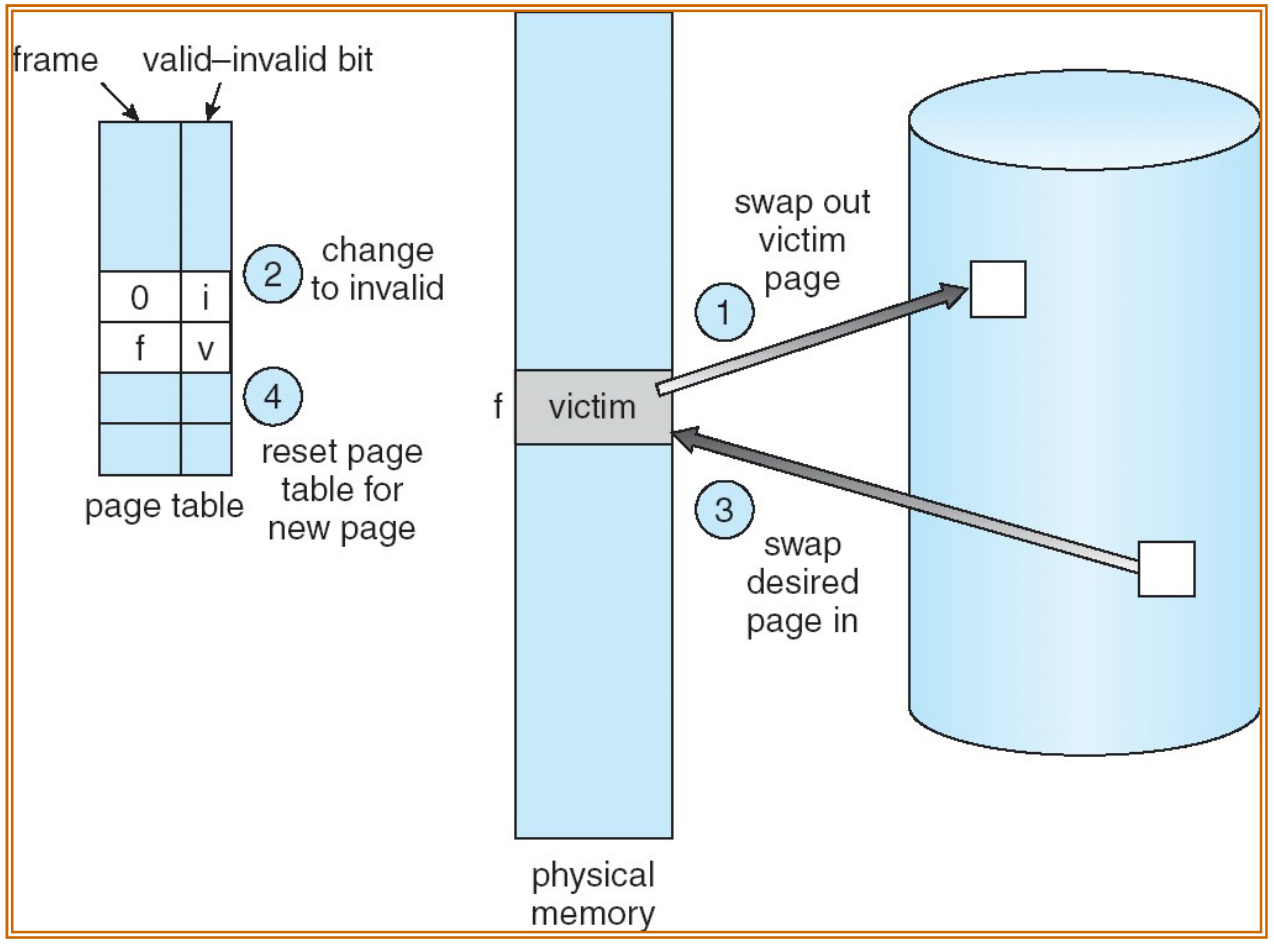
- What happens if there is no free frame?
  - find some page in memory, but not really in use, swap it out
- Page Replacement
  - When process has used up all frames it is allowed to use
  - OS must select a page to eject from memory to allow new page
  - The page to eject is selected using the Page Replacement Algorithm

- Goal: Select page that minimizes future page faults

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

Page Replacement

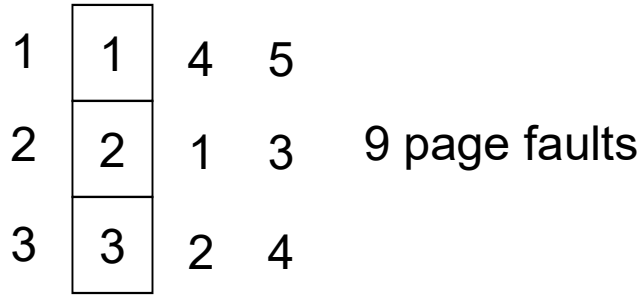


Page Replacement Algorithms

- Random: Pick any page to eject at random
  - Used mainly for comparison
- FIFO: The page brought in earliest is evicted
  - Ignores usage
  - Suffers from “Belady’s Anomaly”
    - Fault rate could increase on increasing number of pages
    - E.g. 0 1 2 3 0 1 4 0 1 2 3 4 with frame sizes 3 and 4
- OPT: Belady’s algorithm
  - Select page not used for longest time
- LRU: Evict page that hasn’t been used the longest
  - Past could be a good predictor of the future

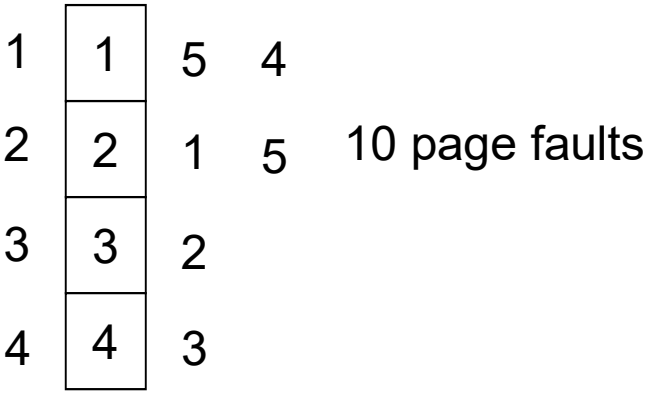
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



9 page faults

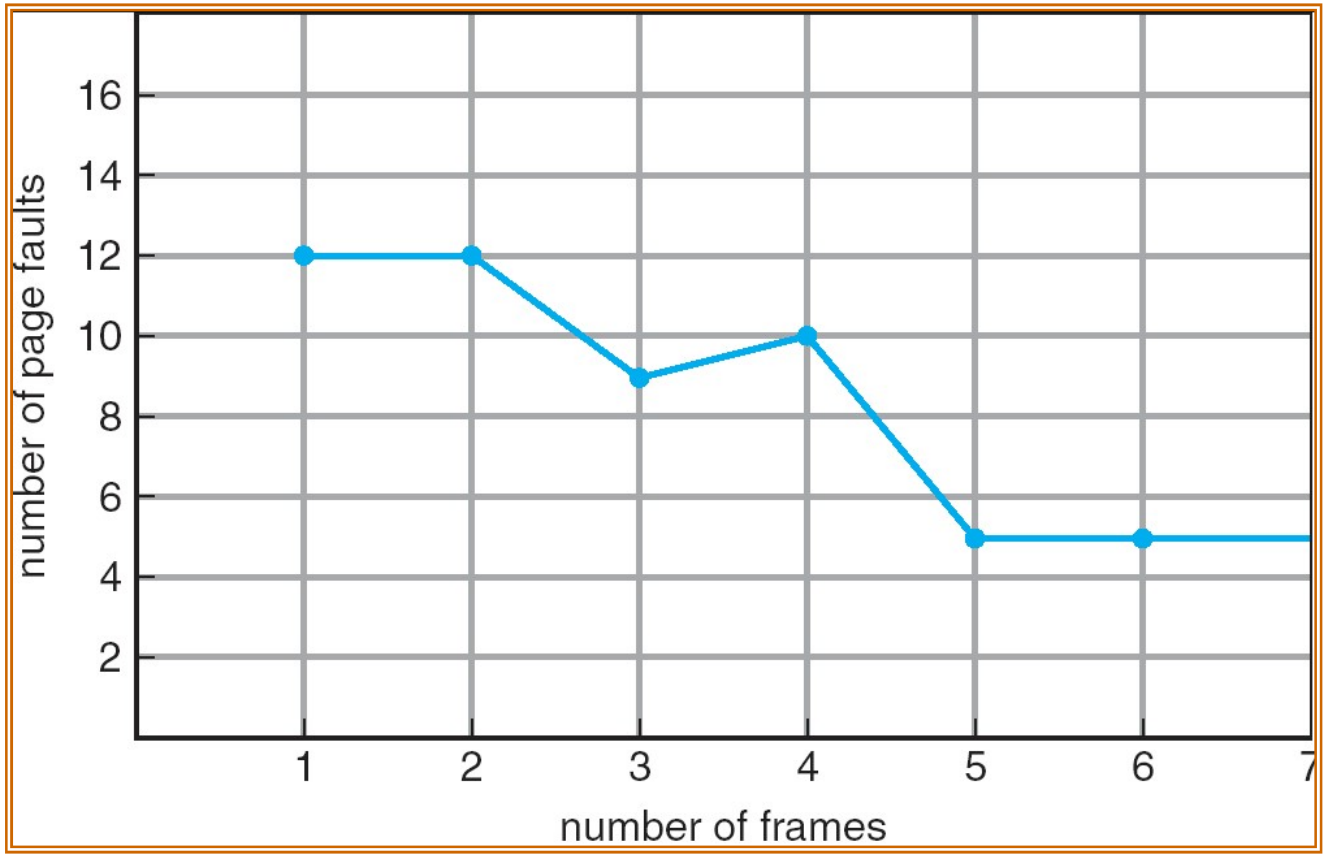
- 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



10 page faults

- Belady's Anomaly: more frames $\Rightarrow$ more page faults

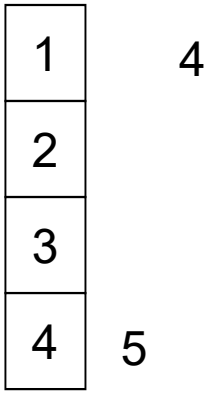
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

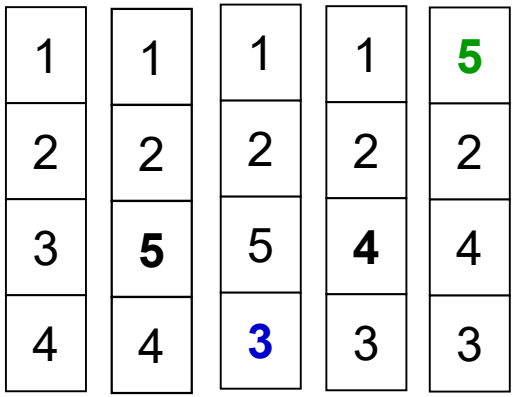


6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



Exam LRU, Δ=3

While "filling" memory we are likely to get a page fault on almost every reference. Usually we don't include these events when computing the hit ratio

R	3	7	9	5	3	6	3	5	6	7	9	7	9	3	8	6	3	6	8	3	5	6
S	3	7	9	5	3	6	3	5	6													6
	∅	3	7	9	5	3	6	3	5	6				7	9	3	8	6	3	6	8	3	5
	∅	∅	3	7	9	5	6	3	5	6	6	6	7	9	3	8	8	3	6	8	3		
In	3	7	9	5	3	6	∅	∅	∅	7	9	∅	∅	3	8	6	∅	∅	∅	∅	5	6	
Out	∅	∅	∅	3	7	9	∅	∅	∅	3	5	∅	∅	6	7	9	∅	∅	∅	∅	6	8	

Hit ratio: 9/19 = 47%
Miss ratio: 10/19 = 53%

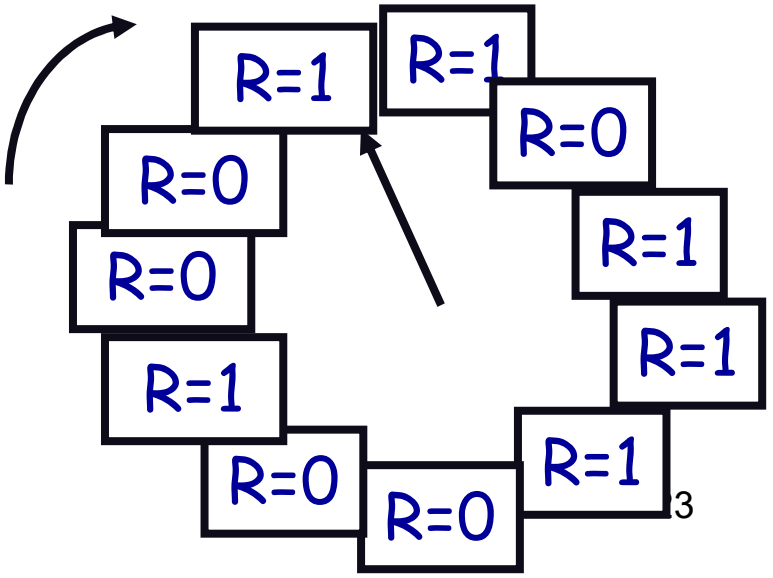
R(t): Page referenced at time t. S(t): Memory state when finished doing the paging at time t. In(t): Page brought in, if any. Out(t): Page sent out. ∅: None.

Implementing Perfect LRU

- On reference: Time stamp each page
- On eviction: Scan for oldest frame
- Problems:
  - Large page lists
  - Timestamps are costly
- Approximate LRU
  - LRU is already an approximation!

LRU: Clock Algorithm

- Each page has a reference bit
  - Set on use, reset periodically by the OS
- Algorithm:
  - FIFO + reference bit (keep pages in circular list)
    - Scan: if ref bit is 1, set to 0, and proceed. If ref bit is 0, stop and evict.
- Problem:
  - Low accuracy for large memory

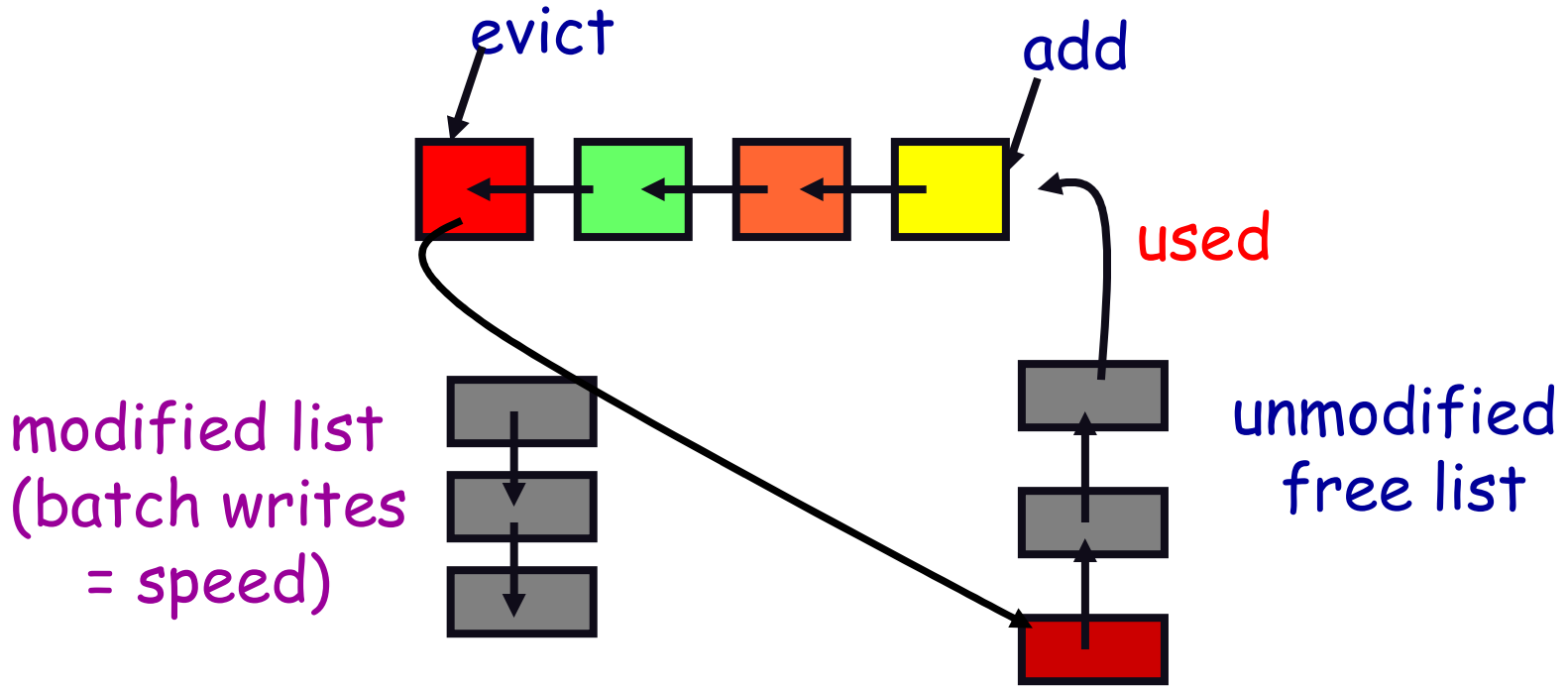


Clock Algorithm: Discussion

- Sensitive to sweeping interval
  - Fast: lose usage information
  - Slow: all pages look used
- Clock: add reference bits
  - Could use (ref bit, modified bit) as ordered pair
  - Might have to scan all pages
- LFU: Remove page with lowest count
  - No track of when the page was referenced
  - Use multiple bits. Shift right by 1 at regular intervals.
- MFU: remove the most frequently used page
- LFU and MFU do not approximate OPT well

Page Buffering

- Cute simple trick: (XP, 2K, Mach, VMS)
  - Keep a list of free pages
  - Track which page the free page corresponds to
  - Periodically write modified pages, and reset modified bit



Allocating Pages to Processes

- Global replacement
  - Single memory pool for entire system
  - On page fault, evict oldest page in the system
  - Problem: protection
- Local (per-process) replacement
  - Have a separate pool of pages for each process
  - Page fault in one process can only replace pages from its own process
  - Problem: might have idle resources

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Thrashing

- Def: Excessive rate of paging that occurs because processes in system require more memory
  - Keep throwing out page that will be referenced soon
  - So, they keep accessing memory that is not there

- Why does it occur?
  - Poor locality, past != future
  - There is reuse, but process does not fit
  - Too many processes in the system

Summary

- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss needs get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - OPT: replace page that will be used farthest in future
  - LRU: Replace page that hasn't be used for the longest time
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- Thrashing: a process is busy swapping pages in and out

Exercises

- 9.4
- 9.5
- 9.13