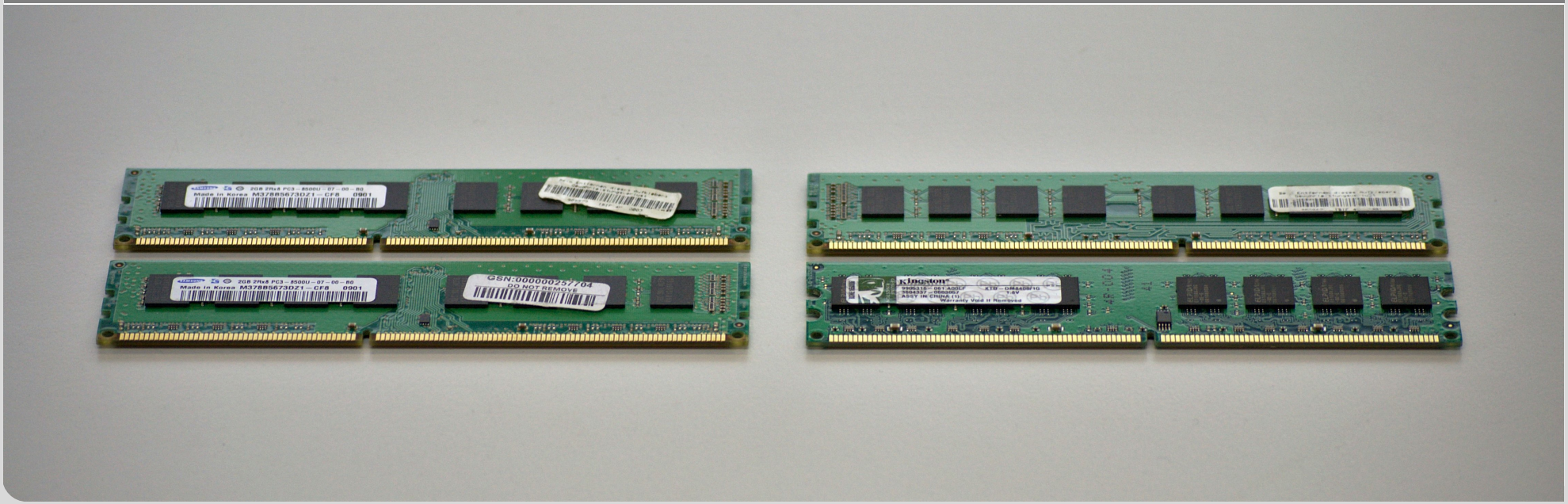
Karlsruhe Institute of Technology

Betriebssysteme

13. Page Replacement Policies

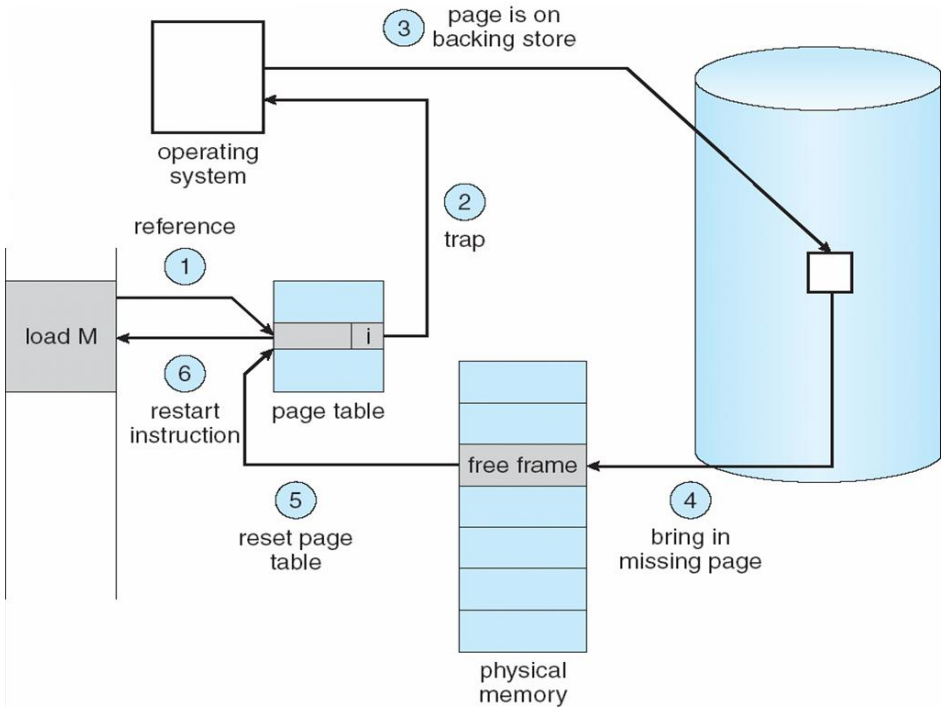
Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP



Page Fault Handling

- Access to page that is currently not present in main memory causes page fault (exception that invokes OS)



1. OS checks validity of access (requires additional info)
2. **Get empty frame**
3. Load contents of requested page from disk into frame
4. Adapt page table
5. Set valid-invalid bit of respective entry to **valid**
6. Restart instruction that caused the page fault

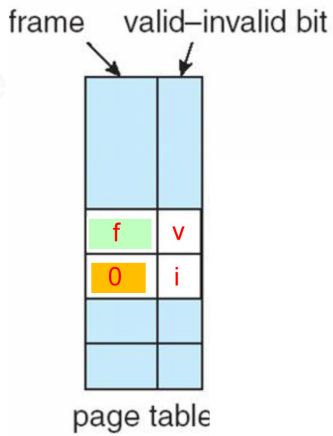
Today: How to pick/make an empty frame

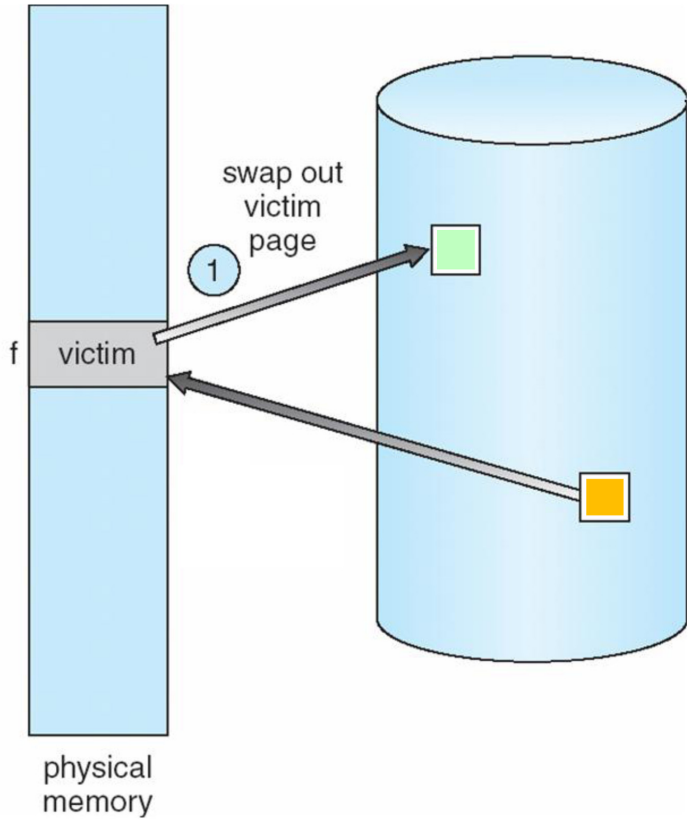
Page Replacement Policies

How to find a page to evict from memory

Naïve Page Replacement

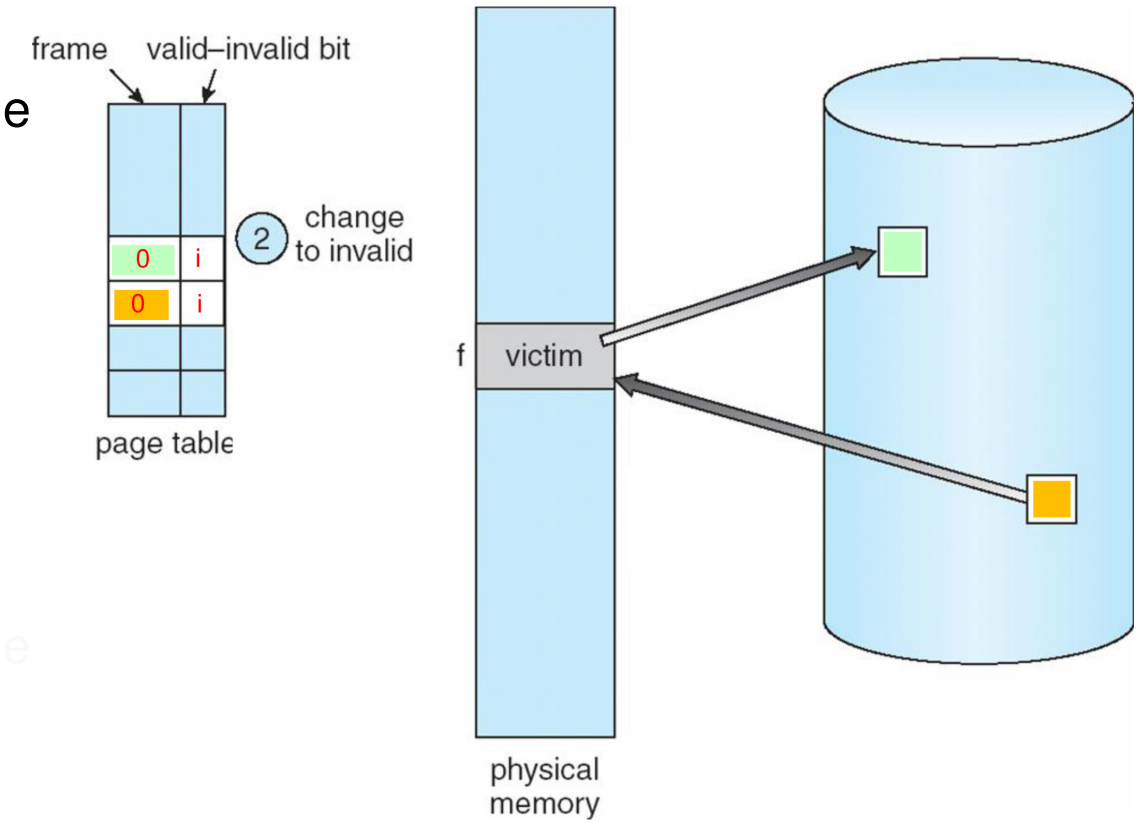
- Save/clear victim page
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty (unless **`MAP_COPY`**)
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - Invalidate PTE + flush cache
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - Invalidate PTE + flush cache





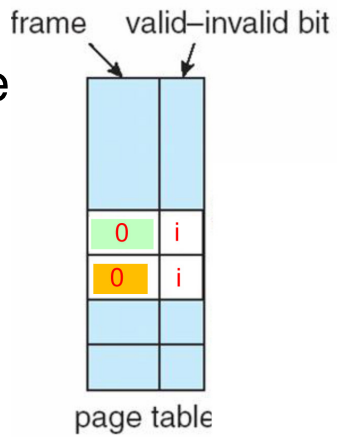
Naïve Page Replacement

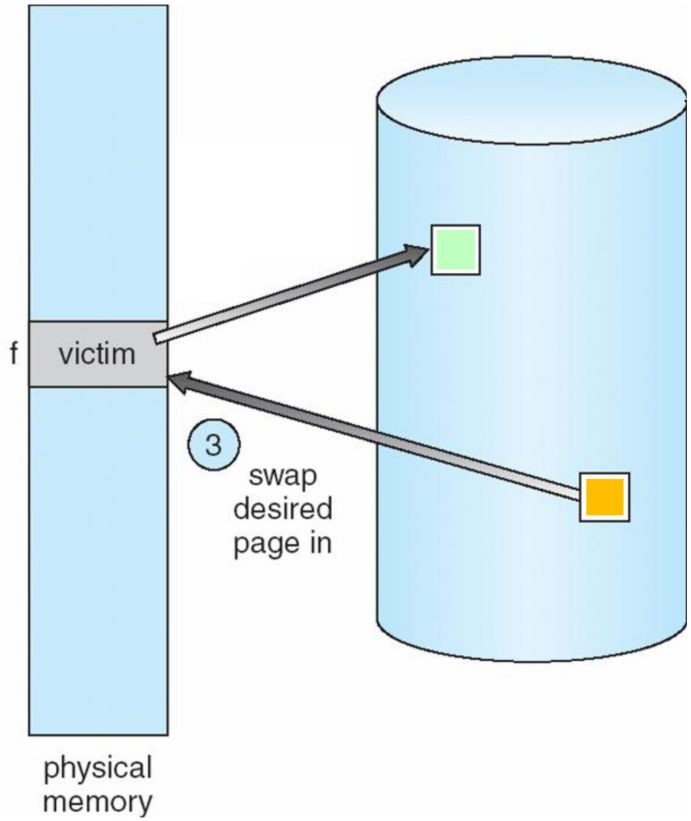
- Save/clear victim page
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty (unless **`MAP_COPY`**)
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - Invalidate PTE + flush cache
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - Invalidate PTE + flush cache



Naïve Page Replacement

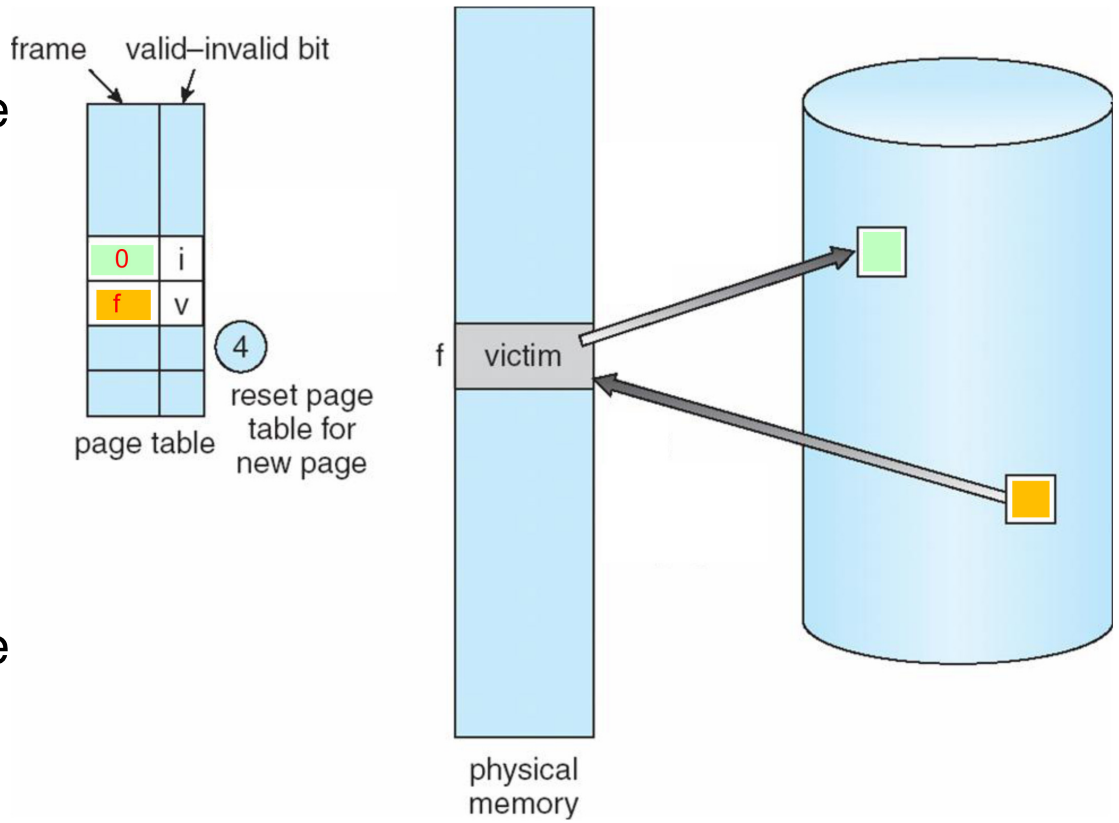
- Save/clear victim page
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty (unless **`MAP_COPY`**)
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - Invalidate PTE + flush cache
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - Invalidate PTE + flush cache





Naïve Page Replacement

- Save/clear victim page
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty (unless **`MAP_COPY`**)
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - Invalidate PTE + flush cache
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - Invalidate PTE + flush cache

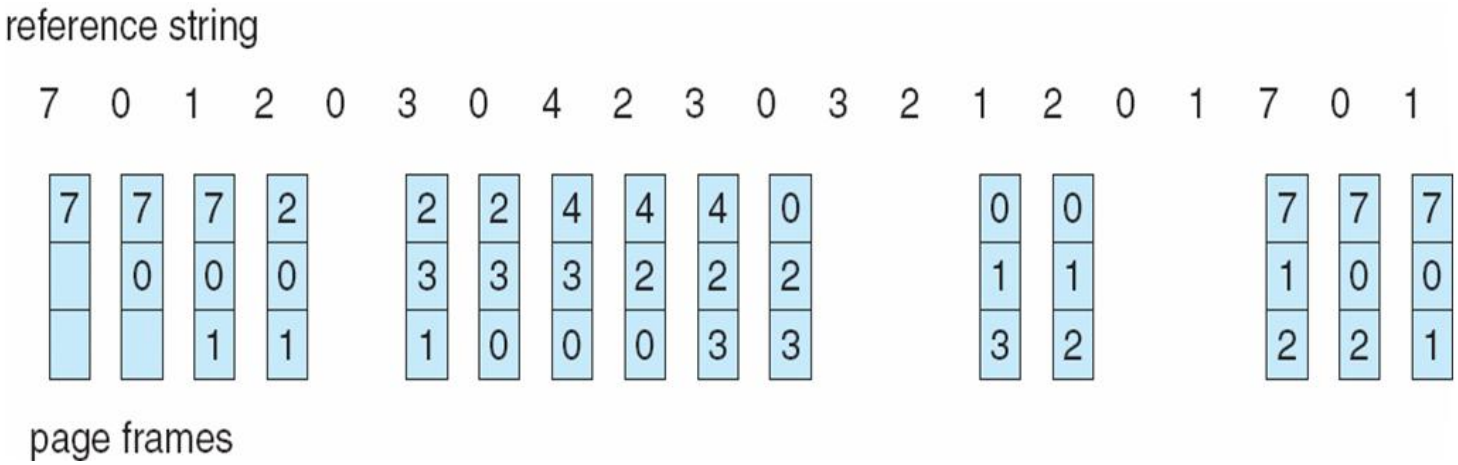


Page Buffering

- Problem: Naïve page replacement encompasses two I/O transfers swapping out (demand cleaning) and swapping the new page in
  - Both operations block the page fault from completing
- Goal: Reduce I/O from critical page fault path to speed up page faults
- Idea: Keep pool of free page frames (pre-cleaning)
  - On a page fault, use a page frame from the free pool
  - Run a daemon that cleans (write back changes), reclaims (unmap), and scrubs (zero out) pages for the free pool in the background
- Such a free pool smoothes out I/O and speeds up paging significantly
- Remaining problem: Which pages to select as victims?
  - Goal: Identify a page that has left the working set of its process to add it to the free pool
  - Success metric: Low overall page fault rate

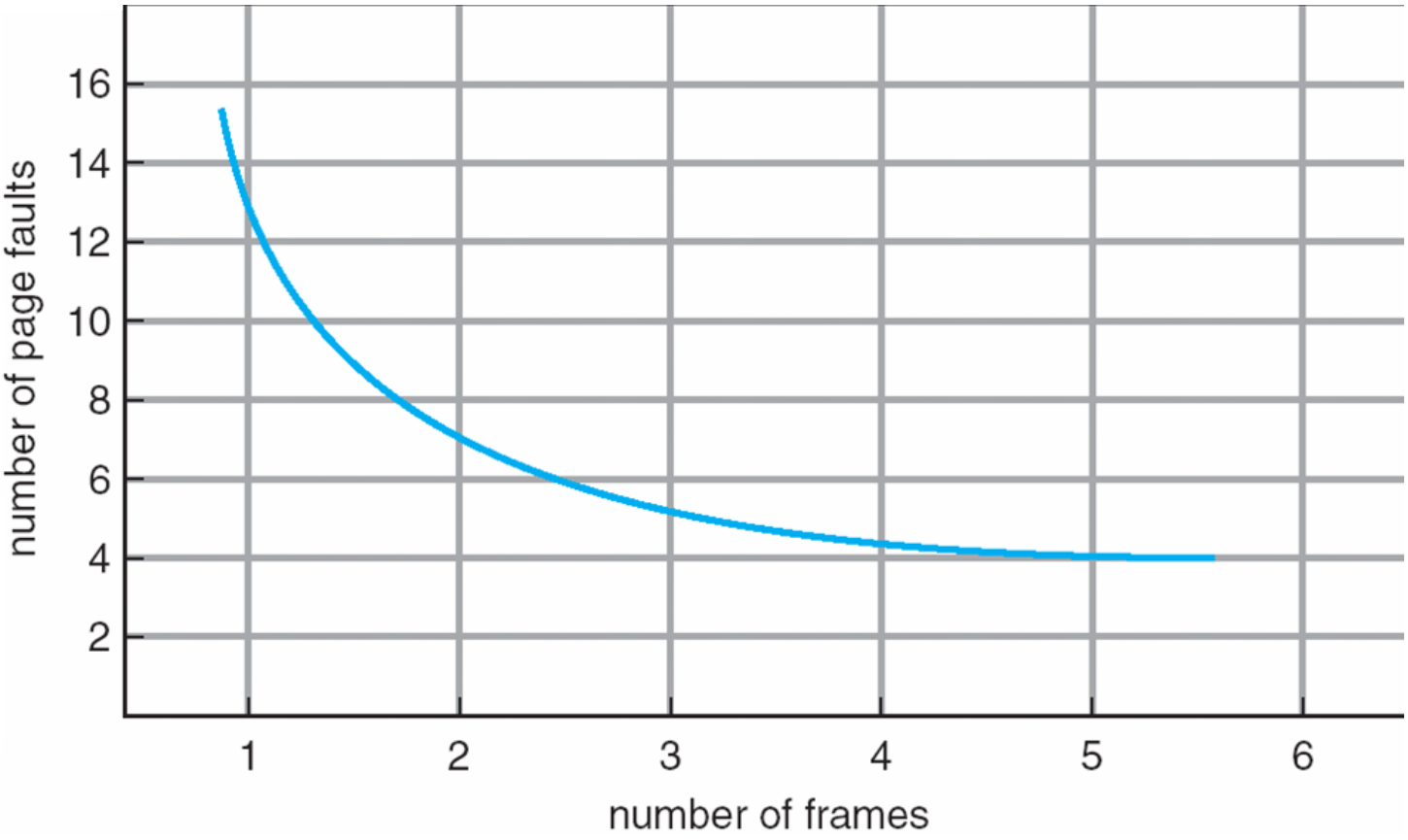
First-In-First-Out (FIFO) Page Replacement

- Evict the oldest fetched page in the system



Intuition: Page Fault Rate vs. Number of Frames

- Intuitively one would say that the page fault rate decreases when the amount of memory increases
- This is true most of the time, but not universally

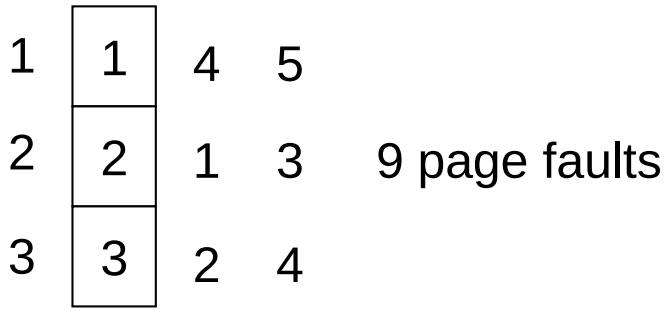


Belady's Anomaly

- Reference string for all our examples: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Belady's Anomaly
  - When using FIFO page replacement, for every number **N** of page frames you can construct a reference string that performs worse with **N+1** frames

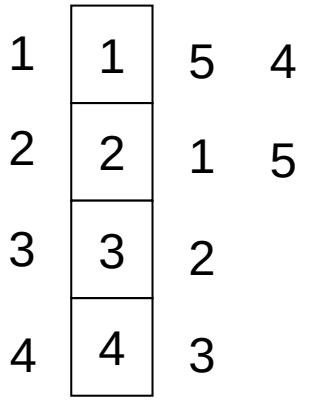
→ With FIFO it is possible to get more page faults with more page frames

3 frames



9 page faults

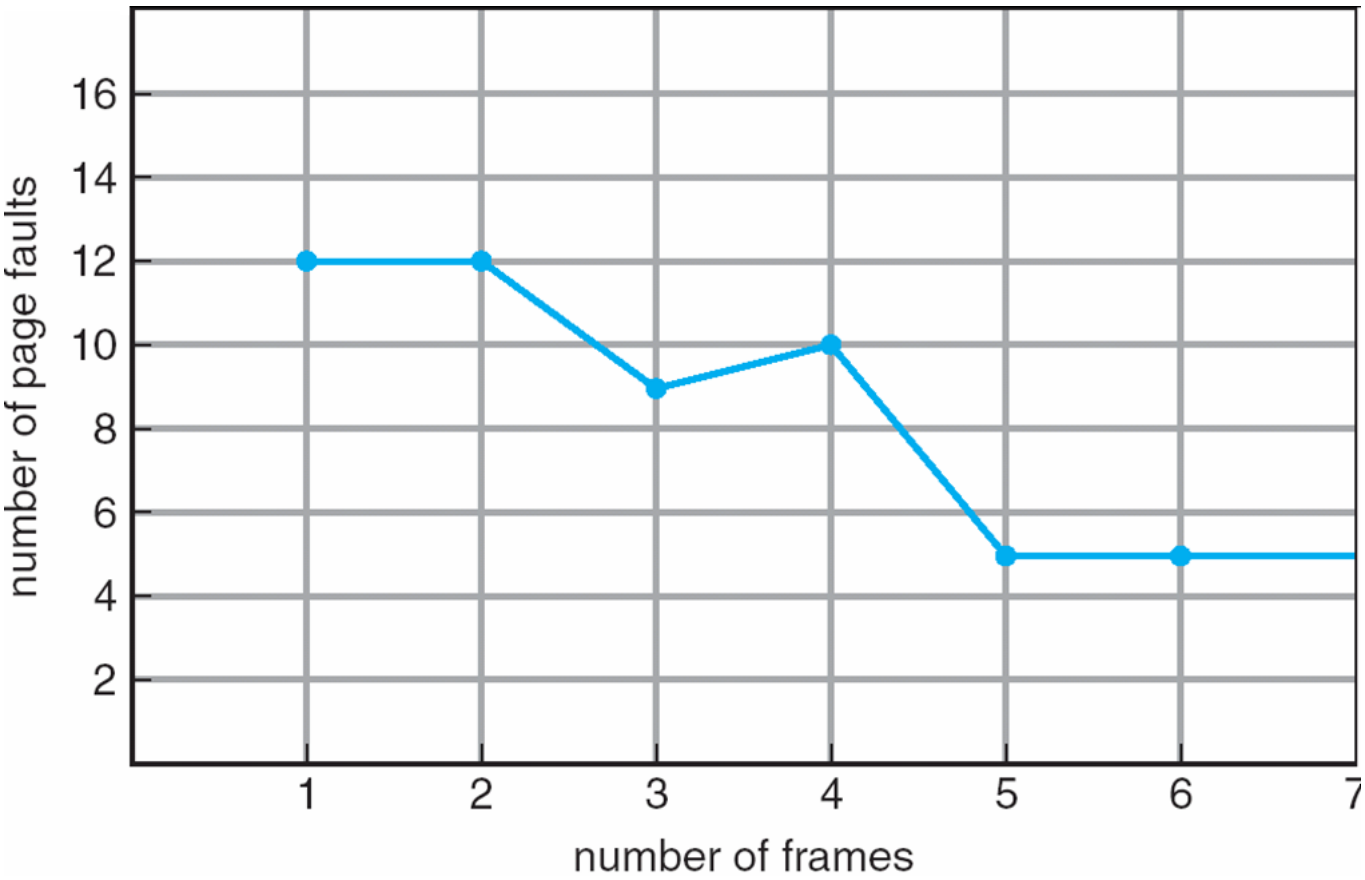
4 frames



10 page faults

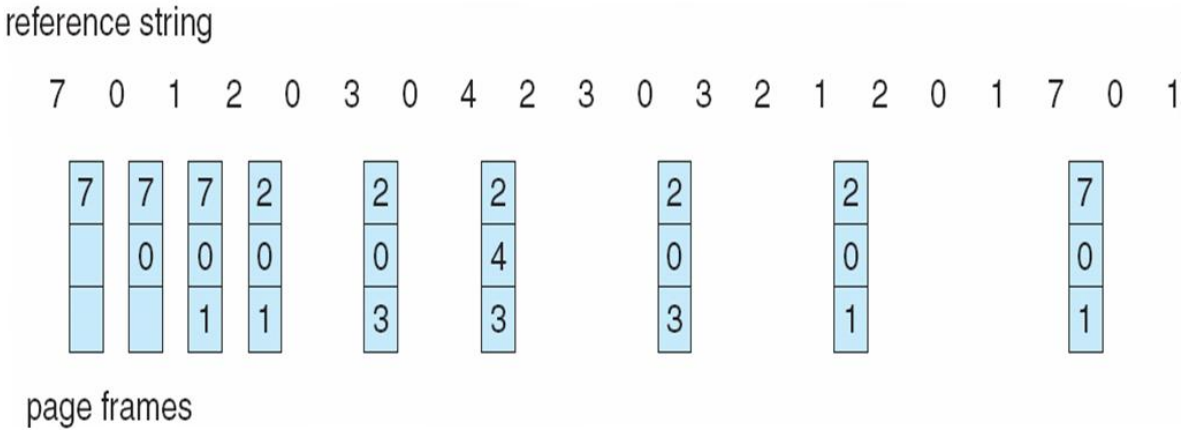
Belady's Anomaly using FIFO page replacement

- More physical memory doesn't always imply fewer faults

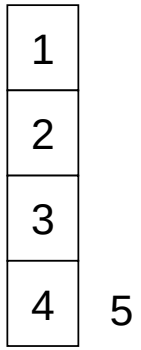


Oracle: Optimal Page Replacement

- The optimal page replacement strategy is to replace the page whose next reference is furthest in the future



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



4

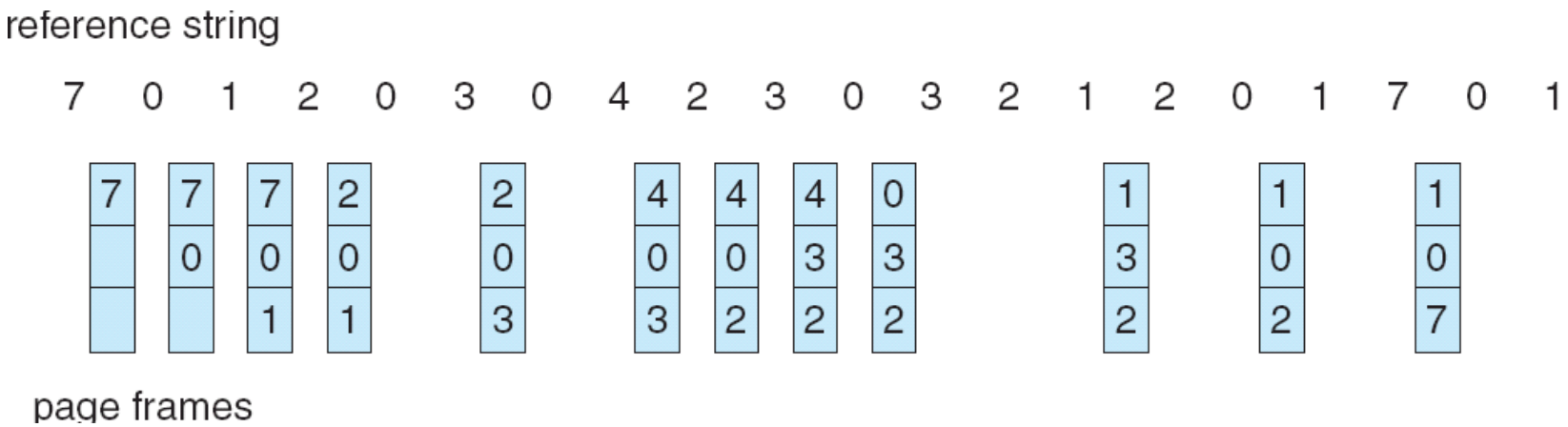
5

6 page faults

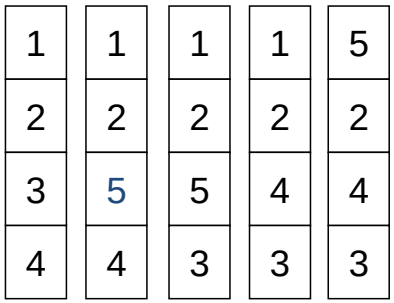
- Cannot predict the future
  - Not suitable in practice
  - However: Good metric to check how well other algorithms perform

Least Recently Used (LRU) Page Replacement

- Goal: Approximate Oracle page replacement
- Idea: Past often predicts the future well
- Assumption: Page used furthest in past is used furthest in the future



- Reference string
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  8 page faults



LRU: Easy to understand, hard to implement well

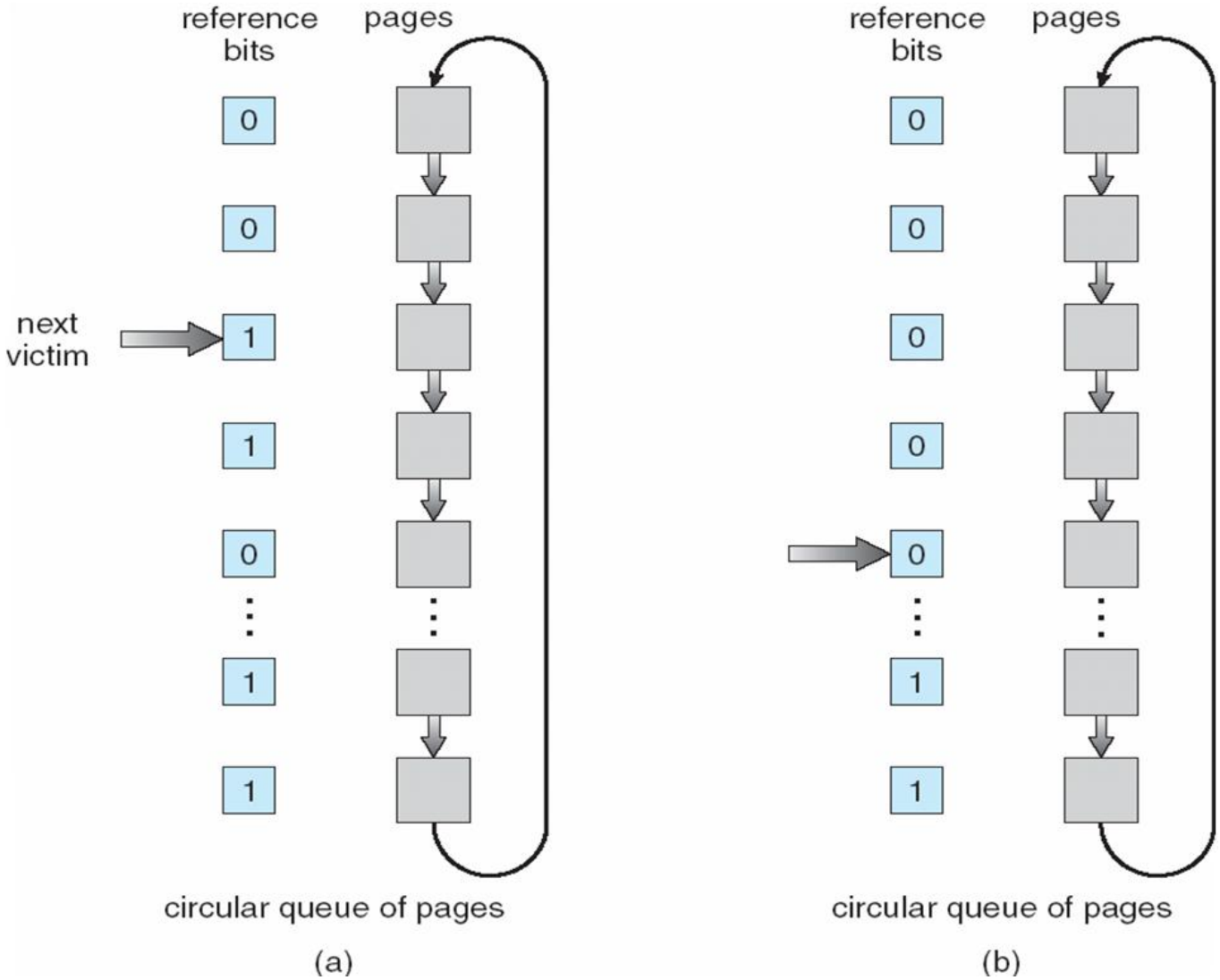
- Cycle counter implementation
  - Have MMU write CPU's time stamp counter to PTE on every access
  - On a page fault: Scan all PTEs to find oldest counter value
  - \+ Cheap at access if done in HW
  - – Memory traffic for scanning
- Stack implementation
  - Keep a doubly linked list of all page frames
  - Move each referenced page to tail of list
  - \+ Can find replacement victim in O(1)
  - – Need to change 6 pointers at every access
- No silver bullet
  - Observation: Predicting the future based on the past is not precise
  - Conclusion: Relax requirements – maybe perfect LRU is not needed?
    → Approximate LRU

LRU Approximation: Clock Page Replacement

- Clock page replacement is a.k.a. second chance page replacement
- Precondition: MMU sets reference bit in PTE
  - Supported natively by most hardware (e.g., IA-32, x86-64, . . .)
  - Can easily emulate in systems with software managed TLB (e.g., MIPS)
- Keep all pages in circular FIFO list
- When searching for a victim scan pages in FIFO's order
  - If reference bit is `0` → use page as victim and advance hand[1]
  - If reference bit is `1` → set to `0` and continue scanning
- Large memory → most pages referenced before scanned
  - Use 2 arms: Leading arm clears reference bit, trailing arm selects victim

[1] Reference bit will be set by hardware after page fault when retrying access

Clock Page Replacement



Other replacement strategies

- Random eviction
  - Just pick a victim at random
  - Dirt simple and in reality not overly horrible

- Use larger counter: Use n-Bit reference counter instead of reference bit
  - Least frequently used (LFU)
    - Idea: Rarely used page is not in a working set

    → Replace page with smallest count
  - Most frequently used (MFU)
    - Idea: The page with the smallest count was probably just brought in and will be used soon

    → Replace page with the largest count
  - Neither LFU nor MFU are common (no such hardware + not that great)

Summary

- When handling page faults, the OS needs to select a victim page frame for eviction
  - Evicting a page frame after the page fault happens is not a good idea
  - Page buffering keeps the eviction out of the critical path
- Different victim selection policies have been implemented in the past
  - FIFO → Belady's Anomaly
  - Oracle → Cannot predict the future
  - Random → Unpredictable, never great but rarely very bad
  - LRU → Hard to implement efficiently
- LRU works "OK", but need to approximate to lower overhead
  - Clock
  - 2-armed clock

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition:
  - Pages 209–222