CSCI315 – Operating Systems Design

Department of Computer Science
Bucknell University

Demand Paging and Page Replacement - 1

Ch 10.2-10.4

This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.
Xiannong Meng, Fall 2021.

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):**

EAT = [(1 – *p*) (memory access)] + [*p* (page fault overhead)]

where:

page fault overhead = [swap page out] + [swap page in]
+ [restart overhead]

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 (p = 0.001), then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!! (in comparison to 200 ns memory time)
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Improve Performance

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

- **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
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- 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

The Need To Replace A Page

- When a page is referenced by a process, it is possible that the needed page is not in memory, resulting in a page fault.
- The missing page needs to be brought into the memory.
- What if there is no free memory frame for the needed page?
- We need to remove an existing page to make space for the new page!

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.

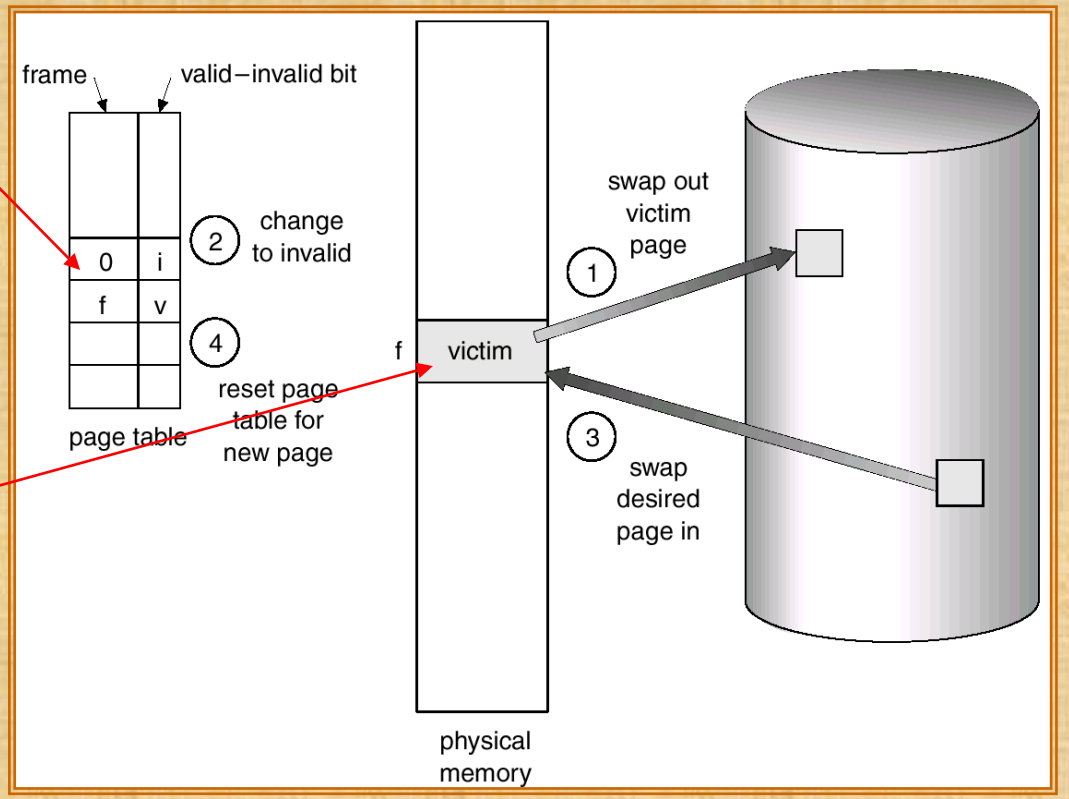
3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process.

Page Replacement

victim page

victim frame

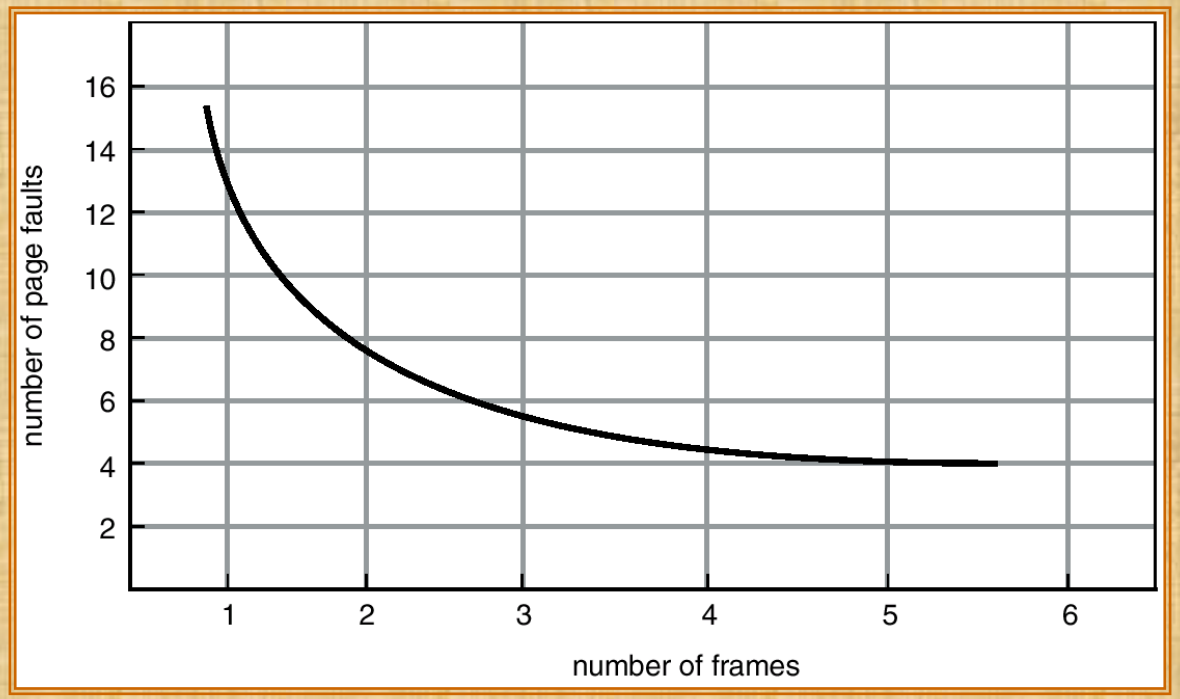


Page Replacement Algorithms

- **Goal:** Produce a low page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (***reference string***) and computing the number of page faults on that string.
- The reference string is produced by tracing a real program or by some stochastic model. We look at every address produced and strip off the page offset, leaving only the page number. For instance:

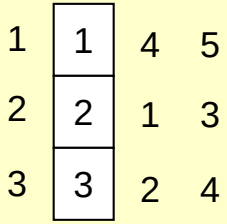
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults vs The Number of Frames



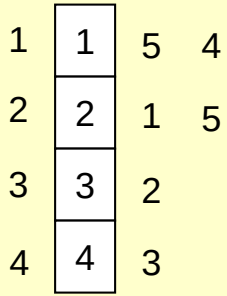
FIFO Page Replacement

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- 3 frames



9 page faults

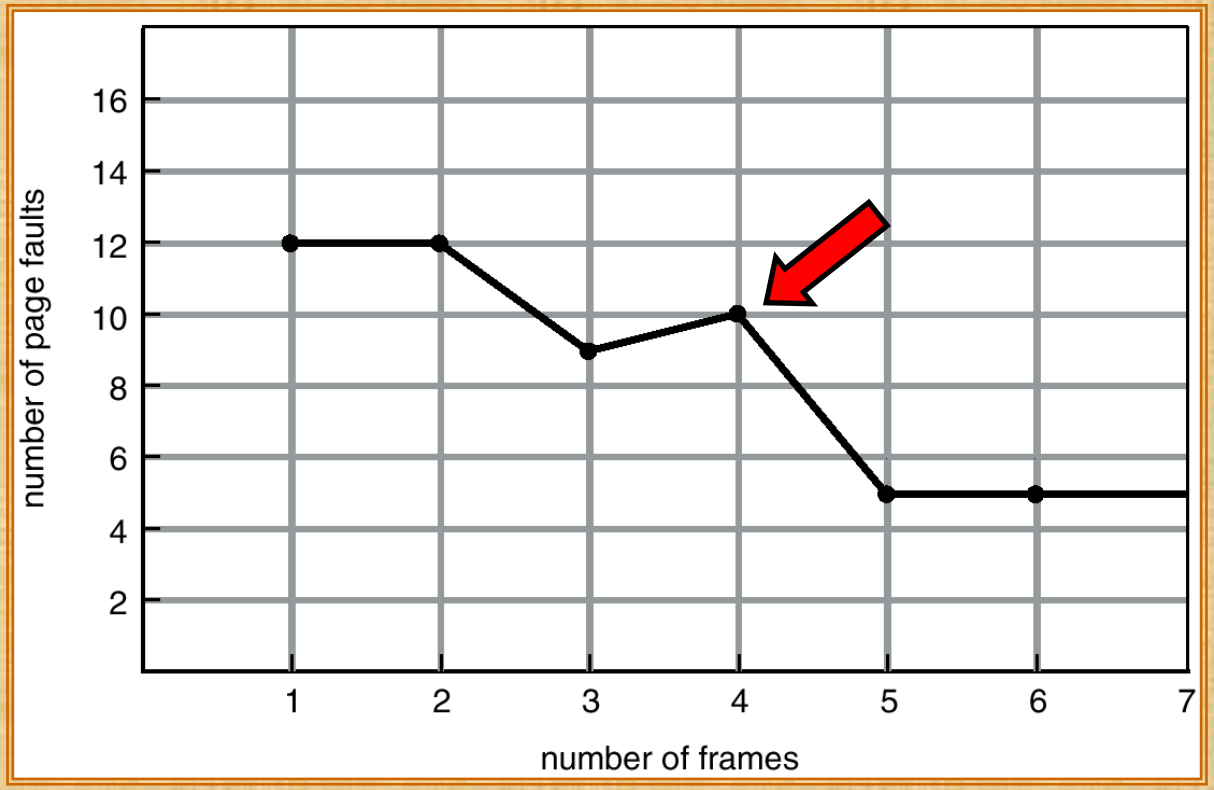
- 4 frames



10 page faults

- FIFO Replacement ⇒ **Belady's Anomaly:** more frames, *more* page faults, in this case.

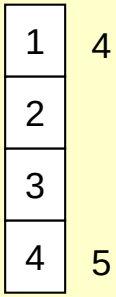
FIFO (Belady's Anomaly)



Belady's Anomaly is unbounded: (it can occur at any range)
https://en.wikipedia.org/wiki/B%C3%A9l%C3%A1dy%27s_anomaly

Optimal Algorithm

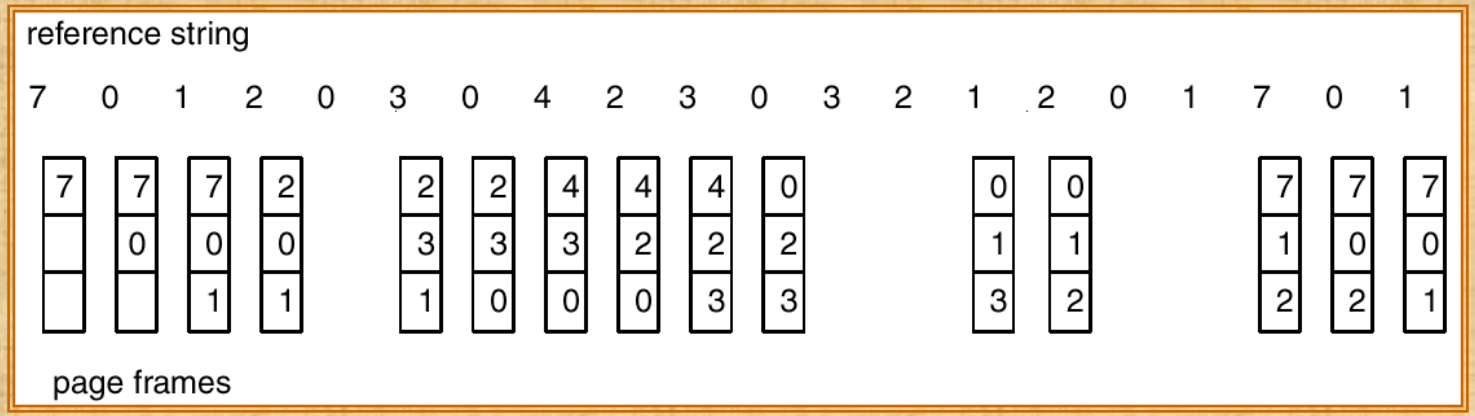
- Replace the page that will not be used for the **longest** period of time in the future.
- 4 frames example: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



6 page faults

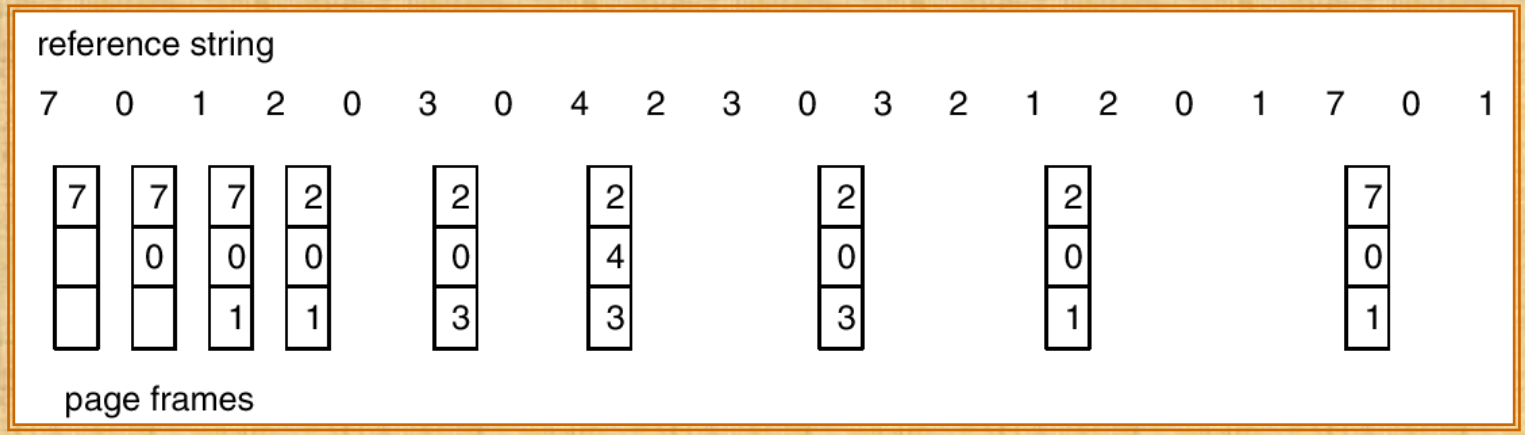
- Used for measuring how well your algorithm performs.
- How can you know what the future references will be?

Another FIFO Page Replacement Example



FIFO: 15 page faults

Optimal Page Replacement



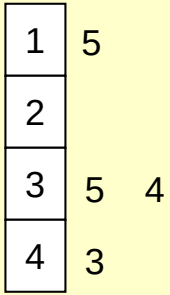
Optimal: 9 page faults with the same reference string

Optimal not Practical!

- Optimal page replace algorithm works great, except it is not practical!
  - Compare to optimal CPU scheduling algorithm (Shortest-Remaining-Time-First)
- We will try to approximate the optimal algorithm
  - In CPU scheduling, we try to predict the next CPU burst length and use it to approximate the SJF
- In page replacement, we use **LRU** (Least Recently Used) to approximate the optimal algorithm

LRU Algorithm

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



Optimal: 6 page faults
LRU: 8 page faults

- It works great!
- But, how do we implement the LRU algorithm? (more later.)