Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 14: Page Replacement

Prof. Ryan Huang

Administration

Fill in midterm teaching evaluation (due tomorrow)

Midterm grade stats

Mean	61.85
Median	63.25
Standard deviation	16.0

Midterm review session

- This Thursday 6-7:30 pm, CHRYS 220
- Regrade requests open after review session for a week

Project 3 is out

- Due in about three weeks (March 28th)

Page Replacement

When a page fault occurs, the OS needs a physical page to load the faulted page from disk into

What if all the page frames are in use?

The OS must choose a page frame to evict

- Free it up for use

Page replacement algorithm determines this

- **Goal:** minimize page faults
- Greatly affect performance of paging (virtual memory)
- Also called page eviction policies

Locality

All paging schemes depend on locality

- Processes reference pages in localized patterns

Temporal locality

- Locations referenced recently likely to be referenced again

Spatial locality

- Locations near recently referenced locations are likely to be referenced soon

Processes usually exhibit both kinds of locality, making paging practical despite its costs

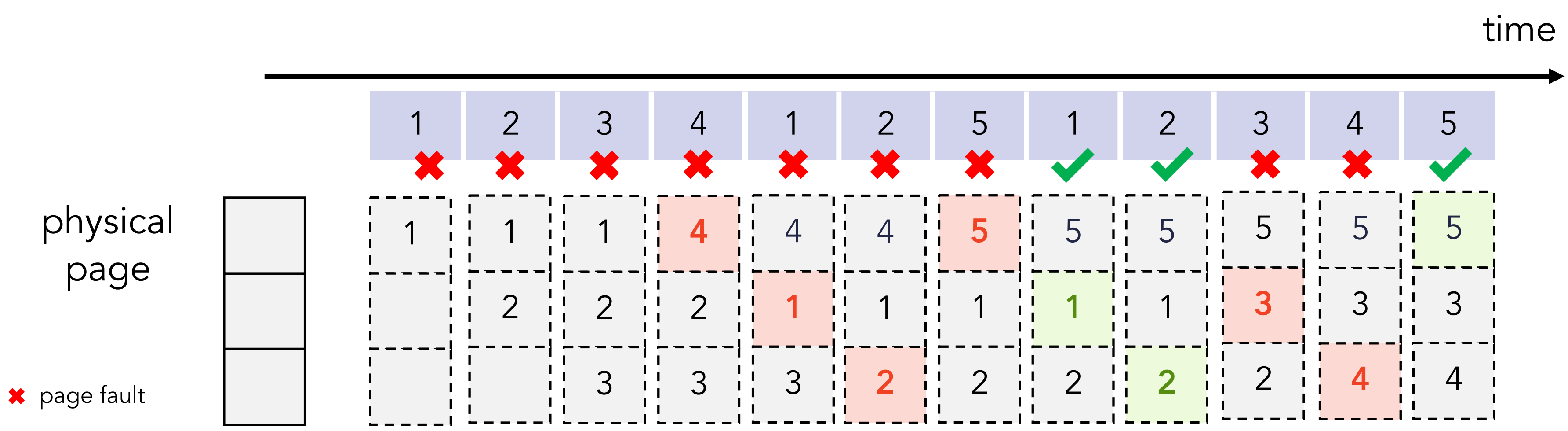
First-In First-Out (FIFO)

Evict "oldest" page

- Brought into memory longest time ago

Example

- Access trace (virtual page #): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages



9 page faults

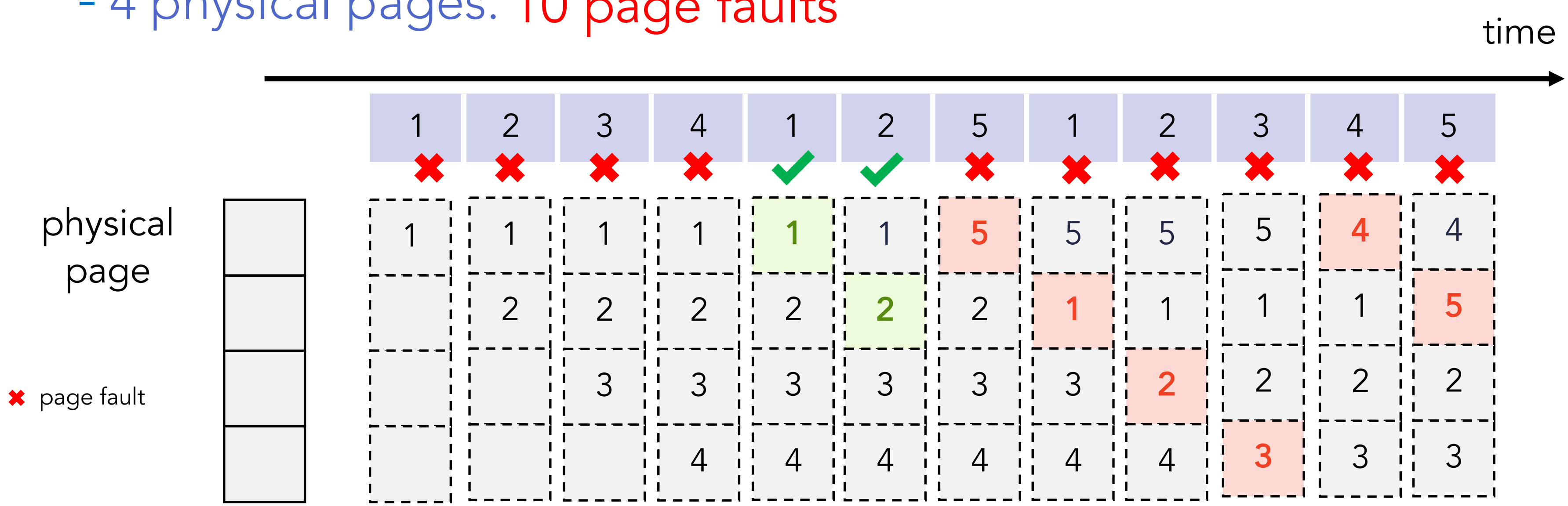
First-In First-Out (FIFO)

Evict "oldest" page

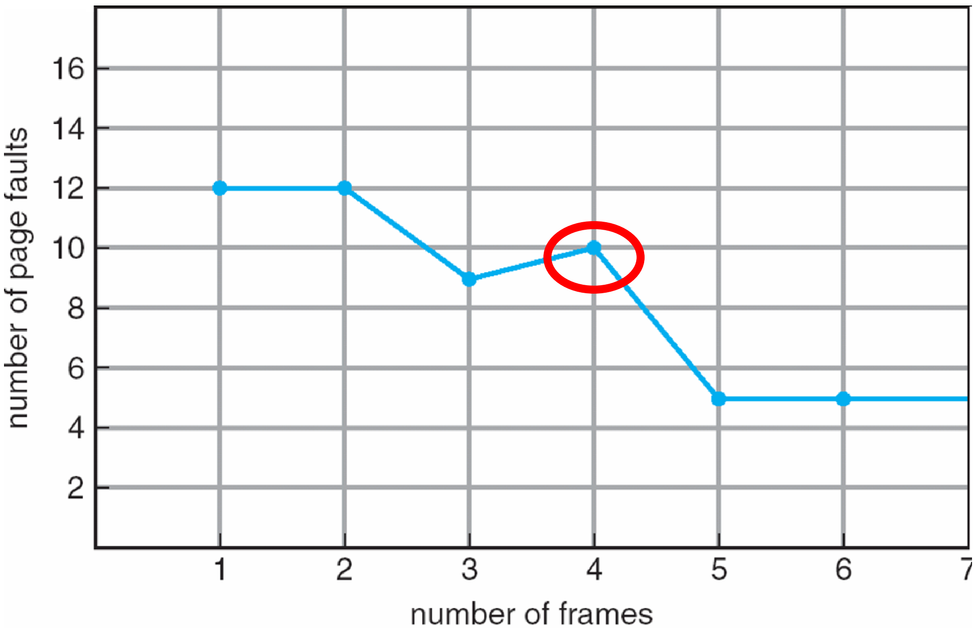
- Brought into memory longest time ago

Example

- Access trace (virtual page #): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages: 9 page faults
- 4 physical pages: 10 page faults



Belady's Anomaly



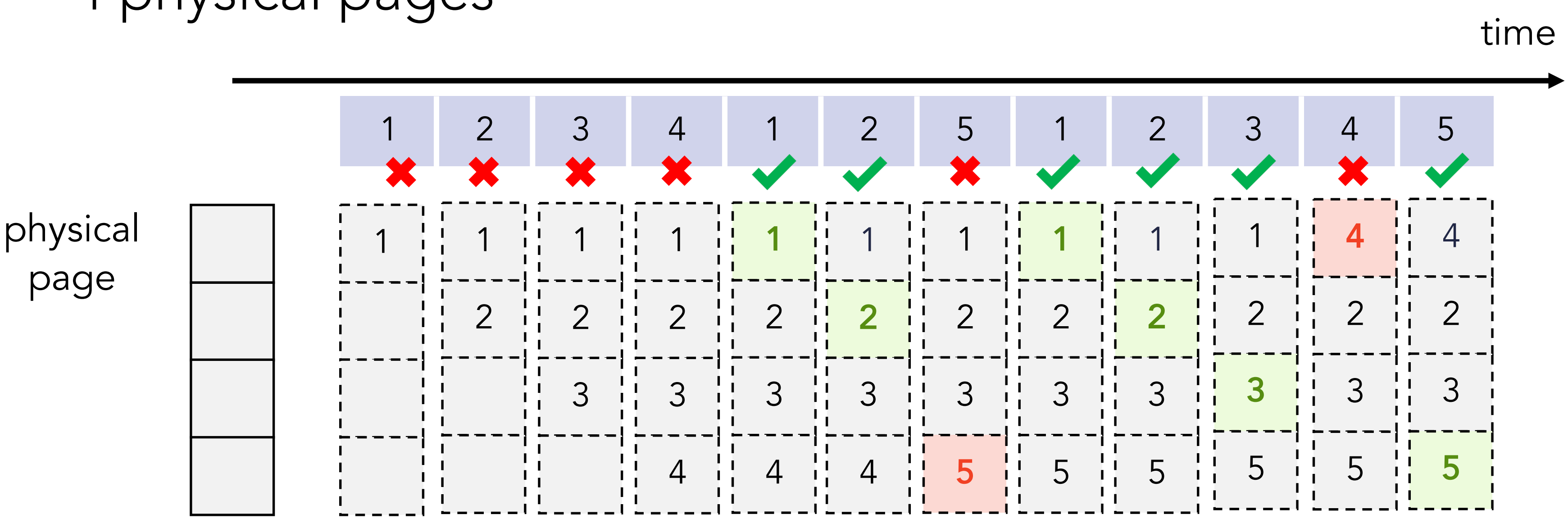
More physical memory doesn't always mean fewer faults!

Optimal replacement (OPT)

Replace page that will not be used for longest time in the future

Example

- Access trace (virtual page #): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 physical pages



6 page faults

Optimal replacement (OPT)

Belady's Algorithm

- Property: minimal page faults
- Rationale: the best page to evict is the one never touched again
  - Never is a long time, so picking the page closest to "never" is the next best thing
- Proved by Belady

Problem?

- Requiring knowing the future!

Why is Belady's algorithm useful then?

- Use it as a yardstick
- Compare page replacement algorithms with the optimal
  - If optimal is not much better, then algorithm is pretty good
  - If optimal is much better, then algorithm could use some work

Least recently used (LRU)

Replace the page that was last used longest ago

- If page hasn't been used for a while, it probably won't be used for a long time in the future

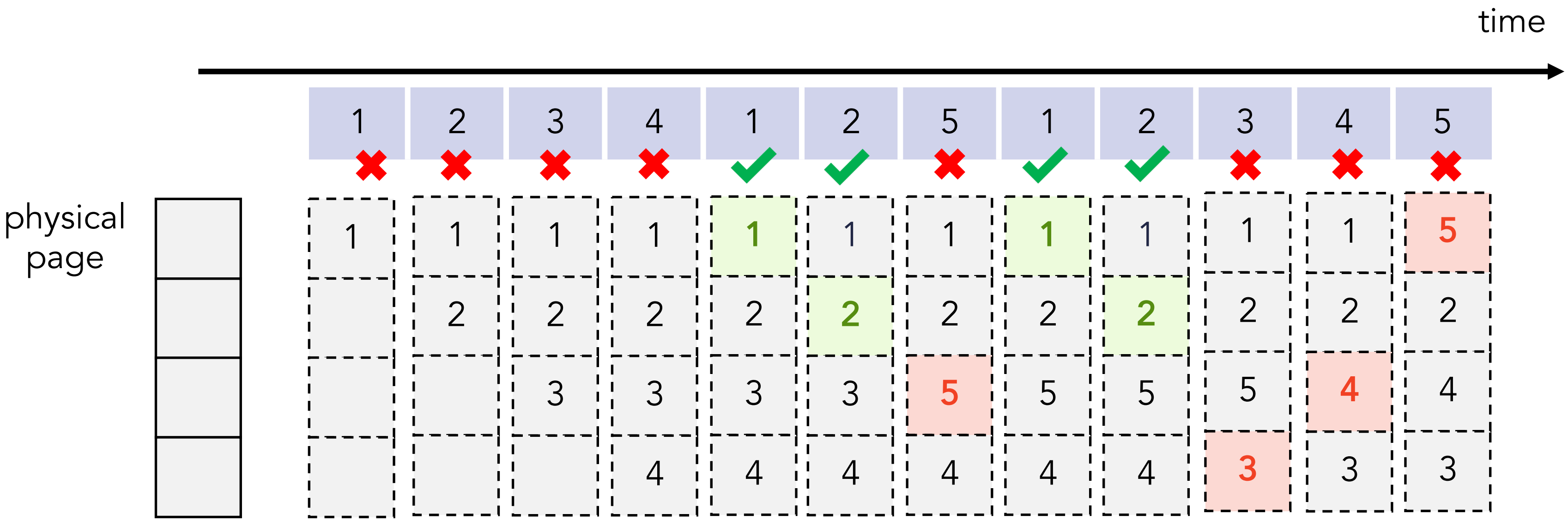
Approximates OPT

- Temporal locality: the future tends to reflect the past

Least recently used (LRU)

Example

- Access trace (virtual page #): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 physical pages 8 page faults



Problems

- Can be pessimal – example?
  - Looping over memory (then want MRU eviction)
- How to implement?

Strawman LRU Implementations

Stamp PTEs with timer value?

- E.g., CPU has cycle counter
- Automatically writes value to PTE on each page access
- Scan page table to find oldest counter value = LRU page
- Problem?
  - Would double memory traffic!

Keep doubly-linked list of pages?

- On access remove page, place at tail of list
- Problem: again, very expensive

What to do?

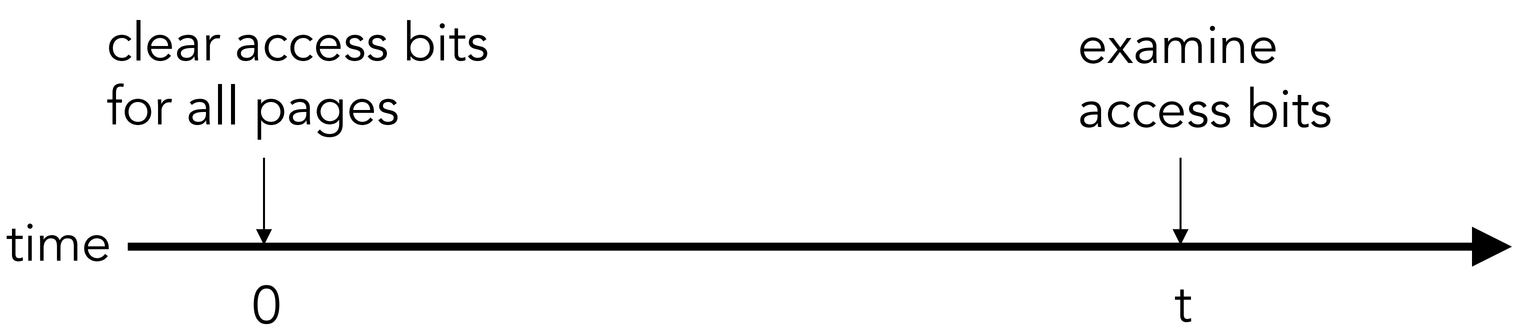
- Approximate LRU, don't try to do it exactly

Approximating LRU

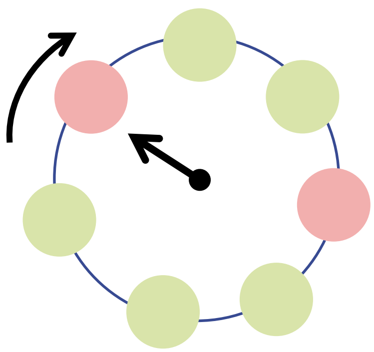
Most MMUs maintain an accessed/referenced bit

- In PTEs
- Set by MMU when a page is read or written
- Can be cleared by OS

How to use access bit to identify pages that have *not been used for a while*?



Clock algorithm



Do FIFO but skip accessed pages

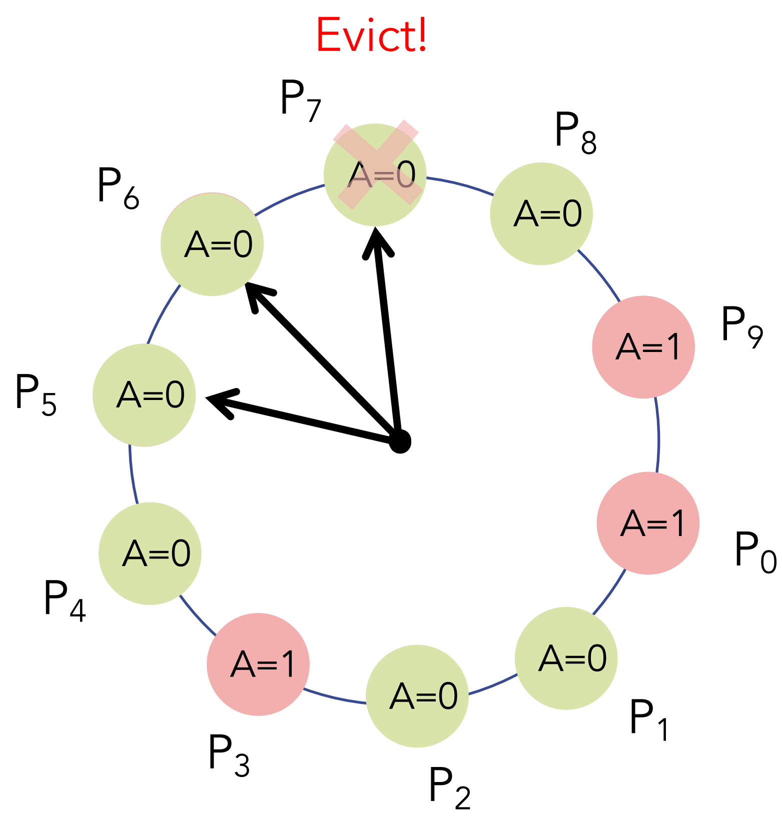
Keep resident pages in circular list

Scan: inspect page pointed by clock hand

- If access bit is set, this page has been accessed "recently"
  - Clear the bit, skip this page, advance clock hand
- If access bit is not set, this page has not been accessed "recently"
  - Evict

A.k.a. second-chance replacement

Clock example



What if all pages were referenced since last sweep?

Page eviction

What to do with data from evicted page? Why?

When do you not need to write page to disk?

This is a "write-back" cache

How to tell?

- Most MMUs provide a "dirty" bit for each resident page
- MMU sets "dirty" bit when page is written
- "dirty" means "content different from disk"

Why not write to disk on every store (write-through cache)?

Optimizing when to do work

do work earlier: may reduce waiting later

do work later: may reduce total work

time

When would you not need to write a new data value to disk?

Project 3 memory manager should be as lazy as possible (with CLOCK replacement)

MMU algorithm

read by MMU: physical page #, valid, resident, protection

written by MMU: dirty, referenced

PTE	physical page #	valid	resident	protection	dirty	referenced

```
if (virtual page is invalid or non-resident or protected) {
        trap to OS fault handler
        retry access
} else {
        physical page # = pageTable[virtual page #].physPageNum
        pageTable[virtual page #].referenced = true
        if (access is write) {
                pageTable[virtual page #].dirty = true
        }
        access physical memory at {physical page #}{offset}
}
```

Page table contents

PTE	physical page #	valid	resident	protection	dirty	referenced

```
if (virtual page is ~~invalid~~ or ~~non-resident~~ or protected) {
    trap to OS fault handler
    ...
```

Other information about page is stored in OS data structure (e.g., location on disk)

Do we *have to* keep a valid bit in PTE?

- Mark invalid pages as non-resident, then sort out specifics in OS after fault

Do we *have to* keep a resident bit in PTE?

- Mark non-resident pages as non-readable/non-writable, then sort out specifics in OS after fault

Page table contents

minimalist

PTE | physical page # | protection | dirty | referenced

Do we have to keep the dirty bit?

- Have OS (not MMU) maintain dirty bit in its own data structure
- Naive solution: trap on every store & mark dirty
  - improvement: which store instructions <u>change</u> "dirty" bit?

Do we have to keep the reference bit?

General pattern:

- What information are you maintaining?
- What accesses change this information?
- Set protection to trap on these accesses

H/W typically do keep more info than the minimal

dirty

accessed (reference)

present (resident)

31 ... 12	11	10	9	8	7	6	5	4	3	2	1	0
Physical page number	Ignored			G	AT	D	A	CD	WT	U	W	P

x86 PTE