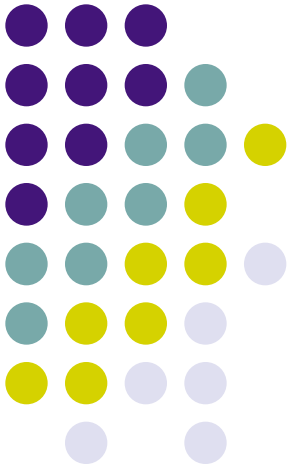
Virtual Memory, Demand Paging

ECE469, March 22

Yiying Zhang



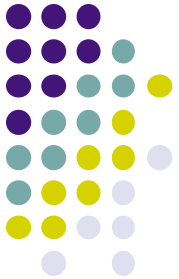
replacement policy not in midterm

demanding in midterm

Reading

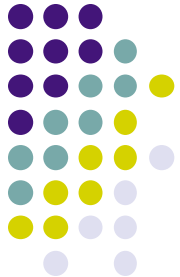
- Dinosaur: Chapter 8, 9
- Comet: Ch 21 22

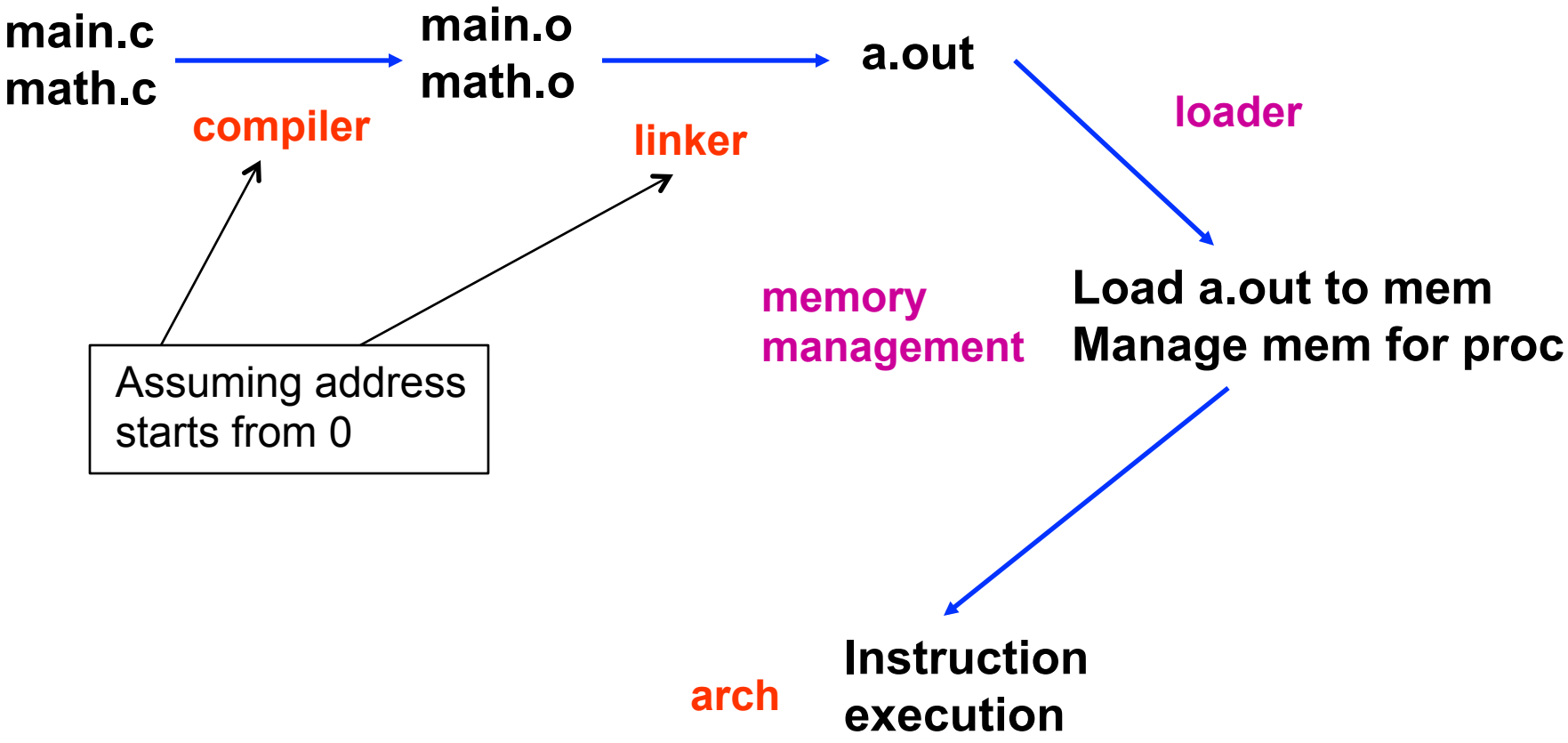
Rough Format of midterm



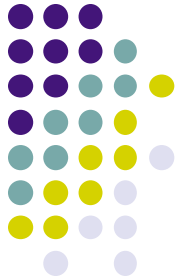
- 14 True or False (28 pts)
- 5 multiple choices (10 pts)
- 3 short-answer questions (12 pts)
- 3 big problems (16+24+10 pts)
  - Each big problem split into few smaller problems
  - Write (debug) some code

[lec13] The big picture – connecting the dots

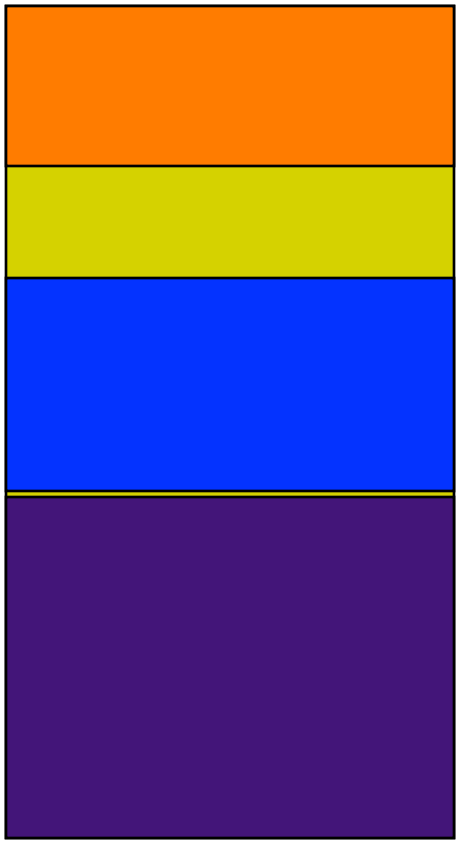




Review: sharing main memory

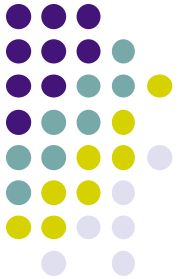


- Simple multiprogramming



4 drawbacks?

[lec15] Sharing main memory

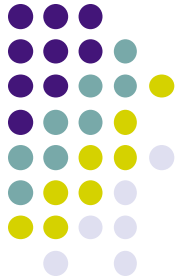


- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
  - Single segment per process

Dynamic Memory Relocation

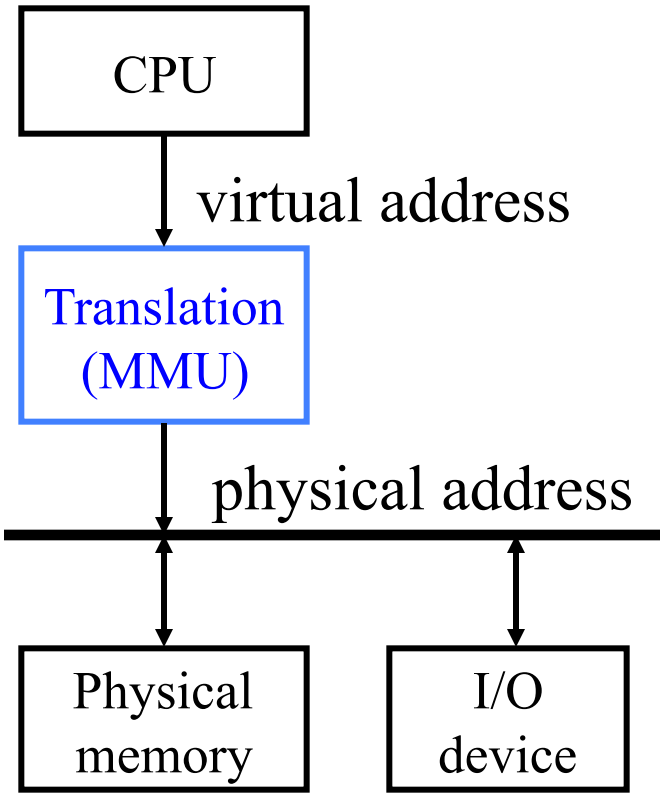
  - Entire address space needs to fit in mem

[lec15] 3. Dynamic memory relocation



- Instead of changing the address of a program before it's loaded, change the address dynamically *during every reference*
  - Under dynamic relocation, each program-generated address (called a *logical address* or *virtual address*) is translated in hardware to a *physical* or *real address* at runtime

[lec15] Translation overview

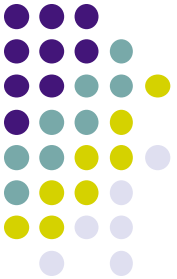


- Actual translation is in hardware (MMU)
- Controlled in software

- CPU view
  - what program sees, virtual memory
- Memory view
  - physical memory

What is the essence of going through MMU?

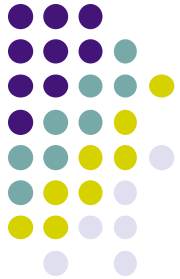
[lec15] Sharing main memory



- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
    → Base-and-bound
  - Single segment per process
    → Paging, segmentation, etc.
  - Entire address space needs to fit in mem

Dynamic Memory Relocation

[lec15] Sharing main memory



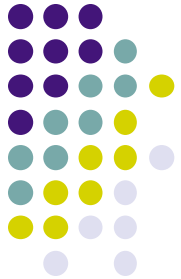
- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically

    → dynamic memory relocation: base&bound
  - Single segment per process

    → dynamic memory relocation: segmentation, paging

  - Entire address space needs to fit in mem
    - More need for swapping
    - Need to swap whole, very expensive!

The last drawback

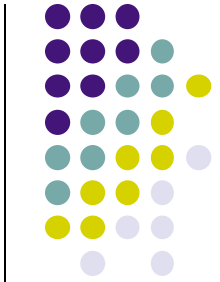


- So far we've separated the process's view of memory from the OS's view using a mapping mechanism
  - Each sees a different organization
  - Allows OS to shuffle processes around
  - Simplifies memory sharing
  - *What is the essence of the mechanism that enables this?*

- But, a user process had to be completely loaded into memory before it could run

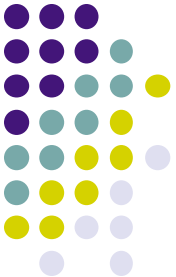
→ Wasteful since a process only needs a small amount of its total memory at any time (*reference locality!*)

Instances of extra level of indirection



- Dynamic memory relocation
  - Base and bound
  - Segmentation
  - 1-level paging
  - 2-level paging
  - Segmentation plus paging

[lec1] What is an OS?



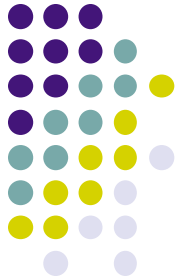
Extended (abstract) machine (answer 2)

- Much more ideal environment than the hardware
  - Ease to use
  - Fair (well-behaved)
  - Portable (back-compatible)
  - Reliable
  - Safe?

- Illusion of infinite, private resources
  - Single processor → many separate processors
  - Single memory → many separate, larger memories

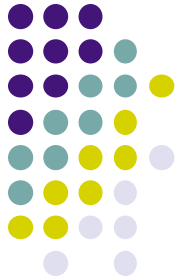
?

Virtual Memory



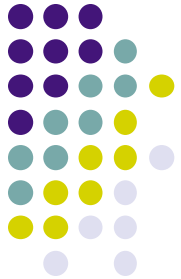
- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory
- Key idea: Virtual address space translated to either
  - Physical memory (small, fast) or
  - Disk (backing store), large but slow
- Deep thinking – what made above possible?
- Objective:
  - To produce the illusion of memory as big as necessary

Virtual Memory



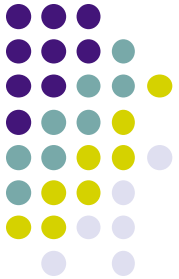
- “To produce the illusion of memory as big as necessary”
  - Without suffering a huge slowdown of execution
  - What makes this possible?
  - *Principle of locality*
    - Knuth’s estimate of 90% of the time in 10% of the code
    - There is also significant locality in data references

Virtual Memory Implementation



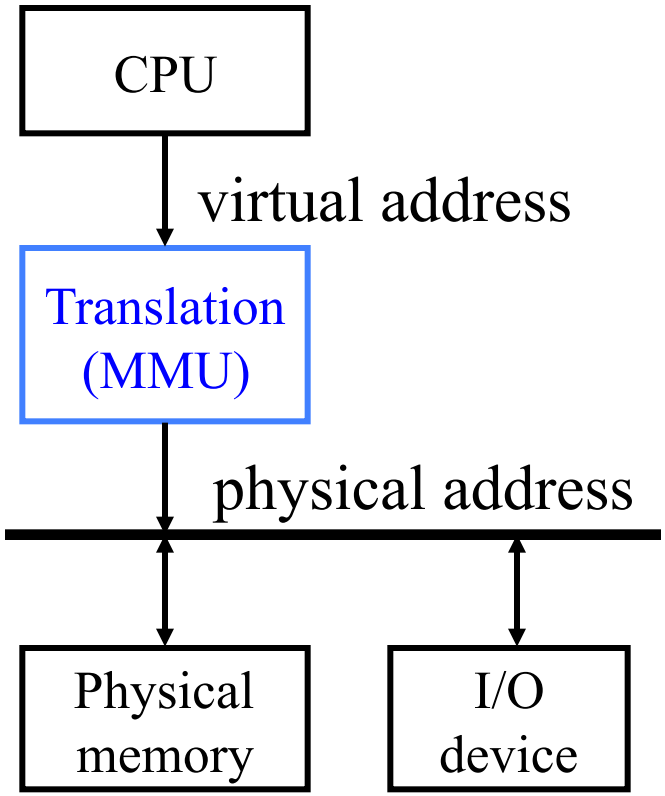
- Virtual memory is typically implemented via *demand paging*
- *demand paging:*
  - Load memory pages (from storage) "**on demand**"
  - paging with swapping, e.g., physical pages are swapped in and out of memory

Demand Paging (paging with swapping)

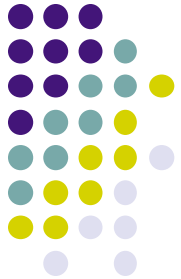


- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

- How to detect this?
  - In software?



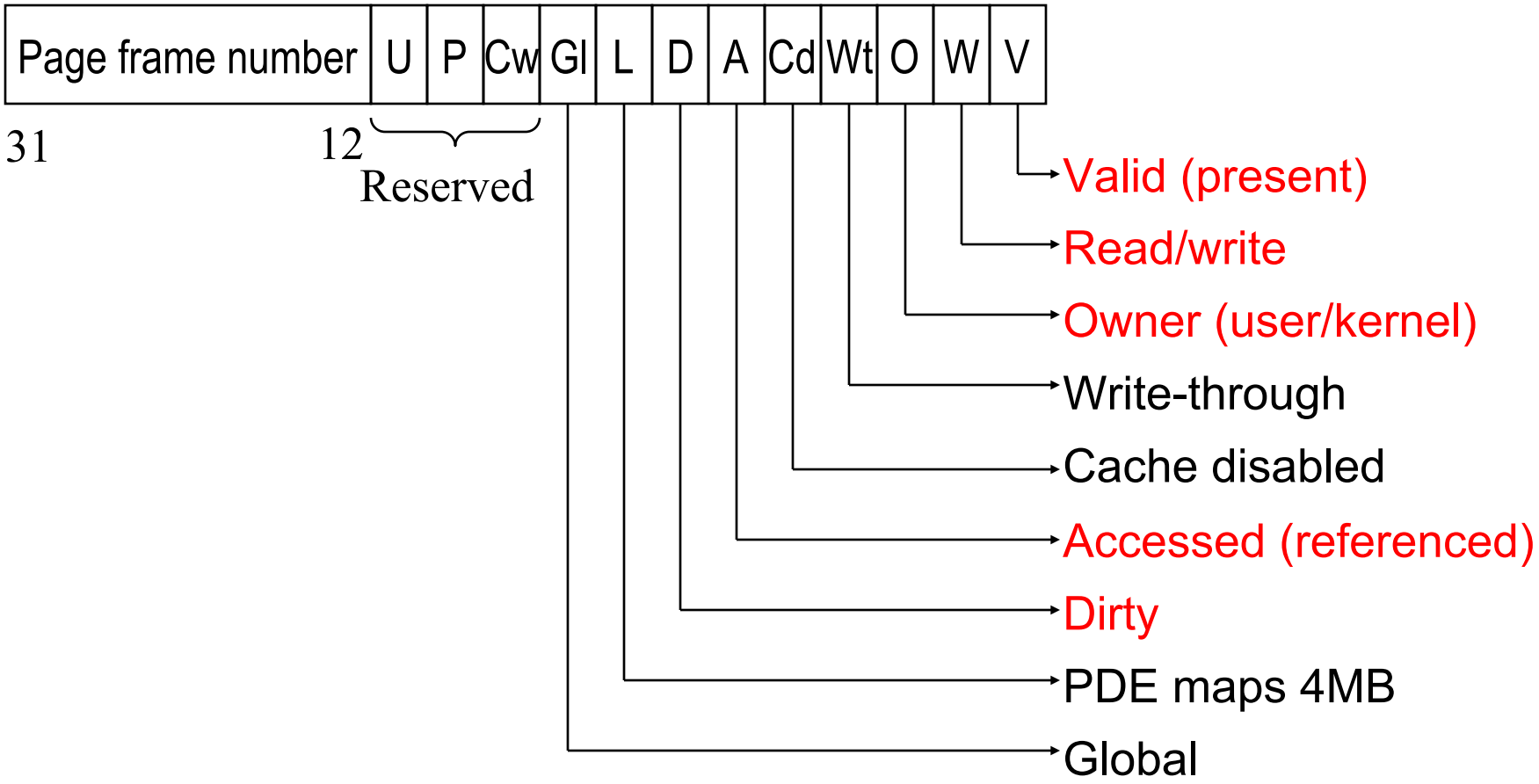
Demand Paging (paging with swapping)



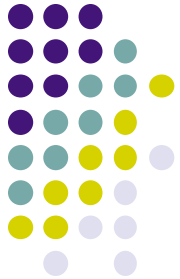
- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

- Hardware/software cooperate to make things work
  - Extend PTEs with an extra bit "present"
  - Any page not in main memory right now has the "present" bit cleared in its PTE
  - If "present" isn't set, a reference to the page results in a trap by the paging hardware, called *page fault*
  - What needs to happen when page fault occurs?

x86 Page Table Entry



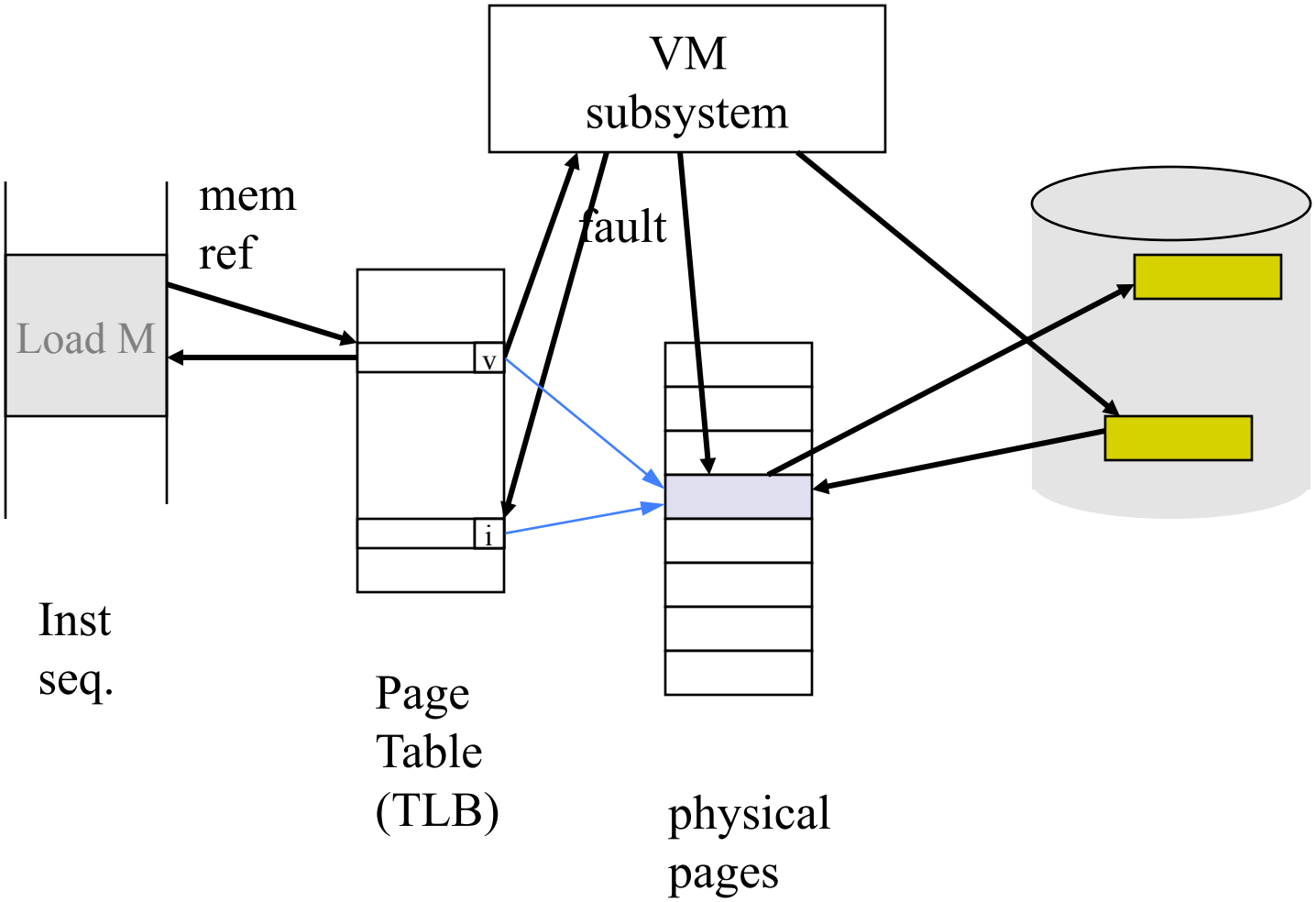
What is happening before and after malloc()?



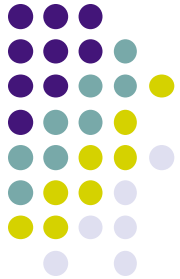
- Before malloc()?
- After malloc()?
- Upon first access?

- How to capture the first write to a virtual page?
  - e.g. want to trap into page fault handler
    - Use valid bit
  - In handler, check if vpage is malloced.
    - If not, segmentation fault
    - Else allocate physical page

Page Fault Handling in Demand Paging



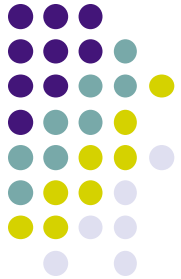
Page fault handling (cont)



- On a page fault
  - Find an unused phy. page or a used phy. page (how?)
  - If the phy. page is used
    - If it has been modified (how to know?), write it to disk
    - Invalidate its current PTE and TLB entry (how?)
  - Load the new page from disk
  - Update the faulting PTE and its TLB entry
  - Restart the faulting instruction

- Supporting data structure
  - For speed: A list of unused physical pages (more later)
  - Data structure to map a phy. page to its pid and virtual address
    - Sounds faimiliar?

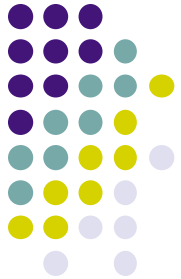
Deep thinking:
Virtual Address vs. Virtual Memory



- Virtual address
  - Supported with dynamic memory relocation
    - Segmentation
    - Paging

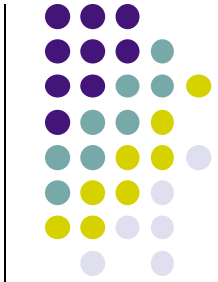
- Virtual memory
  - Dynamic memory relocation + swapping
    - Demand paging
    - Demand segmentation

Tasks of the VM subsystem



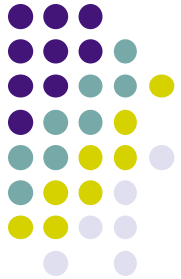
- Once the hardware has provided basic capabilities for VM, OS must make two kinds of decisions
  - Page selection: when to bring pages into memory
  - Page replacement: which page(s) should be thrown out, and when

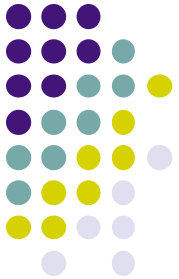
Page selection (when)



- Prepaging
  - Allocate physical page and (if from swap) bring a page into memory before it is referenced
  - Hard to do without a “prophet”, may unnecessarily bring pages
- Request paging
  - Let user say which pages are needed when
  - Users don’t always know best
  - And aren’t always impartial
- Demand paging
  - Start up process with no pages loaded
  - Load a page when a page fault occurs, i.e., wait till it MUST be in memory
  - Almost all paging systems are demand paging

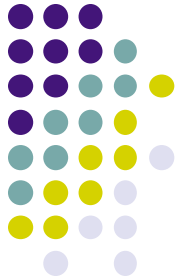
break





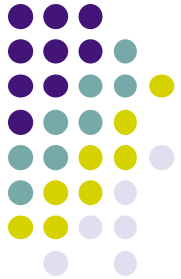
Names Every Systems Student Should Know

Why Are We Doing This?



- Really, you should know these people!
- They did some great work!
- Their work (still) have influence now

How Are We Going to Do This?



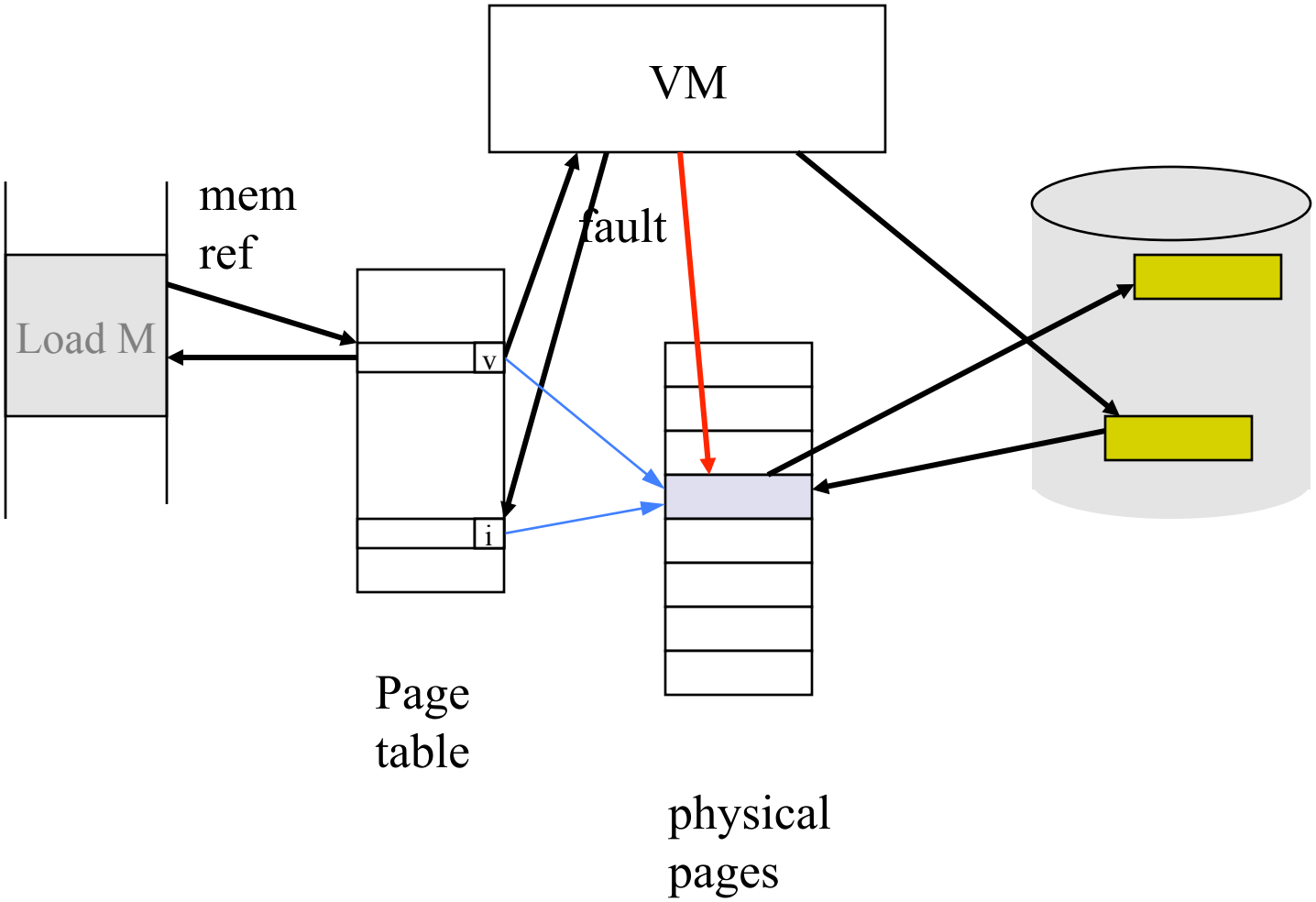
- One to two per lecture
- Turing award winners related to systems
- Scientists who have made contributions comparable to a turing award
- People who are still active (and alive)
  - Mainly systems people, some other
  - Academia and industry
- Disclaimer: The list is definitely incomplete. I may miss some, doesn't mean they are not important!

Alan Turing

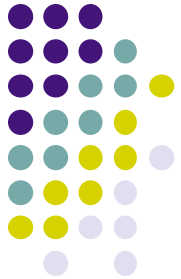
Edsger Dijkstra

- Algorithms/systems
- Father of algorithms
- the “THE” OS

Page replacement algorithms? (which)



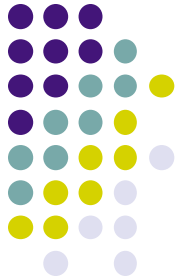
The BIG picture: Running at Memory Capacity



- Expect to run with all phy. pages in use
- Every "page-in" requires an eviction
- Goal of page replacement
  - Maximize hit rate → kick out the page that's least useful
- Challenge: how do we determine utility?
  - Kick out pages that aren't likely to be used again

- Page replacement is a difficult policy problem

What makes finding the least useful page hard?

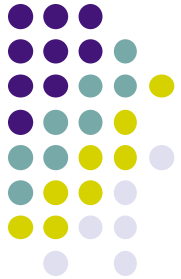


- Don't know future!
- Past behavior is a good indication of future behavior! (e.g. LRU)
  - temporal locality → kick out pages that have been used recently
- Perfect (past) reference stream hard to get
  - Every memory access would need bookkeeping
  - Is this feasible (in software? In hardware?)
- Minimize overhead
  - If no memory pressure, ideally no bookkeeping
  - In other words, make the common case fast (page hit)

→ Going after imperfect information, available cheaply

- What is minimum hardware support that need to added?

Definitions
(or, jargons asked during interviews)



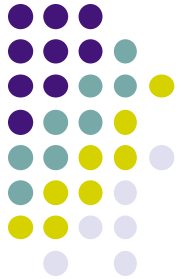
- **Pressure** – the demand for some resource (often used when demand exceeds supply)

  ex: the system experienced memory pressure
- **Eviction** – throwing something out

  ex: cache lines and memory pages got evicted
- **Pollution** – bringing in useless pages/lines

  ex: this strategy causes high cache pollution

More definitions



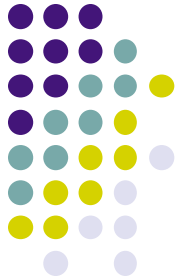
- **Thrashing** – extremely high rate of paging, usually induced by other decisions

- **Locality** – re-use – it makes the world go rounds!
- **Temporal Locality** – re-use in time
- **Spatial Locality** – re-use of close by locations

Performance metric for page replacement policies

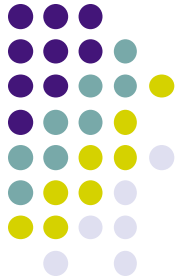
- Give a sequence of memory accesses, minimize the # of page faults
  - Similar to cache miss rate
  - What about hit latency and miss latency?

Optimal or MIN

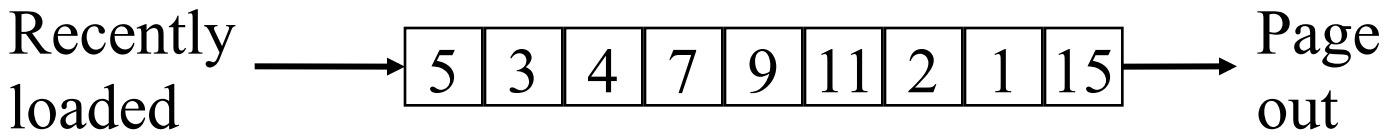


- Algorithm (also called Belady's Algorithm)
  - Replace the page that won't be used for the **longest** time
- Pros
  - Minimal page faults (can you prove it?)
  - Used as an **off-line** algorithm for perf. analysis
- Cons
  - **No on-line** implementation
- What was the CPU scheduling algorithm of similar nature?

What can we do without extra hardware support?

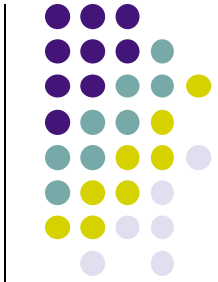


First-In-First-Out (FIFO)



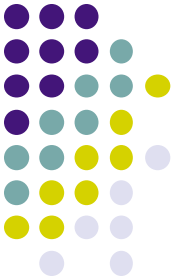
- Algorithm
  - Throw out the oldest page
- Pros
  - Low-overhead implementation
- Cons
  - No frequency/no recency → may replace the heavily used pages

Predicting future based on past



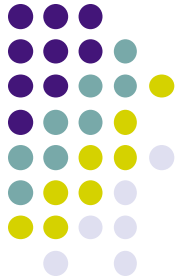
- "Principle of locality"
  - Recency:
    - Page recently used are likely to be used again in the near future
  - Frequency:
    - Pages frequently used (recently) are likely to be used frequently again in the near future
- Is this temporal or spatial locality?
- Why not spatial locality?

How to record locality in the past?



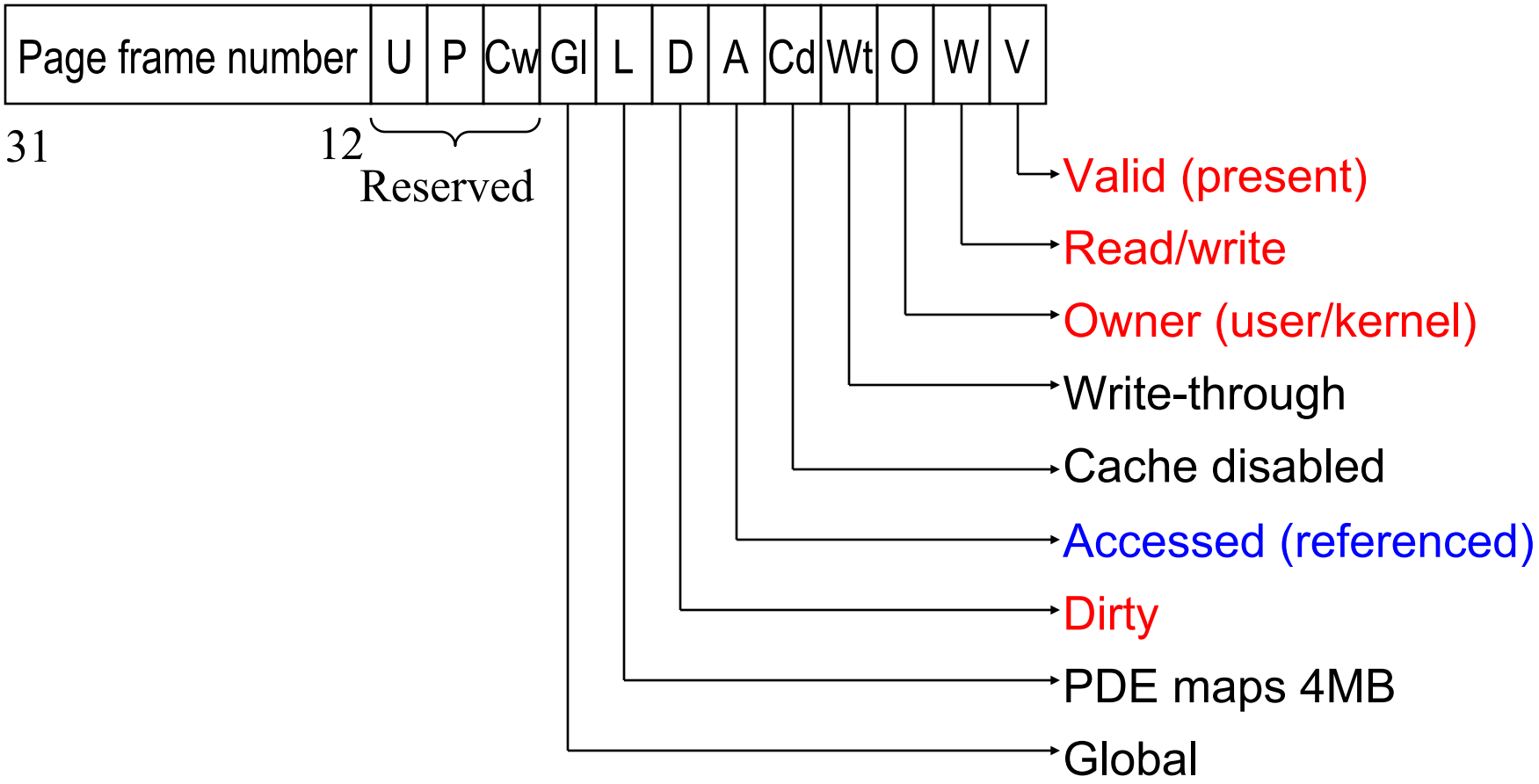
- Software solution?

Exploiting locality needs some hardware support

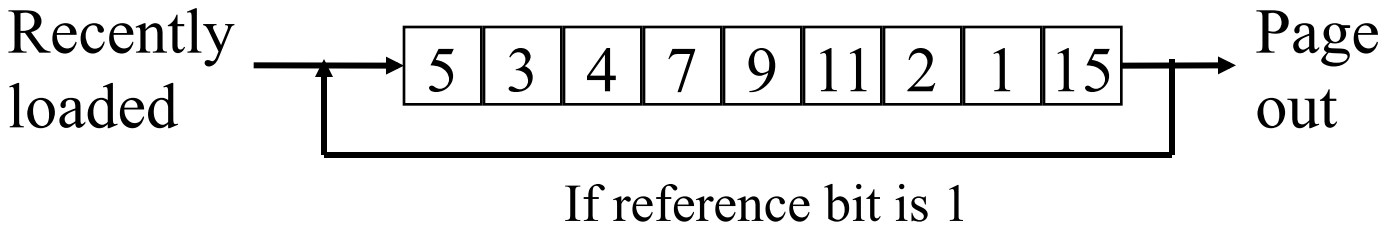


- Reference bit
  - A hardware bit that is set whenever the page is referenced (read or written)

x86 Page Table Entry

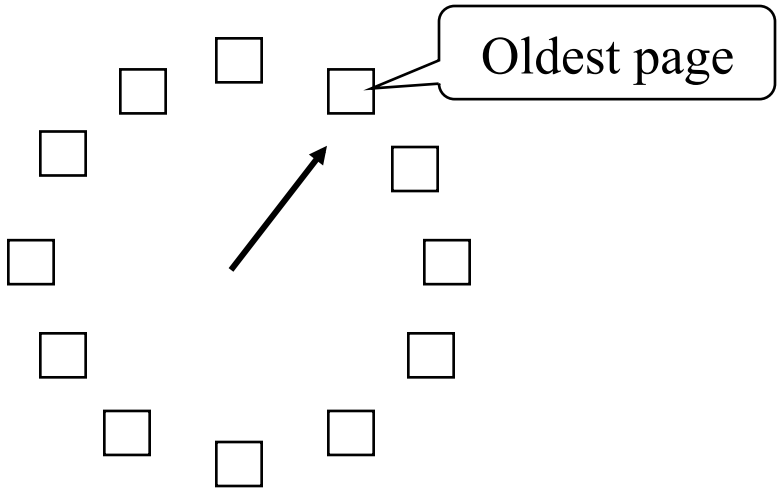


FIFO with Second Chance



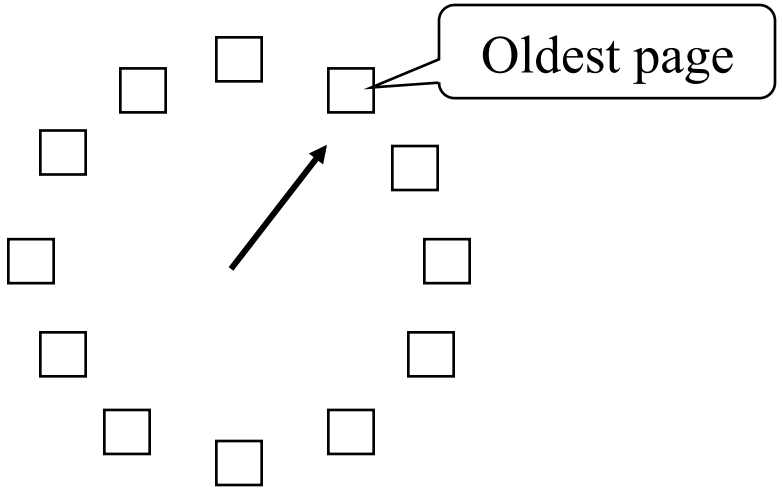
- Algorithm
  - Check the reference-bit of the oldest page (first in)
  - If it is 0, then replace it
  - If it is 1, clear the referent-bit, put it to the end of the list, and continue searching
- Pros
  - Fast
  - Frequency → do not replace a heavily used page
- Cons
  - The worst case may take a long time

Clock: a simple FIFO with 2nd chance



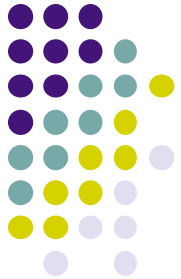
- FIFO clock algorithm
  - Hand points to the oldest page
  - On a page fault, follow the hand to inspect pages
- Second chance
  - If the reference bit is 1, set it to 0 and advance the hand
  - If the reference bit is 0, use it for replacement
- What is the difference between Clock and the previous one?
  - Mechanism vs. policy?

Clock: a simple FIFO with 2nd chance



- What happens if all reference bits are 1?
- What does it suggest if observing clock hand is sweeping very fast?
- What does it suggest if clock hand is sweeping very slow?

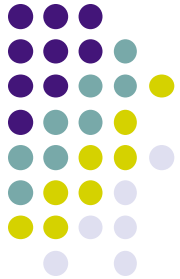
We've focused on miss rate. What about miss latency?



- Key observation: it is cheaper to pick a "clean" page over a "dirty" page
  - Clean page does not need to be swapped to disk

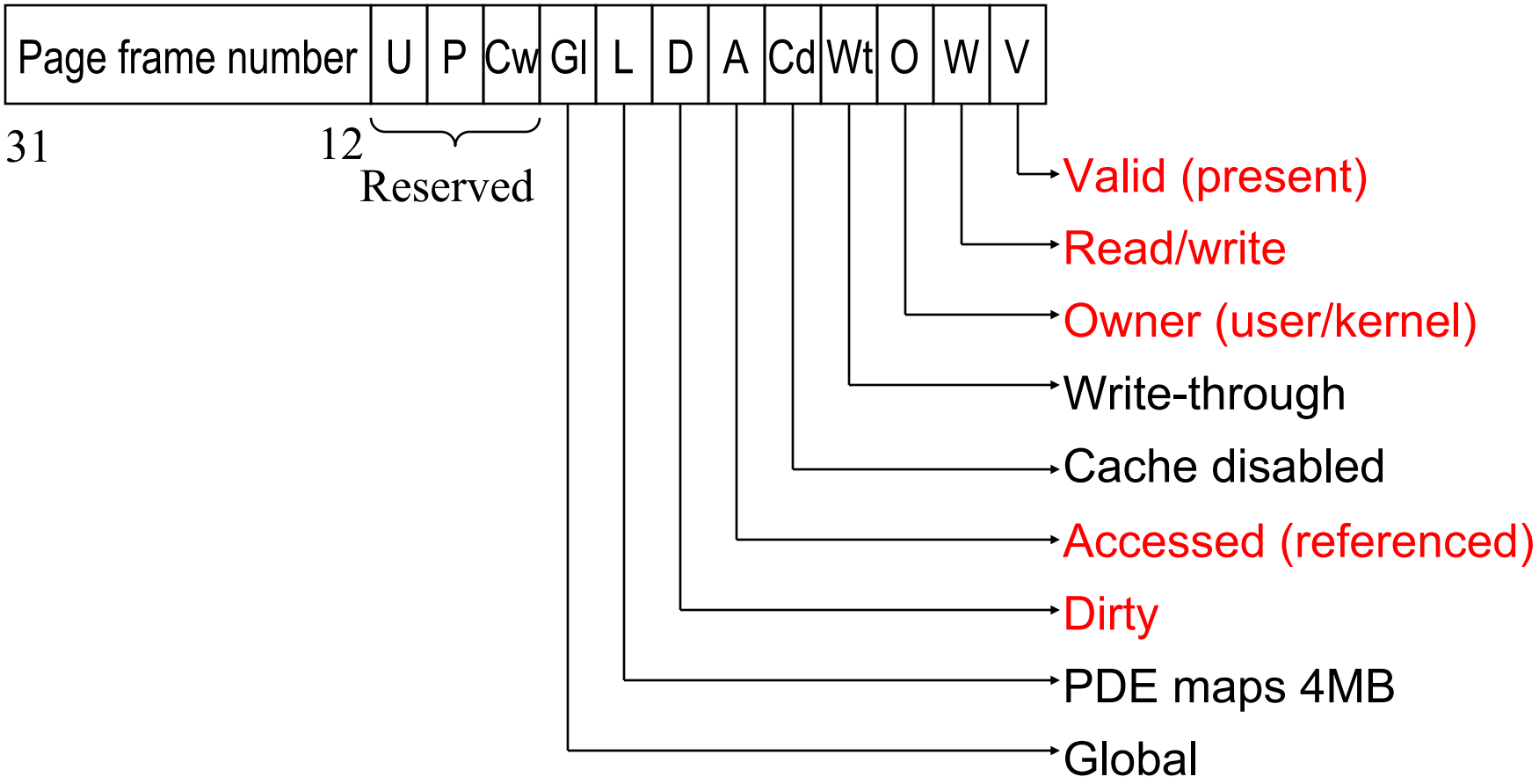
- Challenge:
  - How to get this info?

Refinement by adding extra hardware support

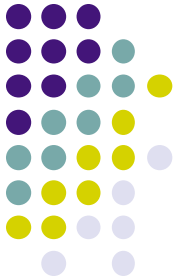


- Reference bit
  - A hardware bit that is set whenever the page is referenced (read or written)
- Modified bit (dirty bit)
  - A hardware bit that is set whenever the page is written into

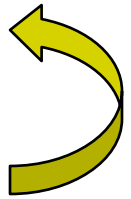
x86 Page Table Entry



Enhanced FIFO with 2nd-Chance Algorithm (used in Macintosh VM)

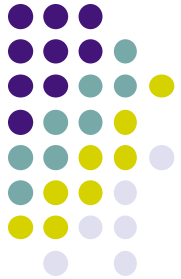


- Same as the basic FIFO with 2nd chance, except that it considers both (reference bit, modified bit)
  - (0,0): neither recently used nor modified (good)
  - (0,1): not recently used but dirty (not as good)
  - (1,0): recently used but clean (not good)
  - (1,1): recently used and dirty (bad)
  - When giving second chance, only clear reference bit



- Pros
  - Avoid write back
- Cons
  - More complicated, worse case scans multiple rounds

Enhanced FIFO with 2nd-Chance Algorithm – implementation



- On page fault, follow hand to inspect pages:
  - Round 1:
    - If bits are (0,0), take it
    - if bits are (0,1), record 1st instance
    - Clear ref bit for (1,0) and (1,1), if (0,1) not found yet
  - At end of round 1, if (0,1) was found, take it
  - If round 1 does not succeed, try 1 more round