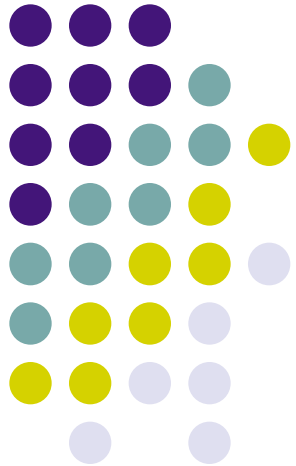
Page Replacement, Thrashing

ECE469, March 21

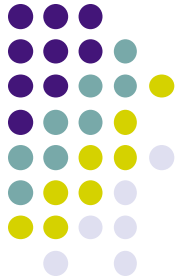
Yiying Zhang



Reading

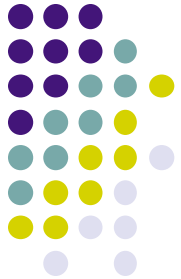
- Chapter 9

[lec16] The Policy Issue: Page selection and replacement



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

[lec 17] Page replacement problem



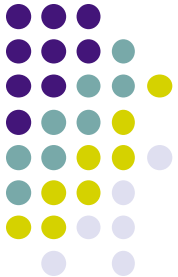
Definition:

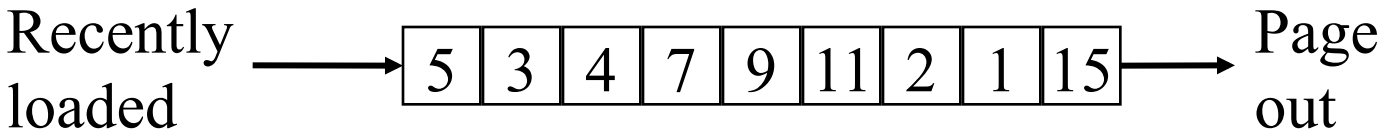
- Expect to run with all physical pages in use
- Every "page-in" requires an eviction to swap space
- How to select the victim page?

- Performance goal:
  - Give a sequence of memory accesses, minimize the # of page faults
  - → given a sequence of virtual page references, minimize the # of page faults

- Intuition: kick out the page that is least useful
- Challenge: how do you determine "least useful"?
  - Even if you know future?

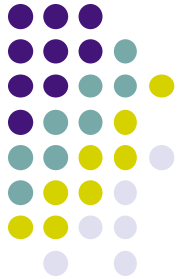
[lec16] First-In-First-Out (FIFO)





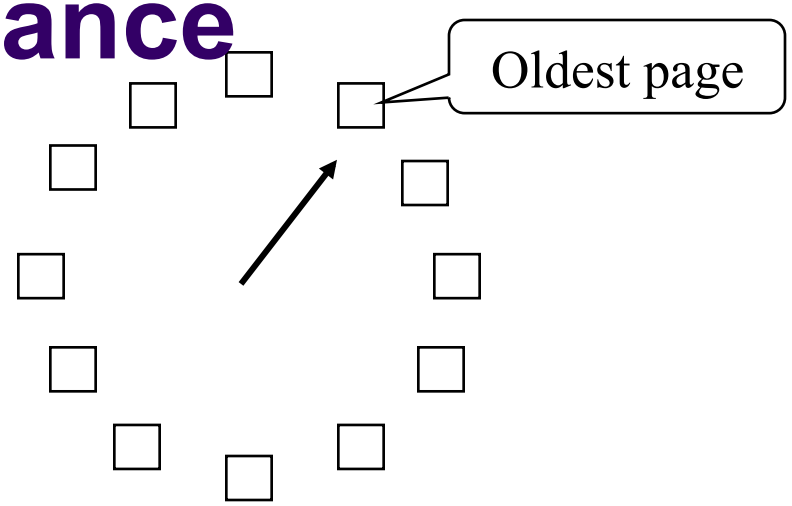
- Algorithm
  - Throw out the oldest page
- Pros
  - Low-overhead implementation
- Cons
  - No frequency/no recency → may replace the heavily used pages

[lec17] Belady's anomaly



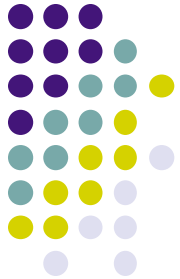
Belady's anomaly states that it is possible to have more page faults when increasing the number of page frames while using FIFO method of frame management. Laszlo Belady demonstrated this in 1970. Previously, it was believed that an increase in the number of page frames would always provide the same number or fewer page faults.

[lec16] Clock: a simple FIFO with 2nd chance



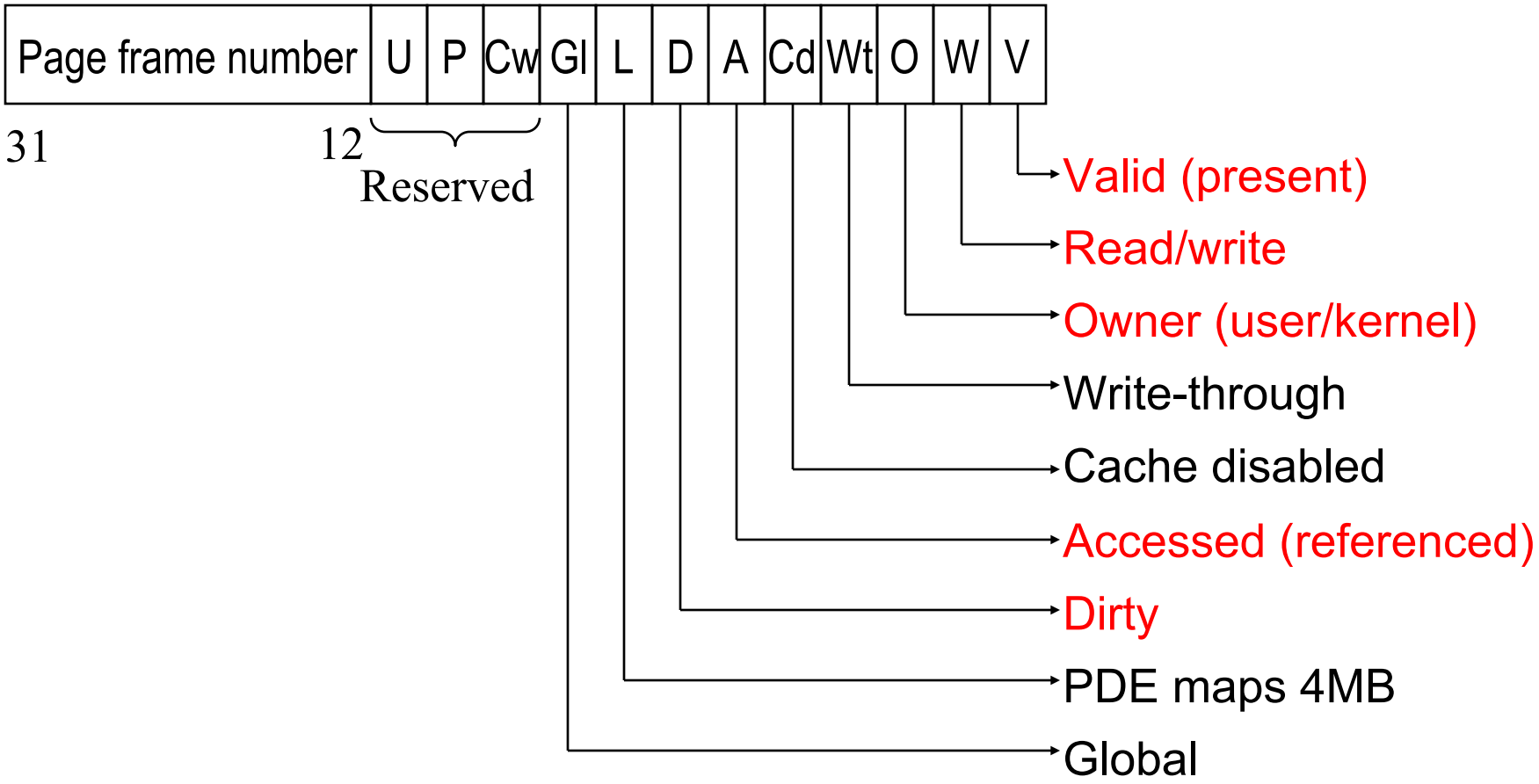
- FIFO clock algorithm
  - Hand points to the oldest page
  - On a page fault, follow the hand to inspect pages
- Second chance
  - If the reference bit is 1, set it to 0 and advance the hand
  - If the reference bit is 0, use it for replacement
- What is the difference between Clock and the previous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

Refinement by adding extra hardware support

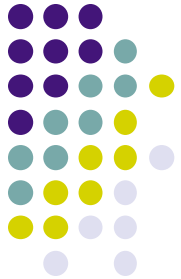


- Reference bit
  - A hardware bit that is set whenever the page is referenced (read or written)
- Modified bit (dirty bit)
  - A hardware bit that is set whenever the page is written into

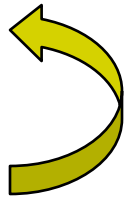
x86 Page Table Entry



[lec17] Enhanced FIFO with 2nd-Chance Algorithm (used in Macintosh VM)

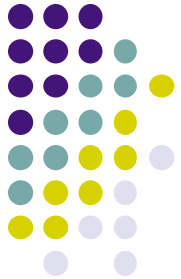


- Same as the basic FIFO with 2nd chance, except that it considers both (reference bit, modified bit)
  - (0,0): neither recently used nor modified (good)
  - (0,1): not recently used but dirty (not as good)
  - (1,0): recently used but clean (not good)
  - (1,1): recently used and dirty (bad)
  - When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



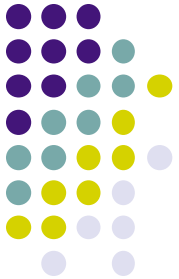
- Pros
  - Avoid write back
- Cons
  - More complicated, worse case scans multiple rounds

[lec17] Enhanced FIFO with 2nd-Chance Algorithm – implementation



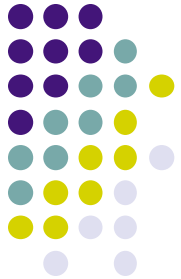
- On page fault, follow hand to inspect pages:
  - Round 1:
    - If bits are (0,0), take it
    - if bits are (0,1), record 1st instance
    - Clear ref bit for (1,0) and (1,1), if (0,1) not found yet
  - At end of round 1, if (0,1) was found, take it
  - If round 1 does not succeed, try 1 more round

Least Recently Used (LRU)



- Algorithm
  - Replace page that hasn't been used for the longest time
- Advantage: with locality, LRU approximates Optimal
- Question
  - What hardware mechanisms are required to implement *exact* LRU?

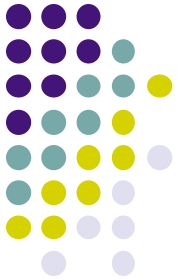
Implementing LRU: hardware

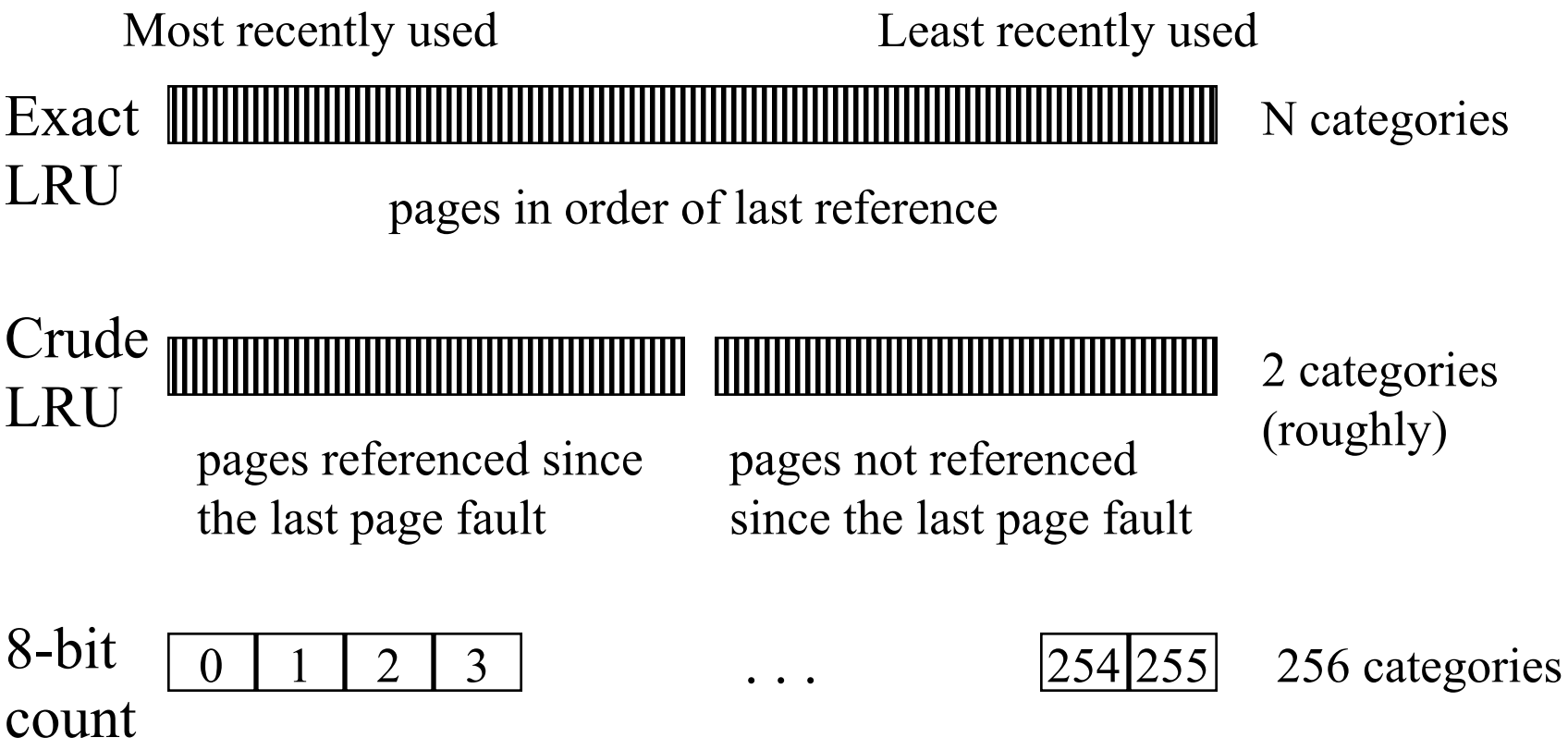


- A counter for each page
- Every time page is referenced, save system clock into the counter of the page
- Page replacement: scan through pages to find the one with the oldest clock

- Problem: have to search all pages/counters!

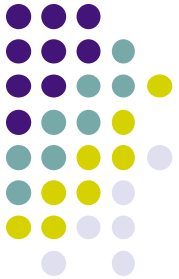
Approximate LRU





Keep 8-bit counter for each page in a table in memory

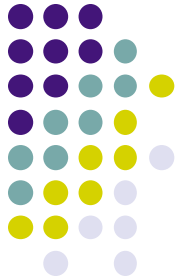
Approximate LRU



Interval 1	Interval 2	Interval 3	Interval 4	Interval 5
00000000	00000000	10000000	01000000	10100000
00000000	10000000	01000000	10100000	01010000
10000000	11000000	11100000	01110000	00111000
00000000	00000000	00000000	10000000	01000000

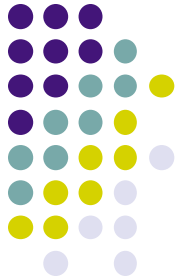
- Algorithm
  - At regular interval, OS shifts reference bits (in PTE) into counters (and clear reference bits)
  - Replacement: Pick the page with the "smallest counter"
- How many bits are enough?
  - In practice 8 bits are quite good
- Pros: Require one reference bit, small counter/page
- Cons: Require looking at many counters (or sorting)

Implementing LRU: software



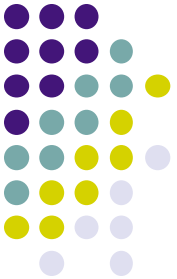
- A doubly linked list of pages
- Every time page is referenced, move it to the front of the list
- Page replacement: remove the page from back of list
  - Avoid scanning of all pages
- Problem: too expensive
  - Requires 6 pointer updates for each page reference info
  - High contention on multiprocessor

Not Recently Used (NRU)



- Algorithm
  - Randomly pick a page from the following (in this order)
    - Not referenced and not modified
    - Not referenced and modified
    - Referenced and not modified
    - Referenced and modified
- Pros
  - Easy to implement
- Cons
  - No frequency → Not very good performance
  - Takes time to classify

Page replacement algorithms: Summary



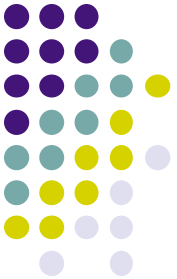
- Optimal

- FIFO
- Random

- Approximate LRU (NRU)

- FIFO with 2nd chance
- Clock: a simple FIFO with 2nd chance

Thinking



- Performance design philosophy:
- Make common case fast!

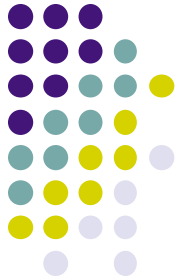
- Amadhl's Law

$$\text{Overall Speedup} = \frac{1}{(1-F)+\frac{F}{S}}$$

F = The fraction enhanced

S = The speedup of the enhanced fraction

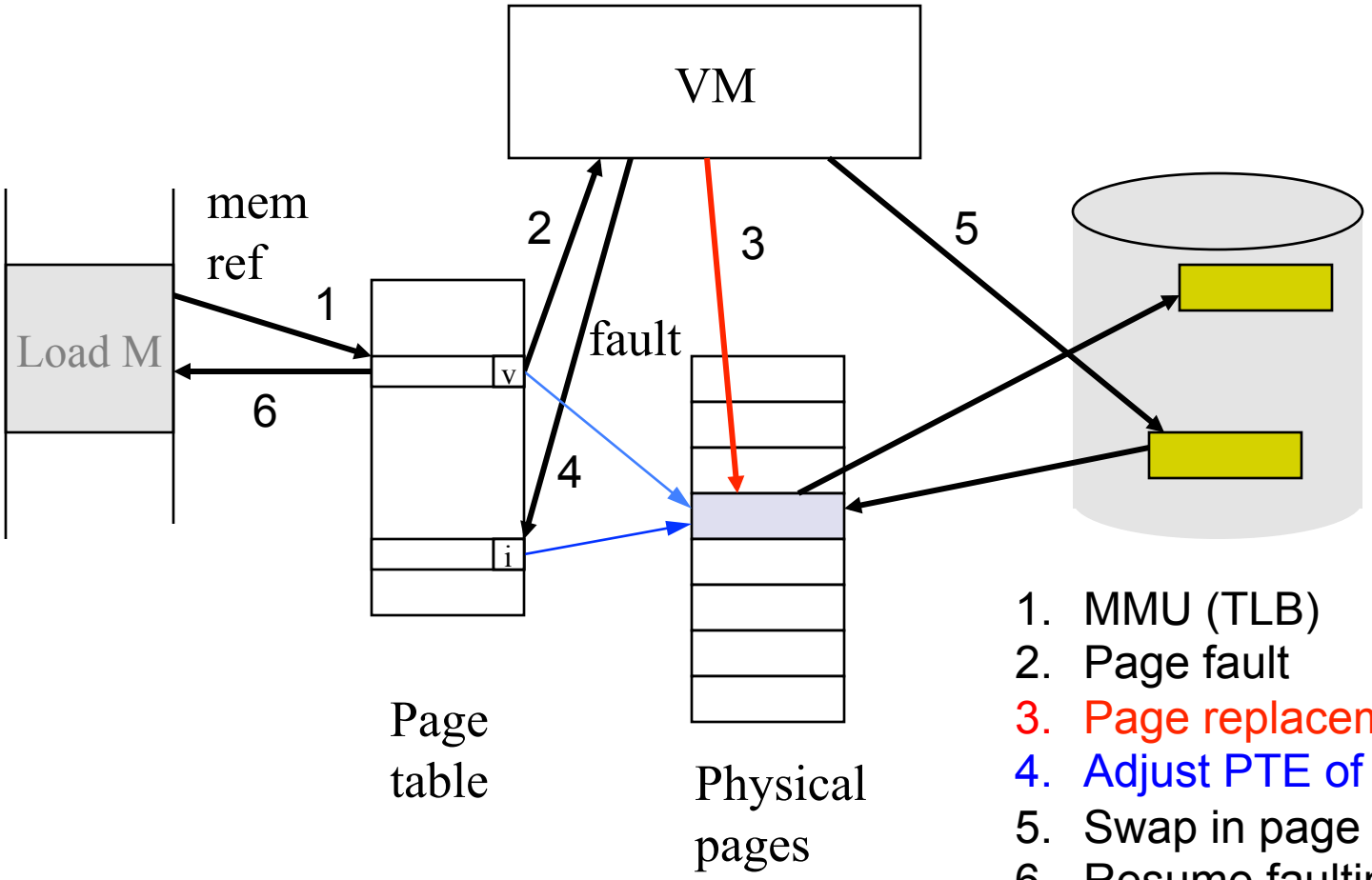
Factors that affect paging performance



- # of memory misses
  - So far, mostly have talked about algorithms to reduce misses (increase hit rate)

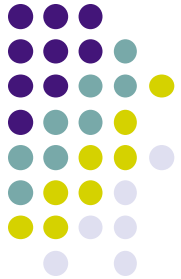
- Cost of a memory miss (replacement)

[lec16] Page Fault Handling in demand paging



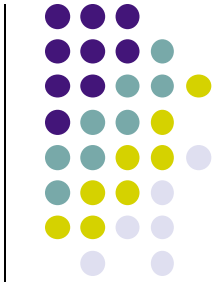
1. MMU (TLB)
2. Page fault
3. Page replacement
4. Adjust PTE of victim pg
5. Swap in page from swap
6. Resume faulting intr

Page out on critical path?



- If no free page pool, page in has to wait till page out is finished
  - Page fault handling time = proc. overhead + 2 * I/Os
- There is a chance of paged out page being referenced soon

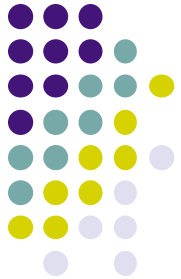
Page buffering techniques



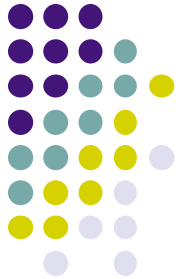
1. System maintains a pool of free pages
   - When a page fault occurs, victim page chosen as before
   - But desired page paged into a free page right away before victim page paged out
   - Dirty victim page written out when disk is idle

2. Also remember what was in free pages - maybe reused before reallocated

break

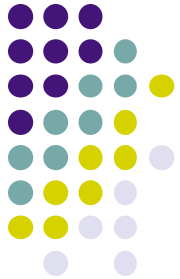


Linus Torvalds



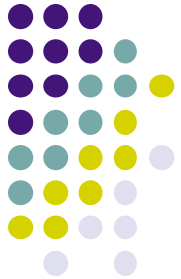
- Linux
- git

Andrew Tanenbaum

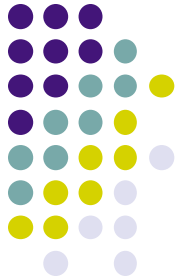


- Minix
- OS textbooks

Thrashing

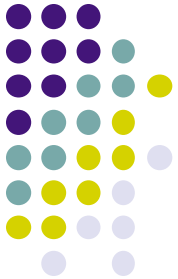


The BIG picture



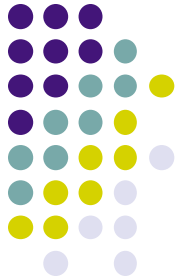
- We've talked about single evictions
- Most computers are multiprogrammed
  - Single eviction decision still needed
  - New concern – processes compete for resources
  - How to be "fair enough" and achieve good overall throughput

Possible replacement strategies



- Global replacement:
  - All pages from all processes are lumped into a single replacement pool
  - Most flexibility, least (performance) isolation
- Local replacement
  - Per-process replacement:
    - Each process has a separate pool of pages
  - Per-user replacement:
    - Lump all processes for a given user into a single pool
- In local replacement, must have a mechanism for (slowly) changing the allocations to each pool

[lec17] Virtual Memory

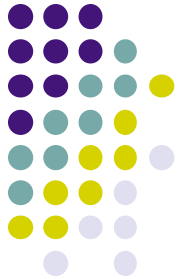


- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory

- VM is typically implemented by demand paging
  - Virtual address space translated to either
    - Physical memory (small, fast) or
    - Disk (backing store), large but slow

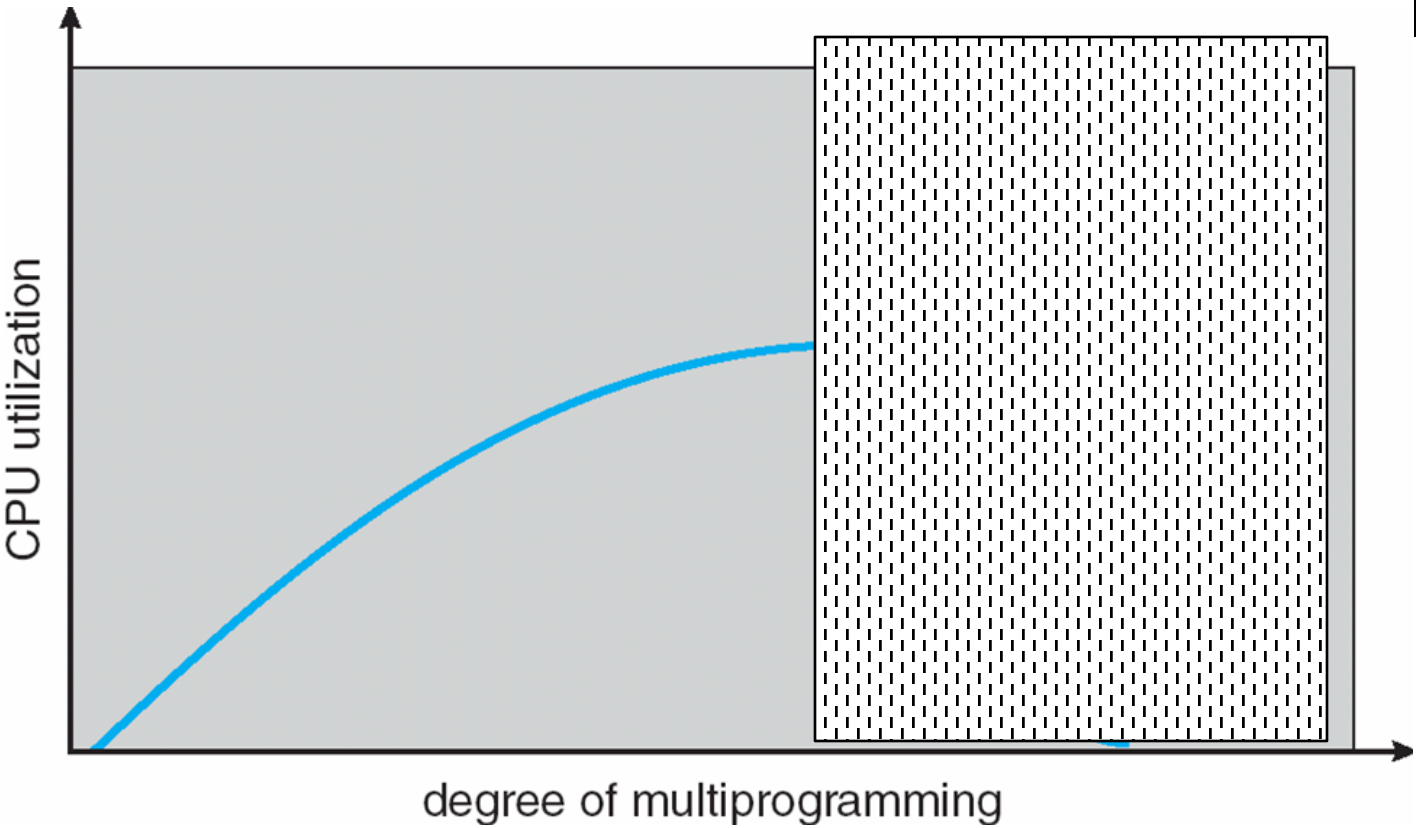
- Objective:
  - To produce the illusion of memory as big as necessary

Improving CPU utilization in multiprogramming

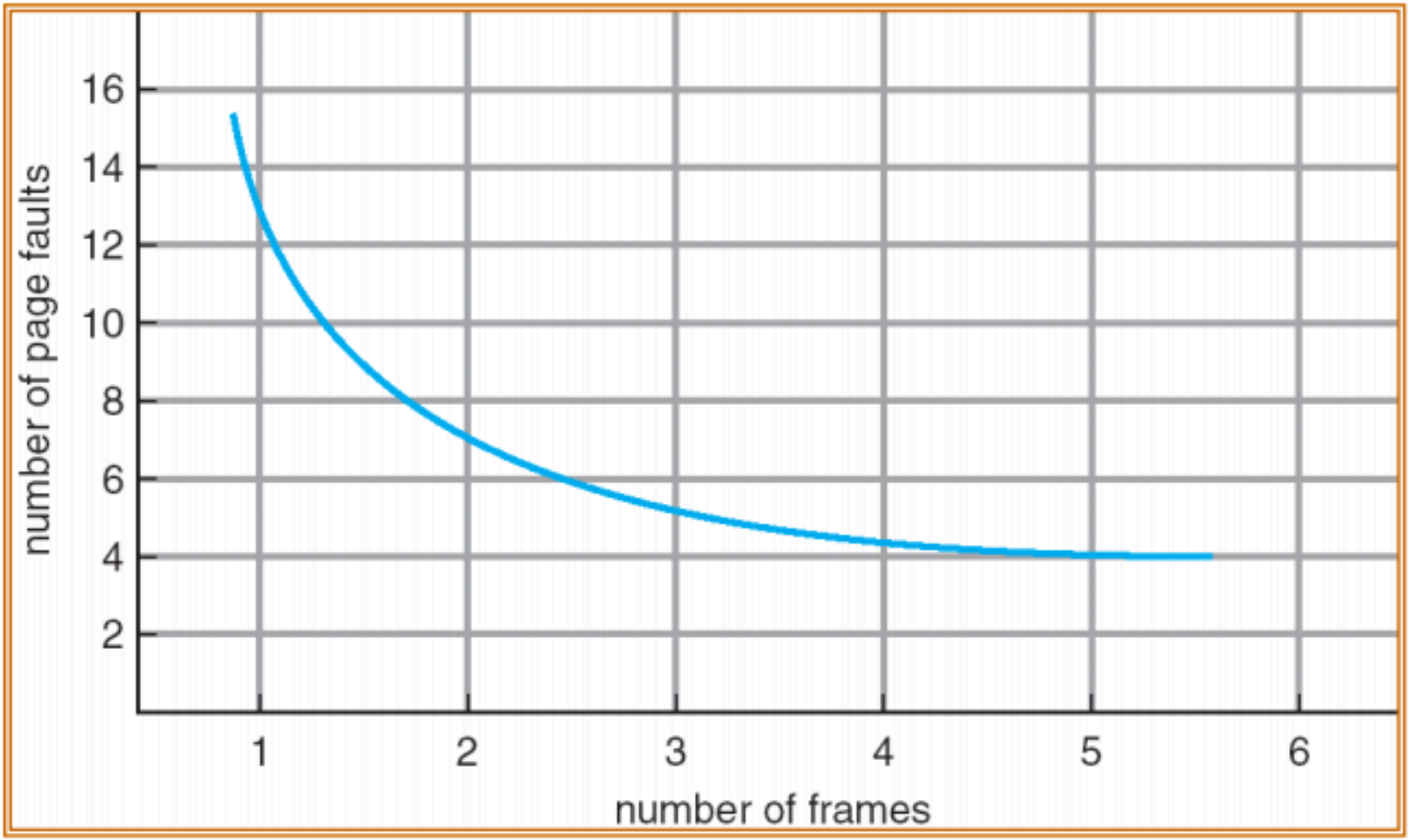


- In multiprogramming, when OS sees the CPU utilization is low,
  - It thinks most processes are waiting for I/O
  - it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - It adds/loads another process to the system
    - Assume I/O capacity is large, every job spends 50% of time performing I/O, how many such jobs are needed to keep CPU 100% utilized?

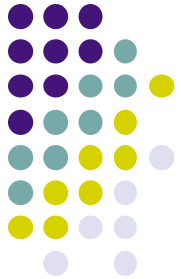
Towards Improving CPU utilization



Ideal curve of # of page faults v.s. # of physical pages



When there are not enough page frames

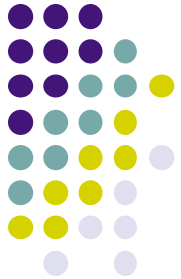


- Suppose many processes are making frequent references to 50 pages, memory has 49
- Assuming LRU
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out
- Btw, what is the optimal strategy here?
  - MRU

- What is the average memory access time?
- The system is spending most of its time paging!

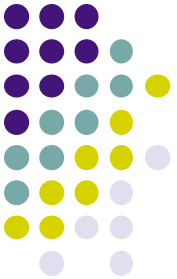
- The progress of programs makes it look like "*memory access is as slow as disk*", rather than "*disk being as fast as memory*"

Thrashing

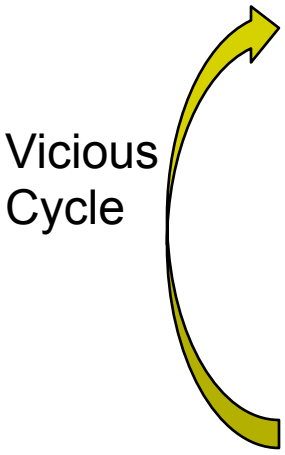


- **Thrashing** ≡ a process is busy swapping pages in and out

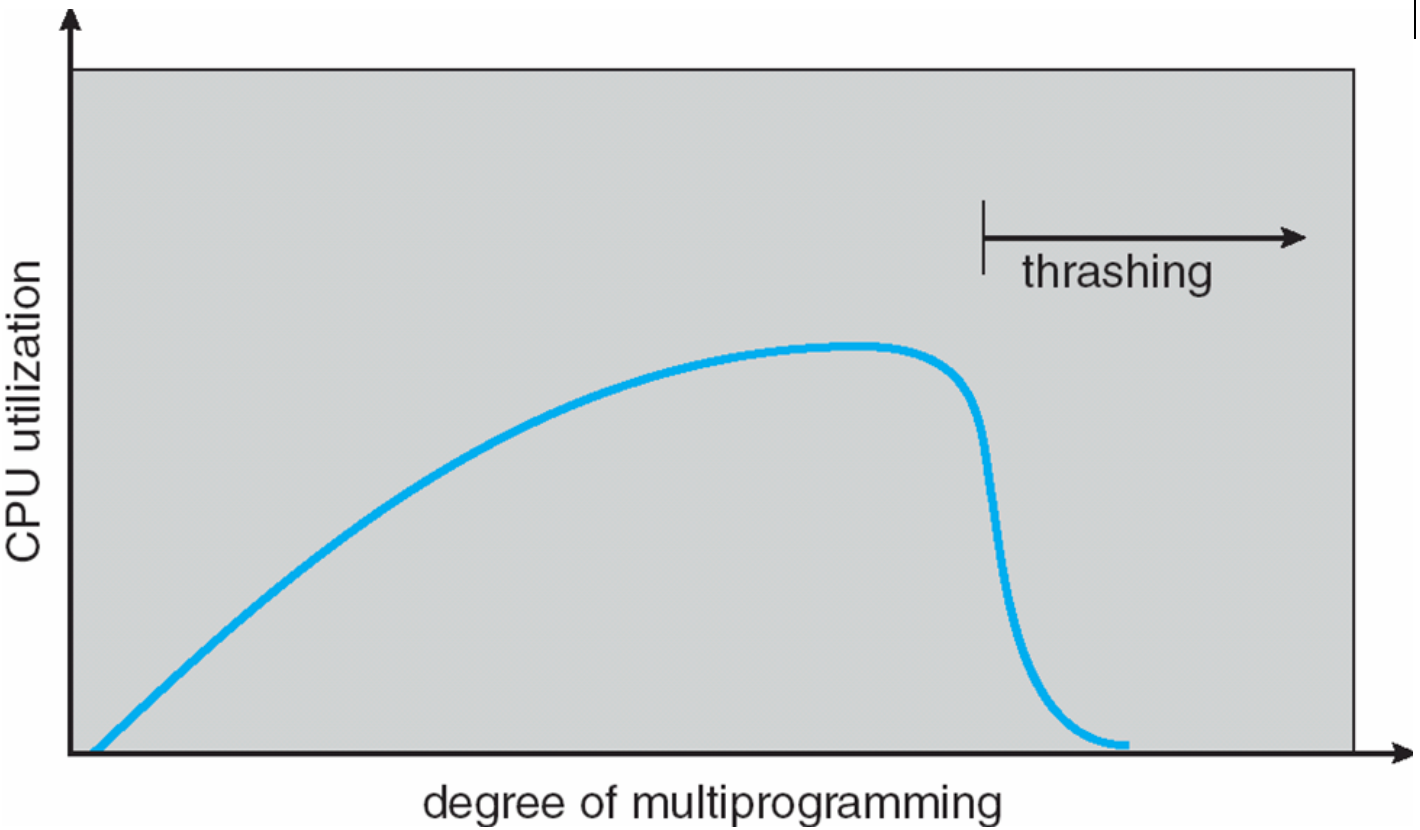
Thrashing can lead to vicious cycle



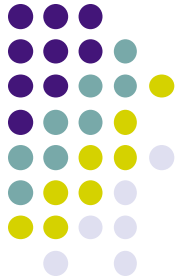
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - another process added to the system
  - page fault rate goes even higher



Thrashing (Cont.)



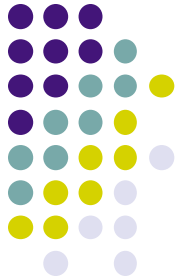
What causes thrashing?



- The system does not know it has taken more work than it can handle

- What do humans do when thrashing?
  - Dropping or degrading a course if taking too many than you can handle ☺

Demand paging and thrashing



- Why does demand paging work?
  - Data reference exhibits locality

- Why does thrashing occur?
  - Σ size of locality > total memory size

Locality in a memory-reference pattern (OSC, 8th ed, Fig 9.19)