Virtual memory
Paging to disk

M1 MOSIG – Operating System Design

Renaud Lachaize

Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (most slides/figures directly adapted from those of the CS140 class)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Textbooks (Silberschatz et al., Tanenbaum)

Outline

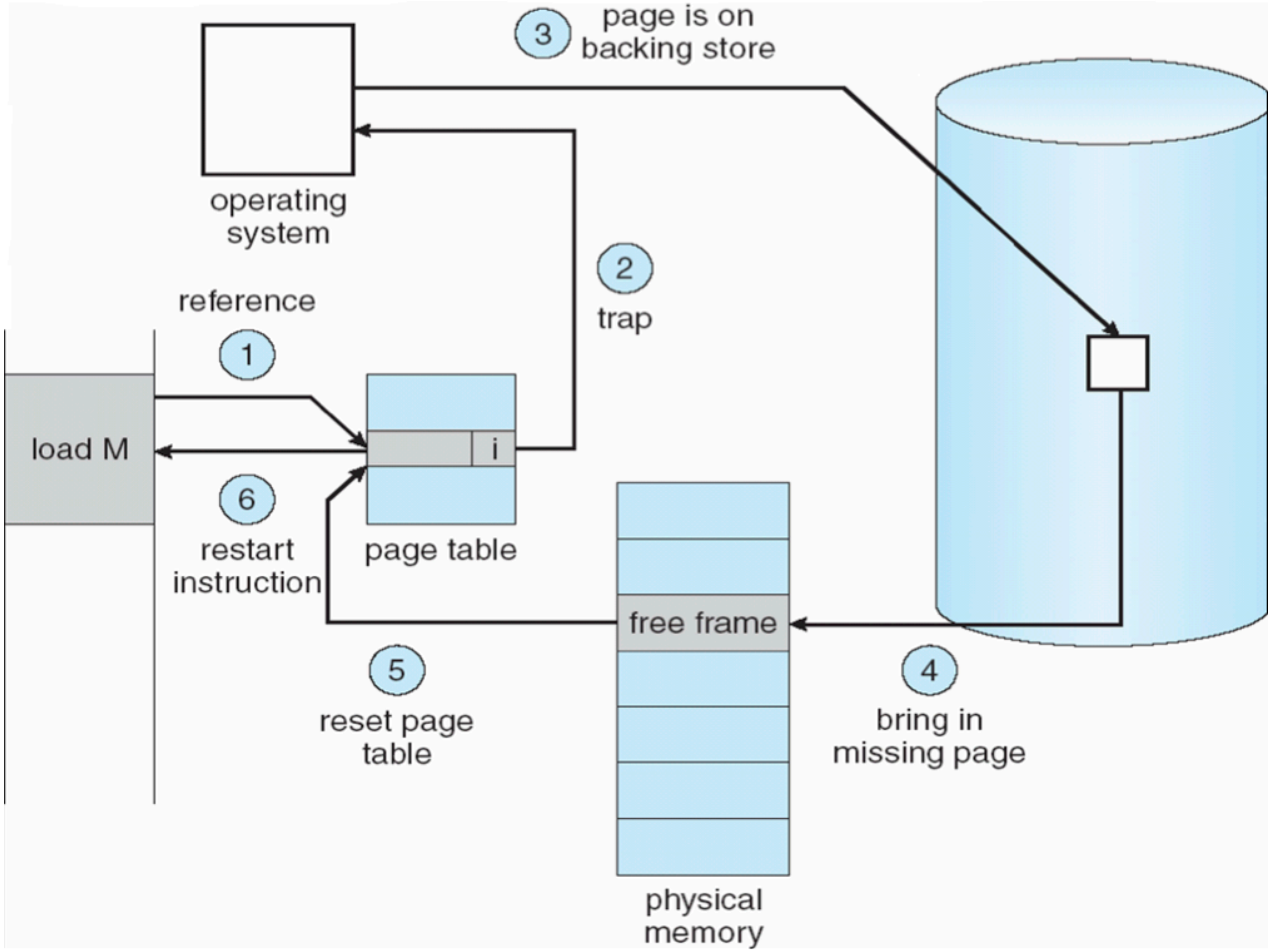
- Principles
- Challenge 1: resuming a process
- Challenge 2: choosing what to fetch
- Challenge 3: choosing what to eject
- Further problems and optimizations

Paging to disk

- Motivation: use secondary storage (disk) to provide a virtual memory with a larger capacity than the physical memory

- The RAM acts like a cache for the disk

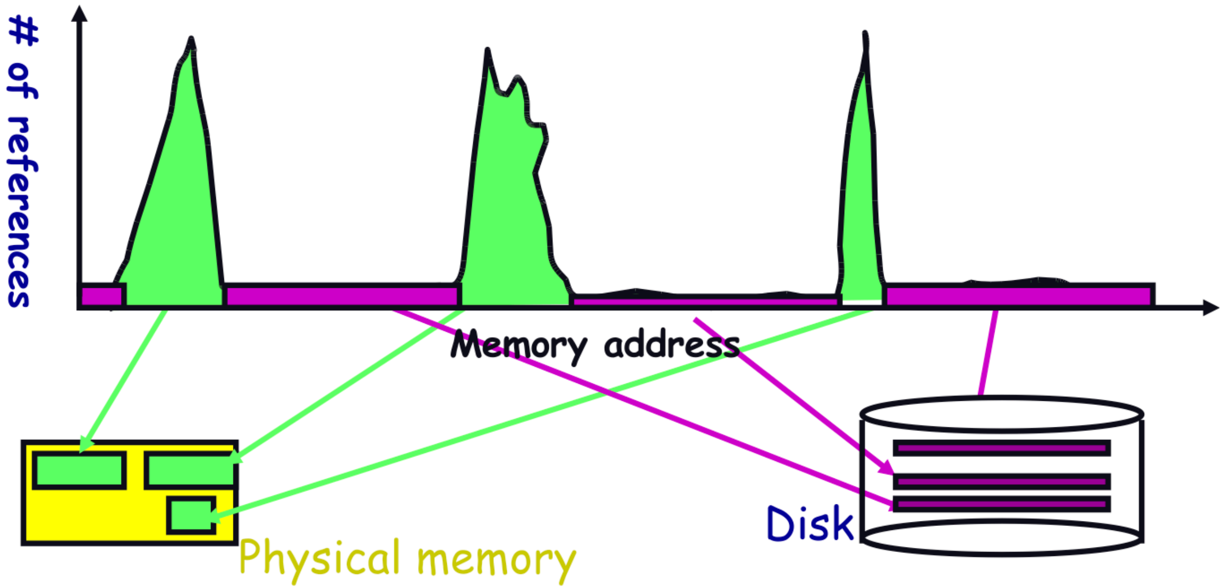
Paging to disk (continued)



Paging to disk
Example

- `gcc` needs a new page of memory
- The kernel reclaims an idle page from `emacs`
- If the reclaimed page is ***clean*** (i.e., also stored on disk, with the same contents)
  - E.g., page of text from `emacs` binary on disk
  - This page can always be re-read from disk
  - OK to discard contents and give page (frame) to `gcc`
- If the reclaimed page is **dirty** (i.e., is the only valid copy)
  - The kernel must write the page to disk first before giving it to `gcc`
- Either way:
  - Mark page invalid in `emacs`'s paging information
  - `emacs` will trigger fault on next access to this virtual page
  - On fault, the kernel reads page data back from disk into a new physical page, maps new page into `emacs`, resumes execution of `emacs`

Working set model



- The disk is much, much slower than memory
  - Goal: run at memory speed, not disk speed
- 90/10 (or 80/20) rule: 10% of memory gets 90% of memory references
  - So, keep that 10% in real memory, the other 90% on disk
  - How to pick which 10%?

Paging challenges

- How to resume a process after a fault?
  - Need to save state and resume
  - Process might be in the middle of an instruction
- What to fetch?
  - Just needed page or more?
- What to evict?
  - How to allocate physical pages among processes?
  - Which pages of a particular process to keep in memory?

Re-starting instructions

- Hardware provides kernel with info about page fault
  - Faulting virtual address
  - Address of instruction that caused fault
  - Was the access a read or write? Was it an instruction fetch?
  - Was it caused by user access to kernel-only memory? (protection fault)

- Hardware must allow resuming after a fault

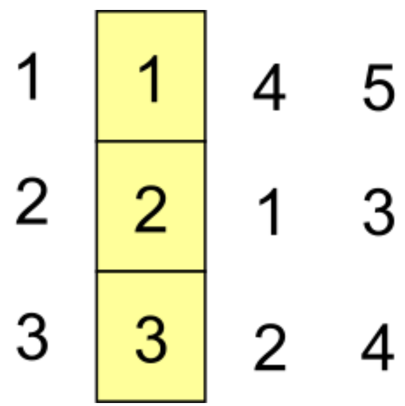
- Idempotent instructions are easy
  - E.g., simple load or store instruction can be restarted
  - Just re-execute any instruction that only accesses one address

What to fetch?

- Bring in page that caused page fault
- Pre-fetch surrounding pages?
  - In many cases, reading two disk blocks is approximately as fast as reading one
  - If application exhibits spatial locality, then big win to store and read multiple contiguous pages

What to evict? Selecting pages
Straw man: FIFO eviction

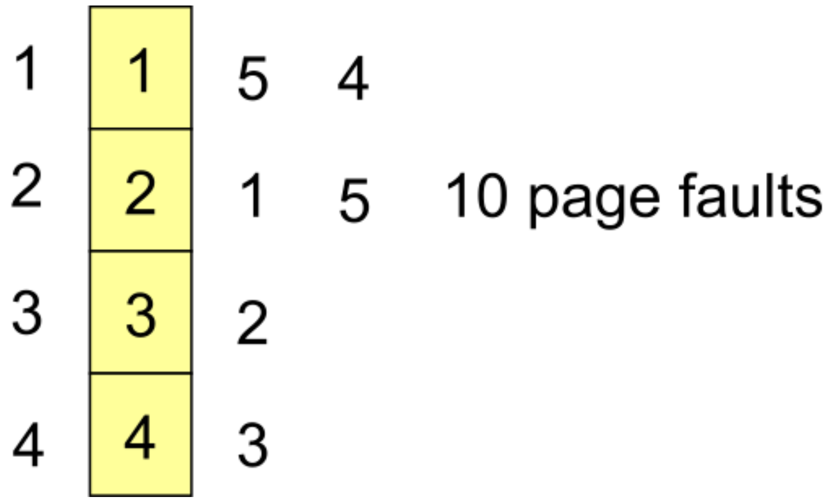
- Evict oldest page fetched in system
- Example - consider the following reference string:
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With a capacity of 3 physical pages: 9 page faults



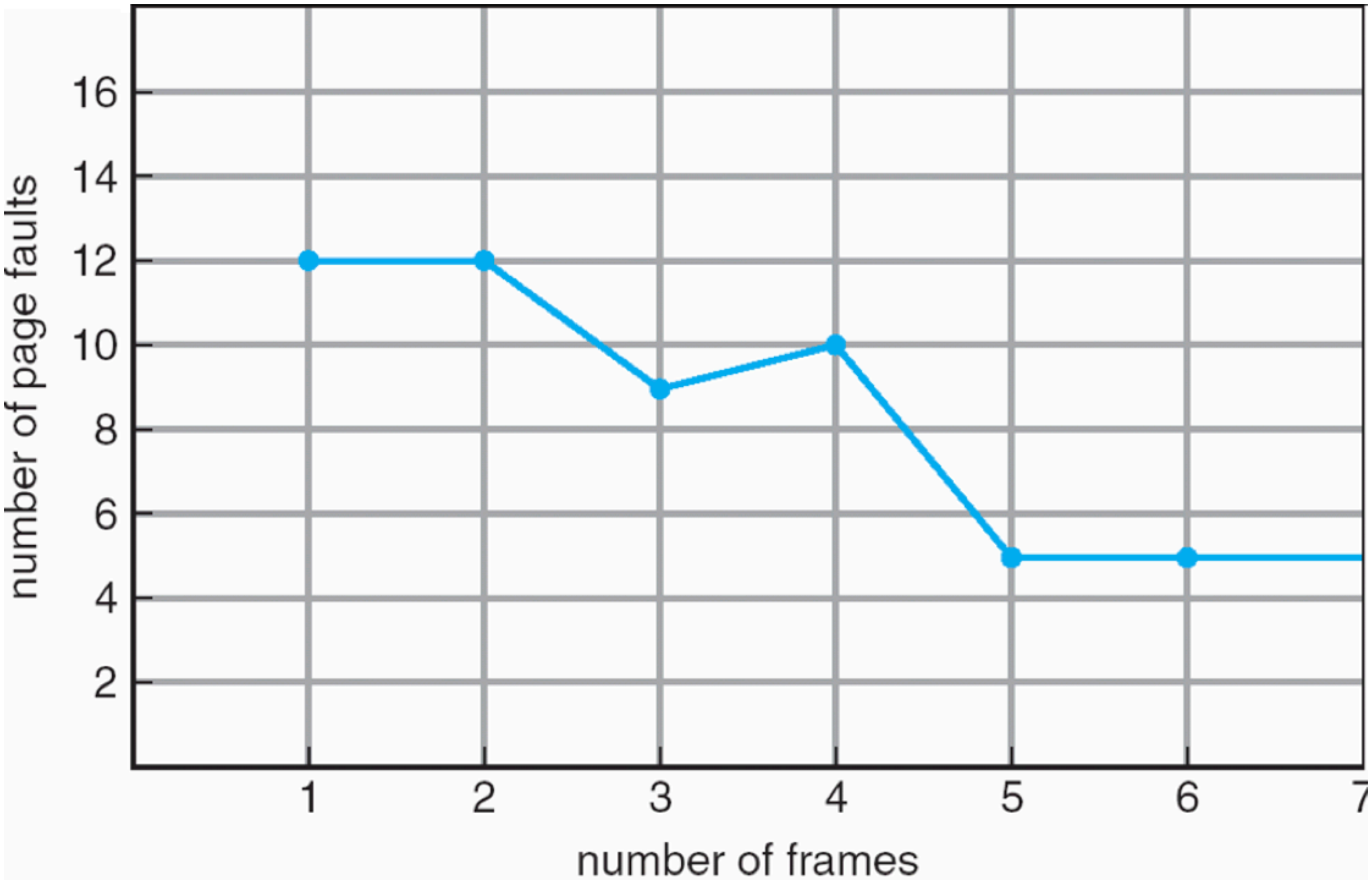
9 page faults

What to evict? Selecting pages
Straw man: FIFO eviction

- Evict oldest page fetched in system
- Example - consider the following reference string:
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With a capacity of 3 physical pages: 9 page faults
- With a capacity of 4 physical pages: 10 page faults



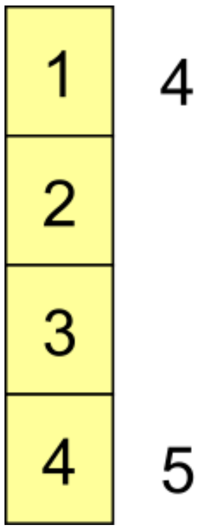
Selecting physical pages
Belady’s anomaly



- More physical memory does not always mean fewer faults

Optimal page replacement

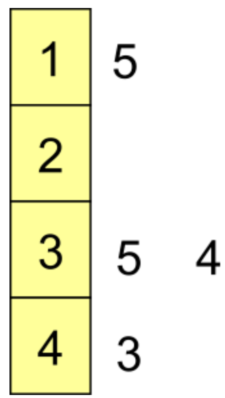
- What is optimal (if you knew the future)?
  - Replace page that will not be used for the longest period of time
- Example – with reference string
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages



6 page faults

LRU page replacement

- Approximate optimal with least recently used
  - Because past often predicts the future
- Example – with reference string
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages: 8 page faults



- Problem 1: can be pessimal – example?
  - Looping over memory (then want MRU eviction)

- Problem 2: How to implement?

Straw man LRU implementations

- Idea 1: Stamp PTEs with timer value
  – E.g., using the CPU cycle counter
  – Automatically write value to PTE on each page access
  – (When page selection is needed) Scan page table to find oldest counter value = LRU page
  – Problem: would dramatically increase the memory traffic

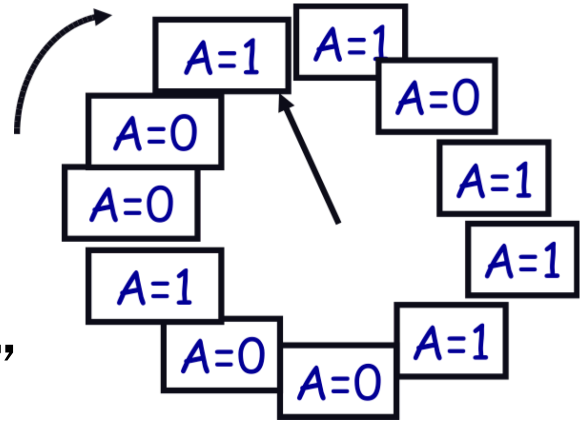
- Idea 2: Keep doubly-linked list of pages
  – On access, remove page, place at tail of list
  – Problem: again, very expensive

- What to do?
  – Just approximate LRU, don’t try to do it exactly

Clock algorithm

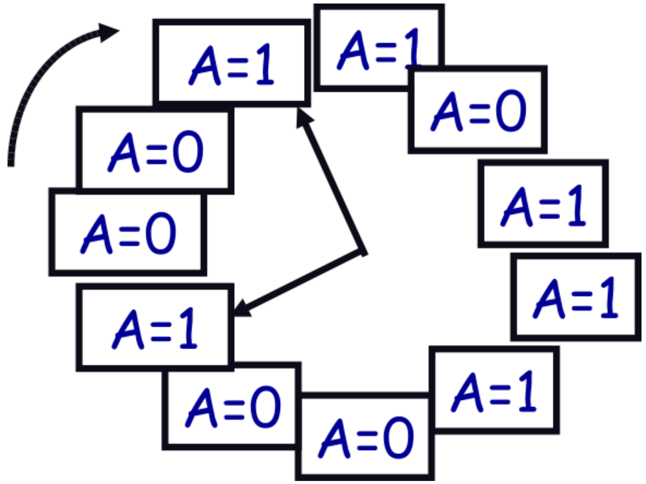
- Use "accessed" bit supported by most hardware
  - E.g., Pentium (x86) will write 1 to "A" bit in PTE on first access
  - Software managed TLBs like MIPS can do the same
- Do FIFO but skip accessed pages

- Keep pages in circular FIFO list
- Scan:
  - If page's "A" bit == 1, set to 0 and skip
  - Else, if "A" == 0, evict
- A.k.a. "second-chance replacement"



Clock algorithm (continued)

- Large memory may be a problem
  - Most pages referenced in long interval
  - So we may end up having all pages with A=1
- Add a second clock hand
  - Two hands move in lockstep
  - Leading hand clears "A" bit
  - Trailing hand evicts pages with "A"==0



- Can also take advantage of hardware "dirty bit"
  - Each page can be *(unaccessed, clean)*, *(unaccessed, dirty)*, *(accessed, clean)* or *(accessed, dirty)*
  - Consider clean pages for eviction before dirty ones
- Or use n-bit variable **`count`** instead of just "A" bit
  - On sweep: **`count`** = (A << (n-1)) | (**`count`** >> 1)
  - Evict page with lowest count

Other replacement algorithms

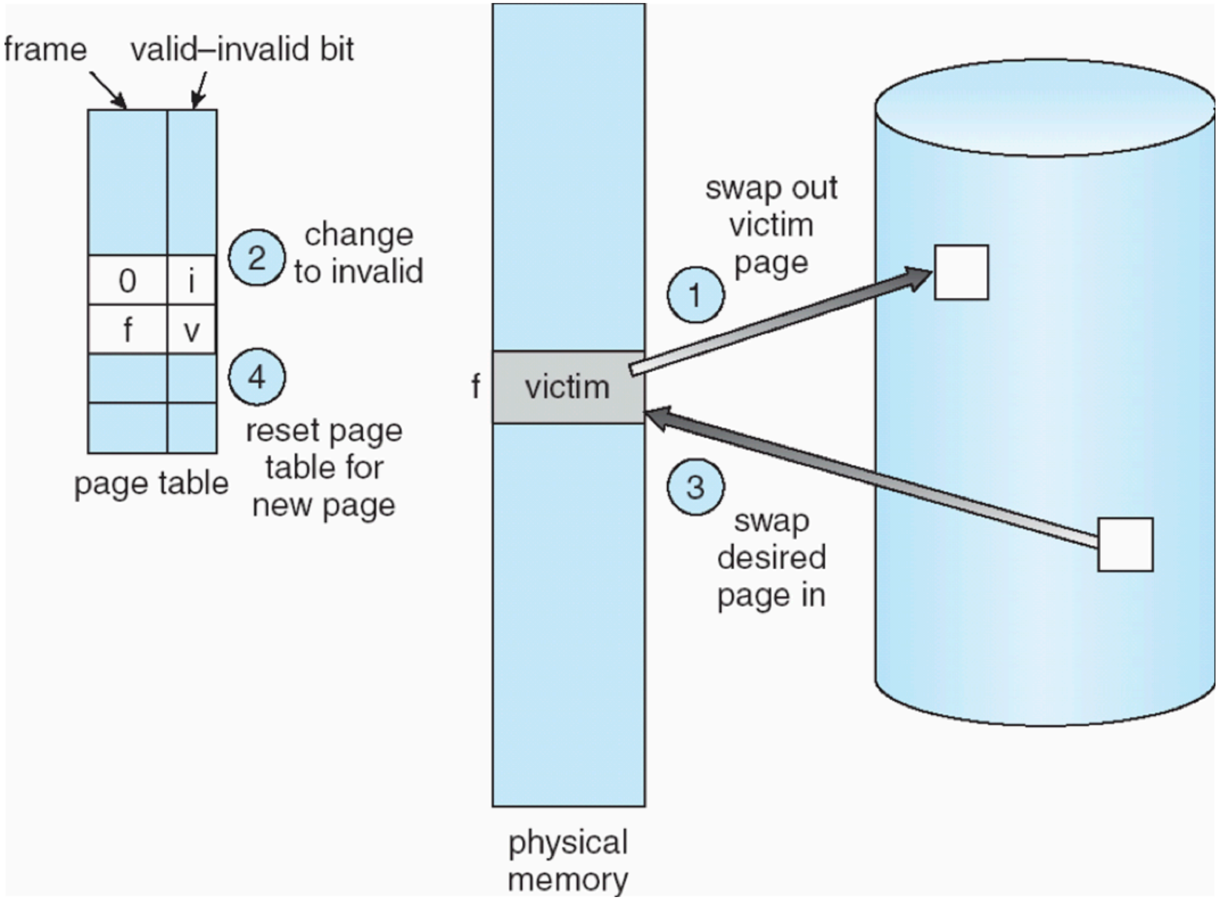
- Random eviction
  - Very simple to implement
  - Not overly horrible results (avoids Belady and pathological cases)

- LFU (least frequently used) eviction
  - Instead of just “A” bit, count the number of times each page is accessed
  - Least frequently accessed page must not be very useful (or maybe was just brought in and is about to be used)
  - Decay usage counts over time (for pages that fall out of usage)

- MFU (most frequently used) algorithm
  - Idea: page with the smallest count was probably just brought in and has yet to be used (so it should not be evicted)

- Neither LFU nor MFU used very commonly

Naïve paging



- Naïve page replacement: 2 disk I/Os per page fault

Page buffering

- Idea: reduce number of I/Os on the critical path

- Keep pool of free page frames
  - On fault, still select victim page to evict
  - But read fetched page into already free page
  - Can resume execution while writing out victim page
  - Then add victim page to free pool

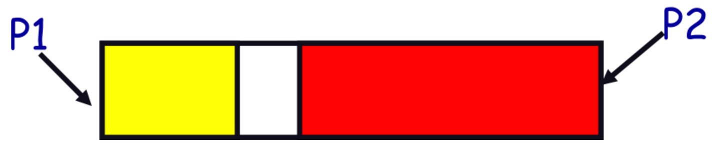
- Can also yank pages back from free pool
  - Contains only clean pages, but may still have data
  - If page fault on page still in free pool, recycle

Outline

- Introduction to paging
  - Principle
  - Data structures and implementation examples
- Speed considerations
  - The memory wall
  - TLB
- Paging to disk
  - Principles
  - Challenge 1: resuming a process
  - Challenge 2: choosing what to fetch
  - Challenge 3: choosing what to eject
  - **Further problems and optimizations**

Page allocation

- **Allocation can be *global* or *local***
- **Global allocation does not consider page ownership**
  - E.g., with LRU, evict least recently used page of any process
  - Works well if P1 needs 10% of memory and P2 needs 70%



  - Does not protect you from “memory pigs” (imagine P2 keeps looping through array that is size of mem)
- **Local allocation isolates processes (or users)**
  - Separately determine how much memory each process should have
  - Then use LRU/clock/etc. to determine which pages to evict within each process

Thrashing

- Thrashing: processes on system require more memory than it has
  - Each time one page is brought in, another page, whose contents will be soon referenced, is thrown out
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - I/O devices at 100% utilization but system not getting much useful work done

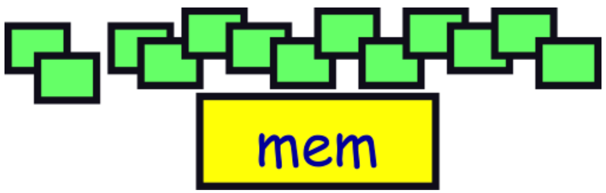
- What we wanted: virtual memory as large as the disk with access time as low as the one of the physical memory
- What we have: memory with access time of the disk ☹

Reasons for thrashing

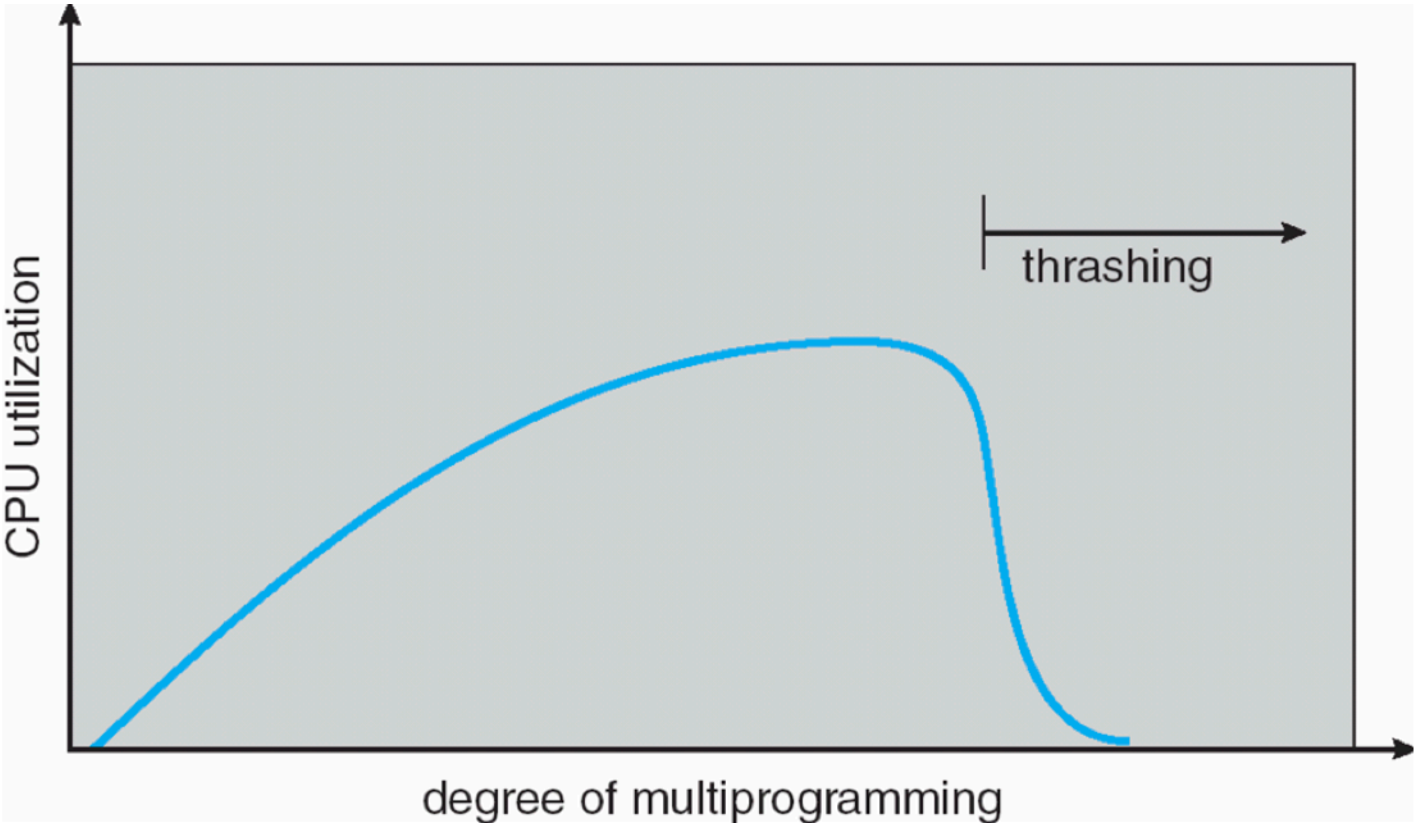
- Process does not reuse memory, so caching does not work (past != future)

- Process does reuse memory, but it does not "fit"

- Individually, all processes fit and reuse memory, but too many for system
  - At least, this case is possible to address (see next slides)



Multiprogramming and thrashing

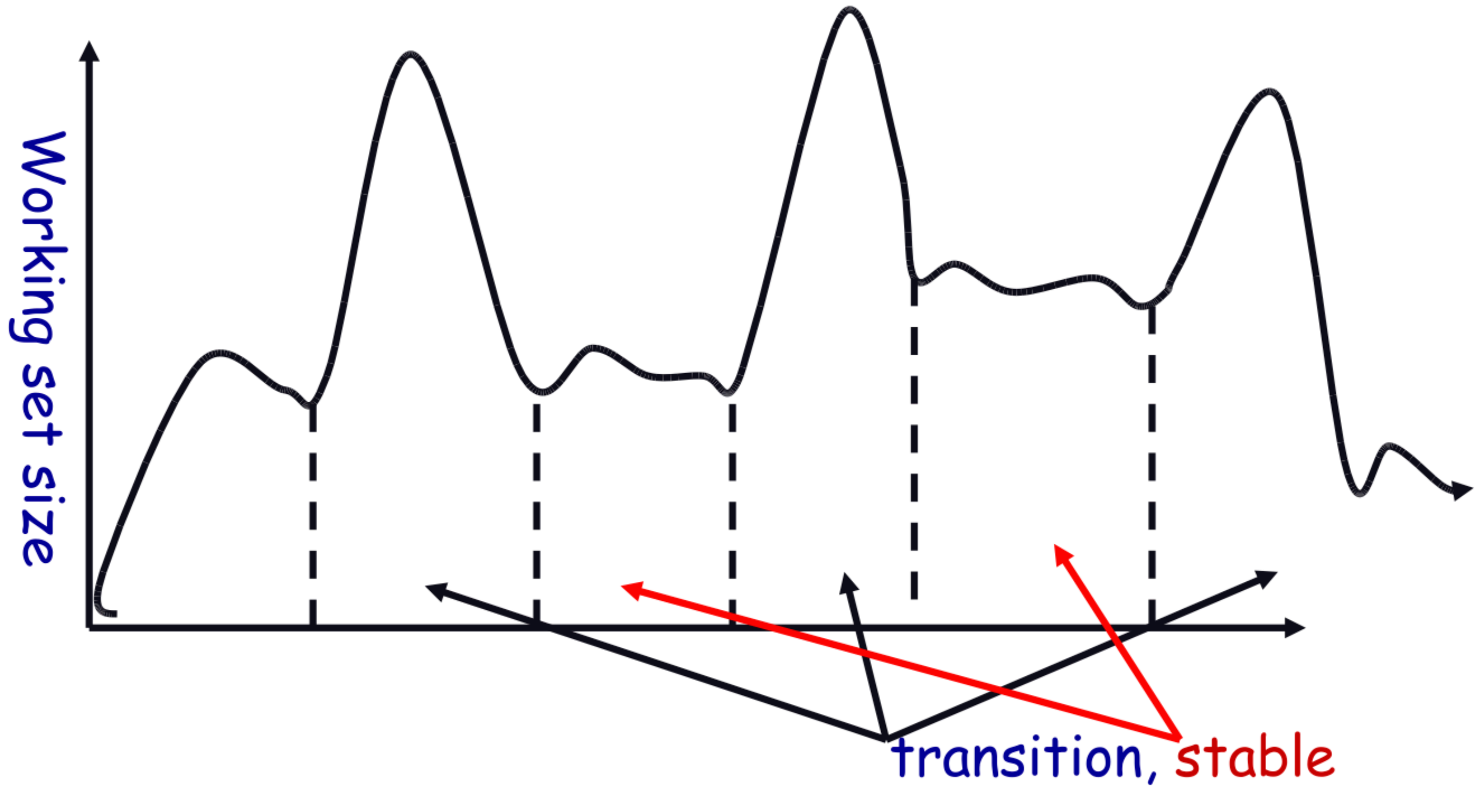


- Need to shed load when thrashing

Dealing with thrashing

- Approach 1: working set
  - Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
  - Or: how much memory does process need in order to make reasonable progress (its working set)?
  - Only run processes whose memory requirements can be satisfied
- Approach 2: page fault frequency (PFF)
  - Thrashing viewed as poor ratio of “page fetch” to “useful work”
  - PFF = page faults / instructions executed
  - If PFF rises above threshold, process needs more memory. If not enough memory on the system, swap out.
  - If PFF sinks below threshold, memory can be taken away

Working sets



- Working sets changes across phases
  - Baloons during transition

Calculating the working set

- Working set: all the pages that a process will access in next T time frame
  – Cannot calculate without predicting the future
- Approximate by assuming past predicts future
  – So working set ~ pages accessed in last T interval
- Keep idle time for each page
- Periodically scan all resident pages in the system
  – "A" bit set? Clear it and clear the page's idle time
  – "A" bit clear? Add CPU consumed since last scan to idle time
  – Working set is pages with idle time < T

Two-level scheduler

- Divide processes into active and inactive
  - Active – means working set resident in memory
  - Inactive – working set intentionally not loaded
- Balance set: union of all active working sets
  - Must keep balance set smaller than physical memory
- Use long-term scheduler
  - Moves processes from active to inactive state until balance set is small enough
  - Periodically allows inactive to become active
  - As working set changes, must update balance set
- Complications
  - How to chose idle time threshold T?
  - How to pick processes for active set?
  - How to count shared memory (e.g., libc.so)?

Recap

- Paging brings nice benefits
  – Removes the fragmentation issue (in the context of address space management)
  – Enables to offload the RAM (demand paging) and thus to fit more processes in RAM
  – Enables to run processes requiring more memory than the available RAM

- Page replacement issues
  – When the RAM is full, a page must be evicted, stored back on the disk and replaced in RAM by the requested one
  – This content management has similarities with the ones in caches, TLB, ... but is implemented in software
  – Good policies build on locality, regularity of memory accesses
  – Workload and speed/size of the different memory/disk components call for different policies, data structures and tradeoffs

References

- Bruce Jacob and Trevor Mudge. Virtual memory: issues of implementation. IEEE Computer, June 1998.

- AMD and Intel documentations (see previously mentioned links)

- Replacement policies and working sets: see textbooks