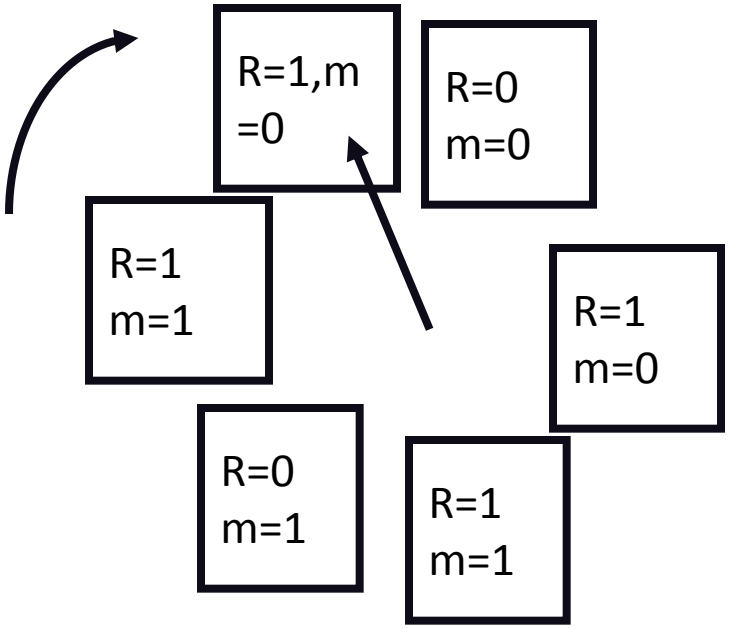
The problem with dirty pages

- Simplistic view:
  - Pages are scarce, allocate them all
  - When need a page, take one from some process and use
  - Problem: What's cost of evicting a dirty page?

- Simple hack:
  - use "dirty" bit to give preference to dirty pages
  - Example: MacOS

MacOS: modification sensitive clock alg

- 4 states
  - not ref'd, not modified (r = 0, m = 0)
  - ref'd, not modified (r = 1, m = 0)
  - not ref'd, modified (r = 0, m = 1)
  - ref'd, modified (r = 1, m = 1)
- 2 passes
  - look for r=0,m=0
  - look for r=0,m=1
  - & clear r
  - repeat
- When is this not a great idea?



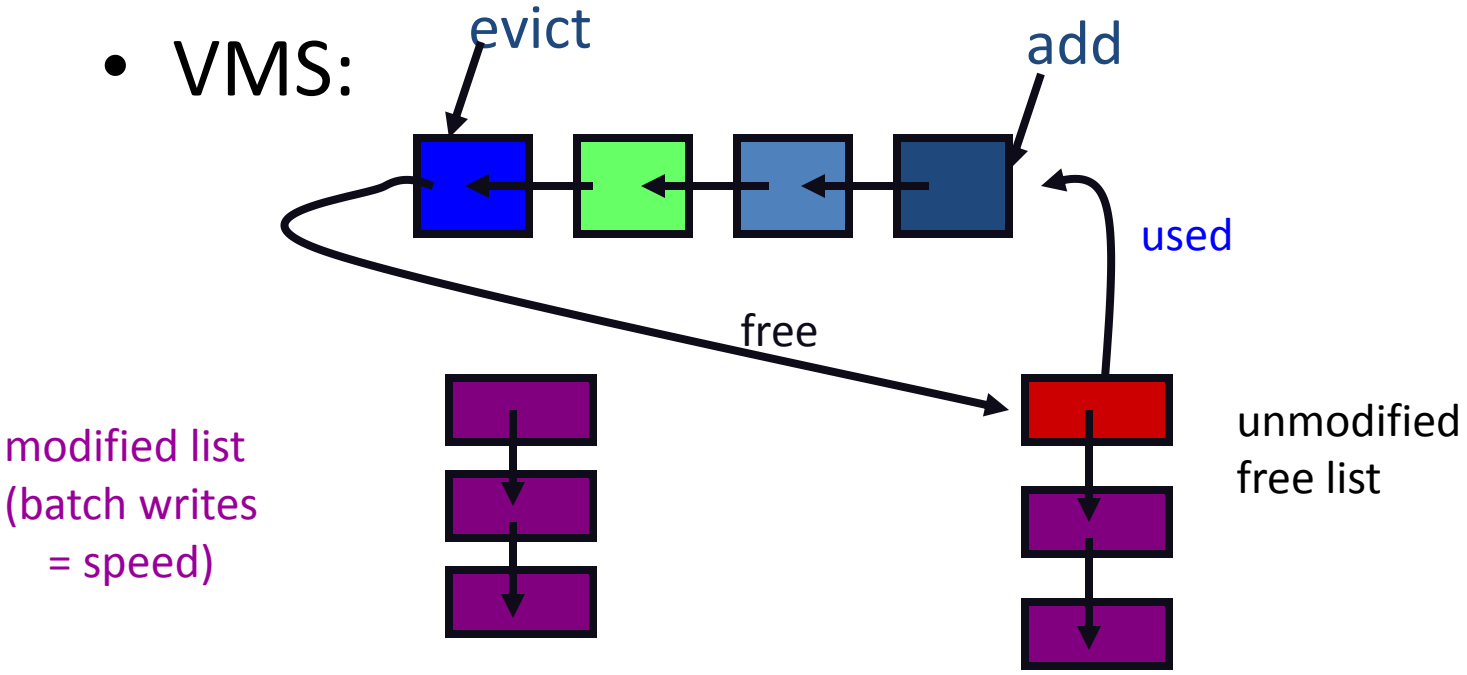
More common approaches

- Sol'n 1: waste space
  - Only allocate (say) 90% of memory
  - When evict dirty page, start write and allocate from spare.
  - Remember identity of free pages, if need again, reclaim
  - Example: VMS

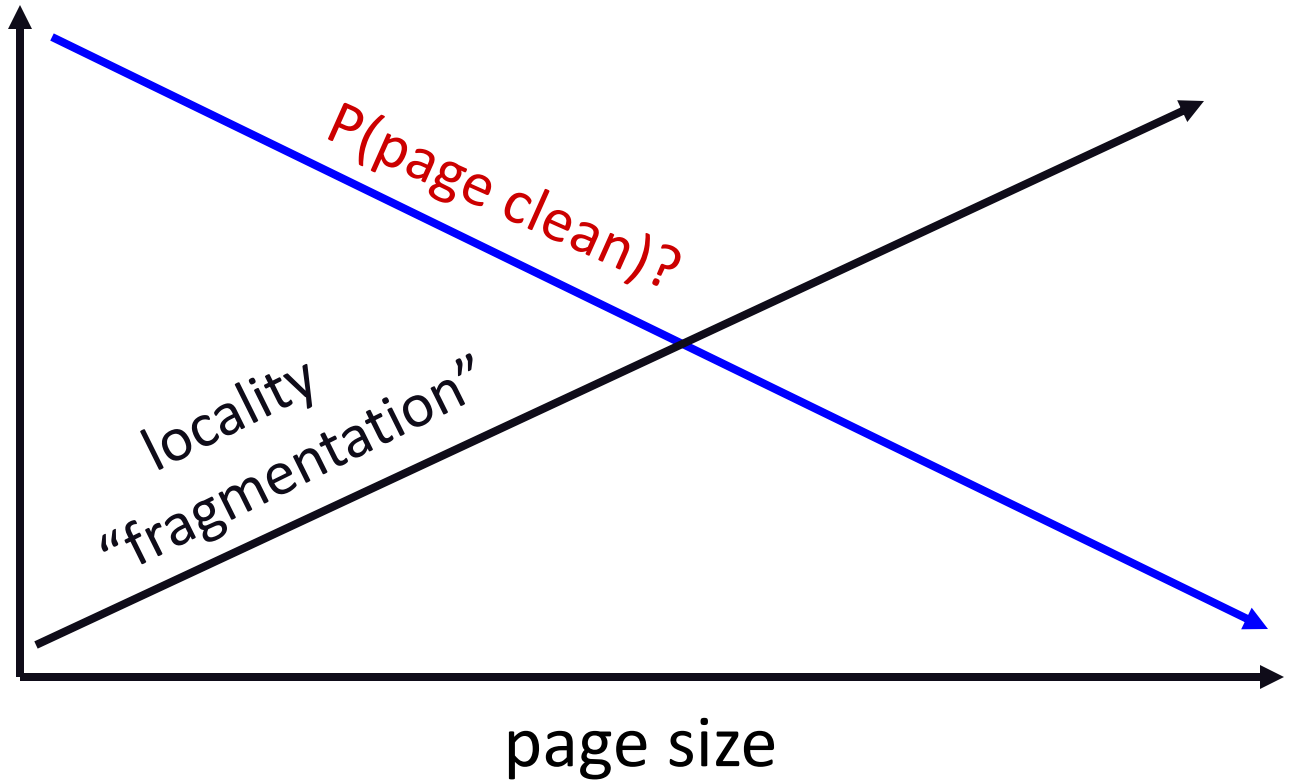
- Sol'n 2: pre-do writes:
  - have daemon process that periodically writes dirty pages back to disk. (And, of course, do above reclaim trick)
  - Most systems do this. Example: Unix.

A different take: page buffering

- Simple trick (VMS, Mach):
  - keep spare of free pages; recycle in FIFO order
  - but record what free page corresponds to: if used before overwritten put back in place.
- VMS:



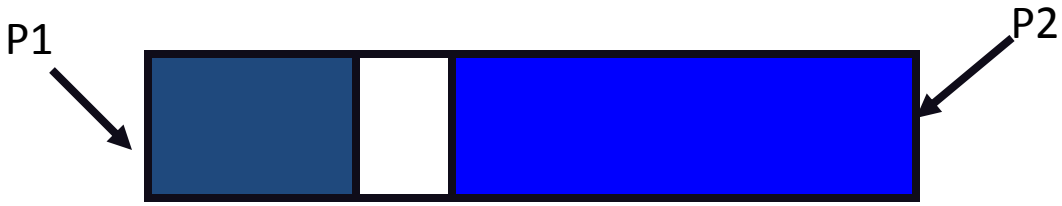
Page size = big paging impact



- Larger page = more unrelated "things" on page
- Larger page = more chance that you've written to it.

Global or local?

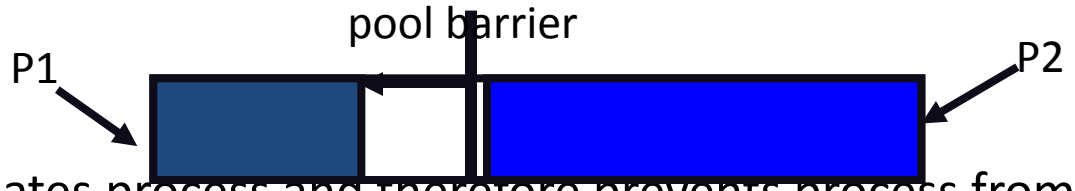
- So far, we've implicitly assumed memory comes from a single global pool ("Global replacement")
  - when process P faults and needs a page, take oldest page on entire system
  - Good: shared caches are adaptable. Example if P1 needs 20% of memory and P2 70%, then they will be happy.



  - Bad: too adaptable. No protection from pigs
    - What happens to P1 if P2 sequentially reads array about the size of memory?

Per-process and per-user page replacement

- Per-process (per-user same)
  - each process has a separate pool of pages
  - a page fault in one process can only replace one of this process's frames
  - isolates process and therefore relieves interference from other processes



  - but, isolates process and therefore prevents process from using other's (comparatively) idle resources
  - efficient memory usage requires a mechanism for (slowly) changing the allocations to each pool
  - Qs: What is "slowly"? How big a pool? When to migrate?
  - Should we completely swap some processes out of memory

Allocation of Page Frames (Memory Pages)

- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle *from*
    - 2 pages to handle *to*

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes⇒process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

    s_i = size of process p_i and $S = \Sigma s_i$

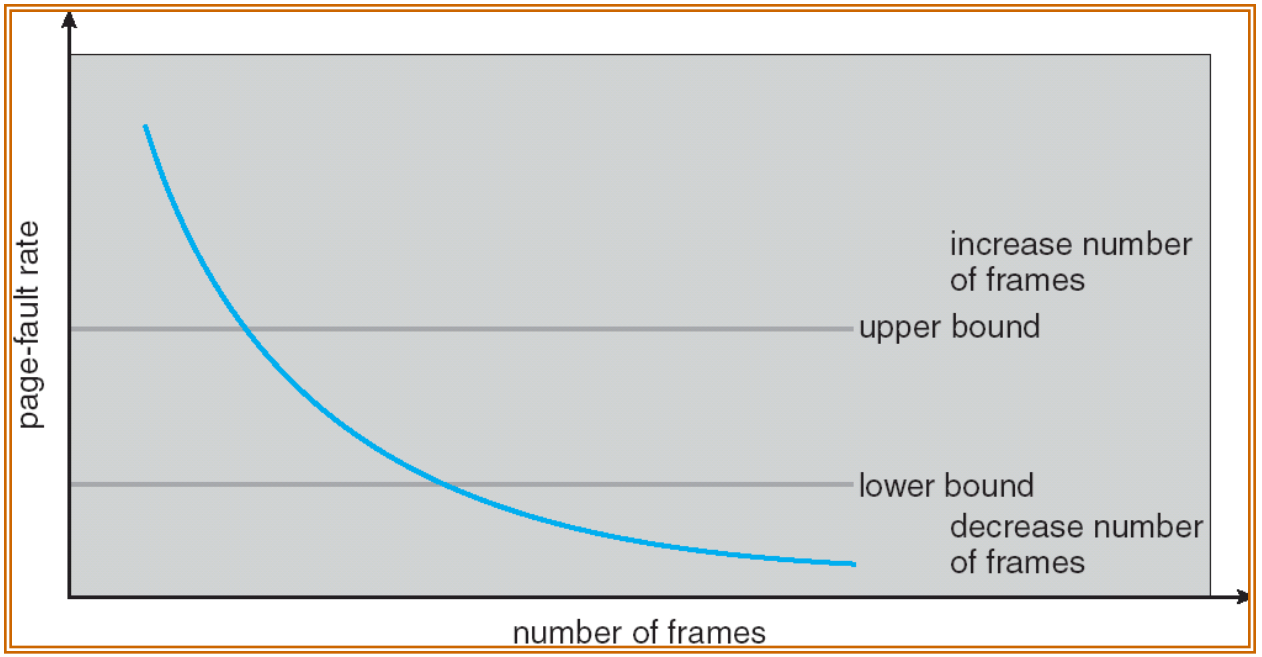
    m = total number of frames

    a_i = allocation for p_i = $\frac{s_i}{S} \times m$

- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory? (next lecture)

Why does VM caching look so different?

- Recall: formula for speedup from cache:

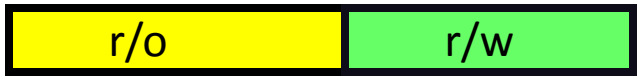
  p = % of accesses that hit in cache
  access time = p * (cache access time)
  + (1-p) * (real access time + cache miss overhead)

- TLB and memory cache need access times ~ that of instruction
  - Not a whole lot of time to play around.

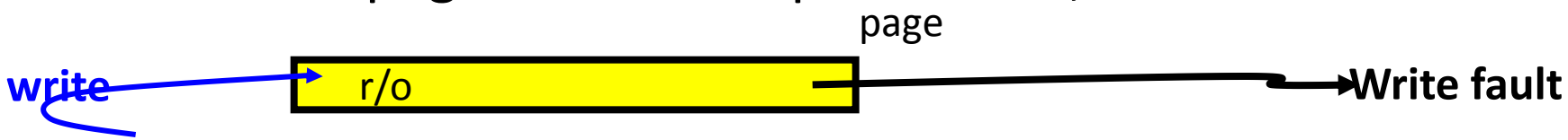
- VM caching measured closer to that of a disk access.
  - Miss cost so expensive, easily hide high associativity cost, and overhead of sophisticated replacement algorithms

Flashback: Faults + resumption = power

- The ability to resume after a fault lets the OS emulate a huge number of things.
  - ("every problem can be solved with layer of indirection")
- Example: sub-page protection
  - nice thing about segmentation: lets us protect arbitrary sized byte ranges.
- To protect sub-page region in paging system:



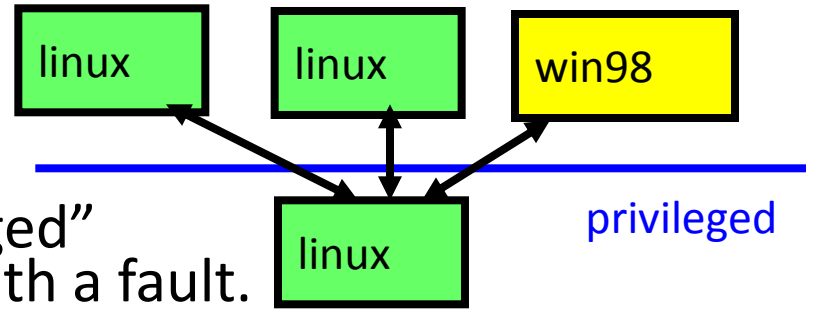
  - Set entire page to weakest permission; record in PT



  - Any access that violates perm will cause an access fault
  - Fault handler checks if page special, and if so, if access allowed. Continue or raise error, as appropriate

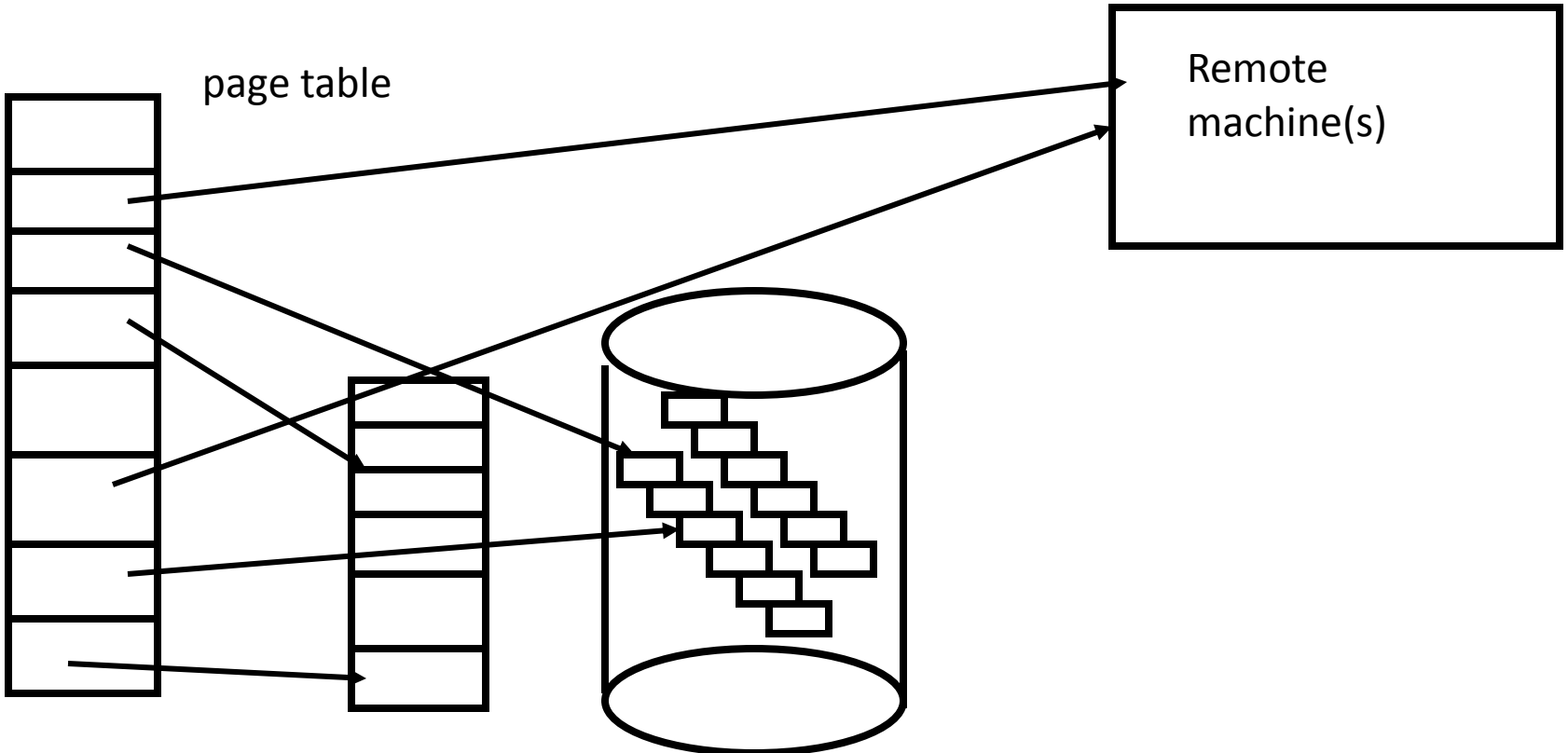
Fault resump. lets us lie about many things

- Emulate reference bits:
  - Set page permissions to “invalid”.
  - On any access will get a fault: Mark as referenced
- Emulate non-existent instructions:
  - Give inst an illegal opcode. When executed will cause “illegal instruction” fault. Handler checks opcode: if for fake inst, do, otherwise kill.
- Run OS on top of another OS!
  - Slam OS into normal process
  - When it does something “privileged” the real OS will get woken up with a fault.
  - If op allowed, do it, otherwise kill.
  - IBM’s VM/370. More recent: vmware.com



Distributed shared memory

- Virtual memory allows us to go to memory or disk
  - But, can use the same idea to go anywhere! Even to another computer. Page across network rather than to disk. Faster, and allows network of workstations (NOW)

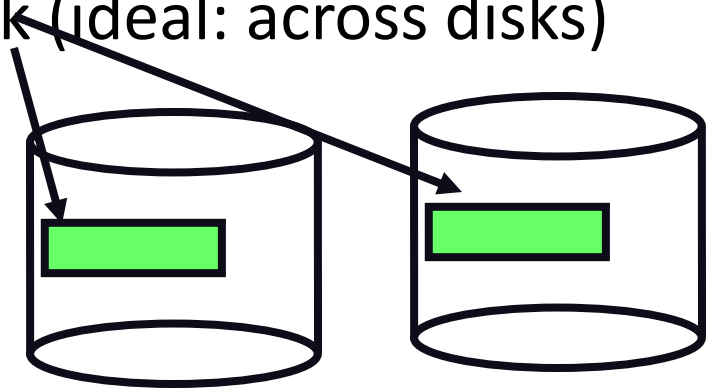


DSM “details” (still about 10,000 ft up)

- Simplest approach: each page has one owner
  - (trivially coordinates multiple updates to same page)
- Page table tracks resident/non-resident
  - for page on disk: the block holding it
  - for page on network: the owning machine
  - Difference? The latter may be wrong.
  - Truth in a distributed setting tends to be expensive. (coordinating all machines does not scale.)
  - So, location is frequently just a hint. If not at the location, use more expensive algorithm to find it.
- Page fault:
  - on disk? Fetch. On network, find owner, claim page.
- Tradeoffs in page size: large page -> “false sharing”

Swapping mechanics

- Allocate contiguous region on disk (ideal: across disks)



- When to allocate space?
  - When page allocated? (BSD 4.3)
  - When you need to page?
- What about code pages?
  - Get from file? (BSD 4.3)
  - Page to swap? (Why?)
- When to swap process?
  - 4.3 BSD ‘swapper’ (pid 0): 20 seconds idle (gone home), when thrashing, take longest resident of 4 largest ones.
- Note: Early systems used swapping for protection!