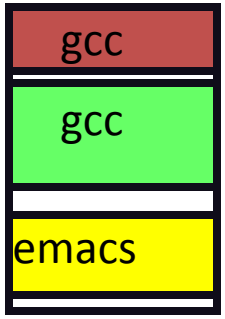
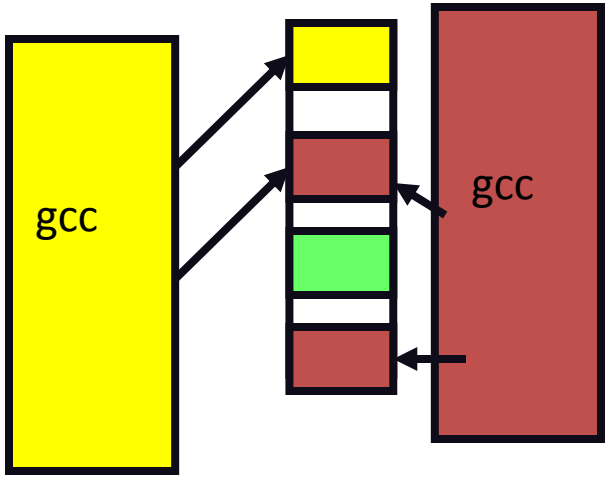
Past: Making physical memory pretty

- Physical memory:
  - no protection
  - limited size
  - almost forces contiguous allocation
  - sharing visible to program
  - easy to share data



- Virtual memory
  - each program isolated from others
  - transparent:can't tell where running
  - can share code, data
  - non-contiguous allocation
  - Today: some nuances + illusion of infinite memory

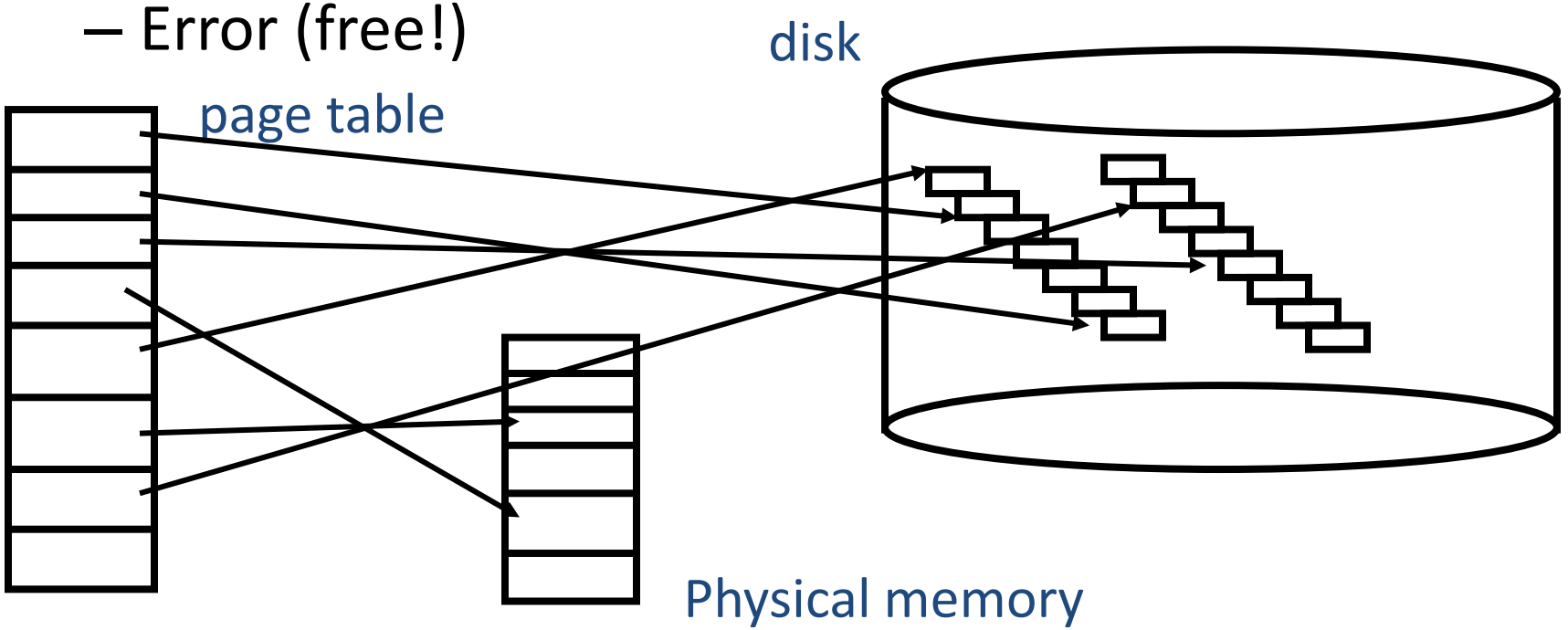


Paging

- Readings for this topic: Chapter 10
- Our simple world:
  - load entire process into memory. Run it. Exit.
- Problems?
  - slow (especially with big process)
  - wasteful of space (process doesn't use all of its memory)
- Solution: partial residency
  - demand paging: only bring in pages actually used
  - paging: only keep frequently used pages in memory
- Mechanism:
  - use virtual memory to map some addresses to physical pages, some to disk

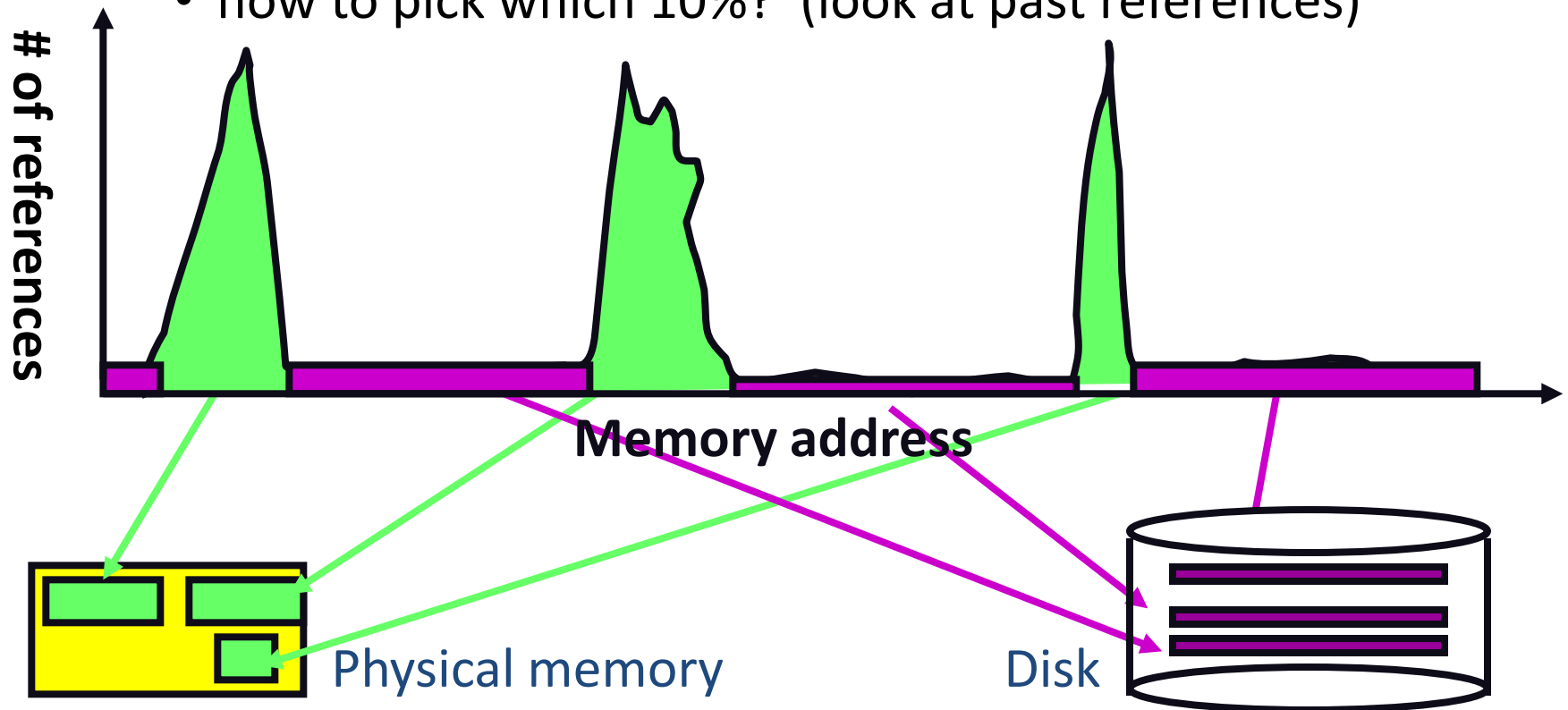
Demand paging from 50,000 feet

- Virtual address translated to:
  - Physical memory ($0.1/meg). Very fast, but small
  - Disk ($.001/meg). Very large, but verrrrrry slow (millis vs nanos)
  - Error (free!)



Demand paging = fool the process

- Want: disk-sized memory that's fast as physical mem
  - 90/10 rule: 10% of memory gets 90% of memory refs
  - so, keep that 10% in real memory, the other 90% on disk
    - how to pick which 10%? (look at past references)

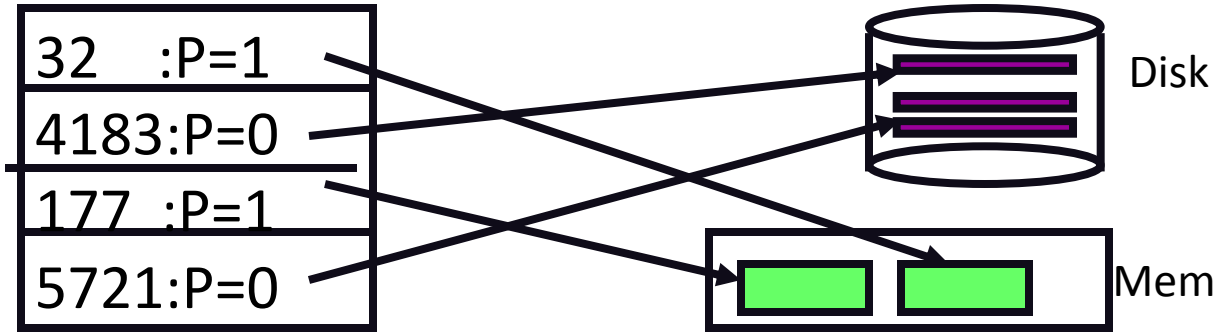


Demand Paging is Caching

- Since Demand Paging is Caching, must ask:
  - What is block size?
    - 1 page
  - What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
    - Fully associative: arbitrary virtual→physical mapping
  - How do we find a page in the cache when look for it?
    - First check TLB, then page-table traversal
  - What is page replacement policy? (i.e. LRU, Random…)
    - This requires more explanation… (kind of LRU)
  - What happens on a miss?
    - Go to lower level to fill miss (i.e. disk)
  - What happens on a write? (write-through, write back)
    - Definitely write-back. Need dirty bit!

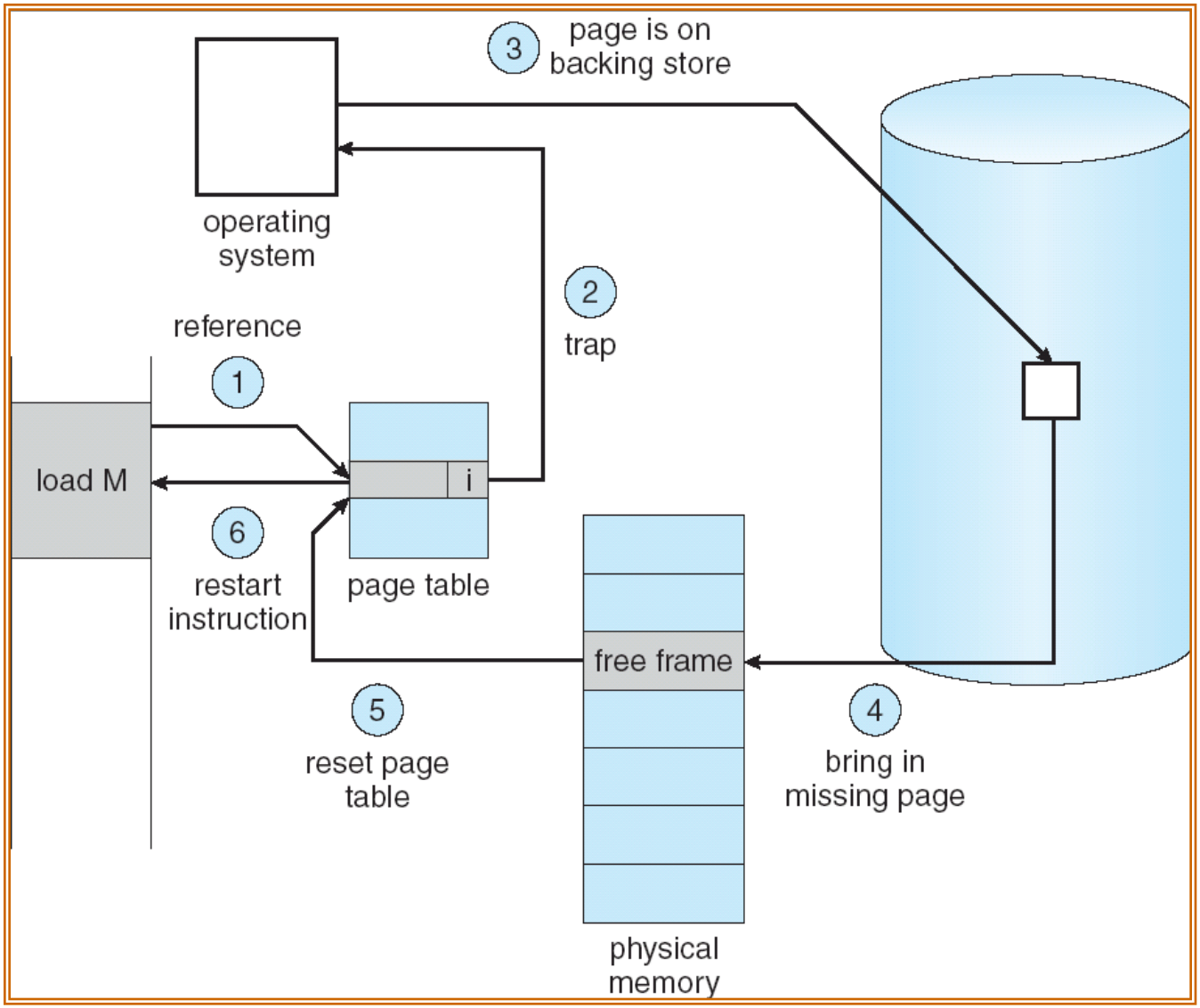
Virtual memory mechanics

- Extend page table entries with extra bit ("present")
  - if page in memory? present = 1, on disk, present = 0
  - translations on entries with present = 1 work as before
  - if present = 0, then translation causes a page fault.



- What happens on page fault?
  - OS finds a free page or evicts one (which one??)
  - issues a disk request to read in data into that page
  - puts process on blocked Q, cswitches to new process
  - when disk completes: set present = 1, put back on run Q

Steps in Handling a Page Fault

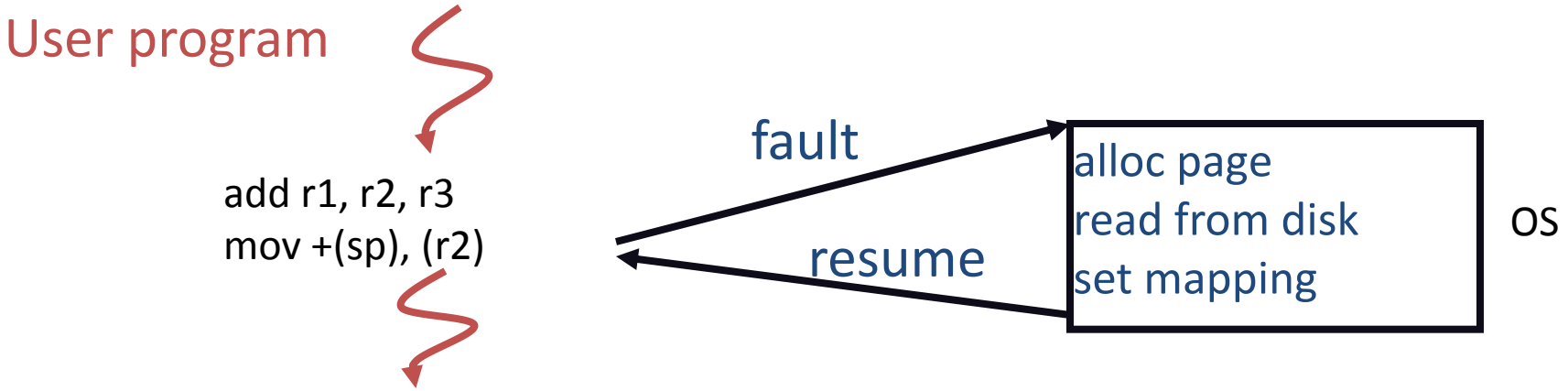


Virtual memory problems

- Problem 1: how to resume a process after a fault?
  - Need to save state and resume.
  - Process might have been in the middle of an instruction!
- Problem 2: what to fetch?
  - Just needed page or more?
- Problem 3: what to eject?
  - Cache always too small, which page to replace?
  - Want to know future use...

Problem 1: resuming process after a fault

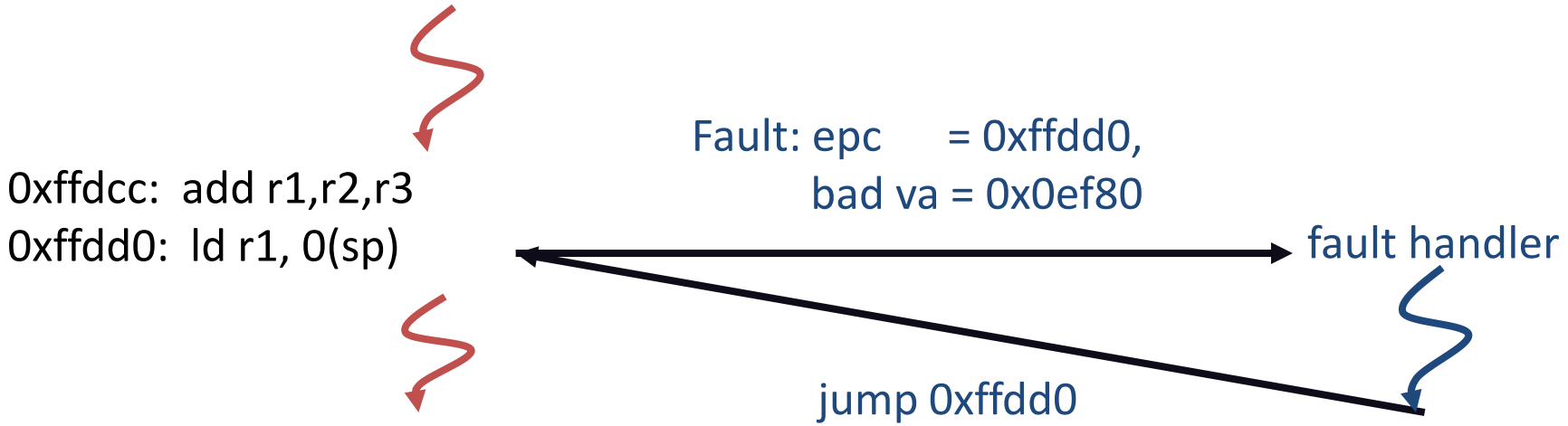
- Fault might have happened in the middle of an inst!



  - Our key constraint: don't want user process to be aware that page fault happened (just like context switching)
  - Can we skip the faulting instruction? Uh, no.
  - Can we restart the instruction from the beginning?
    - Not if it has partial-side effects.
  - Can we inspect instruction to figure out what to do?
    - May be ambiguous where it was.

Solution: a bit of hardware support

- RISC machines are pretty simple:
  - typically instructions idempotent until references done!
  - Thus, only need faulting address and faulting PC.
- Example: MIPS



- CISC harder:
  - multiple memory references and side effects
  - Notion of precise exceptions

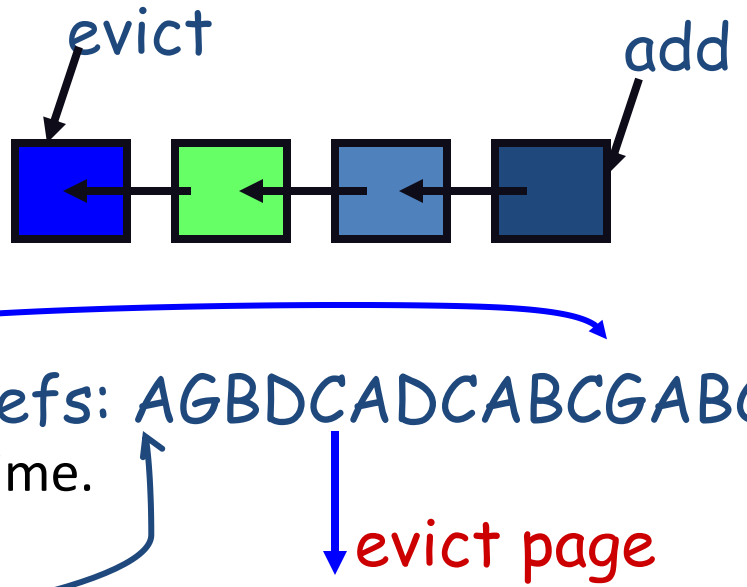
Problem 2: what to fetch?

- Page selection: when to bring pages into memory
  - Like all caches: we need to know the future.
- Doesn't the user know? (Request paging)
  - Not reliably.
  - Though, some Oses do have support for prefetching.
- Easy load-time hack: demand paging
  - Load initial page(s). Run. Load others on fault.

  - When will startup be slower? Memory less utilized?
  - Most systems do some sort of variant of this
- Tweak: pre-paging. Get page & its neighbors (why?)

Problem 3: what to eject & when?

- Random: pick any page.
  - Pro: good for avoiding worst case
  - con: good for avoiding best case
- FIFO: throw out oldest page
  - fair: all pages get = residency
  - dopey: ignores usage.
- MIN (optimal):
  - throw out page not used for longest time.
  - Impractical, but good yardstick
- Least recently used.
  - throw out page that hasn't been used in the longest time.
  - Past = future? LRU = MIN.

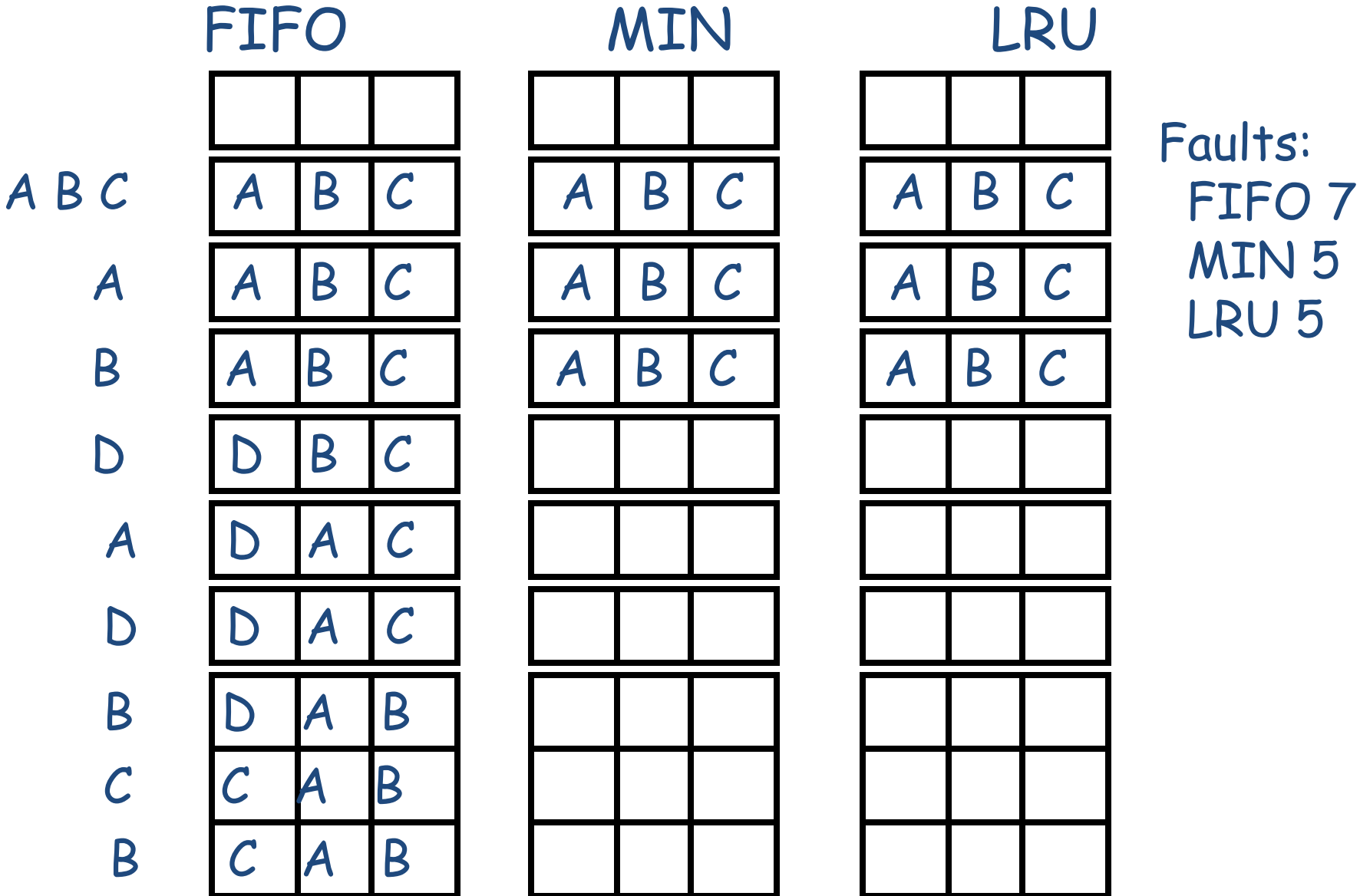


Associativity vs. Miss rate

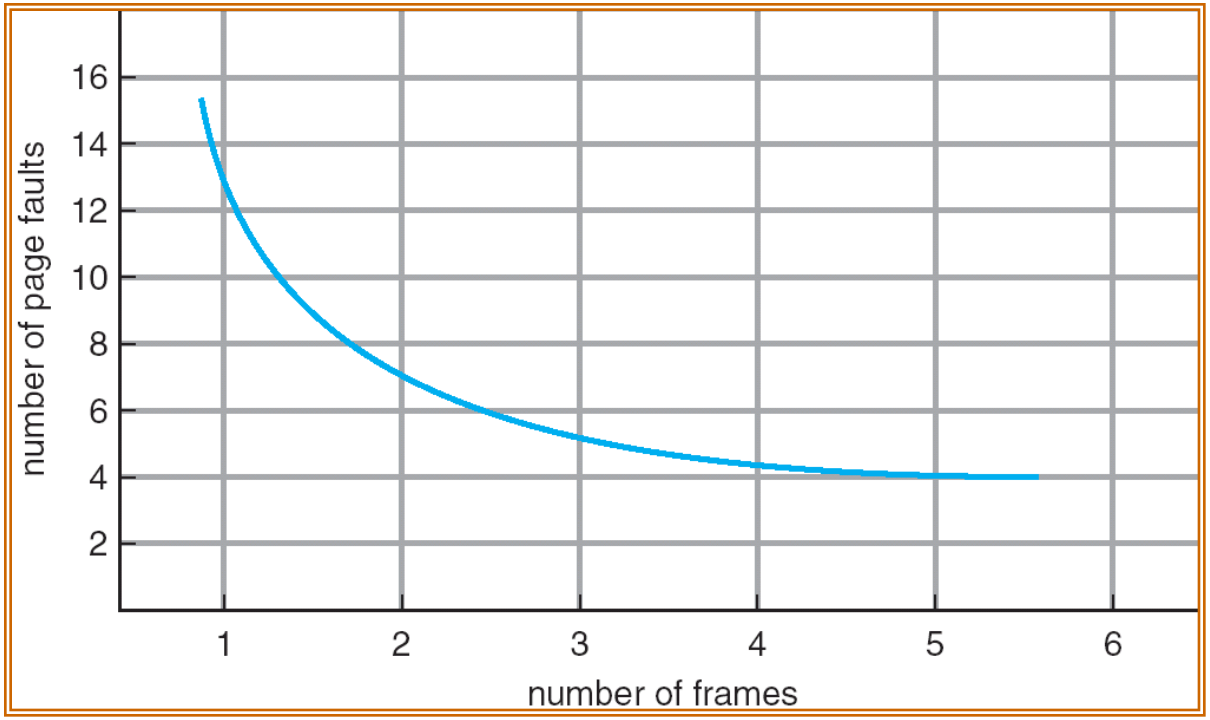
- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

	2-way		4-way		8-way	
Size	LRU	Rand	LRU	Rand	LRU	Rand
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Reference string: A B C A B D A D B C B



Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: Belady's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

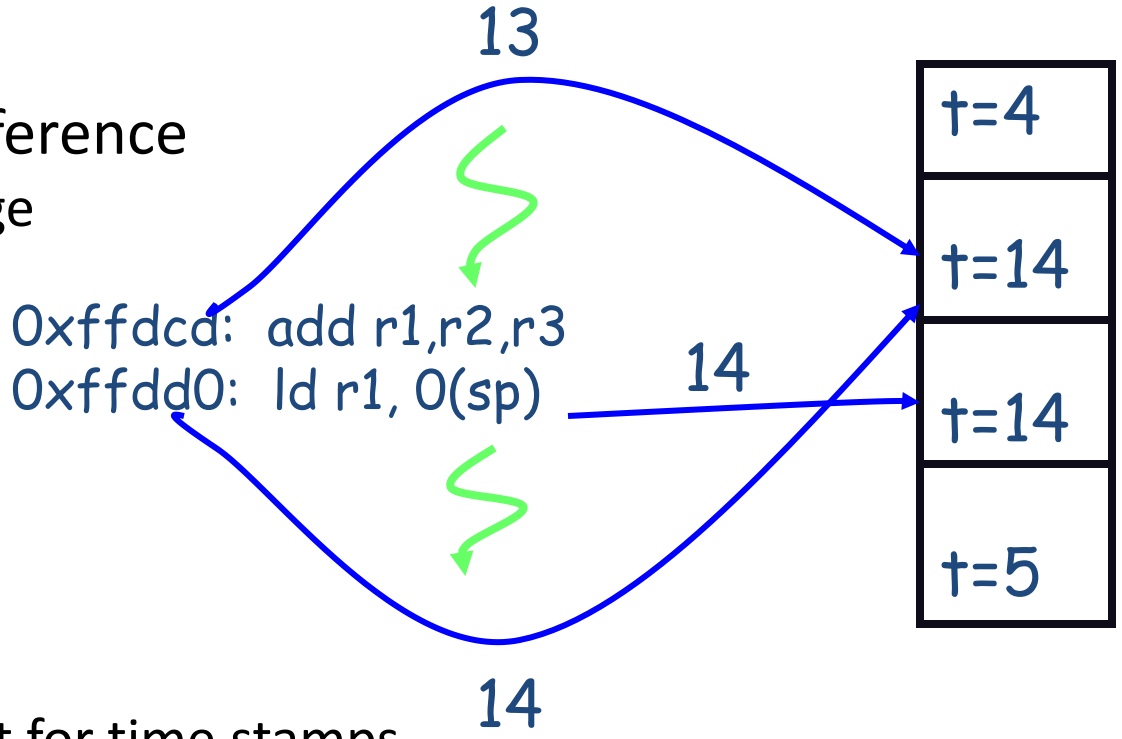
Ref: / Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: / Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Implementing Perfect LRU

- On every memory reference
  - time stamp each page
- At eviction time:
  - scan for oldest

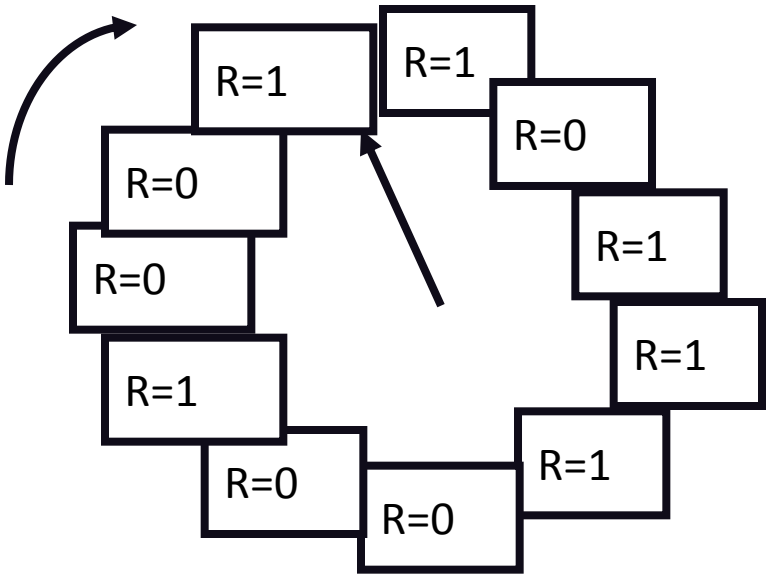


- Problems:
  - large page lists
  - no hardware support for time stamps
- "Sort of" LRU
  - do something simple & fast that finds an old page
  - LRU an approximation anyway, a little more won't hurt…

LRU in the real world: the clock algorithm

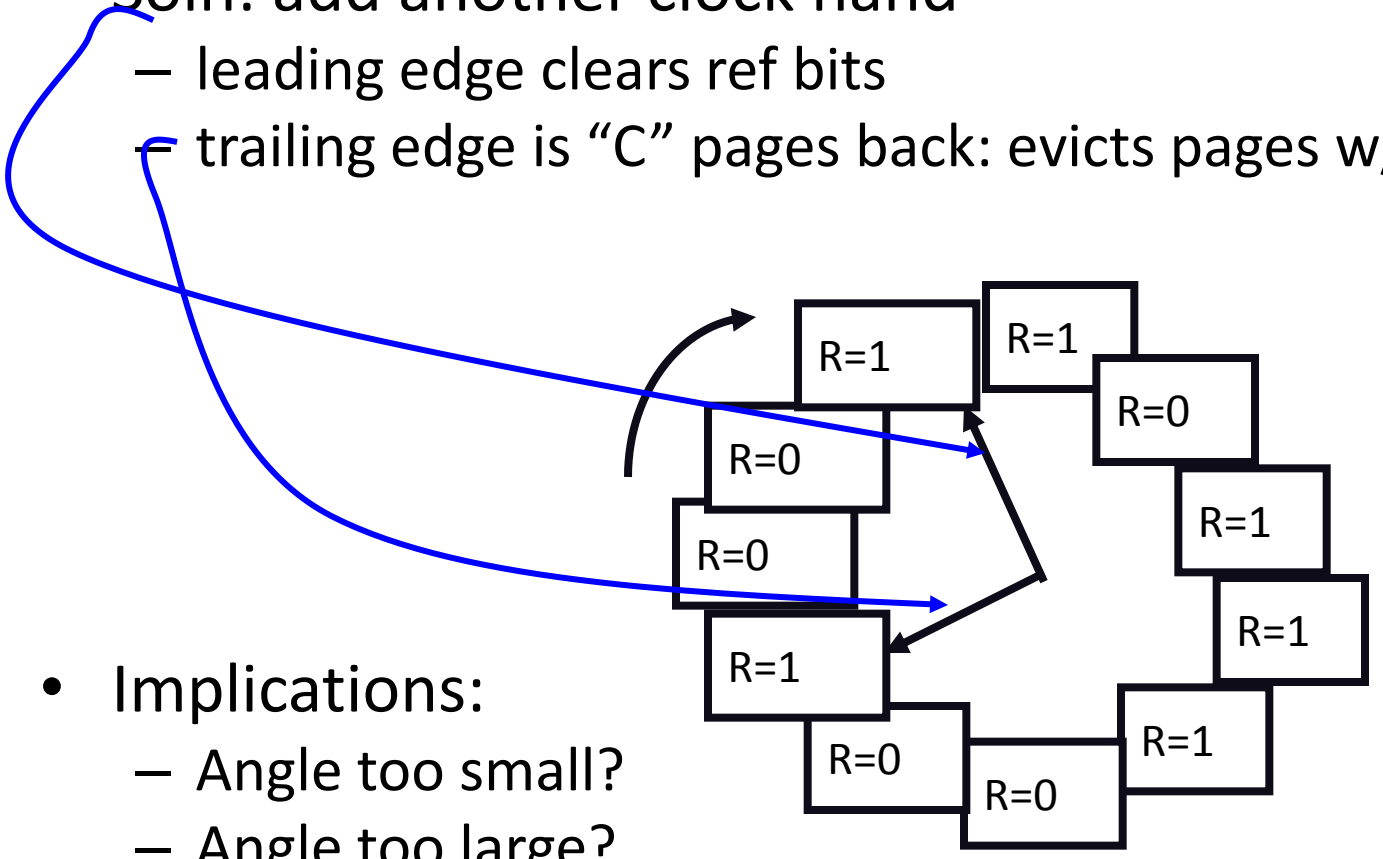
- Each page has reference bit
  - hardware sets on use, OS periodically clears
  - Pages with bit set used more recently than without.
- Algorithm: FIFO + skip referenced pages
  - keep pages in a circular FIFO list
  - scan: page's ref bit = 1, set to 0 & skip, otherwise evict.

- Hand sweeping slow?
  - Good sign or bad sign?
- Hand sweeping fast?



Problem: what happens as memory gets big?

- Soln: add another clock hand
  - leading edge clears ref bits
  - trailing edge is “C” pages back: evicts pages w/ 0 ref bit



- Implications:
  - Angle too small?
  - Angle too large?

BSD Unix: Clock algorithm in Action!

- use vmstat on SunOS/BSD unix to see
  - bigmachine: vmstat -s # -s: pages scanned by clock/second
    - 2*92853 pages examined by the clock daemon
    - 6 revolutions of the clock hand
    - 127878 pages freed by clock daemon
  - smallmachine: vmstat -s # smaller machine
    - 15086 revolutions of the clock hand # buy more mem!
    - 672474 forks

The clock algorithm improved

- Problem: crude & overly sensitive to sweeping interval
  - Infrequent? all pages look used.
  - Frequent? Lose too much usage information
  - Simple changes = more accurate & robust w/ ~same work
- Clock: 1 bit per page
  - when page used: set use bit
  - sweep: clear use bit
  - select page? FIFO + skip if use bit set
- Clock': n bits per page
  - when page used: set use bit
  - sweep: use_count = (use_bit << n-1) | (use_count >> 2)
    - (why shift?)
  - select page? take lowest use count

N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1⇒clear use and also clear counter (used in last sweep)
    - 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced

- How do we pick N?
  - Why pick large N? Better approx to LRU
    - If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - Otherwise might have to look a long way to find free page

- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - Clean pages, use N=1
    - Dirty pages, use N=2 (and write back to disk when N=1)

Clock Algorithms: Details

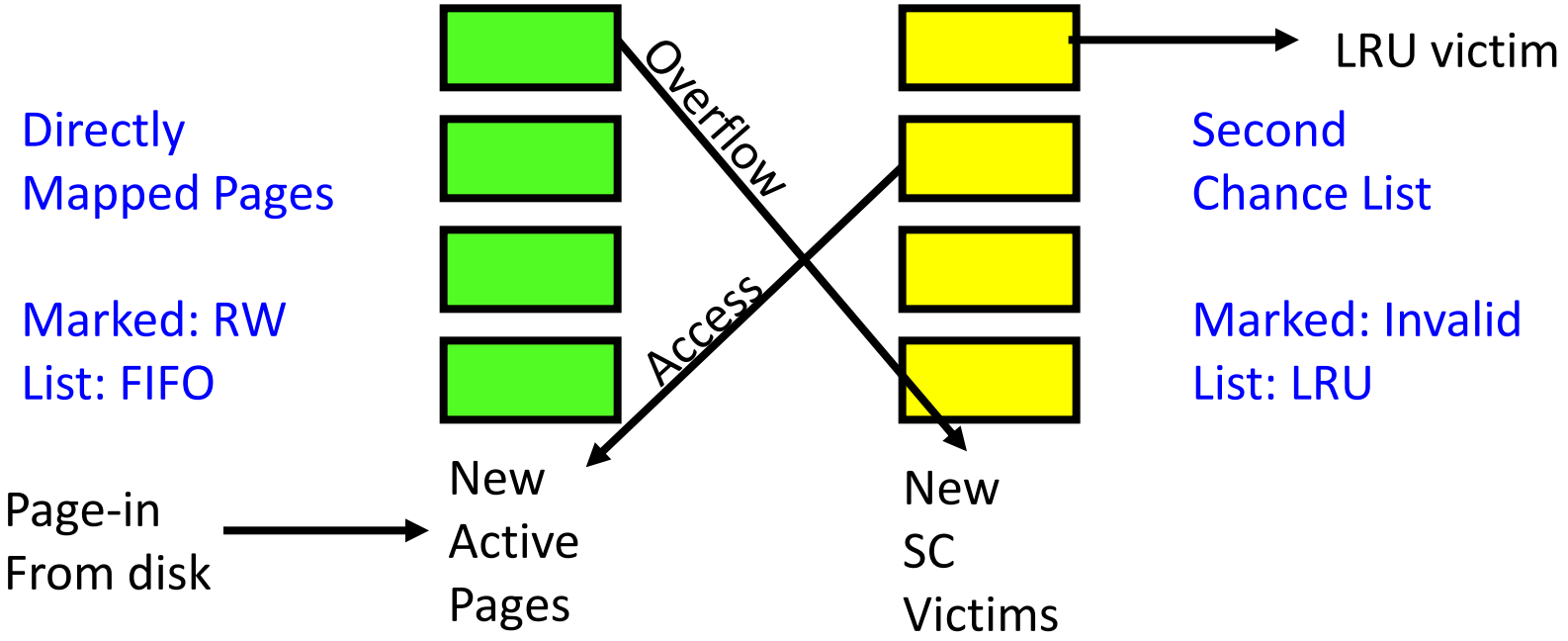
- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - For example for catching modifications to code pages!
- Do we really need hardware-supported "modified" bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    - Initially, mark all pages as read-only, even data pages
    - On write, trap to OS. OS sets software "modified" bit, and marks page as read-write.
    - Whenever page comes back in from disk, mark read-only

Clock Algorithms Details (continued)

- Do we really need a hardware-supported “use” bit?
  - No. Can emulate it similar to above:
    - Mark all pages as invalid, even if in memory
    - On read to invalid page, trap to OS
    - OS sets use bit, and marks page read-only
  - Get modified bit in same way as previous:
    - On write, trap to OS (either invalid or read-only)
    - Set use and modified bits, mark page read-write
  - When clock hand passes by, reset use and modified bits and mark page as invalid again

- Remember, however, that clock is just an approximation of LRU
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify an old page, not oldest page!
  - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)



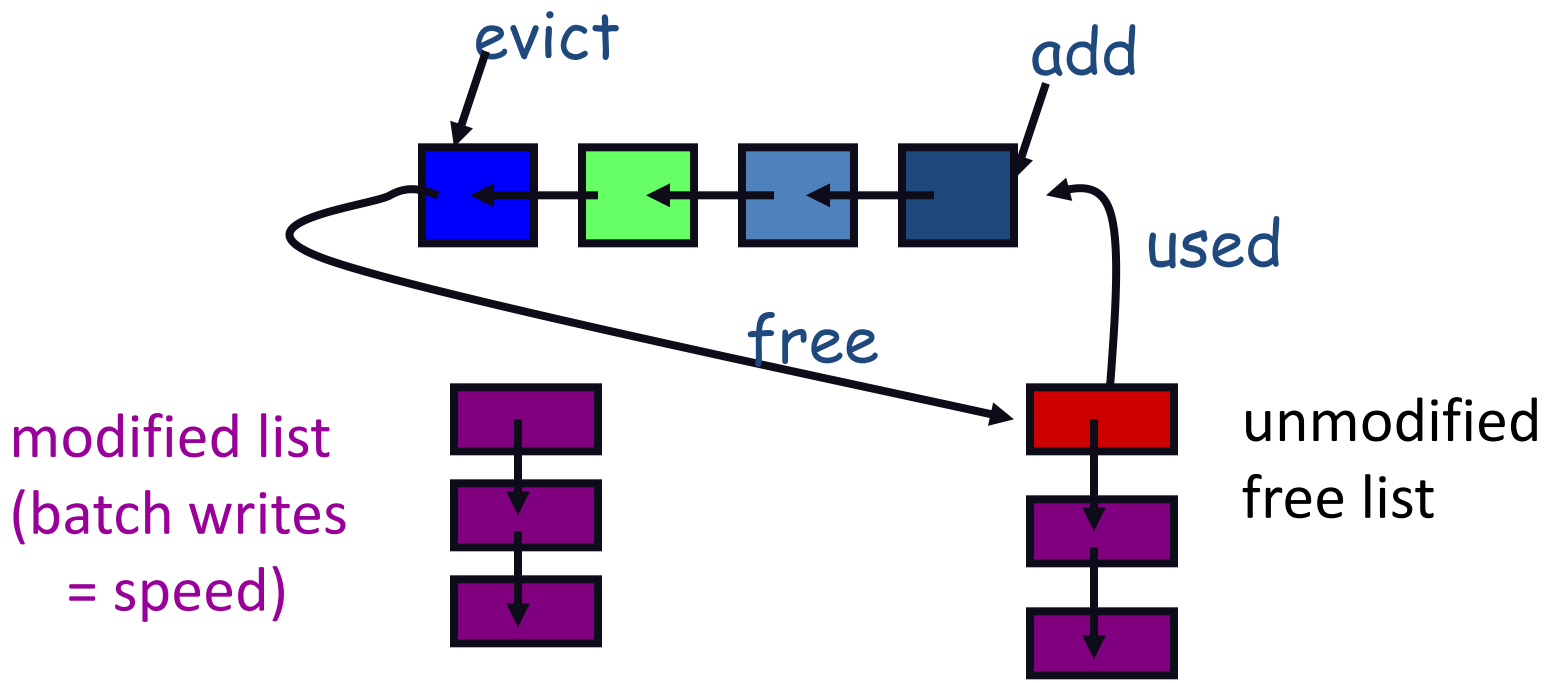
- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm

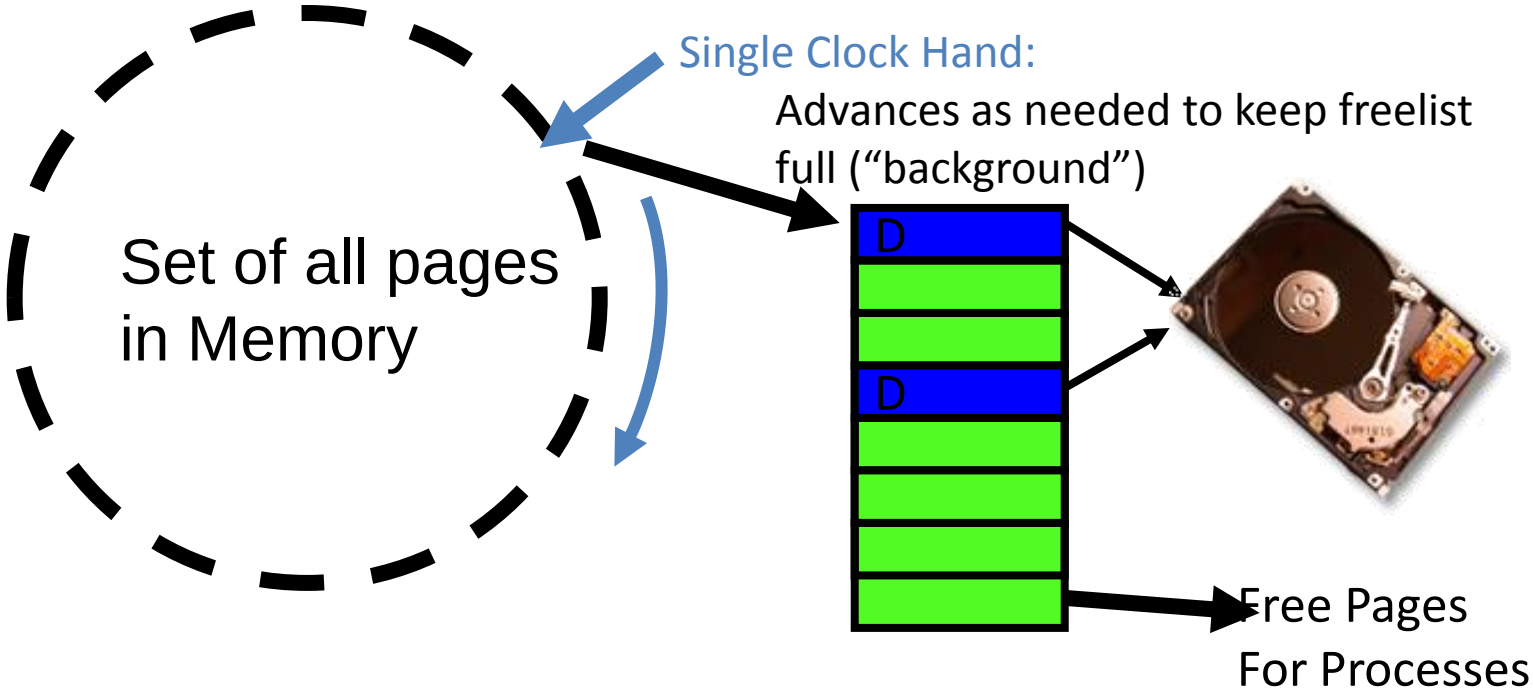
- How many pages for second chance list?
  - If 0 $\Rightarrow$ FIFO
  - If all $\Rightarrow$ LRU, but page fault on every page reference
- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- Question: why didn't VAX include "use" bit?
  - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
  - He later got blamed, but VAX did OK anyway

Another take: page buffering

- VMS:



Free List



- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique (“Pageout demon”)
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Demand Paging (more details)

- Does software-loaded TLB need use bit?
  Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page $\rightarrow$ physical page
  - Do we need a reverse mapping (i.e. physical page $\rightarrow$ virtual page)?
    - Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - Can't push page out to disk without invalidating all PTEs