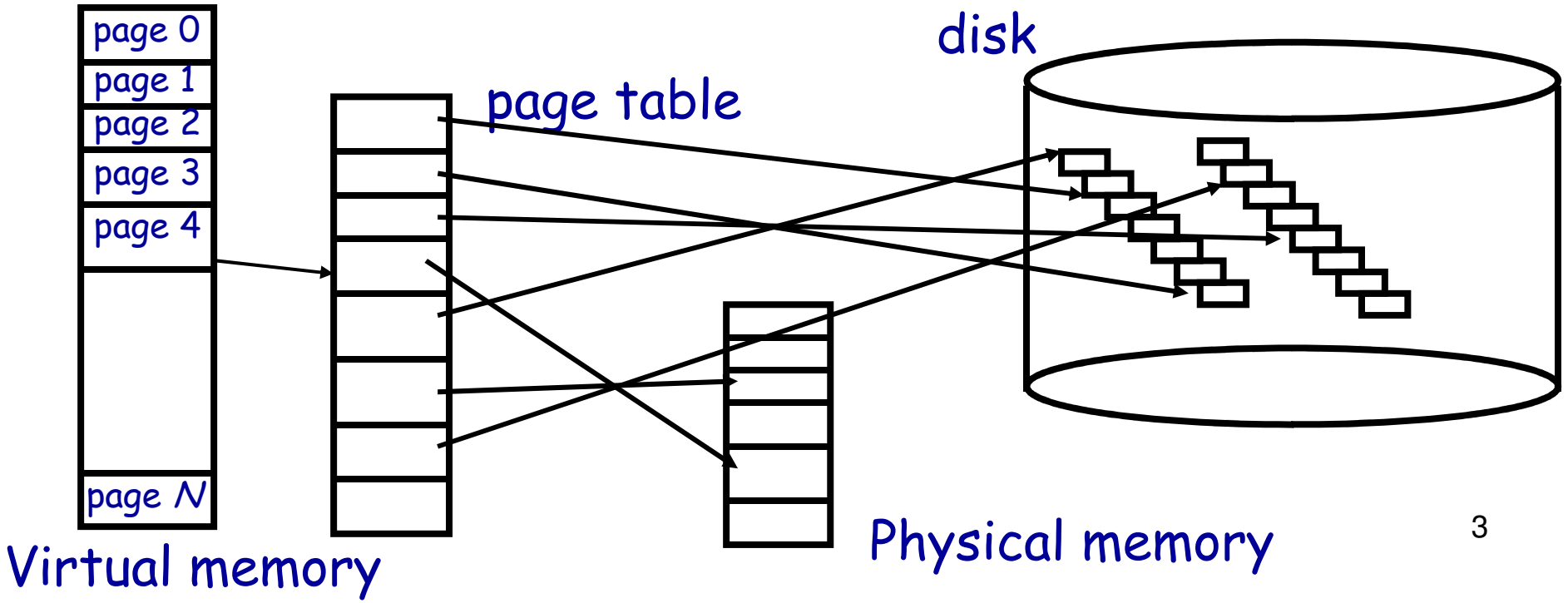
Virtual Memory

Goals for Today

- Virtual memory
- Mechanism
  - How does it work?
- Policy
  - What to replace?
  - How much to fetch?

What is virtual memory?

- Each process has illusion of large address space
  - 2^{32} for 32-bit addressing
- However, physical memory is much smaller
- How do we give this illusion to multiple processes?
  - Virtual Memory: some addresses reside in disk



Virtual memory

- Separates users logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation

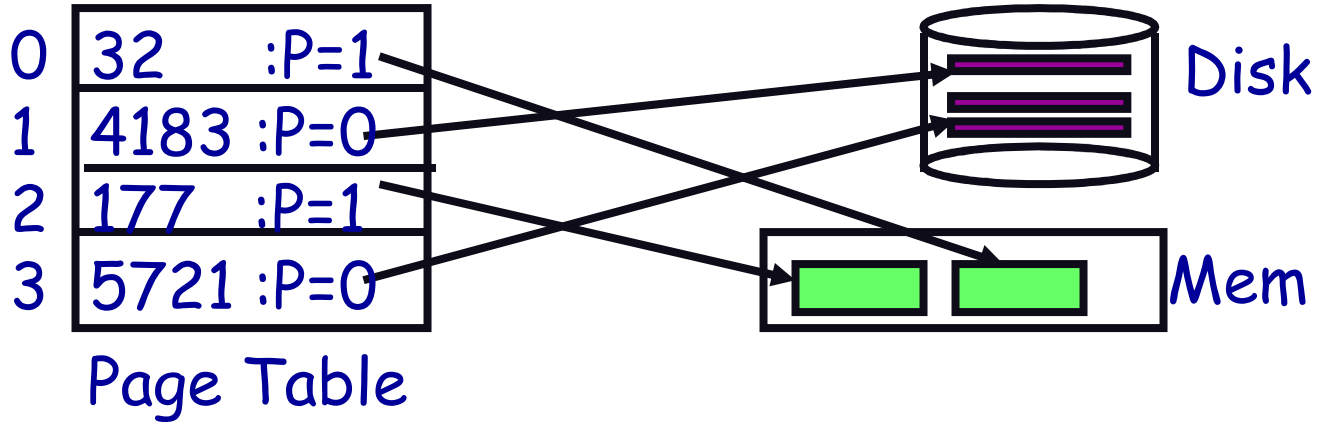
Swapping vs Paging

- Swapping
  - Loads entire process in memory , runs it, exit
  - Is slow (for big, long-lived processes)
  - Wasteful (might not require everything)
- Paging
  - Runs all processes concurrently, taking only pieces of memory (specifically, pages) away from each process
  - Finer granularity, higher performance
  - Paging completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

- The verb “to swap” is also used to refer to pushing contents of a page out to disk in order to bring other content from disk; this is distinct from the noun “swapping”

How does VM work?

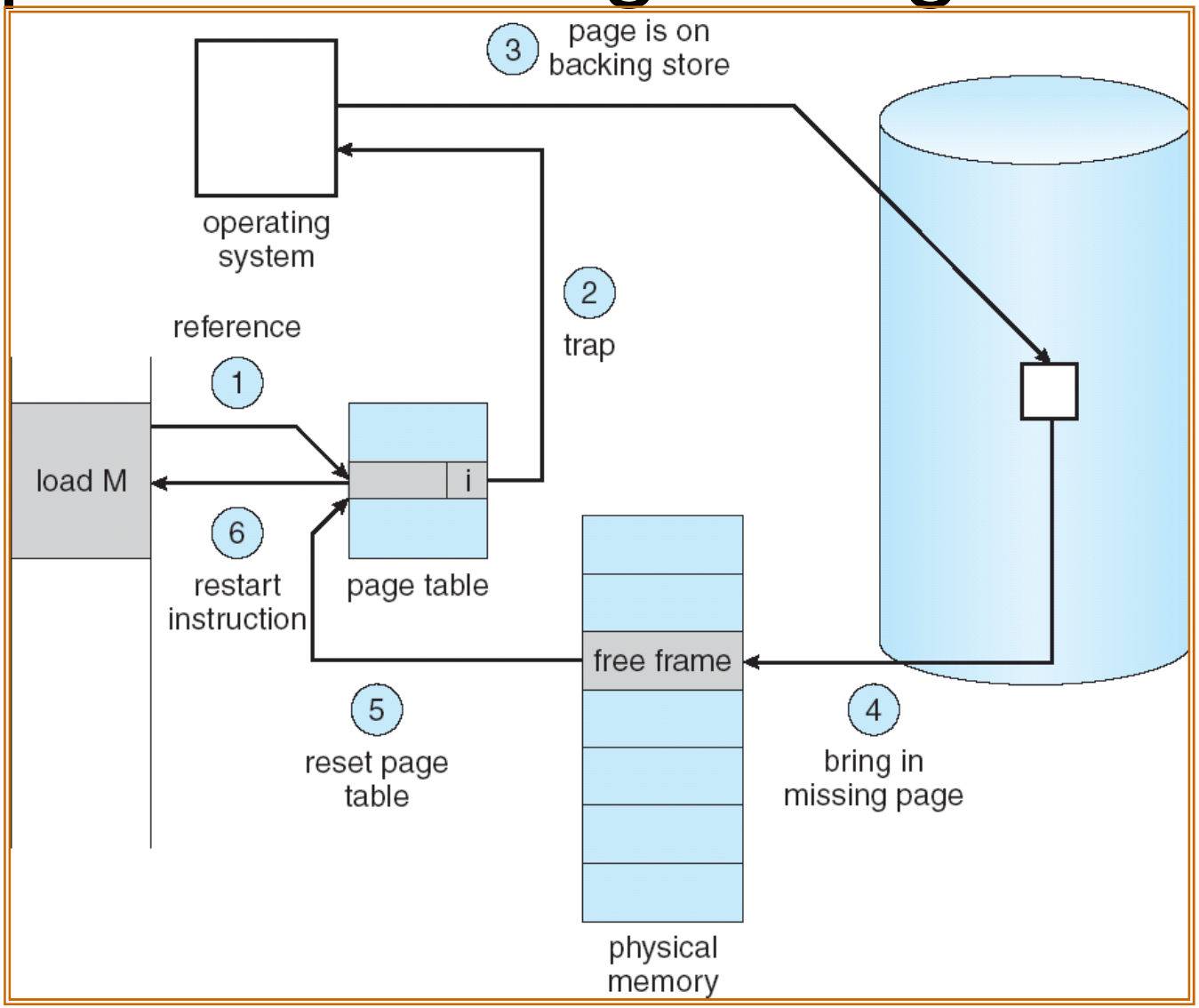
- Modify Page Tables with another bit ("is present")
  - If page in memory, *is_present = 1*, else *is_present = 0*
  - If page is in memory, translation works as before
  - If page is not in memory, translation causes a **page fault**



Page Faults

- On a page fault:
  - OS finds a free frame, or evicts one from memory (which one?)
    - Want knowledge of the future?
  - Issues disk request to fetch data for page (what to fetch?)
    - Just the requested page, or more?
  - Block current process, context switch to new process (how?)
    - Process might be executing an instruction
  - When disk completes, set present bit to 1, and current process in ready queue

Steps in Handling a Page Fault



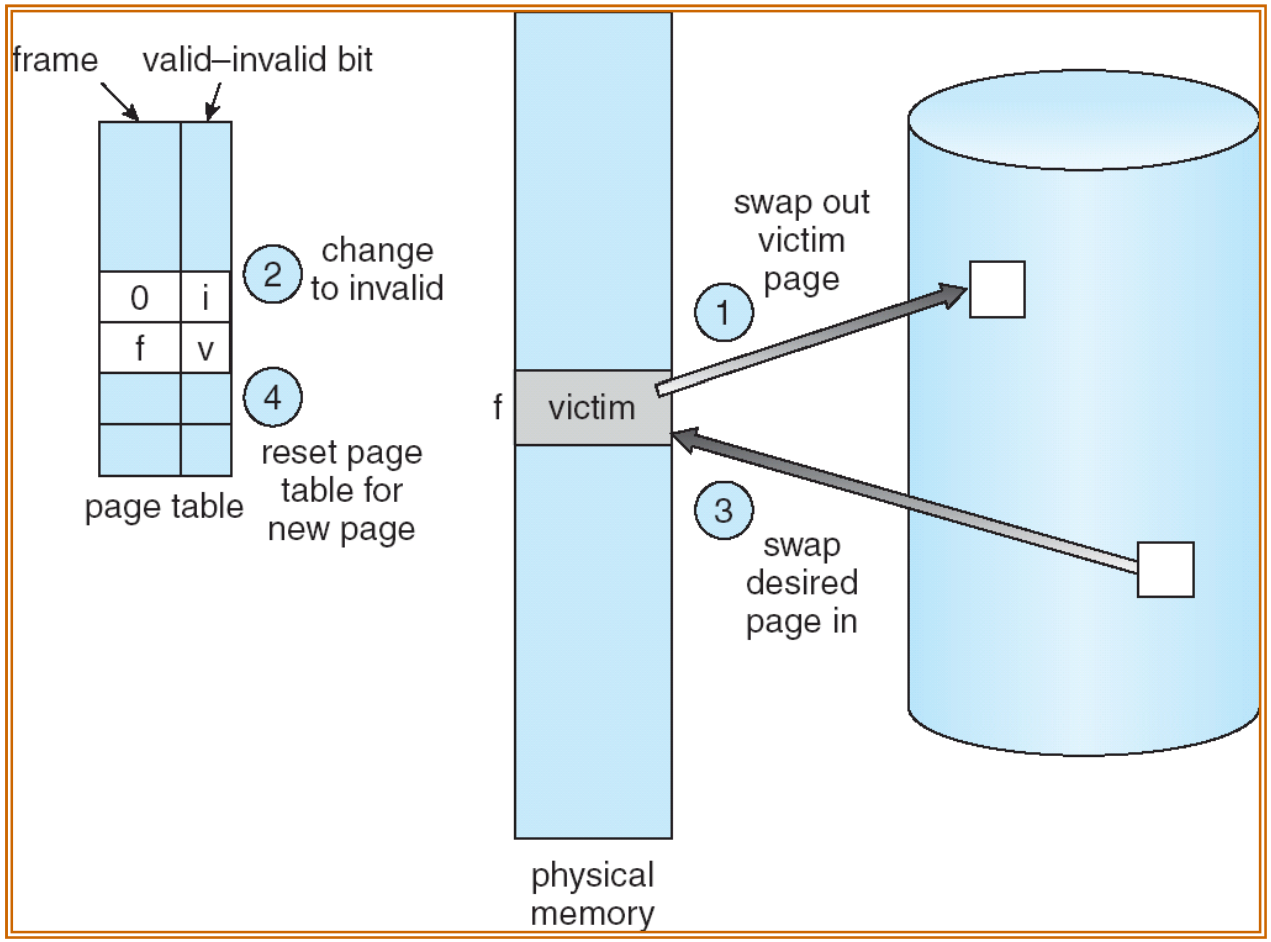
What to replace?

- What happens if there is no free frame?
  - find a suitable page in memory, swap it out
- Page Replacement
  - When process has used up all frames it is allowed to use
  - OS must select a page to eject from memory to allow new page
  - The page to eject is selected using the Page Replacement Algo
- Goal: Select page that minimizes future page faults

Modified/Dirty Bits

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk, non-modified pages can always be brought back from the original source
  - Process text segments are rarely modified, can bring pages back from the program image stored on disk

Page Replacement

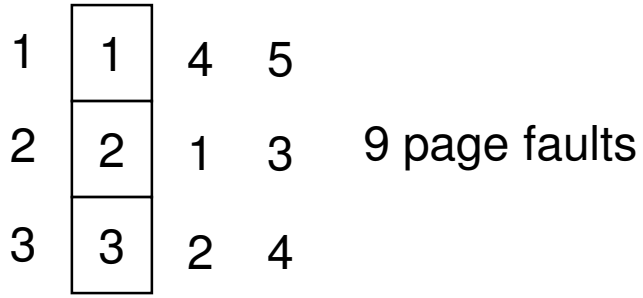


Page Replacement Algorithms

- Random: Pick any page to eject at random
  - Used mainly for comparison
- FIFO: The page brought in earliest is evicted
  - Ignores usage
- OPT: Belady's algorithm
  - Select page not used for longest time
- LRU: Evict page that hasn't been used the longest
  - Past could be a good predictor of the future
- MRU: Evict the most recently used page
- LFU: Evict least frequently used page

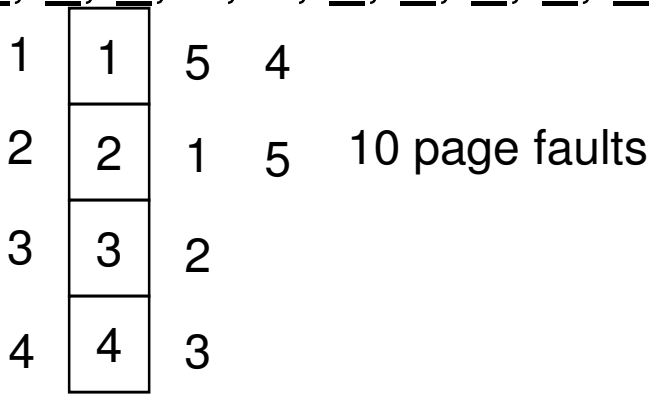
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



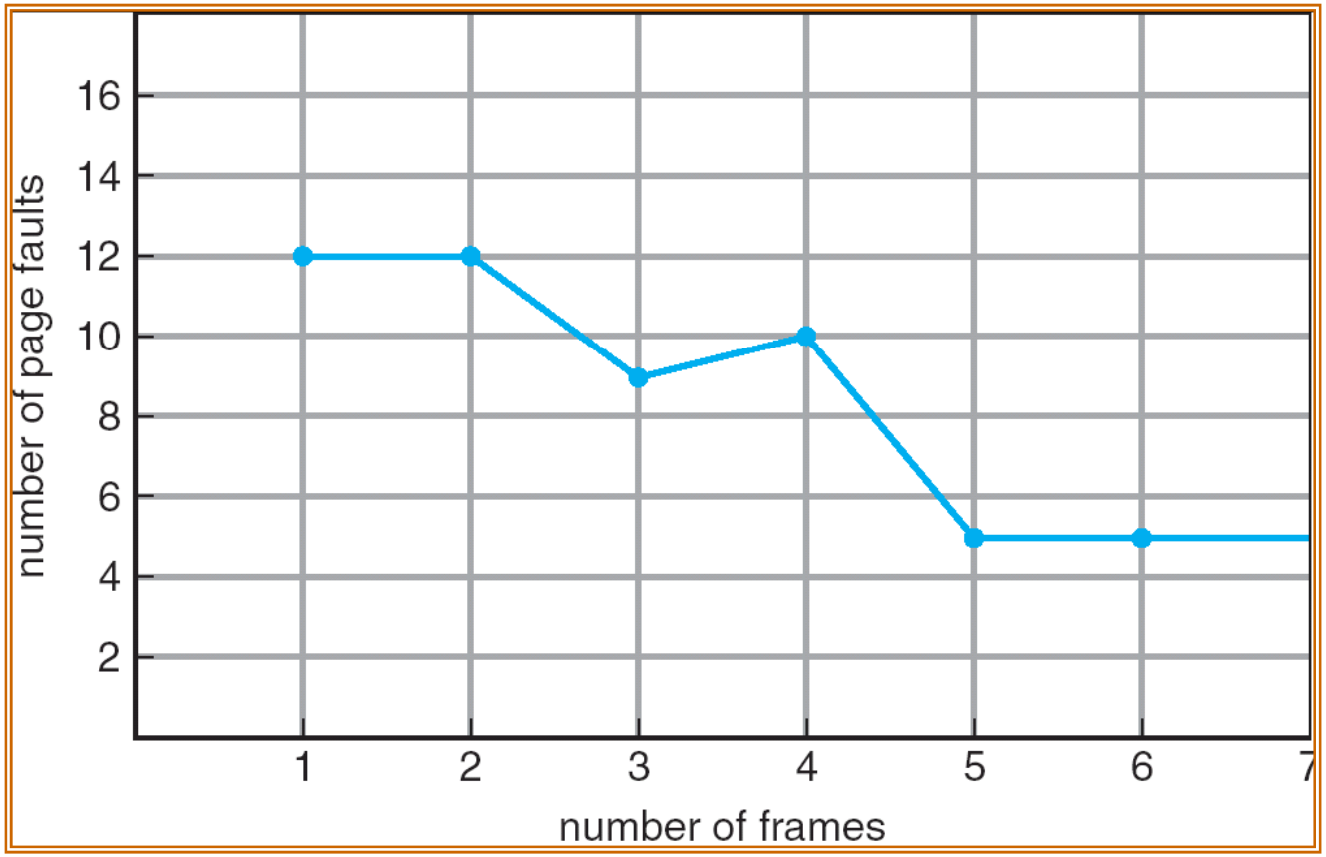
9 page faults

- 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



10 page faults

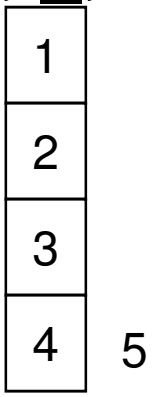
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



4

5

6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Example: FIFO, OPT

Reference stream is A B C A B D A D B C

OPTIMAL

A B C A B D A D B C B

5 Faults

toss C

toss A or D

A
B
C
D
A
B
C

FIFO

A B C A B D A D B C B

7 Faults

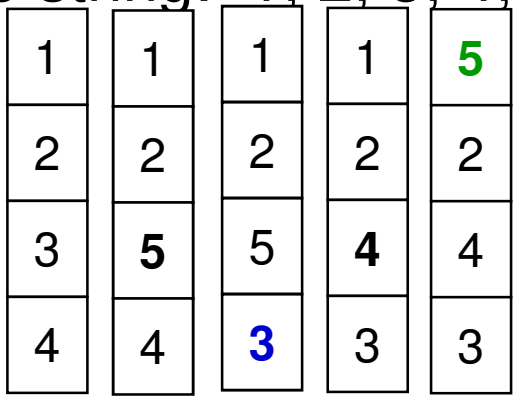
toss A

toss ?

OPT Approximation

- In real life, we do not have access to the future page request stream of a program
  - No crystal ball, no way to know definitively which pages a program will access
- So we need to make a best guess at which pages will not be used for the longest time

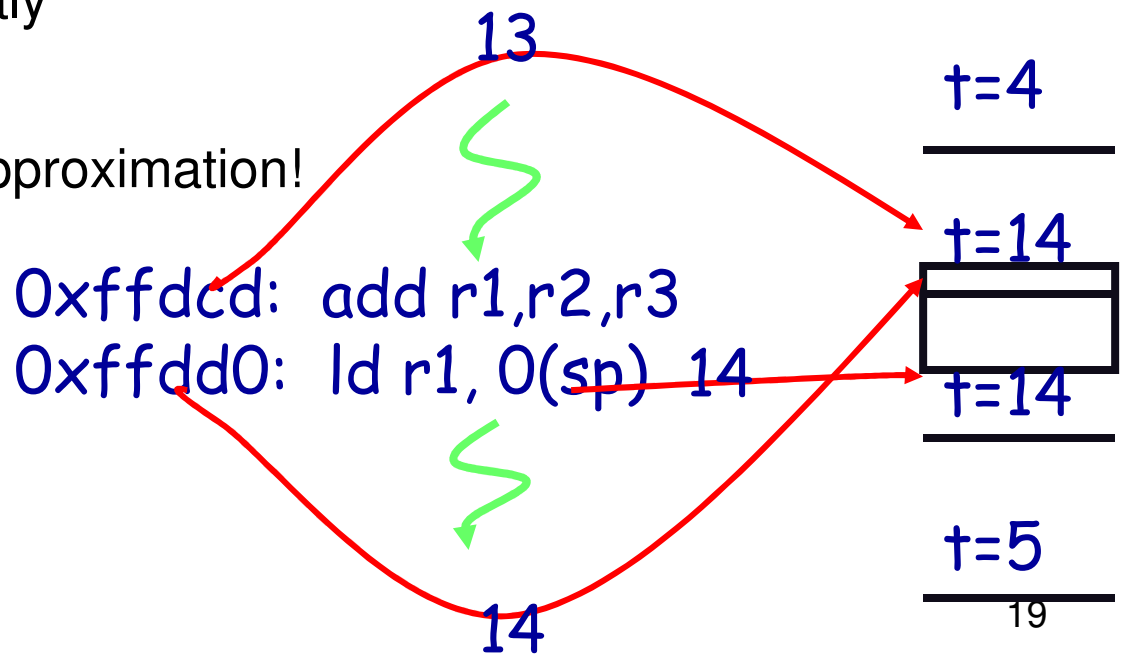
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**
- 

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

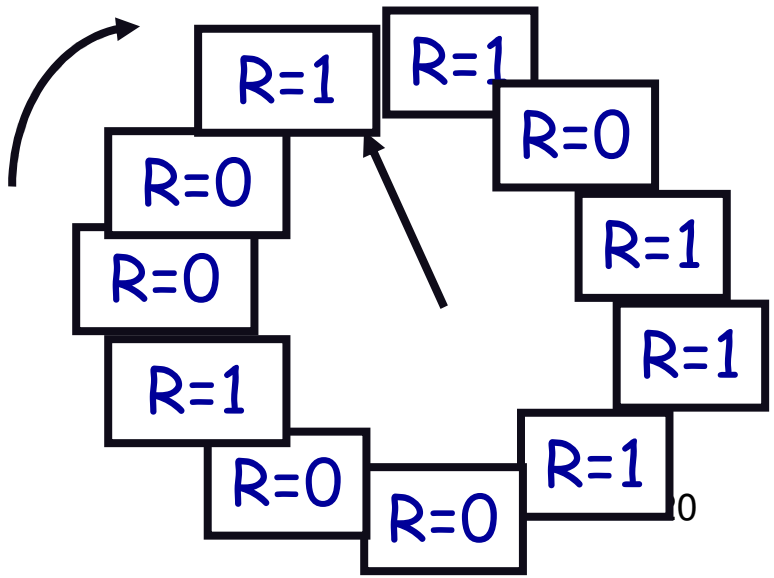
Implementing Perfect LRU

- On reference: Time stamp each page
- On eviction: Scan for oldest frame
- Problems:
  - Large page lists
  - Timestamps are costly
- Approximate LRU
  - LRU is already an approximation!



LRU: Clock Algorithm

- Each page has a reference bit
  - Set on use, reset periodically by the OS
- Algorithm:
  - FIFO + reference bit (keep pages in circular list)
    - Scan: if ref bit is 1, set to 0, and proceed. If ref bit is 0, stop and evict.
- Problem:
  - Low accuracy for large memory

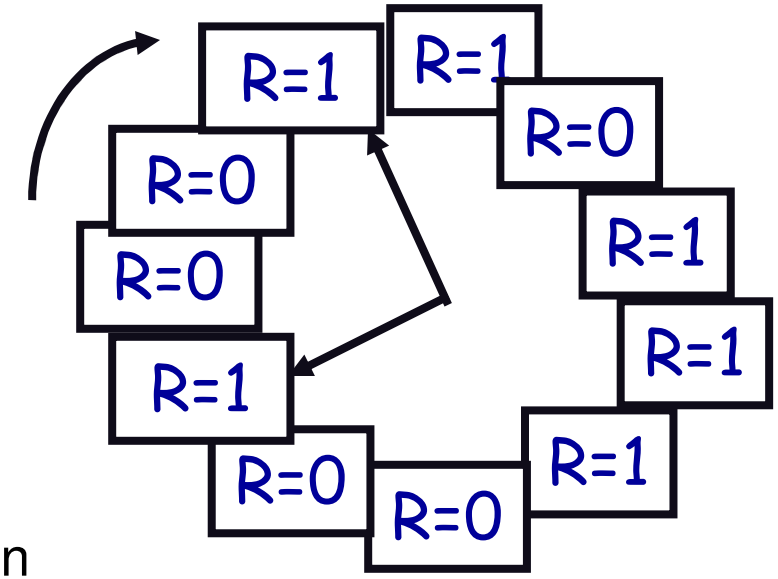


LRU with large memory

- Solution: Add another hand
  - Leading edge clears ref bits
  - Trailing edge evicts pages with ref bit 0

- What if angle small?
- What if angle big?

- Sensitive to sweeping interval and angle
  - Fast: lose usage information
  - Slow: all pages look used



Other Algorithms

- MRU: Remove the most recently touched page
  - Works well for data accessed only once, e.g. a movie file
  - Not a good fit for most other data, e.g. frequently accessed items
- LFU: Remove page with lowest count
  - No track of when the page was referenced
  - Use multiple bits. Shift right by 1 at regular intervals.
- MFU: remove the most frequently used page
- LFU and MFU do not approximate OPT well

Allocating Pages to Processes

- Global replacement
  - Single memory pool for entire system
  - On page fault, evict oldest page in the system
  - Problem: lack of performance isolation
- Local (per-process) replacement
  - Have a separate pool of pages for each process
  - Page fault in one process can only replace pages from its own process
  - Problem: might have idle resources

Thrashing

- Def: Excessive rate of paging
  - May stem from lack of resources
  - More likely, caused by bad choices of the eviction algorithm
  - Keep throwing out page that will be referenced soon
  - So, they keep accessing memory that is not there

- Why does it occur?
  - Poor locality, past != future
  - There is reuse, but process does not fit model
  - Too many processes in the system

Working Set

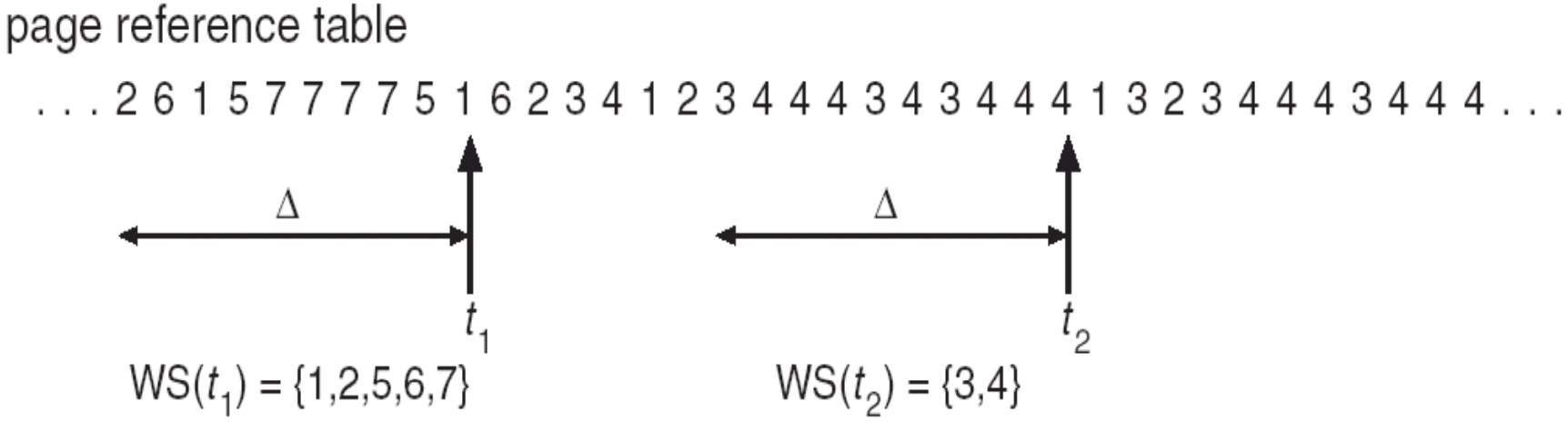
- Peter Denning, 1968
  - He uses this term to denote memory locality of a program

  Def: pages referenced by process in last Δ time-units comprise its working set

  For our examples, we usually discuss WS in terms of Δ, a "window" *in the page reference string*. But while this is easier on paper it makes less sense in practice!

  In real systems, the window should probably be a period of time, perhaps a second or two.

Working Sets



- The working set size is *num* pages in the working set
  - the number of pages touched in the interval [t-Δ+1..t].
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - Within that period of time, you will have a larger working set size.
- Goal: keep WS for each process in memory.
  - E.g. If Σ WS_i for all *i runnable* processes > physical memory, then suspend a process

Working Set Approximation

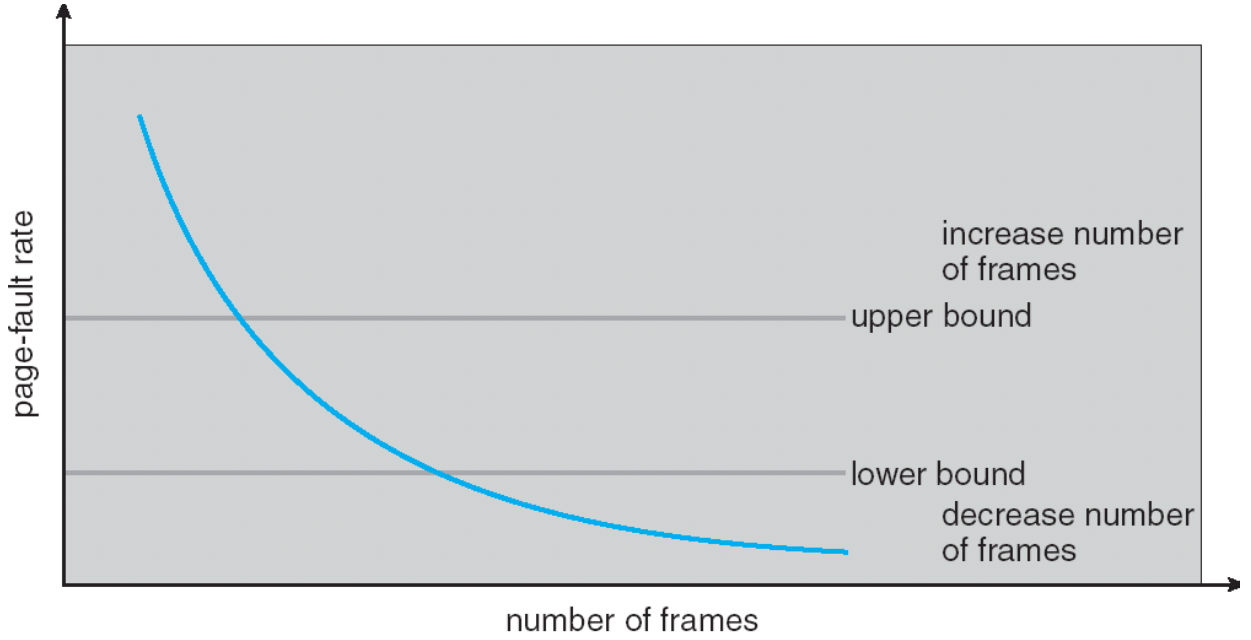
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Cannot tell (within interval of 5000) where reference occured
- Improvement = 10 bits and interrupt every 1000 time units

Using the Working Set

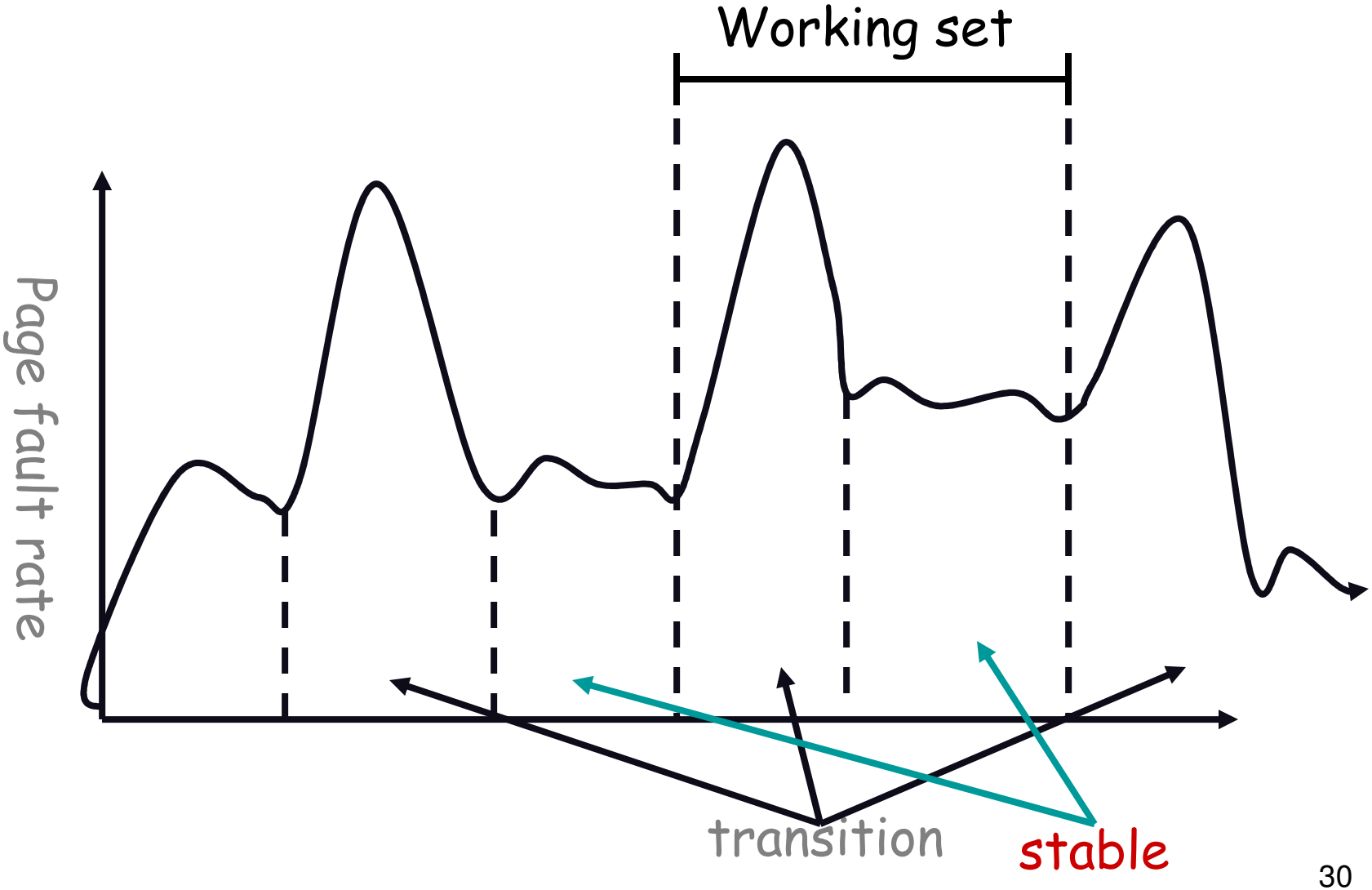
- Used mainly for prepaging
  - Pages in working set are a good approximation
- In Windows processes have a *max* and *min* WS size
  - At least *min* pages of the process are in memory
  - If > *max* pages in memory, on page fault a page is replaced
  - Else if memory is available, then WS is increased on page fault
  - The *max* WS can be specified by the application

Page Fault Frequency

- Thrashing viewed as poor ratio of fetch to work
- PFF = page faults / instructions executed
- if PFF rises above threshold, process needs more memory
  - not enough memory on the system? Swap out.
- if PFF sinks below threshold, memory can be taken away



Working Sets and Page Fault Rates



OS and Paging

- Process Creation:
  - Allocate space and initialize page table for program and data
  - Allocate and initialize swap area
  - Info about PT and swap space is recorded in process table
- Process Execution
  - Reset MMU for new process
  - Flush the TLB
  - Bring processes' pages in memory
- Page Faults
- Process Termination
  - Release pages