Chapter 9: Page Replacement Policies

CSCI 3753 Operating Systems

William Mortl & Prof. Rick Han

University of Colorado Boulder

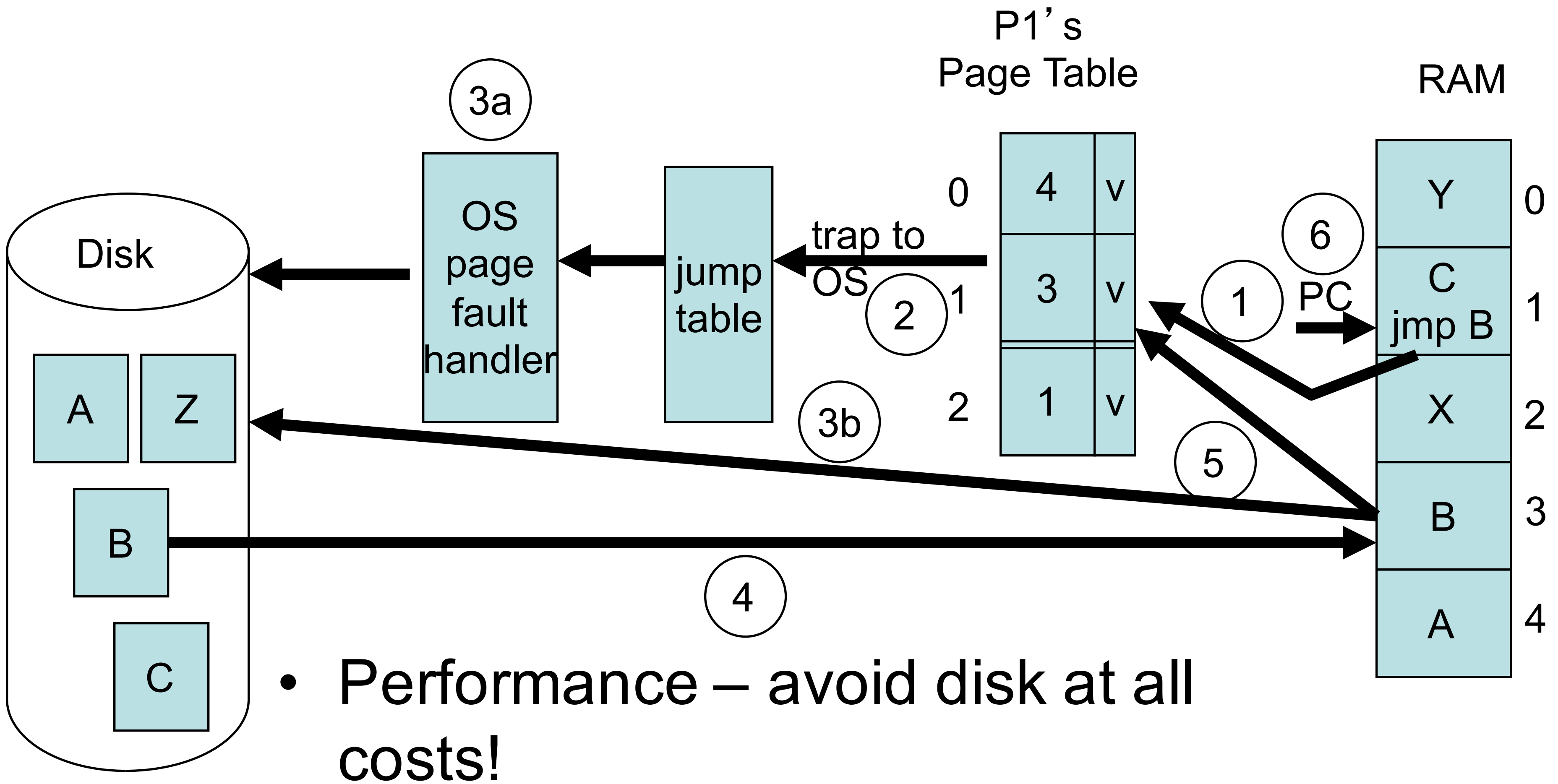
Recap

- Sparse address space => very empty page table
  - Inverted page tables – a single table for all physical memory
    - Use hashing + linked lists to speed search of IPTs
  - Hashed page tables = hashing + linked lists
- Segmentation
  - Divide logical address space into variably sized segments
  - segment table translates to physical addresses
  - Base and limit registers store location and size of segment

Recap

- On-demand paging = keep a subset of pages in memory
  - These are the "most important" pages currently being used by a process
  - As pages are needed, they are paged in from disk
  - Use a valid bit in page table to record whether each page is in memory or not
  - Advantages: decouple size of virtual memory from size of physical memory, fit in more processes in physical memory, sparseness no longer a problem, decrease swap time

On-Demand Paging & Page Replacement



- Performance – avoid disk at all costs!
  - Page replacement policy: set dirty bit to modified so as to reduce disk writes

Page Replacement Policies

- Which page does OS choose to replace?
  - Assume we replace a clean page before a dirty page, to save on a disk write
  - Still, there may still be multiple clean pages (dirty bit = 0)
    - Which clean page to evict?
    - Evicting the wrong clean page that is referenced again soon, will cause a page fault to page it back in = performance hit

Page Replacement Policies

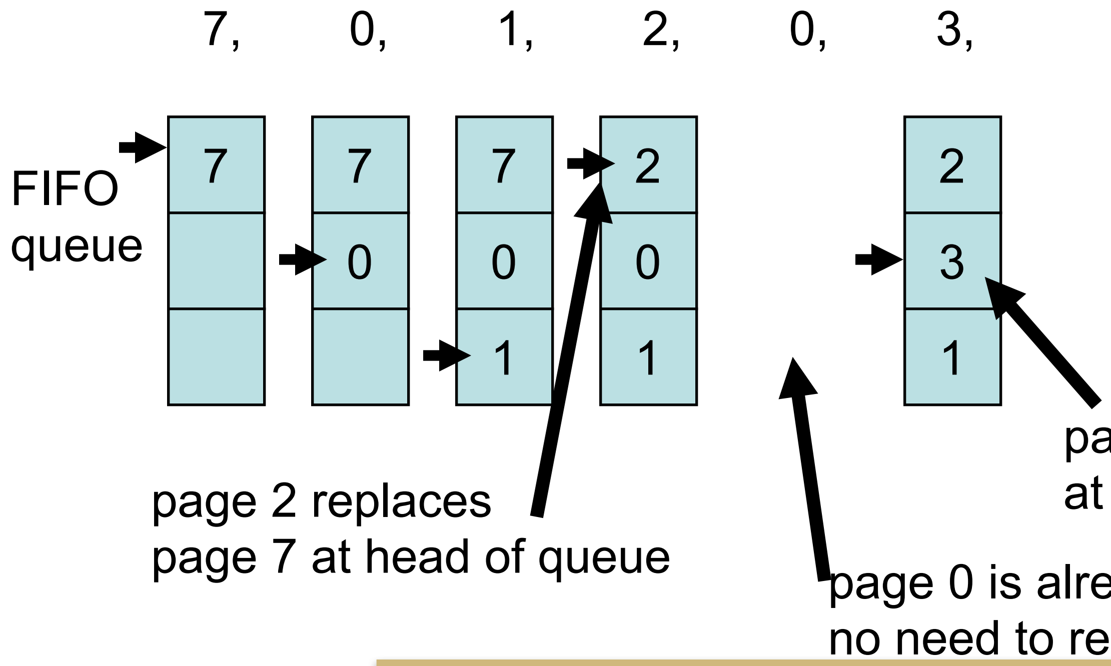
- Which page does OS choose to replace?
  - if all pages are dirty, again need a policy to decide which of the dirty pages to replace
  - In general, I need to develop a page replacement algorithm that works with a mixture of clean and dirty pages

Page Replacement Policies

- Some page replacement policies:
  - FIFO
  - OPT
  - LRU (least recently used)
- all of the above are usually evaluated against a *reference string* of page accesses,
  - e.g. a representative trace of page demands, to see how many page faults are generated
- algorithm with lowest # of page faults is most desirable

FIFO Page Replacement

- FIFO - create a FIFO queue of all pages in memory
  - example reference string: 7, 0, 1, 2, 0, 3, ...
  - assume also that there are 3 frames of memory total


7, 0, 1, 2, 0, 3,
FIFO queue
7
7 0
7 0 1
2 0 1
2 3 1
page 2 replaces page 7 at head of queue
page 0 is already in memory, no need to replace
page 3 replaces page 0 at head of FIFO queue

Note: pages 7, 0, 1, 2, and 3 all caused page faults

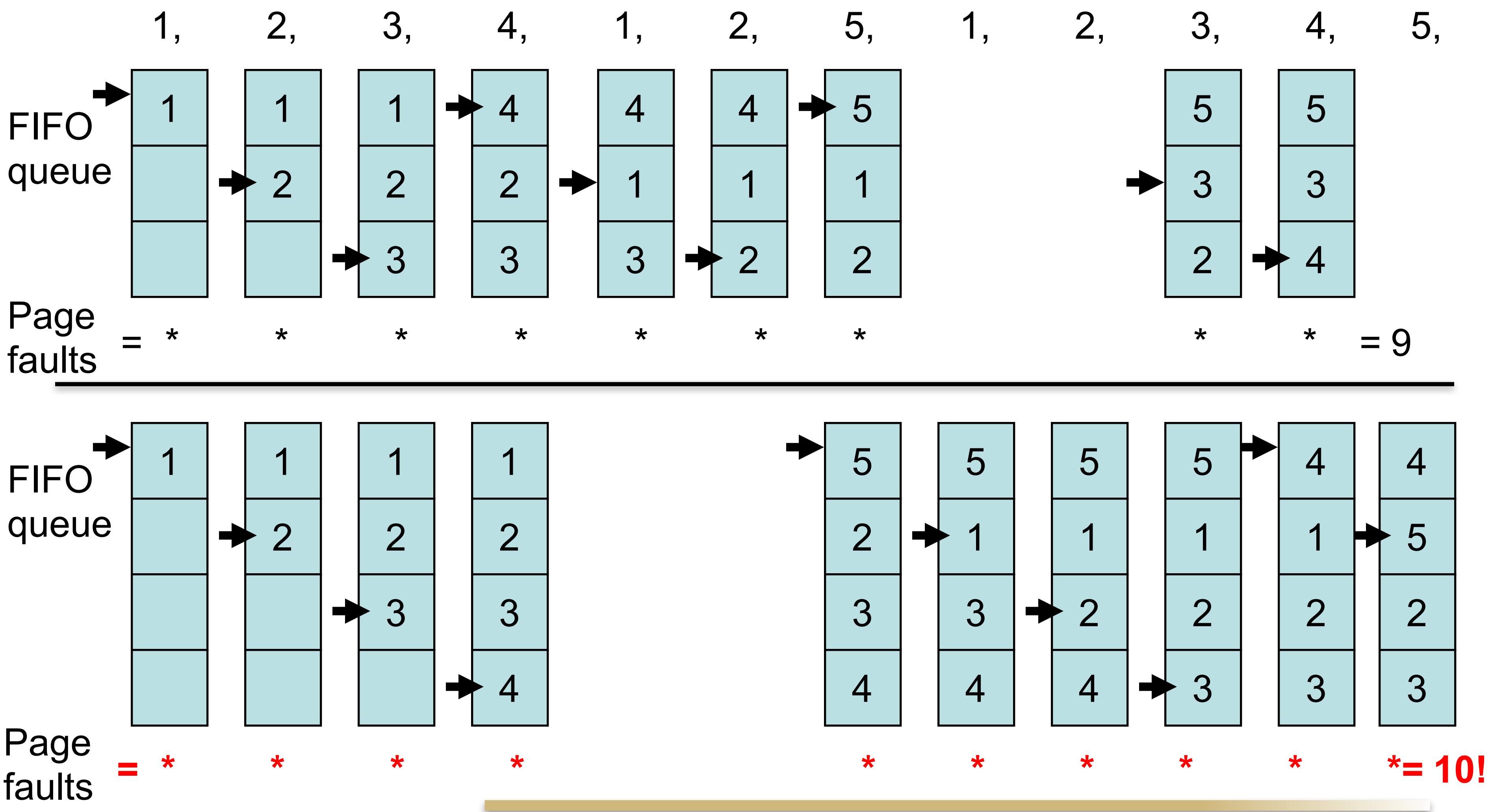
FIFO Page Replacement

- FIFO is easy to understand and implement
- Drawback #1: FIFO performance can be poor
  - If the replaced page 7 was active/frequently referenced, then page 7 will be referenced again soon, causing a page fault because it's not in memory
  - In the worst case, each page that is paged out could be the one that is referenced next, leading to a high page fault rate
- Ideally, keep around the pages that are about to be used next – this is the OPT algorithm (next slide)

FIFO Page Replacement

- Drawback #2: FIFO can exhibit Belady's anomaly
  - when the # of frames of RAM allocated to a task is increased (larger FIFO queue), normally we would expect fewer page faults
    - Because there are more frames in the FIFO queue, so there should be a greater chance of finding a page in the queue, hence fewer page faults
  - but with Belady's anomaly, sometimes we see that there are more page faults! This can happen with a FIFO policy.

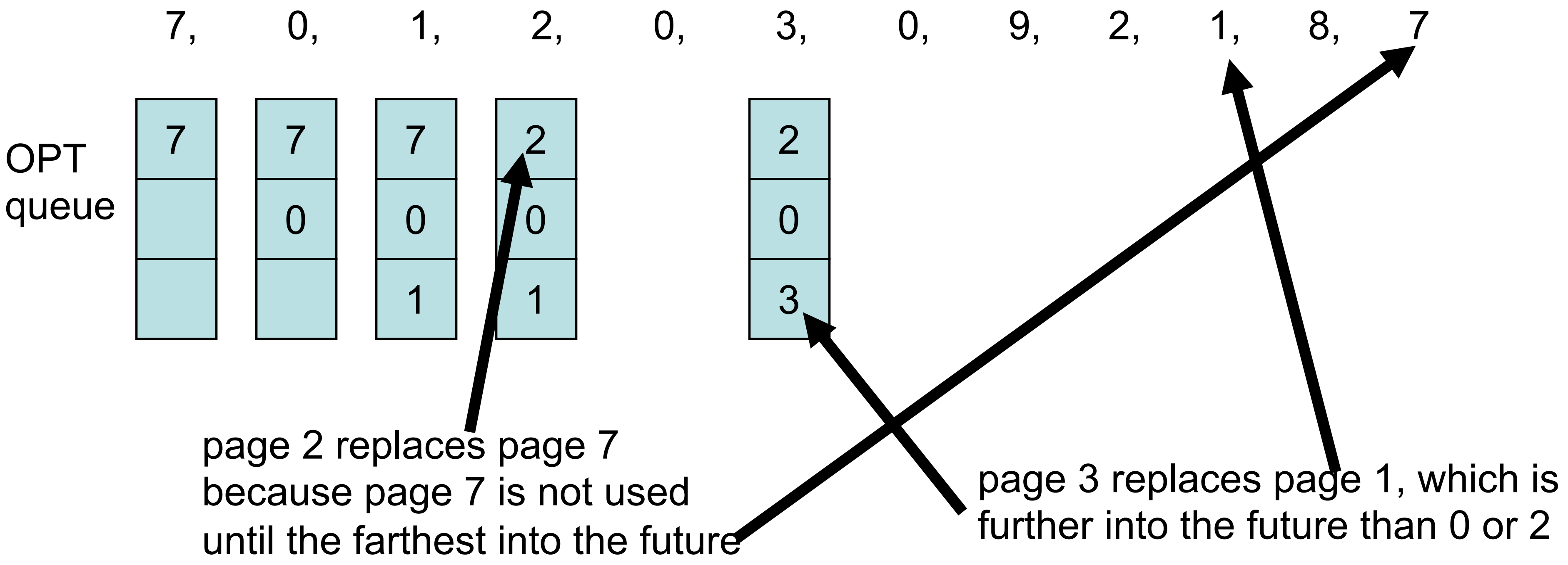
Belady's Anomaly



OPT Page Replacement

- Replace the page that will not be referenced for the longest time
- Why is this OPTimal in terms of generating the lowest possible page fault rate?
  - with FIFO, if I evict a page that is used immediately next, then that causes the highest page fault rate
  - So the opposite is to evict a page that is used the furthest in the future, which causes the fewest page faults
  - Any other policy will be worse than OPT, i.e. evict a page that will be needed sooner, hence higher page fault rate!

OPT Page Replacement



7, 0, 1, 2, 0, 3, 0, 9, 2, 1, 8, 7
OPT queue
7
7 0
7 0 1
2 0 1
2 0 3
page 2 replaces page 7 because page 7 is not used until the farthest into the future
page 3 replaces page 1, which is further into the future than 0 or 2

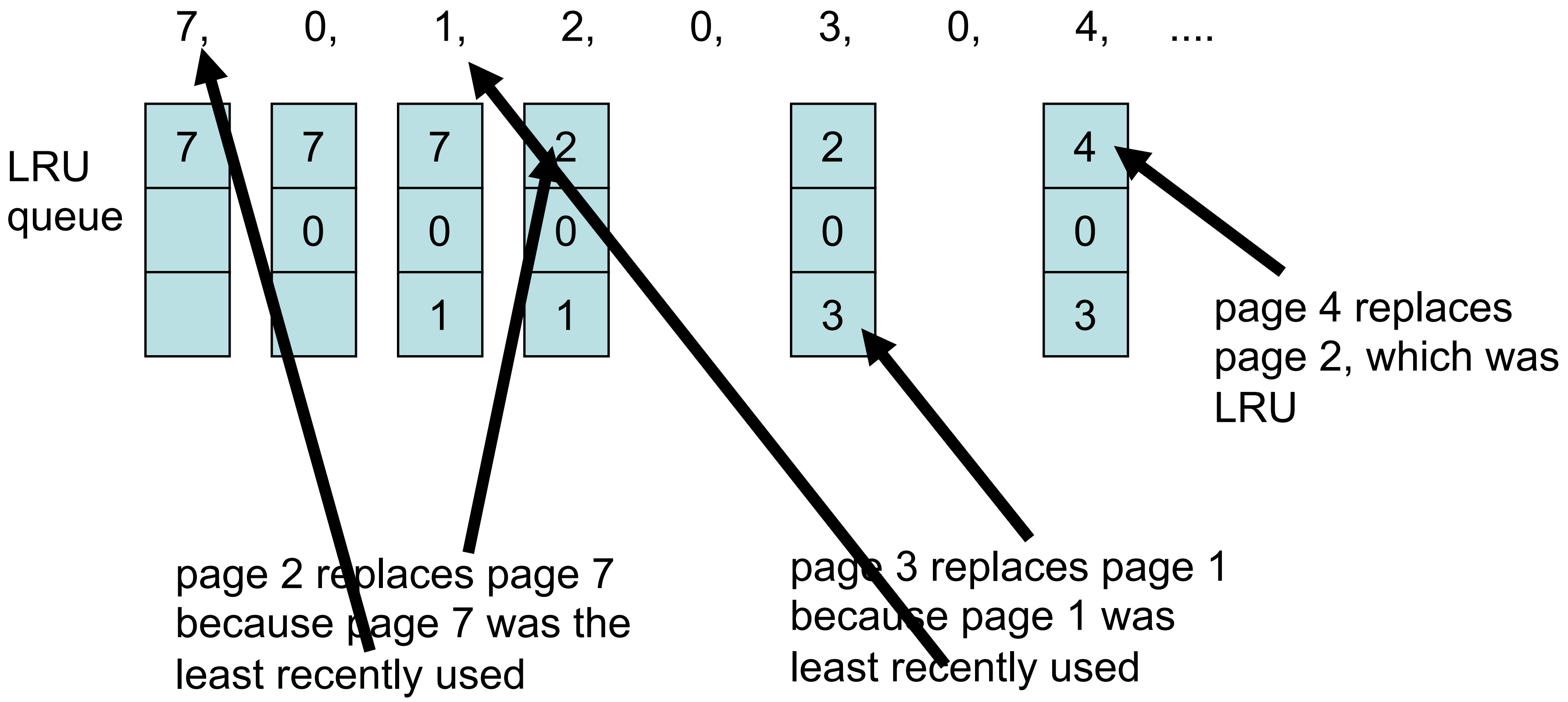
- Problem with OPT?
  - OPT requires future knowledge, i.e. need to know the pattern of future page accesses
    - Many apps will execute non-deterministically, with unknown page access patterns

LRU Page Replacement

- LRU = Least Recently Used
  - use the past to predict the future
    - if a page wasn't used recently, then it is unlikely to be used again in the near future
    - if a page was used recently, then it is likely to be used again in the near future
    - so select a victim that was least recently used
  - approximation of OPT
    - page fault rate LRU > OPT, but LRU < FIFO
  - variations of LRU are popular
  - LRU is not subject to Belady's Anomaly – why?
    - See textbook for explanation

LRU Page Replacement

- LRU example

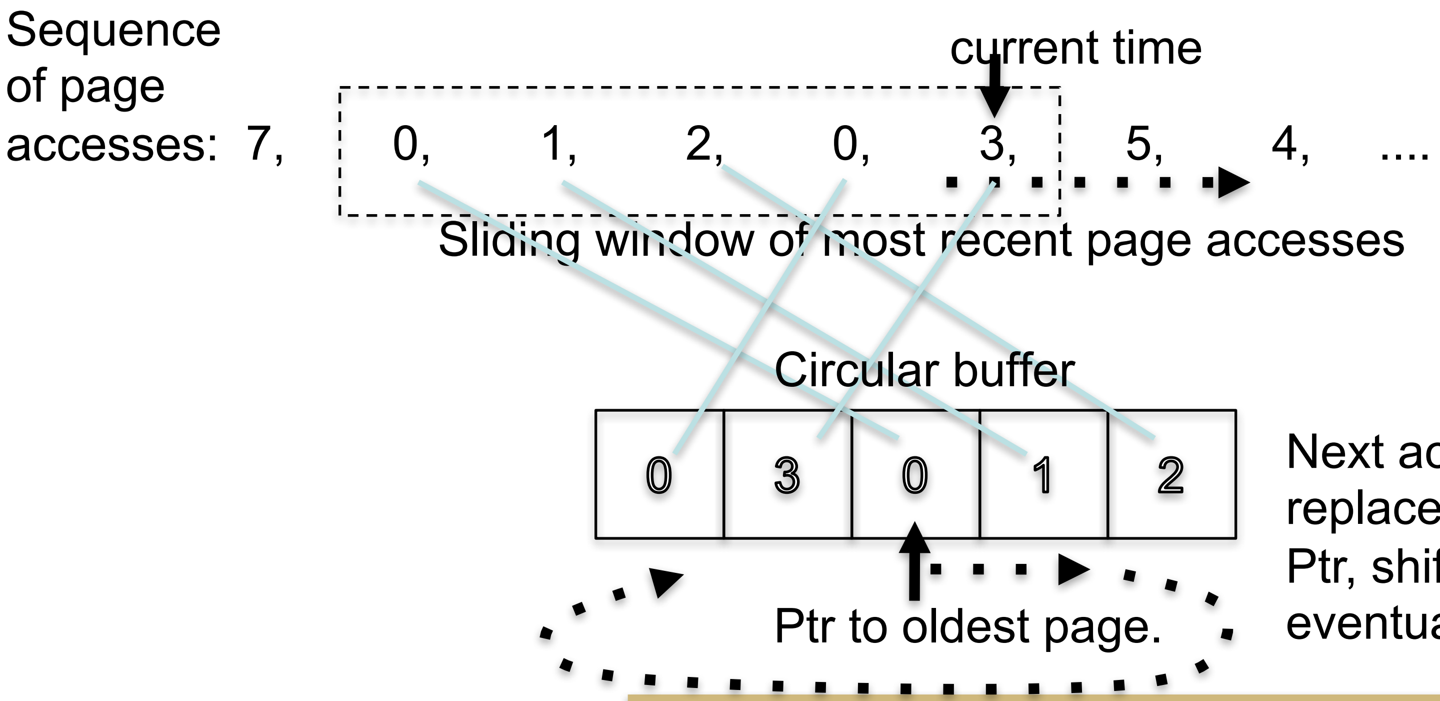


7, 0, 1, 2, 0, 3, 0, 4, ...
LRU queue
7
7 0
7 0 1
2 0 1
2 0 3
4 0 3
page 4 replaces page 2, which was LRU
page 2 replaces page 7 because page 7 was the least recently used
page 3 replaces page 1 because page 1 was least recently used

In worst case, LRU page could be the next page, but that's unusual

LRU Implementation Options

- History
  - Keep a history of past page accesses, either the entire history (lots of memory) or a sliding window
    - Sliding window could be implemented as a circular buffer with a moving pointer:



Sequence of page accesses: 7, 0, 1, 2, 0, 3, 5, 4, ...
current time
Sliding window of most recent page accesses
Circular buffer
0
3
0
1
2
Ptr to oldest page.

Next access to page 5 would replace entry 0 pointed to by Ptr, shifting Ptr right by one, eventually wrapping around

LRU Implementation Options

- History (cont.)
  - Problems of sliding window:
    - how big a window? Hard to determine…
    - Need a new data structure outside of page table
    - Replacing oldest page P indicated by the pointer is not necessarily the least recently used, because P may be accessed more recently.
      - e.g. in our example on the prior slide, page 0 is the oldest, yet was also used more recently
      - To find the LRU page in the window, have to search from current position of pointer back in history through the circular buffer
        - » build an ordered list and remember to eliminate older duplicates in that list. This is complicated and slow.

LRU Implementation Options

- Timers
  - keep an actual time stamp for each page as to when it was last used
  - Problems:
    - Expensive in delay (consult system clock on each page reference)
    - Expensive in storage (at least 64 bits per absolute time stamp)
    - Expensive in search (find the page with the oldest time stamp, so either search or keep min, which requires an if-then-else comparison to may reset the min on each page reference)

LRU Implementation Options

- Counters
  - Approximate time stamp in the form of a counter that is incremented with any page reference, i.e. each page's counter must be incremented on each page reference
  - Is stored with that entry in the page table. Counter is reset to 0 on a reference.
  - Problem: expensive in update (each page's counter must be incremented on each page reference) and in search

LRU Implementation Options

- Linked List
  - whenever a page is referenced, put it on the end of the linked list, removing if it from within the linked list if already present in list
  - Front of linked list is LRU
  - Problem: managing a (doubly) linked list and rearranging pointers becomes expensive
- Similar problems with a Stack

Reference-bit based LRU approximation algorithms

- add an extra HW bit called a *reference bit*
  - this is set any time a page is referenced (read or write)
  - allows OS to see what pages have been used, though not fine-grained detail on the order of use
  - Reference-bit based algorithms only *approximate* LRU, i.e. they do not seek to exactly implement LRU
  - 3 types of reference-bit LRU approximation algorithms:
    - Additional Reference-Bits Algorithm
    - Second-Chance (Clock) Algorithm
    - Enhanced Clock Algorithm with Dirty/Modify Bit

Reference-bit based LRU approximation algorithms

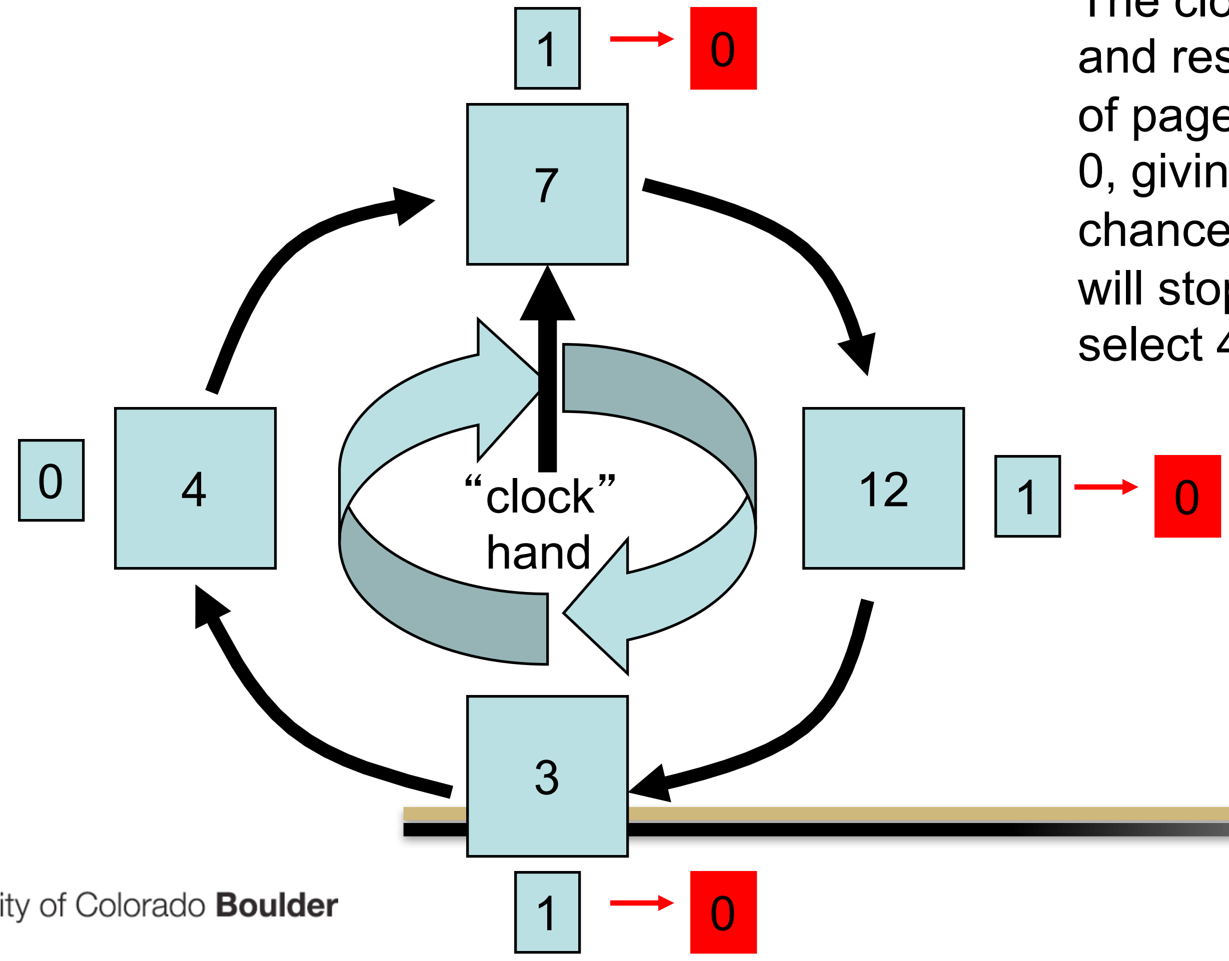
- Additional Reference-Bits Algorithm
  - record the last 8 reference bits for each page
  - periodically a timer interrupt shifts the reference bit into the MSB on a record
  - example: 11000100 has been used more recently than 01110111
  - So LRU = lowest valued record

Reference-bit based LRU approximation algorithms

- Second-Chance (Clock) Algorithm
  - Goal is to find the page that has been least recently referenced using just one reference bit
  - In-memory pages + reference bits conceptually form a *circular* queue
  - Setup:
    - Arrange all N allocated pages in a circular buffer, i.e. circularly as if they're at the edges of a clock. Create a rotating pointer that is the clock hand.

Reference-bit based LRU approximation algorithms

- Clock algorithm example:



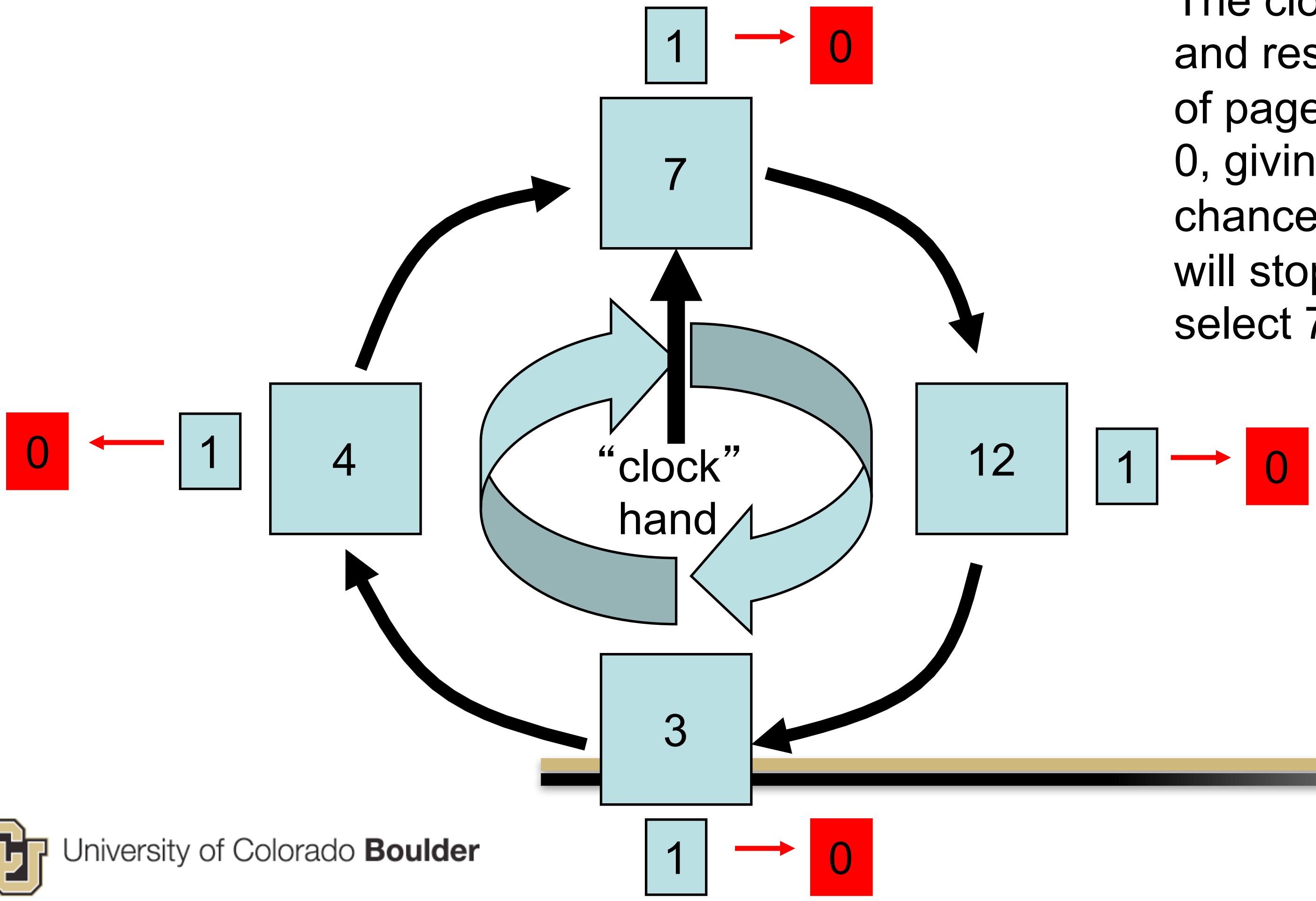
The clock hand will rotate and reset the reference bits of pages 7, 12, and 3 to 0, giving them all 2nd chances. Then the hand will stop at page 4, and select 4 for replacement.

Reference-bit based LRU approximation algorithms

- Second-Chance (Clock) Algorithm
  - Policy: when selecting a victim, rotate a current pointer or clock hand through the circular queue
    - if current frame's reference bit = 0, replace this page
      - this page was not read or written recently (since the last rotation of the clock hand through the circular queue), so replace this page
    - if current frame's reference bit = 1, give the page a second chance and just clear its bit to 0
      - this page was referenced recently (since the last rotation of clock hand) so keep it around in memory
    - As the clock hand advances, it resets referenced frames to 0, giving them a 2nd chance, and stops at the first unreferenced frame

Reference-bit based LRU approximation algorithms

- Clock algorithm example:



The clock hand will rotate and reset the reference bits of pages 7, 12, 3 & 1 to 0, giving them all 2nd chances. Then the hand will stop at page 7, and select 7 for replacement.

Reference-bit based LRU approximation algorithms

- How does this approximate LRU?
  - Example:
    - clock hand starts at 12 o'clock.
    - If first four page accesses are 7,0,2, and 4, then 7 goes to 12 o'clock with ref bit = 1, 0 goes to 3 o'clock with ref bit = 1, 2 goes to 6 o'clock with ref bit = 1, and 4 goes to 9 o'clock with ref bit = 1.
    - If next page is 5, find a page to replace. Clock algorithm starts at 12 o'clock, resets 7's ref bit to 0, moves to 3 o'clock, resets 0's ref bit to 0, keeps rotating around resetting ref bits to 0 until it hits 7 at 12 o'clock. Since 7's ref bit = 0, then it is replaced. Note 7 is the Least recently used page, because it was the first to arrive.

Reference-bit based LRU approximation algorithms

- How does this approximate LRU?
  - More generally, given any position of a clock hand, any pages that have been (recently) used since the clock hand last rotated through will have ref bit = 1. Any other pages will be 0. These are the least recently used page candidates.

Reference-bit based LRU approximation algorithms

- How does this approximate LRU?
  - Of these ref bit 0 candidates, the one closest clockwise to the clock hand will have been in 0 state the longest, i.e. unreferenced the longest – this is the LRU candidate
    - Pages only go to unreferenced state after the clock hand has swept through and changed them from 1->0.
    - Therefore, the 0's closest counter-clockwise to the clock hand were generated most recently, but the 0's clockwise were generated most remotely in time

Reference-bit based LRU approximation algorithms

- Note this algorithm is only an approximation, and in some corner cases does not select the LRU
  - Example: pages 7, 0, 2, and 4 are in memory and used again and again in any order.
    - Let's say clock hand points to 7, and the most recent page accesses are 4, 2, 0, 7 and all occurred since the last clock hand rotation.
    - If the next page is 5, then in choosing a page to replace, the clock hand will circle around, reset all ref bits to 1, then select 7 to evict, even though 7 is the most recently used.
    - Normally, will have some unreferenced pages, so these are closer to LRU pages.

Reference-bit based LRU approximation algorithms

- Second-Chance (Clock) Algorithm
  - Advantages: simple to implement (one pointer for the clock hand + 1 ref bit/page.
    - Note the circular buffer is actually just the page table with entries where the valid bit is set, so no new circular queue has to be constructed)
    - fast to check reference bit and usually fast to find first page with a 0 reference bit, and approximates LRU
  - Disadvantages: in the worst case, have to rotate through the entire circular buffer once before finding the first victim frame

Reference-bit based LRU approximation algorithms

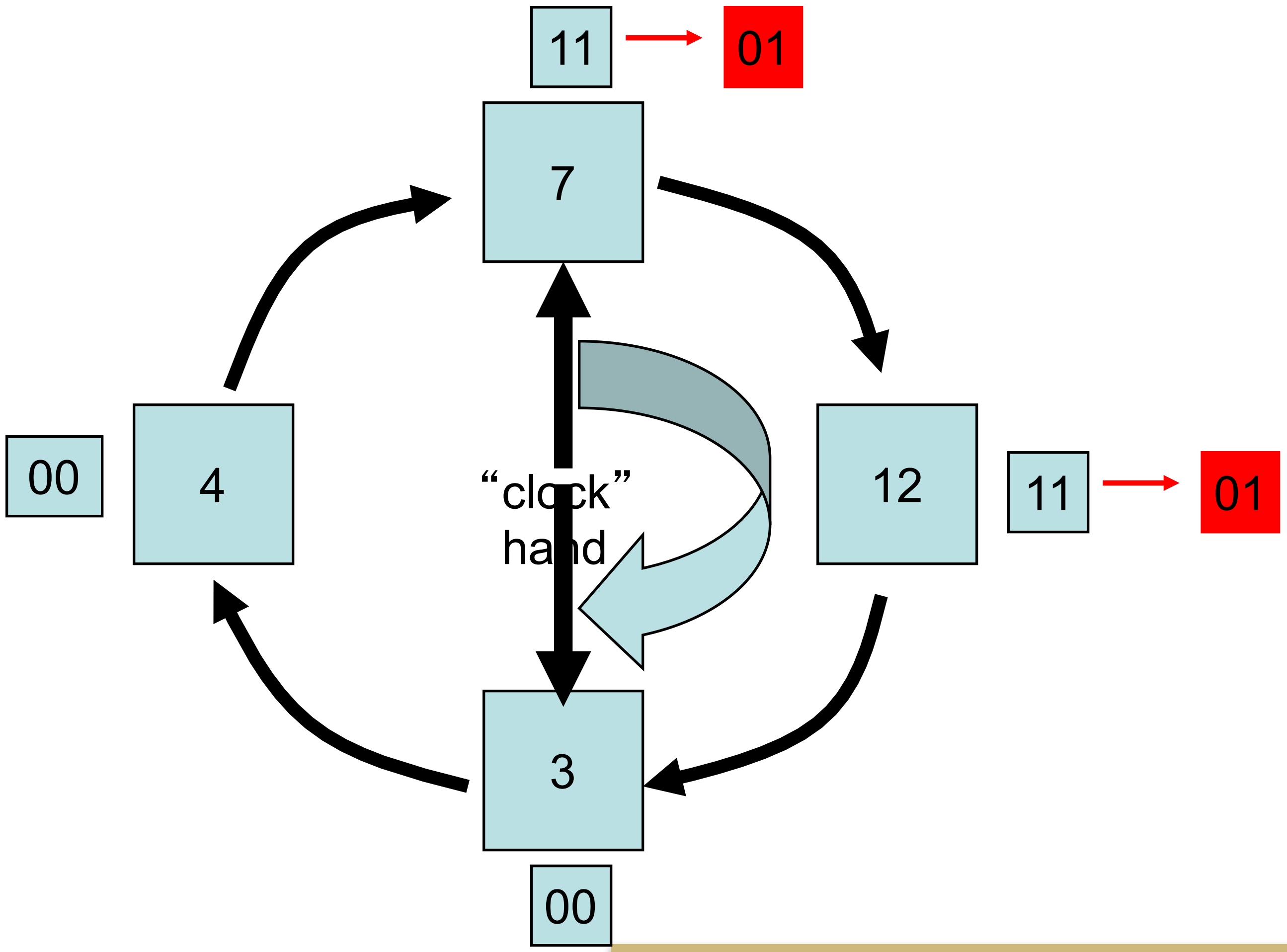
- Enhanced Second-Chance (Clock) Algorithm
  - Add a dirty/modify bit to the reference bit and consider them as a pair
  - when selecting a victim, rotate a current pointer or clock hand through the queue as in the clock algorithm, and replace the first page encountered in the lowest nonempty class
    - if current frame's reference bit = 0, and modify bit =0, then this page has been neither recently used nor modified – best page to replace
      - Select this page, you're done.

Reference-bit based LRU approximation algorithms

- Enhanced Second-Chance (Clock) Algorithm (continued)
    - if current frame = (0, 1), page is not recently used, but was modified – not quite as good because replacement requires an extra write
        - Remember the first (0 1) page that is encountered
    - If current frame = (1, 0), page is recently used and clean – probably will be used again soon, reset reference bit to 0
    - If current frame = (1, 1), page is recently used and not clean – probably used again soon, and would require a disk write, reset reference bit to 0
  - in the worst case, have to repeat the rotation through the entire circular buffer multiple times before finding the first victim frame

Reference-bit based LRU approximation algorithms

- Enhanced Clock algorithm example 1:



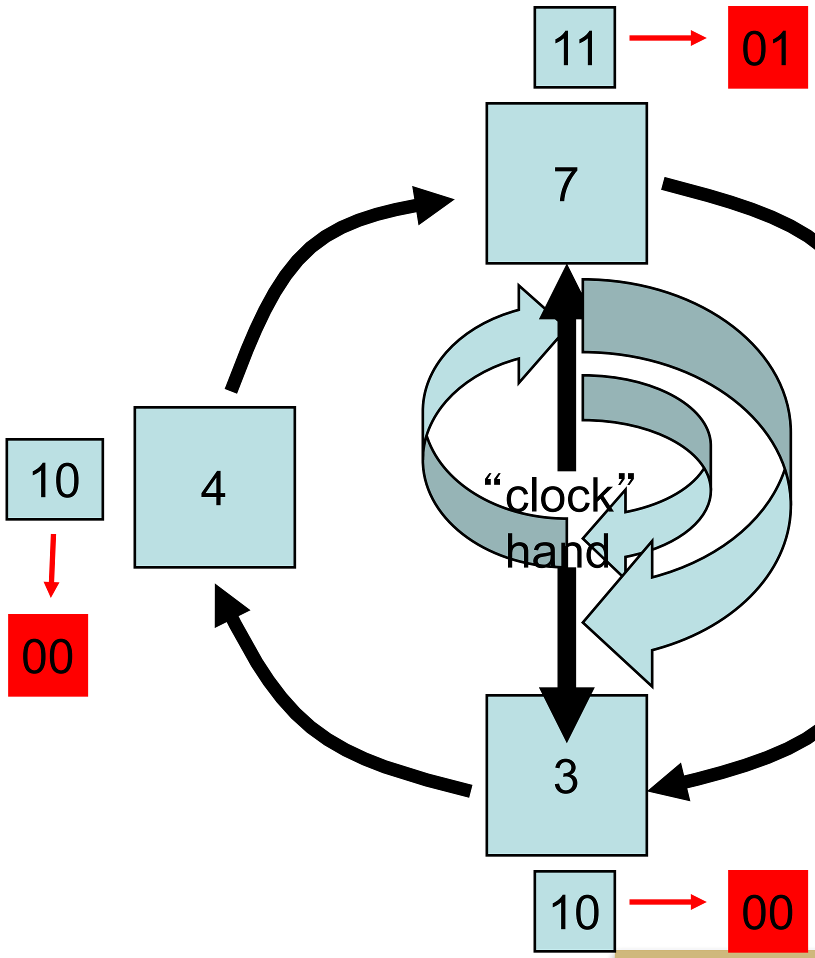
In this example, start with clock hand pointing at 12 o'clock at page 7. The 1st two pages are currently at (11).

As the clock hand rotates clockwise through, it gives the 1st two pages a second chance:
- (11) ➔ (01)

Eventually, we find the first (00) at six o'clock, page 3, and replace it.

Reference-bit based LRU approximation algorithms

- Enhanced Clock algorithm example 2:



Again let’s say clock hand is at 12 o’clock. As the clock hand rotates, all pages are given a second chance:

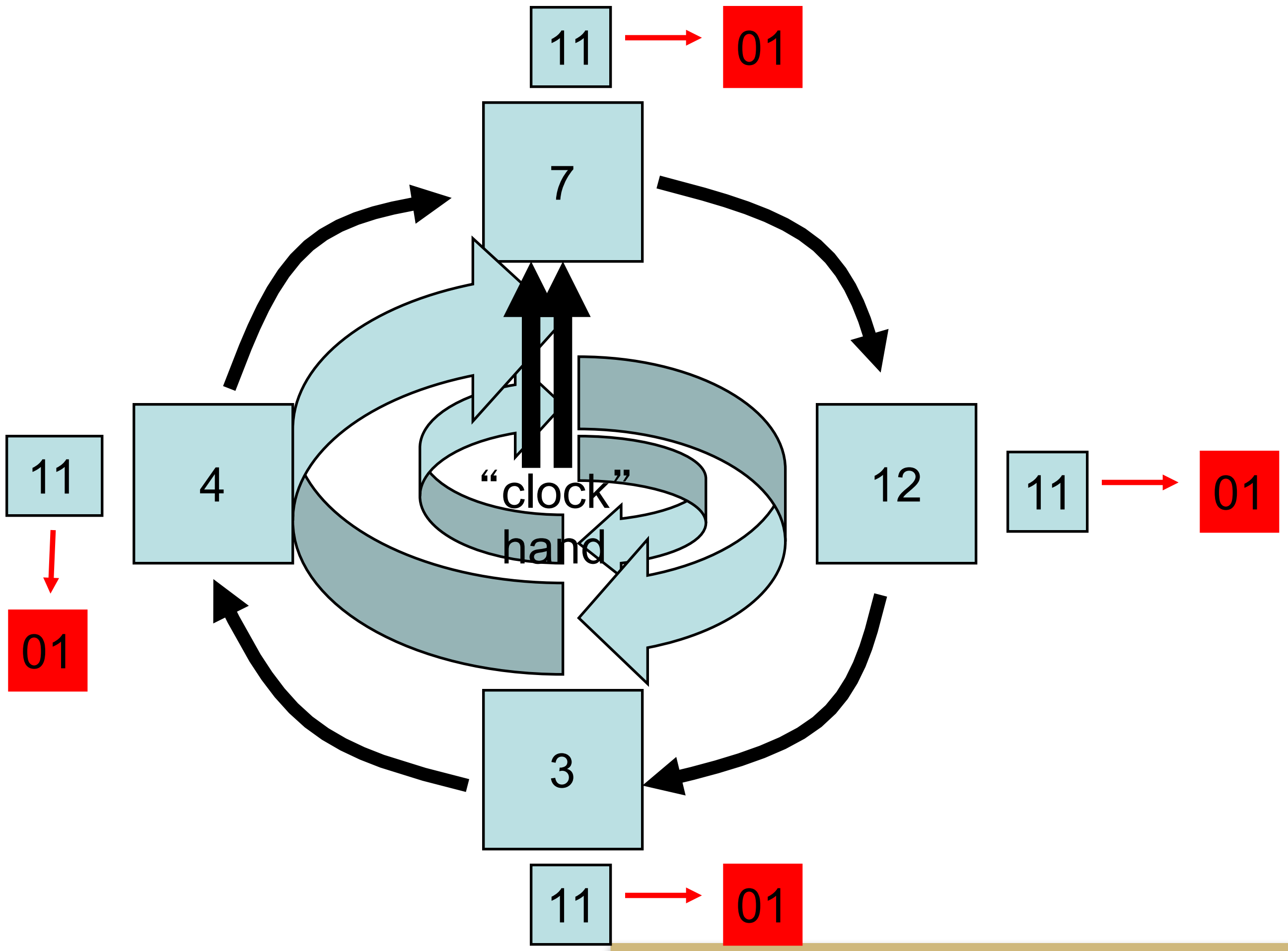
- (11) ➔ (01)
- (10) ➔ (00)

In the 2nd pass, the clock hand keeps rotating and finds page 3 at 6 o’clock with the value (0 0).

The clock hand has in this example rotated 1 ½ times through the circular buffer.

Reference-bit based LRU approximation algorithms

- Worst case Enhanced Clock algorithm example:



Clock hand points at 12 o'clock. All pages are (11).
In the 1st pass, the clock hand rotates through and resets all (11) ➔ (01).
In the 2nd pass, the clock hand remembers the first (0 1) page is page 7. It rotates all the way around, because there are no (0 0) pages.
When the hand returns to page 7 at 12 o'clock, the algorithm selects the 1st (0 1) page, i.e. page 7.
Worst case occurs with two full rotations to search for LRU!