Announcements

- Midterm Exam
  - Thursday, October 27 in class
  - 1 hr 15 min, closed book, closed notes
  - Chapters 1 – 6 and 13; Lecture Sets one – eleven
  - Similar to the sample midterm exam available on Moodle
- Solutions for problem set 1 and 2 and sample midterm exam are posted on Moodle

Recap…

- Paging
  - No external fragmentation; some internal fragmentation
  - No need for contiguous memory to store a process
- Efficient address translation is crucial
  - Choose a page size that is a power of 2
  - Hardware support
    - Page Table Base Register (PTBR)
    - Translation Lookaside Buffers (TLBs)
- Page table storage: contiguous memory and size
  - Hierarchical page tables
  - Inverted page tables

- Problem: A process must be fully loaded in memory to run

CSCI 3753
Operating Systems

Memory Management
Virtual Memory

Chapters 8 and 9

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder

Last Update: 10/20/16

On-Demand Paging

- So far, we have assumed that a running process must be completely loaded in memory
- Key observation: Not all pages in a logical address space need to be kept in memory
  - In the simplest case, just keep the current page where the process is executing
  - All other pages could be on disk
  - When another page is needed, retrieve the page from disk and place in memory before executing
    - This would be costly and slow, because it would happen every time that a page different from the current one is needed

On-Demand Paging

- Instead of keeping just one page, keep a subset of a process’s pages in memory
- Key questions: Which pages to keep in memory?
  - Pages that will be referenced in near future
  - Need to know what pages will be referenced in near future
- Rely on program’s behavior

Principle of *Locality of reference*

- If an instruction or data in a page was recently referenced, then that page is likely to be referenced again in the near future
- Most programs exhibit some form of locality
  - Looping locally through the same set of instructions
  - Branching through the same code
  - Executing linearly, the next instruction is typically the next one immediately following the previous instruction, rather than some random jump

On-Demand Paging

- On-demand paging: page in new pages from disk to RAM only when a page is referenced
  - Can page in an entire process on demand, starting with “zero” pages - the reference to the first instruction causes the first page of the process to be loaded on demand into RAM. Subsequent pages are loaded on demand into RAM as the process executes

Demand Paging

- Questions
  - What happens when a referenced page is not loaded in memory
    - Page fault
  - How many page frames in memory should be allocated to a process?
    - Working set theory
  - If the # of page frames allocated to a process is exceeded, how do you choose which page to replace?
    - Page replacement algorithm

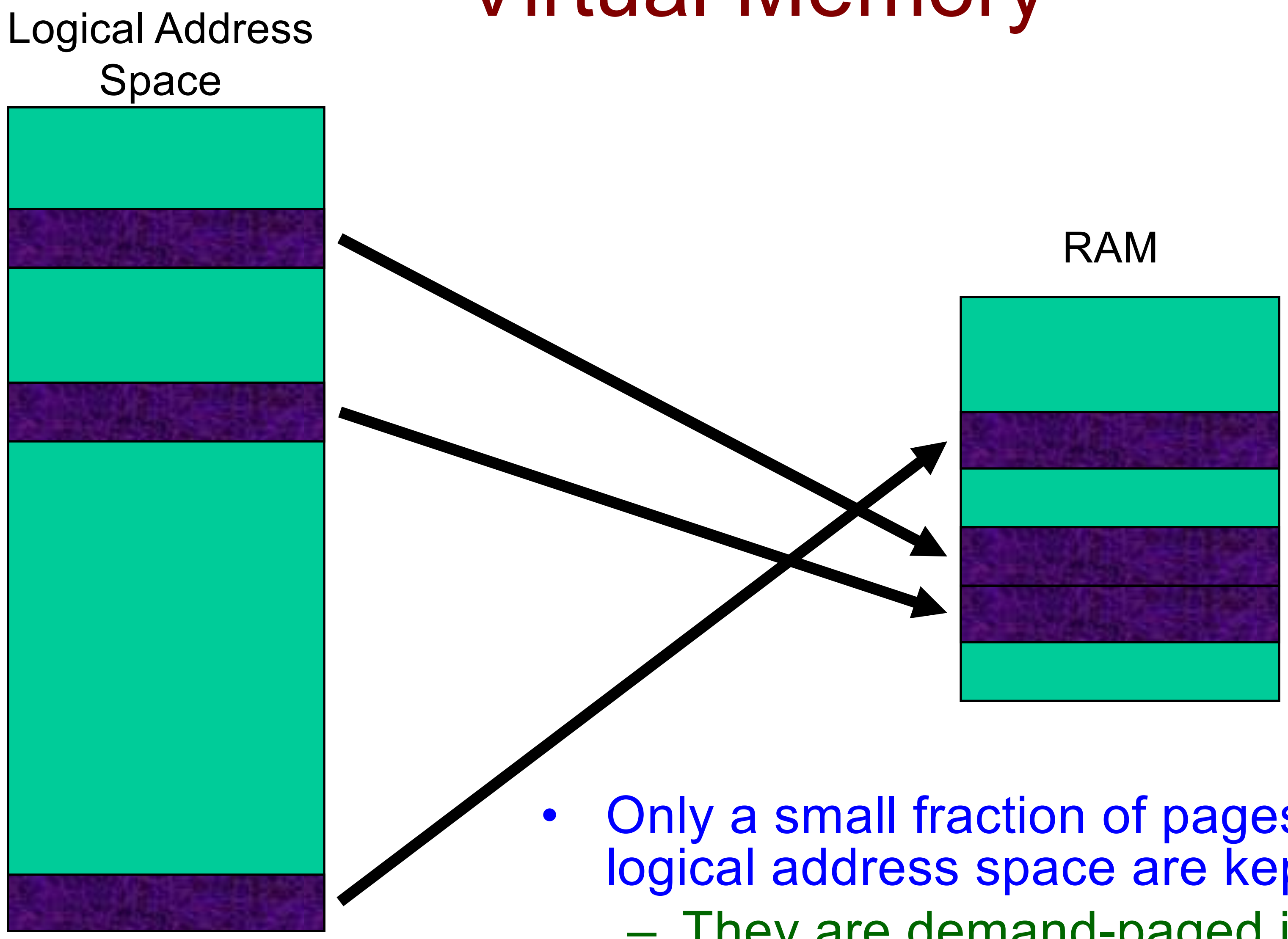
Virtual Memory

- Memory that appears to exist as main storage although most of it is supported by data held in secondary storage and transfer between the two is done automatically as needed.
- Advantages
  - Can fit many more processes in memory!
  - Decreases swap time – there is less to swap
  - The logical address space of a process (*virtual address space*) can now exceed physical RAM!

Virtual Memory

- Thus, we have now truly decoupled a process' logical address space from physical memory allotted to it
- Can have large sparse address spaces, in which most of the address space is unused, without taking up lots of physical RAM
- Example, create a virtual address space that has a large heap and stack. These remain mostly unused/empty and don't take up much actual RAM until needed, i.e. when the stack or heap grows very large

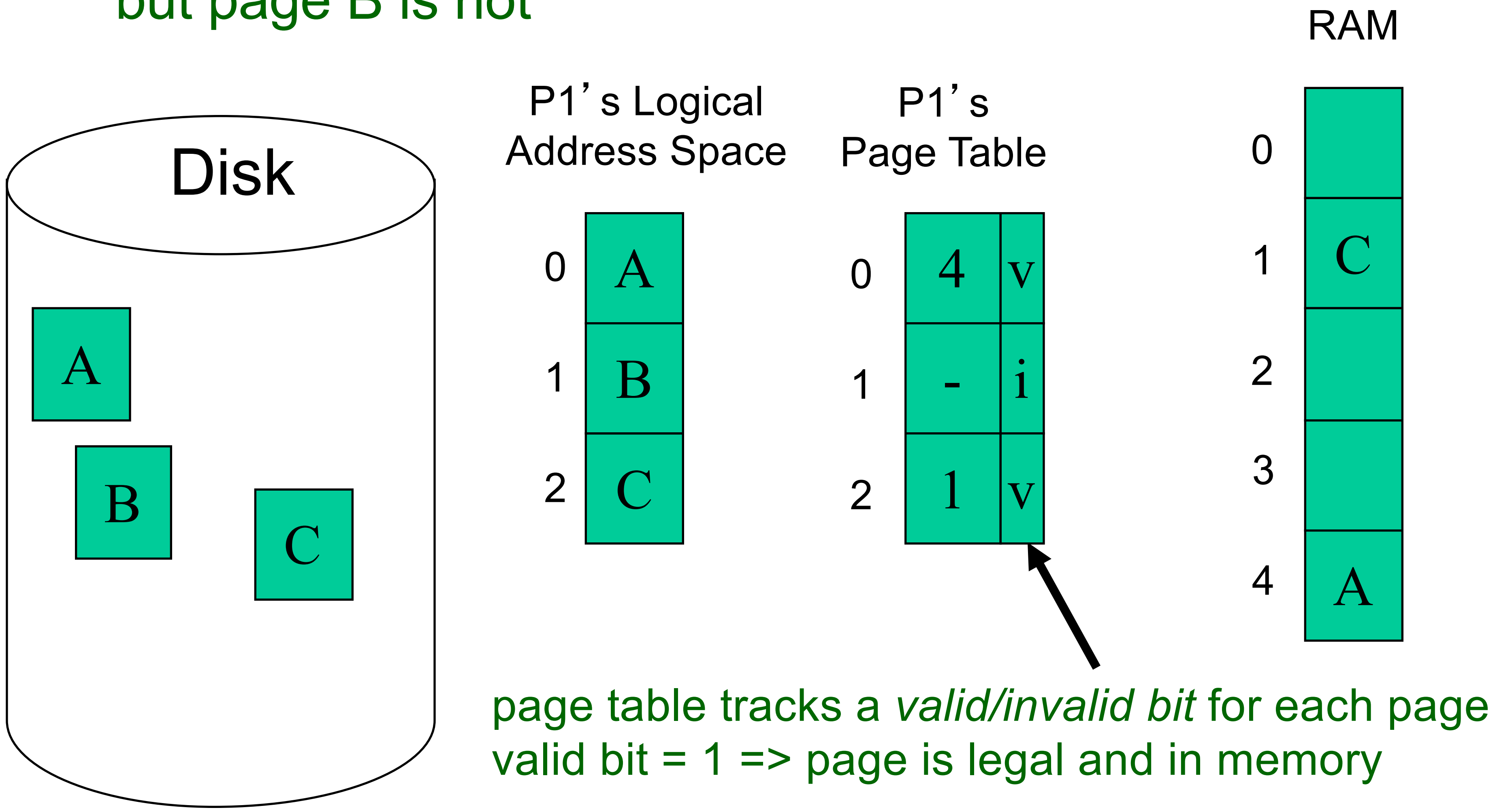
Virtual Memory



- Only a small fraction of pages from the logical address space are kept in RAM
  - They are demand-paged into RAM
- The logical address space can be much larger than physical RAM, hence the logical addresses form a *virtual memory*

Virtual Memory

- On-demand paging loads a page from disk into RAM only when needed
  - In the example below, pages A and C are in memory, but page B is not



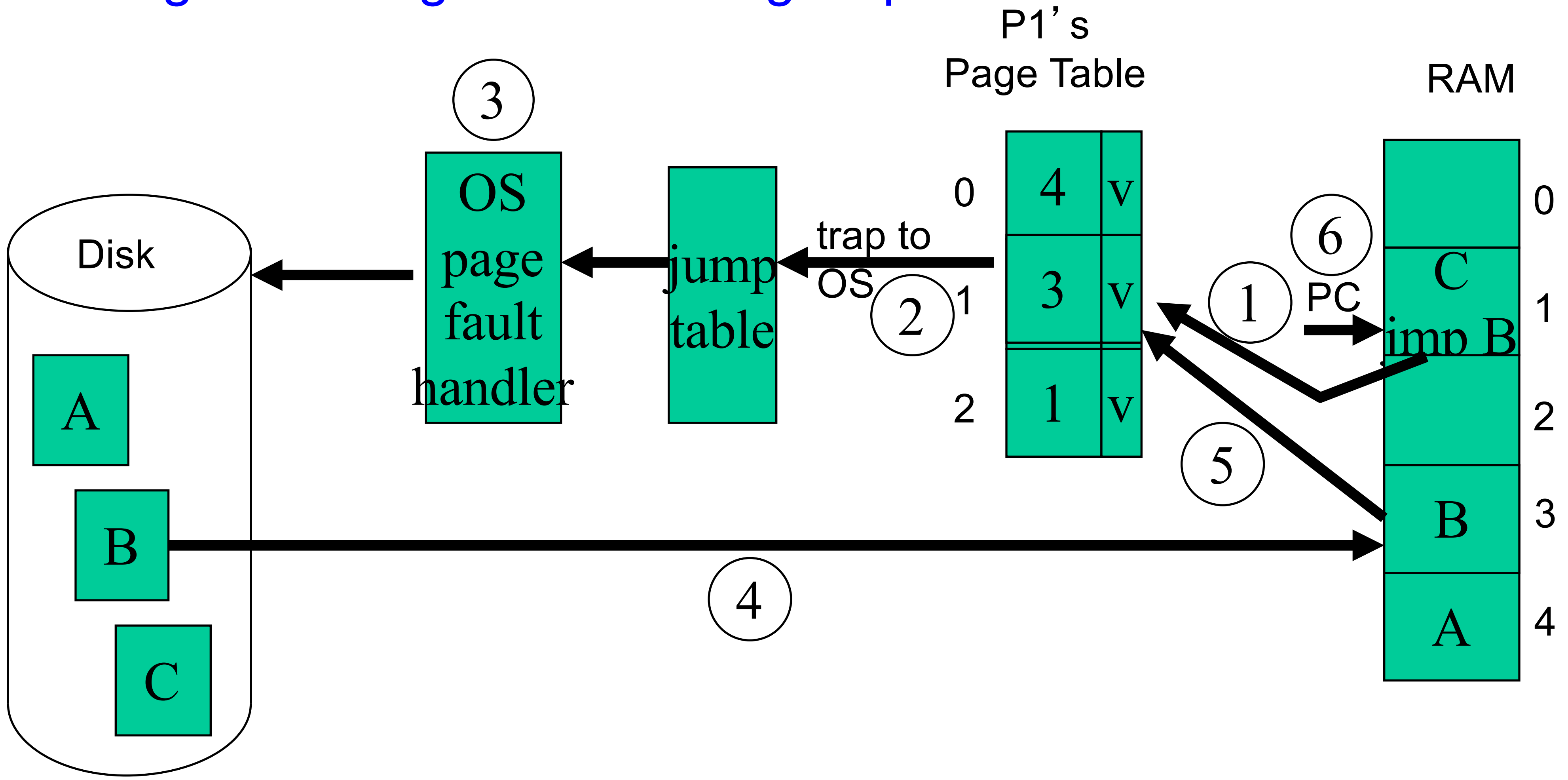
page table tracks a *valid/invalid bit* for each page
valid bit = 1 => page is legal and in memory

Virtual Memory: Page fault

1. MMU detects a page is not in memory (invalid bit set) which causes a *page-fault trap* to OS
2. OS saves registers and process state. Jumps to page fault handler
3. Page fault handler
   a) If referenced page not in logical A.S. → seg fault.
   b) Else load page
   c) OS finds a free frame
   d) OS schedules a disk read. Other processes may run in meantime
4. Disk returns with interrupt when done reading desired page. OS writes page into free frame
5. OS updates page table, sets valid bit of page and its physical location (page frame number)
6. Restart interrupted instruction that caused the page fault

Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



Virtual Memory

- OS can retrieve the desired page either from the file system, or
- from the swap space/backing store on disk
  - faster, avoids overhead of file system lookup
  - pages can be in swap space either because
    - the entire executable file was copied into swap space when the program was first started. Not only does this avoid the file system, but also allows the copied executable to be laid out contiguously on disk's swap space, for faster access to pages (no seek time)
    - as pages have to be replaced in RAM, they are written to swap space instead of the file system's portion of disk. The next time they're needed, they're retrieved quickly from swap space, avoiding a file system lookup.

Virtual Memory: Performance

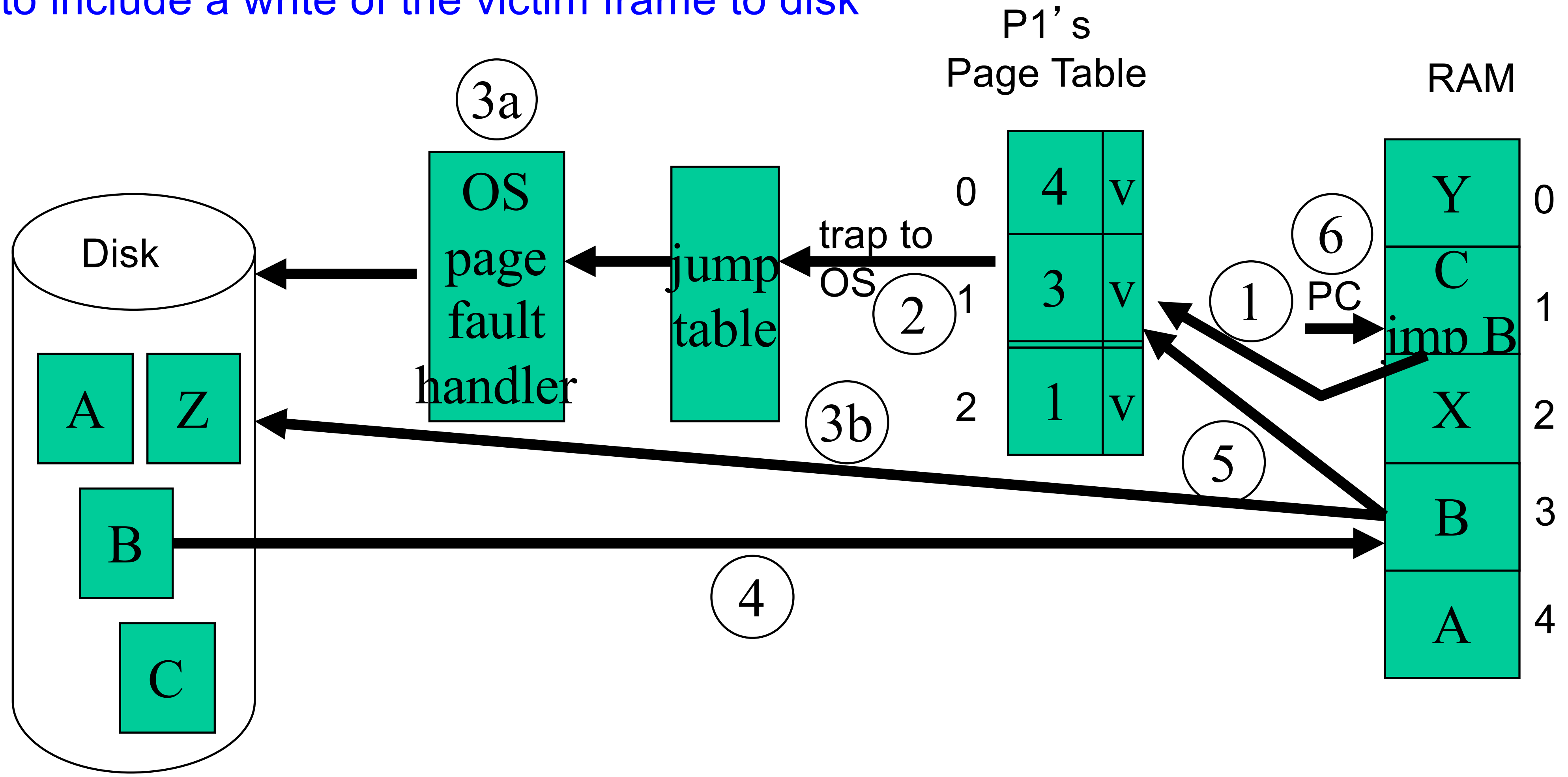
- Want to limit the number/frequency of page faults, which cause a read from disk, which slows performance
  - disk read is about 10 ms
  - memory read is about 10 ns
- What is the average memory access time?
  - average access time = p*10 ms + (1-p)*10 ns
  - p = probability of a page fault
  - if p=.001, then average access time = 10 μs >> 10 ns (1000 X greater!)
  - to keep average access time within 10% of 10 ns, would need a page fault rate lower than $p<10^{-7}$
- Thus, any extra bit of effort to reduce the page fault frequency can result in big performance gains

Page Replacement Policies

- As processes execute and bring in more pages on demand into memory, eventually the system runs out of free frames
- Need a *page replacement policy*
  1. Select a victim frame that is not currently being used
  2. Save or write the victim frame to disk, update the page table (page now invalid)
  3. Load in the new desired page from disk
- If out of free frames then each page fault causes 2 disk operations, one to write the victim, and one to read the desired page - this is a big performance penalty

Page Replacement Policies

In Step 3b, we modify traditional on-demand paging to include a write of the victim frame to disk

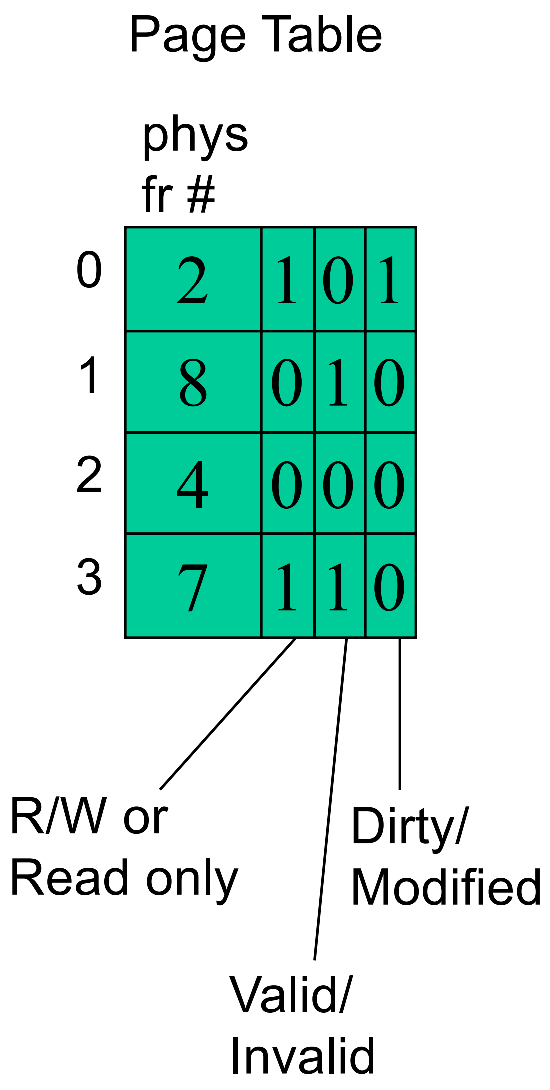


Page Replacement Policies

- To reduce the performance penalty of 2 disk operations, systems can employ a *dirty/modify bit*
  - modify bit = 0 initially
  - when a page in memory is written to, set the bit = 1
  - when a victim page is needed, select a page that has not been modified (dirty bit = 0)
    - such an unmodified page need not be written to disk, because its mirror image is already on disk!
    - this saves on disk I/O - reduces to only 1 disk operation (read of desired page)

Page Table Status Bits

- Each entry in the page table can actually store several extra bits of information besides the physical frame # f
  - *R/W or Read-only bits* - for memory protection, writing to a read-only page causes a fault and a trap to the OS
    - Make code pages read-only to protect them from being written to
  - *Valid/invalid bits* - for memory protection, accessing an invalid page causes a page fault
    - is the page in memory or not?
  - *dirty bits* - has the page been modified for page replacement?



Page Replacement Policies

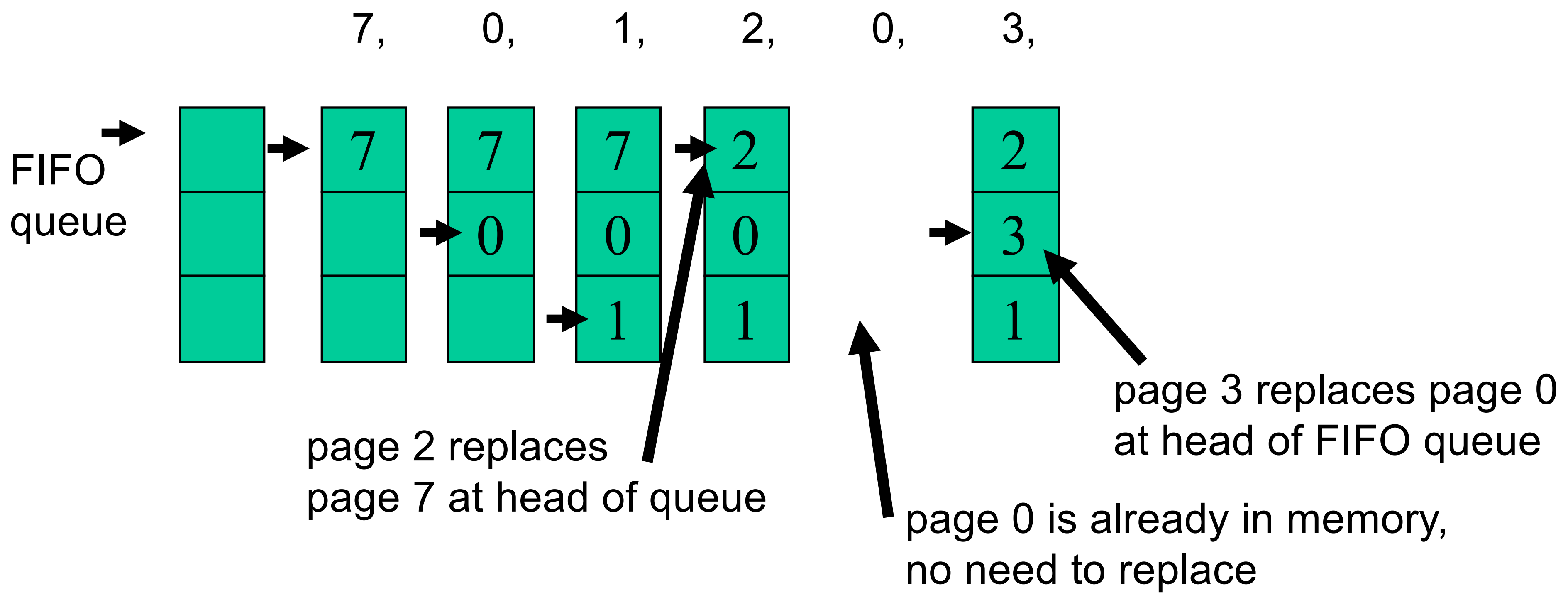
- Which page does OS choose to replace?
  - Assume the policy is to choose to replace a clean page before a dirty page, to save on a disk write
  - Even so, there may still be multiple clean pages (dirty bit = 0), which forces me to select one among these. If I choose the wrong one, which is referenced again soon, will have to page it back in – performance hit
  - If all pages are dirty, I 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

- 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

FIFO Page Replacement

- FIFO is easy to understand and implement
- Performance can be poor
  - Suppose page 7 that was replaced was a very active page that was frequently referenced, then page 7 will be referenced again very soon, causing a page fault because it's not in memory any more
  - 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 basis of the OPT algorithm in the next slide

FIFO Page Replacement: Another example

Let page reference stream, $\mathcal{R}$= `012301401234`

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	$\underline{0}$	0	0	$\underline{3}$	3	3	$\underline{4}$	4	4	4	4	4
1		$\underline{1}$	1	1	$\underline{0}$	0	0	0	0	$\underline{2}$	2	2
2			$\underline{2}$	2	2	$\underline{1}$	1	1	1	1	$\underline{3}$	3

- FIFO with m = 3 has 9 faults
- Goal: To reduce the number of page fault
  - Increase the size of memory

FIFO Page Replacement: Another example

Let page reference stream, $\mathcal{R}$ = 012301401234

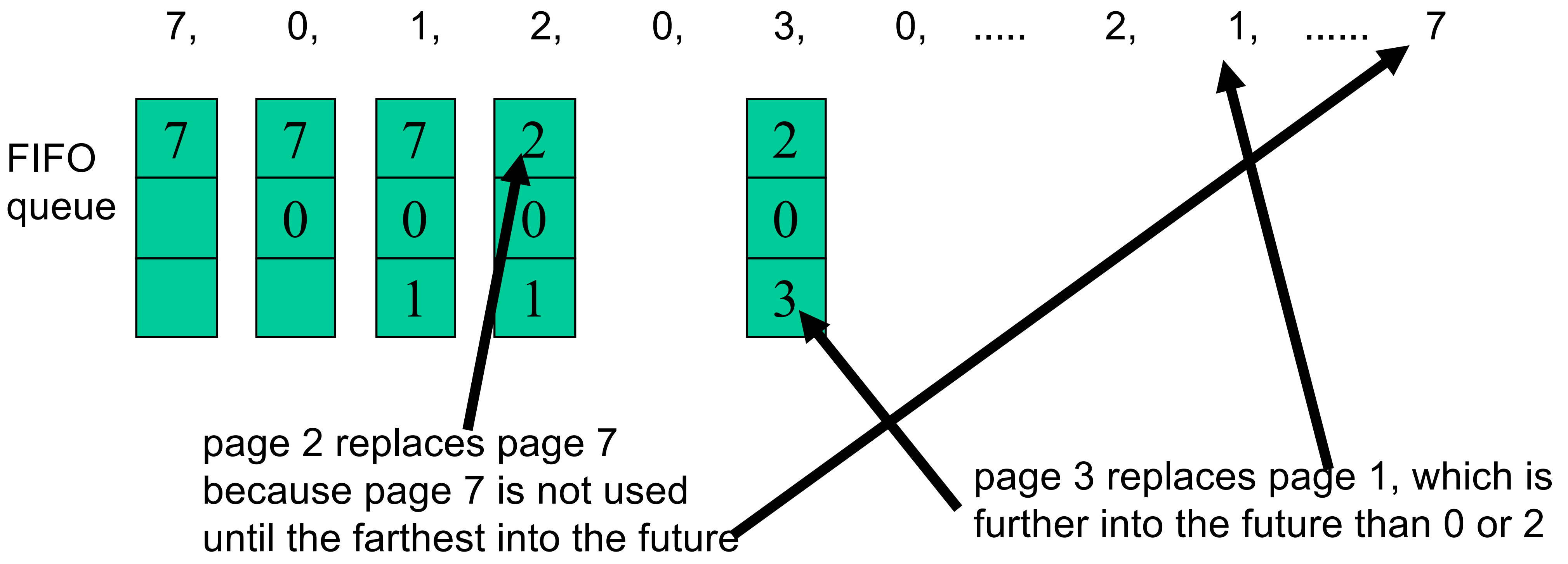
Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	$\underline{0}$	0	0	0	0	0	$\underline{4}$	4	4	4	$\underline{3}$	3
1		$\underline{1}$	1	1	1	1	1	$\underline{0}$	0	0	0	$\underline{4}$
2			$\underline{2}$	2	2	2	2	2	$\underline{1}$	1	1	1
3				$\underline{3}$	3	3	3	3	3	$\underline{2}$	2	2

- FIFO with m = 4 has 10 faults

Belady’s anomaly: Increasing the size of memory may result in increasing the number of page faults for some programs.

OPT Page Replacement

- OPT = Optimal
  - Replace the page that will not be referenced for the longest time
  - Guarantees the lowest page-fault rate
  - Problem: requires future knowledge

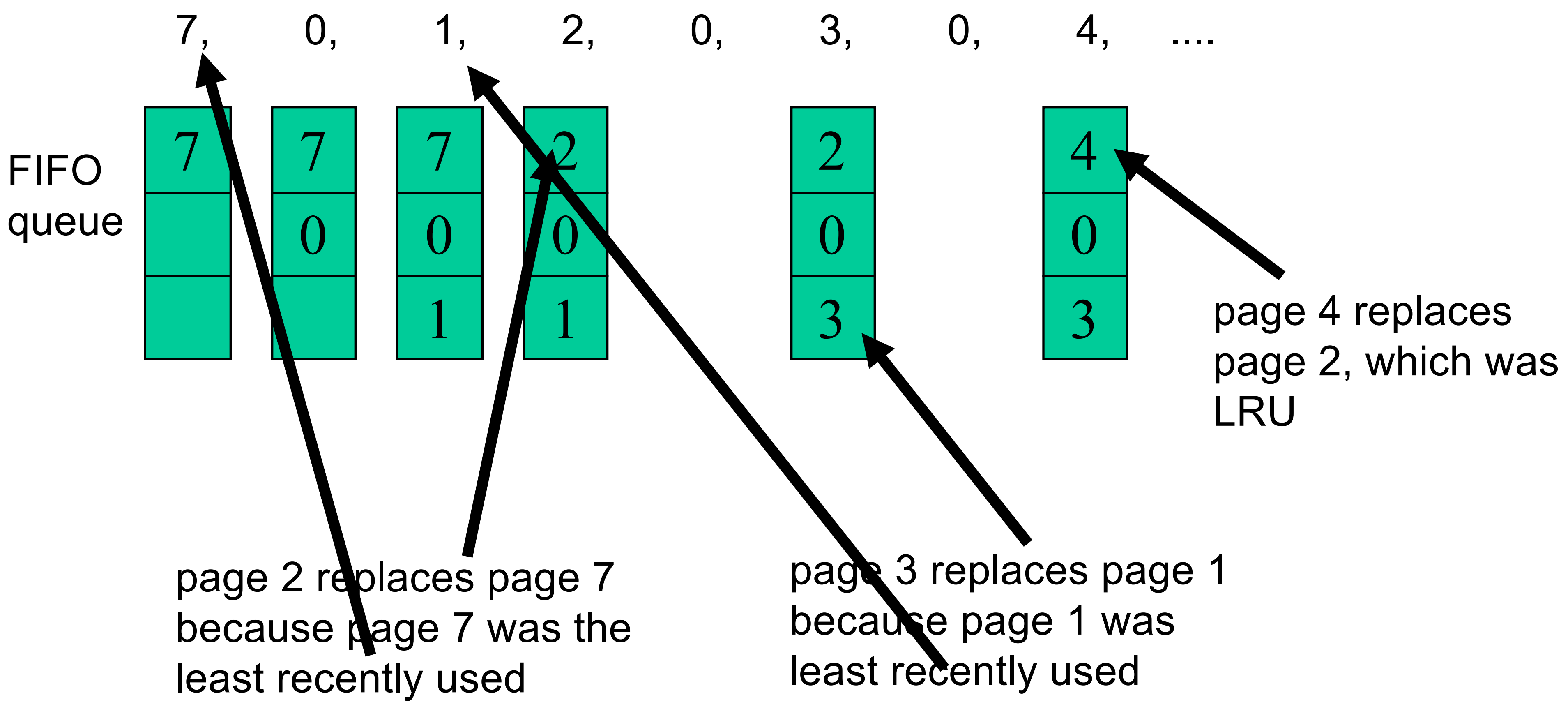

7, 0, 1, 2, 0, 3, 0, ... 2, 1, ... 7
FIFO queue
7
7 0
7 0 1
2 0 1
2 0 3
page 2 replaces page 7 because page 7 is not used until the farthest into the future
page 3 replaces page 1, which is further into the future than 0 or 2

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 Page Replacement

- LRU example


7, 0, 1, 2, 0, 3, 0, 4, ...
FIFO 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

Stack Algorithms

- Stack algorithms are a class of page replacement algorithms that do not suffer from Belady's anomaly
- Key property: Set of pages in memory for *n* frames is always a subset of the set of pages that would be in memory with *n+1* frames, irrespective of the page reference string
- OPT and LRU are stack algorithms, while FIFO is not a stack algorithm
  - See textbook for explanation (Page 417)

LRU Implementation Options

- History: Keep a history of past page accesses, either the entire history (lots of memory) or a sliding window
  - Complicated and slow

LRU Implementation Options

- Timers
  - keep an actual time stamp for each page as to when it was last used
  - Problem: expensive in delay (consult system clock on each page reference), storage (at least 64 bits per absolute time stamp), and search (find the page with the oldest time stamp)

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 right the bits in the record and puts the reference bit into the MSB
  - Example: 11000100 has been used more recently than 01110111
  - So LRU = lowest valued record

Reference-bit based LRU approximation algorithms

- Second-Chance (Clock) Algorithm
  - In-memory pages + reference bits conceptually form a *circular* queue; a hand points to a page.
  - If page pointed to has R = 0, it is replaced and the hand moves forward.
  - Otherwise, set R = 0; hand moves forward and checks the next page.

- 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
  - Reference bit is cleared periodically
  - 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
  - Four classes are formed
    - Class 0: R = 0; M = 0. (Least heavily used class)
    - Class 1: R = 0; M = 1.
    - Class 2: R = 1; M = 0.
    - Class 3: R = 1; M = 1. (Most heavily used class)

Non-LRU Counting-Based Page Replacement

- Keep a counter of number of page accesses for each page since its introduction, which is an activity or popularity index
  - Compare to LRU counters, where every page's counter is incremented at each page access. Here, only the accessed page's counter is incremented.
- Least Frequently Used
  - Replace page with lowest count
  - What if a page was heavily used in the beginning, but not recently? Age the count by shifting its value right by 1 bit periodically - this is exponential decay of the count.
- These kinds of policies are not so popular, because their implementation is expensive, and they aren't necessarily close to OPT