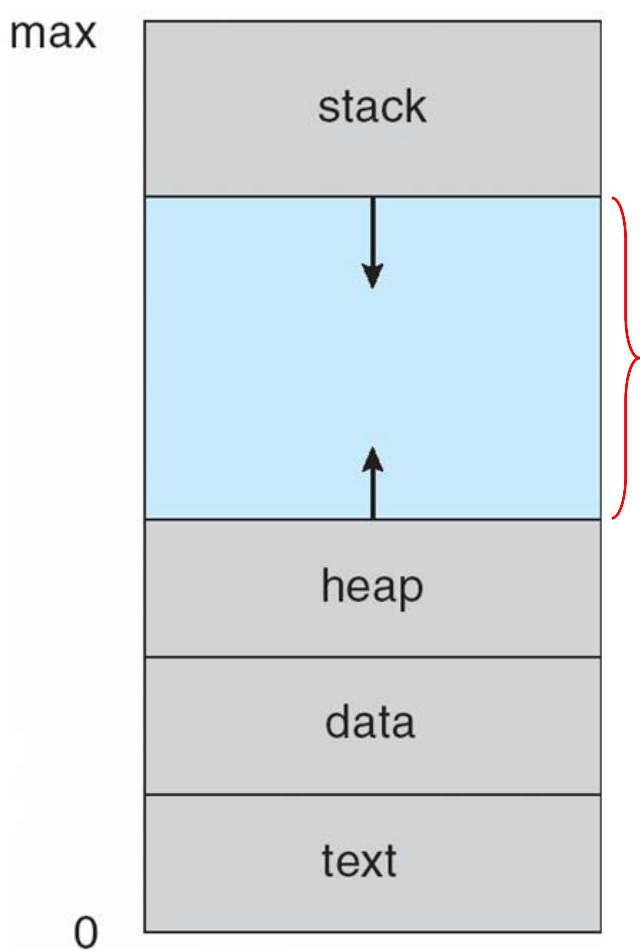
Virtual Memory(1)

Dept. of Computer Science

Hanyang University

Size of a Logical Address Space



Using 32bits addresses, max = $2^{32} - 1$

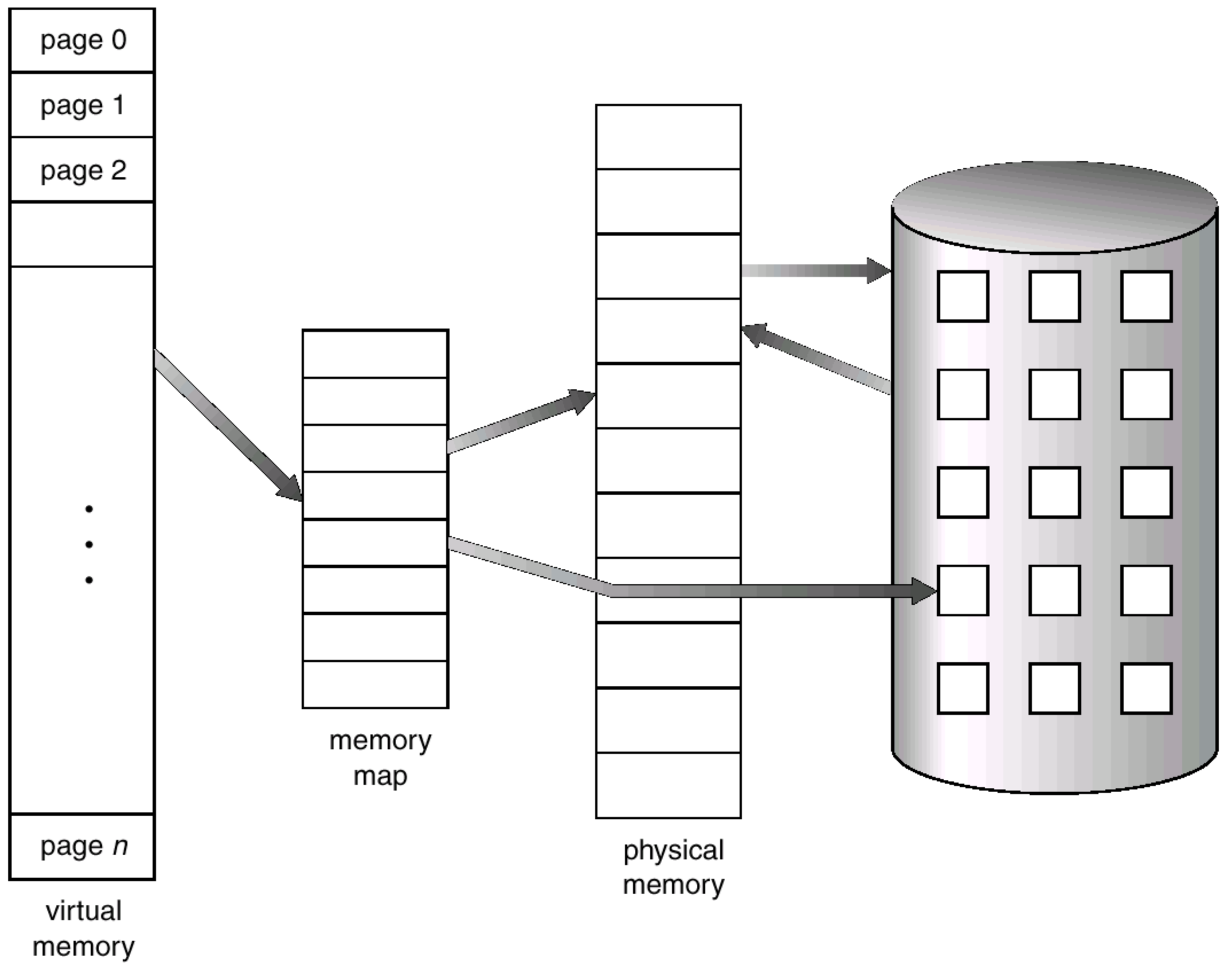
➔ 4 GB of logical address space for each process

➔ A large portion of the address space is unused

Background

- Virtual memory: separation of 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
  - Need to allow pages to be swapped in and out
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Larger than Physical Memory



Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- Lazy swapper
  - Never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

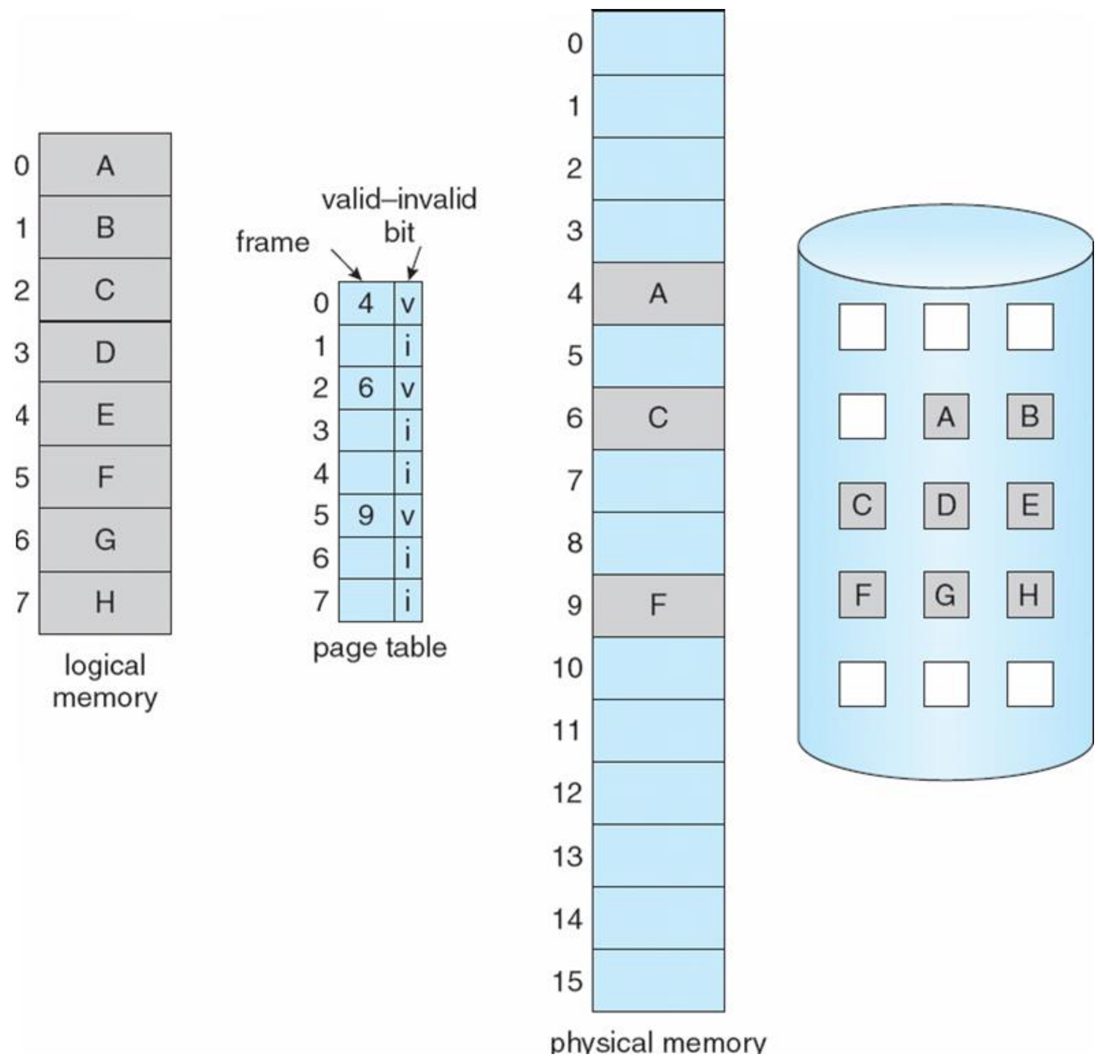
Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated
  - Valid (v) : in-memory
  - "Invalid" means both:
    - *illegal*: this page is not within the address space of the process
    - *not-in-memory*: this page has never been loaded from disk before
    - *obsolete*: it is from disk, but disk copy (original) has been updated

      eg: KAL reservation system:

      one global disk (headquater) – N computers / branch
  - Initially all entries are set to *invalid*
- During address translation, if the page is `invalid' ⇒ **"page fault"**

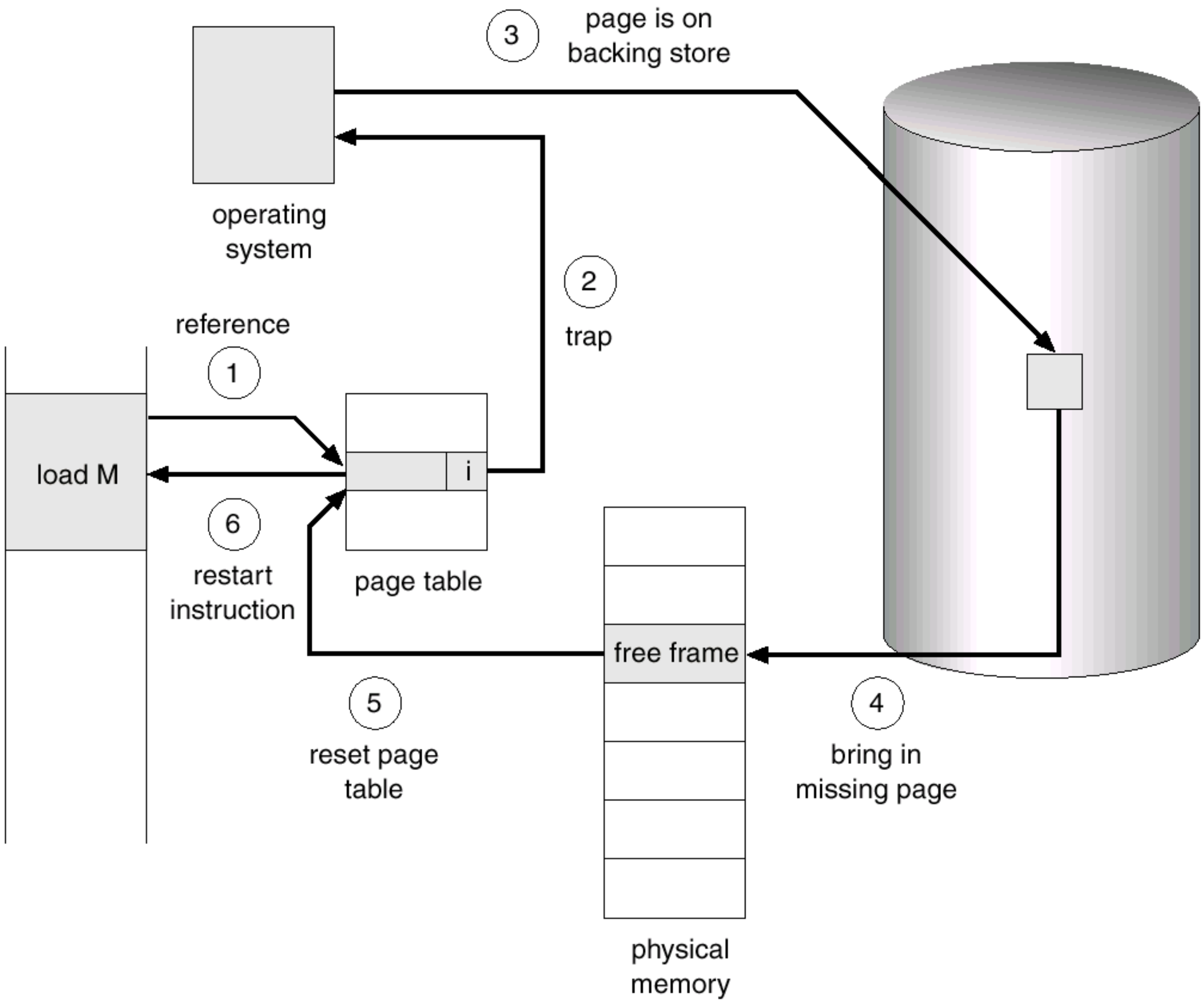
Page Table when Some Pages are not in Main Memory



Page Fault

- Access to invalid page causes HW (MMU) trap – page fault trap
- Trap handler is within OS: page fault handler is invoked
- OS handles the page fault as follows:
  1. OS looks at another table to decide
     - illegal reference? eg. bad address, protection violation $\Rightarrow$ abort process
     - not in memory? Then continue
  2. Get an empty page frame (If no free frame, replace!)
  3. Read the page into the frame from disk
     - The process remains in *'wait'* state until this disk I/O finishes
     - After disk I/O finishes, page table entries are updated (frame #, valid/invalid bit = "valid")
     - Move the process to the Ready queue – dispatch later
  4. Page fault trap finishes when CPU is assigned to the process, again
  5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Difficulties in actual HW design

When does Page Fault occur?

1. on instruction fetch: okay
2. on operand fetch:

   restart needed → (instruction fetch, decode, operand-fetch)

3. Worst case: when an instruction updates multiple locations
   - Ex: Block copy instruction:

copy	count	from_address	to_address
<op-code>	<bytes>	<source>	<destination>

spans two blocks

   - If page fault occurs while trying to write to the second block?
     - Undo
     - Needs additional H/W that stores temporary addresses and values

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – *p*) x memory access + *p* (page fault overhead + [swap page out if needed] + swap page in + restart overhead)

- Demand paging example
  - Memory access time = 200 nanoseconds (*ns*)
  - Average page-fault service time = 8 miliseconds (*ms*)
  - EAT = (1 – p) x 200 + p x 8 ms = (1 – p) x 200 + p x 8,000,000 = 200 + p x 7,999,800 *ns*
  - If one access out of 1,000 causes a page fault, EAT = 8.2 *μs*
  - This is a slowdown by a factor of 40!!

Performance of Demand Paging

- Pure demand paging
  - Never swap-in until referenced
  - Start program with no page in memory
- Locality of reference
  - Occurs in almost all workloads
  - Page references occur to very small set of pages in a certain time interval
  - Makes Paging system feasible

What happens if there is no free frame?

- Page replacement
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
  - Use *modify (dirty) bit* to reduce overhead of page transfers – only modified pages are written (swap-out) to disk
  - Page replacement completes separation between logical memory and physical memory
    - Large virtual memory can be provided on a smaller physical memory
  - Same page may be brought into memory several times during run

- Page replacement algorithm
  - Algorithm for choosing victim page for replacement
  - Goal – minimize the number of page faults

Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and free frame tables

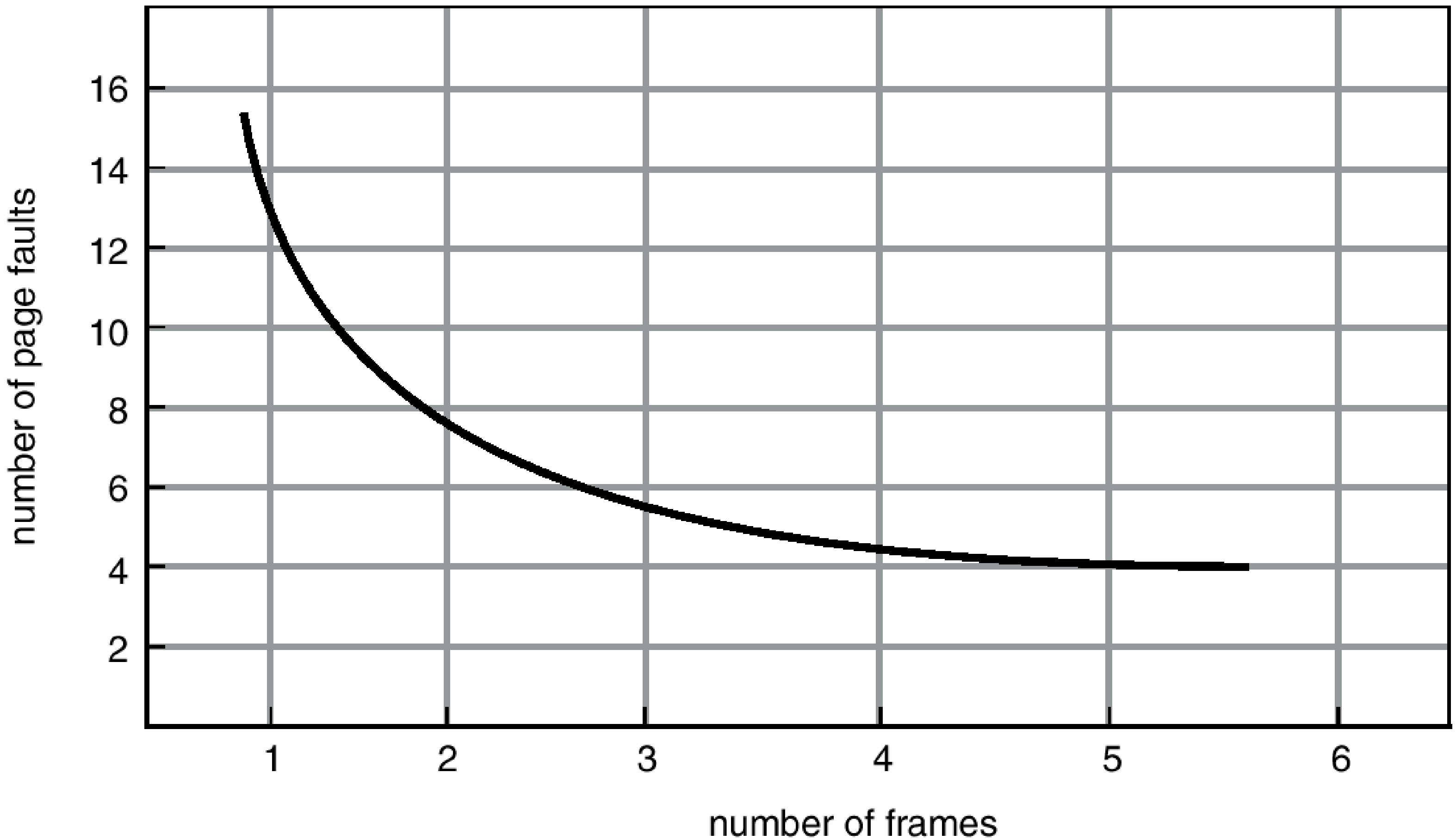
4. Restart the process

Page-Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

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

Graph of Page Faults v.s. the Number of Frames



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	1	4	5
2	2	1	3
3	3	2	4

9 page faults

More page faults?

- **4** frames

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

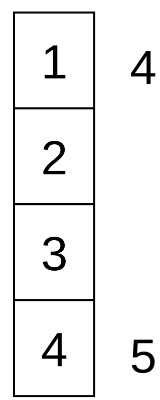
10 page faults

- FIFO Replacement – Belady’s Anomaly
  - more frames $\Rightarrow$ more page faults

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



6 page faults

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

Least Recently Used (LRU) Algorithm

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

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

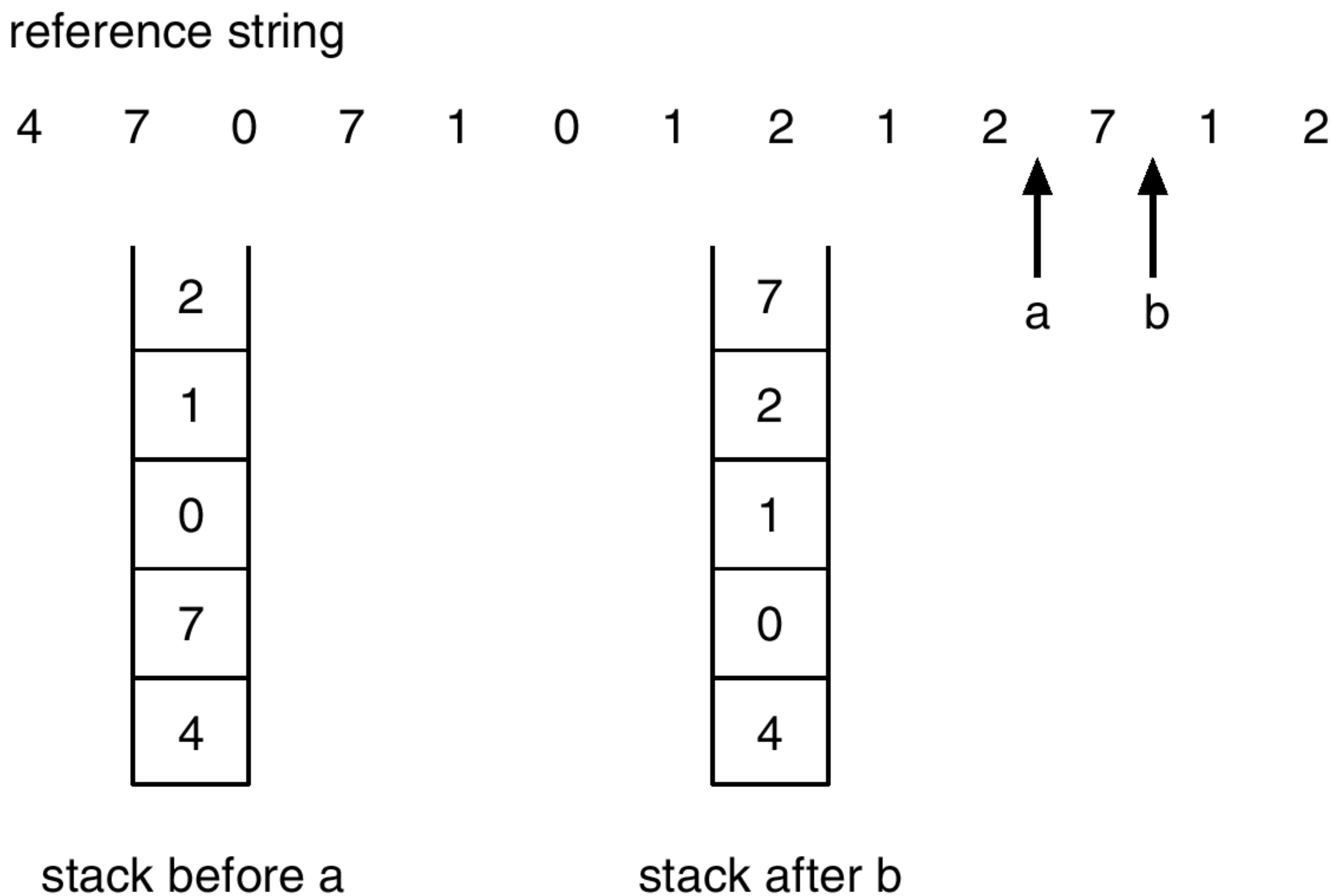
- Problem: How to implement LRU ?
  - If we implement LRU as it is,
    - Timestamp needed for every page: extra memory (page table) traffic
    - Need to find the page whose timestamp is the smallest
  - Too large space/time overhead to be incorporated into the Kernel
  - Approximation model for implementation needed

LRU Implementation Algorithms

- Counter implementation
  - Every page entry has a counter;
  - CPU counter is incremented at every memory reference (logical clock)
    CPU counter = Number of total memory references
  - When page A is accessed, copy the CPU counter into the A's counter
  - At replacement, search page table for minimum counter
    - Extra memory access (counter write time) – in each memory access
    - Search (time) overhead – in each replacement
    - Counter (space) overhead, ….

- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page A referenced:
    - move page A to the top
    - requires 6 pointers to be changed (including pointer to stack top)
  - No search for replacement

Use of a Stack to Record the Most Recent Page References



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, Initially = 0
  - When page is referenced, bit set to 1
  - Replace the one whose reference bit is 0 (if one exists)
    - We do not know the order, however

- Additional-Reference-Bits Algorithm
  - 8 bits for additional reference bits
  - Reference bit is shifted to the highest order bit of the additional reference bits, periodically
  - Shift the other bits right 1 bit, discarding the low-order bit
  - Ex: 00000000 – never referenced page

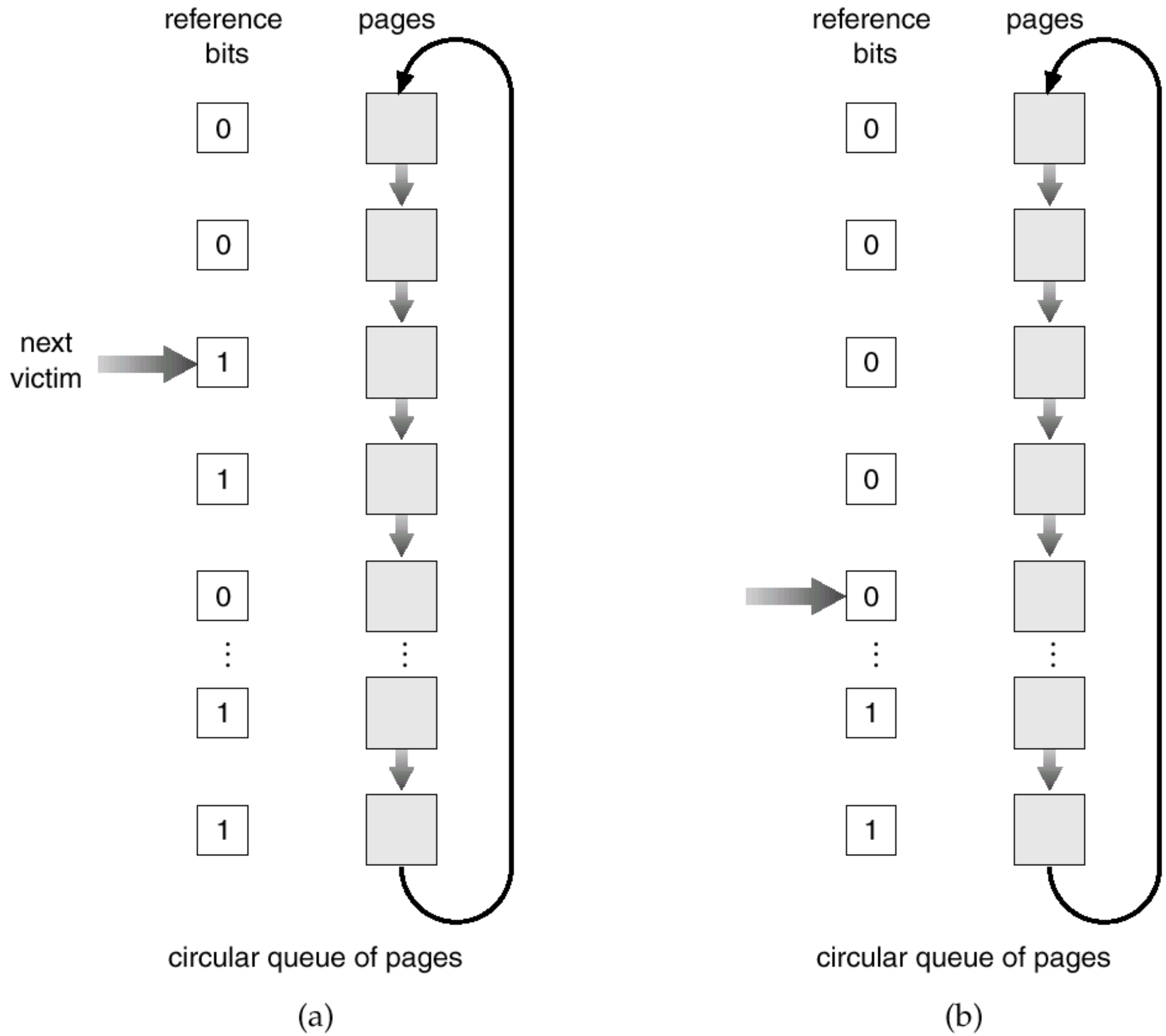
    10101010 – accessed in every two periods

LRU Approximation Algorithms

- Second chance (clock) algorithm
  - Needs a reference bit
  - Circular queue of pages
  - Advance pointer until it finds reference bit 0 (never referenced)
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - Set reference bit to 0 and leave the page in memory
    - Replace next page (in clock order), subject to same rules
  - Characteristics:
    - 포인터 이동하는 중에 reference bit 1 은 모두 0 으로 바꿈
    - 한 바퀴 되돌아와서도 (second chance) 0이면 그때에는 replace 당함
    - 자주 사용되는 페이지라면 second chance 가 올때 1
    - 최악의 경우 모든 bit 이 1 이면 FIFO 가 됨
- Enhanced Second chance algorithm

	Reference bit	Modify bit	
•	Not-Referenced	not-modified	첫번째로 replace
•	Referenced	modified	가장 나중 replace

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU (Least Frequently Used) Algorithm
  - Replaces page with smallest count
- MFU (Most Frequently Used) Algorithm
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used