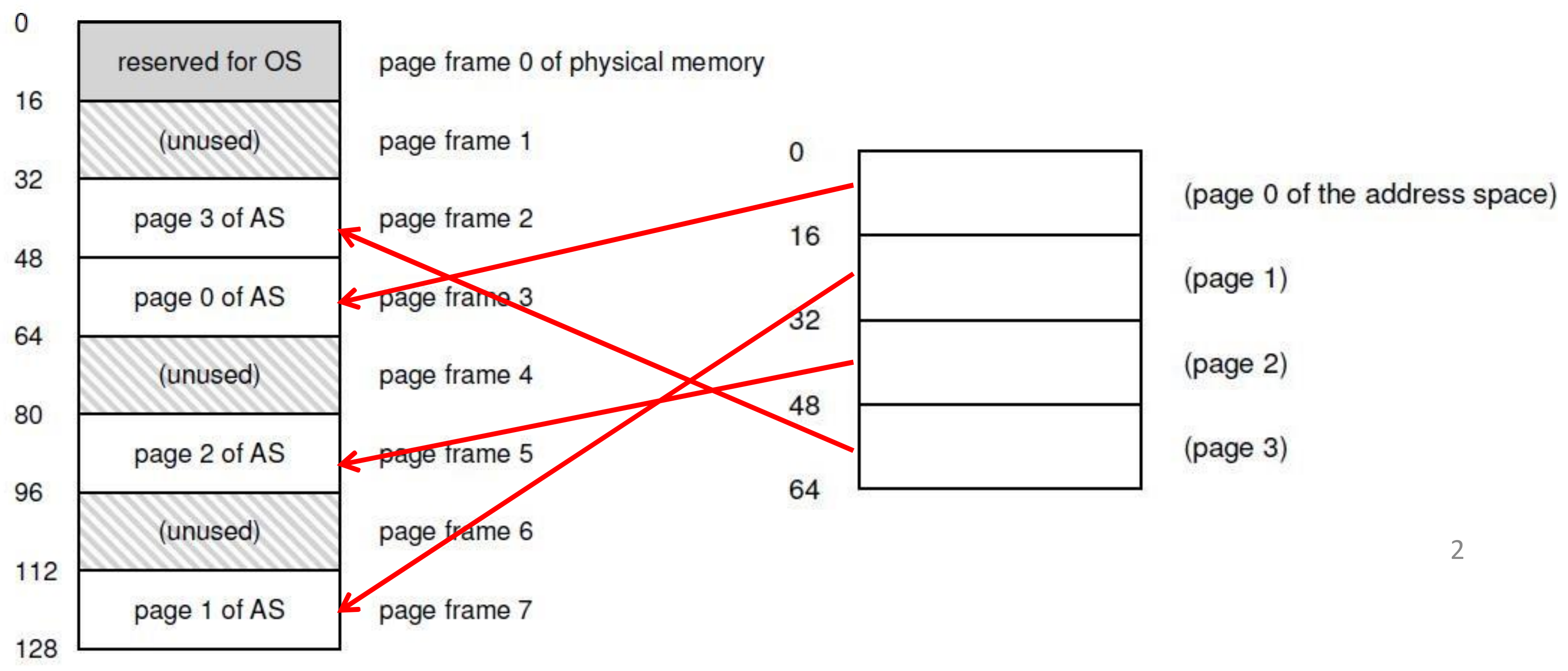
Paging and Demand Paging

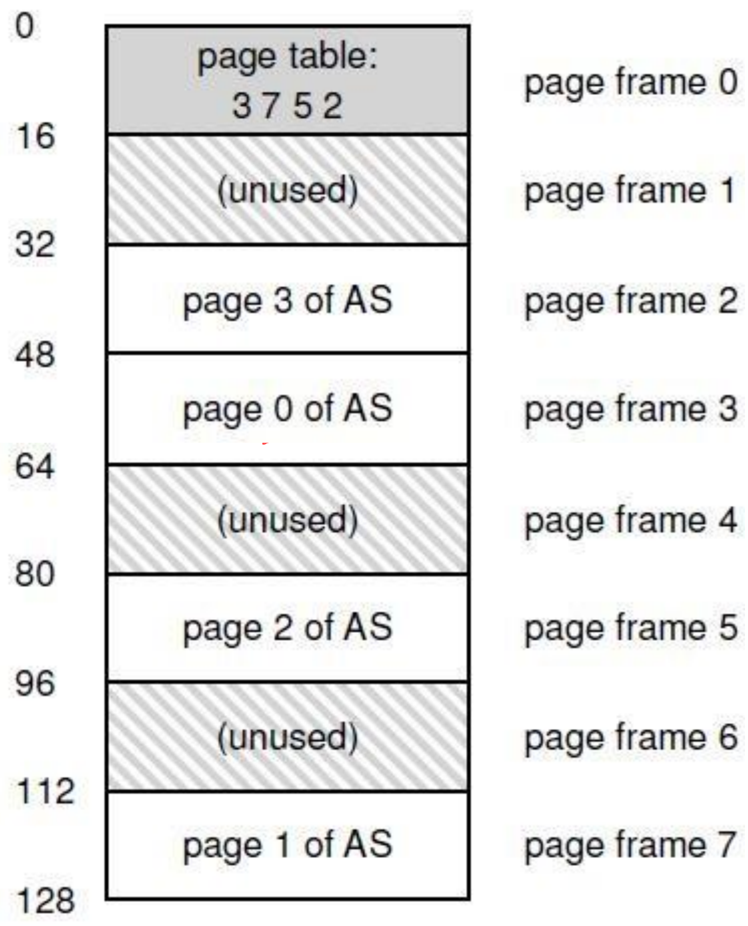
Paging

- Allocate memory in fixed size chunks ("pages")
- Avoids external fragmentation (no small "holes")
- Has internal fragmentation (partially filled pages)



Page table

- Per process data structure to help VA-PA translation
- Array stores mappings from virtual page number (VPN) to physical frame number (PFN)
  - E.g., VP 0 → PF 3, VP 1 → PF 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address translation
- OS updates page table upon context switch

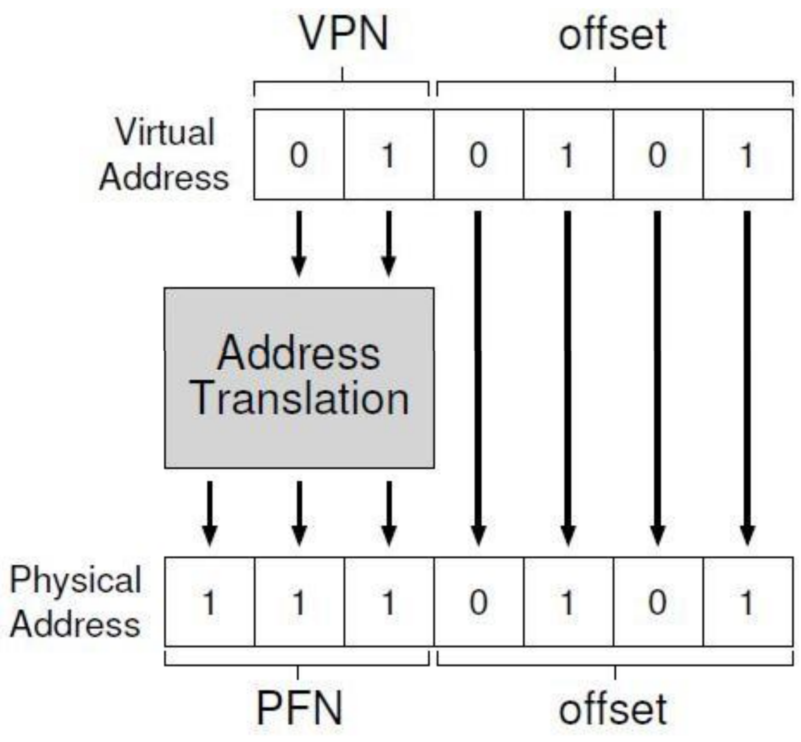


Page table entry (PTE)

- Simplest page table: linear page table
- Page table is an array of page table entries, one per virtual page
- VPN (virtual page no.) is index into this array
- Each PTE contains PFN (physical frame number) and few other bits
  - Valid bit: is this page used by process?
  - Protection bits: read/write permissions
  - Present bit: is this page in memory? (more later)
  - Dirty bit: has this page been modified?
  - Accessed bit: has this page been recently accessed?

Address translation in hardware

- Most significant bits of VA give the VPN
- Page table maps VPN to PFN
- PA is obtained from PFN and offset within a page
- MMU stores (physical) address of start of page table, not all entries.
- “Walks” the page table to get relevant PTE



What happens on memory access?

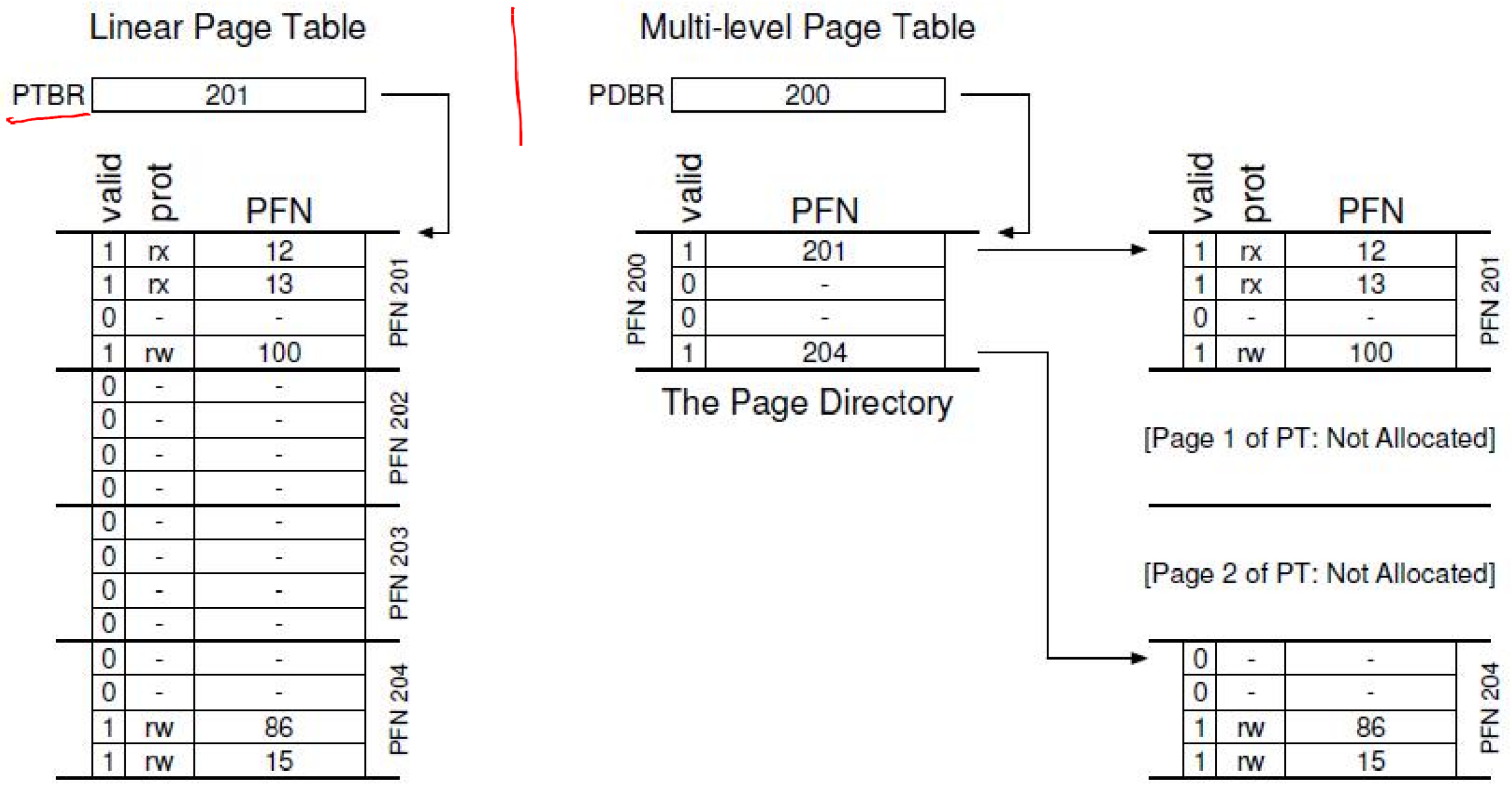
- CPU requests code or data at a virtual address
- MMU must translate VA to PA
  - First, access memory to read page table entry
  - Translate VA to PA
  - Then, access memory to fetch code/data
- Paging adds overhead to memory access
- Solution? A cache for VA-PA mappings

Translation Lookaside Buffer (TLB)

- A cache of recent VA-PA mappings
- To translate VA to PA, MMU first looks up TLB
- If TLB hit, PA can be directly used
- If TLB miss, then MMU performs additional memory accesses to "walk" page table
- TLB misses are expensive (multiple memory accesses)
  - Locality of reference helps to have high hit rate
- TLB entries may become invalid on context switch and change of page tables

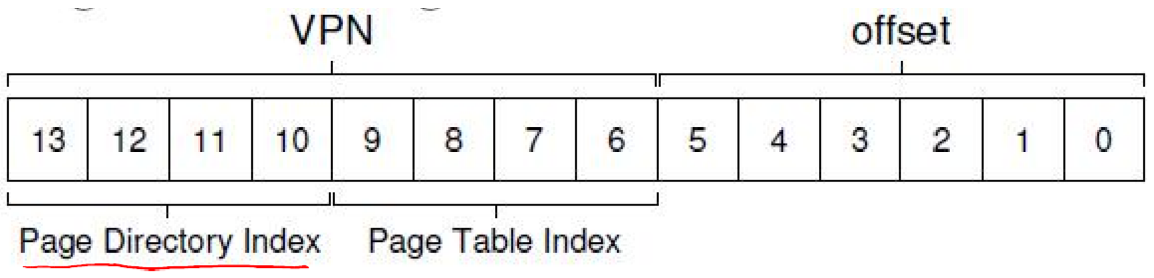
Multilevel page tables (1)

- A page table is spread over many pages
- An “outer” page table or page directory tracks the PFNs of the page table pages



Multilevel page tables (2)

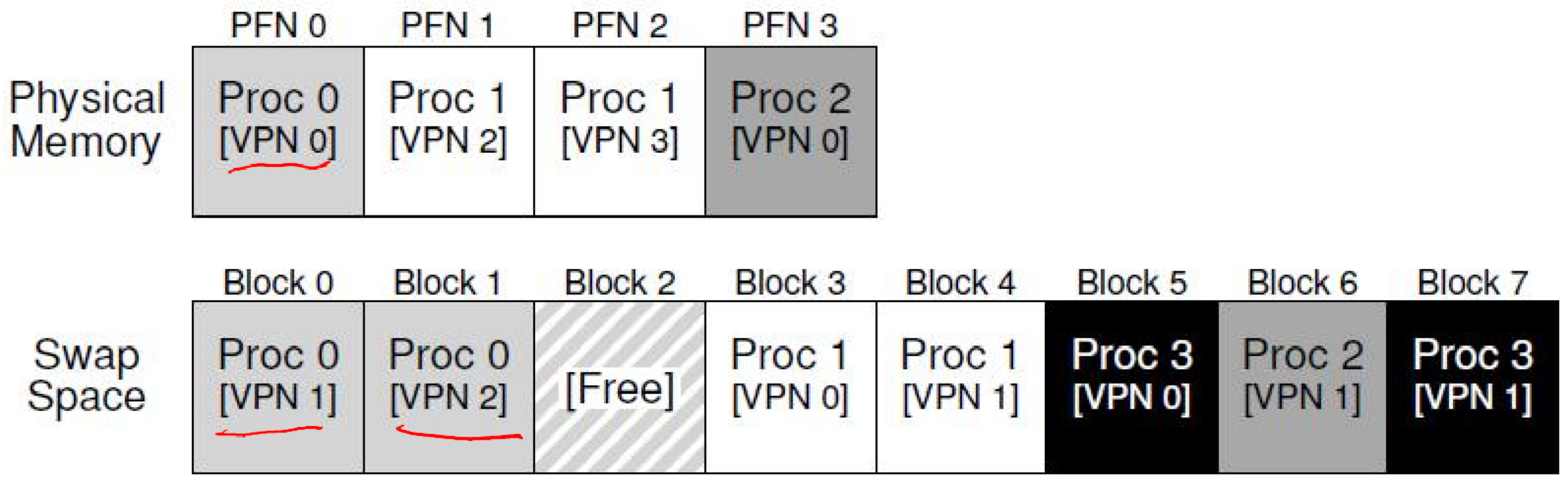
- Depending on how large the page table is, we may need more than 2 levels also
  - 64-bit architectures may need 7 levels
- What about address translation?
  - First few bits of VA to identify outer page table entry
  - Next few bits to index into next level of PTEs



- In case of TLB miss, multiple accesses to memory required to access all the levels of page tables

Is main memory always enough?

- Are all pages of all active processes always in main memory?
  - Not necessary, with large address spaces
- OS uses a part of disk (swap space) to store pages that are not in active use



Page fault

- Present bit in page table entry: indicates if a page of a process resides in memory or not
- When translating VA to PA, MMU reads present bit
- If page present in memory, directly accessed
- If page not in memory, MMU raises a trap to the OS – page fault

Page fault handling

- Page fault traps OS and moves CPU to kernel mode
- OS fetches disk address of page and issues read to disk
  - OS keeps track of disk address (say, in page table)
    - OS context switches to another process
      - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process scheduled again, OS restarts the instruction that caused page fault

Summary: what happens on memory access

- CPU issues load to a VA for code or data
  - Checks CPU cache first
  - Goes to main memory in case of cache miss
- MMU looks up TLB for VA
  - If TLB hit, obtains PA, fetches memory location and returns to CPU (via CPU caches)
  - If TLB miss, MMU accesses memory, walks page table, and obtains page table entry
    - If present bit set in PTE, accesses memory
    - If not present but valid, raises page fault. OS handles page fault and restarts the CPU load instruction
    - If invalid page access, trap to OS for illegal access

More complications in a page fault

- When servicing page fault, what if OS finds that there is no free page to swap in the faulting page?
- OS must swap out an existing page (if it has been modified, i.e., dirty) and then swap in the faulting page – too much work!
- OS may proactively swap out pages to keep list of free pages handy
- Which pages to swap out? Decided by page replacement policy.

Page Replacement Algorithms

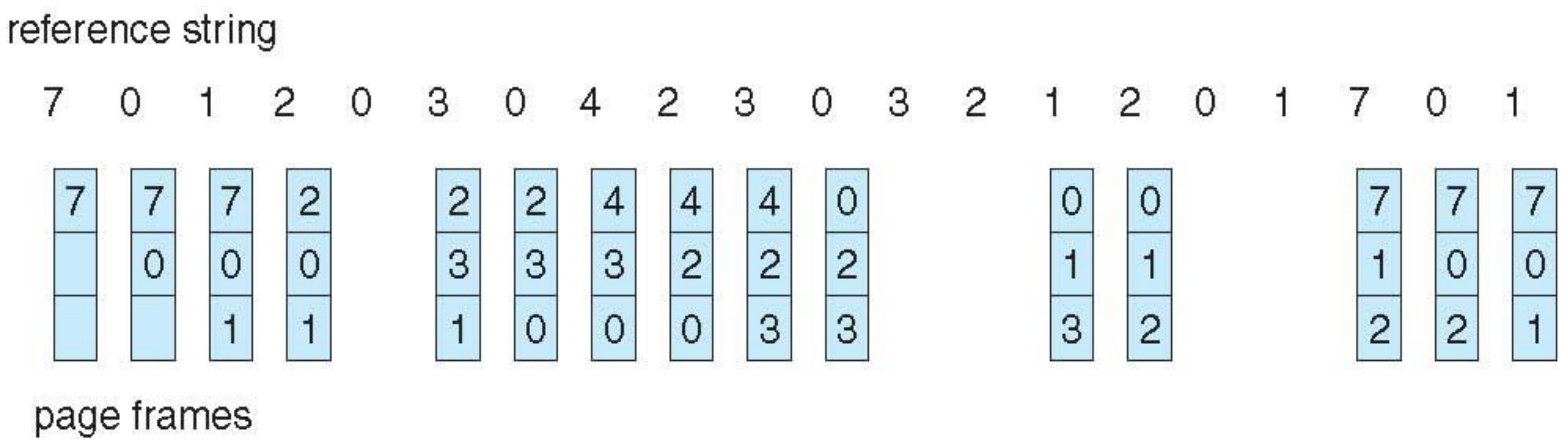
Want lowest page-fault rate

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

FIFO Page Replacement

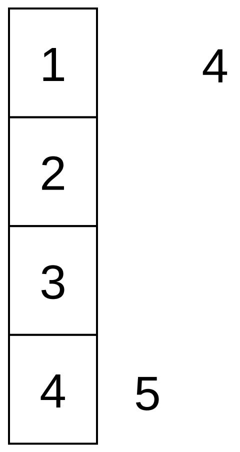


Optimal Algorithm

Replace page that will not be used for longest period of time

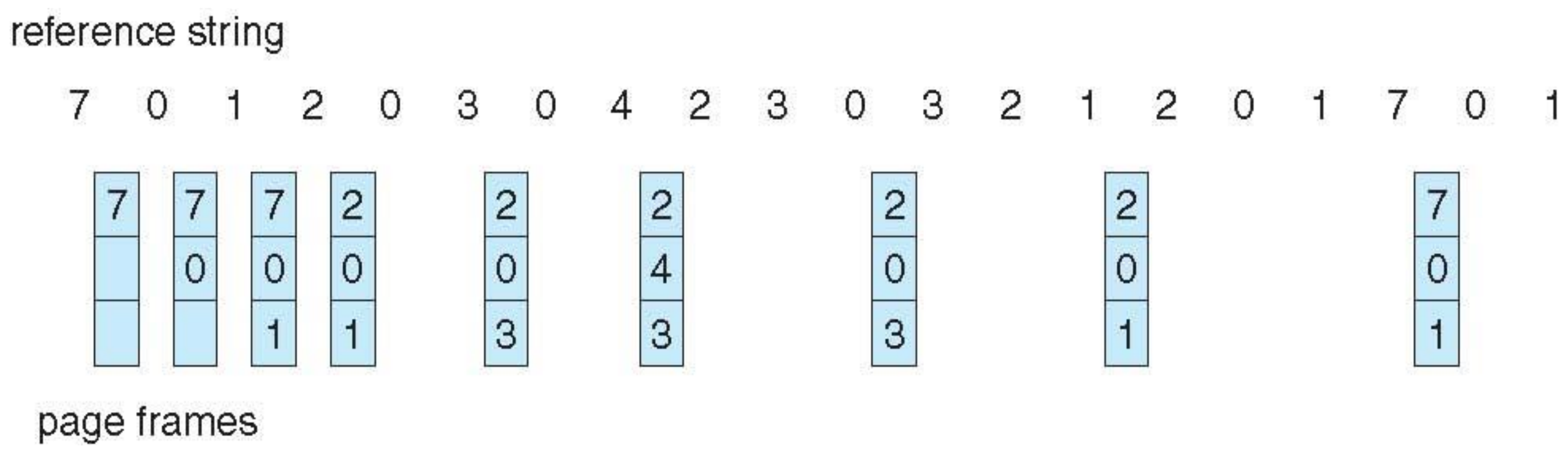
4 frames example

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



6 page faults

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

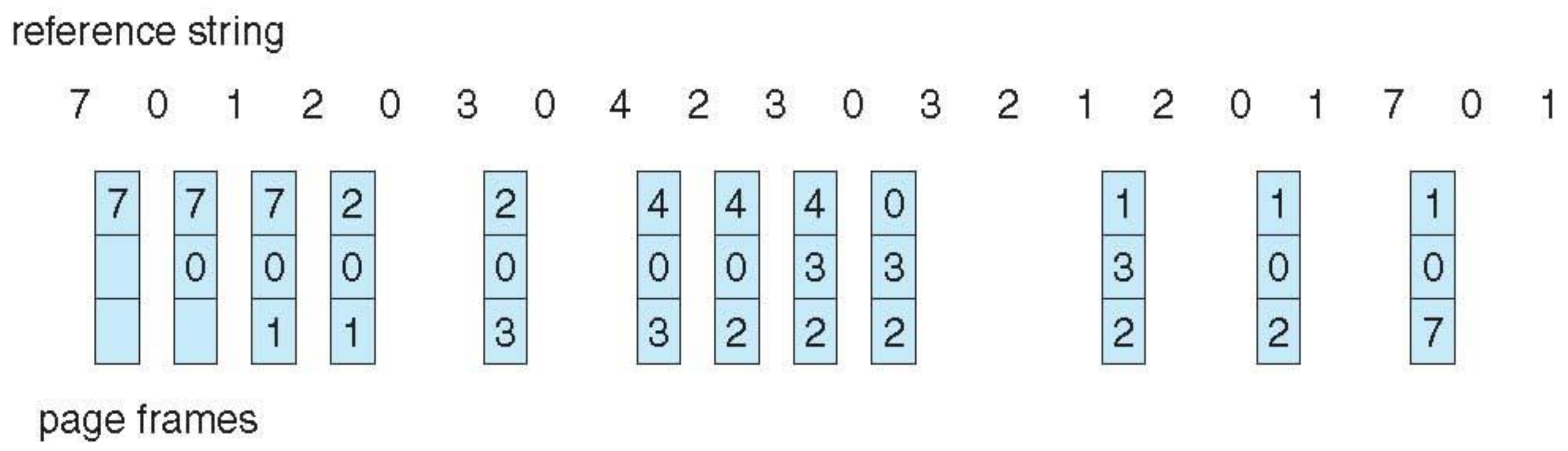
Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

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

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

LRU Page Replacement



LRU Algorithm (Cont.)

Stack implementation – keep a stack of page numbers in a double link form:

- Page referenced:
  - move it to the top
  - requires 6 pointers to be changed
- No search for replacement