Memory Management - IV

Page Replacement Algorithms

The Need To Replace A Page

- When a page is referenced by a process, it is possible that the needed page is not in memory, resulting in a page fault.
- The missing page needs to be brought into the memory.
- What if there is no free memory frame for the needed page?
- We need to remove an existing page to make space for the new page!

Paging depends on Locality

- Processes tend to reference pages in sequential manner unless JMP/CALL is encountered but that constitutes a fraction of program (except for intentional spaghetti like programs with JMPs).
- Paging: keep pages required in immediate future.
- Temporal locality: more likely to be referenced again
- Spatial locality: future references are grouped together (in memory space)
- Both properties are true for both data and code

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. Read the desired page into the (newly) free frame. Update the page and frame tables.
4. Restart the process.

Virtual Memory Management

- Page Fault
  - Virtual Page not mapped to Physical Memory
  - Triggered by clear ‘present’ bit
- Page Fault: Interrupt to OS
  - Interrupt handler locates and loads the page
  - Information available in PCB (process control block)
- Location of a Page: Hard disk (file system), hard disk (swap area), memory (page fault in cache)

Page Fault

- Demand Paging
  - Page to be brought in only when needed
- Anticipatory Paging
  - Bring in page in advance anticipating its requirement in near future
  - Spatial and Temporal locality justifies this choice.
  - Also called prepaging
- OS may use hybrid of these.

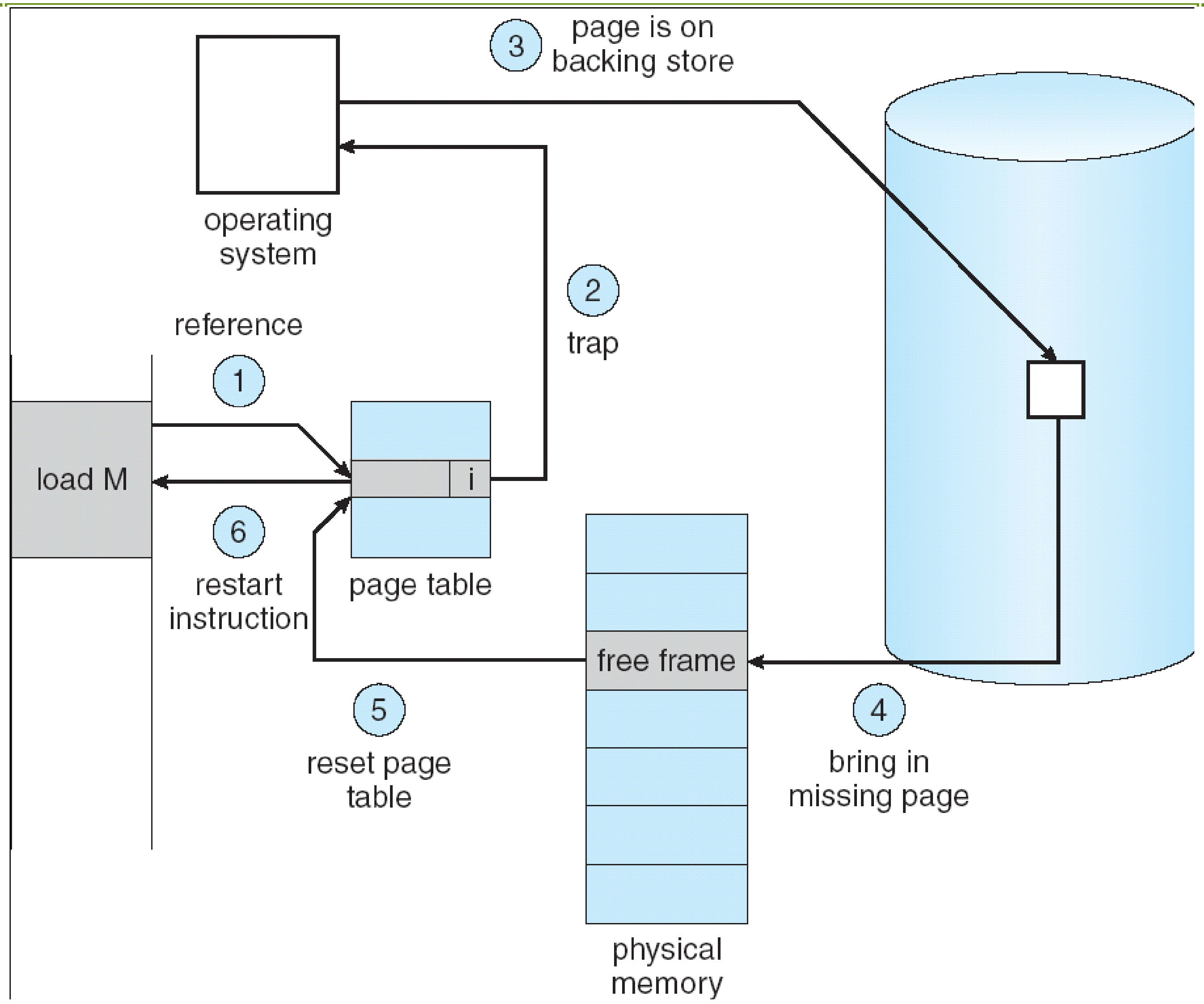
How much to page in at a time

- Demand paging: pros and cons
  - Loads a page when necessary -> physical memory usage is minimum
  - Reading only one page at a time -> performance overhead (slower)
  - Batching pages: bringing in a batch of pages improve performance
- Prepaging (Anticipatory Paging)
  - Is where the system tries to anticipate page faults before they happen
  - Bring in more page than just the requisite page (use locality to performance advantage)
  - Problem is that page faults can not be predicted.

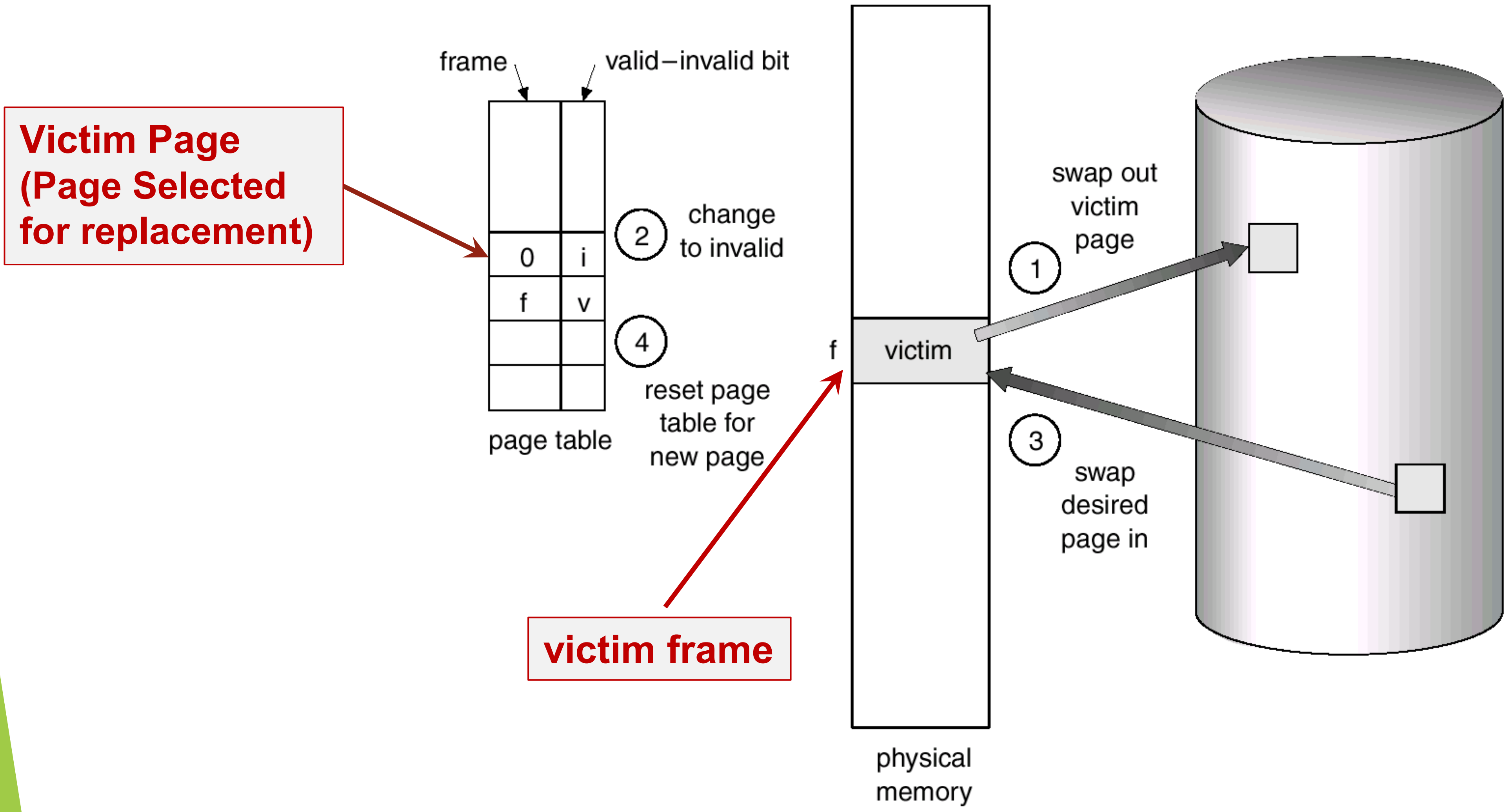
Demand Paging Mechanisms

- Page Fault: user references page with invalid PTE?
- Memory Management Unit (MMU) traps to OS
- What does OS do on a Page Fault?:
  - Choose an old page to replace
  - If old page modified (“D=1”), write contents back to disk
  - Change its PTE and any cached TLB to be invalid
  - Load new page into memory from disk
  - Update page table entry, invalidate TLB for new entry
  - Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue

Page Fault: Fetching



Page Fault: Replacement



Time for Page Fault Handling

- Hit: page in memory (valid bit set)
- Miss: page not in memory (valid bit reset)

```
If page found in memory
        Access from memory
Else
        Access from hard disk
```

- Hit time: time to access a page in event of hit
  - Hit time: avg. memory access
- Miss time: time to access a page in event of miss
  - Miss time: avg. memory access + avg. disk access
- EAT: Effective Access Time
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time

Time for Page Fault Handling

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1- p = Probably of hit

- Express EAT in terms of p.
- Page Fault is 1 in 1000 ($p = 0.001$), what is EAT and slowdown?
- What is page fault if slowdown by less than 10%?

Time for Page Fault Handling

- EAT: Effective Access Time
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - EAT = 200ns + p x 8 ms
    = 200ns + p x 8,000,000ns
- Page Fault is 1 in 1,000, what is EAT and slowdown?
- What is page fault if want slowdown by less than 10%?

Time for Page Fault Handling

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1- p = Probably of hit
- Express EAT in terms of p.
  - EAT = 200ns + p x 8 ms = (200 + p x 8,000,000) ns
- If one access out of 1000 causes a page fault, then
  - EAT = 8.2 μs (slowdown by a factor of 40)
- What if want slowdown by less than 10%?
  - 200ns x 1.1 < EAT $\Rightarrow p < 2.5 \times 10^{-6}$
  - This is about 1 page fault in 400000!

Finding the Best Page

- One option is to increase memory
  - More physical page frames: more pages can be accommodated and page faults can be reduced
  - More expensive
  - Physical memory is not infinite. There is a limit.
- Replace the page that is not needed again
  - That way, no faults.
  - Replacing the page that won't be used for the longest period of time absolutely minimizes the number of page faults.
- Problem is how to determine this?
  - Predict based on its usage

Metrics: Page Replacement Algorithms

- **Objective:** Produce a low page-fault rate.
- ***Reference string:*** sequence of memory accesses in terms of page numbers
  - For every address accessed in a program, page offset is removed (through right shift k times) and only page number is retained.
  - Generated through traces of real programs in benchmarking applications
  - Synthetic traces through stochastic model(s)
- Evaluate algorithm on reference string and compute number of page faults.

Page Replacement Algorithms

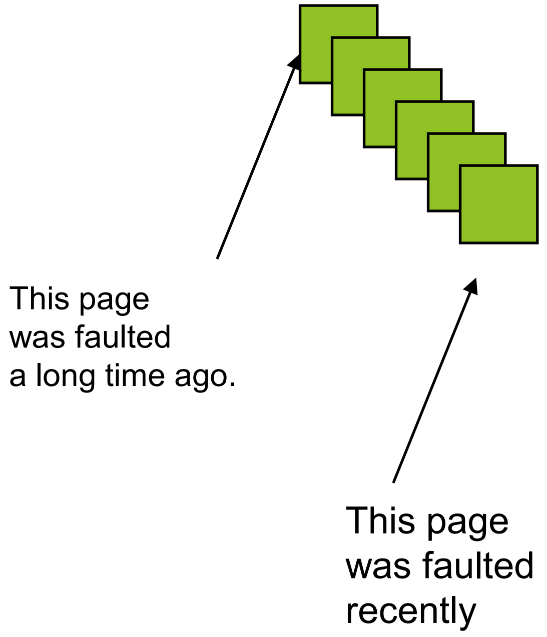
- FIFO (First In First Out):
  - A fair policy and replaces oldest page.
  - some pages are needed more than others. So may be fairness is not the best criteria.
- OPT (Optimal): theoretical best as it minimizes lowest fault rate.
  - Not practical / feasible.
- LRU (Least recently used)
  - least recently accessed page is selected for replacement
  - Argument is that a page not referenced for a long duration may not be needed in future
- LFU (Least Frequently used):
  - Basis is that a page not accesses too frequently in past may not be needed again in future.

Page Replacement Algorithms

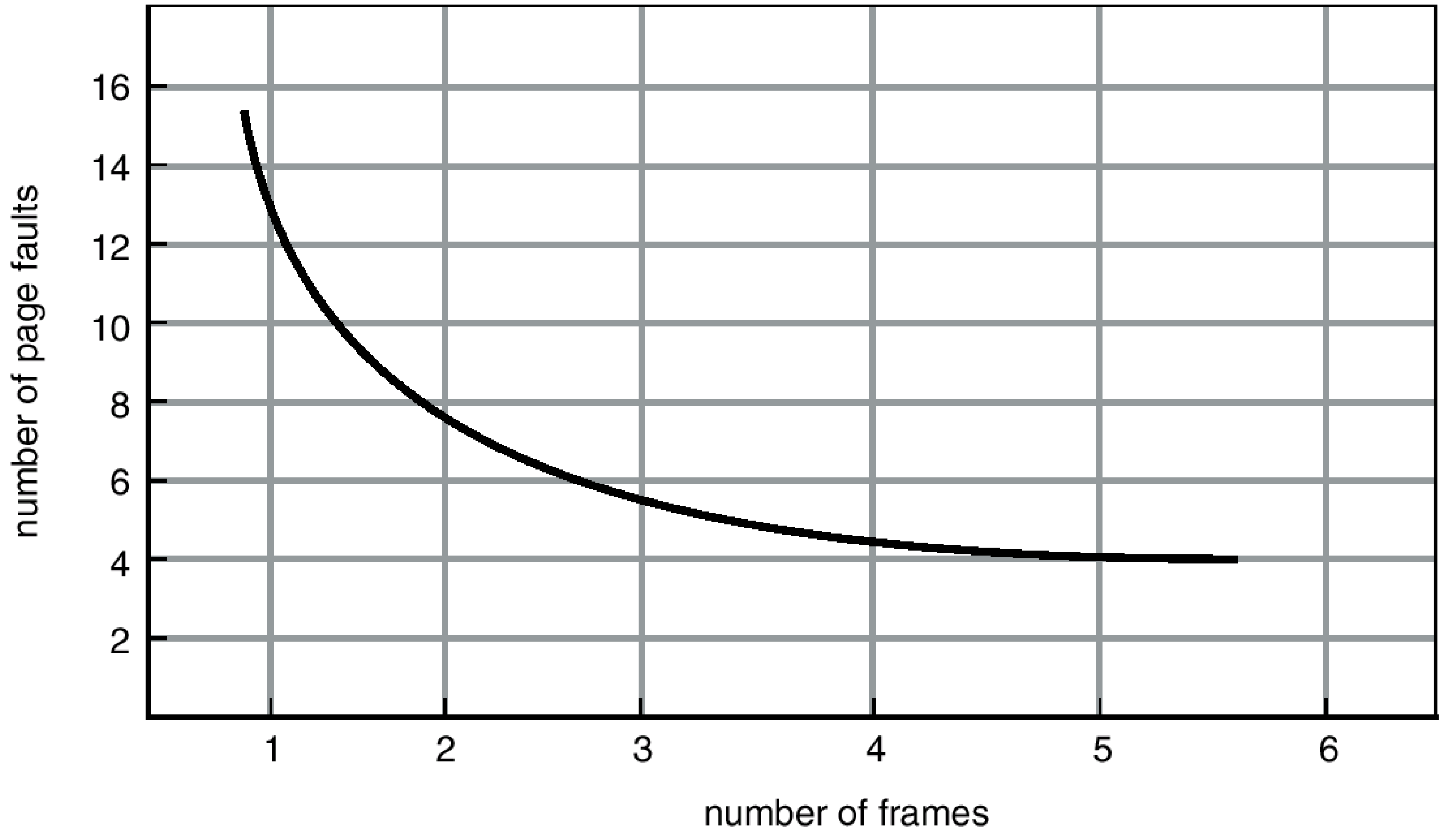
- MRU (Most recently used)
  - Argument is that a page just referenced may not be needed in future
  - programs do *not* read the same addresses multiple times. For example, a media player will read a byte and then move on, never to read it again.
- MFU (Most Frequently used):
  - Basis is that a page accessed too frequently may not be needed again in future.
- Clock/Second Chance
  - Objective: avoid a heavily referenced page from being replaced

FIFO

- FIFO: simple to implement.
- Can be implemented through queue:
  - New page at the rear of queue
  - Delete from front of queue.
- On replacement, remove the one brought in the longest time ago.
- What if first page is one being referenced a lot?
- FIFO suffers from “Belady’s anomaly”
  - the fault rate might actually increase when the algorithm is given more memory -- a bad property.


This page was faulted a long time ago.
This page was faulted recently

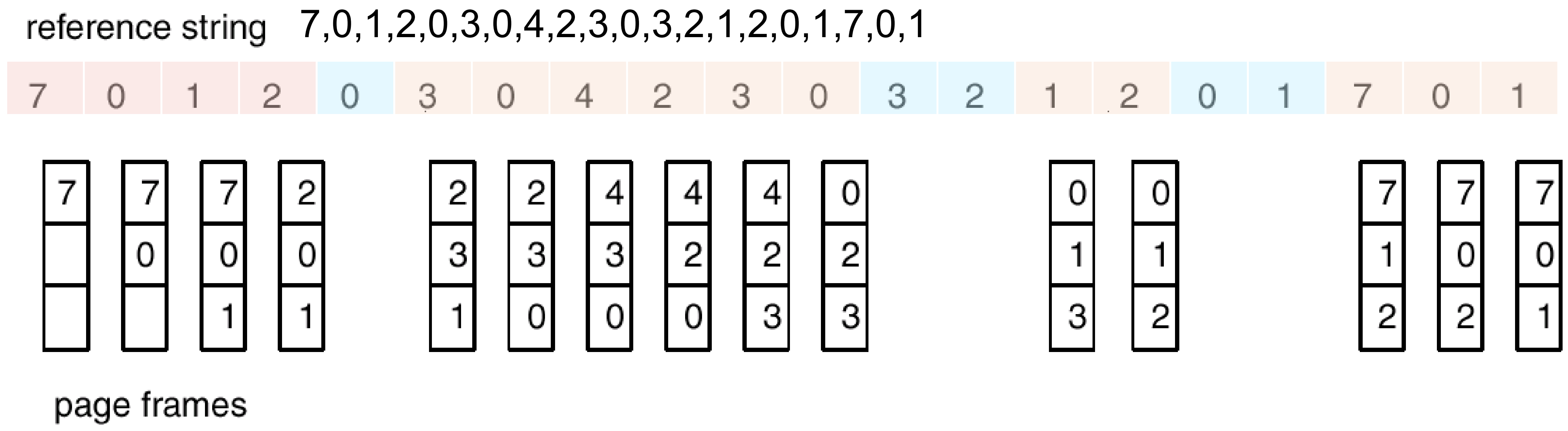
Page Faults vs The Number of Frames



Number of page faults may not reduce even after allocating more frames in FIFO algorithm.

FIFO Page Replacement

Red shade indicates page fault.
Blue shade indicates that page is in memory and hence no page fault.



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):
- 4 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):

  9 page faults

- 4 frames:
- FIFO Replacement manifests Belady’s Anomaly:
  - more frames $\Rightarrow$ more page faults

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Red (and asterisk) shows page fault
- 3 frames (3 pages can be in memory at a time):

  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**

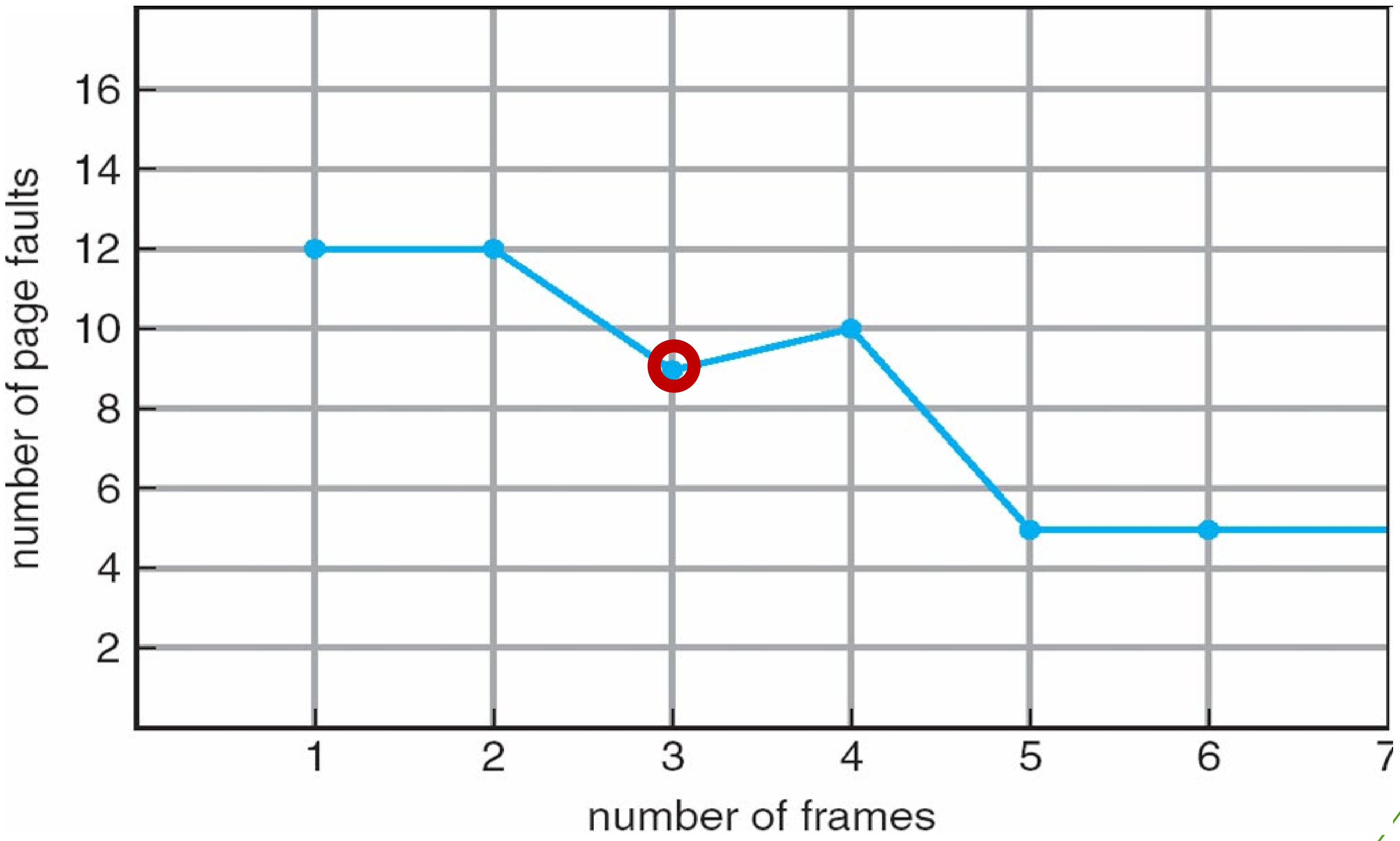
  10 page faults

- FIFO Replacement manifests Belady's Anomaly:
  - more frames $\Rightarrow$ more page faults

Belady Anomaly

- Another Example:
  - Reference String 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5
  - # Page Faults = 9 (3 frames) and 10 (4 frames)

FIFO Illustrating Belady's Anomaly

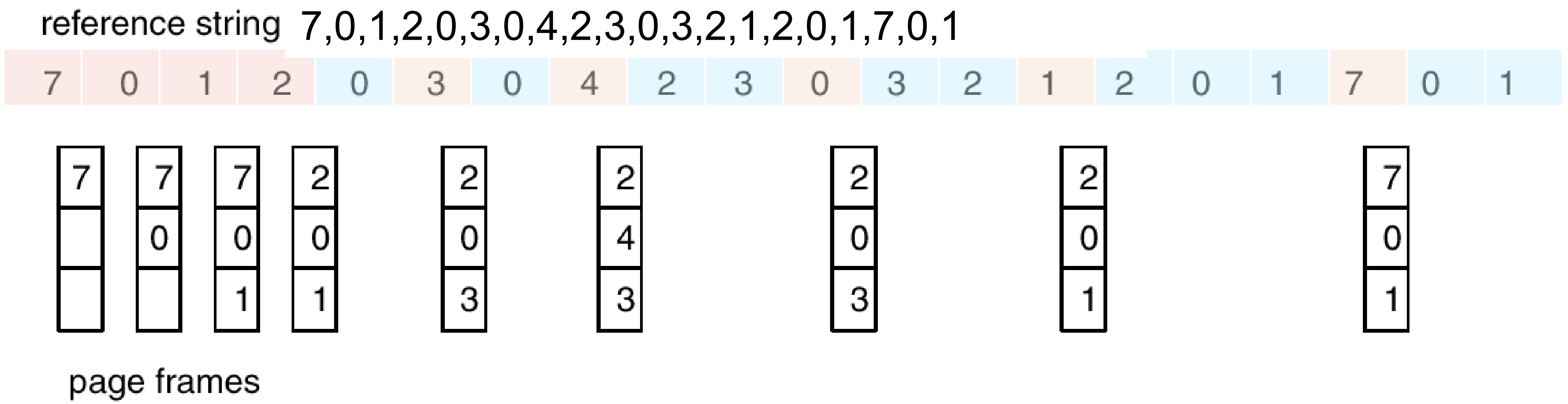


Optimal Page Replacement

- The Optimal policy selects for replacement the page that will not be used for longest period of time.
- Requires knowledge of memory accesses in future
- Practically infeasible (Impossible to implement)
- Useful as a standard to evaluate other algorithms

Optimal Page Replacement

Red shade indicates page fault.
Blue shade indicates that page is in memory and hence no page fault.



Optimal: 9 page faults

Optimal not Practical!

- Optimal page replacement gives minimum number of page faults.
- Downside: algorithm is not practical!
  - Compare to optimal CPU scheduling algorithm (Shortest-Remaining-Time-First)
- Future unknown; predicting future is only possibility
- Predict future from past to approximate optimal algorithm
- In page replacement, we use **LRU** (Least Recently Used) to approximate the optimal algorithm

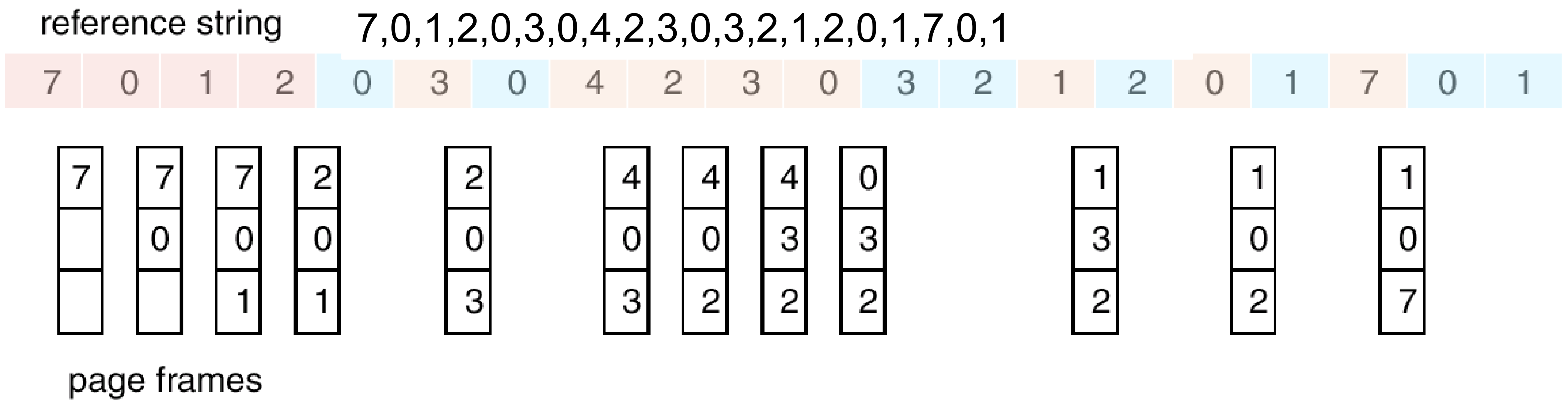
Least Recently Used (LRU)

- Predict future from past.
- LRU: on replacement, remove the page that has not been used for the longest time in the past.
  - By the principle of locality, this page least likely to be referenced in the near future.
  - performs nearly as well as the optimal policy.
- Implementation:
  - time stamp every reference

LRU Page Replacement

Red shade indicates page fault.
Blue shade indicates that page is in memory and hence no page fault.

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1



LRU: 12 page faults

LRU Algorithm

- Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- Page Frames: 4
- Page Faults?

LRU Implementations

- Counter implementation:
  - Every page entry has a counter; every time a page is referenced through this entry, RESET this counter.
  - Page with maximum value of counter is least recently used.
- Stack implementation
  - Implement a stack through double linked list
  - Move the page referenced to the top
  - Bottommost page is LRU
  - No search for replacement (if a pointer to bottom is maintained)
- Requires 6 pointers to be changed

Approximating LRU

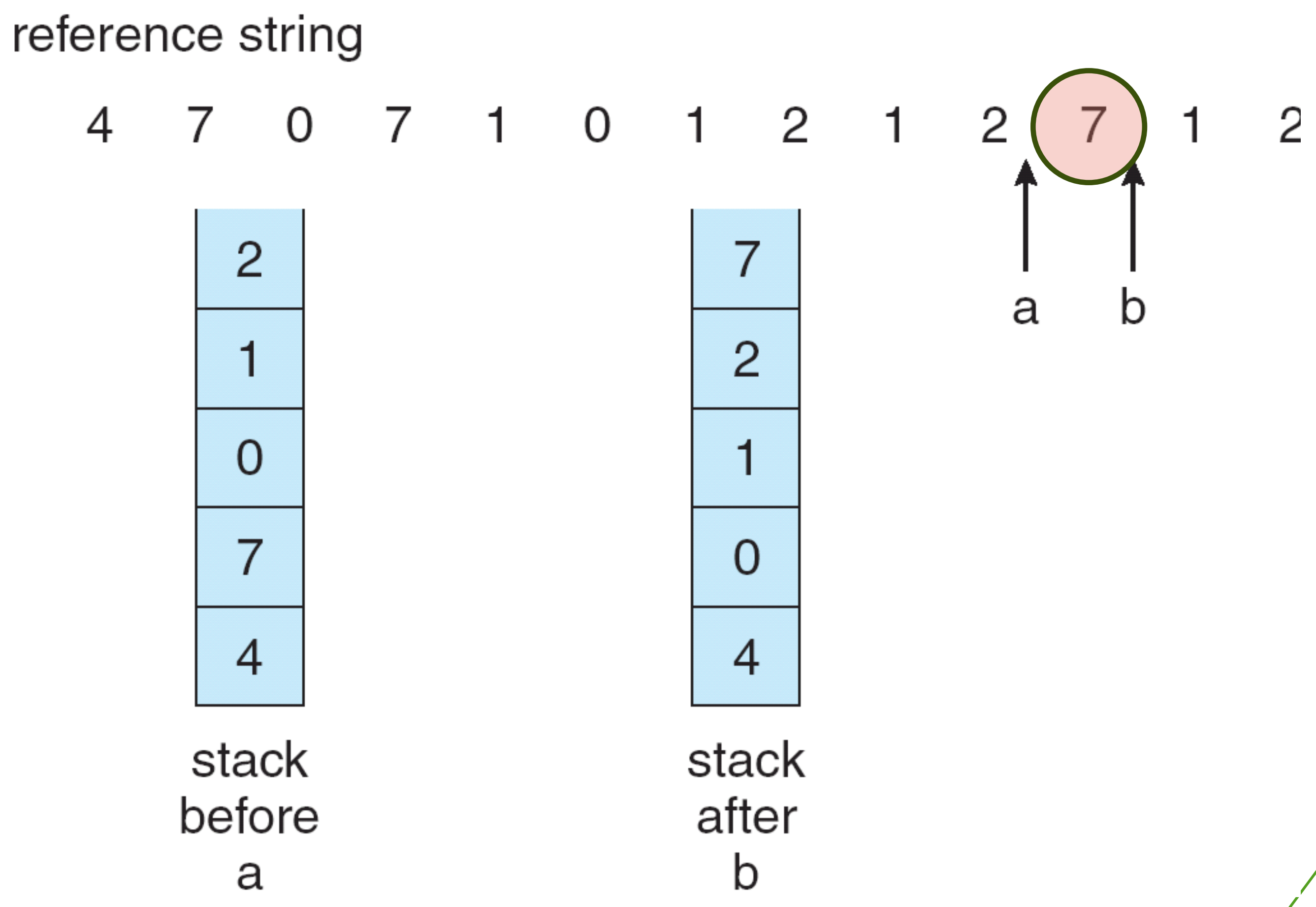
- Keep a counter for every page.
- Page is referenced
  - Reset its counter to zero
  - Increment counter for all other pages
  - Page with maximum counter is LRU
- Expensive: O(n) at every reference
- Size of Counter? How many bits?
  - 4-bit counter, 16 pages (is LRU page selected always)
  - 4-bit counter, 32 pages (is LRU page selected always)

LRU Implementation: Stack

- Stack implementation
- Double linked list

- Page referenced:
- move it to the top

- Page 7 is accessed



LRU Implementation: Hardware Matrix

Pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

$$\begin{array}{c|c|c|c|c|} & \multicolumn{4}{c}{\text{Page}} \\ & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 0 & 0 \\ \hline 3 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(a)

$$\begin{array}{|c|c|c|c|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(b)

$$\begin{array}{|c|c|c|c|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(c)

$$\begin{array}{|c|c|c|c|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(d)

$$\begin{array}{|c|c|c|c|} \multicolumn{4}{c}{\text{Page}} \\ 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 \\ \hline \end{array}$$

(e)

$$\begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ \hline \end{array}$$

(f)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(g)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(h)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 \\ \hline \end{array}$$

(i)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(j)

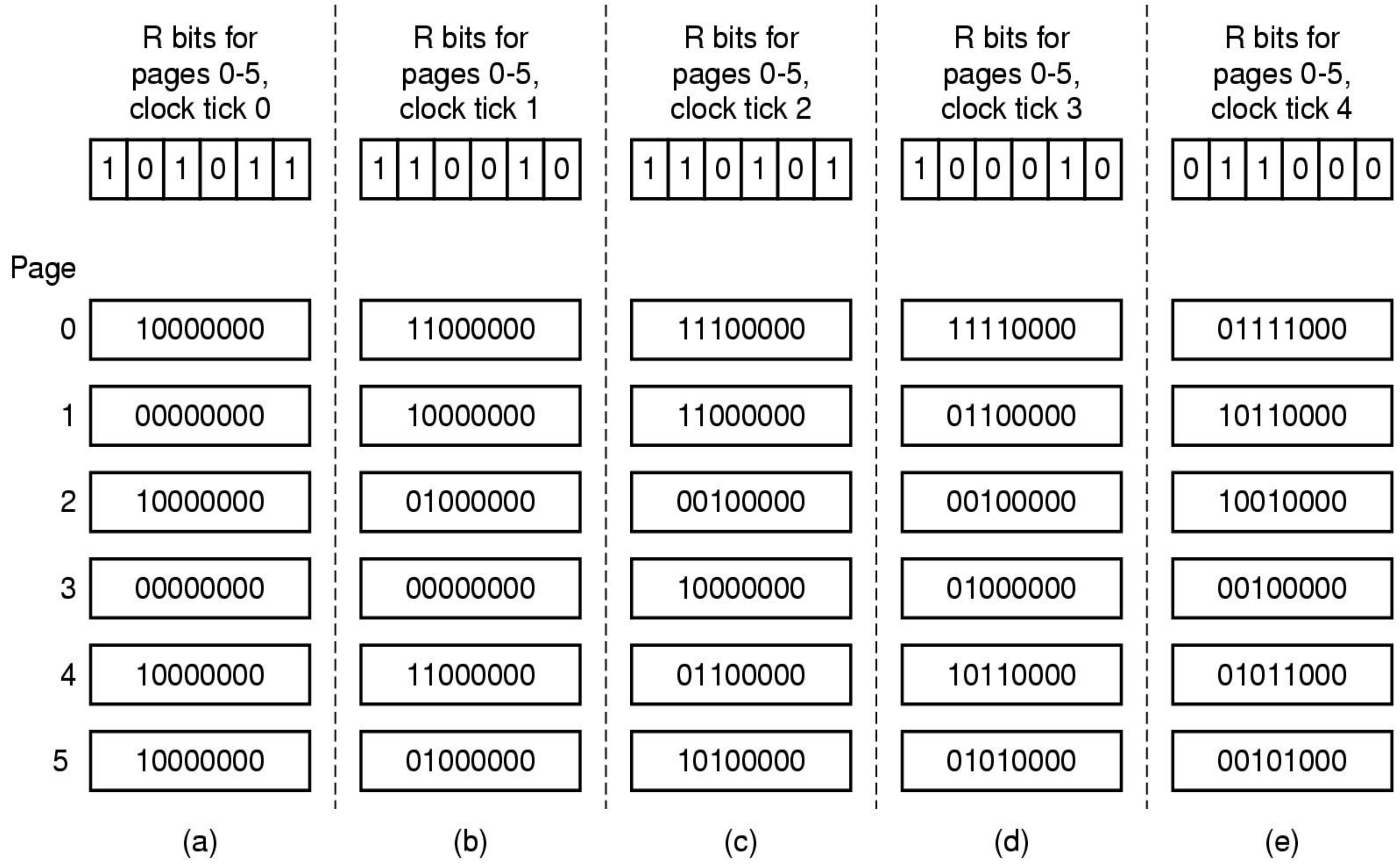
LRU Approximation: Reference Bit

- Simplest: 1-bit counter
  - With each page associate a bit, initially = 0
  - When page is referenced, bit is set to 1.
  - Replace the page which is 0 (if one exists)
  - Otherwise, select a page at random (may not replace LRU)
- Reference Bit can be used

LRU Approximation: Ageing

- Use a counter (RefCount) for number of page ace
- RefCount: Keeps count of references to page
- Initialization: new/unreferenced page, RefCount = 0
  - At regular intervals (say, every 20 ms), left shift the reference bit of each page into the high-order bit of the RefCount.
  - RefCount records the history of the page accesses (aging) for the last eight time intervals.
  - If we interpret RefCount as an unsigned integer, the page with the lowest number is the LRU page.

LRU: Example (Ageing)

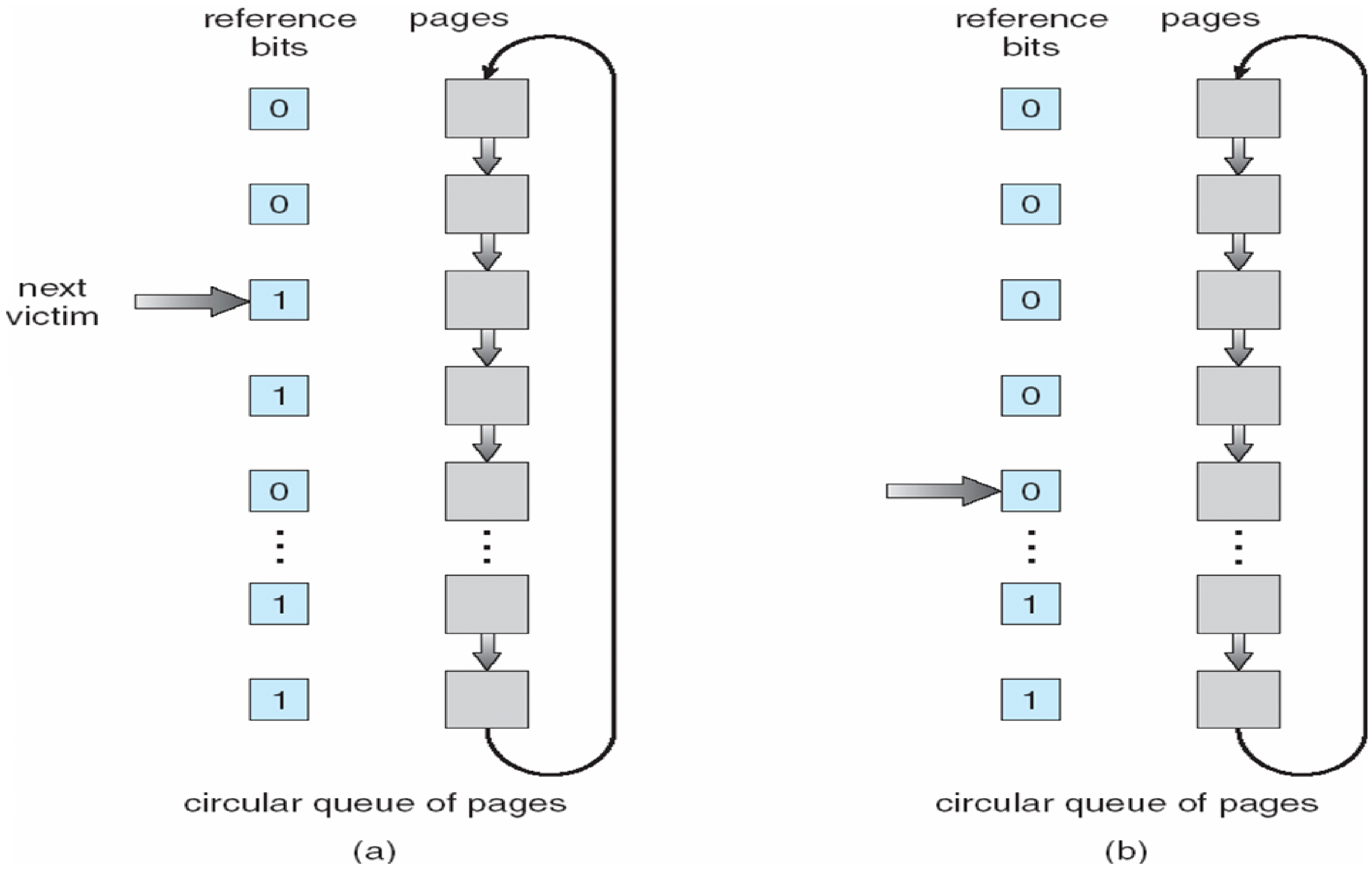


	(a)	(b)	(c)	(d)	(e)
	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Page					
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

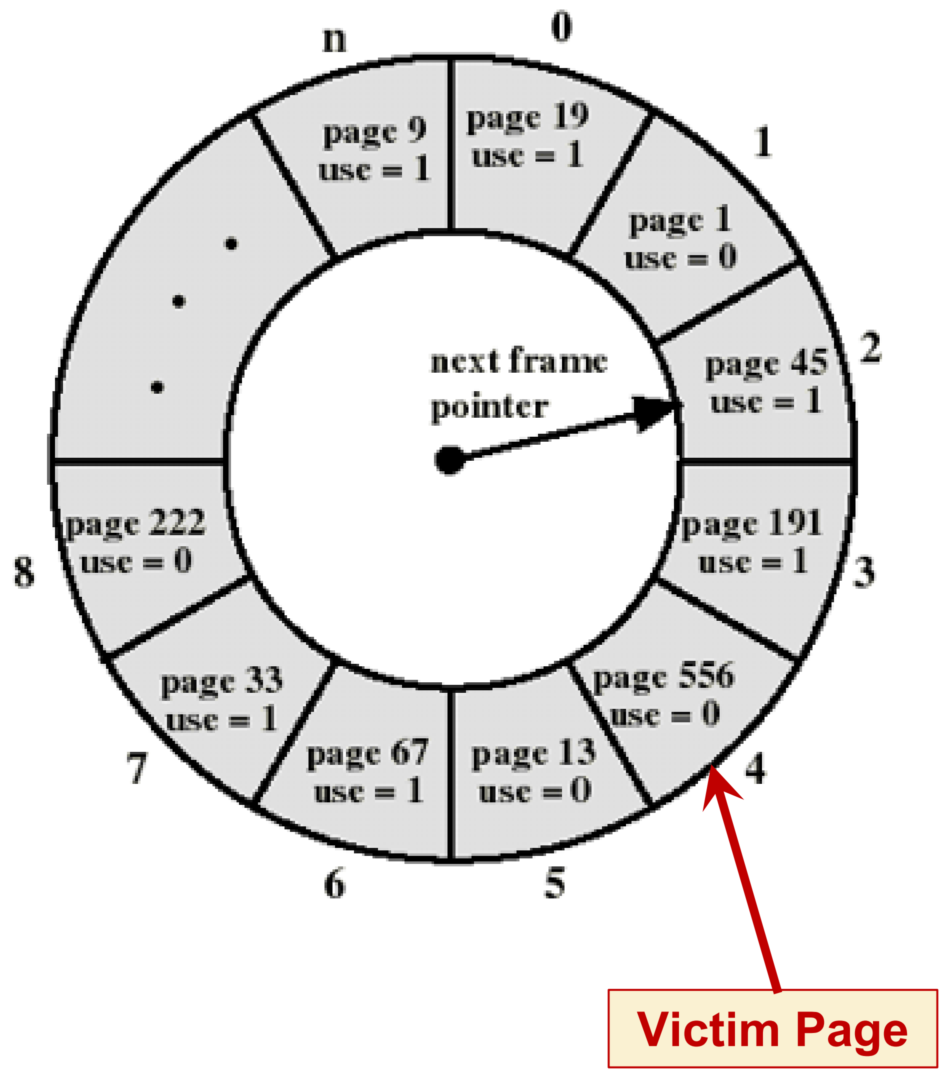
The Clock (Second Chance) Policy

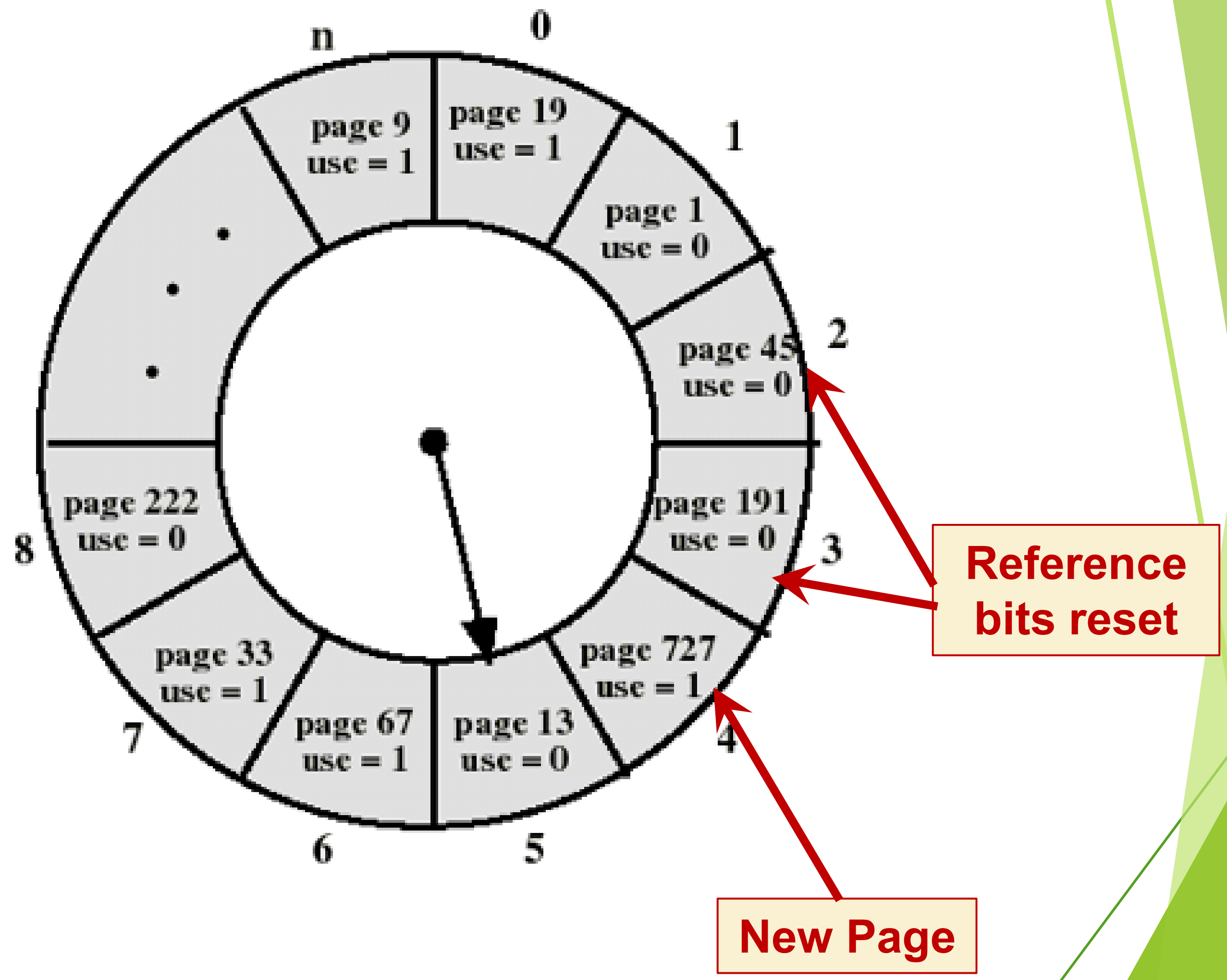
- The set of frames, potential candidates for replacement, is considered as a circular buffer.
- A reference bit for each frame is set to 1 whenever:
  - a page is first loaded into the frame.
  - the corresponding page is referenced.
- When it is time to replace a page, the first frame encountered with the reference bit set to 0 is replaced:
  - During the search for replacement, each reference bit set to 1 is reset to 0 again.
- When a page is replaced, a pointer is set to point to the next frame in buffer

Clock Page-Replacement Algorithm



Clock Algorithm: Another Example

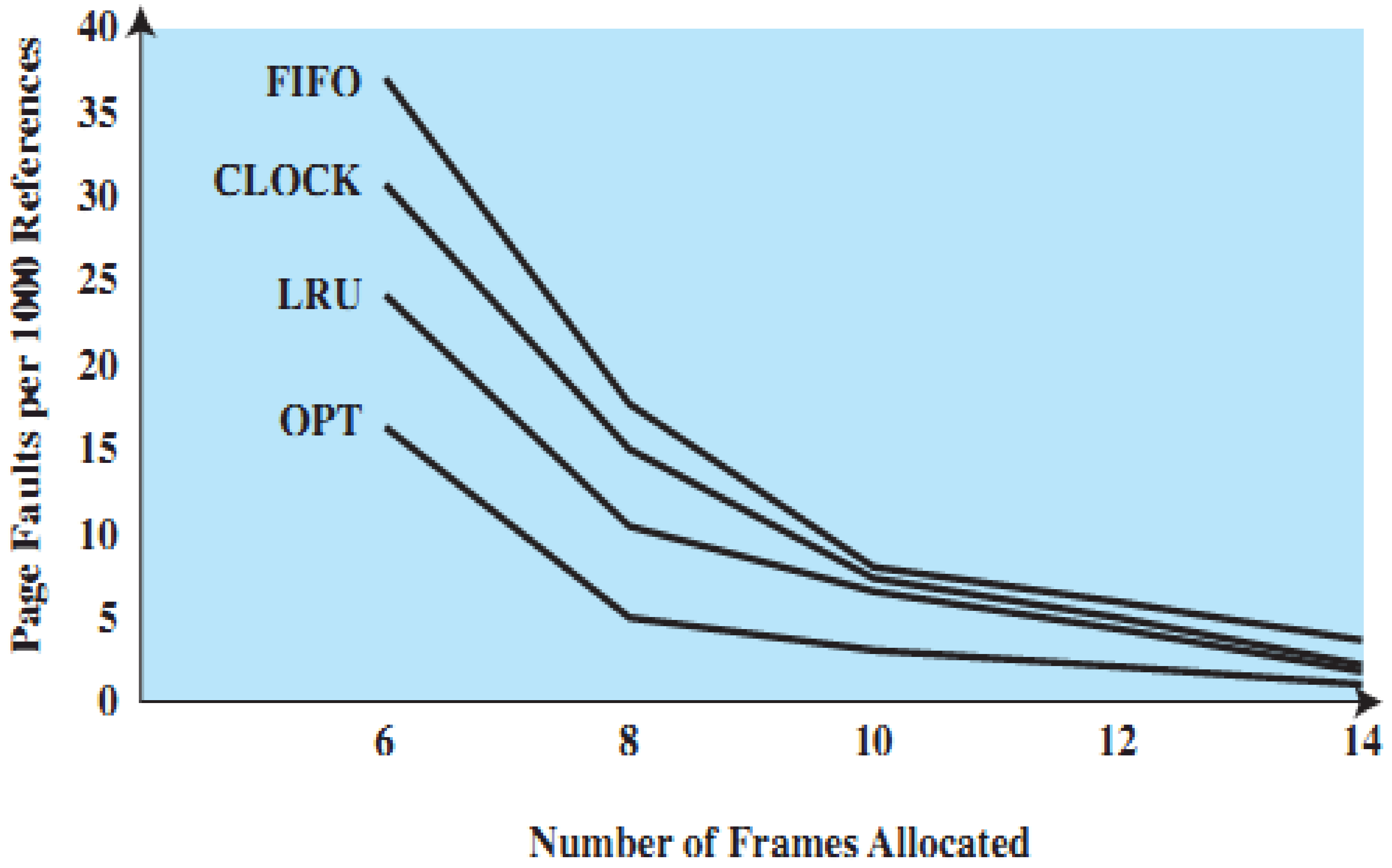




Comparison of Page Replacement Algorithms

- Experimental Setup:
  - number of frames allocated to each process is fixed
  - local page replacement
- Results
  - When few (6 to 8) frames are allocated per process, there is almost a factor of 2 of page faults between LRU and FIFO (LRU performs better)
  - LRU and FIFO performs comparatively similar when several (more than 12) frames are allocated. (more main memory is needed to support the same level of multiprogramming).
  - Numerical experiments tend to show that performance of Clock is close to that of LRU.

Fixed-Allocation, Local Page Replacement



Counting-based Algorithms

- Keep a counter of the number of references that have been made to each page.
- Two possibilities: Least/Most Frequently Used (LFU/MFU).
- LFU Algorithm: replaces page with smallest count; others were and will be used more.
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page Buffering

- Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO.
- Two lists of pointers are maintained: each entry points to a frame selected for replacement:
  - a free page list for frames that have not been modified since brought in (no need to swap out).
  - a modified page list for frames that have been modified (need to write them out).
- A frame to be replaced has a pointer added to the tail of one of the lists and the present bit is cleared in corresponding page table entry; but the page remains in the same memory frame.

Page Buffering contd.

- At each page fault the two lists are first examined to see if the needed page is still in main memory:
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the matching entry in the relevant list).
  - If it is not, then the needed page is brought in, it is placed in the frame pointed by the head of the free frame list (overwriting the page that was there); the head of the free frame list is moved to the next entry.
- The frame number in the page table entry could be used to scan the two lists, or each list entry could contain the process id and page number of the occupied frame.
- The modified list also serves to write out modified pages in cluster (rather than individually).

Cleaning Policy

- When should a modified page be written out to disk?
- Demand cleaning:
  - a page is written out only when its frame has been selected for replacement
  - but a process that suffers a page fault may have to wait for 2 page transfers (transferring in new page and transferring out dirty page)
- Pre-cleaning:
  - modified pages are written before their frames are needed so that they can be written out in batches:
  - but what if pages are modified again before being replaced.

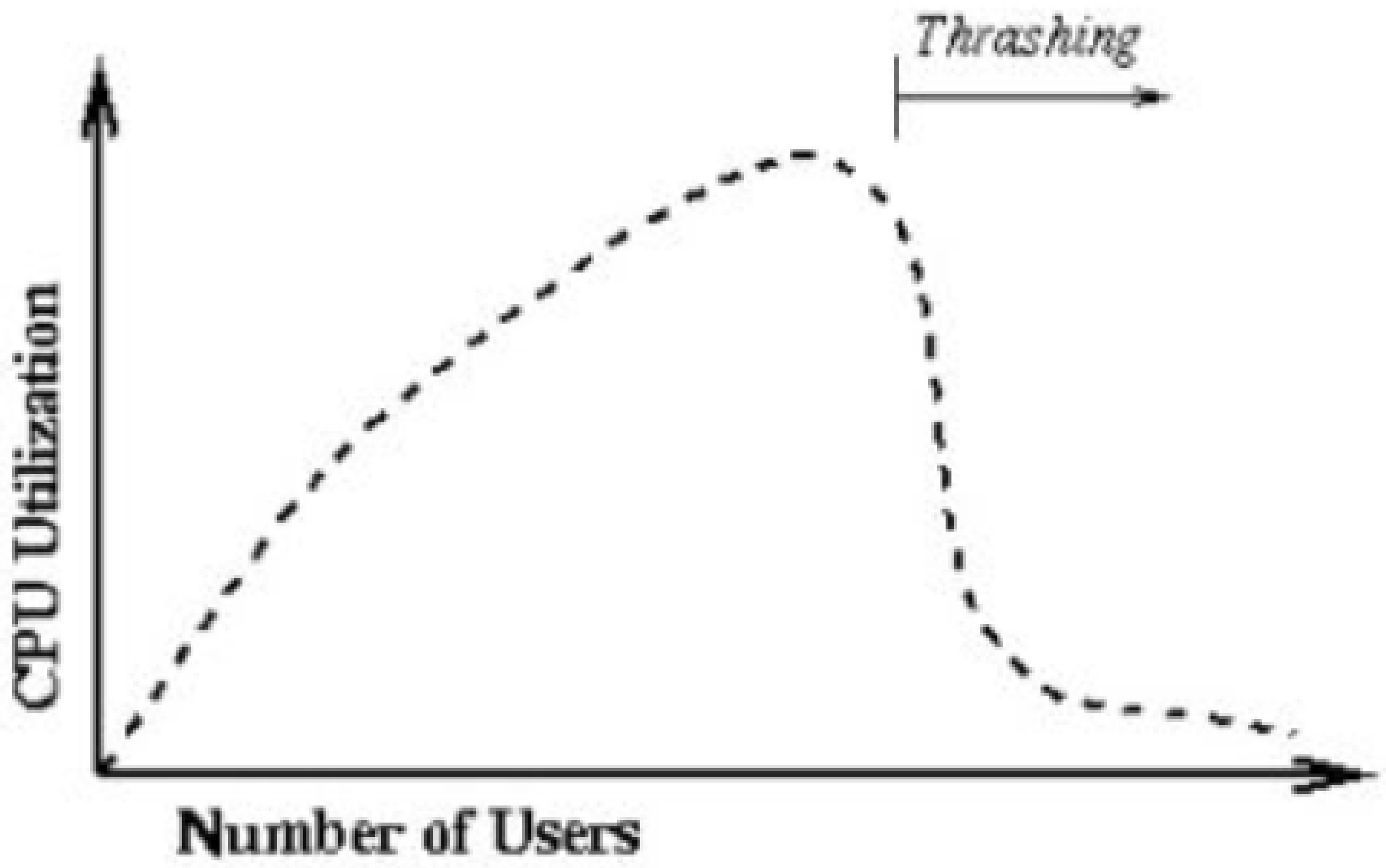
Cleaning Policy contd..

- A good compromise can be achieved with page buffering:
  - recall that pages chosen for replacement are maintained either on a free (unmodified) list or on a modified list.
  - pages on the modified list can be periodically written out in batches and moved to the free list.
  - a good compromise since:
    - **not all dirty pages are written out but only those chosen for replacement.**
    - **writing is done in batch.**

Thrashing

- Thrashing: high degree of paging activity
- Why? Processes do not have enough pages in memory
- Factors:
  - High degree of multiprogramming
  - Global page replacement can result in earlier onset of thrashing especially when degree of multiprogramming is high
  - Long term scheduler (OS) may accelerate as it may increase degree of multiprogramming (by bringing in new processes resulting in less frames available for each process) to improve CPU utilization
- Outcome: Low CPU utilization
- Local page replacement can be a mitigation but to a certain extent only

Thrashing



Thank you.