Page replacement

Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) find free frame (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created (kinit1, knit2 in xv6)
- Frames are used for allocating memory to a process
  - But may also be used for managing kernel's own data structures also (in xv6, all kernel data is statically allocated)
- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”
  - More multiprogramming -> overallocation of frames
  - Also in demand from the kernel, I/O buffers, etc
  - How much to allocate to each process? How many processes to allow?
- Page replacement – find some page(frame) in memory, but not really in use, page it out
  - Questions : terminate process? Page out process? replace the page?
  - For performance, need an algorithm which will result in minimum number of page faults
- Bad choices may result in same page being brought into memory several times

Need for Page Replacement

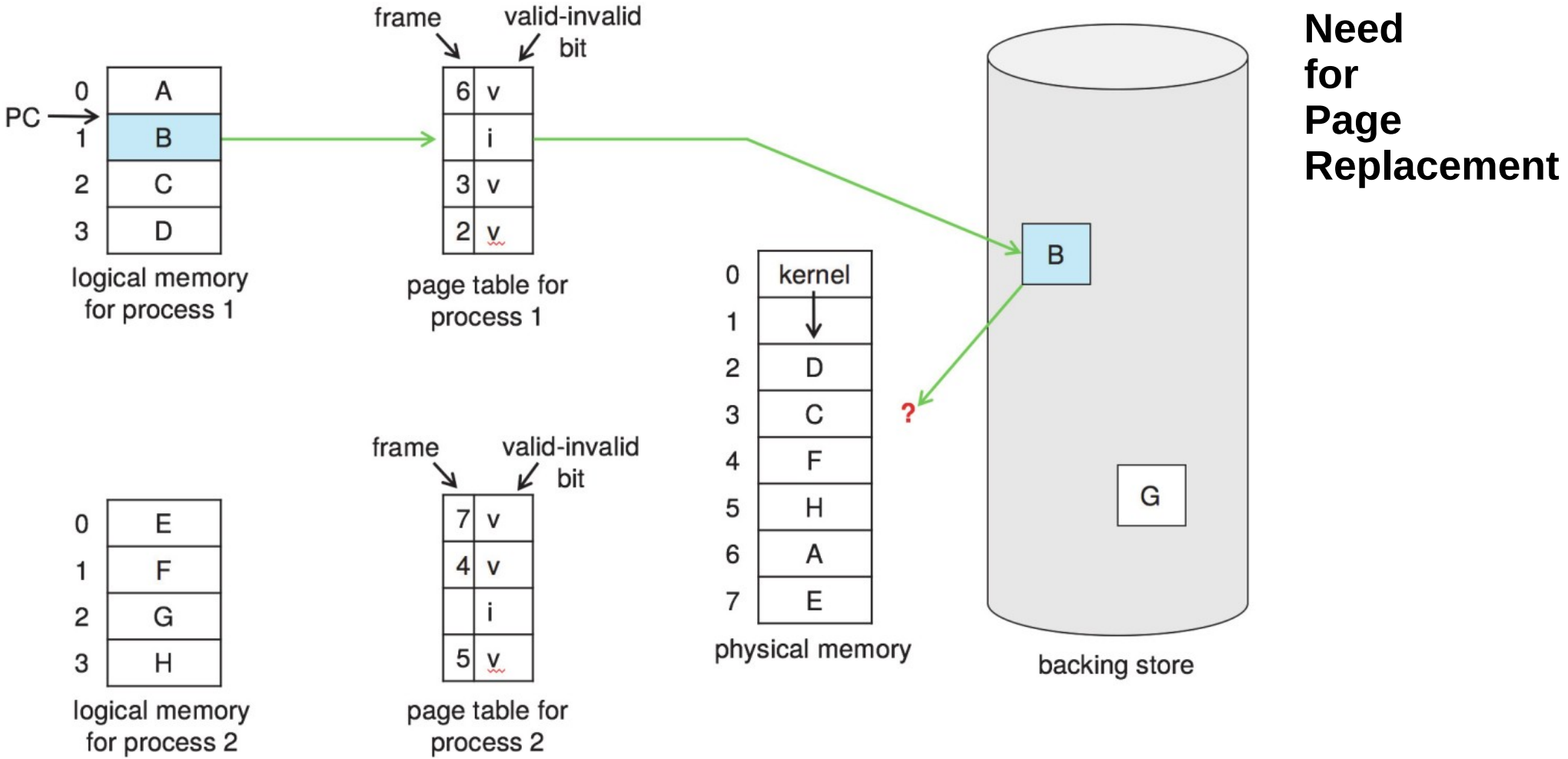


Figure 10.9 Need for page replacement.

Page replacement

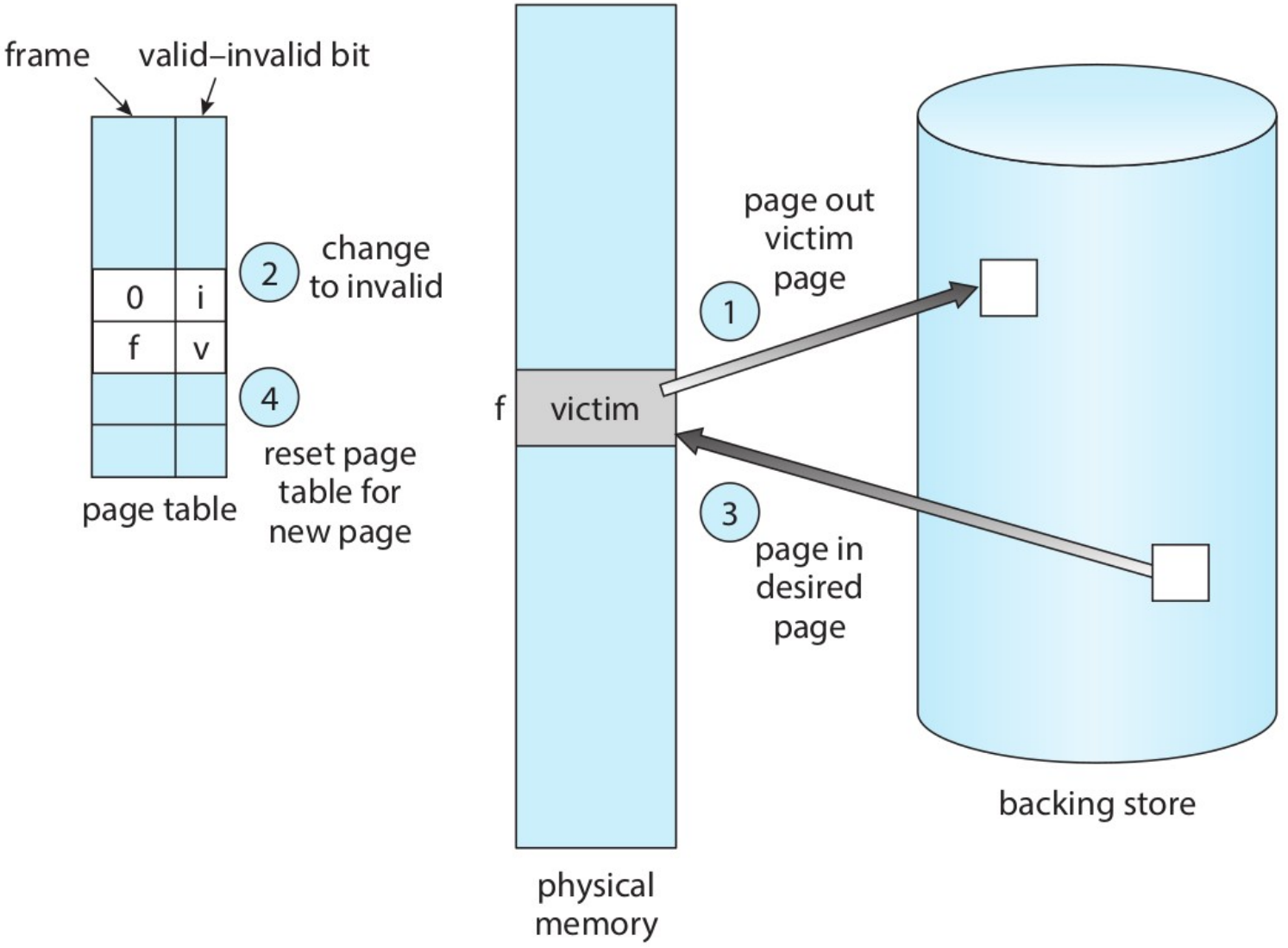
- Strategies for performance
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
  - Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

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 & write victim frame to disk if dirty
3) Bring the desired page into the free frame; update the page table of process and global frame table/list
4) Continue the process by restarting the instruction that caused the trap

- Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

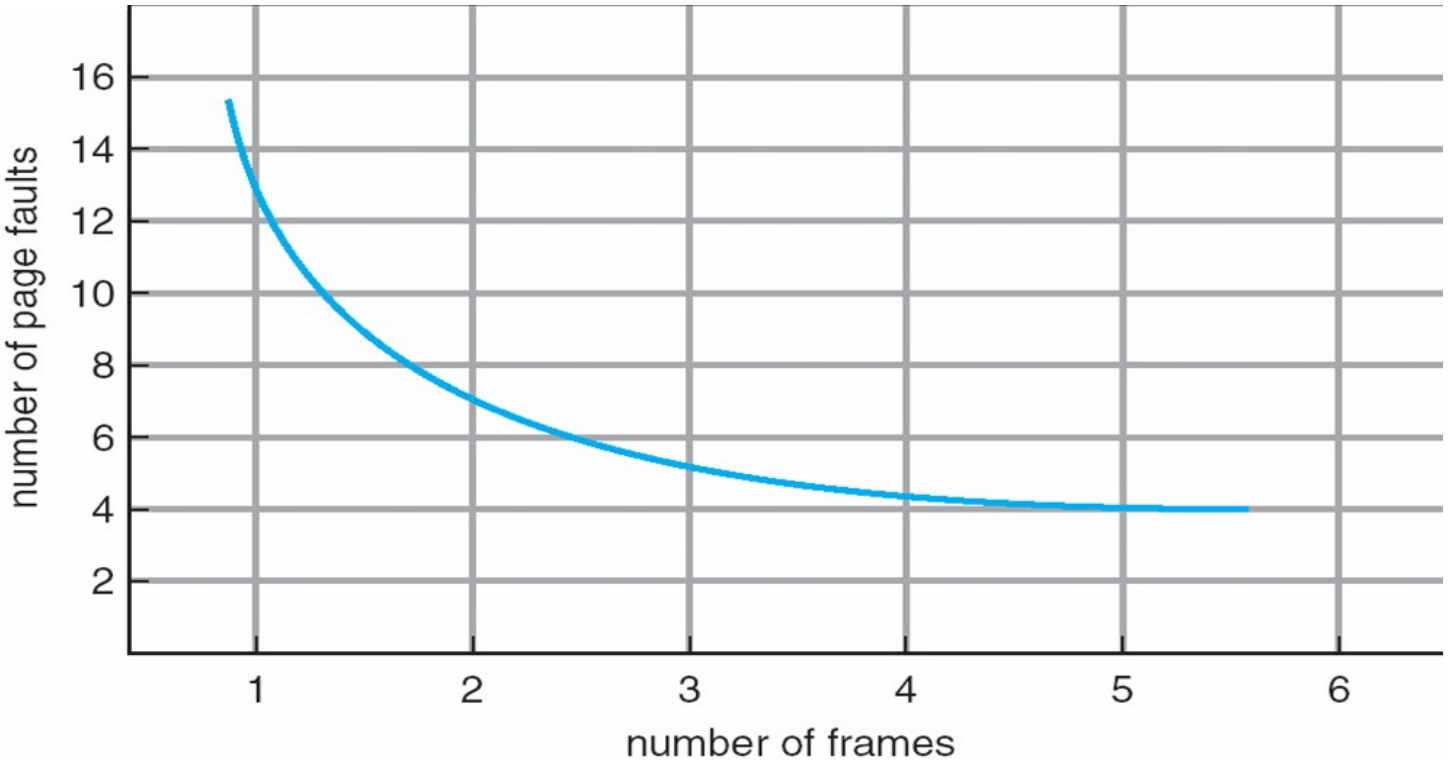


Two problems to solve

- Frame-allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm
  - Want lowest page-fault rate on both first access and re-access

Evaluating algorithm: Reference string

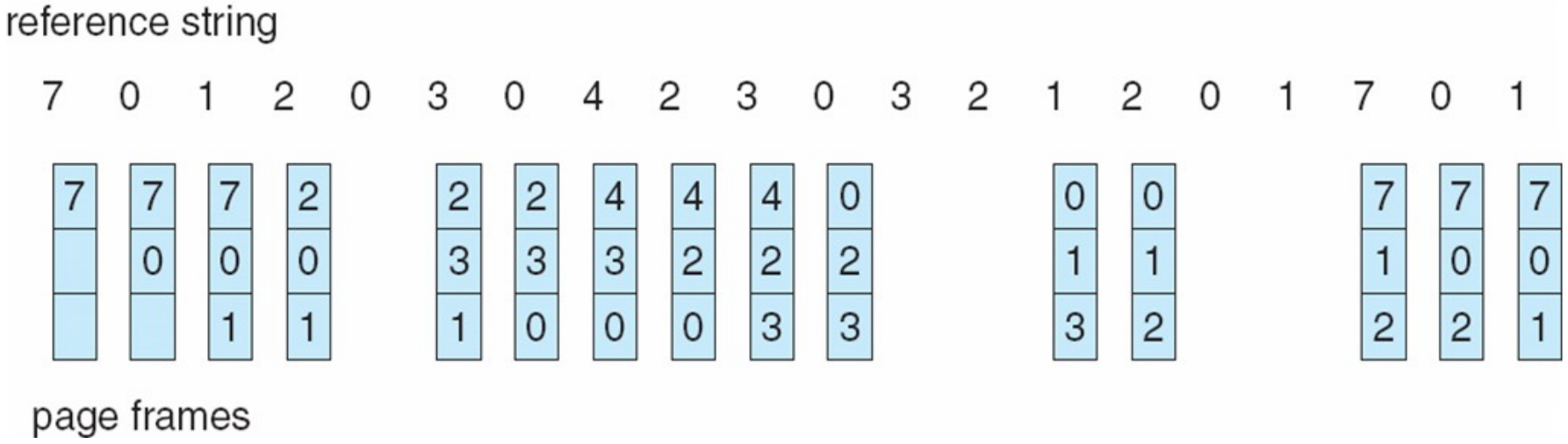
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just *page numbers*, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



An Expectation

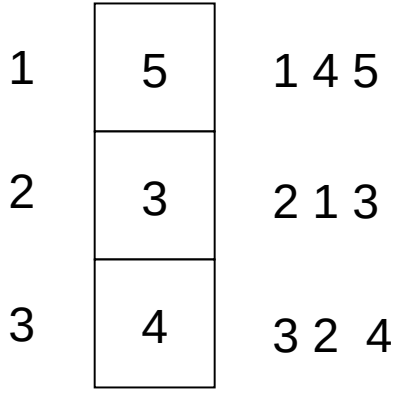
More page Frames Means less faults

FIFO Algorithm

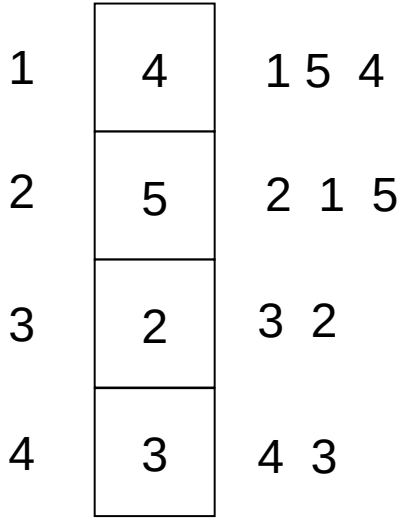


15 page faults

FIFO Algorithm



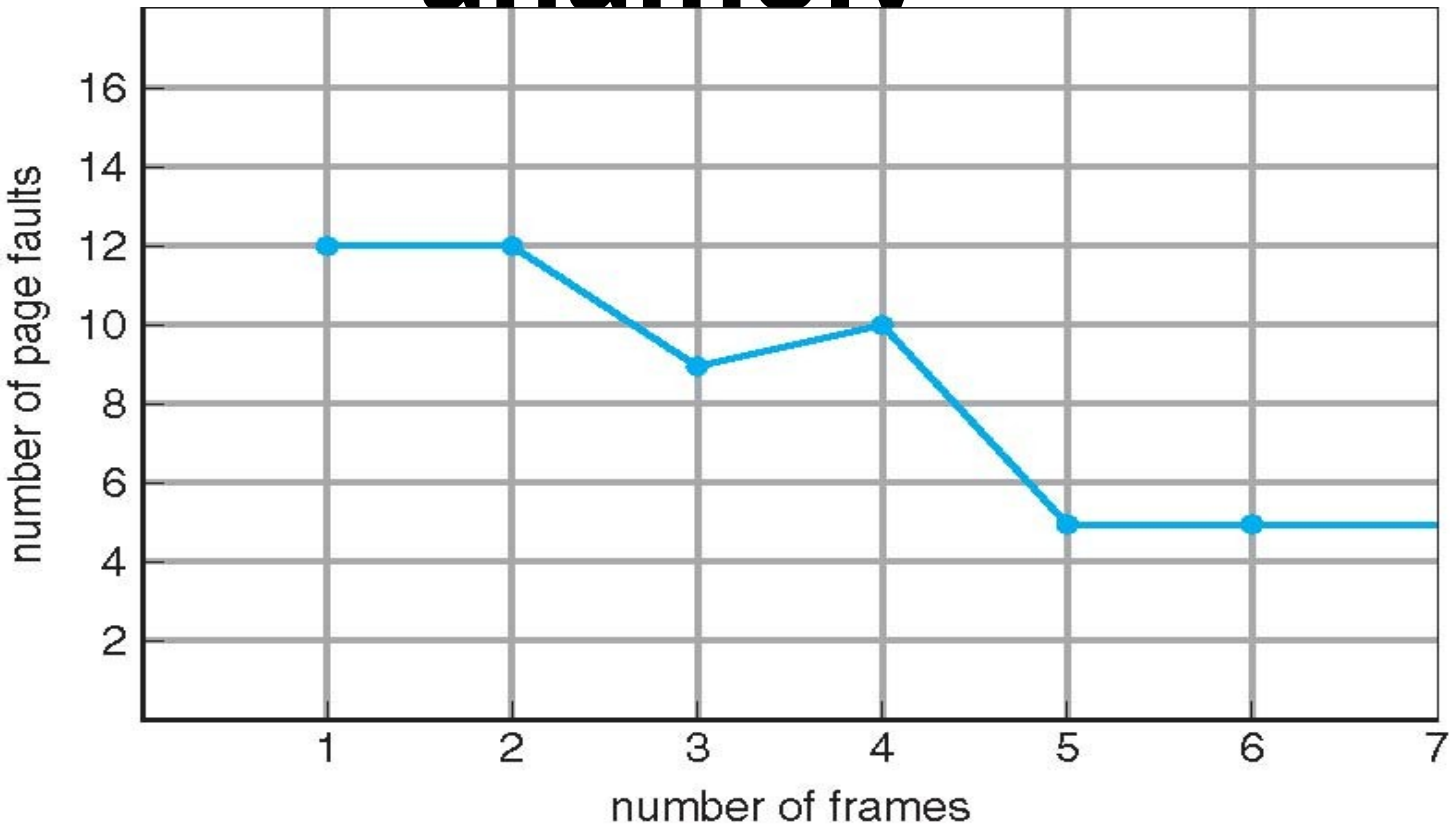
9 PF



10 PF

- **Belady's Anomaly**
- Adding more frames can cause more page faults!
  - Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

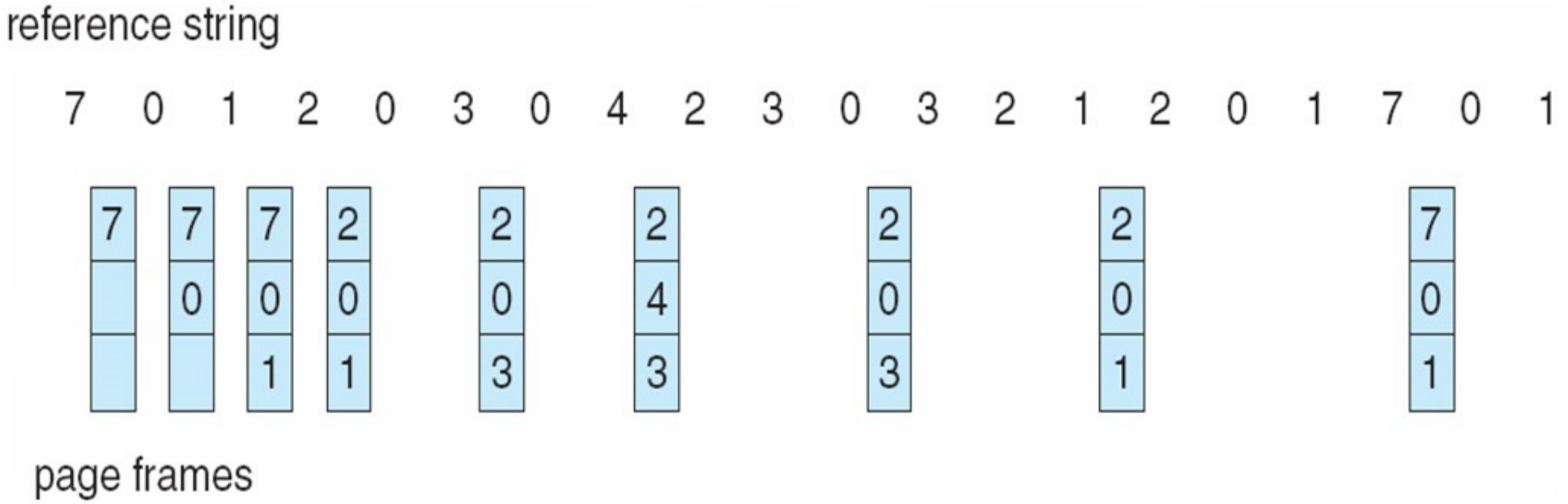
FIFO Algorithm: Balady's anamoly



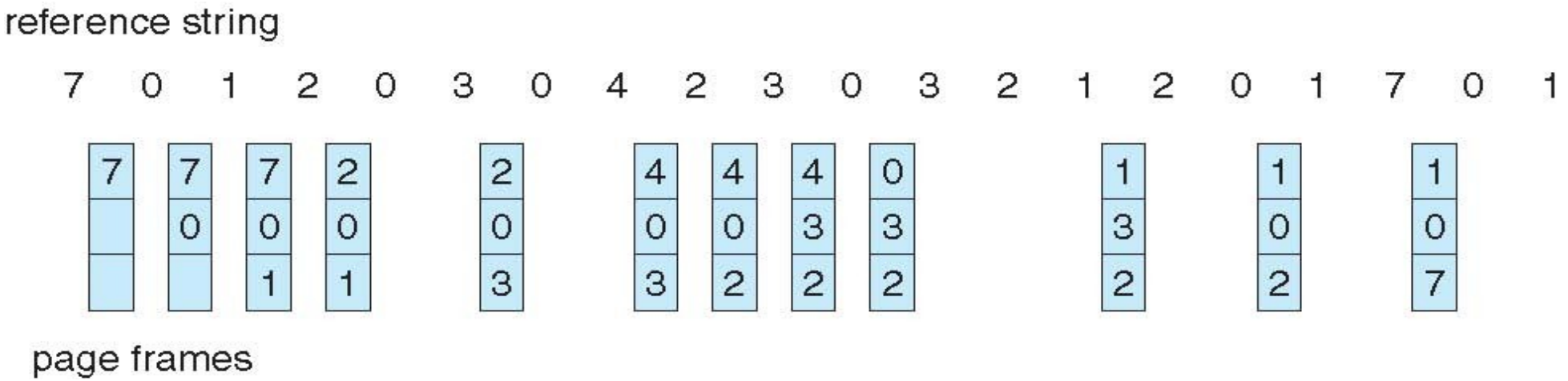
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal #replacements for the example on the next slide
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

Optimal page replacement



Least Recently Used: an approximation of optimal



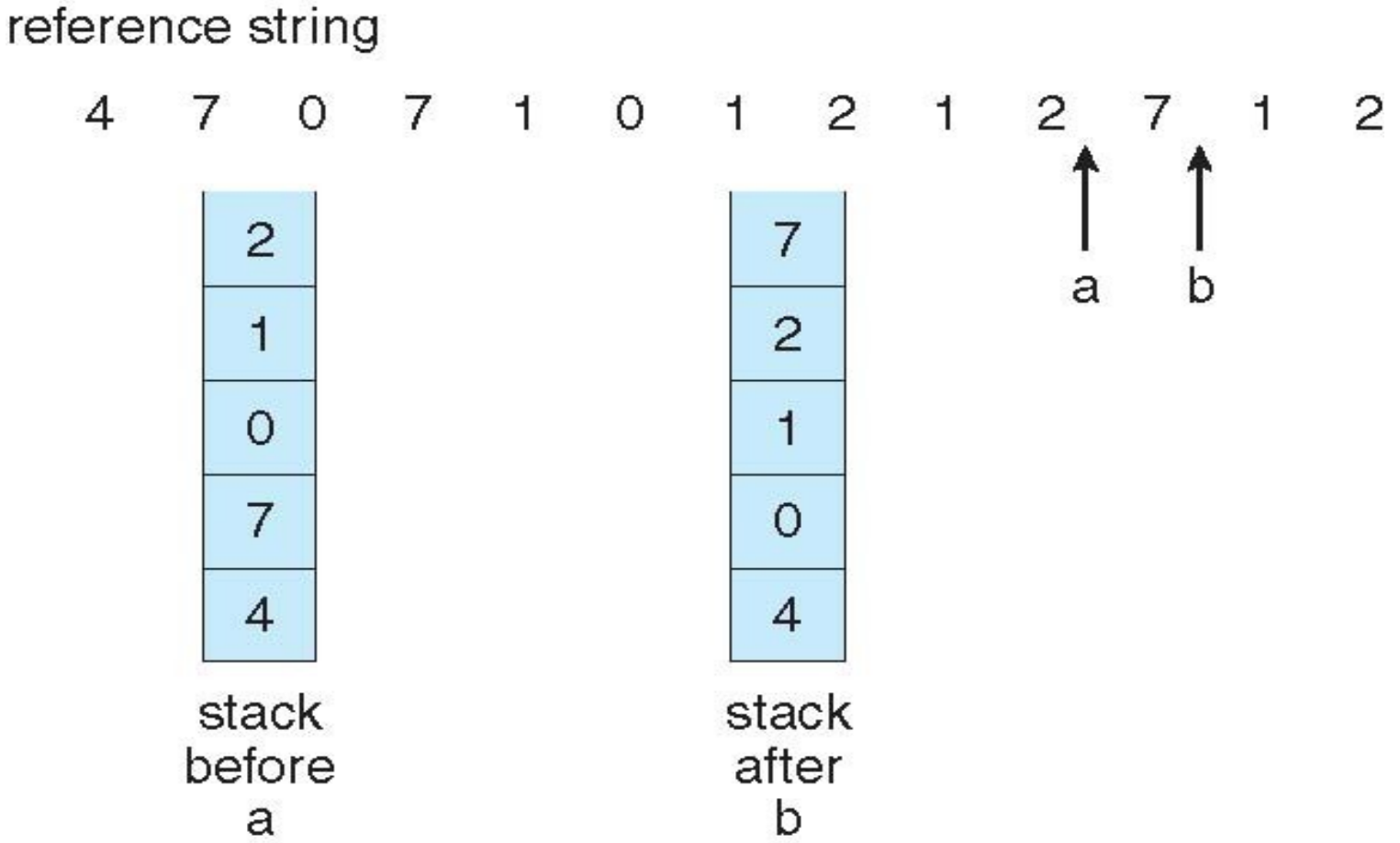
Use past knowledge rather than future
Replace page that has not been used in the most amount of time
Associate time of last use with each page
12 faults – better than FIFO but worse than OPT
Generally good algorithm and frequently used
But how to implement?

LRU: Counter implementation

- **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 find smallest value
    - Search through table needed

LRU: 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 and
  - each update more expensive
  - But no need of a search for replacement



Use of a Stack to Record The

Most Recent Page References

Stack algorithms

- An algorithm for which it can be shown that the 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
- Do not suffer from Balady’s anamoly
- For example: Optimal, LRU

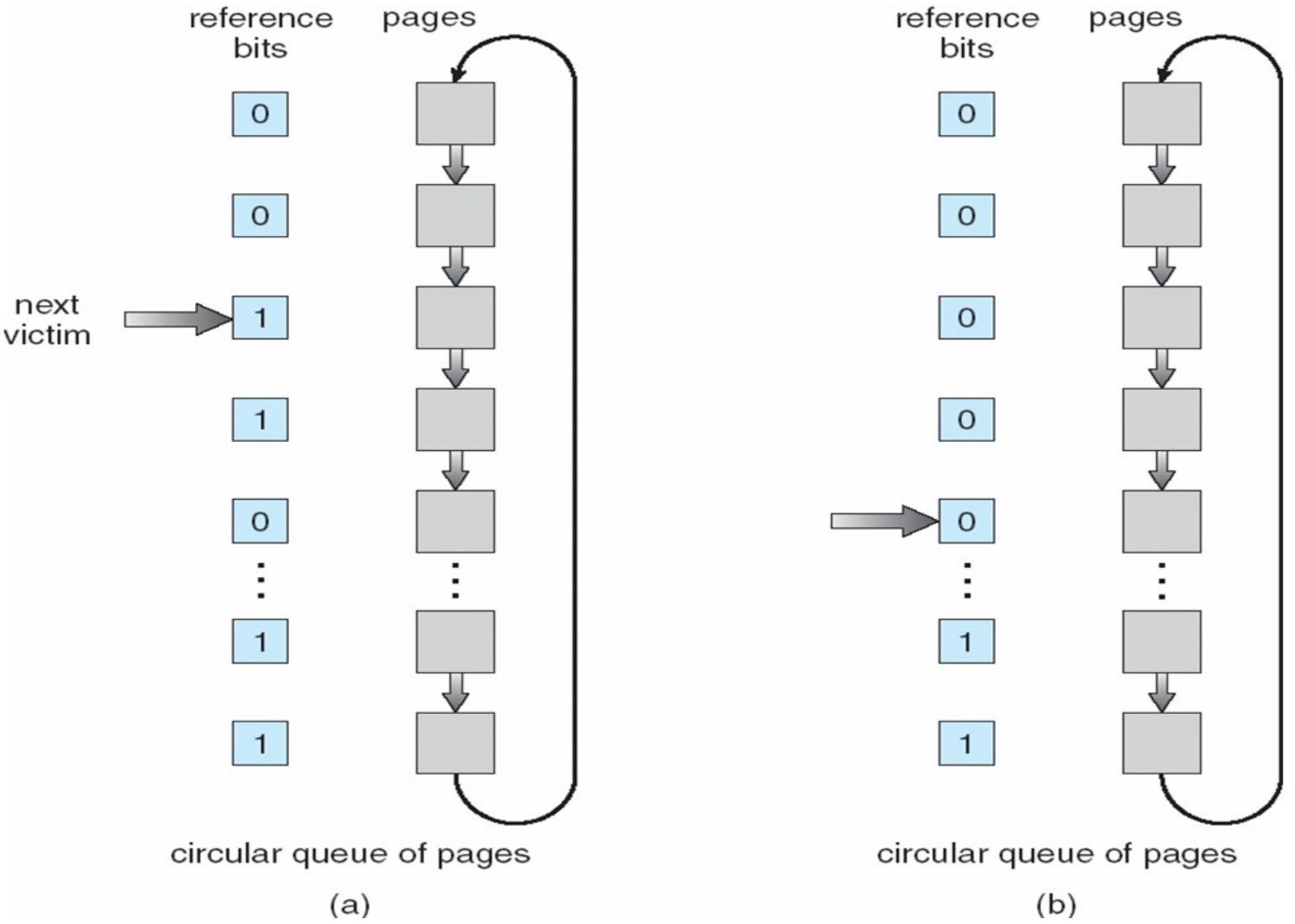
LRU: Approximation algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
  - We do not know the order, however

LRU: Approximation algorithms

- **Second-chance algorithm**
  - FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.
- **An implementation of second-chance: Clock replacement**
  - If page to be replaced has
  - Reference bit = 0 -> replace it
  - reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **LFU Algorithm**: replaces page with smallest count
- **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 algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
  - Fault + page not in memory
- Minor fault
  - Fault, but page is in memory
  - For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run

```
$ ps -eo min_flt,maj_flt,cmd
```

Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc

Allocation of frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Maximum of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$s_i \equiv \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

Priority Allocation of frames

- Use a proportional allocation scheme using priorities rather than size
- If process Pi generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global Vs Local allocation

- **Global replacement –** process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement –** each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory