Virtual Memory

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.

Dec. 2020

Virtual memory

Separation of user logical memory from physical memory

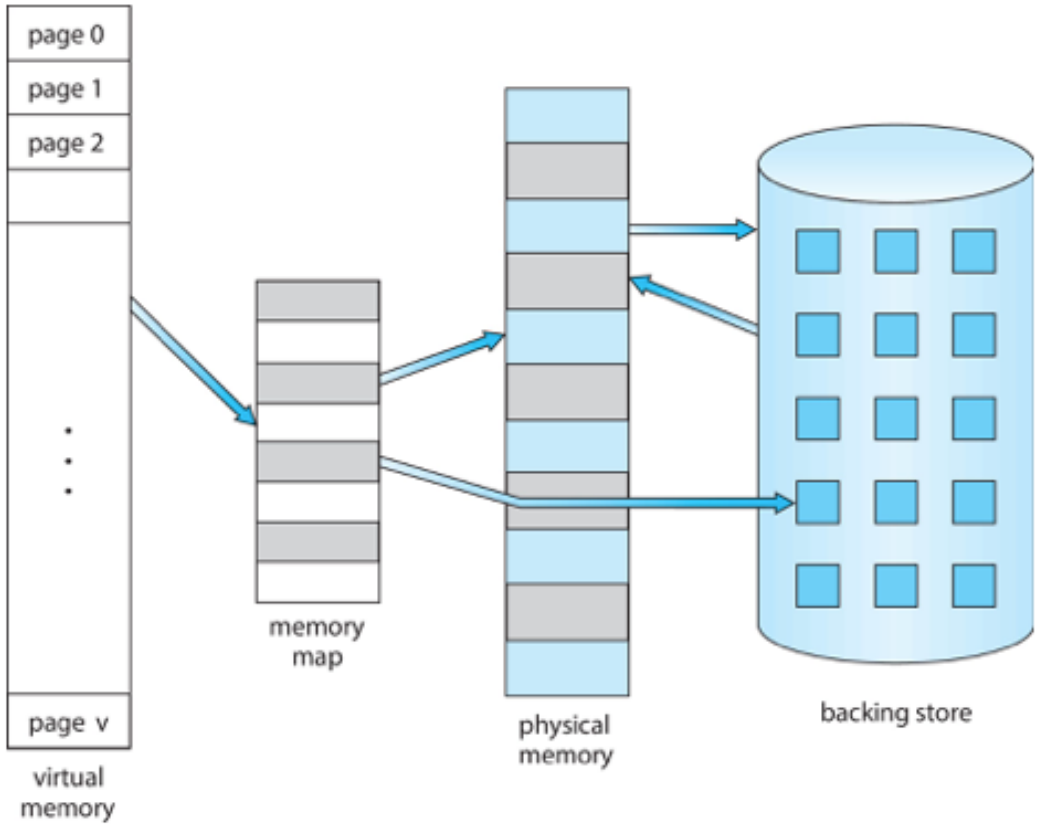
- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes

Virtual address space - logical view of how process is stored in memory

Virtual memory can be implemented via:

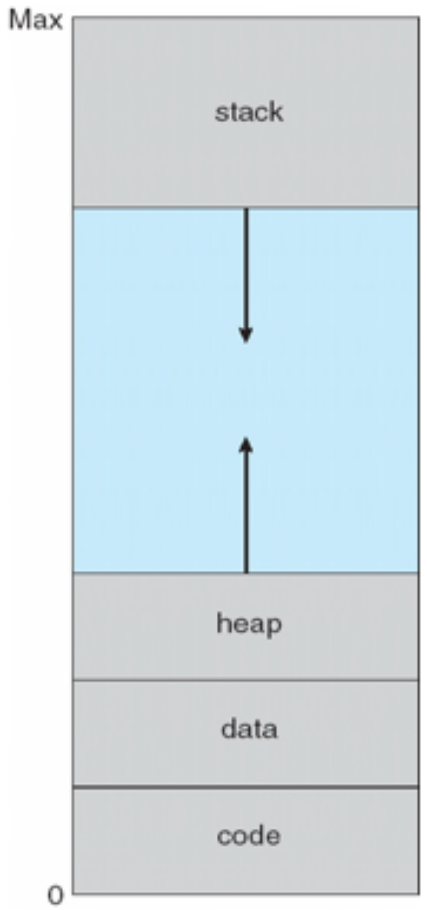
- Demand paging
- Demand segmentation

Virtual Memory That is Larger Than Physical Memory

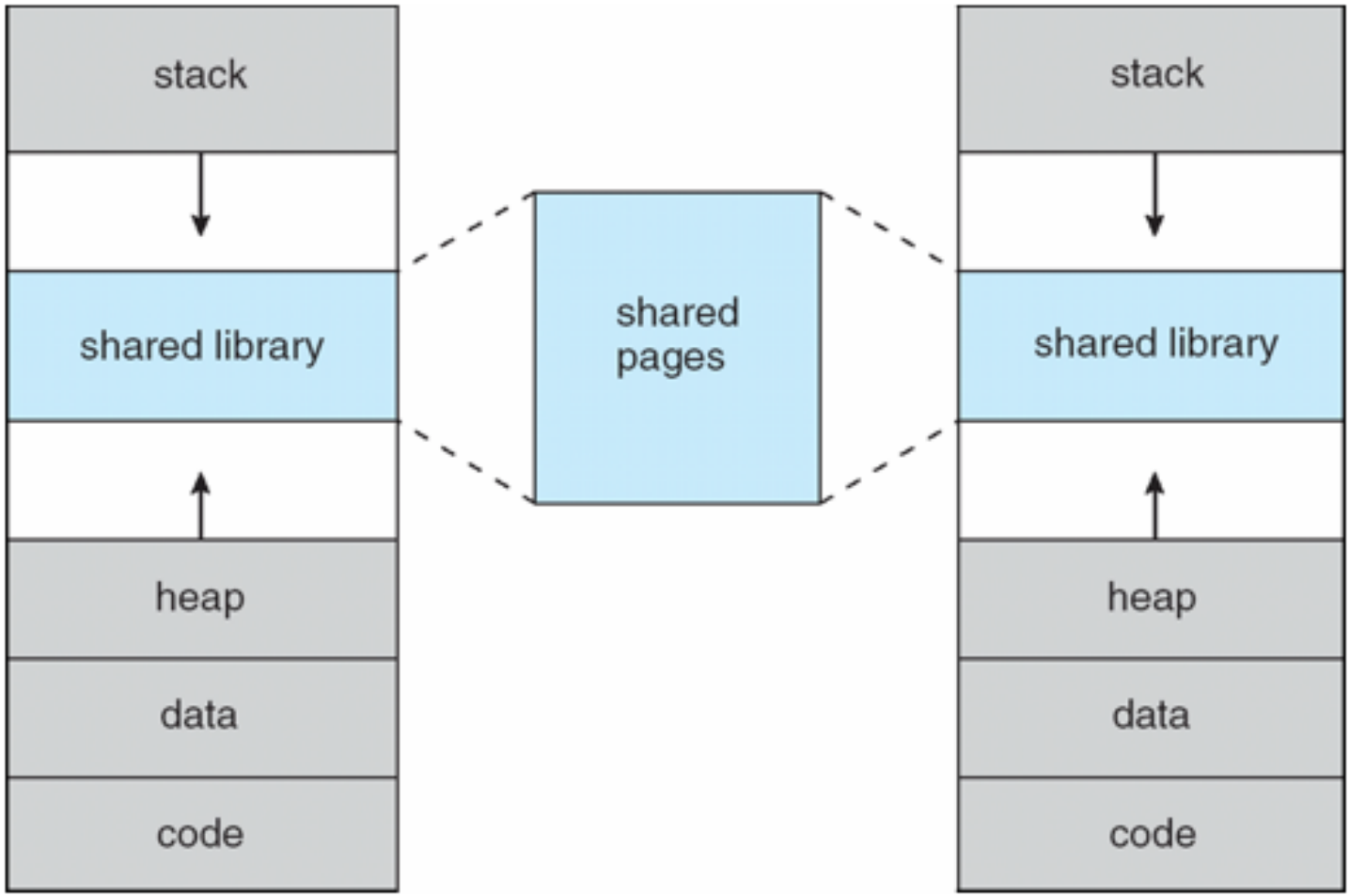


Virtual-address Space

- Logical address space for stack to start at Max and grow "down" while heap grows "up"
- Enables sparse address spaces with holes left for growth, dynamically linked libraries, stc.
- System libraries shared via mapping into virtual address space



Shared Library Using VM



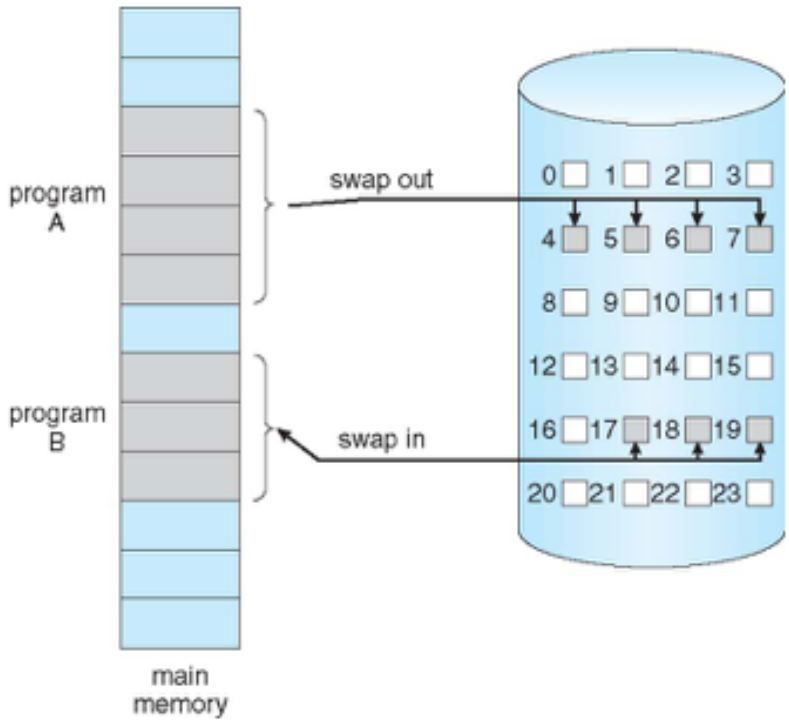
Demand Paging

Could bring entire process into memory at load time

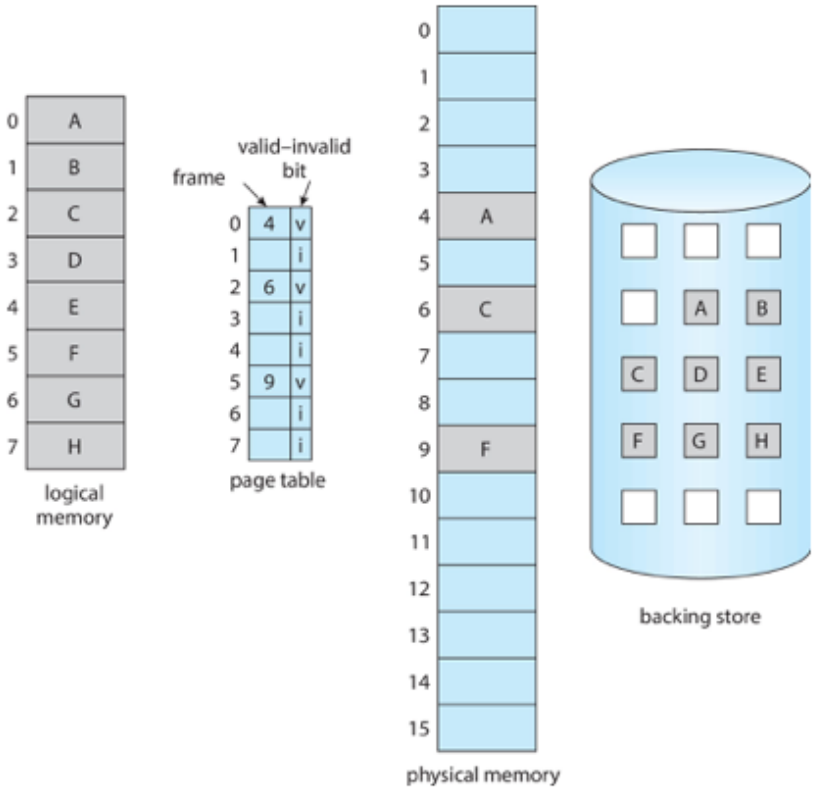
Or bring a page into memory only when it is needed

If page is needed reference to it

- invalid reference -> abort
- not-in-memory -> bring to memory



Page Table When Some Pages Are Not in Main Memory



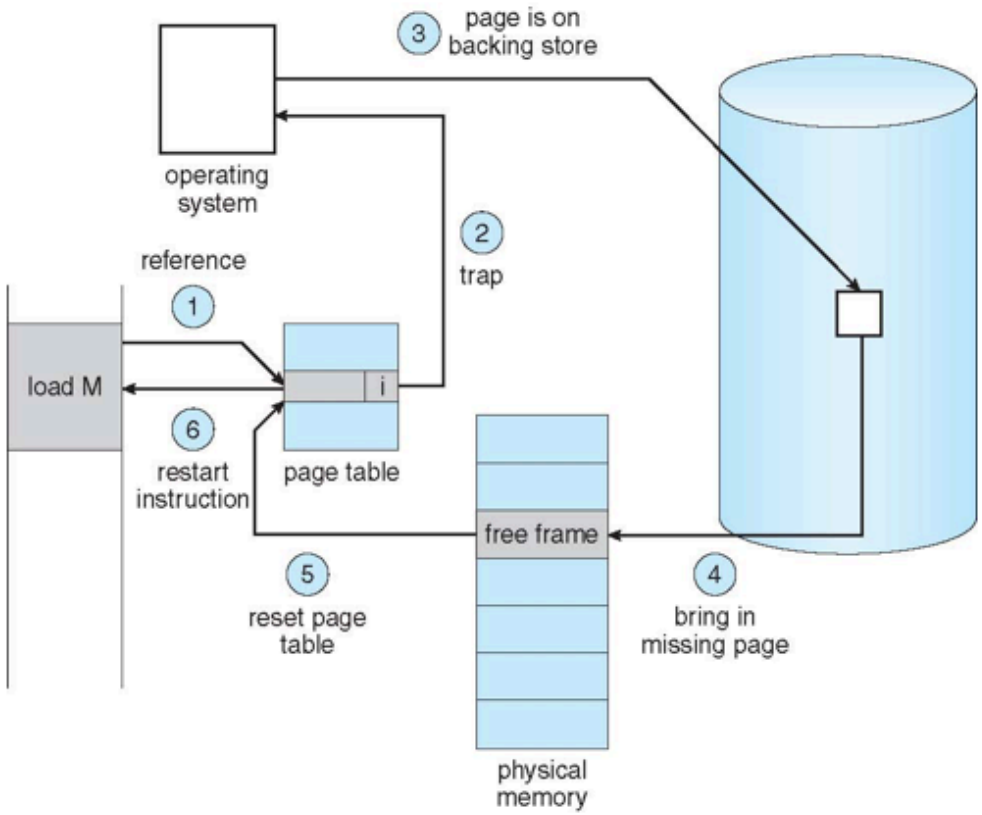
With each page table entry a valid-invalid bit is associated

During MMU address translation, if valid-invalid bit in page table entry is invalid -> page fault exception

Demand Paging

- Get the bad virtual address that caused the page fault
- Allocate a physical page
- Call the file system to copy the page from the disk to the physical page in the memory.
- Update the page table
- Re-execute the instruction

Steps in Handling a Page Fault



Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.

Most operating systems maintain a free-frame list - a pool of free frames for satisfying such requests

Zero-fill-on-demand - the content of the frames zeroed-out before being allocated.

When a system starts up, all available memory is placed on the free-frame list.

Performance of Demand Paging

Three major activities: (1) Service the interrupt (2) Read the page (3) Restart the process

Page Fault Rate $0 \leq p \leq 1$

Effective Access Time (EAT)
EAT = (1-p) $\times$ memory access +
p $\times$ (page fault overhead + swap page out + swap page in)

Demand Paging Example

For memory access time = 200ns, and average page-fault service time 8ms.
EAT = (1-p)$\times$ 200ns + p $\times$ 8000000ns
For p = 0, E = 200ns
For p=1/1000, EAT=8μs => slowdown by a factor of 40!

Basic Page Replacement

Find the location of the desired page on disk.

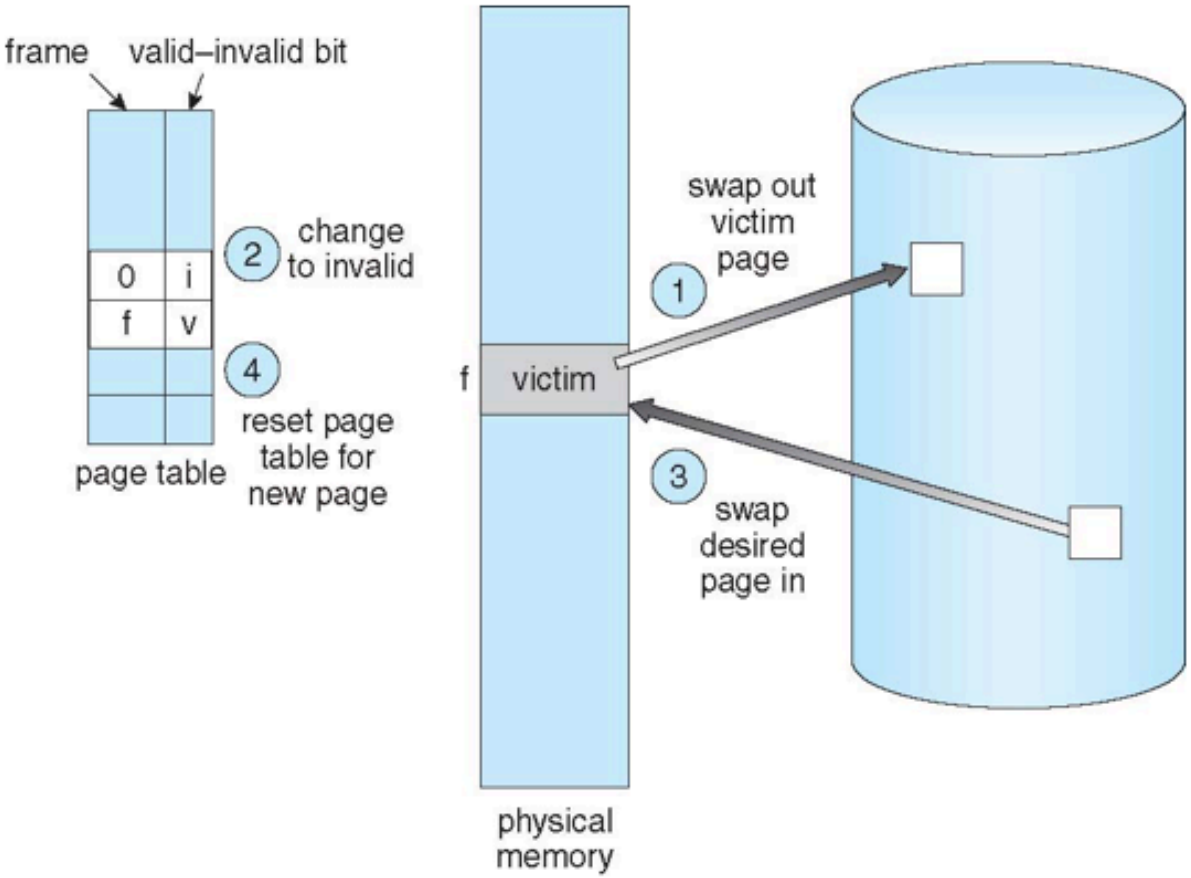
Find a free frame

- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm to select a victim frame and write victim frame to disk.

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

Restart the instruction that caused the trap

Page Replacement



Page and Frame Replacement Algorithms

FIFO (First-In-First-Out)

OPT (Optimal Algorithm)

LRU (Least Recently Used)

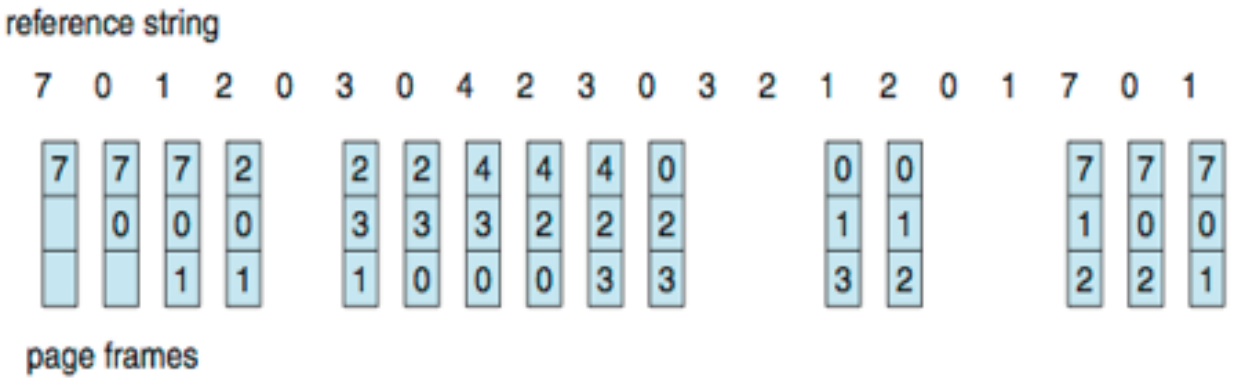
LFU (Least Frequently Used)

MFU (Most Frequently Used)

First-In-First-Out (FIFO) Algorithm

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

3 frames (3 pages can be in memory at a time per process)

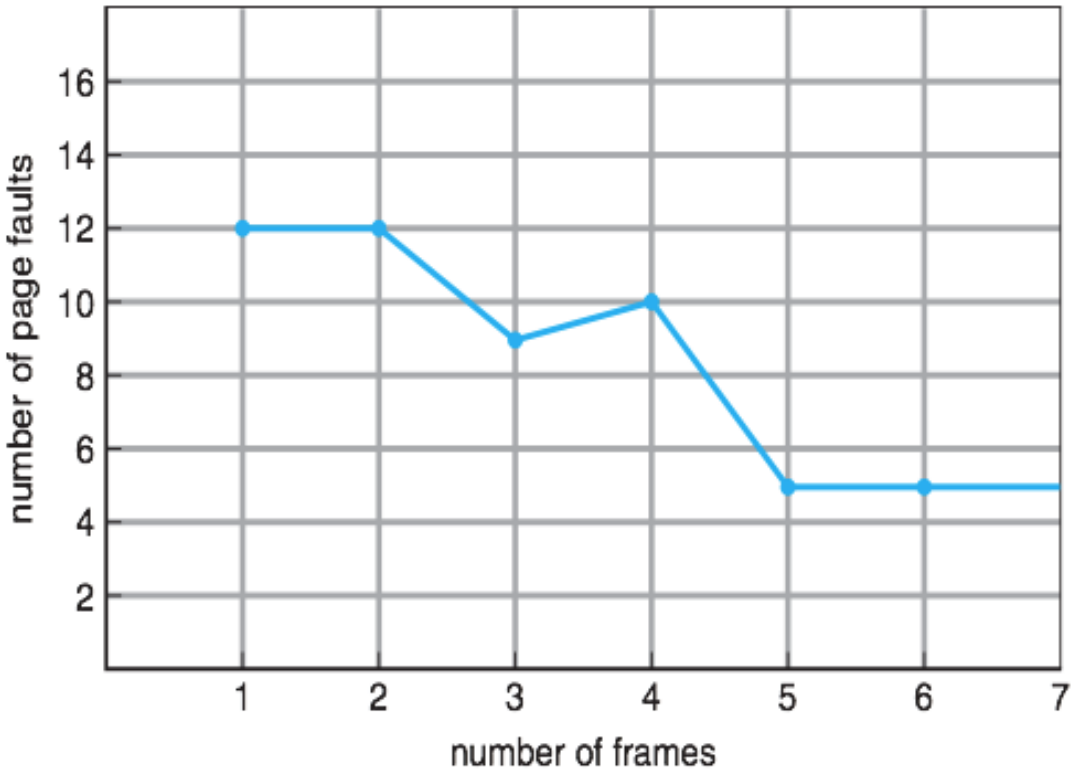


15 page faults!

Adding more frames can cause more page faults - Belady's Anomaly.

FIFO - Belady's Anomaly

Thus, a bad replacement choice increases the page-fault rate and slows process execution.



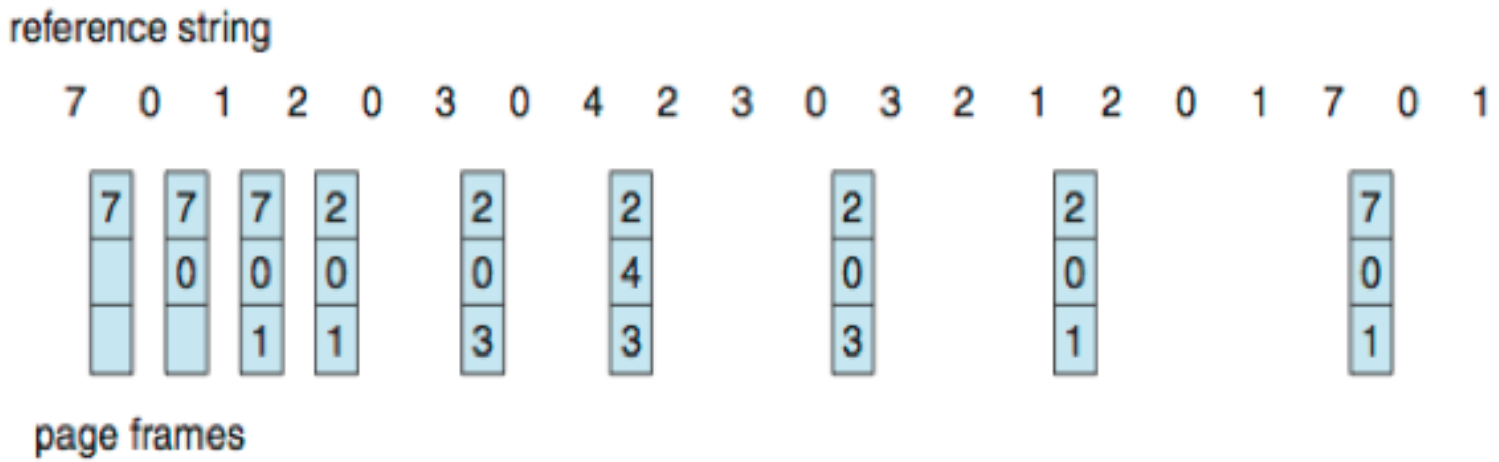
Consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

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

How do you know this? We can't read the future!

Used for measuring how well your algorithm performs

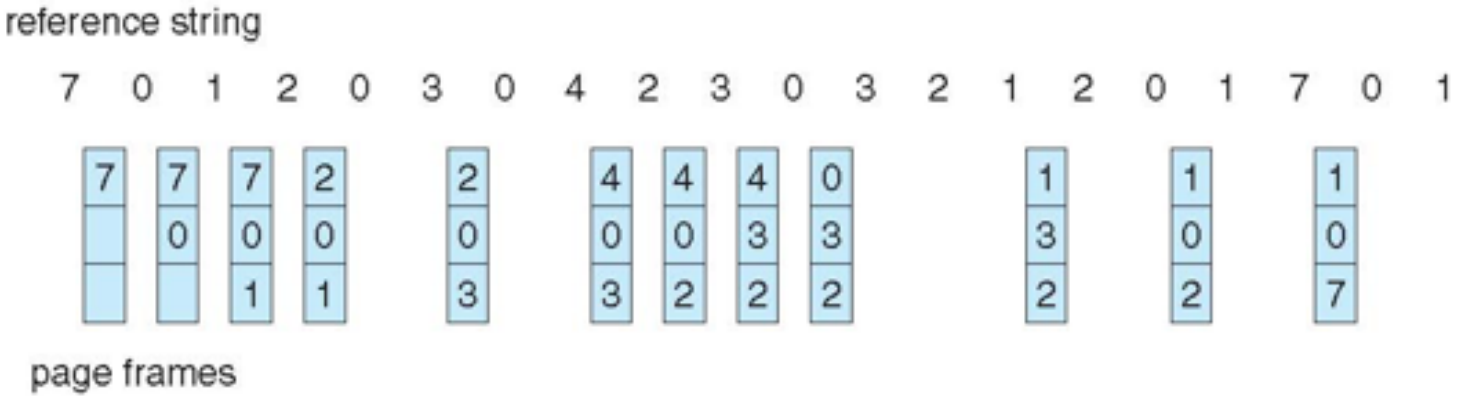


Only 9 page faults!

Least Recently Used (LRU) Algorithm

Replace page that has not been used in the most amount of time

Associate time of last use with each page



12 page faults!

Better than FIFO but worse than OPT

Generally good algorithm and frequently used

LRU Algorithm (Cont.)

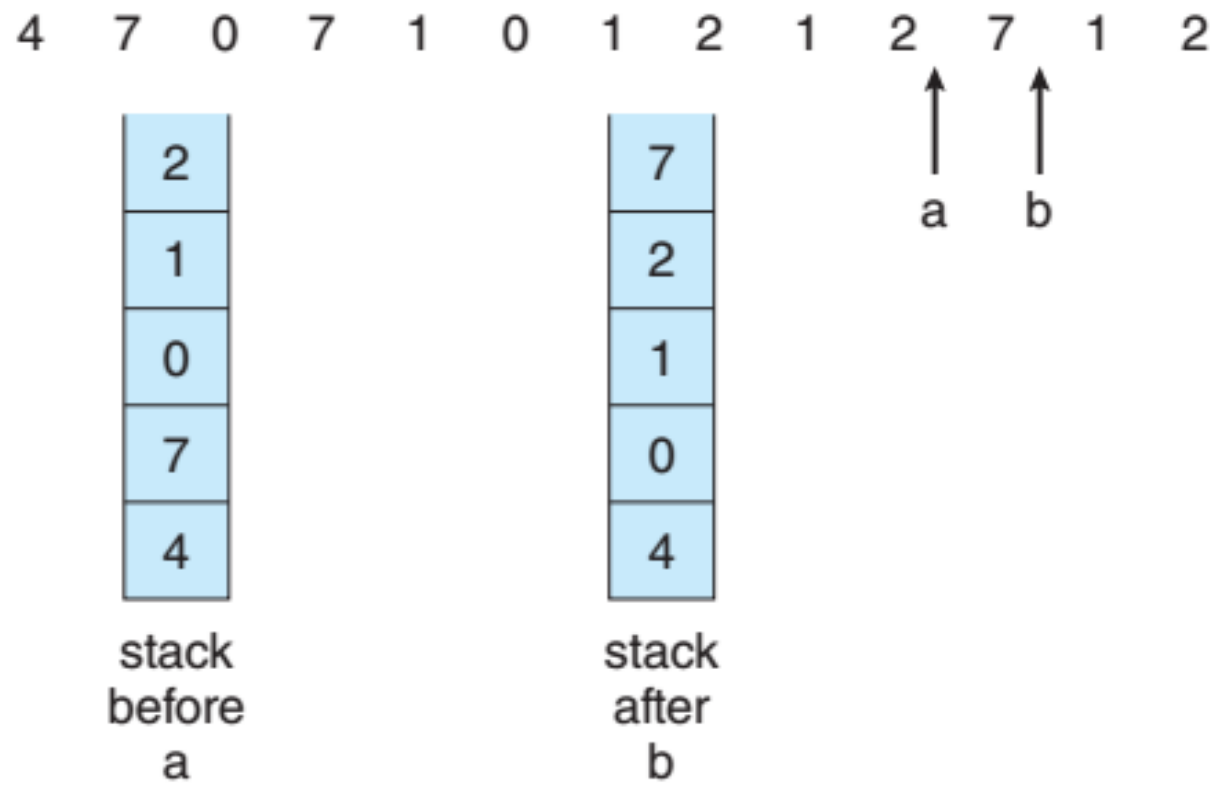
Counter implementation

- Every page entry has a counter. Every time page is referenced, copy the clock into the counter. In this way, we always have the "time" of the last reference to each page.
- When a page needs to be changed, look at the counters to find smallest value.

Stack implementation

- Keep a stack of page numbers in a double link form.
- Whenever a page is referenced, it is removed from the stack and put on the top.

Stack Implementation



The least recently used page is always at the bottom

LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly

Approximation of LRU - Clock Algorithm

LRU needs special hardware

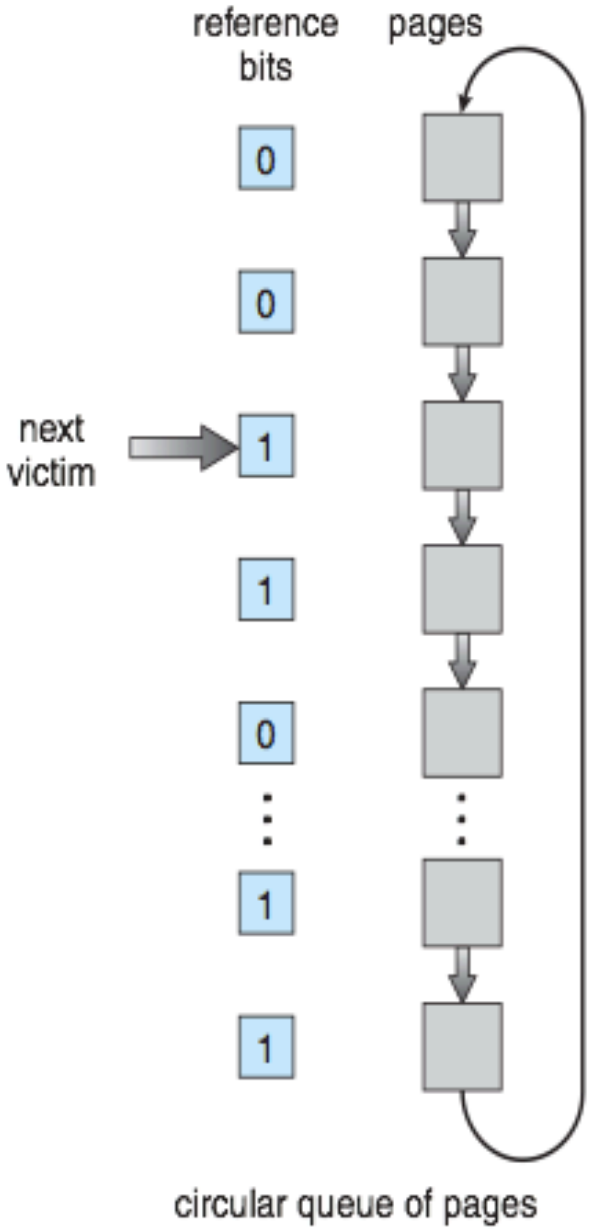
Reference bit algorithm

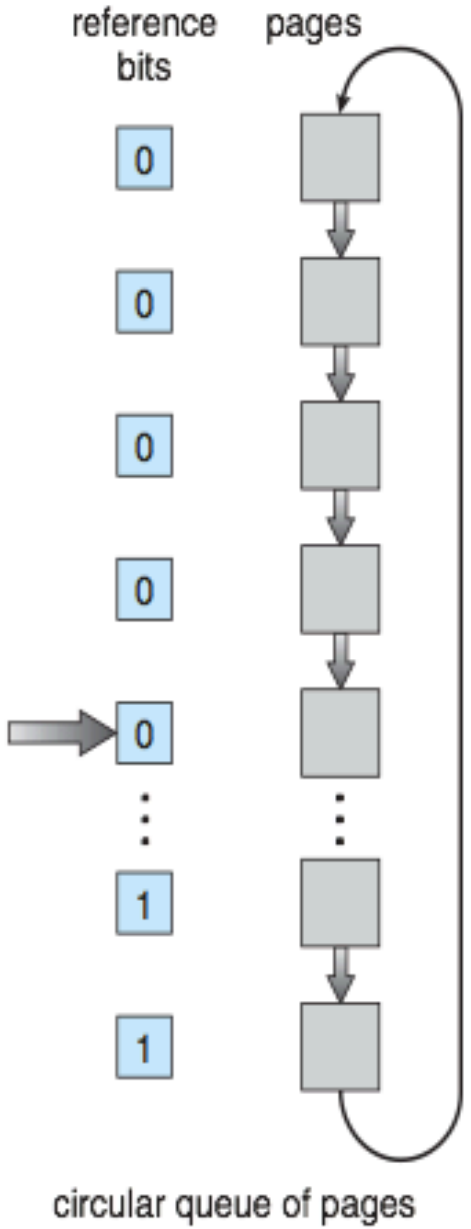
- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods.

Second-chance algorithm

- If page to be replaced has reference bit = 0 -> replace it
- If 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

Least Frequently Used (LFU) Algorithm: replaces page with smallest count

Most Frequently Used (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

- Read page into free frame and select victim to evict and add to free pool

Possibly, keep list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to secondary storage.

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

All of these algorithms have OS guessing about future page access

Thrashing and Working Set Model

Thrashing: a process is spending more time paging than executing. Memory is as slow as disk.

Working set model

- Basis: locality
- Working set window (WSW) (a time frame)
- Working set (WS) (a set of pages referenced in the time frame)
- Working set size (WSS) (number of pages in WS)

Page replacement can be determined by working set model.

Working set model can prevent thrashing!

Thrashing and Working Set

How does the system detect thrashing?

The system can detect thrashing by evaluating the level of CPU utilization compared with the level of multiprogramming.

Is it possible for a process to have two working sets, one representing data and another representing code?

Yes. In fact, many processors provide two TLBs for this very reason. As an example, the code being accessed by a process may retain the same working set (WS) for a long time. However, the data the code accesses may change, thus reflecting a change in the WS for data accesses.

Thank you !

Operating Systems are among the most complex pieces of software ever developed !