

# Virtual Memory

**CSCI 315 Operating Systems Design**  
Department of Computer Science

**Notice:** The slides for this lecture were based on those *Operating Systems Concepts, 9th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.

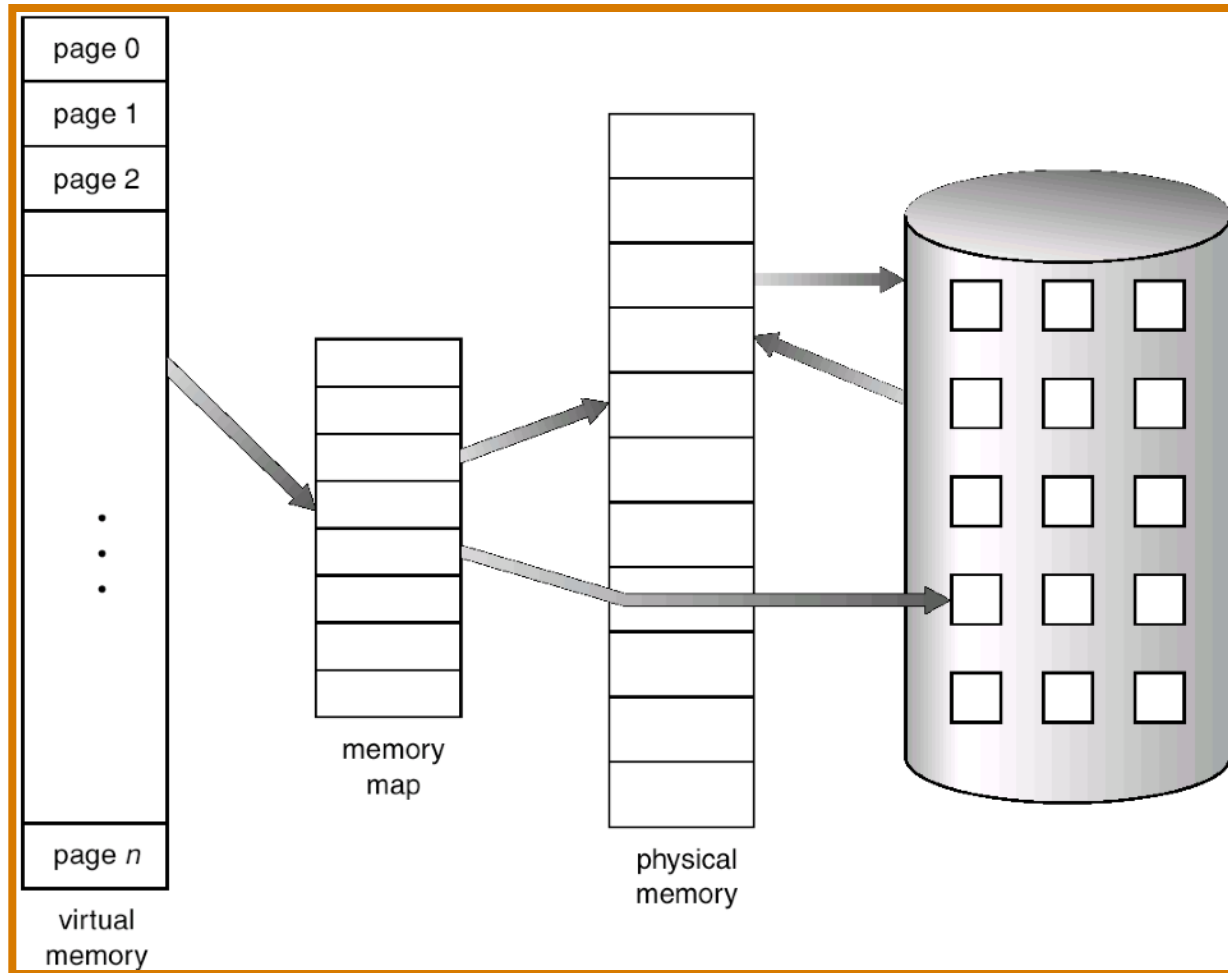


# Virtual Memory

- **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.
  - Allows for more efficient process creation.
- 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.
- **When a page is referenced:**
  - if invalid reference  $\Rightarrow$  **abort with error message.**
  - if not-in-memory  $\Rightarrow$  **bring to memory.**
  - if already in memory  $\Rightarrow$  access the location

# Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (1  $\Rightarrow$  in-memory, 0  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to 0 on all entries.
- Example of a page table snapshot.

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

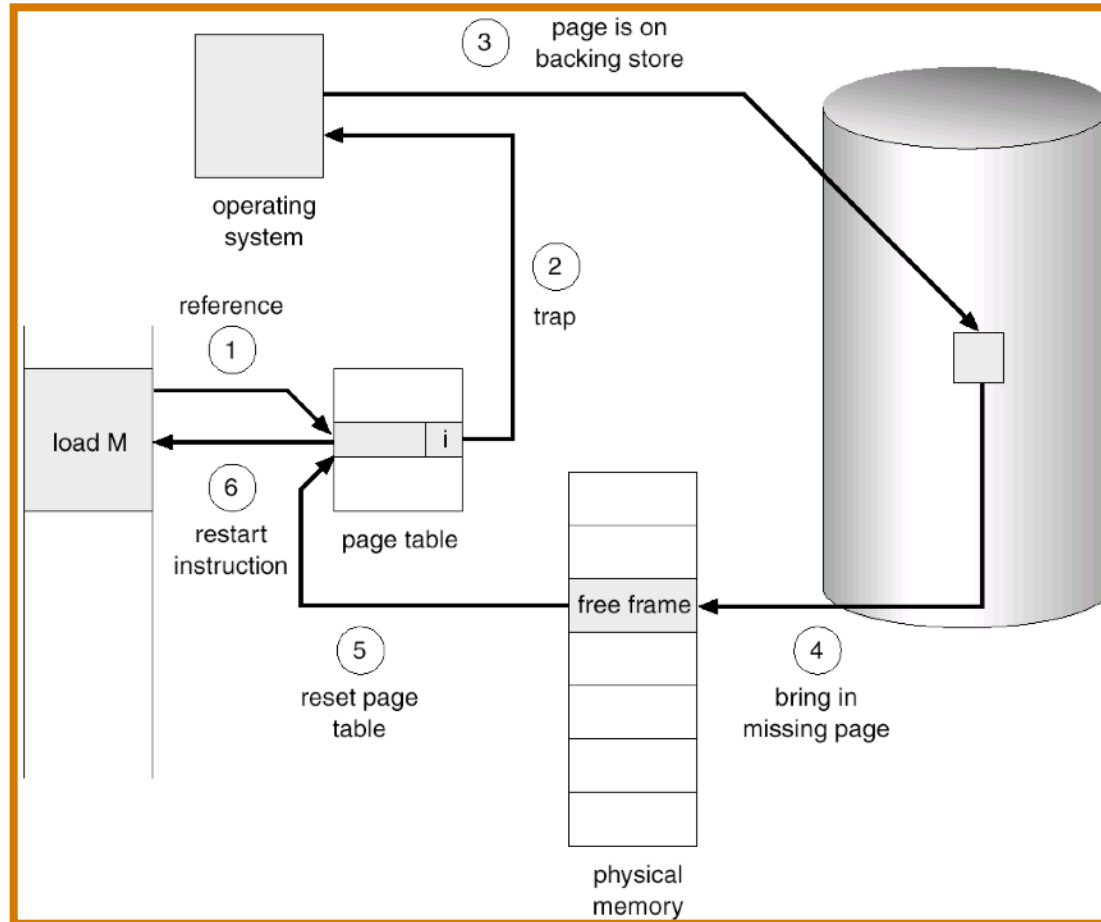
- During address translation, if valid–invalid bit in page table entry is 0  $\Rightarrow$  page fault.



# Page Fault

- If there is ever a reference to a page, first reference will **trap** to OS ⇒ page fault.
- OS looks at page table to decide:
  - **If it was an invalid reference ⇒ abort with error message.**
  - **If it was a reference to a page that is not in memory, continue.**
- Locate an empty frame.
- Swap page into frame.
- Correct the page table and set validation bit = 1.
- Restart the instruction that caused the page fault.

# Steps in Handling a Page Fault



# No free frame: now what?

- **Page replacement:** Are all those pages in memory being referenced? Choose one to *swap out* to disk and make room to load a new page.
  - **Swap out:** Do you *really* have to save it to disk?
  - **Algorithm:** How do you choose a victim?
  - **Performance:** What algorithm will result in the *lowest possible number* of page faults?
- **Life with VM:** The same page may be brought in and out of memory several times.

# 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):**

$$\text{EAT} = [(1 - p) (\text{memory access})] + [p (\text{page fault overhead})]$$

where:

$$\begin{aligned} \text{page fault overhead} &= [\text{swap page out}] + [\text{swap page in}] \\ &+ [\text{restart overhead}] \end{aligned}$$

# Page Table

**frame #**

**page #** →

...	
7	
6	
5	
4	
3	
2	
1	
0	

# Page Table

	<b>frame #</b>	<b>valid</b>
...		
7		
6		
5		
<b>page #</b> → 4		
3		
2		
1		
0		

# Page Table

	<b>frame #</b>	<b>valid</b>	<b>dirty</b>
...			
7			
6			
5			
<b>page #</b> → 4			
3			
2			
1			
0			

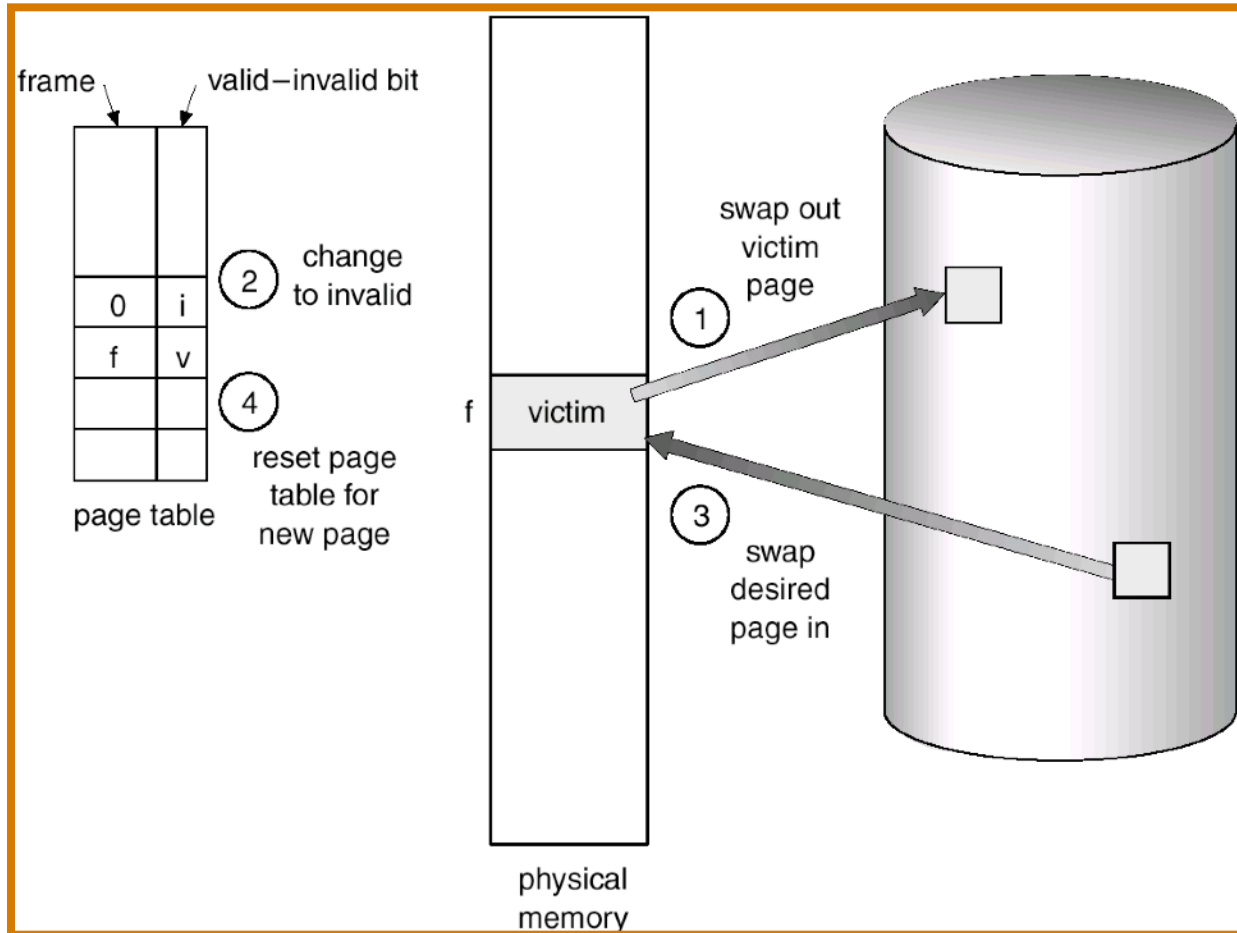
# Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
- Use a ***dirty (modified) bit*** to reduce overhead of page transfers – only modified pages are written back to disk.
- Page replacement completes the separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

# 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 table*** and ***frame table***.
4. Restart the instruction.

# Page Replacement

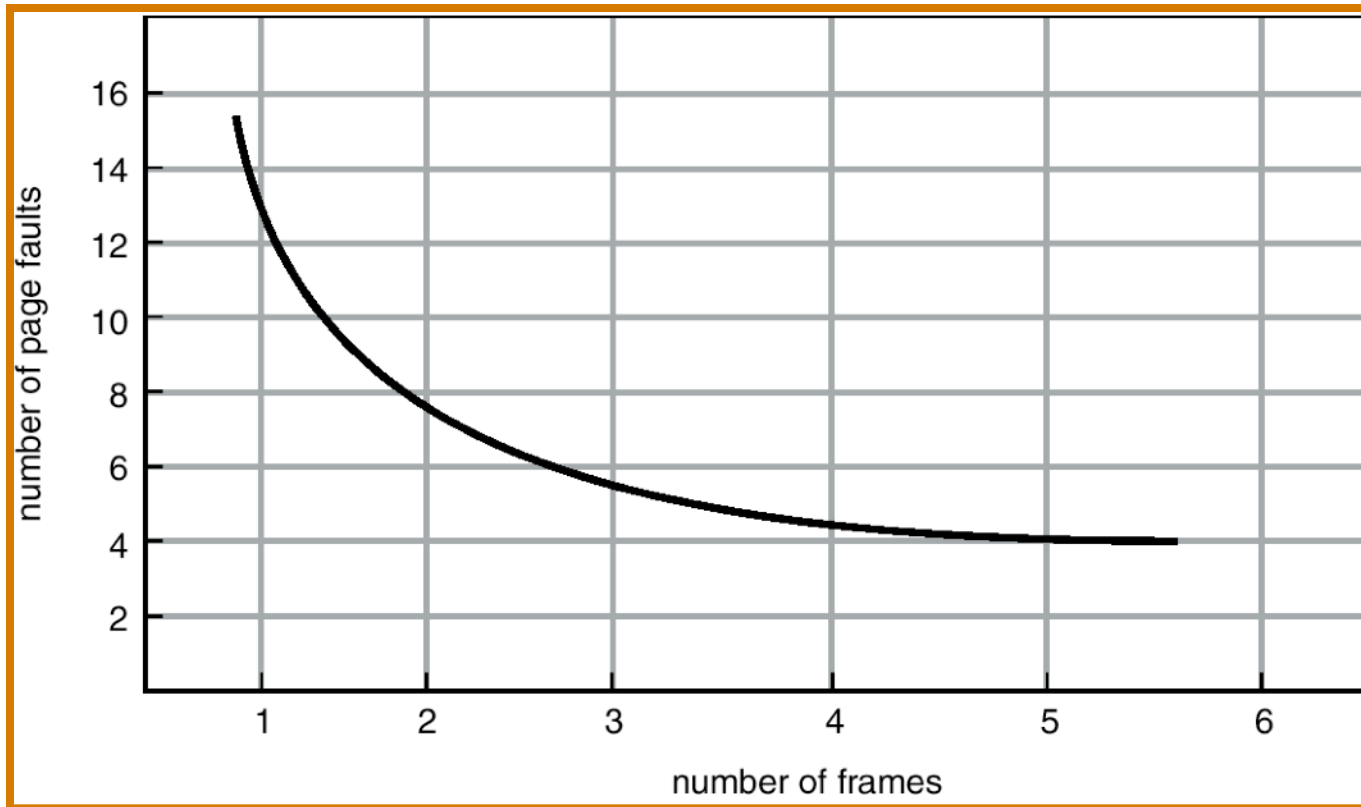


# Page Replacement Algorithms

- **Goal:** Produce a low 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.
- The reference string is produced by tracing a real program or by some stochastic model. We look at every address produced and strip off the page offset, leaving only the page number. For instance:

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

# Graph of Page Faults Versus The Number of Frames



# FIFO Page Replacement

- 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)

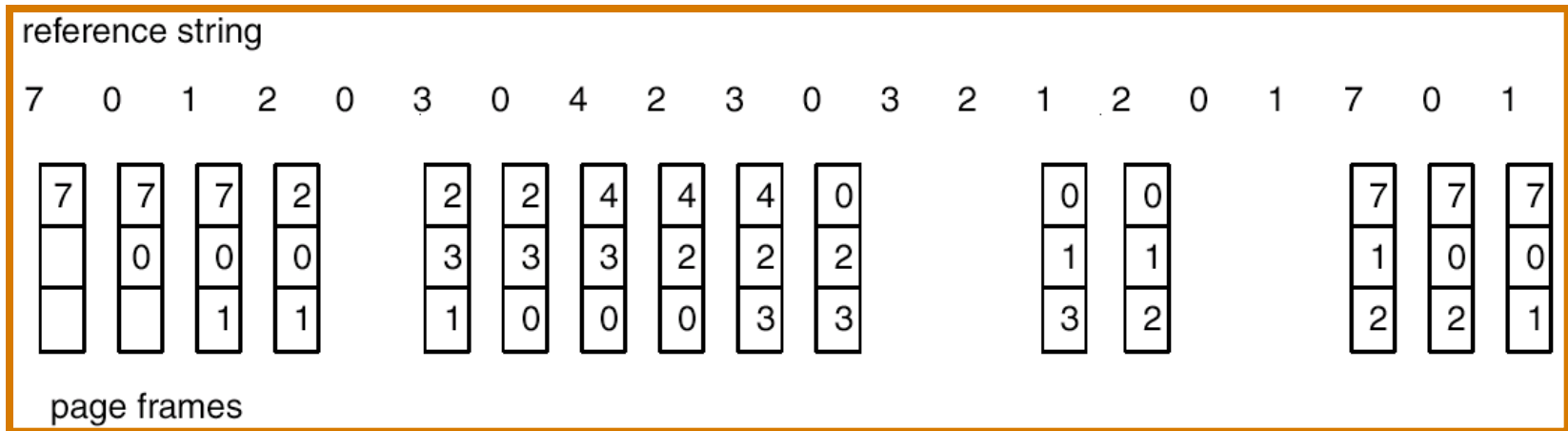
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

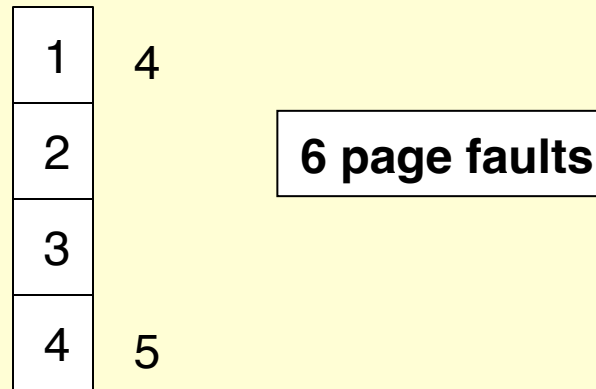
- FIFO Replacement  $\Rightarrow$  **Belady's Anomaly**: more frames, *more* page faults.

# FIFO Page Replacement



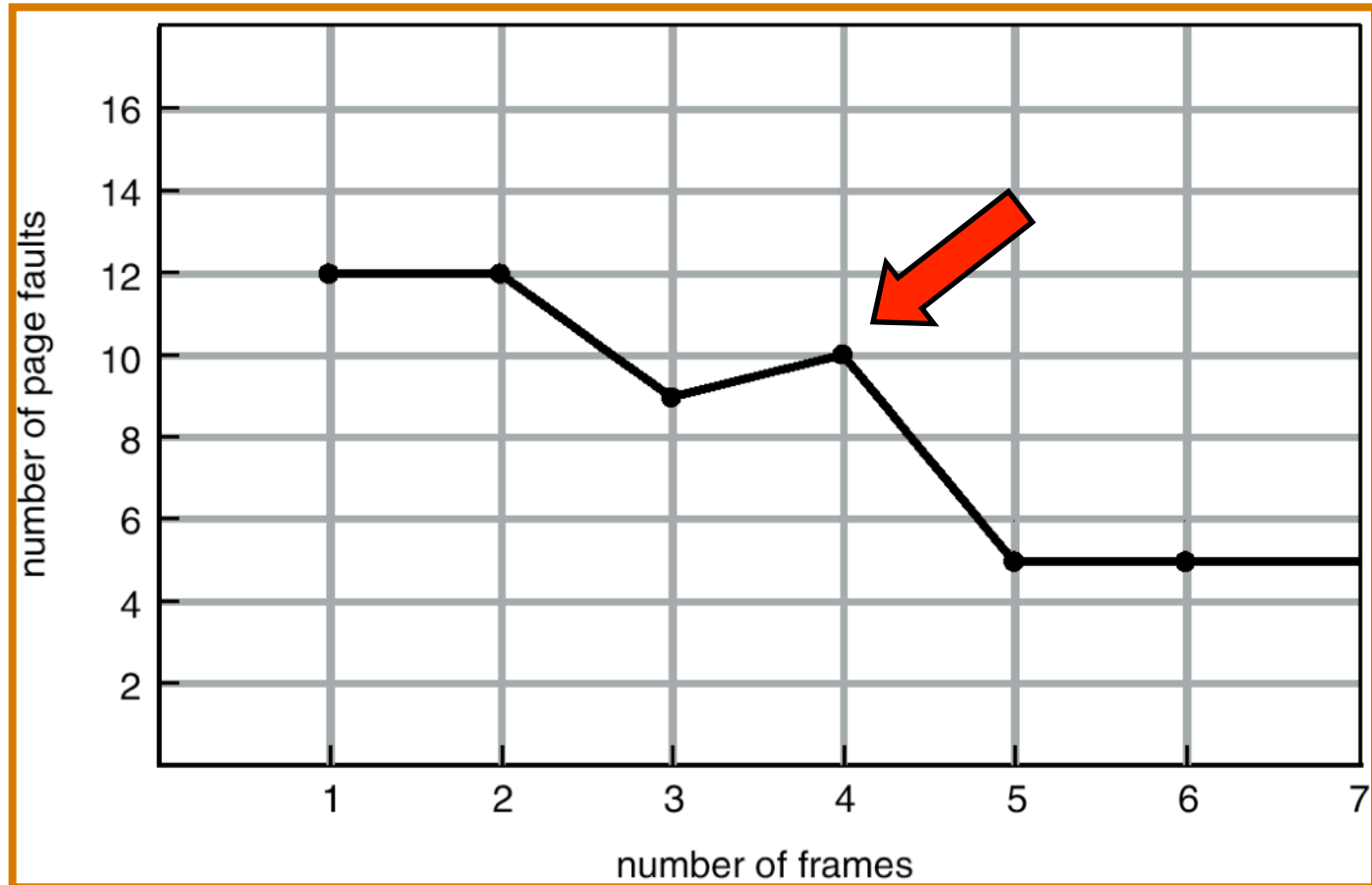
# Optimal Algorithm

- Replace the page that will not be used for longest period of time. (How can you know what the future references will be?)
- 4 frames example: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



- Used for measuring how well your algorithm performs.

# FIFO (Belady's Anomaly)





# LRU Algorithm

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

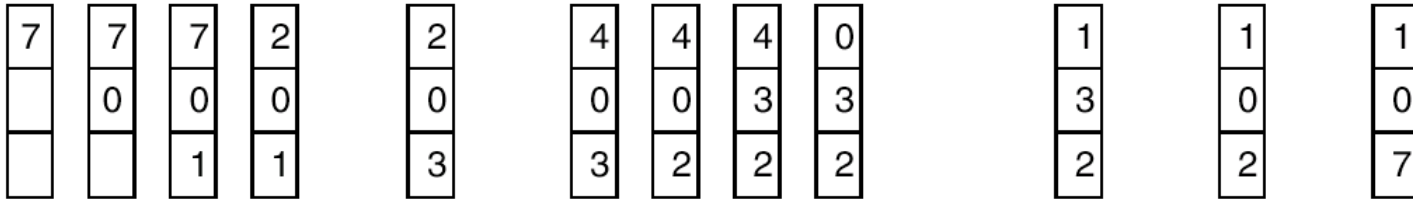
1	5	
2		
3	5	4
4	3	

- 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

reference string

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



page frames

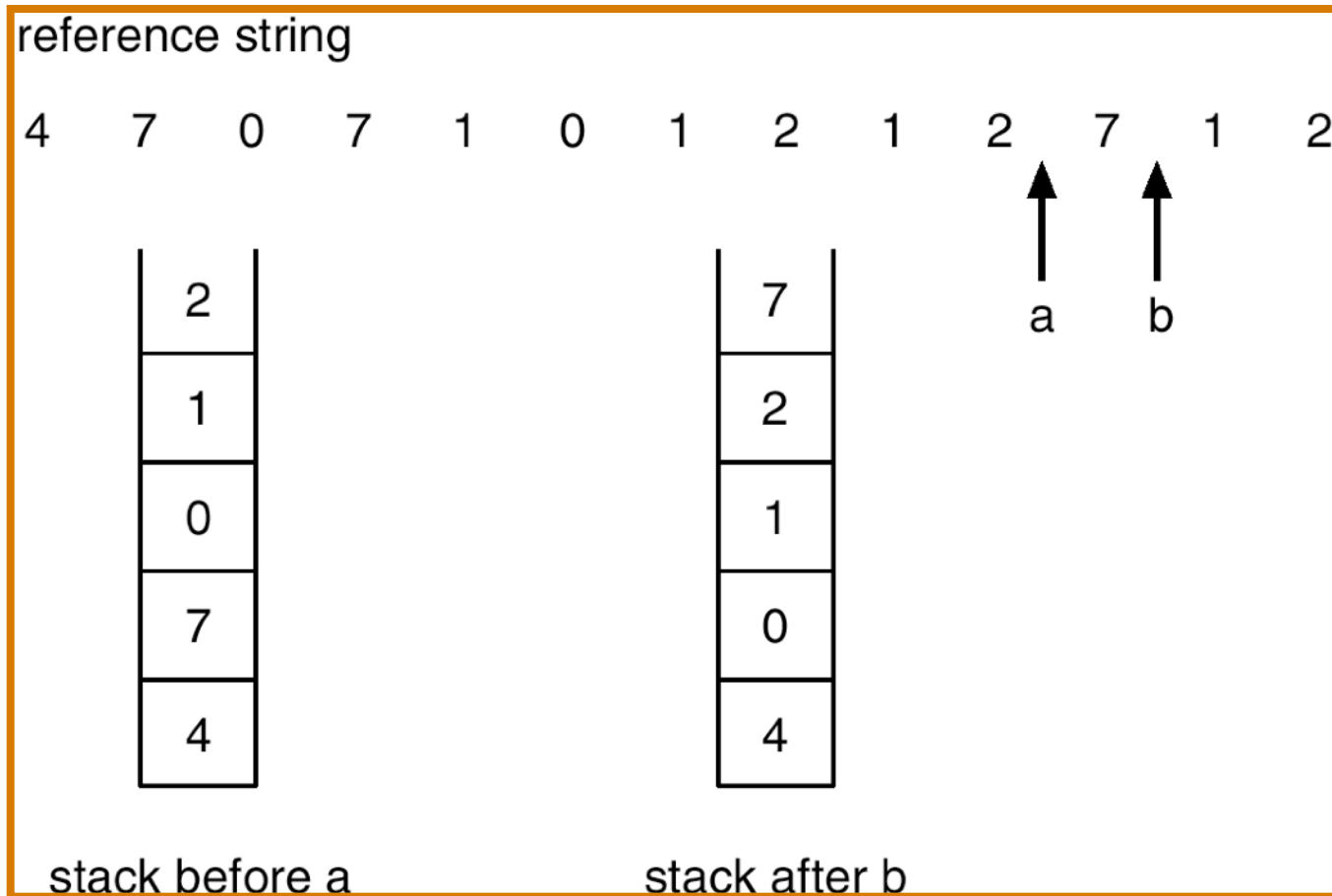
# LRU Algorithm (Cont.)

- 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.

# LRU and Belady's Anomaly

- LRU does not suffer from Belady's Anomaly (OPT doesn't either).
- It has been shown that algorithms in a class called **stack algorithms** can never exhibit Belady's Anomaly.
- A **stack algorithm** is one for which the set of pages in memory for  $n$  frames is a subset of the pages that would be in memory if you had  $n+1$  frames.

# Use Of A Stack to Record The Most Recent Page References



# LRU Approximation Algorithms

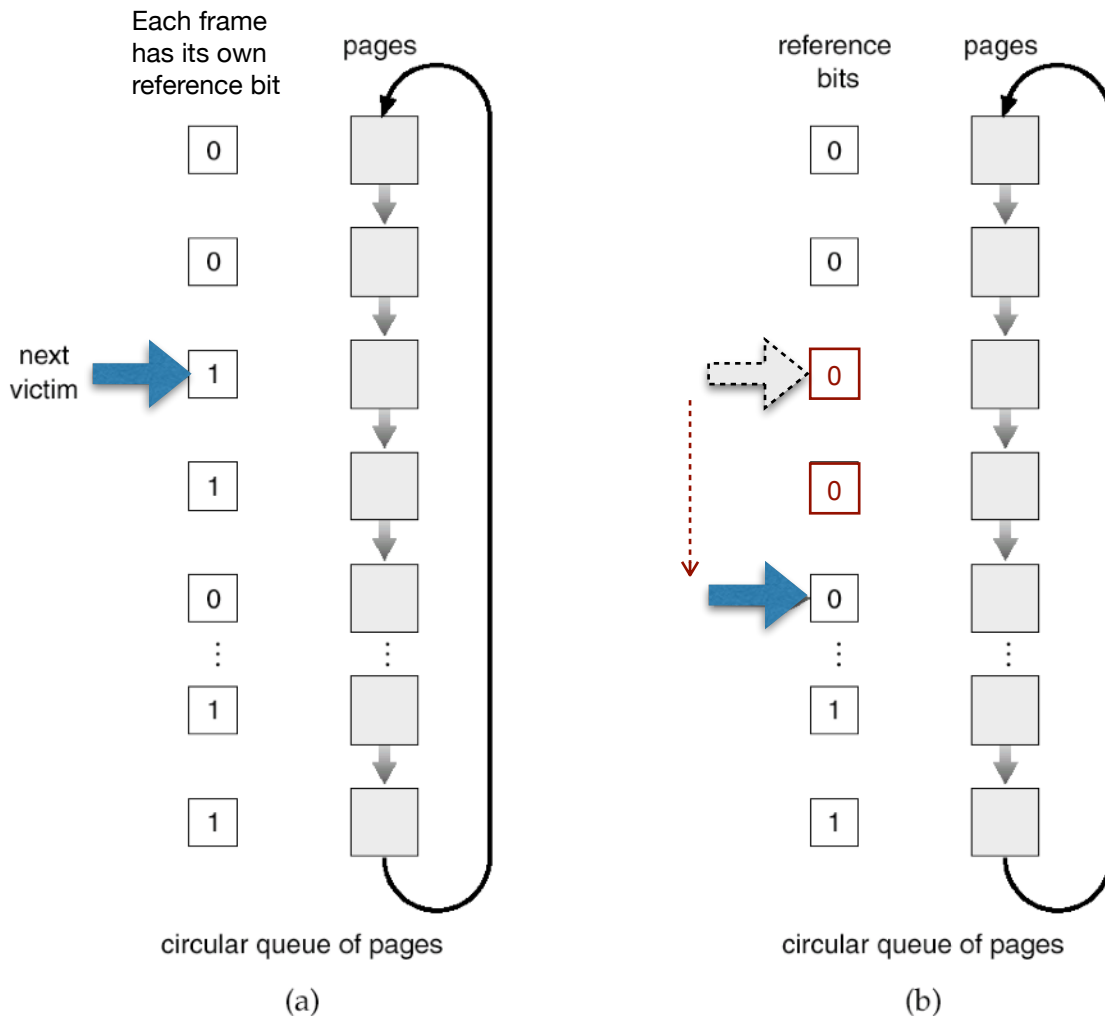
- **Reference bit**

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

- **Second chance**

- Need reference bit.
- Clock replacement.
- If page to be replaced (in clock order) has reference bit = 1. then:
  - set reference bit 0.
  - leave page in memory.
  - replace next page (in clock order), subject to same rules.

# Second-Chance (clock) Page-Replacement Algorithm



## When a page fault occurs:

- Look at the page under the clock hand
- If  $R = 0$ 
  - Evict this page (it has not been used recently)
- If  $R = 1$ 
  - ➔ Set  $R = 0$  (give it a second chance)
  - ➔ Move the clock hand forward
- Repeat the process

# Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- **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.

**Sidebar: optimizations**

# 1. Handling Writes to VM

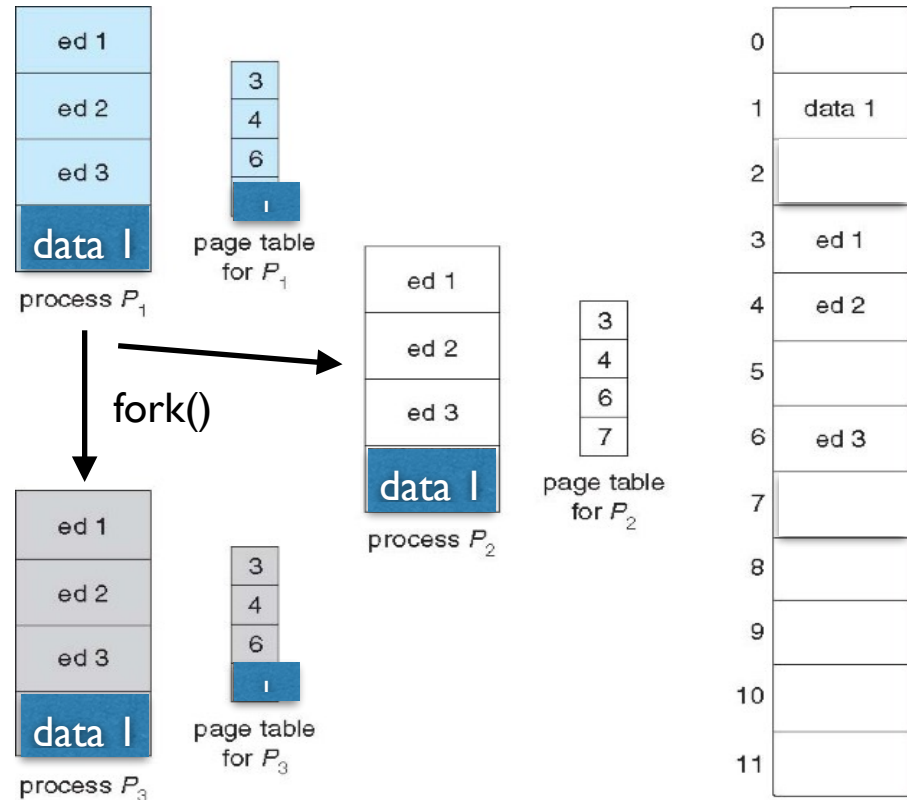
Remember the two policies used in cache memory for dealing with writes?

- **Write through**
- **Write back**

Discuss whether they are both applicable to handling writes to pages of virtual memory.

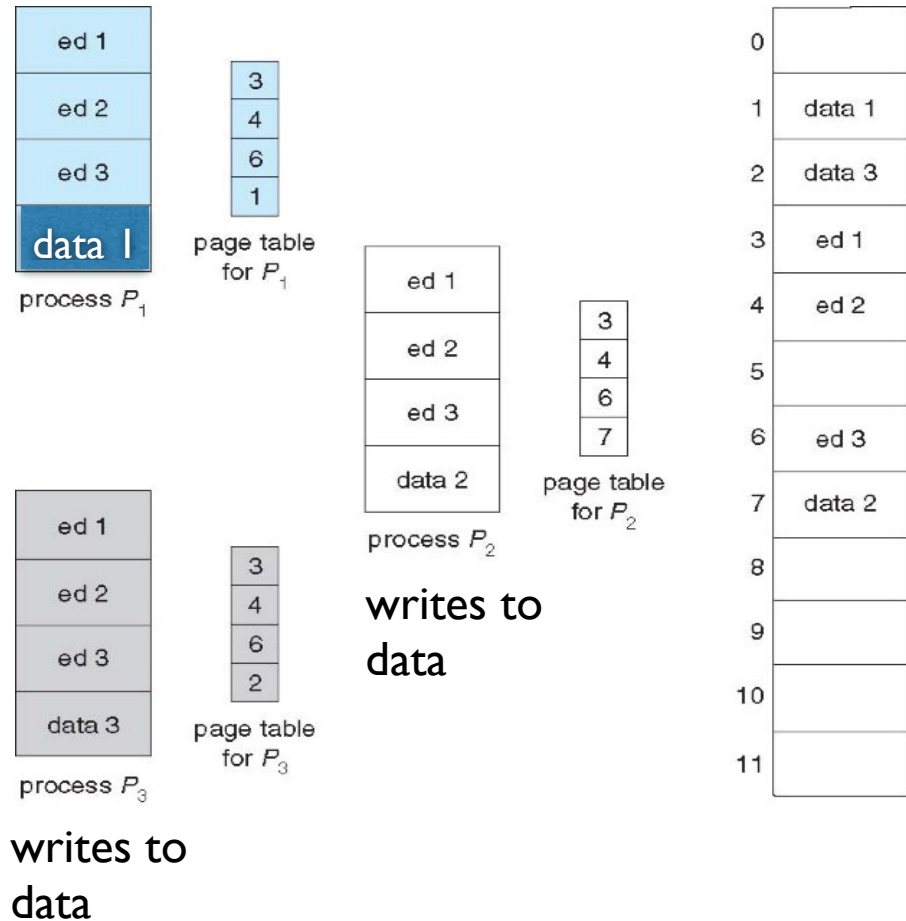
# 2. Copy-On-Write

When two processes are related by birth, there's an interesting **optimization** that comes very naturally with VM...



# 2. Copy-On-Write

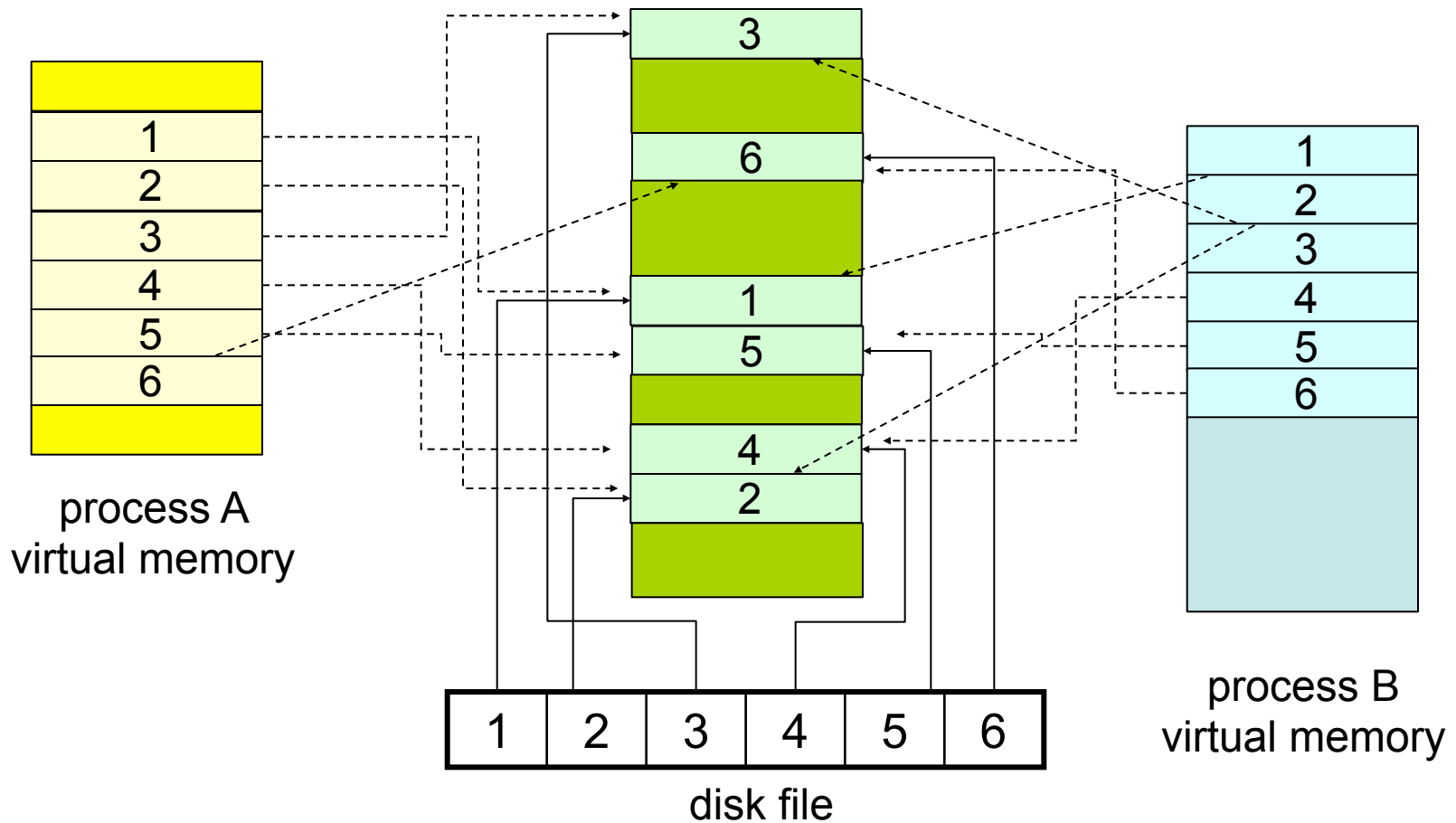
When two processes are related by birth, there's an interesting **optimization** that comes very naturally with VM...



# Memory-mapped Files

- Memory mapping a file can be accomplished by mapping a disk block to one or more pages in memory.
- A page-sized portion of the file is read from the file system into a physical page. Subsequent `read()` and `write()` operations are handled as memory (not disk) accesses.
- Writing to the file in memory is not necessarily synchronous to the file on disk. The file can be committed back to disk when it's closed.

# Memory-mapped Files



# Prepaging

- **Prepaging**: In order to avoid the initial number of page faults, the system can bring into memory all the pages that will be needed all at once.
- This can also be applied when a swapped-out process is restarted. The smart thing to do is to remember the working set of the process.
- One question that arises is whether all the pages brought in will actually be used...
- Is the cost of prepaging less than the cost of servicing each individual page fault?

**End Sidebar**

# Allocation of Frames

- Each process needs a **minimum** number of pages.
- There are two major allocation schemes:
  - **fixed allocation**
  - **priority allocation**

# Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

–  $s_i$  = size of process  $p_i$

–  $S = \sum s_i$

–  $m$  = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

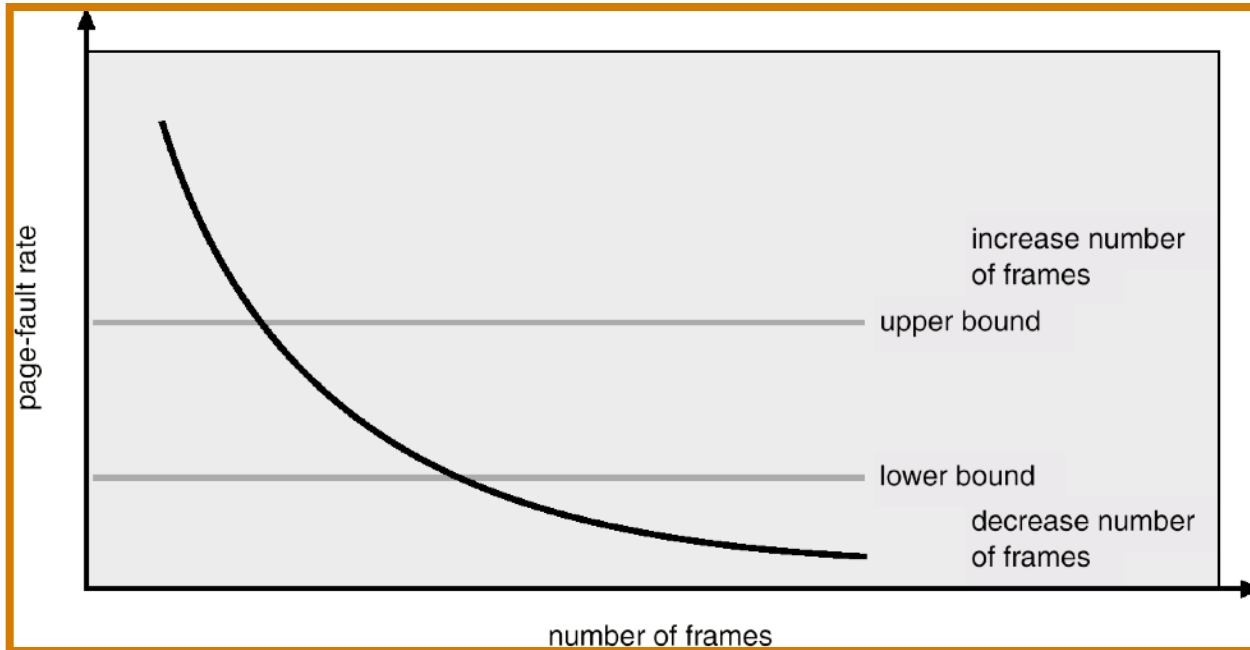
$$a_1 = \frac{10}{137} \times 64 \approx 5$$

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

# Priority Allocation

- The **proportional allocation** scheme can use **priorities instead of size**.
- If process  $P_i$  generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

# Page-Fault Frequency Scheme



**Establish “acceptable” page-fault rate.**

- If actual rate too low, process loses frame.
- If actual rate too high, process gains frame.

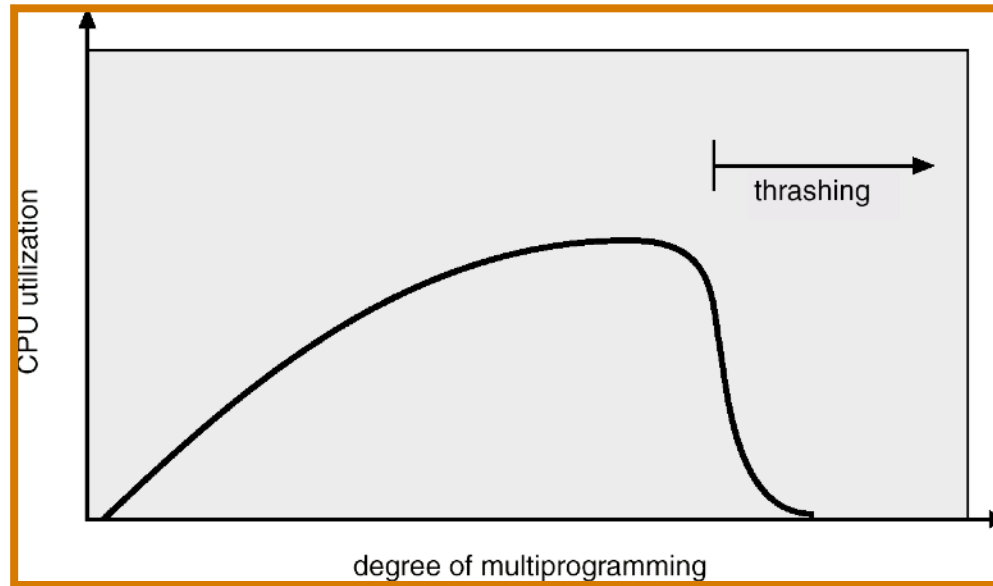
# Global vs. Local Replacement

- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local** replacement – each process selects from only its own set of allocated frames.

# Thrashing

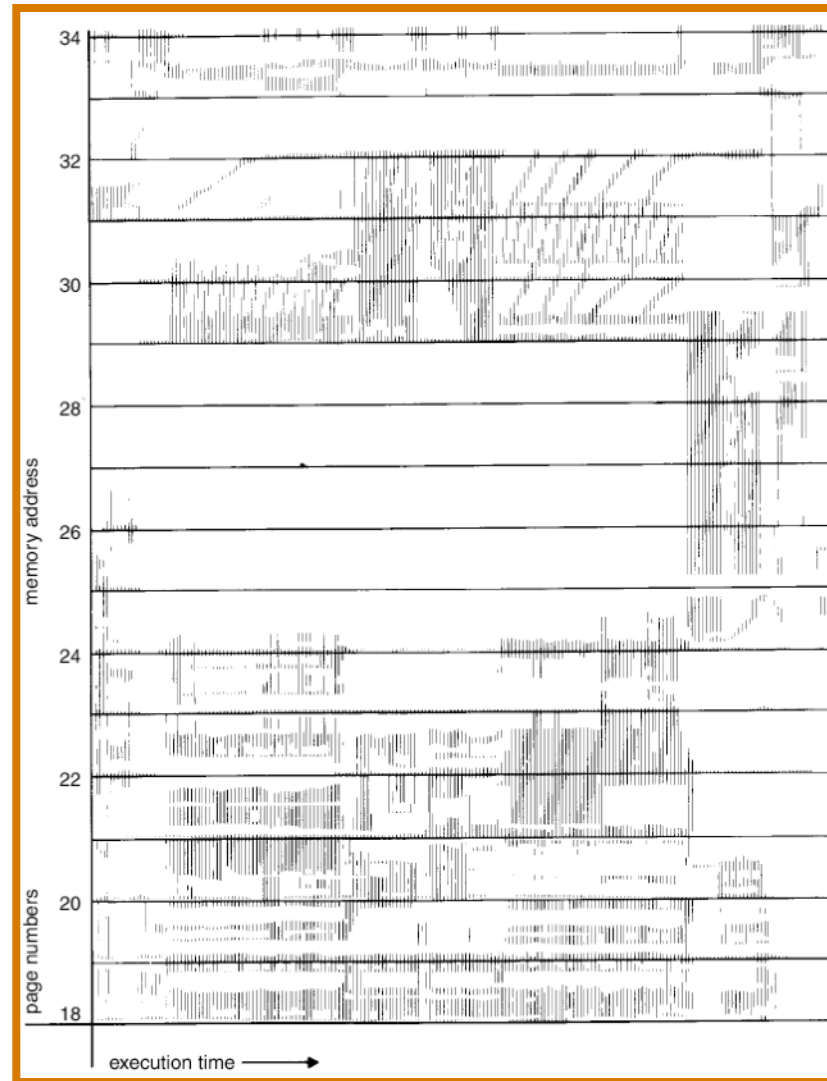
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - **Low CPU utilization.**
  - Operating system thinks that it needs to increase the degree of multiprogramming.
  - Another process added to the system.
- **Thrashing**  $\equiv$  a process is busy swapping pages in and out.

# Thrashing



- Why does paging work?  
Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?  
 $\Sigma$  size of locality > total memory size

# Locality in Memory-Reference Pattern



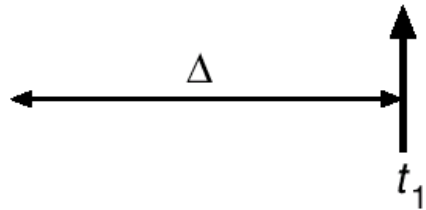
# Working-Set Model

- $\Delta \equiv$  **working-set window**  $\equiv$  a fixed number of page references.
- $WSS_i$  (working set of Process  $P_i$ ) = total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality.
  - if  $\Delta$  too large will encompass several localities.
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program.
- $D = \sum WSS_i \equiv$  total demand frames
- if  $D > m \Rightarrow$  **Thrashing**
- Policy if  $D > m$ , then suspend one of the processes.

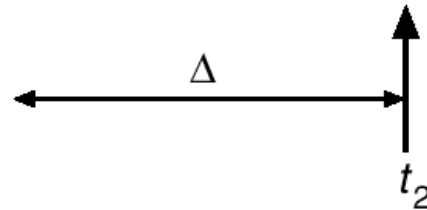
# Working-set model

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example:  $\Delta = 10,000$ 
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts copy and set the values of all reference bits to 0.
  - If one of the bits in memory = 1  $\Rightarrow$  page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.