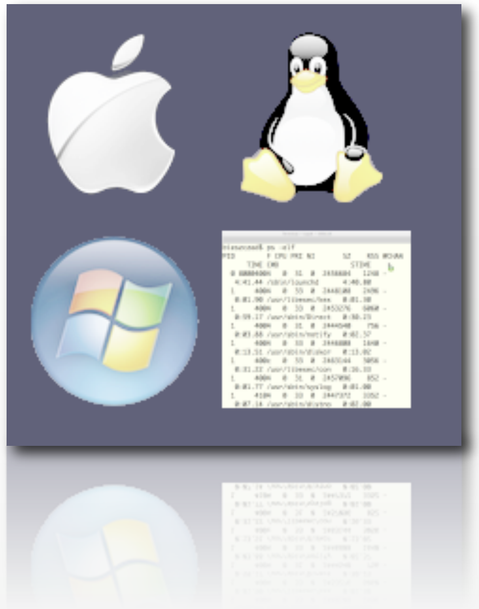


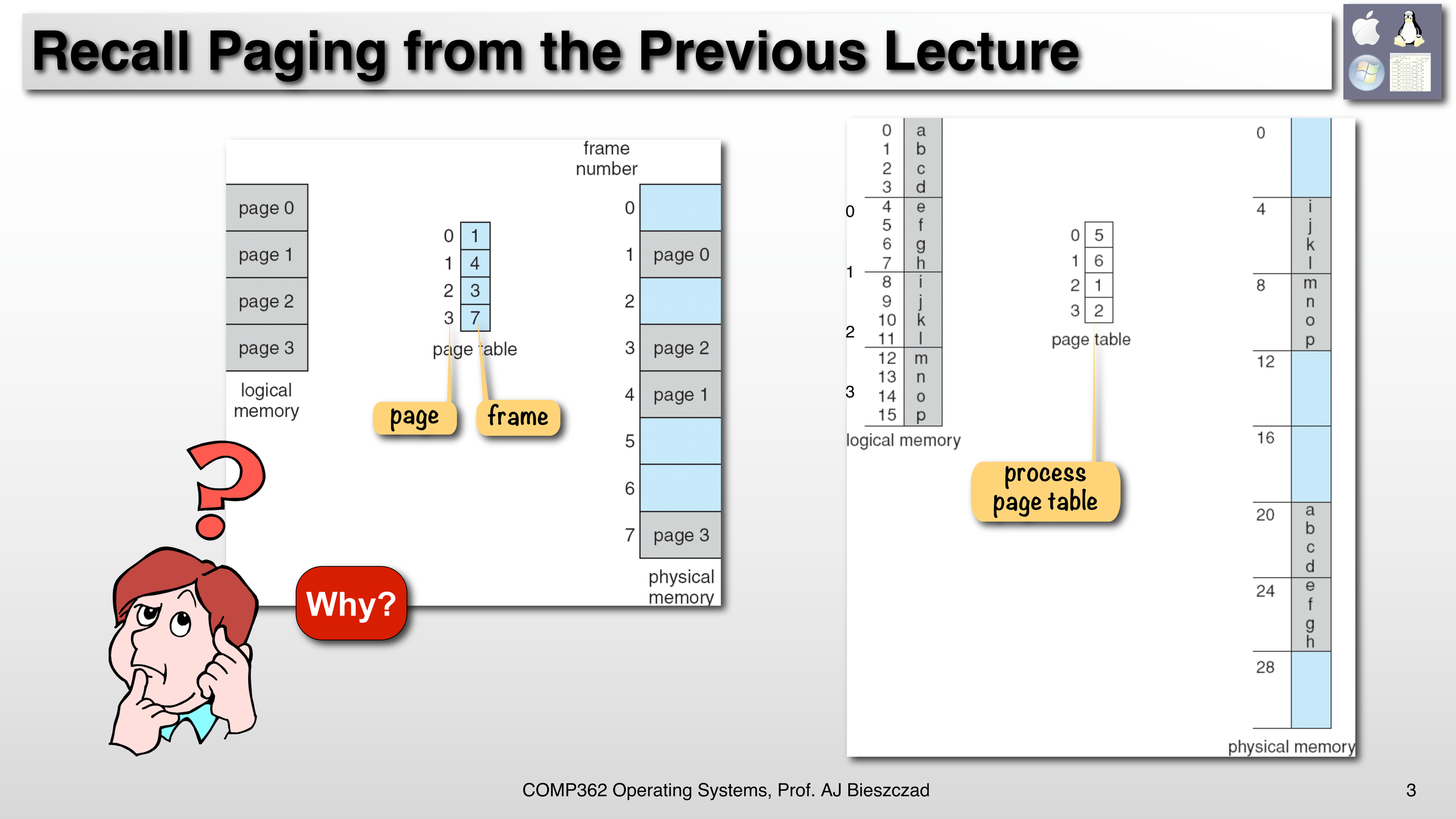
Lecture 10: Virtual Memory

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline

- Virtual memory concept
- On-demand paging
- Page replacement
- Allocation of frames to processes
- Problem with page thrashing
- Related issues
  - Translation Look-aside Buffer (TLB) reach
  - page size
  - pre-paging
  - copy-on-write (COW)

Recall Paging from the Previous Lecture

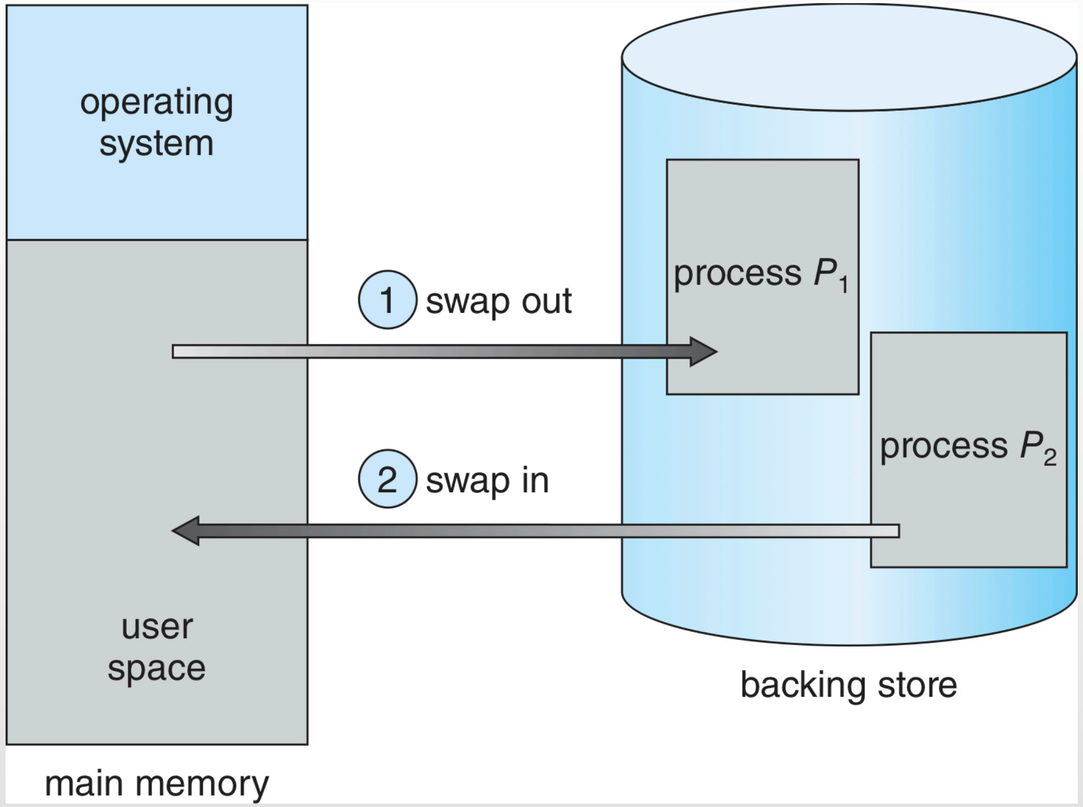


Swapping Processes Out and In

- Often, OS cannot afford keeping all running programs in main memory, so programs must be swapped in and out of the memory

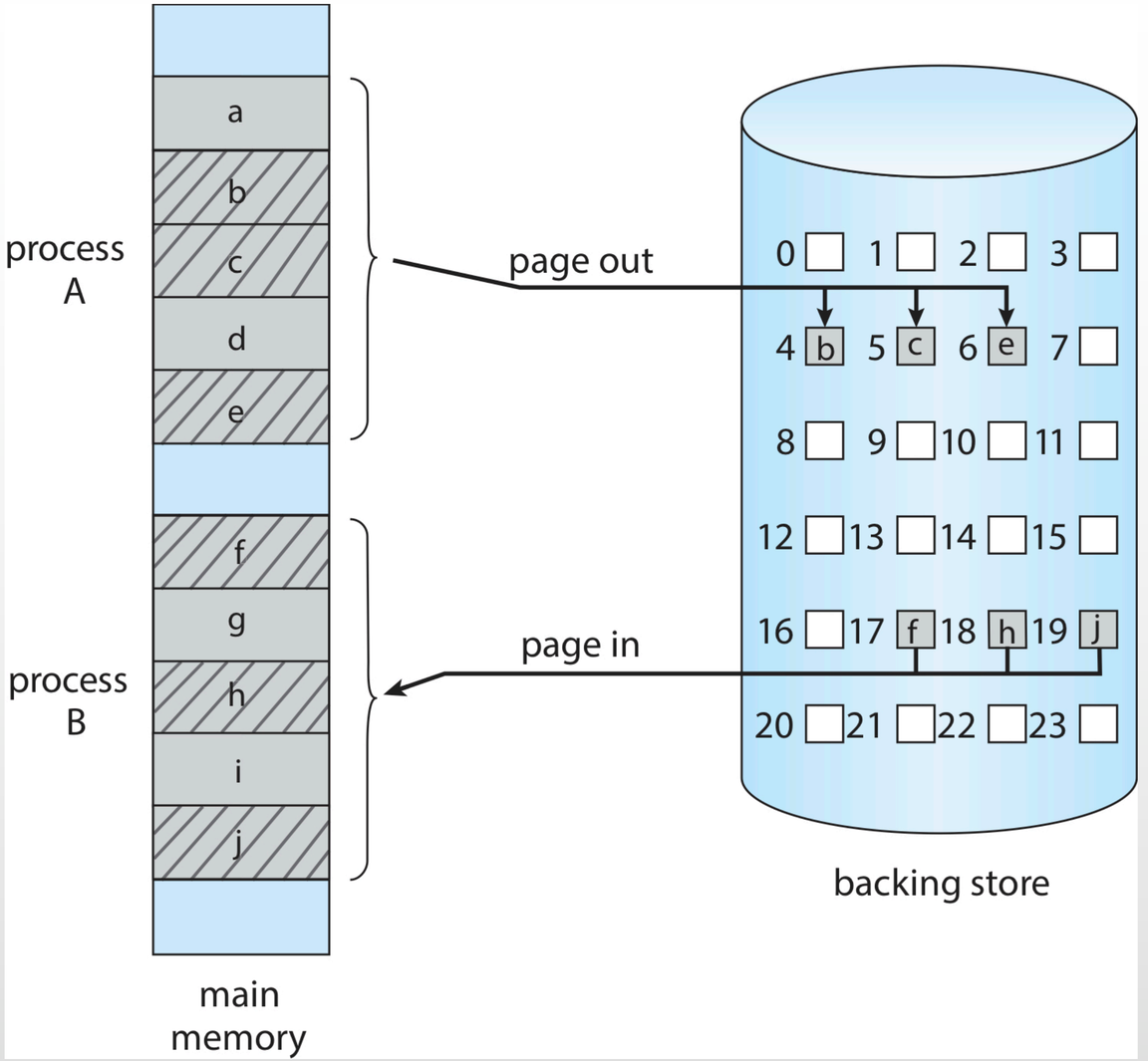
- A process image can be swapped out of memory temporarily to a **backing store**, and then brought back (swapped in) into memory for continued execution
  - e.g., when another process is scheduled for execution, and there is no room for its code

Versions of swapping are common on many systems (e.g., UNIX, Linux, and Windows)



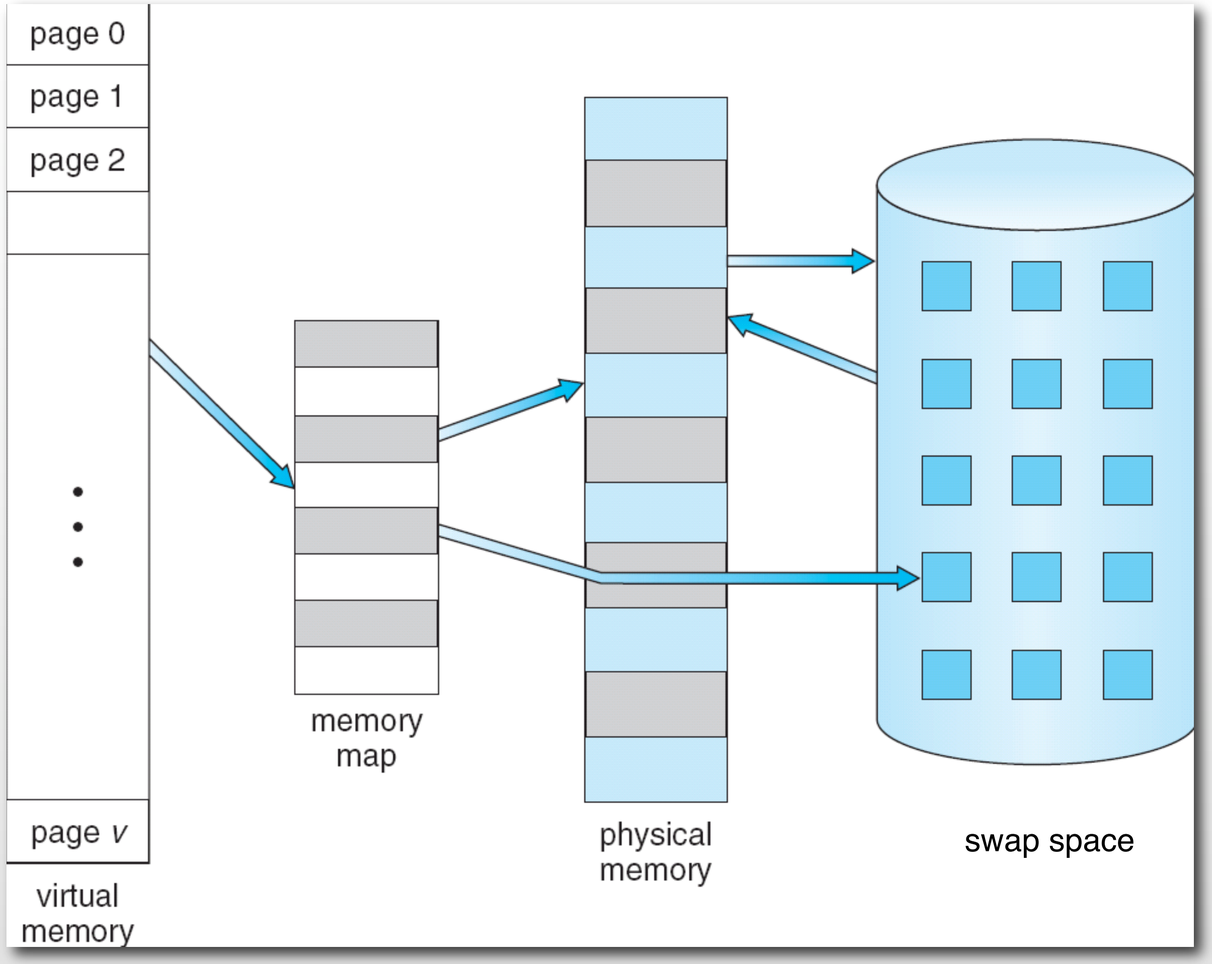
Efficient Page-Based Backing Store

- Backing store (**swap space**: file or partition)
  - fast disk space large enough to accommodate copies of all memory images for all users
  - must provide fast access to these memory images
    - major part of swap time is transfer time
      - total transfer time is directly proportional to the amount of swapped memory
- Swapping out and in pages rather than whole processes is much more flexible and faster
- System maintains a registry of process pages that are swapped out
  - more about that in a few slides



Virtual Memory Concept

- **Virtual memory**
  - further separation of user logical memory from physical memory
- Only part of the program needs to be in memory for execution
  - virtual logical address space can therefore be much larger than physical address space
- Just like in paging, each process has a memory map (page table) that maps process pages into physical memory frames
- However, **a frame might be either in the physical memory, or on a disk**
  - if a reference is made to a frame that is not in the memory, then the OS has to fetch it from the disk
  - recall: swap partition, or swap file, holds all pages of the running programs that are not in the physical memory
- **Lazy swapper** swaps a page into memory only when page is needed (**on-demand paging**)

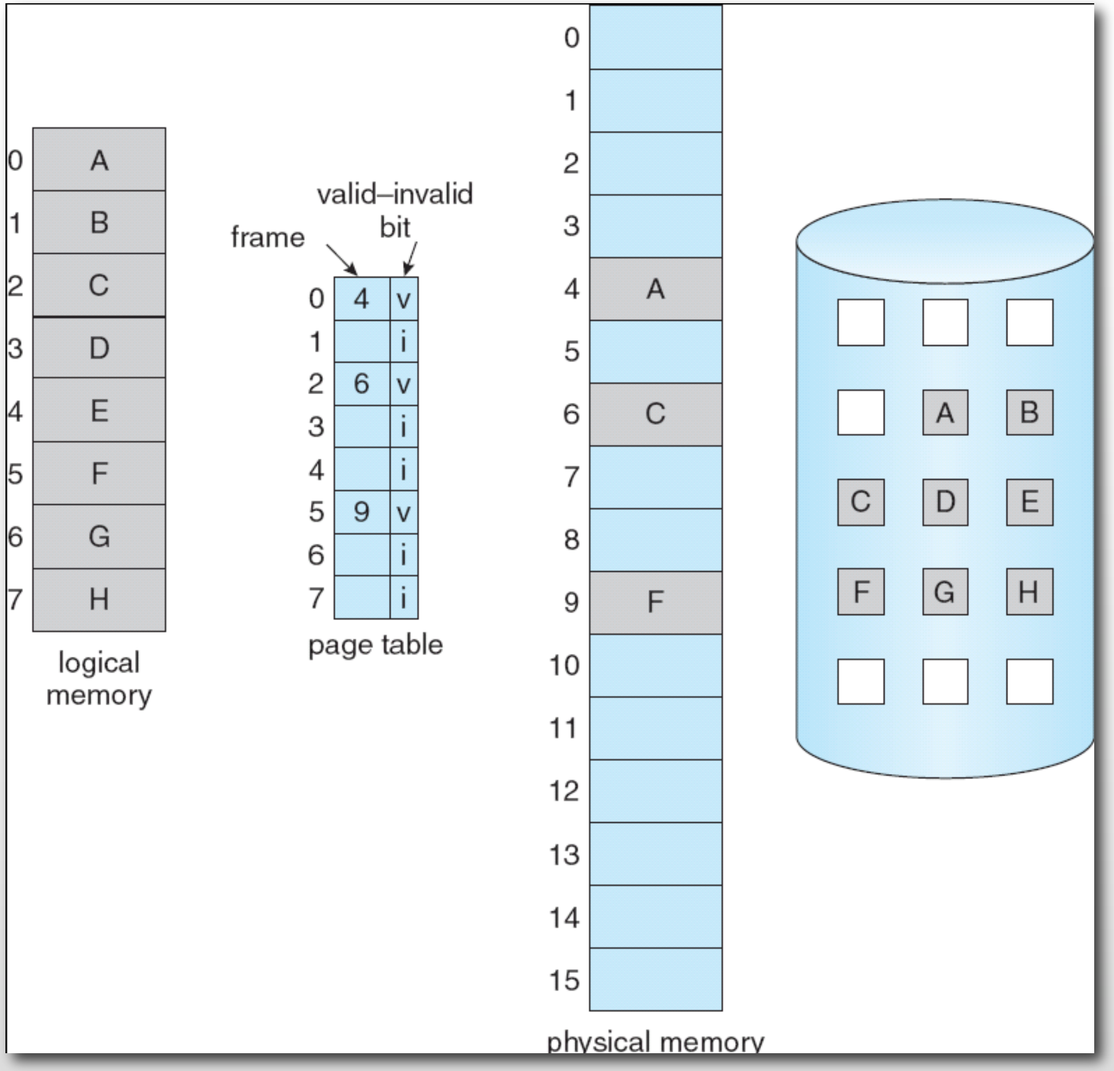


Benefits of Memory Virtualization

- Flexible process management
  - Process does not need to wait for a chunk of memory to fit in
    - only some portion of a process can be loaded into the memory
    - better system response
  - Total logical space for all processes can be larger than the available physical address space
    - more concurrent processes
    - more concurrent users
  - Address spaces can be shared by several processes
    - the more processes use the shared space, the more likely for the frames to be in the memory
      - e.g., dynamic libraries
- Less I/O
  - fewer bytes to swap in and out
  - improved performance

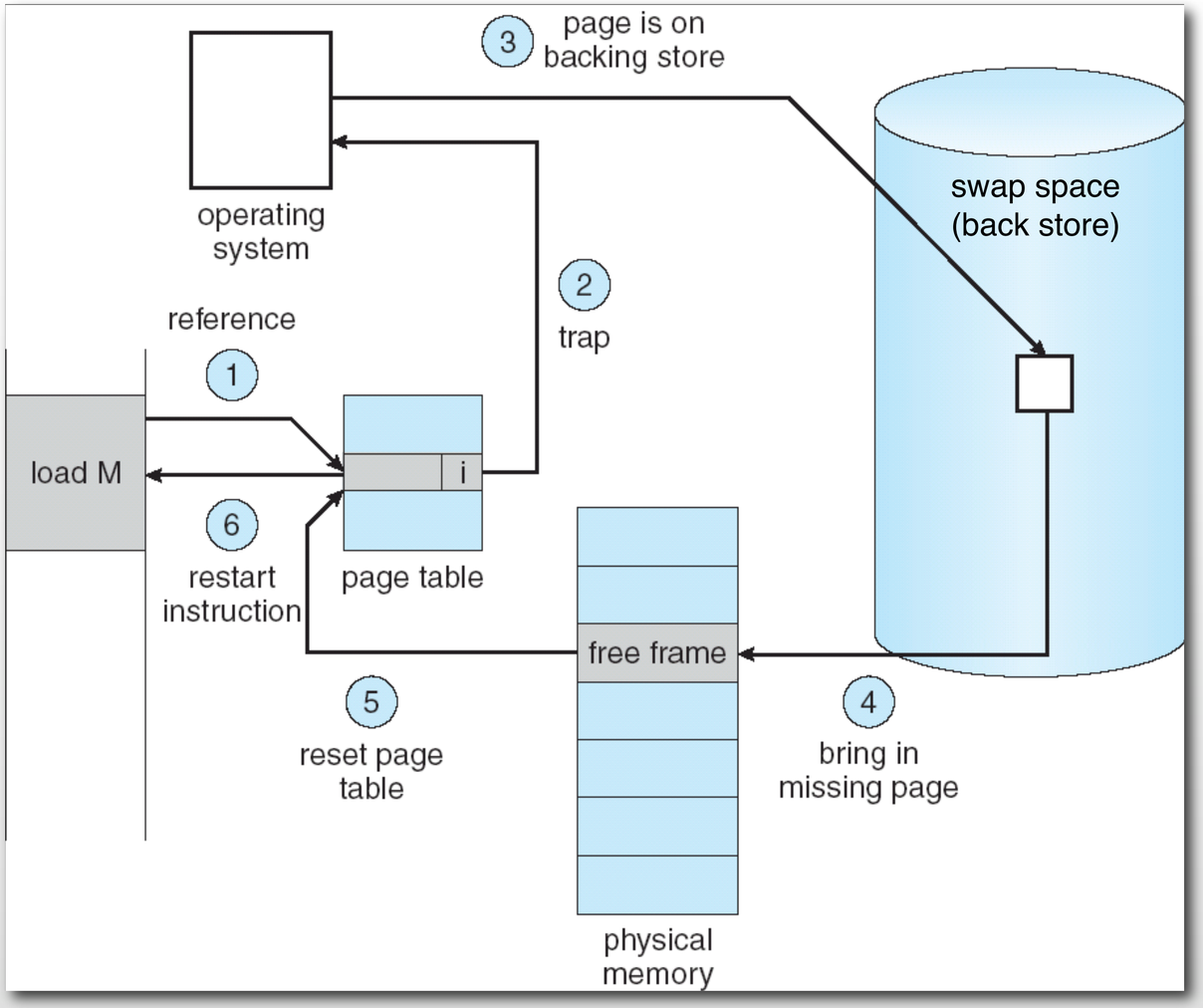
Valid-Invalid Bit in Page Tables

- A valid–invalid bit is associated with each entry in the process page table
  - **v (valid)** ⇒ page in-memory; just use it
    - note that this is a table for a given process, so we are past the security checks for legality of the access
  - **i (invalid)** ⇒ page not-in-memory
    - fetch the page from the disk
    - either find a free frame, or overwrite another one using a page replacement policy
    - initially valid–invalid bit can be set to invalid on all entries
      - during address translation, any needed page will be brought into memory because its valid–invalid bit in page table entry is invalid (the essence of lazy swapping/on-demand paging)



Page Fault Service Routine

- If there is a reference to a page **(1)**, first reference to that page (“page invalid”) will trap to operating system **(2)**
  - this is called a page fault
- Operating system checks the validity of the reference for the given process
  - invalid ⇒ abort (“segmentation violation” error)
  - valid ⇒ a reference to a page that is not in memory **(3)** ⇒ bring the page to the memory:
    - get an empty frame **(4)**
    - swap the page into the frame **(4)**
    - reset page and frame tables **(5)**
    - set the validation bit to valid **(5)**
    - restart the instruction that caused the page fault **(6)**



Page Replacement

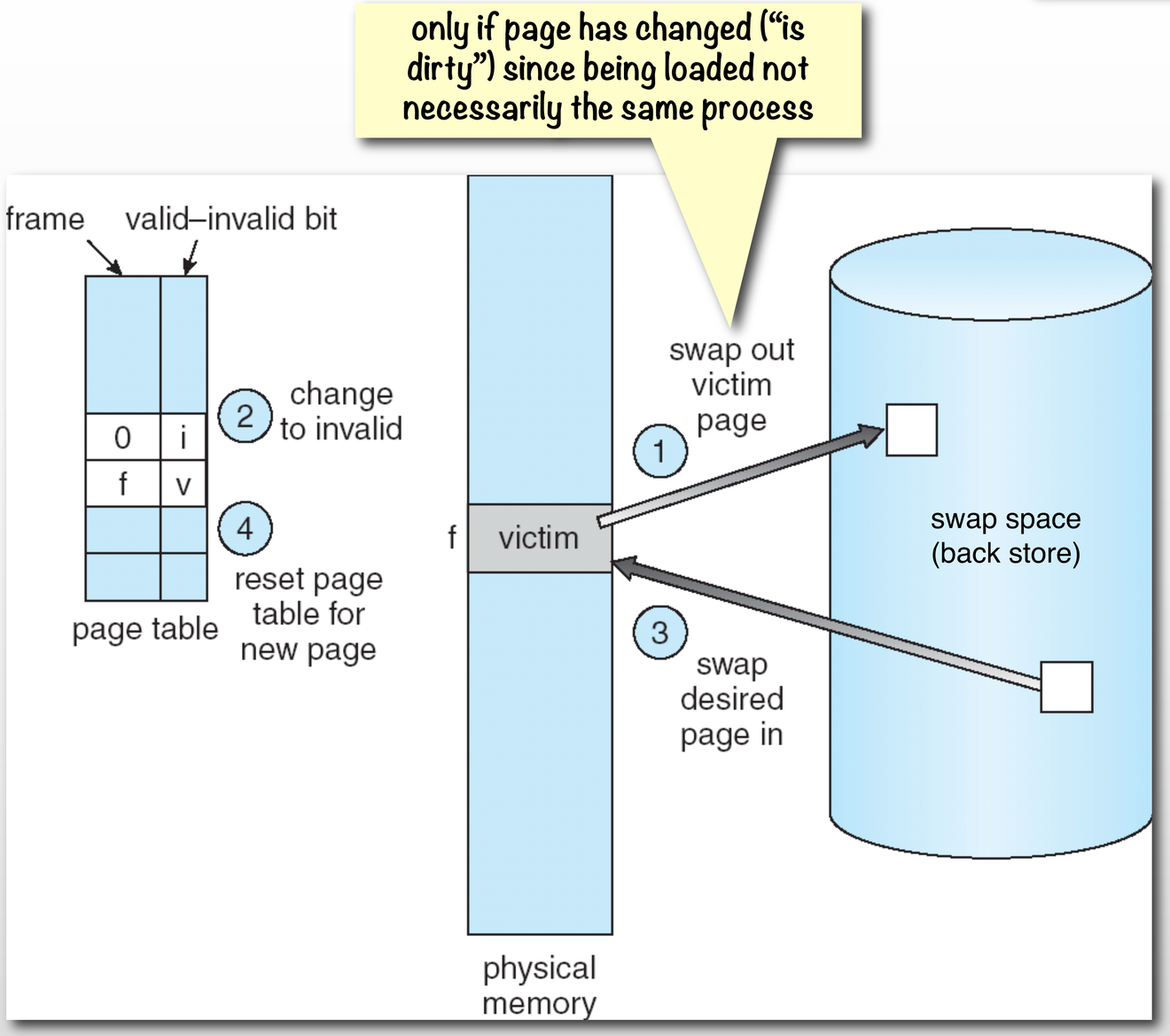
you have already answered this question in the lab!

- What happens if there is no free frame for the missing page?
- The system must perform a **page replacement**
  - find a frame in memory that holds a page that is not currently in use,
  - swap the page out freeing the frame, and
  - swap in the page that is needed
- Must take care of:
  - the algorithm to find the "**victim**" page
  - performance
    - want an algorithm which will result in minimum number of page faults
- Same page may need to be brought into memory (and swapped out) numerous times
- Page-fault service routine needs to include page replacement algorithm
- Use a modify ("**dirty**") bit to reduce overhead of page transfers
  - only tagged modified ("dirty") pages are written to disk when being replaced
    - otherwise, there is already a valid copy on the disk: the original
- Page replacement completes separation between logical memory and physical memory

Large virtual memory can be provided on a smaller physical memory

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
    - if the victim page is “dirty”, then save it to the disk (swap partition/file)
- Bring the desired page into the (newly-) freed frame and update the page table
- Restart address translation



Performance of Demand Paging

- Page Fault Rate $0.0 \le p \le 1.0$
  - probability of a page not being in memory
    - if $p = 0$ no page faults
    - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

```
EAT = (1 – p) * memory access + p * ( page fault trap overhead
                                      + swap page out
                                      + swap page in
                                      + restart overhead )
```

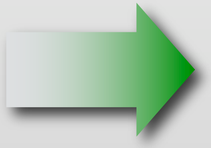
- Example

```
Memory access time = 200 nanoseconds
Average page-fault service time (disk access!) = 8 milliseconds (just reads; double that for writes if dirty bit set)

EAT = (1 – p) x 200 + p * (8 milliseconds)
      = (1 – p) * 200 + p * 8,000,000
      = 200 + p * (8,000,000 - 200)

If one access out of 1,000 causes a page fault (p = 0.001), then EAT ~= 8.2 microseconds (8200 ns)
```

  - there is a slowdown by a factor of ~41 (200 ns vs. 8200 ns)...
    - ...and we considered neither the overhead nor I/O buffer transfer time yet!



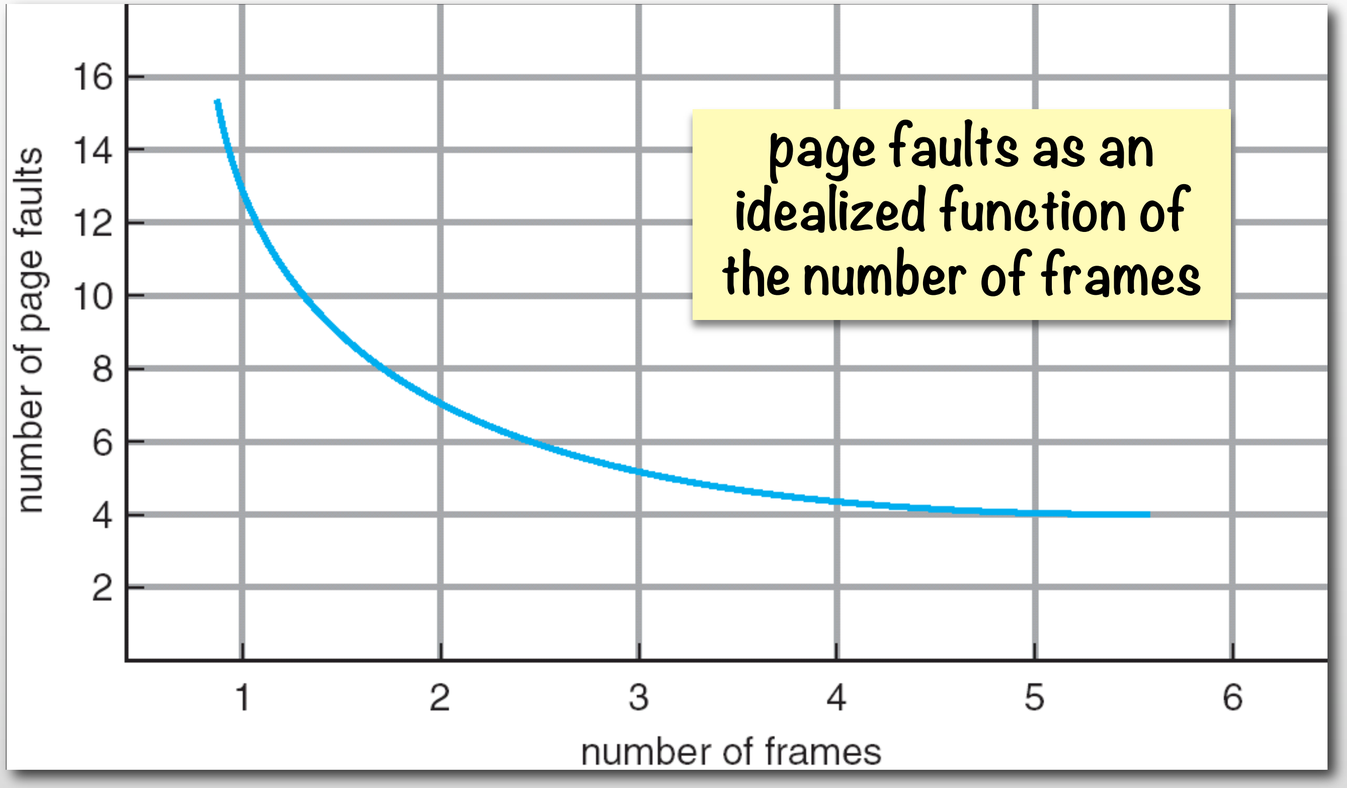
Need very efficient page replacement algorithms to avoid unnecessary swapping!

Page Replacement Algorithms

- Objective: minimize page-fault rate
- Problem: How to evaluate the quality of an algorithm?

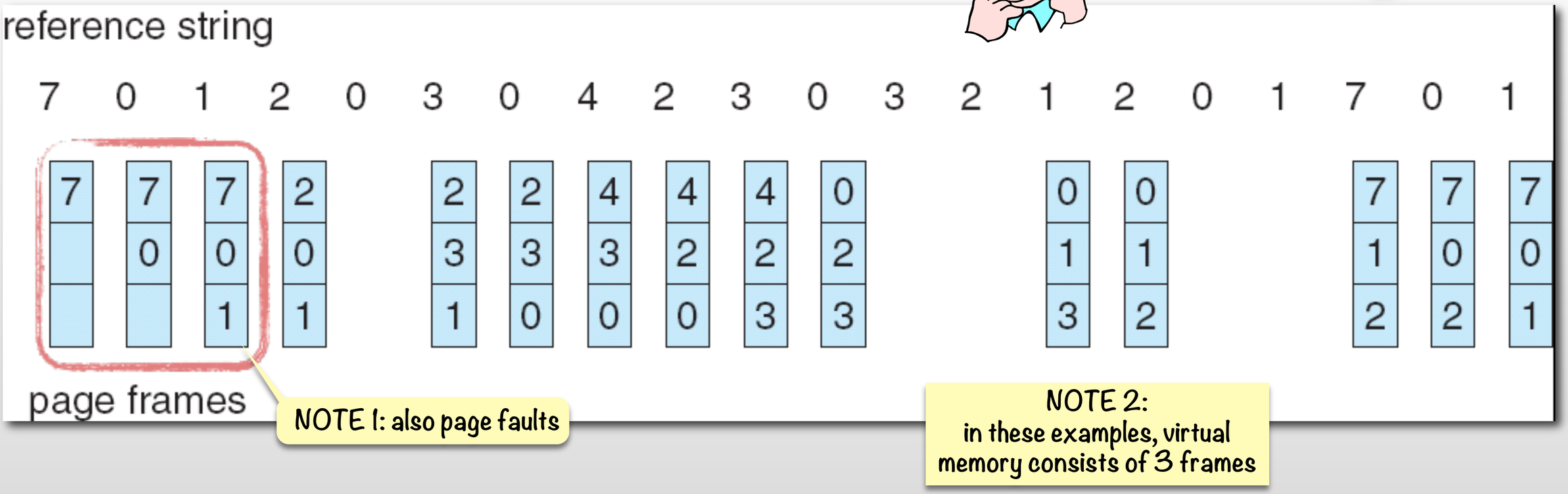
Idea!

- Run the algorithm on a sample string of memory references called reference string and count the number of page faults on that string
- Prefer an algorithm that has consistently the lowest page fault count
  - consistently == on large number of tests



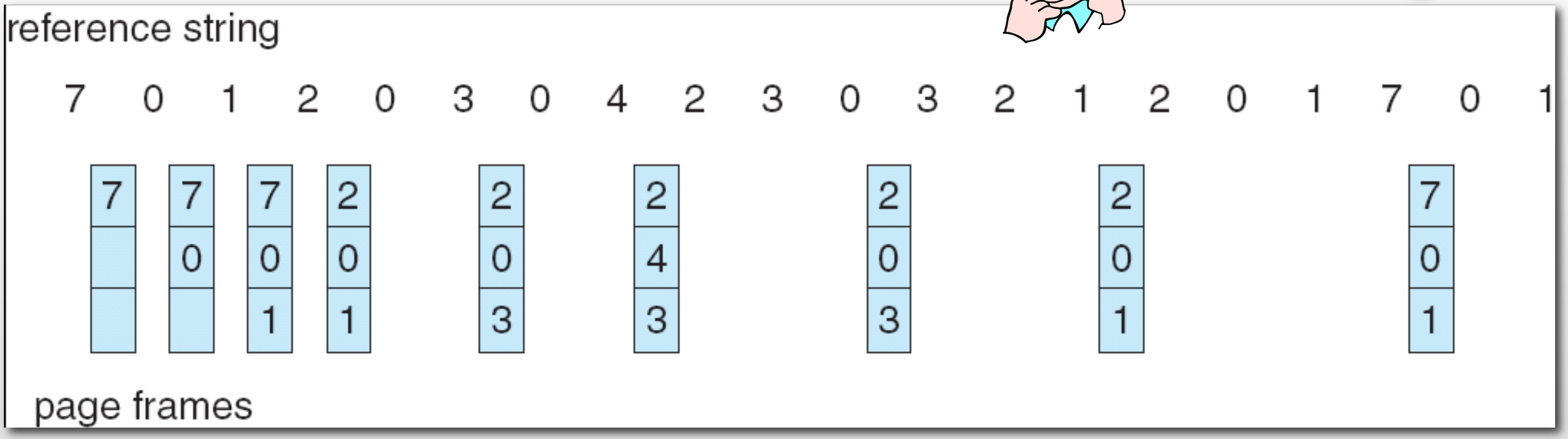
Page Replacement Algorithms: FIFO

- **F**irst-**I**n-**F**irst-**O**ut
  - replace the oldest page
  - (a.k.a. first-come-first-serve)



Page Replacement Algorithms: OPT

- **OPT**imal replacement
- replace the page that will not
  be used for longest period of time



Page Replacement Algorithms: LRU

- **L**east-**R**ecently **U**sed
- replace the page that was used
  before any other page

reference string

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

page frames

Belady's Anomaly

- One would think that increasing the number of frames leads to less faults
- However: sometimes more frames ⇒ more page faults
  - e.g., the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames

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

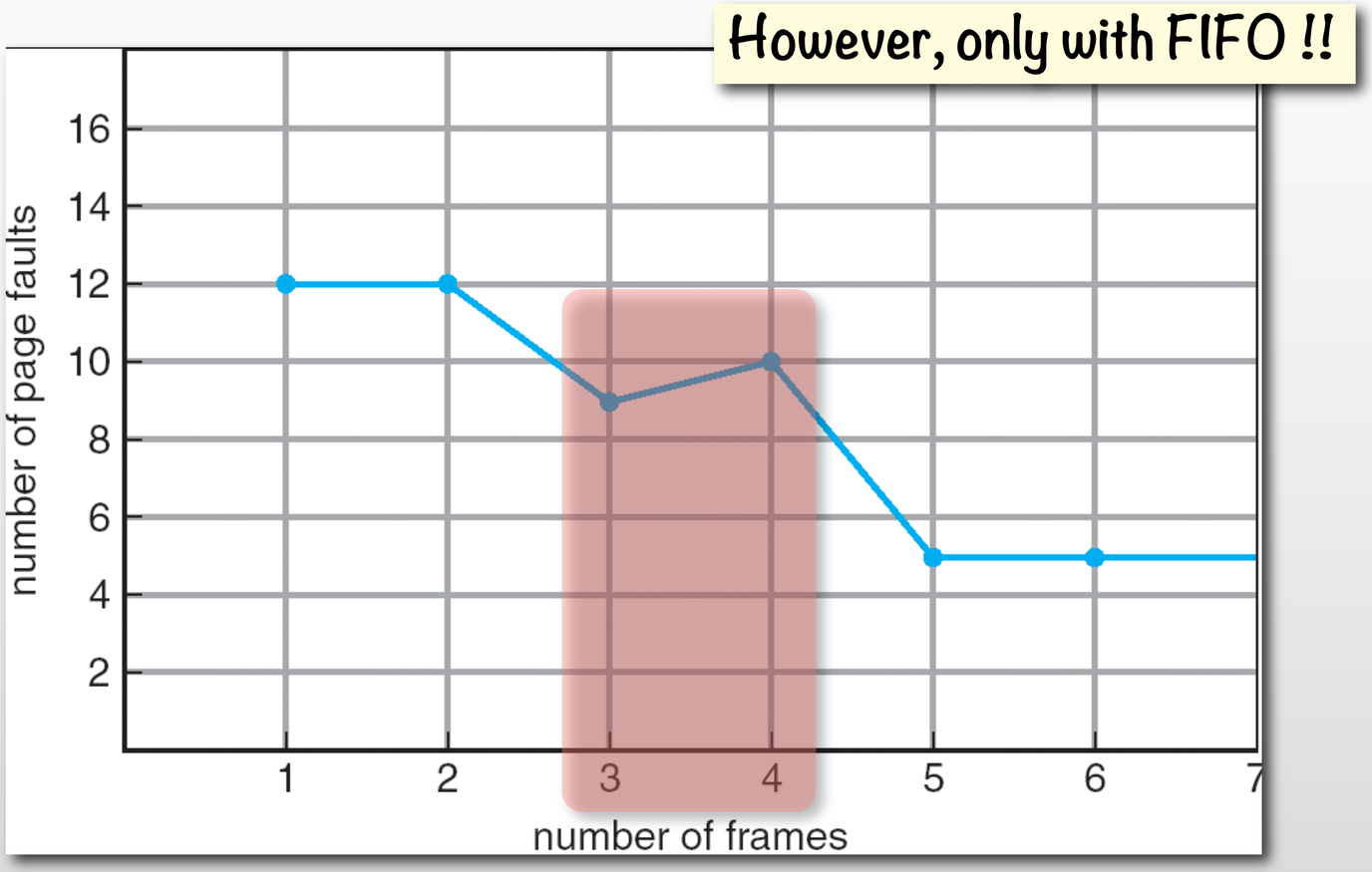
9 page faults

- 4 frames

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

10 page faults

However, only with FIFO !!



OPT Approximation

- 4 frames
  - Reference string:
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- OPT
  How do you know which page it is?
  - easy by hand when we know the future, but in real world we don't!
  - OPT is impractical, but it is used to evaluate how well another replacement algorithm performs
    - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- LRU
  - in general: LRU approximates OPT
  - more exhaustive experiments confirm, so we can claim this with confidence
- Neither OPT nor LRU suffer from Belady's anomaly

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

6 page faults

you will verify this claim in the lab

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

7 page faults

"Trivial" Implementations of LRU

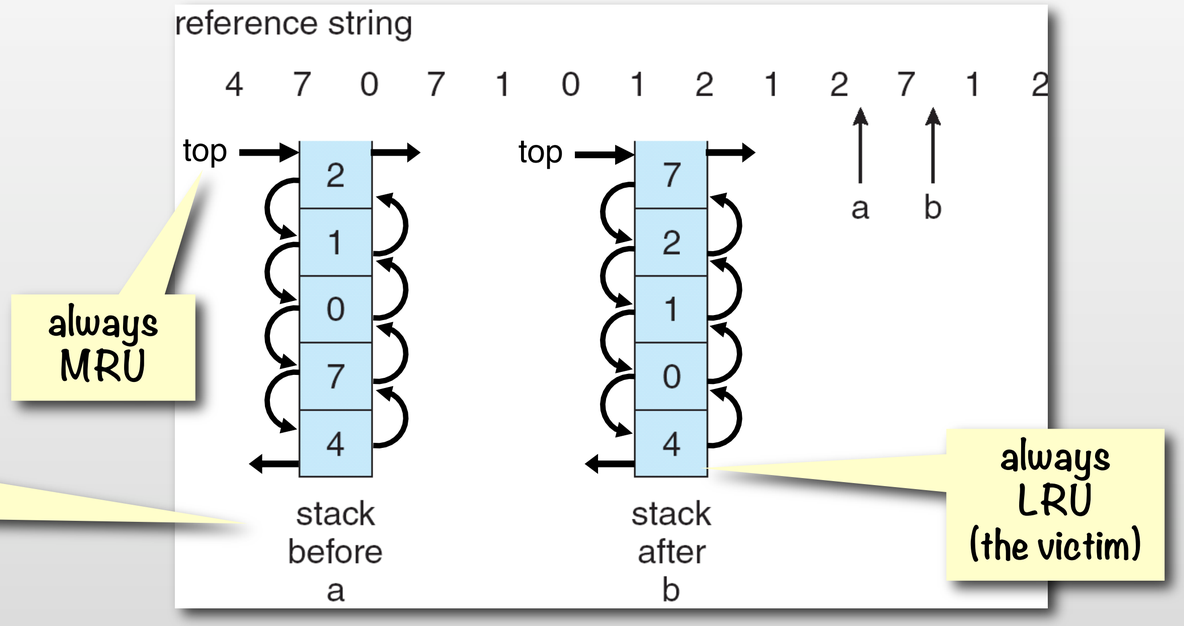
- Recall the replacement algorithm that you implemented in the lab to manage the inverse page table!
  - What did it take to find a victim?

- Timestamp implementation
  - every page entry has a tag
    - as you did in the lab!
  - every time page is referenced through this entry, timestamp the entry by copying the system clock into the tag
  - when searching for a victim page, look at the timestamps to determine which one should be replaced
  - Expensive searches!

you have already implemented this in the lab!

- Can we do better?

- LRU Quasi-Stack implementation
  - for example:
    - keep a quasi-stack of page numbers in a double link form
    - when a page is referenced move it to the top
  - requires 6 pointers to be changed each time a page is referenced
  - BUT: No search for replacement!



always MRU

you will implement this in the lab!

always LRU (the victim)

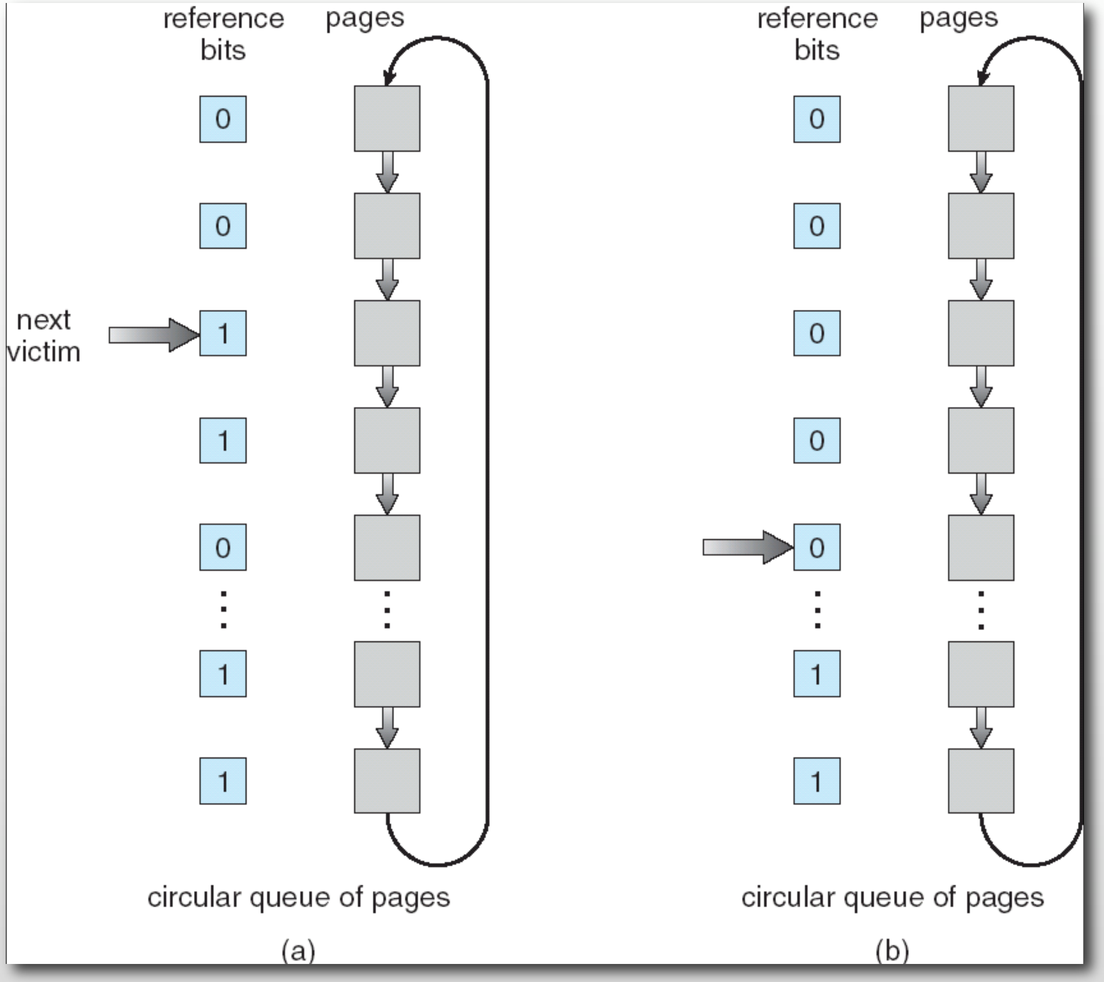
Efficient LRU Algorithms

- IDEA: approximate age/usage and then use it to compute the LRU
  - it's an approximation of the LRU algorithm

- Algorithm:
  - with each page associate a reference bit
  - initialize all reference bits to 0
  - on page reference change to reference bit to 1
  - to search for the LRU check the reference bits cyclically:
    - if the bit is 1
      - change it to 0
      - continue
    - if the bit is 0
      - set to bit to 1
      - return the page as the LRU (approximate)

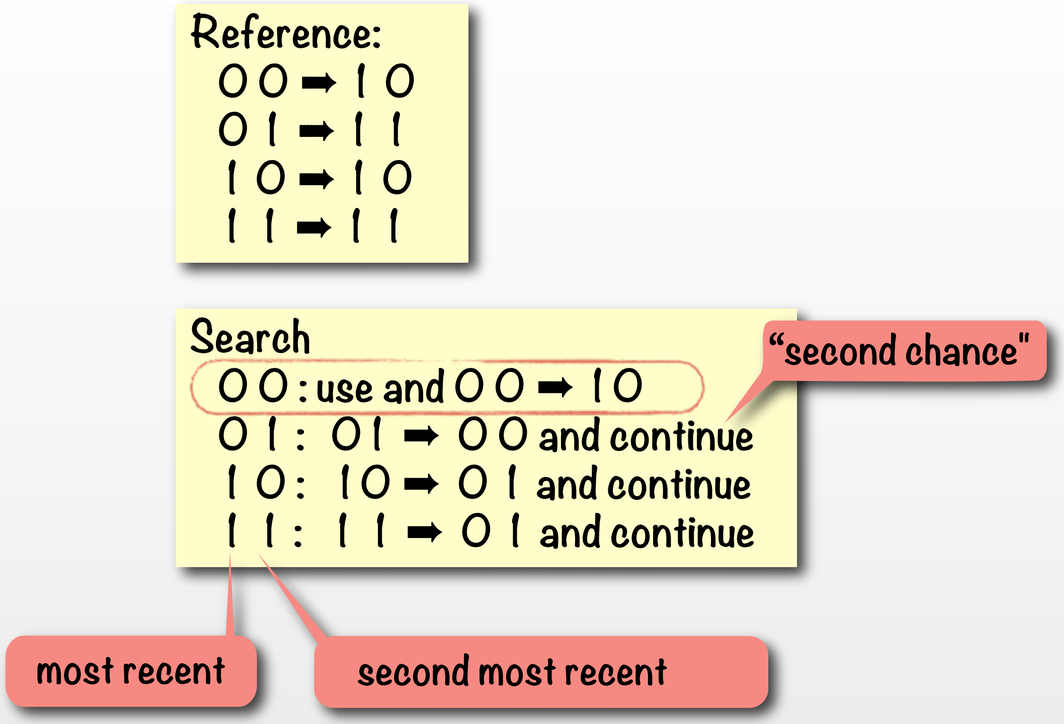
Reference
0: 0 ➡ 1
1: 1 ➡ 1

Search:
0: use and 0 ➡ 1
1: 1 ➡ 0



Efficient LRU Algorithms: Second Chance

- Second chance
  - need second reference bit
    - simulate clock by shifting the first reference bit to the second one
  - if page to be replaced (i.e., the first reference bit is 0) has the second reference bit = 1 then:
    - set the second reference bit to 0
    - leave page in memory
    - replace next page (in clock order) with bit 0, subject to same rules
- More variations possible with more bits



Counting Replacement Algorithms

- A variety of other algorithms have been tried by researchers over the years
- In **counting algorithms**, the system keeps a counter of the number of references that have been made to each page

- **L**east **F**requently **U**sed (**LFU**) Algorithm
  - replaces page with smallest count

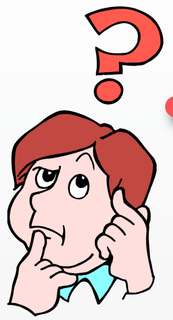
- **M**ost **F**requently **U**sed (**MFU**) Algorithm
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used

- Expensive to implement
  - e.g., expensive sorting or search

Experiments have shown that these algorithms do not approximate OPT very well

Frame Allocation

- Replacement scopes
  - **Global replacement**
    - process selects a replacement frame from the set of all frames
    - one process may be given a frame taken away from another
  - **Local replacement**
    - each process selects only from its own set of allocated frames
    - **Fixed allocation**
      - Either equal allocation
        - give each process the same number of frames
        - equality is not always good!
      - Or proportional allocation
        - allocate according to the size of process
    - **Dynamic allocation**
      - e.g., priority allocation
      - Use a proportional allocation scheme using priorities rather than size
      - If process generates a page fault, select for replacement a frame from a process with lower priority number



How to distribute available frames between the processes?

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

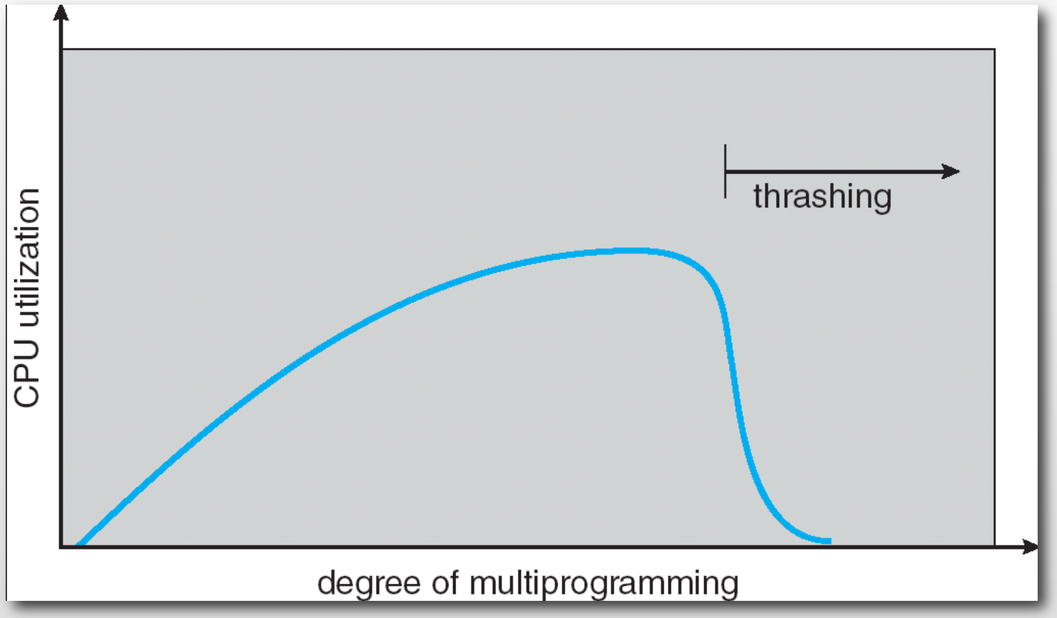
$a_1 = \frac{10}{137} \times 64 \approx 5$

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

May also combine the size and priority in a similar scheme

Page Thrashing

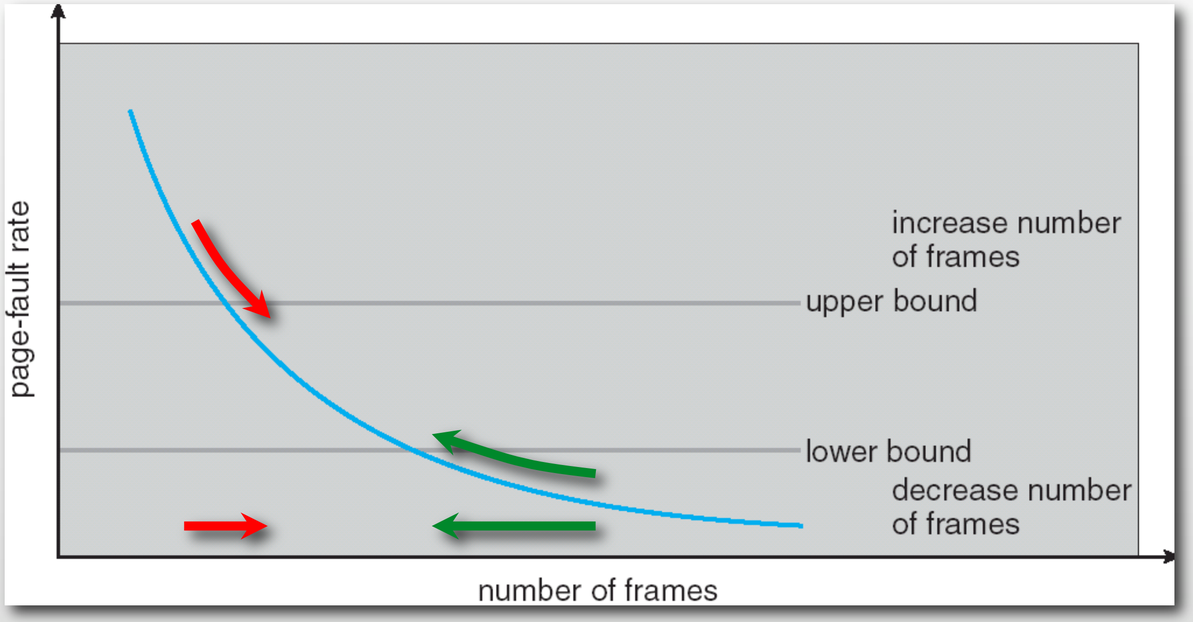
- If a process does not have sufficient number of pages in memory, the page-fault rate is very high
- The OS may end up being too busy swapping pages (an I/O activity) in and out with little time left for anything else
- This leads to a chain reaction:
  - low CPU utilization (due to mostly I/O activity)
    - operating system thinks that it needs to increase the degree of multiprogramming
    - another process added to the system
    - more page faults
    - lower CPU utilization
  - the cycle is closed, so the system basically stops doing any useful work



Trivial Page-Fault Frequency Scheme

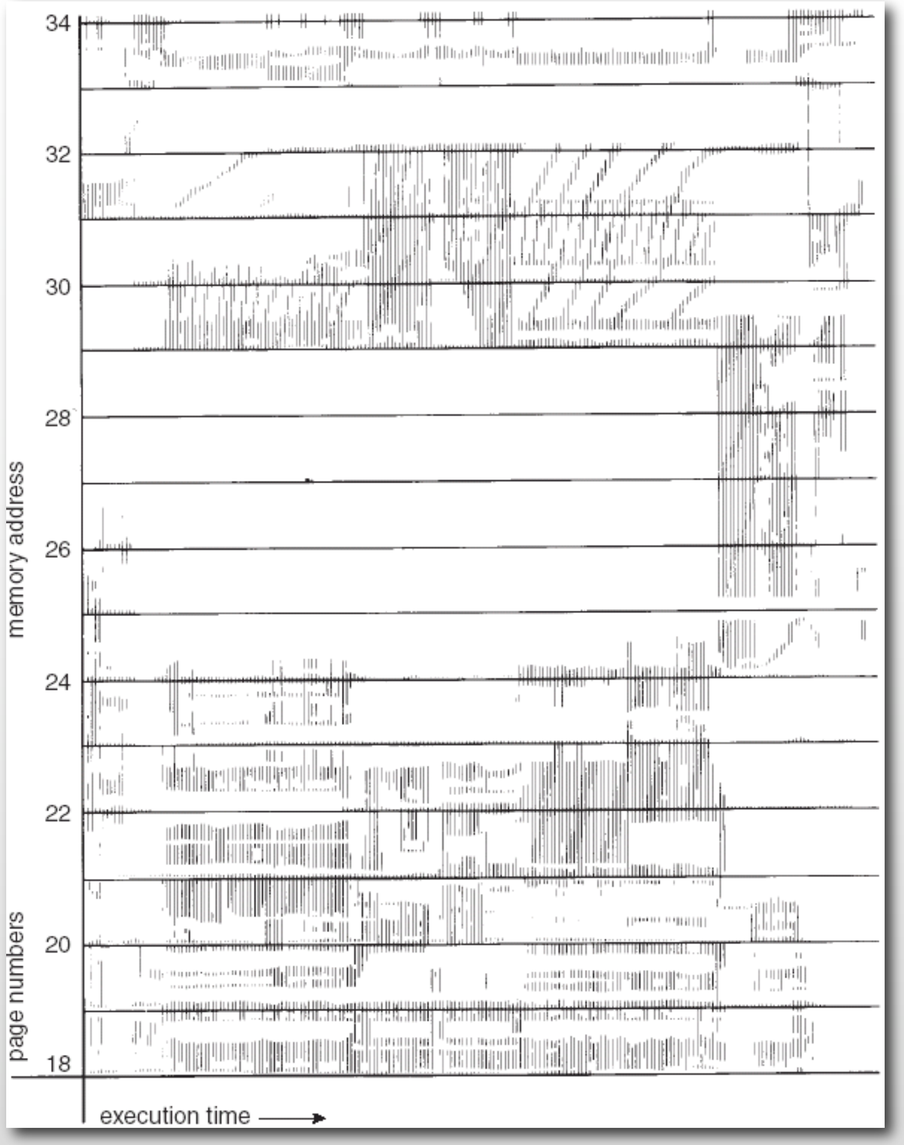
- Simple, rigid, but efficient allocation scheme
- Establish “acceptable” page-fault rate
  - if actual rate falls below the lower bound ➠ process loses frames
  - if actual rate rises above the upper bound ➠ process gains frames
- May lead to an oscillation reaction
  - the system is oscillating between the two allocation extremes

Can we do better?



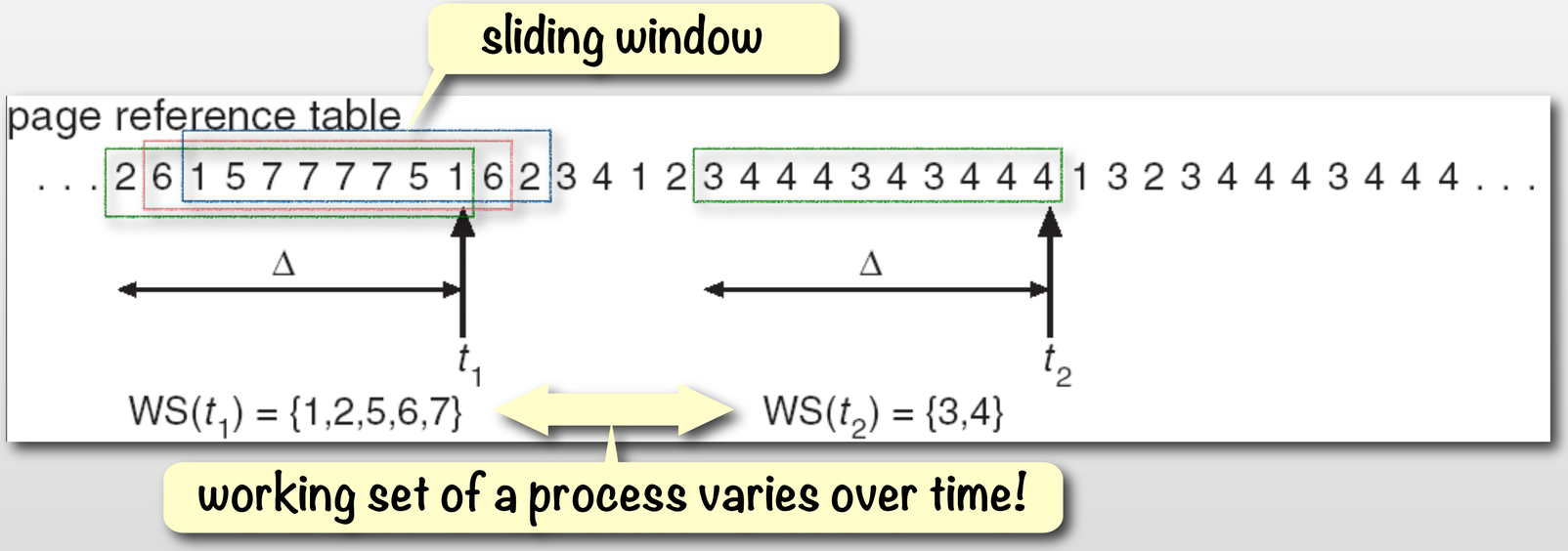
Locality Model

- Caching works if we can predict which pages will be referenced
- Experiments show that each process has a locality model
  - references to addresses are localized in time periods
  - over time, process execution migrates from one address locality to another
  - localities may overlap
- If we know the size of the address locality at a given time, then we can allocate suitable number of frames for the process, so all pages in the current locality are in the memory (swapped in)
- How can we measure the size of the locality of a process?



Working-Set Model

- **Working set** is used to have sets of pages available to prevent thrashing
  - it tries to build an approximation of the locality model
- $\Delta \equiv$ **working-set window** $\equiv$ a fixed number of page references
  - e.g., 10,000 instructions
- **WSi** (working set of process Pi)
  = total number of pages referenced in the most recent window Δ



Working-Set Model

- Issues
  - if Δ too small ➠ will not encompass entire locality (will need to load)
  - if Δ too large ➠ will encompass several localities (waste of frames)
  - if $\Delta = \infty$ ➠ will encompass entire program

total number of pages needed in memory

- $D = \Sigma$ WSi ≡ total demand frames for all processes

recall: total number of frames in memory

- If D > m ➠ thrashing
  - potential remedy: if D > m, then suspend one of the processes

Efficient Tracking of the Working Set

- Trivial implementation is not efficient
  - so, approximate working set with interval timer + a reference bit
    - basically, build a history of the references in the last time slot

  - Example: Δ = 10,000
    - timer interrupts after every 5000 time units
    - keep in memory 2 reference bits for each time slot
    - whenever a timer interrupts, copy the reference bit to the second bit and set it to 0
    - now, if one of the bits in memory = 1 ➠ page is in the working set for this Δ
      - e.g., 00 ➠ no references in Δ; 01 or 10 ➠ one reference; 11 ➠ two references

- Not completely accurate
  - no ordering data from within the 5000 interval
  - potential improvement: use 10 bits and interrupt every 1000 time units
    - price: adding bits leads to increased overhead

TLB Reach

- Recall Translation Look-aside Buffer (TLB)
  - hardware-assisted cache for storing page references
  - the amount of memory accessible from the TLB

    TLB Reach = (TLB Size) X (Page Size)

- Ideally, the working set of each process is stored in the TLB
  - otherwise there is a high degree of TLB misses
    - as you recall, that more than doubles the memory EAT

Page Size

- Alleviate the problem by
  - increasing the page size
    - this may lead to an increase in internal fragmentation as not all applications require a large page size
  - provide multiple page sizes
    - this allows applications that require larger page sizes the opportunity to use them without an increase in overall memory fragmentation
      - more complex

- Page size
  - size selection must take into consideration:
    - fragmentation (external and internal)
    - page table size
    - I/O overhead
    - locality

Pre-Paging

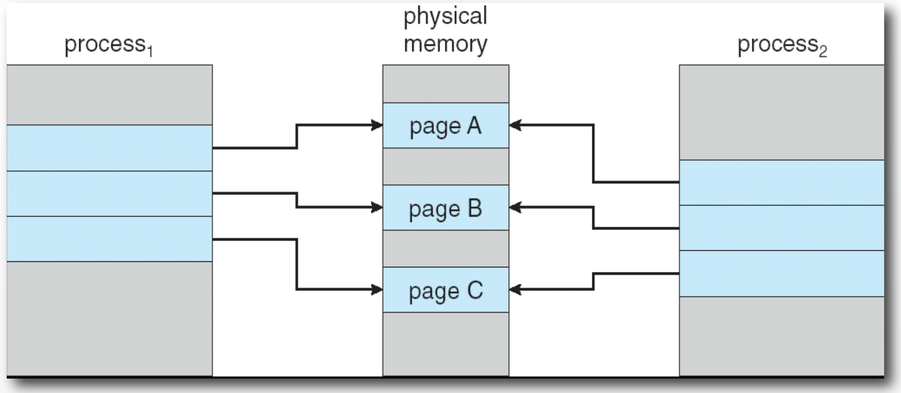
- Pre-loading pages to reduce the large number of page faults that occurs at process startup pre-page all or some of the pages a process will need, before they are referenced

- Dilemma: if pre-paged pages are not used as expected, then I/O and memory was wasted
  - assuming that σ pages are pre-paged and α is a probability that a pre-loaded page is used, is the gain of avoiding ($\sigma * \alpha$) page faults more, or less than the cost of pre-paging ($\sigma * (1- \alpha)$) unnecessary pages?
  - if α near zero ➠ pre-paging is not good

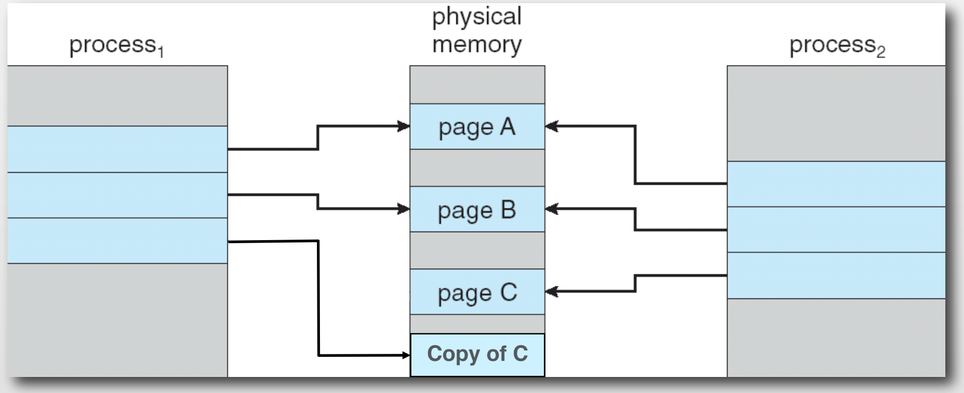
If we collect page fault statistics, then we can approximate α and use it in the pre-paging algorithm.

Copy-on-Write (COW)

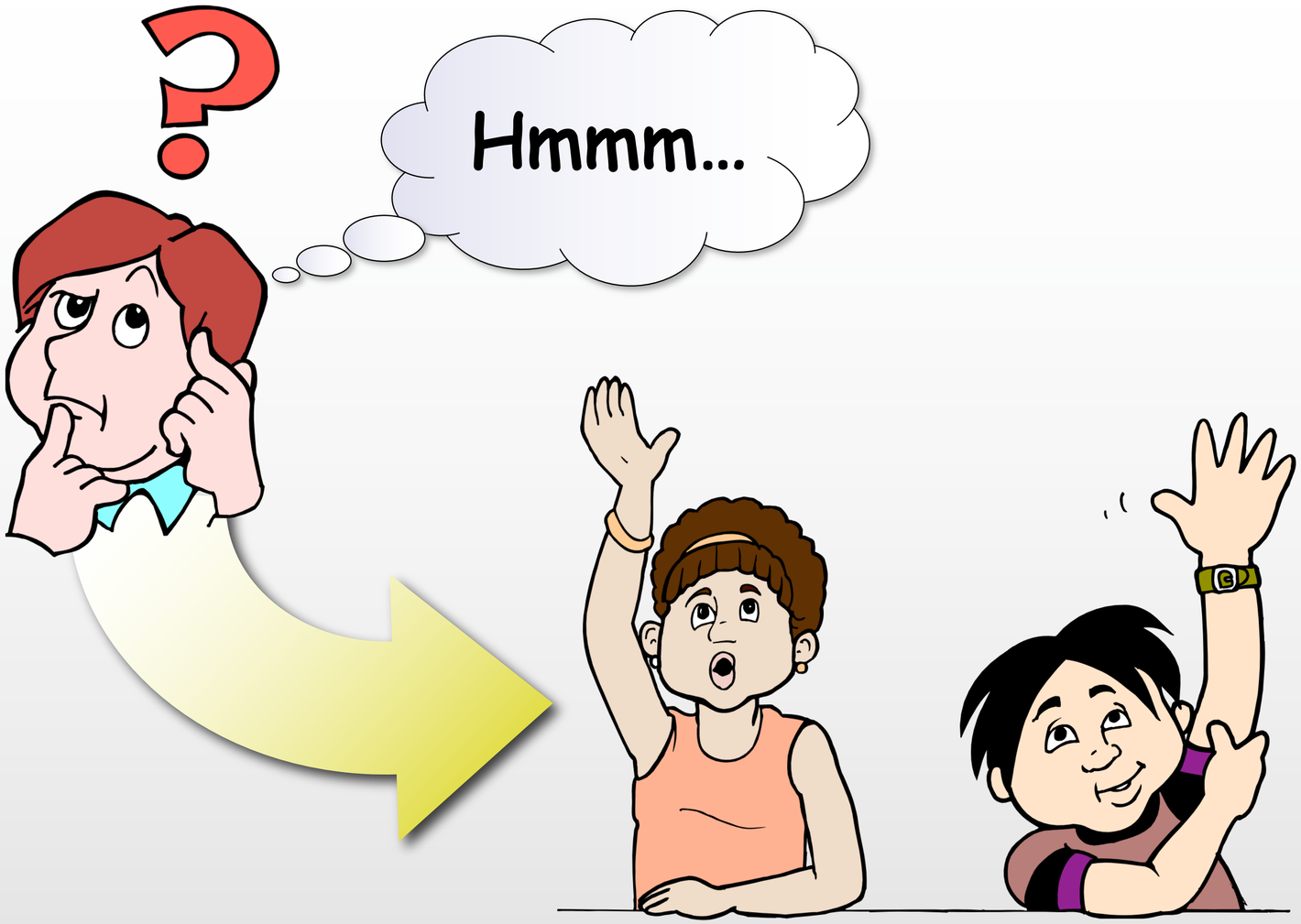
- One way to help with efficiency
  - reuse, reuse, reuse…
- Copy-on-Write (COW) allows two processes to initially share the same pages in memory
  - e.g., the parent and the child while forking a process
- If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a pool of zeroed-out pages



Before Process 1 Modifies Page C



After Process 1 Modifies Page C



COMP362 Operating Systems

Prof. AJ Bieszczad