CS343: Operating System

Virtual Memory, Demand Paging, Page Replacement

Lect34 : 2nd Nov 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Memory Management
  - Paging
  - Virtual memory
  - Demand Paging
  - **Page Replacement**
  - **Frame Allocation**

Memory Allocation: Top Down

Assume 4GB of RAM,
Avg Process Size 200MB
4 Segments/Process

Very low complexity

Very High Mem Wastage

Finding Hole
First Fit
Worst fit
Best Fit

Process Continuous Allocation
10 process and 20 holes

Segment continuous Allocation
40 segment and 80 holes

low complexity

High Memory Wastage

Paging: 200MB process, 4KB page
Two Issue: pages and frames

Very Low Mem Wastage

High Complexity: 51200 pages/Process

Loading 51K pages in 10^6 frames

High Complexity: $4GB/4KB=2^{20}=10^6$ frames

SOL

Demand Paging Load the required 50-100 pages and load as an when necessary

4GB paged

SOL

Allocate 100-200 frames per process: reduce search Space

Demand Paging

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

Demand Paging

- Similar to paging system with swapping
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Page Table : Some Pages Are Not in Memory

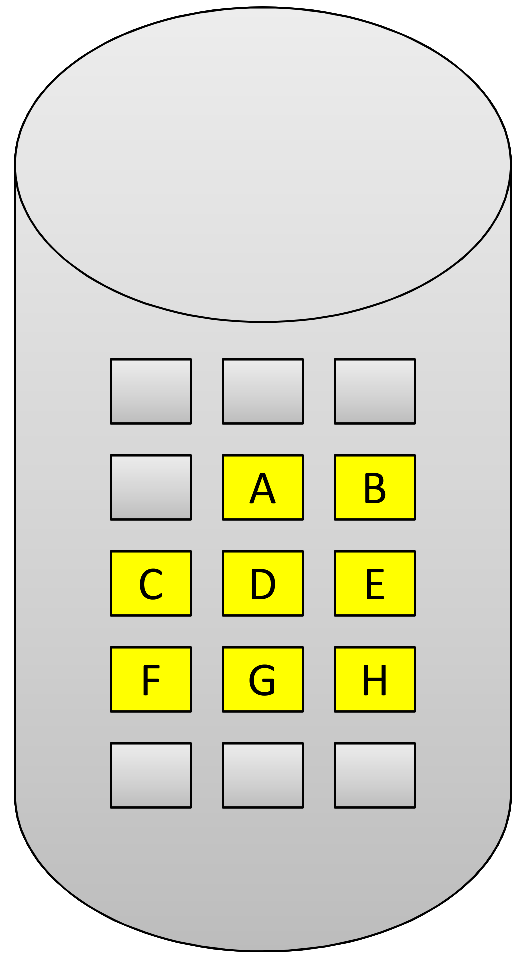
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

Logical Memory

#P	#F	V
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

Page Table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	



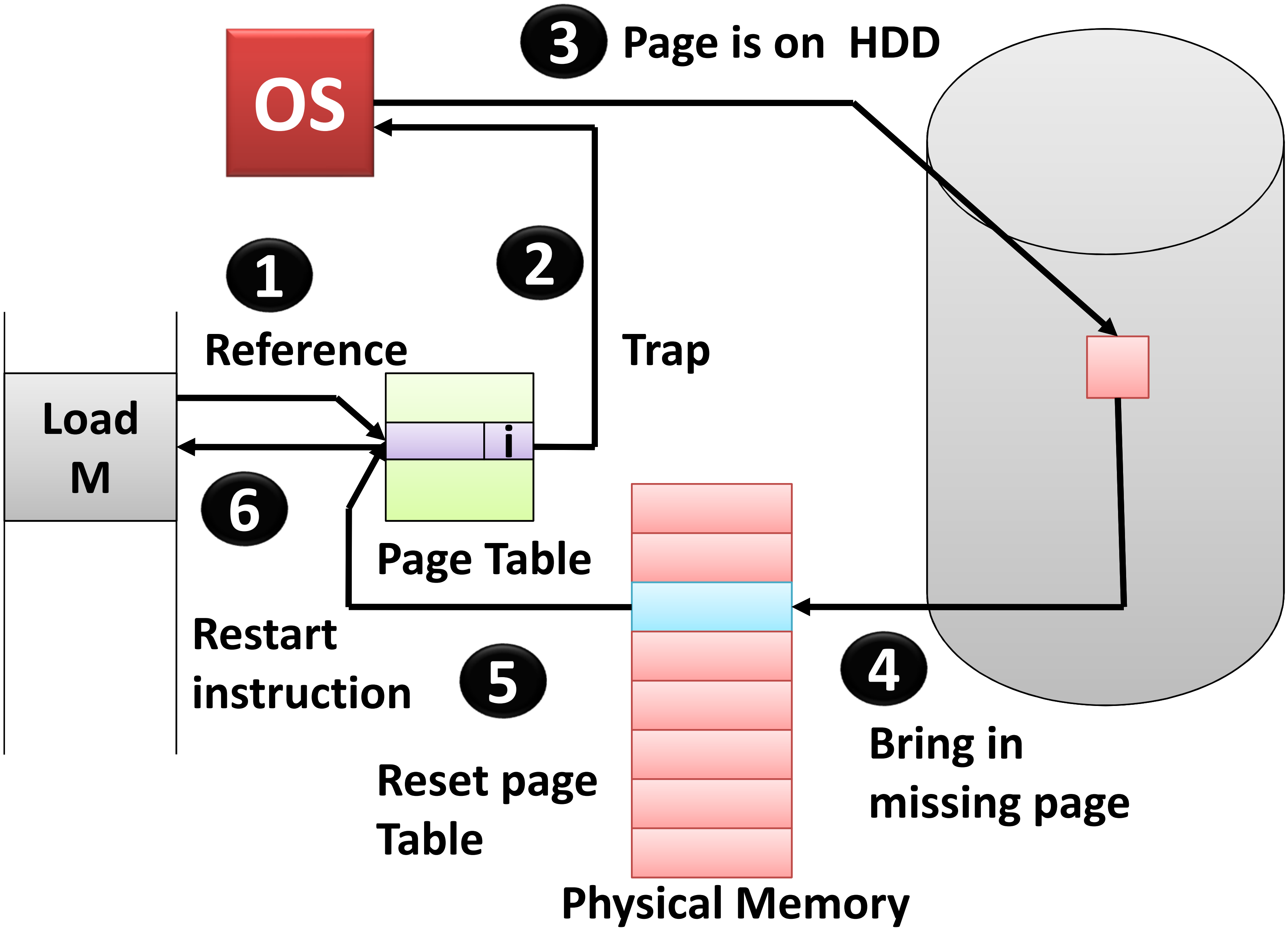
Main memory

Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Page reference
2. OS looks at page table to decide:
   - **Invalid reference $\Rightarrow$ abort**
   - Just not in memory
3. Find free frame
4. Swap page into frame via disk operation
5. Indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused page fault

Steps in Handling a Page Fault



Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access} + p\,(\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of **40!!**

Demand Paging Example

- **Virtual memory similar to caching in processor level**
  - **Except higher page fault rate is not acceptable**
  - **Page fault rate < < Cache miss rate**
- If want performance degradation < 10 percent
  - $220 > 200 + 7{,}999{,}800 \times p$
    $20 > 7{,}999{,}800 \times p$
  - $p < .0000025$
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

Demand Paging Optimizations

- Demand page in from program binary on disk, but discard rather than paging out
  - **Why to swap-out read only data?**
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system

Demand Paging Optimizations

- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

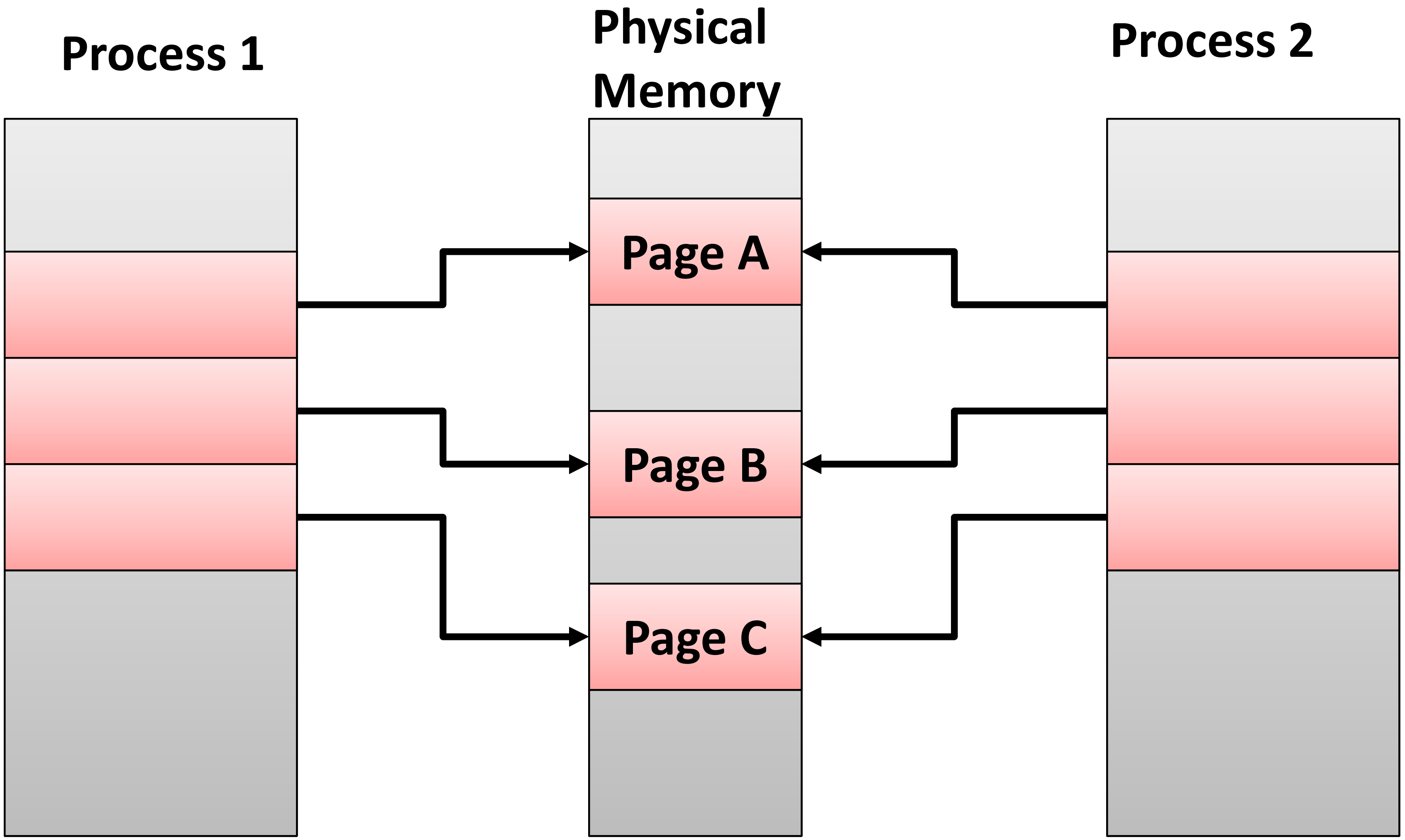
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - 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

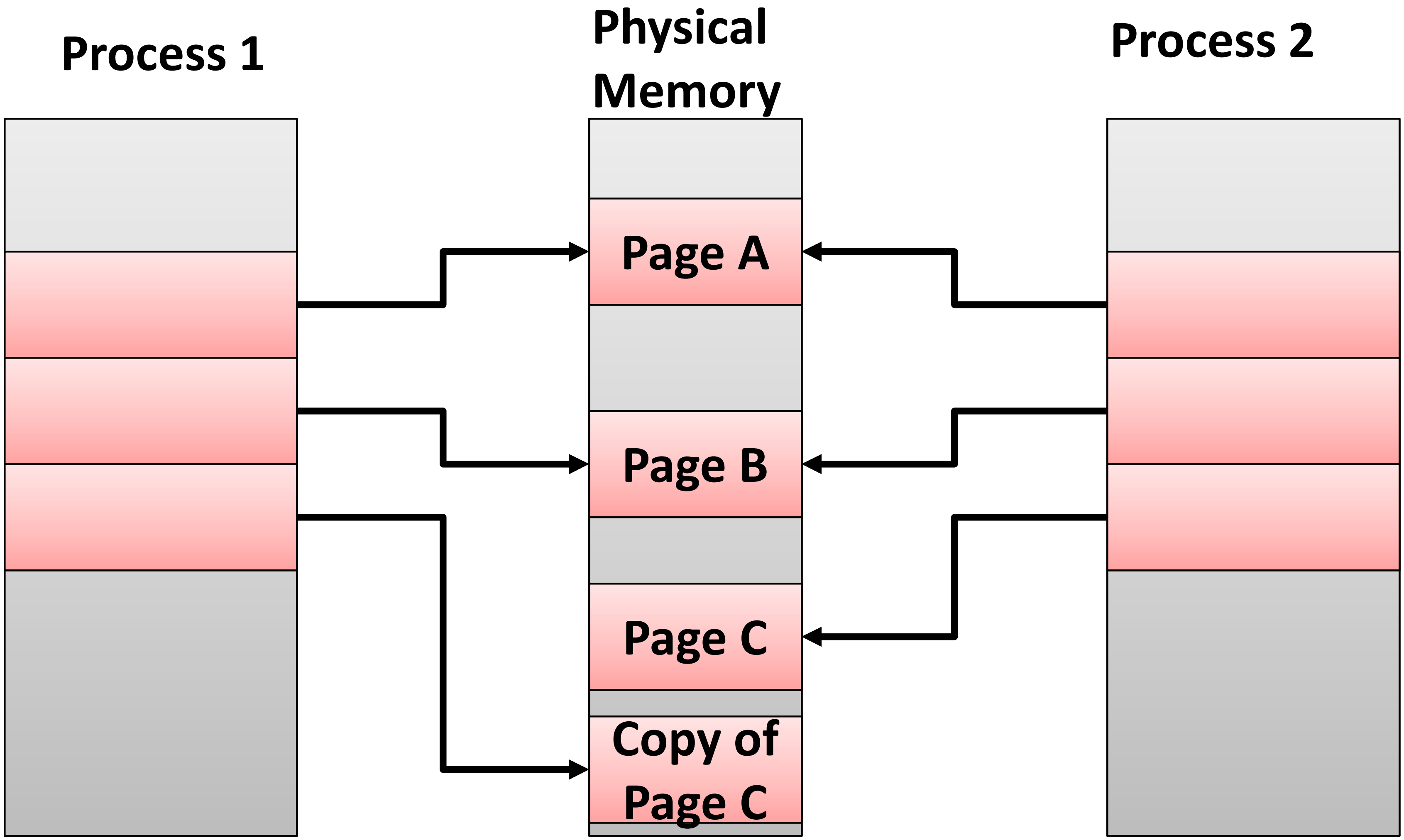
Copy-on-Write

- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don’t want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What Happens if : no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

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

Page Replacement Example

- 2 Process : 4 logical pages per process
- 8 frame, but 2 allocated for OS
- 6 are available for user
  - Need to run 2 process of total 8 logical pages

Need For Page Replacement

	0	H
PC →	1	Load K
	2	J
	3	M

#P	#F	V
0	3	v
1	4	v
2	5	v
3		i

PT for P1

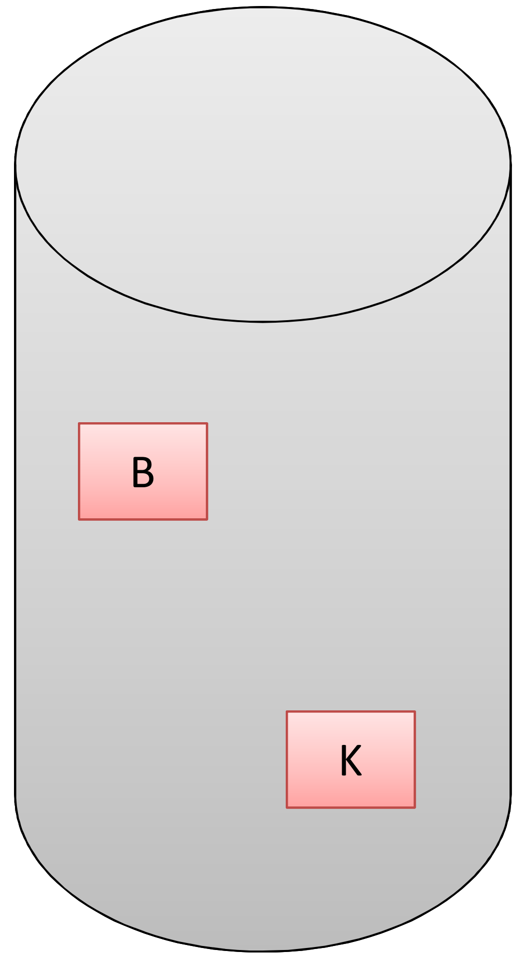
0	A
1	B
2	D
3	E

#P	#F	V
0	6	v
1		i
2	2	v
3	7	v

PT for P2

0	Monitor
1	
2	D
3	H
4	Load K
5	J
6	A
7	E

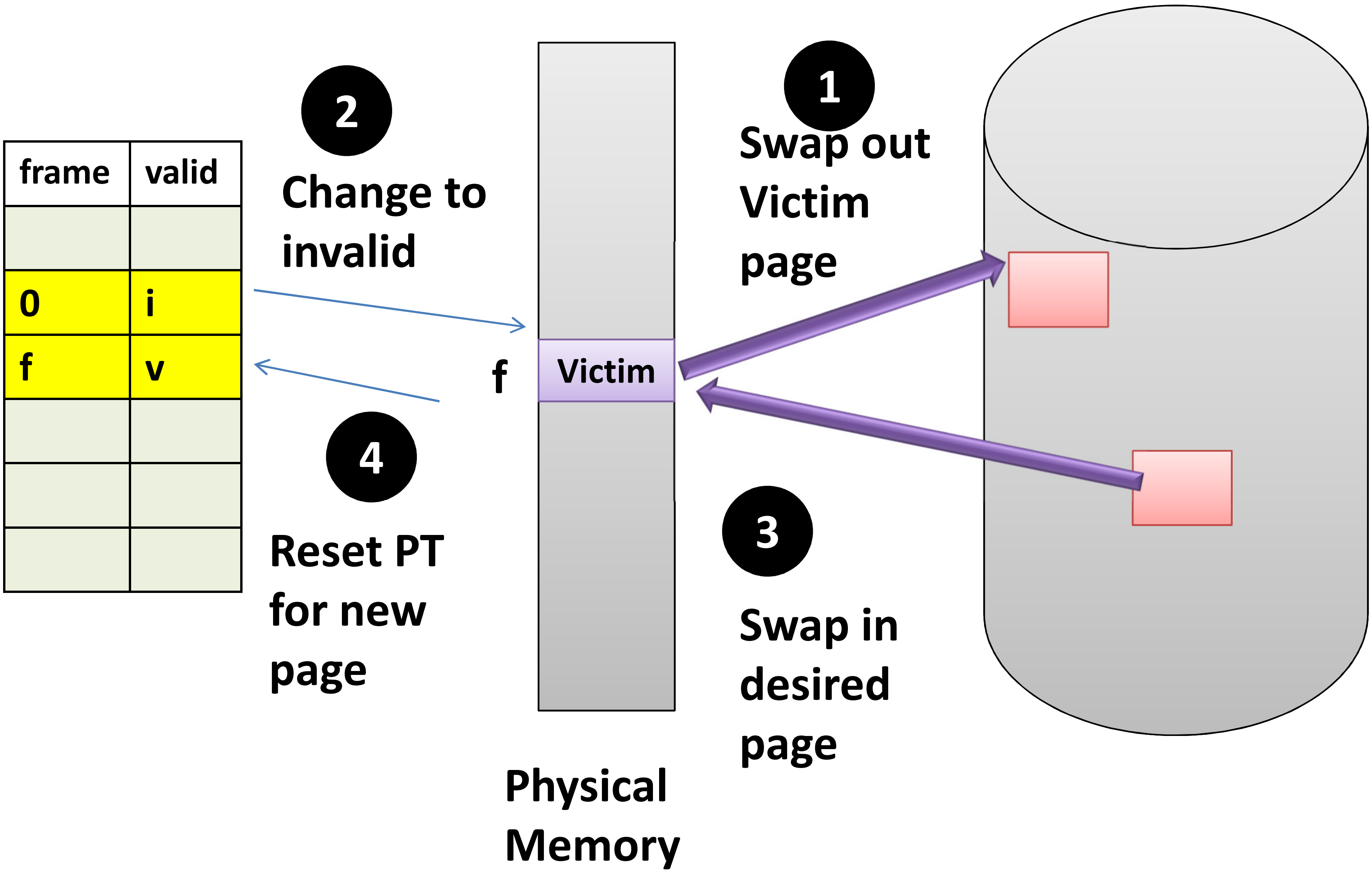
Main memory



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**
  - Write victim frame to disk if dirty
- Bring the desired page into the (newly) free frame; update the page and frame tables
- Continue the process by restarting the instruction that caused the trap
- Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



Page and Frame Replacement Algorithms

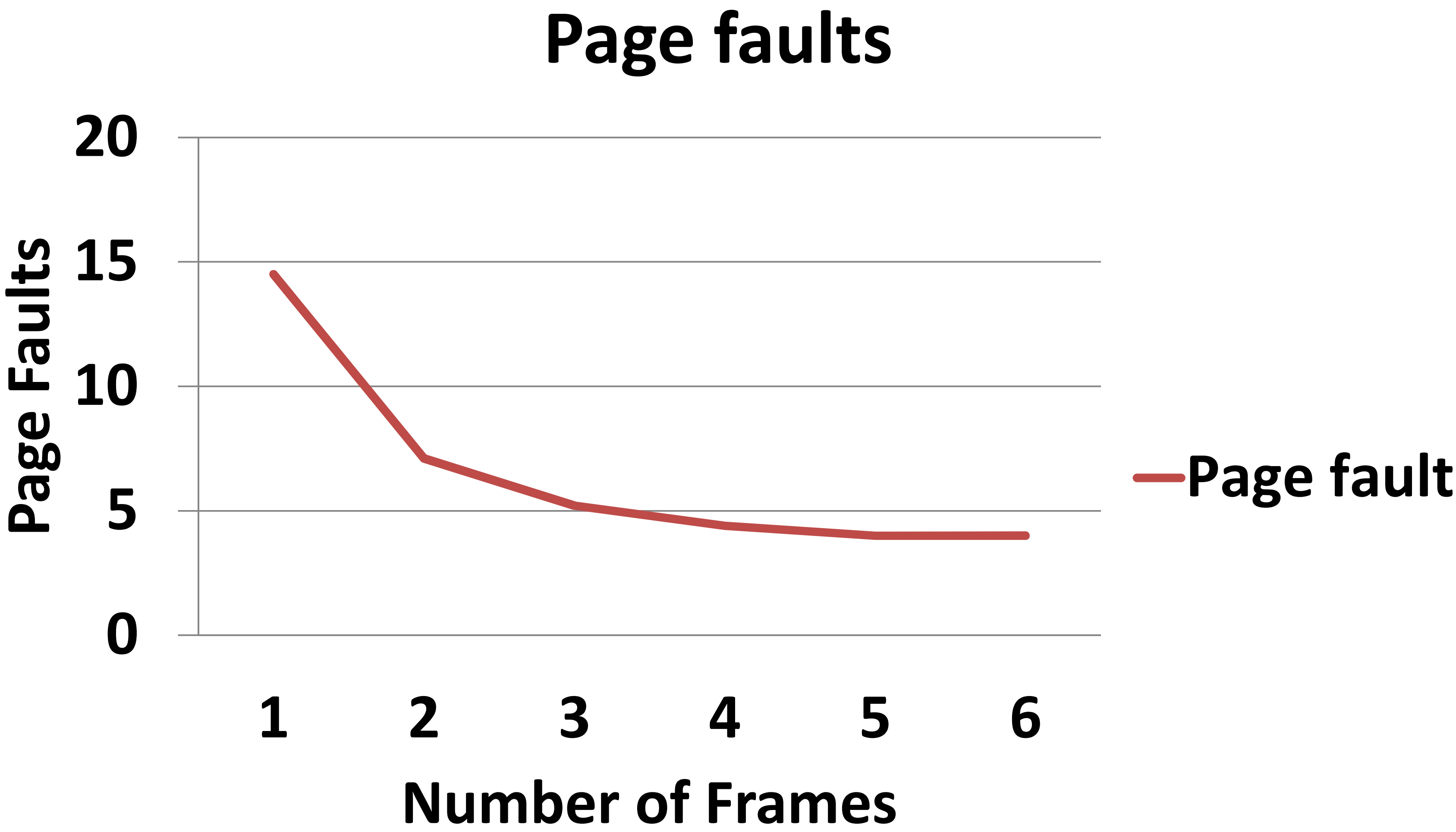
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

Page and Frame Replacement Algorithms

- 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
  - Results depend on number of frames available
- **Example of reference string** of referenced page numbers is

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

Graph of Page Faults Versus The Number of Frames



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)

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

15 page faults

First-In-First-Out (FIFO) Algorithm

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

Example Belady's Anomalies : with FIFO

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

3 frames

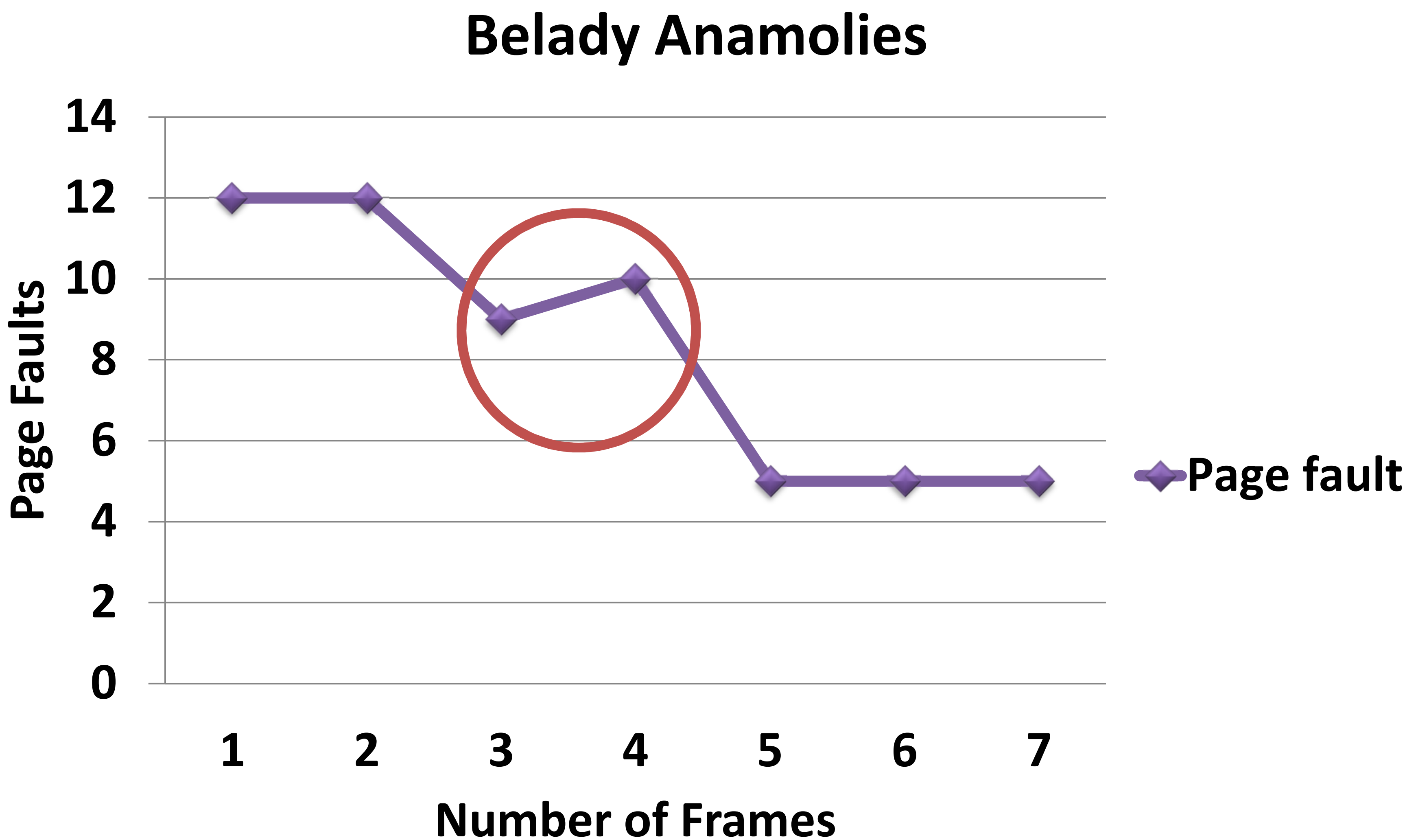
9 page faults

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

4 frame

10 page faults

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- **How do you know this?**
  - **Can't read the future**
- Used for measuring how well your algorithm performs
- **Suppose you are running your program 2nd time with same data**
  - **You know what will happen in the program**

Optimal Algorithm

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

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

Least Recently Used (LRU) Algorithm

- 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
- Use past knowledge rather than future

LRU Algorithm

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

- 12 faults – better than FIFO (15) but worse than OPT (9)
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm Cont.

- 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 : Bad**

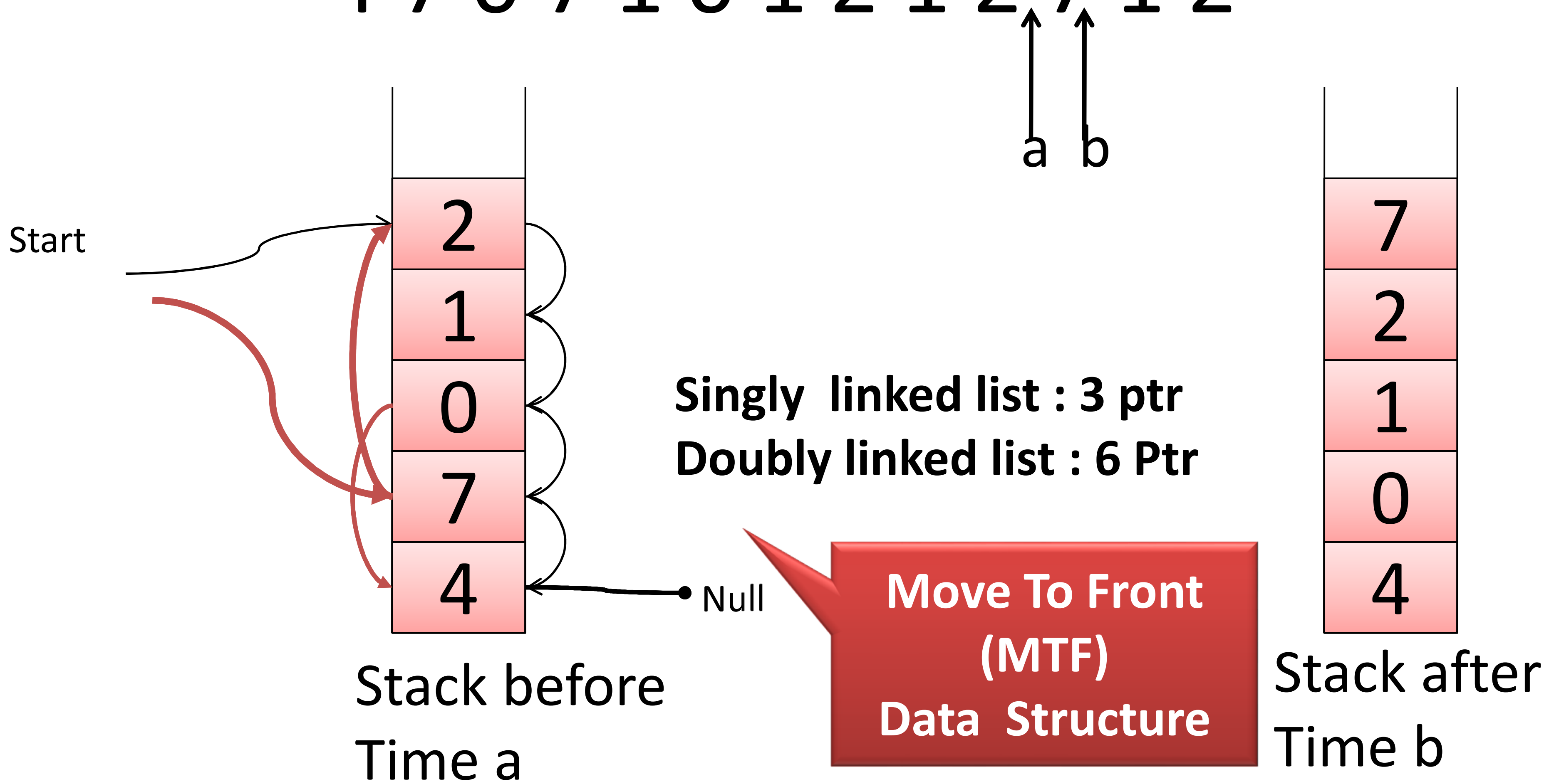
LRU Algorithm Cont.

- 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 (OK)**
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use Of A Stack to Record Most Recent Page References

Reference string

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



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

- Suppose every page associated with 8 bits reference bit (with a shift reg)
- When a reference made bit set to 1
- Every page reference bit shifted
- For last 8 references
  - Every page will get different value of shift reg
  - 00000000 in a Page : signifies no reference
  - 11111111 in a Page : signifies reference all the time
  - 10010101 is LRUed then 00101000
- **All page Shift reg need to shift : Bad Part**

LRU Approximation Algorithms

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit and
  - **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

LRU Approximation Algorithms

- **Second-chance algorithm with modify bit**
- 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
- May be combined with modified bit
  - Ref bit (0) + Modified (0) ==> best guy to replace
  - 01 ➔ not recently used but modified ➔ not good
  - 10 => recently used but clean => probably will be used again
  - 11 ➔ used and modfied ➔ bad candidate

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease 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
  - 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

Page-Buffering Algorithms

- Possibly, keep list of modified pages
  - When backing store 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
  - **Same as Victim Buffer**

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases

Applications and Page Replacement

- 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
- OS can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

Page Replacement Algorithm

- Input : Reference string, number of frame
- Output: number of page fault

- **Timing information was missing in Reference string**
- **How to allocate the frame for process?**
  - **Static (fixed size for process life time)**
  - **Dynamic**

Thanks