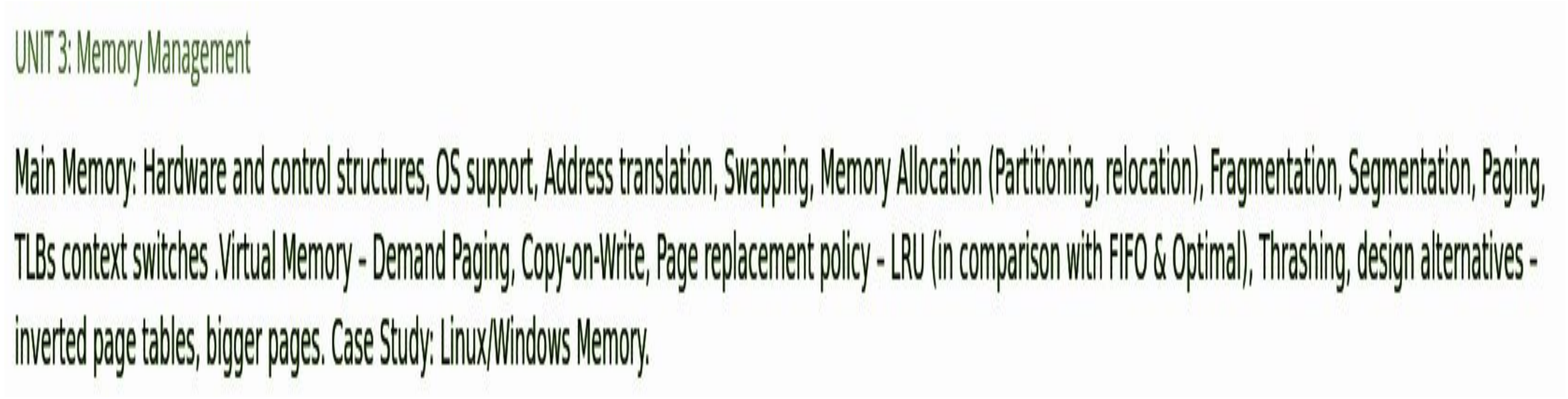
OPERATING SYSTEMS

Memory Management - 7

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

Course Syllabus - Unit 3


UNIT 3: Memory Management

Main Memory: Hardware and control structures, OS support, Address translation, Swapping, Memory Allocation (Partitioning, relocation), Fragmentation, Segmentation, Paging, TLBs context switches .Virtual Memory - Demand Paging, Copy-on-Write, Page replacement policy - LRU (in comparison with FIFO & Optimal), Thrashing, design alternatives - inverted page tables, bigger pages. Case Study: Linux/Windows Memory.

OPERATING SYSTEMS

Course Outline - Unit 3

25	8.1	Main Memory: Hardware and control structures, OS support, Address translation,	8	64.2
26	8.2-8.3	Swapping, Memory Allocation (Partitioning, relocation), Fragmentation,	8	
27	8.4	Segmentation	8	
28	8.5	Paging	8	
29	8.5	TLBs context switches	8	
30	8.6	Structure of page tables	8	
31	8.6.3,8.7	design alternatives – inverted page tables, bigger pages	8	
32	9.1-9.2	Virtual Memory – Demand Paging	9	
33	9.3,9.4.1-9.4.3	Copy-on-Write, Page replacement: Basic page replacement (FIFO page replacement and optimal page replacement)	9	
34	9.4.4, 9.5	LRU Page replacement, Allocation of frames	9	
35	9.6	Thrashing	9	
36	9.10	Case Study: Linux/Windows Memory	9	

Topic Outline

- **Virtual Memory - Background**
- **Virtual Memory that is Larger Than Physical Memory**
- **Virtual Address Space**
- **Shared Library Using Virtual Memory**
- **Demand Paging**

Topic Outline

- **Swapping - Basic Concepts**
- **Valid-Invalid Bit**
- **Page Table When Some Pages Are Not in Main Memory**
- **Page Fault**
- **Steps in Handling a Page Fault**

Topic Outline

- Aspects of Demand Paging
- Performance of Demand Paging
- Demand Paging Example
- Demand Paging Optimizations
- Copy-on-Write

Topic Outline

- First-In-First-Out (FIFO) Algorithm
- FIFO illustrating Belady's Anomaly
- Optimal Page Replacement Algorithm
- Least Recently Used (LRU) Algorithm
- Use of a Stack to Record Most Recent Page References

Topic Outline

- **LRU Algorithm Implementation**
- **LRU Approximation Algorithm**

Topic Outline

- **Virtual Memory - Thrashing**
- **Demand Paging and Thrashing**
- **Working-Set Model**
- **Keeping Track of the Working Set**
- **Page-Fault Frequency**

Virtual Memory - Background

- Code needs to be in memory to execute, but entire program rarely used
    - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
    - Program no longer constrained by limits of physical memory
    - Each program takes less memory while running -> more programs run at the same time
        - Increased CPU utilization and throughput with no increase in response time or turnaround time

Virtual Memory - Background

- 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
  - More programs running concurrently
  - Less I/O needed to load or swap processes

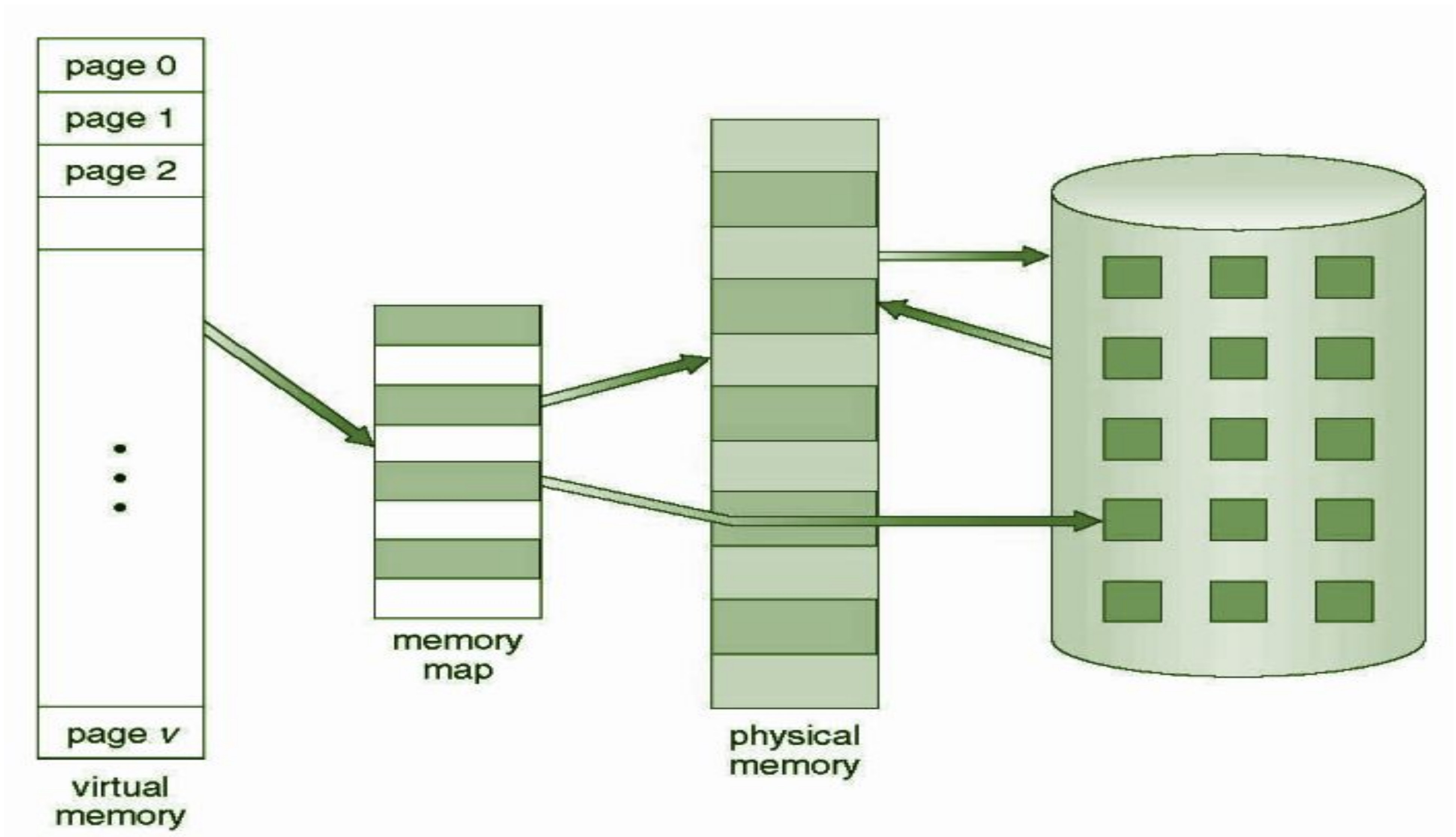
Virtual Memory - Background

- Virtual address space => logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical

Virtual Memory - Background

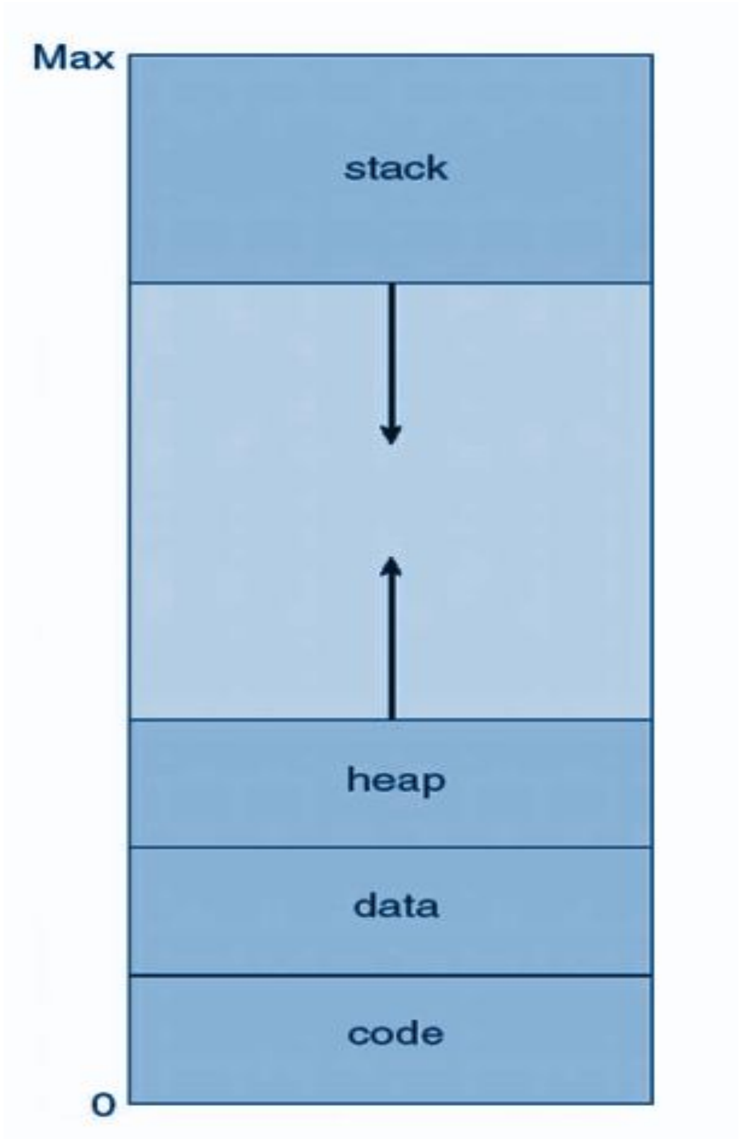
- Virtual memory can be implemented using:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



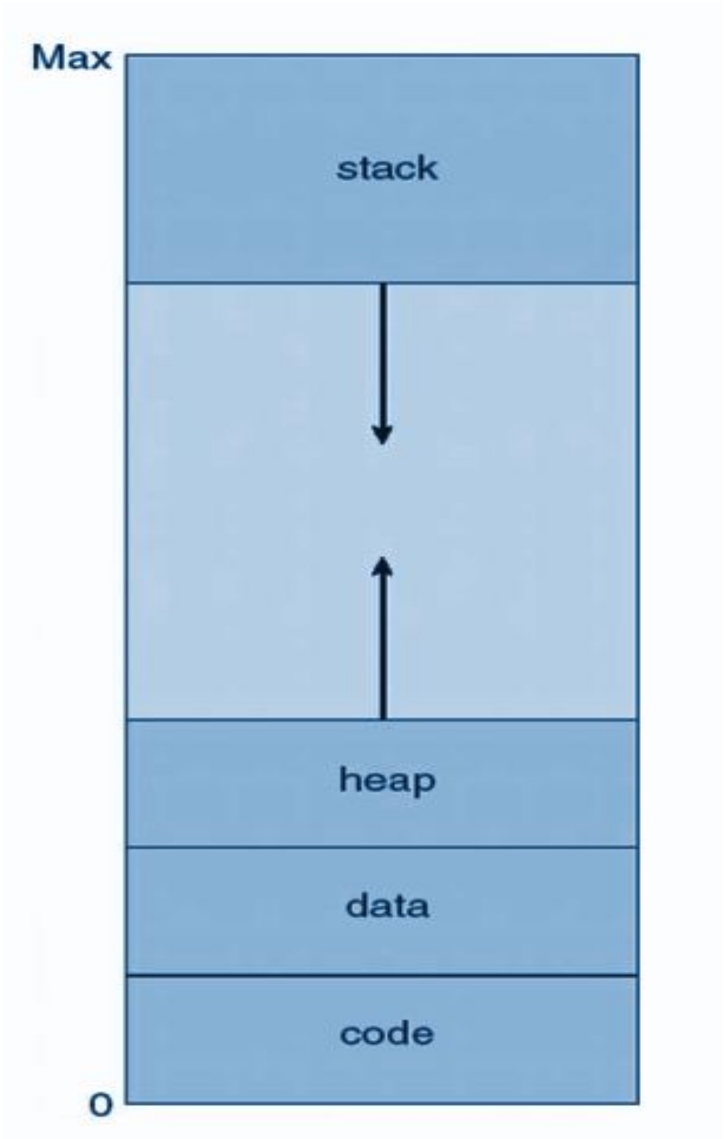
Virtual Address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole or CC
  - No physical memory needed until heap or stack grows to a given new page

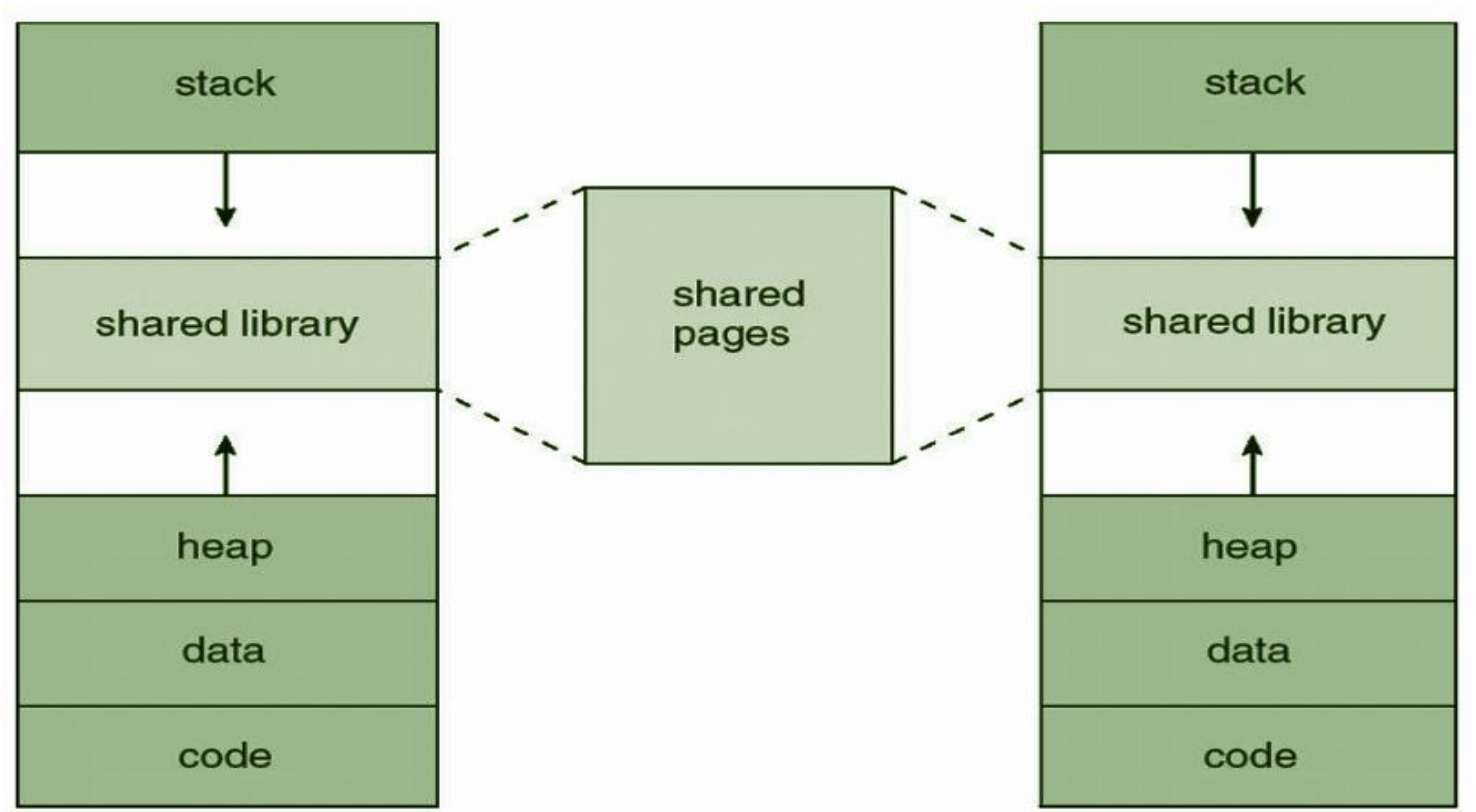


Virtual Address Space

- Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during fork(), speeding process creation

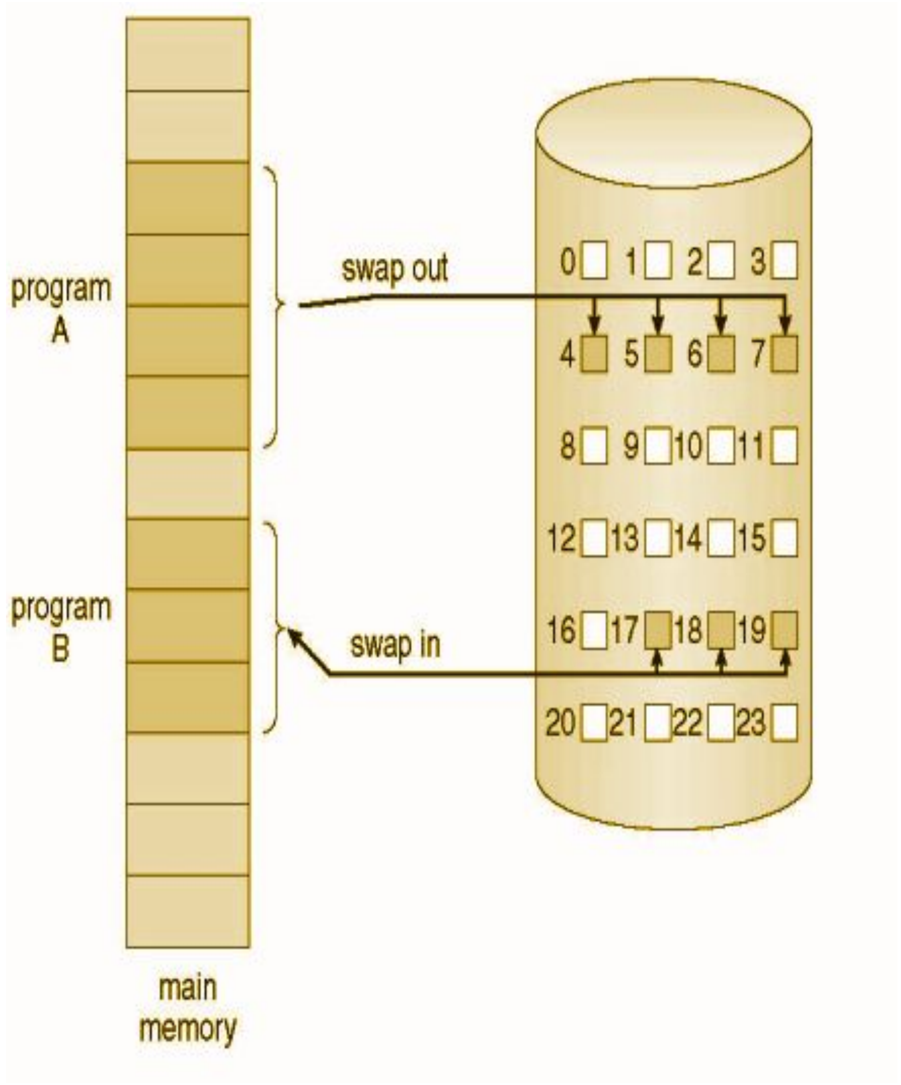


Shared Library Using Virtual Memory



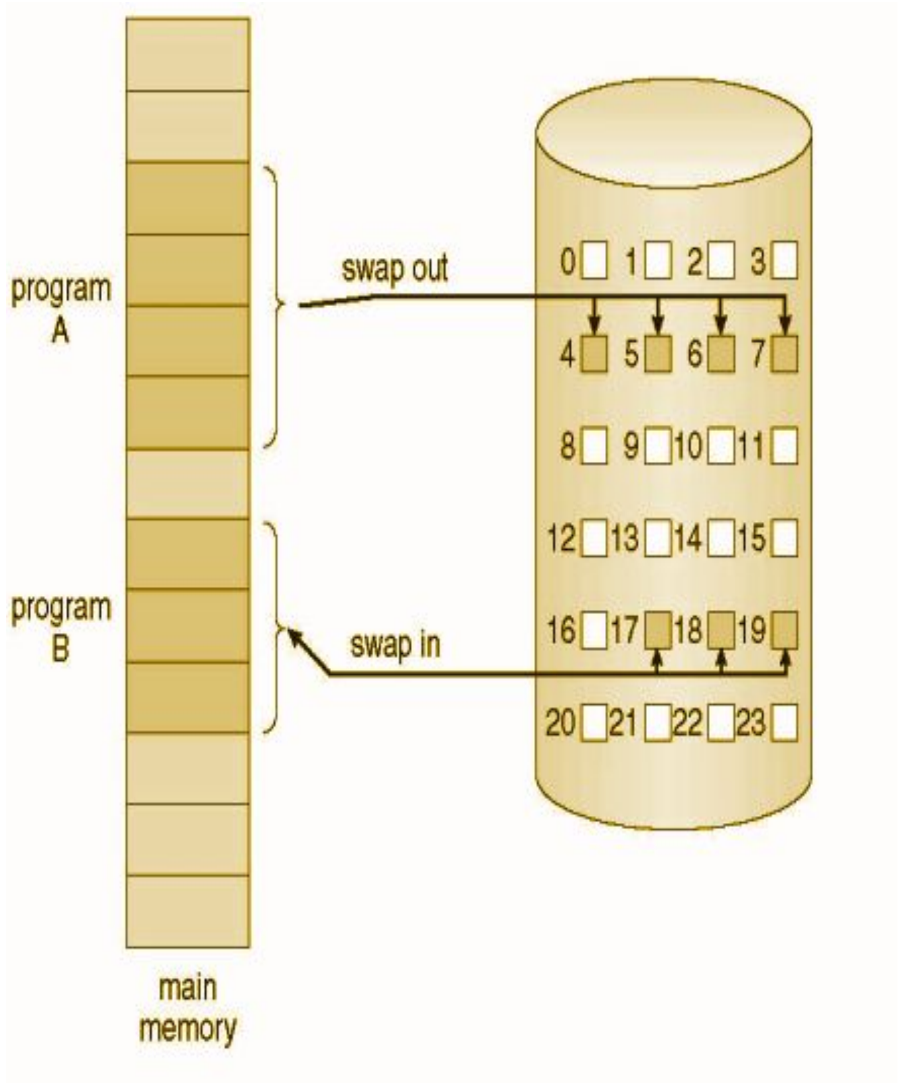
Demand Paging

- 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
- Similar to paging system with swapping



Demand Paging

- Page is needed reference to it
  - invalid reference abort
  - not-in-memory bring to memory
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **Pager**



Swapping - Basic Concepts

- With swapping, **Pager** guesses which pages will be used before swapping out again
- Instead, **Pager** brings in only those pages into memory
- How to determine that set of pages?
    - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
    - No difference from non demand-paging

Swapping - Basic Concepts

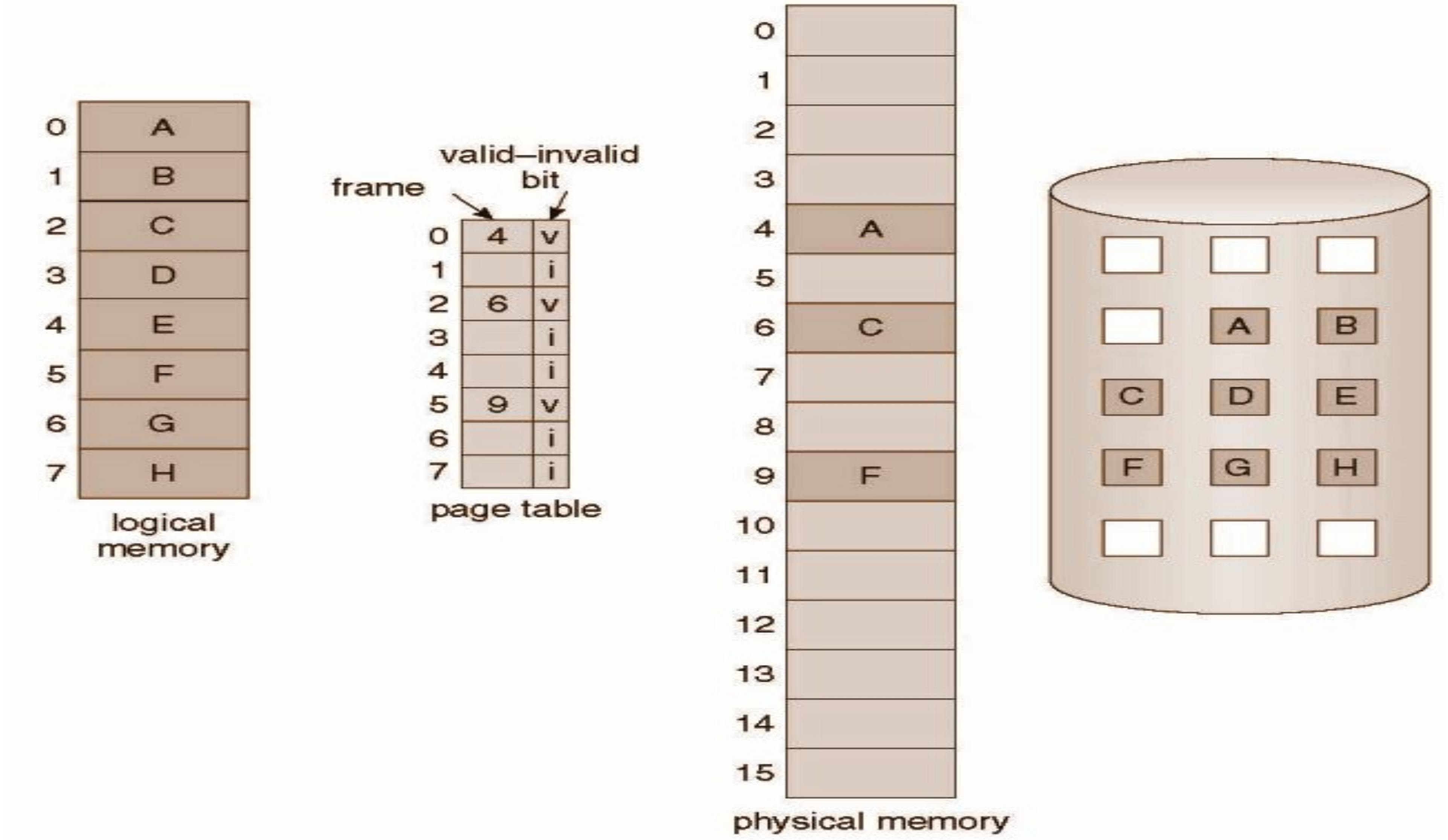
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
  - Without changing program behavior
  - Without programmer needing to change code

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - **v** in-memory – memory resident,
  - **i** not-in-memory
- Initially valid–invalid bit is set to **i** on all entries
- During MMU address translation, if valid–invalid bit in page table entry is **i** => **Page Fault**

Page #	Frame #	Valid - Invalid Bit
0		**i**
1		**i**
2	23 - belongs to a shared page	**v**
3	23 - belongs to a shared page	**v**
4	9	**v**
5	11	**v**
6		**i**
Page Table		

Page Table When Some Pages Are Not in Main Memory



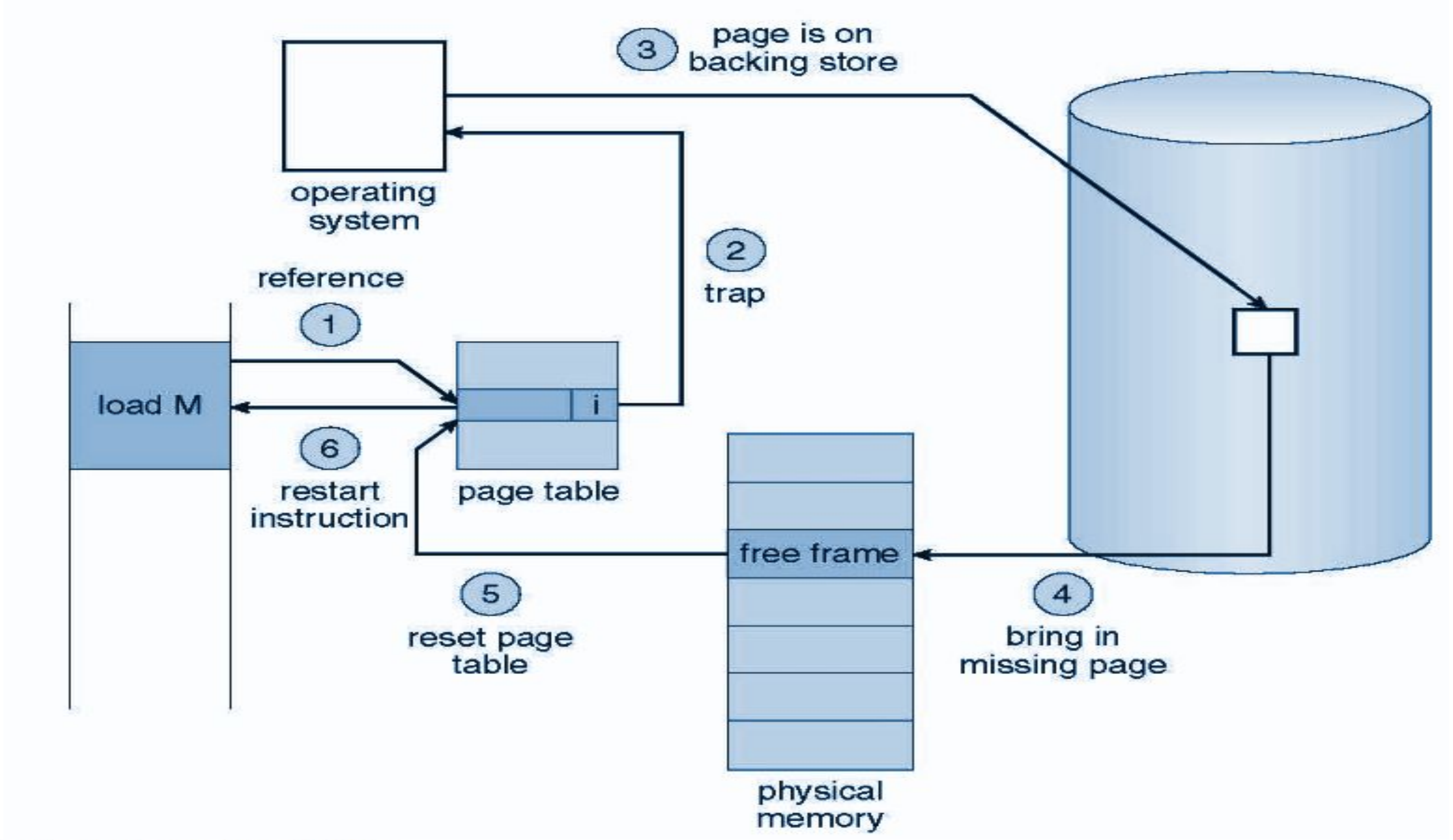
Page fault

- If there is a reference to a page, first reference to that page will trap to operating system => **Page Fault**

Page fault

1. Operating system looks at another table to decide:
   1. **Invalid reference abort**
   2. **Just not in memory**
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
5. Set validation bit => **v**
6. Restart the instruction that caused the **Page Fault**

Page fault


3
page is on backing store
operating system
2
trap
reference
1
load M
i
6
restart instruction
page table
free frame
5
reset page table
4
bring in missing page
physical memory

Aspects of Demand Paging

- Extreme case => start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident => **Page Fault**
- For every other process pages on first access **Pure Demand Paging**

Aspects of Demand Paging

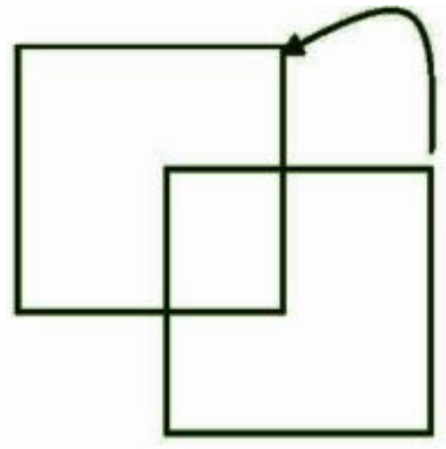
- Actually, a given instruction could access multiple pages => multiple page faults
- Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
- Pain decreased because of **Locality of Reference**

Aspects of Demand Paging

- Hardware support needed for Demand Paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart

Aspects of Demand Paging

- Consider an instruction that could access several different locations
  - block move
  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?



Performance of Demand Paging - Worst Case

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame

Performance of Demand Paging - Worst Case

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging - Worst Case

- 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

Performance of Demand Paging - Worst Case

- Page Fault Rate 0 <= p <= 1
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)
  - EAT = (1 – p) x memory access
    + p (page fault overhead=> + [swap
    page **out]** + swap page **in** +
    Restart Overhead)

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if No Page Fault i.e p=>0 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 0) * 200 + 0 * PFST

  => 200ns

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if Page Fault i.e p=>1 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 1) * 200 + 1 * PFST

  => PFST

Performance of Demand Paging - Worst Case

- 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

Performance of Demand Paging - Worst Case

- Page Fault Rate 0 <= p <= 1
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)
  - EAT = (1 – p) x memory access
    + p (page fault overhead=> + [swap
    page **out]** + swap page **in** +
    Restart Overhead)

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if No Page Fault i.e p=>0 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 0) * 200 + 0 * PFST

  => 200ns

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if Page Fault i.e p=>1 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 1) * 200 + 1 * PFST

  => PFST

Demand Paging - Performance Calculation

- 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 access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Demand Paging - Performance Calculation

- 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 access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x 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 when freeing frame
  - 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)

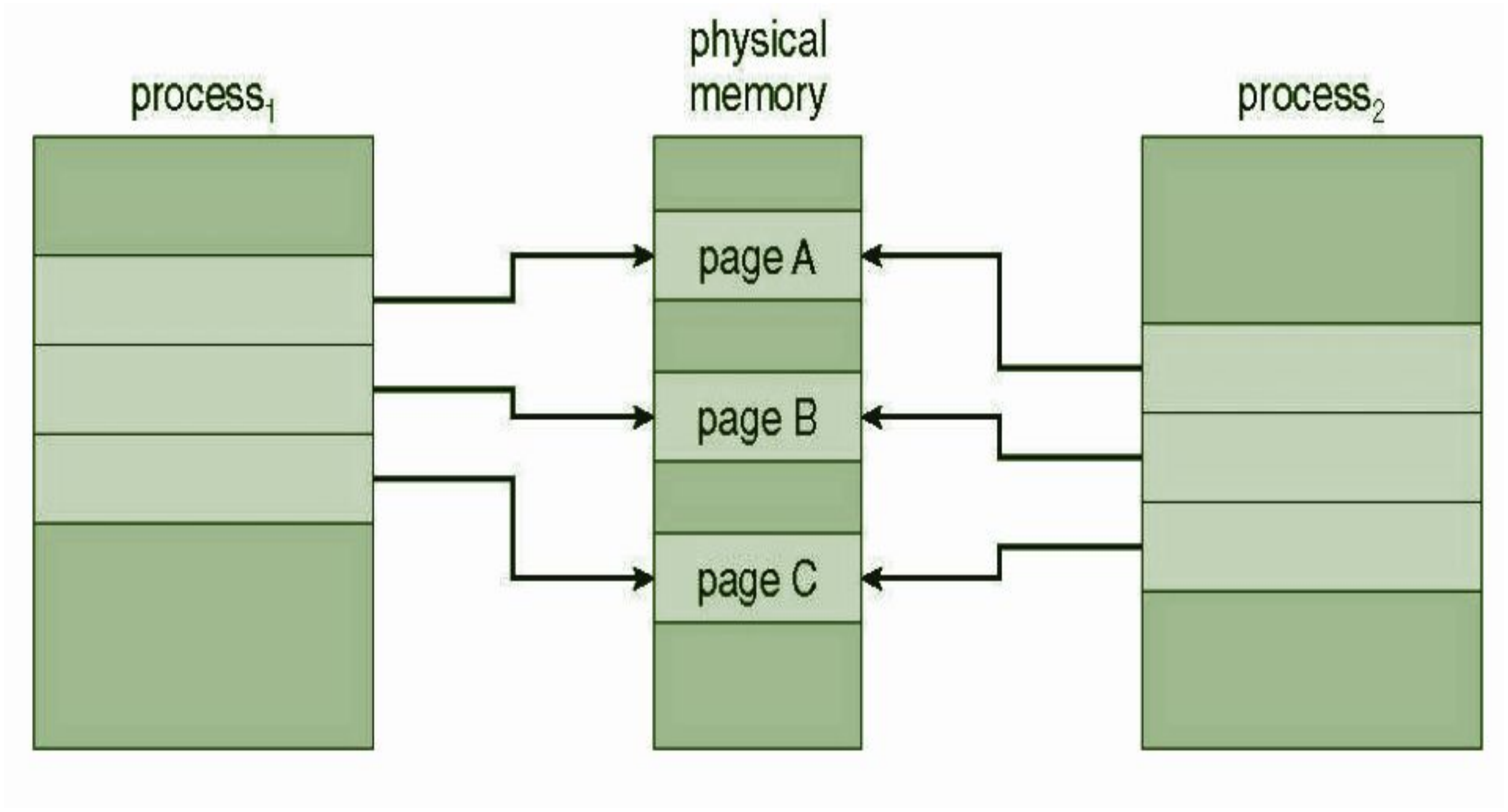
Copy-on-Write

- 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
- 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 ?

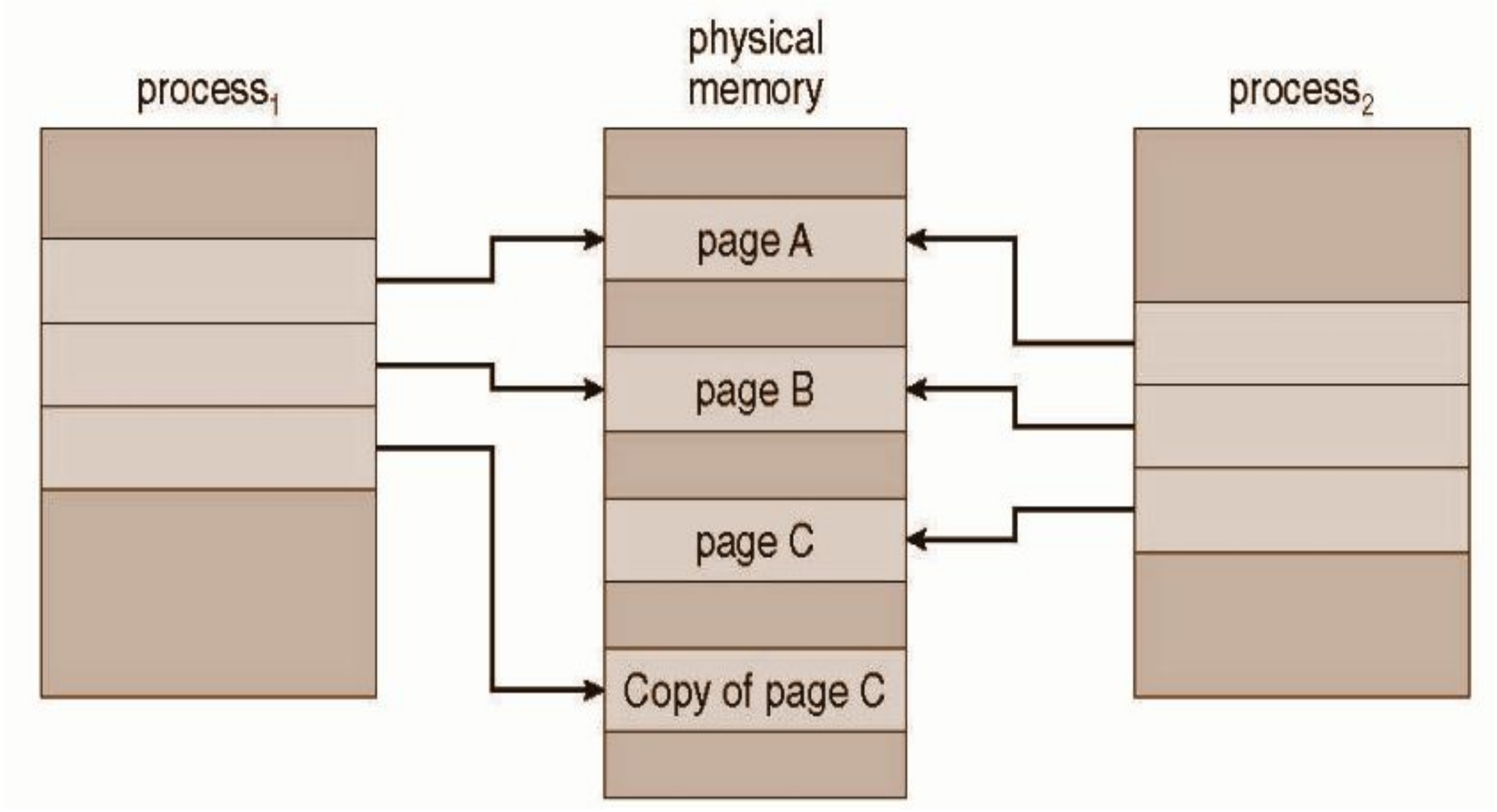
Copy-on-Write

- 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

Copy-on-Write: Before Process 1 Modifies Page C



Copy-on-Write: Before Process 1 Modifies Page C



What happens if there is no free Frame ?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How many frames to allocate to each process requesting pages to be loaded ?
- 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 to 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

1. Find the location of the desired page on disk
2. Find a free frame:
    i. If there is a free frame, use it
    ii. If there is no free frame, use a page replacement algorithm to
    iii. Select a **Victim** frame
        - Write **Victim** frame to disk if **dirty**

Page Replacement

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

4. Continue the process by restarting the instruction that caused the trap

Note: Potentially 2 page transfers for page fault – increasing EAT, if the page is swapped

Page and Frame Replacement Algorithms

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

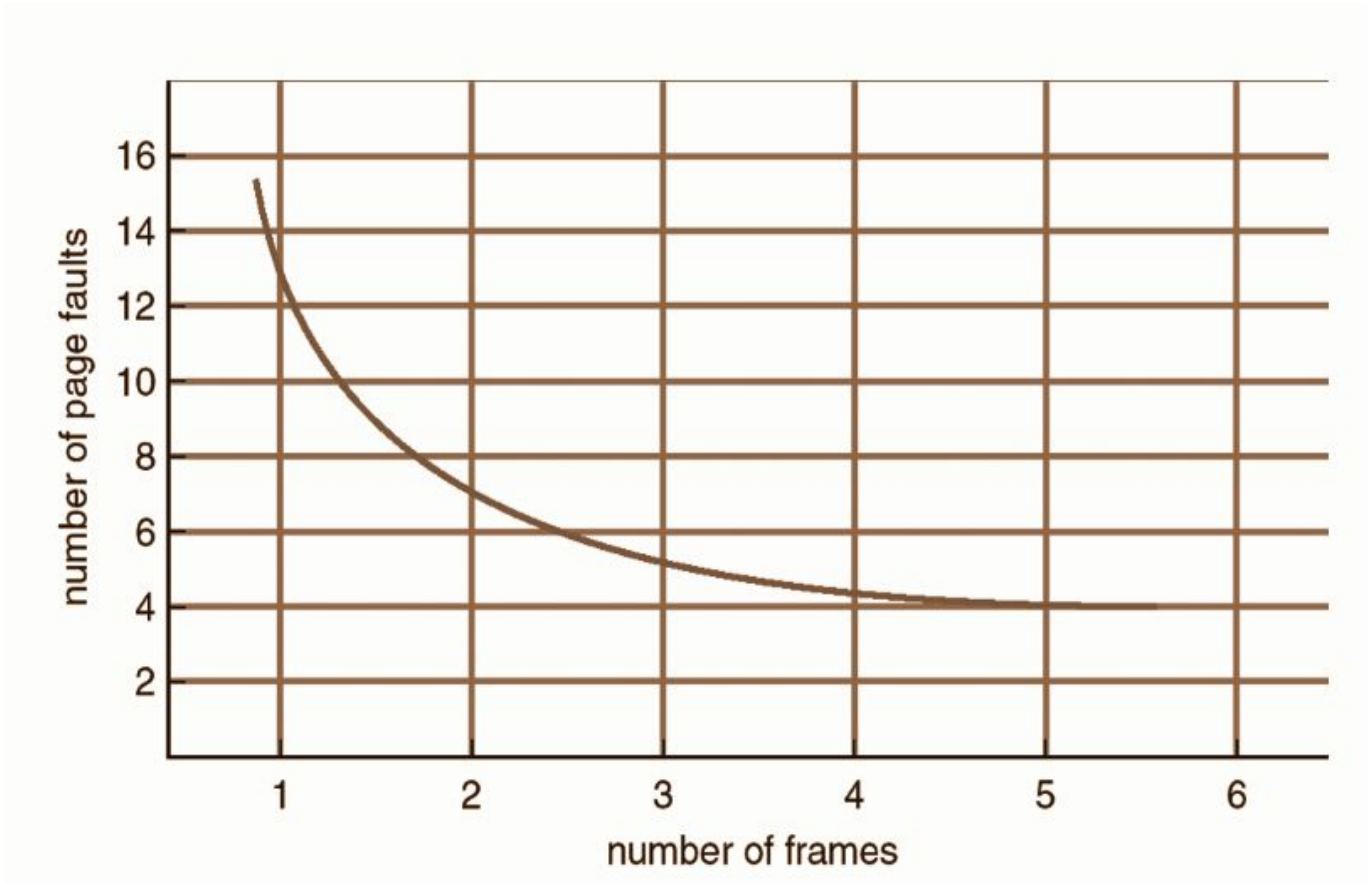
Page and Frame Replacement Algorithms

- 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 if available in the frame
  - Results depend on number of frames available
- In all our examples, the reference string of referenced page numbers of the same process is

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
Page #	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1

Reference String

Graph of Page Faults versus the Number of Frames



PRA : First in First Out (FIFO) Algorithm

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
Page #	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	0	0	7	7	7
2		0	0	0	0	3	3	3	2	2	2	2	2	2	2	1	1	1	1	1	0	0
3			1	1	1	1	0	0	0	3	3	3	3	3	3	3	2	2	2	2	2	1
Flag	PF	PF	PF	PF	PH	PF	PF	PF	PF	PF	PF	PH	PH	PH	PH	PF	PF	PH	PH	PF	PF	PF

Working Set => { 7, 0, 1, 2 ,3, 4 }

Total Number of Page Requests = > 22

Total Number of Frames => 3

Total Number of Page faults => 15

% of Page Faults = > 15/22 => 68.18%

% of Page Hits => 31.81 %

Legend
Page Fault => PF

Page Hit => PH

Req #-	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
Page #	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1

OPERATING SYSTEMS

PRA: FIFO Illustrating Belady's Anomaly

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Page #	1	2	3	4	1	2	5	1	2	3	4	5

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1												
2												
3												
Flag												

- 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

Working Set => { }

Total Number of Page Requests = >

Total Number of Frames =>

Total Number of Page faults =>

% of Page Faults = >

% of Page Hits =>

Legend
Page Fault => PF

Page Hit => PH

*Anomaly => Something that deviates from what is standard, normal, or expected

PRA: FIFO Illustrating Belady's Anomaly

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Page #	1	2	3	4	1	2	5	1	2	3	4	5

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1	1	1	1	1			5	5	5	5	4	4
2		2	2	2			2	1	1	1	1	5
3			3	3			3	3	2	2	2	2
4				4			4	4	4	3	3	3
Flag	PF	PF	PF	PF			PF	PF	PF	PF	pf	pf

- 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

Working Set => { }

Total Number of Page Requests = >

Total Number of Frames =>

Total Number of Page faults =>

% of Page Faults = >

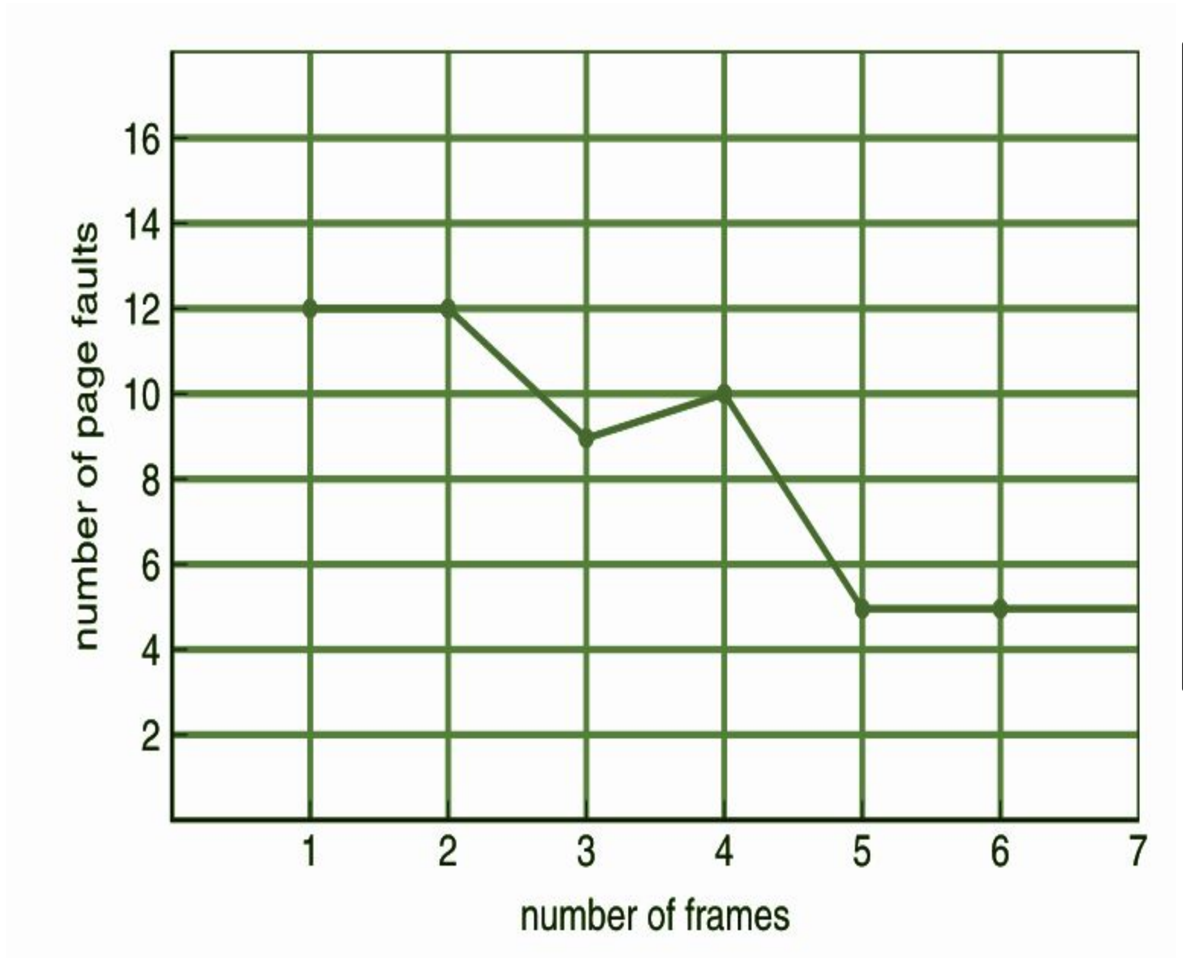
% of Page Hits =>

Legend
Page Fault => PF

Page Hit => PH

*Anomaly => Something that deviates from what is standard, normal, or expected

PRA: FIFO Illustrating Belady's Anomaly



- 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

Optimal Page Replacement Algorithm

- Replace page that will not be used for longest period of time
- How do you know this ?
- Can’t read the future
- Used for measuring how well your algorithm performs

PRA : Optimal Page Replacement Algorithm

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
Page #	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0
3			1	1	1	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Flag	PF	PF	PF	PF	PH	PF	PH	PF	PH	PH	PF	PH	PH	PH	PH	PF	PH	PH	PH	PF	PH	PH

Working Set => { 7,0,1,2,3,4 }

Total Number of Page Requests = > 22

Total Number of Frames => 3

Total Number of Page faults => 9

% of Page Faults = > 9/22 => 40.90%

% of Page Hits => 59.09 %

Legend
Page Fault => PF

Page Hit => PH

Least Recently Used Page Replacement 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
- Generally good algorithm and frequently used
- But how to implement ?

PRA : Least Recently Used Page Replacement Algorithm - LRU

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
Page #	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22
1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	1	1	1	1	1	1	1
2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	0	0	0	0	0
3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	2	2	7	7	7
Flag	PF	PF	PF	PF	PH	PF	PH	PF	PF	PF	PF	PH	PH	PH	PH	PF	PH	PF	PH	PF	PH	PH

Working Set => { 7,0,1,2,3,4 }

Total Number of Page Requests = > 22

Total Number of Frames => 3

Total Number of Page faults => 12

% of Page Faults = > 12/22 => 54.54%

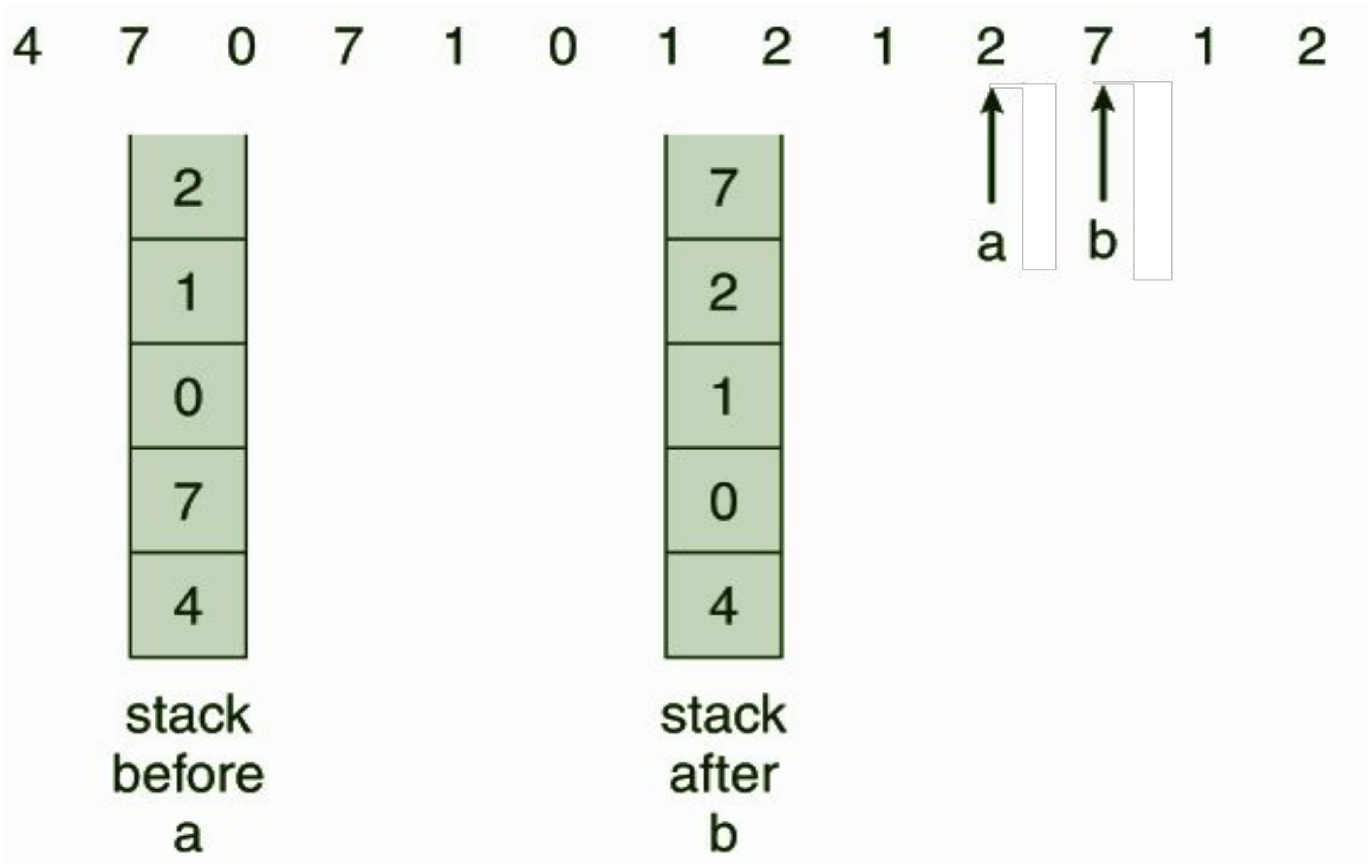
% of Page Hits => 45.45%

Legend

Page Fault => PF

Page Hit => PH

Least Recently Used Page Replacement Algorithm



Page Replacement Algorithms - Comparison

First in First Out Page Replacement Algorithm

Working Set => { 7, 0, 1, 2 ,3, 4 }

Total Number of Page Requests = > 22

Total Number of Frames => 3

Total Number of Page faults => 15

% of Page Faults = > 15/22 => 68.18%

% of Page Hits => 31.81 %

Legend
Page Fault => PF

Page Hit => PH

Optimal Page Replacement Algorithm

Working Set => { 7,0,1,2,3,4 }

Total Number of Page Requests = > 22

Total Number of Frames => 3

Total Number of Page faults => 9

% of Page Faults = > 9/22 => 40.90%

% of Page Hits => 59.09 %

Legend
Page Fault => PF

Page Hit => PH

Least Recently Used Page Replacement Algorithm

Working Set => { 7,0,1,2,3,4 }

Total Number of Page Requests = > 22

Total Number of Frames => 3

Total Number of Page faults => 12

% of Page Faults = > 12/22 => 54.54%

% of Page Hits => 45.45%

Legend
Page Fault => PF

Page Hit => PH

Optimal Page Replacement Algorithm - Practice Problem

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Page #	1	2	3	4	1	2	5	1	2	3	4	5

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1												
2												
3												
Flag												

Working Set => { }

Total Number of Page Requests = >

Total Number of Frames =>

Total Number of Page faults =>

% of Page Faults = >

% of Page Hits =>

Legend

Page Fault => PF

Page Hit => PH

Optimal Page Replacement Algorithm - Practice Problem

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Page #	1	2	3	4	1	2	5	1	2	3	4	5

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1												
2												
3												
4												
Flag												

Working Set => { }

Total Number of Page Requests = >

Total Number of Frames =>

Total Number of Page faults =>

% of Page Faults = >

% of Page Hits =>

Legend
Page Fault => PF

Page Hit => PH

OPERATING SYSTEMS

Least Recently Used Page Replacement Algorithm - Practice Problem

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Page #	1	2	3	4	1	2	5	1	2	3	4	5

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1												
2												
3												
Flag												

Working Set => { }

Total Number of Page Requests = >

Total Number of Frames =>

Total Number of Page faults =>

% of Page Faults = >

% of Page Hits =>

Legend
Page Fault => PF

Page Hit => PH

OPERATING SYSTEMS

Least Recently Used Page Replacement Algorithm - Practice Problem

Req #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Page #	1	2	3	4	1	2	5	1	2	3	4	5

Fr #	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
1												
2												
3												
4												
Flag												

Working Set => { }

Total Number of Page Requests = >

Total Number of Frames =>

Total Number of Page faults =>

% of Page Faults = >

% of Page Hits =>

Legend
Page Fault => PF

Page Hit => PH

Virtual Memory - Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

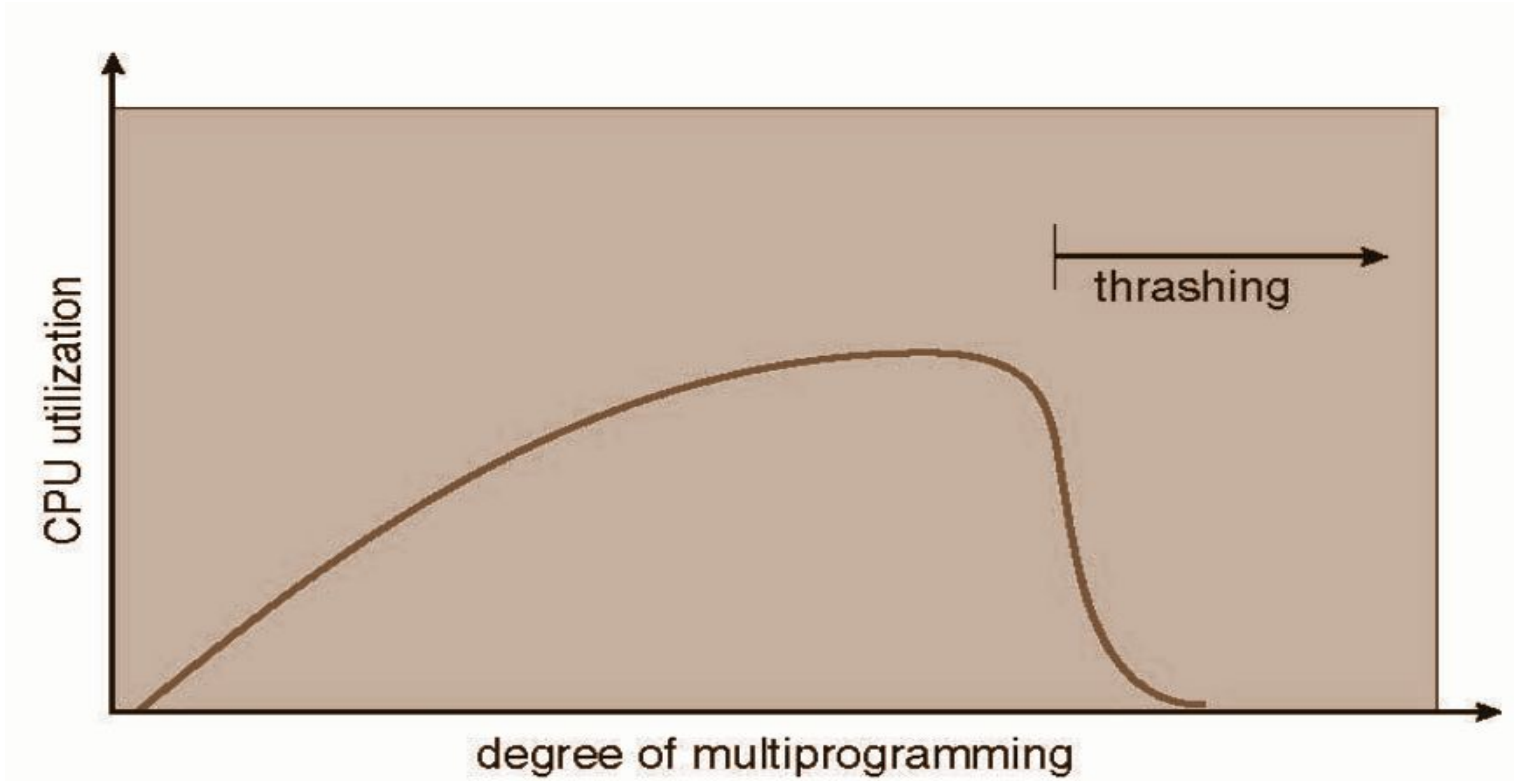
Virtual Memory - Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
    - This leads to:
        - Low CPU utilization
        - Operating system thinking that it needs to increase the degree of multiprogramming
        - Another process added to the system

Virtual Memory - Thrashing

Thrashing is a process of keeping system busy with swapping pages **in** and **out**

Virtual Memory - Thrashing



Demand Paging and Thrashing

- Why does demand paging work ?
  - Locality model
    - Process migrates from one locality to another
    - Localities may overlap

Demand Paging and Thrashing

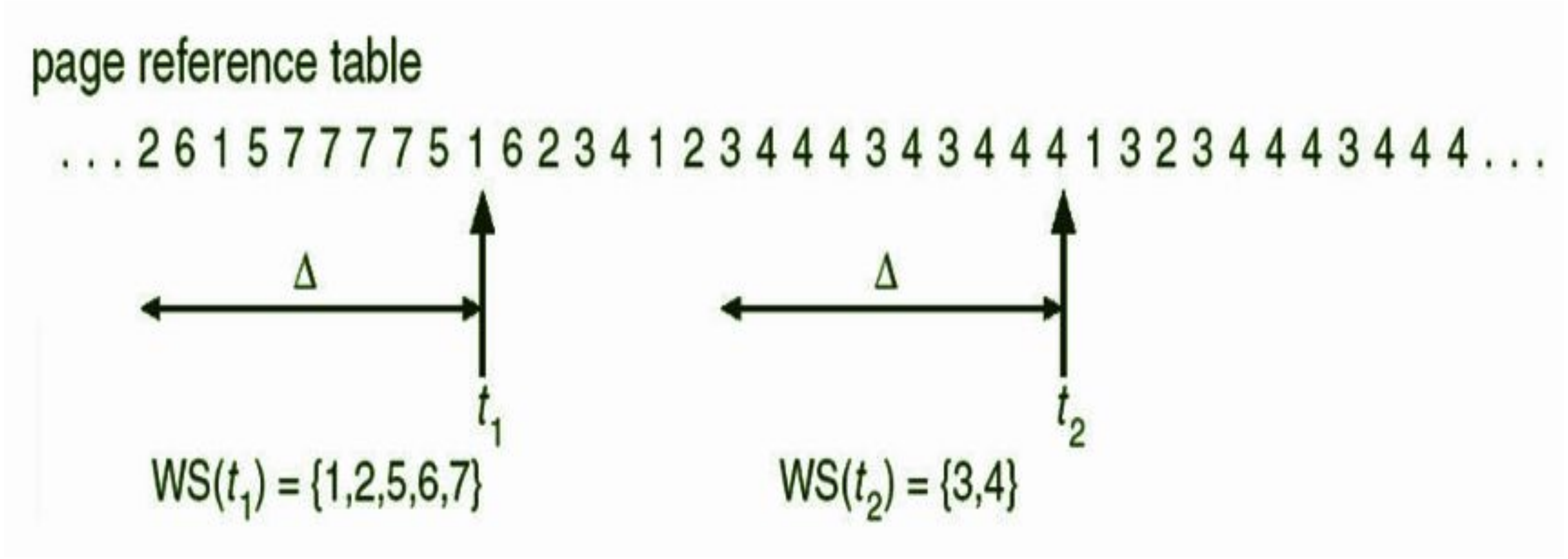
- Why does thrashing occur ?
  - **Size of Locality** > Total Memory Size
  - Limit effects by using local or priority page replacement

Working Set Model

- $\Delta \equiv$ working-set window => a fixed number of page references
- Example: 10,000 instructions
- WSSi (working set of Process Pi) = total number of pages referenced in the most recent (varies in time)
  - if Δ is too **small** => will **not** encompass entire locality
  - if Δ is too **large** => **will** encompass several localities
  - if $\Delta = \infty$ will encompass **entire program**
- D = $\sum$ WSSi **total** demand frames
  - Approximation of locality

Working Set Model

- If D > m => Thrashing
- Policy if D > m, then suspend or swap out one of the processes



Working Set Model - Additional Input

- Working Sets: Conceptual model proposed by Peter Denning to prevent thrashing.
- **Informal definition:** the collection of pages a process is using actively, and which must thus be memory-resident to prevent this process from thrashing.
- If the sum of all working sets of all runnable threads or active process exceeds the size of memory, then stop running some of the threads or process for a while.

Working Set Model - Additional Input

- Divide processes into two groups: **active** and **inactive**:
    - When a process is **active** its entire working set must always be in memory: never execute a thread or a process whose working set is not resident.
    - When a process becomes **inactive**, its working set can migrate to disk.
    - Threads or child processes from inactive processes are never scheduled for execution.
    - The collection of **active** processes is called the **balance set**.
    - The system must have a mechanism for gradually moving processes into and out of **the balance set**.
    - As working sets change, **the balance set** must be adjusted.

Working Set Model - Additional Input

- How to compute working sets ?
  - Denning proposed a working set parameter T: all pages referenced in the last T seconds comprise the working set.
  - Can extend the clock algorithm to keep an idle time for each page.
  - Pages with idle times less than T are in the working set.
- Difficult questions for the working set approach:
  - How long should T be (typically minutes) ?
  - How to handle changes in working sets ?
  - How to manage the balance set ?
  - How to account for memory shared between processes ?

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 sets the values of all reference bits to 0
  - If one of the bits in memory = 1 page in working set
- Why is this not completely accurate ?
- Improvement = 10 bits and interrupt every 1000 time units

Current Situation - Additional Input

In practice, today's operating systems don't worry much about thrashing:

- With personal computers, users can notice thrashing and handle it themselves:
    - Typically, just buy more memory
    - manage balance set by hand
- Thrashing was a bigger issue for timesharing machines with dozens or hundreds of users:
    - **For a typical question like why should I stop my processes just so you can make progress ?**
    - **Answered by making the System to handle thrashing automatically.**
- Technology changes make it unreasonable to operate machines in a range where memory is even slightly overcommitted; better to just buy more memory.

Page-Fault Frequency

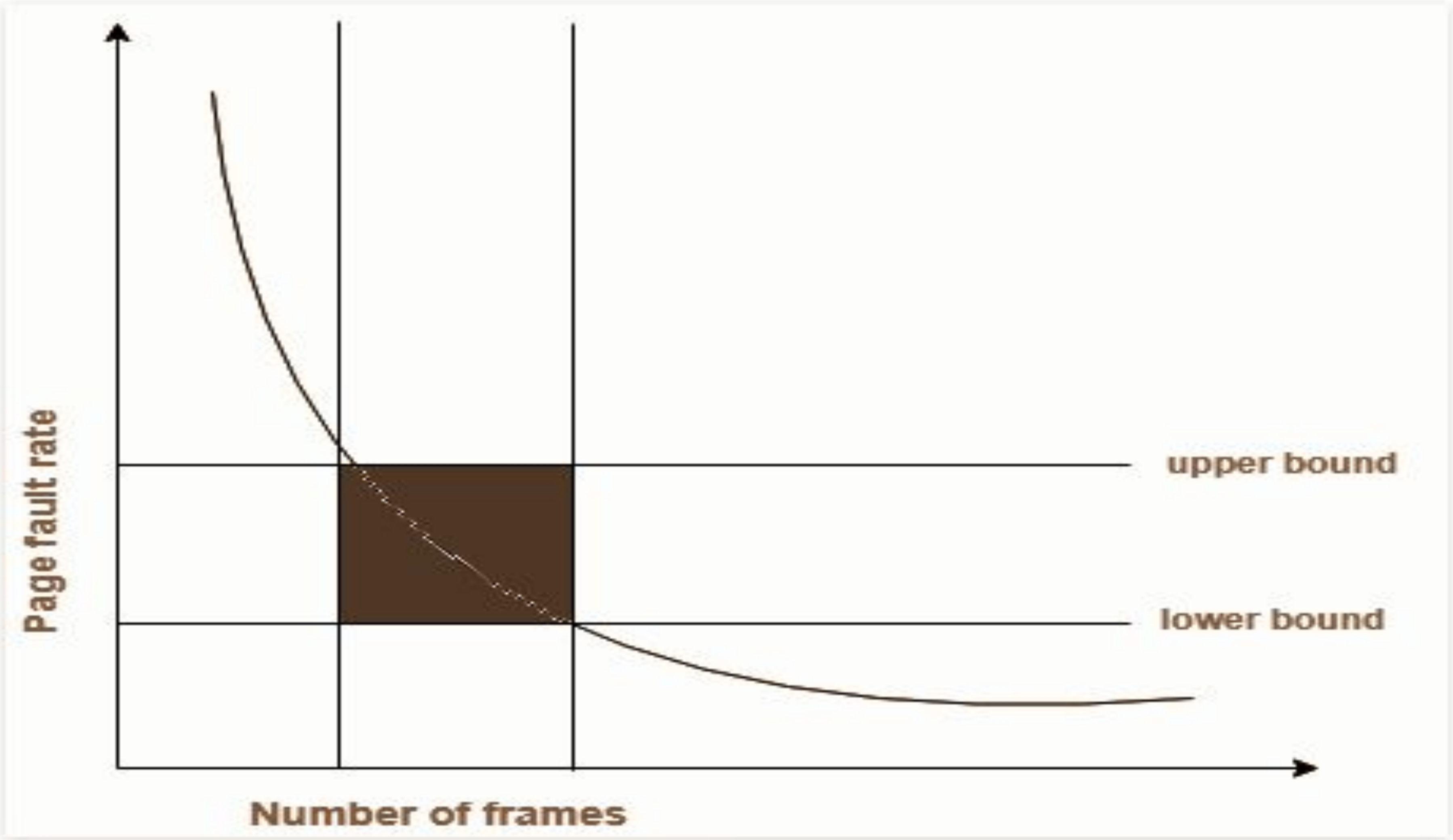
- More direct approach than Working set of process S
- Establish “acceptable” page-fault frequency (PFF) rate for a process and use local replacement policy
  - If actual rate too **low**, process **loses** frame
    - A low page fault rate indicates that the process has **too many frames**.
  - If actual rate too **high**, process **gains** frame
    - If the page fault rate is too high, it indicates that the process **has too few frames** allocated to it.

Page-Fault Frequency - Additional Input

Page Fault Frequency: another approach to preventing thrashing.

- Per-process replacement; at any given time, each process is allocated a fixed number of physical page frames.
- Monitor the rate at which page faults are occurring for each process.
- If the rate gets too high for a process, assume that its memory is overcommitted; increase the size of its memory pool.
- If the rate gets too low for a process, assume that its memory pool can be reduced in size.
- If the sum of all memory pools don't fit in memory, deactivate some processes.

Page-Fault Frequency - Additional Input



THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com