CMP325 Operating Systems Lecture 24, 25

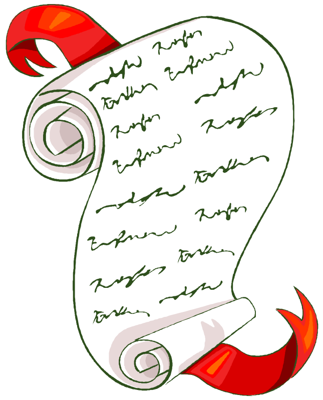
Virtual Memory

Muhammad Arif Butt, PhD

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of
- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:
OS with Linux:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8
System Programming:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

Today's Agenda



- Review of Previous Lecture
- Introduction to Virtual Memory
  - Demand Paging
  - Servicing the page faults
  - Performance of Demand Paging
  - Memory Mapped Files
  - Page Replacement Algorithms
    - FIFO
    - Optimal
    - LRU
    - Page Buffering
- Thrashing
- Working Set Model

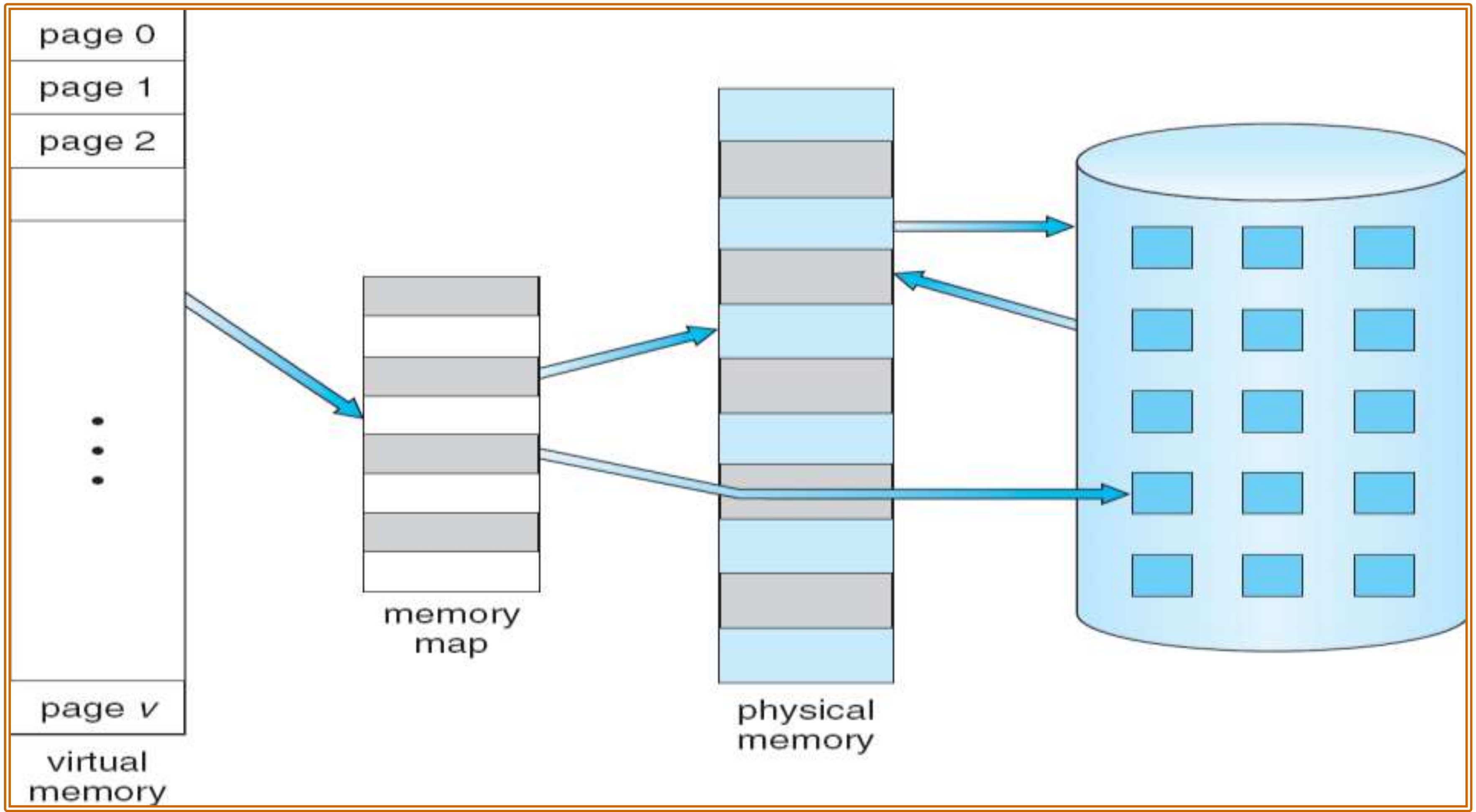
INTRODUCTION TO VIRTUAL MEMORY

- We have seen that a user process had to be completely loaded into memory before it could run
- This is wasteful since a process only needs a small amount of its total address space at any one time (locality)
- Knuth's estimate that 90% of the time, a process executes in 10% of its code
- Virtual memory is a technique used by operating systems that allows a process to execute programs having size larger than the available main memory
- Only a part of a process needs to be loaded in the main memory for it to execute

INTRODUCTION TO VIRTUAL MEMORY

- If an entire process is not loaded while it executes, it may be possible that the process requests for an instruction/data in a page that is not in the main memory rather is there in the backing store. Hardware and software cooperate to make things work
- Extend the page tables with extra bit "present bit" or "valid bit"; if it is one that means the page in loaded and vice versa
- In such situations, the operating system need to take two kinds of scheduling decisions:
  - **Page Selection.** When to bring pages into memory
    - **Demand Paging**. Start up process with no pages loaded, load a page when a page fault for it occurs
    - **Request Paging**. Let user say which pages are needed. Limitation is user don't always knows best
    - **Pre-paging.** Bring a page into memory before it is referenced. When one page is referenced , bring in the next one as well
  - **Page Replacement**. Which page(s) should be thrown out, and when

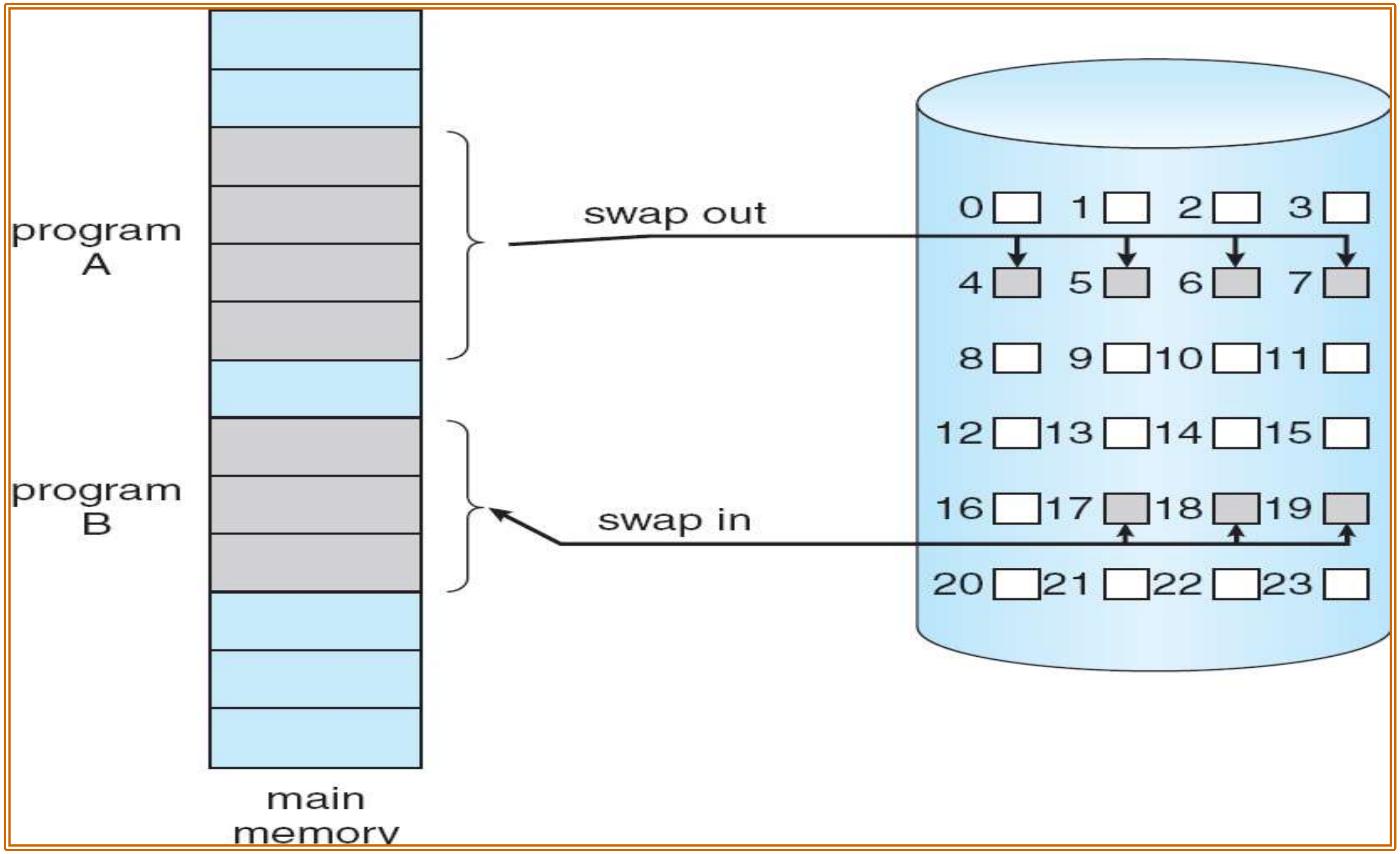
INTRODUCTION TO VIRTUAL MEMORY



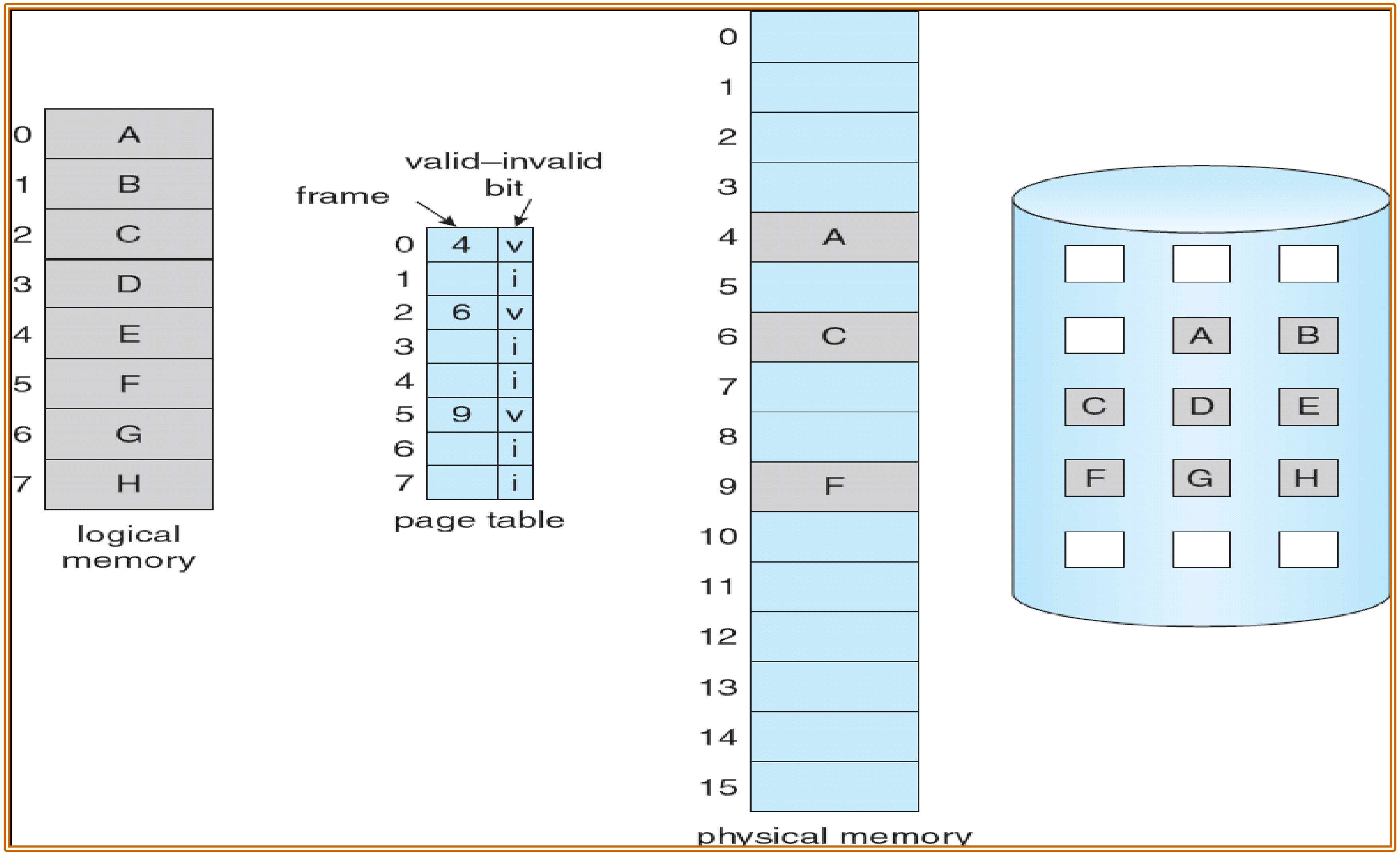
DEMAND PAGING

- Demand paging says; **never bring a page into memory until it is needed**
- Advantages of demand paging are:
    - –Potentially Less I/O needed
    - –Potentially Less memory needed
    - –Faster response
    - –High degree of multiprogramming

TRANSFER OF A PAGE FROM MEMORY TO DISK



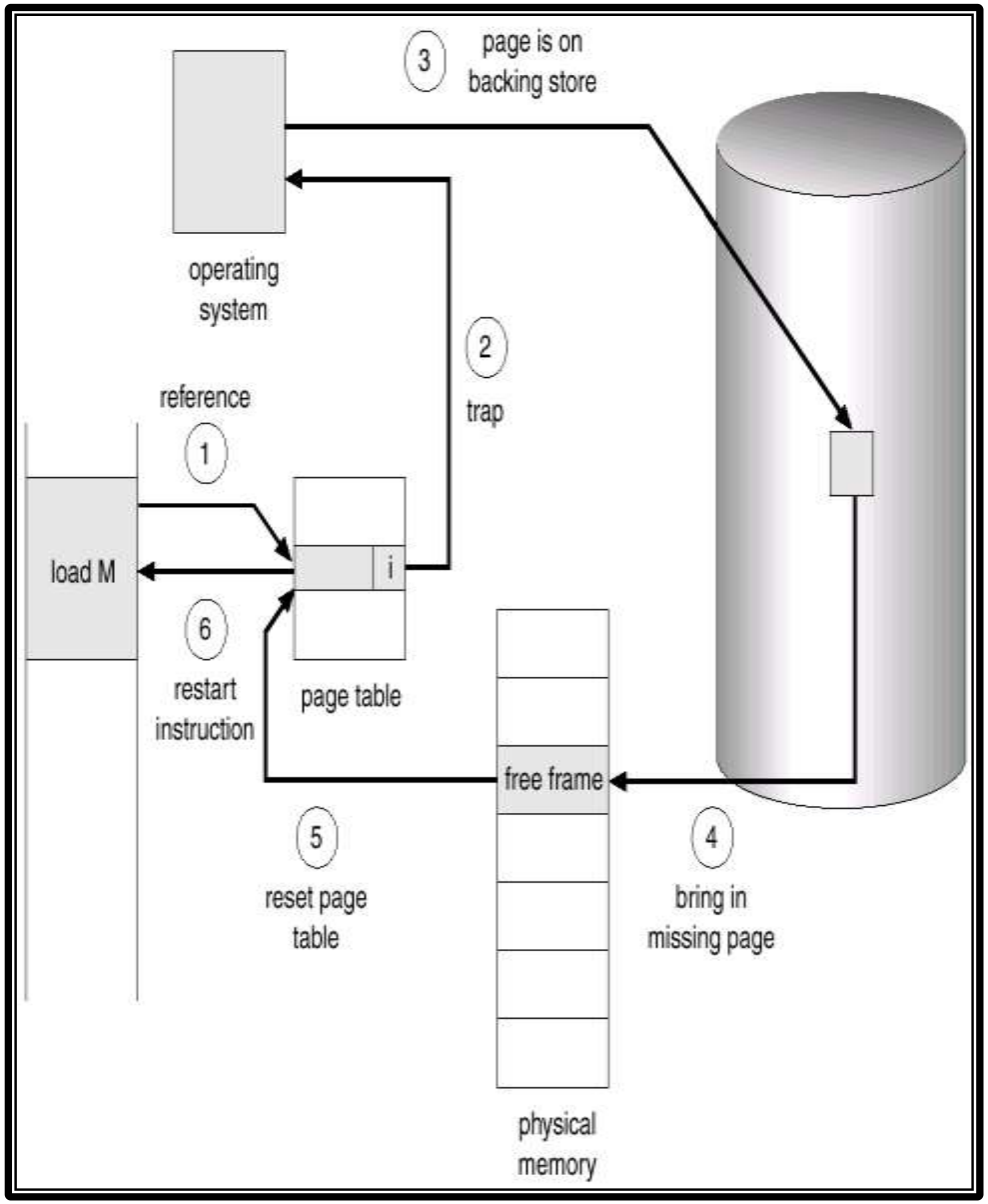
PAGE TABLE WHEN SOME PAGES ARE NOT IN MAIN MEMORY



SERVICING A PAGE FAULT

When ever a page reference is made, the operating system checks:

1. If it is an invalid reference a trap to the OS is made and the process is aborted
2. If it is a valid reference then the resident bit of the page is checked. If the resident bit is zero that means the page is not loaded in the memory, so a page fault is generated. To service this page fault following steps are taken:
   a. Locate the page on the disk
   b. Allocate an empty/free frame from the frames allocated to that process and load the page into that frame
   c. If there is no free frame available with that process then select a victim frame and swap out that victim frame/page back to the hard disk
   d. Swap in the desired page into the selected frame
   e. Store the frame number in the appropriate page table entry, and set resident bit to one
   f. Restart instruction



PERFORMANCE OF DEMAND PAGING

- Page Fault Rate $0 \le p \le 1.0$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- **Effective Access Time (EAT)**

  EAT = (No page fault) * mat + (page fault) * (page fault service time)

  EAT = (1 – p) x mat + p * (page fault overhead + swap page out + swap page in+ restart overhead)

Problem 54

Consider a system with Memory Access Time of 100 nanosecond. Total Page fault Service Time is 25 millisecond. Calculate Effective Access Time. Discuss how much the system will be slowed down if one out of 1000 accesses causes a page fault.

$$T_{effective} = (1 - p) * 100 + p\,(25 \times 10^6)$$

Let p = 1/1000

$$T_{effective} = (1 - 1/1000) * 100 + 1/1000 * (25 \times 10^6)$$

$$T_{effective} = 99.9 + 25000$$

$$T_{effective} = 25099.9 \text{ nsec} = 25000 \text{ nsec}$$

So the system is slowed down by a factor of 250 times, by just one page fault out of 1000 page accesses.

PERFORMANCE OF DEMAND PAGING

- If we want less than 10 percentage degradation in effective memory access time then we have the following inequality

$T_{effective} = (1 - p) * 100 + p (25 * 10^6)$

$110 > (1 - p) * 100 + 25000000 * p$

$10 > 100 - 100 * p + 25000000 * p$

$10 > 100 * p + 250000 * p$

$10 > 24999900 * p$

$10/24999900 > p$

$p < 1/2499990$

- This means we can allow only one page fault every 2,500,000. (2.5 million)

PERFORMANCE OF DEMAND PAGING

Problem 55

Consider a system with MAT of 200 nanosecond. Average Page fault service time is 8 ms. Compute Effective Access Time and discuss how much the system will be slowed down if one out of 1000 accesses causes a page fault. Also compute the page fault rate (p), if we want a slow down by less than 10%

PAGE FAULT AND NO FREE FRAME

- Want lowest 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
- We are going to discuss various page replacement algorithms as coming up on next slides
  - **Local replacement** – If a process P1 having no free frame demands a page and the victim frame selected is of the same process P1 then it is called Local replacement
  - **Global replacement** – If a process P1 having no free frame demands a page and the victim frame selected is of some other process then it is called Global replacement

PROCESS CREATION and VIRTUAL MEMORY

`fork()`

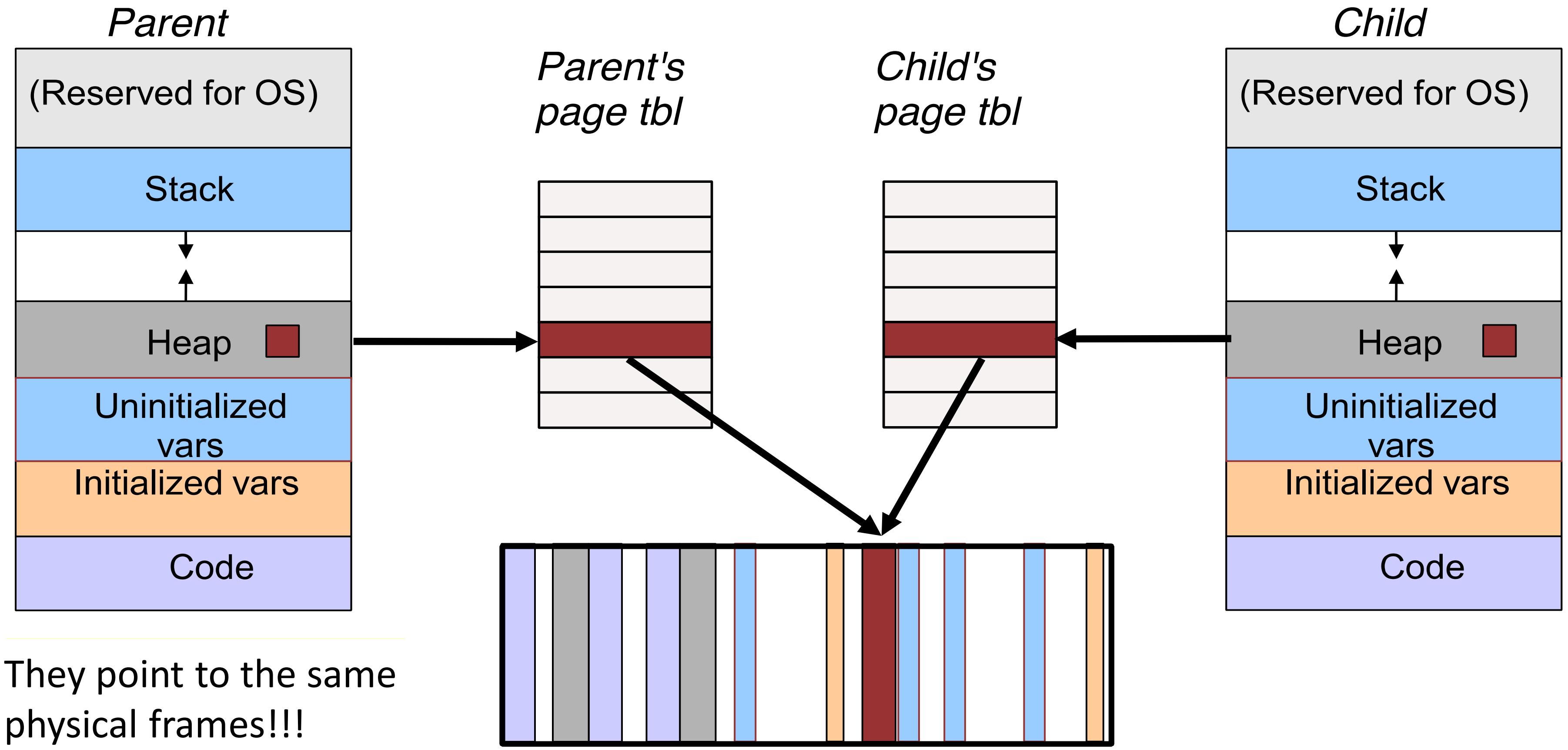
- Exact copy of parent's memory image is created
- Copying parent's pages to child's pages is a costly affair. It may not always be required to copy parent's pages to child's pages as the child is going to call `exec` system call immediately in most of the cases
- To avoid this over head some OS use **`vfork()`** and some use copy-on-write semantics.

`vfork()`

- – Parent is suspended and child uses parent's address space
- – Pages altered by the child process are visible to parent
- – Therefore, `vfork()` must be used with caution, ensuring that the child process does not modify the address space of parent
- – Intended to be used when child process calls `exec()` system call immediately after its creation

COPY ON WRITE

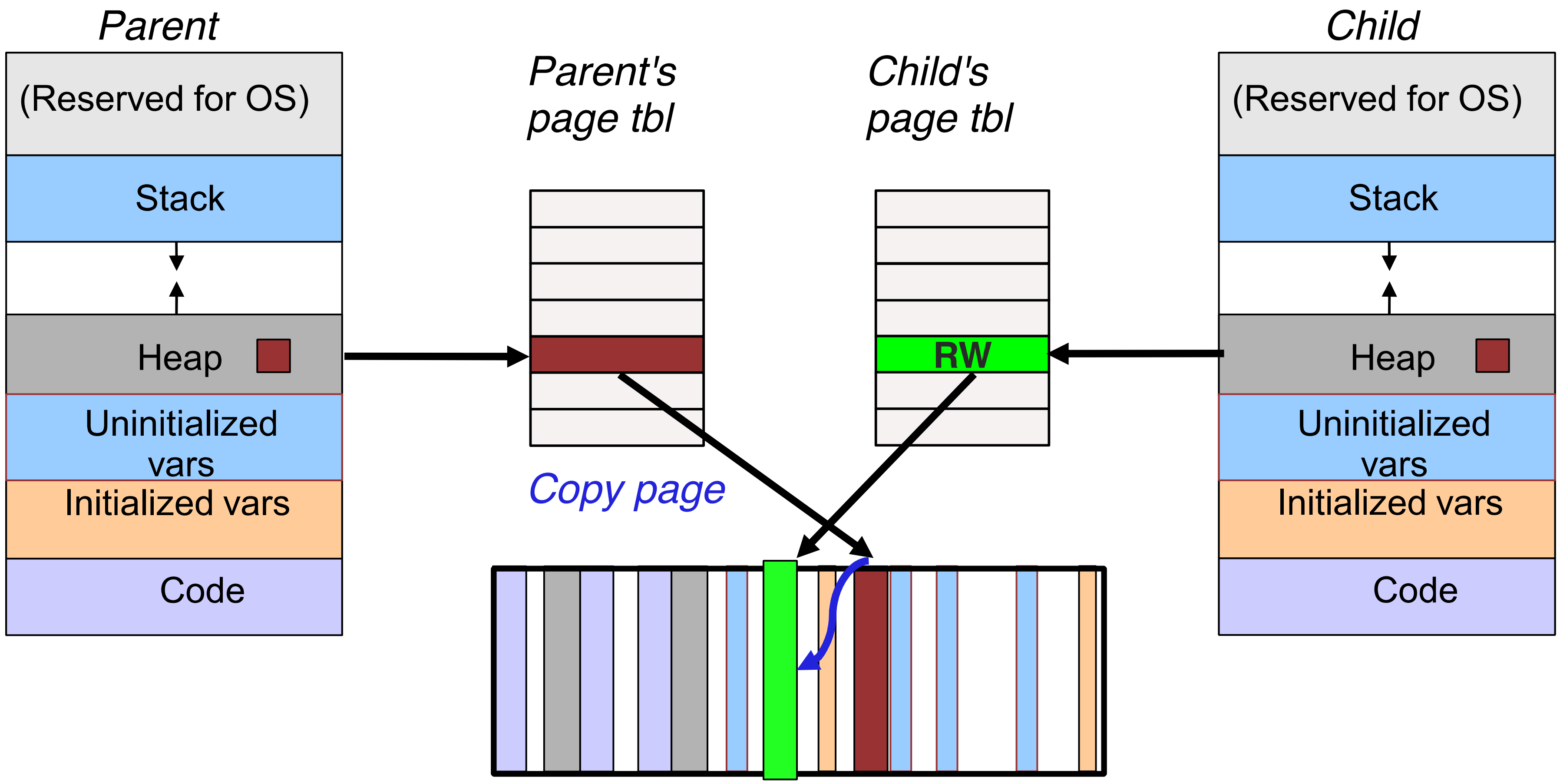
- Parent forks a child process
- Child gets a copy of the parent's page table
- Pages which may change are marked "copy-on-write" ; i.e. the pages are not copied for the child rather the child starts sharing the pages and the writable pages are marked "copy-on-write"
- **What happens when the child reads the page?**
  - Just accesses same memory as parent…. (Niiiiice)



They point to the same physical frames!!!

COPY ON WRITE

- What happens when the child/parent writes the page?
  - If either process (eg. child) tries to modify a shared page, a page fault occurs and the page is copied and inserted in the page table for that particular process
  - The other process (who later faults on write) discovers it is the only owner; so no copying takes place



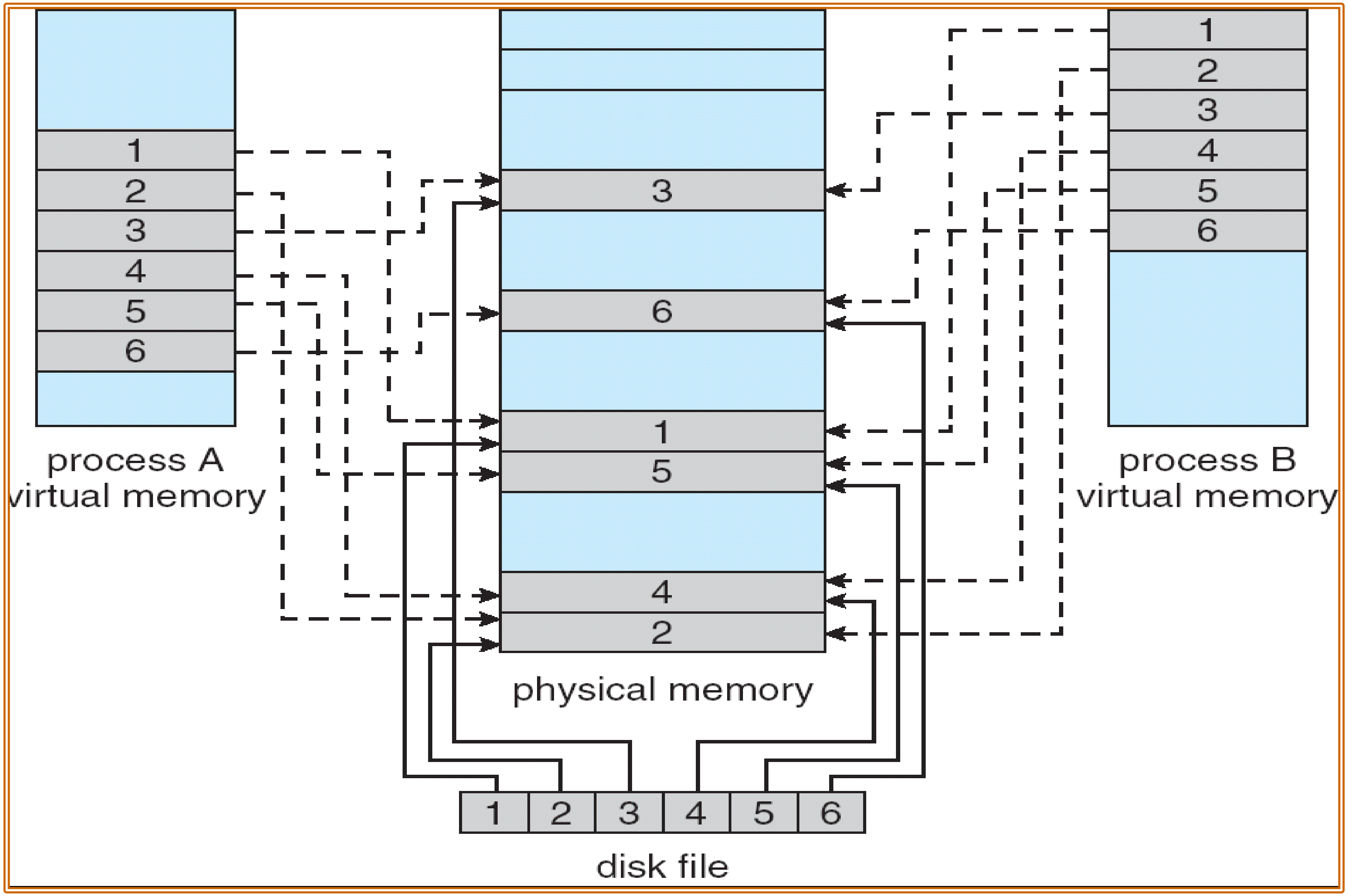
MEMORY MAPPED FILES

- Files are normally located on secondary storage (e.g. Disk). Every time a file is accessed, it requires a system call and disk access.
- Virtual memory techniques can be used to treat file I/O as routine memory access. This approach is known as memory mapping a file, allowing a part of the virtual address space to be logically associated with a file.
- Memory mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file

MEMORY MAPPED FILES

- A file is initially read using demand paging. A page sized portion of the file is read from the file system into a physical page/frame. Subsequent reads/writes from/to the file are treated as ordinary memory accesses. It simplifies file access by treating file I/O through memory rather than **read**( ) and **write**( ) system calls.
- Closing the file results in all the memory mapped data being written back to disk and removed form the virtual memory of the process
- Moreover, memory mapped files can be very useful, especially on systems that don't support shared memory segments

MEMORY MAPPED FILES



PAGE REPLACEMENT ALGORITHMS

PAGE REPLACEMENT

- When doing a page replacement the operating system has to make decisions regarding the fetching, placement and replacement of the pages
  - **Fetch Strategies** decides **when** a page should be brought into main memory from secondary storage (demand, request, & pre-paging)
  - **Placement Strategies** decides **where** a page should be placed in the main memory
  - **Replacement Strategies** decides **which** page (victim page) from main memory should be swapped out on disk if there is no more free frame available with that process (Replacement algos, e.g., FIFO, Optimal, LRU, …)
    - Page fault service routine is modified to include page replacement as well
    - Dirty bit is used to reduce overhead of page transfers, i.e., only modified pages are written back to disk. This bit is set by hardware when data is written to a page and is checked by operating system at page replacement time

PAGE EVICTION

- When doing a page replacement the operating system has to make decisions regarding the fetching, placement and replacement of the pages
- How do we replace page:
  - Find the location of 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 the victim frame
  - Read the desired page into the (newly) free frame.
  - Update the page table
  - Restart the process

EXPLOITING LOCALITY

- **Temporal Locality:** Memory accessed recently tends to be accessed soon
- **Spatial Locality:** Memory locations near recently accessed memory is likely to be accessed soon
- Locality helps reduce the frequency of paging, i.e., once something is in memory, it should be used many times.
- Fundamental technique to exploit locality is Caching, i.e., Keep a subset of dataset in a more accessible but space-limited location
- Caches are every where in system:
  - Registers are cache for L1 cache, which is a cache for L2 cache, which is a cache for memory, which is a cache for disk, which might be a cache for a remote file server.
- Key goal: minimize cache miss rate
  - Minimize page fault rate (in context of paging), which requires a good algorithm

EVICTING THE BEST PAGE

- Goal of the page replacement algorithm:
  - Reduce page fault rate by selecting the best page to evict.
- The best pages are those that will never be used again
  - However, it is impossible to know in general whether a page will be touched
  - If you have information on future access patterns, it is possible to prove that evicting those pages that will be used the furthest in the future will minimize the page fault rate.
- Principle of Optimality
  - Replace the page that will not be used again the farthest time in the future

PAGE REPLACEMENT ALGORITHMS

- FIFO - First in First Out
  - Replace the page that has been in main memory the longest
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time
- LFU - Least Frequently Used
  - Replace the page that is used least often
- Optimal
  - Evict page that won't be used for the longest time in the future
- Random Page Replacement
  - Choose a page randomly
- Working Set
  - Keep in memory those pages that the process is actively using

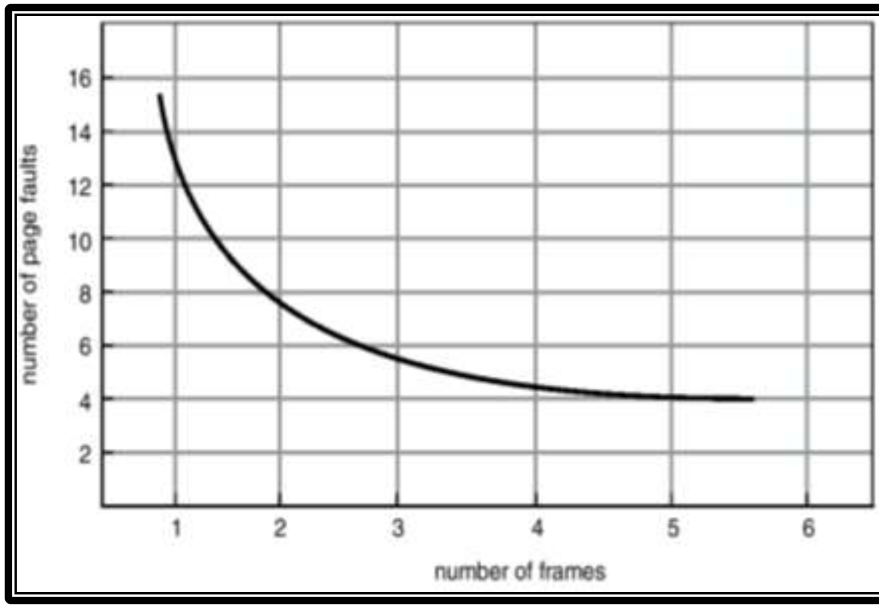
NOTE: Want lowest page-fault rate. Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

FIRST IN FIRST OUT

1. Simplest algorithm
2. Associates with each page the time when that page was brought into memory. When a page is to be replaced the oldest is chosen
3. Instead of recording time when a page is brought in, we use a FIFO queue
4. Suffers with Belady's anomaly. "For some page replacement algorithms, by increasing the allocated frames to a process, the page fault rate may also increase"

BELADY'S ANOMALY

1. Given a reference string, it would be natural to assume that: The more the total number of frames in main memory, the fewer the number of page faults.
2. Belady's anomaly: For some page replacement algorithms, by increasing the allocated frames to a process, the page fault rate may also increase
3. Not true for some algorithms like FIFO



FIRST IN FIRST OUT (cont…)

A sample string showing the Belady's anomaly in FIFO algorithm

- Number of frames allocated = 3
- Reference string:
- 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Number of page faults = 9

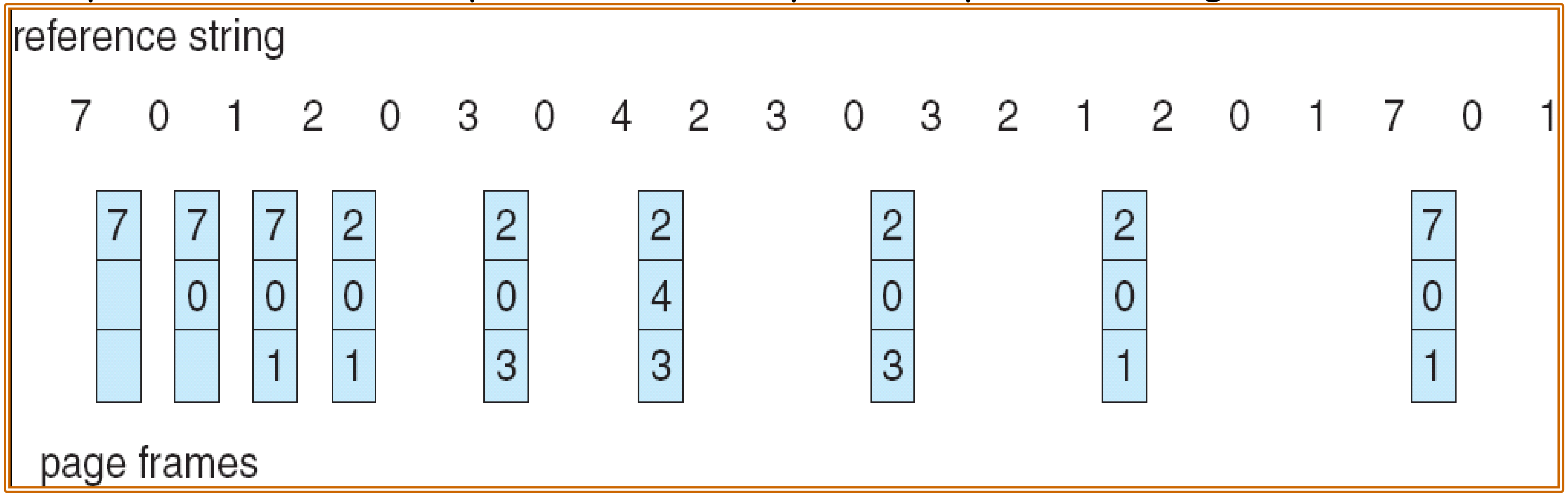
1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

- Number of frames allocated = 4
- Reference string:
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Number of page faults = 10

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

OPTIMAL

1. Best possible algorithm, but impossible to implement
2. Replace the page that will not be used for the longest period of time. (Look forward but how?)
3. Has the lowest page fault rate
4. Never suffer from Belady's anomaly
5. Unrealizable, because at the time of page fault, the OS has no way of knowing when each of the pages will be referenced next
6. Used as Bench Mark, whenever an algorithm is to be evaluated its performance is compared with the optimal replacement algorithm



LRU

- An approximation of optimal algorithm; assumes that pages that have not been used for ages will probably remain unused for a long time
- LRU policy replaces the page in memory that has not been referenced for the longest time
- Example
  - Number of frames allocated = 3
  - Reference string:

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Number of page faults = 10

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

LRU IMPLEMENTATION

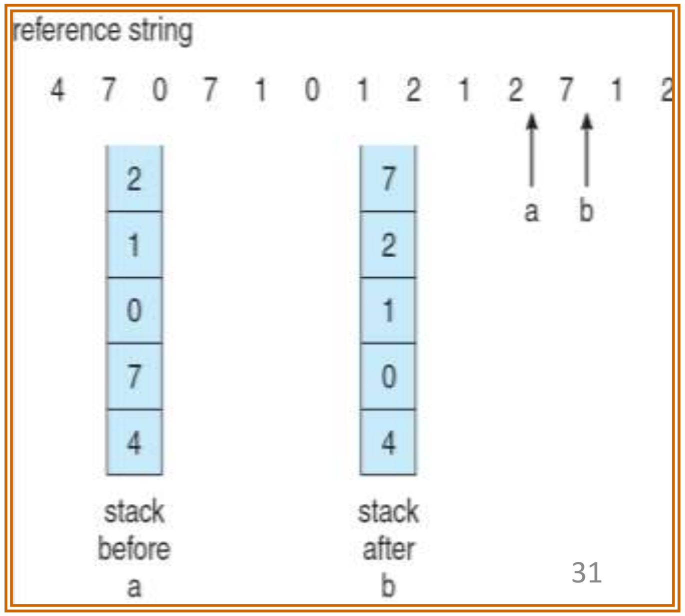
- **Counter Based Implementation.** Every page entry has a counter, every time page is referenced through this entry, copy the clock into the counter. When a page needs to be replaced, choose the one with minimum counter value
- **Stack Based Implementation.** Keep a stack of page numbers. Whenever a page is referenced remove it form its current location and push it at the top of stack. The LRU page is at the bottom of the stack. So when you need to replace a page, you replace the page whose number is at the bottom of the stack
- *A* doublee linked list implementation of stack requires six pointer changes ?

Reference string

4 7 0 7 1 0 1 2 1 2 $\uparrow$ 7 $\uparrow$ 1 2

a b

Give the contents of stack before a and after b



COUNTING BASED PAGE REPLACEMENT

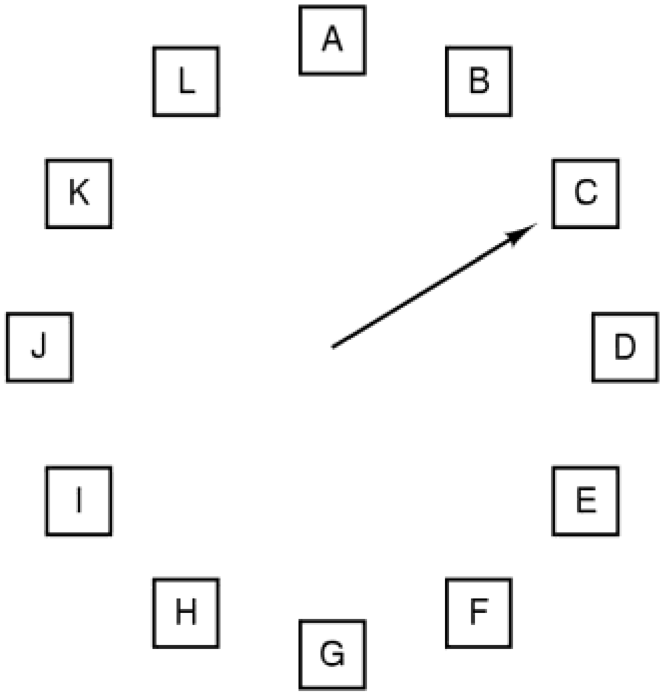
- Keep a counter of the number of references that have been made to each page and develop the following two schemes:
- **Least Frequently Used (LFU)**
  - This algo is based on locality of reference concept; the least frequently used page is not in the current locality
  - Replace the page with the smallest reference count. The reason of this selection is that an actively used page should have a large reference count. The problem is if a page is used heavily during the initial phase of a process but then is never used again. Since it was heavily used in the beginning it has a large count and remains in memory even though it is no longer needed. Solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average user count
- **Most Frequently Used (MFU)**
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used; i.e., it will be in the locality that has just started
  - Neither of these is used as they are expensive and they do not approximate Optimal replacement algorithm well.

SECOND CHANCE PAGE REPLACEMENT

- Maintain a FIFO
- Pages kept in a link list, with oldest at the end of the list
- Look at the oldest page
  - If reference bit equals 0
    - Select page for replacement
  - Else
    - Page was recently used, so don't replace it.
    - Clear reference bit
    - Move to the end of the list
- What if the page was used in the last clock tick?
  - Select a page at random

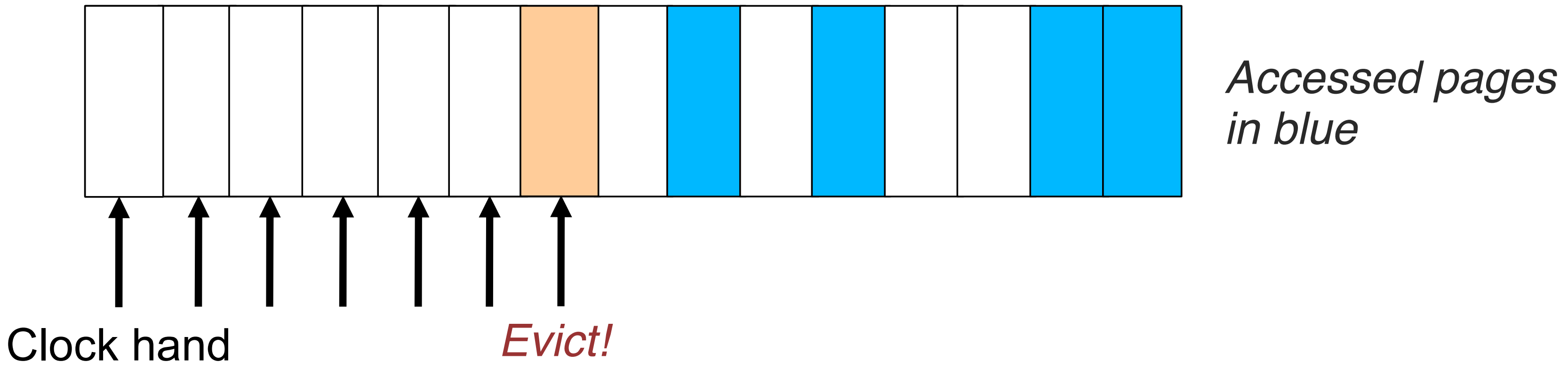
CLOCK ALGORITHM

- Maintain a circular list of pages in memory



CLOCK ALGORITHM

- Maintain a circular list of pages in memory



PAGE REPLACEMENT ALGORITHMS (cont…)

Drawbacks of FIFO

1. A page which is being accessed quite often may also get replaced because it arrived earlier than those present
2. Ignores locality of reference. A page which was referenced last may also get replaced, although there is high probability that the same page may be needed again

Drawbacks of LRU

1. A good scheme because focuses on replacing dormant pages and takes care of locality of reference, but a page which is accessed quite often may get replaced because of not being accessed for some time, although being a significant page may be needed in the very next access

Drawbacks of NFU

1. Does not take care of locality of reference, and reacts slowly to changes in it. If a program changes the set of pages it is currently using, the frequency counts will tend to cause the pages in the new location to be replaced even though they are being used currently

Pages like page directory are very important and should not at all be replaced, thus some factor indicating the context of the page should be considered in every replacement policy, which is not done in any of above

SAMPLE PROBLEMS

Problem 56 Compute total page faults for the given page Trace for 4 frames using following algorithms:

- FIFO
- OPTIMAL
- LRU
- Page Trace/Reference String is given below:

2, 3, 4, 5, 6, 4, 5, 2, 7, 8, 9, 4, 9, 0, 8, 9, 1, 6, 5, 6, 5, 3

Problem 57 Compute total page faults for the given page Trace for 3 frames using FIFO, Optimal, and LRU algorithm with:

- Page Trace

6, 5, 4, 3, 0, 6, 5, 1, 3, 2, 5, 6, 4, 3, 2, 1, 6, 0, 3, 4, 1, 6

Problem 58 Compute total page faults for the given page Trace:

8, 9, 0, 1, 2, 0, 1, 8, 3, 4, 5, 0, 5, 6, 4, 5, 7, 2, 1, 2, 1,9

- Do it using 4 frames and then using 3 frames.
- Use following algorithms.
  a. FIFO
  b. LRU
  c. OPTIMAL

SAMPLE PROBLEMS

Problem 59

- Consider a paging system with physical memory of 12 Bytes & Page size of 4 Bytes. A program stores and accesses characters A to Z. Compute the physical addresses of logical addresses where following program instructions are loaded:

  F, S, V, A, O, K, D, J, X, Z, H, A
- Use FIFO while constructing the Page Table
- Use Optimal while constructing the Page Table
- Use LRU while constructing the Page Table

Problem 60

- Consider a paging system with physical memory of 12 Bytes & Page size of 4 Bytes. A program stores and accesses characters A to Z. Compute the physical addresses of logical addresses where following program instructions are loaded:

  E, U, X, B, N, J, F, L, W, Y, H, A
- Use FIFO while constructing the Page Table
- Use Optimal while constructing the Page Table
- Use LRU while constructing the Page Table

Solution of Previous Problem

Instruction	E	U	X	B	N	J	F	L	W	Y	H	A
Page # ⇨												
F#0												
F#1												
F#2												
Hits/Misses												

Total%Page%Faults:%_______%

Logical%Memory%

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P
16	Q
17	R
18	S
19	T
20	U
21	V
22	W
23	X
24	Y
25	Z

Page%Table%

0	
1	
2	
3	
4	
5	
6	
7	

Physical%Memory%

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

Instruction	L.A	P#	Physical Address
E			
U			
X			
B			

ALLOCATION OF FRAMES

- Minimum number of frames that can be allocated to a process is dependent upon architecture. Maximum is dependent upon amount of available memory
- Each process needs a minimum number of frames so that its execution is guaranteed on a given machine. E.g. consider the following **mov** instruction on an IBM 370:

  MOV X,V

  - Instruction itself is 6 bytes, might span 2 pages
  - 2 pages to handle **from**
  - 2 pages to handle **to**
- So to execute the above instruction, we need to allocate a minimum of six frames, so that it can execute fully, else a page fault will keep occurring and the instruction will never be able to execute.
- Three major allocation schemes:
  - **Fixed Allocation**. Free frames are equally divided among processes
  - **Proportional Allocation**. Number of frames allocated to a process is proportional to its size

  Frames to be alloacate to P_i = {(No pages requested by P_i)/(Total demand of pages)} * (Available no of frames in the system)
  - **Priority Allocation**. Number of frames allocated as per the process priority

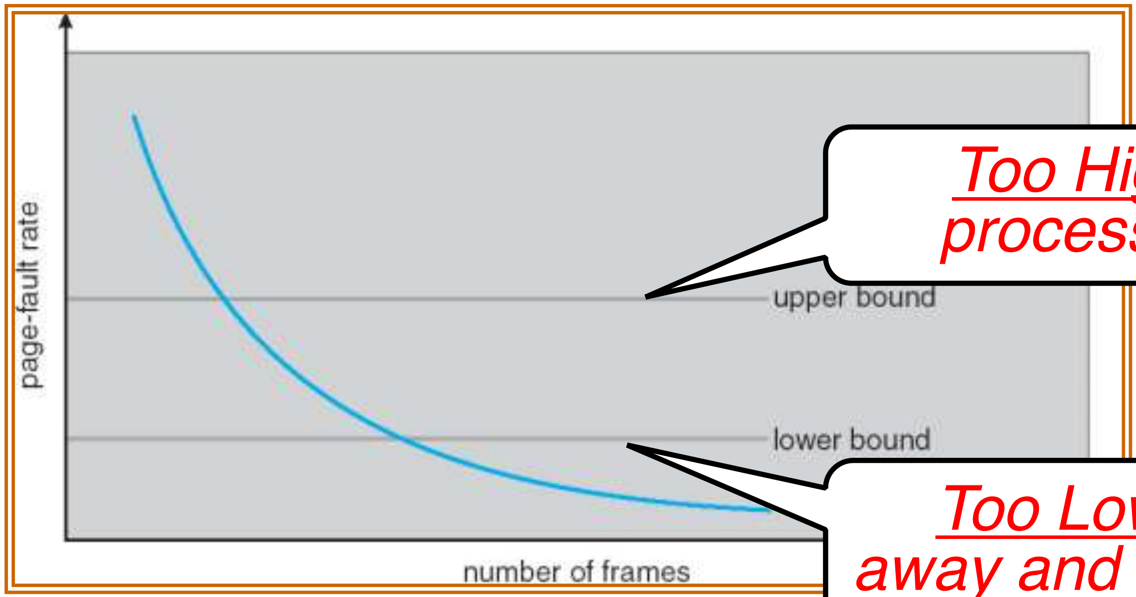
Problem 61

Consider a system with 64 frames. At a particular instant of time there are three processes P1(10 pages), P2(40 pages) and P3(127 pages).

Allocate frames to processes using Fixed and prop

PAGE FAULT FREQUENCY

- Can we reduce capacity misses by dynamically changing the number of frames per process?
- Establish acceptable page fault rate
  - If actual rate too low, process looses frames
  - If actual rate too high, process gains frames
- Question: What if we just don't have enough memory?

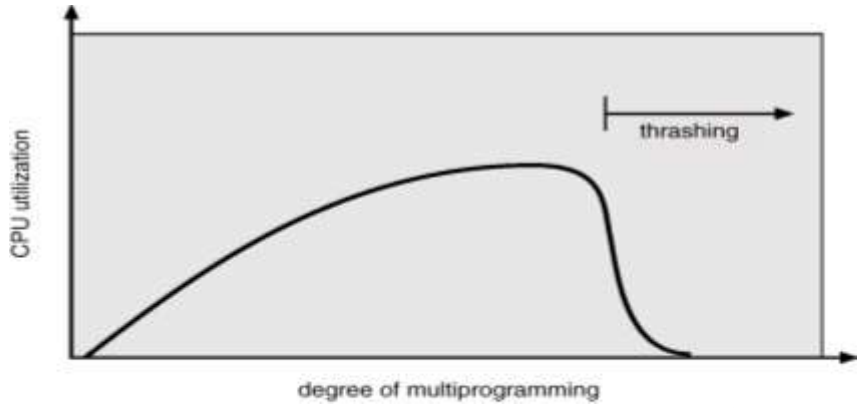


<u>Too High</u>: Need to give this process some more frames!

<u>Too Low</u>: Take some frames away and give to other processes!

THRASHING

- If a process does not have "enough" frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization
  - OS thinks that it needs to increase the degree of multi - programming
  - Another process is added to the system and that cause serious problems
- Thrashing is a phenomenon in which CPU spends much of its time swapping pages in and out, rather than executing instructions
- Thus in order to stop thrashing:
  - The degree of multiprogramming needs to be reduced
  - The effects of thrashing can be reduced by using a local page replacement. So if one process starts thrashing it cannot steal frames from another process and cause the later to thrash also
- Thrashing results in severe performance problems:
  - Low CPU utilization
  - High disk utilization
  - Low utilization of other I/O devices



THRASHING EXAMPLE

- Consider a process having 3 frames allocated to it. Page replacement algorithm it uses is LRU. Reference string is: 1234123412341234……….4th access onward all causes page fault

- **What do humans do when thrashing?**
  - If flunking all courses at midterm time, withdraw one or two
- **What does OS do?**
  - If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash
  - If the problem arises because of the sum of several processes
    - Figure out how much memory each process needs
    - Change scheduling priorities to run processes in groups whose memory needs can be satisfied

RESIDENT SET MANAGEMENT

- **Resident Set Size** is the number of frames allocated to a process (can be fixed or variable)
- **Replacement Scope** can be local or global; activated on a page fault and there are no free frames available with the process

	Local Replacement	Global Replacement
Fixed Allocation	• Number of frames allocated to process are fixed • Page to be replaced is chosen from among the frames allocated to that process	• Not possible
Variable Allocation	• Number of frames allocated to a process may be changed from time to time, to maintain the working set of the process • Page to be replaced is chosen from among the frames allocated to that process	• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary

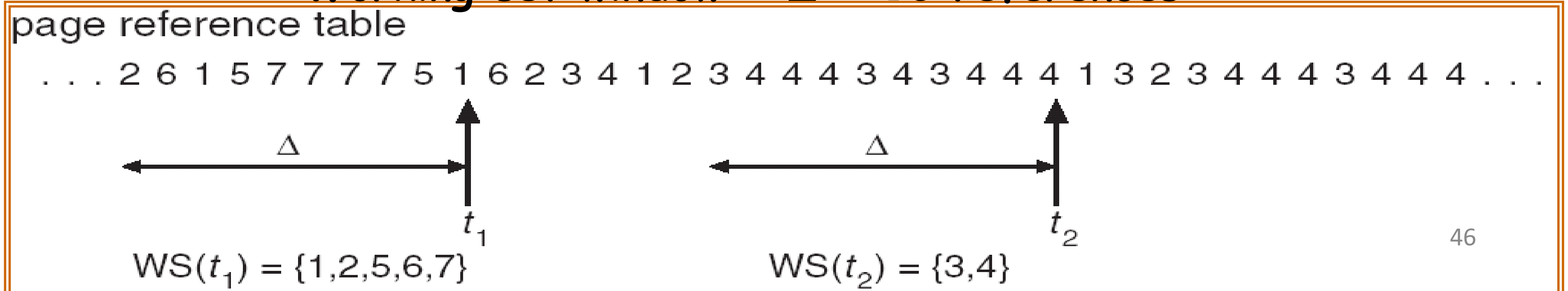
WORKING SET

- QUESTION:
  - How much memory does a process need to keep the most recent computation in memory with a very few page faults?
- How can we determine this:
  - Determine the working set of a process
  - The principal of locality
    - Temporal Locality
    - Spatial Locality

WORKING SET MODEL

- Working Sets are a solution proposed by Peter Denning. **WS is the collection of pages referenced by a process in last T seconds of execution. Must thus be resident if the process is to avoid thrashing**
- Working set changes over the life time of a process
  - Periods of high locality exhibits smaller WS
  - Periods of low locality exhibits larger WS
- WS Model uses a parameter Δ to define the **working set window**
- $\Delta \equiv$ A fixed number of page references, e.g., 10,000 instructions
- WSS_i is the working set size of Process P_i (varies in time)
- WSS_i = Total number of pages referenced in the most recent Δ
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working set window = Δ = 10 references



WORKING SET MODEL (cont…)

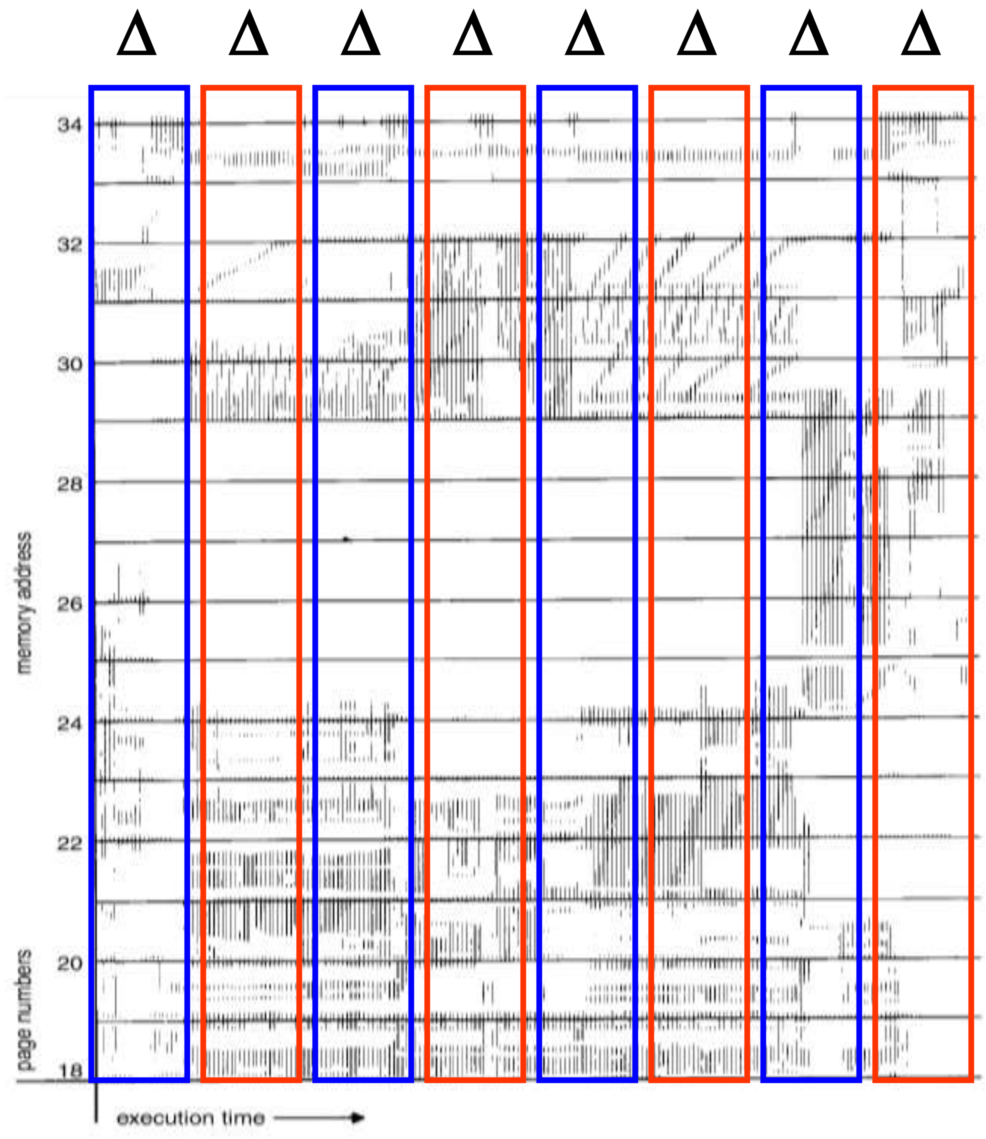
Locality of Reference

The first two and last localities are:

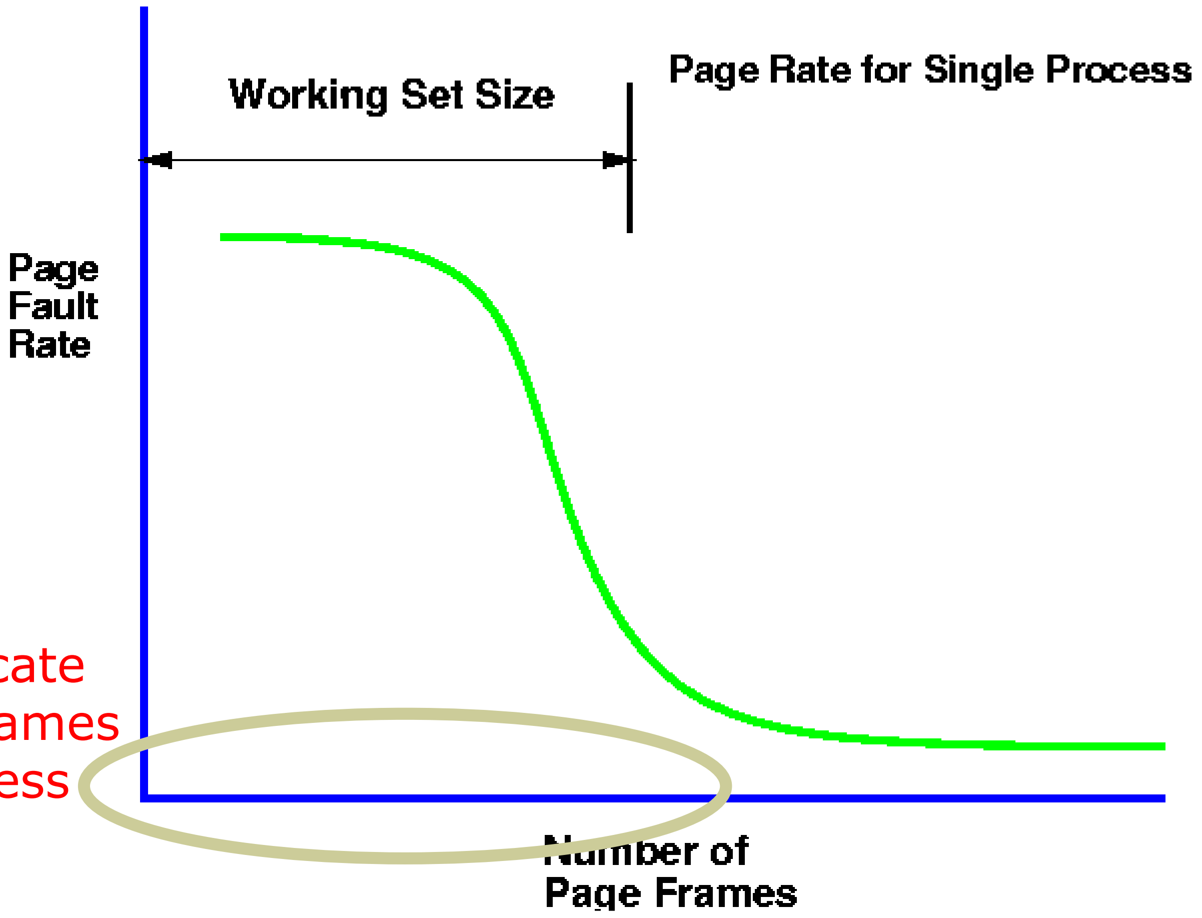
L1 = {18-26, 31-34}

L2 = {18-23, 29-31, 33}

Last = { 18-20, 24-34}



WORKING SET SIZE



At least allocate
this many frames
for this process

WORKING SET IN ACTION TO AVOID THRASHING

- Let at any instant of time, **D** is the total demand of frames by all processes, so we can say that **D** is equal to the sum of working set size of all processes in the system, i.e., $\mathbf{D} = \Sigma\, \mathbf{WSS}_i$
- Let at any instant of time, **m** is the total number of free frames available in the system
- Now, if at any time **D** becomes greater than **m**, the system will start thrashing, i.e., **if** $\mathbf{D} > \mathbf{m} \Rightarrow$ **Thrashing**
- So, WSs are not enough by themselves to make sure memory doesn't get overcommitted. We must also introduce the idea of **Balance Set**
- A **Balance Set** is a collection of pages that an **active process** is working with, and which must thus be resident if the process is to avoid thrashing
- Divide runnable processes up into two groups: active and inactive. When a process is made active its WS is loaded, when it is made inactive its WS is allowed to migrate back to disk
- If the sum of the WSs of all runnable processes (D) is greater than the size of available memory, then refuse to run some of the processes (for a while)

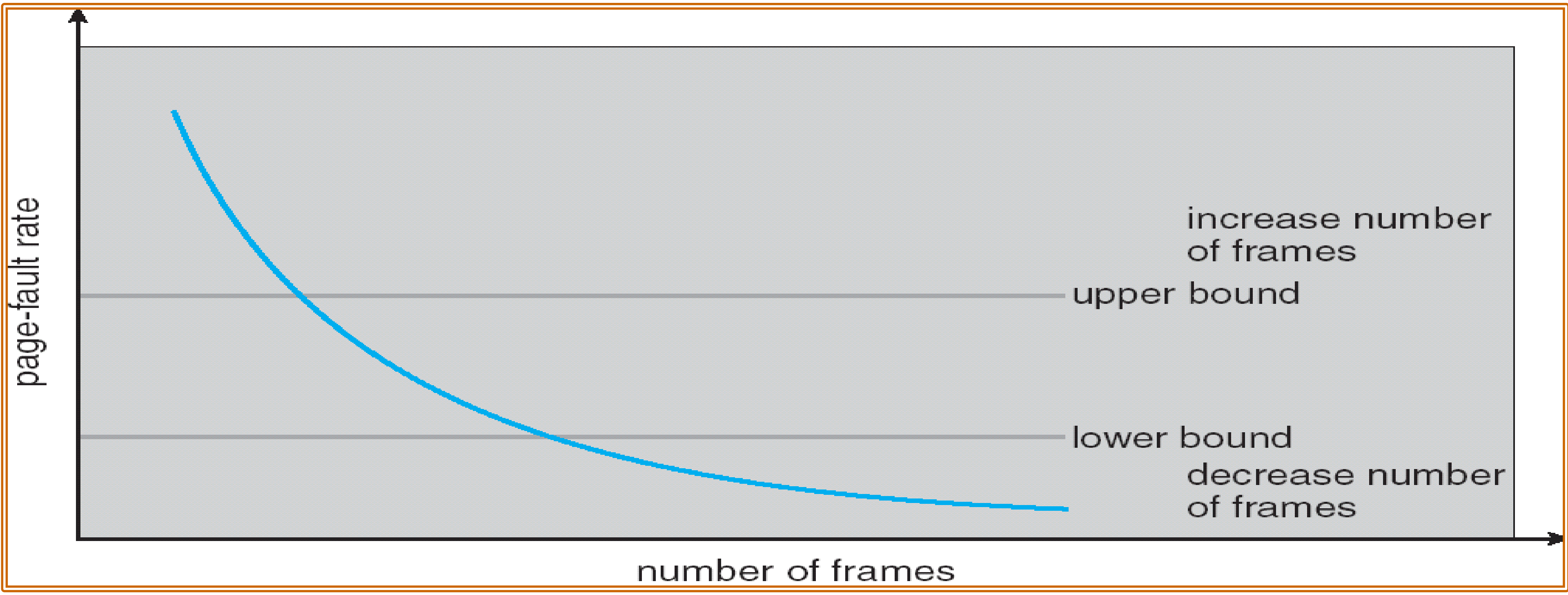
WORKING SET IN ACTION TO AVOID THRASHING

Keeping track of Working Set. How do we come to know at any given instant of time about the working set of a particular process?

- A WS can be implemented using a fixed interval timer and a reference bit
- Example
  - Δ = 10,000 references
  - Timer interrupts after every 5000 references
  - Copy and set the values of all reference bits to 0
  - Keep in memory two bits for every page, indicates if page was used within last 10000 to 15000 references
  - Whenever a timer interrupts, copy all reference bits and set the value of this copied bit to 0
  - After Δ references, if any one of the two bits, the reference bit or the copied bit for a page is 1 that means the page is in the working set. Means this page has been referenced in last Δ references
- **Why is this not completely accurate?**
- Improvement = For the same Δ = 10,000 references, use 10 reference bits and interrupt should occur after every 1000 references instead of 5000 references. (Do it at your own)

PAGE FAULT FREQUENCY

- This is another method to control thrashing
- Operating System keeps track of the upper and lower bounds on the page-fault rates of processes
- If page-fault rate falls below the lower limit, the processes loses frames. If page-fault rate goes above the upper limit, process gains frames



CLEANING POLICY

- A cleaning policy is the opposite of a Fetch policy; it is concerned with determining when a modified page should be written out to secondary memory. Two common alternatives are:
- **Demand Cleaning**
  - A page is written out to secondary memory only when it has been selected for replacement
  - Writing of a dirty page is coupled to, and precedes, the reading in of a new page. This technique do minimize page writes, but it means that a process that suffers a page fault may have to wait for two page transfers before it can be unblocked
- **Pre-cleaning**
  - Write modified pages before their page frames are needed so that pages can be written out in batches
  - A page is written out but remains in main memory until the page replacement algorithm dictates that it need to be removed. OS may need to write the same page again and again on secondary storage

PAGE SIZE CONSIDERATION

- **Small Pages**
  - Large page tables
  - Minimizes internal fragmentation
  - Good for locality of reference
  - Disk seek time dominates transfer time (It takes the same amount of time to bring a large page as a small page page from disk to memory)
- **Large Pages**
  - Significant numbers of pages may not be referenced
  - Enable more data per seek
- **Real Systems may be configured**
  - Linux default page size is 4 KB
  - Windows default page size is 8 KB

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 9.1 to 9.7, 9.8.1, 9.9
- Solve all the sample problems given in slides to understand the concepts discussed in class