SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

18CSC205J-Operating Systems

Unit- IV

18CSC205J Operating Systems
Unit IV

Course Learning Rationale

- Emphasize the importance of Memory Management concepts of an Operating system
- Realize the significance of Device Management part of an Operating system

Course Learning Outcomes

- Understand the need of Memory Management functions of an Operating system
- Find the significance of Device management role of an Operating system

Learning Resources

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013
- William Stallings, Operating Systems-Internals and Design Principles, 7th ed., Prentice Hall, 2012
- Andrew S.Tanenbaum, Herbert Bos, Modern Operating systems, 4th ed., Pearson, 2015
- Bryant O‟Hallaxn, Computer systems- A Programmer ‟s Perspective, Pearson, 2015

Contents

- Virtual Memory– Background
- Understanding the need of demand Paging
- Virtual Memory – Basic concepts – page fault handling
- Understanding, how an OS handles the page faults
- Performance of Demand paging
- Understanding the relationship of effective access time and the page fault rate
- Copy-on write,
- Understanding the need for Copy-on write
- Page replacement Mechanisms: FIFO, Optimal, LRU and LRU approximation Techniques
- Understanding the Pros and cons of the page replacement techniques
- Counting based page replacement and Page Buffering Algorithms
- To know on additional Techniques available for page replacement strategies
- Allocation of Frames - Global Vs. Local Allocation
- Understanding the root cause of the Thrashing
- Thrashing, Causes of Thrashing
- Understanding the Thrashing
- Working set Model
- Understanding the working set model for controlling the Working set Model

Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in Main memory.

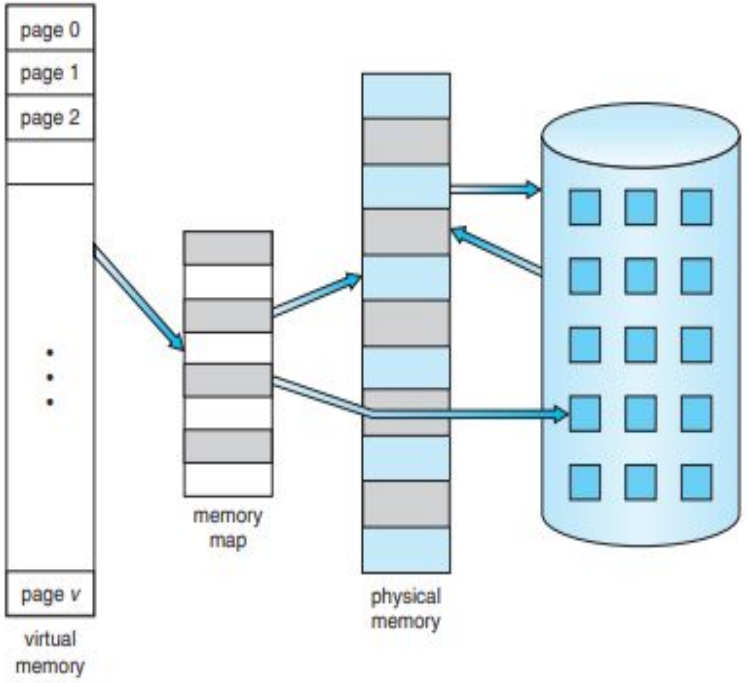
Virtual memory – involves the separation of logical memory as perceived by users from physical memory.

Need for virtual memory

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols. Certain options and features of a program may be used rarely. Even in those cases where the entire program is needed, it may not all

Virtual Memory

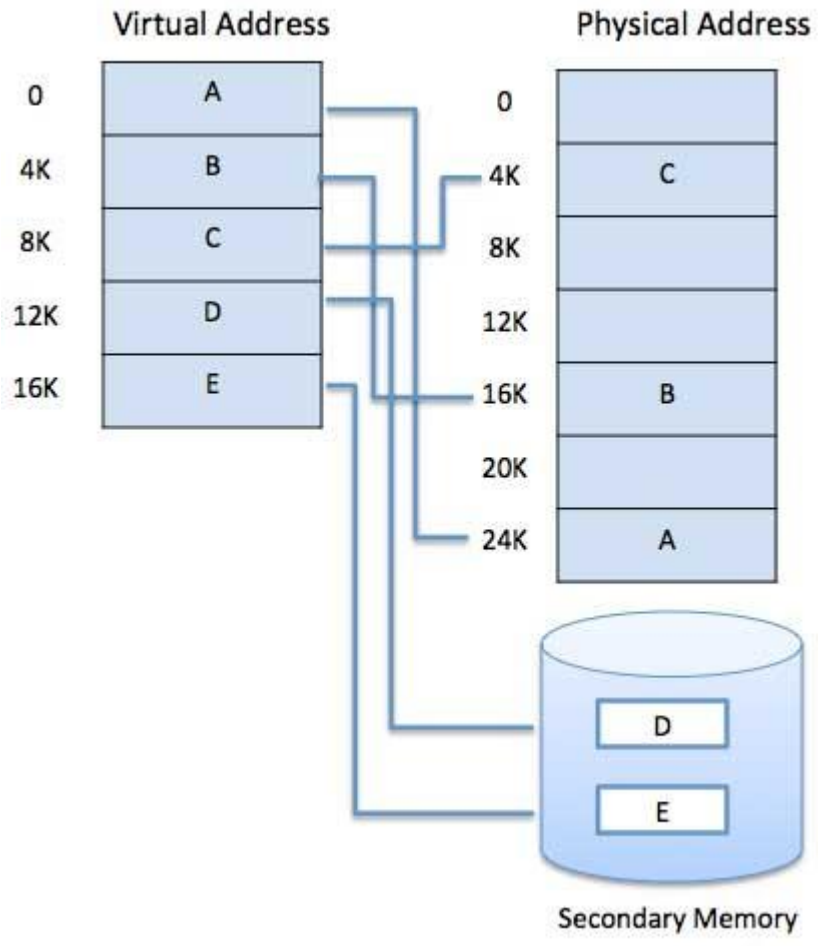
- One major advantage of this scheme is that programs can be larger than physical memory.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.



- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available.
- This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory.
- In addition, it provides an efficient mechanism for process creation.

Virtual Memory

- A program would no longer be constrained by the amount of physical memory that is available.
- Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- Because each user program could take less physical memory.
- More programs could be run at the same time, with a corresponding increase in CPU utilization and throughput.
- With no increase in response time or turnaround time.

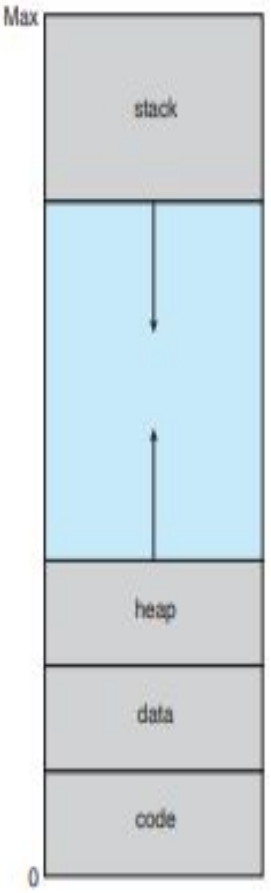


Virtual address space

- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
- The virtual address space for a process is the set of virtual memory addresses that it can use.
- The address space for each process is private and cannot be accessed by other processes unless it is shared.
- A virtual address does not represent the actual physical location of an object in memory.
- Instead, the system maintains a page table for each process, which is an internal data structure used to translate virtual addresses into their corresponding physical addresses.
- Each time a thread references an address, the system translates the virtual address to a physical address.

Virtual address space

- In this view, a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure.
- Here it is allowed the heap to grow upward in memory as it is used for dynamic memory allocation.
- Allows the stack to grow downward in memory through successive function calls.
- The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.
- Virtual address spaces that include holes are known as sparse address spaces.
- Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

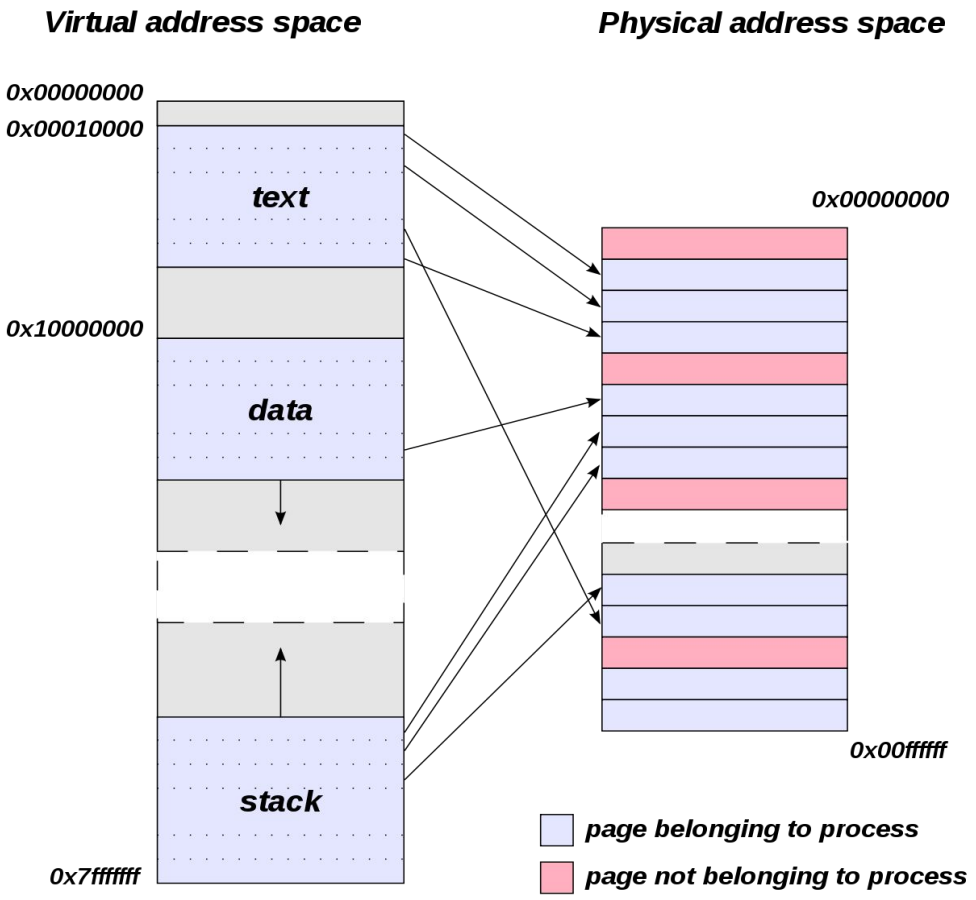


Virtual address space

Virtual Address Space vs Physical Address Space

- Address uniquely identifies a location in the memory.
- The logical address is a virtual address and can be viewed by the user.
- The user can't view the physical address directly.
- The logical address is used like a reference, to access the physical address.

- **Logical address** is generated by CPU during a program execution whereas, the **physical address** refers to a location in the memory unit.



Shared Library Using Virtual Memory

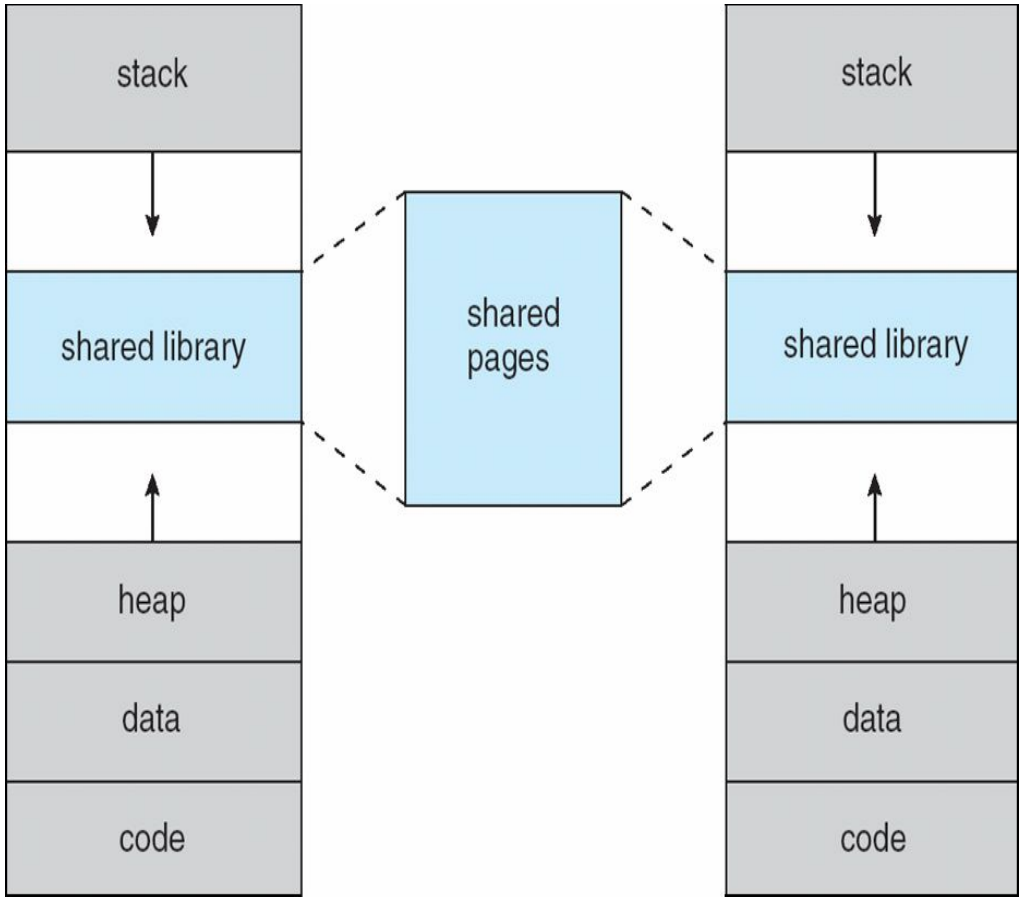
- In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing.

This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
- Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.
- Typically, a library is mapped read-only into the space of each process that is linked with it, Similarly, processes can share memory.
- Two or more processes can communicate through the use of shared memory.
- Pages can be shared during process creation with the fork() system call, thus speeding up process creation.

Shared Library Using Virtual Memory

- Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure.



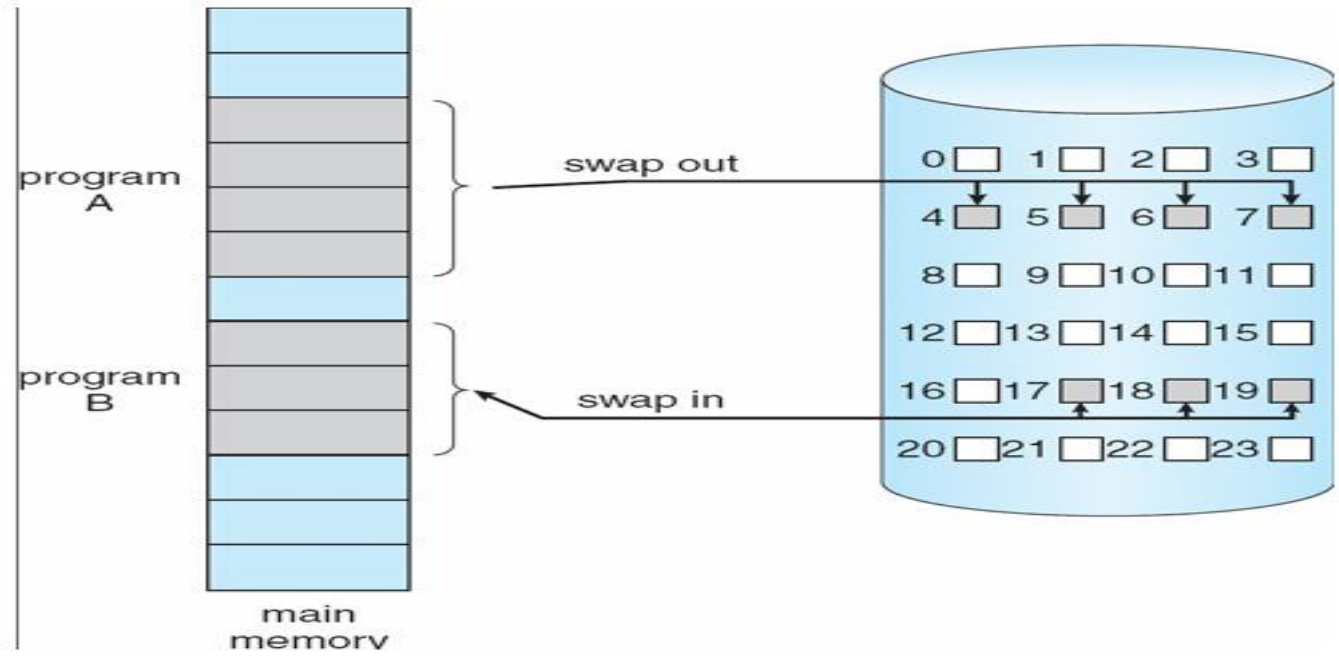
Demand Paging – Basic Concepts

How an executable program might be loaded from disk into memory?

- Load the entire program in physical memory at program execution time.
  - problem of this method, we may not initially need the entire program in memory
- An alternate approach to resolve this problem to load pages only as they are needed. This method called as **Demand Paging.**
- Demand paging commonly used in virtual memory systems.
- Demand paged virtual memory, pages are loaded only when they are needed during program execution.

Demand Paging

- A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance.

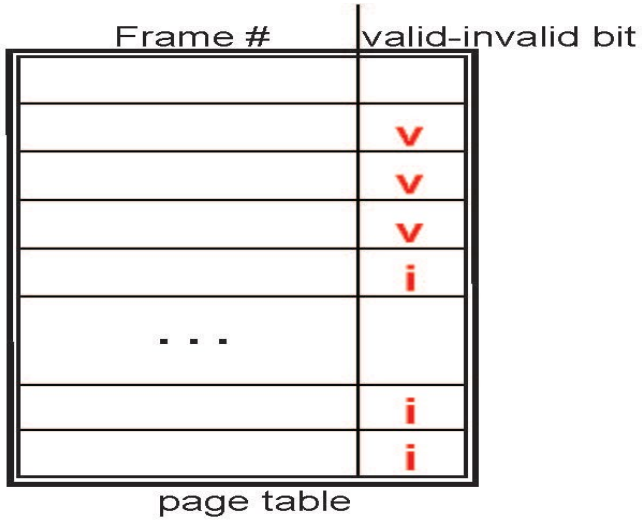


Demand Paging

- In the context of a demand-paging system, use of the term “swapper” is technically incorrect.
- Because **swapper manipulates entire process into memory,** whereas a pager is concerned with the individual pages of a process.
- We thus use “pager,” rather than “swapper,” in connection with demand paging.
- **Lazy swapper –** will not swap the entire process into memory, A lazy swapper never swaps a page into memory unless that page will be needed.

Valid-Invalid Bit

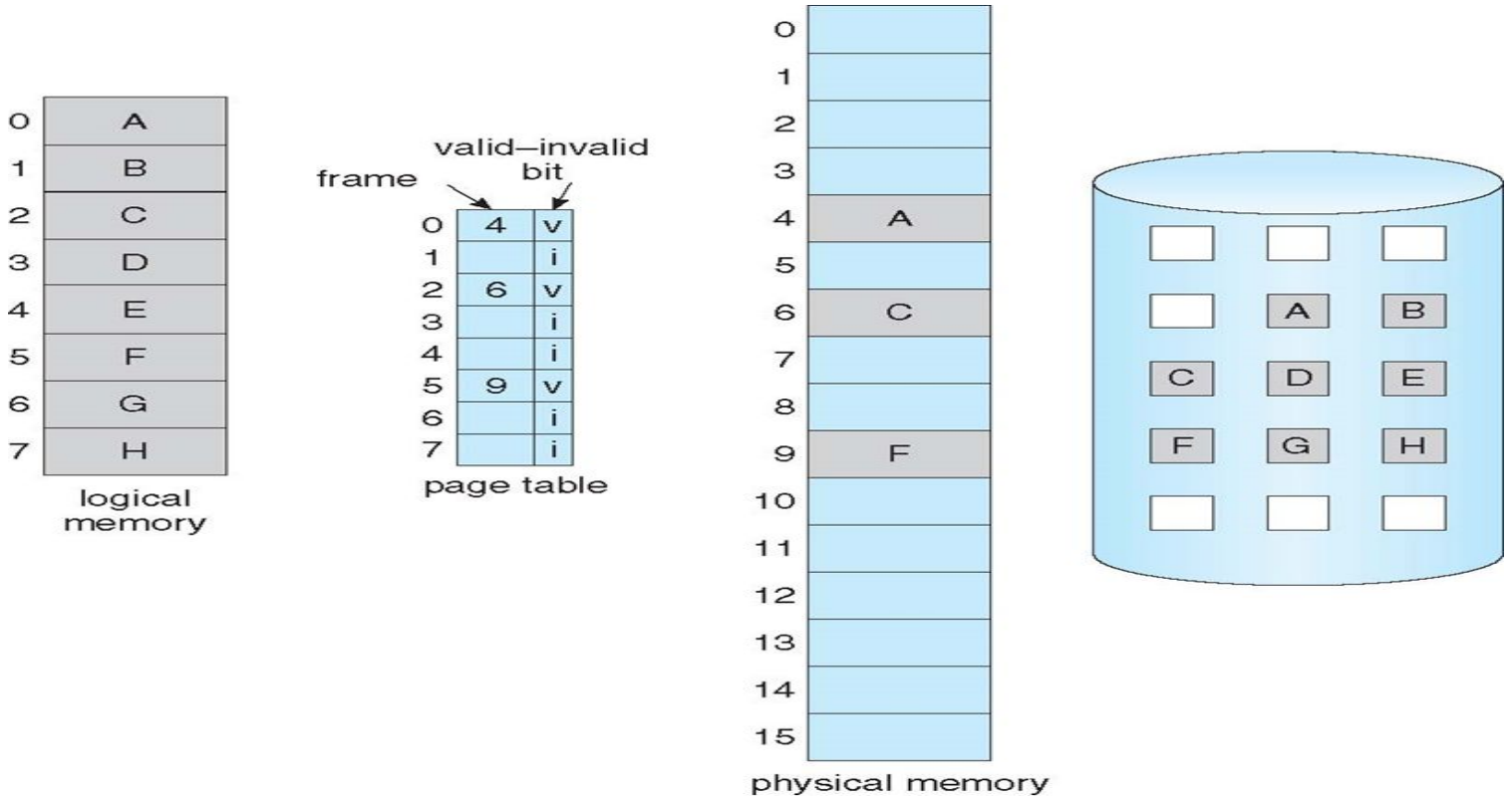
- Need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.
- The **valid–invalid bit** scheme can be used for this purpose.
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries.
- Example of a page table snapshot:



Page Table

When Some Pages Are Not in Main Memory

- **Page fault** - if the process tries to access a page that was not brought into memory.
- Operating system's failure to bring the desired page into memory.

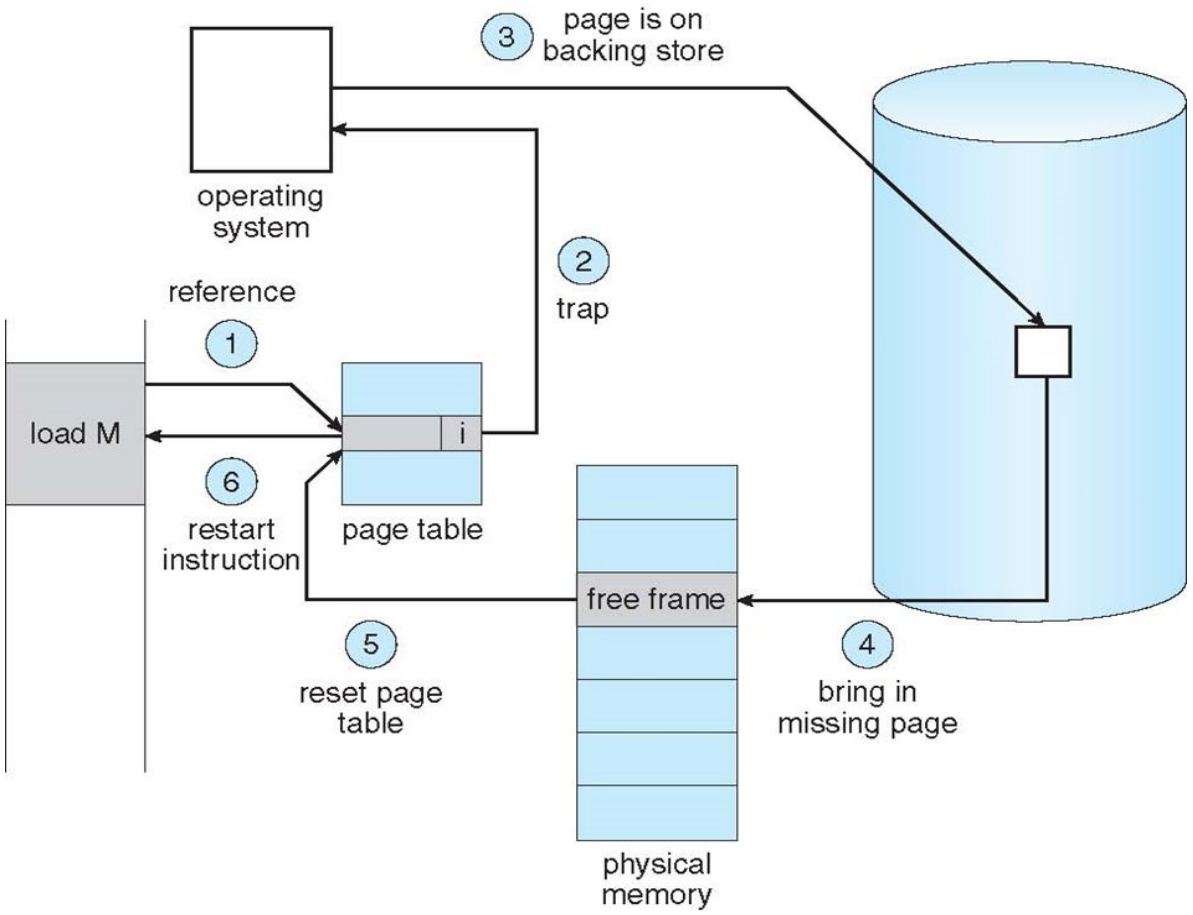


Page Fault

The procedure for handling page fault

1. Check an internal table (usually kept with the process control block) whether the reference was a valid or an invalid memory access.
2. If the reference
    1. **Invalid**, we terminate the process.
    2. **Valid** but we have not yet brought in that page, we now page it in.
3. Find a free frame
4. Schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Steps in Handling a Page Fault



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
  - After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.
  - At that point, it can execute with no more faults.
  - This scheme is **pure demand paging**: never bring a page into memory until it is required.
- The hardware to support demand paging
  - **Page table** - This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
  - **Secondary memory -** This memory holds those pages that are not present in main memory.
  - The secondary memory is usually a high-speed disk. It is known as the **swap device**

Performance of Demand Paging

- Demand paging can significantly **affect the performance** of a computer system.
- **Memory-access time** denoted **ma**, ranges from 10 to 200 nanoseconds. If there is no page faults, the **effective access time** is equal to the memory access time.
- If page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Let **p** be the probability of a page fault ($0 \leq p \leq 1$).
- We would expect **p to be close to zero,** expect to have only a few page faults.
- The effective access time is then

  effective access time = (1 − p) × ma + p × page fault time.

- To compute the effective access time, we must know how much time is needed to service a page fault.

Performance of Demand Paging

A page fault causes the following sequence to occur:

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
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

- Not all 12 steps are necessary in every case.
- When the CPU is allocated to another process while waiting I/O occurs.
  - Allows **multiprogramming** to maintain CPU utilization but **requires additional time** to resume the page-fault service routine when the I/O transfer is complete.
- In any case, we are faced with three major components of the page-fault service time:
  - Service the page-fault interrupt.
  - Read in the page.
  - Restart the process.

Performance of Demand Paging

- The first and third tasks can be reduced, with careful coding to several hundred instructions. These tasks may take from 1 to 100 microseconds each.
- Probably, The page-switch time - 8 milliseconds.
  - A typical hard disk has an average latency of 3 milliseconds,
  - A seek of 5 milliseconds
- Remember also that we are looking at only the device-service time.
- If a queue of processes is waiting for the device, we have to **add device-queueing time**

Performance of Demand Paging

For example,

Average page-fault service time - 8 milliseconds

memory access time - 200 nanoseconds

The effective access time?

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time.}$$

$$\text{effective access time} = (1 - p) \times (200) + p\ (8 \text{ milliseconds})$$

$$= (1 - p) \times 200 + p \times 8{,}000{,}000$$

$$= 200 + 7{,}999{,}800 \times p.$$

Let **p** be the probability of a page fault ($0 \leq p \leq 1$)

- We see, then, that the effective access time is **directly proportional** to the **page-fault rate**.

Performance of Demand Paging

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

Demand Paging

Advantages

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.

Copy-on-Write

- The fork() system call creates a child process that is a duplicate of its parent. Creating a **copy of the parent's address space for the child,** duplicating the pages belonging to the parent.

- Child processes invoke the exec() system call immediately after creation, the copying of the parent's address space

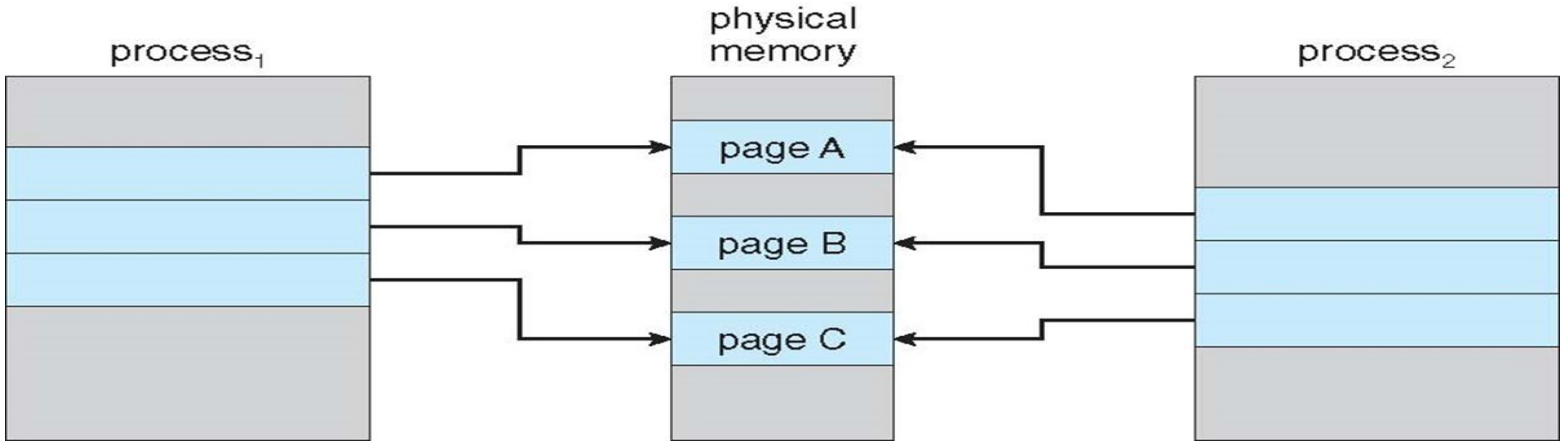
- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied

- COW allows more efficient process creation as only modified pages are copied

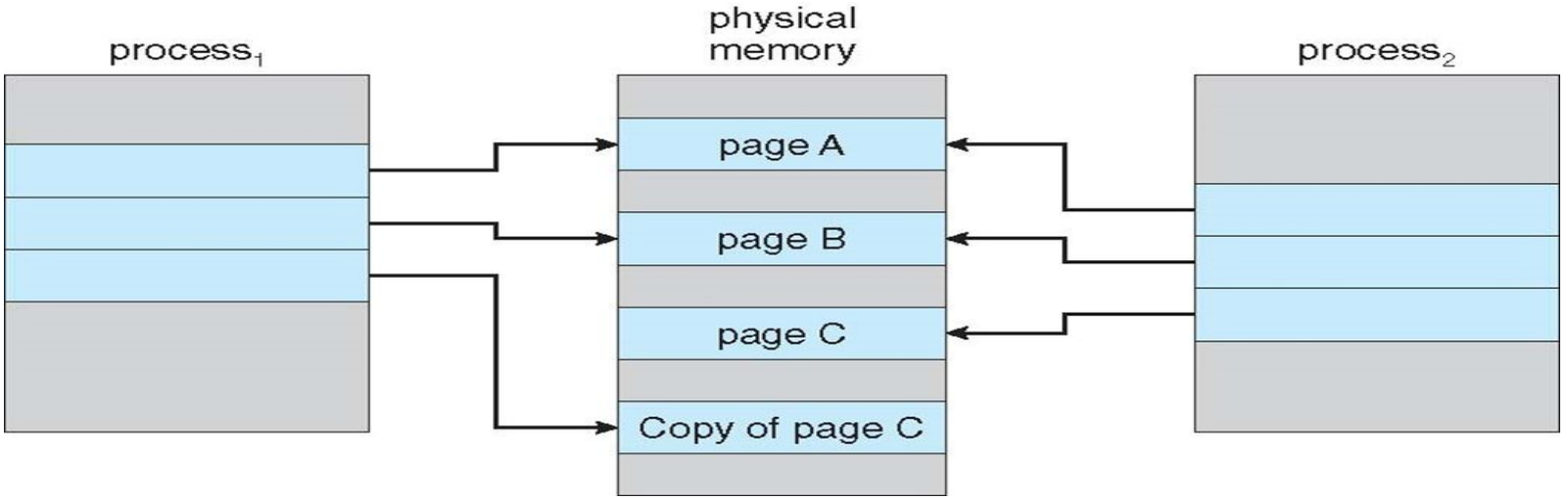
Copy-on-Write

- The child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
- The operating system will create a copy of this page, mapping it to the address space of the child process.
- The child process modify its copied page and **not the page belonging to the parent process.**
- Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; **all unmodified pages can be shared by the parent and child processes.**
- The following figure shows copy on write on page C

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



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

Need for Copy-on-Write

- **Copy-on-write or CoW** is a technique to efficiently copy data resources in a computer system.
- If a unit of data is copied but not modified, the "copy" can exist as a reference to the original data.
- Only when the copied data is modified is a copy created, and new bytes are actually written.
- Copy-on-write is closely related to data deduplication.
- Whereas data deduplication analyzes chunks or blocks of data, copy-on-write applies to entire files or allocated units of memory.

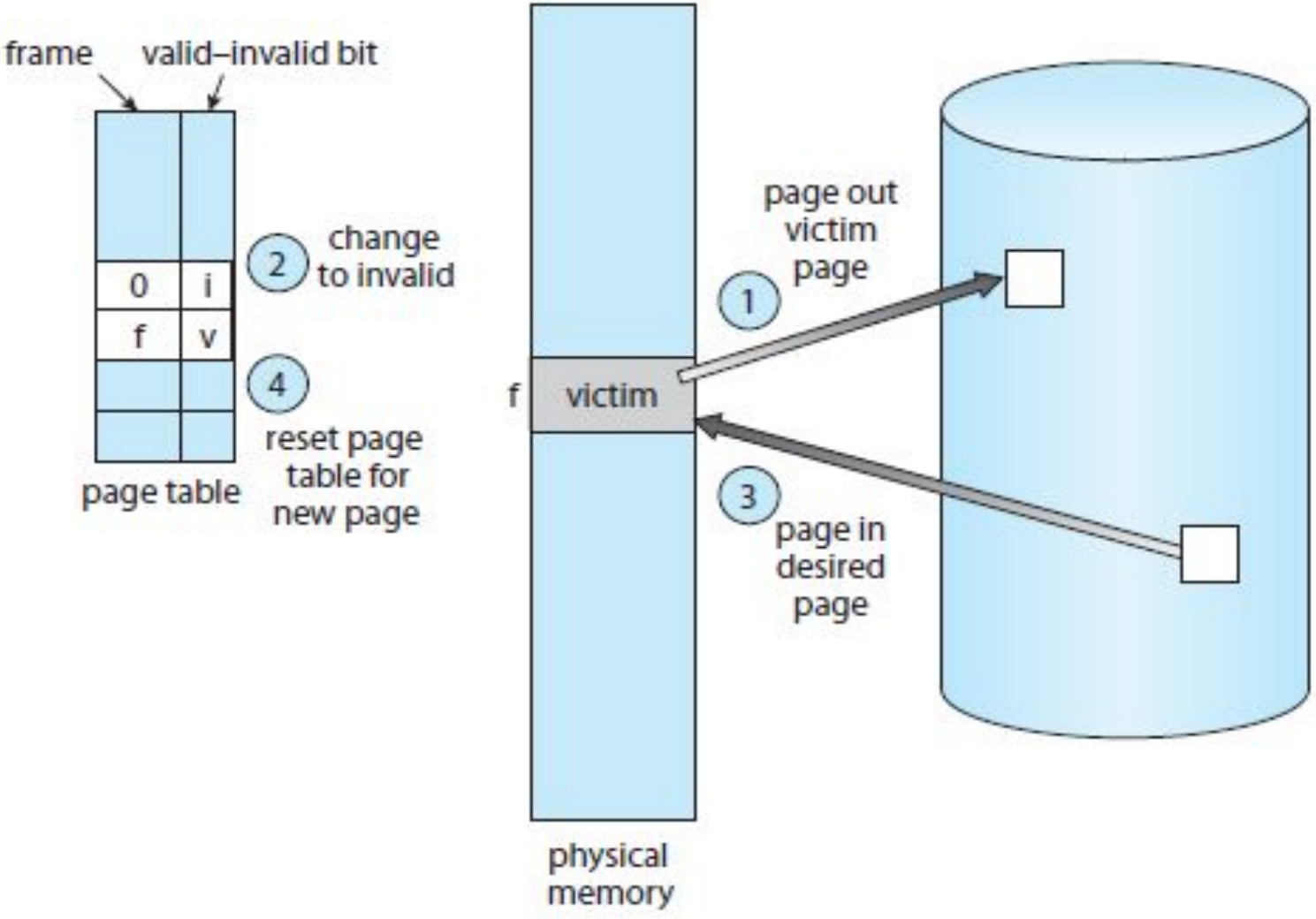
Page Replacement Algorithm

- Page replacement is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page.
- Reason for Page Replacement
  - Page fault
    - A page fault occurs when a page referenced by the CPU is not found in the main memory.
    - The only solution for the page fault is, the required page has to be brought from the secondary memory into the main memory.
    - This is done by replacing the already occupied page in the main memory with the desired page.

Steps in Page Replacement

- Identify the location of the desired page on the disk.
- If the page fault occurs, identify a free frame
- If there is a free frame then locate the desired page into the free frame
- If there is no free frame then use a page-replacement algorithm to select a victim frame that is to be replaced.
  - Now write the victim page to the disk then update the page and frame tables.
  - Read the desired page into the newly freed frame then change the page and frame tables.
  - Continue the process when there is a page fault occurs again

Steps in Page Replacement



Page Replacement Algorithms

- Page replacement algorithms help to decide which page must be swapped out from the main memory to create a room for the incoming page.
- Various page replacement algorithms are
  - First In First Out (FIFO) page replacement algorithm
  - Optimal page replacement algorithm
  - Least Recently Used (LRU) page replacement algorithm
  - LRU Approximation page replacement algorithm
    - Additional Reference Bits Algorithm
    - Second Chance / Clock Algorithm
    - Enhanced Second Chance Algorithm
  - Counting Based page replacement algorithm
    - Least Frequently Used (LRU) page replacement algorithm
    - Most Frequently Used (MRU) page replacement algorithm
  - Page Buffering algorithm

FIFO Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of "**First in First out**".
- It replaces the oldest page that has been present in the main memory for the longest time.
- It is implemented by keeping track of all the pages in a queue.
- **Steps in FIFO**
  - Bring the page into the queue
  - If page is available then do
  - Else there is a page fault
    - If there is a free frame, locate the desired page which causes the page fault
    - If there is no free frame, identify the victim frame to be replaced
    - The page which comes first into the queue will be chosen as the victim frame.
    - Now this victim frame will be replaced with the desired page and the queue is updated for further process.

FIFO Page replacement algorithms

FIFO Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in FIFO = 6

4*
7
6

Initial Queue

7*
6
1

Queue after first replacement

6*
1
2

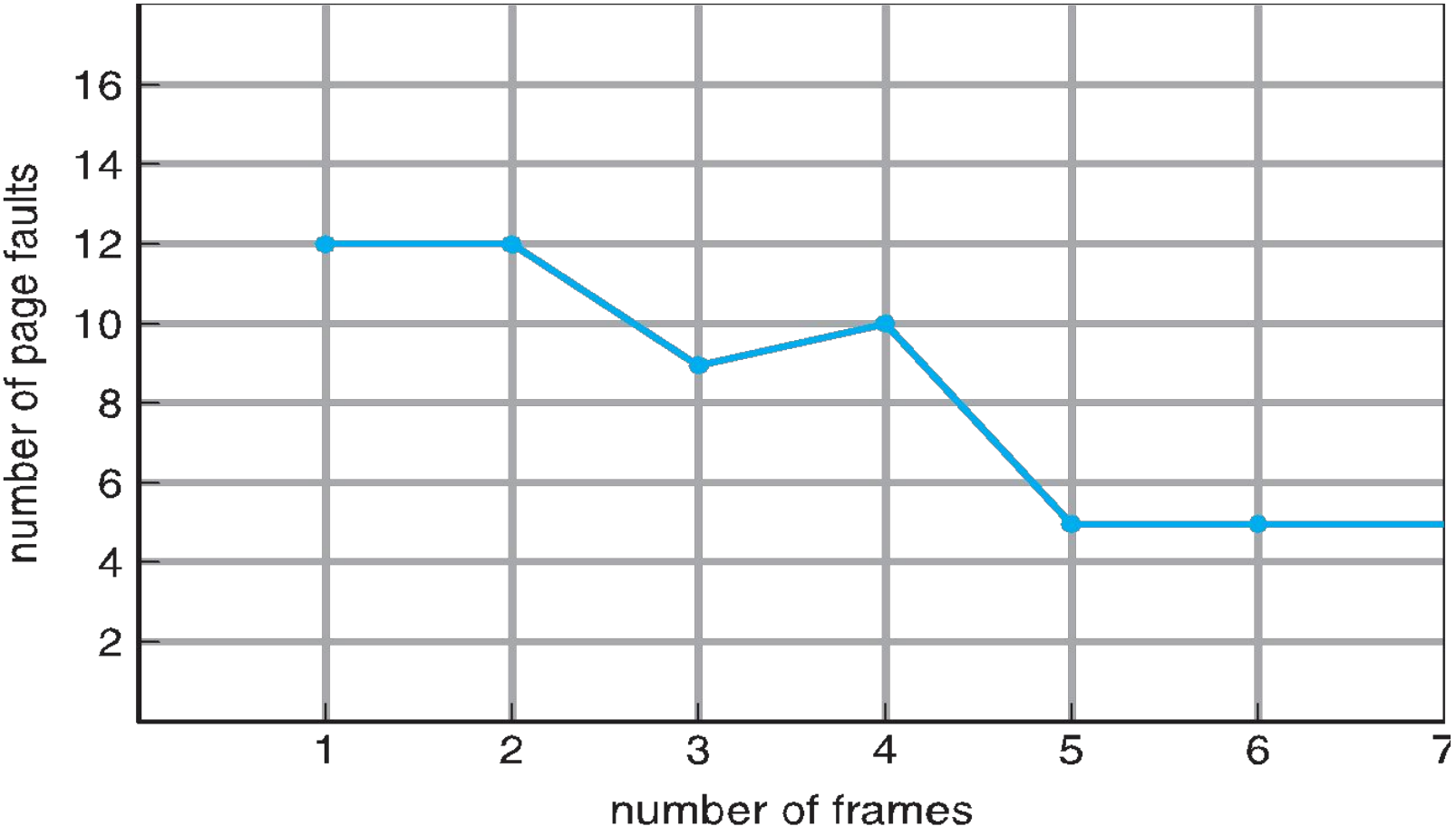
Queue after second replacement

1*
2
7

Queue after third replacement

* Indicates the top element of the queue or the element which comes first into the queue

FIFO Illustrating Belady's Anomaly



If number of page frames increases page fault also increases. This is called as Belady's Anomaly

Optimal Page Replacement Algorithm

- This algorithm replaces the page that will not be referred by the CPU in future for the longest time.
- It is practically difficult to implement this algorithm.
- This is because the pages that will not be used in future for the longest time cannot be predicted.
- However, it is the best known algorithm and gives the least number of page faults.
- Hence, it is used as a performance measure criterion for other algorithms.

Optimal Page Replacement Algorithm

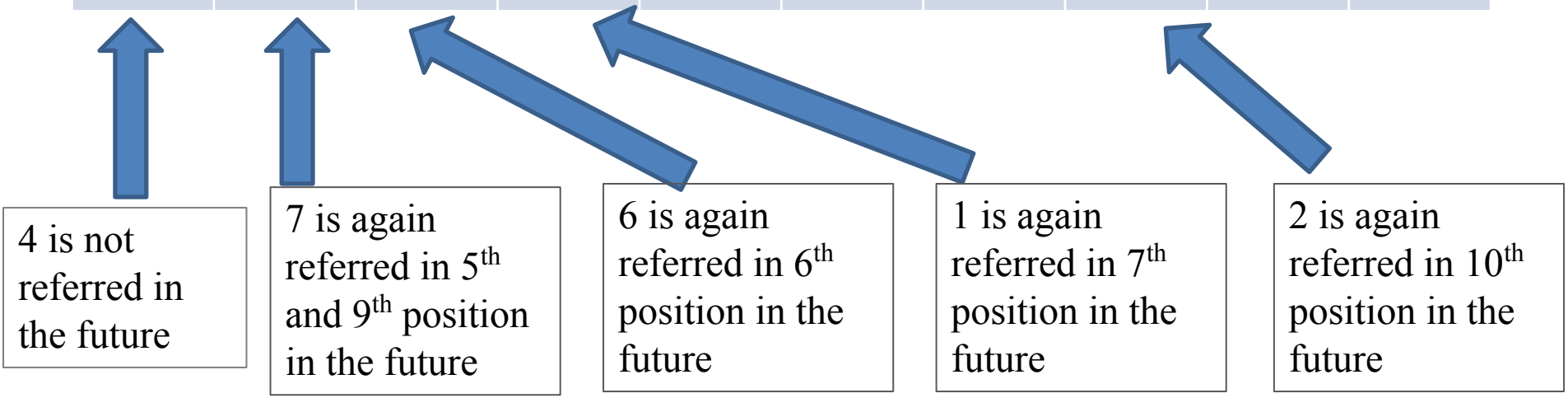
- Steps in the Working Process (This is only for understanding purpose)
  - Create required number of columns based on the reference strings used
  - A pointer is used which moves from MSB to LSB of the columns
  - The pointer identifies the future reference of the strings
  - If the page fault occurs, identify the free frame
    - If there is a free frame, locate the desired page into the free frame
    - If there is no free frames, choose a victim frame that is to replaced with the desired page
    - The victim frame is chosen by the pointer, where the pointer choses a victim frame which will be referred very later in the future
    - Now the desired page is located in the main memory which caused the page fault

Optimal Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	2	2	2
Frame 2		7	7	7	7	7	7	7	7	7
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Hit	Hit

Number of Page Faults in Optimal Page Replacement Algorithm = 5

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



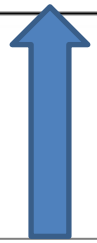
LRU Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of "**Least Recently Used**".
- It replaces the page that has not been referred by the CPU for the longest time.
- Use past knowledge rather than future
- Additionally LRU can be implemented using counter and stack

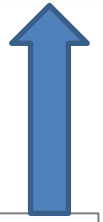
LRU Page Replacement Algorithm

Positions	1	2	3	4	5	6	7	8	9	10
Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in LRU = 6



4 is referred in the 1st position before, so it is replaced



7 is referred in the 2nd position before, so it is replaced

6 is referred in the 3nd position before, so it is replaced

LRU IMPLEMENTATION

Counter implementation

- Every page entry has a counter
- every time page is referenced through this entry
- When a page needs to be changed, look at the counters to find smallest value Search through table

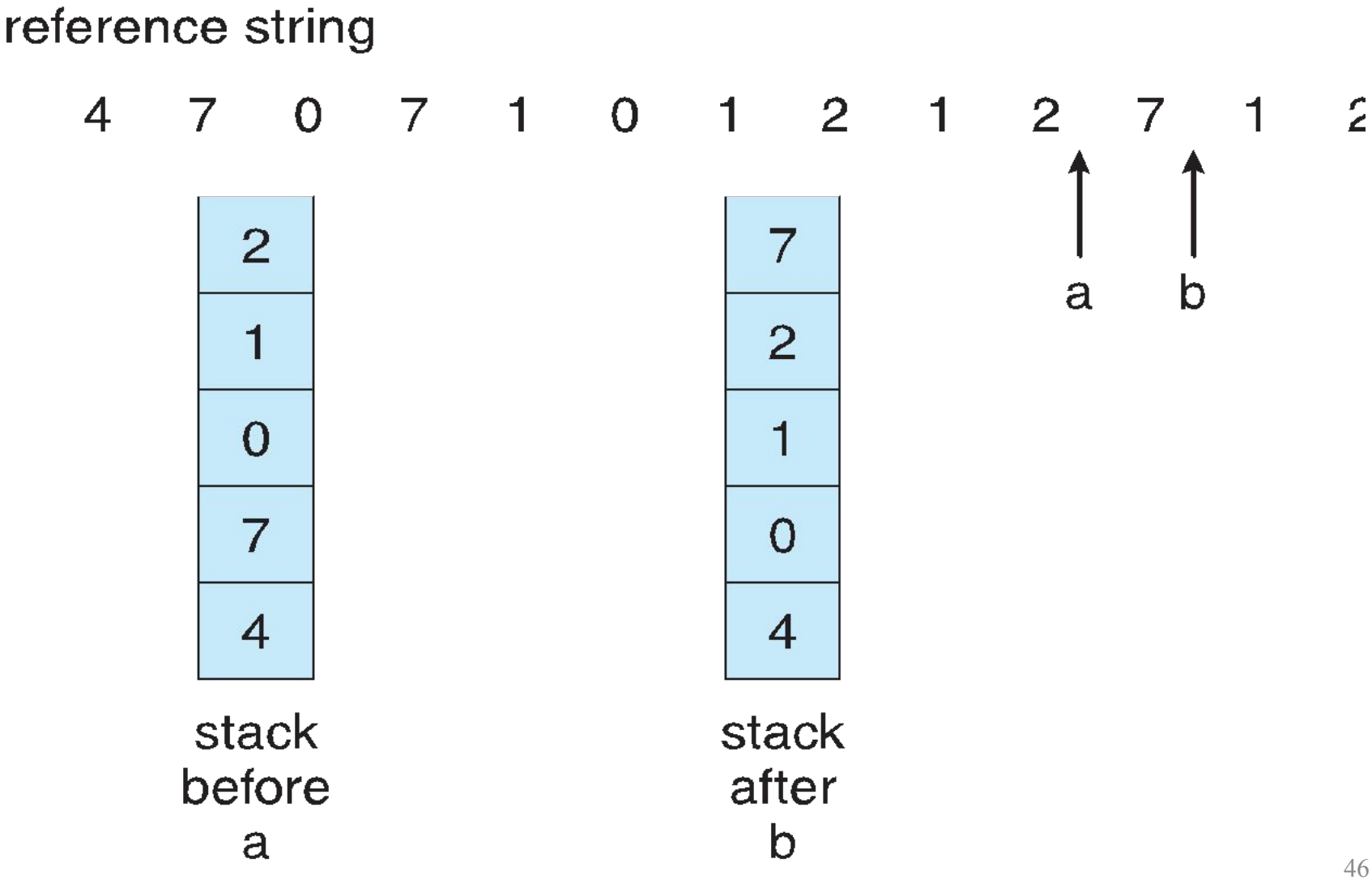
LRU IMPLEMENTATION

Stack implementation

- Keep a stack of page numbers in a double link form
- If Page referenced
  - move it to the top
  - requires 6 pointers to be changed
- But each update more expensive
- No search for replacement

- LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly

Stack to Record Most Recent Page References



LRU Approximation Algorithms

- The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).
- Reference bits are associated with each entry in the page table.
  - With each page associate a bit, initially as ‘0’
  - When page is referenced, set the bit to ‘1’
  - Now replace the page whose bit is ‘0’ if bit ‘1’ exists
  - Order of the page is not known but the used pages are known

LRU Approximation – Additional Reference Bits Algorithm

- Limitations in using single reference bit had been solved by providing the order of the pages
- Additional reference bits are ‘1’ byte in capacity
- Already existing reference bits are shifted to right in the additional reference bits table
- Page with the lowest number is the least recently used which will be chosen for replacement
- FIFO can be used at the time of tie

- Initially all the bits are set to ‘0’

	Reference Bit	Additional Bits							
Page 0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Page 1	0	0	0	0	0	0	0	0	0
Page 2	0	0	0	0	0	0	0	0	0
Page 3	0	0	0	0	0	0	0	0	0
Page 4	0	0	0	0	0	0	0	0	0

- Pages 1 and 3 are referred

	Reference Bit	Additional Bits							
Page 0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Page 1	1	0	0	0	0	0	0	0	0
Page 2	0	0	0	0	0	0	0	0	0
Page 3	1	0	0	0	0	0	0	0	0
Page 4	0	0	0	0	0	0	0	0	0

- The referred bit is copied to the MSB of the additional bits and reset the reference bit to ‘0’

	Reference Bit	Additional Bits							
Page 0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Page 1	0	1	0	0	0	0	0	0	0
Page 2	0	0	0	0	0	0	0	0	0
Page 3	0	1	0	0	0	0	0	0	0
Page 4	0	0	0	0	0	0	0	0	0

- Now pages 1 and 2 are referred. Then the bits which are present already in the MSB of the Additional bits table is shifted to right one place

	Reference Bit	Additional Bits							
Page 0		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Page 1	1	0	1	0	0	0	0	0	0
Page 2	1	0	0	0	0	0	0	0	0
Page 3	0	0	1	0	0	0	0	0	0
Page 4	0	0	0	0	0	0	0	0	0

- Again the process continues by copying the referred bits to the MSB of Additional bits table

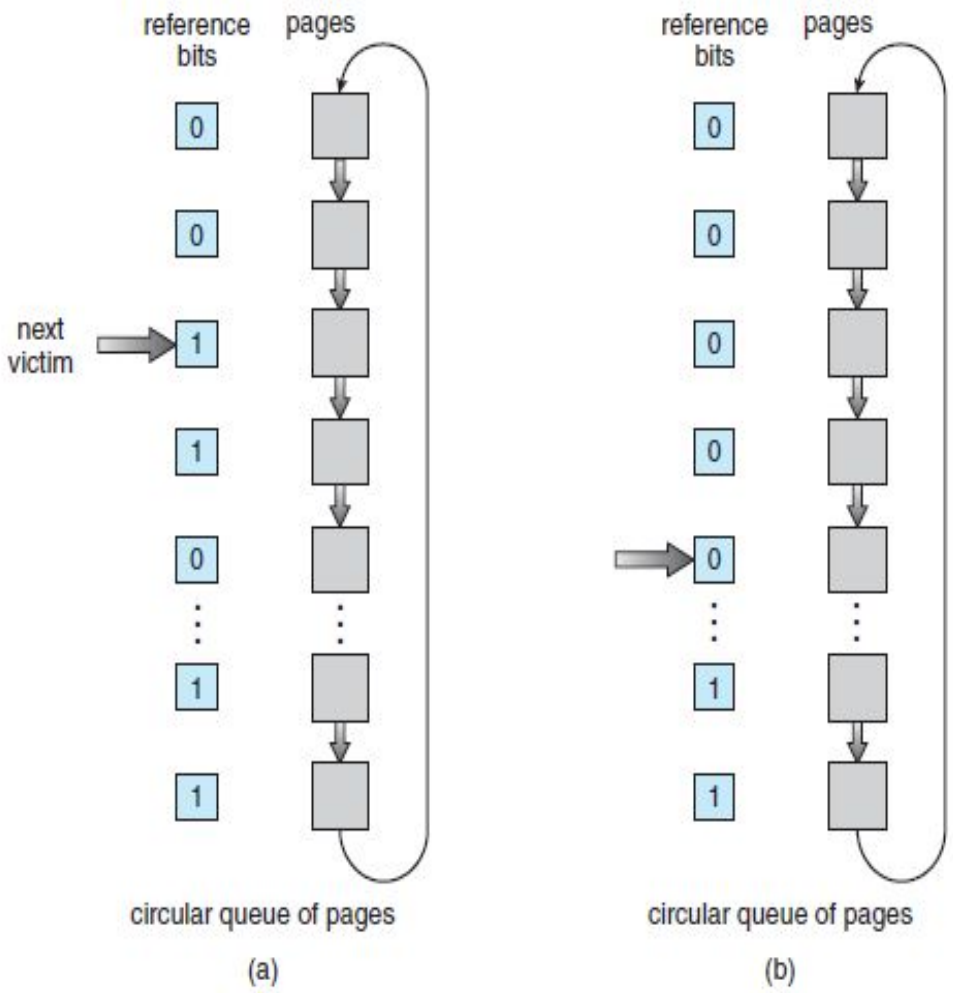
	Reference Bit	Additional Bits							
Page 0		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Page 1	0	1	1	0	0	0	0	0	0
Page 2	0	1	0	0	0	0	0	0	0
Page 3	0	0	1	0	0	0	0	0	0
Page 4	0	0	0	0	0	0	0	0	0

LRU Approximation – Second Chance/Clock Algorithm

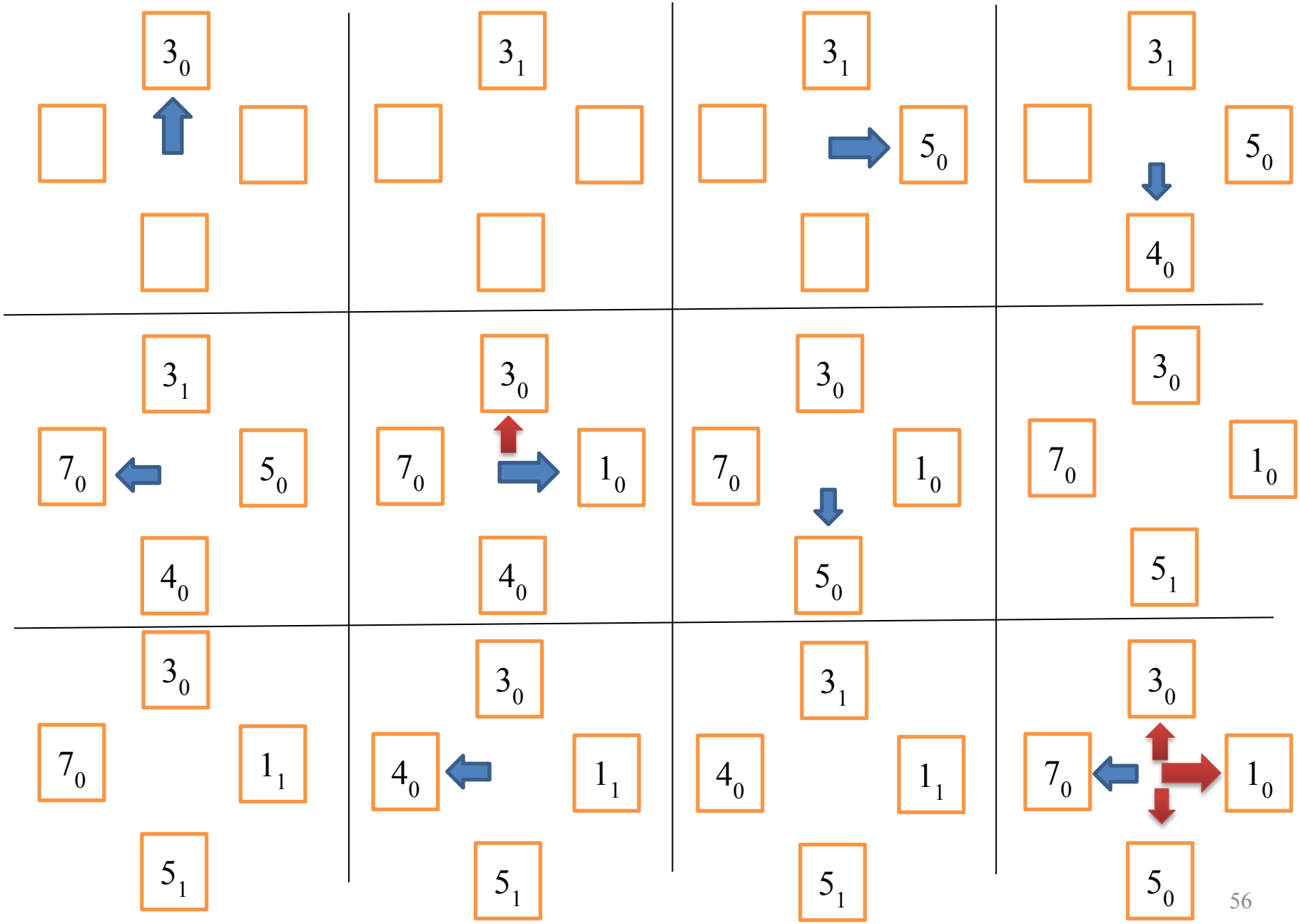
- By default FIFO is used
- Page is replaced based on reference bit
- If the reference bit is ‘0’ then the page is considered as not recently used and taken for replacement
- If the reference bit is ‘1’ then the page is considered as recently used and it is given a second chance to stay in the memory
- Now its reference bit is reset to ‘0’

- Fig (a) denotes next victim as page containing reference bits as 1, Fig (b) denotes that second chance is given to the referred pages and points the next victim as page containing reference bit as ‘0’

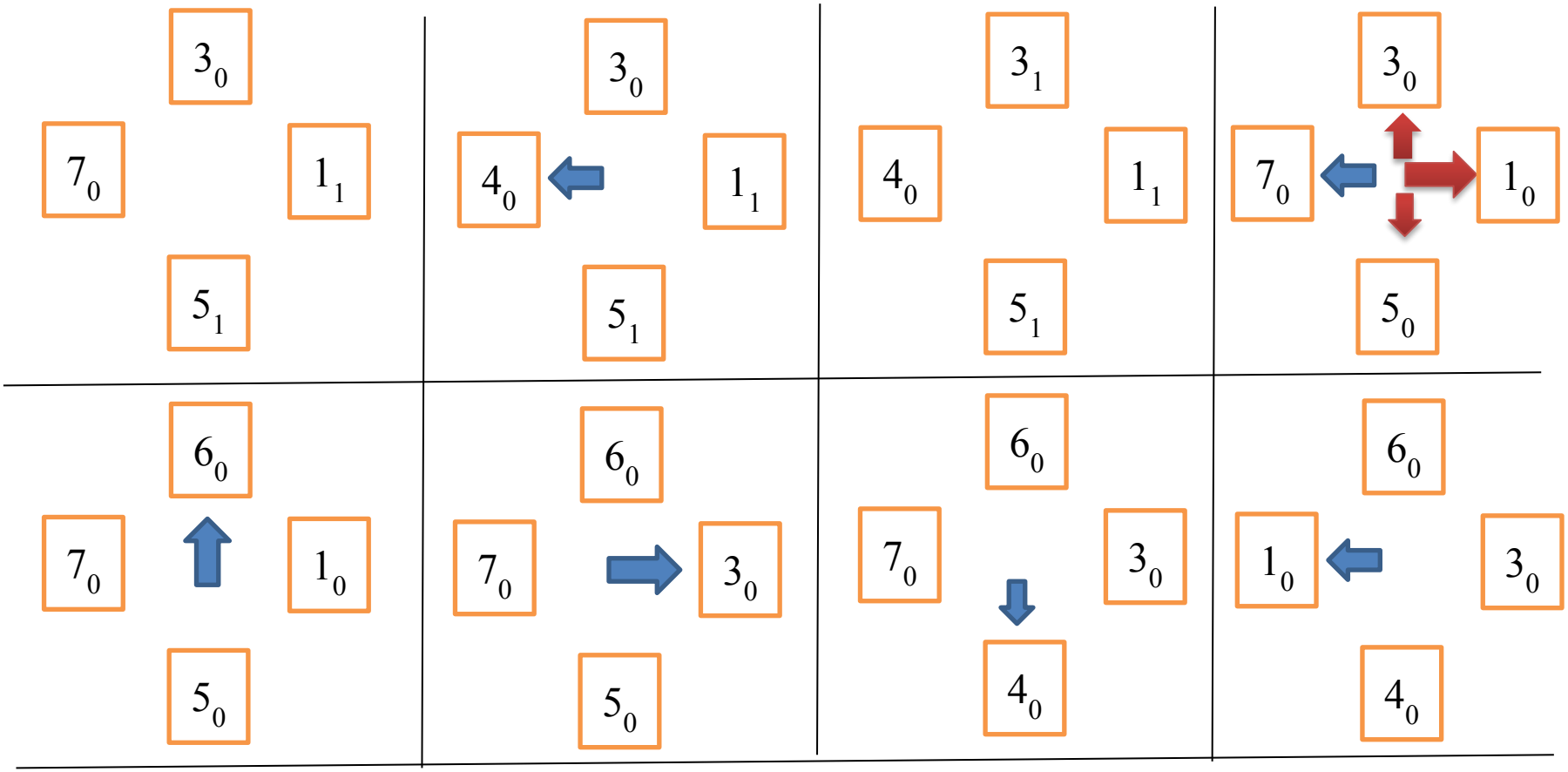
- At the mean time the reference bit is reset to ‘0’ for the page who got the second chance



- reference strings - 3 3 5 4 7 1 5 5 1 4 3 7 6 3 4 1



- reference strings - 3 3 5 4 7 1 5 5 1 4 3 7 6 3 4 1



- Number of page hits – 4
- Number of page faults – 12

LRU Approximation – Enhanced Second Chance Algorithm

- Use both the reference bit and modify bit for the replacement
- (0,0) – neither recently used or modified – best to replace
- (0,1) – not recently used but modified – page needs to be written before replacement
- (1,0) – recently used but clean – probably will be used again soon
- (1,1) – recently used and modified – may be used again, has to be written to the disk
- Out of these four classes, the page which falls under (0,0) will be replaced first, then (0,1) will be chosen for replacement, next (1,0) and finally (1,1) will be chosen for replacement

Understanding the Pros and cons of the page replacement techniques

FIFO Page Replacement Algorithm

Advantages

- Easy to understand and execute

Disadvantages

- It is not very effective
- System needs to keep track of each frame
- Page fault increases when increasing the page frames, undergoes Belady's anomaly

Understanding the Pros and cons of the page replacement techniques

Optimal Page Replacement Algorithm

Advantages

- Lowest page fault rate
- Never suffers from Belady's anomaly
- Twice as good as FIFO

Disadvantages

- Difficult to implement
- It needs forecast i.e. Future knowledge

Understanding the Pros and cons of the page replacement techniques

LRU Page Replacement Algorithms

Advantages

- It is amenable to full statistical analysis
- Never suffers from Belady's anomaly
- Easy to identify the faulty page that is not needed to long time

Disadvantages

- It has more complexity
- Need additional Data Structure and high hardware support

COUNTING ALGORITHMS

- It keeps count of number of references (frequencies) made to each page in a reference string
- Whenever a page comes in its frequency increases
- Whenever a page leaves the memory its frequency is reset
- **Least frequently used page replacement**
  - Page with least number of frequency is replaced first
  - Assumption – Heavily used page will be referred again
  - When there is a tie in frequencies, then use FIFO

COUNTING ALGORITHMS

- **Most Frequently used Page replacement algorithm**
- A page which has maximum number of frequencies in the counter will be replaced first
- Assumption – The page just brought in the memory is yet to use again
- If there is a tie in frequencies use FIFO for replacement

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

0	2	0	1	0	3	2	1	2	3

Counter value of each page	Page 0 : 0	Page 1 : 0	Page 2 : 0	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

0	2	0	1	0	3	2	1	2	3
0									

F									

Counter value of each page	Page 0 : 1	Page 1 : 0	Page 2 : 0	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F								

Counter value of each page	Page 0 : 1	Page 1 : 0	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H							

Counter value of each page	Page 0 : 2	Page 1 : 0	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H	F						

Counter value of each page	Page 0 : 2	Page 1 : 1	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H	F	H					

Counter value of each page	Page 0 : 3	Page 1 : 1	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H	F	H	F				

Counter value of each page	Page 0 : 3	Page 1 : 1	Page 2 : 0	Page 3 : 1

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H	F	H	F	F			

Counter value of each page	Page 0 : 3	Page 1 : 0	Page 2 : 1	Page 3 : 1

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H	F	H	F	F	F		

Counter value of each page	Page 0 : 3	Page 1 : 1	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

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

F	F	H	F	H	F	F	F	H	

Counter value of each page	Page 0 : 3	Page 1 : 1	Page 2 : 2	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform LFU

0	2	0	1	0	3	2	1	2	3
0	0	0	0	0	0	0	0	0	0
	2	2	2	2	3	3	1	1	3
			1	1	1	2	2	2	2
F	**F**	**H**	**F**	**H**	**F**	**F**	**F**	**H**	**F**

Counter value of each page	Page 0 : 3	Page 1 : 0	Page 2 : 2	Page 3 : 1

- Number of page hits – 3
- Number of page faults – 7

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

0	2	0	1	0	3	2	1	2	3

Counter value of each page	Page 0 : 0	Page 1 : 0	Page 2 : 0	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

0	2	0	1	0	3	2	1	2	3
0									
F									

Counter value of each page	Page 0 : 1	Page 1 : 0	Page 2 : 0	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F								

Counter value of each page	Page 0 : 1	Page 1 : 0	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H							

Counter value of each page	Page 0 : 2	Page 1 : 0	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F						

Counter value of each page	Page 0 : 2	Page 1 : 1	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F	H					

Counter value of each page	Page 0 : 3	Page 1 : 1	Page 2 : 1	Page 3 : 0

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F	H	F				

Counter value of each page	Page 0 : 0	Page 1 : 1	Page 2 : 1	Page 3 : 1

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F	H	F	H			

Counter value of each page	Page 0 : 0	Page 1 : 1	Page 2 : 2	Page 3 : 1

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F	H	F	H	H		

Counter value of each page	Page 0 : 0	Page 1 : 2	Page 2 : 2	Page 3 : 1

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F	H	F	H	H	H	

Counter value of each page	Page 0 : 0	Page 1 : 2	Page 2 : 3	Page 3 : 1

Reference String – 0, 2, 0, 1, 0, 3, 2, 1, 2, 3
Total Frames allocated – 3
Perform MFU

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

F	F	H	F	H	F	H	H	H	H

Counter value of each page	Page 0 : 0	Page 1 : 2	Page 2 : 3	Page 3 : 2

- Number of page hits – 6
- Number of page faults – 4

Page-Buffering Algorithms

- Used in addition to page replacement algorithms
- System keeps a pool of free frames
- When a page fault occurs
  - Victim frame is chosen
    - but requires some time to choose the victim frame
  - At that time desired page is read into a free frame which is maintained by the system
    - By doing this the desired page can start doing its work without waiting for the swapping process of the victim frame in the main memory
  - Victim frame is written out later
  - Finally Victim frame is added to the pool of free frames

Page-Buffering Algorithms

- Another idea is to maintain a list of modified pages
- Whenever paging device is idle, a modified page is selected and written to the disk
  - When there is read/write operation at that time paging device does this job
- Modify bit is then reset
- Increases the probability that a page is clean when chosen for replacement
  - Used to identify the clean pages very quickly, because after performing this operation maximum modified pages will be already written to the disk
  - Makes the replacement very easy

Page-Buffering Algorithms

- Yet Another idea is to keep a pool of free frames
- Remember which page was in each frame
- If the same page is needed again, it is taken from the pool of free frames and used
  - But the page should not be modified for reusing
  - No I/O operation is required
- If the required page is not in the pool of free frames, another free frame is selected
- The new page is read into free frame

Allocation of Frames

- How do we allocate the fixed amount of free memory among the various process?
  - Example: If we have 93 free frames and 2 processes, How many frames does each process get?
- Consider a single-user system with 128 KB of memory composed of pages 1 KB in size.
  - This system has 128 frames.
    - The operating system may take 35 KB of memory and
    - Leaving remaining 93 frames for the user process.
- Other possible strategy: the user process is allocated with any free frame.

Demand paging Vs. Pure Demand paging

Demand paging

- In demand paging, a page is not loaded into main memory until it is needed.
- In Demand paging follows that pages should only be brought into memory if the executing process demands them.

Pure Demand paging

- In pure demand paging, even a single page is not loaded into memory initially. Hence pure demand paging causes a page fault. Page fault, the situation in which the page is not available whenever a processor needs to execute it.
- while in pure demand paging swapping, where all memory for a process is swapped from secondary storage to main memory during the process startup.

Allocation of Frames – Pure Demand Paging

Under Pure demand paging:

- All 93 frames - initially kept on free-frame list.
- When a user process started execution, it would generate a sequence of page faults.
- There is not page fault up to 93 frames.
- When the free-frame list was exhausted then a page-replacement algorithm would be called to select one of the 93 in-memory pages to be replaced with the 94th, and so on.
- When the process terminated, the 93 frames would once again be placed on the free-frame list.

Allocation of Frames

- Each process needs minimum number of frames
  - Allocating minimum number of frames impacts on the performance.
  - If the number of frames allocated to each process decreases, the page-fault rate increases, which slow downs the execution of process
  - Also when a page fault occurs before completion of an executing then the instruction must be restarted.
- It is necessary to hold all the different pages that any single instruction can reference.
- Example: IBM 370 – 6 pages to handle storage to storage (SS) MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from* one storage area
  - 2 pages to handle *to* another storage area

Allocation schemes

- Minimum number of frames per process is defined by the architecture.
- Maximum number is defined by the amount of available physical memory in the system
- In between is still left with significant choice of frame allocation.
- Two major allocation schemes
  - Fixed allocation
    - Equal Allocation
    - Proportional Allocation
  - Priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted

- s_i = size of process p_i
- $S = \sum s_i$
- m = total number of frames
- a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Fixed Allocation – Contd.

- In equal and proportional allocation
  - If the multiprogramming level is increased, each process will lose some frames for the new process.
  - if the multiprogramming level decreases, the frames that were allocated from departed process are spread to the remaining processes.

Priority Allocation

- Suppose if we want to give the high-priority process with more memory to speed up its execution than the low-priority processes.
  - Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Frame Allocation - Another factor

- Multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories namely Global and Local.

Global replacement

- Process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

Local replacement

- Each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Global vs. Local Allocation – Contd.

- If, we allow high-priority processes to select frames from low-priority processes for replacement
  - Global replacement, approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process
  - Local replacement, the number of frames allocated to a process does not change.

Non-Uniform Memory Access (NUMA)

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Root cause of the Thrashing

- Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault.
- The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming.
- It can be eliminated by reducing the level of multiprogramming.

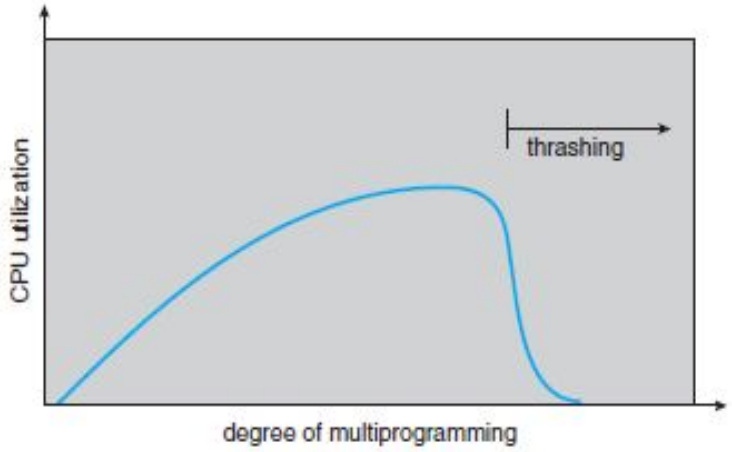
Thrashing

- A process is busy with swapping pages in and out. This high paging activity is called Thrashing.
- A process is in thrashing if it is spending more time on paging rather than executing.

Cause of Thrashing

If a process does not have "enough" pages, the page-fault rate is very high. This leads to:

- low CPU utilization.
- operating system thinks that it needs to increase the degree of multiprogramming.
- another process added to the system.

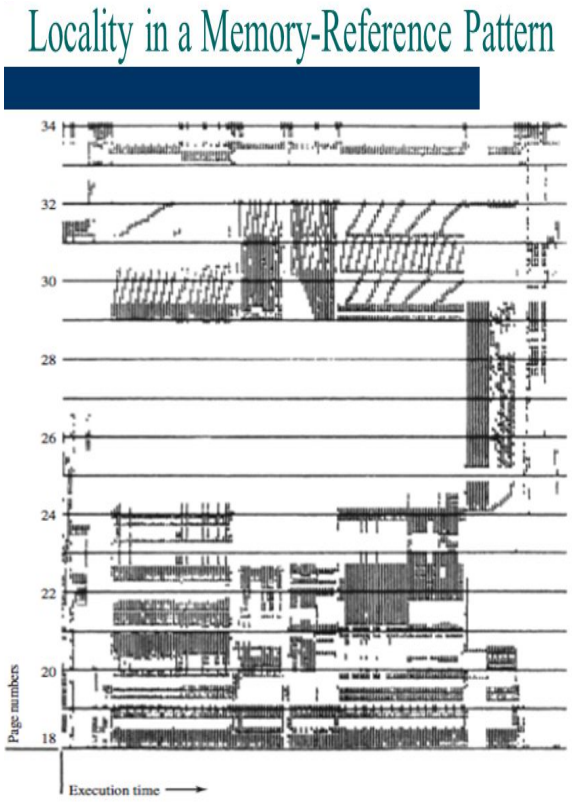


Demand Paging and Thrashing

- Why does demand paging work?
  - To prevent thrashing, we must provide a process with as many frames as it needs.
- But how do we know how many frames it “needs”?
- Locality model - working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution

Demand Paging and Thrashing - Contd

- Process migrates from one locality to another (Figure)
- Localities may overlap
- Example:
  - Function is called, that defines a new locality.
  - In this, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use.



Demand Paging and Thrashing – Cont.

- Why does thrashing occur?
  - Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement
  - Local replacement algorithms - if one process starts thrashing it cannot steal frames from another process and cause thrashing latter.
    - However if processes are thrashing, they will be in the queue for the paging device most of the time.
    - The average service time for a page fault will increase, due to the longer queue for the paging device.
    - Thus the effective access time will increase even for a process that is not thrashing.

Models to Avoid Thrashing

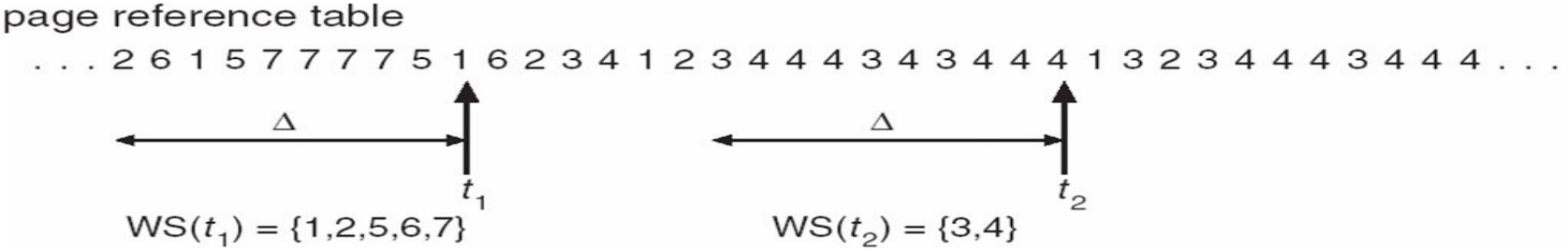
- There are 2 Models to Avoid Thrashing
  - Working – set Model
    - Works based on Locality of Reference
  - Page Fault Frequency Model
    - Based on Upper and Lower Page Fault Rate

Working Set Model

- It is based on the assumption of locality.
- Δ ≡ working-set window ≡ a fixed number of page references
  Example: 10,000 instruction
- The idea is to examine the most recent Δ page references.
- WSS_i (working set of Process Pi) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if Δ = ∞ ⇒ will encompass entire program.
- $D = \Sigma\ WSS_i$ ≡ total demand frames
- if D > m ⇒ Thrashing
  - Policy if D > m, then suspend one of the processes.

Working Set Model – Contd.

- The sequence of memory references shown in Figure,
- if Δ = 10 memory references,
  - Then the working set at time
  - t_1 is {1, 2, 5, 6, 7}
  - t_2, the working set has changed to {3, 4}.



The **working set model** states that a process can be in RAM if and only if all of the pages that it is currently using (often approximated by the most recently used pages) can be in RAM.

Keeping Track of the Working Set

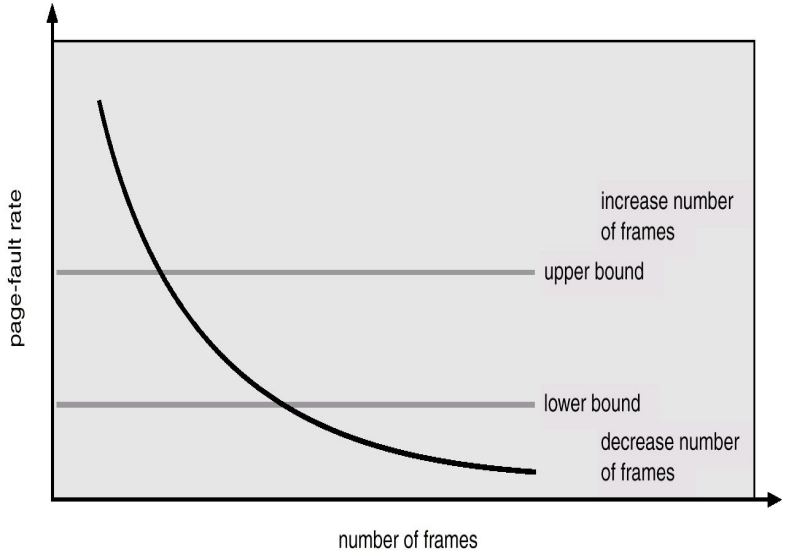
- The working set prevents thrashing while keeping the degree of multiprogramming as high as possible, in order to optimize the CPU utilization.
- Difficulty: keeping track of the working set

  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 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.

Page-Fault Frequency

- More direct approach than Working Set model.
  - Where working set seems a clumsy way to control thrashing.
- If page fault rate is too high, then the process need more frames.
- Whereas page fault is too low, then the process may have too many frames.

- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Page-Fault Frequency – Contd.

- As with the working-set strategy, we may have to swap out a process.
- If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store.
- The freed frames are then distributed to processes with high page-fault rates