Chapter 9: Virtual Memory

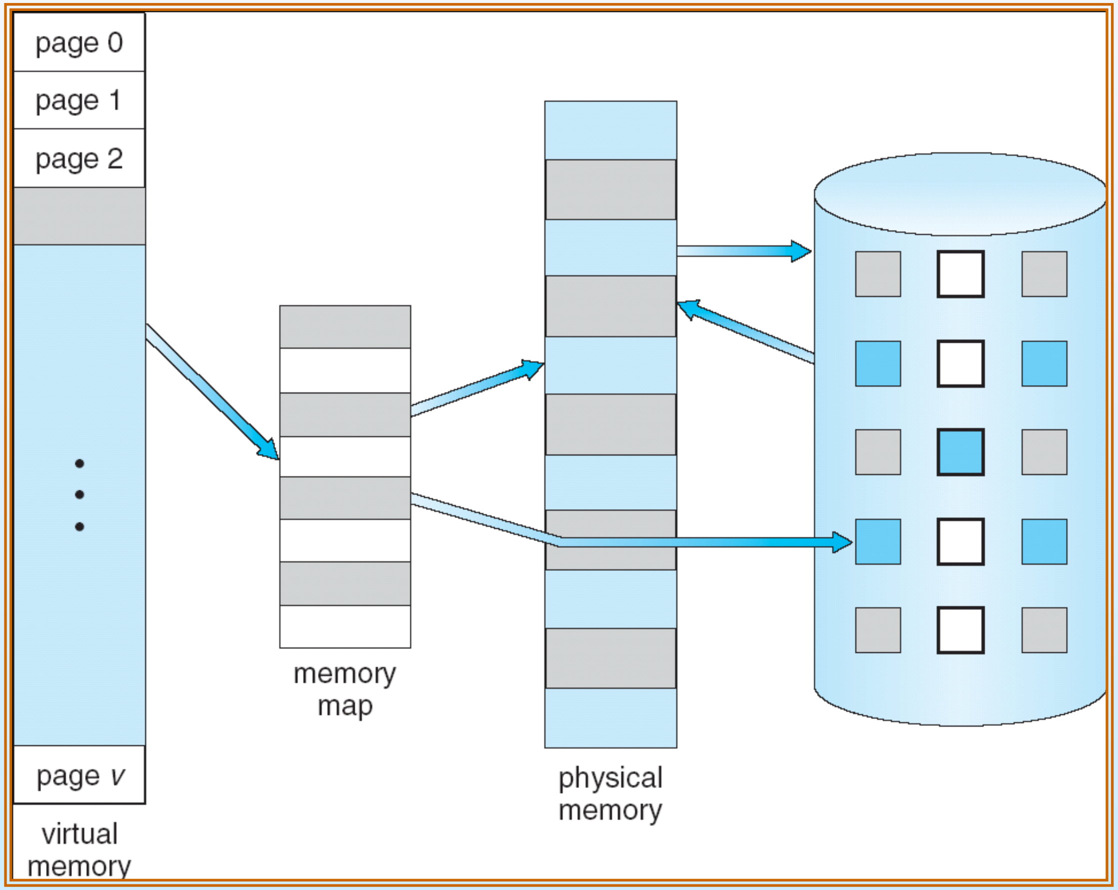
Narzu Tarannum

CSE, BU

Intro

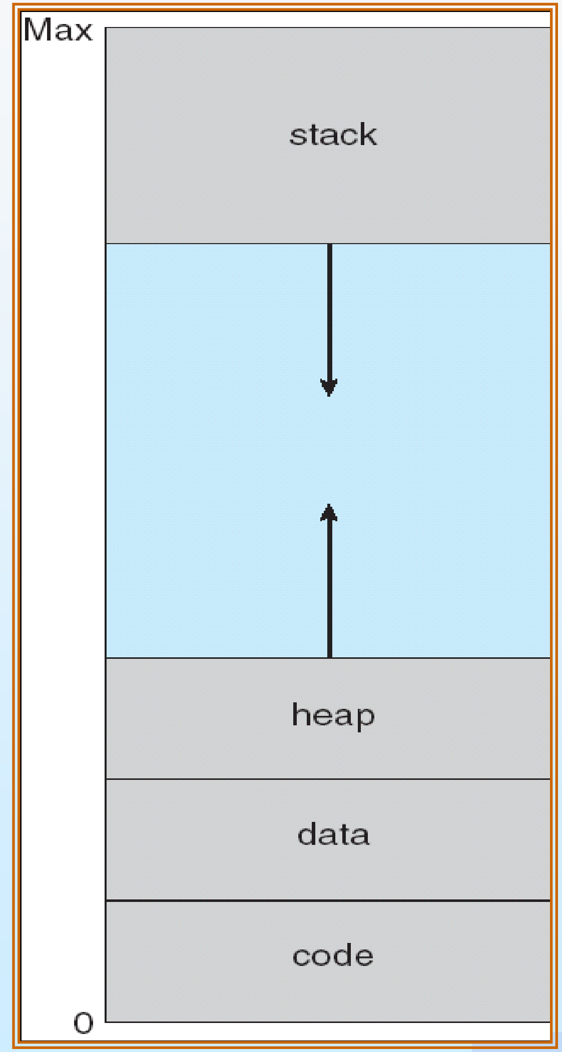
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
- Virtual address space – logical view of how process is stored in memory

Virtual Memory That is Larger Than Physical Memory



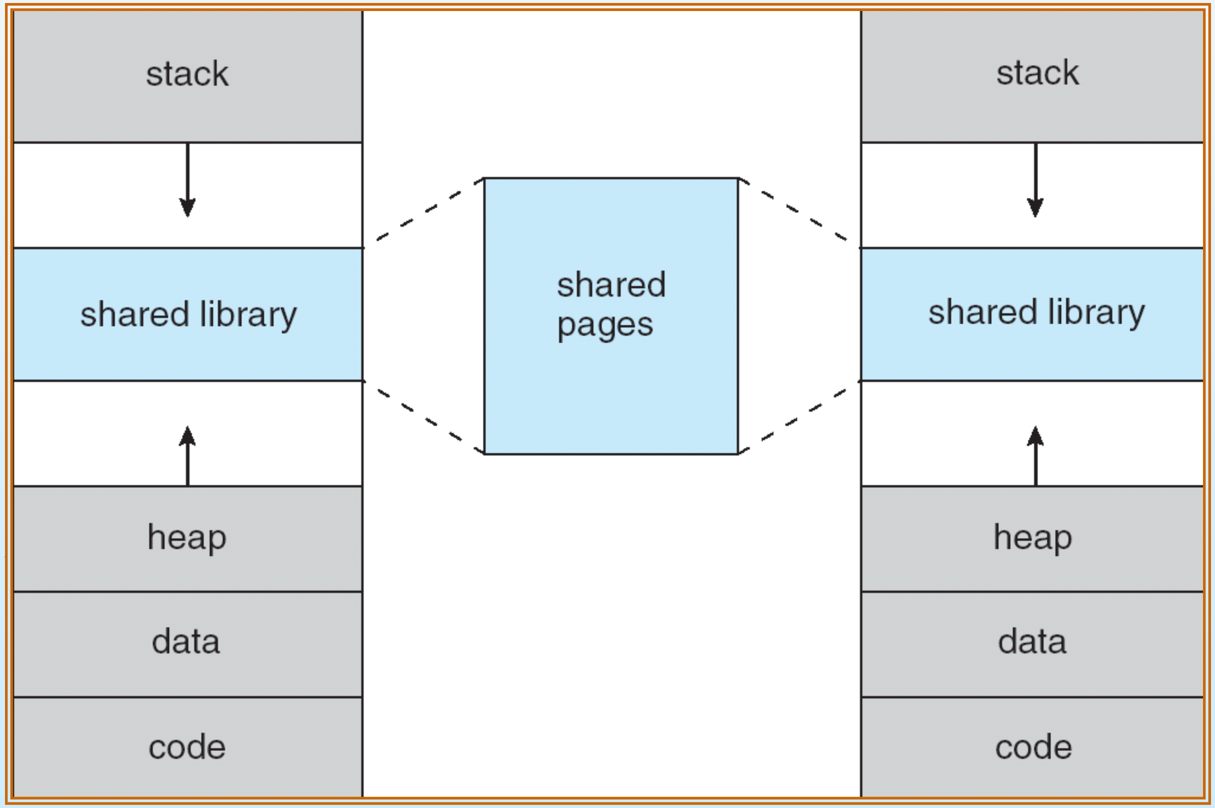
Virtual-address Space

- After compilation, process / executable code presented by its virtual address space
- It is a contiguous address space from address 0 to Max



Shared Library Using Virtual Memory

- Virtual memory also allows files and memory to be shared by two or more processes.



Demand Paging

- Bring a page into memory only when it is needed
- Demand paging follows that pages should only be brought into memory if the executing process demands them. This is often referred to as lazy evaluation or lazy swapper as only those pages demanded by the process are swapped from secondary storage to main memory.
- Contrast this to pure swapping/ pure demand paging, where all memory for a process is swapped from secondary storage to main memory during the process startup.
- Commonly, to achieve this process a page table implementation is used. The page table maps logical memory to physical memory.

Swapper that deals with pages is a **pager**

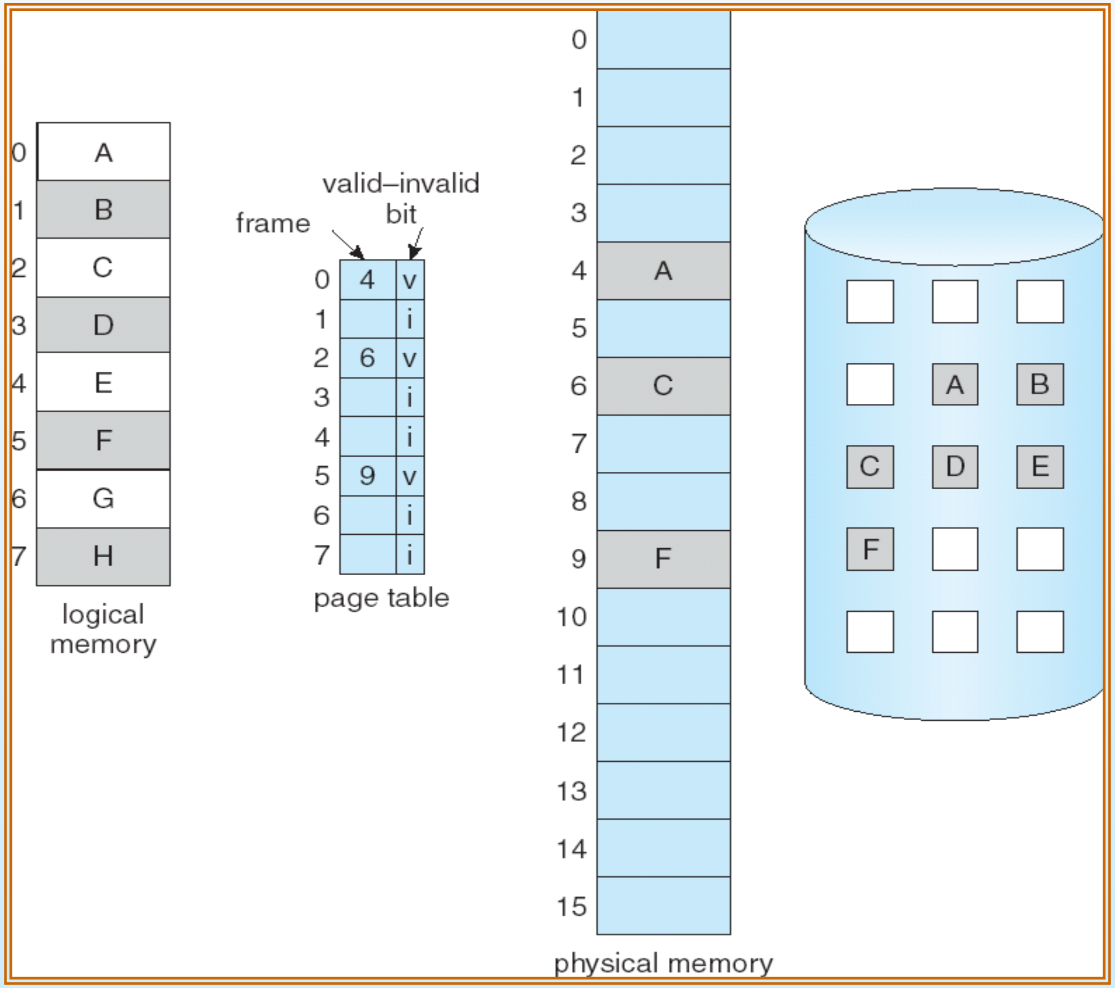
Demand Paging

- Advantages
  - Less memory needed
  - Faster response
  - More users/ Degree of multiprogramming increase.
  - Less I/O needed
  - Reduce memory requirement
  - Swap time is also reduce
- Disadvantages
  - Page fault interrupt
- **locality of reference**, also known as the principle of **locality**, is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern.
  - Results in reasonable performance from demand paging.
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory

Valid Invalid bit

- The page table uses a bitwise operator to mark if a page is valid or invalid. A valid page is one that currently resides in main memory. An invalid page is one that currently resides in secondary memory. When a process tries to access a page, the following steps are generally followed:
  - Attempt to access page.
  - If page is valid (in memory) then continue processing instruction as normal.
  - If page is invalid then a **page-fault trap / page-fault interrupt** occurs.
  - Page is needed $\Rightarrow$ reference to it
    - invalid reference $\Rightarrow$ abort
    - not-in-memory $\Rightarrow$ bring to memory
  - Restart the instruction that was interrupted by the operating system trap.
- With each page table entry a valid–invalid bit is associated
  (1 $\Rightarrow$ in-memory, 0 $\Rightarrow$ not-in-memory)

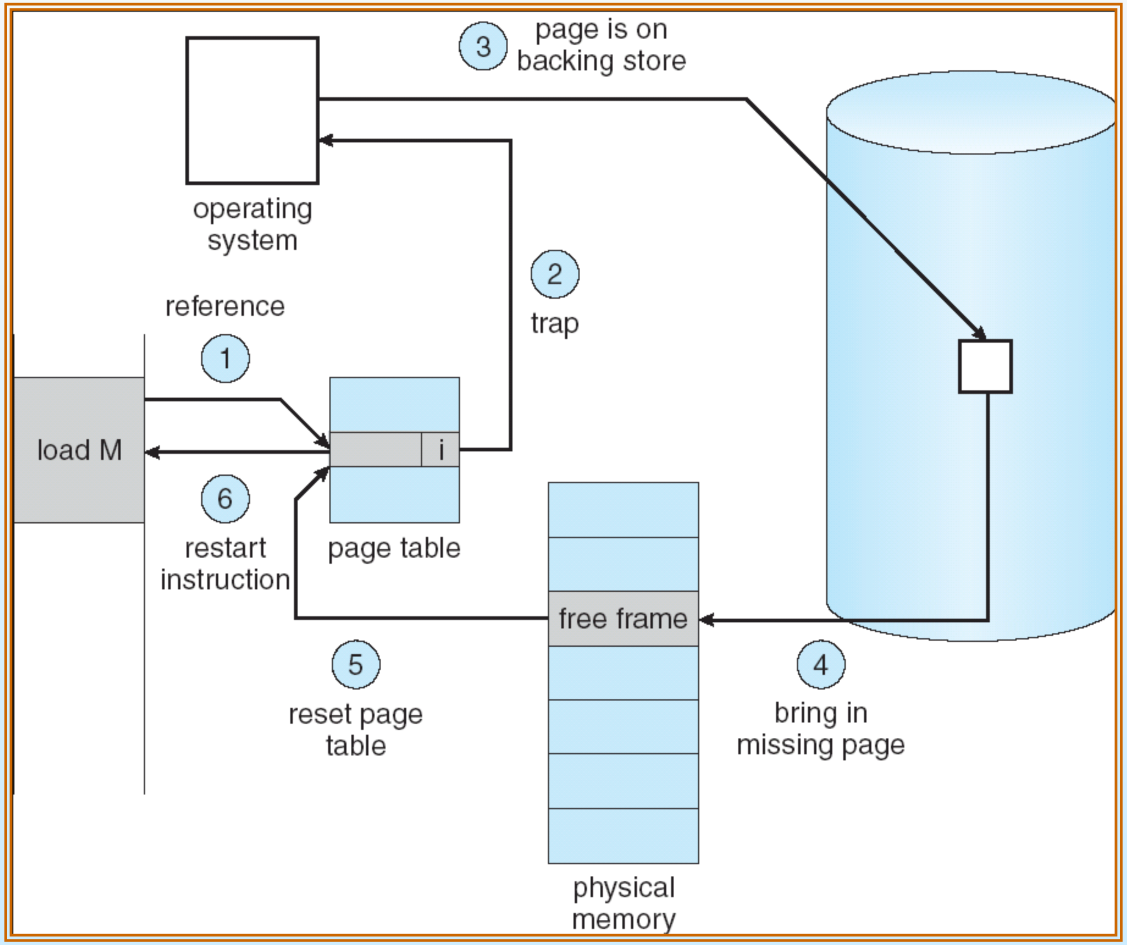
Page Table When Some Pages Are Not in Main Memory



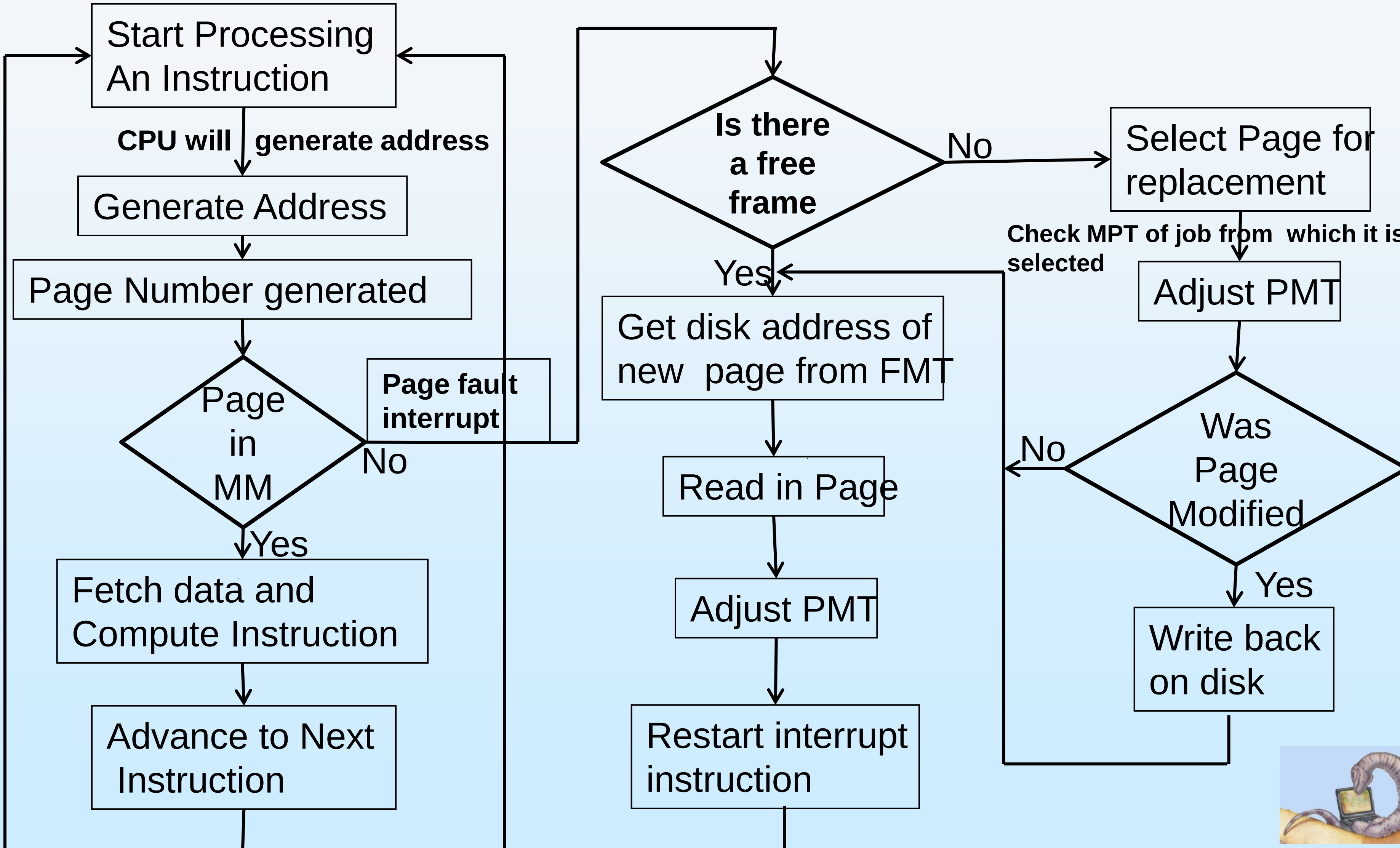
Page Fault

- If there is ever a reference to a page, first reference will trap to OS $\Rightarrow$ **page fault**
- OS looks at another table to decide:
  - Invalid reference $\Rightarrow$ abort.
  - Just not in memory.
- Find empty frame.
- Load page from disk into frame.
- Reset tables, validation bit = 1.
- Restart instruction that caused page fault

Steps in Handling a Page Fault



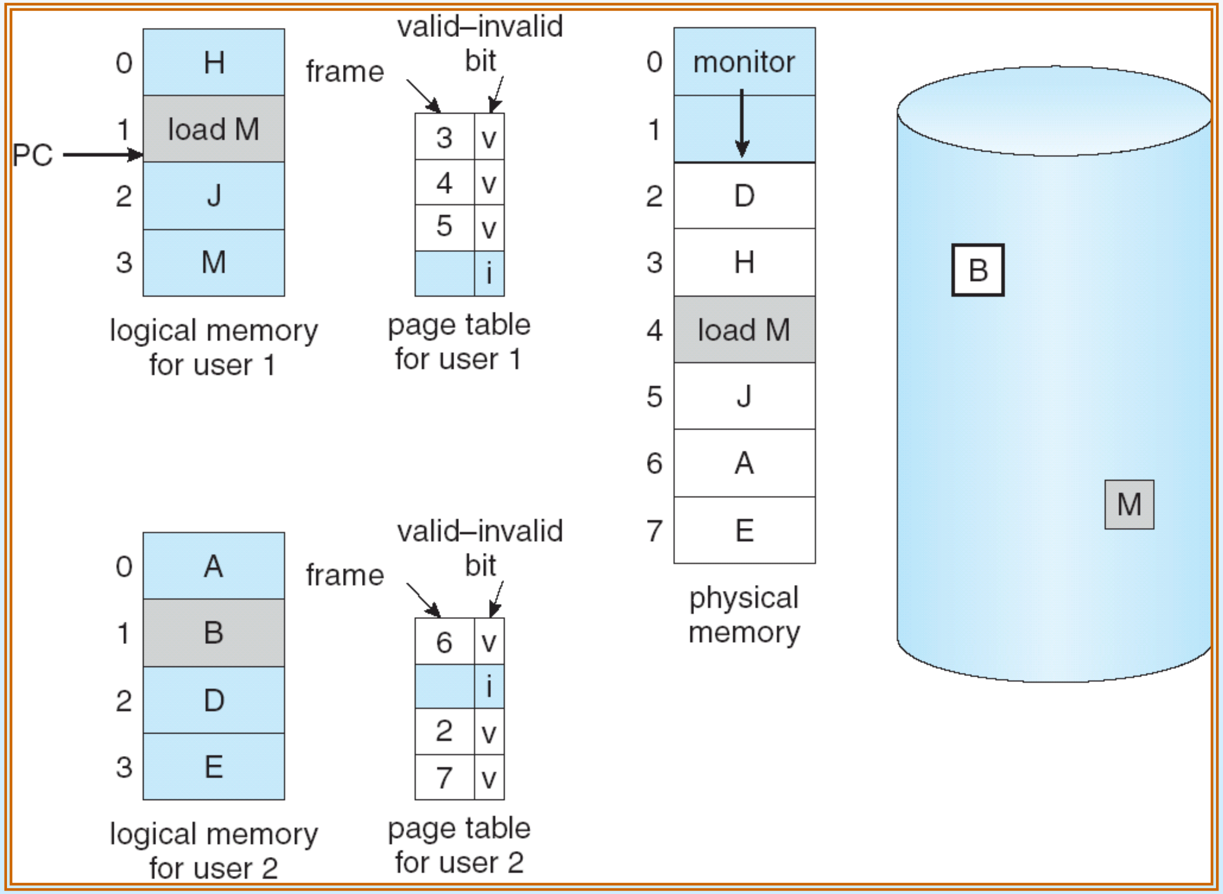
Flowchart of Demand Paging



What happens if there is no free frame?

- **Page replacement** – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Need For Page Replacement



Performance of Demand Paging

- Demand paging can significantly affect the performance of computer system
- Let's compute the effective access time (EAT) for a demand page memory:
  - If "*p*" be the probability of a page fault where $0 \le p \le 1$.
    - $p = 0$ no page faults
    - $p = 1$, every reference is a fault
  - We should expect *p* to be close to 0 or only a few page fault

Performance of Demand Paging

- Then,
  - EAT = (1 – p) x ma + p x page fault time
- Let,
  - ma = memory access time (200 nano-seconds)
  - Average page fault service time = 8 mili-seconds

  * 1ms=1000000ns
- EAT = (1 – p) x 200 + p (8 milli-seconds)

  = 200 – 200p + 8,000,000p (nano-seconds)

  = 200 + 7,999,800p
- We can see EAT is directly proportional to page fault rate
  - For Example if one access out of one thousand causes a page fault

  = 200 + 7,999,800 x 1/1000

  = 200 + 7999.8 = 8199.8 ns

  = 8199.8 x 1/1000 micro-second [1ms=1000ns]

  = 8.1998 micro-seconds ≈ 8.2 micro-seconds

Performance of Demand Paging

- EAT= 8.2 micro-seconds
- This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent

  $220 > 200 + 7999800p$ [were EAT= 200 + 7999800p]

  $220 - 200 > 7999800p$

  $20 > 7999800p$

  $p < 20 / 7999800$

  $p < .0000025$

- It means one page fault in every 400,000 memory accesses
- **So, we can conclude, it is important to keep the page fault rate very low in a demand paging system**

Basic Page Replacement

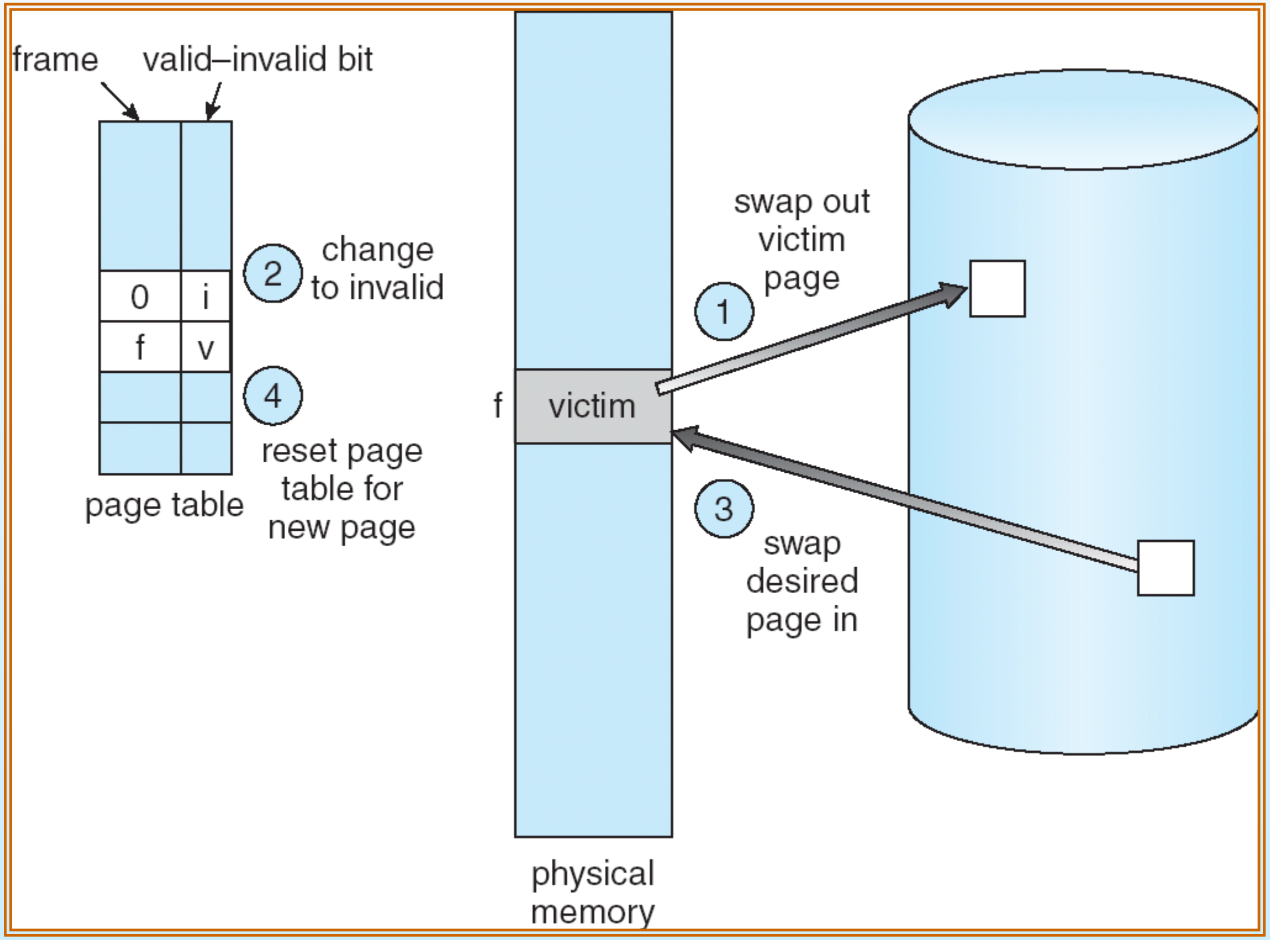
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process

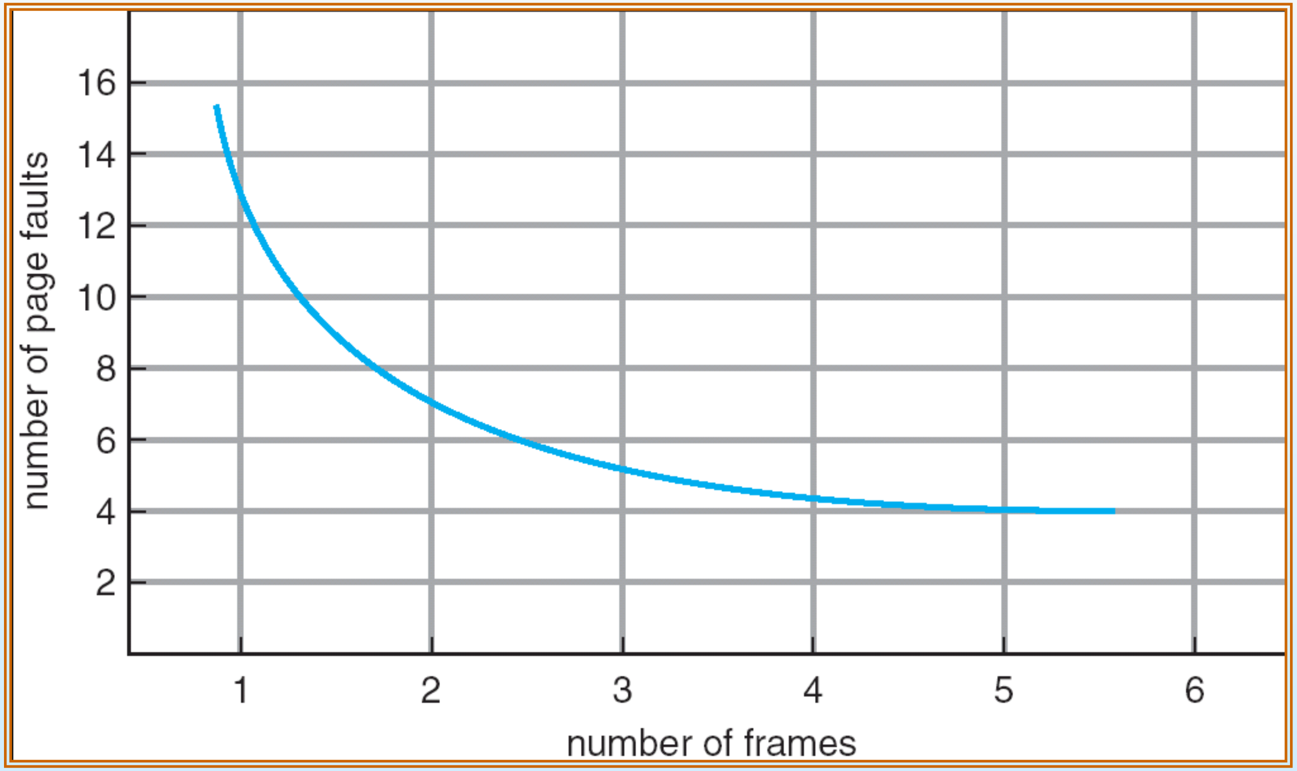
Page Replacement



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
- Solve two problems in demand paging implementation:
  - **Frame-allocation algorithm** – how many frames to allocate to each process
  - **Page-replacement algorithm** – select frames to be replaced

Graph of Page Faults Versus The Number of Frames



Page Replacement Algorithms

- 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
- In all our examples, the reference string is

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

First-In-First-Out (FIFO) Algorithm

- The page which brought first will be replace first
- Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- 3 frames (3 pages can be in memory at a time per process)

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

9 page faults

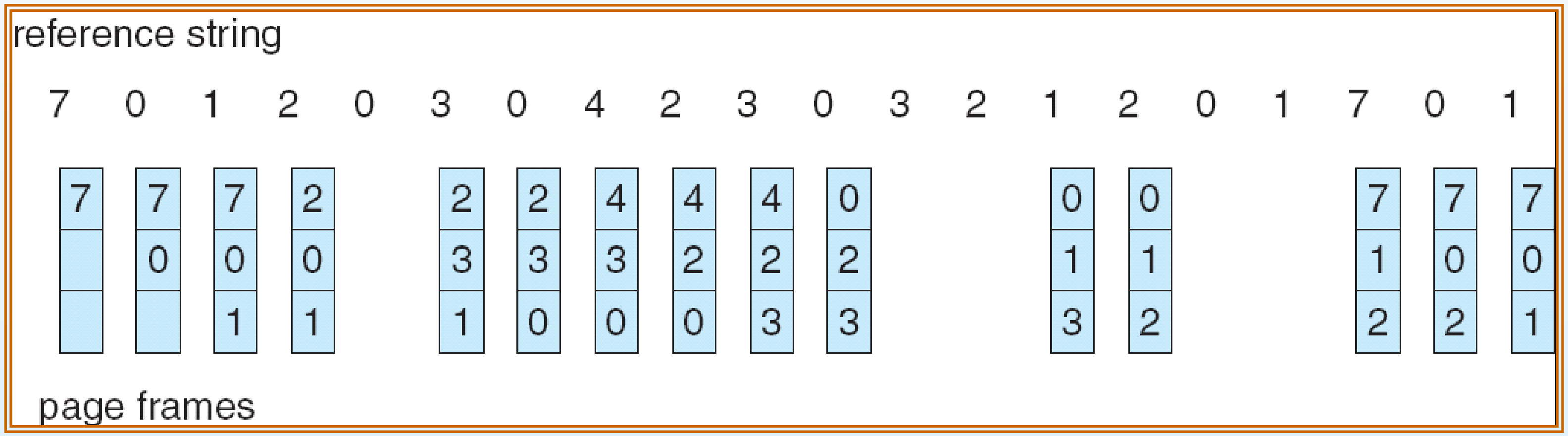
- 4 frames

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

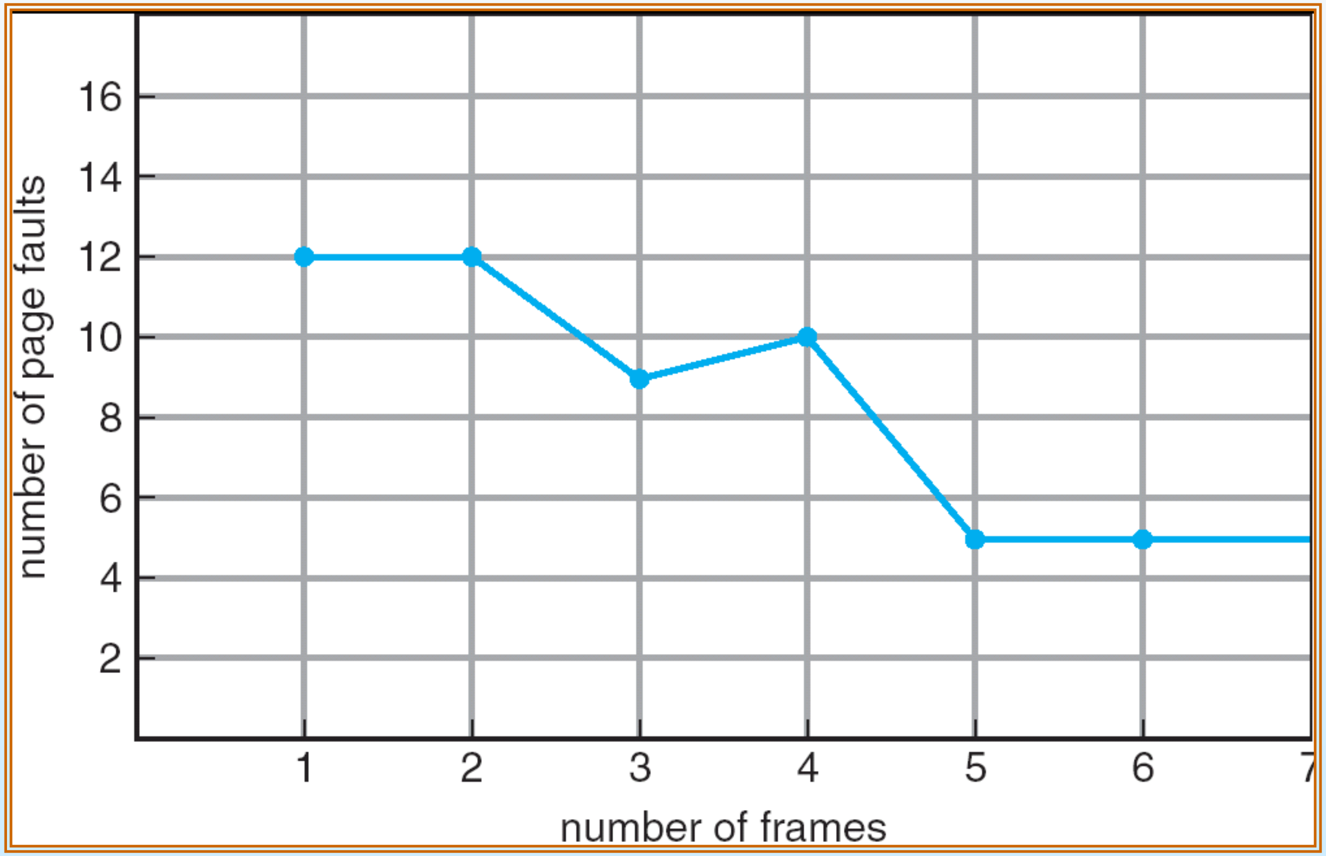
10 page faults

- FIFO Replacement – Belady’s Anomaly
  - more frames $\Rightarrow$ more page faults

FIFO Page Replacement



FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

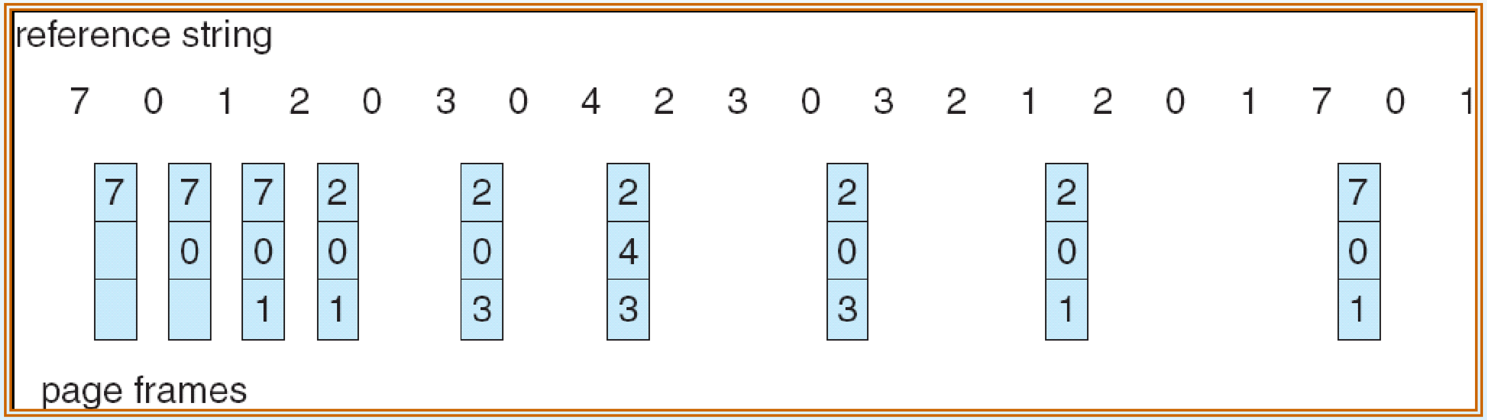
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

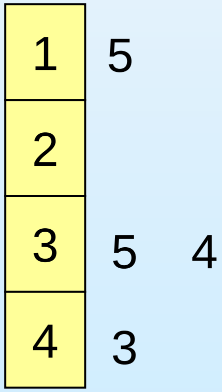
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



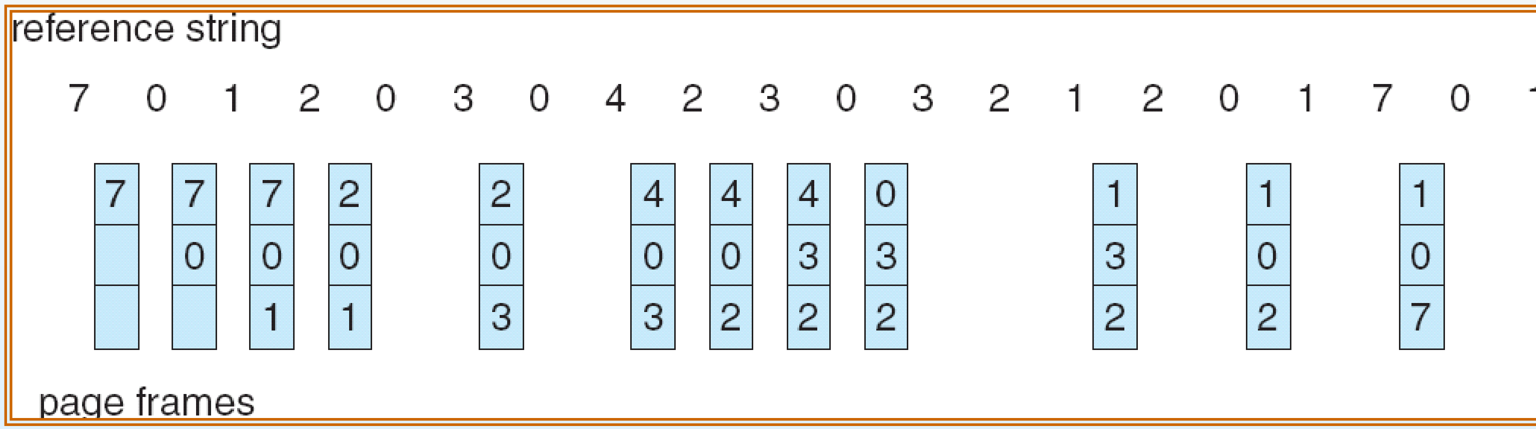
Least Recently Used (LRU) Algorithm

- LRU replaces page that has not been used for the longest time
- Use the recent past to predict the future
- Reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



8 page faults

LRU Page Replacement

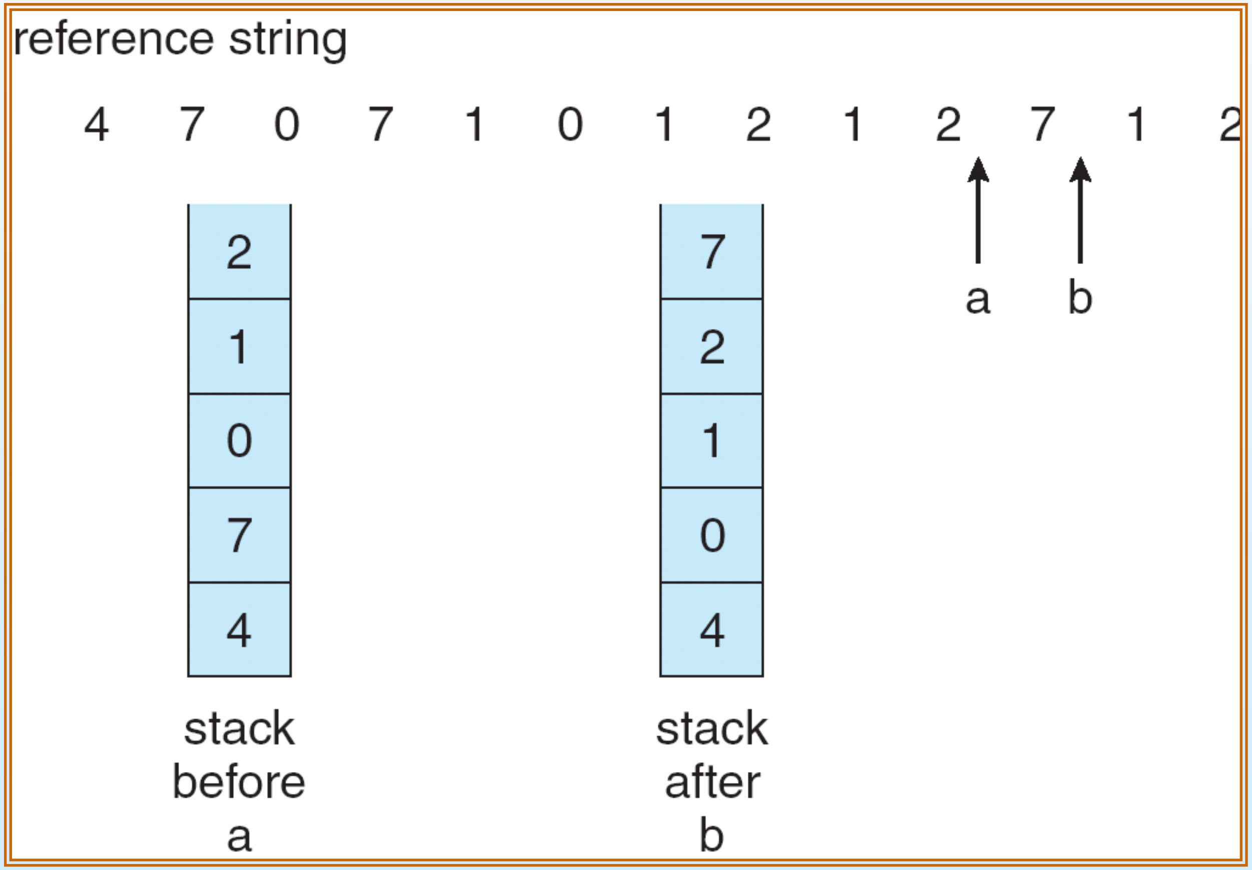


LRU Algorithm (Cont.)

- **Counter** implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be replaced, look at the counters to determine which has the oldest time-of-access

- **Stack** implementation – keep a stack of page numbers in a double link form:
  - Page referenced -> move it to the top of stack
    - bottom of stack will be the LRU page
    - requires 6 pointers to be changed
  - No search for replacement

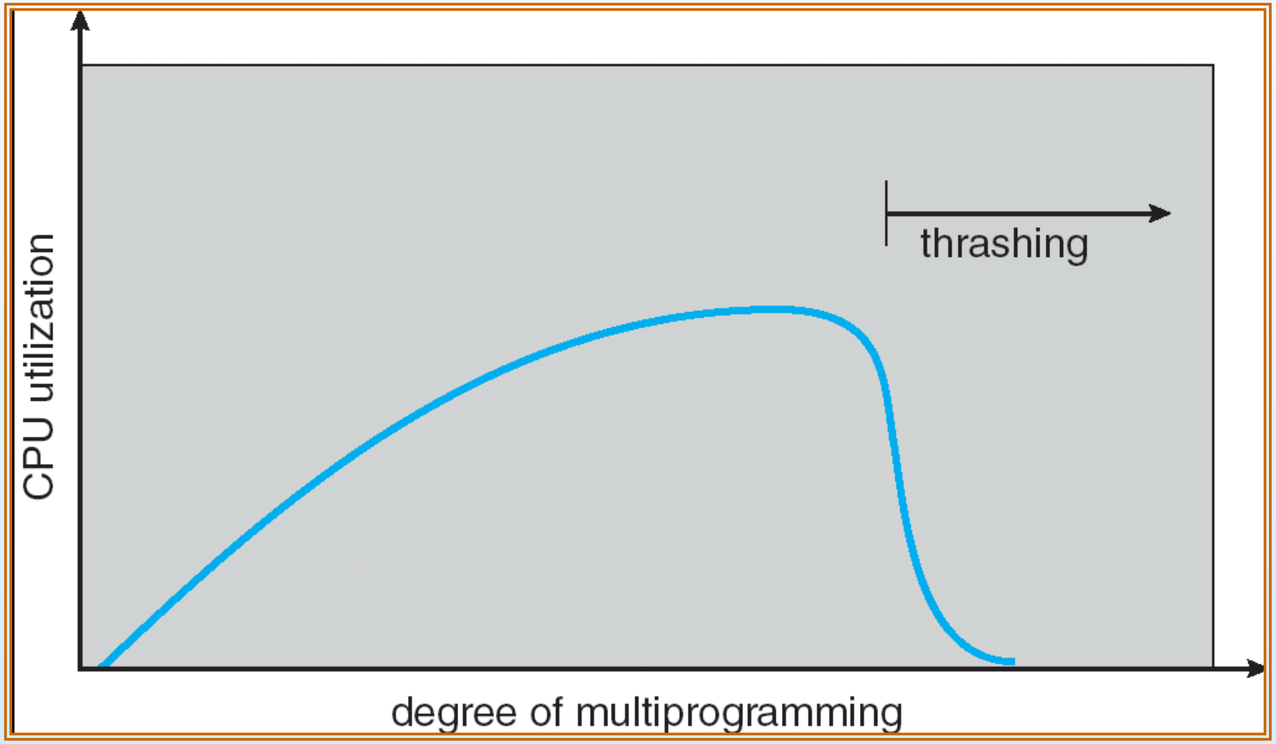
Use Of A Stack to Record The Most Recent Page References



Thrashing

- It is a situation, if the system spends more time in paging/swapping instead of their execution.
- If a process does not have "enough" frames, 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
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

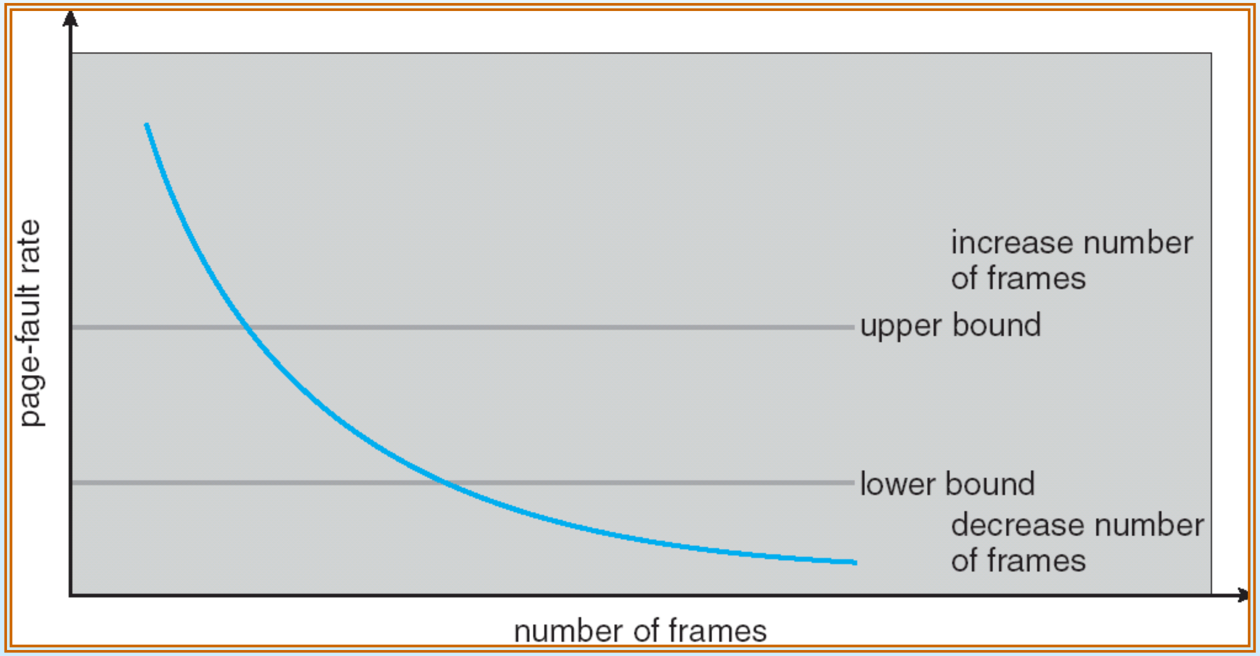
- Why does demand paging work?

- **Locality** model
  - Locality = set of pages in active use
  - Process migrates from one locality to another, e.g. main function, subroutine
  - Localities may overlap

- Why does **thrashing** occur?
  - size of locality > size of allocated frames

Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Other Issues -- Prepaging

- **Prepaging**
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and a fraction α of the s pages is used ($0 <= \alpha <= 1$)
    - Is cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses
    - α near one $\Rightarrow$ prepaging wins

Other Issues – Page Size

- **Page size selection** must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

End of Chapter 9