The picture can't be displayed.

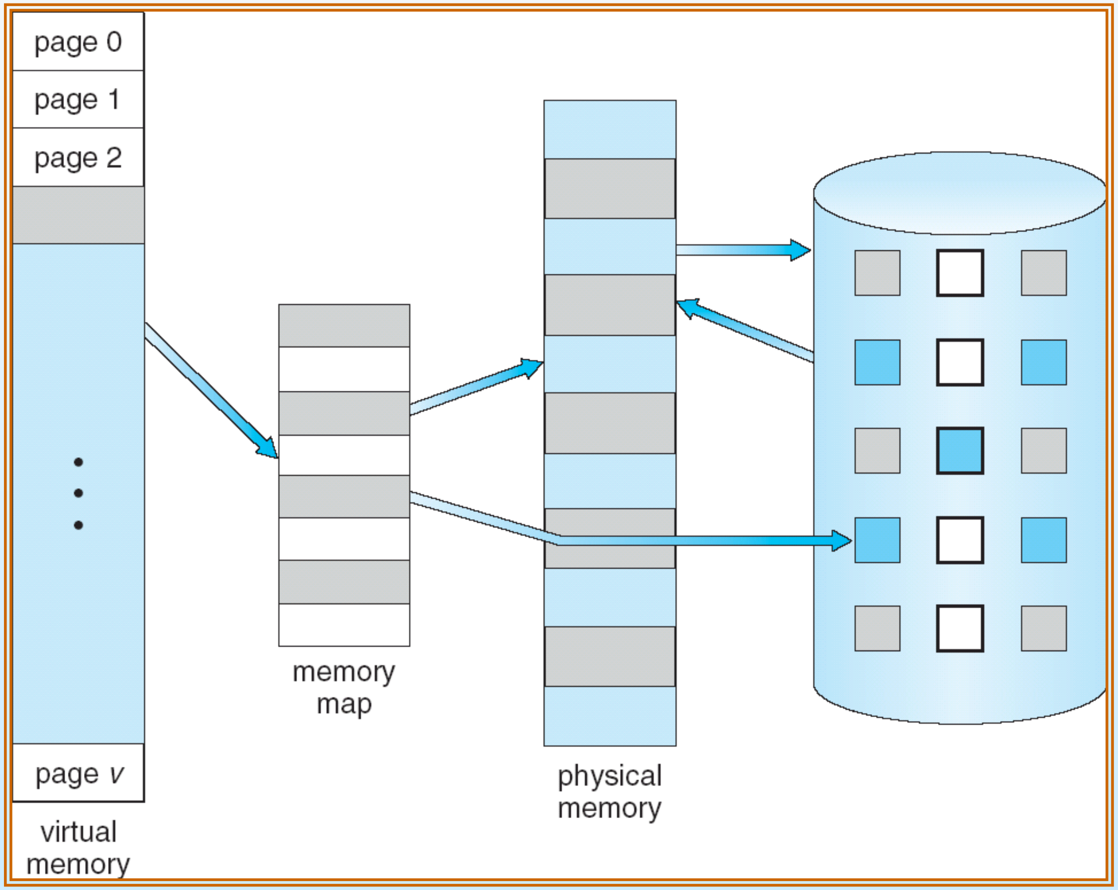
Chapter 9: Virtual Memory

Background

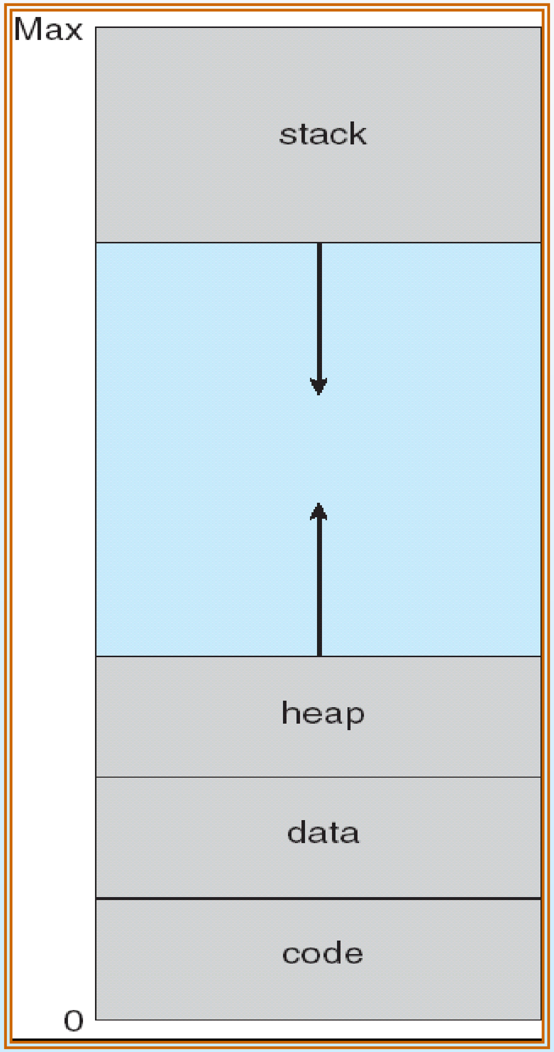
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows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

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

Virtual Memory That is Larger Than Physical Memory



Virtual-address Space

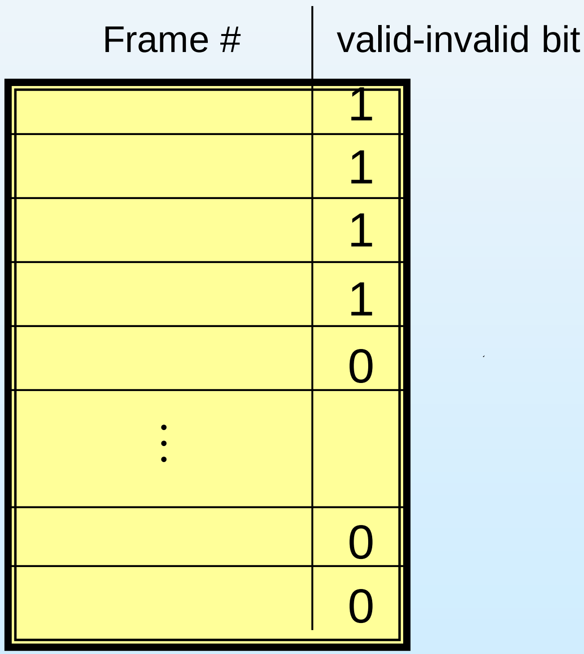


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

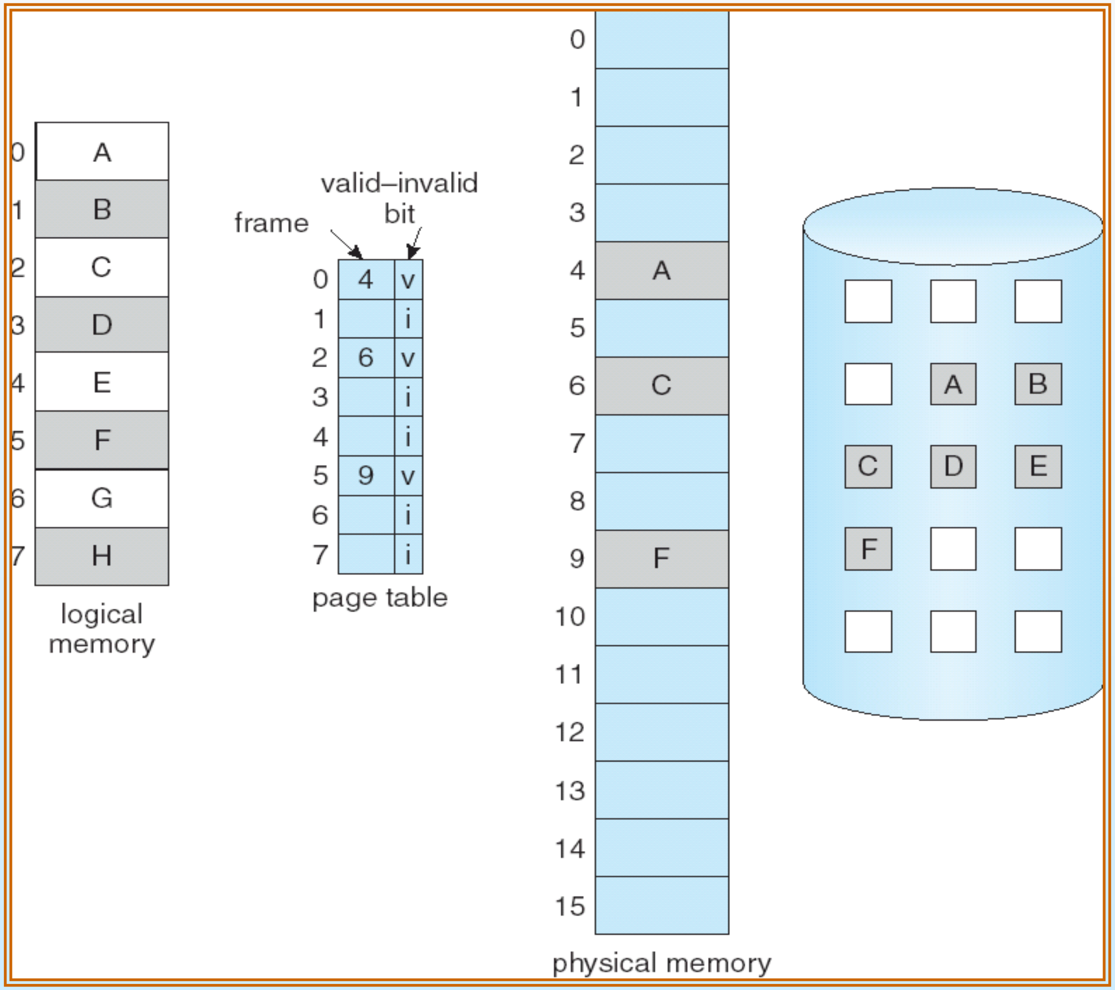
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:



page table

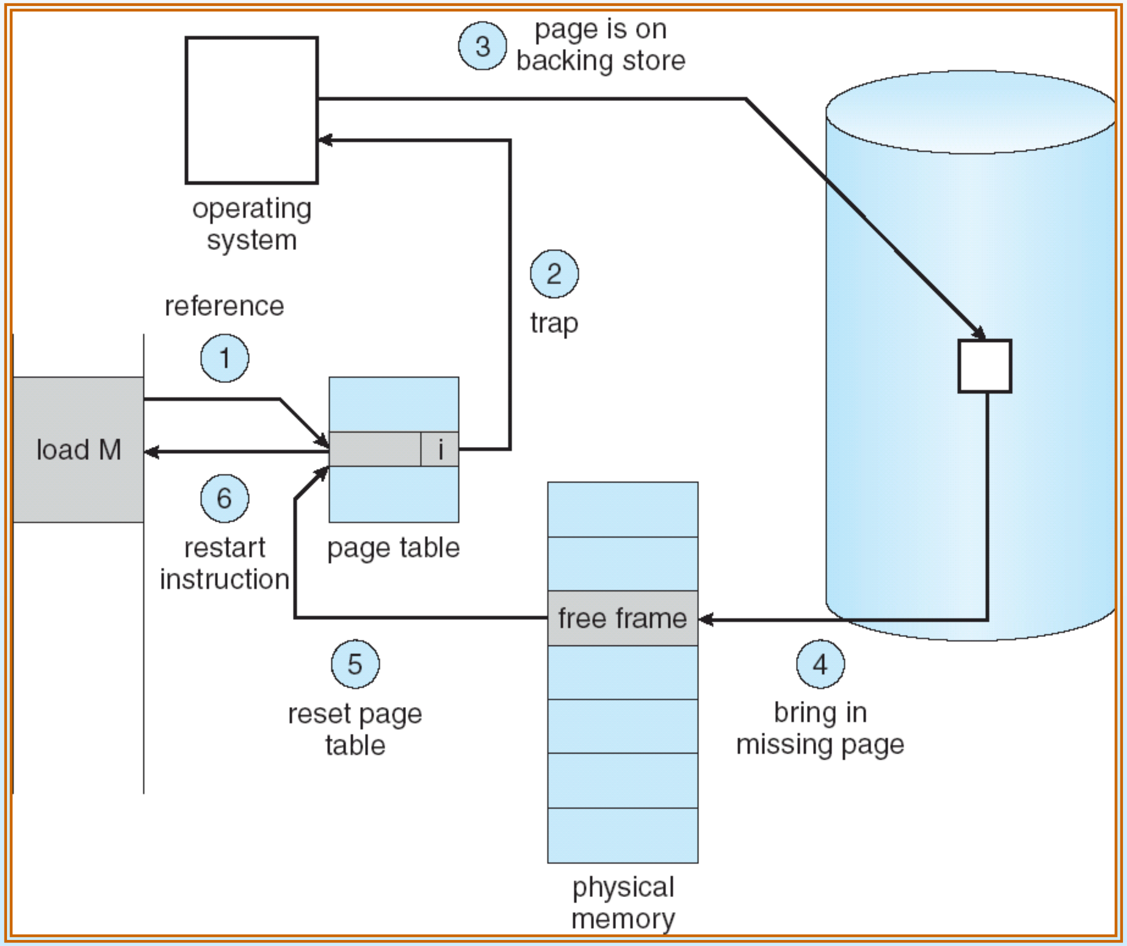
Page Table When Some Pages Are Not in Main Memory



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



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

Basic Page Replacement

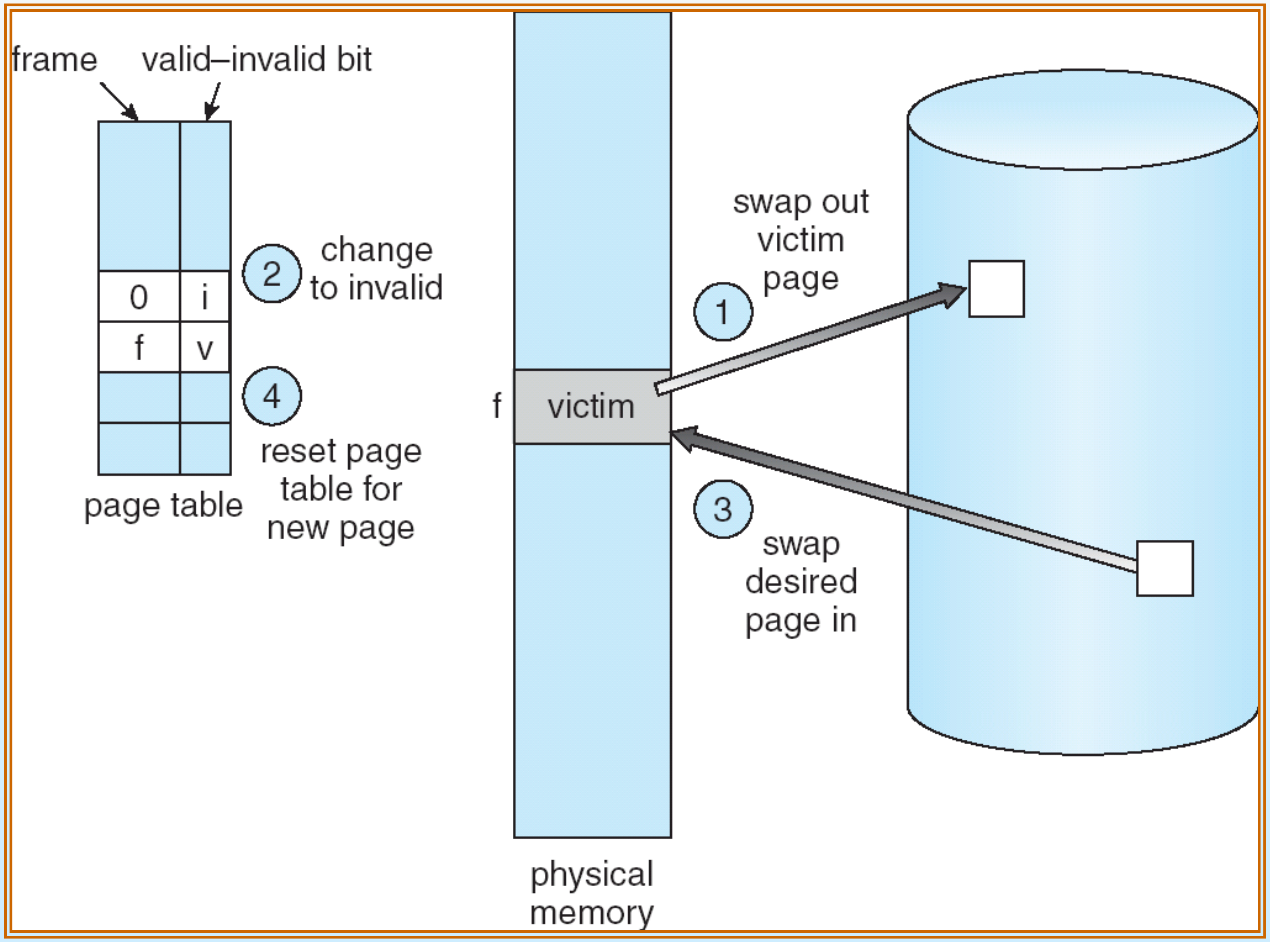
1. Find the location of the desired page on disk

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

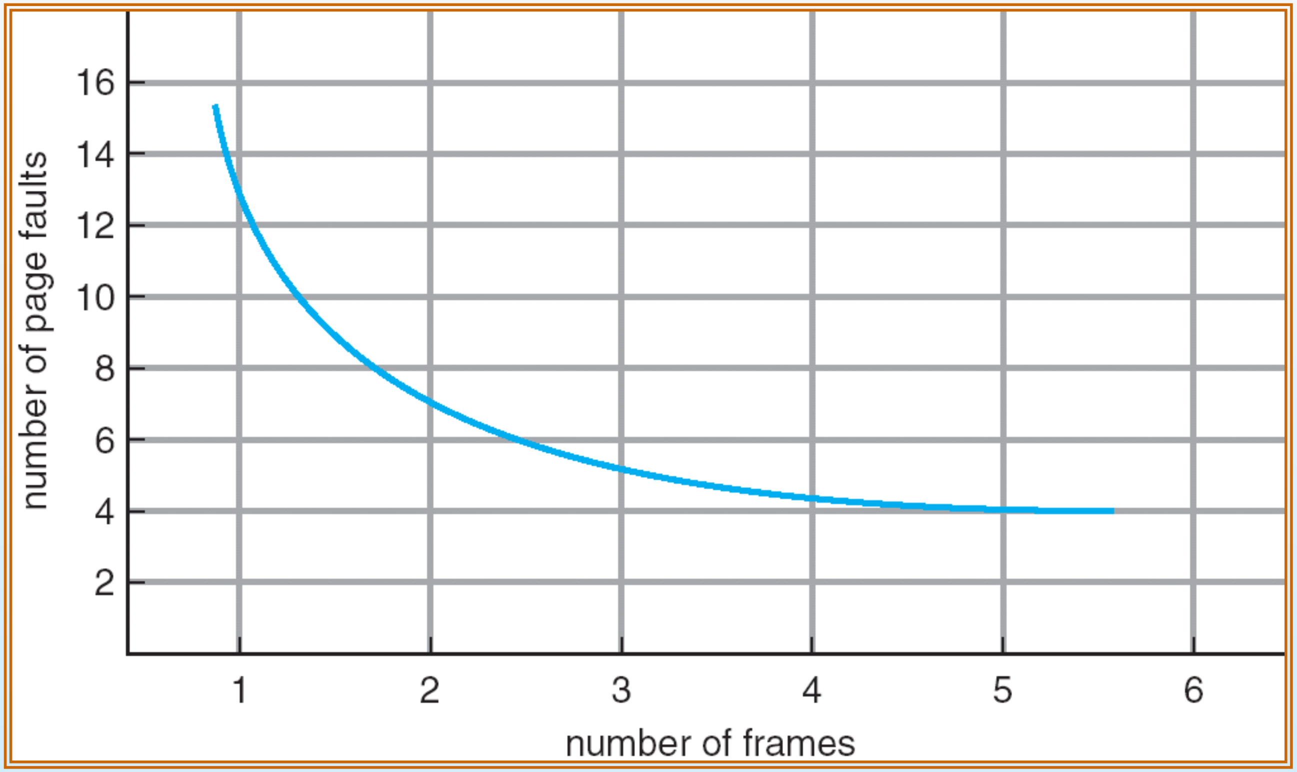
Page Replacement



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

- 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)

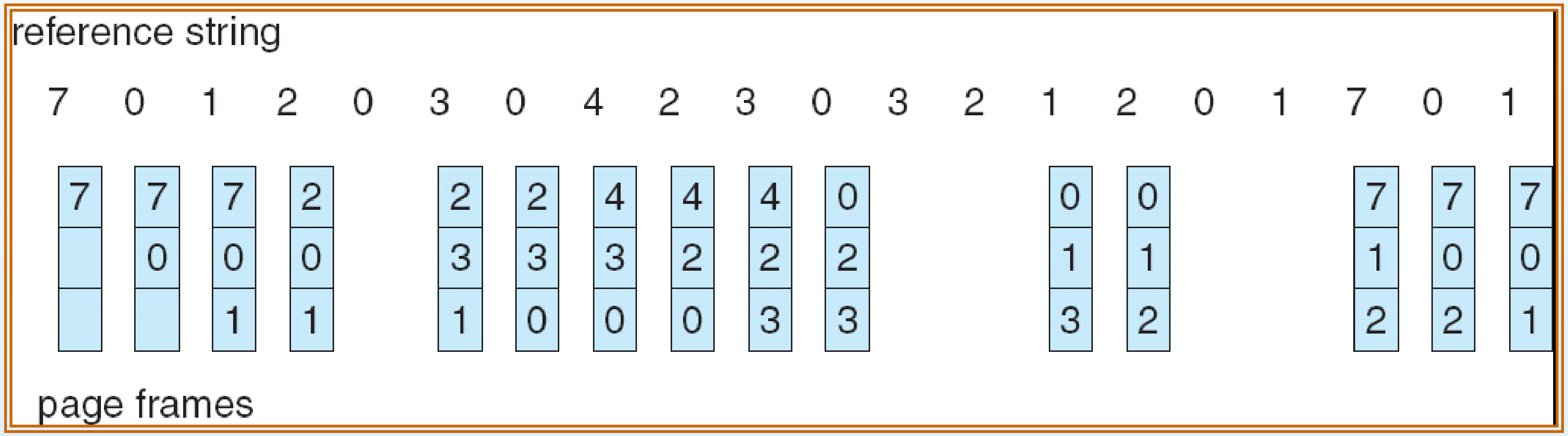
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

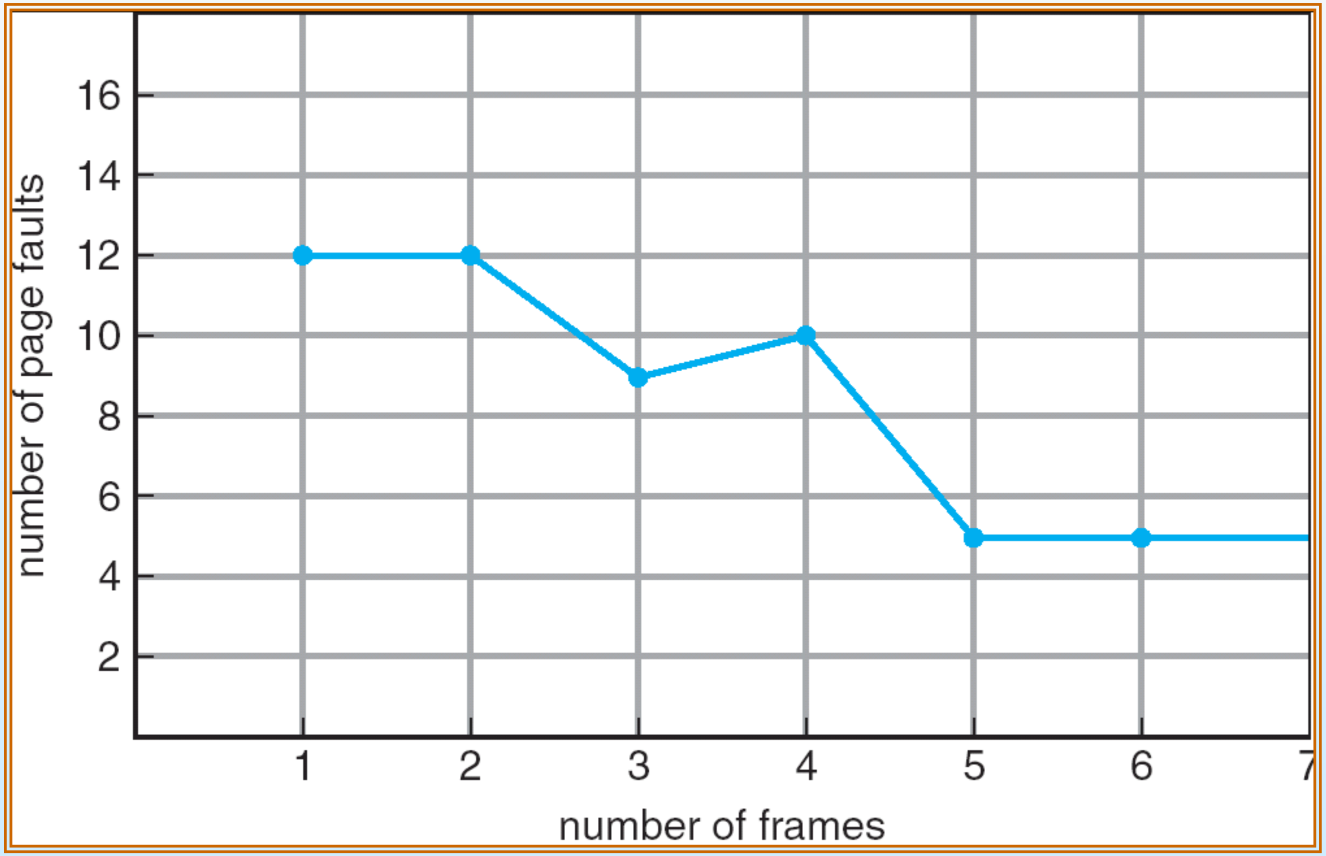
1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

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

FIFO Page Replacement



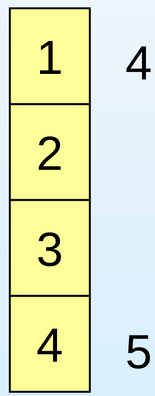
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

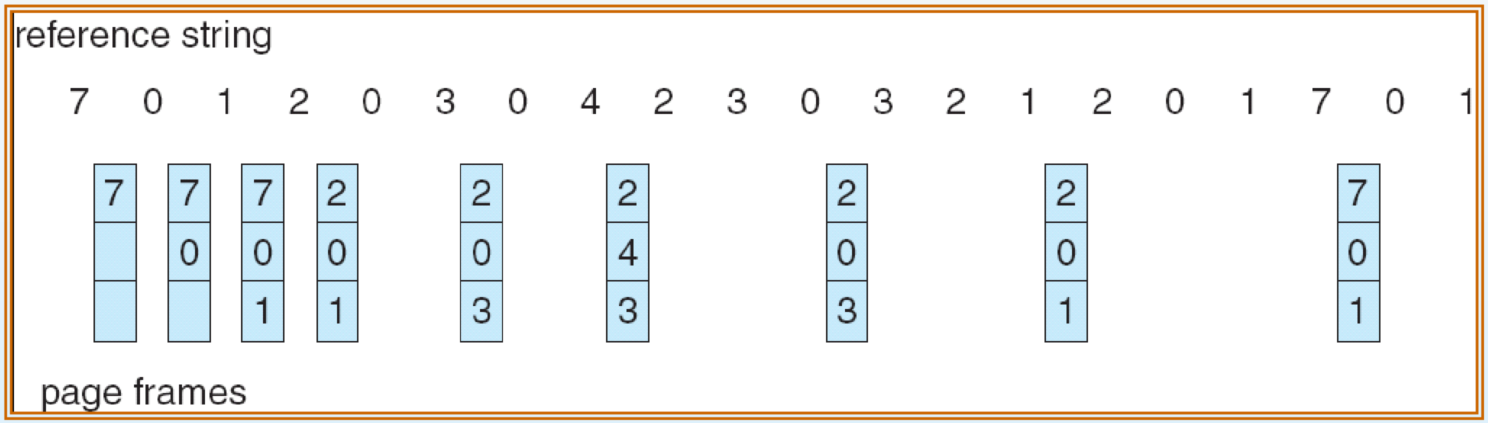
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

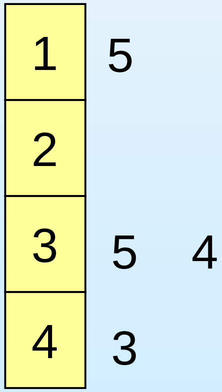
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



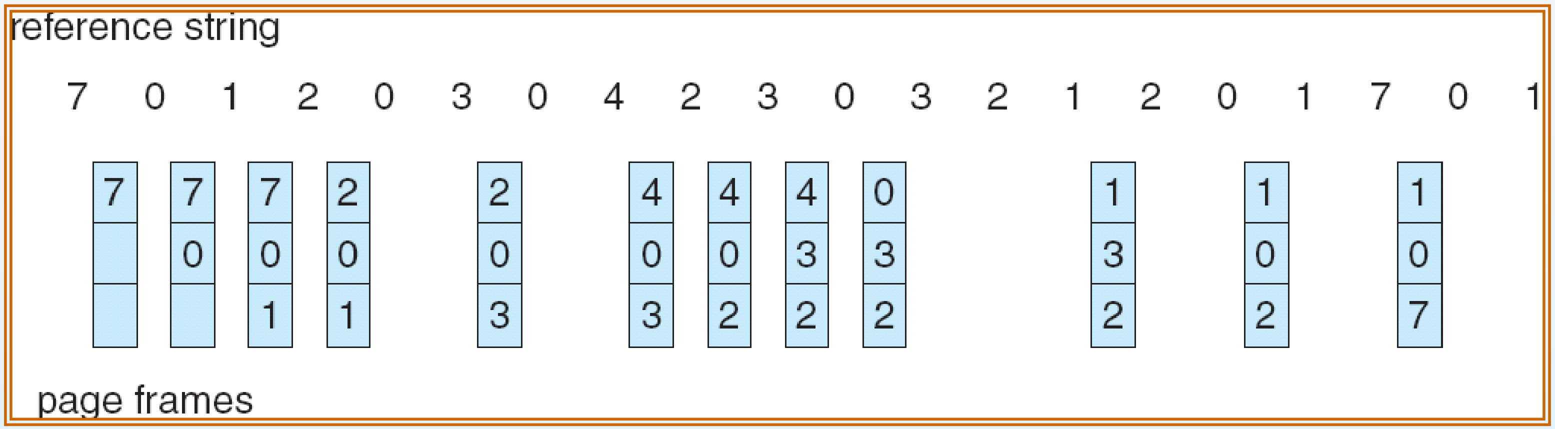
Least Recently Used (LRU) Algorithm

- 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

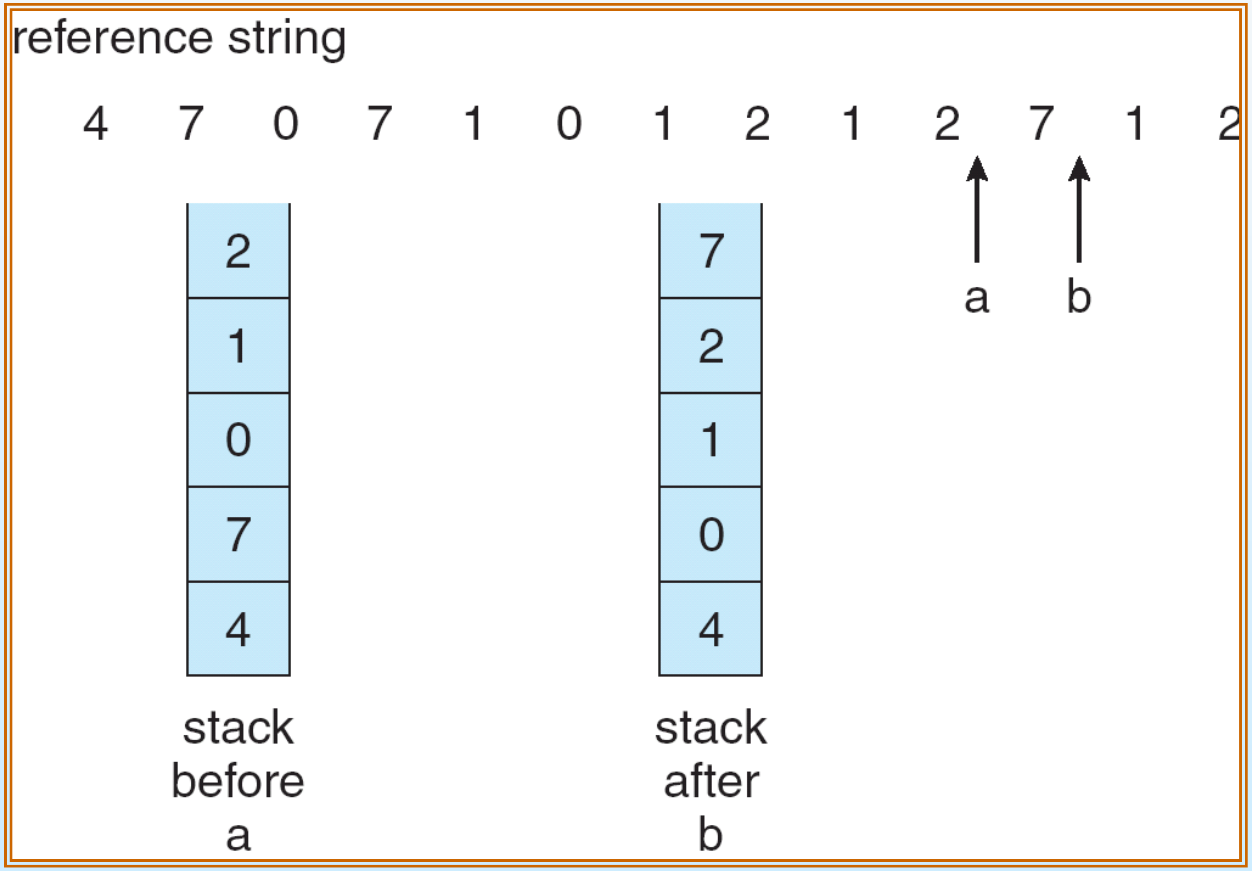
LRU Page Replacement



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
  - No search for replacement

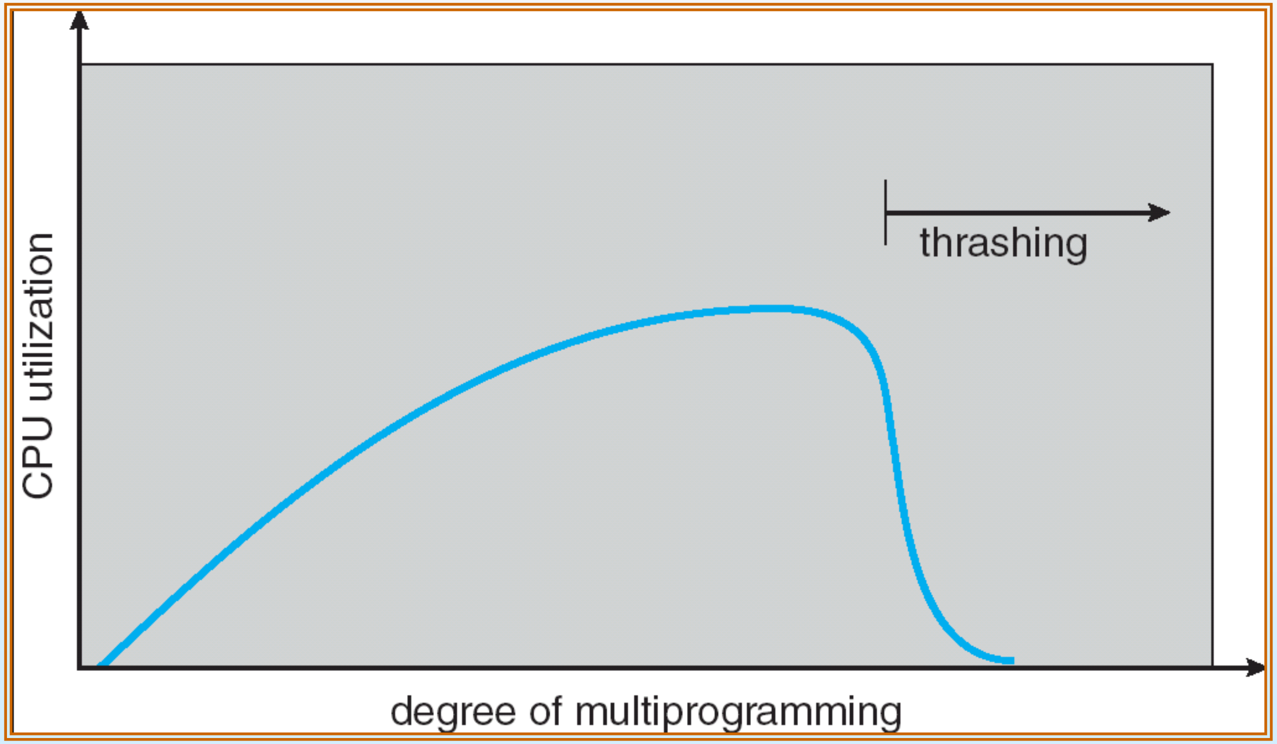
Use Of A Stack to Record The Most Recent Page References



Thrashing

- 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.)



Let the page fault service time be 1ms in a computer with average memory access time being 2ns. If one page fault is generated for every 1000000 memory access, what is the effective access time for the memory in nanosecond? [1 millisecond= 1000000 nanosecond]

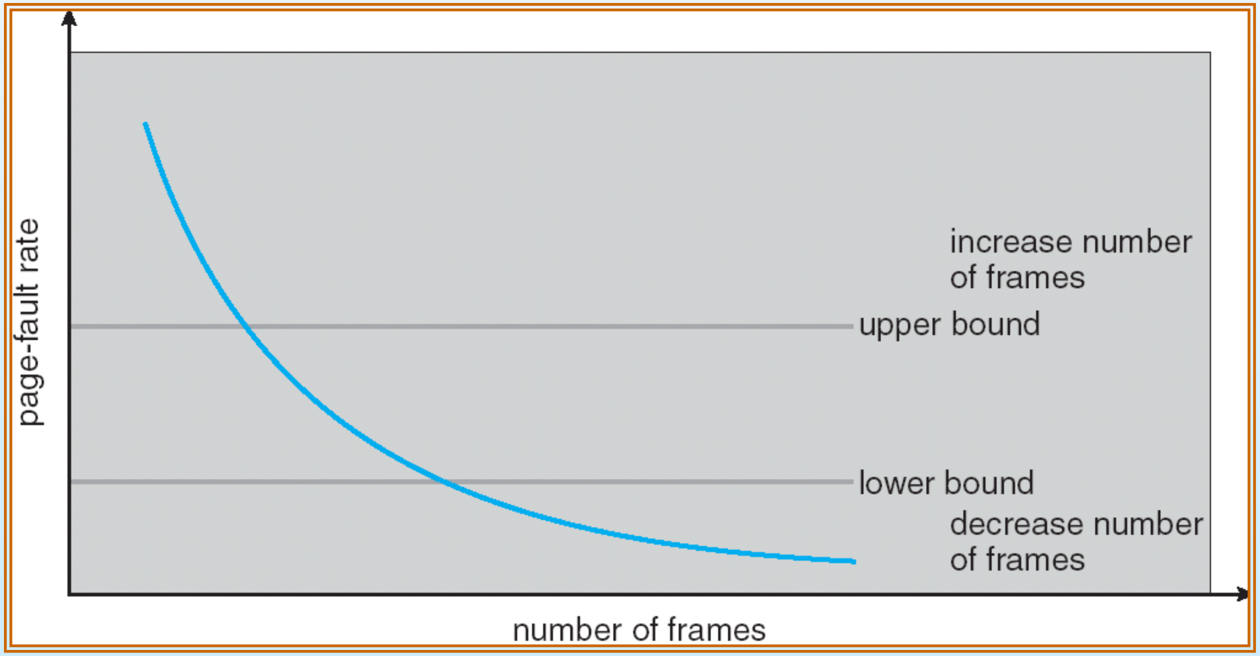
```
Let P be the page fault rate
Effective Memory Access Time = p * (page fault service time) + (1
- p) * (Memory access time)
= ( 1/(10^6) )* 1 * (10^6) ns + (1 - 1/(10^6)) * 2 ns
```

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

The picture can't be displayed.

End of Chapter 9