COP 4600
Operating Systems Design

Spring 2019

Virtual Memory

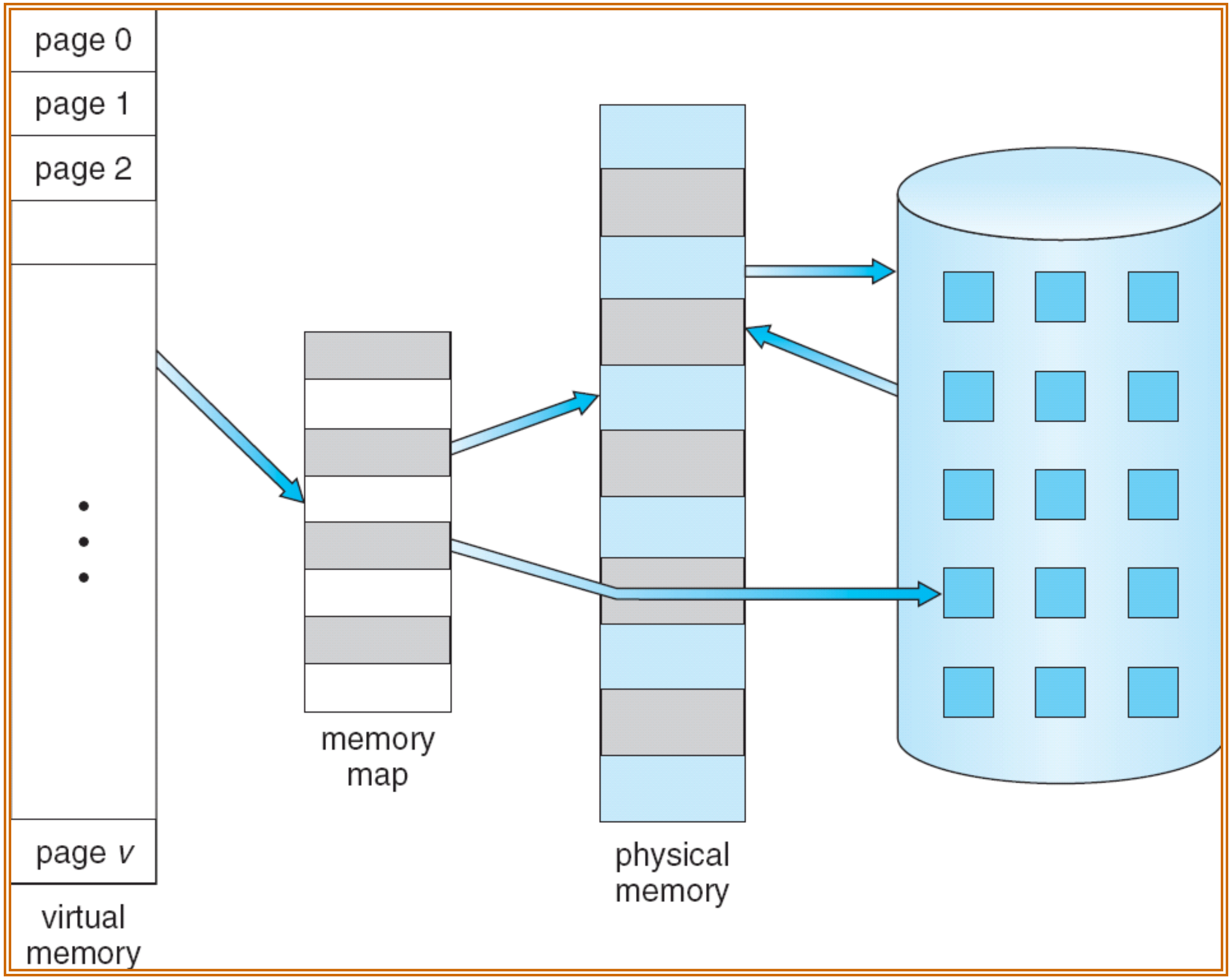
Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

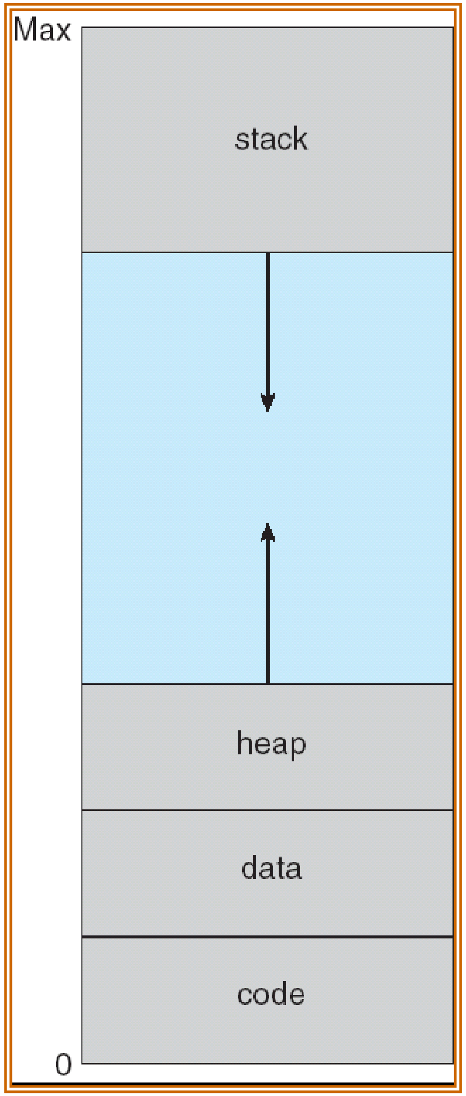
Background

- **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

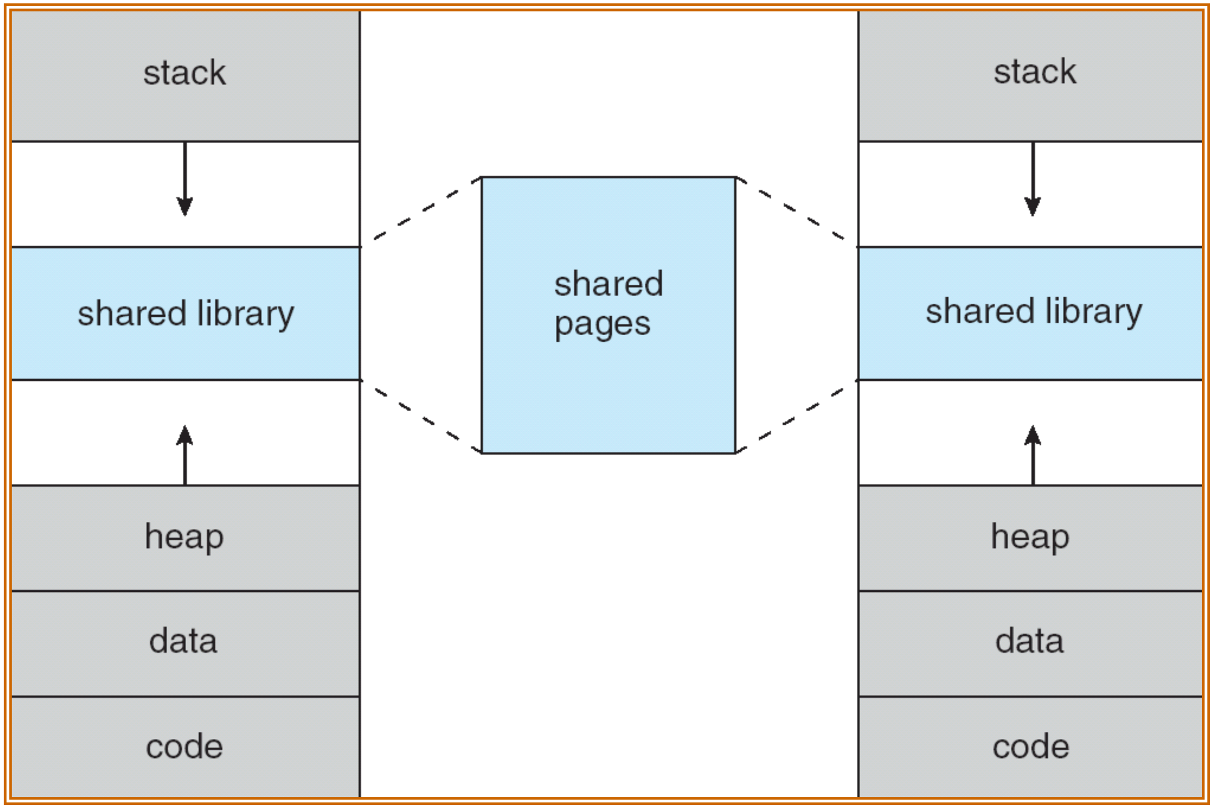
Virtual Memory Can Be Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory

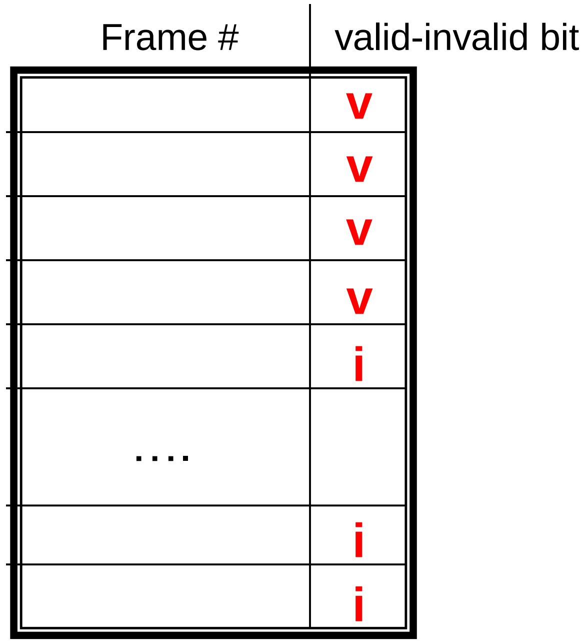


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - More users
- **Lazy swapper** – never swaps a page into memory unless page will be needed

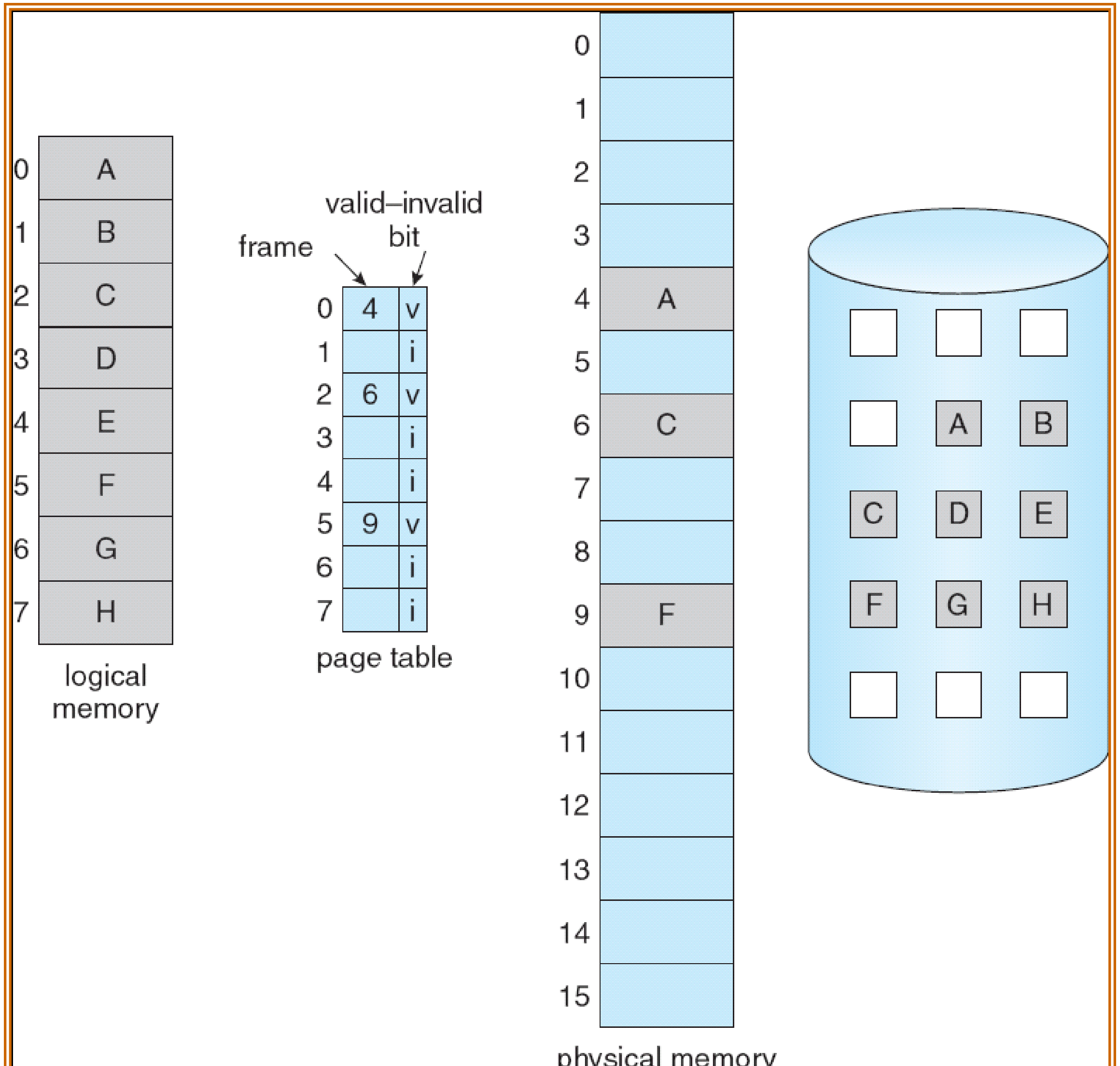
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (**v** $\Rightarrow$ in-memory, **i** $\Rightarrow$ not-in-memory)
- What's the initial value?
  - Initially valid–invalid bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry
  is **i** $\Rightarrow$ page fault



page table

Page Table When Some Pages Are Not in Main Memory



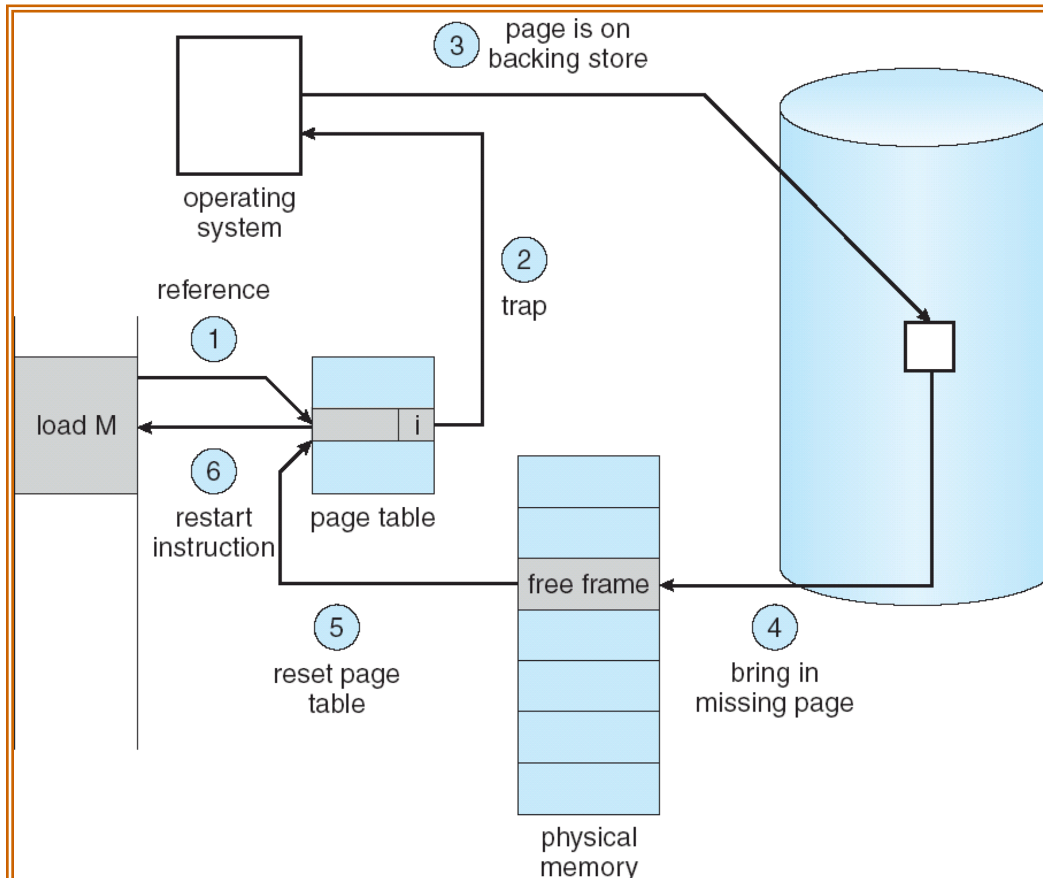
Page Fault

- If there is a reference to a page that is not in memory, first reference to that page will trap to operating system:

page fault

Operating system looks at an internal table (kept with PCB) to decide:

- Invalid reference ⇒ abort
- Just not in memory



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)

$$
\begin{aligned}
\text{EAT} = (1 - p) &\times \text{memory access} \\
&+ p\,(\text{page fault overhead} \\
&\qquad + \text{swap page out} \\
&\qquad + \text{swap page in} \\
&\qquad + \text{restart overhead} \\
&\quad)
\end{aligned}
$$

Demand Paging Example

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

- EAT = $(1 – p) \times 200 + p \times$ (8 milliseconds)

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

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

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

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

Benefits to Process Creation

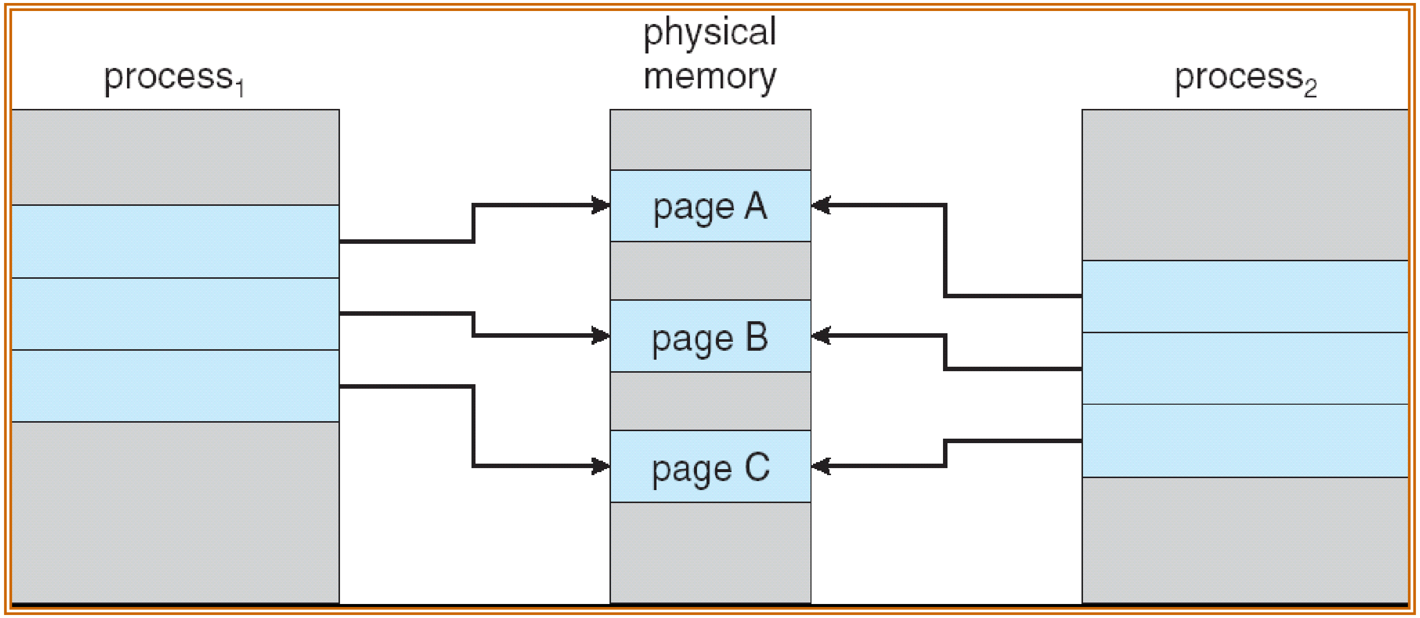
- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

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



What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
- Performance
  - Algorithm – want an algorithm which will result in minimum number of page faults
  - Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

Basic Page Replacement

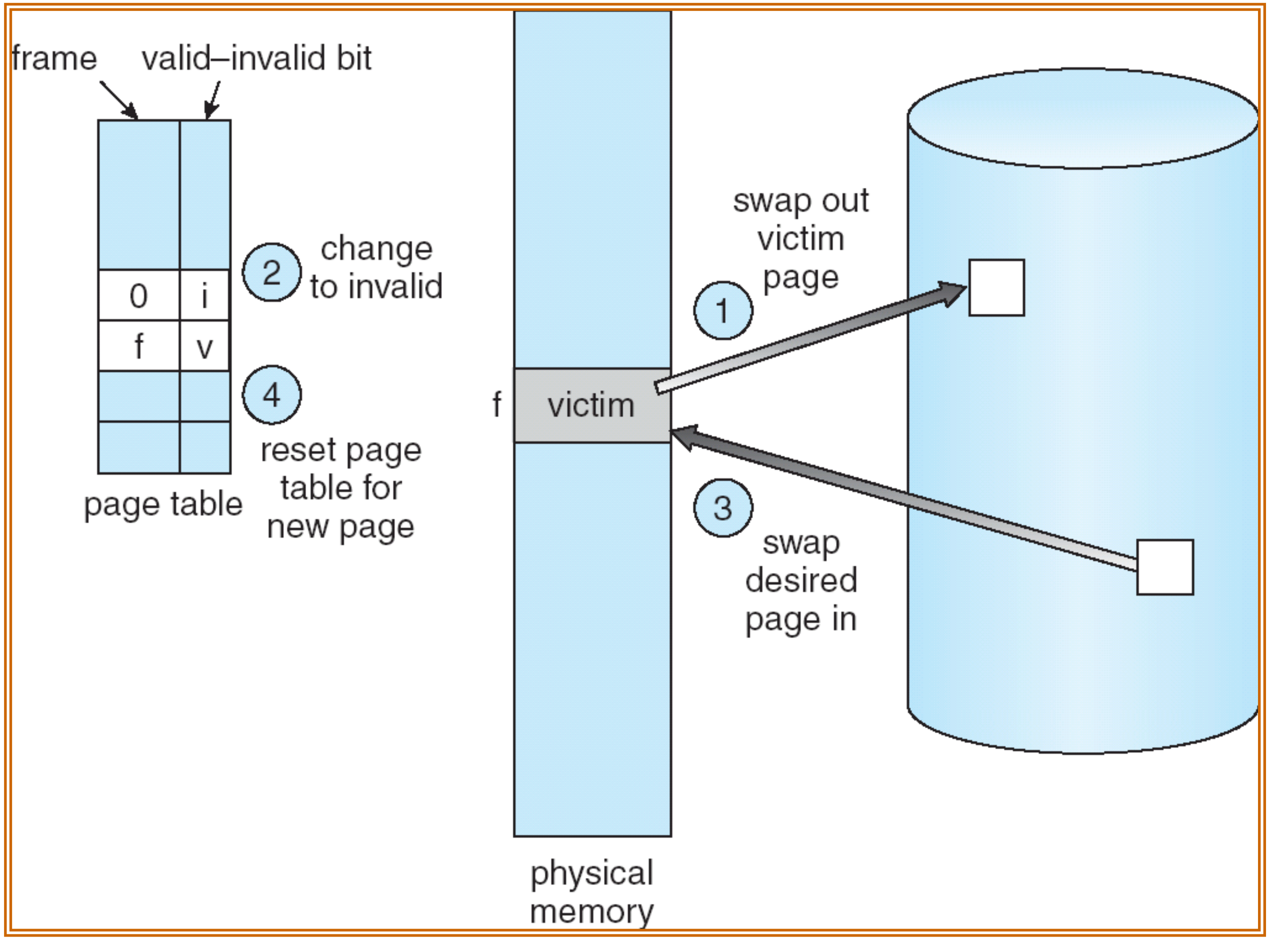
1. Find the location of the desired page on disk

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

4. Restart the process

Page Replacement



Page Replacement Algorithms

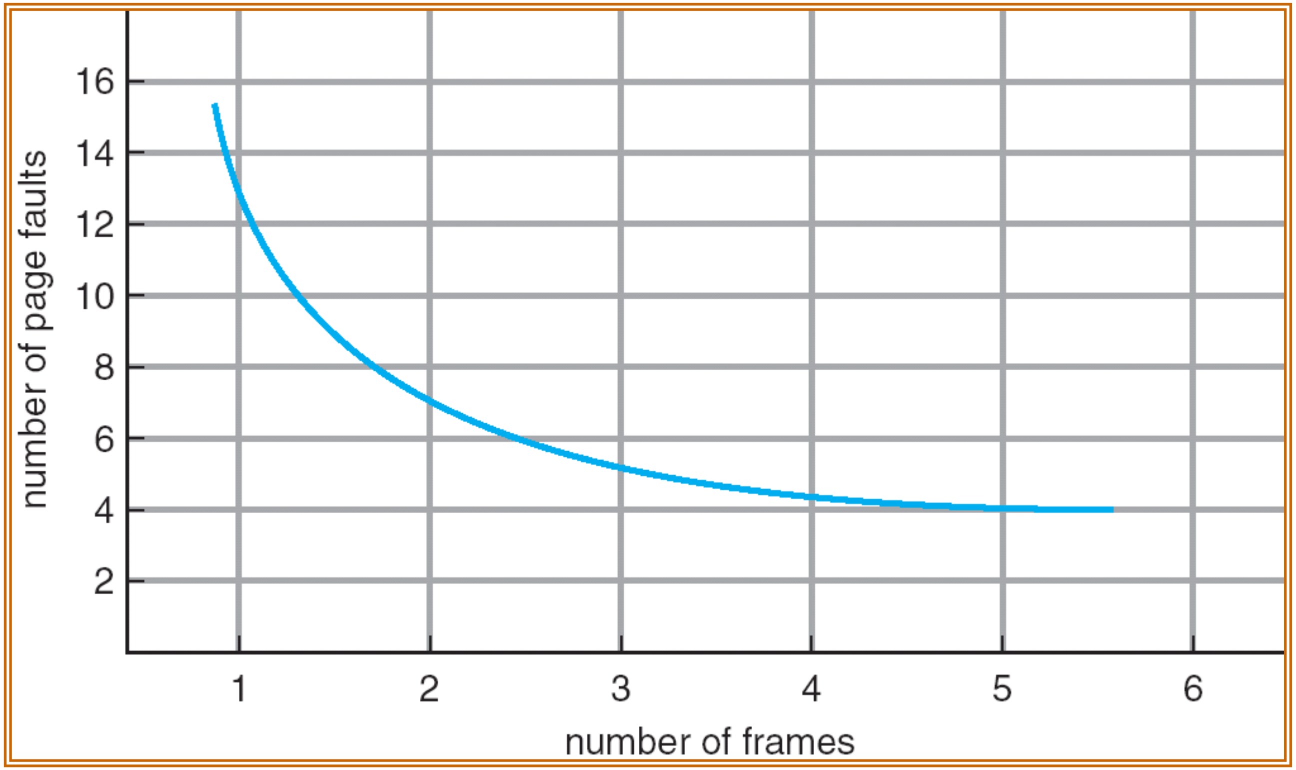
- Goal
  - want lowest page-fault rate

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

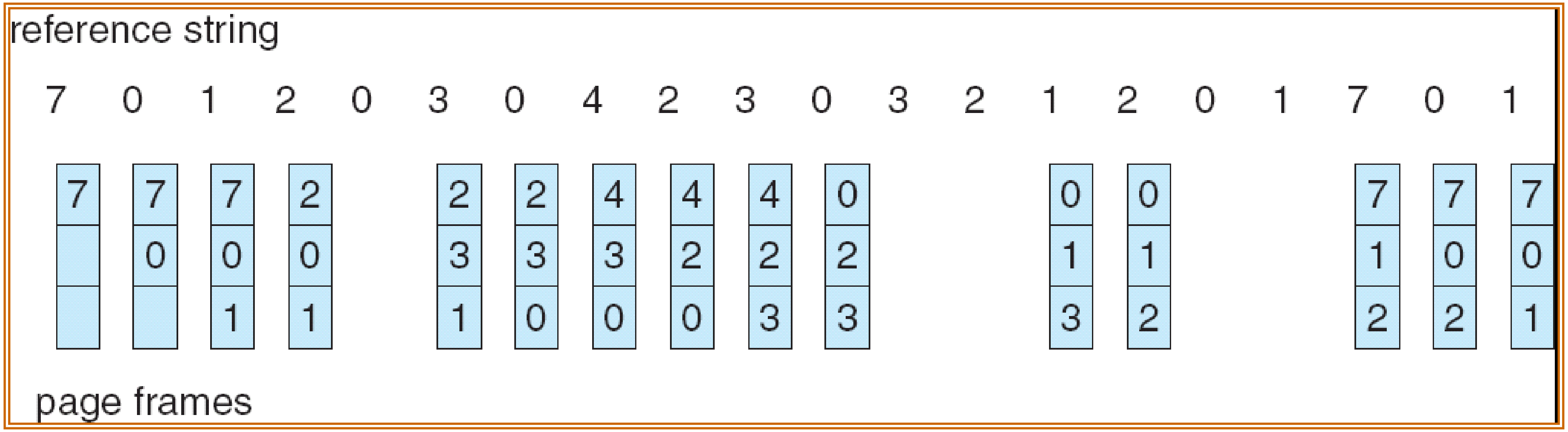
- In all our examples, the reference string is

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

Graph of Page Faults Versus The Number of Frames



FIFO Page Replacement



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	

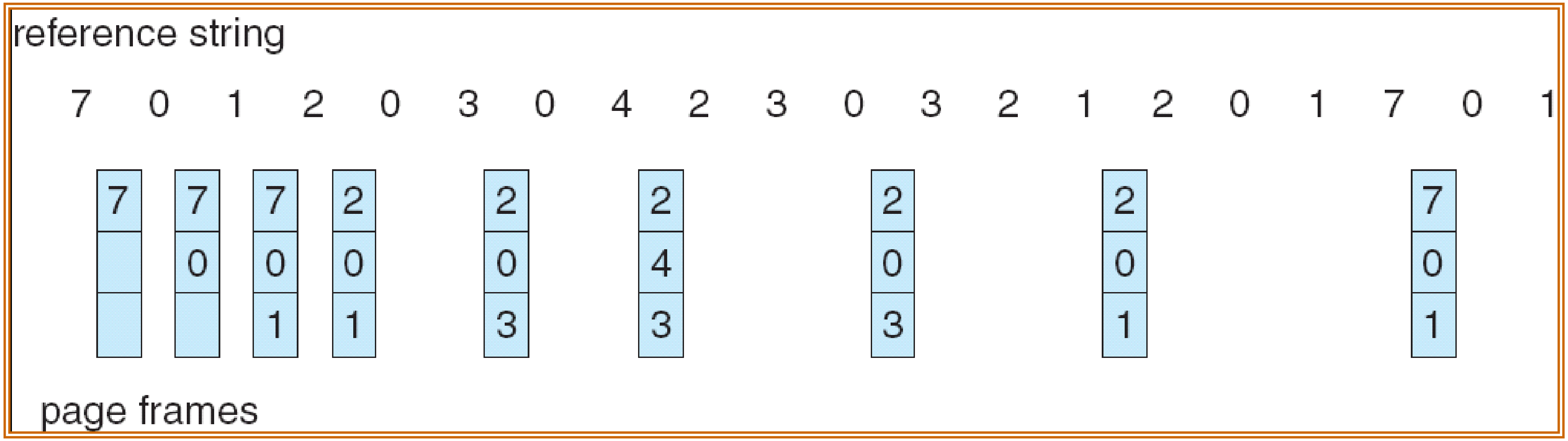
- 4 frames

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

Anomaly: more frames ⇒ more page faults

Optimal Algorithm

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



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

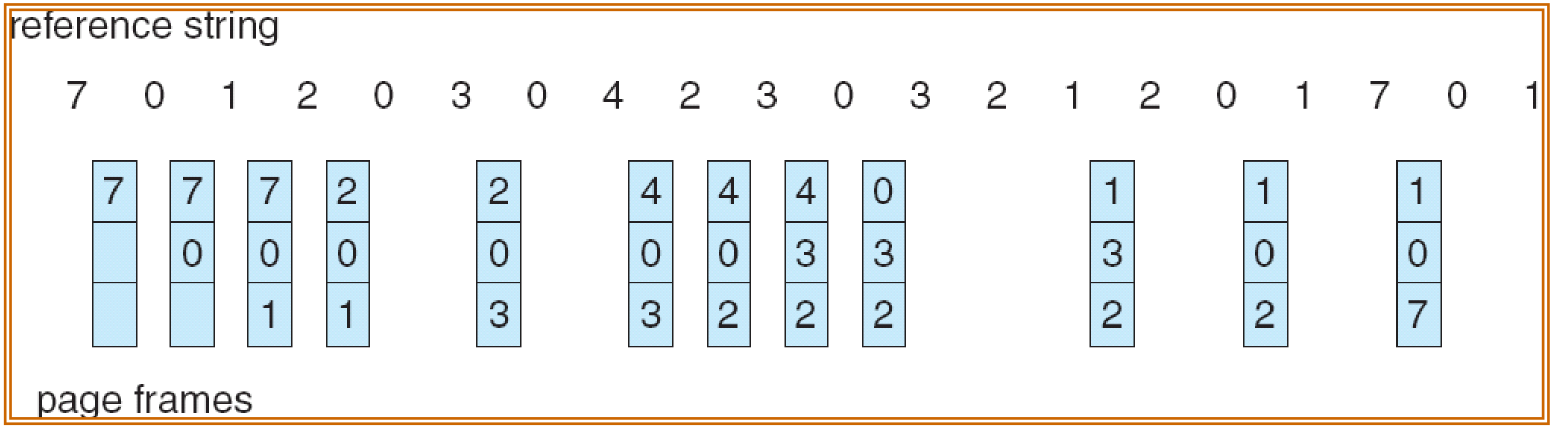
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

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

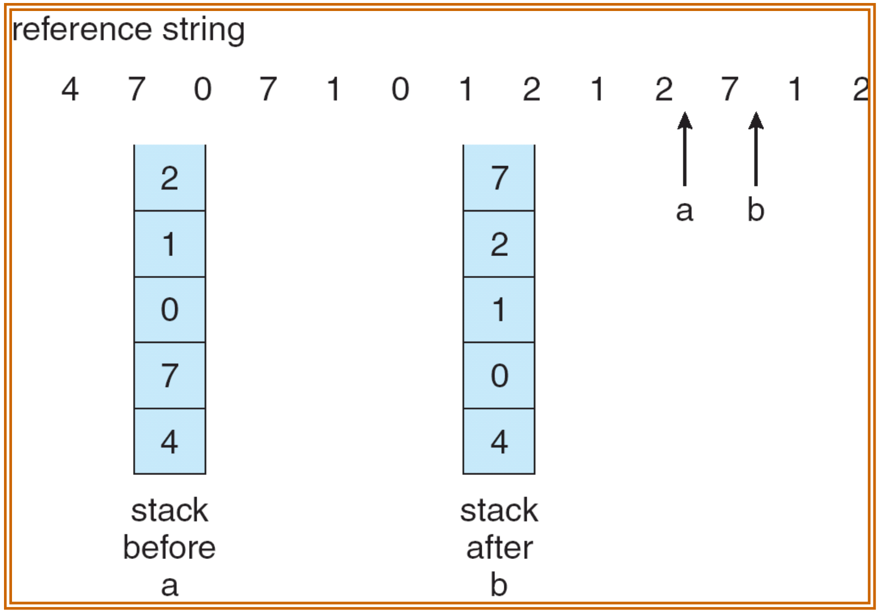
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement



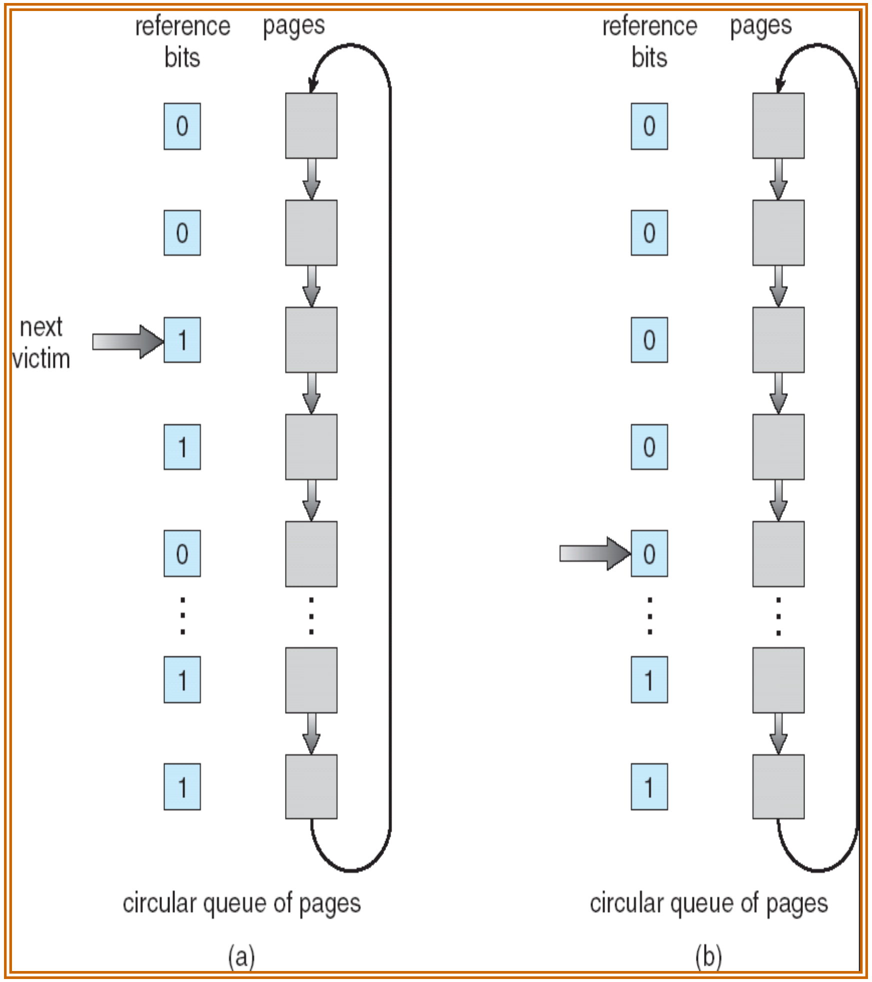
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
  - No search for replacement



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced, bit is set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules



Counting Algorithms

- Keep a counter of the number of references that have been made to each page

- **LFU (least frequently used) page-replacement Algorithm**: replaces page with smallest count

- **MFU (most frequently used) page-replacement Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used too much

Allocation of Frames

- Each process needs *minimum* number of pages
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

Priority Allocation

- 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

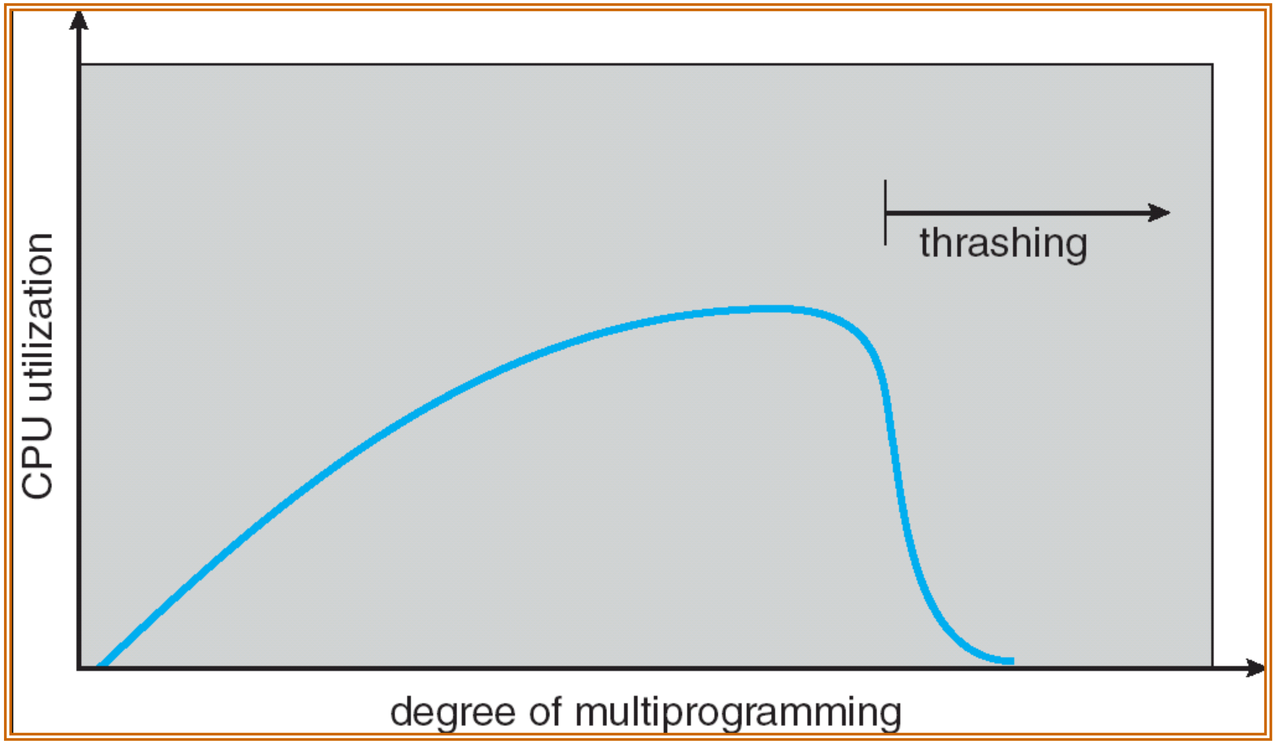
Global vs. Local Replacement

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization. (why?)
  - 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.)



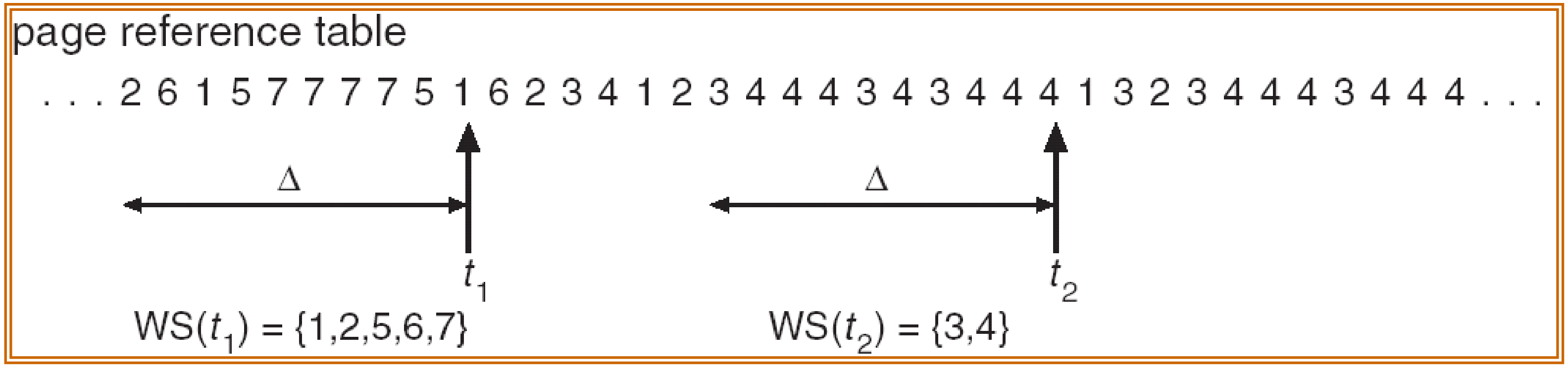
Two Approaches to Prevent Thrashing

- Working-set Model
- Page-Fault Frequency

Working-Set Model

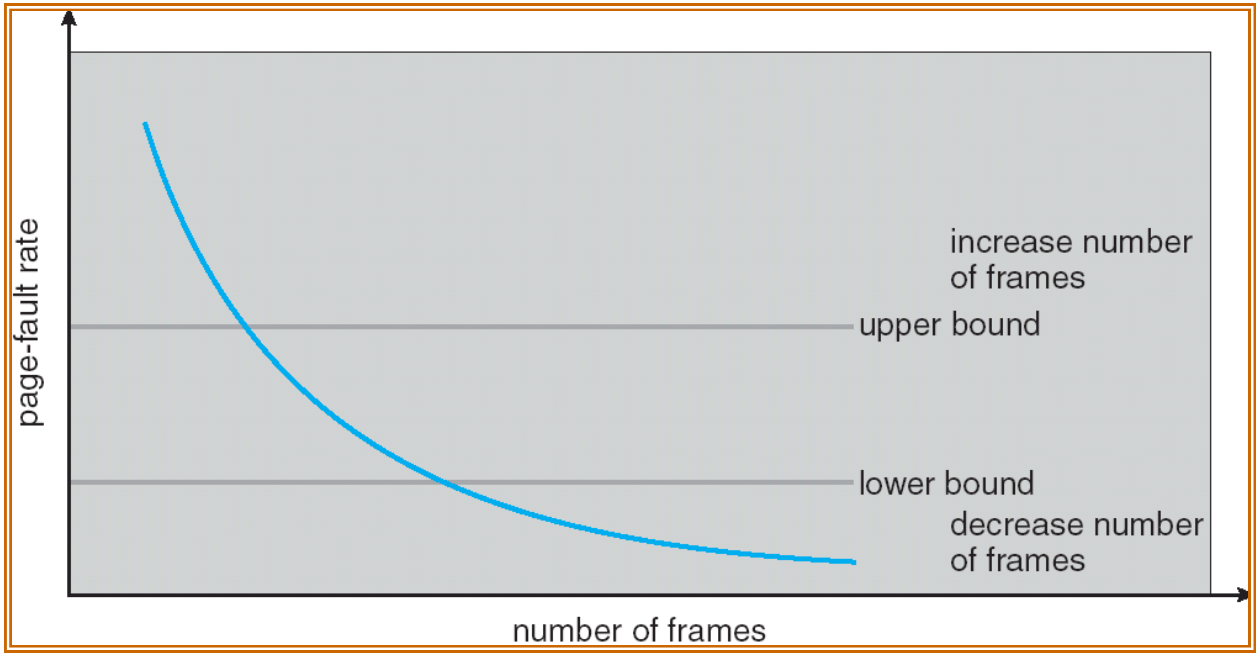
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m$ *(memory)* $\Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model

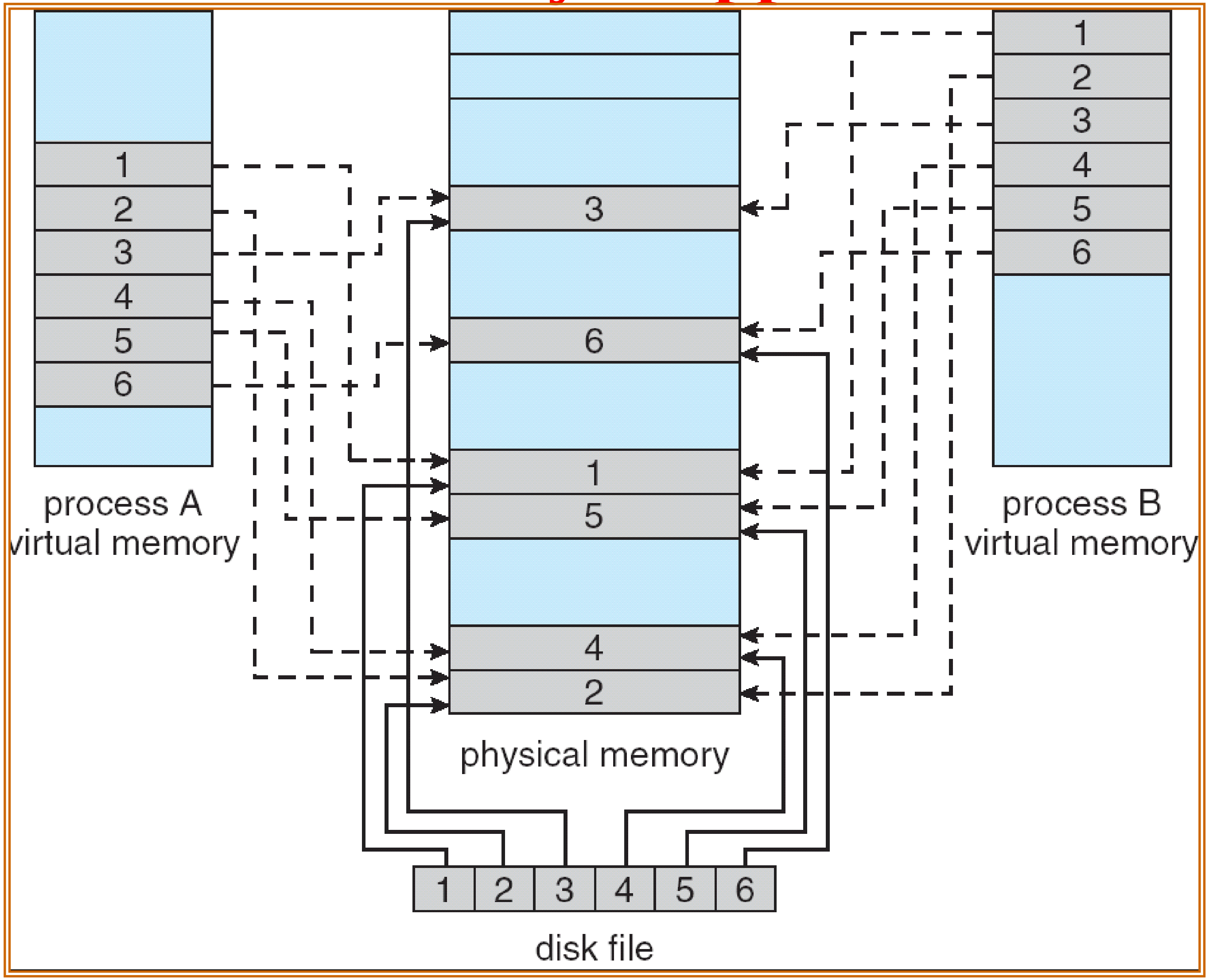


Page-Fault Frequency Scheme

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



Memory Mapped Files



Memory-Mapped Files

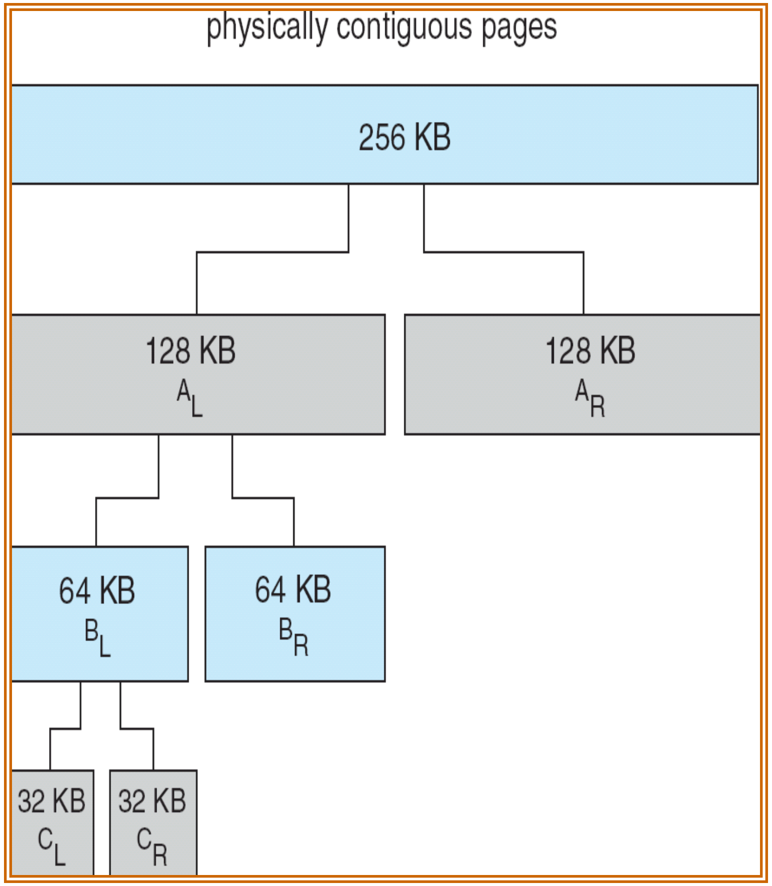
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **`read() write()`** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- Any Benefits?

Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
- Approaches
  - Buddy System
  - Slab Allocation

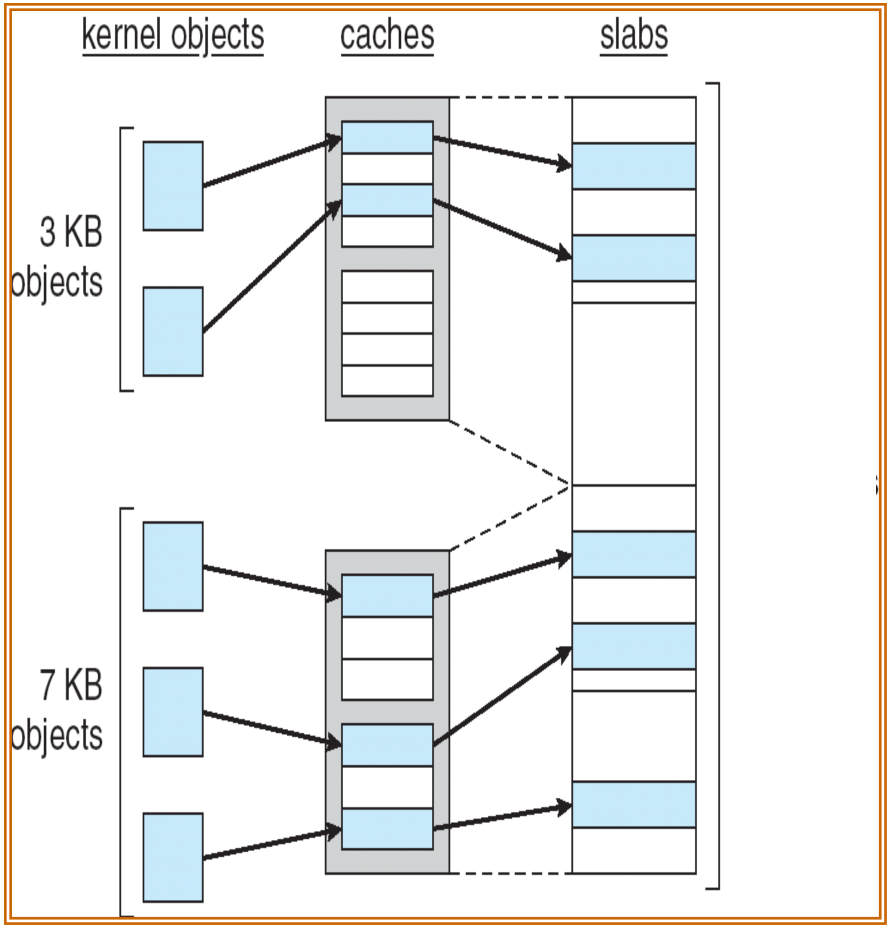
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- Drawback?



Slab Allocator

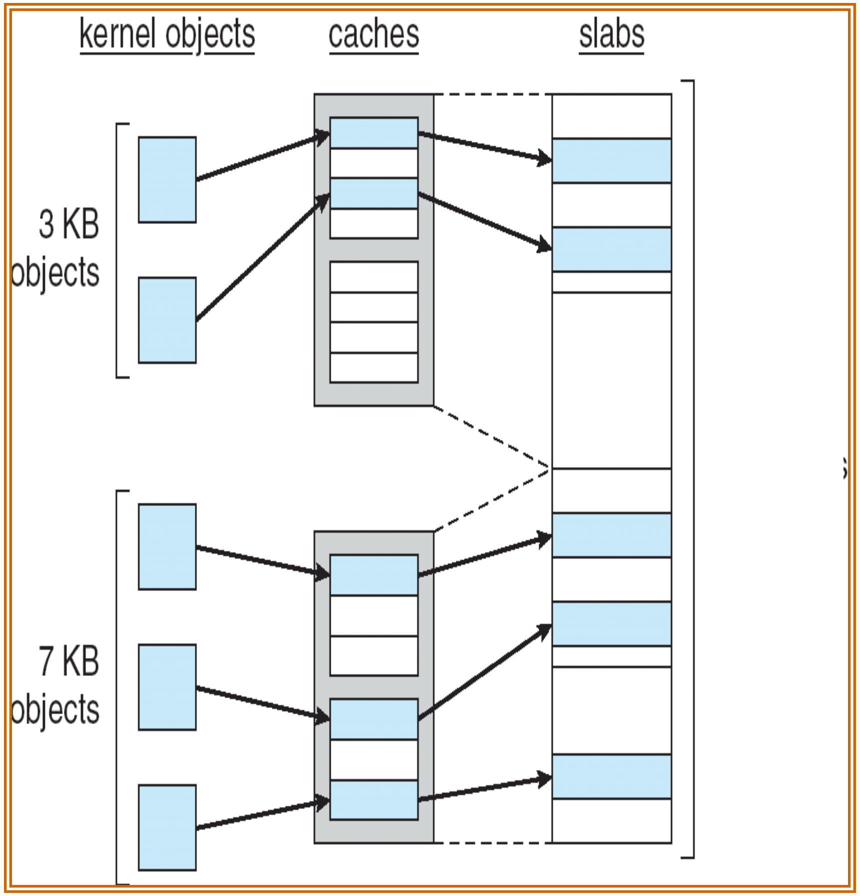
- Alternate strategy
- **Slab** is one memory segment which preserves a specific object.
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure



Slab Allocator

- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include
  - no fragmentation
  - fast memory request satisfaction

    (allocated and deallocated)



Other Issues

- Prepaging
  - To reduce 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
- Page size selection must take into consideration:
  - (internal) fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Approaches to increase TLB Reach. ***Hint: Page Size?***
  - Increase the Page Size
    - This may lead to an increase in fragmentation as not all applications require a large page size
  - Provide Multiple Page Sizes
    - Has to manage TLB in software

Other Issues – Program Structure

- Program structure
  - `Int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

OS Example: Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
  - Working set trimming removes pages from processes that have pages in excess of their working set minimum