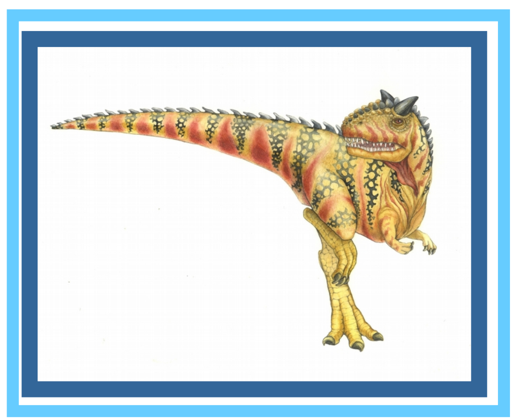
Chapter 8: Virtual Memory



Chapter 8: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

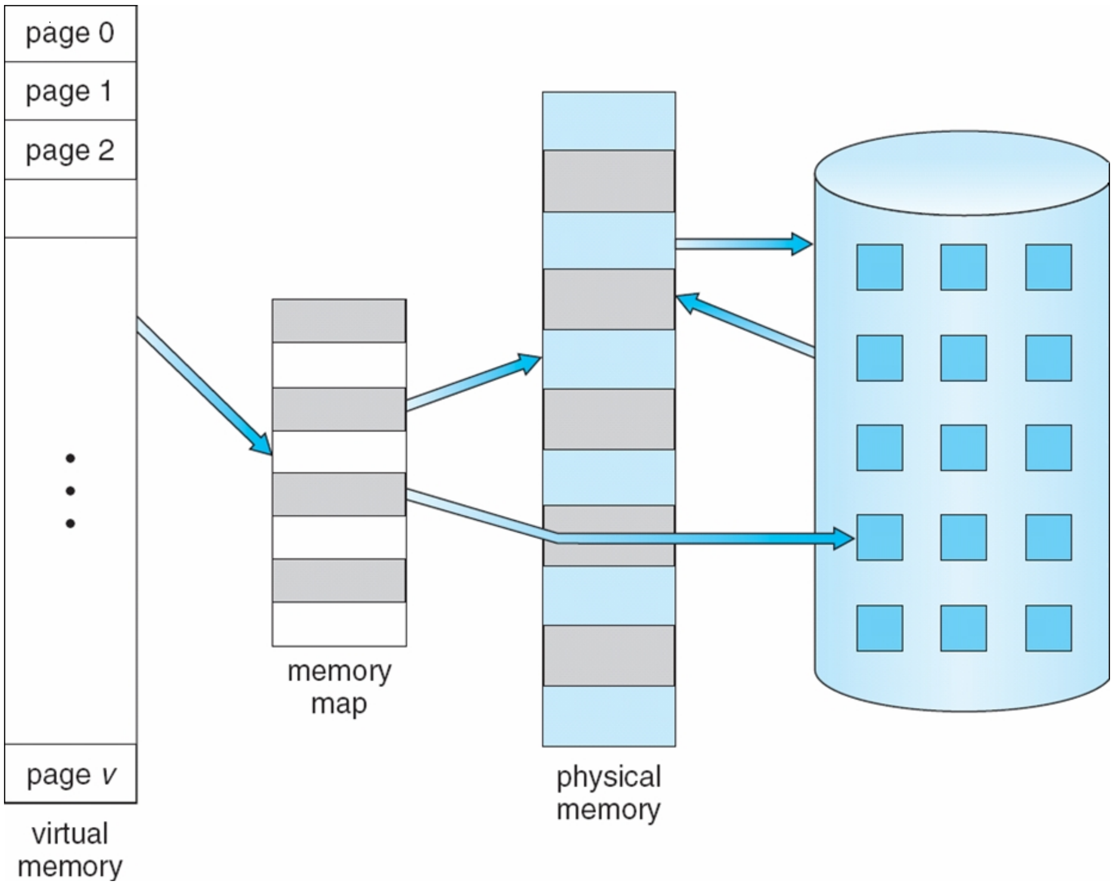
Objectives

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

Background

- **Virtual memory** – separation of 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

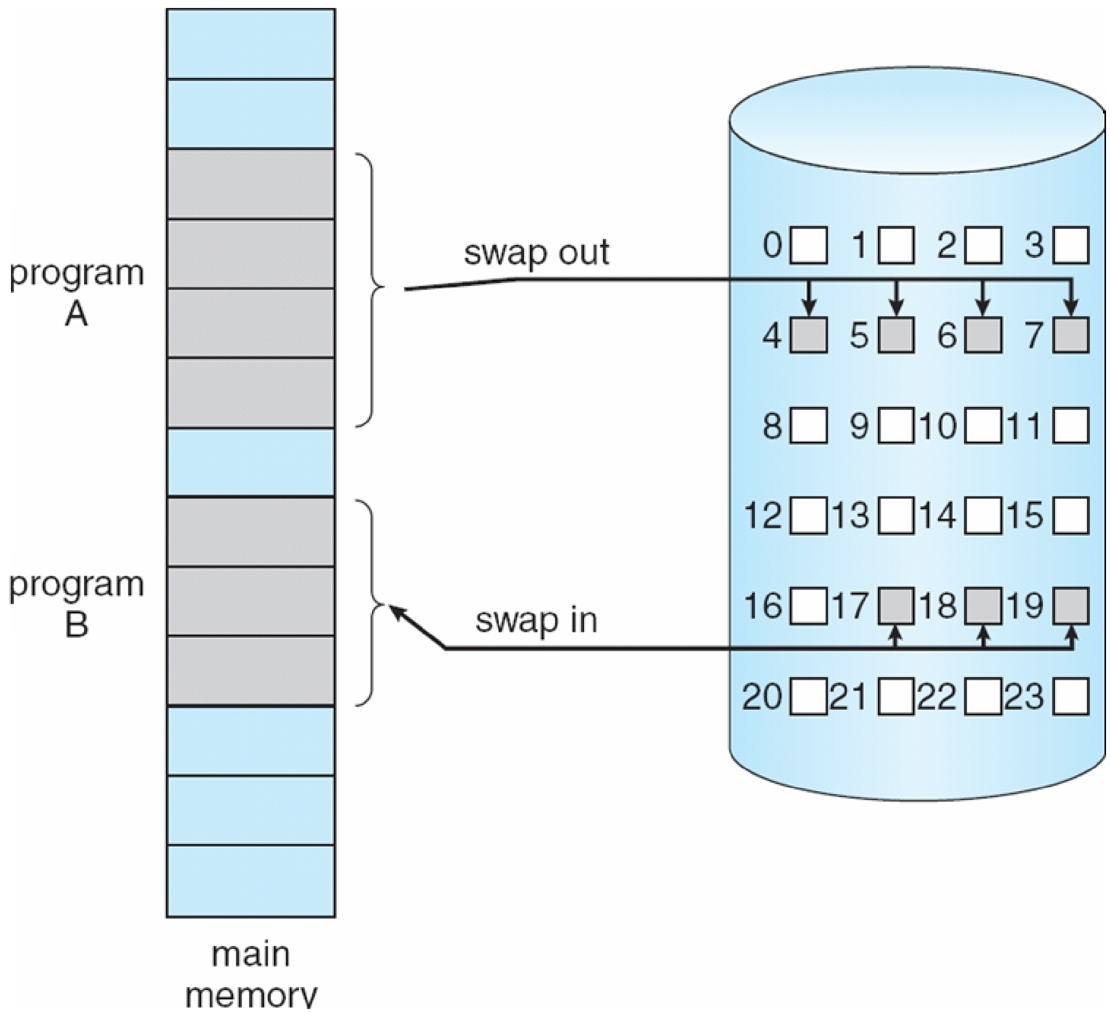
Virtual Memory That is Larger Than Physical Memory



Demand Paging

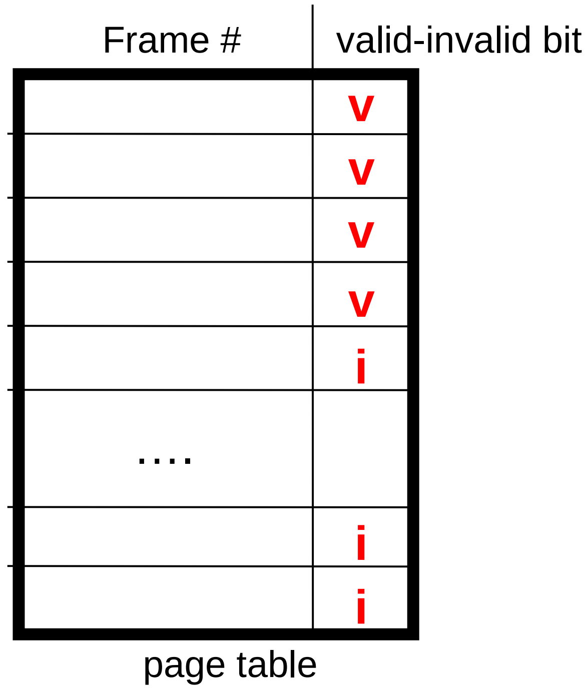
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page is needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



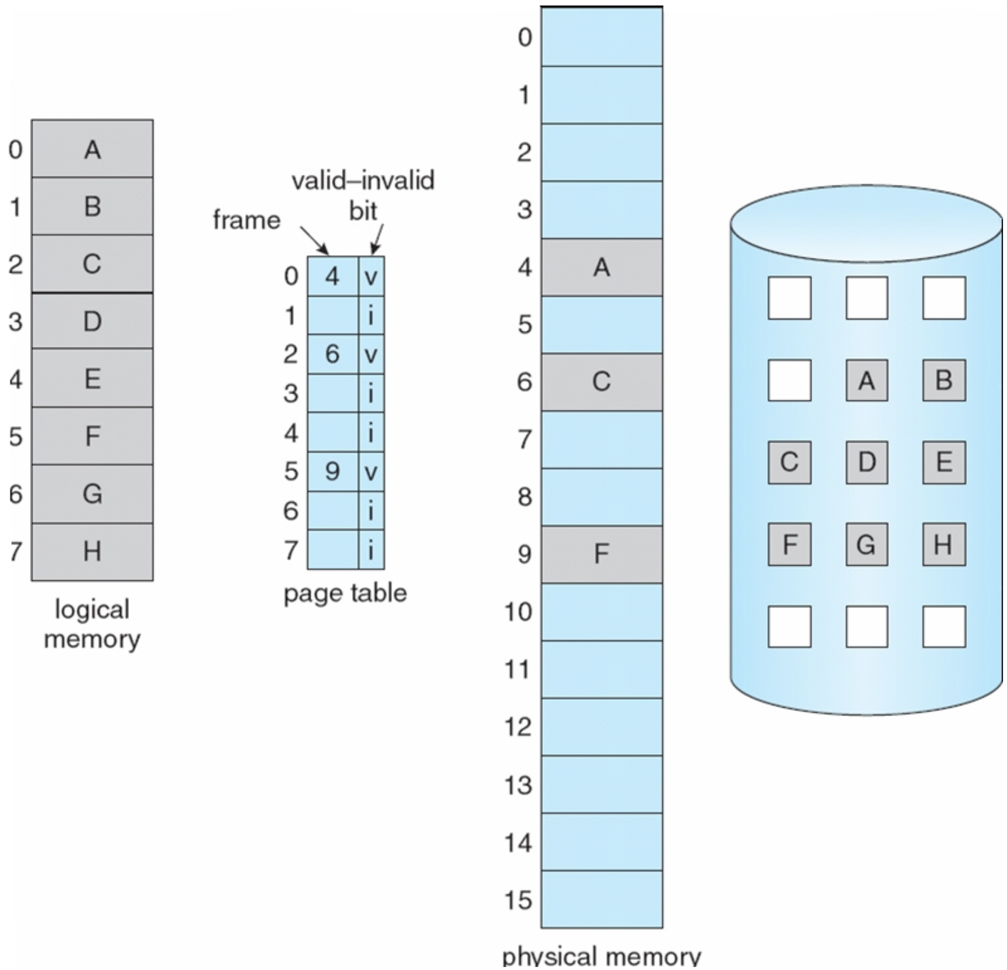
Valid-Invalid Bit

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



- During address translation, if valid–invalid bit in page table entry
is **I** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

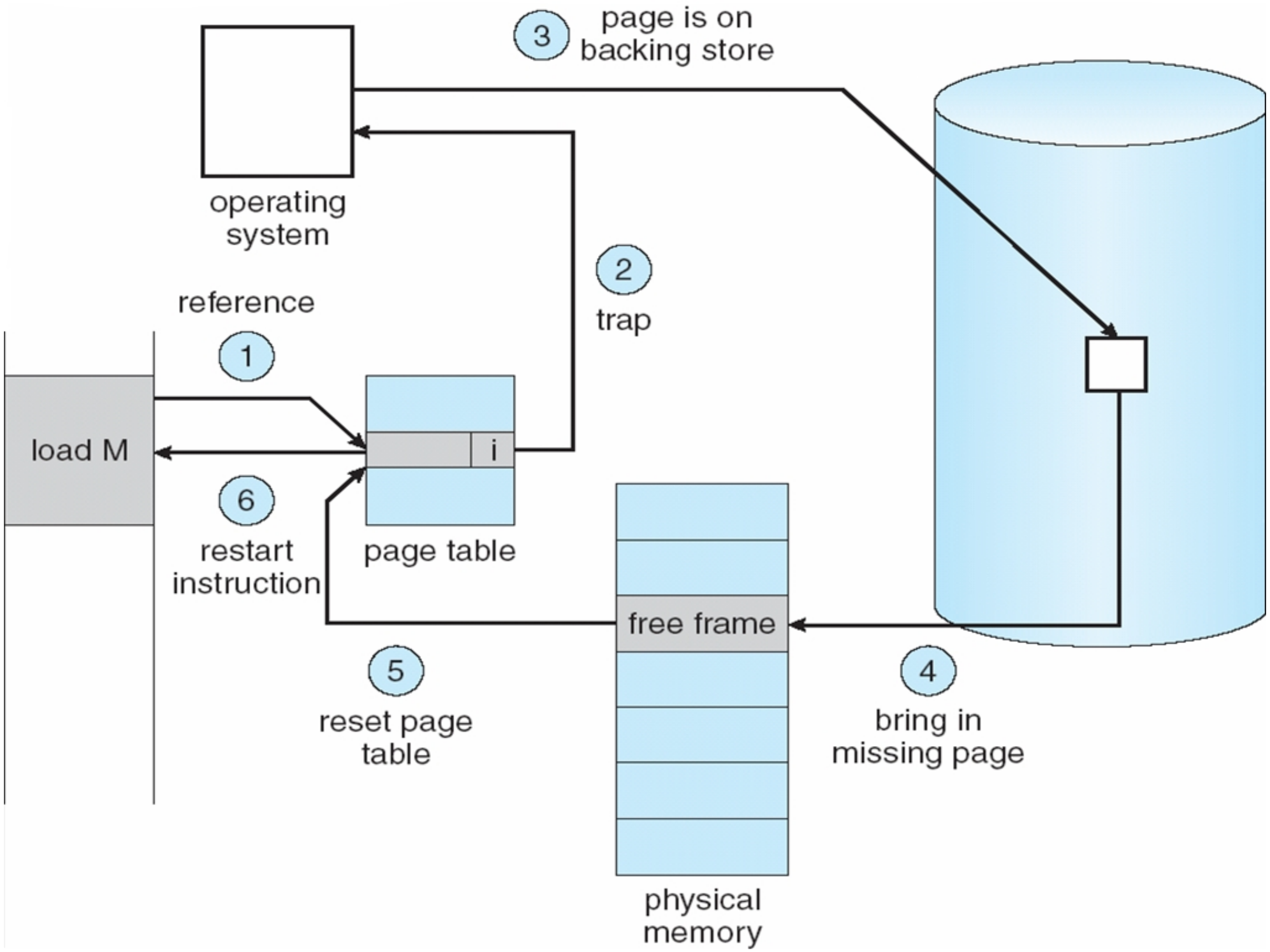
  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables

- Set validation bit = **v**

1. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



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) for virtual memory

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

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) for virtual memory

  EAT = (1 – p) x memory access

  + p (page fault overhead

  + swap page out (may be unnecessary)

  + swap page in

  + restart overhead

)

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) for virtual memory

  EAT = (1 – p) x memory access // EAT for paged memory

  // computed in Chap. 7

  + p (page fault overhead

  + swap page out (may be unnecessary)

  + swap page in

  + restart overhead

)

Chapter 7: Effective Access Time

Example

- Associative Lookup α = 0.05 time unit (= 10 ns)
- Assume time unit is 190 ns
- Hit ratio $\alpha = .99$

$$\text{EAT} = 190\text{ns}\ (2 + \varepsilon\alpha - \alpha) = 190\text{ns}\ (2 + 0.0495 - .99) =$$

$$= 201.305\text{ns} \approx 200\text{ns}$$

Demand Paging Example

- Assume the memory access time = 200 nanoseconds

Average page-fault service time in microseconds (µs)

- Interrupt ~ 0 µs
- Save context ~ 10 – 20 µs
- Recognize page fault ~ 1 µs
- Check validity of page reference and find disk addr ~ 5 µs
- Issue a read from the disk to a free frame = 0 µs
- **Wait in the waiting queue 0 µs (if the ready queue is nonempty)**
- While waiting, allocate CPU to another process p ~ 10 - 20 µs
- Accept END interrupt from DMA ~ 0 µs

Demand Paging Example 2

- Assume the memory access time = 200 nanoseconds

- Average page-fault service time (cont'd)
  - Save context of process p ~ 10 – 20 µs
  - Update page table ~ 2 µs
  - **Wait in the ready queue 0 µs**
  - Restore the context ~ 10 – 20 µs
- **Total** ~ 48 – 88 µs, say, **70 µs** on average

Demand Paging Example 3

- Assume the memory access time = 200 nanoseconds

- Average page-fault service time = 70 µs = 70,000 ns

- EAT = (1 – p) x 200 + p x 70,000

  = 200 + p x 69,800

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

  EAT = 269.8 nanoseconds ≈ 270ns .

  This is a slowdown by 35%

If one access out of 10,000 causes a page fault, then

EAT = 207 nanoseconds.

This is a slowdown by 3.5%

Demand Paging Example 3

- If one access out of 10,000 causes a page fault, then

  EAT = 207 nanoseconds.

  This is a slowdown by 3.5%

Unrealistic value from the textbook:

$$220 > 200 + 7{,}999{,}800 \times p,$$
$$20 > 7{,}999{,}800 \times p,$$
$$p < 0.0000025.$$

lowdown due to paging at a reaso

memory access out of 399,990 to

Demand Paging Example 3

- If one access out of 10,000 causes a page fault, then

  EAT = 207 nanoseconds.

  This is a slowdown by 3.5%

- The question is how can one assure such a low page-fault rate?

Demand Paging Example 3

- If one access out of 10,000 causes a page fault, then

  EAT = 207 nanoseconds.

  This is a slowdown by 3.5%

- The question is how can one assure such a low page-fault rate?

- Page replacement algorithm is the key to an answer.

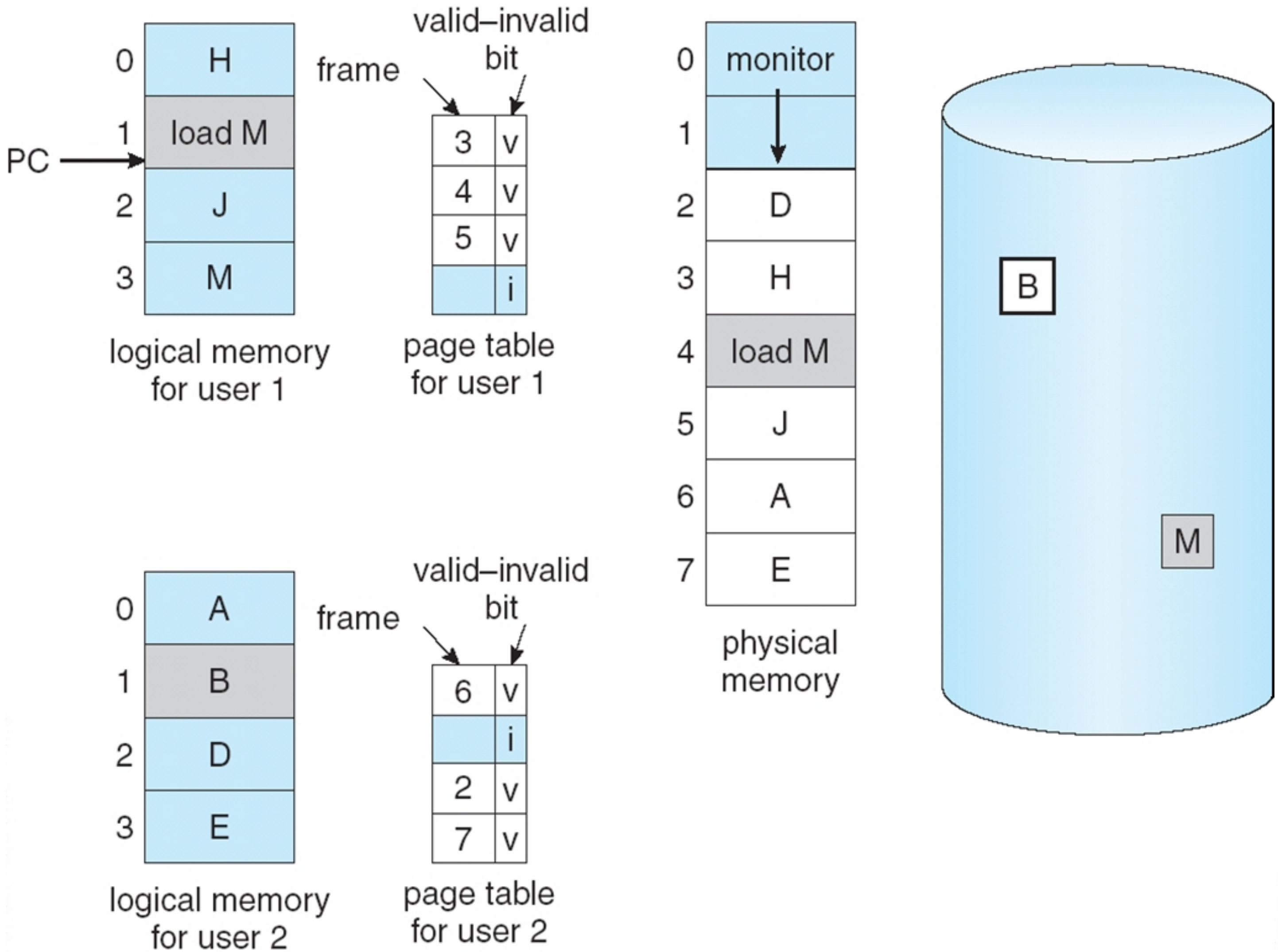
What happens if there is no free frame?

- 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

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include 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

Need For Page Replacement



Basic Page Replacement

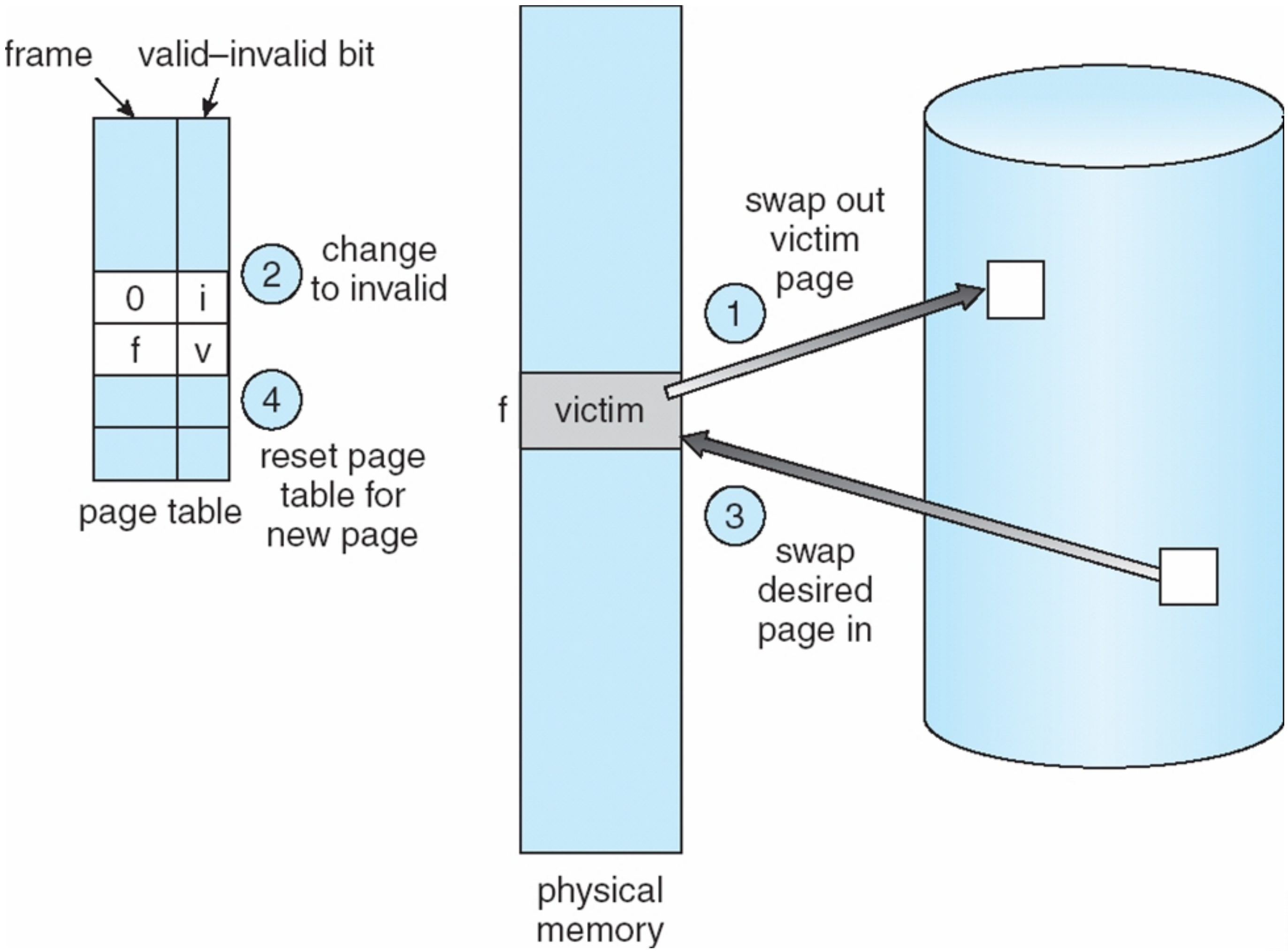
1) Find the location of the desired page on disk

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

- Want lowest page-fault rate
- Example string of **memory** references: (assuming page length 100):
- 0100, 0101, 0232, 0311, 0404, 0100, 0102, 0103, 0233, 0252, 0532, 0104, 0100, 0101, 0233, 0312, 0405, 0532
- Evaluate algorithm by running it on a particular string of **page** references with consecutive duplicate references collapsed to single ones (reference string) and computing the number of page faults on that string

Page Replacement Algorithms 2

- In PowerPoint examples, the reference string is

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

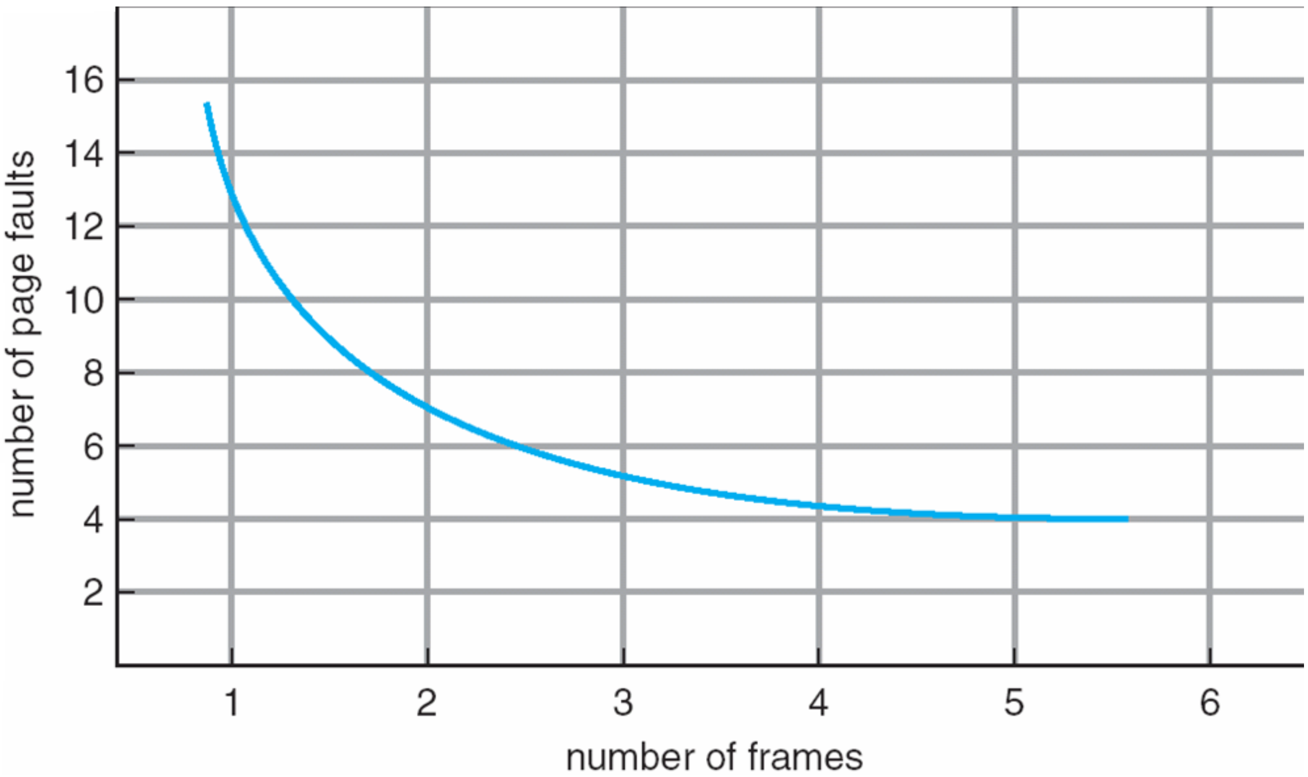
for a total of 5 pages and minimum 5 page faults.

- In textbook examples, the reference string is

  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a total of 6 pages and minimum 6 page faults.

Graph of Page Faults Versus The Number of Frames



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
3	3	2	4

9 page faults

Net faults = 9 – 5 = 4.

- 4 frames

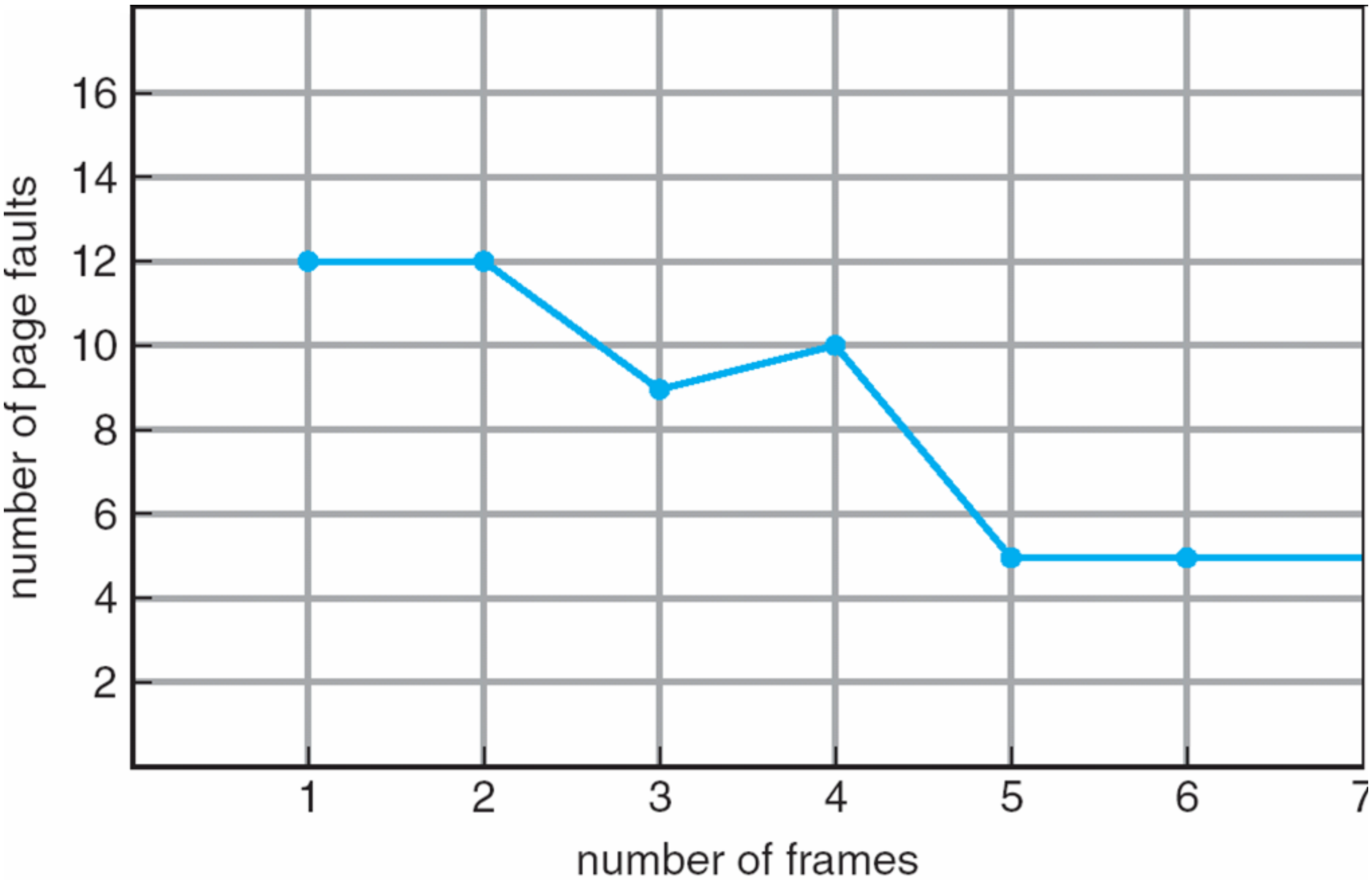
1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

10 page faults

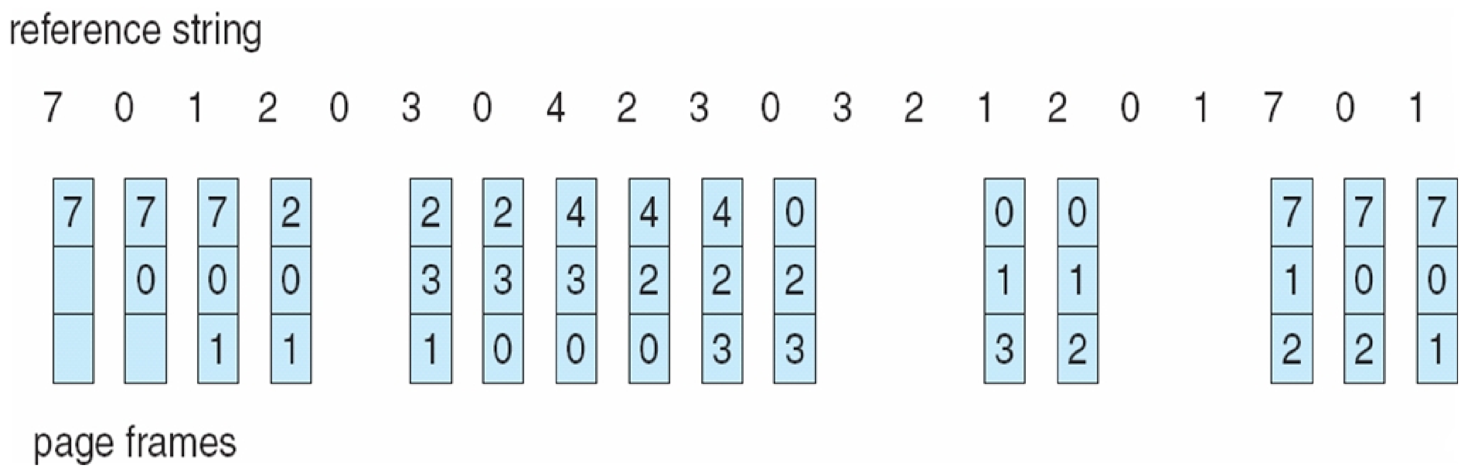
Net faults = 10 – 5 = 5.

- **Belady's Anomaly: more frames ⇒ more page faults**

FIFO Illustrating Belady's Anomaly



FIFO Page Replacement

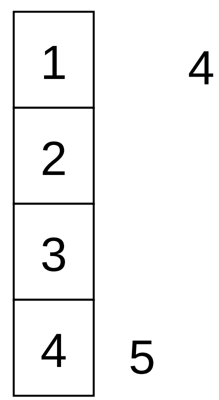


Net faults = 15 – **6** = 9.

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

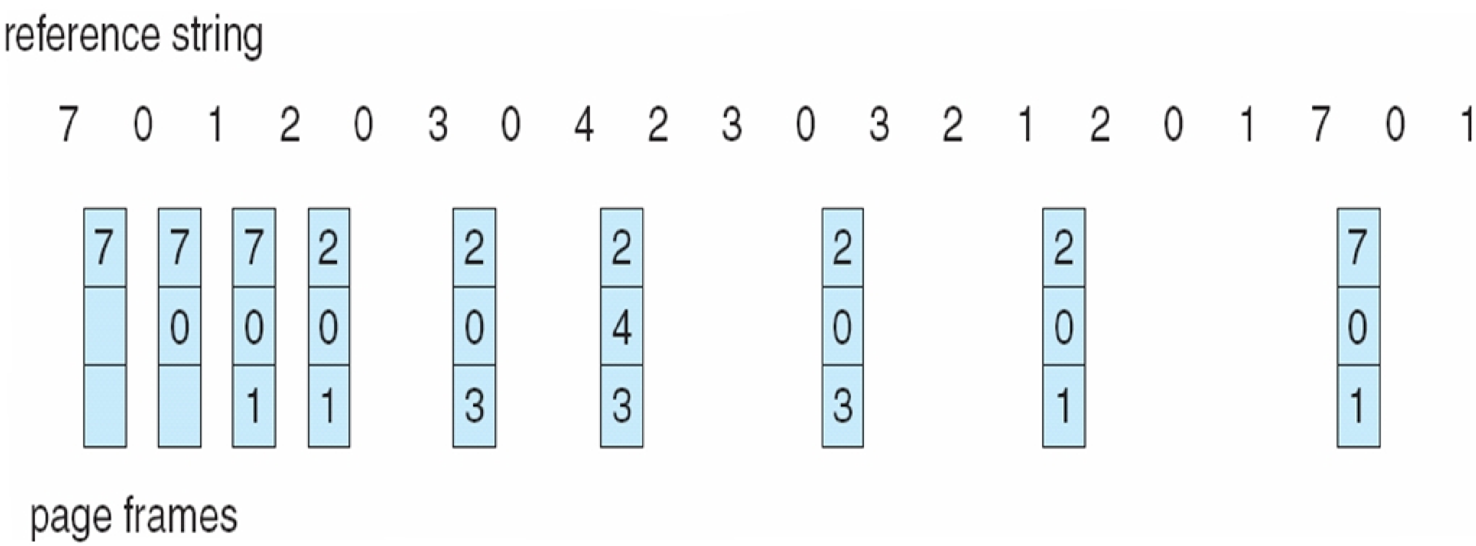


6 page faults

Net faults = 6 – 5 = 1.

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

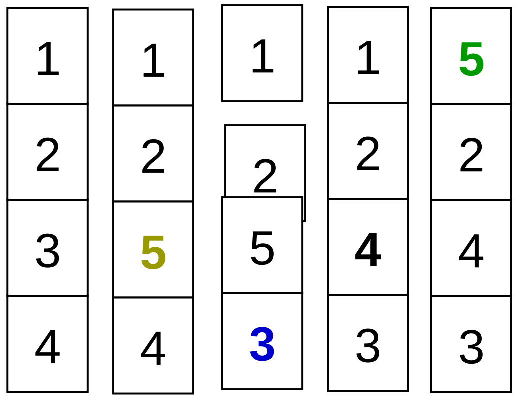
Optimal Page Replacement



Net faults = 9 – **6** = 3.

Least Recently Used (LRU) Algorithm

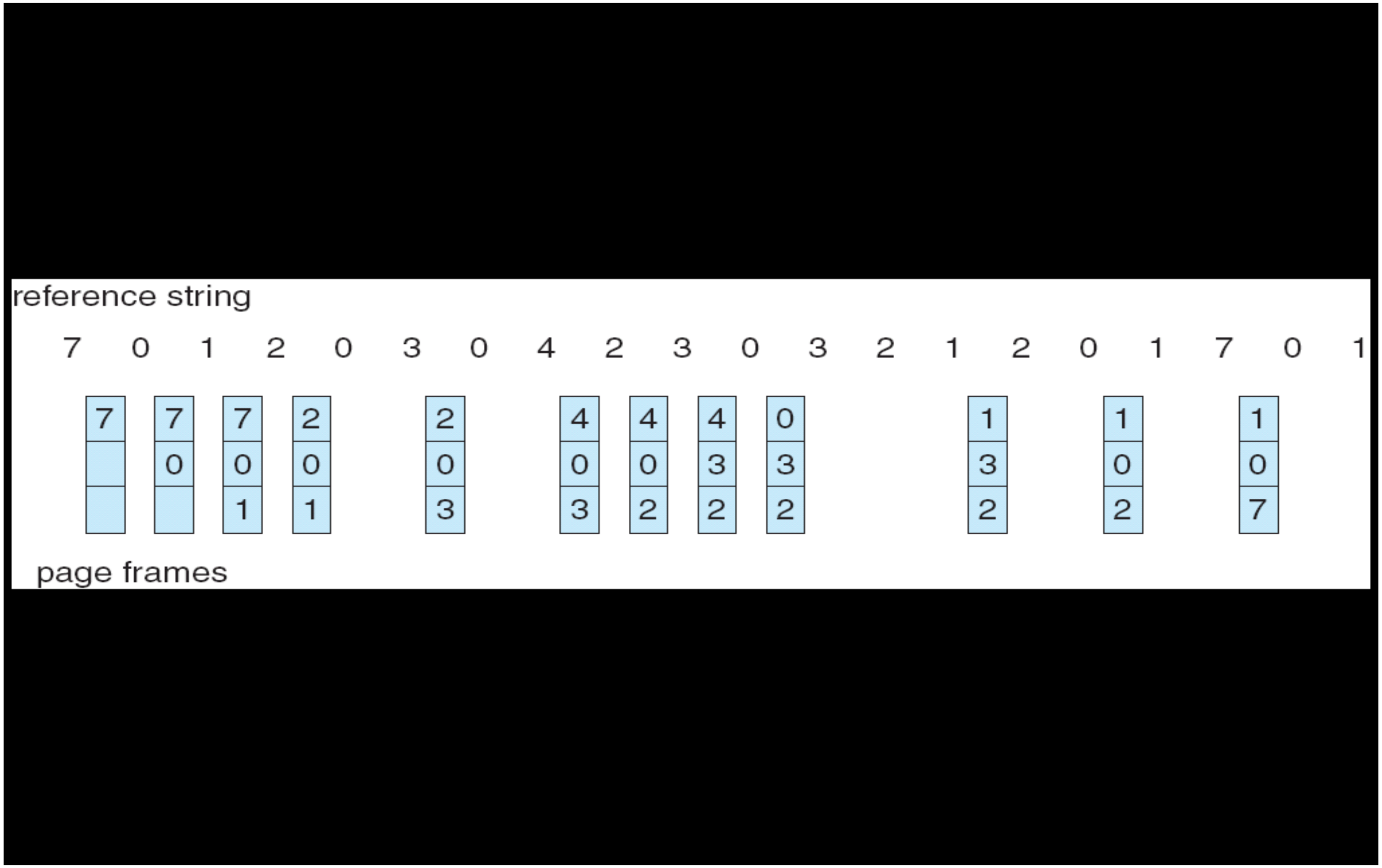
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



Net faults = 8 – 5 = 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

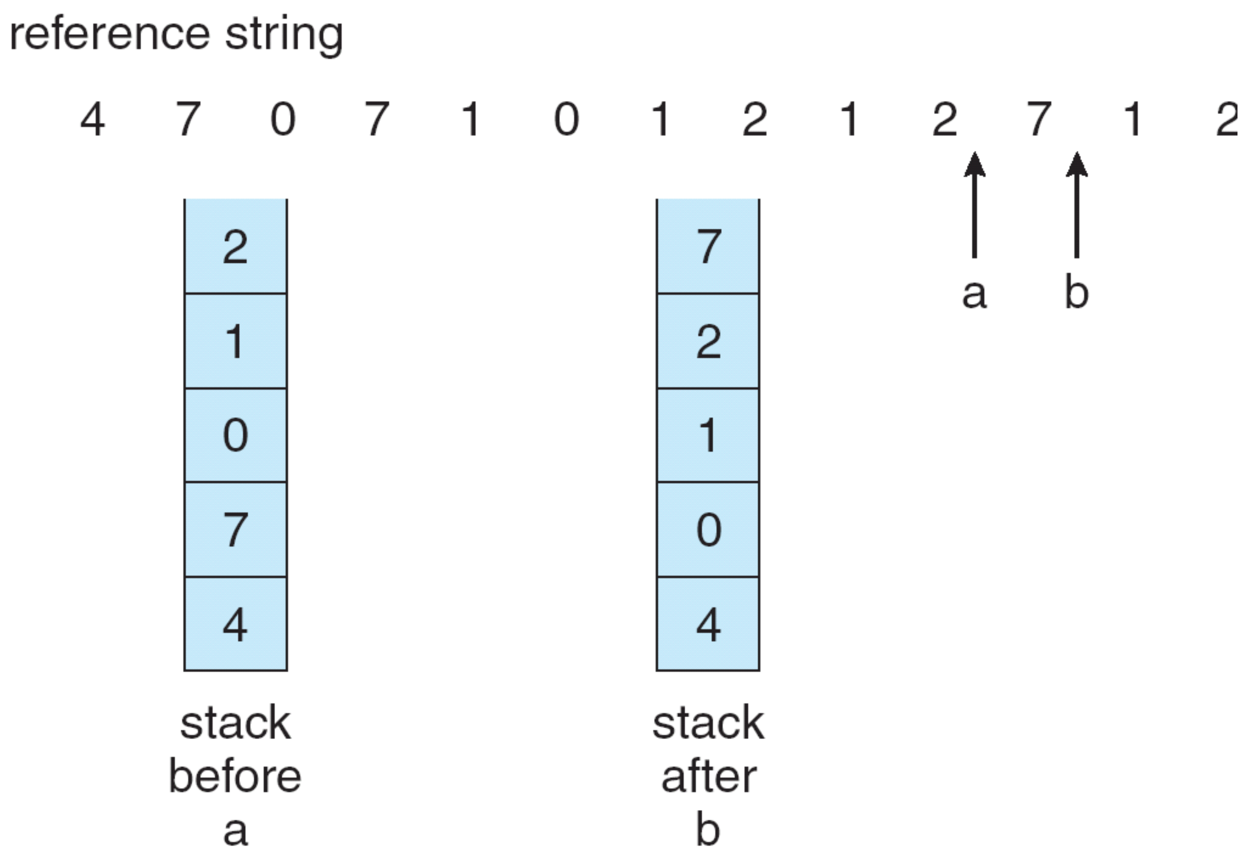


Net faults = 12 – **6** = 6.

LRU Algorithm (Cont.)

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

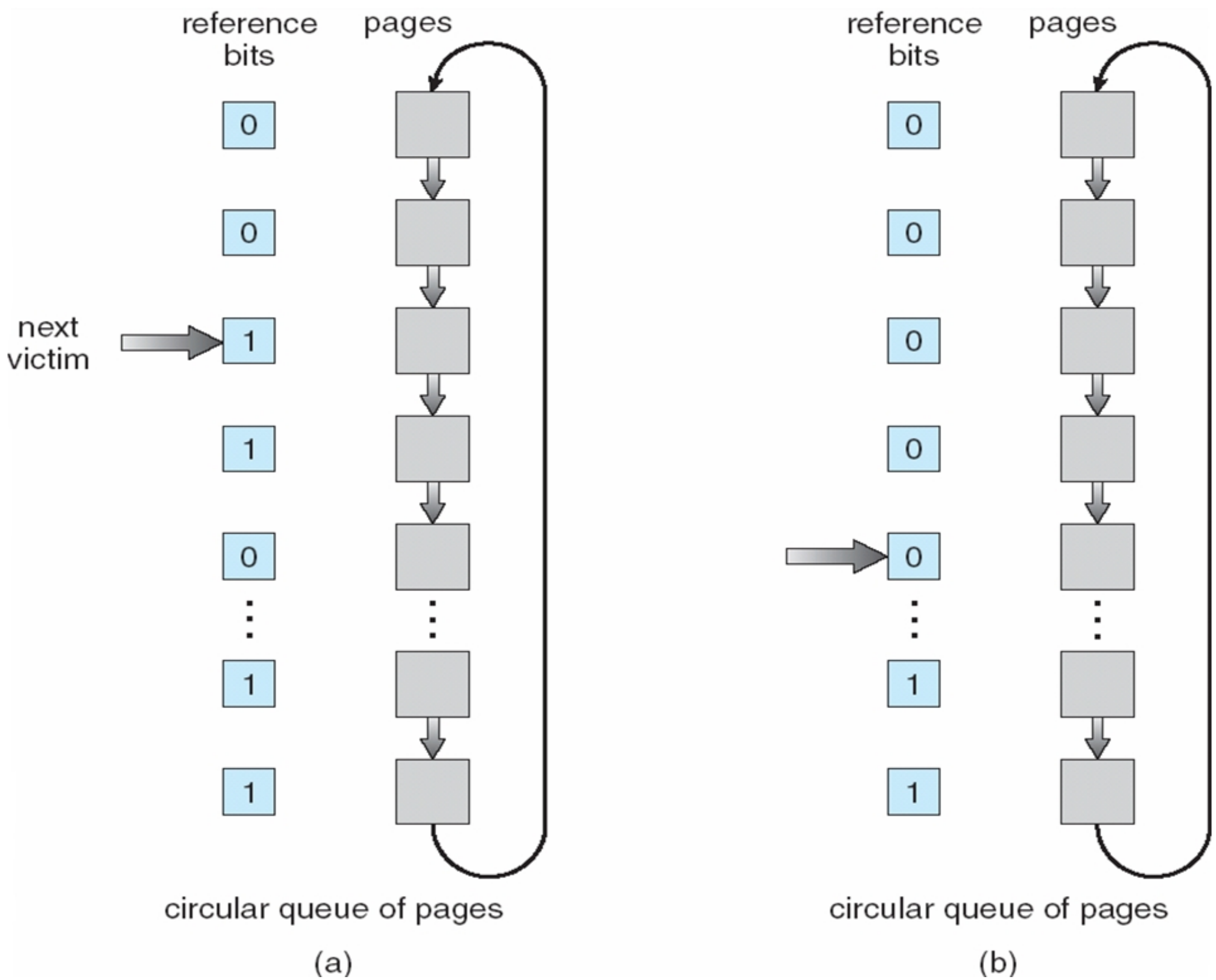
Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - 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

Second-Chance (clock) Page-Replacement Algorithm



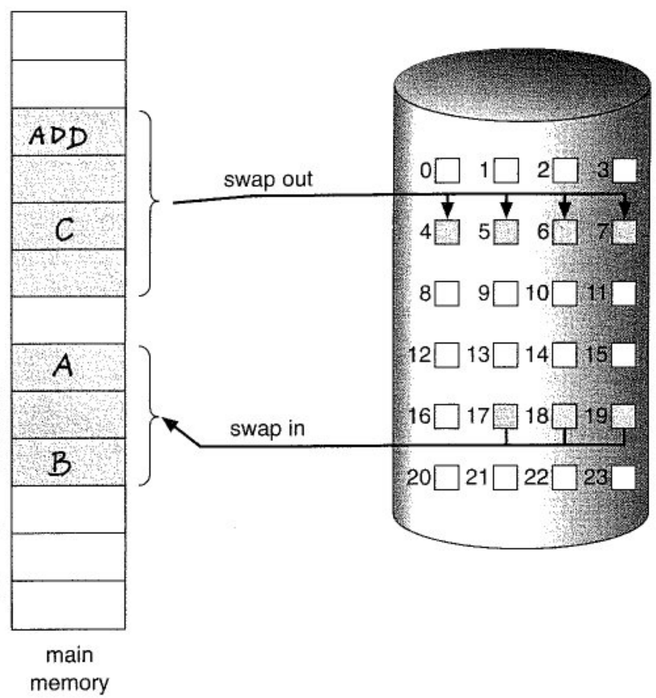
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.



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

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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

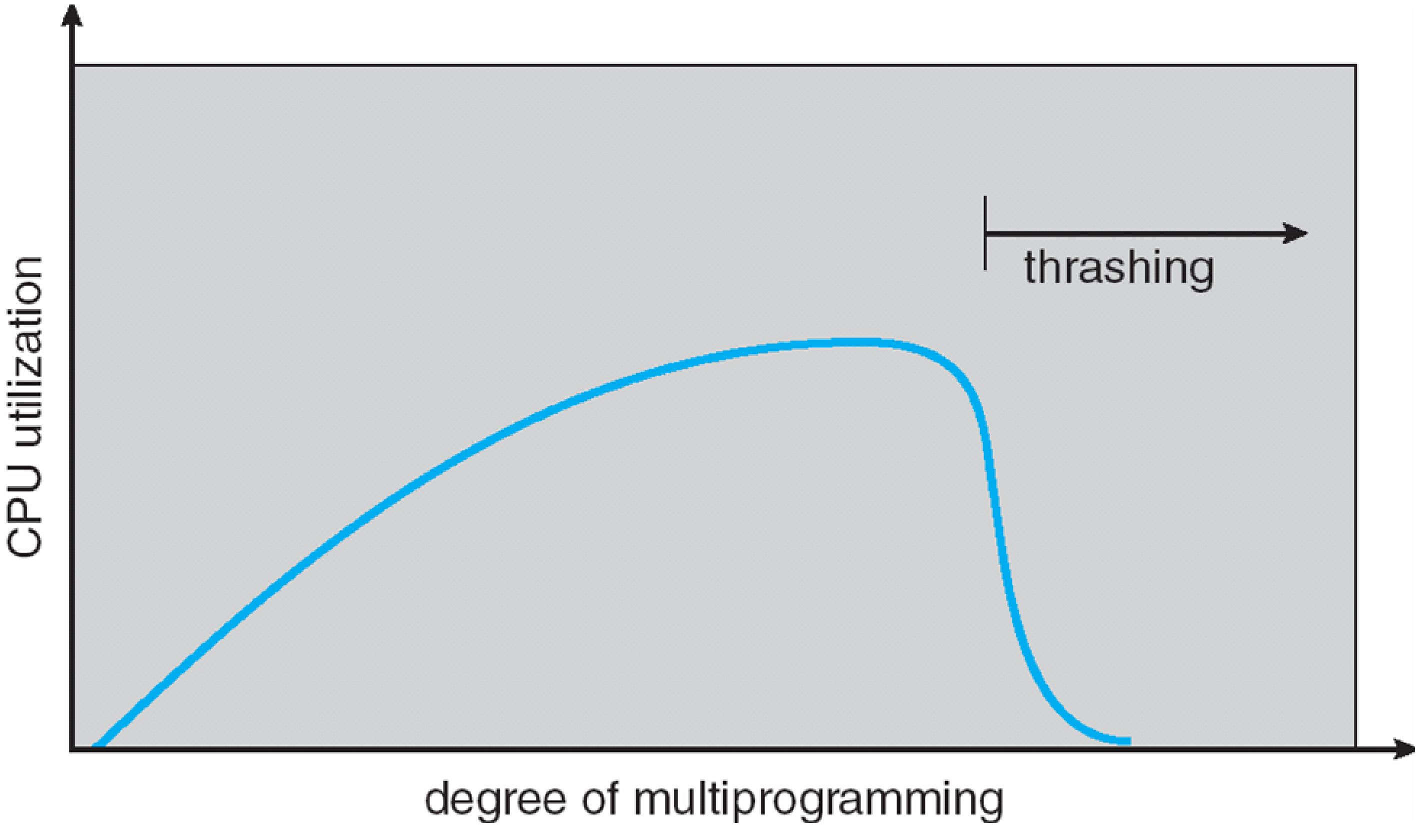
Global vs. Local Allocation

- **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
  - 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.)



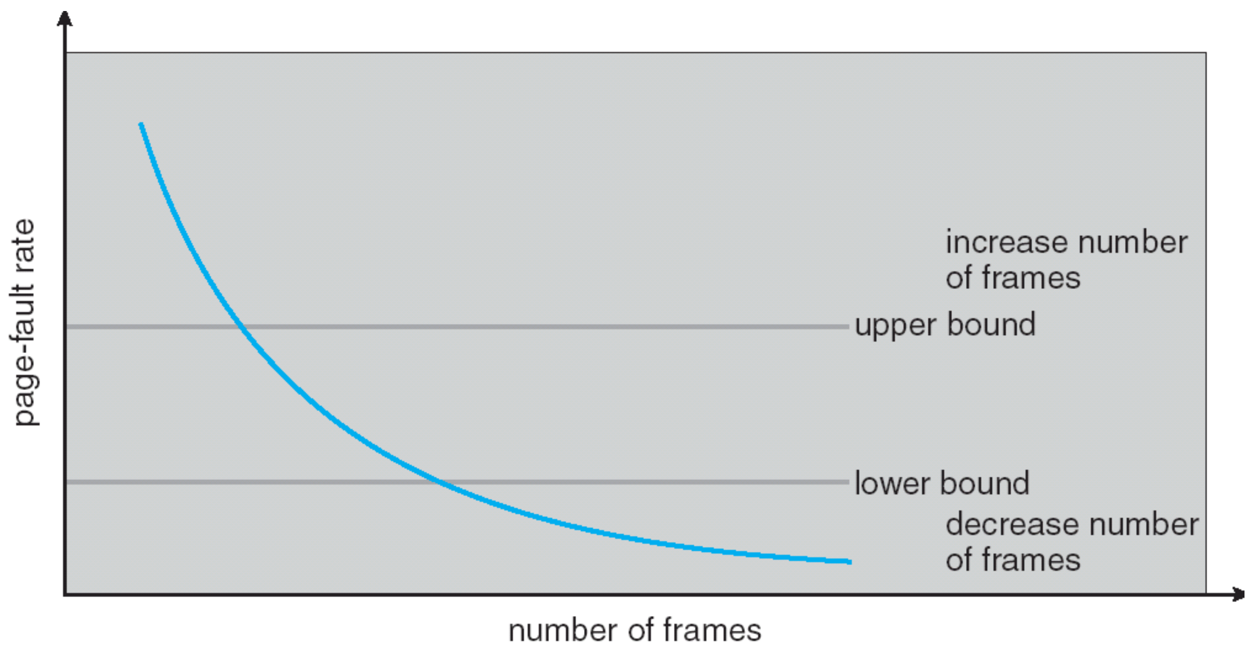
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size

Page-Fault Frequency Scheme

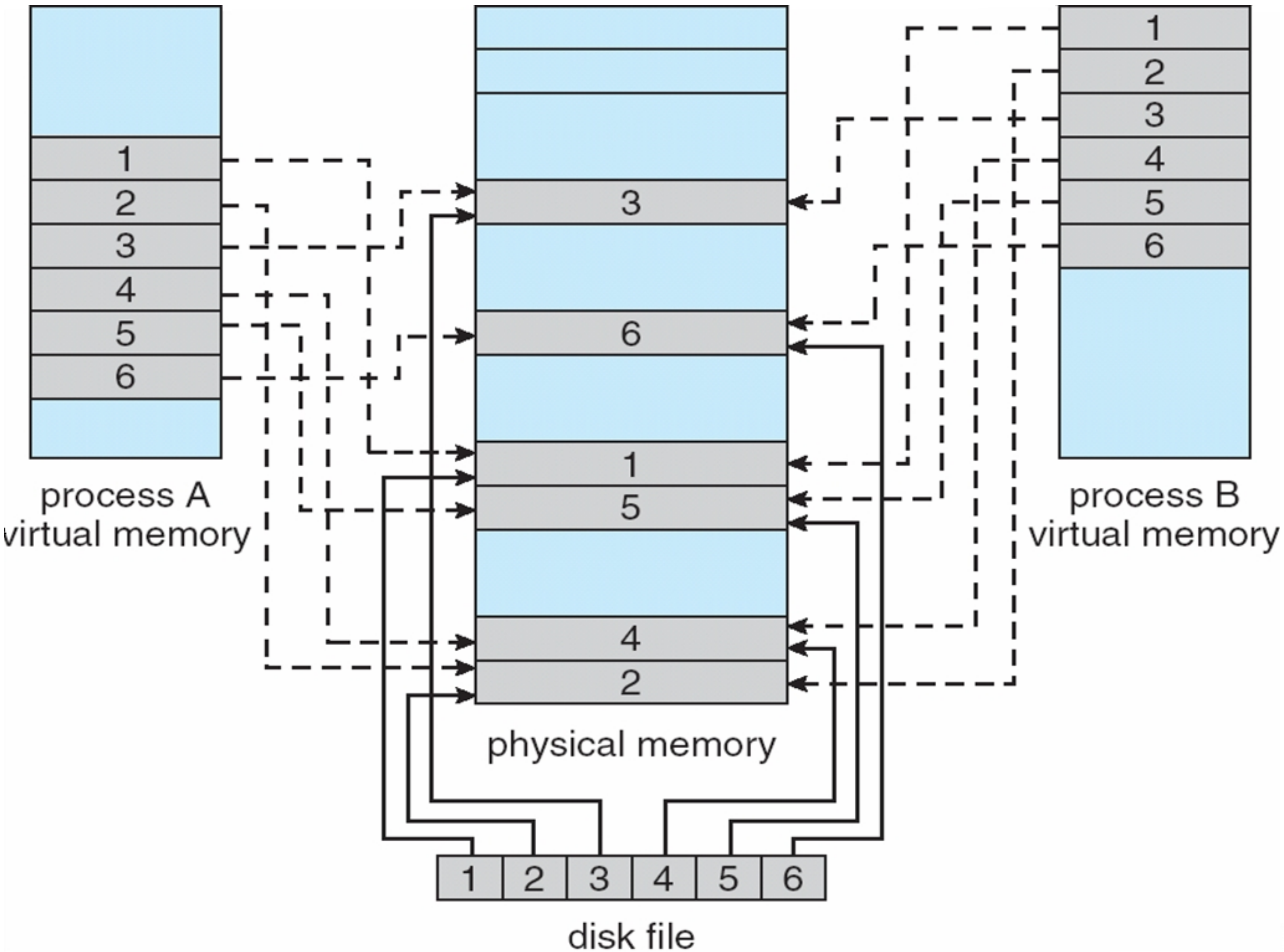
- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Memory-Mapped Files

- Memory-mapped 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

Memory Mapped Files



Other Issues – Page Size

- Page size selection must take into consideration:
  - 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
- 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
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

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

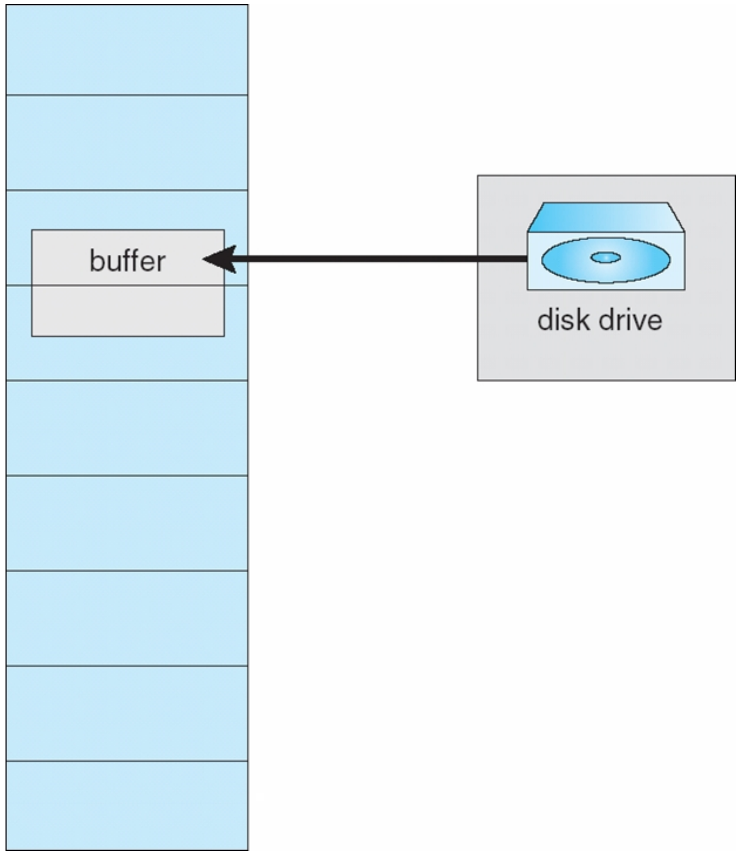
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

Reason Why Frames Used For I/O Must Be In Memory



End of Chapter 8