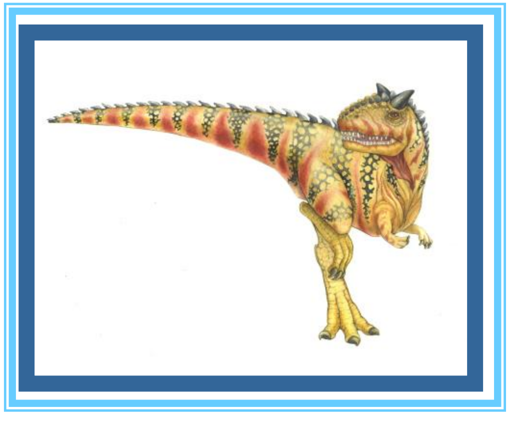
Chapter 9: Virtual-Memory Management



Chapter 9: Virtual-Memory Management

- 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**
- 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

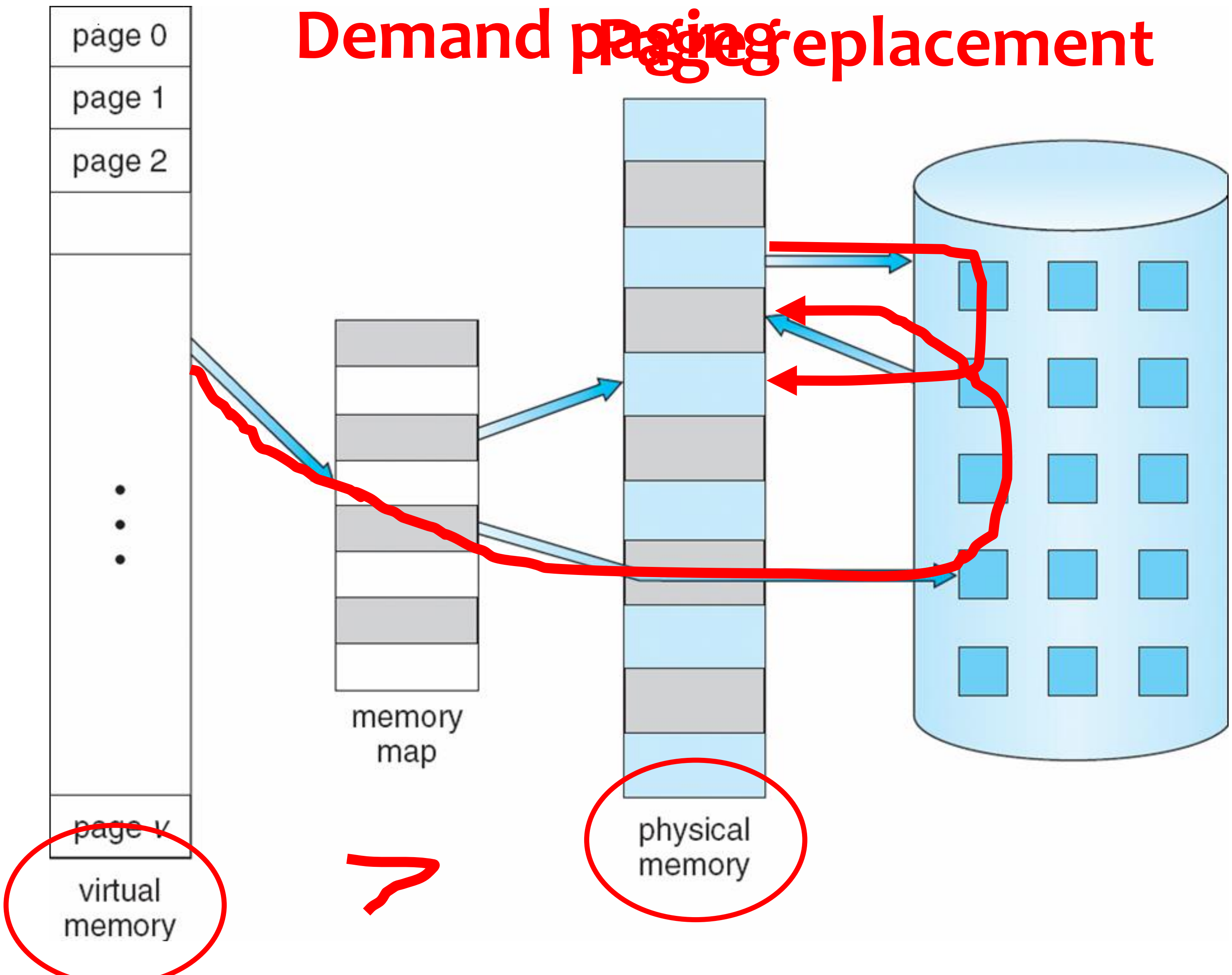
Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

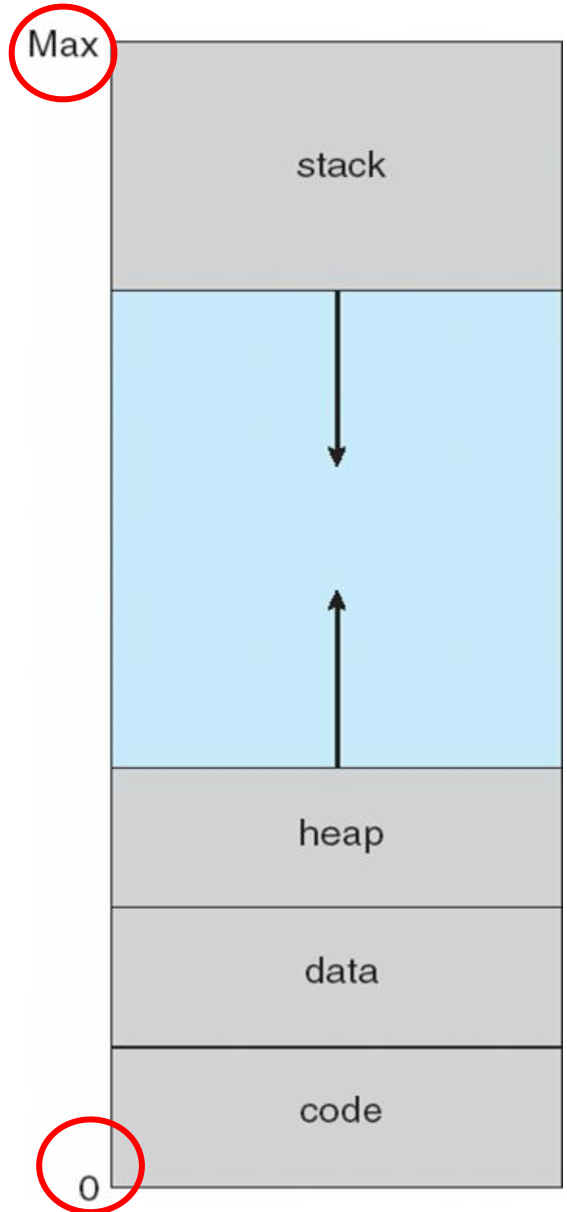
Virtual Memory > Physical Memory

Demand paging

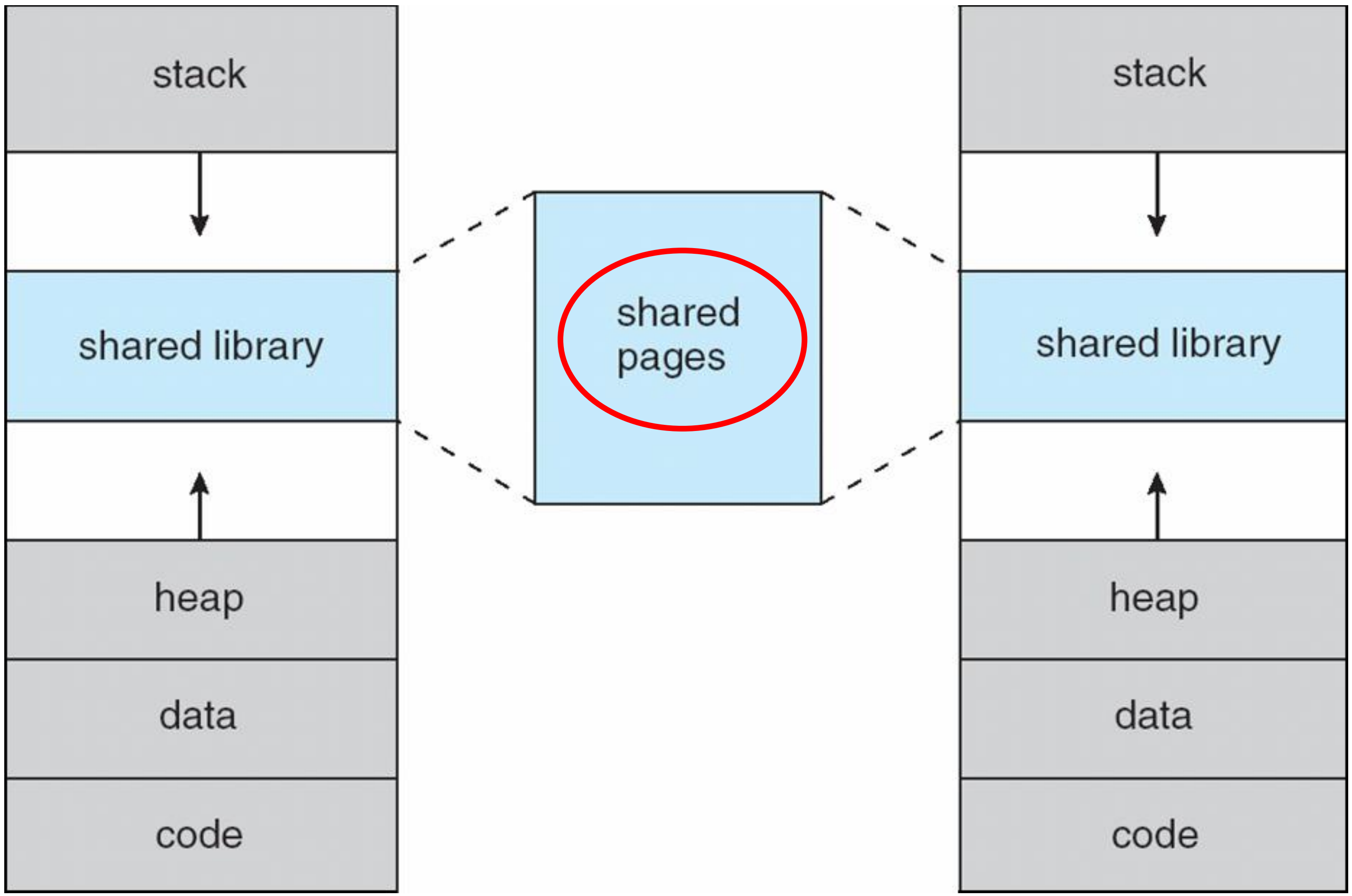
Page replacement



Virtual-address Space



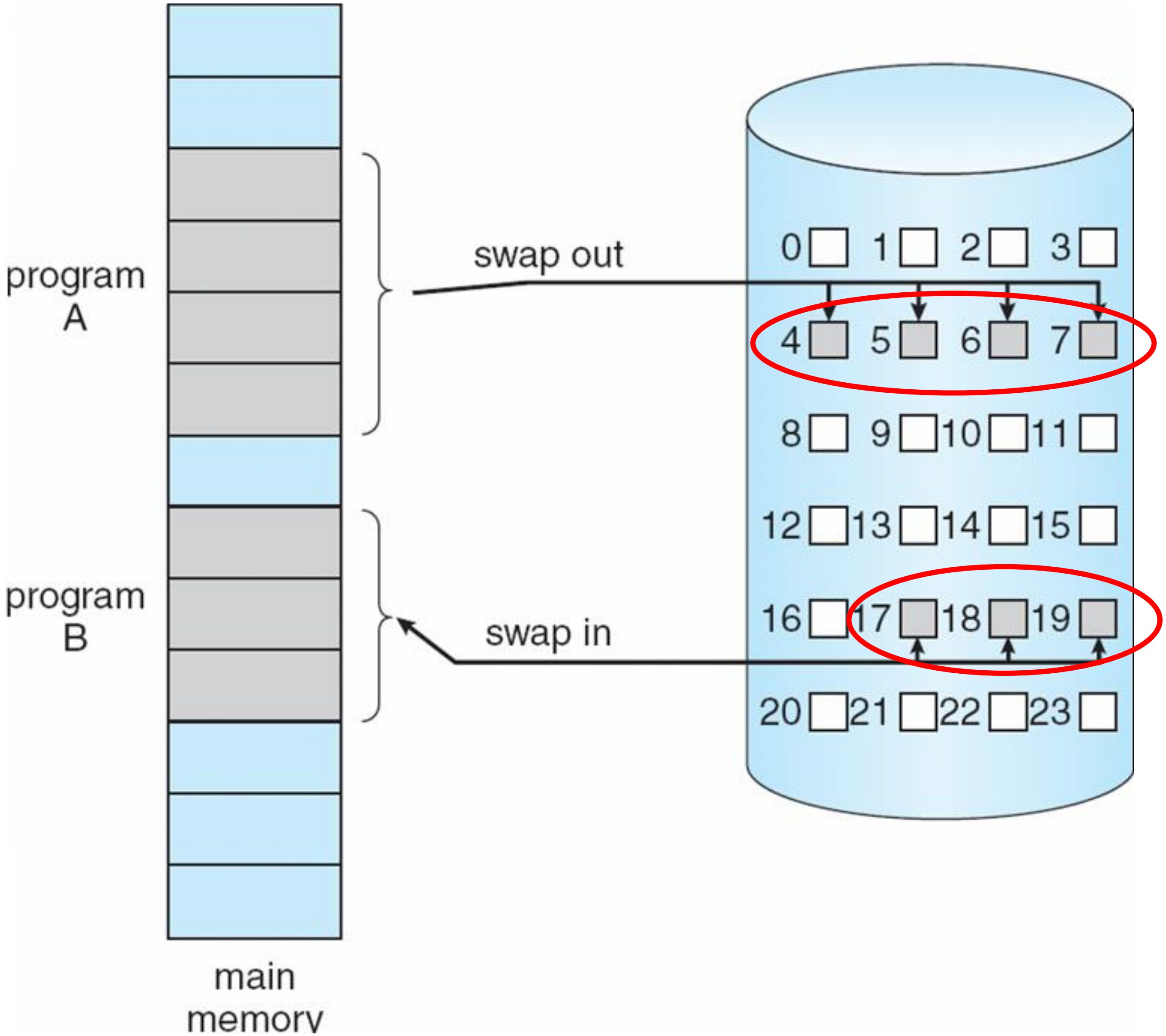
Shared Library Using Virtual Memory



Demand Paging

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

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

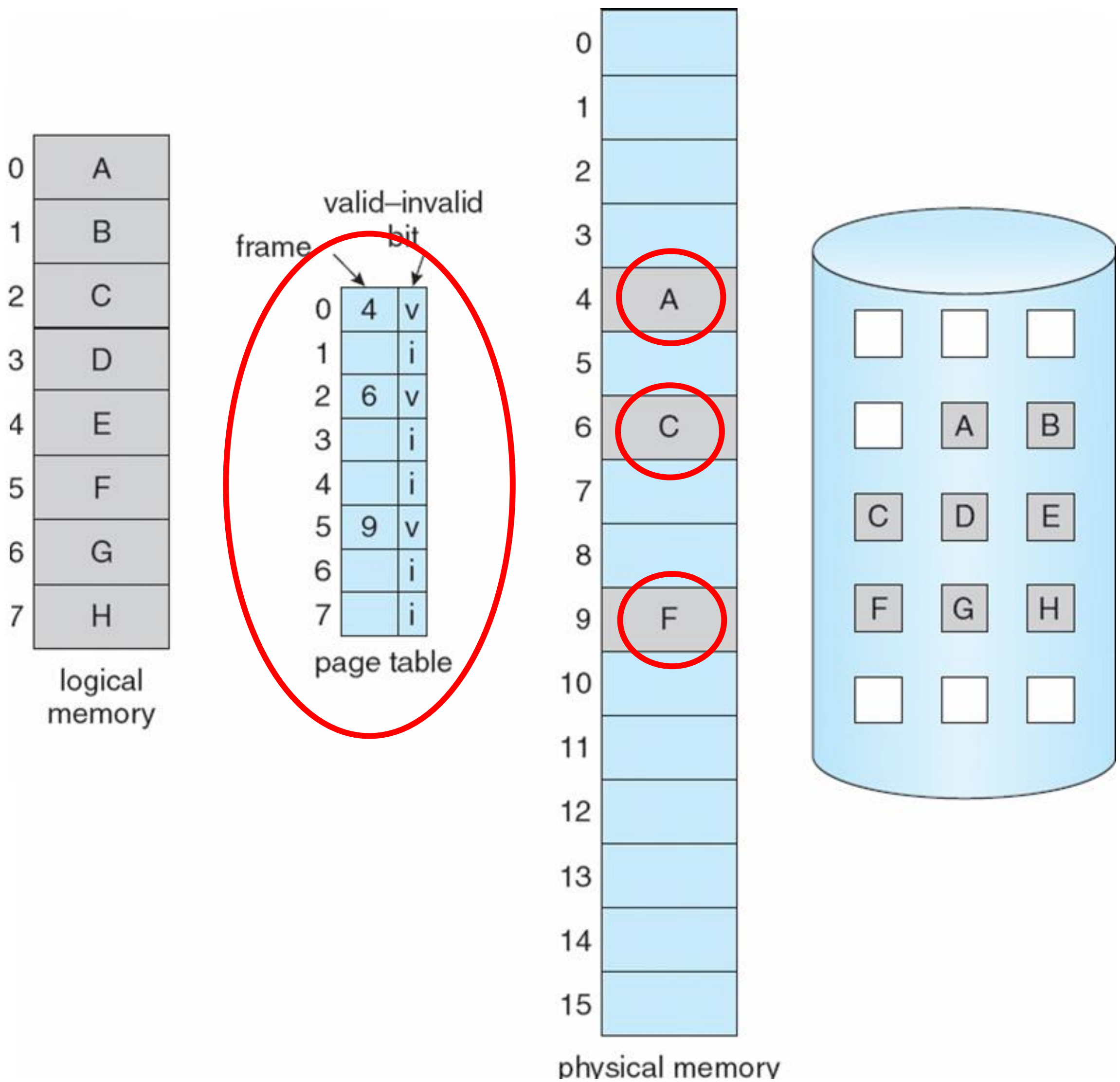
- With each page table entry a valid–invalid bit is associated
(**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- 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

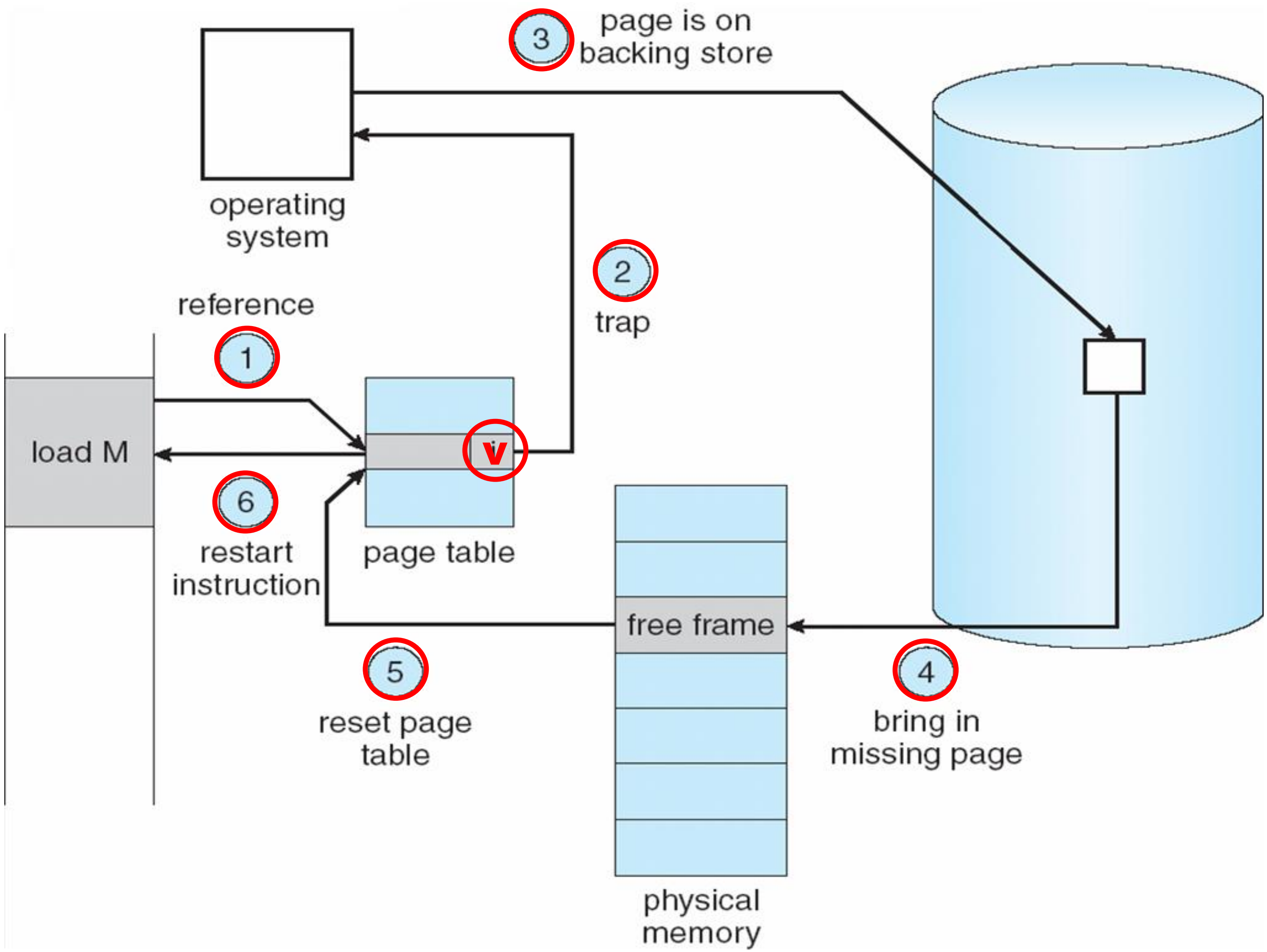
1. Operating system looks at another table to decide:

   Invalid reference $\Rightarrow$ abort

   Just not in memory

2. Get empty frame from physical memory
3. Swap page into frame from disk
4. Reset tables
5. Set validation bit = v
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging

Page Fault Rate $0 \leq p \leq 1.0$

if $p = 0$ no page faults

if $p = 1$, every reference is a fault

Effective Access Time (EAT)

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

Demand Paging Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p) x 200 + p x 8,000,000

= 200 + p x 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!!**

Process Creation

Virtual memory allows other benefits during process creation:

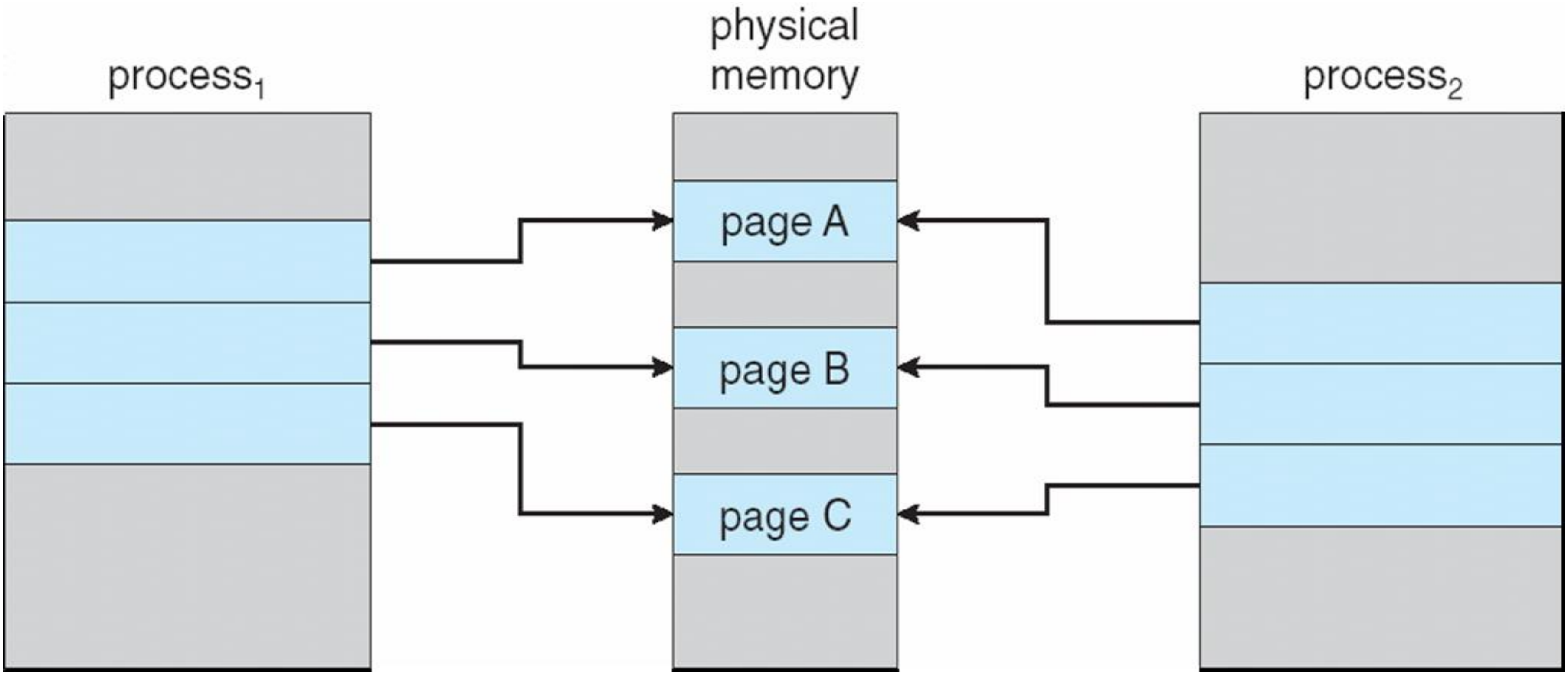
- Copy-on-Write

- Memory-Mapped Files (later)

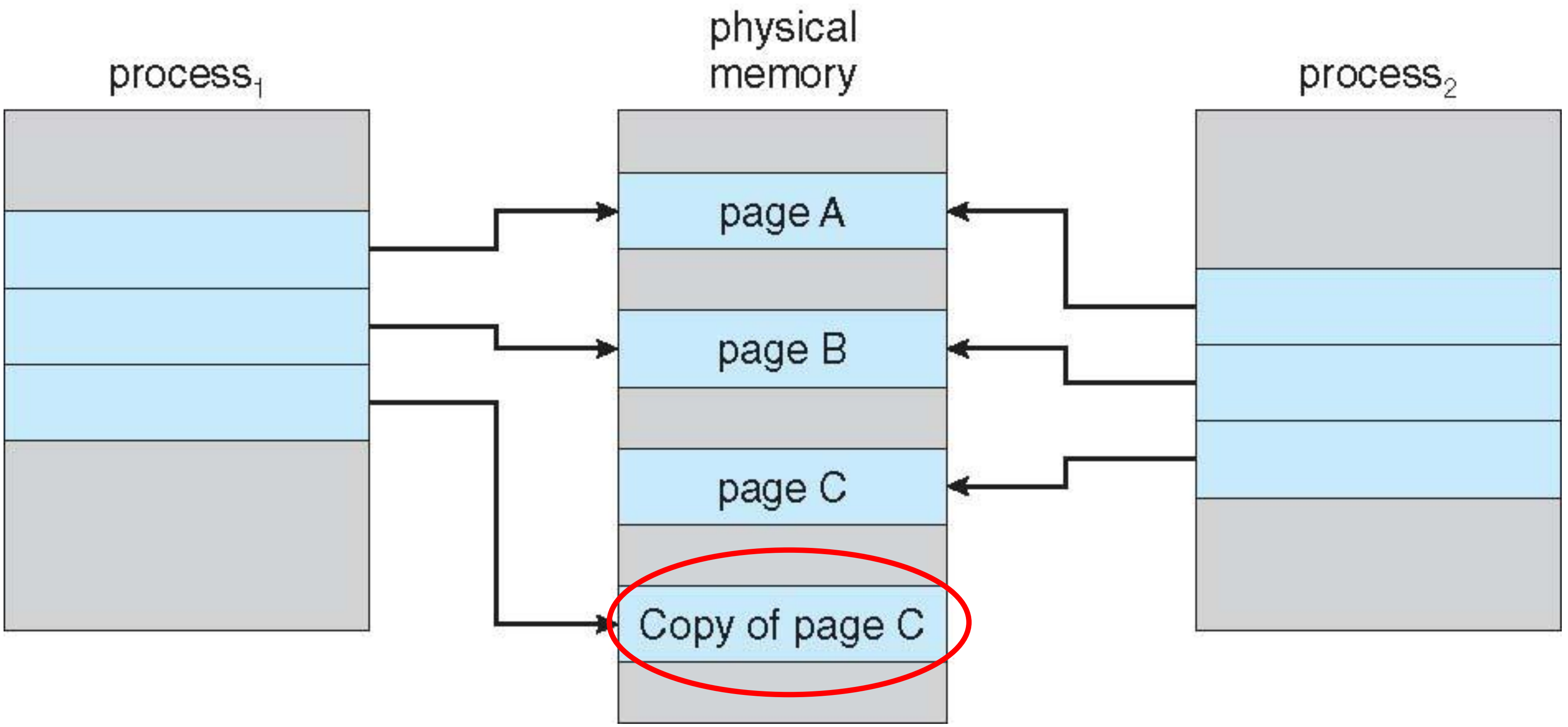
Copy-on-Write

- 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
- Free pages are allocated from a pool of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



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

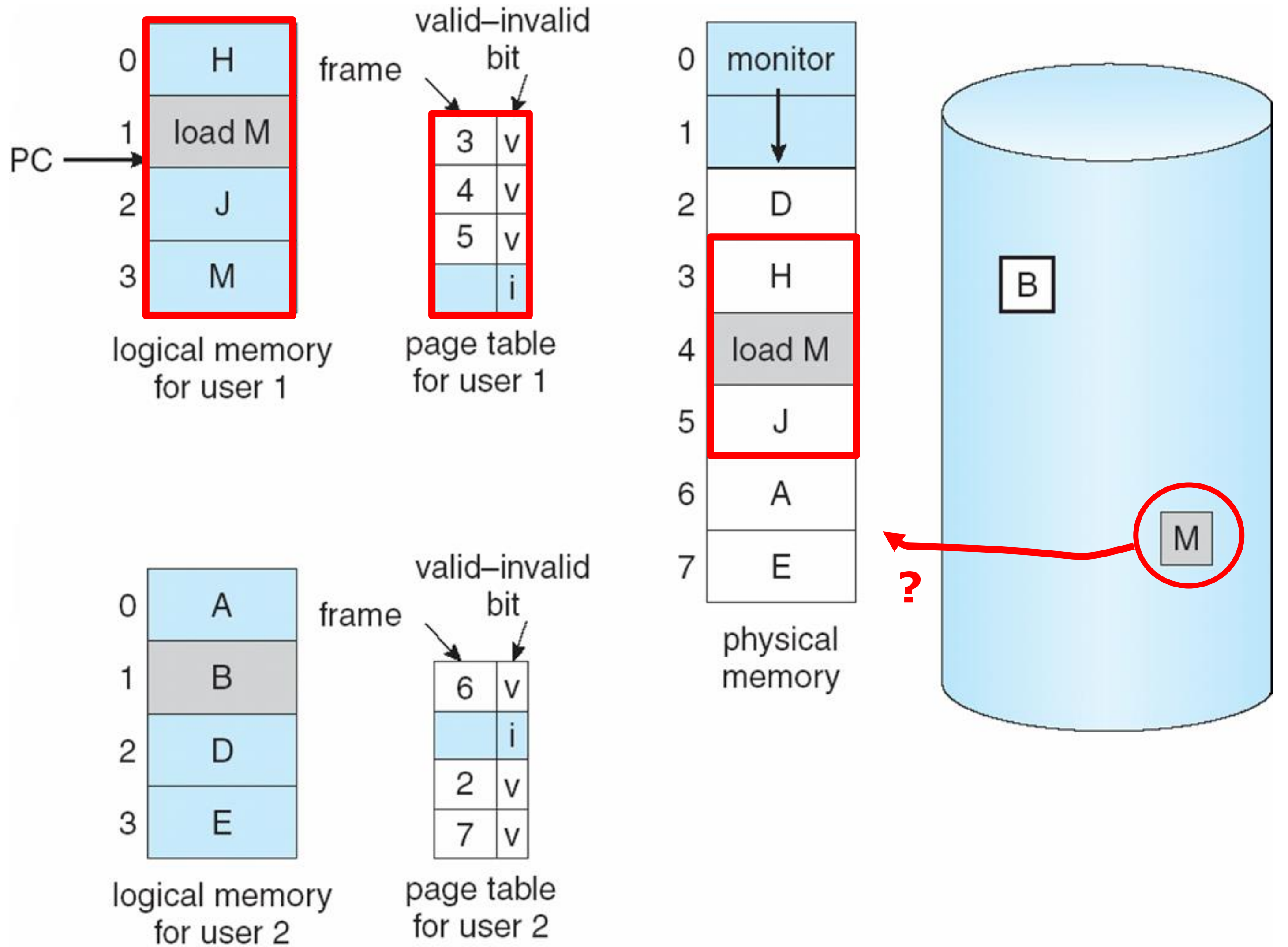
Page Replacement

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

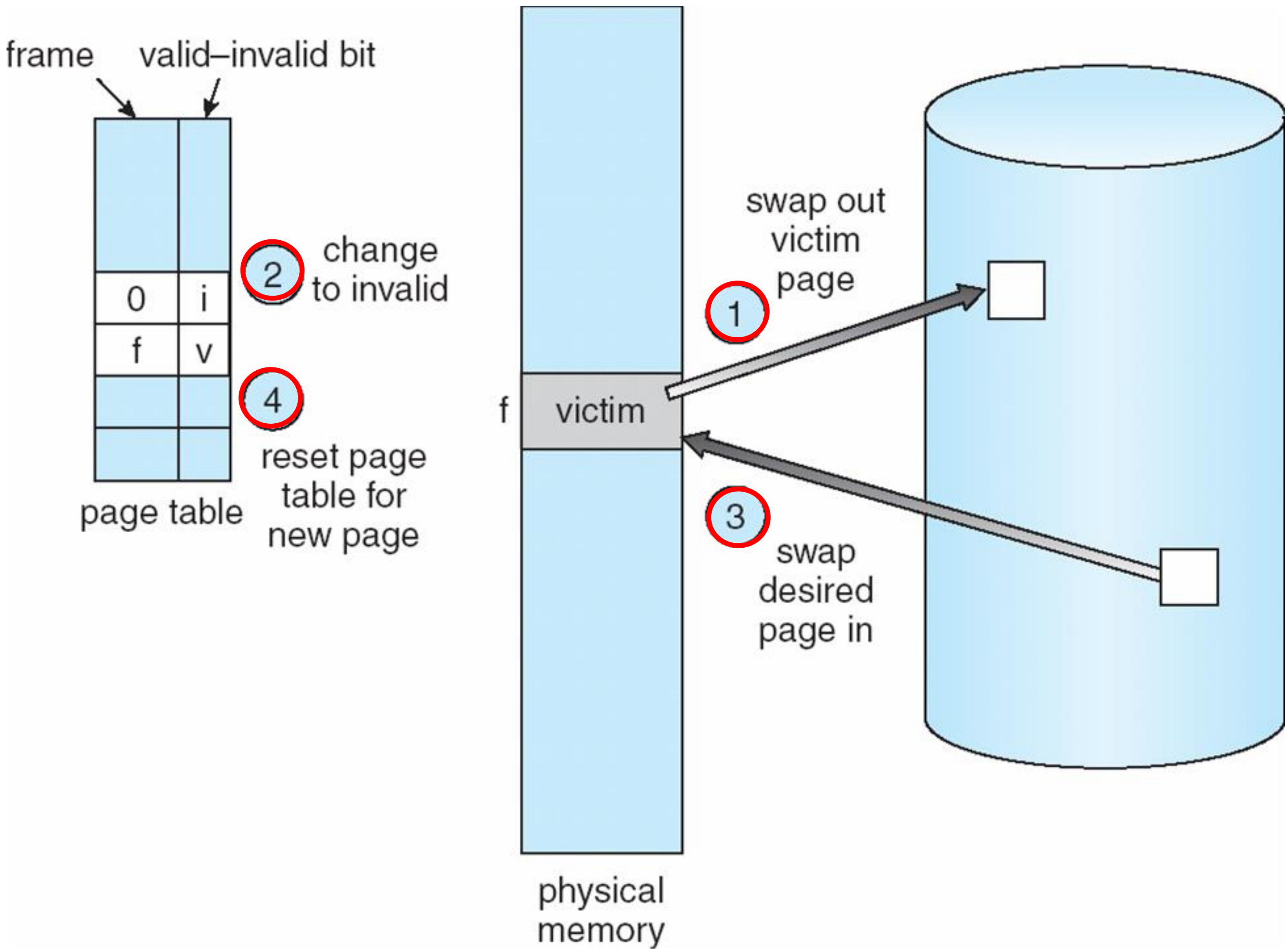
Need For Page Replacement



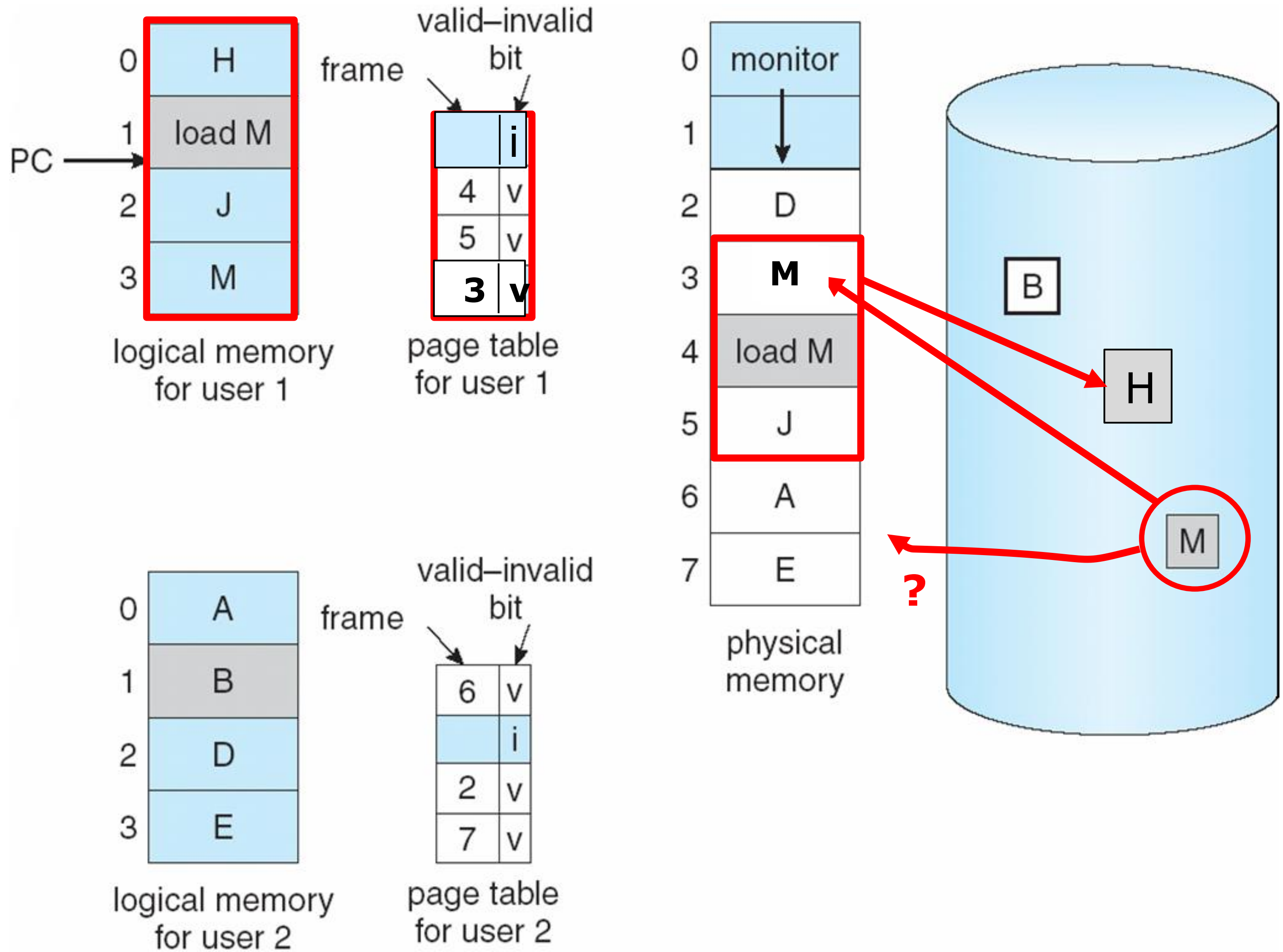
Basic Page Replacement

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 Example

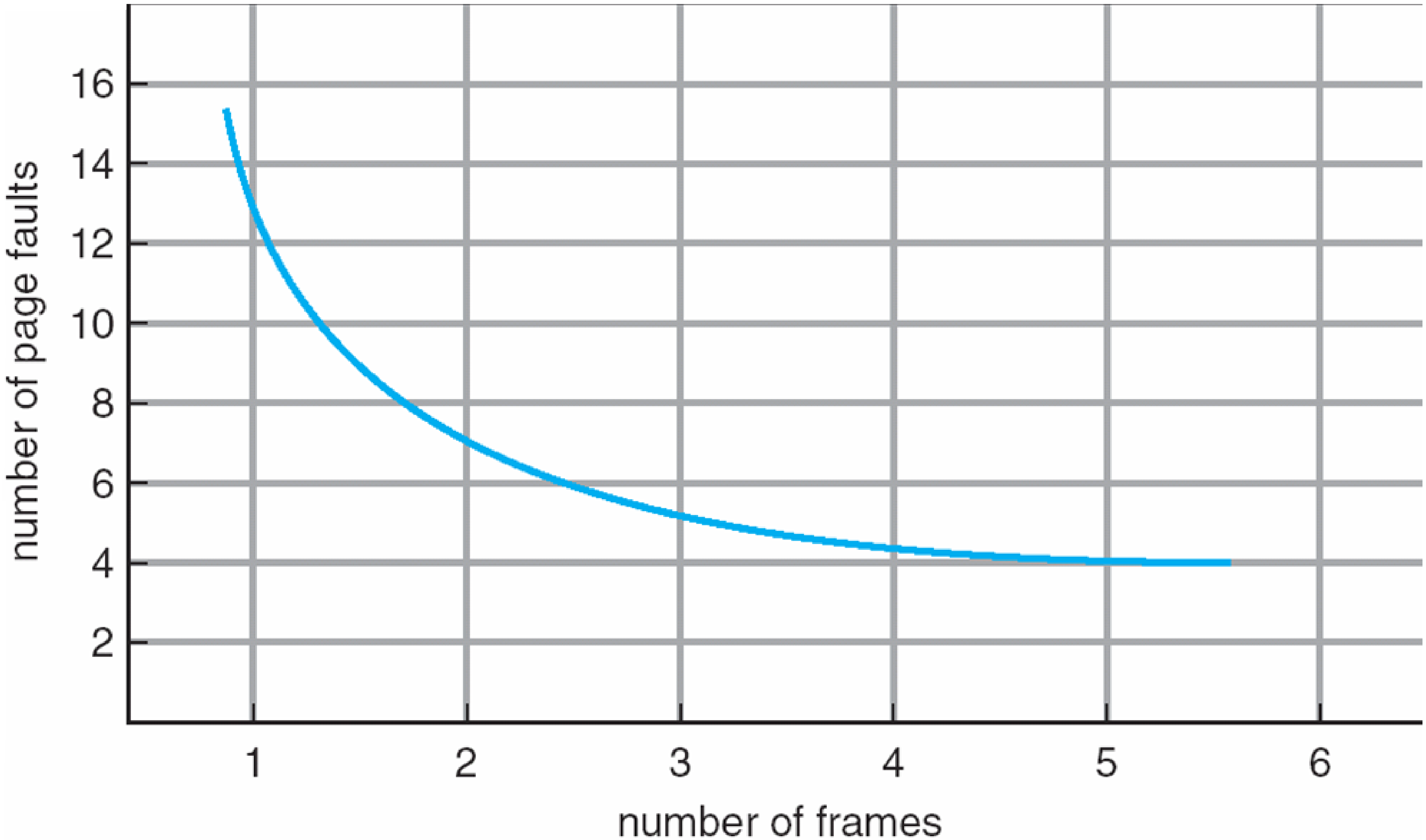


Page Replacement Algorithms

- 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

Graph of Page Faults Versus The Number of Frames

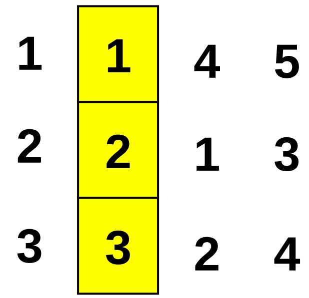


allocated for a process

First-In-First-Out (FIFO) Algorithm

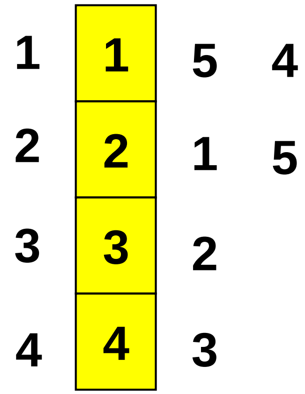
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames (3 pages can be in memory at a time per process)



9 page faults

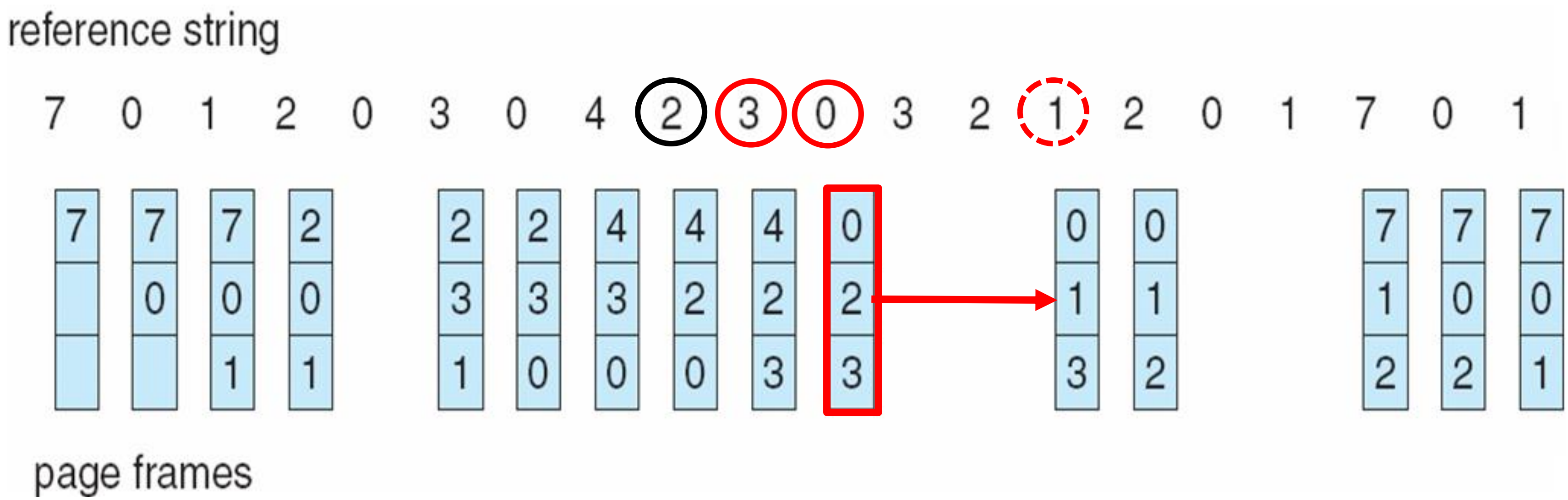
4 frames



10 page faults

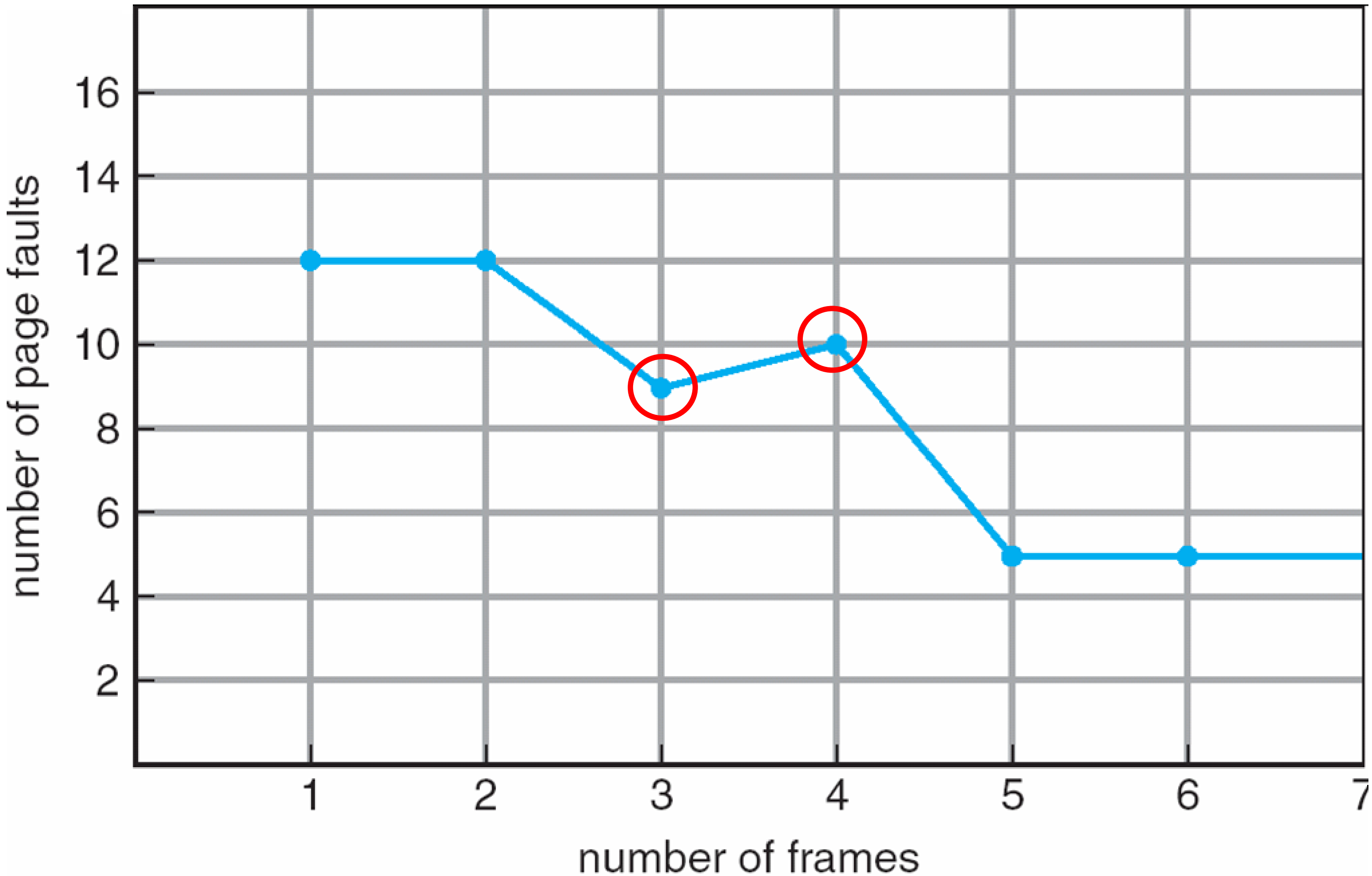
Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



Total number of page faults = 15

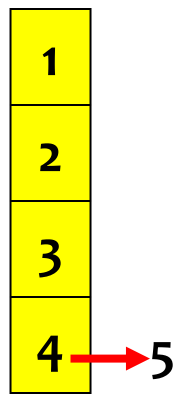
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

Replace page that will not be used for longest period of time (最久之後用到)

4 frames example 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



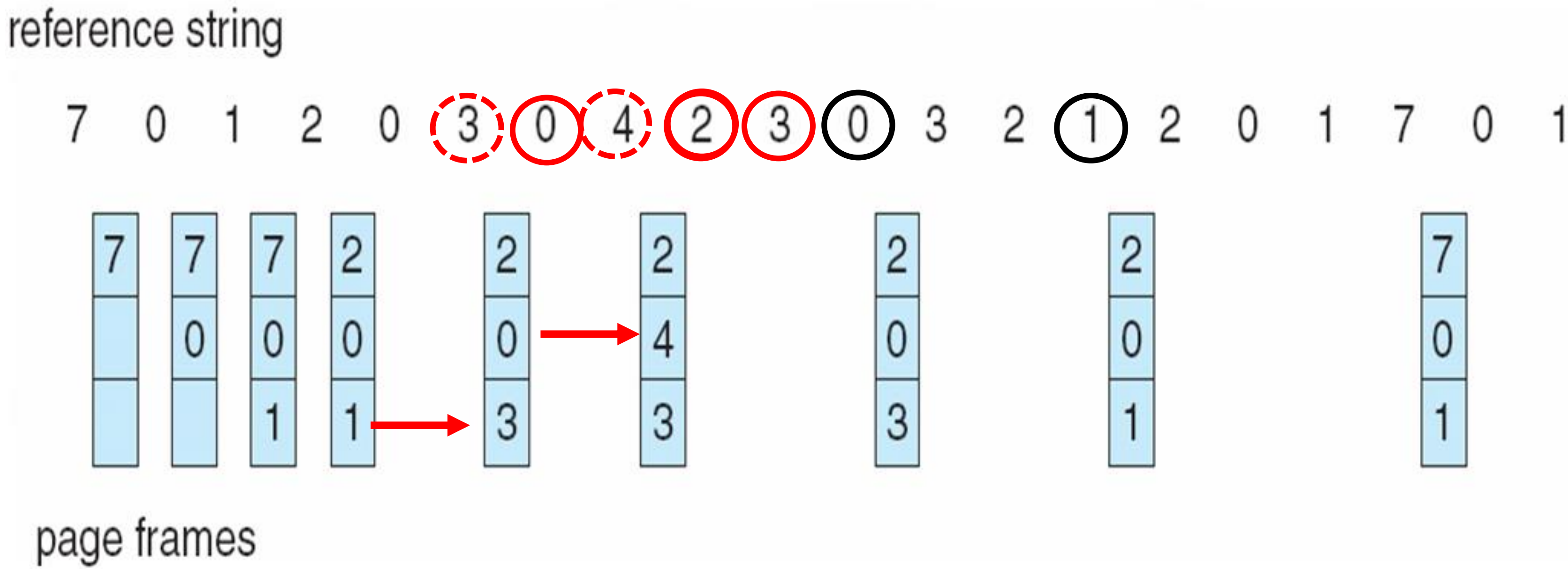
4

6 page faults

How do you know this?

Used for measuring how well your algorithm performs (lower bound)

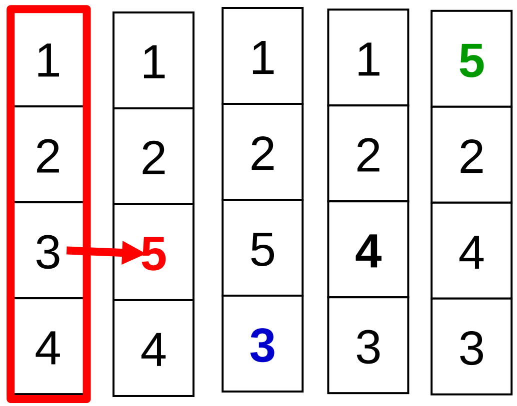
Optimal Page Replacement



Total number of page faults = 9

Least Recently Used (LRU) Algorithm

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



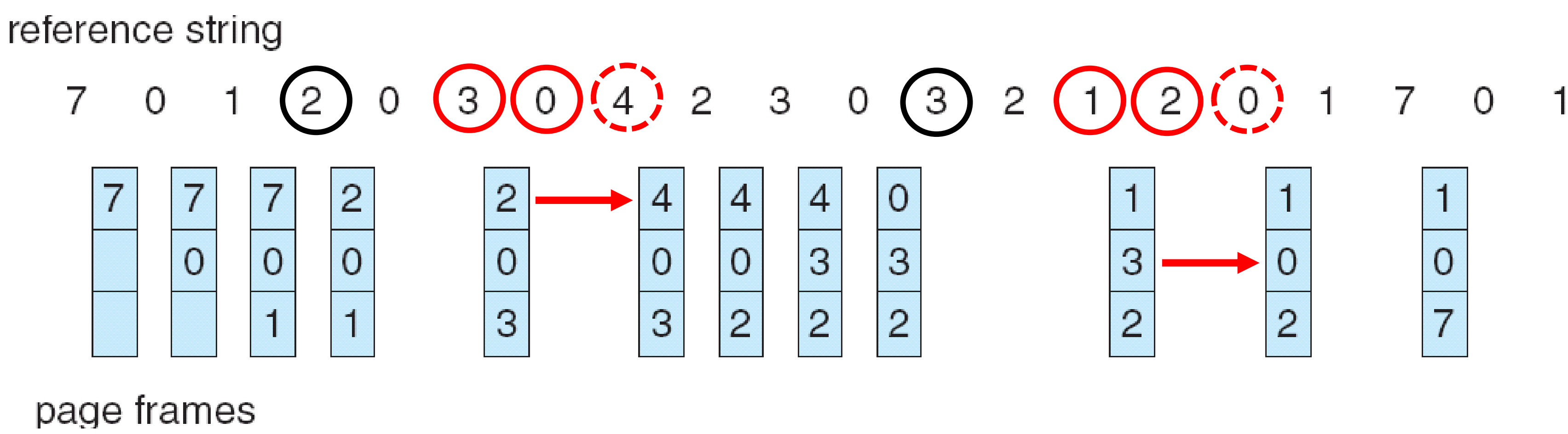
(最早之前用過)

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 is to change

LRU Page Replacement



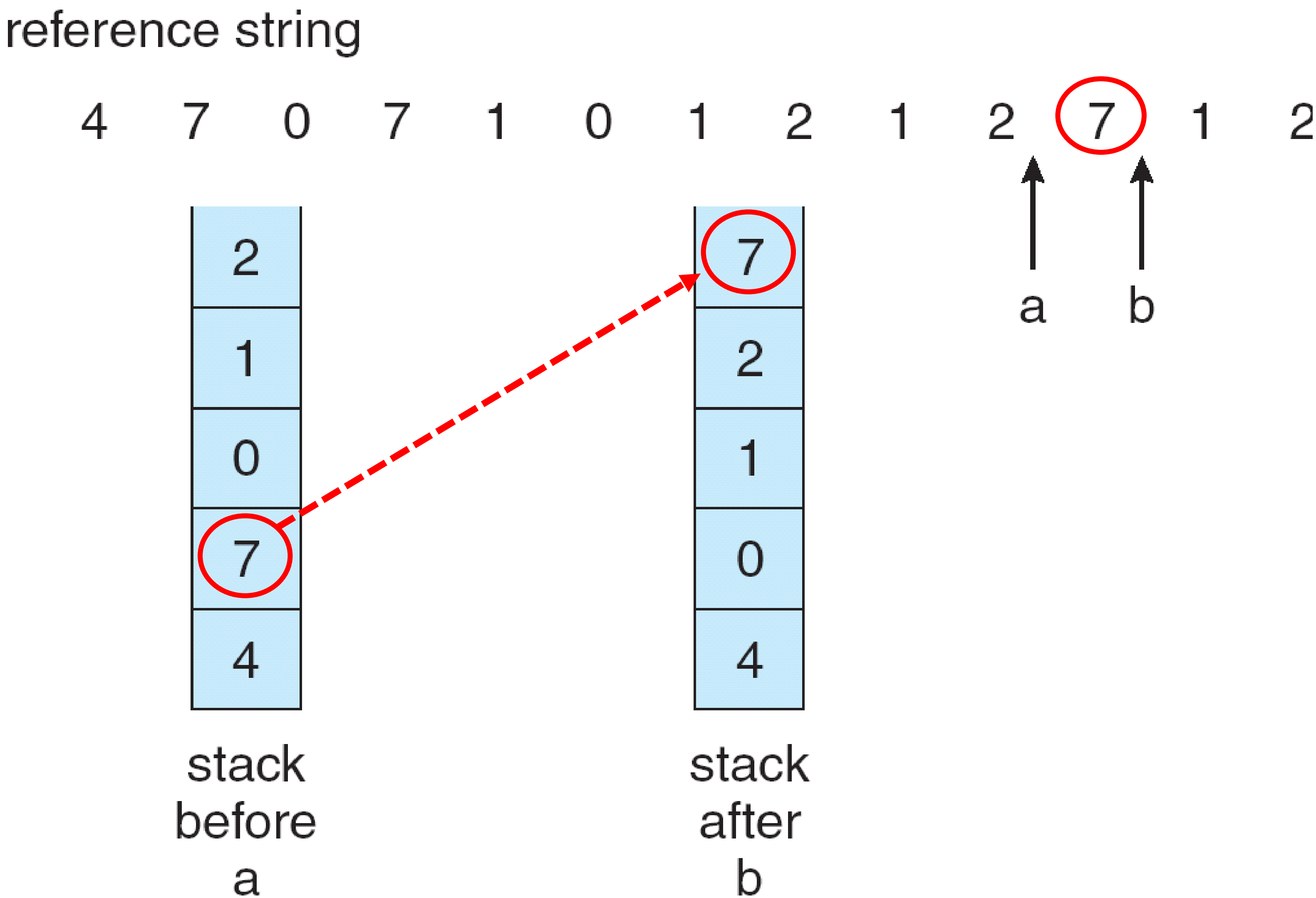
Total number of page faults = 12

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 – Replace the bottom page of the stack

Stack implementation



LRU Approximation Algorithms

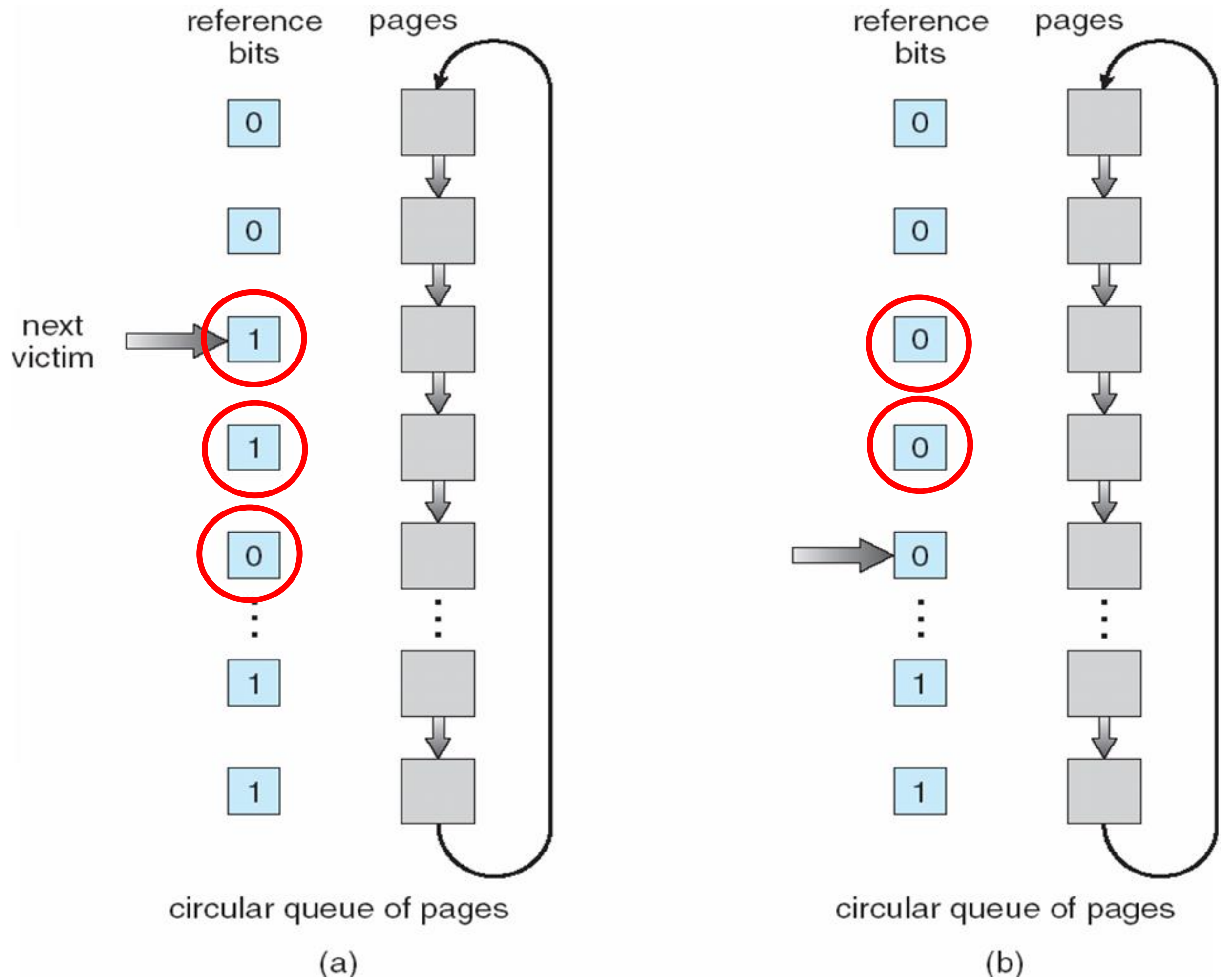
Reference bit

- 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

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

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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

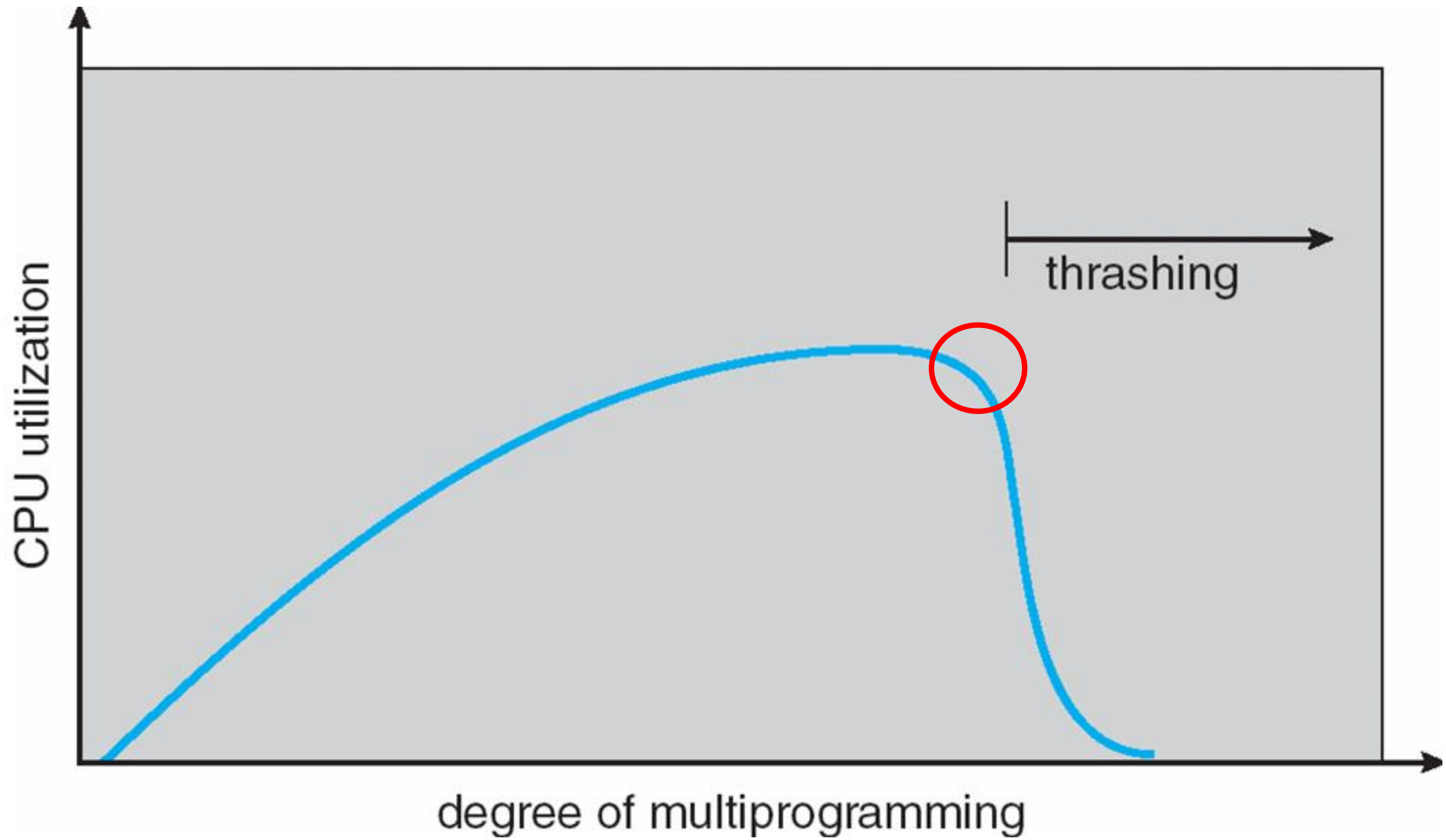
Thrashing

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 $\equiv$ a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

Why does demand paging work?

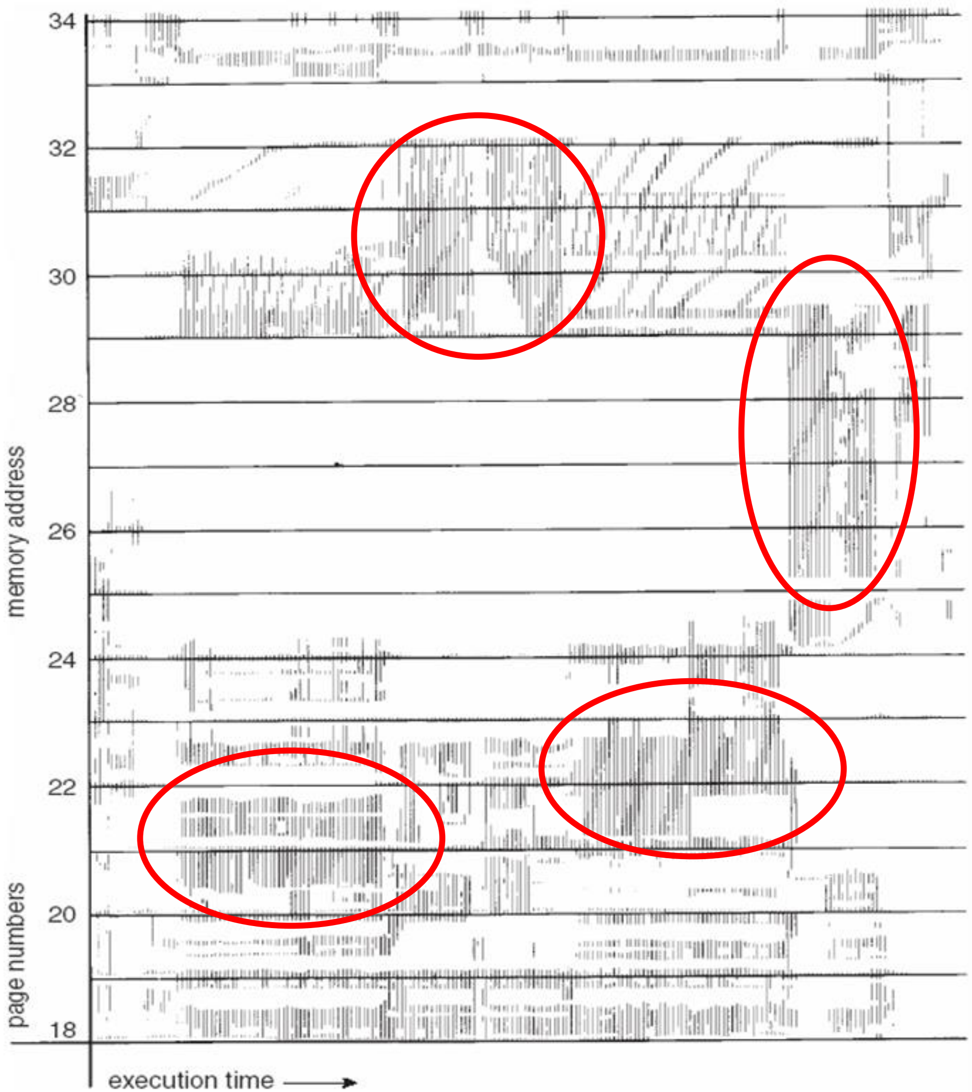
Locality model

- Process migrates from one locality to another
- Localities may overlap

Why does thrashing occur?

Σ size of locality > total memory size

Locality In A Memory-Reference Pattern



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)

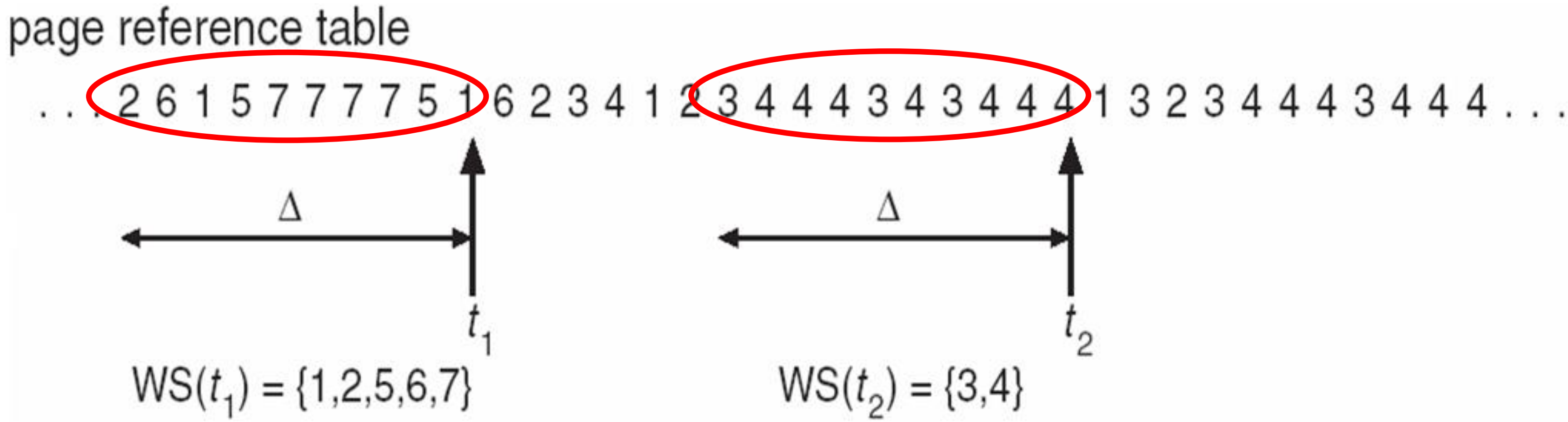
- if Δ too small will not encompass entire locality
- if Δ too large will encompass several localities
- if $\Delta = \infty \Rightarrow$ will encompass entire program

$D = \Sigma\, WSS_i \equiv$ total demand frames

if $D > m \Rightarrow$ Thrashing

Policy if $D >$ m, then suspend one of the processes

Working-set model



Keeping Track of the Working Set

Approximate with interval timer + a reference bit

Example: $\Delta = 10{,}000$

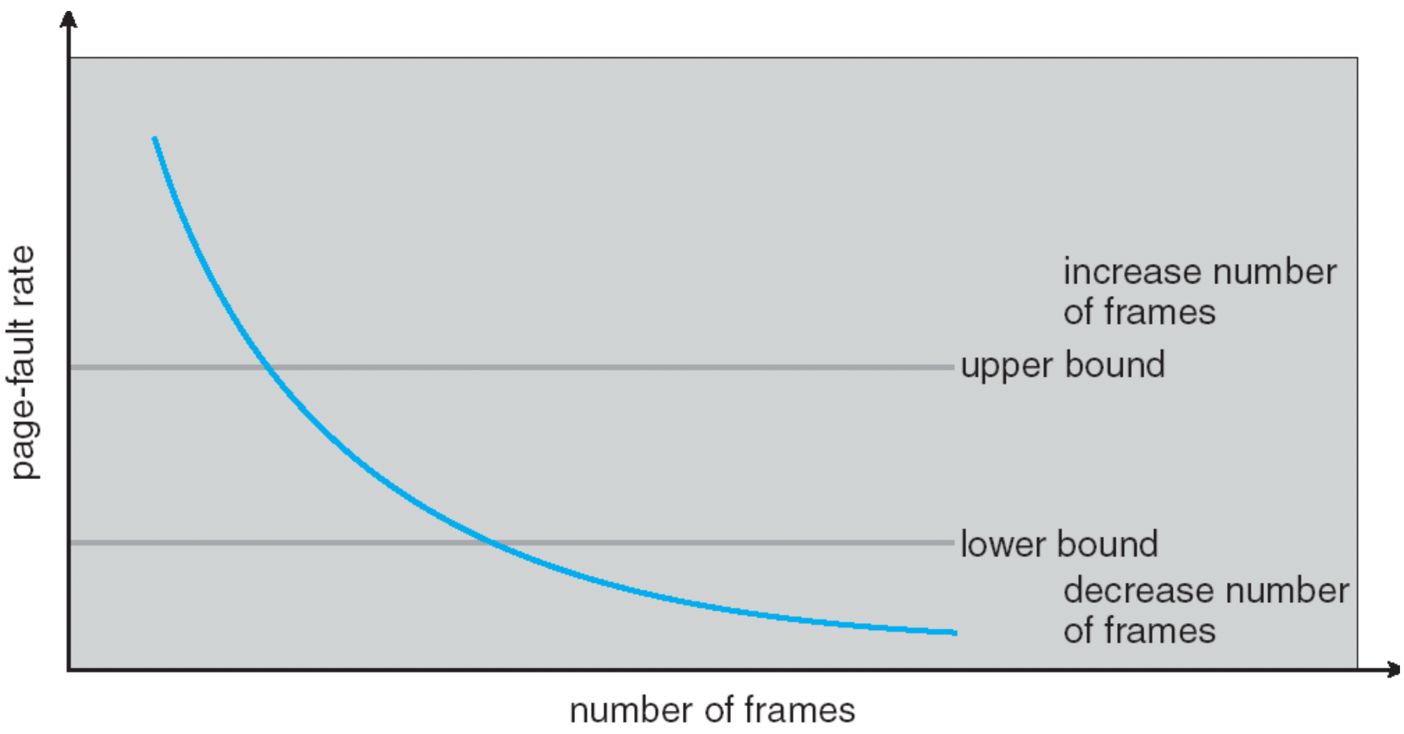
- Timer interrupts after every 5000 time units
- Keep in memory 2 bits for each page
- Whenever a timer interrupts copy and set the values of all reference bits to 0
- If one of the bits in memory = 1 $\Rightarrow$ page in working set

Why is this not completely accurate?

Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

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

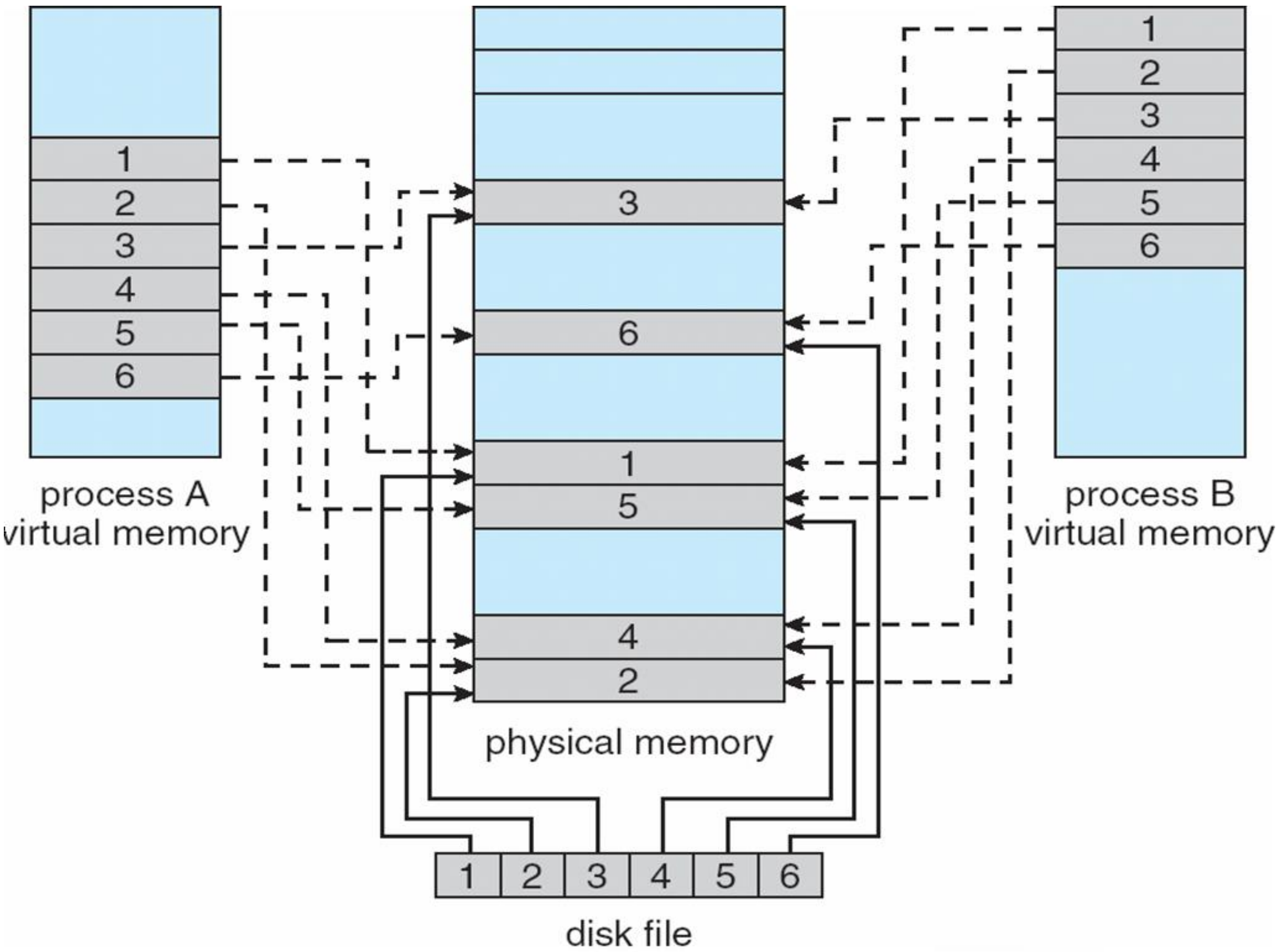


Working Sets and Page Fault Rates

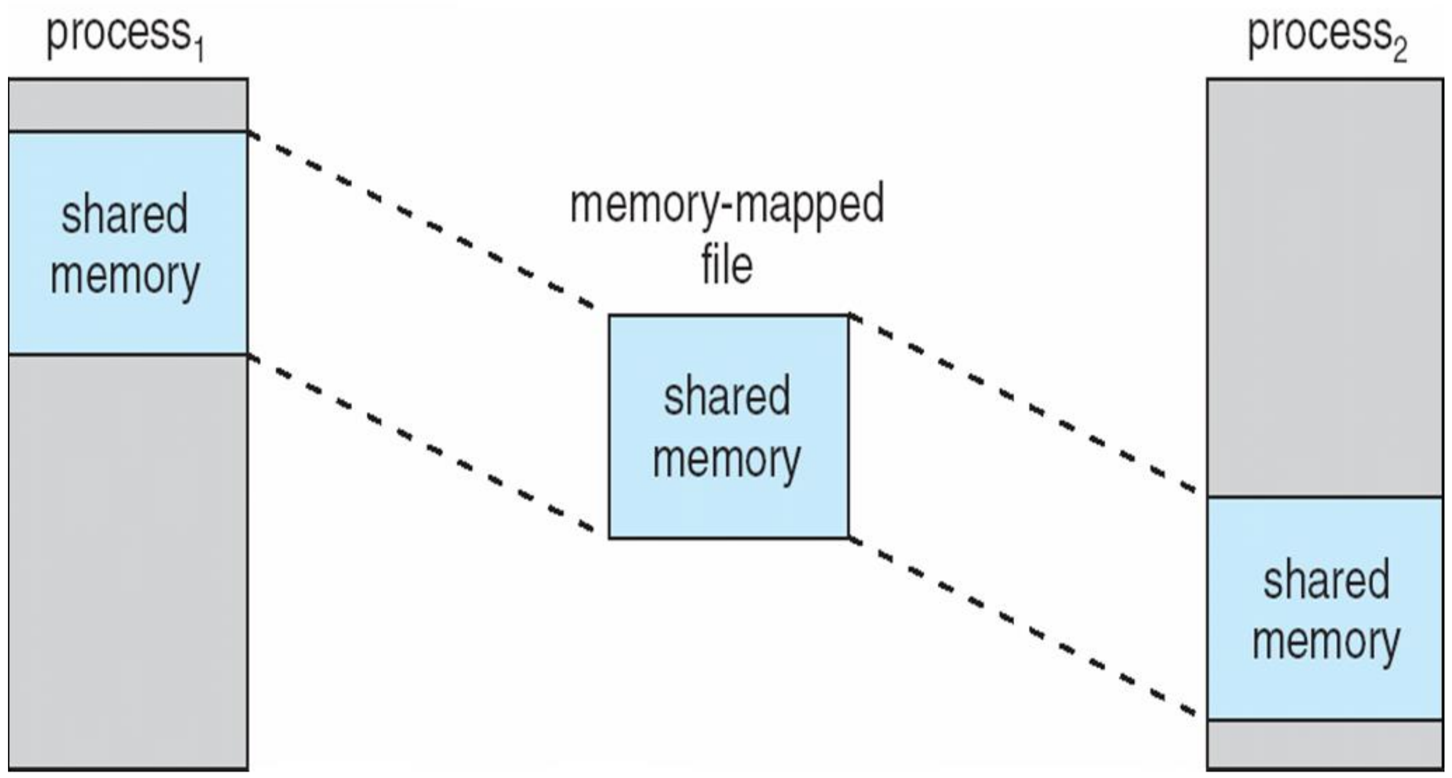
Memory-Mapped Files

- 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 from/to 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



Memory-Mapped Shared Memory in Windows



Allocating Kernel Memory

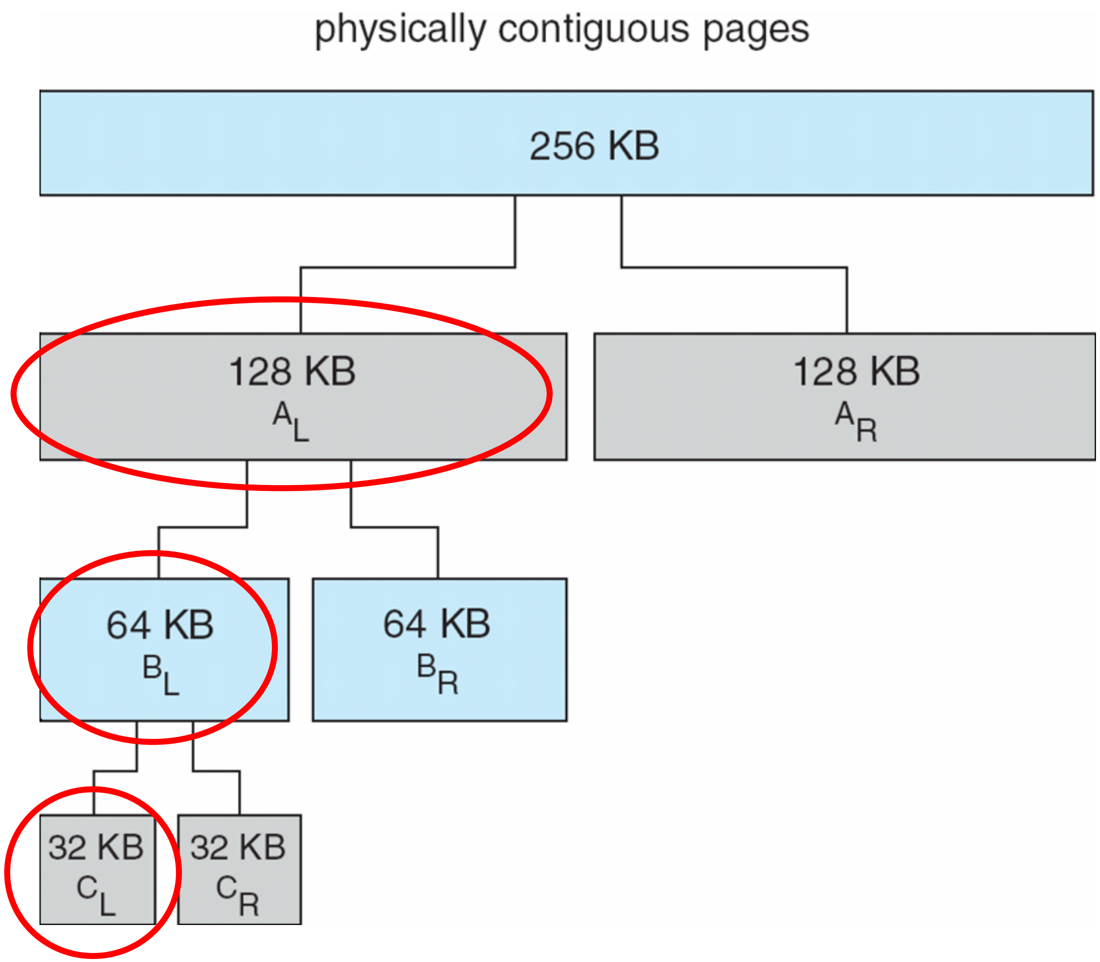
- Treated differently from user mode memory (list of free..)
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes, some of which are less than a page in size.
  - The kernel must use memory conservatively and attempt to minimize waste due to fragmentation.
  - Many OS do not subject kernel code or data to the paging system.
  - Some kernel memory needs to be contiguous due to certain hardware devices interact directly with physical memory – without the benefit of a virtual memory interface.
- Two strategies: Buddy System and Slab Allocation

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory is allocated from this segment using a power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2, for 11kB, it is satisfies with a 16-KB segment
  - 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

Buddy System Allocator

Assume a size of a memory segment is initially 256KB and the kernel requests 21 KB of memory. C_L is the segment allocated to this request.



Buddy System Allocator

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing.

When kernel releases CL,

$C_L + C_R \rightarrow B_L$,

$B_L + B_R \rightarrow A_L$,

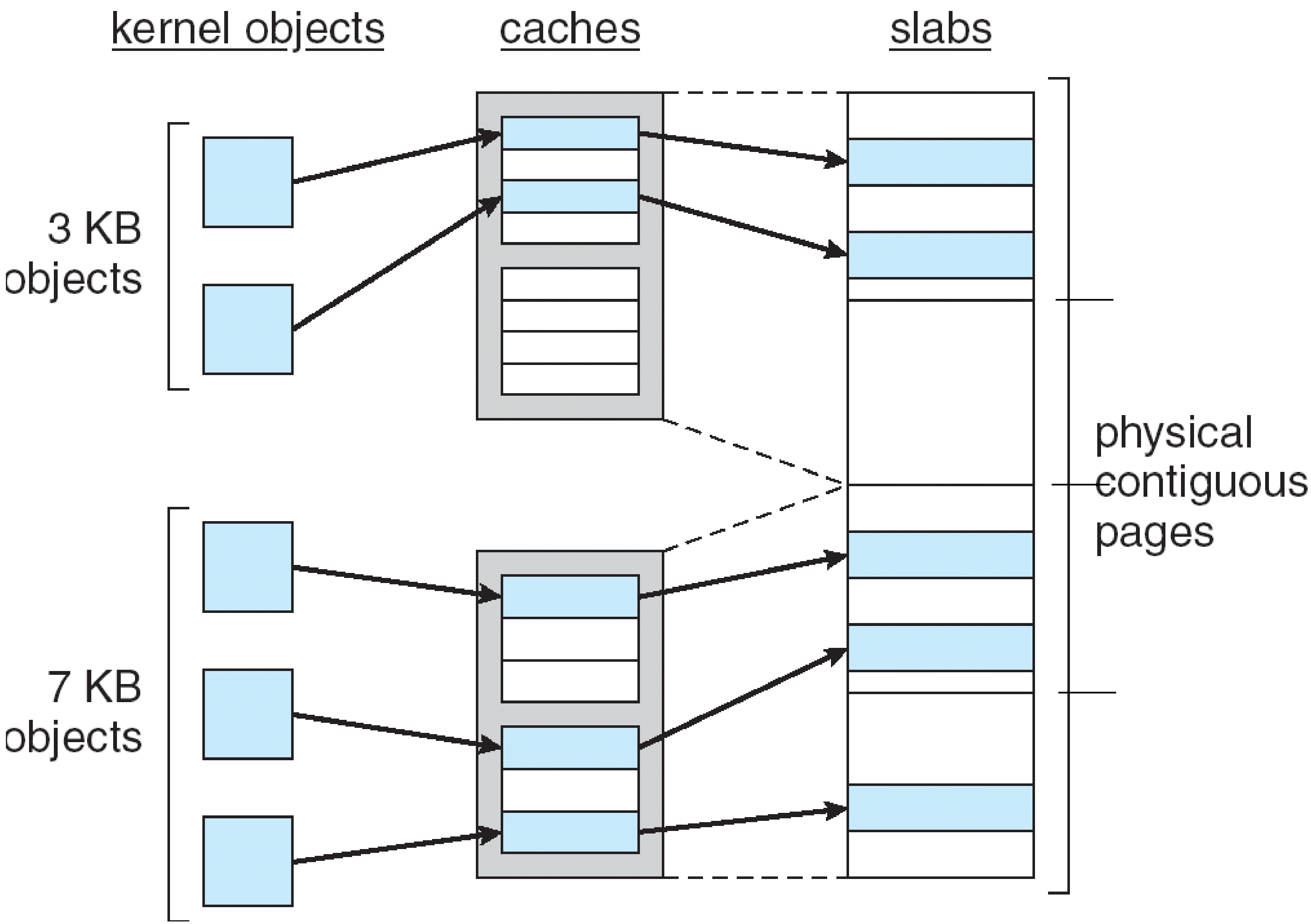
$A_L + A_R \rightarrow$ 256KB segment.

- Drawback: cause fragmentation within allocated segments.
- The next one is a memory allocation scheme where no space is lost due to fragmentation

Slab Allocation

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- **Single cache for each unique kernel data structure**
  - A separate cache for the data structure representing process descriptor
  - A separate cache for file objects
  - A separate cache for semaphores
- Each cache filled with **objects** – **instantiations of the kernel data structure** the cache represents.

Slab Allocation



Slab Allocation

The slab-allocations algorithm uses caches to store kernel objects

When a cache is created, a number of objects are allocated to the cache (initially marked as free).

The number of objects in the cache depends on the size of the associated slab. A 12-KB slab can store six 2-KB objects.

Slab Allocation

When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request.

The object assigned from the cache is marked as used

If slab is full of used objects, next object allocated from empty slab

- If no empty slabs, new slab allocated

Two main benefits

- No memory is wasted due to fragmentation.
- Memory request can be satisfied quickly. (Objects are created in advance and thus can be quickly allocated from cache)

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 α of the pages is used
  - 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

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 (translation look-aside buffers, Chapter 8)

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

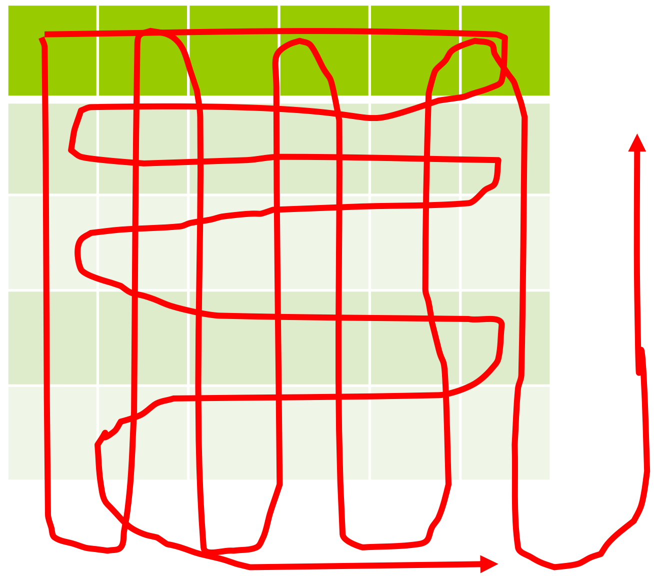
Int[128,128] data;

Each row is stored in one page (need 128 pages to store)

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

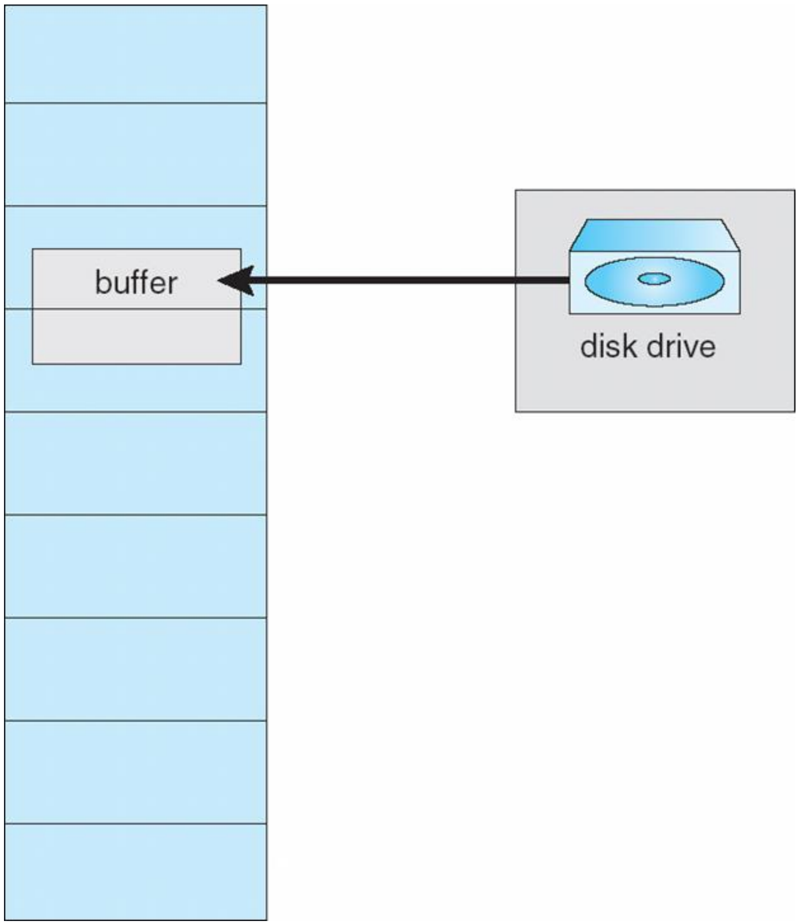
```
for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
              data[i,j] = 0;
```

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



Operating System Examples

Windows XP

Solaris

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
- A process may be assigned as many pages up to its working set maximum
- 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

Solaris

Maintains a list of free pages to assign faulting processes (threads)

Lotsfree – threshold parameter (amount of free memory, usually 1/64 of the physical memory) to begin paging

Desfree – threshold parameter to increasing paging (from 4 times to 100 times/sec)

Minfree – threshold parameter to begin swapping processes, thereby freeing all pages allocated to the swapped processes.

Paging is performed by *pageout* process

Pageout scans pages (four times per second) using modified clock algorithm (two hands)

Solaris

The front hand scans the pages and sets the ref bit to 0

The back hand checks and appends each page with ref bit still equals 0 to the free list.

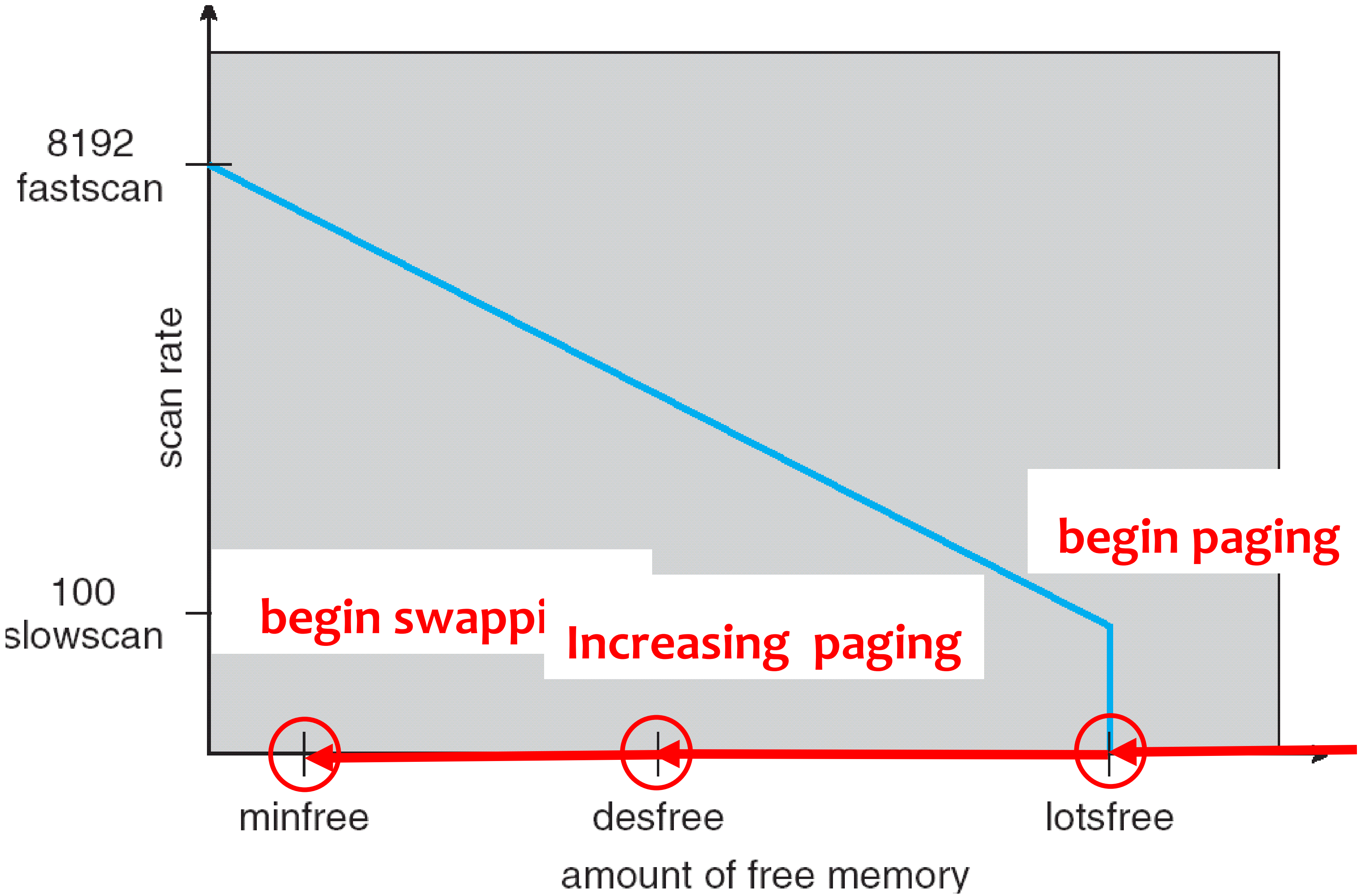
Handspread: the distance (in pages) between the two hands

Scanrate is the rate at which pages are scanned. This ranges from *slowscan* *(100 pages/sec)* to *fastscan* *(up to 8192 pages/sec)*

The amount of time between the two hands depends on scanrate and handspread. For scanrate = 100/sec, handspread = 1000 pages, we have 10 sec.

Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner



End of Chapter 9