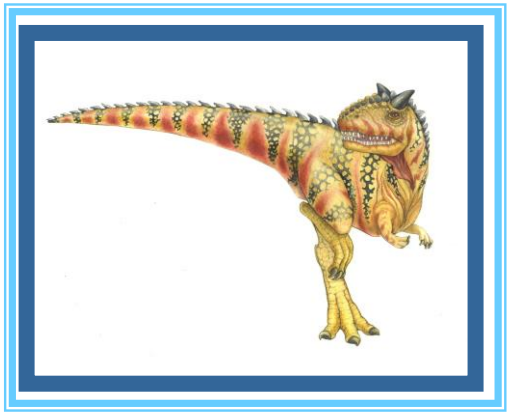
Chapter 8: Memory-Management Strategies



Chapter 8: Memory Management Strategies

Background

Swapping

Contiguous Memory Allocation

Paging

Structure of the Page Table

Segmentation

Example: The Intel Pentium

Objectives

To provide a detailed description of various ways of organizing memory hardware

To discuss various memory-management techniques, including paging and segmentation

To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

Program must be brought (from disk) into memory and placed within a process for it to be run

Main memory and registers are only storage CPU can access directly

Register access in one CPU clock (or less)

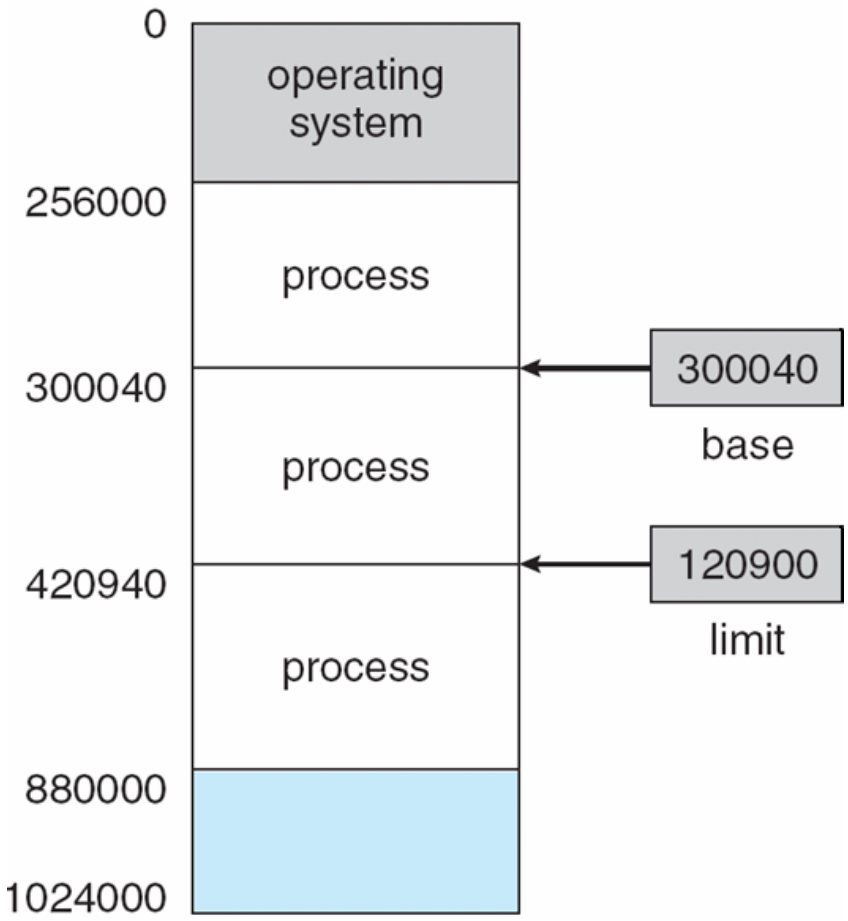
Main memory can take many cycles

Cache sits between main memory and CPU registers

Protection of memory is required to ensure correct operation

Base and Limit Registers

A pair of base and limit registers define the logical address space

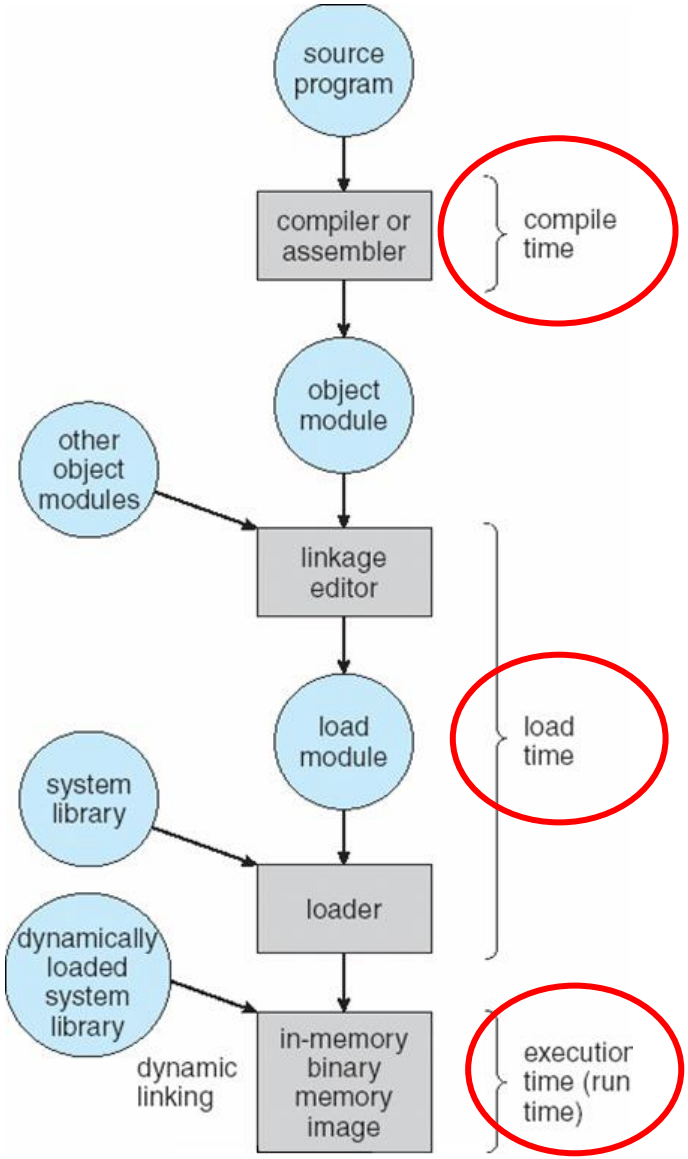


Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management

- Logical address – generated by the CPU; also referred to as virtual address
- Physical address – address seen by the memory unit

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

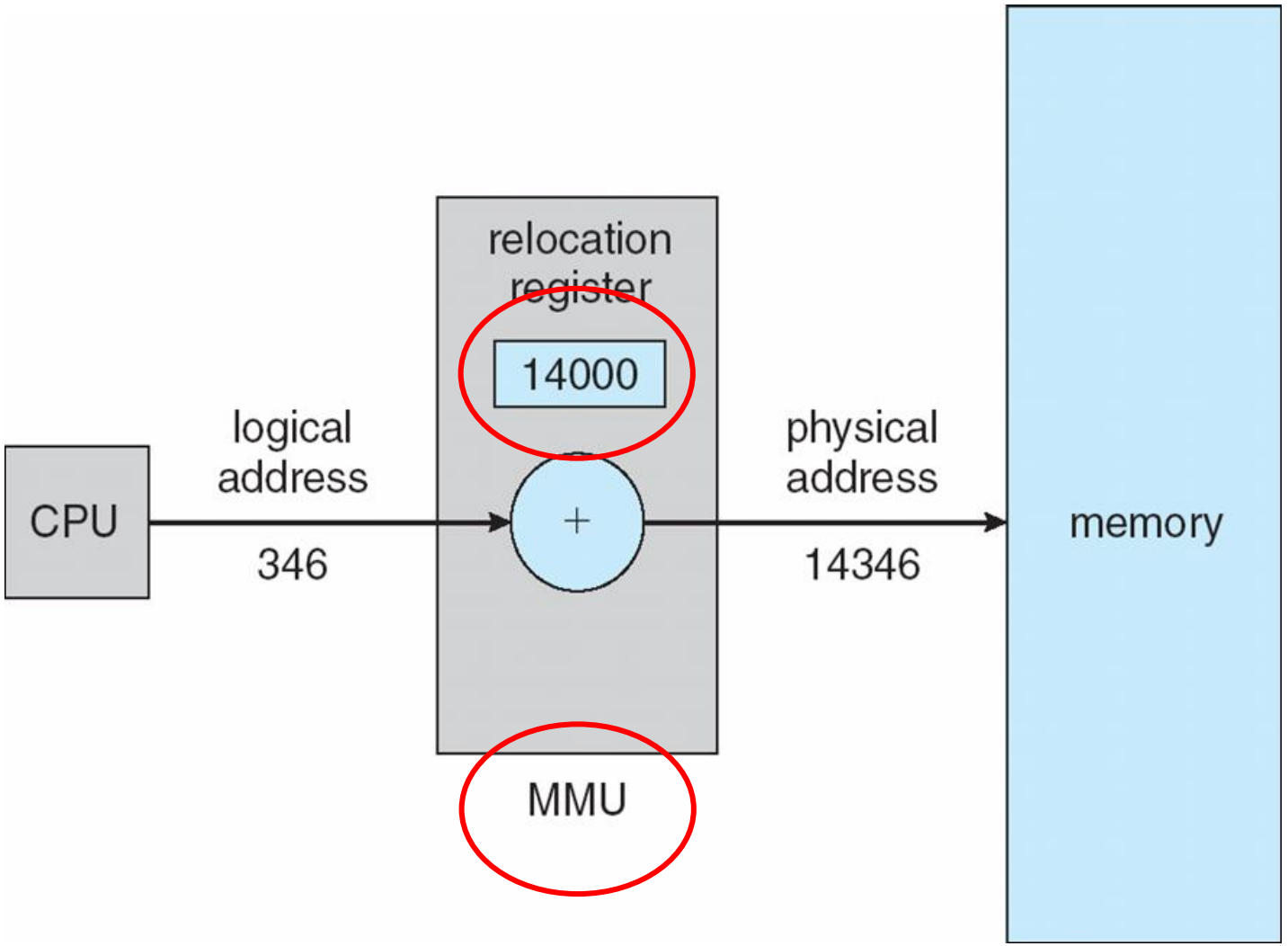
Memory-Management Unit (MMU)

Hardware device that **maps** virtual to physical address

In MMU scheme, the value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory

The user program deals with ***logical*** addresses; it never sees the ***real*** physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

Routine is not loaded until it is called

Better memory-space utilization; unused routine is never loaded

Useful when large amounts of code are needed to handle infrequently occurring cases

No special support from the operating system is required

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as shared libraries

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

Roll out, roll in – swapping variant used for priority-based scheduling algorithms;

- lower-priority process is swapped out so higher-priority process can be loaded and executed

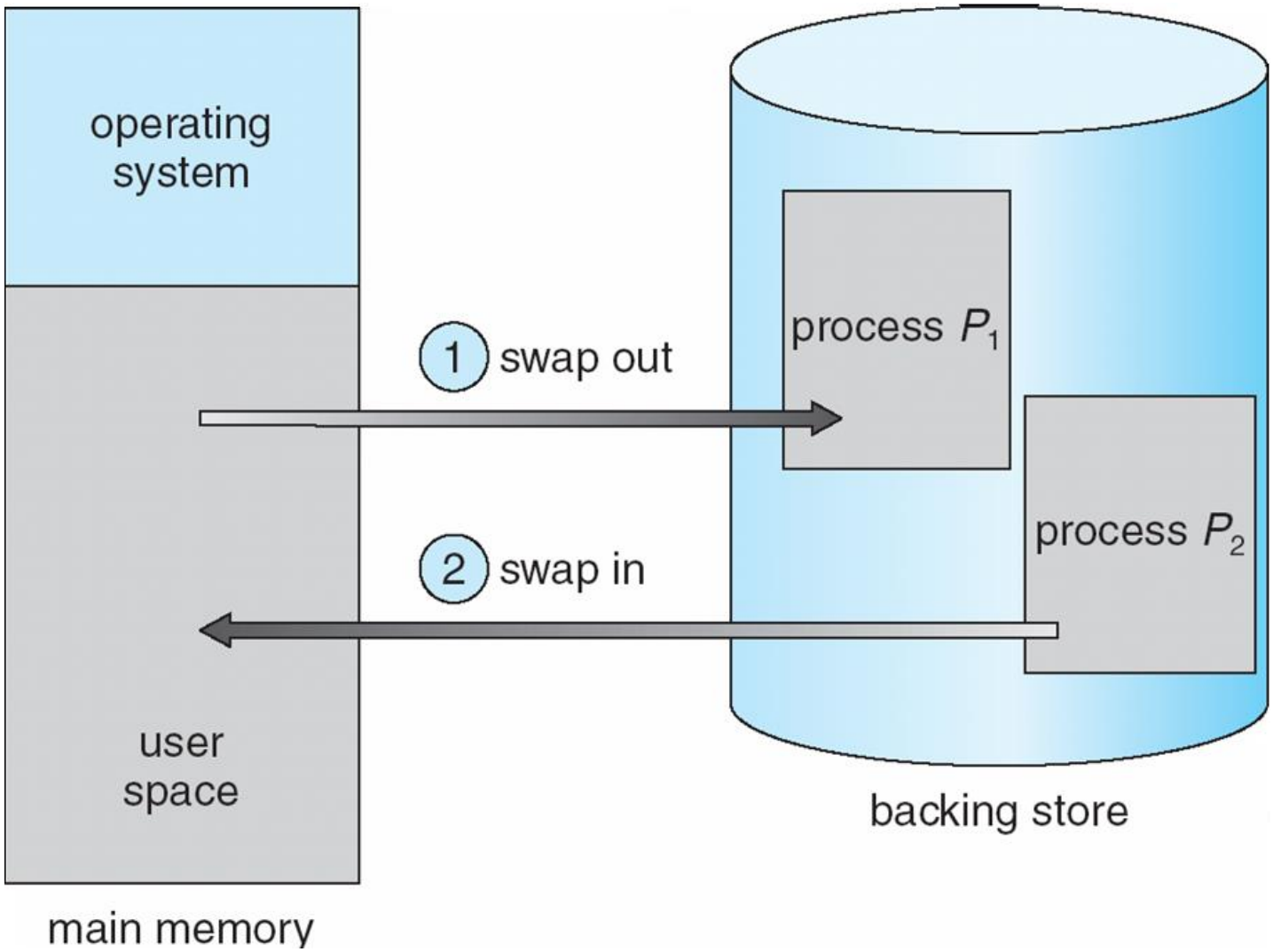
Swapping

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

System maintains a ready queue of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Contiguous Allocation

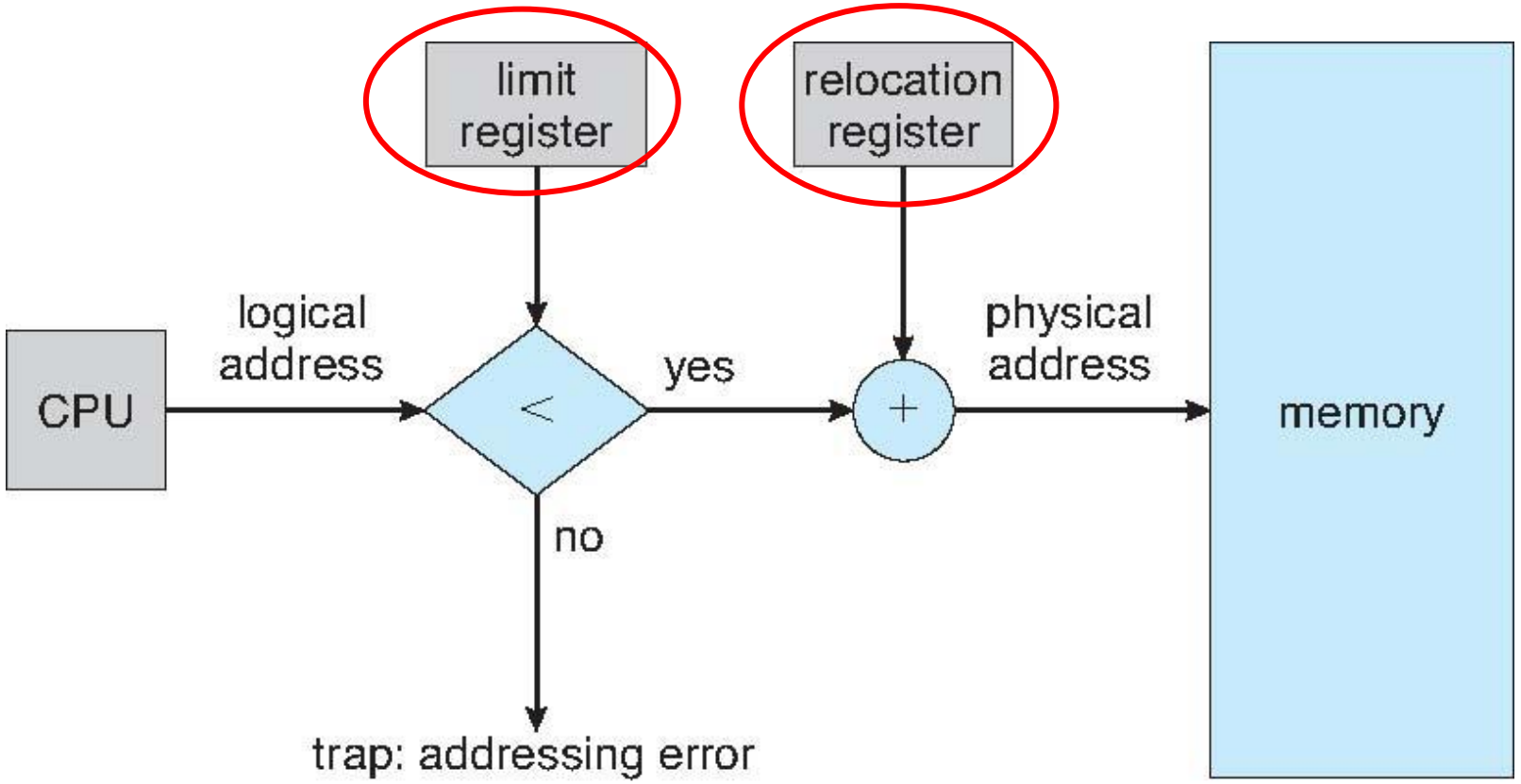
Main memory usually divides into two partitions:

- Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory

Relocation registers used to protect user processes from each other, and from changing operating-system code and data

- Base register contains value of smallest physical address
- Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers



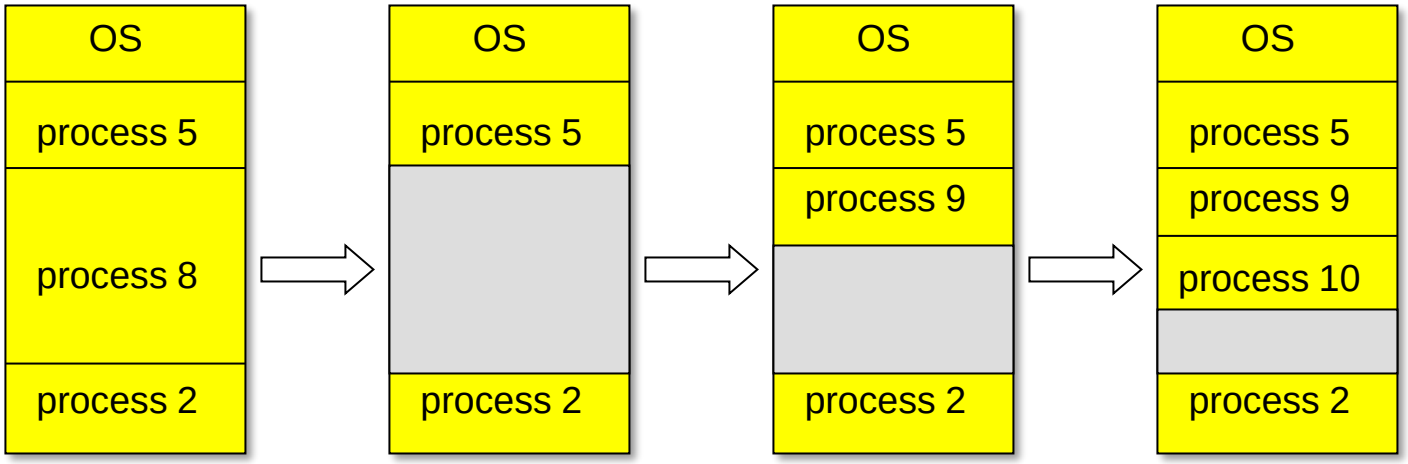
Contiguous Allocation (Cont)

Multiple-partition allocation

Hole – block of available memory; holes of various size are scattered throughout memory

When a process arrives, it is allocated memory from a hole large enough to accommodate it

Operating system maintains information about:
a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes ?

First-fit: Allocate the *first* hole that is big enough

Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size

Produces the smallest leftover hole

Worst-fit: Allocate the *largest* hole; must also search entire list

Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous

Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by compaction

Shuffle memory contents to place all free memory together in one large block

Compaction is possible *only* if relocation is dynamic, and is done at execution time

I/O problem

- Latch job in memory while it is involved in I/O
- Do I/O only into OS buffers

Paging

Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

Paging avoids external fragmentation and the needs for compaction.

Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 - 8,192 bytes)

Divide logical memory into blocks of same size called pages

Keep track of all free frames

To run a program of size *n* pages, need to find *n* free frames and load program

Set up a page table to translate logical to physical addresses

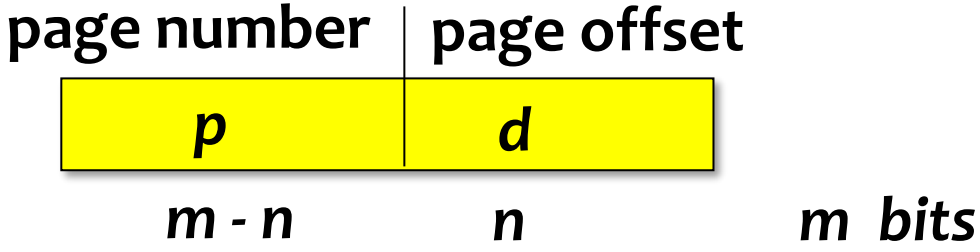
Internal fragmentation

Address Translation Scheme

Address generated by CPU is divided into:

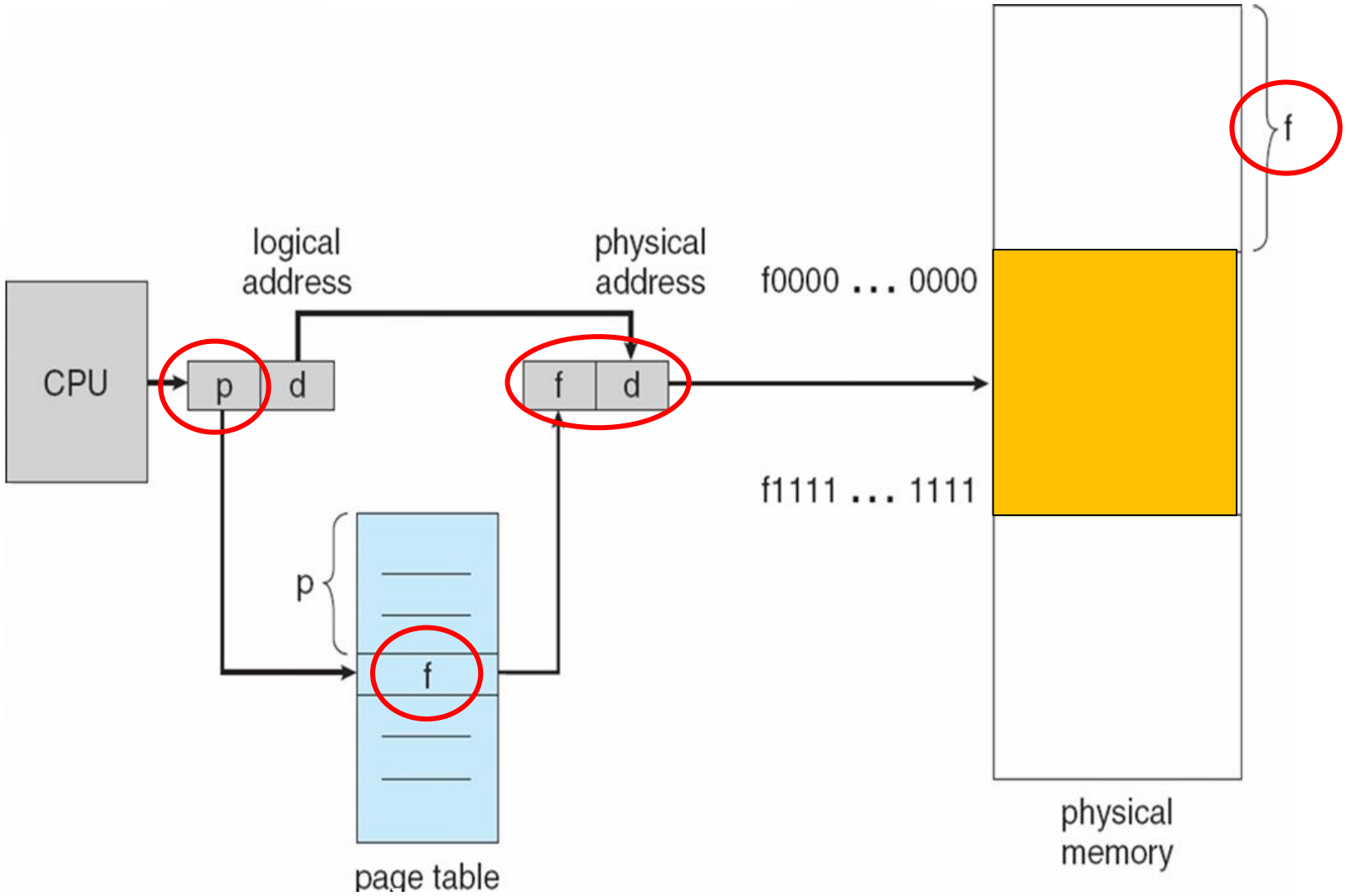
Page number (*p*) – used as an index into a *page table* which contains base address of each page in physical memory

Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

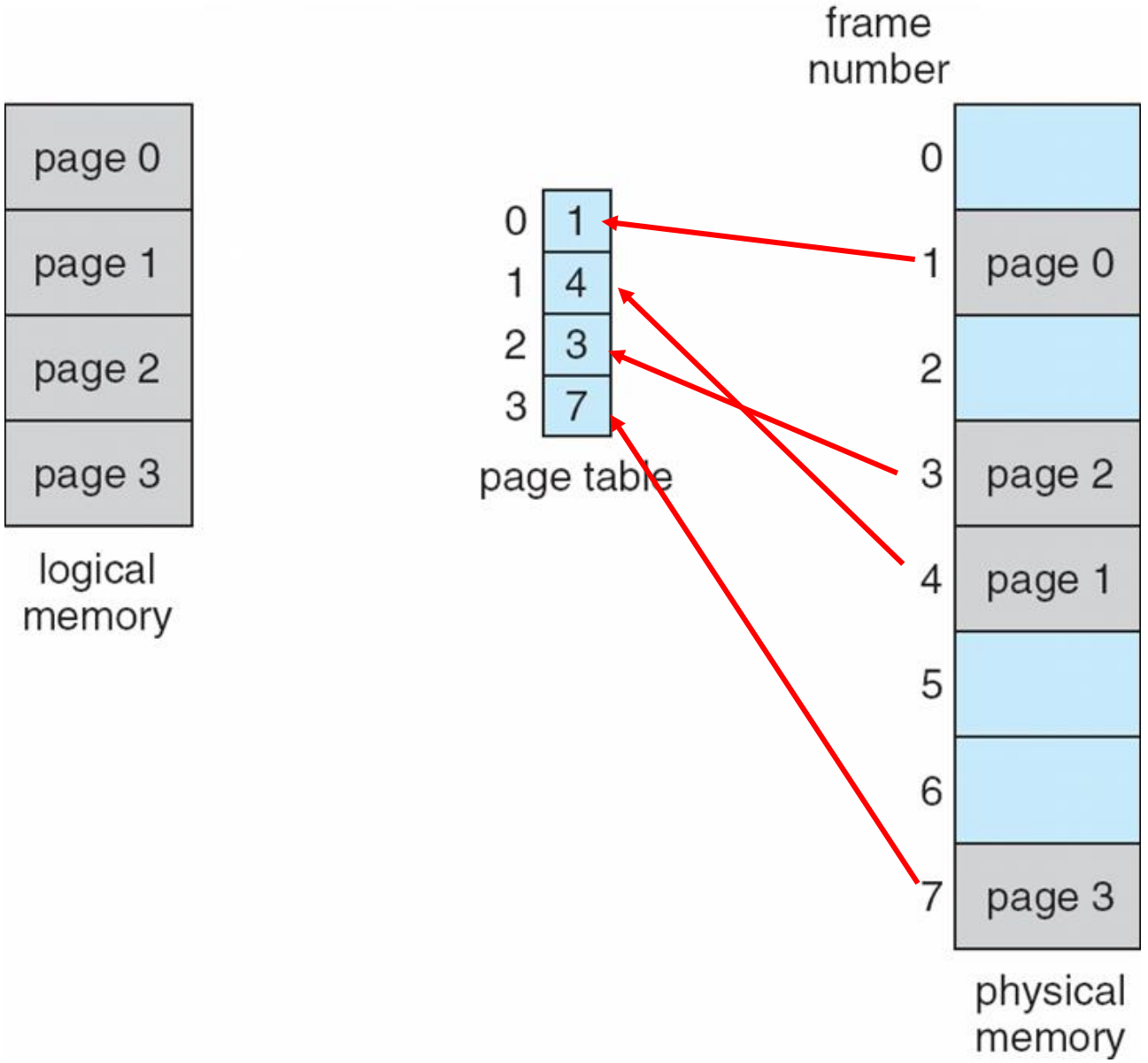


For given logical address space 2^m and *page size* 2^n

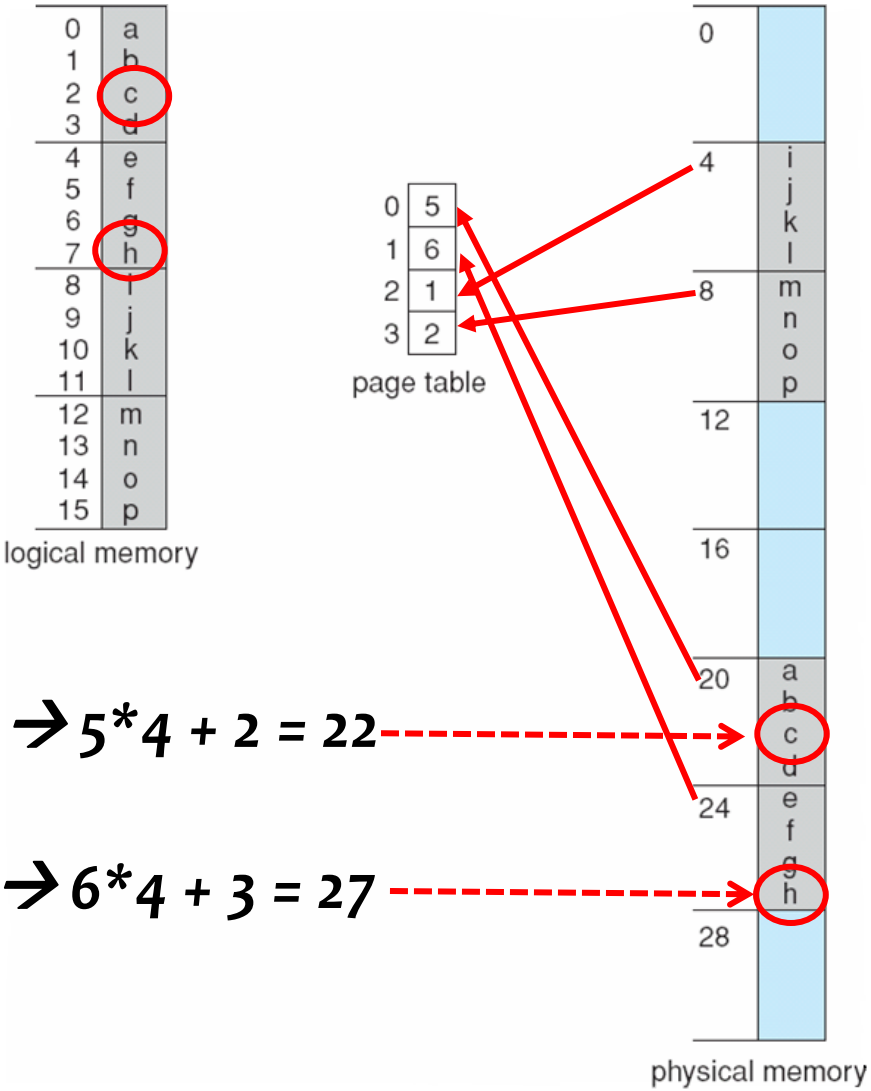
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



32-byte memory and 4-byte pages

Free Frames

Before allocation

After allocation

Implementation of Page Table

Page table is kept in main memory

Page-table base register (PTBR) points to the page table

Page-table length register (PRLR) indicates size of the page table

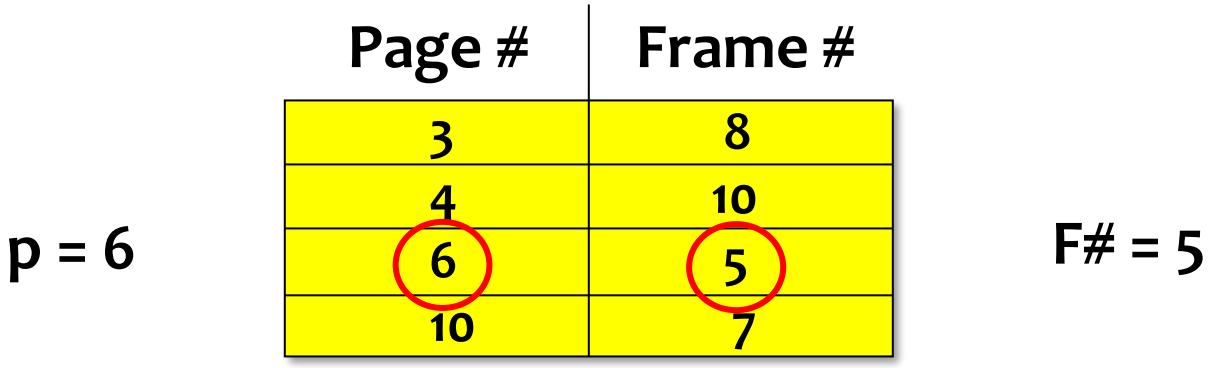
In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

Associative memory – provides parallel search

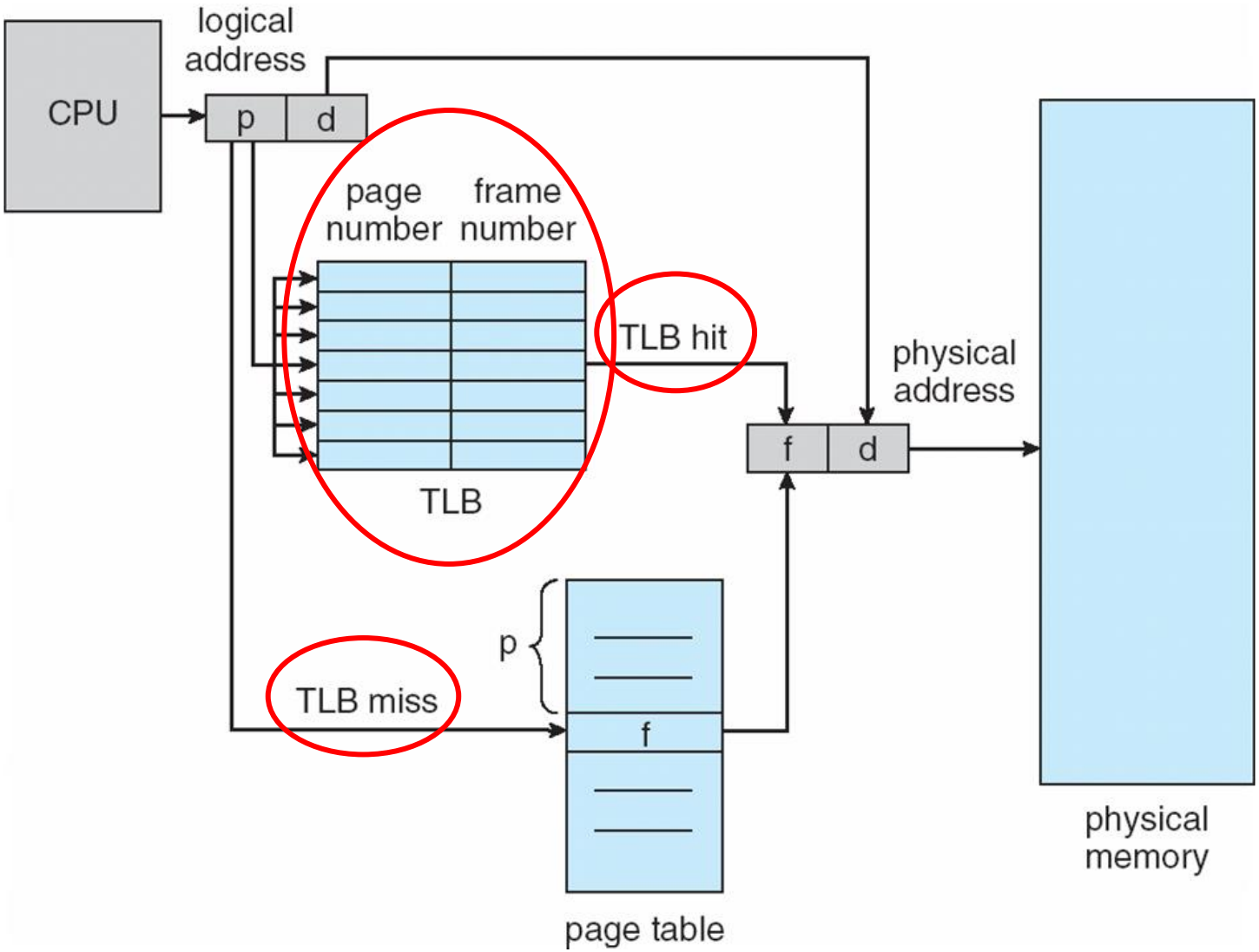


- Address translation (p, d)

  If p is in associative register, get frame # out

  Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

Associative Lookup = ε time unit

Assume memory cycle time is 1 microsecond

Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Hit ratio = α

Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

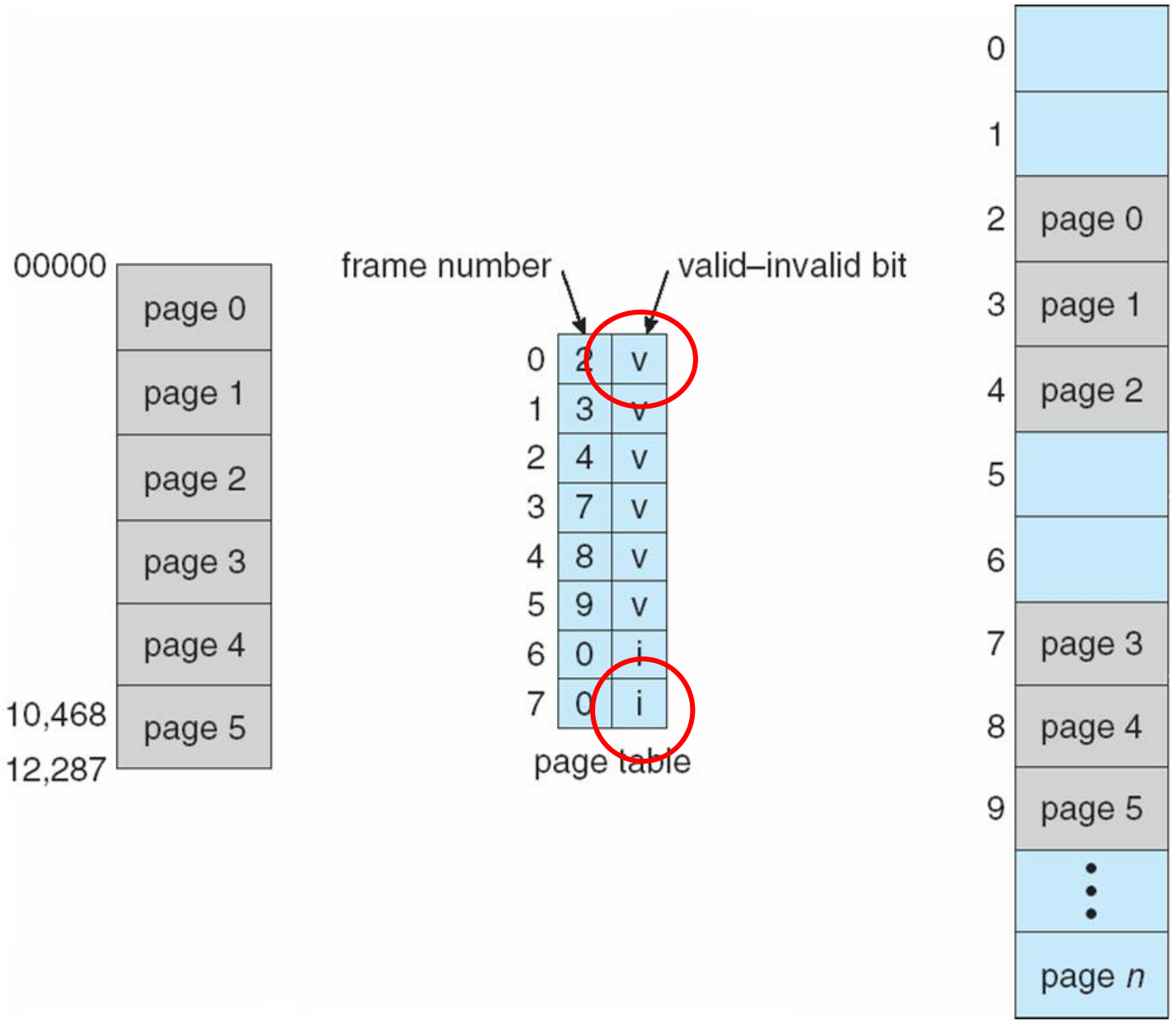
Memory Protection

Memory protection implemented by associating protection bit with each frame

Valid-invalid bit attached to each entry in the page table:

- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space

Valid (v) or Invalid (i) bit in a Page Table



Shared Pages

Shared code

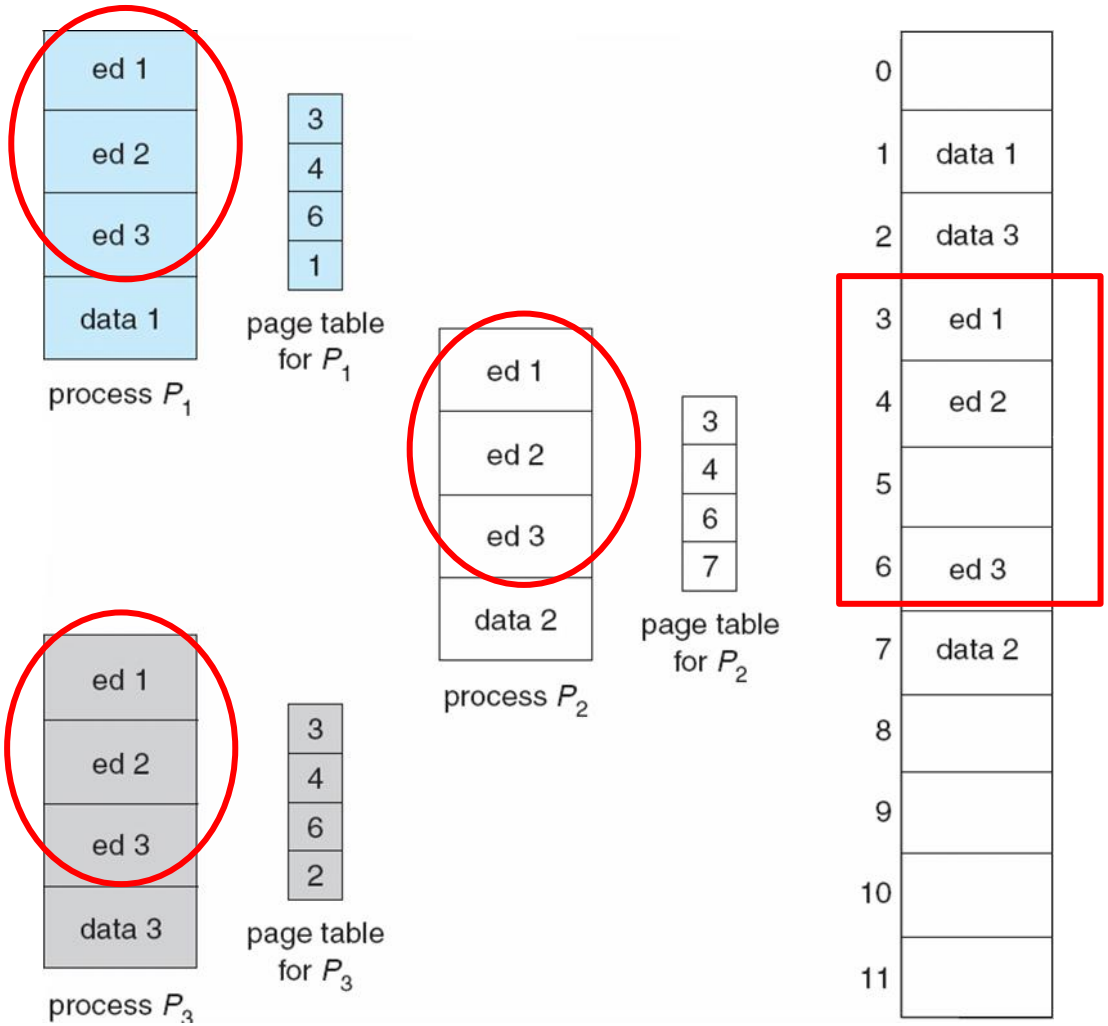
- **One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).**
- **Reentrant code is non-self-modifying code: it never changes during execution.**

Private code and data

- **Each process keeps a separate copy of the code and data**

Some operating systems implement shared memory using shared pages.

Shared Pages Example



Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

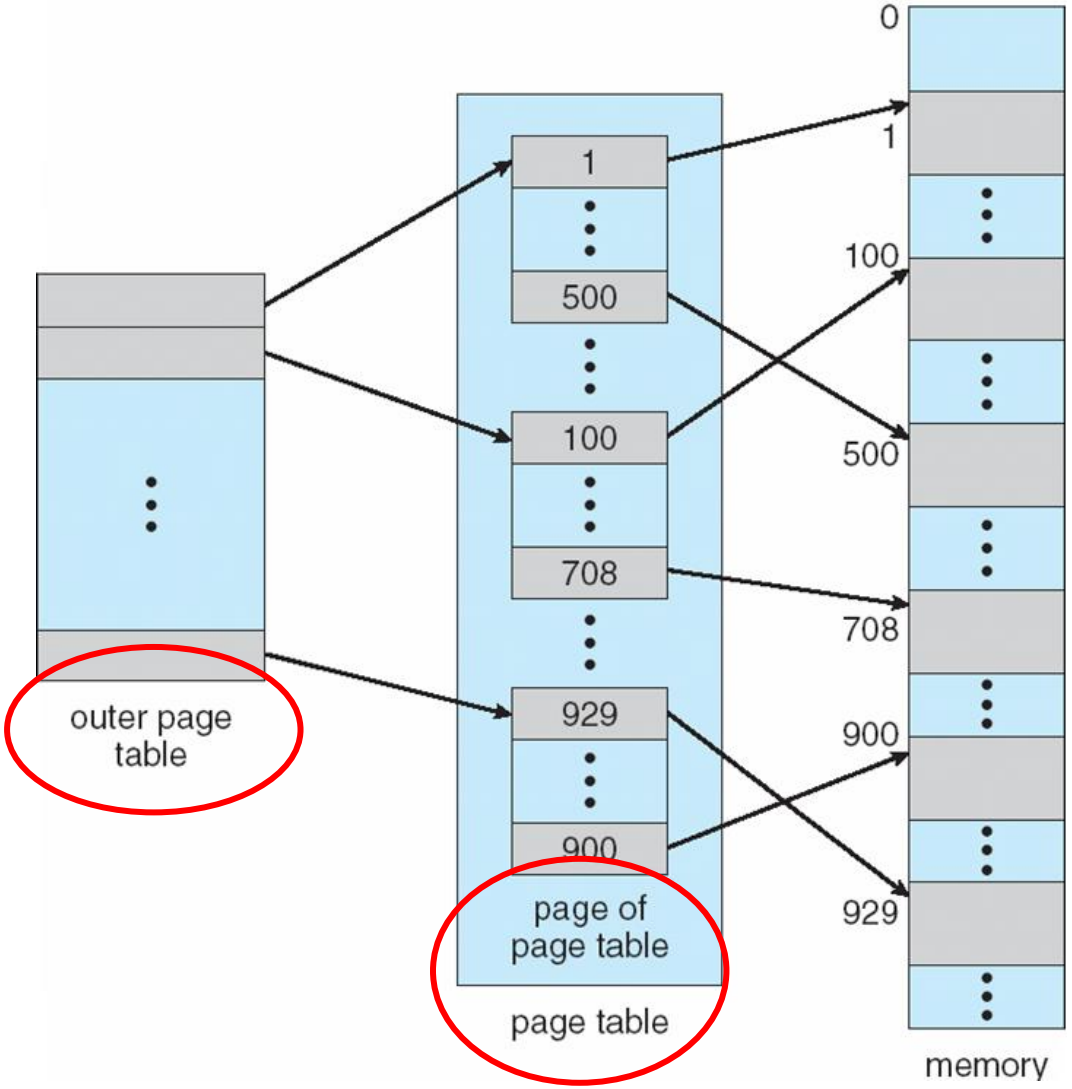
Modern computer systems support a large logical address space 2^{32} to 2^{64}. The page table itself becomes excessively large.

For 32-bit logical address space, and page size of 4K, then a page table consists of 1 million entries ($2^{32}/ 2^{12} = 2^{20}$ = 1 million).

Break up the logical address space into multiple page tables

A simple technique is a two-level page table

Two-Level Page-Table Scheme



Two-Level Paging Example

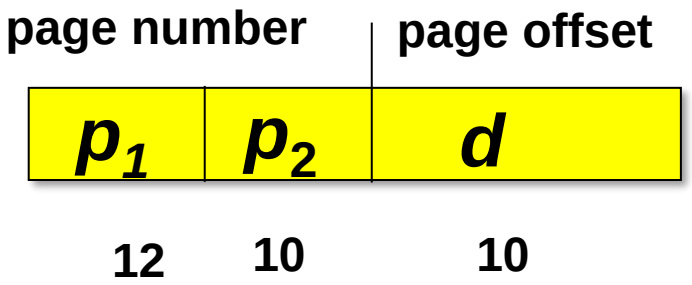
A logical address (on 32-bit machine with 1K page size) is divided into:

- a page number consisting of 22 bits
- a page offset consisting of 10 bits

Since the page table is paged, the page number is further divided into:

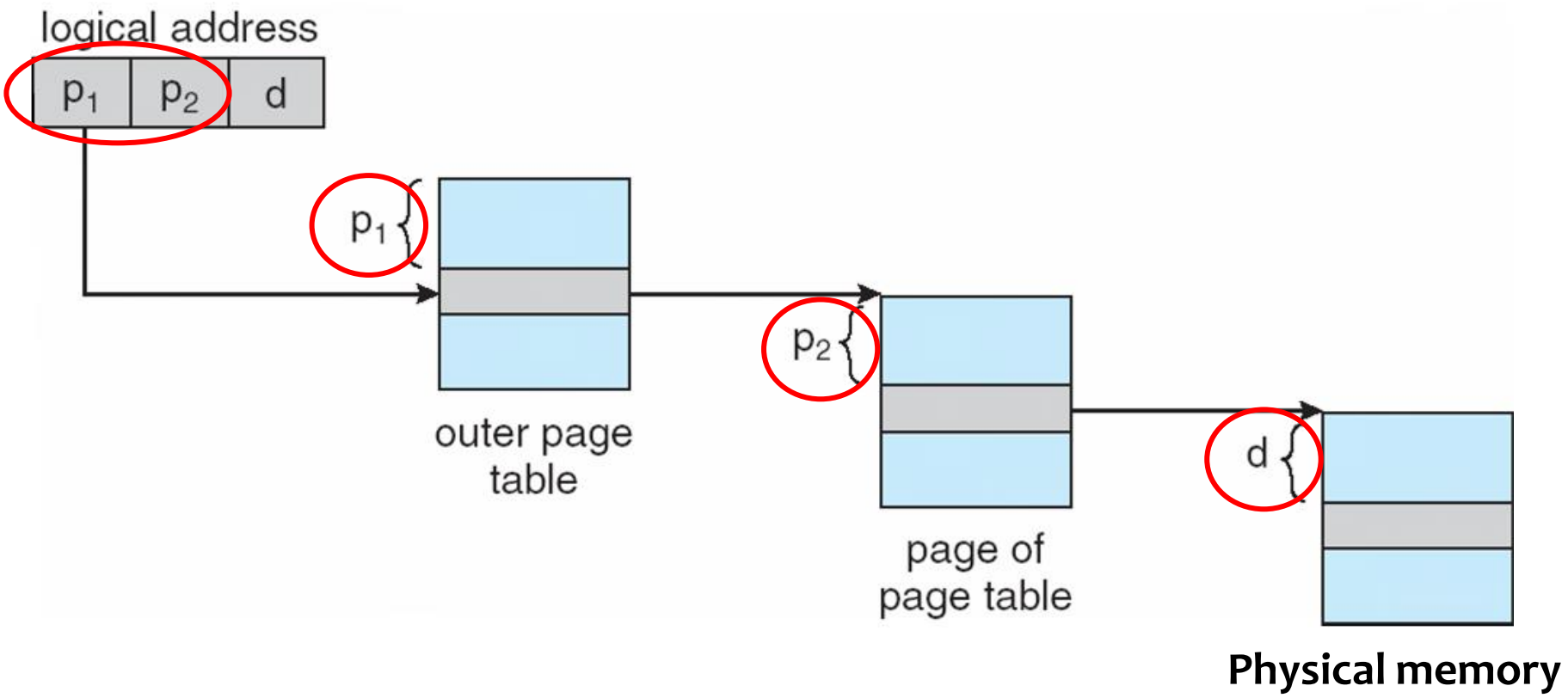
- a 12-bit page number
- a 10-bit page offset

Thus, a logical address is as

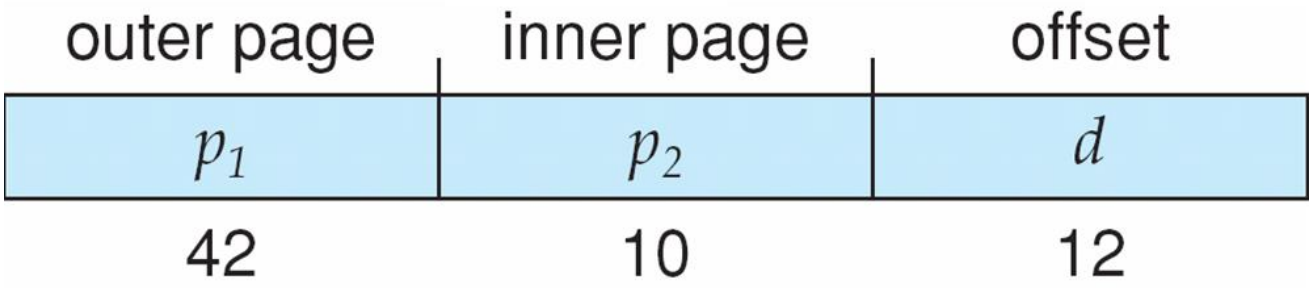


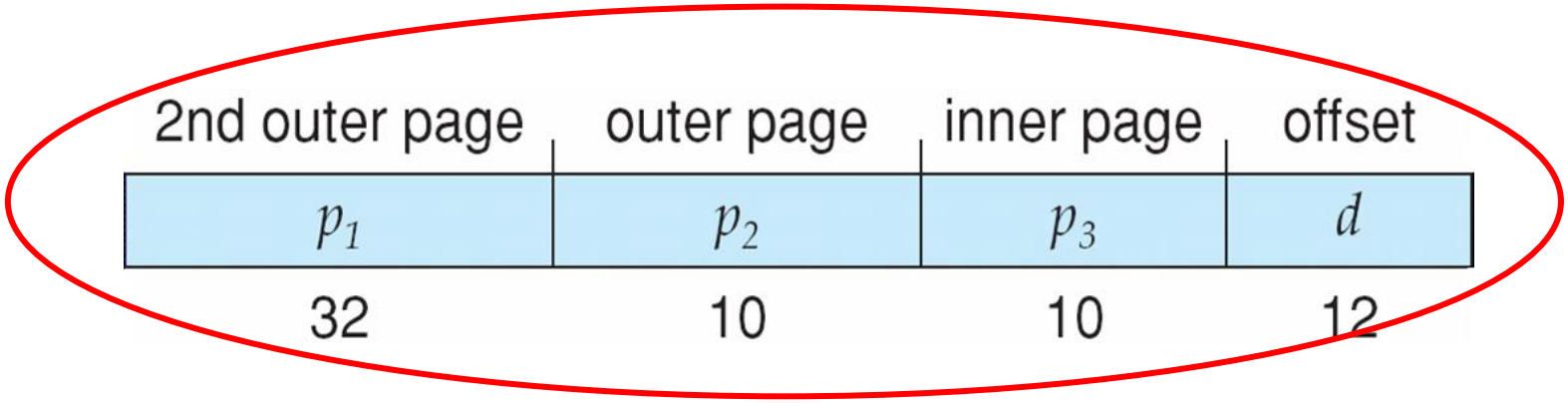
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address-Translation Scheme



Three-level Paging Scheme



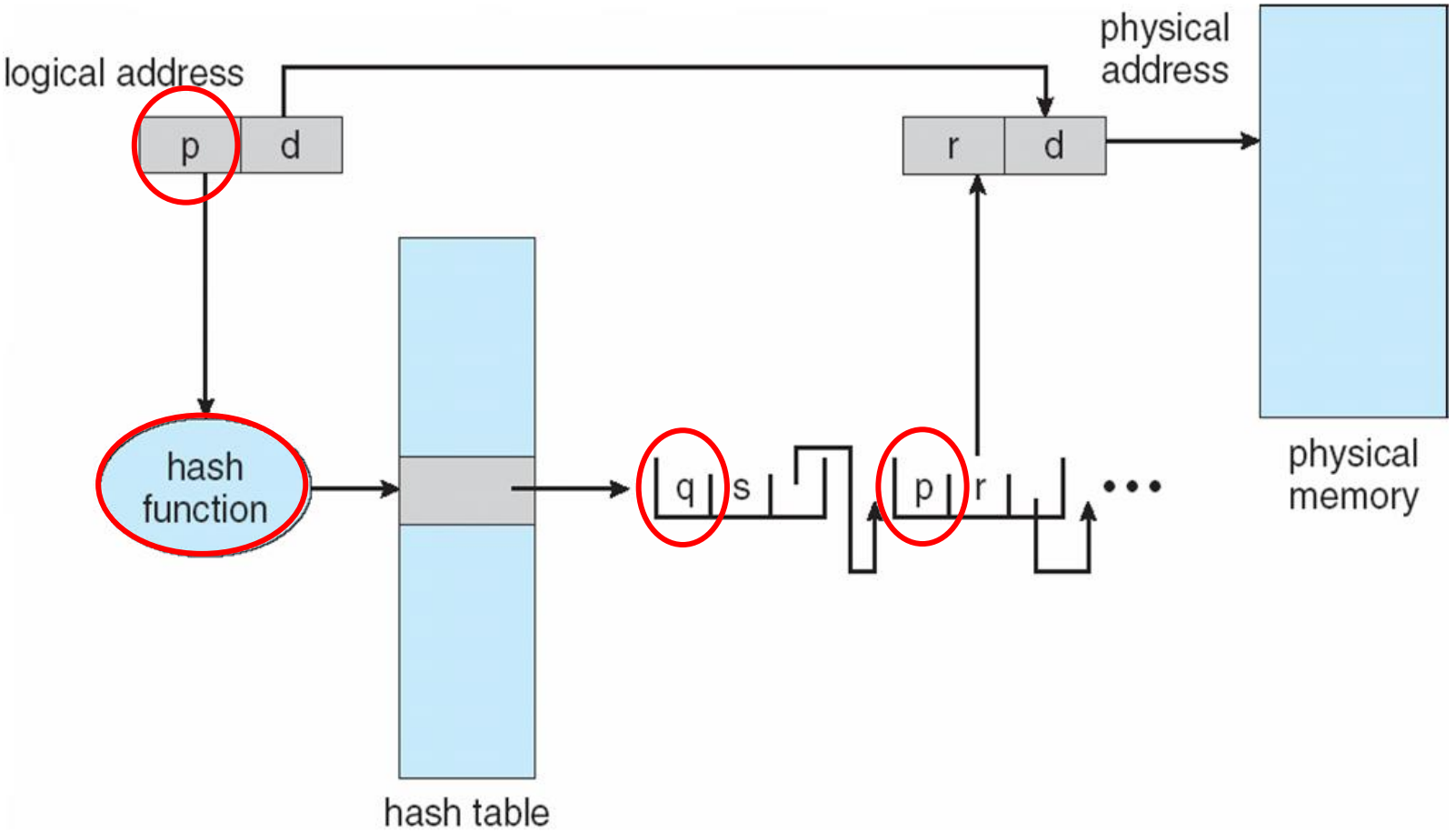


64-bit machine with 4K page

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Hashed Page Table



Inverted Page Table

One entry for each real page of memory

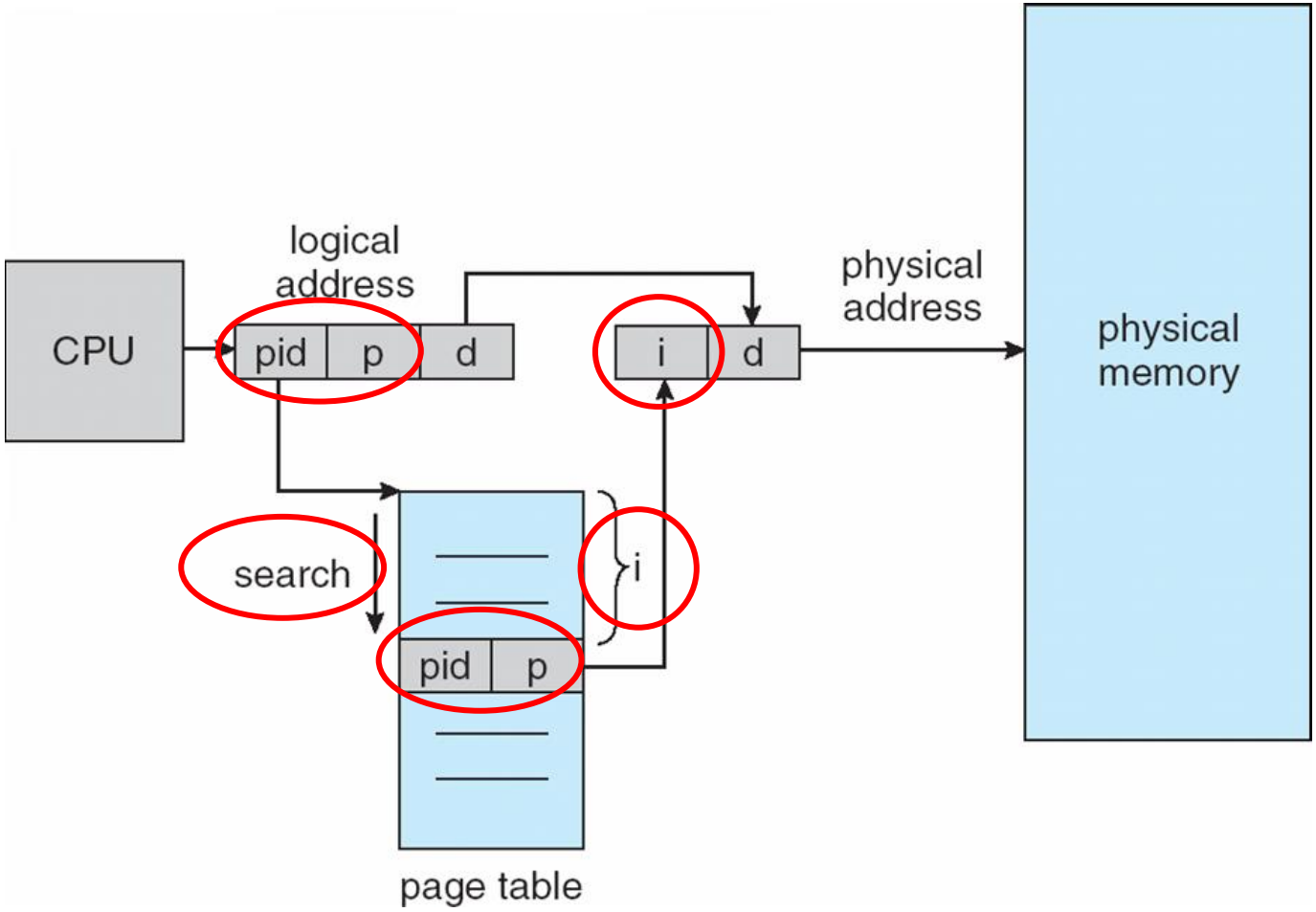
The page table is shared by all processes

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture



The search can be done sequentially, or
by hash function, or
by associative memory

Segmentation

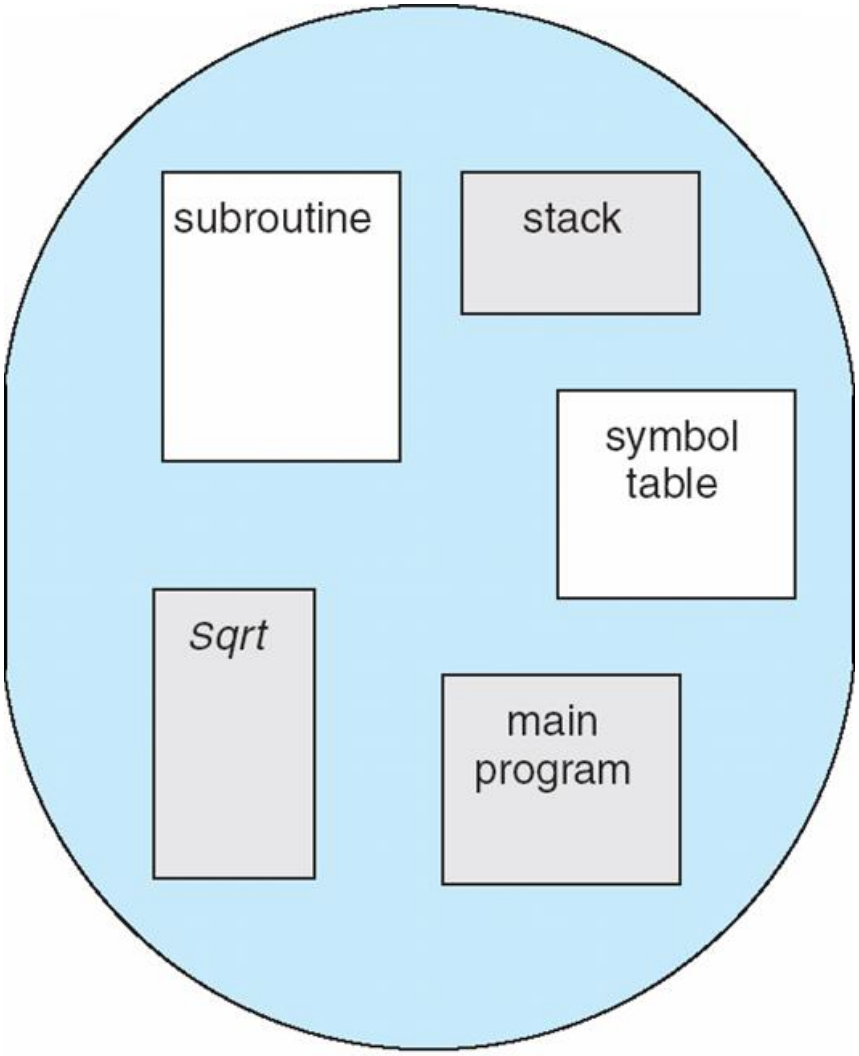
Memory-management scheme that supports user view of memory

A program is a collection of segments

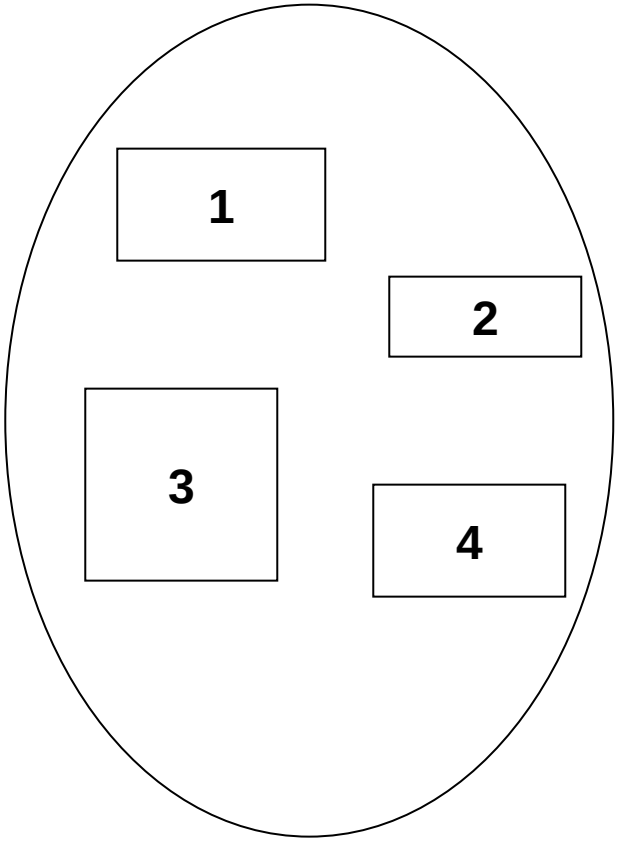
A segment is a logical unit such as:

- main program
- procedure
- function
- method
- object
- local variables, global variables
- common block
- stack
- symbol table
- arrays

User's View of a Program



Logical View of Segmentation



user space

physical memory space

Segmentation Architecture

Logical address consists of a two-tuple:

<segment-number, offset>,

Segment table – maps two-dimensional physical addresses; each table entry has:

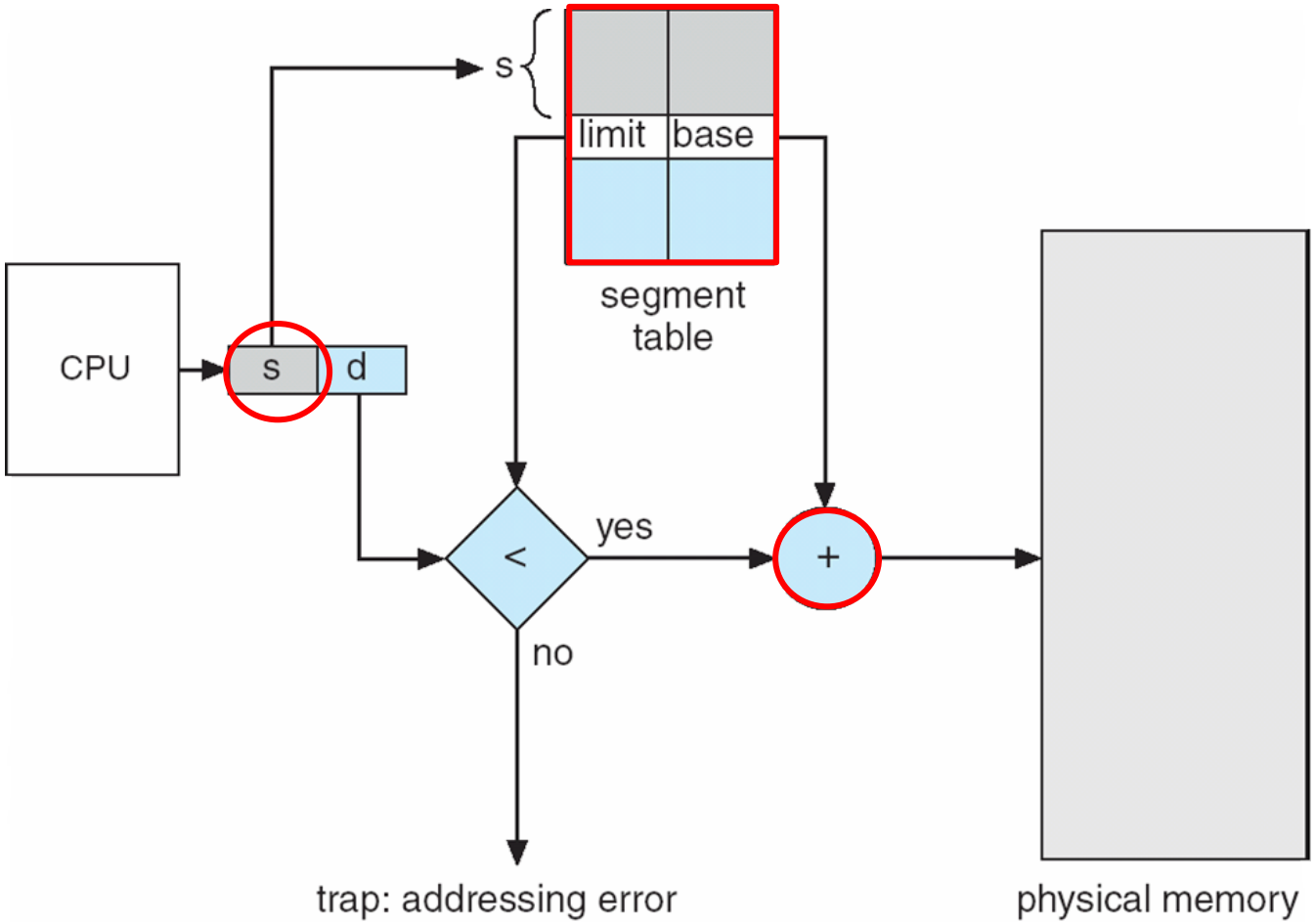
- base – contains the starting physical address where the segments reside in memory
- limit – specifies the length of the segment

Segment-table base register (STBR) points to the segment table's location in memory

Segment-table length register (STLR) indicates number of segments used by a program;

segment number *s* is legal if *s* < STLR

Segmentation Hardware



Segmentation Architecture (Cont.)

Protection

With each entry in segment table associate:

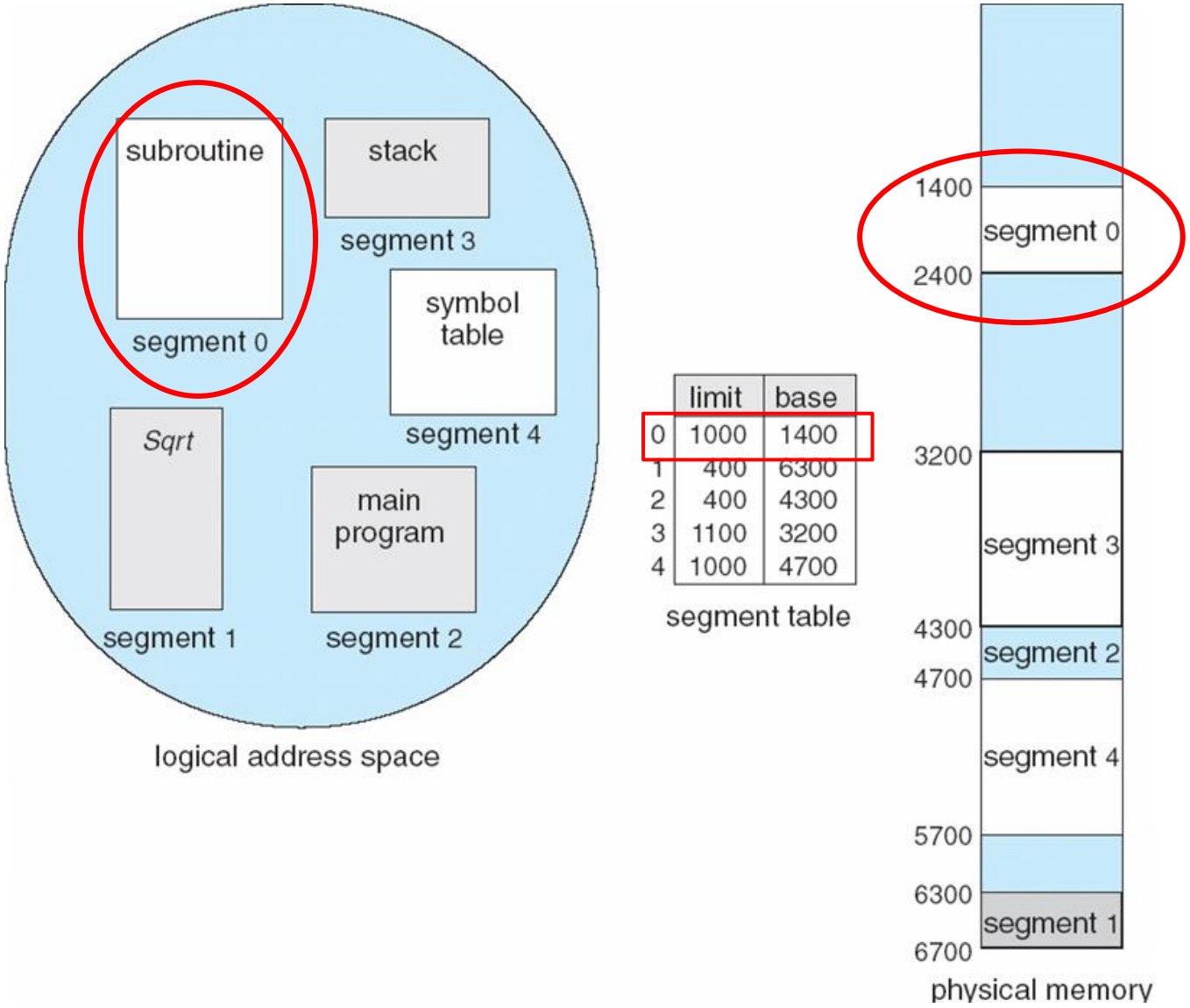
- validation bit = 0 ⇒ illegal segment
- read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level

Since segments vary in length, **memory allocation is a dynamic storage-allocation problem**

A segmentation example is shown in the following diagram

Example of Segmentation

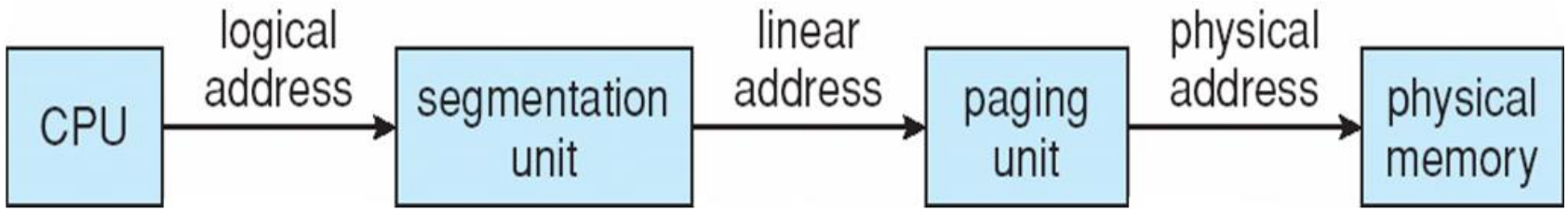


Example: The Intel Pentium

Supports both segmentation and segmentation with paging

CPU generates logical address

- Given to segmentation unit
  - Which produces linear addresses
- Linear address given to paging unit
  - Which generates physical address in main memory
  - Paging units form equivalent of MMU



Pentium Segmentation

A segment is allowed to be as large as 4GB (32 bits), and the maximum number of segments per process is 16K.

The logical address space is divided into two partitions:

- The first partition up to 8K segments that are private to that process.
- The 2nd partition up to 8K segments that are shared among all processes.

Local Descriptor Table (LDT): Information about the 1st partition

Global Descriptor Table (GDT): Information about the 2nd partition.

Each entry of LDT and GDT is an 8-byte descriptor of a particular segment.

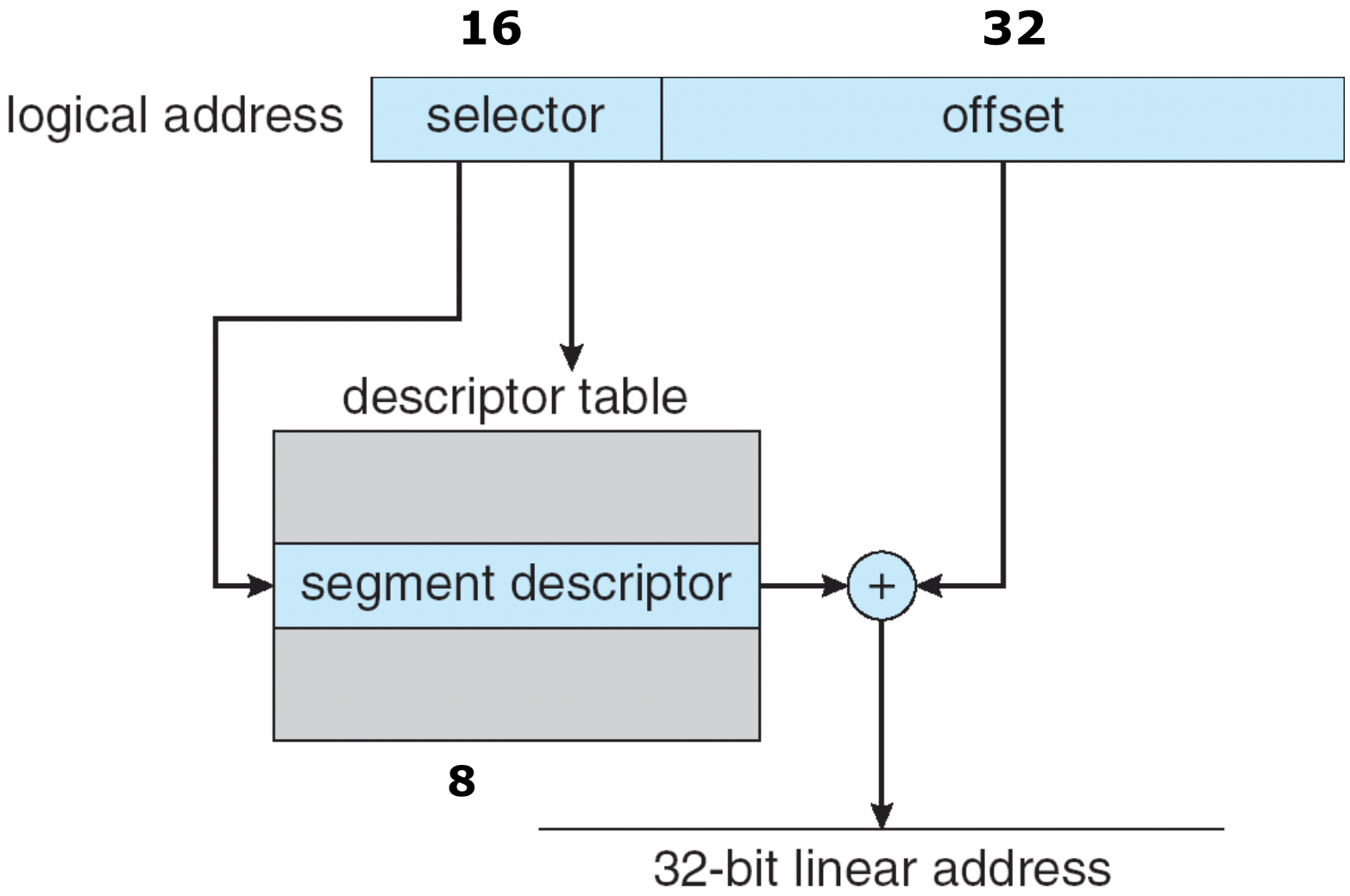
Pentium Segmentation

The logical address is a pair (selector, offset), where the selector is a 16-bit number and offset is a 32-bit number:

The machine has six segment registers, allowing six segments to be addressed at any one time by a process.

The linear address is 32 bits long and is formed as follows.

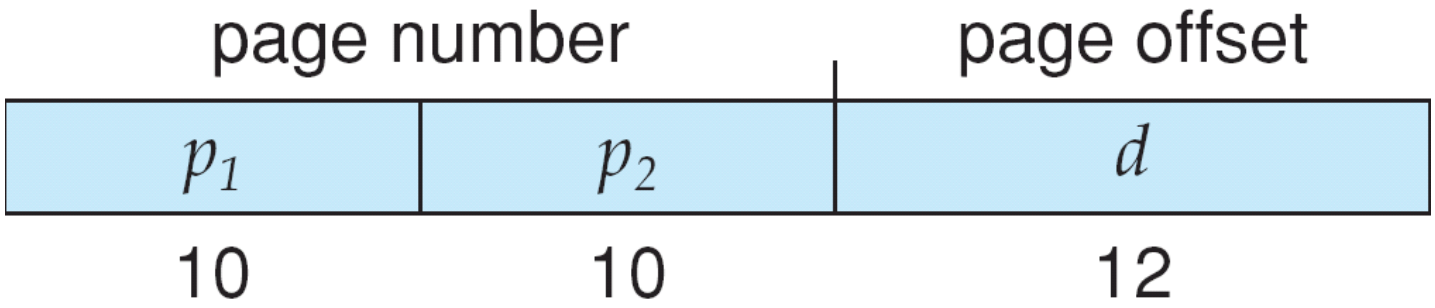
Intel Pentium Segmentation



Pentium Paging

A page is allowed to be as 4kB or 4MB.

For 4-KB pages, a two level paging scheme is used (32-bit linear address)

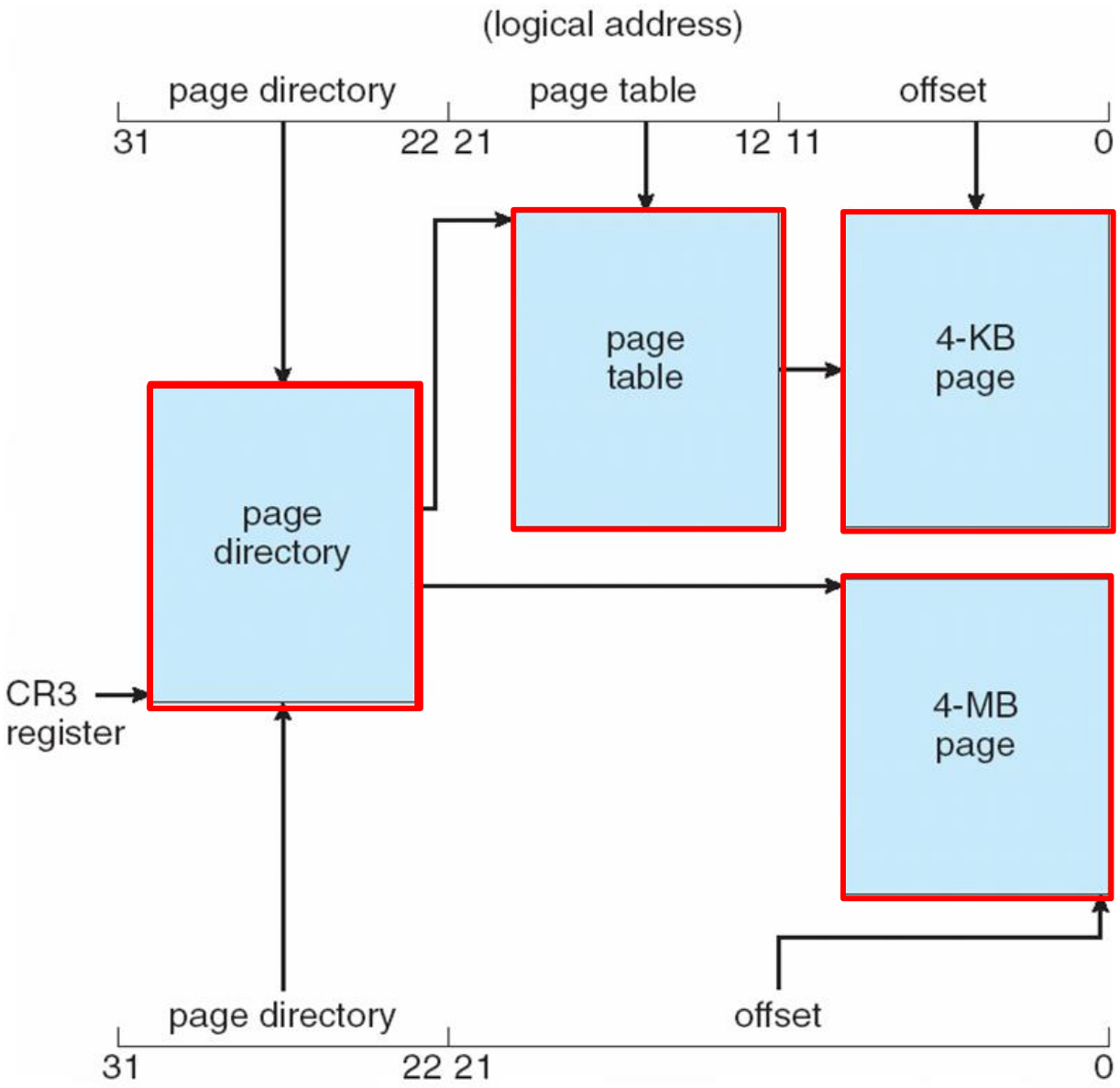


The address-translation scheme is shown as follows.

Page directory

Page size flag

Pentium Paging Architecture



Linux on Pentium Systems

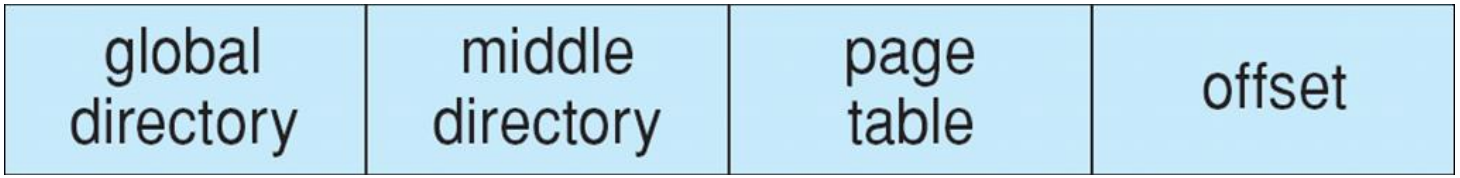
Linux does not reply on segmentations and used it minimally.

On the Pentium, Linux uses only six segments:

- A segment for kernel code
- A segment for kernel data
- A segment for user code
- A segment for user data
- A task-state segment (TSS)
- A default LDT segment

Linear Address in Linux

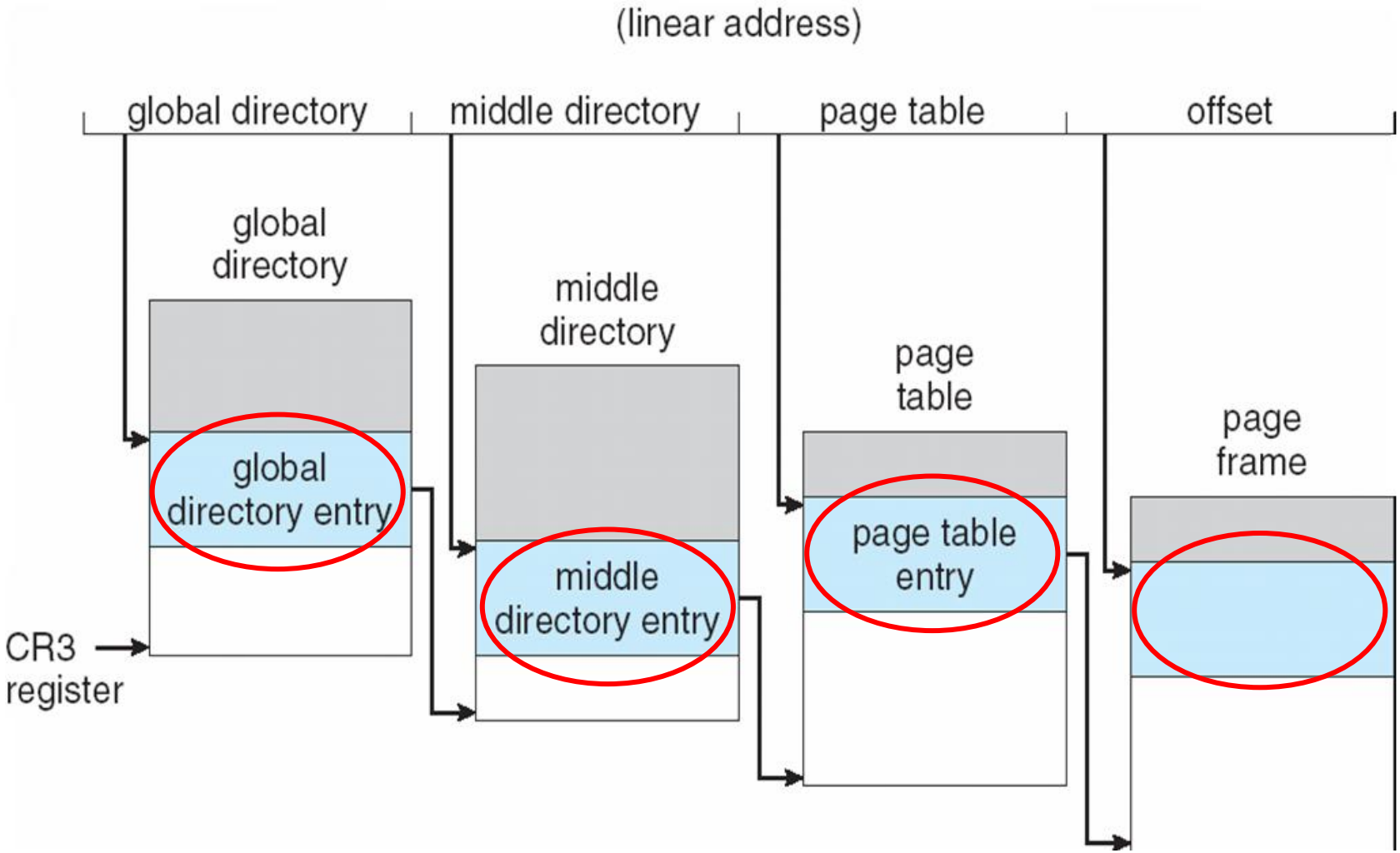
- The linear address in Linux is broken into four parts:



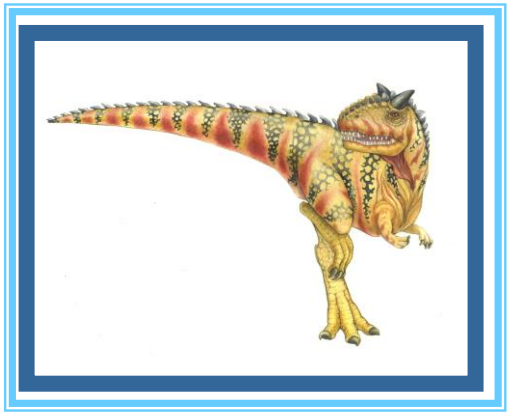
- Each task in Linux has its own set of page tables and the CR3 register points to the global directory for the task currently executing.

- During a context switch, the value of CR3 register is restored in the TSS segments of the tasks involved in the context switch.

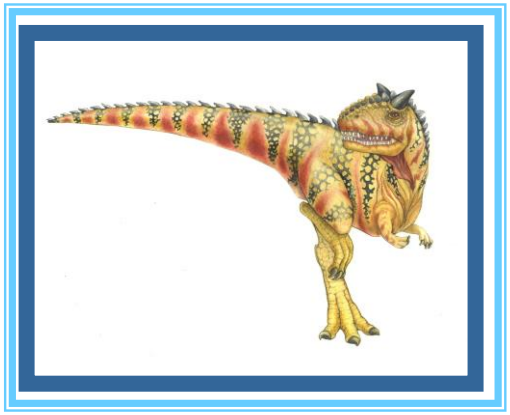
Three-level Paging in Linux



End of Chapter 8



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

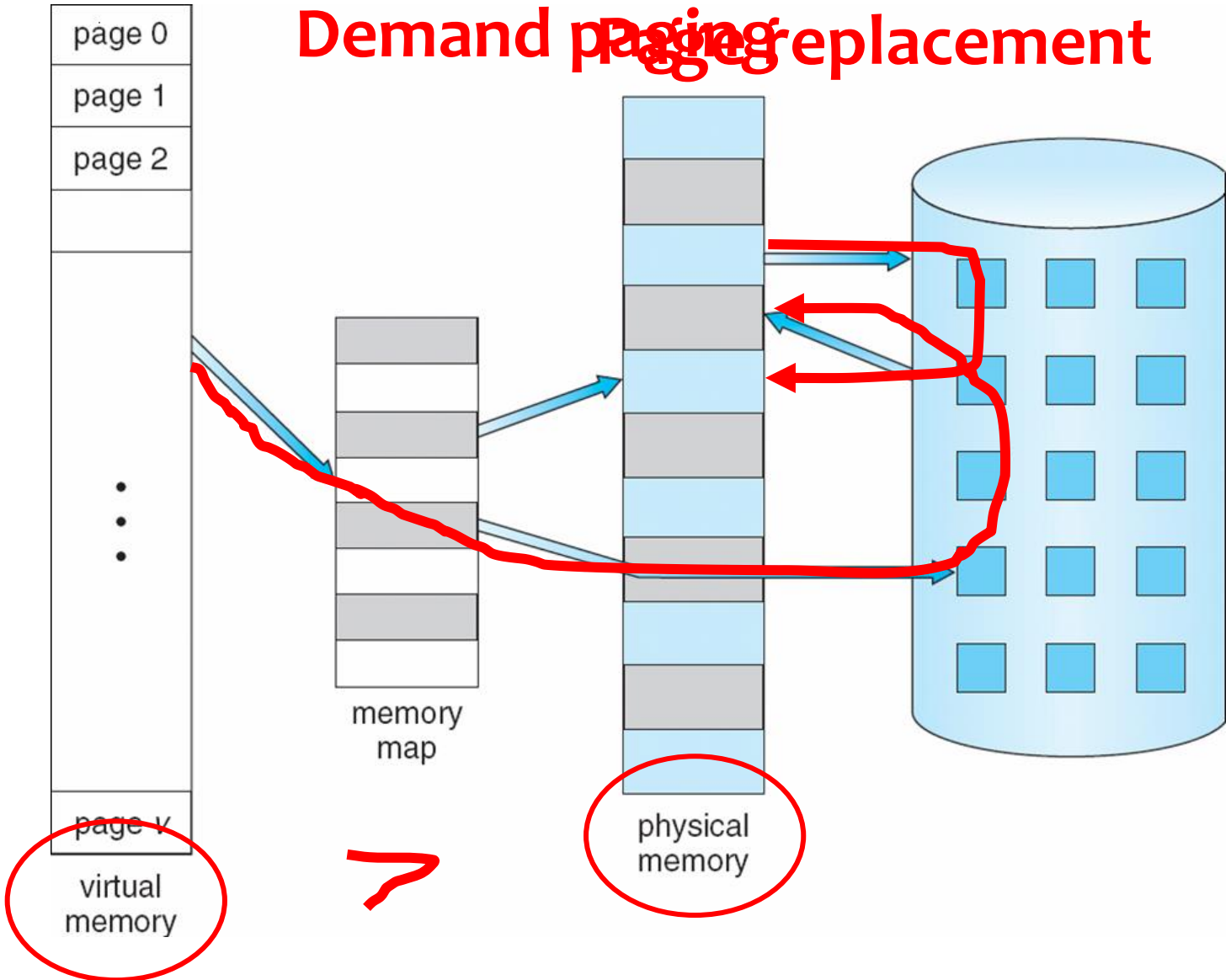
Virtual memory – separation of user logical memory from physical memory.

- 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

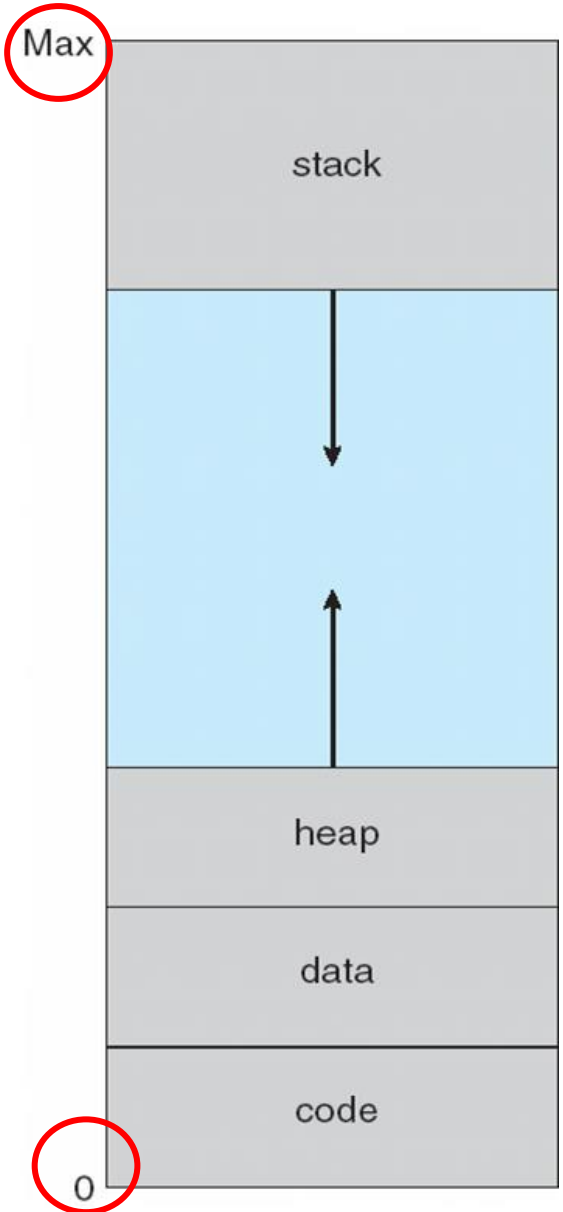
Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

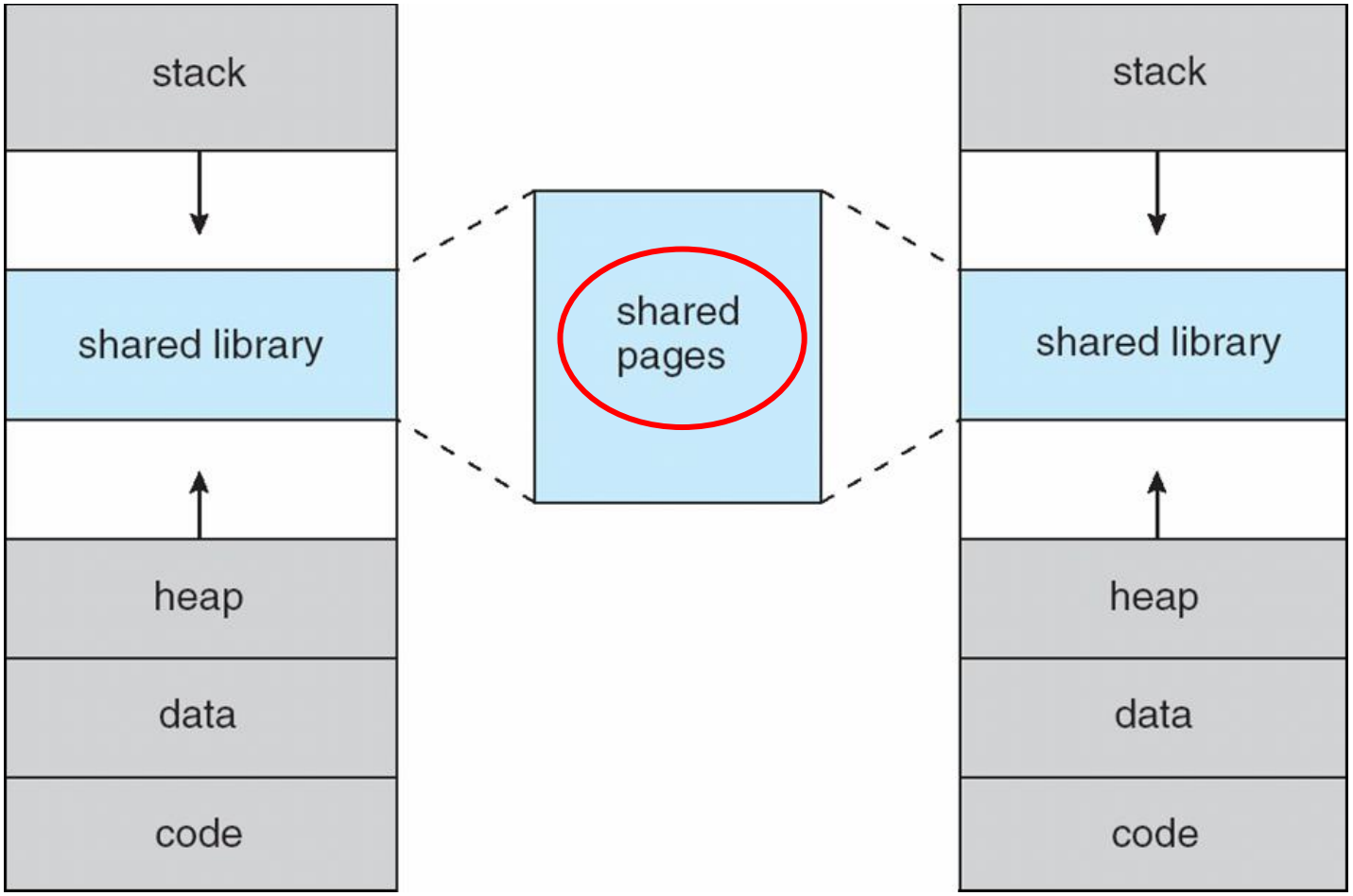
Virtual Memory > Physical Memory



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

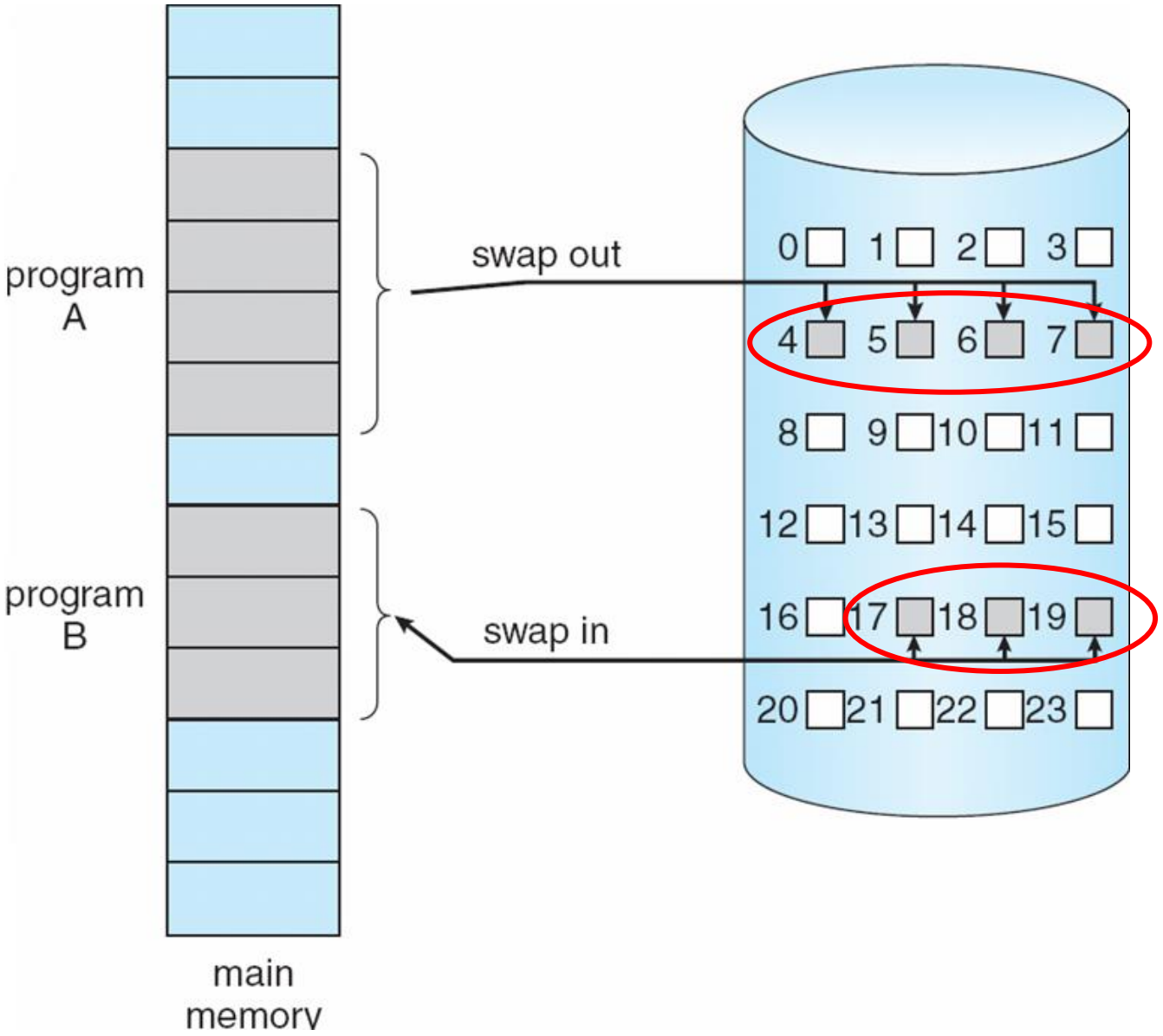
Page is needed ⇒ reference to it

- 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

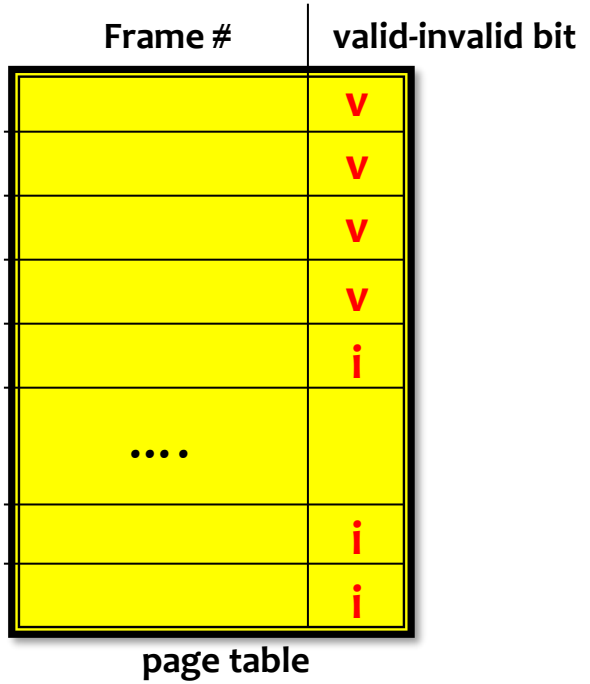


Valid-Invalid Bit

With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, i ⇒ not-in-memory)

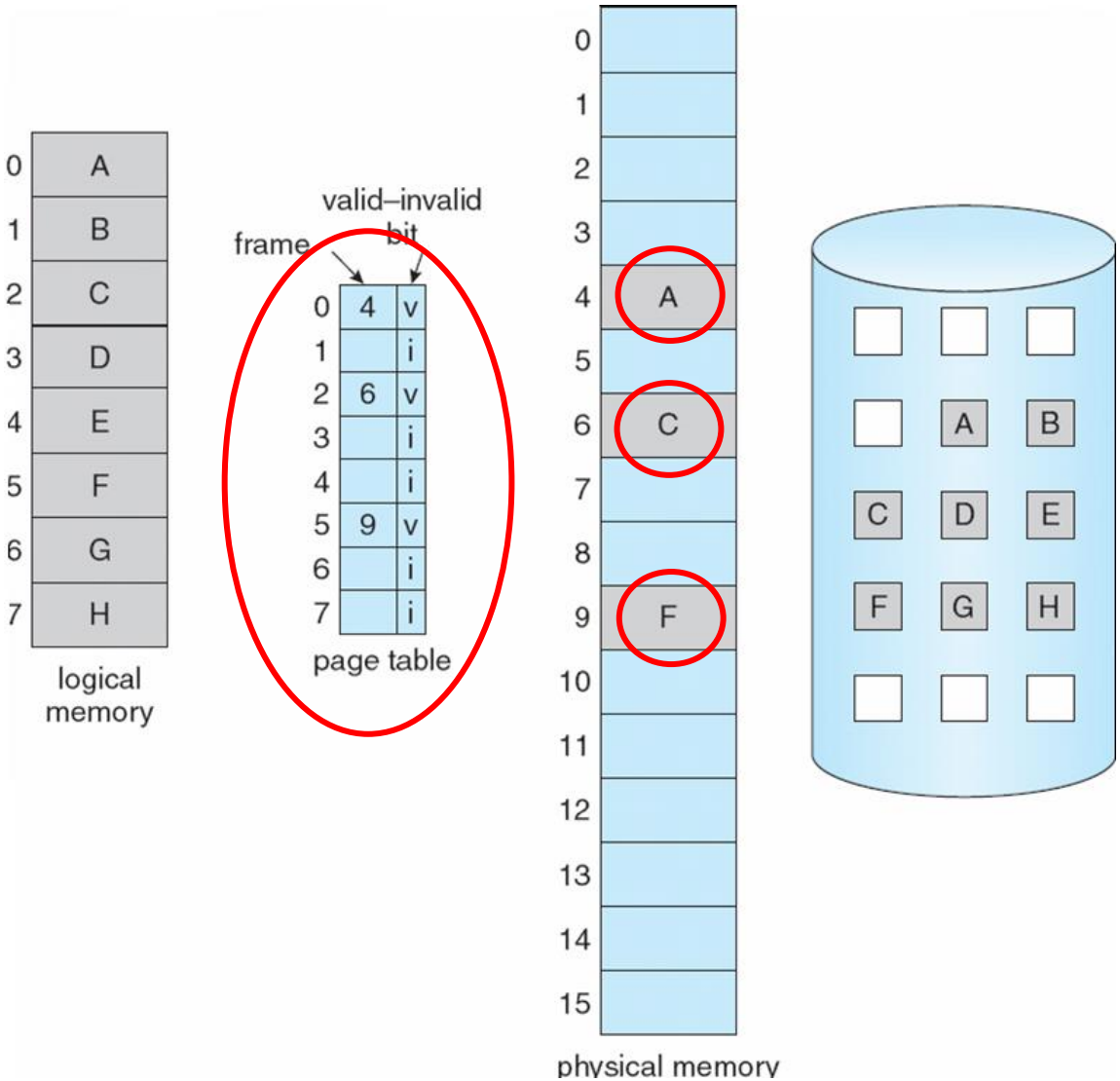
Initially valid–invalid bit is set to i on all entries

Example of a page table snapshot:



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:

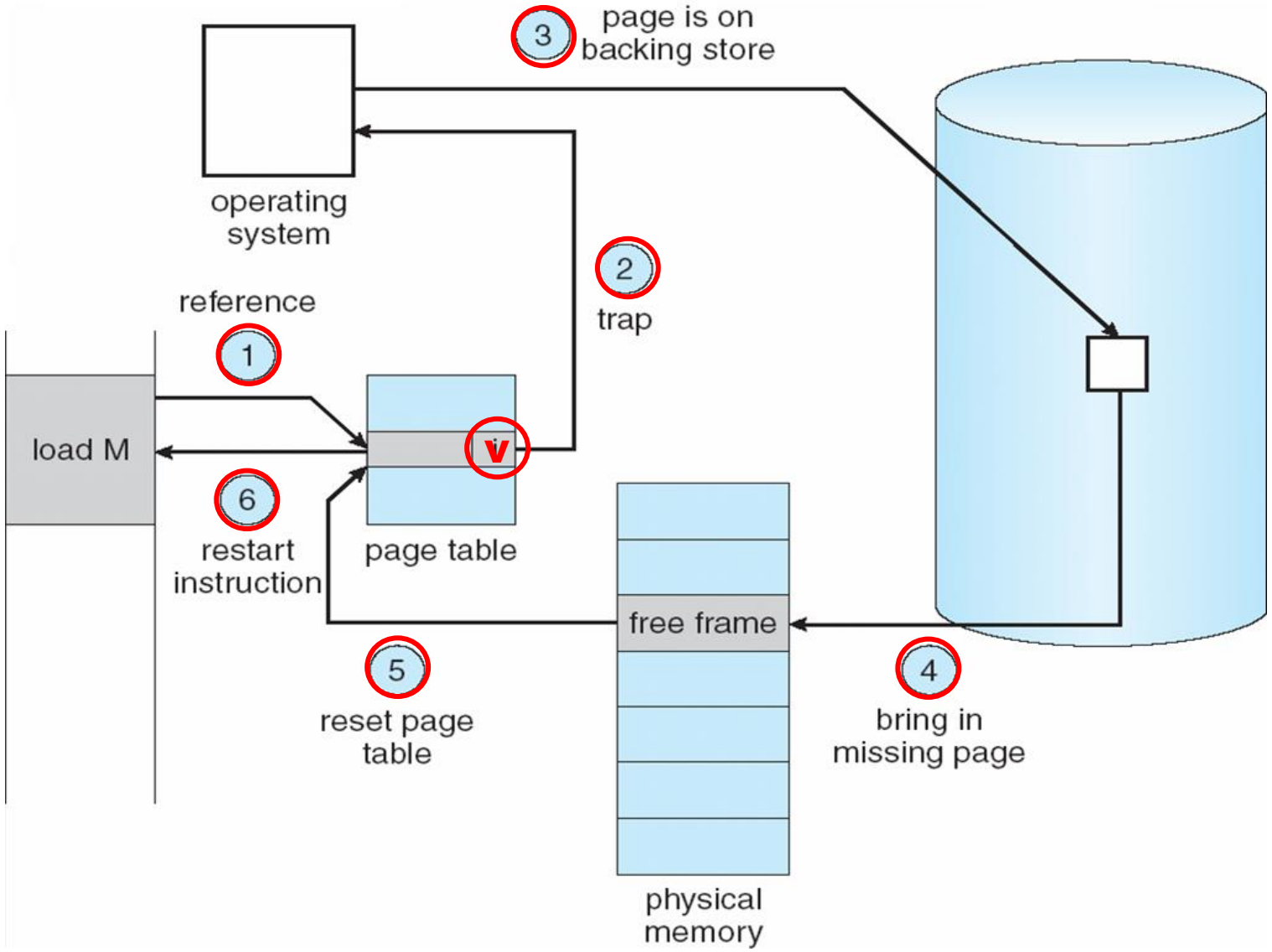
page fault

1. Operating system looks at another table to decide:

   Invalid reference ⇒ abort

   Just not in memory
2. Get empty frame 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 $0 \le p \le 1.0$

if $p = 0$ no page faults

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

Effective Access Time (EAT)

EAT = $(1 - p)$ x memory access

+ p (page fault overhead

+ swap page out

+ swap page in

+ restart overhead)

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

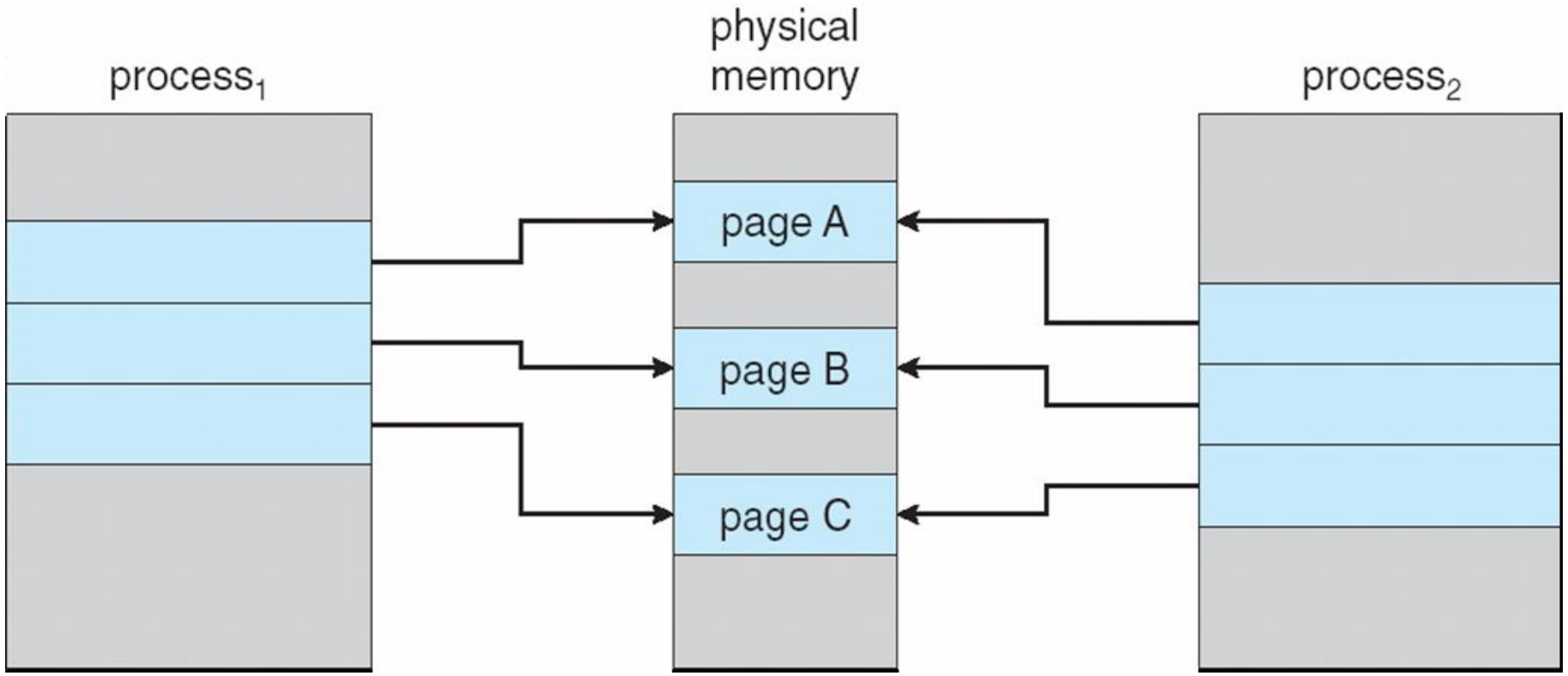
Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

If either process modifies a shared page, only then is the page copied

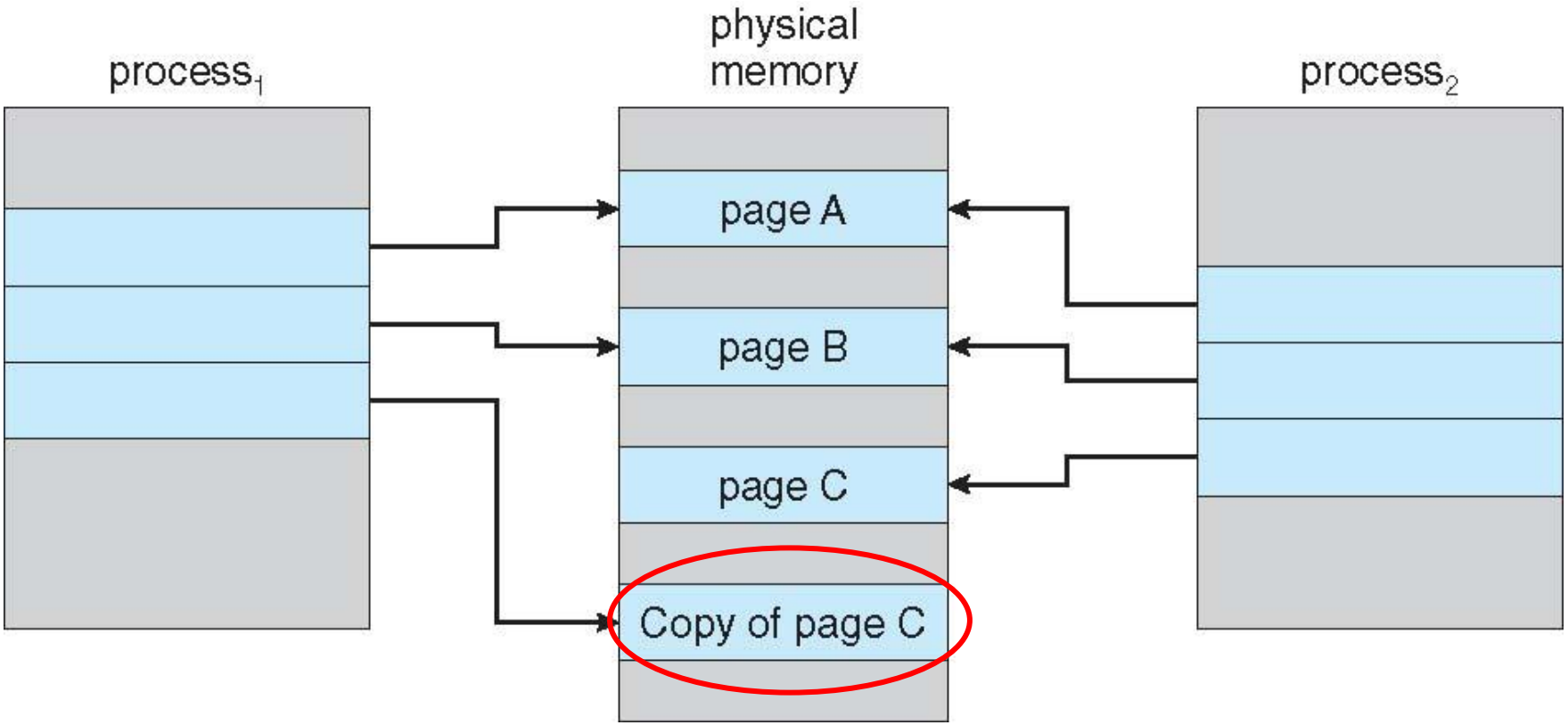
COW allows more efficient process creation as only modified pages are copied

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

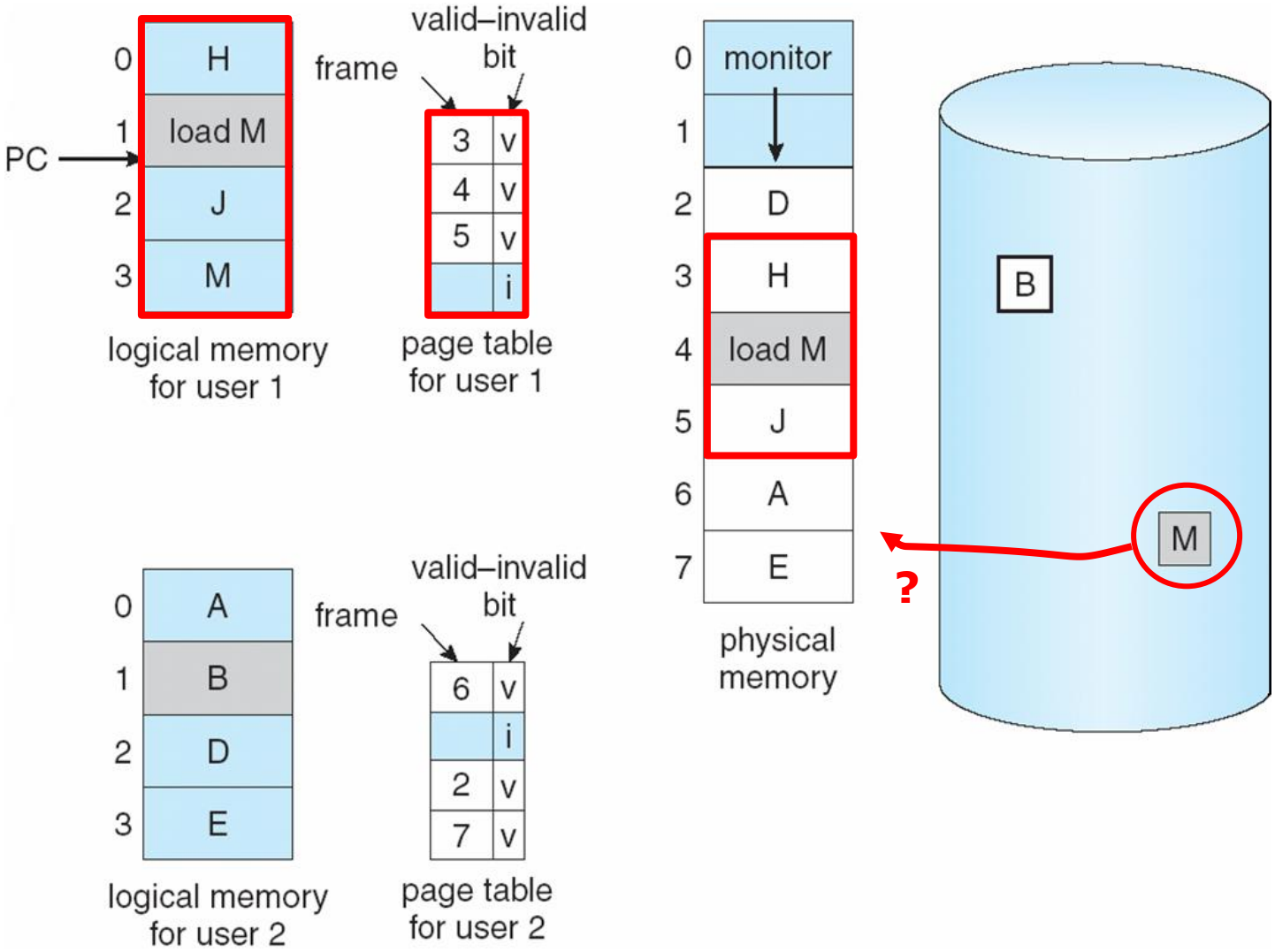
Page Replacement

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

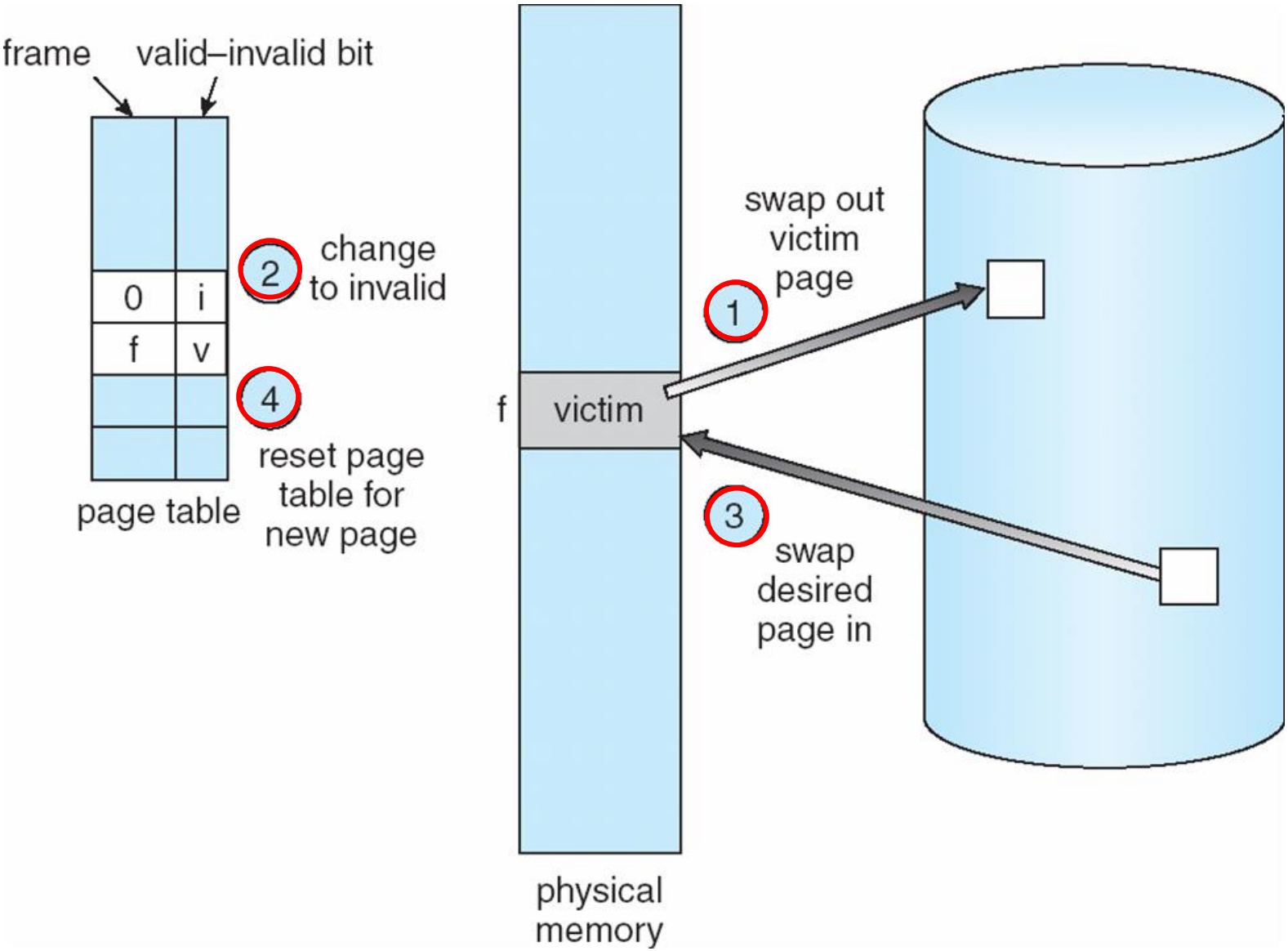
Need For Page Replacement



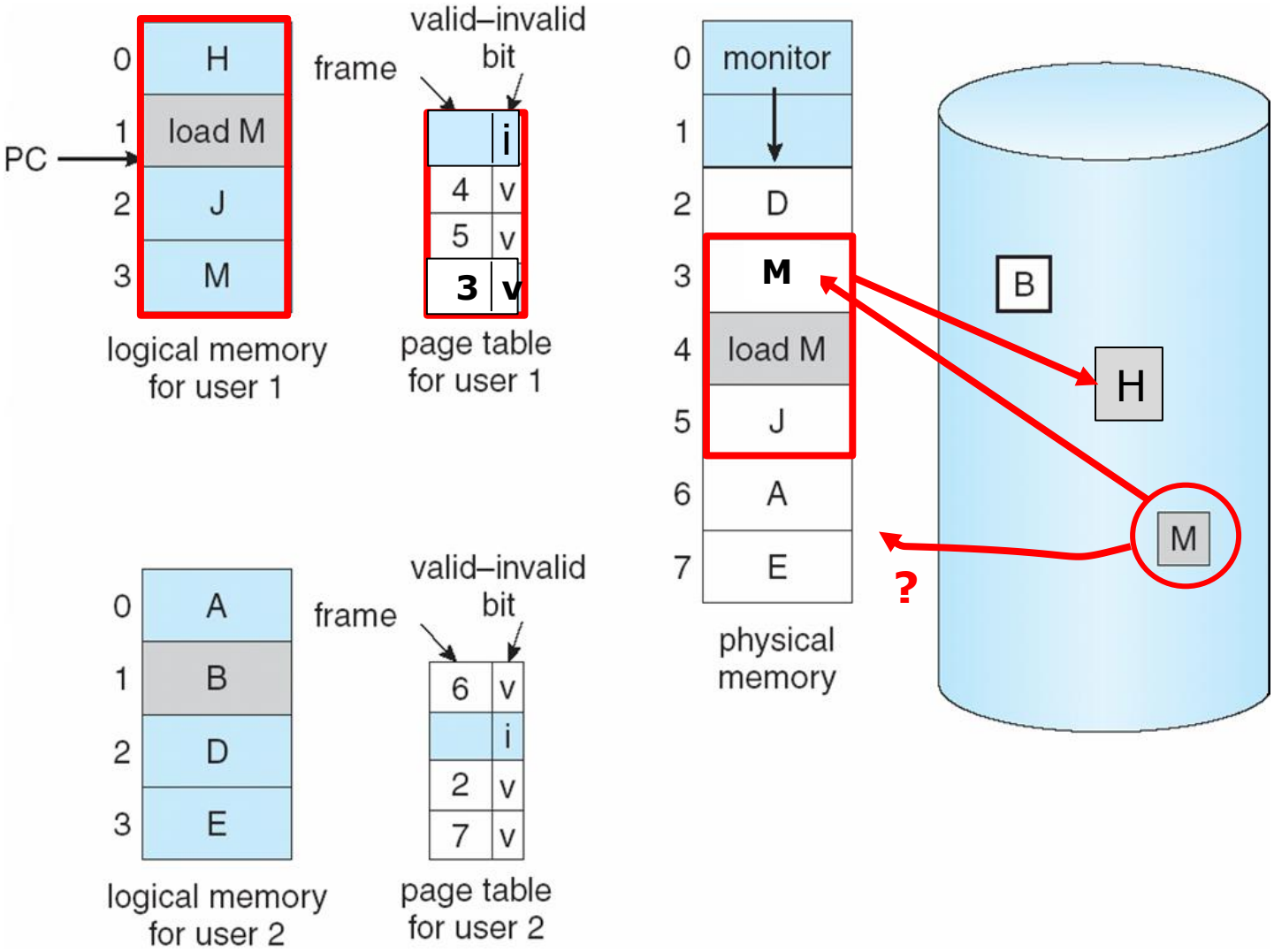
Basic Page Replacement

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

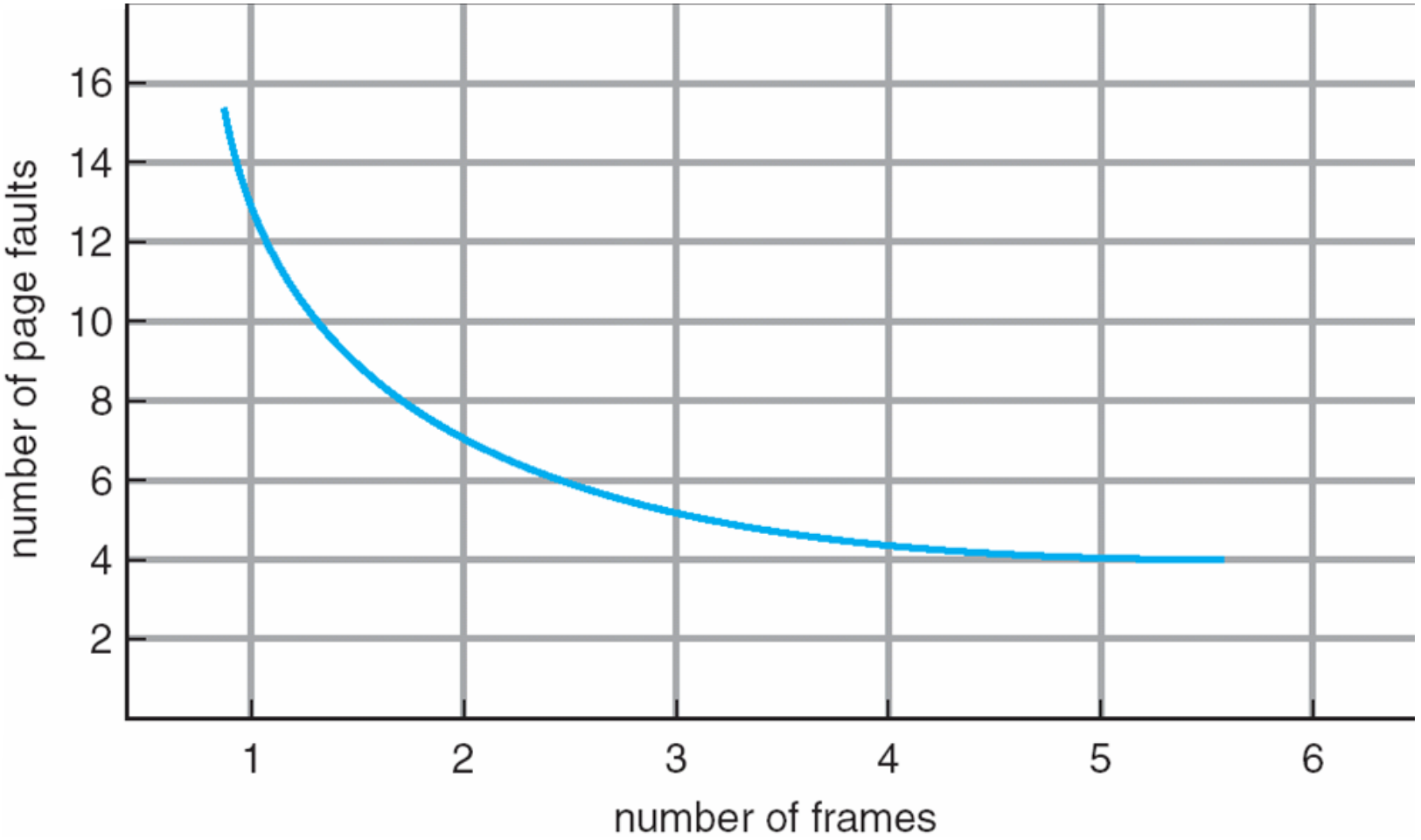
Want lowest page-fault rate

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)

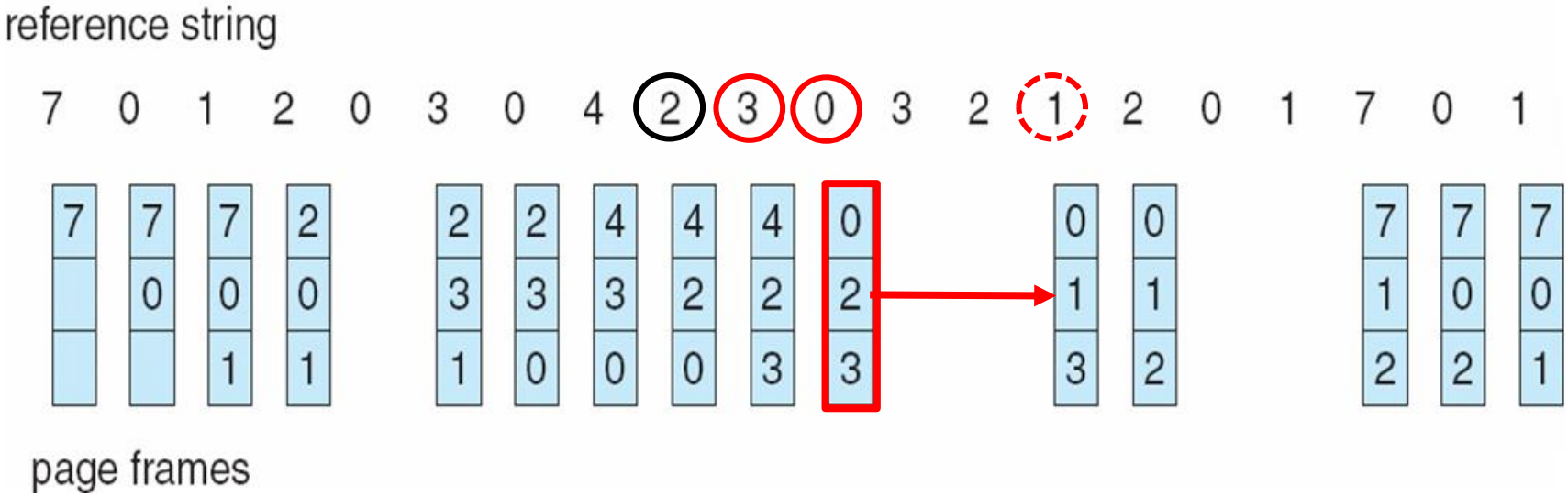
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

4 frames

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

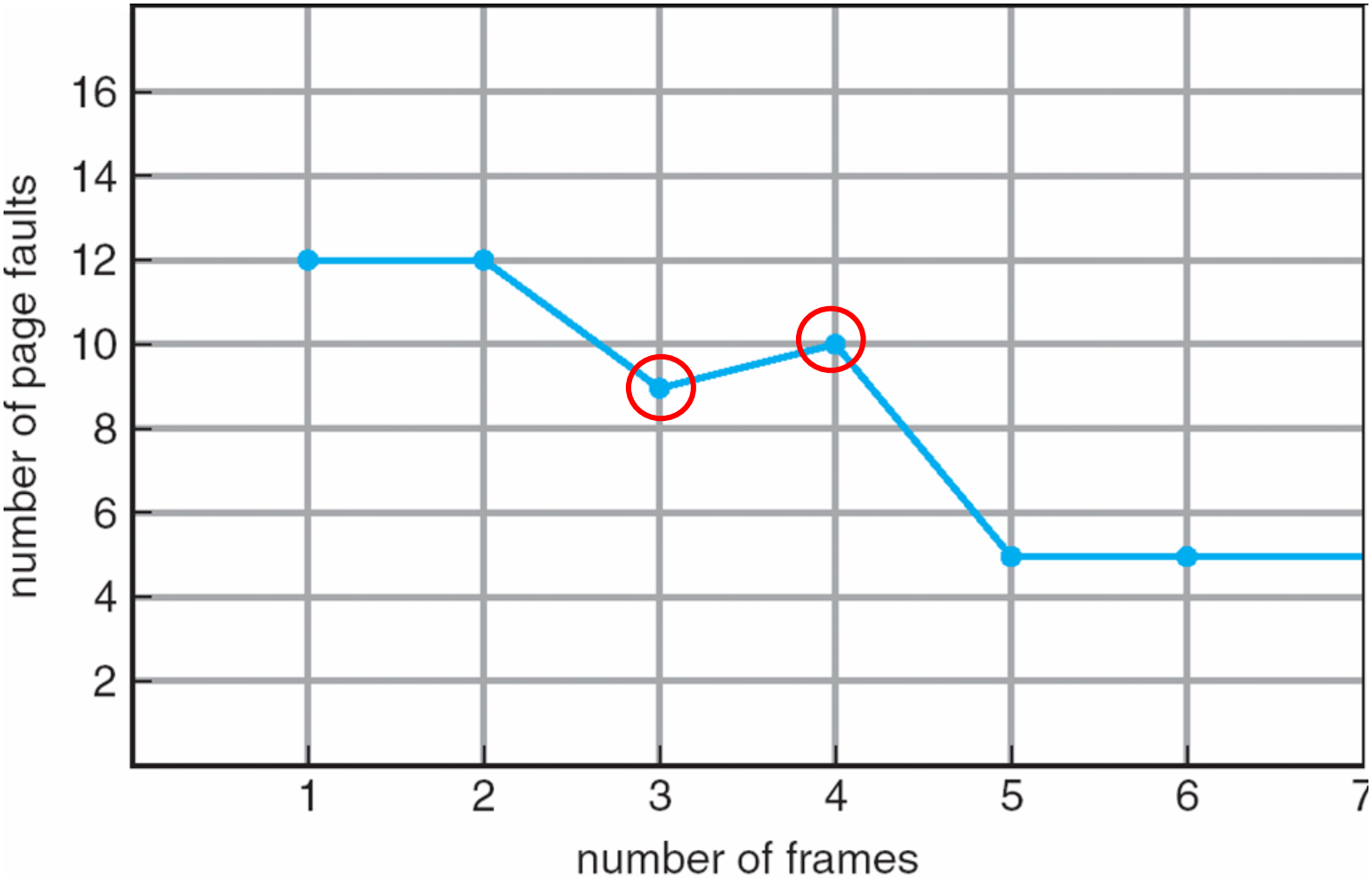
Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



Total number of page faults = 15

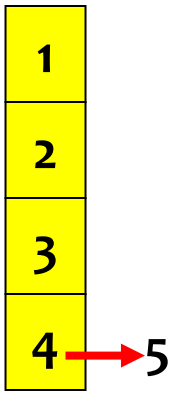
FIFO Illustrating Belady's Anomaly



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



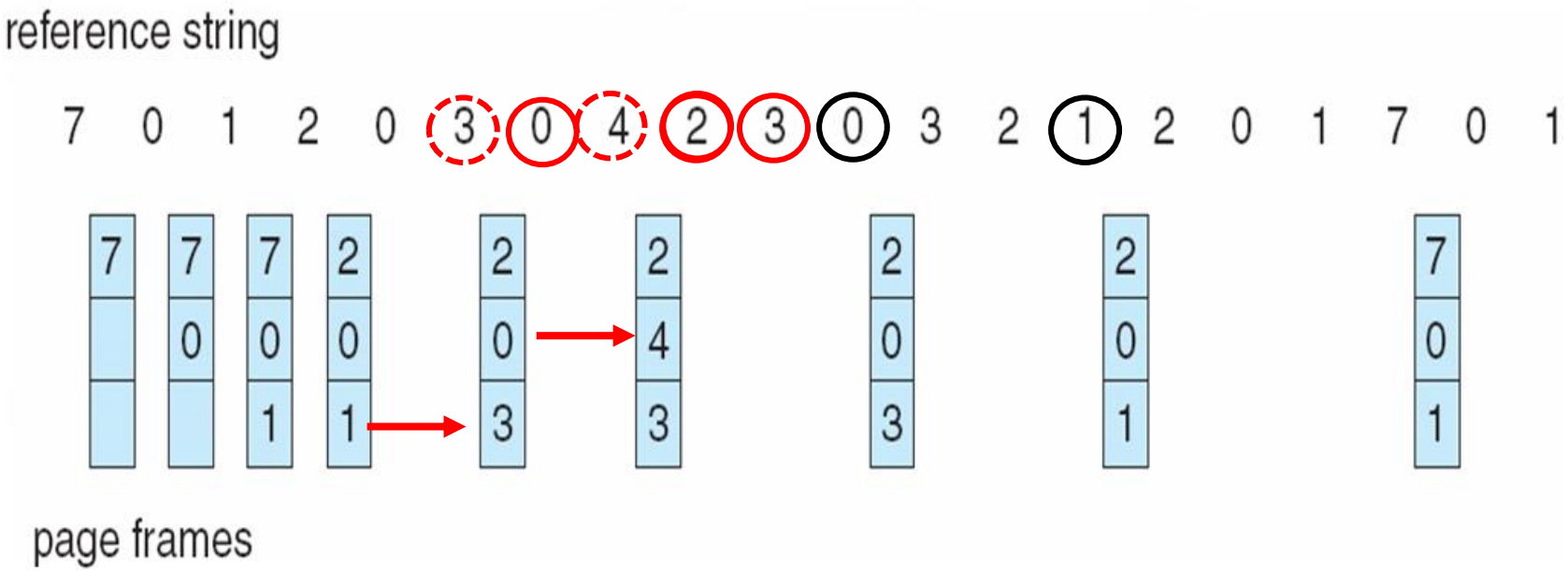
4

6 page faults

How do you know this?

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

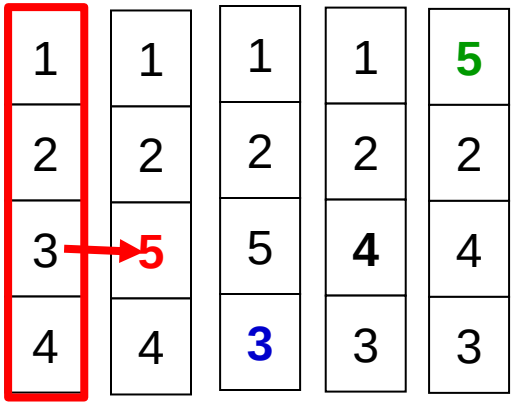
Optimal Page Replacement



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



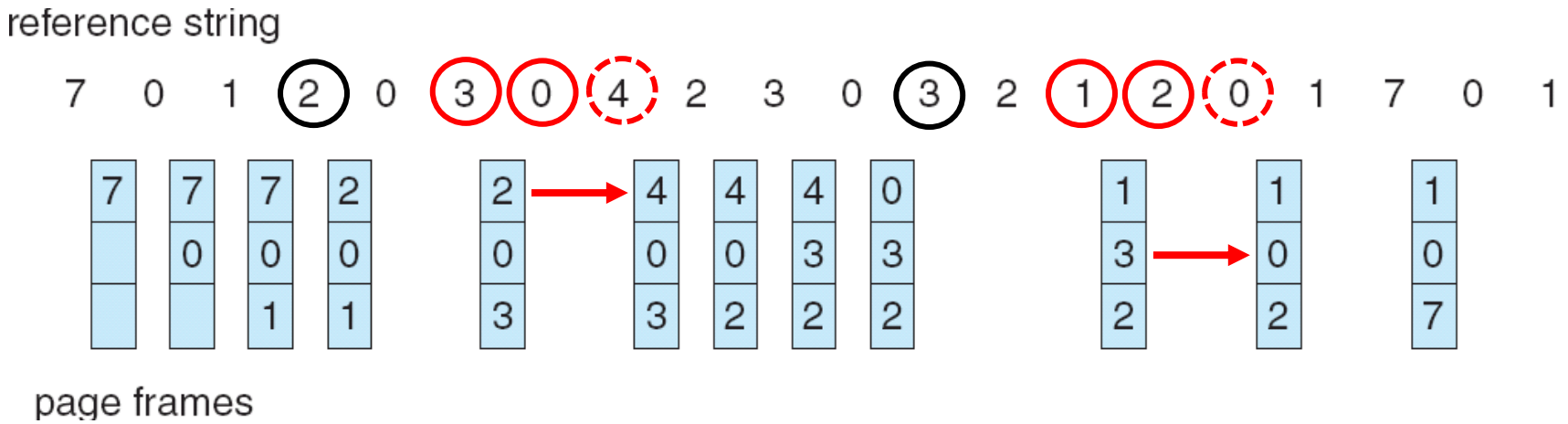
(最早之前用過)

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.)

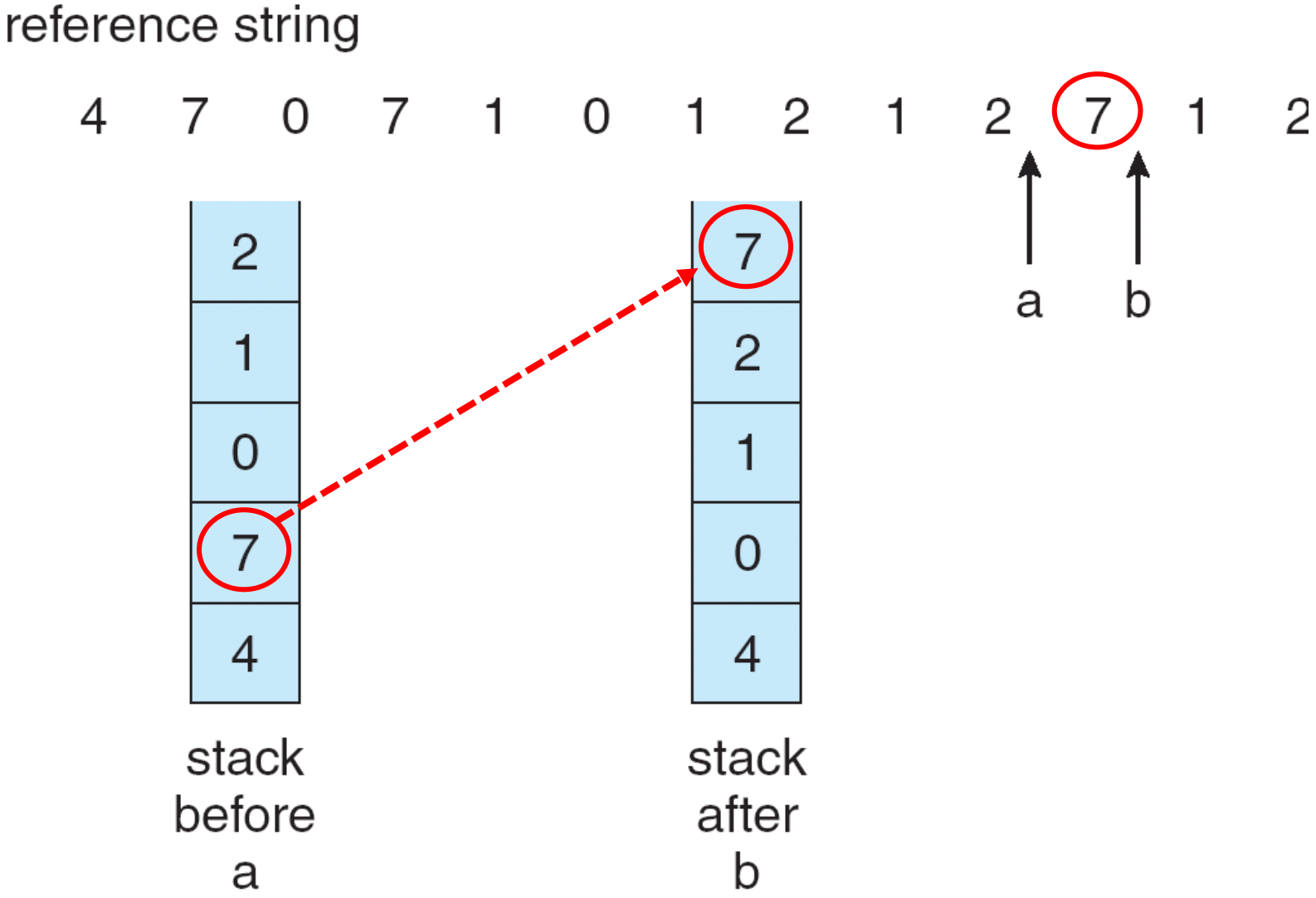
Stack implementation – keep a stack of page numbers in a double link form:

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

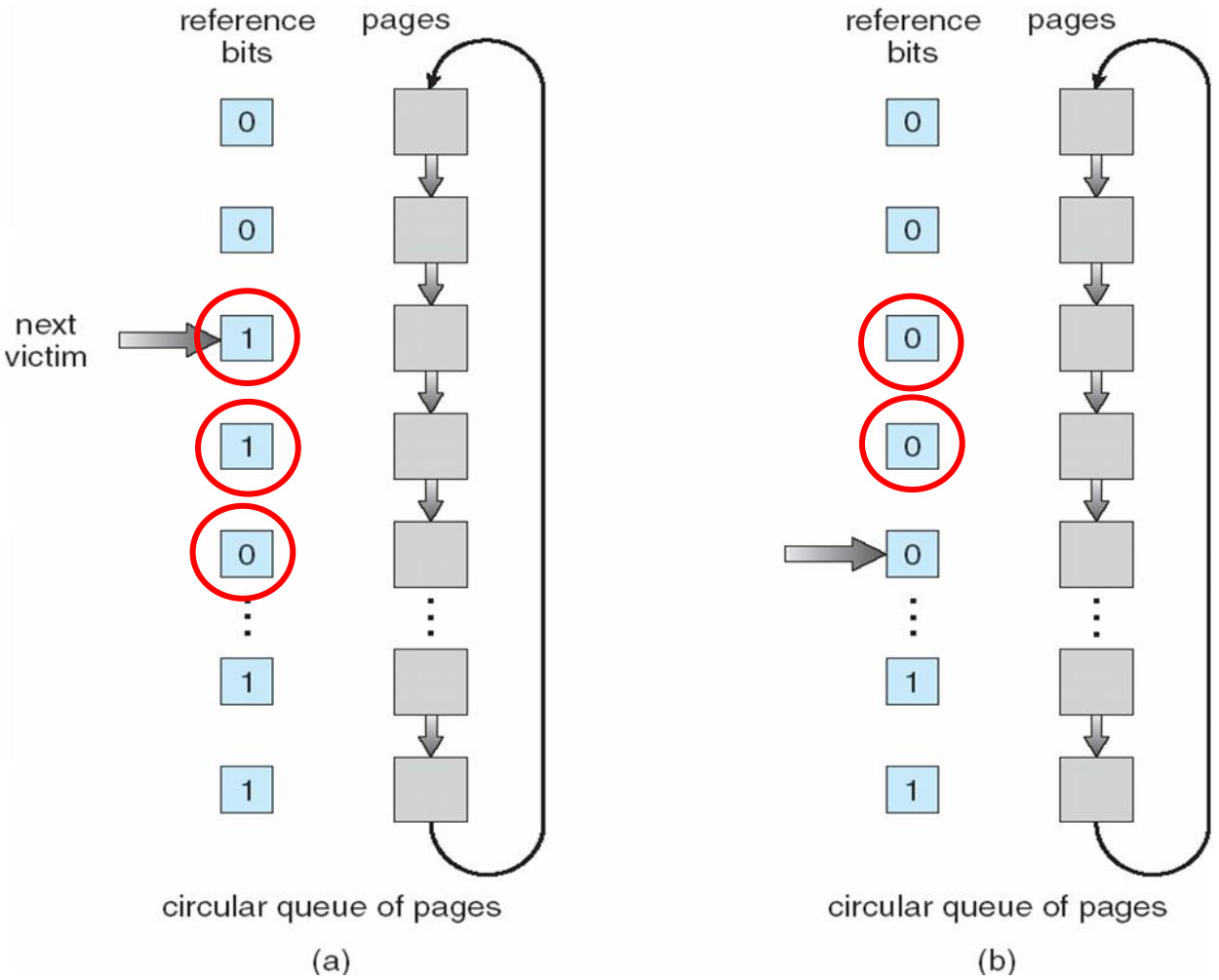
Need reference bit

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 = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

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

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

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

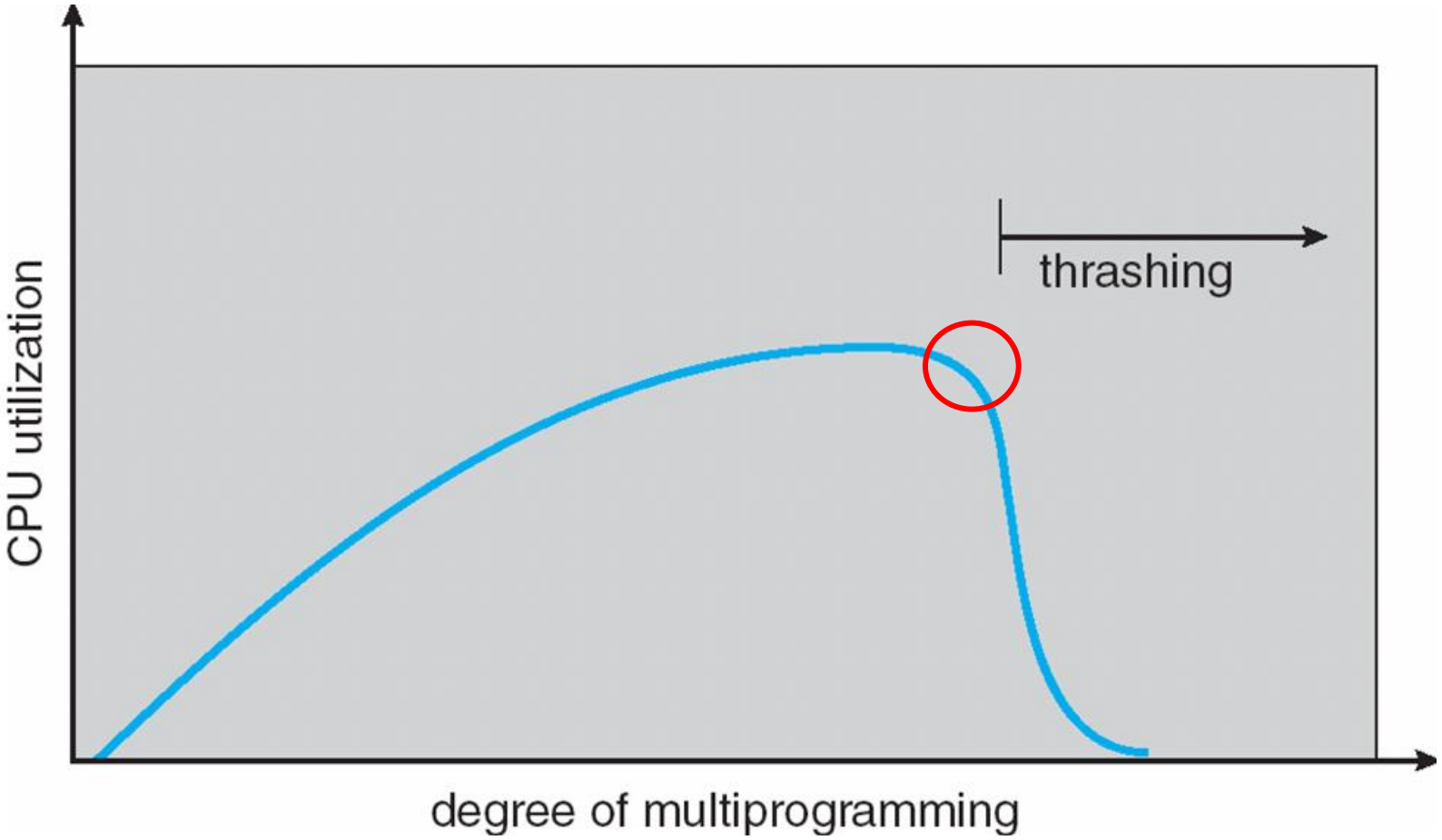
Thrashing

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 ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

Why does demand paging work?

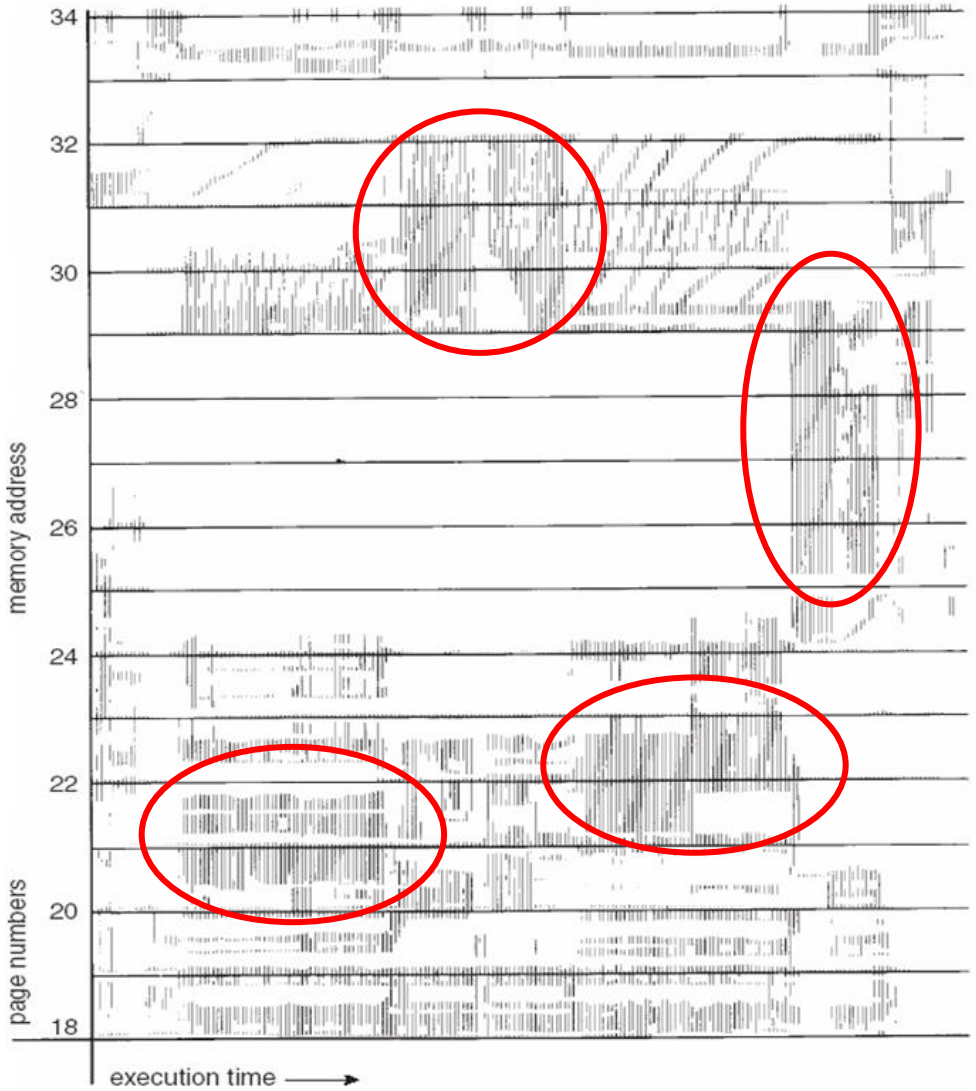
Locality model

- 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)

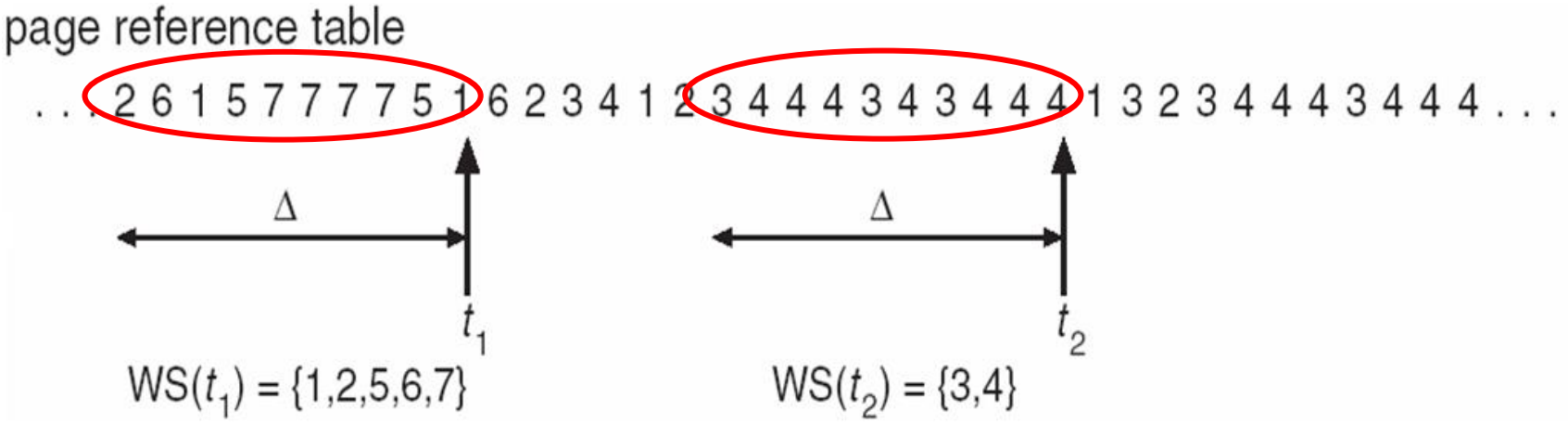
- if Δ too small will not encompass entire locality
- if Δ too large will encompass several localities
- if $\Delta = \infty \Rightarrow$ will encompass entire program

$D = \Sigma\ WSS_i \equiv$ 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: Δ = 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

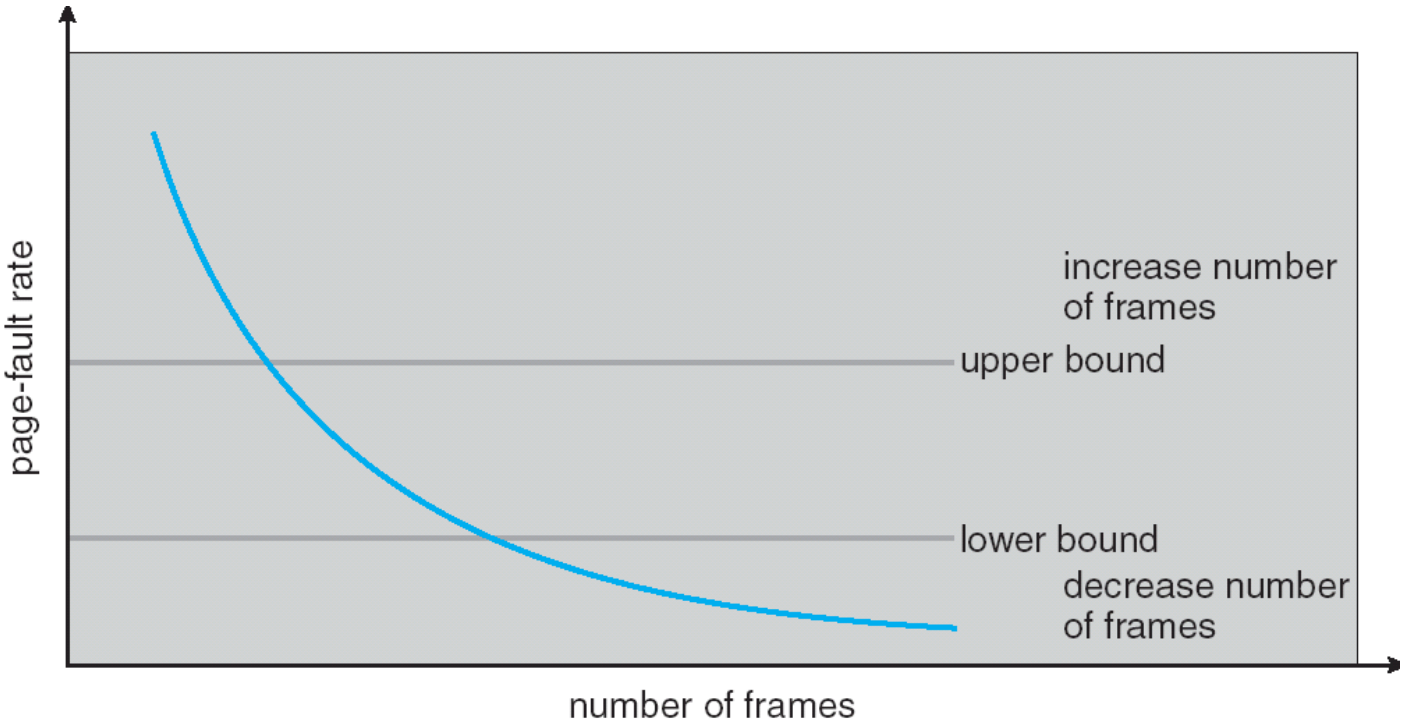
Why is this not completely accurate?

Improvement = 10 bits and interrupt every 1000 time units

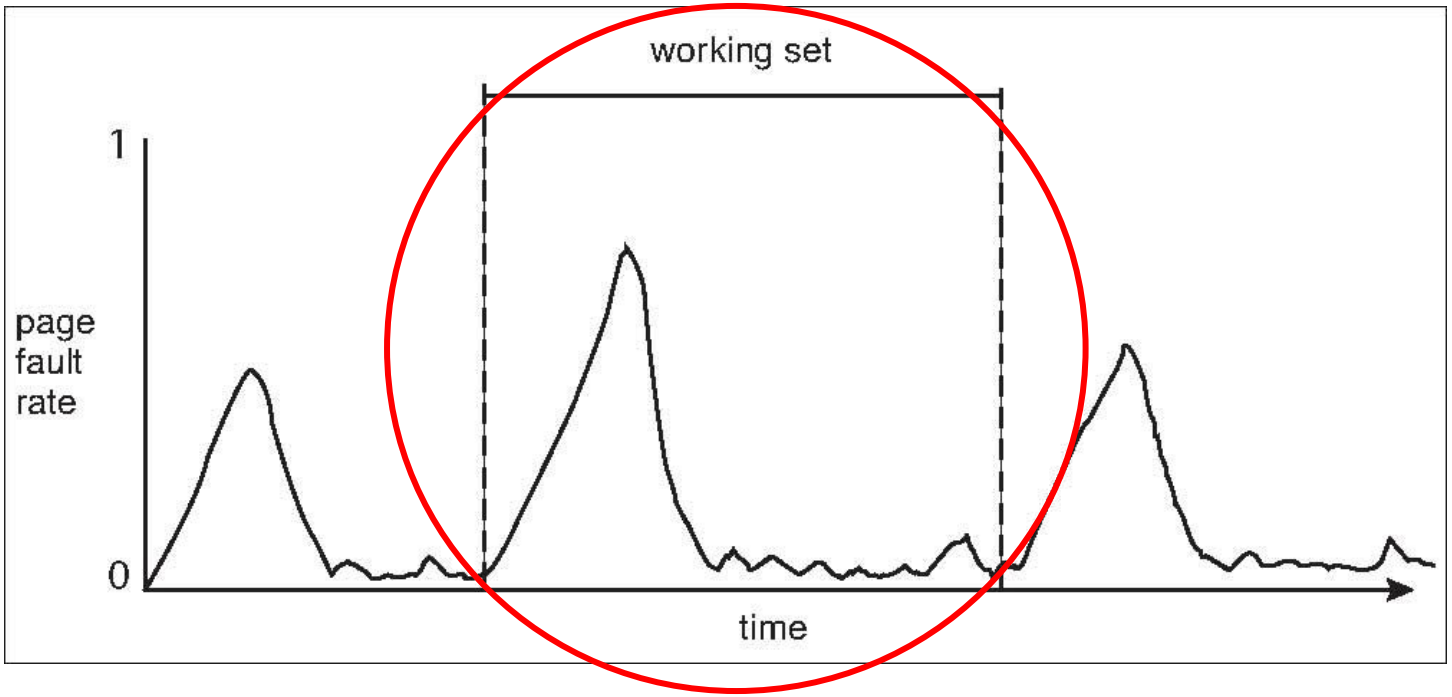
Page-Fault Frequency Scheme

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

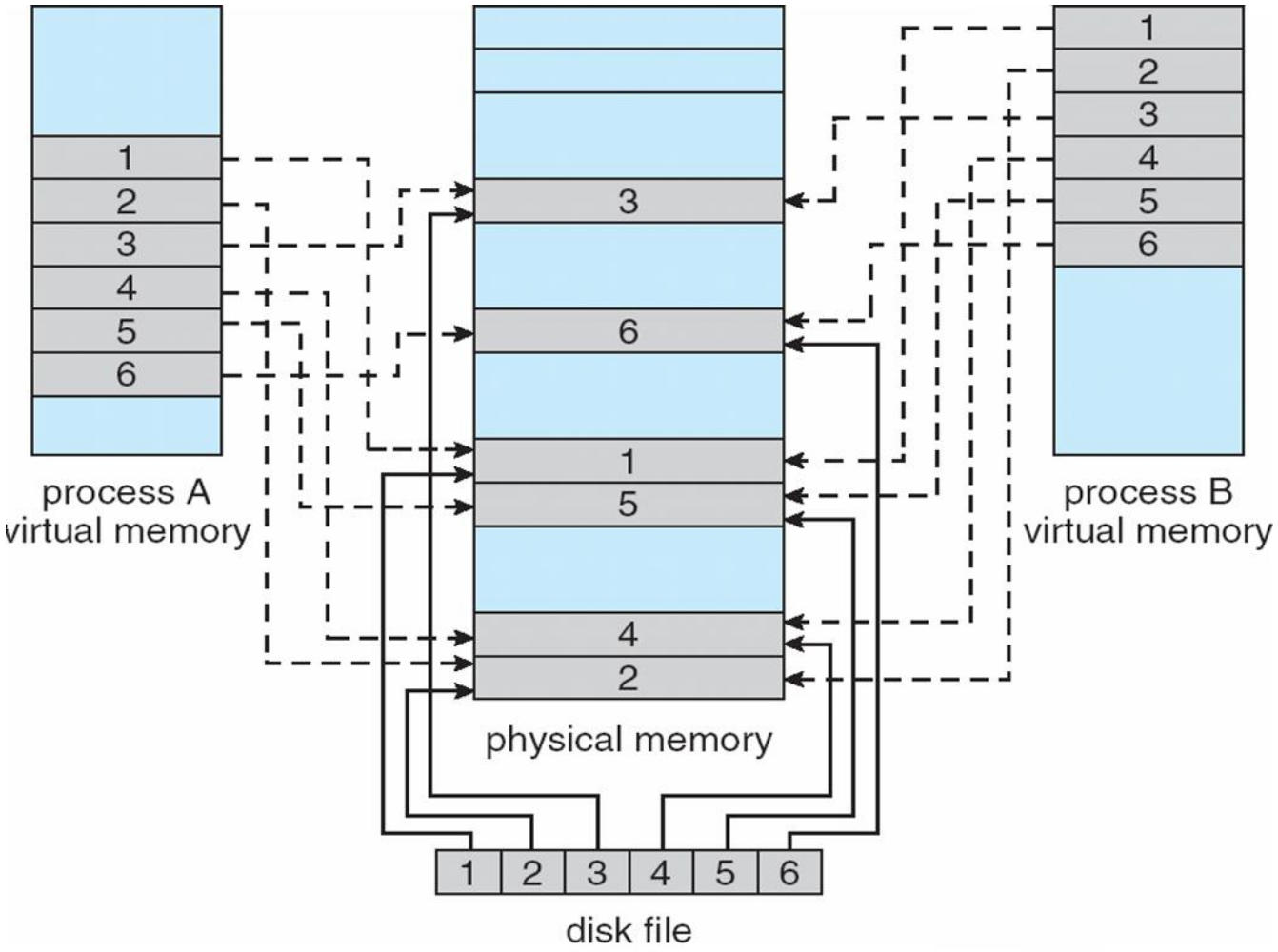
Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory

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.

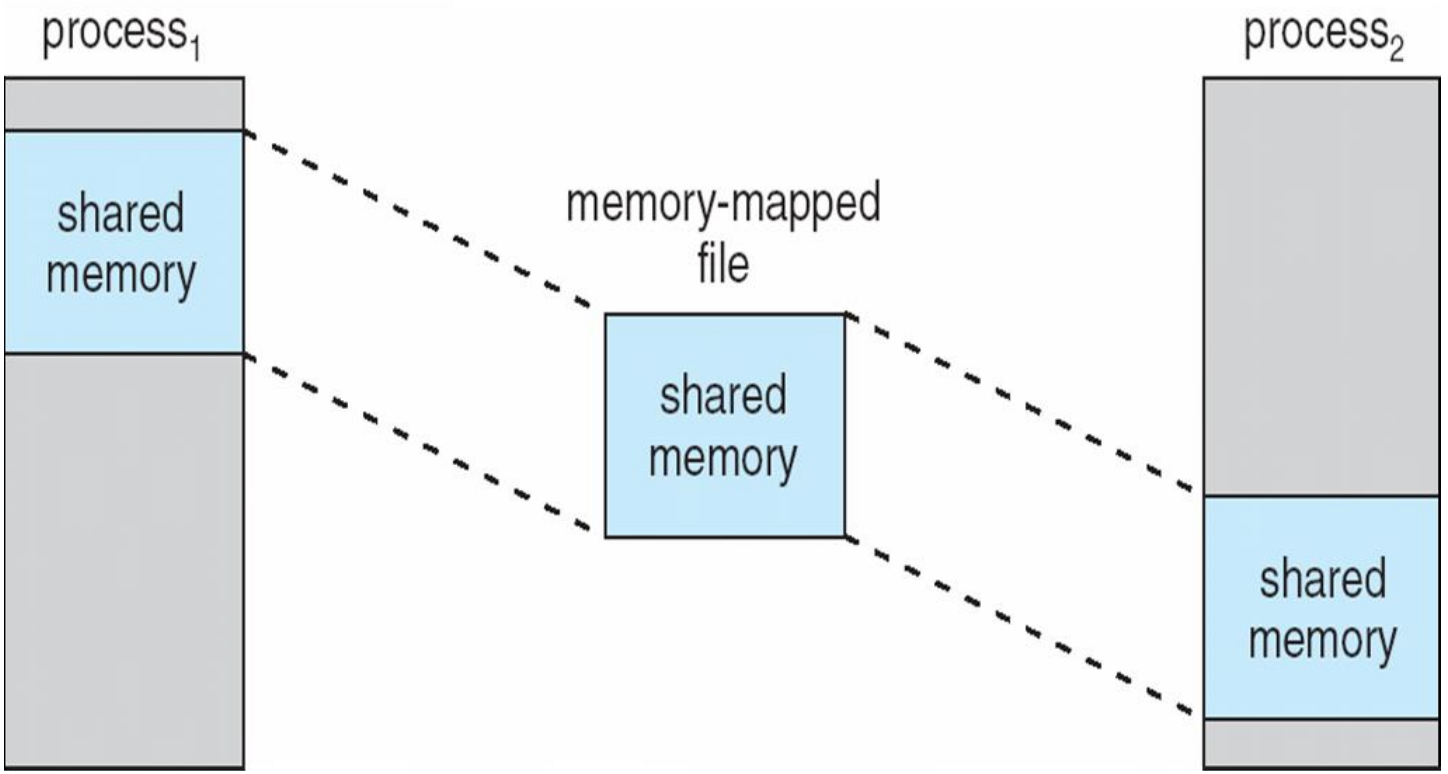
Simplifies file access by treating file I/O through memory rather than `read()`, `write()` system calls

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

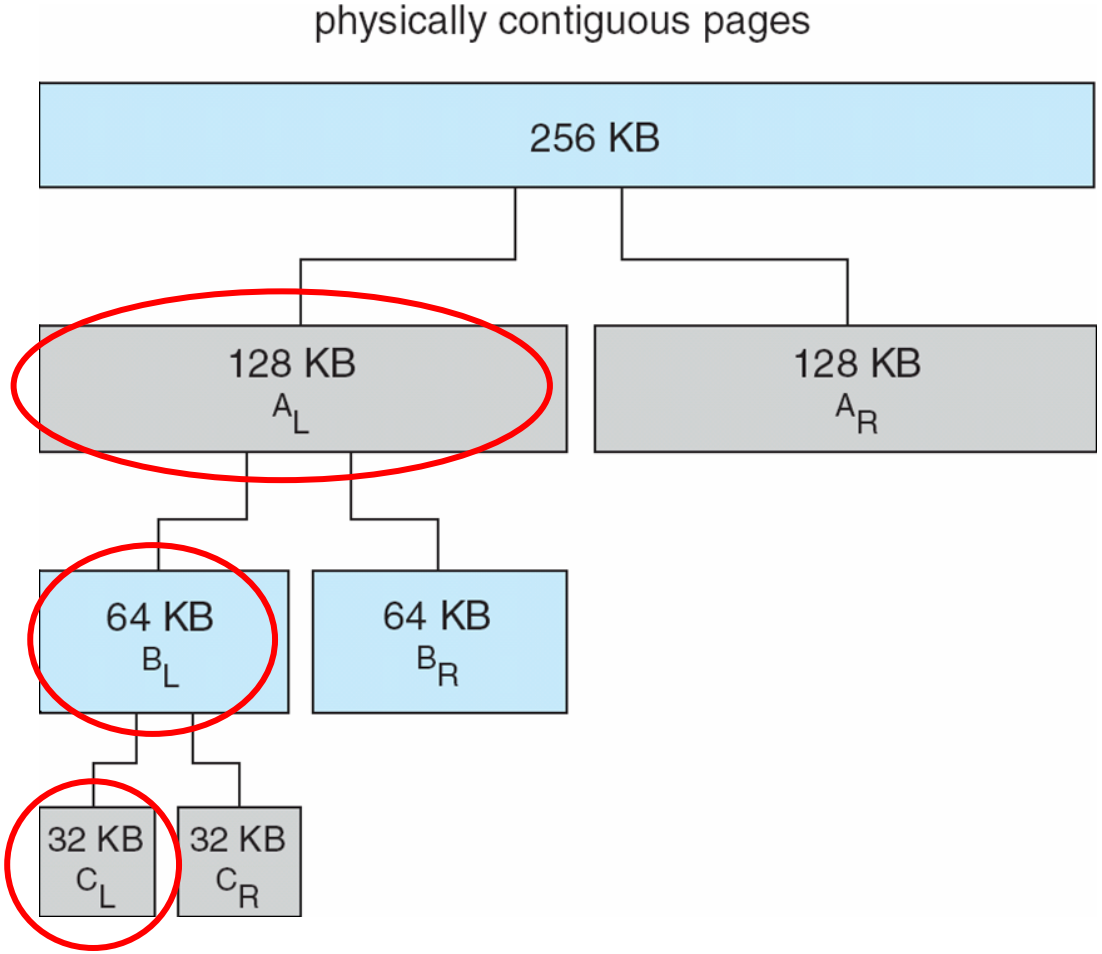
Request rounded up to next highest power of 2, for 11kB, it is satisfies with a 16-KB segment

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

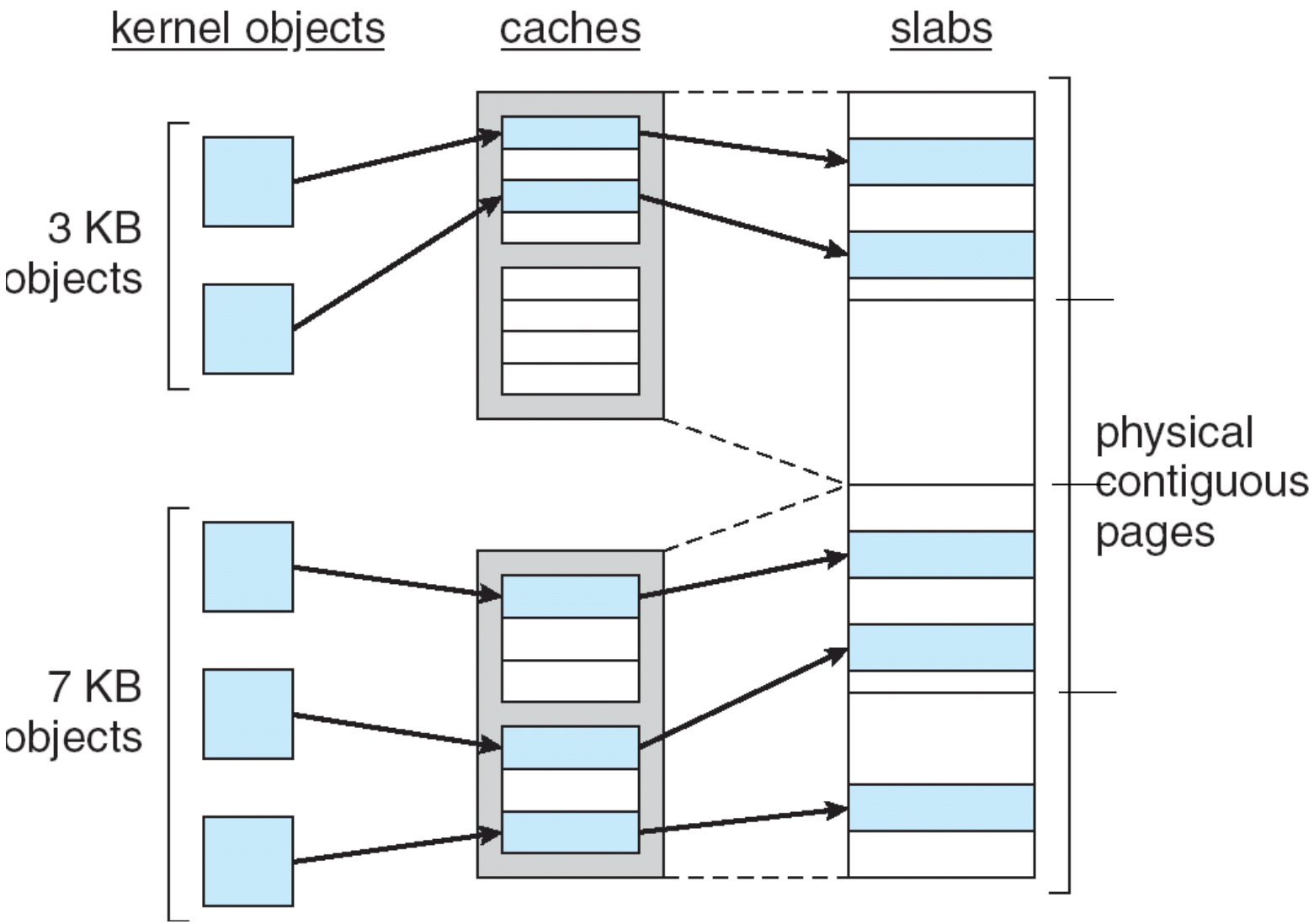
Cache consists of one or more slabs

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

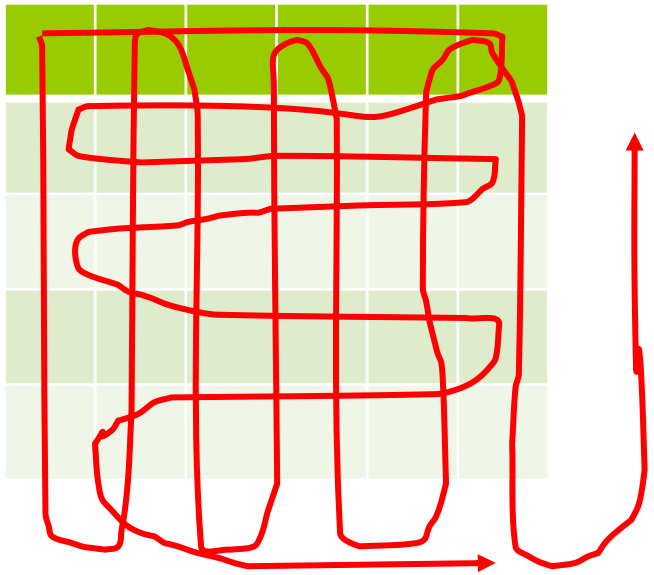
Program structure

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

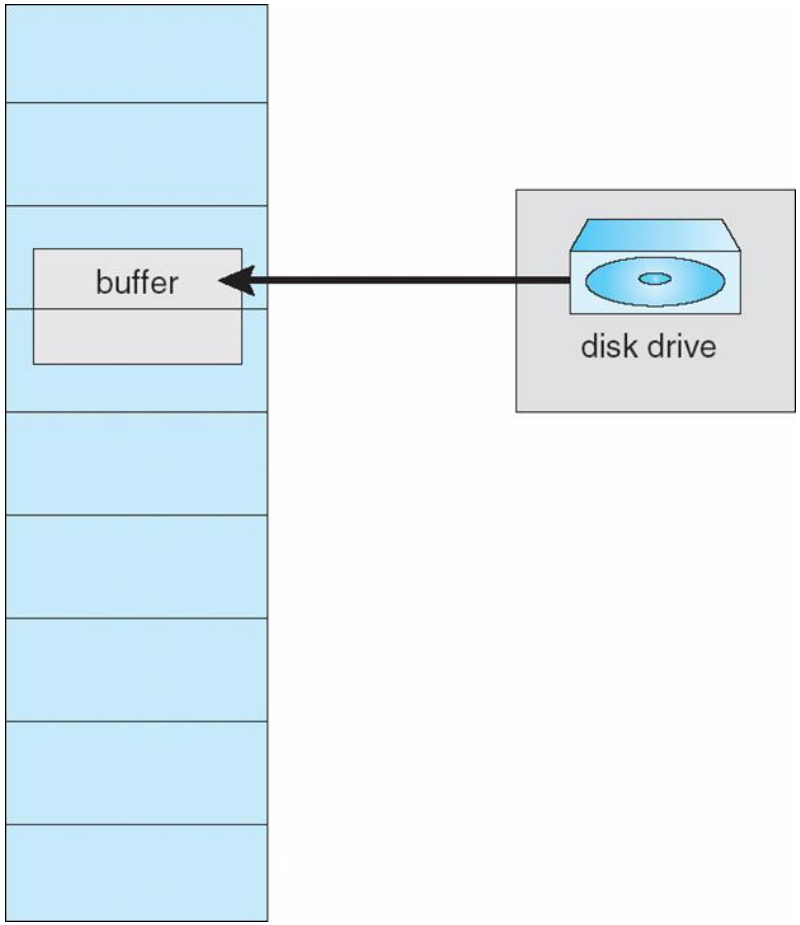
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

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.

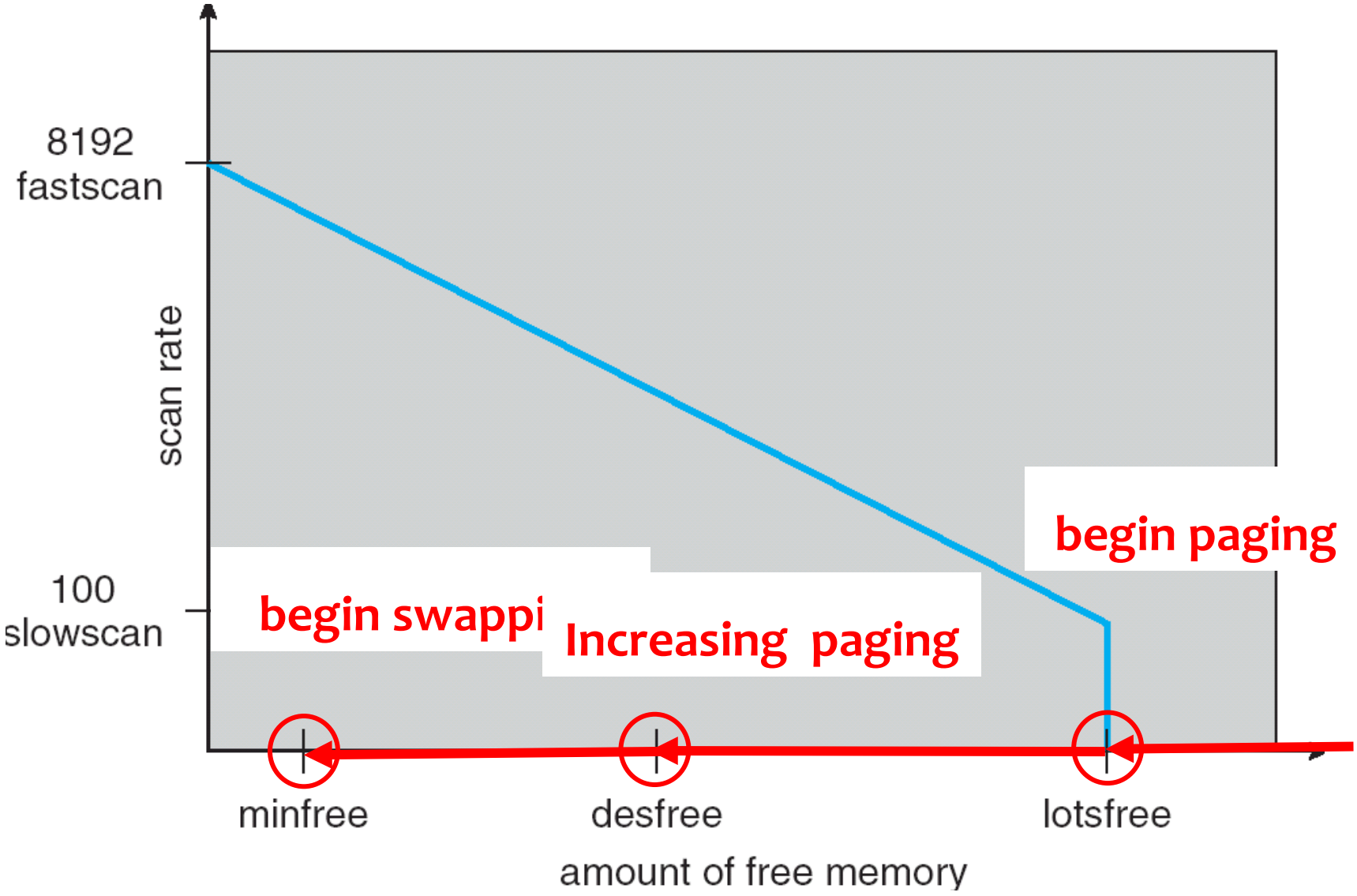
Handspread: the distance (in pages) between the two hands

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