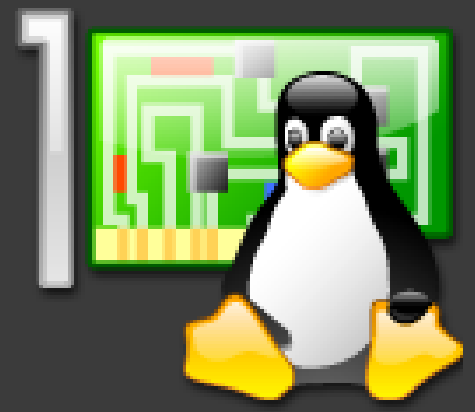


MEMORY MANAGEMENT

Objectives

- Explain the importance of managing main memory in the execution of the process;
- Describe the different memory management schemes;
- Discuss virtual memory in the form of demand paging and examine its complexity and cost

Background

- Just as processes share the CPU, they also share physical memory. This module is about mechanisms for doing that sharing.
- Program must be brought in memory for the process to be run.
  - **Memory** – A large array of words or bytes, each with its own address.
- OS must efficiently manage the primary memory of the computer.
  - This task is handled by the **Memory Manager**.

Memory Manager (MM)

- Responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory;
- Since every process needs memory, the MM is crucial to the performance of the entire system.

Primary goals: Memory Manager

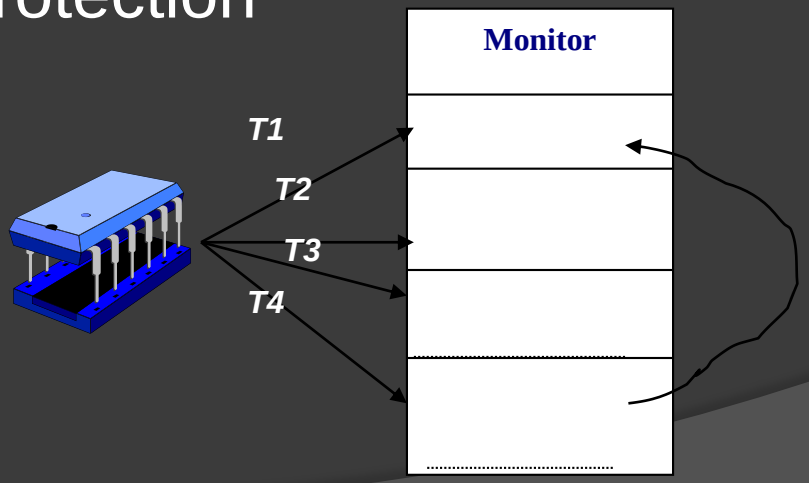
1. Managing the sharing of the primary memory
2. Minimizing memory access time.

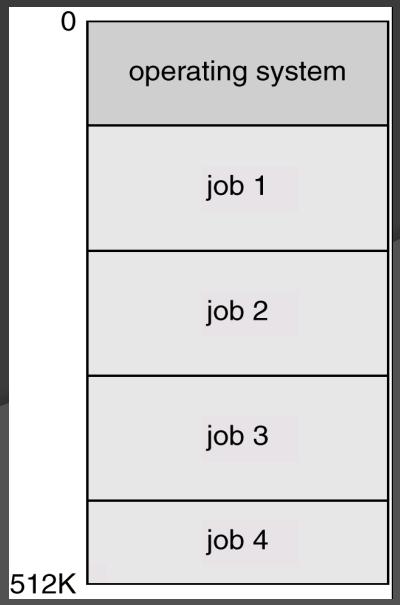
Memory Management Concepts

- Both program and its data must reside in the main memory.
- CPU fetches instructions and data from the memory;
- Modern multiprogramming systems are capable of storing more than one program, together with the data in the main memory

Memory Management Concepts

- A fundamental task of the memory management component of OS is to ensure safe execution of programs by providing:
  - Sharing of memory
  - Memory protection





Issues in sharing memory

- **Transparency** – several processes may co-exist, unaware of each other, in the main memory and run regardless of the number and location of other processes;

- **Safety** (or protection) – processes must not corrupt each other (nor the OS);

Issues in sharing memory

- **Efficiency** – CPU Utilization must be preserved and memory must be fairly allocated;
- **Relocation** – ability of a program to run in different memory locations

Storage allocation

- Information stored in main memory can be classified in a variety of ways:
  - Program (code) and data (variables and constants)
  - Read-only (code, constants) and read-write (variables)
  - Address (pointers) or data (other variables); binding (when memory is allocated for objects); static or dynamic

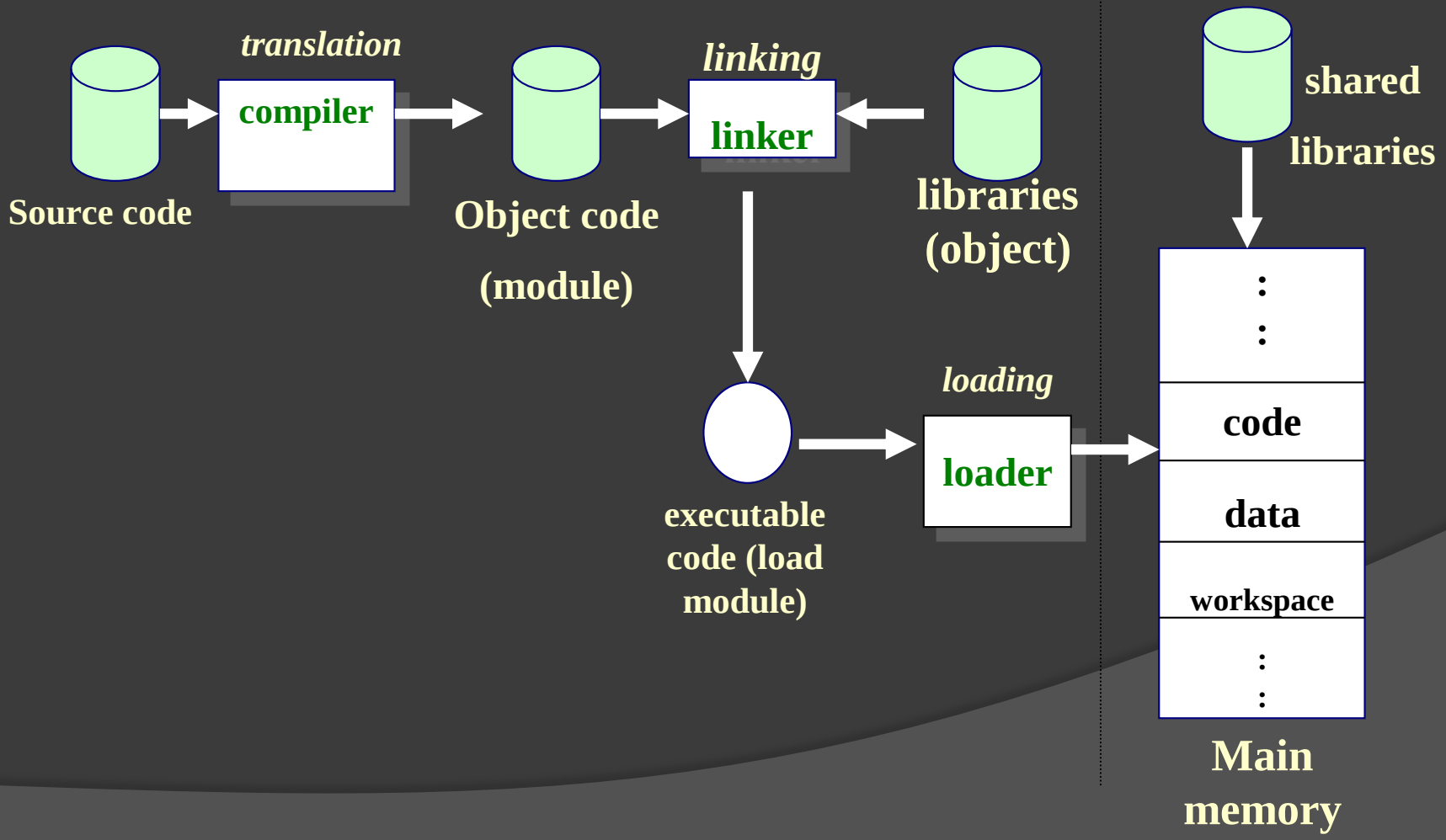
Storage allocation (cont.)

- The compiler, linker, loader and run-time libraries must all cooperate to manage the information

Creating an executable code

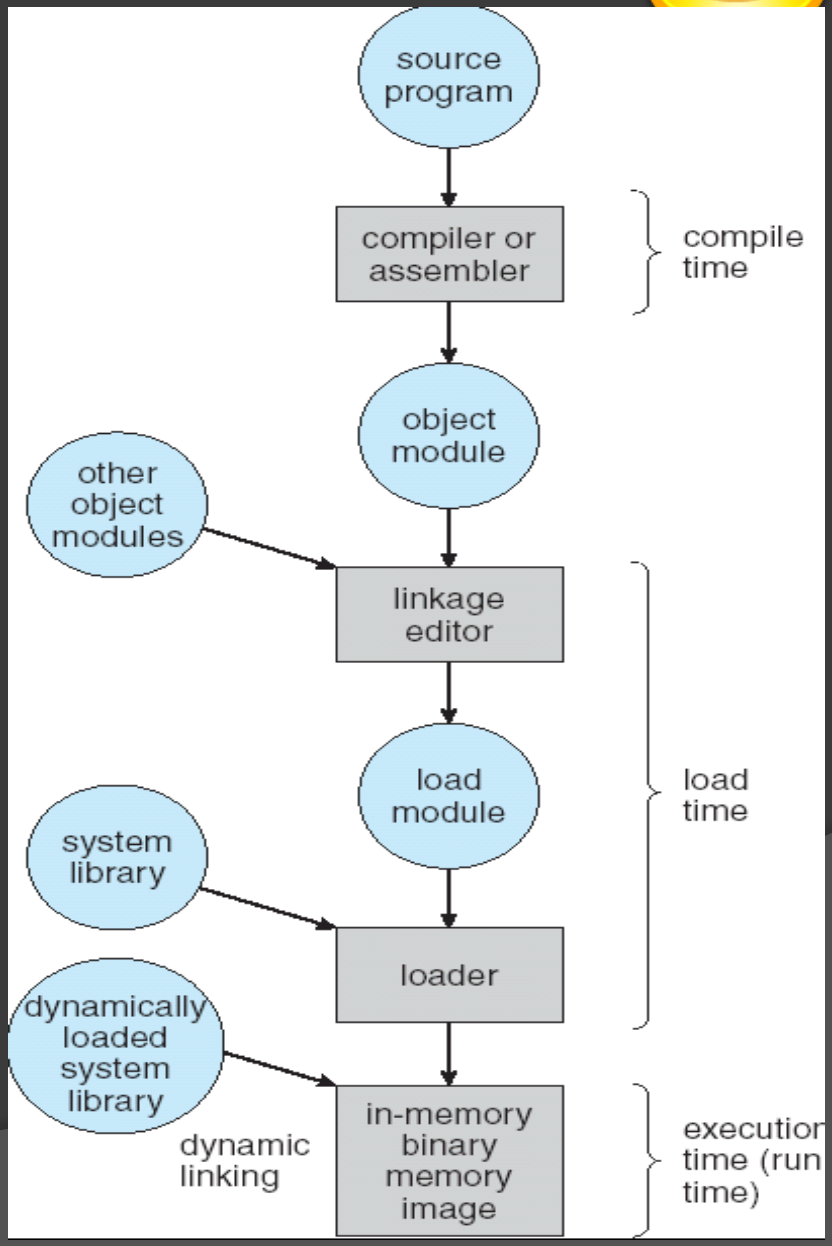
- Before a program can be executed by the CPU, it must go through several steps:
  - **Compiling (translating)** – generates the object code
  - **Linking** – combines the object code to a single self-sufficient executable code.
  - **Loading** – copies the executable code into memory.
  - **Execution** – dynamic memory allocation

Creating an executable code



Address Binding

- The process of associating program instructions and data (addresses) to physical memory address.
  - Static
  - Dynamic



Static Memory Allocation

- New locations are determined before execution
- **Compile time** – Compiler or assembler translates symbolic addresses to absolute addresses.
- **Load time** – Compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses.

Dynamic Memory Allocations

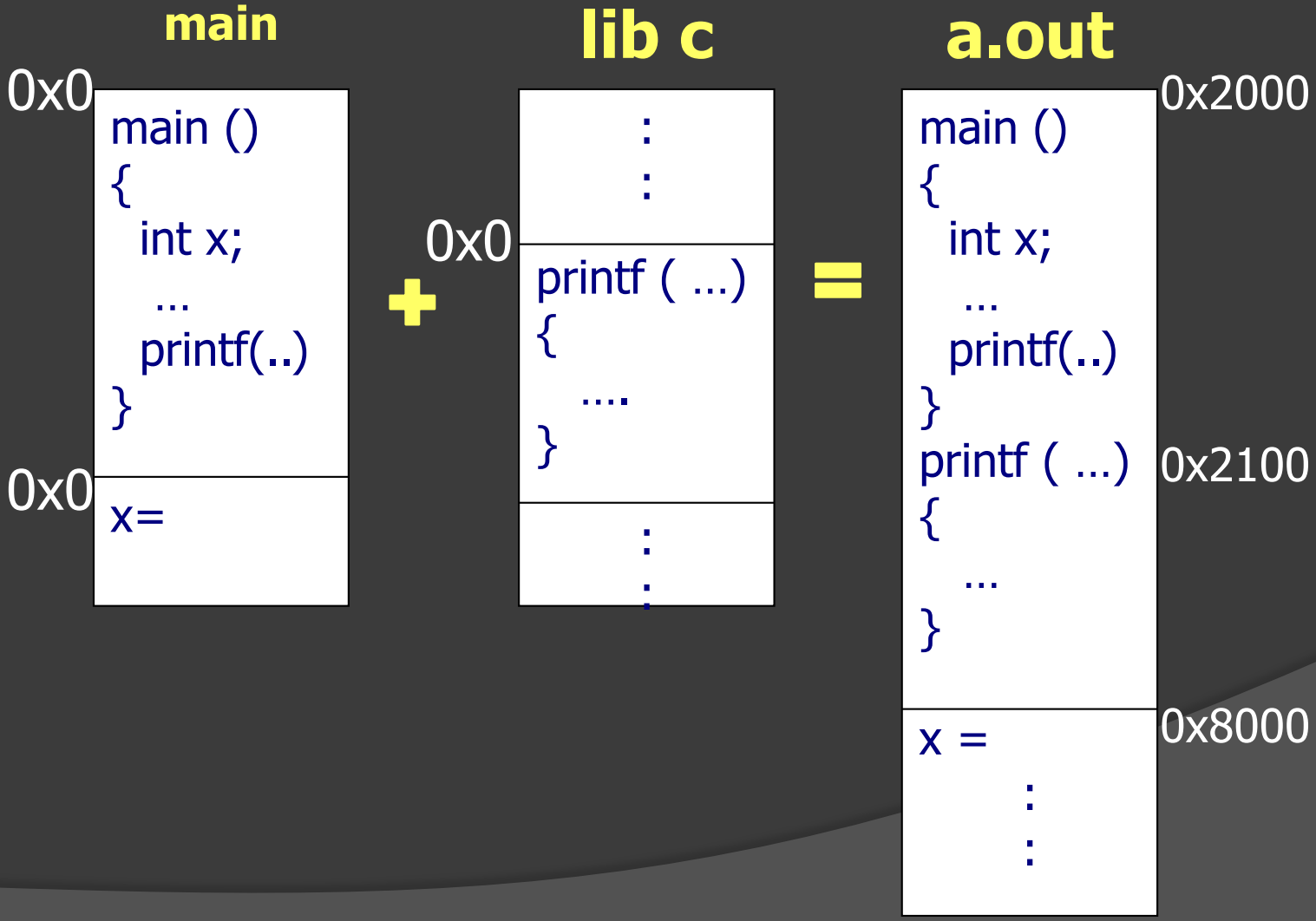
- **Run Time (Execution Time)** – program retains its relative addresses. The absolute addresses are generated by hardware. (e.g. Base and limit registers)

Functions of a linker

- A compile-time linker collects (if possible) and puts together all the required pieces to form the executable code

- Issues:
  - **Relocation** – where to put pieces
  - **Cross reference** – where to find pieces
  - **Re-organization** – new memory layout

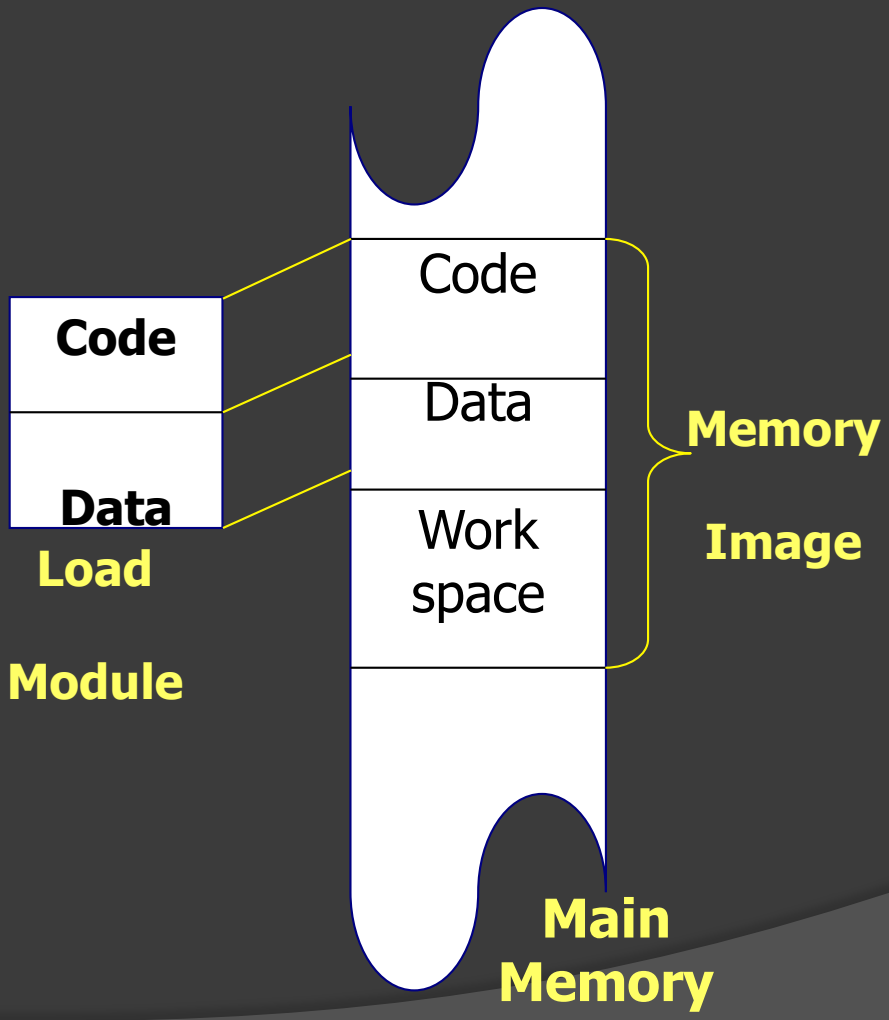
Functions of a linker



Functions of a loader

- Places the executable code in main memory starting at a predetermined location (base or start address). This can be done in several ways, depending on the hardware architecture.
  - **Absolute loading** – always loads programs into a designated memory location
  - **Relocatable loading** – allows loading programs in different memory locations
  - **Dynamic (run-time) loading** – loads functions when first called (if ever)

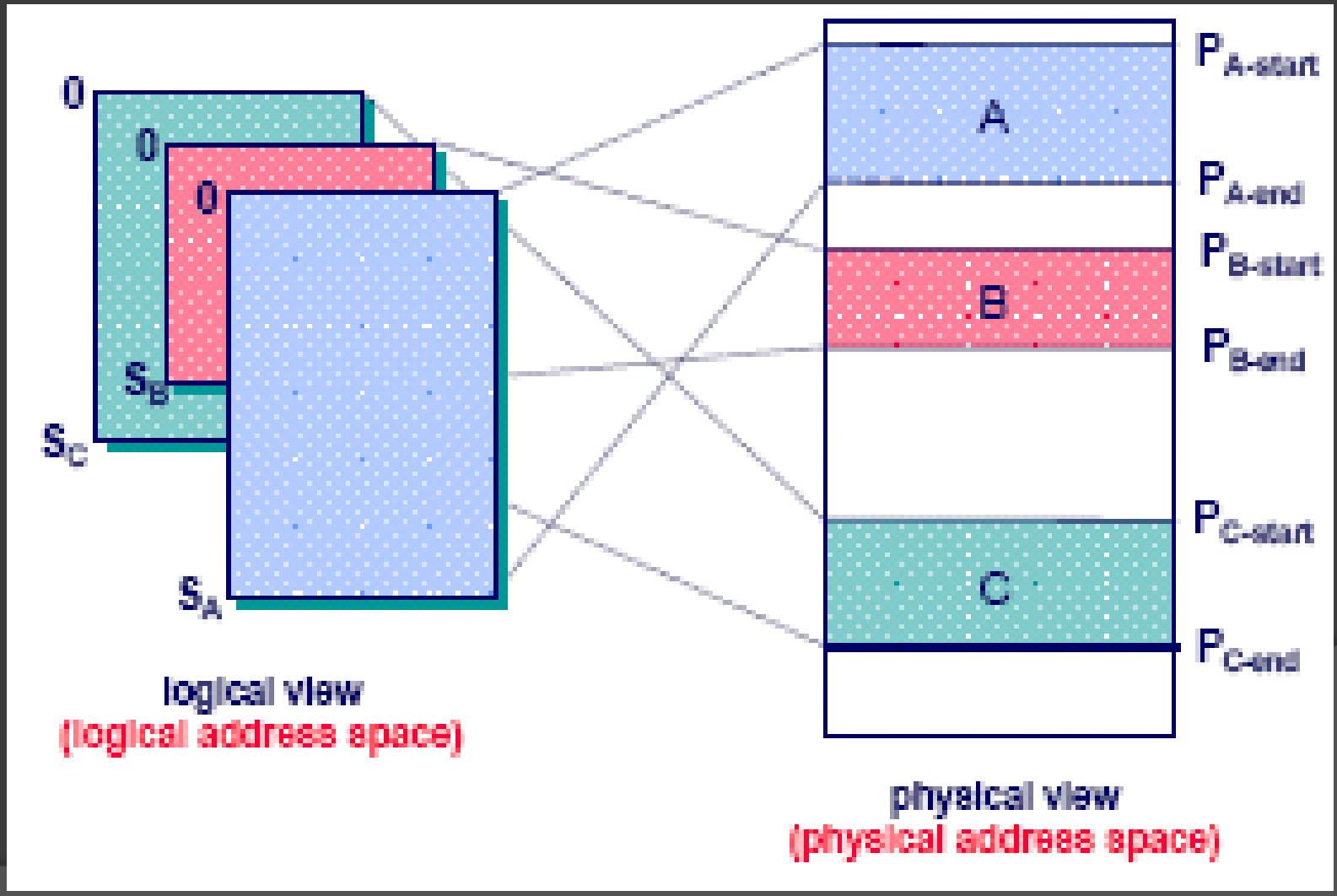
Functions of a loader (cont.)



Logical vs. Physical Address Space

- **Logical address**
  - generated by the CPU; also referred to as ***virtual address.***
- **Physical address**
  - address seen by 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

Logical vs. Physical addresses



Binding logical to Physical

```
 4 void      main()
 5    {
 6    printf( "Hello, from main\n" );
 7    b();
 8 }
 9
10
11 voidb()
12    {
13    printf( "Hello, from 'b'\n" );
14 }
```

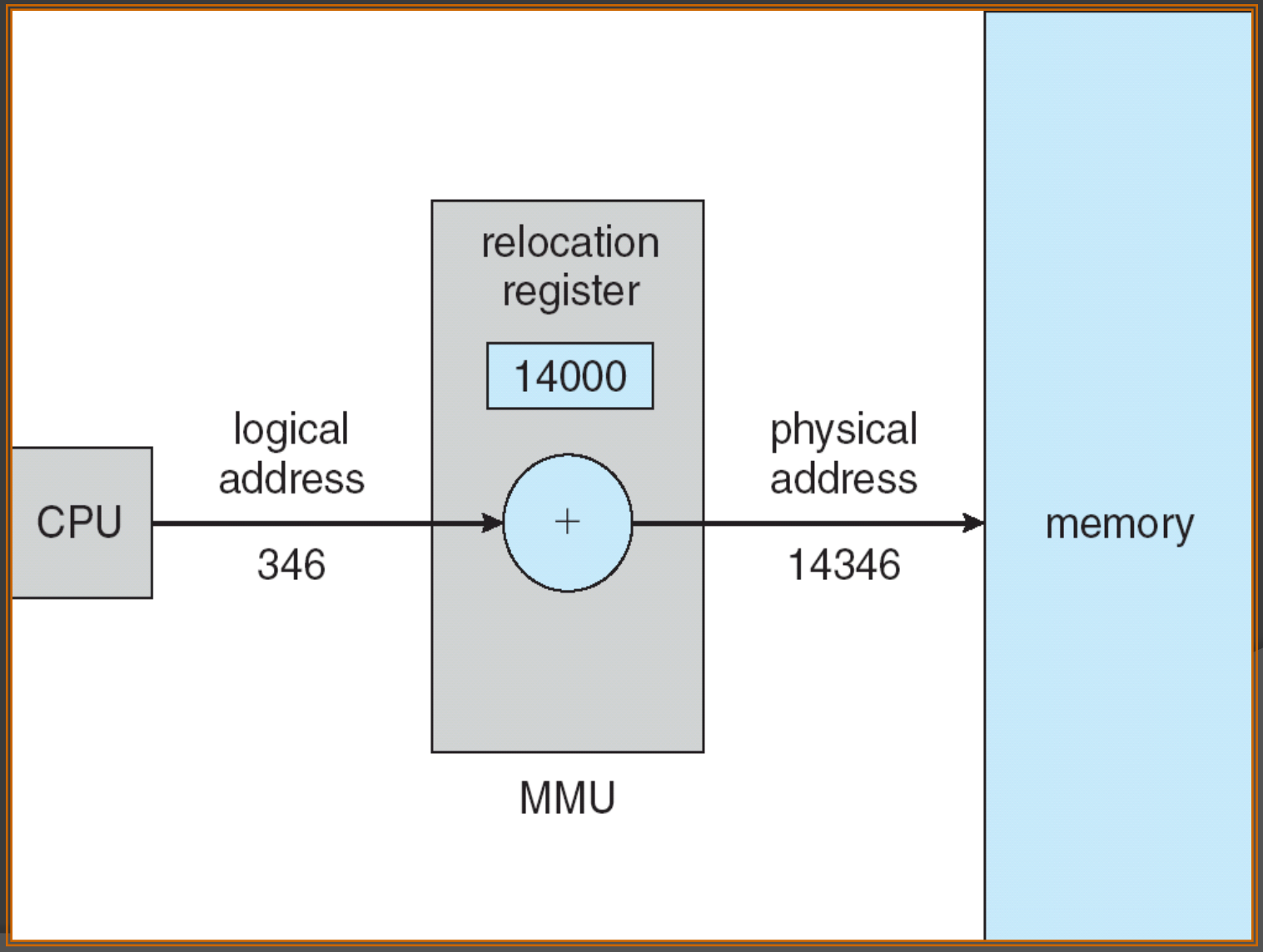
Binding logical to Physical

```
ASSEMBLY LANGUAGE LISTING

000000B0: 6BC23FD9  stw     %r2,-20(%sp       ; main()
000000B4  37DE0080  ldo     64(%sp),%sp
000000B8  E8200000  bl      0x000000C0,%r1    ; get current addr=BC
000000BC  D4201C1E   depi   0,31,2,%r1
000000C0  34213E81  ldo     -192(%r1),%r1     ; get code start area
000000C4  E8400028  bl      0x000000E0,%r2    ; call printf
000000C8  B43A0040   addi   32,%r1,%r26       ; calc. String loc.
000000CC  E8400040  bl      0x000000F4,%r2    ; call b
000000D0  6BC23FD9   stw    %r2,-20(%sp)      ; store return addr
000000D4  4BC23F59  ldw     -84(%sp),%r2
000000D8  E840C000  bv      %r0(%r2)          ; return from main
000000DC  37DE3F81   ldo    -64(%sp),%sp
                                                    STUB(S) FROM LINE 6
000000E0: E8200000  bl      0x000000E8,%r1
000000E4  28200000   addil  L%0,%r1
000000E8: E020E002  be,n    0x00000000(%sr7,%r1)

000000EC  08000240  nop                                          void    b()
000000F0: 6BC23FD9  stw     %r2,-20(%sp)
000000F4: 37DE0080  ldo     64(%sp),%sp
000000F8  E8200000  bl      0x00000100,%r1      ; get current addr=F8
000000FC  D4201C1E   depi   0,31,2,%r1
00000100  34213E01  ldo     -256(%r1),%r1     ; get code start area
00000104  E85F1FAD  bl      0x000000E0,%r2    ; call printf
00000108  B43A0010   addi   8,%r1,%r26
0000010C  4BC23F59  ldw     -84(%sp),%r2
00000110  E840C000  bv      %r0(%r2)          ; return from b
00000114  37DE3F81   ldo    -64(%sp),%sp
```

Memory-Management Unit (MMU)

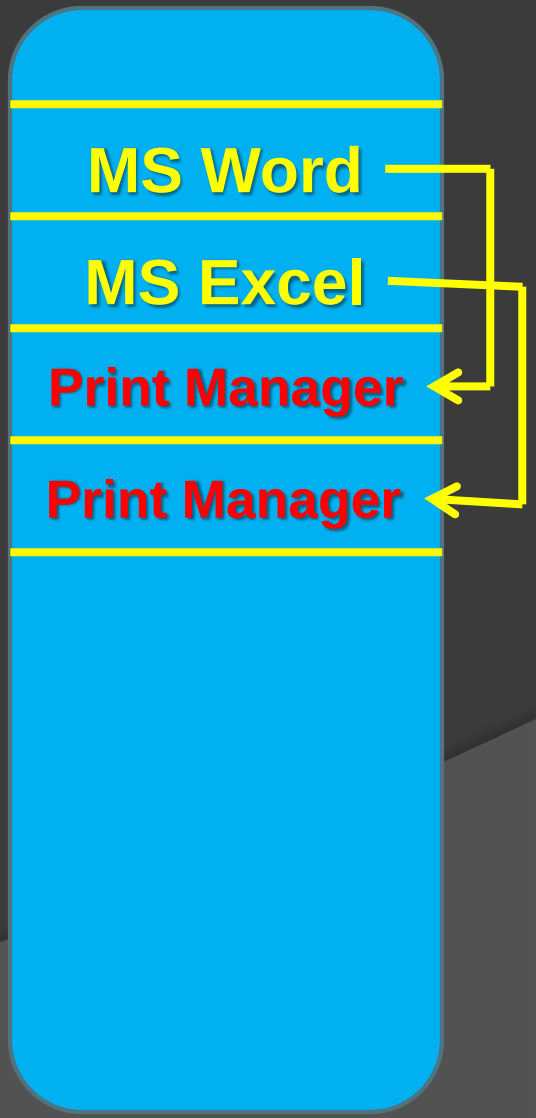
- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



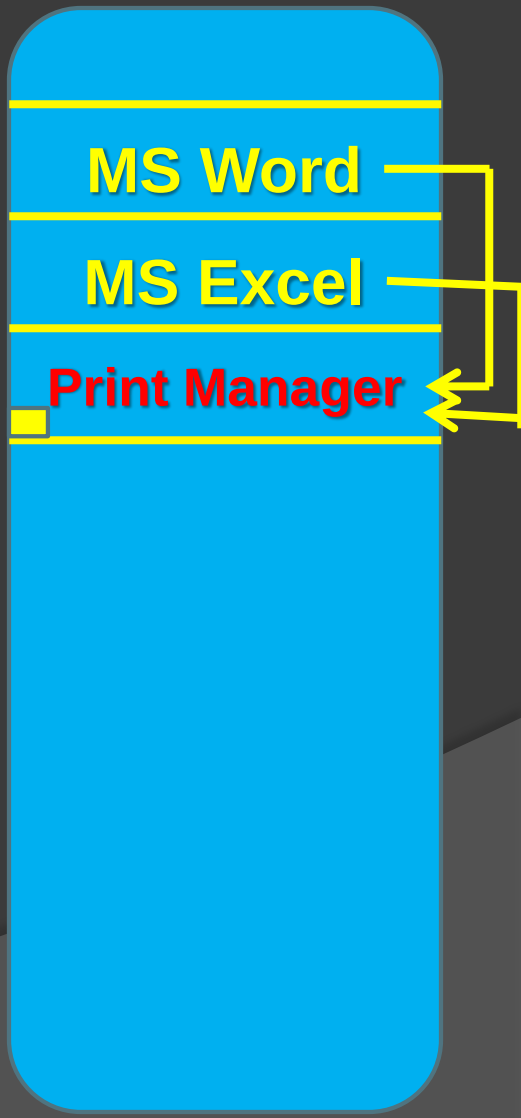
Dynamic Loading

- 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 OS is required implemented through program design.



Dynamic Linking

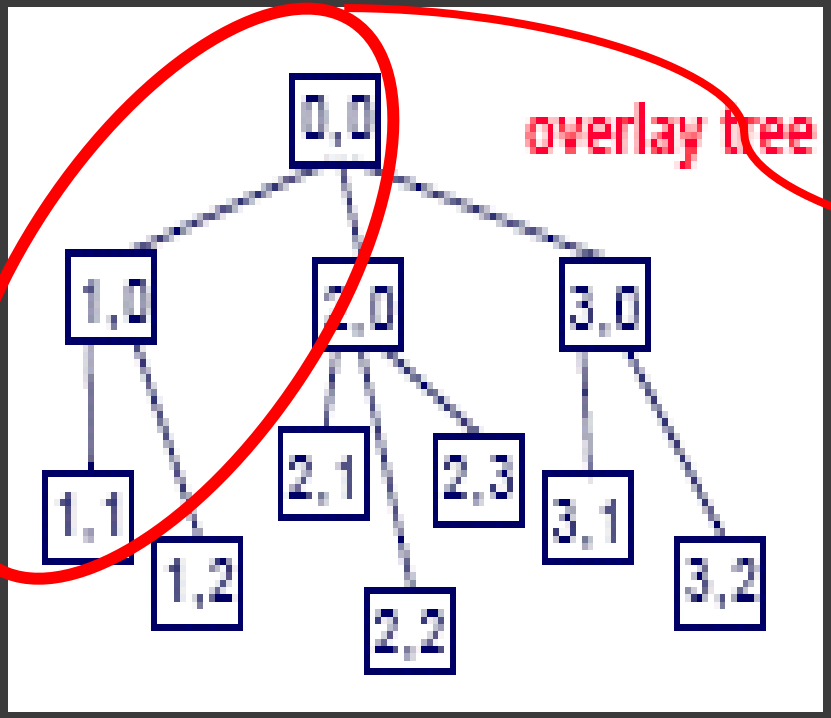
- Small piece of code, ***stub*,** used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- OS is needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries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

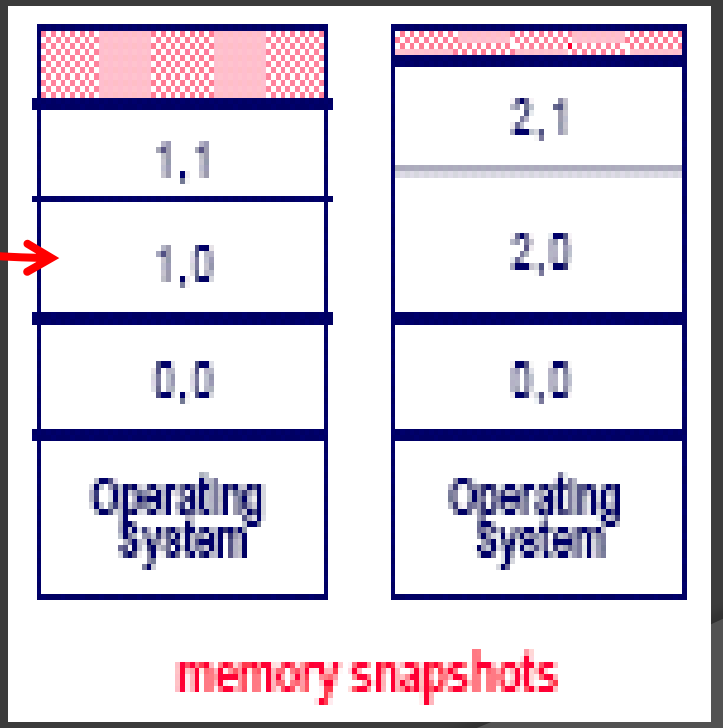


Overlays

- Needed when process is larger than amount of memory allocated to it.
- Keep in memory only those instructions and data that are needed at any given time.
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

Overlays


0,0
1,0
2,0
3,0
1,1
1,2
2,1
2,2
2,3
3,1
3,2
overlay tree


1,1
1,0
0,0
Operating System
2,1
2,0
0,0
Operating System
memory snapshots

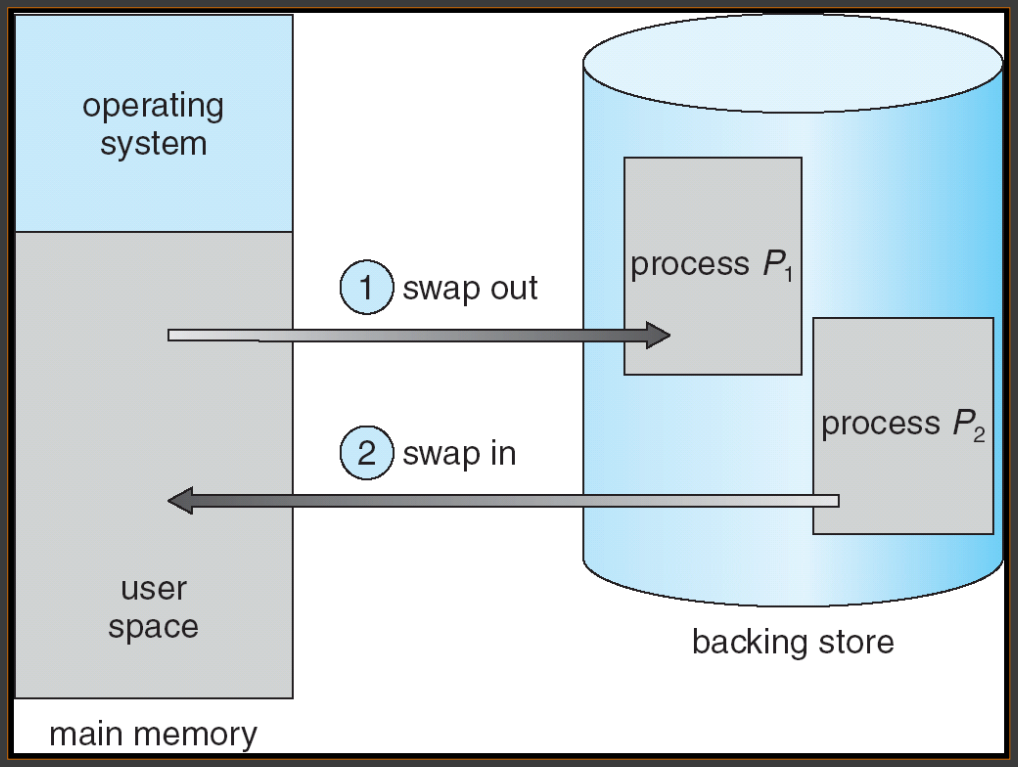
Swapping

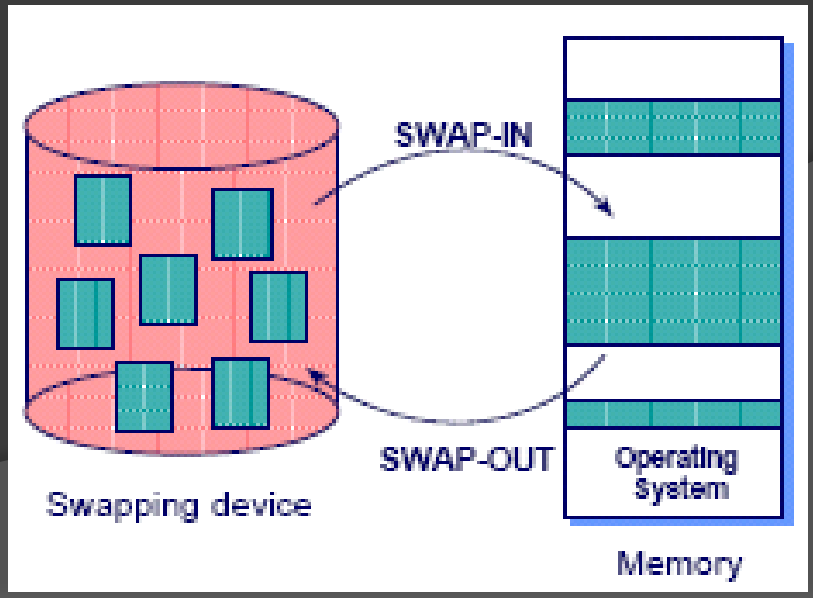
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll 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 (cont.)

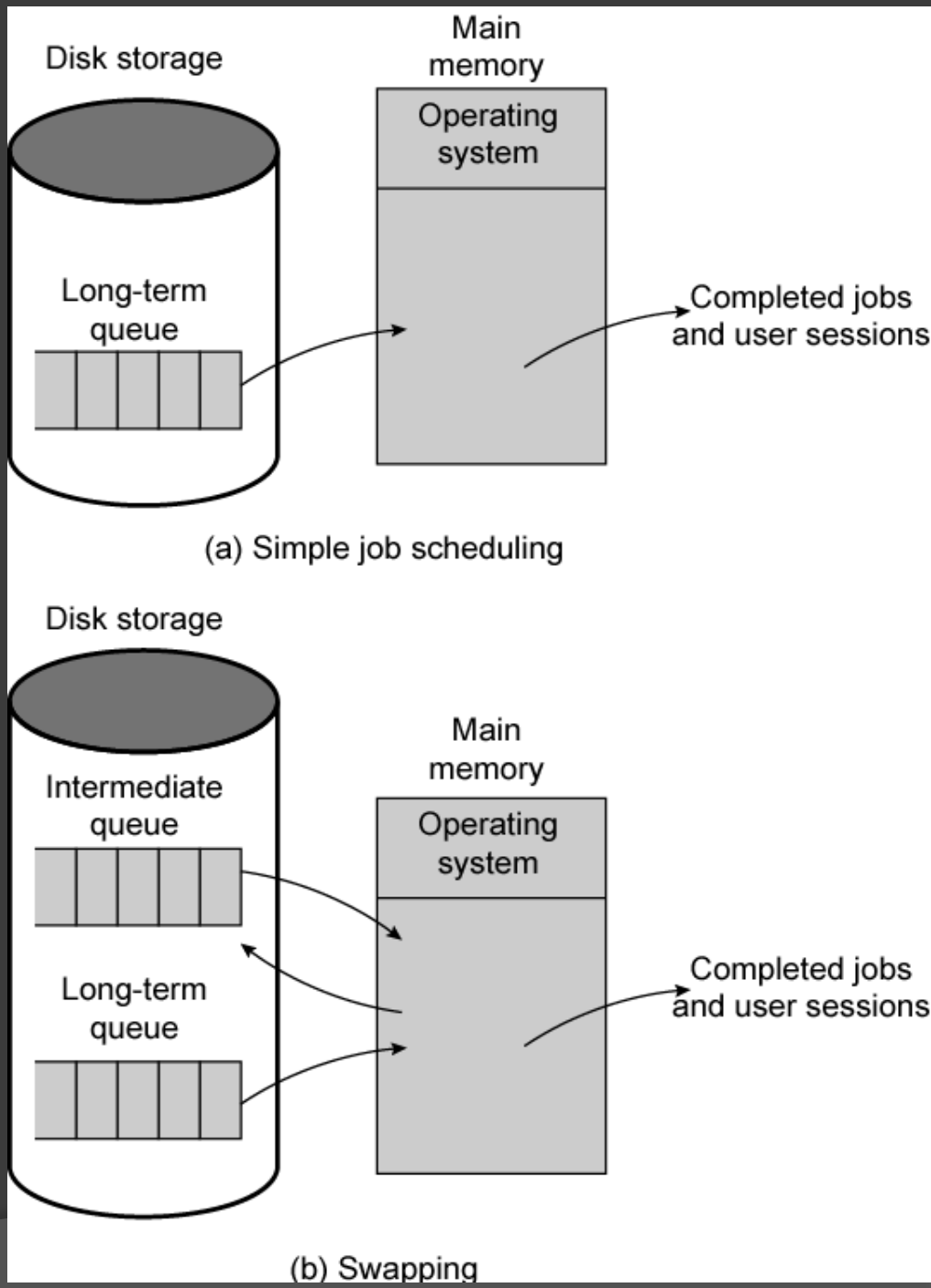
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

Schematic View of Swapping





Use of swapping



Partitioning:

- Simple method to accommodate several programs in memory at the same time to support multiprogramming;
- Memory is divided into a number of contiguous regions, called *partitions*.

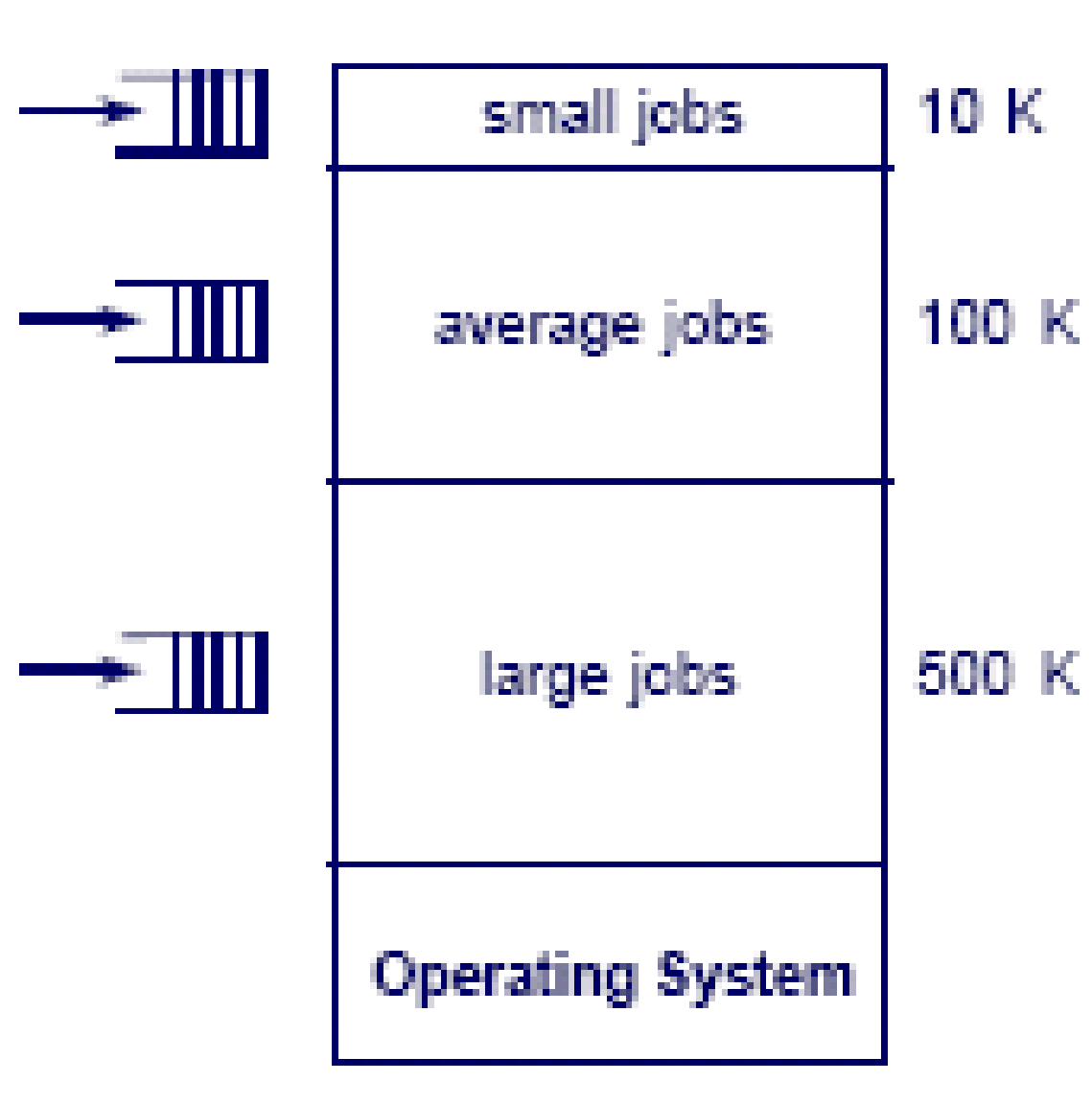
- Two forms of memory partitioning:
  - Static partitioning
  - Dynamic partitioning

Static Partitioning:

- Main memory is divided into *fixed* number of (fixed size) partitions during system generation or startup.

- Programs are queued to run in the smallest available partition. An executable prepared to run in one partition may not be able to run in another, without being re-linked. This technique uses *absolute loading*.

Static Partitioning:



MFT: A Static partitioning

- Stands for Multiprogramming with a Fixed Number of Tasks (MFT)
- Regions/partitions are fixed
- If jobs are placed in memory, it can compete for the CPU
- When a process terminates, it frees memory space and another job may fit the region.

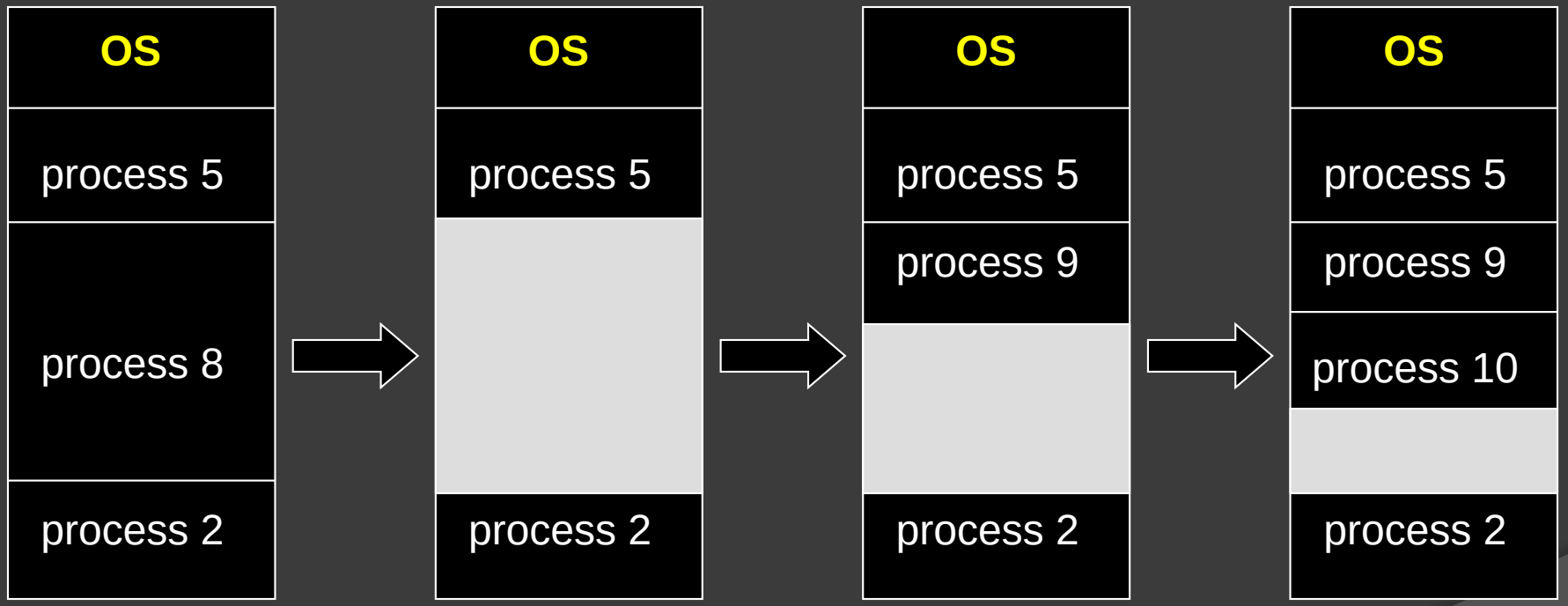
Dynamic Partitioning:

- Any number of programs can be loaded to memory as long as there is room for each.
- When a program is loaded (*relocatable loading*), it is allocated memory in exact amount as it needs.
- Also, the addresses in the program are fixed after loaded, so it cannot move. The OS keeps track of each partition (their size and locations in the memory.)

Dynamic Partitioning (cont.)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various sizes 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)

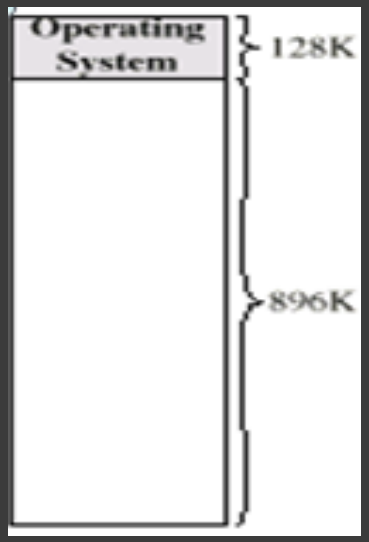
Dynamic Allocation Example

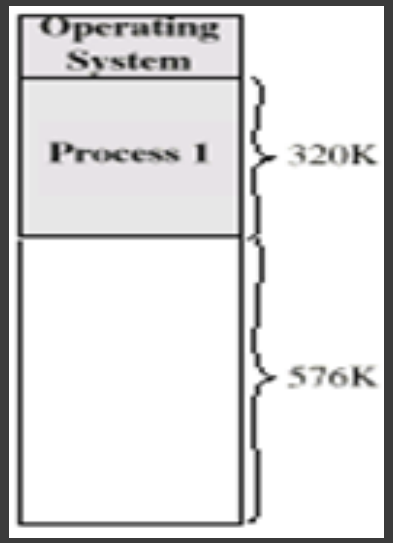


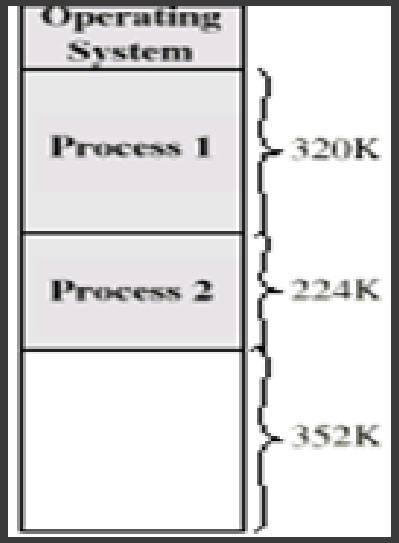
MVT: A Dynamic Partitioning

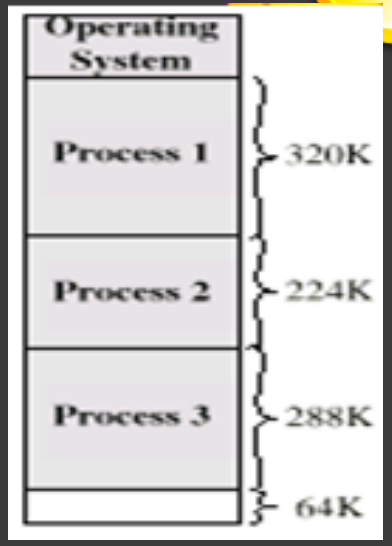
- Multiprogramming with a Variable number of Tasks (MVT)
- Region sizes can change
- OS keeps a table indicating which part of the memory are available and which part are not.
- Job is allocated only the amount of memory needed.
- After use, the job releases the memory back to the sets of holes
- Adjacent holes are merged together.

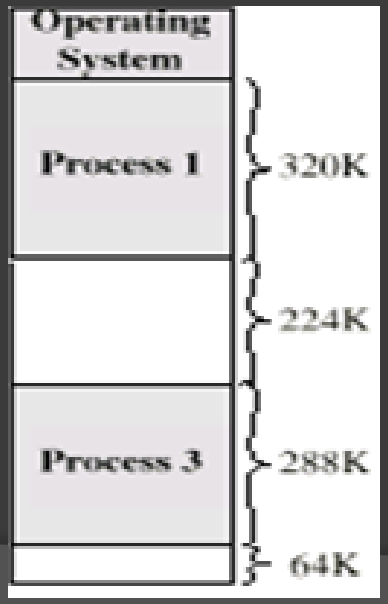
Effect of Dynamic Partitioning

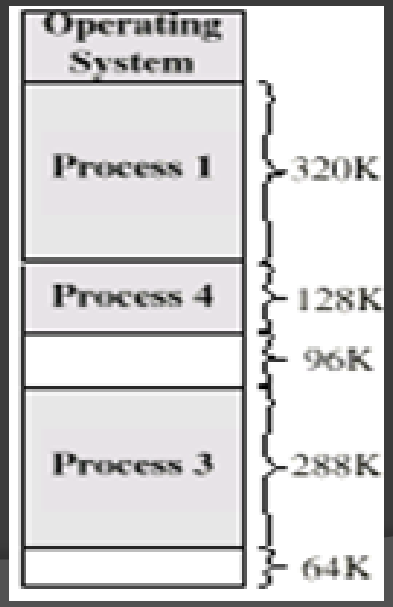


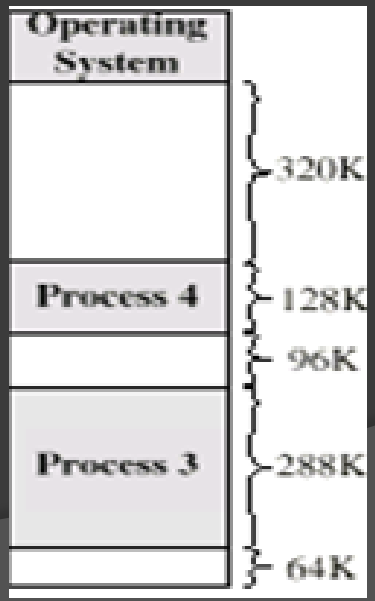


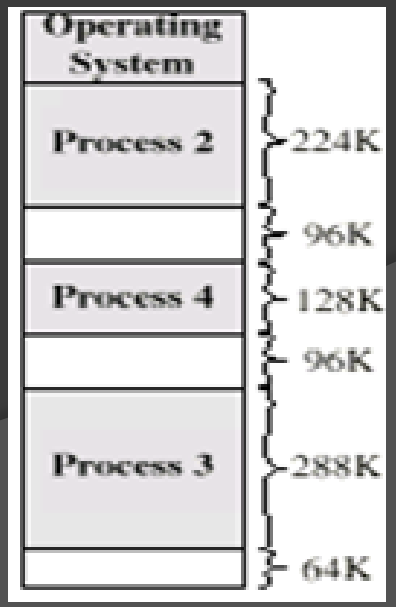












Example:

- Assume a 2,560k of available memory and a resident OS of 400k.

- Given the input queue in the figure that uses FCFS in allocating jobs and by using a round-robin CPU scheduling with a quantum time of 1 time unit, show how to allocate all the processes in memory and determine when each process terminates.

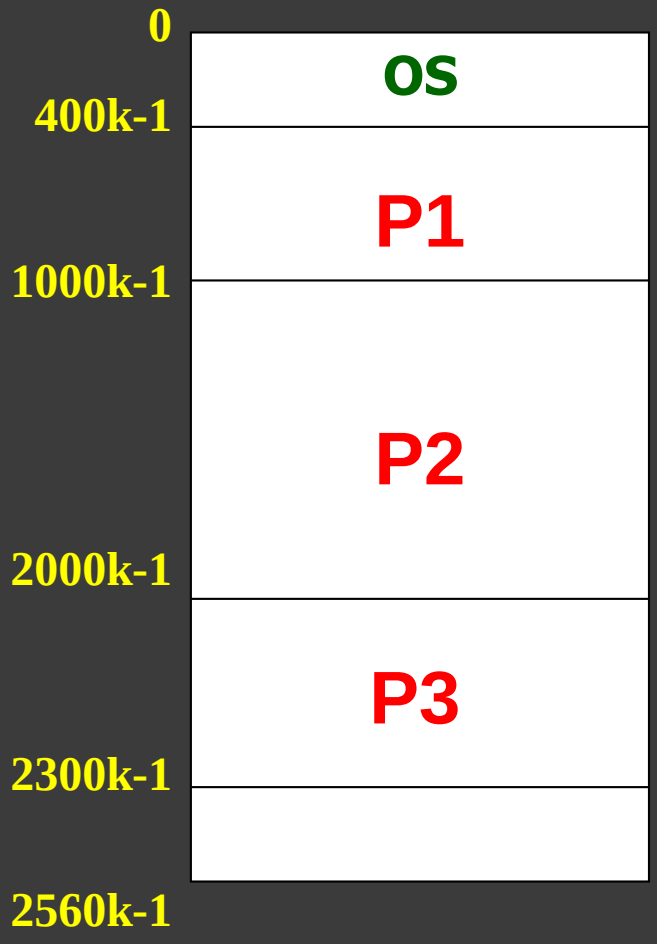
Job Queue

Process	Memory	Burst Time
P1	600k	10
P2	1000k	5
P3	300k	20
P4	700k	8
P5	500k	15

Solution:

Job Queue

Process	Memory	Burst Time
P1	600k	10
P2	1000k	5
P3	300k	20
P4	700k	8
P5	500k	15



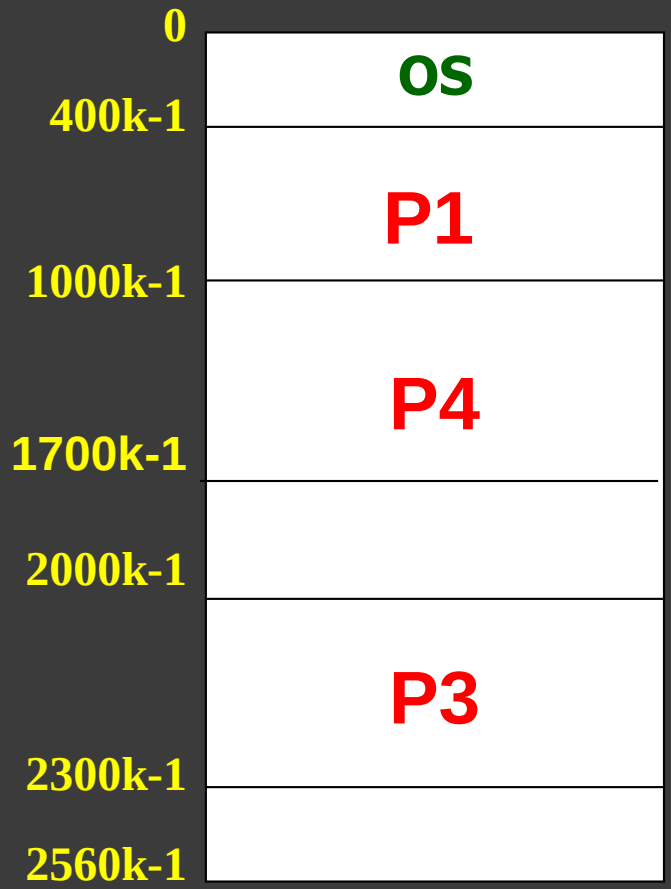
There will only be 3 active processes out of 5

Which process will terminate first?
At what time will that process terminate?

Solution:

Job Queue

Process	Memory	Burst Time
P1	600k	10
P2	1000k	5
P3	300k	20
P4	700k	8



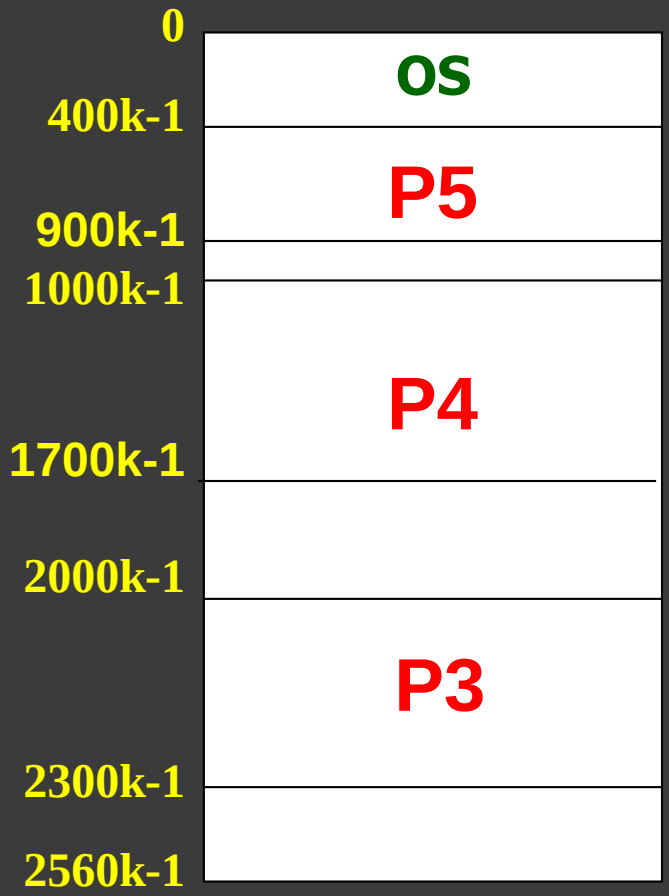
Process **P2** terminates first at time =14

and replaces the space occupied by process P4.

What process will terminate next?
At what time will that process end?

Solution:

Job Queue		
Process	Memory	Burst Time
P1	600k	10
P3	300k	20
P4	700k	8
P5	500k	15



Process **P1** terminates next at time = 28

and replaces the occupied space by process **P5**.

What process will terminate next?
At what time will that process end?

Dynamic Storage-Allocation Problem (MVT)

- How to satisfy a request of size *n* from a list of free holes?
  - **First-fit**: Allocate the *first* hole that is big enough
    - Algorithms Next fit, and Quick fit are used
  - **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 are better than worst-fit in terms of speed and storage utilization

Example

- Given memory partitions of 100k, 500k, 200k, 300k and 600k (in order),
- How would each of the FF, BF and WF algorithms allocate processes of 212k, 417k, 112k and 426k (in order)

Solution: Allocation of 212k, 417k, 112k and 426k (in order)

100	
500	**212k**
200	**112k**
300	
600	**417k**

First Fit **426k**

100	
500	**417k**
200	**112 k**
300	**212k**
600	**426k**

Best Fit

100	
500	**417k**
200	
300	**112k**
600	**212k**

Worst Fit **426k**

Fragmentation

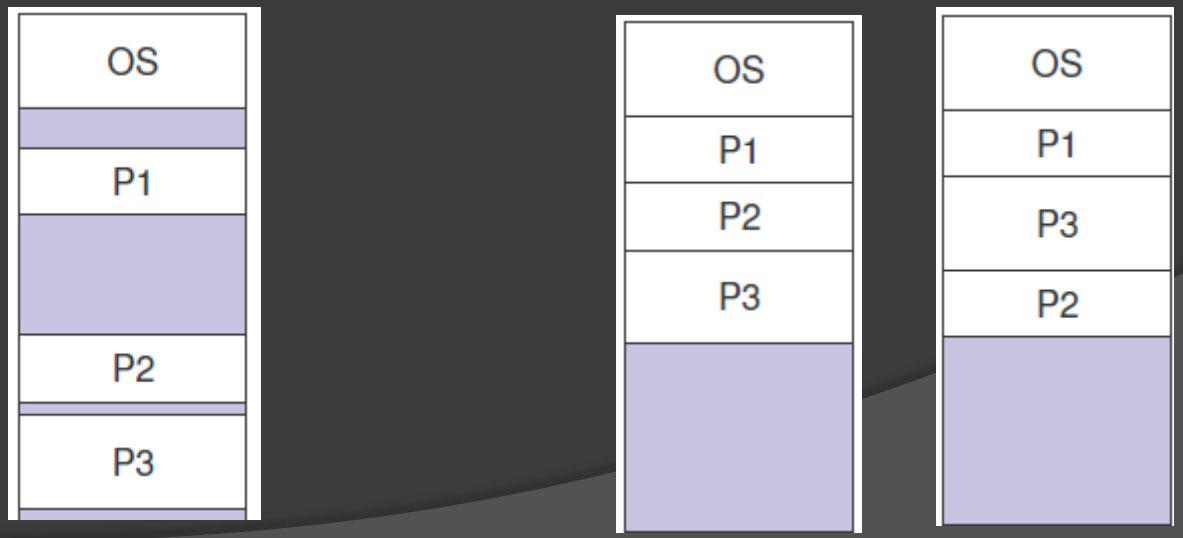
- Result of loading and removing processes from memory, the free memory space is broken into little pieces.
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Solution to External Fragmentation

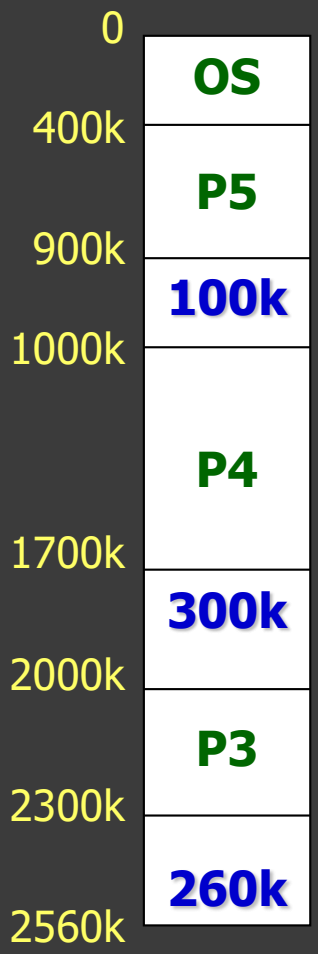
- Compaction
- Paging
- Segmentation

Solution 1: Compaction

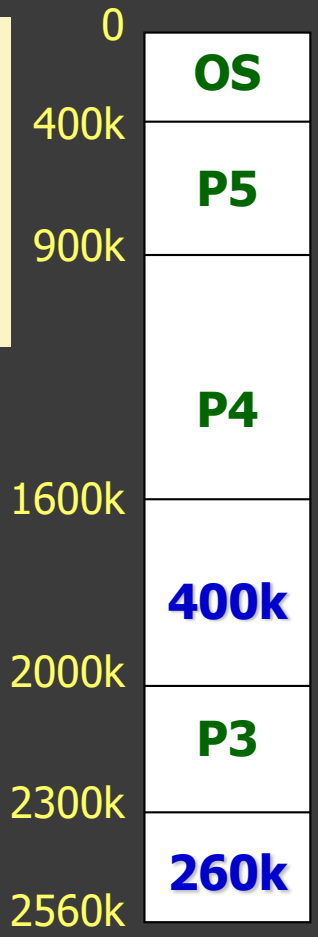
- Shuffle memory contents to place all free memory together in one large block.
- Compaction is possible only if relocation is dynamic, and is done at execution time.
  - Many ways to move programs in memory



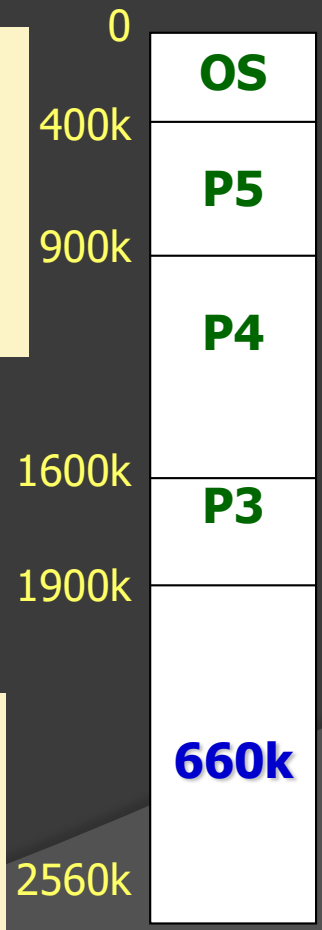
Compaction problem:



move **P4** up with 100k distance



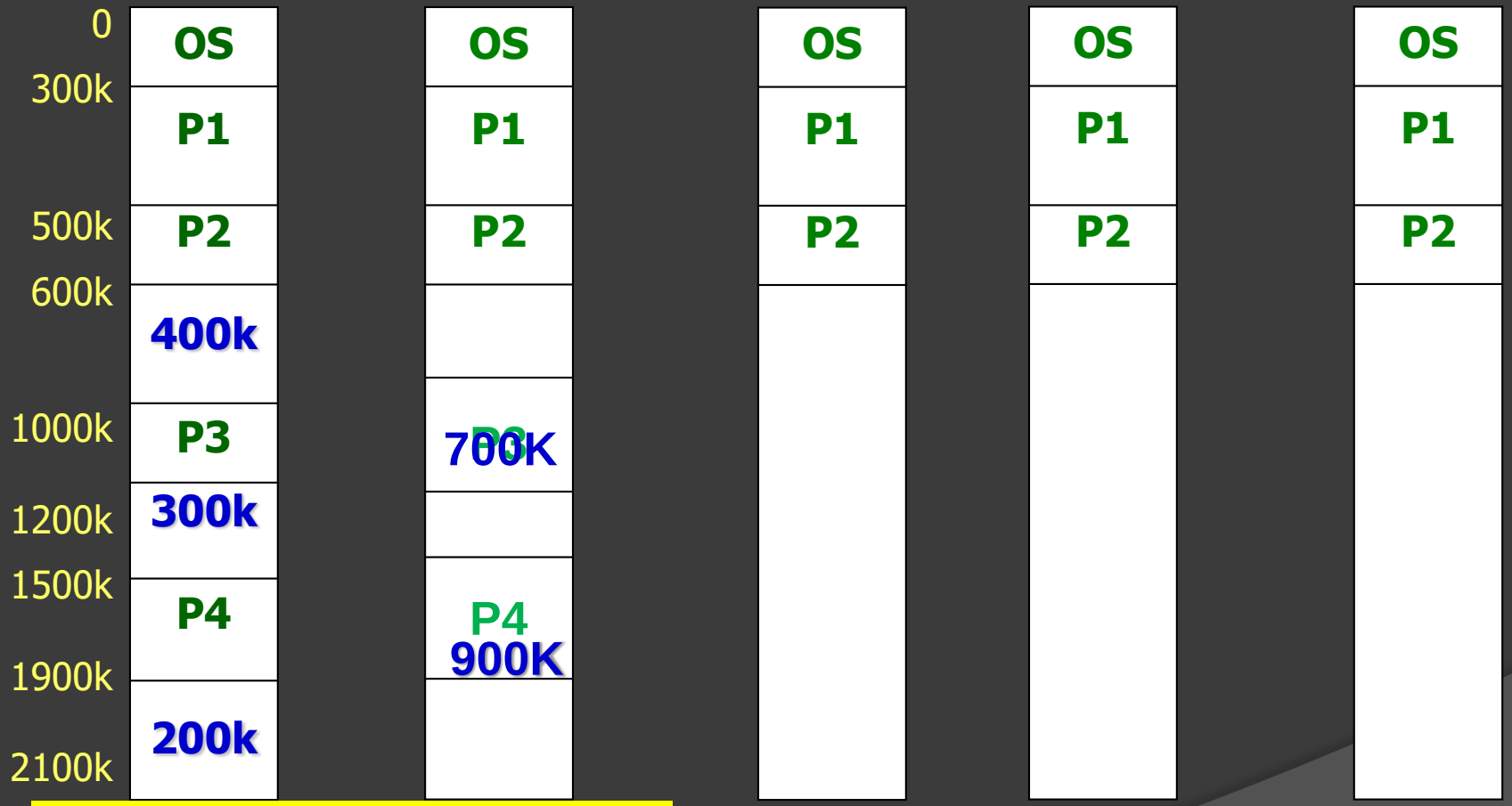
move **P3** up with 400k distance



300k + 100k merged together to form 400k

400k + 260k merged together to form 660k

Another Compaction Problem:



Total Cost = 400 + 700 = 1,100

If 100k of movement is worth Php100,
how much will each compaction algorithm costs?

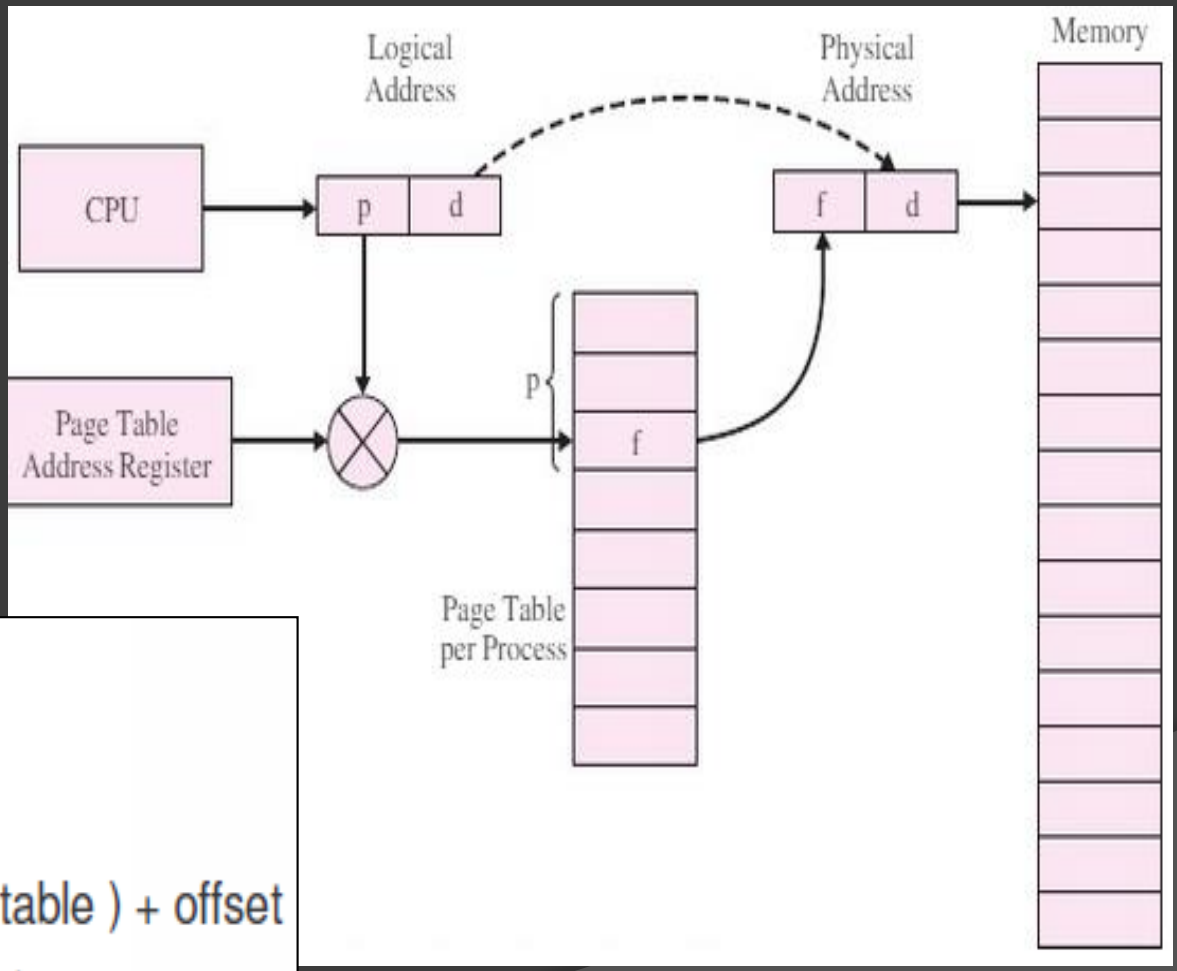
Solution 2: Paging

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

Basic Method for Paging

- Divide physical memory into fixed-sized blocks called **frames** (size: power of 2, normally from 512 to 8kbytes long. Normal: common page size: 4Kb)
- Divide logical memory into blocks of same size called **pages**
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- Set-up a page table to translate logical to physical address

Paging hardware



HARDWARE

An address is determined by:

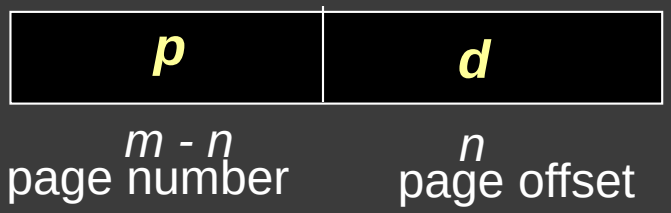
page number (index into table) + offset

---> mapping into --->

base address (from table) + offset.

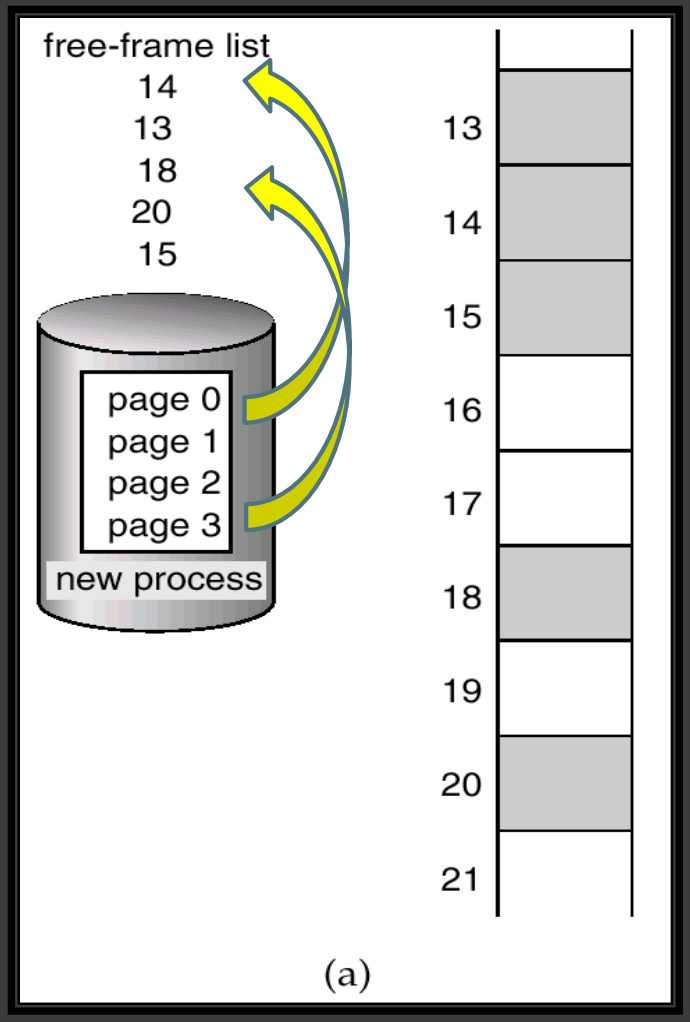
Address Translation Scheme

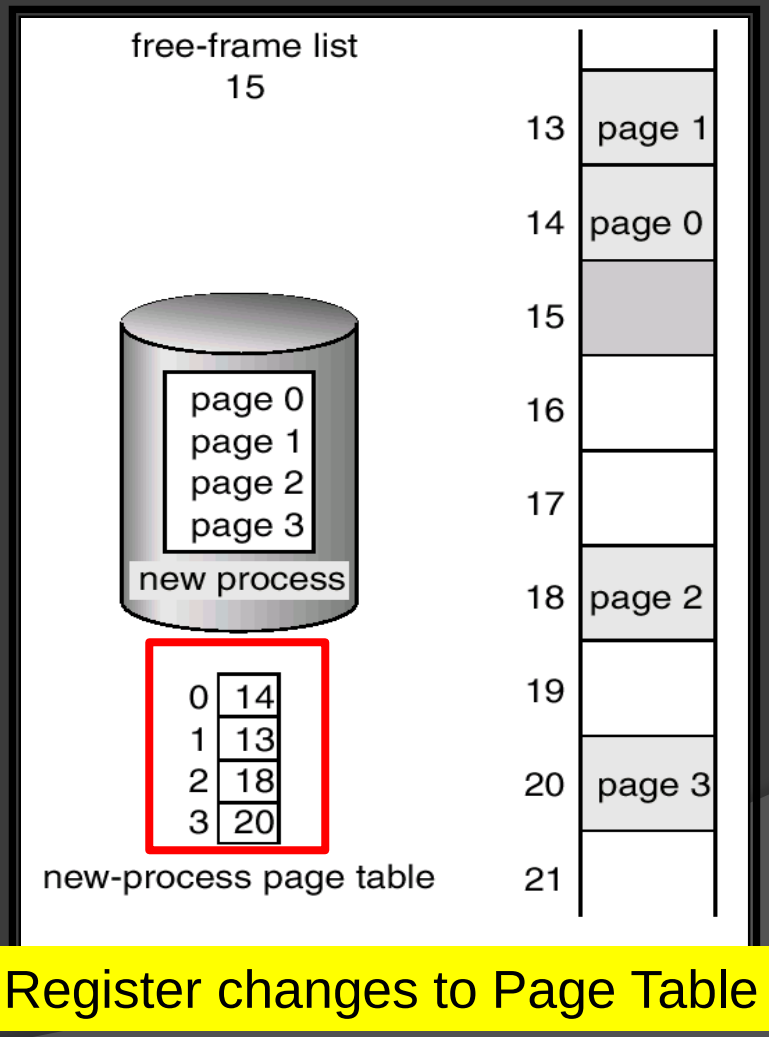
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



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

Paging implementation

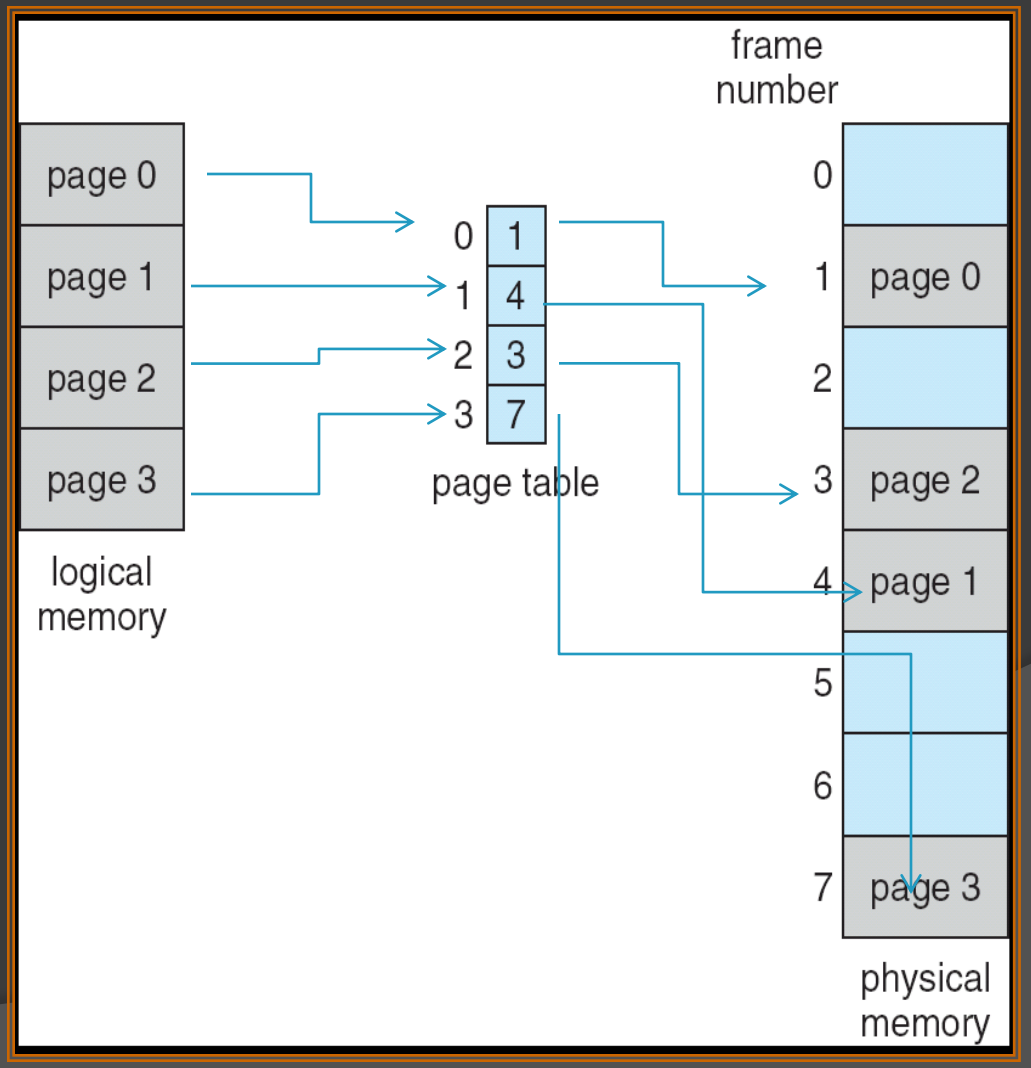




Register changes to Page Table

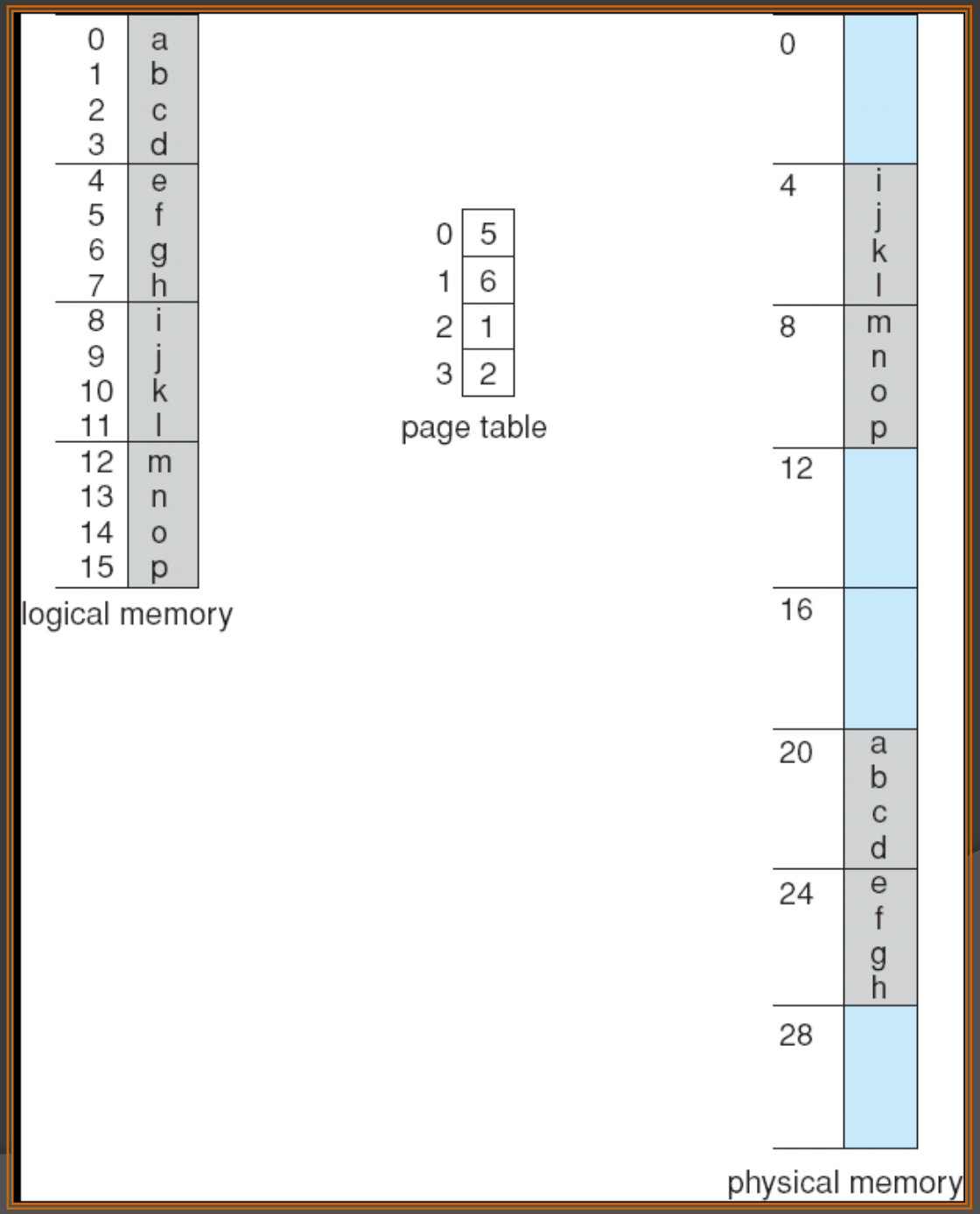
Sample 1

Consider the memory with page size of 4 bytes and has a physical memory of 32 bytes (8 pages), show how the user's view of the memory can be mapped into physical memory



Sample 2

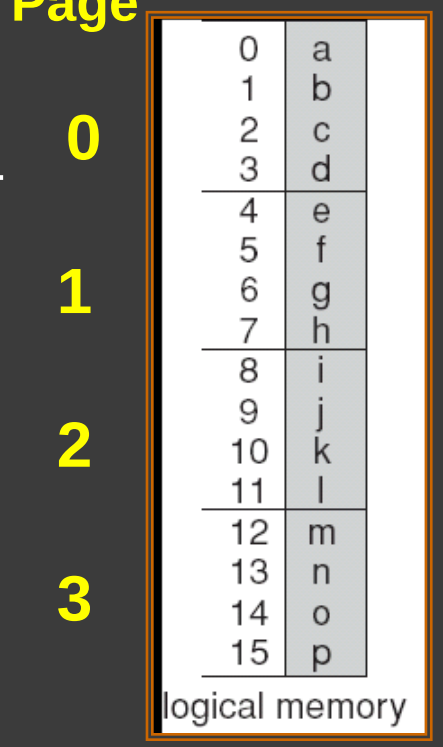
- Determine the physical address (PA) of the ff. instructions:
  - d
  - g
  - j
  - m
- PA = (frame number * frame size) + displacement



Solution:

- PA = (frame number * frame size) + displacement

1. **d** (belongs to page 0 w/ offset 3)
   => PA = (5 *4) + 3 = 23
2. **g** (belongs to page 1 w/ offset 2)
   => PA = (6 *4) + 2 = 26
3. **j** (belongs to page 2 w/ offset 1)
   => PA = (1 *4) + 1 = 5
4. **m** (belongs to page 3 w/ offset 0)
   => PA = (2 *4) + 0 = 8



Implementation of a Page Table

1. Using a Register – to determine the number of registers needed in the implementation of PT depends on the following:
   - Minimum number of registers – refer to the number of pages
   - Maximum number of registers – refer to the number of frames

Implementation of a Page Table

2. Using Main Memory

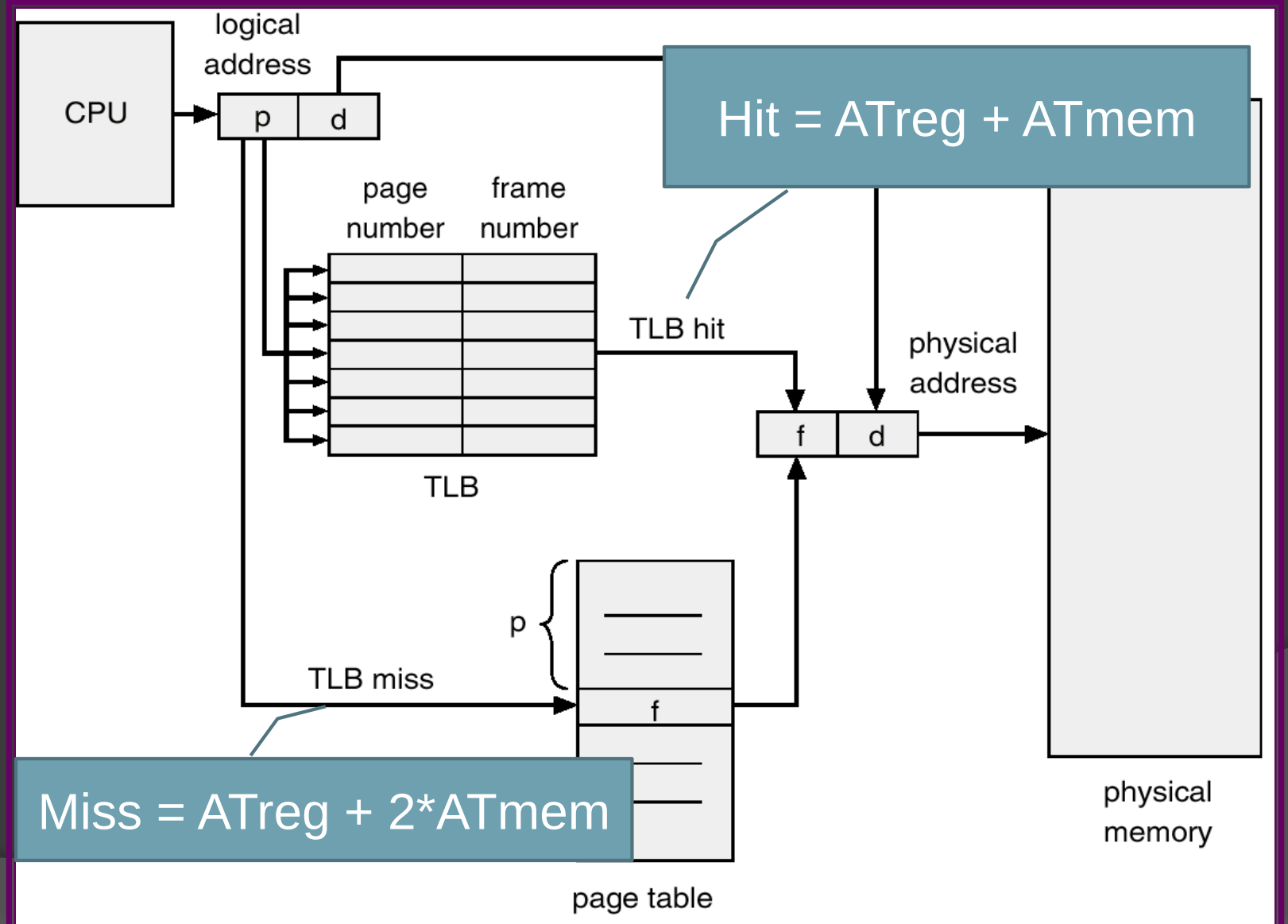
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- Every data/instruction access requires 2 memory accesses (1 for accessing the PT and 1 for the actual data/instruction)

Implementation of a Page Table

3. Combination of Register and Main Memory

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)**

Paging Hardware With TLB



Effective Mem. Access Time (EAT)

- EAT = (%HR x hit access time) +
  (%MR x miss access time)

  = HR (ATreg + ATmem) +
  (1-HR) (ATreg + 2ATmem)

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

Sample problem:

- *Consider a paging system with the page table stored in memory.*

  a. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

  b. If we add associative registers, and 75% of all page-table references are found in the associative registers, what is the effective memory reference? (Assume that finding a page-table entry in the associative registers take zero time, if the entry is there.)

Solution

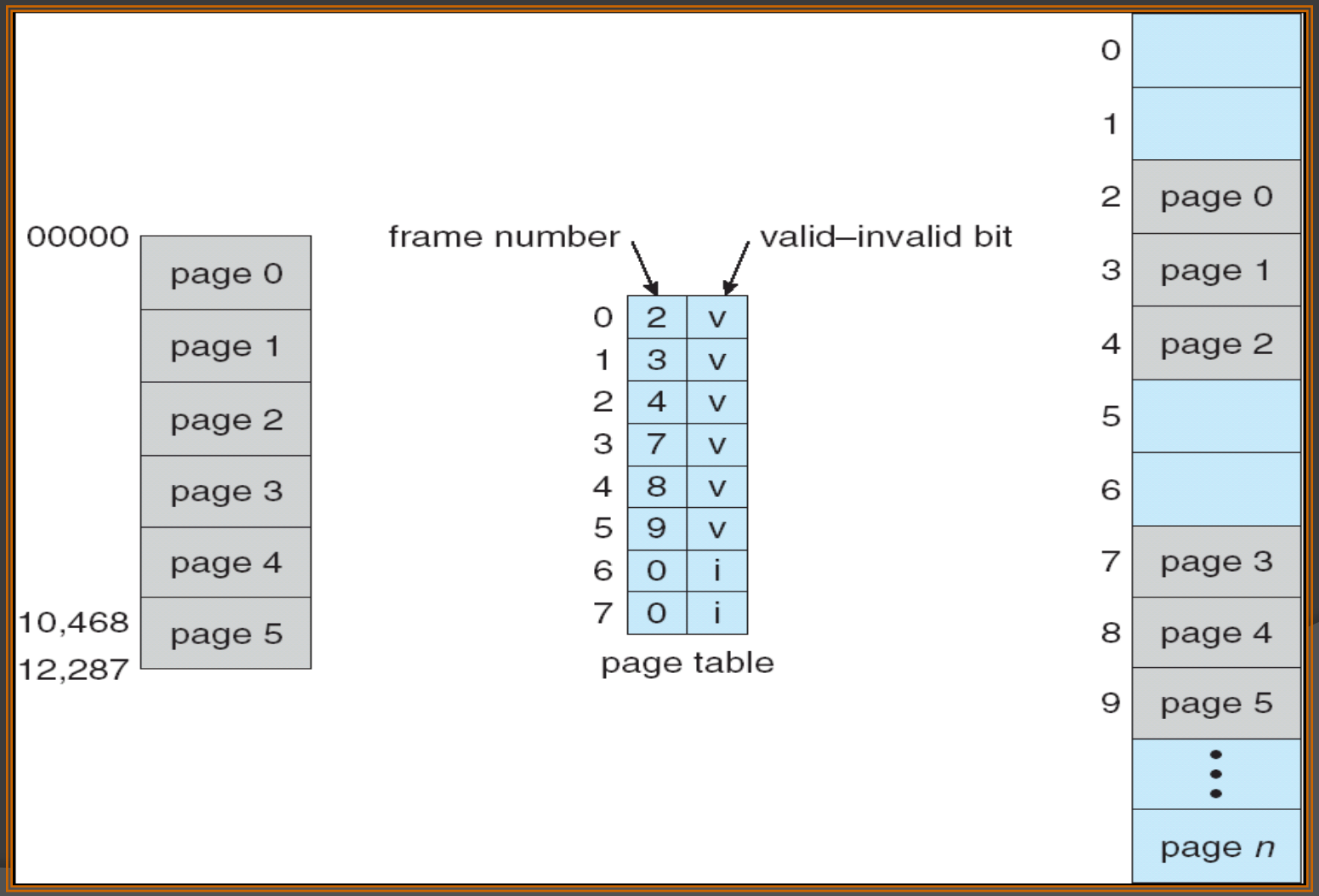
a. ATMem = 2(200nsec) = 400 nsec

b. AT = .75(0 + 200) + (.25) (0 + 2 * 200)
= 250 nsec.

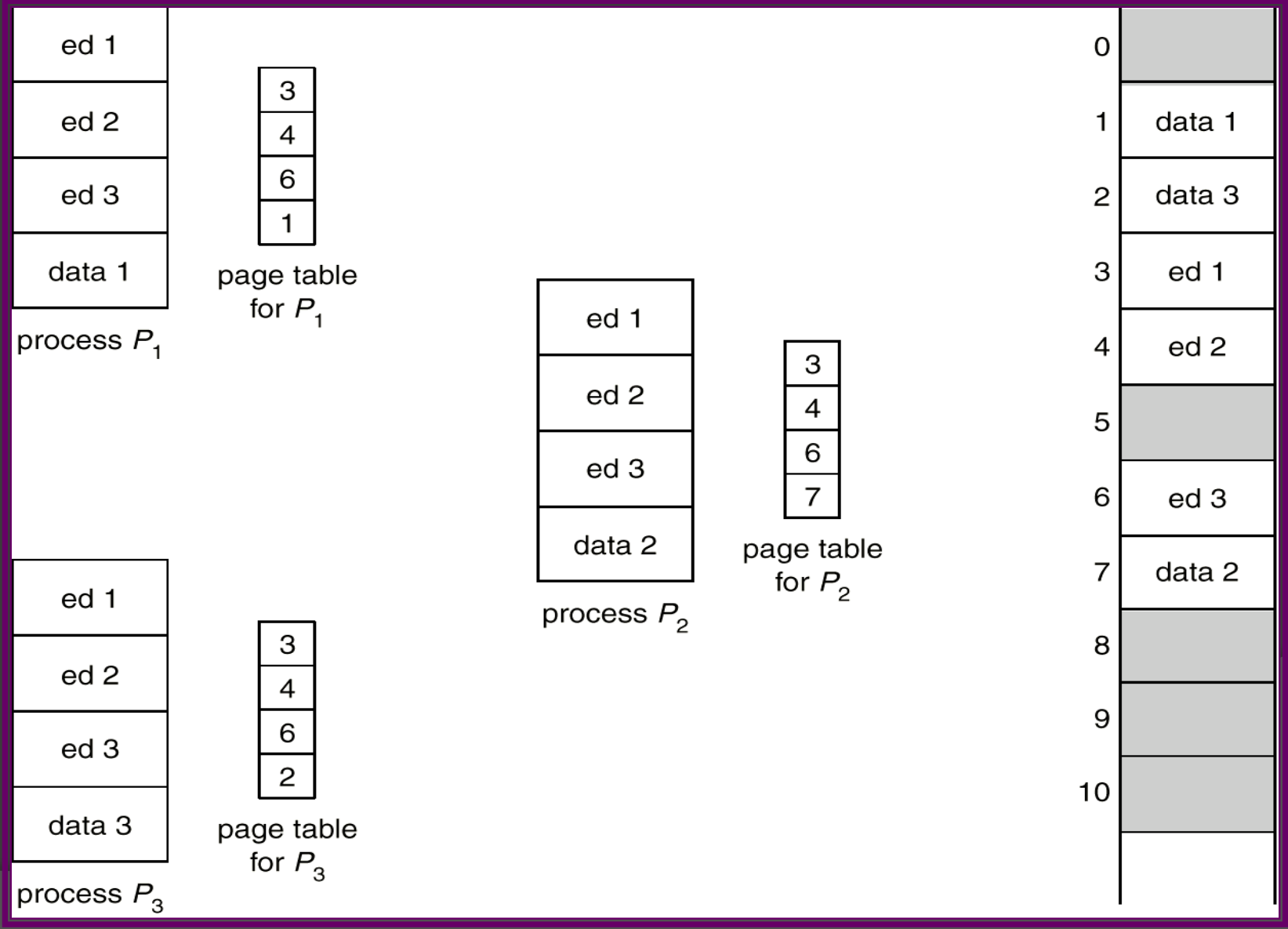
Page Access Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" - the associated page is in the process' logical address space, and is a legal page
  - "invalid" - the page is not in the process' logical address space

Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages



Paging Problem:

- A paging system with 2^{20} bytes of physical memory, 128 pages of logical address space, and a page size of 512 bytes,
  - How many frames are there?
  - How many bits are needed to form the logical address?
  - What is the frame size (in bytes)?
  - How many bits are needed to store an entry in the page table including the valid/invalid 1-bit for each entry (how wide is the page table)?
  - How many bits are needed to form the physical address?

Paging Problem: (cont)

- A paging system with 2^{20} bytes of physical memory, 128 pages of logical address space, and a page size of 512 bytes,
  - If the PT is implemented using registers, how many registers (max and minimum) are needed?
  - If the PT is part of the main memory with 250nsec access time, how long does a paged memory reference take?
  - If the PT is implemented using TLBs that takes 85nsec. and MM that takes 210nsec, what is the total access time if 60% of all memory references find their entries in the associative registers?

Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level paging but can also be multi-level paging scheme

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset

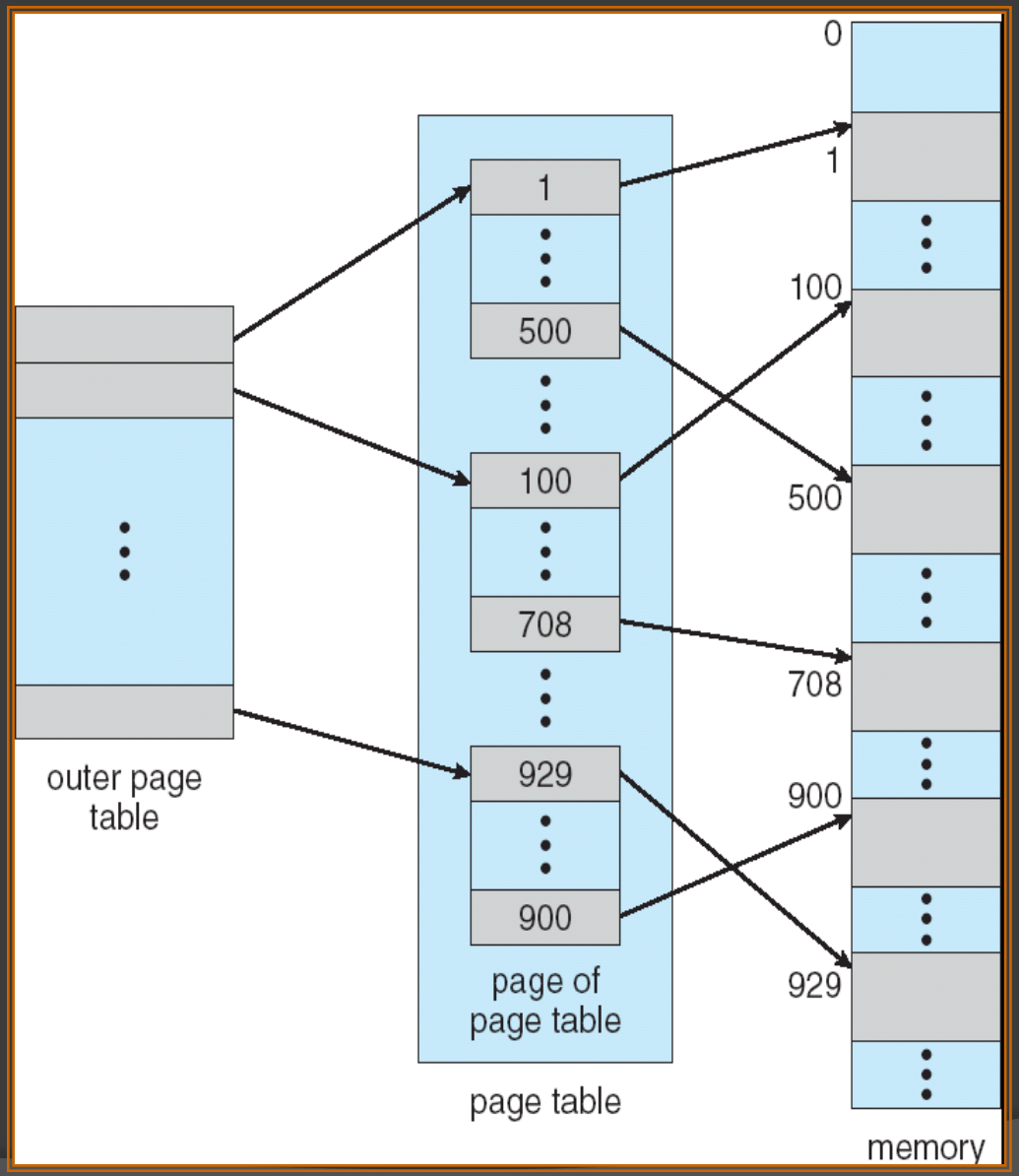
Two-Level Paging Example

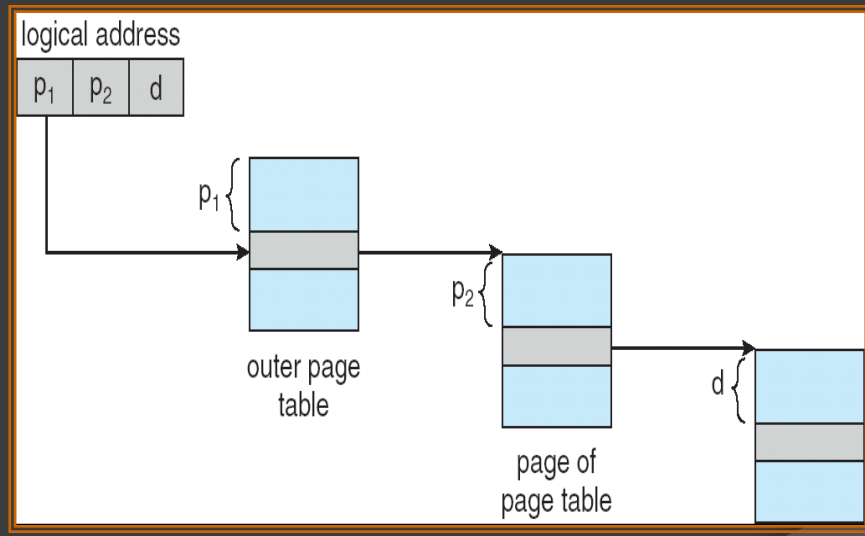
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12 = 32 bit machine

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

Two-Level Page-Table Scheme

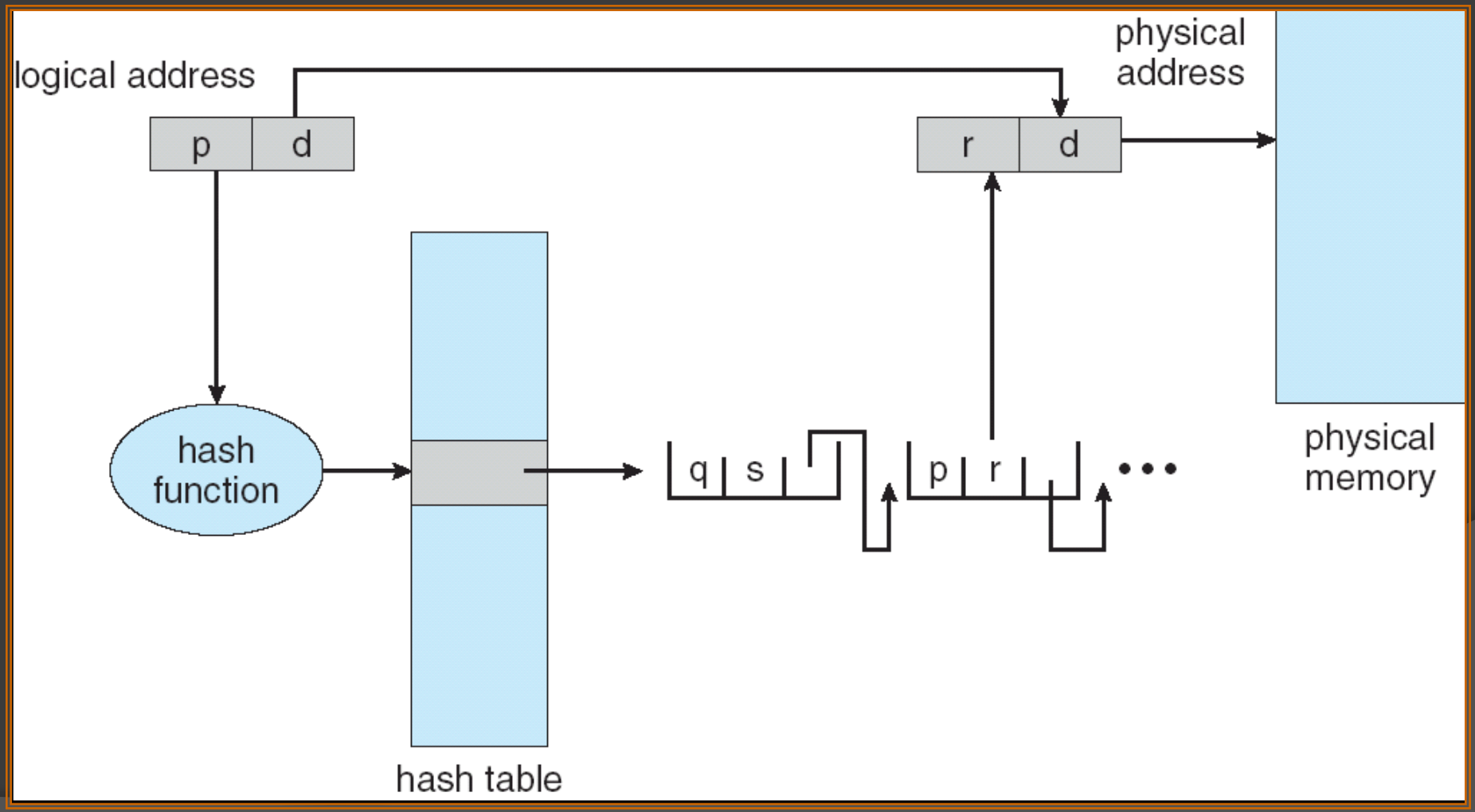




Hashed Page Tables

- Common in address spaces > 32 bits
- Virtual page number is hashed into a page table. 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.

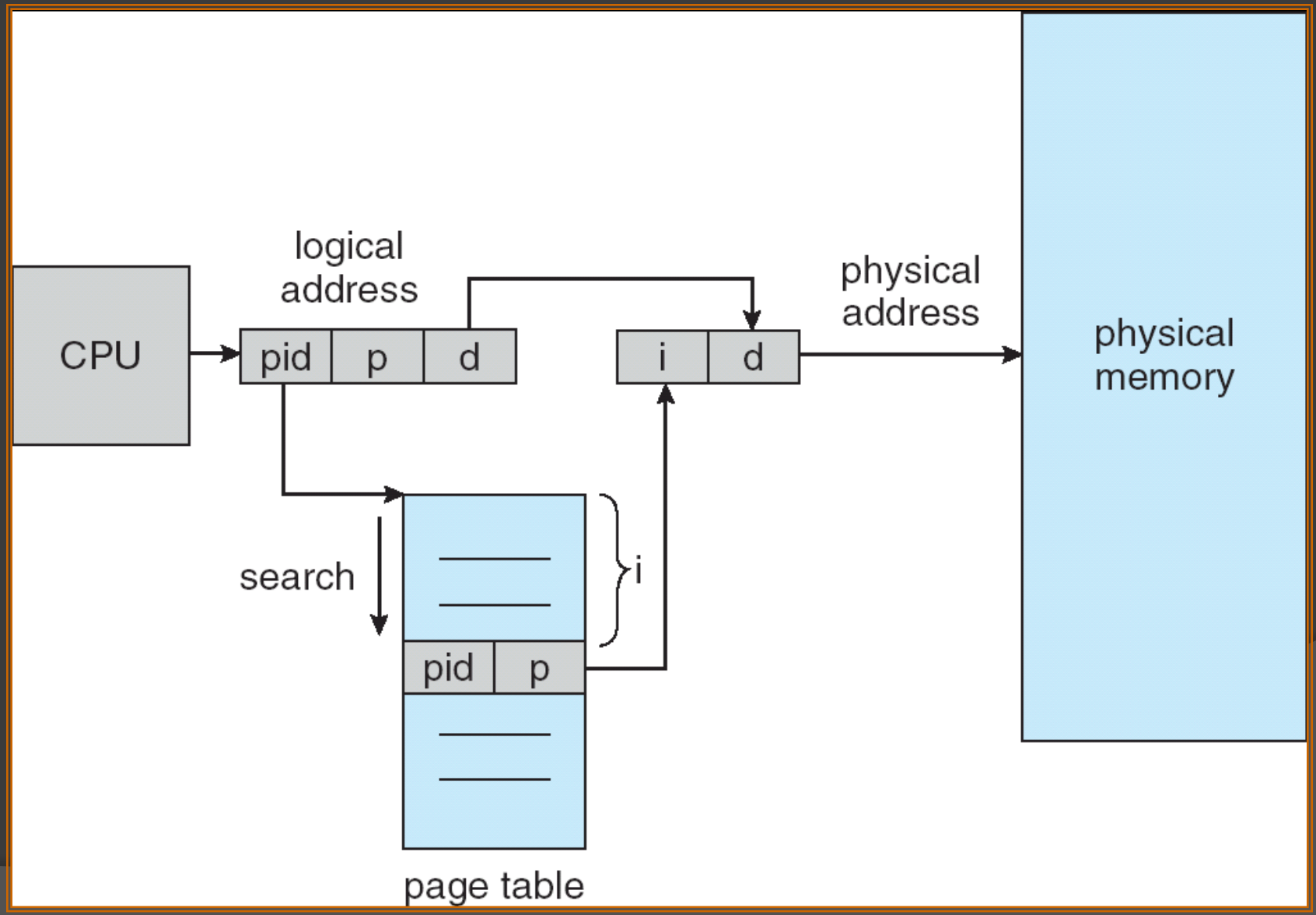
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- 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

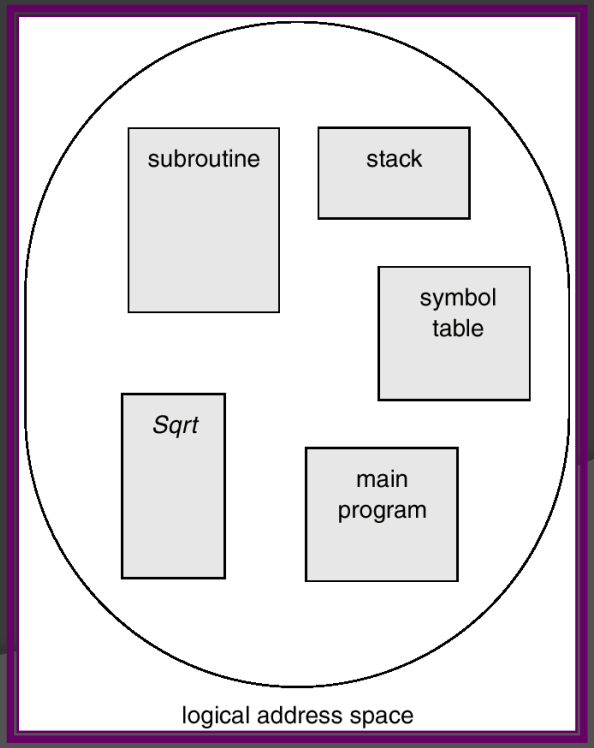


Solution #3: Segmentation

- 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



Segmentation Architecture

- Logical address consists of a two entries:

  <segment-number, offset>

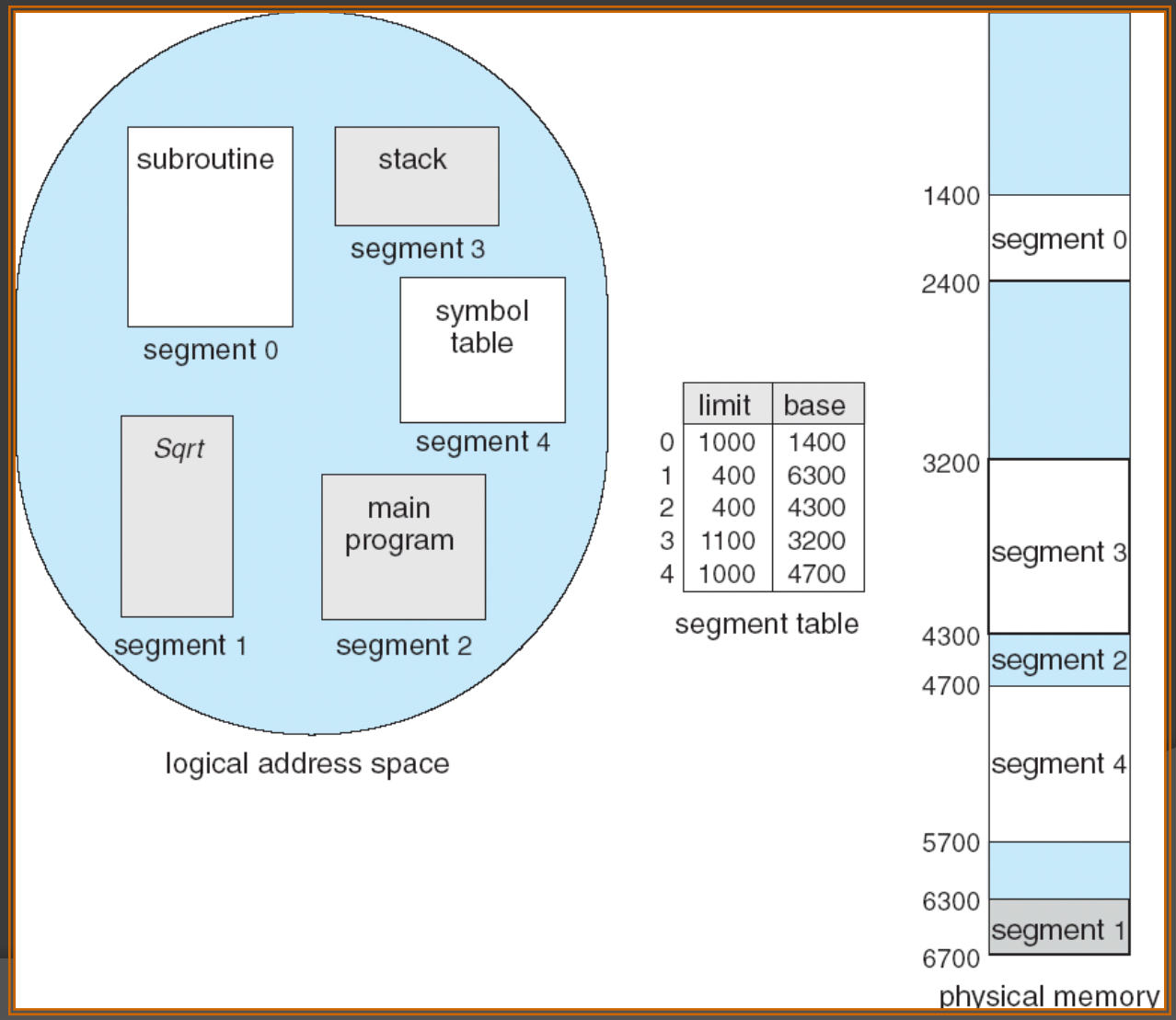
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment

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

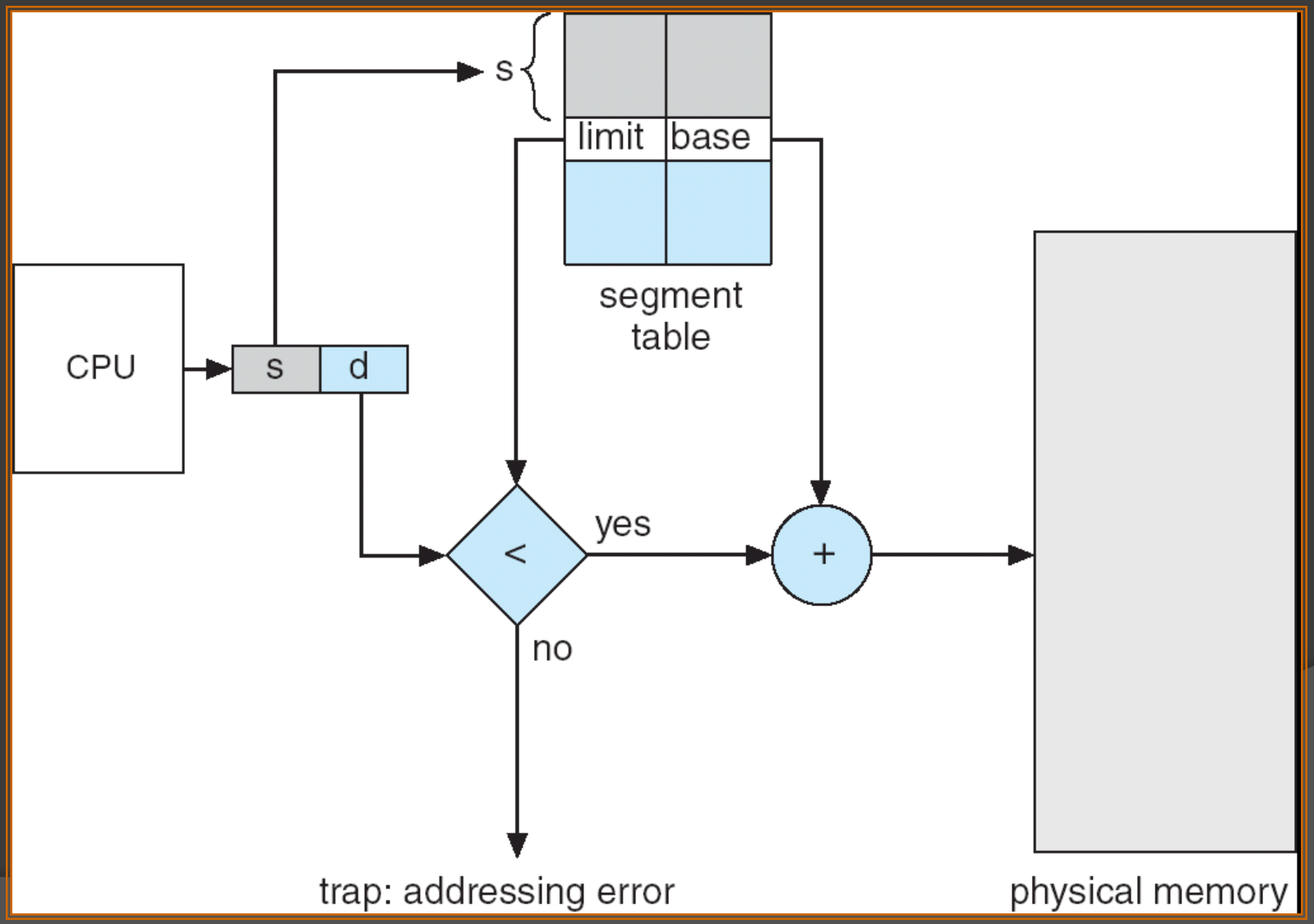
- **Segment-table length register (STLR)** indicates number of segments used by a program;

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

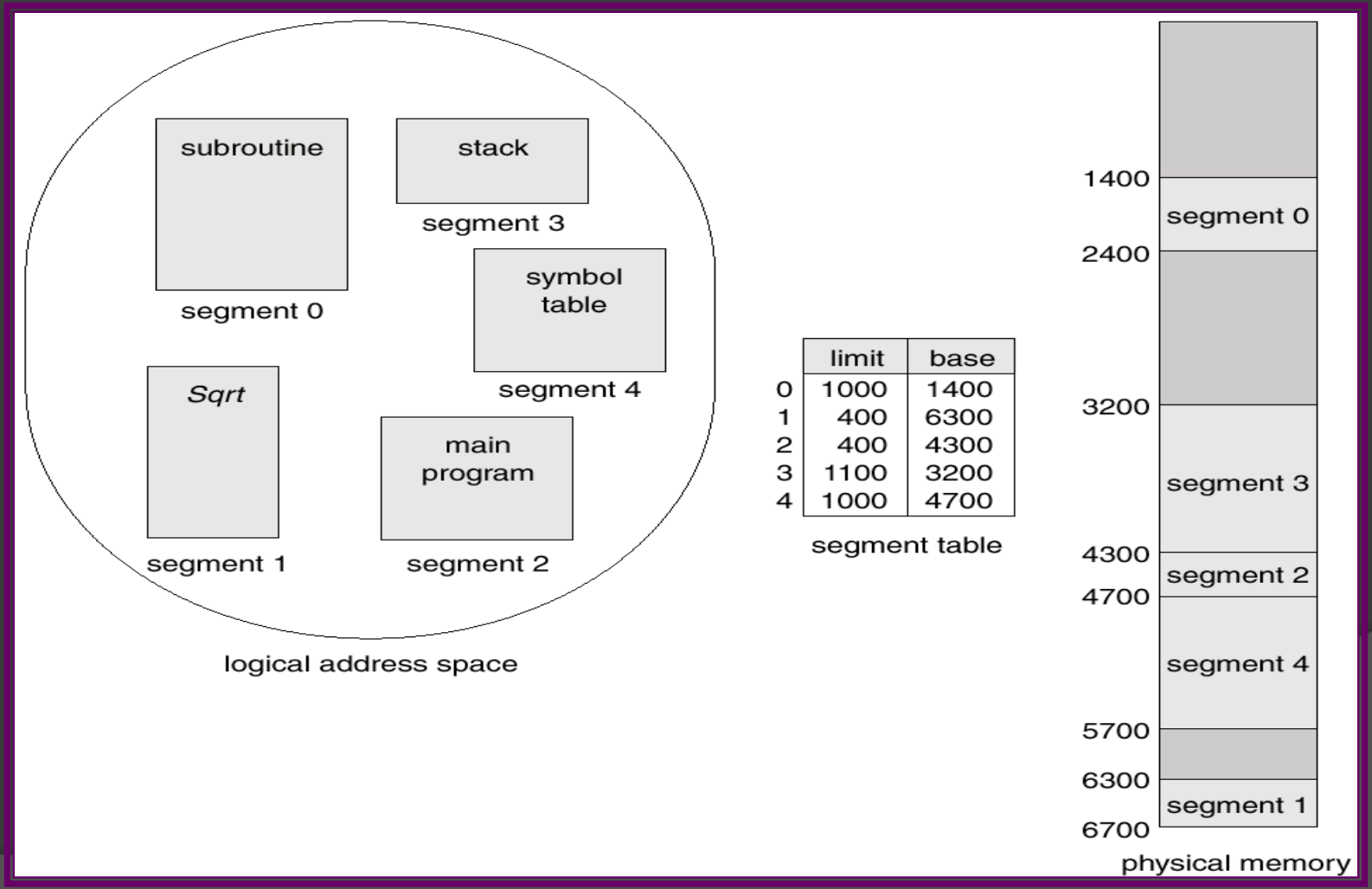
Example of Segmentation



Segmentation Hardware



Sharing of Segments



Sample problem:

- *Consider the ff. segment table:*

Segment	***Base***	***Length***
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

- **What are the physical addresses for the following logical addresses?**

a. 0, 430

b. 1, 10

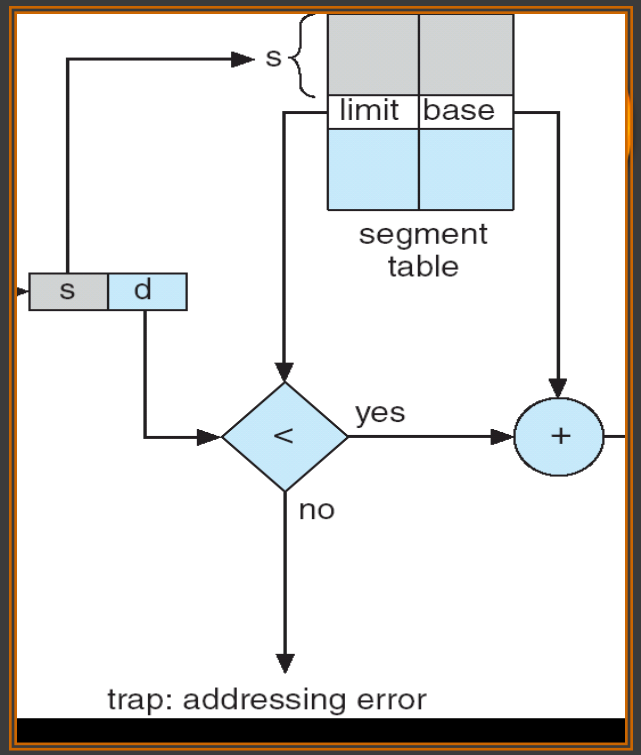
c. 2, 500

d. 3, 400

e. 4, 112

Solutions:

Segment #	*Base*	*Length*
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96



a. 0, 430

= Is 430 < 600 ? Yes

= 430 + 219

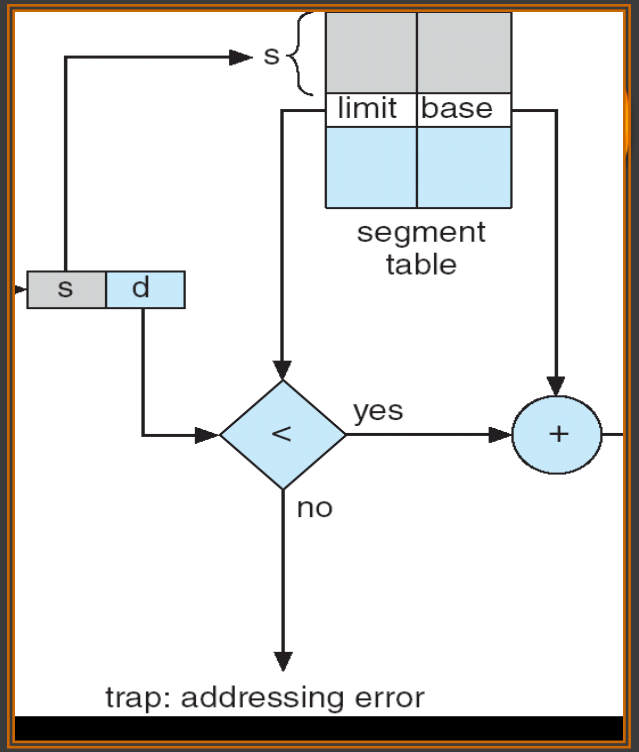
b. 1, 10

= Is 10 < 14 ? Yes

= 2300 + 10

Solutions:

Segment #	*Base*	*Length*
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96



c. 2, 500

= Is 500 < 100 ? No, therefore = TRAP

d. 3, 400 = Yes, thus 400 + 1327

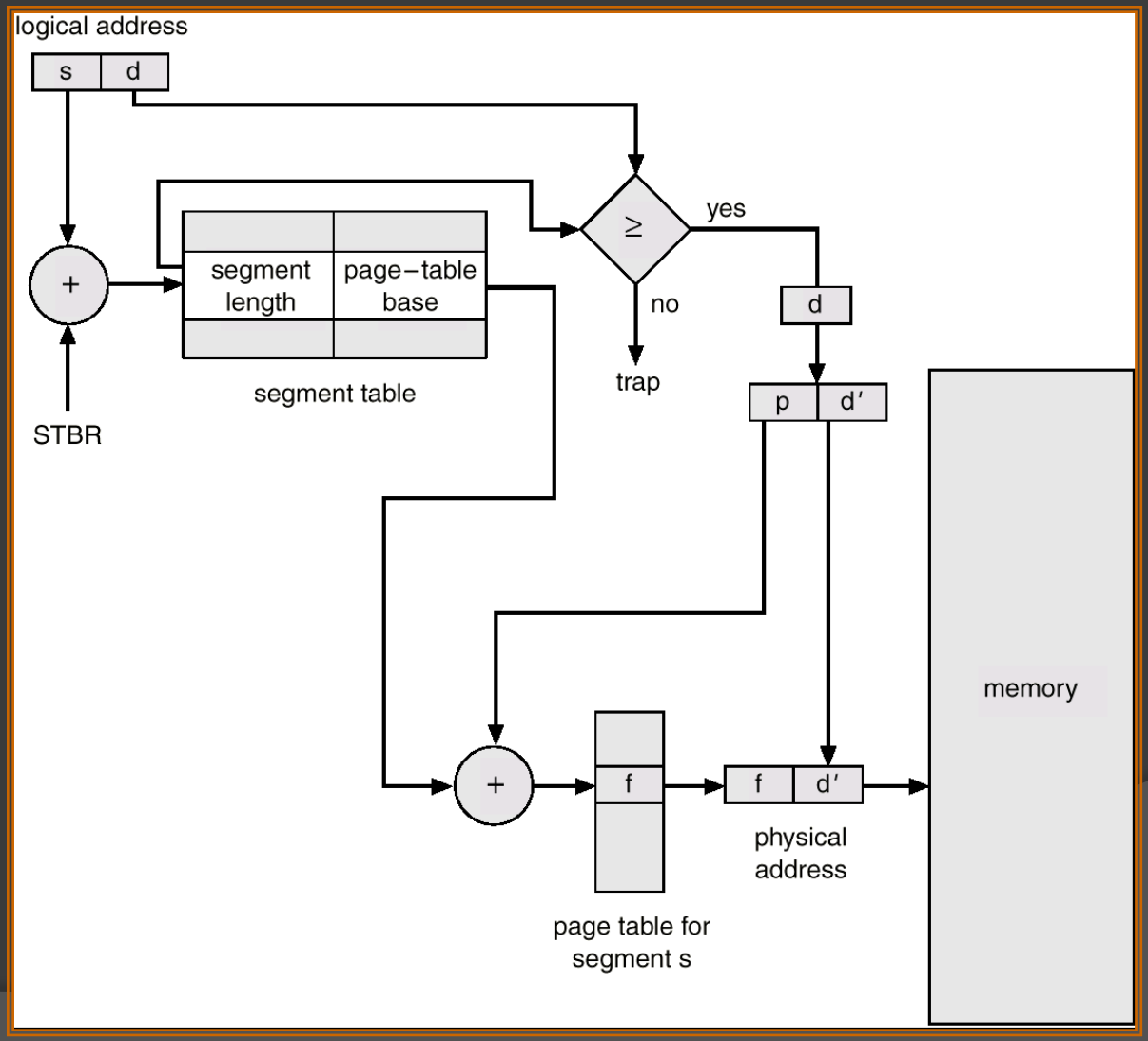
e. 4, 112 = No, thus TRAP

Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments

- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment

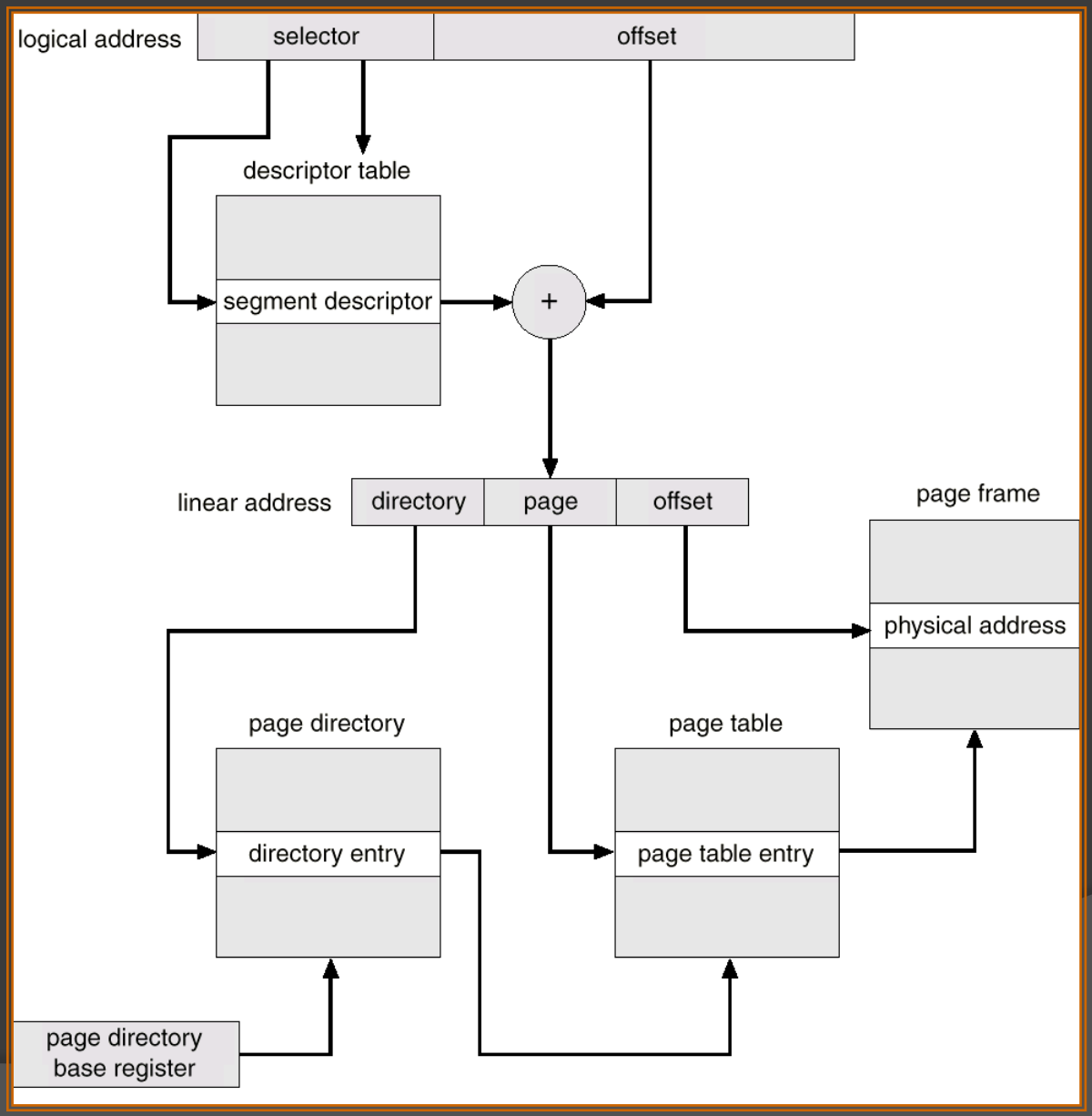
MULTICS Address Translation Scheme



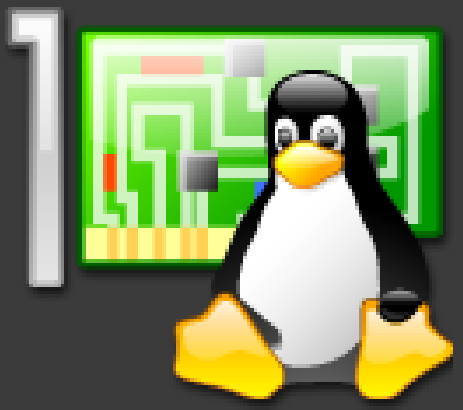
Segmentation with Paging – Intel 386

- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme

Intel 30386 Address Translation



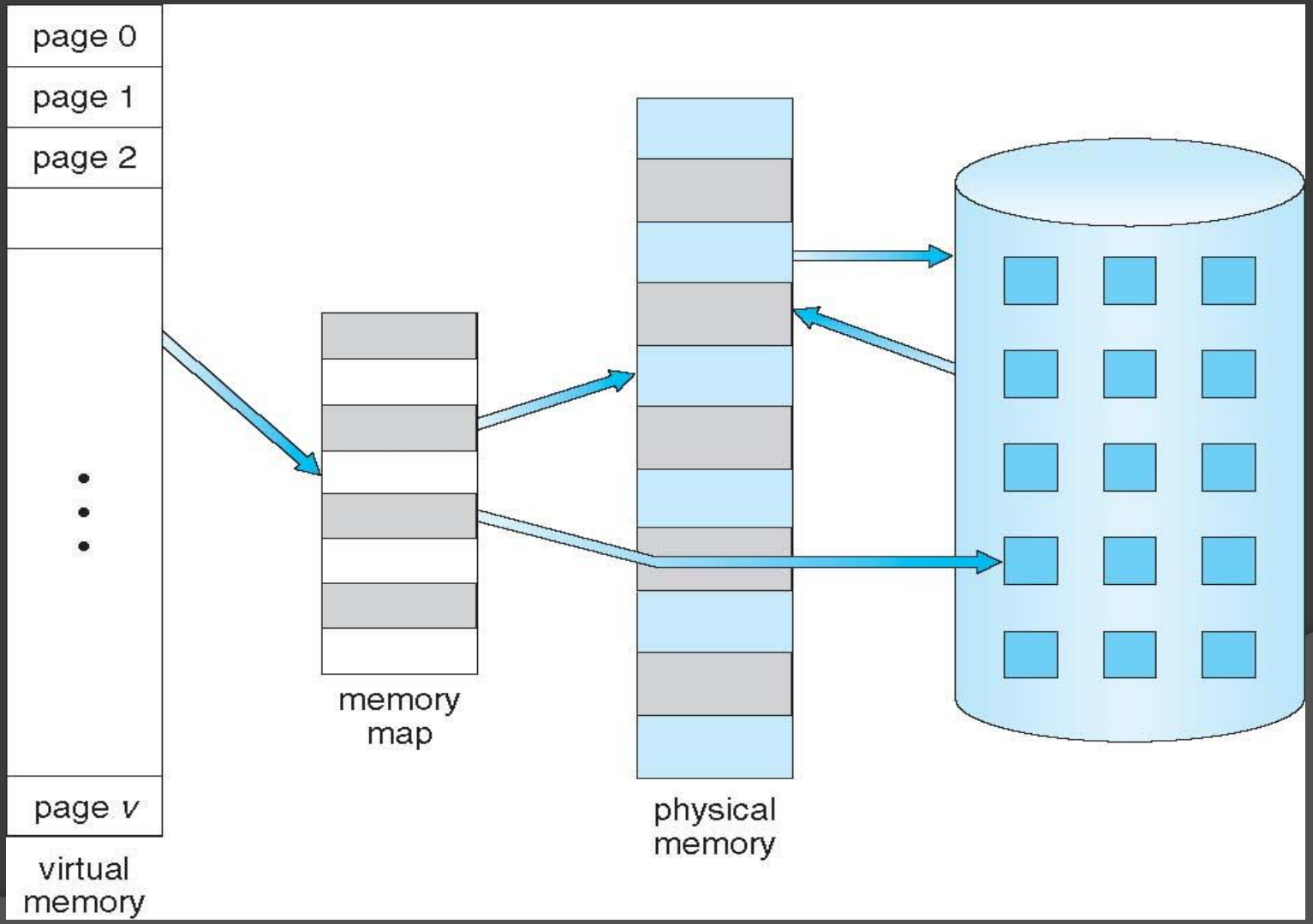
Virtual Memory



Virtual memory

- Technique allowing the execution of a process that may not be completely in memory.
- Only part of the program needs to be in memory for execution (anyway, the entire program is not needed or may not be needed at the same time)
- Logical address space can therefore be much larger than physical address space.

Virtual Memory That is Larger Than Physical Memory



Virtual Memory Implementation

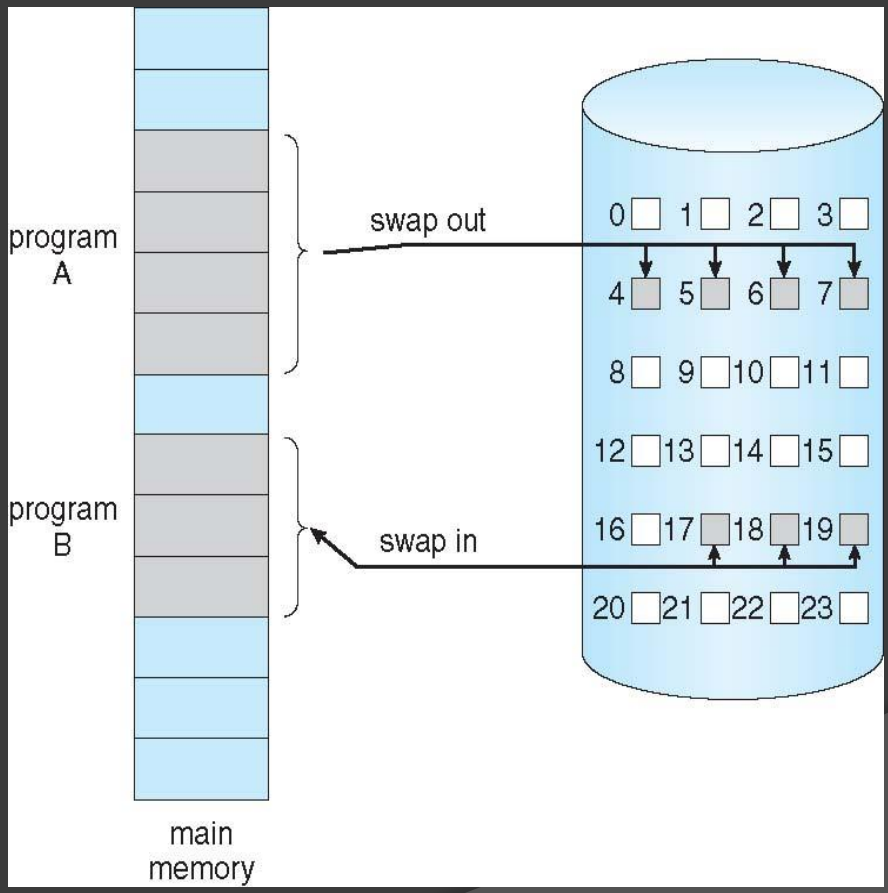
- Demand Paging
- Demand Segmentation

Demand Paging

- Bring entire process into memory at load time
- Or 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

Use of a Lazy swapper

- Never swaps a page into memory unless that page will be needed
- It guesses which pages will be used before the process is swapped out again



Transfer of a Paged Memory to Contiguous Disk Space

Valid-Invalid Bit

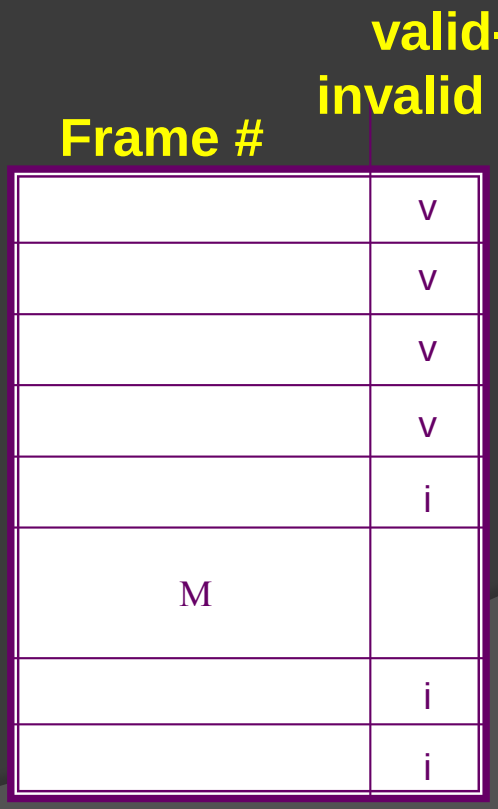
- With each page table entry a valid–invalid bit is associated.

  v ⇒ in-memory,

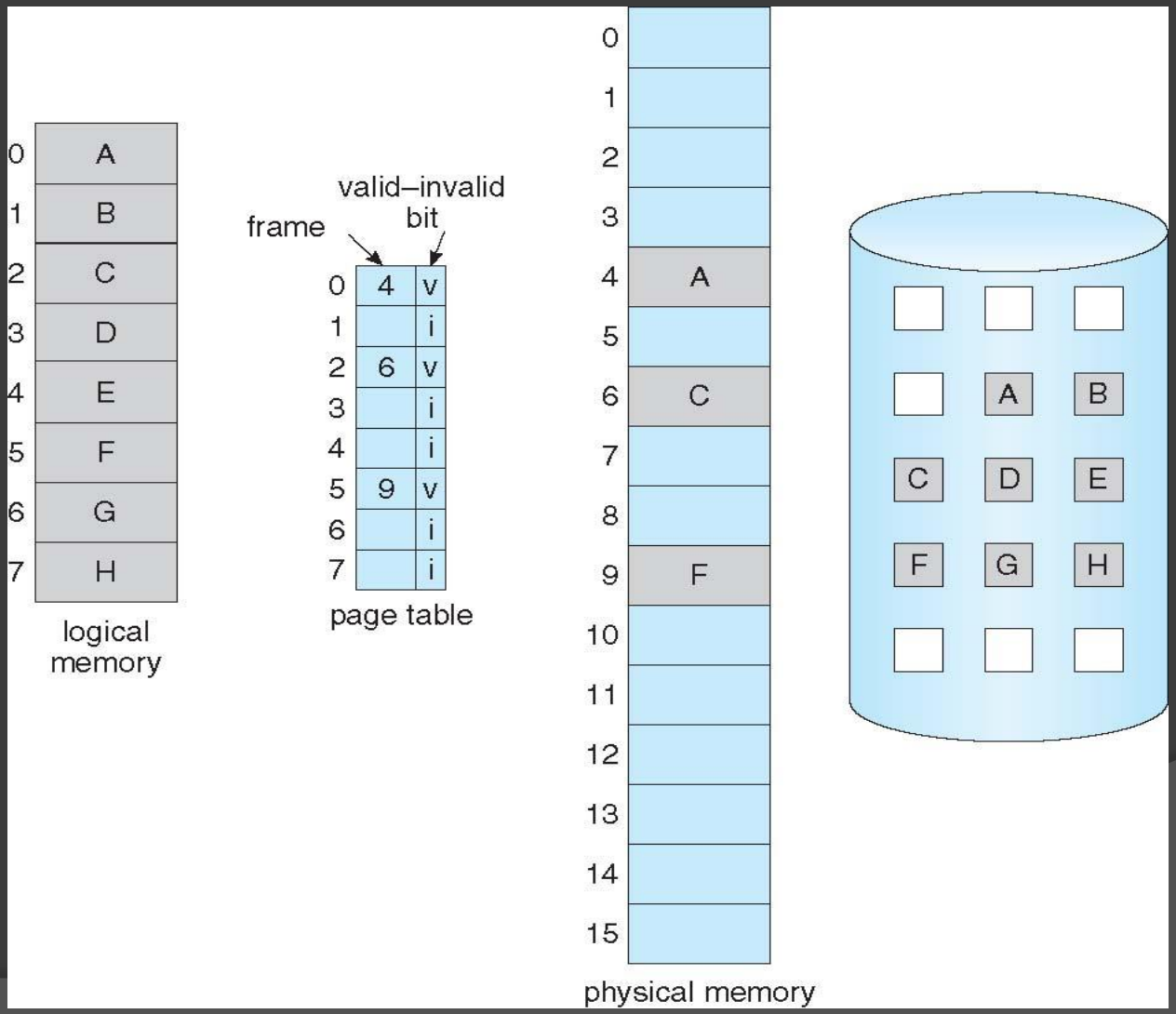
  i ⇒ not-in-memory

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

- During address translation, if valid–invalid bit in PT entry is i then there is a page fault.



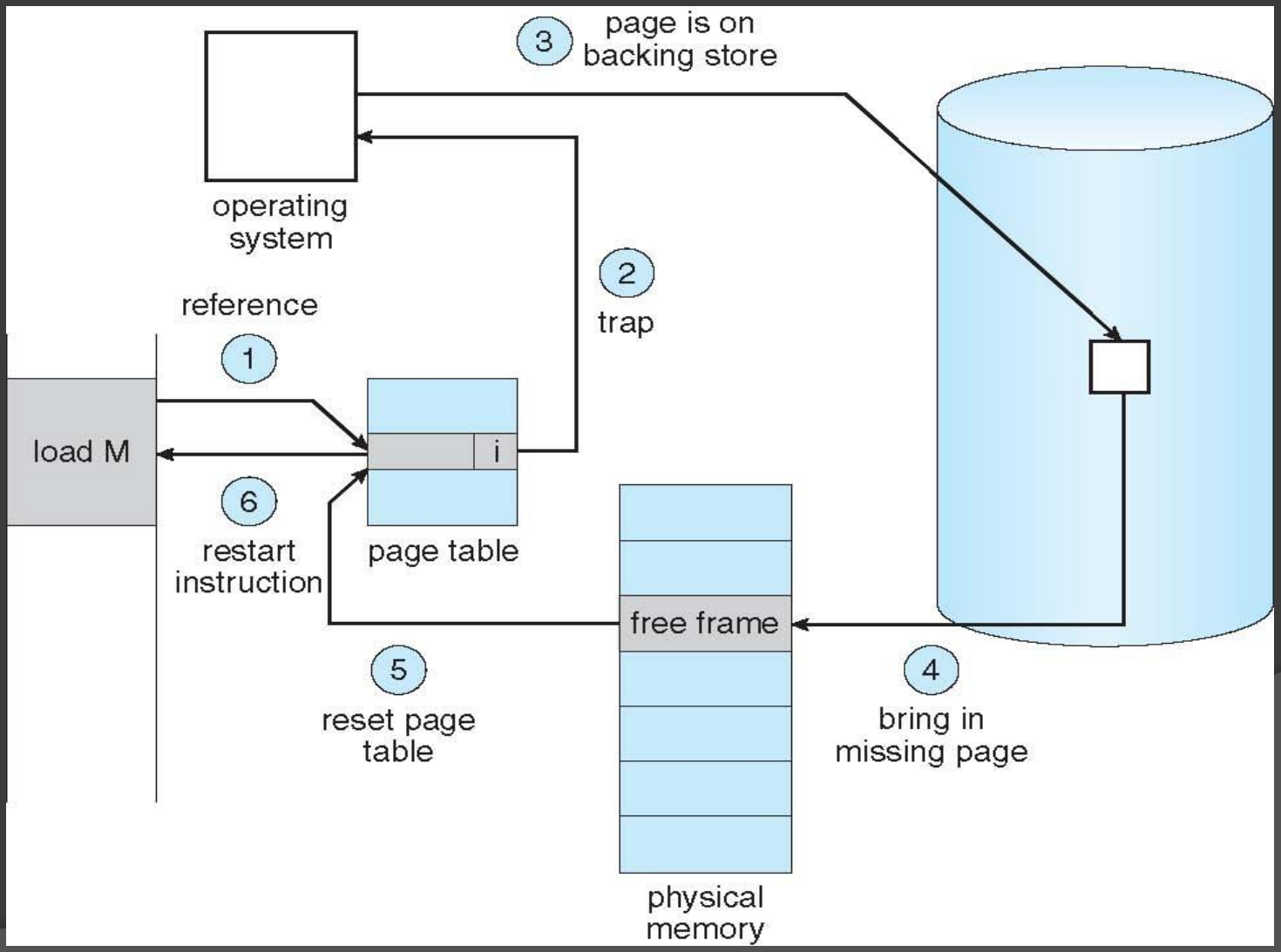
Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is ever a reference to a page, first reference will trap to OS $\Rightarrow$ page fault
- OS looks at another table to decide:
- Invalid reference $\Rightarrow$ abort.
- Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = v.
- Restart instruction: Least Recently Used

Steps in handling Page Fault



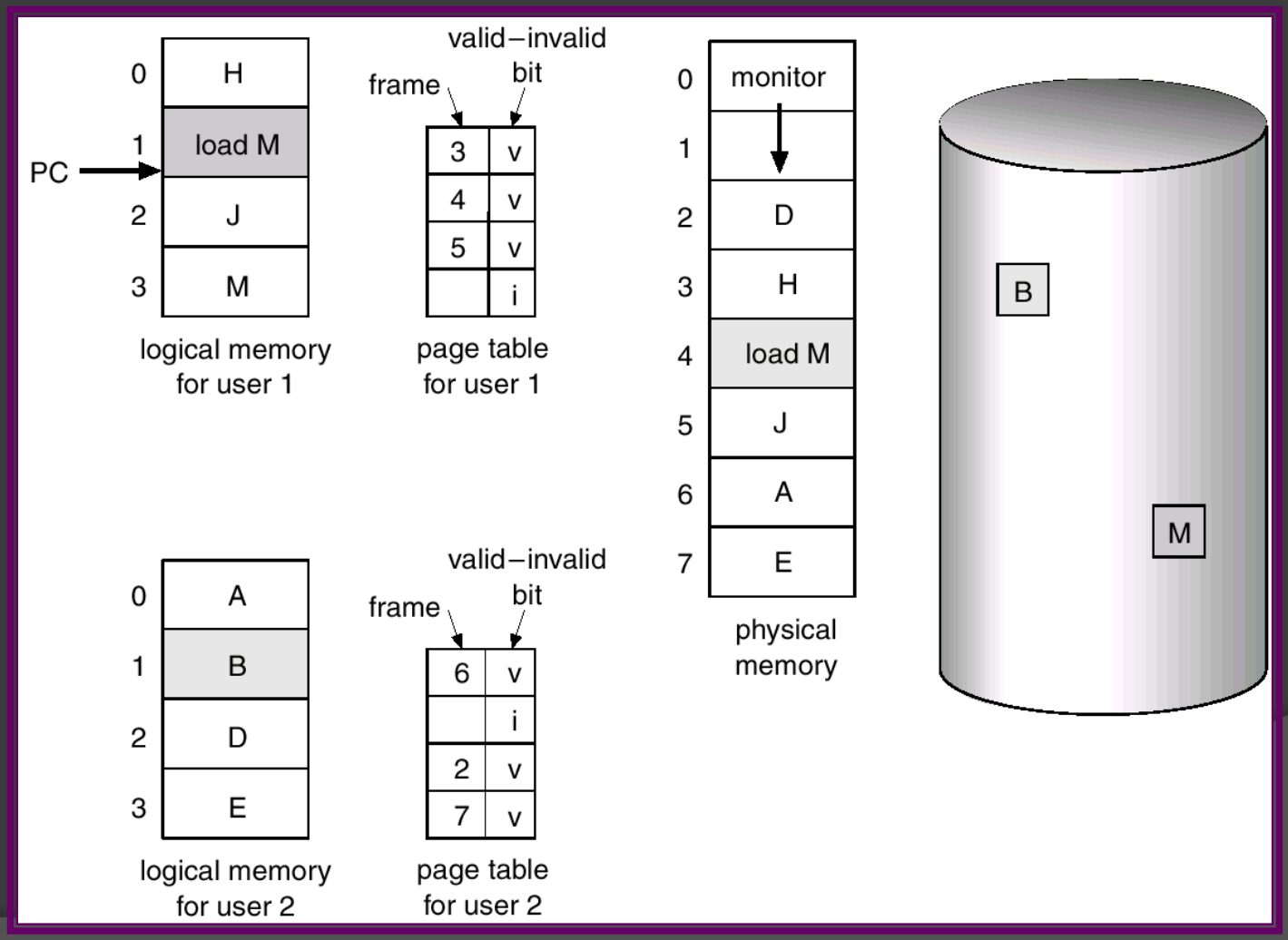
Steps in Handling a Page Fault

- **What happens if there is no free frame?**
- Implement page replacement algorithm
  - **Page replacement** – find some page in memory, but not really in use, swap it out.
  - Evaluate Algorithm to use in terms of:
    - Performance – want an algorithm which will result in minimum number of page faults.

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.

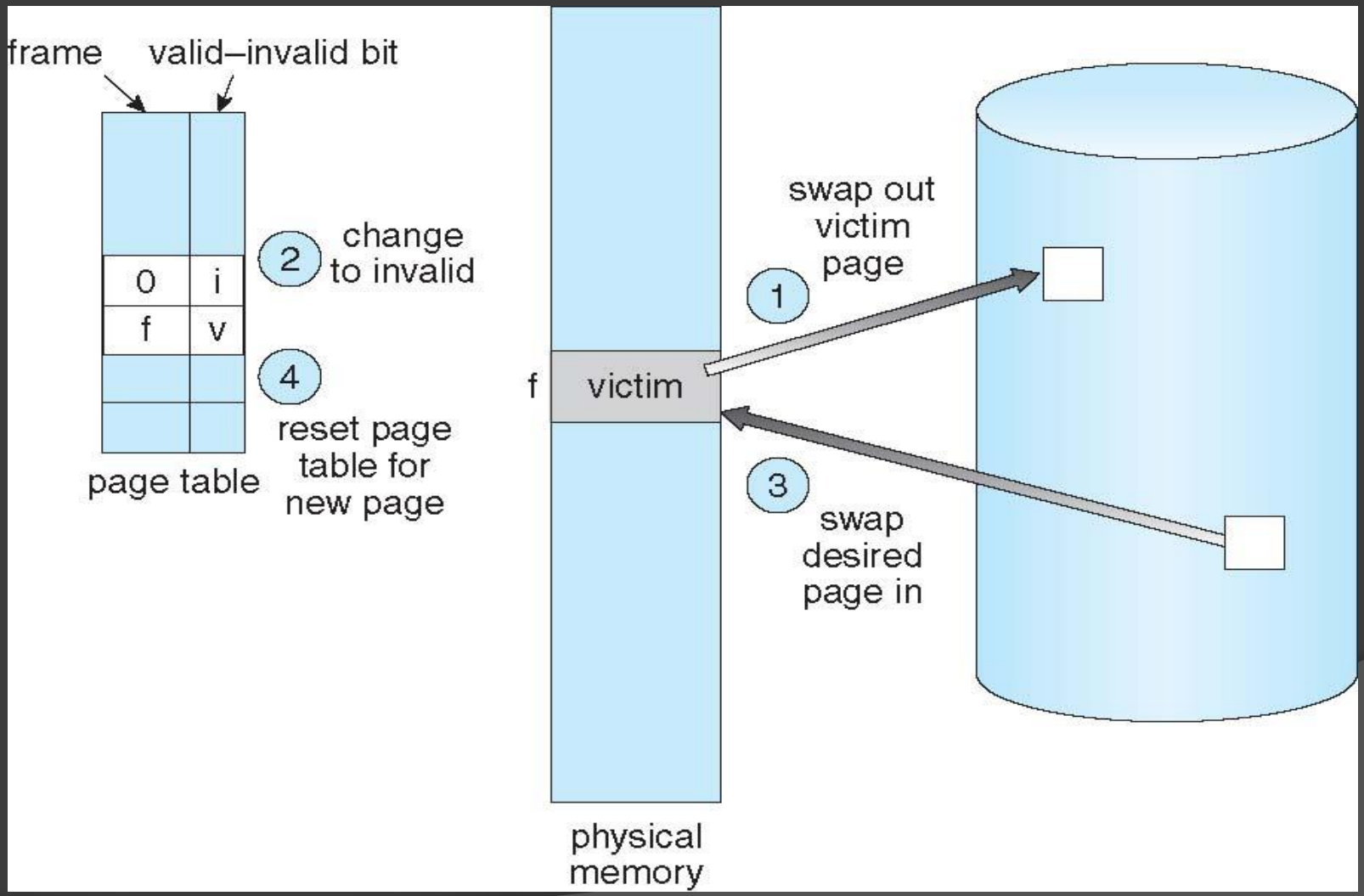
Page Replacement (cont.)



Basic Page Replacement

1. Find the location of the desired page on disk.
2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a *victim* frame.
3. Read the desired page into the (newly) free frame. Update the page and frame tables.
4. Restart the process.

Basic Page Replacement (cont.)



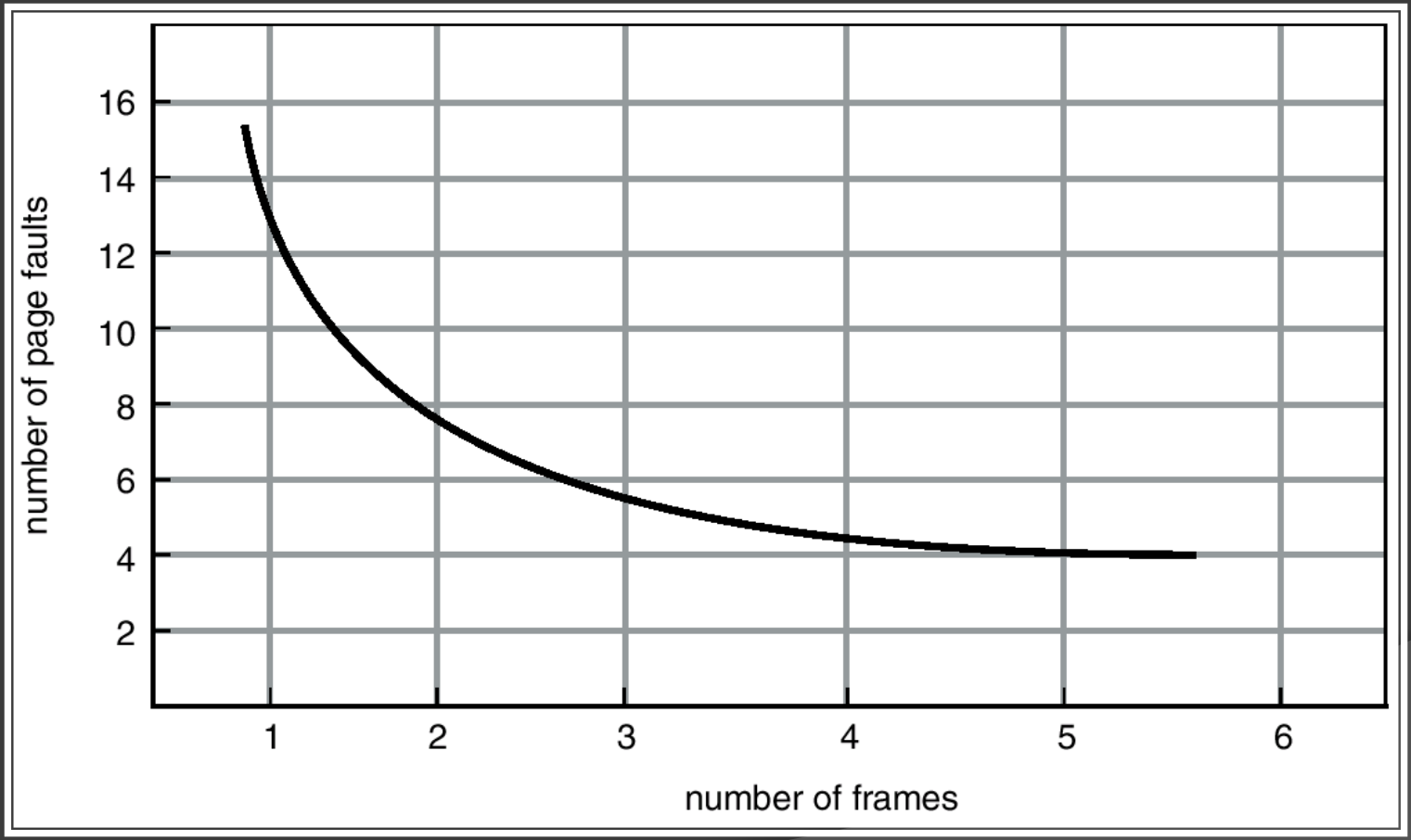
Page Replacement Algorithms

- **First-In-First-Out (FIFO) Algorithm**
- **Least Recently Used (LRU) Algorithm**
- **Optimal Algorithm**
- **LFU** and **MFU - additional algos.**

Note: Need to have the lowest page-fault rate.

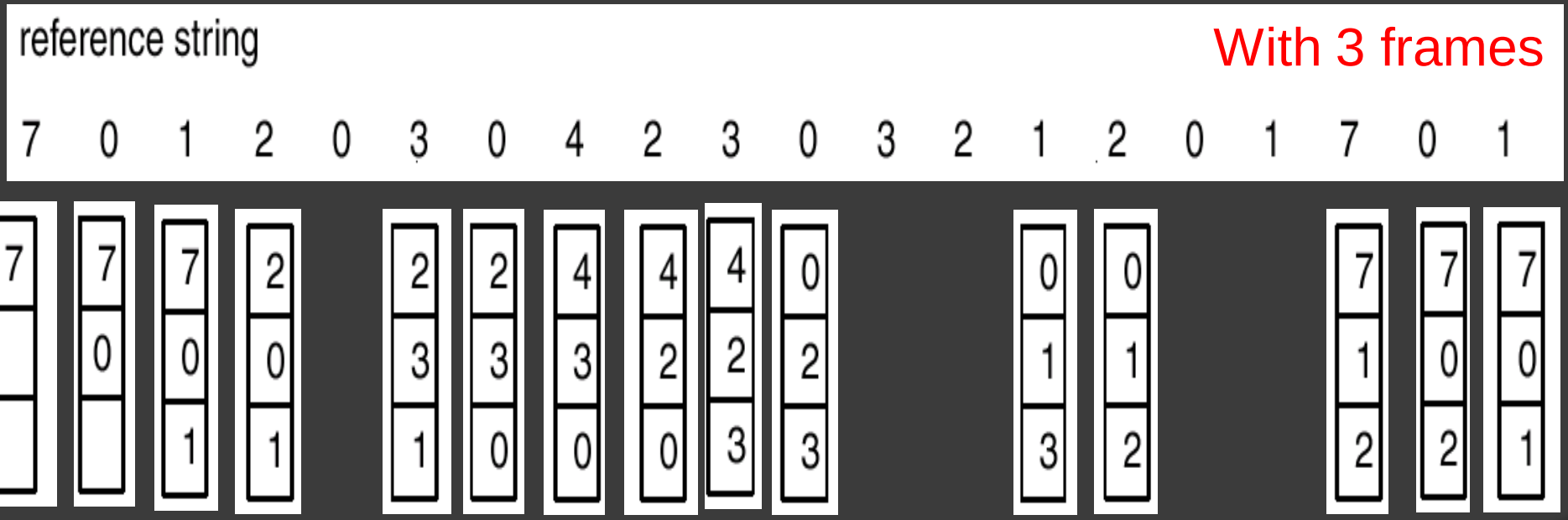
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

Relationship between number of frames with number of page faults



First-In-First-Out (FIFO) Algorithm

- The oldest page will be replaced.



PF = 15

How to track ages of pages?

- Just use a FIFO queue

Another FIFO algo Example

- Given the following reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, show how FIFO page replacement algorithm is implemented with the following number of frames.

1	4	5
2	1	3
3	2	4

PF = 9

Another FIFO algo Example

- Given the following reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, show how FIFO page replacement algorithm is implemented with the following number of frames.

1	5	4	
2	1	5	
3	2		
4	3		

PF = 10

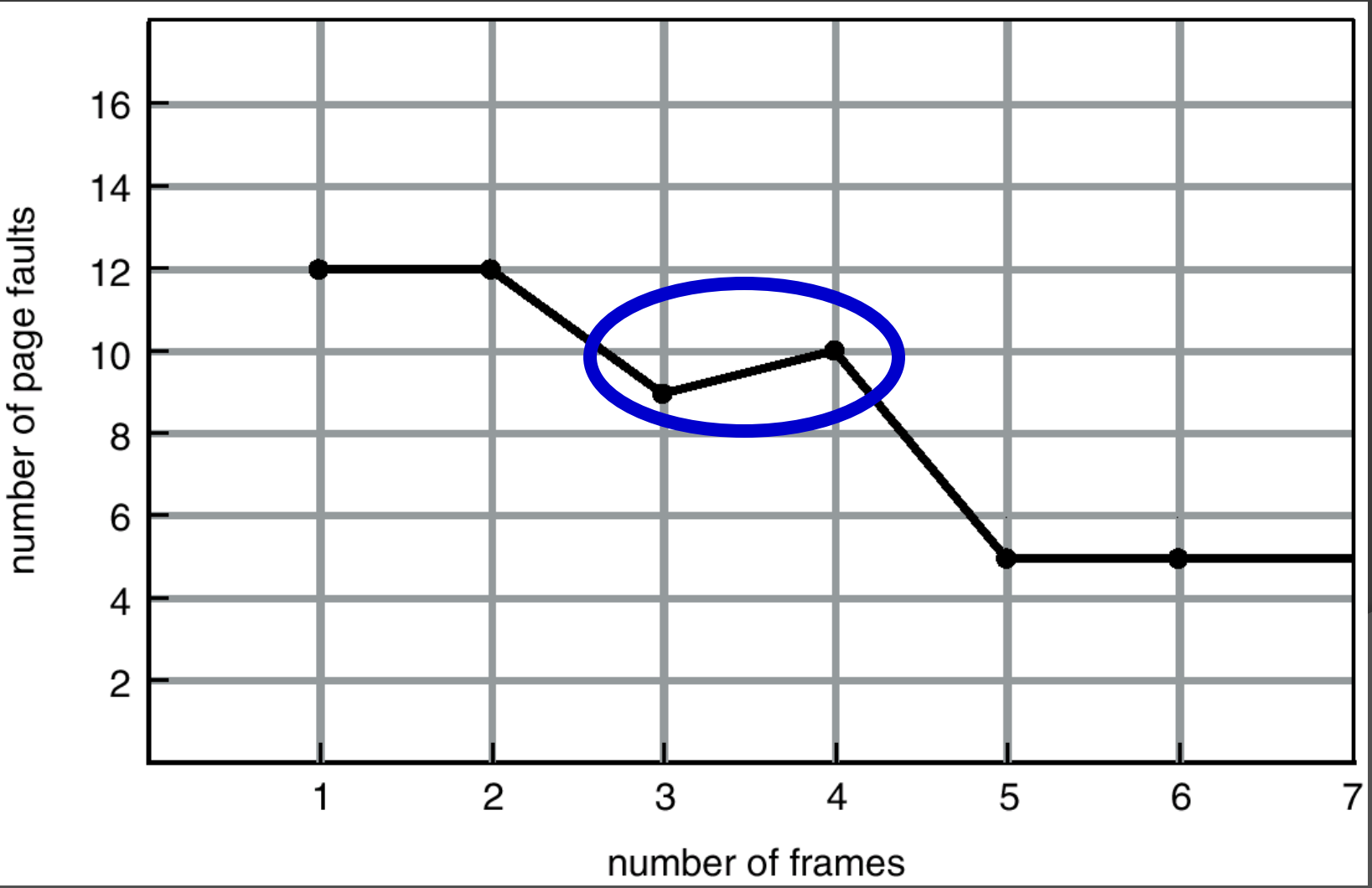
3 frames = 9
4 frames = 10

Belady's anomaly

1
2
3
4
5

PF = 5

FIFO Illustrating Belady's Anomaly

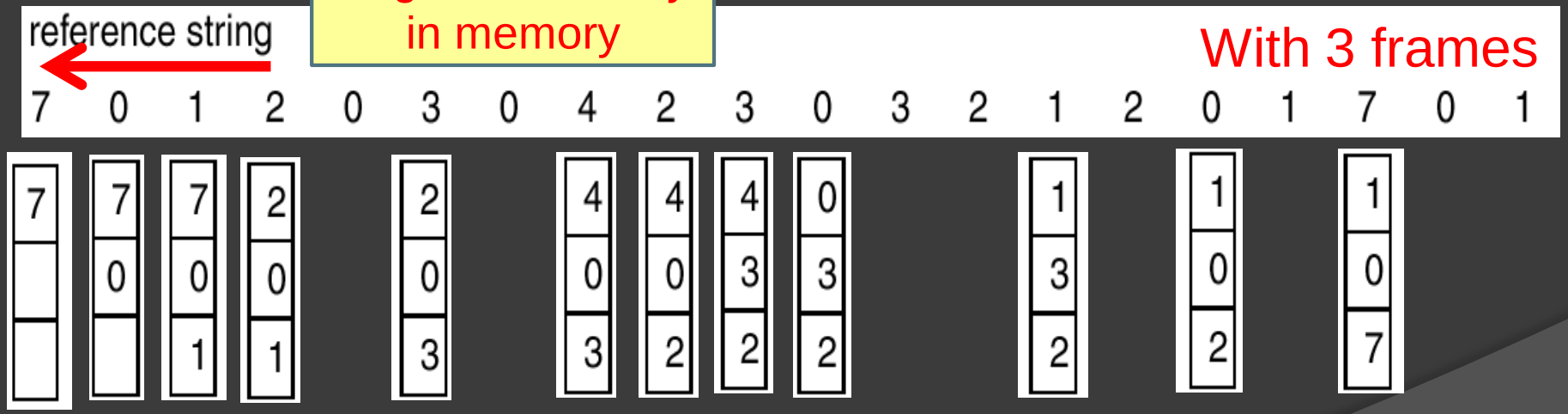


LRU (Least Recently Used) Page Replacement

- Replace a page that has not been used in the most amount of time (use past knowledge).

Page 7 is the least used

Page 0 is already in memory



How to determine which page to replace?

- Associate time of last use with each page

Another LRU algo Example

- Show how LRU algo is implemented with reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

1	4	5	3
2	1	4	
3	2	5	

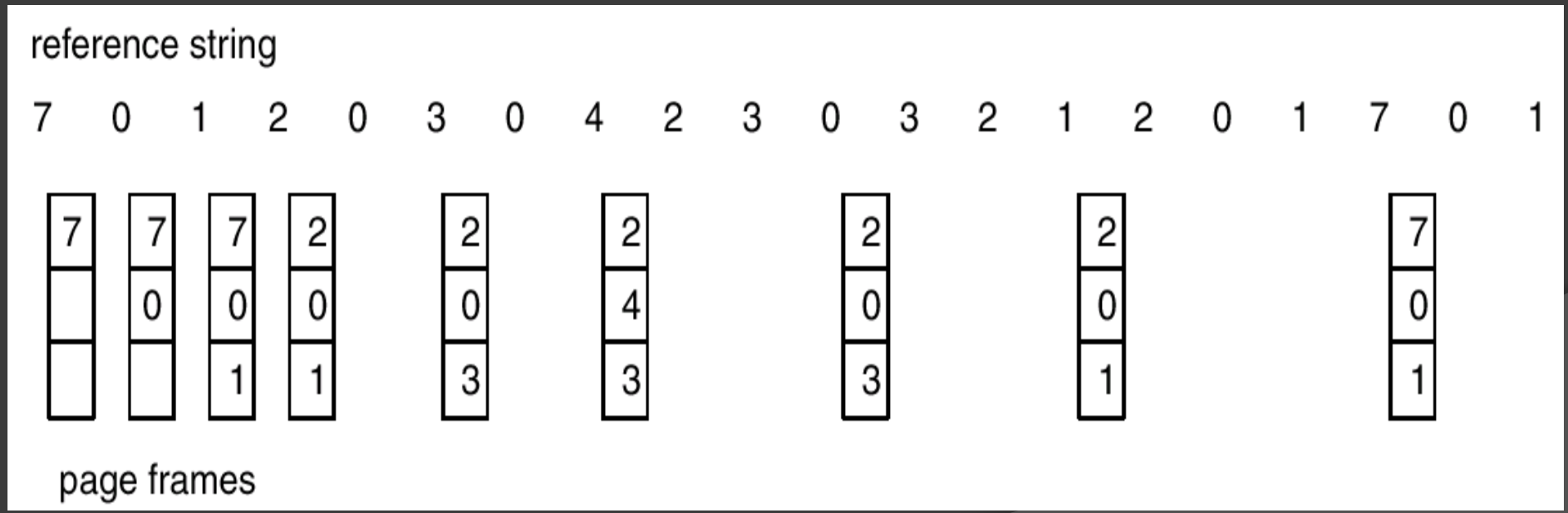
PF = 10

1	5		
2			
3	5	4	
4	3		

PF = 8

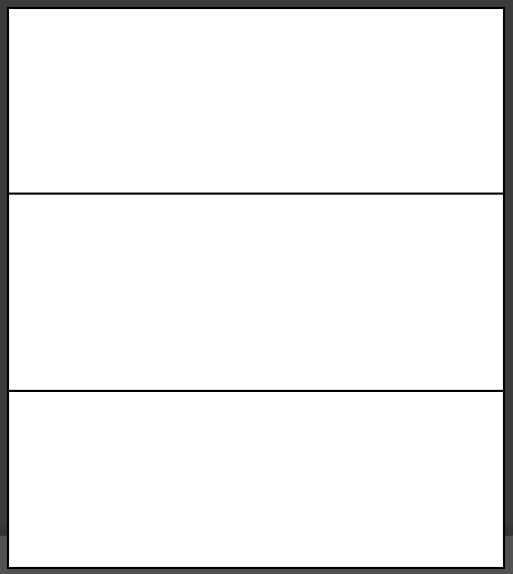
Optimal Page Replacement

Reference String: 7, 0, 1, 2, 0, 3, 0, 0, 4, 2, 3, 0, 3, 2, 1, 2 0, 1, 7, 0, 1

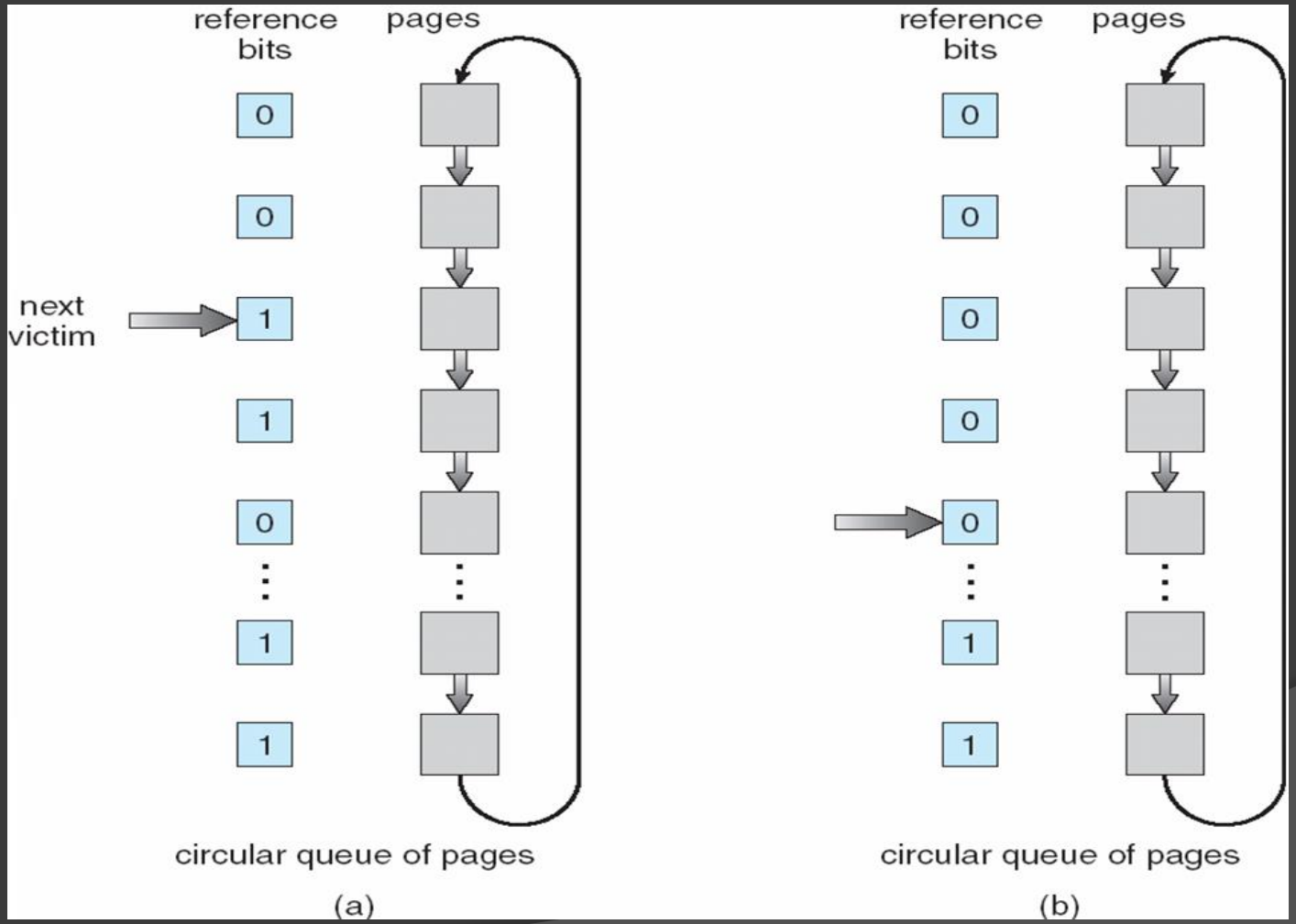


Another Optimal algo Example

- Given the following reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, show how Optimal page replacement algorithm is implemented with the following number of frames.



Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- Algorithms used:
  - **LFU (Least Frequently Used) Algorithm** - replaces page with smallest count.
  - **MFU (Most Frequently Used) 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

$\frac{Si}{S}M$

Fixed Allocation

- **Equal Allocation**
  - Allocate same number of frames to all processes.
  - *Example:* 100 frames with 5 processes, allocate each process with 20 frames.
- **Proportional Allocation**
  - Allocate process with frames proportionate to its size.

Fixed Allocation Example:

- Allocate 64 frames to process's P1 that requires 10 frames and P2 that requires 127 frames

For Equal Allocation:

$$\frac{\text{64 frames}}{\text{2 processes}} = 32$$

- **P1** is allocated to **32** frames
- Also, **P2** is allocated to **32** frames

For Proportional Allocation:

$$P1 = \frac{10}{10+127} x64 \approx 5$$

$$P2 = \frac{127}{10+127} x64 \approx 59$$

- **P1** is allocated to **5** frames
- **P2** is allocated to **59** frames

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

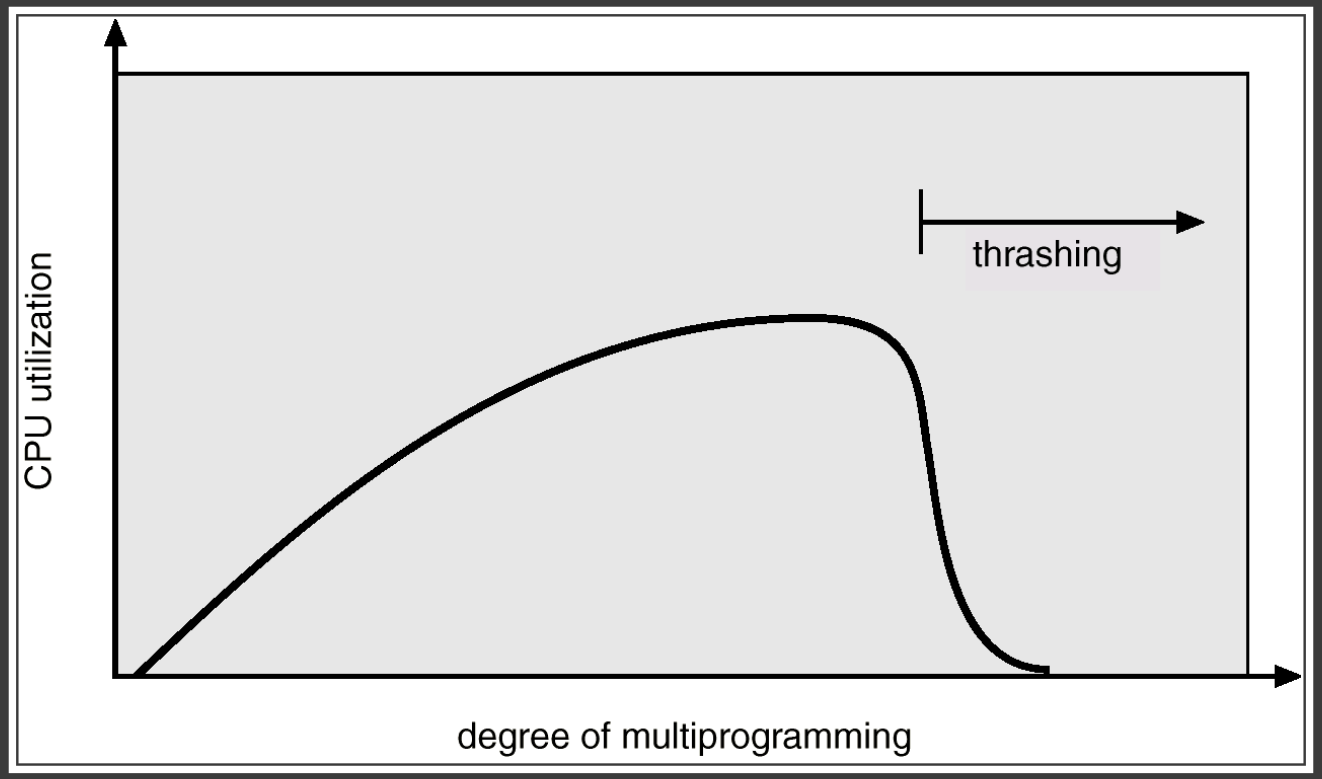
- **Global** replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local** replacement – each process selects from only its own set of allocated frames.

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - OS 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.

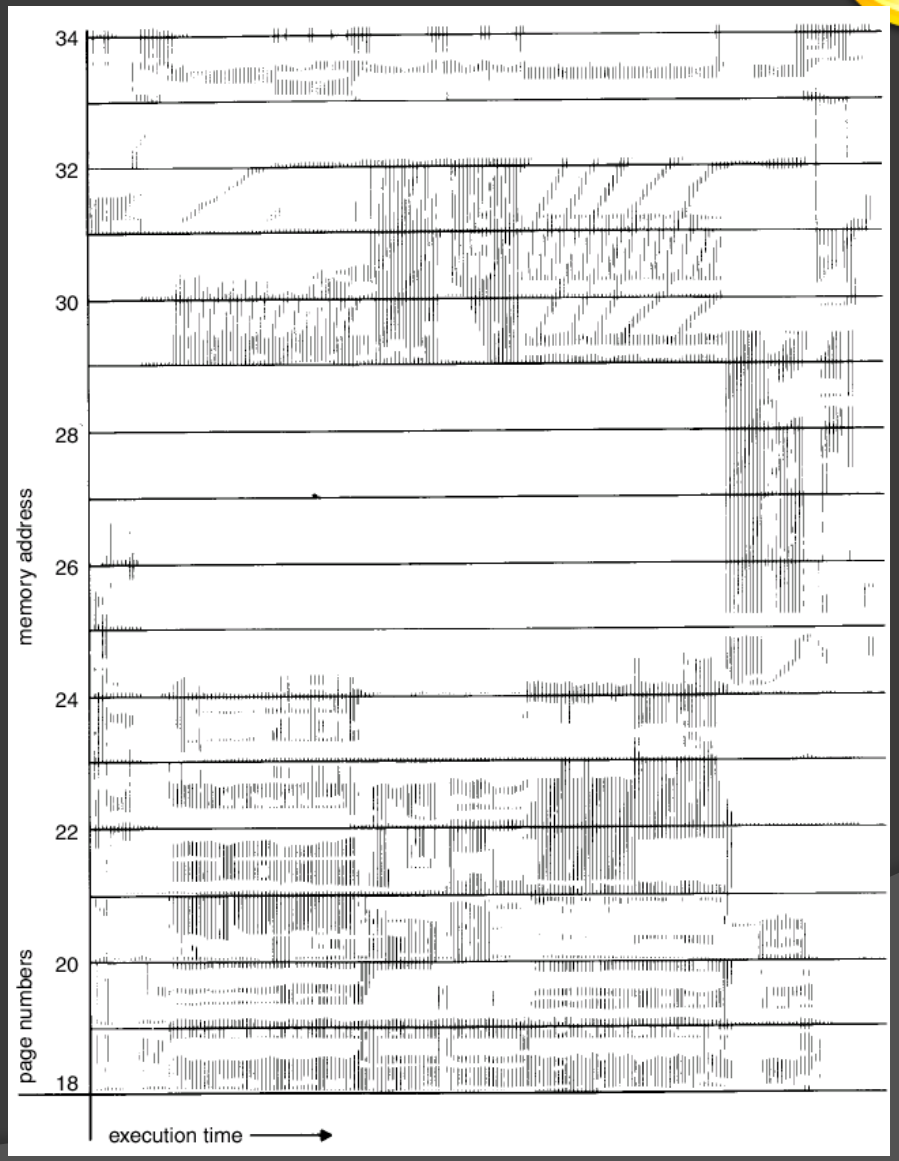
Thrashing (cont.)



Why does thrashing occur?

Σ size of locality > total memory size

Locality In A Memory-Reference Pattern



End of Presentation