

OPERATING SYSTEMS

CS3500

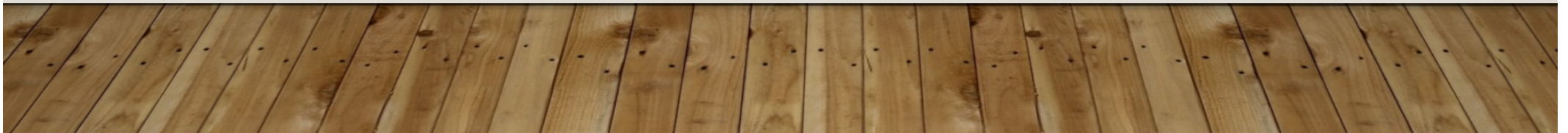
**PROF. SUKHENDU DAS DEPTT. OF COMPUTER SCIENCE
AND ENGG., IIT MADRAS, CHENNAI – 600036.**

Email: sdas@cse.iitm.ac.in

URL: <http://www.cse.iitm.ac.in/~vplab/os.html>

Sept. – 2022.

MAIN MEMORY - I



OUTLINE

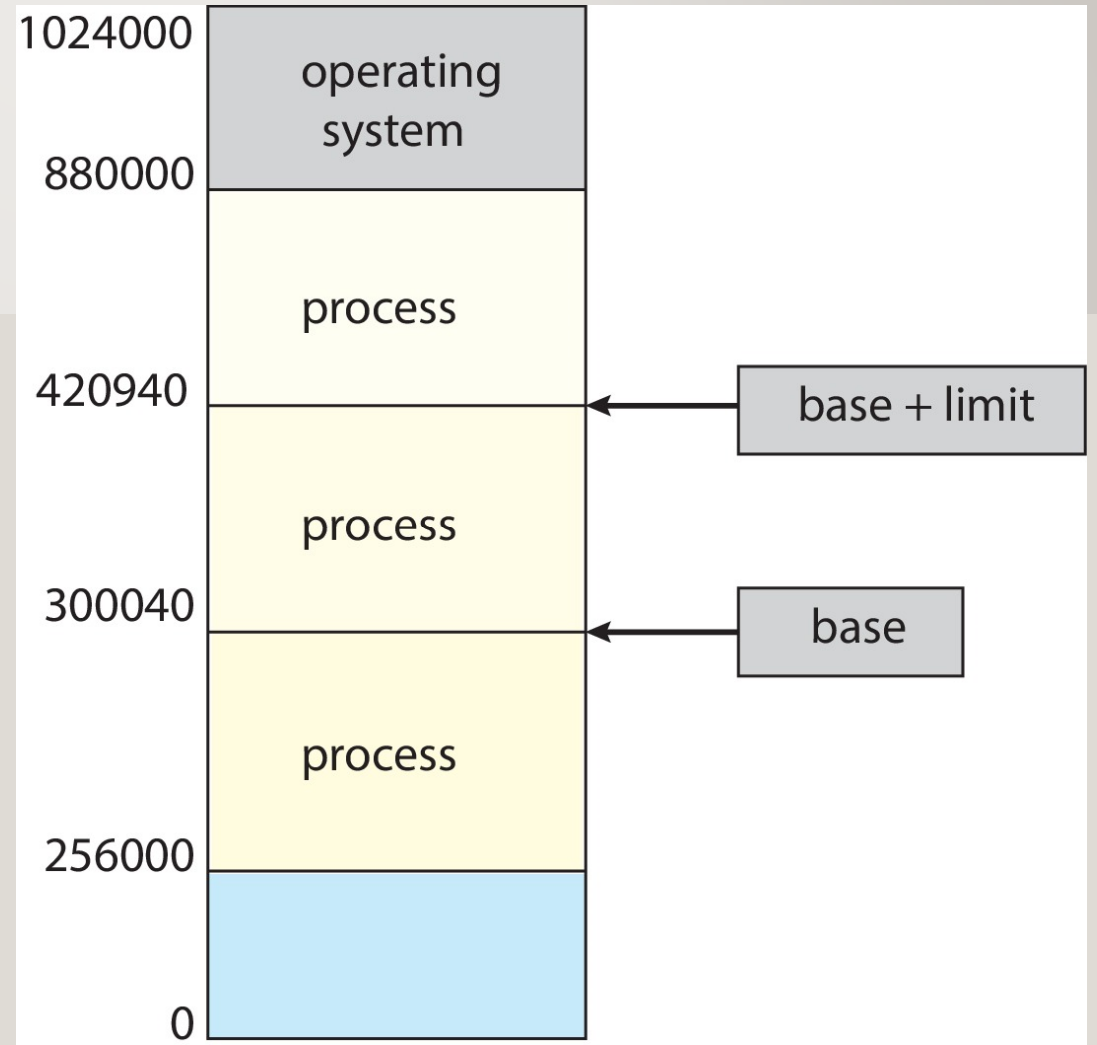
- Background
- Contiguous Memory Allocation
- Paging
- TLB
- Hashed Page Table
- Inverted Page Table

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
- Memory unit only sees a stream of:
 - addresses + read requests, or
 - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

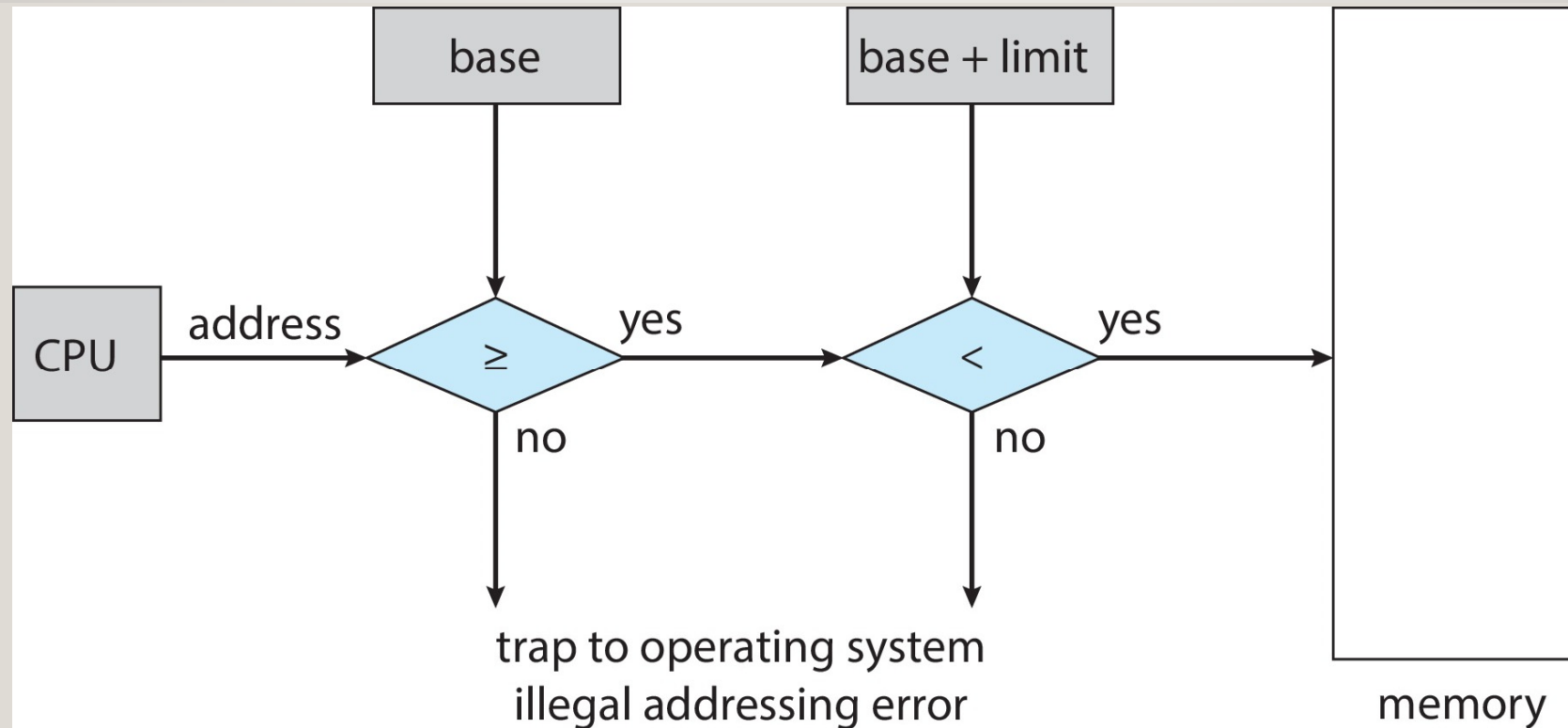
PROTECTION

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



HARDWARE ADDRESS PROTECTION

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- the instructions to loading the base and limit registers are privileged

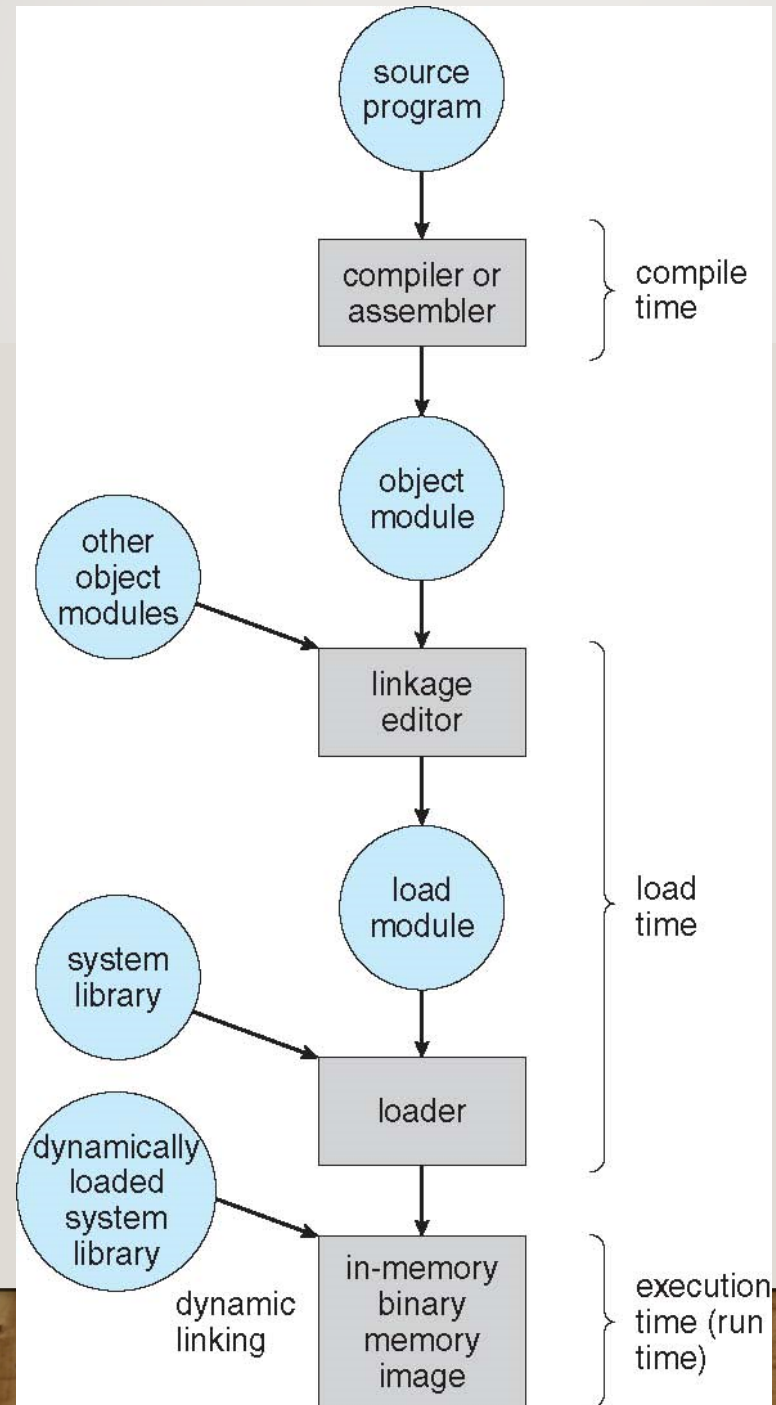
ADDRESS BINDING

- Programs on disk, ready to be brought into memory to execute form an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- Addresses represented in different ways at different stages of a program's life
 - Source code addresses usually symbolic
 - Compiled code addresses **bind** to relocatable addresses
 - i.e., “14 bytes from beginning of this module”
 - Linker or loader will bind relocatable addresses to absolute addresses
 - i.e., 74014
 - Each binding maps one address space to another

BINDING OF INSTRUCTIONS AND DATA TO MEMORY

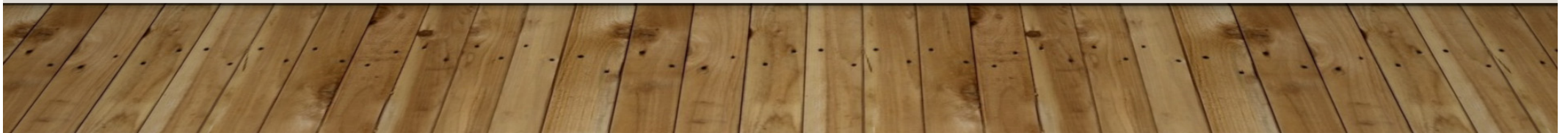
- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory 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



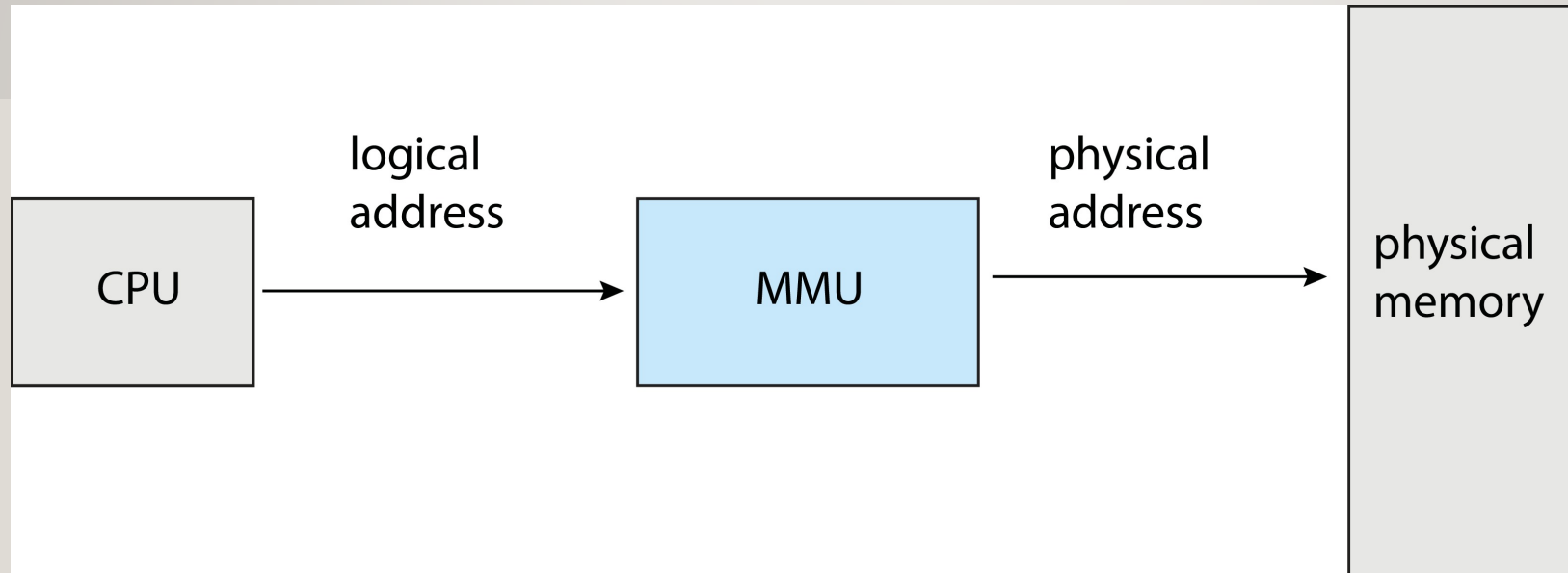
LOGICAL VS. PHYSICAL ADDRESS SPACE

- 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
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program



MEMORY-MANAGEMENT UNIT (MMU)

- Hardware device that at run time maps virtual to physical address



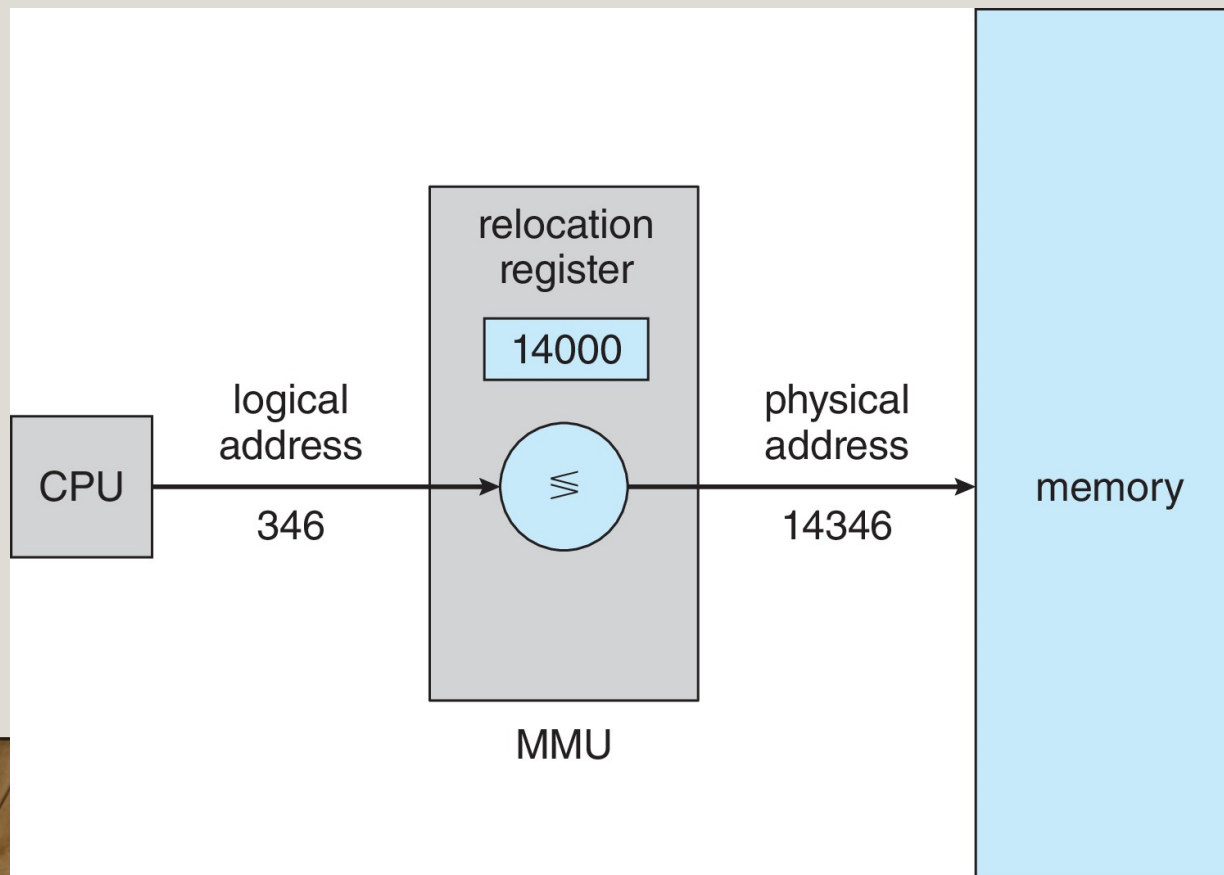
- Many methods possible, covered in the upcoming slides.

MEMORY-MANAGEMENT UNIT (CONT.)

- Consider simple scheme. which is a generalization of the base-register scheme.
 - The base register now called **relocation register**
- 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
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses

MEMORY-MANAGEMENT UNIT (CONT.)

- Consider simple scheme. which is a generalization of the base-register scheme.
 - The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



DYNAMIC LOADING

- The entire program does not need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading

DYNAMIC LINKING

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- 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 checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
 - Versioning may be needed

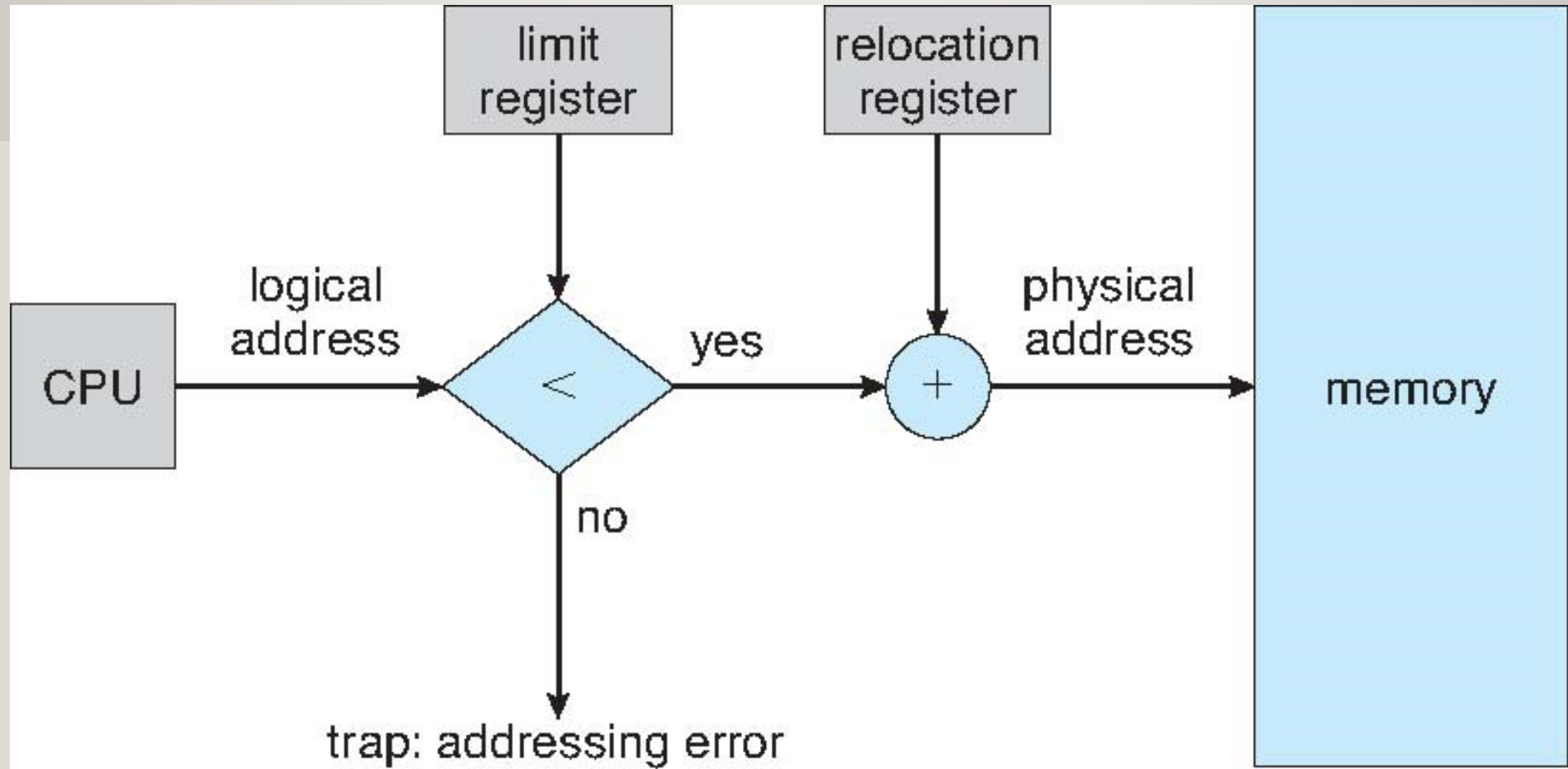
CONTIGUOUS ALLOCATION

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory

CONTIGUOUS ALLOCATION (CONT.)

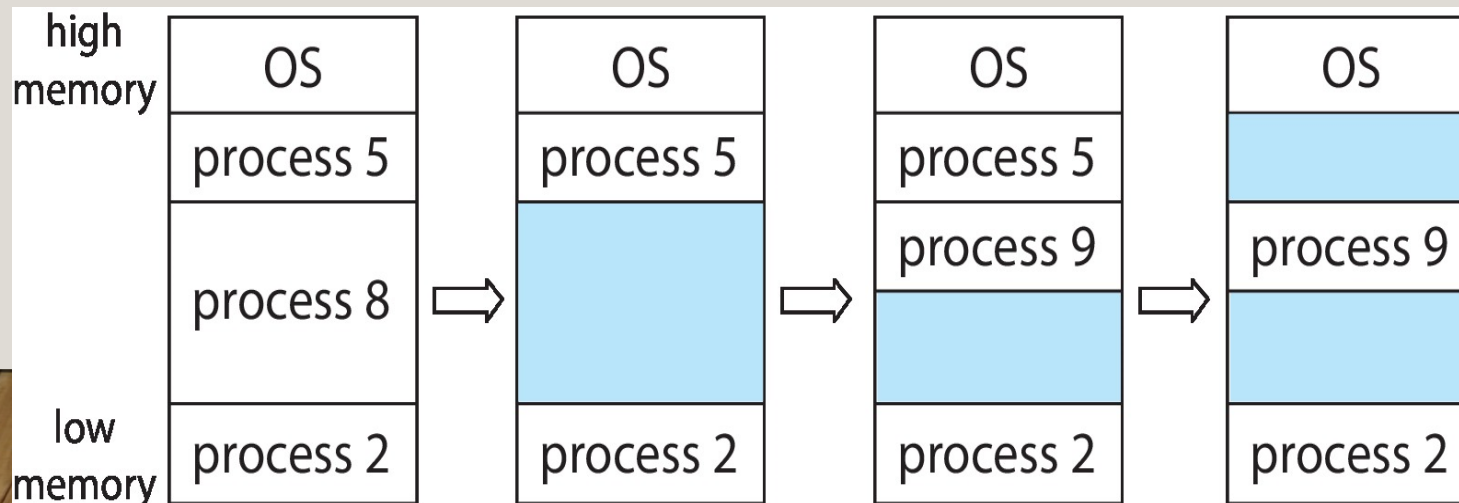
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size (eg. case of rarely used device drivers)

HARDWARE SUPPORT FOR RELOCATION AND LIMIT REGISTERS



VARIABLE PARTITION

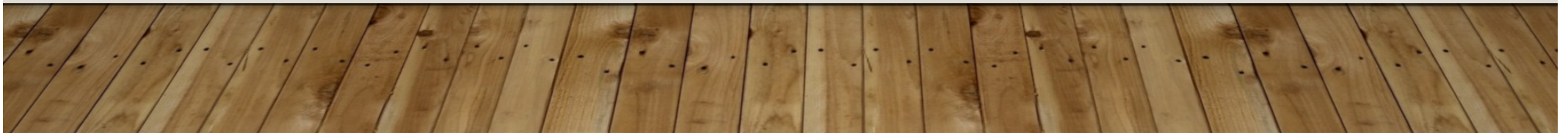
- Multiple-partition allocation
 - Degree of multiprogramming limited by number of partitions
 - **Variable-partition** sizes for efficiency (sized to a given process' needs)
 - **Hole** – block of available memory; holes of various size are scattered throughout memory
 - When a process arrives, it is allocated memory from a hole large enough to accommodate it
 - Process exiting frees its partition, adjacent free partitions combined
 - 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

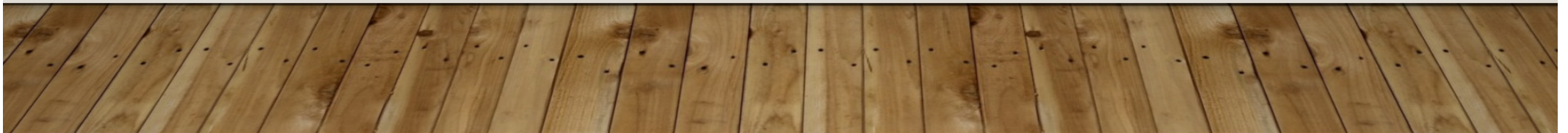


FRAGMENTATION

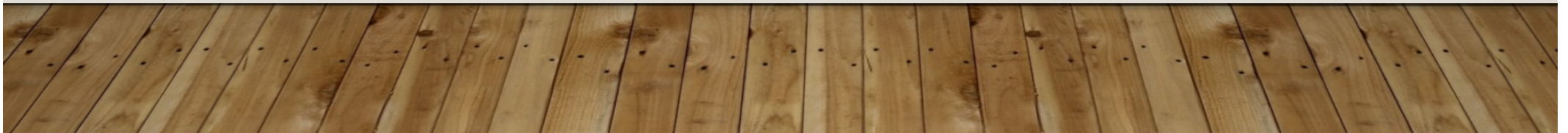
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation
 - $1/3$ may be unusable -> **50-percent rule**

For every N blocks allocated, $0.5N$ blocks are lost to fragmentation.

Unusable memory = $(0.5N)/(N+0.5N) = 1/3$

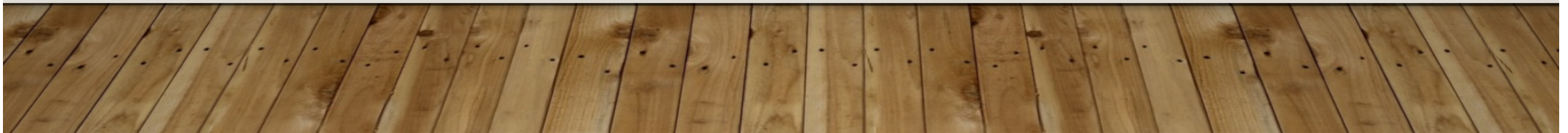


- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems



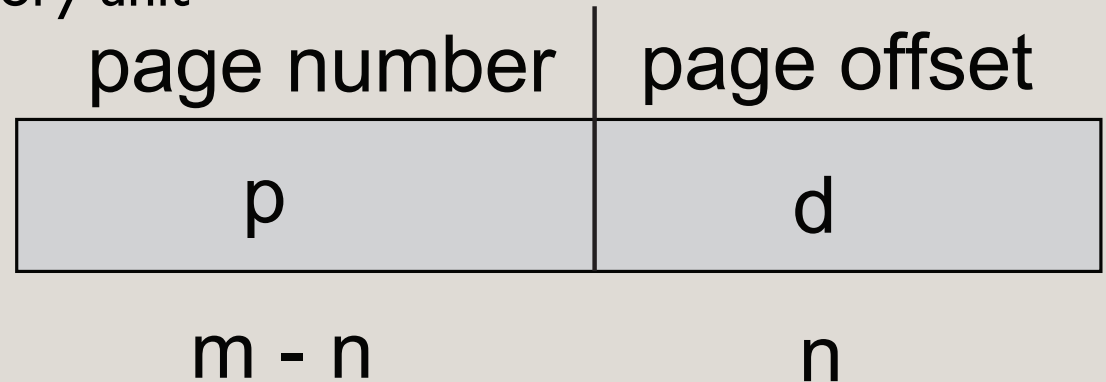
PAGING

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- 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
- Backing store likewise split into pages
- Still have Internal fragmentation



ADDRESS TRANSLATION SCHEME

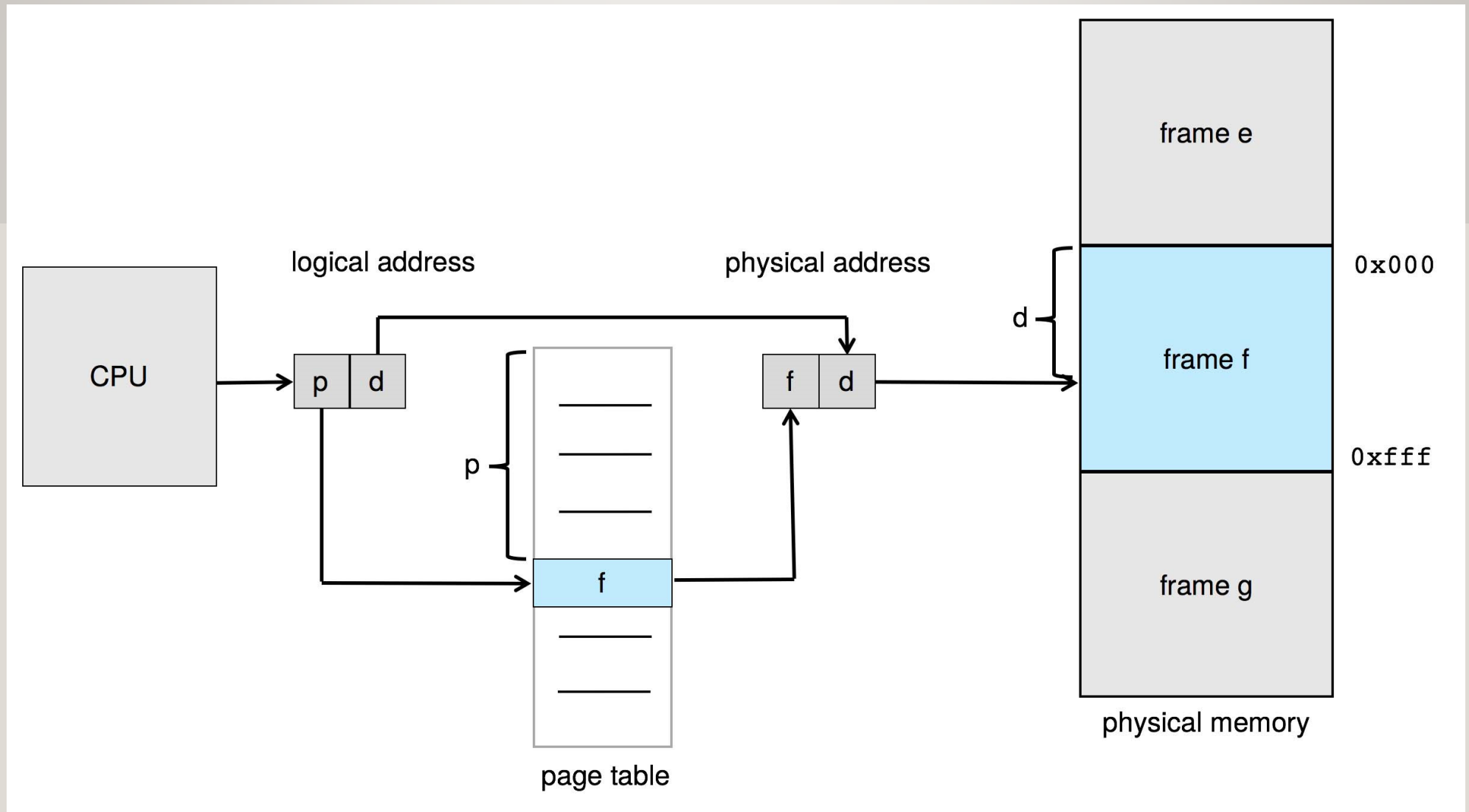
- Address generated by CPU is divided into:
 - **Page number** (p) – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset** (d) – combined with base address to define the physical memory address that is sent to the memory unit



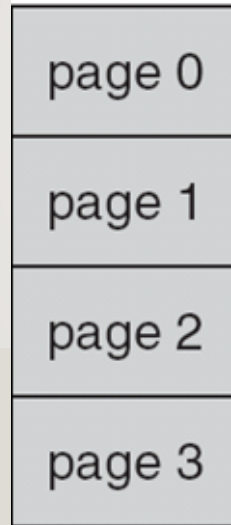
- For given logical address space 2^m and page size 2^n

When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the **backing store**). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames

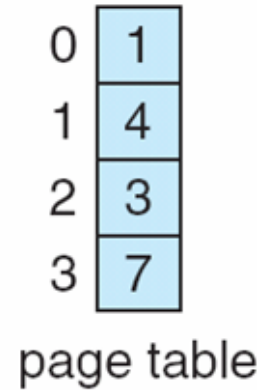
PAGING HARDWARE



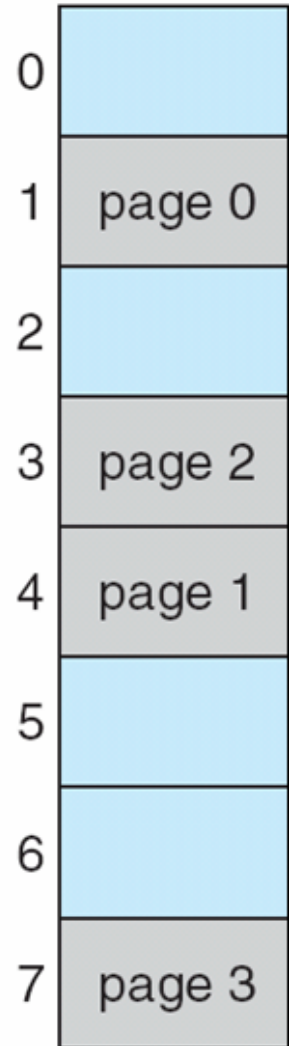
PAGING MODEL OF LOGICAL AND PHYSICAL MEMORY



logical memory



frame number



physical memory

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f from the page table.
3. Replace the page number p in the logical address with the frame number f .

As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

The page size (= the frame size)

PAGING EXAMPLE

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

- Logical address:
- $n = 2$ and $m = 4$.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages/frames)

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Logical address 0 maps to physical address 20 [= (5 × 4) + 0].

Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].

Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].

Logical address 13 maps to physical address 



PAGING -- CALCULATING INTERNAL FRAGMENTATION

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- Results in ?? → pages + bytes
- Internal fragmentation of $2,048 - 1,086 = 962$ bytes
- Worst case fragmentation = N frames + 1 byte = $(N+1)$ frames
- On average fragmentation = $(1 / 2)$ frame size

- So small frame sizes desirable?
- But each page table entry takes memory to track; more disk I/O involved
- Page sizes growing over time; I/O B/W better for larger transfer/access
 - Solaris supports two page sizes – 8 KB and 4 MB; see “*huge pages*”

On a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well.

A 32-bit entry can point to one of 2^{32} physical page frames.

If the frame size is 4 KB (2^{12}), then a system with 4-byte entries can address  bytes (or  TB) of physical memory.

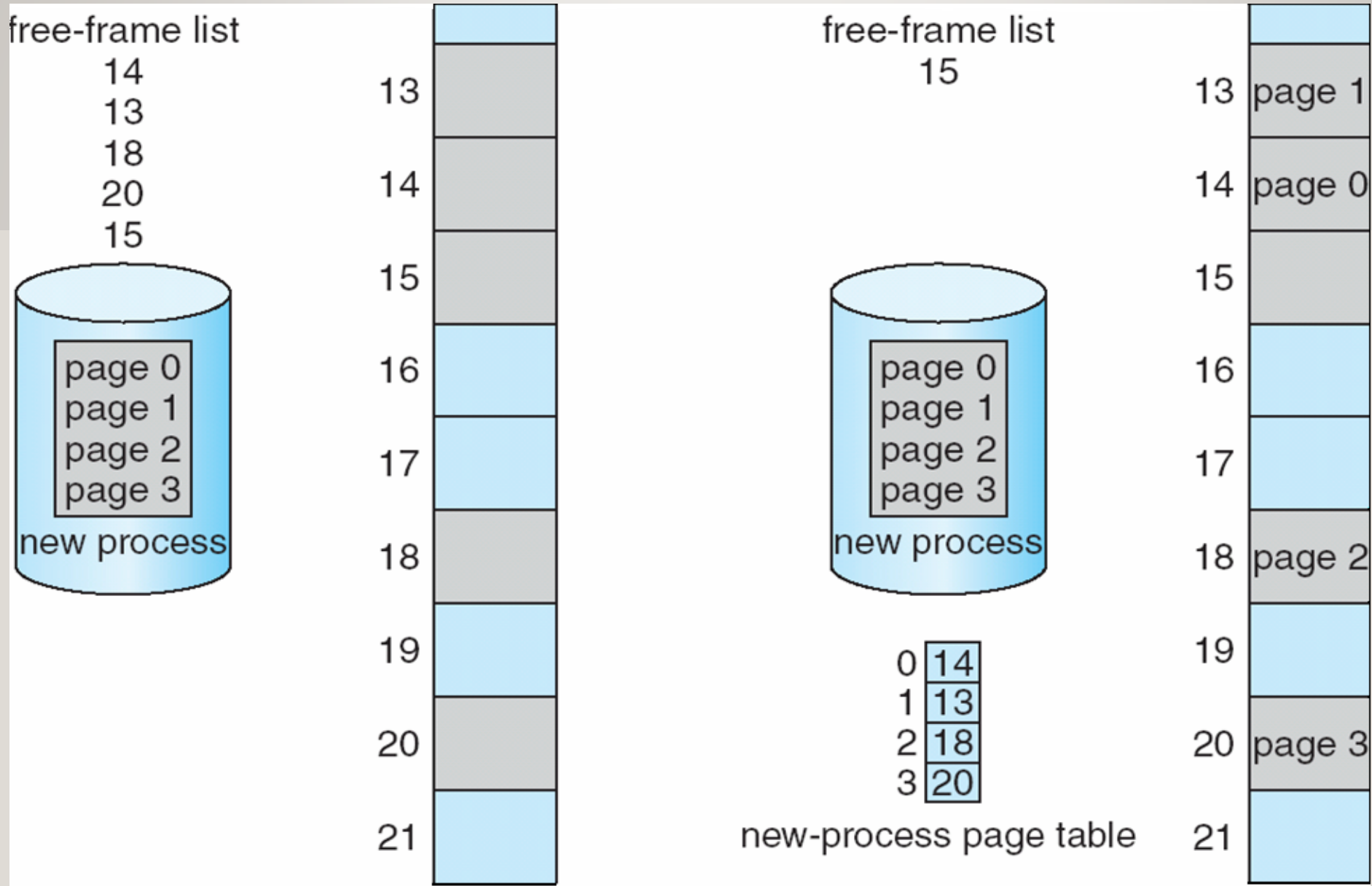
Note here that the size of physical memory in a paged memory system is typically different from the maximum logical size of a process.

Other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum.

In next slide –

clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory.

FREE FRAMES – IN FRAME TABLE

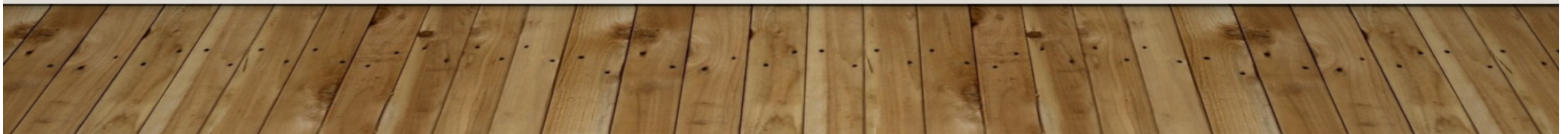


(a) Before allocation

(b) After allocation

IMPLEMENTATION OF PAGE TABLE

- If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a **copy of the page table for each process**, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually.
 - It is also **used by the CPU dispatcher to define the hardware page table** when a process is to be allocated the CPU. **Paging therefore increases the context-switch time.**
- Page table is kept in main memory
 - **Page-table base register (PTBR)** points to the page table
 - **Page-table length register (PTLR)** 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 **translation look-aside buffers (TLBs)** (also called **associative memory**).



TRANSLATION LOOK-ASIDE BUFFER

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
 - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered
 - Some entries can be **wired down** for permanent fast access

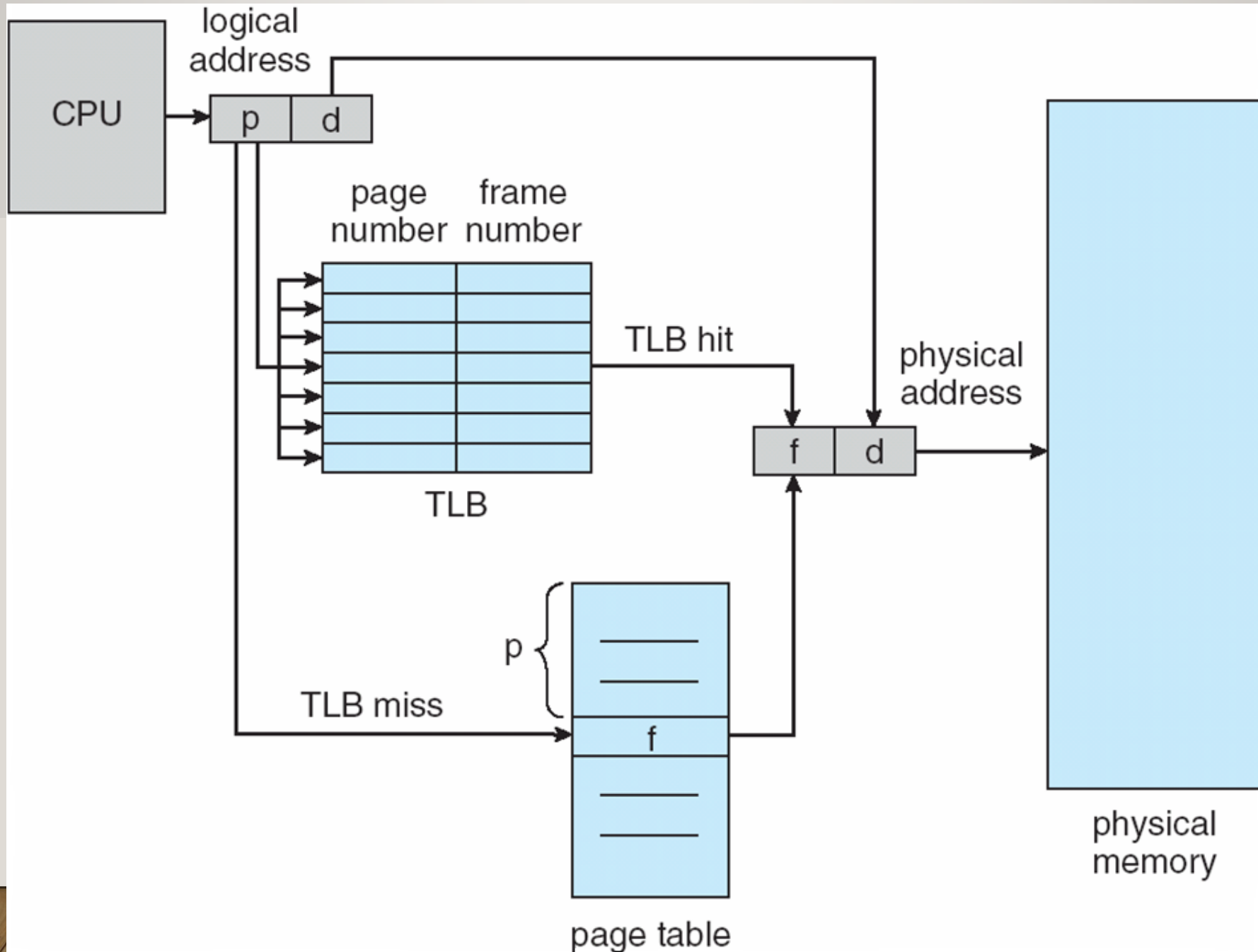
HARDWARE

- Associative memory – parallel search

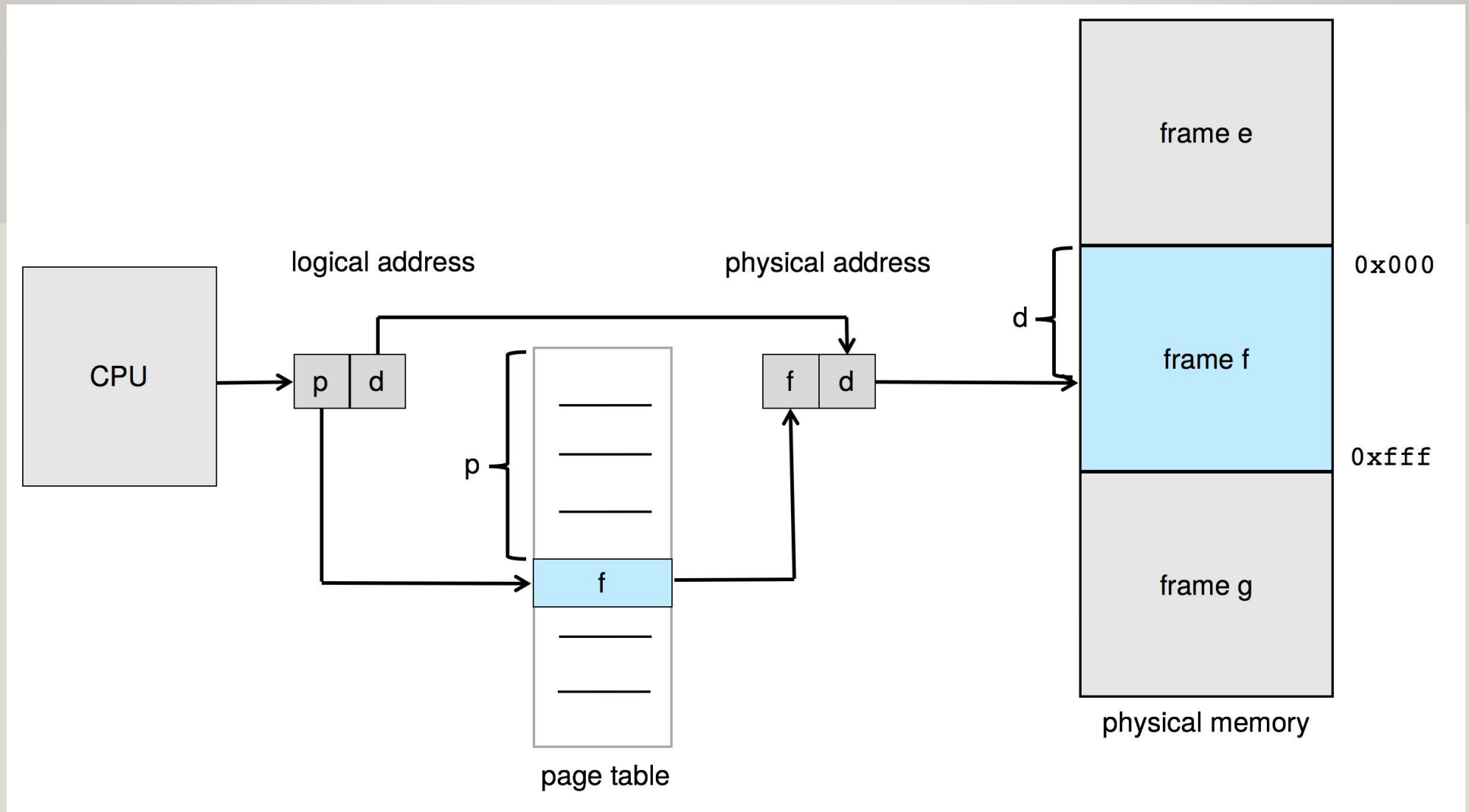
Page #	Frame #

- 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



PAGING HARDWARE (RELOOK W/O TLB)



EFFECTIVE ACCESS TIME

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If we find the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time (EAT)**

$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying 20% slowdown in access time

- Consider amore realistic hit ratio of 99%,
$$\text{EAT} = 0.99 \times 10 + 0.01 \times 20 = 10.1 \text{ ns}$$

implying only 1% slowdown in access time.

MEMORY PROTECTION

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **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
 - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel

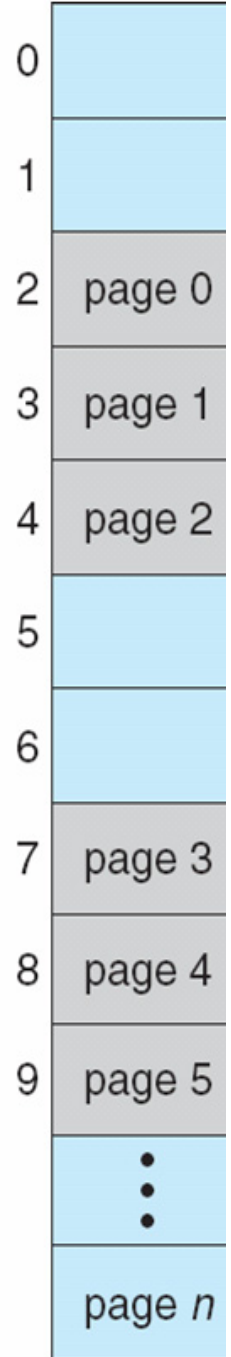
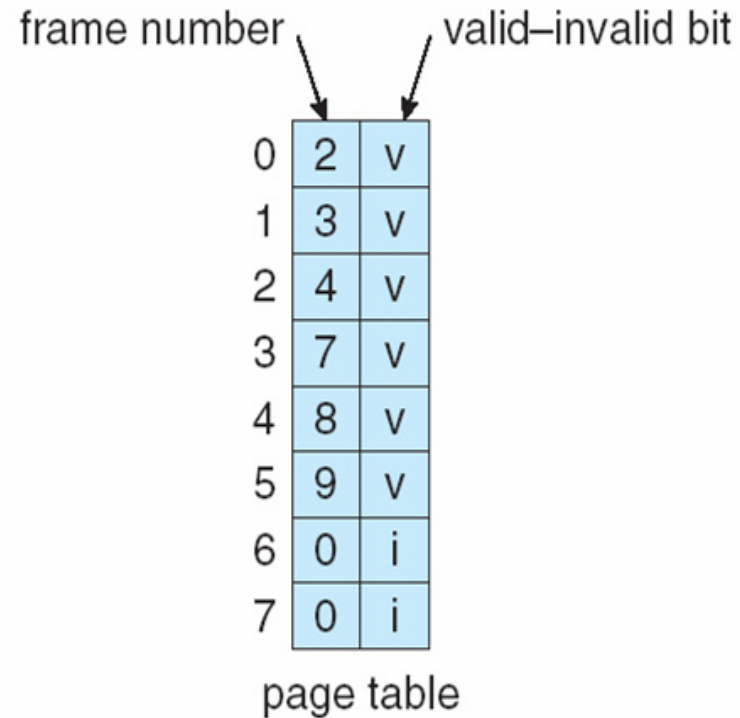
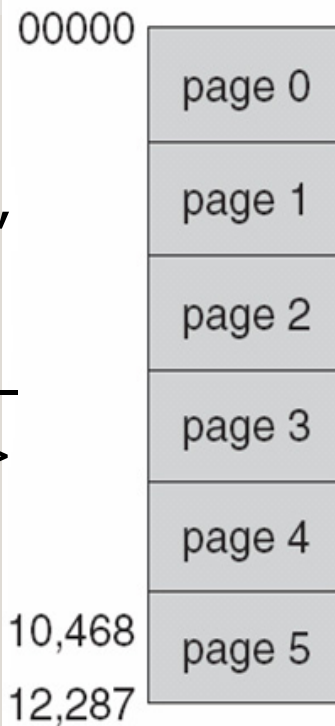
VALID (V) OR INVALID (I) BIT IN A PAGE TABLE

Use a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Page size 2 KB;

Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table – Sure ?? Why 12287 ? ->

Accesses to addresses up to 12287 are valid.

Only the addresses from 12288 to 16383 are invalid, but PT says... ☹



Internal FRAG.. issue – solution using **PTLR**

SHARED PAGES

- **Shared code**

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

- **Private code and data**

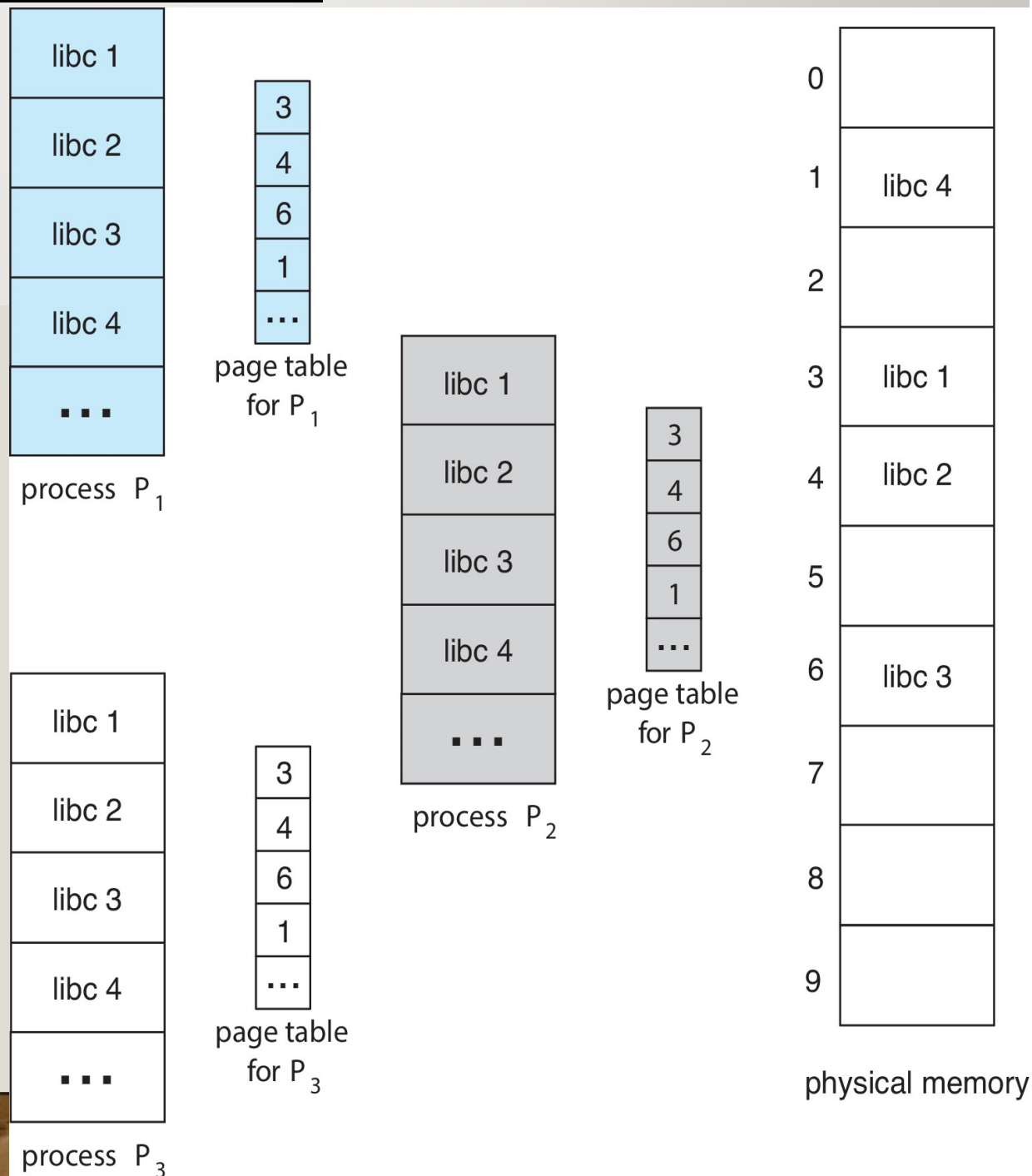
- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

SHARED PAGES EXAMPLE


Thus, to support 40 processes, we need only one copy of the library, and the total space now required is 2 MB instead of 80 MB—a significant saving.

Reentrant code is non-self-modifying code: it never changes during execution.

Thus, two or more processes can execute the same code at the same time.

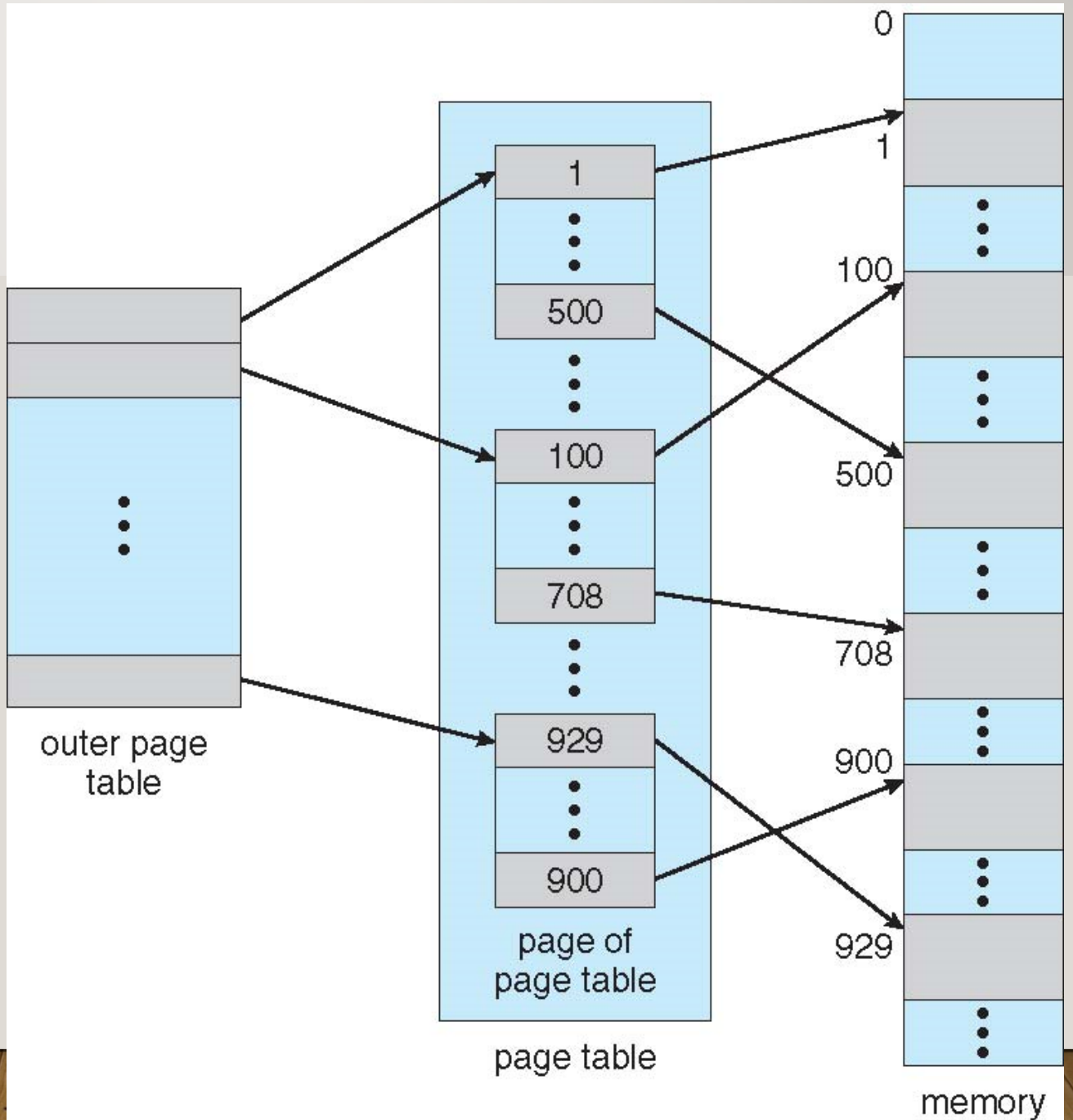


STRUCTURE OF THE PAGE TABLE

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have #entries ?? - > 
 - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
 - Don't want to allocate that contiguously in main memory
 - One simple solution is to divide the page table into smaller units
 - Hierarchical Paging
 - Hashed Page Tables
 - Inverted Page Tables

HIERARCHICAL PAGETABLES

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



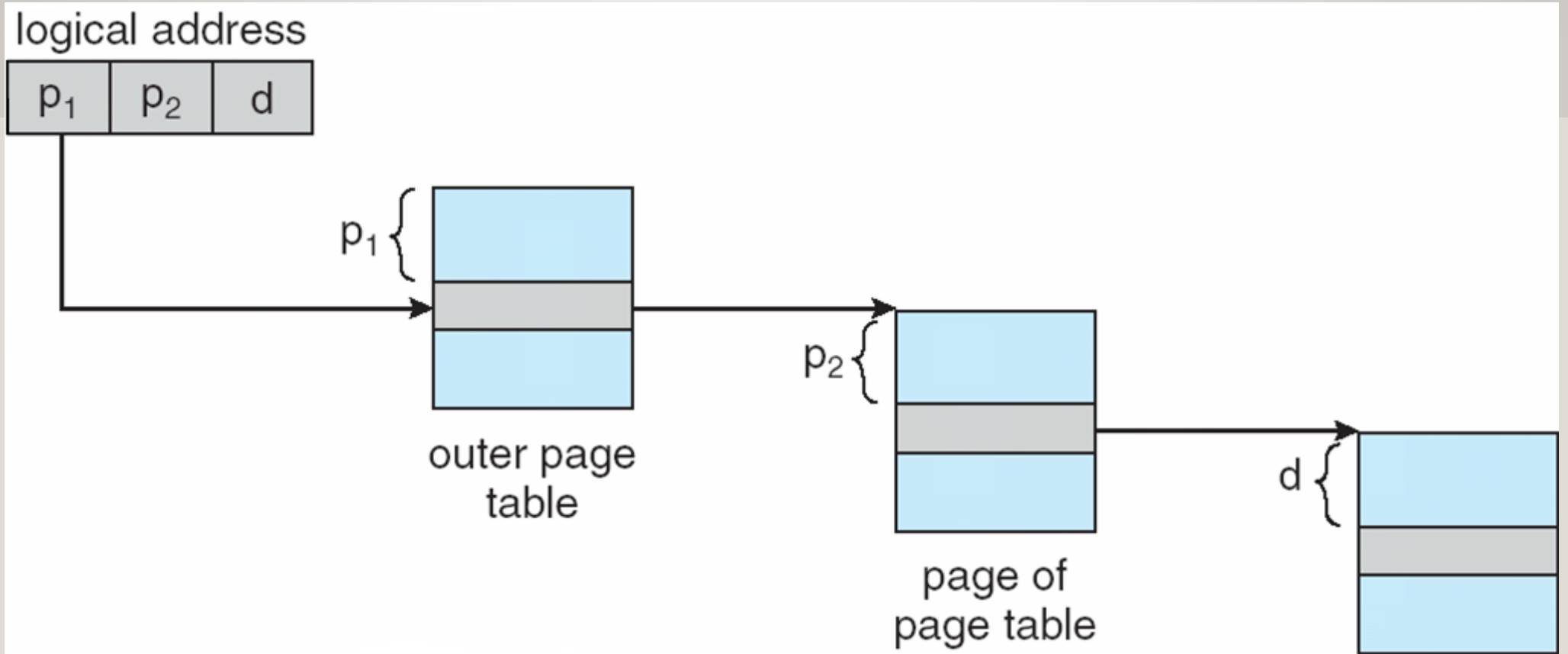
TWO-LEVEL PAGING EXAMPLE

- 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
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

- where p_1 is an **index** into the outer page table, and p_2 is the **displacement** within the page of the inner page table
- Known as **forward-mapped page table**

ADDRESS-TRANSLATION SCHEME



64-BIT LOGICAL ADDRESS SPACE

- Even two-level paging scheme not sufficient (say, 64-bit LA space)
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like

outer page	inner page	offset
p_1	p_2	d
42	10	12

- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - And possibly 4 memory access to get to one physical memory location

THREE-LEVEL PAGING SCHEME

outer page	inner page	offset
p_1	p_2	d
42	10	12

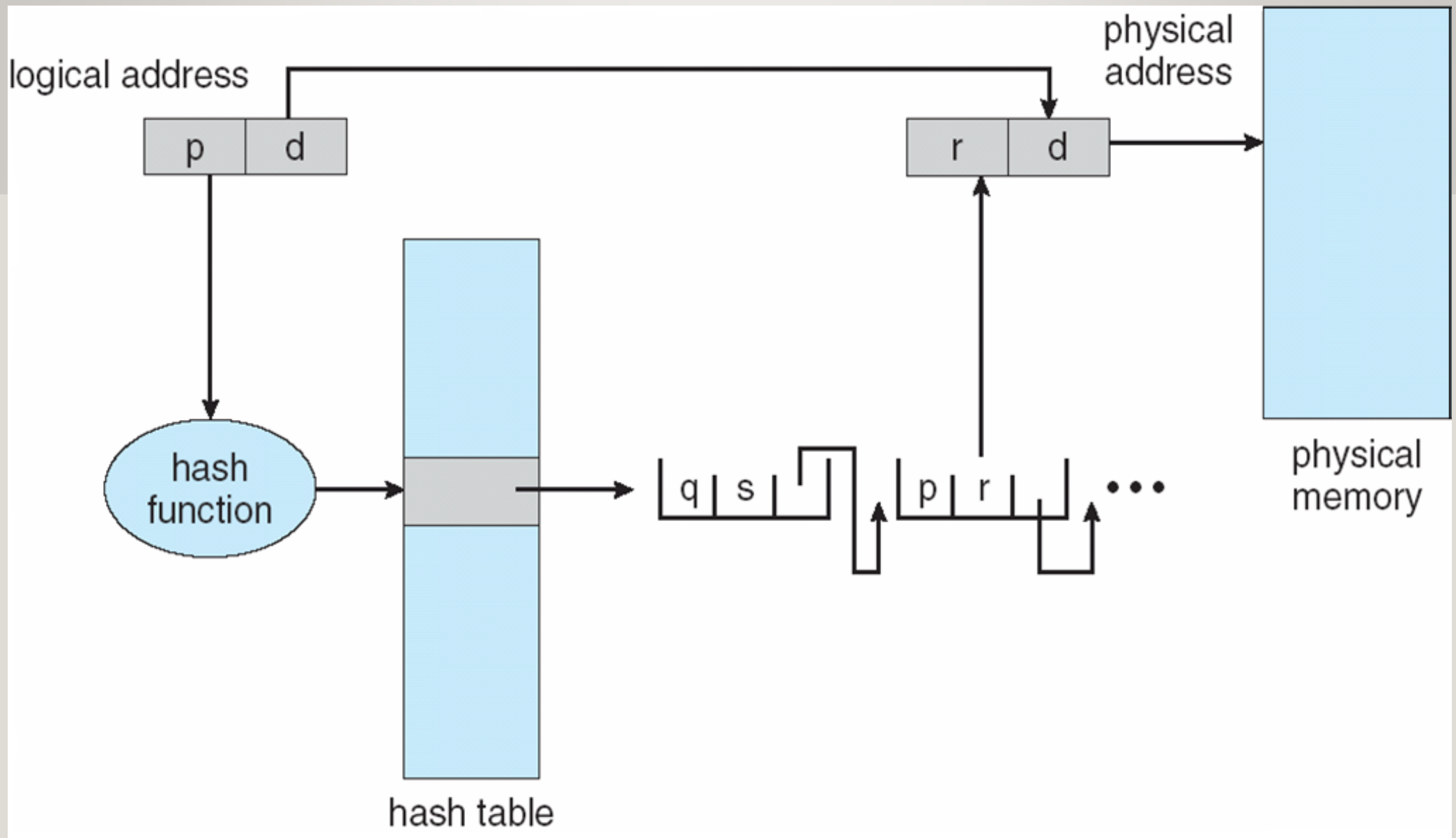
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

The outer page table is still () in size.

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
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain (linked list) searching for a match
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than *1*
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

HASHED PAGE TABLE



INVERTED PAGE TABLE

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry of page table for each real page (frame) of physical memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process (ID) 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
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address

Pages

Frames

0

X

1

X

2

F1

3

F3

4

F6

5

X

6

F5

Page Table of P1

Pages

Frames

0

F2

1

F4

2

F7

3

X

4

X

5

X

6

F0

Page Table of P2

Pages

Frames

0

OS

1

2

3

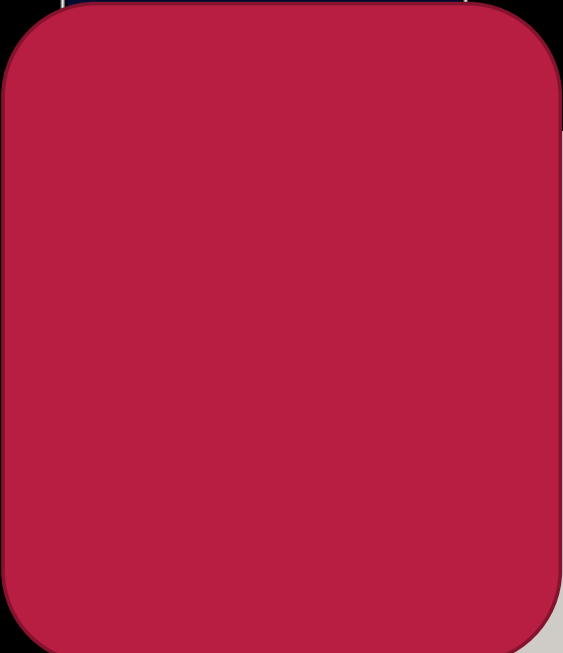
4

5

6

7

Inverted Page Table



INVERTED PAGE TABLE ARCHITECTURE

