Module 5
Memory Management

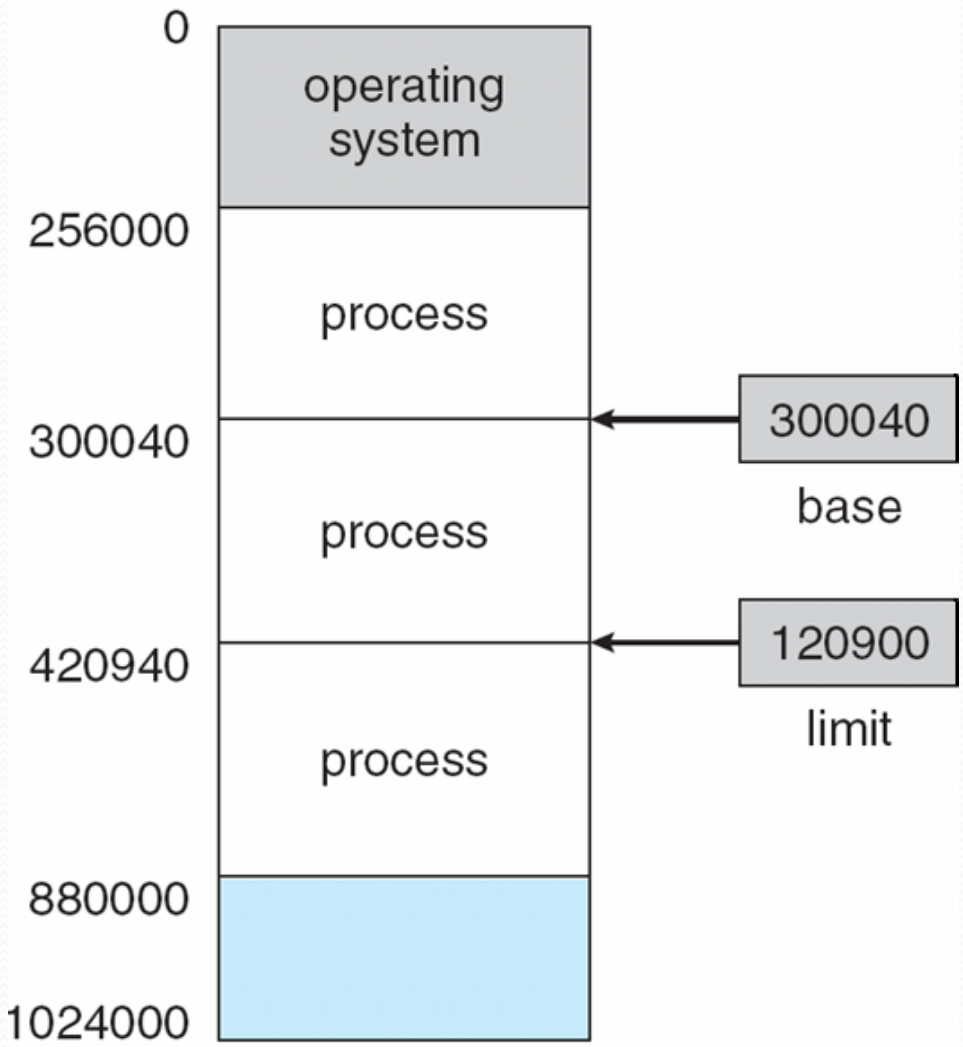
Background

- Memory - a large array of bytes, each with its own address
- The CPU fetches instructions from memory according to the value of the program counter
- The instruction is then decoded and may cause operands to be fetched from memory
- After the instruction has been executed on the operands, results may be stored back in memory
- Program must be brought (from disk) into memory before the CPU can operate on them
- Main memory and registers are only storage CPU can access directly
- **Cache** sits between main memory and CPU
- Protection of memory is required to ensure correct operation

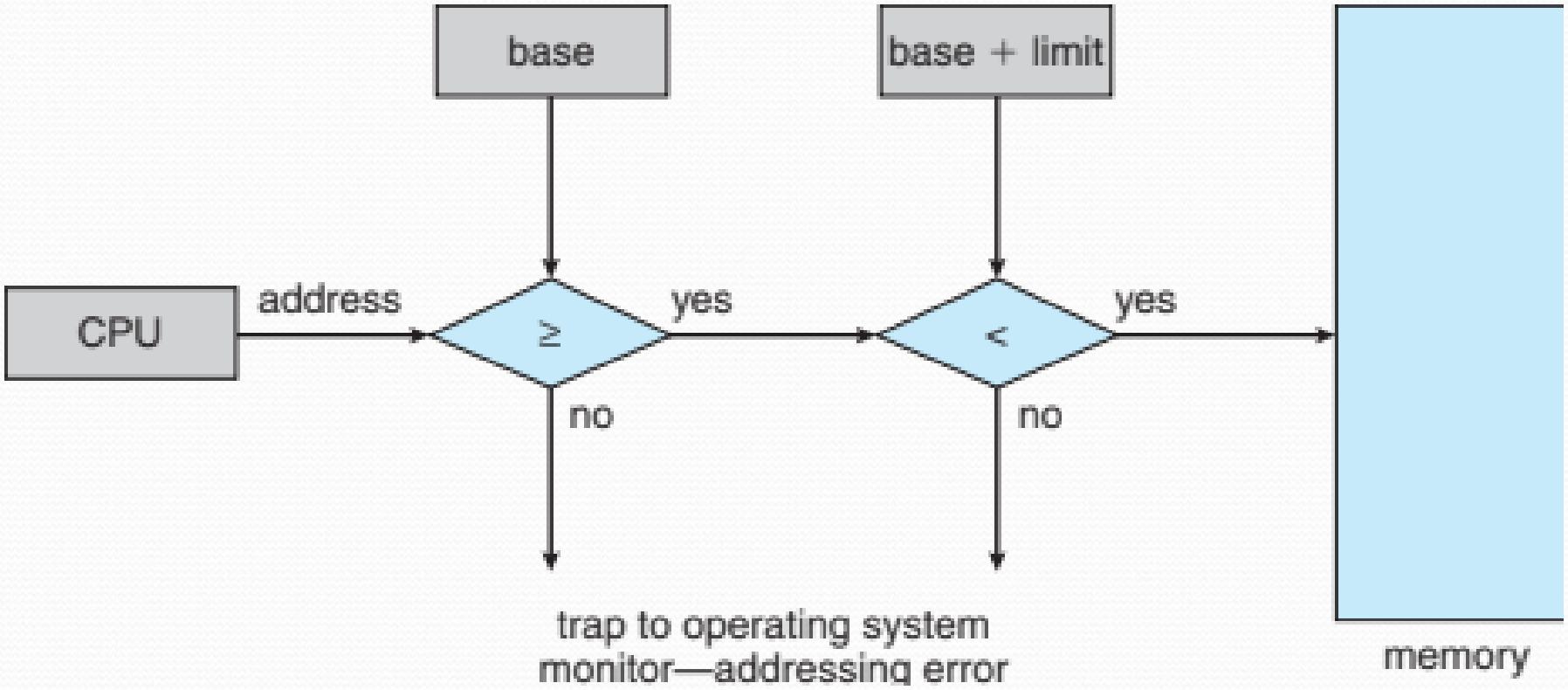
Base and Limit Registers

- Each process has a separate memory space – for concurrent execution
- A pair of **base** and **limit registers** define the logical address space
- **Base register** holds the smallest legal physical memory address
- **Limit register** specifies the size of the range
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction

Base and Limit Registers



Hardware Address Protection

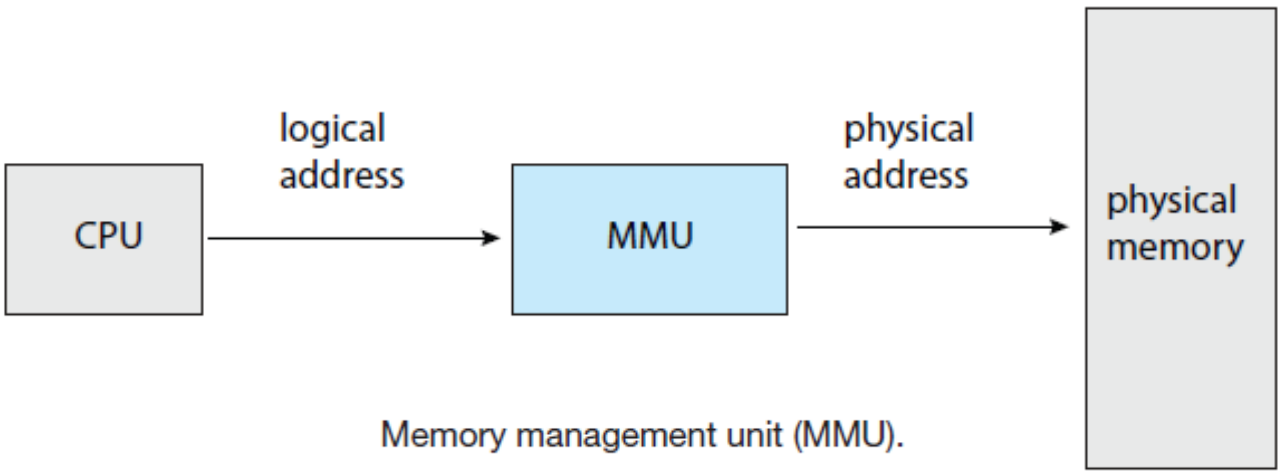


Address Binding

- Usually, a program resides on a disk as a binary executable file
- Programs on disk, ready to be brought into memory to execute form an **input queue**
- A user program goes through several steps
- Addresses may be represented in different ways during these steps
  - Code addresses are 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

Logical vs. Physical Address Space

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

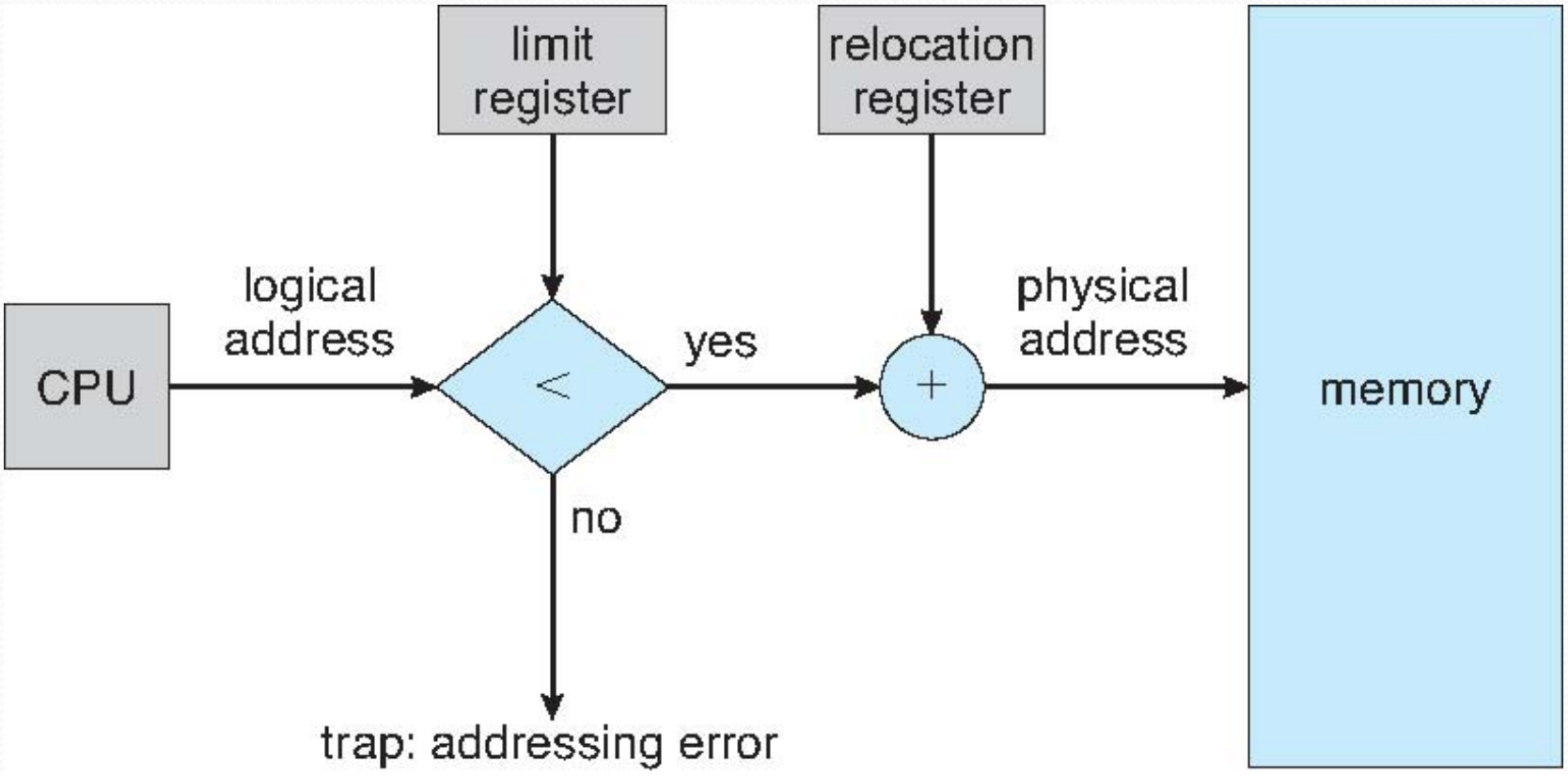
Contiguous Memory Allocation

- Main memory must support both OS and user processes
- Main memory is usually divided into two partitions:
  - Resident operating system, held either in low or high memory addresses
  - User processes
- In contiguous memory allocation, each process is contained in a single section of memory

Contiguous Allocation (Cont.)

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

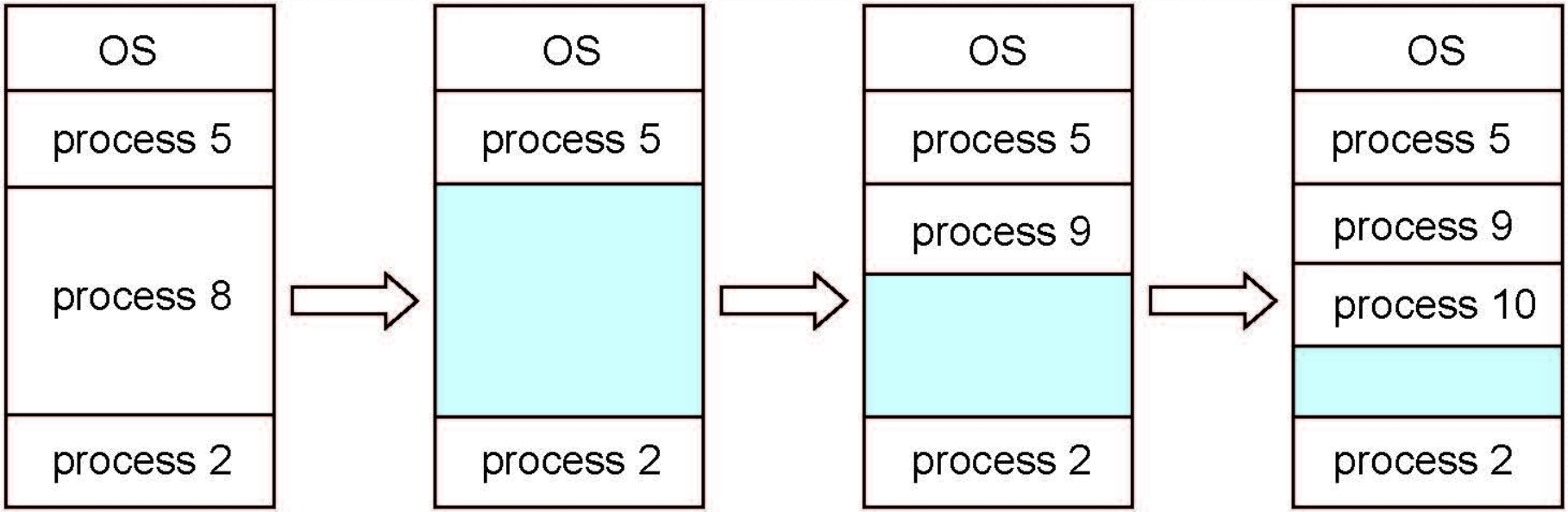
Hardware Support for Relocation and Limit Registers



Memory allocation

- **Variable-partition** - 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)

Multiple-partition allocation



Dynamic Storage-Allocation Problem

It concerns 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

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

Solutions to External fragmentation

- **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

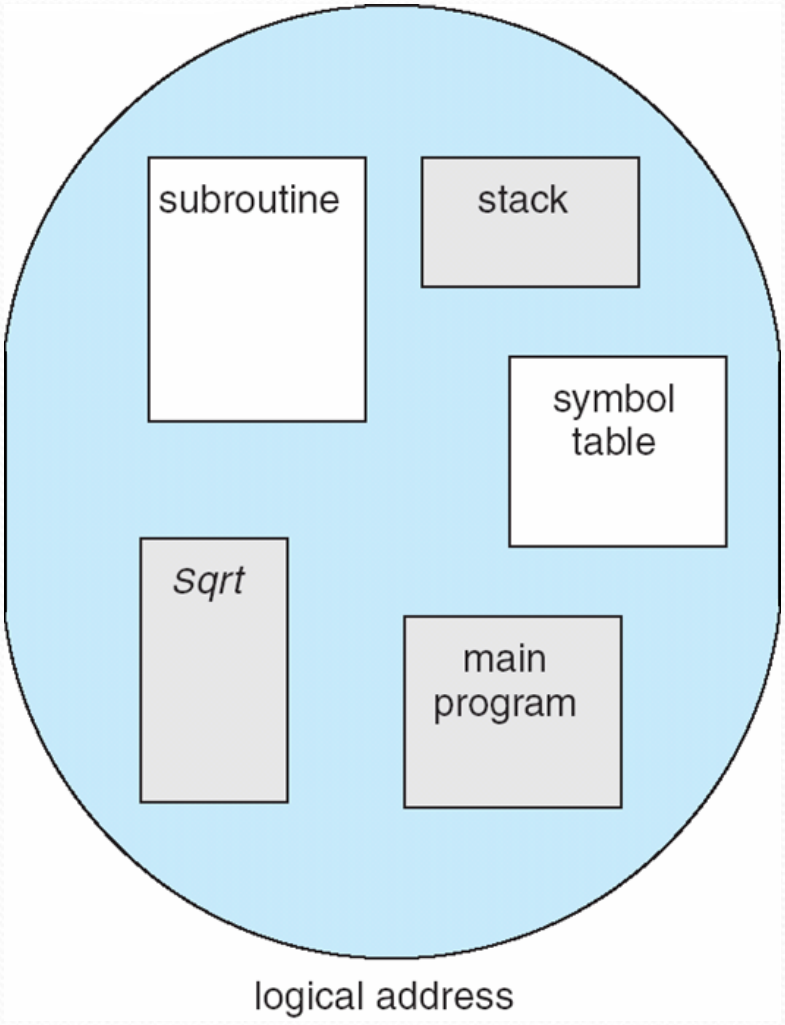
- **Segmentation and Paging**
  - permit the logical address space of the processes to be non-contiguous

Segmentation

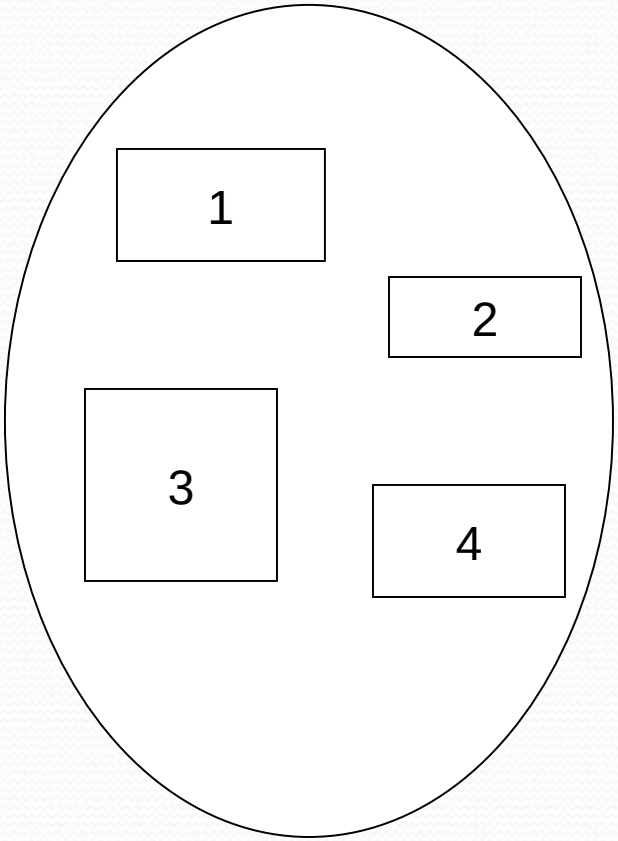
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    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

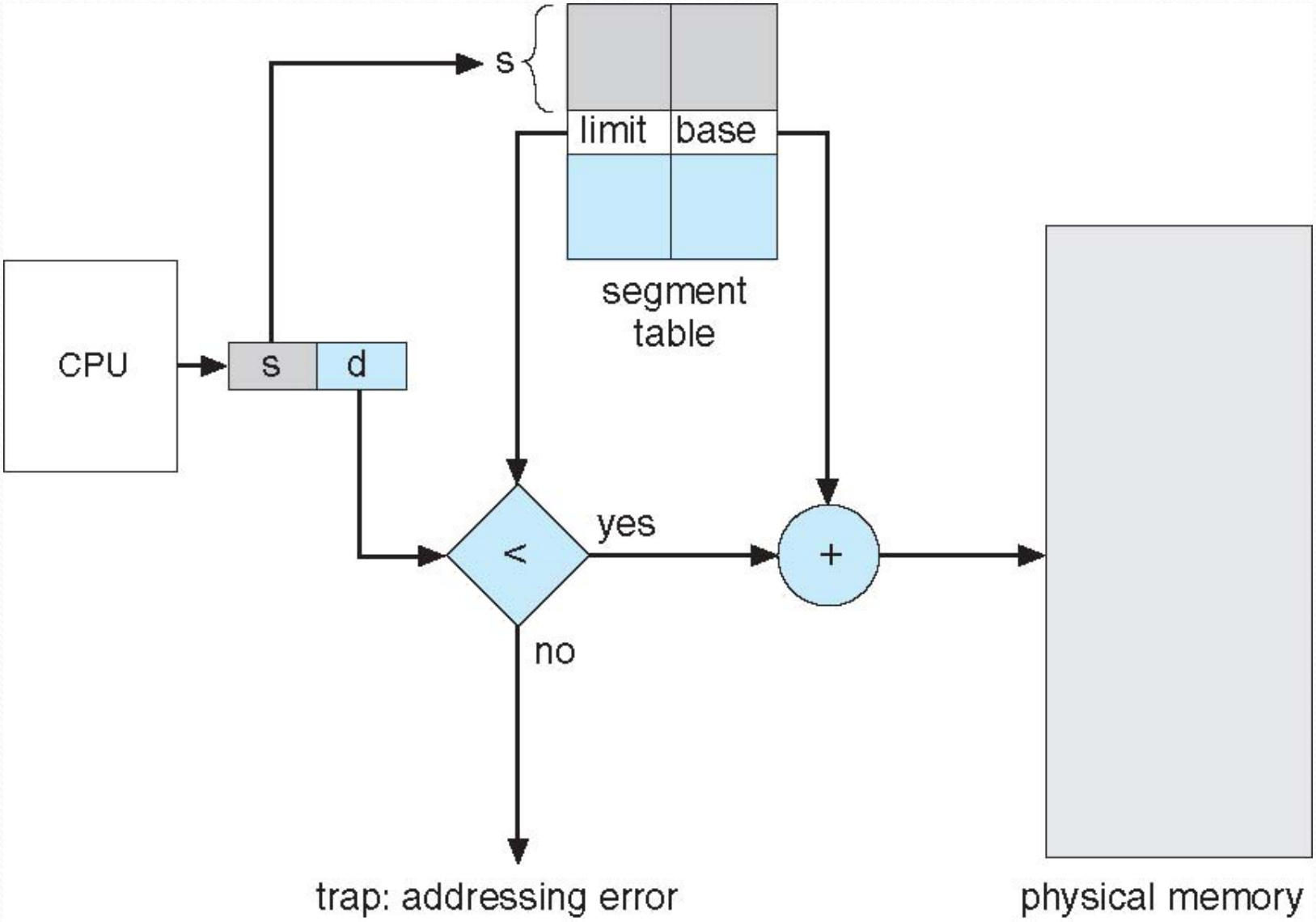
- Each segment has a name and a length
- Logical address consists of a two tuple:

  <segment-number, offset>,

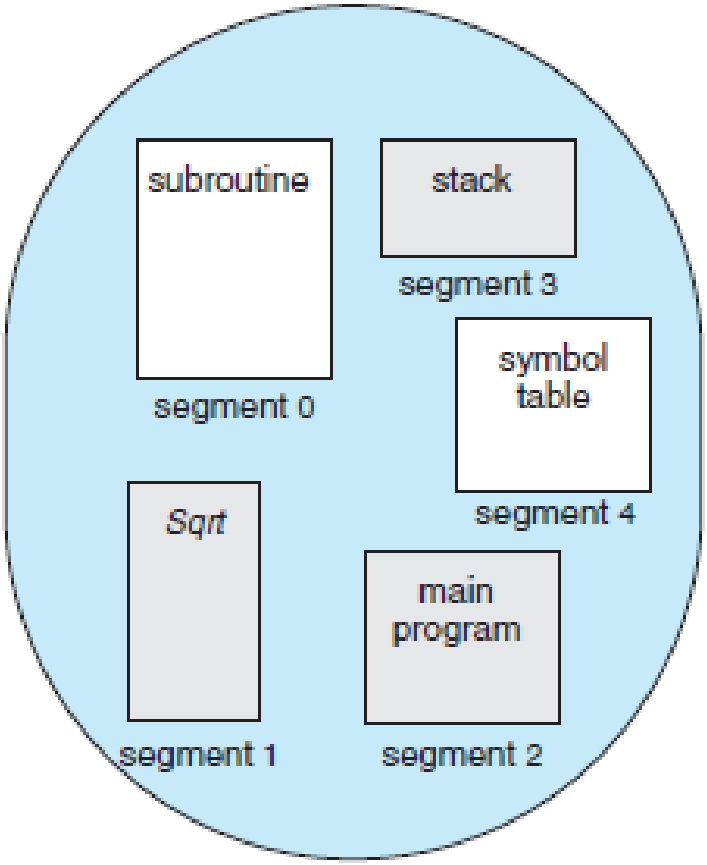
- **Segment table** – maps two-dimensional user-defined addresses (logical address) into one-dimensional physical address; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table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

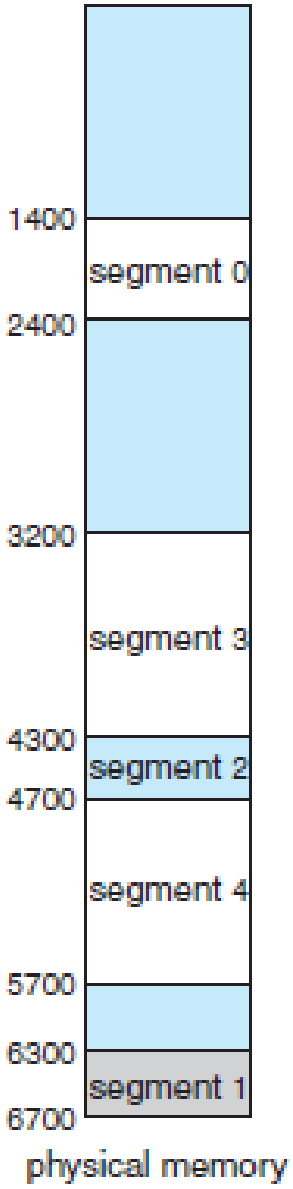


Segmentation Example



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

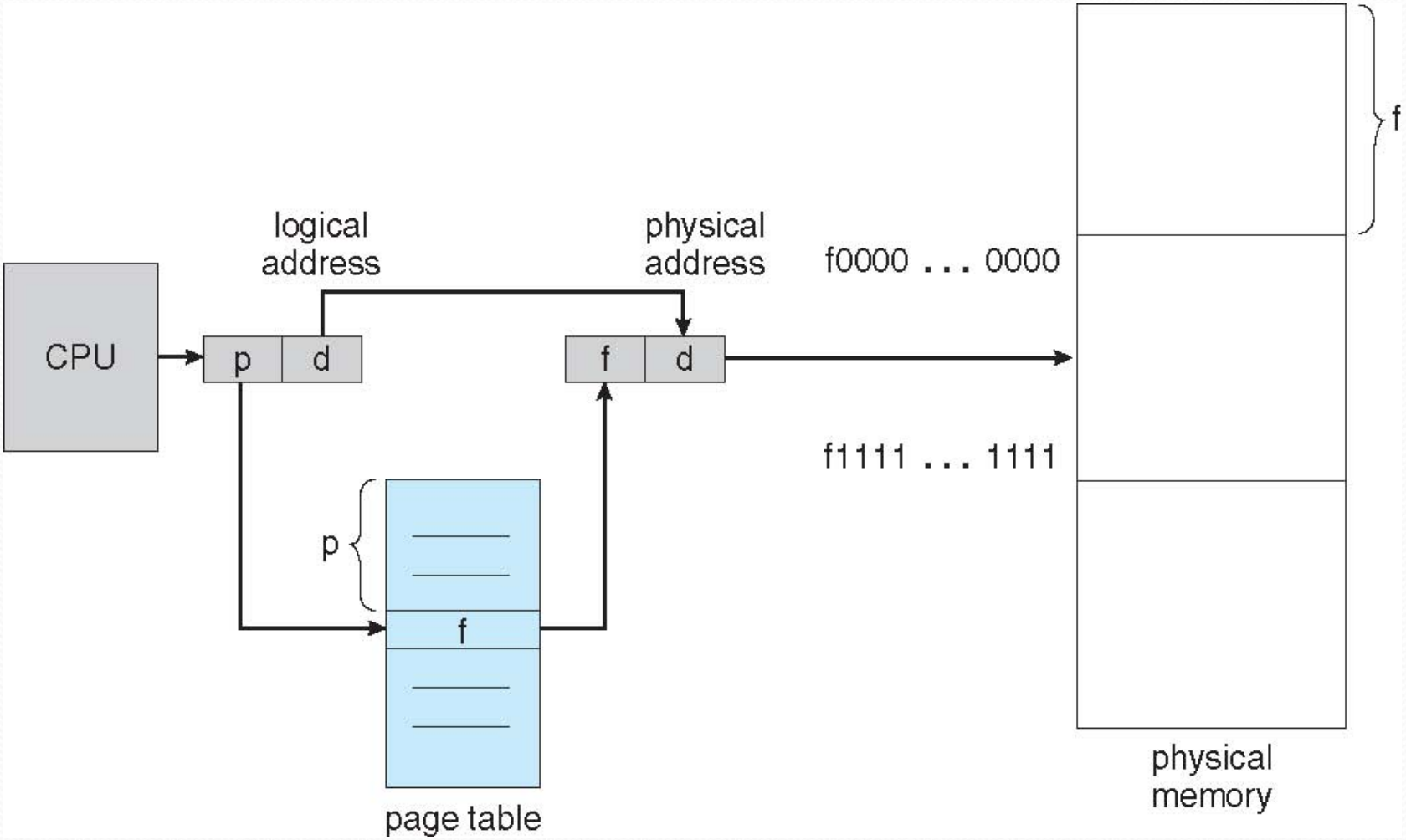
segment table



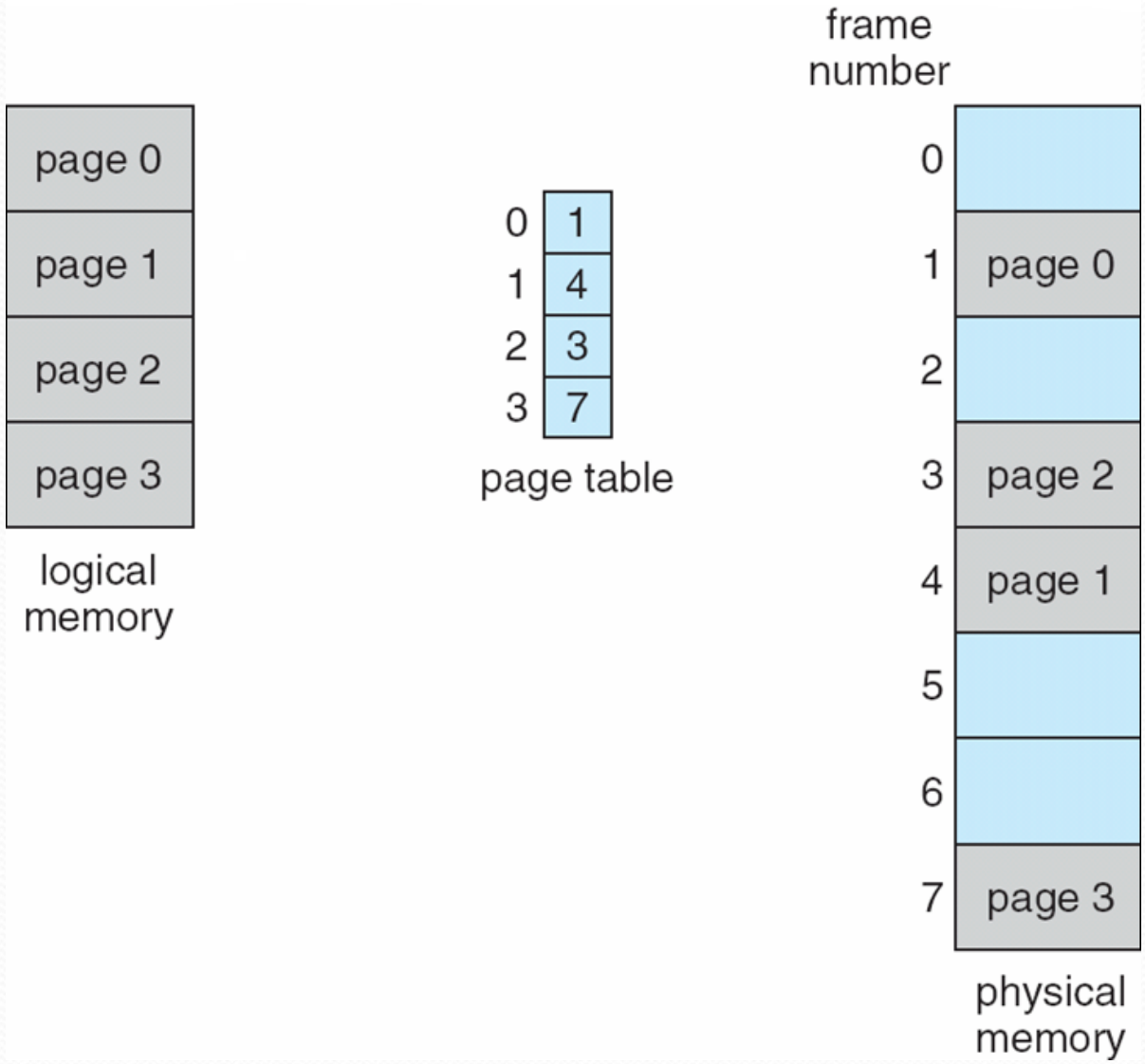
Paging

- Memory management scheme
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks - **frames**
  - Size is power of 2, between 512 bytes and 1 GB
- Divide logical memory into blocks of same size -**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
- Still have Internal fragmentation

Paging Hardware



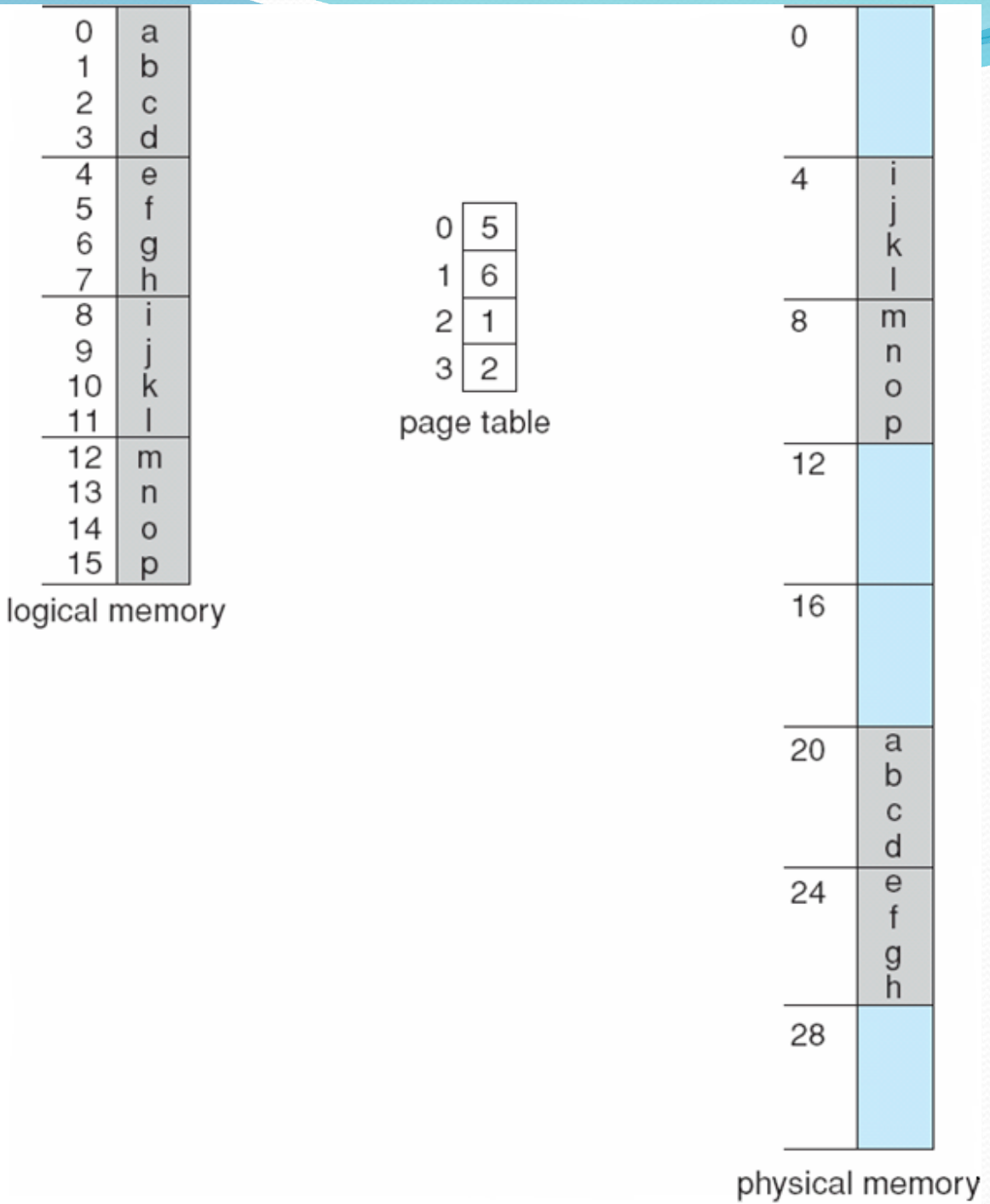
Paging Model of Logical and Physical Memory



Paging Example

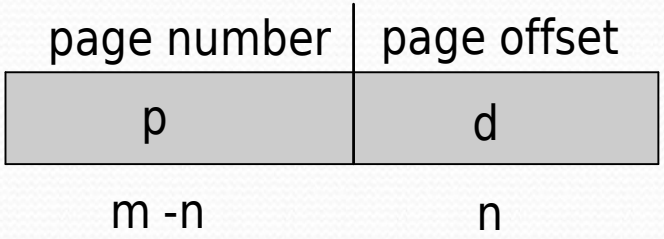
n=2 and *m*=4

32-byte memory and
4-byte pages



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 (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes

  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size

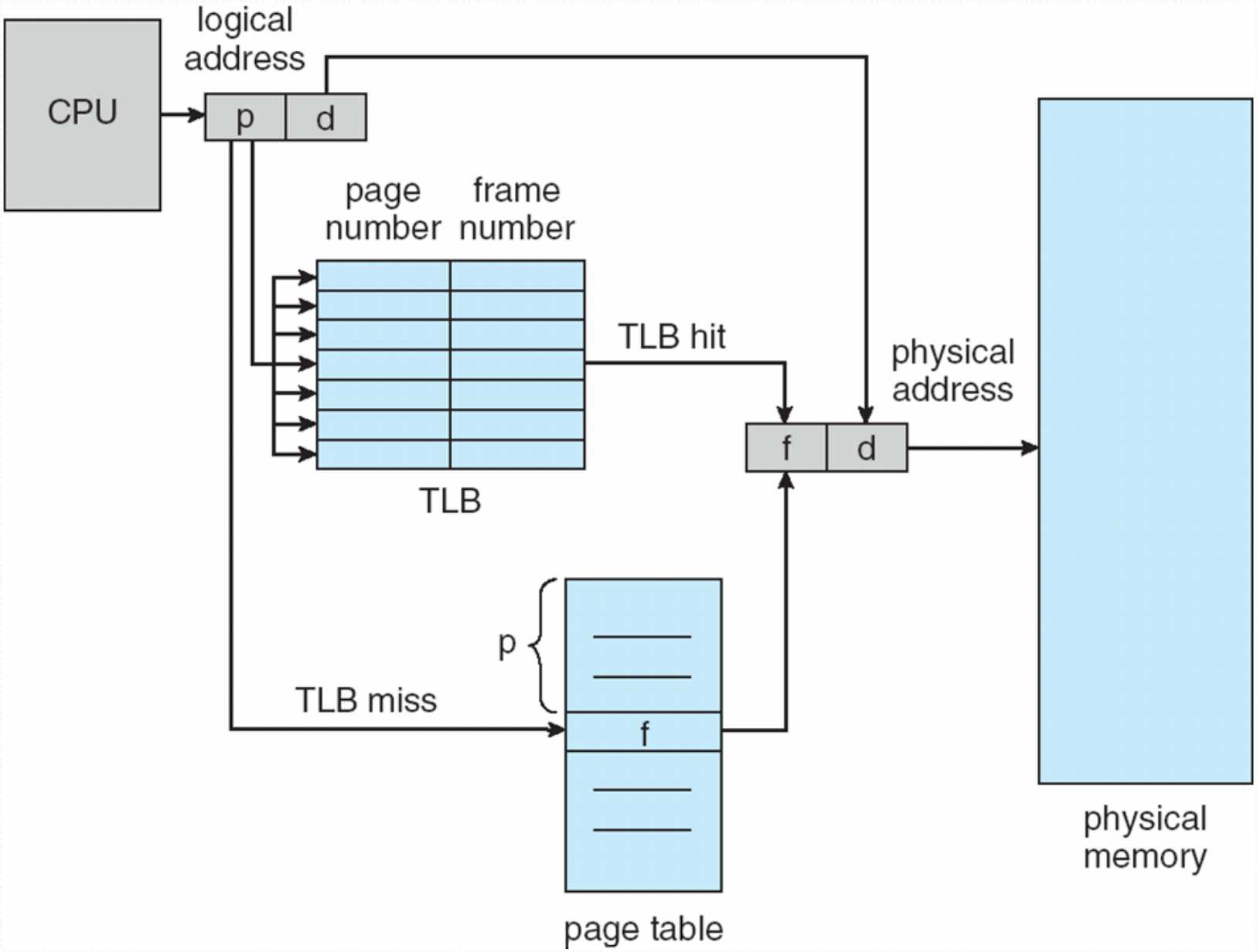
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**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**)

Translation Look-aside Buffer(TLB)

- Associative, high-speed memory
- Each entry in the TLB consists of two parts: a key (or tag) and a value
- TLBs are typically small (64 to 1,024 entries)
- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered

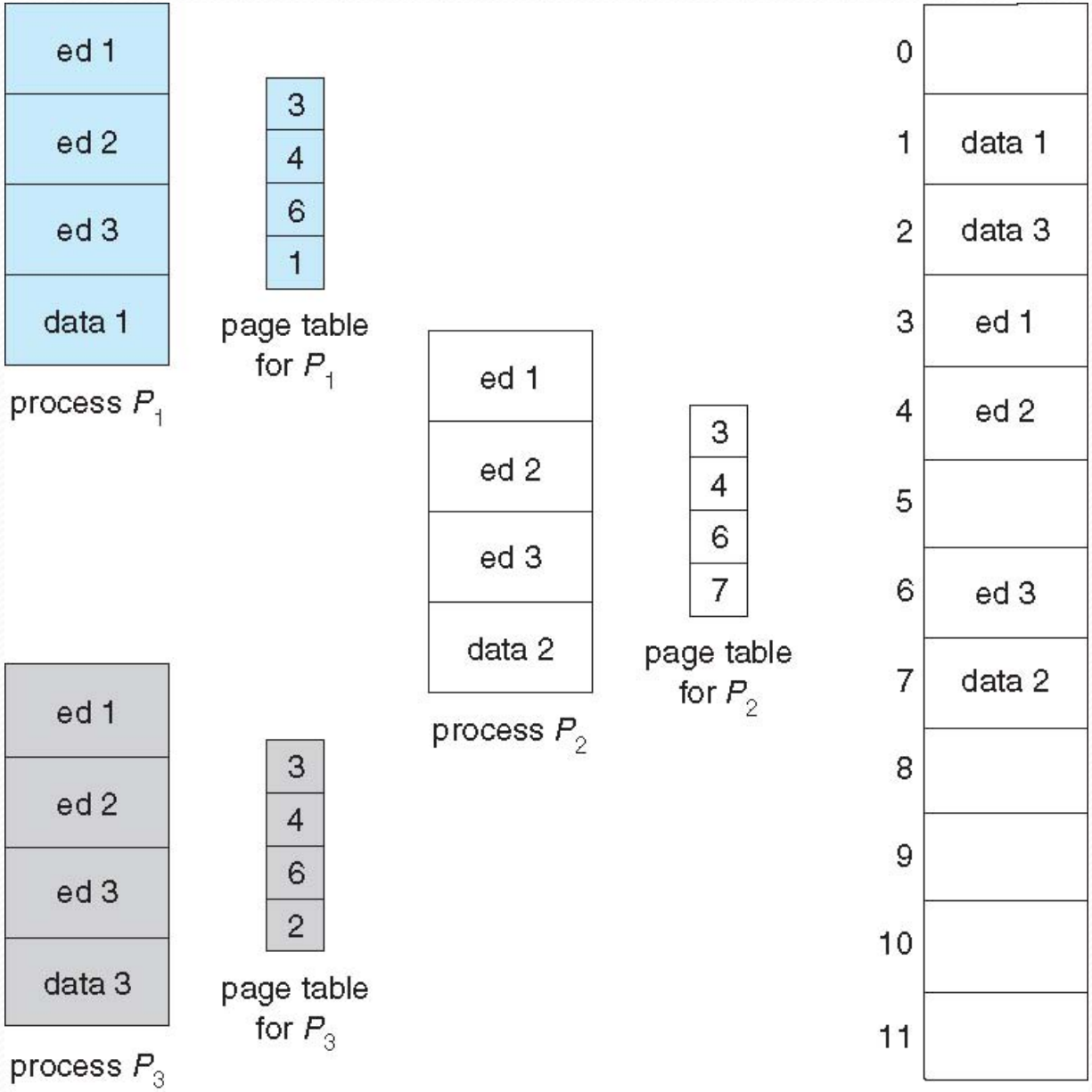
Paging Hardware With TLB



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 inter-process 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



Structure of the Page Table

Some of the most common techniques for structuring the page table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

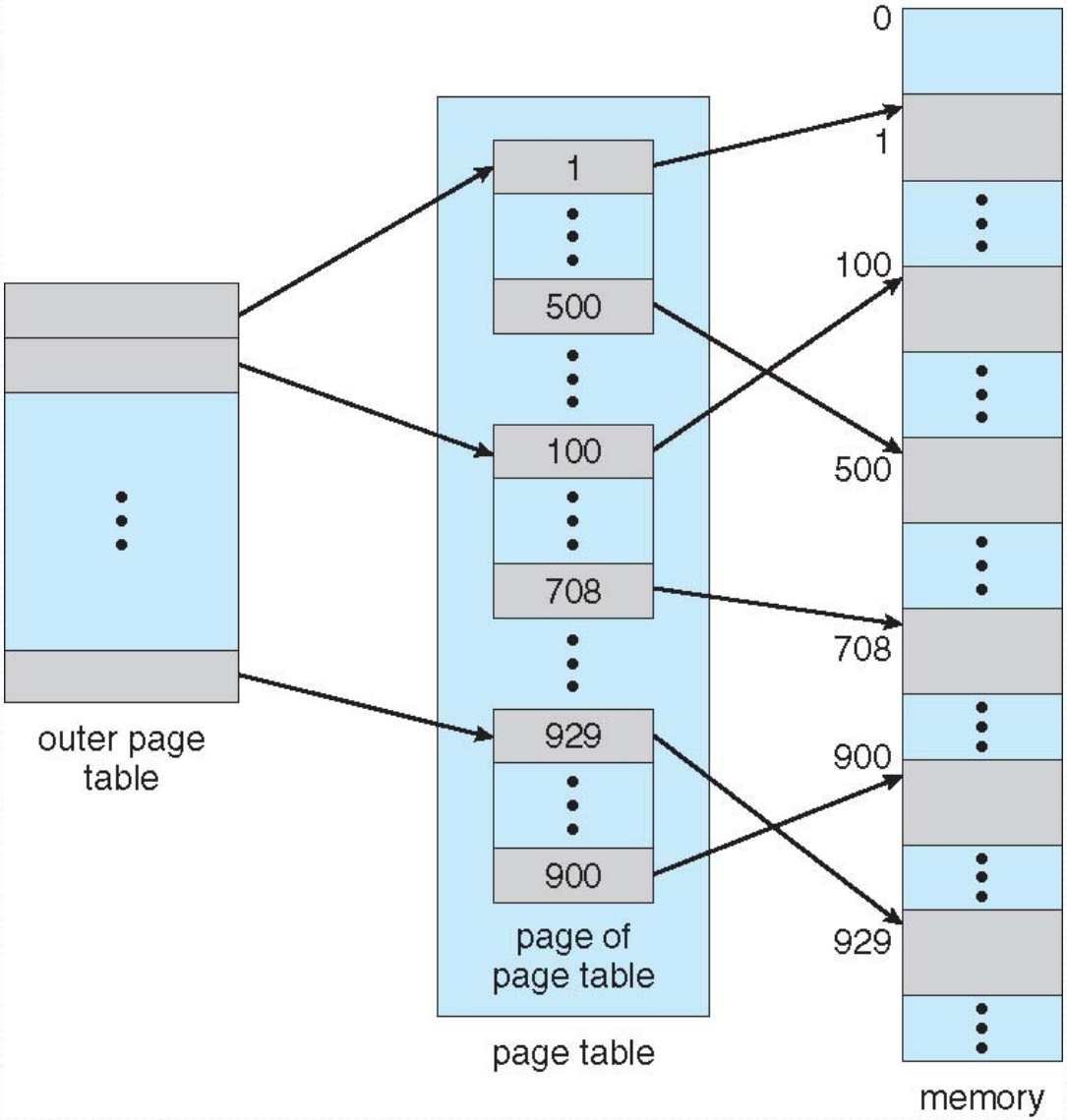
Consider a 32-bit logical address space as on modern computers and a Page size of 4 KB (2^{12})

- Page table would have 1 million entries (2^{32} / 2^{12})
- If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone

Hierarchical Page Tables

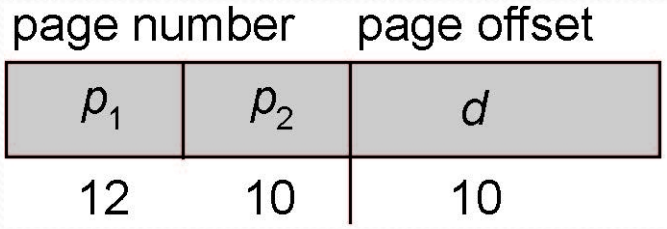
- One simple solution to this problem is to divide the page table into smaller pieces
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



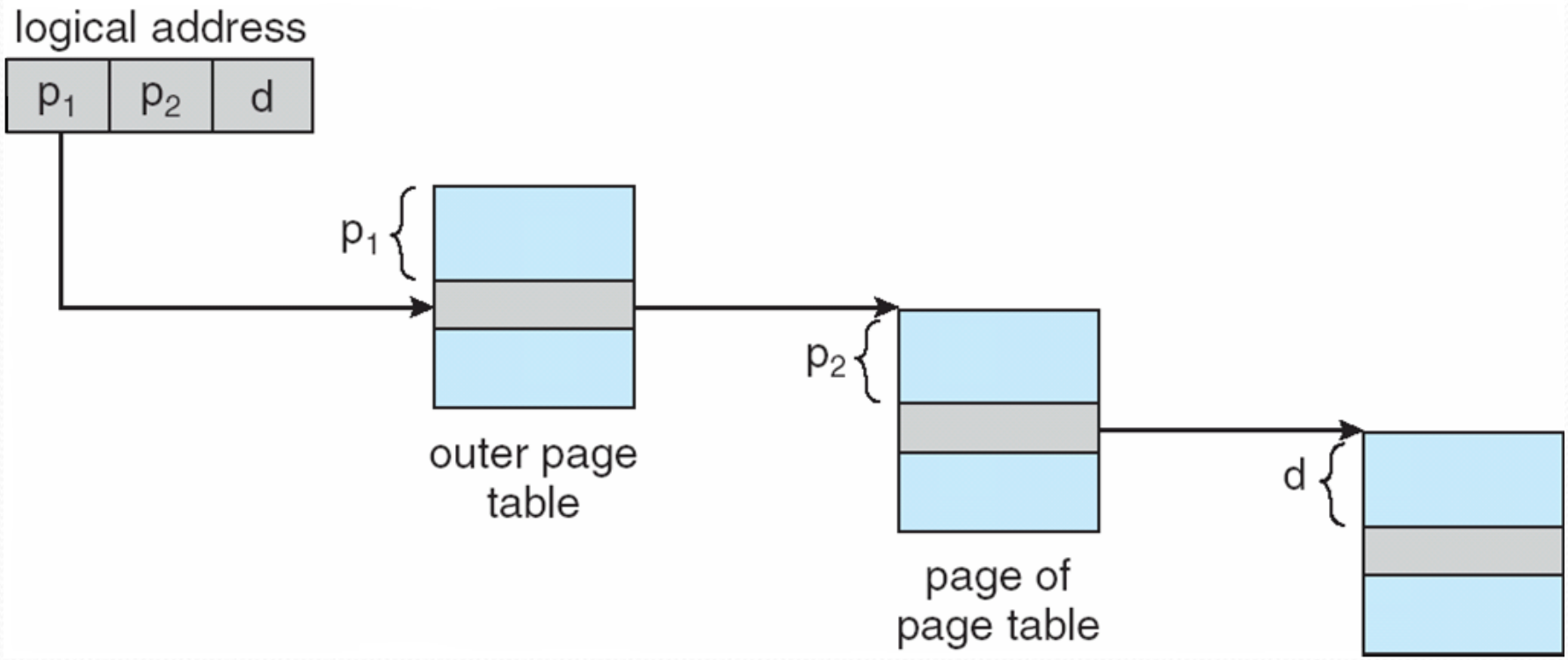
Two-Level Paging Example

- 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 follows:



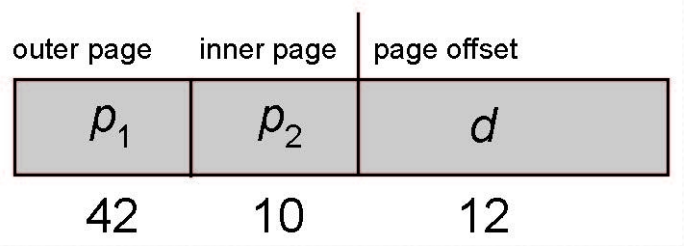
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



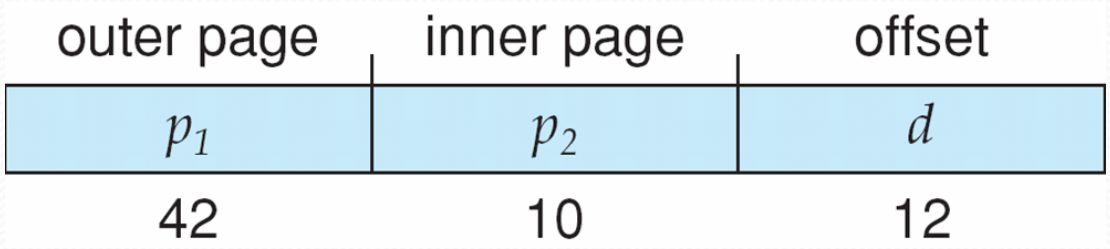
64-bit Logical Address Space

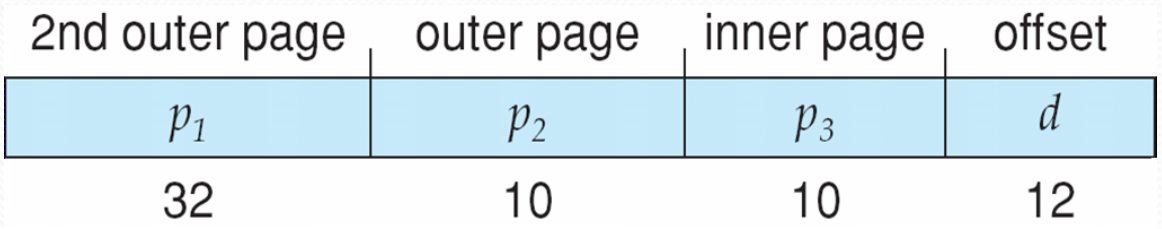
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} 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

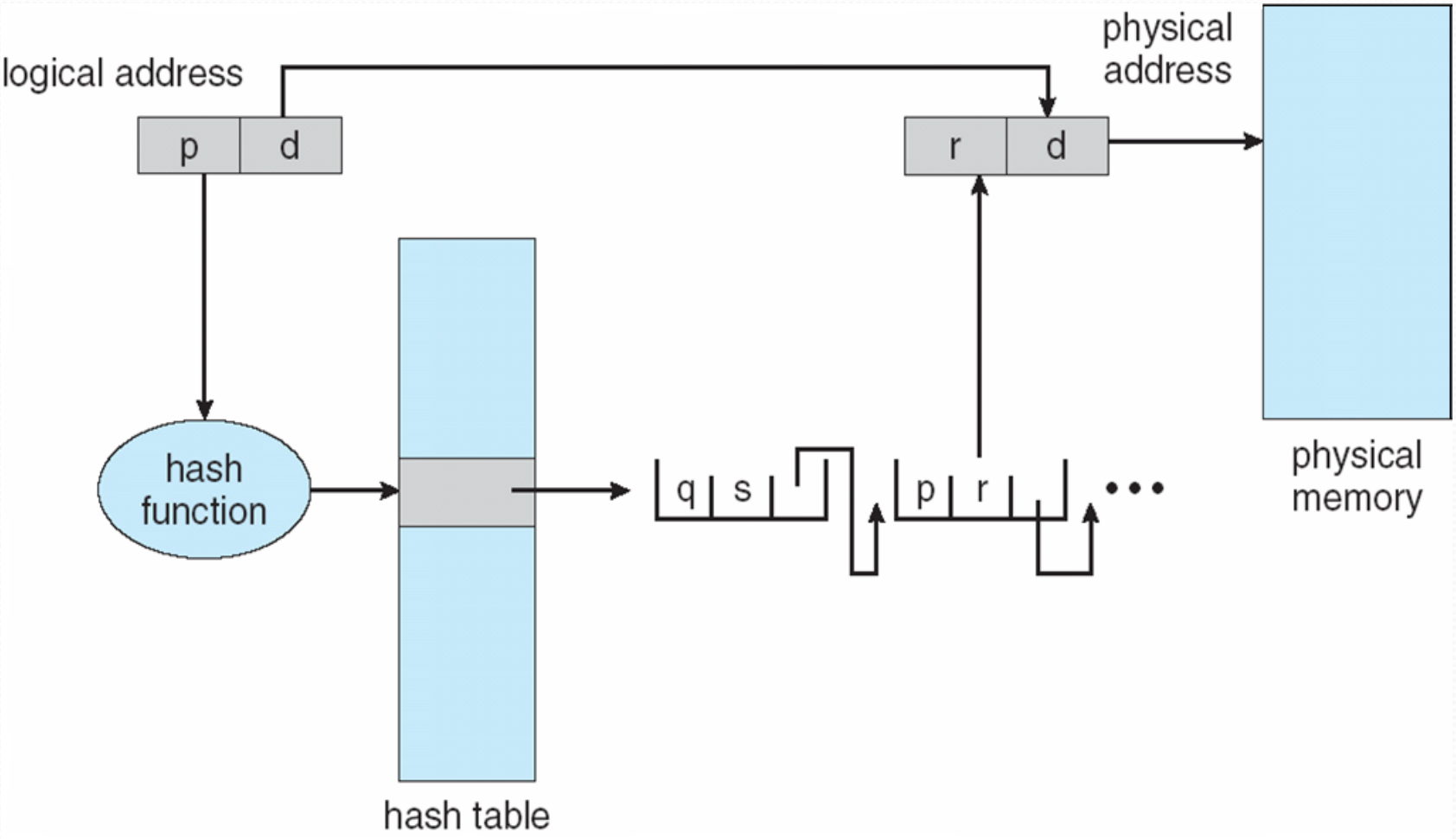




Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**
- 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 searching for a match
  - If a match is found, the corresponding physical frame is extracted

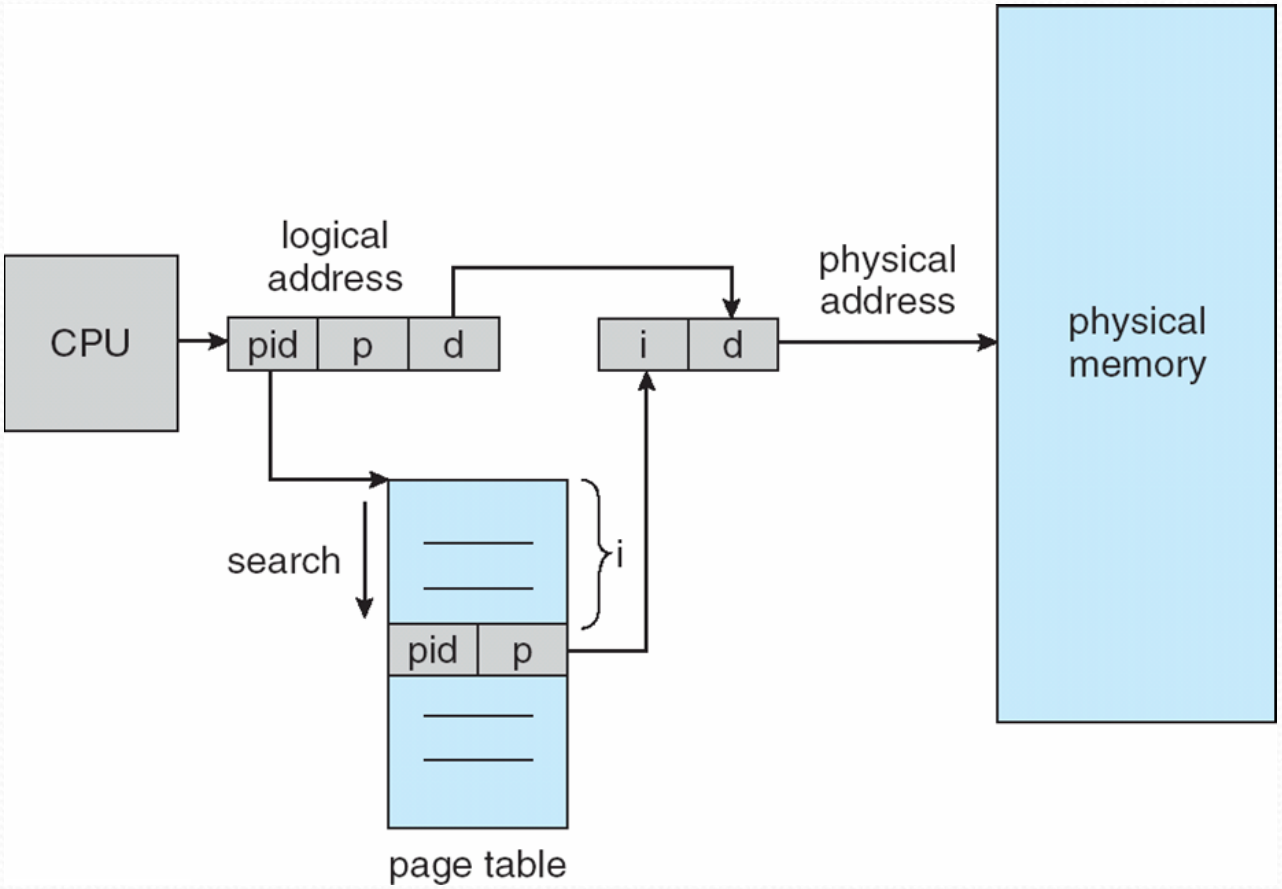
Hashed Page Table



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 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
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Any Queries?