SLIIT

Discover Your Future

IT2060/IE2061

Operating Systems and System Administration

Lecture 08

Memory Management

U. U. Samantha Rajapaksha

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

Samantha.r@slit.lk

SLIIT
FACULTY OF COMPUTING

Chapter 8: Memory Manageme

- Background
- Swapping
- Contiguous Memory Allocation
- Paging

Main Memory – Chapter 8

- Memory is a large array of words or bytes, each with its own address
  - It is a repository of quickly accessible data shared by CPU and I/O devices.
  - Memory and registers are the only storage that CPU can directly access
- Main memory is a volatile storage device
  - It loses its contents in the case of system failure.
- A program must be mapped to absolute addresses and loaded into memory.
- Selection of a memory management scheme for a specific system depends on many factors, especially on the hardware design of the system.
- The OS is responsible for the following activities:
  - Keep track of which parts of memory are being used and by whom.
  - Decide which processes to load next when memory space becomes available.
  - Allocate and deallocate memory.

For n bits address, the memory capacity = 2^n bytes

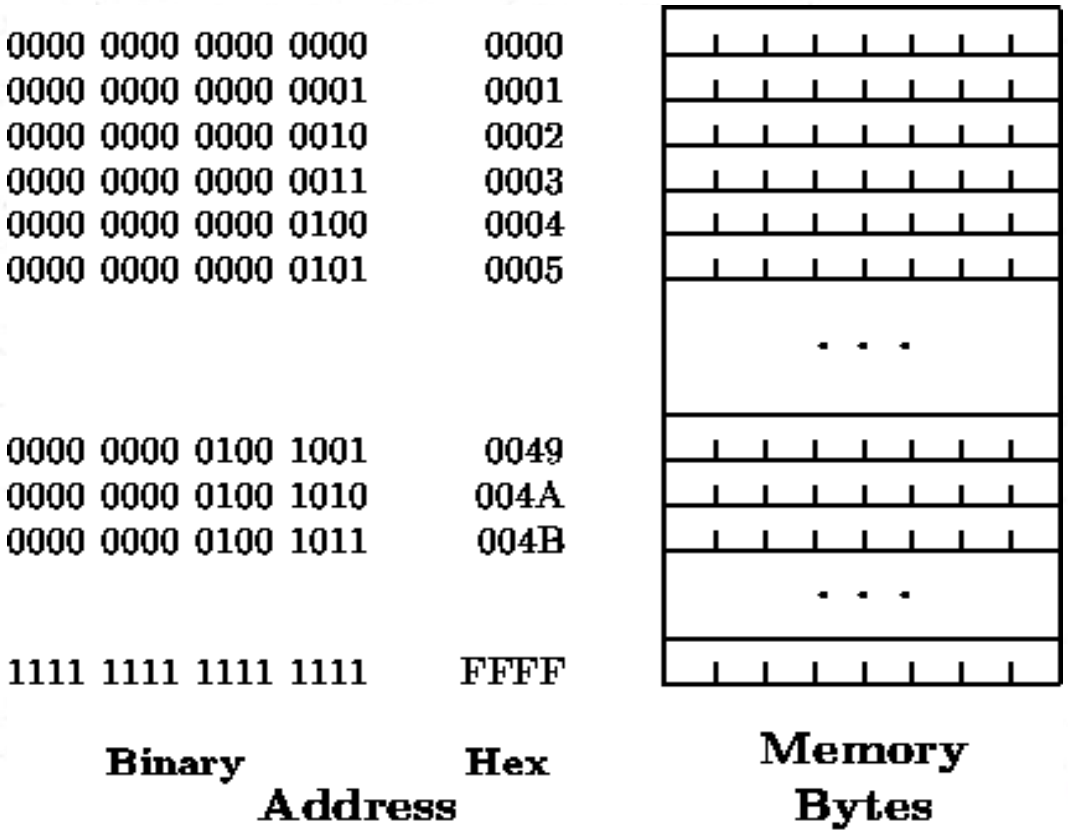


Figure 1.2: Memory and Addresses

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 in one CPU clock (or less)
- Protection of memory required to ensure correct operation

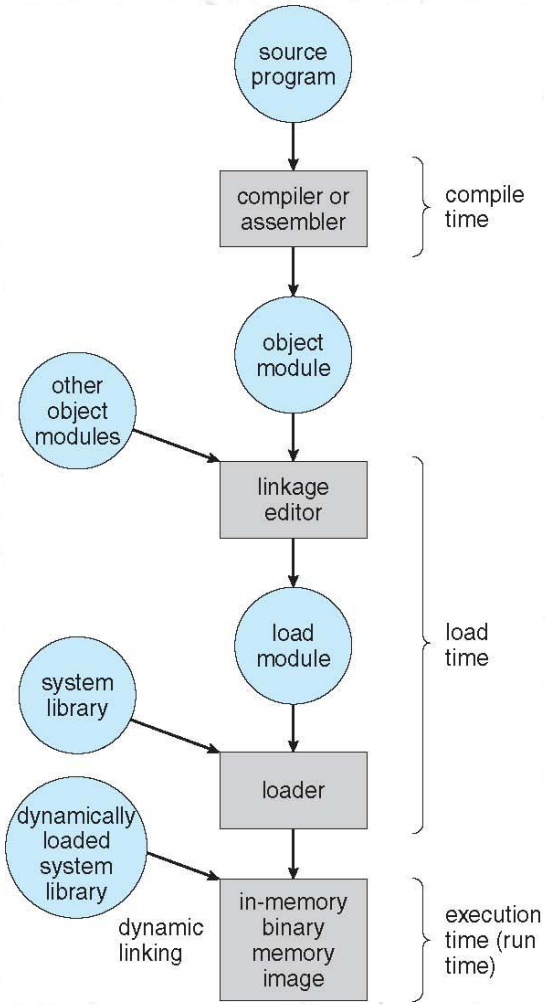
Address Binding

- 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?
- Further, 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 memo 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 Sp

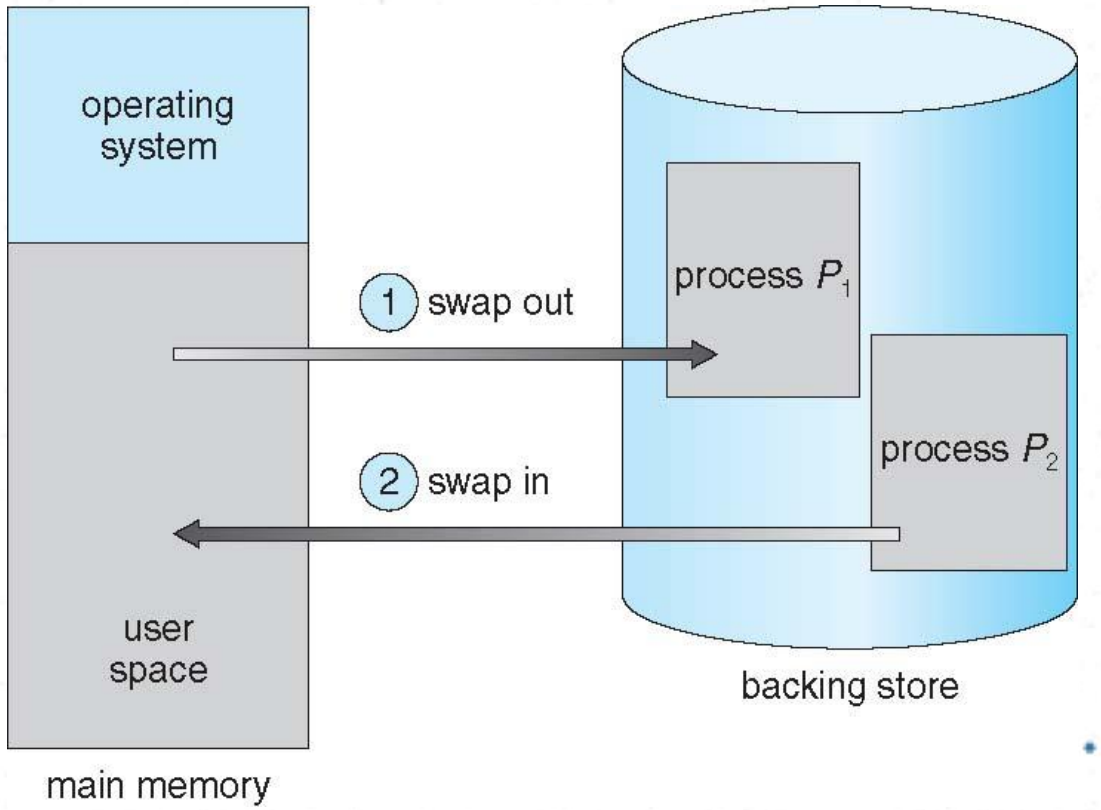
[redacted]

- The concept of a logical address space that is boun 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

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping

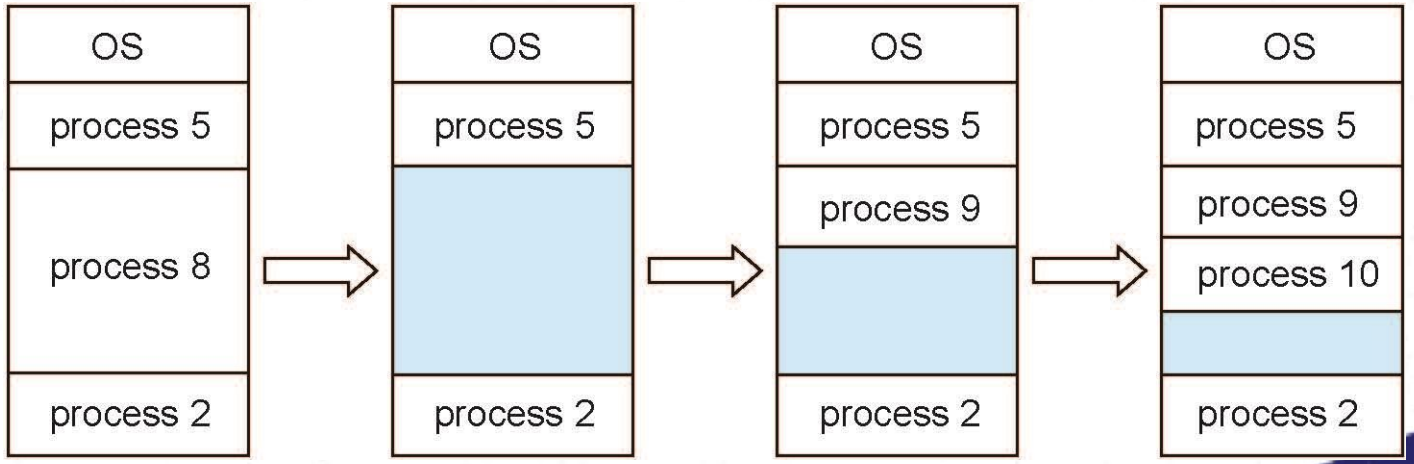


Contiguous Allocation

- 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

Multiple-partition allocation

- 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 Pr

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**

Fragmentation (Cont.)

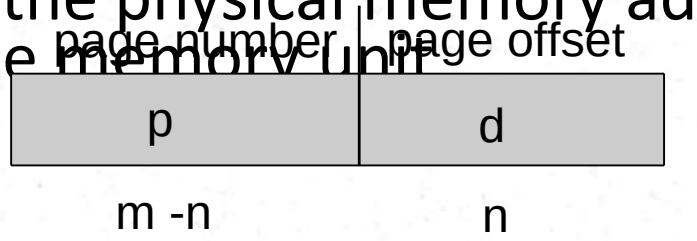
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O 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

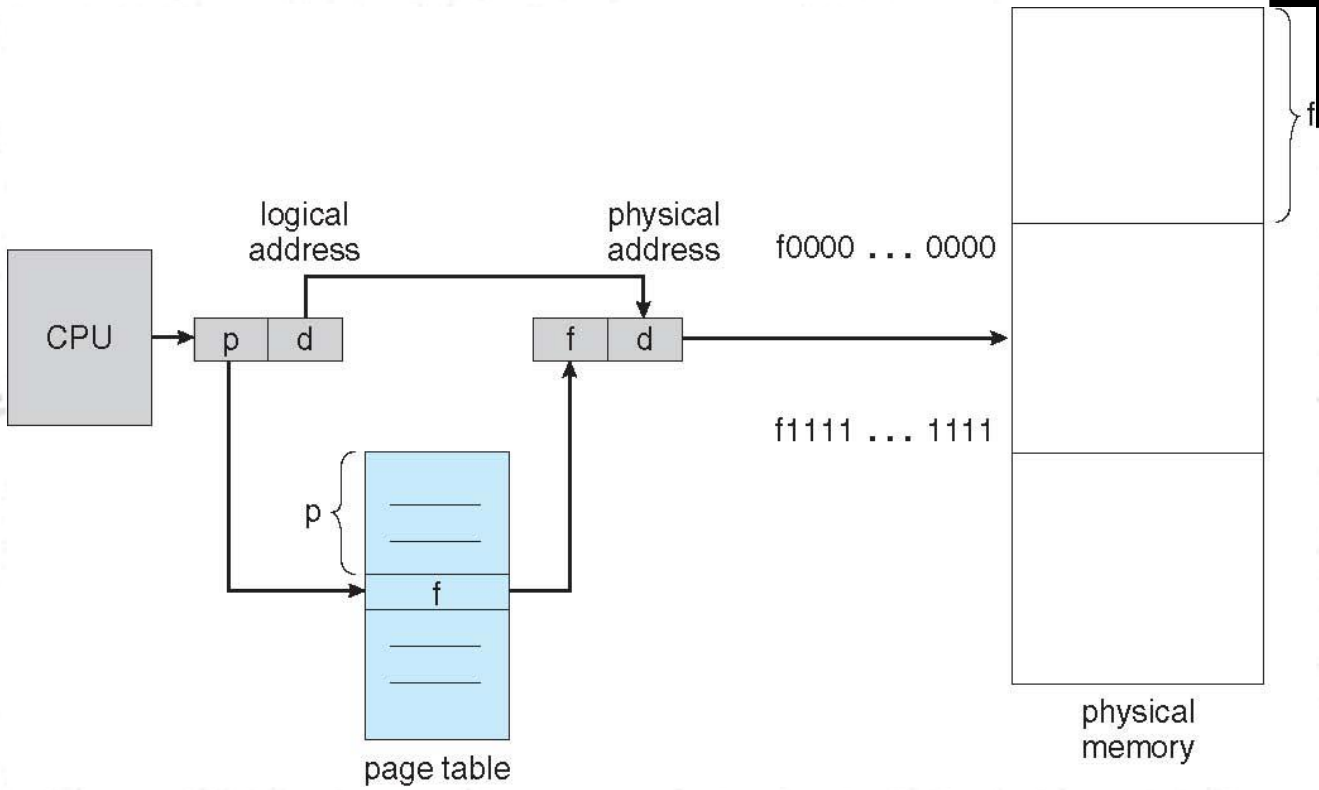
Address Translation Scheme

- 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

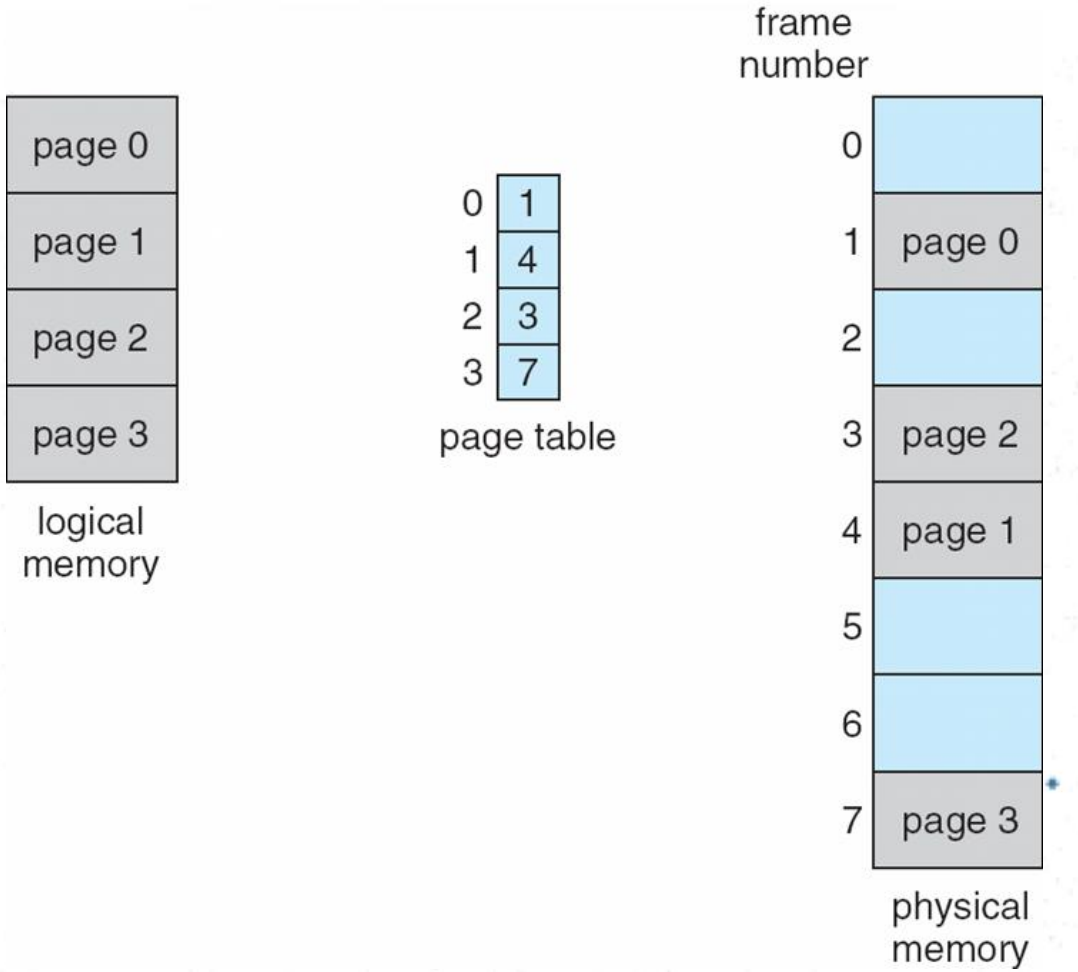


  - For given logical address space 2^m and page size 2^n

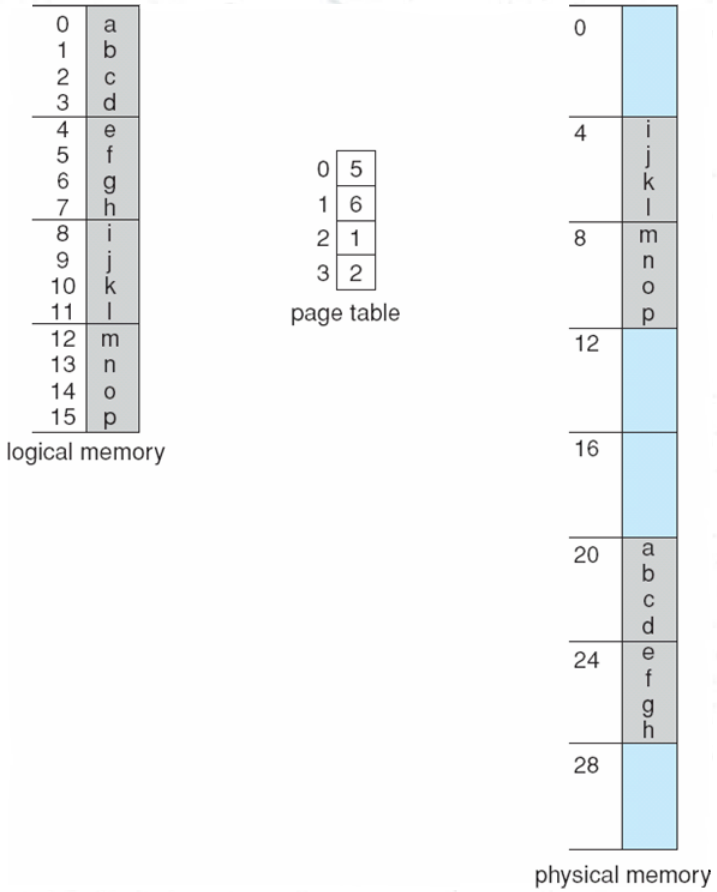
Paging Hardware



Paging Model of Logical and Physical Memory

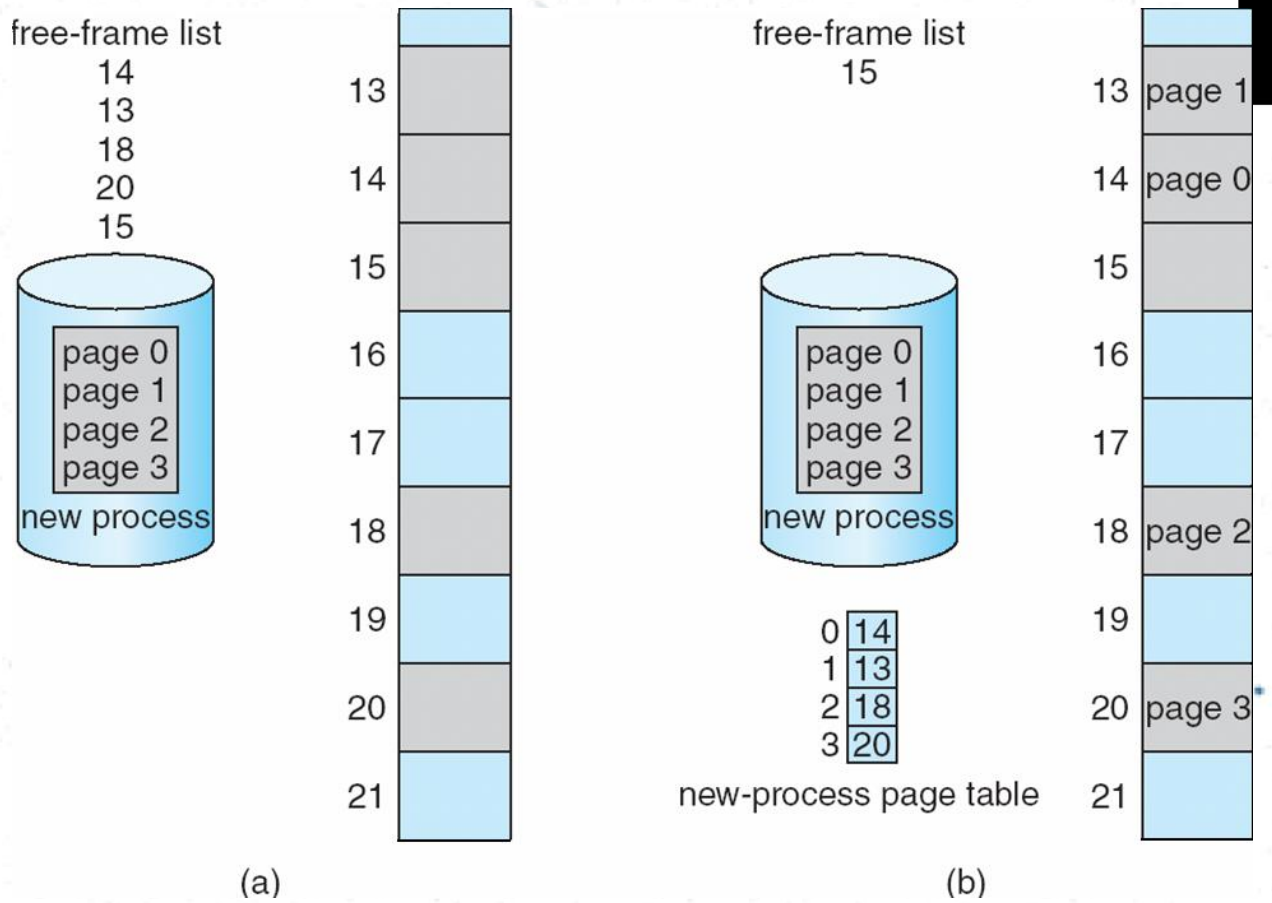


Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

Free Frames



Before allocation

After allocation

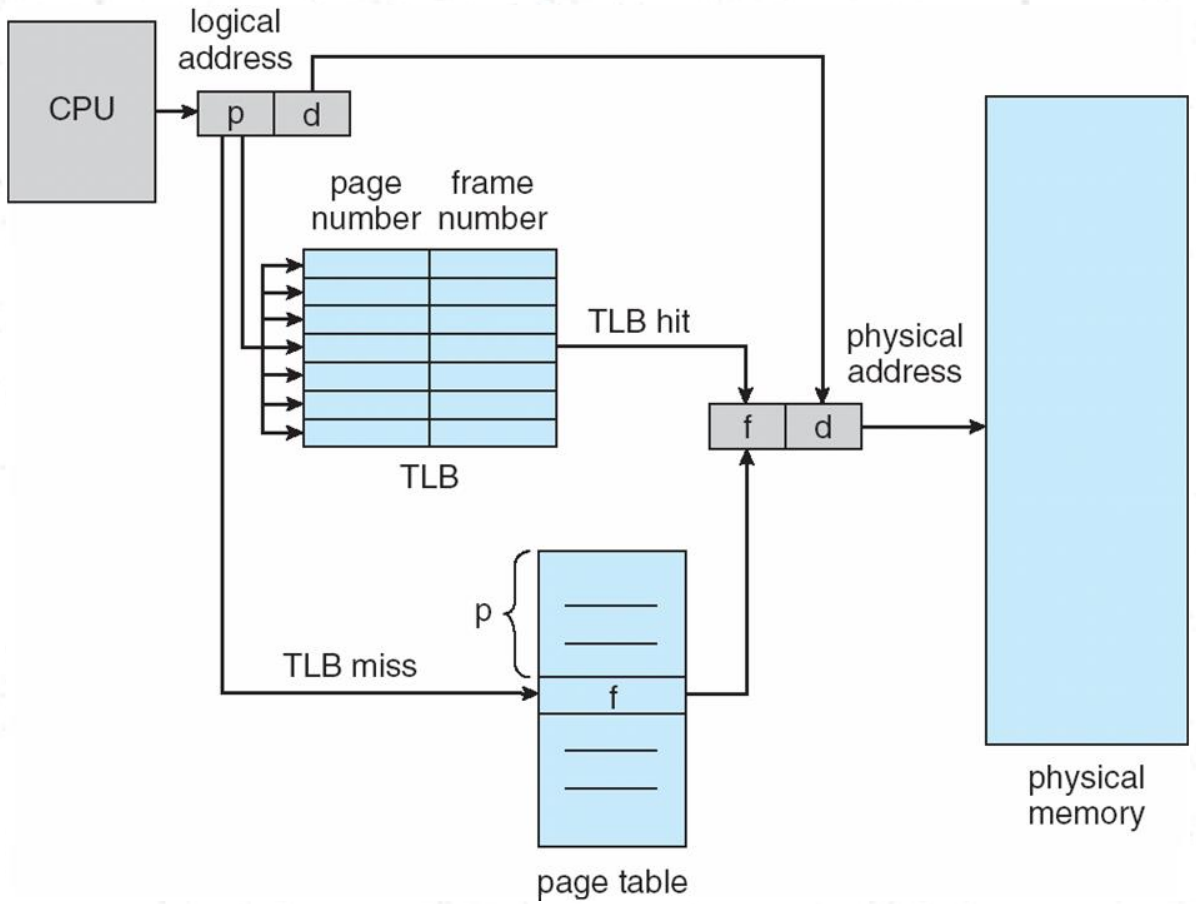
Associative Memory

- 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



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

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

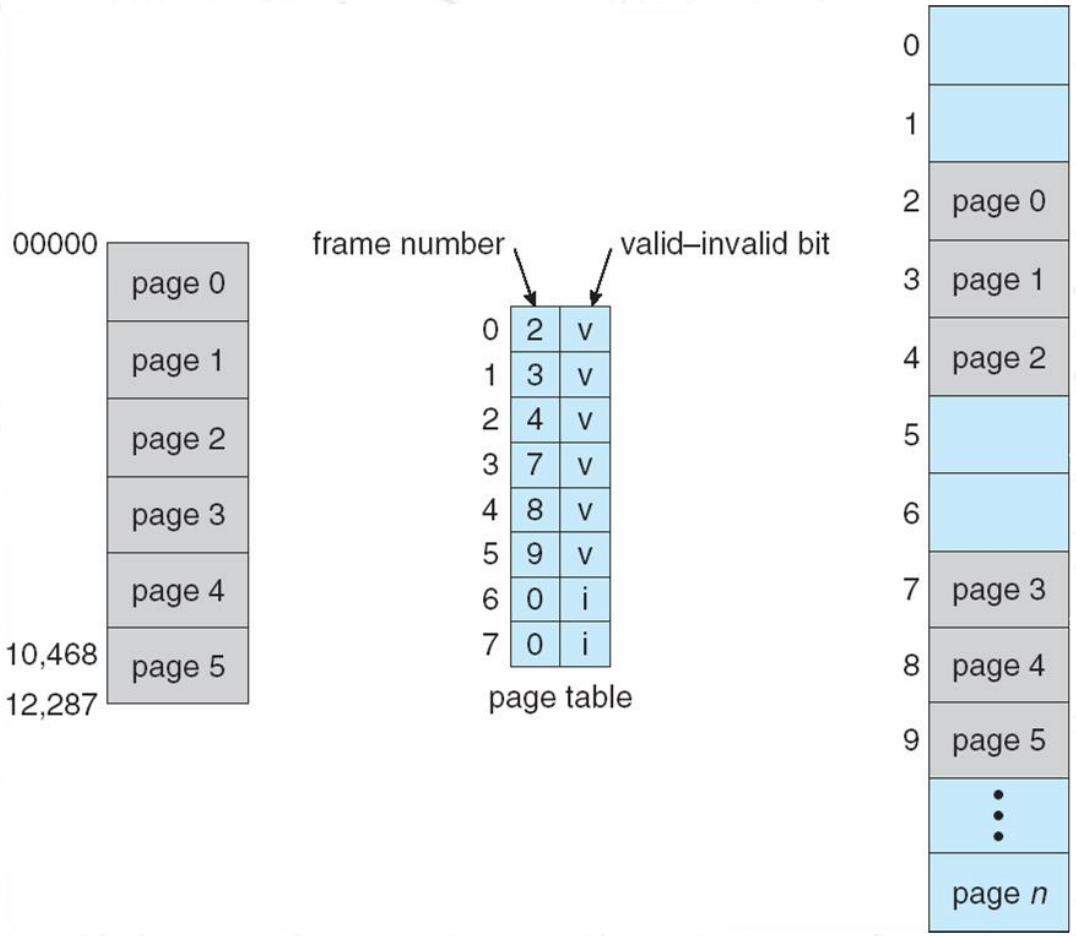
$$= 2 + \varepsilon - \alpha$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

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



Page Table Structure

- Hierarchical Paging.
- Hashed Page Tables.
- Inverted Page Tables.

Hierarchical Paging

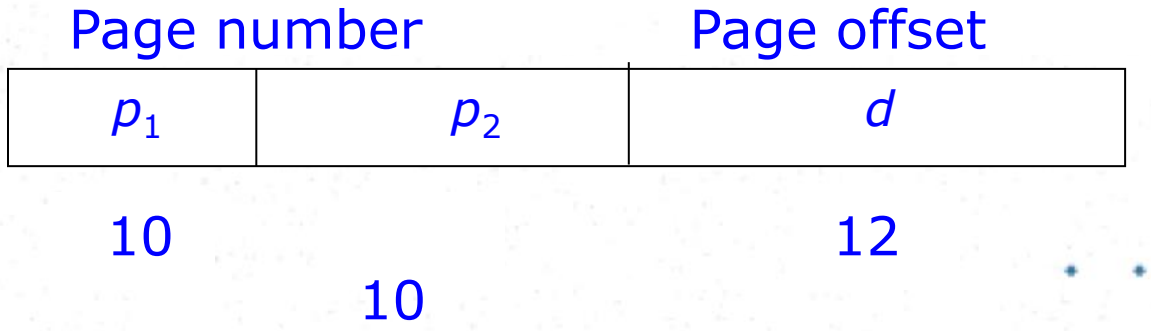
- Most modern computer systems support a very large logical address, e.g., 2^{32} to 2^{64} → page table becomes very large.

  Example: For a system with 32 bit logical address, and page size = 4KB (12 bits offset)

  - page table = 1 Million entries (32-12 = 20 bits) → 4 MB of physical address spaces for page table; 2^{20} * 4 bytes/entry
- One solution is to divide the page table into smaller pieces
  - For a two-level paging scheme, the page table itself is also paged.

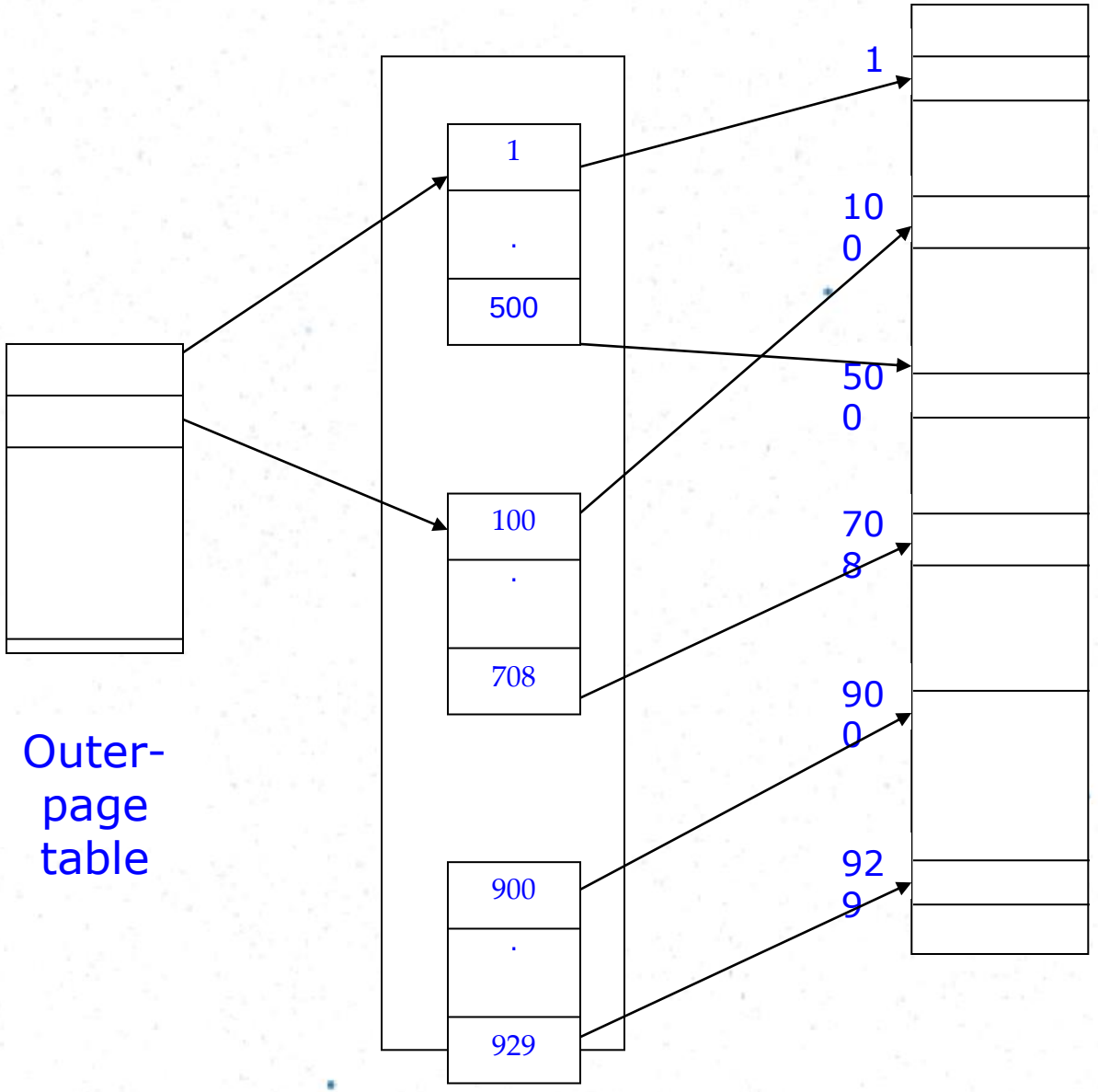
Two-level paging example

- 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:



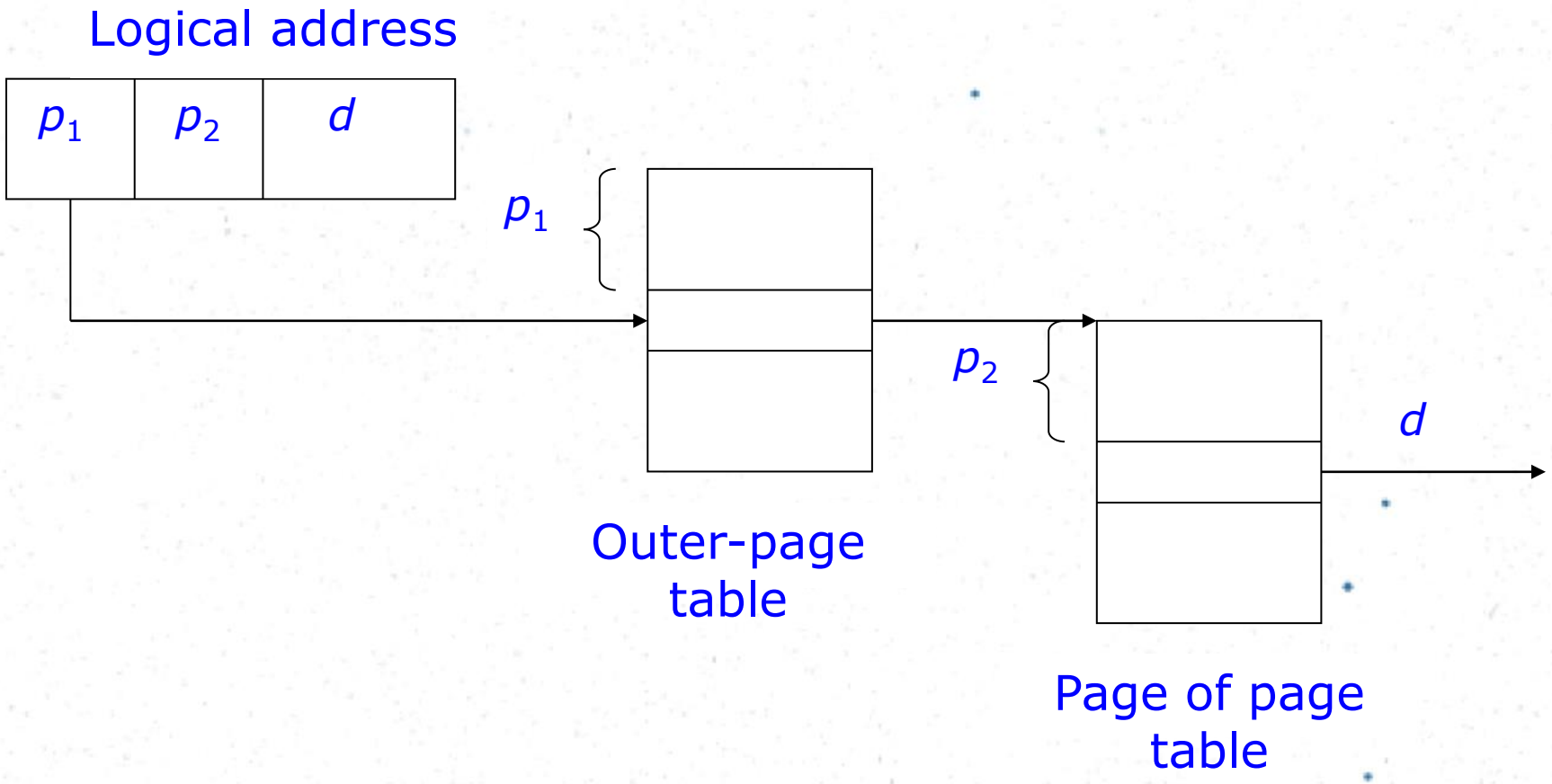
- p_1 is an index into the *outer page table*, and p_2 is the displacement within the page of the *inner page table*
  - Translation starts from the outer page table

Two-level paging example



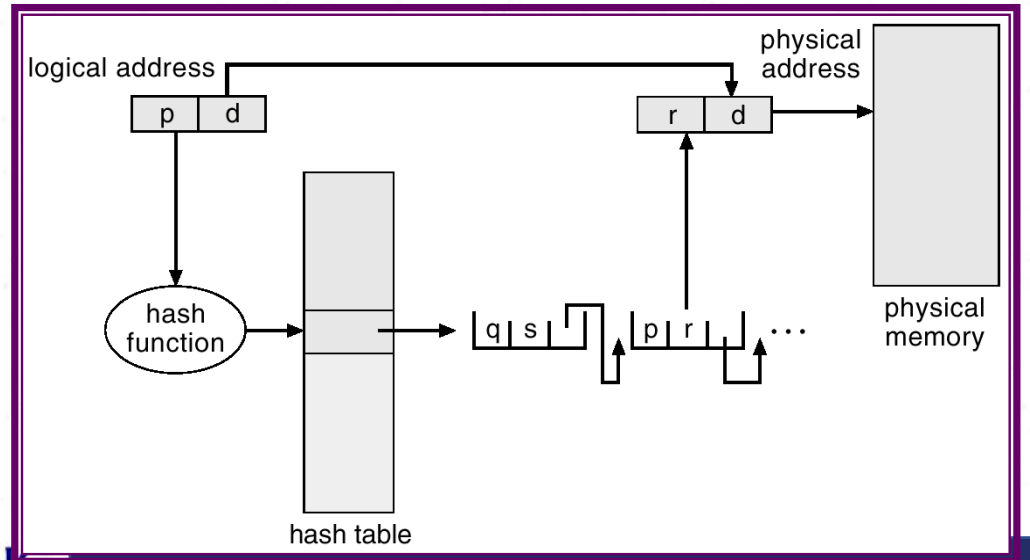
Address-translation scheme

Two-level 32-bit paging architecture



Hashed Page Tables

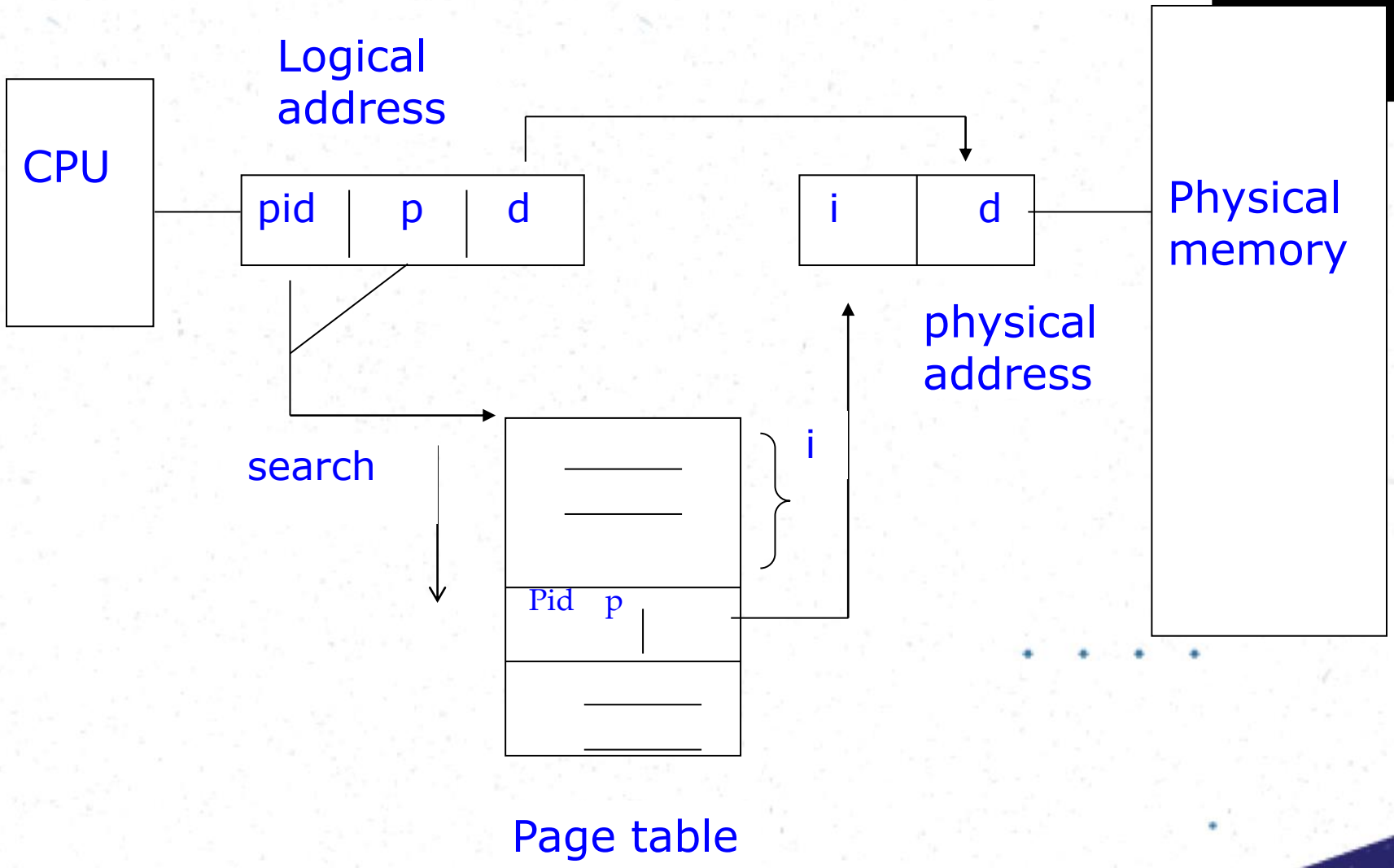
- Is commonly used when address spaces > 32 bits.
- The virtual page number is hashed into a page table.
  - This page table contains a chain of elements hashing to the same location.
  - Each element contains: virtual page number, the value of the mapped frame, pointer to the next element
- Virtual page number is compared in this chain searching for a match.
  - If a match is found, the corresponding physical frame is extracted.
- For address spaces > 64 bits, use Clustered page tables.
  - Similar to hash page table except each entry in page table refers to several pages (e.g., 16).



Inverted Page Table

- In the paging system, each process has a page table associated with it.
  - Drawback: each page table may consist of millions of entries → page tables consume large amount of physical memory.
- To solve this problem, use Inverted Page Table.
  - It uses ONLY one table that has one entry for each real page of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process (e.g., PID) that owns that page.
- This scheme *decreases* memory needed to store each page table, but *increases* time needed to search the table when a page reference occurs.
  - It may use hash table to limit the search to one – or at most a few page table entry.

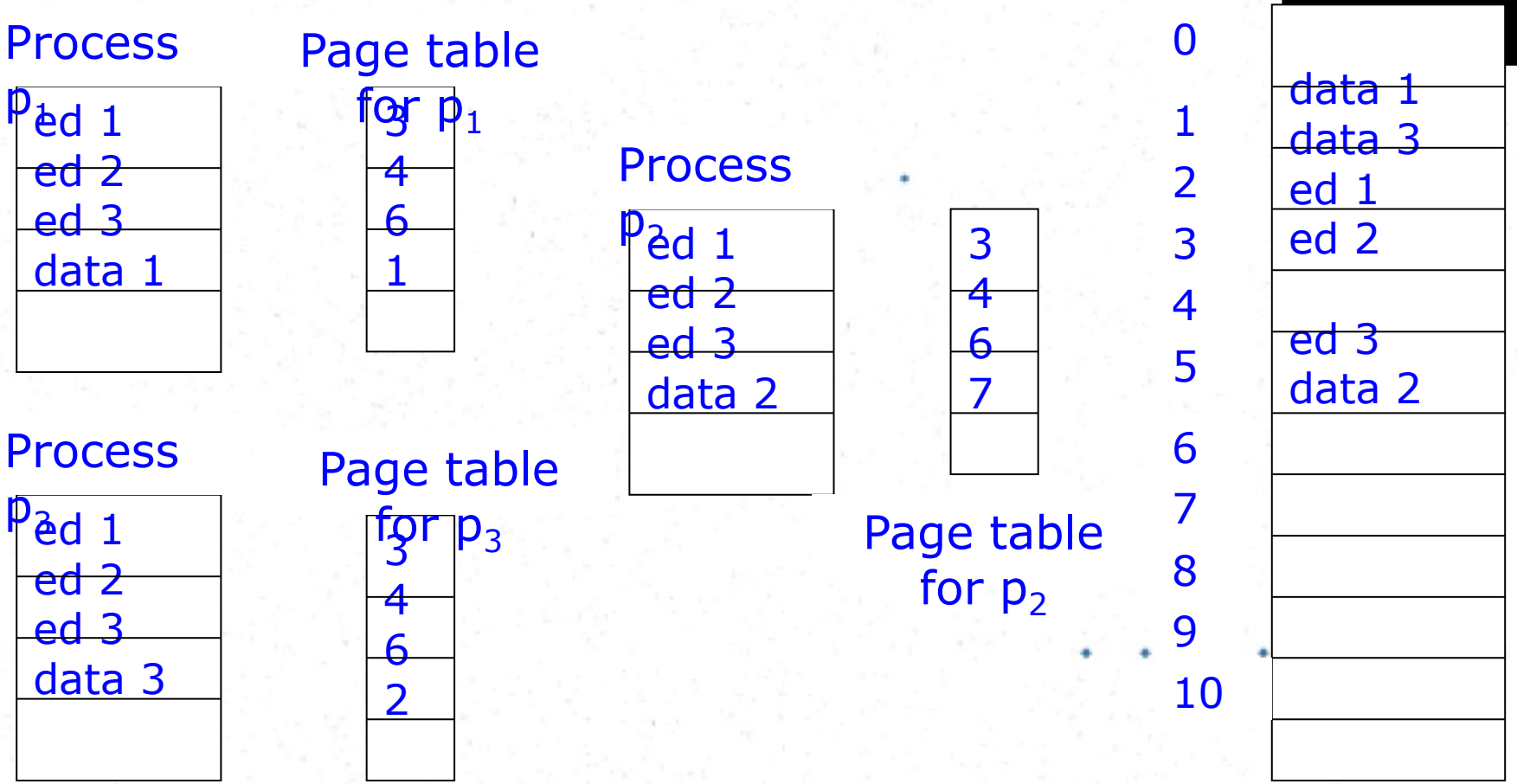
Inverted page table architecture



Shared pages

- Other advantage of paging is the possibility of *sharing* common code (reentrant code also called pure code).
- Reentrant code is non-self-modifying code → its code never change during execution.
- Shared code.
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- 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.

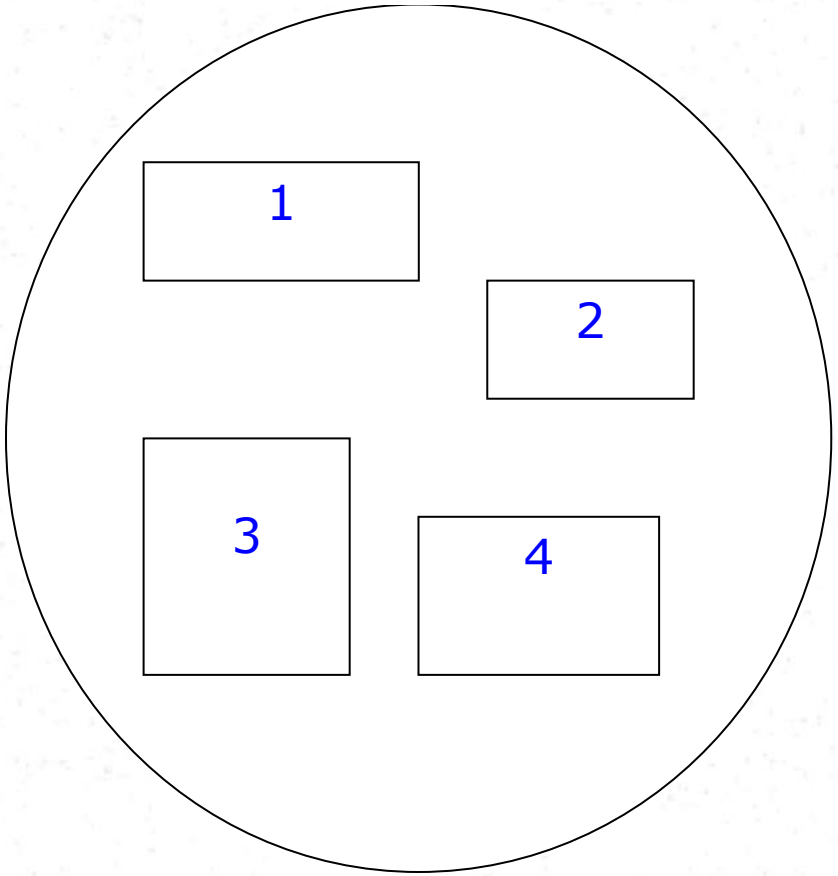
Shared pages example

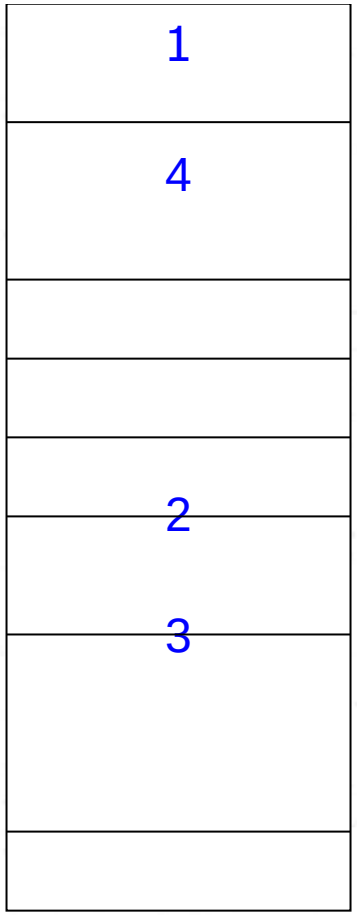


Segmentation

- Memory management with paging makes separation between user view of memory and the actual physical memory.
- Segmentation is memory-management scheme that supports user view of memory.
- From user's view, a program is a collection of segments, and a segment is a logical unit such as:
  - Main program.
  - Procedure / Function.
  - Local variables, global variables.
  - Common block.
  - Stack.
  - Symbol table, arrays.
- Each segment has a name and length; the addresses specify both the segment name and an offset.
- Program specifies each address by two quantities: a segment name, and an offset (within the segment).

Logical view of segmentation





Segmentation architecture

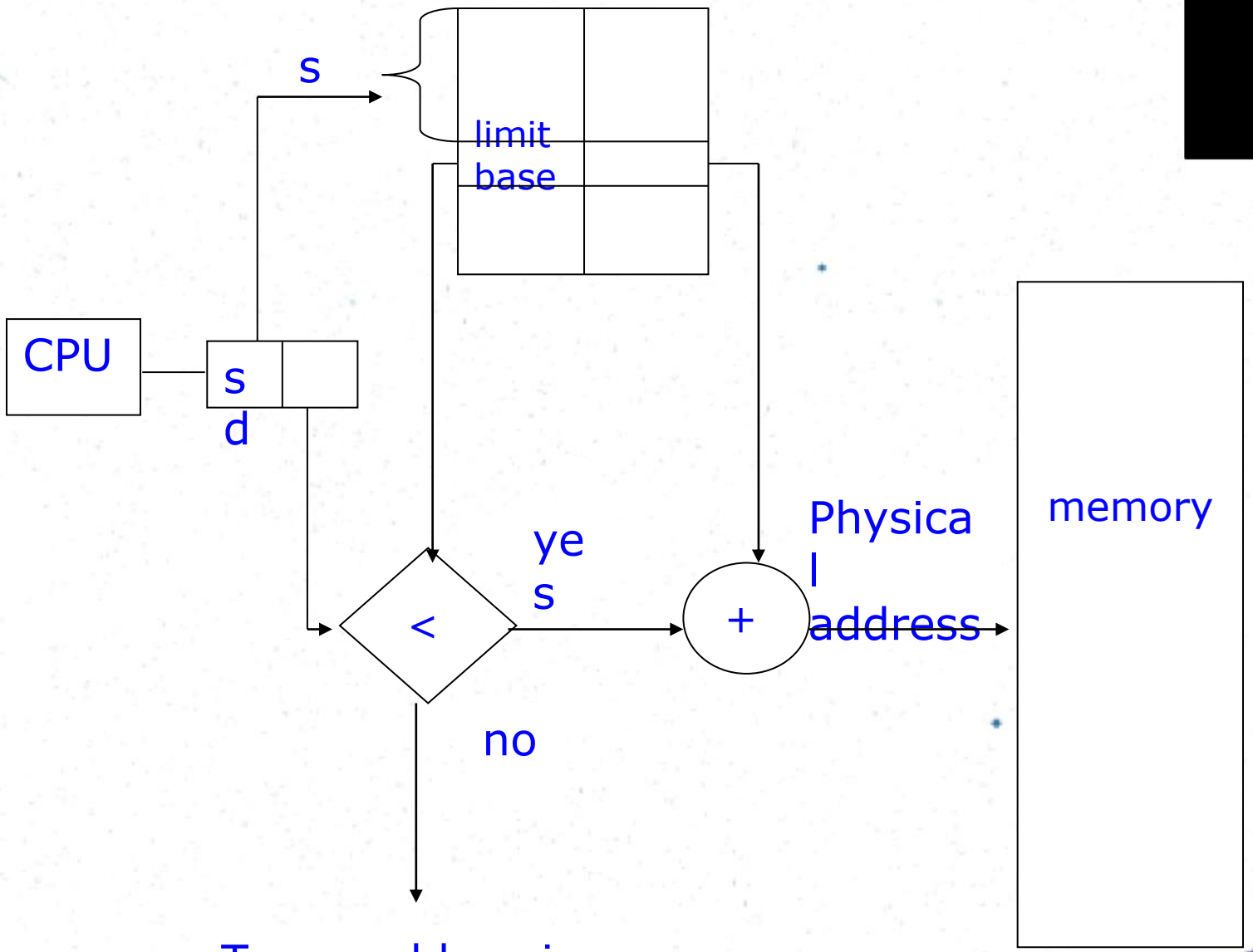
- Logical address consists of a two tuple: <segment-number, offset>.
- *Segment table* – maps two-dimensional user-defined addresses into one-dimensional physical addresses.
  - Each table entry has:
    - *Base* – contains the starting physical address where the segments reside in memory.
    - *Limit* – 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 the number of segments used by a program;
  - segment number *s* is legal if *s* < STLR.

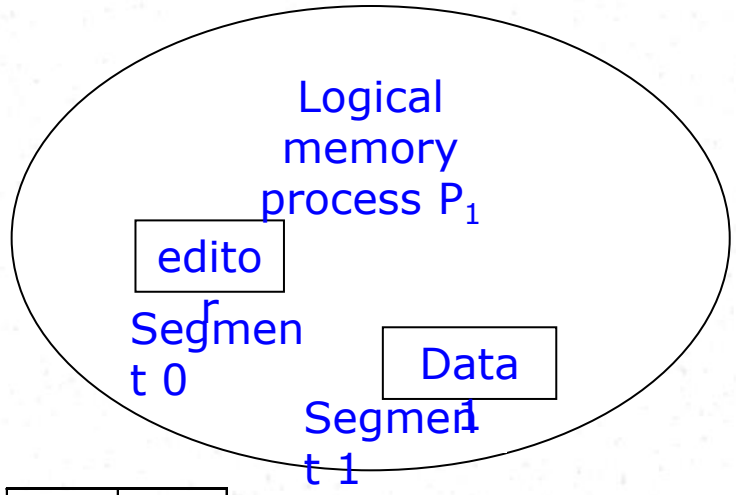
Segmentation architecture (cont.)

- Sharing.
  - Shared segments.
  - Same segment number.
- Allocation
  - First fit/best fit.
  - External fragmentation.
- Protection

  With each entry in segment table associate:
  - Validation bit = 0 → illegal segment.
  - Read/write/execute privileges.
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

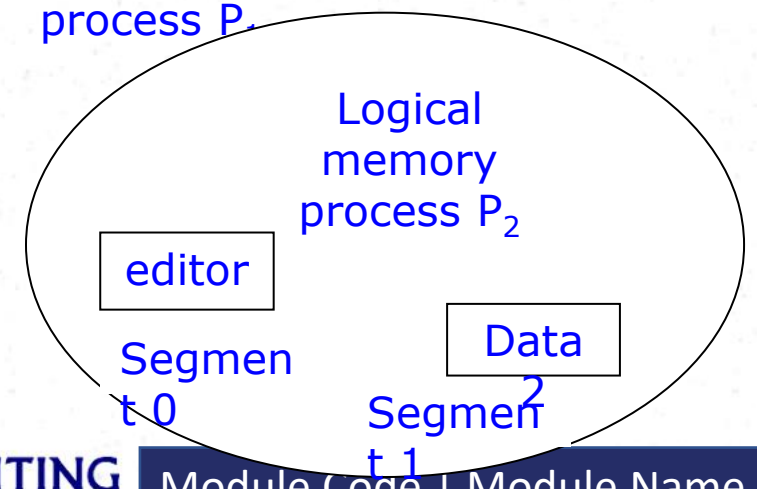
Segmentation Hardware



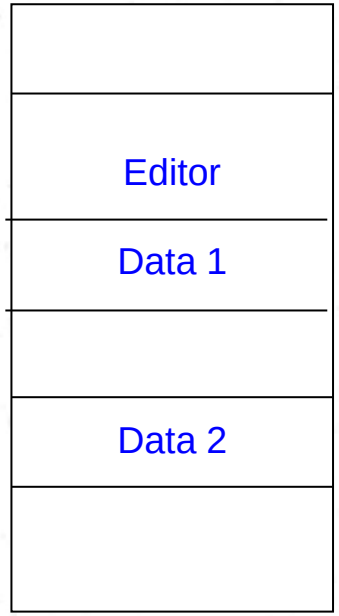


	limit	base
0	25286	43062
1	4425	68348

Segment table process P_1



43062
68348
72773
90003
98553



Physical memory

	limit	base
0	25286	43062
1	8850	90003

Segment table process P_2