Memory Management

Introduction

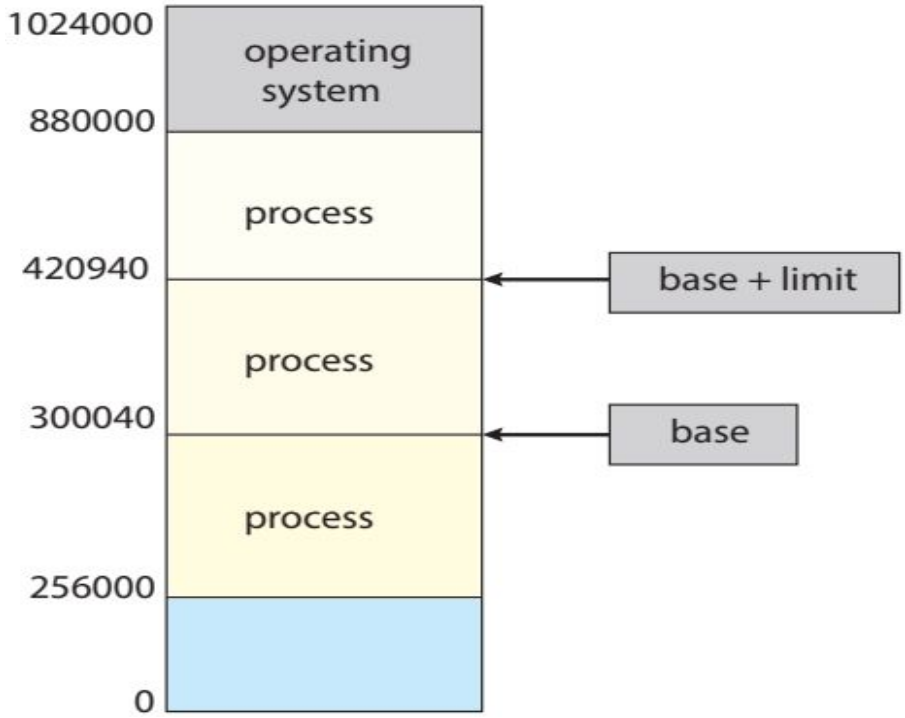
- A typical instruction-execution cycle,
- First fetches an instruction from memory.
- The instruction is then decoded
- Read of an Effective Address (operands to be fetched from memory).
- Executing of Instruction and then the instruction has been executed on the operands, results may be stored back in memory.
- Program (set of instructions) is permanently kept on **backing store** (disk)
- For a program to be run it must be brought from backing store into memory and placed within a process

Introduction

- Main memory and registers are the only storage devices the 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
- **Cache** sits between main memory and CPU registers
- **Protection** of memory is required to ensure correct operation.

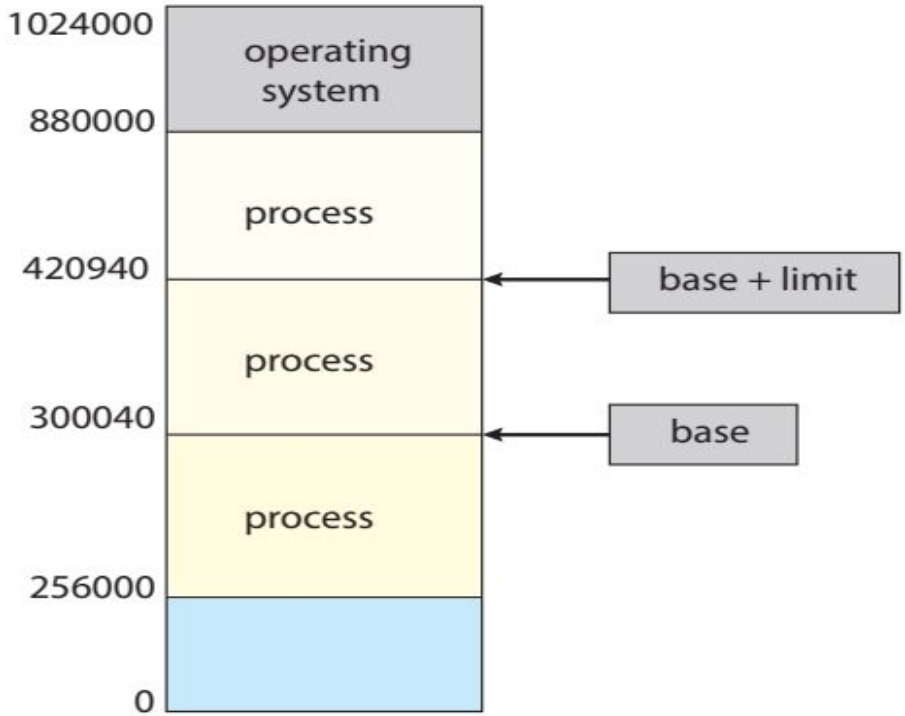
Introduction

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



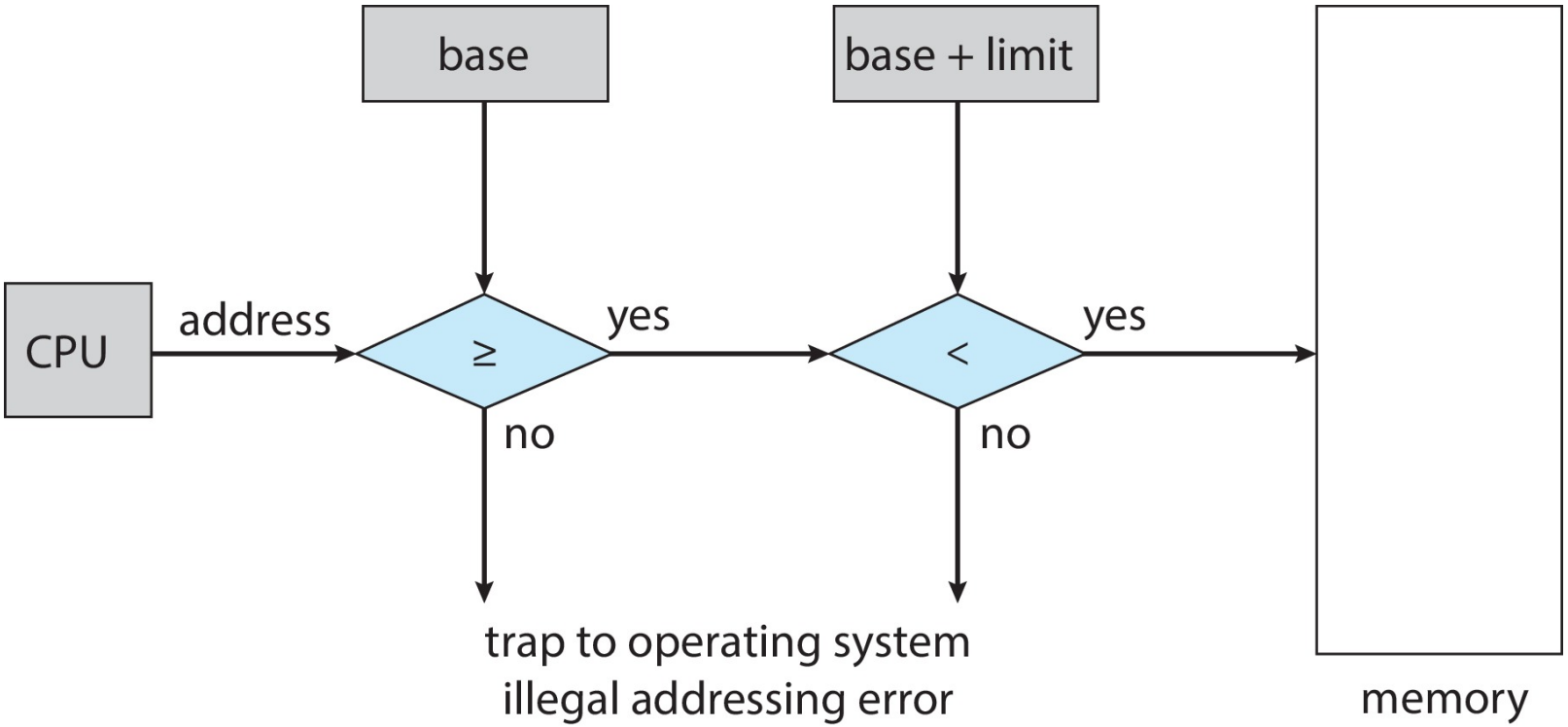
Introduction

- The **base register** holds the **smallest legal physical memory address**.
- The **limit register** specifies the **size of the range**.
- For example, if the base register holds 300040 and the limit register is 120900
- Then the program can legally access all addresses from 300040 through 420939 (inclusive).



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, are placed in an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
- Addresses represented in different ways at different stages of a program's life

✔ Source code addresses are usually symbolic.

✔ Compiled code addresses **bind** to relocatable addresses

✔ Linker or loader will bind relocatable addresses to absolute addresses

✔ Each binding maps one address space to another.

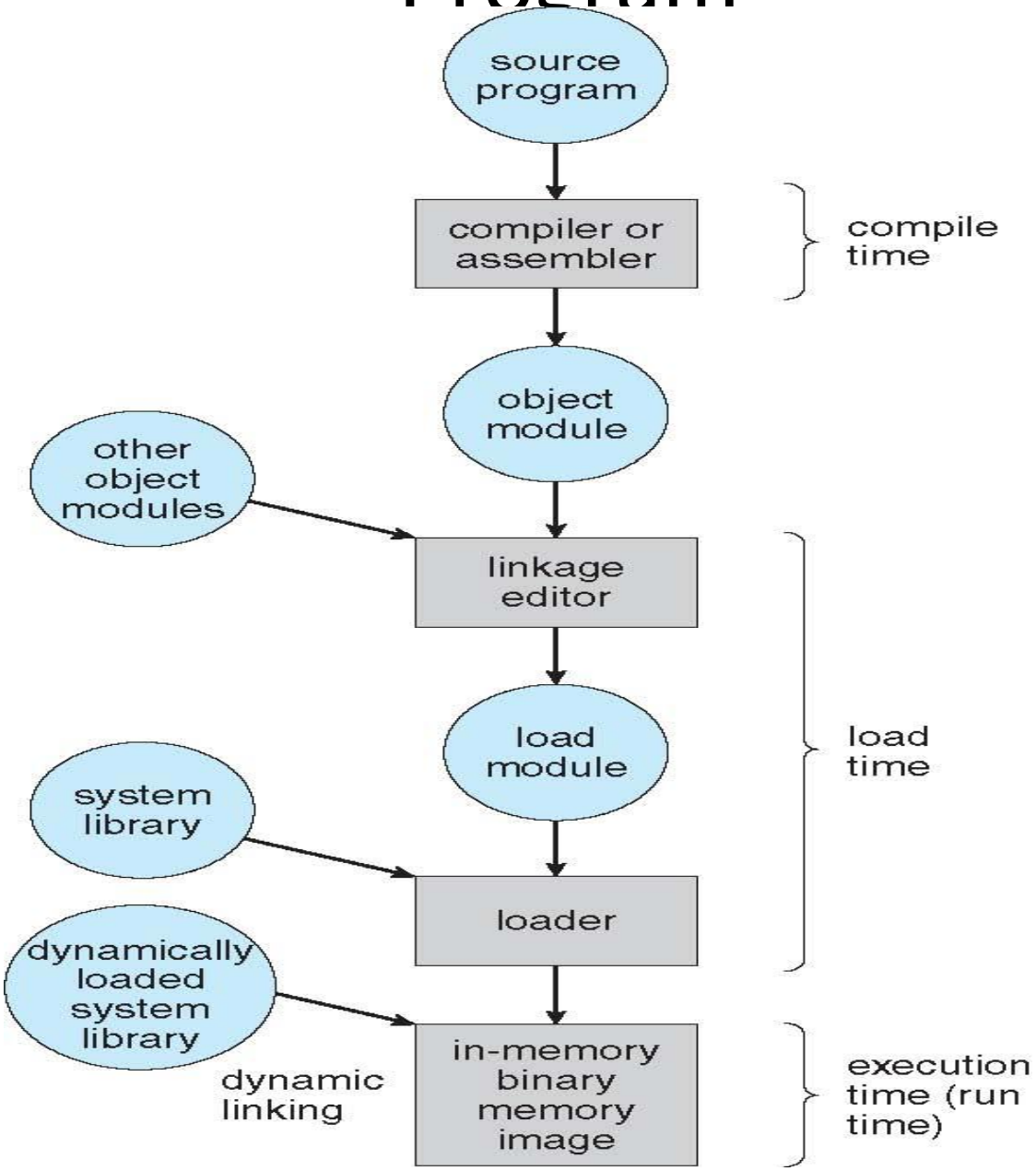
Binding of Instructions and Data to Memory

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

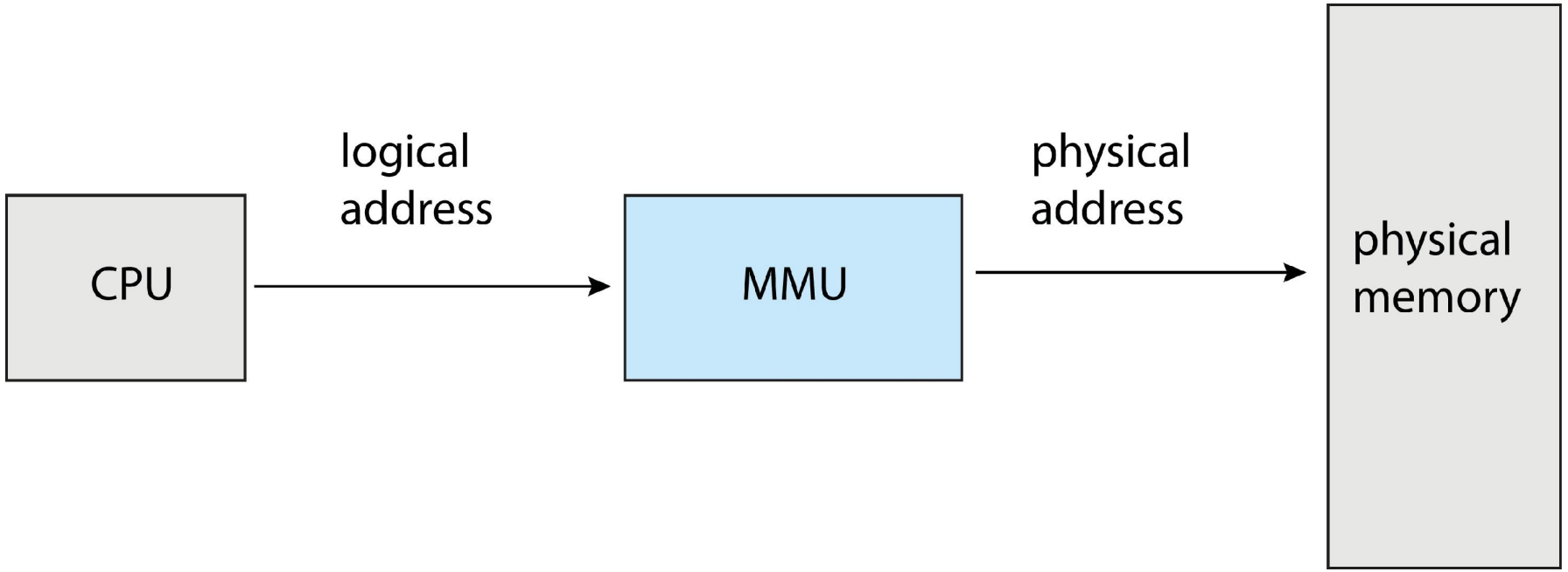


Logical vs. Physical Address Space

- 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

Memory-Management Unit (MMU)

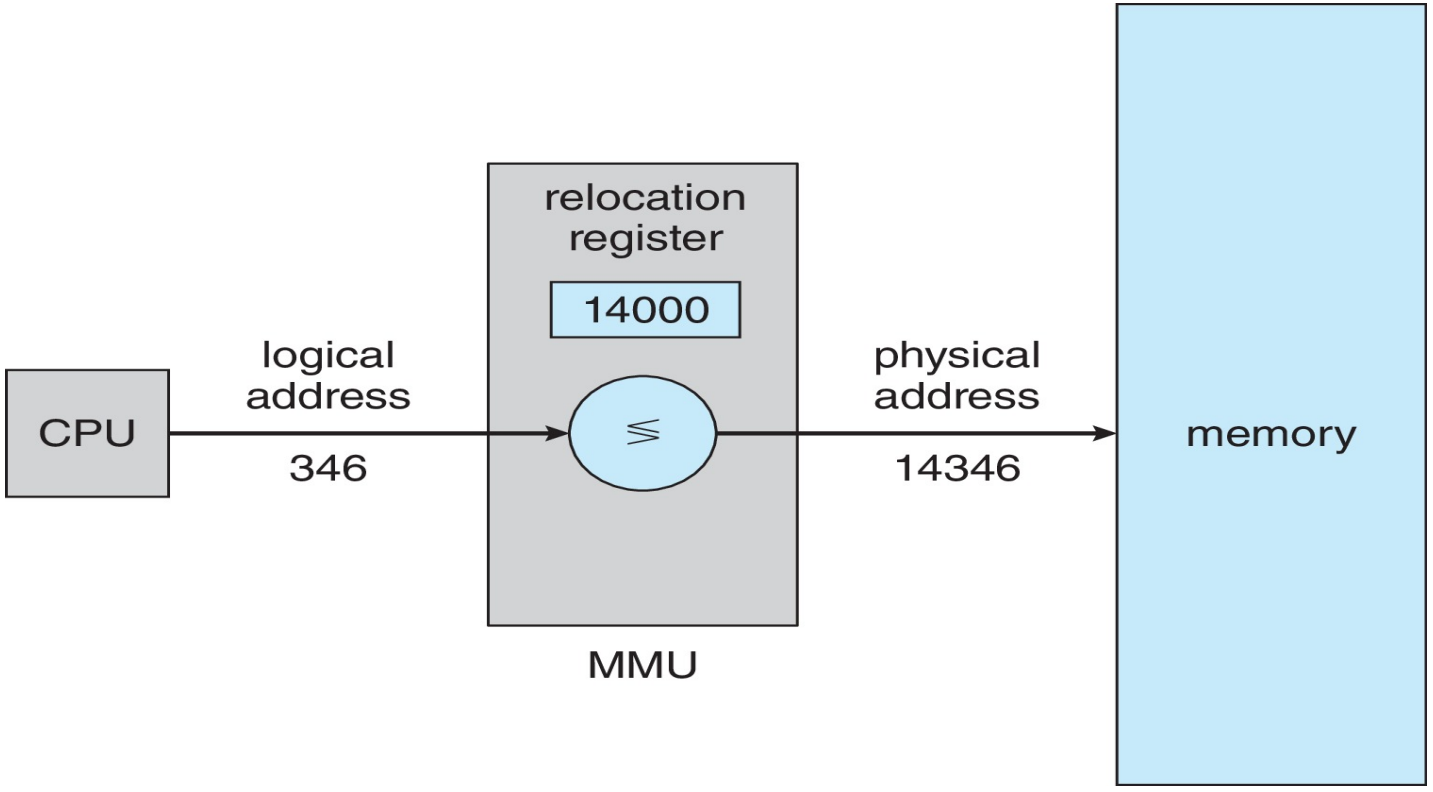
- Hardware device that at **run-time** maps logical address to physical address.



- Many methods possible, covered in the rest of this ppt.

Relocation Register

- 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.



Relocation Register

- 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

Dynamic Loading

- To obtain better memory-space utilization, we can use **dynamic loading**.
- The program consist of main part and a number of routines.
- With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.

Dynamic Loading

- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, the relocatable linking loader is called to load the desired routine into memory.
- Then control is passed to the newly loaded routine.
- The advantage of dynamic loading is that a routine is **loaded only when it is needed**.

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.
- Dynamic linking, in contrast, is similar to dynamic loading.
- Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run.
- DLLs are also known as shared libraries.
- DLLs libraries can be shared among multiple processes, so that only one instance of the DLL in main memory.

Memory Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method.
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.
- Main memory usually into two **partitions**:
  - ✔ one for the operating system, and
  - ✔ one for the user processes.
- We can place the operating system in either low memory addresses or high memory addresses.

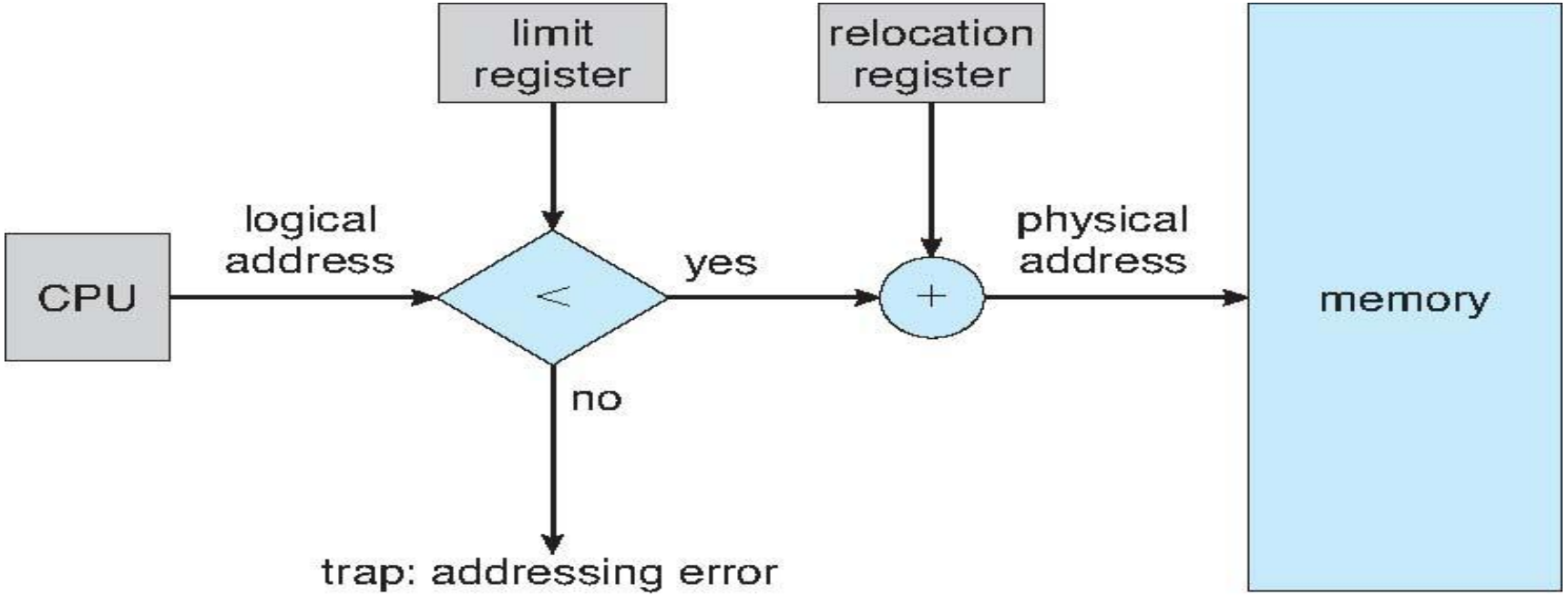
Contiguous Allocation

- 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**

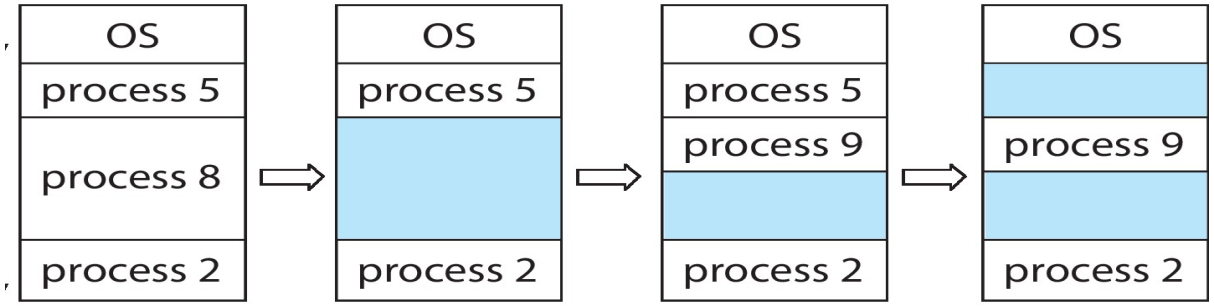


Memory Allocation

- Degree of multiprogramming limited by number of partitions
- **Fixed-size partition**
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- Each partition may contain exactly one process.
- **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)

Variable Partition Allocation

- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts.
- One part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger 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

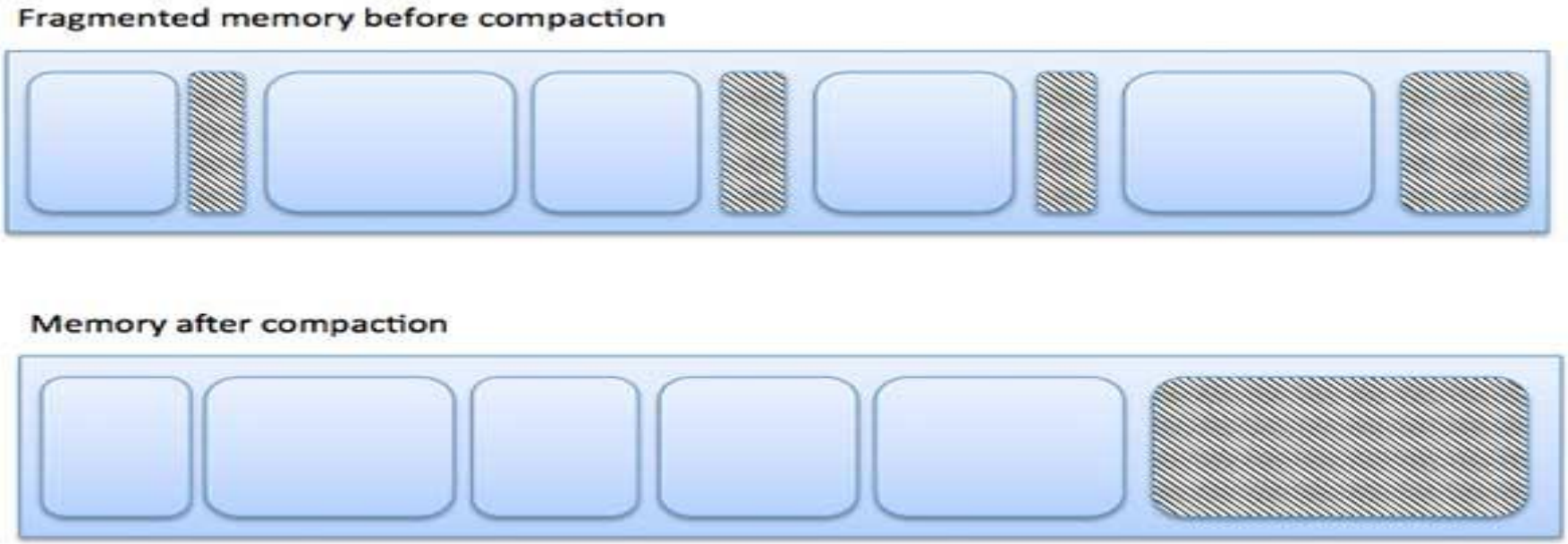
Fragmentation

- **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

Internal fragmentation	External fragmentation
In internal fragmentation fixed-sized memory, blocks square measure appointed to process.	In external fragmentation, variable-sized memory blocks square measure appointed to method.
The solution of internal fragmentation is best-fit block.	Solution of external fragmentation is compaction, paging and segmentation.
Internal fragmentation occurs when memory is divided into fixed sized partitions.	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, is called External fragmentation .

Fragmentation

- 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
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.



Fragmentation

- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another 0.5 N blocks will be lost to fragmentation.
- That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.
- Now consider that backing store has same fragmentation problems

Fragmentation

- Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.
- This is the strategy used in **paging**, the most common memory-management technique for computer systems.

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

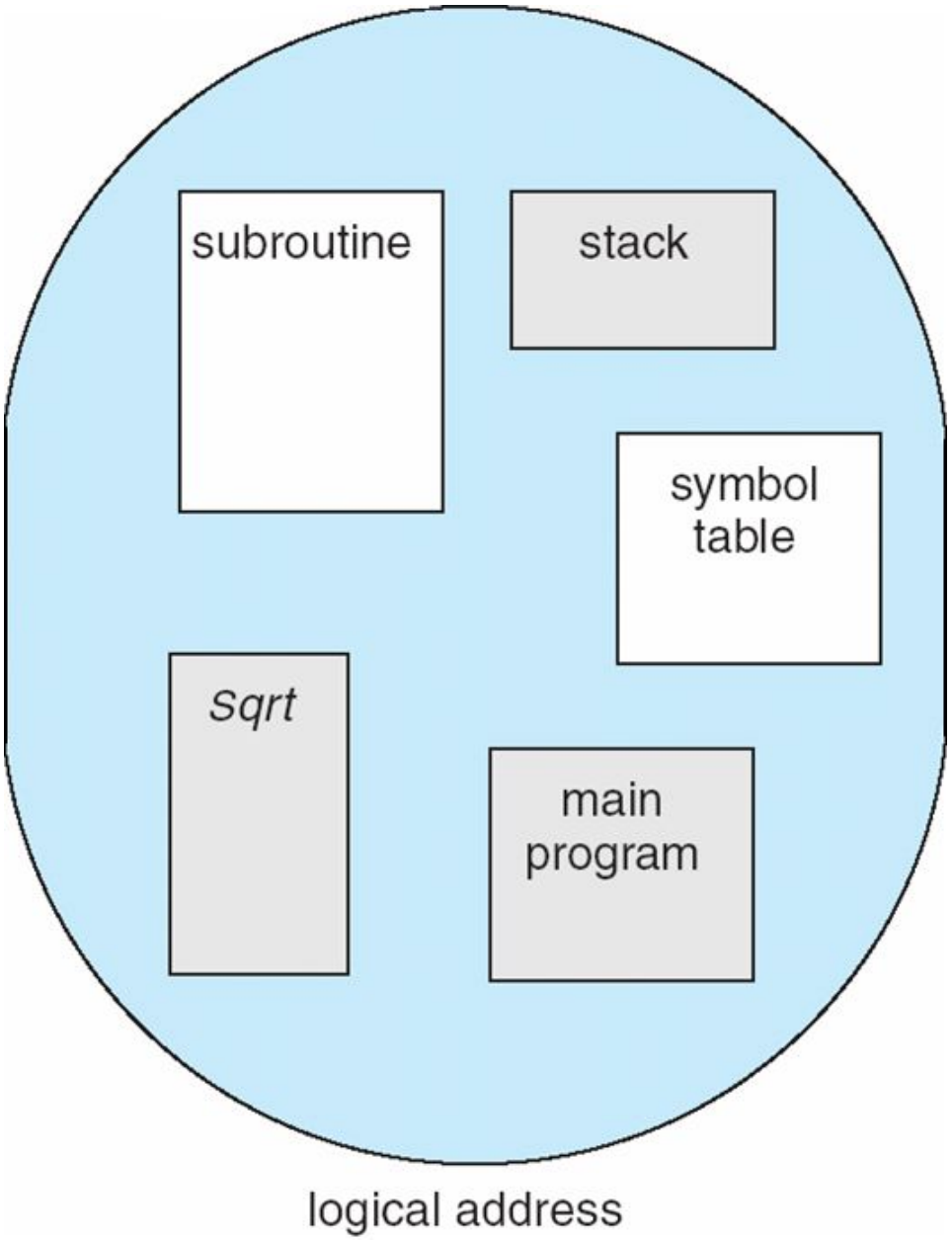
  common block

  stack

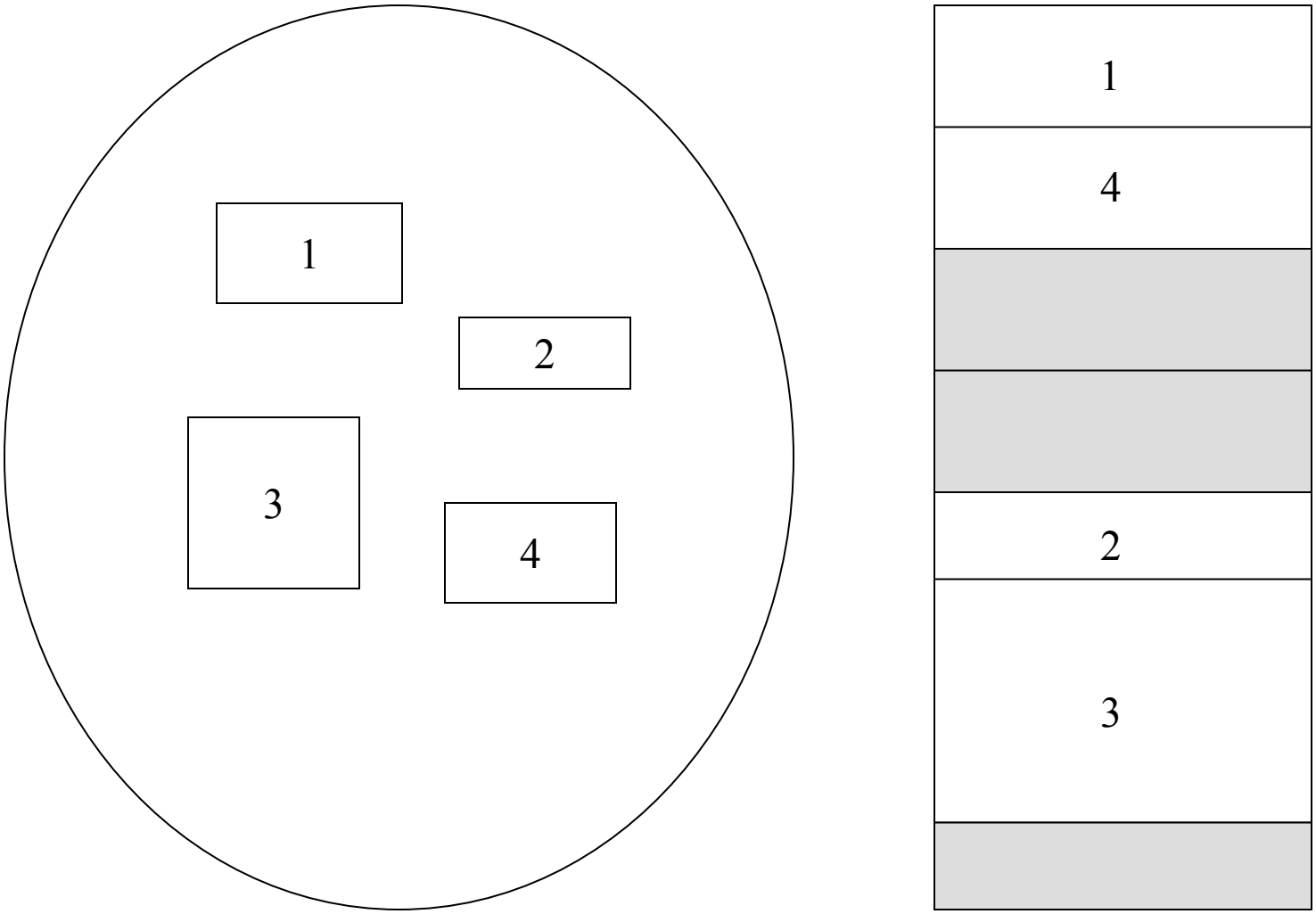
  symbol table

  arrays

User's View of a Program



Logical View of Segmentation



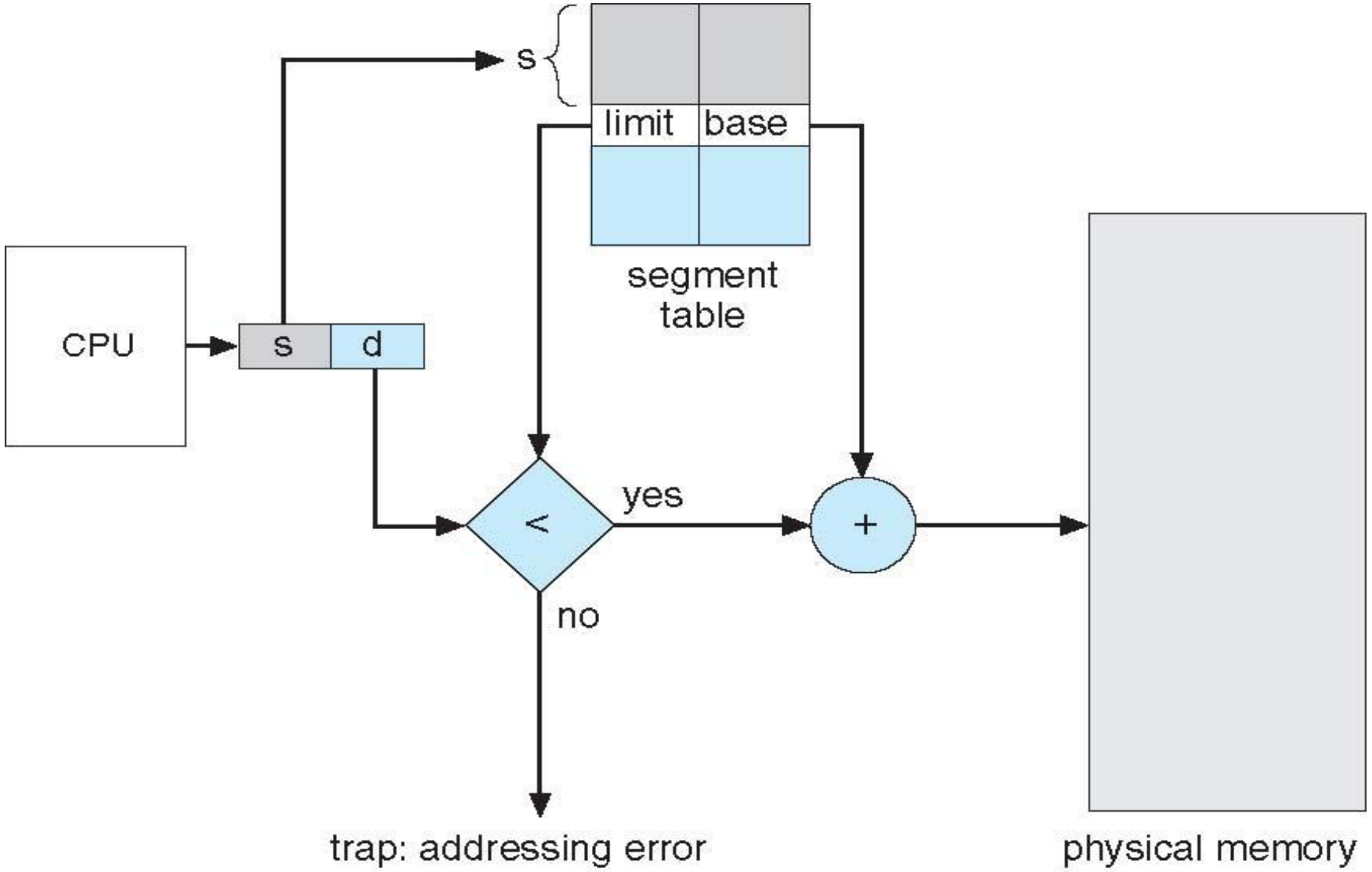
Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register** (**STBR**) points to the segment table's location in memory
- **Segment-table length register** (**STLR**) indicates number of segments used by a program
  - Segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Hardware

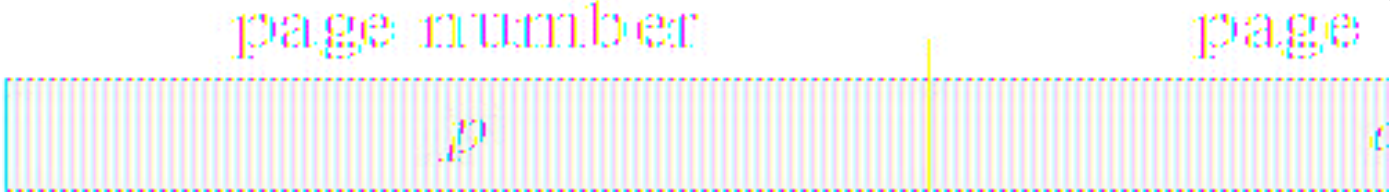


Paging

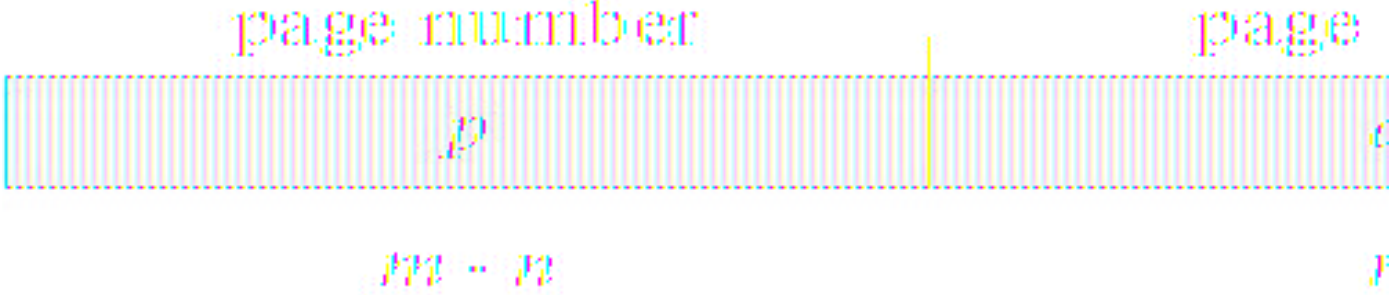
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - ✔Avoids external fragmentation
  - ✔Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory 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

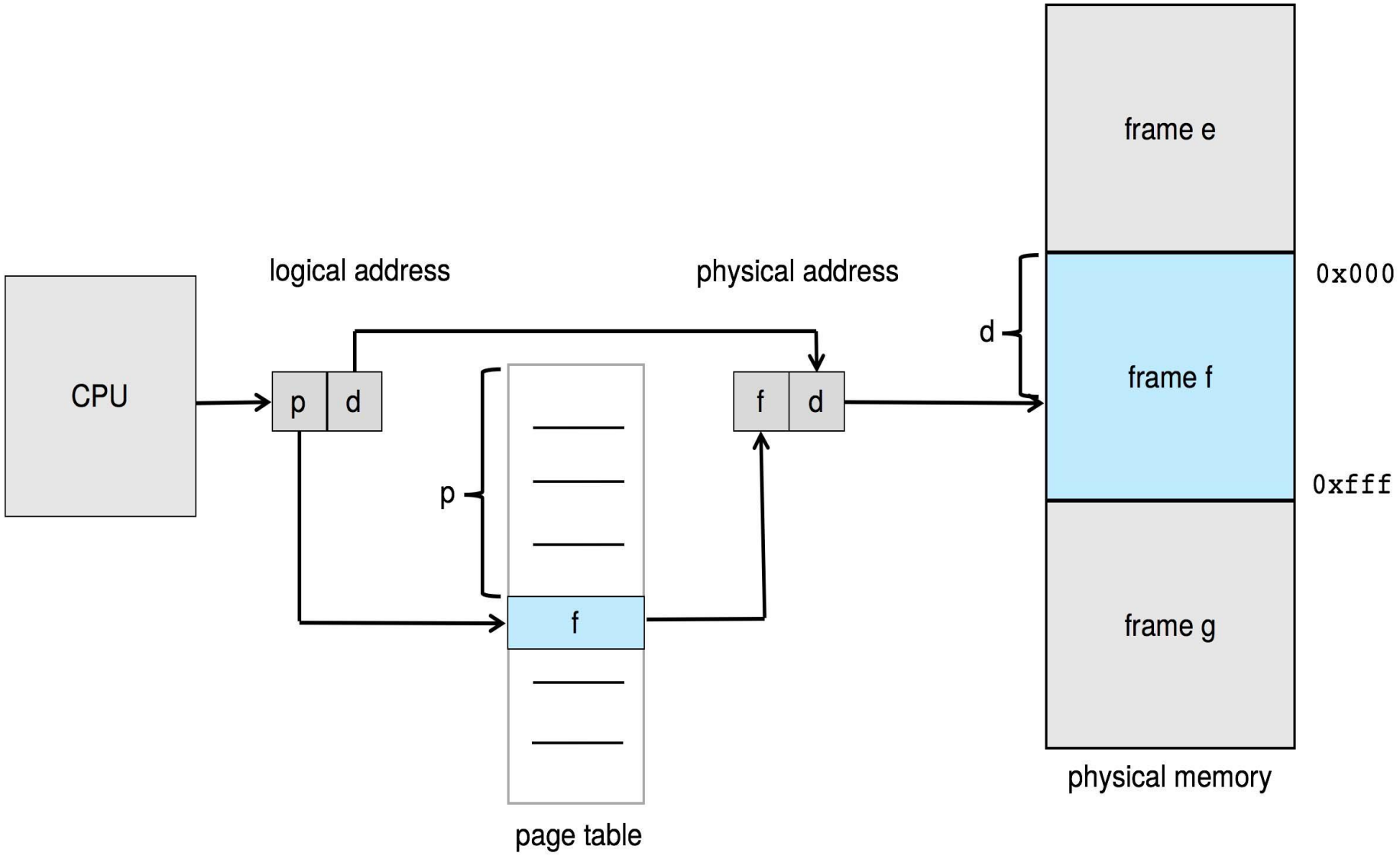
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



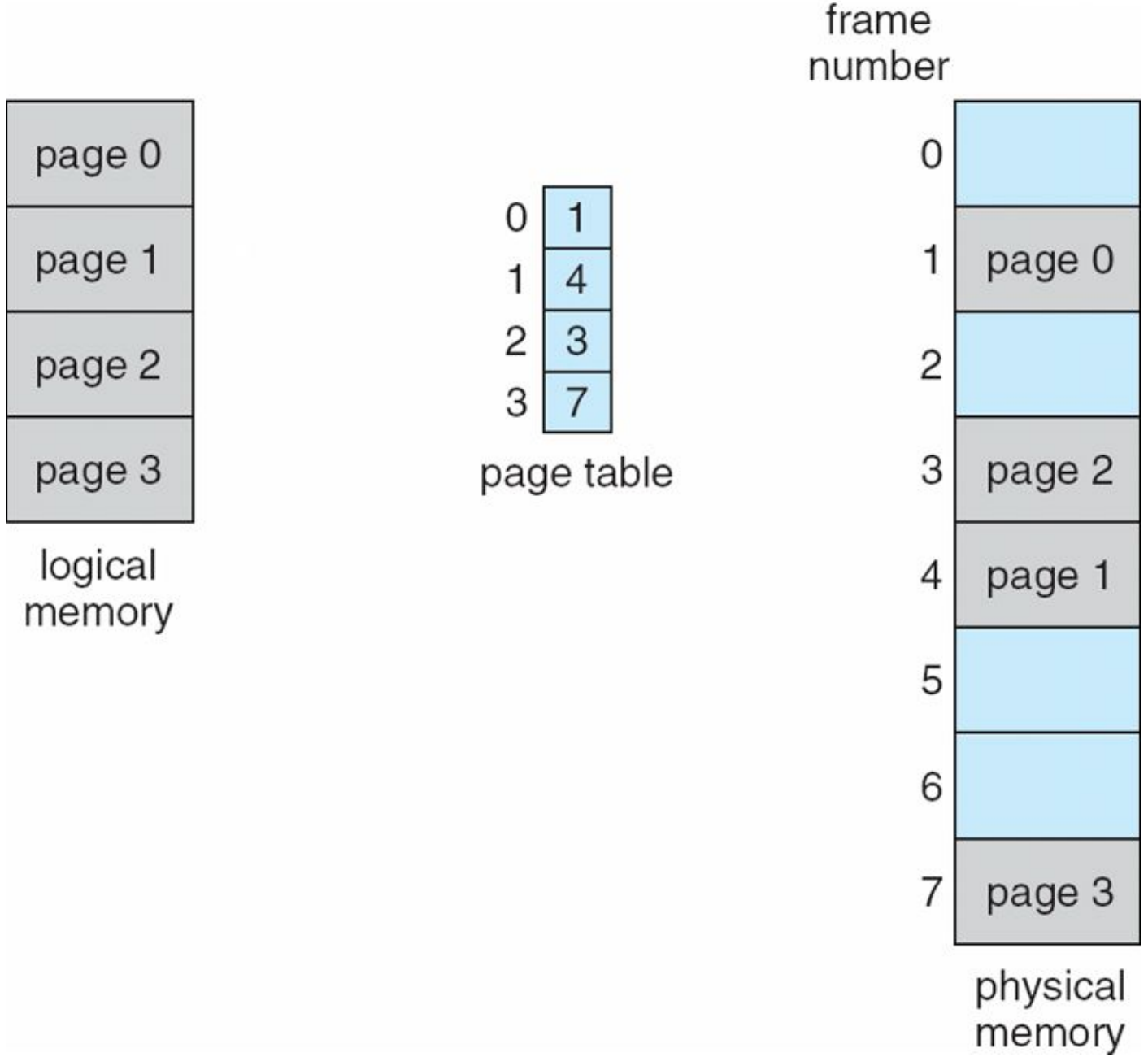
- For given logical address space 2^m and page size 2^n



Paging Hardware

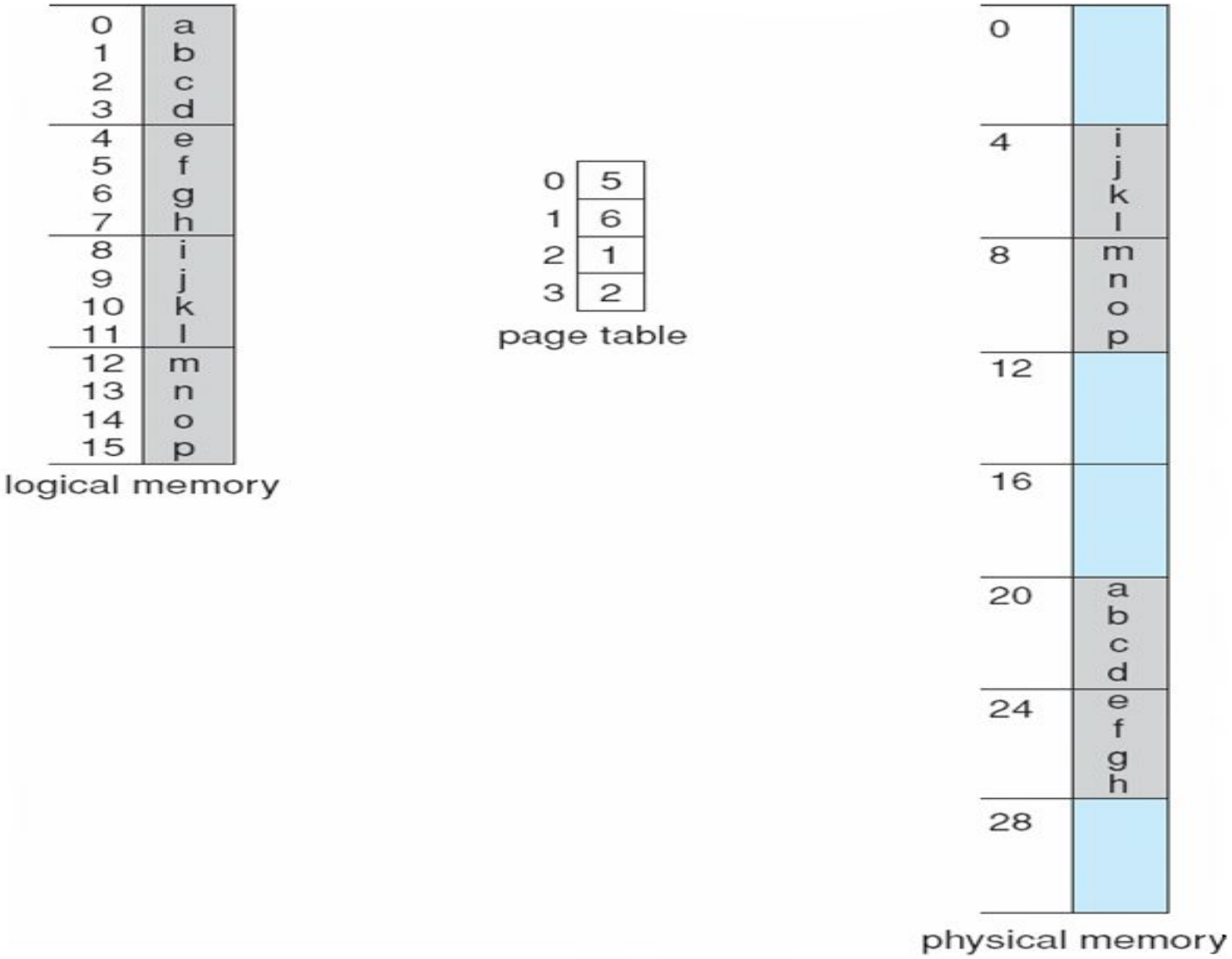


Paging Model of Logical & Physical Memory



Paging Example

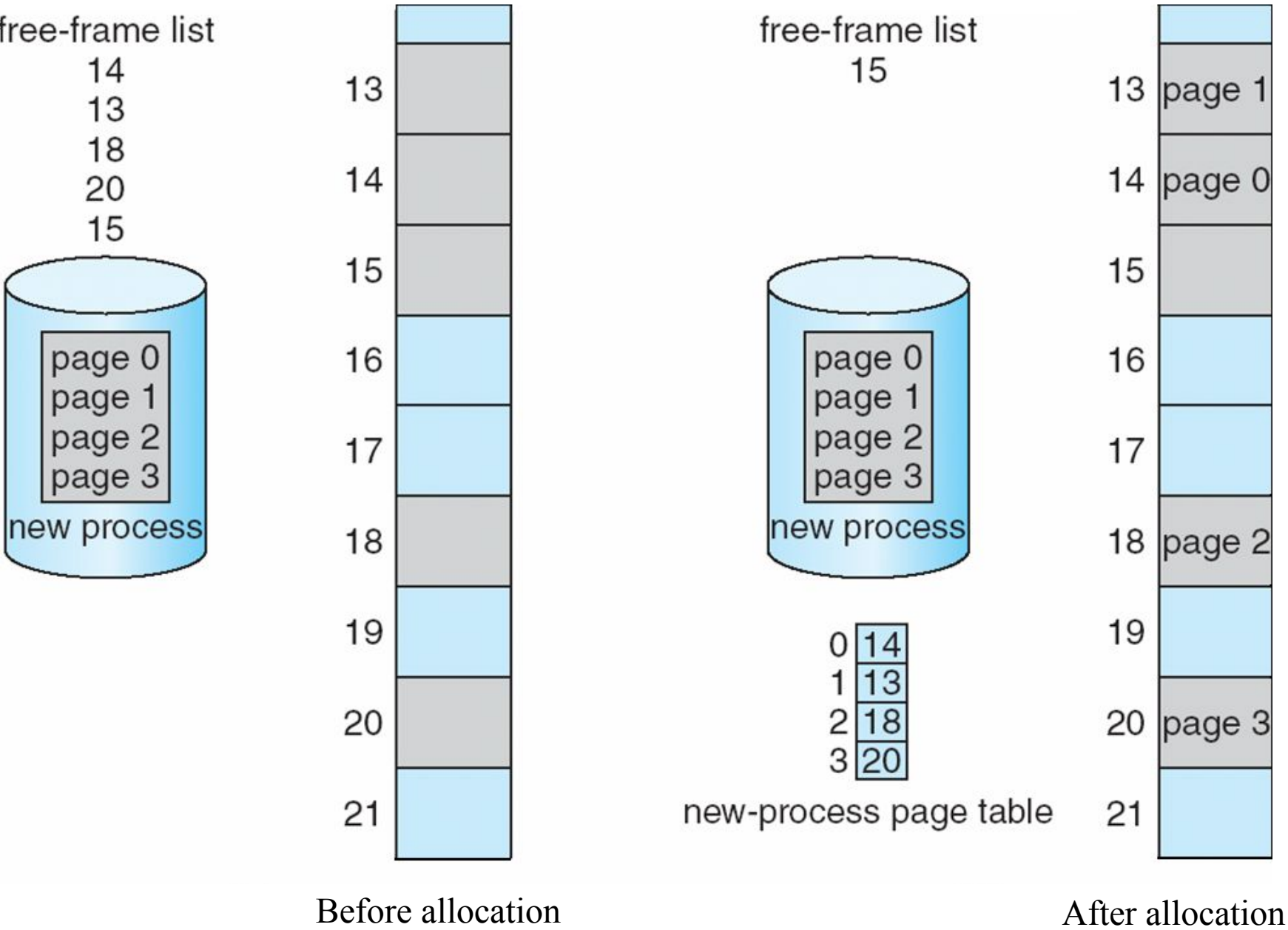
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



Paging -- 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
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

After allocation

Implementation of Page Table

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

- 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
- 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 the TLB at every context switch

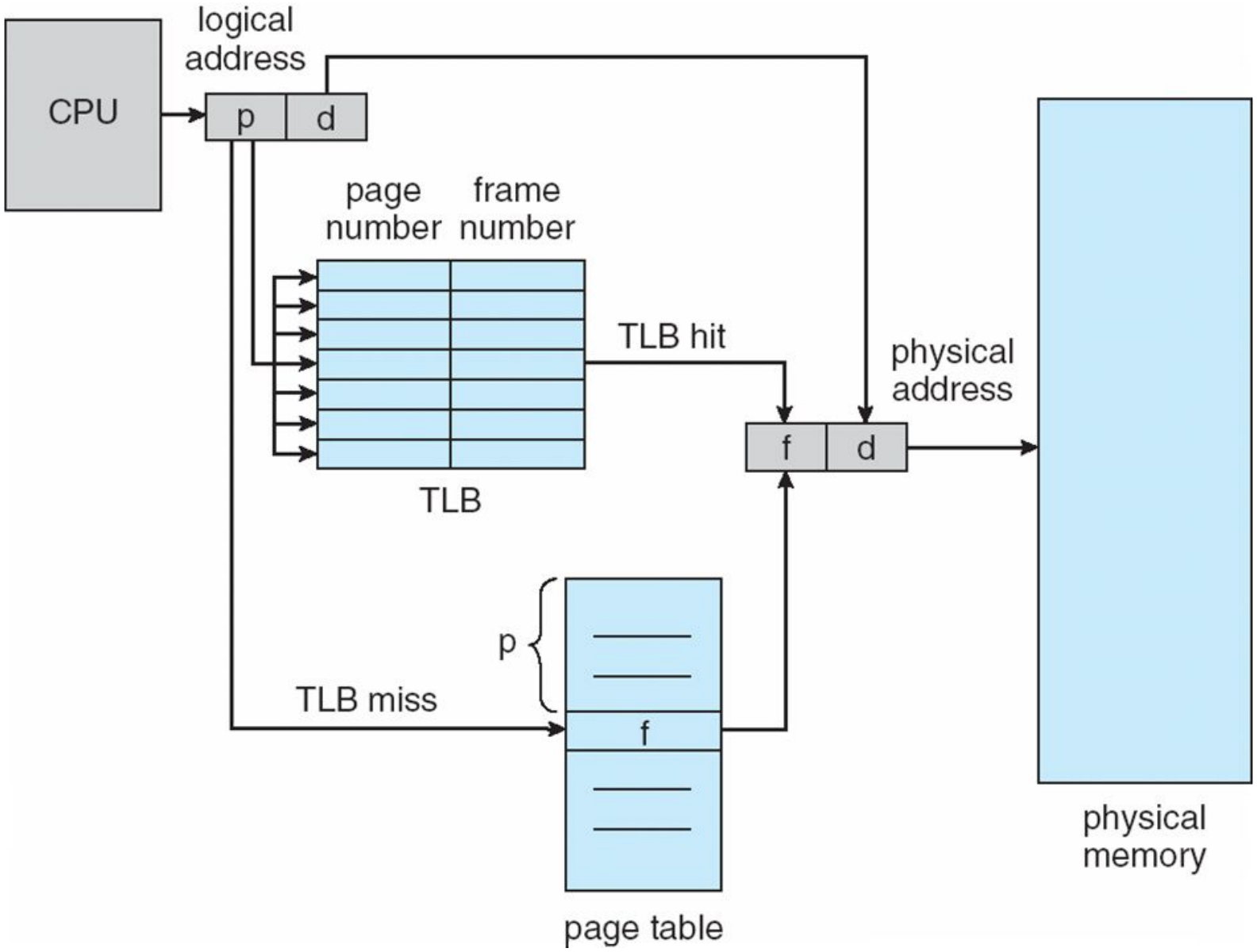
Hardware

- 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

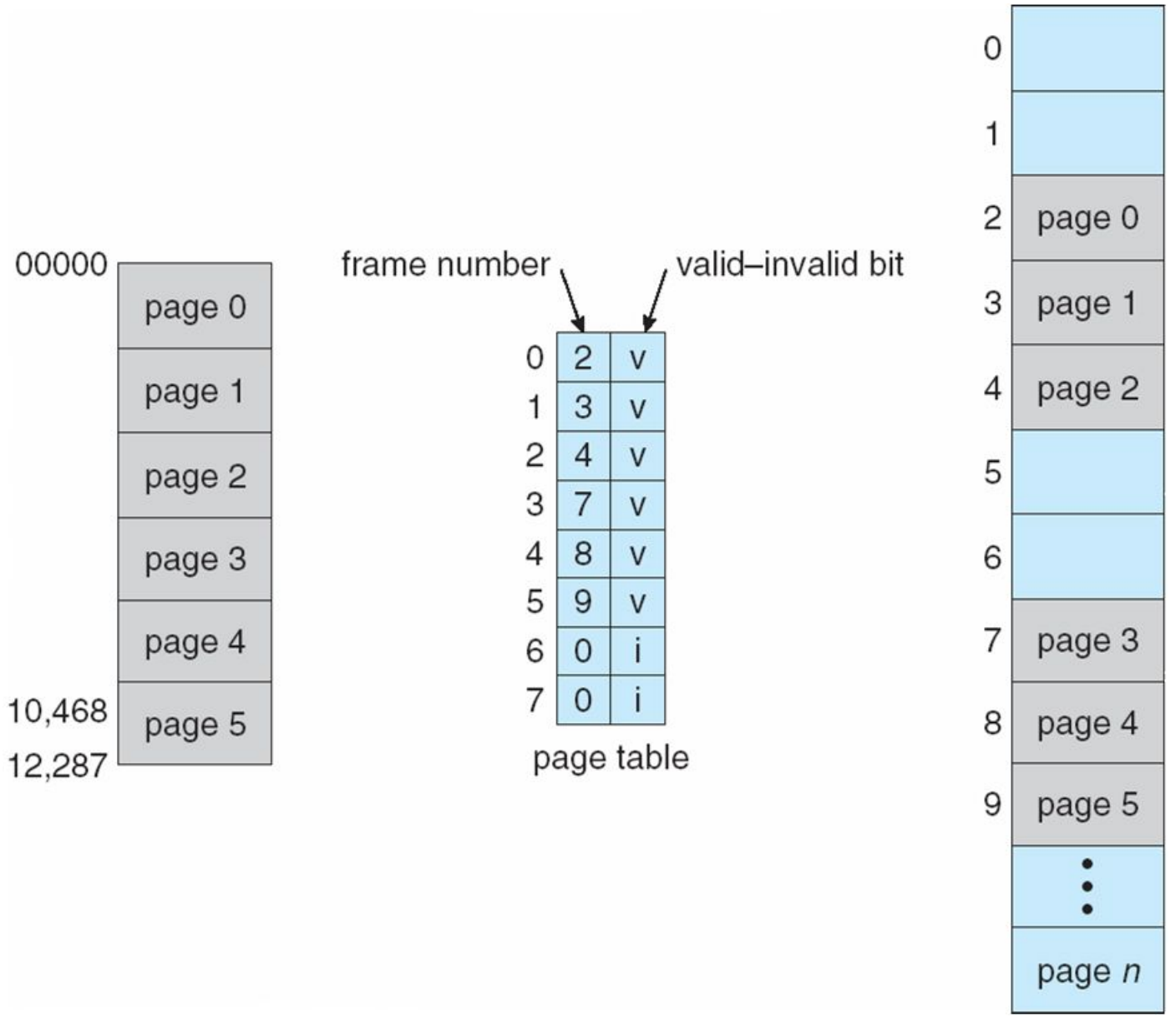
Paging Hardware With TLB



Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if access is allowed
- **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
- Can also add more bits to indicate if read-only, read-write, execute-only is allowed.

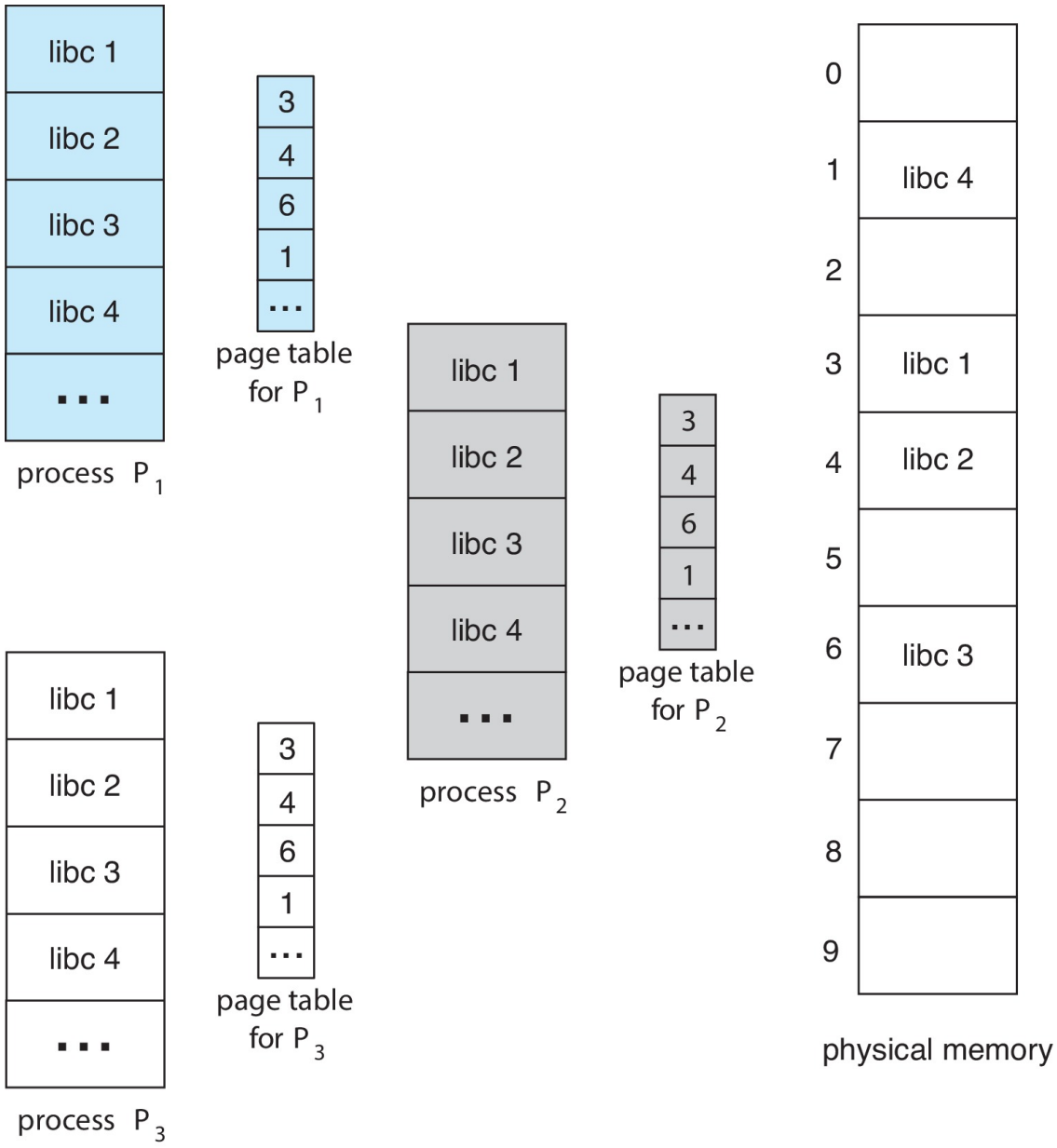
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - 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

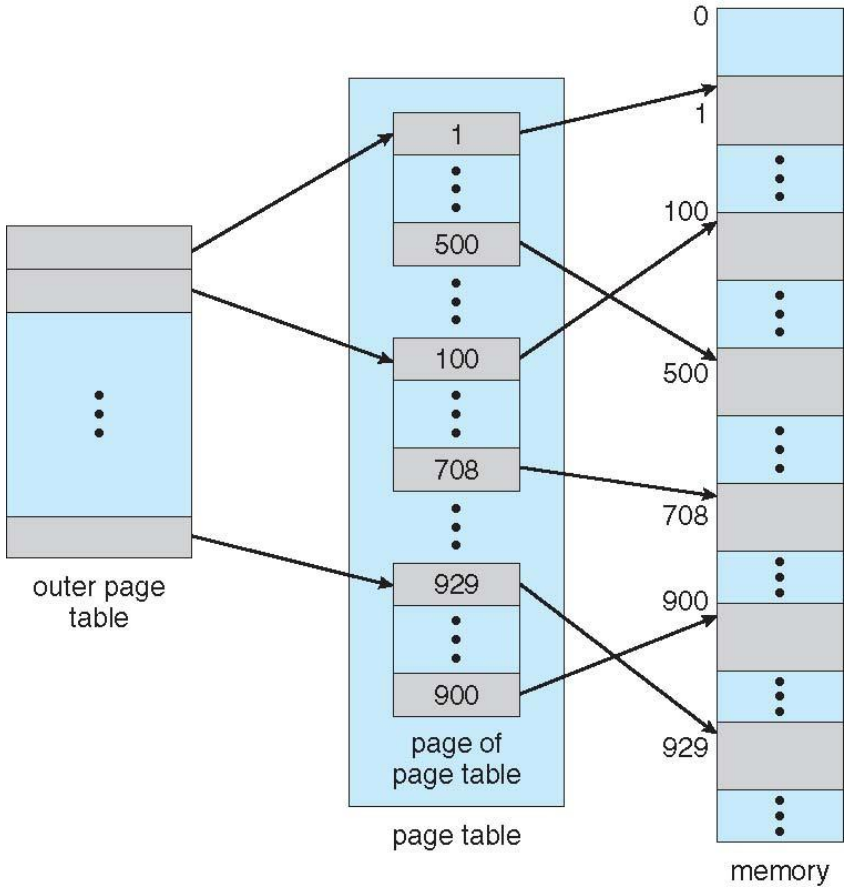


Structure of the Page Table

- 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 1 KB (2^{10})
  - Page table would have 1 million entries (2^{32} / 2^{10})
  - If each entry is 4 bytes each process requires 16 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

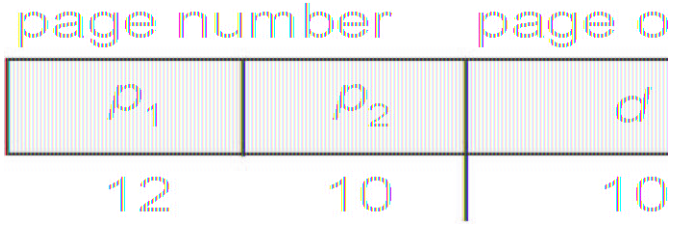
Hierarchical Page Tables

- 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



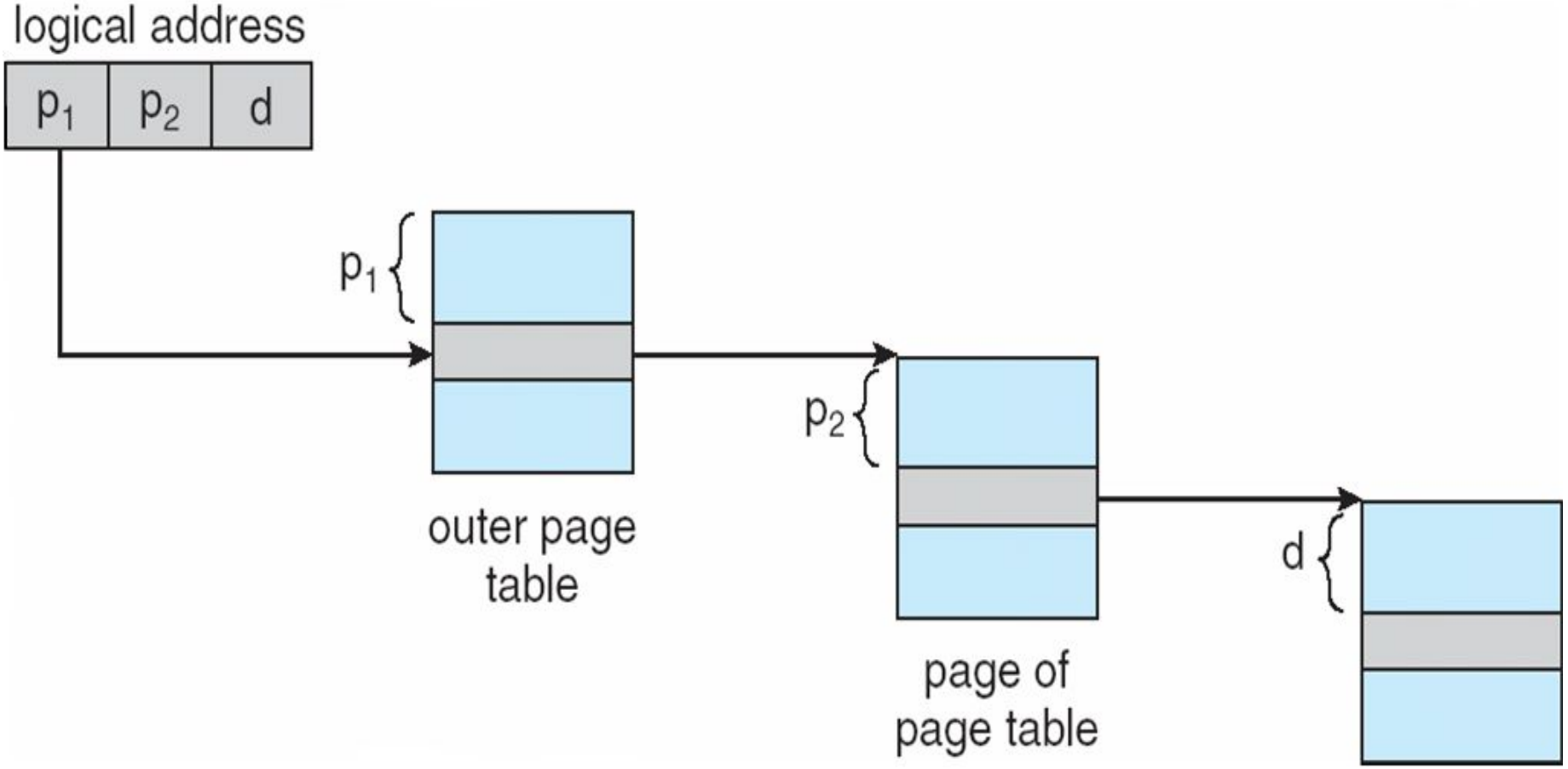
Two-Level Paging Example

- A logical address (on 32-bit machine with 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

Address-Translation Scheme



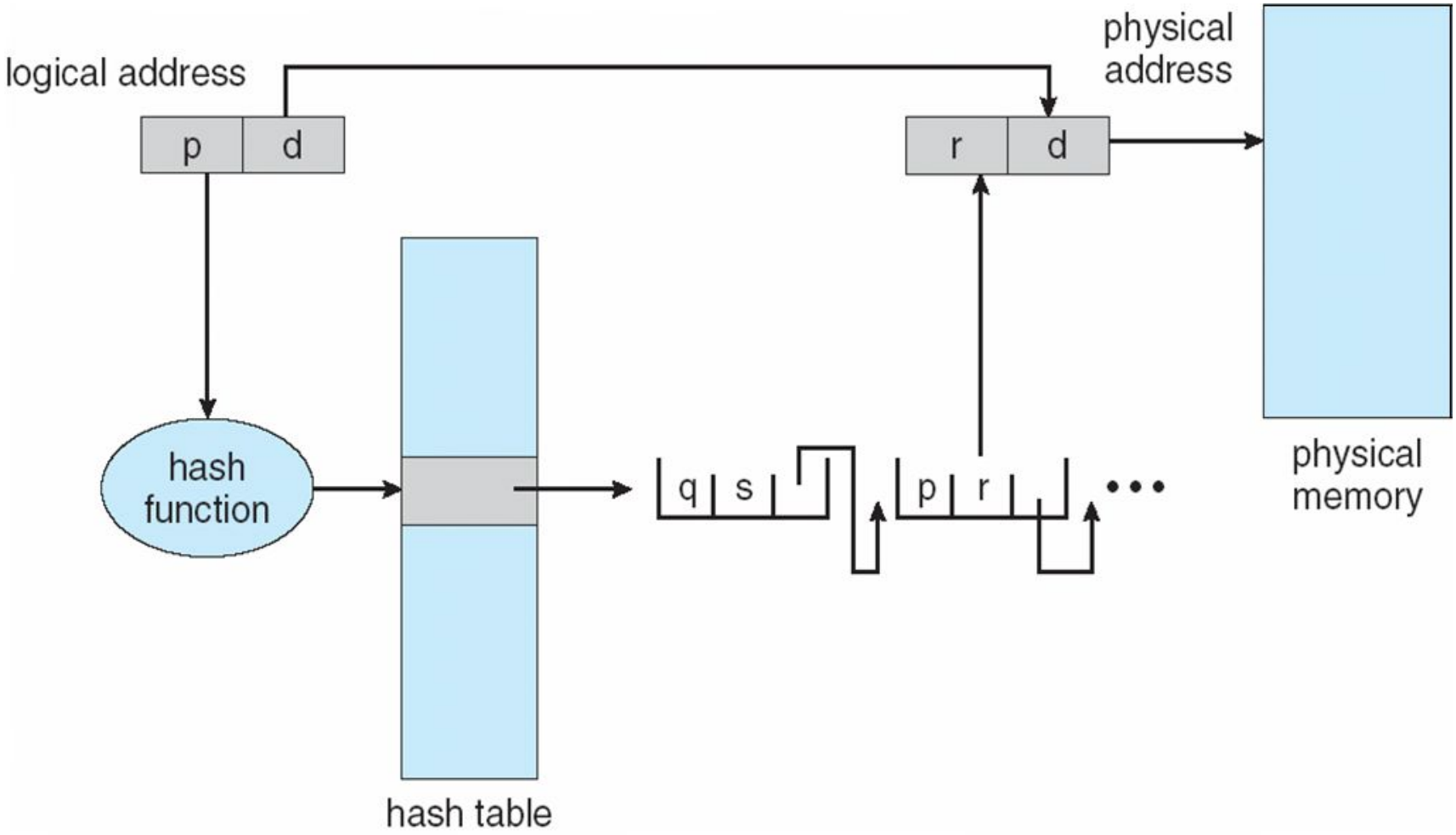
Hashed Page Tables

- Used in architecture with 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 searching for a match
  - If a match is found, the corresponding physical frame is extracted

Hashed Page Tables

- 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)

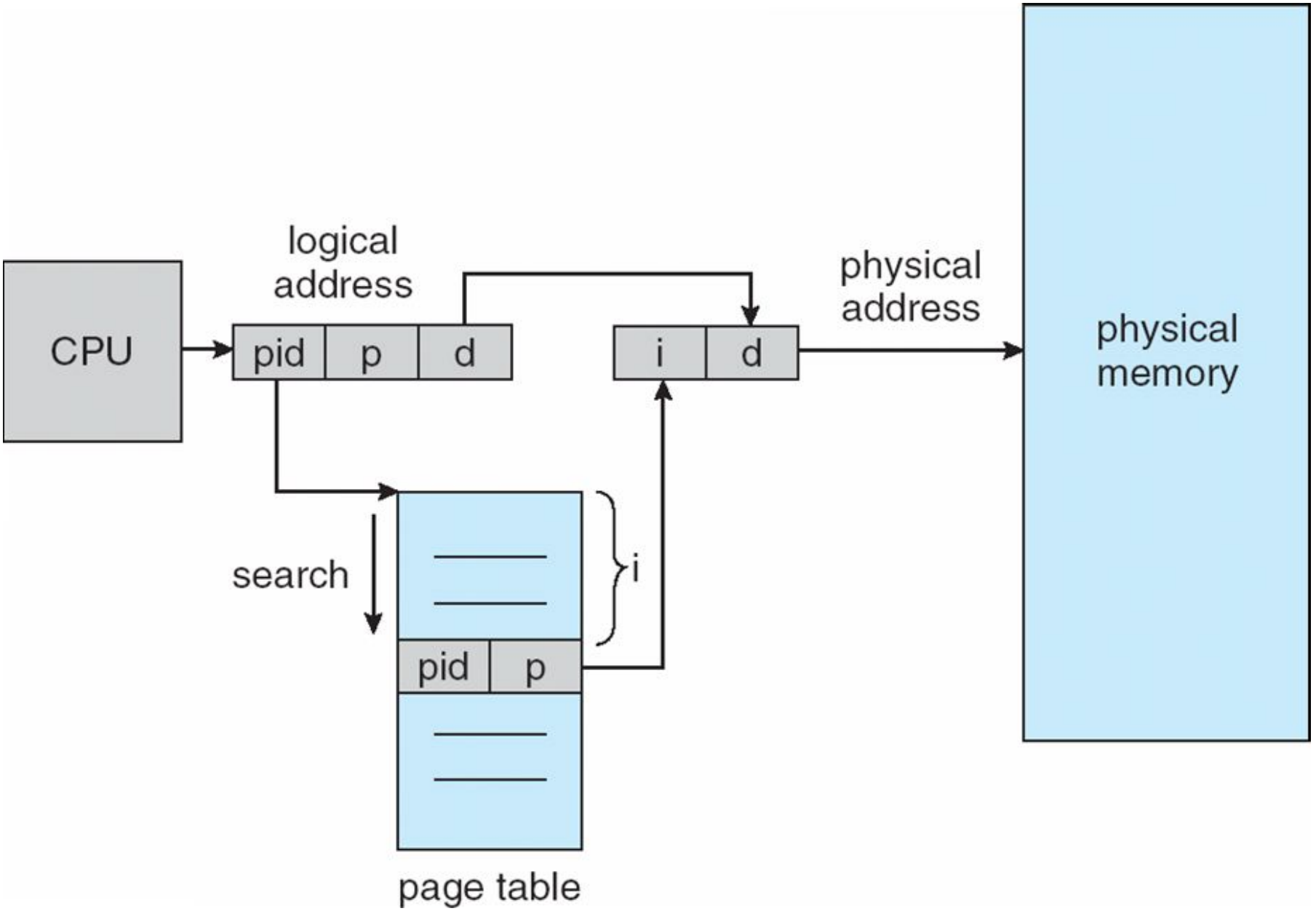
Hashed Page Table



Inverted Page Table

- Rather than having each process keep a page table & 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



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
- **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