Memory Management

References:

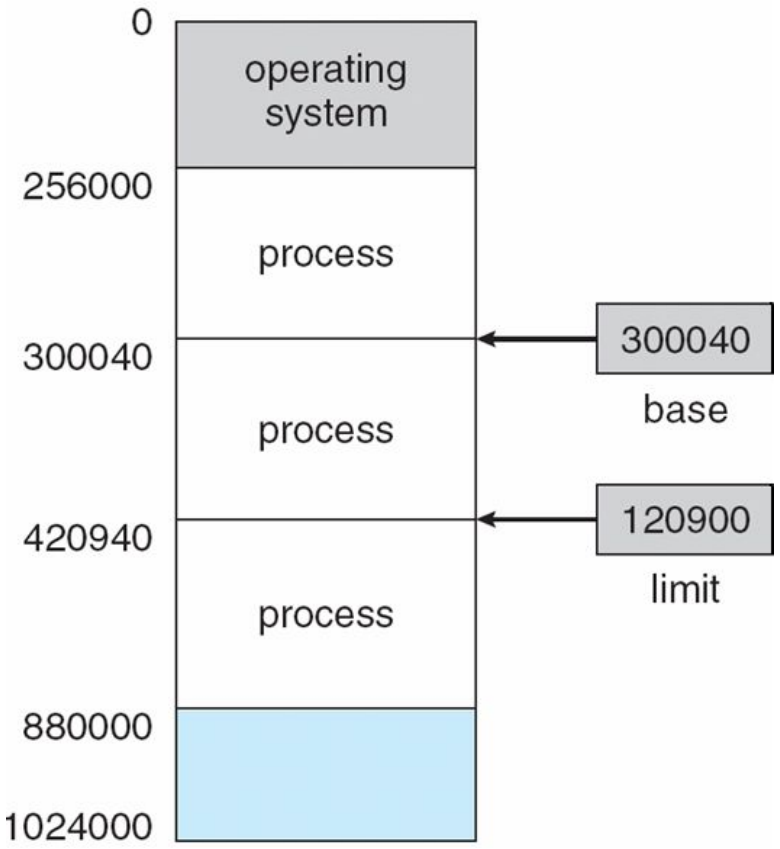
1. **Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Operating system concepts with Java*. Wiley Publishing, 2009.**
2. **Stallings, William. *Operating Systems 5th Edition*. Pearson Education India, 2006.**

Background

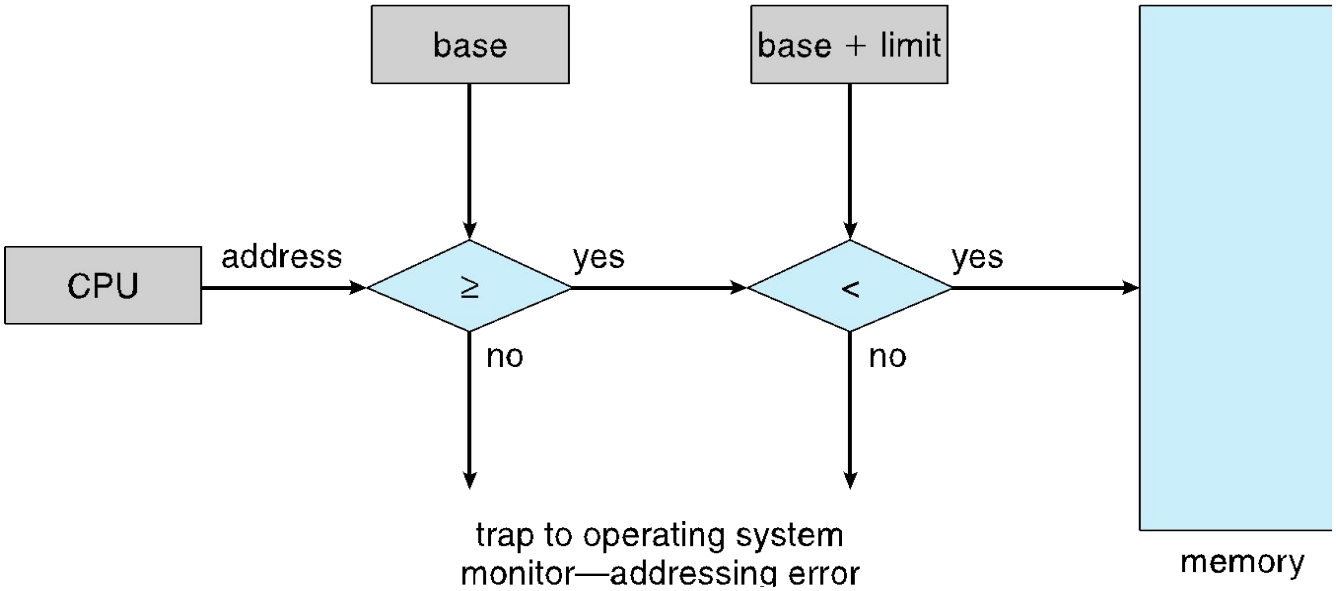
- Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.
- Multi-tasking OSes compounds the complexity of memory management, because as processes are swapped in and out of the CPU, all at high speeds and without interfering with any other processes.
- Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it.
- Register access 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

Base and Limit Registers

- User processes must be restricted and access only the memory locations that "belong" to that particular process.
- A pair of **base** and **limit registers** define the logical address space for each process.
- ***Every*** memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated.
- Changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.



Hardware Address Protection



Address Binding

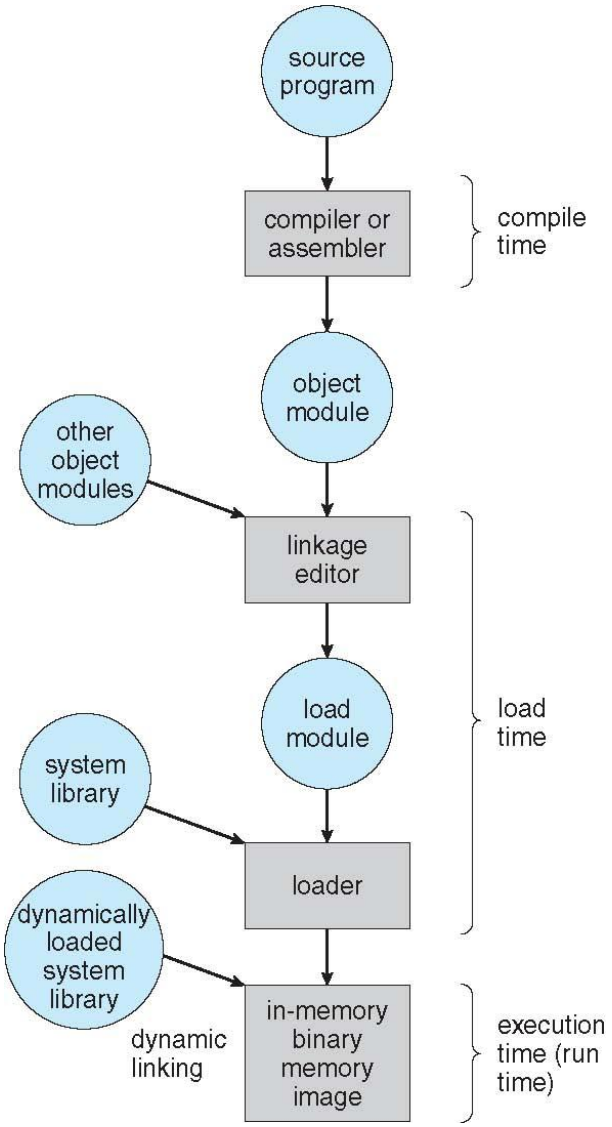
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different 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 memory addresses can happen at three different stages
  - **Compile Time** - If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses.

    However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.
  - **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  - **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.

Multistep Processing of a User Program



Addresses

Logical

- reference to a memory location independent of the current assignment of data to memory

Relative

- address is expressed as a location relative to some known point

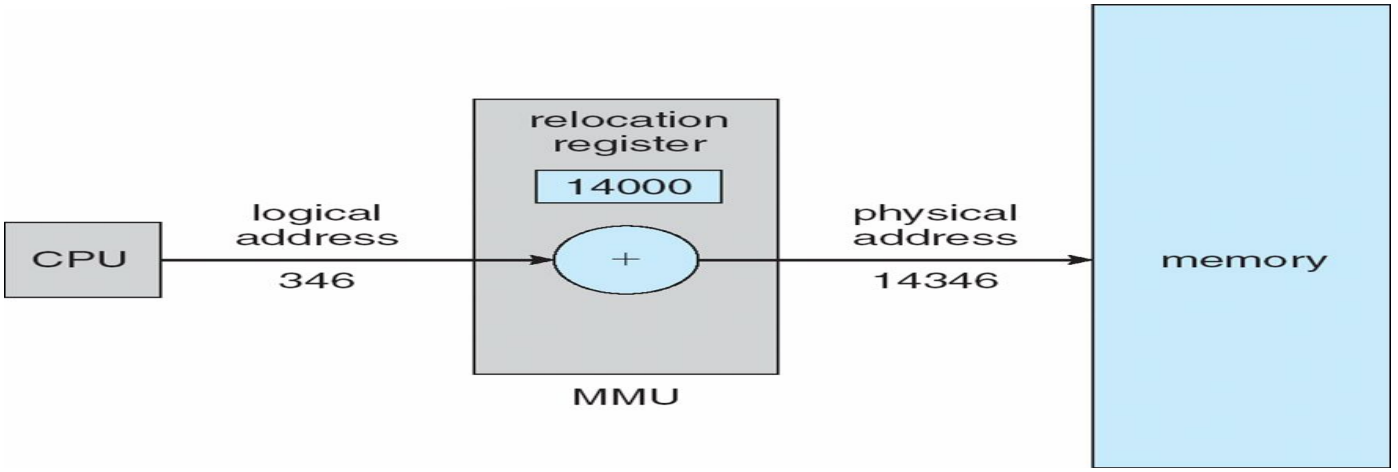
Physical or Absolute

- actual location in main memory

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- 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 relocation using a relocation register



- Dynamic loading loads up each routine as it is called.
- Unused routines need never be loaded.
- Reducing total memory usage and generating faster program startup times.
- The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then then loading it up if it is not already loaded.

Dynamic Linking

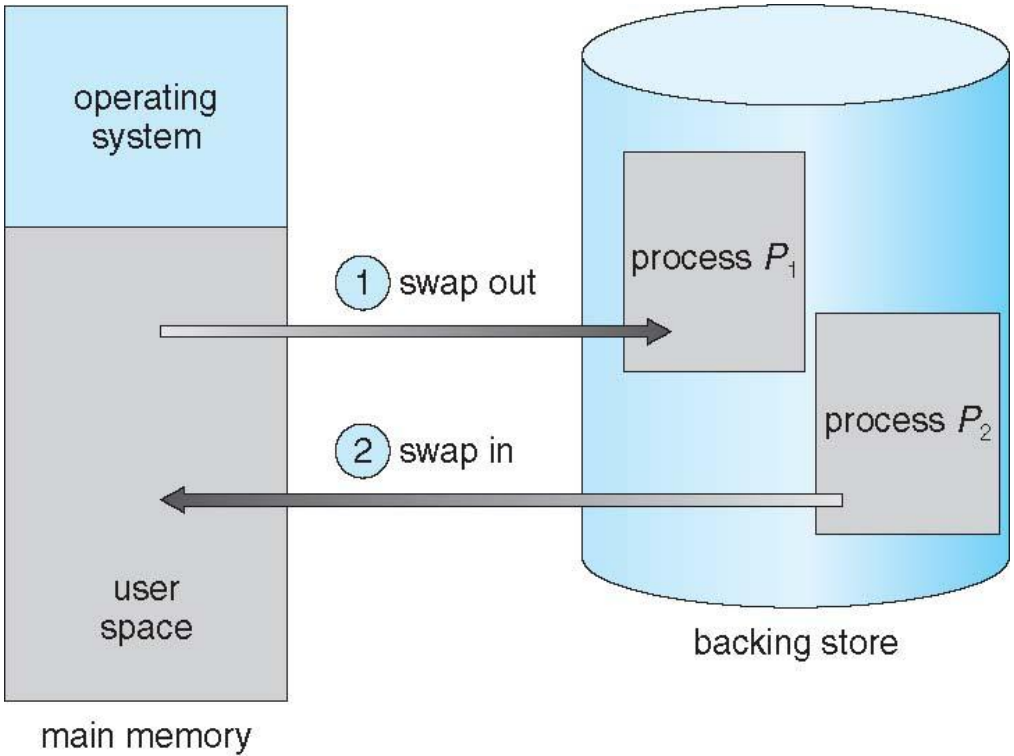
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address 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

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
- Does the swapped out process need to swap back in to same physical addresses?

Swapping…

- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

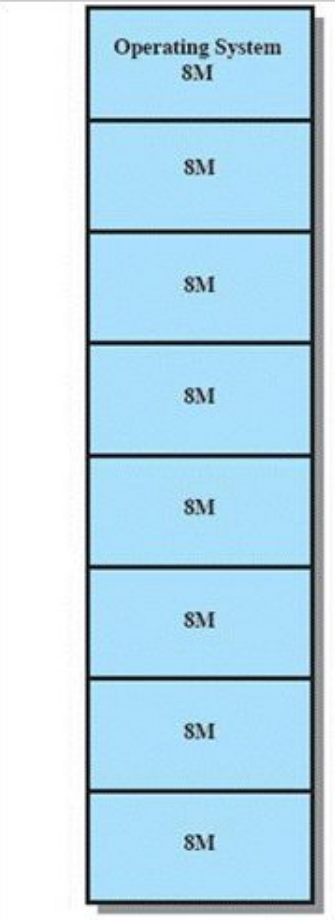


Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Fixed Partitioning

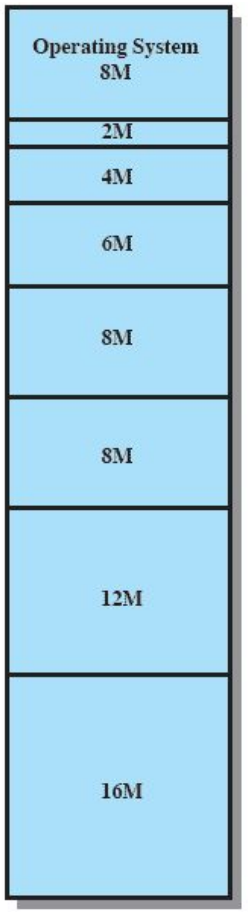
- **Equal-size partitions:** any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap out a process if all partitions are full and no process is in the Ready or Running state
- A program may be too big to fit in a partition, program needs to be designed with the use of **overlays.**
- **Disadvantage:**
  - Main memory utilization is inefficient any program, regardless of size, occupies an entire partition
  - The number of partitions specified at system generation time limits the number of active processes in the system
  - ***internal fragmentation*** **:**wasted space due to the block of data loaded being smaller than the partition



(a) Equal-size partitions

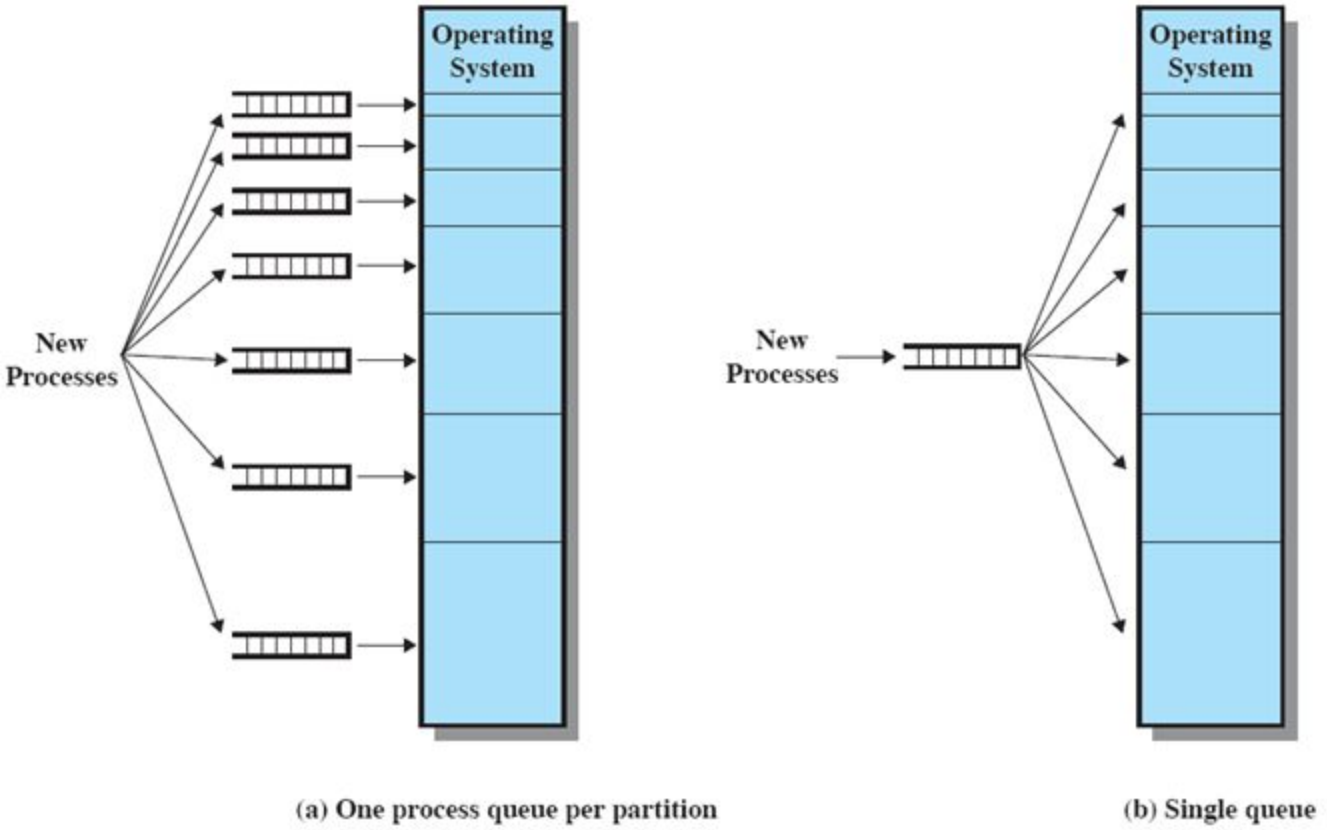
Unequal Size Partitions

- Using unequal size partitions helps lessen the problems
  - programs up to 16M can be accommodated without overlays
  - partitions smaller than 8M allow smaller programs to be accommodated with less internal fragmentation



(b) Unequal-size partitions

Memory assignment for fixed partitioning



(a) One process queue per partition

(b) Single queue

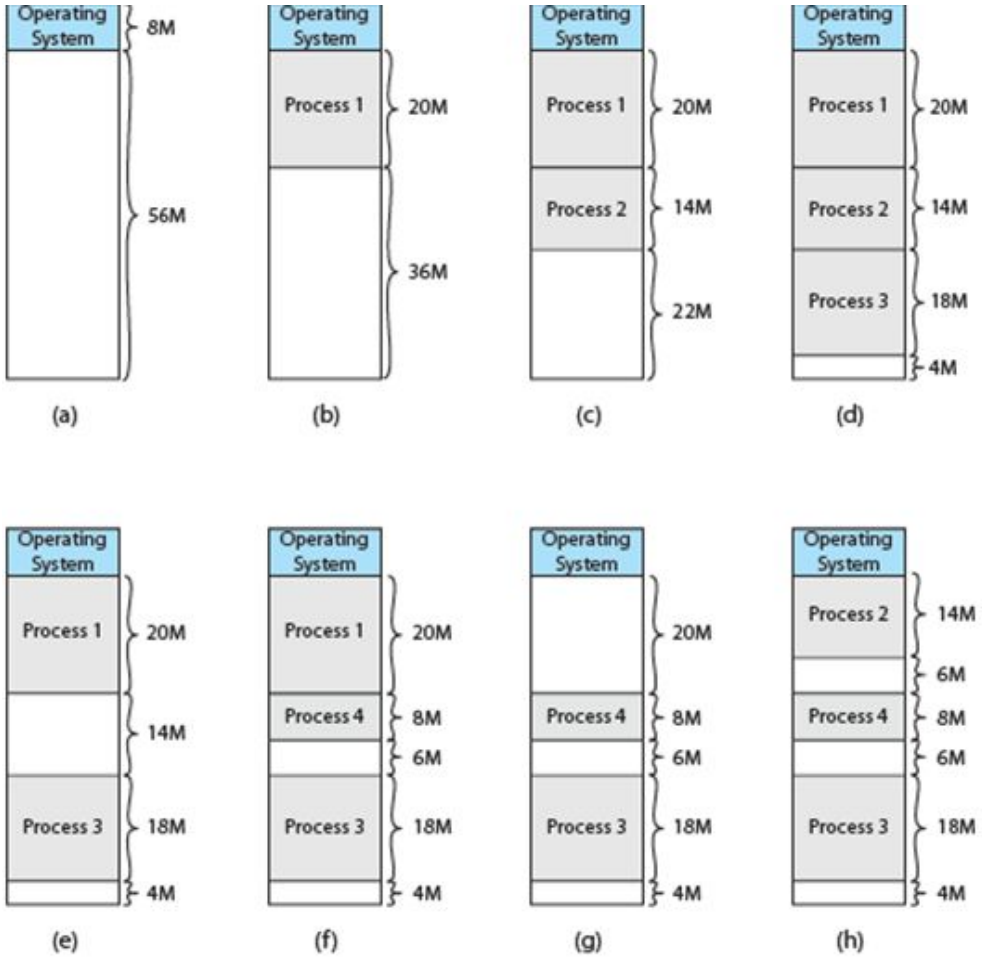
Figure 7.3 Memory Assignment for Fixed Partitioning

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBM's mainframe operating system, OS/MVT

Disadvantage:

- **External Fragmentation:** memory becomes more and more fragmented, memory utilization declines
- **Compaction:** technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous free memory is together in one block.
- time consuming and wastes CPU time



Placement Algorithms

Best-fit	First-fit	Next-fit	Worst-fit
• chooses the block that is closest in size to the request	• begins to scan memory from the beginning and chooses the first available block that is large enough	• begins to scan memory from the location of the last placement and chooses the next available block that is large enough	• Allocate the largest block; must also search entire list • Produces the largest leftover hole

Question?

- Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?
- First-fit:
  - 212K is put in 500K partition
  - 417K is put in 600K partition
  - 112K is put in 288K partition (new partition 288K = 500K - 212K)
  - 426K must wait
- Best-fit:
  - 212K is put in 300K partition
  - 417K is put in 500K partition
  - 112K is put in 200K partition
  - 426K is put in 600K partition
- Worst-fit:
  - 212K is put in 600K partition
  - 417K is put in 500K partition
  - 112K is put in 388K partition
  - 426K must wait

Question?

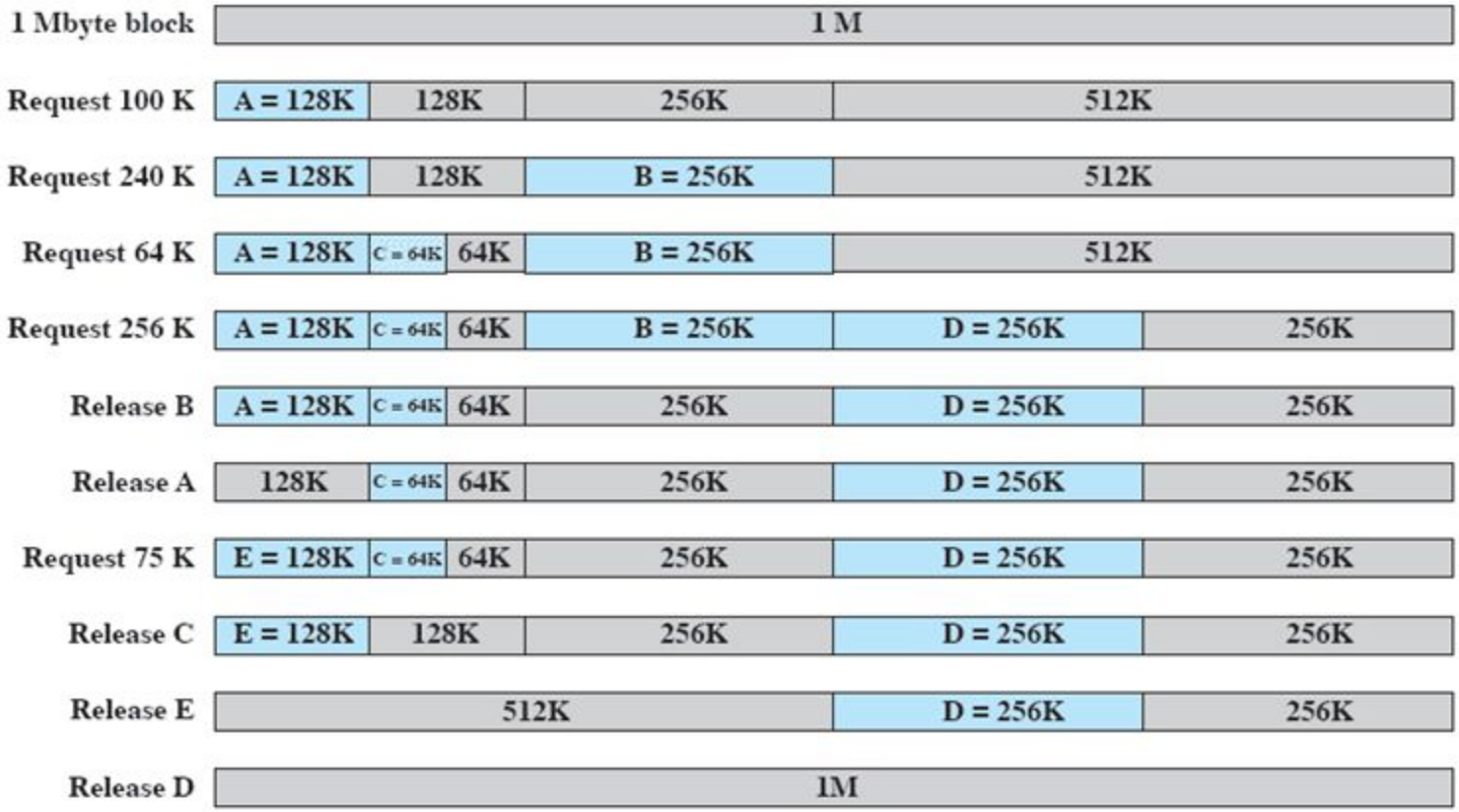
- **Exercise:** Consider the requests from processes in given order 300K, 25K, 125K and 50K. Let there be two blocks of memory available of size 150K followed by a block size 350K.
  Which of the following partition allocation schemes can satisfy above requests?
  A) Best fit but not first fit.
  B) First fit but not best fit.
  C) Both First fit & Best fit.
  D) neither first fit nor best fit.

https://www.geeksforgeeks.org/partition-allocation-methods-in-memory-management/?ref=rp

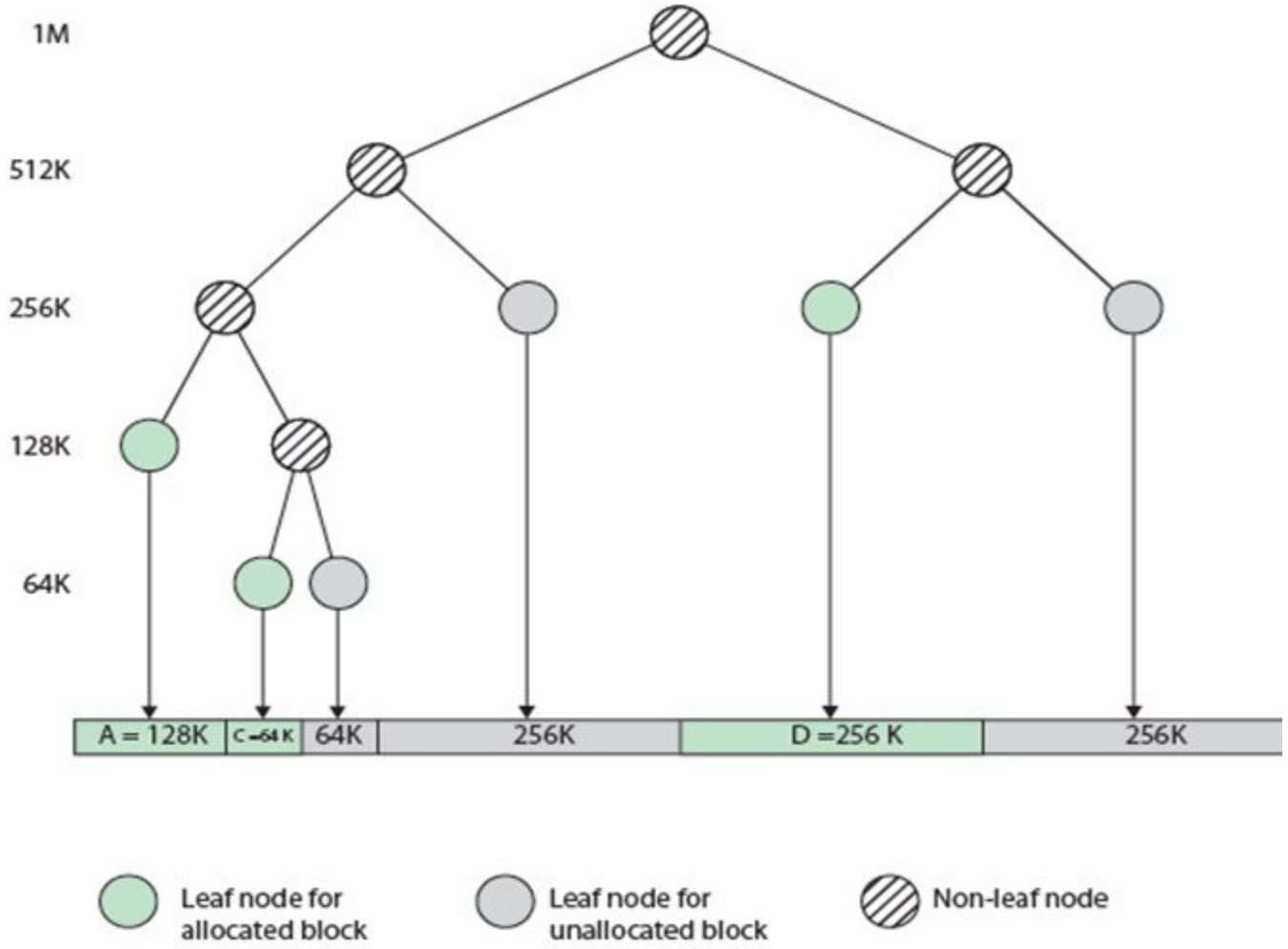
Buddy System

- Comprised of fixed and dynamic partitioning schemes
- Space available for allocation is treated as a single block
- Memory blocks are available of size 2^K *words,* $L \leq K \leq U$, where
    - 2^L = *smallest size block that is allocated*
    - 2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation

Example of buddy system



Tree representation of buddy system



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
- Partition memory into equal fixed-size blocks that are relatively small
- Process is also divided into small fixed-size blocks of the same size
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Still have Internal fragmentation

Pages	***Frames***
• Blocks of a process	• Available blocks of memory

Page Table

- Maintained by operating system for each process
- Contains the frame location for each page in the process(translate logical to physical addresses)
- Processor must know how to access the page table for the current process
- Used by processor to produce a physical address

Question?

- (Gate 2015) Consider a system with byte-addressable memory, 32−bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is

total no of pages $= \frac{2^{32}}{2^{12}} = 2^{20}$

We need a PTE for each page and an entry is 4 bytes. So,

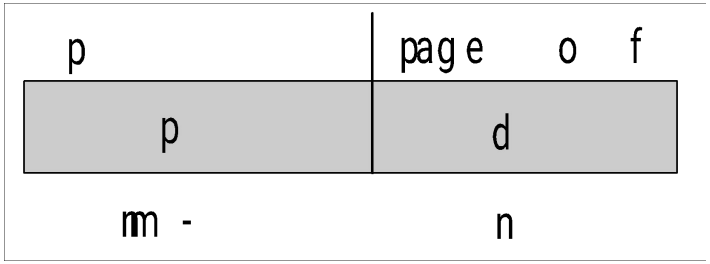
page table size $= 4 \times 2^{20} = 2^{22} B = \mathbf{4MB}$

Internal Fragmentation

- 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

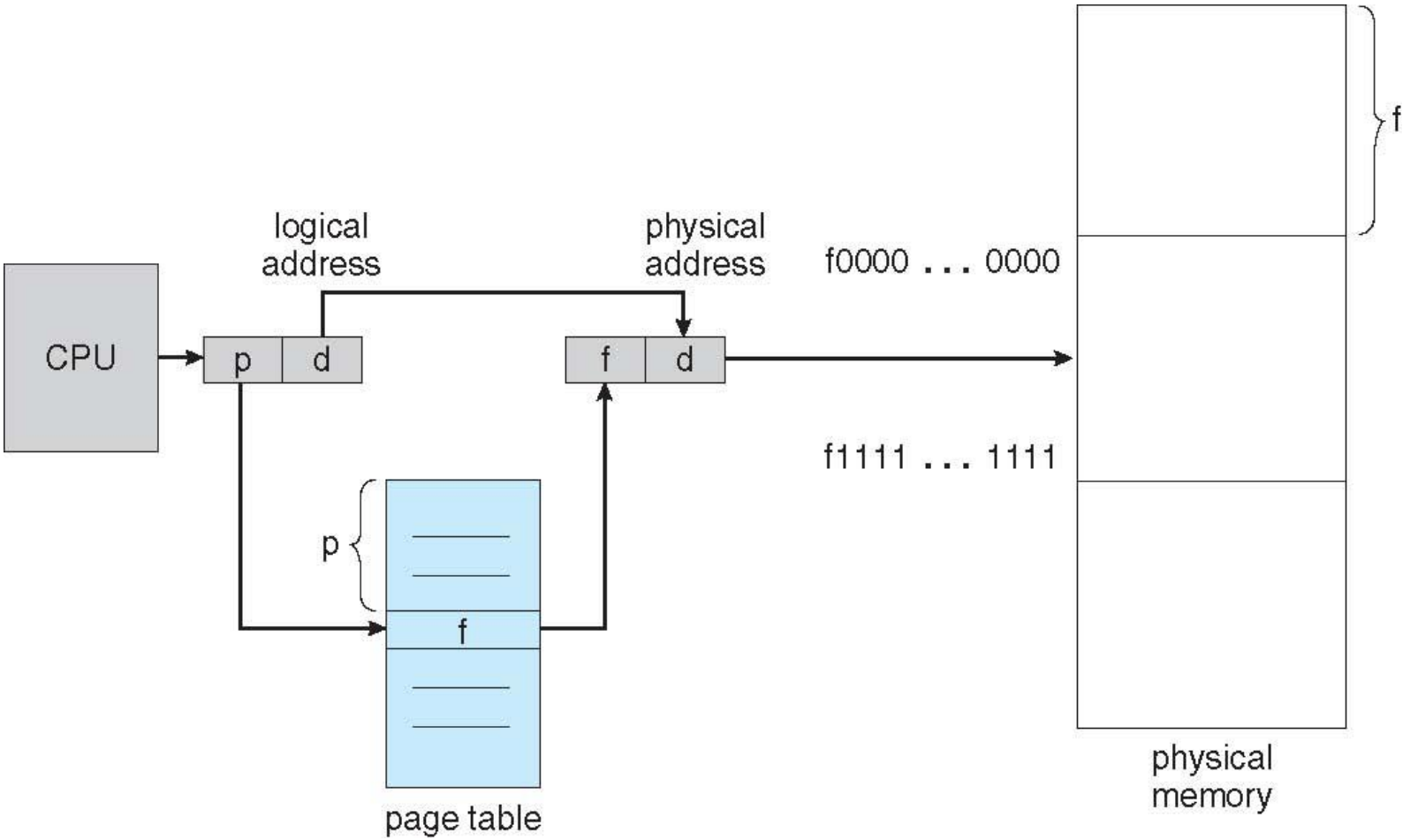
Address Translation Scheme

- 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

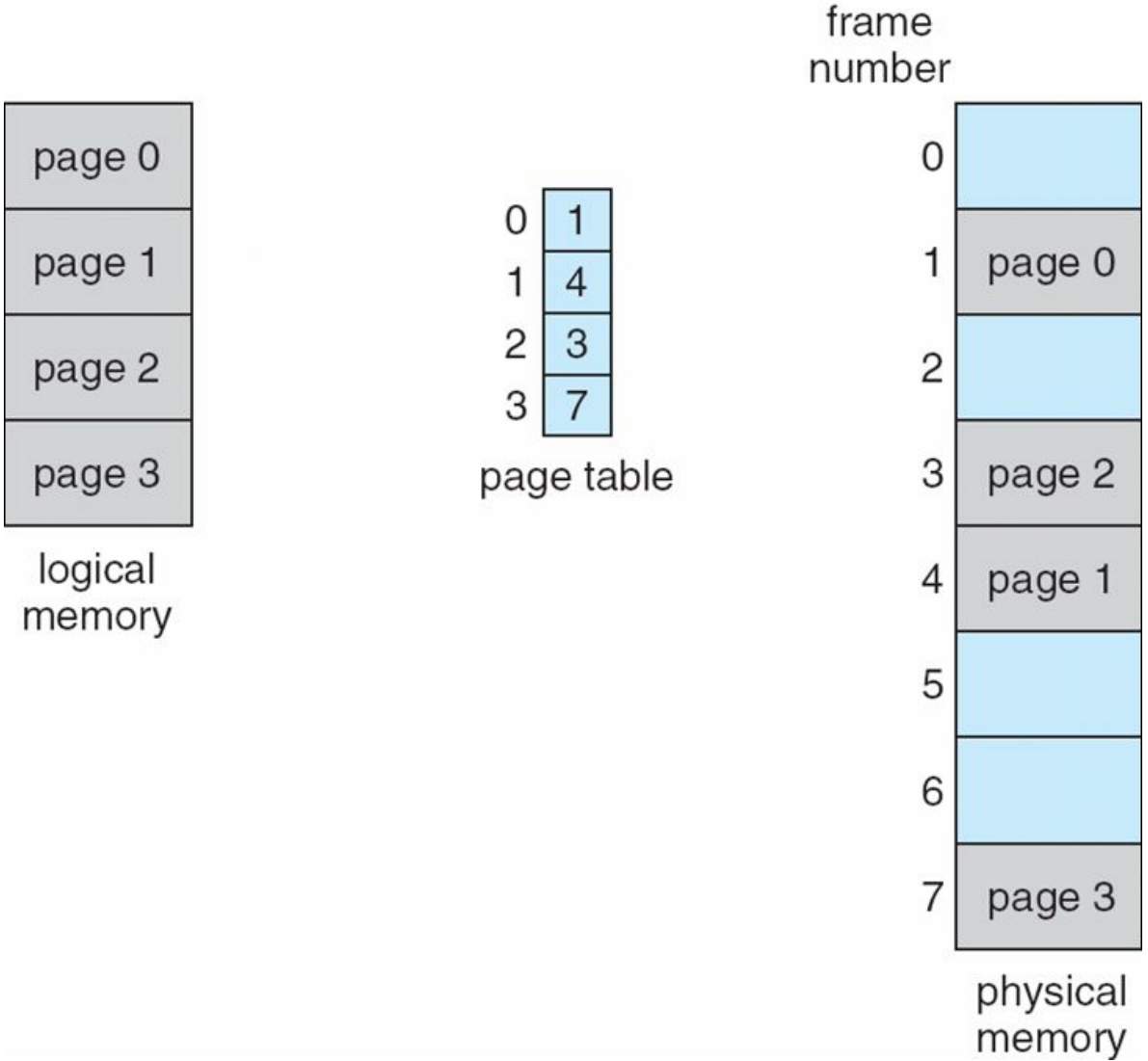


  - For given logical address space 2^m and page size 2^n

Paging Hardware

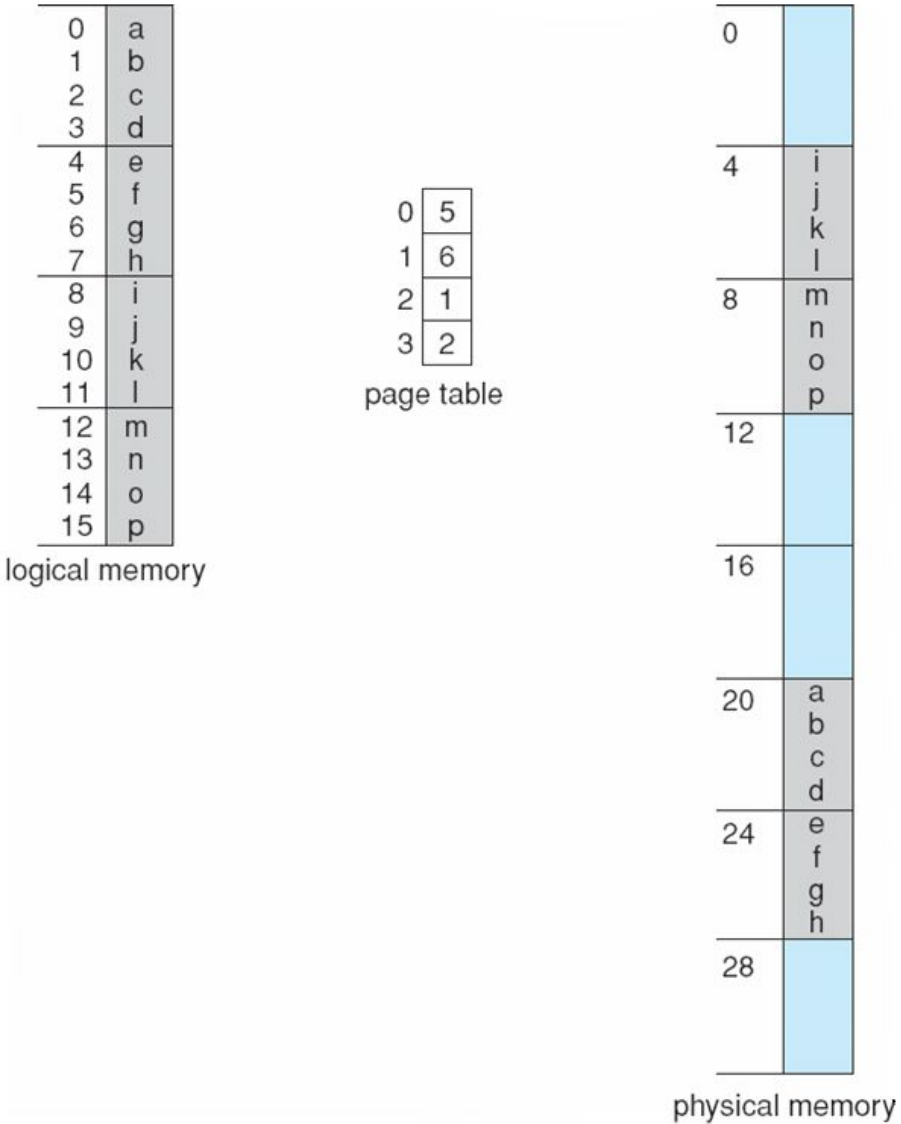


Paging Model of Logical and Physical Memory

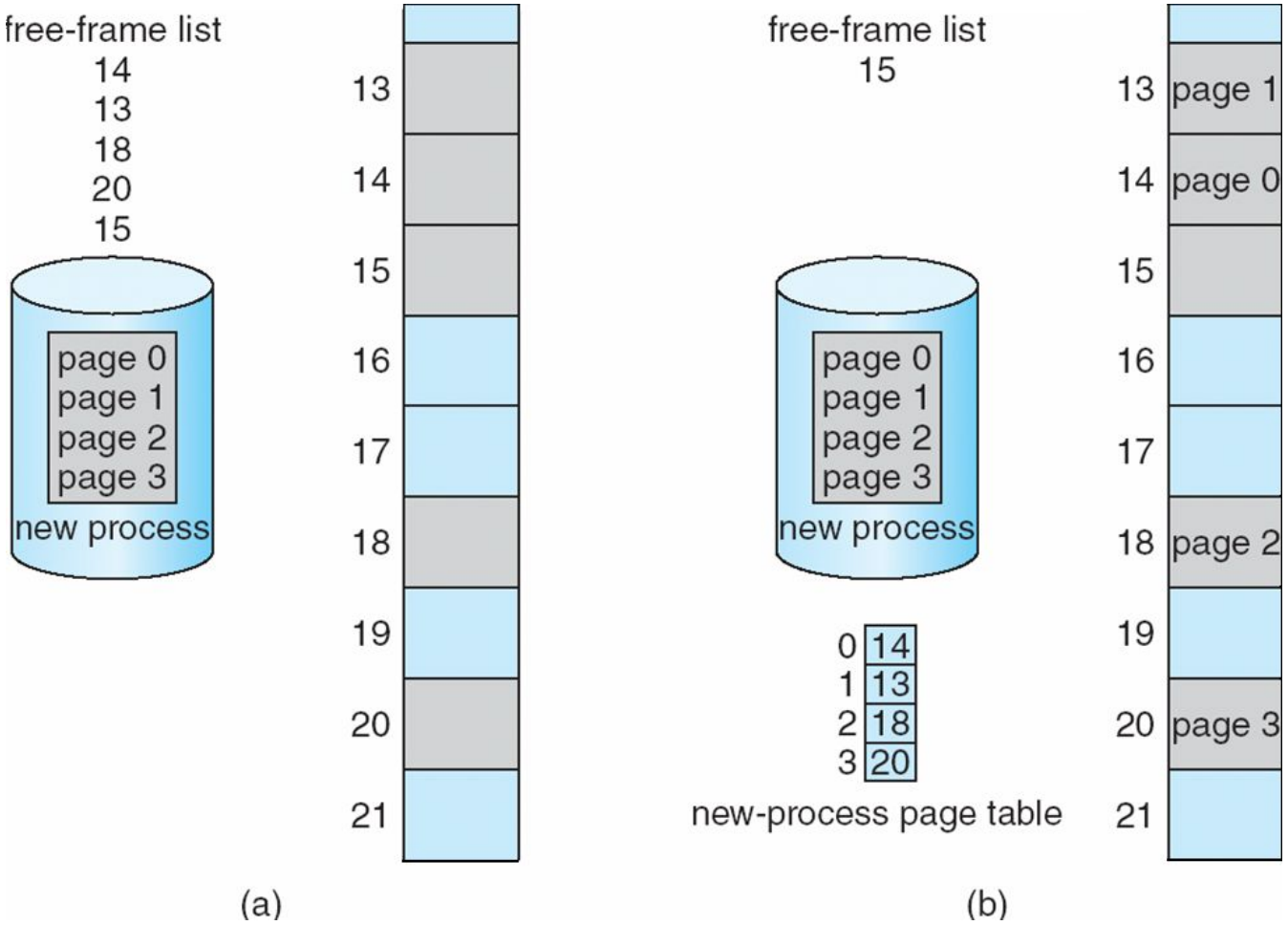


Paging Example

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



Free Frames



Before allocation

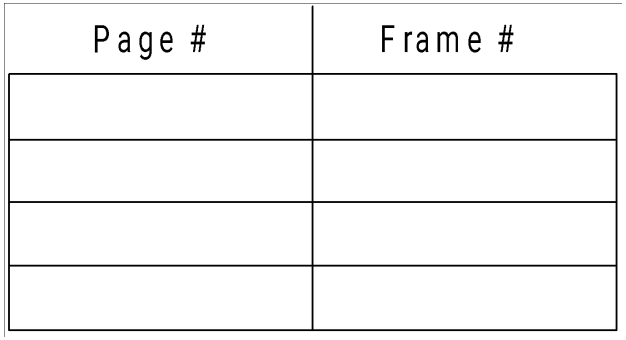
After allocation

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 **associative memory** or **translation look-aside buffers** (**TLBs**)

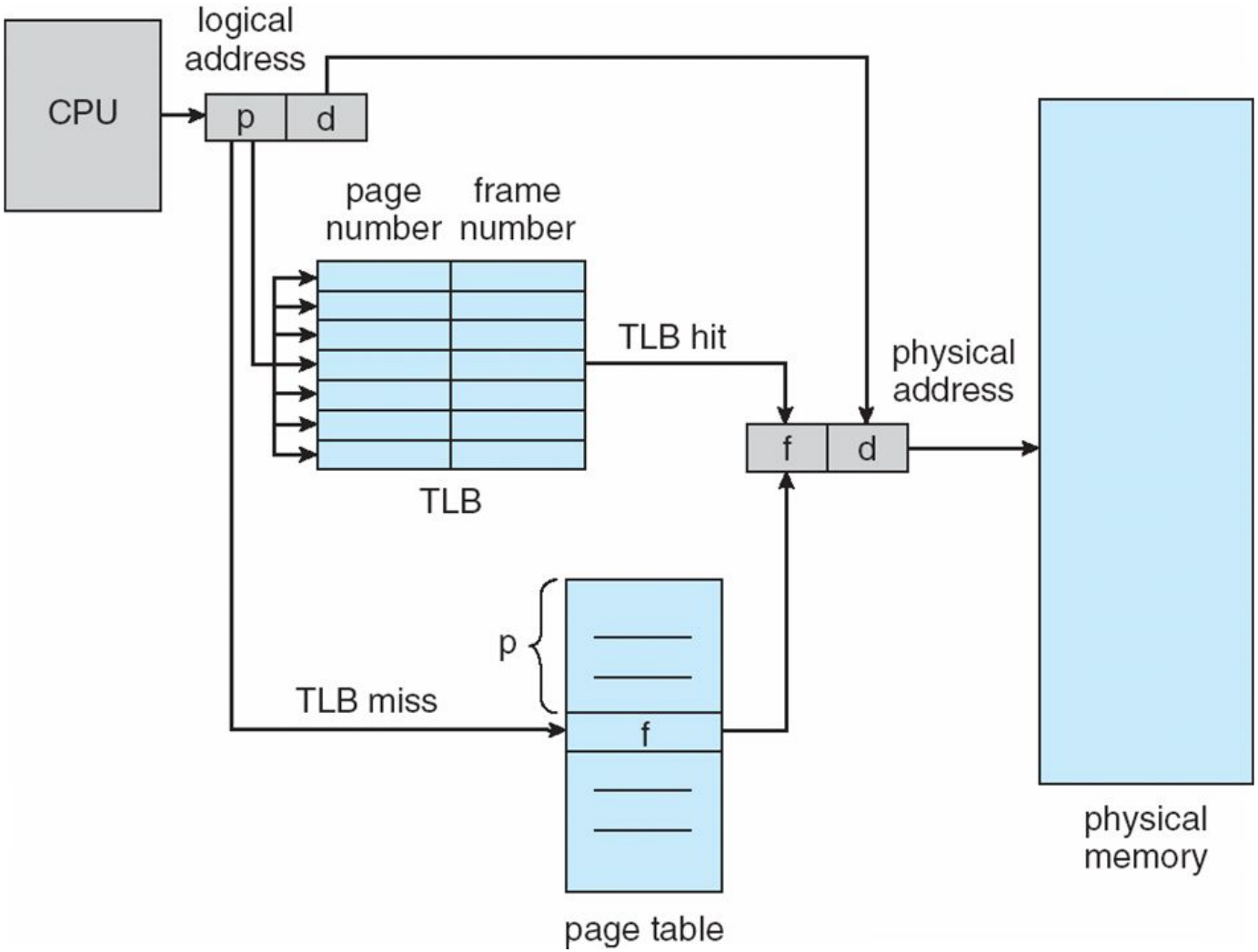
Associative Memory

- Associative memory – parallel search



- Address translation (p, d)
  - If p is in associative register, get frame # out(**TLB Hit**)
  - Otherwise get frame # from page table in memory(**TLB Miss**)

Paging Hardware With TLB

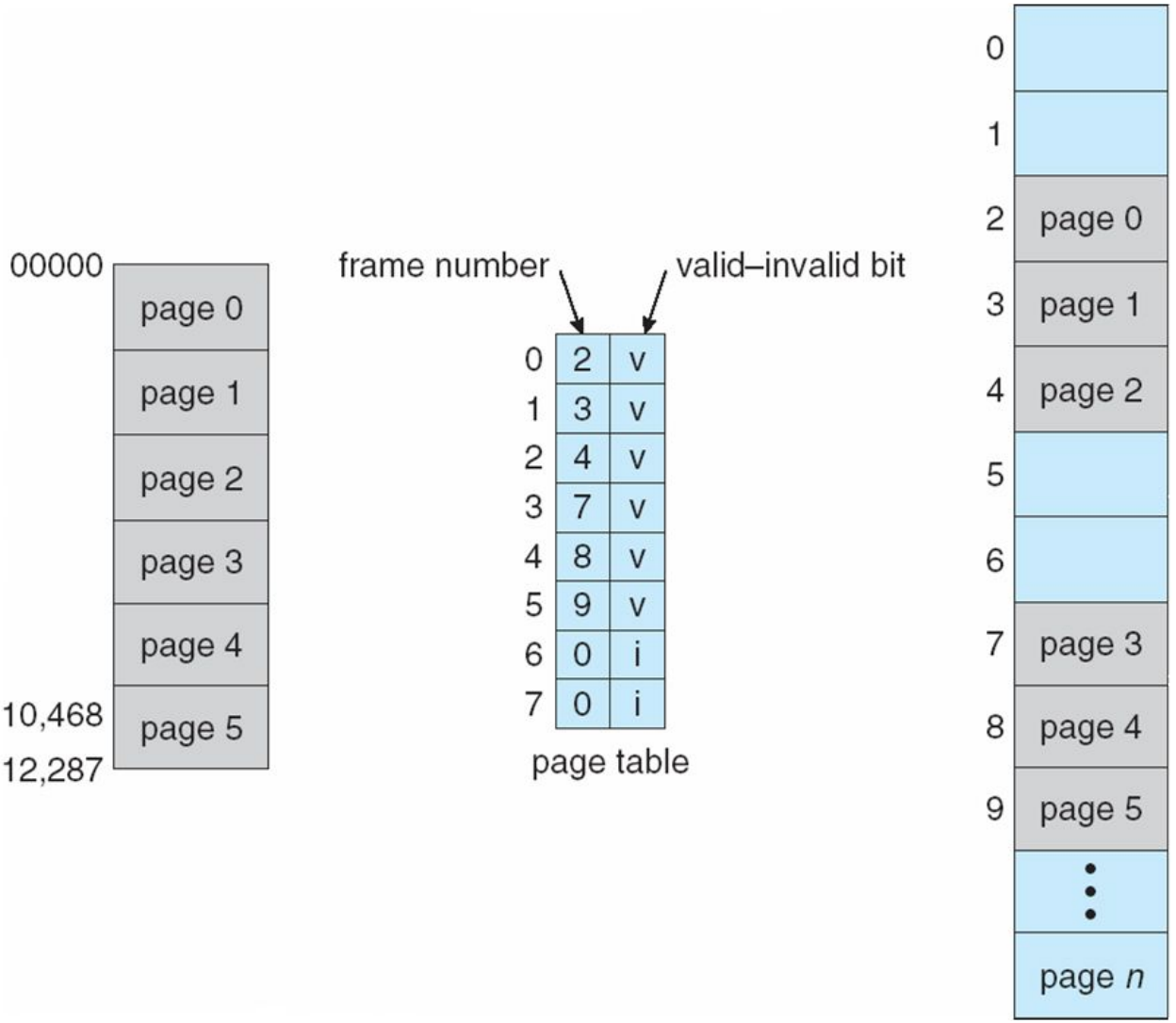


Effective Memory Access Time

- 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
- **EMAT**=TLB hit × (TLB access time + memory access time) + TLB miss(TLB access time + page table access time + memory access time)
- (**Gate 2014**) Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 milliseconds to search the TLB and 80 milliseconds to access the physical memory. If the TLB hit ratio is 0.6, the effective memory access time (in milliseconds) is _____.

(A) 120 **(B)** 122 **(C)** 126 **(D)** 130

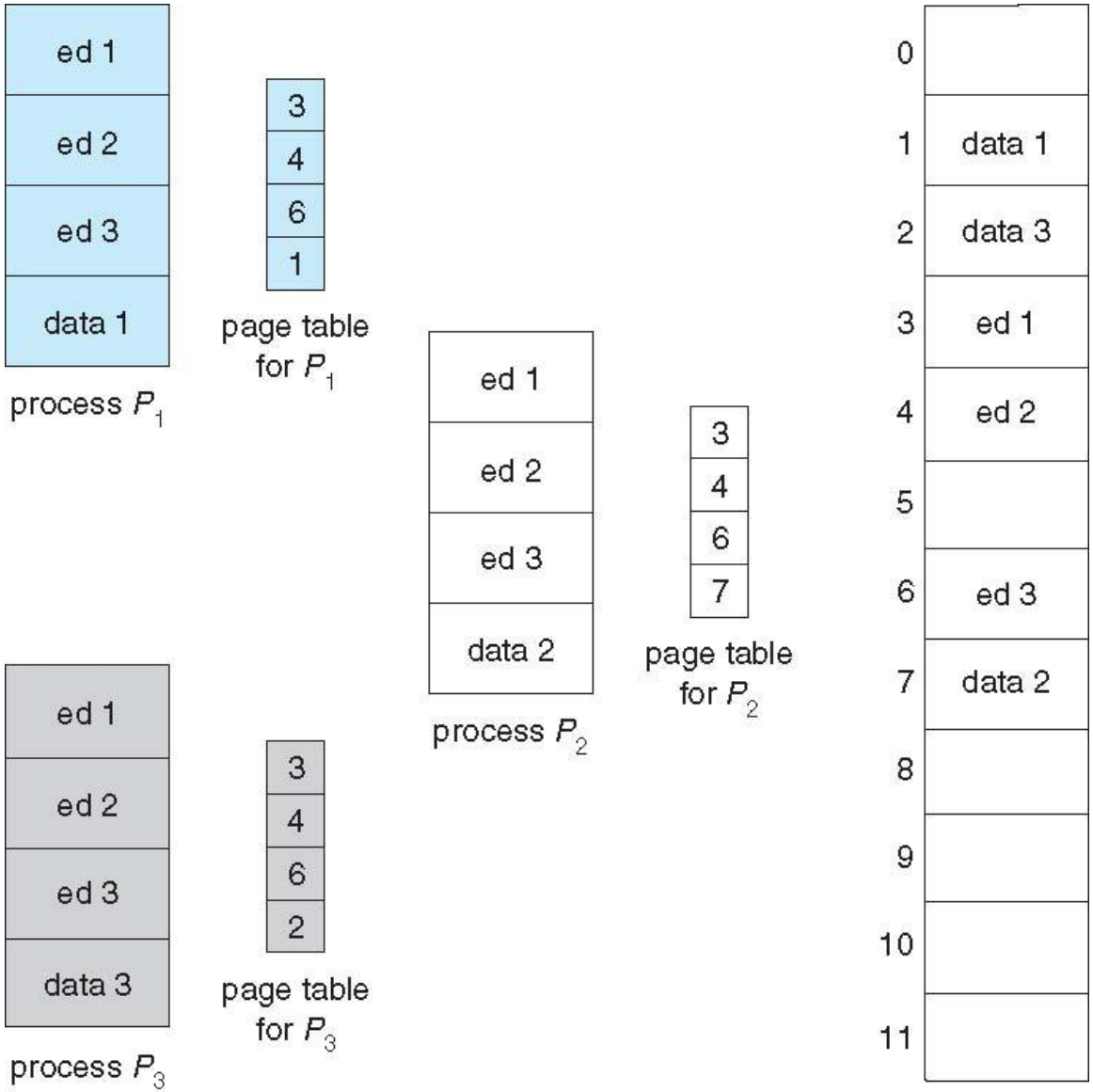
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

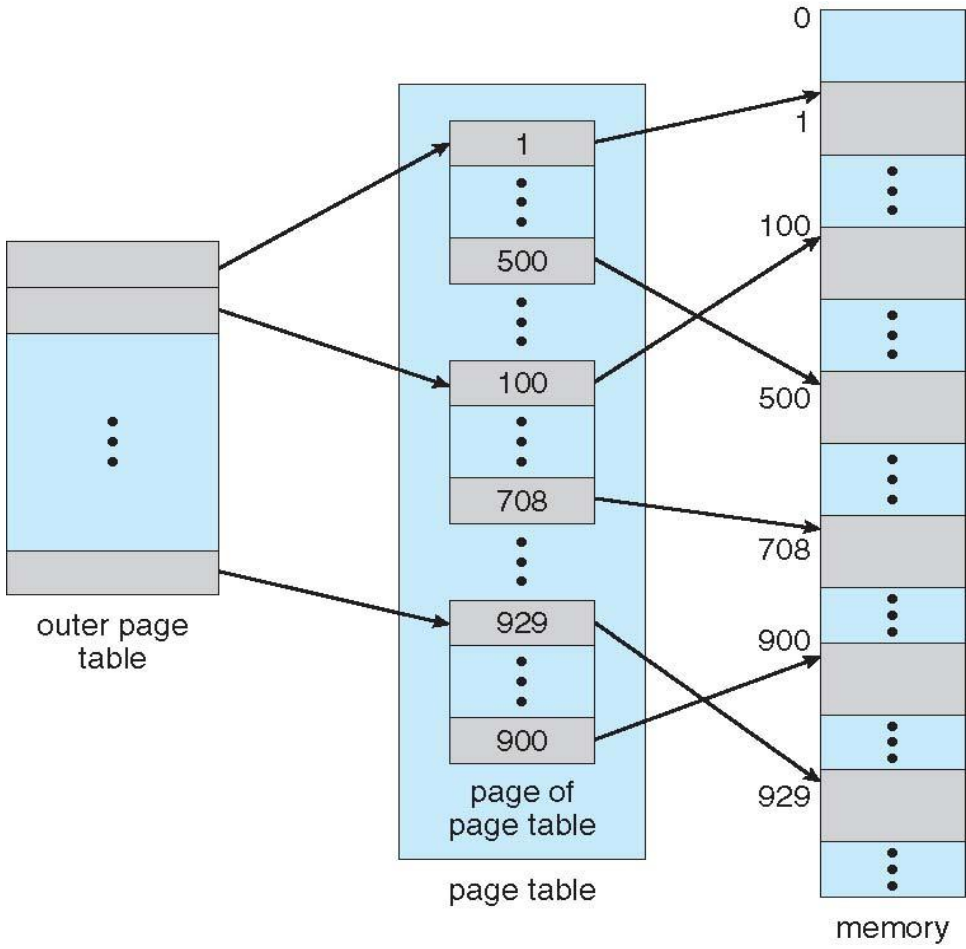
- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code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



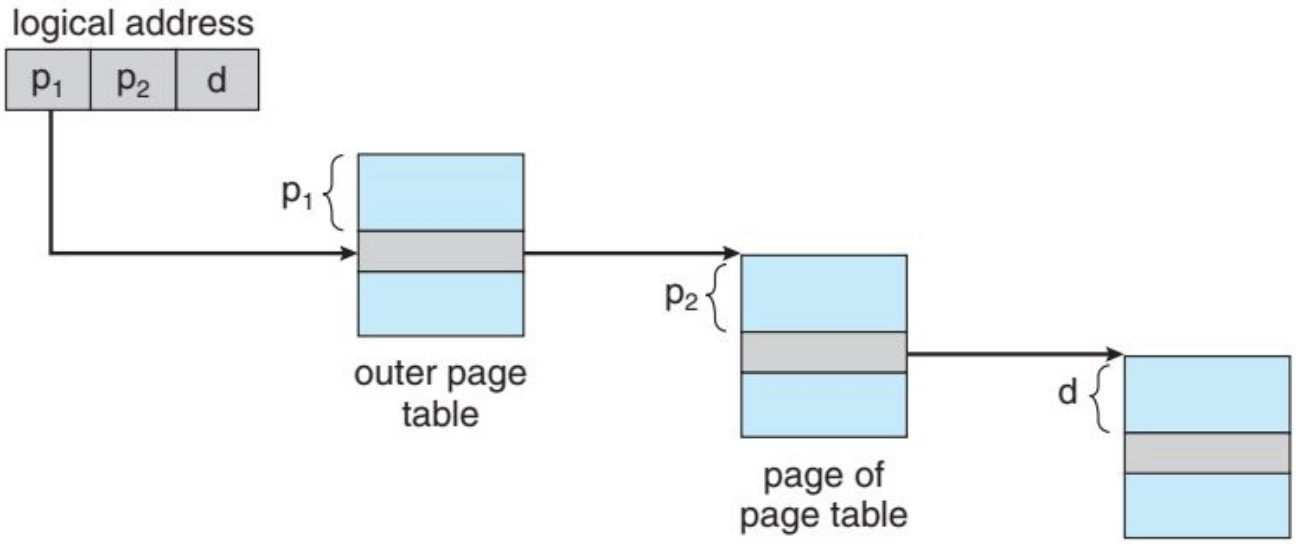
Hierarchical Page Tables

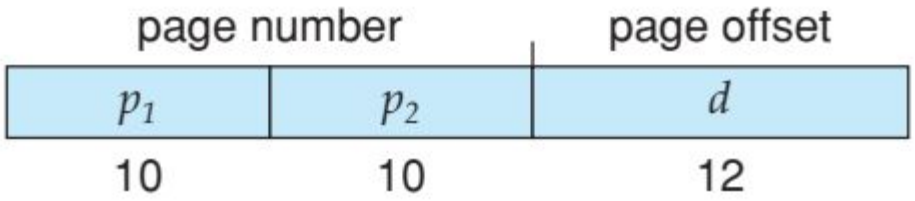
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- Page the page table



Two-Level Page-Table Scheme

Hierarchical Page Tables

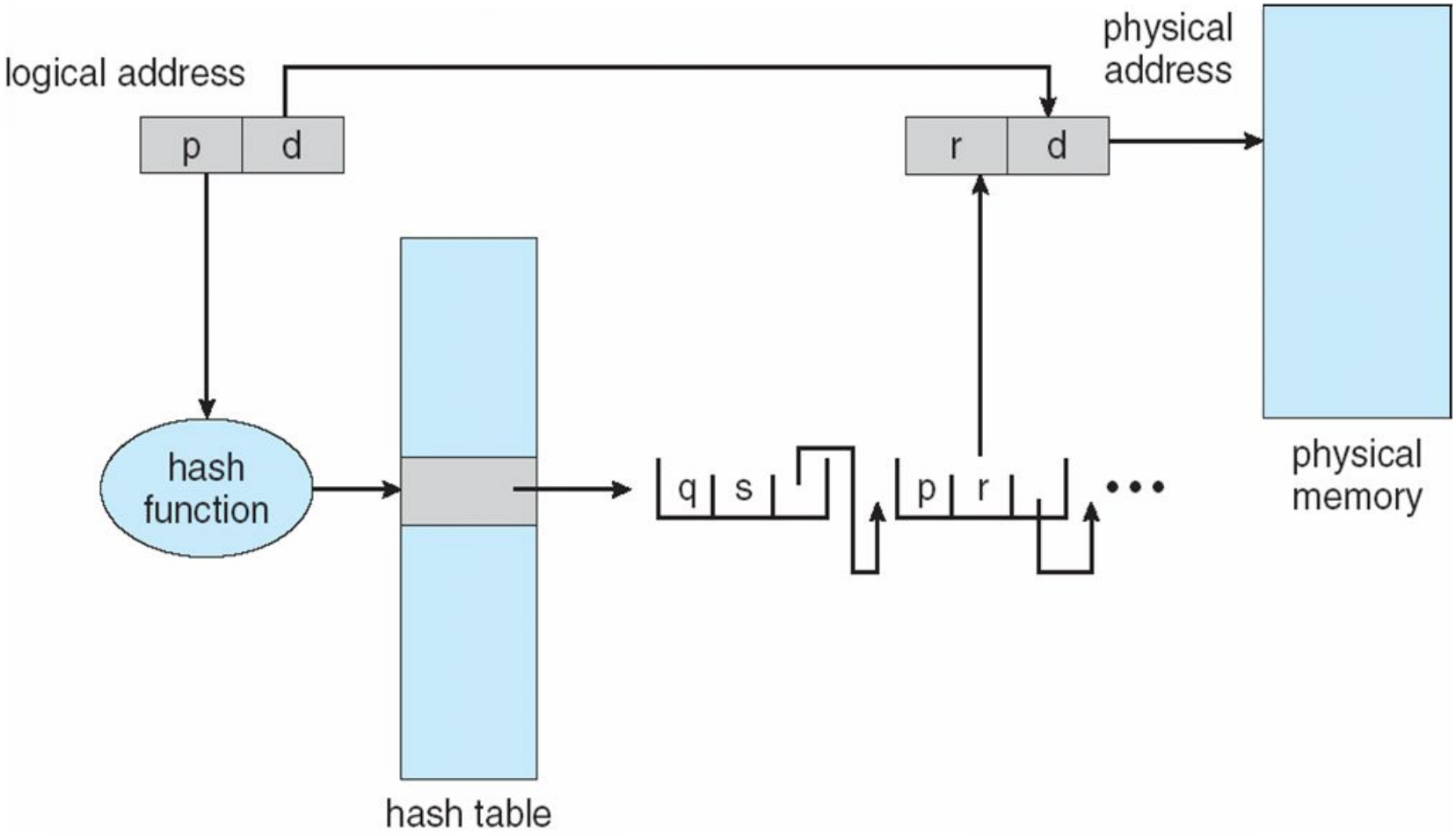




Hashed Page Tables

- 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 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 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