Chapter 9: Main Memory

Part 1

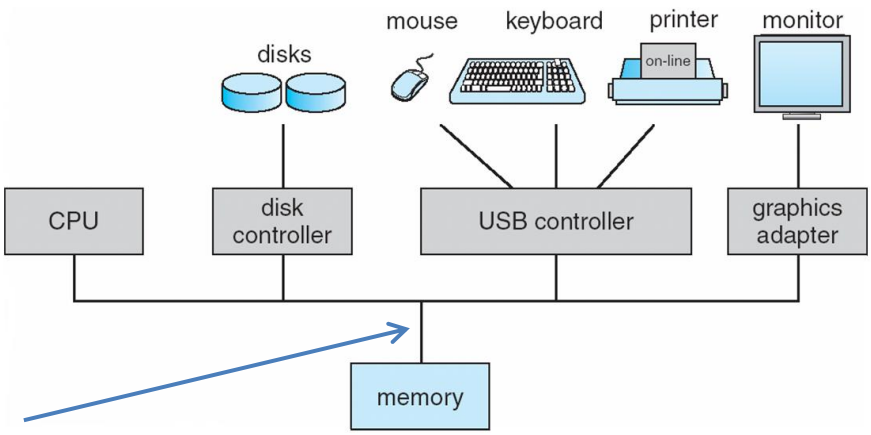
9.1 and 9.2 Contiguous Allocation

Part 2

9.3 Paging

Introduction and Background:

Add memory controller to this picture.



Memory Controller

We will be discussing the activities in the Memory Controller for the most part.

9.1 Logical vs Physical Address

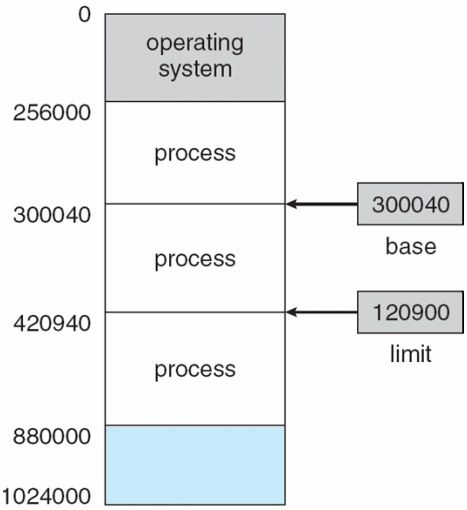
- CPU registers and Main Memory are only storage CPU can access directly
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Memory Controller unit only sees a stream of *(addresses + read)* requests, or *address + data +write* requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers (We will omit cache

discussion here)

- Protection of memory required to ensure correct operation

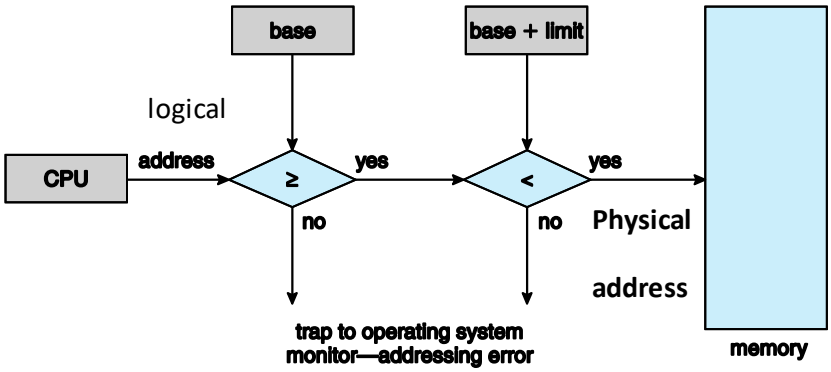
9.1 Base and Limit Registers

- A pair of **base (also called relocation register)** and **limit registers** define the logical address space of a process.
- CPU must check every memory access generated in user mode to be sure it is between *base* and *base+limit* for that user



9.1 Address Translation

The process of taking CPU's *logical* address request and translating that to *physical* address on the RAM is called address translation

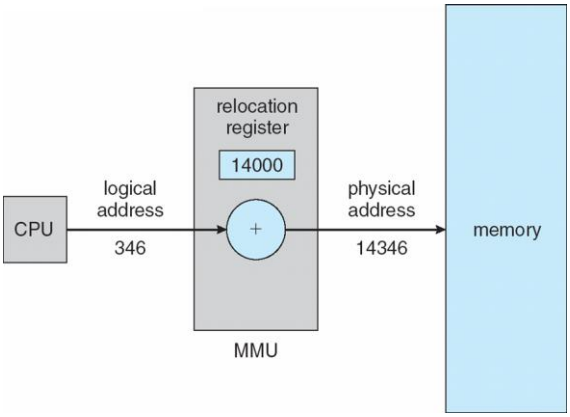


9.1 Logical vs. Physical Address

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
    - **Logical address (virtual address)** --generated by the CPU;
    - **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
- Hardware device that at run time maps virtual to physical address is called **Memory Management Unit MMU, part of the memory controller**

Dynamic Relocation using a relocation register

- CPU generates address relative to the beginning of the code. This is the logical (virtual) address.
- Then MMU adds the relocation register value (base register) to get the physical address.
- CPU does not know the physical address.
- The relocation register can change when the program does an I/O and comes back or just like that.
- So it is called Dynamic (Run-Time) address determination.



9.2 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

- Relocation registers used in MMU 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*

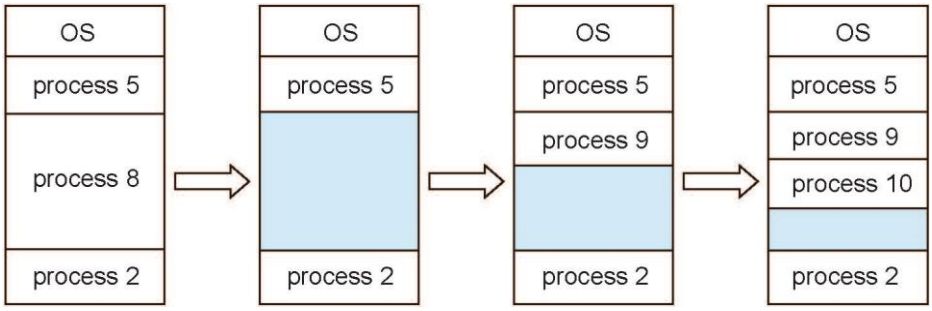
- Size of the kernel may change if code is being **transient** due to device drivers being added etc.

9.2 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)



9.2 Dynamic Storage-Allocation

- **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

9.2 Example

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?

9.2 Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request,

but it is not contiguous

External Fragmentation = 1 – (Largest Block of Free Memory/Total Free Memory)

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

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-half of memory may be unusable! This property is known as the **50-percent rule**.

- **Internal Fragmentation** – Suppose a process is 799 bytes and the partitions are multiples of 100 bytes, then we have to keep track of 2 bytes.. instead we allocate 900 bytes. So, allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

9.2 Fragmentation-Removal

- 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

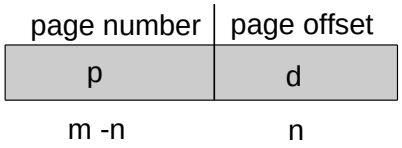
Part 2

9.3 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

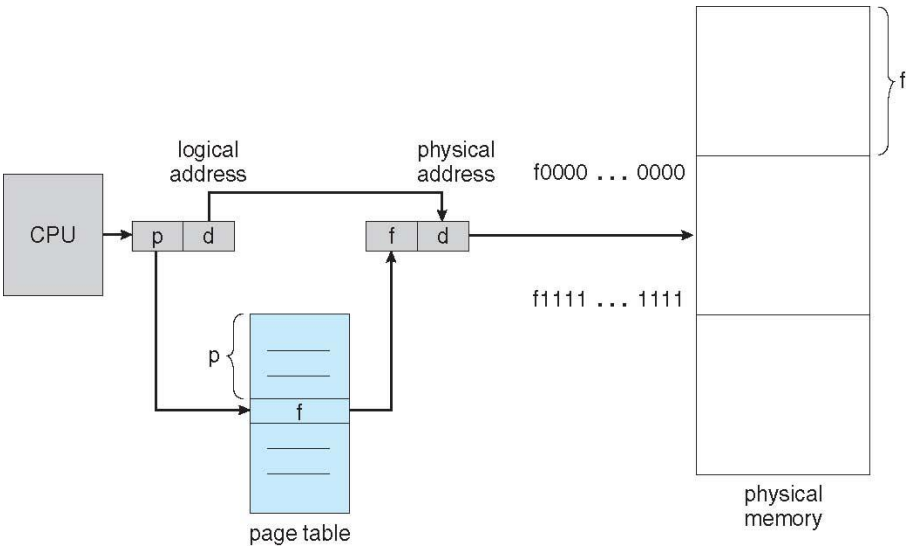
9.5 Address Translation Scheme

- 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

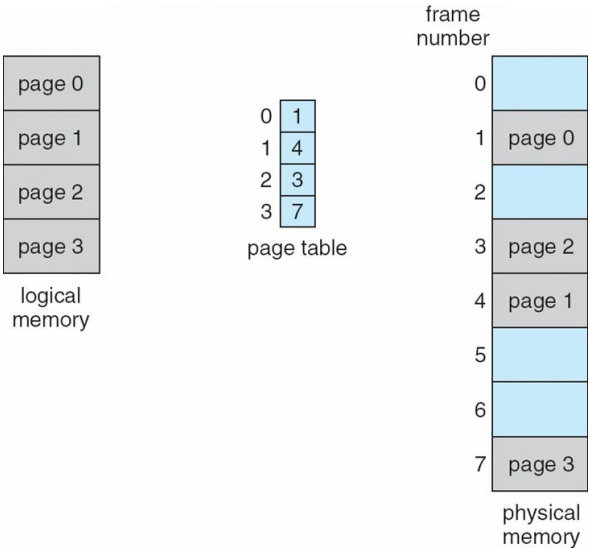


- For given logical address space 2^m and page size 2^n

9.5 Paging Hardware

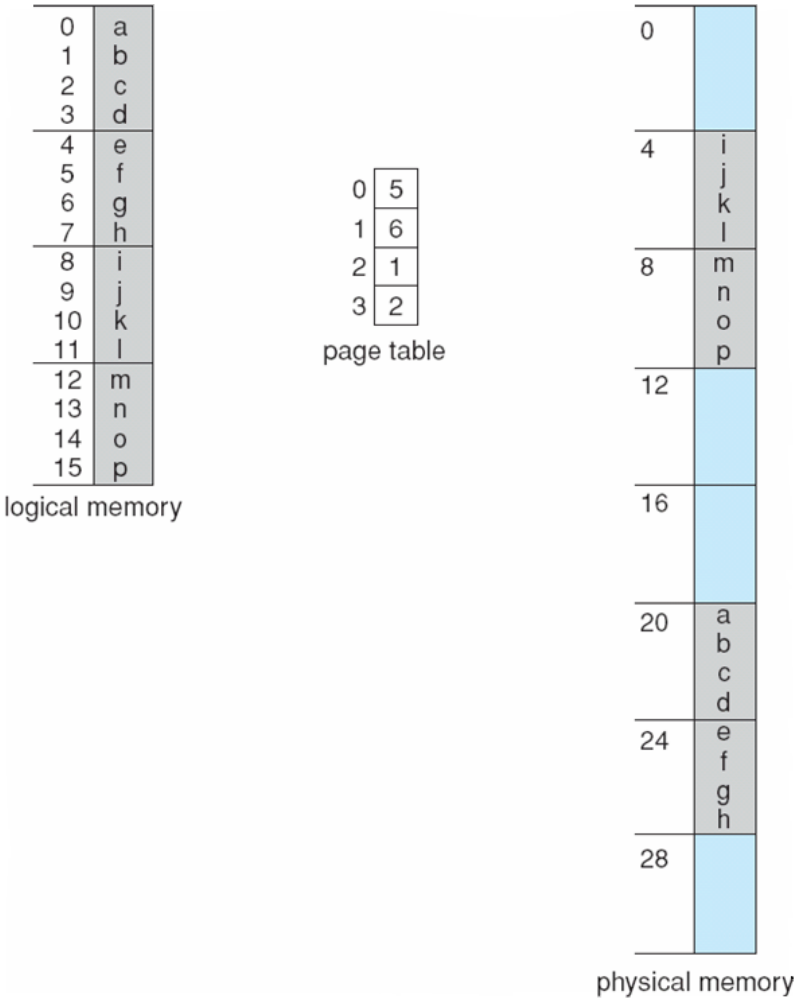


9.5 Paging Model of Logical and Physical Memory



9.5 Paging Example

m=5, n=2 32=(2^5)= =(2^m)= bit memory and 4-bit (2^2)= 2^n pages



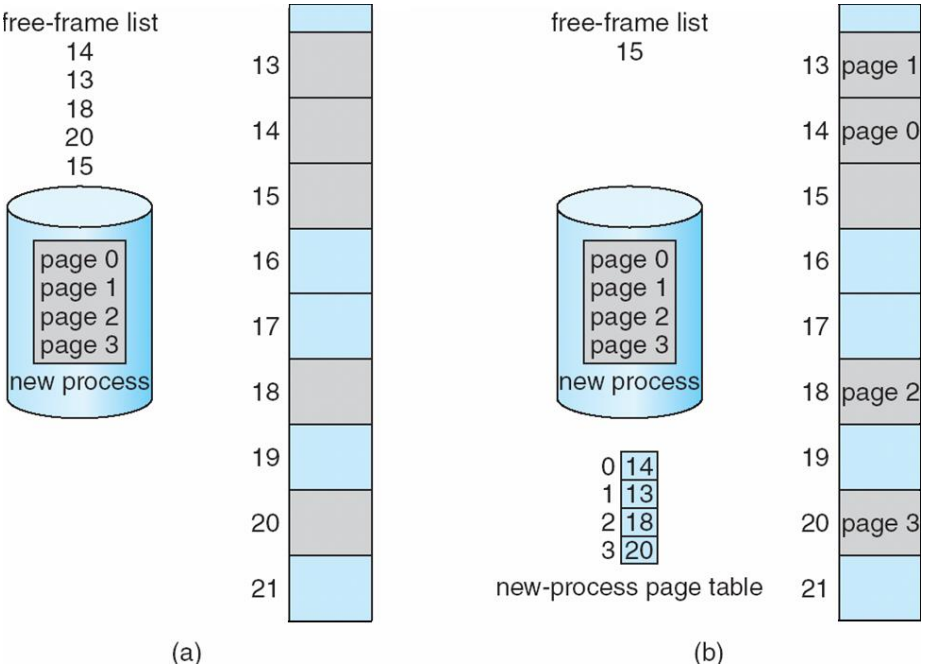
- It is possible for a process to have more pages than physical RAM size itself!. So, valid/invalid bits are used (see later)

Example: Assuming a 1 KB page size, (2^10 = 1024) what are the page numbers and offsets for the following address references (provided as decimal numbers):
a. 2375
b. 19366
c. 30000
d. 256
e. 16395

Answer:

a. page = 2; offset = 327
b. page = 18; offset = 934
c. page = 29; offset = 304
d. page = 0; offset = 256
e. page = 16; offset = 1

9.5 Free Frames



- Calculating internal fragmentation
    - Page size = 2,049 bytes
    - Process size = 72,766 bytes
    - 35 pages + 1,096 bytes
    - Internal fragmentation of 2,049 - 1,096 = 962 bytes
- What is a good page size?
    - 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 – 9 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

9.5 Implementation of Page Table

- List of free frames is kept in memory, could be an array called frame table, indicating the list of free frames.
- 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

9.5 Effective Access Time with Pure Paging

- Effective access time is defined as the time needed to access the physical address given the logical address
- With paging, this time includes
  -- consulting the page table to find the frame number
  --- accessing the physical RAM at the given frame number

- Effective Access Time :
  Assuming it takes 100ns for memory access,
  --- If the page table is stored in the RAM then 100ns for accessing the page table
  ----- 100 ns for accessing data at the physical frame.

So the **EAT** is 200 ns.

9.5 Translation Lookup Buffers (TLBs)

- 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)

- TLBs typically small (64 to 1,024 entries)
- **Wired Down** technique may be used to indicate not to remove an entry from TLB like the kernel pages.
- 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

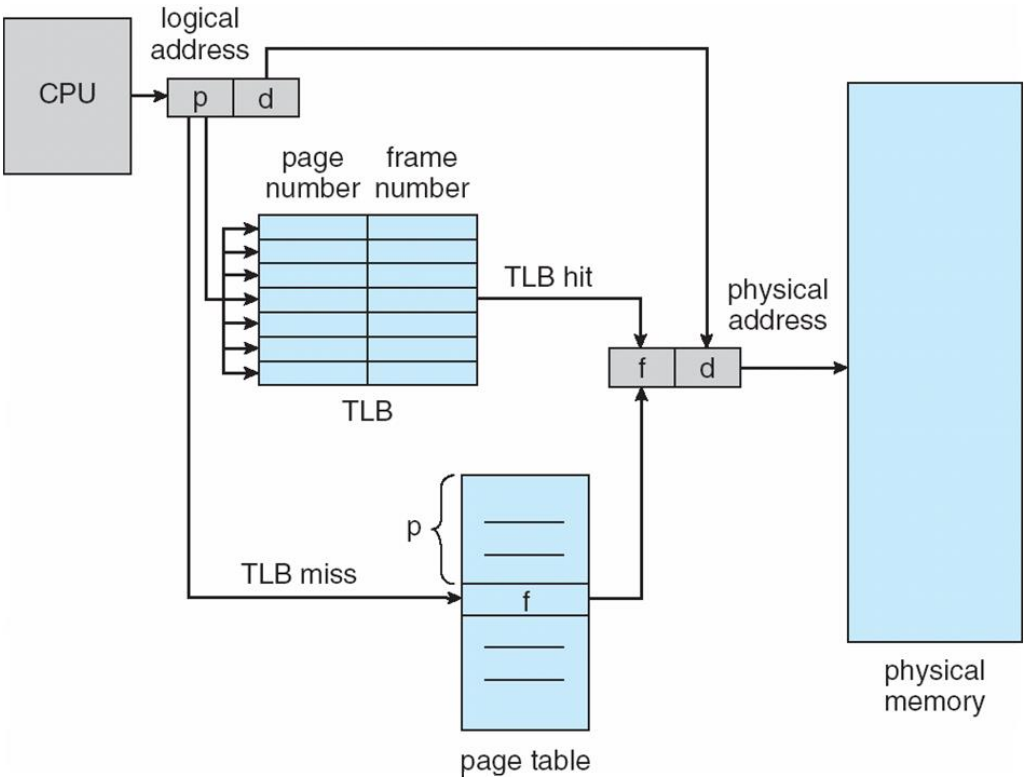
9.5 Translation Lookup Buffers

- 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

9.5 Paging Hardware With TLB



9.5 Effective Access Time with TLBs

- 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

- **Effective Access Time** (**EAT**)

  Consider α = 90%, ε = 20ns for TLB search, 100ns for memory access
  EAT =20 (TLB) + 0.90 x 100 + 0.20 x 200 = 20 + 120 = 140ns
  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 = 20 + 101 = 121ns

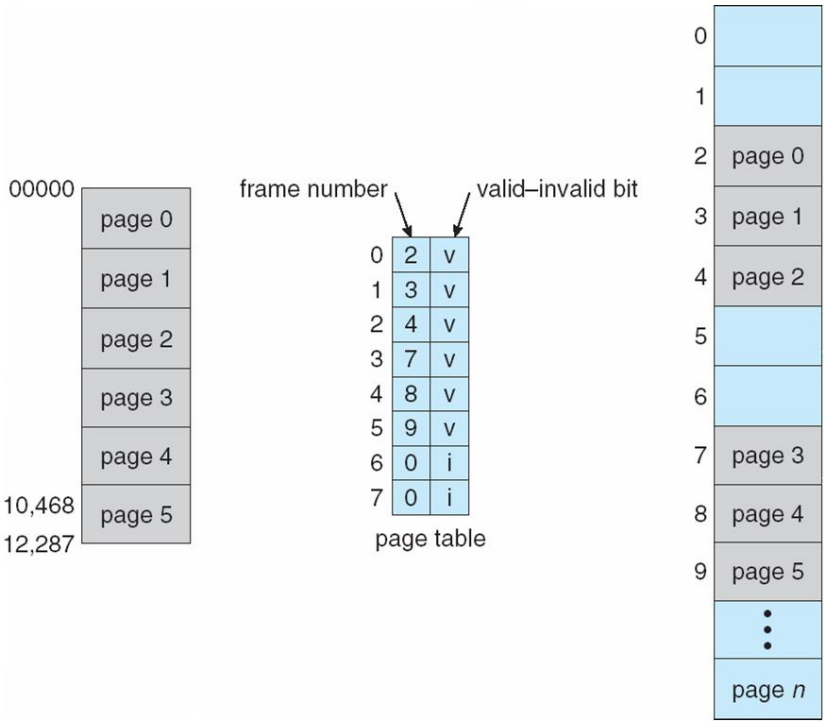
9.5 Valid-Invalid Bit

- **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
- Any violations result in a trap to the kernel

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

Initially ALL pages are marked Invalid. Each page faults first time it is brought in(Demand paging). When it is in, then it is marked valid.

Also by using additional Protection bits we can mark Read-Only, Read-Write pages for each page.



9.5 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

9.5 Shared Pages Example