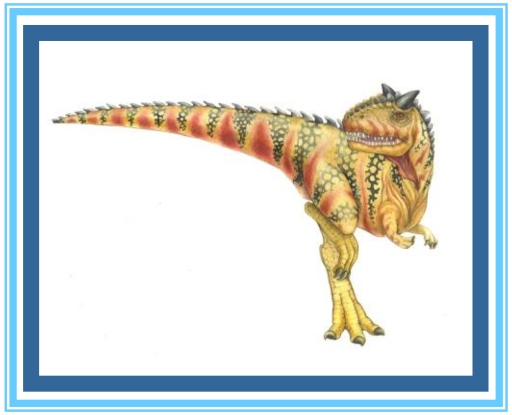
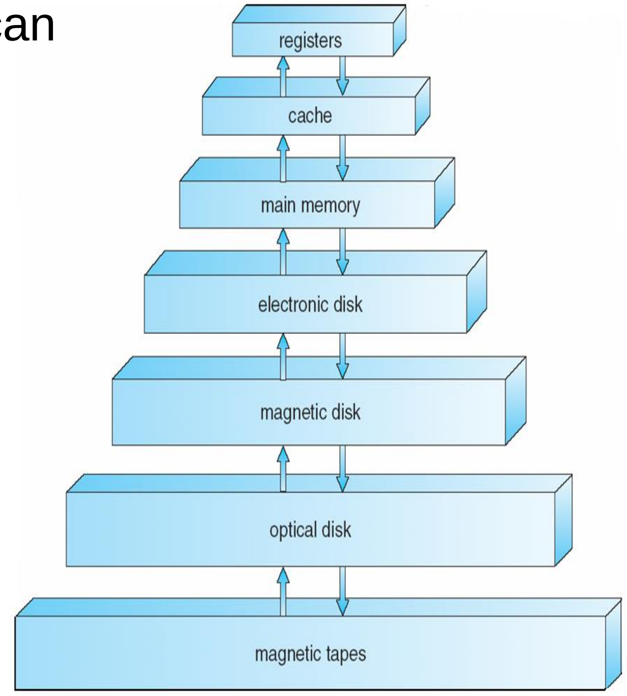
Chapter 9: Main Memory

Section 9.1-9.2,9.3.1



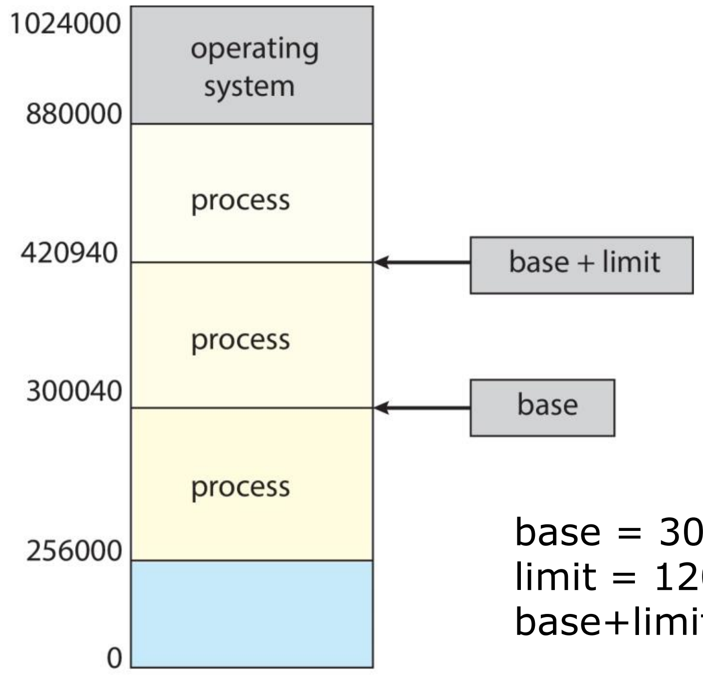
Background

- 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:
  - read requests & addresses, or
  - write requests & address + data
- Register access is done in one CPU cycle (or less)
- Main memory access can take many CPU cycles, causing processor to **stall**
- **Cache** is typically added on CPU chip to speedup memory access



Memory Protection

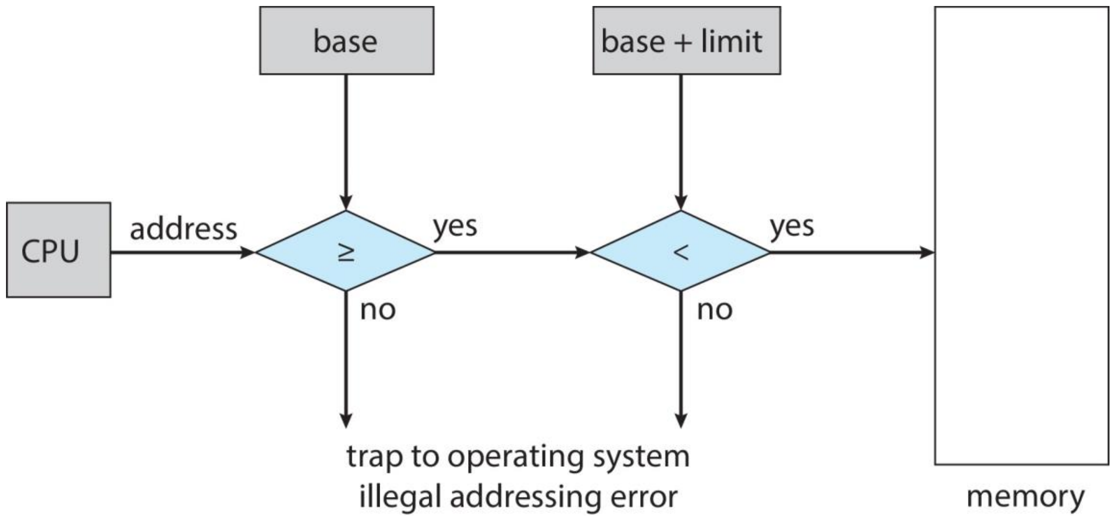
- Need to ensure that a process can access only those memory addresses in its address space
- We can provide this protection using **base** and **limit registers** that define the range of legal memory addresses that a process may access
  - **Base register** holds the smallest legal memory address
  - **Limit register** specifies the size of the range



base = 300040
limit = 120900
base+limit=420940

Memory Protection Hardware

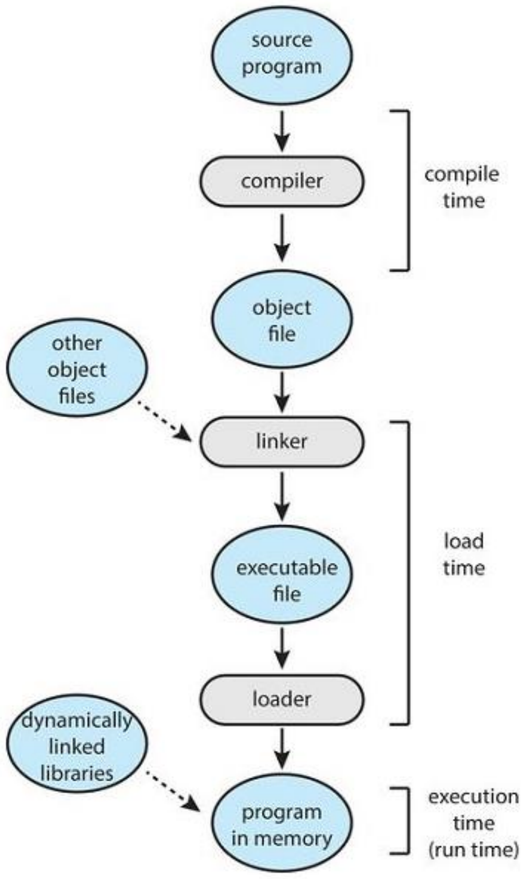
- CPU hardware must compare every memory access generated in user mode with base and limit registers



- Instructions to load base and limit registers are privileged instructions

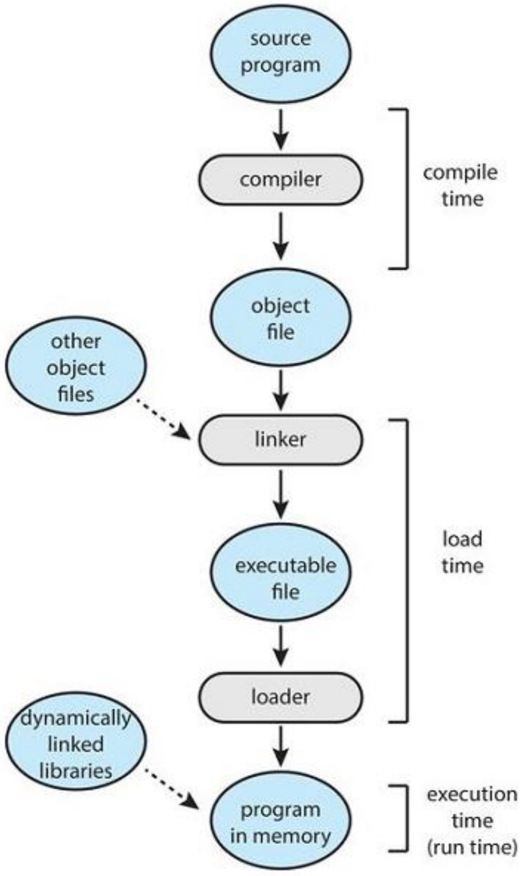
Address Binding (1)

- A user process may reside in any part of the physical memory
  - The first address of a user process need not be 0000
- Addresses represented in different ways at different stages of a program's life
  - Addresses in source program usually symbolic (e.g., `int count`)
  - A compiler **binds** symbolic addresses to relocatable addresses (e.g. "14 bytes from beginning of this module")
  - Linker or loader binds relocatable addresses to absolute addresses
    - e.g., relocatable address 14 is bound to absolute address 74014
  - Each binding maps one address space to another



Address Binding (2)

- Binding of instructions and data to memory addresses can happen at three different stages:
  - **Compile time**: If memory location of process known a priori, **absolute code** can be generated;
    - Must recompile code if starting location changes
  - **Load time**: Compiler must generate **relocatable code** if memory location of process is not known at compile time; final address binding is delayed until load time
    - If starting address changes, need reload the user code
  - **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 mapping (e.g., relocation register)
    - Used by most operating systems

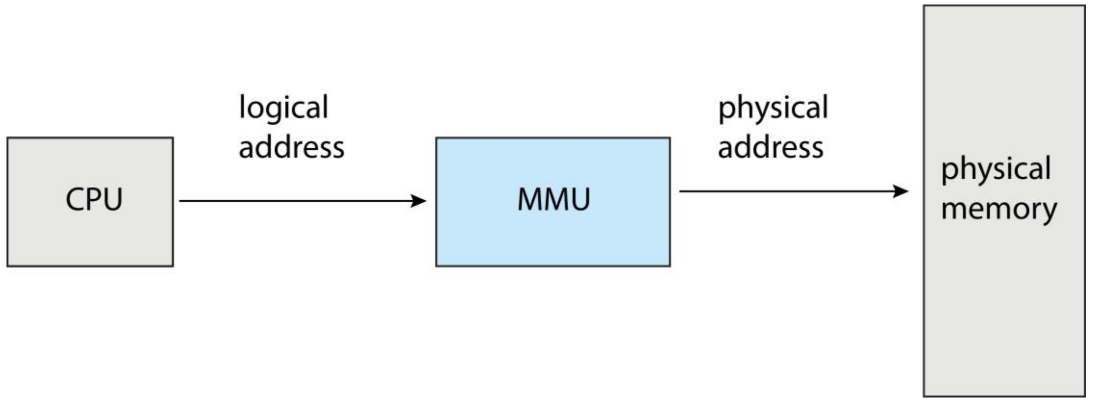


Logical vs. Physical Address Space

- 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 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
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

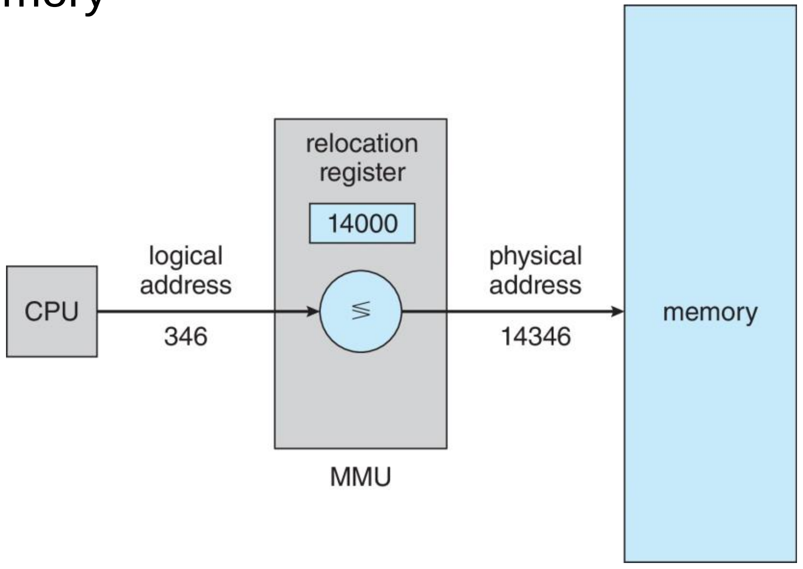
- MMU is a hardware device that maps logical addresses to physical addresses at run time



- Different mapping methods possible, covered in the rest of this chapter

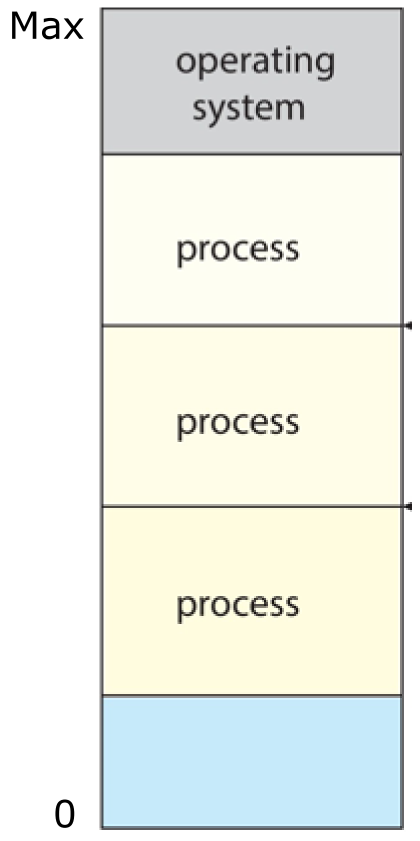
Memory-Management Unit (Cont.)

- Consider a simple mapping 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
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when a reference is made to a location in memory



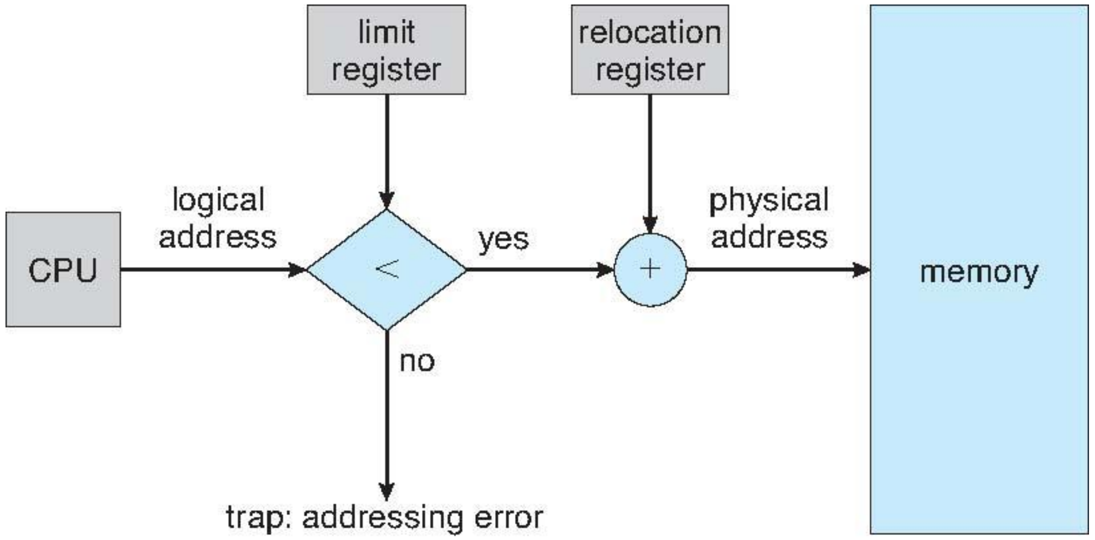
Contiguous Memory Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually divided into two **partitions**: one for OS and one for user processes
  - OS usually held in high memory
  - Each user process contained in a single contiguous section of memory



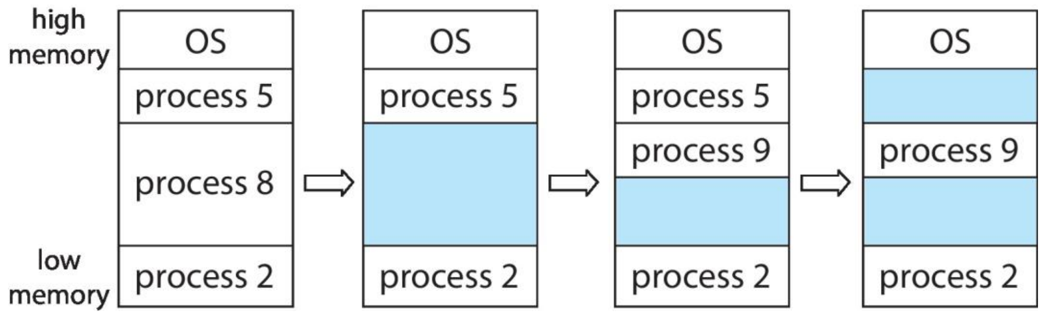
Contiguous Memory Allocation (Cont.)

- Relocation register and limit register used to protect user processes from each other, and from changing OS code and data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address to physical address *dynamically*



Memory Allocation

- Initially, all memory is available and is considered one large **hole**
  - A **hole** a is block of available memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - If the hole is too large, the unused part is returned to the set of holes
- When a process terminates, it releases its block of memory
  - If the new hole is adjacent to other holes, these holes are merged to form a larger hole
- In general, holes of various sizes are scattered throughout memory
- Operating system maintains information about a) allocated memory and b) free memory (holes)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of holes?

- Commonly used strategies:
  - **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, unless ordered by size
    - Produces the largest leftover hole

- First-fit and best-fit better than worst-fit in terms of storage utilization
- First-fit generally faster than best-fit

Example

Given six memory holes of

300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 125 KB (in order),

how would the first-fit, best-fit, and worst-fit algorithms place a process of size 115 KB?

- **First fit:** The process is put in 300-KB hole, leaving

  185 KB (=300KB-115KB), 600 KB, 350 KB, 200 KB, 750 KB, 125 KB

- **Best fit:** 115 KB is put in 125-KB hole, leaving

  300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 10 KB

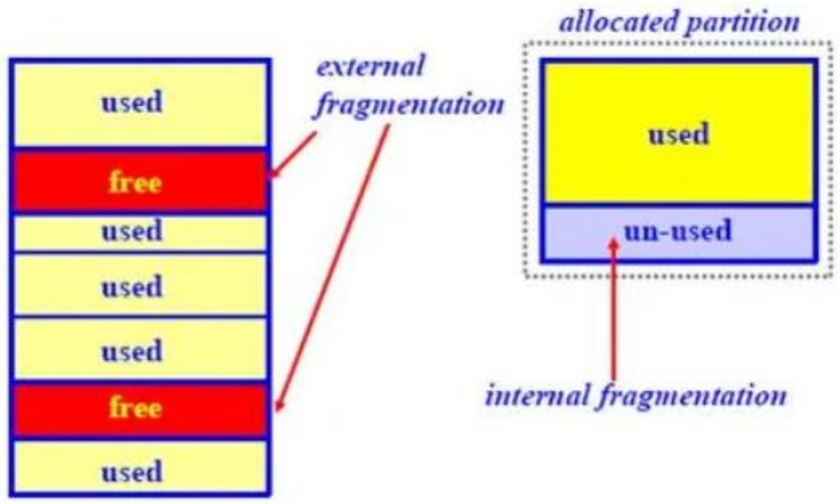
- **Worst fit:** 115 KB is put in 750-KB hole, leaving

  300 KB, 600 KB, 350 KB, 200 KB, 635 KB (=750KB-115KB), 125 KB

how would the first-fit, best-fit, and worst-fit algorithms place another process of size 200 KB?

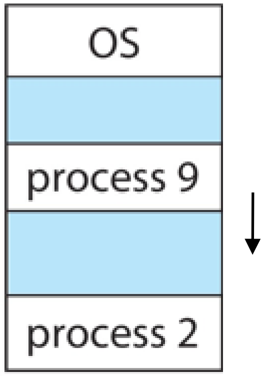
Fragmentation

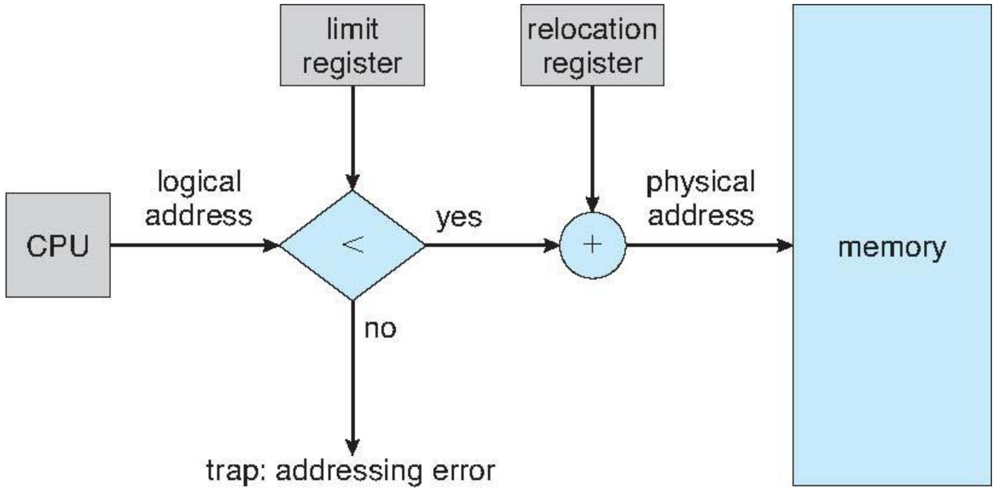
- **External Fragmentation** – there is enough total memory space to satisfy a request, but the available spaces are not contiguous
  - Statistical analysis of first-fit reveals that given *N* allocated blocks, another 0.5 *N* blocks will be lost to external fragmentation
  - That is, 1/3 of memory may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; the size difference is memory internal to a partition, but not being used
  - E.g., 4000 bytes requested, 4096 bytes (4KB) allocated



Overcoming External Fragmentation

- One solution to external fragmentation is **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, i.e., address binding done at execution time using a relocation register

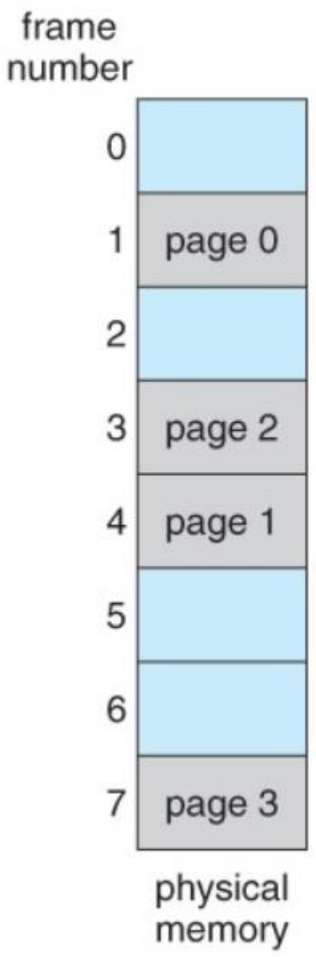




- Another solution is to permit the physical address space of a process to be noncontiguous
  - E.g., paging

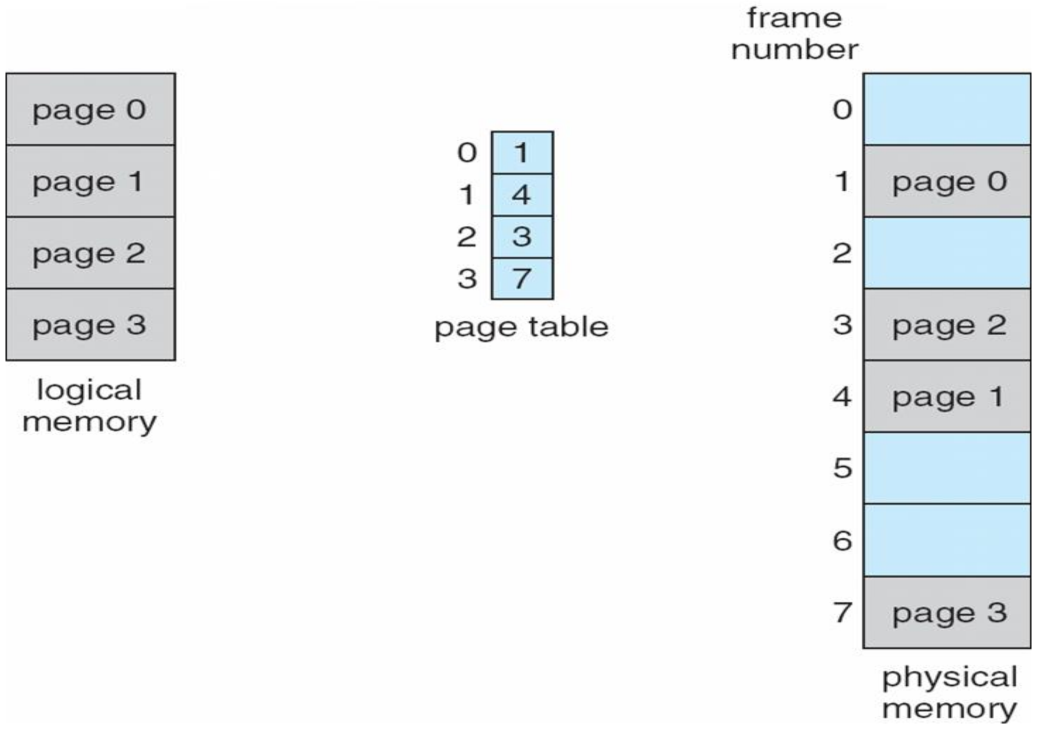
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames**
  - Frame size is power of 2, typically between 4KB (2^{12} bytes) and 1GB (2^{30} bytes)
- Divide logical memory into blocks of same size called **pages**
- To run a process of size ***N*** pages, need to find ***N*** free frames and load pages of process to frames
- OS keeps track of all free frames
- External fragmentation is avoided; still have internal fragmentation
- Paging is used in most operating systems



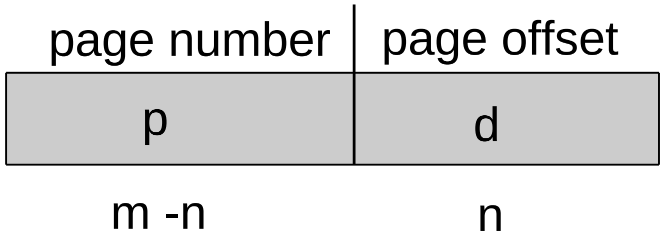
Paging

- Kernel maintains a **page table** for each process
  - Page table has an entry for each page of the process, entry indicates the memory frame allocated to the page
  - Page table is used to translate logical addresses to physical addresses

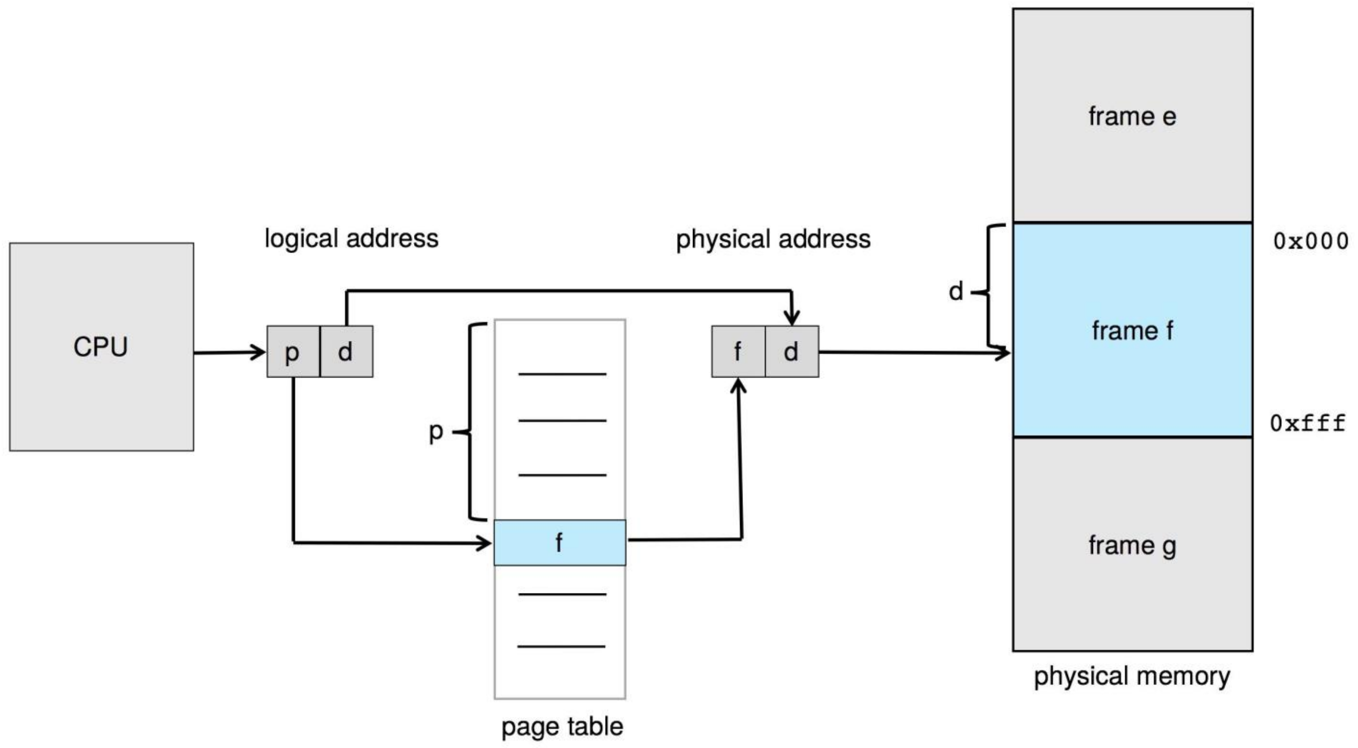


Address Translation Scheme

- Address generated by CPU (i.e., logical address) is divided into:
  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each frame in physical memory
  - **Page offset** (*d*) – location in the frame
  - Base address of frame is combined with the page offset to define the physical memory address that is sent to the memory unit
- If size of logical address space = 2^m bytes and page size = 2^n bytes, then
  - High-order m-n bits of logical address represent the page number
  - Low-order n bits represent the page offset



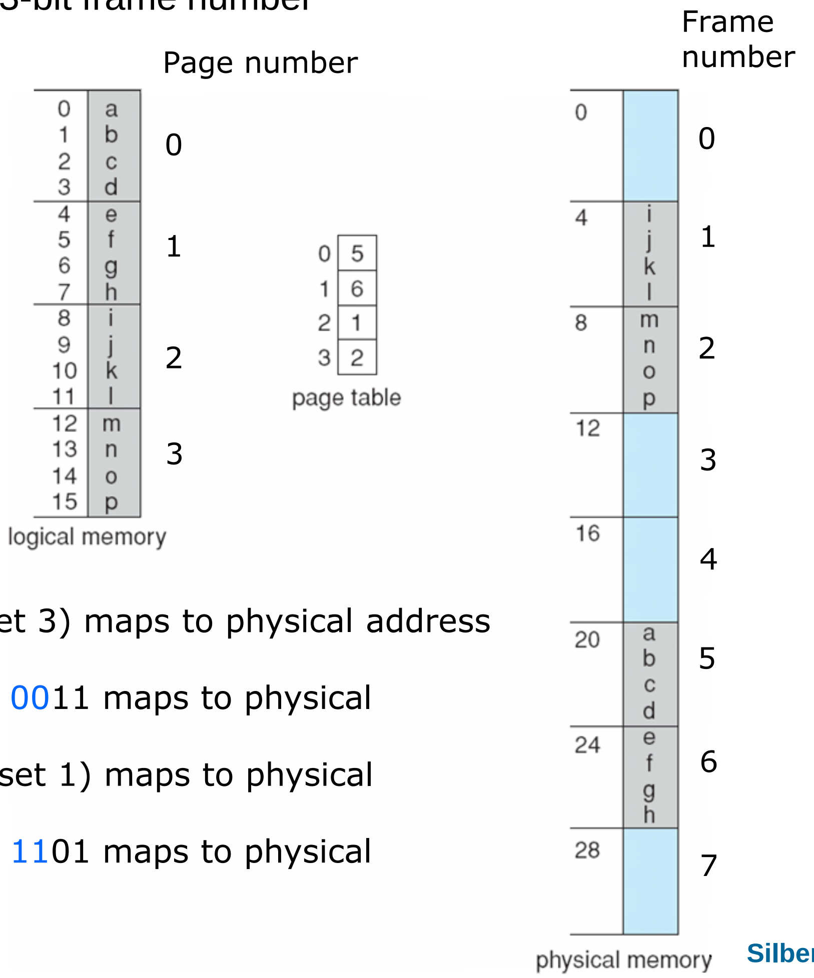
Paging Hardware



MMU translates a logical address to a physical address

Paging Example

- Page size = 4 bytes, physical memory = 32 bytes (i.e., 8 frames), logical memory = 16 bytes (i.e., 4 pages)
  - 2-bit page offset (n = 2)
  - 4-bit logical address (m = 4), 2-bit page number (m – n = 2)
  - 5-bit physical address, 3-bit frame number


Page number
Frame number
0 a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8 i 9 j 10 k 11 l 12 m 13 n 14 o 15 p
0 1 2 3
logical memory
0 5
1 6
2 1
3 2
page table
0 4 i j k l 8 m n o p 12 16 20 a b c d 24 e f g h 28
0 1 2 3 4 5 6 7
physical memory

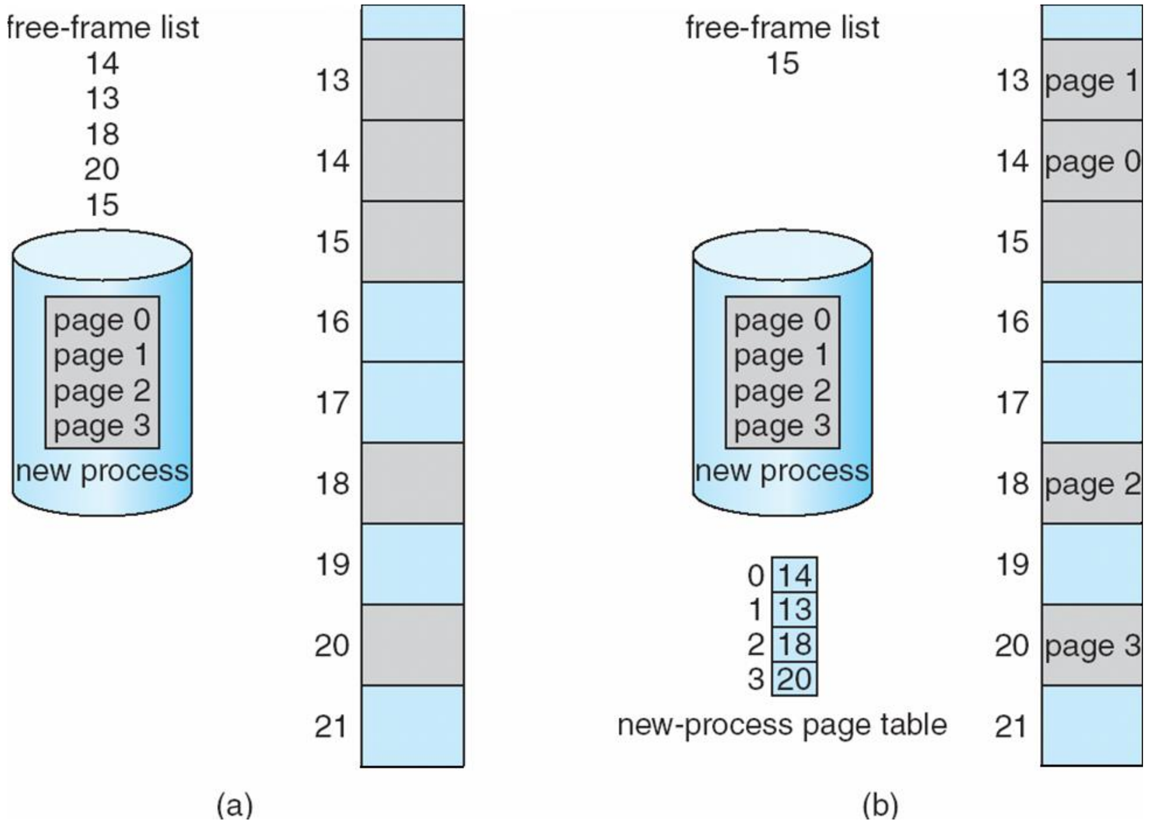
- Logical address 3 (page 0, offset 3) maps to physical address 5*4+3=23
  - In binary, logical address 0011 maps to physical address 10111
- Logical address 13 (page 3, offset 1) maps to physical address 2*4+1=9
  - In binary, logical address 1101 maps to physical address 01001

Internal Fragmentation in 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 happens when a process needs n pages plus 1 byte → internal fragmentation is almost 1 frame
  - On average internal fragmentation = 1/2 frame size
- So small page sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes have grown over time
  - Today page sizes typically 4KB or 8KB

Free Frames

- OS keeps track of free frames



Before allocation

After allocation