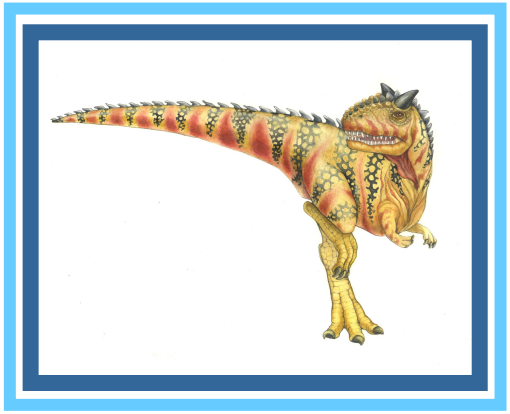
Chapter 8: Main Memory



Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Objectives

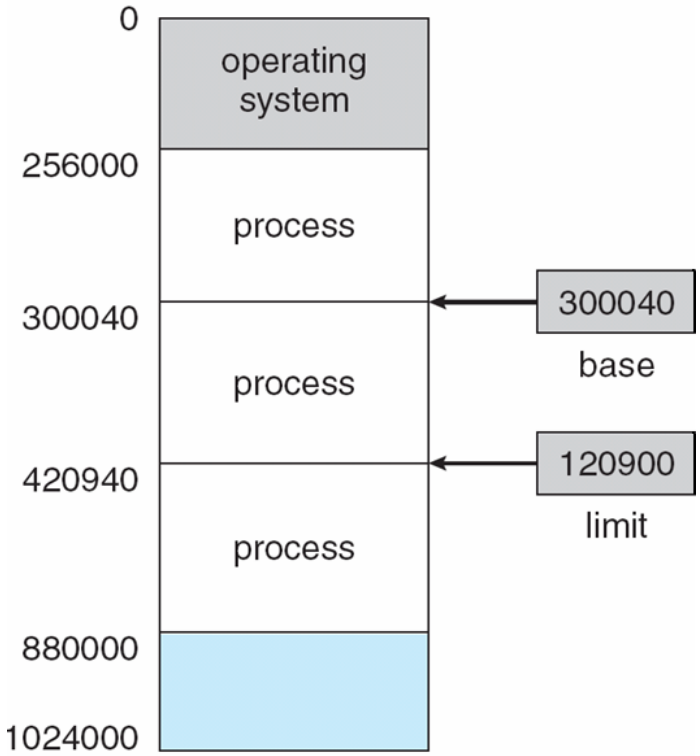
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

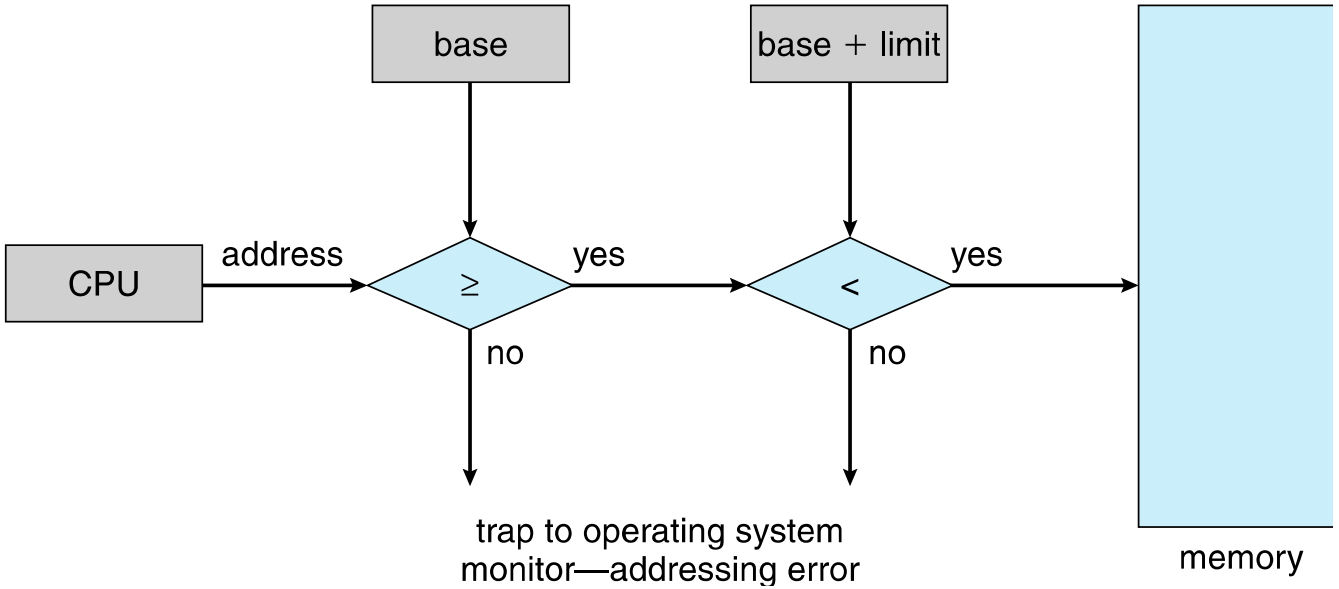
- Main memory and registers are the only storage CPU can access directly
- Program must be brought (from disk) into memory for it to be run
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
  - Protect OS from user programs
  - Protect user programs from each other

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- Only OS can load base and limit regs, using privileged instructions



Hardware Address Protection



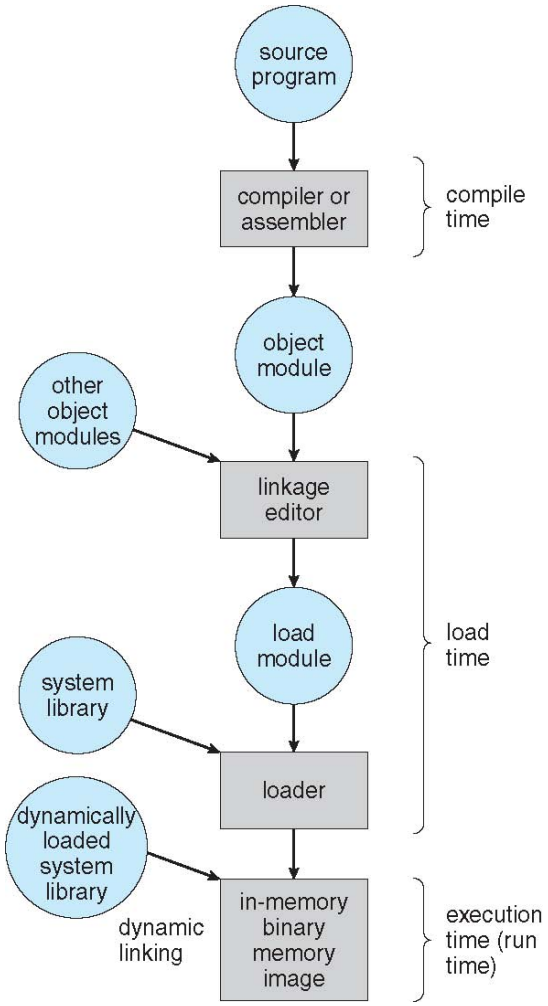
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
- Inconvenient to have first user process physical address always at 0000
- 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"
  - 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 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



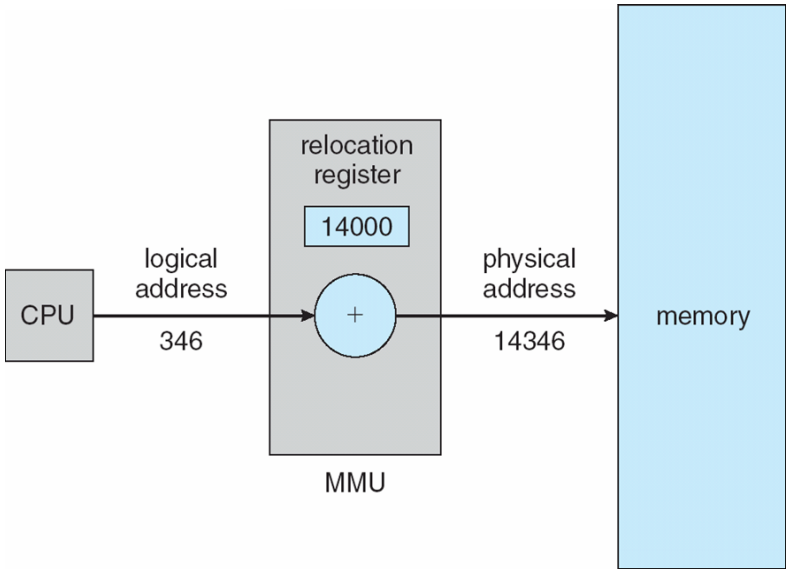
Logical vs. Physical Address Space

- 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 addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps a virtual address to a physical address
- 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**
- 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 linking using a relocation register



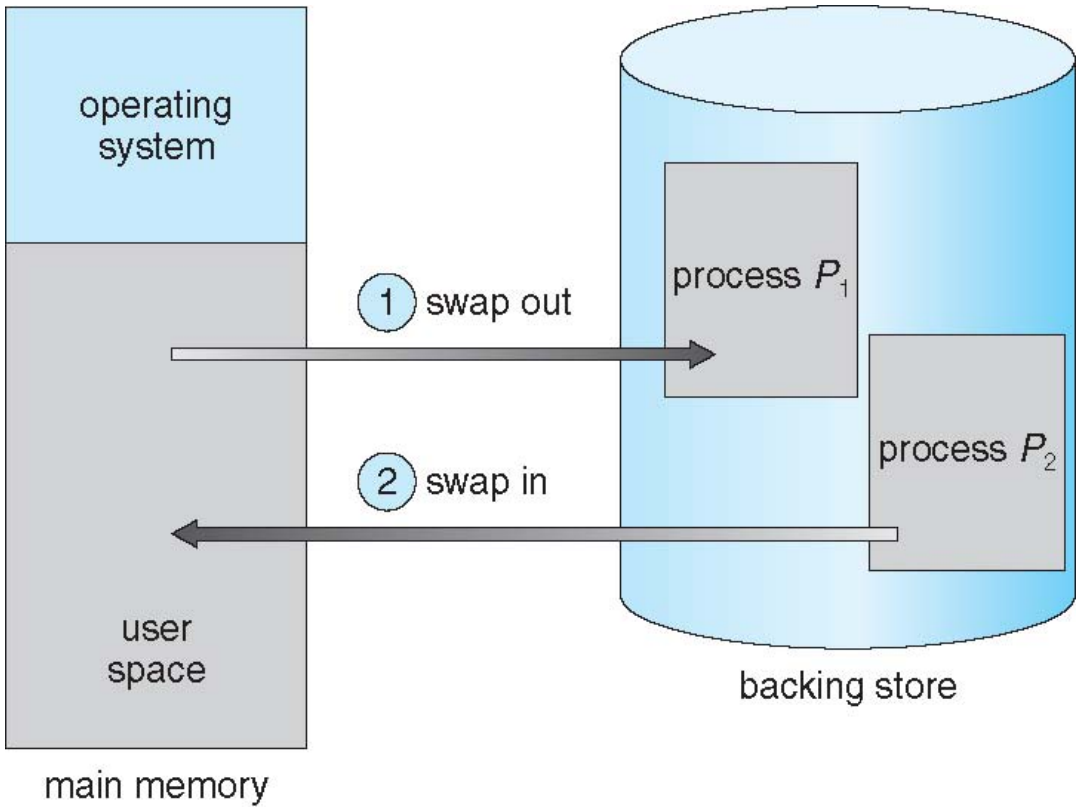
Dynamic Linking

- **Static linking:** system libraries and program code combined by the linker and the loader into the binary program image
- **Dynamic linking:** linking postponed until execution time
- System libraries also known as **shared libraries**
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine or load it if not already present
- Stub replaces itself with the address of the routine, and executes the routine
- Next time the routine is referenced, no dynamic linking cost
- Dynamic linking is particularly useful for libraries
- Consider applicability to patching system libraries
  - Versioning may be needed

Swapping

- Total physical memory space of processes can exceed physical memory
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Schematic View of Swapping



Problems with Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4 seconds
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to process
    - Known as **double buffering**, adds overhead

Swapping in Modern Systems

- Standard swapping not used in modern operating systems
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if memory demand exceeds a certain threshold
  - Disabled again once memory demand drops below threshold

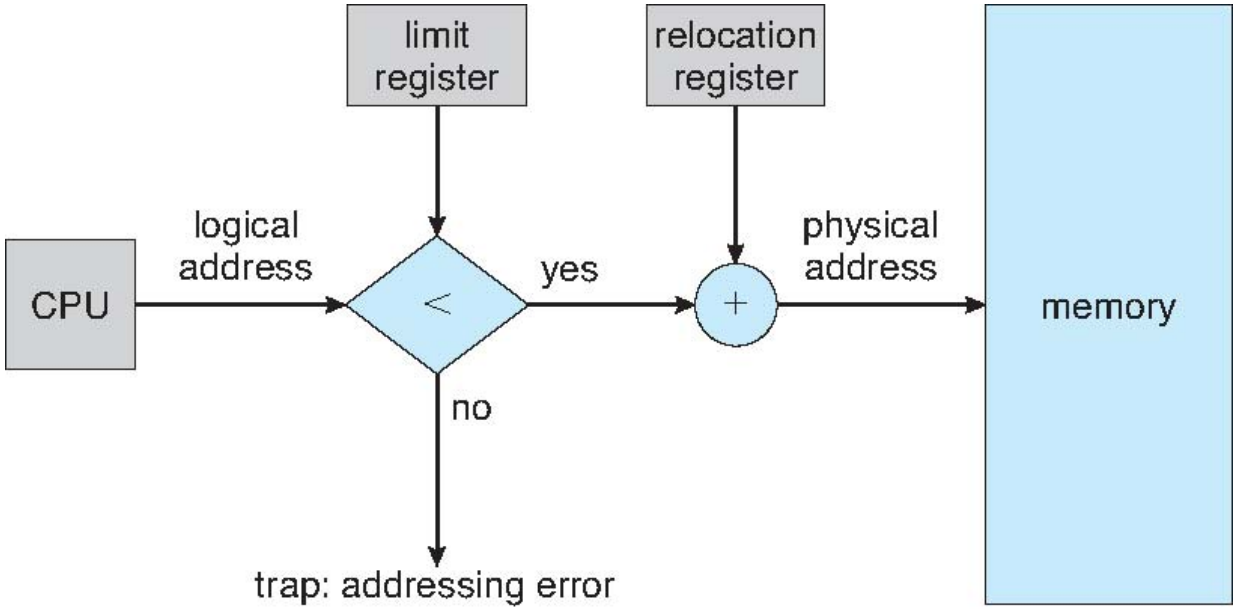
Contiguous Allocation

- 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 held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

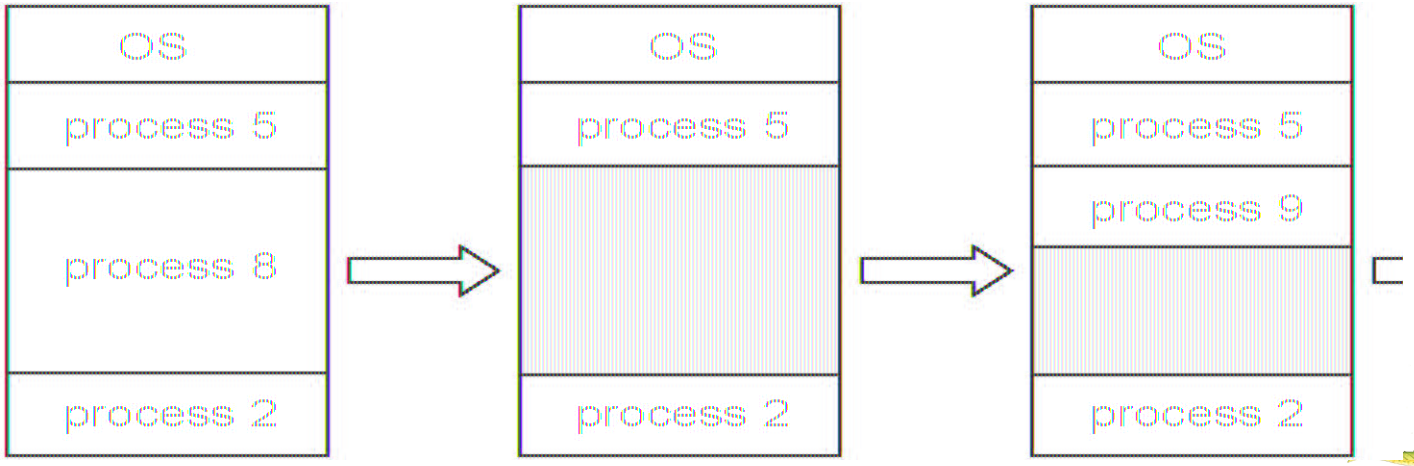
- Relocation registers used to protect user processes from each other and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Correct values are loaded into relocation and limit registers as part of context switching
  - Can then allow actions such as changing kernel size dynamically
    - E.g. Code that is not commonly used can be made **transient**

Hardware Support for Relocation and Limit Registers



Multiple-partition Allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** 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 (holes)



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 storage utilization

First-fit is faster than best-fit

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

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
- More effective solutions to fragmentation permit logical address space of a process to be non-contiguous
  - Segmentation
  - Paging

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A compiler might create separate segments for the following:

    The code

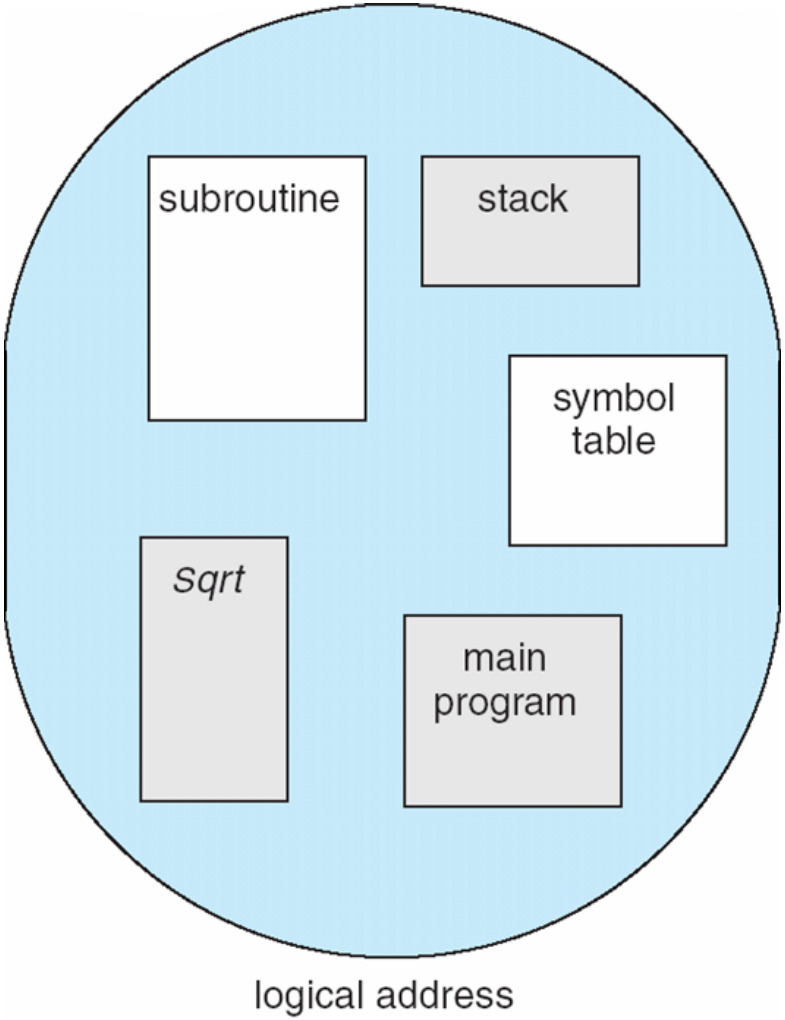
    Global Variables

    The heap (dynamic memory allocation)

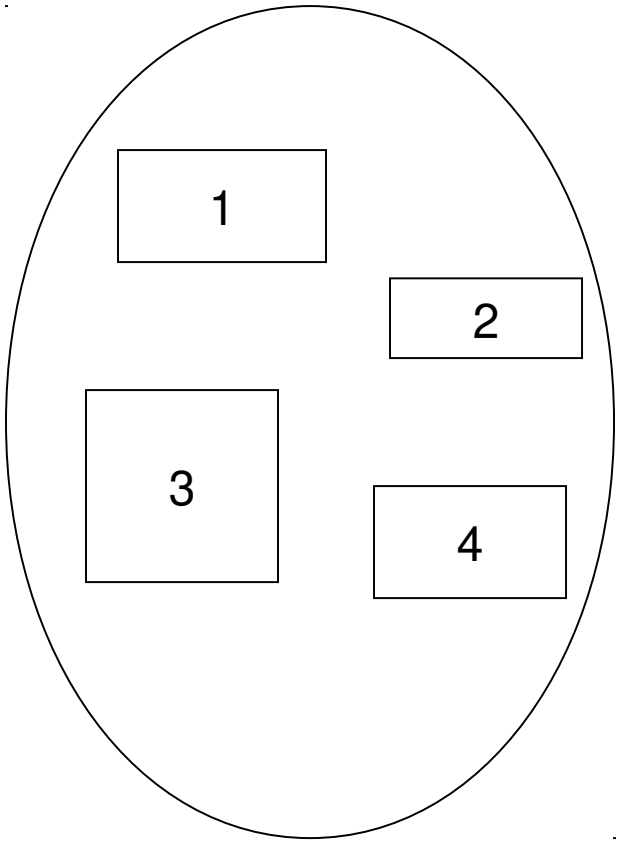
    Stacks used by threads

    Language Library

User's View of a Program



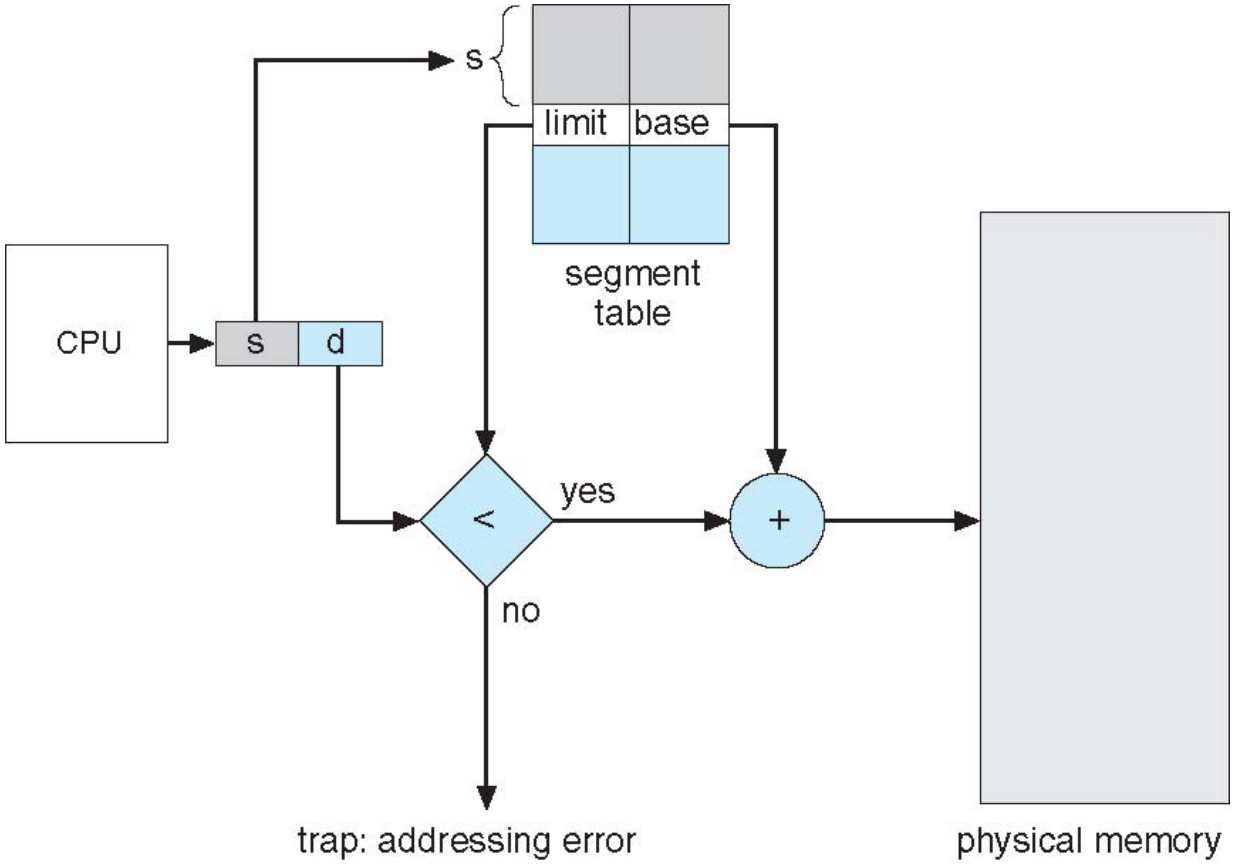
Logical View of Segmentation



user space

physical memory space

Segmentation Hardware



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

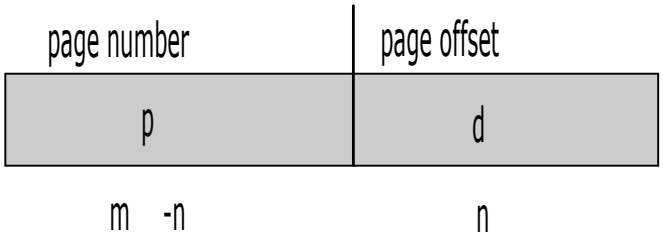
- **Segment table** – maps two-dimensional user addresses into one-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segment resides in memory
  - **limit** – specifies the length of the segment

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation and the need for compaction
  - Avoids problem of varying sized memory chunks
  - Still has internal fragmentation
- 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

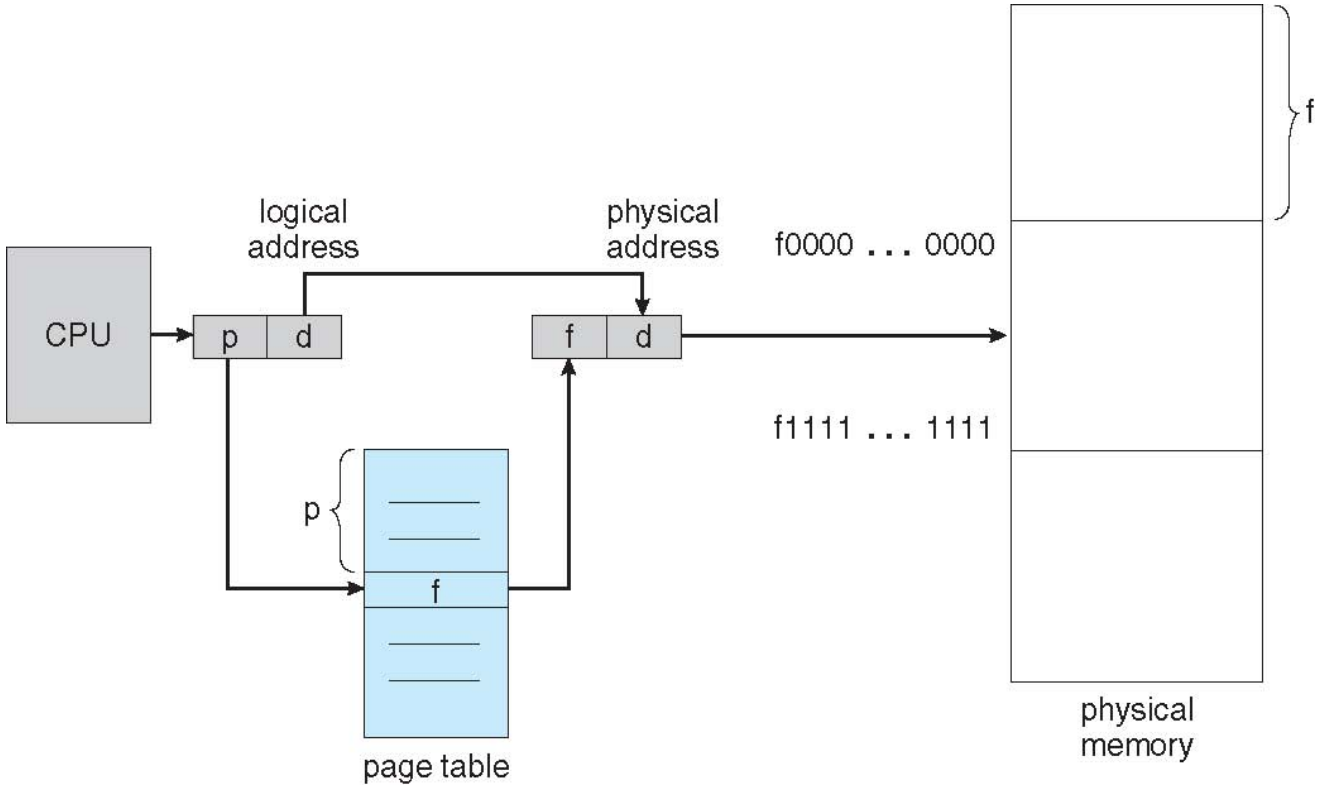
Address Translation Scheme

- 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

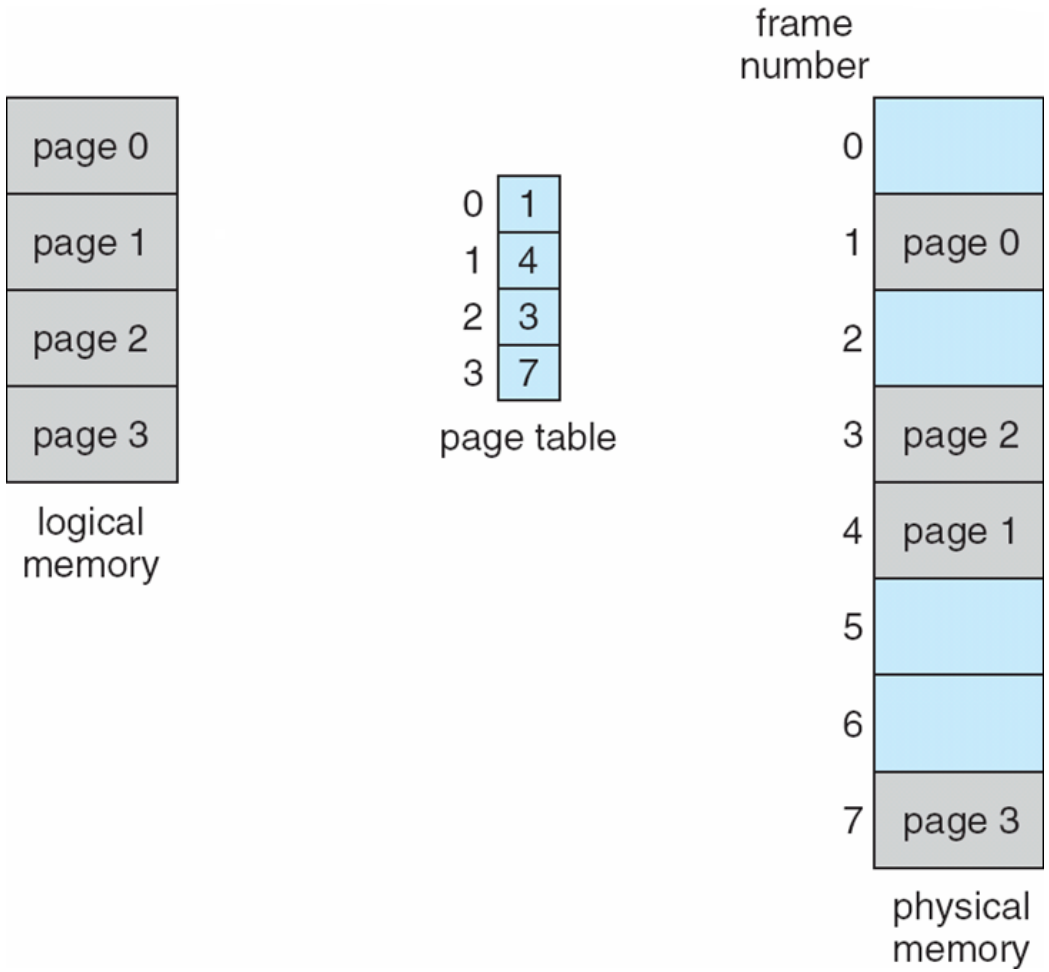


  - For given logical address space 2^m and page size 2^n
    - E.g. m=32, n=10 (page size = 1K)

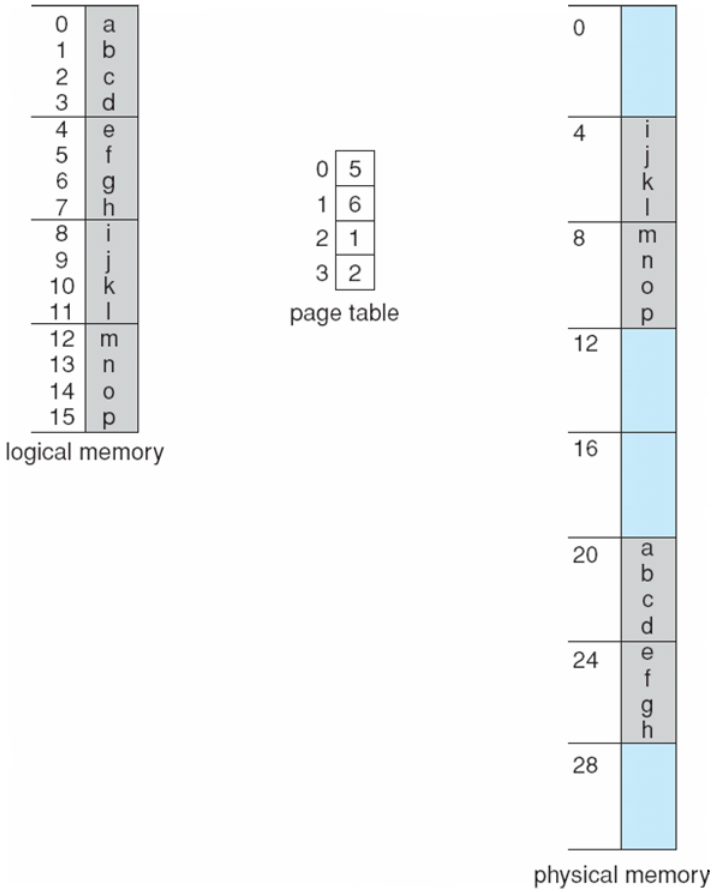
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



4-byte pages, 16-byte logical memory, 32-byte physical memory

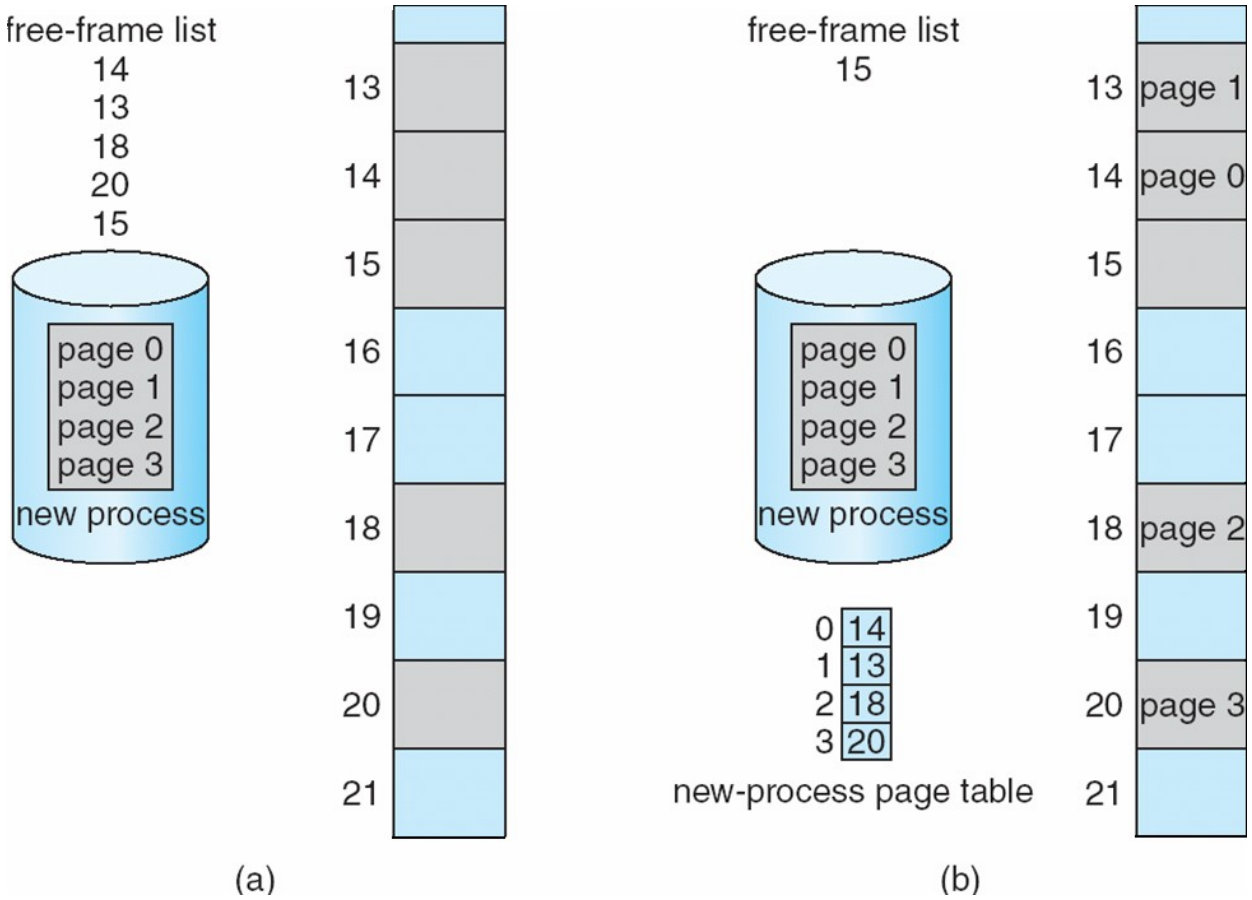
Internal Fragmentation

- Calculating internal fragmentation
  - Page size = 512 bytes
  - Process size = 2060 bytes
  - 4 pages + 12 bytes
  - Internal fragmentation of 512 - 12 = 500 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So, using a smaller frame size reduces fragmentation, but each page table entry takes memory to track -> tradeoff!
  - Page sizes growing over time
    - Typically 4 KB to 8 KB
    - Some systems support larger frames

Addressing

- How much memory can we address with 4-byte page table entries?
  - 32-bit entry can address up to 2^{32} page frames
  - If the frame size is 4KB (2^{12}), then we can address 2^{44} bytes (16 TB)
- Actual size will be smaller, because other information will have to be kept in a page table entry
- Process view and physical memory now very different
  - Process view is contiguous, while it is actually scattered throughout physical memory
- By implementation, process can only access its own memory
  - A process has no way of addressing memory outside of its page table
  - Its page table incudes only the frames that the process owns

Free Frames



Before allocation

After allocation

Implementation of Page Table

- If the page table is small enough, it can be implemented as registers
  - But, this is not feasible on most modern systems that need large page tables
- So, 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 (number of entries)
- 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 buffer** (**TLB**) which is implemented using **associative memory (high speed).**

Implementation of Page Table (Cont.)

- TLB has a key or tag (page number) and a value (frame number).
- TLBs typically small (32 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 at every context switch

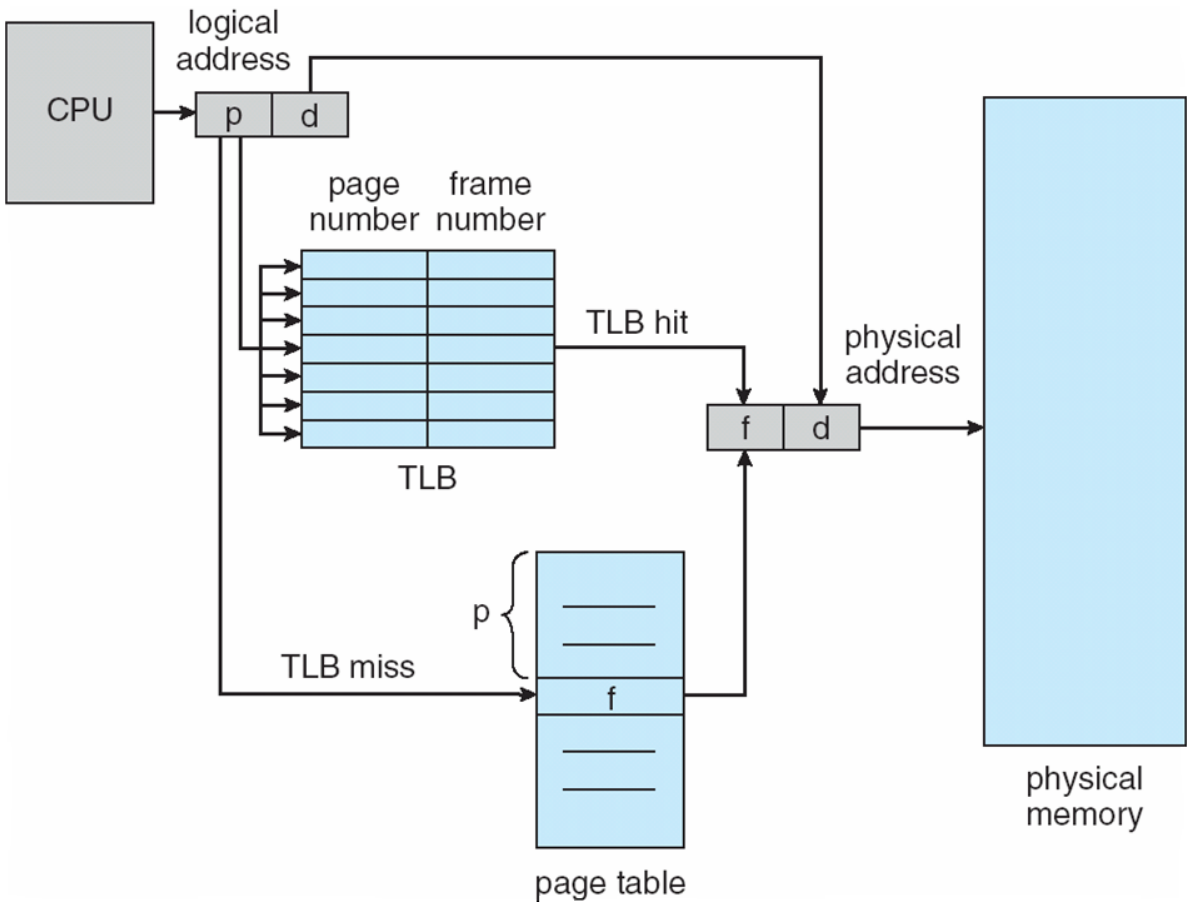
Associative Memory

- 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
  - Much smaller than 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**)

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

  M: memory access time
- Consider α = 80%, ε = 2ns for TLB search (negligible), 100ns for memory access:
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε = 2ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

End of Chapter 8