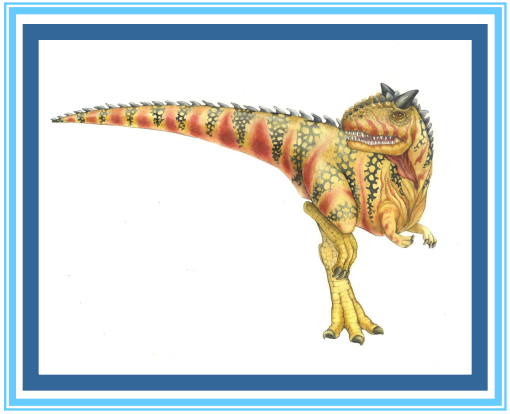
Module V:
Memory Management



Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging

Background

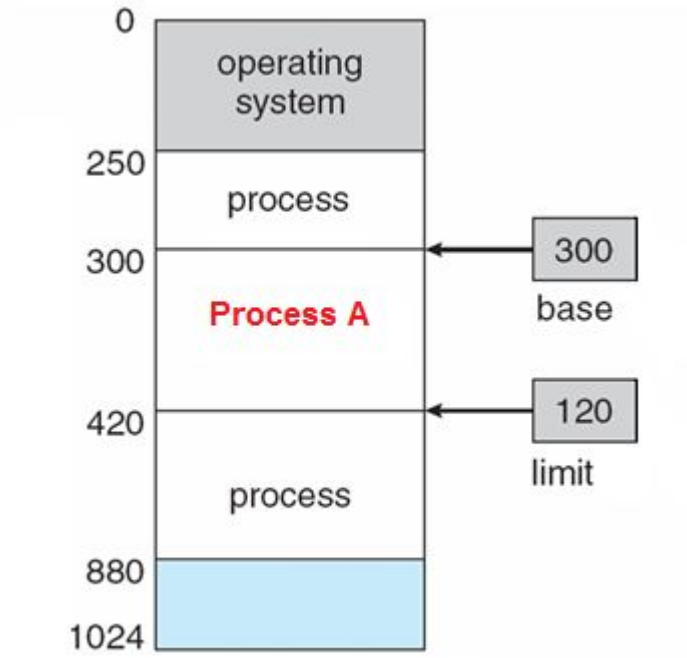
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Memory unit only sees
  - a stream of addresses + read requests, OR
  - address and data + write requests
- Main memory and registers are only storage CPU can access directly

Background

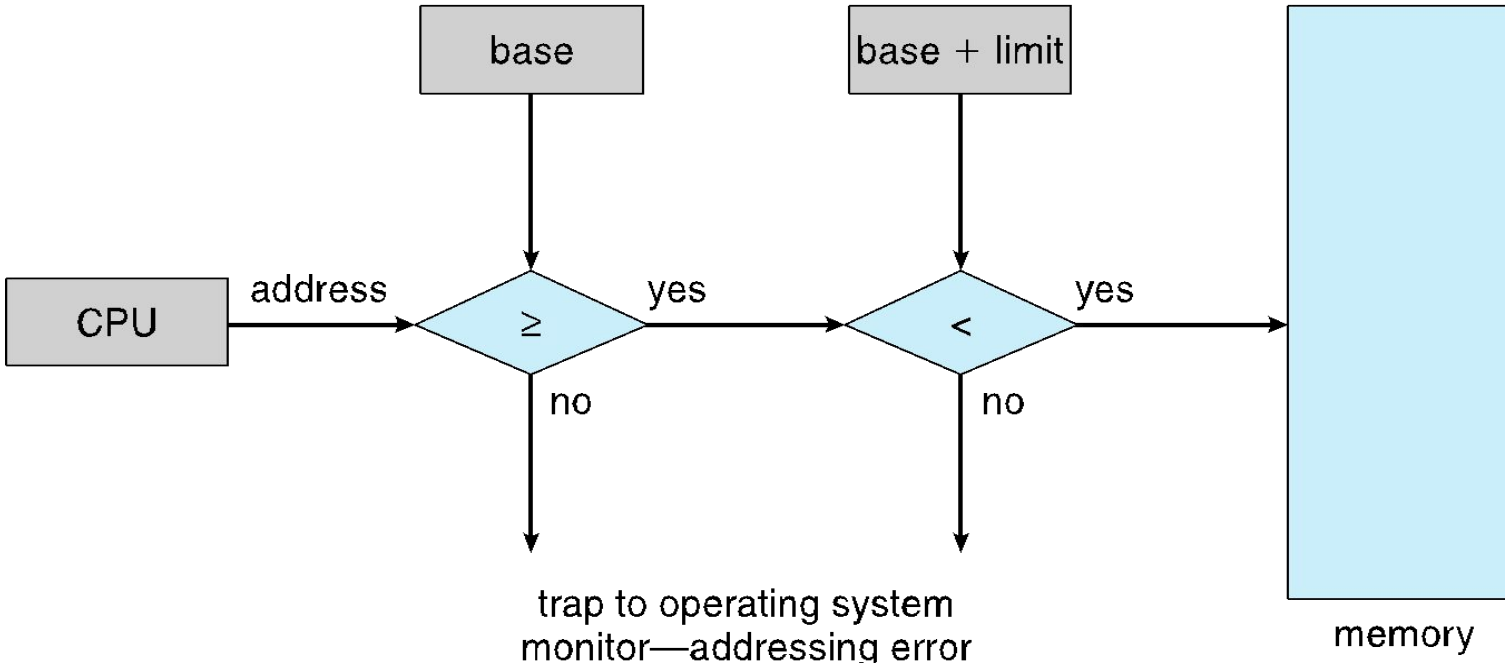
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

- 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



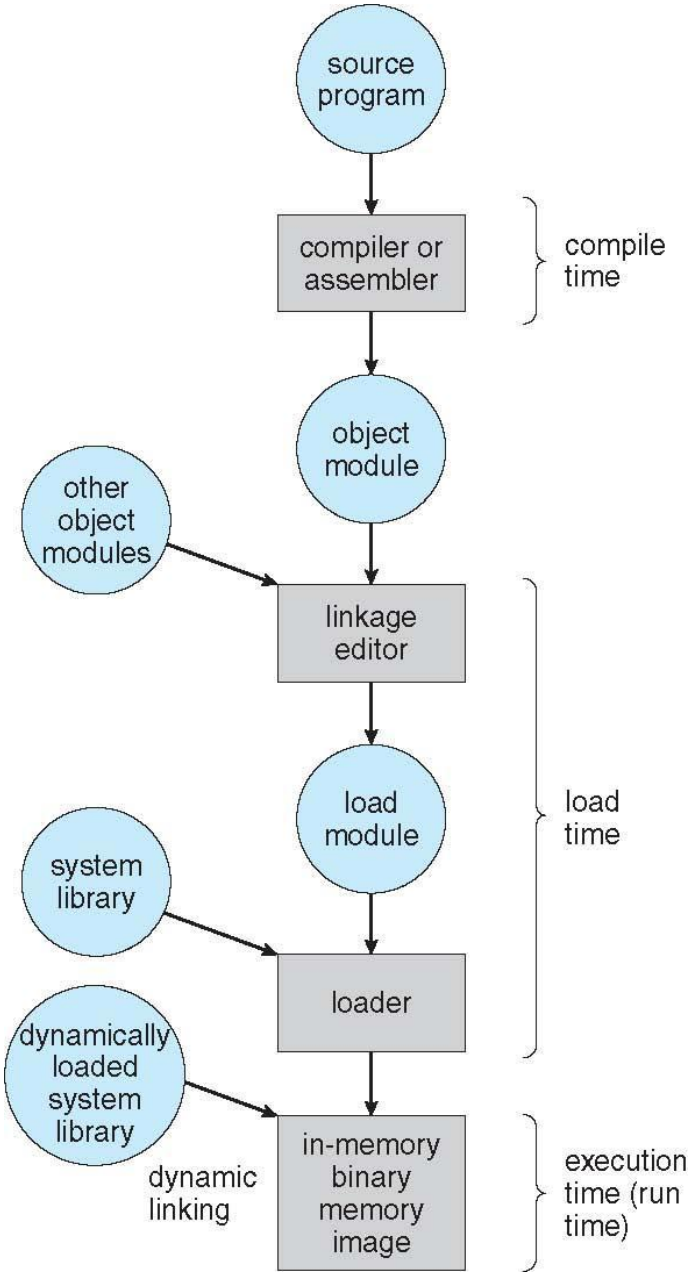
Hardware Address Protection



Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 00000
- 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. 314
  - Each binding maps one address space to another

Multistep Processing of a User Program

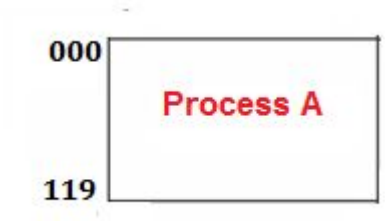


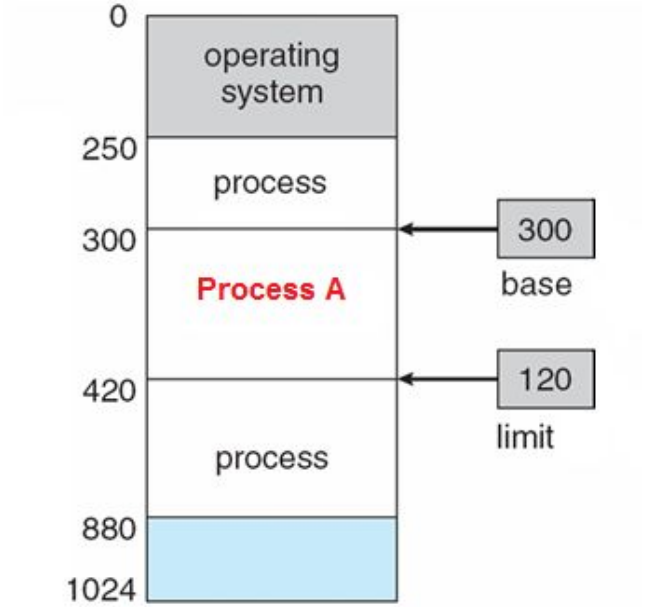
Binding of Instructions and Data to Memory

- 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)

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 vs. Physical Address Space

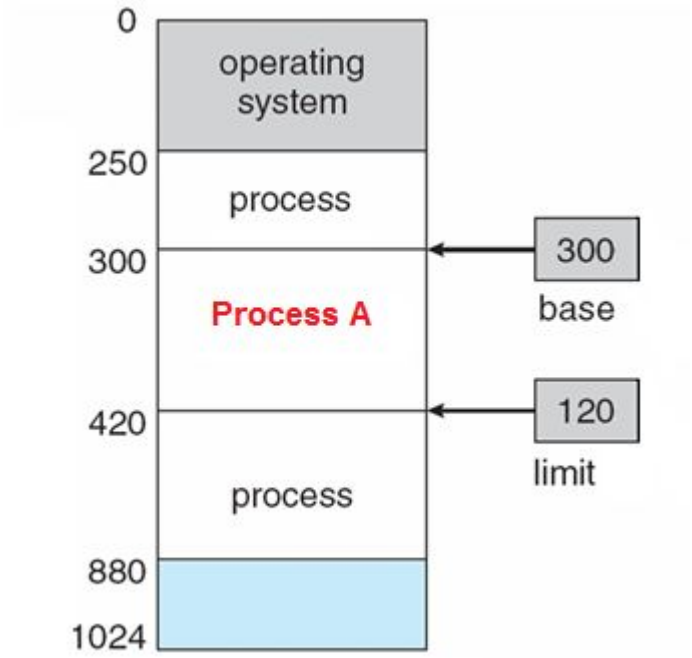
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (**virtual**) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

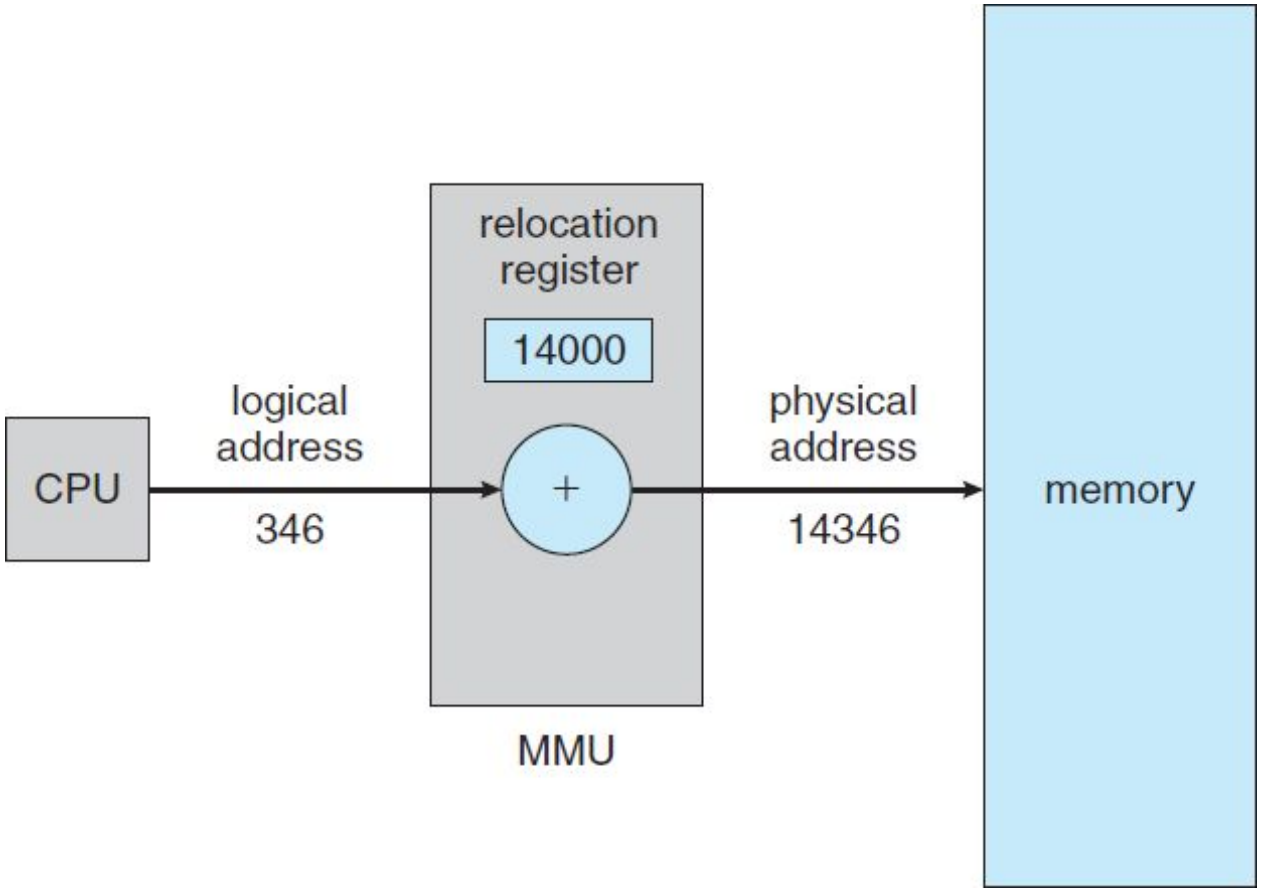
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **Memory-Management Unit (MMU)**.
- Many methods possible, covered in the rest of this chapter

Memory-Management Unit (MMU)

- 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**



Memory-Management Unit (MMU)



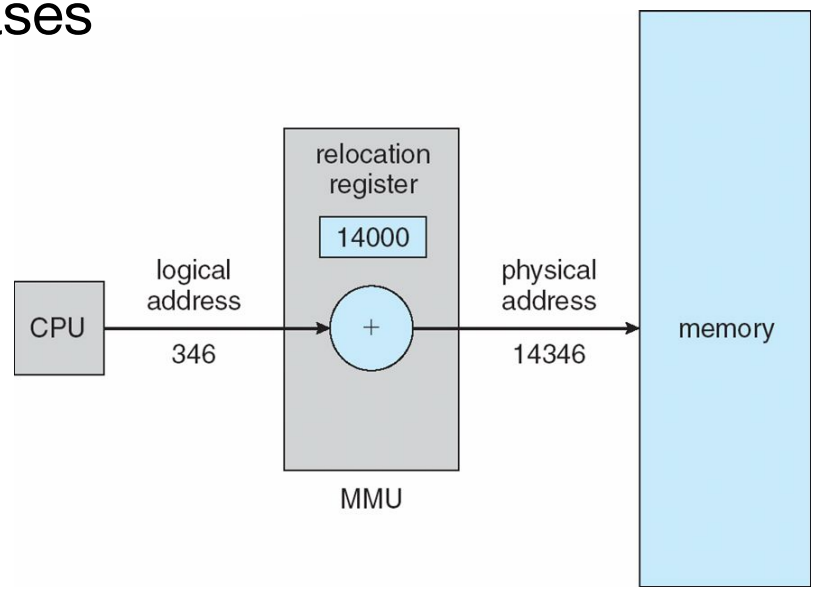
Dynamic relocation using a relocation register

Memory-Management Unit (MMU)

- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic Loading

- All routines kept on disk in relocatable load format
- Routine is not loaded until it is called
- Known as **Dynamic Loading**
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases



Dynamic Linking

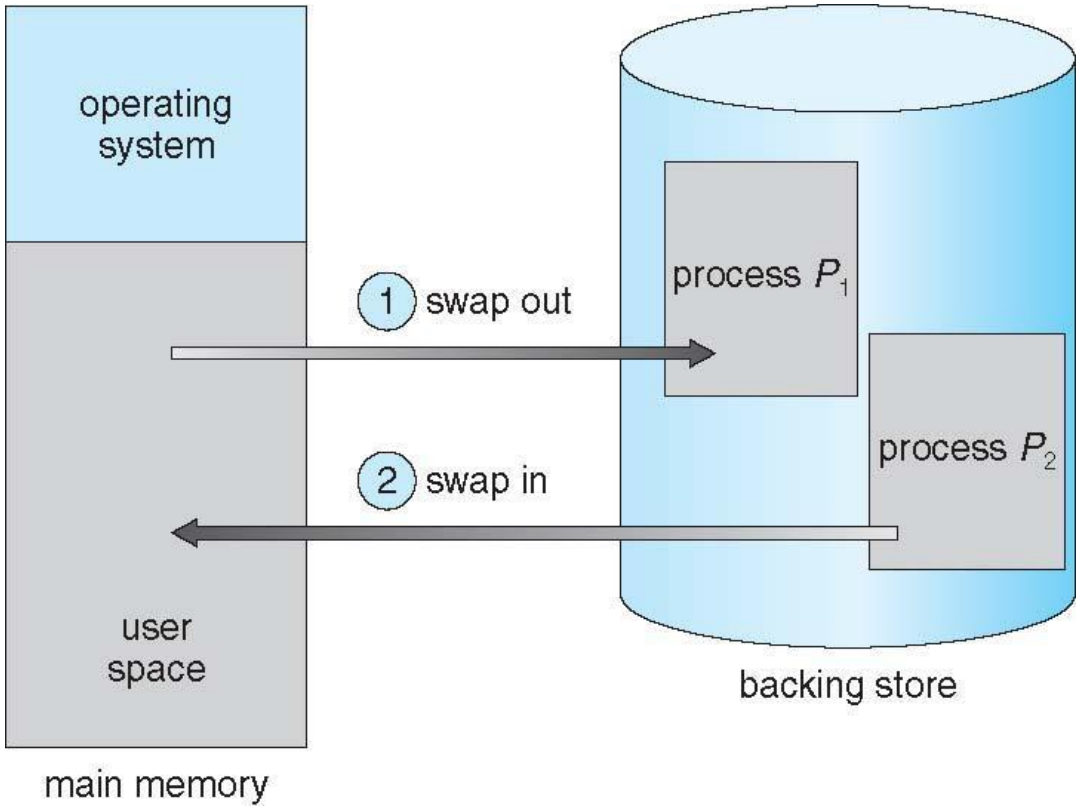
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** – linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the 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 that are often updated. Such system is also known as **shared libraries**

SWAPPING

Swapping

- A process must be in memory to be executed
- But it 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

Schematic View of Swapping



Swapping

- System maintains a **ready queue** of ready-to-run processes which have memory images on the backing store or in memory and are ready to run
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Context Switch Time including 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 2 seconds (2000 ms), Plus swap in of same sized process
  - Total context switch time of 4 seconds (4000ms)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used

Other factors affecting Swapping

- To swap a process, it must be completely idle
- If any pending I/O
  - Assume an I/O operation is queued for P1 (swapped out)
  - then I/O operation might attempt to use memory of P2
- Two solutions
  1. Never swap a process with pending I/O
  2. execute I/O operations only into OS buffers and transferred to process memory only when process is swapped in
     - Known as **double buffering**, adds overhead

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to Or 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

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support **Paging** as discussed later

CONTIGUOUS MEMORY ALLOCATION

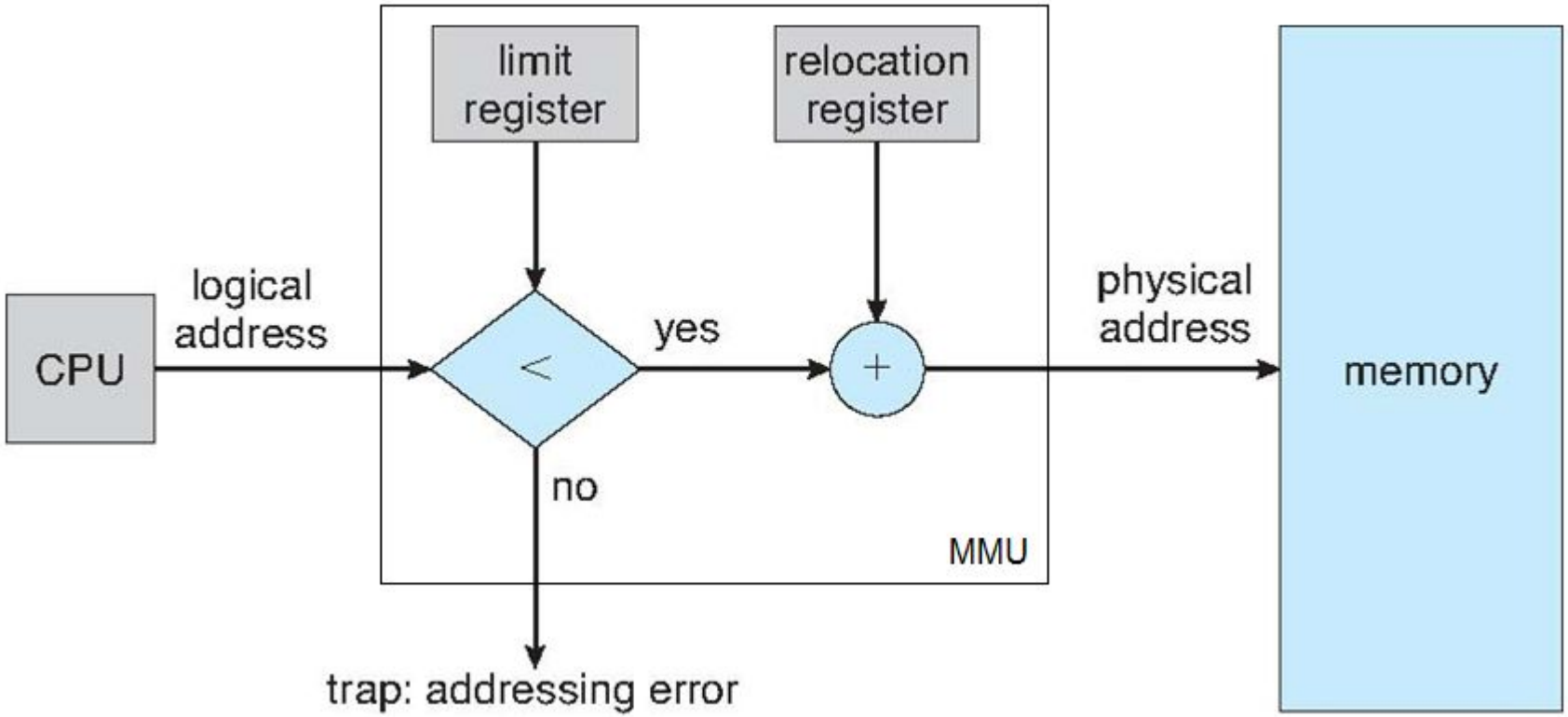
Contiguous Memory Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually 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

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*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

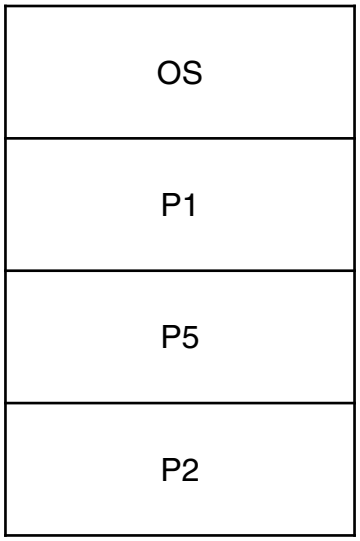
Memory Protection



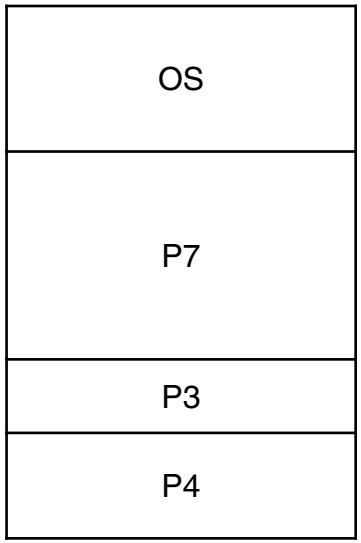
Hardware Support for Relocation and Limit Registers

Memory Allocation

- Two schemes of memory allocation:
  - **Fixed-sized partitions (multiple partition method)**
    - Degree of multiprogramming limited by number of partitions
  - **Variable-partition scheme**



Fixed-Sized

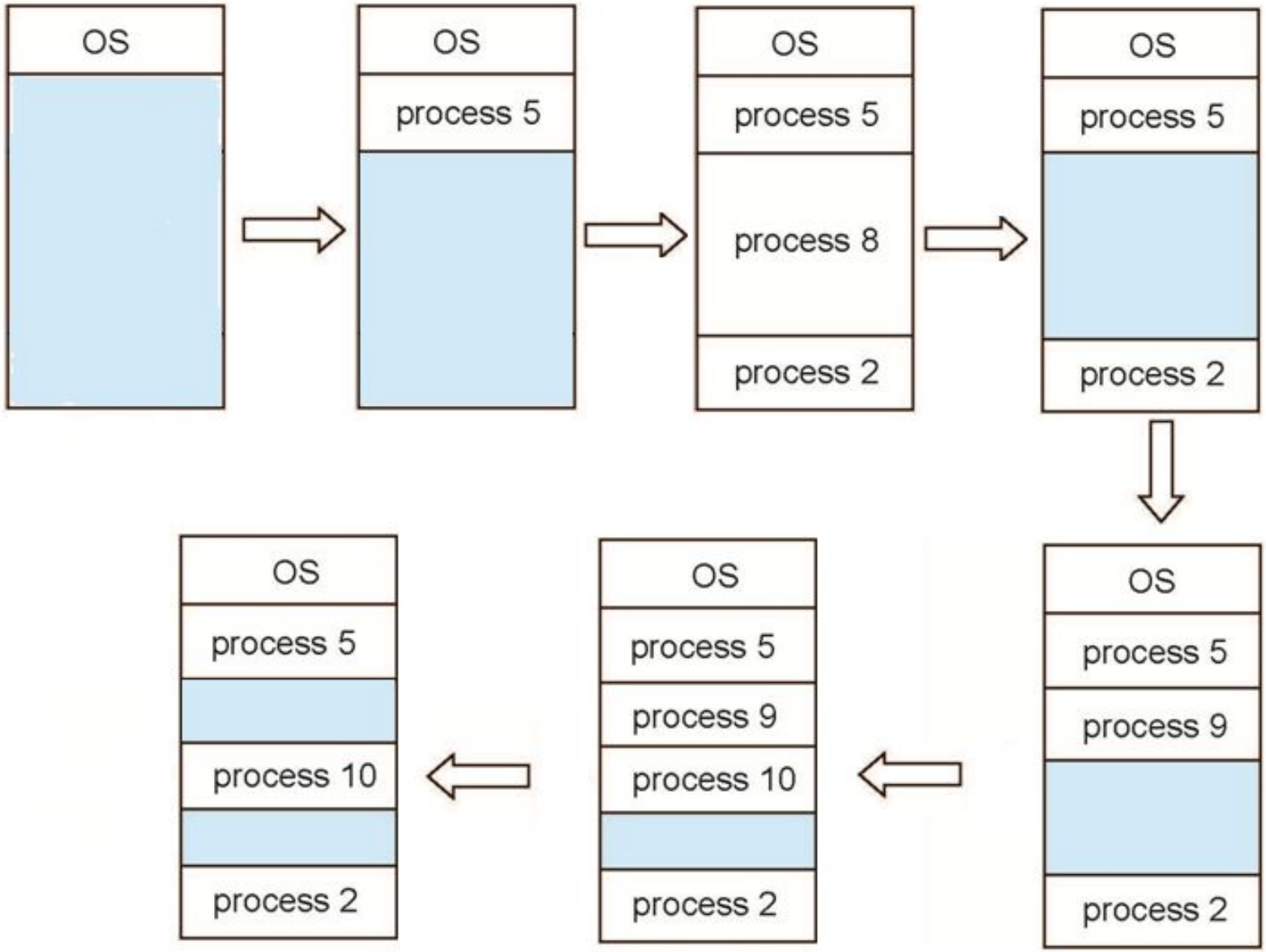


Variable-Sized

Memory Allocation

- **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)


OS
OS
process 5
OS
process 5
process 8
process 2
OS
process 5
process 2
OS
process 5
process 9
process 2
OS
process 5
process 9
process 10
process 2
OS
process 5
process 10
process 2

Dynamic Storage-Allocation Problem

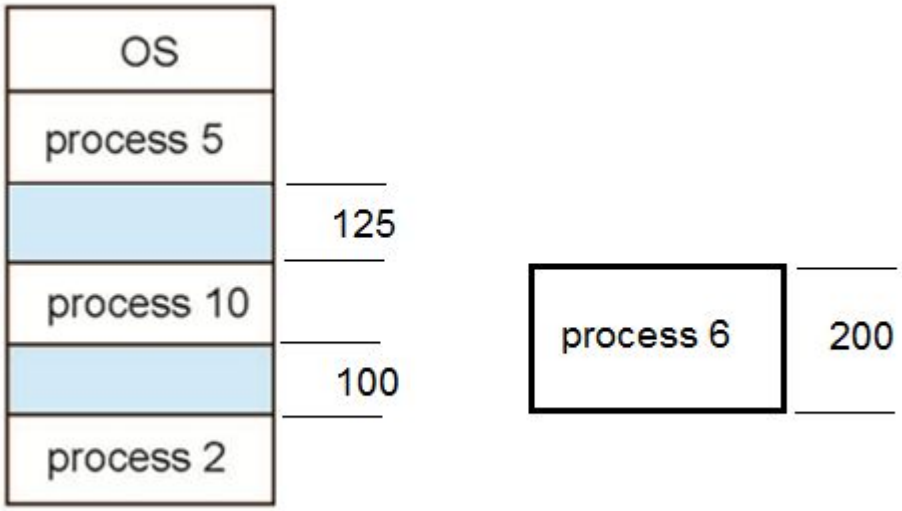
How to satisfy a request of size ***n*** from a list of free holes?

- **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

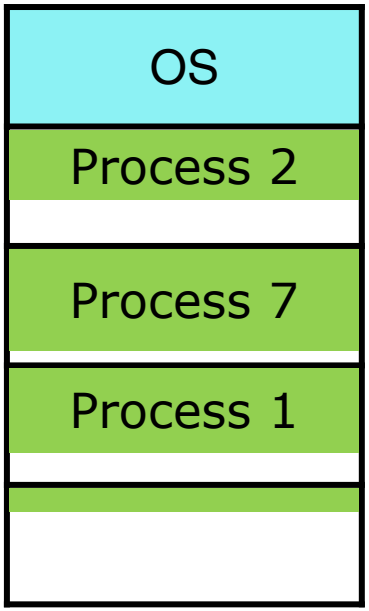
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous



Fragmentation

- **External Fragmentation**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used



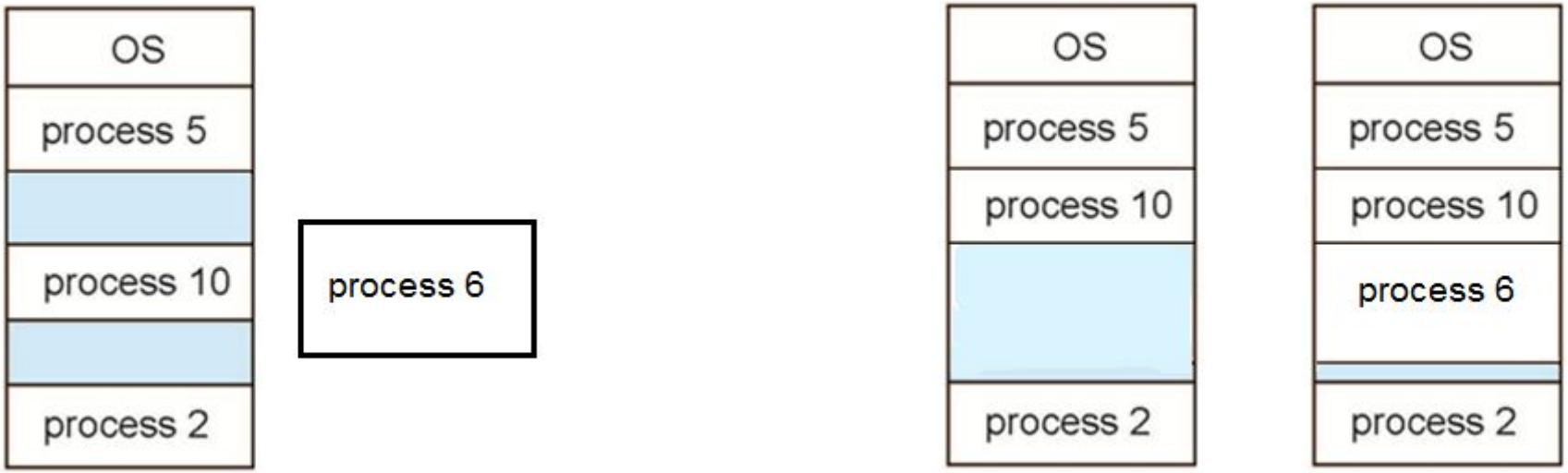
Fragmentation

- **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

- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block



After Compaction

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
  - 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

Fragmentation (Cont.)

- Another possible solution to the external-fragmentation: noncontiguous memory
- Thus allowing a process to be allocated physical memory wherever such memory is available.
- Two complementary techniques achieve this solution:
  - Segmentation
  - Paging

SEGMENTATION

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

local variables, global

Function

Method

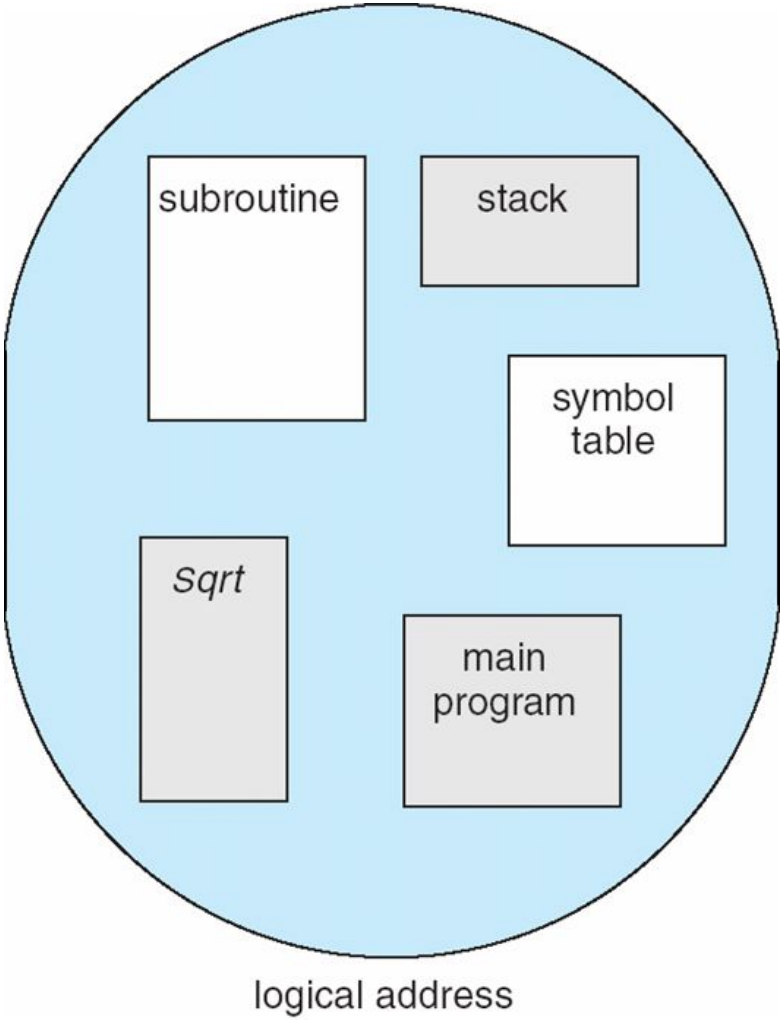
Object

common block

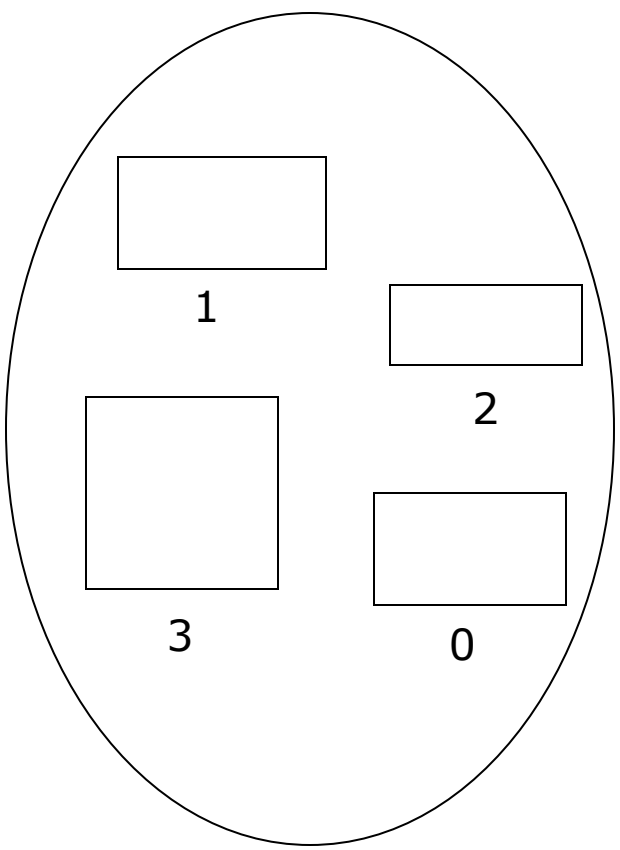
Stack

symbol table

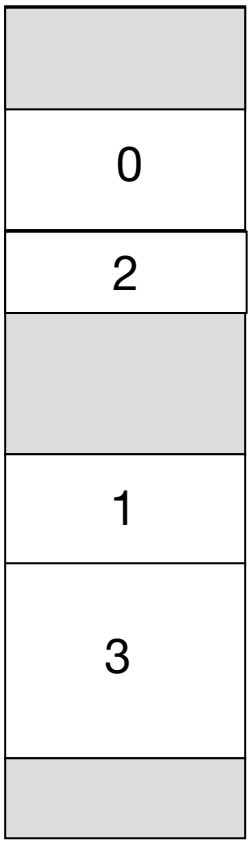
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation

- **Segmentation** is a memory-management scheme where logical address space is a collection of segments
- Each segment has a name and a length
- The programmer therefore specifies each address by two quantities:

  a segment name and an offset.

Segmentation Architecture

- Logical address consists of a two tuple:

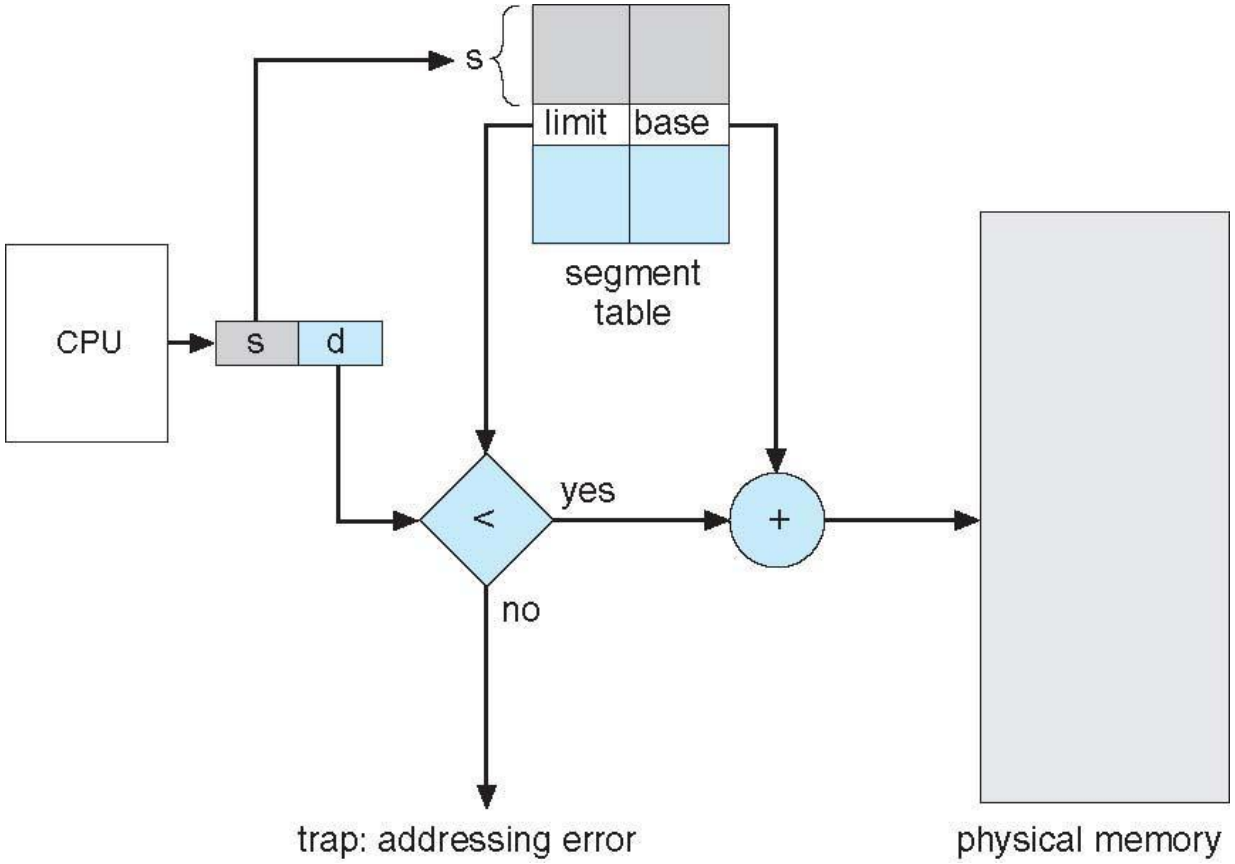
  <segment-number, offset>

- **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

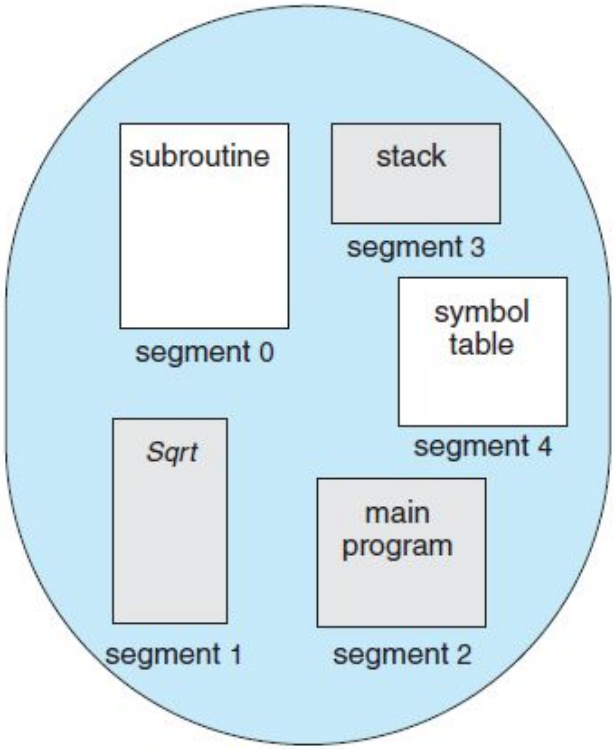
Segment Table

	Limit	Base
0	150	3200
1	80	5000
2	50	3350
3	300	5080

Segmentation Hardware

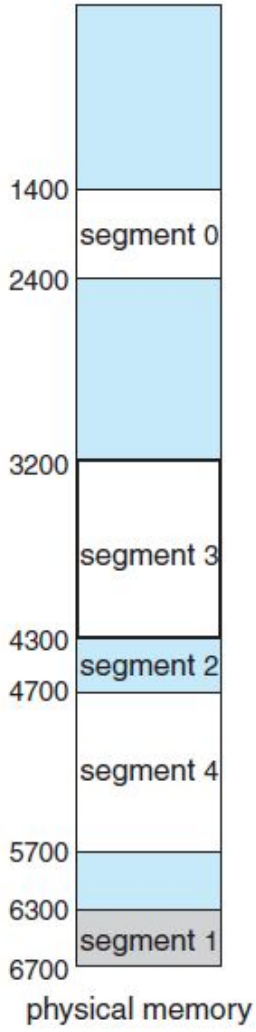


Example of Segmentation



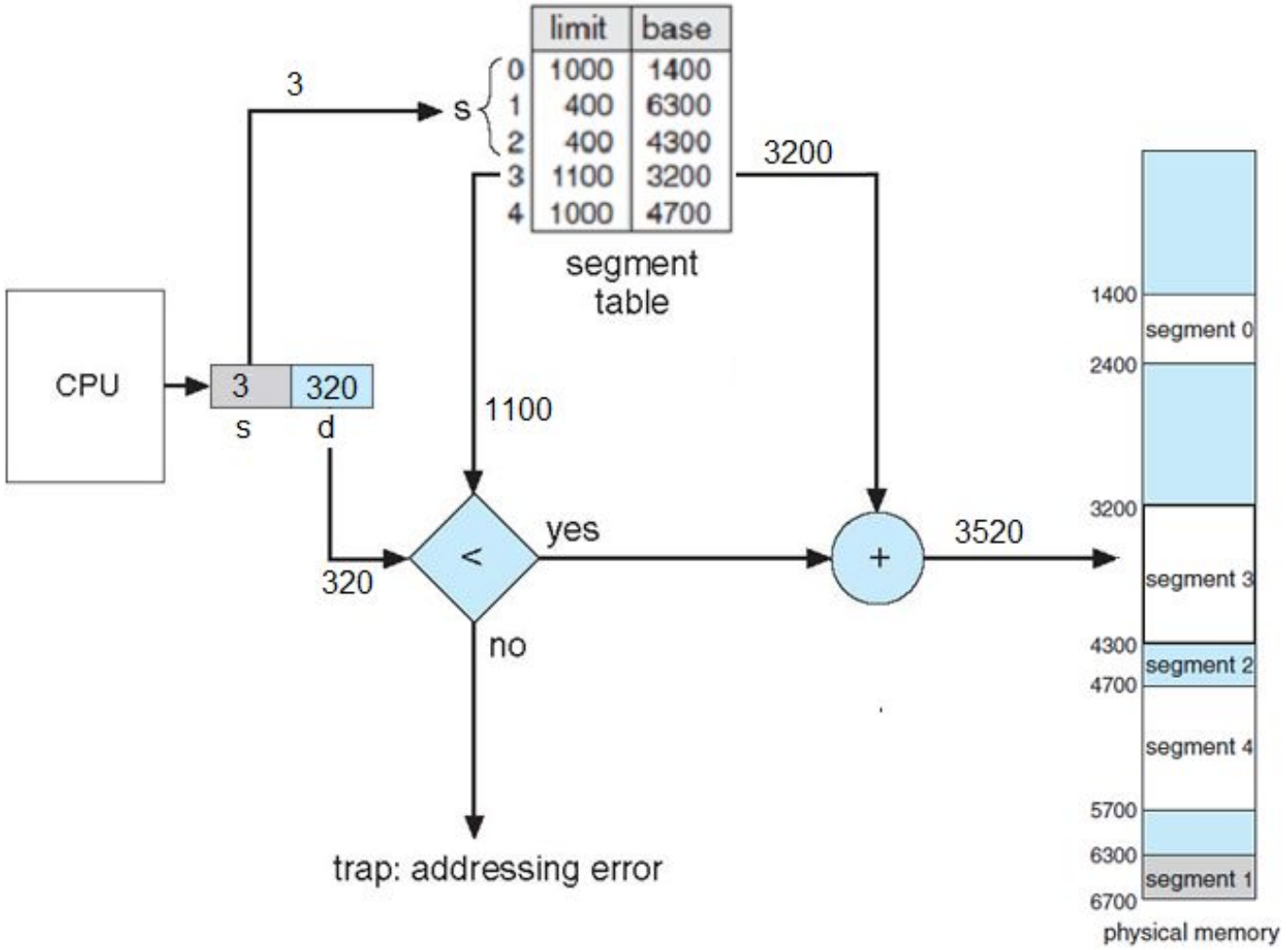
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Need to access segment **3**, offset **320**

The logical address is (**3, 320**)



Segmentation

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

Problem in Segmentation

- Consider the given segment table:

Segment	Base	Length
0	1215	50
1	8006	268
2	240	150
3	3456	500

Find the physical addresses for the following logical addresses?

a) 2, 80

b) 0, 120

c) 1,20

d) 2,150

PAGING

Paging

- Segmentation is a non-contiguous scheme but
  - it suffered external fragmentation
  - And need for compaction
- Paging is another non-contiguous scheme where process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks

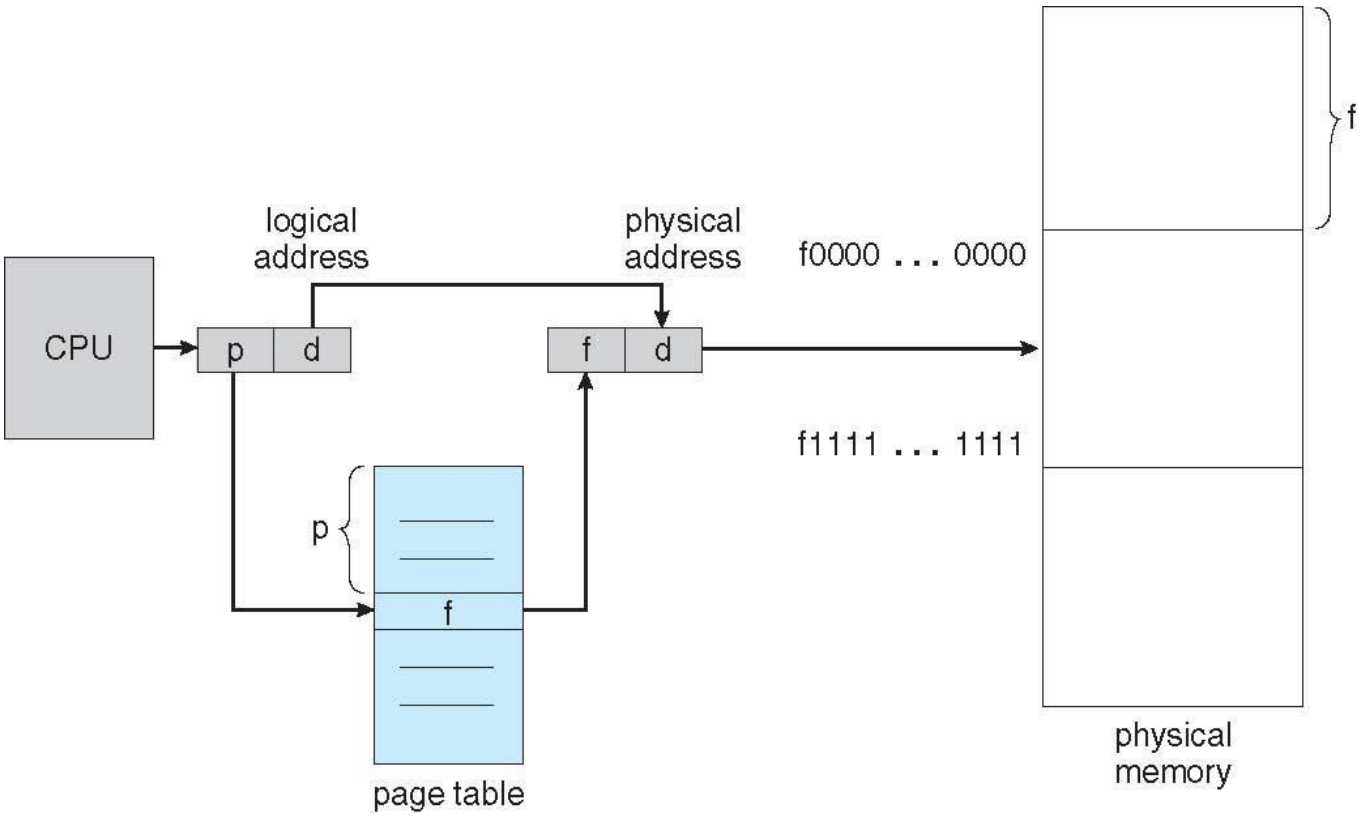
Paging

- 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

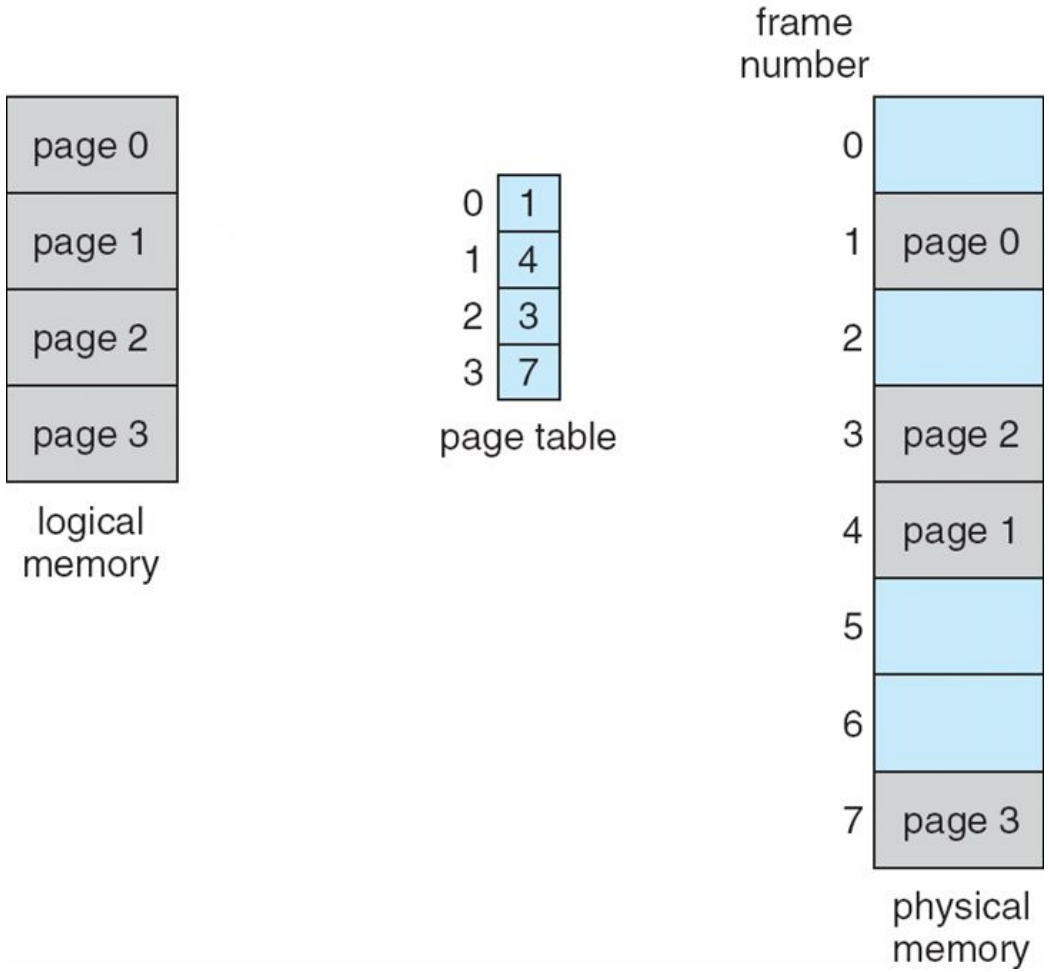
Address Translation Scheme

- 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

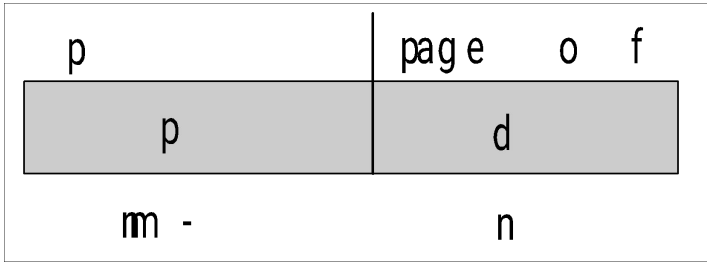
Paging Hardware



Paging Model of Logical and Physical Memory

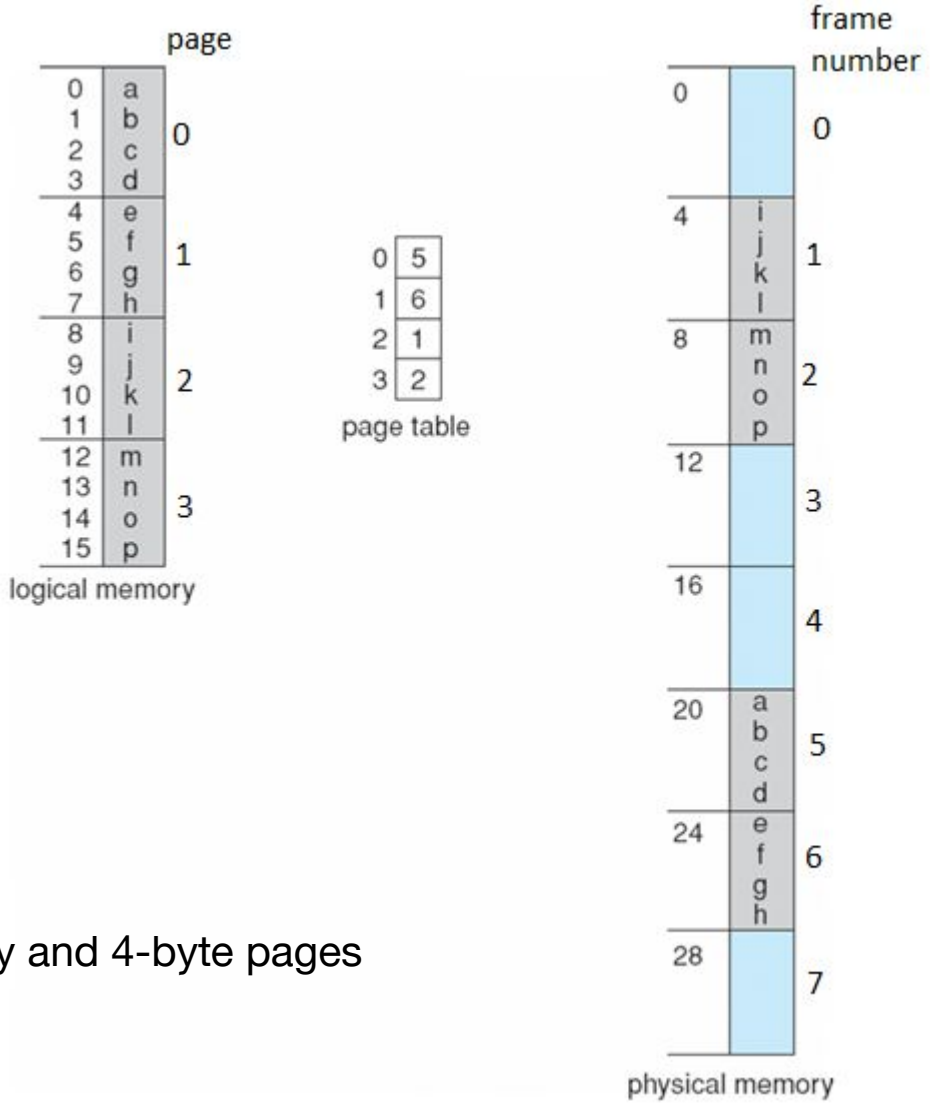


- If the size of the logical address space is 2^m, *and a page size is* 2^n *bytes, then the high-order m–n bits of a logical address designate the page* number, and the *n low-order bits designate the page offset.*
- *Thus, the logical* address is as follows:



where *p is an index into the page table and d is the displacement within the* page

Paging Example



n=2 and *m*=4

32-byte memory and 4-byte pages

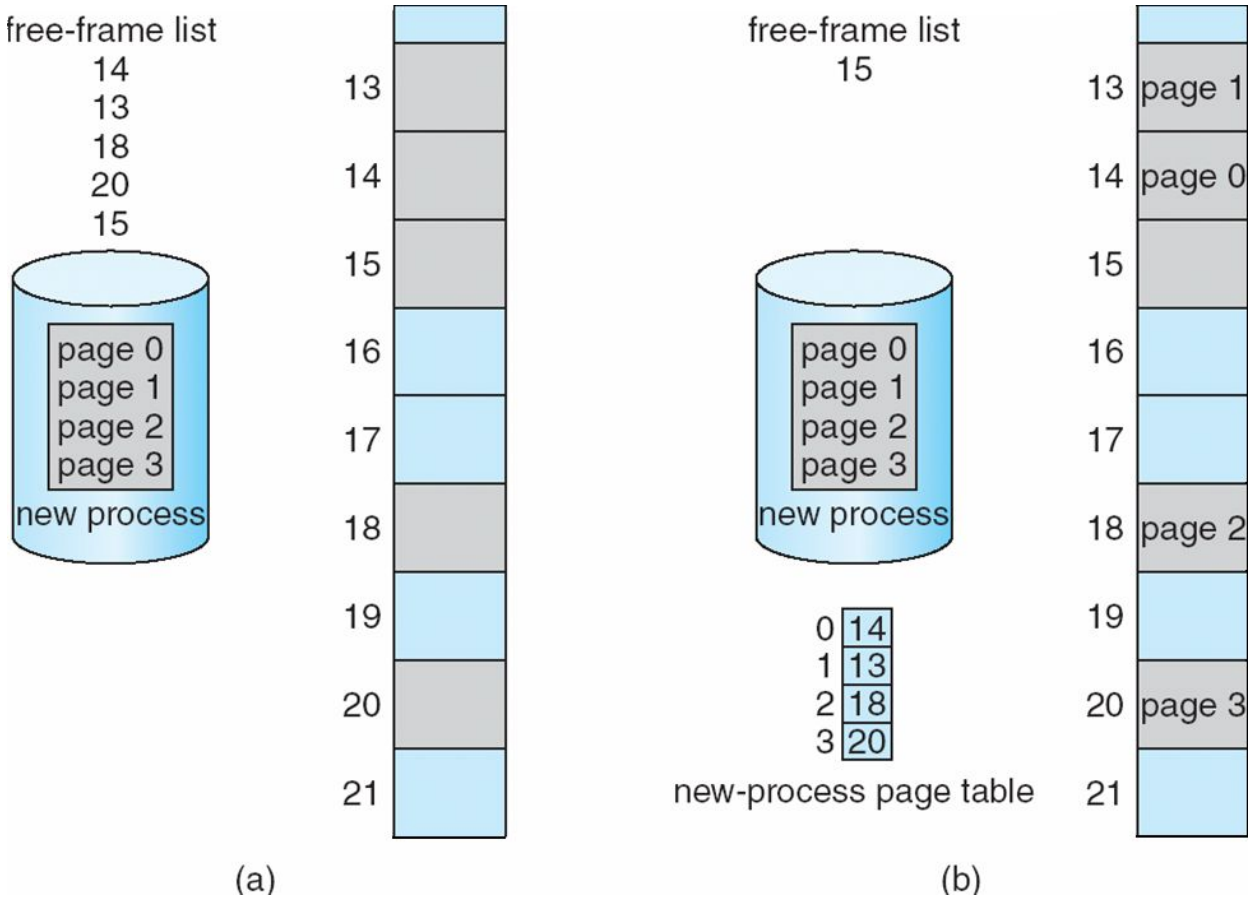
Paging (Cont.)

- 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

Paging (Cont.)

- Calculating internal fragmentation
  - Worst case fragmentation = 1 frame + 1 byte
  - On average fragmentation = 1/2 frame size per process
  - 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
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

Paging (Cont.)

- Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory
  - which frames are allocated,
  - which frames are available,
  - how many total frames there are, and so on.
- This information is kept in **frame table**

Implementation of Page Table

The page table can be implemented in several ways:

- As a set of dedicated **registers**
  - High speed
  - Page table size will be limited (to about 256 entries)
- Page table kept in main memory
  - Large page tables
  - Low speed

Implementation of Page Table

Page table 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**)

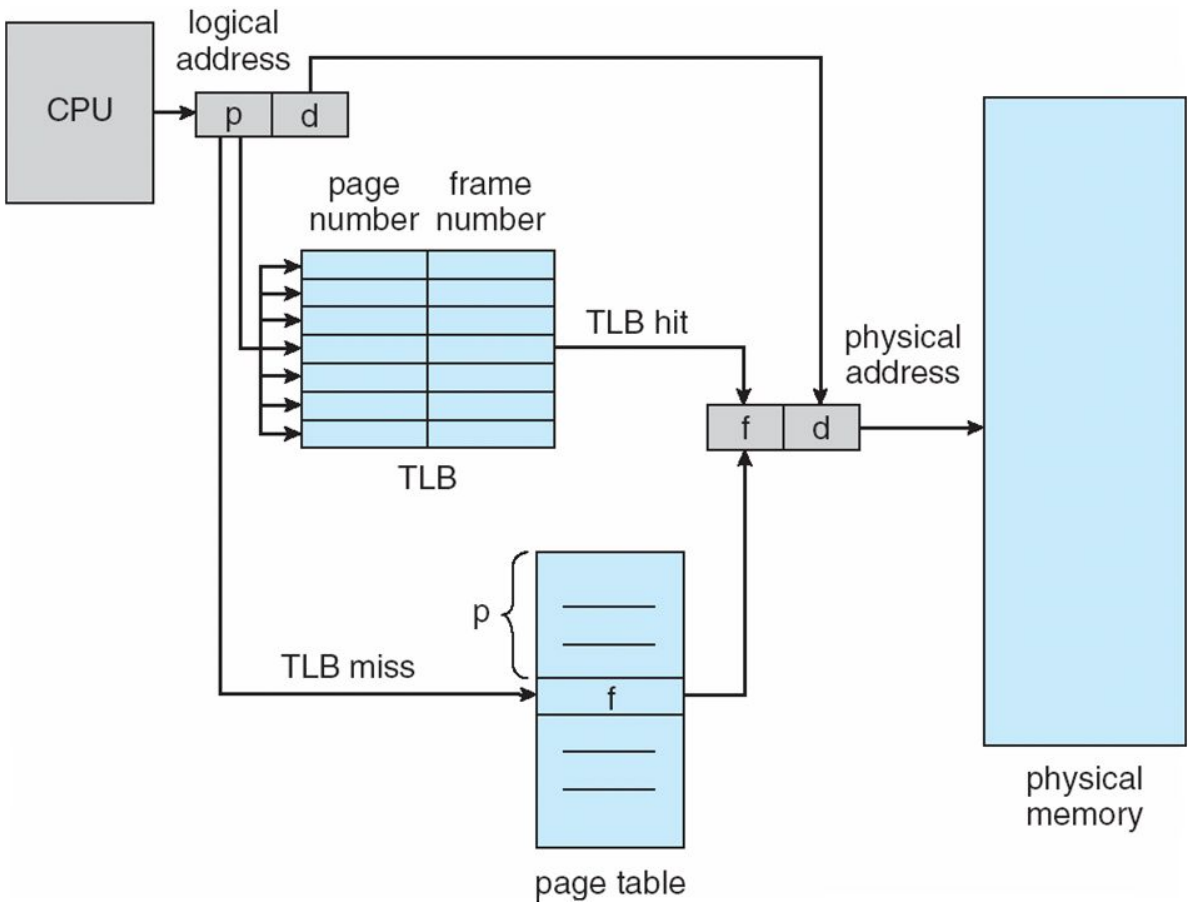
Associative Memory

- Associative memory – parallel search

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



Implementation of Page Table (Cont.)

- TLBs typically small (64 to 1,024 entries)
- TLB hit
- TLB miss
  - value is loaded into the TLB for faster access next time
  - Replacement policies must be considered (LRU or RR)
  - Some entries can be **wired down** for permanent fast access

Implementation of Page Table (Cont.)

- 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

Effective Access Time

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio.**
- Assume memory access takes 100ns
- **Effective Access Time** (**EAT**)
  - For 80% hit ratio

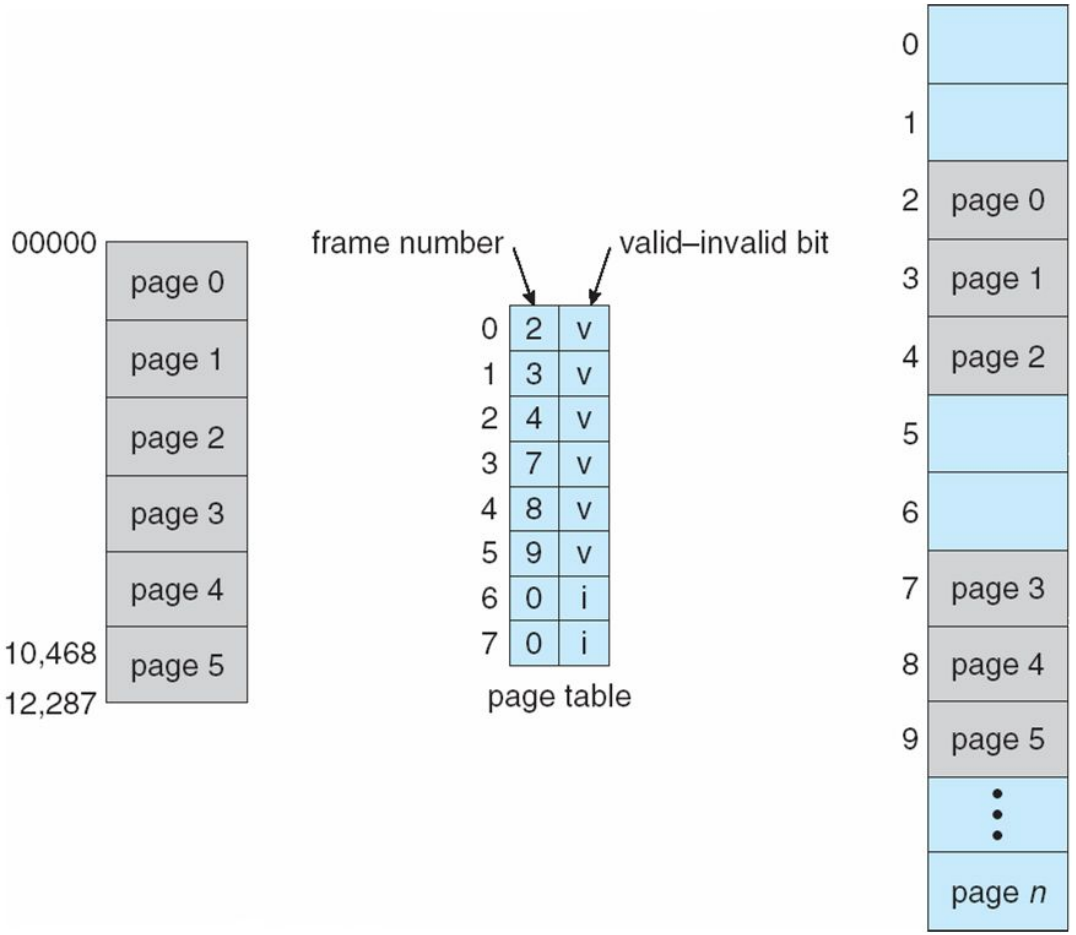
    EAT = 0.80 x 100 + 0.20 x 200 = 120ns

- Consider more realistic hit ratio = 99%,
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

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

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