OPERATING SYSTEMS

MEMORY MANAGEMENT

Background

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

- Register access in one CPU clock (or less)
- Cache sits between main memory and CPU registers
- Main memory can take many cycles, causing a stall

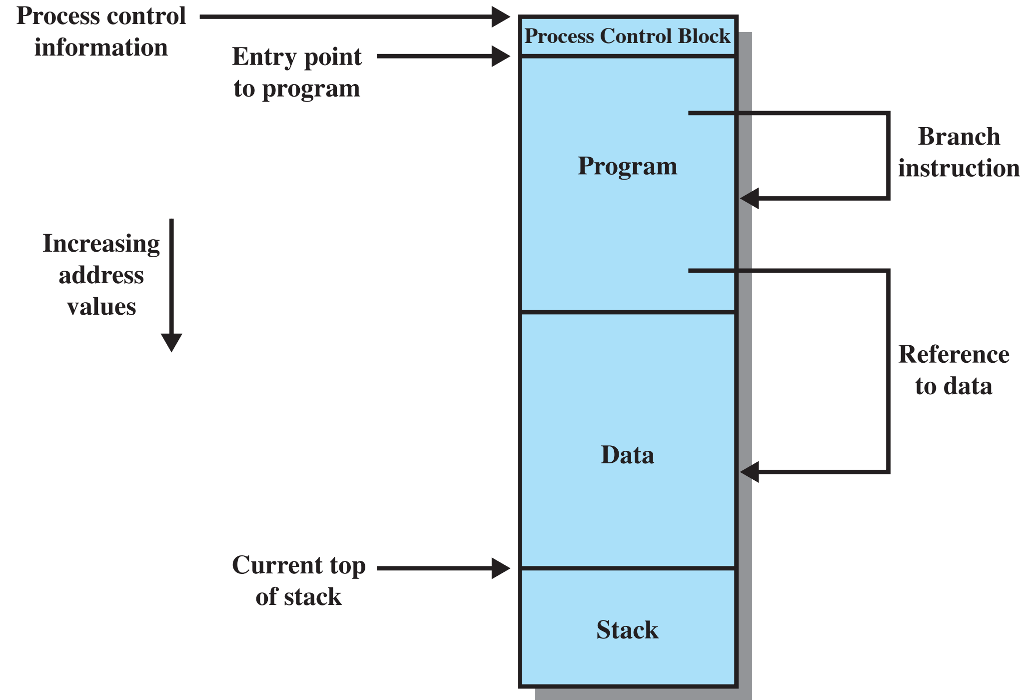
Memory Management Requirements

- Memory management is intended to satisfy the following requirements
  - **Relocation**
  - **Protection**
  - **Sharing**
  - **Logical organization**
  - **Physical organization**

Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- The OS may need to relocate the process to a different area of memory when it is swapped back in
  - Specifying that a process must be placed in the same memory region would be limiting

Addressing Requirements for a Process



Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection

Sharing

- Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection
- Mechanisms used to support relocation support sharing capabilities

Logical Organization

- Memory is organized as linear

Programs are written in modules

- Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Sharing on a module level corresponds to the user's way of viewing the problem

- The main memory can be logically organized in segments corresponding to the program modules

Physical Organization

Cannot leave the programmer with the responsibility to manage memory

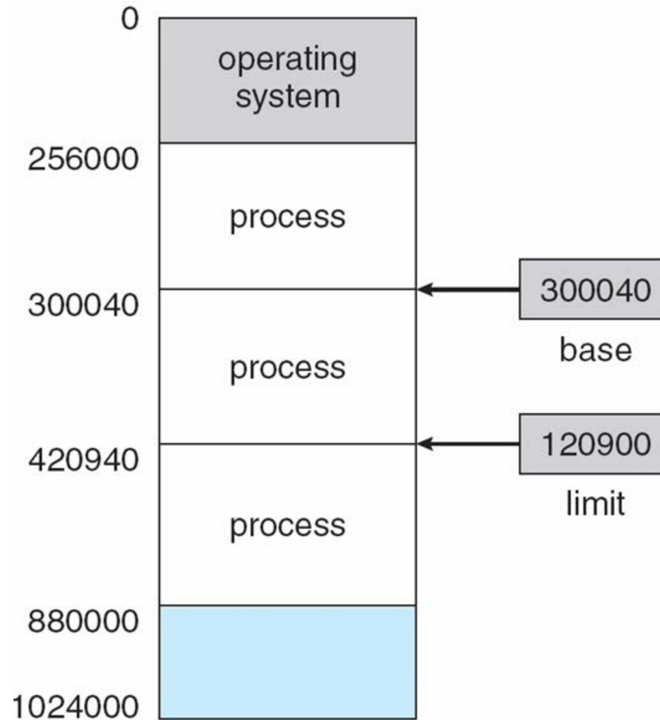
Memory available for a program plus its data may be insufficient

Programmer does not know how much space will be available

Overlaying allows various modules to be assigned the same region of memory but is time consuming to program

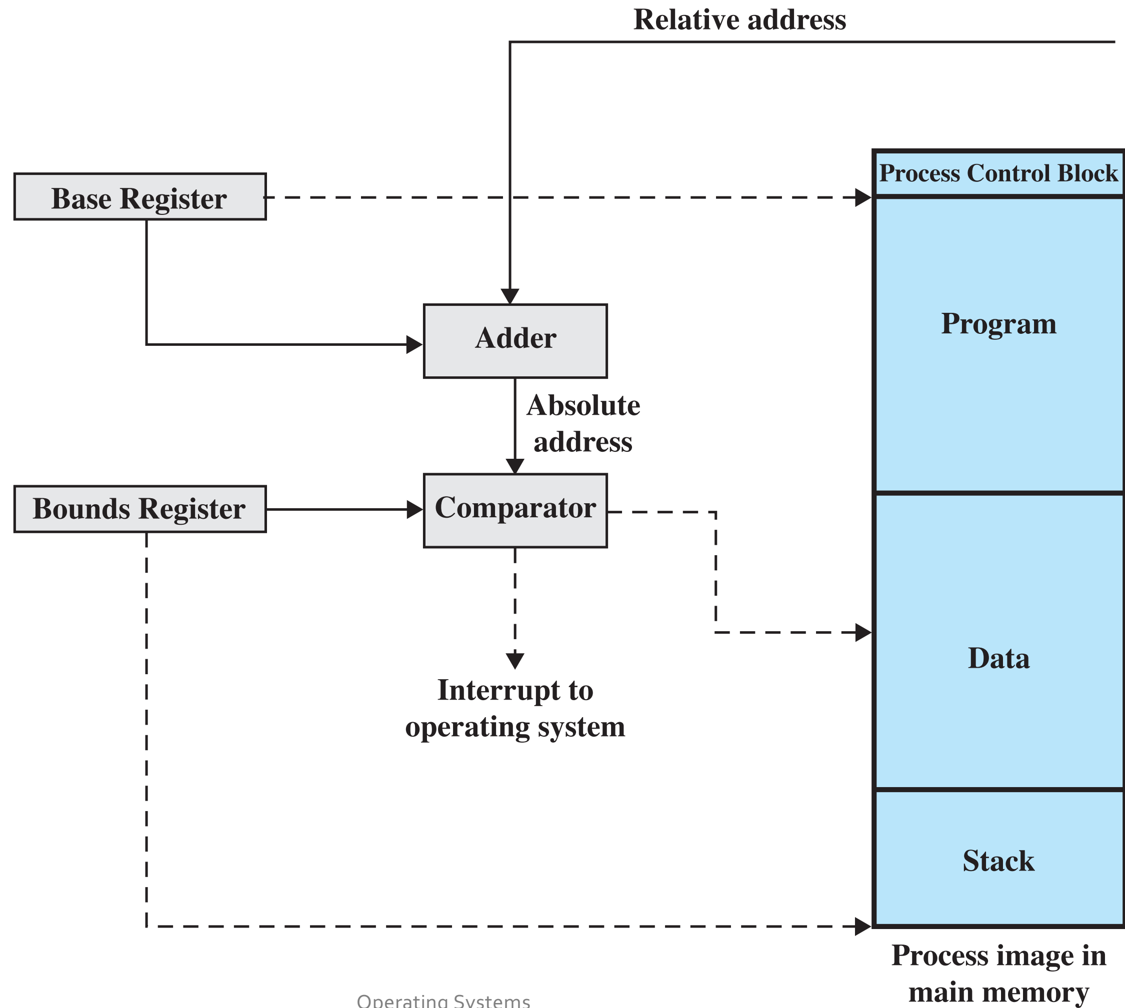
Memory access

Base and Limit Registers

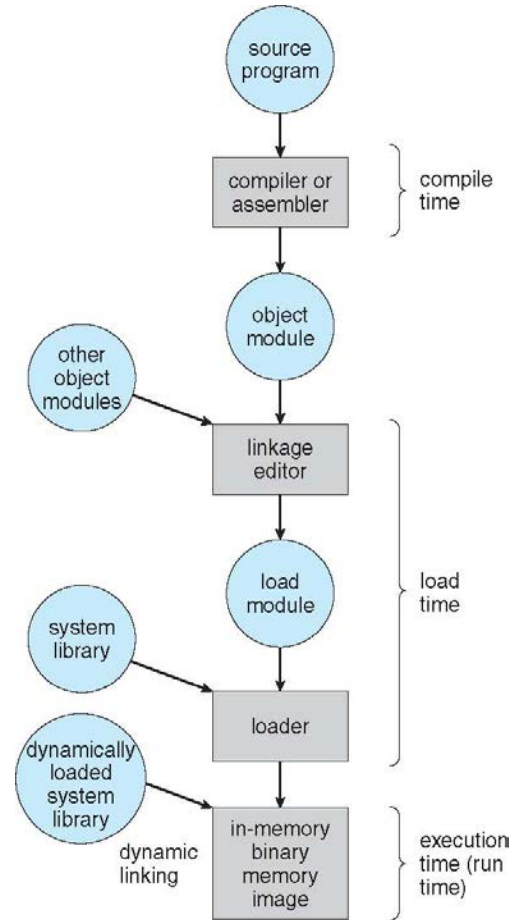


- A pair of base and limit registers define the logical address space
- The CPU must check that every memory access generated in user mode is between base and limit

Hardware Address Protection



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 is known a priori, absolute code can be generated. Recompilation needed 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
    - Need hardware support for address maps

Logical vs. Physical Address Space

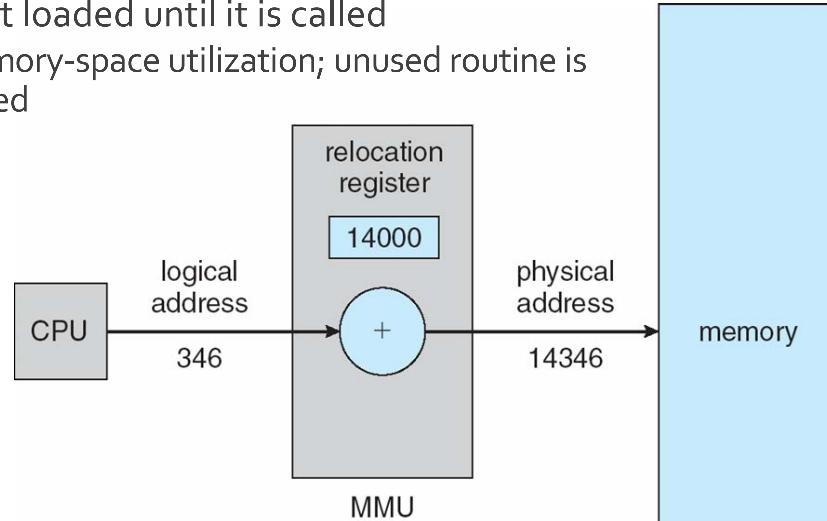
- The OS manages a logical address space that is bound to a separate physical address space
  - **Logical address**
    generated by the CPU, also referred to as virtual address
  - **Physical address**
    address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- In the simplest scheme 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

Dynamic relocation using a relocation register

- All modules kept on disk in relocatable load format
- A module is not loaded until it is called
  - Better memory-space utilization; unused routine is never loaded



- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required

Swapping

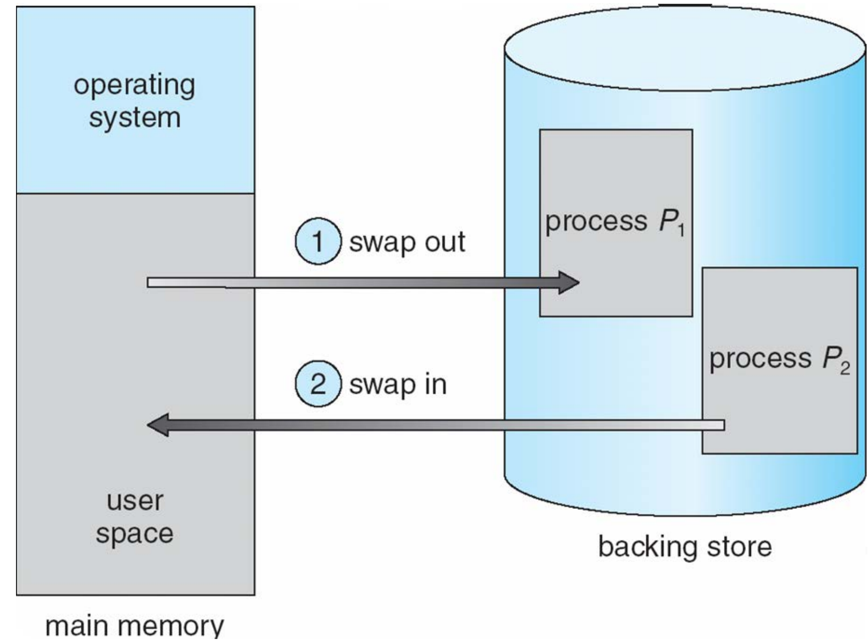
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store**
  fast long-term memory storage (e.g., disks), large enough to accommodate copies of memory images for all users
- **Roll out, roll in**
  swapping variant used for priority-based scheduling algorithms
  - lower-priority processes are swapped out so higher-priority processes can be loaded and executed

Schematic View of Swapping

Major part of swap time is transfer time

The OS maintains a ready queue of ready-to-run processes which have memory images on disk



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
- Example: 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 4000ms (4 seconds)
- Standard swapping not used in modern operating systems, but modified version common
  - e.g., swap only when free memory extremely low and one process is neither performing any action, nor awaiting for any event

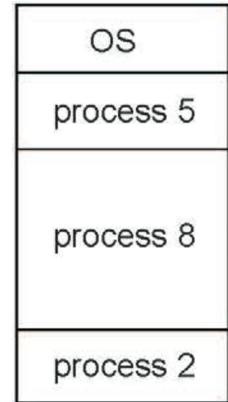
Swapping on Mobile Systems

- Not supported because there is no backing store
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU
- When memory is running out of free space
  - 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

Memory Partitioning

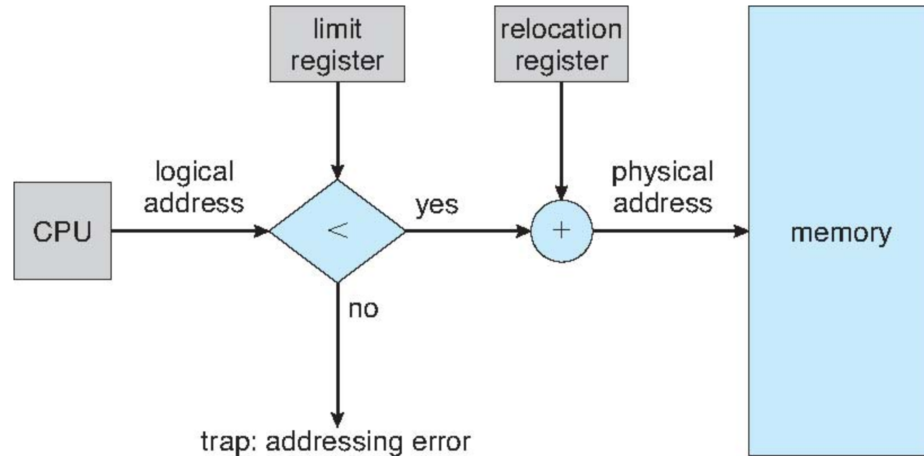
Contiguous Allocation

- Main memory must support both OS and user processes
- Contiguous allocation is one early method
- Main memory usually divided 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

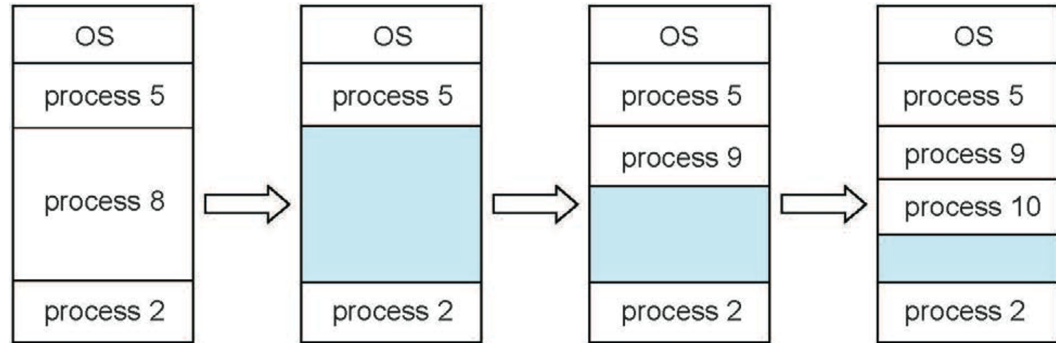


Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions in the contiguous allocation scheme
- Variable-partition sizes for efficiency (sized to a given process' needs)
- **Hole**
  block of available memory. Holes of various size are scattered throughout memory

Multiple-partition allocation

- 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)

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
- **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.)

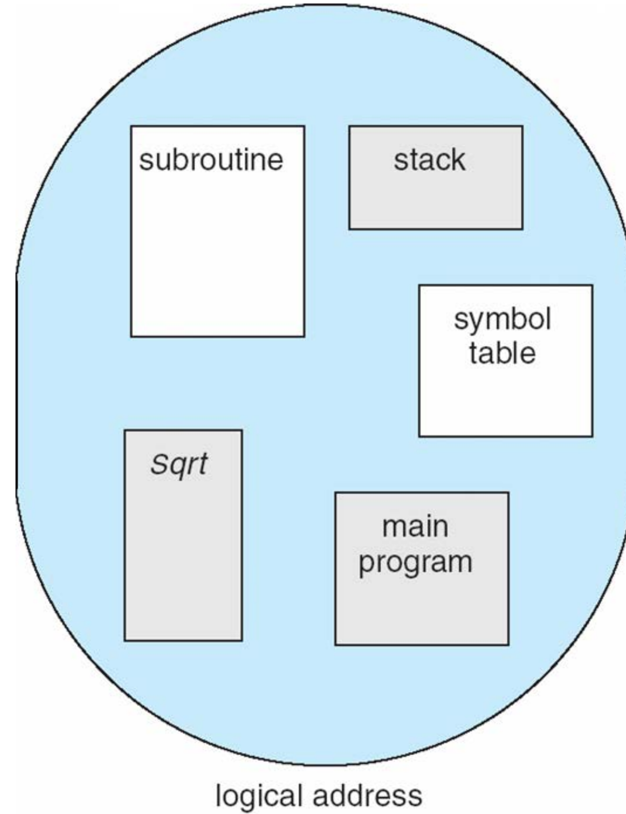
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Segmentation

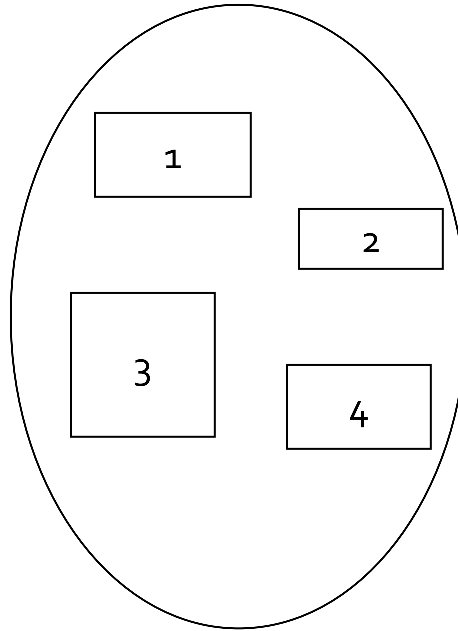
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments, where a segment is a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays

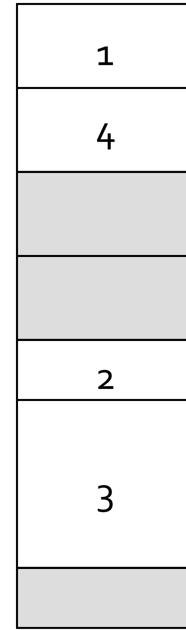
User's View of a Program



Logical View of Segmentation



user space

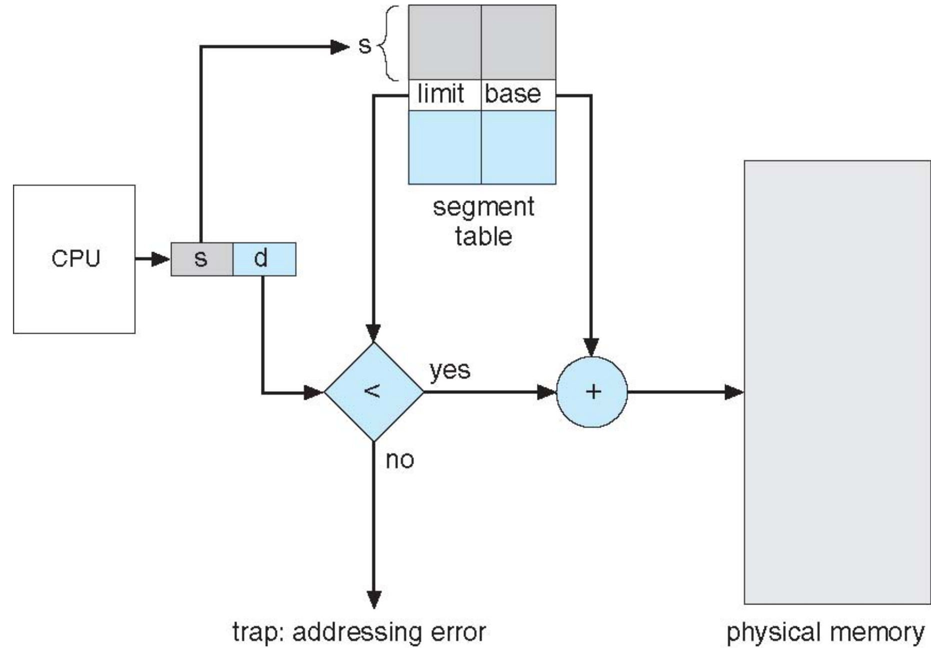


physical memory space

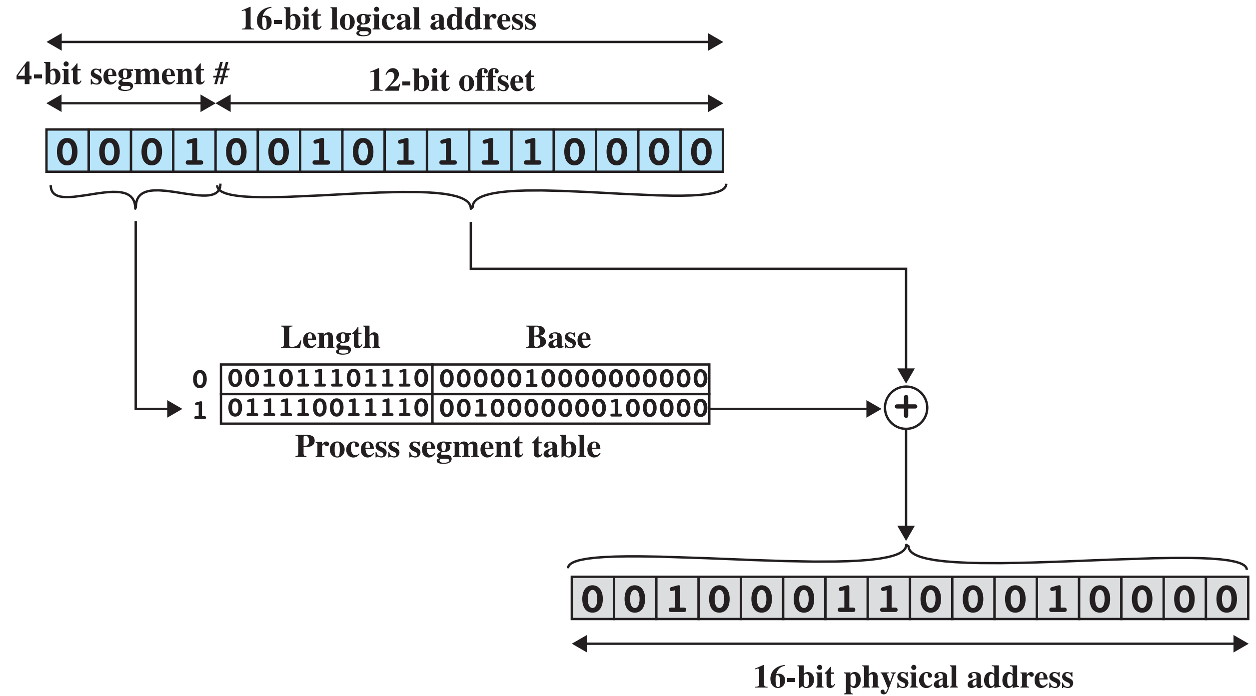
Segmentation Architecture

- Logical address consists of a tuple
  $\langle \text{segment-number, offset} \rangle$
- **Segment table**
  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 base register (STBR)
  points to the segment table's location in memory
- Segment-table length register (STLR)
  indicates number of segments used by a program

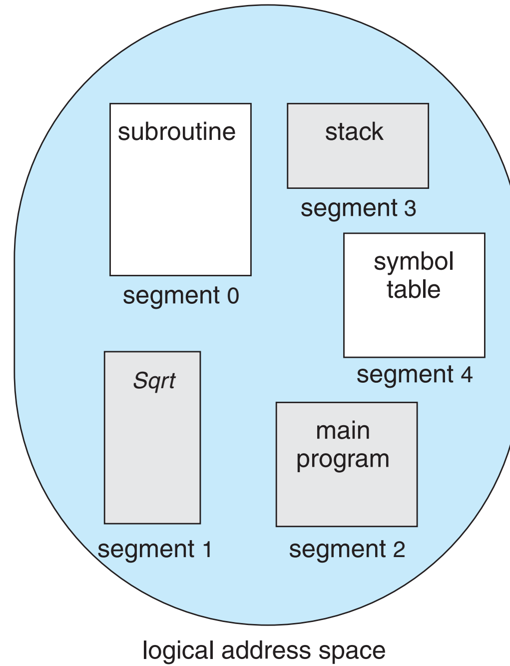
Segmentation Hardware



Example of Logical-to-Physical Address Translation

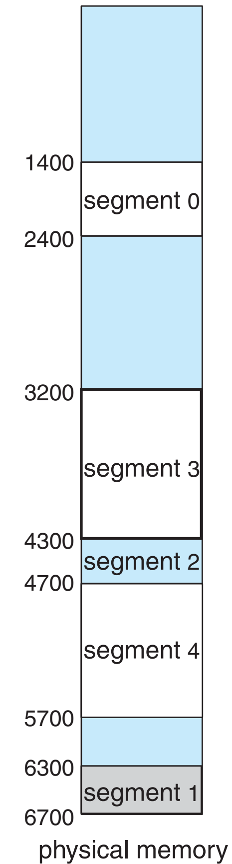


Segmentation Example



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

segment table



Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments
- Code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

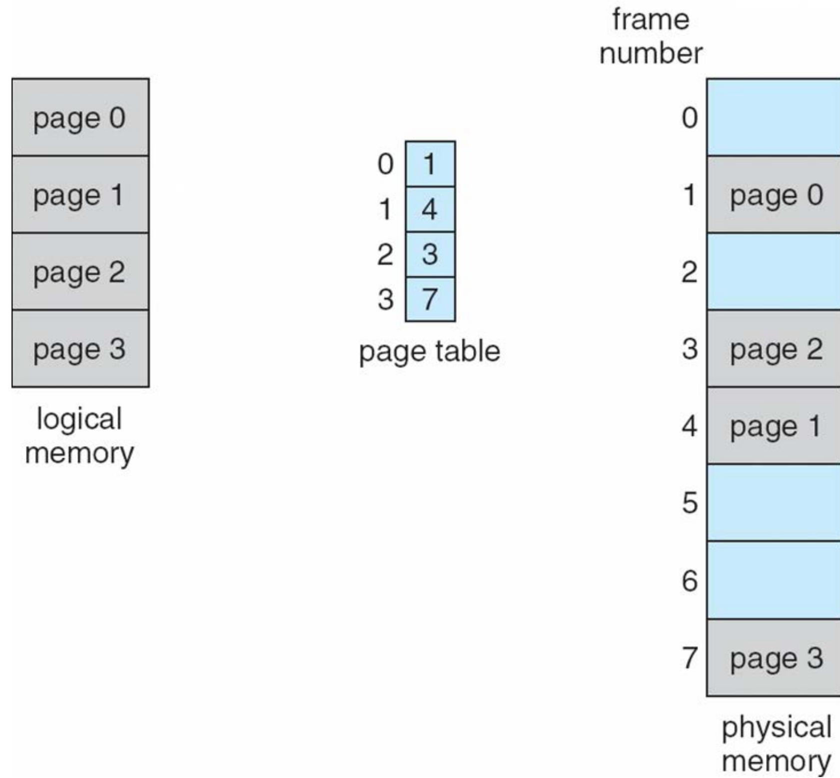
Paging

Paging

- Partition memory into equal fixed-size chunks that are relatively small
- Process is also divided into small fixed-size chunks of the same size

- To run a program of size *N* pages, need to find N free frames and load program

Paging Model of Logical and Physical Memory

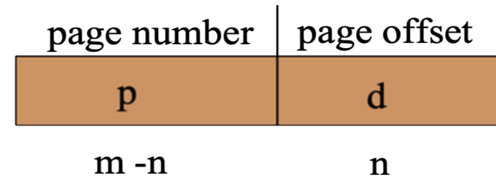


Paging

- 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
- Set up a page table to translate logical to physical addresses
- Backing store likewise split into pages
- Still have internal fragmentation

Address Translation Scheme

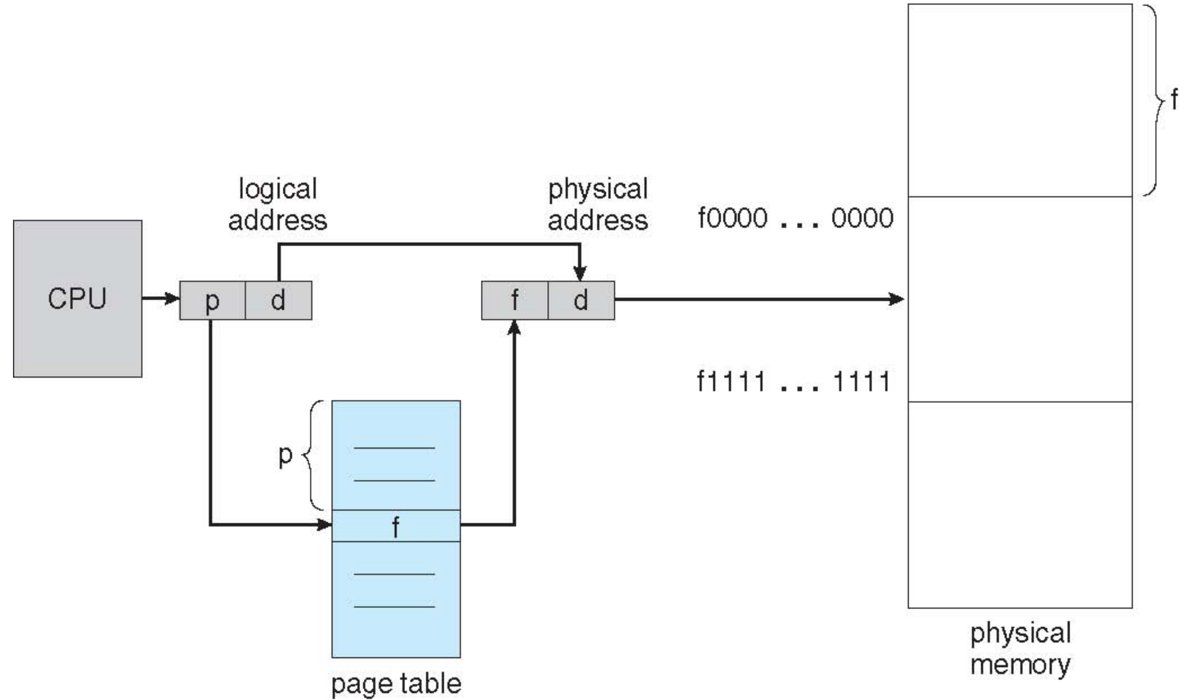
- Address generated by CPU is divided into
  - **Page number (*p*)**
    used as an index into a page table which contains base address of each page in physical memory
  - **Page offset (*d*)**
    combined with base address to define the physical memory address that is sent to the memory unit



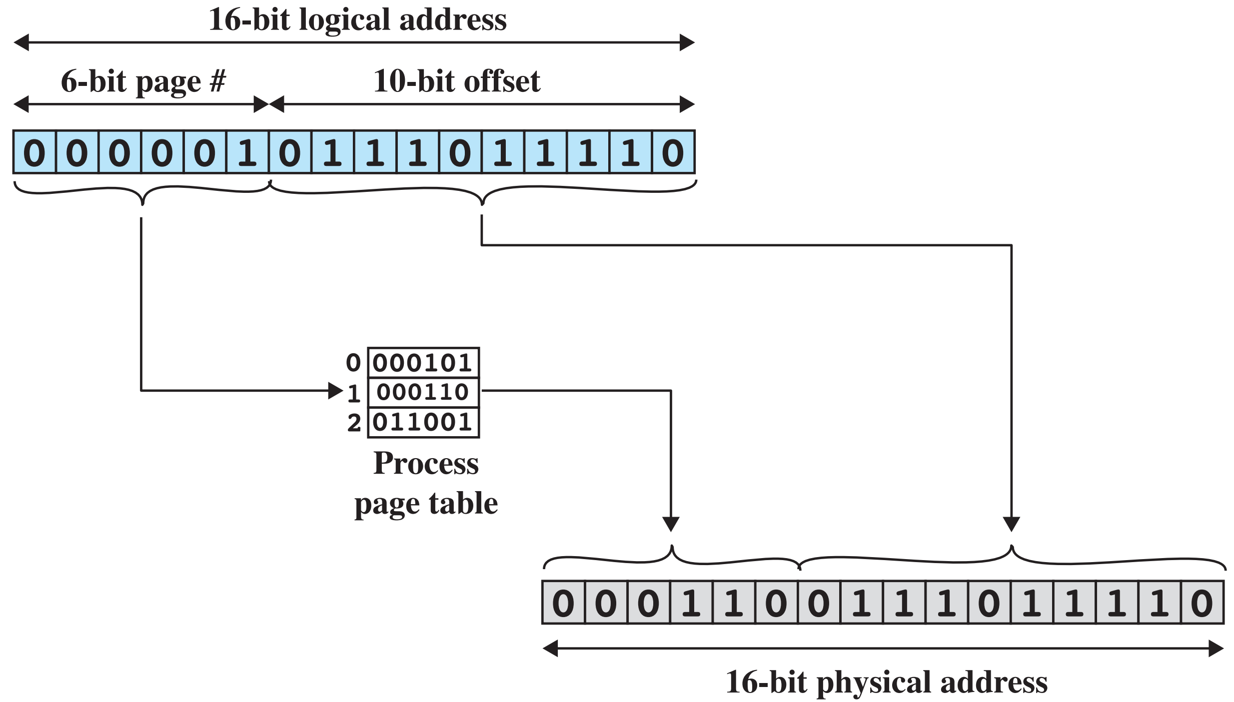
For given logical address space 2^m and page size 2^n

Page size is power of 2, between 512 bytes and 16 Mbytes

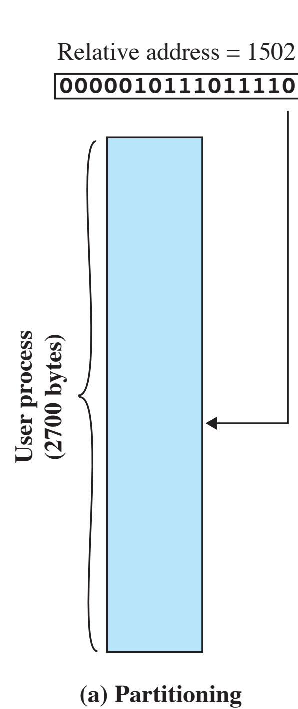
Paging Hardware



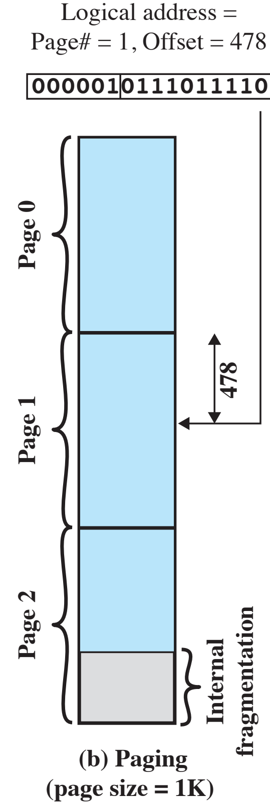
Example of Logical-to-Physical Address Translation



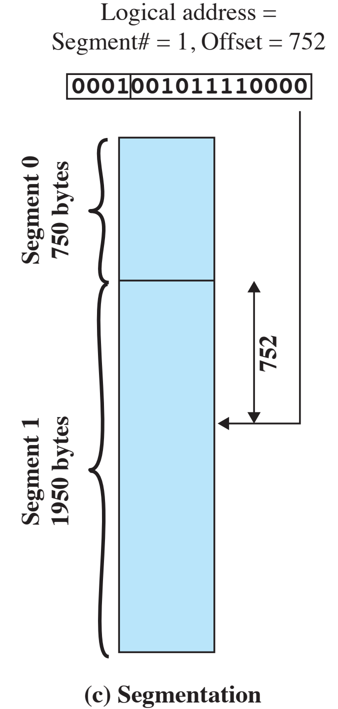
Logical Addresses Comparison



(a) Partitioning



(b) Paging
(page size = 1K)



(c) Segmentation

Paging Example

logical address	logical memory
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

logical memory

page	frame
0	5
1	6
2	1
3	2

page table

physical address	physical memory
0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

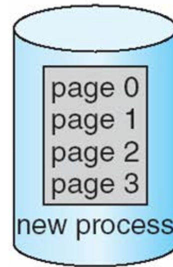
physical memory

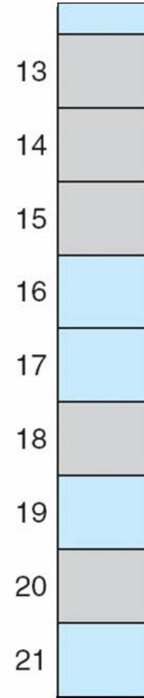
n=2 and m=4

32-byte memory and 4-byte pages

Free Frames

free-frame list
14
13
18
20
15

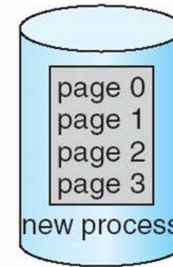




(a)

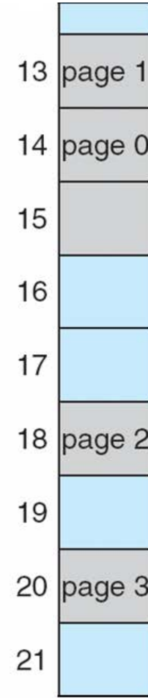
Before allocation

free-frame list
15



0	14
1	13
2	18
3	20

new-process page table



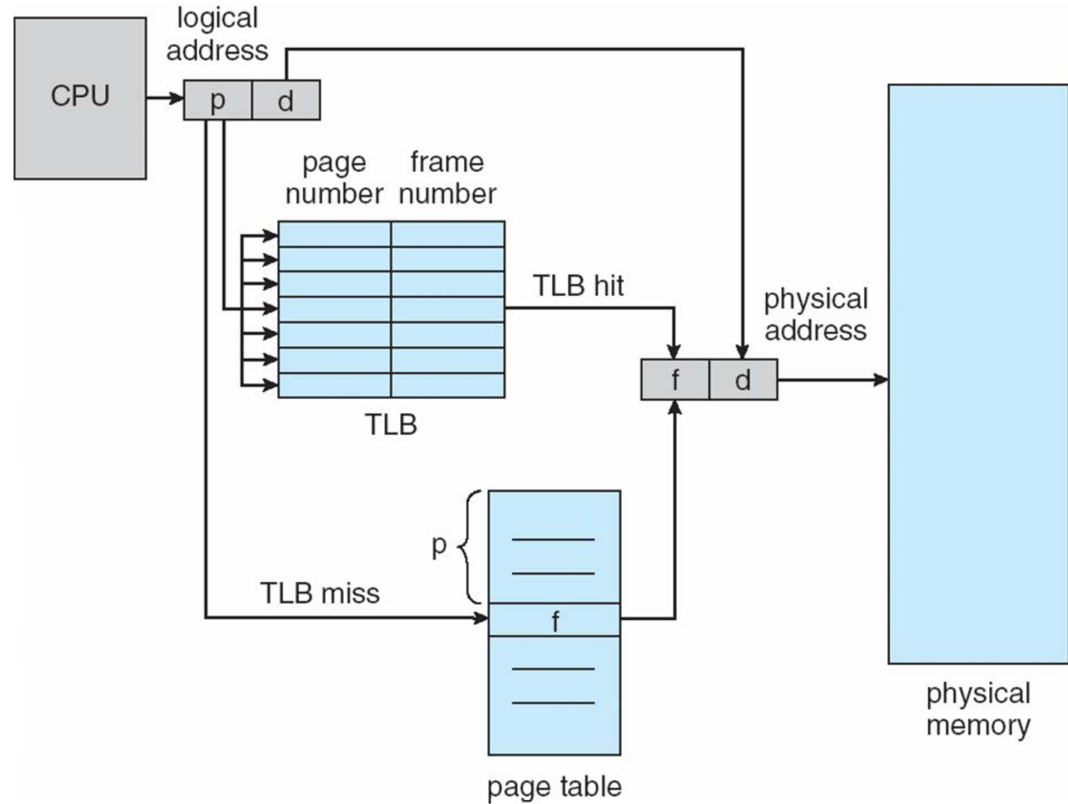
(b)

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
- To speed-up the access to the page table, a special fast-lookup hardware cache is used for the page table associative memory or translation look-aside buffers (TLBs)

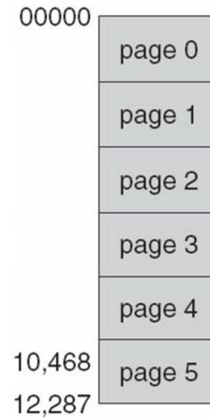
Paging Hardware with TLB

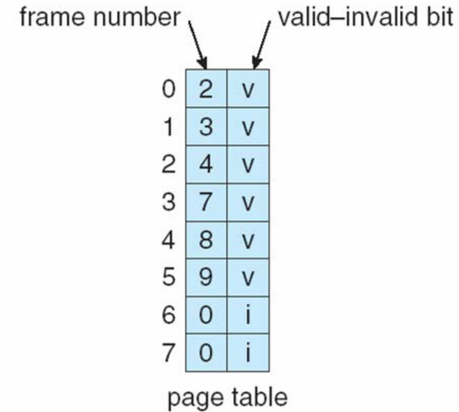


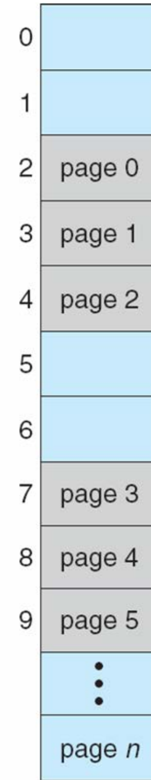
Memory Protection

- 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**
    the associated page is in the process' logical address space, and is thus a legal page
  - **invalid**
    the page is not in the process' logical address space
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit in a Page Table



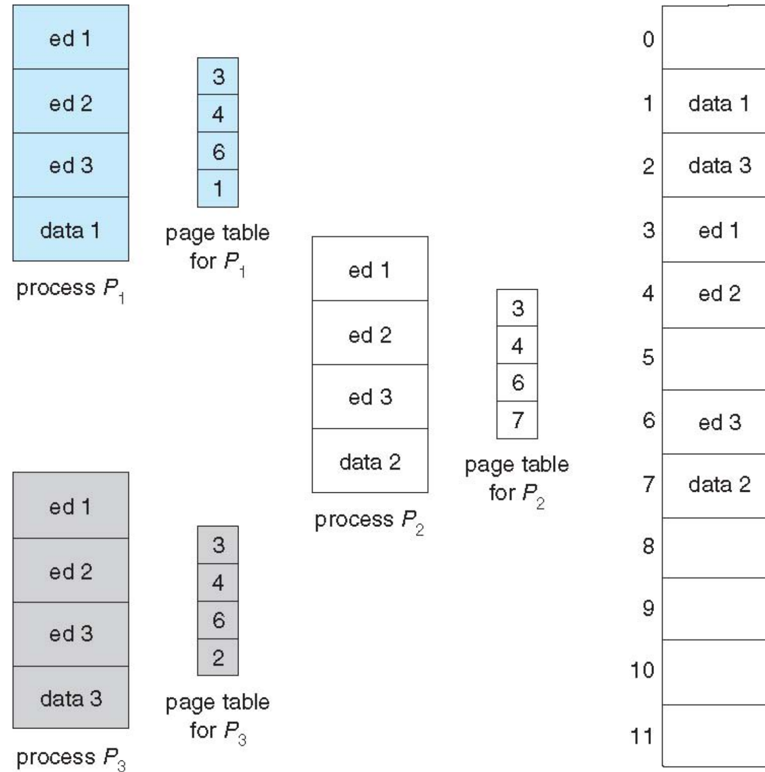




Shared Pages

- **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

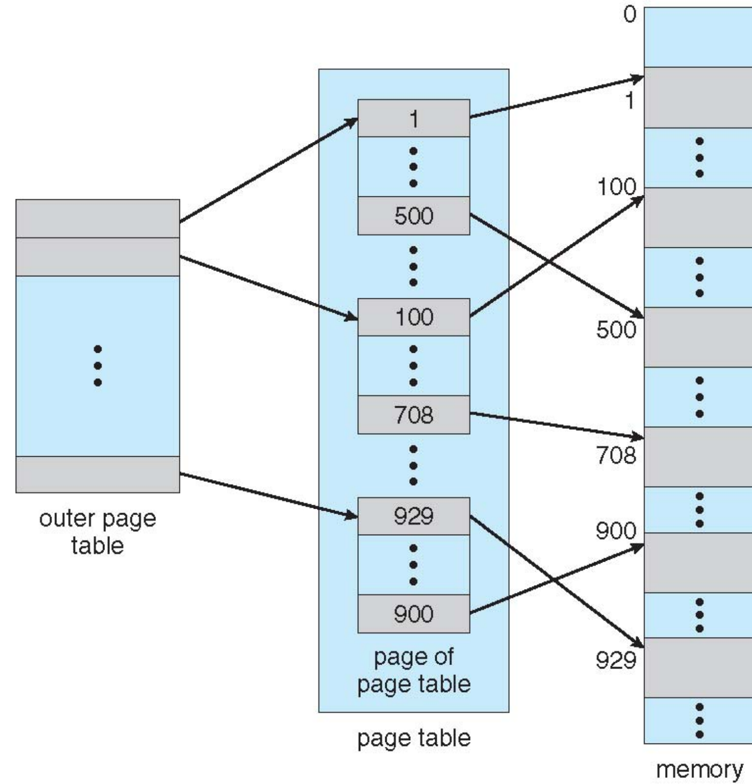
Shared Pages Example



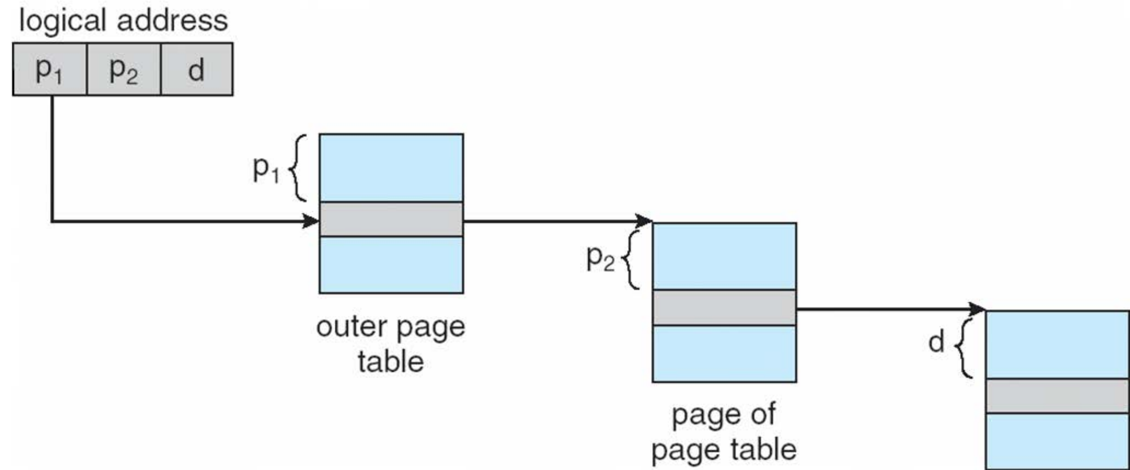
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers and a page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

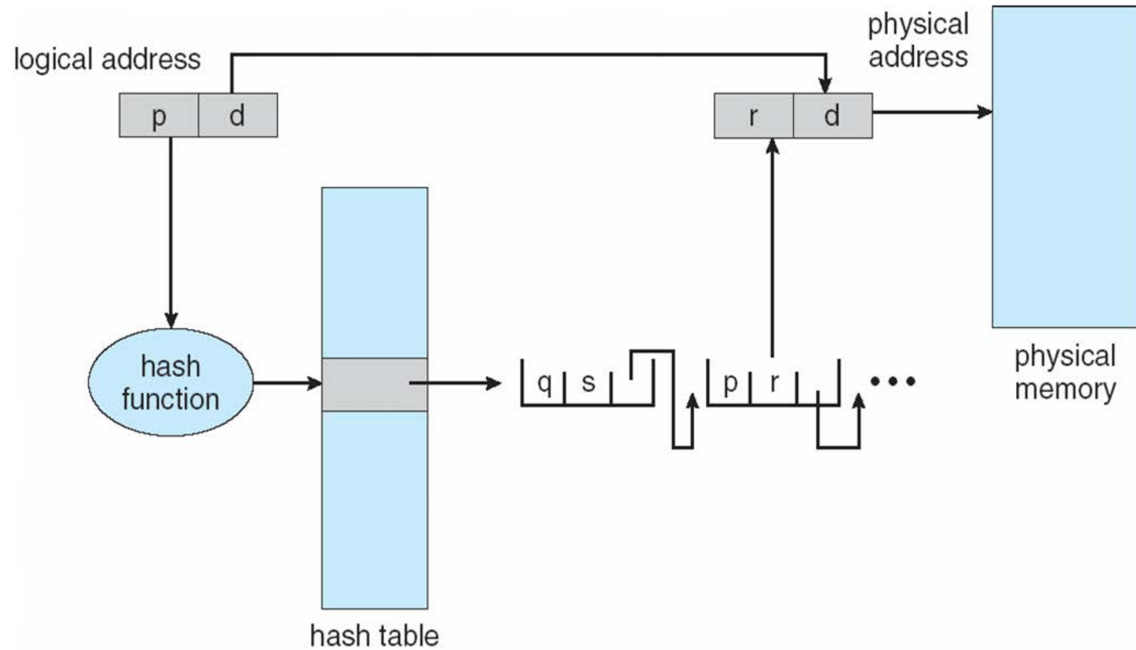
Two-Level Hierarchical Page-Table Scheme



Hierarchical Page Table Address-Translation Scheme

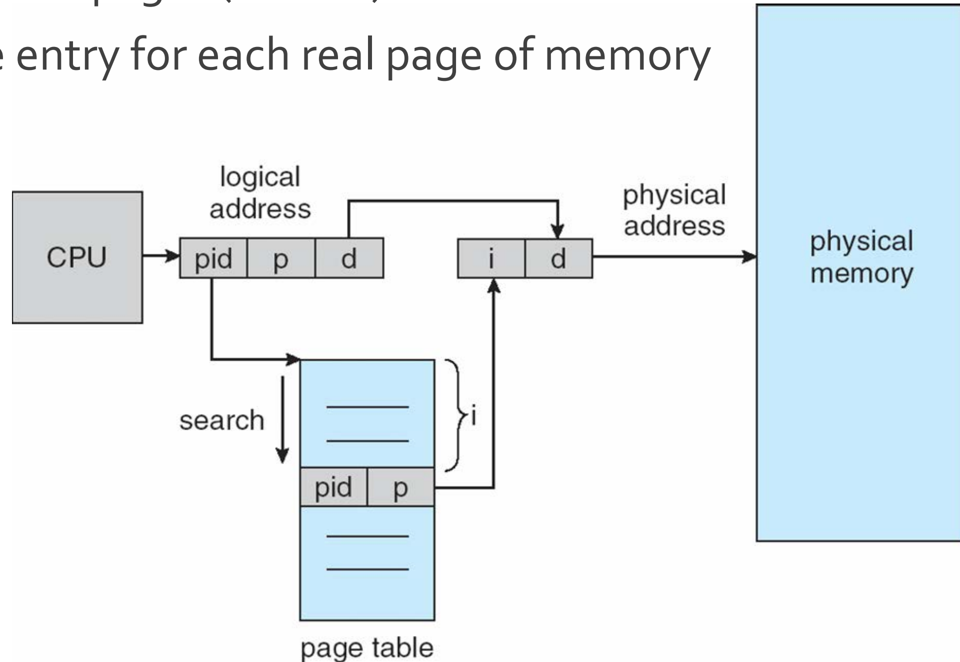


Hashed Page Table



Inverted Page Table Architecture

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages (frames)
- One entry for each real page of memory

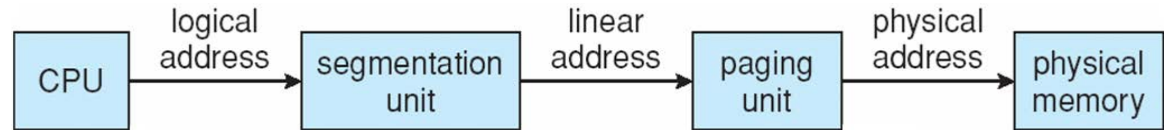


Examples
Intel IA-32 and ARM

The Intel IA-32 Architecture

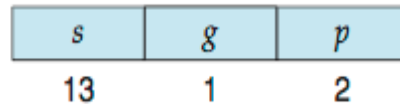
- Supports both segmentation and segmentation with paging
- Each segment can be 4 GB
- Up to 16 K segments per process
- Divided into two partitions
  - First partition of up to 8K segments private to process (kept in local descriptor table (LDT))
  - Second partition of up to 8K segments shared among all processes (kept in global descriptor table (GDT))

Logical to Physical Address Translation in IA-32



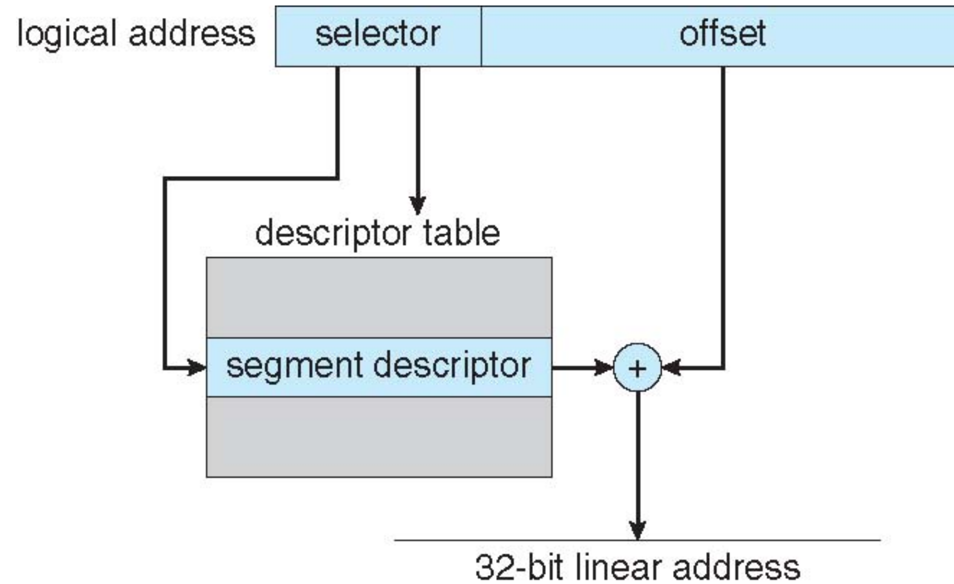
The logical address is a (selector, offset) pair.

The selector is a 16-bit number



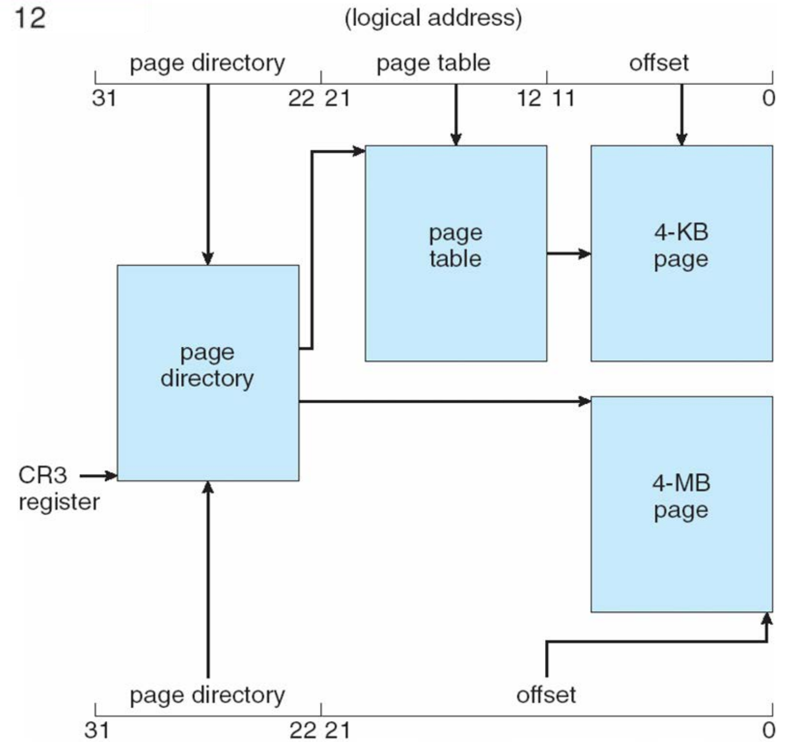
The offset is a 32-bit number

Intel IA-32 Segmentation



Intel IA-32 Paging Architecture

page number		page offset
p_1	p_2	d
10	10	12



Example: ARM Architecture

- Dominant mobile platform chip (e.g., Apple iOS and Google Android)
  - Energy efficient 32-bit CPU
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages (termed *sections*)
- Two levels of TLBs
  - Outer level has two micro TLBs (data, instruction)
  - Inner is single main TLB