Chapter 8:

Main Memory Management

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Objectives/Outline

Objectives

- Describe various ways of organizing memory hardware (which are pertinent to various memory managing techniques)
- Discuss various memory management techniques (including paging and segmentation)
- Provide a detailed description of Intel Pentium which supports both pure segmentation and segmentation with paging

Outline

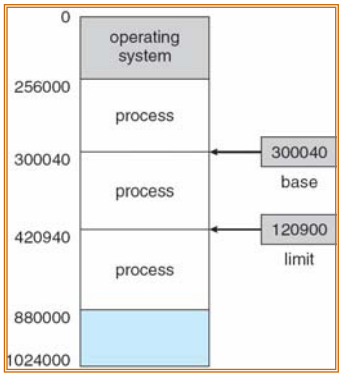
- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
- Example: Intel Pentium

Background

- A program (together with the data it needs) must be brought (from the disk) into main memory (at least partially) before execution
- A typical instruction execution cycle:
  - The CPU first fetches instructions from memory according to the value of the program counter
  - Decode the instruction, may cause operands to be fetched from memory
  - Execute the instruction, may need to store results in memory
- To improve CPU utilization and response time, the computer must keep several processes in memory
- Memory management – is responsible for sharing memory among processes to ensure correct operation
- There are many memory management schemes ranging from a primitive bare machine approach to paging and segmentation
  - The effectiveness and selection of a memory management scheme for a system depends on several factors especially hardware support

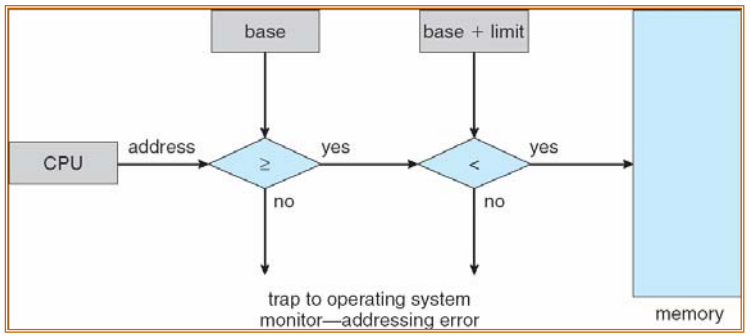
Basic Hardware

- Main memory consists of a large array of words or bytes each with its own address
- Main memory and registers are only storage CPU can access directly
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles
  - A cache is used to improve the access time
- Memory system only sees a sequence of memory addresses without knowing how they are generated nor whether they are for instructions or data
- A pair of **base** and **limit** registers define the address space



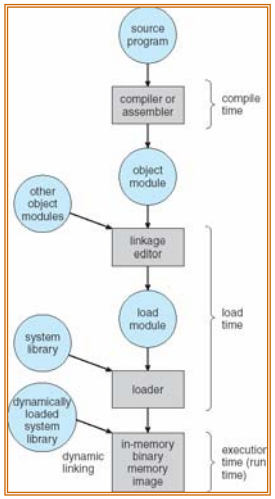
HW address protection with base and limit registers

- Protection of memory space is achieved by CPU hardware



Address Binding

- A user program goes through several steps
  - Compile
  - Link
  - Load
  - Execute
- Addresses are represented in different ways during these steps
  - Source code – symbolic addresses
  - Object module – relocatable addresses
  - Binary memory image – absolute addresses
- Address binding is mapping from one address space to another



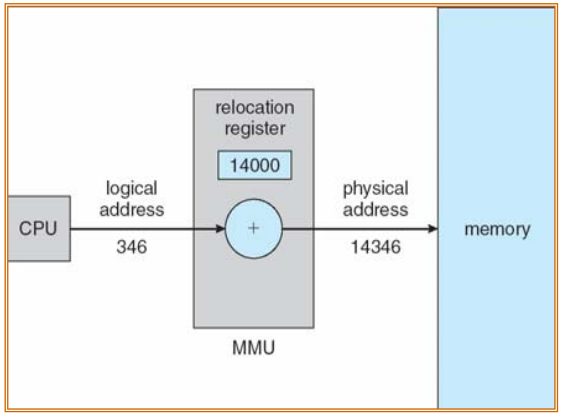
Address Binding (cont.)

- Address binding of instructions and data to memory addresses can happen at any of 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**: If memory location is not known at compile time, compiler must generate relocatable code
  - **Execution time**: Binding is 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

- Because of swapping, a process may occupy different main memory locations during its lifetime
  - Hence physical memory references by a process cannot be fixed
- This problem is solved by distinguishing between logical address and physical address
  - Logical address : address generated by the CPU; also referred to as virtual address
  - Physical address : address seen by the memory unit
- During compile-time and load-time, logical and physical addresses are the same, but during execution-time, logical (virtual) and physical addresses are different
- Hardware device called memory-management unit (MMU) maps virtual to physical address

Dynamic relocation using a relocation register



- In a simple MMU 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

Dynamic Loading

- Routine 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; only a library to implement dynamic loading
- Implemented through program design

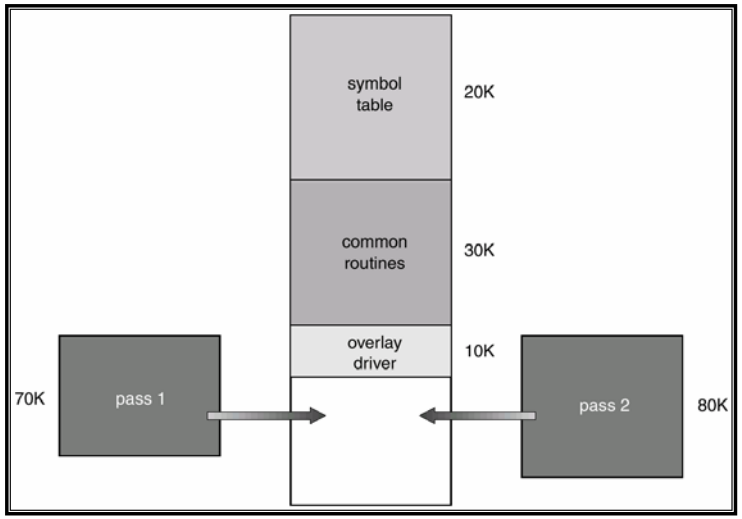
Dynamic Linking

- 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 needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries

Overlays

- Early operating systems did not have nice ways of managing "virtual" memory (more later) so everything had to fit into the (small!) physical memory
- Users developed techniques to allow large programs to fit by reusing memory when certain components weren't needed
- A program was organized (by the user) into a tree-like structure of object modules, called *overlays*

Overlays for a Two-Pass Assembler



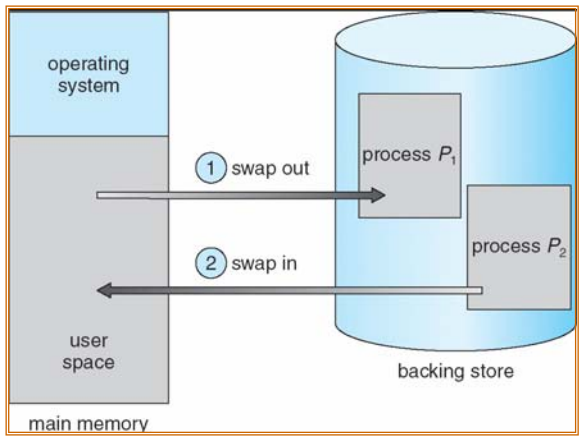
Overlays

- Keep in memory only those instructions and data that are needed at any given time
- Needed when process is larger than amount of memory allocated to it
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex
- Therefore, automatic techniques emerged to run large programs in a limited physical memory

Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory to continue execution
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped
- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Swapping (cont.)



Schematic View of Swapping

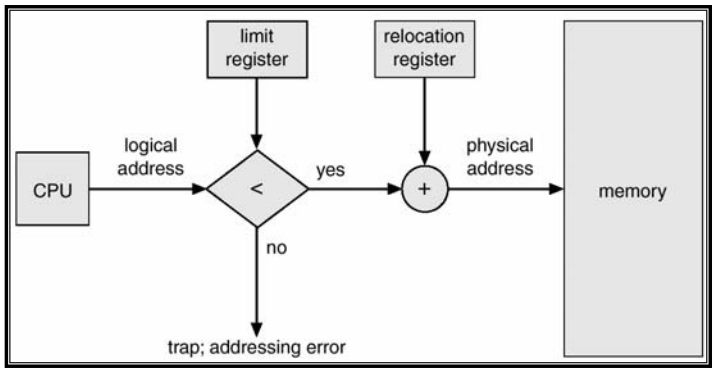
Swapping (cont.)

- The responsibilities of a swapper include:
  - Selection of processes to swap out
    - *criteria*: suspended/blocked state, low priority, time spent in memory
  - Selection of processes to swap in
    - *criteria*: time spent swapped out, priority
  - Allocation and management of swap space on a swapping device
    - Swap space may be:
      - system wide (normal)
      - dedicated to specific users/processes

Contiguous Memory Allocation

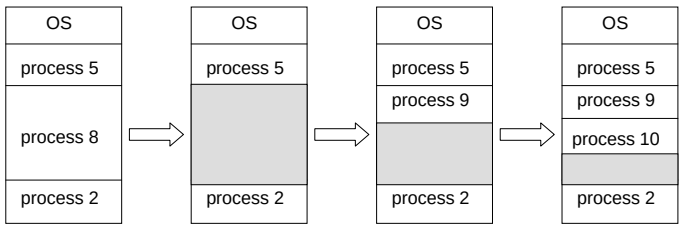
- Main memory must accommodate both the OS and the various user processes
- Main memory usually is divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes 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*

Memory Protection: Hardware Support for Relocation and Limit Registers



Continuous Memory Allocation (cont.)

- Multiple-partition allocation
  - 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
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)
- When memory is partitioned, we can have: a) fixed partition or b) dynamic partition



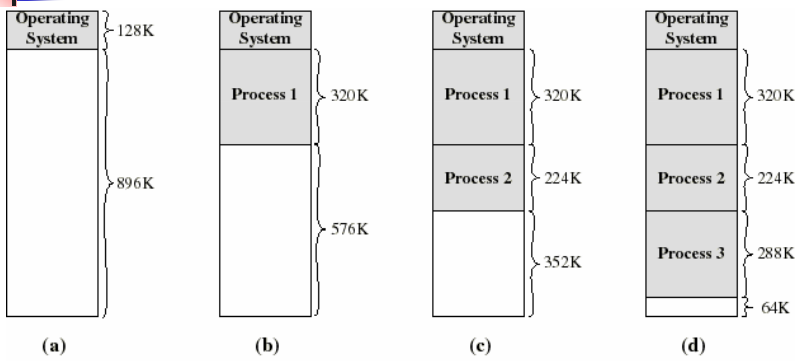
Fixed Partition

- Partition main memory into a set of non overlapping fixed-sized partitions
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation
- Unequal-size partitions lessens these problems
- Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks)
- This method is no longer used

Dynamic Partitioning

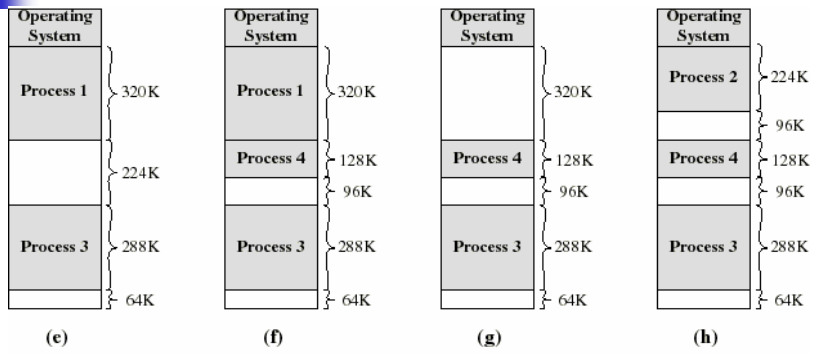
- Partitions are of variable length and number
- A process is allocated exactly as much memory as it requires
- Eventually holes are formed in main memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block
- Used in IBM's OS/MVT (Multiprogramming with a Variable number of Tasks)
- For example, assume that we have 4 processes: process 1 (320 K), process 2 (224 K), process 3 (228 K), and process 4 (128 K)

Dynamic Partitioning (Example)



- A hole of 64K is left after loading 3 processes
- Eventually each process is blocked. The OS swaps out process 2 to bring in process 4

Dynamic Partitioning (Example)



- another hole of 96K is created
- Eventually each process is blocked. The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created...
- Compaction would produce a single hole of 256K

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

- *Fragmentation* is the unintentional division of a large space into smaller, disconnected chunks of space
- There are two types of Fragmentation:
  - *Internal Fragmentation*
    - Waste of memory *within* a partition, caused by the difference between the size of a partition and the process loaded into it
    - This can be severe in static (i.e. fixed) partitioning schemes
  - *External Fragmentation*
    - Waste of memory *between* partitions, caused by scattered noncontiguous free space
    - Total memory space exists to satisfy a request, but it is not contiguous
    - Can be severe in dynamic partitioning schemes

External Fragmentation

- A 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, and is done at execution time.
  - If addresses are relocated statically at assembly or load time, then compaction is not possible
  - Problems with compaction: I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Another possible solution is to permit the logical-address space of a process to be noncontiguous

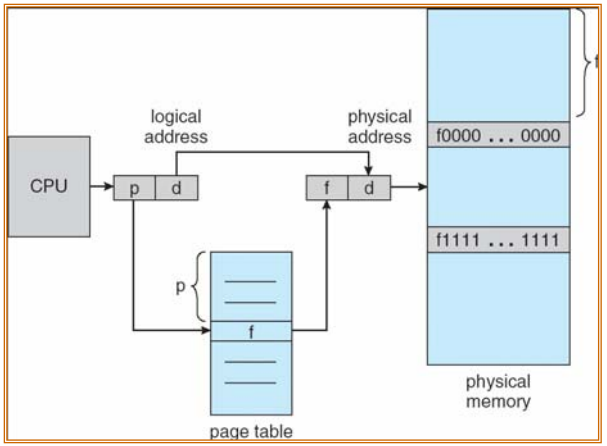
Paging

- Logical address space of a process can be noncontiguous
- Process is allocated physical memory
- Basic Method of Paging:
  - Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16MB bytes)
  - Divide logical memory into blocks of same size called **pages**
  - To run a program of size *n* pages, need to find *n* free frames
  - External fragmentation is resolved but can have internal fragmentation
    - If pages are 2,048 bytes, a process of 72,766 bytes needs 35 pages plus 1,086 bytes. That is, we need to allocate 36 frames resulting in an internal fragmentation
    - What is the size of the internal fragmentation???
  - Smaller page size is preferred but can lead to increased overhead

Address Translation Scheme

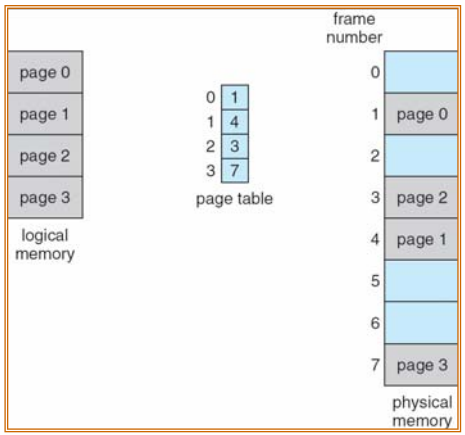
- Address generated by CPU is divided into:
  - *Page number (p)*
    - used as an index into a *page table* which contains base address of each page in physical memory
  - *Page offset (d)*
    - combined with base address to define the physical memory address that is sent to the memory unit
- Page size is defined by the hardware and is usually a power of 2
- Example:
  - If the logical address space is 2^m and a page size is 2^n
  - Then, the logical address is $p = m - n$ and $d = n$

Address Translation Architecture



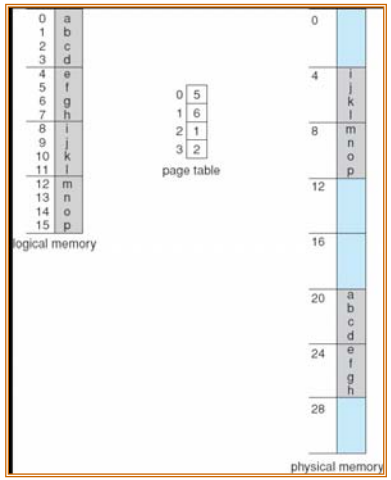
N.B. relocation can be done by changing the page table

Paging Example



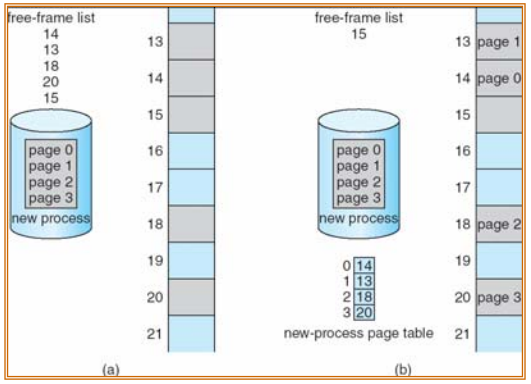
Paging Example (cont.)

- Using a page size of 4 bytes
- Physical memory of 32 bytes (i.e., 8 pages)
- The user's view of memory can be mapped into physical memory as shown in the figure
  - Example, logical address 3 (00011)
    - (page 0, and offset 3) maps to physical address 23 = 5 X 4 + 3



Free Frames

- When a process arrives to be executed, its size (expressed in pages) is examined
  - Each page of the process needs one frame
  - If the process requires n pages, at least n frames must be available in memory
  - Allocate pages to frames in the free frame list and insert a record in the page table



Before allocation After allocation

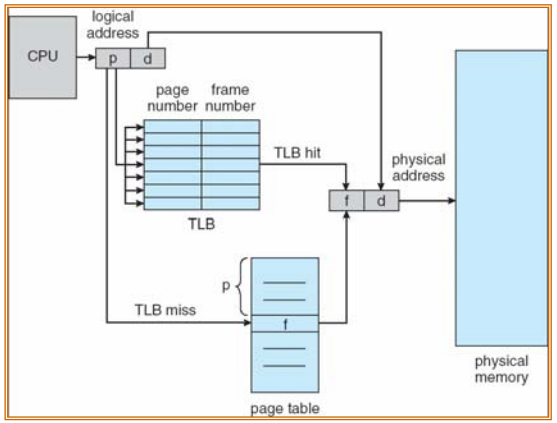
Implementation of Page Table

- Each OS has its own methods of storing page tables
  - Some OSs allocate a page table for each process
  - A pointer to the page table is stored in the process control block (PCB)
  - When a dispatcher starts a process, it must define the correct hardware page table values from the stored page table
- Hardware implementation of page table
  - Implement the page table as a set of dedicated registers
    - This method is satisfactory if the page table is reasonably small (e.g. 256 entries)
    - These registers must be very efficient in paging address translation
    - Unfortunately, page tables can be very large (one million entries)
      - **Must have alternatives**

Implementation of Page Table (cont.)

- Page table is kept in main memory
- Page-table base register (PTBR) points to the page table
  - Changing between page tables requires changing only this one register. This reduces the time for context switching
- The problem with this scheme is the time required to access a user memory location (two memory accesses, why?)
- The two memory access problem can be solved by the use of a special, fast hardware cache called translation look-aside buffer (TLB) (an associate high speed memory)

Paging Hardware With TLB



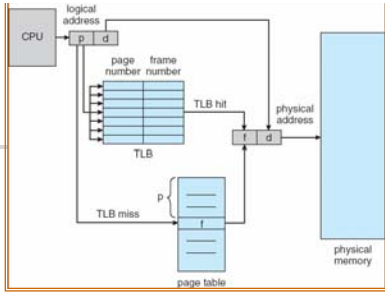
Associative Memory

- Associative memory is a special, small, and fast (but expensive) lookup hardware
- Associative memory -- parallel search

Page #	Frame #

- When the associative memory is represented with the page number, the page number is compared with all frames simultaneously
  - If a frame is found, get frame # out
- Otherwise get frame # from page table in memory

Effective Access Time



- Associative Lookup (i.e., to find the desired page number in TLB) = ε time unit
- Assume memory cycle time (i.e., to access memory) is *m* microseconds
- Hit ratio is percentage of times a page number is found in the associative registers
- Hit ratio = α
- Effective Access Time (EAT)

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

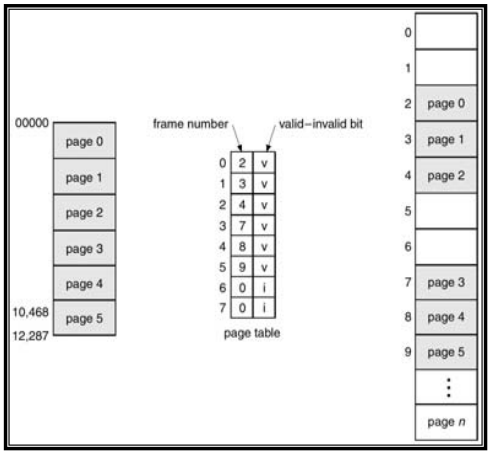
  $= 2m + \varepsilon - \alpha m$

Memory Protection

- Memory protection implemented by associating protection bit with each frame
- Valid-invalid bit is attached to each entry in page table:
  - "valid" indicates the associated page is in the process' logical address space
  - "invalid" indicates the page is not in the process' logical address space
- We can easily extend this approach to provide a finer level of protection
  - Read-only, read-write, or execute-only

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

- In a system with 14-bit address space (0 to 16383)
- We may have a program that uses only addresses 0 to 10,468
- Given a page size of 2 KB, we get the situation in the figure



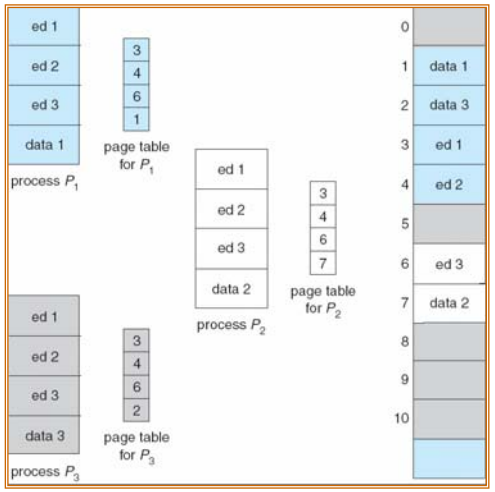
Outline

- Shared pages
- Page table structure
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- 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



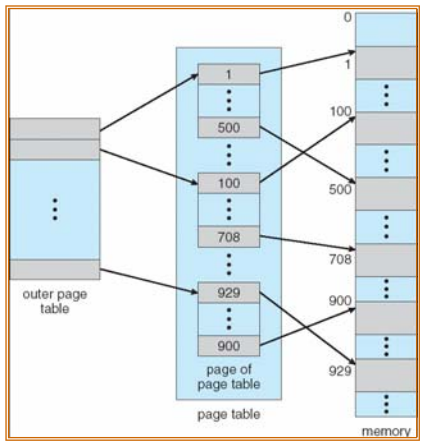
Hierarchical Page Tables

- Most modern OS support large logical address space (2^{32} to 2^{64})
- Consider a system with 32-bit logical address space and 4 KB page size
  - Assume that each entry consists of 4 bytes, then each process may need up to 4 MB of physical space just to allocate the page table (in contiguous)
- Solution: Break up the logical address space into multiple page tables
- A simple technique is a two-level page table (to spread the table)
- Two-Level Paging Example :
  - A logical address (on 32-bit machine with 4K page size) is divided into:

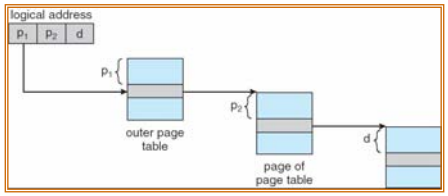
page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Hierarchical Page Tables (cont.)



A two-level page table scheme

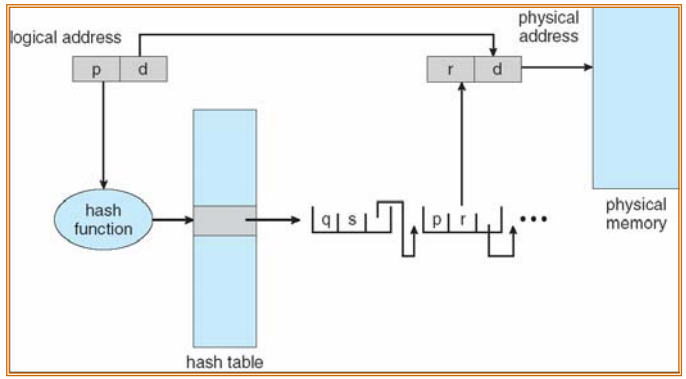


Address Translation for a two-level 32-bit paging architecture

Hashed Page Tables

- Common in address spaces > 32 bits
- The logical (virtual) page number is hashed into a page table
- This page table contains a chain of elements hashed to the same location
- Each element consists of
  - The logical page number
  - The value of the mapped page frame
  - A pointer to the next element in the linked list
- Logical page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted

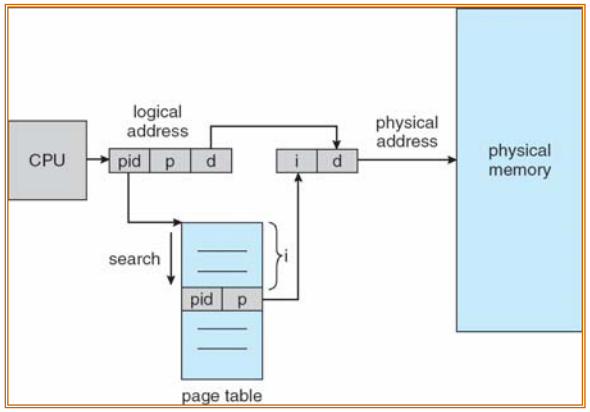
Hashed Page Table Architecture



Inverted Page Tables

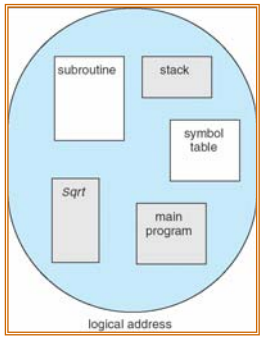
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Tables Architecture



Segmentation

- Memory-management scheme supports user view of memory
- A program is a collection of segments. A segment is a variable-size logical unit such as
  - Main program
  - Procedure
  - Function
  - Object
  - Local variables, global variables
  - Stack
  - Arrays, etc



Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number, offset>
- *Segment table* – maps two-dimensional physical addresses
- Each table entry has:
  - base – starting physical address where 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;
  segment number *s* is legal if *s* < STLR

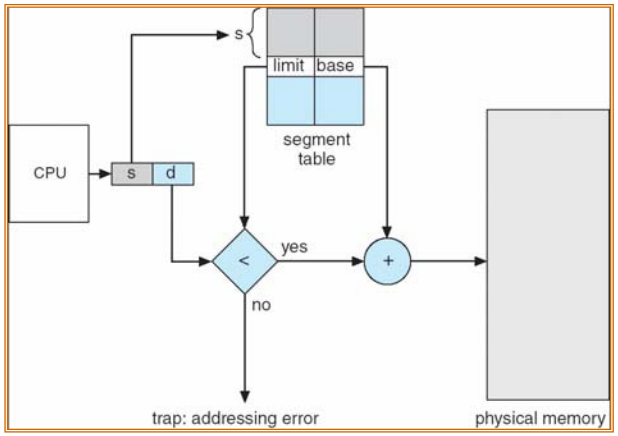
Segmentation Architecture

- Relocation
  - dynamic
  - by segment table
- Sharing
  - shared segments
  - same segment number
- Allocation
  - first fit/best fit
  - external fragmentation

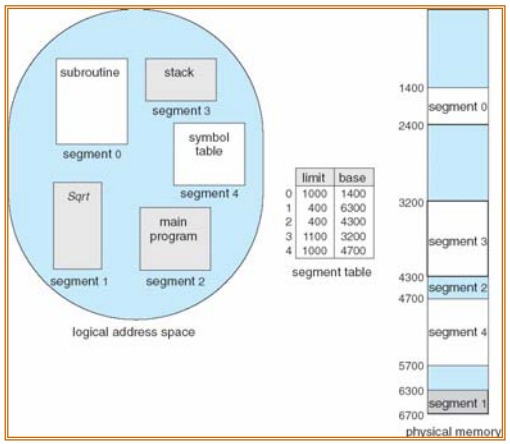
Segmentation Architecture

- Protection: with each entry in segment table associate:
  - validation bit = 0 ⇒ 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
- A segmentation example is shown in the following diagram

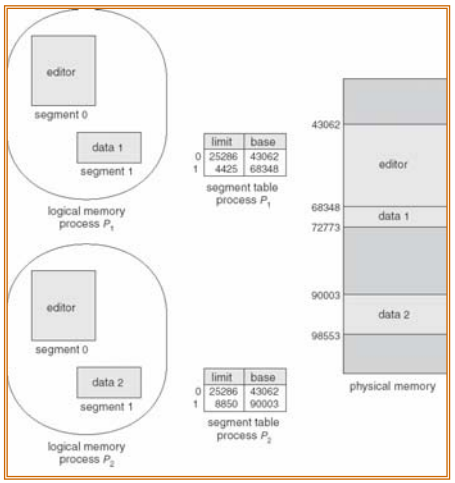
Segmentation Hardware



Segmentation Example



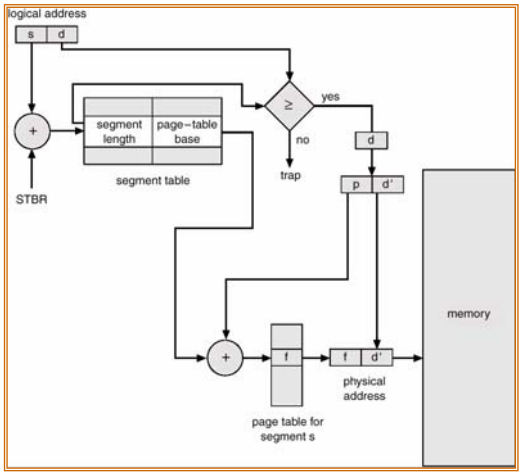
Sharing of Segments



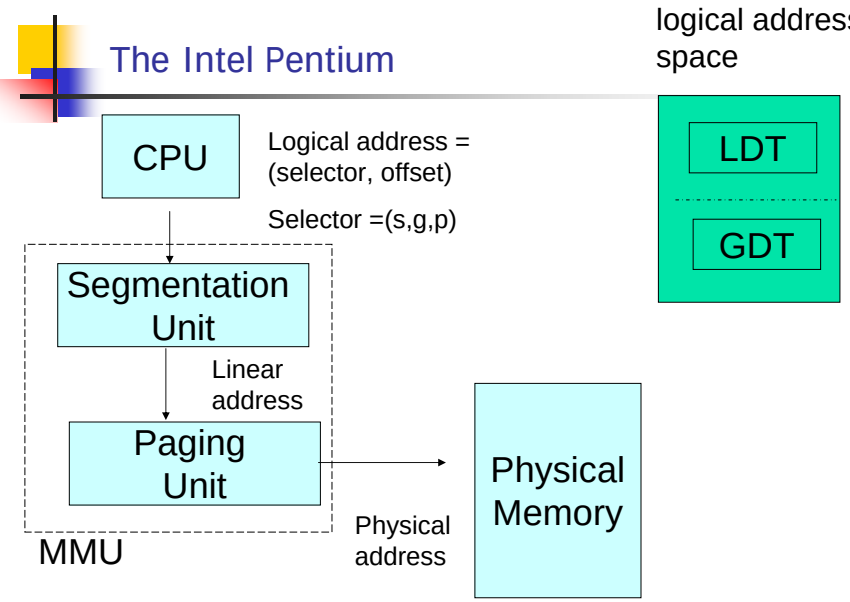
Segmentation with Paging

- Solves problems of external fragmentation and lengthy search
- Implementation:
  - Each segment is broken into pages
  - Each segment has a page table
  - Each entry of the segment table has a segment base and a segment limit. The segment limit is used to check for address validity
  - The linear address is divided into a page number and a page offset
  - The corresponding physical address is found by using page table
- Segmentation with Paging differs from pure segmentation
  - The segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment

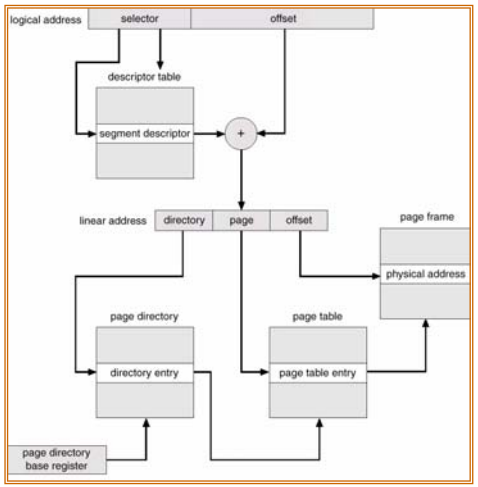
MULTICS Address Translation Scheme



The Intel Pentium



Intel 30386 Address Translation



Linux on Intel 80x86

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode

Memory Management: Summary

- Address Binding
- Swapping
- Contiguous Memory Allocation
  - Internal and external fragmentation
- Paging
  - Page table structure
- Segmentation
- Segmentation with Paging

End of Chapter 8

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005