BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE 8
MEMORY MANAGEMENT STRATEGIES (CHAPTER 8)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium

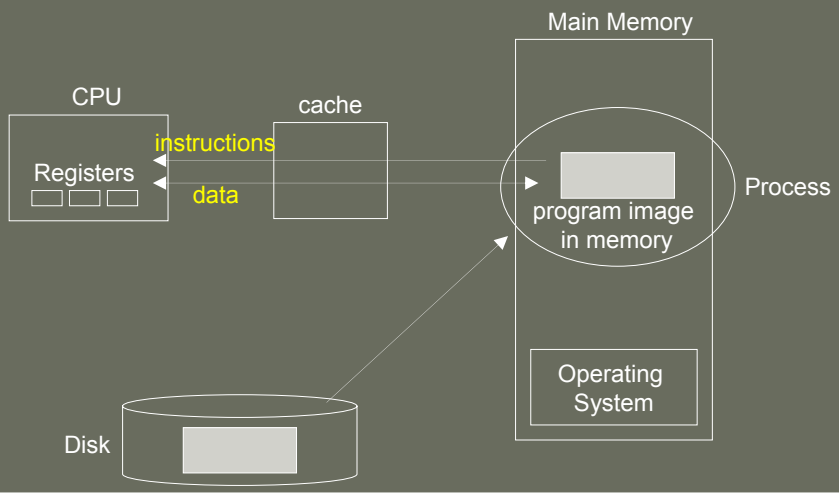
Objectives

- 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

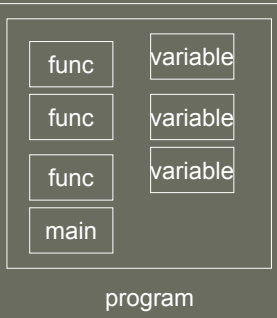
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Background

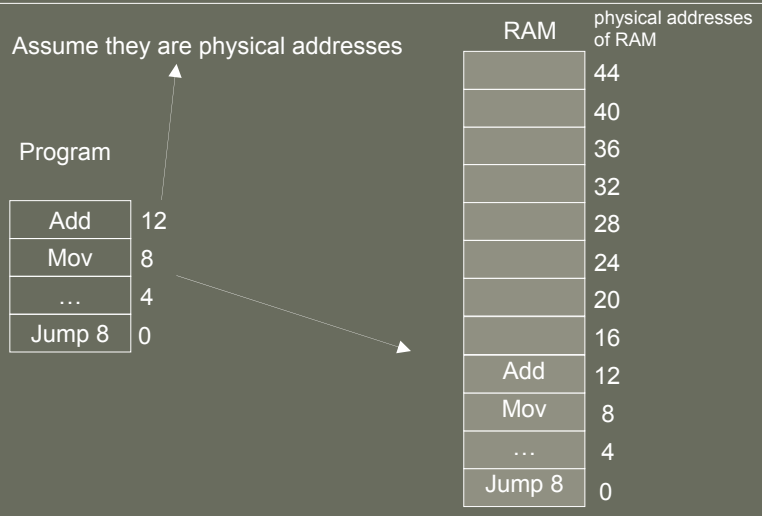


Program addresses and memory

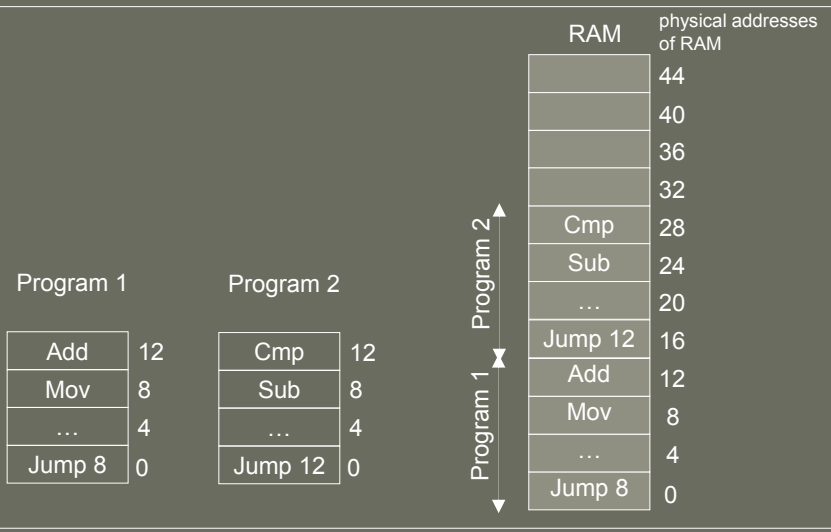
- When code is generated (or assembly program is written) we use memory addresses for variables, functions and branching/jumping.
- Those addresses can be physical or logical memory addresses.
- In very early systems they are just physical memory addresses.
- A program has to be loaded to that address to run.
  - No relocation



Program addresses and memory

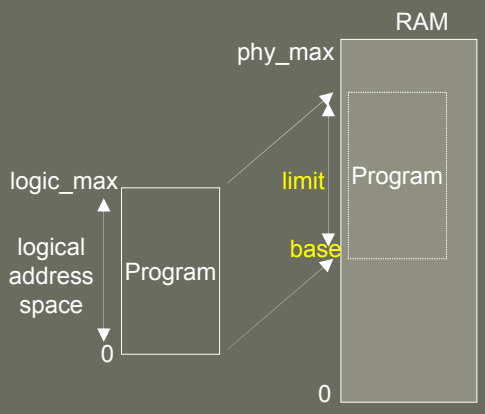


Program addresses and memory



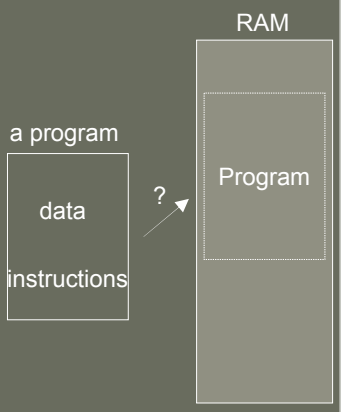
Logical address space concept

- We need logical address space concept, that is different that the physical RAM (main memory) addresses.
- A program uses logical addresses.
- Set of logical addresses used by the program is its logical address space
  - Logical address space can be, for example, [0, max_address]
- Logical address space has to be mapped somewhere in phsyical memory



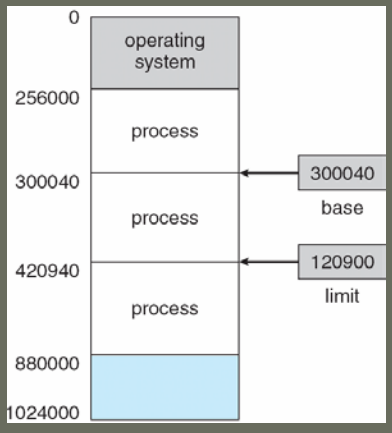
Binding of Instructions and Data to Memory

- Address binding of instructions and data to (physical) 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)

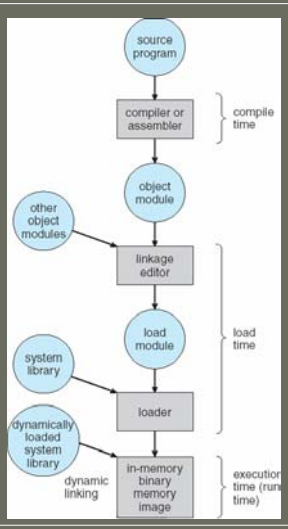


Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space



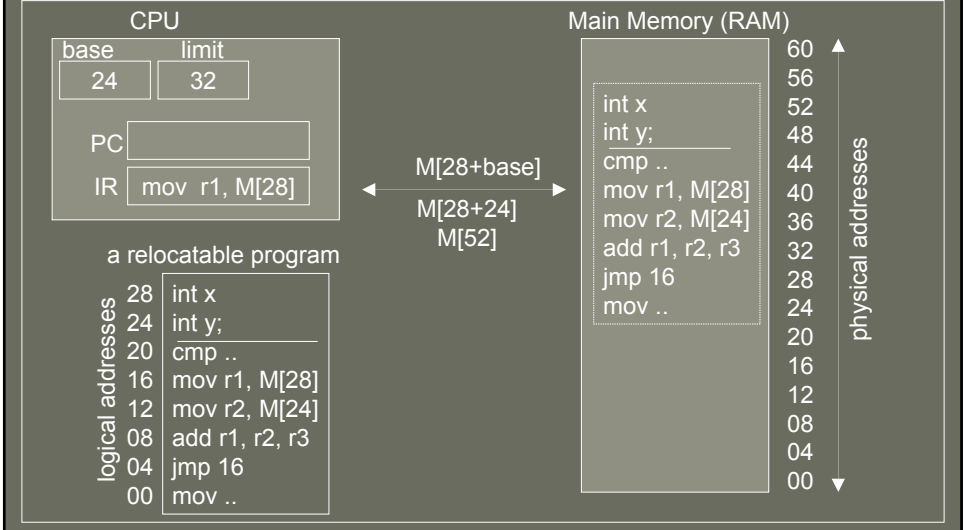
Multistep Processing of a User Program



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

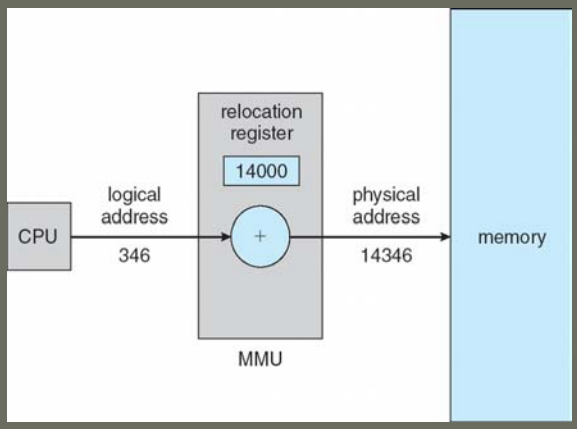
Logical and physical addresses



Memory-Management Unit (MMU)

- Hardware device that maps logical (virtual) to physical address
- In 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
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



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 implemented through program design

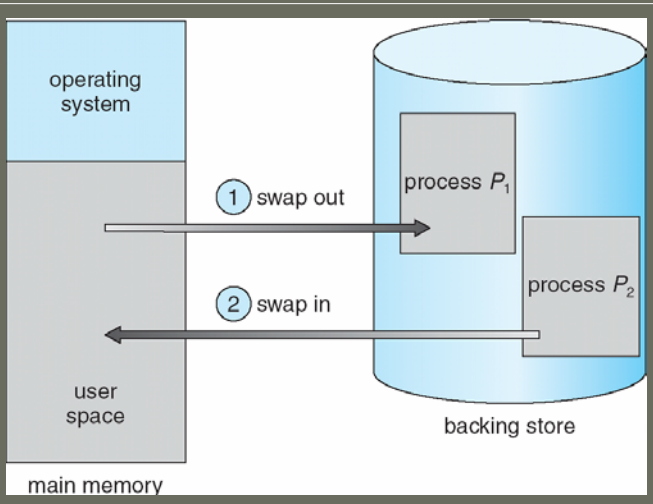
Dynamic Linking

- 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
  - Standard C library is shared library that is dynamically linked, not statically linked.
  - You can link statically if you want.
- System also known as **shared libraries**

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 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

Schematic View of Swapping



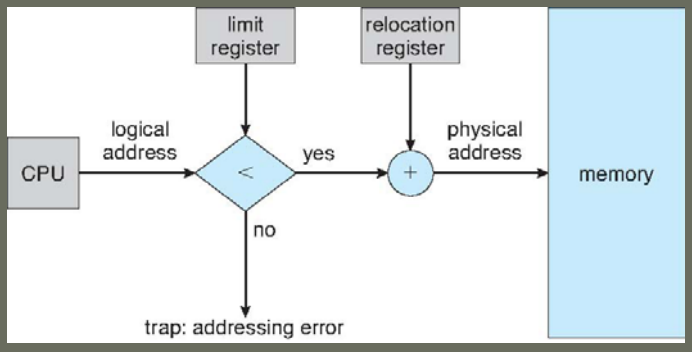
Contiguous Allocation

- Main memory is partitioned usually 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 addresses *dynamically*

Basic Memory Allocation Strategies

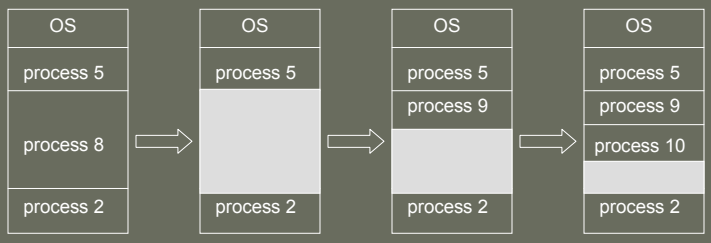
- In this chapter, we will cover 3 basic main memory allocation strategies to processes
  - 1) Contiguous allocation
  - 2) Paging
  - 3) Segmentation

Hardware Support for Relocation and Limit Registers



Contiguous 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)



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

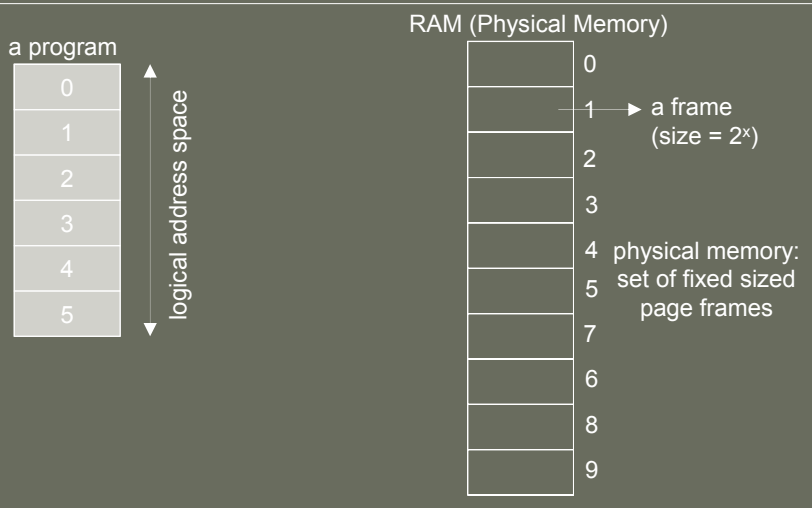
Fragmentation

- **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 (allocation), but not being used
- 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

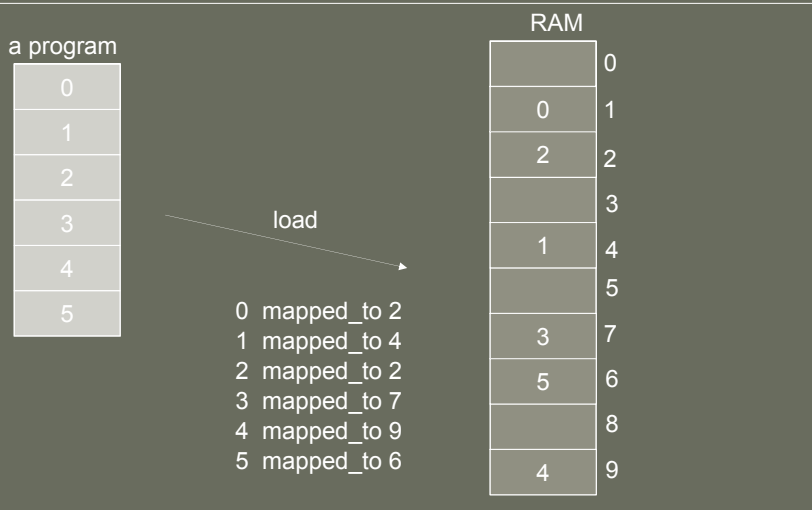
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Physical address space will also be noncontiguous.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- 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
- Internal fragmentation

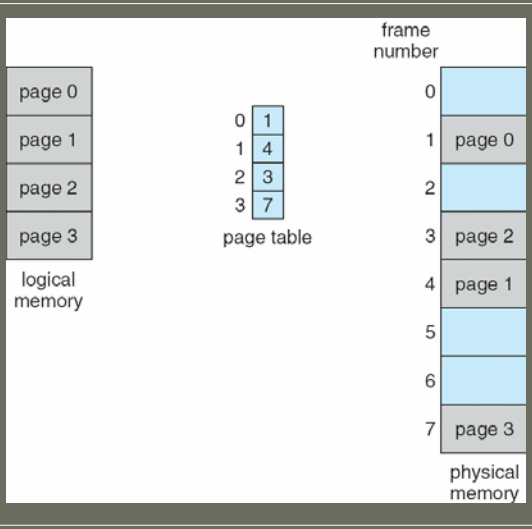
Paging



Paging



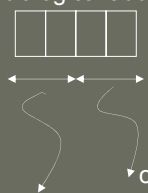
Paging Model of Logical and Physical Memory

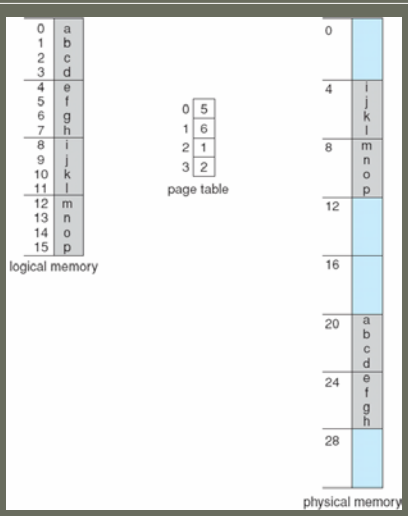


Paging Example

page size = 4 bytes
= 2^2

4 bit logical address





LA = 5
PA = ?

5 is 0101
PA = 11001

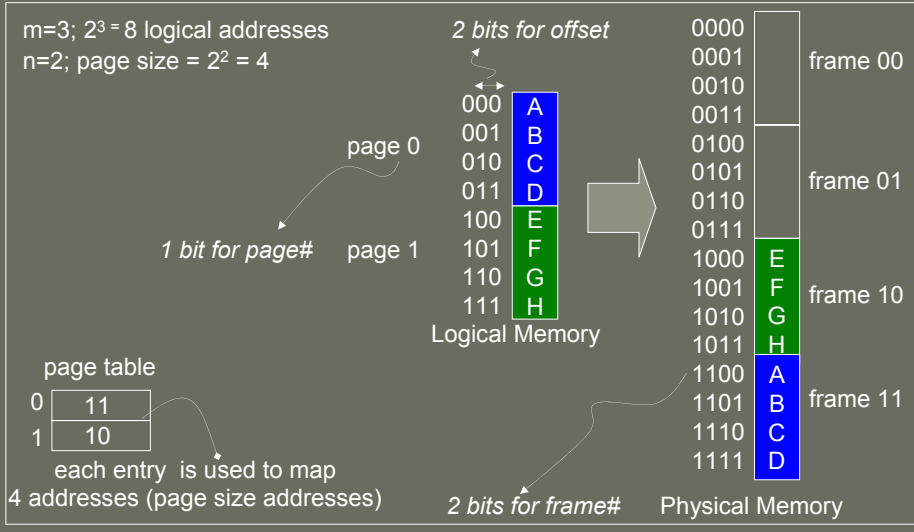
LA = 11
PA = ?

11 is 1011
PA = 00111

LA = 13
PA = ?

13 is 1101
PA = 01001

Address Translation (Mapping)



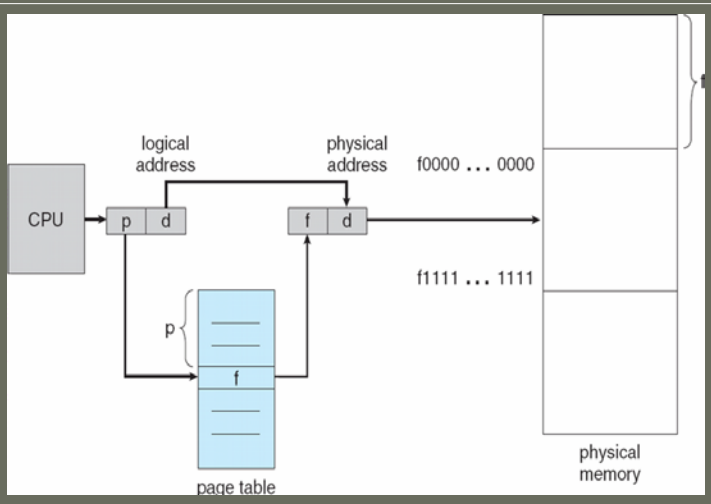
Address Translation Scheme

- 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 number	page offset
p	*d*
m - n	*n*

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

Paging Hardware



Example

16 bit logical address

0010000000000100

p# offset

mapping

f# offset

1100000000000100

15 bit physical address

15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

page table

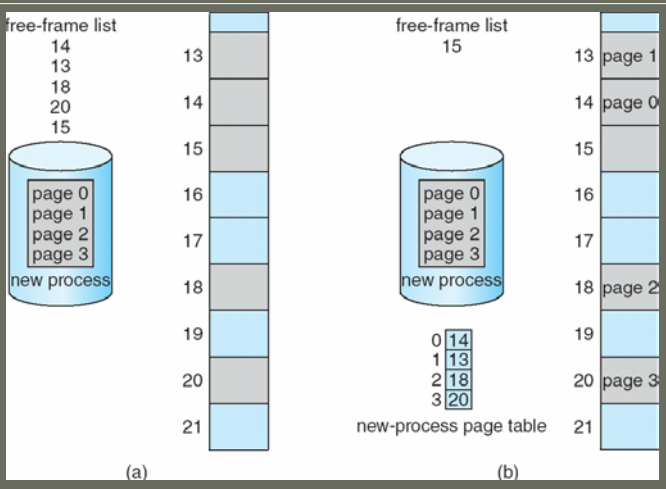
frame number

valid/invalid bit

page size = 4096 bytes

(offset is 12 bits)

Free Frames



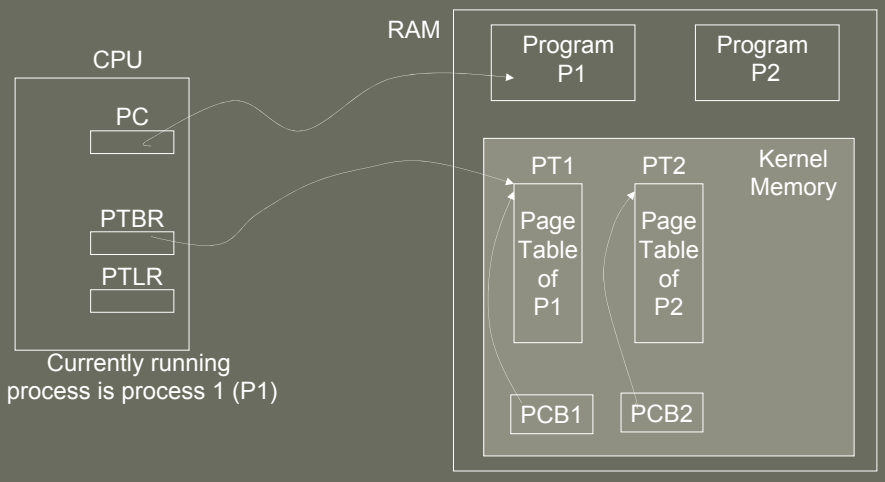
Before allocation After allocation

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.

- 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)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Implementation of Page Table



TLB Associative Memory

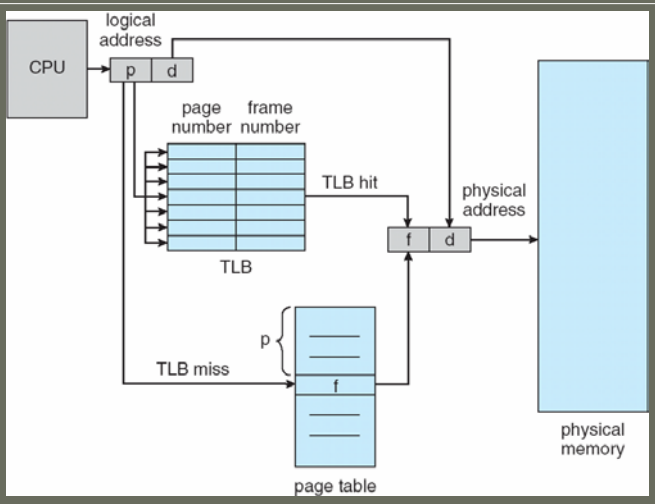
- Associative memory – parallel search

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 Memory Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α

- **Effective Access Time** (EAT)

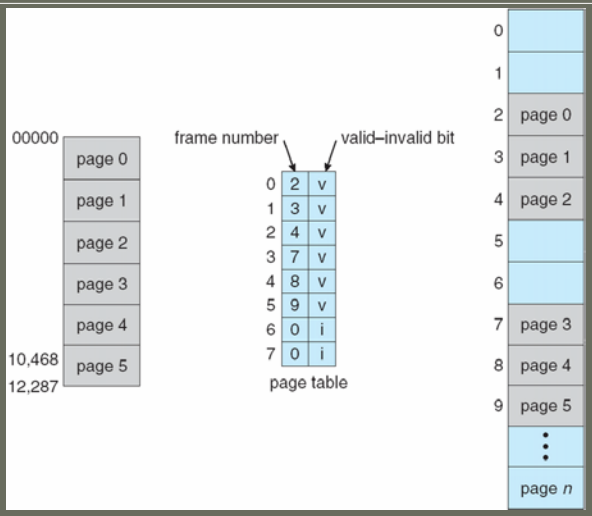
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame

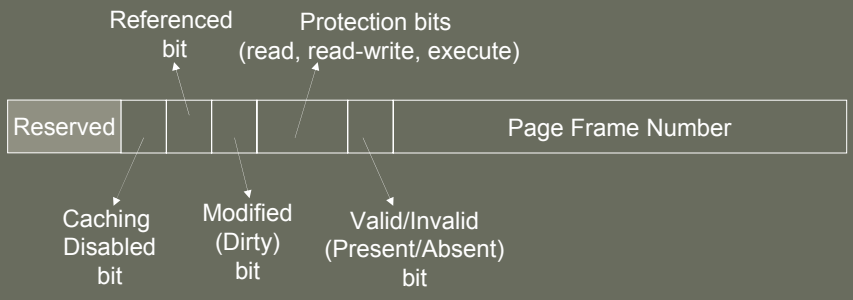
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

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



Page Table Entry Structure

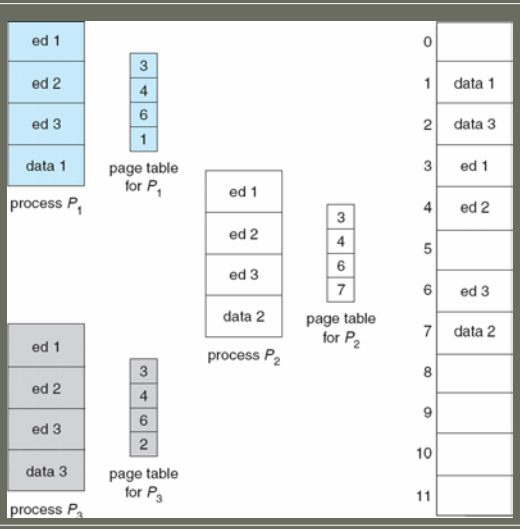
- A typical size of a page table entry can be 32 bits. Depends on the architecture
- Typically we have the following fields in a page table entry.



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



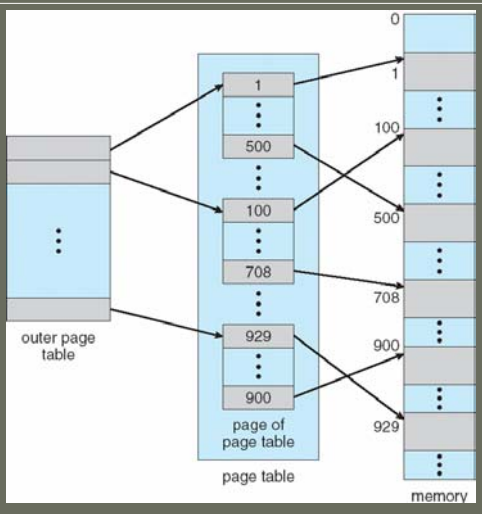
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

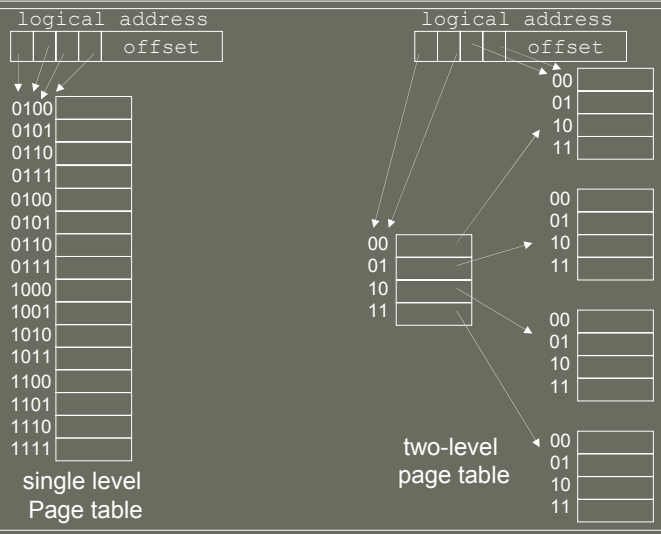
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

Two-Level Paging Scheme

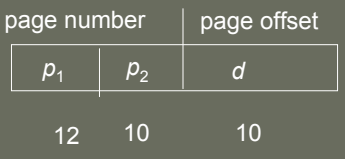


Two-Level Paging Scheme



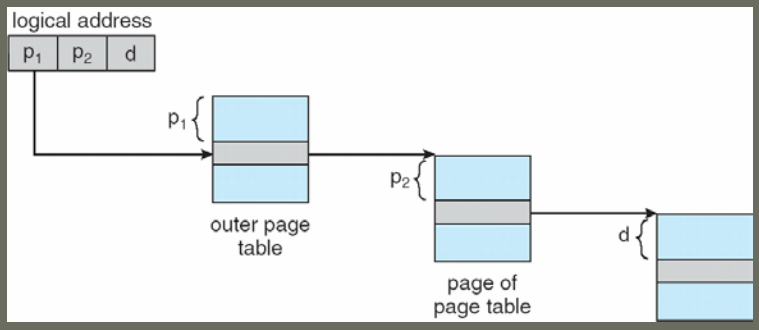
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



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

Address-Translation Scheme



Example: two level page table need

logical address length = 32 bits
pagesize = 4096 bytes
logical address division: 10, 10, 12

2^{32} bytes = 4 GB

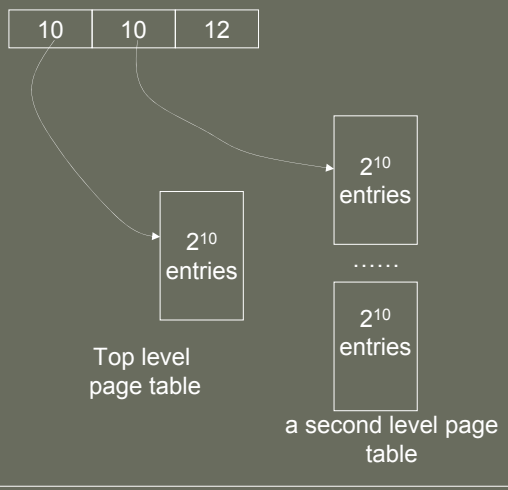
logical address space size
(only partially used)

8 MB

12 MB

What is total size of two level page table if entry size is 4 bytes?

Example: two level page table need

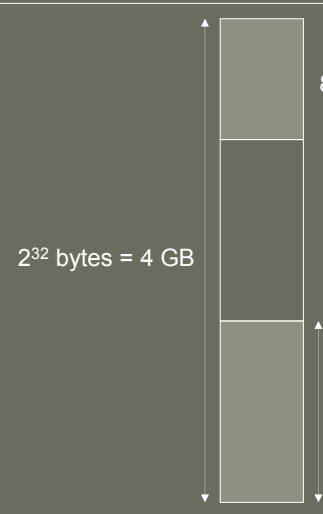


Each entry of a second level page table translates a page# to a frame#; i.e. each entry maps a page which is 4096 bytes

There are 1024 entries In a second level page table

Hence, a second level page table can map $2^{10} * 2^{12} = 2^{22} = 4$ MB of logical address space

Example: two level page table need

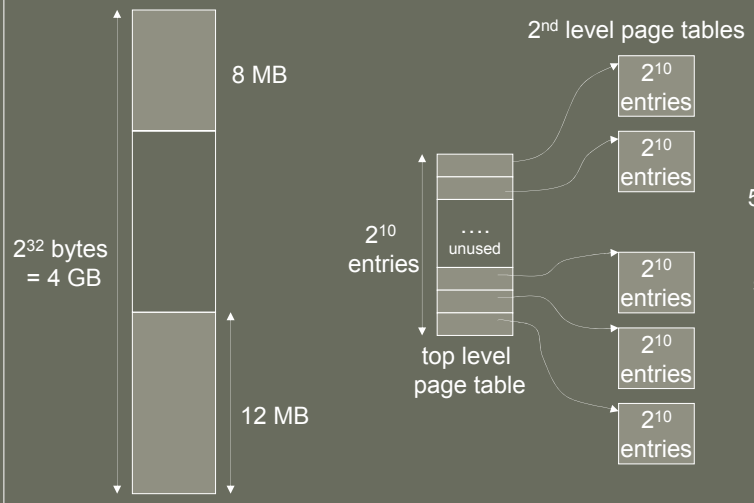


8 MB / 4 MB = 2 second level page tables required to map 8 MB of logical memory

Total = 3 + 2 = 5 second level page tables required

12 MB / 4 MB = 3 second level page tables required to map 12 MB of logical memory

Example: two level page table need



Three-level Paging Scheme

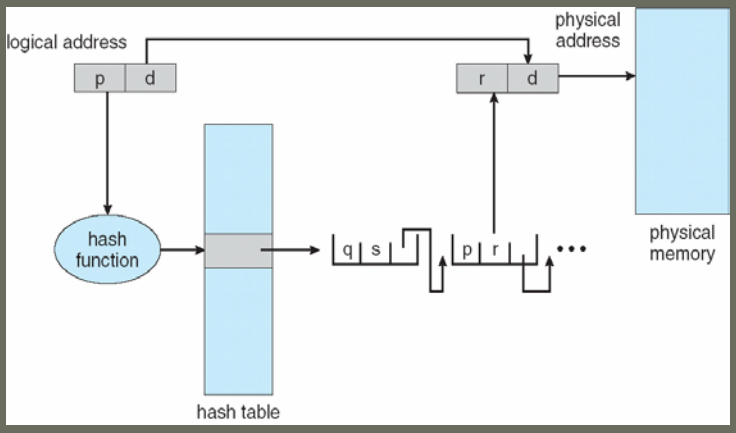
outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

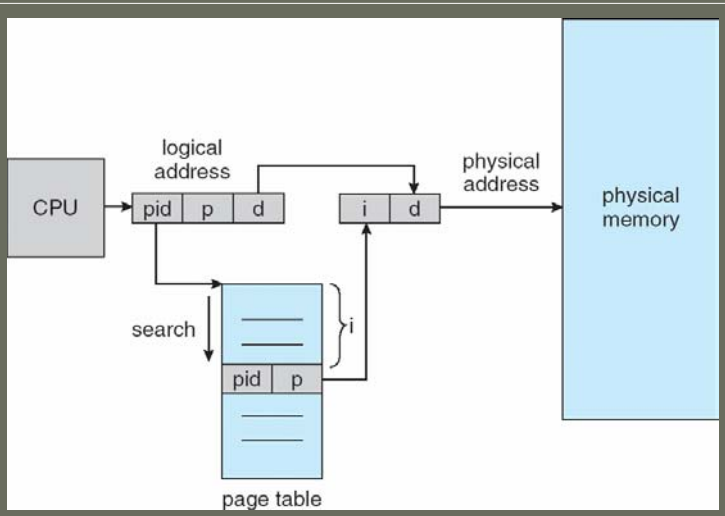
Hashed Page Table



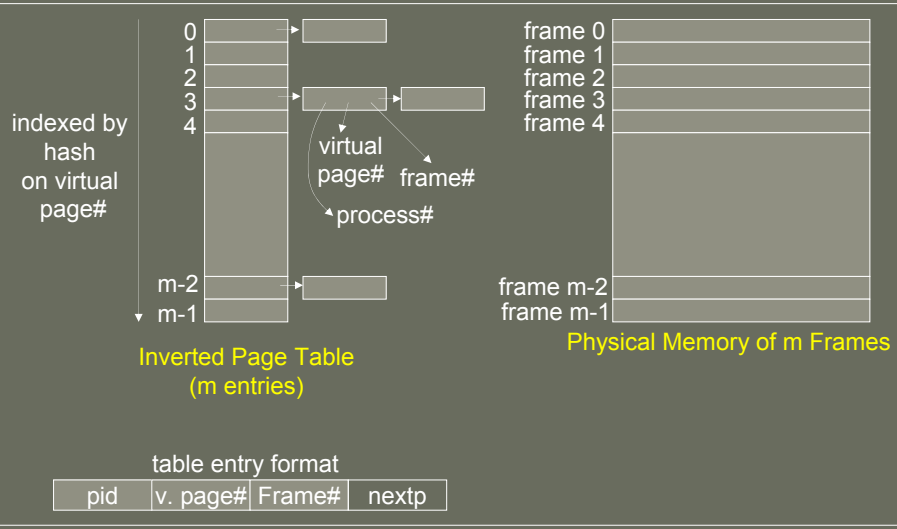
Inverted Page Table

- 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

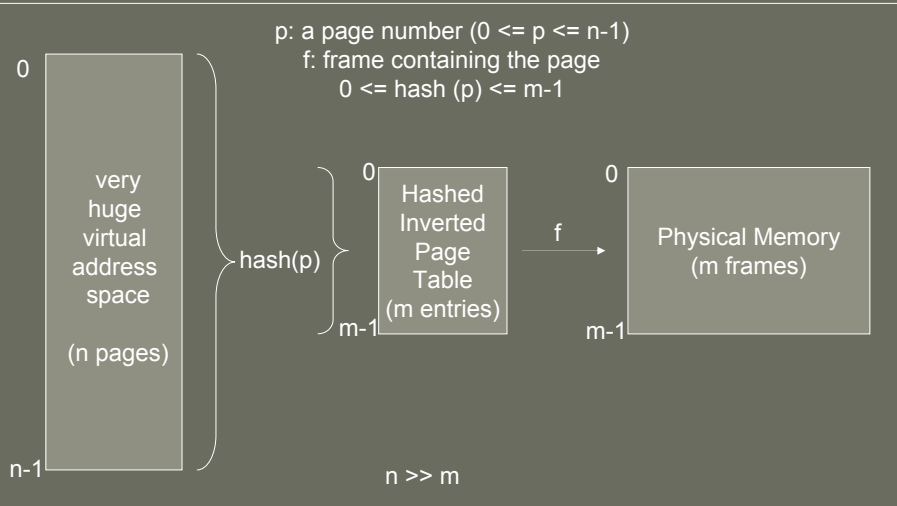
Inverted Page Table Architecture



Inverted Page Table with Hashing



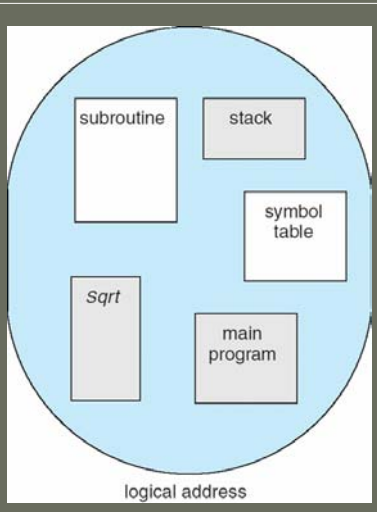
Inverted Page Table with Hashing



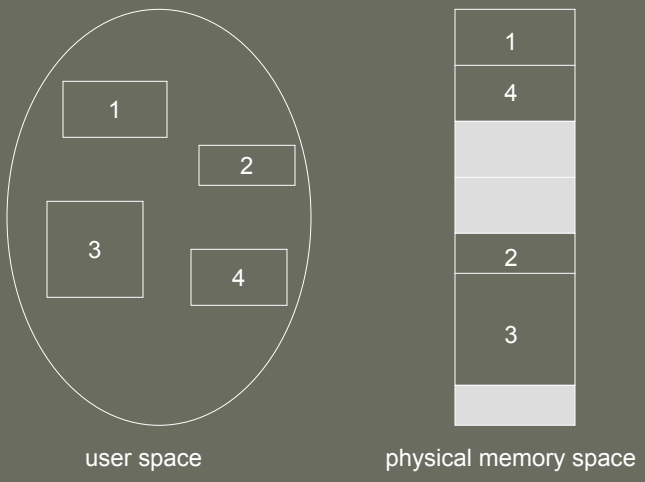
Segmentation

- 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
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

User's View of a Program



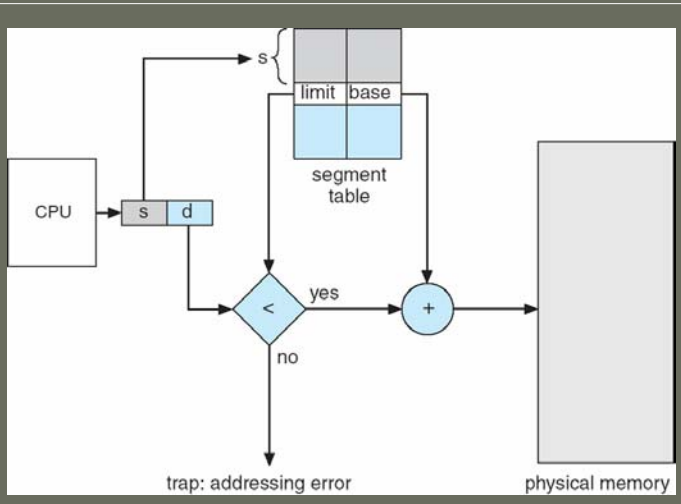
Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number, offset>,
- **Segment table** – maps two-dimensional logical 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 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**

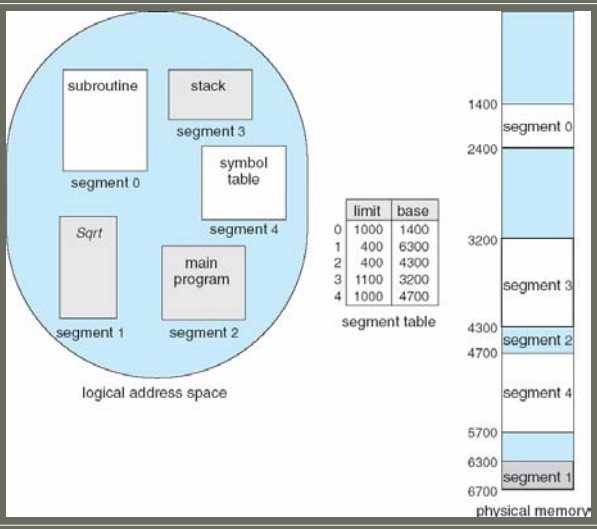
Segmentation Hardware



Segmentation Architecture (Cont.)

- 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
  - Code segment: READONLY; sharable; …
  - Data segment: RED-WRITE; not-sharable
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

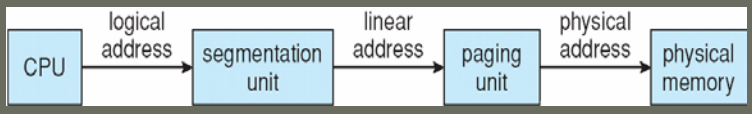
Example of Segmentation



Example: The Intel Pentium

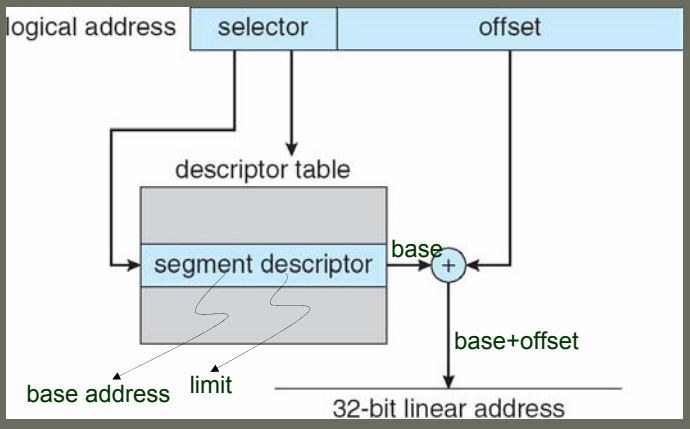
- Supports both segmentation and segmentation with paging
- CPU generates logical address (<segment#, offset> pairs)
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

Logical to Physical Address Translation in Pentium



page number		page offset
p_1	p_2	d
10	10	12

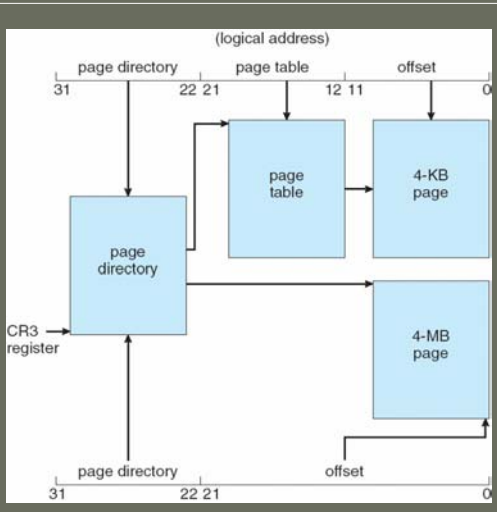
Intel Pentium Segmentation



Pentium Paging Architecture

- Pentium architecture allows a page size of either 4 KB or 4 MB.
- For 4 KB pages, two-level paging scheme is used in a 32 bit machine
  - Address division: <10, 10, 12> bits
- For 4 MB pages, we can skip the inner page tables (secondary page table). A top level page table entry will point directly to a 4 MB page.

Pentium Paging Architecture



Linux on Pentium: Segmentation

- Linux is designed to run on a lot of hardware platforms: Pentium, Arm, Motorola, Sparc, MIPS, …
- Therefore it does not rely on segmentation and makes minimal use of segmentation in Pentium.

Linux on Pentium: Paging

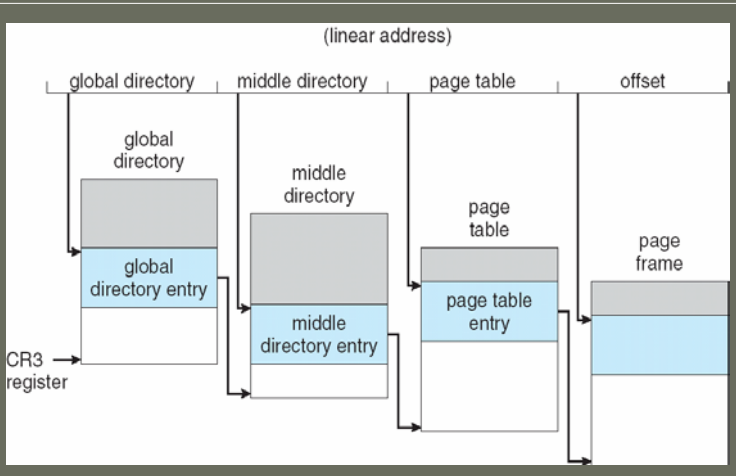
- Linux can run both on 32 bit and 64 bit machines.
  - Therefore having just two level paging is not enough.
  - Linux adapted a three level paging that can be used both in 32bit and 64bit machines.
- A linear address in Linux is broken into 4 parts:
  - P1: global directory
  - P2: middle directory
  - P3: page table
  - P4: offset
- Number of bits in each part depends on the architecture

Linear Address in Linux

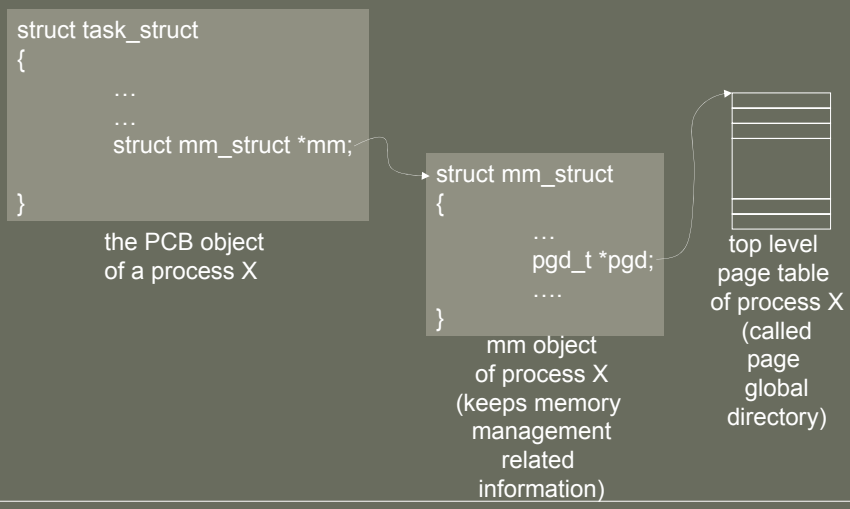
Broken into four parts:

- In a 64 bit machine (very large address space), we use all 4 parts
- In a 32 bit Intel machine, that uses two-level paging, we ignore the middle directory (its size is 0), and use 3 parts.

Three-level Paging in Linux



Linux Paging: some kernel structures



End of Lecture