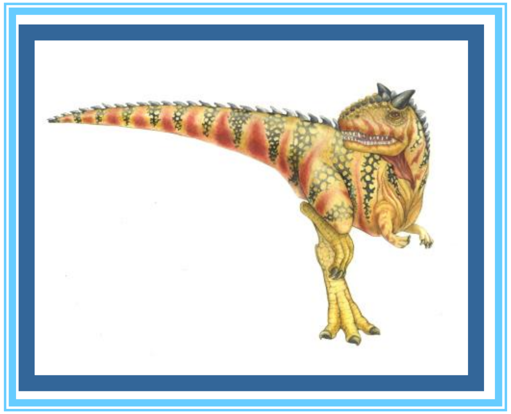
Chapter 8: Memory-Management Strategies



Chapter 8: Memory Management Strategies

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

Objectives

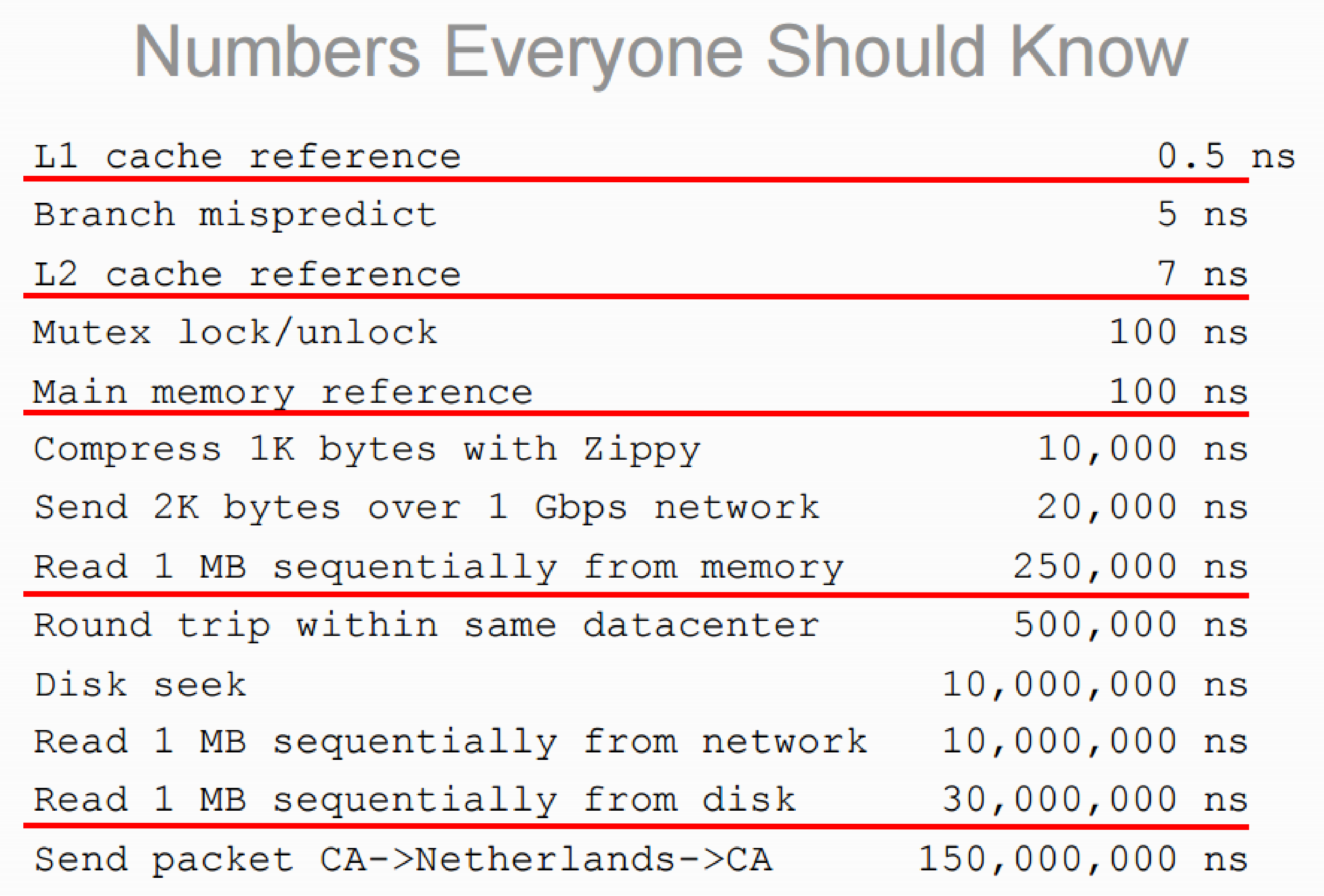
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

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

- Jeff Dean：

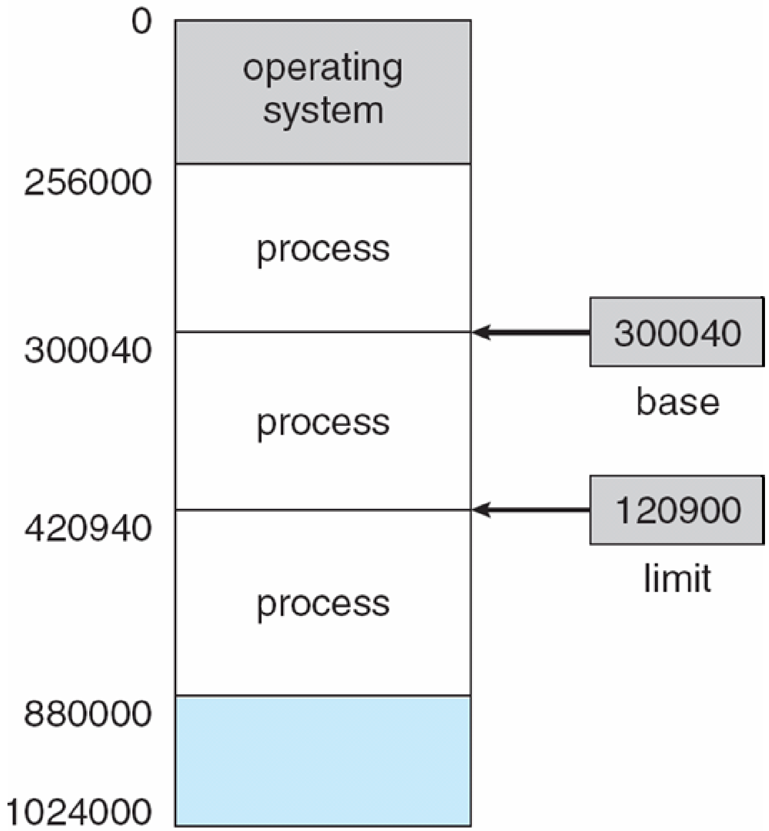


Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Base and Limit Registers

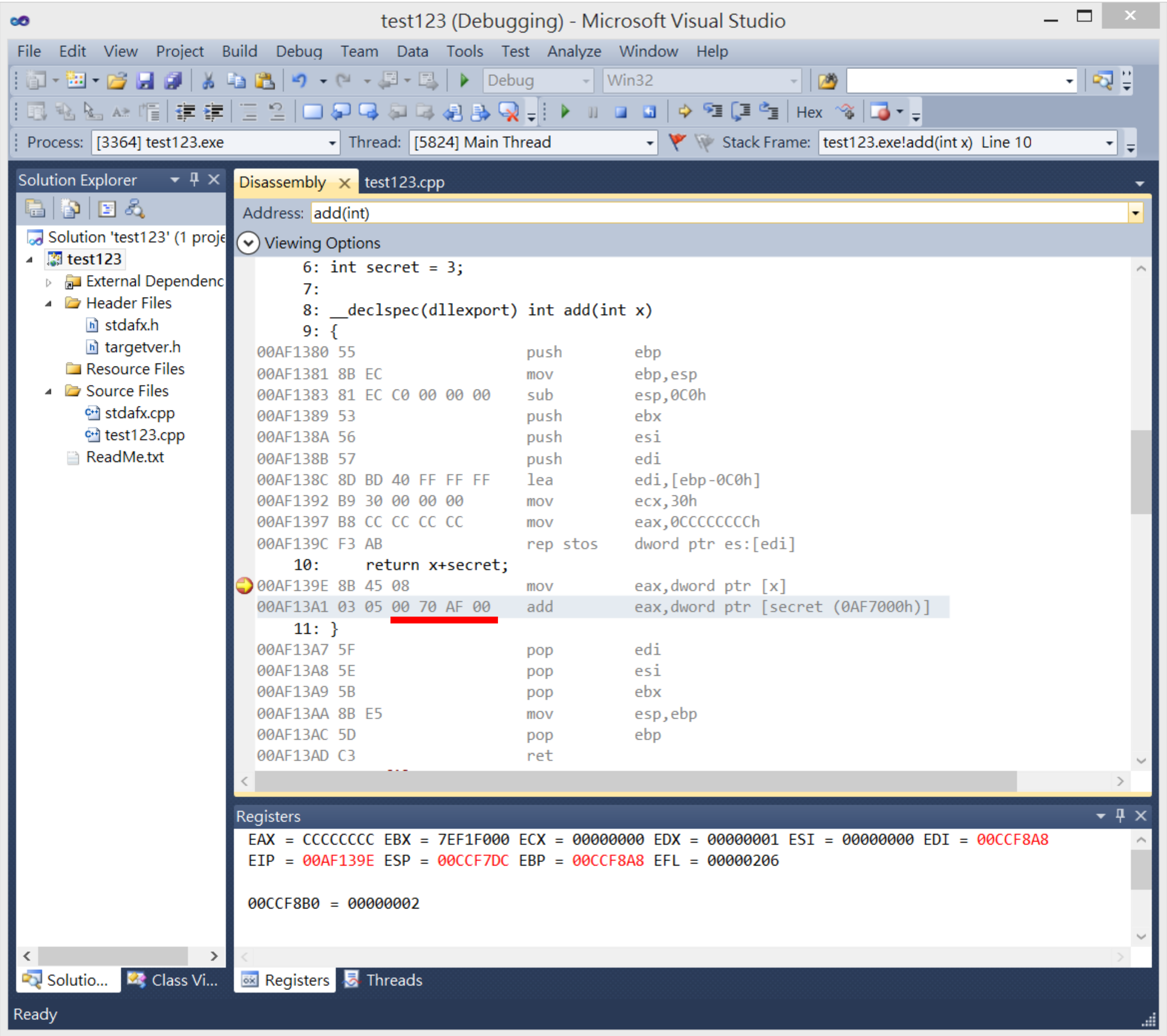
- A pair of **base** and **limit** registers define the logical address space



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)

Binding of Instructions and Data to Memory



Process: [3364] test123.exe

Thread: [5824] Main Thread

Stack Frame: test123.exe!add(int x) Line 10

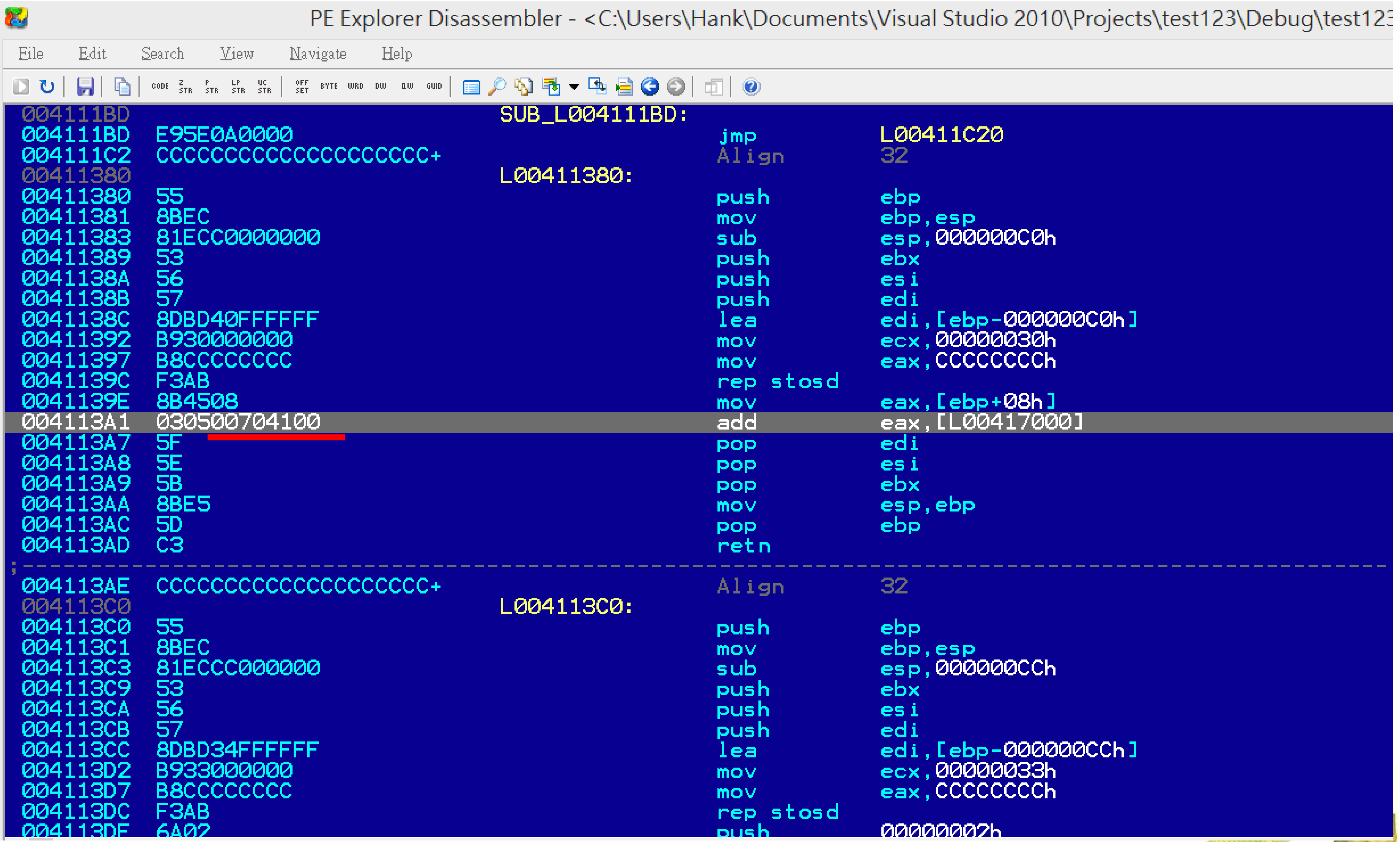
Address: add(int)

```
     6: int secret = 3;
     7: 
     8: __declspec(dllexport) int add(int x)
     9: {
00AF1380 55                   push        ebp
00AF1381 8B EC                mov         ebp,esp
00AF1383 81 EC C0 00 00 00    sub         esp,0C0h
00AF1389 53                   push        ebx
00AF138A 56                   push        esi
00AF138B 57                   push        edi
00AF138C 8D BD 40 FF FF FF    lea         edi,[ebp-0C0h]
00AF1392 B9 30 00 00 00       mov         ecx,30h
00AF1397 B8 CC CC CC CC       mov         eax,0CCCCCCCCh
00AF139C F3 AB                rep stos    dword ptr es:[edi]
    10:     return x+secret;
00AF139E 8B 45 08             mov         eax,dword ptr [x]
00AF13A1 03 05 00 70 AF 00    add         eax,dword ptr [secret (0AF7000h)]
    11: }
00AF13A7 5F                   pop         edi
00AF13A8 5E                   pop         esi
00AF13A9 5B                   pop         ebx
00AF13AA 8B E5                mov         esp,ebp
00AF13AC 5D                   pop         ebp
00AF13AD C3                   ret
```

```
EAX = CCCCCCCC EBX = 7EF1F000 ECX = 00000000 EDX = 00000001 ESI = 00000000 EDI = 00CCF8A8
EIP = 00AF139E ESP = 00CCF7DC EBP = 00CCF8A8 EFL = 00000206

00CCF8B0 = 00000002
```

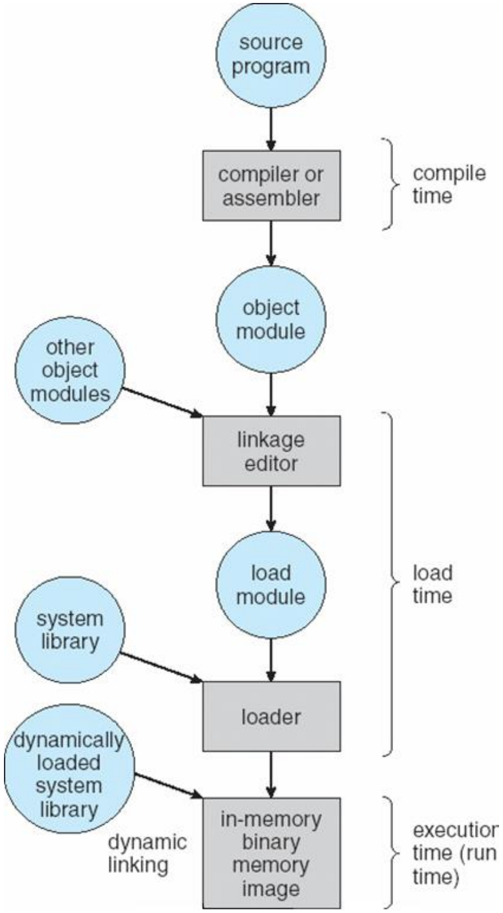
Binding of Instructions and Data to Memory



PE Explorer Disassembler - <C:\Users\Hank\Documents\Visual Studio 2010\Projects\test123\Debug\test123
File
Edit
Search
View
Navigate
Help

```
004111BD                                    SUB_L004111BD:
004111BD  E95E0A0000                                          jmp          L00411C20
004111C2  CCCCCCCCCCCCCCCCCCCC+                               Align        32
00411380                                    L00411380:
00411380  55                                                  push         ebp
00411381  8BEC                                                mov          ebp,esp
00411383  81ECC0000000                                        sub          esp,000000C0h
00411389  53                                                  push         ebx
0041138A  56                                                  push         esi
0041138B  57                                                  push         edi
0041138C  8DBD40FFFFFF                                        lea          edi,[ebp-000000C0h]
00411392  B930000000                                          mov          ecx,00000030h
00411397  B8CCCCCCCC                                          mov          eax,CCCCCCCCh
0041139C  F3AB                                                rep stosd
0041139E  8B4508                                              mov          eax,[ebp+08h]
004113A1  030500704100                                        add          eax,[L00417000]
004113A7  5F                                                  pop          edi
004113A8  5E                                                  pop          esi
004113A9  5B                                                  pop          ebx
004113AA  8BE5                                                mov          esp,ebp
004113AC  5D                                                  pop          ebp
004113AD  C3                                                  retn
;----------------------------------------------------------------------------------------
004113AE  CCCCCCCCCCCCCCCCCCCC+                               Align        32
004113C0                                    L004113C0:
004113C0  55                                                  push         ebp
004113C1  8BEC                                                mov          ebp,esp
004113C3  81ECCC000000                                        sub          esp,000000CCh
004113C9  53                                                  push         ebx
004113CA  56                                                  push         esi
004113CB  57                                                  push         edi
004113CC  8DBD34FFFFFF                                        lea          edi,[ebp-000000CCh]
004113D2  B933000000                                          mov          ecx,00000033h
004113D7  B8CCCCCCCC                                          mov          eax,CCCCCCCCh
004113DC  F3AB                                                rep stosd
004113DE  6A02                                                push         00000002h
```

Multistep Processing of a User Program



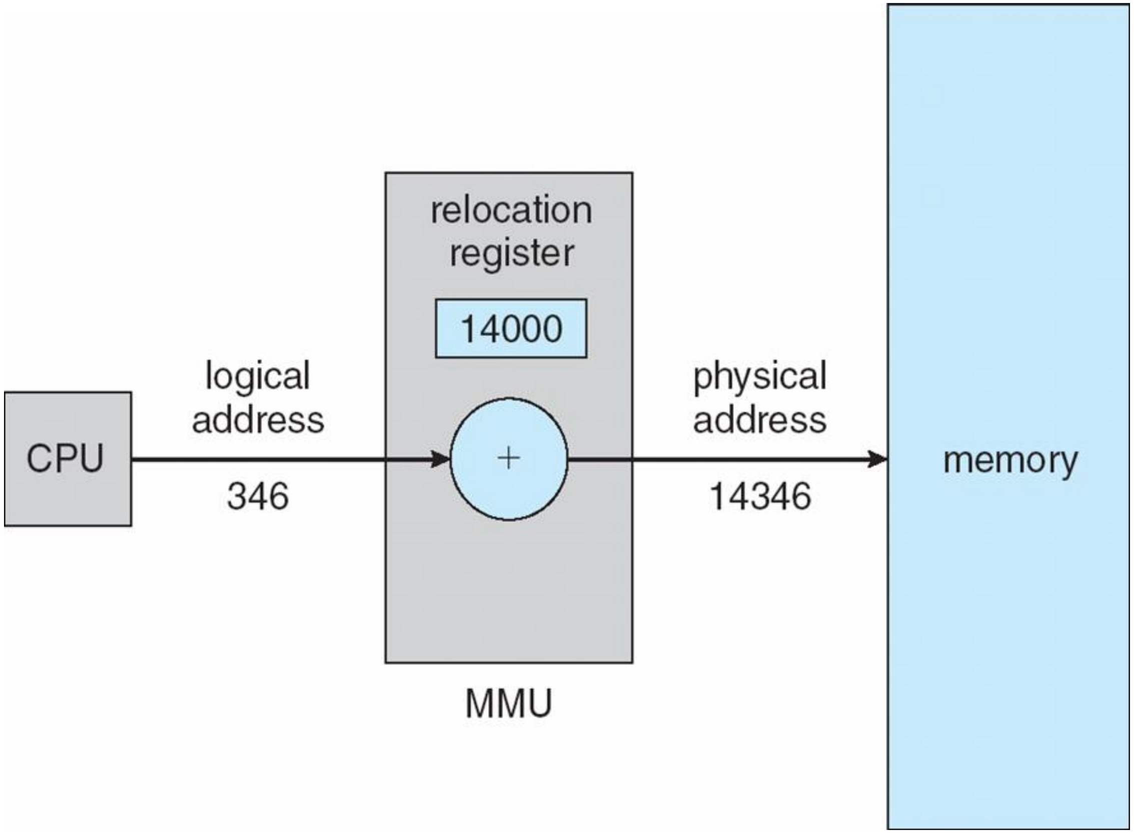
Logical vs. Physical Address Space

- 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

Memory-Management Unit (MMU)

- Hardware device that maps 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
- System also known as **shared libraries**

Dynamic Linking

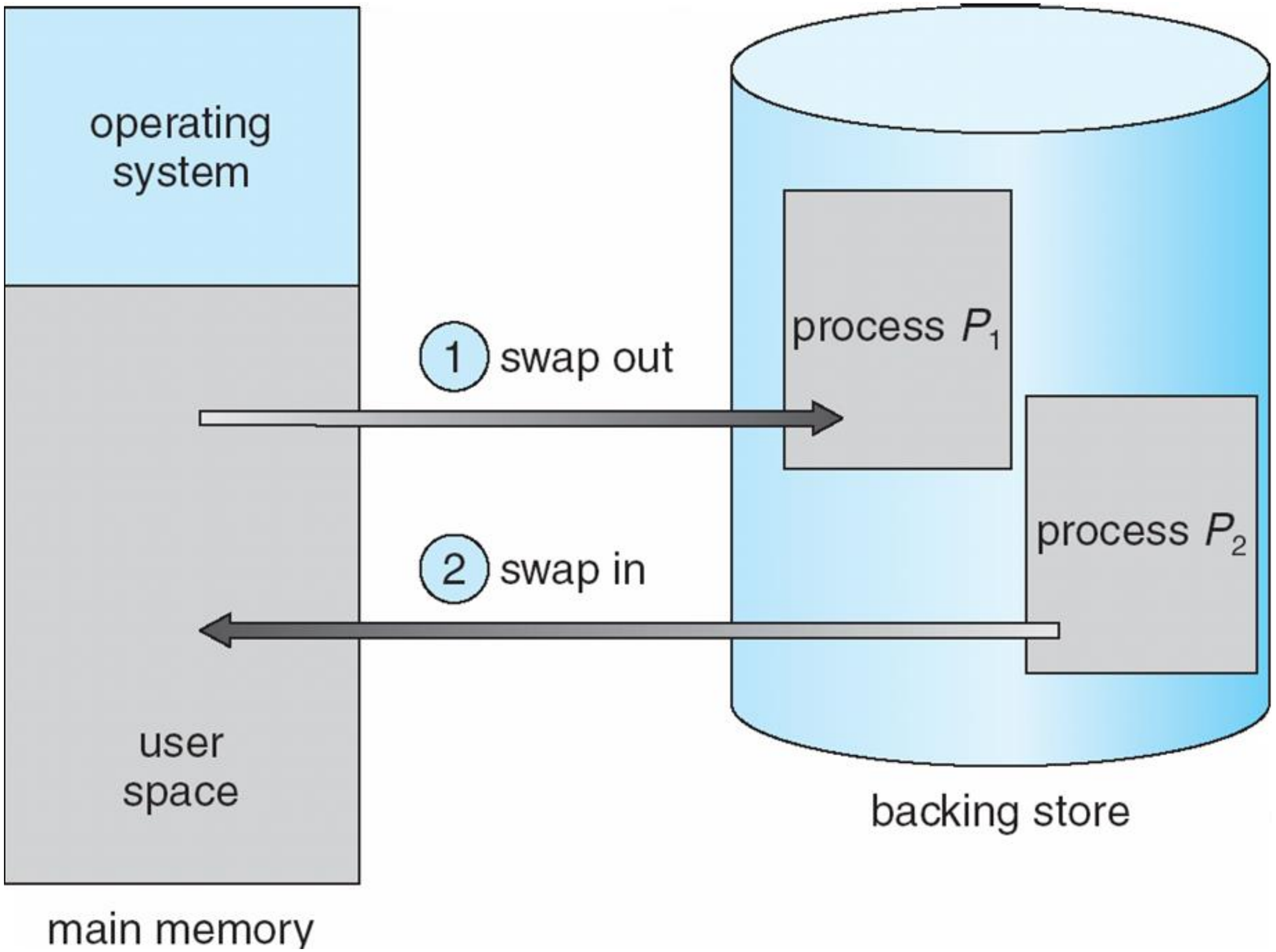
- http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/
- http://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html
- http://www.acsu.buffalo.edu/~charngda/elf.html

- Sample code for creating dynamic linking / loading library
  - http://sense.cs.nctu.edu.tw/courses/code/library_test.tgz

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

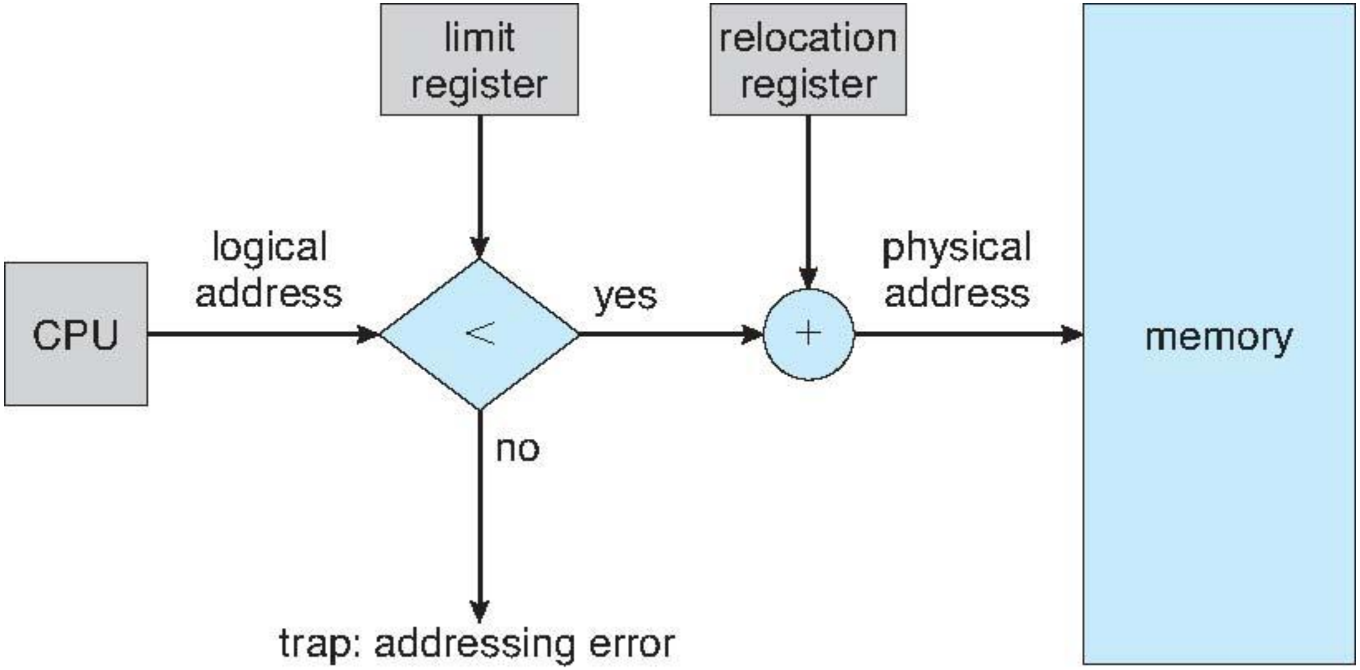
Schematic View of Swapping



Contiguous Allocation

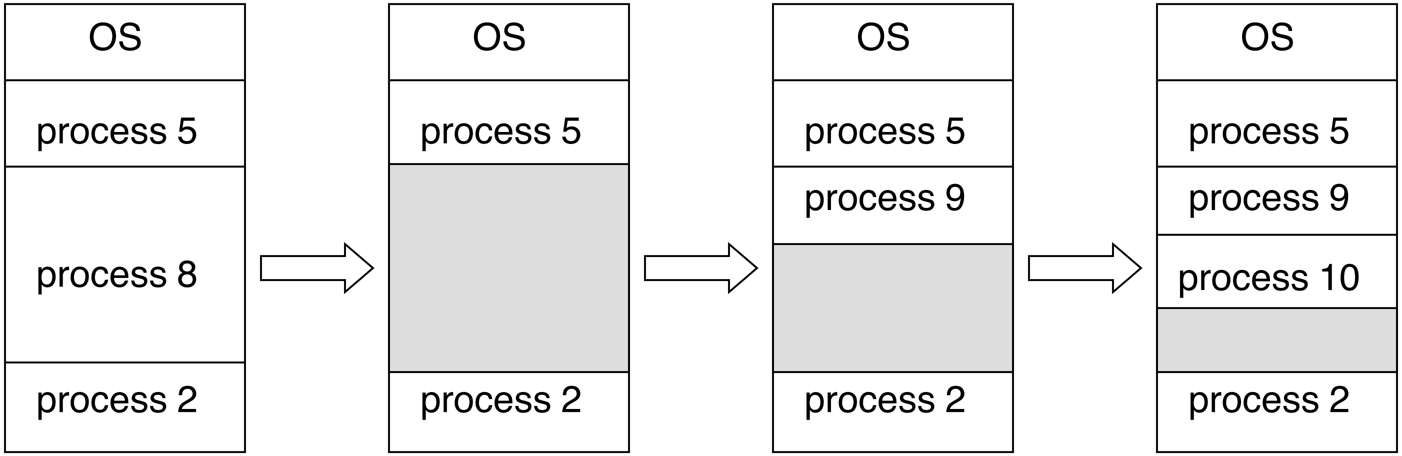
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register 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 address *dynamically*

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)



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

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
- 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
- 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

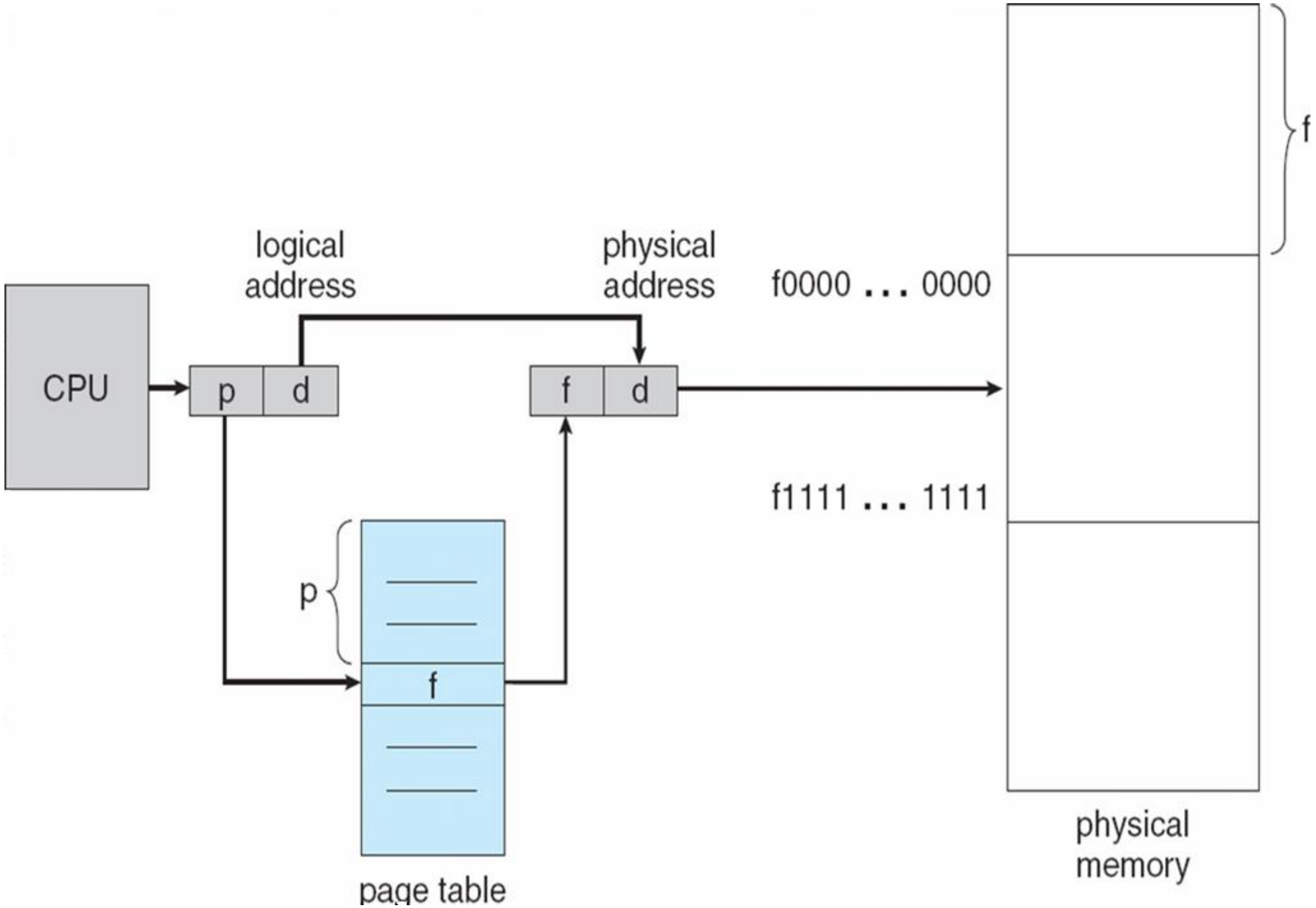
Address Translation Scheme

- 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

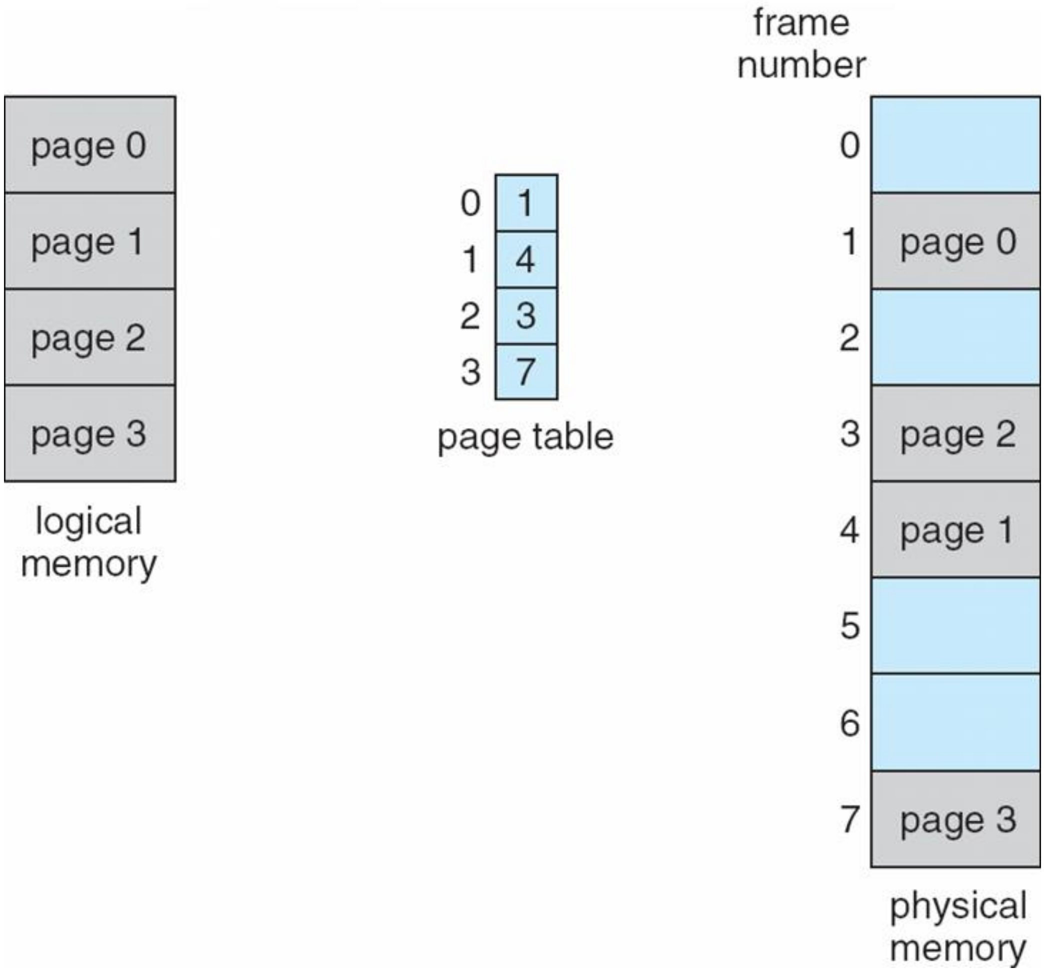
page number	page offset
p	*d*
m - n	*n*

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

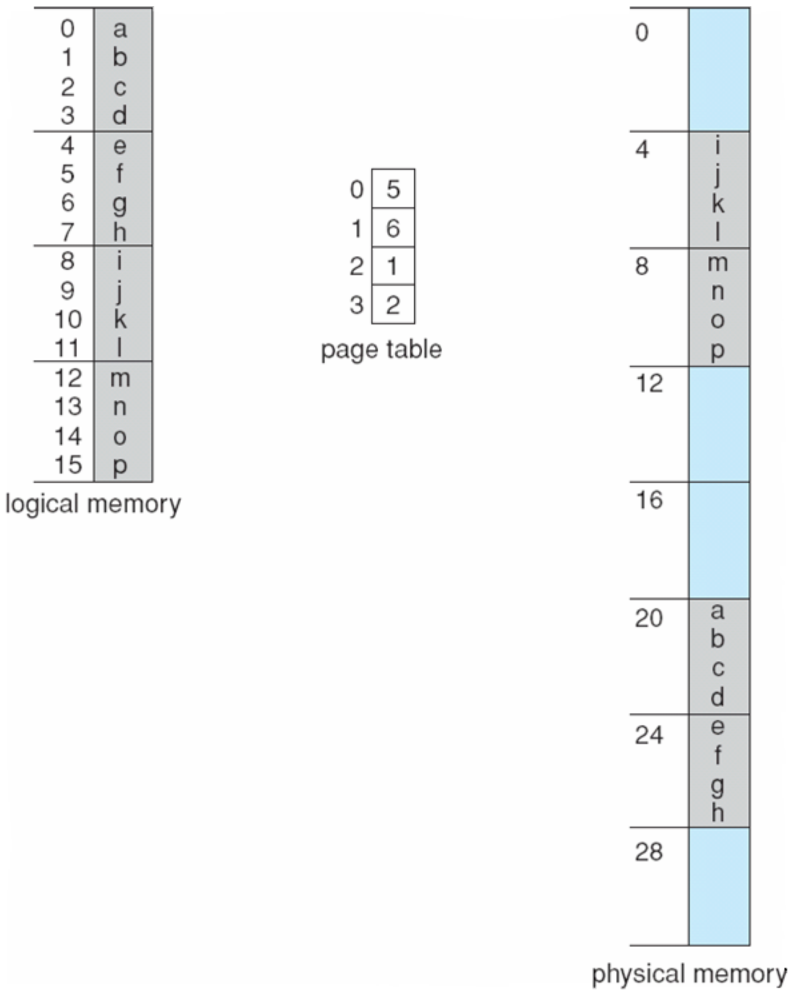
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



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

0	5
1	6
2	1
3	2

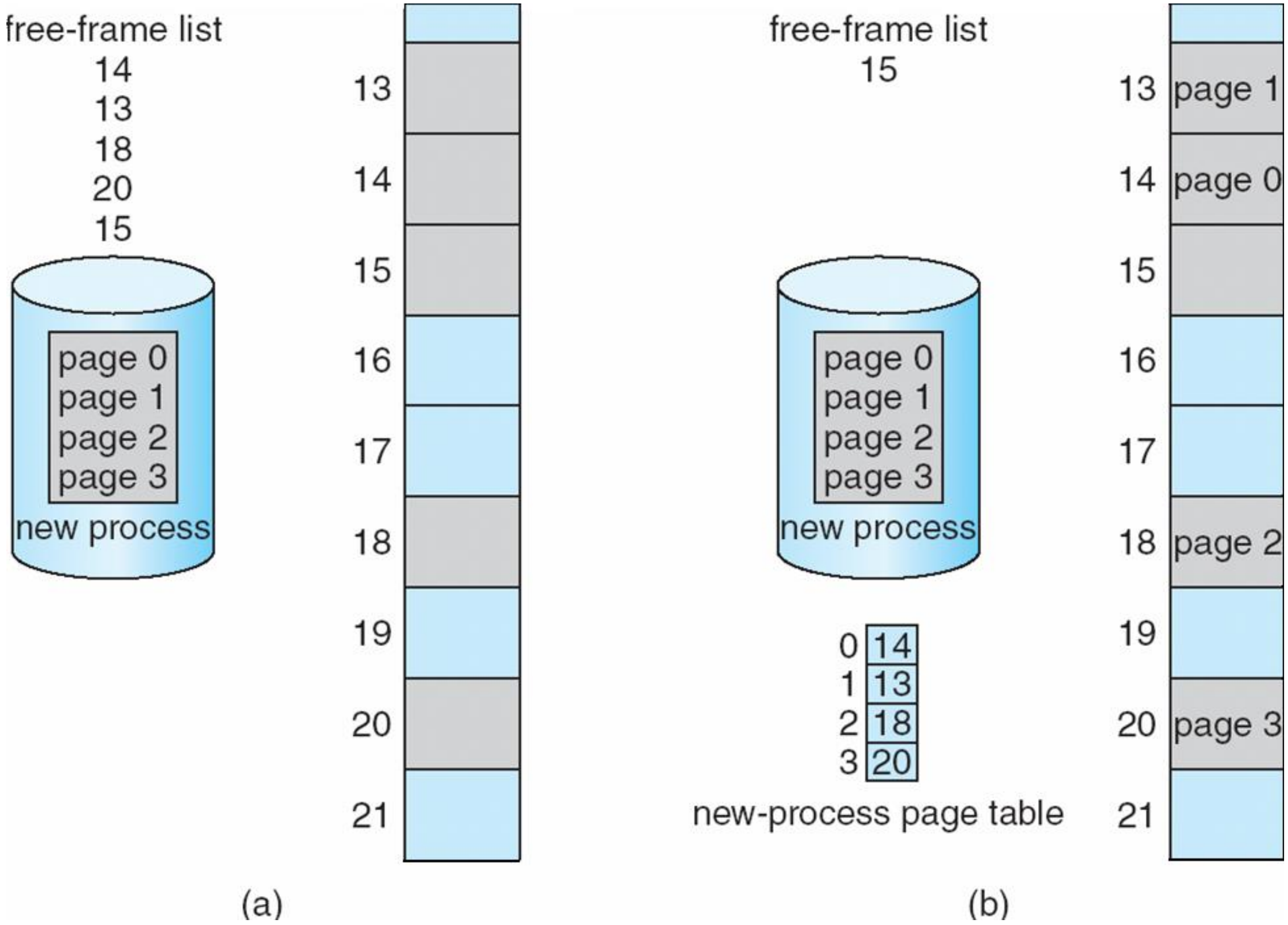
page table

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

32-byte memory and 4-byte pages

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 (PRLR)** 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

Associative Memory

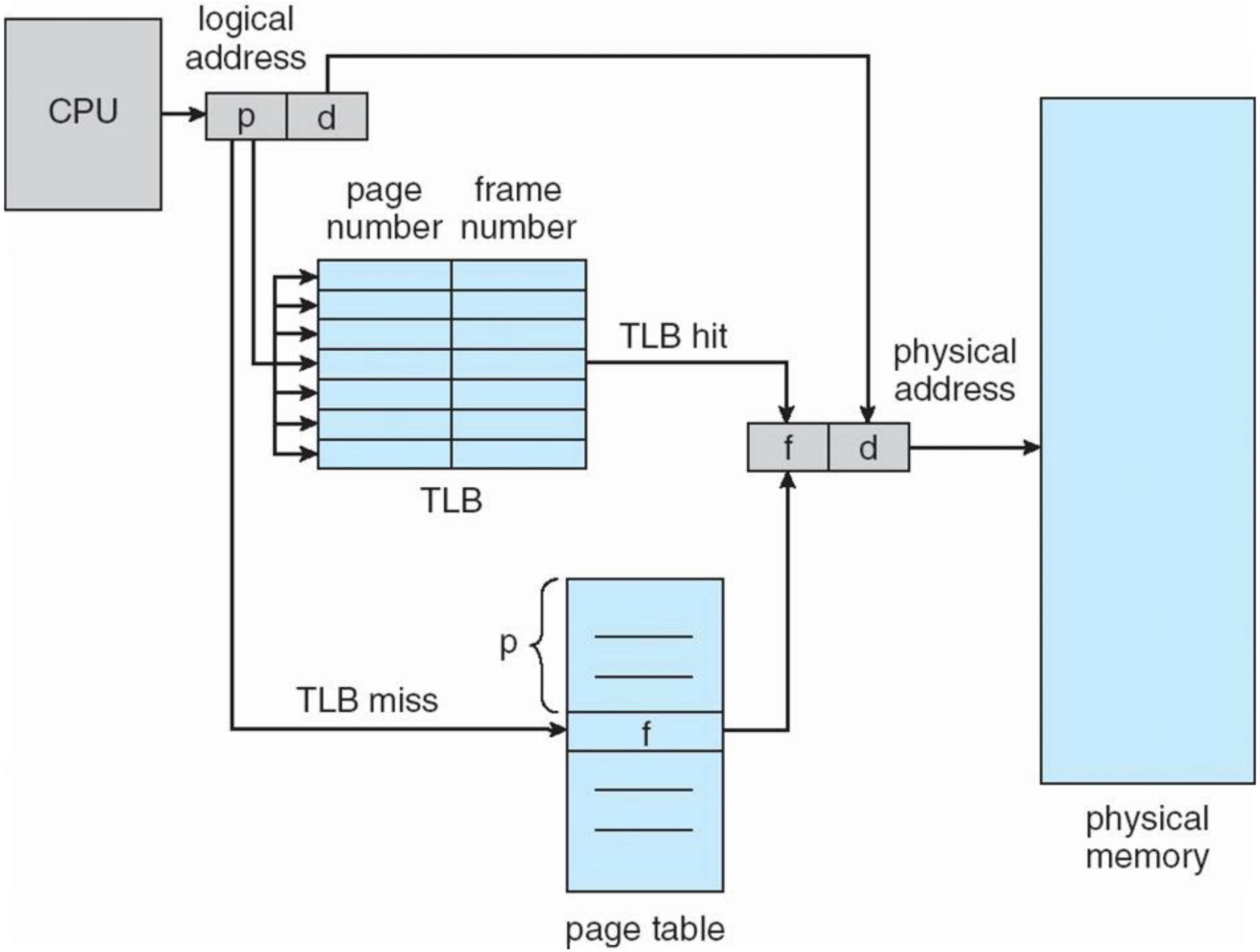
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

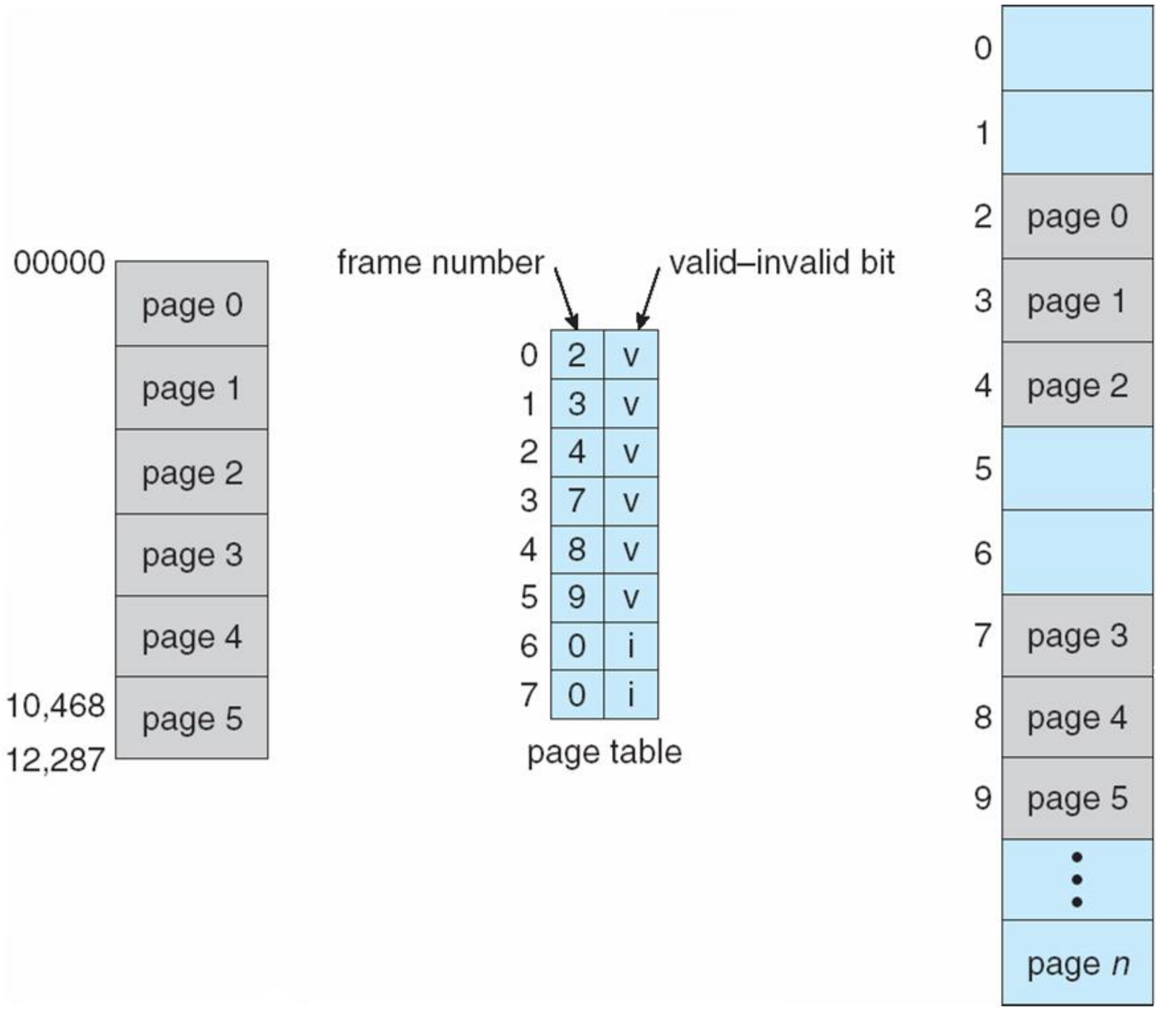
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

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



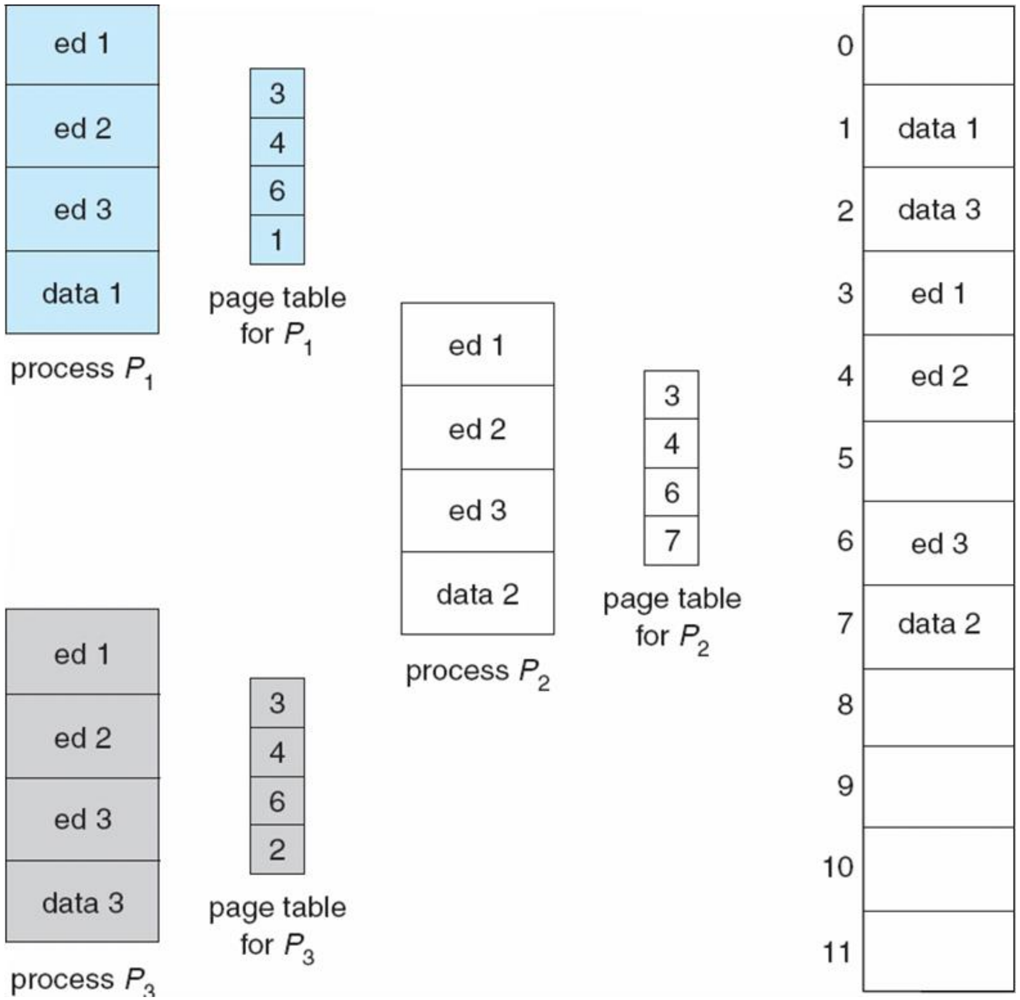
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

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



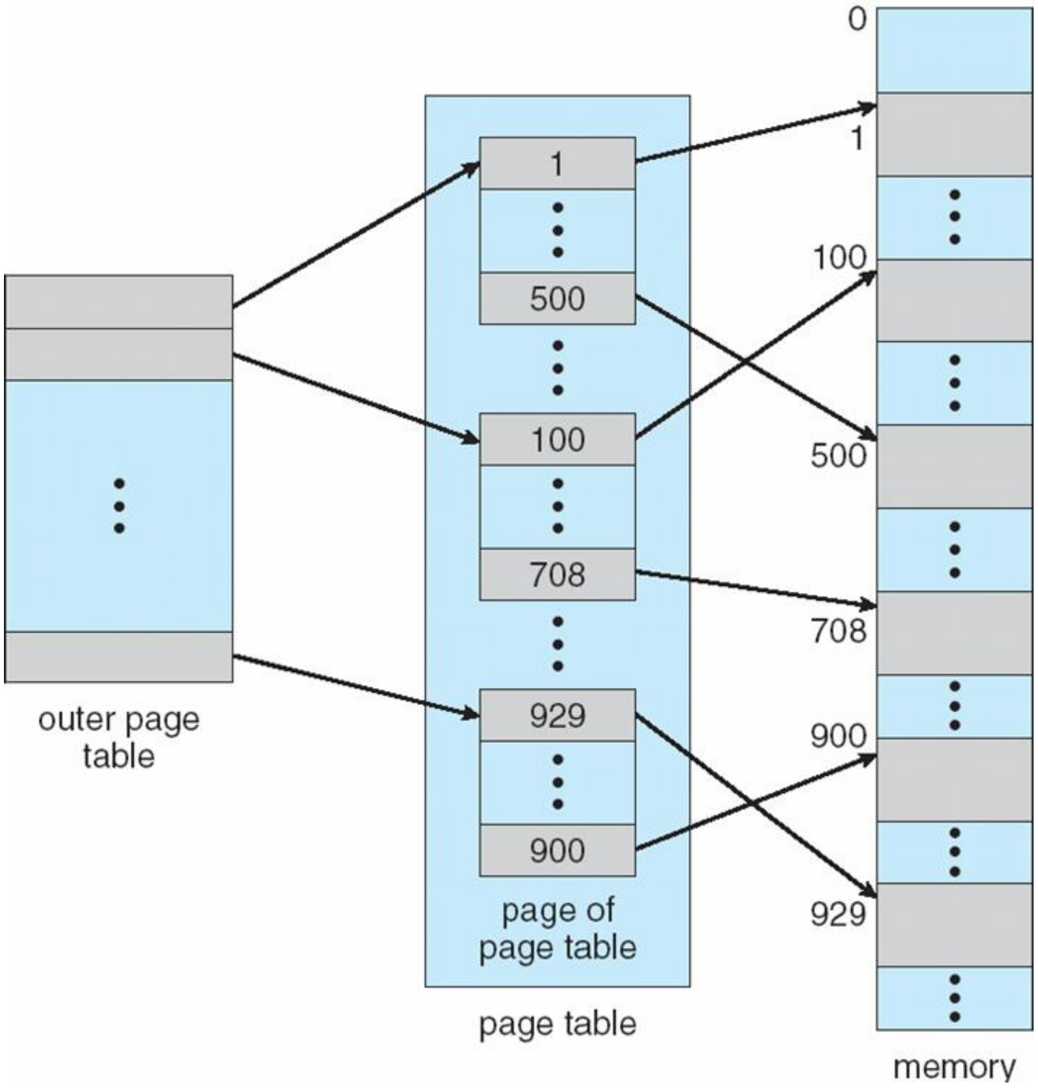
Structure of the Page Table

- 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 Page-Table Scheme



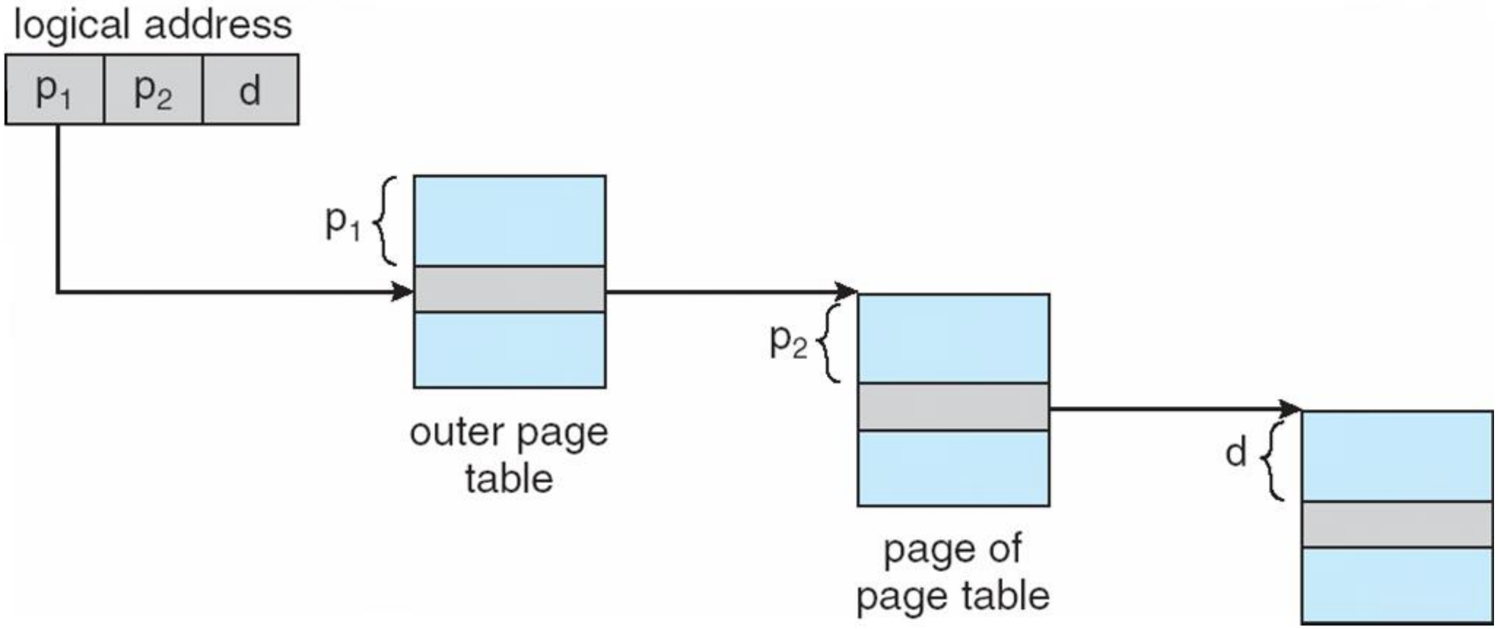
Two-Level Paging Example

- 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:

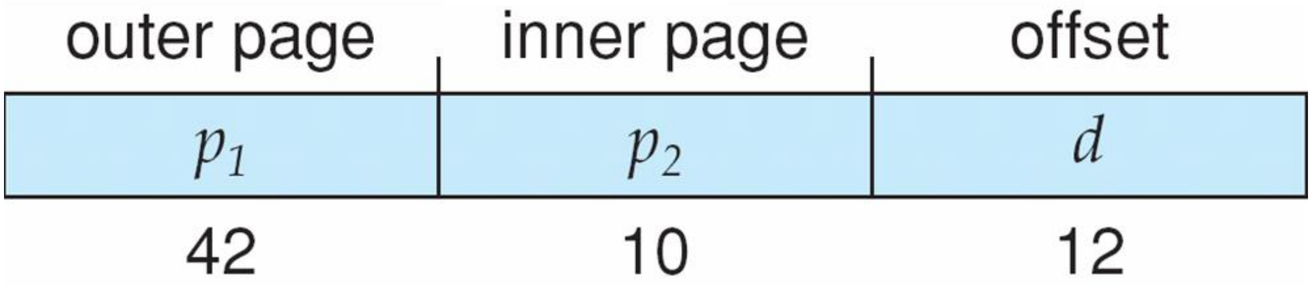
page number		page offset
p_i	p_2	d
12	10	10

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

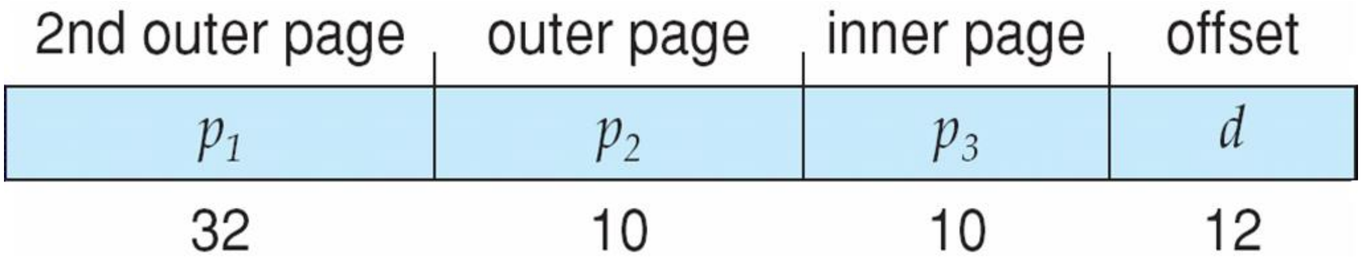
Address-Translation Scheme



Three-level Paging Scheme



outer page	inner page	offset
p_1	p_2	d
42	10	12

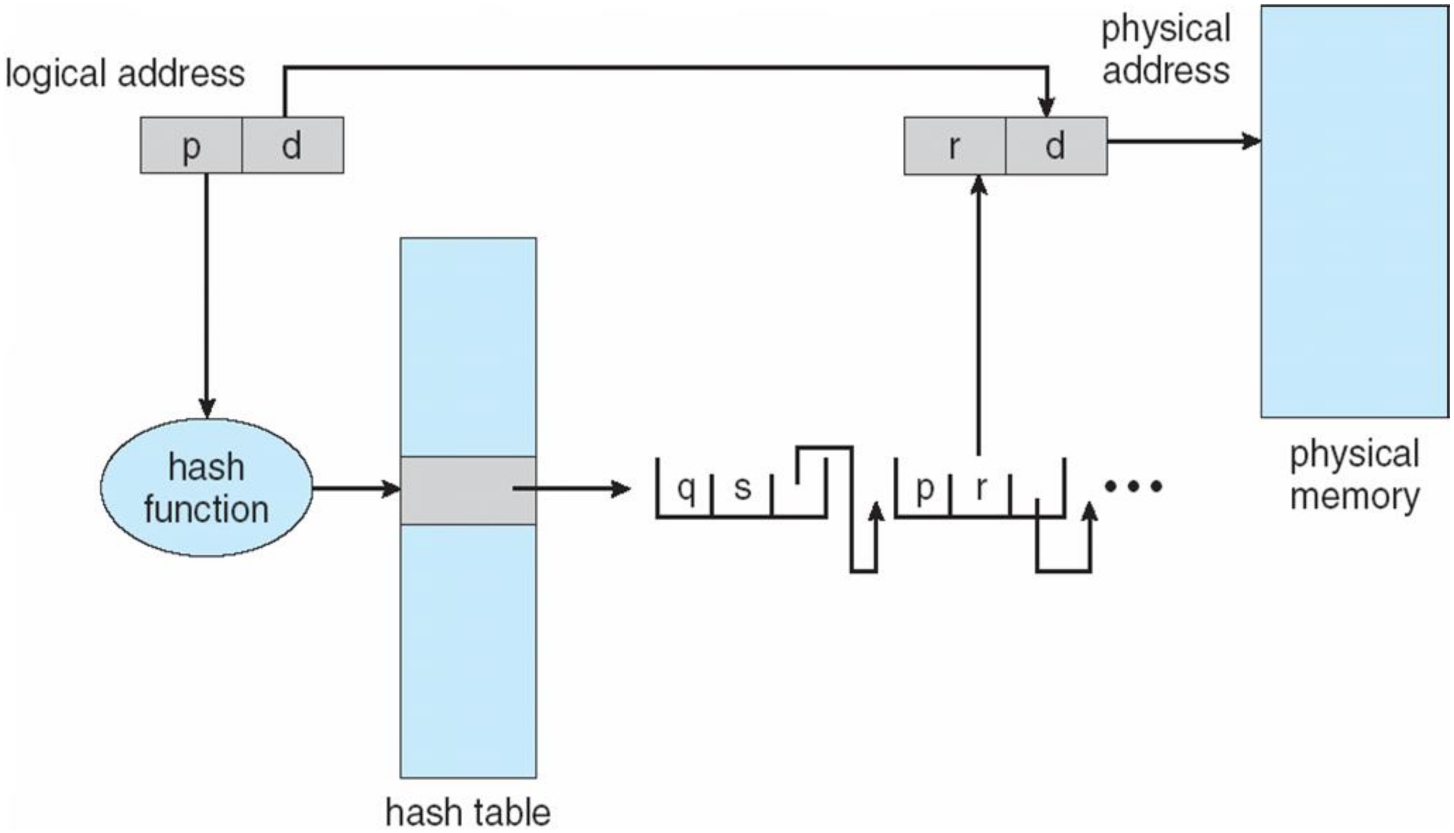


2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

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

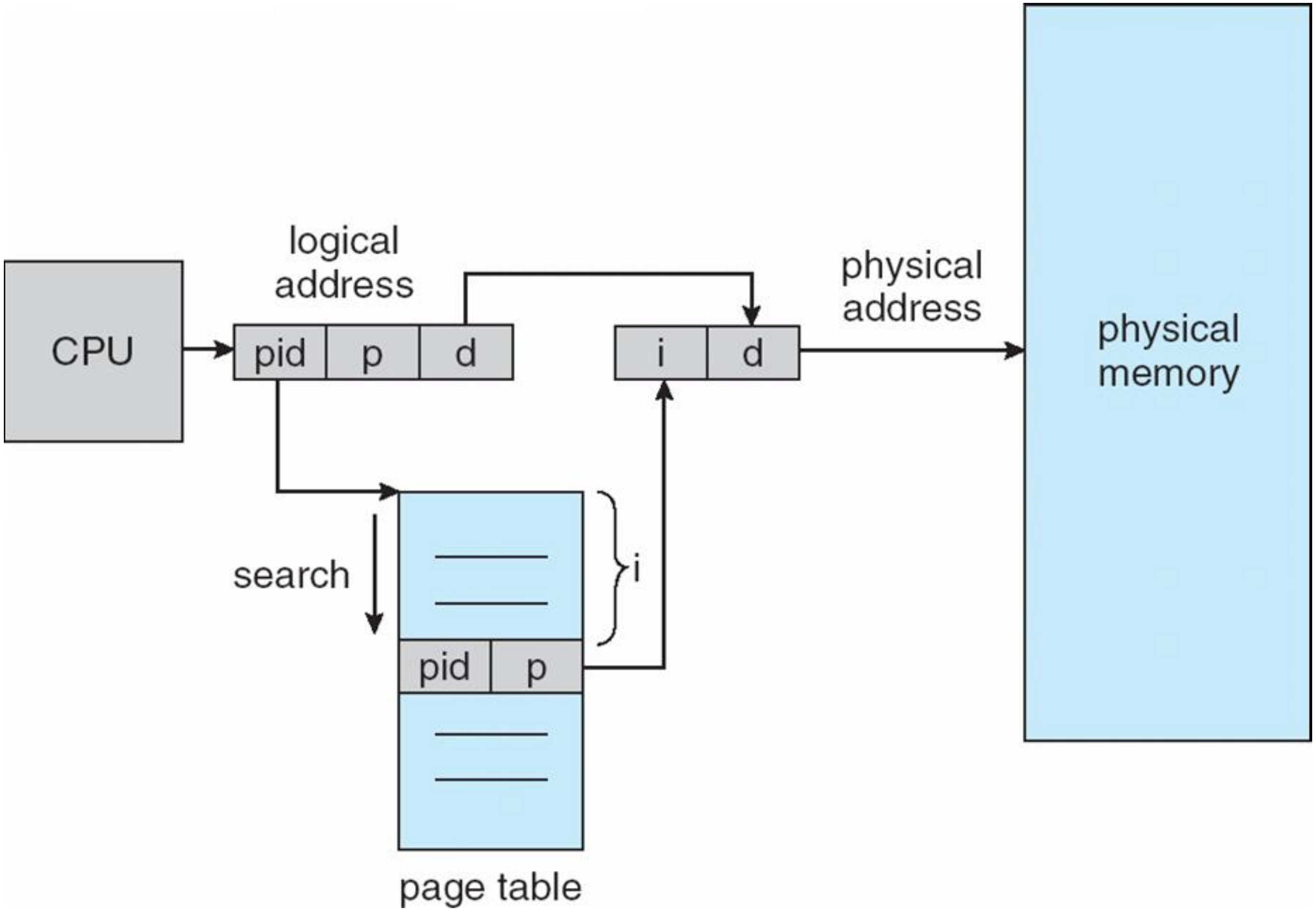
Hashed Page Table



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



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

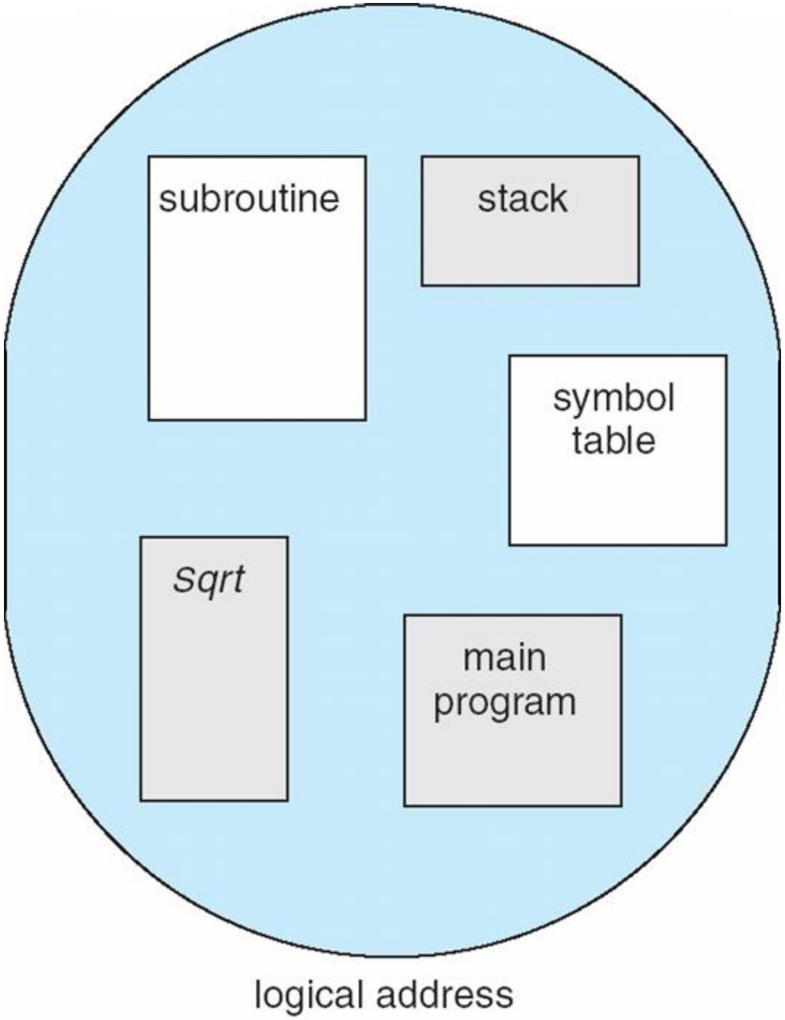
    common block

    stack

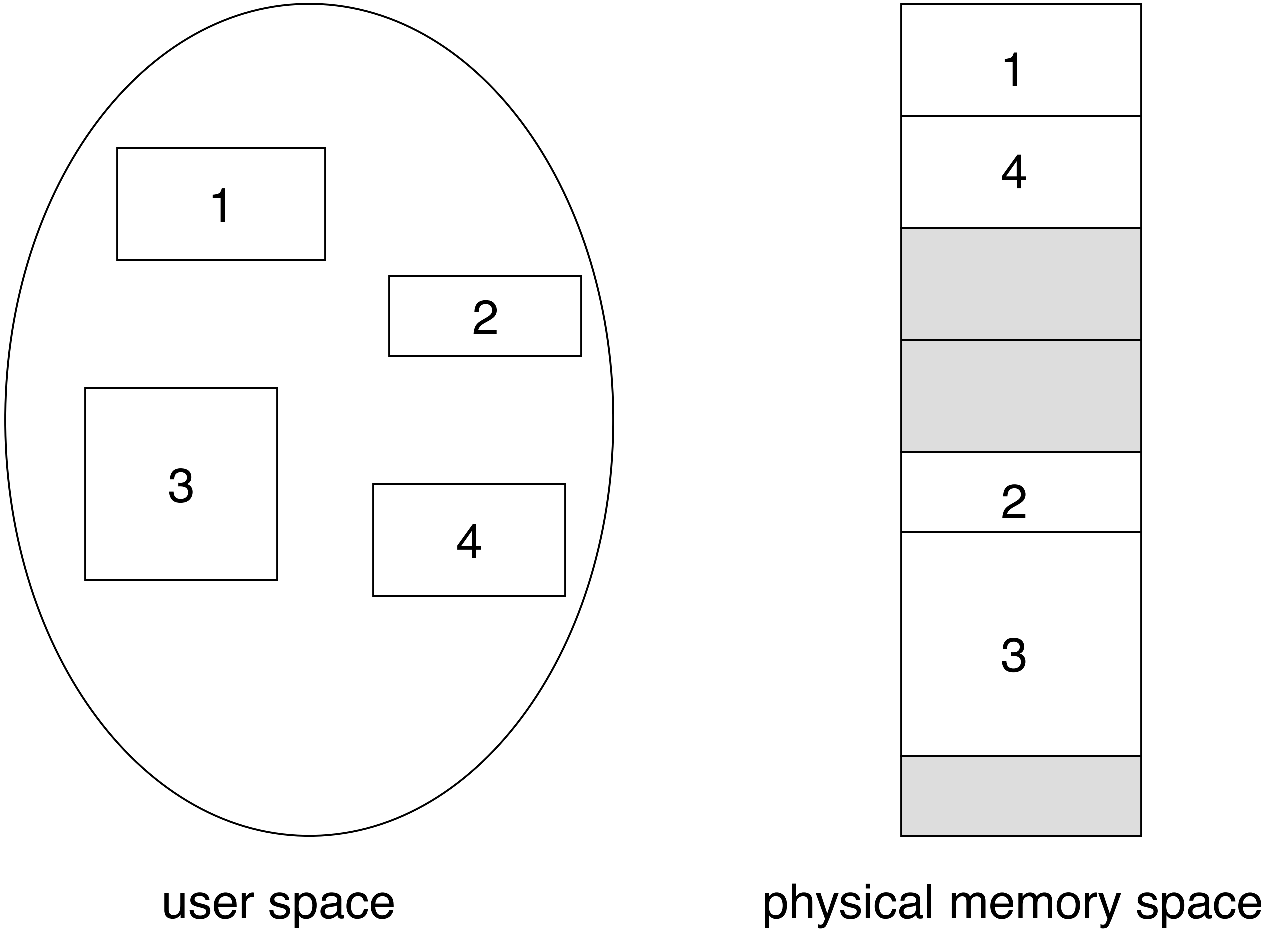
    symbol table

    arrays

User’s View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

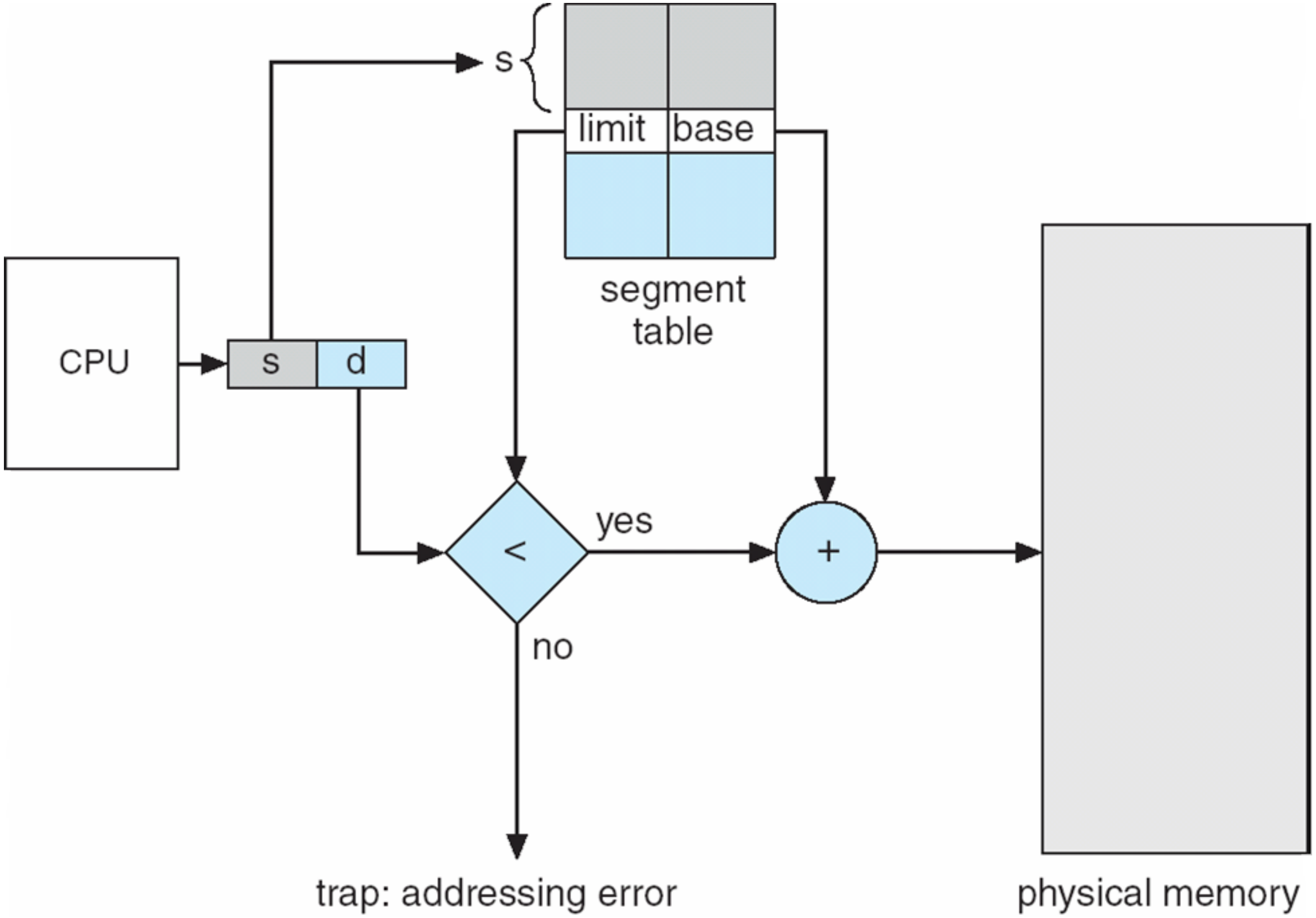
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

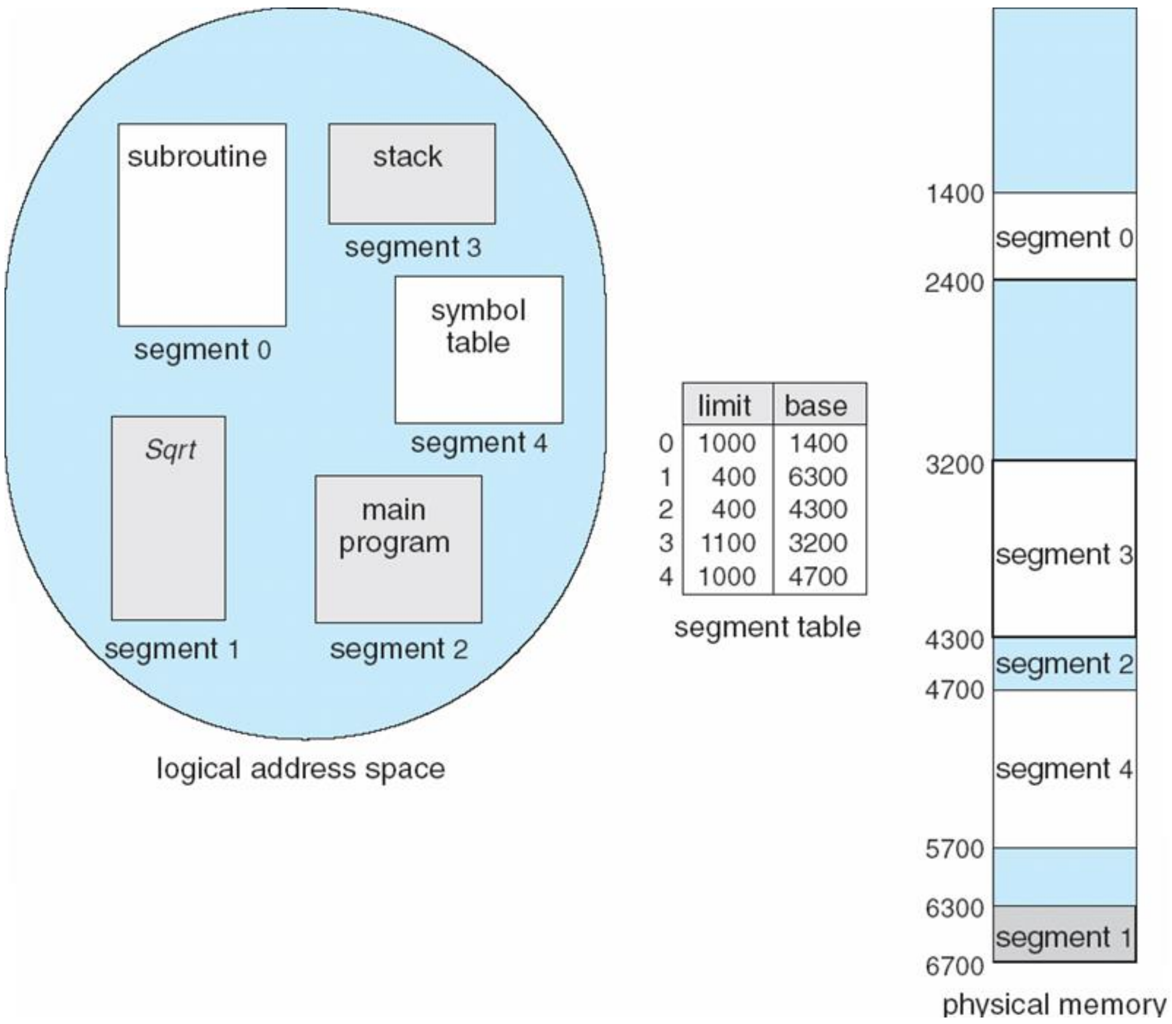
Segmentation Architecture (Cont.)

- 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
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



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

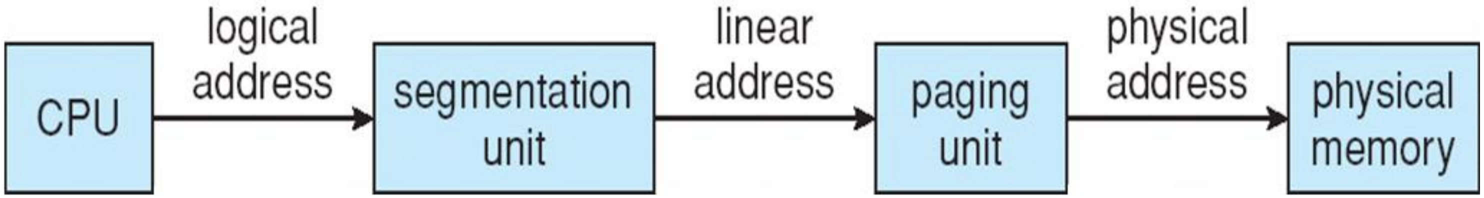
segment table

Example: The Intel Pentium

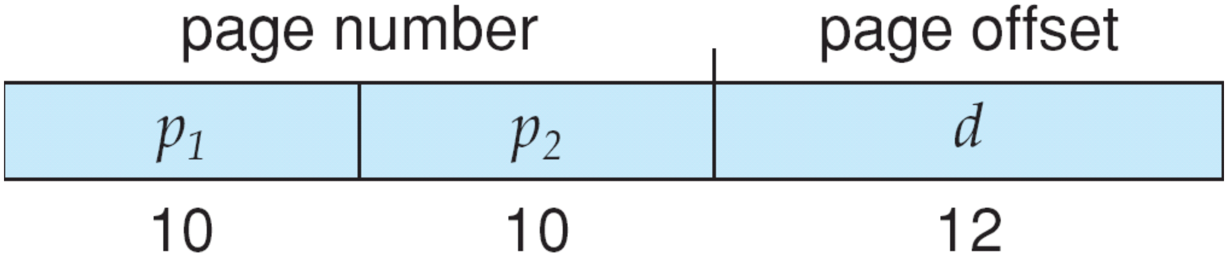
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Logical to Physical Address Translation in Pentium

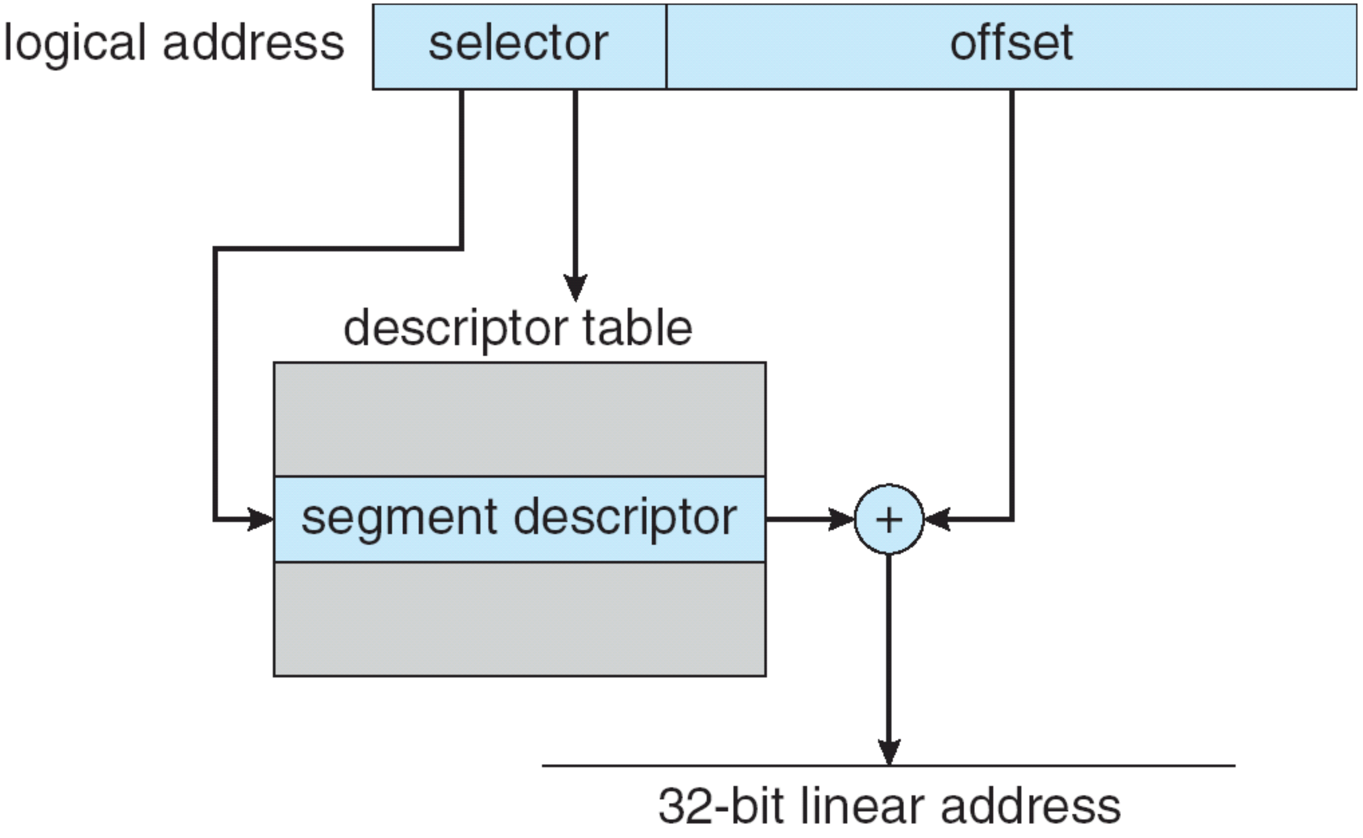


CPU → logical address → segmentation unit → linear address → paging unit → physical address → physical memory

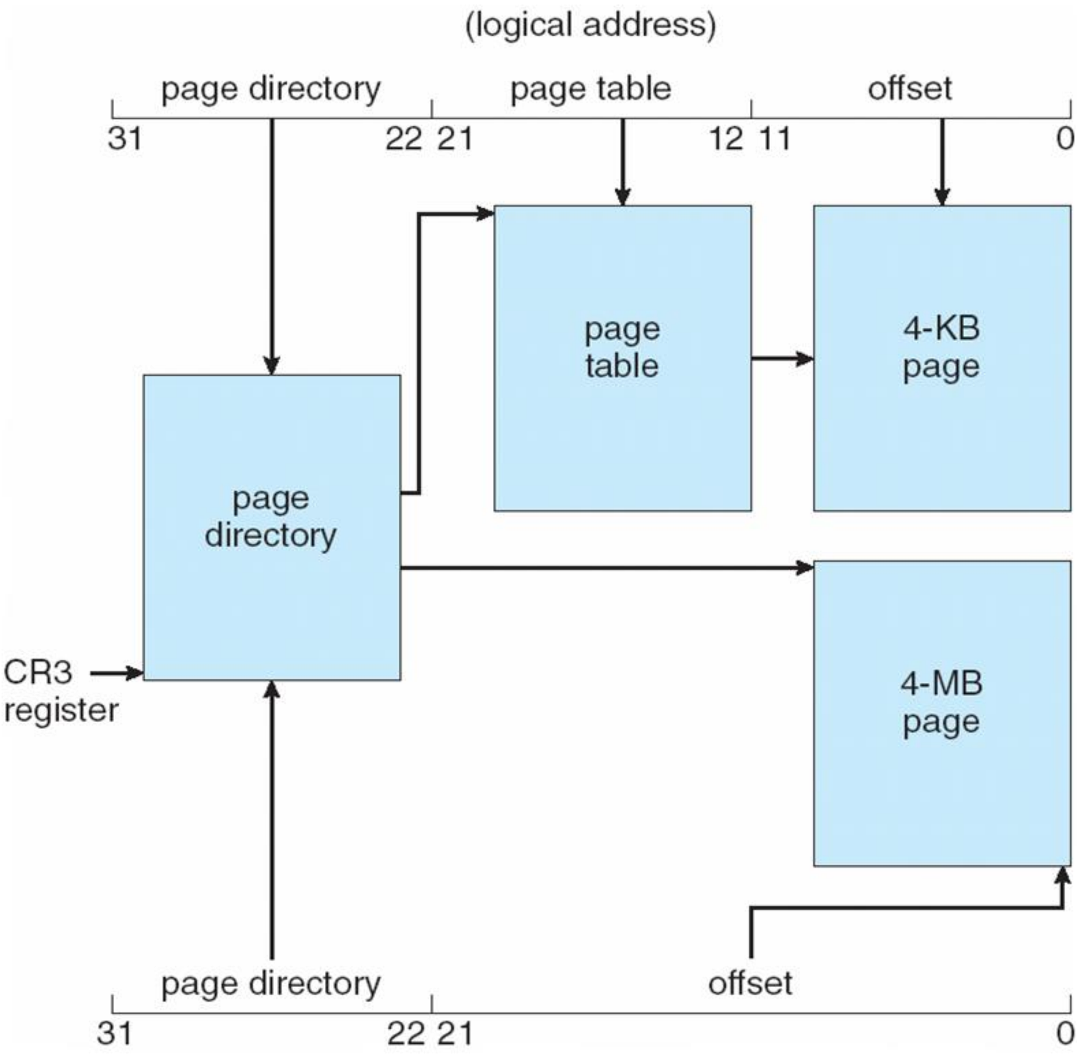


page number		page offset
p_1	p_2	d
10	10	12

Intel Pentium Segmentation

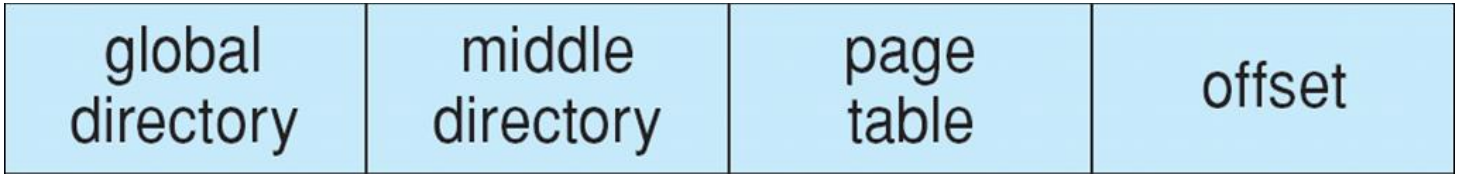


Pentium Paging Architecture



Linear Address in Linux

Broken into four parts:



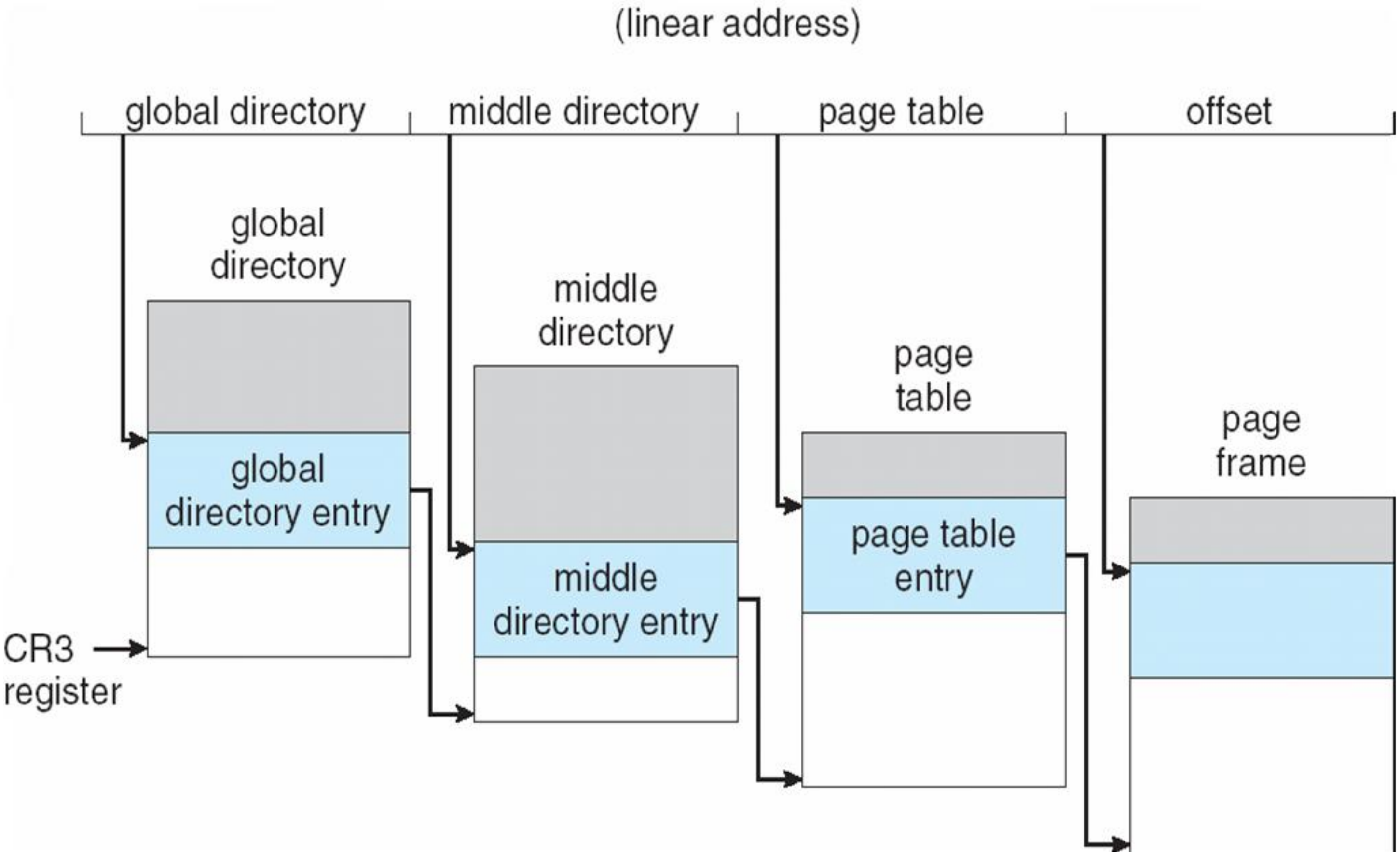
global directory

middle directory

page table

offset

Three-level Paging in Linux



AT&T assembly syntax:

- source comes before destination
- mnemonic suffixes indicate the size of the operands (`q` for quad, etc.)
- registers are prefixed with `%` and immediate values with `$`
- effective addresses are in the form `DISP(BASE, INDEX, SCALE)` (DISP + BASE + INDEX * SCALE)
- absolute jump/call operands indicated with `*` (as opposed to IP relative)

End of Chapter 8