Chapter 8: Main Memory

Chapter 8: Memory Management

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

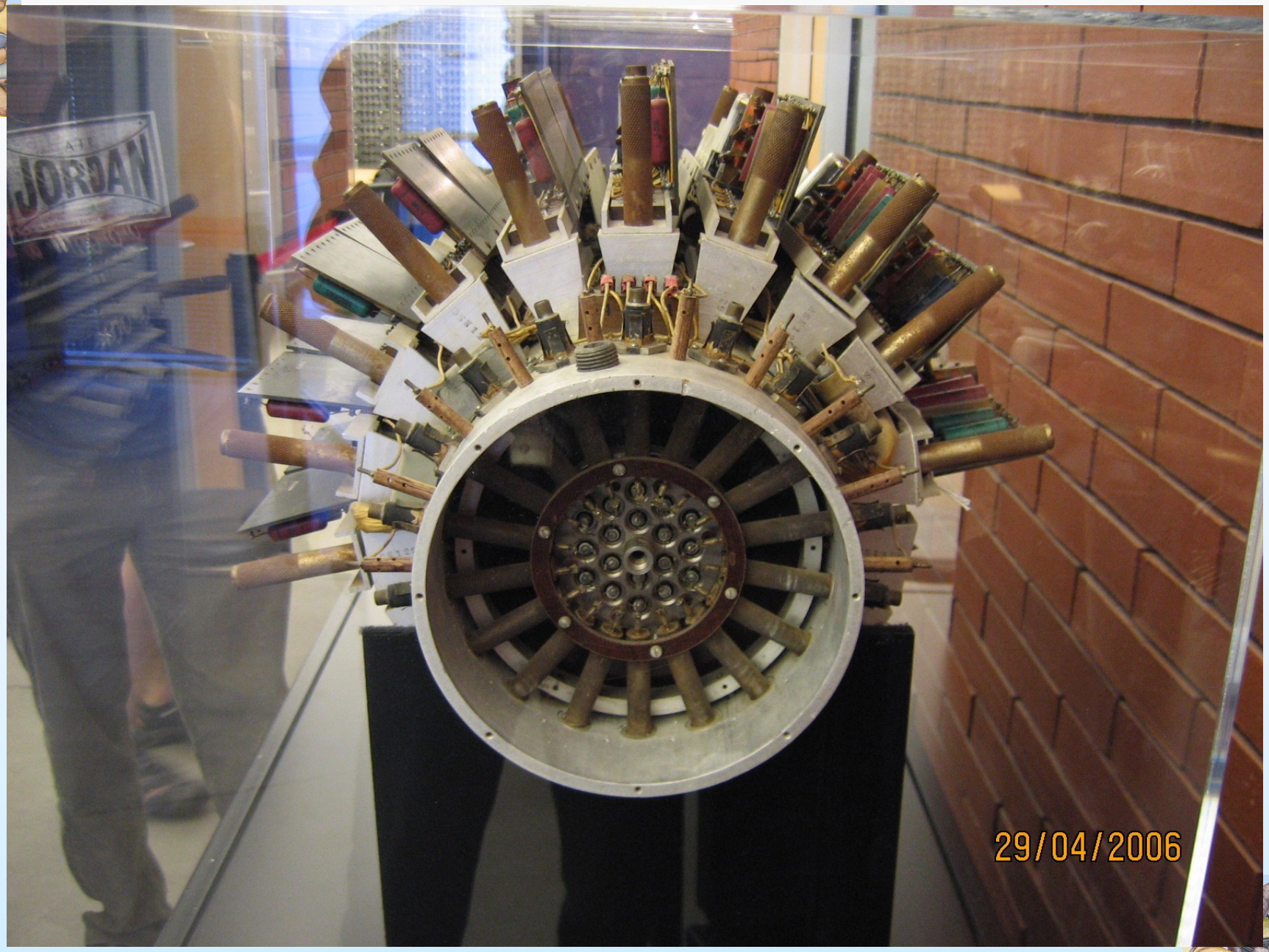
Objectives

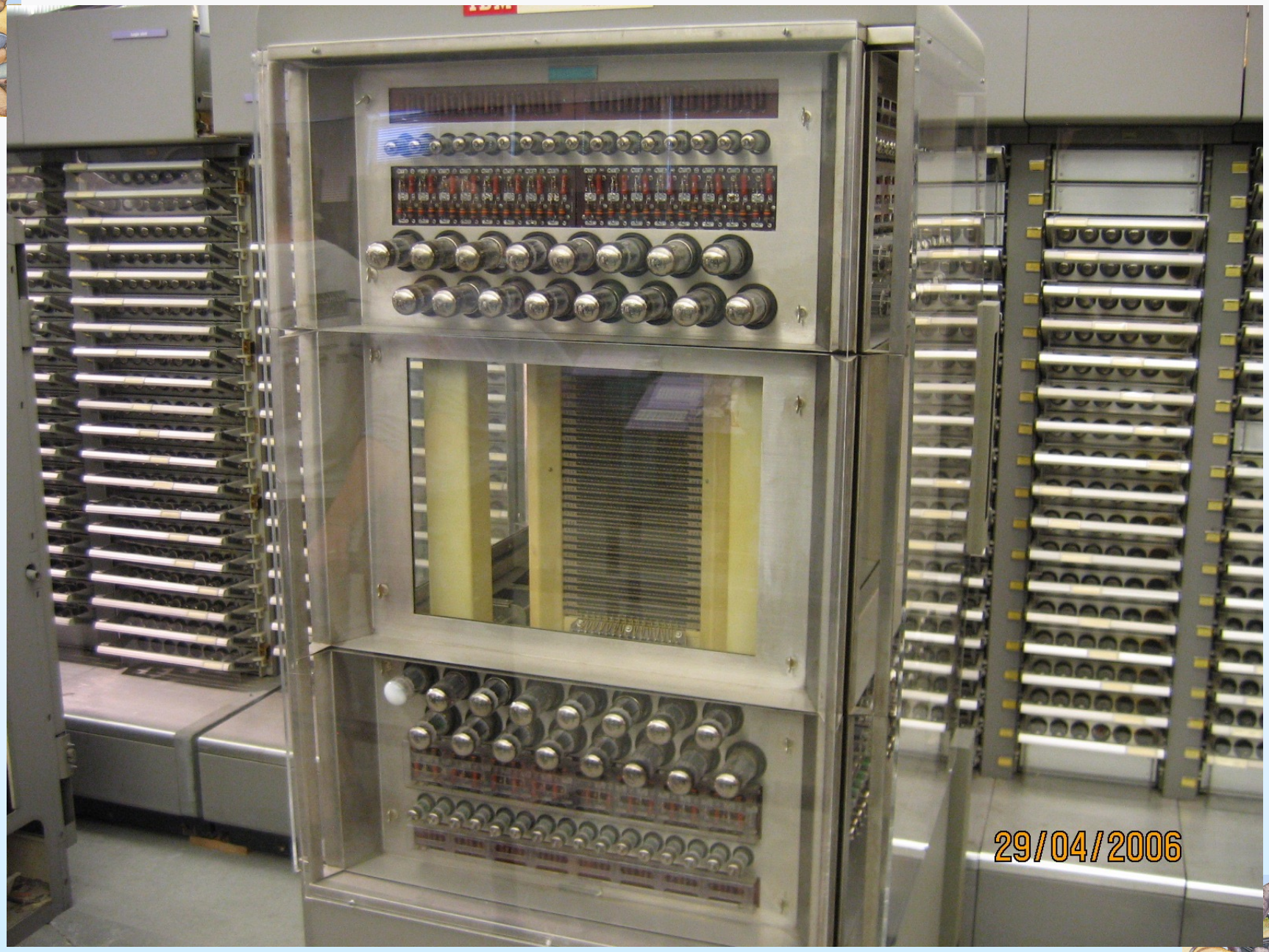
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

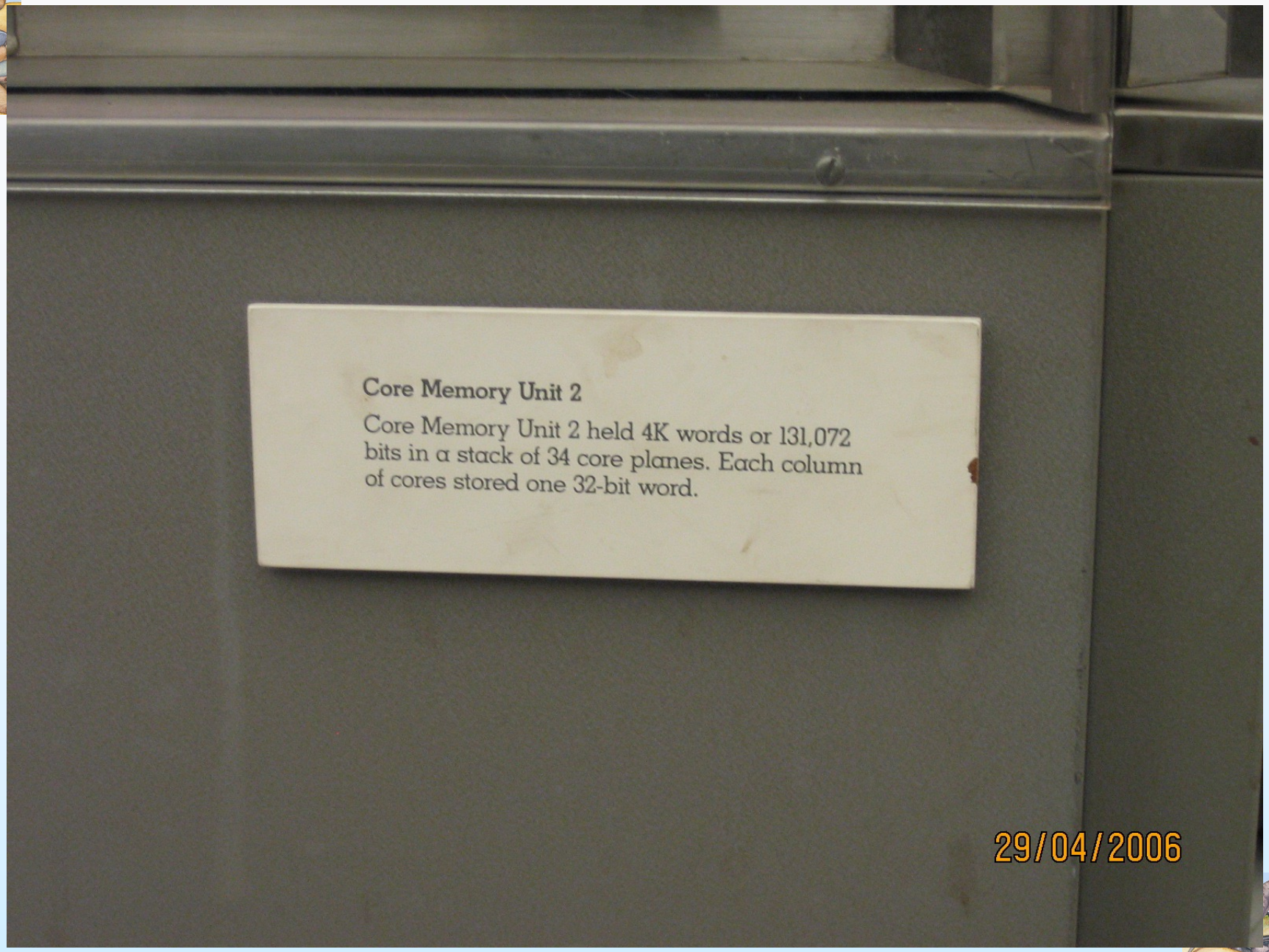
Background

- 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


29/04/2006


JORDAN
29/04/2006

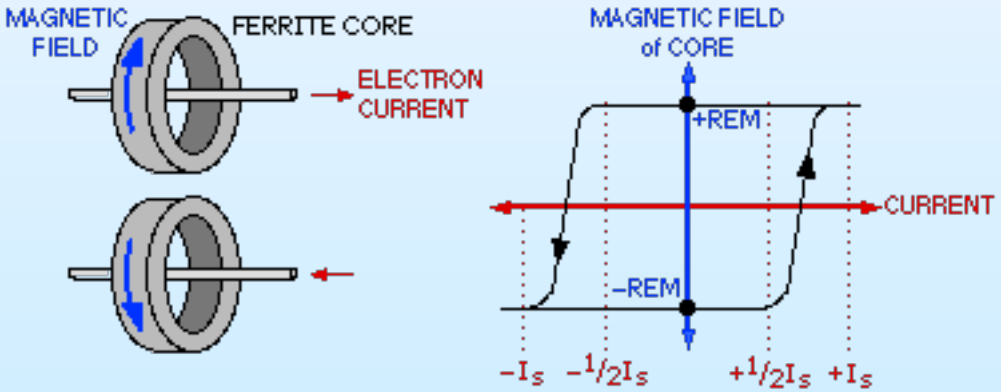

29/04/2006


Core Memory Unit 2
Core Memory Unit 2 held 4K words or 131,072 bits in a stack of 34 core planes. Each column of cores stored one 32-bit word.
29/04/2006

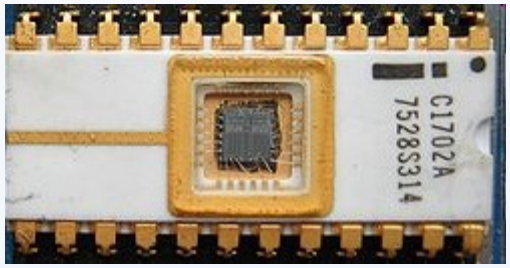
Patent Sold to IBM in 1956

An Wang sells his patent for ferrite core memory to IBM for $500,000. One of the most important inventions in computer history, ferrite core memory was widely used in digital computers from the mid-1950s until the mid-1970s. The U.S. Patent Office awarded Wang the patent for what he called a pulse transfer controlling device in 1949. Jay Forrester at MIT is considered the inventor of core memory.

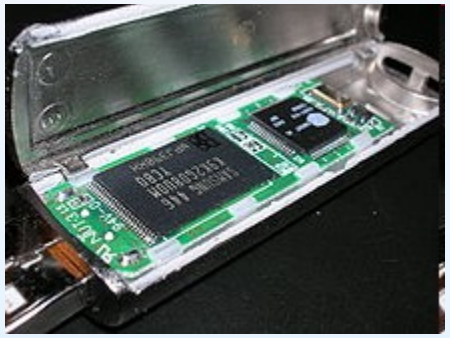
Figure 2a: Magnetising a ferrite core



Semiconductor ROM/RAM

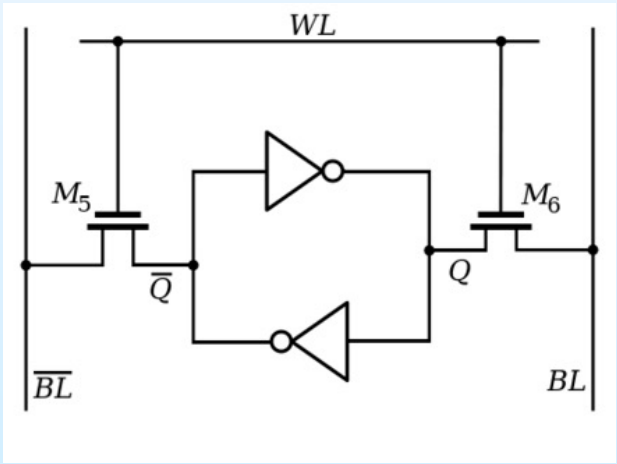


EPROM chip



Internal of a USB drive

DRAM chips



Future: Phase-change memory?

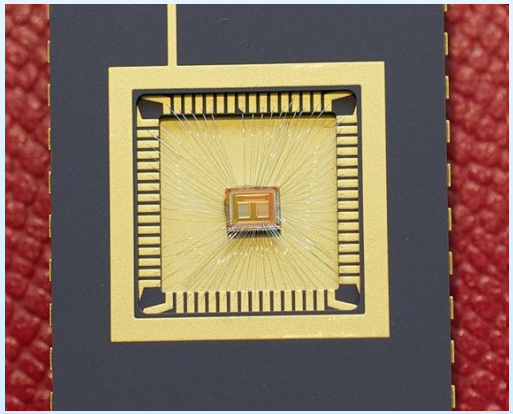
- In April 2010, Numonyx (Micron) announced the Omneo line of 128-Mbit NOR-compatible phase-change memories and Samsung: 512 Mb phase-change RAM (PRAM) in a multi-chip package (MCP) for mobile handsets in 2010.
- In June 2011, IBM announced that they had created stable, reliable, multi-bit phase change memory with high performance and stability.

IBM develops 'instantaneous' memory, 100x faster than flash

By Sharif Sakr posted Jun 30th 2011 12:01AM

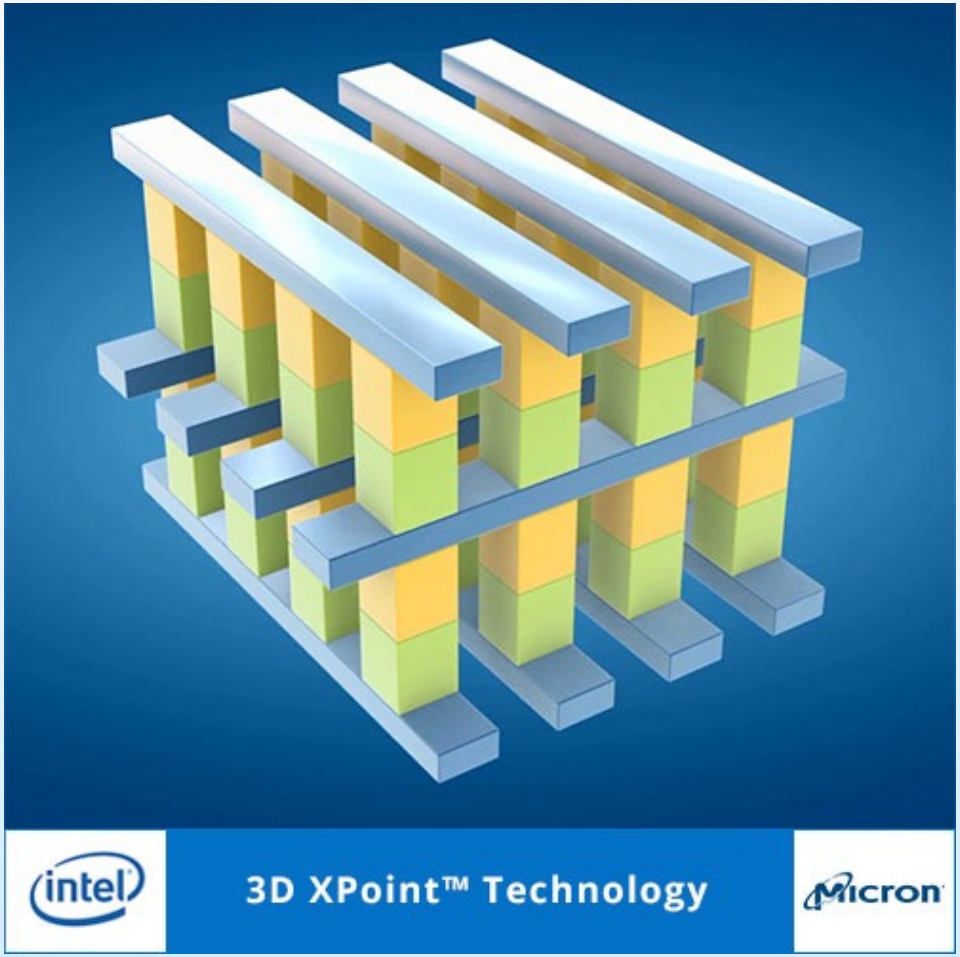
engadget

PR

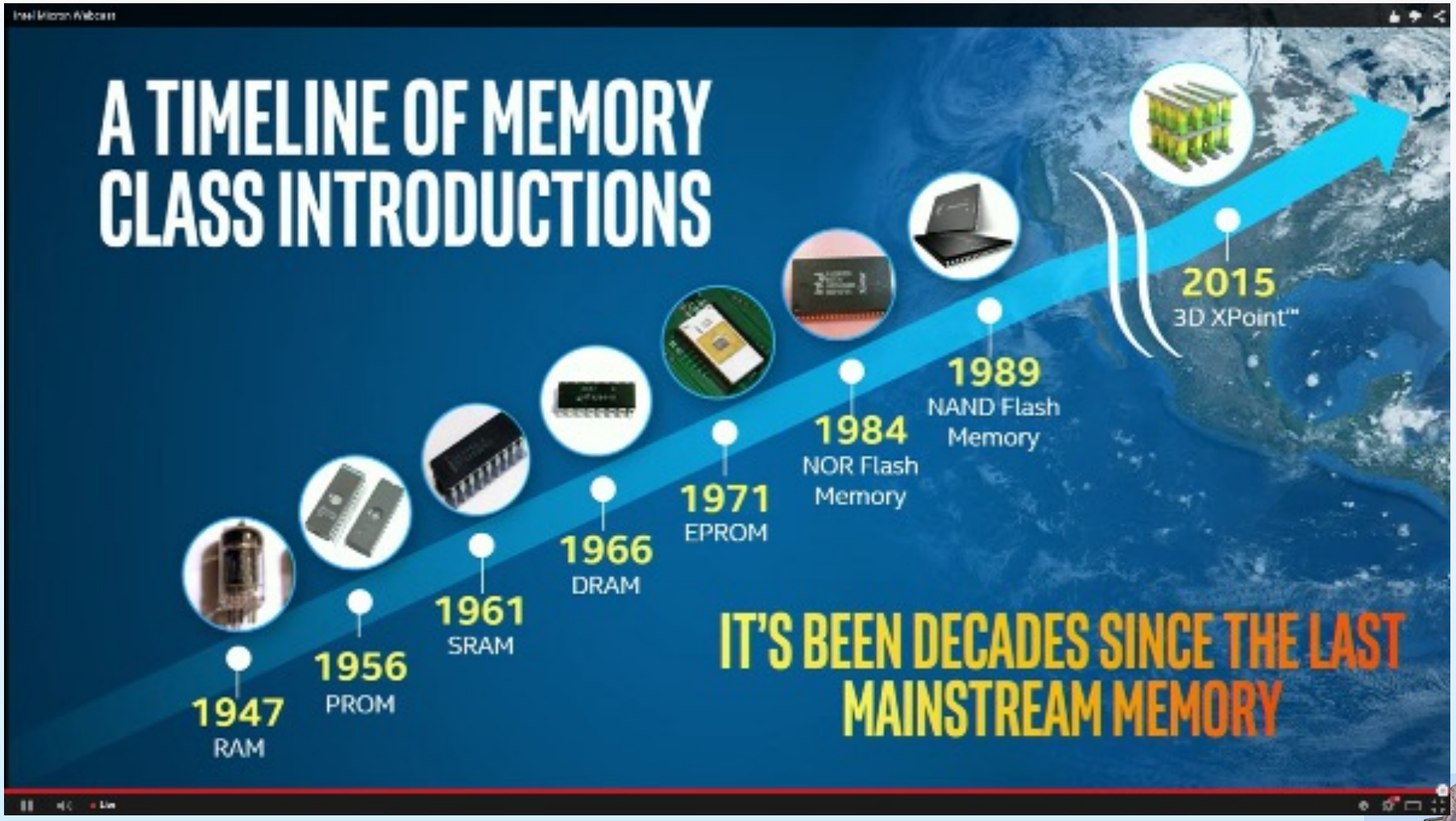


Future: Intel's 3D XPoint Memory

non-volatile memory speeds up to 1,000 times faster1 than NAND



Memory History

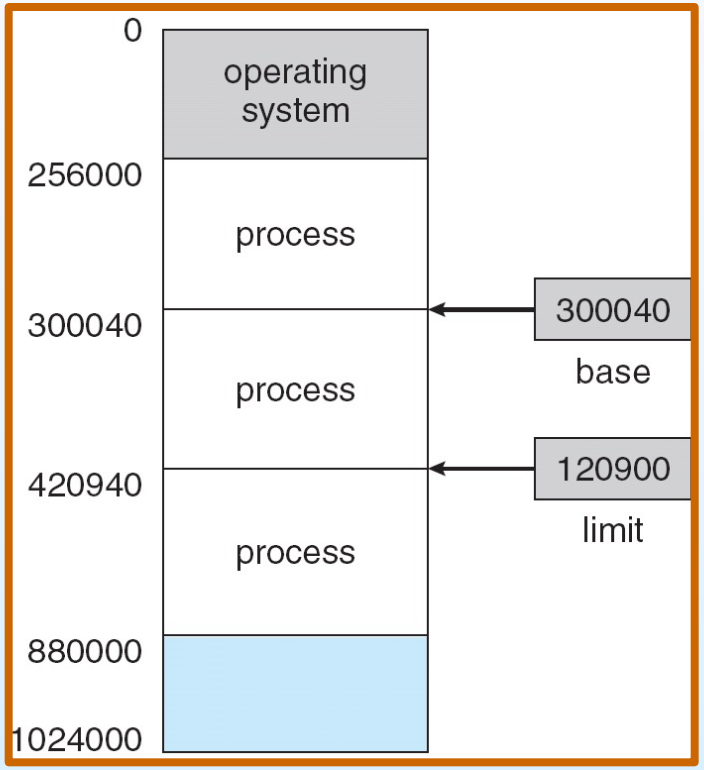


Memory Hierarchy

	L1 Cache	L2 Cache	Main memory
Block size	16--32 bytes	32--64 bytes	4--16 KB
Size	16--64 KB	256KB—8MB	
Hit time	1 Clock Cycle	1—4 Clock Cycles	10—40 Clock Cycles
Backing Store	L2 Cache	Main memory	Disk
Block replacement	Random	Random	Replacement strategies
Miss penalty	4-20 clock cycles	40-200 clock cycles	~6M clock cycles

Base and Limit Registers

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

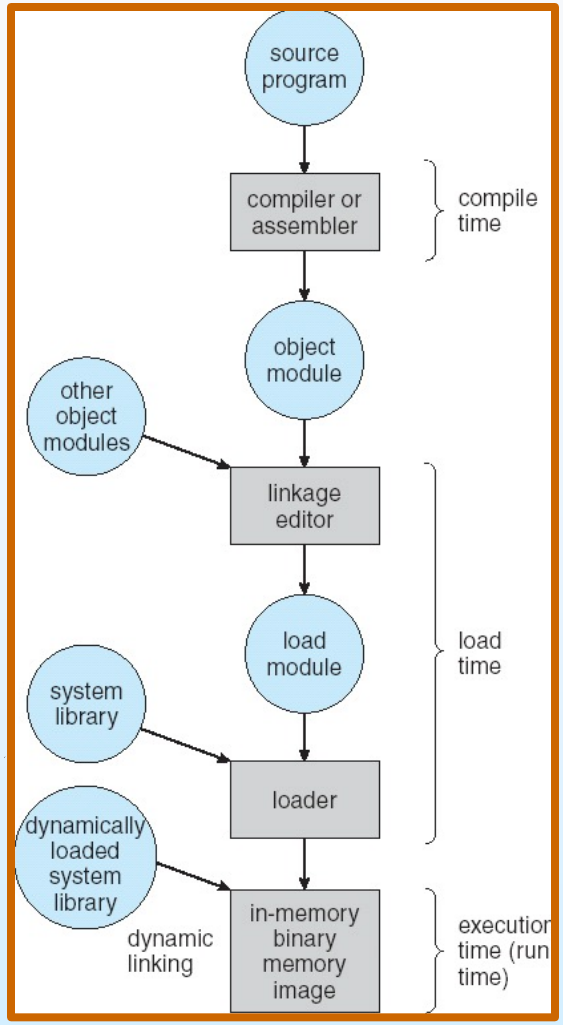


Binding of Instructions and Data to Memory

Mapping From one address space to another

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**（编译时刻）: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**（装入时刻）: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**（执行时刻）: 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)

Multistep Processing of a User Program



A **compiler** is a computer program (or set of programs) that transforms source code written in a computer language (the ***source language***) into another computer language (the ***target language***, often having a binary form known as ***object code***).

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program. On Unix variants the term **loader** is often used as a synonym for linker.

Logical vs. Physical Address Space

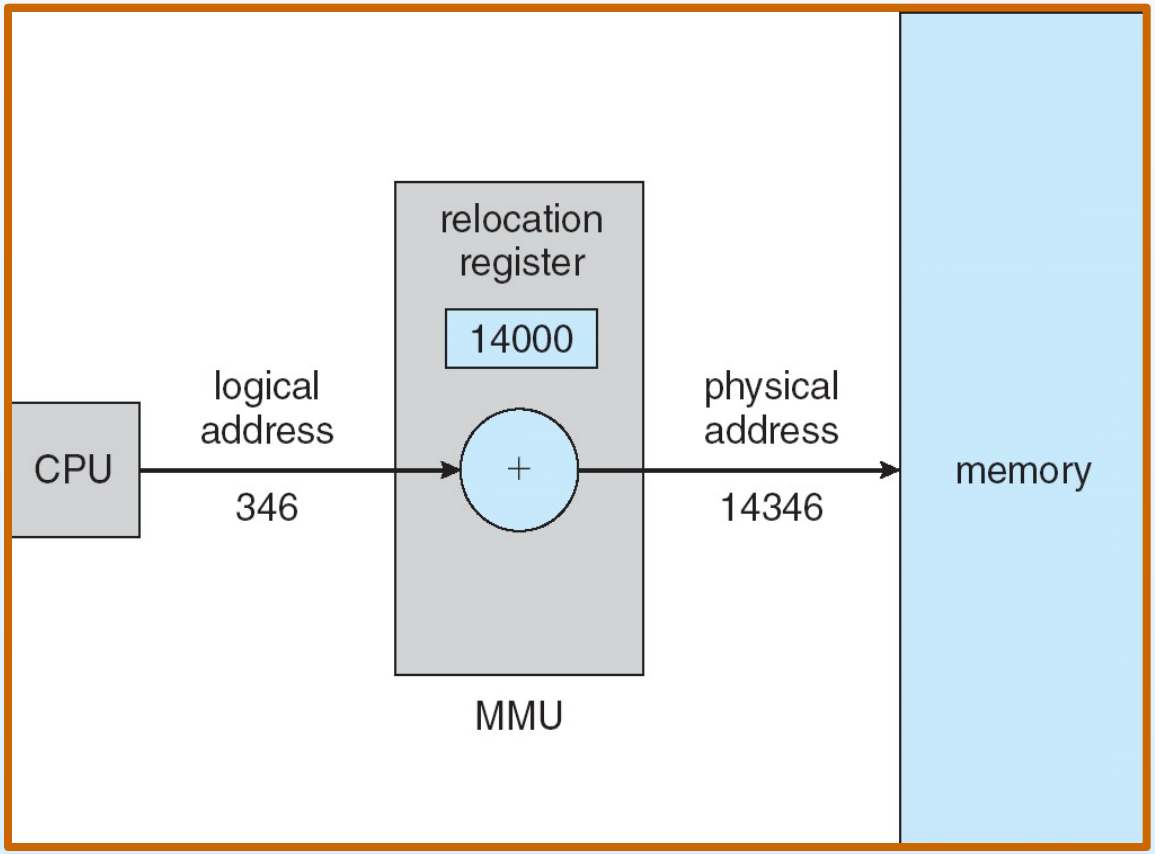
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical 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

A simple MMU



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

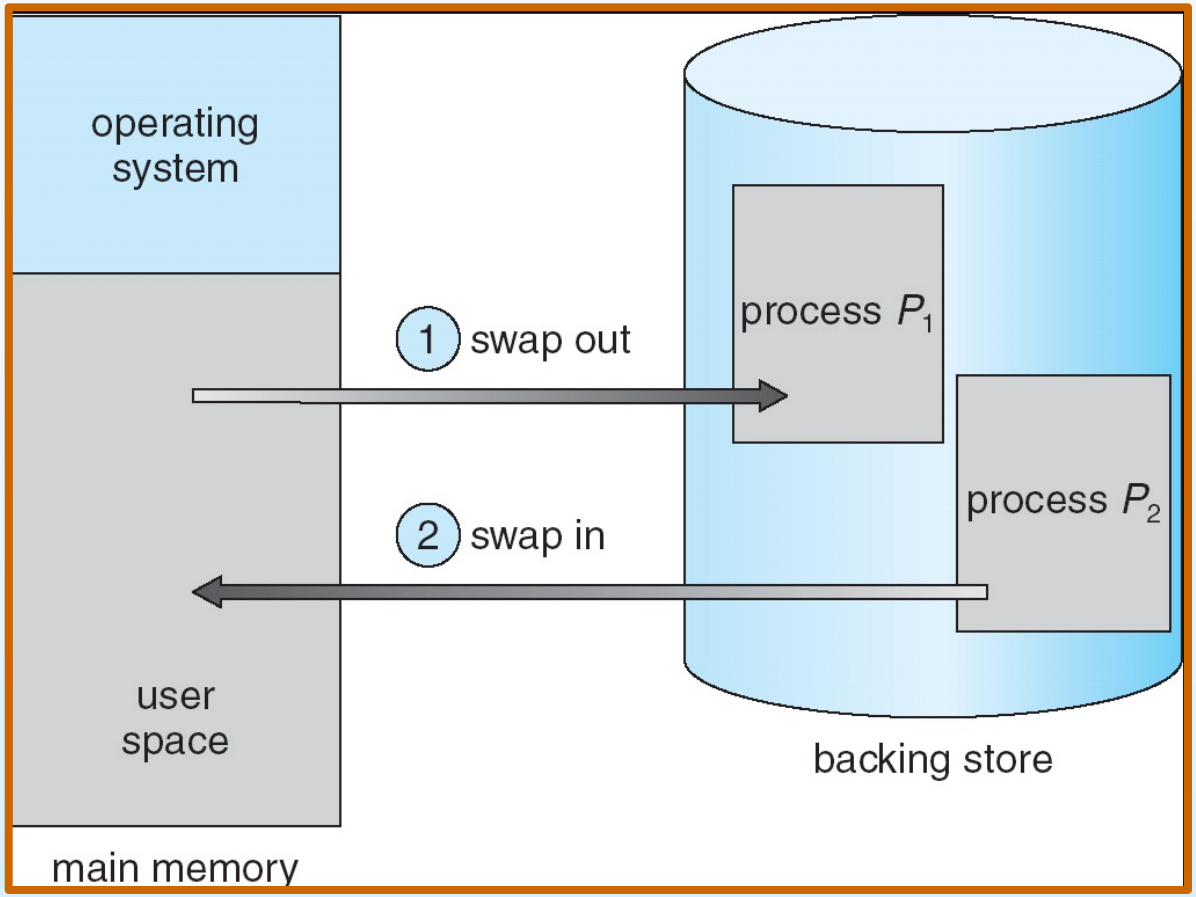
Dynamic Linking

- 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**
- Relinking of new library not needed

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

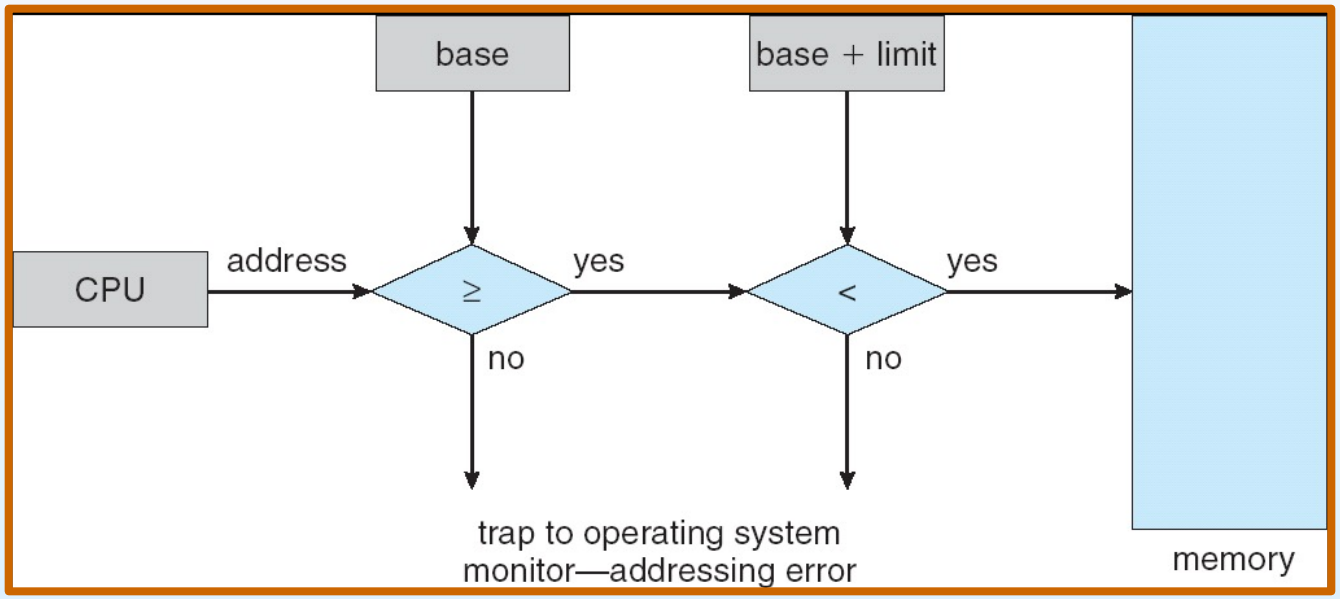


Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory

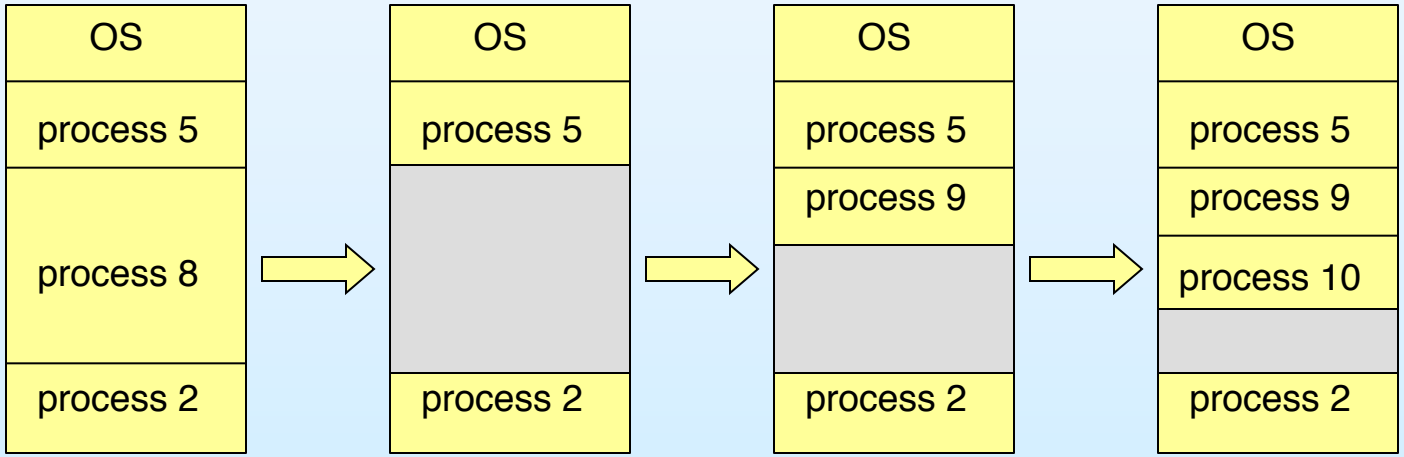
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest *physical* address
  - Limit register contains range of *logical* addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

HW address protection with base 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

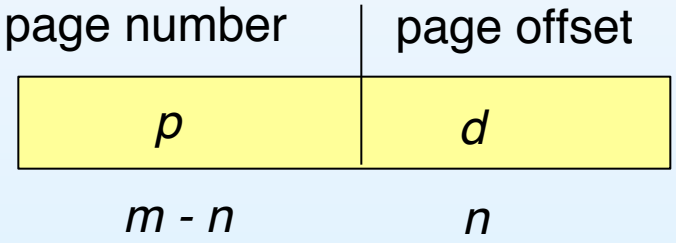
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** (refer to the textbook p287) – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- 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
- Another solution to external frag. is non-contiguous allocation

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

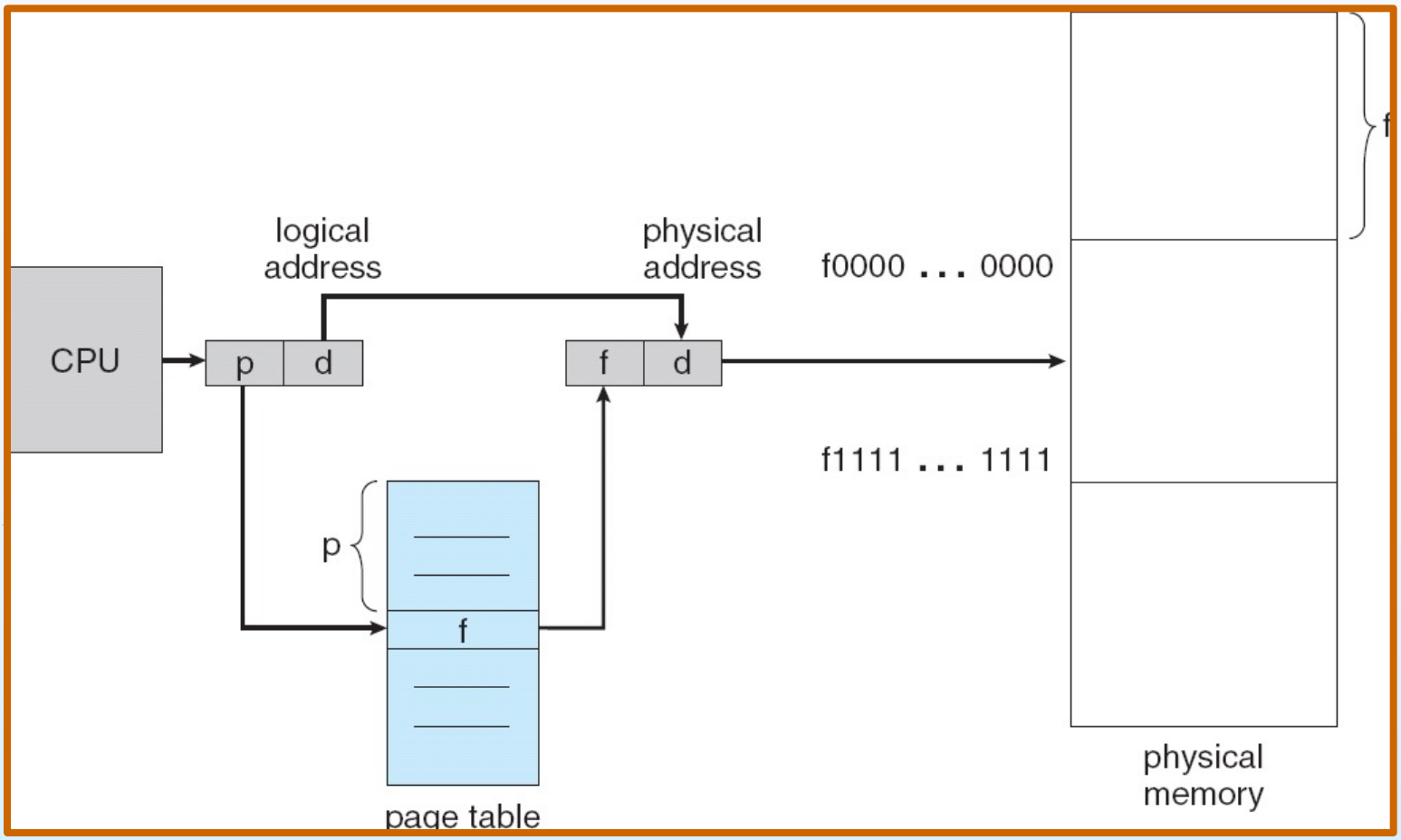
Address Translation Scheme

- 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

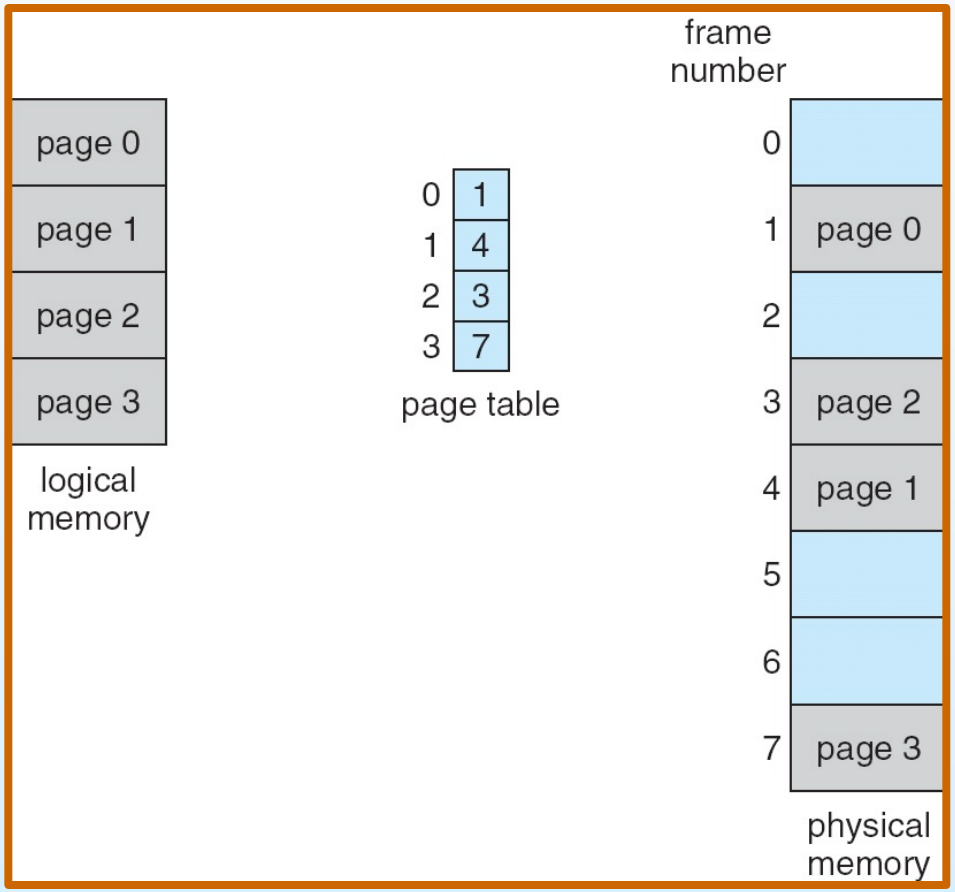


  - For given logical address space 2^m *and page size* 2^n

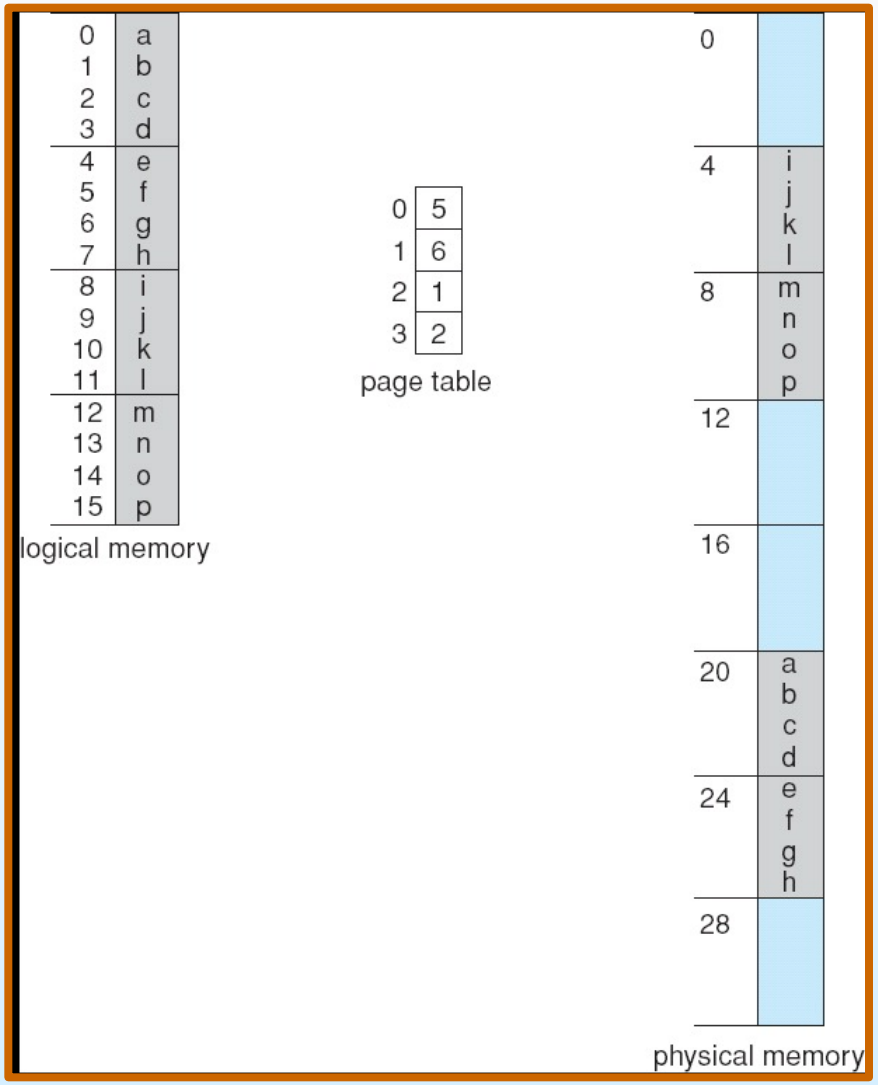
Paging Hardware



Paging Model of Logical and Physical Memory

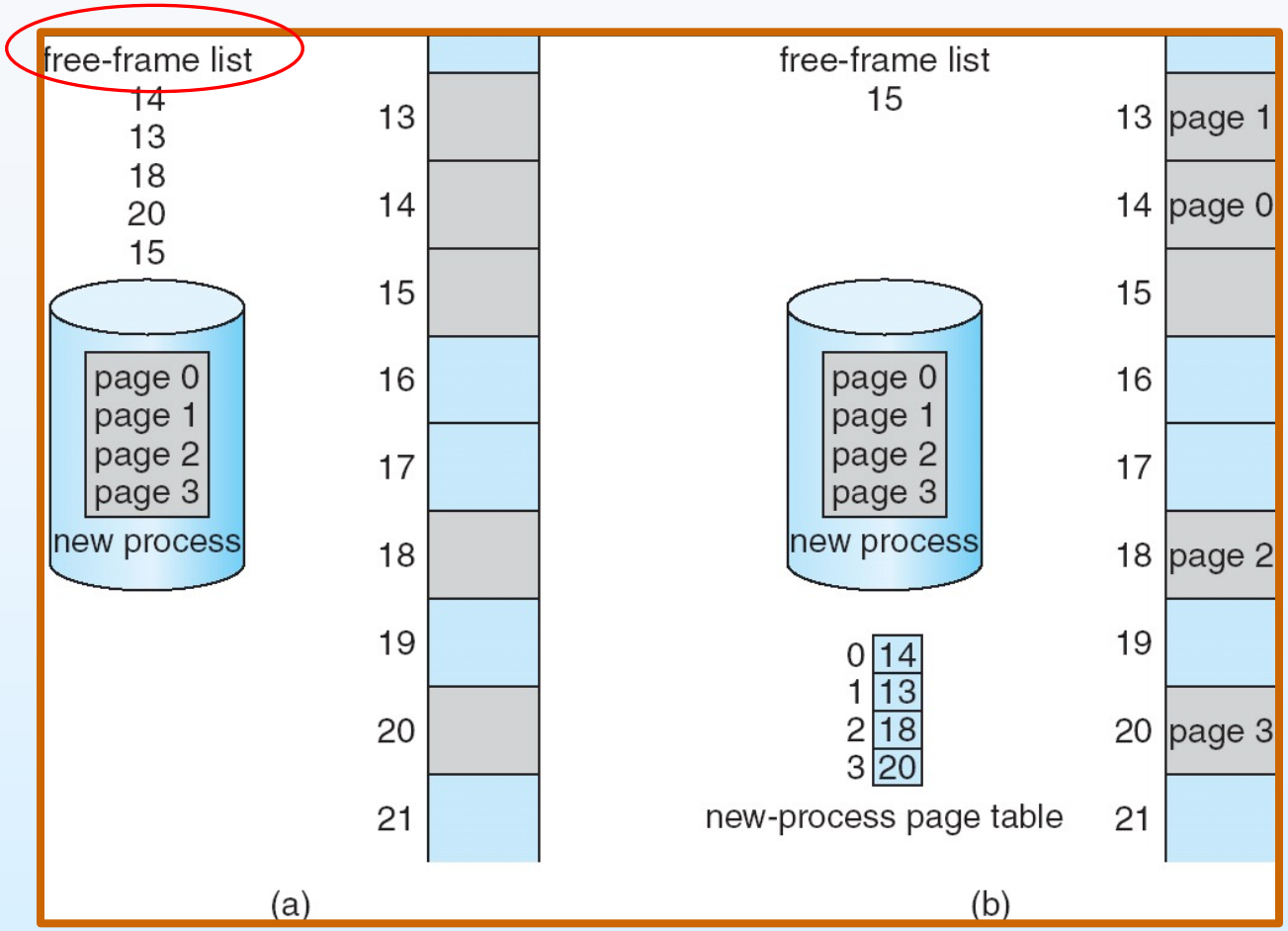


Paging Example



32-byte memory and 4-byte pages

Free Frames



Before allocation

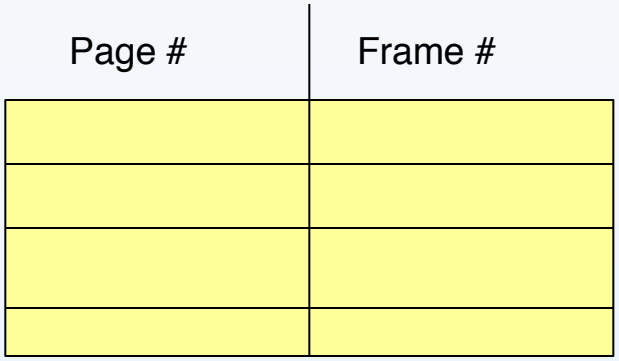
After allocation

Hardware 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** 转换旁视缓冲**,** 一称快表**)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

TLB

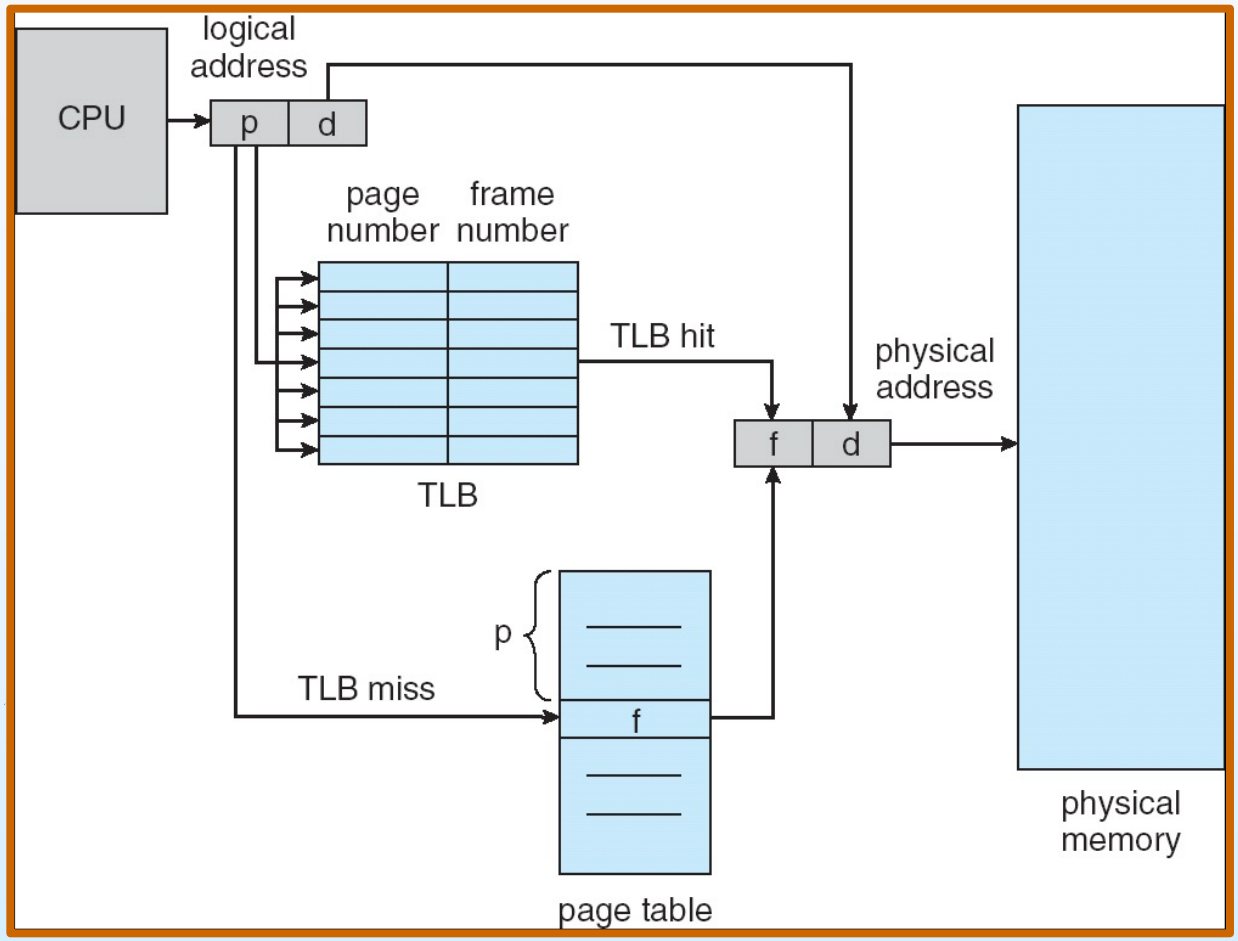
- TLB – parallel search



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

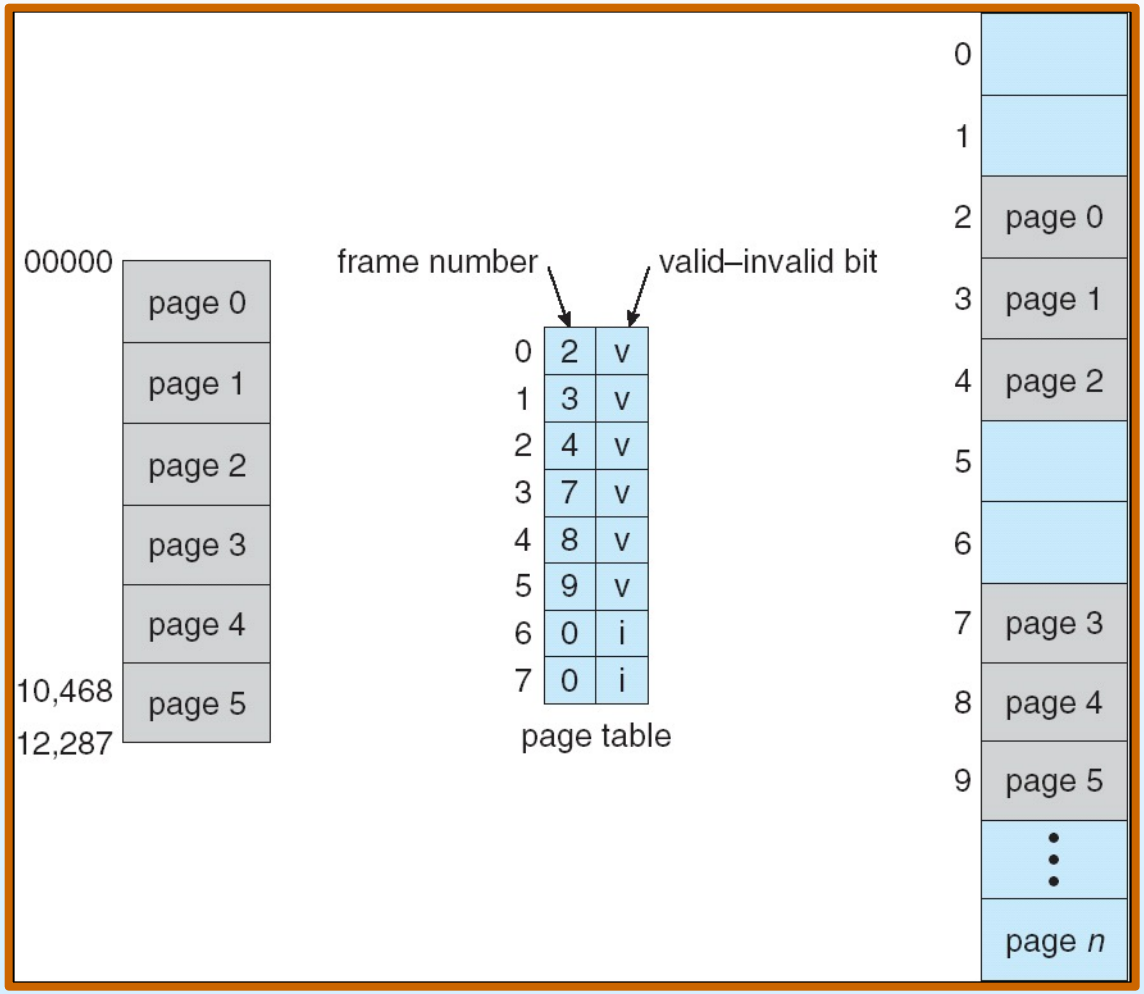
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

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

Memory Protection in Paged Scheme

- 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

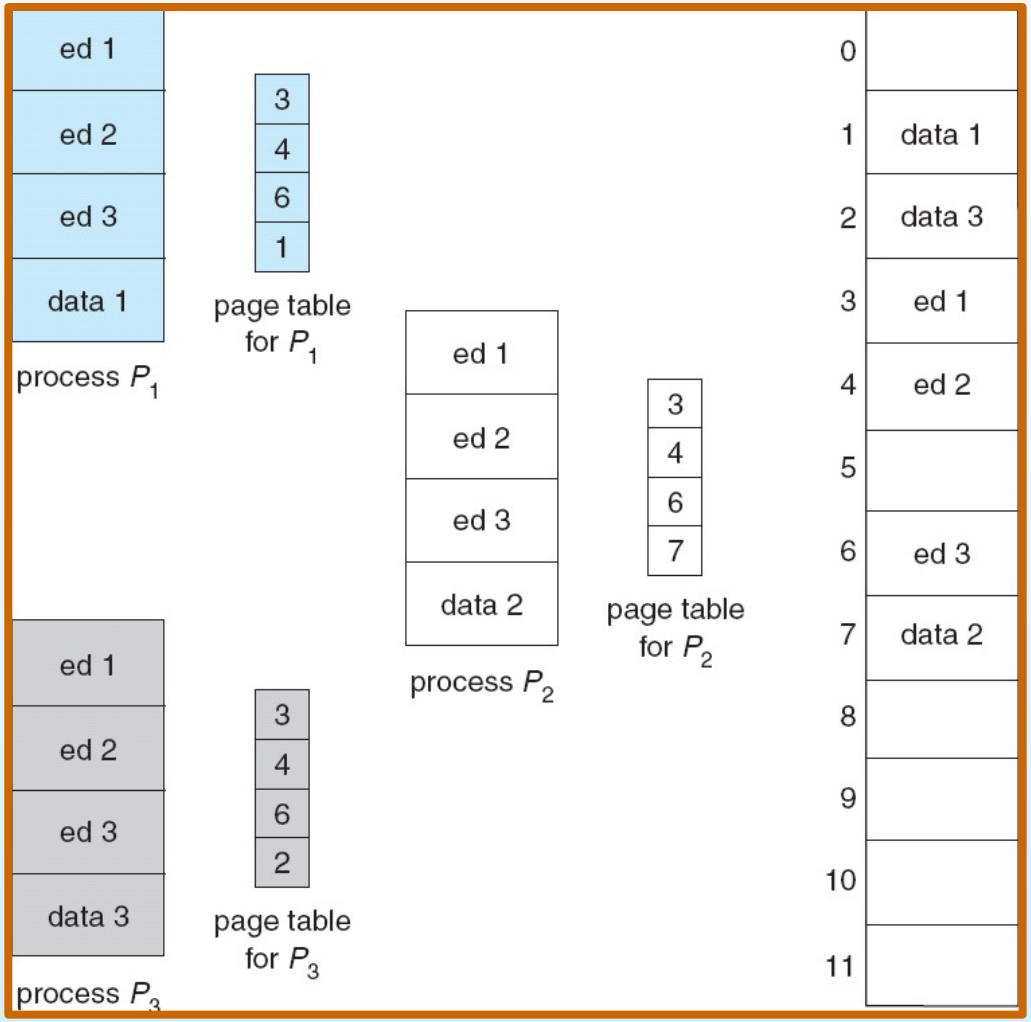


Shared Pages

- **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



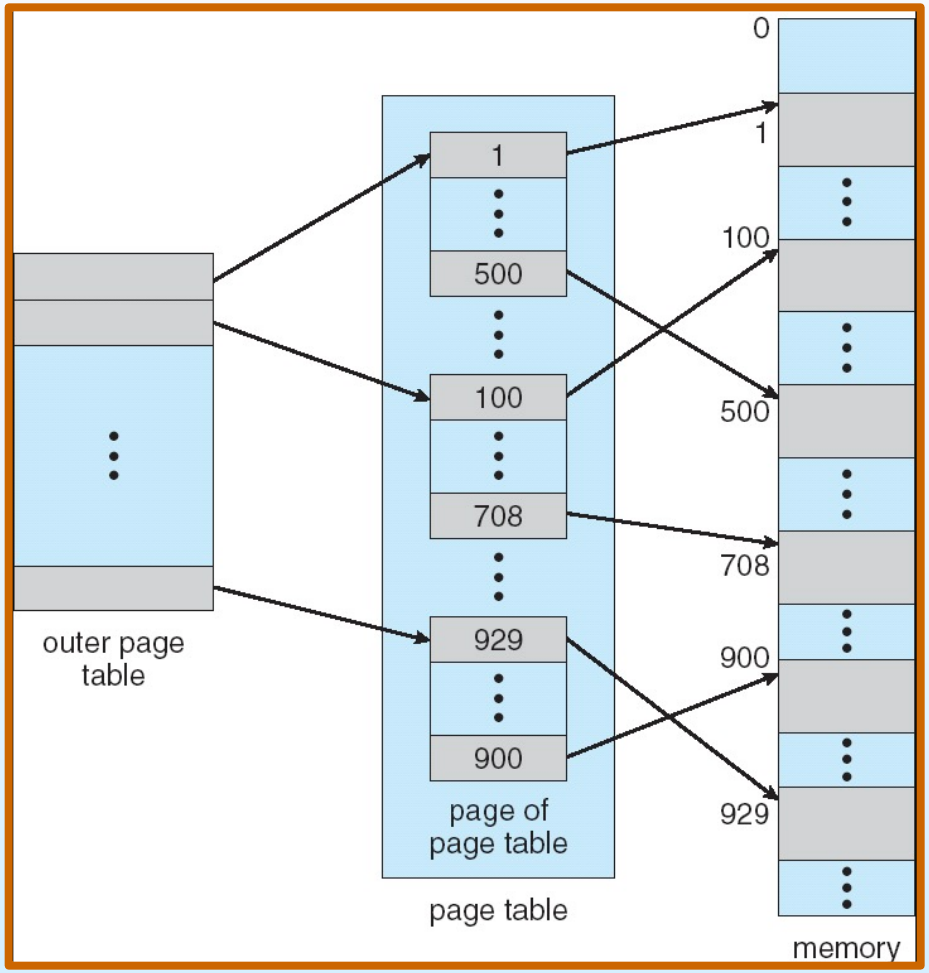
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

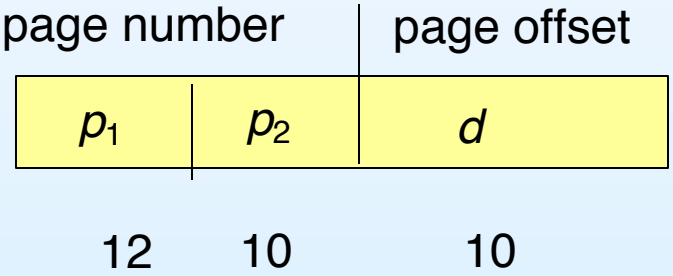
- Break up the logical address space into multiple page tables – to page the page table
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



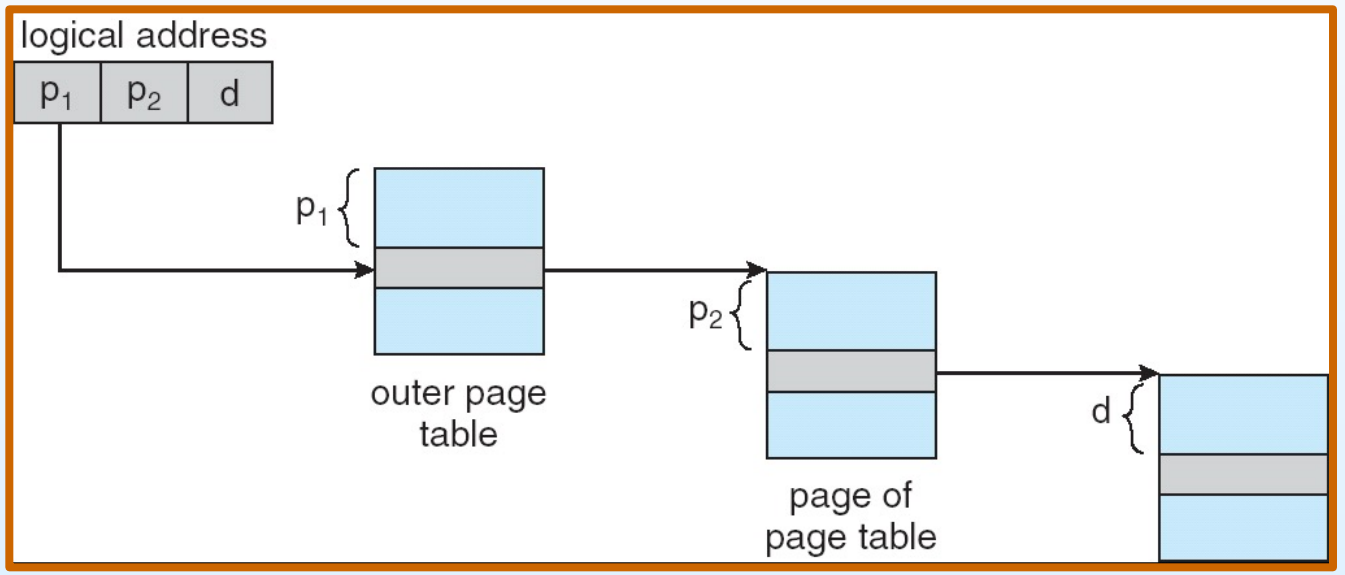
Two-Level Paging Example

- 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:



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

Address-Translation Scheme



Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

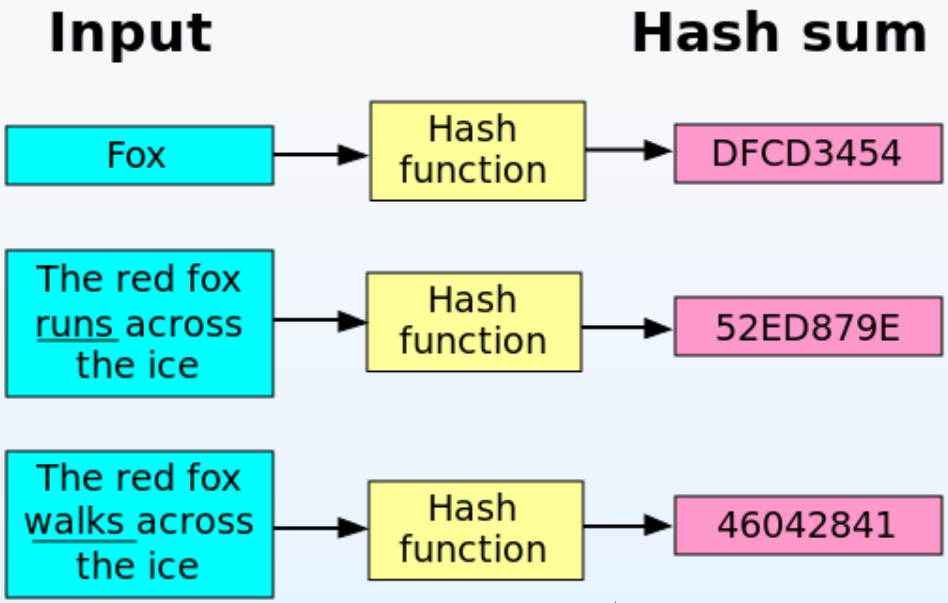
太大了, 所以要切分.

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

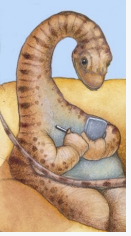
Hashed Page Tables

- 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.

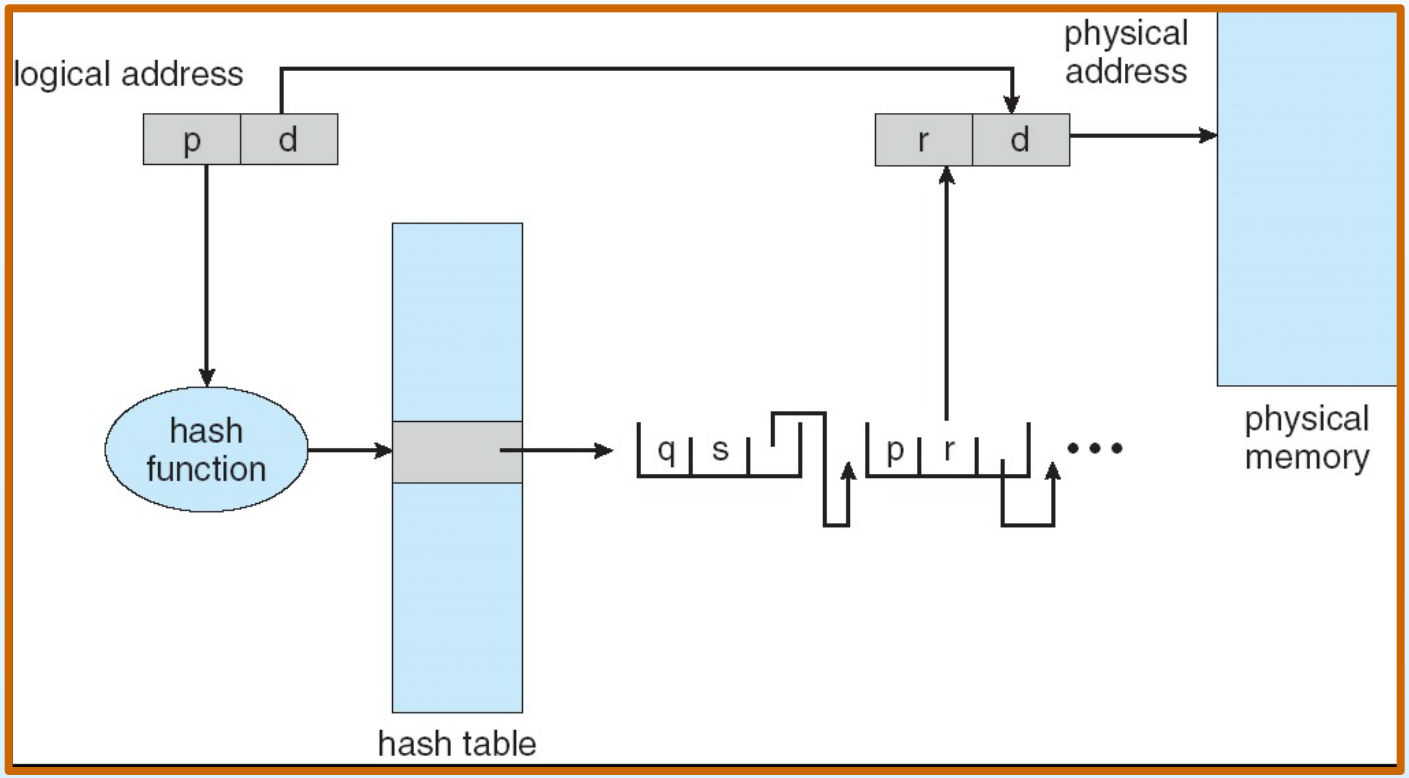
Hash Table



$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$



Hashed Page Table



Inverted Page Table

Only one page table in the whole system

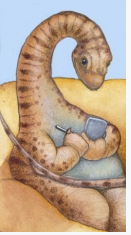
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference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

Why Inverted Page Table (IPT)

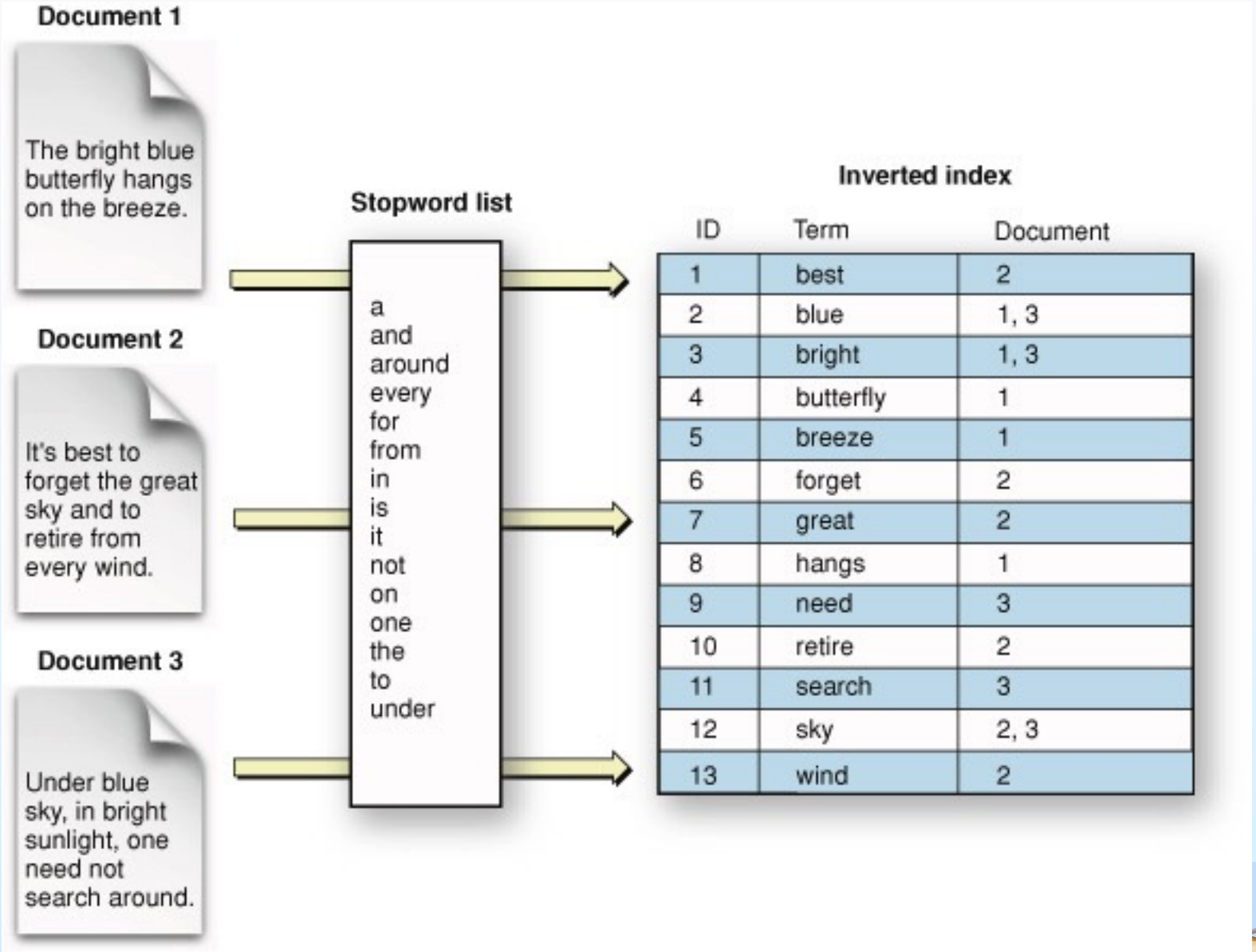
- Problem:
  - Page table overhead increases with address space size
  - Page tables get too big to fit in memory!
- Consider a computer with 64 bit addresses
  - Assume 4 Kbyte pages (12 bits for the offset)
  - Virtual address space = 2^{52} pages!
  - Page table needs 2^{52} entries!
  - This page table is too large for memory!
    - Many peta-bytes per process page table

Why Inverted Page Table (IPT)

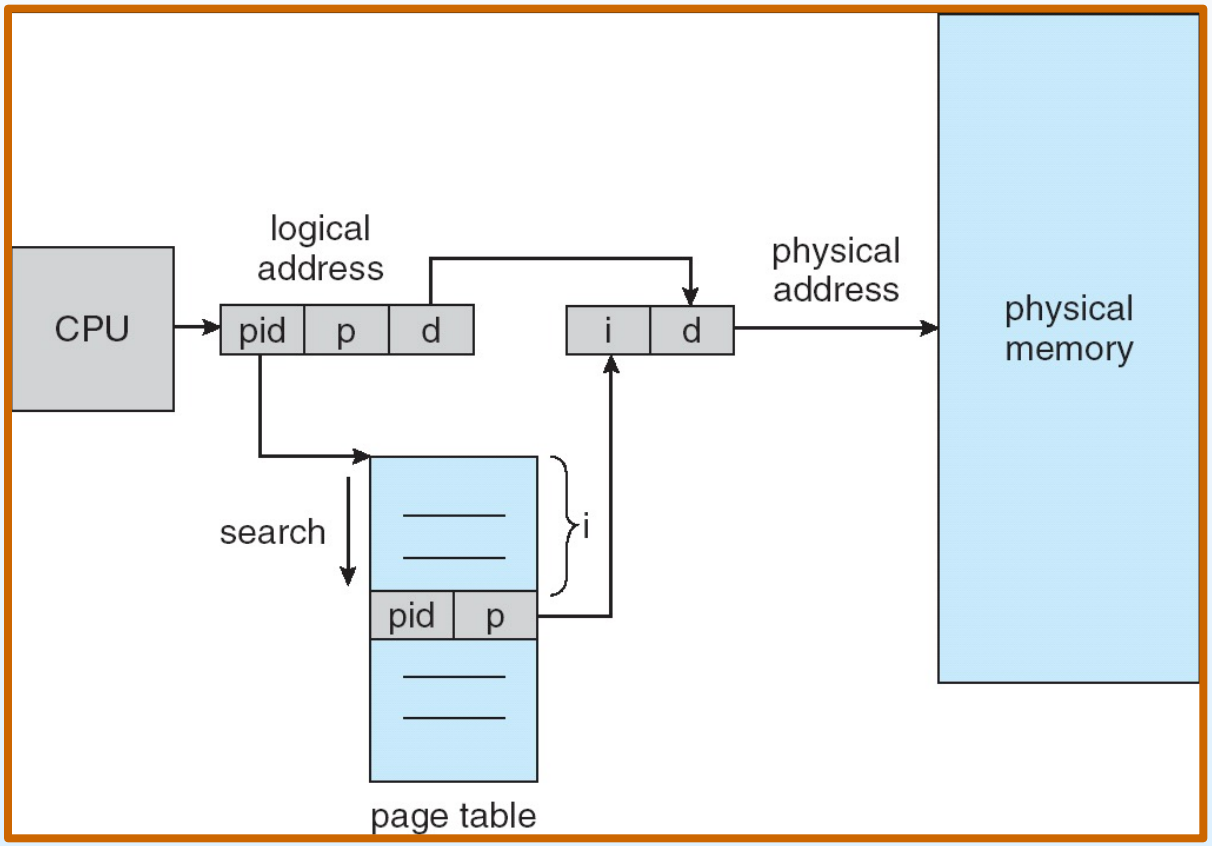
- How many mappings do we need (maximum) at any time?
- We only need mappings for pages that are in memory!
  - A 256 Kbyte memory can only hold 64 4Kbyte pages Only need 64 page table entries on this computer!
- An inverted page table
  - Has one entry for every frame of memory
  - Records which page is in that frame
  - Is indexed by frame number not page number!
  - So how can we search an inverted page table on a TLB miss fault?



What is the Inverted Index?

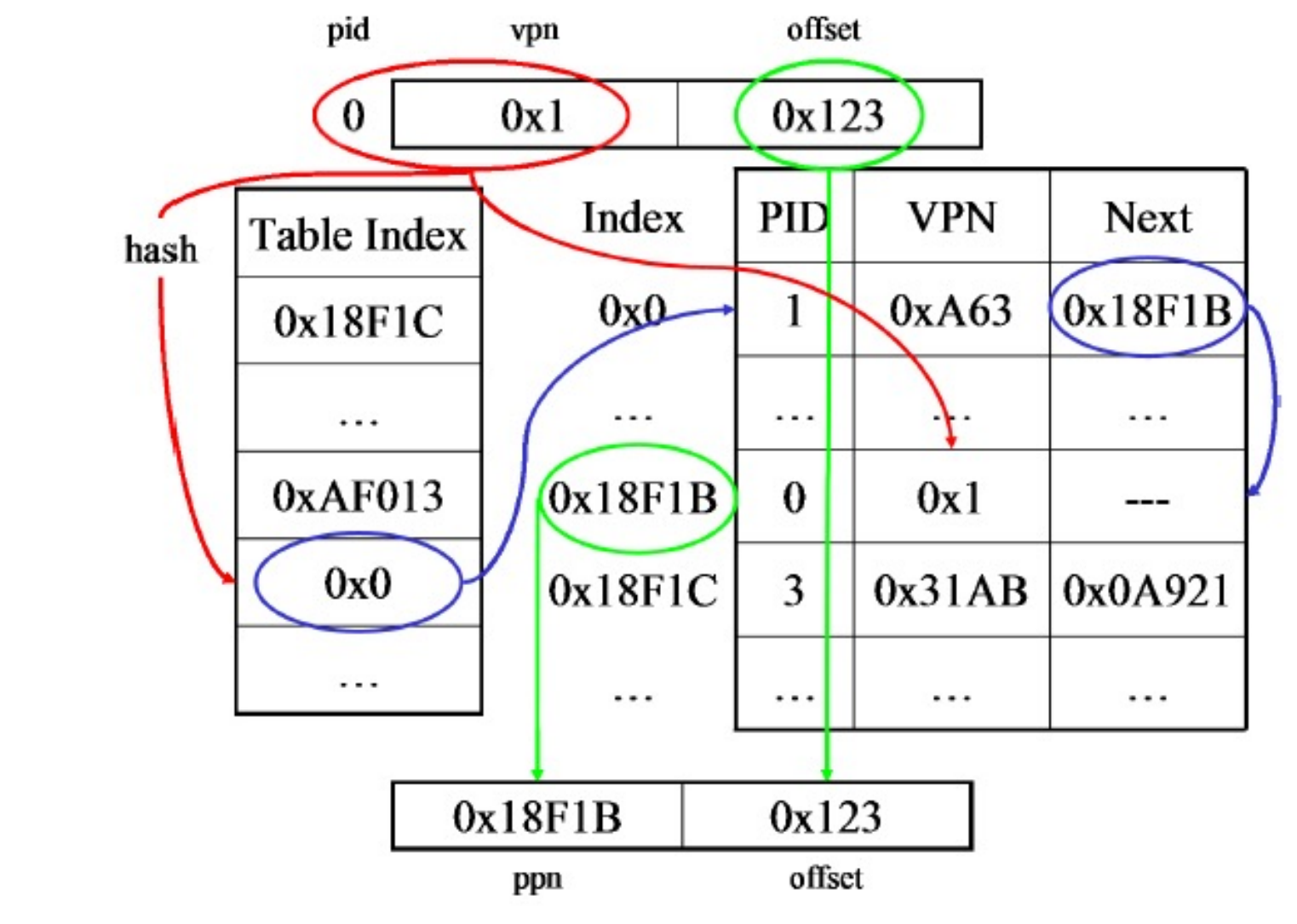


Inverted Page Table Architecture



Search is slow, so put page table entries into a hash table.
TLB can be used to speed up hash-table reference.

Inverted Page Table Architecture



Inverted Page Table (Exercise)

斯坦福期末考试

Assume we have a simple, demand paging environment, with no segmentation. Processes P1 and P2 both have logical memory addresses in the range 0 . . . 99, inclusive. The page size is 5.

The hash table portion of the structure uses a hash function which simply calculates the index by adding the numerical portion of the process identifier and the logical page number. For example, if process P1 wants to access logical page 3, we get a hash value 1+3=4. Note that this is a really bad hash function because it requires a large hash table, but for our exercise, it is sufficient. The hash table and the inverted page table shown are below.

Inverted Page Table (Exercise)

Hashtable

Index	Process ID / logical page	Frame ID
…	…	…
3		
4	P2 2	1
5	P1 3	4
…	…	…
18	P1 17	2
19	P1 18	3
…	…	…
30		

Inverted Page Table

Index	Logical address info (minimized for simplicity)
0	
1	P2:RW
2	P1:RW
3	P1:R
4	P1:RW

Inverted Page Table (Exercise)

Please compute hash values for the below memory requests where the number after the colon is the logical address (not the logical page).

(a) P1:30,

(b) P2:22

(a)hash value is: 1+30/5 = 7
(b)hash value is: 2+22/5 = 6

Inverted Page Table (Exercise)

Calculate the physical addresses for the following logical addresses where the number after the colon is the logical address (not the logical page). Assume the first page of a process is page number 0, and that a mechanism exists to find a free frame in memory.

Logical page number is 86/5 = 17, offset is 1
so the hash value is 17+1=18
based on the page table, the position 18 is mapped to physical frame 2(2marks).
So the physical address is computed as 2*5+1=11

Inverted Page Table (Exercise)

When process P2 attempts to read logical address 7, what happens? Specifically, describe the changes in the data structures, and what the process perceives of these changes.

This causes a page fault. (1 mark)
Frame 0 is allocated, and the entry in the IPT is updated with the logical address information for P2 frame 1. (1 mark)
Next the hashtable entry at index 3 (1 + 2) is mapped from P2:1 to frame 0. (1 mark)
7/5 +P2的2
After this processing is complete, the P2 resumes execution as before. (1 mark)

缺页, 要找到物理地址中空的地方, 然后填上page table

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,

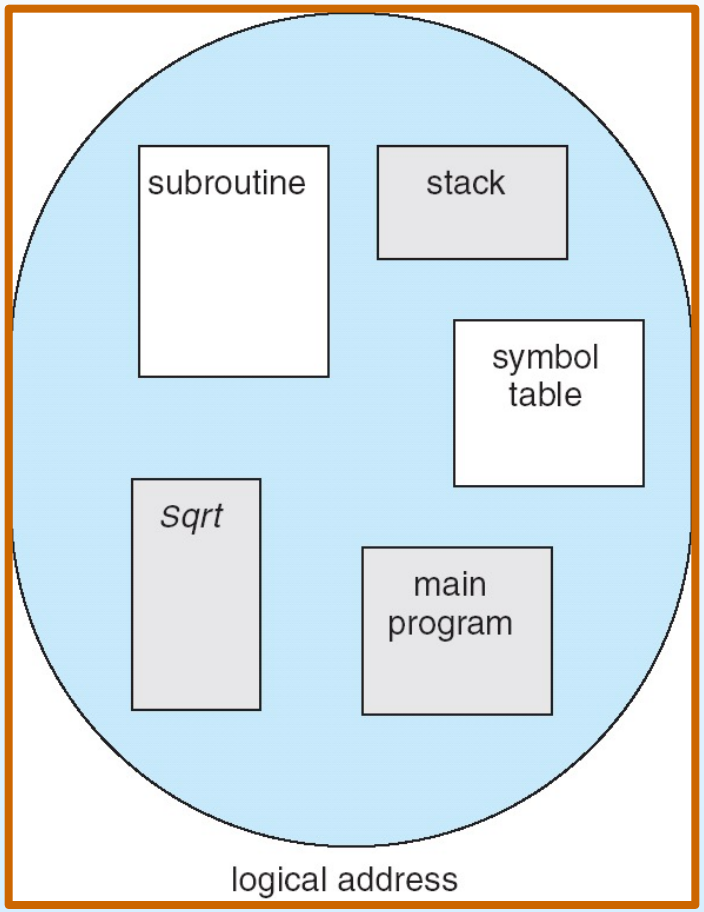
  local variables, global variables,

  common block,

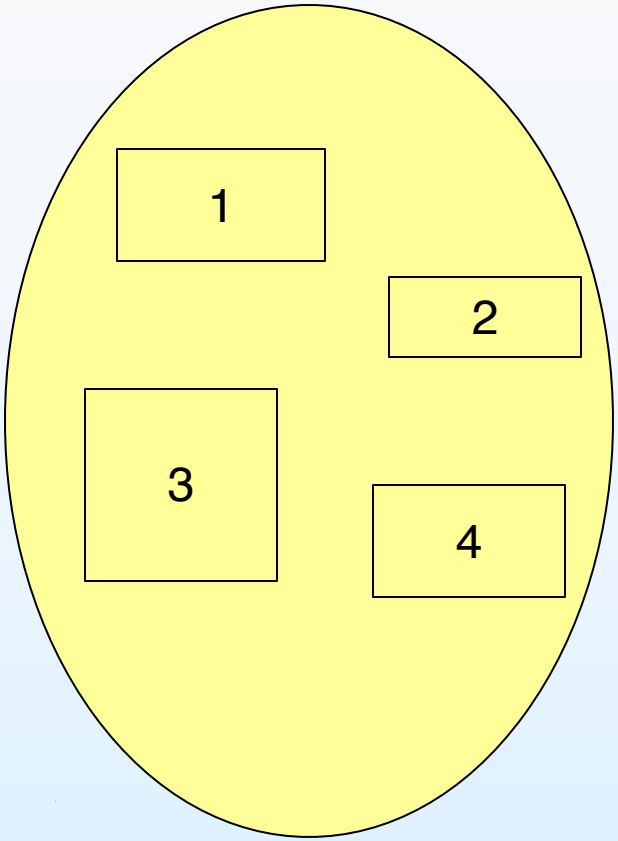
  stack,

  symbol table, arrays

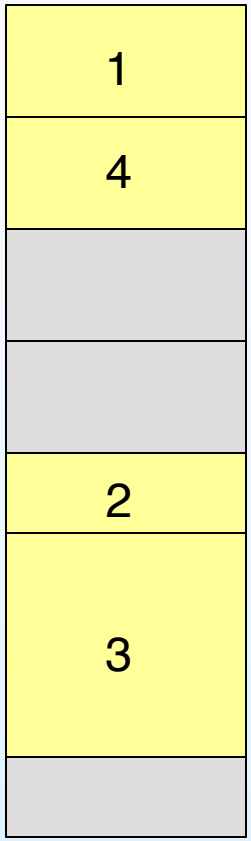
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

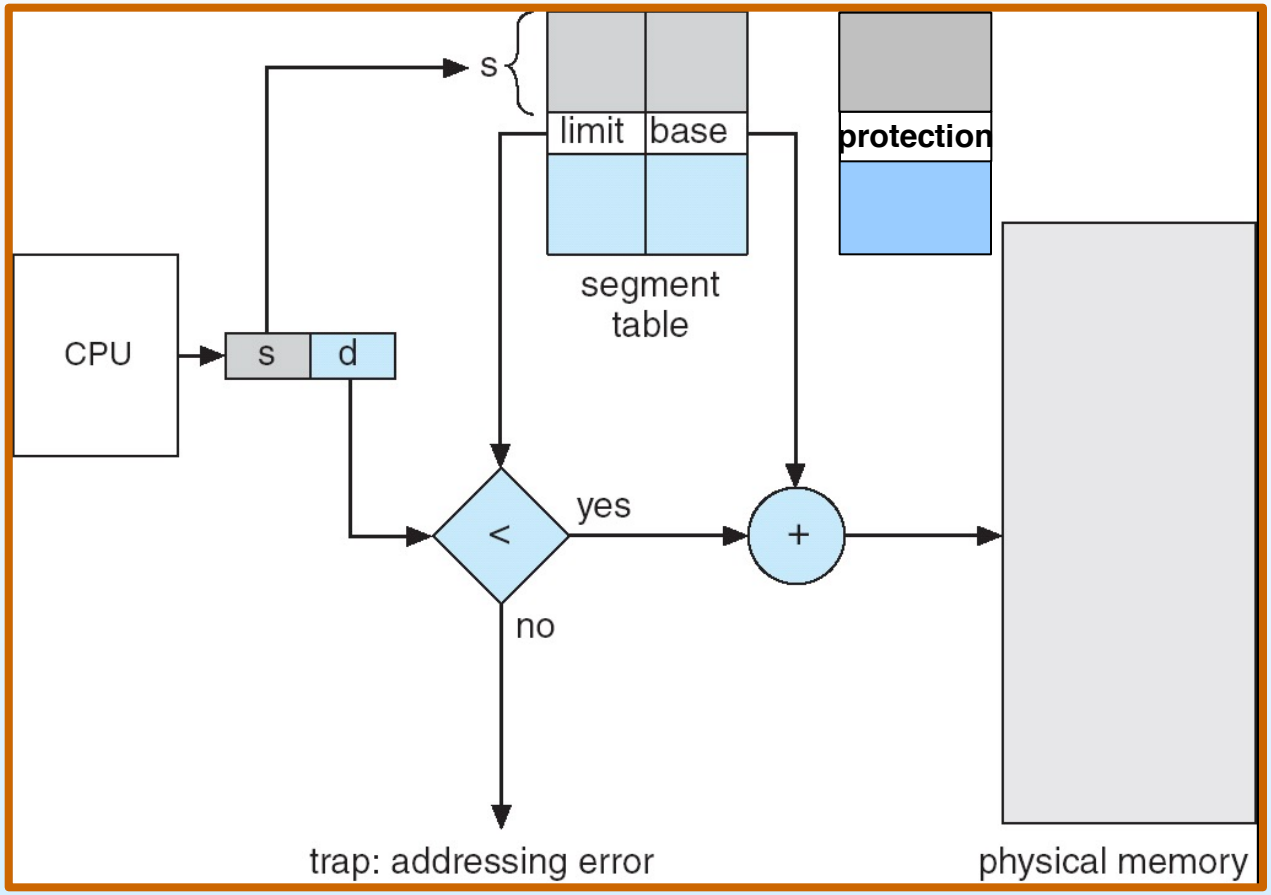
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

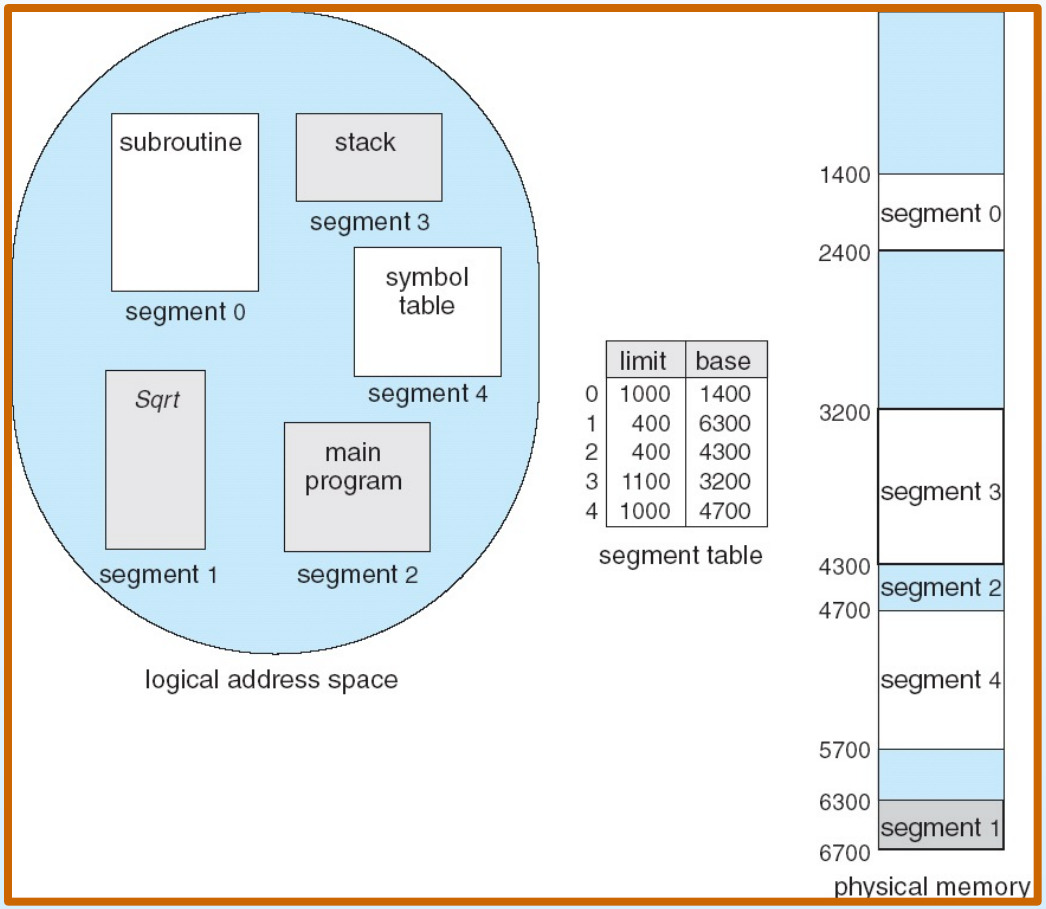
Segmentation Architecture (Cont.)

- 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



Segmentation vs. Paging

Segment is good logical unit of information
- sharing, protection

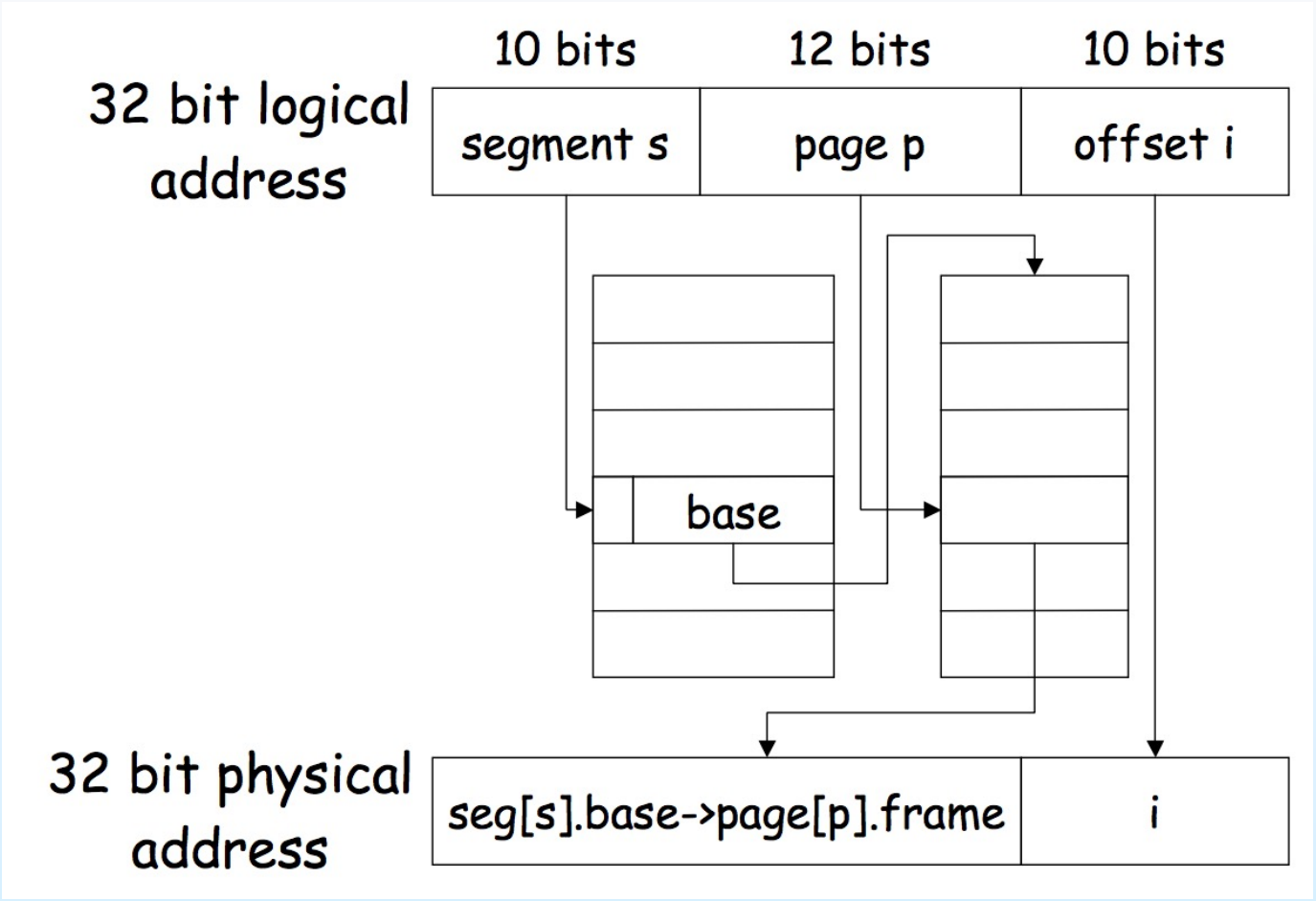
Page is good physical unit of information
- simple memory management

Best of both
- segmentation on top of paging

Segmentation on Paging

- Logical memory is composed of segments
- Each segment is composed of pages
- Segment table
  - per process, in memory pointed to by register
  - entries map seg # to page table base
  - shared segment: entry points to shared page table
- Page tables (like before)

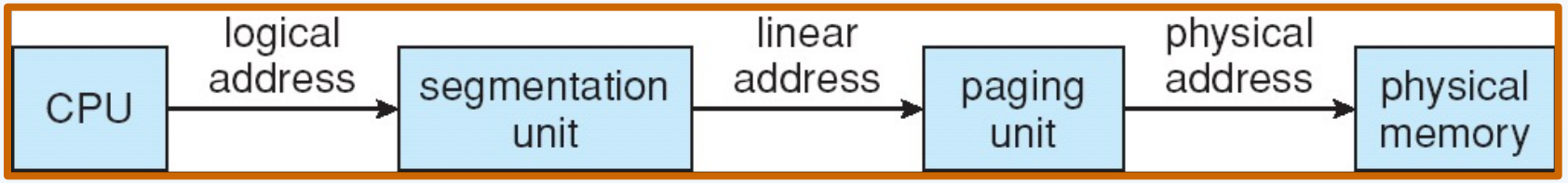
Segmentation + Paging



Example: The Intel Pentium

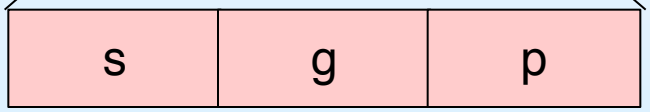
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Logical address space divided into local and global partitions.
  - LDT(local descriptor table)vs. GDT(global descriptor table)
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear 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



Logical address (a two-tuple): (selector, offset)

16-bit 32-bit



13 1 2

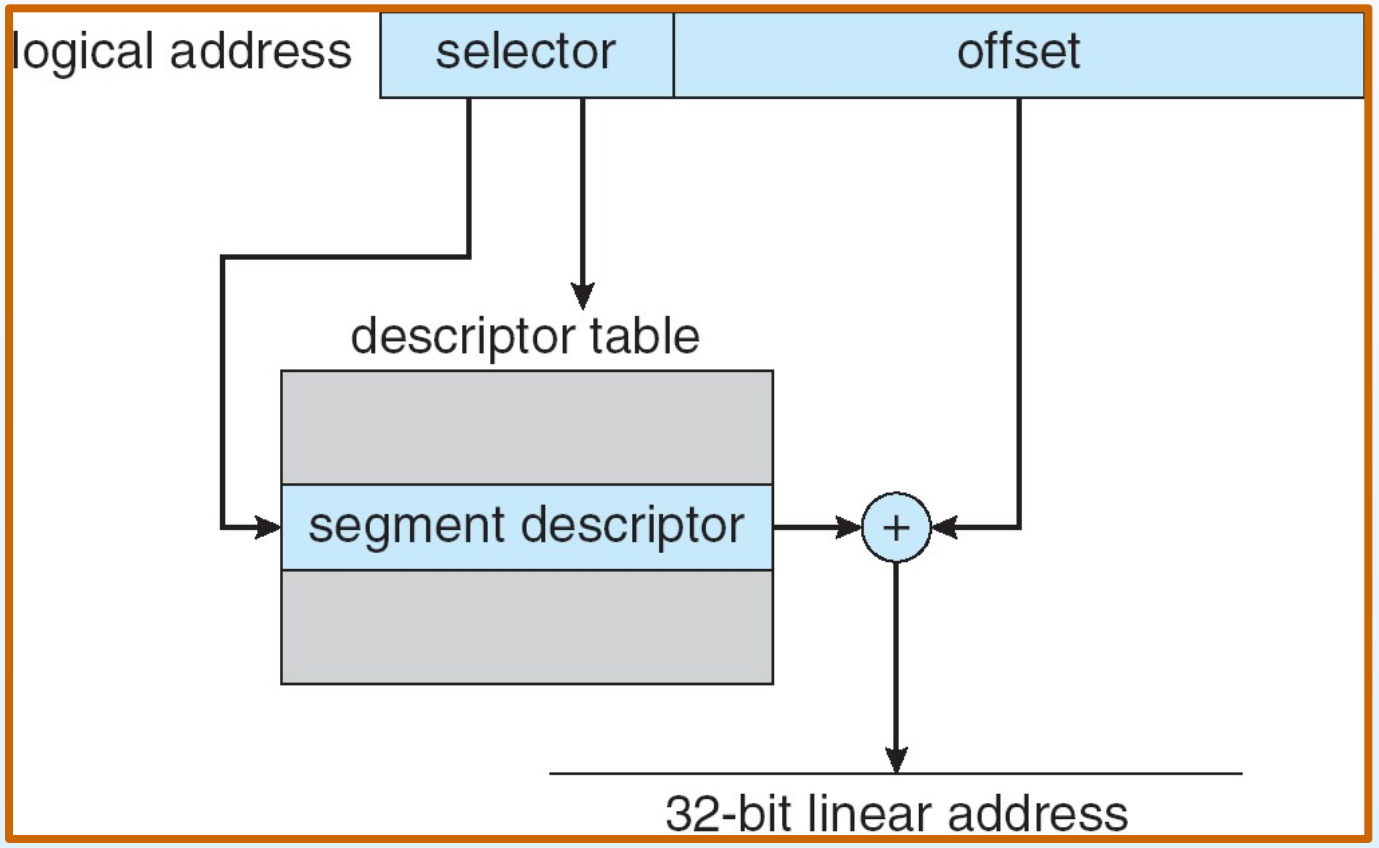
Segment GDT Protection

Linear Address

32-bit linear address

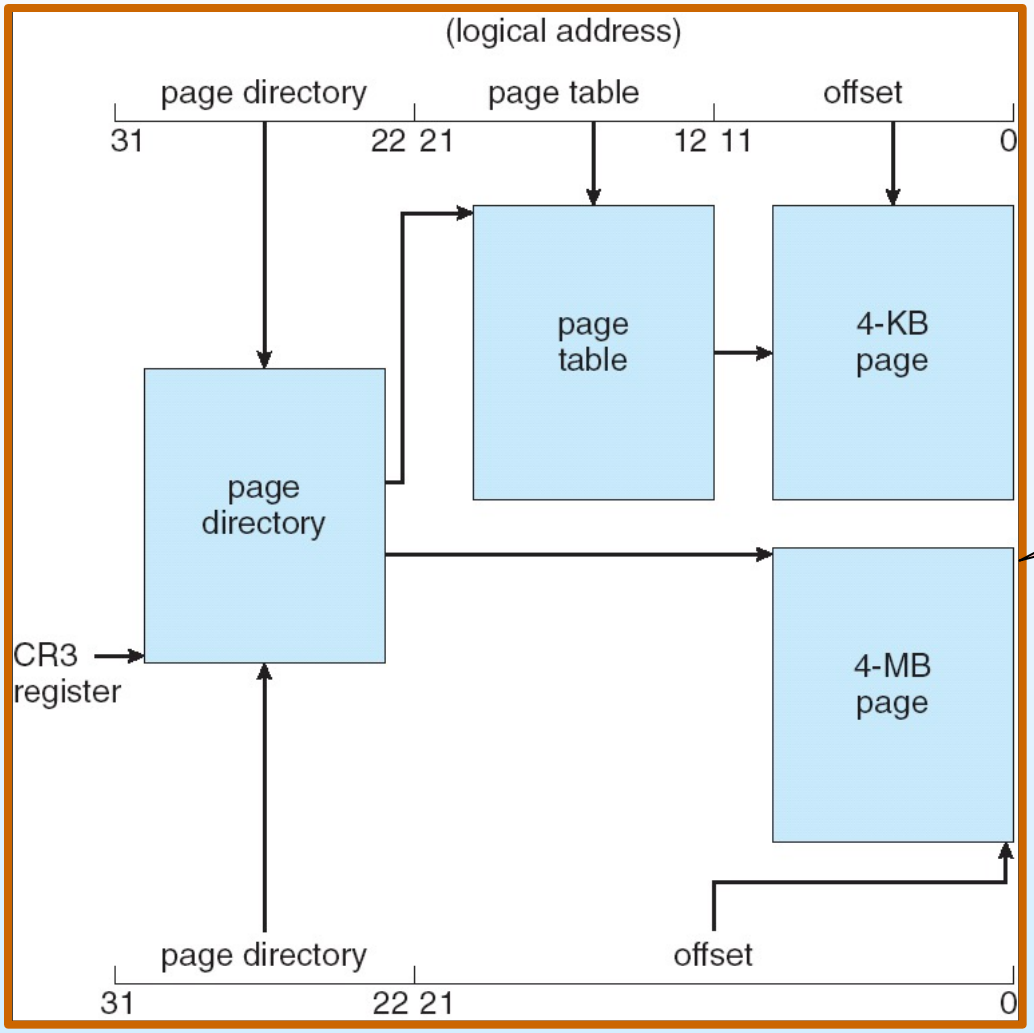
page number		page offset
p_1	p_2	d
10	10	12

Intel Pentium Segmentation



Each 8-byte segment descriptor contains base location and limit of the Segment.

Pentium Paging Architecture

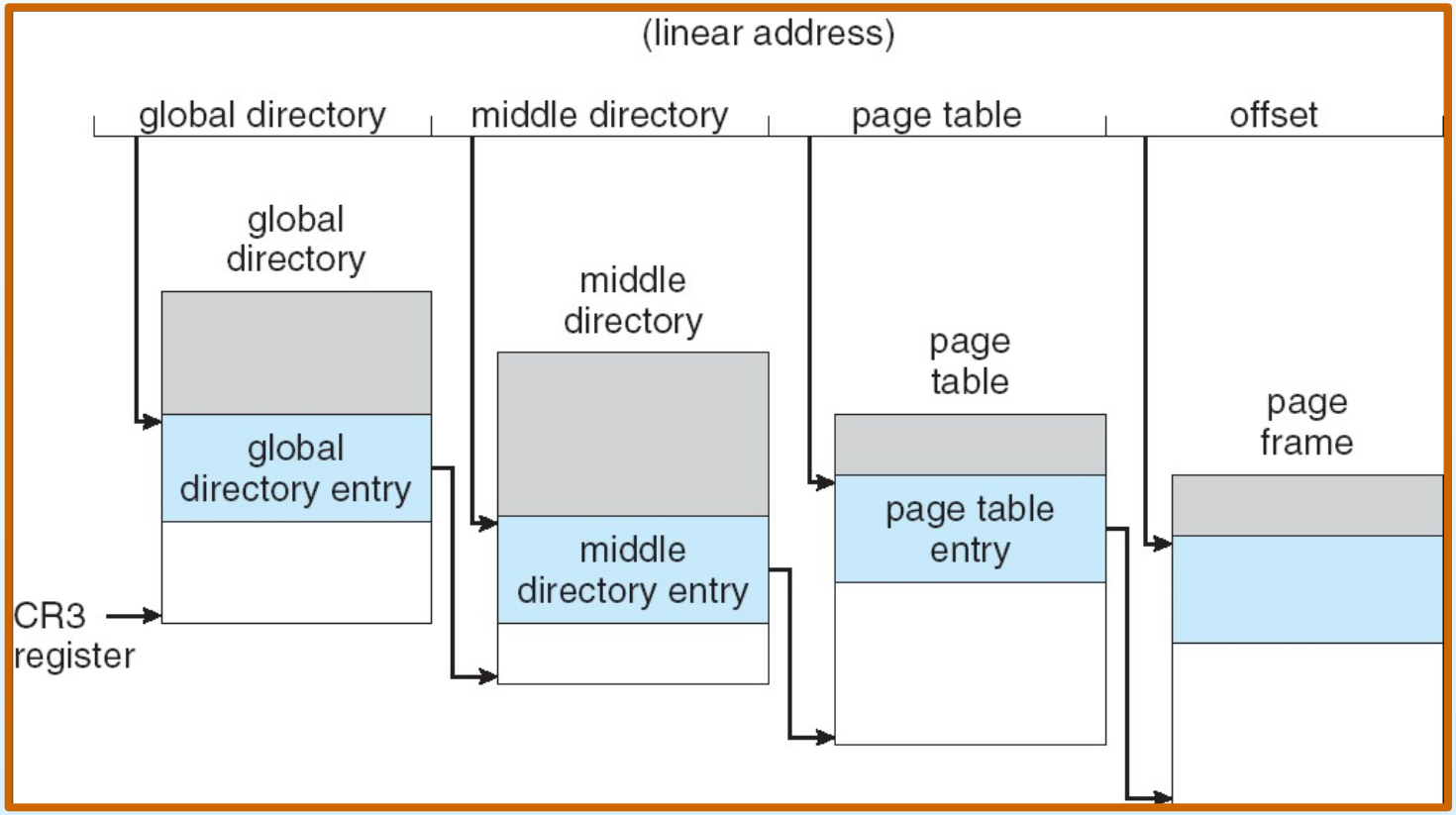


A page size entry in the page directory determines the frame size 4MB/4KB

Linear Address in Linux

Broken into four parts:

Three-level Paging in Linux



Segments in Linux

- __KERNEL_CS
- __KERNEL_DS
- __USER_CS (shared by all processes running in user mode)
- __USER_DS (shared by all processes running in user mode)
- TSS (task state segment)
- Default_LDT

End of Chapter 8

Exercise 1

Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

Exercise 1

a.

FCFS: P1-10-P2-1-P3-2-P4-1-P5-5

SJF: P2-1-P4-1-P3-2-P5-5-P1-10

Priority: P2-1-P5-5-P1-10-P3-2-P4-1

RR: P1-1-P2-1-P3-1-P4-1-P5-1-P1-1-P3-1-P5-1-P1-1-P5-1-P1-1-P5-1-P1-1-P5-1-P1-5

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

Exercise 1

Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
c. What is the waiting time of each process for each of the scheduling algorithms in part a?
d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?

Exercise 1

b.

	P1	P2	P3	P4	P5
FCFS	10	11	13	14	19
SJF	19	1	4	2	9
Priority	16	1	18	19	6
RR	19	2	7	4	14

c.

	P1	P2	P3	P4	P5
FCFS	0	10	11	13	14
SJF	9	0	2	1	4
Priority	6	0	16	18	1
RR	9	1	5	3	9

d.

FCFS (0+10+11+13+14)/5=9.6
SJF (9+0+2+1+4)/5=3.2
Priority (6+0+16+18+1)/5=8.2
RR (9+1+5+3+9)/5=5.4
Of course, SJF results in the minimal average waiting time.

Exercise 2

- The Sleeping-Barber Problem
- A barbershop consists of awaiting room with n chairs and a barber room with one barber chair.
  - If there are no customers to be served, the barber goes to sleep.
  - If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.
  - If the barber is busy but chairs are available, then the customer sits in one of the free chairs.
  - If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

Exercise 2

// Initial Values
define n 5； //Chair number is 5

semaphore mutex=1； //

semaphore customers=0；//Customer Number

semaphore barbers=1；//Barber number

int waiting=0； //Number of Occupied Chairs

Exercise 2

```
//Barber thread
void barber()
{
     while(true) //whether some customer is waiting?
     {
      wait(customers); //if no customer, sleeps
      wait(mutex); //
      waiting--; //number of waited customers minus 1
      signal(mutex); //
      signal(barber); //ask the client to get hair cut
      cut_hair; //
     }
}
```

Exercise 2

```
//customer thread
void customer()
{
      wait(mutex); //
      if (waiting<n) //empty chair available, so waiting
      {
       waiting++; //
       signal(mutex); //
       signal(customers); //wake up the barbers
       wait(barber); //barber is busy, so waiting
              get_haircut; //start cutting my hair
      }
      else
       signal(mutex); //it's full, just leave
}
```

Exercise 3

Consider the following snapshot of a system:

	Allocation	Max	Available
	ABCD	ABCD	ABCD
P0	0012	0012	1520
P1	1000	1750	
P2	1354	2356	
P3	0632	0652	
P4	0014	0656	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix Need?

b. Is the system in a safe state?

c. If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

Exercise 3

a.
P0 0000;
P1 0750;
P2 1002;
P3 0020;
P4 0642;

b.
Yes, P0 and P3 could get resources to run. After they finishing their jobs and release the resources, available is 1,11,6,4. Then others can run at the sequence < P0,P3,P2,P1,P4>. So it is at the safe state.

c.
Yes. Available is 1520>0420; Now the resources is 1100; P0 can run and release the resources it holds. And they can run following the sequence <P0,P2,P3,P1,P4>. So it is also at the safe state.

Exercise

- Four processes R1, R2, W1, and W2 share a buffer space named B. R1 reads a number from the keyboard and saves it into B, so that the saved number is only consumed by W1, which prints the number to the screen. R2 reads a character from a mouse and also saves the character into B, so that only W2 can print the character to a printer. Please write the synchronization code for the four processes so that no race condition may arise among them. Define semaphores if necessary.