

# Lesson 7 Memory management

## Content

1. Memory management - history
2. Segmentation
3. Paging and implementation
4. Page table
5. Segmentation with paging

# Why memory?

- CPU can perform only instruction that is stored in internal memory and all its data are stored in internal memory too
- Memory architecture:
  - Harvard architecture – different memory for program and for data,
  - von Neumann - the same memory for both program and data
- **Physical address space** – physical address is address in internal computer memory
  - Size of physical address depends on CPU, on size of address bus
  - Real physical memory is often smaller than the size of the address space
    - Depends on how much money you can spend for memory.
- **Logical address space** – generated by CPU, also referred as virtual address space. It is stored in memory, on hard disk or doesn't exist if it was not used.
  - Size of the logical address space depends on CPU but not on address bus

# How to use memory

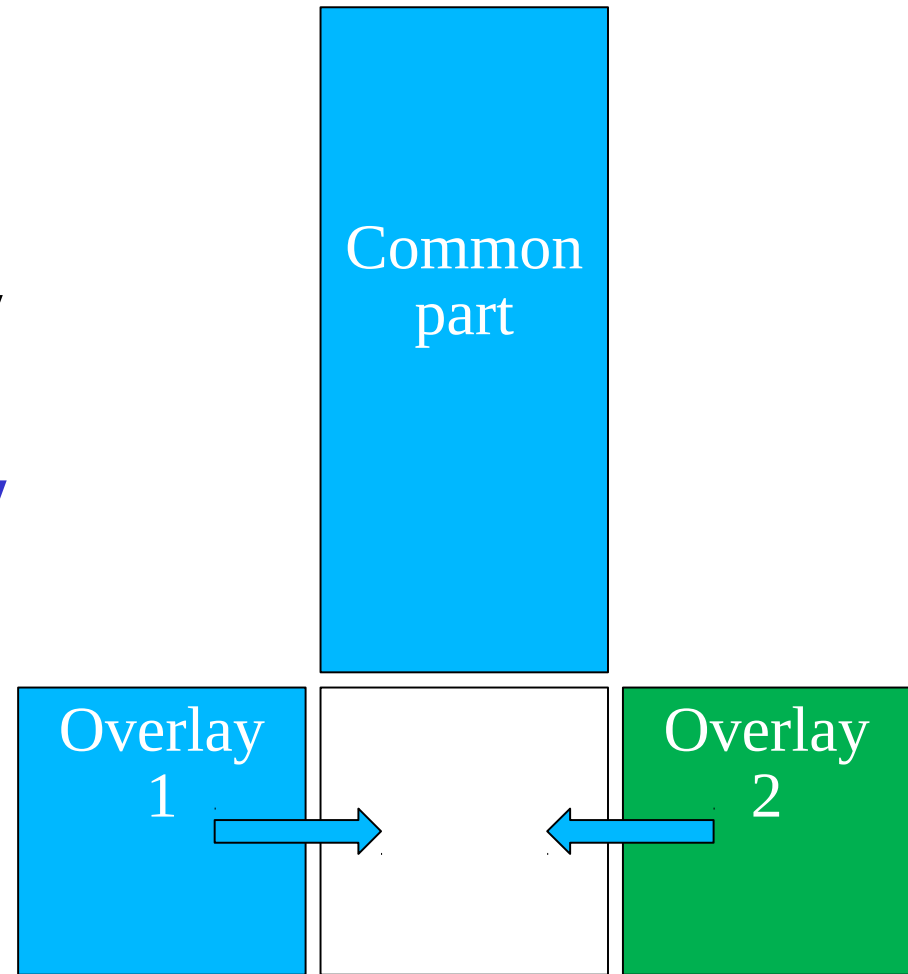
- Running program has to be placed into memory
- Program is transformed to structure that can be implemented by CPU by different steps
  - OS decides where the program will be and where the data for the program will be placed
  - Goal: **Bind address** of instructions and data to real address in address space
- Internal memory stores data and programs that are running or waiting
  - Long term memory is implemented by secondary memory (hard drive)
- Memory management is part of OS
  - Application has no access to control memory management
    - Privilege action
  - It is not safe to enable application to change memory management
    - It is not effective nor safe

# History of memory management

- First computer has no memory management – direct access to memory
- Advantage of system without memory management
  - Fast access to memory
  - Simple implementation
  - Can run without operating system
- Disadvantage
  - Cannot control access to memory
  - Strong connection to CPU architecture
  - Limited by CPU architecture
- Usage
  - First computer
  - 8 bits computers (CPU Intel 8080, Z80, ...) - 8 bits data bus, 16 bits address bus, maximum 64 kB of memory
  - Control computers – embedded (only simple control computers)

# First memory management - *Overlays*

- First solution, how to use more memory than the physical address space allows
  - Special instruction to switch part of the memory to access by address bus
- **Overlays** are defined by user and implemented by compiler
  - Minimal support from OS
  - It is not simple to divid data or program to overlays



# Virtual memory

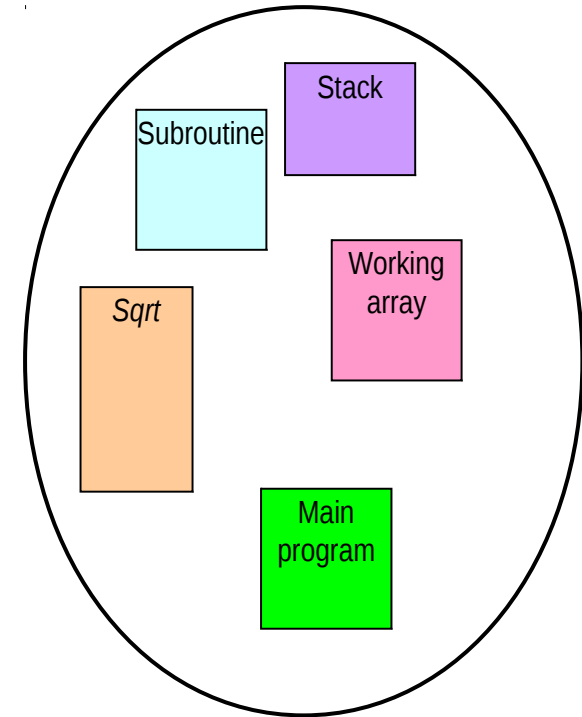
- Demand for bigger protected memory that is managed by somebody else (OS)
- Solution is virtual memory that is somehow mapped into real physical memory
- 1959-1962 first computer Atlas Computer from Manchesteru with virtual memory (size of the memory was 576 kB) implemented by paging
- 1961 - Burroughs creates computer B5000 that uses segment for virtual memory
- Intel
  - 1978 processor 8086 – first PC – simple segments
  - 1982 processor 80286 – protected mode – real segmentation
  - 1985 processor 80386 – full virtual memory with segmentation and paging

# Simple segments – Intel 8086

- Processor 8086 has 16 bits of data bus and 20 bits of address bus. 20 bits is problem. How to get 20 bits numbers?
- Solution is “simple” segments
- Address is composed with 16 bits address of segment and 16-bits address of offset inside of the segment.
- Physical address is computed as:  
$$(\text{segment} \ll 4) + \text{offset}$$
- It is not real virtual memory, only system how to use bigger memory
- Two types of address
  - near pointer – contains only address inside of the segment, segment is defined by CPU register
  - far pointer – pointer between segments, contains segment description and offset

# Segmentation – protected mode Intel 80286

- Support for user definition of logical address space
  - Program is set of segments
  - Each segment has it's own meaning: main program, function, data, library, variable, array, ...
- Basic goal – how to transform address (segment, offset) to physical address

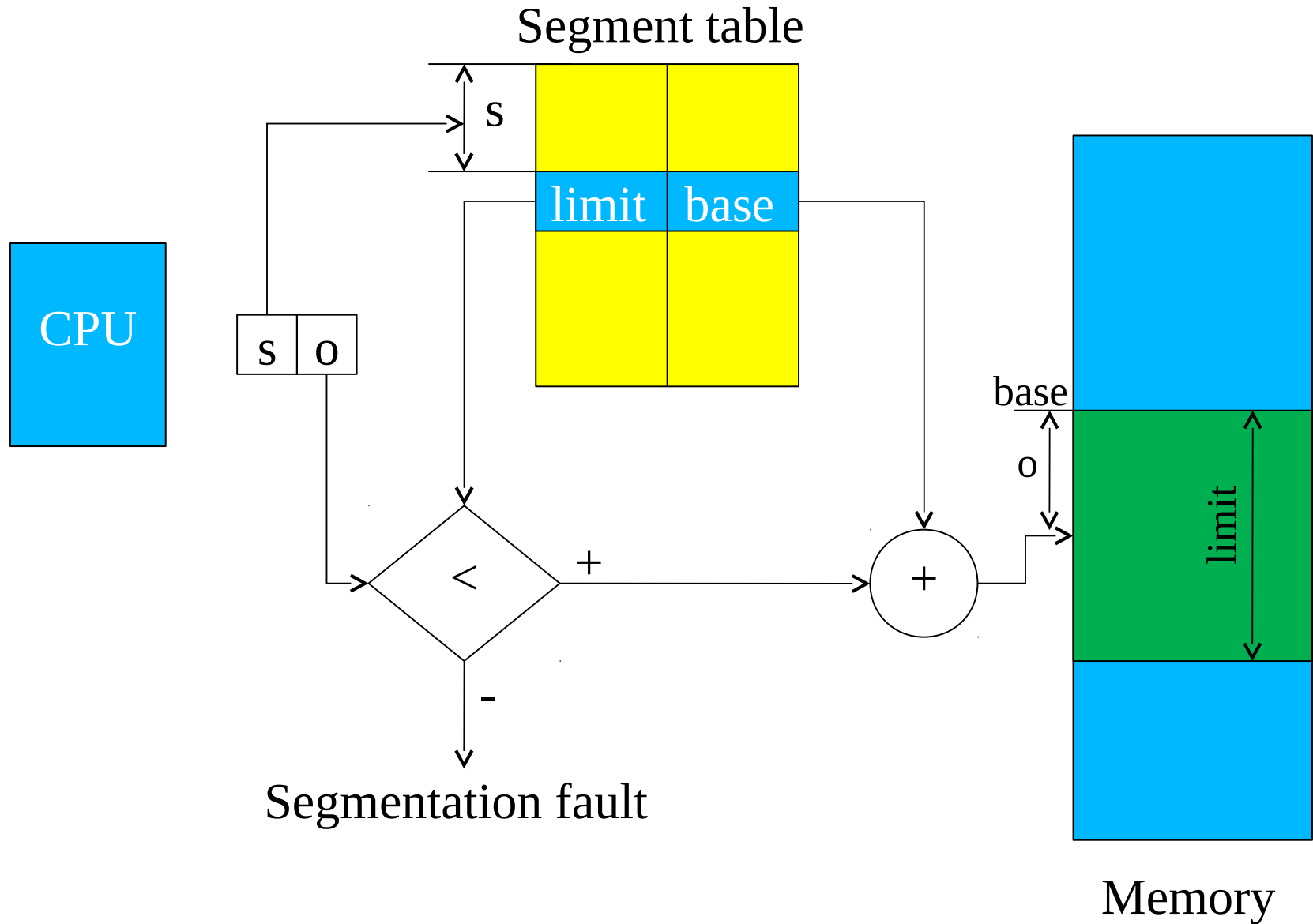


- Segment table – ST

- Function from 2-D (segment, offset) into 1-D (address)
- One item in segment table:
  - **base** – location of segment in physical memory, **limit** – length of segment
- **Segment-table base register (STBR)** – where is ST in memory
- **Segment-table length register (STLR)** – ST size



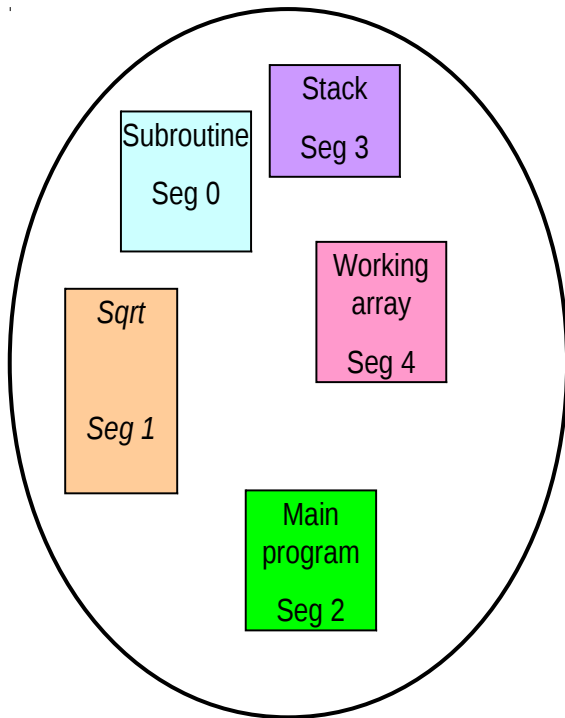
# Hardware support for segmentation



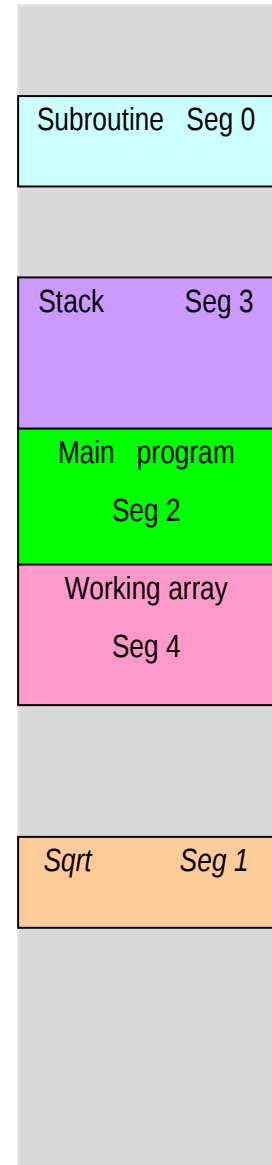
# Segmentation

- **Advantage of the segmentation**
  - Segment has defined length
  - It is possible to detect access outside of the segment. It throws new type of error – segmentation fault
  - It is possible to set access for segment
    - OS has more privilege than user
    - User cannot affect OS
  - It is possible to move data in memory and user cannot detect this shift (change of the segment base is for user invisible)
- **Disadvantage of segmentation**
  - How to place segments into main memory. Segments have different length. Programs are move into memory and release memory.
  - Overhead to compute physical address from virtual address (one comparison, one addition)

# Segmentation example



	limit	base
0	1000	1400
1400	6300	
2400	4300	
31100	3200	
41000	4700	



- It is not easy to place the segment into memory
  - Segments has different size
  - Memory fragmentation
  - Segment moving has big overhead (is not used)

# Paging

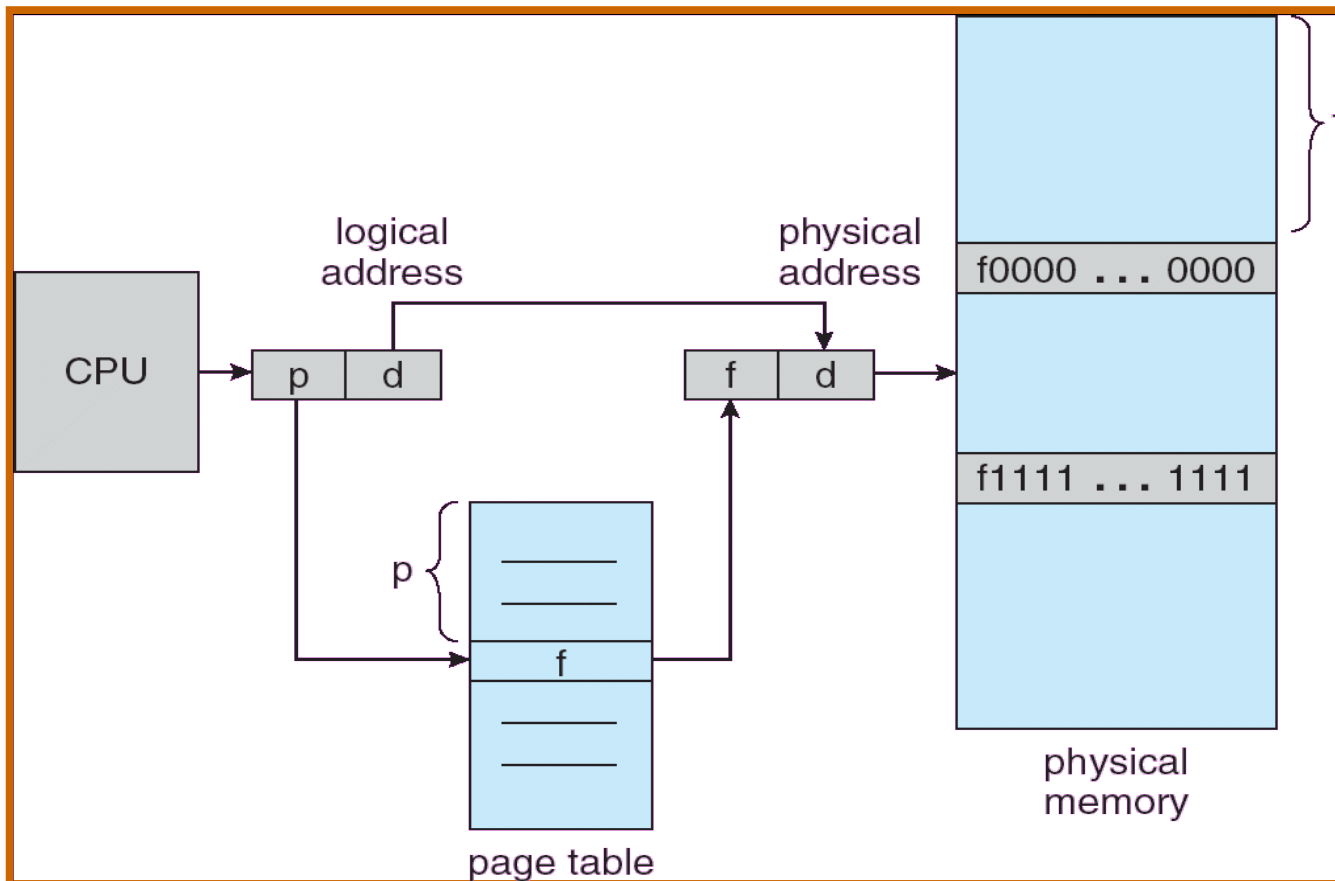
- Different solution for virtual memory implementation
- Paging remove the basic problem of segments – different size
- All pages has the same size that is defined by CPU architecture
- Fragmentation is only inside of the page (small overhead)

# Paging

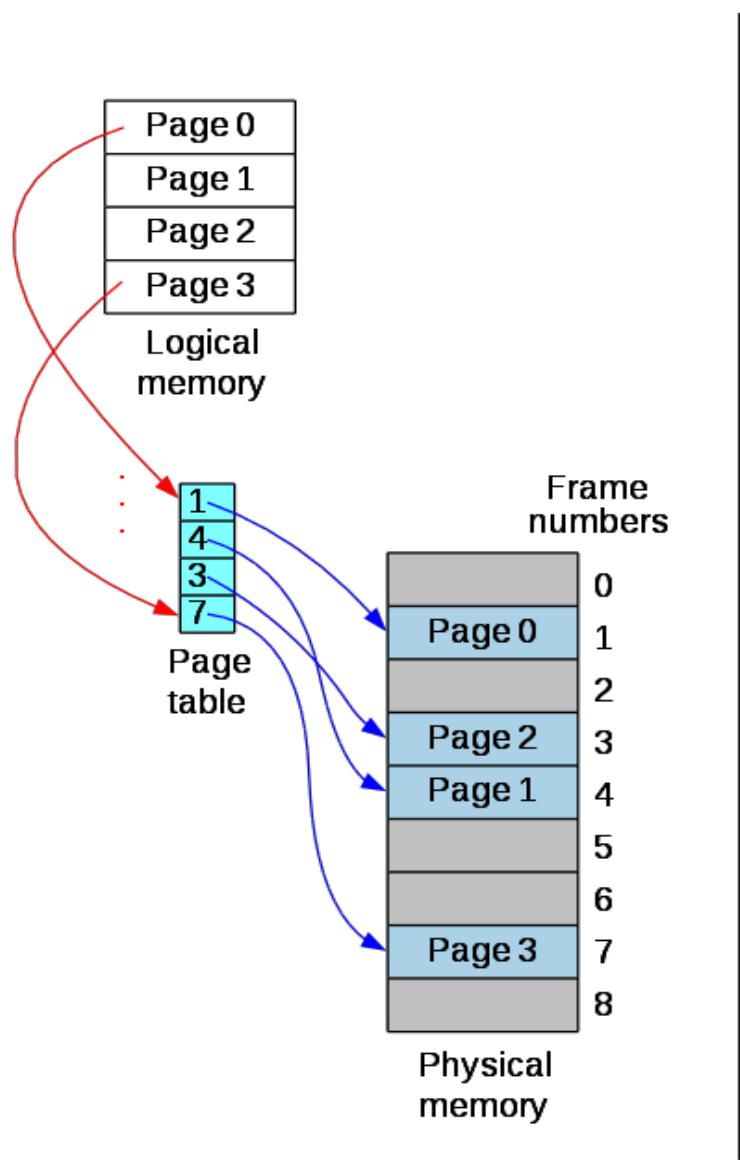
- Contiguous logical address space can be mapped to noncontiguous physical location
  - Each page has its own position in physical memory
- Divide physical memory into fixed-sized blocks called **frames**
  - The size is power of 2 between 512 B, 4 096 B, and 4MiB
- Dived logical memory into blocks with the same size as frames. These blocks are called **pages**
- OS keep track of all frames
- To run process of size  **$n$  pages** need to find  **$n$  free frames**, Transformation from logical address → physical address by
  - **$PT = Page Table$**

# 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



# Paging Examples

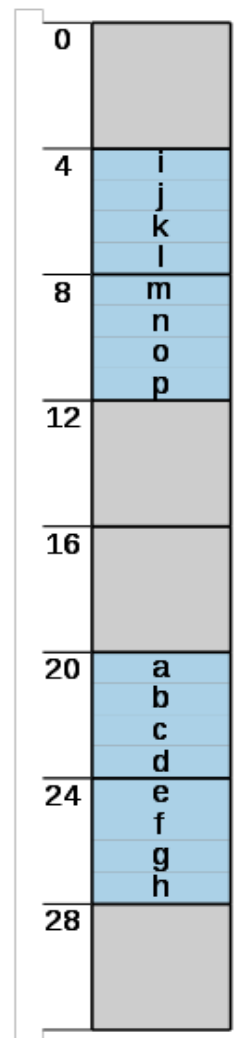


0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

Logical memory

0	5
1	6
2	1
3	2

Page table



Physical memory

# Implementation of Page Table

- Paging is implemented in hardware
- 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)**



# Associative Memory

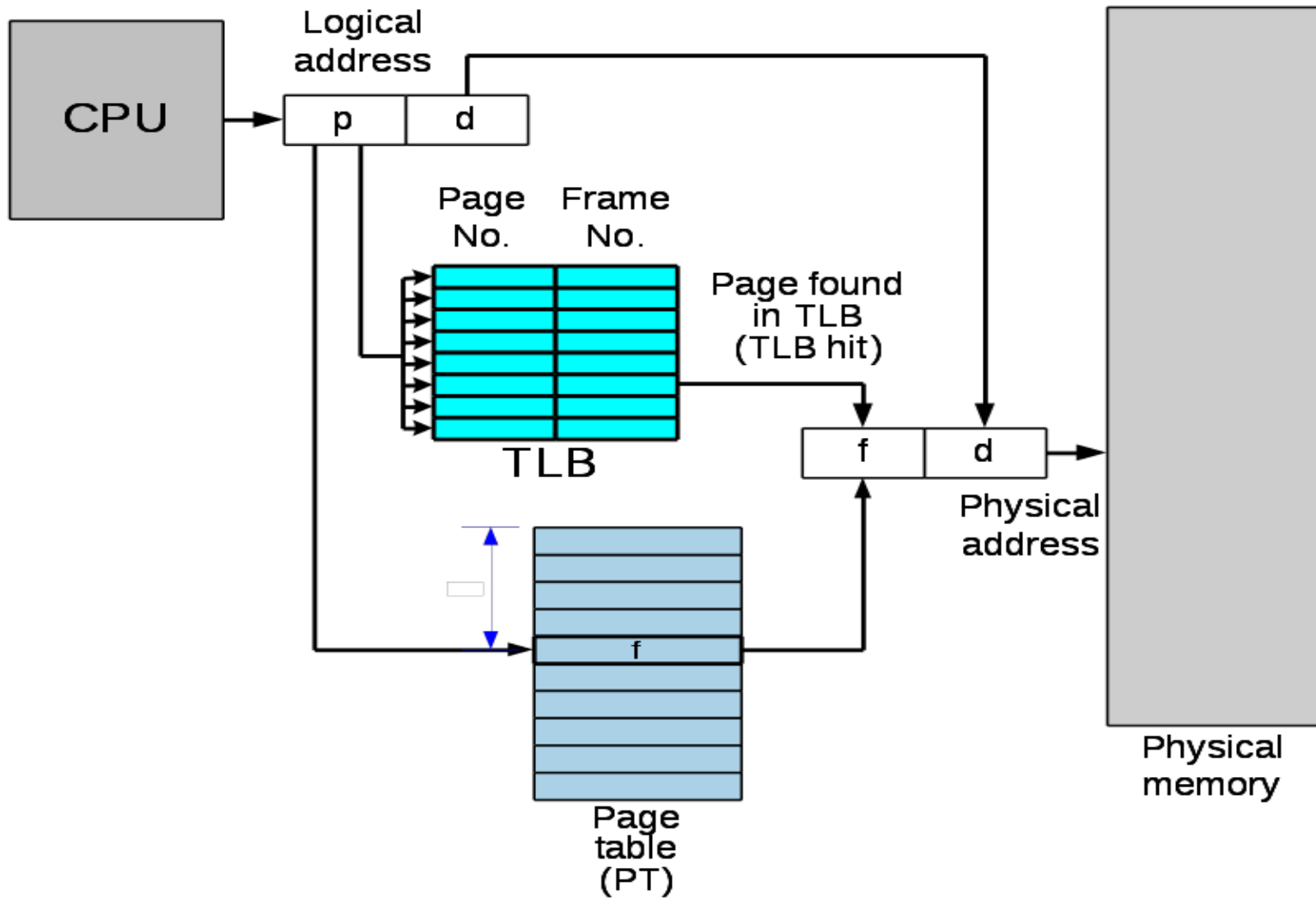
- Associative memory – parallel search – content-addressable memory
- Very fast search

## TBL

Input address	Output address
100000	ABC000
100001	201000
300123	ABC001
100002	300300

- Address translation ( $A'$ ,  $A''$ )
  - If  $A'$  is in associative register, get Frame
  - Otherwise the TBL has no effect, CPU need to look into page table
- **Small TBL can make big improvement**
  - Usually program need only small number of pages in limited time

# Paging Hardware With TLB



# Paging Properties

- Effective Access Time with TLB

- Associative Lookup =  $\epsilon$  time unit
- Assume memory cycle time is  $t = 100$  nanosecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers, Hit ratio =  $\alpha$
- **Effective Access Time (EAT)**

$$EAT = (t + \epsilon) \alpha + (2t + \epsilon)(1 - \alpha) = (2 - \alpha)t + \epsilon$$

Example for  $t = 100$  ns

<i>PT</i> without <i>TLB</i>		<i>EAT</i> = 200 ns	Need two access to memory
$\epsilon = 20$ ns	$\alpha = 60$ %	<i>EAT</i> = 160 ns	<i>TLB</i> increase significantly access time
$\epsilon = 20$ ns	$\alpha = 80$ %	<i>EAT</i> = 140 ns	
$\epsilon = 20$ ns	$\alpha = 98$ %	<i>EAT</i> = 122 ns	

# TLB

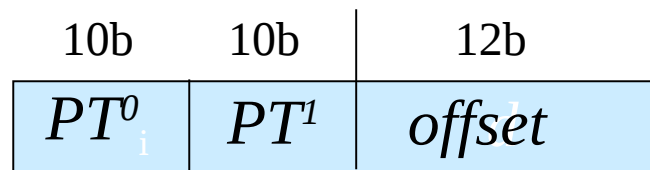
- Typical TLB
  - Size 8-4096 entries
  - Hit time 0.5-1 clock cycle
  - PT access time 10-100 clock cycles
  - Hit ration 99%-99.99%
- Problem with context switch
  - Another process needs another pages
  - With context switch invalidates TBL entries (free TLB)
- OS takes care about TLB
  - Remove old entries
  - Add new entries

# Page table structure

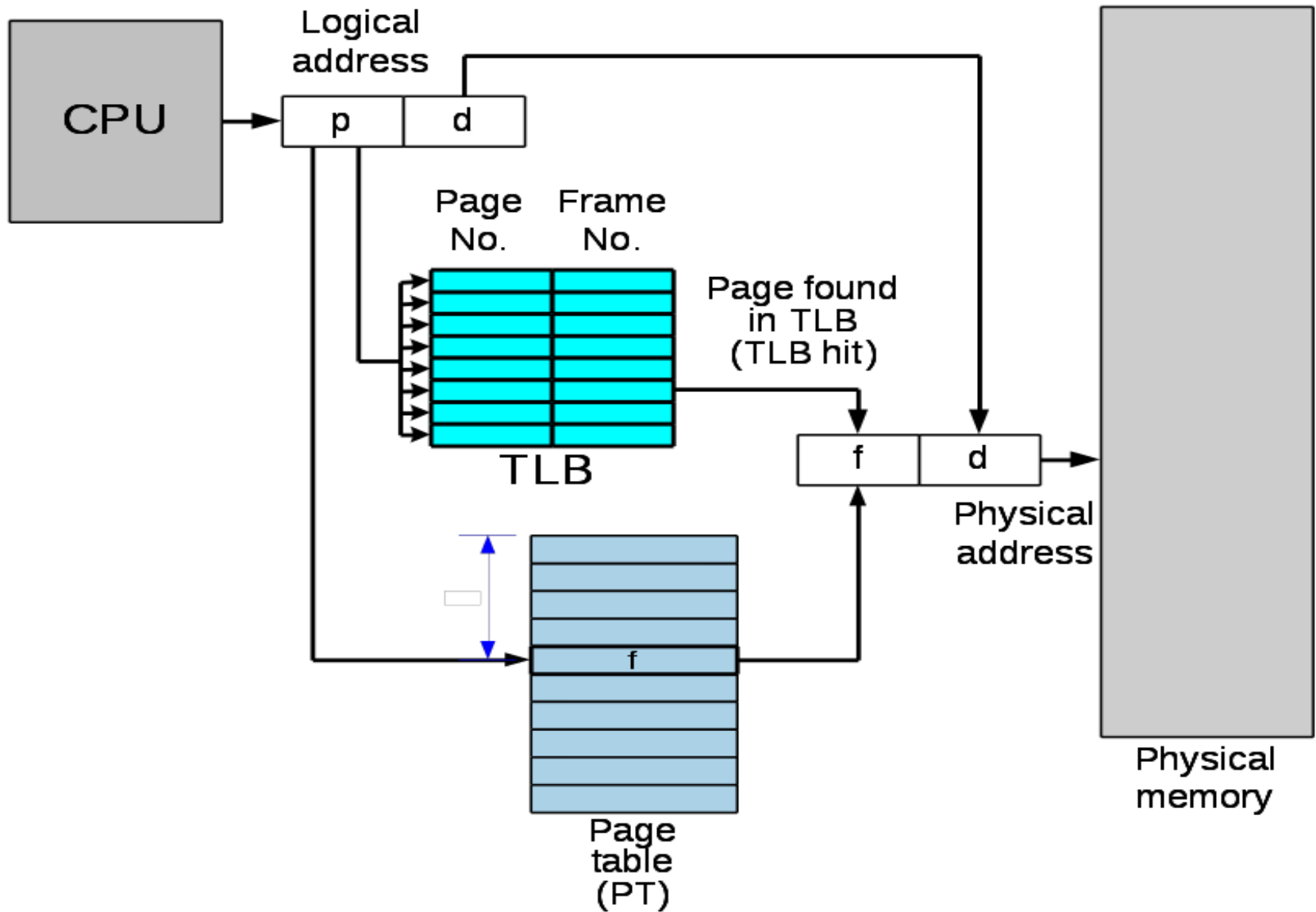
- Problem with PT size
  - Each process can have it's own PT
  - 32-bits logical address with page size 4 KB → PT has 4 MB
    - PT must be in memory
- Hierarchical PT
  - Translation is used by PT hierarchy
  - Usually 32-bits logical address has 2 level PT
  - $PT^0$  contains reference to  $PT^1$
  - ,Real page table  $PT^1$  can be paged need not to be in memory
- Hash PT
  - Address  $p$  is used by hash function  $hash(p)$
- Inverted PT
  - One PT for all process
  - Items depend on physical memory size
  - Hash function has address  $p$  and process pid  $hash(pid, p)$

# Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
  - A logical address (on 32-bit machine with 4K page size) is divided into:
    - a page number consisting of 20 bits
    - a page offset consisting of 12 bits
  - Since the page table is paged, the page number is further divided into:
    - a 10-bit page number
    - a 10-bit page offset
  - Thus, a logical address is as follows:



# Two-Level Page-Table Scheme

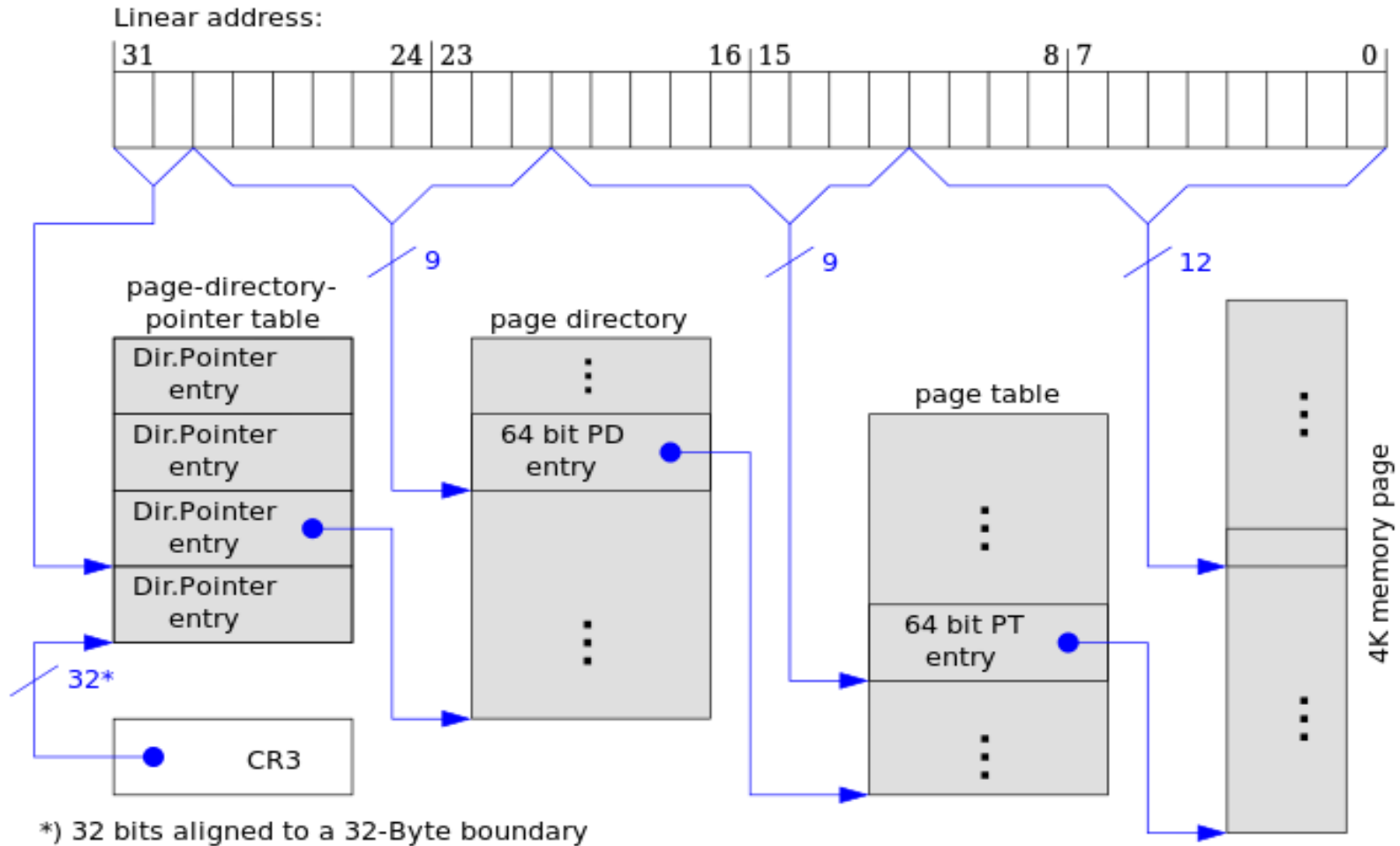


# PAE

- Price of 8GB RAM is low but you cannot use this memory with 32-bit system. Solution 64-bit system or PAE
- Physical Address Extension = PAE
- Using PAE you change 32-bit address space to 36-bit address space, it can address 64 GB RAM
- Change of page table:
  - Page table translate 20bits of page number to 24bits of frame number
  - Page table size is increased twice, because there was no space for additional 4 bits
  - Maximal linear size for one process is still 4GB
  - 2 processes can use 8GB
  - Intel change 2level page table into 3 level to keep smaller size of PT



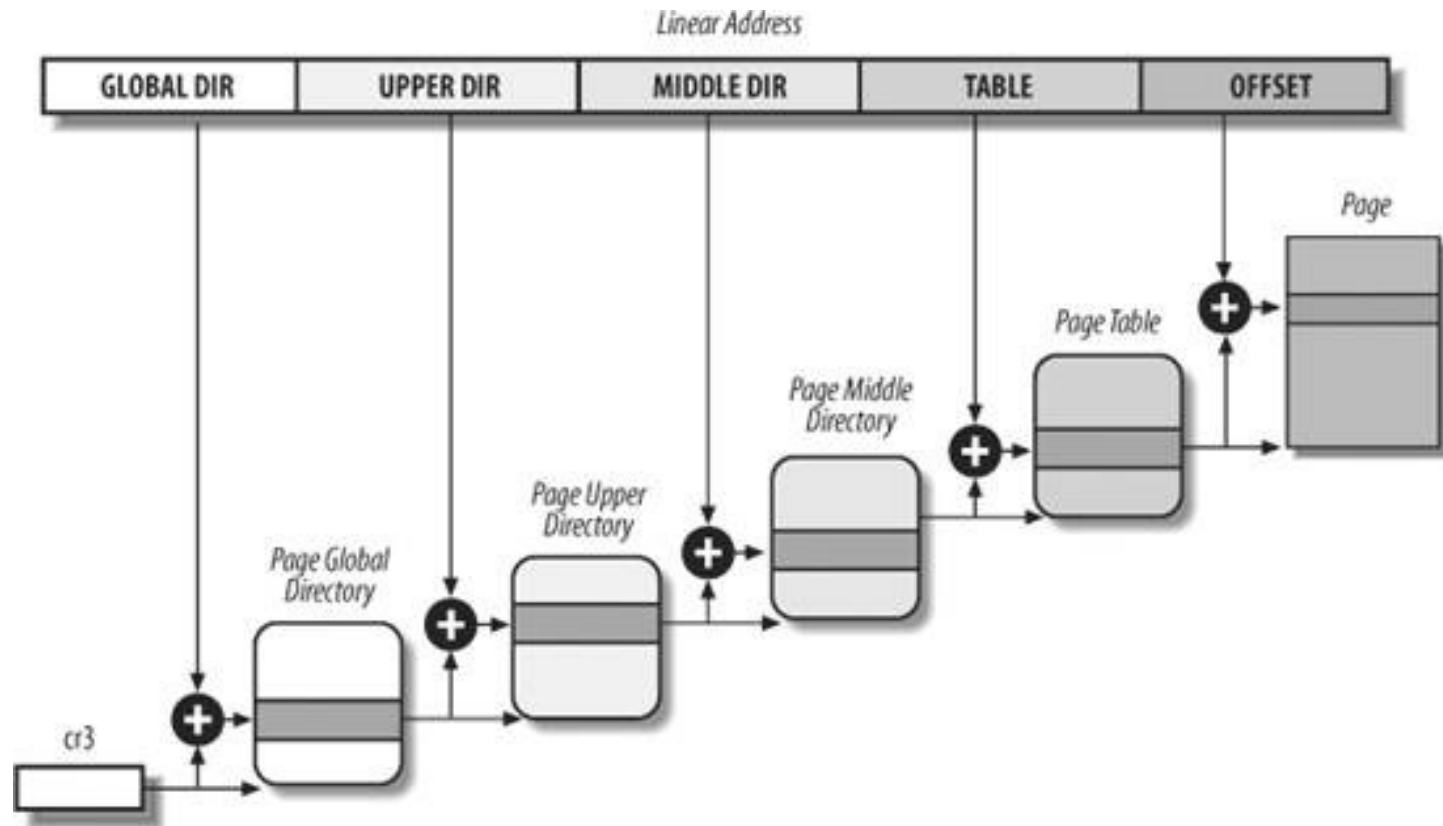
# PAE Intel



# Hierarchical PT

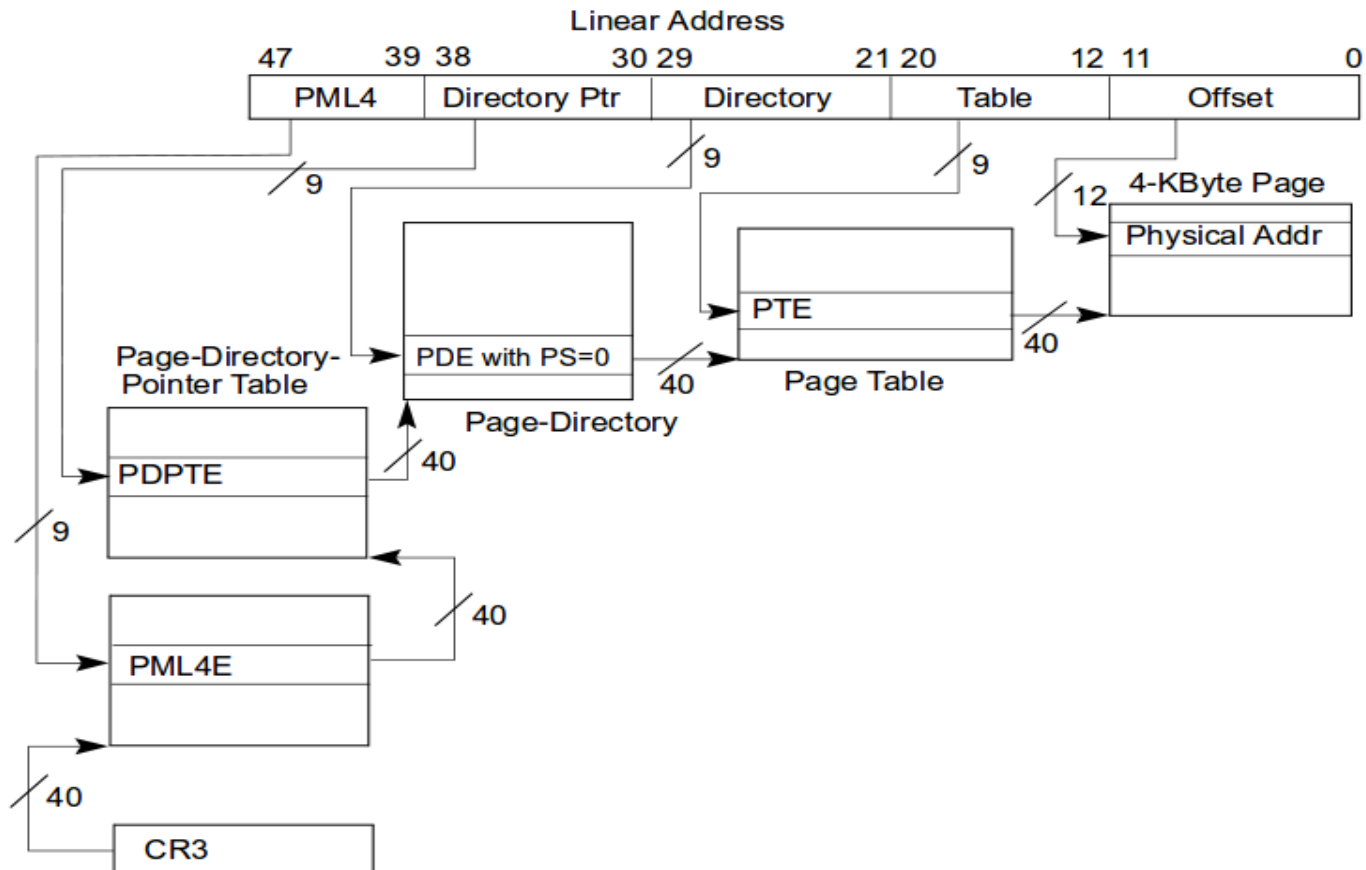
- 64-bits address space with page size 4 KB
  - 52 bits page number → 4 Peta (4096 Tera) Byte PT
- It is problem for hierarchical PT too:
  - Each level brings new delay and overhead, 7 levels will be very slow
- UltraSparc – 64 bits ~ 7 level → wrong
- Linux – 64 bits (Windows similar)
  - Trick: logical address uses only 48 bits, other bits are ignored
  - Logical address space has only 256 TiB
  - 4 level by 9 bits of address
  - 12 bits offset inside page
  - It is useful solution

# AMD x86-64 paging mode



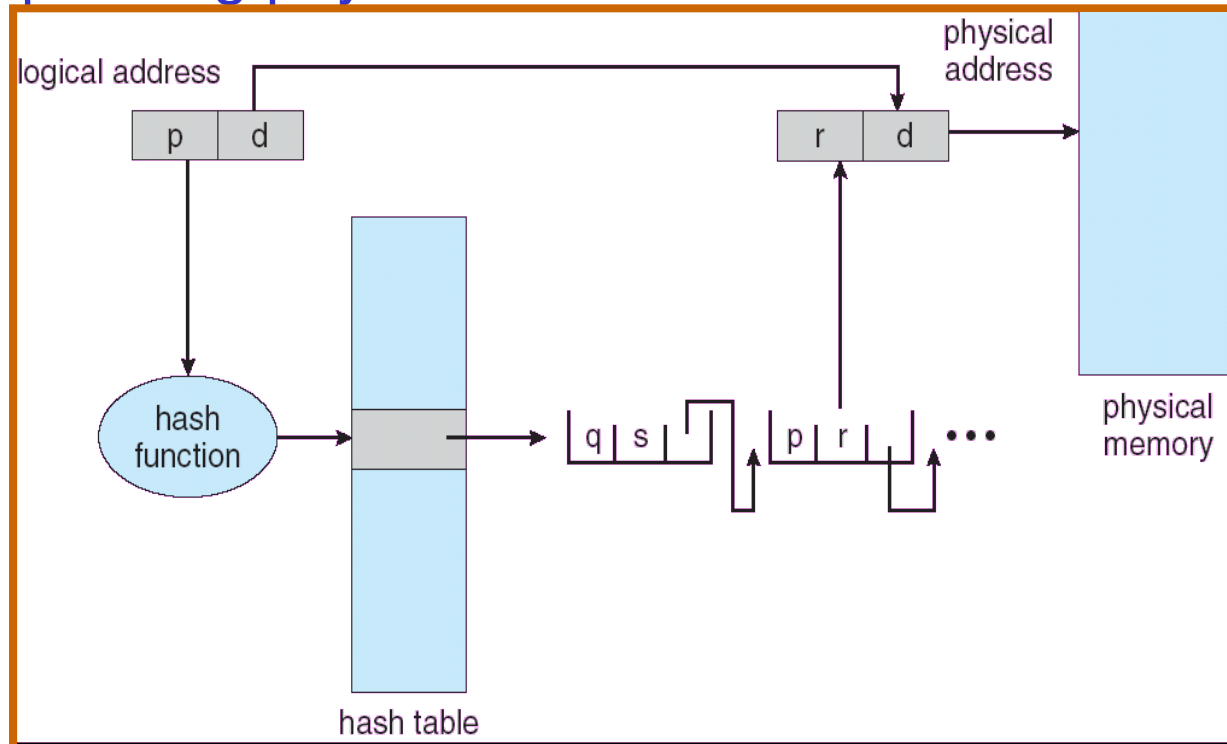
# Paging IA-32e

- Linear address 48 bits
- Physical address 52 bits – it is 4 PiB RAM
- Variants with 2MiB or 1GiB page size



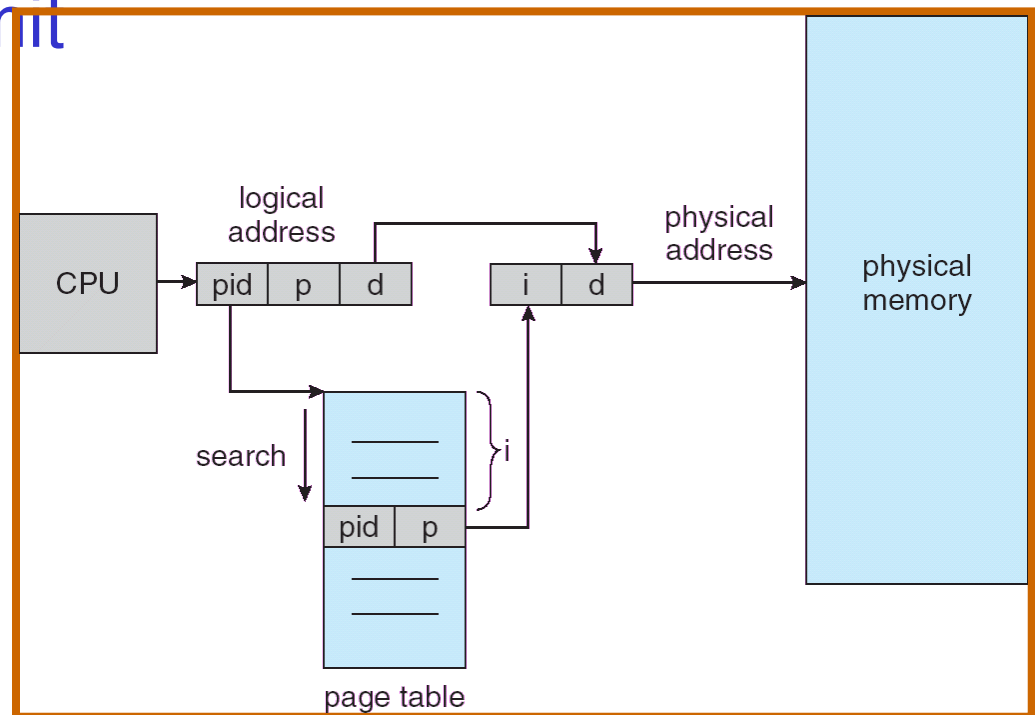
# Hashed Page Tables

- 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.



# 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



# 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

# Segmentation with paging

- Combination of both methods
- Keeps advantages of segmentation, mainly precise limitation of memory space
- Simplifies placing of segments into virtual memory. Memory fragmentation is limited to page size.
- Segmentation table ST can contain
  - address of page table for this segment PT
  - Or linear address this address is used as virtual address for paging



# Segmentation with paging

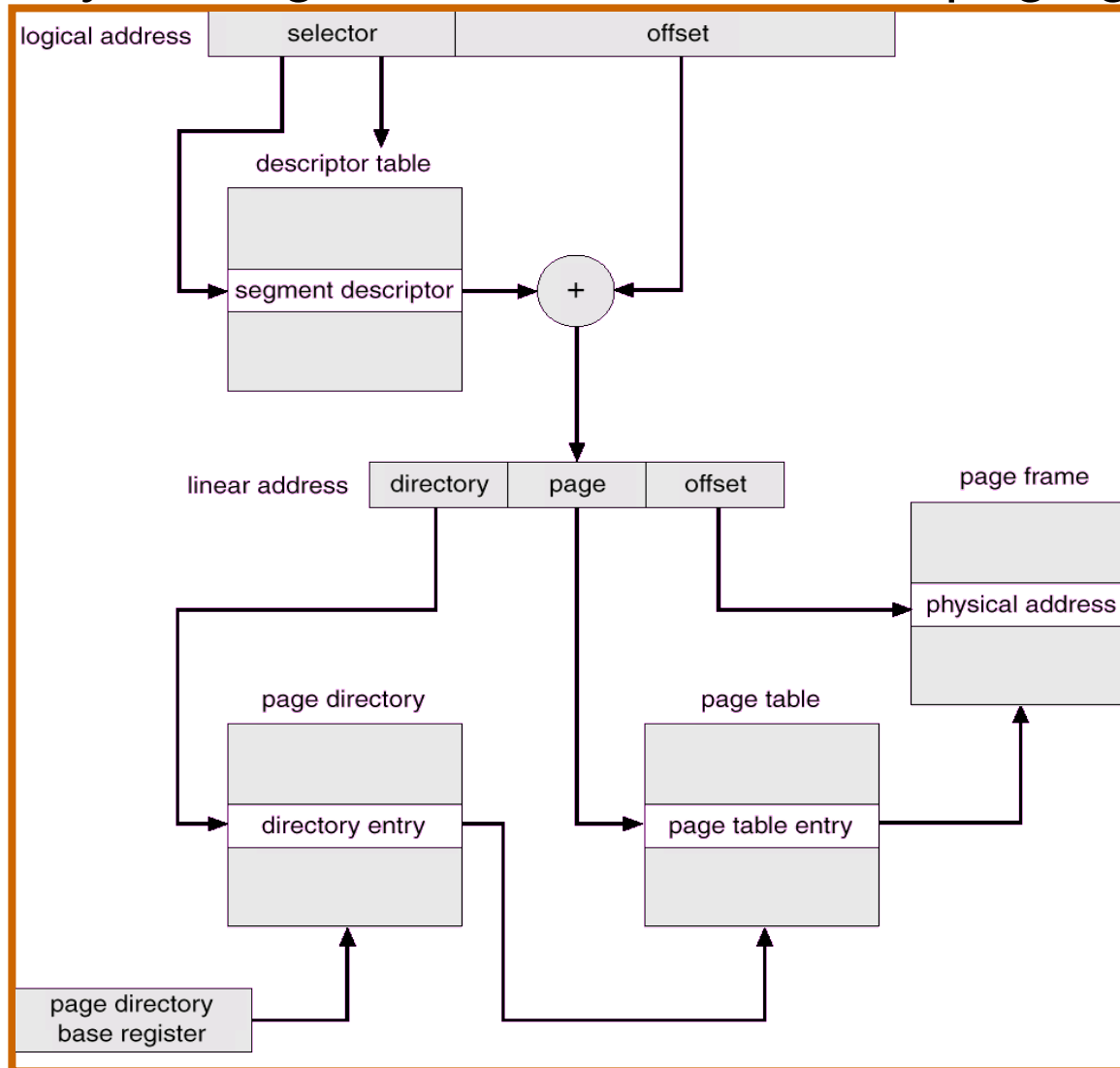
- Segmentation with paging is supported by architecture IA-32 (e.g. INTEL-Pentium)
- IA-32 transformation from logical address space to physical address space with different modes:
  - **logical linear space** (4 GB), transformation identity
    - Used only by drivers and OS
  - **logical linear space** (4 GB), paging,
    - 1024 oblastí à 4 MB, délka stránky 4 KB, 1024 tabulek stránek, každá tabulka stránek má 1024 řádků
    - Používají implementace UNIX na INTEL-Pentium
  - **logical 2D address (segemnt, offset)**, segmentation
    - $2^{16} = 16384$  of segments each 4 GB ~ 64 TB
  - **logical 2D address (segemnt, offset)**, segmentatation with paging
    - Segments select part of linear space, this linear space uses paging
    - Used by windows and linux

# Segmentation with paging IA-32

- 16 K of segments with maximal size 4 GB for each segment
- 2 logic subspaces (descriptor TI = 0 / 1)
  - 8 K private segments – Local Description Table, LDT
  - 8 K shared segments – Global Description Table, GDT
- Logic address = (segment descriptor, offset)
  - offset = 32-bits address with paging
  - Segment descriptor
    - 13 bits segment number,
    - 1 bit *descriptorTI*,
    - 2 bits Privilege levels : OS kernel, ... ,application
    - Rights for *r/w/e* at page level
- Linear address space inside segment with hierarchical page table with 2 levels
  - Page size 4 KB, offset inside page 12 bits,
  - Page number 2x10 bits

# Segmentation with Paging – Intel 386

- IA32 architecture uses segmentation with paging for memory management with a two-level paging scheme



# Linux on Intel 80x86

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