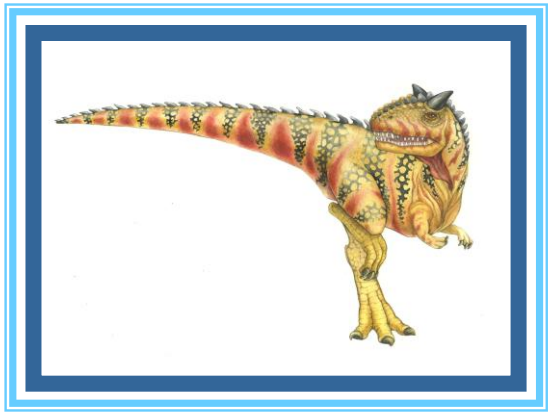
Chapter 9: Main Memory



Chapter 9: Memory Management

Background

Contiguous Memory Allocation

Paging

Structure of the Page Table

Swapping

Example: The Intel 32 and 64-bit Architectures

Example: ARMv8 Architecture

Objectives

To provide a detailed description of various ways of organizing memory hardware

To discuss various memory-management techniques,

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

register > cache > main memory

Main memory and registers + cache are only storage CPU can access directly

Memory unit only sees a stream of:

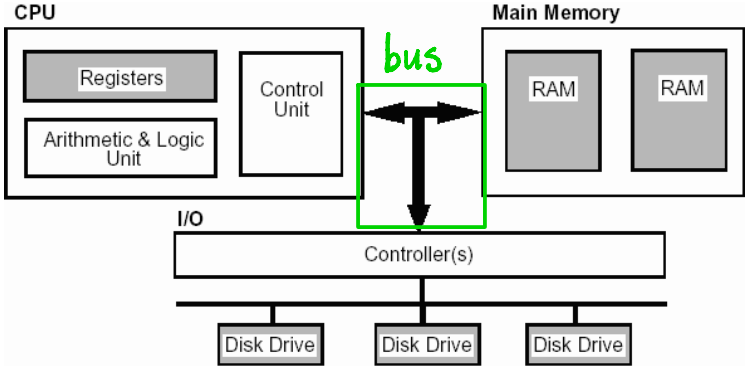
- addresses + read requests, or
- address + data and write requests

Register access is done in one CPU clock (or less)

Main memory can take many cycles, causing a **stall** (waiting for response)

Cache sits between main memory and CPU registers

Protection of memory required to ensure correct operation ⇒ generate exception (kind of interrupt) when processes try to access other process' content



movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	Imm (immediate)	Reg	movq $0x4, %rax	temp = 0x4;
		Mem	movq $-147, (%rax)	*p = -147;
	Reg (register)	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem (memory)	Reg	movq (%rax), %rdx	temp = *p;

register value itself

value of address which is pointed by register

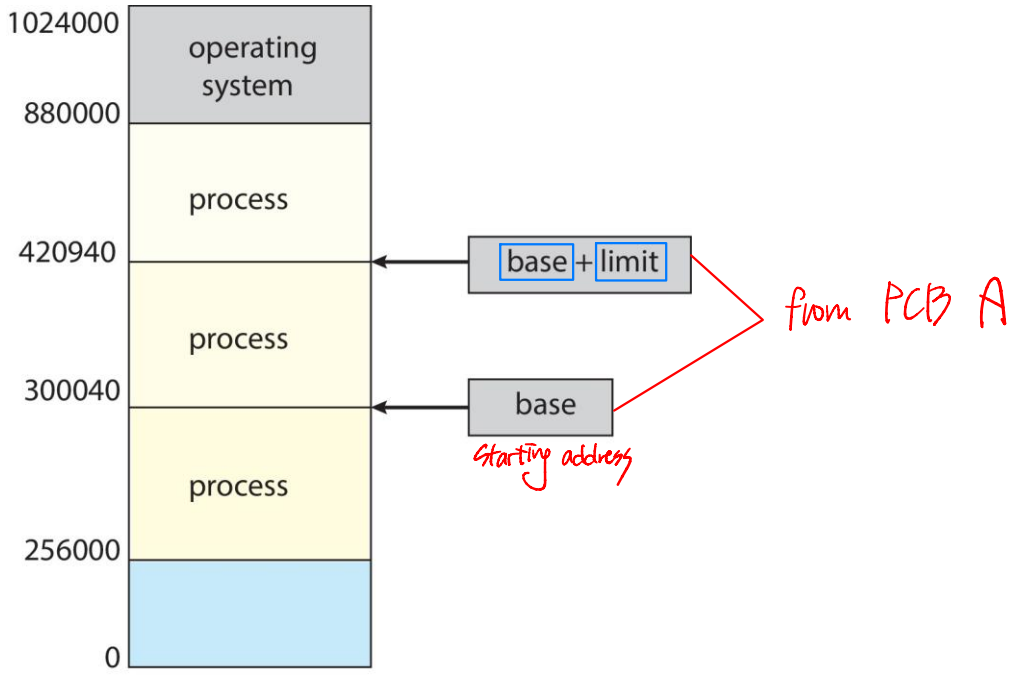
⇒ movq (%rax) %rcx
movq %rcx (%rdx)

Cannot do memory-memory transfer with a single instruction

Protection

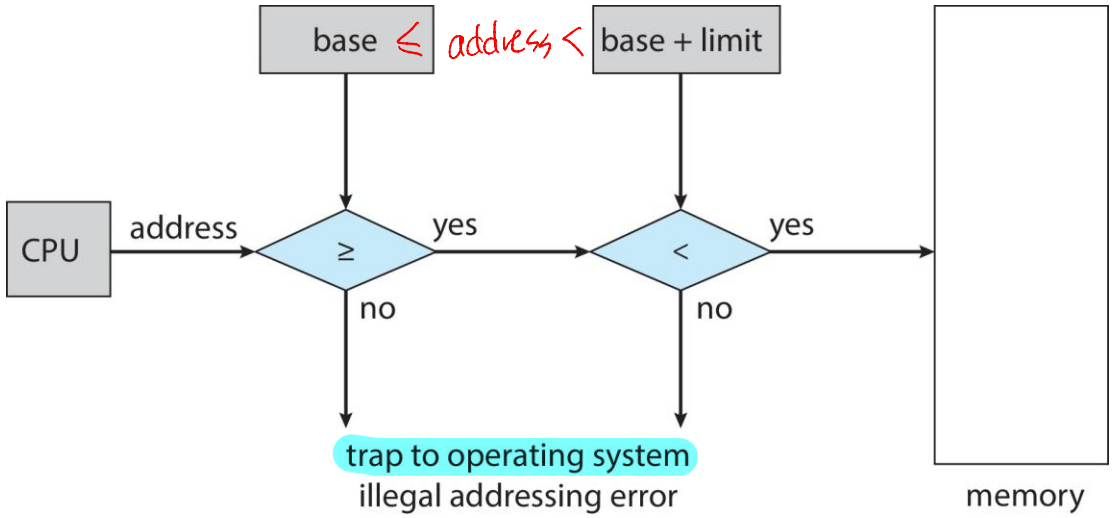
Need to ensure that a process can only access those addresses in it address space. by OS

We can provide this protection by using a pair of **base** and **limit** **registers** define the logical address space of a process



Hardware Address Protection

CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



the instructions to loading the base and limit registers are privileged

Address Binding

Programs on disk, ready to be brought into memory and placed within the context of a process → eligible for execution on an available CPU

Without support, must be loaded into address 0000 ⇒ in multi-process, collision between processes use same space

Inconvenient to have first user process physical address always at 0000

How can it not be? Address Binding

Addresses represented in different ways at different stages of a program's life

Source code addresses usually symbolic e.g. `int i;`

compiler:
- Compiled code addresses **bind** to relocatable addresses
  - i.e. "14 bytes from beginning of this module"

linker / loader:
- Linker or loader will bind relocatable addresses to absolute addresses
  - i.e. 74014

Each binding maps one address space to another

actual address used by program is not determined until it is loaded in MEM.

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages

1

symbolic → relocatable code

Compile time: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes

option ①

⇒ single-process environment only

2

relocatable code → logical address

Load time: Must generate **relocatable code** if memory location is not known at compile time

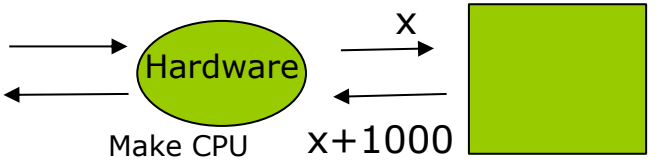
⇒ X is determined in load time

3

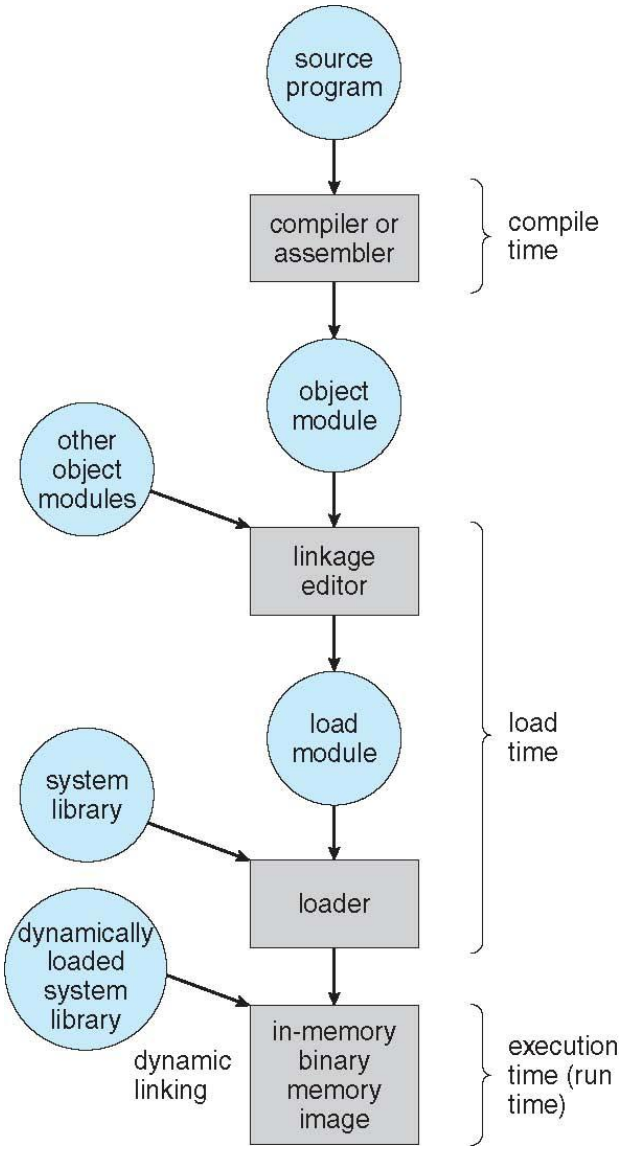
logical address → physical address

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)



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

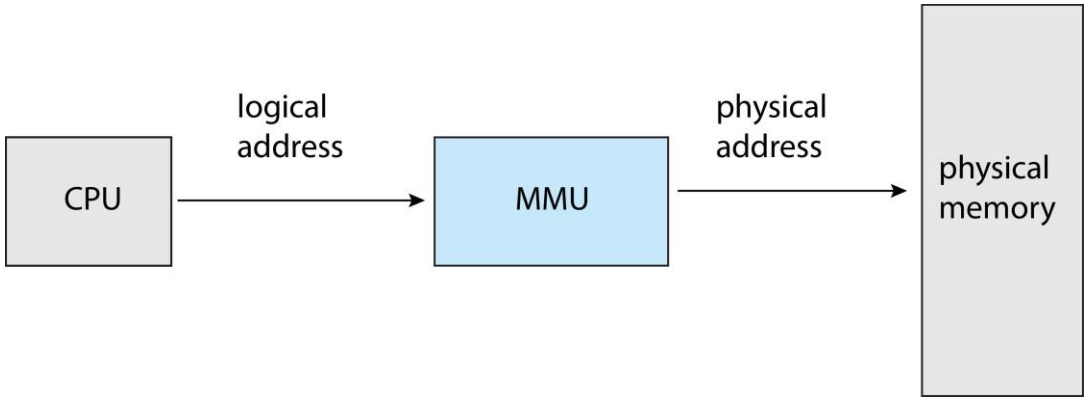
Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Logical address space is the set of all logical addresses generated by a program

Physical address space is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

Hardware device that at run time maps virtual to physical address



Many methods possible, covered in the rest of this chapter

contiguous allocation
paging

Memory-Management Unit (Cont.)

Consider simple scheme. which is a generalization of the base-register scheme.

The base register now called **relocation register**

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

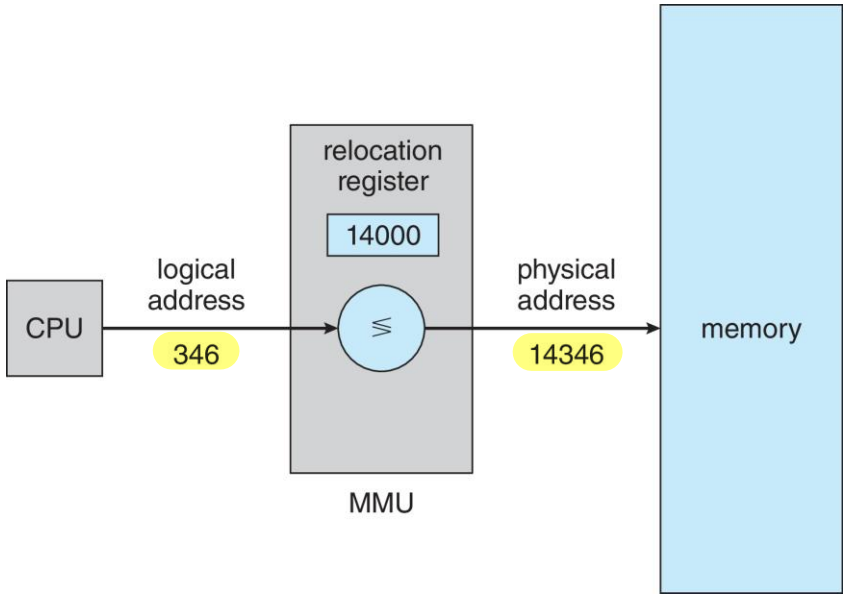
- Execution-time binding occurs when reference is made to location in memory
- Logical address bound to physical addresses

Memory-Management Unit (Cont.)

Consider simple scheme. which is a generalization of the base-register scheme.

The base register now called **relocation register**

The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Dynamic Loading

The entire program does not need to be in memory to execute

Routine is not loaded until it is called

Better memory-space utilization; unused routine is never loaded

All routines kept on disk in relocatable load format

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
- OS can help by providing libraries to implement dynamic loading

Dynamic Linking

Static linking – system libraries and program code combined by the loader into the binary program image ⇒ redundant code, version changing depending on env big size code

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 checks if routine is in processes' memory address
  - If not in address space, add to address space

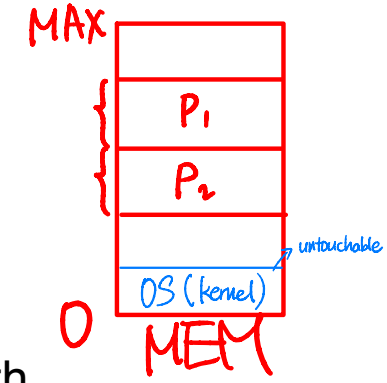
Dynamic linking is particularly useful for libraries

System also known as **shared libraries**

Consider applicability to patching system libraries

- Versioning may be needed

Contiguous Allocation



Main memory must support both OS and user processes

Limited resource, must allocate efficiently

Contiguous allocation is one early method

Main memory usually into two **partitions**:

- Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory
- Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

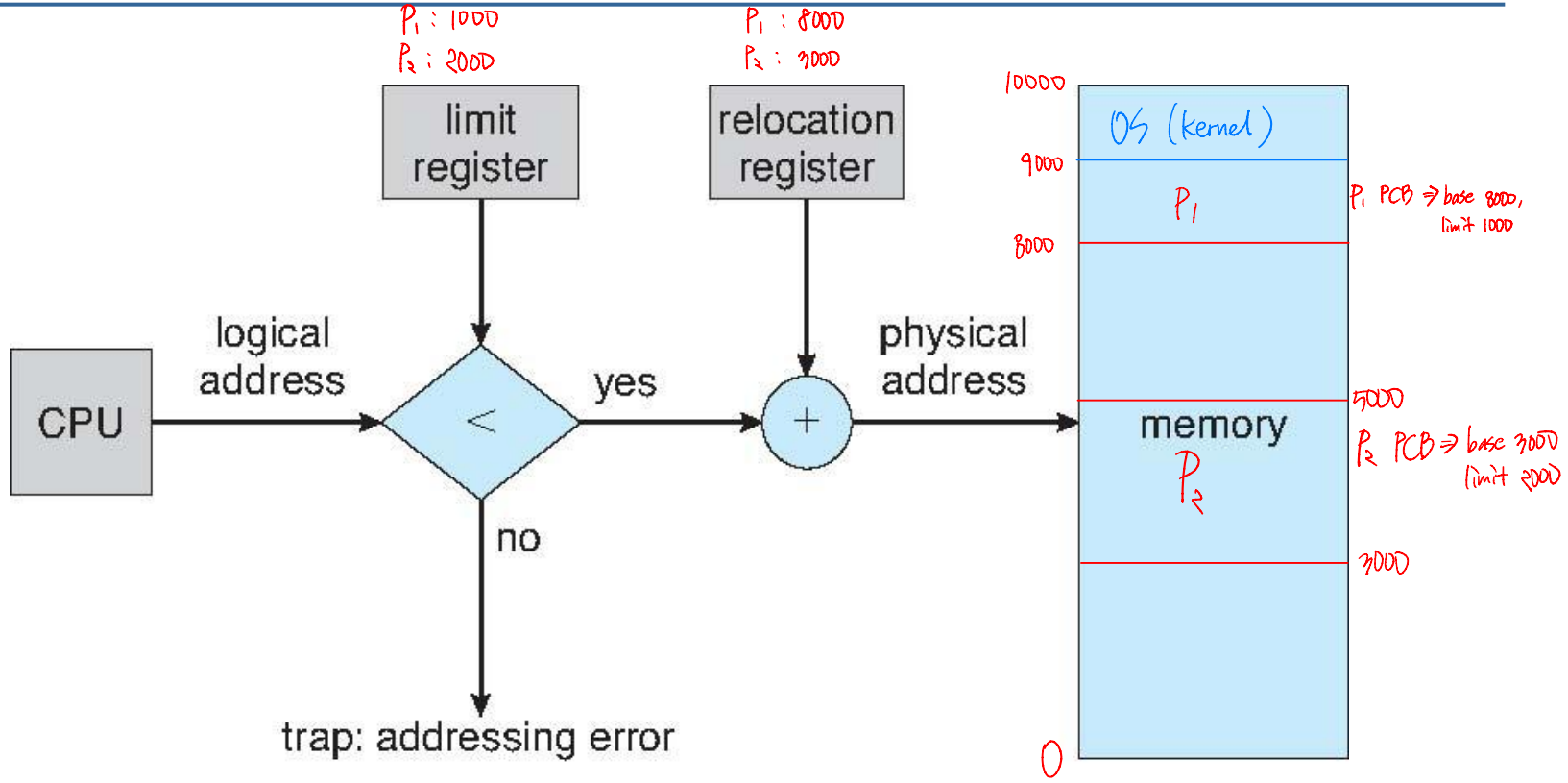
Relocation registers used to protect user processes from each other, and from changing operating-system code and data

- 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

value from PCB

MMU maps logical address *dynamically*

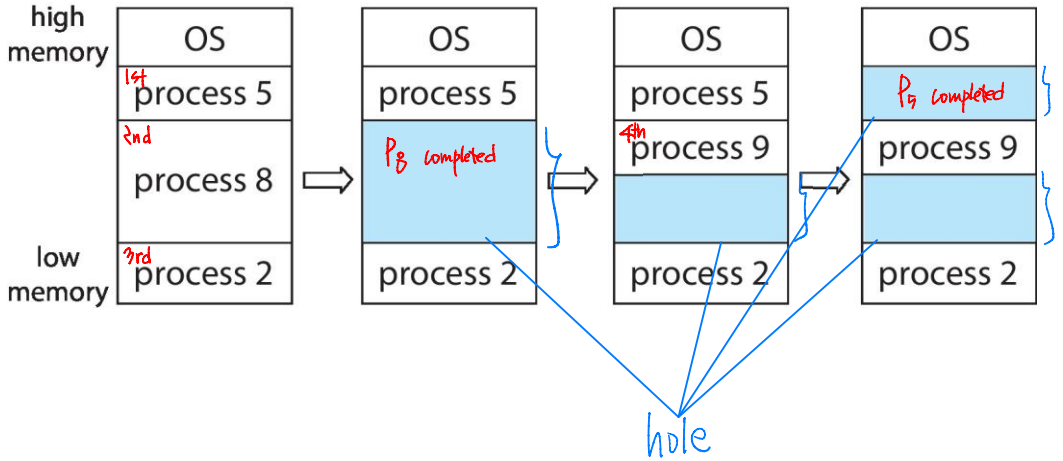
Hardware Support for Relocation and Limit Registers



Variable Partition

Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **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
- Process exiting frees its partition, adjacent free partitions combined
- 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

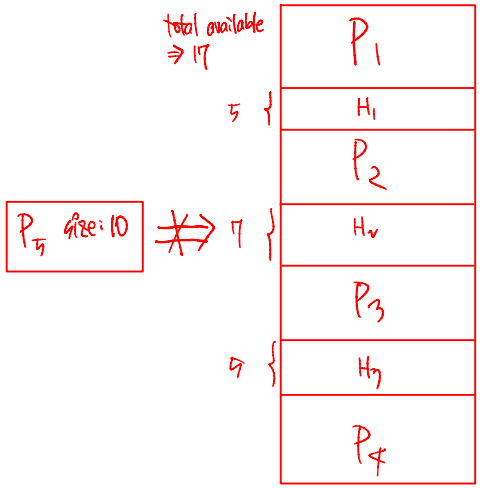
External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous ⇒ contiguous allocation

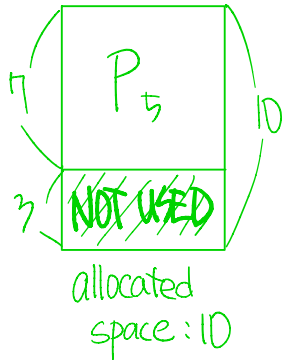
Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used ⇒ contiguous allocation / paging

First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation

contiguous allocation not widely used due to MEM utilization

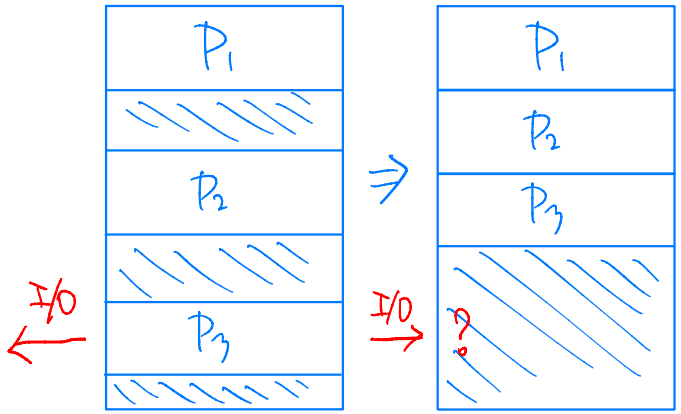
0.5/1.5 1/3 may be unusable -> **50-percent rule**





Fragmentation (Cont.)

- 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 ··· inefficient
- Now consider that backing store has same fragmentation problems



Paging

Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

- Avoids external fragmentation
- Avoids problem of varying sized memory chunks

Divide physical memory into fixed-sized blocks called **frames**

- Size is power of 2, between 512 bytes and 16 Mbytes

Divide logical memory into blocks of same size called **pages**

same size

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

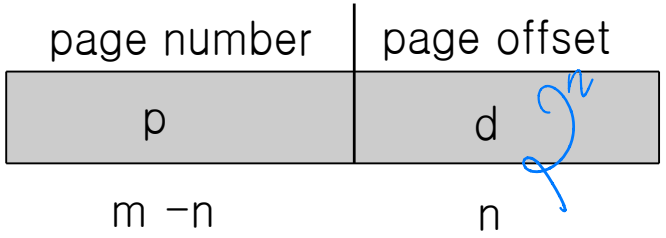
Backing store likewise split into pages ⇒ such as HDD

Still have Internal fragmentation

Address Translation Scheme

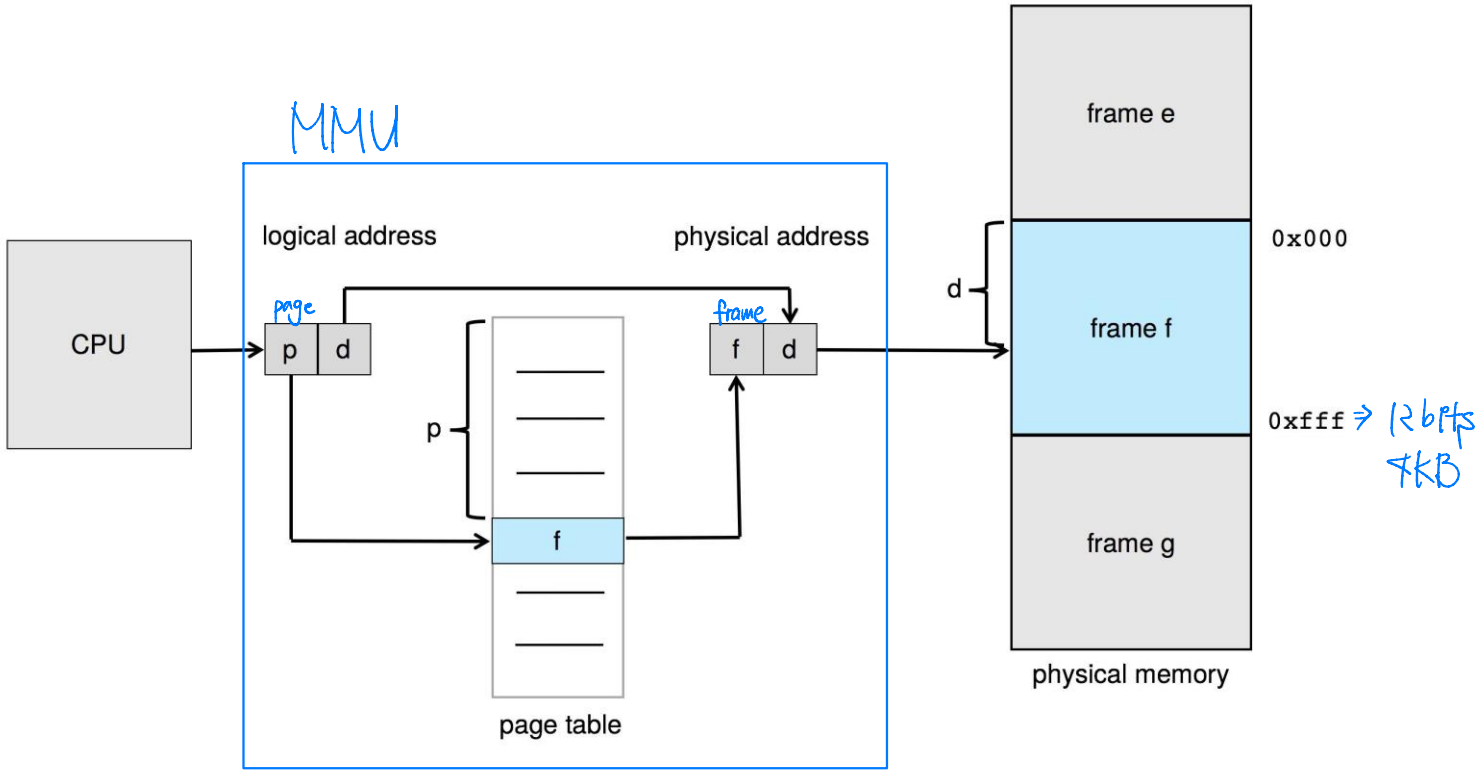
Address generated by CPU is divided into:

- **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

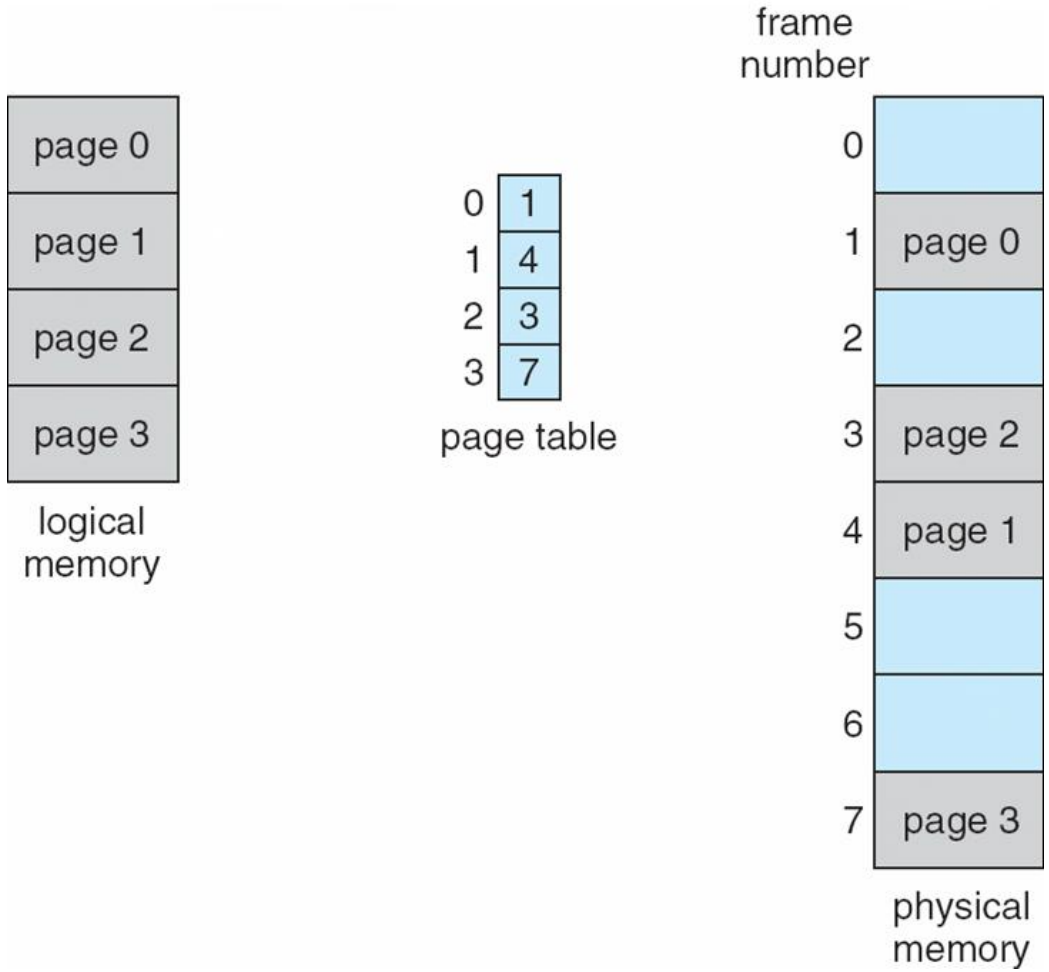


For given logical address space 2^m and page size 2^n

Paging Hardware

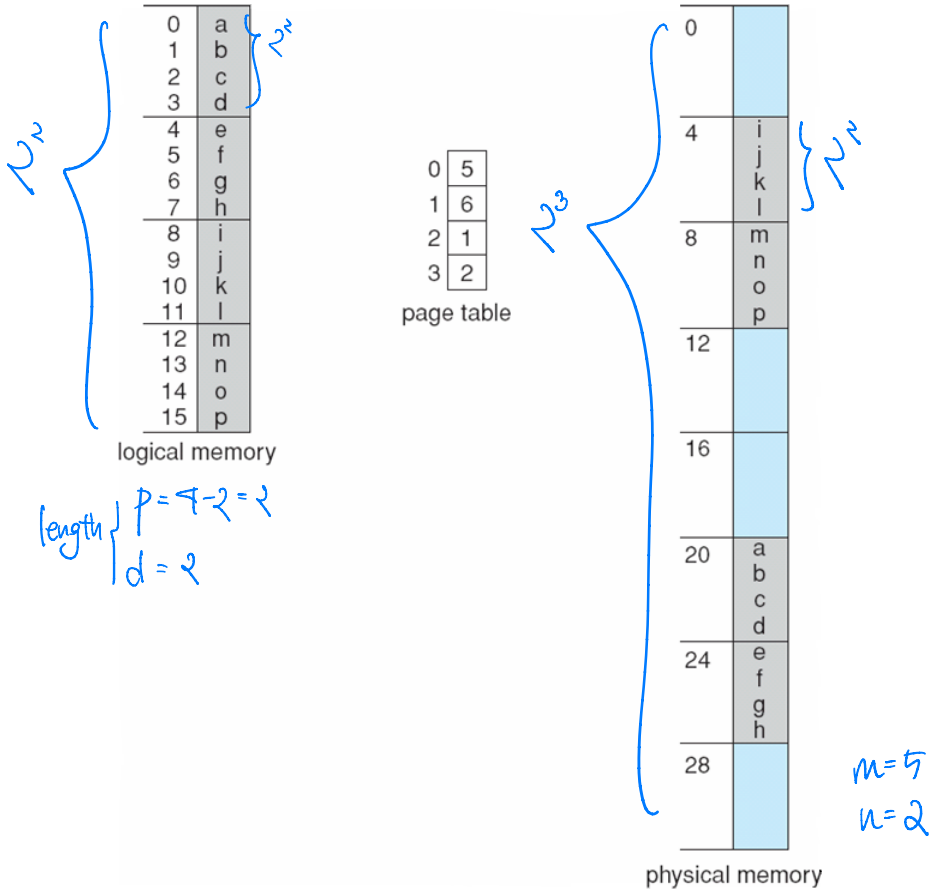


Paging Model of Logical and Physical Memory



Paging Example

Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



Paging -- Calculating internal fragmentation

Page size = 2,048 bytes

Process size = 72,766 bytes

35 pages + 1,086 bytes ⇒ ½ pages needed

Internal fragmentation of 2,048 - 1,086 = 962 bytes

Worst case fragmentation = 1 frame – 1 byte

On average fragmentation = 1 / 2 frame size

trade-off

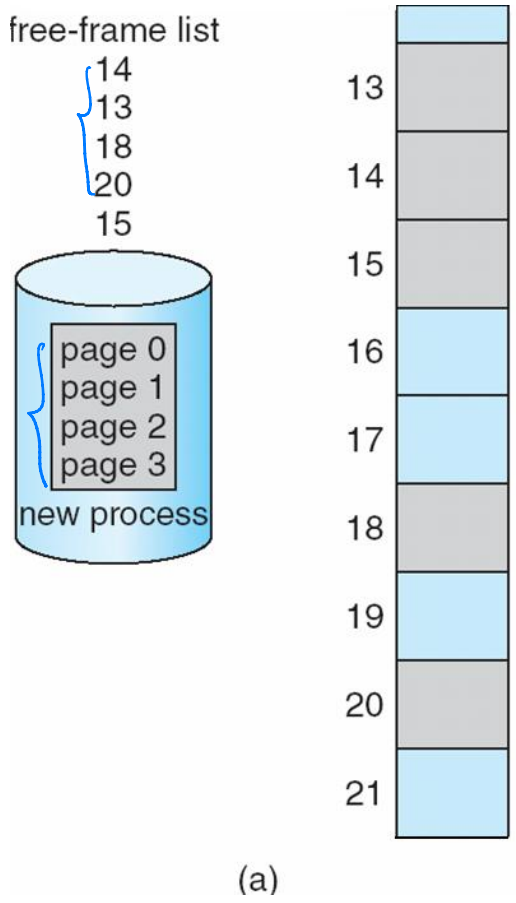
So small frame sizes desirable? frame size ↓

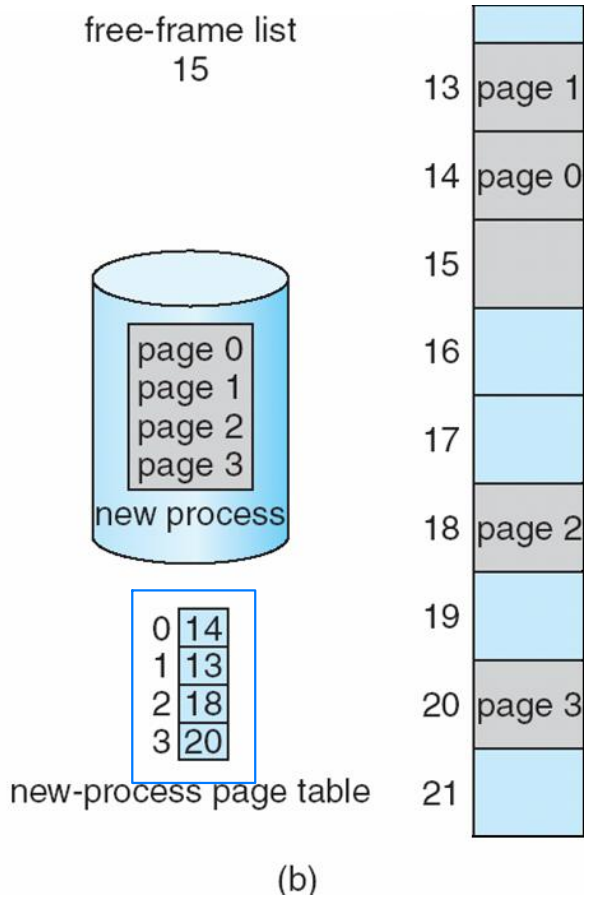
But each page table entry takes memory to track page table size ↑

Page sizes growing over time

Solaris supports two page sizes – 8 KB and 4 MB

Free Frames





Before allocation

After allocation

Implementation of Page Table

Page table is kept in main memory

updated at every context switching

- **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 ① page table and one for the ② data / instruction

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

all entries are searched in parallel

⇒ O(1) ~~O(n)~~

Translation Look-Aside Buffer

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

- Otherwise need to flush at every context switch

TLBs typically small (64 to 1,024 entries)

On a TLB miss, value is loaded into the TLB for faster access next time

- Replacement policies must be considered
- Some entries can be **wired down** for permanent fast access

Hardware

Associative memory – parallel search

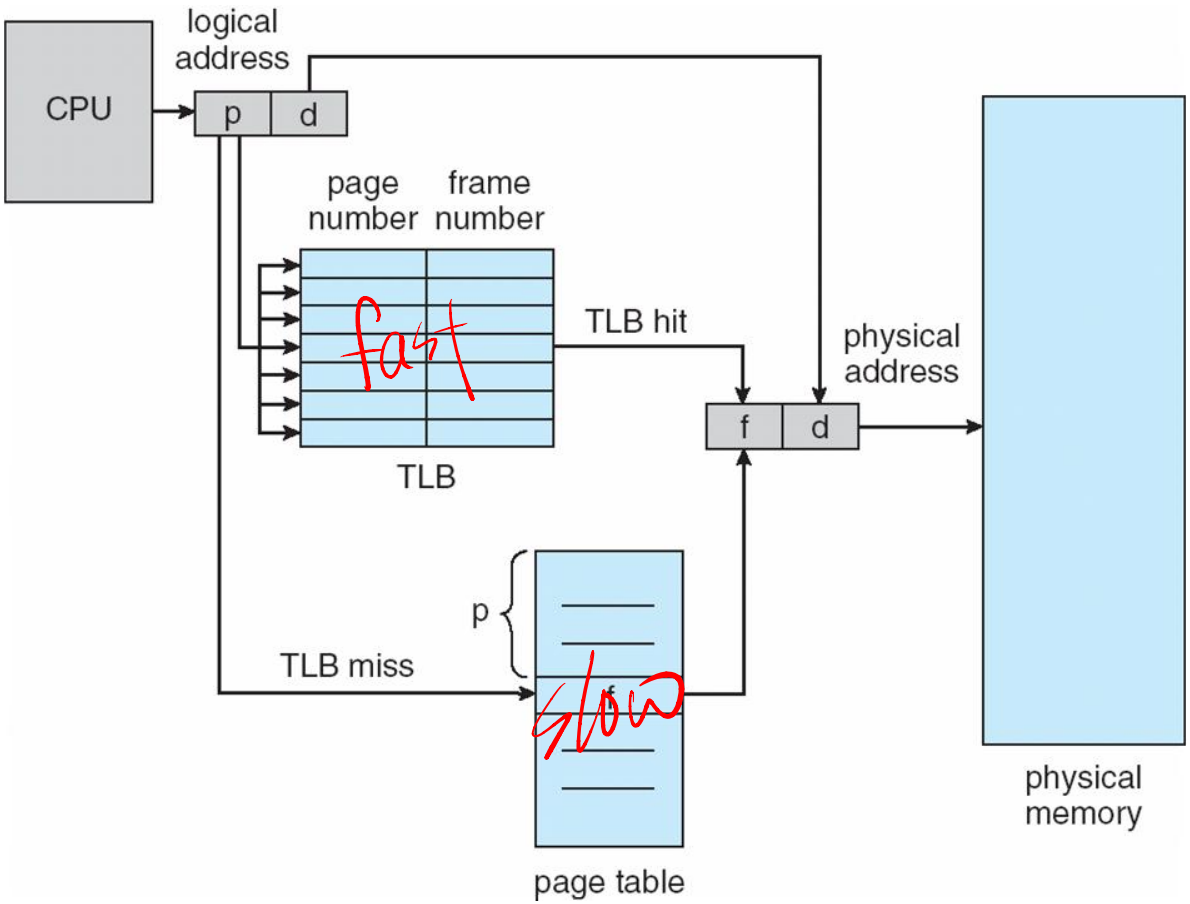
Page #	Frame #

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

Hit ratio – percentage of times that a page number is found in the TLB

An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.

Suppose that 10 nanoseconds to access memory.

If we find the desired page in TLB then a mapped-memory access take 10 ns

Otherwise we need two memory access so it is 20 ns

Effective Access Time (**EAT**)

EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

implying 20% slowdown in access time

Consider amore realistic hit ratio of 99%,

EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

implying only 1% slowdown in access time.

$EAT = P \times t + (1-P) \times 2t$

$= t(2-P)$ $\quad P \leq 1$

$EAT \geq t$

Memory Protection

Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed

Can also add more bits to indicate page execute-only, and so on

Valid-invalid bit attached to each entry in the page table:

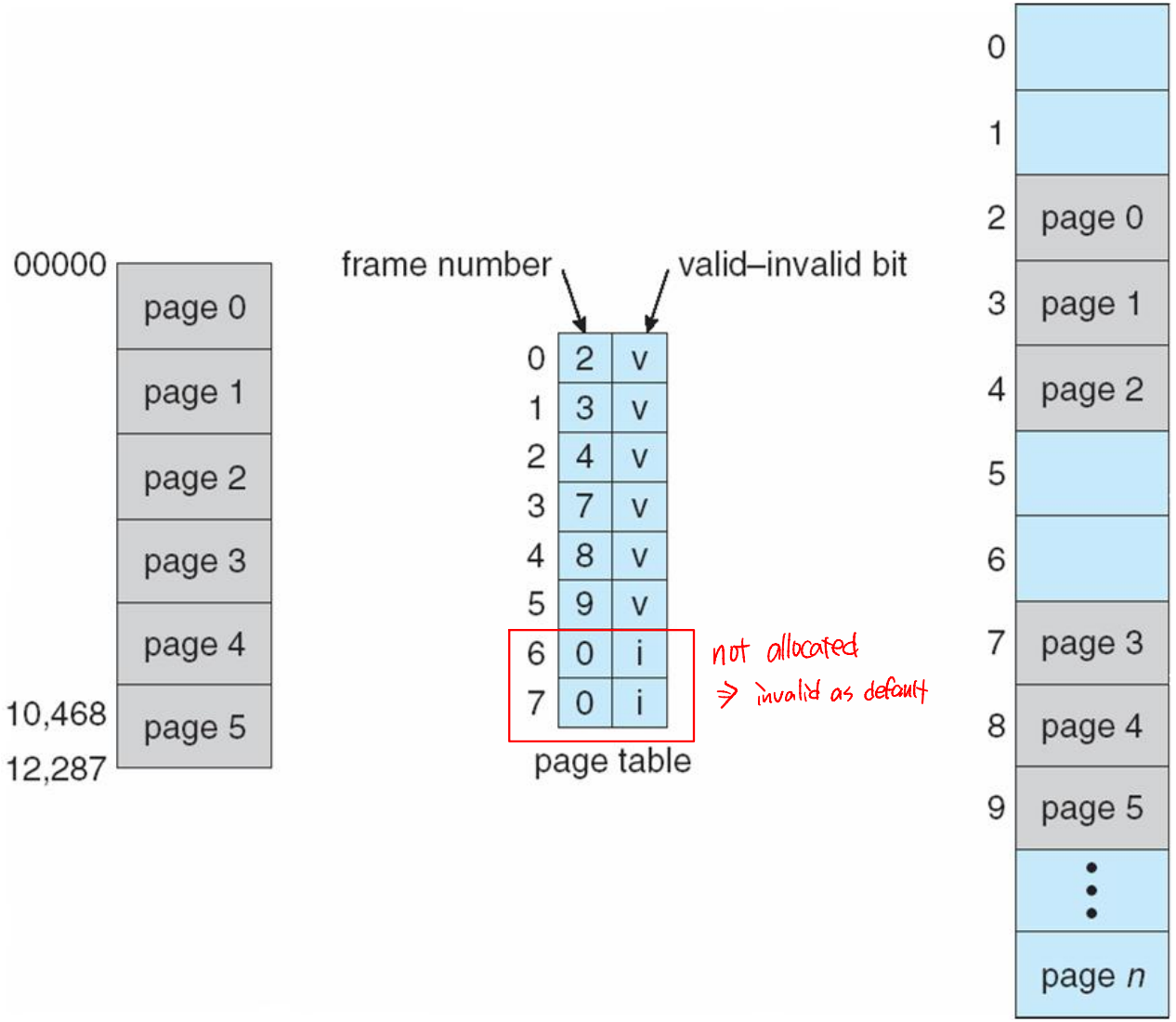
"valid" indicates that the associated page is in the process' logical address space, and is thus a legal page

"invalid" indicates that the page is not in the process' logical address space

Or use **page-table length register** (**PTLR**)

Any violations result in a trap to the kernel

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)

Similar to multiple threads sharing the same process space

Also useful for interprocess communication if sharing of read-write pages is allowed

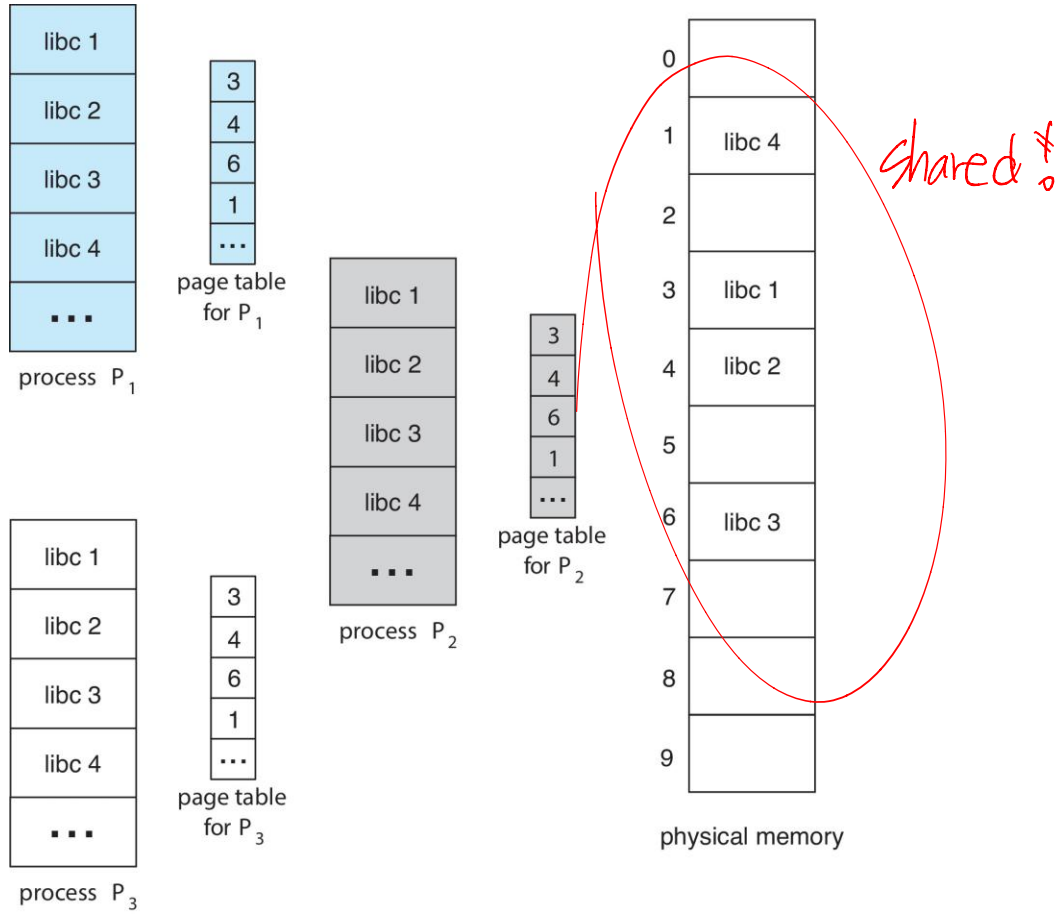
Private code and data

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

코드는 공유
데이터는 각자 관리

Shared Pages Example

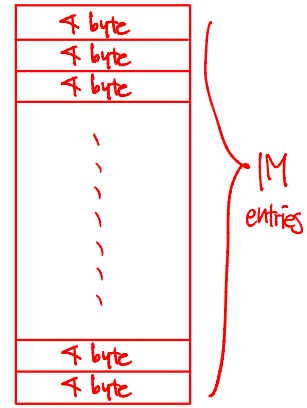


Structure of the Page Table

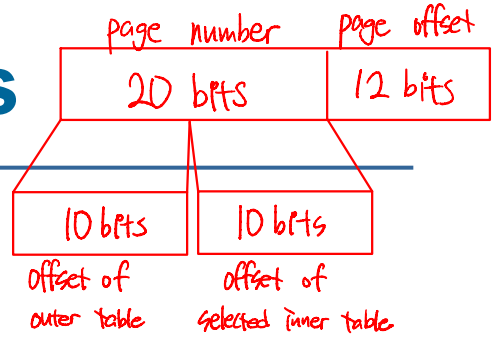
in 32 bit env

Memory structures for paging can get huge using straight-forward methods

- Consider a 32-bit logical address space as on modern computers
- Page size of 4 KB (2^{12}) ⇒ 12 bits needed for offset
- Page table would have 1 million entries ($2^{32} / 2^{12}$) = 2^{20} for representing address space
- If each entry is 4 bytes (32-bits) ➔ each process 4 MB of physical address space for the page table alone
  - Don't want to allocate that contiguously in main memory
- One simple solution is to divide the page table into smaller units
  - Hierarchical Paging (1)
  - Hashed Page Tables (2)
  - Inverted Page Tables (3)



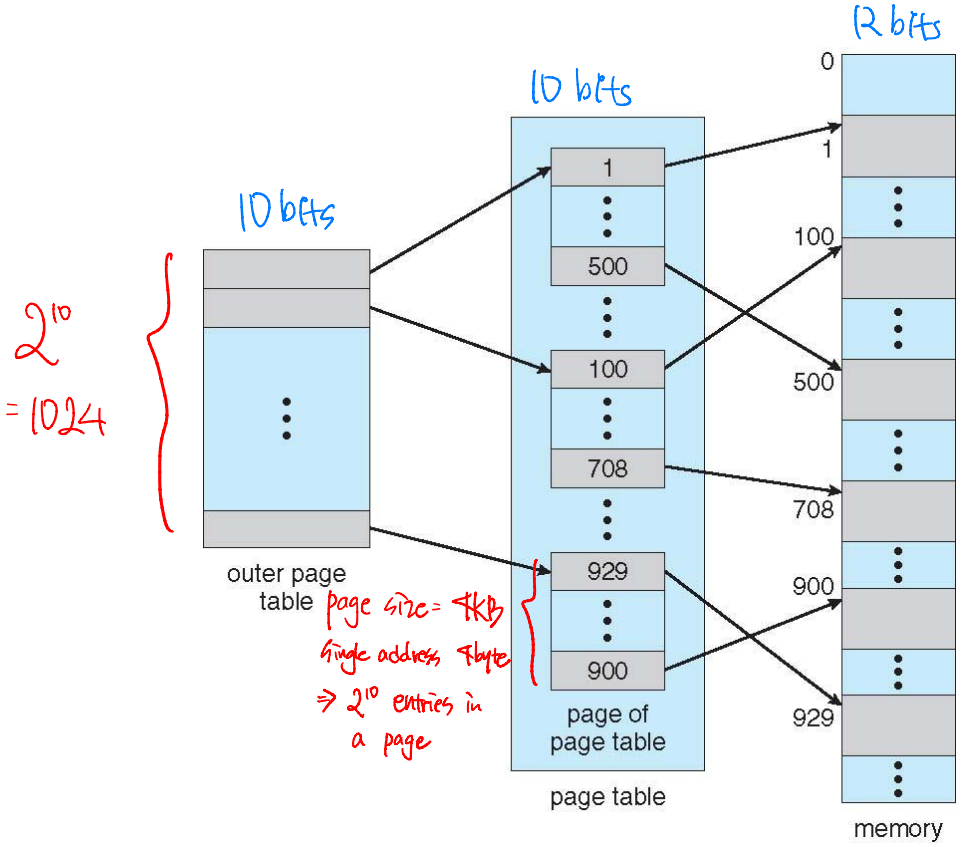
Hierarchical Page Tables



Break up the logical address space into multiple page tables

A simple technique is a two-level page table

We then page the page table



Two-Level Paging Example

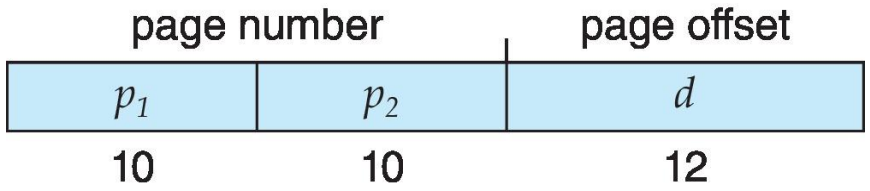
A logical address (on 32-bit machine with 4K page size) is divided into:

- 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 outer page table number
- A 10-bit inner page table offset

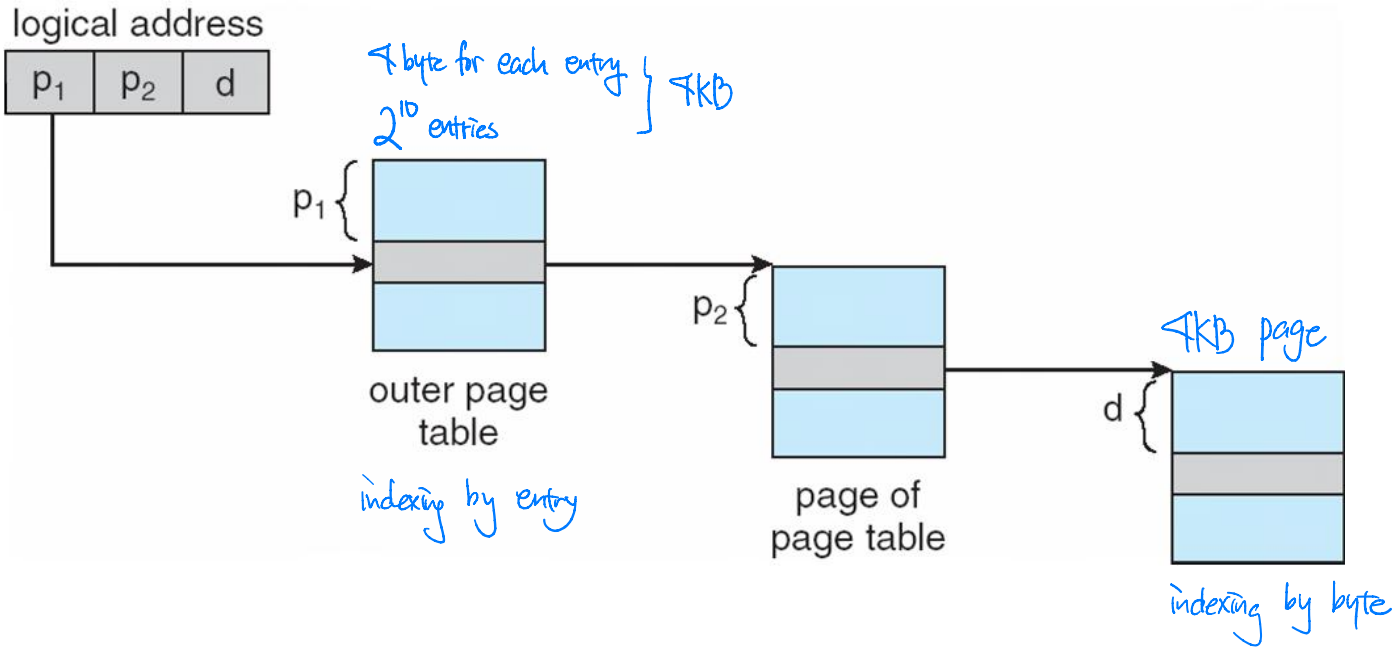
Thus, a logical address is as follows:



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

Known as **forward-mapped page table**

Address-Translation Scheme



64-bit Logical Address Space

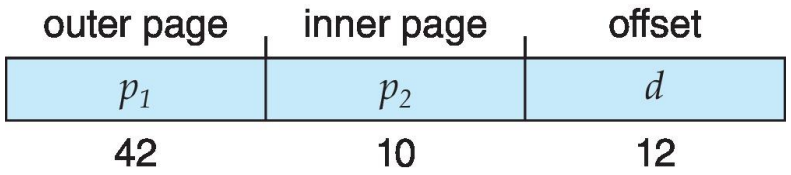
Even two-level paging scheme not sufficient

If page size is 4 KB (2^{12})

Then page table has 2^{52} entries

If two level scheme, inner page tables could be 2^{10} (4-byte) entries

Address would look like



Outer page table has 2^{42} entries or 2^{44} bytes

One solution is to add a 2nd outer page table

not fully use 64 bit address

But in the following example the 2nd outer page table is still 2^{34} bytes in size

- And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2^{34} bytes | 2^{12} bytes | 2^{12} bytes | 2^{12} bytes

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

2^{32} entries

4 bytes for each entry

$\Rightarrow 2^{32} \times 4 = 2^{34}$ bytes

(2)

not ensure unique value ⇒ pigeonhole principle

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 ⇒ subset of entries which have same hash value.

Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element

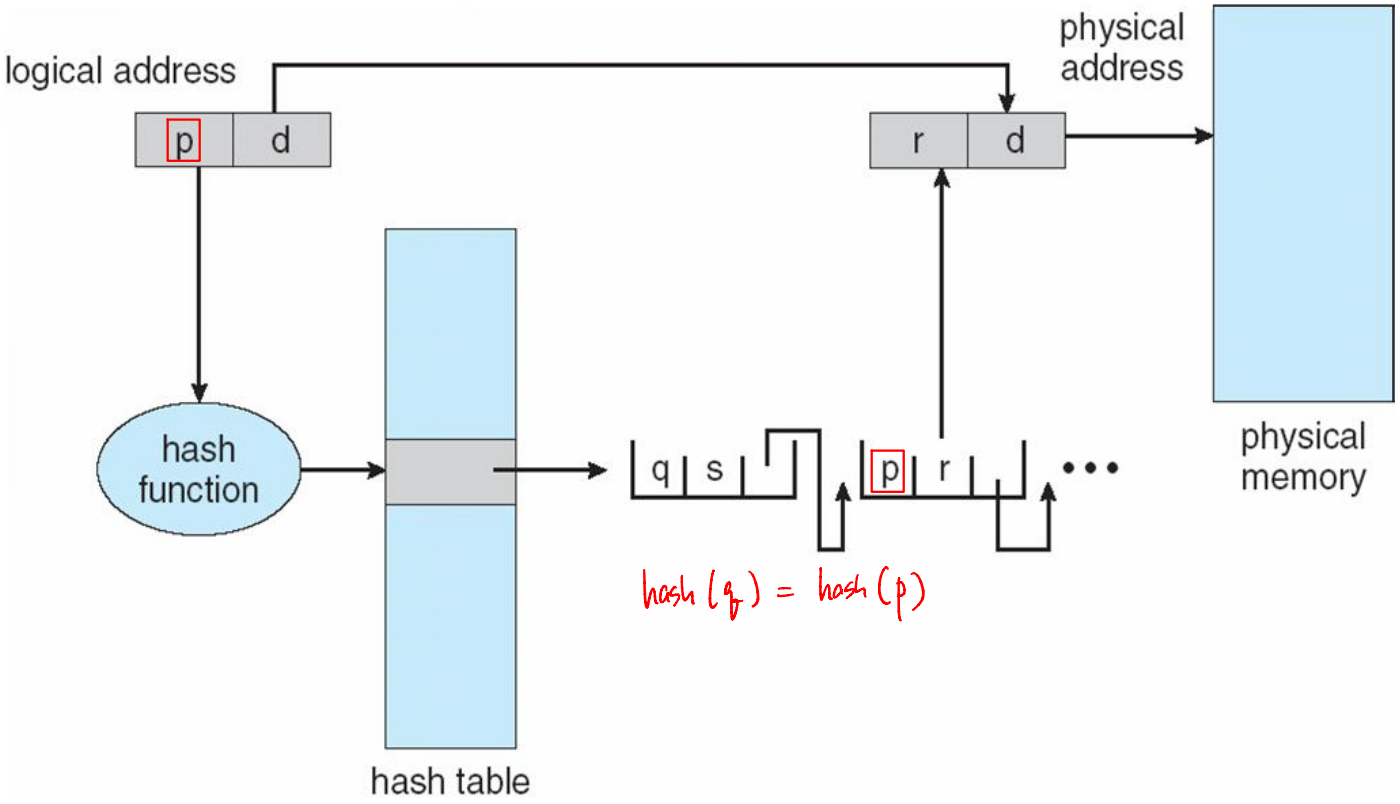
Virtual page numbers are compared in this chain searching for a match

- If a match is found, the corresponding physical frame is extracted

Variation for 64-bit addresses is **clustered page tables**

- Similar to hashed but each entry refers to several pages (such as 16) rather than 1
- Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

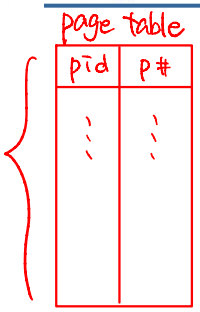
Hashed Page Table



Inverted Page Table

page table: virtual → physical
inverted page table: physical → virtual

only one page table



entry = # frame

Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages

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

process ID
page number

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

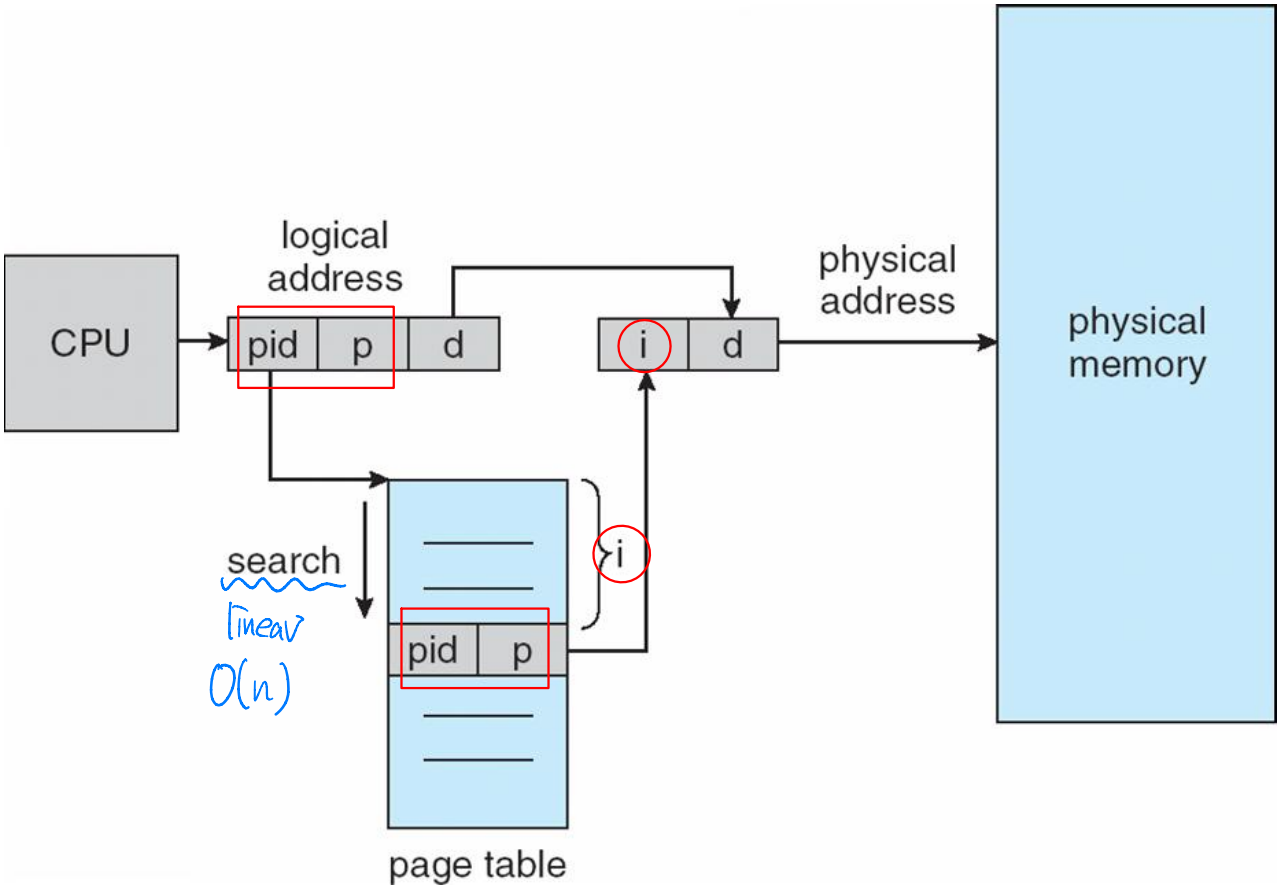
Use hash table to limit the search to one — or at most a few — page-table entries

- TLB can accelerate access

But how to implement shared memory?

- One mapping of a virtual address to the shared physical address ⇒ difficult to implement shared MEM

Inverted Page Table Architecture



Oracle SPARC Solaris

Consider modern, 64-bit operating system example with tightly integrated HW

- Goals are efficiency, low overhead

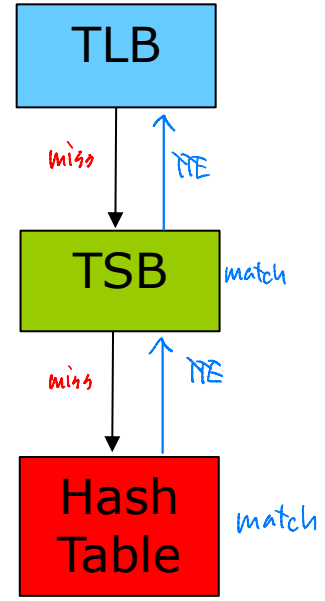
Based on hashing, but more complex

Two hash tables

- One kernel and one for all user processes
- Each maps memory addresses from virtual to physical memory
- Each entry represents a contiguous area of mapped virtual memory,
  - More efficient than having a separate hash-table entry for each page
- Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.



Swapping

A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution

HDD, SSD, ...

- Total physical memory space of processes can exceed physical memory

Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images ⇒ MMU directly access backing store

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

System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

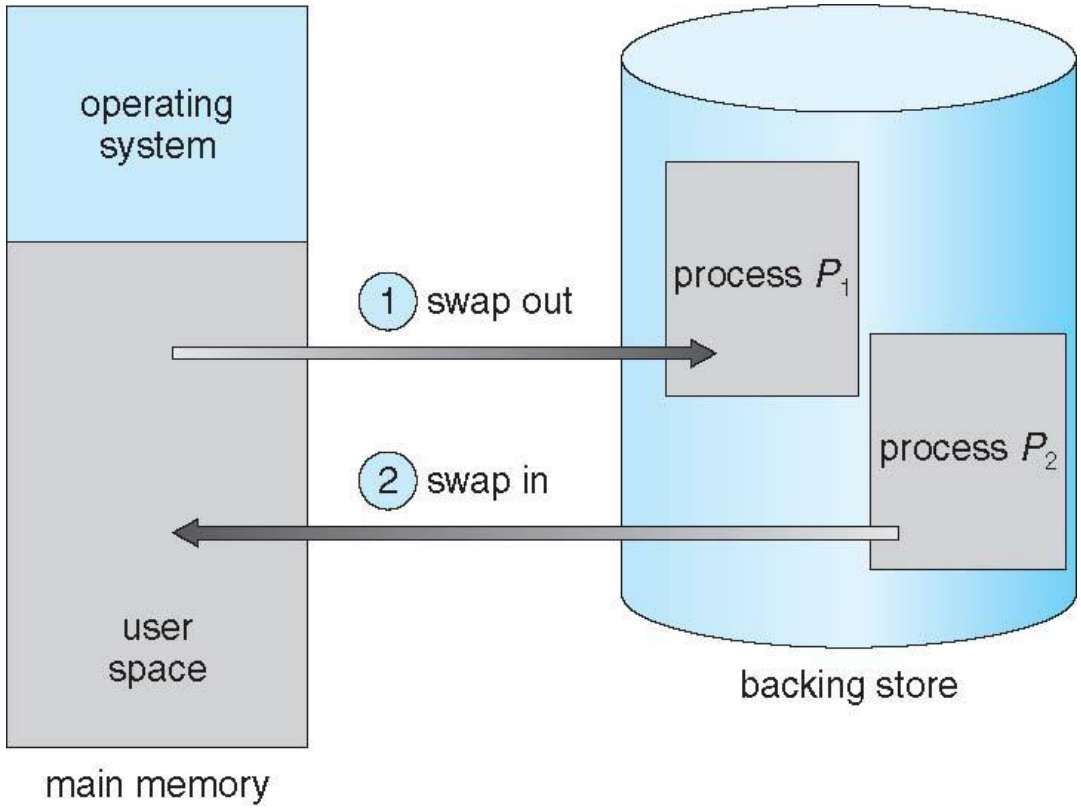
Does the swapped out process need to swap back in to same physical addresses?

- Depends on address binding method
  - load time binding : yes
  - execution time binding : No
  - Plus consider pending I/O to / from process memory space

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

- Swapping normally disabled ∵ cause overhead
- Started if more than threshold amount of memory allocated
- Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Context Switch Time including Swapping

If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process

Context switch time can then be very high

100MB process swapping to hard disk with transfer rate of 50MB/sec

- Swap out time of 2000 ms
- Plus swap in of same sized process
- Total context switch swapping component time of 4000ms (4 seconds)

Can reduce if reduce size of memory swapped – by knowing how much memory really being used

- System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

Other constraints as well on swapping

- Pending I/O – can't swap out as I/O would occur to wrong process
- Or always transfer I/O to kernel space, then to I/O device
  - Known as **double buffering**, adds overhead

Standard swapping not used in modern operating systems

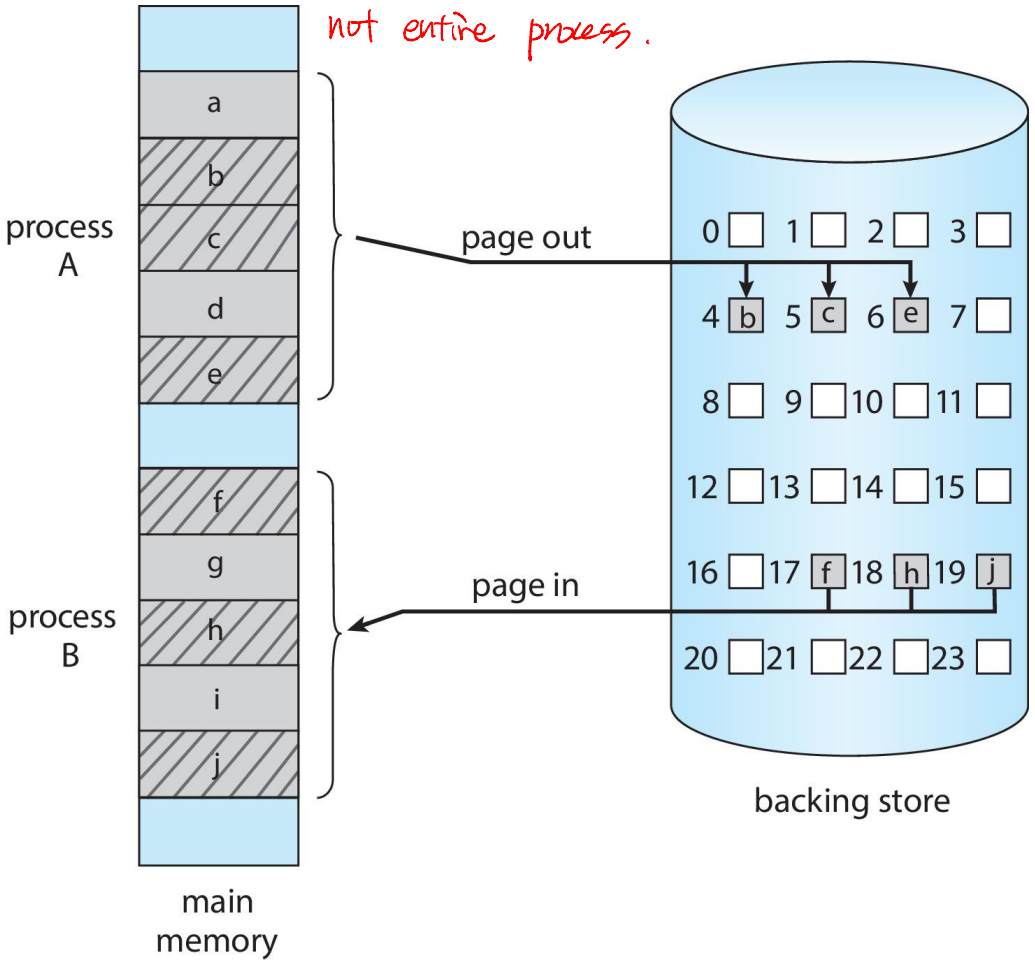
- But modified version common
  - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Swapping with Paging

part of process (page) ⇒ swapping
not entire process.



Example: The Intel 32 and 64-bit Architectures

Dominant industry chips

Pentium CPUs are 32-bit and called IA-32 architecture

Current Intel CPUs are 64-bit and called IA-64 architecture

Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

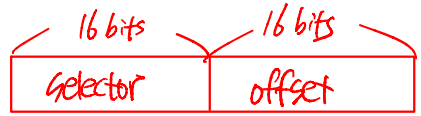
MEM ⇒ several segment (logical unit)

Supports both segmentation and segmentation with paging

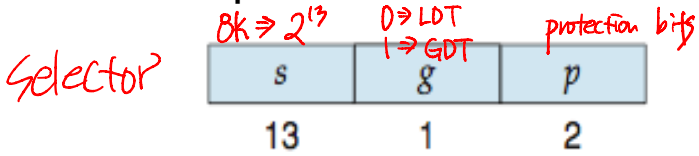
- Each segment can be 4 GB (MAX) ⇒ different # of page for each segment
- Up to 16 K segments per process 2^{14} segments
- Divided into two partitions
  - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**)) — segment
  - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**)) — segment

Example: The Intel IA-32 Architecture (Cont.)

CPU generates logical address (selector,offset)

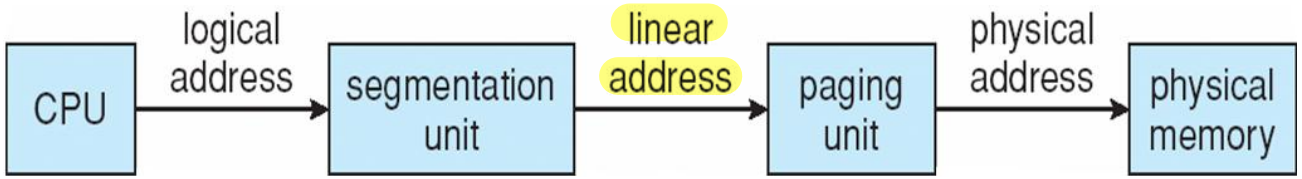


- Selector given to segmentation unit
  - Which produces linear addresses

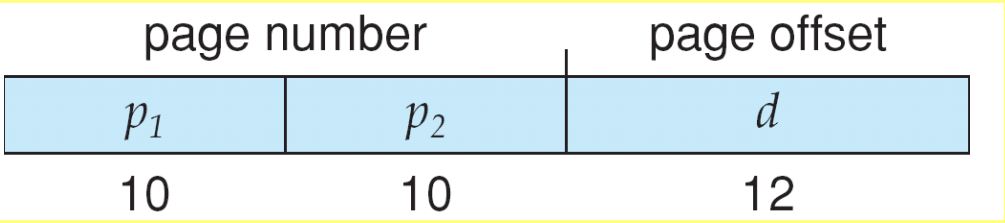


- Linear address given to paging unit
  - Which generates physical address in main memory
  - Paging units form equivalent of MMU
  - Pages sizes can be 4 KB or 4 MB

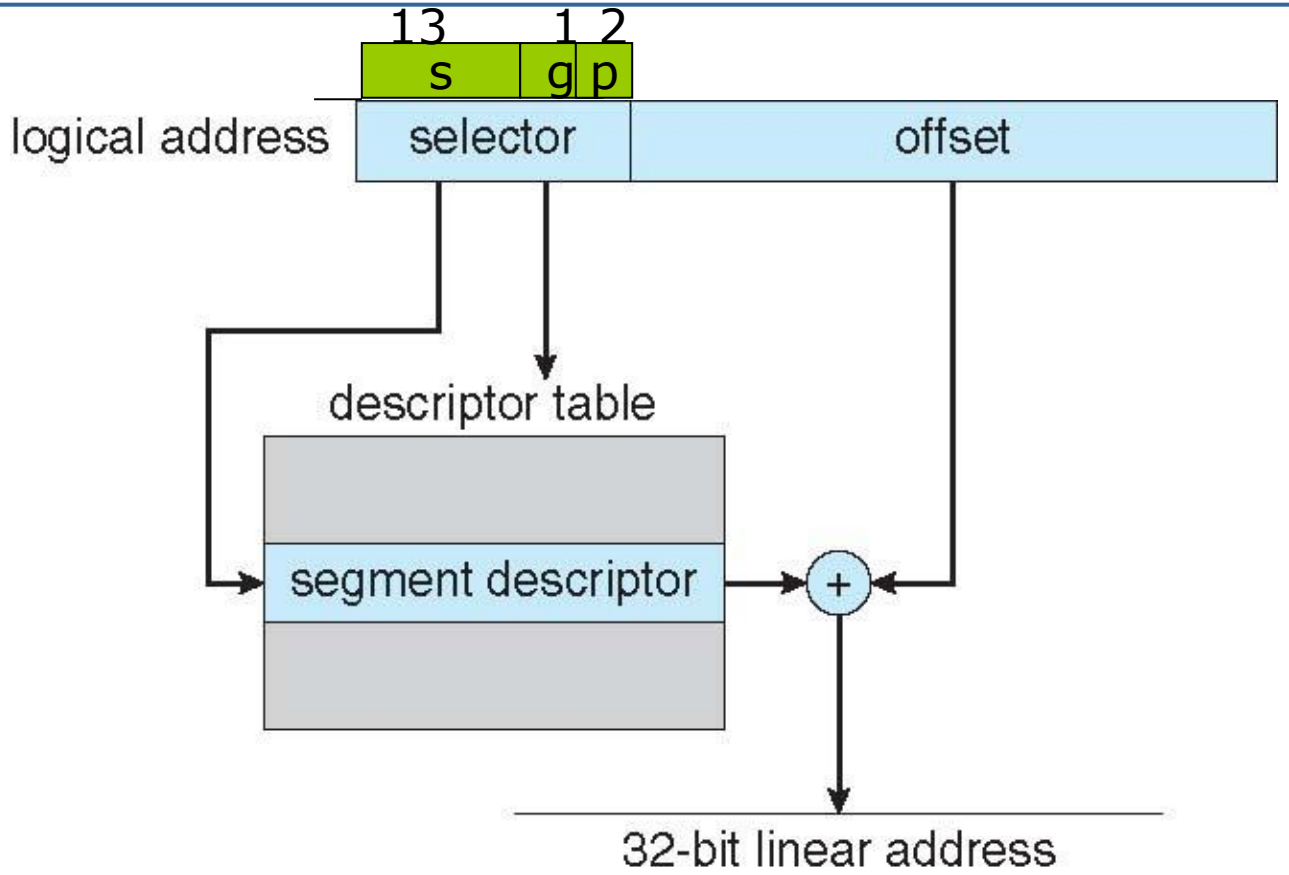
Logical to Physical Address Translation in IA-32



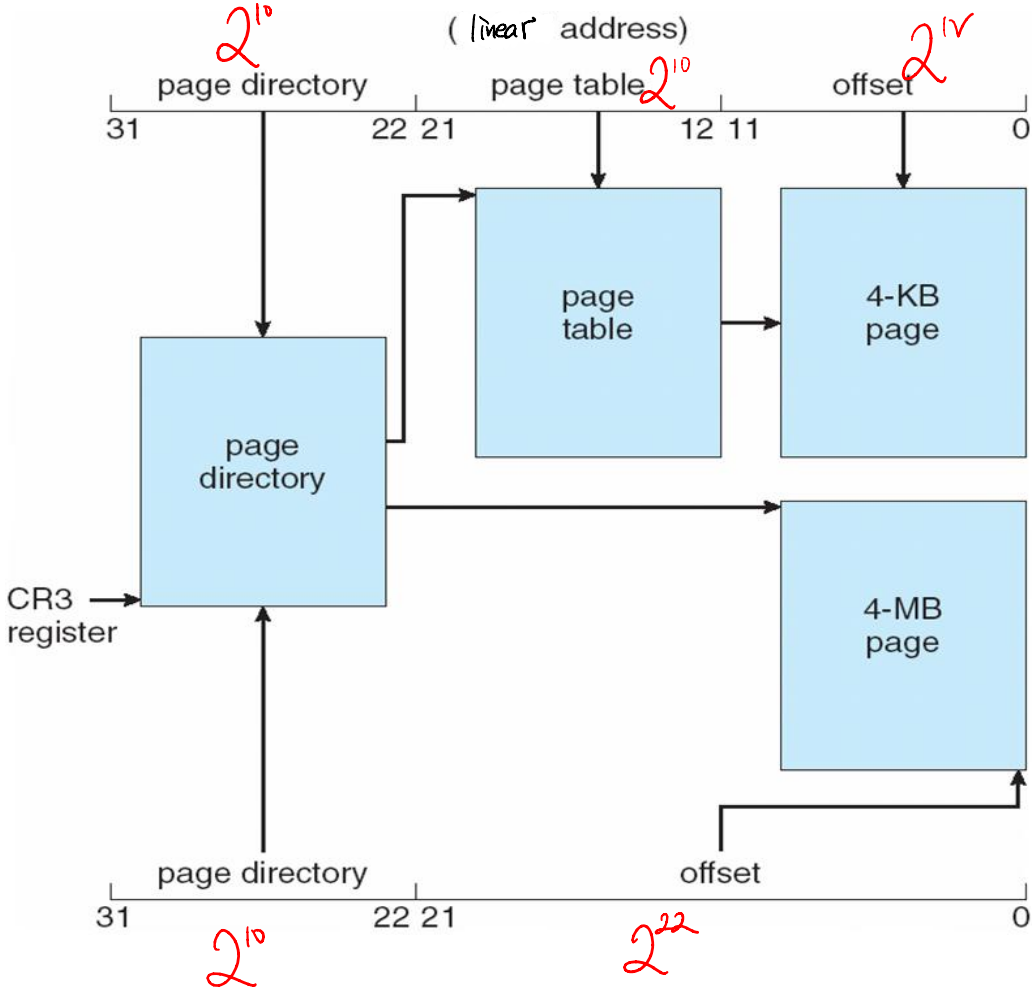
two level paging



Intel IA-32 Segmentation



Intel IA-32 Paging Architecture

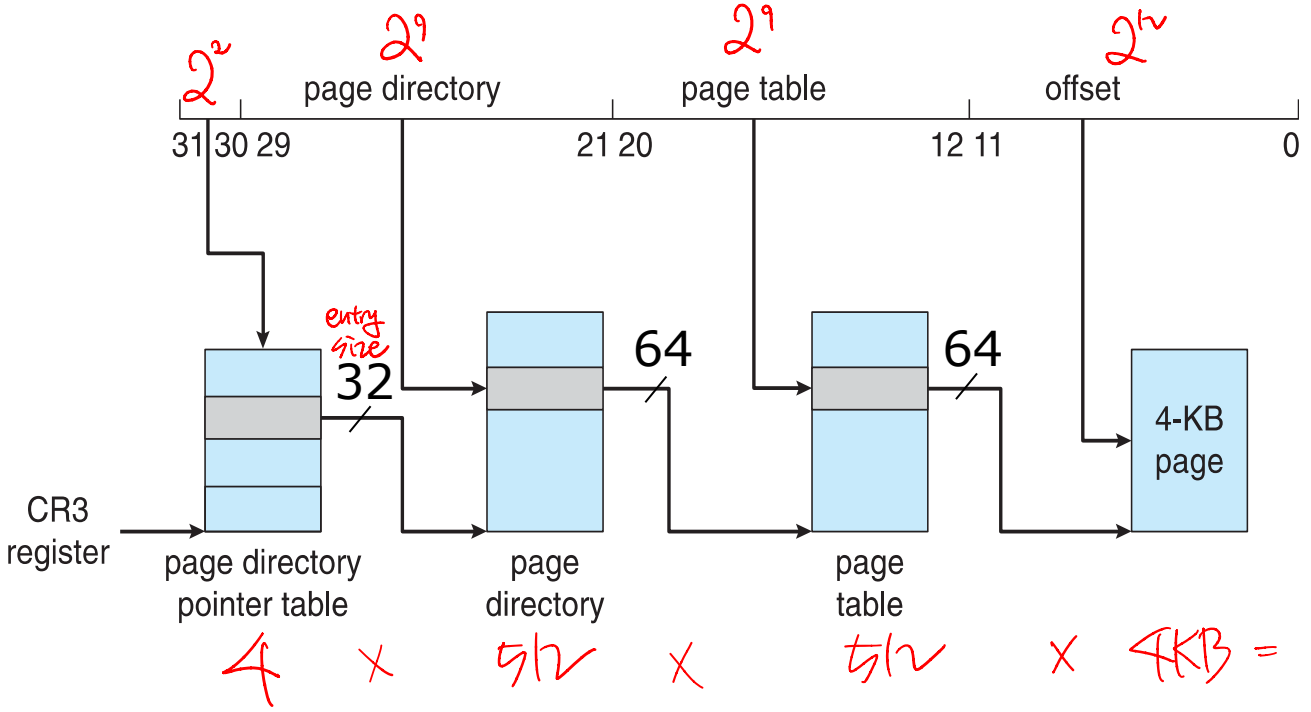


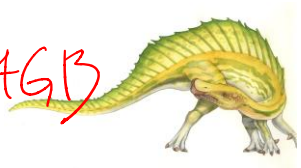
Intel IA-32 Page Address Extensions

Max 4GB MEM

32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space

- Paging went to a 3-level scheme
- Top two bits refer to a **page directory pointer table**
- Page-directory and page-table entries moved to 64-bits in size
- Net effect is increasing address space to 36 bits – 64GB of physical memory





Intel x86-64

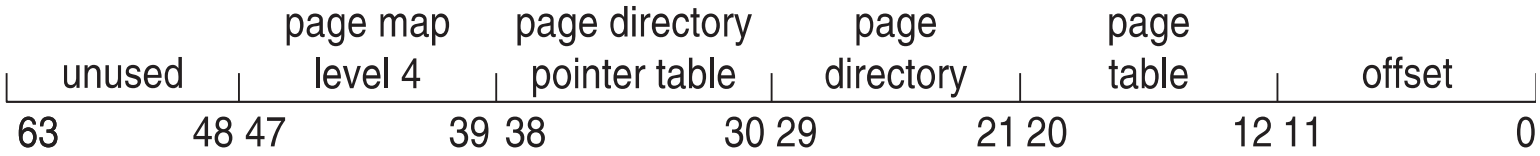
Current generation Intel x86 architecture

64 bits is ginormous (> 16 exabytes)

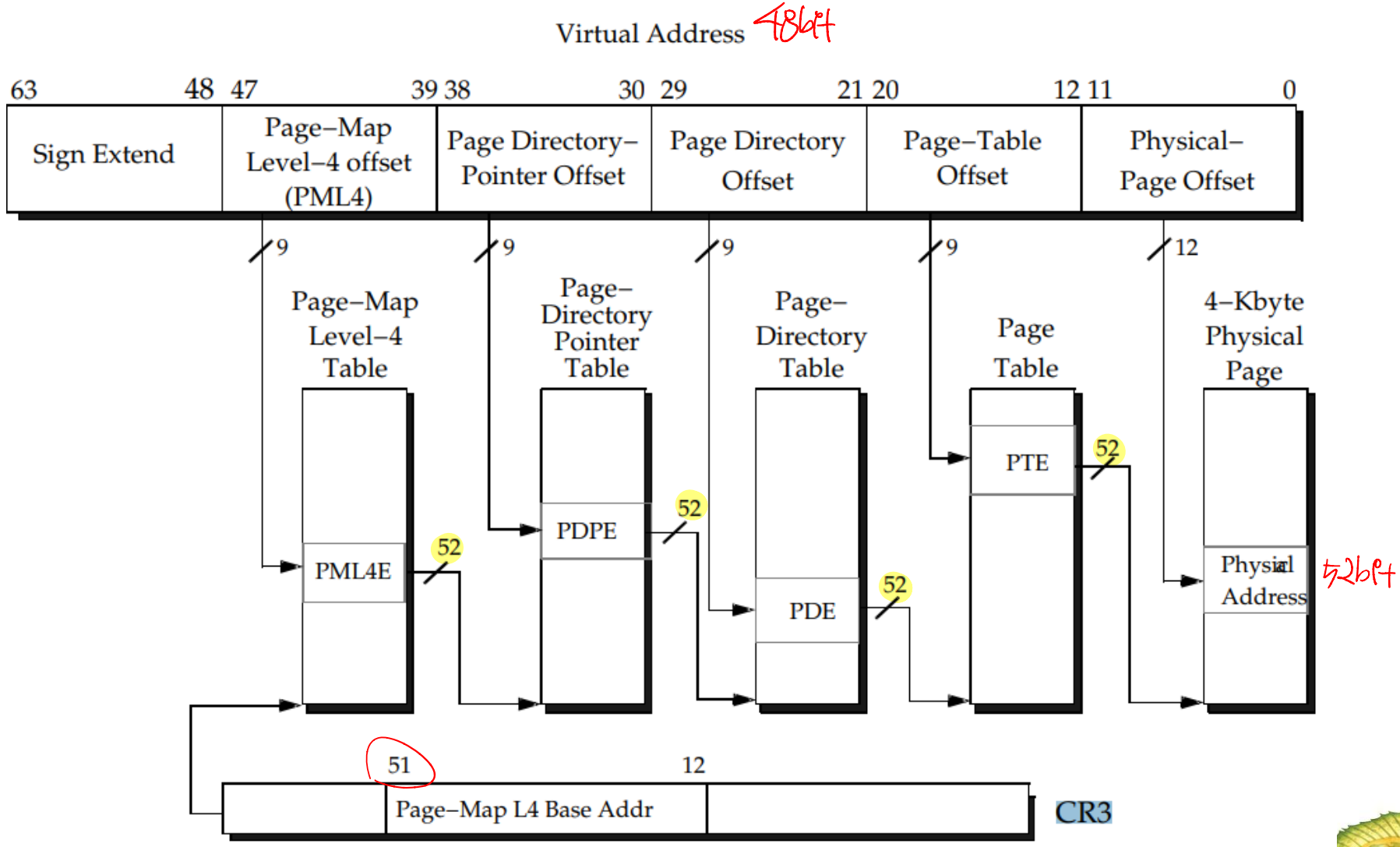
In practice only implement 48 bit addressing

- Page sizes of 4 KB, 2 MB, 1 GB
- Four levels of paging hierarchy

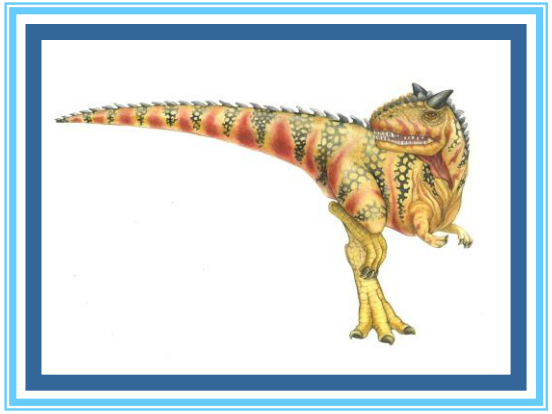
Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



Intel x86-64 (Cont'd) – Long mode PAE



End of Chapter 9



Example of Segmentation