

Chapter 9

Main Memory

Da-Wei Chang

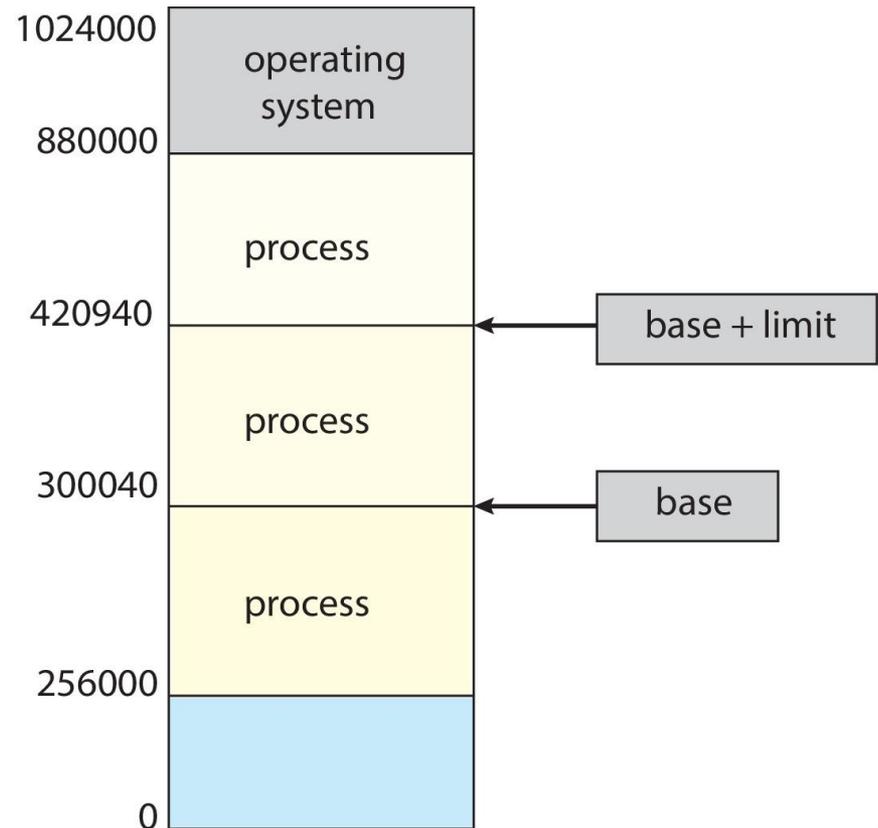
CSIE.NCKU

Outline

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Examples: Intel and ARM Architectures

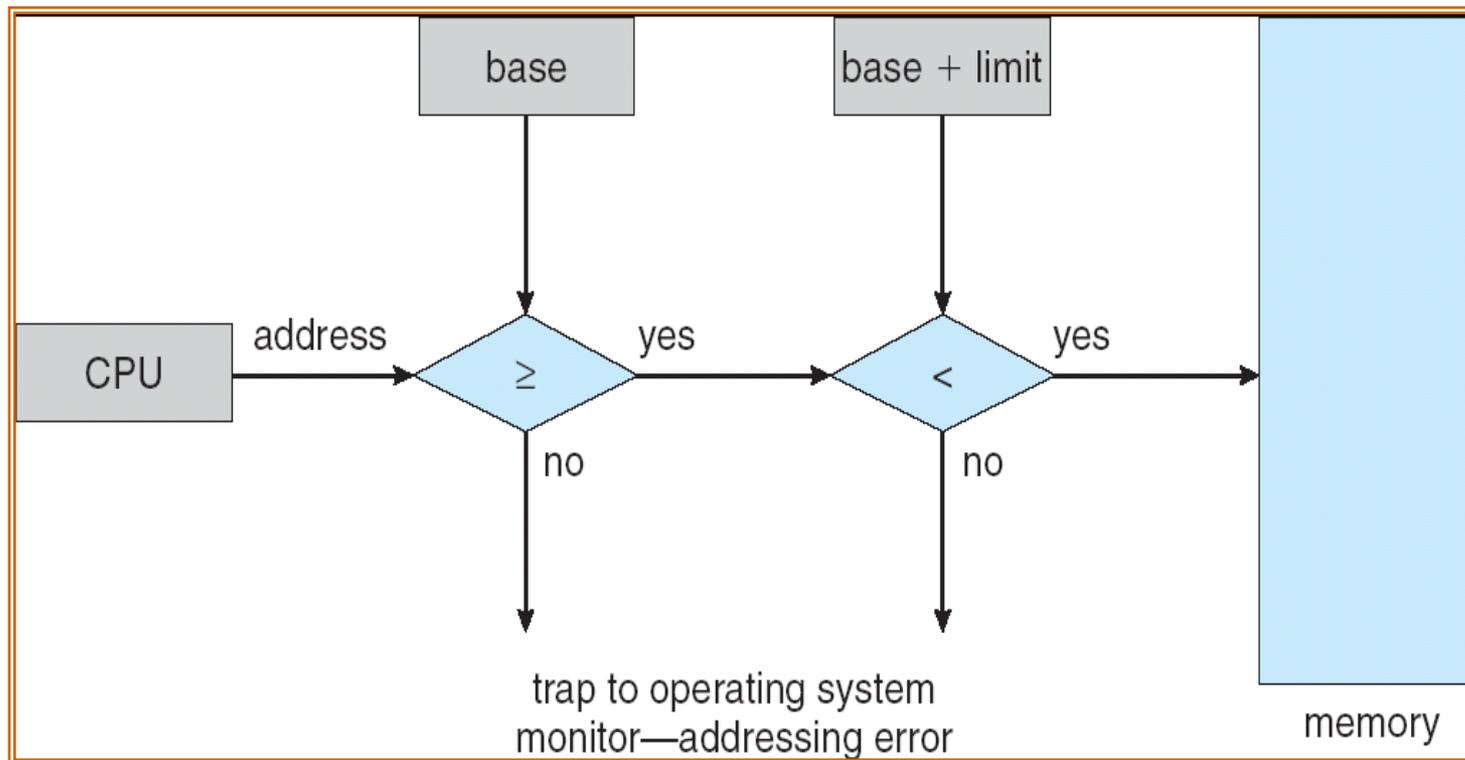
Basic Hardware

- Make sure that each process has a separate memory space
 - Use 2 registers
 - base & limit



Basic Hardware

On each memory access ---

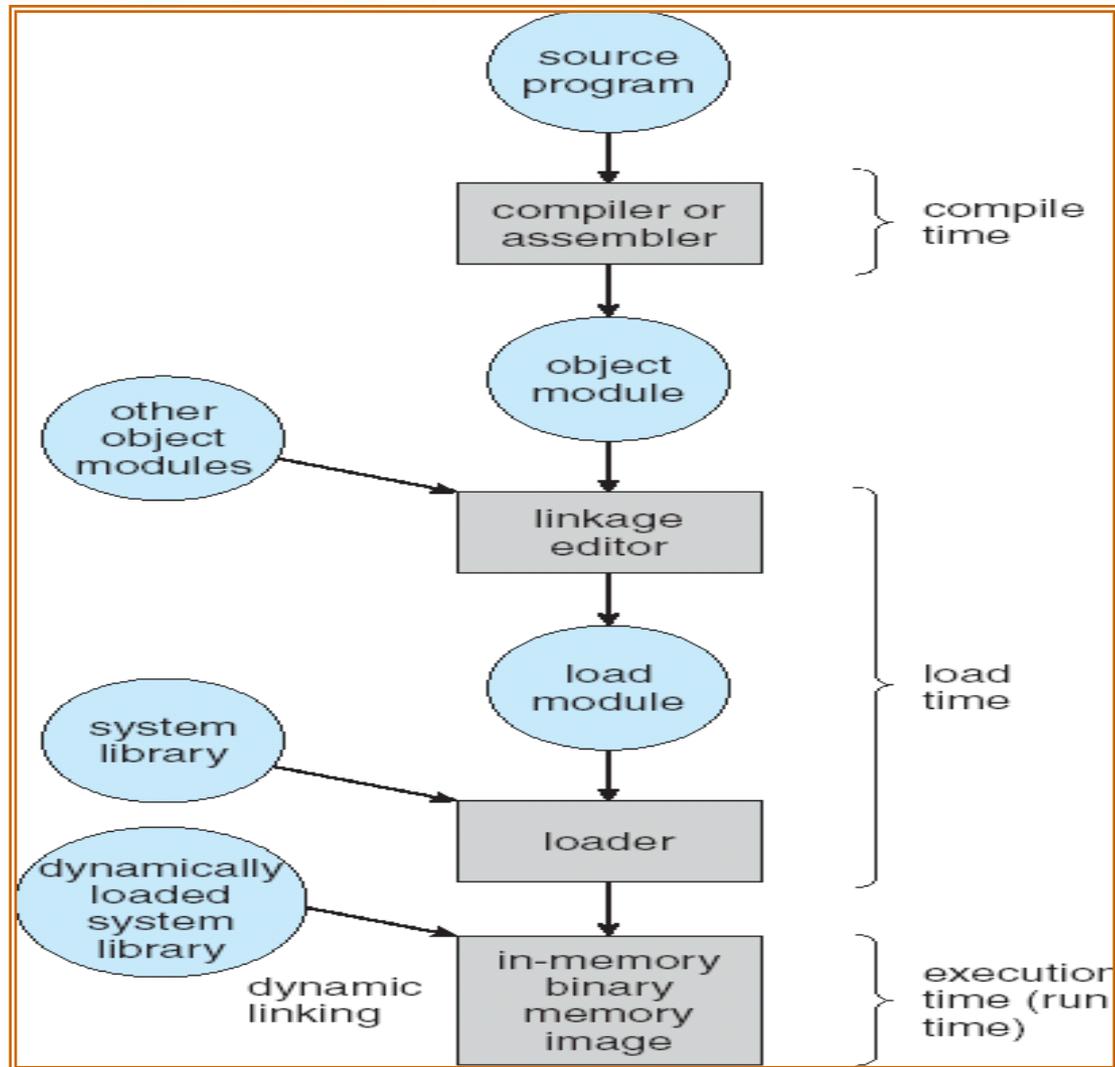


Base and Limit registers are **loaded by OS (by privileged instructions)**
- prevent user programs from changing the register contents

Address Binding

- Programs are originally stored on the disk
- Program must be brought into memory before being run
- User programs go through several steps before being run
 - Addresses may be represented in different ways in these steps
 - Symbol, relocatable address, absolute address

Multistep Processing of a User Program



Binding of Instructions and Data to Memory

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

- **Compile time address binding**: If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes
- **Load time address binding** : the compiler must generate *relocatable code* if memory location is not known at compile time
 - e.g., “byte 104 from beginning of this module”
 - Linker/loader will bind relocatable addresses to absolute addresses
 - e.g, byte 104 of this module = address 01104
- **Execution time address binding**: Binding delayed until run time **if the process can be moved during its execution** from one memory segment to another
 - Most operating systems use this method

Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as *virtual address*
- **Physical address** – addresses seen by the memory unit
- Logical address space
 - All logical addresses generated by a program
- Physical address space
 - Set of physical addresses corresponding to these logical addresses
- The concept of a *logical address space* that is bound to a **separate physical address space** is central to proper memory management in an OS

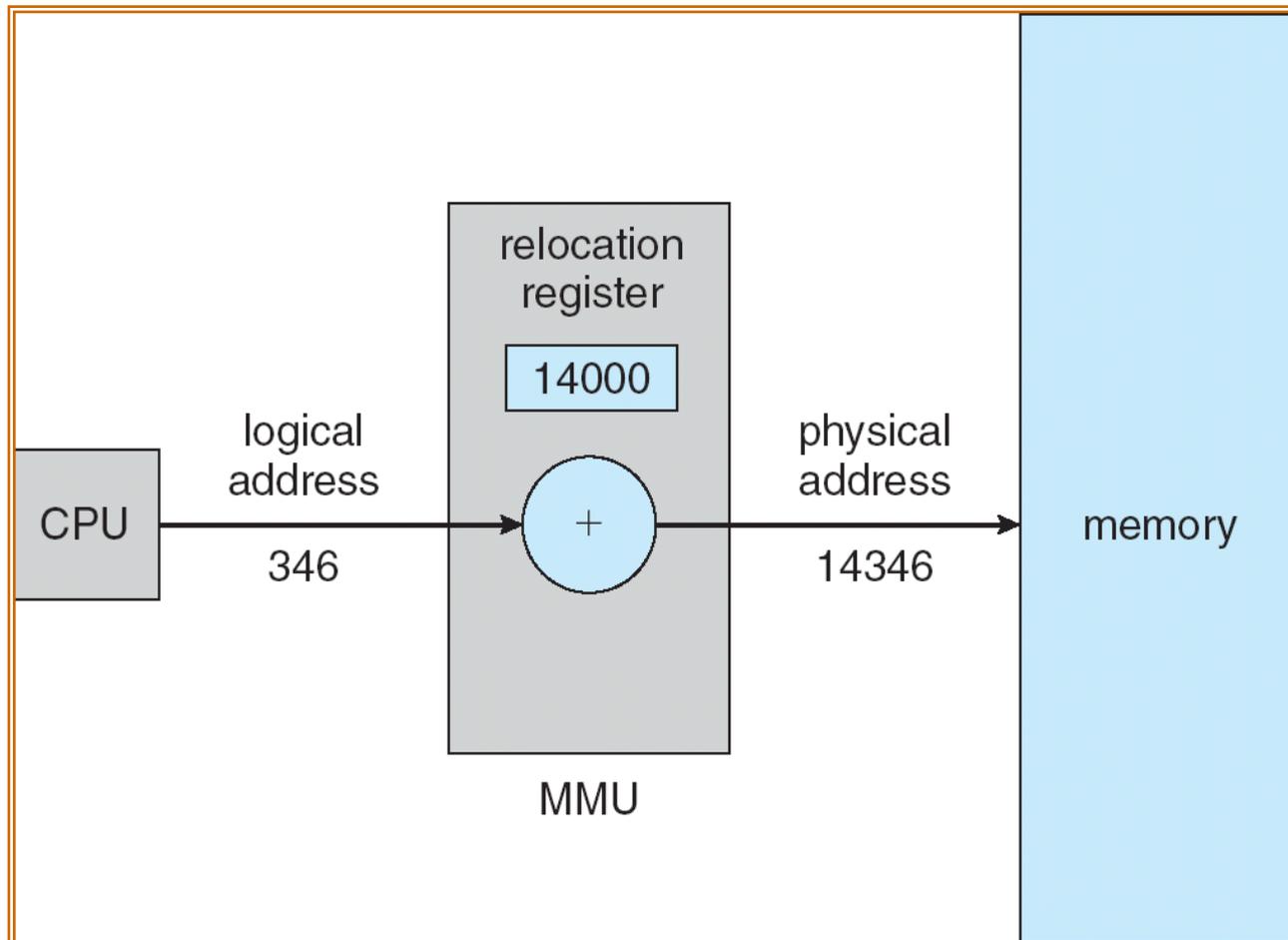
Logical vs. Physical Address Space

- Logical (virtual) and physical addresses are
 - The same in compile-time and load-time address-binding schemes
 - Different in execution-time address-binding scheme

Memory-Management Unit (MMU)

- **Hardware** unit that maps virtual to physical address
- In a simple 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
- **A user program deals with *logical* addresses**; it never sees the *physical* addresses
 - Different from the basic HW (slide 3), the logical address starts from 0

Dynamic Relocation Using a Relocation Register



Dynamic Loading

- So far, **the entire program** must be in physical memory for process execution
 - Size of a process \leq size of the physical memory
- Dynamic Loading
 - A routine/function is **not loaded until it is called**
 - Initially, main function is loaded
 - When a routine is needed, caller checks to see if the routine is loaded
 - Loader loads the routine and update the **program's addr table** if necessary
 - Save memory space; unused routine is never loaded
 - Better memory space utilization...
 - Useful when **large** amounts of code are needed to handle **infrequently** occurring cases
 - No special support from the operating system is required; **implemented by application programs or libraries**

Dynamic Linking

- Static linking
 - Codes (including libraries) are combined into the binary program image
 - Waste space for shared libraries
- Dynamic linking
 - Linking postponed until execution time
 - Usually used with system libraries
 - Executable program does not contain the lib codes
- Small piece of code, *stub*, is included in the program image for each library routine access
 - used to locate the appropriate **memory-resident** library routine

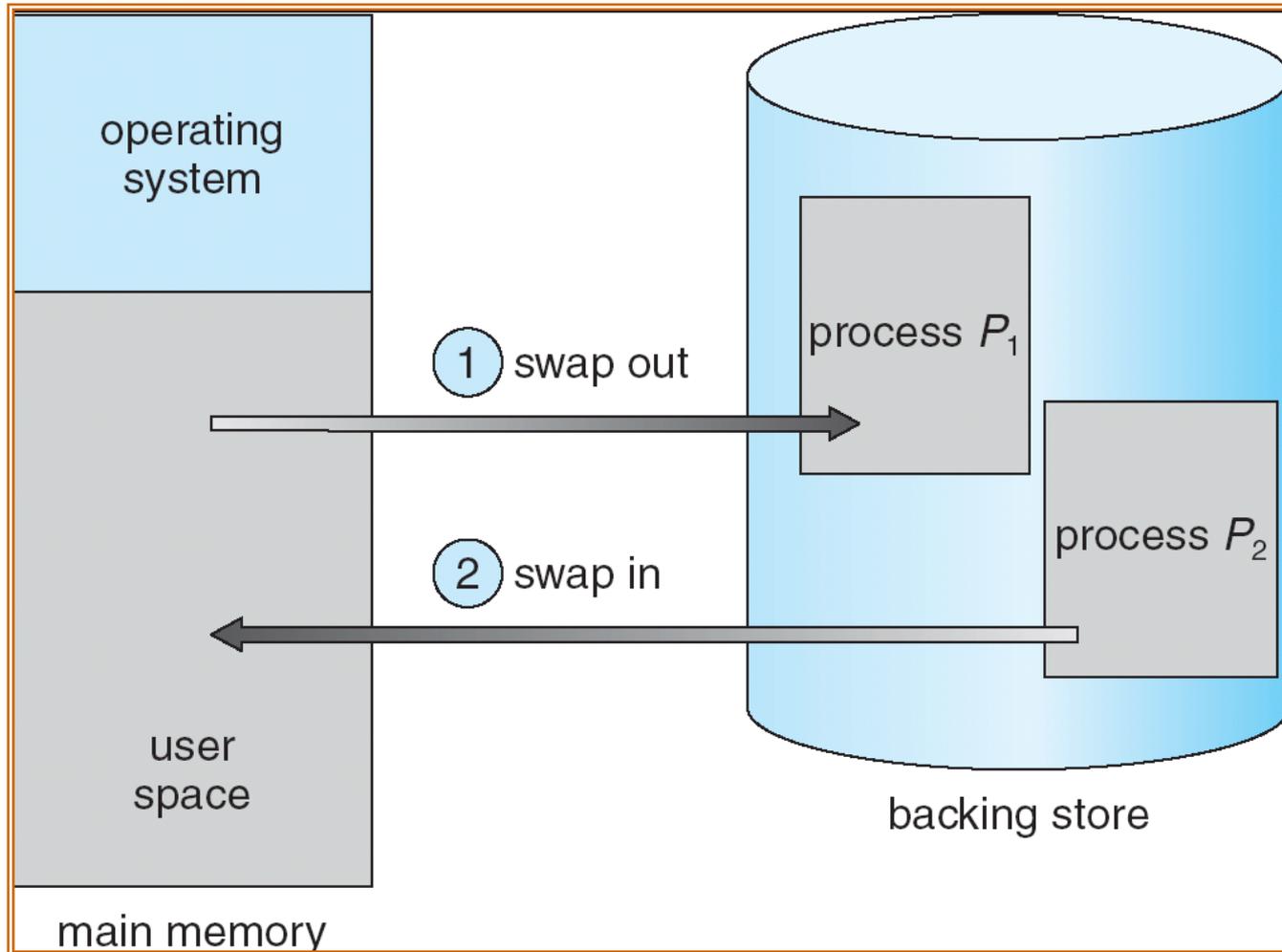
Dynamic Linking

- Stub replaces itself with the address of the routine, and executes the routine
- Requires OS support
 - OS is the only entity that can
 - check if the routine is in another processes' memory space
 - If not in address space, add to the address space
 - allow multiple processes to access the same memory addresses

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** – a disk large enough to accommodate copies of all memory images for all users
- **Which process should be swapped?**
- **Roll out, roll in** – a swapping variant used for **priority-based** scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed. When the higher-priority process terminates, the lower-priority process can be swapped in.

Schematic View of Swapping



Swapping

- Whenever the CPU scheduler decides to execute a process
 - Check to see if the process is in memory
 - Swap in if necessary
 - Swap out (another) if necessary
- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped
 - Swap in/out involves disk IO
 - Swap in a 10MB process on a disk with 40MB/s bandwidth
 - 250 ms (usually larger than a time quantum!!!)
 - Reduce the transfer size
 - Swap what is actually used....(we will see later)

Swapping

- Swapping a process with a pending IO may cause error
 - The memory of the process may be used by other processes
 - So,
 - Never swap a process with a pending IO, or
 - Perform IO with OS buffers
- Standard swapping is used in few systems
 - Too much swapping time
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
 - e.g. enabled only when the load is heavy
 - e.g. **swap only a part of a process**

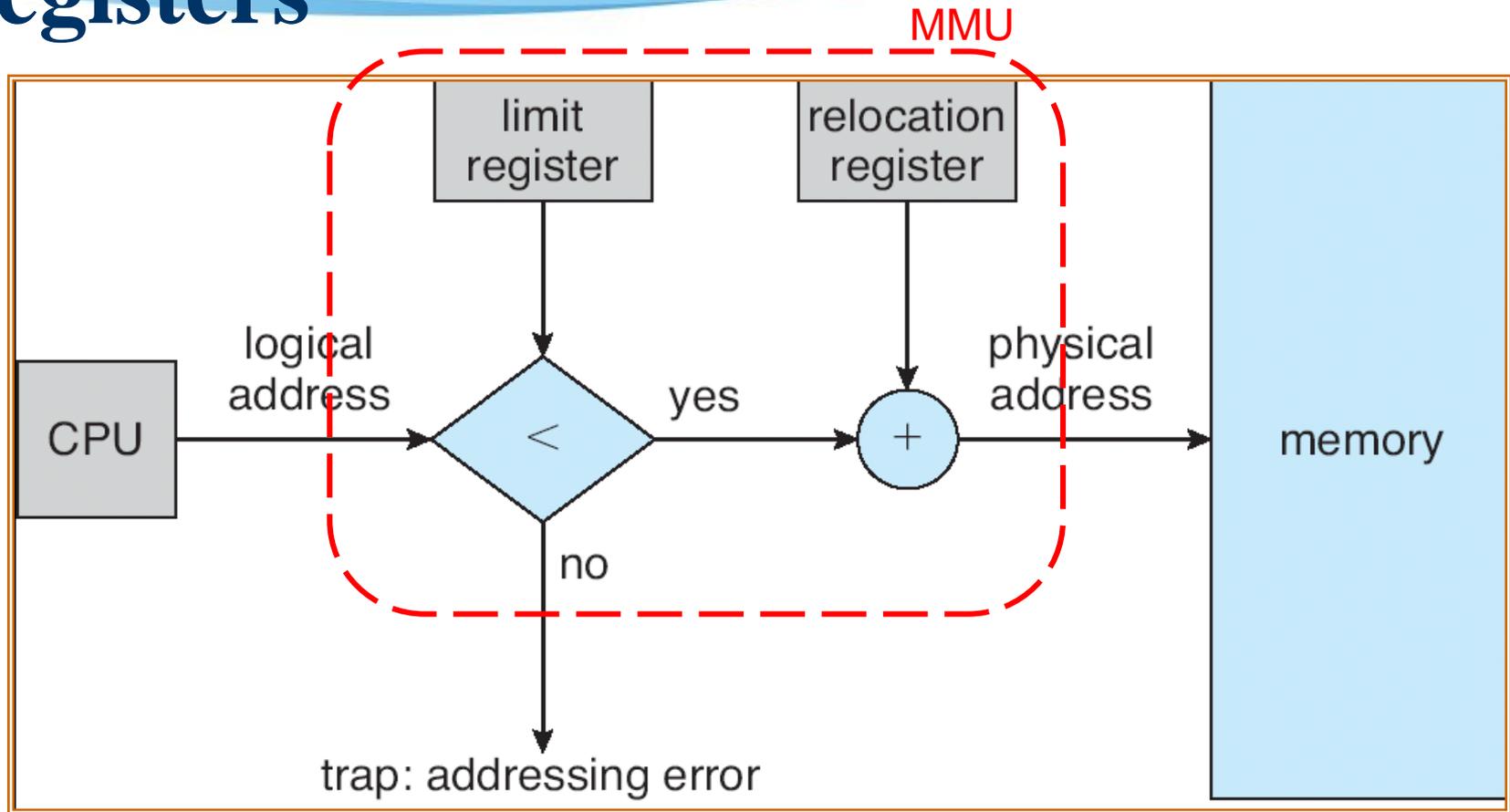
Memory Allocation

- Which memory addresses should be allocated for a process?
- Main memory is usually separated into two partitions
 - Resident operating system, **usually** held in **low** memory with interrupt vector table
 - User processes then usually held in **high** memory
 - So, the high-memory area can be allocated to processes

Contiguous Allocation

- A process is allocated **a range of physical** addresses
- Memory mapping and protection
 - Can be implemented by the **relocation-register scheme**
 - protect user processes from each other, and from changing operating-system code and data
 - **Relocation register**: records the **smallest** physical address of the process
 - Translate logical address to physical address
 - **limit register**: max **logical** addresses + 1
 - Valid logical address ranges from 0 to (limit -1)

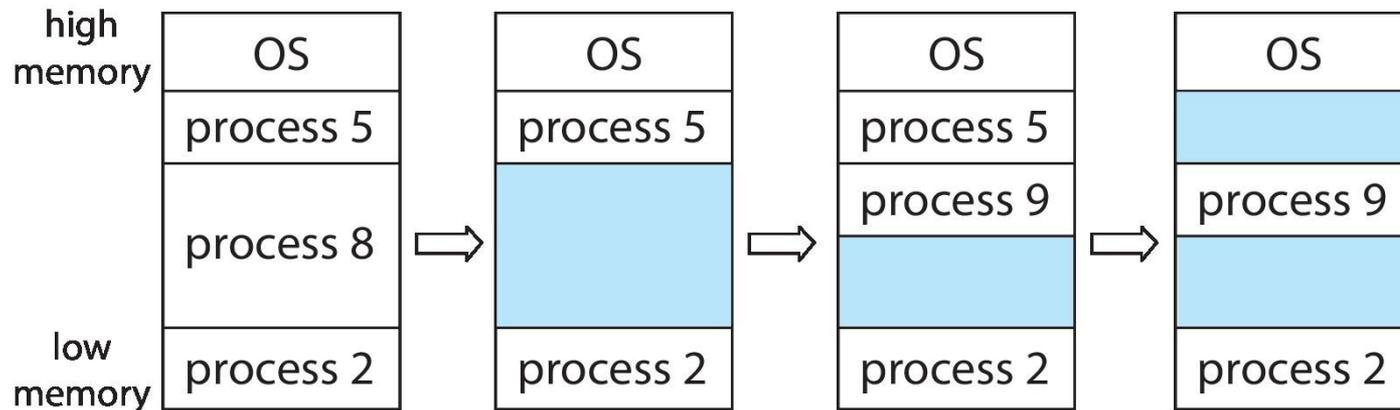
HW of Relocation and Limit Registers



.Valid logical address (0, limit -1)

.The register values will be “switched” during context switches

Contiguous Allocation



- At any time, memory consists of a set of variable-sized **used** partitions and **free** partitions (i.e. **holes**)
- When a process needs to be brought into memory, we need a hole (that is large enough) to accommodate the process
- After the allocation, the values of the **relocation** and **limit** registers are determined.
- When a process exits, its partition is freed, adjacent free partitions are combined

Contiguous Allocation

- How to satisfy a request of **size n** from **a list of 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 is better than best/worst-fit in terms of **speed**

Best-fit is better than worst-fit in terms of **storage utilization**

Fragmentation

- **External Fragmentation** – small holes; total free memory is large enough to satisfy a request, but it is **not contiguous**
 - 50-percent rule: for first fit, given N allocated blocks, another $0.5N$ will be lost to fragmentation on average
 - 1/3 of memory may be unusable!!!
- **Solution 1: 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 with OS buffers

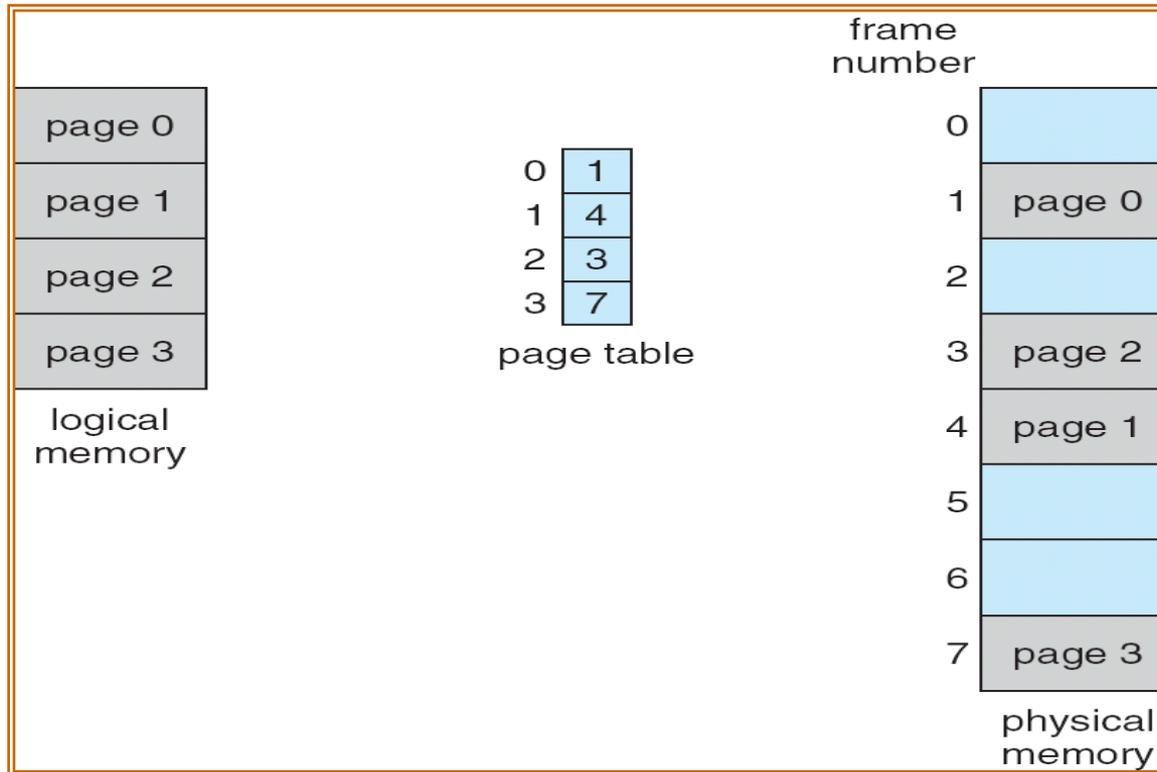
Fragmentation

- Solution 2: non-contiguous memory allocation
 - A process can be contained in a set of small holes
 - One way is to partition memory into fixed-sized blocks and allocate a number of blocks to each in-memory process
 - **Paging** (will be introduced later)
 - Leads to internal fragmentation
 - **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; the extra bytes can not be used by other processes

Paging

- **Physical** 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 8192 bytes)
- Divide **logical** memory into blocks of same size called **pages**
- To run a program of size n pages, the OS needs to find n free frames and then load the program
 - Keep track of all free frames
- Set up a **page table** to translate logical to physical addresses
- **Internal fragmentation**

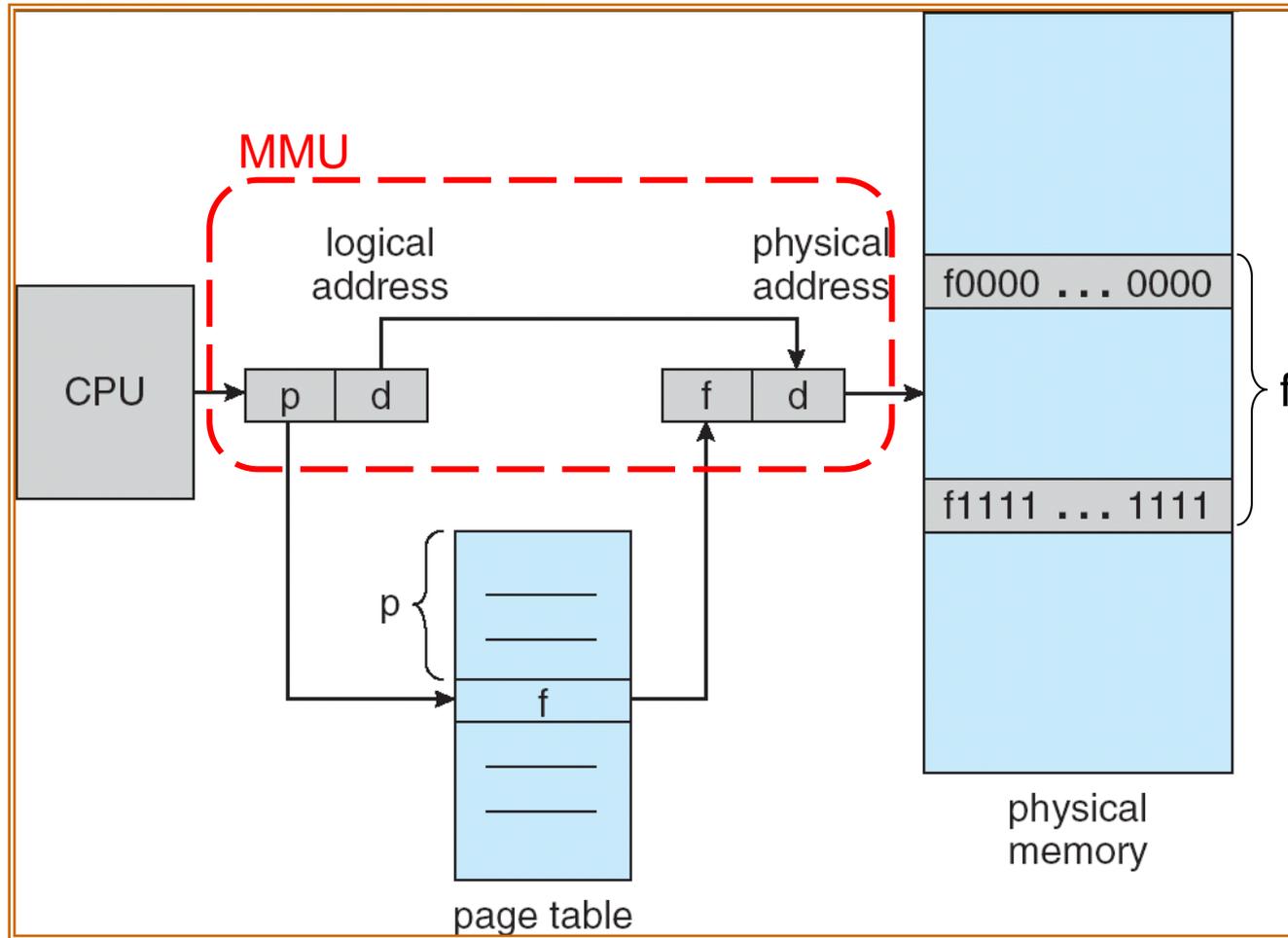
Paging Model of Logical and Physical Memory



.May have internal fragmentation
- average 1/2 page per process

.No external fragmentation
- every frame is useful

Paging Hardware

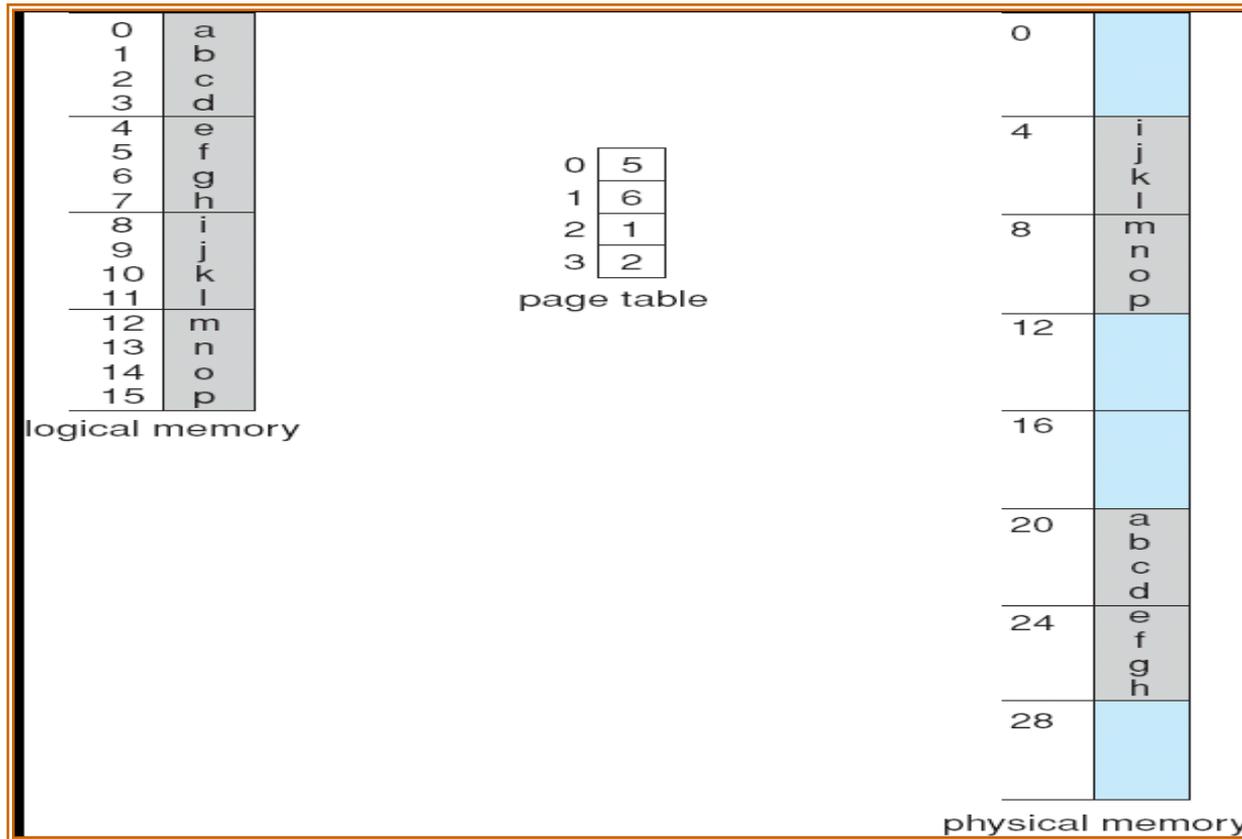


| | |
|-------------------|-------------------|
| p : page number | d : page offset |
|-------------------|-------------------|

$m - n$

n bits

Paging Example (for 4-byte Page)



Logical addr (LA) 0 → page 0, offset 0 → frame 5 → physical addr (PA) $5 * 4 + 0 = 20$

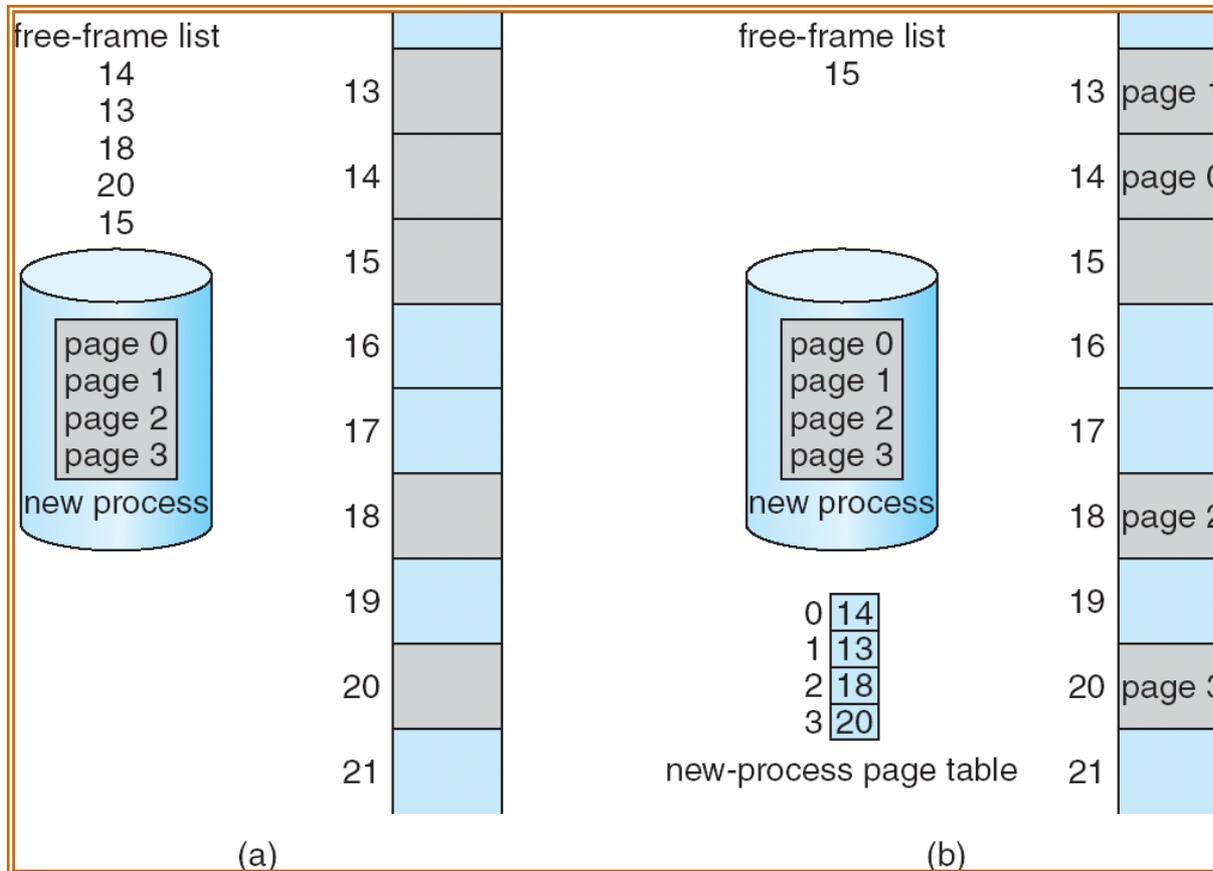
LA = 5 → PA = ?

Page Size

- How to select the page size? large or small?
- Consider internal fragmentation
 - average 1/2 page per process → small page is better
- However
 - Small page → more page table entries (PTEs) for a process → waste space
- Generally, page size increases over time as process (data and code) and physical memory become larger
 - Typically, 4KB or 8KB
- Some CPU supports multiple page sizes
 - e.g., x86 support 2 page sizes: 4KB and 4MB

Allocating Frames to a Process

.A system-wide **free frame list** is maintained by the OS



Before allocation

After allocation

Paging

- Paging provides a clear separation between the user's view of memory and the actual physical memory
 - User program views memory as a **contiguous** space
 - the single space map to **non-contiguous** physical frames by the address translation HW
 - The HW **consults** the page table!!
 - OS **maintains** a page table for each process
 - Base address of the page table is **changed during a context switch**
 - User process has no way of addressing memory outside of its page table

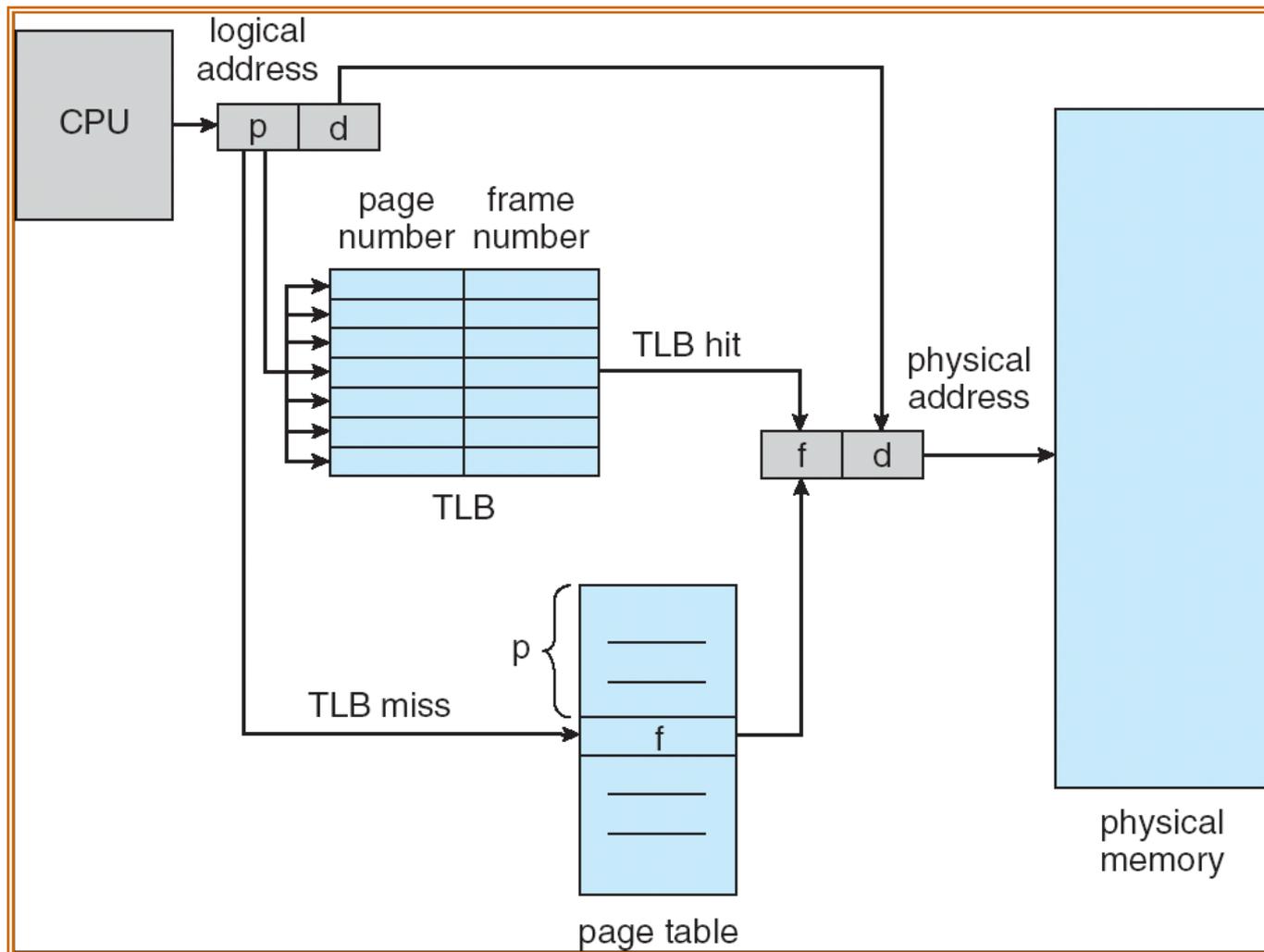
Implementation of Page Table

- Page table is kept in main memory
 - For a 4MB process (page size: 4KB) → **at least** 1K entries
→ register storage is not big enough!!!
- *Page-table base register* (PTBR) points to the page table of the current process
- 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)**

TLB

- A TLB entry has at least two fields
 - Key (tag) : virtual page number
 - value : physical page (frame) number
- Fully associative memory
 - Content search in TLB is done by HW
- Typically, 64 to 1024 entries
 - Cannot contain the full content of a page table, just a **cache**

Paging Hardware with TLB



TLB Miss

- On a TLB miss
 - insert the information about the PTE into the TLB
 - Page number, frame number
 - TLB full?
 - Replacement
 - Random
 - LRU (Least Recently Used)
 - Some TLB entries can be wired-down
 - System will never remove these entries from the TLB
 - Entries for kernel code are typically wired-down

TLB

- TLB needs to be flushed during context switches...
 - Why? E.g., both processes 1 and 2 can access logical addr 1000...
 - Do TLB flushes harmful?
- Add **address space identifier (ASID)** into each TLB entry can **prevent TLB flushes** due to context switches
 - Both tag and ASID should be match
 - Otherwise, TLB miss...

Effective Access Time

- Associative Lookup (TLB) = t time units
- Assume memory cycle time = m time units
- Hit ratio – percentage of times that a page number is found in the associative registers
- Hit ratio = α
- **Effective Access Time (EAT)**

$$\text{EAT} = \alpha (m + t) + (1 - \alpha)(2m + t)$$

e.g., $t = 20\text{ns}$, $m = 100\text{ns}$, $\alpha = 0.8$

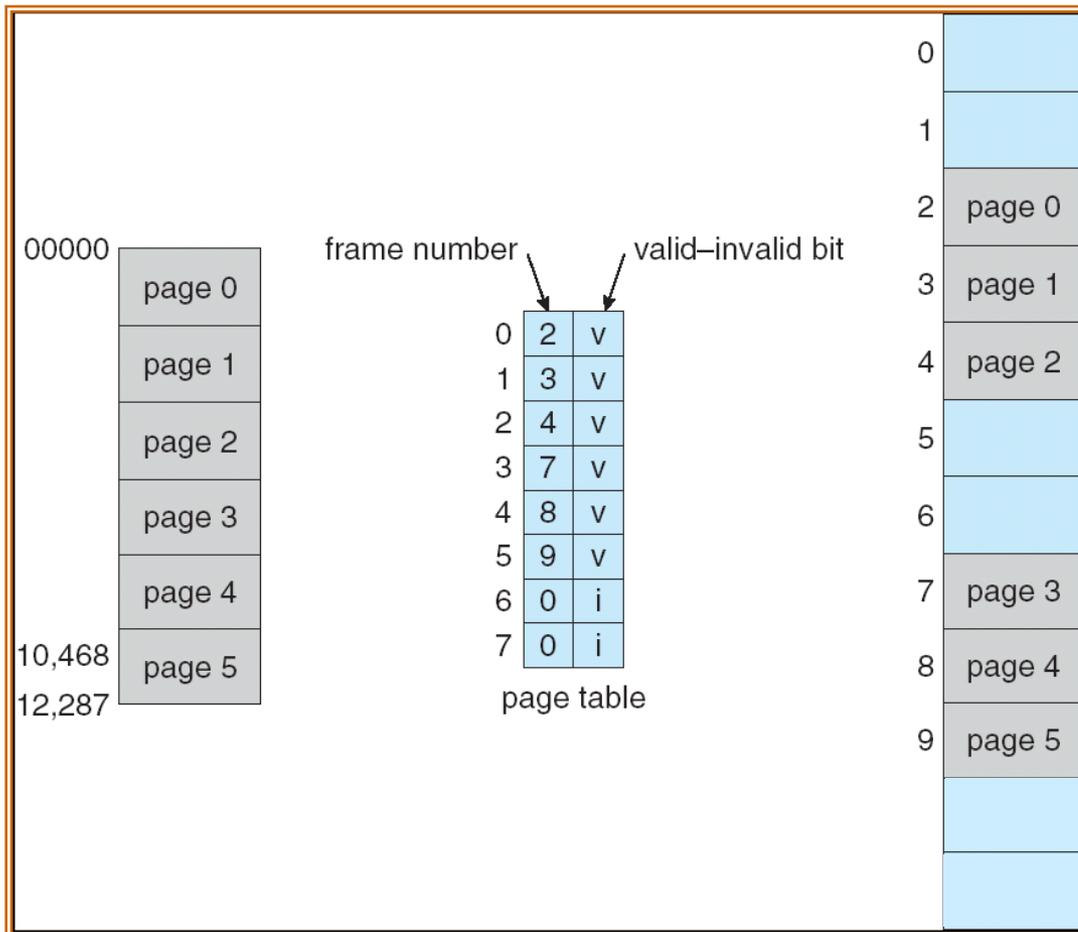
$$\text{EAT} = 0.8 * 120 + 0.2 * 220 = 140\text{ns}$$

Memory Protection

- Memory protection implemented by associating **protection bit** with **each PTE**
 - Read-only, read-write, execute-only
 - Exceptions (**traps**) happen for illegal accesses
 - memory protection violations
- In addition, **valid-invalid bit** attached to **each PTE**:
 - “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) and Invalid (i) Bits in a Page Table

14bit logical address, 2KB page size → page number 0-7



Assume that the process only use addresses 0 - 10468

Valid → do addr. translation

Invalid → trap

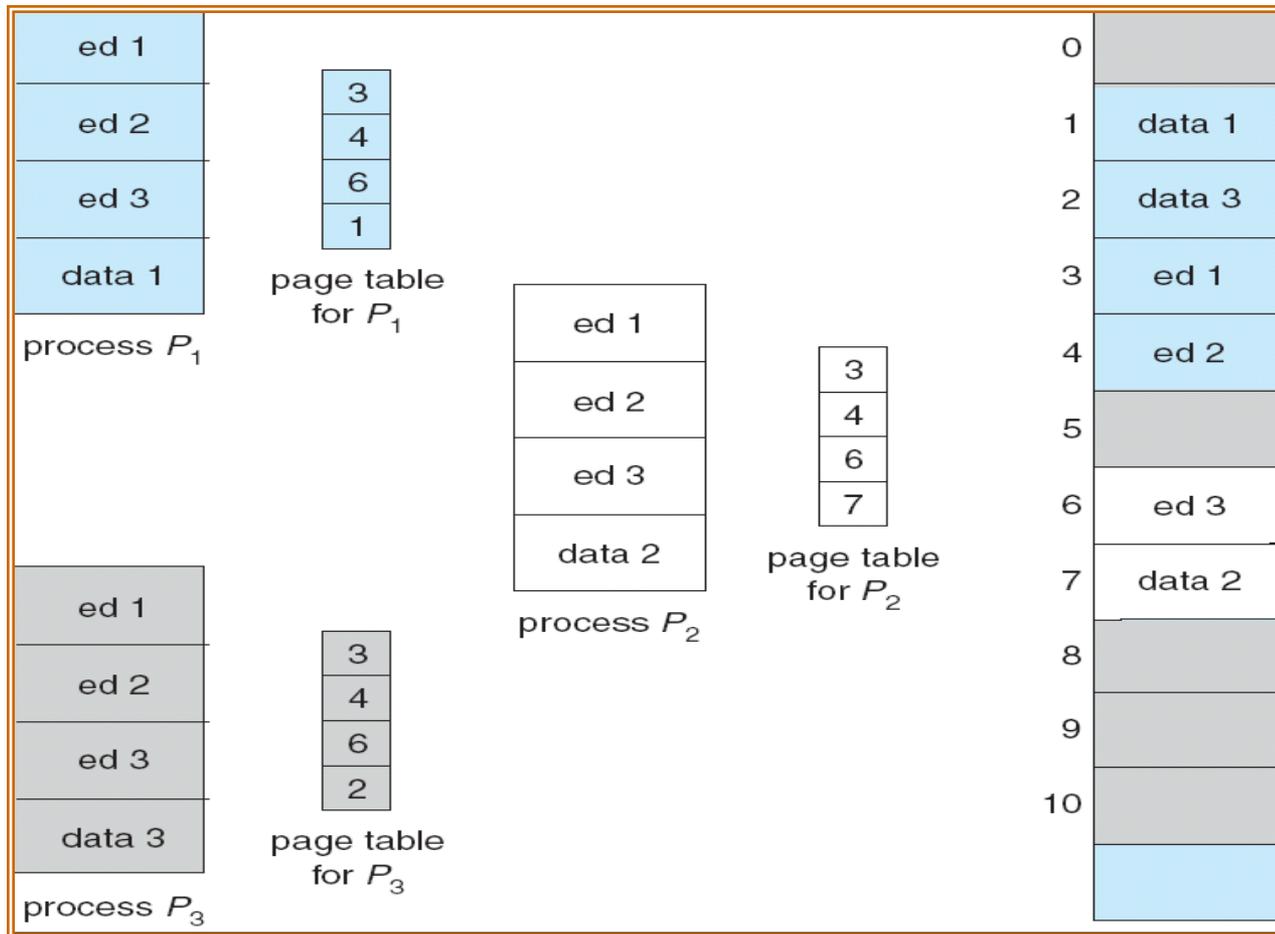
Valid/Invalid Bit in a PTE

- Problems in the previous slide
 - How about accessing address 10469 ?
 - Still OK....but...user program should not access it
 - Most processes use few valid PTEs
 - PTEs for invalid pages should also be presented in memory (to indicate that the corresponding page is invalid)
- A solution: use page table length register (PTLR)
 - Logical address $>$ PTLR \rightarrow trap
 - **Limitation:** PTLR does not help if a process uses both the lowest and the highest memory addresses, even when the process only uses few memory pages

Sharing Code Pages

- Paging enables the possibility of sharing common code
 - One copy of read-only code shared among processes (i.e., text editors, compilers, window systems, **run-time libs**).
 - E.g., 40 users using the same text editor (150KB code and 50KB data)
 - No sharing
 - $(150+50) * 40 = 8000\text{KB}$
 - Code sharing
 - $150 + 50*40 = 2150\text{KB}$ (save almost 3/4 memory space)
- Shared code must be *read-only* and ***reentrant***

Sharing of Code in a Paging Environment



Sharing Data Pages

- Setting up PTE allows sharing data pages
 - Shared memory is implemented by using this technique
- An IPC mechanism

Structures of Page Tables

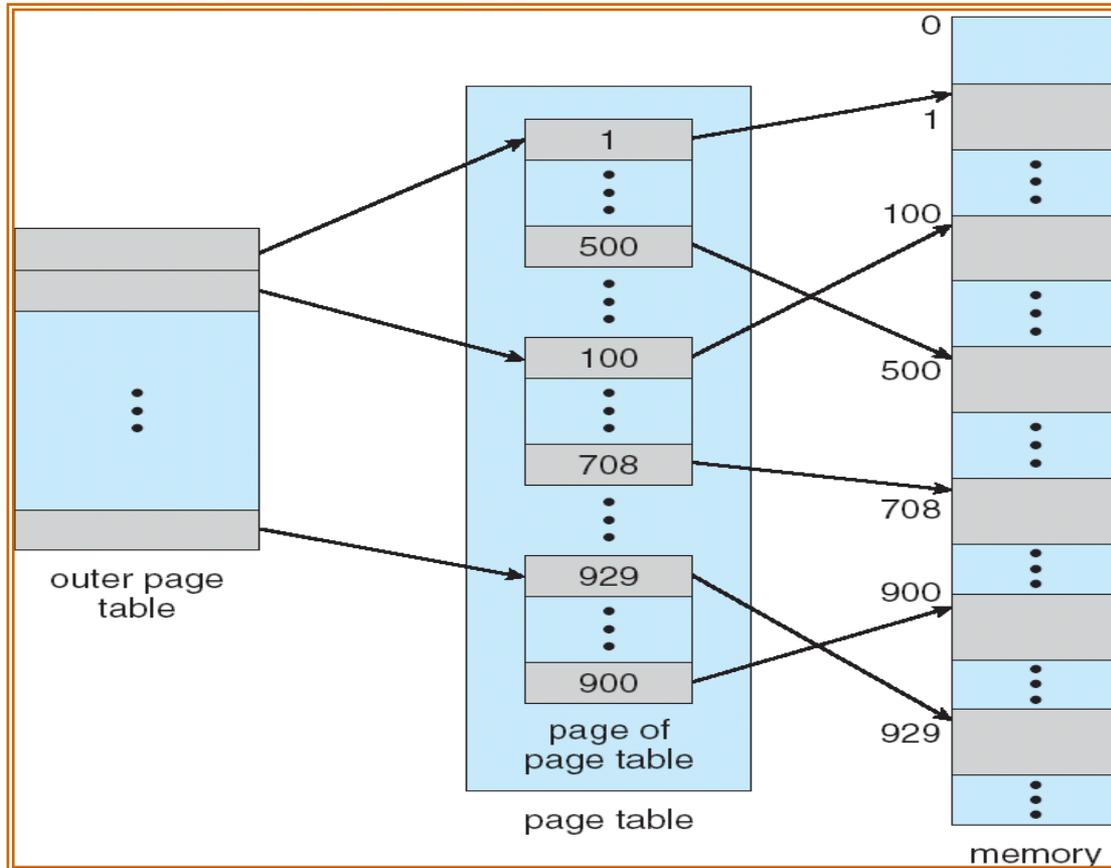


- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Large logical address space
 - $2^{32} - 2^{64}$
 - For 4KB pages, 2^{32} requires **1M PTEs**
 - **Contiguous** memory is required for each PT
 - Required space $1M * \text{PTE size} * \text{process number} !!!!$
- A lot of memory is required for PTs
- Break up the logical address space into multiple page tables to reduce the memory requirement
 - Split a page table into several pages and use extra levels of PTs
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



Page table doesn't need to be contiguous

May use less memory

-not all pages of the table need to be presented!!!

Two-Level Paging Example

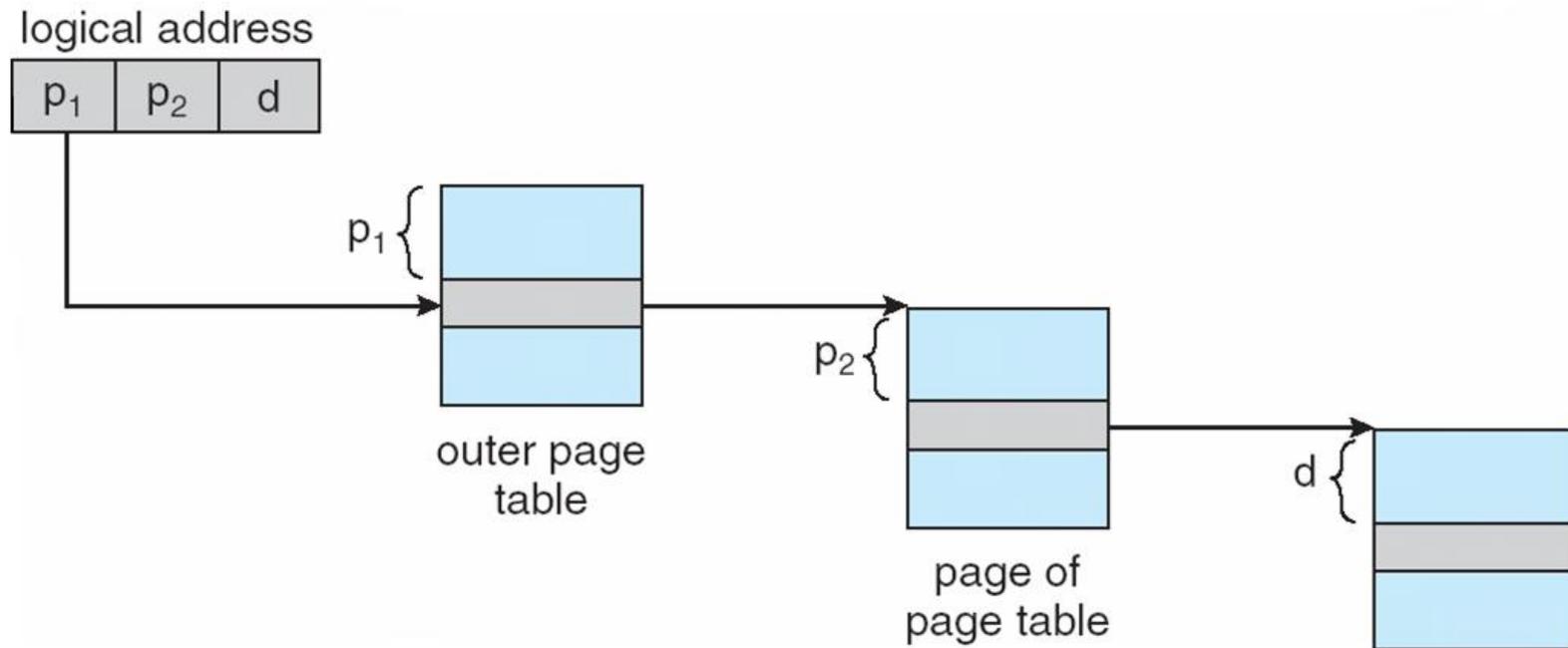
- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a **page number** consisting of 20 bits
 - a **page offset** consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - a 10-bit **page number**
 - a 10-bit **page offset**
- Thus, a logical address is as follows:

| page number | | page offset |
|-------------|-------|-------------|
| p_1 | p_2 | d |
| 10 | 10 | 12 |

where p_1 is an index into the **outer** page table, and p_2 is the offset within the page of the outer page table; p_2 is used to index the (**inner**) page table

Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

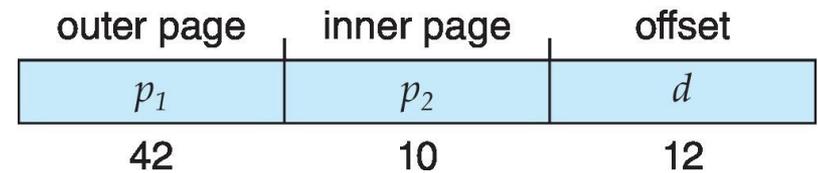


Hierarchical Page Tables

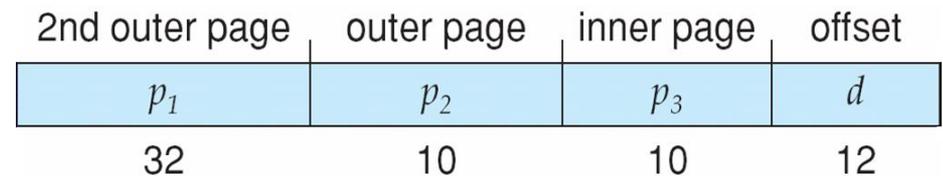
- Some systems use more levels of PT
 - SPARC (32bit) : 3 levels
 - Motorola 68030 (32bit) : 4 levels
- On 64-bit systems, more levels should be used
 - UltraSPARC (64bit) : 7 levels

More than 2-level Paging

- Two-level paging is not sufficient for 64bit systems
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - Address would look like



- Outer page table has 2^{42} entries
- So, add one more level

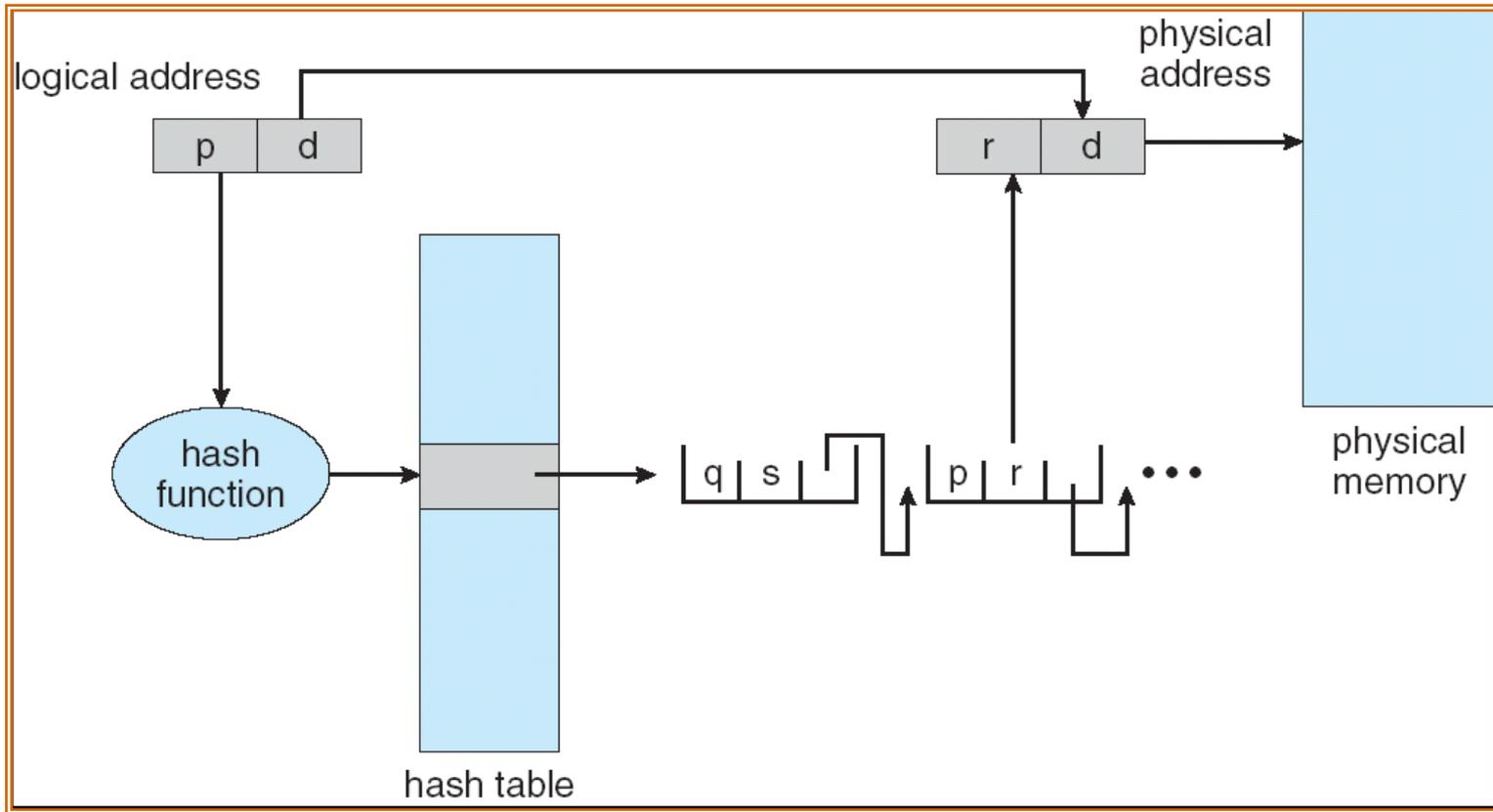


- The 2nd outer page table still has 2^{32} entries
 - And possibly 4 memory access to get to one physical memory location

Hashed Page Tables

- Common in address spaces > 32 bits
- Organize the PT as a hash table
 - Each hash entry
 - Virtual page number
 - Frame number
 - Pointer to the next hash entry
 - Use the **virtual page number** as the **hash key**
 - Elements hashing to the same location are linked together
 - Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

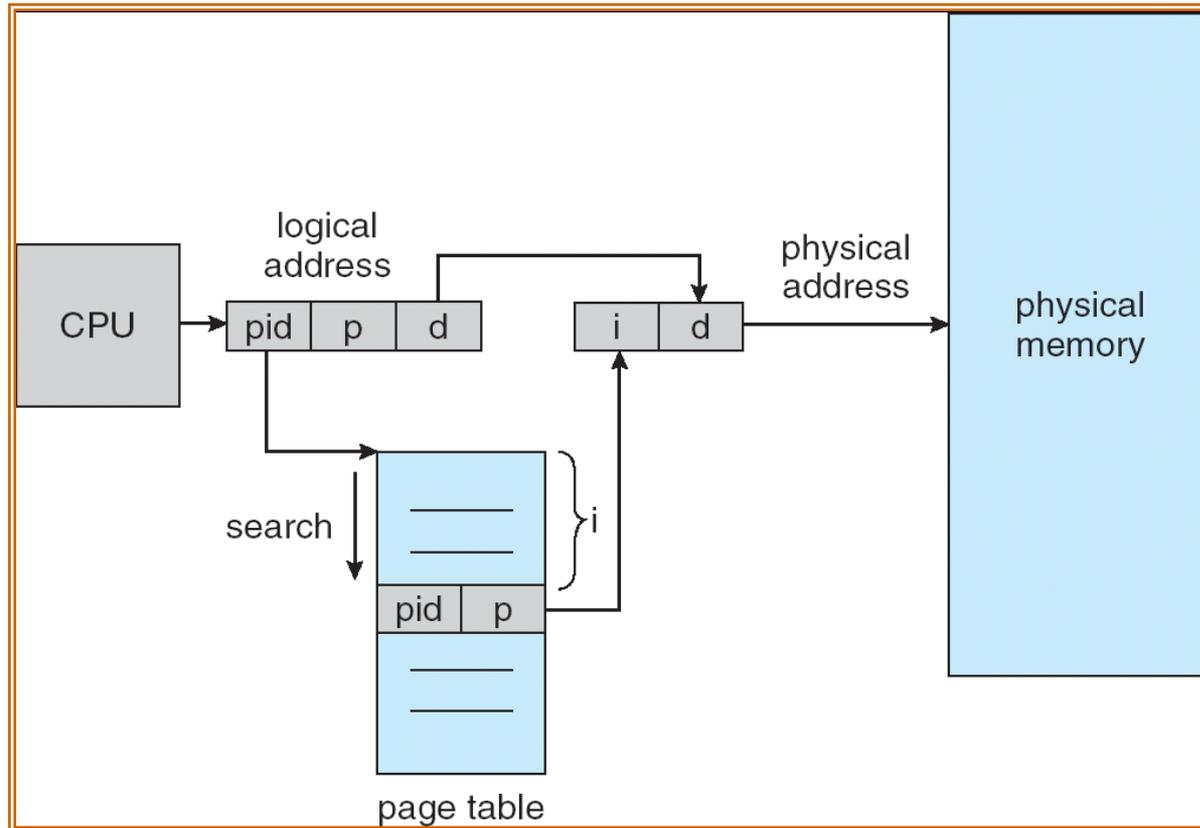
Hashed Page Table



Inverted Page Table

- Use a system wide PT instead of per-process PTs
- One entry for **each page frame**
- Entry consists of
 - the **virtual page** stored in that frame
 - information about the process that owns that page (i.e., **PID**)
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
 - May need to search the whole table
- Use hash table to limit the search to one — or at most a few — page-table entries
- Example: PowerPC

Inverted Page Table Architecture



Each memory access requires n (i.e. PT search) + 1 memory accesses

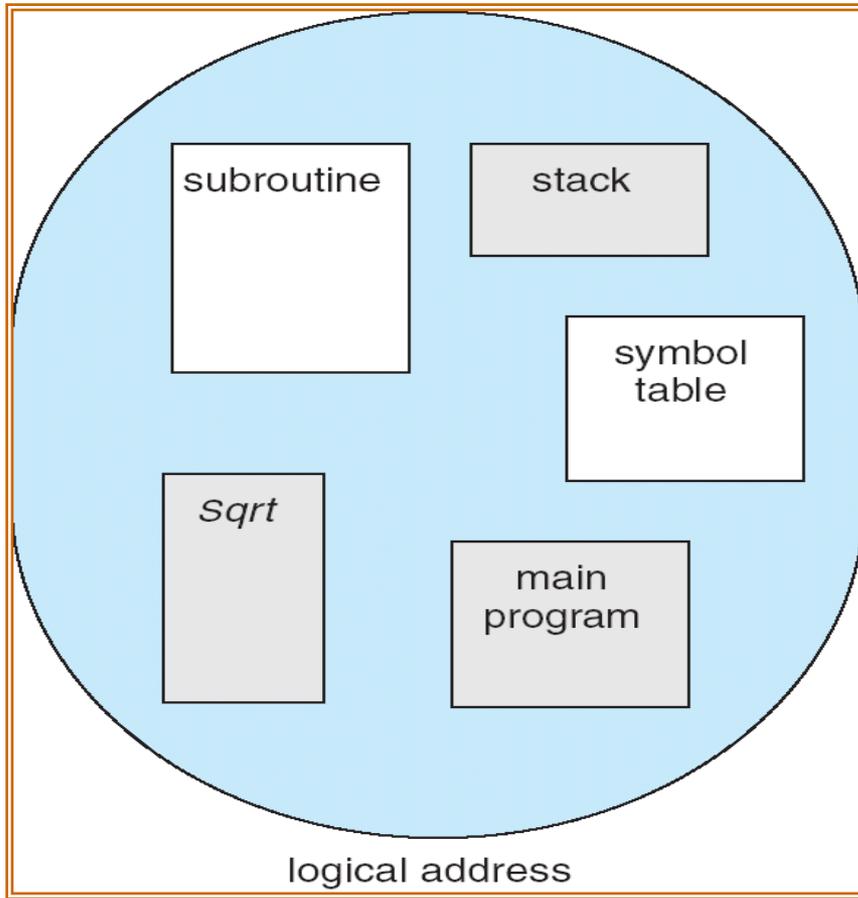
Inverted Page Table

- Difficult to implement shared memory
 - Inverted Page Table (IPT) allows 1:1 mapping between a frame and a virtual page
 - Each IPT entry stores a (PID, logical page) pair
 - However, shared memory requires 1:M mapping
 - Multiple pairs of (PID, logical page) are required for each entry!
- A “solution”
 - Allow one of the virtual page at a time to map the physical page
 - Page fault when the other virtual pages are accessed

Segmentation

- Users view a program as a collection of **variable-sized *segments***, such as
 - main program,
 - procedure, function, method,
 - object,
 - local variables, global variables,
 - common block,
 - stack,
 - symbol table, arrays

User's View of a Program

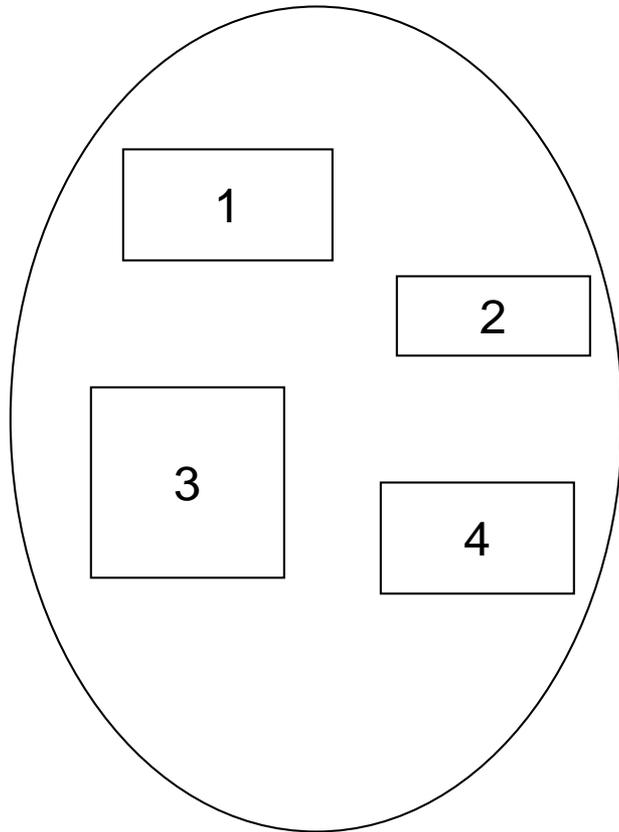


**No address ordering
among the segments**

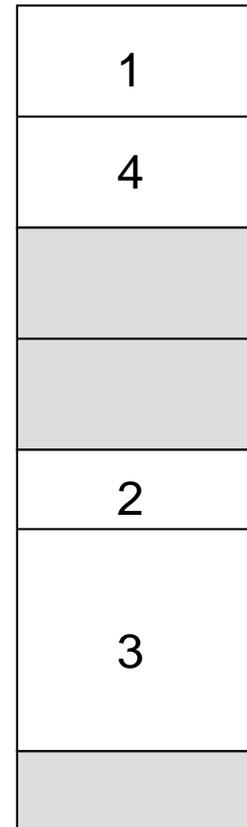
Segmentation

- Normally, **compiler constructs the segments** according to the source programs
 - Code (text)
 - Data
 - Stack
- Loader assigns **identifiers** to the segments
- Segmentation: a memory-management scheme that supports user view of memory

Logical View of Segmentation



Logical space

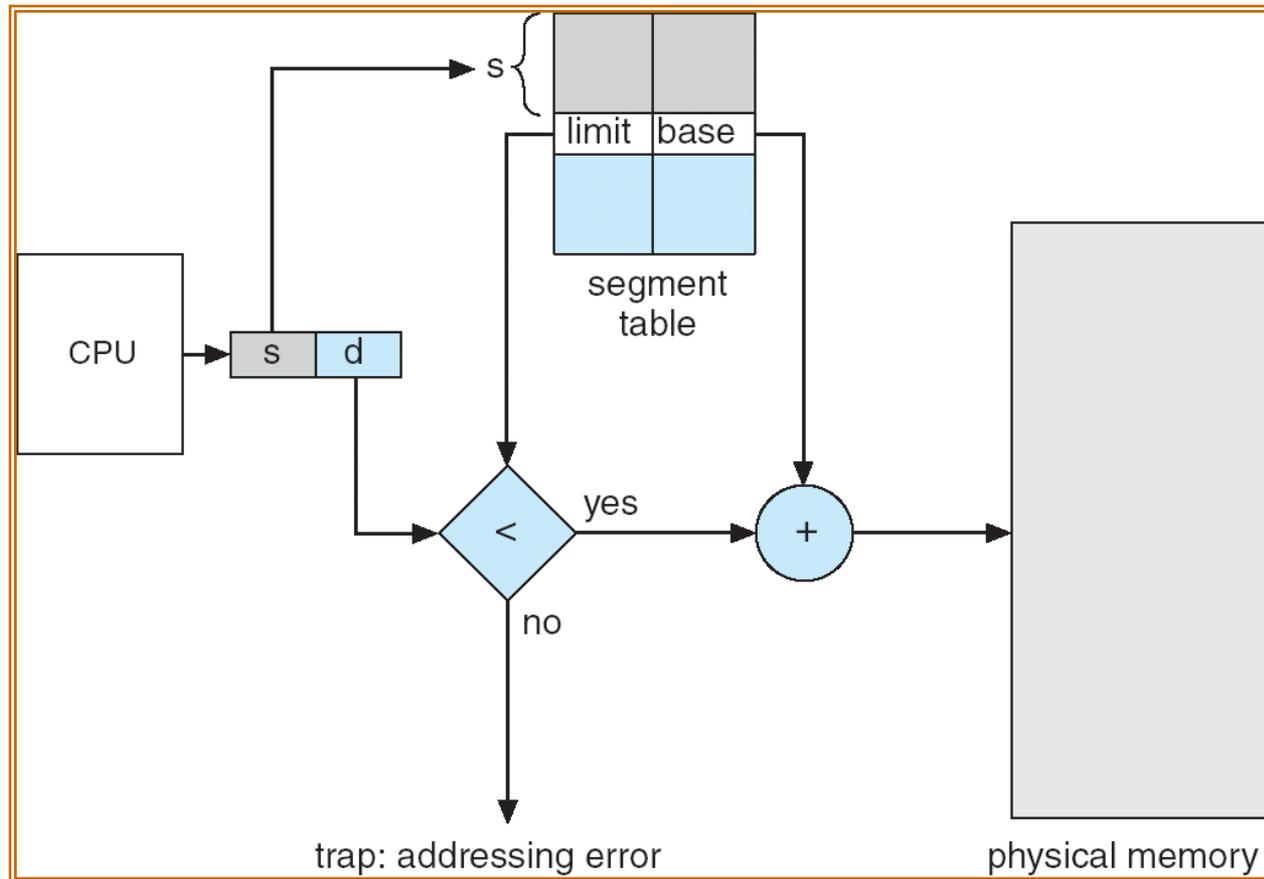


physical memory space

Segmentation Hardware

- Logical address consists of a tuple:
 <segment-number, offset>
- **Segment table** – maps the tuples to 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 $s < \text{STLR}$

Segmentation Hardware



Segmentation Hardware (Cont.)

- **Relocation**

- dynamic
- by segment table (relocated by the base address)

- **Sharing**

- shared segments
- same segment number

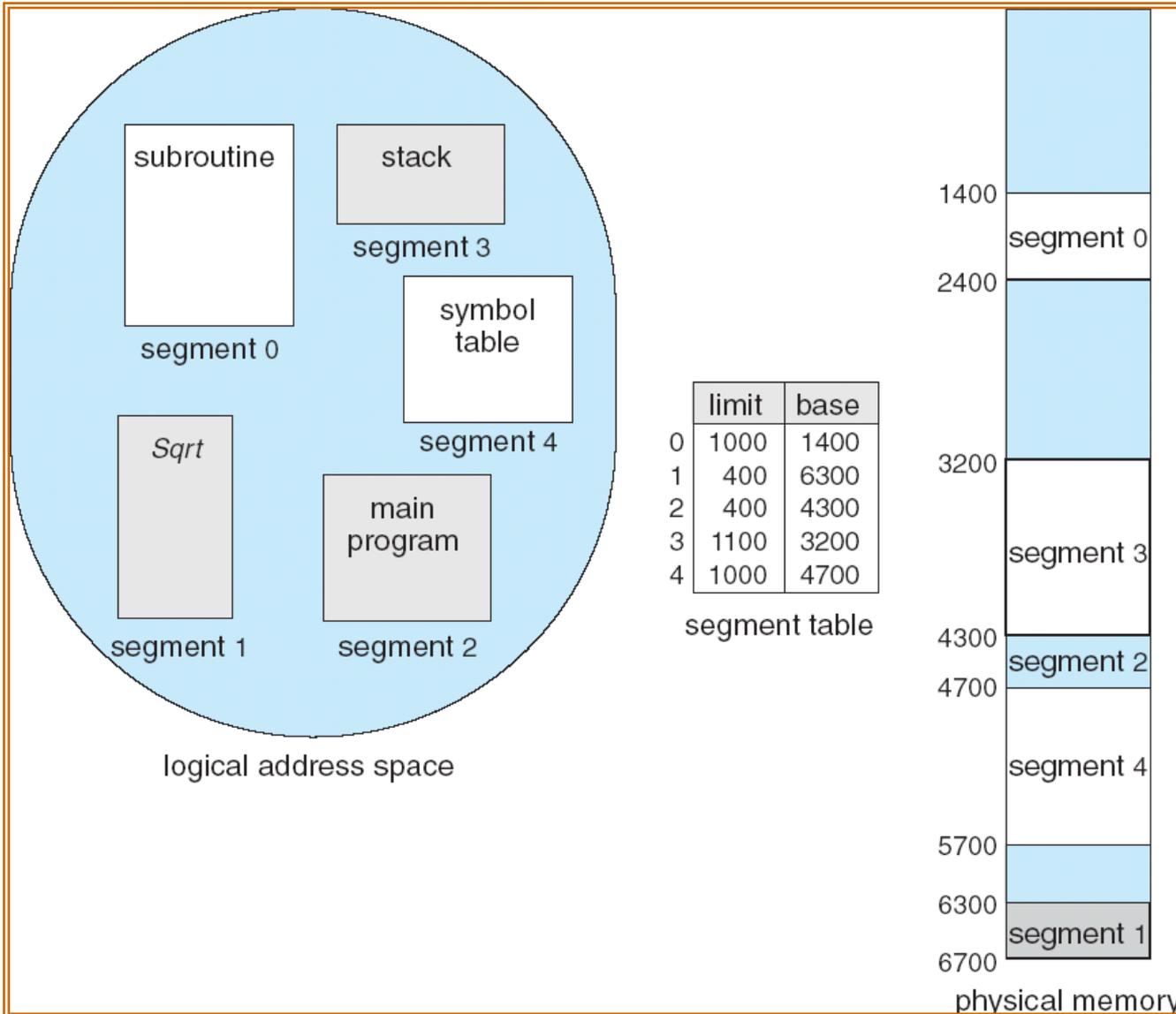
- **Allocation**

- first fit/best fit
- external fragmentation

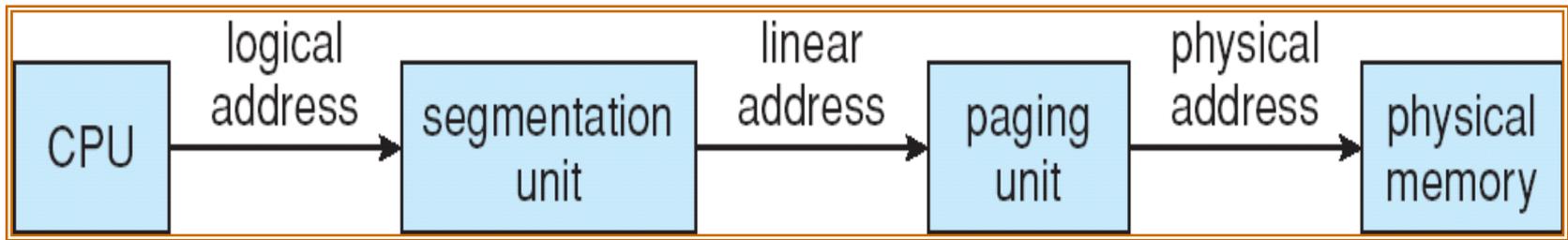
- **Protection**

- each segment table entry associates read/write privileges

Example of Segmentation

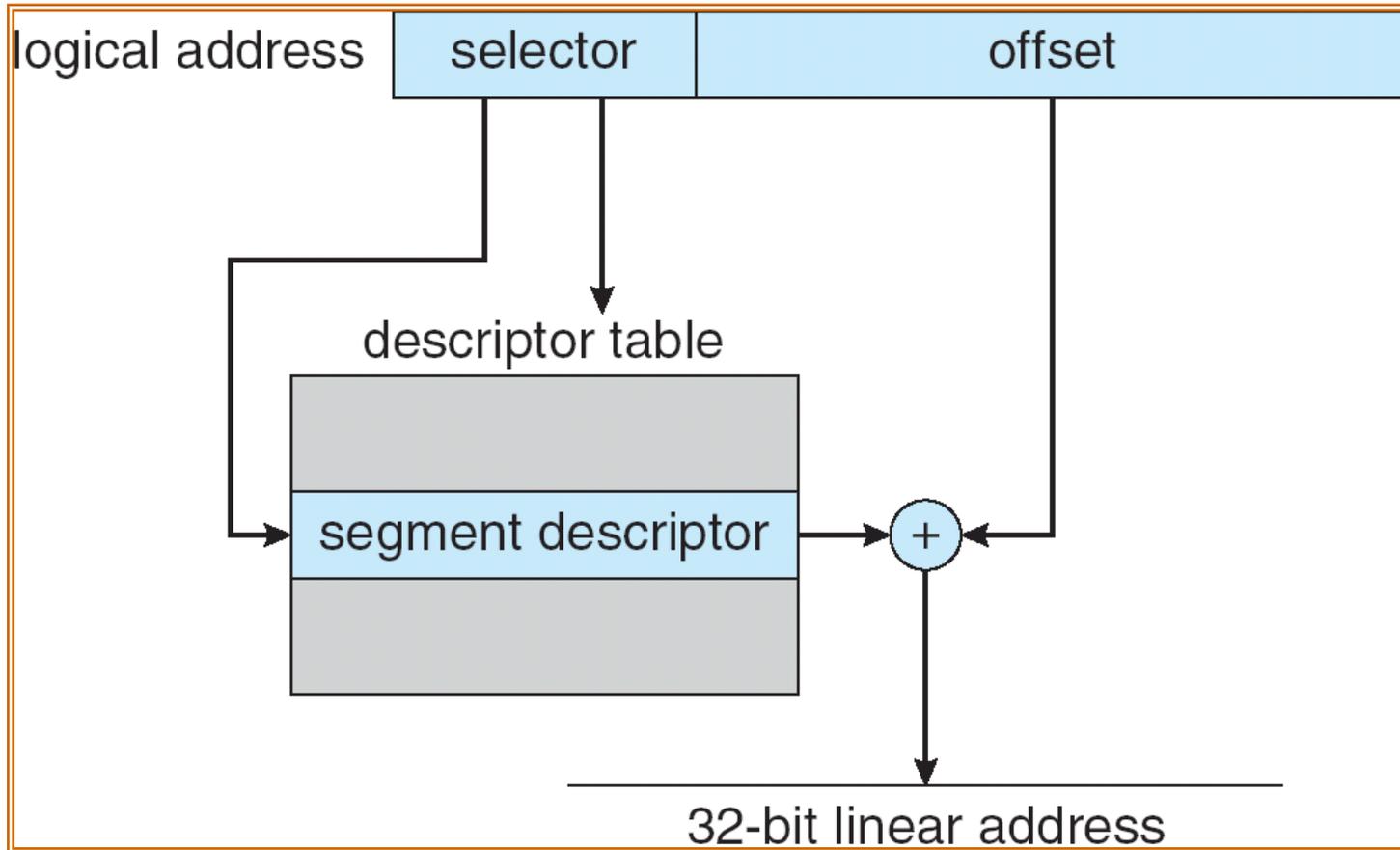


Example: IA-32

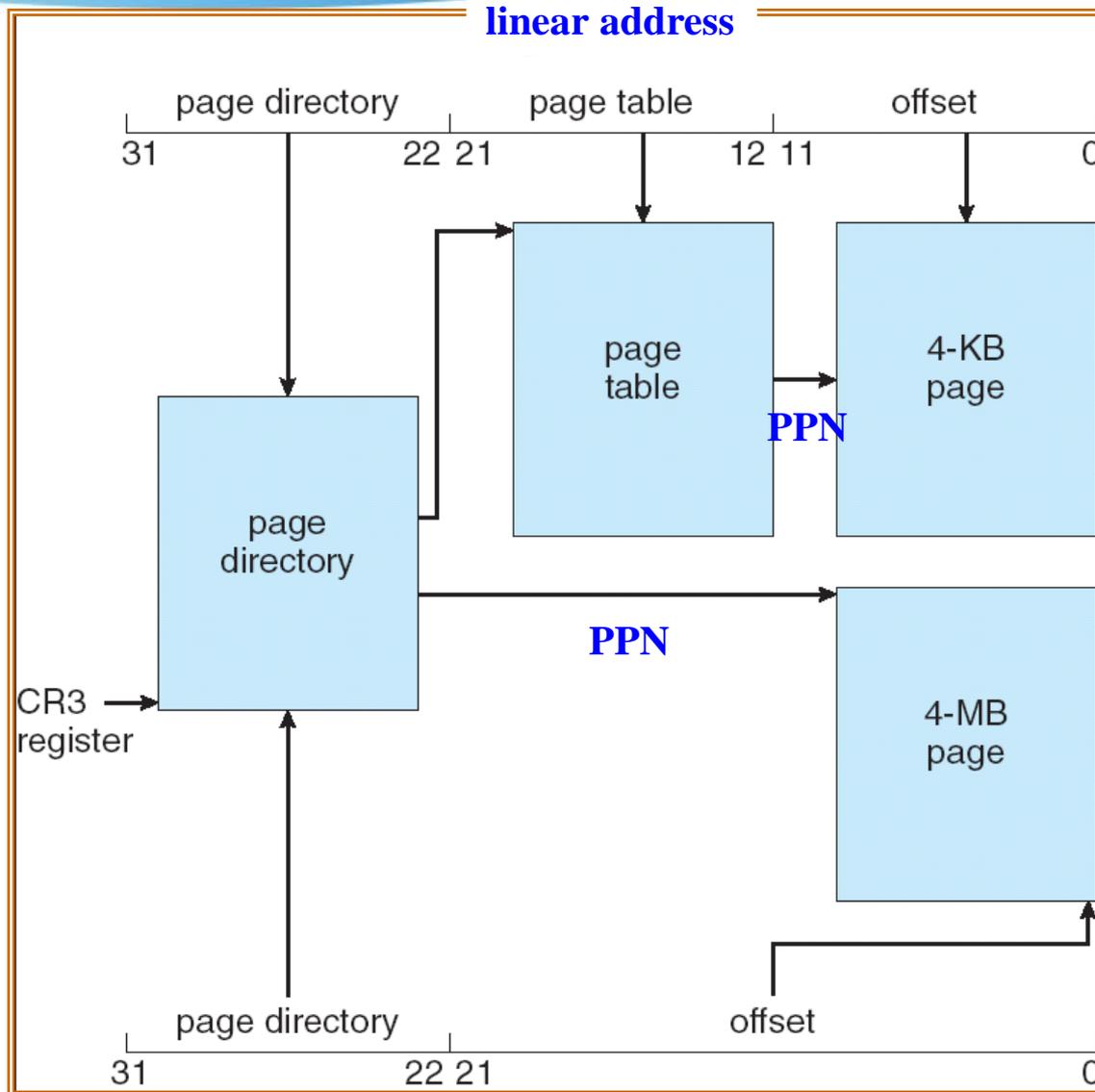


Logical address to physical address translation

IA-32 Segmentation



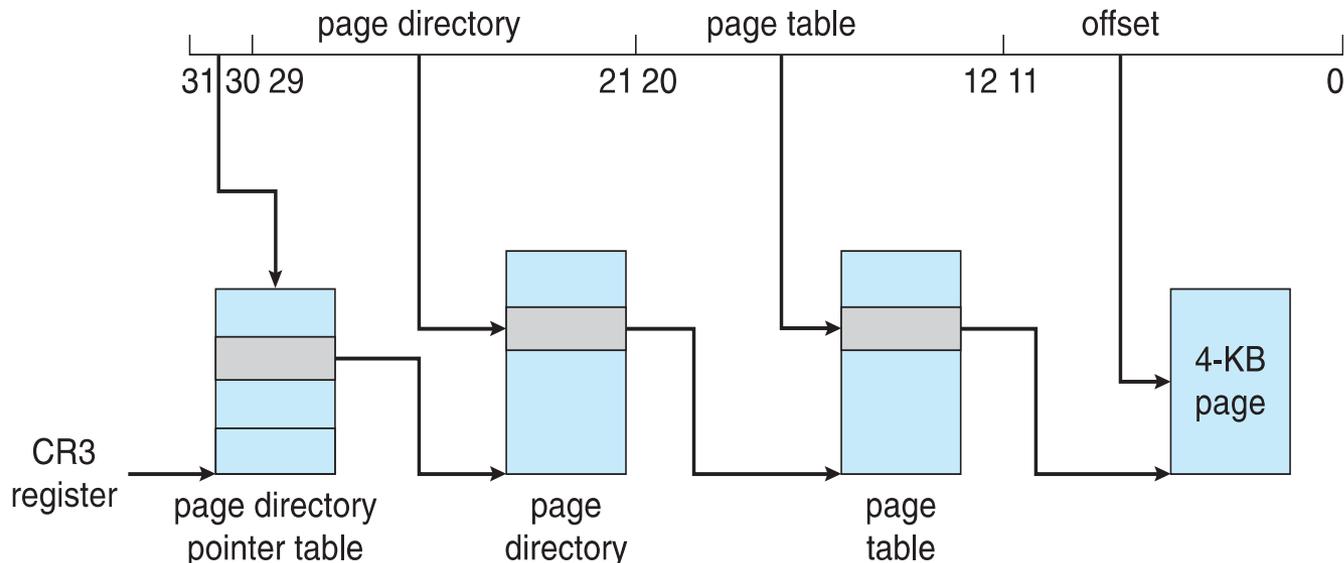
IA-32 Paging



Intel IA-32

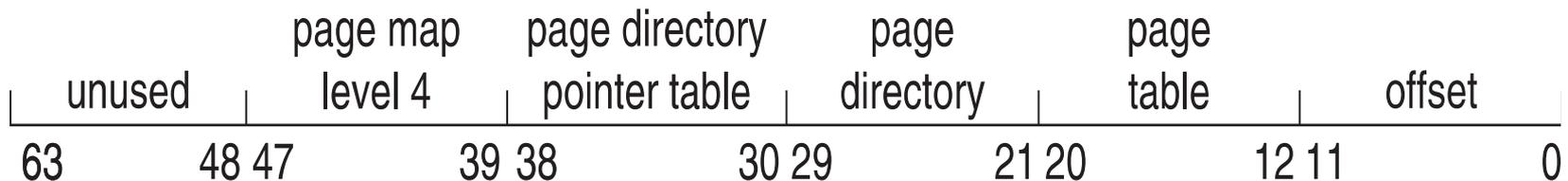
Page Address Extensions

- 32-bit address limits led Intel to create **page address extension (PAE)**, allowing the 32-bit CPU 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**
 - Increasing **physical** address space to 36 bits – 64GB of physical memory
 - **Bits of PT (and page frames) in PTE extend from 20 to 24**
 - Page-directory and page-table entries are 64-bits in size



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



Example: ARM Architecture

- Dominant in mobile platforms
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB small pages
- 1 MB and 16 MB large pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs (similar to two-level cache)
 - Outer level has two **micro TLBs** (one data, one instruction)
 - Inner is single **main TLB**
 - First **outer TLB** is checked, on miss **inner TLB** is checked, and on miss page table walk performed by CPU

ARM Paging

