Chapter 9

Main Memory

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Agenda

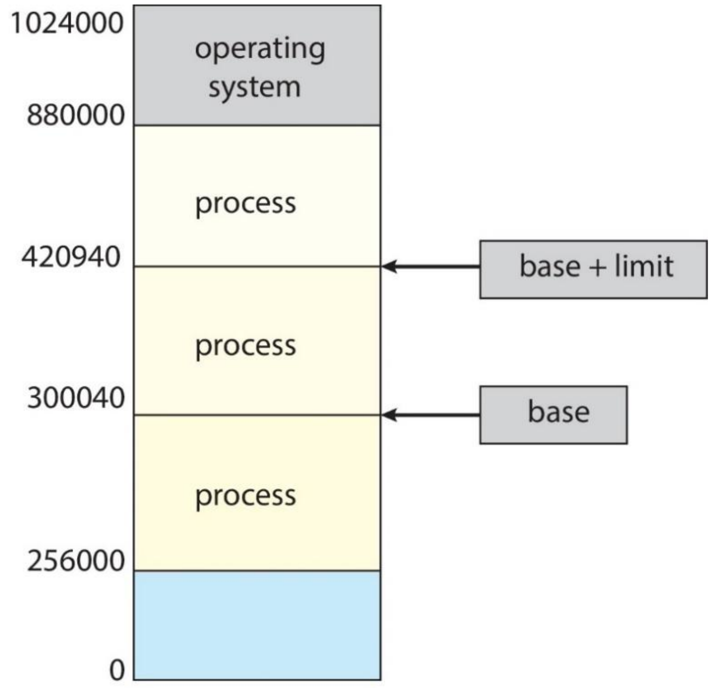
- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architecture

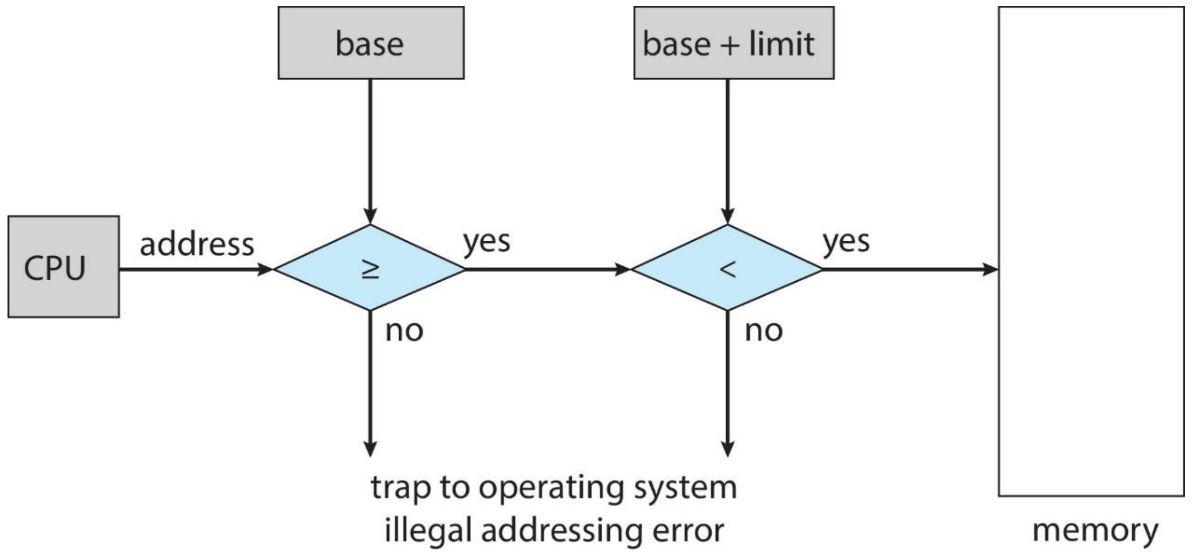
Background

- 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
- 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**
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Basic Hardware

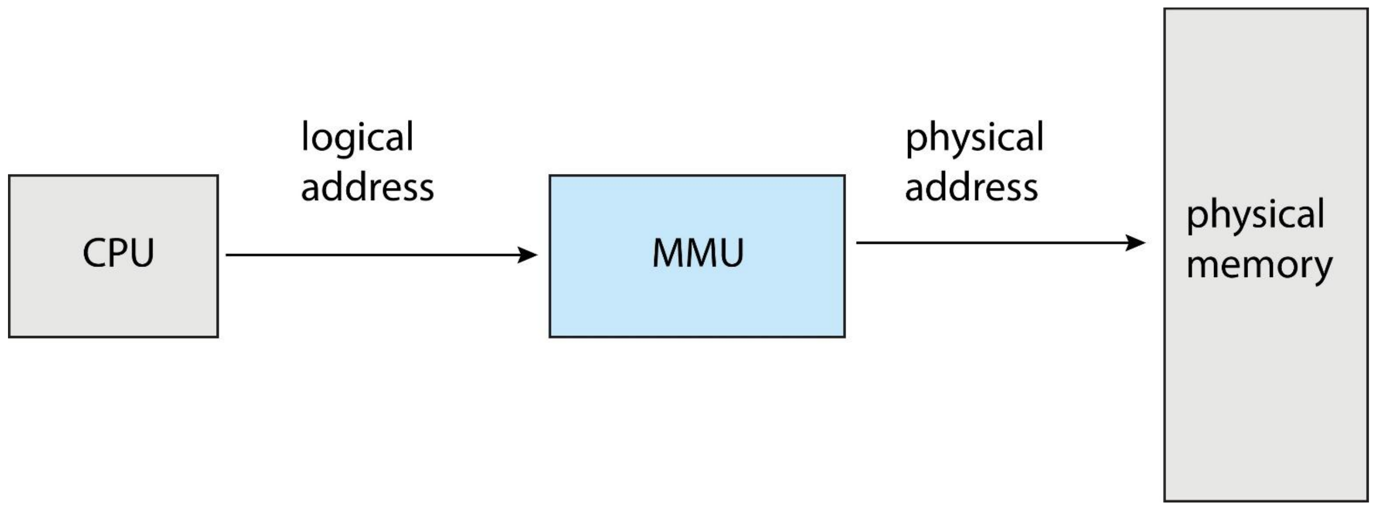
- Protection from other processes
  - Each process has a separate memory space
  - Can be implemented by base register and limit register
  - CPU H/W compares every address generated
  - Base/limit registers can be loaded by privileged instruction





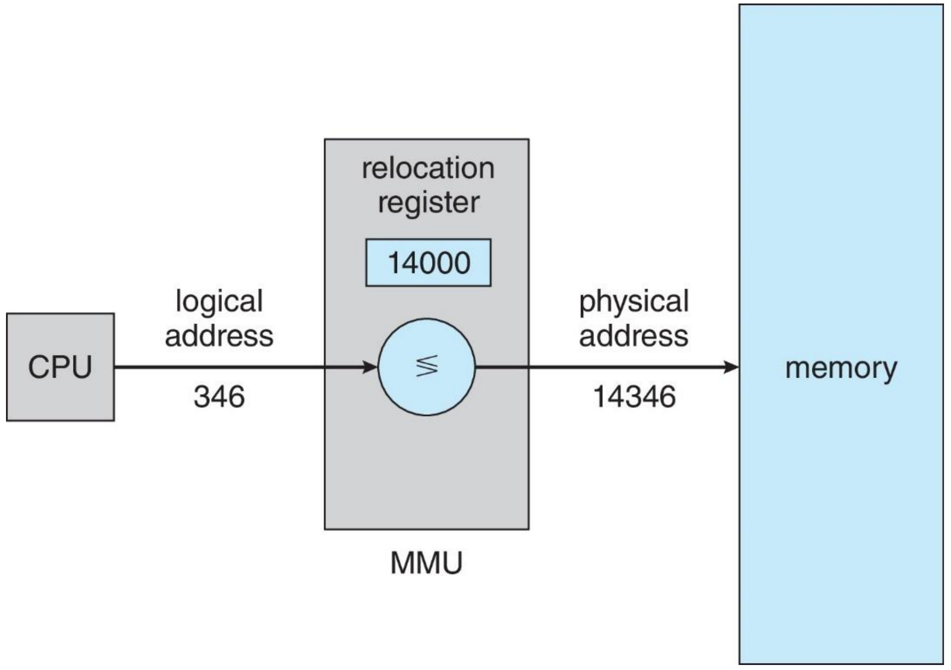
Logical vs. Physical Address Space

- Address
  - **Logical address**: generated by the CPU; also referred to as virtual address
  - **Physical address**: address seen by the memory unit
- **Memory-management unit**: Hardware device that maps virtual address to physical address at run time



Logical vs. Physical Address Space

- A simple method of address mapping using relocation register (base 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



Dynamic Loading

- If the entire program and data must be in physical memory, the size of process is limited by that of physical memory
- With dynamic loading
  - A routine is not loaded until it is called
  - When a routine is called …
    - Calling routine checks whether the routine is loaded or not
    - If not, relocatable linking loader loads the routine and update the program's address tables
    - Then the control is passed to the routine
- Advantage
  - Unused routine is never loaded → better memory-space utilization
  - Does not require special support from OS
    - However, OS can support by providing library routines

Dynamic Linking

- **Static linking**: system libraries and program code combined by the loader into the binary program image
- **Dynamic linking**: linking is postponed until execution time
  - Stub: a small piece of code that indicates …
    - How to locate the appropriate library routine
    - How to load the library if it is not ready
  - After a library routine is dynamically linked, that routine is executed directly
    - stub replaces itself with address of the routine and execute it.
- Advantages of dynamic linking
  - Library code can be shared among processes
  - Library update doesn't require re-linking
- Dynamic linking requires help from OS

Agenda

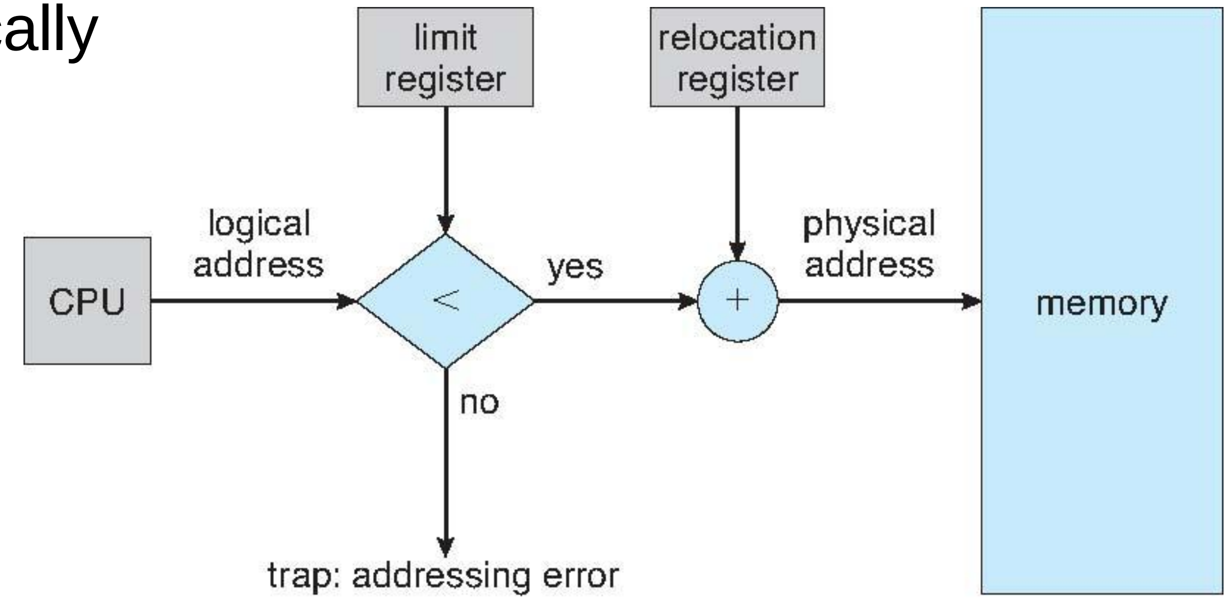
- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architecture

Contiguous Memory 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:
  - Operating system: usually held in high memory
  - User processes: held in low memory
- Each process contained in single contiguous section of memory

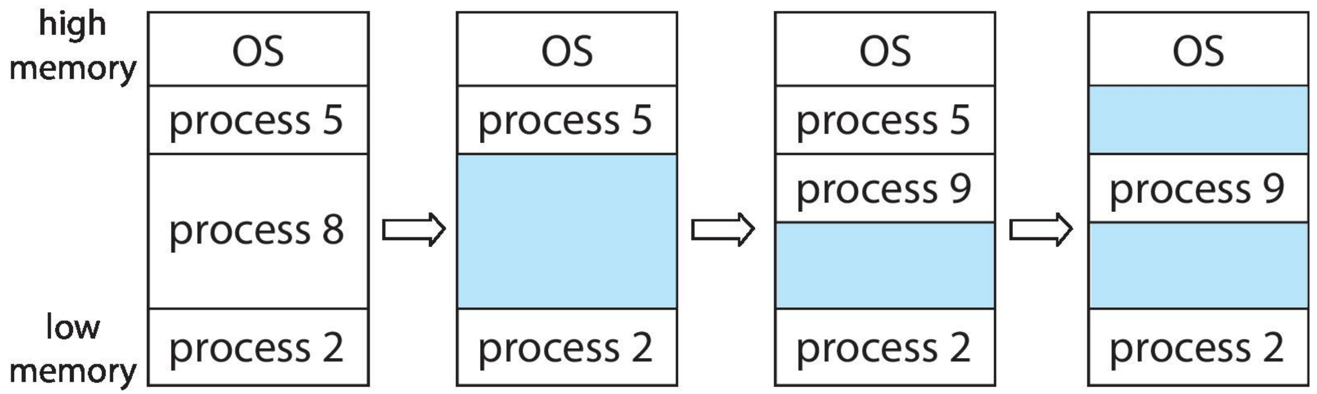
Memory Mapping and Protection

- Relocation registers are 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
  - Relocation and limit registers are loaded by dispatcher during context switching



Variable Partition

- Variable partition scheme
  - **Hole**: block of available memory (holes 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 and adjacent free partitions can be combined
  - Operating system maintains information about:
    - Allocated partitions
    - Free partitions (hole)

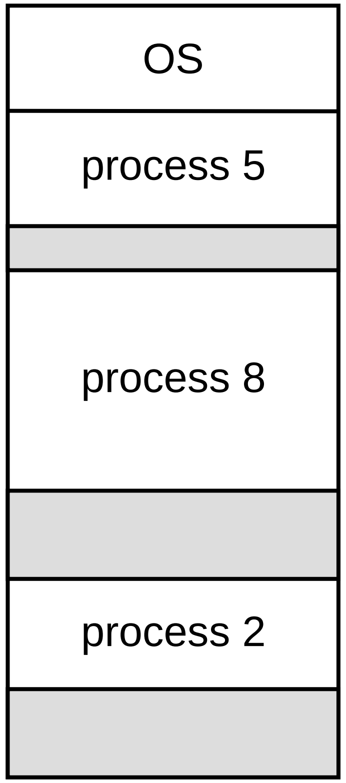


Dynamic Storage Allocation Problem

- Dynamic storage allocation strategies
  - **First-fit**: Allocate the *first* hole that is big enough
  - **Best-fit**: Allocate the *smallest* hole that is big enough
    - We must search entire list, unless the list is sorted by size
    - It produces the smallest leftover hole
  - **Worst-fit**: Allocate the *largest* hole
    - We must also search entire list, unless the list is sorted by size
    - It produces the largest leftover hole
- Which is the best?
  - First-fit and best-fit are better than worst-fit in terms of speed and storage utilization
  - It is not clear whether first fit is better than best fit or not, but first fit is generally faster than best fit

Fragmentation

- **External Fragmentation**: total memory space exists to satisfy a request, but it is not contiguous
  - First-fit and best-fit strategies suffer from external fragmentation
  - Sometimes, it's impossible to allocate large continuous memory although total size of free memory larger than the required memory
  - 50-percent rule (analysis of first fit)
    - Given N allocated blocks, another 0.5N blocks will be lost to fragmentation. (1/3 of memory may be unusable)
- Fragmentation of main memory also affects backing store

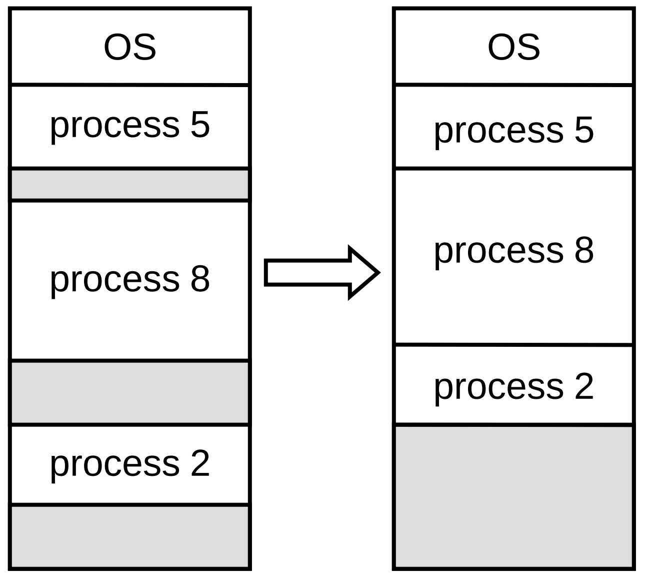


Fragmentation

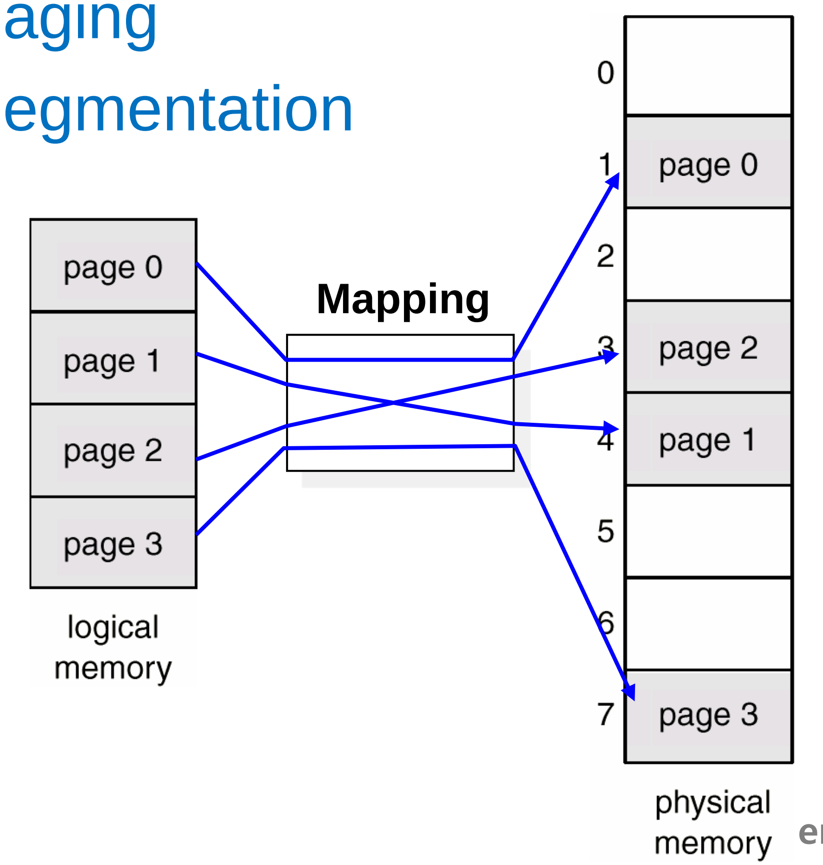
- Memory allocated to a process may slightly larger than requested memory
  - Ex) allocating 18462 bytes from hole whose size is 18464 bytes
  - Reduce overhead to keep small holes
  - Generally, physical memory can be broken into fixed-size blocks
- **Internal fragmentation**: difference between the amounts of allocated memory and the requested memory

Fragmentation

- Remedy of external fragmentation: compaction
  - Possible only if relocation is dynamic and is done at execution time
  - Expensive



- Alternative remedy: permit logical address space of the processes to be noncontiguous
  - Paging
  - Segmentation

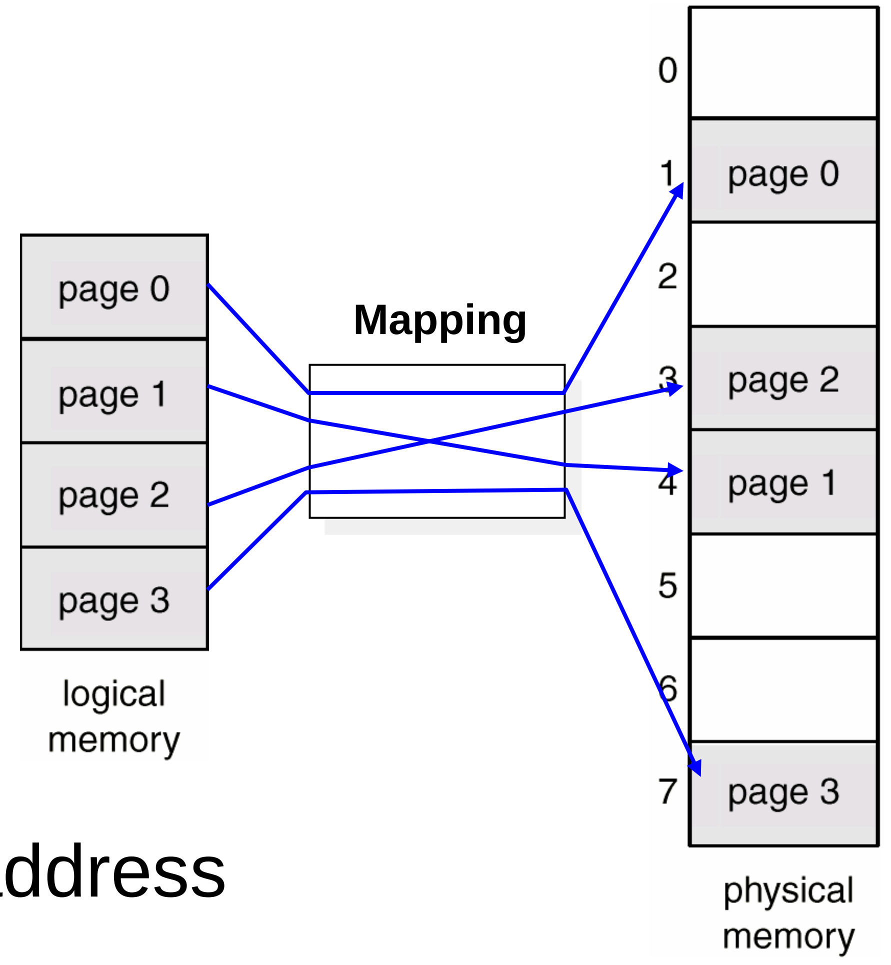


Agenda

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architecture

Paging

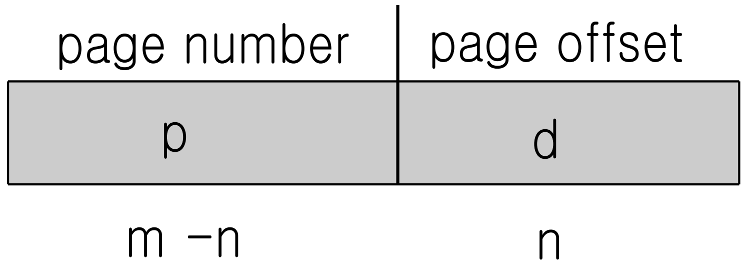
- **Paging**: a memory-management scheme that permits the physical address space of a process to be noncontiguous
- Physical memory: consists of frames (fixed-size blocks)
  - Keep track of all free frames
- Logical memory: consists of pages (fixed-size blocks)
- Pages are mapped with frames through page table → translate logical to physical address



Address Translation Scheme

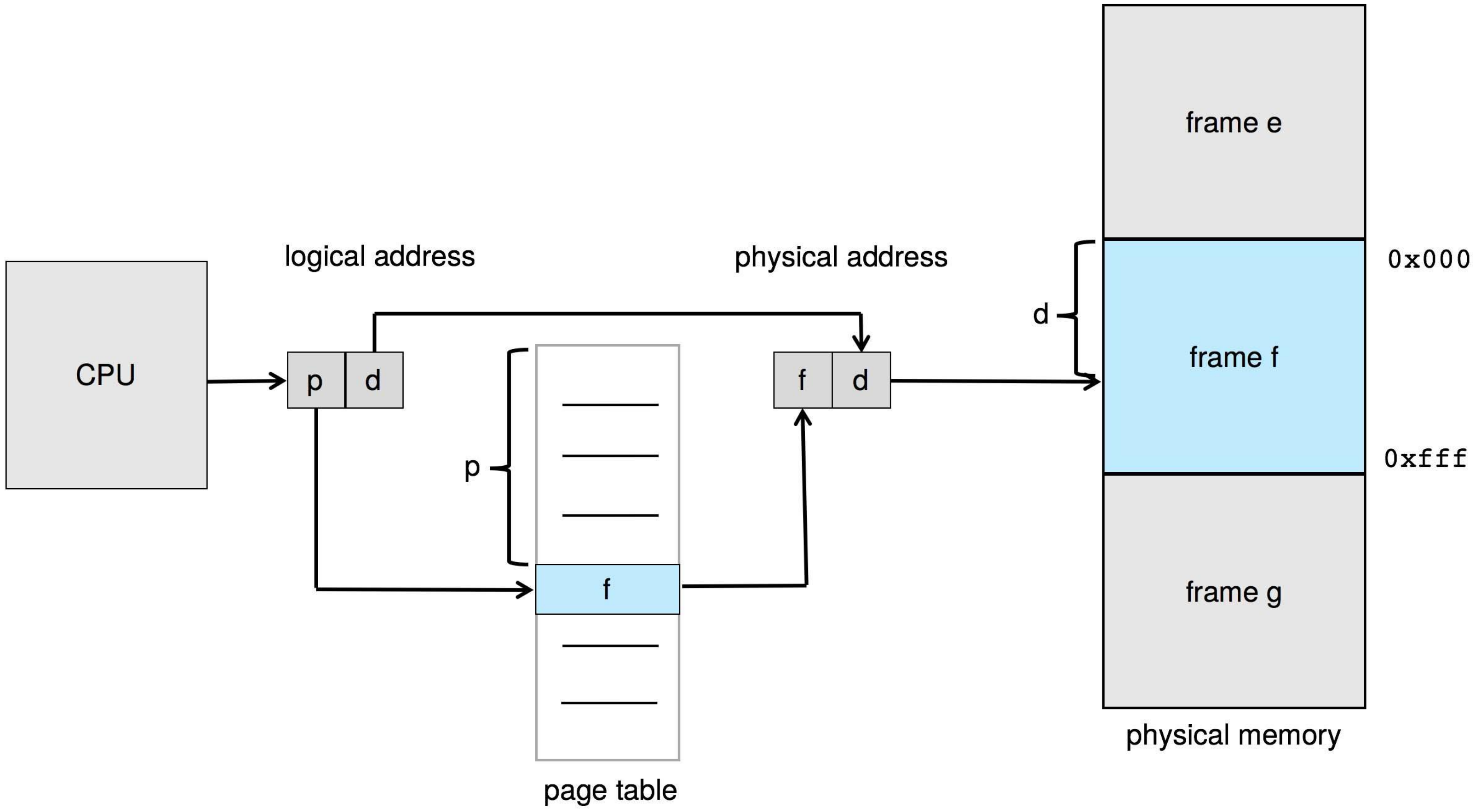
- Every address generated by CPU is divided into:
  - **Page number** (p): used as an index into a per-process page table which contains base address of each page in physical memory
  - **Page offset** (d): location in the frame being referenced

  → The base address of the frame is combined with the page offset to define the physical memory address that is sent to the memory unit



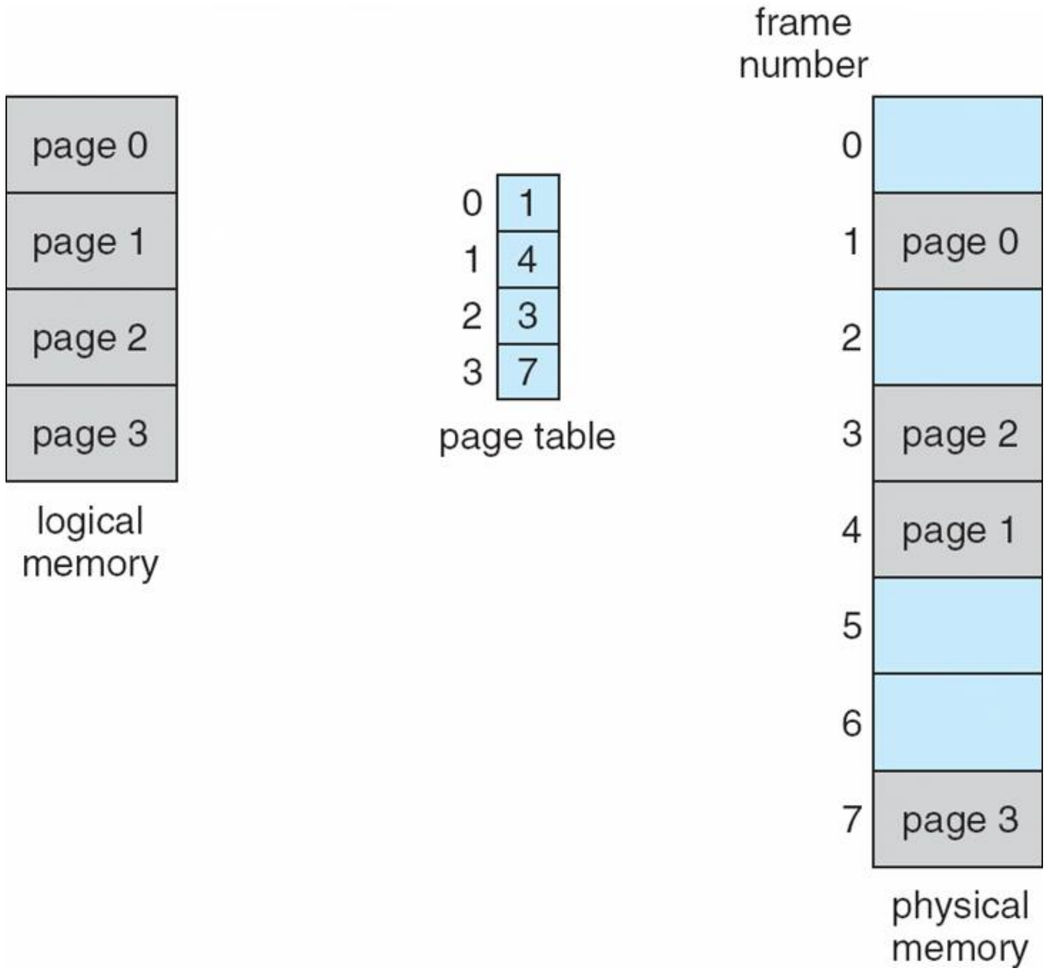
  - Size of logical address space: 2^m
  - Page size: 2^n

Paging Hardware



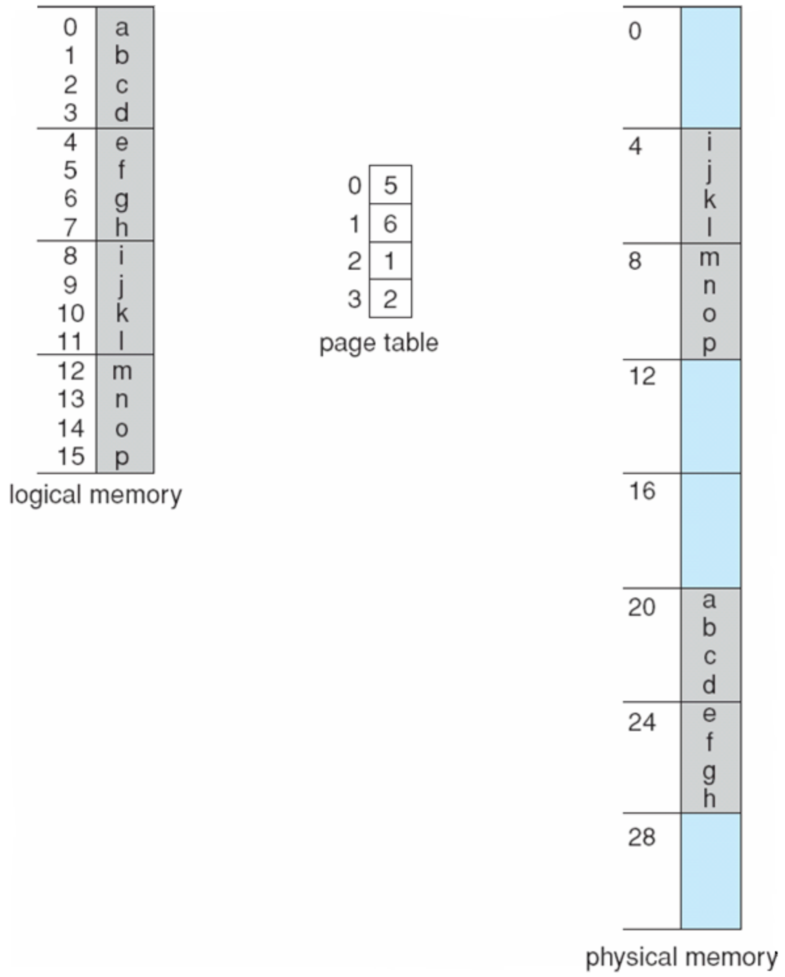
Paging Model

- 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)



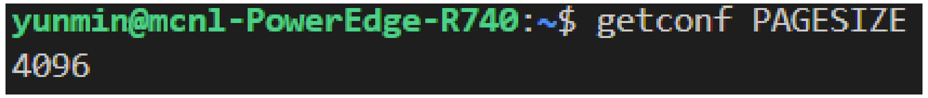
Paging Example

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size

So small page sizes are desirable?

Overhead of Paging

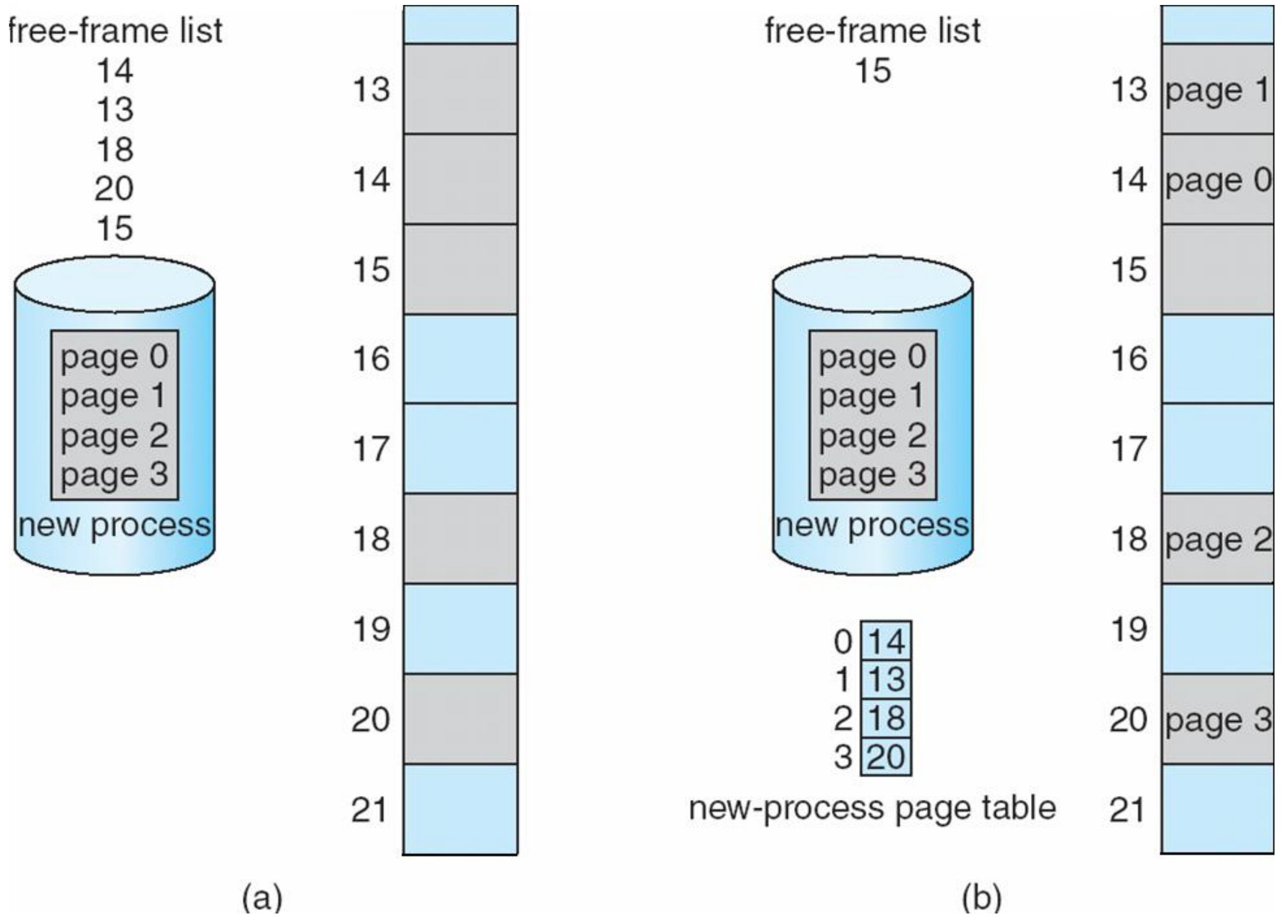
- Each page table entry takes memory to track
  - This overhead is reduced as the size of the page increases
- Disk I/O is more efficient when the amount of data being transferred is larger (Chapter 11)
- Page sizes have grown over time
  - Today, pages are typically either 4KB or 8KB
  - Some CPUs and operating systems support multiple page sizes
    - Windows 10 supports page sizes of 4KB and 2MB
    - Linux supports two page size: 4KB(default) and huge page (architecture-dependent)



```
yunmin@mcnl-PowerEdge-R740:~$ getconf PAGESIZE
4096
```

Free Frames

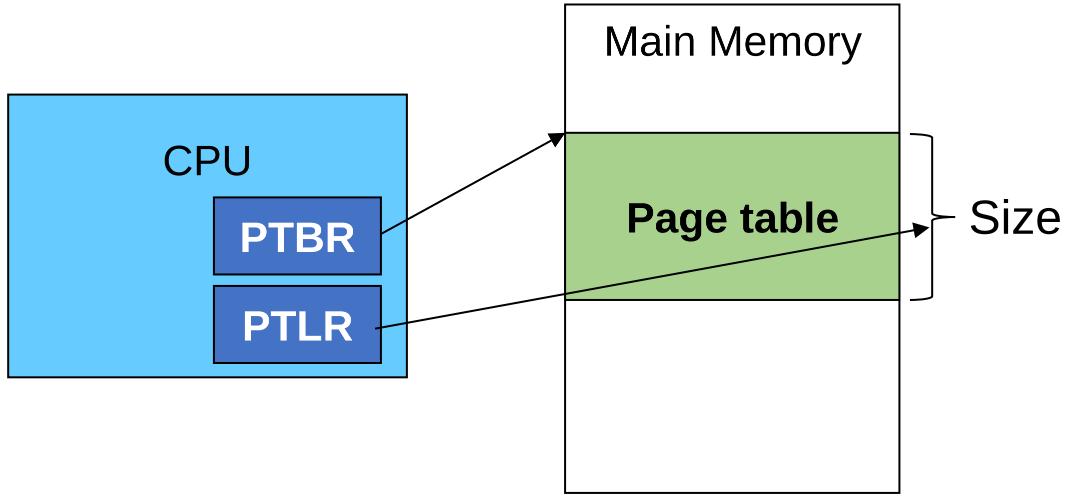
- OS keeps information about frames in a frame table
  - # of total fames
  - Which frames are allocated
  - Which frames are available
  - ETC.
- OS maintains list of free frames



Before allocation

After allocation

Hardware Support



- Page table is stored in main memory
  - **Page-table base register** (**PTBR**) points 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 **translation look-aside buffers** (**TLBs**) (also called **associative memory**)

Hardware Support

- **Translation look-aside buffer** (**TLB**): small fast look up H/W cache
  - Associative, high-speed memory, consists of pairs (key, value)
    - If page number of logical address is in TLB, delay is less than 10% of unmapped memory access
    - Otherwise (TLB miss), access page table in main memory
      - Insert (page number, frame number) into TLB
      - If TLB is full, OS select one for replacement
  - Some TLBs store address-space identifiers (ASIDs) in each TLB entry
    - ASID uniquely identifies each process to provide address-space protection for that process
    - If the TLB does not support separate ASIDs, TLB must be flushed at every context switch
  - TLBs typically small (64 to 1,024 entries)

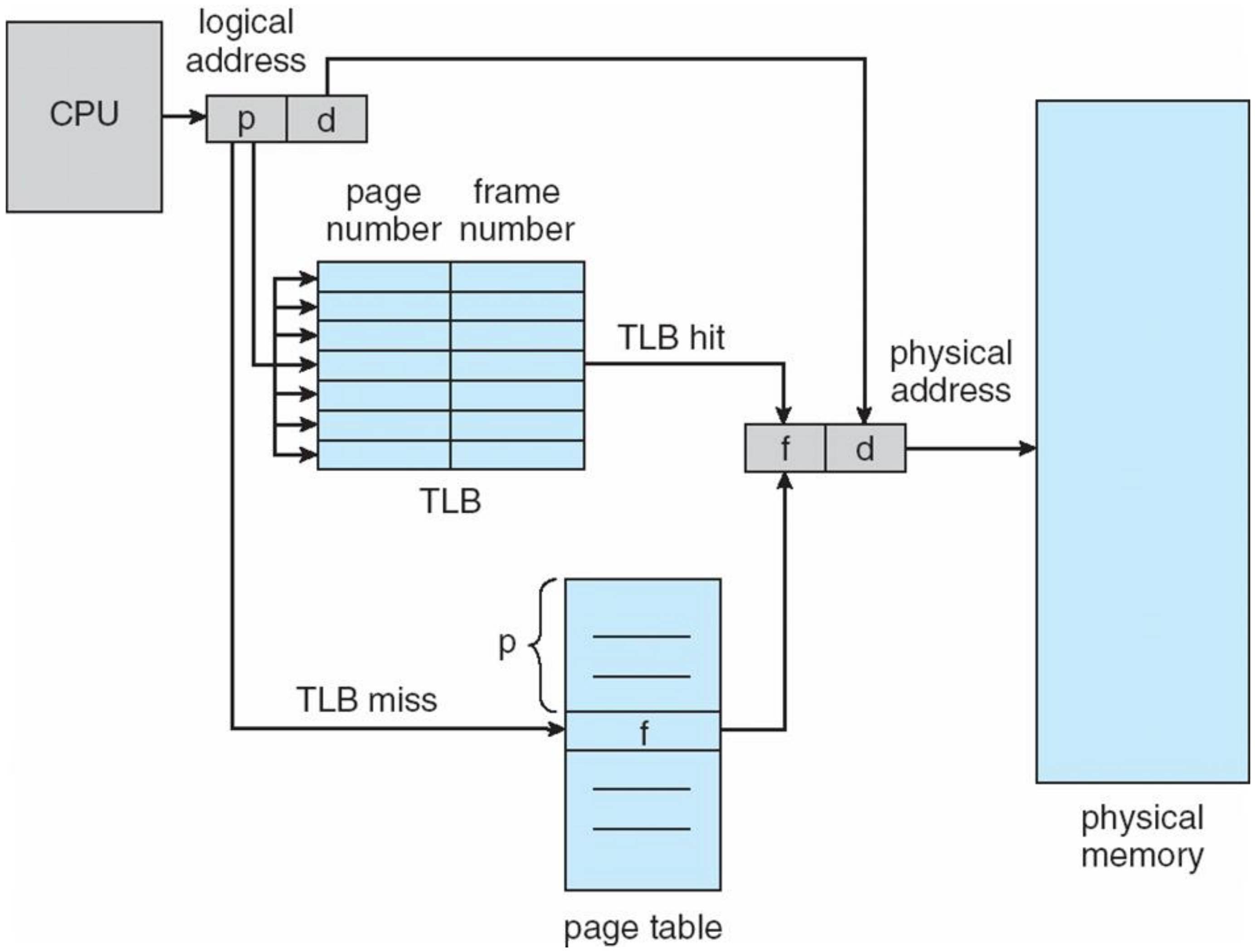
Hardware Support

- 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 a more realistic hit ratio of 99%,

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

    implying only 1% slowdown in access time

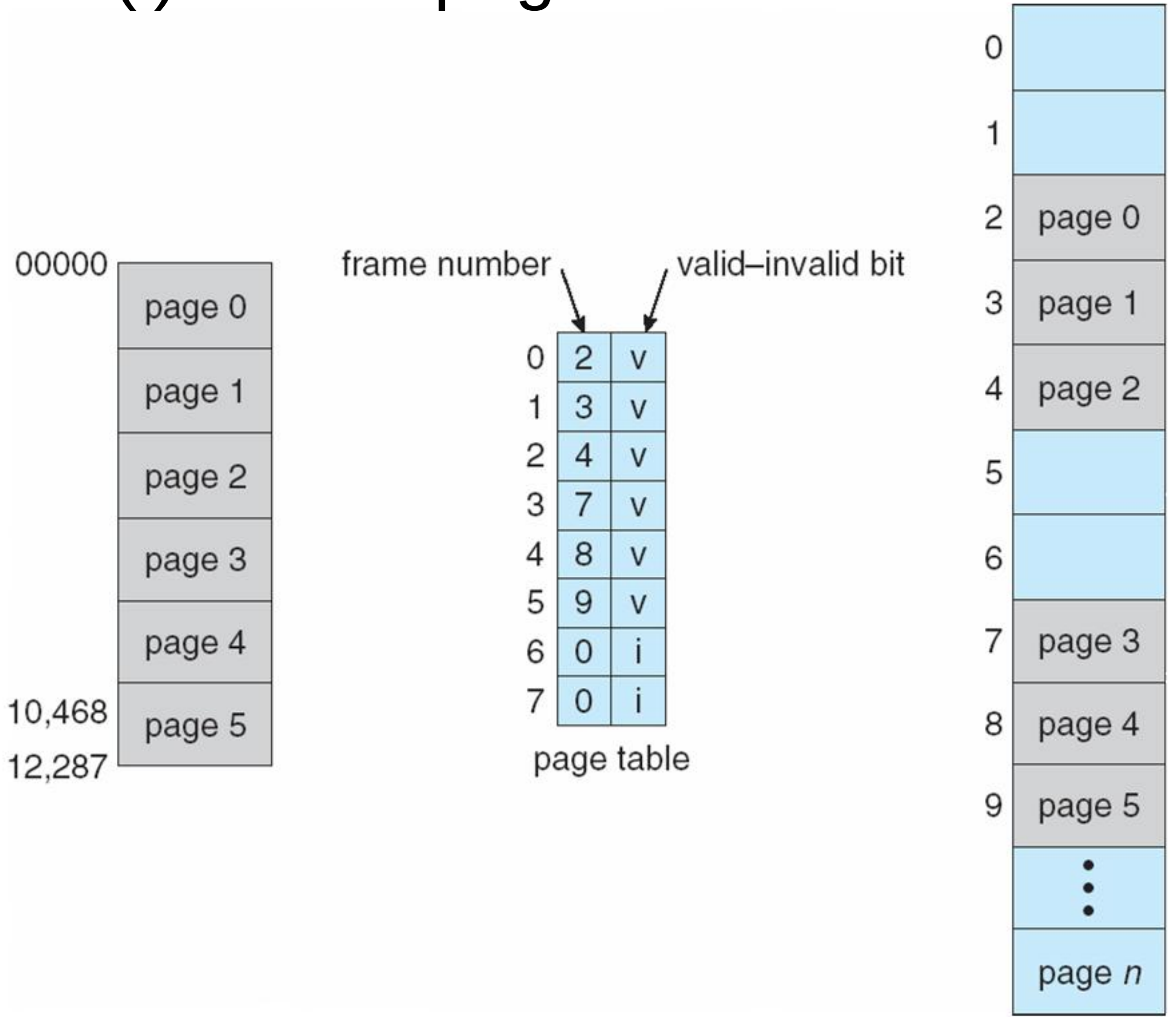
Memory Protection

- Each frame has a **protection bits**
  - Read-write / read only
  - Attempt to write to a read-only page causes H/W trap

    → Extension: read-only, read-write, execute-only, …
    - Combination of them
- Each entry of page table has **valid-invalid bit**
  - "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
    - OS does not allow to access the page
- Rarely does a process use all of its address range
  - Page table can be wasting memory → **page-table length register** (**PTLR**)

Valid-Invalid Bit

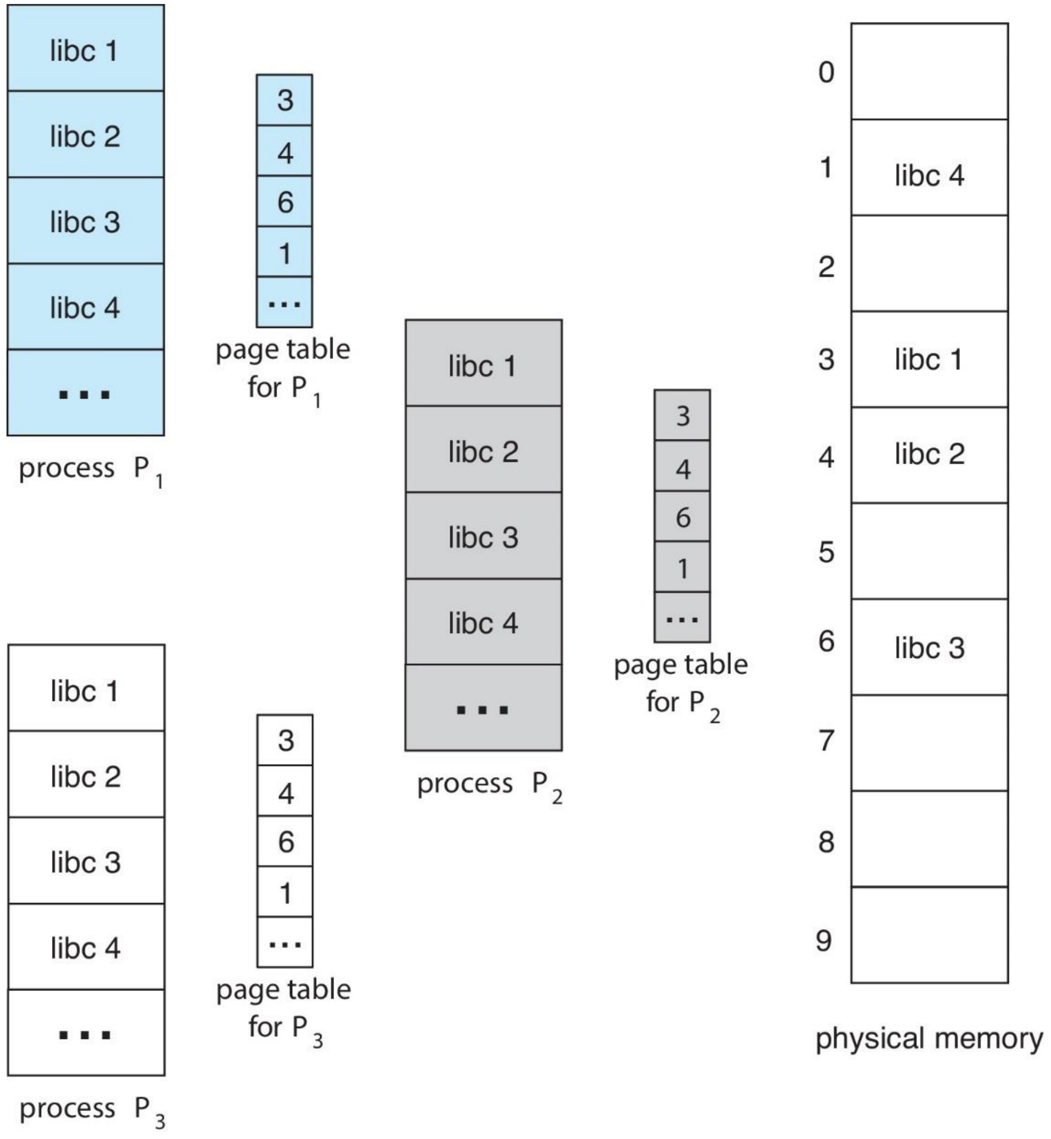
- Valid (v) or Invalid (i) bit in a page table



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 inter-process 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

Shared Pages Example



Agenda

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architecture

OS
Intermediate

HANDONG GLOBAL UNIVERSITY

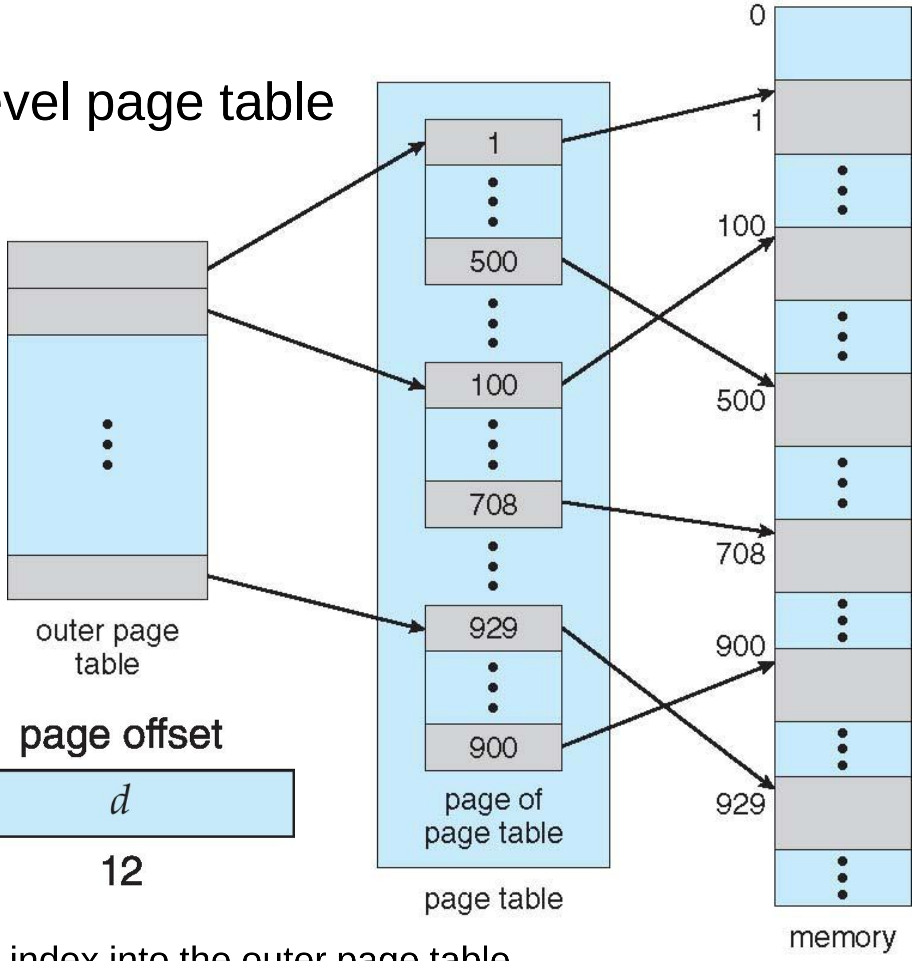
Structure of the Page Table

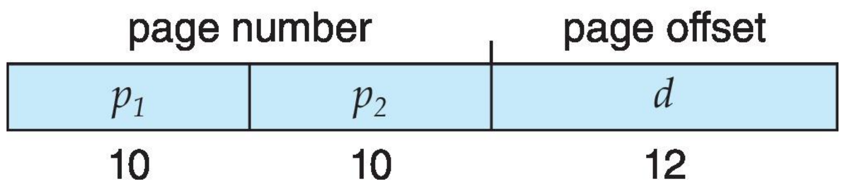
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes → each process 4 MB of physical address 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
    - Hashed Page Tables
    - Inverted Page Tables

Hierarchical Page Tables

- **Hierarchical page tables**: Break up the logical address space into multiple page tables
  - We then page the page table
  - Ex) On 32-bit machine with 4KB page size, a logical address 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
  - Known as **forward-mapped page table**

Two-level page table

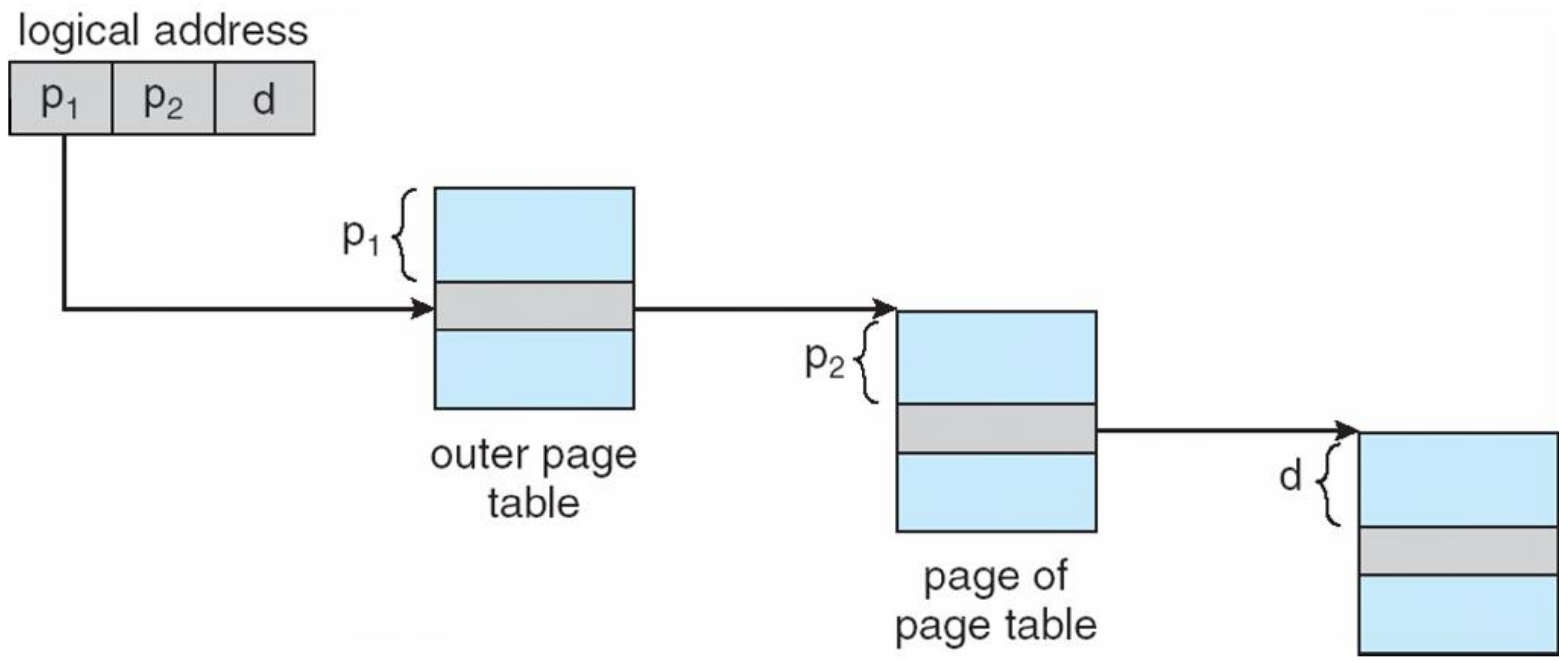




p_1 is an index into the outer page table,
p_2 is the displacement within the page of the inner page table

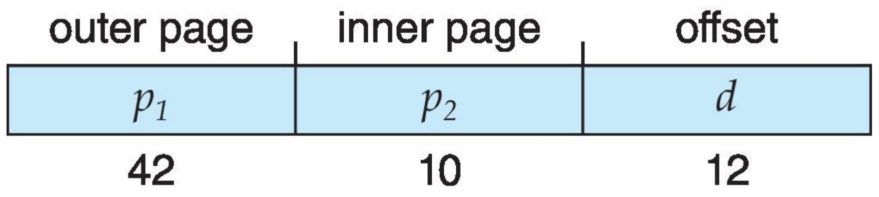
Hierarchical Page Tables

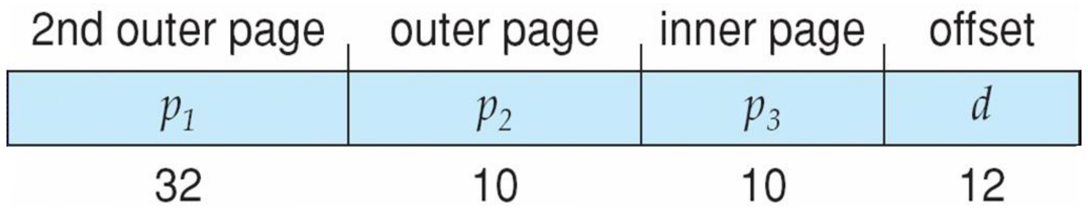
- Address-Translation Scheme



Hierarchical Page Tables

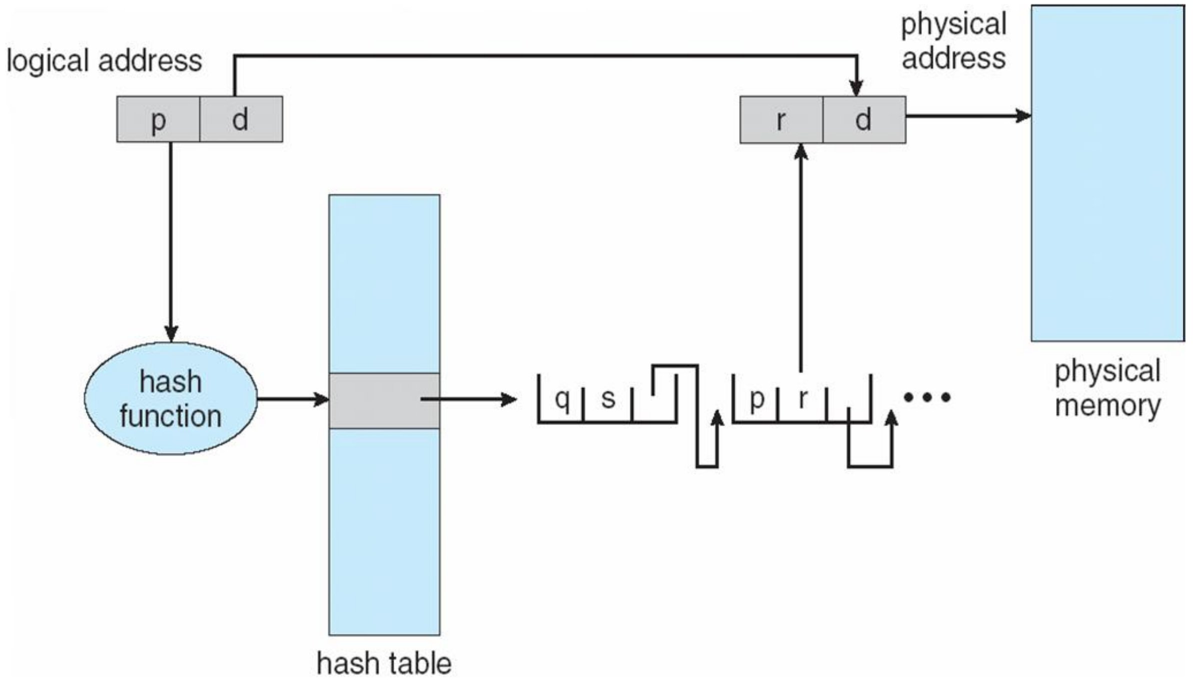
- 64-bit logical address space
  - Even two-level paging scheme not sufficient
  - If page size is 4 KB (2^{12})
    - Then page table has 2^{52} entries
    - If two level scheme, inner page tables could be 2^{10} 4-byte entries
    - Outer page table has 2^{42} entries or 2^{44} bytes
    - One solution is to add a 2nd outer page table
    - 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





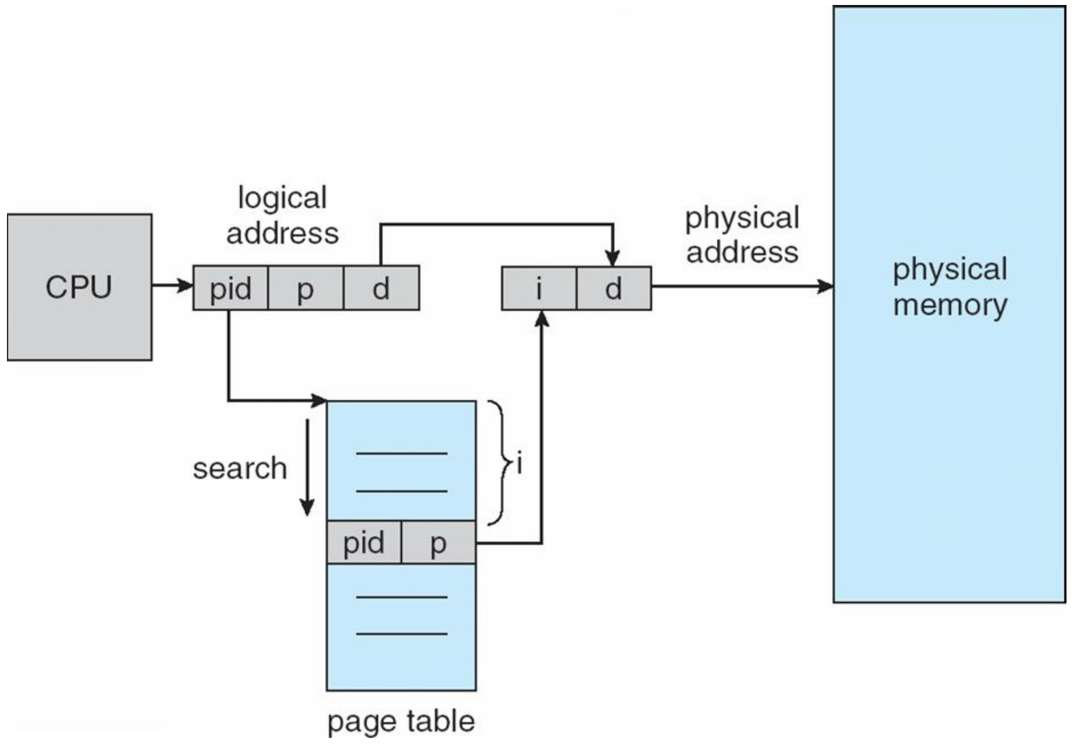
Hashed Page Table

- **Hashed page tables**
  - Common in large 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
  - Each element contains
    - The virtual page number
    - The value of the mapped page frame
    - 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



Inverted Page Table

- **Inverted page table**: Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
  - Only one inverted page table exists in whole system
  - Each entry is for real page of memory
  - Each entry has address-space identifier
  - Entry of inverted page table consists of
    - <process-id, page number>
  - Logical address consists of
    - <process-id, page-number, offset>

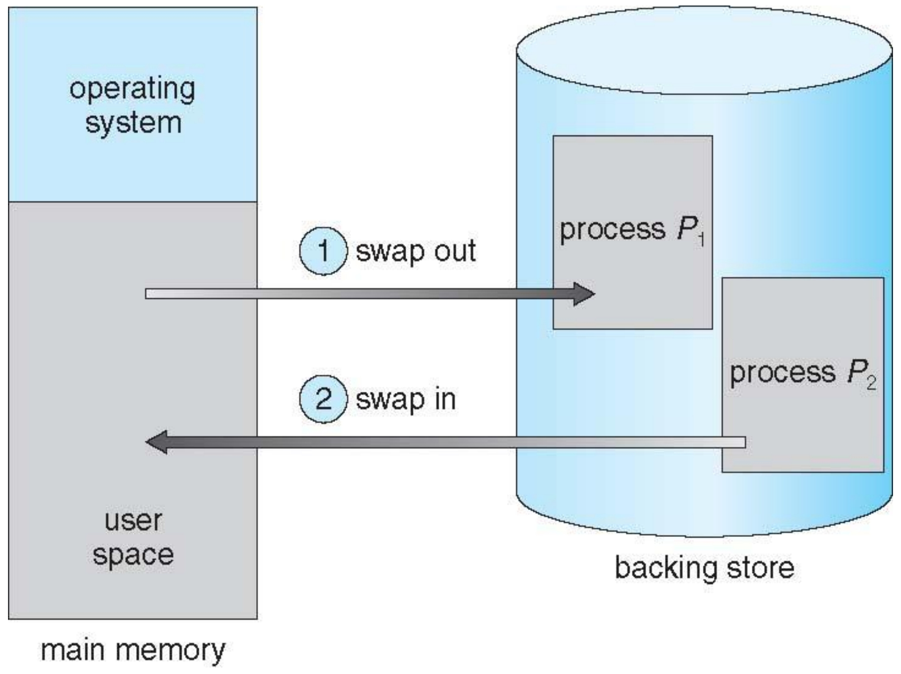


Agenda

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architecture

Swapping

- **Swapping**: a process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - It makes it possible for the 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
  - System maintains a ready queue of ready-to-run processes which have memory images on disk



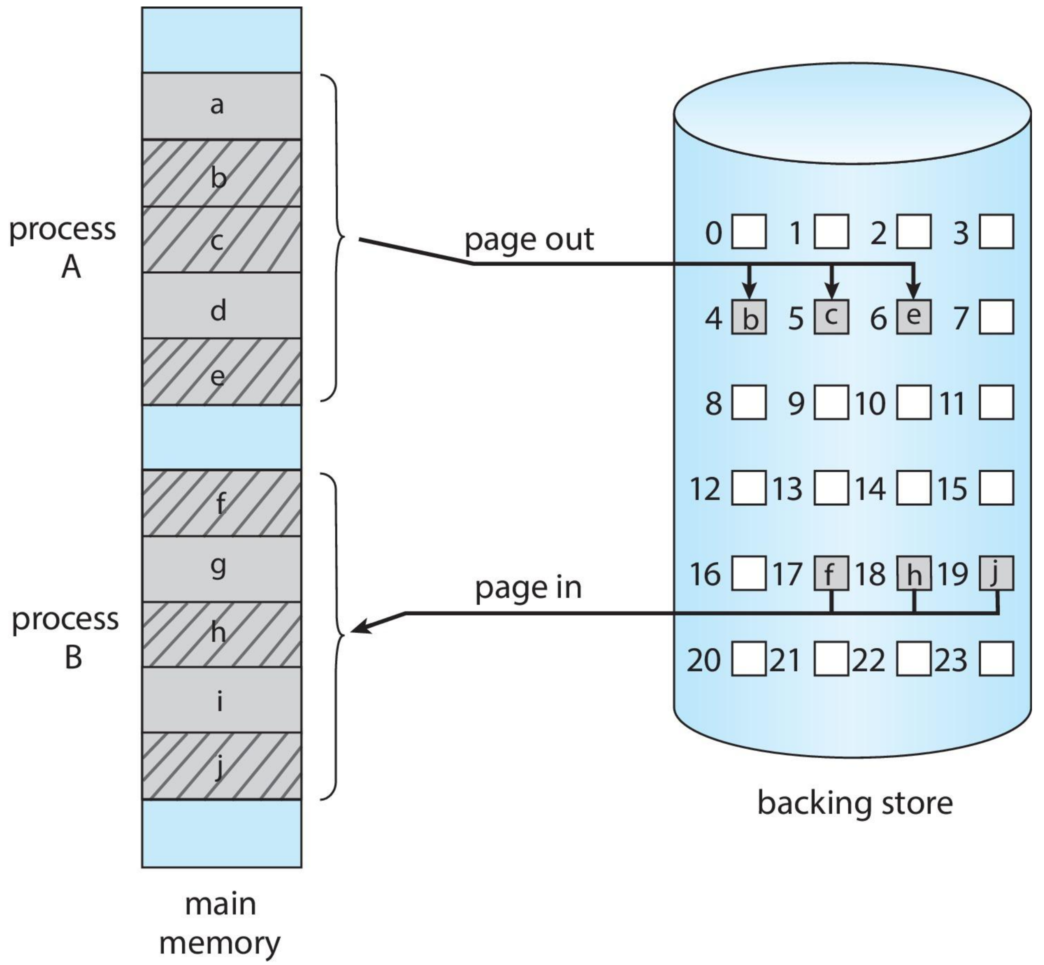
Context Switch Time including Swapping

- 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
  - Major part of swap time is transfer time
    - Total transfer time is directly proportional to the amount of memory swapped
  - Ex) size of process = 100MB
    hard disk transfer rate = 50MB/sec
    total context switch including swapping
    = swap out time (2000ms) + swap in time (2000ms) = 4000ms (4sec)

Context Switch Time and Swapping

- Standard swapping involves moving entire processes between main memory and backing store
  - It allows physical memory to be oversubscribed, so that the system can accommodate more processes than size of physical memory
  - However, standard swapping is not used in modern operating systems
    - The amount of time required to move entire processes is prohibitive
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Swapping with Paging



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

Agenda

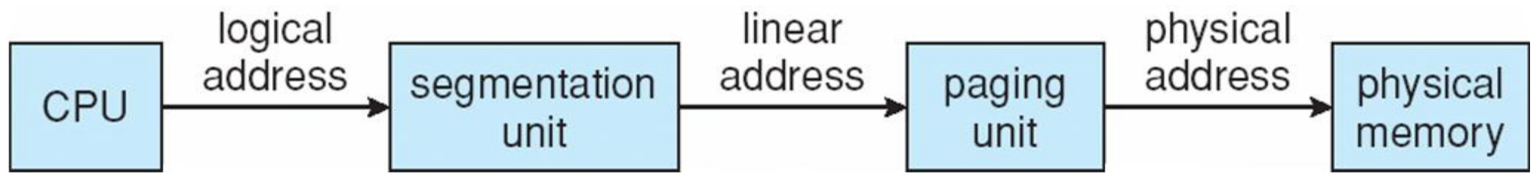
- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architecture

The Intel 32 and 64-bit Architectures

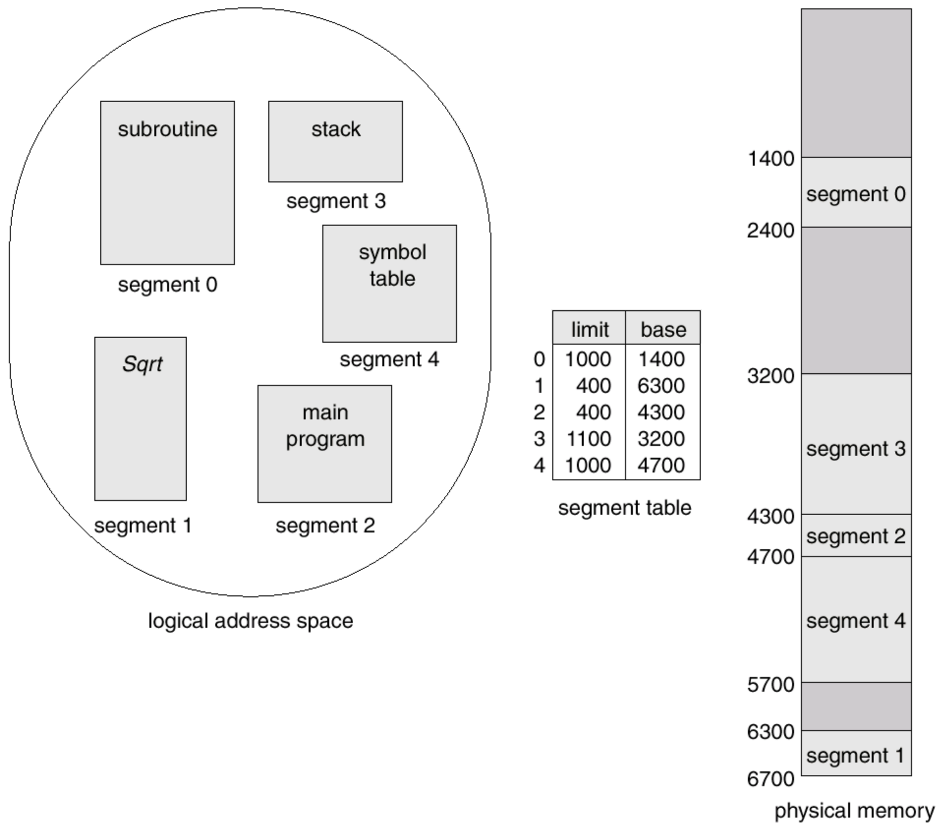
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

The Intel IA-32 Architecture

- Logical to physical address translation in IA-32



- Segmentation



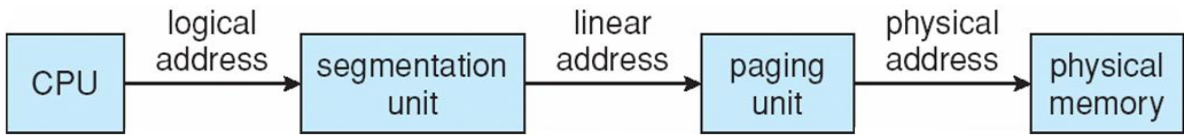
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

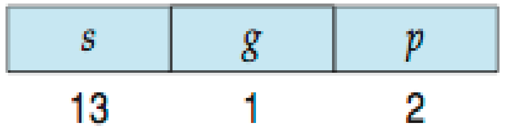
IA-32 Segmentation

- Supports both segmentation and segmentation with paging
  - Each segment can be 4GB
  - Up to 16K segments per process
  - Divided into two partitions
    - First partition of up to 8K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

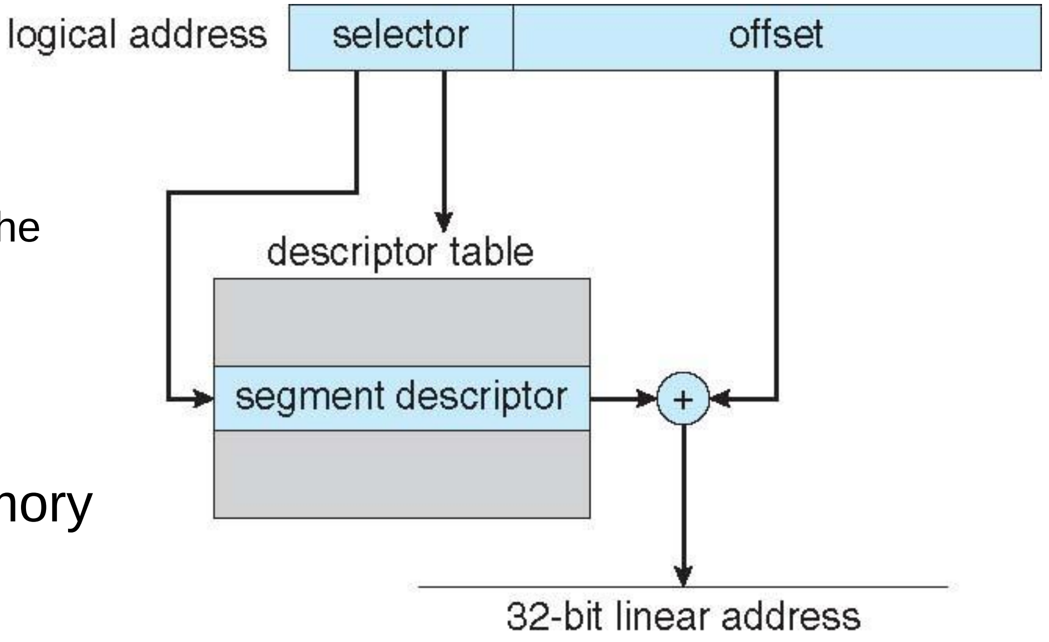
IA-32 Segmentation



- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



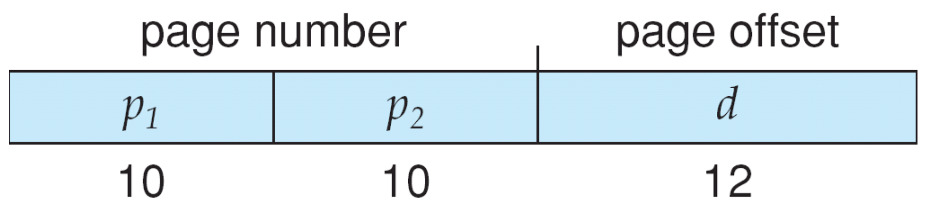
s: segment number
g: whether the segment is in the GDT or LDT
p: deals with protection



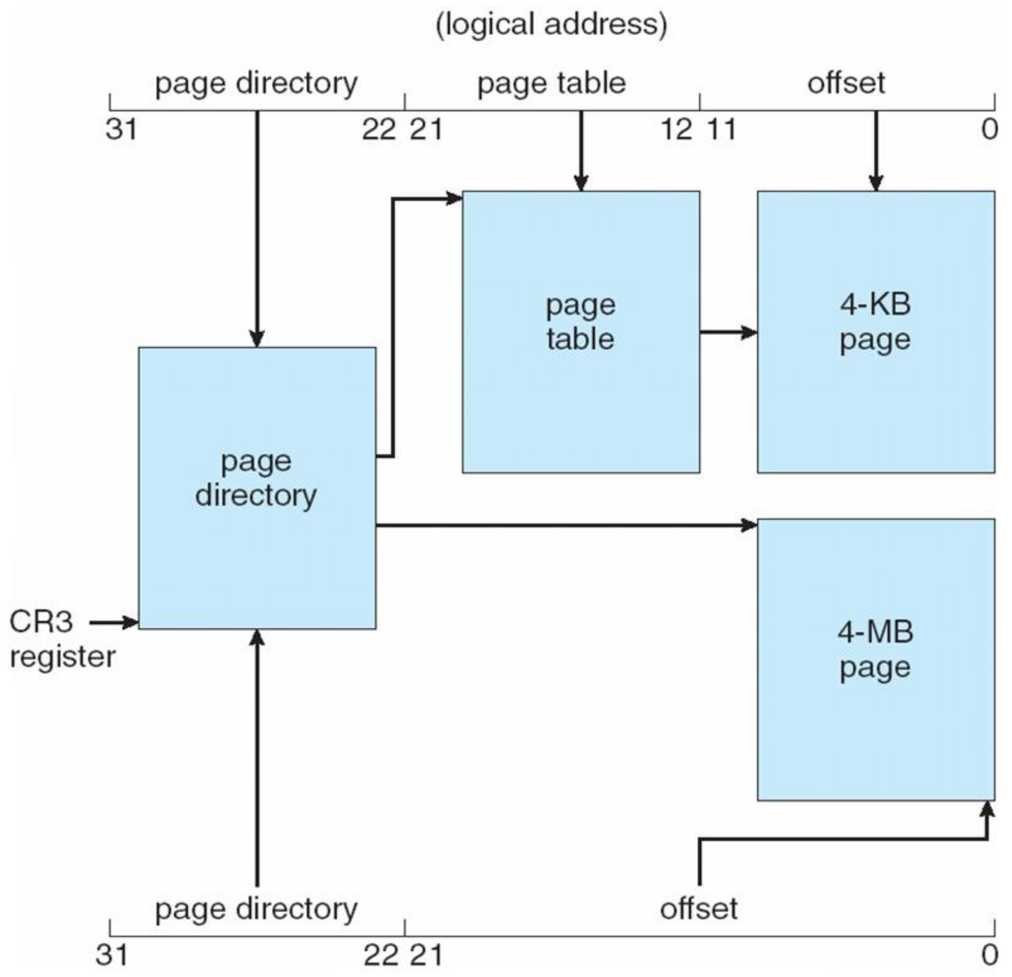
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

IA-32 Paging

- IA-32 allows a page size of either 4KB or 4MB
- For 4KB page, IA-32 uses a two-level paging scheme

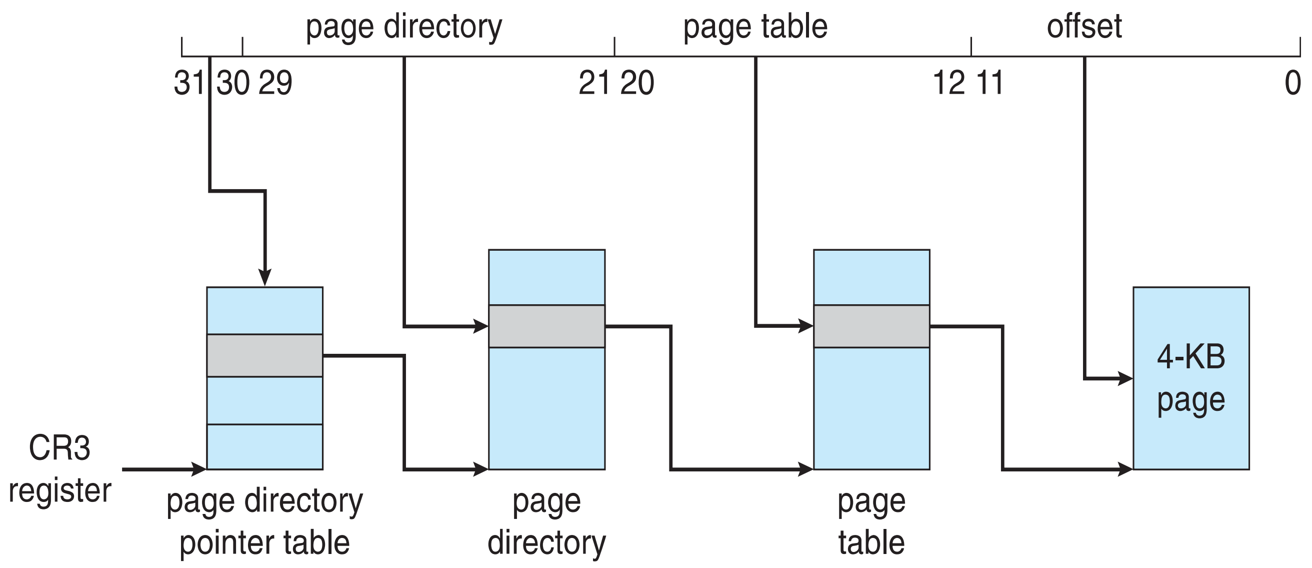


- If `Page_Size` flag is set, the page directory points directly 4MB page
  - Bypassing the inner page table



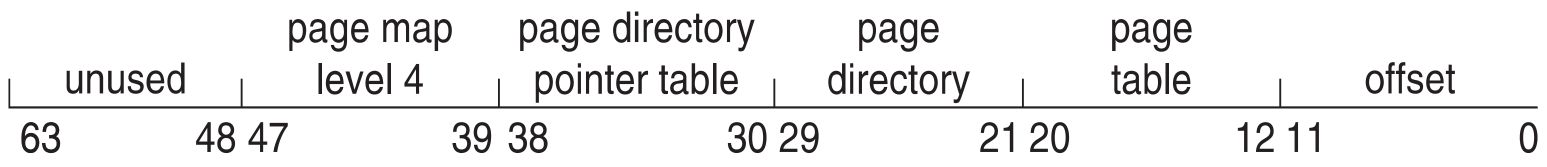
IA-32 Page Address Extensions

- 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 (4096 terabytes)


unused
page map level 4
page directory pointer table
page directory
page table
offset
63
48 47
39 38
30 29
21 20
12 11
0