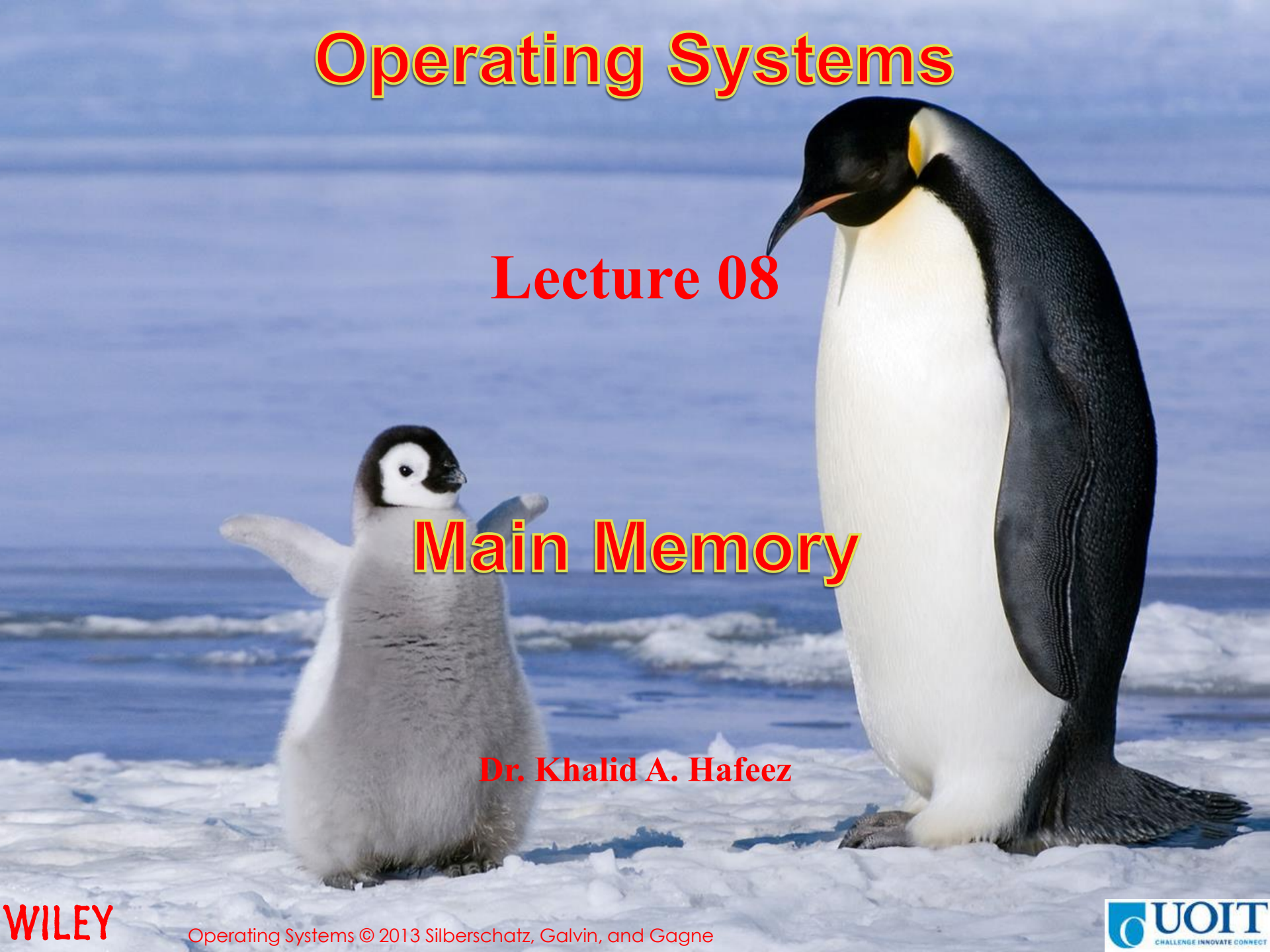
Operating Systems

Lecture 08

Main Memory

Dr. Khalid A. Hafeez



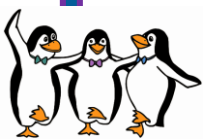
WILEY

Operating Systems © 2013 Silberschatz, Galvin, and Gagne

UOIT
CHALLENGE INNOVATE CONNECT

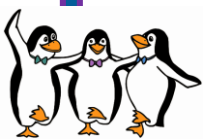
Memory Management

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



Objectives

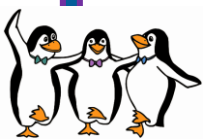
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging



Background

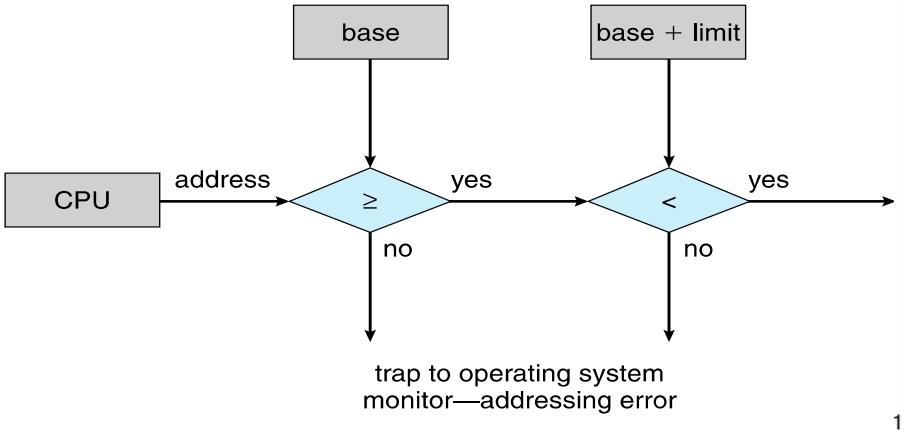
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the only storage that CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Registers can be accessed 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

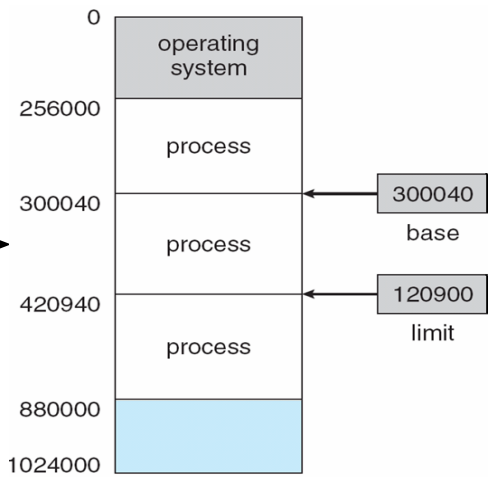
- Memory management is crucial in better utilizing one of the most important resources of the system, i.e., memory



Background

- Basic Hardware
  - Every process must have a separate memory space:
    - To protect the processes from each other and
    - To allow for multiple processes to be loaded in memory for concurrent execution.
  - Base and Limit Registers
    - A pair of **base** and **limit registers** define the logical address space for a process (loaded only by the OS)
    - CPU must check every memory access generated in user mode to be sure that it is between base and limit for that user (process)
    - Hardware Address Protection

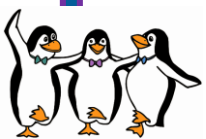




Background

Address Binding

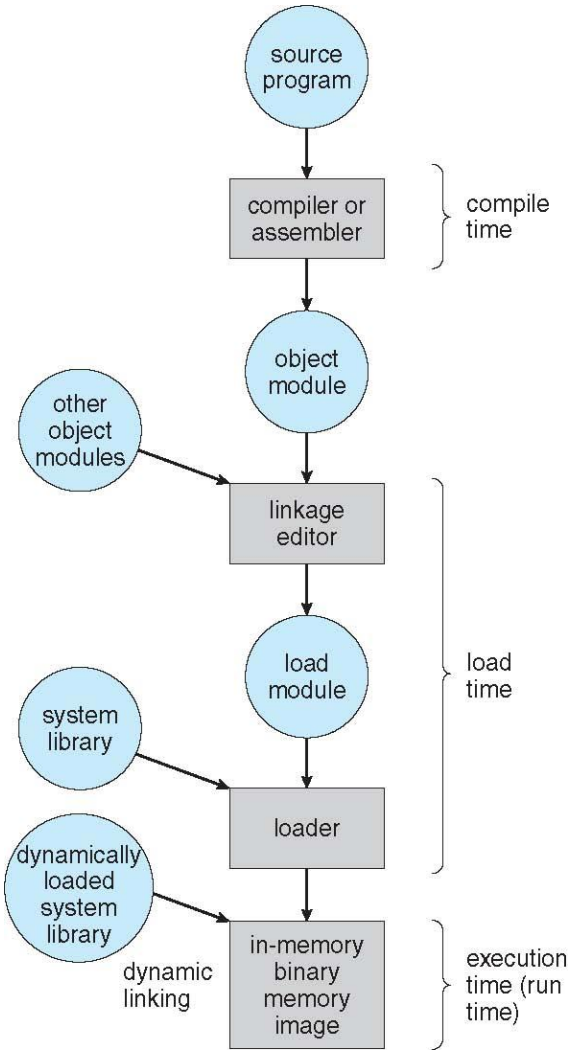
- To be executed, the program must be brought into memory and placed within a process.
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
  - But it is not always at 0000
- Further, addresses represented in different ways at different stages of a program′s life
  - **Source code** addresses usually symbolic (BGT Loop)
  - **Compiled code** addresses **bind** to relocatable addresses
    - i.e. ″14 bytes from beginning of this module″
  - **Linker** or **loader** will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another



Background

Address Binding

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

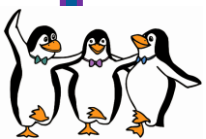


Background

Logical vs. Physical Address Space

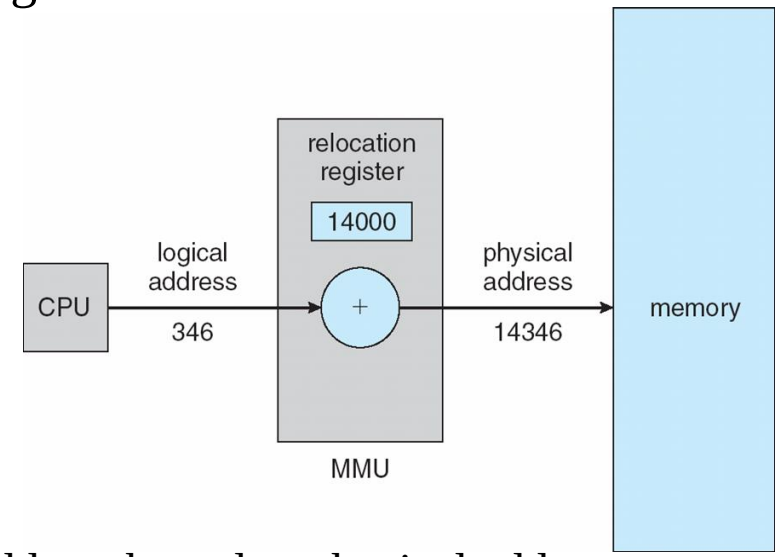
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical address**: address generated by the CPU; also referred to as virtual address
  - **Physical address**: address seen by the memory unit; the one loaded into the memory-address register of the memory
- Logical and physical addresses are the **same** in compile-time and load-time address-binding schemes;
- logical 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 corresponding to these logical addresses



Background

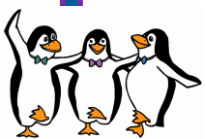
- Memory-Management Unit (MMU)
  - It is a hardware device that at run time **maps** virtual to physical address
  - A simple scheme: add the value in the relocation register to every address generated by a user process at the time it is sent to memory
    - Base register now is called **relocation register**
    - MS-DOS on Intel 80x86 used 4 relocation registers
  - 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

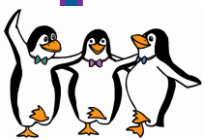
Background

- Dynamic Loading
  - Because running multiple programs requires much more memory locations than is available in physical memory.
    - There fore, a routine is not loaded until it is called
    - This results in better memory-space utilization; unused routine is never loaded
    - In this case all routines kept on disk in relocatable load format
  - Useful when large amounts of code are needed to handle infrequently occurring cases (such as error handling code)
  - No special support from the operating system is required
    - Implemented by programmers through program design
    - OS can help by providing libraries to implement dynamic loading



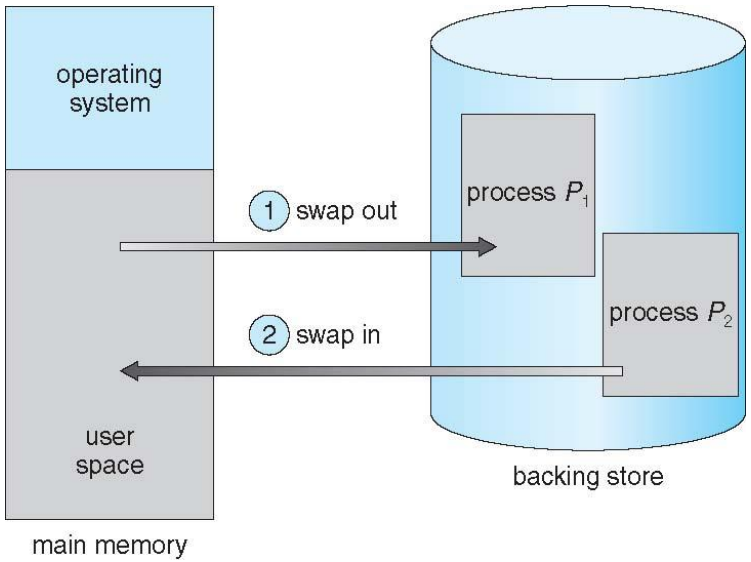
Background

- Dynamic Linking and Shared Libraries
  - Static linking: system libraries and program code combined by the loader into the binary program image (at the loading stage); it **wastes** both disk space and main memory.
  - Dynamic linking: linking postponed until the execution time; so load only the subroutines that are needed not the whole language library
    - Small piece of code (called **stub**) is included in the image for each library routine reference.
      - It is used to locate the appropriate memory-resident library routine and replaces itself with the address of the routine, and executes that routine
    - Under this scheme, all processes that use a language library execute only **one copy** of the library code.
      - Therefore, it needs help from the OS which checks if the required routine is in another processes′ memory address and link it to others
    - This system also known as **shared libraries**



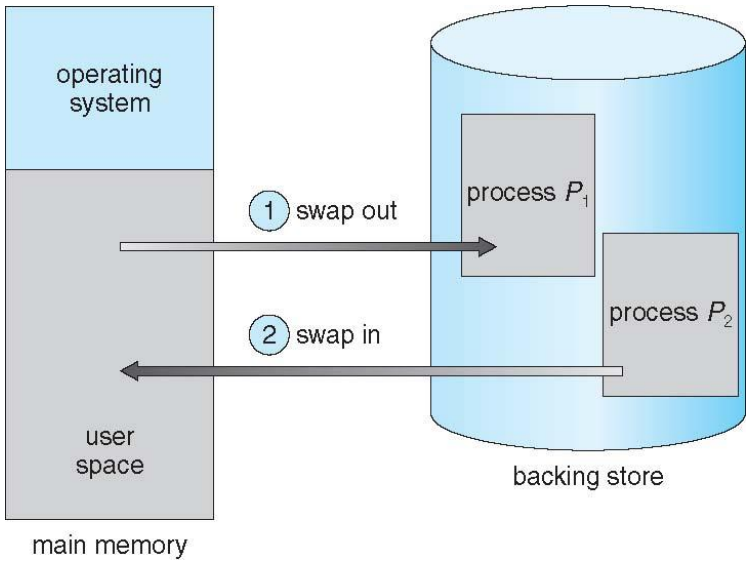
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Therefore, the total physical memory space of processes can exceed the physical memory
- **Backing store**: it is a fast disk large enough to accommodate copies of all memory images for all users; it must provide direct access to these memory images
- **Roll out, roll in**: swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed



Swapping

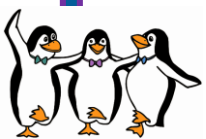
- Major part of swap time is the transfer time (from and to backing store); total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping is normally disabled
  - Swap only when free memory extremely low (below a threshold)
  - Disabled again once memory demand reduced below threshold



Swapping

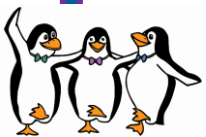
Context Switch Time including Swapping

- 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
  - If the process is of size 100MB and the system is using a hard disk with transfer rate of 50MB/sec, then:
    - Swap out time of 2 sec
    - Plus swap in of same sized process
    - Total context switch swapping component time of 4 seconds
  - This can be reduced if we reduce the size of memory need to be swapped, by knowing how much memory really will be used not how much the process might use
    - The process can use system calls to inform OS of memory use via request_memory() and release_memory()



Swapping

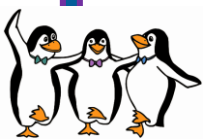
- Context Switch Time including Swapping
  - Other constraints on swapping:
    - Pending I/O: a process that is waiting for I/O can't be swapped out as I/O would occur to a wrong process
    - A solution is to transfer the I/O to kernel space, then to I/O device
      - This is known as **double buffering**, but it adds overhead



Swapping

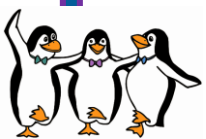
Swapping on Mobile Systems

- Swapping is not typically supported on mobile devices, because:
  - They are flash memory based
    - Small amount of space
    - Limited number of write cycles before it becomes unreliable
    - Poor throughput between flash memory and main memory
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data is thrown out and reloaded from flash if needed
    - If a process did not free memory, it may be terminated
  - Android terminates apps if insufficient free memory is available, but first writes the **application state** to flash for fast restart
- Both OSes support paging, so they do have memory-management abilities.



Contiguous Memory Allocation

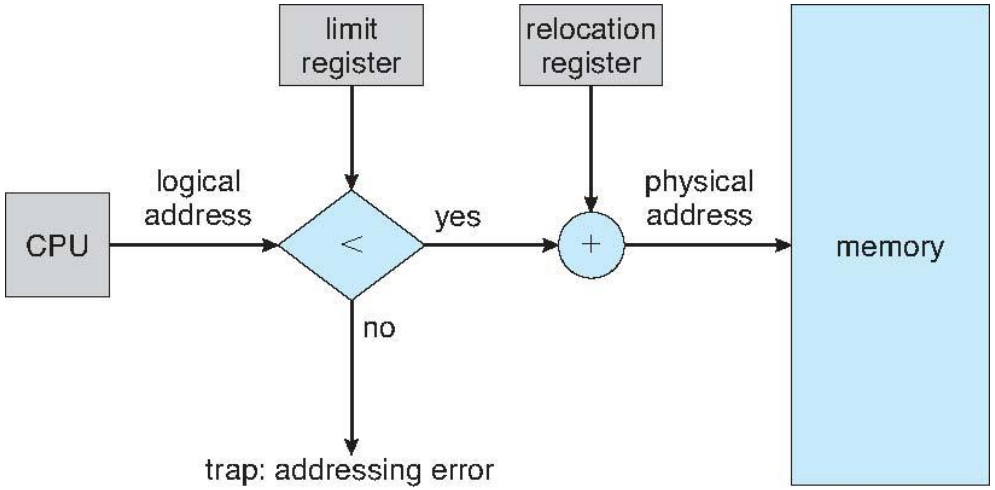
- Main memory must support both OS and user processes
- Because of limited memory resource, we must allocate the process efficiently
- **Contiguous memory allocation scheme:**
  - It is one of the early methods
  - The main memory is usually divided into two **partitions**:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes then held in high memory
      - Each process contained in a single contiguous section of memory



Contiguous Memory Allocation

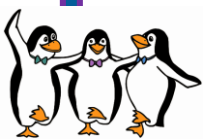
Memory Protection

- 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
  - The MMU maps the logical address to physical address *dynamically*
    - It can then allow the kernel code being **transient** (change its size):
      - The not needed code (e.g. not used device drivers) can be removed and brought back when needed



Contiguous Memory Allocation

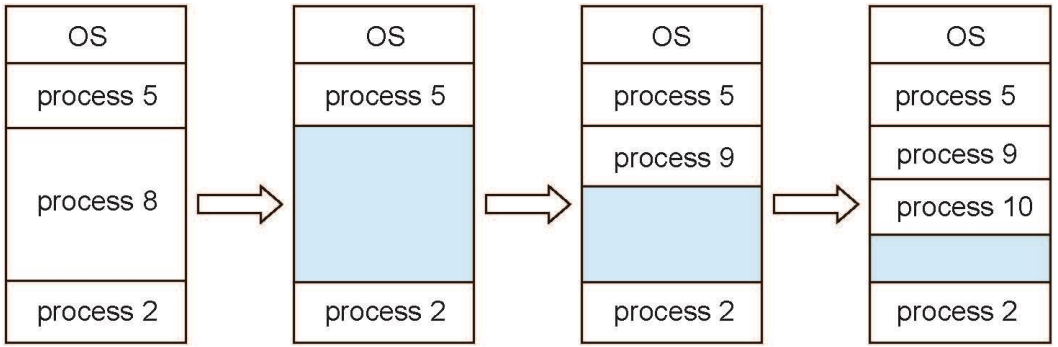
- Single-partition allocation
  - A simple method for allocating memory is to divide it into several fixed-sized partitions. Each partition may contain exactly one process
    - Therefore, the degree of multiprogramming is limited by number of partitions available for user processes



Contiguous Memory Allocation

Variable-partition scheme

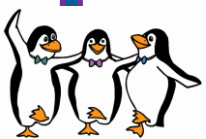
- The operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially all memory is available for user processes (one hole)
  - **Hole**: is a block of available memory
- A process can be given a **variable-partition** size for efficiency (sized to a given process' needs)
- Therefore, holes of various size will be 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 are combined
- OS maintains information about: allocated partitions and free partitions (holes)



Contiguous Memory Allocation

Dynamic Storage-Allocation Problem

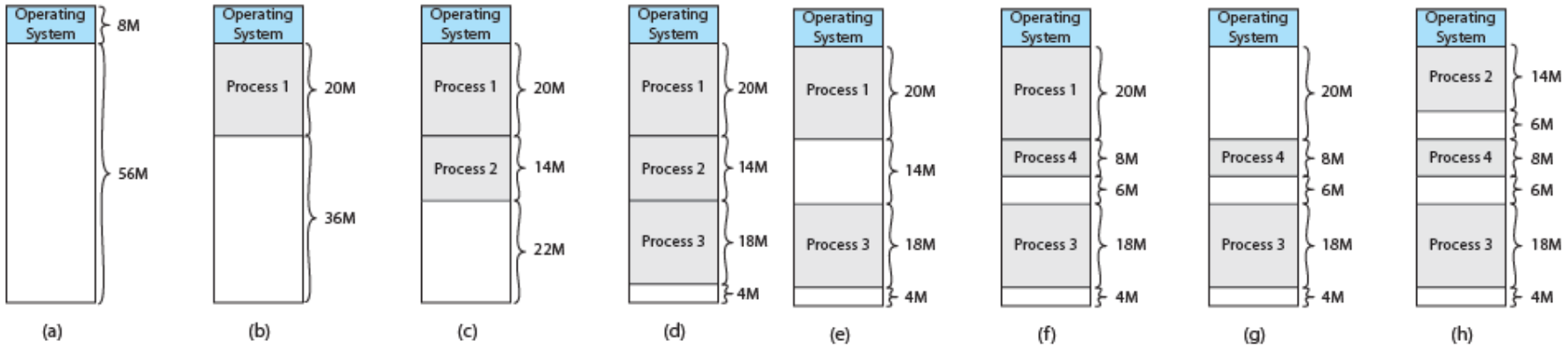
- How to satisfy a request of size ***n*** from a list of free holes?
  - **First-fit**: Allocate the ***first*** hole that is big enough
  - **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search the entire list, unless the list is ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search the entire list
    - Produces the largest leftover hole; so the hole can be used for other process
- **Based on simulation**:
  - First-fit and best-fit are better than worst-fit in terms of speed and storage utilization



Contiguous Memory Allocation

Fragmentation

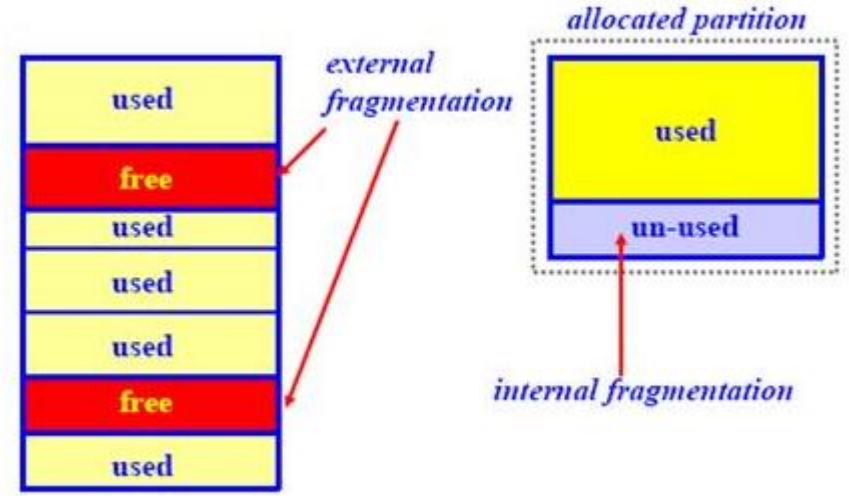
- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation problem:
  - **External Fragmentation**: total available memory space is enough to satisfy a new request, but it is not contiguous
    - First fit analysis reveals that given *N* allocated blocks, 0.5*N* blocks are lost to fragmentation
      - » That is, one-third of memory may be unusable!
        - » This property is known as the 50-percent rule.



Contiguous Memory Allocation

Fragmentation

- Another approach to avoiding scattered small holes in the memory, is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size
  - **Internal Fragmentation**: allocated memory may be slightly larger than requested memory; this size difference in memory is internal to the partition, but not being used

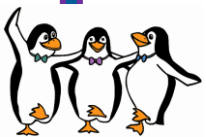


Contiguous Memory Allocation

Fragmentation

- One solution to the problem of external fragmentation is **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

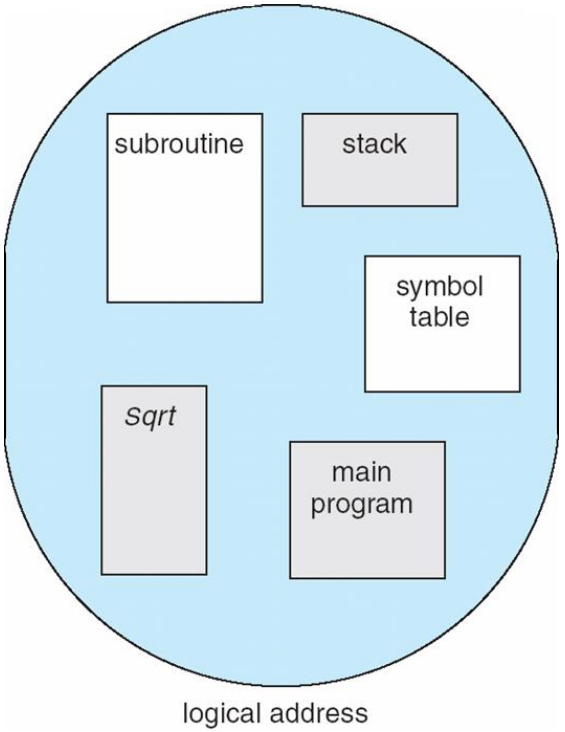
- Another solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous
  - **Two complementary techniques achieve this solution:**
    - Segmentation
    - Paging



Segmentation

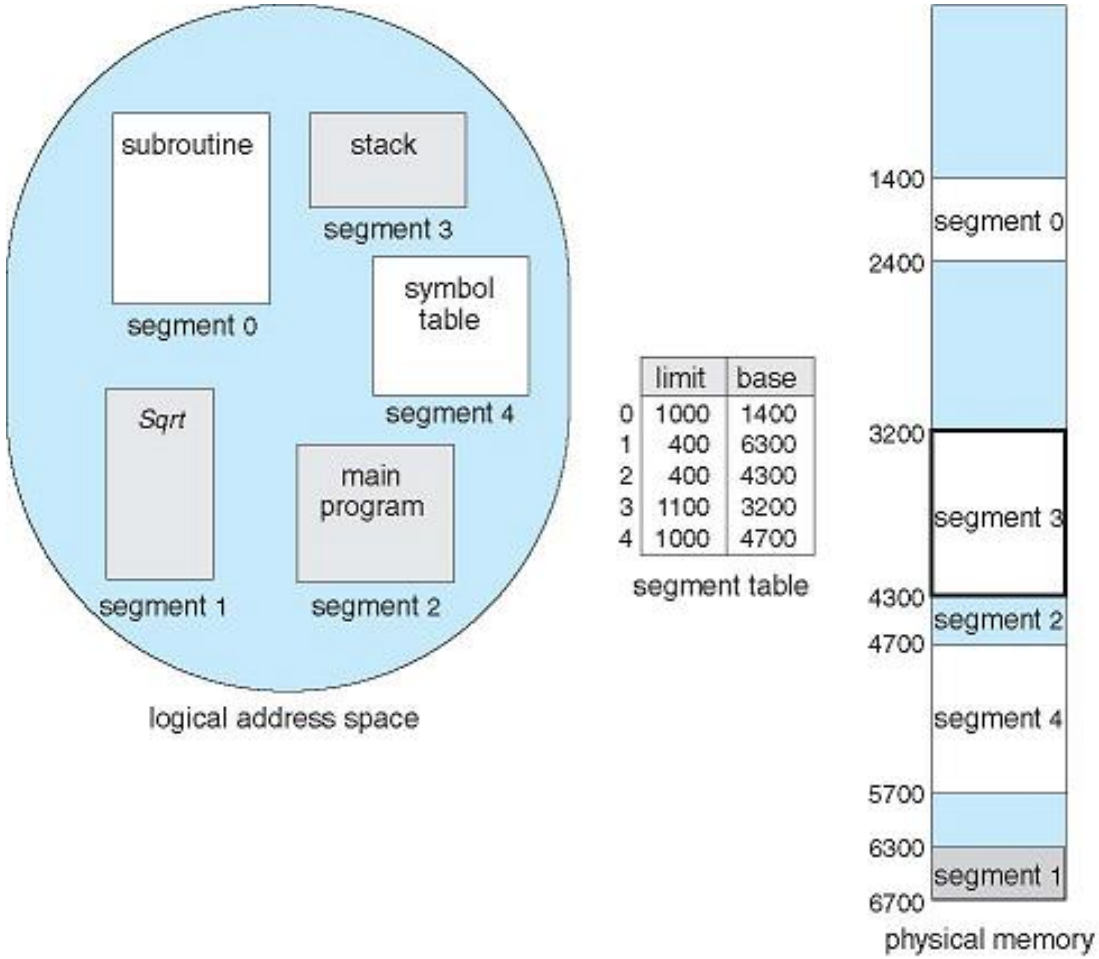
- Segmentation is a memory-management scheme that supports user the view of memory: that is a program is a collection of segments
- Thus, the logical address consists of a two tuple: <segment-number, offset>
- The compiler automatically constructs segments reflecting the input program:
  - A **C** compiler might create separate segments for the following:
    1. The code
    2. Global variables
    3. The heap, from which memory is allocated
    4. The stacks used by each thread
    5. The standard C library

- The loader would take all these segments and assign them segment numbers.



User's View of a Program

Segmentation

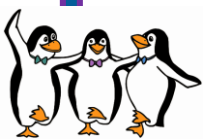


Segmentation

Segmentation Architecture

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

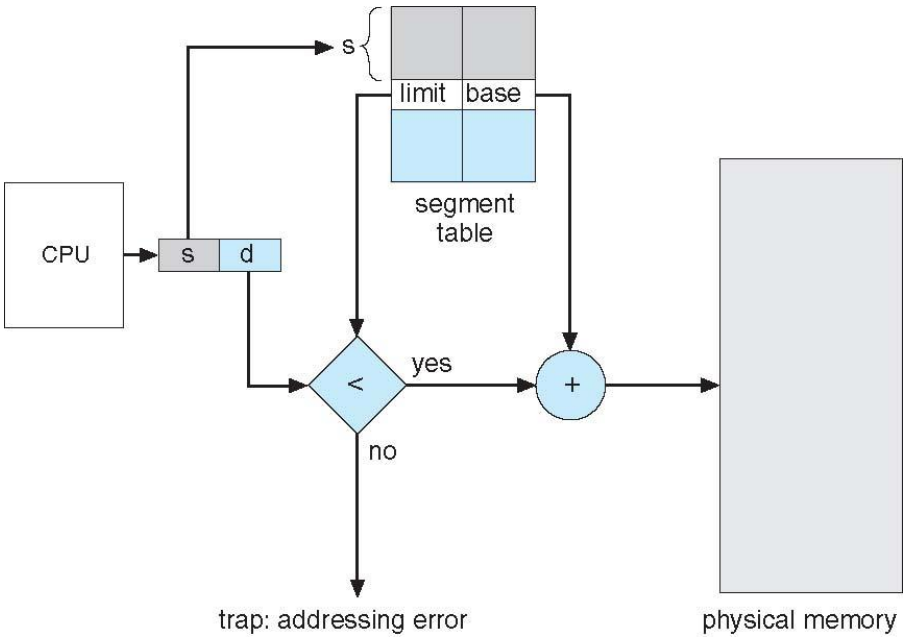
  segment number ***s*** is legal if ***s*** < **STLR**



Segmentation

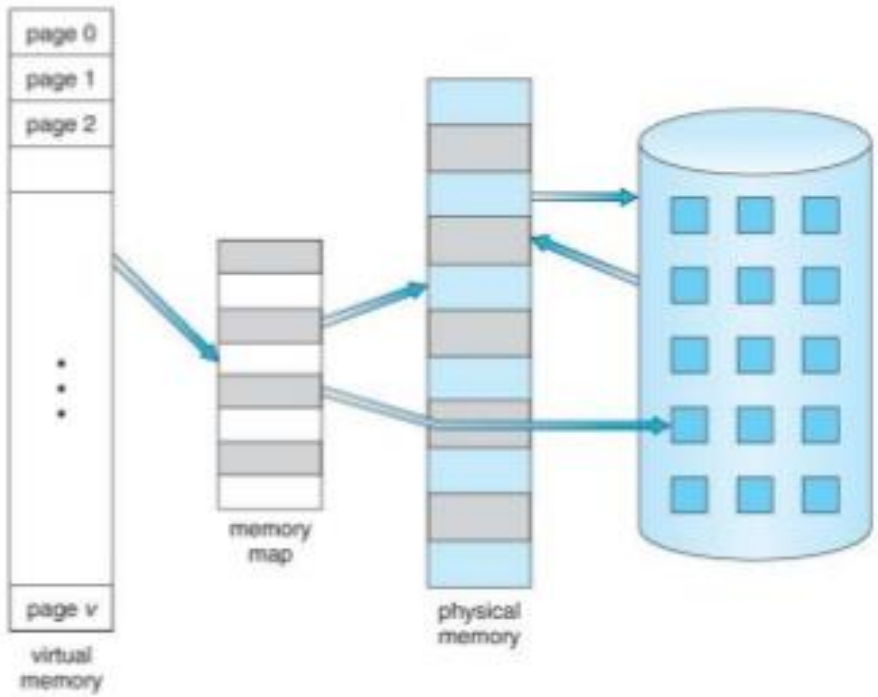
Segmentation Architecture

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem



Paging

- **Paging** is memory-management scheme that permits the physical address space of a process to be noncontiguous.
  - A process is allocated physical memory whenever the latter is available
  - It avoids the external fragmentation problem
  - It avoids the problem of varying sized memory chunks
  - It is implemented through cooperation between the OS and the computer hardware.

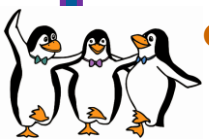


Paging

Basic Method

- The physical memory is divided into **fixed-sized** blocks called **frames**
- Divide the logical memory into blocks of same size called **pages**
- The page size (like the frame size) is defined by the hardware:
  - The size is a power of 2, (512 bytes to 1 GB per page)
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
- Set up a **page table** to translate the logical to physical addresses

- Advantage: the logical address space is totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.
- Disadvantage: Paging is still have the internal fragmentation problem

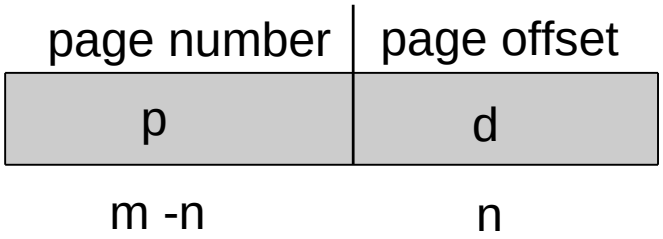


Paging

Basic Method

Address Translation Scheme

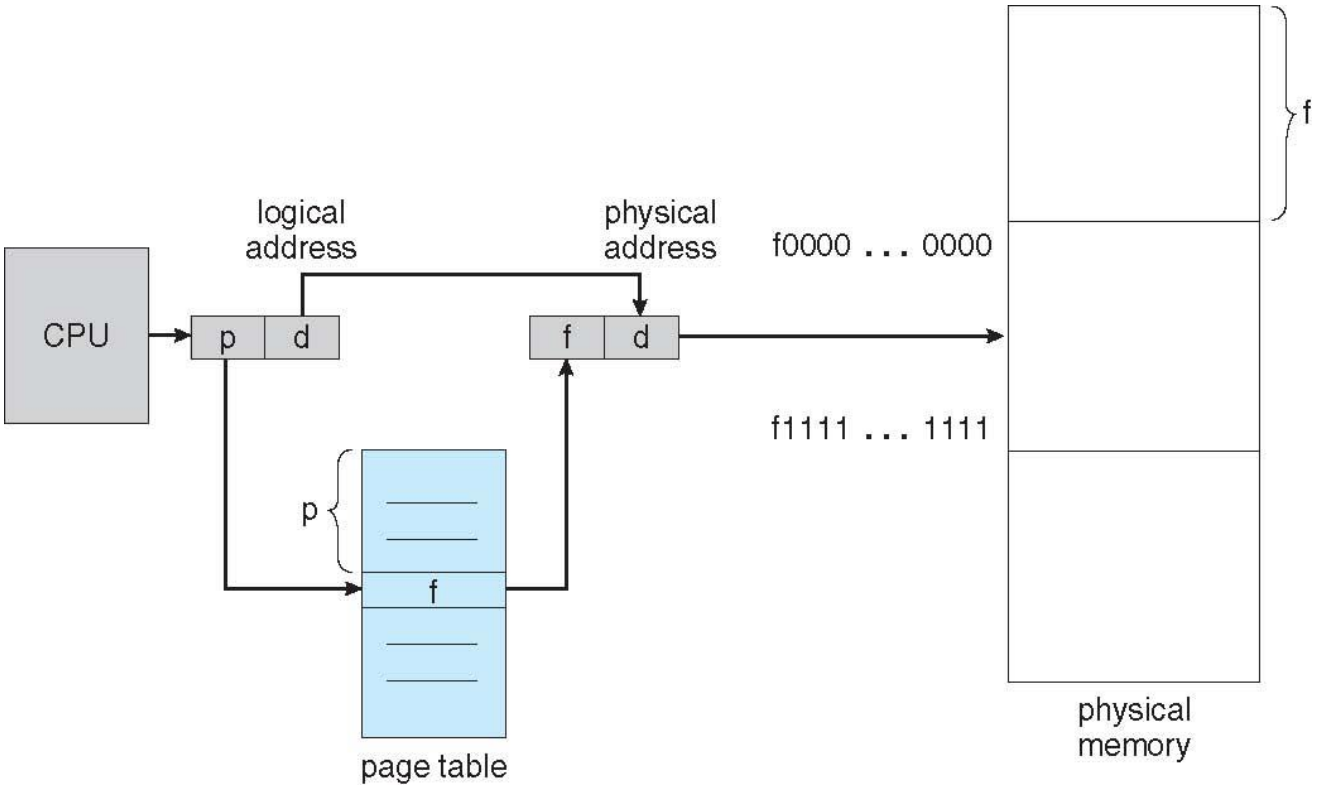
- Address generated by CPU is divided into two parts:
  - **Page number** (***p***): used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***): it is combined with base address to define the physical memory address that is sent to the memory unit



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

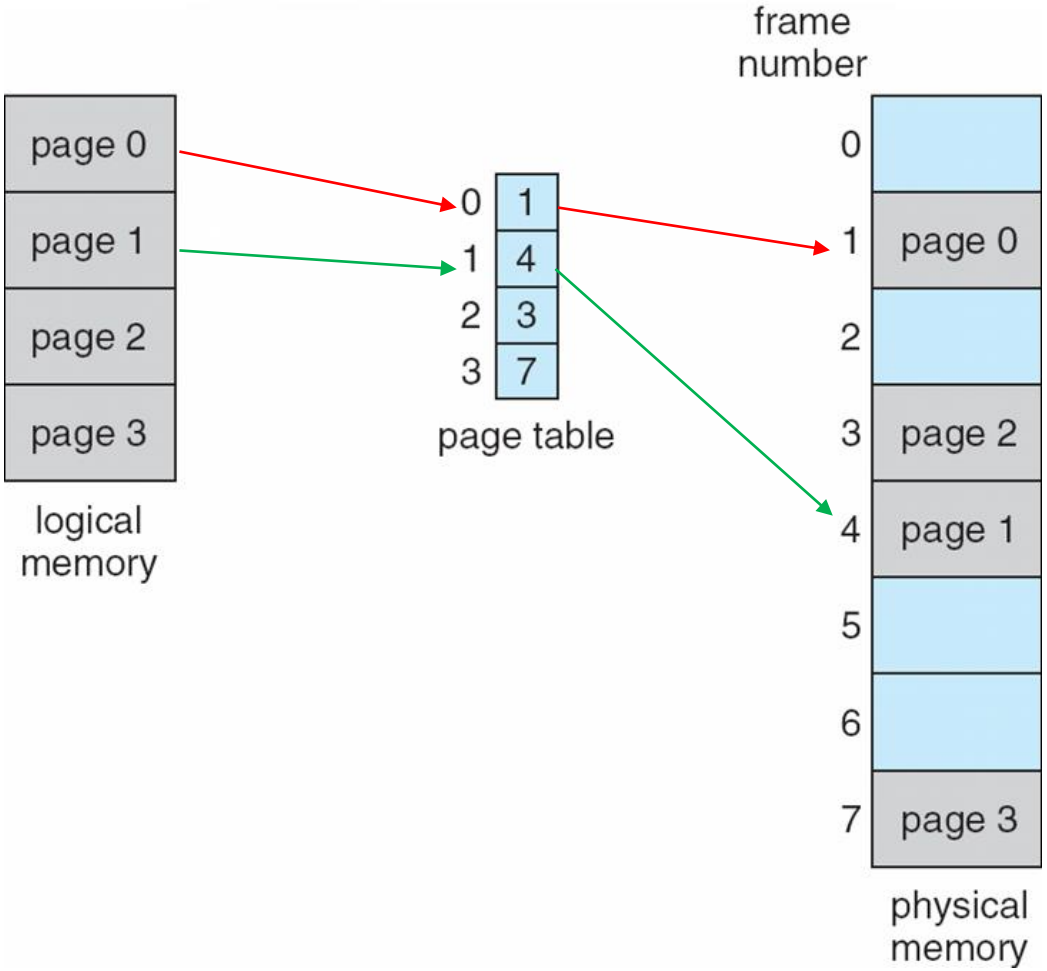
Paging

- Basic Method
  - Paging Hardware



Paging

- Basic Method
  - Paging Model of Logical and Physical Memory

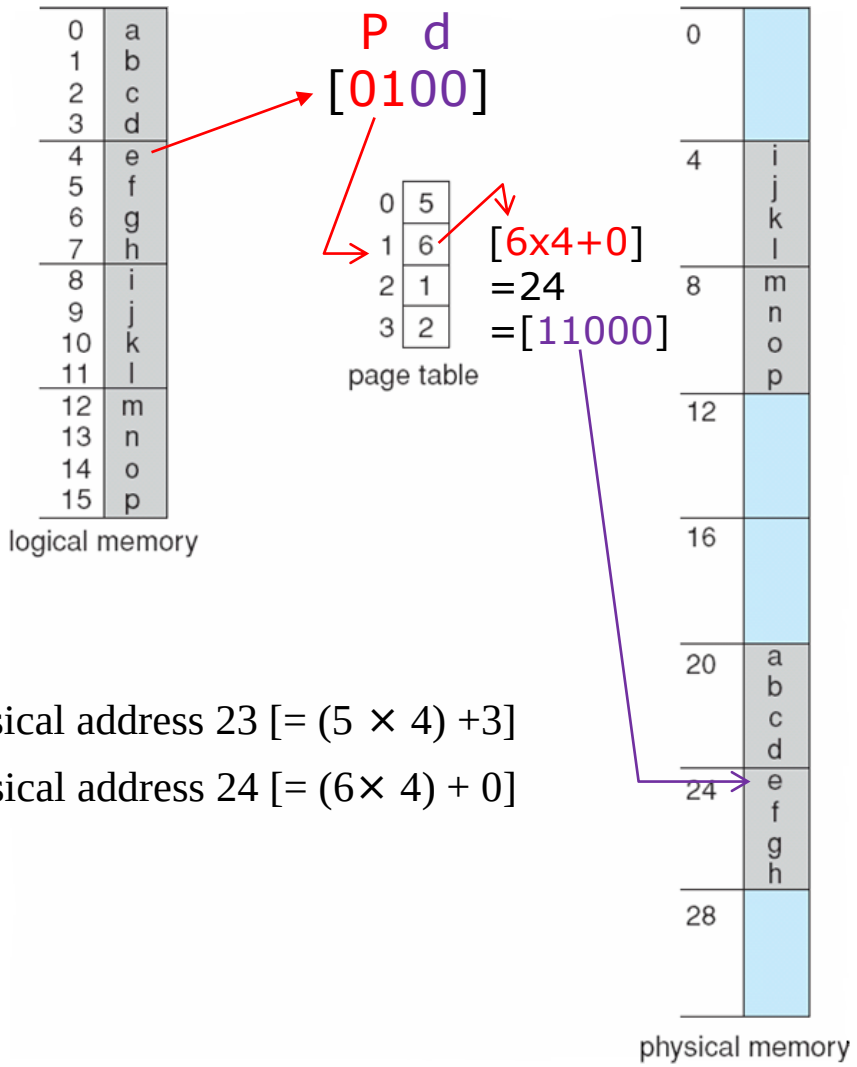


Paging

Basic Method

Example 1

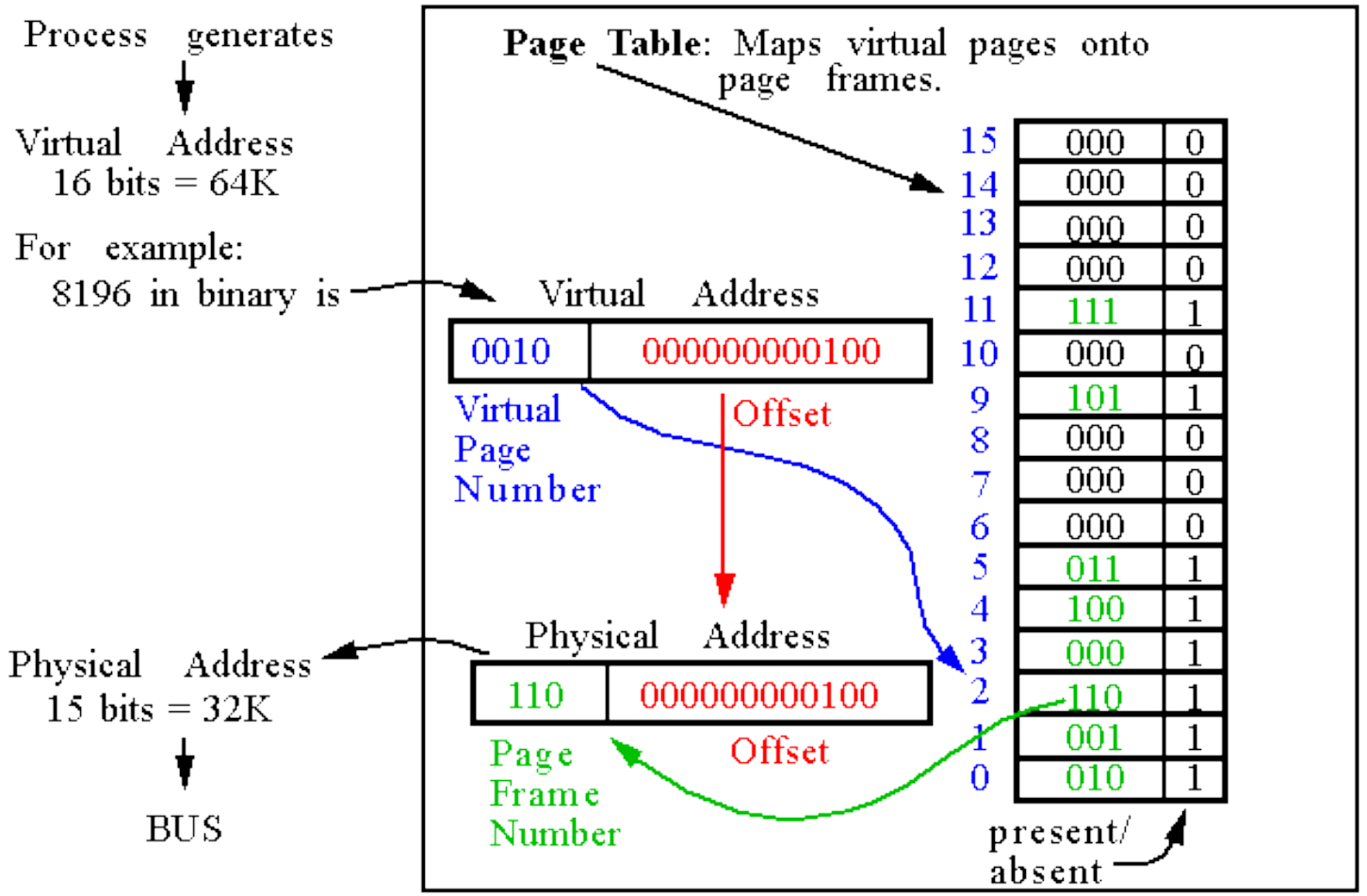
- Assume that *n*=2 and *m*=4
- The logical address space is $2^m = 2^4 = 16$ bytes
- The page size is $2^n = 2^2 = 4$ bytes
- No. of logical pages is $2^{m-n} = 2^2 = 4$ pages
- Assume the physical memory is 32 bytes divided into 32/4=8 frames.



- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) +3]
- Logical address 4 (page 1, offset 0) maps to physical address 24 [= (6× 4) + 0]

Paging

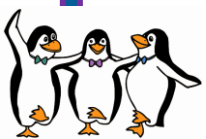
- Basic Method
  - Example 2



Paging

Basic Method

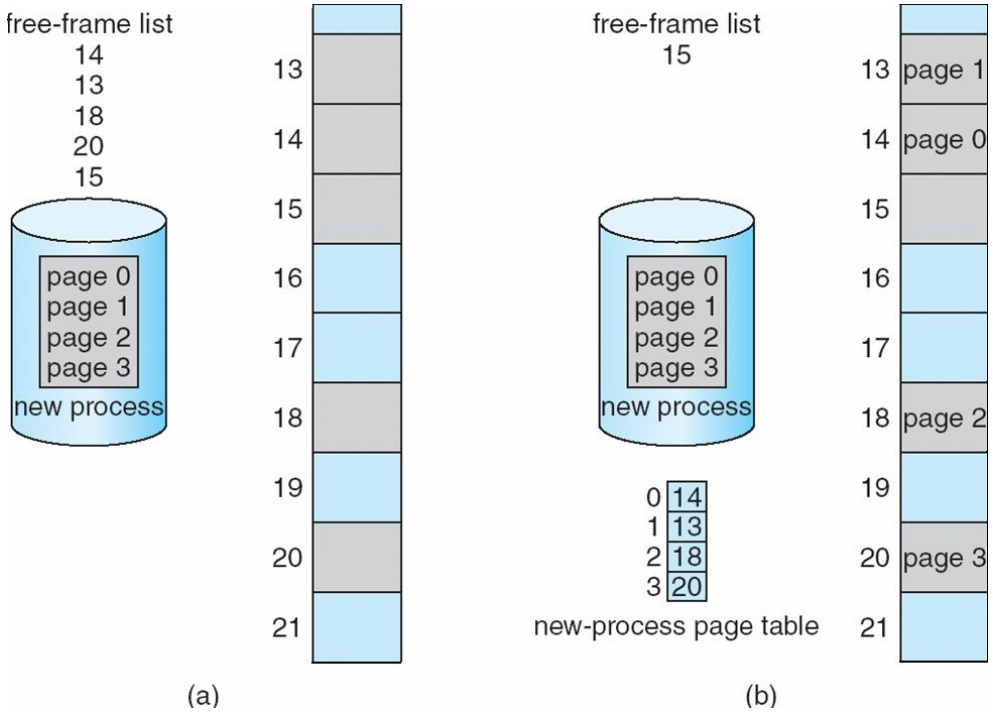
- Calculating internal fragmentation
  - Assume that the page size = 2,048 bytes
    - If the process size = 72,766 bytes, then this process needs 35 pages + 1,086 bytes
    - So we have internal fragmentation of 2,048 - 1,086 = 962 bytes
  - For the worst case, the fragmentation = 1 frame – 1 byte
  - On average, the fragmentation = 1 / 2 frame size
  - So small frame sizes are desirable?
    - But each page table entry takes memory to track
  - Therefore, page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- The size of physical memory in a paged memory system is different from the maximum logical size of a process.
- Paging lets us use physical memory that is larger than what can be addressed by the CPU's address pointer length



Paging

Basic Method

- Example:
  - When a process arrives to be executed, if it requires n pages, at least n frames must be available in memory.
  - The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process, and so on.



Before allocation

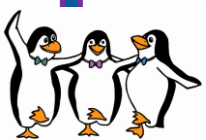
After allocation

Paging

Implementation of Page Table

- Some OSes allocate a page table for each process and stores a pointer to the page table with the other register important registers for that process
- Other OSes provide one or at most a few page tables, to decrease the context-switched overhead.

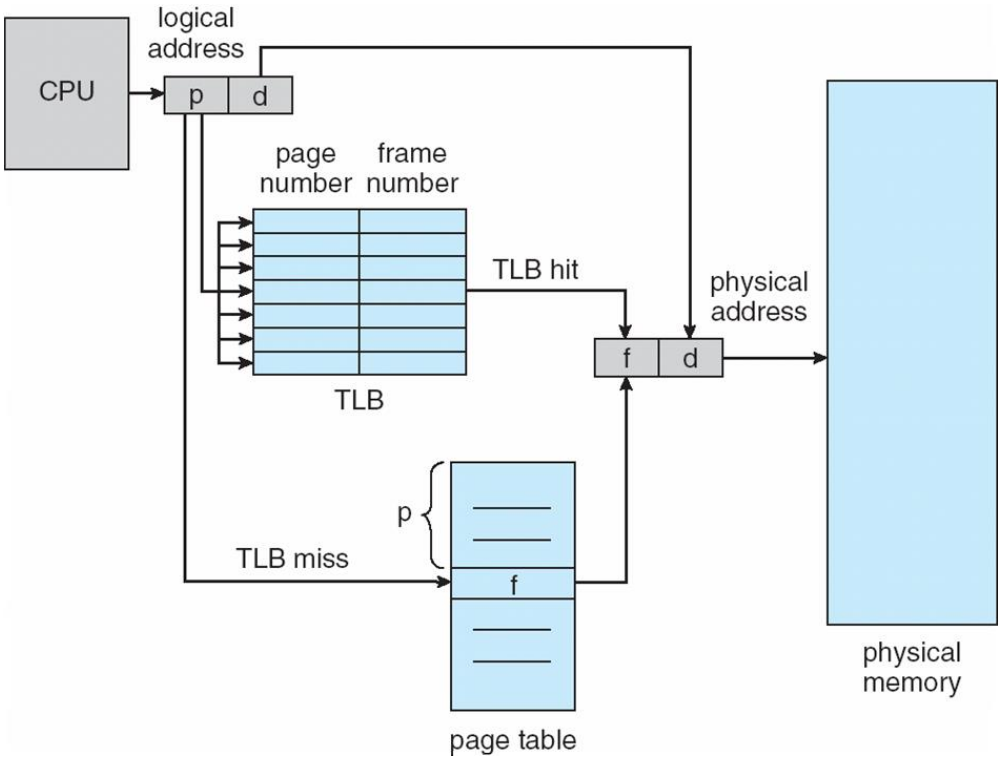
- **The page table is kept in the main memory and:**
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates the size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
  - The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)
    - This very fast cache and does not introduce any performance penalty

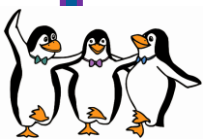


Paging

Implementation of Page Table

- The TLB contains only a few of the page-table entries (like a cache to the page table)
- When a logical address is generated by the CPU, its page number is compared with all keys simultaneously in the TLB.
- If the page number is found, its frame number is used to access memory.
- If the page number is not in the TLB (**TLB miss**), a memory reference to the page table must be made and later may be saved in the TLB for later access.

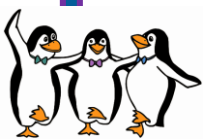




Paging

Implementation of Page Table

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry that uniquely identifies each process to provide address-space protection for that process
  - The ASID allows the TLB to contain entries for several different processes simultaneously.
    - Otherwise need to flush the TLB 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 (like LRU algorithm)
  - Some entries can be **wired down** for permanent fast access
    - Such as the key kernel code



Paging

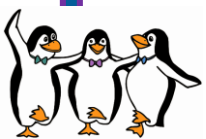
Effective Access Time

- Assume that the Associative Lookup = ε time unit
  - It can be < 10% of memory access time
- Assume that the memory access time is T
- Assume the TLB hit ratio = α
  - Hit ratio: percentage of times that a page number is found in the associative registers of the TLB; ratio related to number of associative registers
- Then, the **Effective Access Time** (**EAT**)

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

$$= (2 - \alpha)T + \varepsilon$$

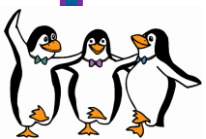
- Consider α = 80%, ε = 1ns for TLB search, 100ns for memory access
  - EAT = (2-0.80) x 100 + 1 = 121ns
- Consider more realistic hit ratio, α = 99%, ε = 1ns for TLB search, 100ns for memory access
  - EAT = (2-0.99) x 100 + 1 = 102ns



Paging

Memory Protection

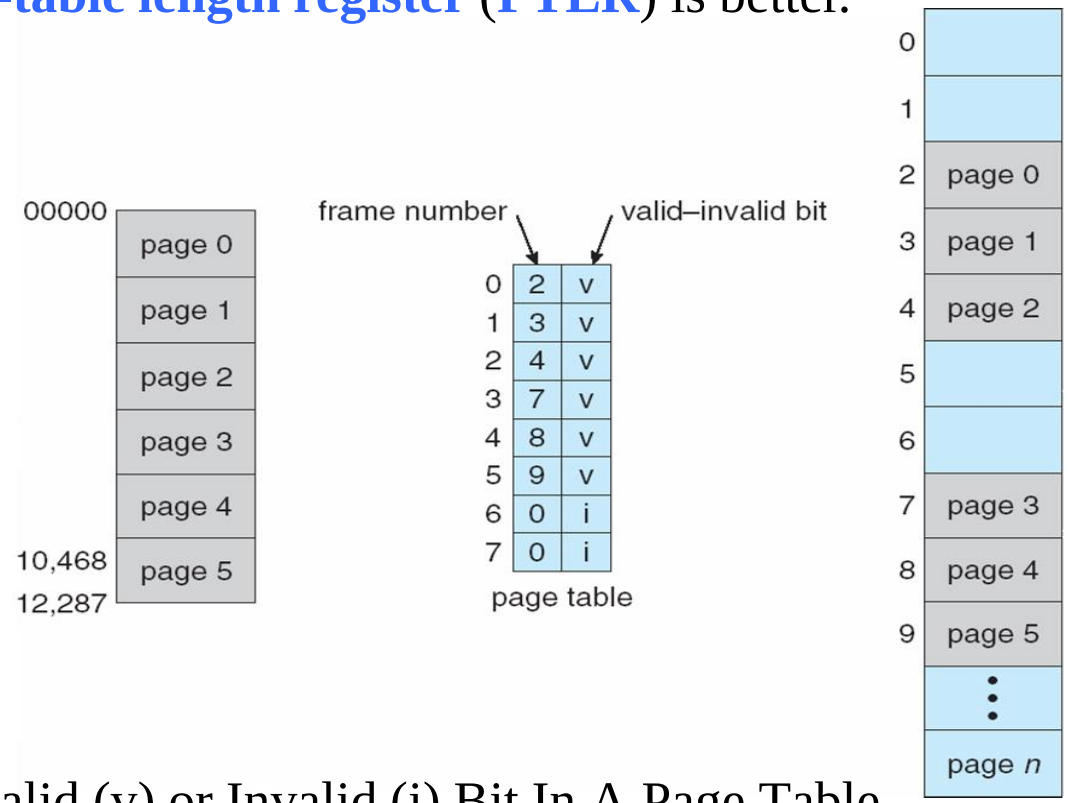
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- A **valid-invalid** bit is attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process's logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process's logical address space
  - Or use **page-table length register** (**PTLR**) to indicate the size of the page table
    - This value is checked against every logical address to verify that the address is in the valid range for the process.
- Any violations result in a trap to the kernel



Paging

Memory Protection

- Assume a system with a 14-bit address (0-16383), and a page size of 2 KB
- If a program has an only addresses 0 to 10468, it will occupy pages 0, 1, 2, 3, 4, and 5.
  - The addresses in page 5 (from 10468 to 12287) are invalid.
- Therefore, using the **page-table length register** (**PTLR**) is better.



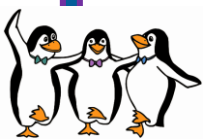
Valid (v) or Invalid (i) Bit In A Page Table

Paging

Shared Pages

- An advantage of paging is the possibility of sharing common code
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
    - » Reentrant code is non-self-modifying code: it never changes during execution.
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed

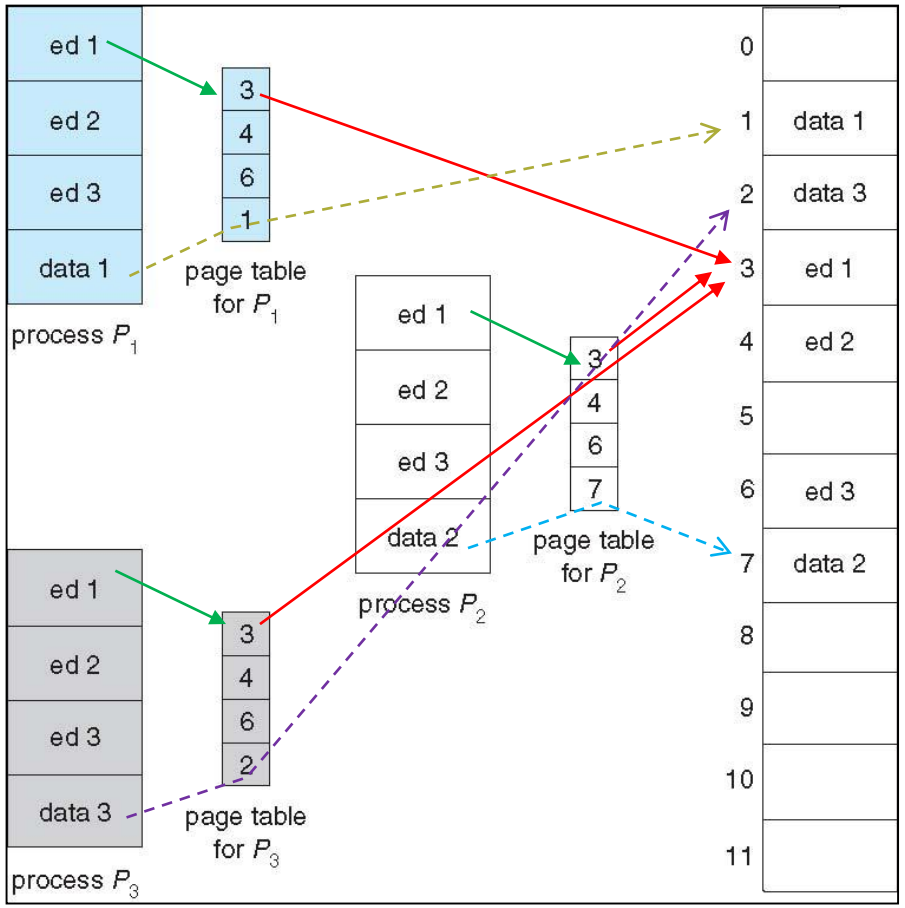
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space



Paging

Shared Pages

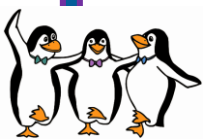
- Example: three processes run the same text editor.
  - Each process maps to the same frames in the memory.
  - Each process has its own copy of registers and data storage to hold the data for the process's execution.



Structure of the Page Table

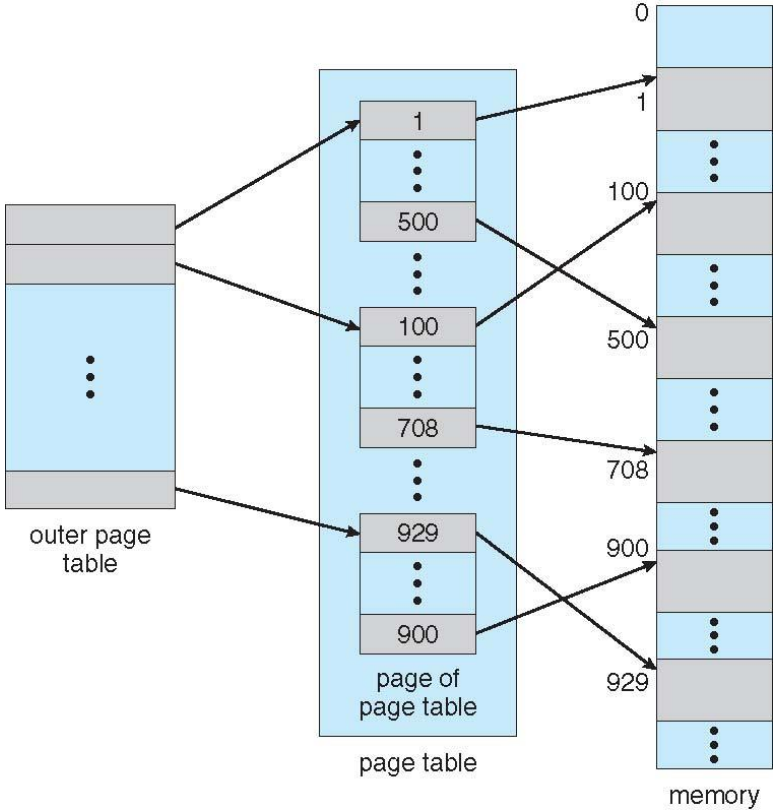
Structure of the Page Table

- 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, then 4 MB of physical address space / memory needed for the page table alone
    - We don't want to allocate that big memory contiguously in main memory
- We can use:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables



Structure of the Page Table

- 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 Page-Table Scheme

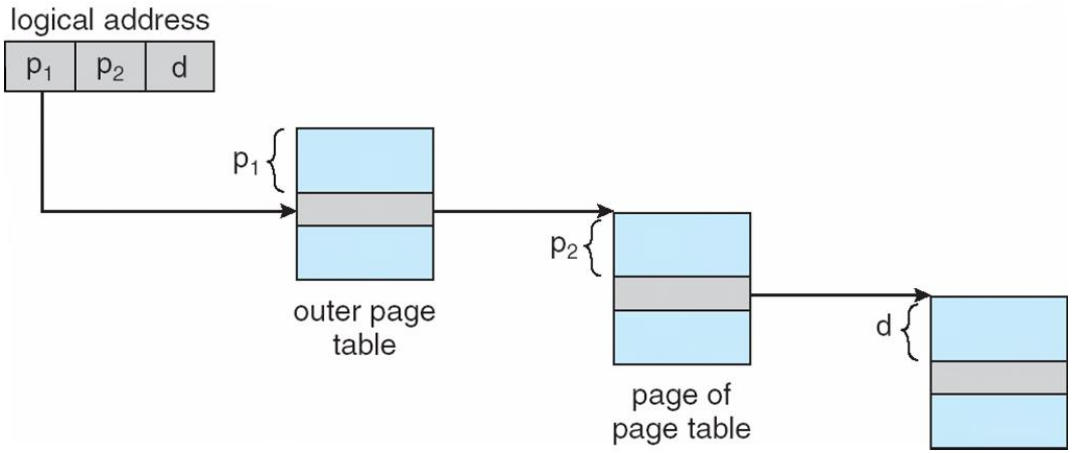
Structure of the Page Table

Hierarchical Page Tables: Two-Level Paging Example 1

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits and a 10-bits page offset
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number and a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

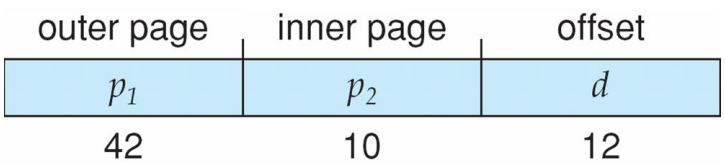
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

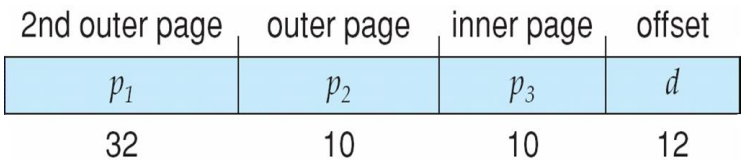


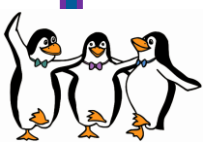
Structure of the Page Table

Hierarchical Page Tables: Two-Level Paging Example 2

- For a system with a 64-bit Logical Address Space
  - Even two-level paging scheme is 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 2^{nd} outer page table
    - But the 2^{nd} outer page table is still 2^{34} bytes in size
      - » We need 4 memory access to get to one physical memory location
- Therefore, hierarchical page tables are inappropriate for 64-bit architectures,



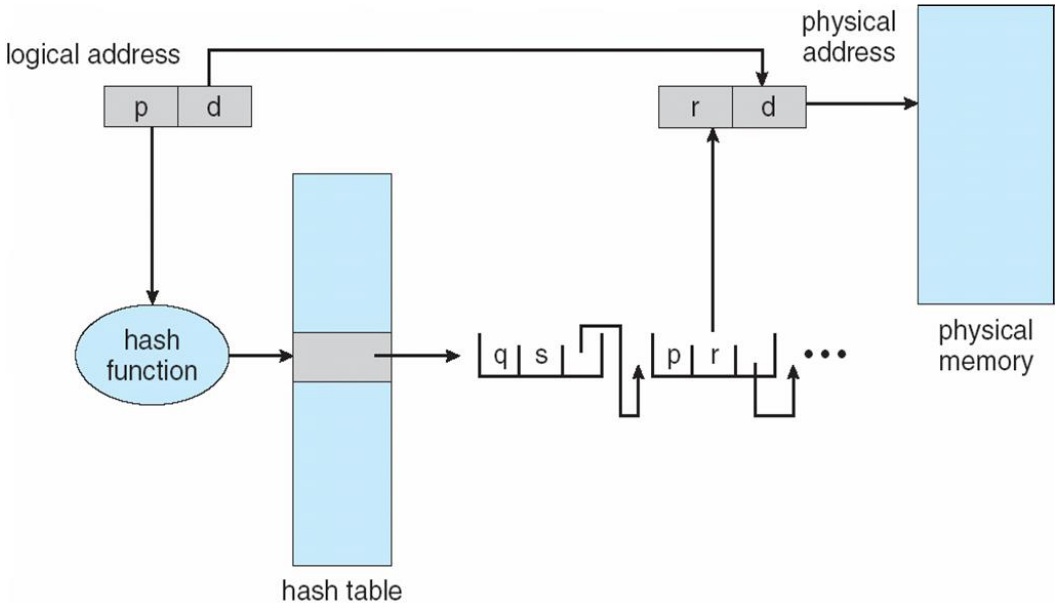




Structure of the Page Table

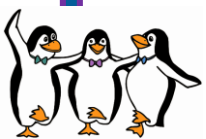
Hashed Page Tables

- Common to be used in address spaces > 32 bits
- The virtual page number is hashed into the page table
  - This page table contains a chain of elements hashing to the same location. Each element contains
    1. The virtual page number
    2. The value of the mapped page frame
    3. A pointer to the next element
- The virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted



Structure of the Page Table

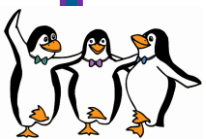
- Hashed Page Tables
  - Variation for 64-bit addresses is the **clustered page tables**
    - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
    - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)



Structure of the Page Table

Inverted Page Table

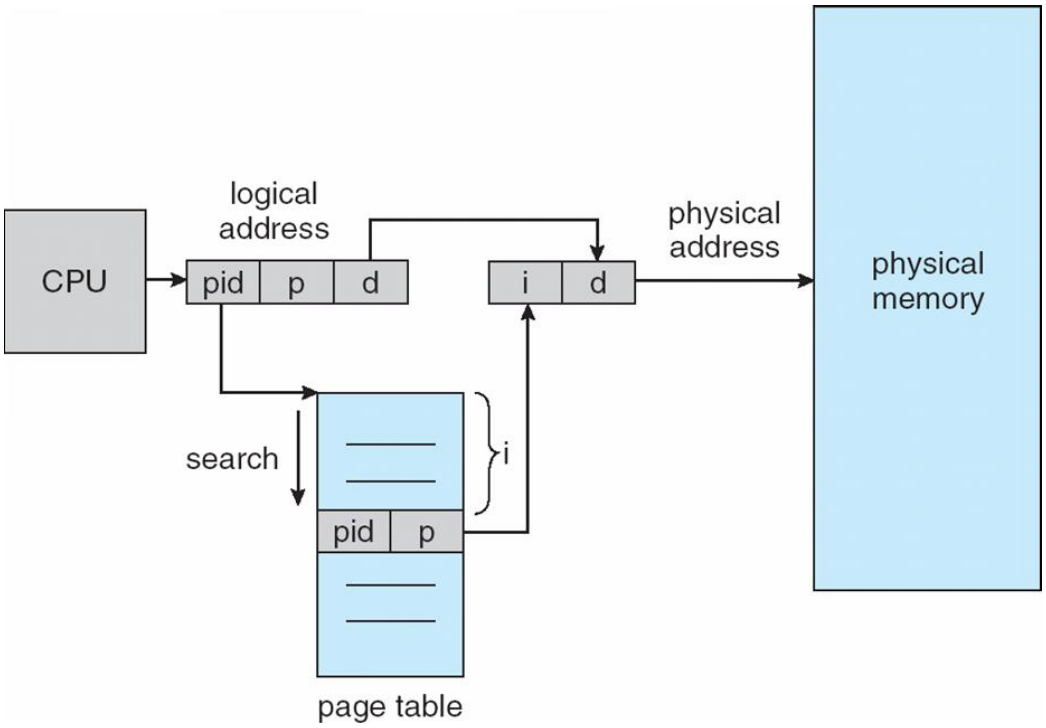
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages (frames)
  - An inverted page table has one entry for each real page of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
  - This decreases the memory needed to store each page table, but increases time needed to search the table when a page reference occurs
  - Thus, only one page table is in the system, and it has only one entry for each page of physical memory.
- Use hash table to limit the search to one (or at most a few) page-table entries
  - TLB can also accelerate the access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address



Structure of the Page Table

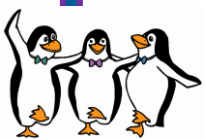
Inverted Page Table Architecture used in IPM RT

- Each virtual address in the system consists of a triple:
  <process-id, page-number, offset>
- Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier.
- To reference a memory, the <process-id, page-number>, is sent to the memory subsystem, which will search the inverted page table for a match:
  - If a match is found (at entry i) then the physical address <i, offset>.
  - If no match is found, then an illegal address access has been attempted.



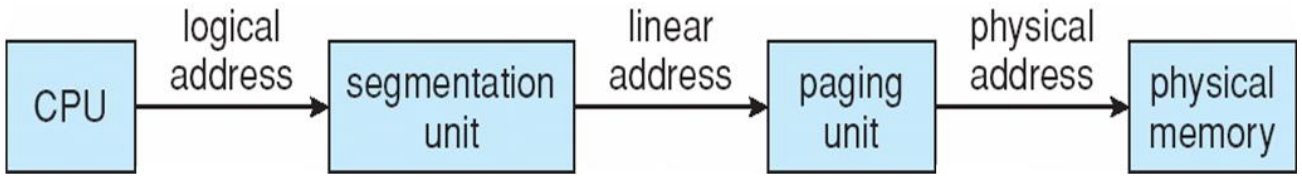
Example: The Intel 32 and 64-bit Architectures

- Dominant the industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture



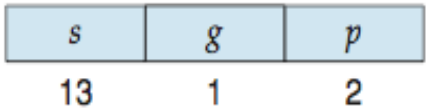
Example: The Intel IA-32 Architecture

- Memory management in IA-32 systems is divided into two components: segmentation and paging:
  - It works as:
    - The CPU generates logical address and sends it to segmentation unit.
    - The segmentation unit produces a linear address for each logical address.
    - The linear address is then given to the paging unit, which generates the physical address in main memory.

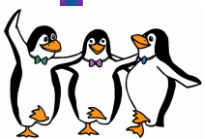


Example: The Intel IA-32 Architecture

- Memory management in IA-32 systems is divided into two components: segmentation and paging:
  - IA-32 Segmentation:
    - The segment can be up to 4 GB and up to 16K segments per process
    - The logical address space of a process is 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**))
    - The logical address is a pair (selector, offset), where the selector is a 16-bit number:

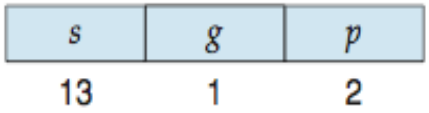


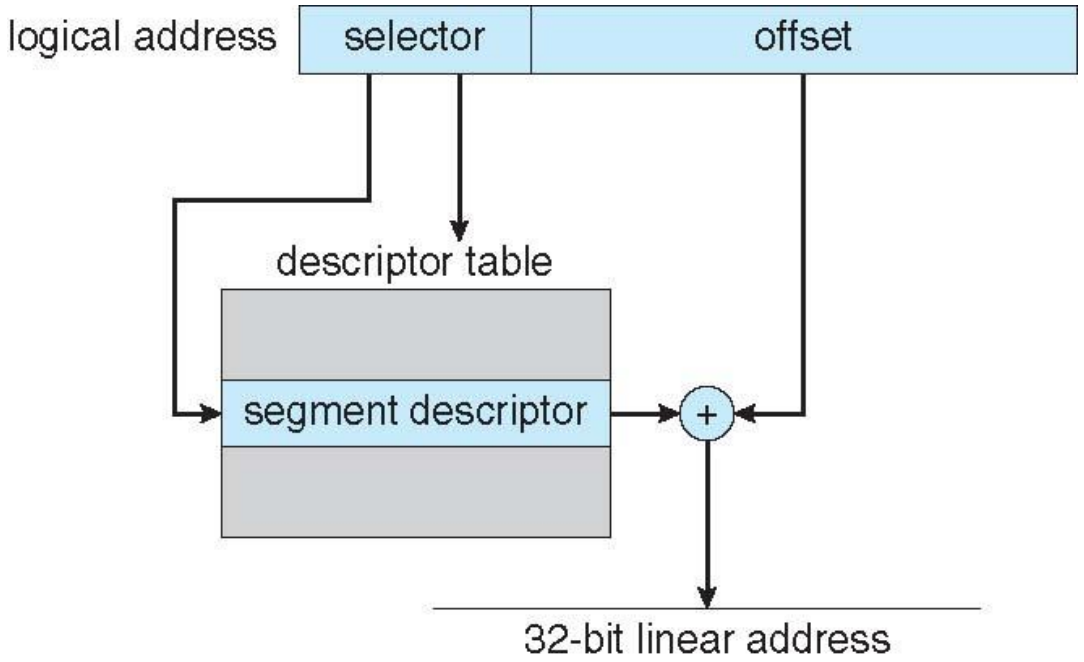
      - s designates the segment number,
      - g indicates whether the segment is in the GDT or LDT,
      - p deals with protection.
      - The offset is a 32-bit number specifying the location of the byte within the segment in question



Example: The Intel IA-32 Architecture

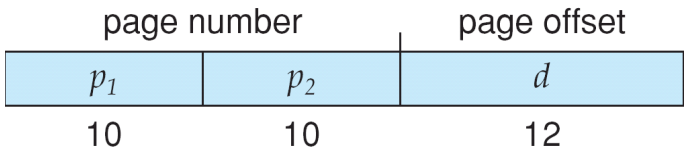
- Memory management in IA-32 systems is divided into two components: segmentation and paging:
  - IA-32 Segmentation:





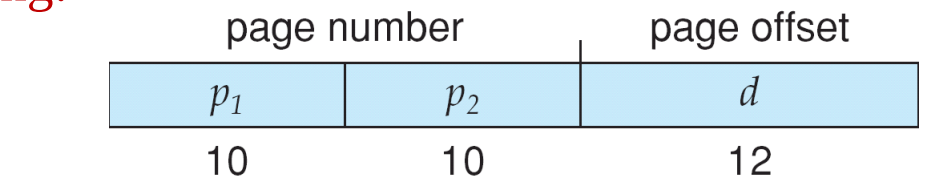
Example: The Intel IA-32 Architecture

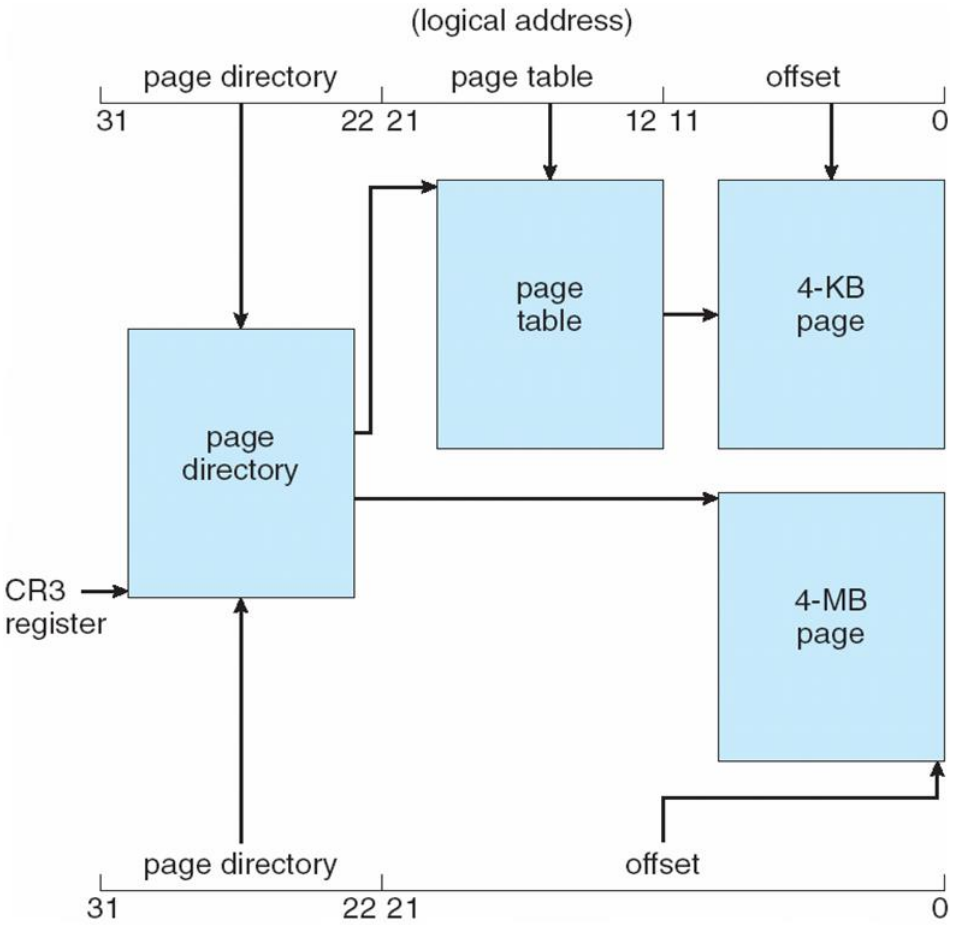
- Memory management in IA-32 systems is divided into two components: segmentation and paging:
  - IA-32 Paging:
    - It allows a page size of either 4 KB or 4 MB.
    - For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



Example: The Intel IA-32 Architecture

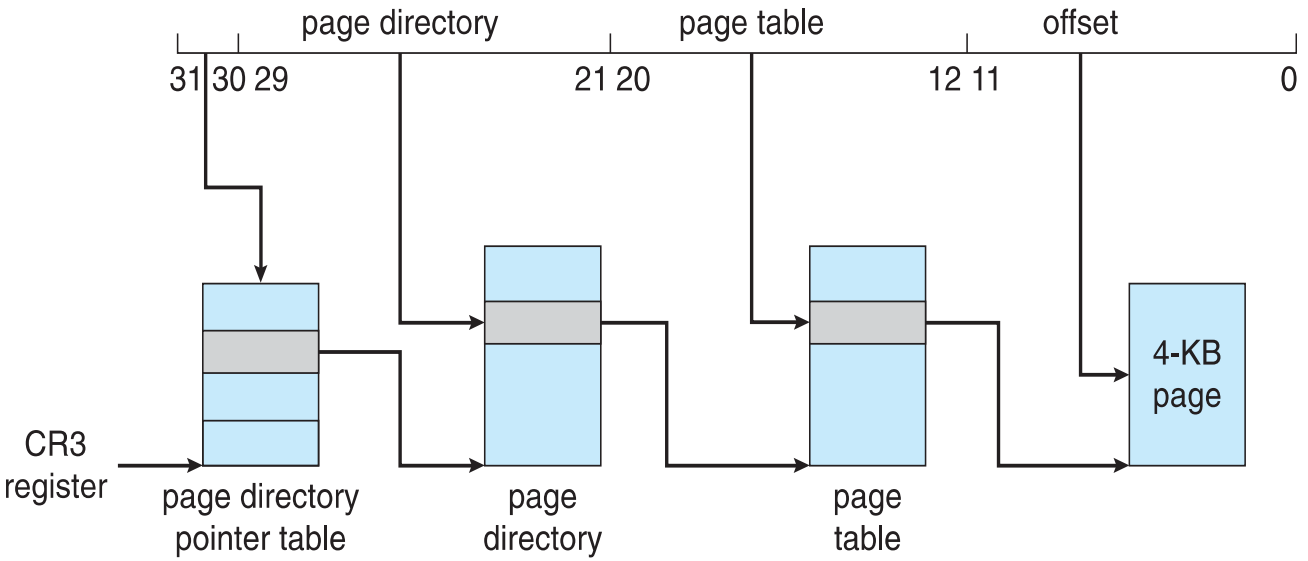
- Memory management in IA-32 systems is divided into two components: segmentation and paging:
  - IA-32 Paging:





Example: The Intel IA-32 Architecture

- Intel 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



Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices)
- ARM processors are energy efficient, and 32-bit CPU
- It supports the following page sizes:
  - 4 KB and 16 KB pages
    - Uses two-level paging
  - 1 MB and 16 MB pages (termed **sections**)
    - Uses One-level paging