Chapter 8: Main Memory

Chapter 8: 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

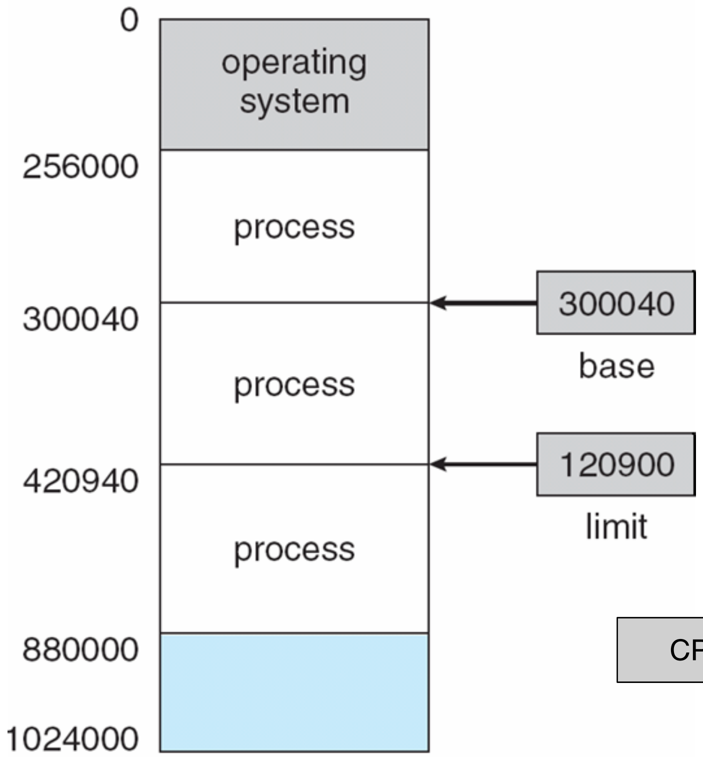
- To understand how OS organizes memory hardware
- To discuss various memory-management techniques, including **paging** and **segmentation**
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
  - So far, **the entire program** should be uploaded in the memory
- Main memory and registers are the only storages CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
  - Ex> Read 32-bit data from memory address of 0x1ABD028F and store it in a register A, add values in register A and B, …
- Register access 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

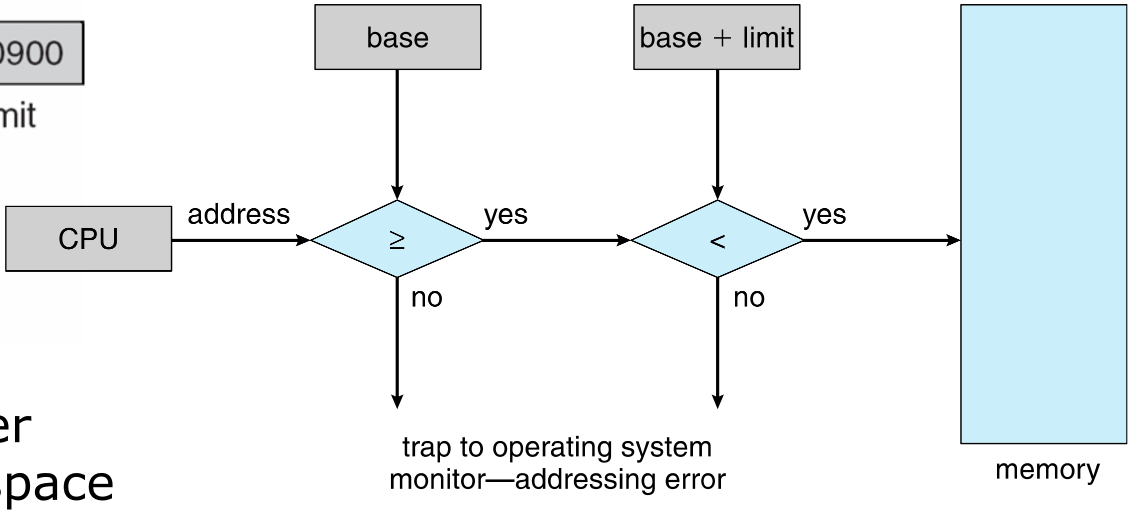
Base and Limit Registers

- A pair of **base** and **limit registers** defines the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



A base and a limit register define a logical address space

Hardware address protection with base and limit registers

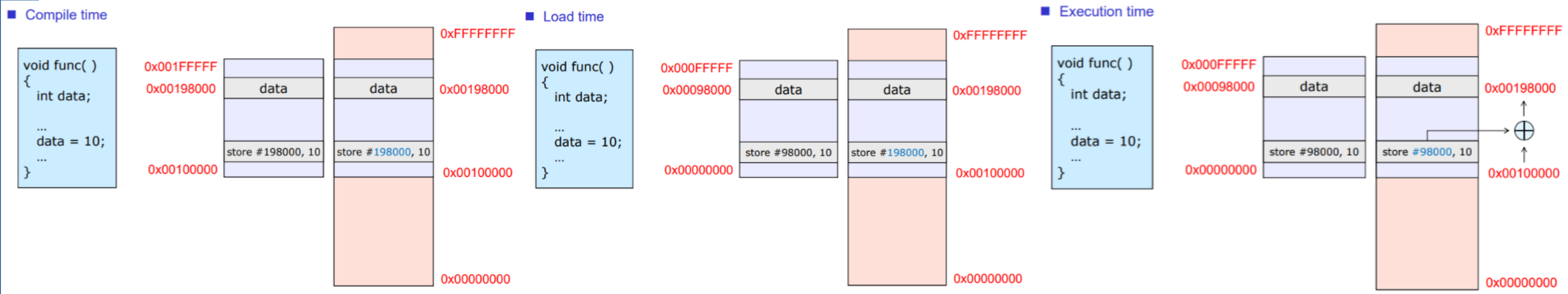


Address Binding

- Programs on disk are brought into memory for execution from an **input queue**
  - Although the *address space of the computer* may start at 00000, the first address of the user process needs not to be 00000.
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses are usually symbolic
  - 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

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location is known before execution, absolute code can be generated; must recompile code if starting location changes. Only applied to .com files in MS-DOS.
  - **Load time**: Must generate relocatable code if memory location is not known at compile time. In this case, final binding is delayed until load 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)

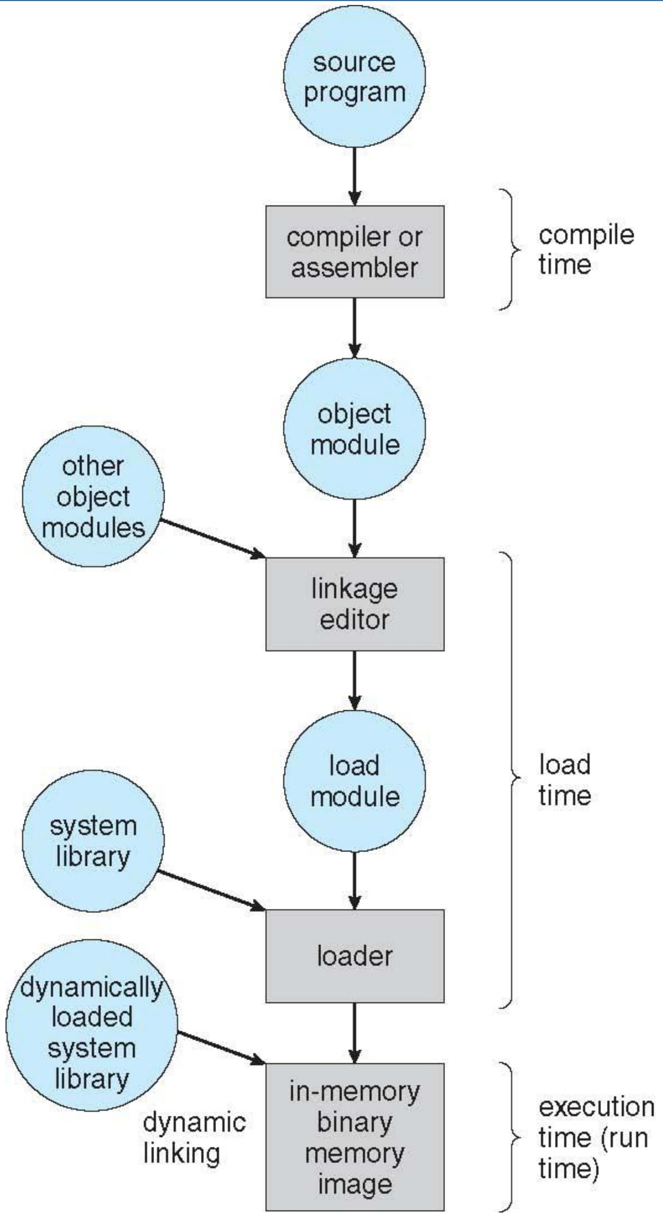


```
void func( )
{
  int data;
  ...
  data = 10;
  ...
}
```

```
void func( )
{
  int data;
  ...
  data = 10;
  ...
}
```

```
void func( )
{
  int data;
  ...
  data = 10;
  ...
}
```

Multistep Processing of a User Program



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by a process; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical (virtual) address space is used for address-binding at both compile-time and load-time; logical and physical addresses differ at execution-time
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses designed by a computer architecture

Memory-Management Unit (MMU)

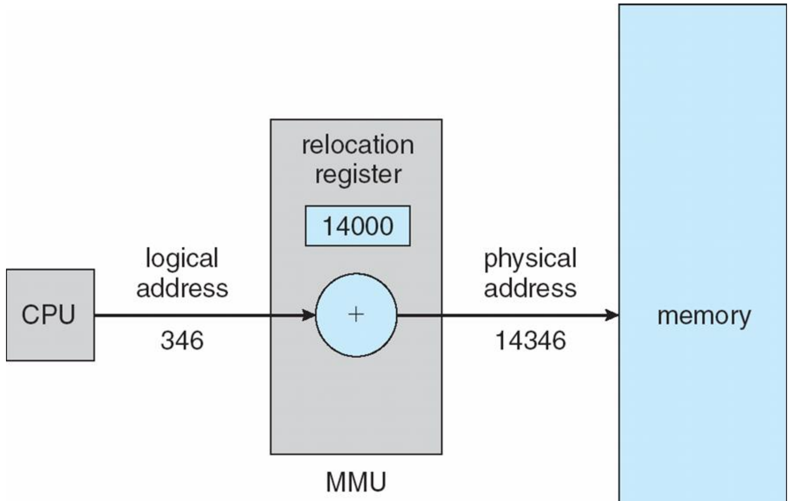
- Hardware device that maps virtual to physical address at run time
- Many methods are possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers (for segments; will be introduced later)
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time mapping occurs when reference is made to locations in memory
  - Logical addresses are mapped to physical addresses

Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required

  - Implemented through program design

  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

- **Static linking** – system libraries and program codes are combined by the loader into the binary program image
- Dynamic linking –linking is postponed until the execution time
- A small piece of code, **stub**, is used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if the routine is in processes' memory address
  - If it is not in its address space, add it to the address space
- Dynamic linking is particularly useful for libraries
- Consider applicability to patching system libraries
  - Versioning may be needed
  - This is known as **shared libraries**

Dynamic Linking

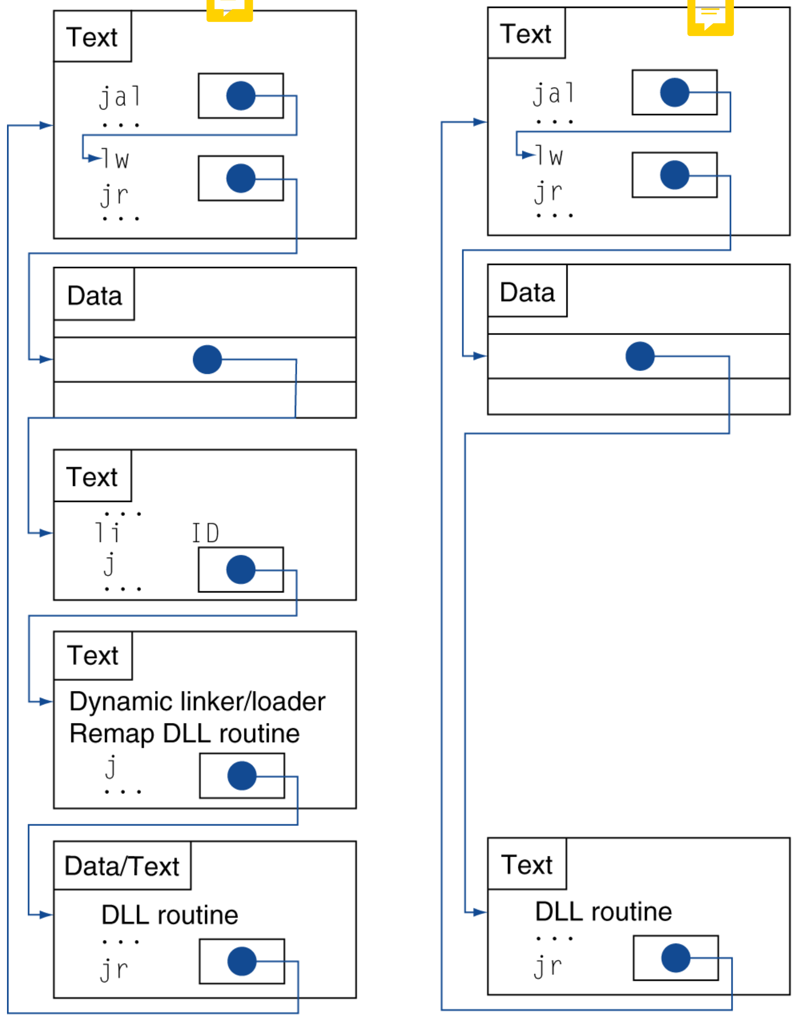
(Copied form "Computer Organization and Design: The Hardware/Software Interface, 5th Ed", David A. Patterson and John L. Hennessy, Morgan Kaufmann, 2013)

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

jal: jump and link
lw: load word
jr: return

How Does OS Handle Processes in Main Memory?

- With the knowledge that we have learned, we know OS uploads programs into main memory in the form of processes

- We do not know how to handle processes that require space larger than the main memory size

  1) When the sum of multiple processes' spaces is larger than the size of main memory

     -> swapping

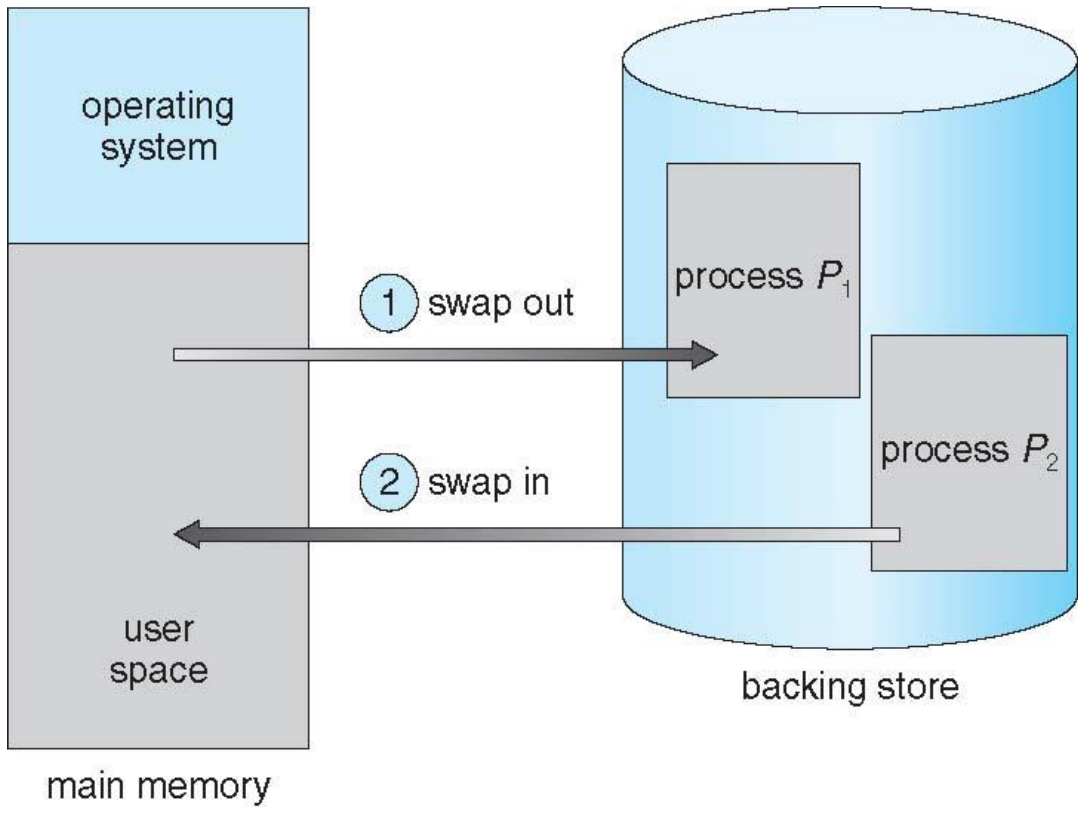
  2) When the required space of a single process is larger than the size of main memory

     -> paging

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space required by processes can exceed the real physical memory
- **Backing store** – A fast disk large enough to accommodate copies of all memory images for all users; **must provide direct access to these memory images**
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk or in memory
  - Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
  - The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
  - If all processes are in memory, no swapping is required
- Context switch time with swapping can be very long
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000ms
  - Plus swap in a process with the same size
  - Total context switch swapping component time of 4000ms (4 seconds)
- This context switch time with swapping can be reduced if the swapping memory size is reduced – by knowing how much memory will be used in real execution
  - Users must keep the system be informed of any changes in memory requirements using system calls (`request_memory()` and `release_memory()`)

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping out a process with
  - Pending I/O (asynchronously accessing the user memory for I/O buffers) – can't swap out this process because I/O would attempt to use its memory that now belongs to wrong process
  - Transferring I/O via kernel space can solve this problem
    - Known as **double buffering**, adds overhead
- Standard swapping is **not** used in modern operating systems
  - But modified version is common on many systems (i.e., UNIX, Linux, and Windows)
    - Swap only when free memory is extremely low
    - Another variation involves swapping partial portions of processes—rather than entire processes—to decrease swap time

Swapping on Mobile Systems

- Typically not supported
  - Flash memory based
    - Smaller amount of space
    - Limited number of write cycles (~1M cycles)
    - Poor throughput between flash memory and CPU on mobile platform (NAND: fast writing, NOR: fast reading)
- Instead use other methods to free memory if the remaining memory is 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 the system has low free memory, but first it writes **application state** to a flash memory for fast restart
  - Both OSes support paging as discussed below

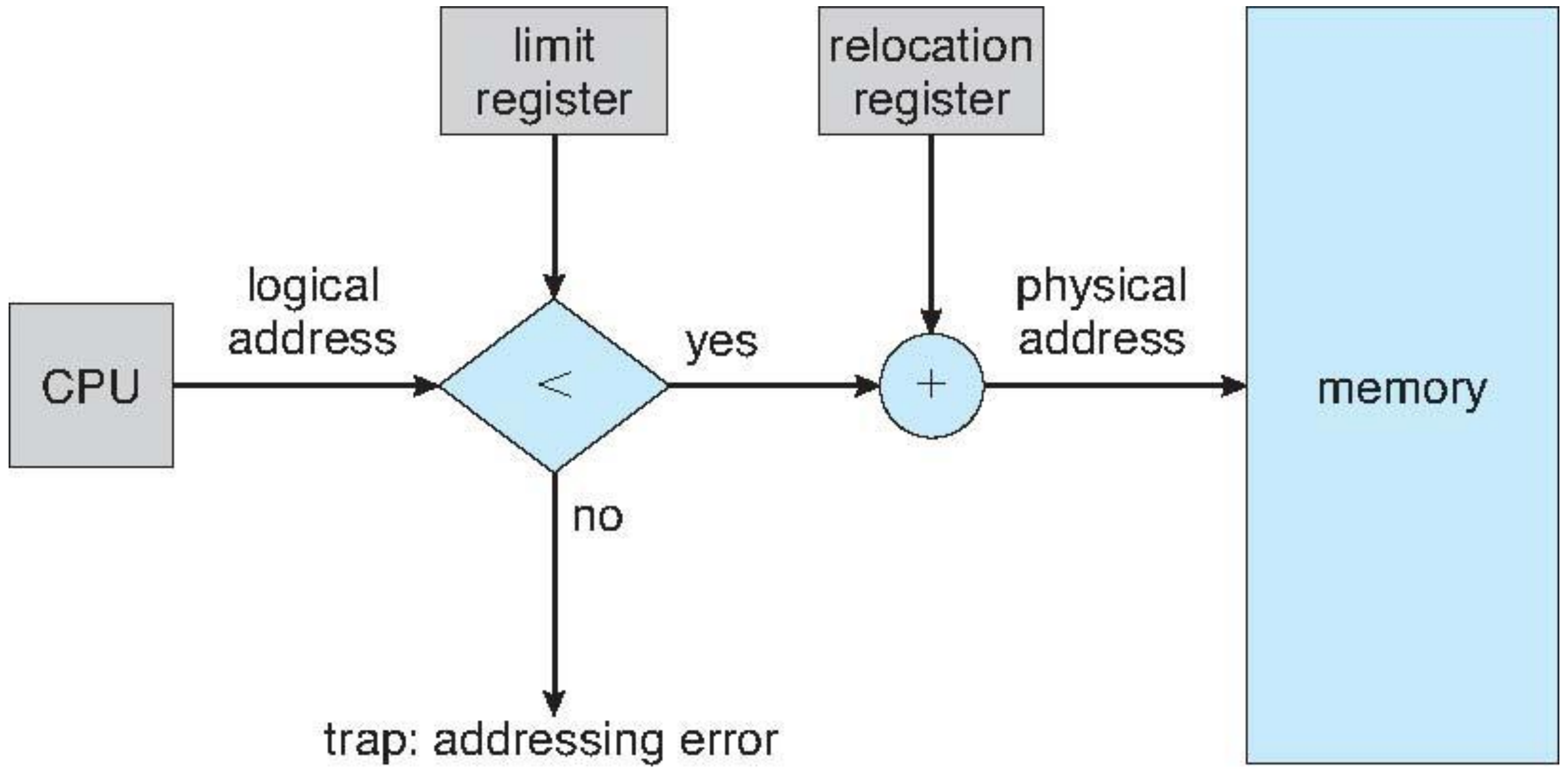
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- 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 single contiguous section of memory

Contiguous Allocation (Cont.)

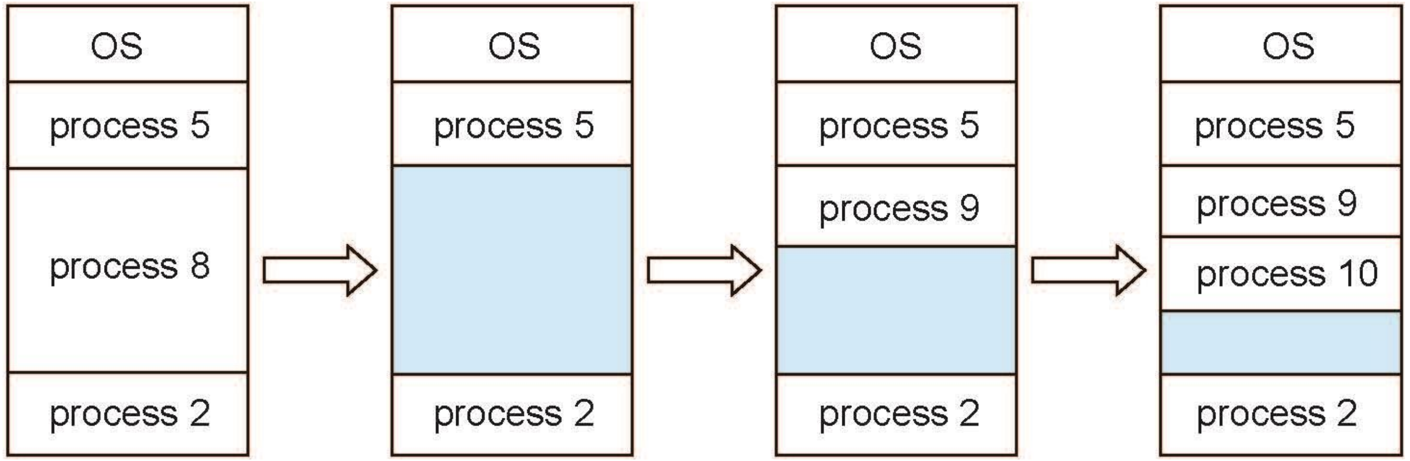
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - The relocation-register scheme can maps logical address *dynamically*
    - This scheme allows a kernel code (e.g., a device driver or other operating-system service) being **transient**. Therefore, the kernel changes its size by the currently loaded kernel modules.

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

- Multiple-partition allocation
  - Degree of multiprogramming is limited by the number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, OS allocates a memory hole large enough to accommodate it
  - When an exiting process frees its partition, adjacent free holes are combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

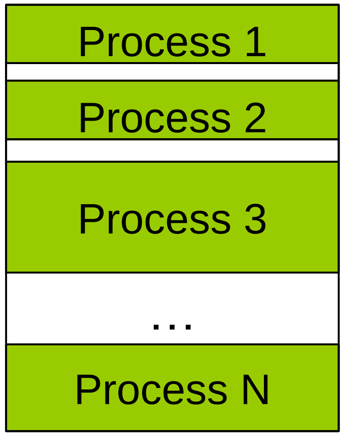
How to satisfy a request of size ***n*** memory allocation from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

Simulations have shown that first-fit and best-fit are better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – the sum of hole sizes is larger then a requested memory size, but it is not contiguous
- **Internal Fragmentation** – the memory allocated to a process is slightly larger than the requested memory; this fragmentation belongs to a process, but it is not being used
- A long-time simulation with the first fit analysis reveals that given *N* allocated blocks, 0.5 *N* blocks lost to fragmentation
  - That is, 0.5/(1 + 0.5) = 1/3 may be unusable -> **50-percent rule**



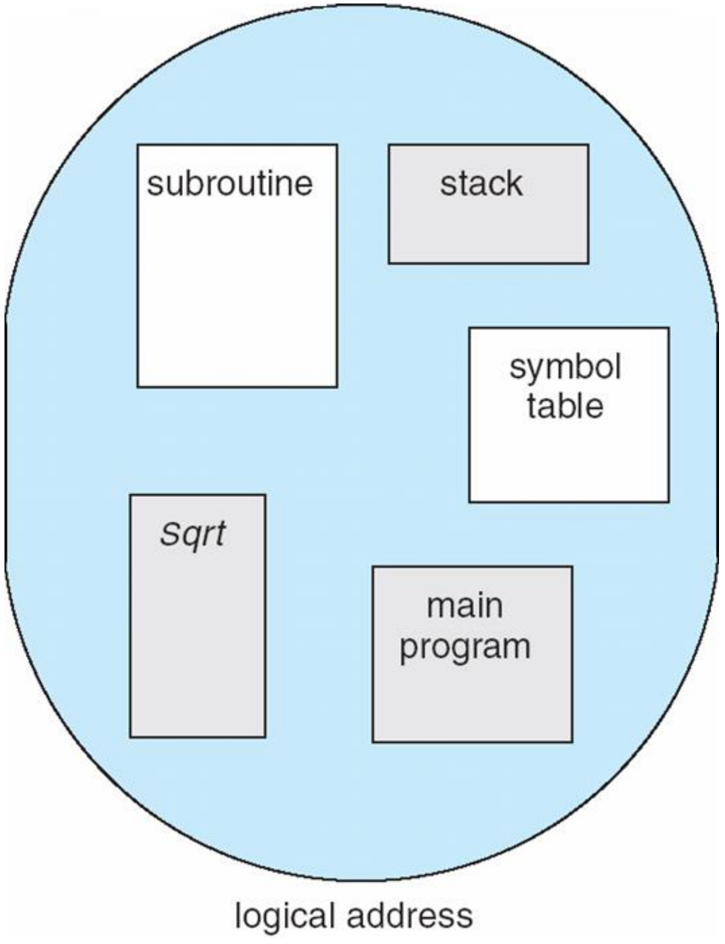
Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has the same fragmentation problems

Segmentation

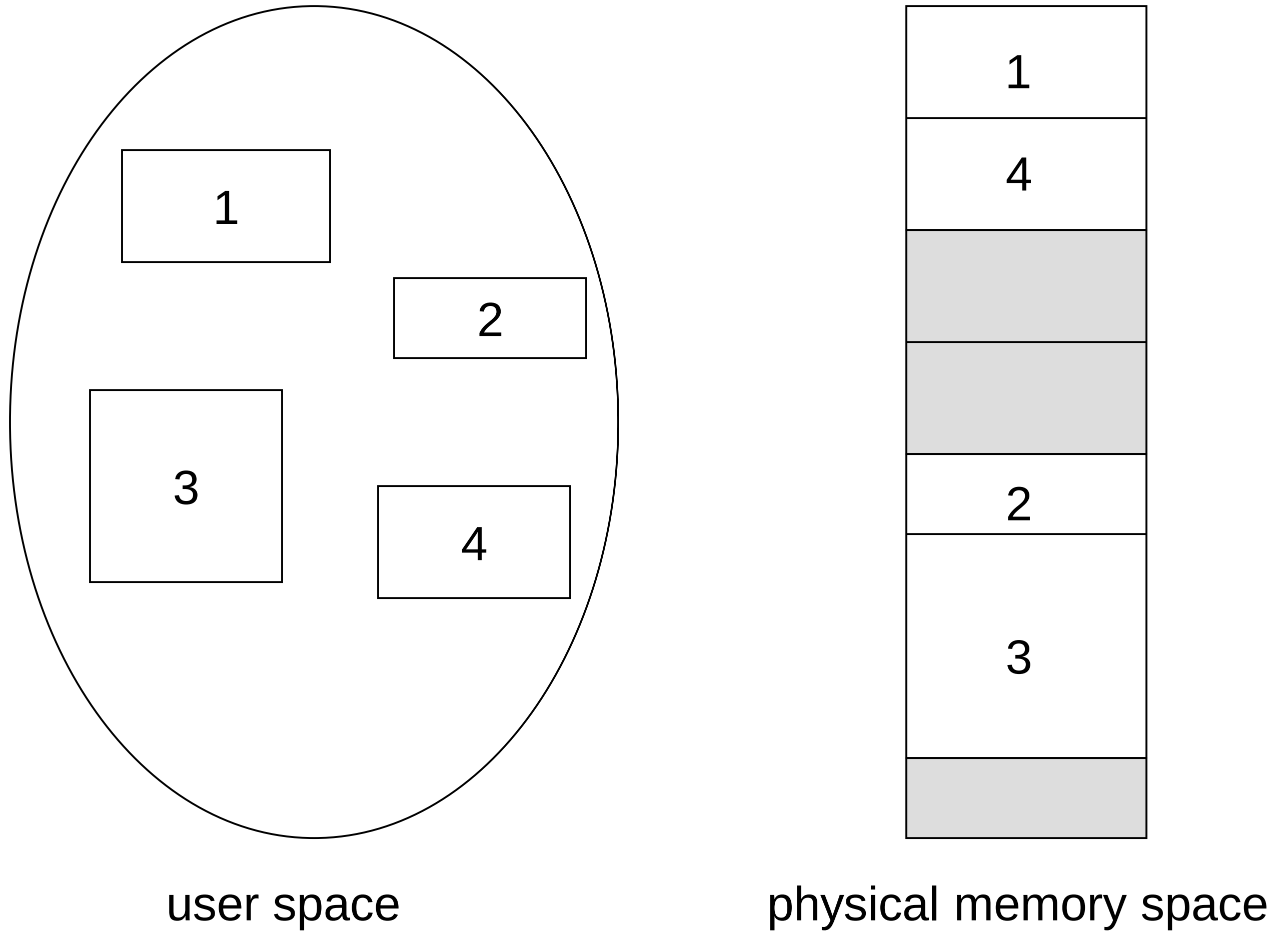
- Memory-management scheme that supports a user's view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables
    global variables
    common block
    stack
    symbol table
    arrays



User' s View of a Program

Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

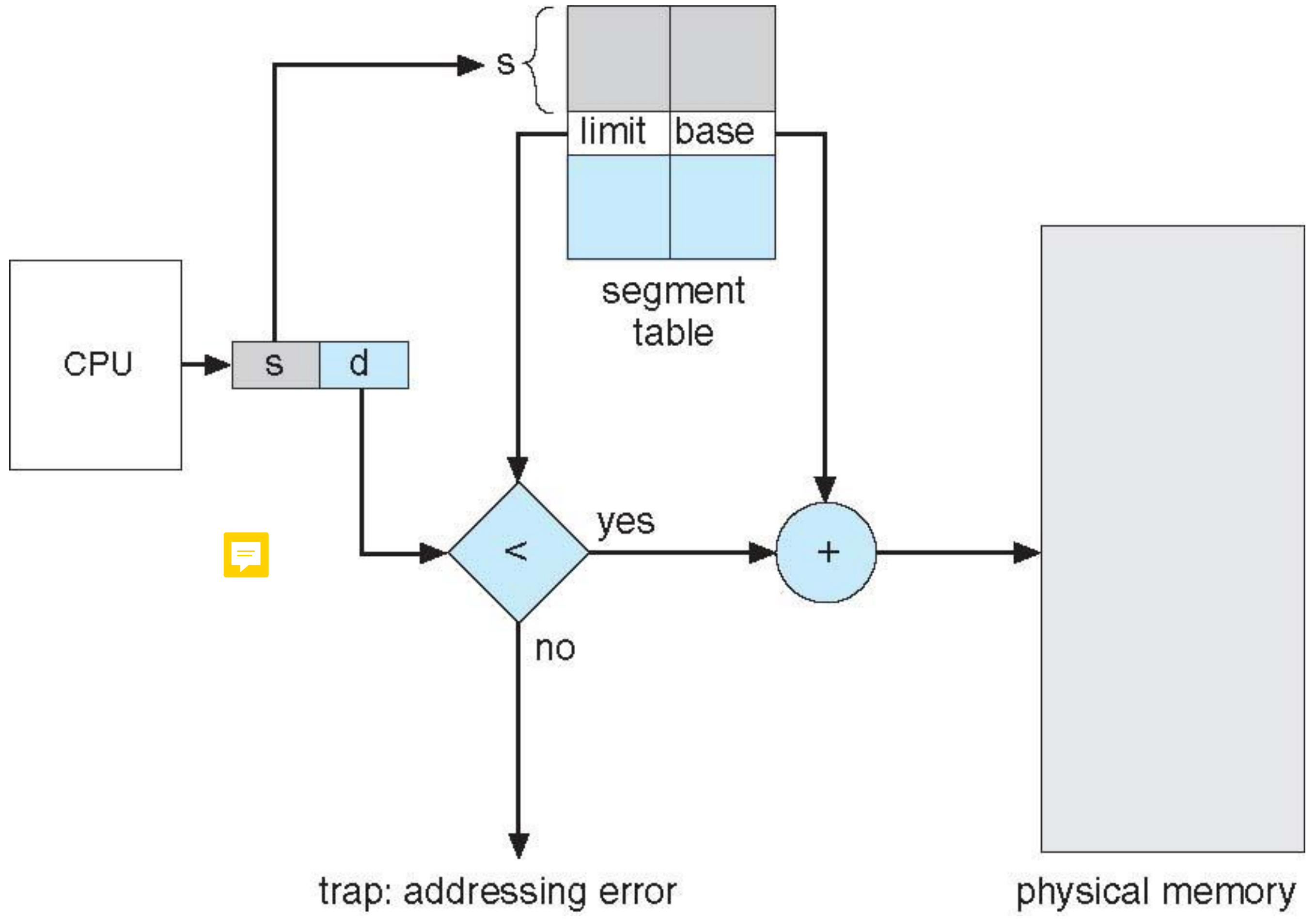
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

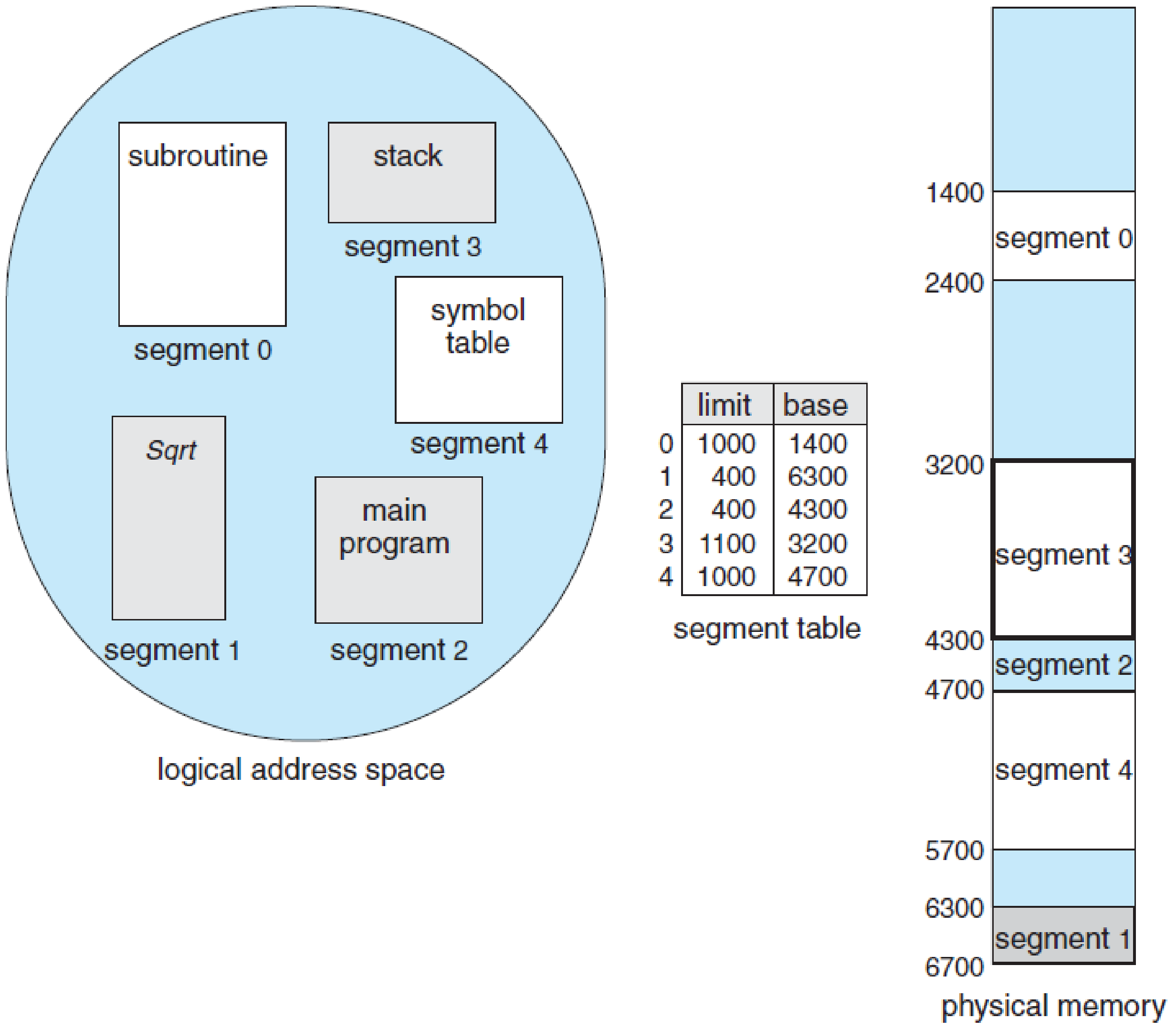
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation has the dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Segmentation Hardware



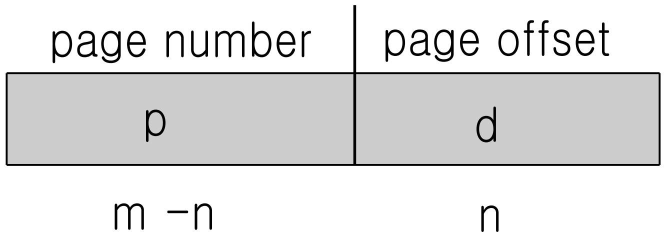
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Paging

- Segmentation is a memory management scheme that supports noncontiguous allocation; Paging is another scheme with noncontiguous allocation
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide **physical** memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide **logical** memory into blocks of the same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise splits into pages
- Still have Internal fragmentation

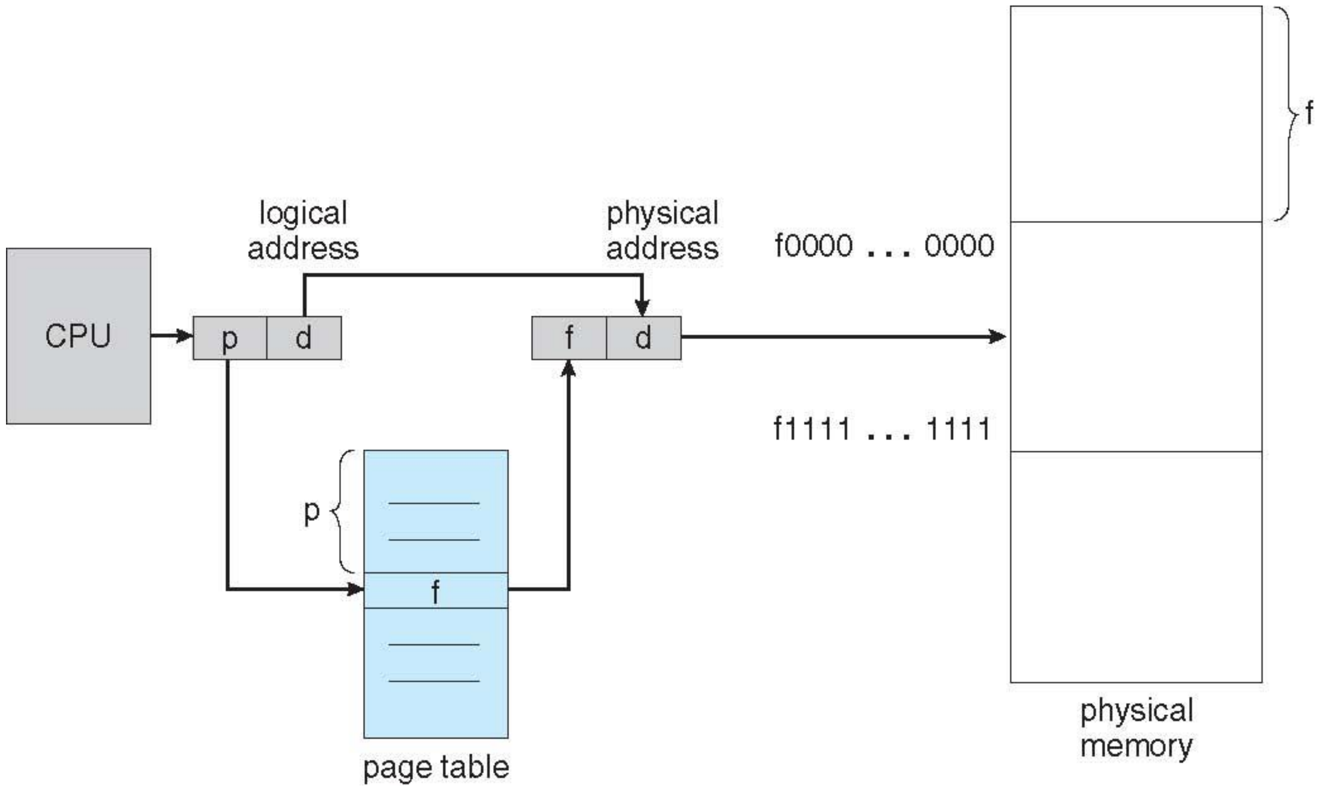
Address Translation Scheme

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

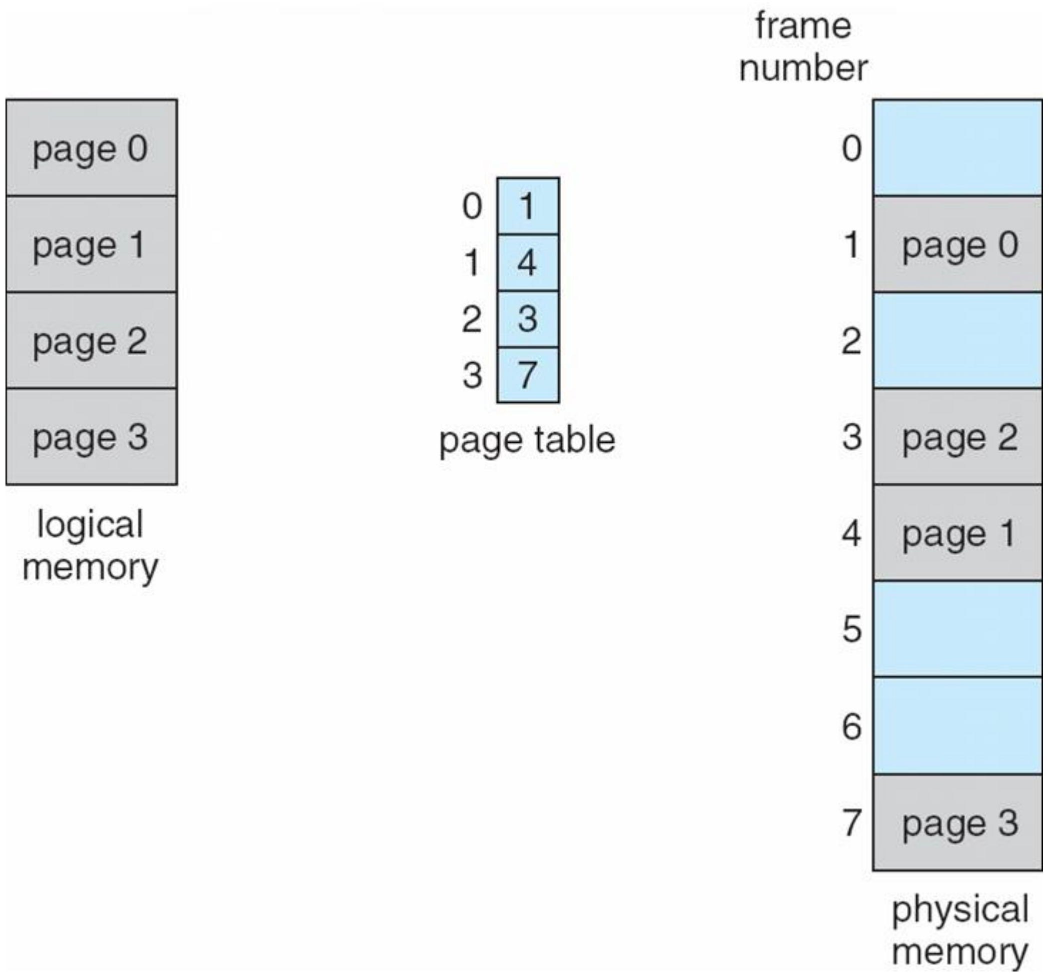


where logical address space 2^m and page size 2^n

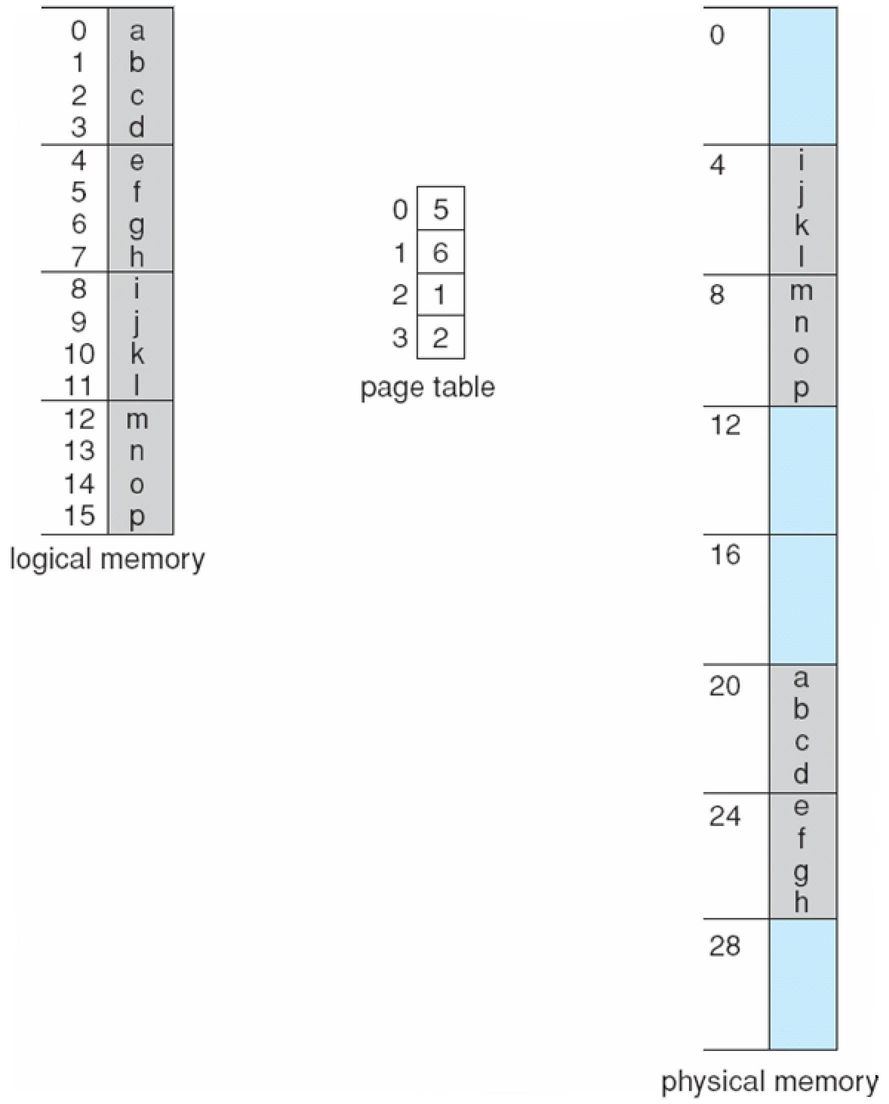
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example

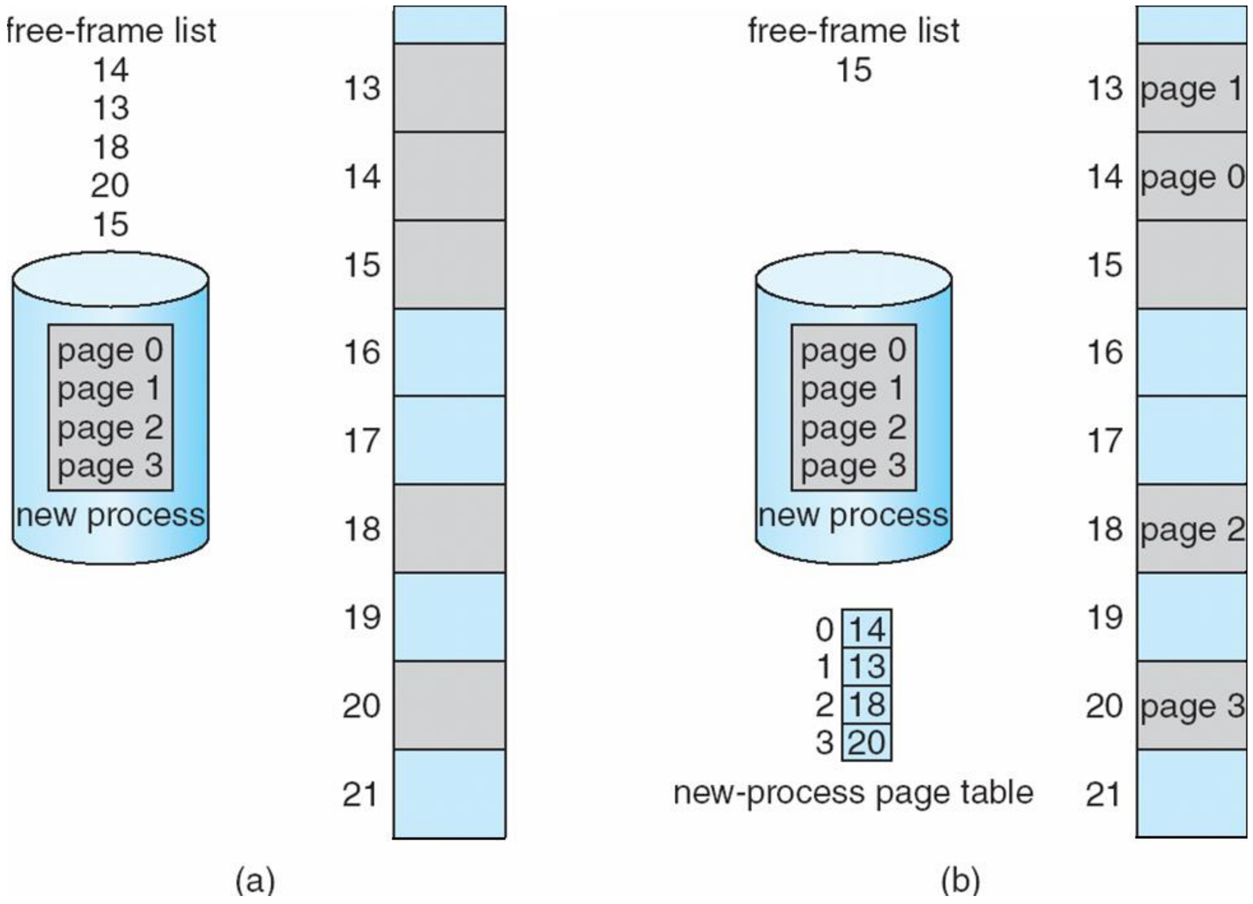


n=2 and *m*=4   32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages (71,680 bytes) + 1,086 bytes
  - Internal fragmentation: 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory layout very different
- By implementation of paging, process can only access its own memory

Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access. Typically, TLB entries for key kernel code are wired down

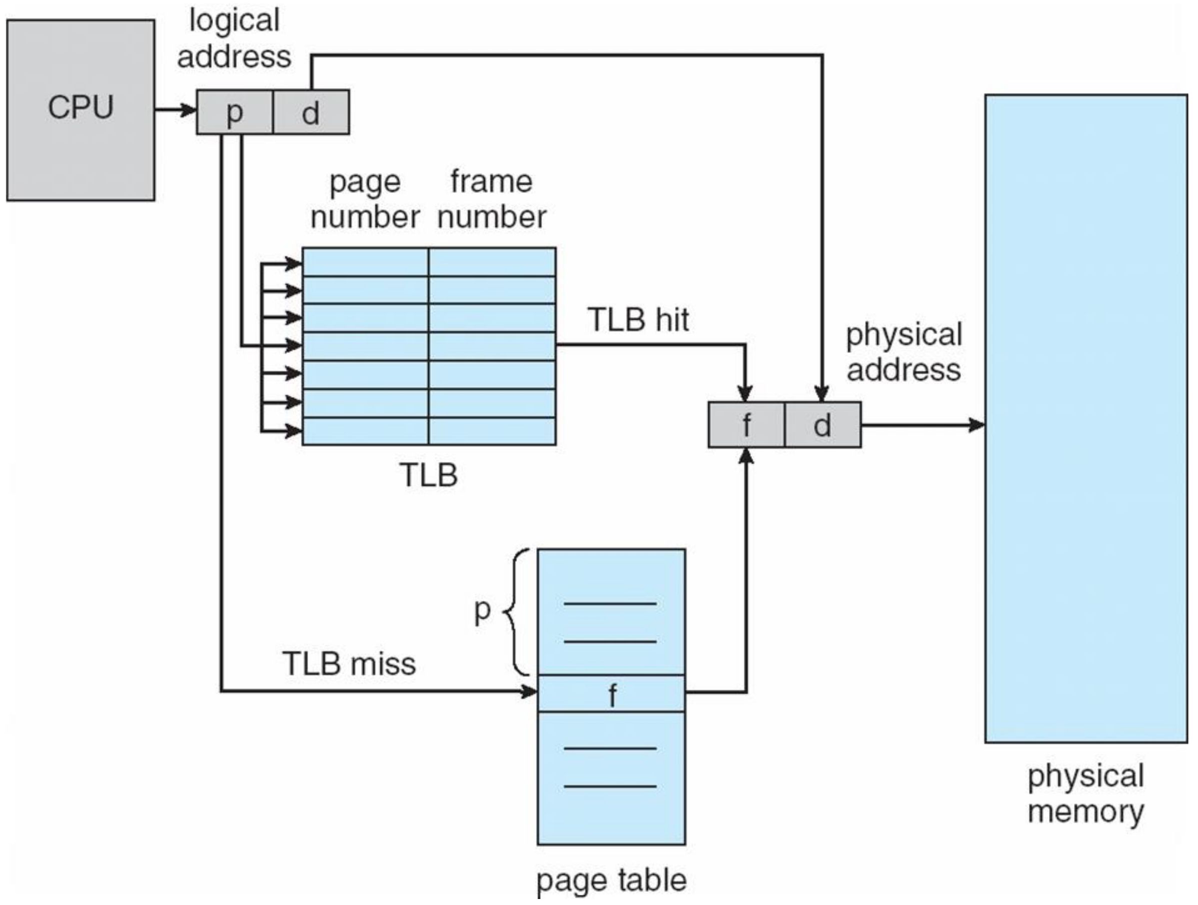
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative memory, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time** (**EAT**)

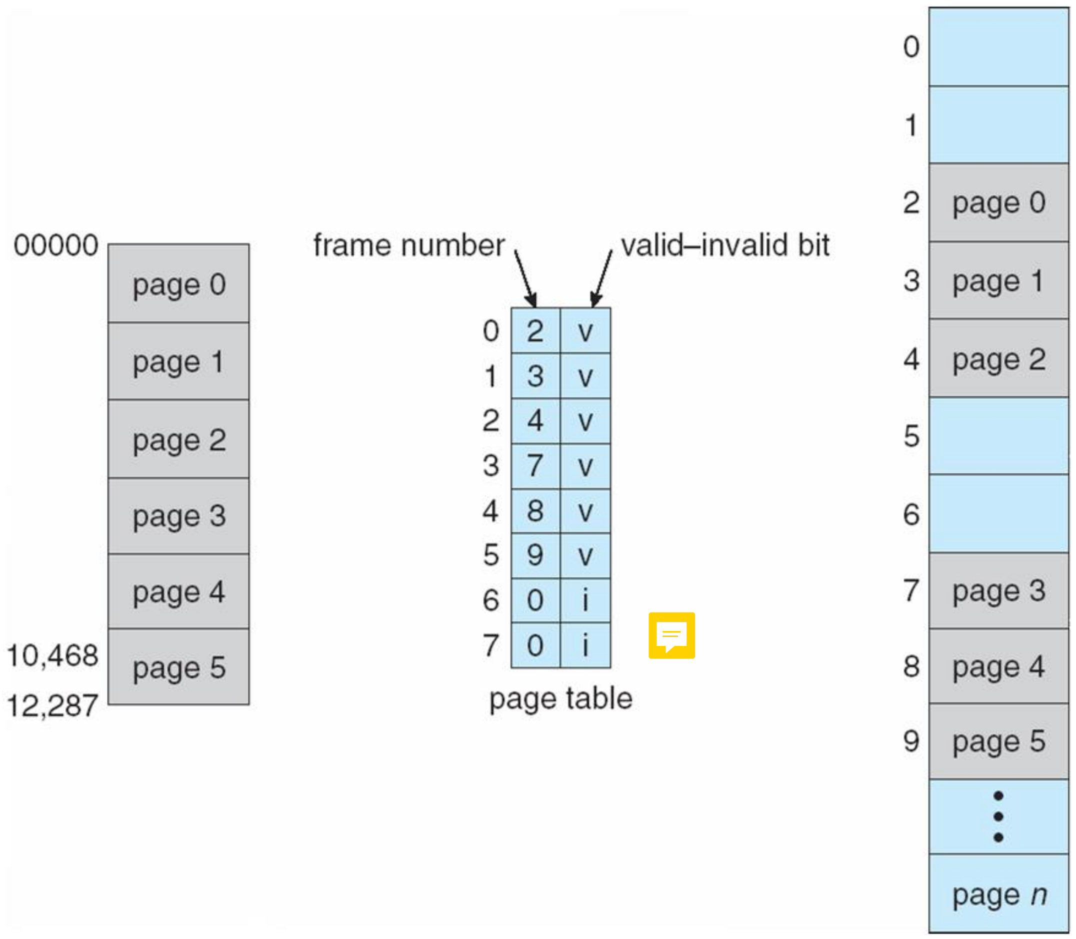
$$\text{EAT} = (m + \varepsilon)\,\alpha + (2m + \varepsilon)(1 - \alpha)$$
$$= (2 - \alpha)m + \varepsilon$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = (100 + 20) x 0.80 + (200 + 20) x 0.20 = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = (100 + 20) x 0.99 + (200 + 20) x 0.01 = 121ns

Memory Protection

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

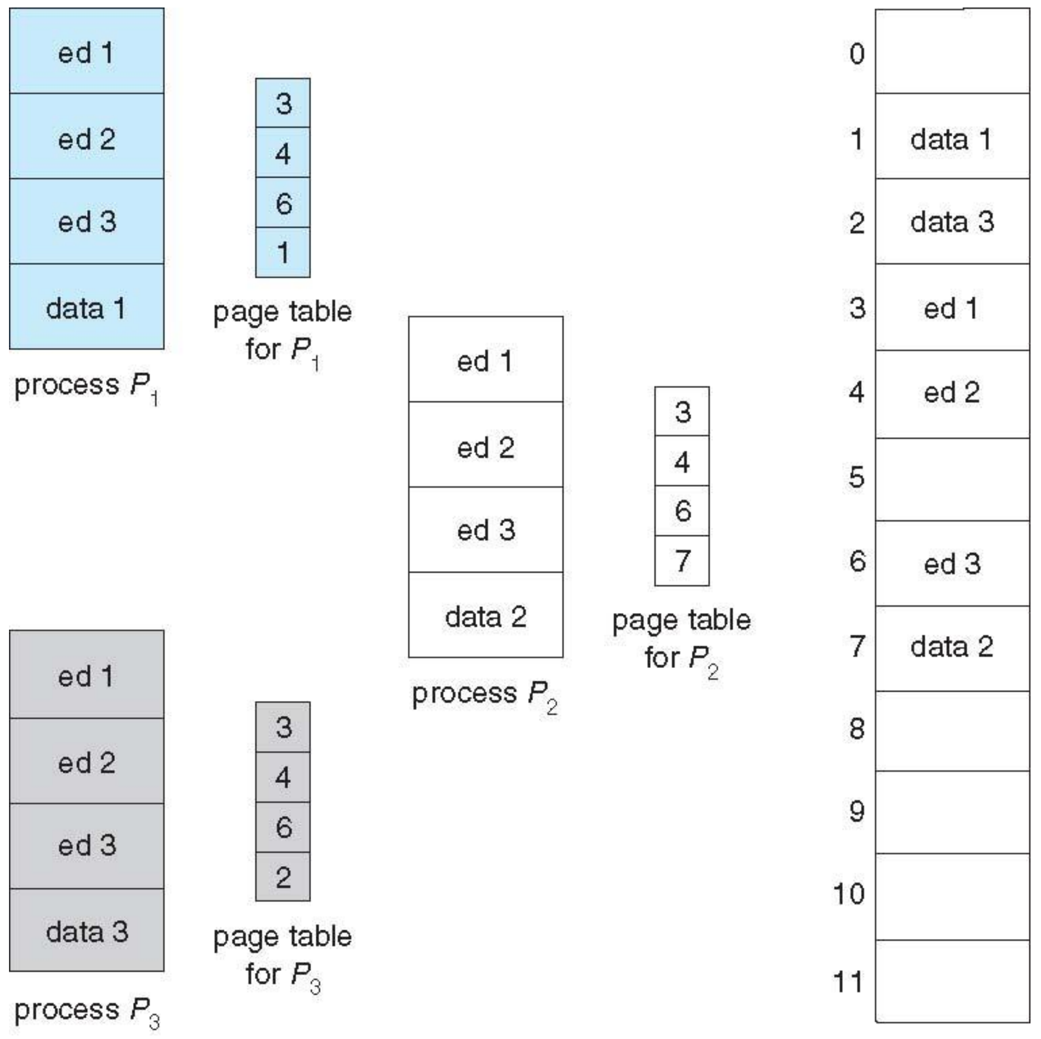
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful as a method 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

Shared Pages Example



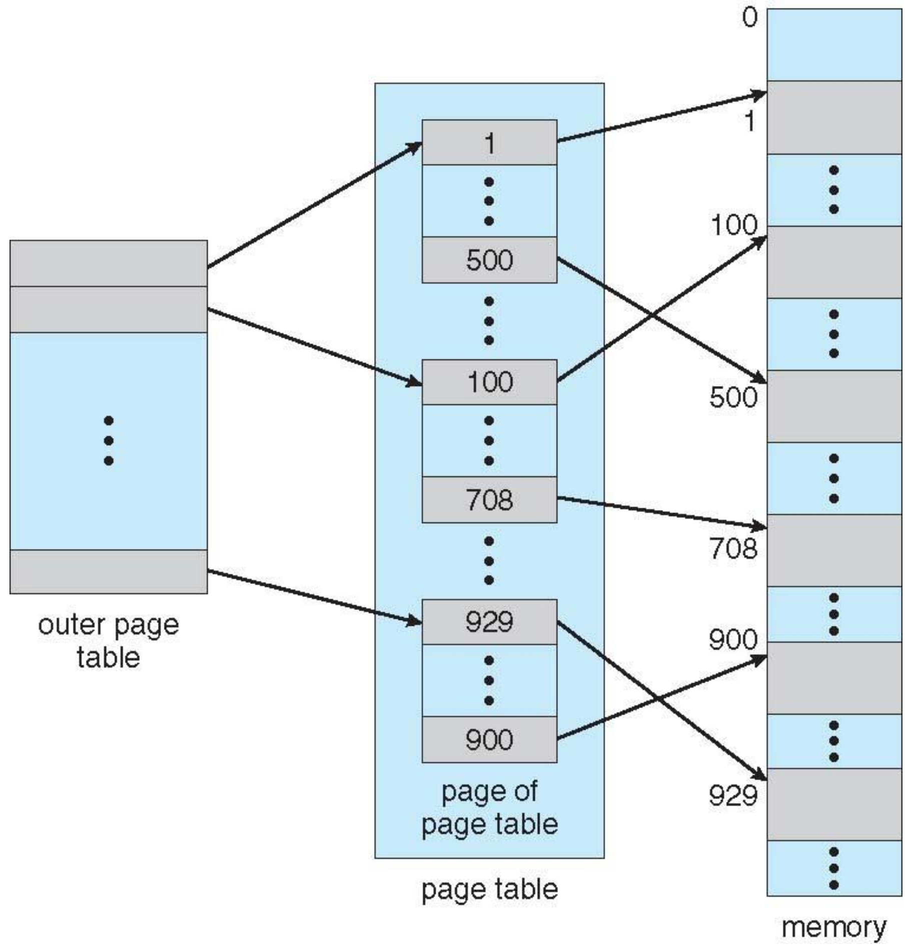
Structure of the Page Table

- 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})
  - Each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory costs a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

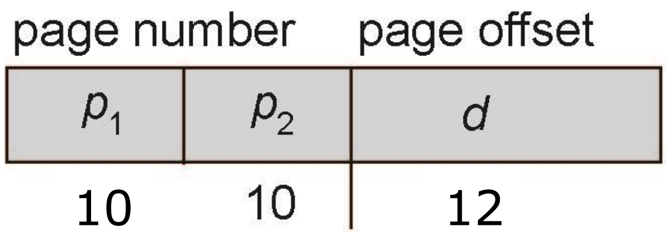
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



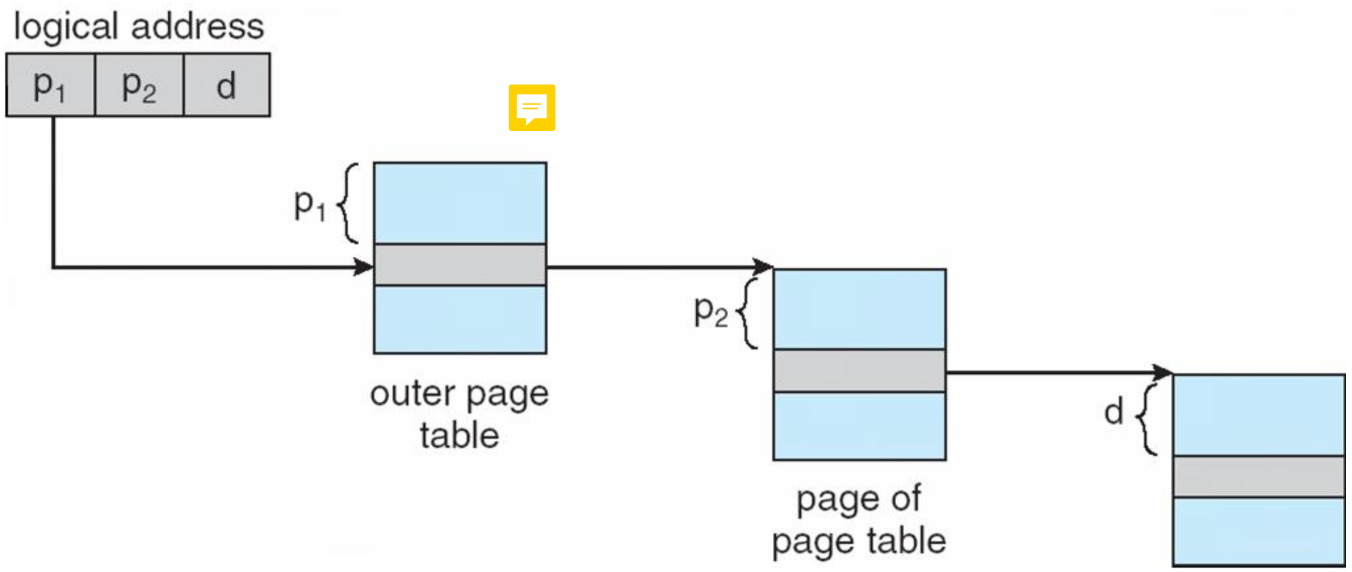
Two-Level Paging Example

- 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 (x 4-byte entry = 4K bytes)
- Thus, a logical address is as follows:



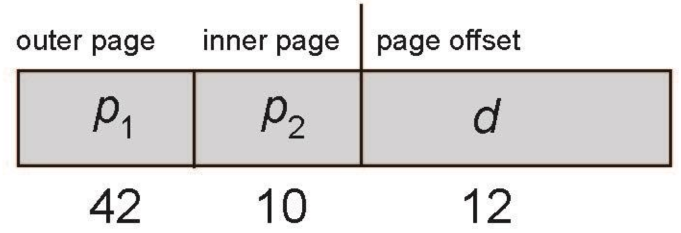
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



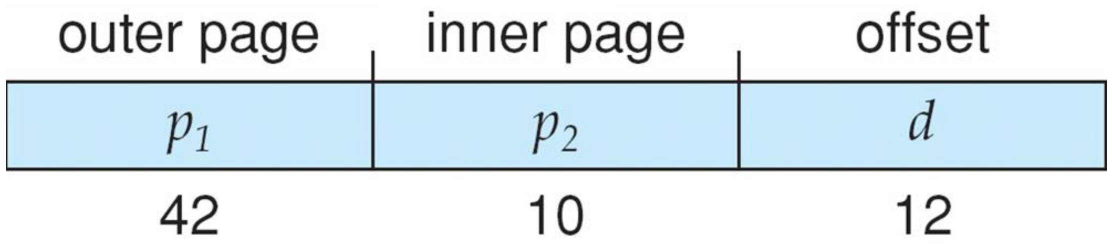
64-bit Logical Address Space

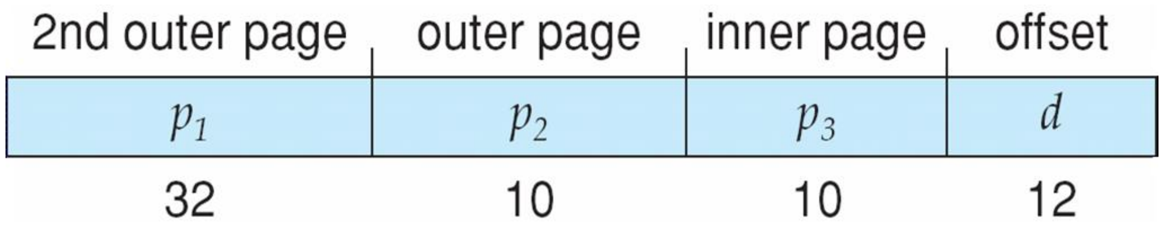
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

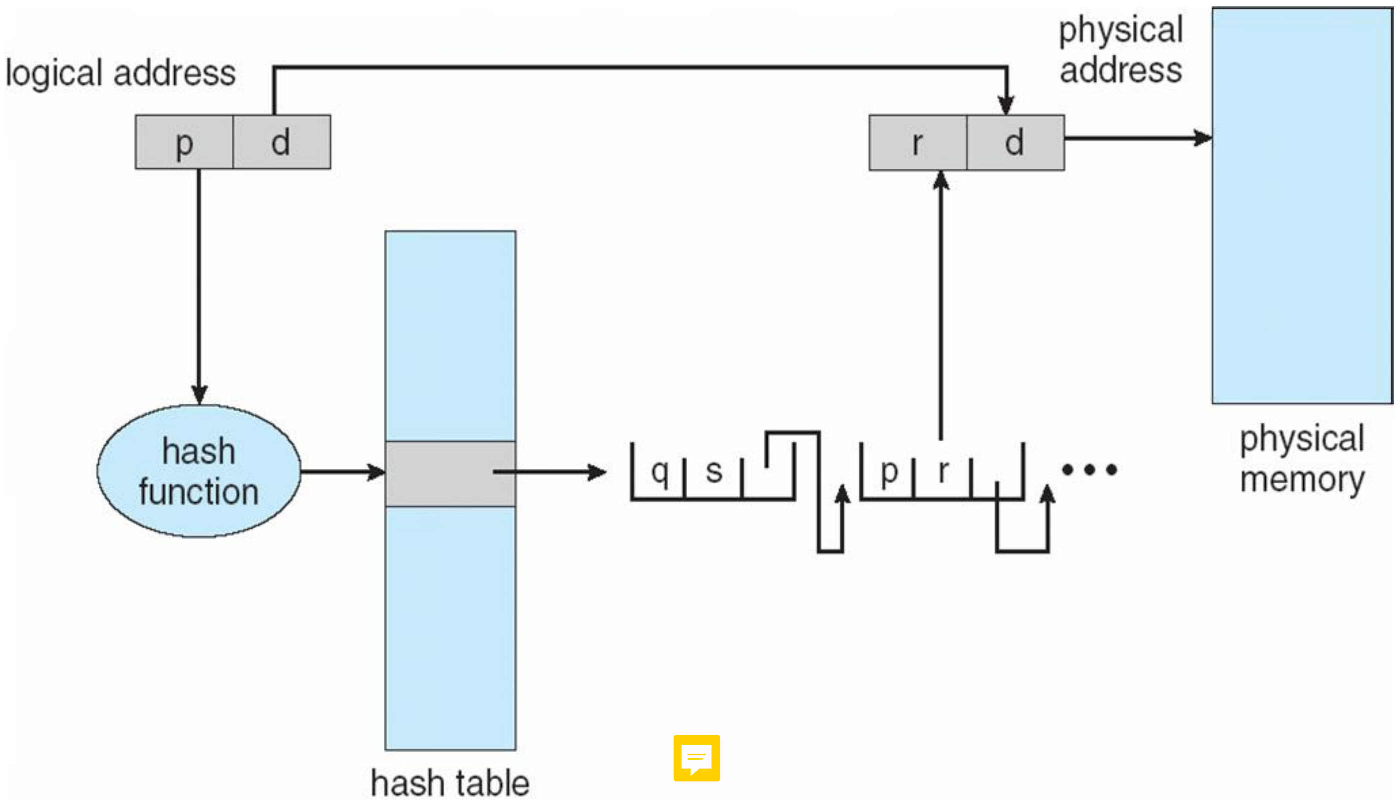




Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

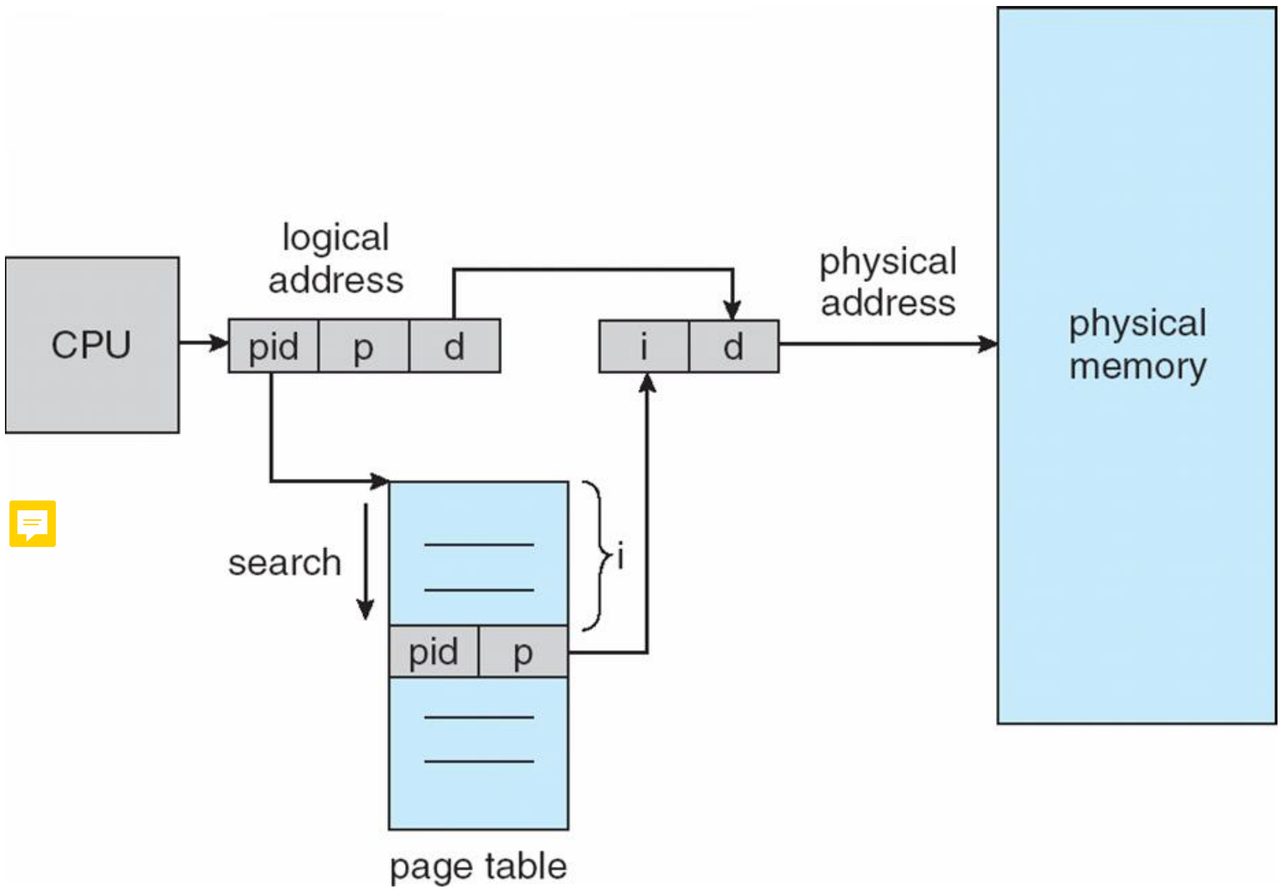
Hashed Page Table



Inverted Page Table

- Rather than each process has a page table and keeps track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Oracle SPARC Solaris

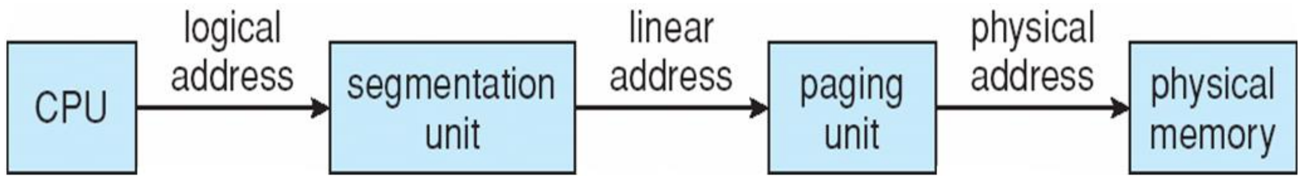
- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
    - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

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

Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
  - Logical to physical address translation in IA-32



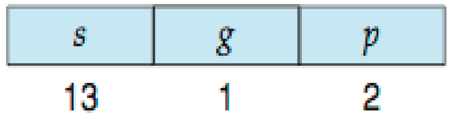
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
    - Divided into two partitions
    - First partition of up to 8 K 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**))

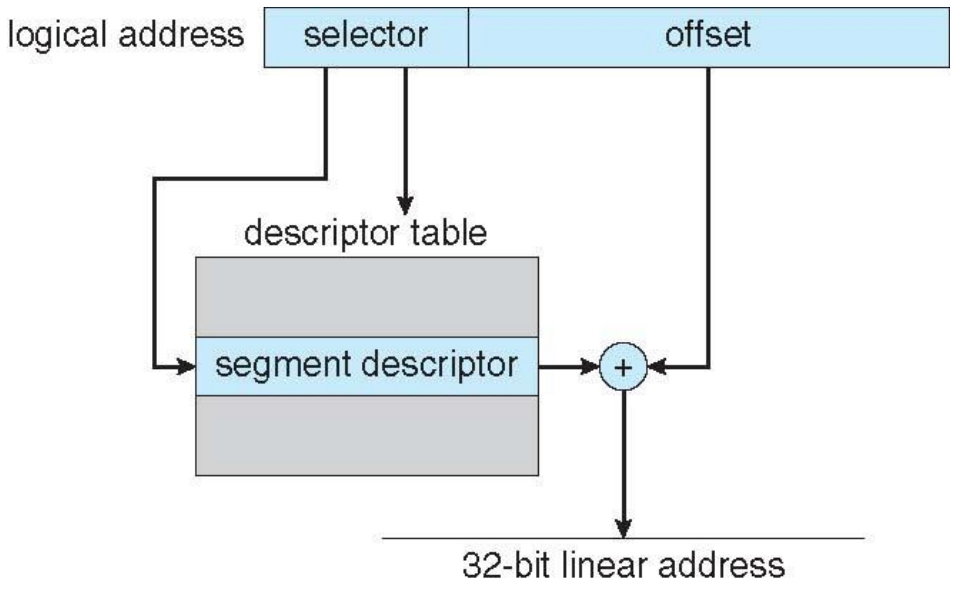
Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address, a pair (selector, offset)
  - 16-bit selector:



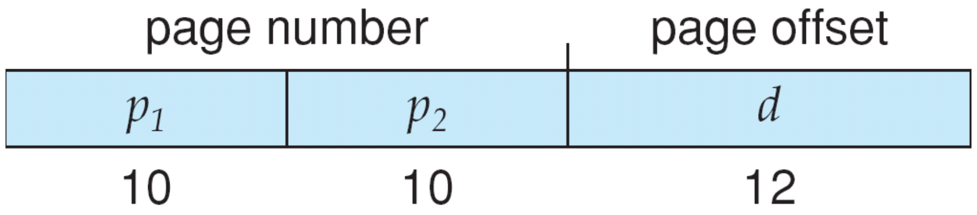
where *s* for segment number, *g* for GDT/LDT, and *p* for protection.

  - Given to segmentation unit, get the segment descriptor (the base and limit information)
    - Which produces 32-bit linear addresses by adding base and offset

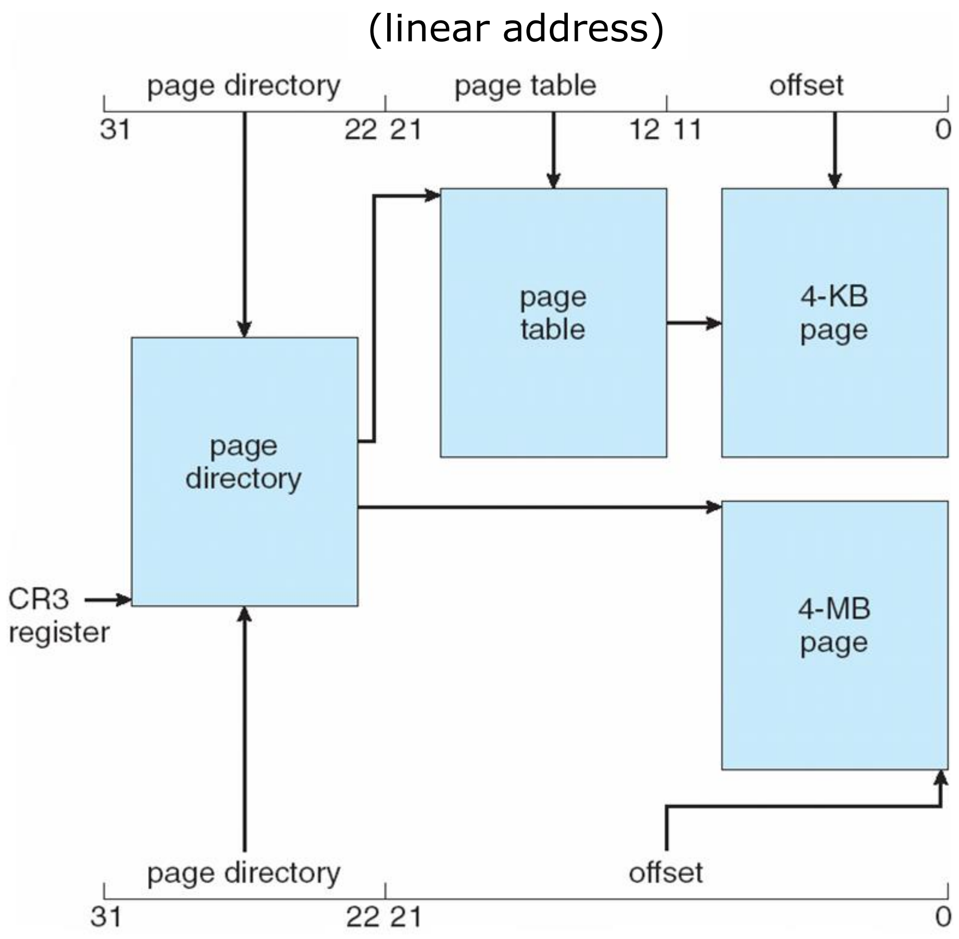


Example: The Intel IA-32 Architecture (Cont.)

- 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
  - A two-level paging scheme
    - The outermost page table (**page directory** in IA-32)

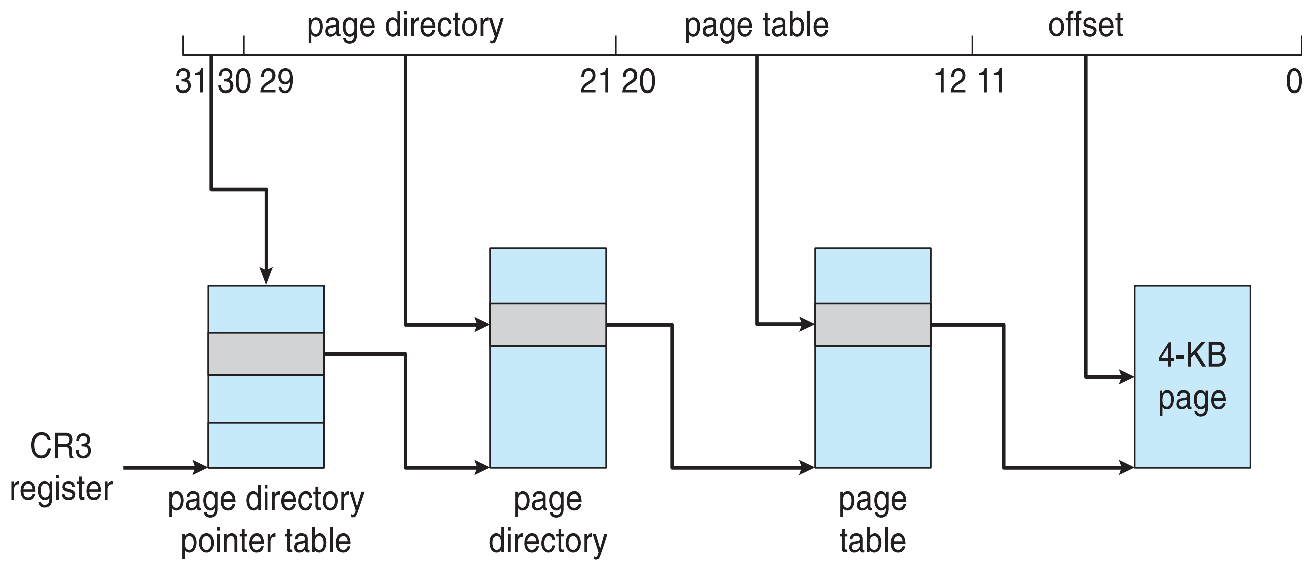


Intel IA-32 Paging Architecture



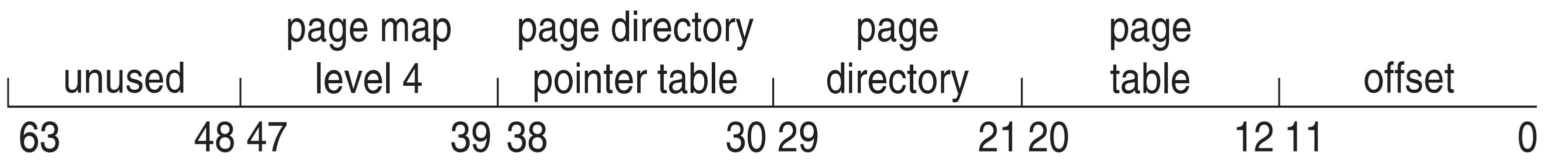
Intel IA-32 Page Address Extensions

- 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 from 32-bits to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



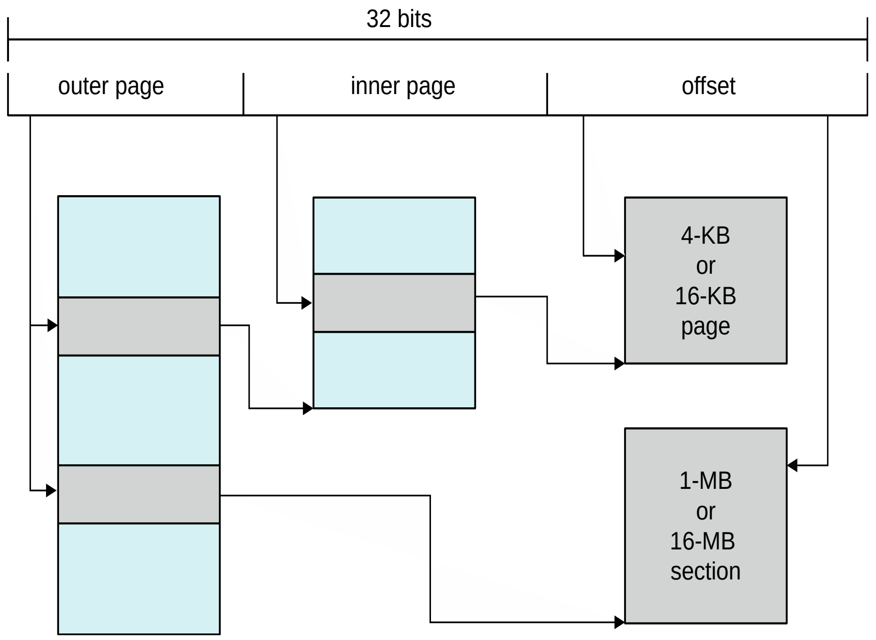
Intel x86-64

- 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



Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two **micro TLBs** (one data, one instruction)
  - Inner is single **main TLB**
  - Translation begins at the micro TLB level. On a miss, the main TLB is checked. On a miss, a page table walk must be performed



End of Chapter 8