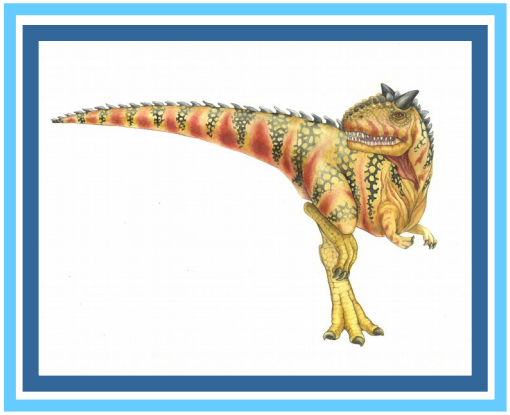
Chapter 9: Main Memory



Chapter 9: 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

- A program must be brought (from disk) into memory and placed within a process for it to be run
- A program can be written in machine language, assembly language, or high-level language.
- Main memory and registers are the only storage entities that a CPU can access directly
- The CPU fetches instructions from main memory according to the value of the program counter.
- Typical instruction execution cycle – fetch instruction from memory, decode the instruction, operand fetch, possible storage of result in memory.

Background (Cont.)

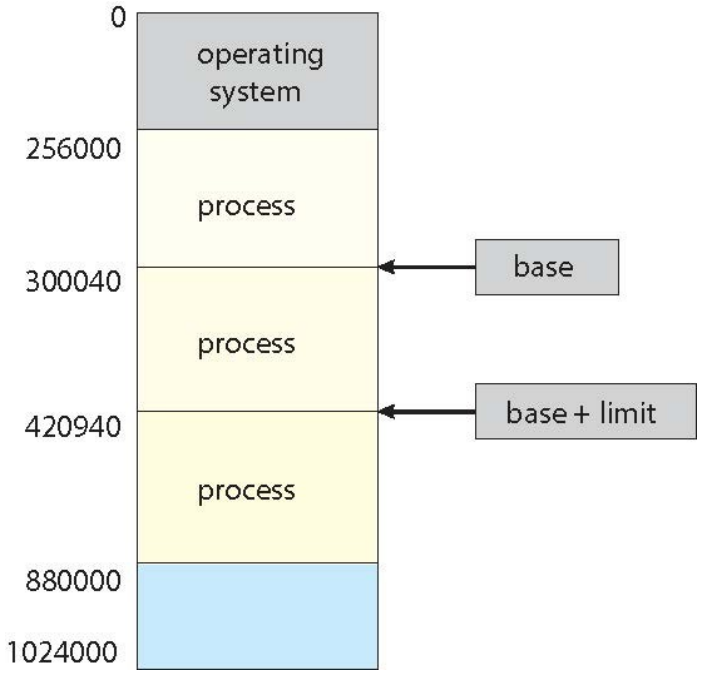
- Memory unit only sees a stream of one of the following:
  - address + read requests (e.g., load memory location 20010 into register number 8)
  - address + data and write requests (e.g., store content of register 6 into memory location 1090)
- Memory unit does not know how these addresses were generated
- Register access can be done in one CPU clock (or less)

Background (Cont.)

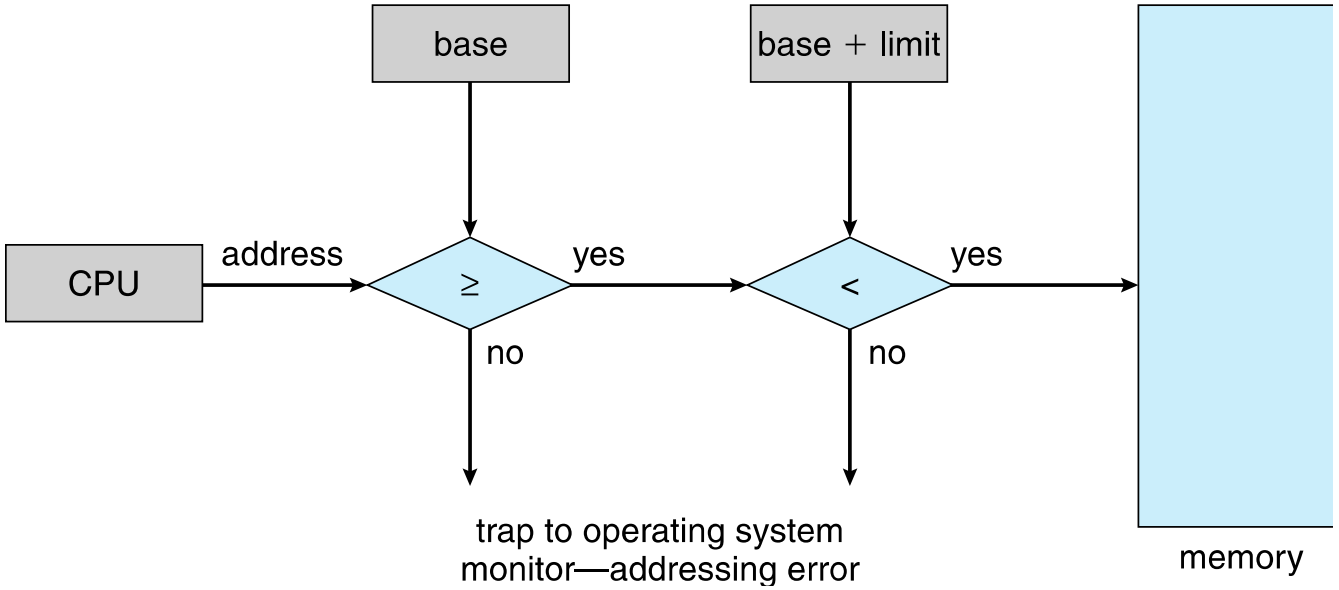
- Completing a memory access may take many cycles of the CPU clock. In such a case the processor needs to **stall** since it does not have the data required to complete the instruction it is executing.
- **Cache** sits between main memory and CPU registers to deal with the "stall" issue.
- Protection of memory is required to ensure correct operation:
  - User process cannot access OS memory
  - One user process cannot access the memory of another user process.

Memory Protection

- A **base register** (holding the smallest legal physical address of a program in memory) and a **limit register** (specifies the size of the program) define the boundary of a program in memory.
- CPU must check that every memory access generated in user mode is between the base and base+limit for that user



Hardware Address Protection



Address Binding

- A program residing on the disk needs to be brought into memory in order to execute. Such a program is usually stored as a binary executable file and is kept in an **input queue.**
- In general, we do not know a priori where the program is going to reside in memory. Therefore, it is convenient to assume that the first physical address of a program always starts at location 0000.
- Without some hardware or software support, program must be loaded into address 0000
- It is impractical to have first physical address of user process to always start at location 0000.
- Most (all) computer systems provide hardware and/or software support for memory management,

Address Binding (Cont.)

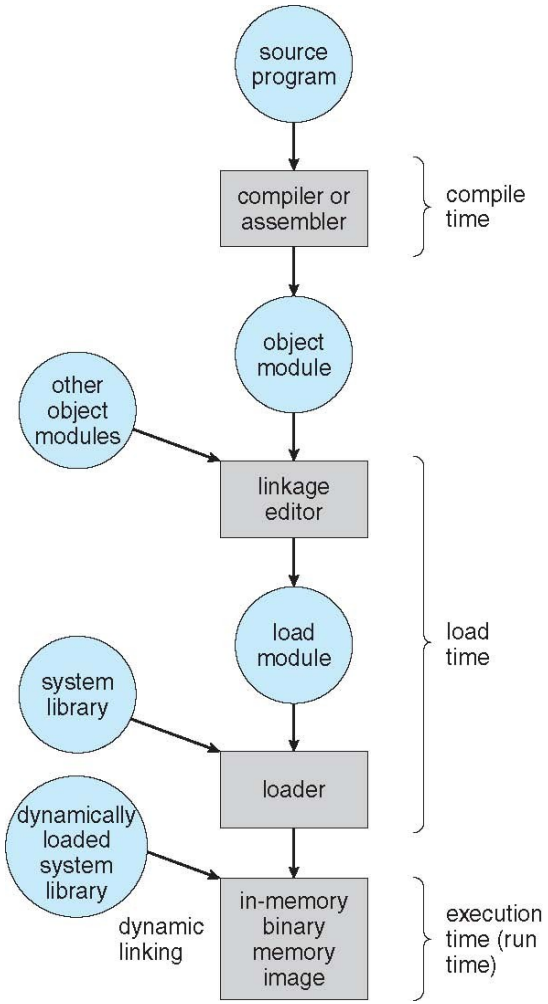
- In general, addresses are represented in different ways at different stages of a program's life
  - Addresses in the source program are generally symbolic
    - i.e., variable "count"
  - A compiler typically **bind**s these symbolic addresses to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute (physical) addresses
    - i.e., 74014
  - Each binding maps one address space to another address space

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different points in time:

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.
- **Load time**: If memory location is not known at compile time and no hardware support is available, **relocatable code** must be generated (in software).
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
  - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



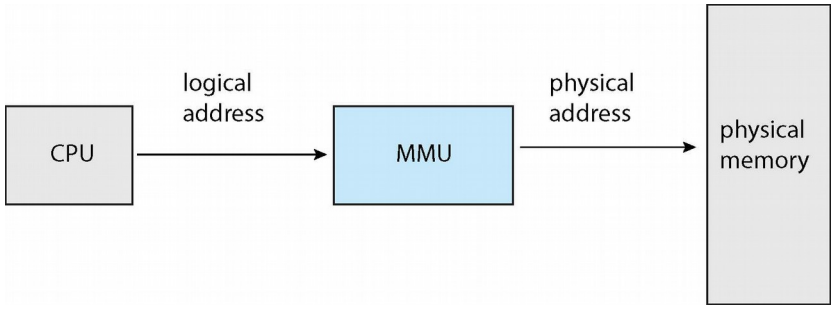
Logical vs. Physical Address Space

- The concept of a **logical address space** that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU.
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are:
  - The same in compile-time and load-time address-binding schemes;
  - They differ in execution-time address-binding scheme. In that case the logical address is referred to as **virtual address.**

  We use Logical address and virtual address interchangeably
- **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 a given logical address space.

Memory-Management Unit (MMU)

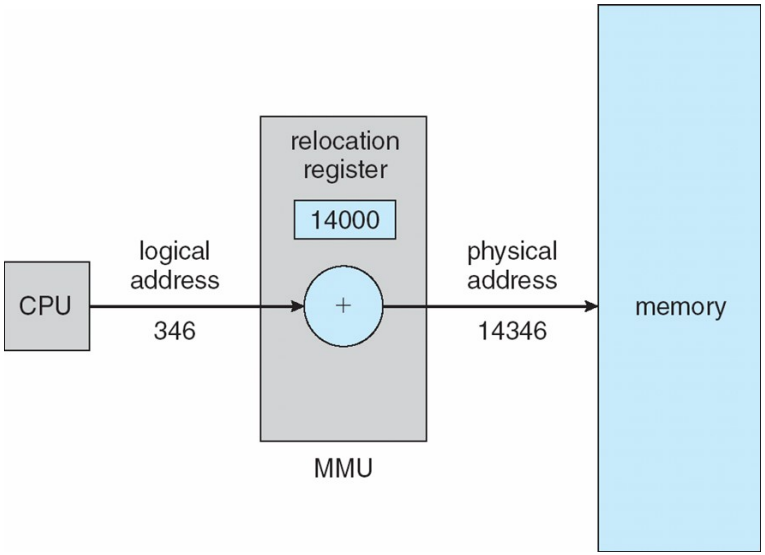
- Hardware device that at run time maps virtual addresses to physical address



- Many methods possible, covered in the rest of this chapter
- 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

Dynamic relocation using a relocation register

- To start, consider simple scheme where the value in the base 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



Dynamic Loading

- Until now we assumed that the entire program and data has to be in main memory to execute
- **Dynamic loading** allows a routine (module) to be loaded into memory only when it is called (used)
- Results in better memory-space utilization; an 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 (e.g., exception handling)
- No special support from the operating system is required
  - It is the responsibility of the users to design their programs to take advantage of such a method
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Dynamically linked libraries** – system libraries that are linked to user programs when the programs are run.
  - Similar to dynamic loading. But, linking rather than loading is postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

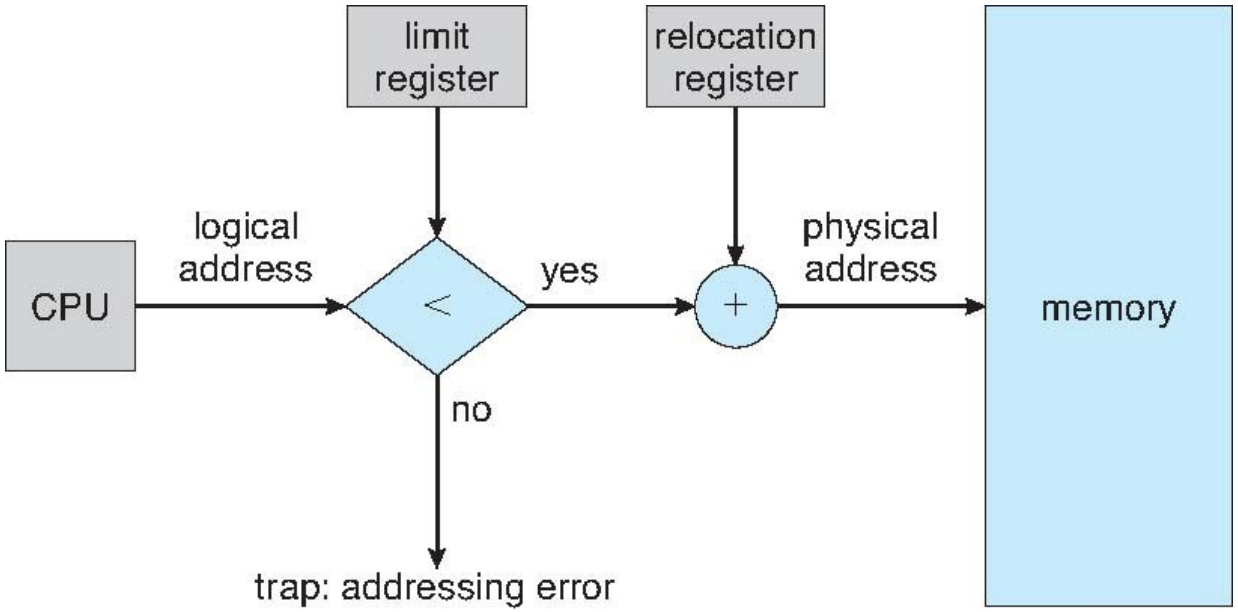
Contiguous Allocation

- 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 are held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

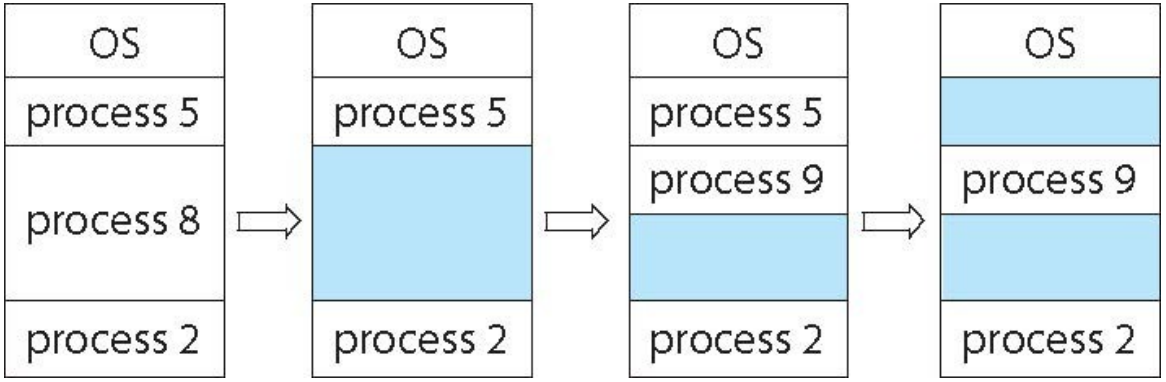
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** – comes and goes as needed**.** Thus, kernel can change size dynamically.

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

- **Variable-partition** -- sized to a given process' needs.
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (holes)



Dynamic Storage-Allocation Problem

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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous and therefore cannot be used.
  - First fit analysis reveals that given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation
    - 1/3 of memory may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory.
  - Can happen if there is hole of size 15,000 bytes and a process needs 14,900 bytes; Keeping a hole of size 100 bytes is not worth the effort so the process is allocated 15,000 bytes.
  - The size difference of 100 bytes is memory internal to a partition, but not being used

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 -- cannot perform compaction while I/O is in progress involving memory that is being compacted.
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

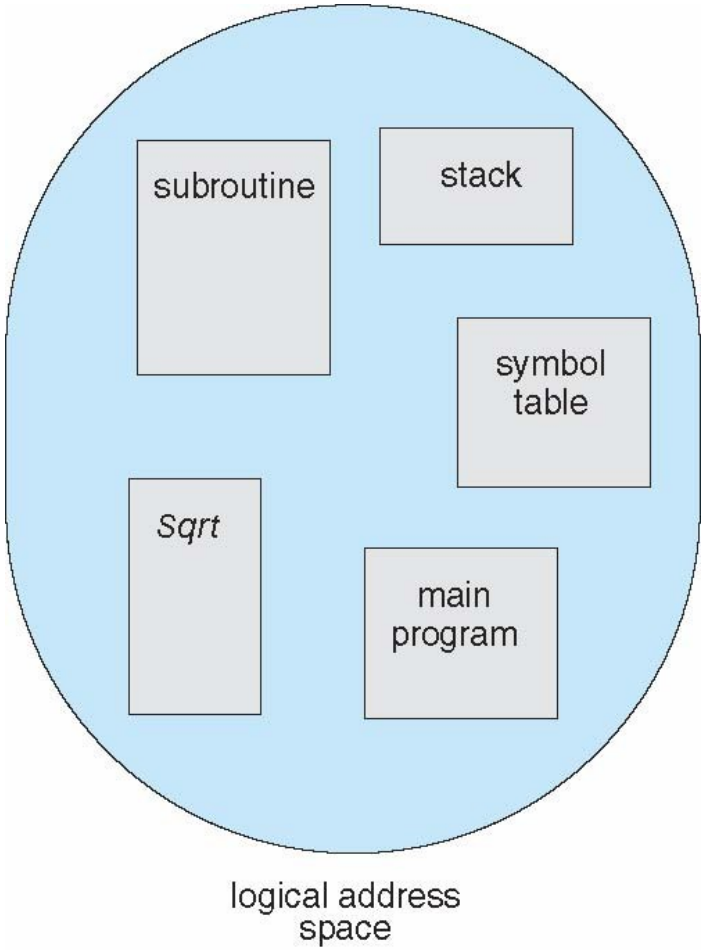
Non-contiguous Allocation

- Partition the a program into a number of small units, each of which can reside in a different part of the memory.
- Need hardware support.
- Various methods to do the partitions:
  - Segmentation.
  - Paging
  - paged segmentation.

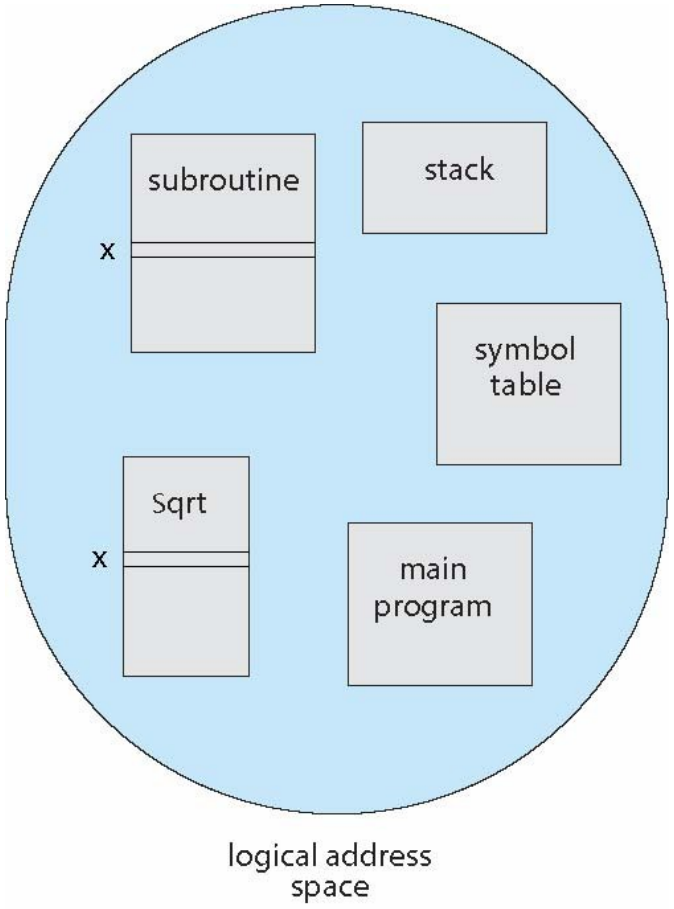
Segmentation

- Memory-management scheme that supports user's view of memory
- A program is a collection of segments -- a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays
- Each segment can to reside in different parts of memory. Way to circumvent the contiguous allocation requirement.

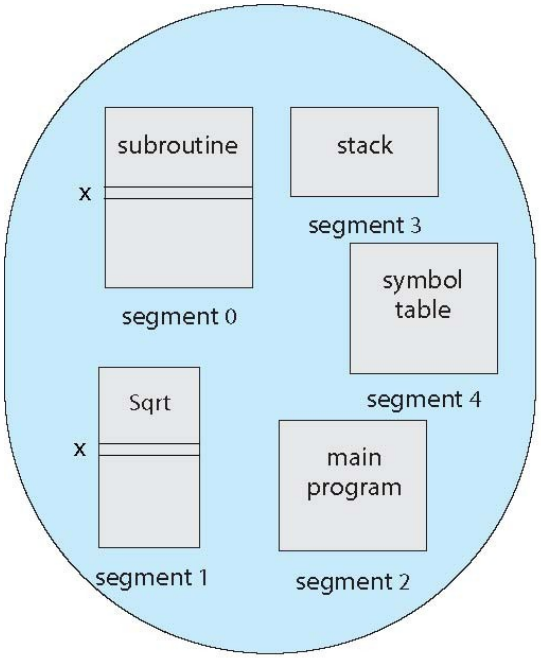
User's View of a Program

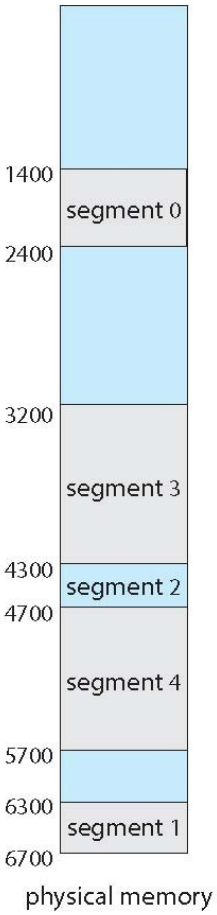


Two Dimensional Addresses



Logical and Physical Memory





Segmentation Architecture

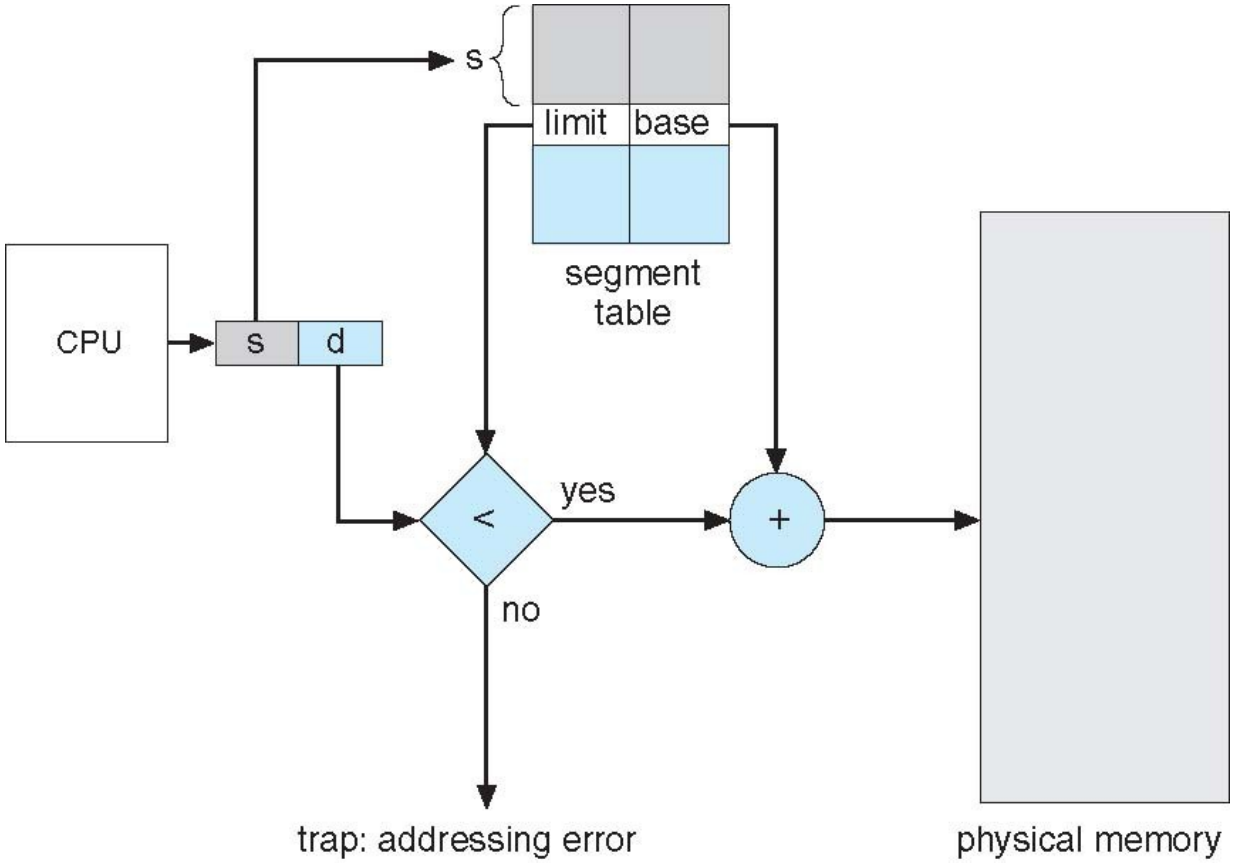
- Logical address consists of a two tuple:

  <segment-number, offset>

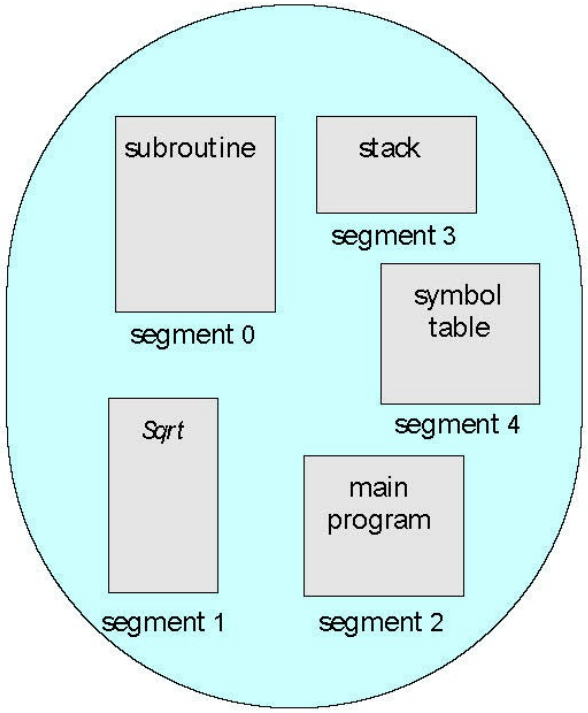
- Need to map a two-dimensional logical addresses to a one-dimensional physical address. Done via **Segment table**:
  - **base** – contains the starting physical address where a segments reside in memory
  - **limit** – specifies the length of the segment
- Segment table is kept in memory
  - **Segment-table 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 Hardware

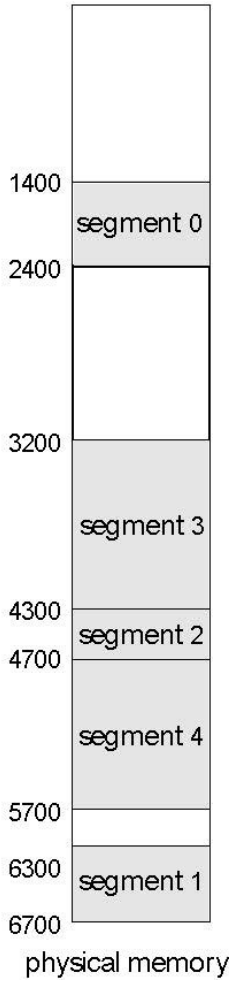


Example of Segmentation



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

segment table



Paging

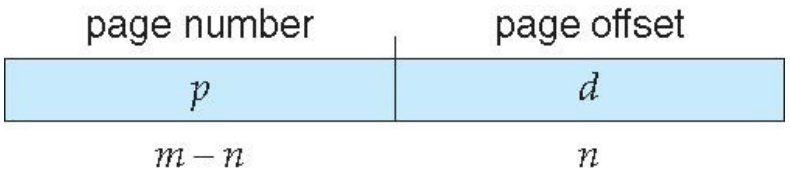
- Physical address space of a process can be non-contiguous.
- Process is divided into fixed-size blocks, each of which may reside in a different part of physical memory.
- Divide physical memory into fixed-sized blocks called **frames**
  - Size of a frame is power of 2 between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size as frames called **pages**
- Backing store (dedicated disk), where the program is permanently residing, is also split into storage units (called **blocks**), which are the same size as the frame and pages.
- Physical memory allocated whenever the latter is available
  - Avoids external fragmentation
  - Still have Internal fragmentation

Paging (Cont.)

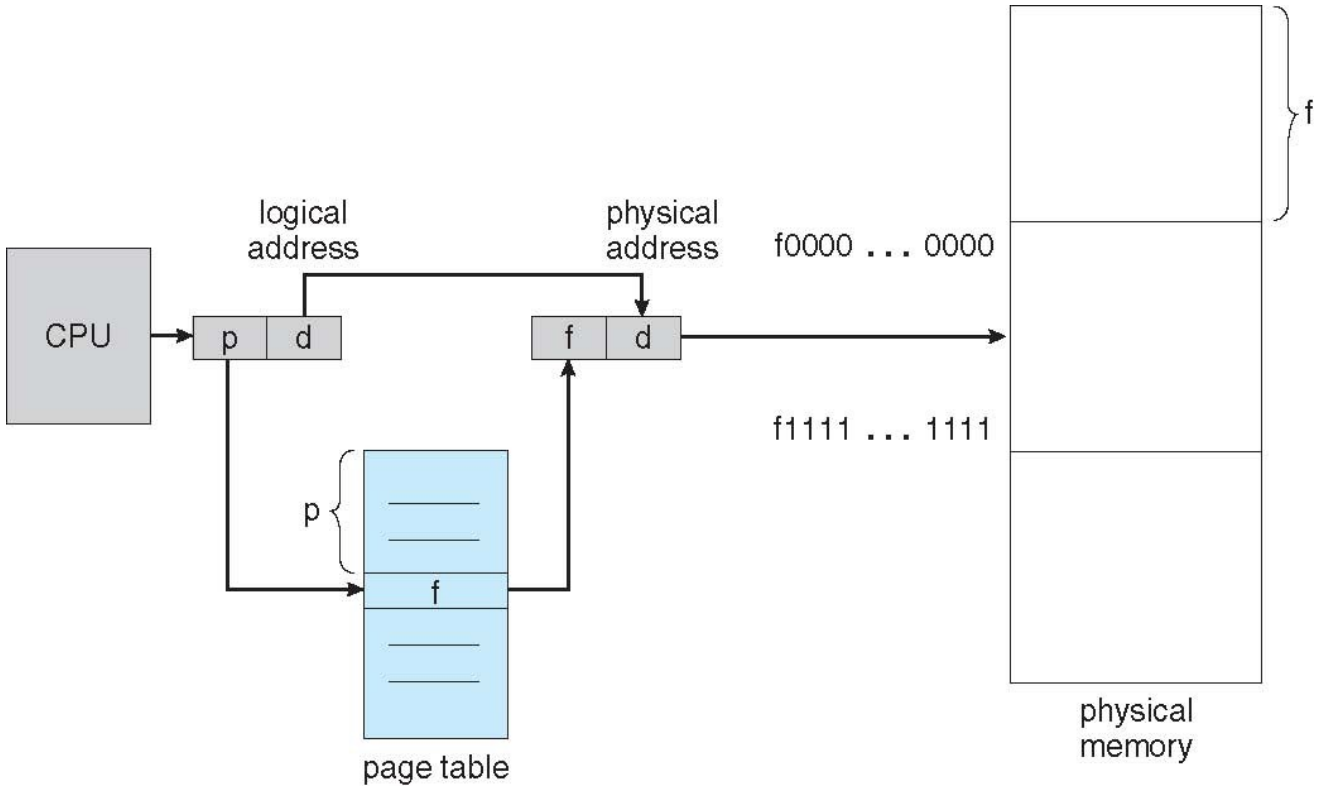
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program from backing store.
- Set up a **page table** to translate logical to physical addresses
- Page table is kept in memory.
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Still have Internal fragmentation (more later)

Address Translation Scheme

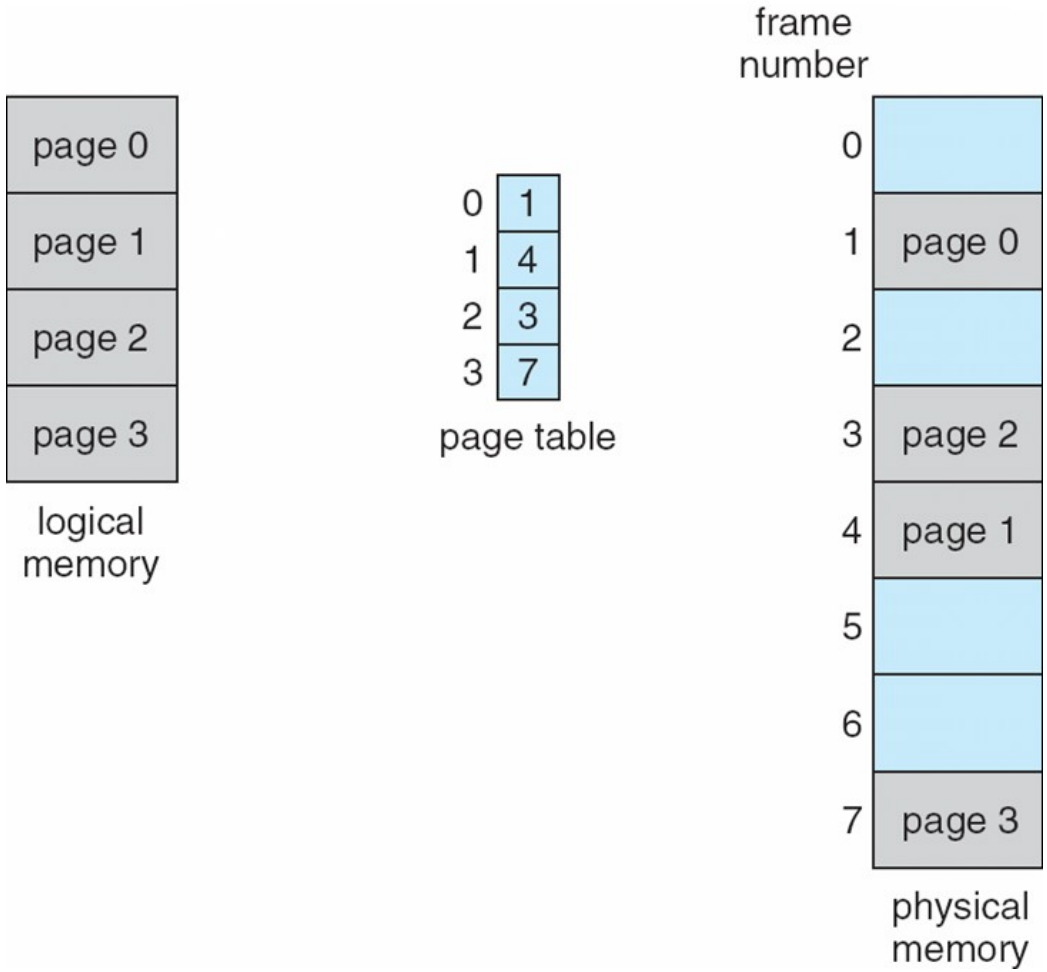
- Assume the logical address space is 2^m. (How is *m* determined?)
- Assume page size is 2^n
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory. Size of ***p*** is "m – n"
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit. Size of ***d*** is "n".



Paging Hardware

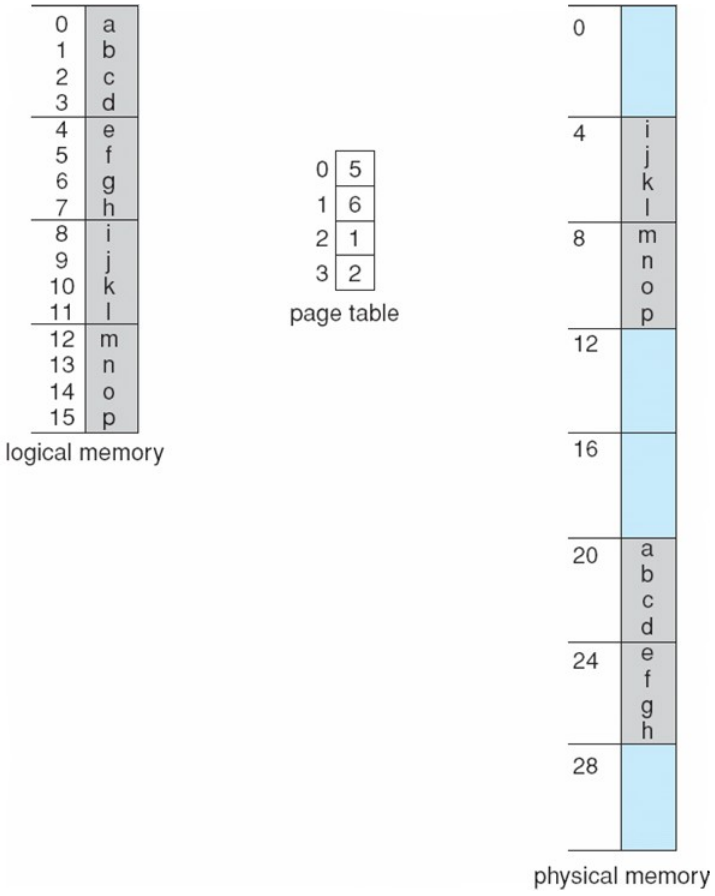


Paging Model of Logical and Physical Memory



Paging Example

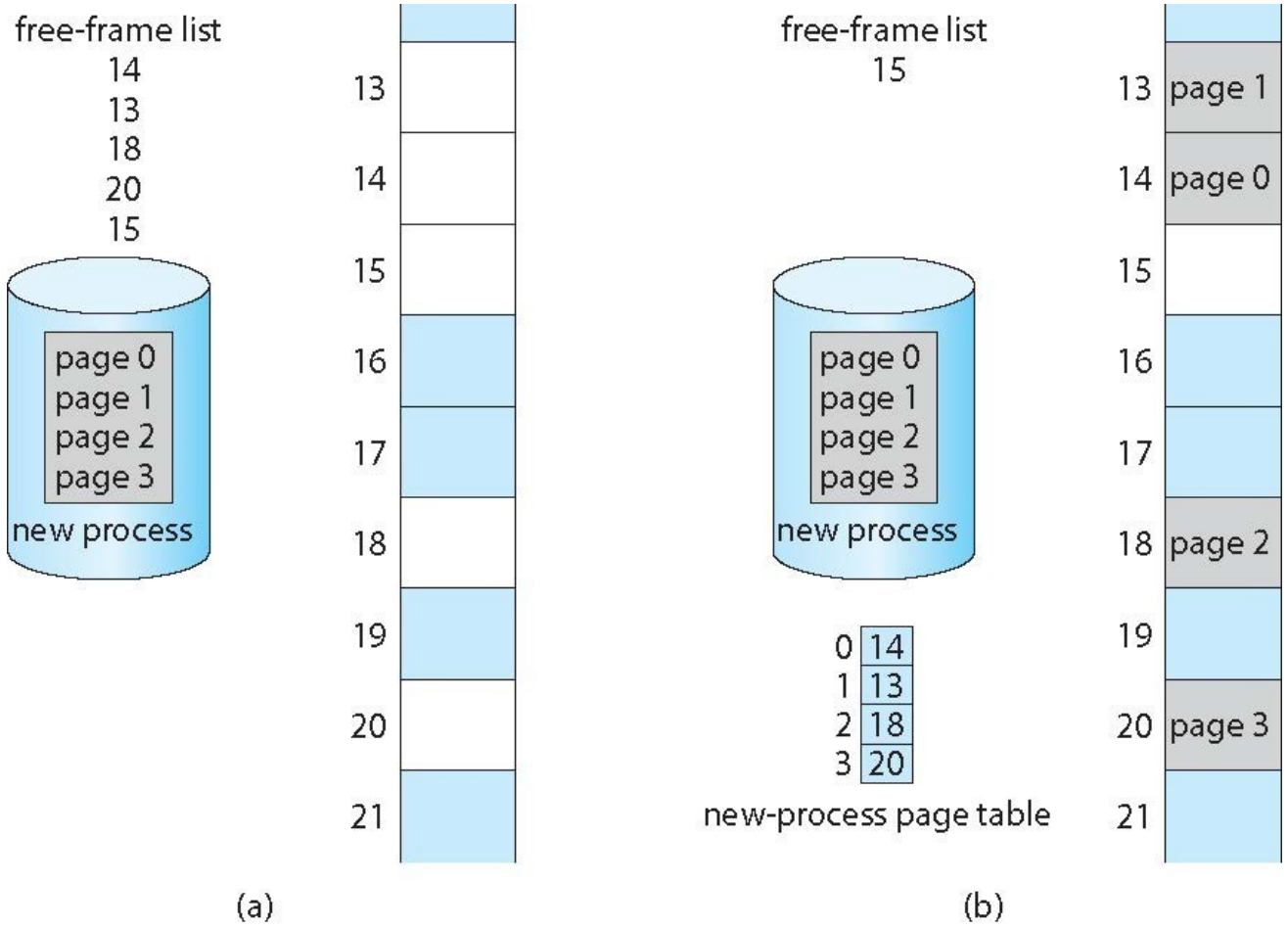
Assume m = 4 and n = 2 and 32-byte memory and 4-byte pages



Internal Fragmentation

- 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 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
- By implementation process can only access its own memory

Allocating Frames to a New Process



Before allocation

After allocation

TLB -- Associative Memory

- If page table is kept in main memory 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 partially solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)
- Associative memory – parallel search

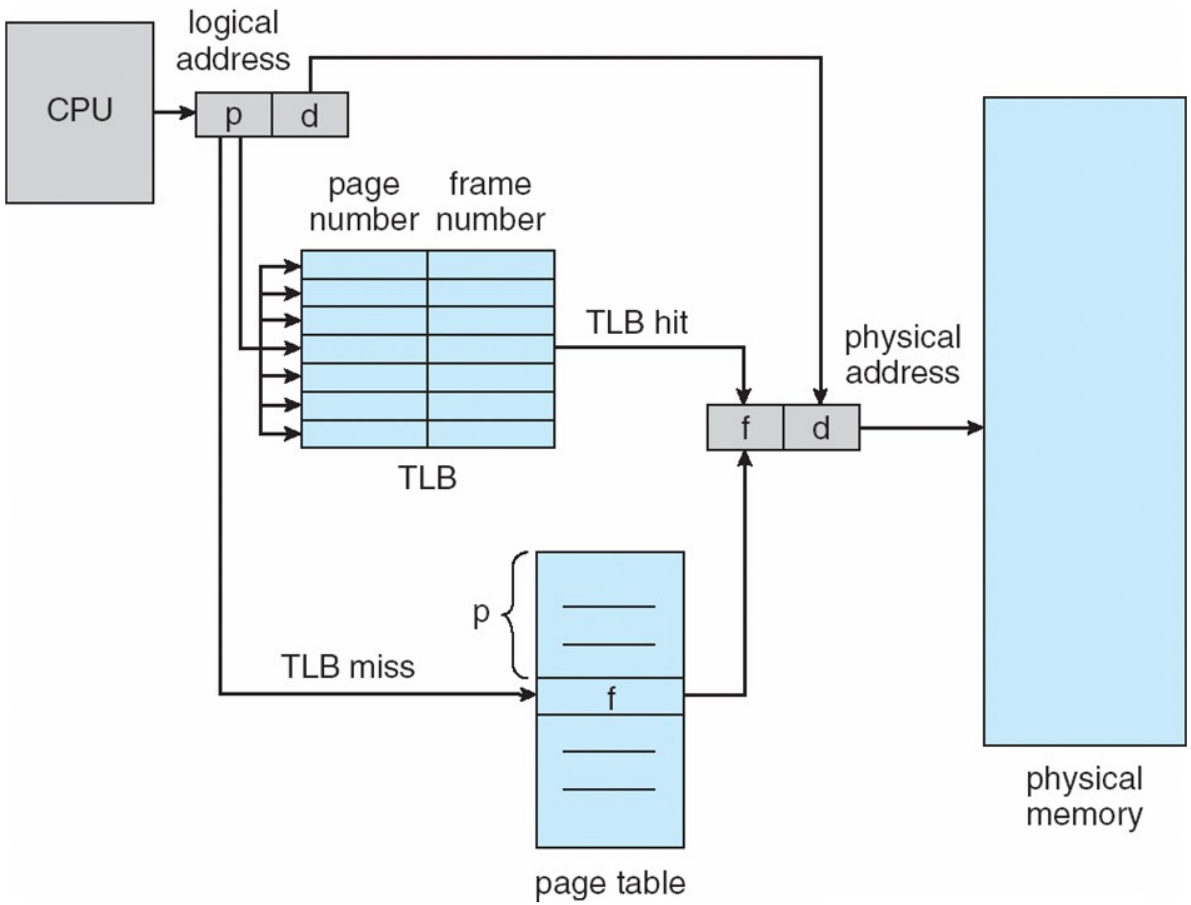
Page #	Frame #

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

TLB issues

- TLB is typically small (64 to 1,024 entries)
- On a TLB miss, the value of the (missed page-table and frame-number), is loaded into the TLB for faster access next time that address is used.
  - What if there is no free TLB entry? Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access
- 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 TLB at every context switch

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**)

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

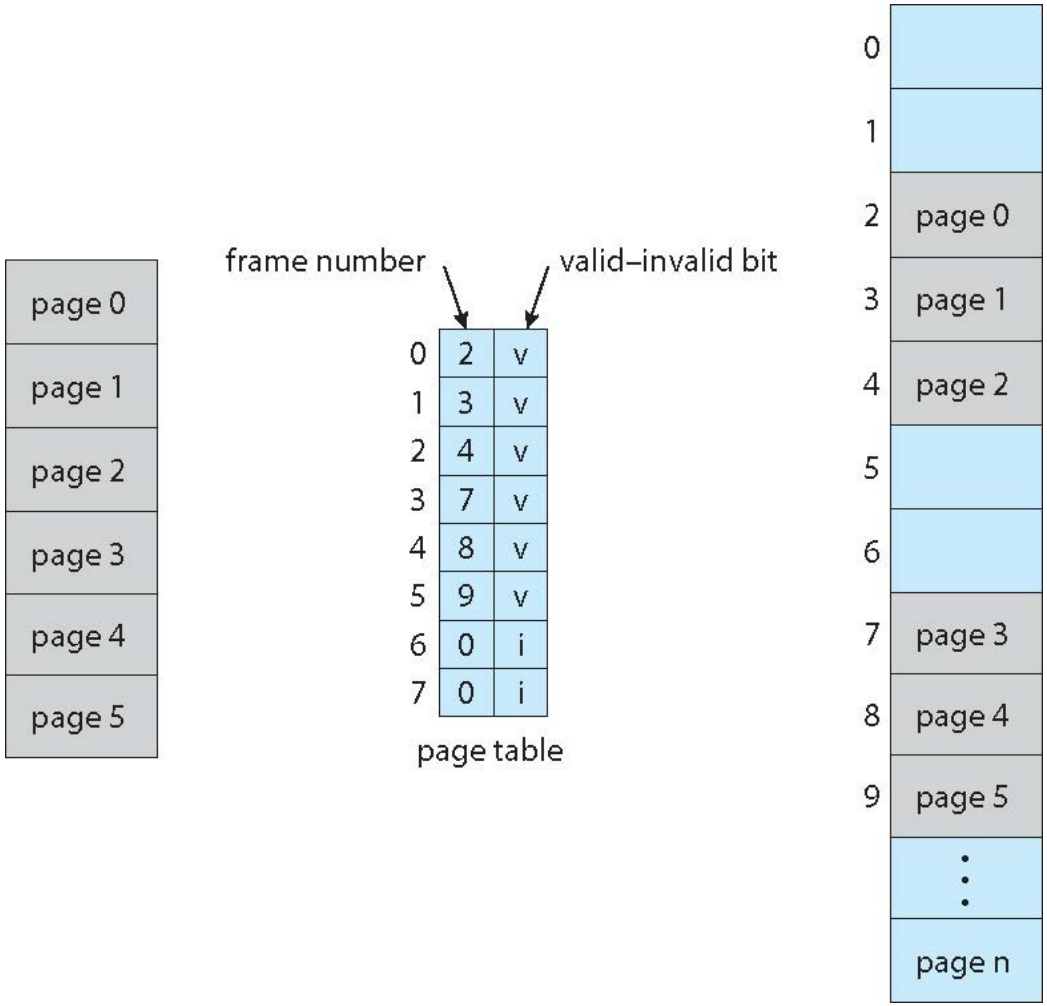
$$= 2 + \varepsilon - \alpha$$

- Consider ε = 20ns for TLB search and 100ns for memory access
  - if α = 80%:
    - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
  - Consider more realistic hit ratio of α = 99%
    - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

- Memory protection implemented by associating protection bits with each frame to indicate if “read-only “ or “read-write” access is allowed
  - Can also add more bits to indicate “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 is a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

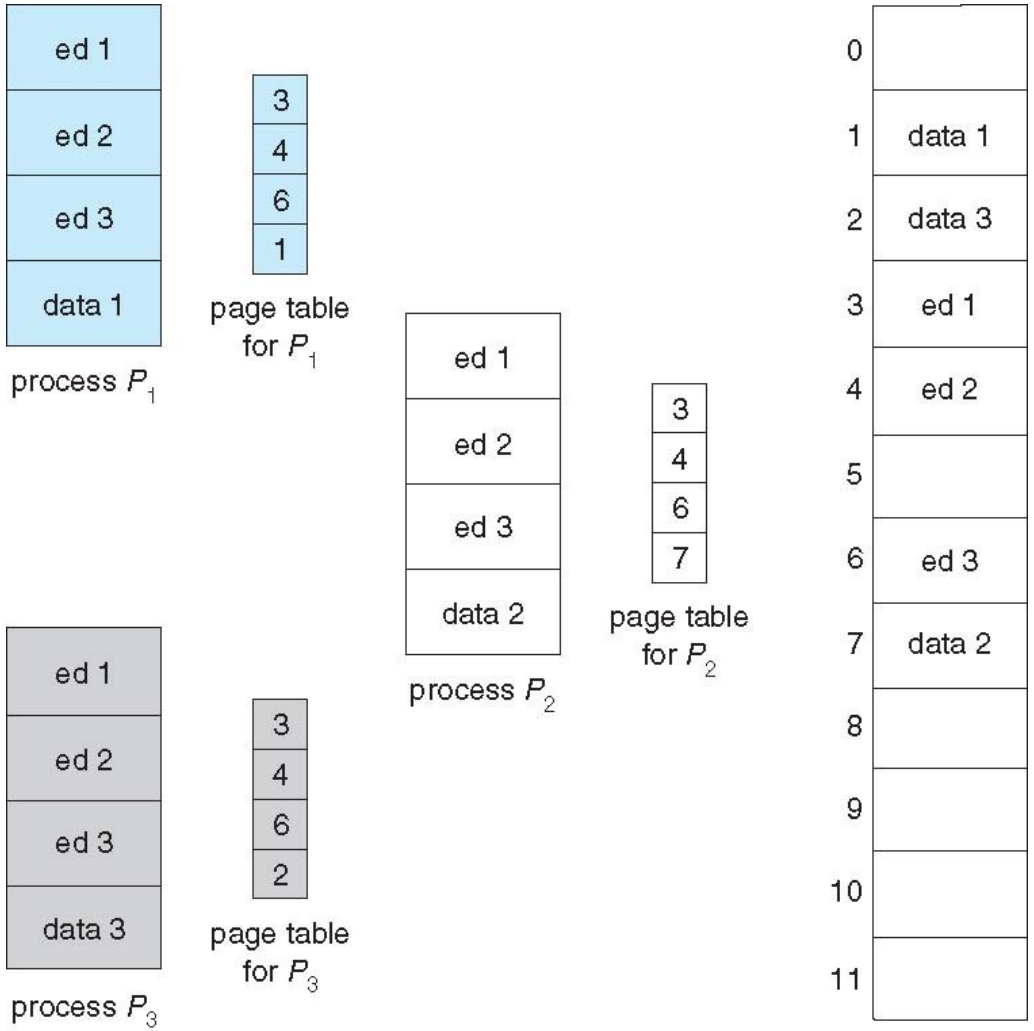
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

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

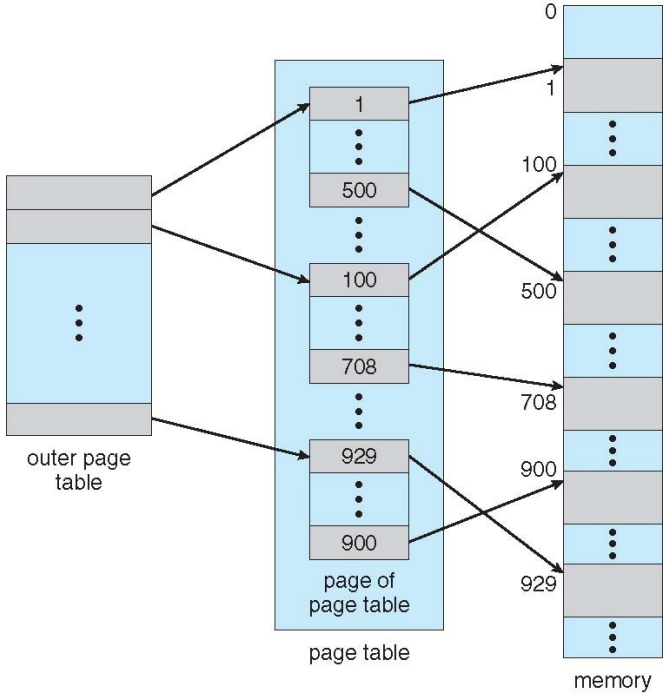
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space
  - Page size of 1 KB (2^{10})
  - Page table would have 4 million entries ($2^{32} / 2^{10}$)
  - If each entry is 4 bytes -> Page table is of size 16 MB
    - That amount of memory used to cost a lot.
    - Do not want to allocate that contiguously in main memory
- What about a 64-bit logical address space?

Page Table for Large address space

- 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



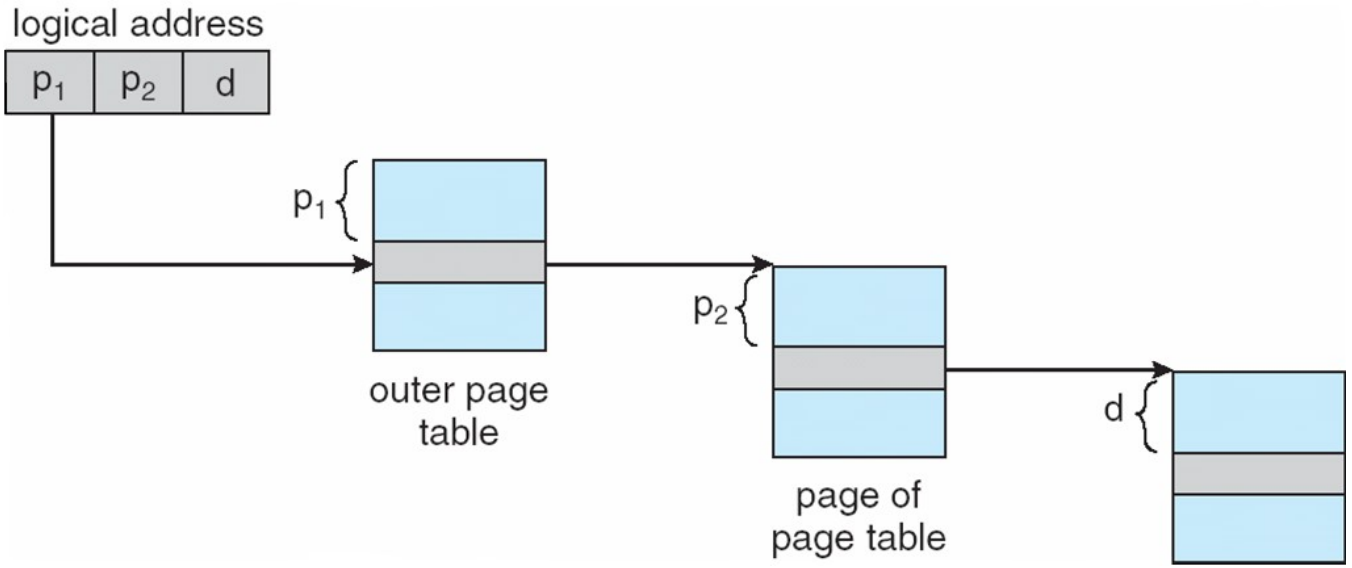
Two-Level Paging Example

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

page number		page offset
p_1	p_2	d
10	10	12

- where p_1 is an index into the outer page table, and p_2 is the 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	inner page	offset
p_1	p_2	d
42	10	12

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

64-bit Logical Address Space (Cont.)

- One solution is to divide the outer page table. Various ways of doing so. Example – three-level page table
  - Even with 2nd outer page table, the outer-outer table is still 2^{34} bytes in size.
  - And possibly 4 memory access to get to one physical memory location.
- The next step would be four-level. But ….

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

64-bit Logical Address Space (Cont.)

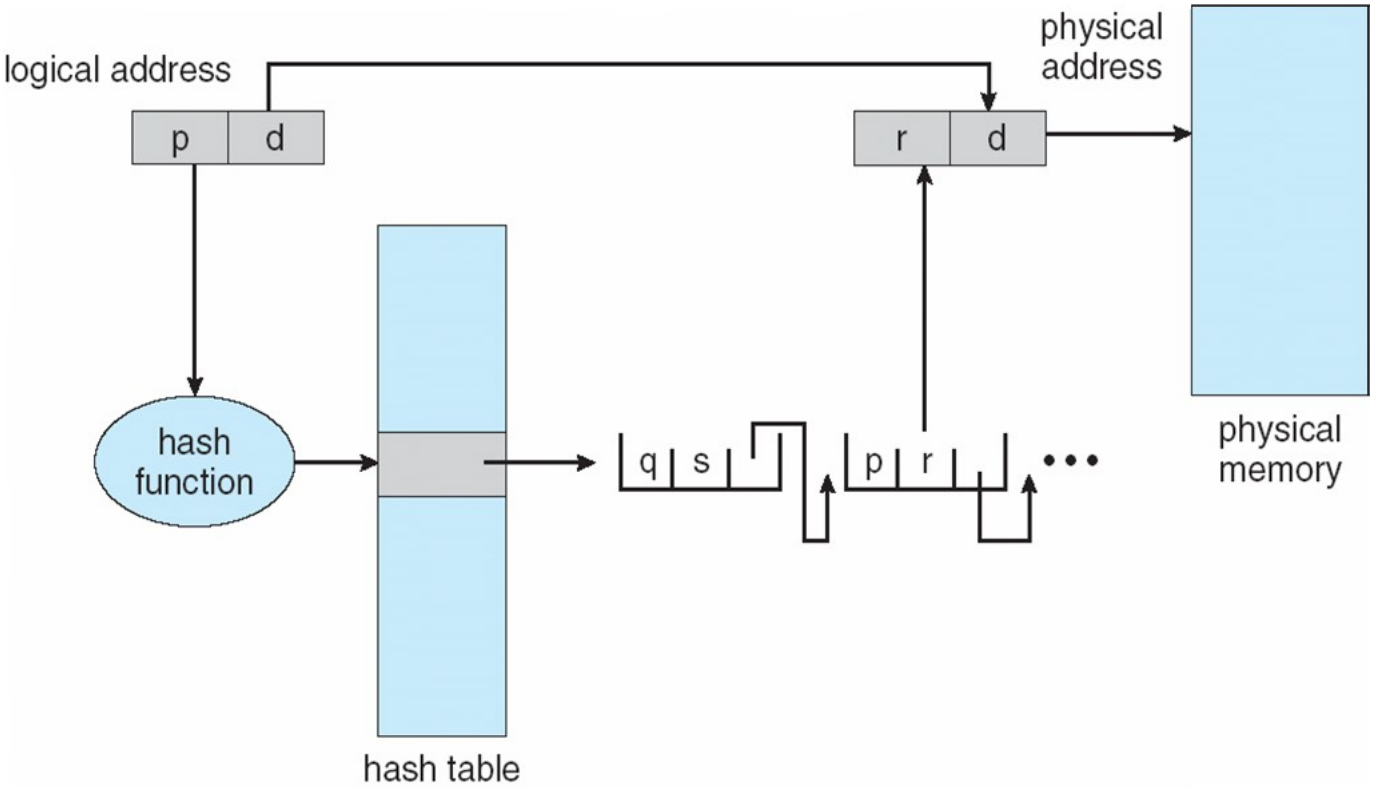
Several schemes for dealing with very large logical address space

- Hashed Page Table.
- Clustered Page Tables
- Inverted Page Table

Hashed Page Table

- 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

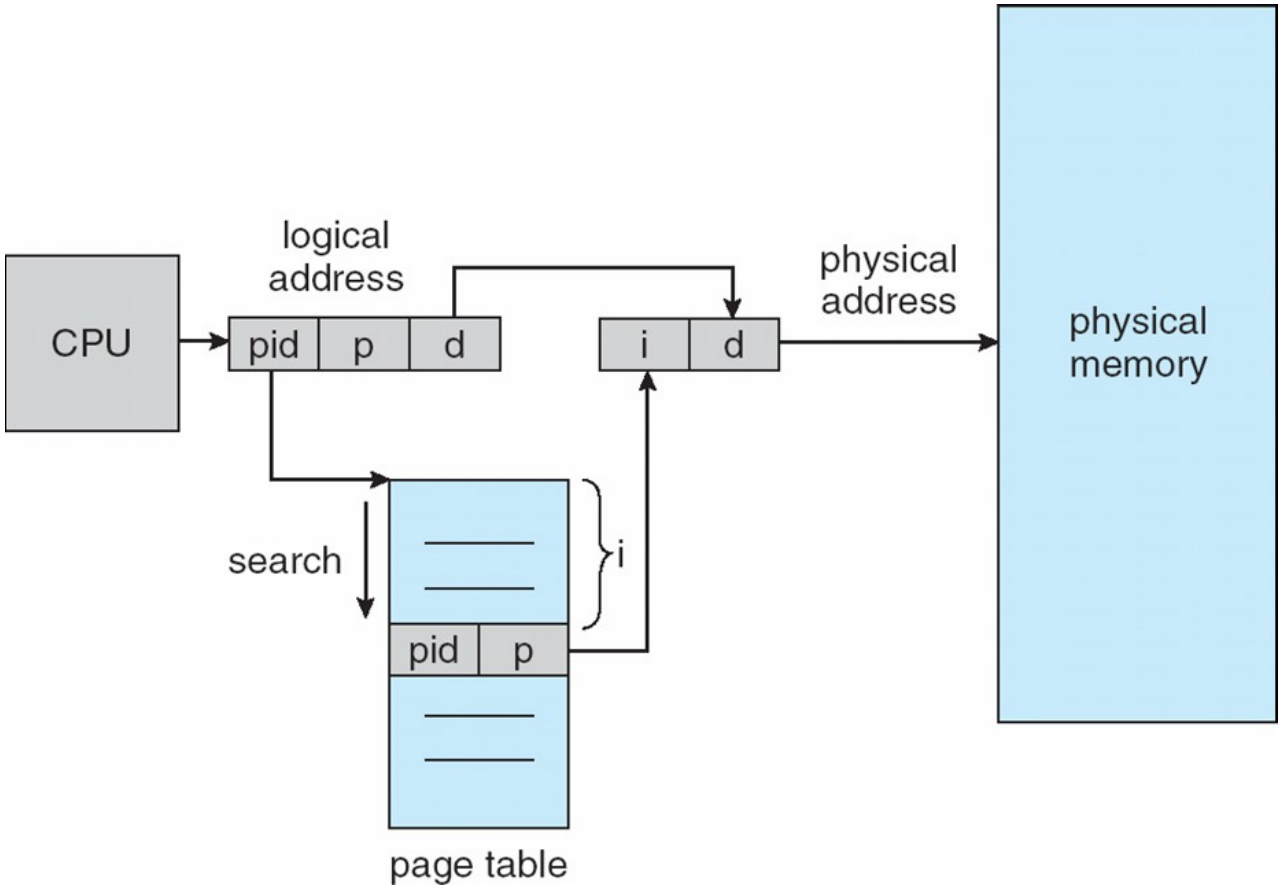
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all the physical pages
- Use **inverted page-table**, which has one entry for each real page of memory
- An entry the inverted-page table consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- What is maximum size of the inverted page-table?

Inverted Page Table Architecture



Inverted Page Table (Cont.)

- Decreases memory needed to store each individual page table, but increases time needed to search the inverted page 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

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

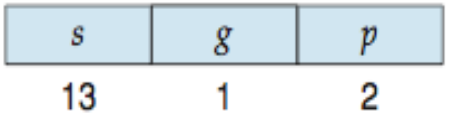
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

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**))

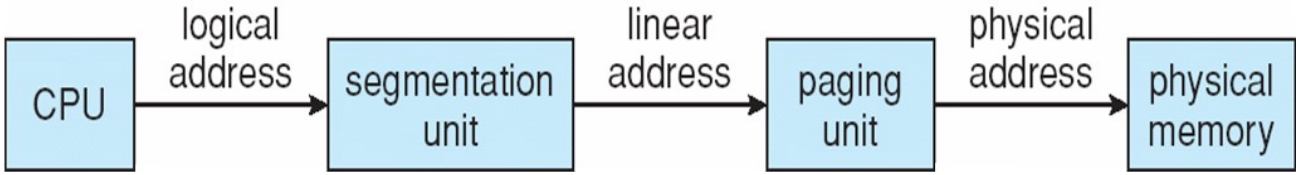
Example: The Intel IA-32 Architecture (Cont.)

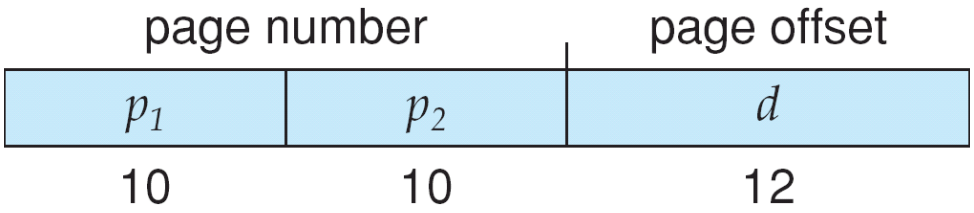
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



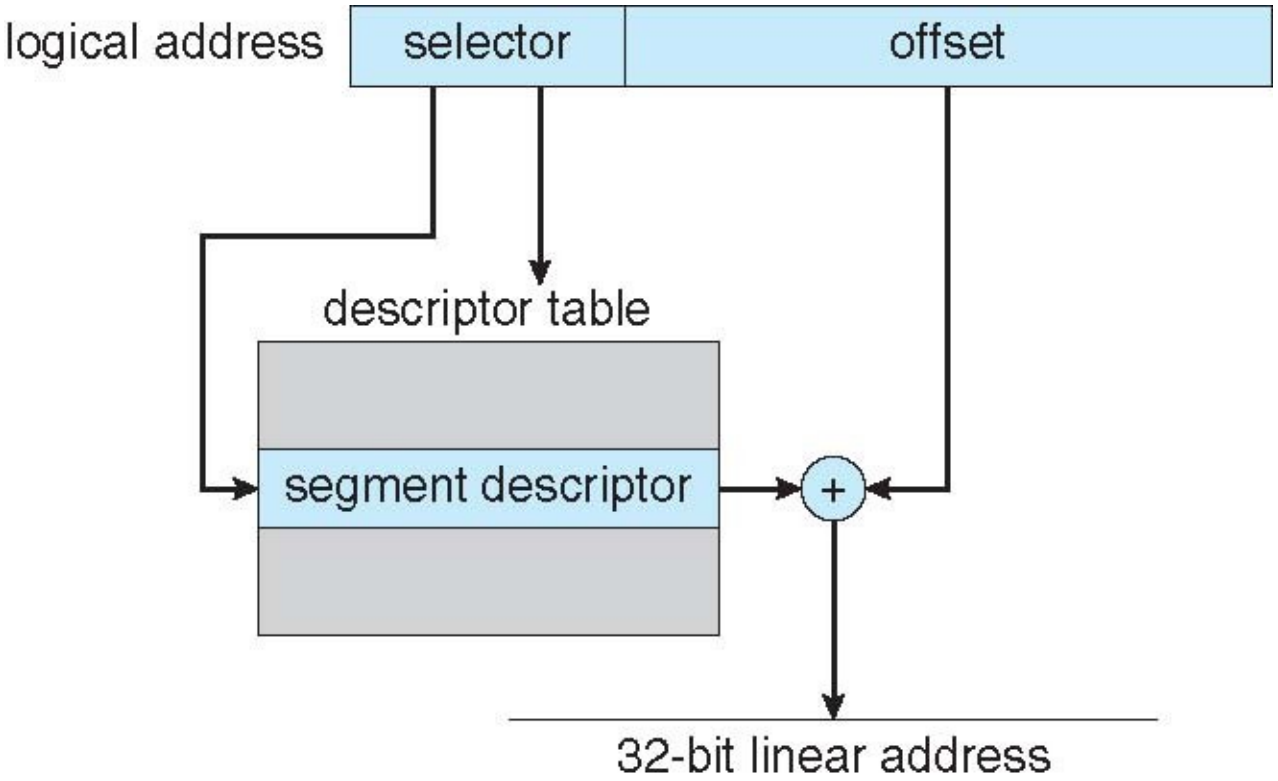
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Logical to Physical Address Translation in IA-32

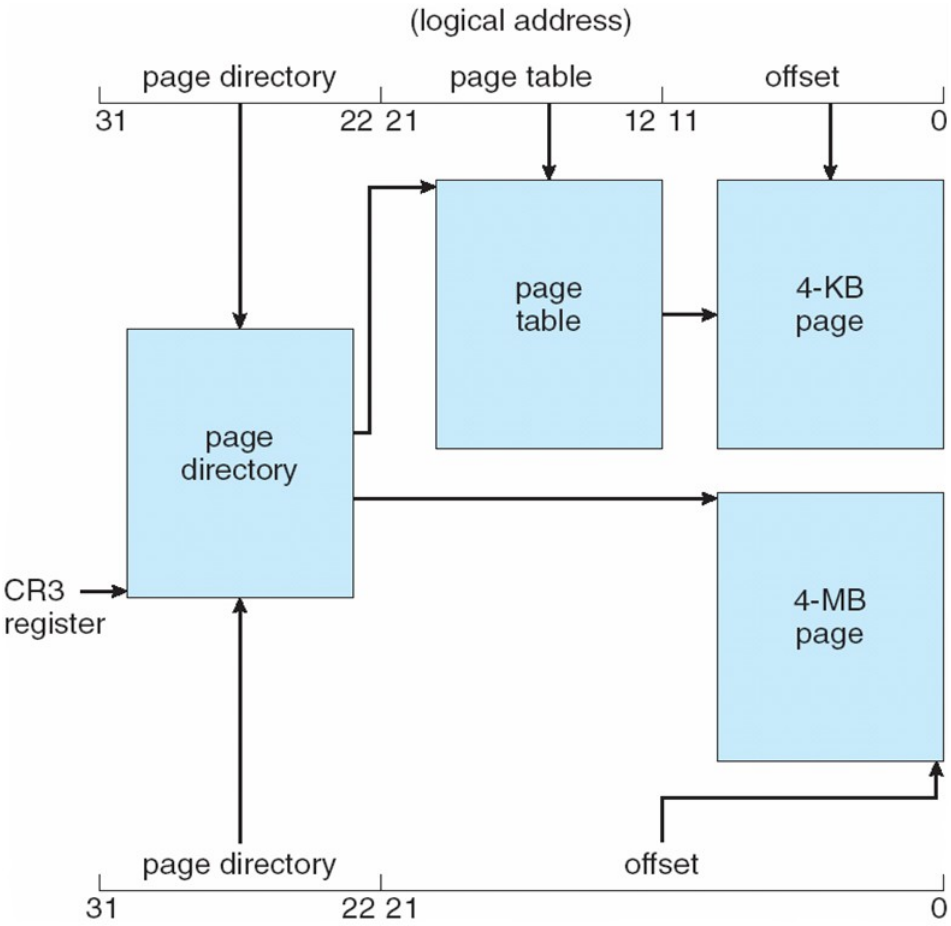




Intel IA-32 Segmentation

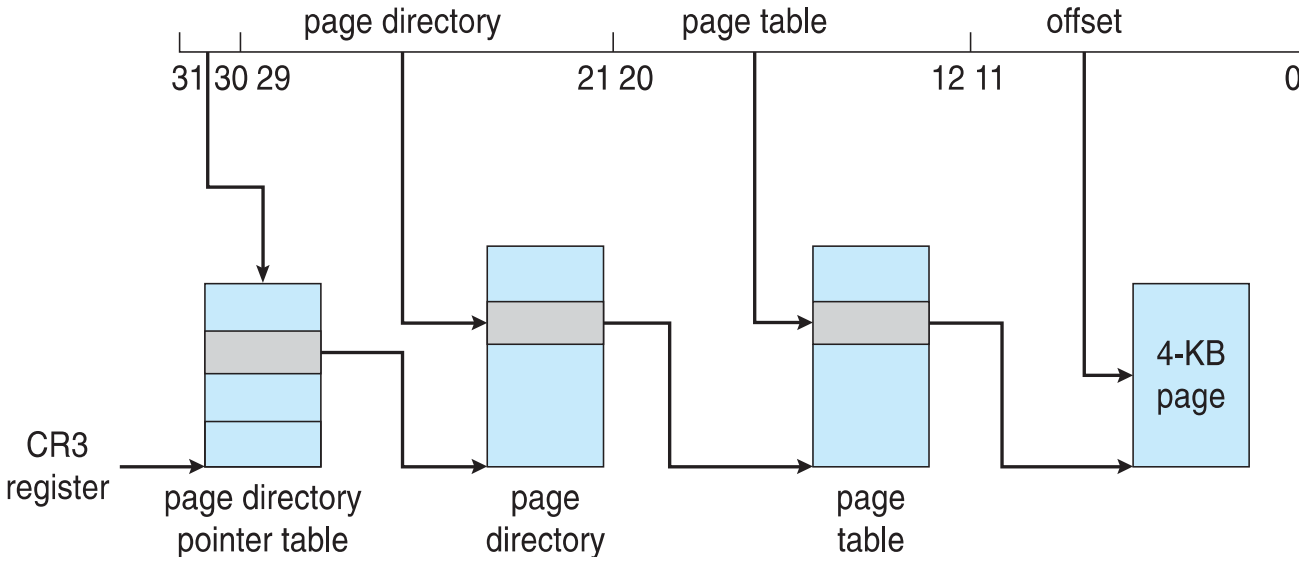


Intel IA-32 Paging Architecture



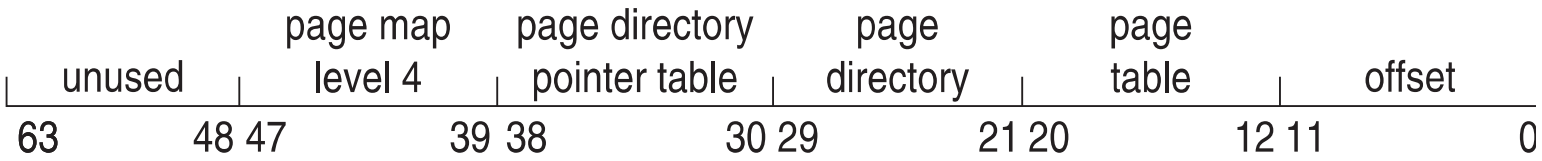
Intel IA-32 Page Address Extensions

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



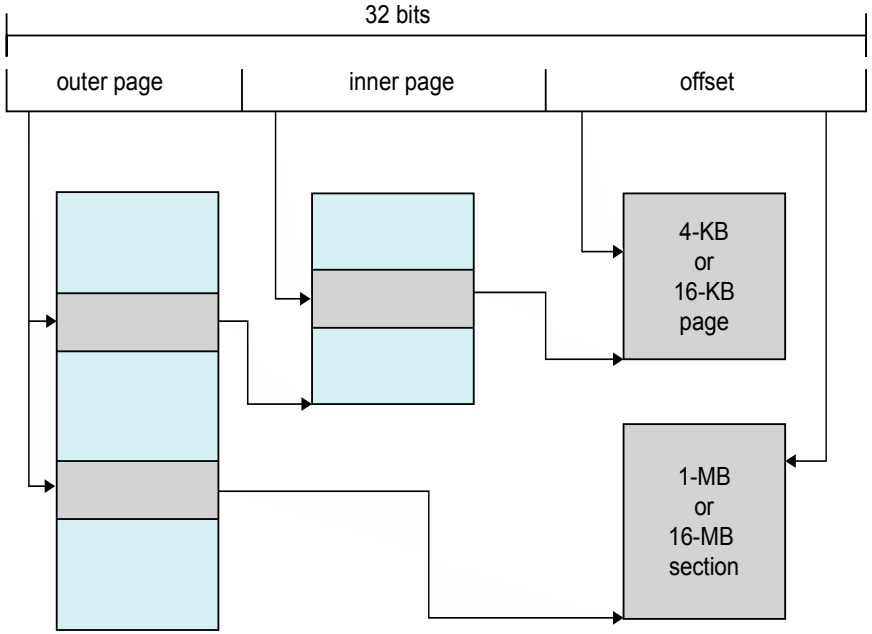
Intel x86-64

- 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
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 9