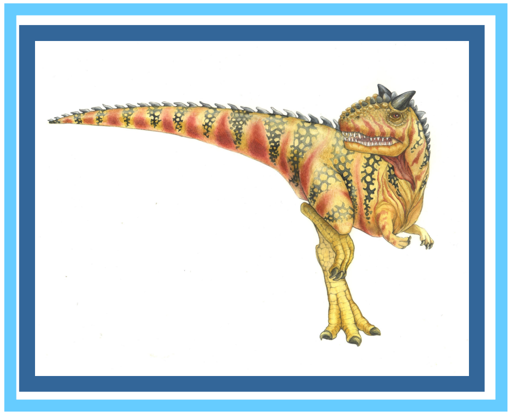
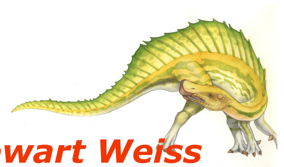
Chapter 8: Main Memory



Chapter 8: Memory Management

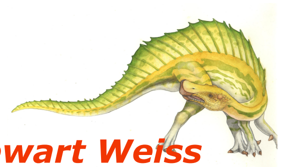
- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation



modified by Stewart Weiss

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



modified by Stewart Weiss

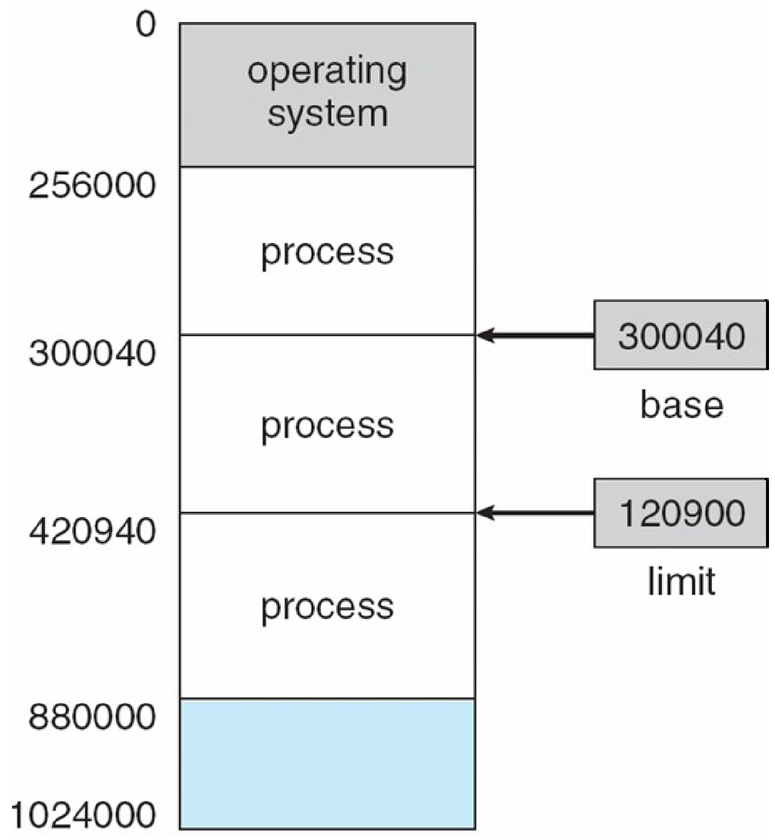
Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

modified by Stewart Weiss

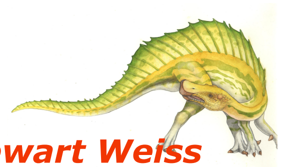
Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space



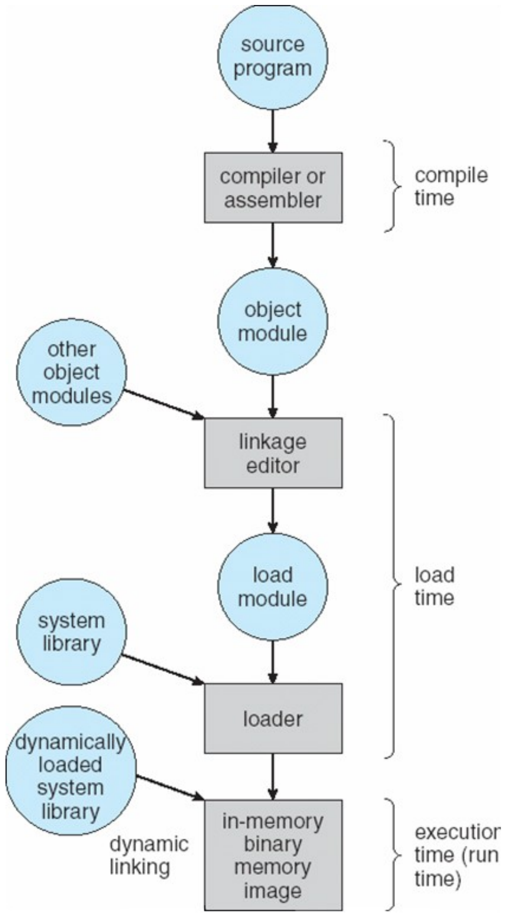
Binding of Instructions and Data to Memory

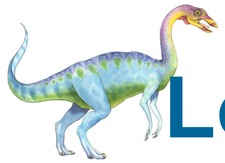
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory 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)



modified by Stewart Weiss

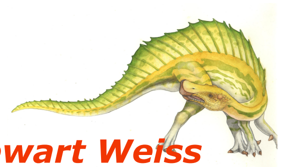
Multistep Processing of a User Program





Logical vs. Physical Address Space

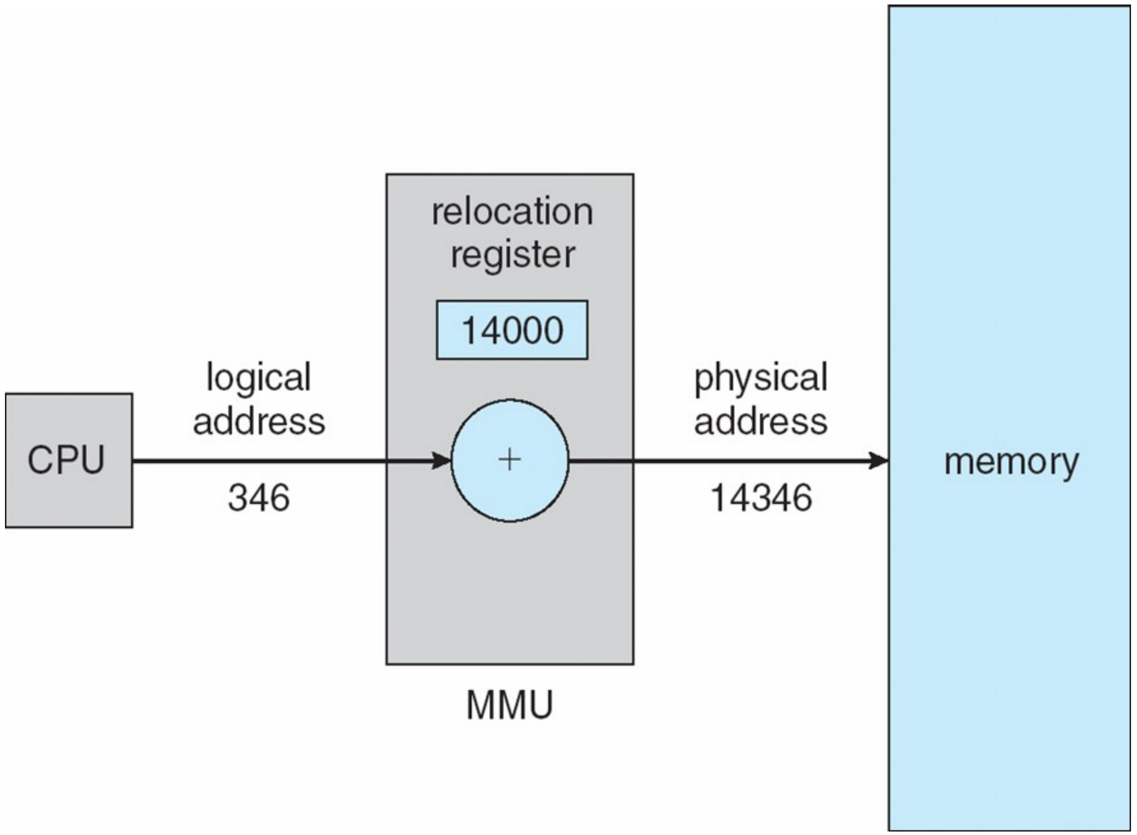
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical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



Memory-Management Unit (MMU)

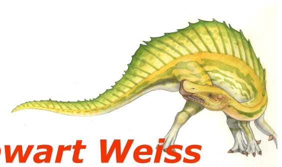
- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- A process is an abstraction that generates a stream of logical addresses.

Dynamic relocation using a relocation register



Dynamic Loading

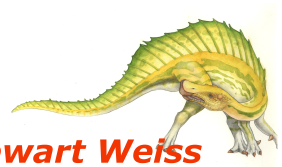
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design



modified by Stewart Weiss

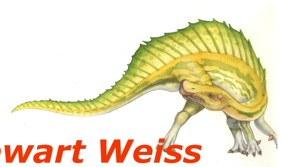
Dynamic Linking

- Linking 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 needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

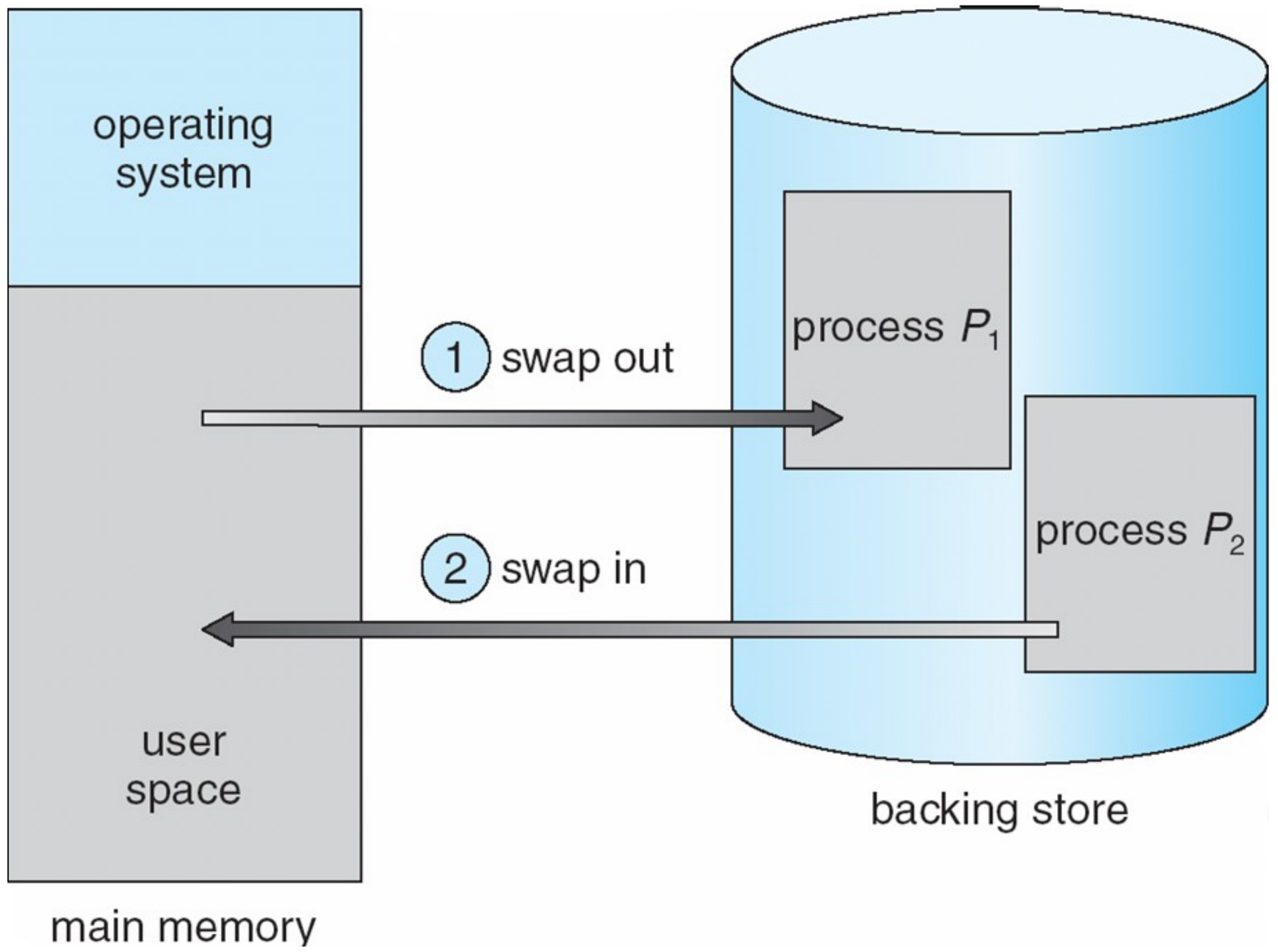


Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; 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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- True swapping is rarely used now; modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

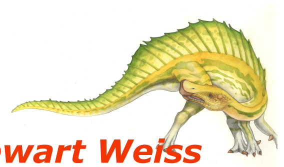


Schematic View of Swapping

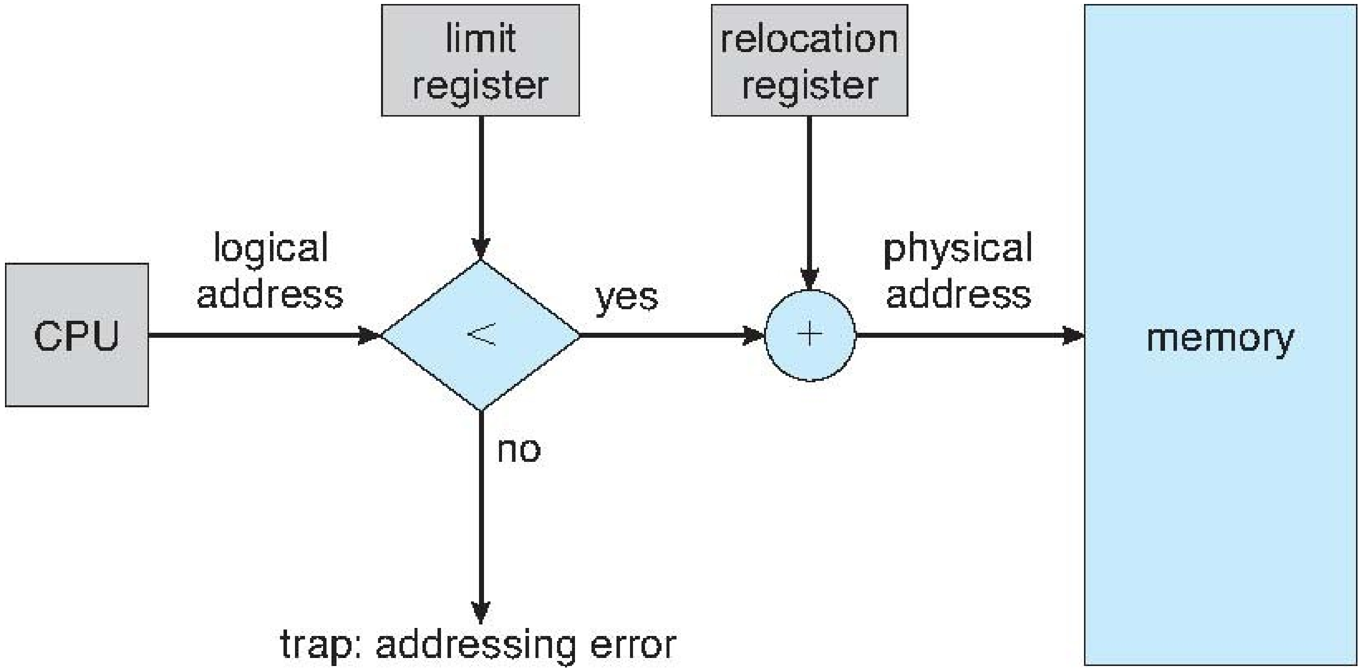


Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- 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*

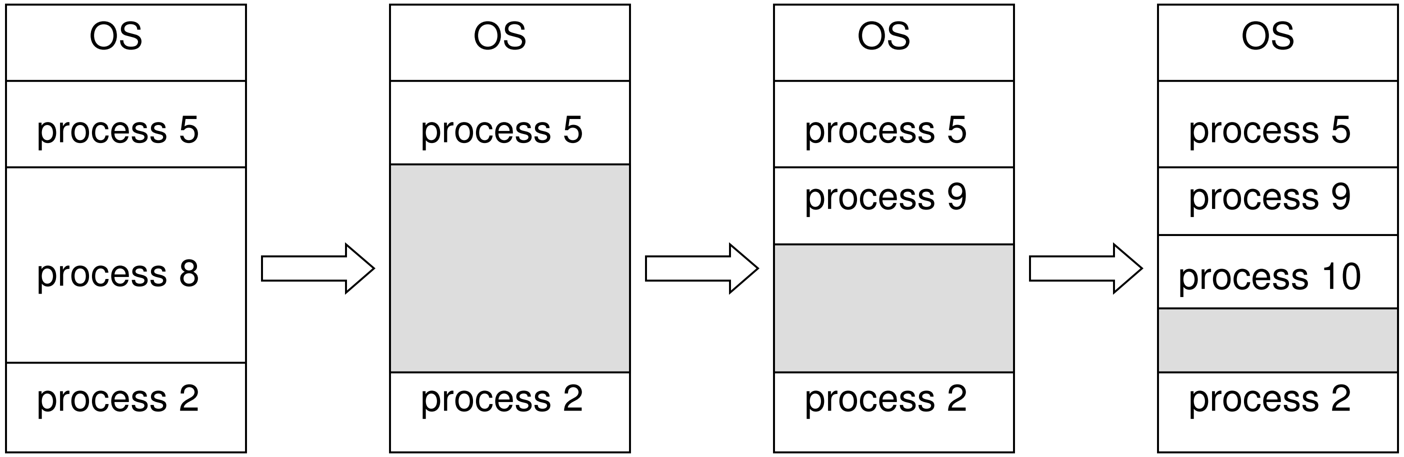


Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont)

- Multiple-partition allocation
  - 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
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)



modified by Stewart Weiss

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 ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit are about equal in terms of memory utilization, both better than worst-fit; first-fit is faster than either.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- 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

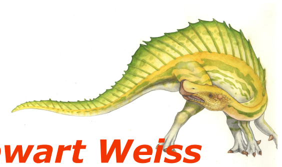
Fragmentation

- **50% Rule** – If there are n processes, then at any given time there should be about n/2 holes. Reasoning: during a process's lifetime, half of the operations on the partition immediately above it are allocations, and half are deallocations. Therefore, 50% of the time, there is a hole above it, and so on average, there are n/2 holes.
- Unused Memory Rule – If k is the average hole size divided by the average process size,
  - then the fraction of unused memory is k/(2+k)

modified by Stewart Weiss

Internal Fragmentation

- The kind of fragmentation described before is ***external fragmentation*** – it occurs outside of the partition of memory allocated to processes.
- In order to make management of holes easier, memory is usually allocated in multiples of a fixed-size block, such as 512 bytes or 1024 bytes. If a process requires 5000 bytes, it will be given ten 512 byte blocks for a total of 5120, for example, with the last block containing 120 bytes of unused memory. This unused memory inside of a partition is called ***internal fragmentation***.



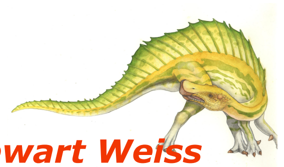
modified by Stewart Weiss

Noncontiguous Memory Allocation

- External fragmentation can be eliminated if the memory allocated to a process does not need to be ***contiguous***. In other words, if a process can be broken up into smaller pieces that can be loaded into the available holes, wherever they are, then as long as there is space available, processes needing memory can be given it.
- There are two such methods of noncontiguous memory allocation – ***paging*** and ***segmentation***.

Paging

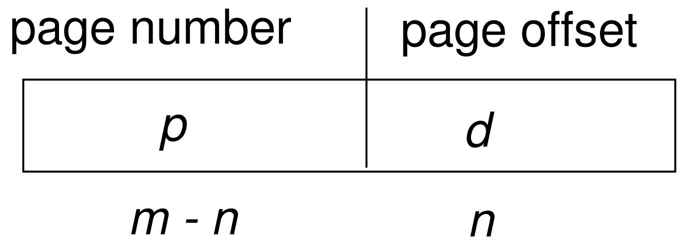
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size 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



modified by Stewart Weiss

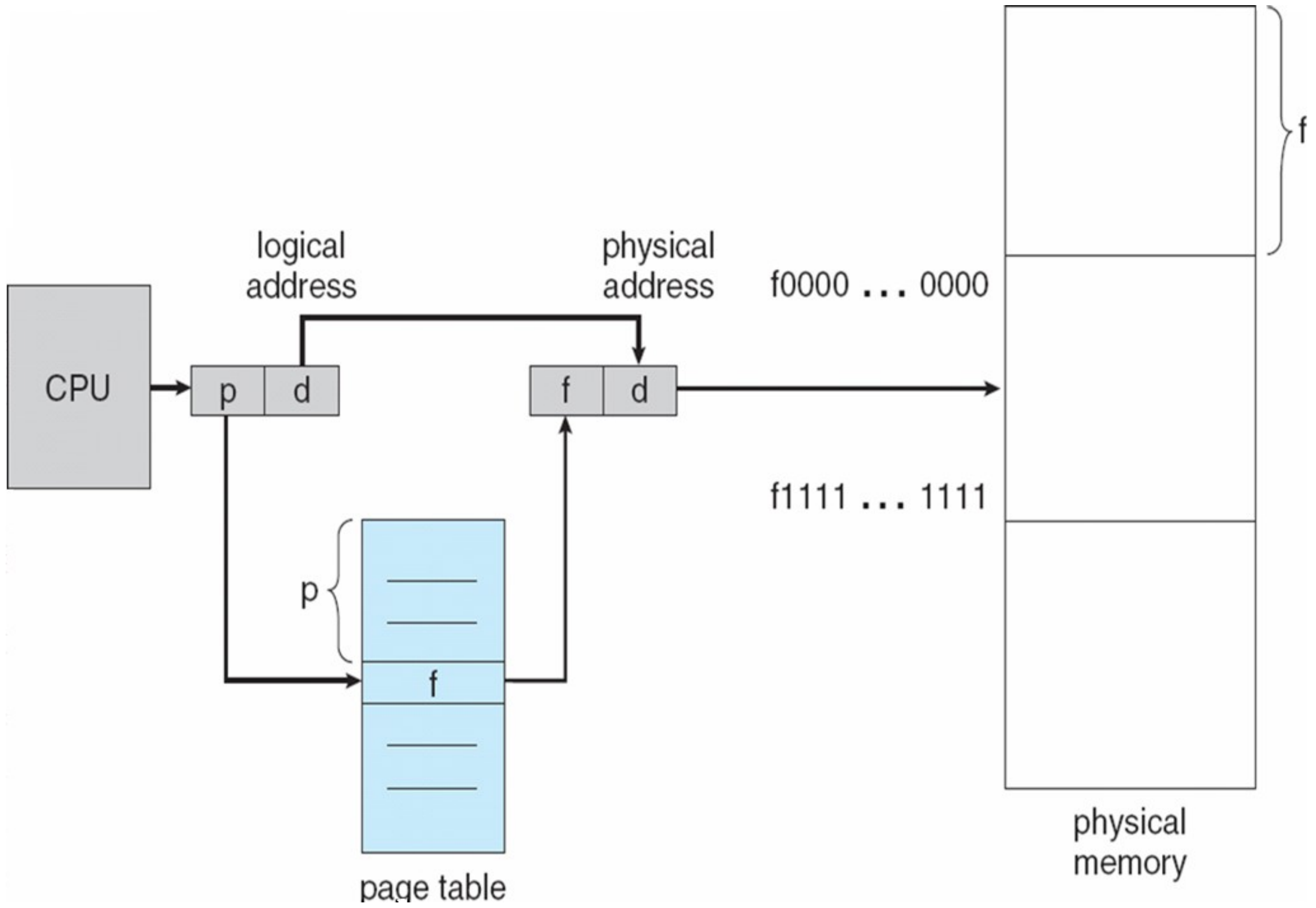
Address Translation Scheme

- Address generated by CPU is divided into a page number and a page offset. If each page has 2^n words, then the low-order n bits are the offset, and the remaining high-order bits are the page number.
  - **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



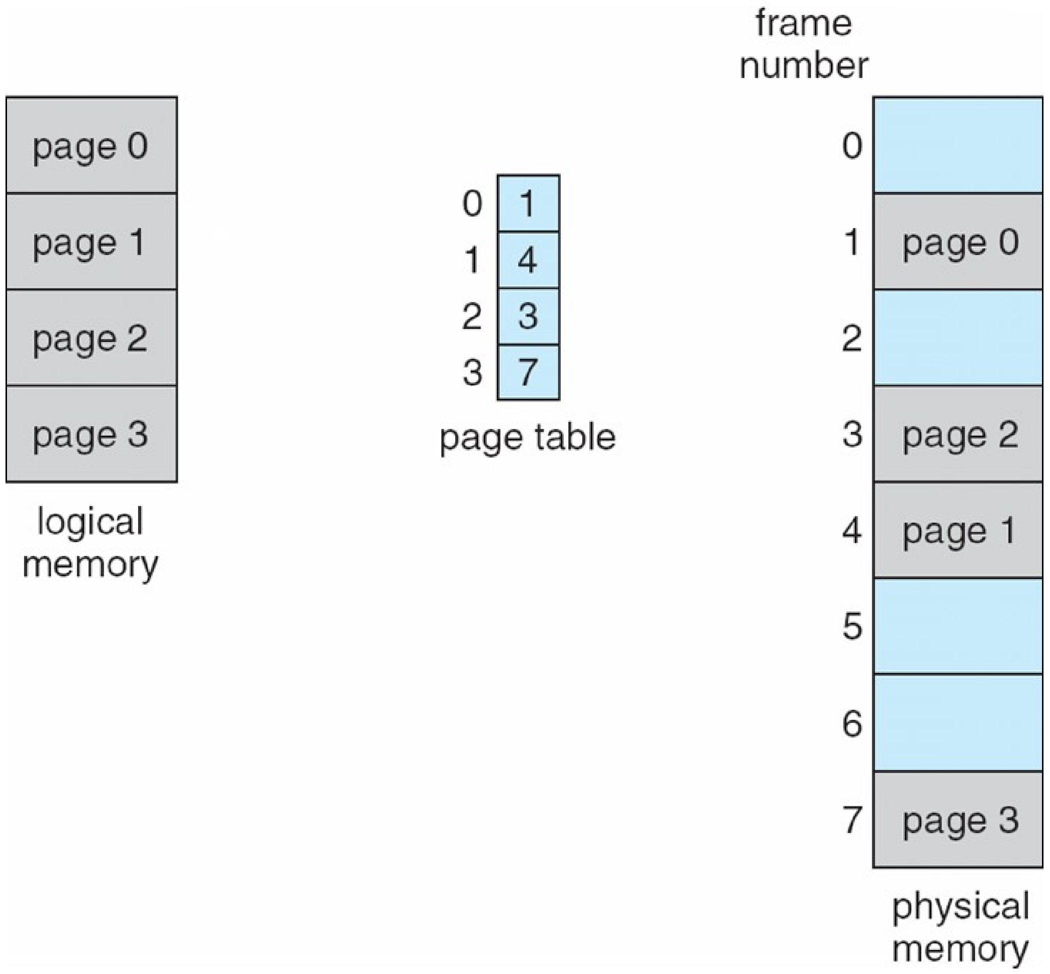
where the logical address space has 2^m *words.*

Paging Hardware for Address Translation



modified by Stewart Weiss

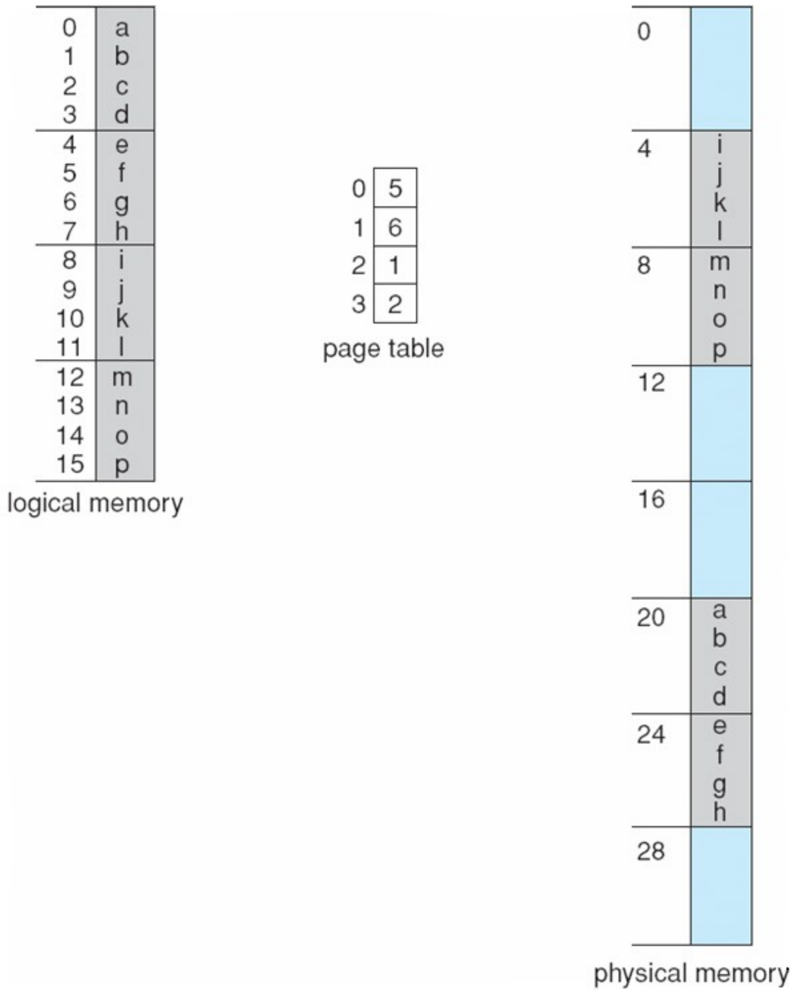
Paging Model of Logical and Physical Memory



Internal Fragmentation in Paging

- In general, the last page of every process is not completely full, since processes are arbitrary sizes. On average it is half full. This is ***internal fragmentation***.
- Therefore, each process has ½ page of unused memory on average. If page size is very large, there is more unused memory.
- If page size is small, there is much less wasted memory, but smaller pages require more page table entries
- We will do the page table math later.

Paging Example

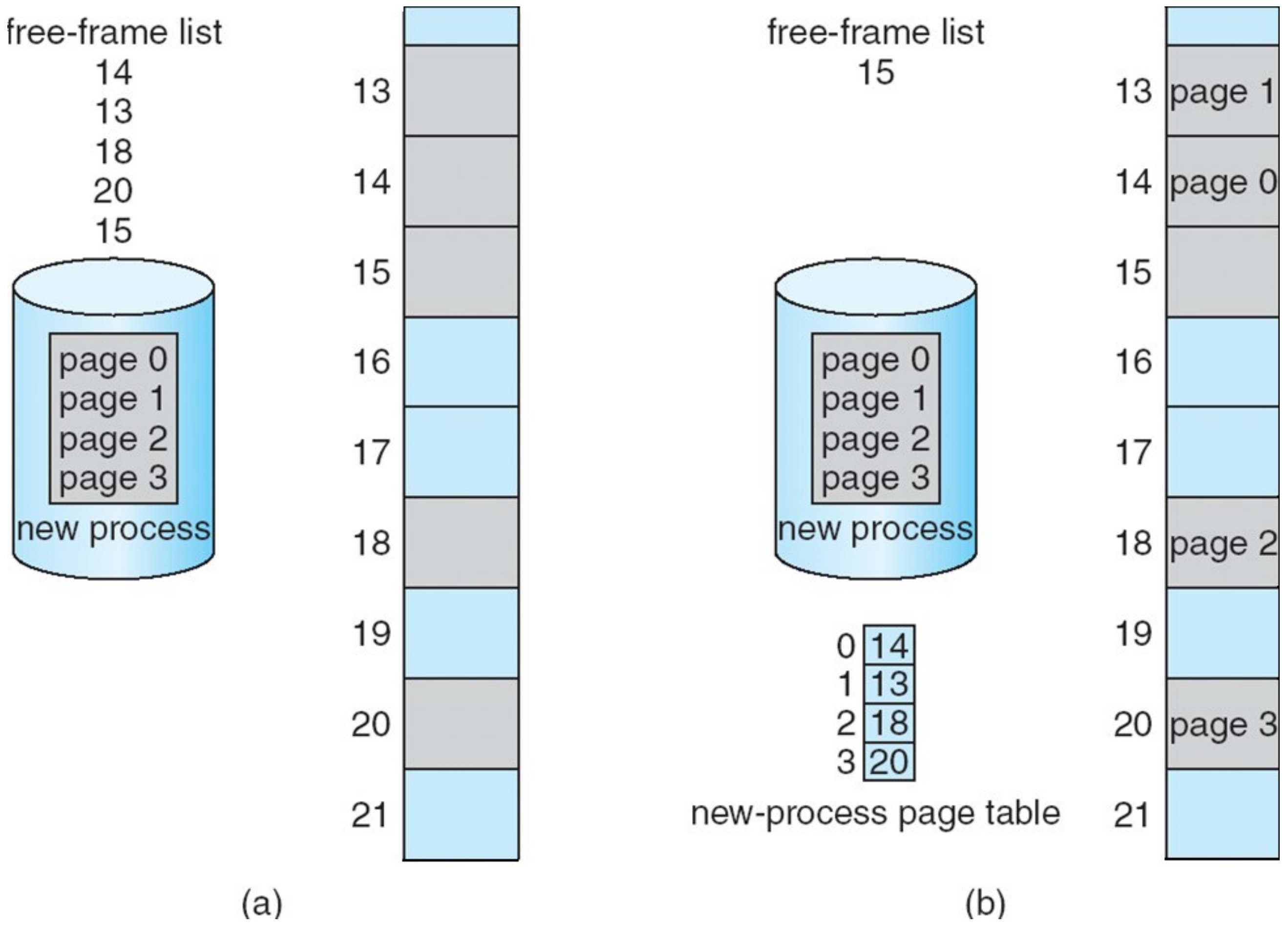


32-byte memory and 4-byte pages

Scheduling and Paging

- If a process with n pages arrives in the system, it will require n free page frames in order to run.
- The medium term scheduler decides whether the process can be loaded. If the frames are available, it is loaded; otherwise it waits for memory to be available.
- The operating system maintains a table, usually called the frame table, that shows which frames are free and which are in use. The table indicates which process is using them and which pages of the process are in which frames.

Free Frames

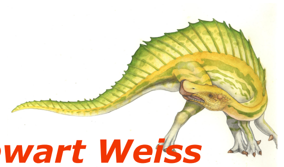


Before allocation

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 (PRLR)** 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)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process



modified by Stewart Weiss

Page Table Math

- Page sizes usually range between 4 KB and 8 KB now.
- A page table entry with 32 bits (4 bytes) can reference 2^{32} frames. If a frame is 4 KB, then 4-byte page table entries can address $2^{32} \times 2^{12} = 2^{44}$ bytes of physical memory.
- Suppose pages are 4 KB = 2^{12} bytes, and a process can be as large as 512 MB = 2^{29} bytes. Then it can have 2^{17} pages. If each page table entry is 4 bytes, then the page table for each process would have $4 \times 2^{17} = 2^{19}$ bytes, or 512 KB, roughly 0.1% of the process's size.

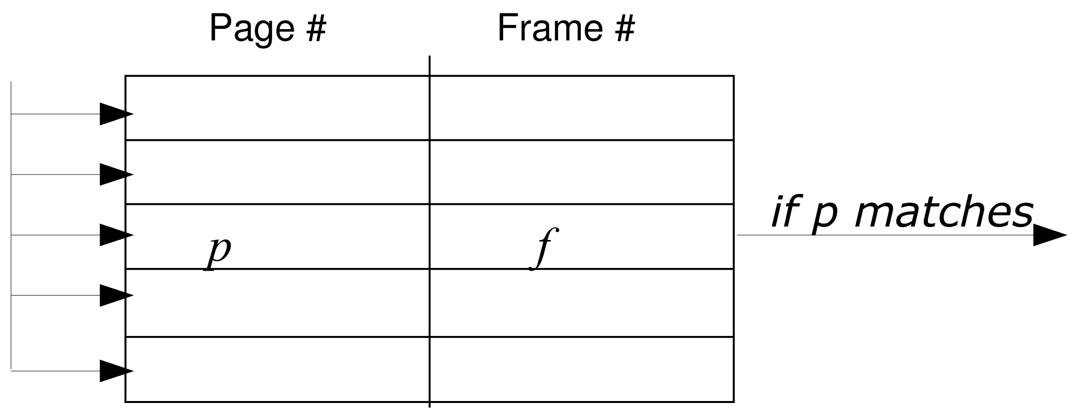
Page Table Implementation

- Page tables can be stored in registers if they are very small.
- Better is to store them in main memory, and a special register, the Page Table Base Register (PTBR) pointing to its starting address. Problem – each memory reference requires first a memory reference to get the page table entry, then to get the actual data, doubling the number of memory references.
- Better still, using a small associative memory to act as a cache of recently used page table entries. Called a Translation Lookaside Buffer (TLB).
- Associative memory is a special kind of table in which all rows are searched simultaneously for a matching key. If the key is found in any row, the corresponding value is put on the data path.

modified by Stewart Weiss

Associative Memory

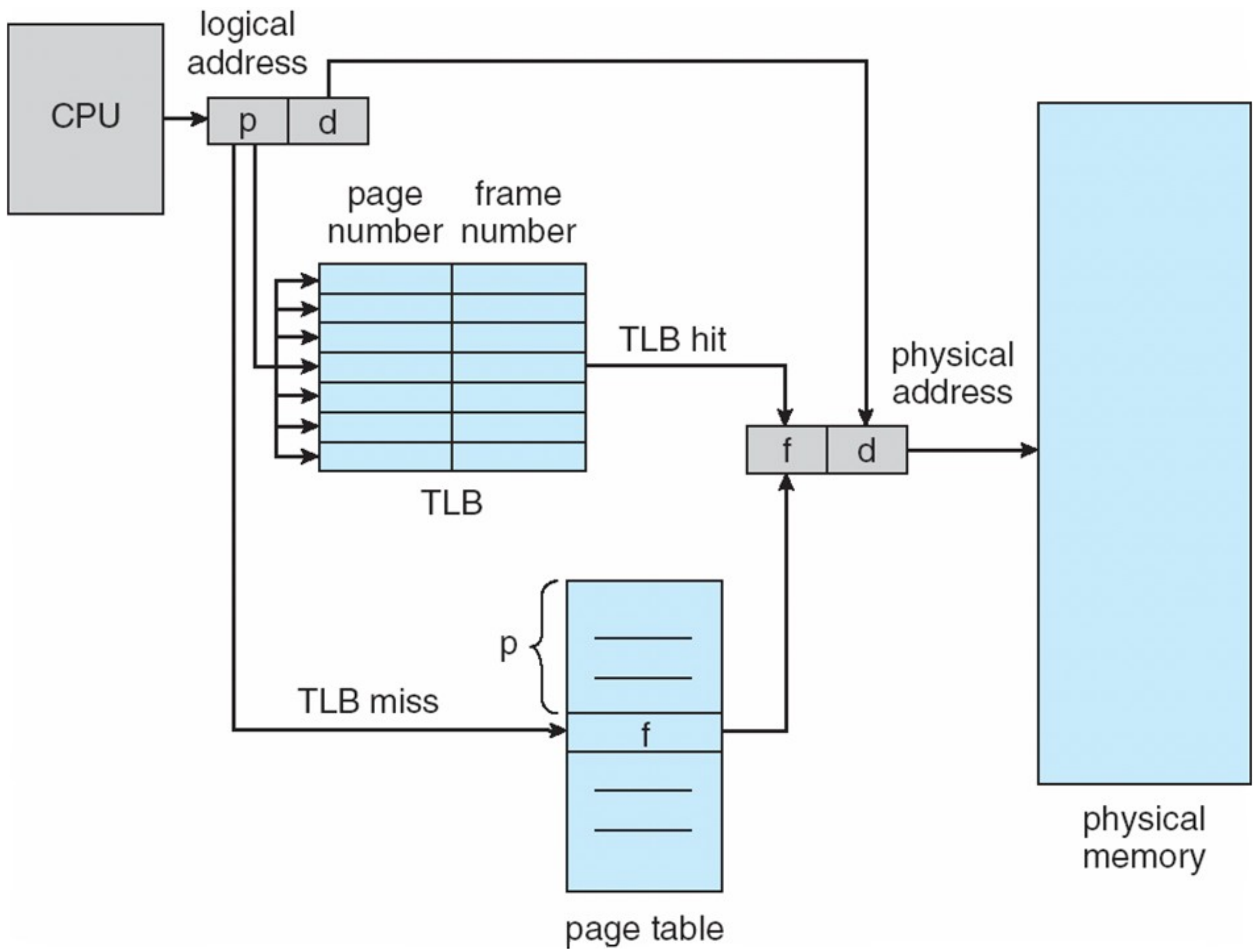
- Associative memory – parallel search



Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



modified by Stewart Weiss

Effective Access Time

- Assume Associative Lookup = ε time units
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Assume Hit ratio = α
- **Effective Access Time** (EAT)

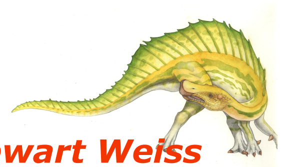
$$\text{EAT} = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

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

- Example: if hit ratio is 0.8 and TLB lookup time is 0.2 microseconds, then EAT is 1.4 microseconds.

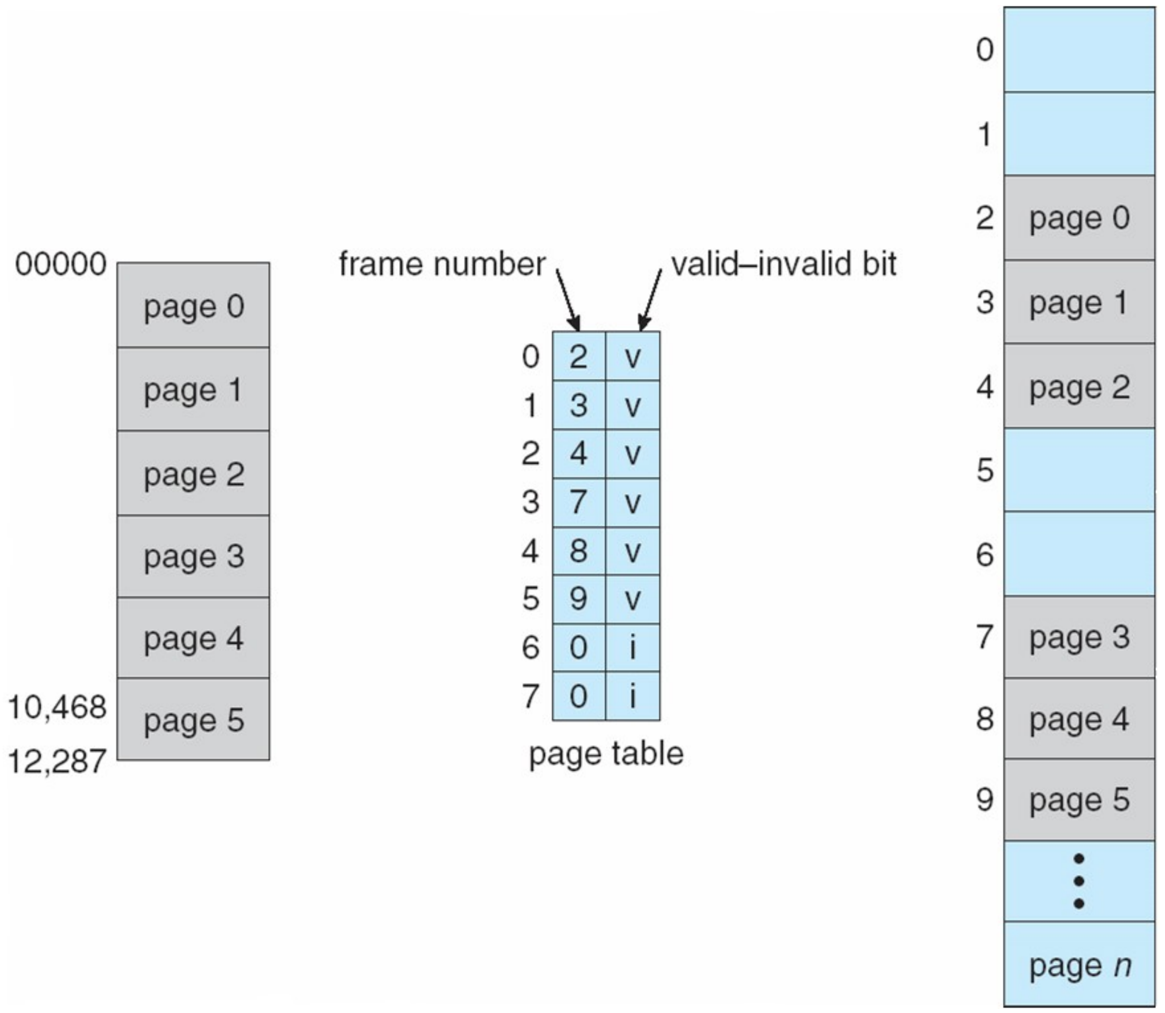
Memory Protection

- Memory protection implemented by associating protection bits with each frame in physical memory.
- Protection bits include: read bit; write-bit; execute-bit
- Process also has protection bits in the process page table for each page; Op Sys sets frame protection according to page protection requirements.
- **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 (outside of process address space but still has entry in table)
  - instead of valid bit, Page Table Length Register is used to indicate length of table; trap if length exceeded.



modified by Stewart Weiss

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

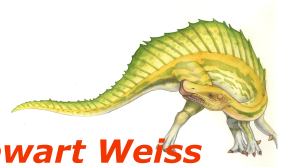


modified by Stewart Weiss

Shared Pages

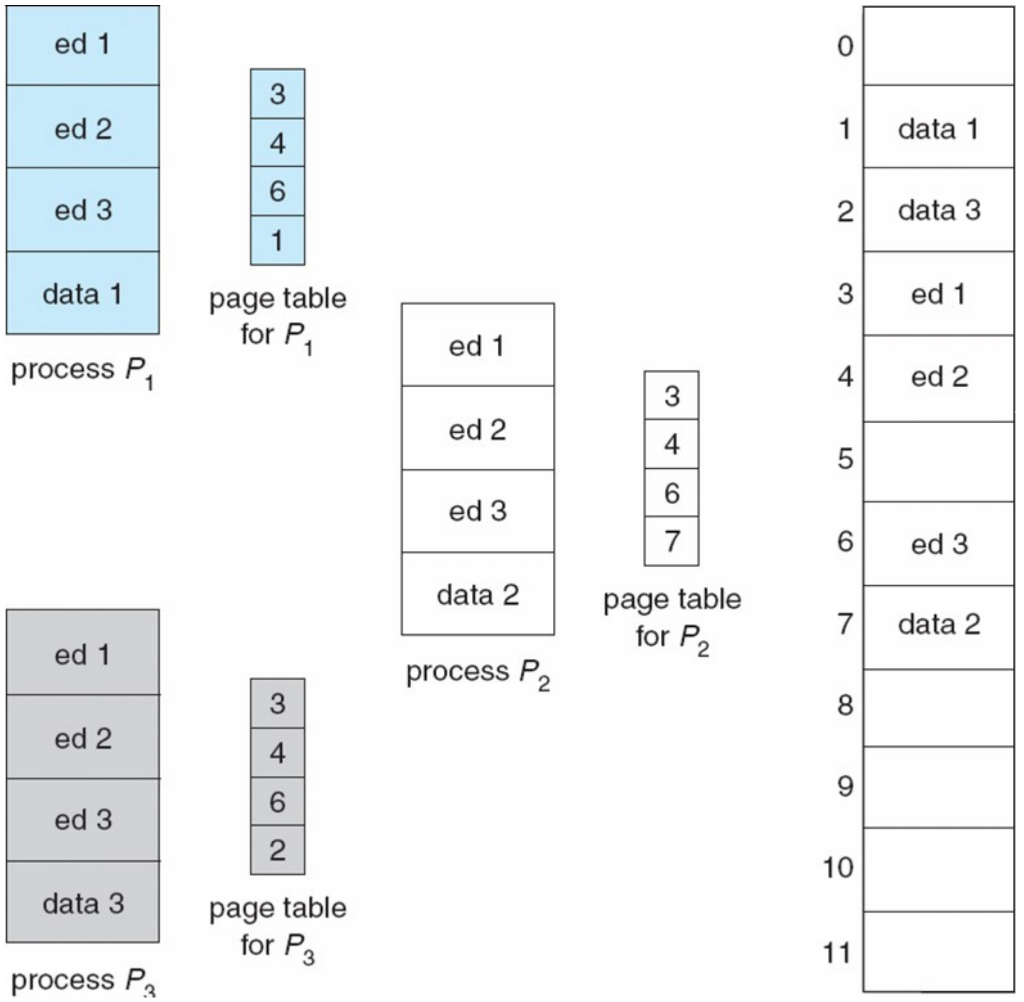
- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **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



modified by Stewart Weiss

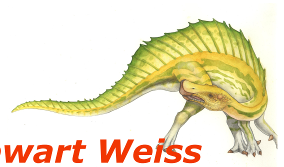
Shared Pages Example



modified by Stewart Weiss

Structure of the Page Table

- Suppose physical memory has 2^{32} addressable bytes and each frame is 2^{12} bytes. Then there are $2^{32}/2^{12} = 2^{20}$ frames of memory.
- Frame table has 2^{20} entries. If each entry has a 4 byte descriptor, then page table is 2^{22} bytes in size, or 1/1024 of physical memory for frame table alone.
- If each process has 2^{28} logical address space, it needs a page table with 2^{16} entries, each with 4 bytes per entry, for a total of 2^{18} bytes in its page table. If this page table were stored contiguously in physical memory, it would make allocating memory difficult.
- Page tables are therefore stored non-contiguously in pages themselves. A page table is needed to access the pages of the page table, leading to a ***hierarchical page table design***.
- Other alternatives are ***hashed page tables*** and ***inverted page tables.***



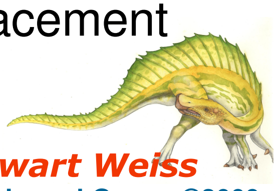
modified by Stewart Weiss

Two-Level Paging Example

- A logical address on 32-bit machine with 4KB page size is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits (4KB = 2^{12} bytes)
- Since the page table is paged, and each entry is 4 bytes, 4KB/4 bytes = 2^{10} entries fit into a page. Since there are 2^{20} entries, the page table itself requires $2^{20}/2^{10} = 2^{10}$ pages. Therefore, the 20 bit page number is further divided into:
  - a 10-bit secondary page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

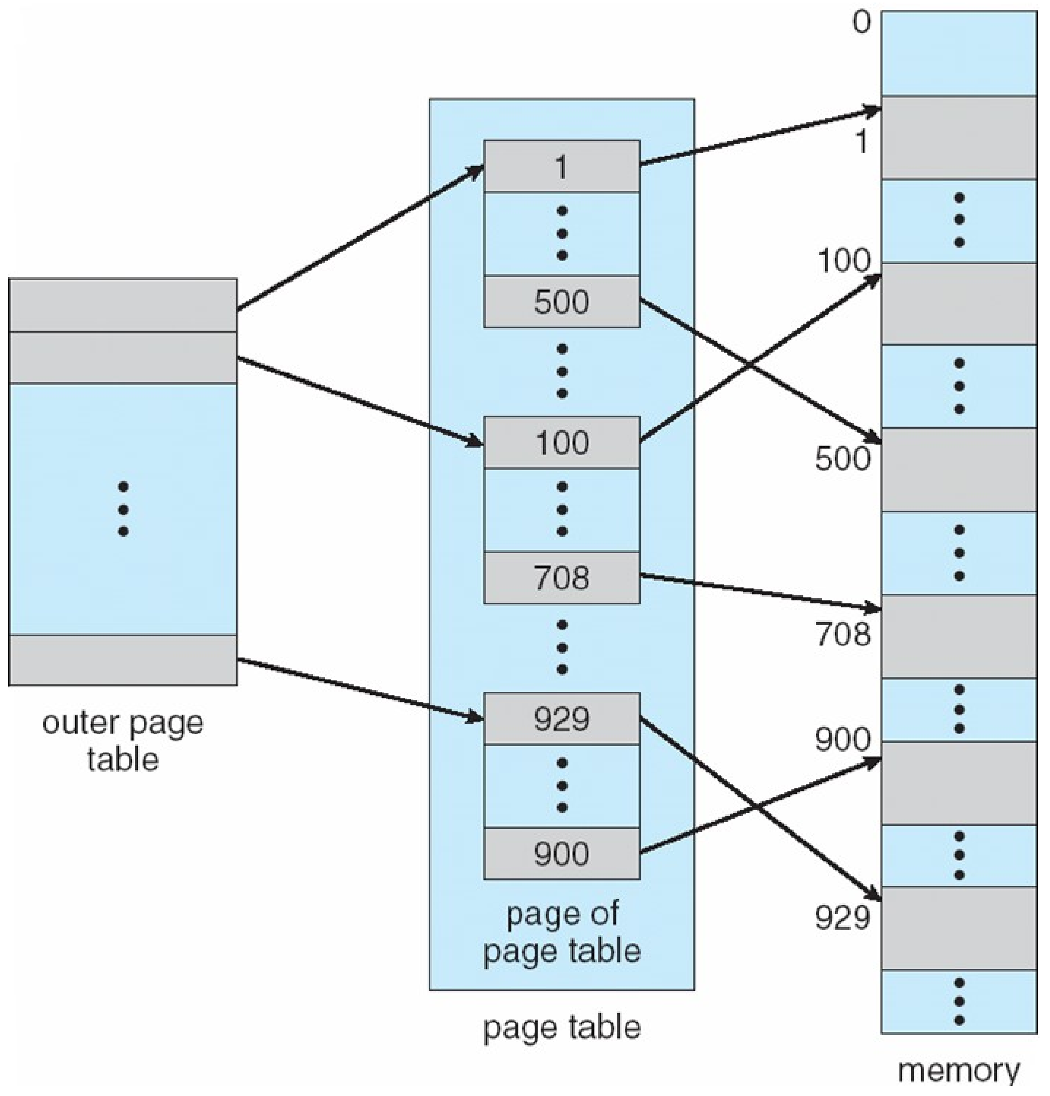
page number		page offset
p_i	p_2	d
10	10	12

- where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table



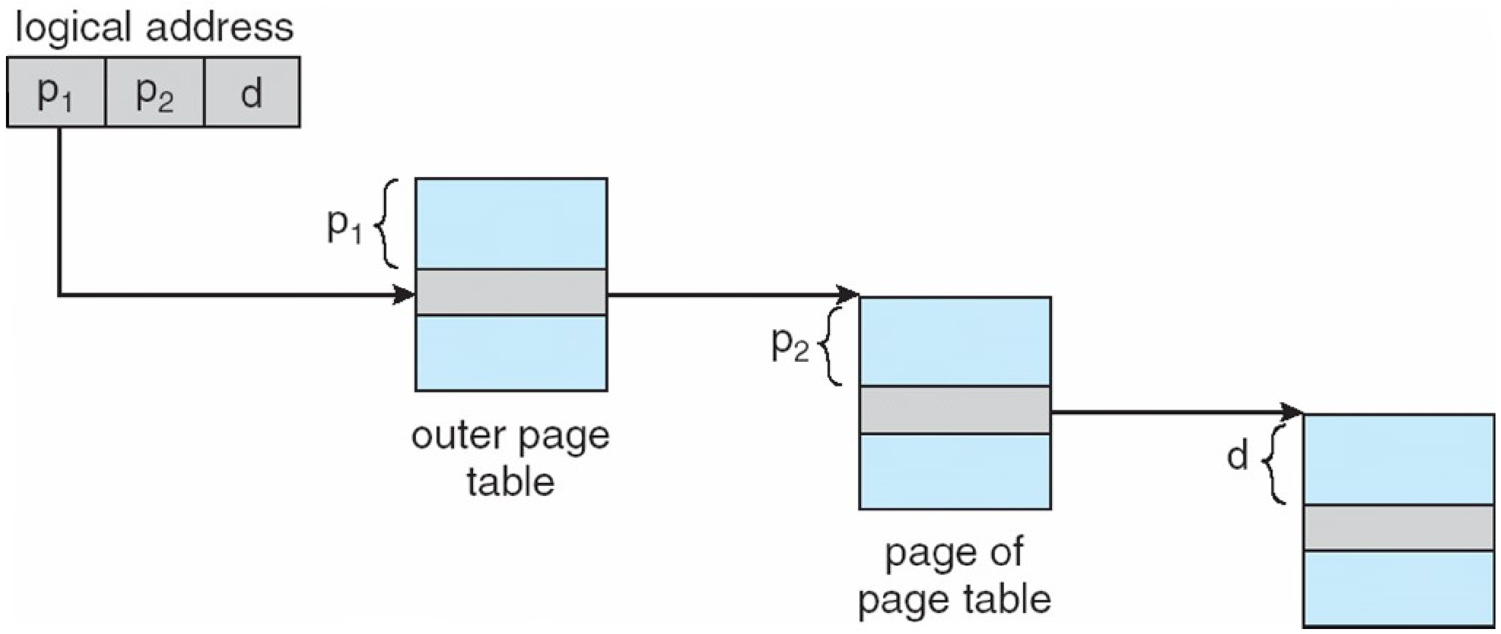
modified by Stewart Weiss

Two-Level Page-Table Scheme



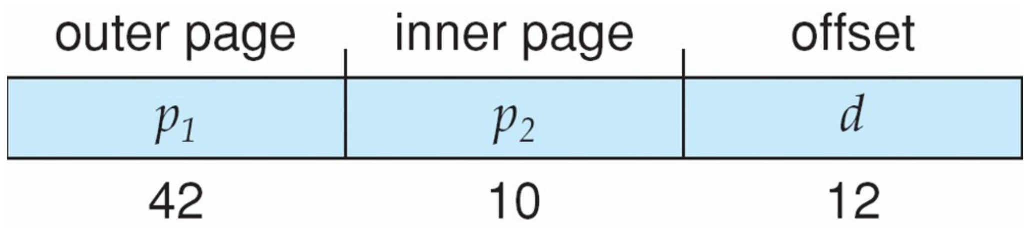
modified by Stewart Weiss

Address-Translation Scheme

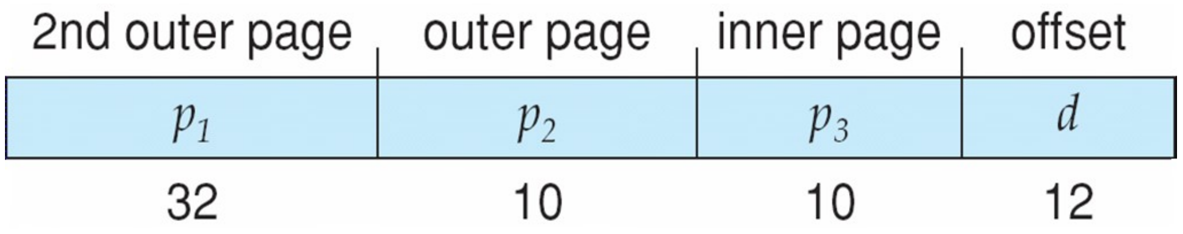


Three-Level Paging

- For 64-bit architectures, two-level paging is a problem. It would lead to an outer table that would be much too large. With 4KB pages, the logical 64-bit address would look like:



- The outer page table itself would have 2^{42} entries, or 2^{44} bytes, which is too large. We would need three levels, like this:

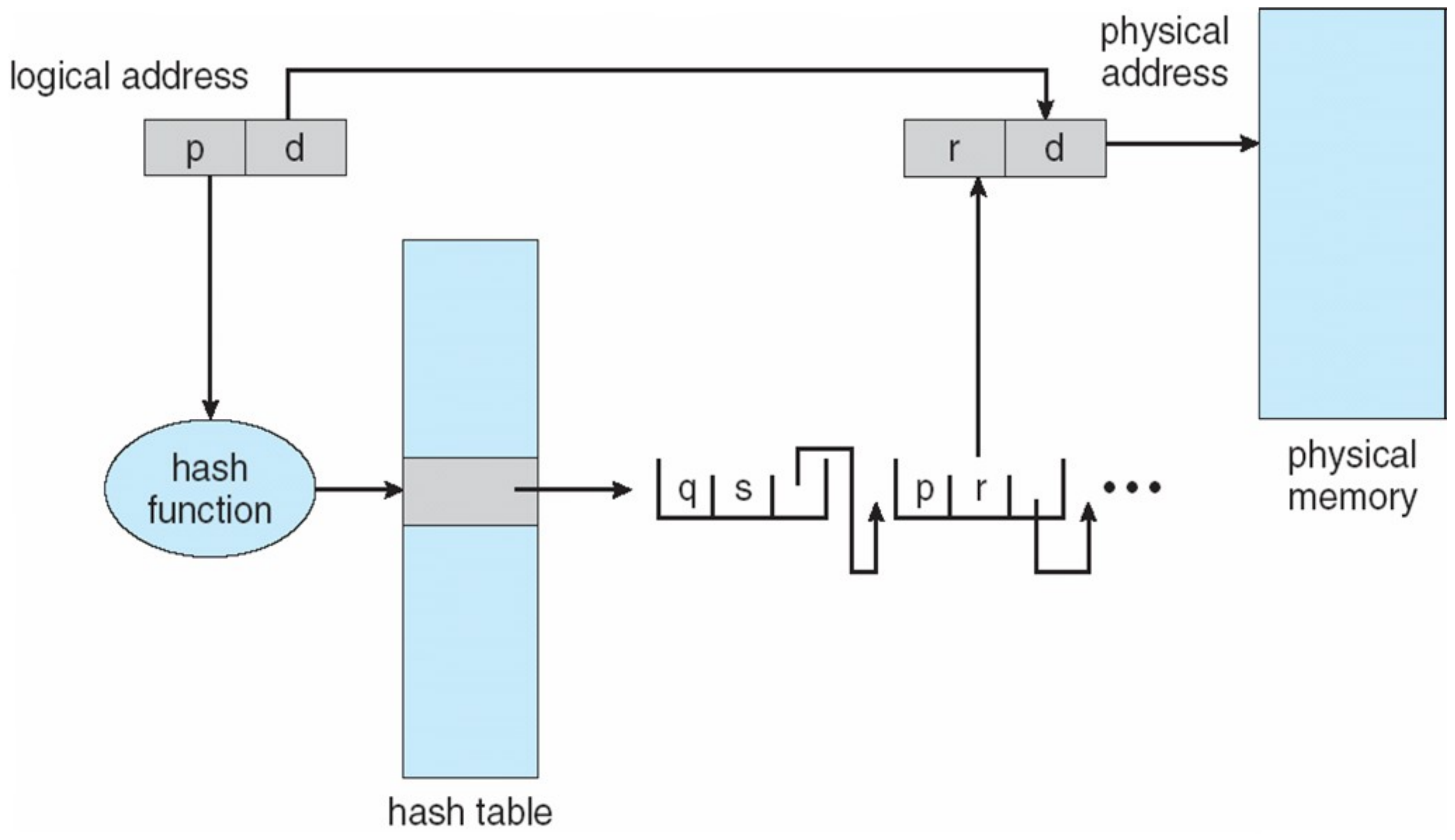


- with an outer level table with 2^{32} entries, which is still too large.

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
- 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

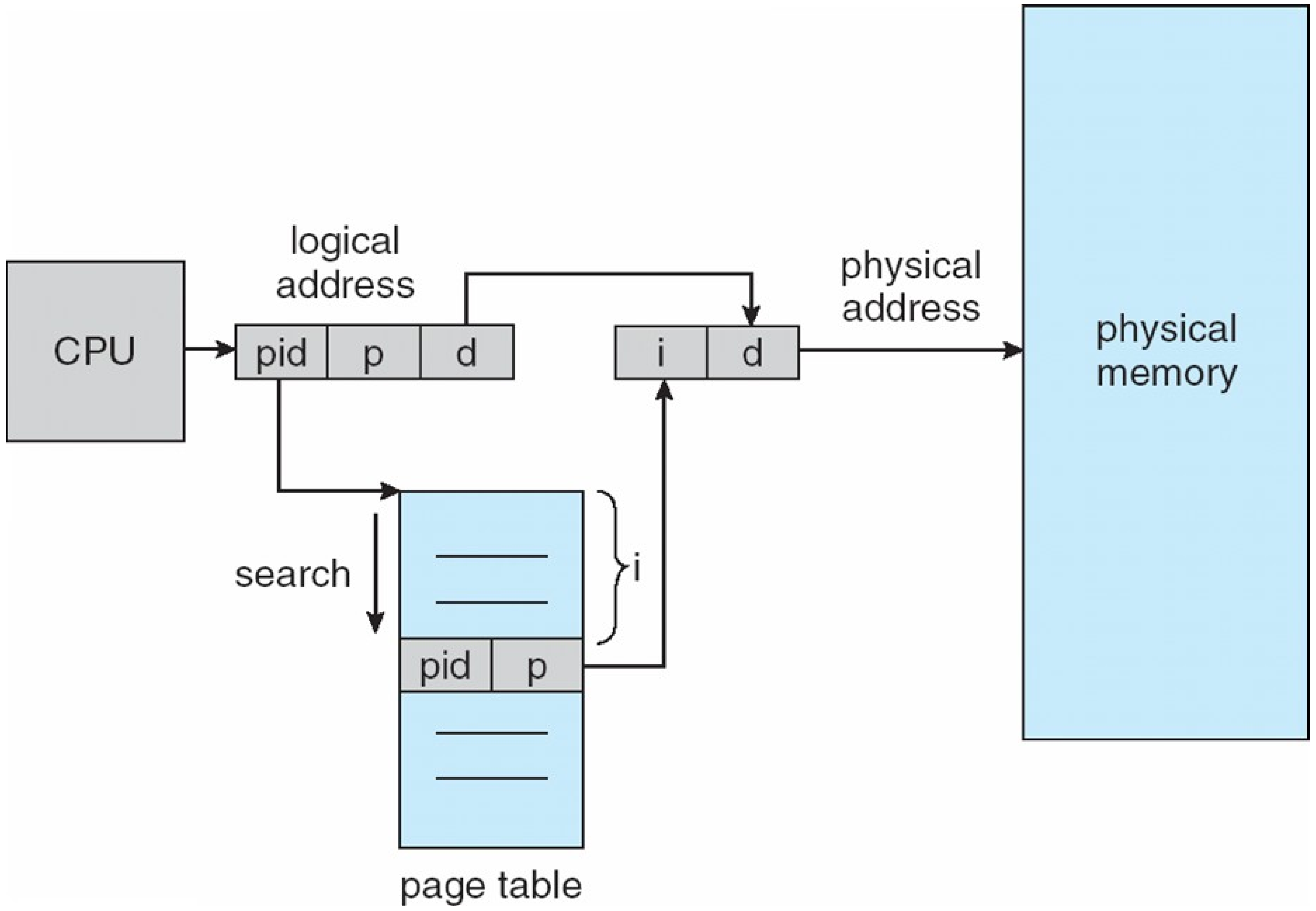


modified by Stewart Weiss

Inverted Page Table

- 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
- Very difficult to implement shared memory

Inverted Page Table Architecture



modified by Stewart Weiss

Segmentation

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

    main program

    procedure

    function

    method

    object

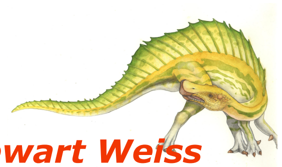
    local variables, global variables

    common block

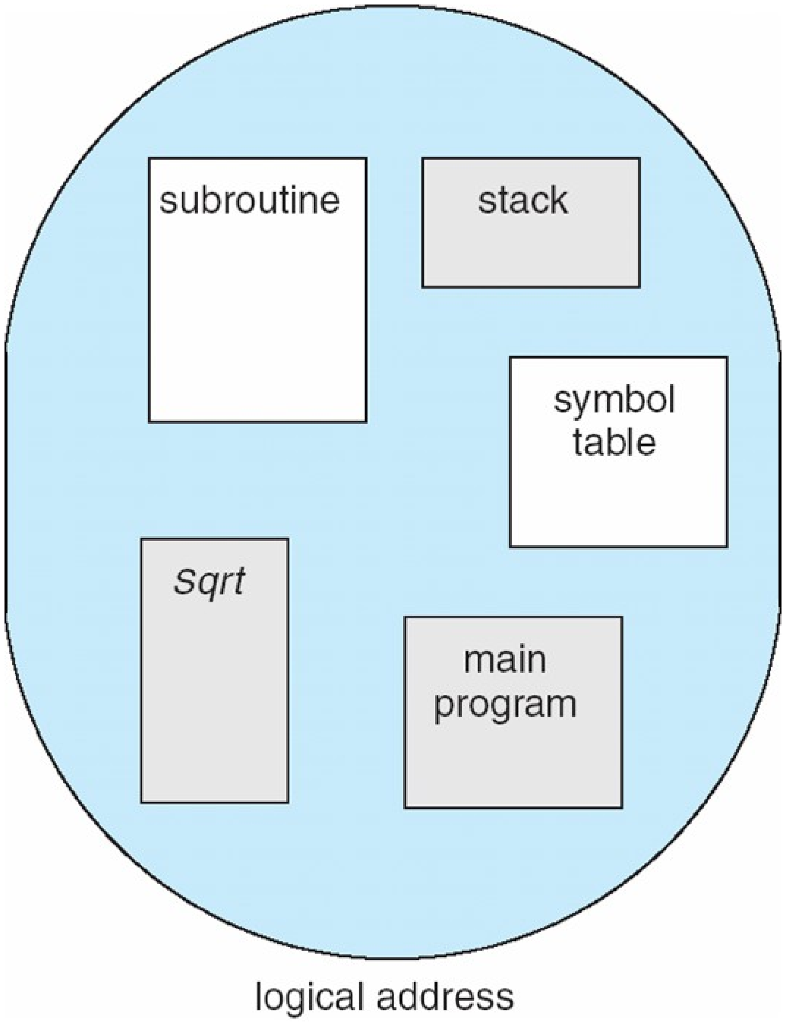
    stack

    symbol table

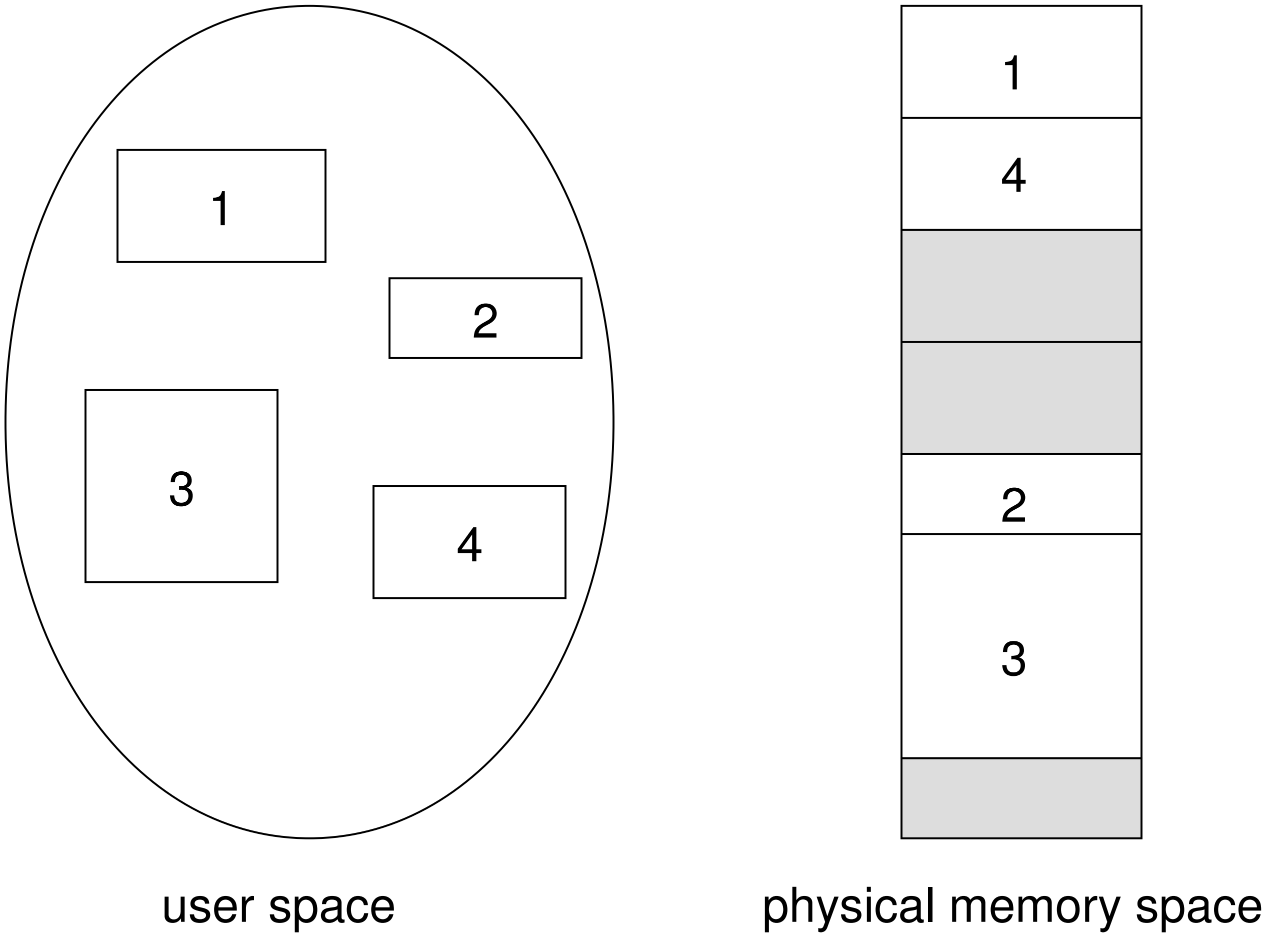
    arrays



User's View of a Program



Logical View of Segmentation



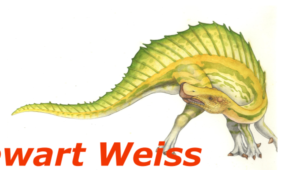
Segmentation Architecture

- 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 number of segments used by a program;

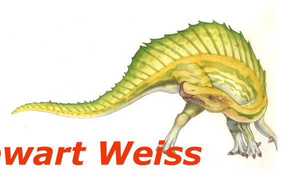
  segment number ***s*** is legal if ***s*** < **STLR**



modified by Stewart Weiss

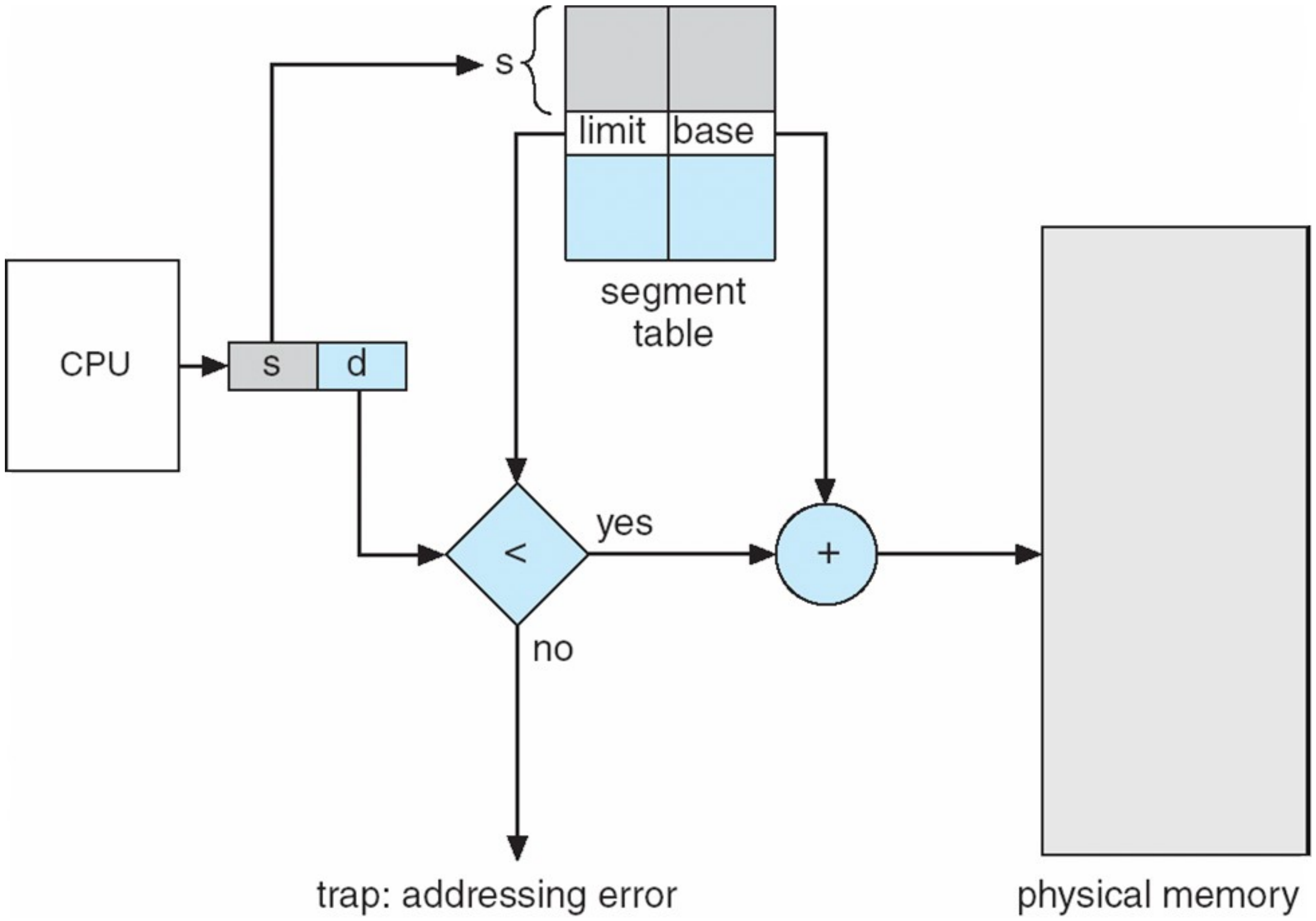
Segmentation Architecture (Cont.)

- 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 is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

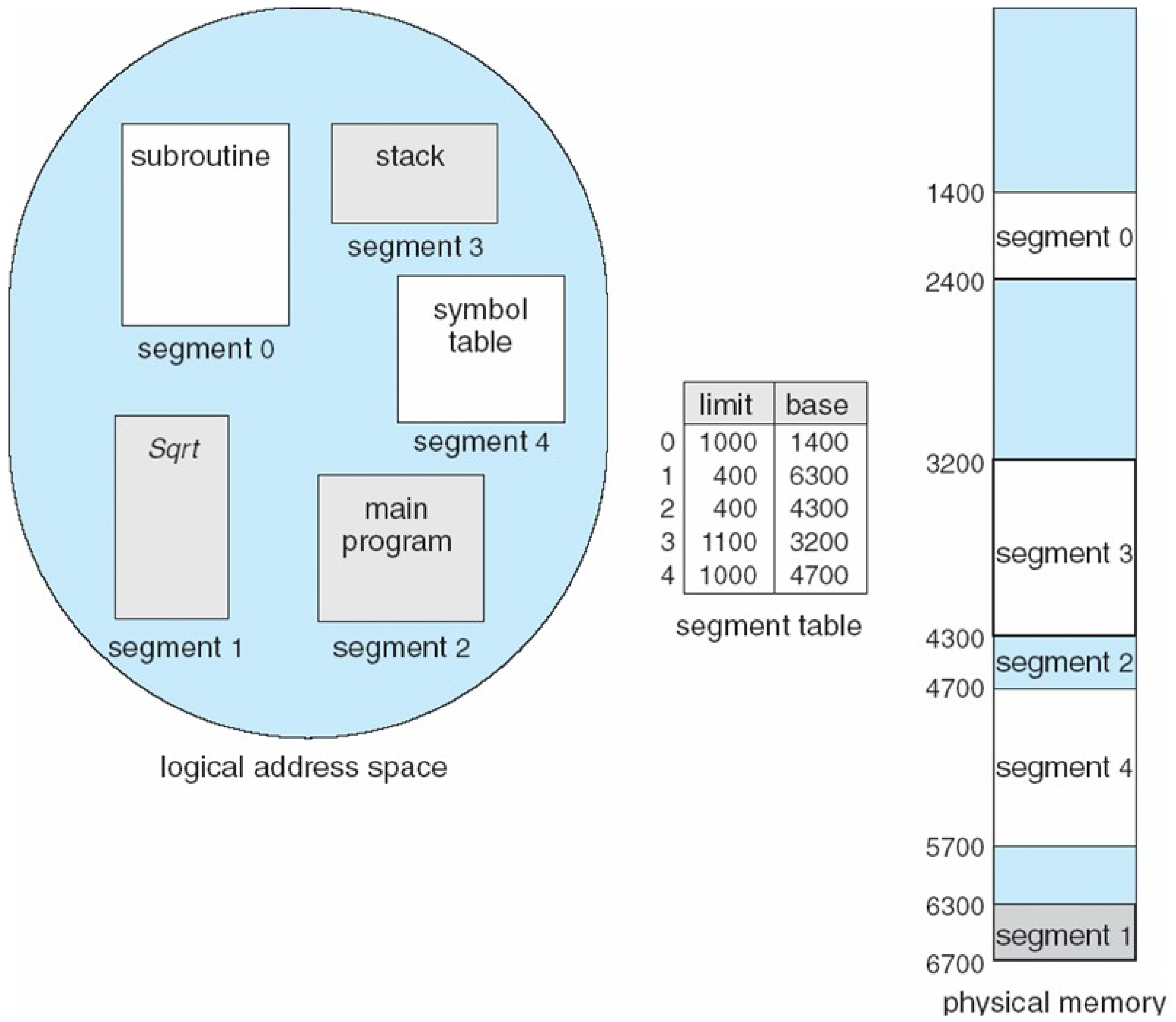


modified by Stewart Weiss

Segmentation Hardware



Example of Segmentation



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

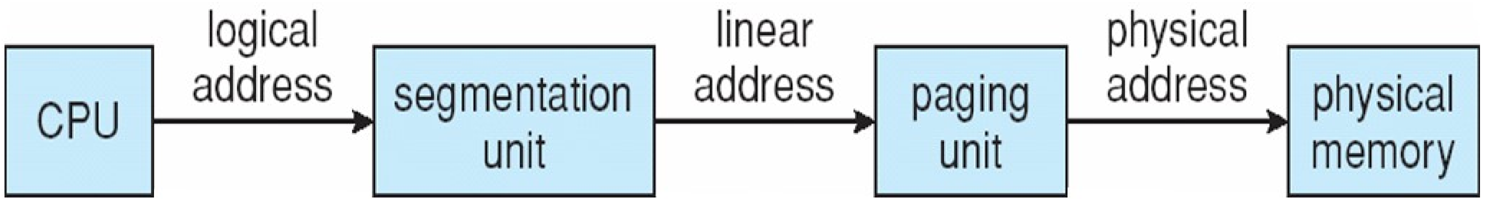
segment table

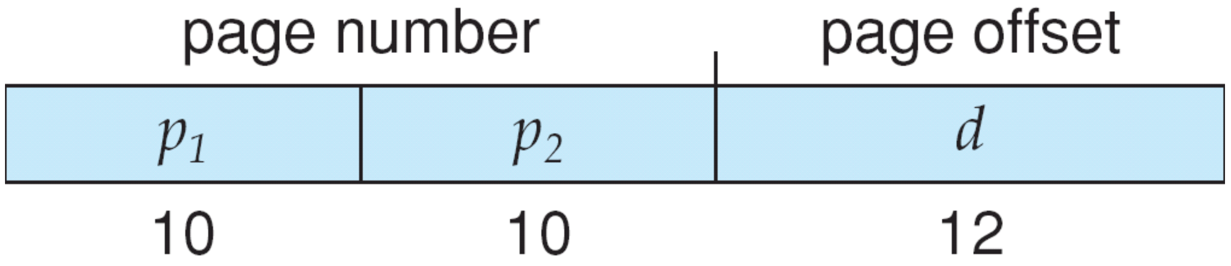
modified by Stewart Weiss

Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - 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

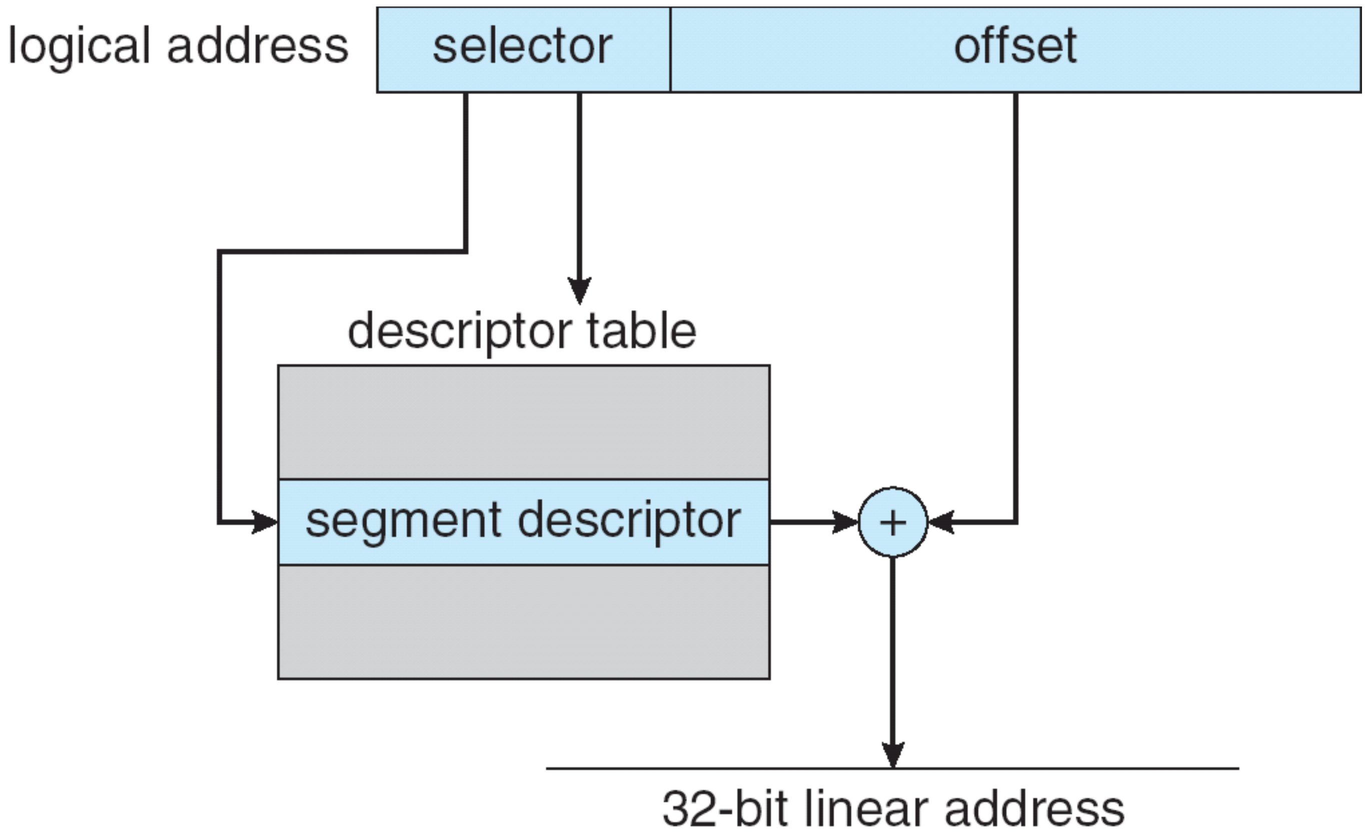
Logical to Physical Address Translation in Pentium



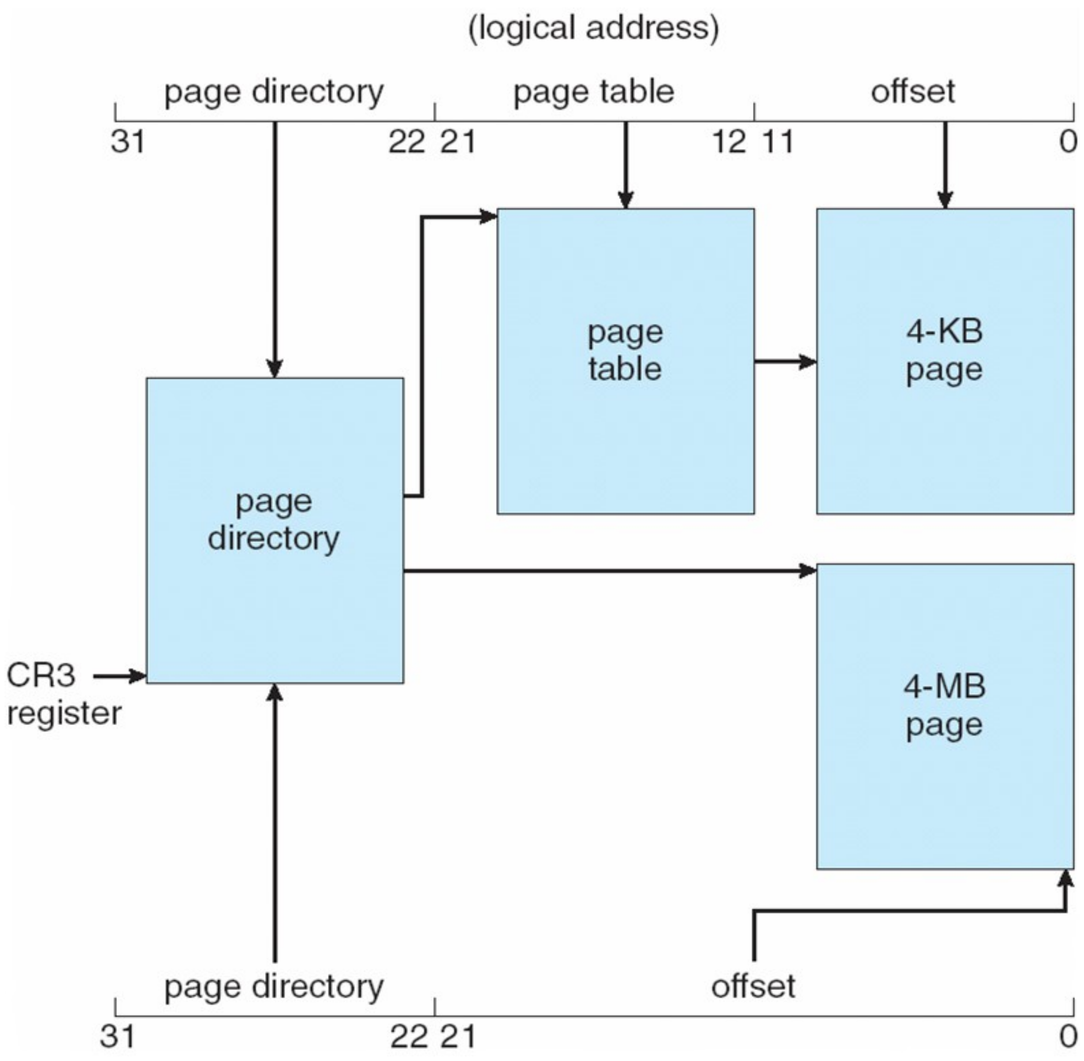


modified by Stewart Weiss

Intel Pentium Segmentation

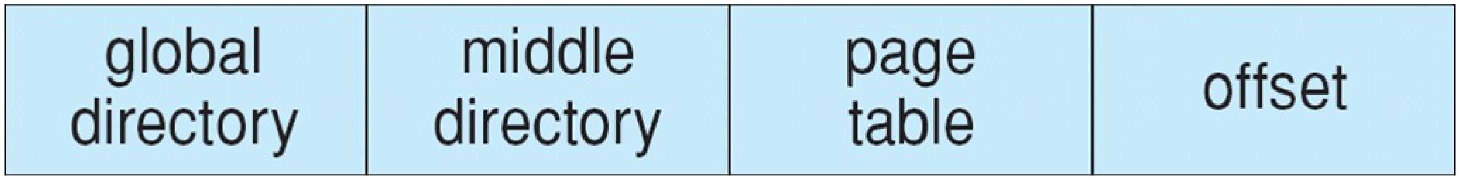


Pentium Paging Architecture

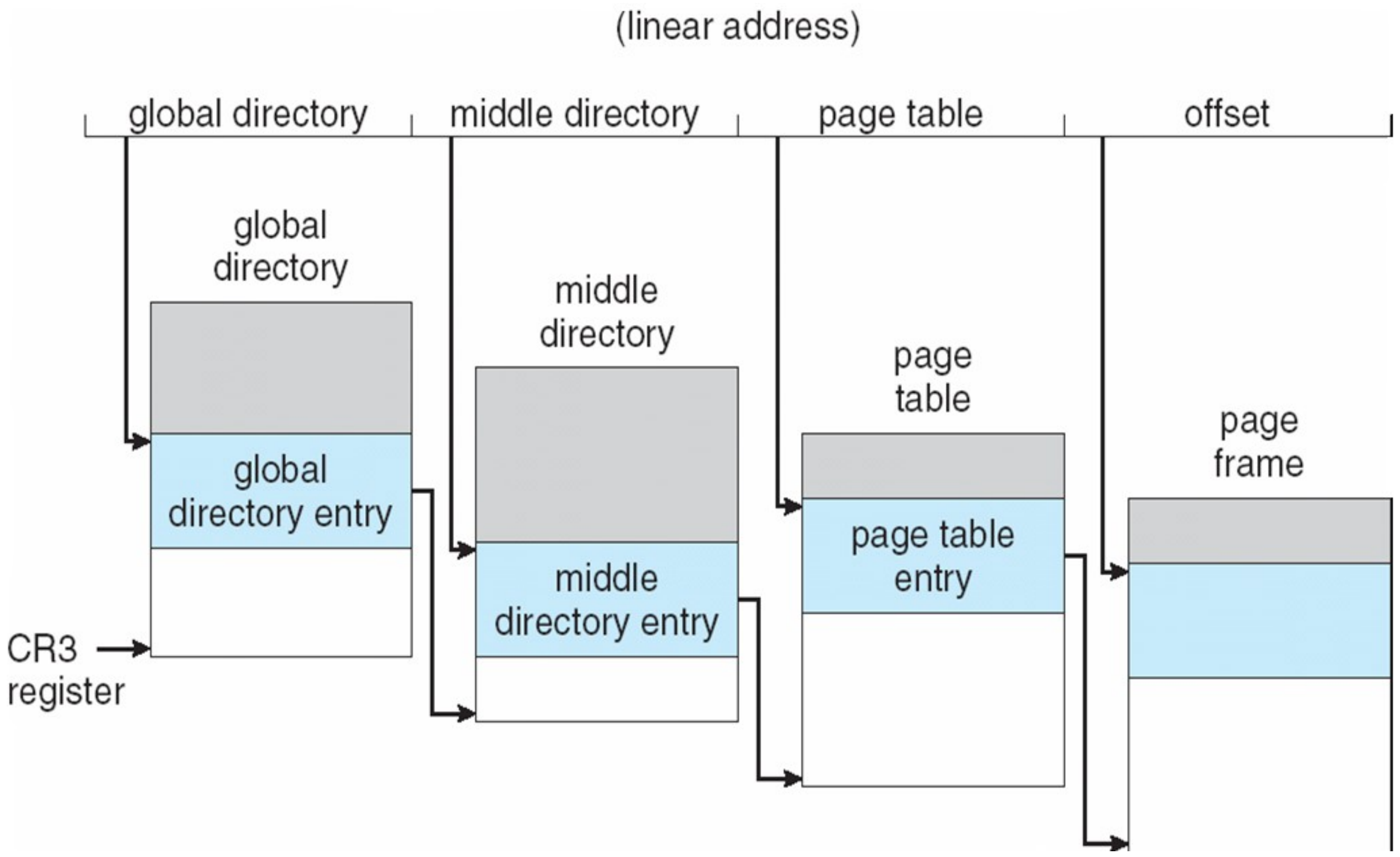


Linear Address in Linux

Broken into four parts:



Three-level Paging in Linux



modified by Stewart Weiss

End of Chapter 8