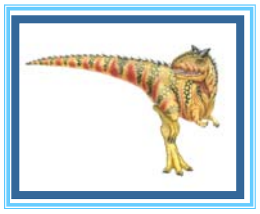
Chapter 8: Main Memory

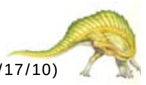


(slides improved by R. Doemer, 05/18/10)

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium

(slide modified by R. Doemer, 05/17/10)



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

(slide modified by R. Doemer, 05/17/10)

Background

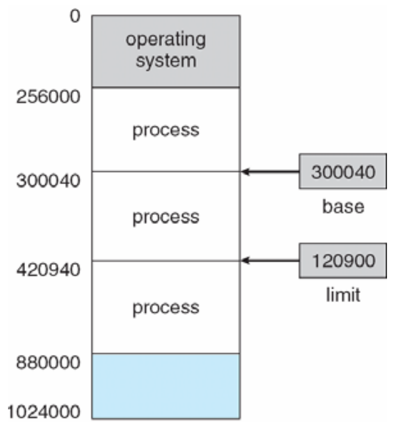
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the only storage the CPU can access directly
- Register access in one CPU clock cycle (or less)
- Main memory access can take many cycles
- **Cache** sits between main memory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

- Protection of memory is required to ensure safe cooperation of processes

(slide modified by R. Doemer, 05/17/10)

Base and Limit Registers

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

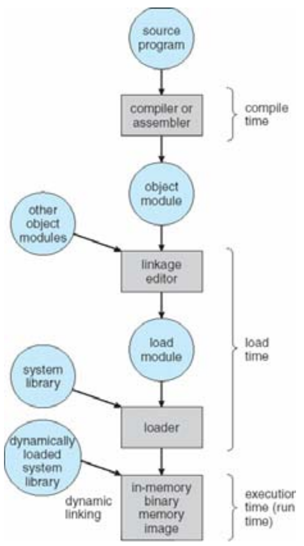


Binding of Instructions and Data to Memory

- **Address binding** of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location is known a priori, *compiler* can generate **absolute code**; must recompile code if starting location changes
  - **Load time**: Compiler must generate **relocatable code** if memory location is not known at compile time; *Loader* completes address binding
  - **Execution time**: Address binding can be delayed until run time if the process can be moved during its execution from one memory segment to another; need hardware support in CPU for address mapping (e.g., base and limit registers); *Memory Management Unit* in CPU determines address binding

(slide modified by R. Doemer, 05/17/10)

Multi-Step Processing of a User Program



Compile time

Link time

Load time

Run time

(slide modified by R. Doemer, 05/17/10)

Dynamic Linking

- **Dynamic Linking**: Linking is postponed until execution time
- Dynamic linking is also known as **shared libraries**

- A small piece of code, a *stub routine*, is used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and then executes the routine
- Operating system needed to check if routine is in the processes' memory address
- Dynamic linking is particularly useful for libraries (which then can be shared by multiple processes)

(slide modified by R. Doemer, 05/17/10)

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

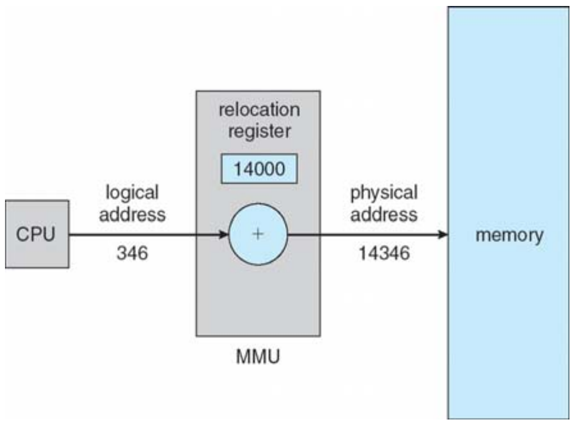
(slide modified by R. Doemer, 05/17/10)

Memory-Management Unit (MMU)

- Memory-Management Unit (MMU):
  Hardware device that maps virtual to physical address
- In MMU scheme, the value in a **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

(slide modified by R. Doemer, 05/17/10)

Conceptual MMU with a Relocation Register



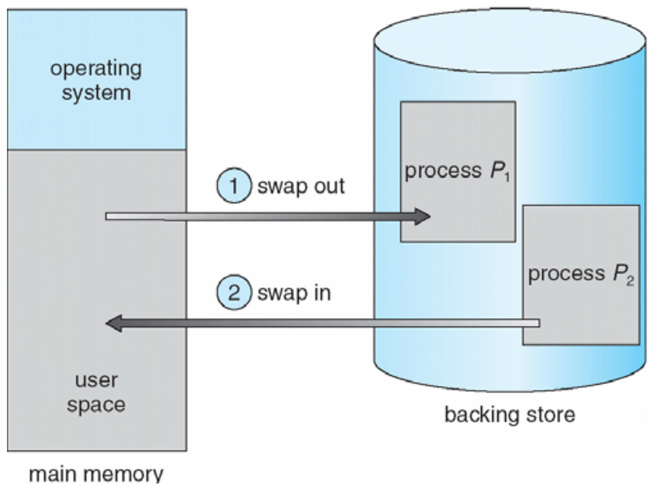
(slide modified by R. Doemer, 05/17/10)

Swapping

- **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 processes;
  must provide direct access to these memory images
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Major part of **swap time** is transfer time;
  transfer time is directly proportional to the amount of memory swapped
- **Roll out, roll in** – swapping variant used for priority-based scheduling;
  lower-priority process is swapped out so that a higher-priority process can be loaded and executed
- Modified versions of swapping are found on many systems
  (i.e., UNIX, Linux, and Windows)

(slide modified by R. Doemer, 05/17/10)

Schematic View of Swapping



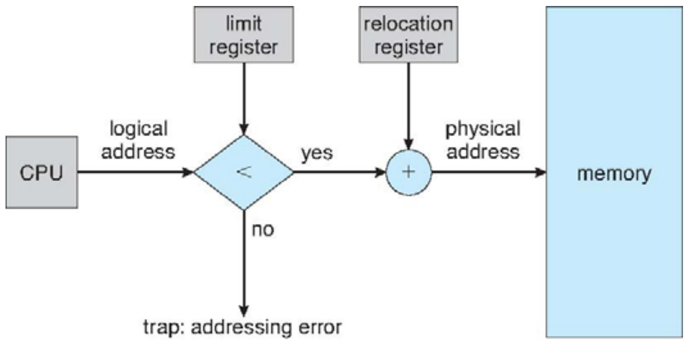
Contiguous Allocation

- 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
- **Relocation registers** are used to protect user processes from each other, and from changing operating-system code and data
  - **Base register** contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address to physical address *dynamically (at run time)*

(slide modified by R. Doemer, 05/17/10)

Contiguous Allocation

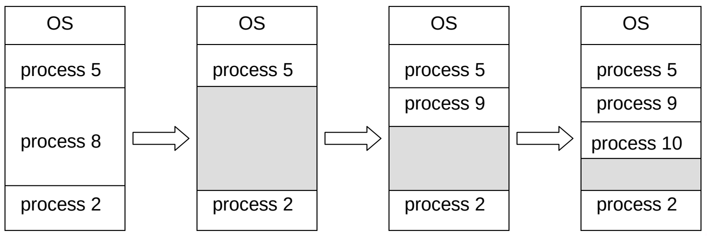
- Hardware Support for Relocation and Limit Registers



(slide modified by R. Doemer, 05/17/10)

Contiguous Allocation

- Multiple-partition allocation
  - **Hole** – block of available memory;
    holes of various sizes 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)



(slide modified by R. Doemer, 05/17/10)

Contiguous Allocation

Dynamic Storage-Allocation Problem:
How to satisfy a request of size *n* from a list of free holes

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

First-fit and best-fit are usually better than worst-fit
in terms of speed and storage utilization.

(slide modified by R. Doemer, 05/17/10)

Memory Fragmentation

- **Internal Fragmentation** –
  allocated memory is often slightly larger than requested memory
  (e.g., 64 bytes allocated for a request of 55 bytes); this size difference
  is *internal* to a memory partition, but is not being used
- **External Fragmentation** –
  many small holes exist between allocated memory partitions;
  total memory space is available for a request, but it is not contiguous
- External fragmentation can be reduced by **compaction**
  - Relocate 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 relocate process while it is involved in I/O
    - Do I/O only into OS buffers

(slide modified by R. Doemer, 05/18/10)

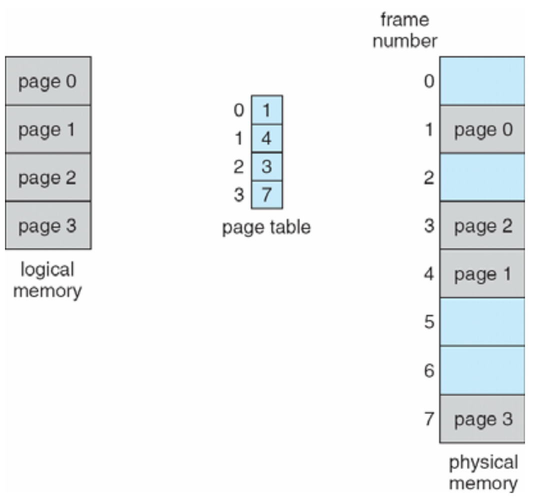
Paging

- **Paging** avoids external fragmentation (and compaction) entirely
  - Internal fragmentation problem remains
- Divide **logical memory** into blocks of same size called **pages** (size is power of 2, typically between 512 bytes and 8,192 bytes)
- Divide **physical memory** into fixed-sized blocks called **frames** (size of a frame is the same as page size)
- Set up a **page table** to translate logical to physical addresses
- Keep track of all free frames

- Then, logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- To run a program of size *n* pages, find *n* free frames and load the program

(slide modified by R. Doemer, 05/18/10)

Paging: Logical and Physical Memory



(slide modified by R. Doemer, 05/18/10)

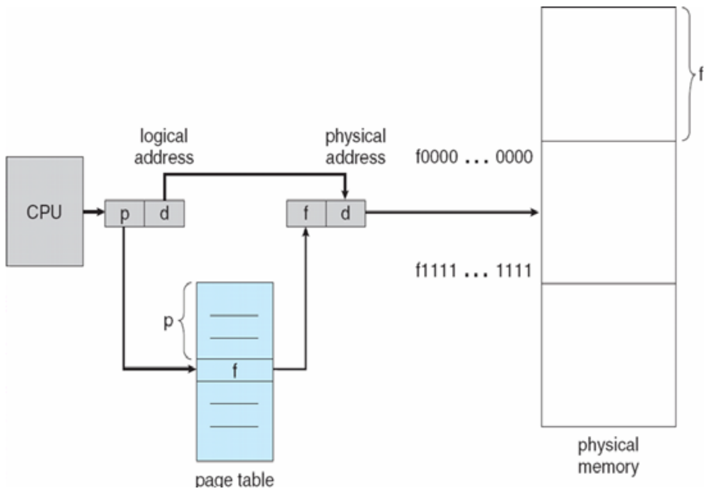
Paging: Address Translation Scheme

- **Logical address** generated by CPU is divided into:
  - **Page number (*p*)** –
    used as an index into a *page table*
    which contains the base address of each page in physical memory
  - **Page offset (d)** –
    is combined with base address to define the physical memory address
    that is sent to the memory unit
- Example for a given logical address space of 2^m *and a page size of* 2^n

page number	page offset
p	*d*
m - n	*n*

(slide modified by R. Doemer, 05/18/10)

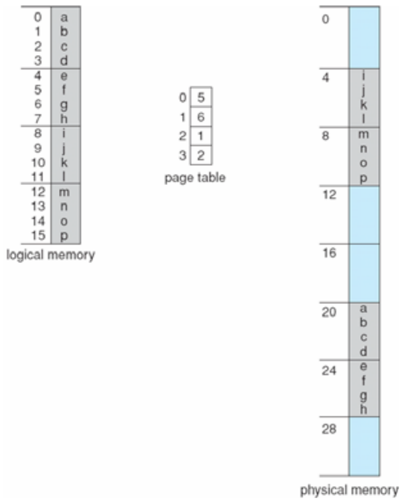
Paging Hardware



Paging: Tiny Example

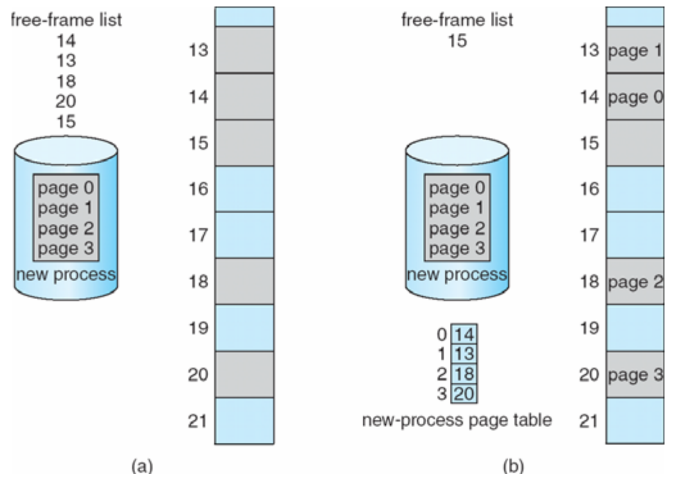
Assume a 32-byte memory with 4-byte pages

- 5-bit address space, divided into
- 3 bits for page number, and
- 2 bits for page offset



(slide modified by R. Doemer, 05/18/10)

Paging: Allocation, Free Frames



Before process allocation After process allocation

(slide modified by R. Doemer, 05/18/10)

Paging: Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** in CPU points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme, *every* data and instruction access requires *two* memory accesses:
  - one access two the page table, and
  - one access for the actual data/instruction.
- Unless treated, this results in running all programs at half the speed!

(slide modified by R. Doemer, 05/18/10)

Paging: Translation Look-aside Buffer

- **Translation Look-aside Buffer (TLB)**
  - a special fast-lookup hardware cache
  - solves the two memory access problem
  - implemented in hardware as an associative memory
- **Associative memory** implements *parallel search*

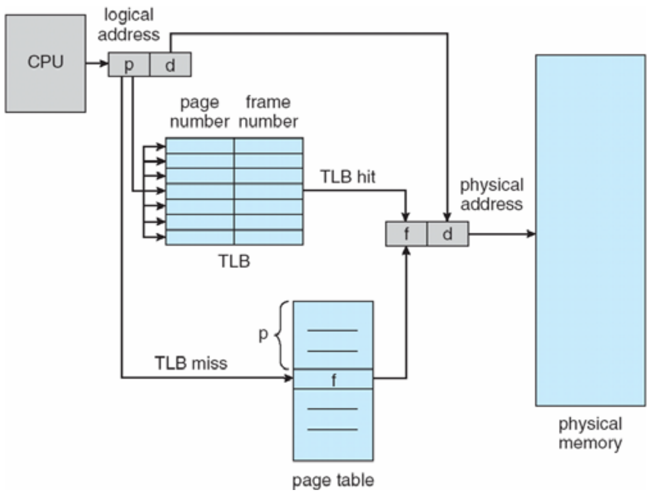
Page #	Frame #
p_1	f_1
p_2	f_2
p_3	f_3
p_4	f_4

Address translation from logical address (p, d) to physical address (f, d)

- If page p_i is in associative memory, get frame number f_i out
- Otherwise, get frame number f from page table in memory and update TLB

(slide modified by R. Doemer, 05/18/10)

Paging Hardware With TLB



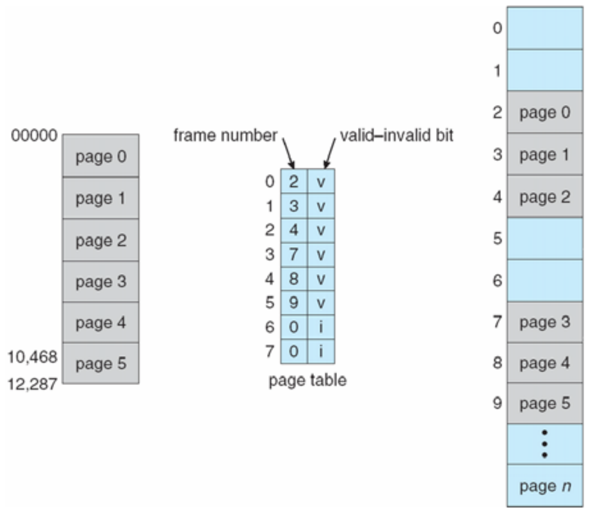
(slide modified by R. Doemer, 05/18/10)

Paging: Memory Protection

- **Memory protection** can be easily implemented in paging scheme by associating a set of **protection bits** with each frame
- **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
- **Read, write, execute bits** control valid access types to pages
  - Write access may be denied to shared libraries
  - Execute access may be denied to data and stack memory (quite effective for virus protection!)
  - etc.

(slide modified by R. Doemer, 05/18/10)

Paging: Valid/Invalid Bit In Page Table



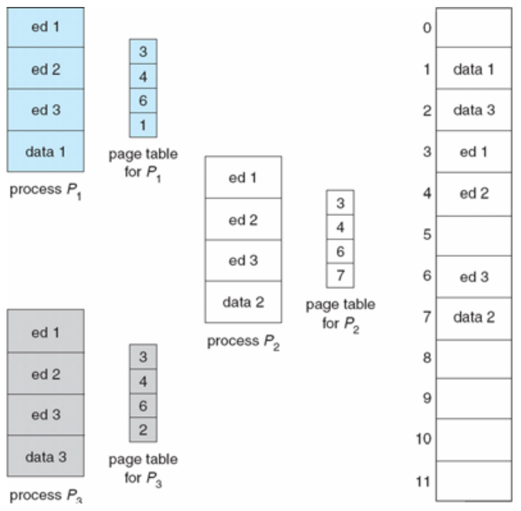
(slide modified by R. Doemer, 05/18/10)

Paging: Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, GUI 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

(slide modified by R. Doemer, 05/18/10)

Paging: Shared Pages Example



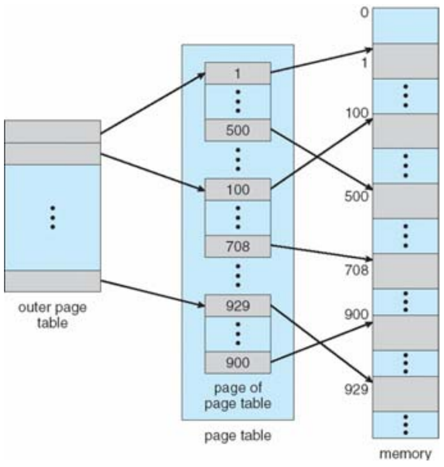
(slide modified by R. Doemer, 05/18/10)

Structure of the Page Table

- 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



(slide modified by R. Doemer, 05/18/10)

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 2x10 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 20-bit page number (10 bits for level 1, 10 bits for level 2)
  - a 12-bit page offset
- Thus, a logical address is composed as follows:

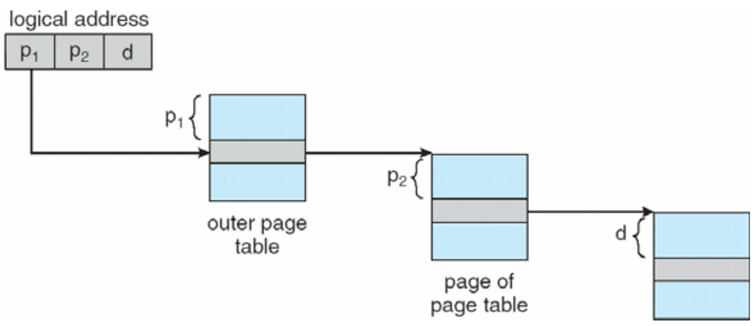
page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table,
and p_2 is an index into the inner page table,
and d the displacement within the page

(slide modified by R. Doemer, 05/18/10)

Two-Level Paging Example

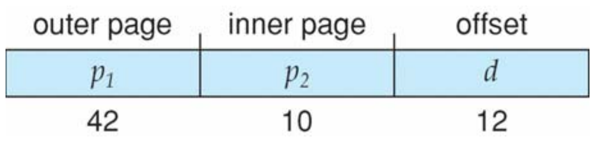
- Address-Translation Scheme



(slide modified by R. Doemer, 05/18/10)

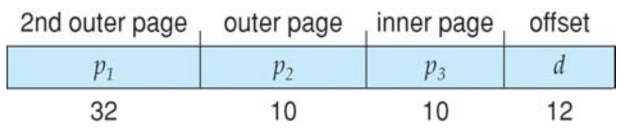
Multi-level Paging Scheme

- For 64-bit machines, 2-level paging is no longer appropriate
  - For 4K pages, the outer page table would contain 2^{42} x 4 bytes!



outer page	inner page	offset
p_1	p_2	d
42	10	12

  - Using 3 levels of paging, the 2nd outer page is still daunting with 2^{34} bytes!



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

  - Thus, 4 or more levels would be needed…

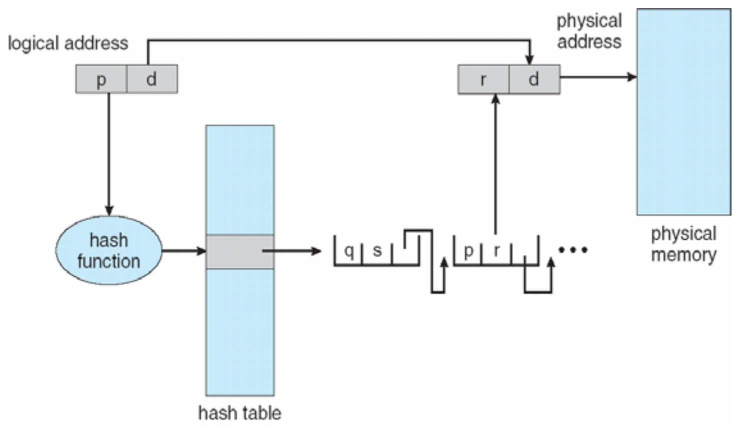
(slide modified by R. Doemer, 05/18/10)

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

(slide modified by R. Doemer, 05/18/10)

Hashed Page Tables



(slide modified by R. Doemer, 05/18/10)

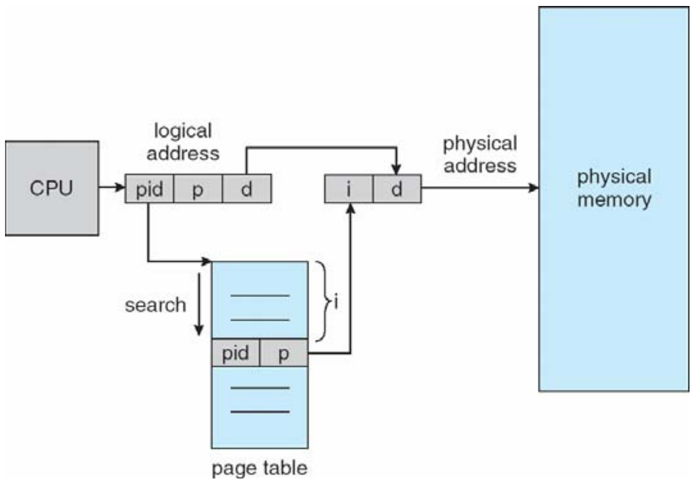
Inverted Page Tables

- Usually, every process has its own associated page table (which may consume a large amount of memory space)

- **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 the 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

(slide modified by R. Doemer, 05/18/10)

Inverted Page Tables



(slide modified by R. Doemer, 05/18/10)

Segmentation

- **Segmentation** is an alternative to paging
- Memory-management scheme that supports *user view* of memory
- A program is a collection of **segments**
  - In the programmer's view, a segment is a logical unit such as:

    main program

    procedure / function / method

    object

    local variables, global variables

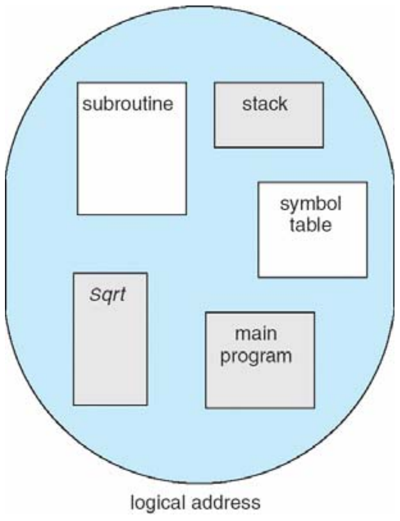
    shared memory block

    stack

    symbol table

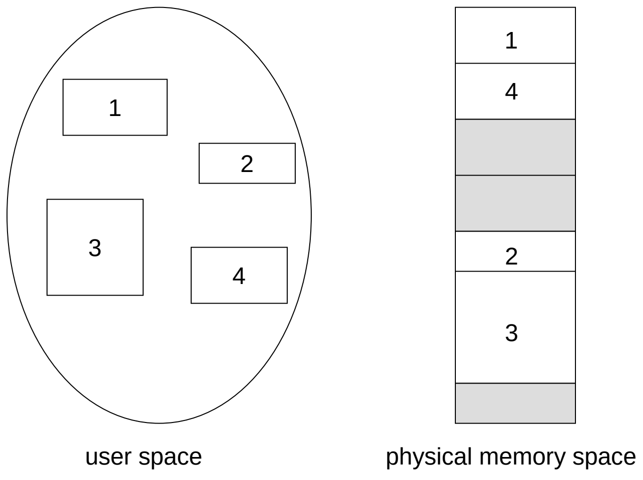
(slide modified by R. Doemer, 05/18/10)

Segmentation: Programmer's View of a Program



(slide modified by R. Doemer, 05/18/10)

Logical View of Segmentation



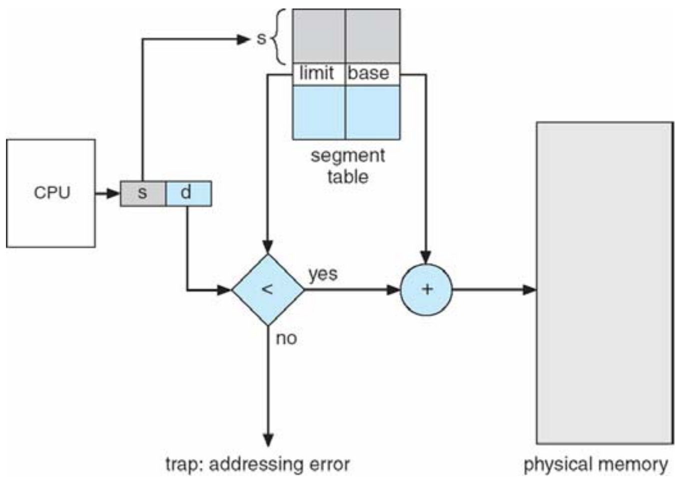
Segmentation Architecture

- Logical address consists of a tuple:

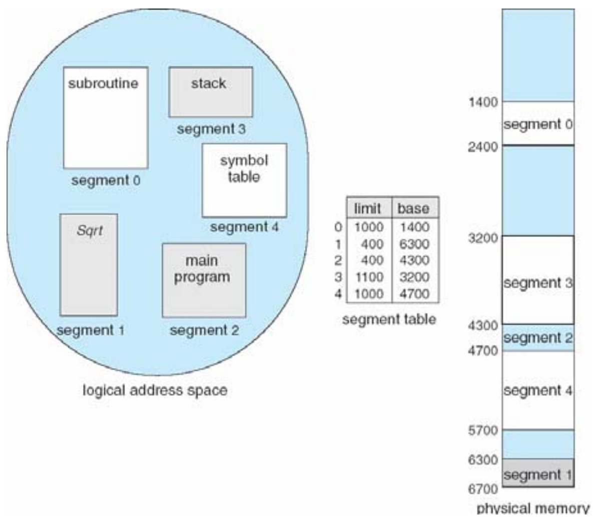
  <segment-number, offset>,
- **Segment table** – maps segment-number to physical address
- Each table entry has:
  - **Base** –
    starting physical address where the segment resides in memory
  - **Limit** –
    length of the segment
- **Segment-table base register (STBR)**
  points to the segment table’s location in memory
- **Segment-table length register (STLR)**
  indicates number of segments used by a program
  - segment number *s* is legal if *s* < STLR

(slide modified by R. Doemer, 05/18/10)

Segmentation Hardware



Example of Segmentation



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

Segmentation: Sharing and Protection

- Protection
  - Similar to protection bits in paging scheme
  - With each entry in segment table, associate:
    - validation bit, if 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Code and data sharing can occur naturally at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

(slide modified by R. Doemer, 05/18/10)

End of Chapter 8