Chapter 8 : Main Memory

The picture can't be displayed.

1

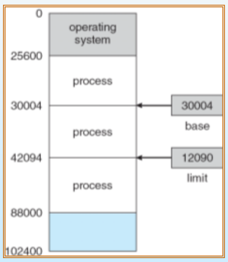
Background

- Program 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

2

Base and Limit Registers

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



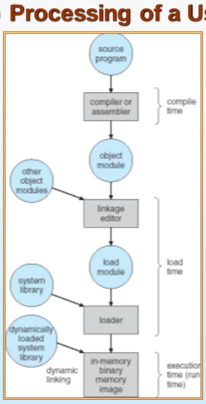
3

Binding of Instructions and Data to Memory

- Address binding (instructions/data) to memory addresses can happen at:
  - **Compile time**: memory location known a priori, **absolute code** generated; must recompile if starting location changes
  - **Load time**: generate **relocatable code** if memory location not known at compile time
  - **Execution time**: **binding** delayed until **run time**. Process can be moved during execution from one memory segment to another. Need **hardware support** for address maps (e.g., base and limit registers)

4

Multistep Processing of a User Program



5

Logical vs. Physical Address Space

- Concept: **logical address space** bound to separate **physical address space**
  - **Logical address** – generated by the CPU (also referred to as **virtual** address)
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses:
  - **same** in compile-time/load-time address-binding schemes;
  - **different** in run-time address-binding scheme

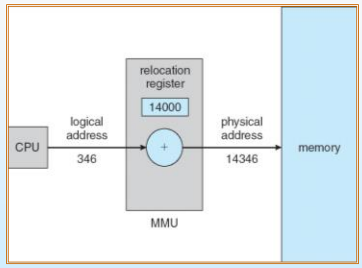
6

Memory-Management Unit (MMU)

- **Hardware** device that maps virtual to physical address
- Value in **relocation register** added to every address generated by a user process before sending to memory
- User program deals with ***logical*** addresses; it never sees the *real* physical addresses

7

Dynamic relocation using a relocation register



8

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 **infrequently**
- No special support from the operating system is required; implemented through **program design**

9

Dynamic Linking

- **Linking** postponed until **run-time**
- Small piece of code, *stub*, **locate** the appropriate library routine (memory-resident)
- Stub **replaces itself** with the address of the routine, and executes the routine
- OS: **check** if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

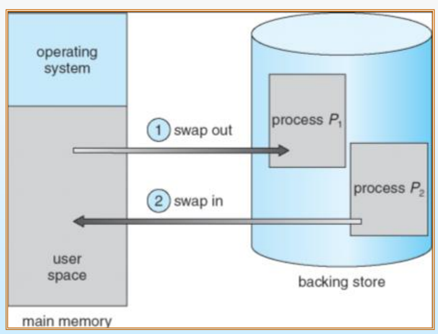
10

Swapping

- Process swapped temporarily from memory to **backing store**, then back into memory for continued execution
- **Fast disk;** large to accommodate copies of **all memory images** for all users; direct access to these memory images
- **Roll out, roll in** –variant used for **priority-based** scheduling (low-priority process swapped out; high-priority executed)
- **Transfer time:** directly proportional to amount of memory swapped
- Modified versions found on many systems (UNIX, Linux, Windows)
- System maintains a **ready queue** (swapped) of processes with memory images on disk

11

Schematic View of Swapping



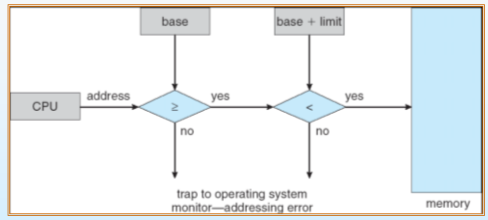
12

Contiguous Allocation

- Main memory divided into **two partitions**:
  - Resident **OS**, 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 also OS 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*

13

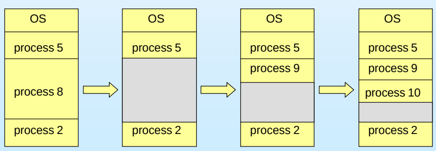
HW address protection with base and limit registers



14

Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - **Hole** – block of available memory; holes of various size scattered throughout memory
  - New process: memory **allocated from a hole** large enough to accommodate it
  - OS maintains information about:
    a) **allocated** partitions b) **free** partitions (hole)



15

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 better than worst-fit in terms of speed and storage utilization

16

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but *not contiguous*
- **Internal Fragmentation** – allocated memory slightly larger than requested memory; size difference is memory *internal to a partition*, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents; place free memory together in *one large block*
  - Compaction possible *only* if relocation is dynamic; done at runtime
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

17

Paging

- *Logical* address space of a process can be **noncontiguous**; process allocated physical memory whenever 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 ***n*** pages, find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

18

Address Translation Scheme

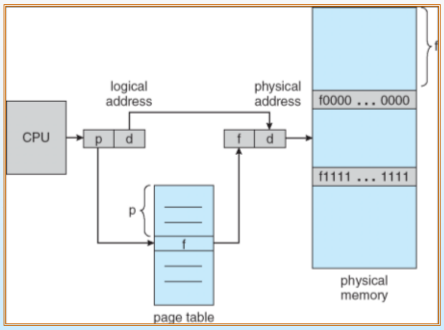
- Address generated by CPU is divided into:
  - **Page number (*p*)** – index into *page table.* Contains base address of *each page* in physical memory
  - **Page offset (d)** – combined with base address to define physical memory address sent to the memory unit

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

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

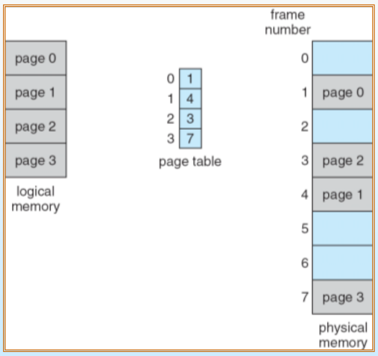
19

Paging Hardware



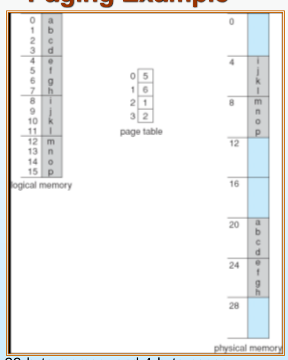
20

Paging Model of Logical and Physical Memory



21

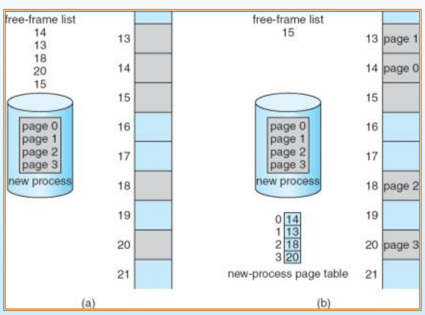
Paging Example



32-byte memory and 4-byte pages

22

Free Frames



Before allocation After allocation

23

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: two memory accesses. One for the page table and one for the data/instruction.
- Two memory access problem can be solved using 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

24

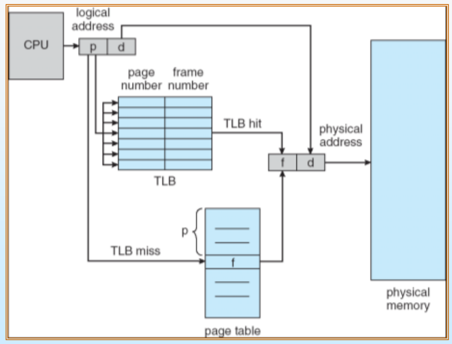
Associative Memory

- 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

Paging Hardware With TLB



Effective Access Time

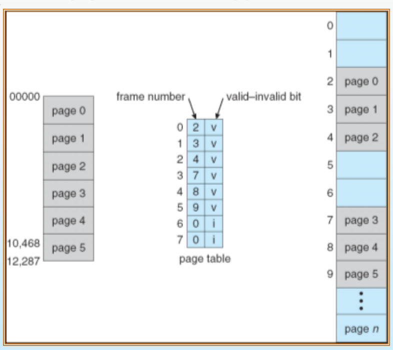
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number found in associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

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

Memory Protection

- Implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid": associated page is in the process' logical address space (legal page)
  - "invalid": page is not in the process' logical address space

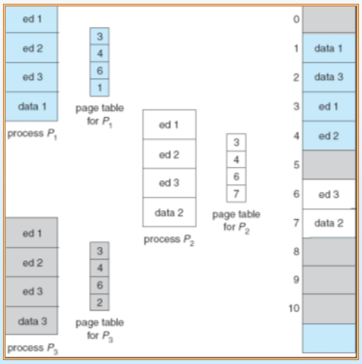
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - *One copy* of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems).
  - 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
  - Pages for the private code and data can appear *anywhere* in the logical address space

Shared Pages Example



31

Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

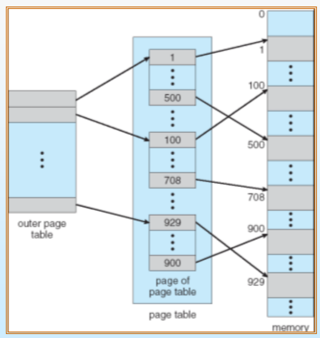
32

Hierarchical Page Tables

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

33

Two-Level Page-Table Scheme



34

Two-Level Paging Example

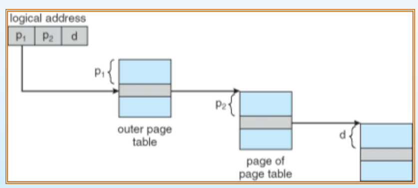
- Logical address (on 32-bit machine with 1K page size) divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is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

page number		page offset
p_i	p_2	d
12	10	10

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

35

Address-Translation Scheme



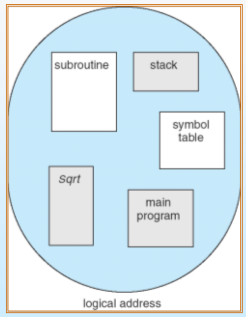
36

Segmentation

- Memory-management scheme that supports user view of memory
- Program: collection of segments. Segment: a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

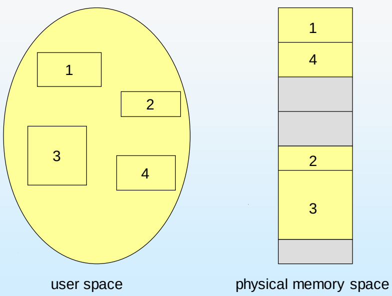
42

User's View of a Program



43

Logical View of Segmentation



44

Segmentation Architecture

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

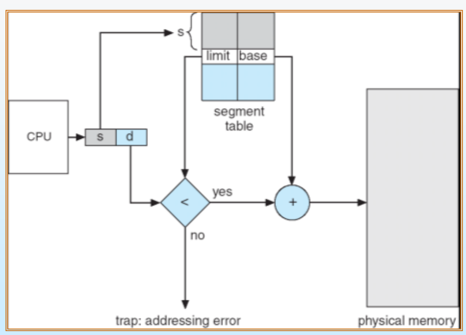
45

Segmentation Architecture (Cont.)

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

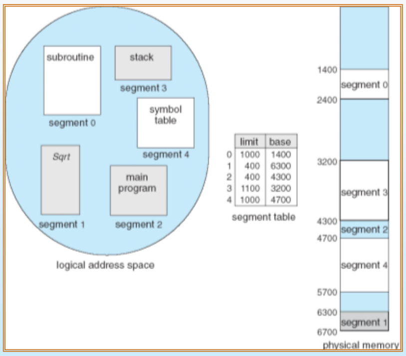
46

Segmentation Hardware



47

Example of Segmentation



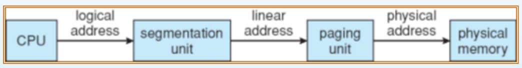
48

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

49

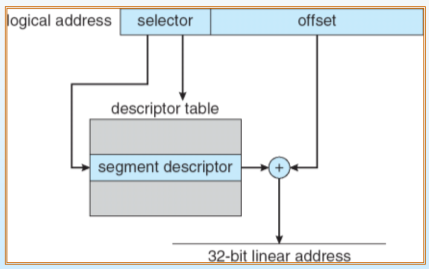
Logical to Physical Address Translation in Pentium



page number		page offset
p_1	p_2	d
10	10	12

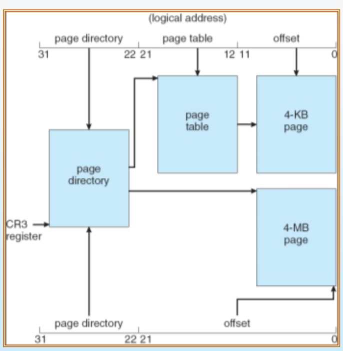
50

Intel Pentium Segmentation



51

Pentium Paging Architecture



52

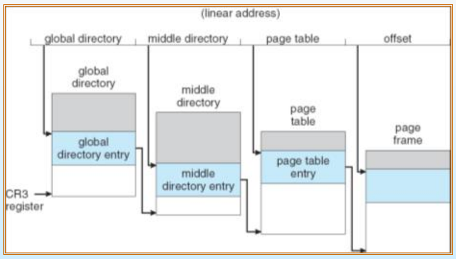
Linear Address in Linux

Broken into four parts:

global directory	middle directory	page table	offset

53

Three-level Paging in Linux



54

The picture can't be displayed.

End of Chapter 8

55