Memory Management

Gordon College

Stephen Brinton

Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Program must be brought into memory and placed within a process for it to be run

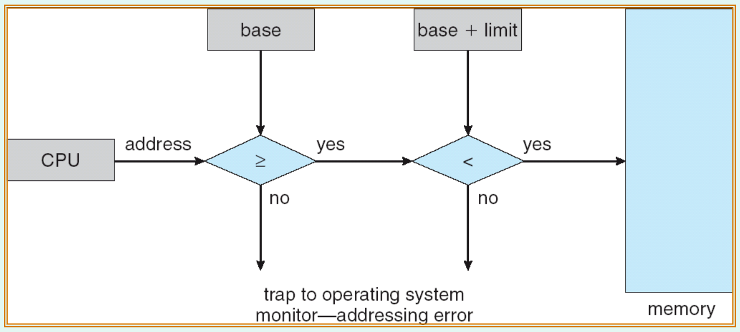
  hardware needs:

  - speed differential (cache)
  - separate memory spaces (base and limit register)

Base Register: 34000
Limit Register: 12000

Address being used by a program must fall within this range or a trap (error) will result.

HW: address protection with base and limit registers



Address Binding

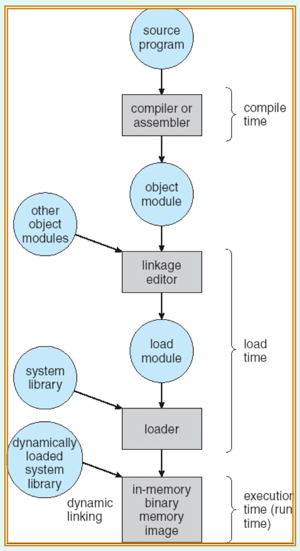
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program
- **Binding** – symbolic addresses in a source program are bound to actually memory addresses.

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 "before execution", *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*).

Multistep Processing of a User Program



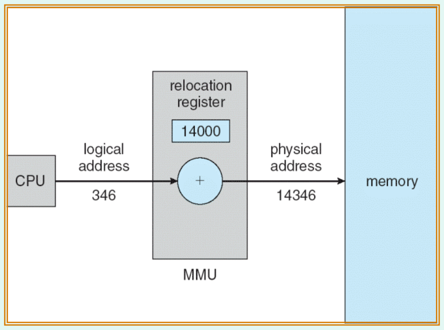
Logical vs. Physical Address Space

- 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
- Note:
  - Logical and physical addresses are the <u>same</u> in compile-time and load-time address-binding schemes;
  - Logical (virtual address) and physical addresses <u>differ</u> 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

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

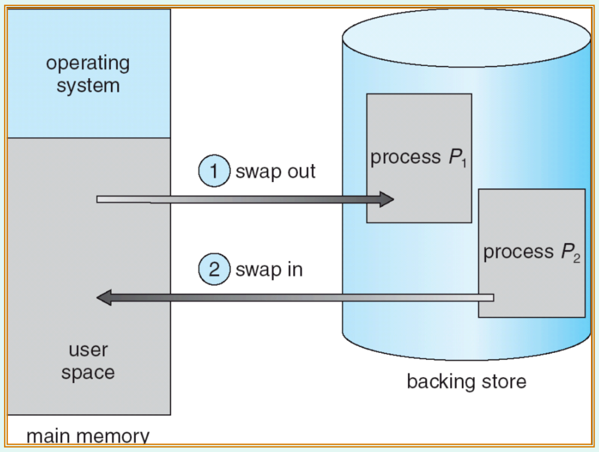
Dynamic Linking

- 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 some process' memory address and allow access to others
- Dynamic linking is particularly useful for libraries
  - Otherwise a program must contain the library code
  - This means wasted disk and memory space

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
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

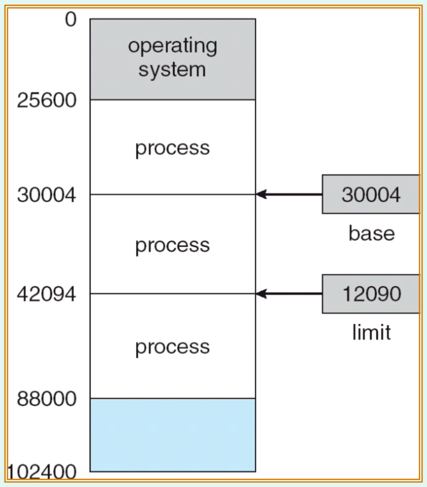
Schematic View of Swapping



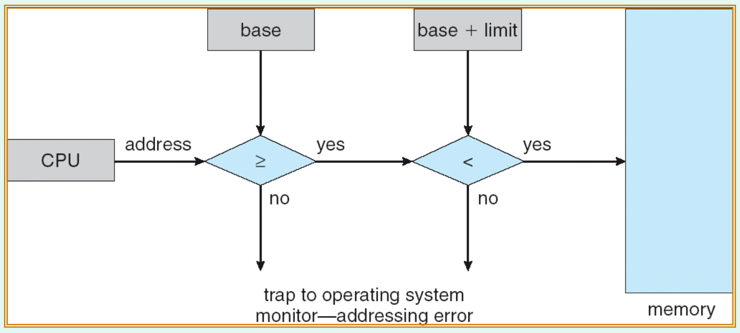
Contiguous Allocation

- Main memory usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Memory mapping and protection
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register

A base and a limit register define a logical address space

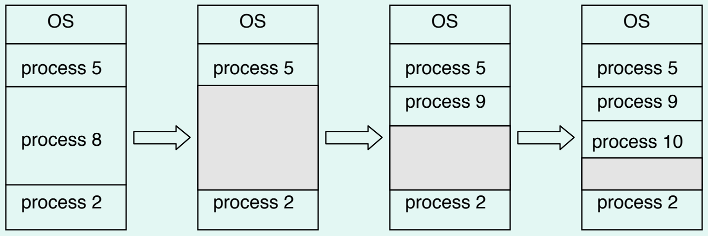


HW address protection with base and limit registers



Contiguous Allocation (Cont.)

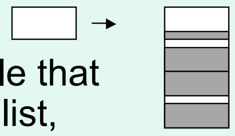
- Multiple-partition allocation
  - *Hole* – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about: a) allocated partitions b) free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



Dynamic Storage-Allocation Problem

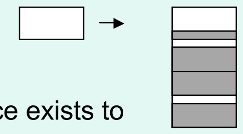
How to satisfy a request of size *n* from a list of free holes

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

Fragmentation

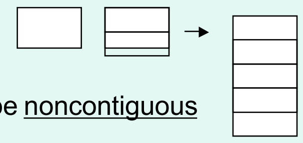


- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

(this is not really a factor when talking about simple contiguous allocation - BUT WAIT, IT IS A PROBLEM WITH PAGING)

- Reduce external fragmentation by **compaction**
  - Goal: 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
  - Another solution to external fragmentation: allow logical space of process to be noncontiguous – paging and segmentation.

Paging



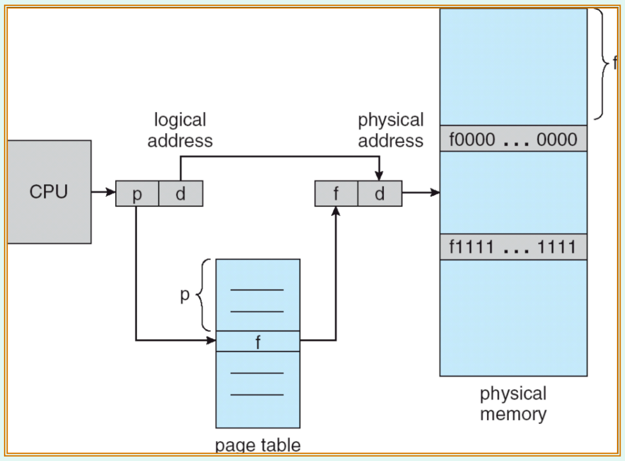
- Logical address space of a process can be noncontiguous
- Basic Method:
  - Divide *physical* memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16MB)
  - Divide *logical* memory into blocks of same size called **pages**.
  - Keep track of all free frames
  - To run a program of size *n* pages, need to find *n* free frames and load program from backing store
  - Set up a page table to translate logical to physical addresses
- Internal fragmentation - left over space within a page

Address Translation Scheme

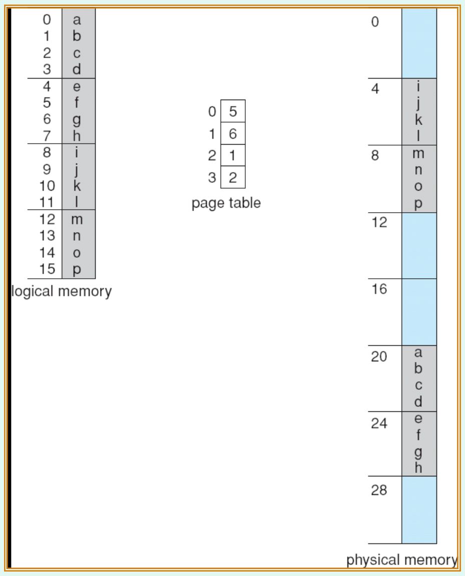
- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a *page table* which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

Page number	Page offset
p	d
m-n	n

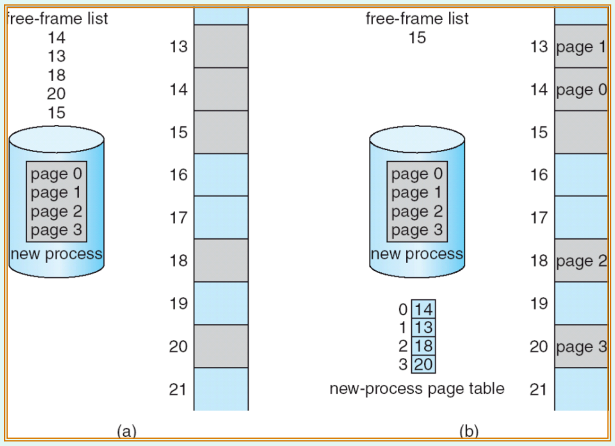
Address Translation Architecture



Paging Example



Free Frames



Before allocation

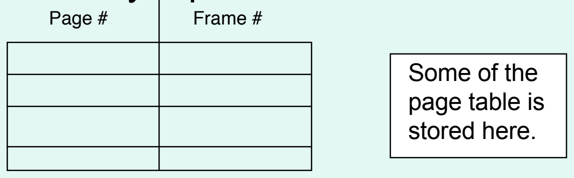
After allocation

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)**

Associative Memory

- Associative memory – parallel search

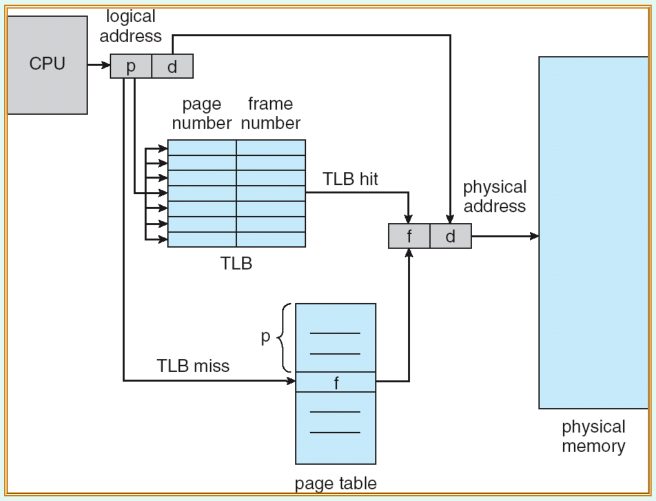


Address translation (A´, A´´)

– If A´ is in associative register, get frame # out : Fast lookup

– TLB miss: 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 (100 ns)
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

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

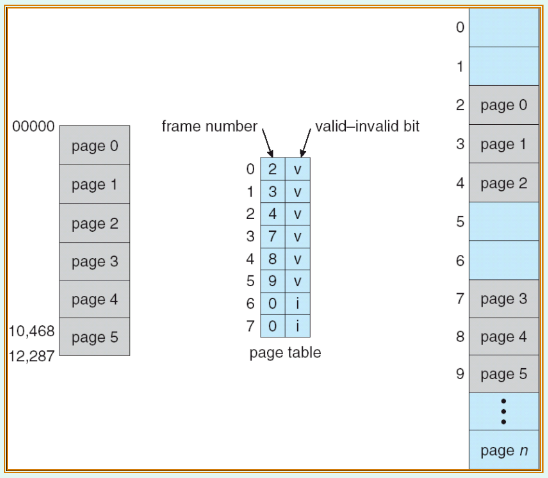
(100 + 20) .8 + (200 + 20)(1-.8)

120 X .80 + 220 * .2 = 140 ns

Memory Protection

- Memory protection implemented by associating protection bit 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

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



Page Table Structure

- 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 <u>two-level page table</u>

Two-Level Paging Example

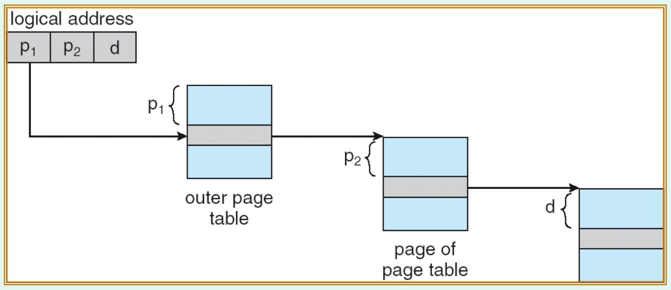
- A *logical address* (on 32-bit machine with 4K page size) is divided into:
  - a <u>page number</u> consisting of 20 bits
  - a <u>page offset</u> consisting of 12 bits
- Since the page table is paged, the <u>page number</u> is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

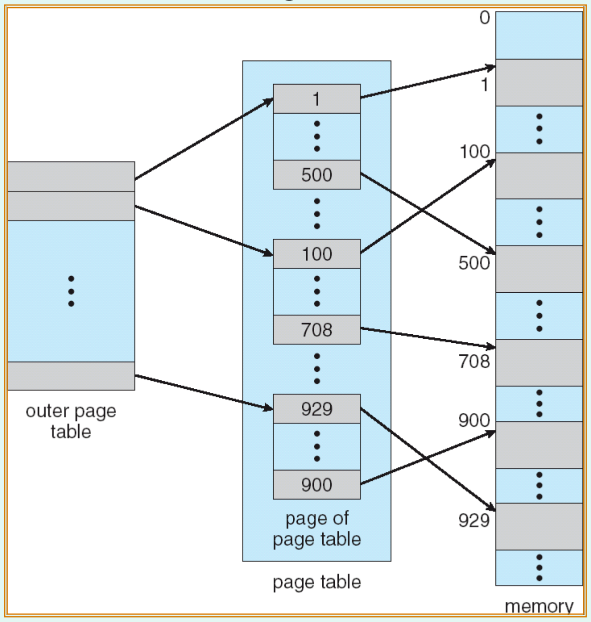
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

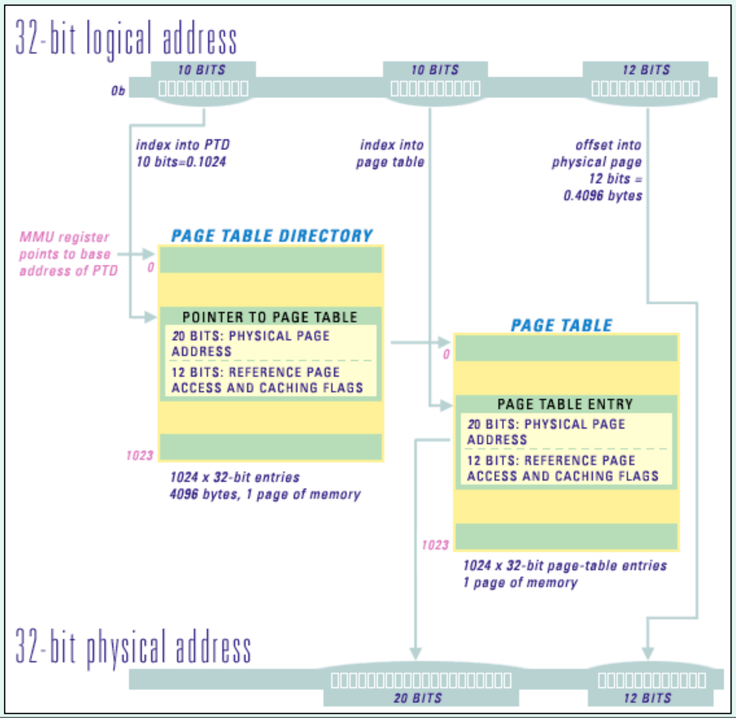
Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture



Two-Level Page-Table Scheme

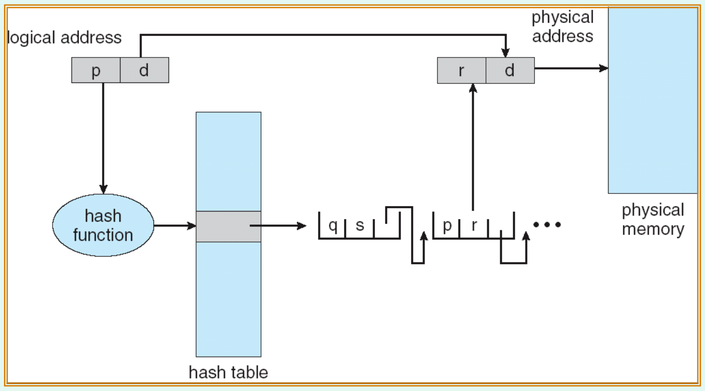



32-bit logical address
10 BITS
10 BITS
12 BITS
0b
index into PTD 10 bits=0.1024
index into page table
offset into physical page 12 bits = 0.4096 bytes
MMU register points to base address of PTD
PAGE TABLE DIRECTORY
0
POINTER TO PAGE TABLE
20 BITS: PHYSICAL PAGE ADDRESS
12 BITS: REFERENCE PAGE ACCESS AND CACHING FLAGS
1023
1024 x 32-bit entries
4096 bytes, 1 page of memory
PAGE TABLE
0
PAGE TABLE ENTRY
20 BITS: PHYSICAL PAGE ADDRESS
12 BITS: REFERENCE PAGE ACCESS AND CACHING FLAGS
1023
1024 x 32-bit page-table entries
1 page of memory
32-bit physical address
20 BITS
12 BITS

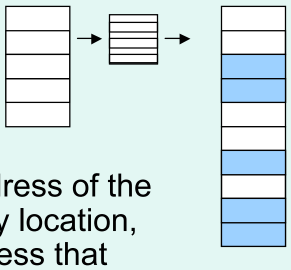
Hashed Page Tables

- 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

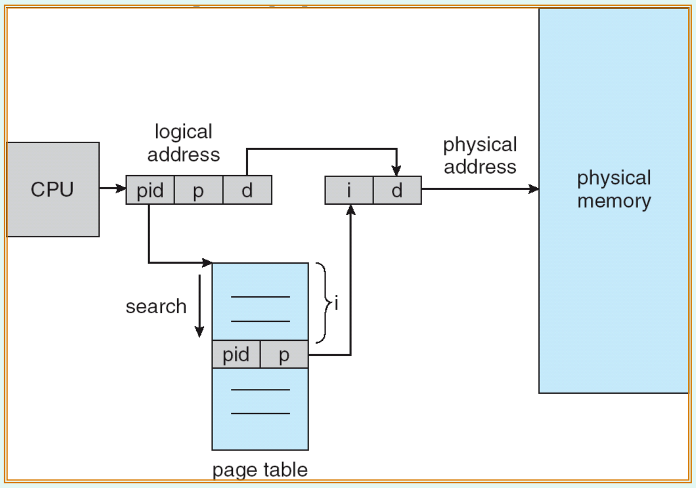


Inverted Page Table



- Not Good: millions of entries in the page table for a process
- Solution: 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
- WHY: 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

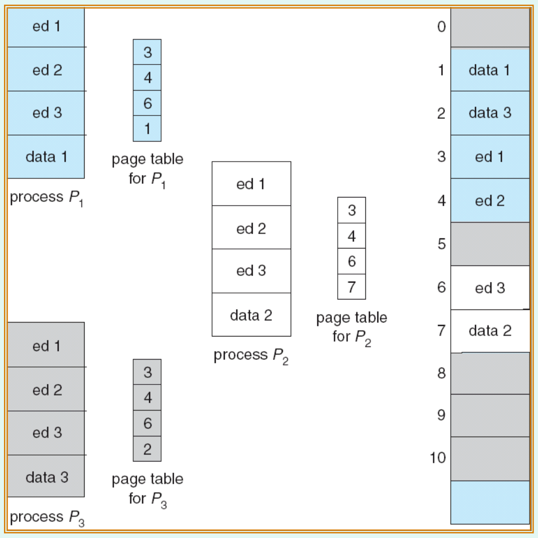
Inverted Page Table



Shared Pages

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

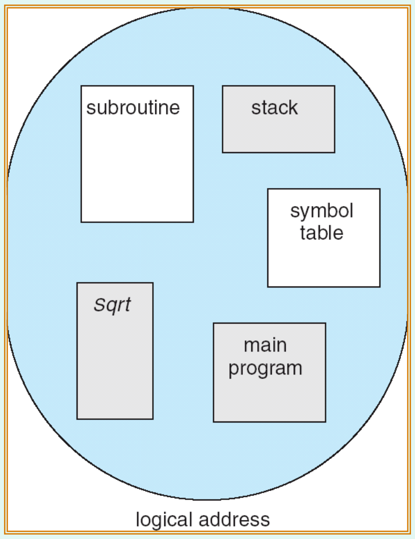
Shared Pages Example



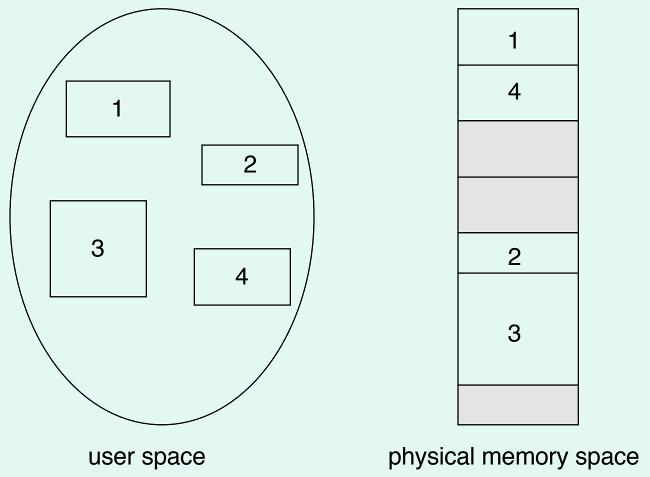
Segmentation

- 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

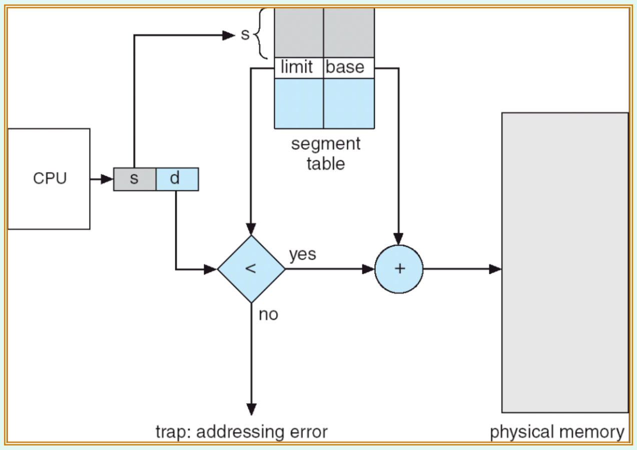
Segmentation Architecture

- **Relocation**.
  - dynamic
  - by segment table
- **Sharing**.
  - shared segments
  - same segment number
- **Allocation**.
  - first fit/best fit
  - external fragmentation

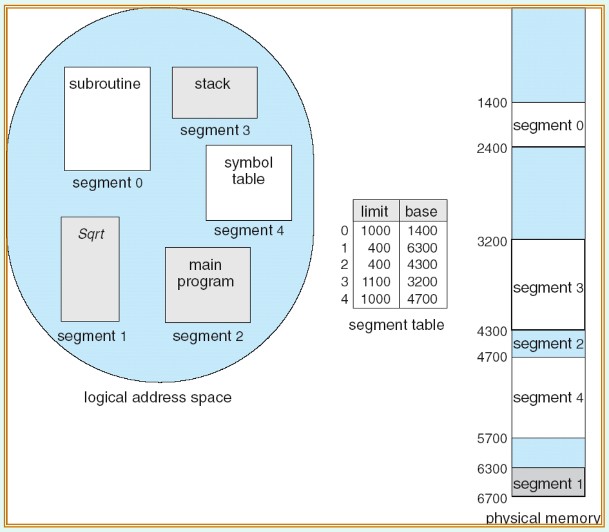
Segmentation Architecture (Cont.)

- 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

Address Translation Architecture



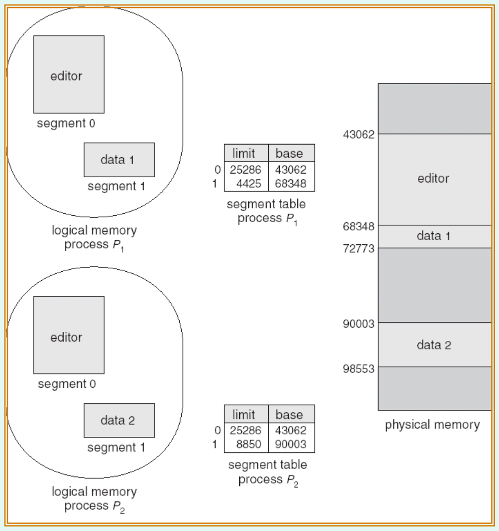
Example of Segmentation



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

segment table

Sharing of Segments



editor
segment 0
data 1
segment 1
logical memory process P_1
limit
base
0
25286
43062
1
4425
68348
segment table process P_1
editor
segment 0
data 2
segment 1
logical memory process P_2
limit
base
0
25286
43062
1
8850
90003
segment table process P_2
43062
editor
68348
data 1
72773
90003
data 2
98553
physical memory