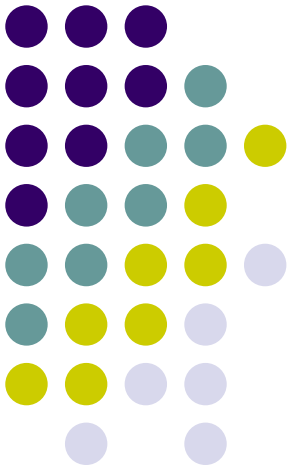
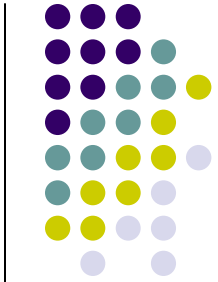
Memory Management



Goals of Memory Management

- Allocate available memory efficiently to multiple processes
- Main functions
  - Allocate memory to processes when needed
  - Keep track of what memory is used and what is free
  - Protect one process's memory from another

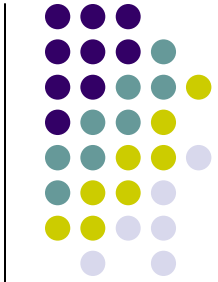
Memory Allocation



- Contiguous Allocation
  - Each process allocated a single contiguous chunk of memory
- Non-contiguous Allocation
  - Parts of a process can be allocated non-contiguous chunks of memory

In this part, we assume that the entire process needs to be in memory for it to run

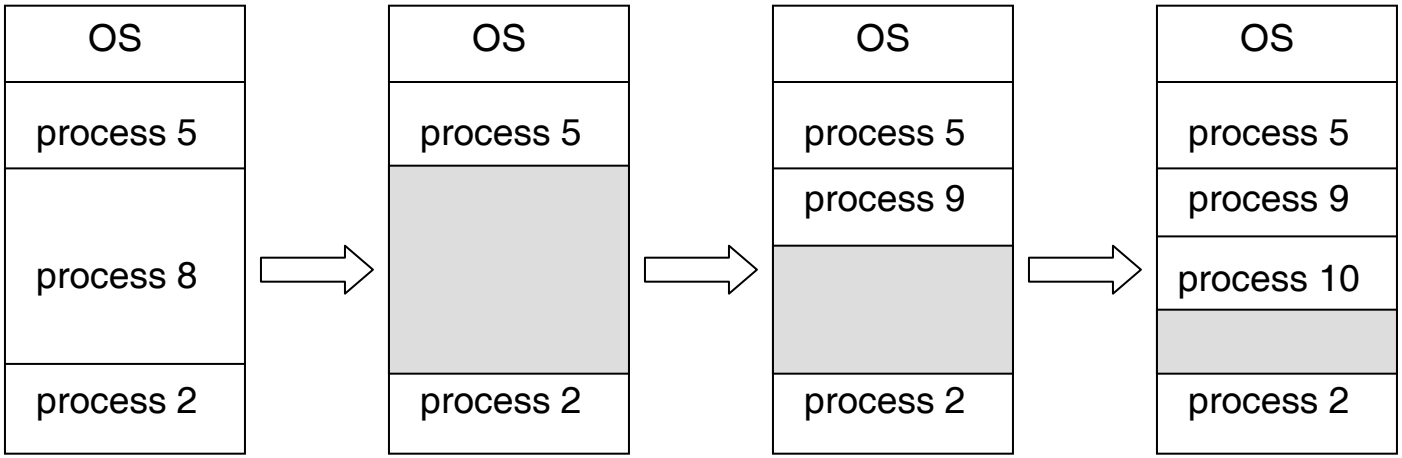
Contiguous Allocation



- Fixed Partition Scheme
  - Memory broken up into fixed size partitions
    - But the size of two partitions may be different
  - Each partition can have exactly one process
  - When a process arrives, allocate it a free partition
    - Can apply different policy to choose a partition
  - Easy to manage
  - Problems:
    - Maximum size of process bound by max. partition size
    - Large internal fragmentation possible

Contiguous Allocation (Cont.)

- Variable Partition Scheme
  - 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)

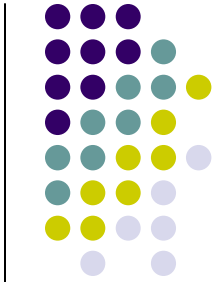


Dynamic Storage-Allocation Problem

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

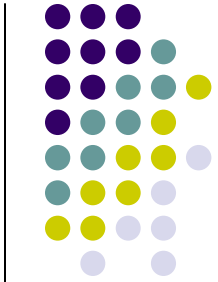
- First-fit: Allocate the *first* hole that is big enough
- Next-fit: Similar to first-fit, but start from last hole allocated
- Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole

Fragmentation



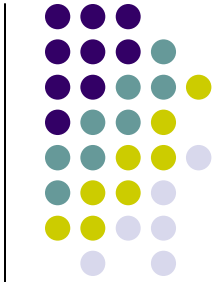
- External Fragmentation: total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation: allocated memory may be larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Costly

Keeping Track of Free Partitions



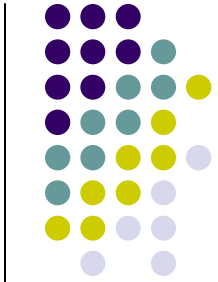
- Bitmap method
  - Define some basic fixed allocation unit size
  - 1 bit maintained for each allocation unit
    - 0 – unit is free, 1 – unit is allocated
  - Bitmap – bitstring of the bits of all allocation units
  - To allocate space of size *n* allocation units, find a run of *n* consecutive 0's in bitmap
- Maintain a linked list of free partitions
  - Each node contains start address, size, and pointer to next free block

Non-contiguous Allocation



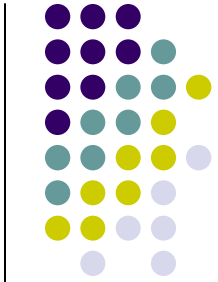
- Paging
- Segmentation

Memory Abstraction

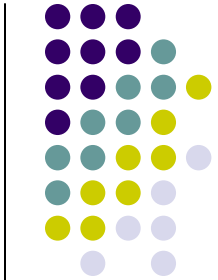


- What does the programmer see as “memory”
- Simplest: No abstraction
  - Programmer sees the physical memory
  - Compiler generates absolute physical memory addresses
- Abstraction: Address Spaces
  - A set of addresses that the process can use to address memory
  - Each process has its own address space

The Case of No Abstraction



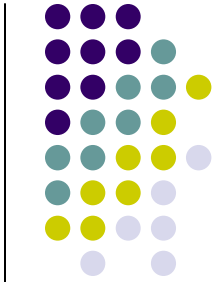
- Addresses generated by compiler (instruction and data) refer to exact physical memory addresses
  - Compile time binding
- Instruction and data must be loaded in exactly the same physical memory locations
- Advantage: Fast execution
  - No address translation overhead during actual memory access
- Problem: Unrelated processes may read/write from/to each others' address space



- Multiple processes can still be run
  - If the behavior of the processes are well-known and they use different ranges of physical address
    - Possible in some closed systems with known processes
  - Swapping
    - Keep one process in memory at one time
    - Copy the memory space of the process to disk when another process is to be run
    - Copy the memory space back from the disk when the process needs to be rerun
- Not good for general purpose multiprogramming systems

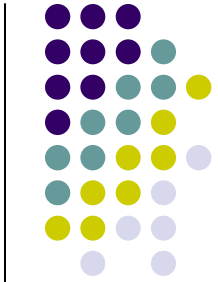
Memory Abstraction: Logical or Virtual Addresses

- Each process has its own address space (Logical Address Space)
- Translating to physical address – Load Time or Run Time
- Load time binding
  - Compiler generates addresses in the process's address space
  - Loader changes addresses during loading depending on where in physical memory the process is loaded
  - Advantage: No address translation overhead during running
  - Problem: total memory requirement of a process needs to be known a-priori
  - Problem: Process cannot be moved during execution
  - Problem: Rogue process can still overwrite other process's memory by writing out of bounds, no runtime check

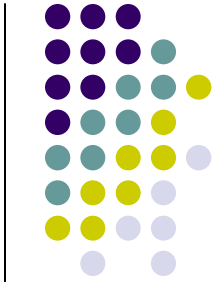


- Load time binding with runtime check
  - Address bound at load time, but checked at run time if within bound
  - Solves the problem of overwriting other process's memory, but increases cost of access
- One simple method
  - H/w provided base and limit registers
    - Accessible only by OS
  - Base register loaded with beginning physical memory address of process given at load time
  - Limit register loaded with length of memory given to process
  - On every access, hardware checks if limit register is exceeded
    - Aborts program if limit is exceeded

Logical or Virtual Address (contd.)

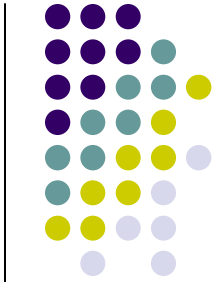


- Execution/Run time binding
  - Physical address corresponding to a logical address found only when the logical address is used
  - Process can be moved during its execution
  - CPU generates logical address
  - Memory Management Unit (MMU): hardware that converts a generated logical address to physical address before access
  - Advantage: Processes can be moved during execution, protects one process from another, can grow process' memory at run time
  - Problem: Address translation overhead at run time



- The user program deals with logical addresses; it never sees the *real* physical addresses
- The same logical address space in the address space of two processes must always map to different physical addresses at runtime
- How to ensure this for run time bindings?

A Simple Solution

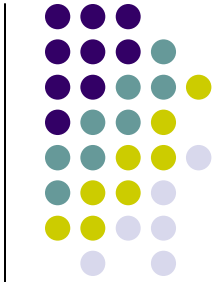


- H/w provided base and limit registers
  - Accessible only by OS
- Programs loaded in consecutive memory locations without relocation during load
- Base register loaded with beginning physical memory address of process
- Limit register loaded with length of process
  - Must be known a-priori
- On every access, MMU adds base register to logical address, and then checks if limit register is exceeded
  - Aborts program if limit is exceeded
- Hard to grow memory if needed, but possible

A Better Solution: Paging

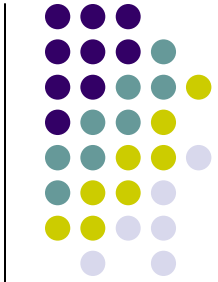
- Allows processes to grow memory as and when needed
- Logical/Virtual address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Allows multiple processes to reside in memory at the same time

Paging



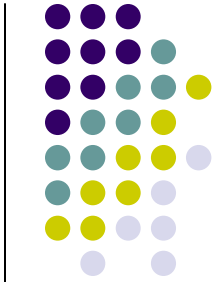
- Divide physical memory into fixed-sized (power of 2) blocks called frames
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames.
- To run a program of size *n* pages, need to find *n* free frames and load program.
- Page table: used to translate logical to physical addresses
  - One page table per process

Page Table



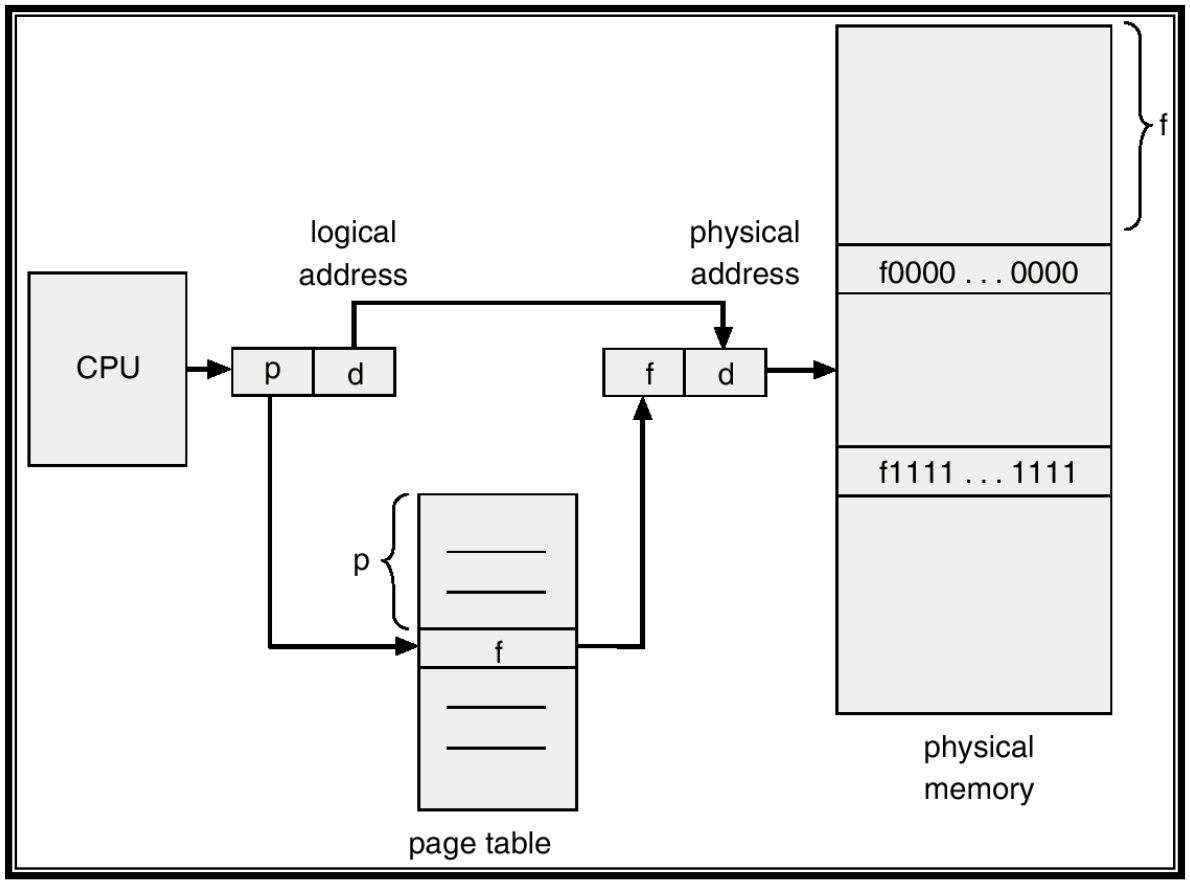
- One entry for each page in the logical address space
- Contains the base address of the page frame where the page is stored
- Also contains a valid bit
  - If set, logical address is valid and has physical memory allocated to it
  - If not set, logical address is invalid

Address Translation Scheme

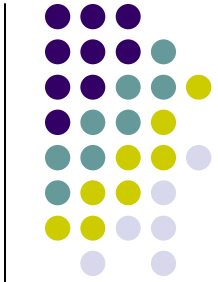


- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into the *page table* which contains base address of the corresponding page frame in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit
- Use page number to index the page table
- Get the page frame start address
- Add offset with that to get the actual physical memory address
- Access the memory

Address Translation Architecture

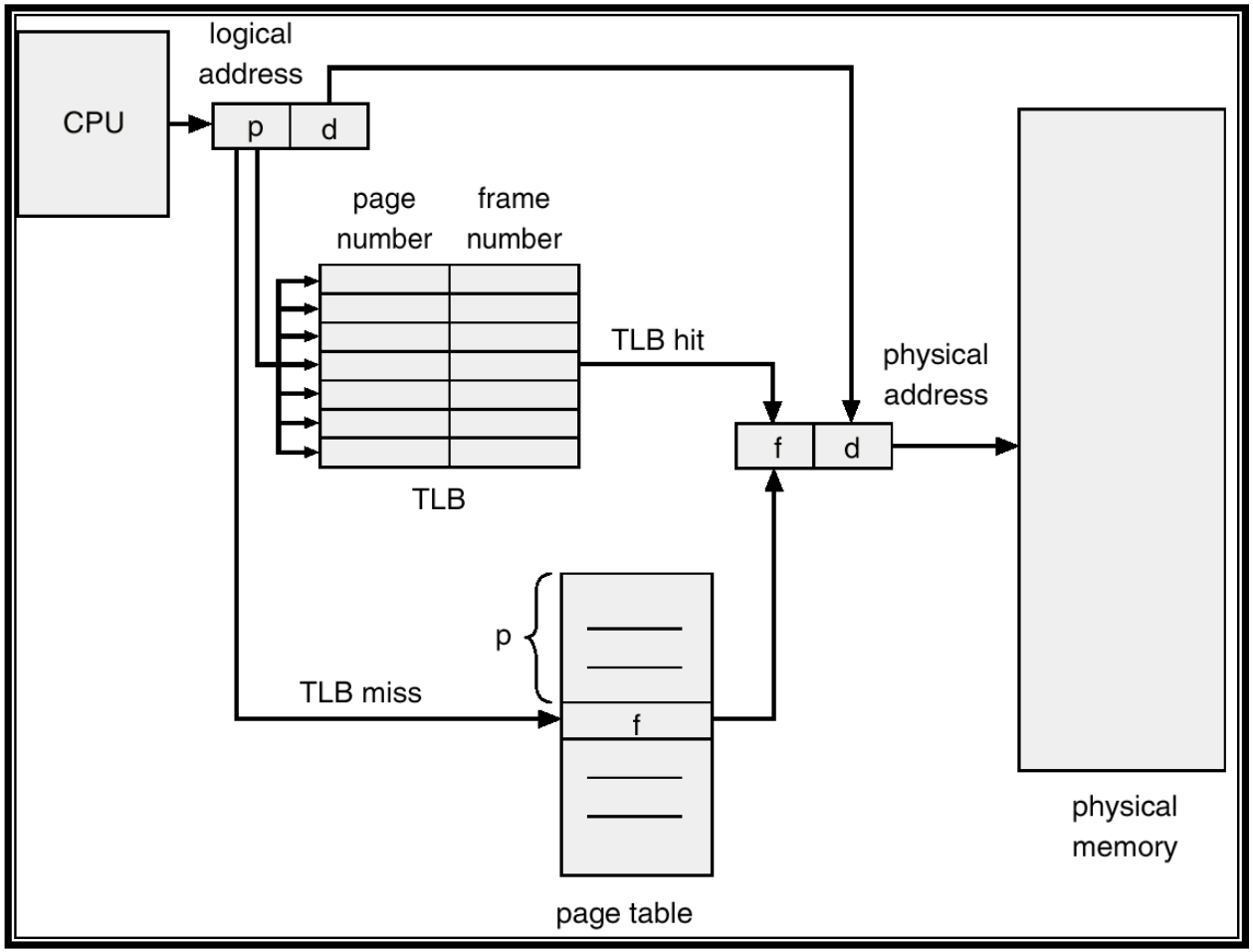


Implementation of Page Table

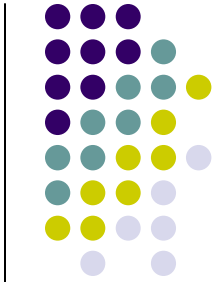


- Page table is kept in main memory.
- Page-table base register (PTBR) points to the page table.
- Page-table length register (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called translation look-aside buffers (TLBs)

Paging Hardware With TLB



Effective Access Time

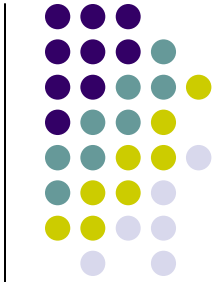


- TLB Lookup time = ε
- Assume memory cycle time is 1 time unit
- Hit ratio – percentage of times that a page number is found in the TLB;
- Hit ratio = α
- Effective Access Time (EAT)

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

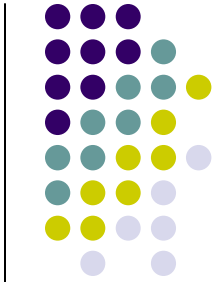
$$= 2 + \varepsilon - \alpha$$

Page Table Structure



- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

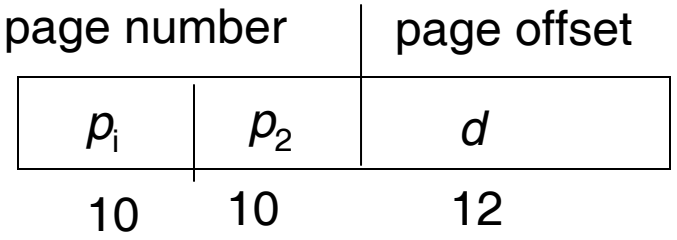


- Break up the logical address space into multiple page tables.

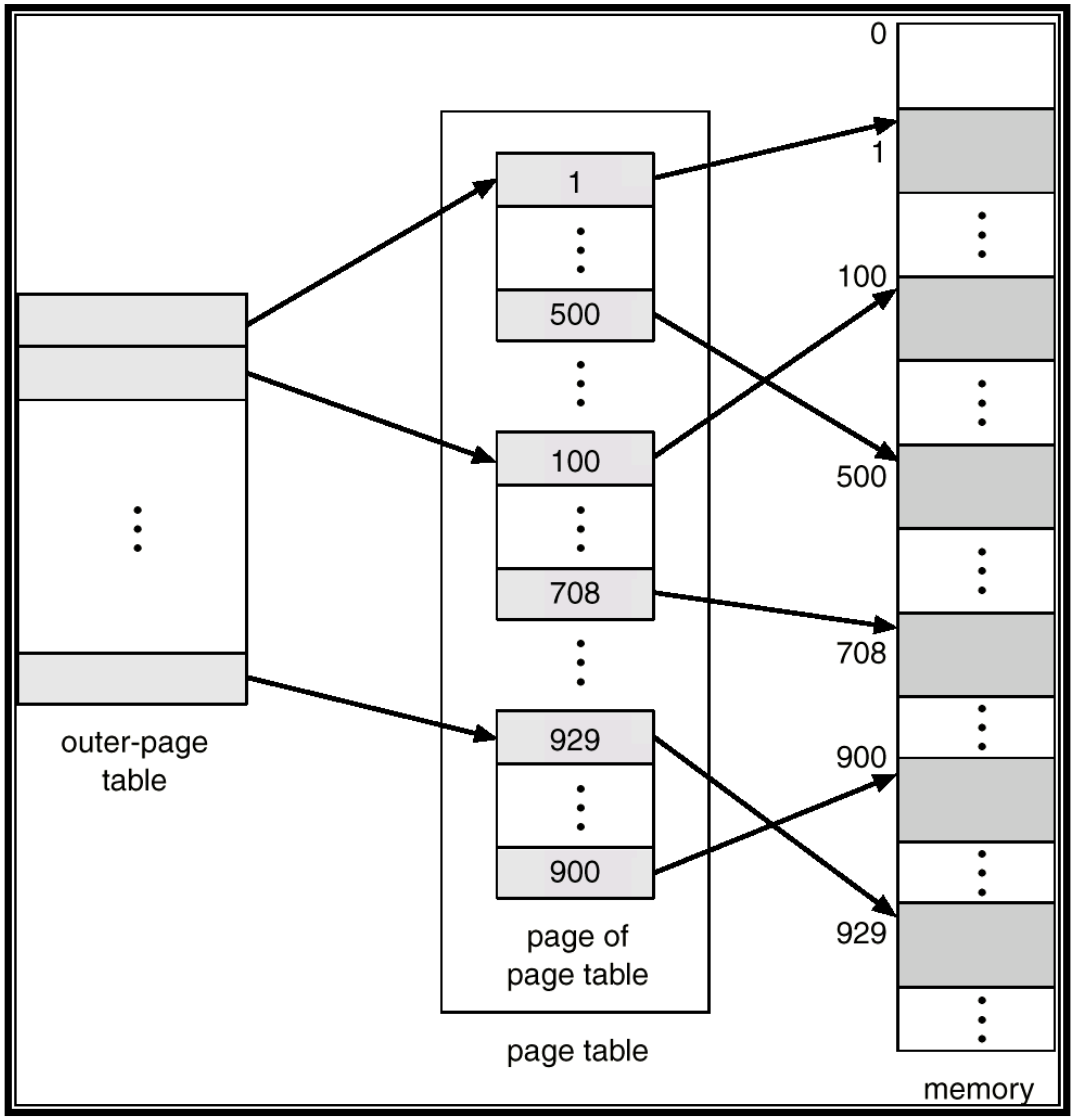
- A simple technique is a two-level page table.

Two-Level Paging Example

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

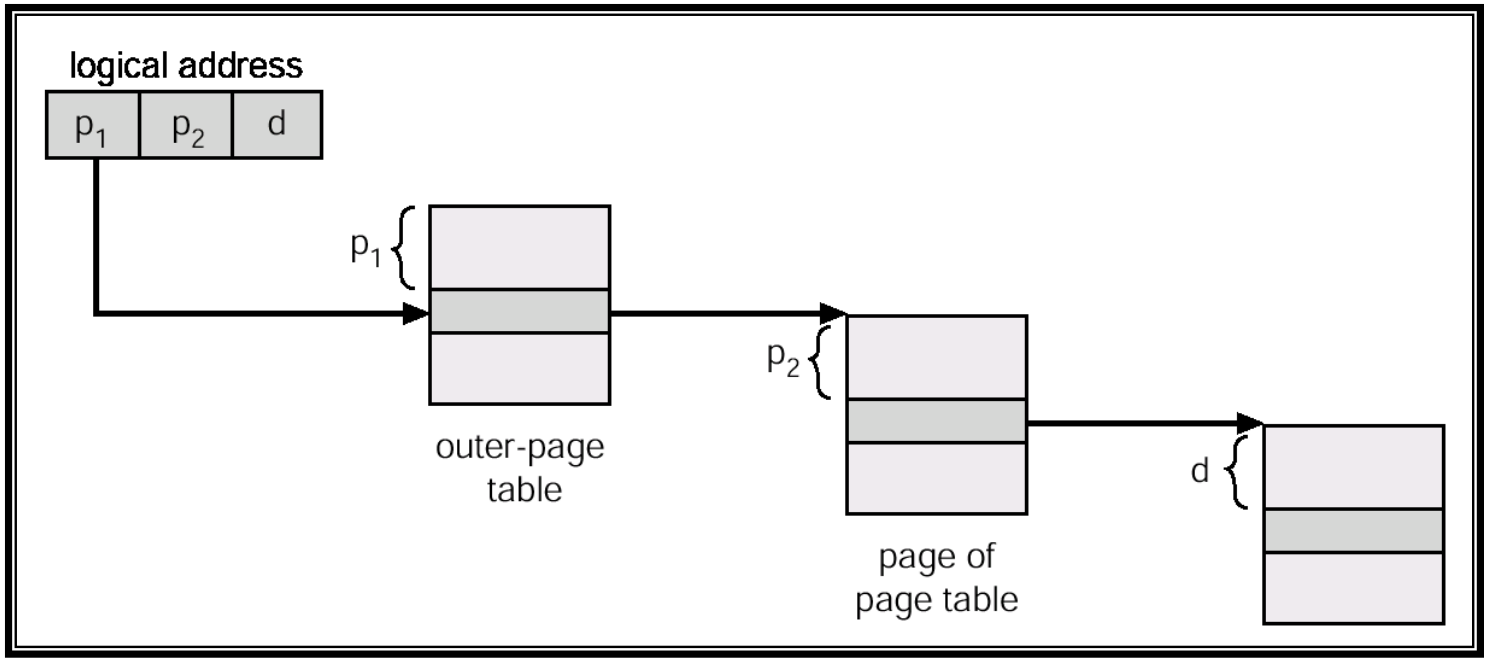


Two-Level Page-Table Scheme

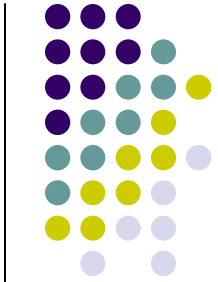


Address-Translation Scheme

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

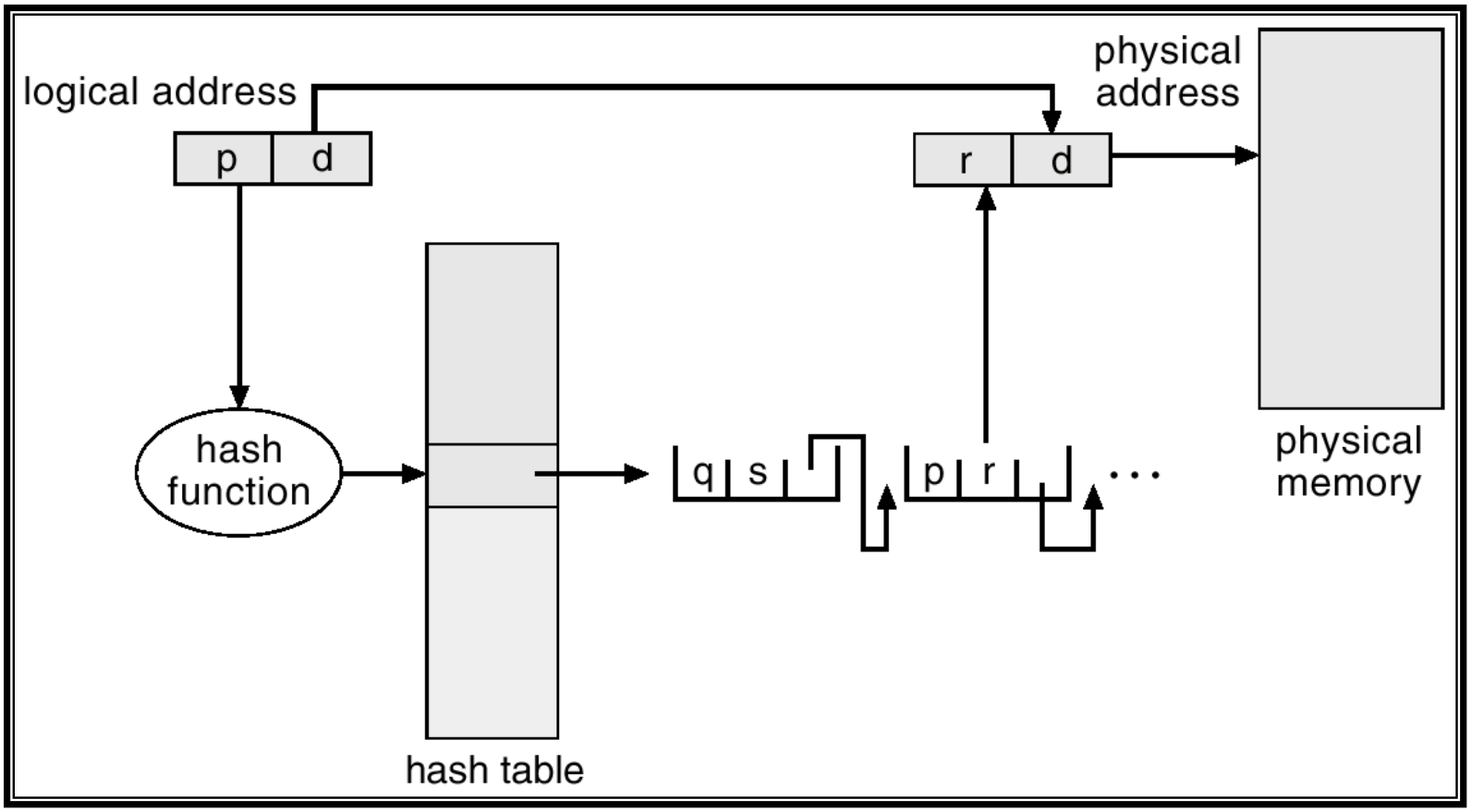


Hashed Page Tables

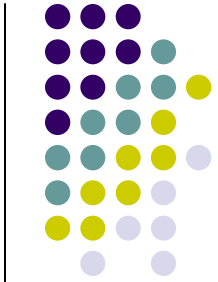


- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Hashed Page Table

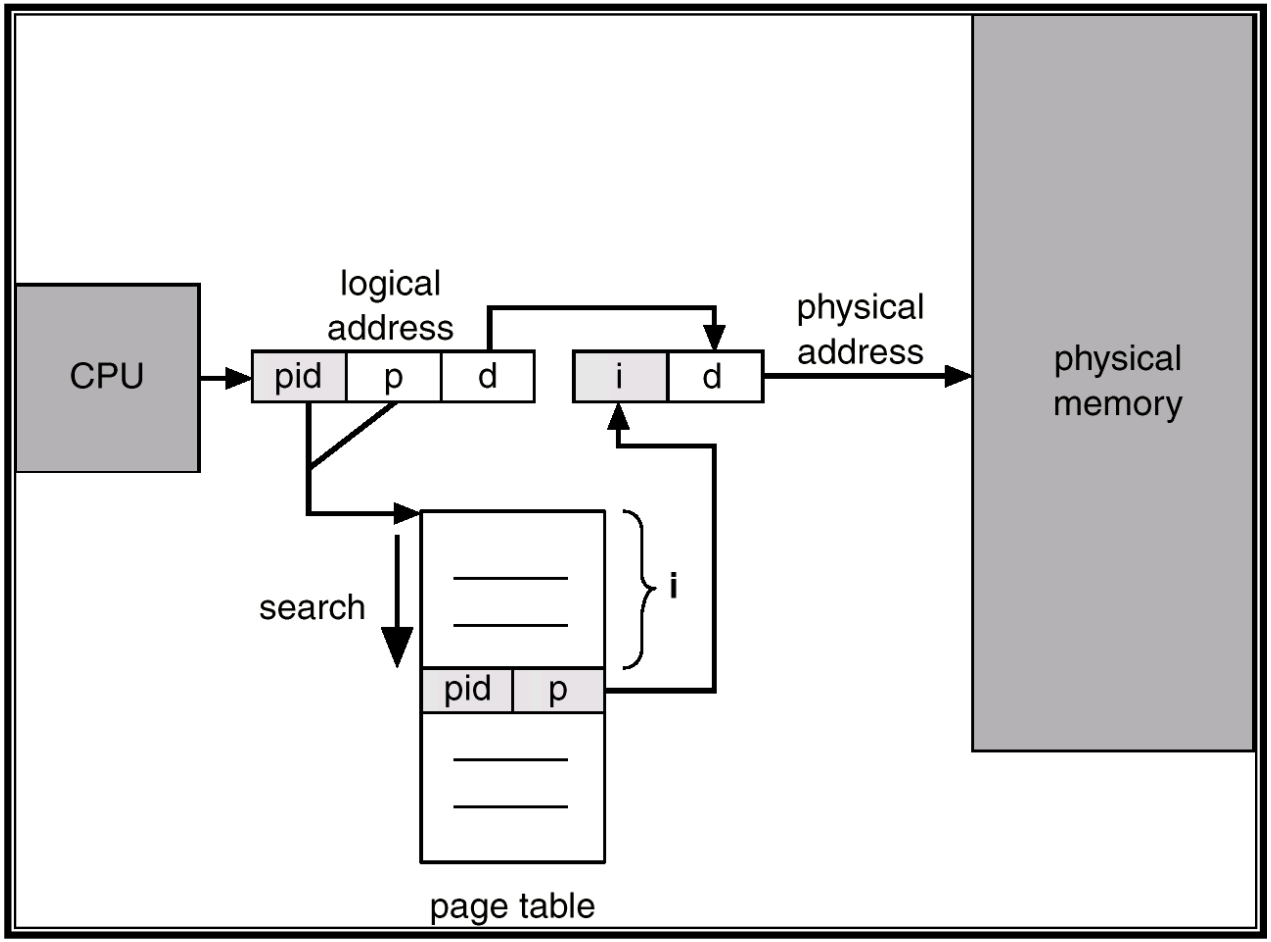


Inverted Page Table

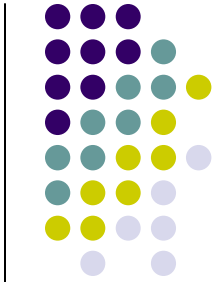


- One entry for each real page of memory (page frame)
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries.

Inverted Page Table Architecture

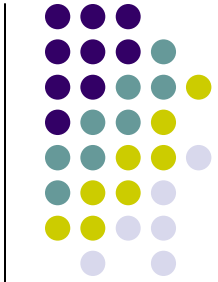


Protection



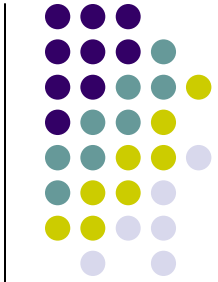
- Protection bit can be there with each page in the page table
  - Ex. – read-only page
  - Bits set by OS
- MMU can check for access type when translating address
  - Traps if illegal access
- More elaborate protections possible with h/w support

Shared Pages



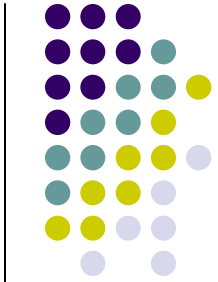
- Example: Shared code
  - One copy of read-only code shared among processes (i.e., text editors, compilers, window systems)
- Store shared page in a single page frame
- Map it to logical address spaces of processes by inserting appropriate entries in their page tables that all point to the shared page frame

Segmentation



- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment can be any logical unit
  - code, global variables, heap, stack,…
- Segment sizes may be different

Segmentation Architecture



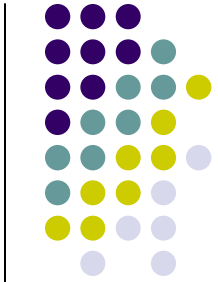
- Logical address consists of a two tuple:

  <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;

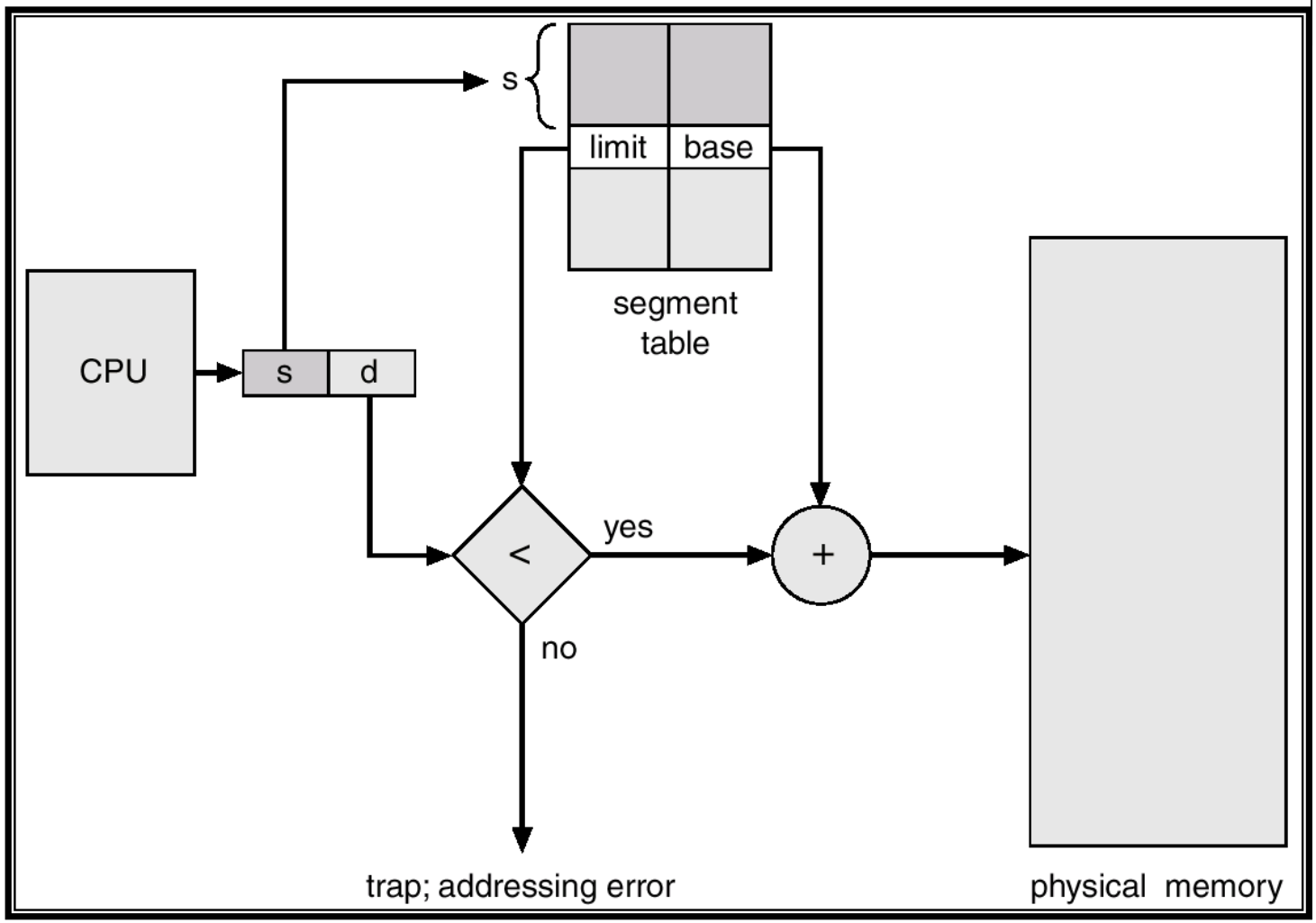
  segment number *s* is legal if *s* < STLR.

Segmentation Architecture (Cont.)

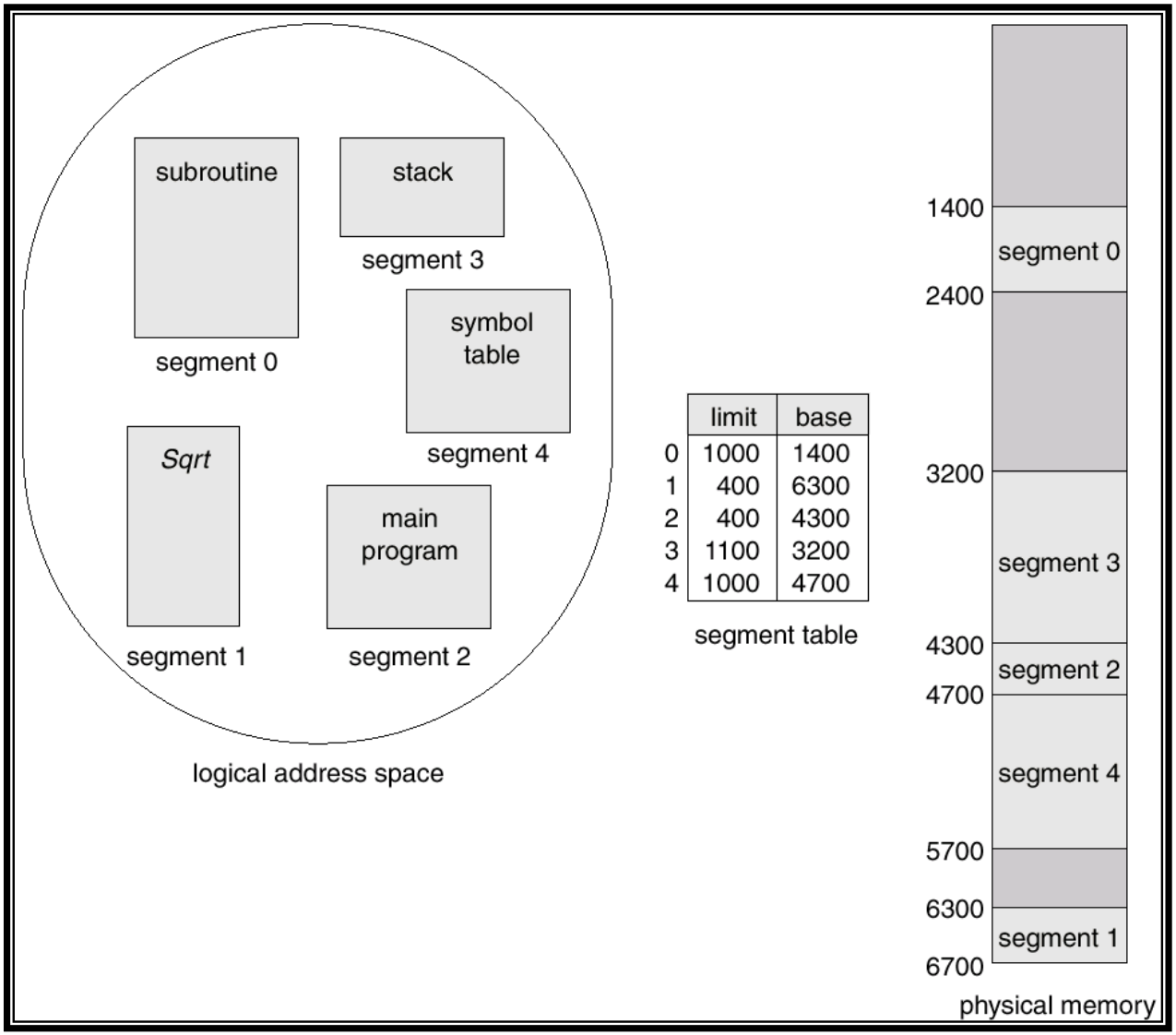


- Protection. With each entry in segment table associate:
  - validation bit = 0 $\Rightarrow$ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

Segmentation Hardware



Example of Segmentation



Sharing of Segments