COMS 3995: Networks, Operating Systems and Security
Memory Management

Guest lecture by Junfeng Yang

Outline

- Dynamic memory allocation
  - Stack
  - Heap
    - Heap allocation strategies
- Intro to memory management
- Paging

Dynamic memory allocation

- Static (compile time) allocation is not possible for all data

- Two ways of dynamic allocation
  - Stack allocation
    - Restricted, but simple and efficient
  - Heap allocation
    - More general, but less efficient
    - More difficult to implement

Stack organization

- Memory is freed in opposite order from allocation. Last in First out (LIFO)
- When useful?
  - Memory usage pattern follows LIFO
    - E.g., function call frames
- Implementation
  - Pointer separating allocated and free space
  - Allocate: increment pointer
  - Free: decrement pointer

Pros and cons of stack organization

- Pros
  - Simple and efficient
  - Keeps all free space continuous
- Cons
  - Not for general data structures

Heap organization

- Allocate from random locations
  - Memory consists of allocated area and free area (or holes)
- When useful?
  - Allocate and free are unpredictable
  - Complex data structures
    - new in C++, malloc in C, kmalloc in Linux kernel

Pros and cons of heap organization

- Pros
  - General, works on arbitrary allocation and free patterns
- Cons
  - End up with small chunks of free space

Dynamic allocation issue: fragmentation

- Small trunks of free memory, too small for future allocations
  - External fragment: visible to system
  - Internal fragment: visible to process (e.g. if allocate at some granularity)
- Goal
  - Reduce number of holes
  - Keep holes large
- Stack fragmentation v.s. heap fragmentation

Heap implementation

- Data structure: linked list of free blocks
  - free list: chains free blocks together
- Allocation
  - Choose block large enough for request
  - Update free list
- Free
  - Add block back to list
  - Merge adjacent free blocks

Heap allocation strategies

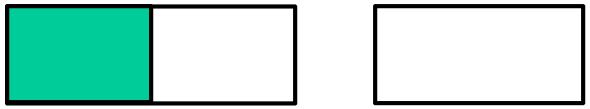
- Best fit
  - Search the whole list on each allocation
  - Choose the smallest block that can satisfy request
  - Can stop search if exact match found
- First fit
  - Choose first block that can satisfy request
- Worst fit
  - Choose largest block (most leftover space)

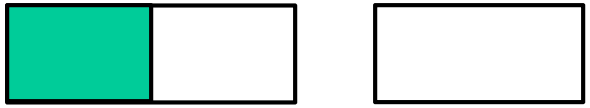
Which is better?

Example

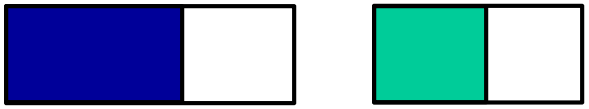
- Free space: 2 blocks of size 20 and 15
- Workload 1: allocation requests: 10 then 20

Best fit 

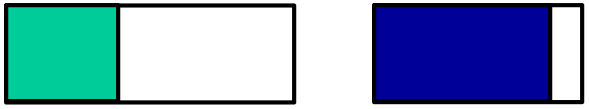
First fit  Request of 20: fail!

Worse fit  Request of 20: fail!

- Workload 2: allocation requests: 8, 12, then 12

Best fit  Request of 12: fail!

First fit 

Worse fit  Request of 12: fail!

Comparison of allocation strategies

- Best fit
  - Tends to leave very large holes and very small holes
  - Disadvantage: very small holes may be useless
- First fit:
  - Tends to leave "average" size holes
  - Advantage: faster than best fit
- Worst fit:
  - Simulation shows that worst fit is worst in terms of storage utilization

Outline

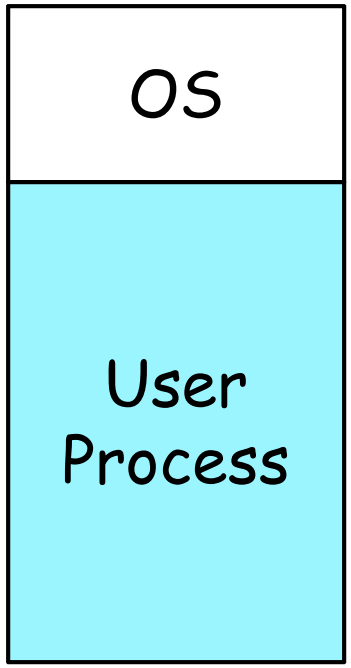
- Dynamic memory allocation
  - Stack
  - Heap
    - Heap allocation strategies
- Intro to memory management
- Paging

Motivation for memory anagement

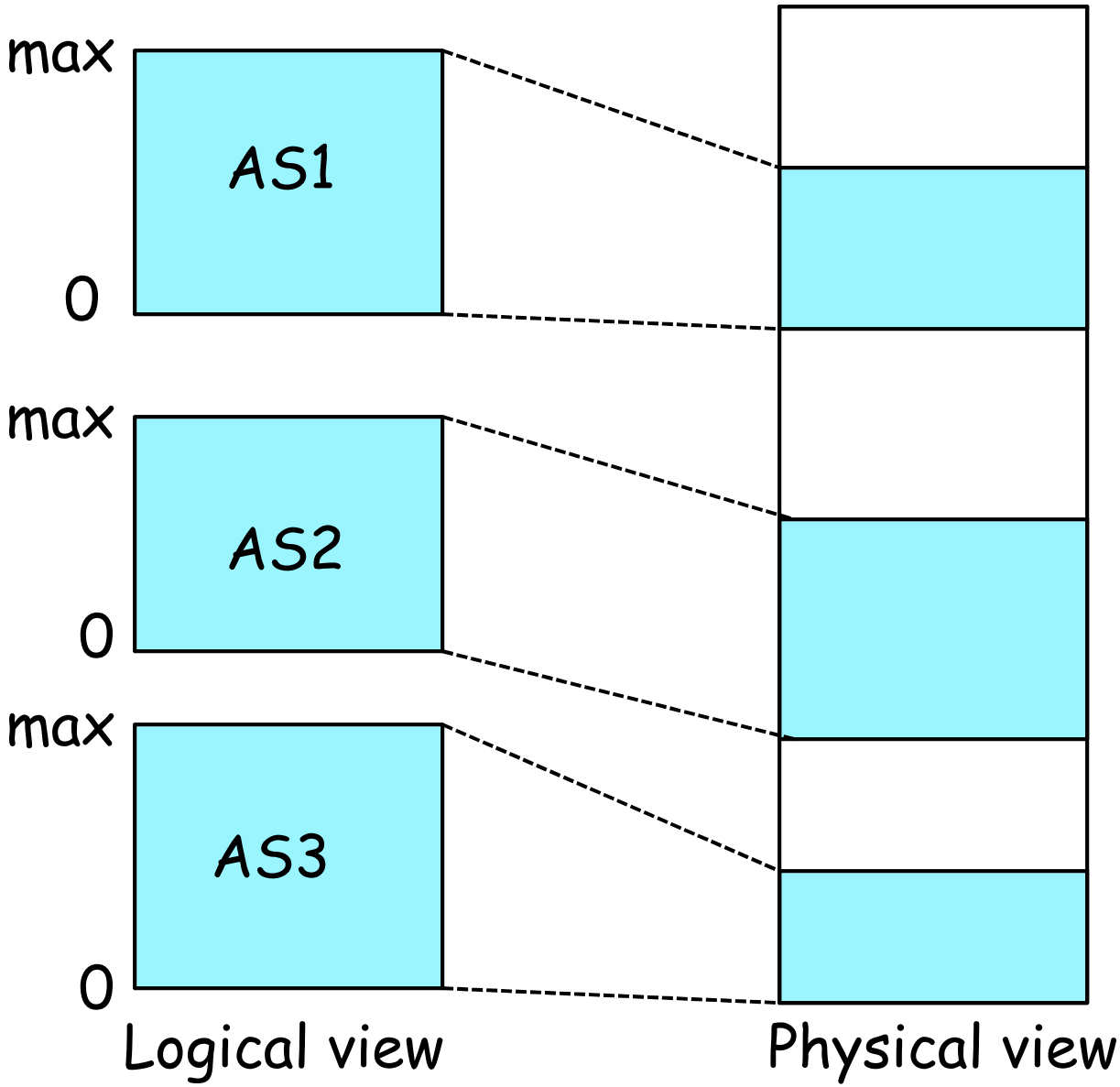
- Simple uniprogramming with a single segment per process

- Uniprogramming disadvantages
  - Only one process can run a time
  - Process can destroy OS

- Want multiprogramming!



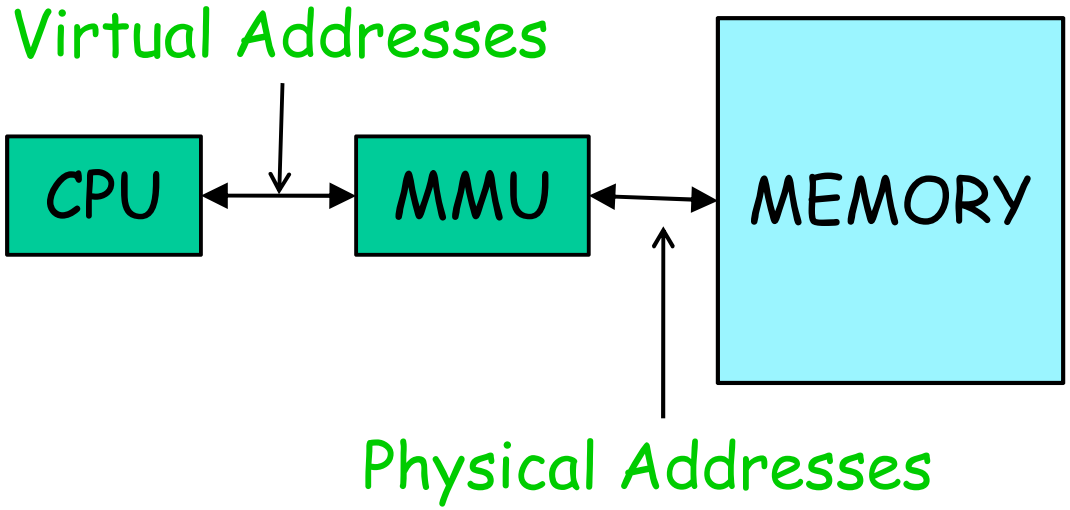
Multiple address spaces co-exist



Multiprogramming wish-list

- Sharing
  - multiple processes coexist in main memory
- Transparency
  - Processes not aware that memory is shared
  - Run regardless of number and/or locations of processes
- Protection
  - Cannot corrupt OS or other processes
  - Privacy: cannot read data of other processes
- Efficiency: should have reasonable performance
  - Purpose of sharing is to increase efficiency
  - Do not waste CPU or memory resources

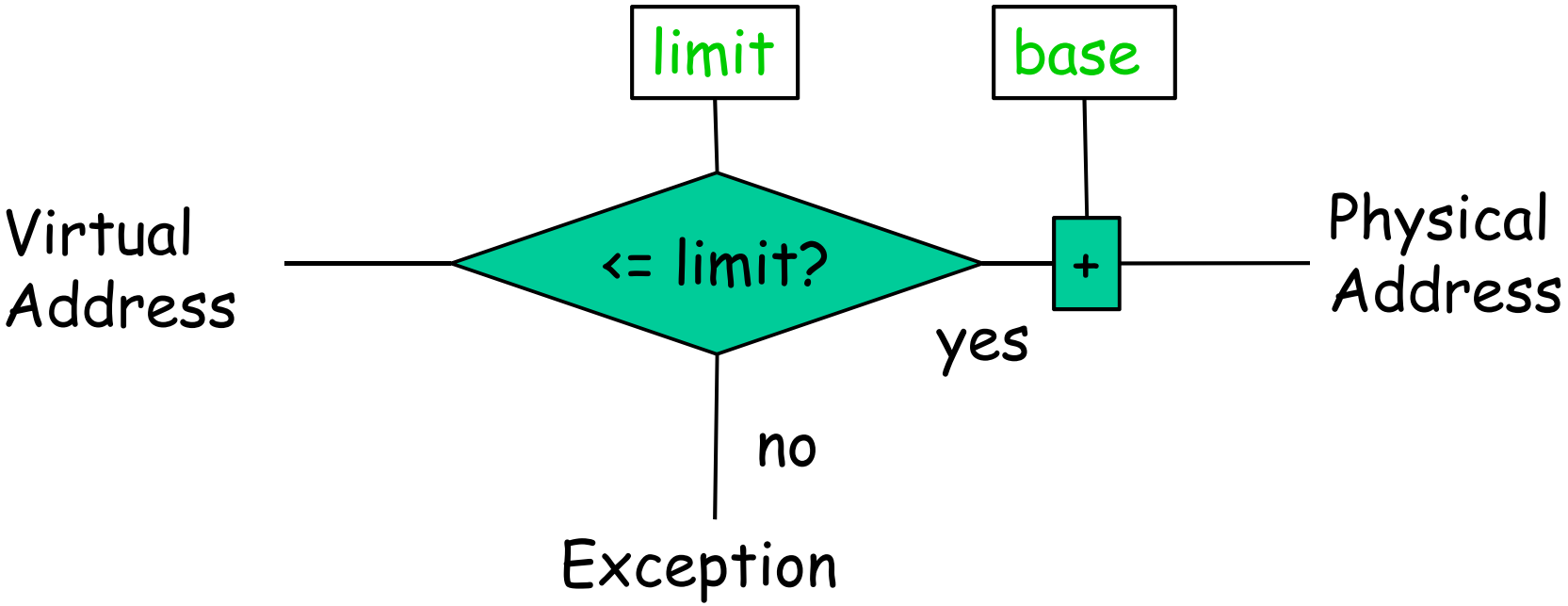
Memory translation and protection



- Map program-generated address (virtual address) to hardware address (physical address) dynamically at every reference
  - MMU: Memory Management Unit
  - Controlled by OS

Simple implementation of memory translation and protection

- Compare logical address to limit register
  - If greater, generate exception
- Add base register to logical address to generate physical address



Managing processes with base and limit

- Does base contain logical or physical address?
- How to relocate process?
- Base and limit registers are per-process or global?
- What to do on a context switch?
- Can user processes modify base and limit registers?

Pros and Cons of Base and Limit

- Advantages
  - Supports dynamic relocation of address space
  - Supports protection across multiple spaces
  - Cheap: few registers and little logic
  - Fast: add and compare can be done in parallel
- Disadvantages
  - Process must be allocated contiguously
  - May allocate memory not used
  - Cannot share limited parts of address space

Outline

- Dynamic memory allocation
  - Stack
  - Heap
    - Heap allocation strategies
- Intro to memory management
- Paging
  - Overview
  - Page translation
  - Page allocation
  - Page protection
  - Translation Look-aside Buffers (TLB)
  - Page sharing
  - Page table structure

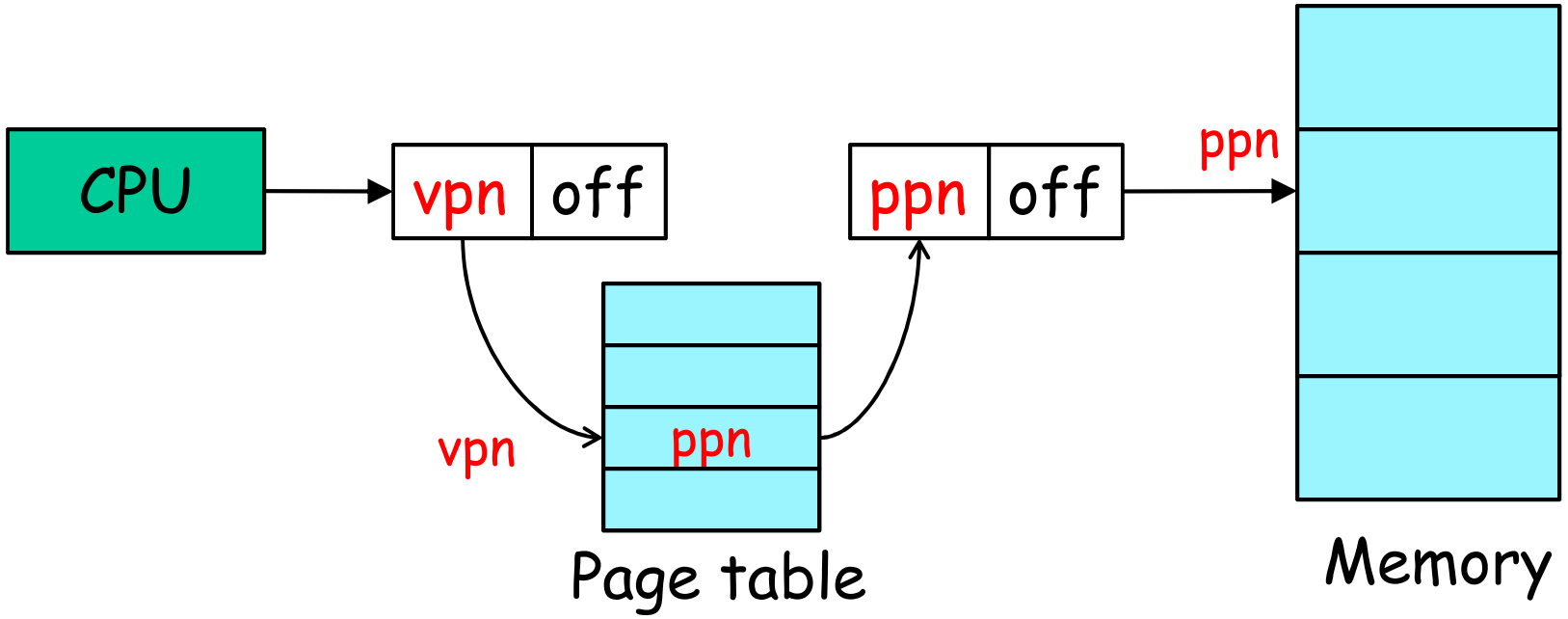
Paging overview

- Goal
  - Eliminate external fragmentation
  - Don't allocate memory that will not be used
  - Enable sharing
- Paging: divide memory into fixed-sized pages
  - Both virtual and physical memory are composed of pages
- Another terminology
  - A virtual page: page
  - A physical page: frame

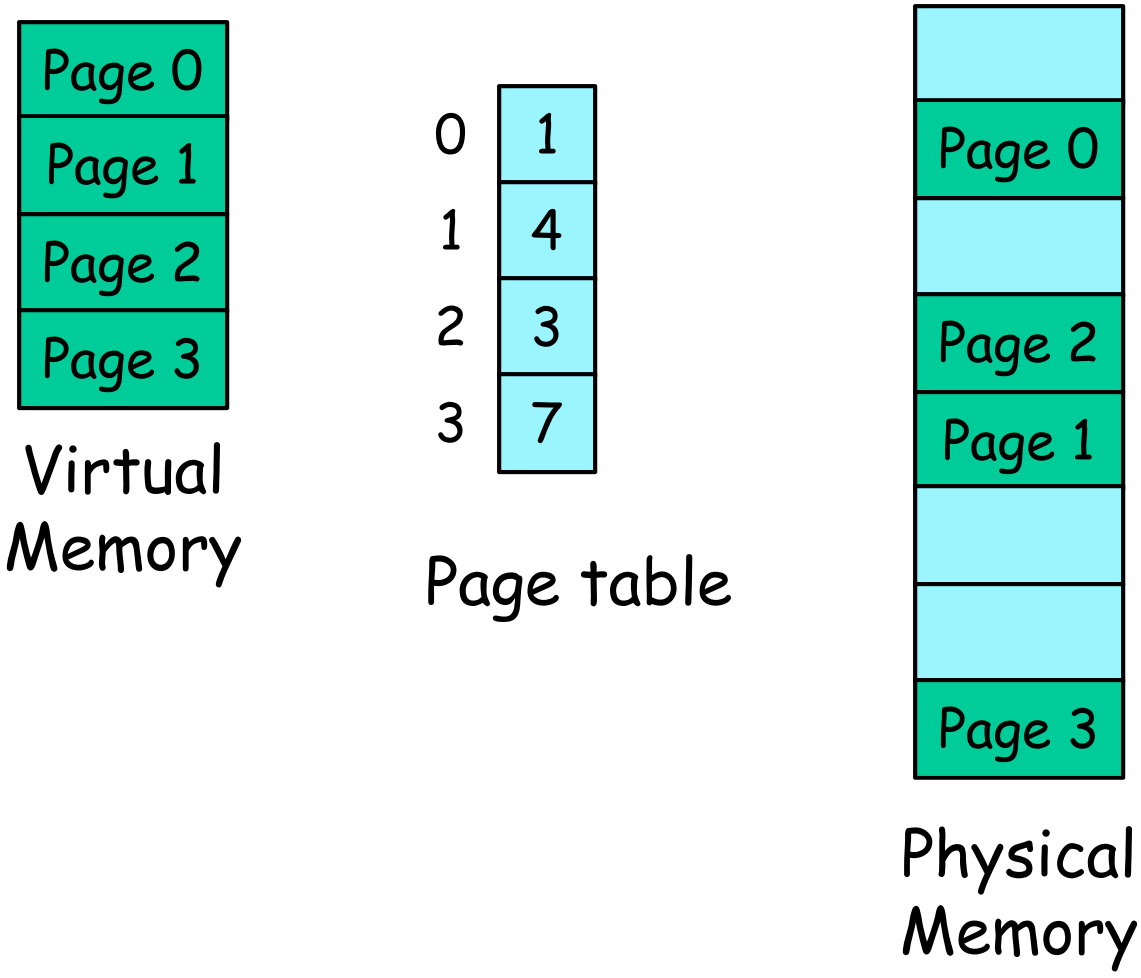
Page translation

- Address bits = page number + page offset
- Translate virtual page number (vpn) to physical page number (ppn) using page table

  pa = page_table[va/pg_sz] + va%pg_sz



Page translation example



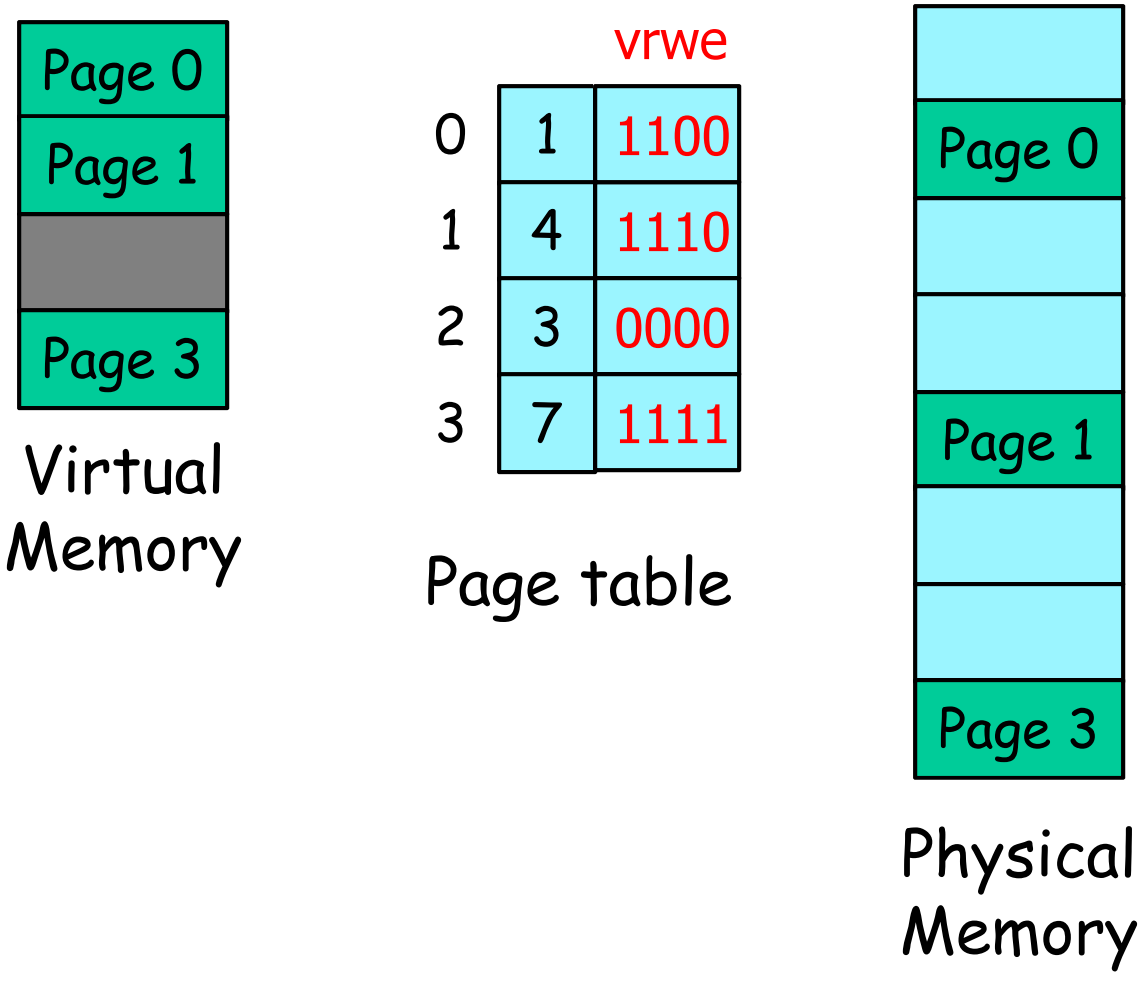
Page translation exercise

- 8-bit virtual address, 10-bit physical address, and each page is 64 bytes
  - How many virtual pages?
  - How many physical pages?
  - How many entries in page table?
  - Given page table = [2, 5, 1, 8], what's the physical address for virtual address 241?
- m-bit virtual address, n-bit physical address, k-bit page size
  - What are the answers to the above questions?

Page protection

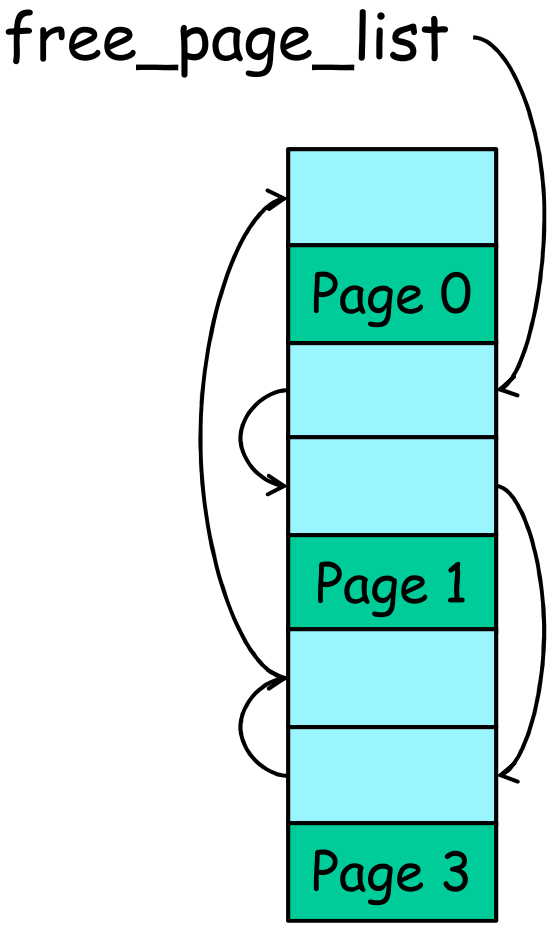
- Implemented by associating protection bits with each virtual page in page table
- Protection bits
  - valid bit: map to a valid physical page?
  - read/write/execute bits: can read/write/execute?
- Checked by MMU on each memory access

Page protection example



Page allocation

- Free page management
  - E.g., can put page on a free list
- Allocation policy
  - E.g., one page at a time, from head of free list



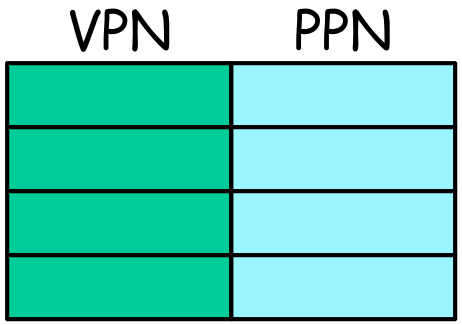
2, 3, 6, 5, 0

Implementation of page table

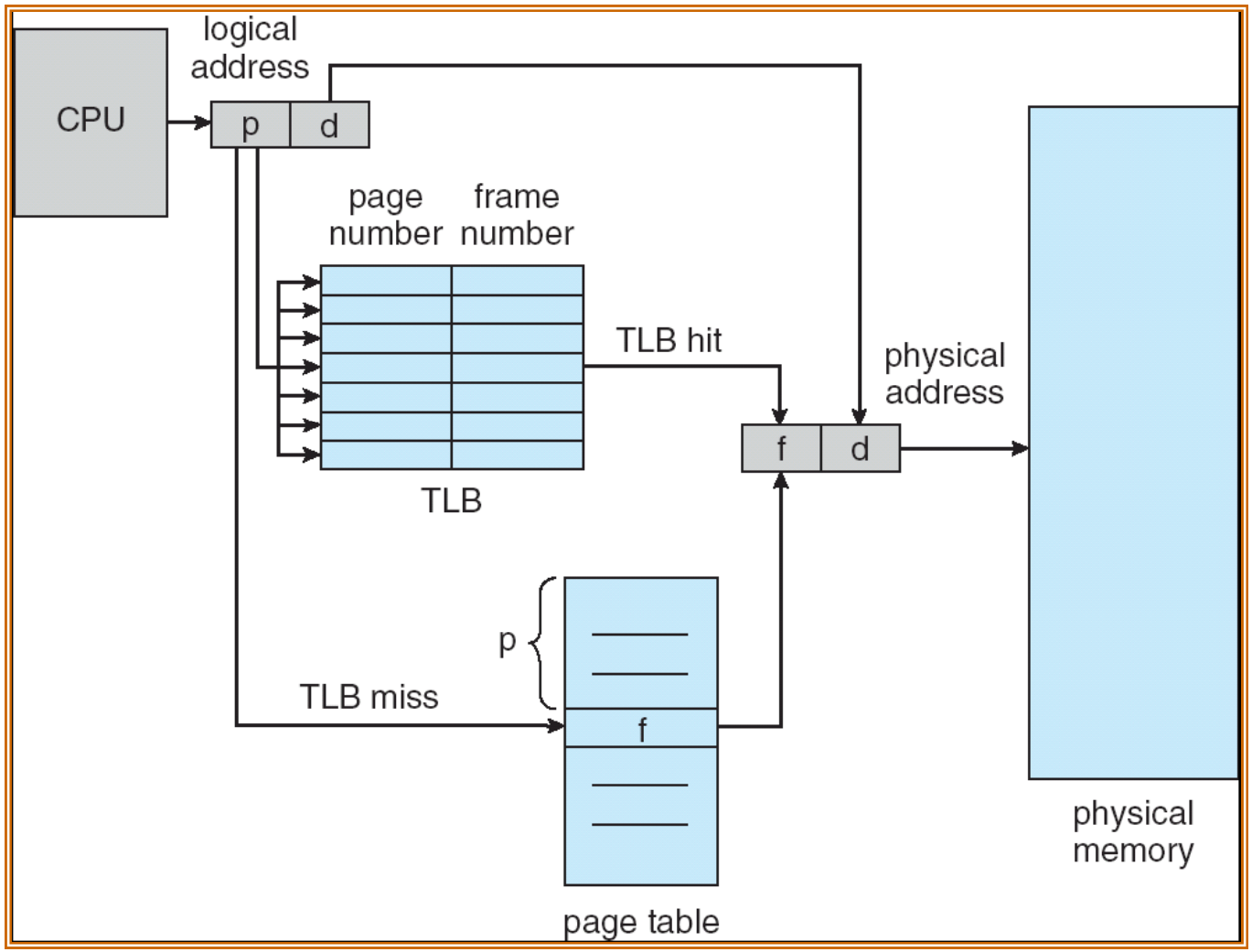
- Page table is stored in memory
  - Page table base register (PTBR) points to the base of page table
  - OS stores the value of this register in process control block (PCB)
  - OS switches PTBR on each context switch
- Problem: each data/instruction access requires two memory accesses
  - Extra memory access for page table

Avoiding extra memory access

- Fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
- Fast parallel search (CPU speed)
- Small



Paging hardware with TLB



Effective access time

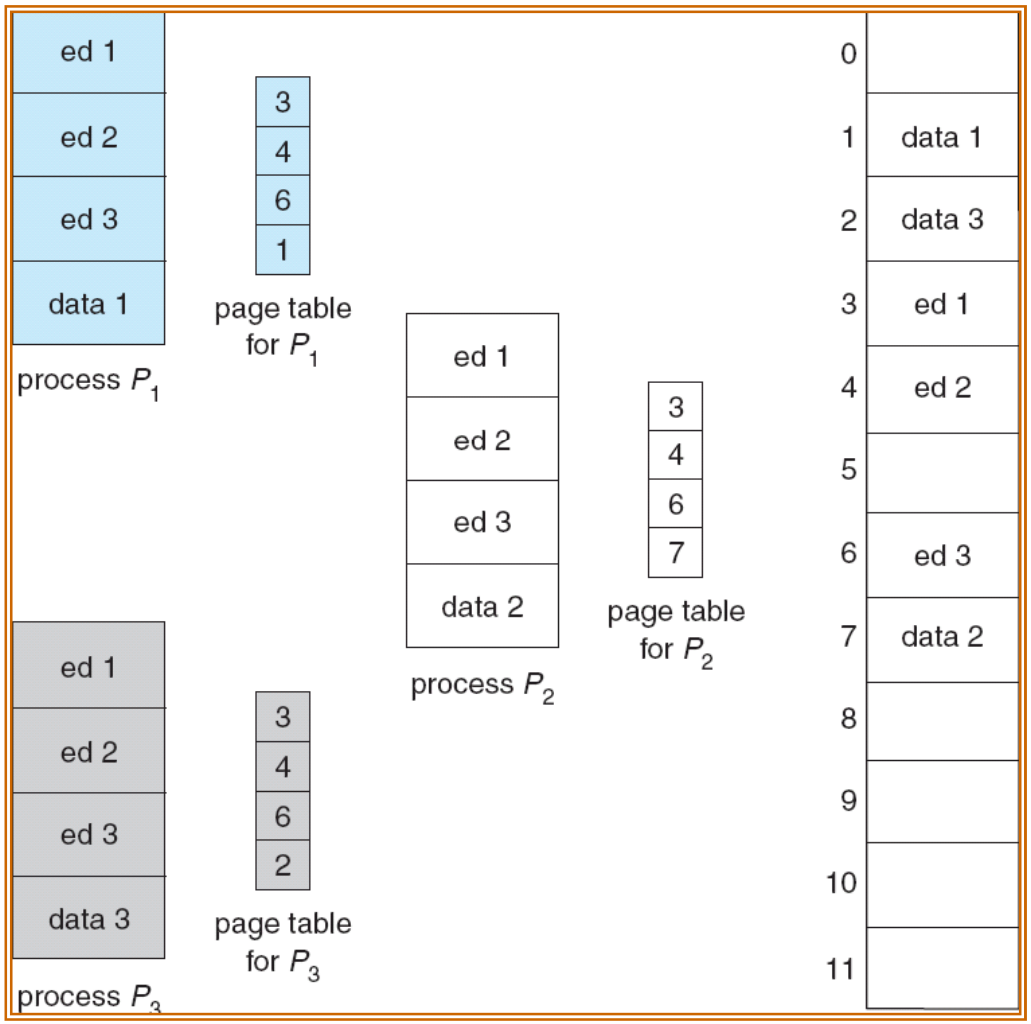
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 ms
- Hit ratio – α
  - percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Effective Access Time (EAT)

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

Motivation for page sharing

- Memory efficiency. E.g., one copy of read-only code/data shared among processes
- Efficient communication. Processes communicate by write to shared pages

Page sharing example



Page table size issues

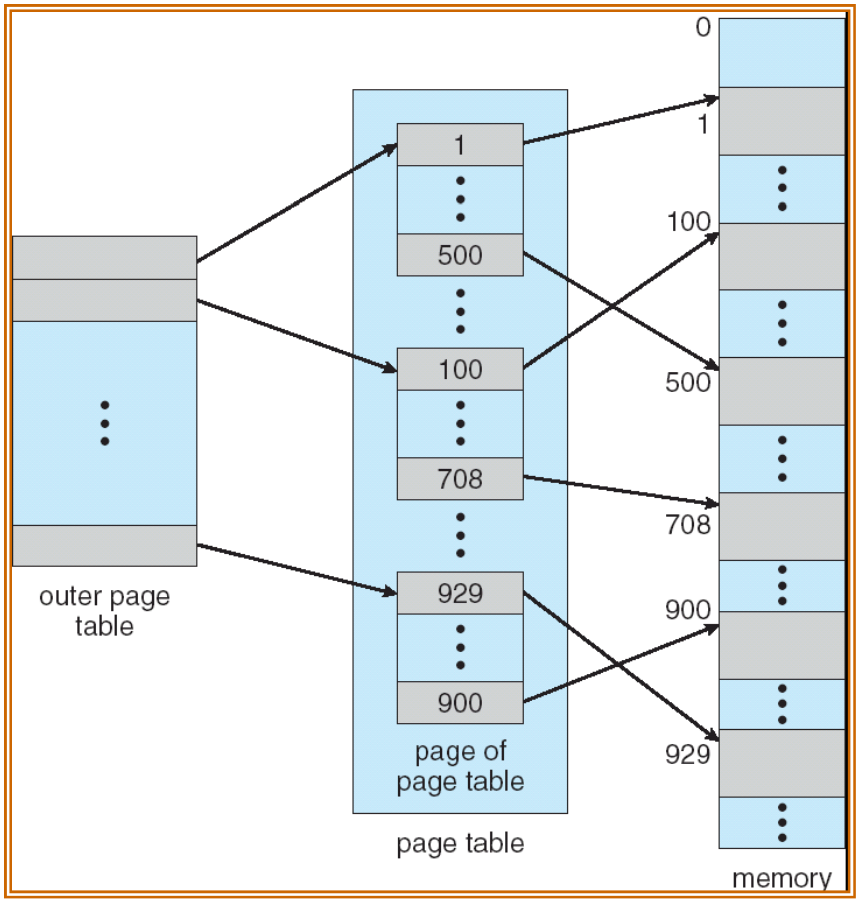
- Given:
  - A 32 bit address space (4 GB)
  - 4 KB pages
  - A page table entry of 4 bytes
- Implication: page table is 4 MB per process!
- Observation: address space are often sparse
  - Few programs use all of 2^32 bits
- Change page table structures to save memory

Page table structures

- Hierarchical paging
- Hashed page tables
- Inverted page tables

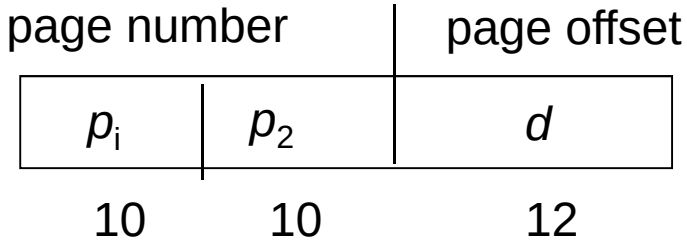
Hierarchical page table

- Break up virtual address space into multiple page tables at different levels

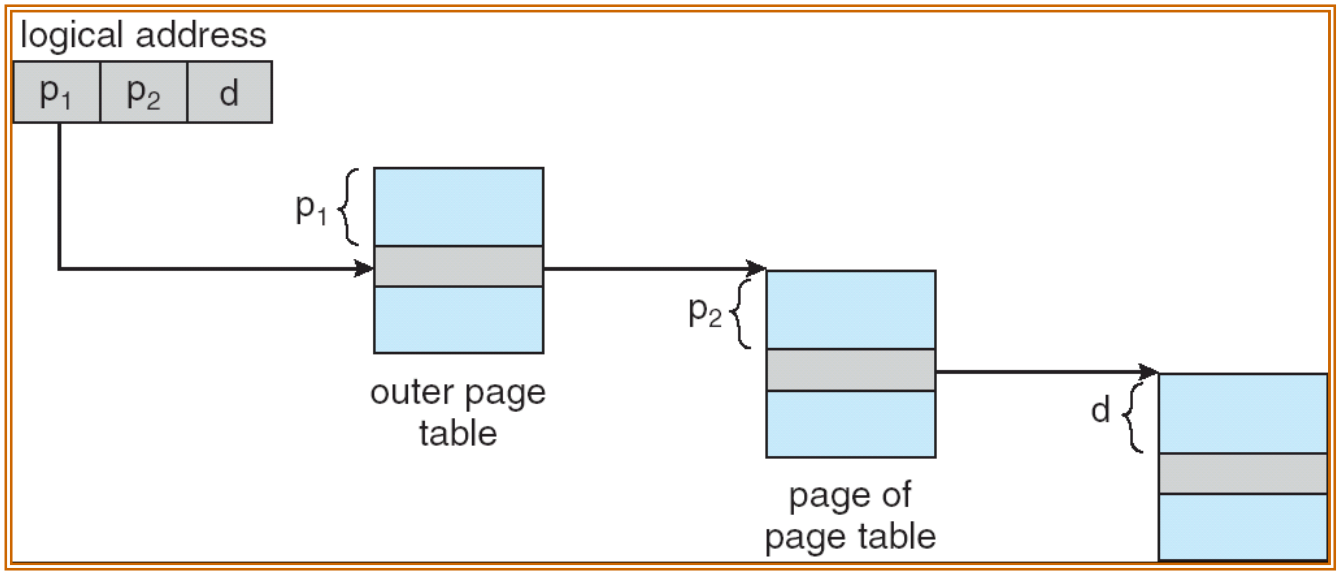


Two-level paging example

- 32-bit address space, 4 KB page
  - 4KB page ➔ 12 bit page offset
- How many bits for 2nd-level page table?
  - Desirable to fit a 2nd-level page table in one page
  - 4KB/4B = 1024 ➔ 10 bit address for 2nd-level page table
- Address bits for top-level page table: 32 – 12 – 12 = 10



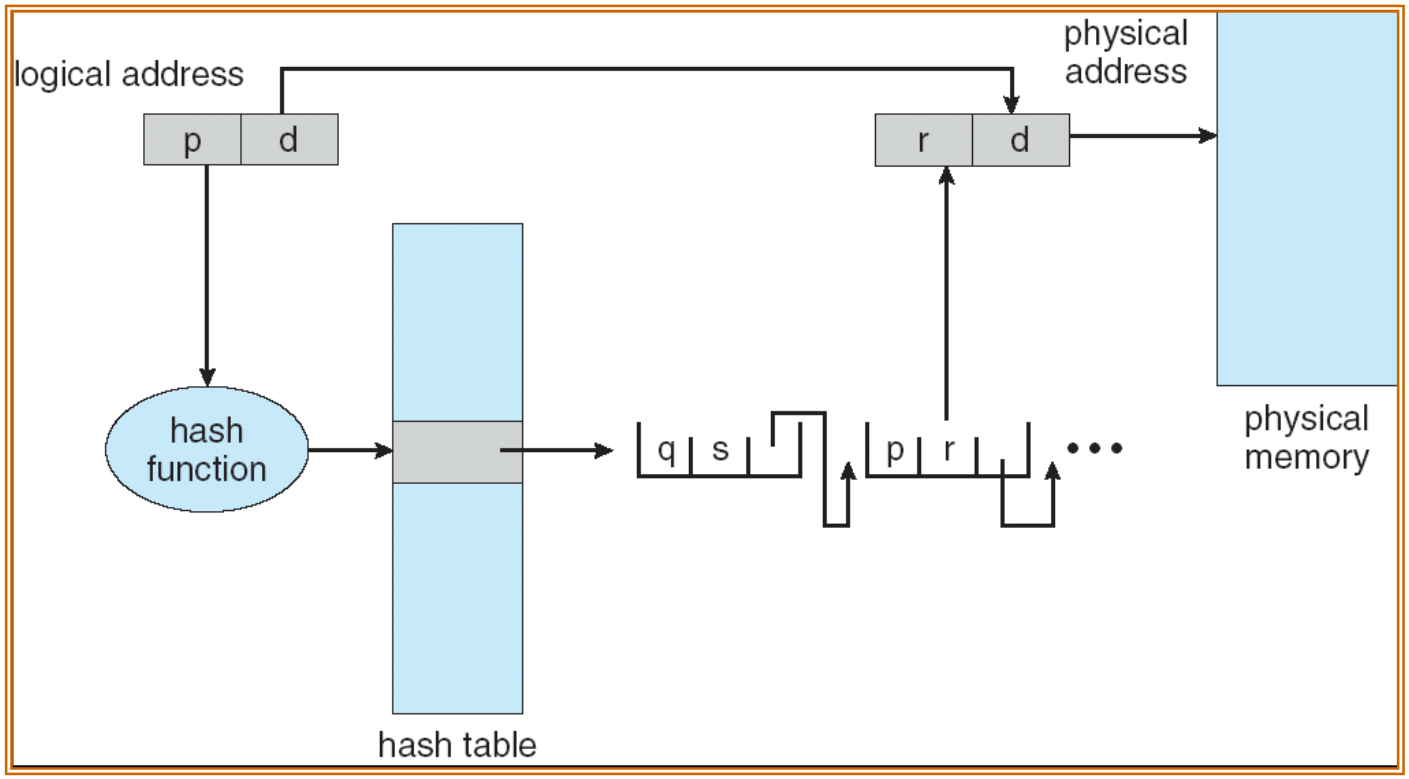
Address-translation scheme



Hashed page table

- Common in address spaces > 32 bits
- Page table contains a chain of elements hashing to the same location
- On page translation
  - Hash virtual page number into page table
  - Search chain for a match on virtual page number

Hashed page table example



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
- Trade translation time for page table space
- Can use hash table to limit the search to one or at most a few page-table entries

Inverted page table example