# CS343: Operating System

# Memory Management

## Lect32 : 30th Oct 2023

**Dr. A. Sahu**

**Dept of Comp. Sc. & Engg.**

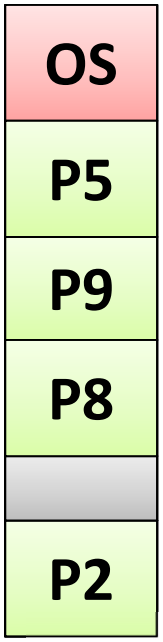
**Indian Institute of Technology Guwahati**

# Outline

- Memory Management
  - Paging
  - Virtual memory
  - Demand Paging
  - Page Replacement
  - Frame Allocation

# Memory Allocation: Top Down

Assume 4GB of RAM,
Avg Process Size 200MB
4 Segments/Process




Finding Hole
First Fit
Worst fit
Best Fit

Very low complexity

Very High Mem Wastage

Process Continuous Allocation
10 process and 20 holes

Segment continuous Allocation
40 segment and 80 holes

low complexity

High Memory Wastage

Paging: 200MB process, 4KB page
Two Issue: pages and frames

Very Low Mem Wastage

High Complexity: 51200 pages/Process

Loading 51K pages in $10^6$ frames

High Complexity: 4GB/4KB=$2^{20}$=$10^6$ frames

SOL

Demand Paging Load the required 50-100 pages and load as an when necessary

4GB paged

SOL

Allocate 100-200 frames per process: reduce search Space

# Paging

# Paging

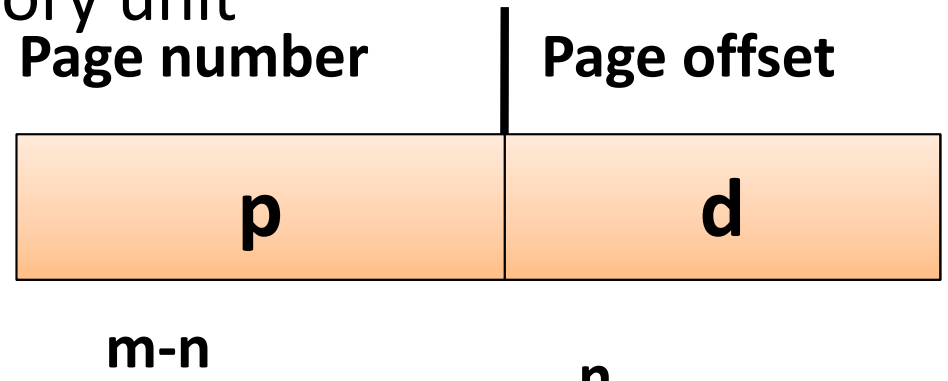
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes , **getpagesize() in C → Demo**

# Paging

- Divide logical/program 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
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

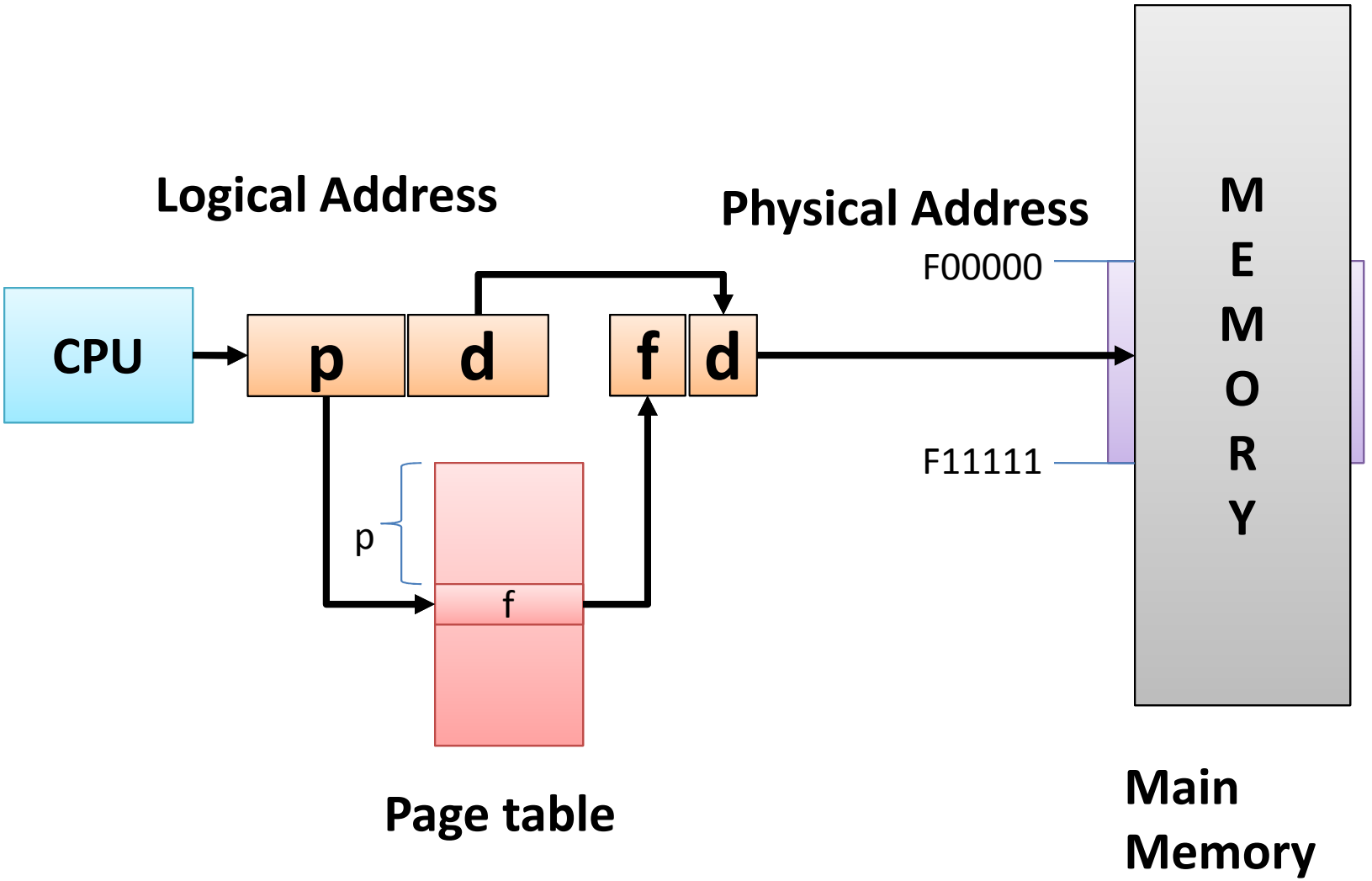
# Address Translation Scheme

- 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

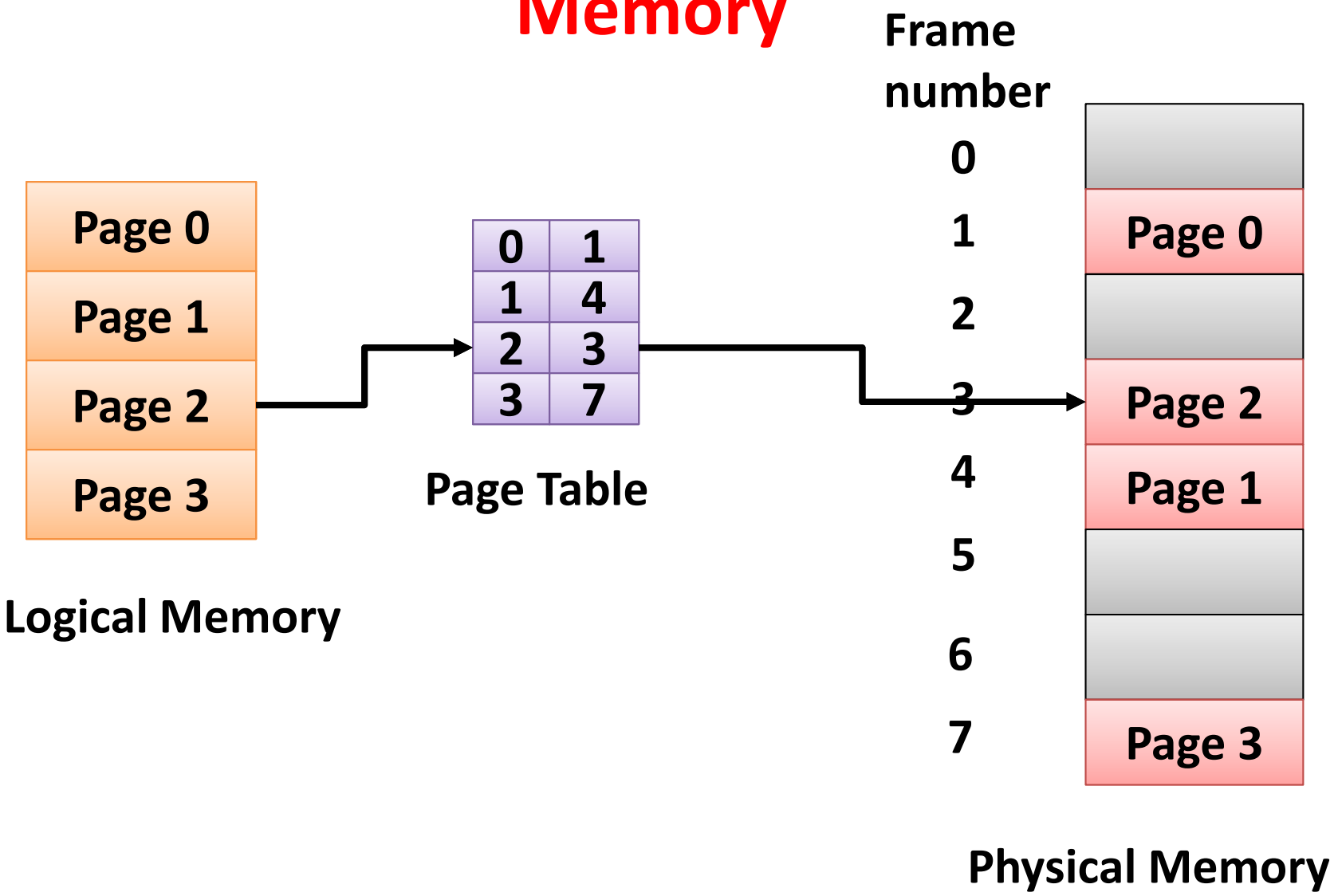



  - For given logical address space $2^m$ and page size $2^n$

# Paging Hardware



# Paging Model of Logical and Physical Memory



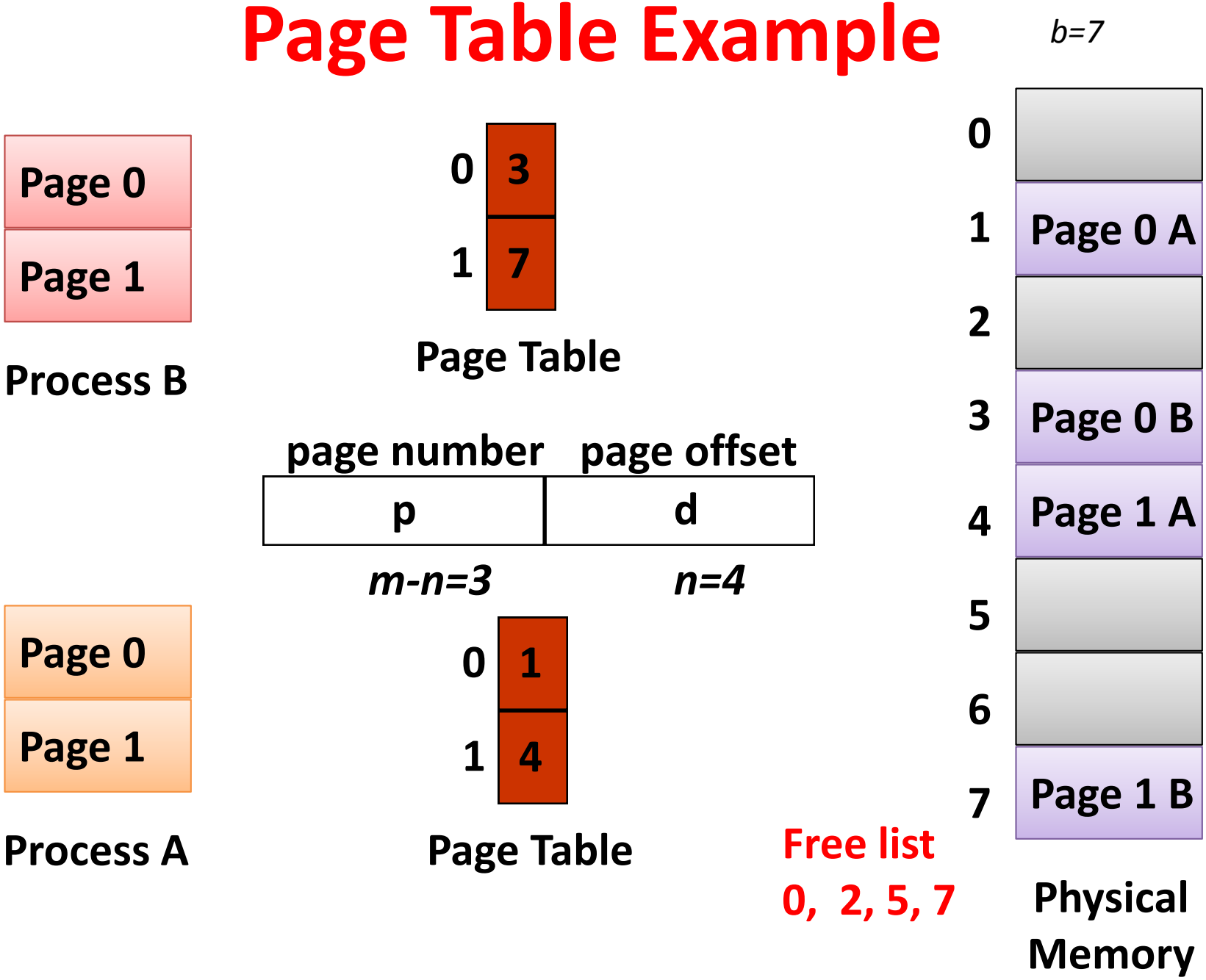
# Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte

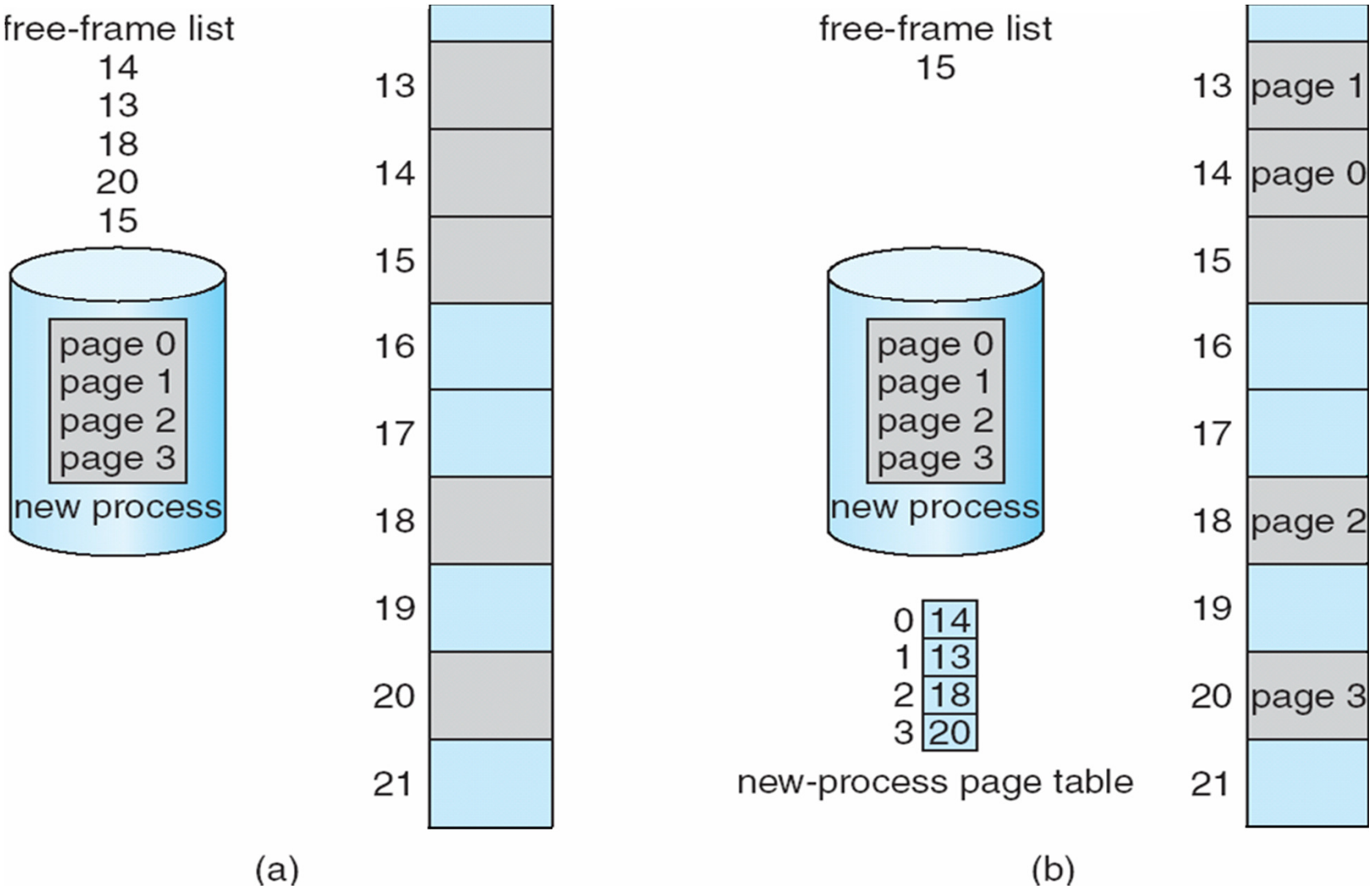
# Paging (Cont.)

- Calculating internal fragmentation Ctd..
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

# Page Table Example



# Free Frames



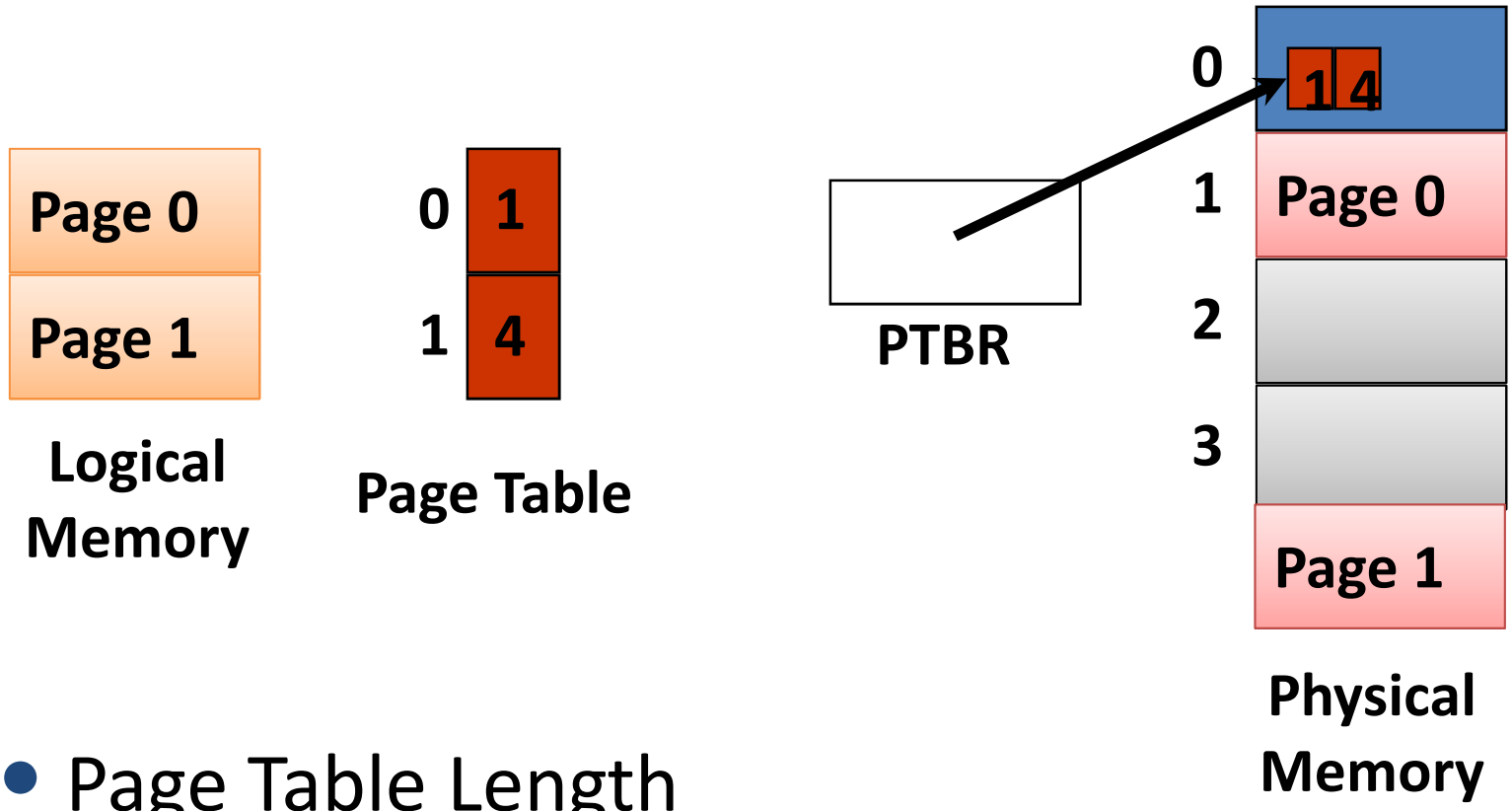

Before allocation

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** (**PTLR**) 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

# Implementation of Page Table




- Page Table Length
- **Two memory accesses per data/inst access.**
  - **Solution? *Associative Registers/CAM***

# Implementation of Page Table (Cont.)

- 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**)
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch

# Implementation of Page Table (Cont.)

- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

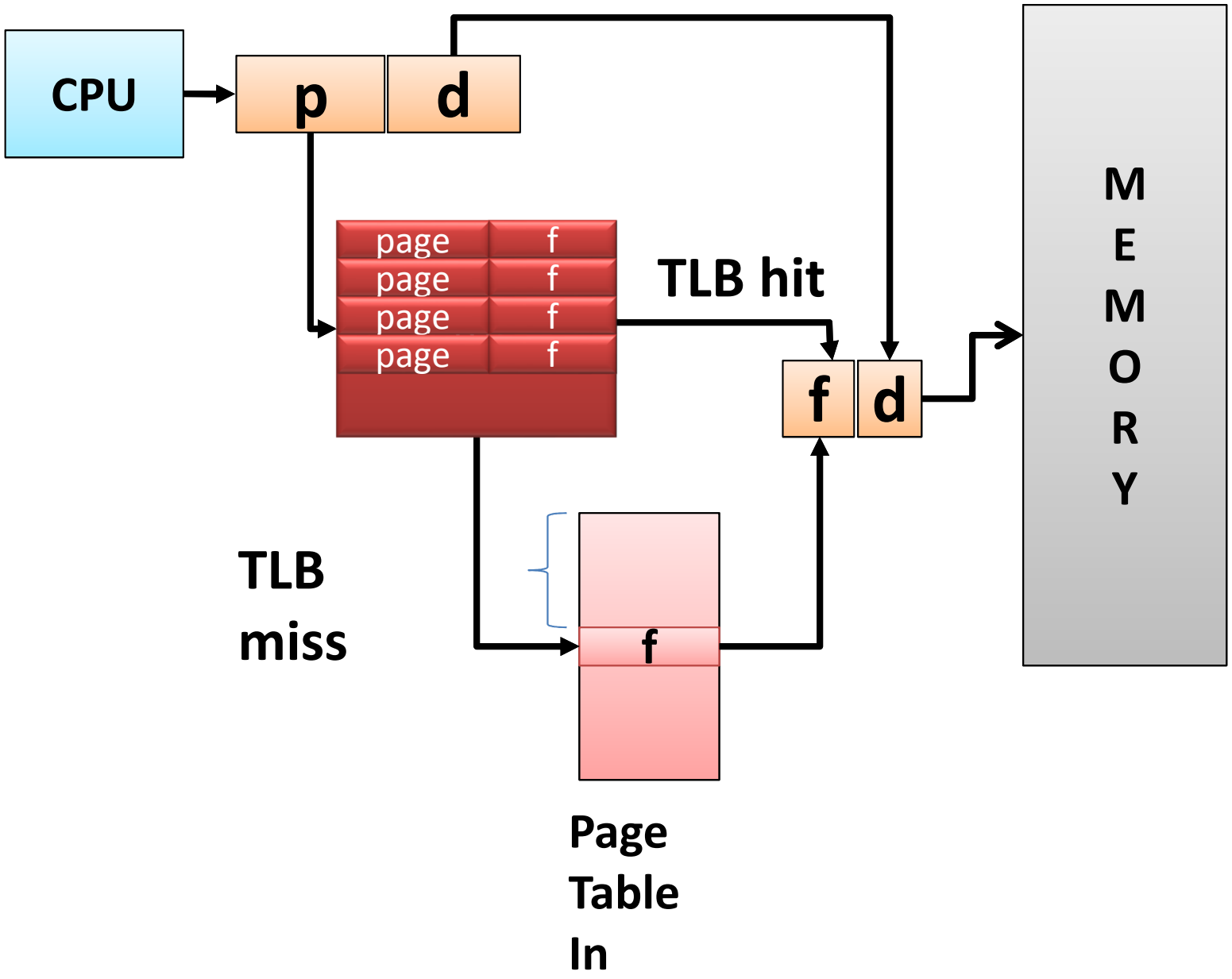
# Associative Memory

- Associative memory – parallel search

| Page # | Frame # |
|---|---|
| 1 | 2 |
| 3 | 15 |
| 2 | 12 |
| | |

- CAM : Content Addressable Memory
- 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 = $\varepsilon$ time unit
  - Can be < 10% of memory access time
- Hit ratio = h
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider h = 80%, $\varepsilon$ = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

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

$$= 2 + \varepsilon - h$$

# Effective Access Time

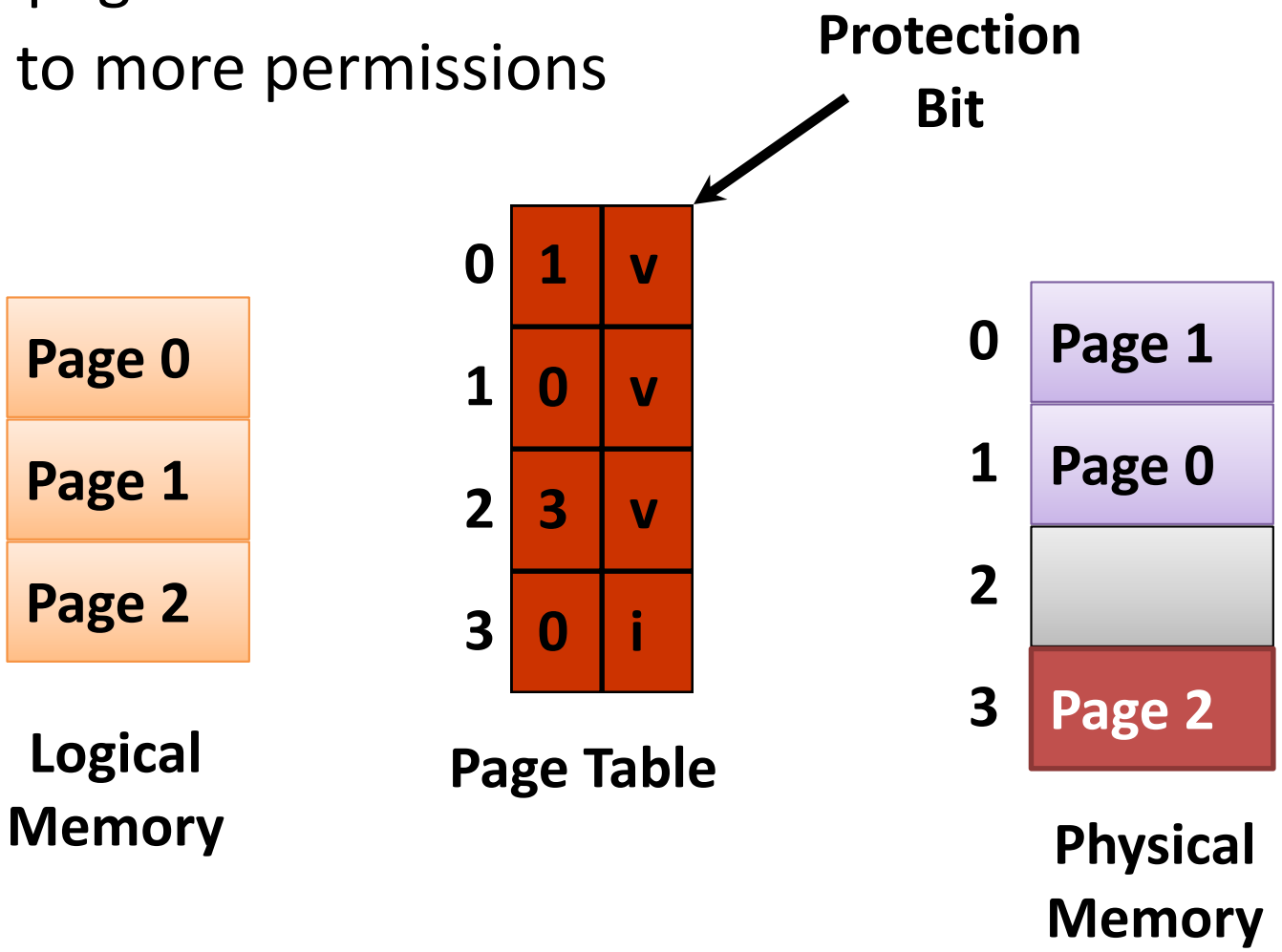
- Consider h = 80%, $\varepsilon$ = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x (100+20) + 0.20 x (200+20) = 140ns
- Consider more realistic hit ratio -> h = 99%, $\varepsilon$ = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x (120) + 0.01 x 220 = 121ns

# Memory Protection

- Associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space: a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

# Protection

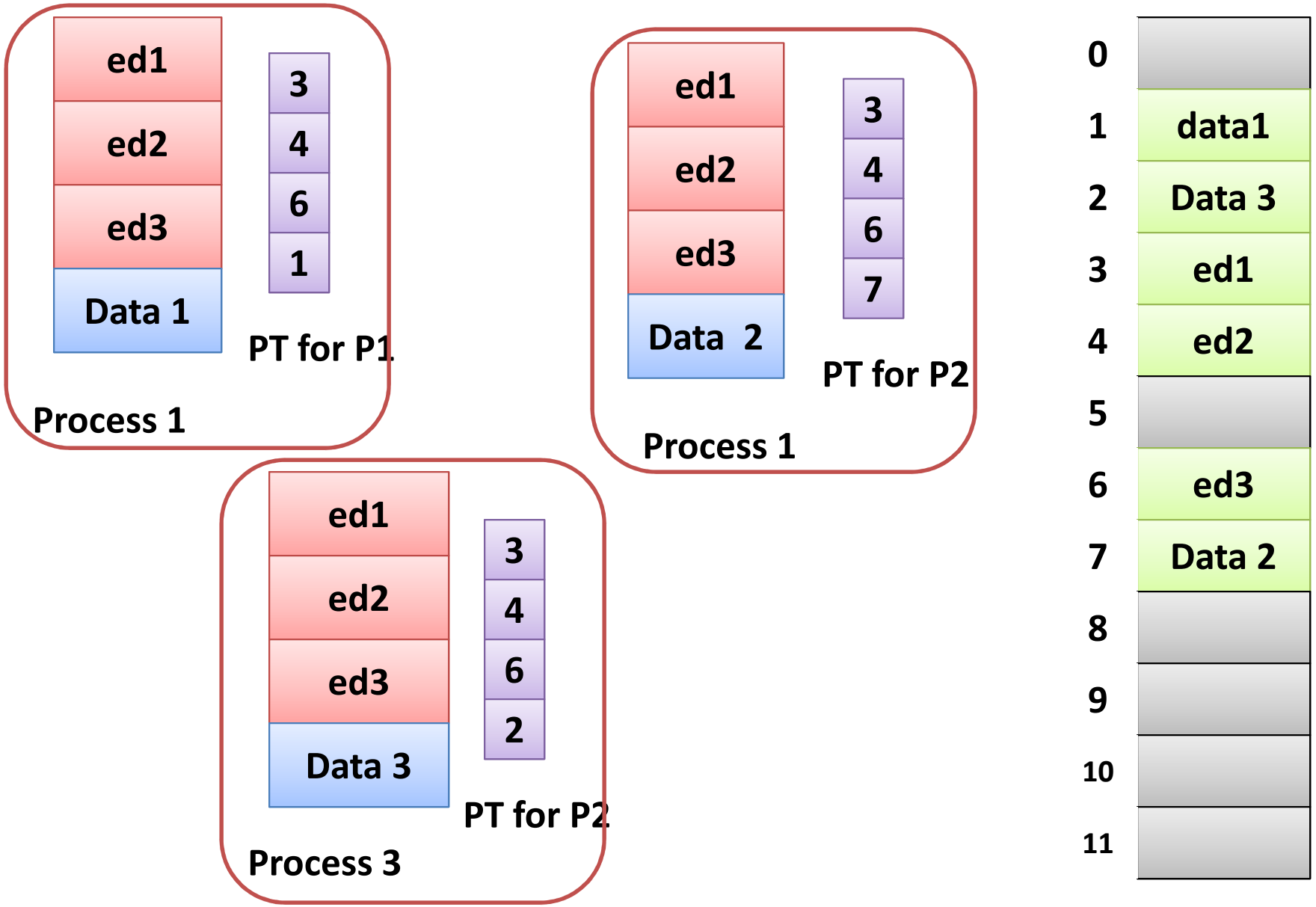
- Protection bits with each frame
- Store in page table
- Expand to more permissions



# Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes
    - (i.e., text editors, compilers, window systems)
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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



# Large Address Spaces

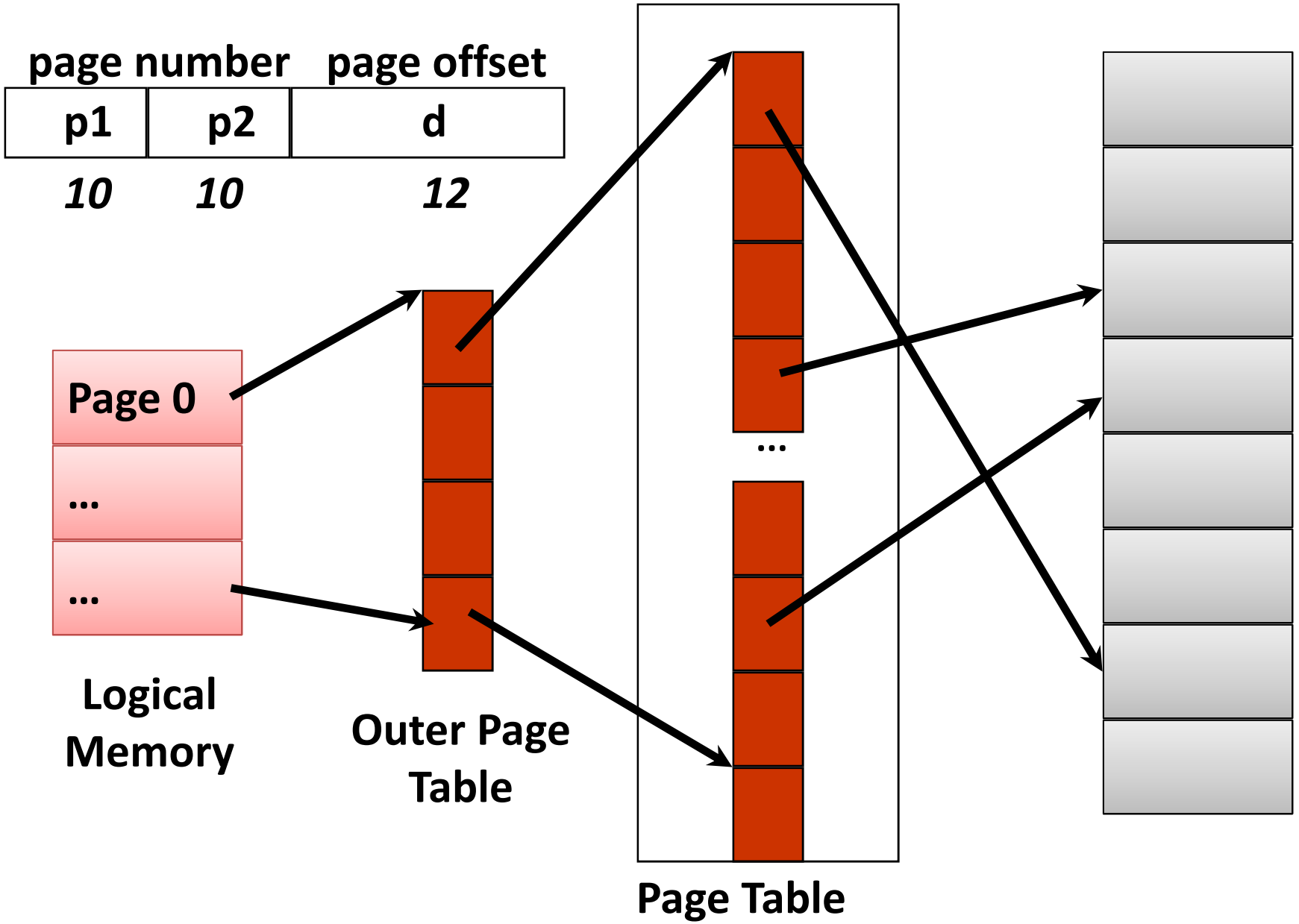
- Typical logical address spaces:
  - 4 Gbytes => $2^{32}$ address bits (4-byte address)
- Typical page size:
  - 4 Kbytes = $2^{12}$ bits
- Page table may have:
  - $2^{32} / 2^{12} = 2^{20}$ = 1million entries
- Each entry 4 bytes => 4MB per process!

  **Linux command "$ps –A |wc –l" show 400 process running most of the time**
- **Do not want that all in RAM**
  - **400*4MB=1.6GB of RAM will be occupied by Page Tables**
- **Solution?**
  - **Page the Page table (Multilevel paging), Hashing, and Inverted Page Table**

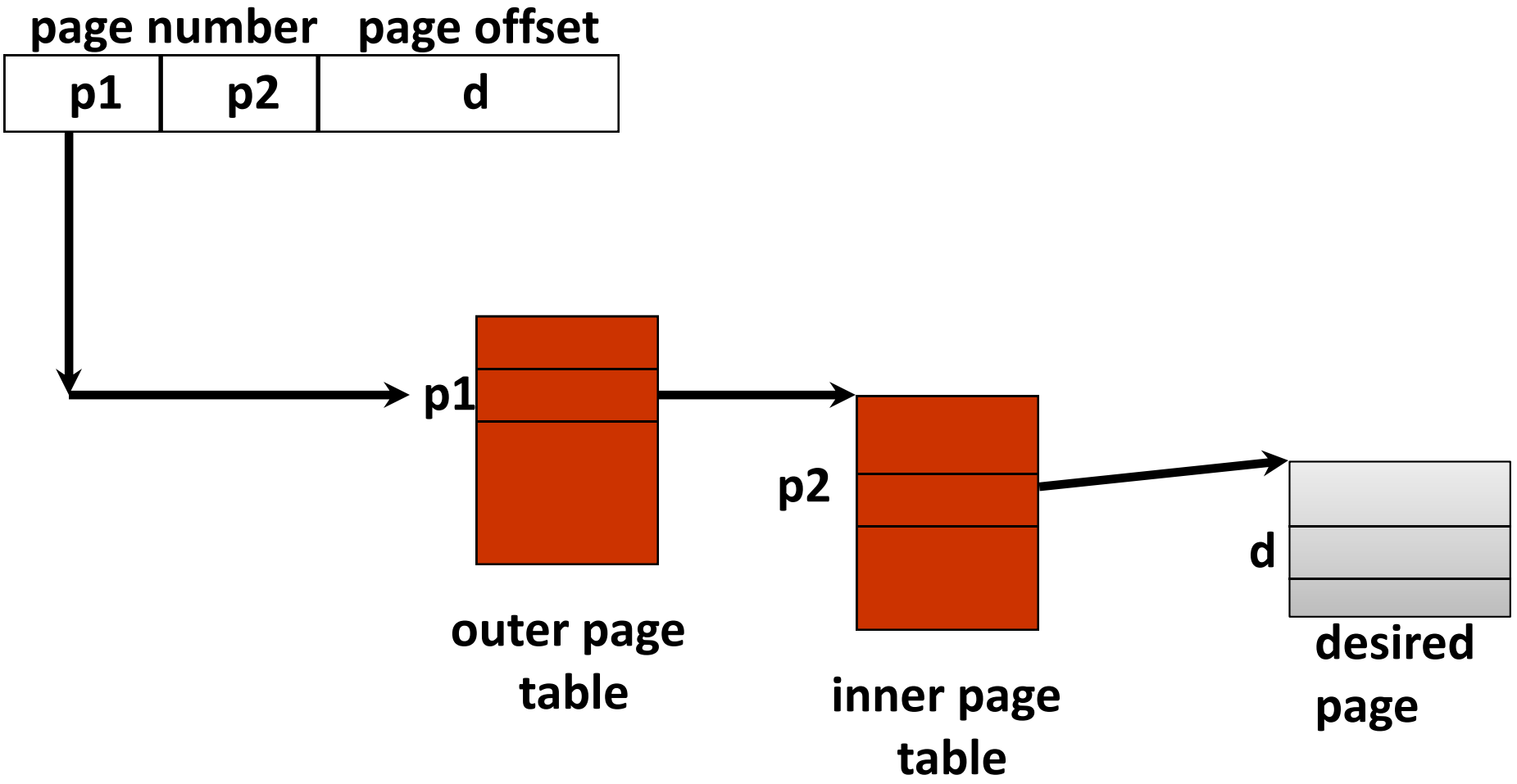
# Large Address Spaces

- Do not want that all 4MB  in RAM
- Solution?
  - Page the Page table (Multilevel paging)
    - We would not want to allocate PT contiguously on memory, divide the PT in to smaller pieces (ML paging)
  - Hashing: if some pages are not to be used of a process is beneficial
  - Inverted Page Table

# Multilevel Paging



# Multilevel Paging Translation



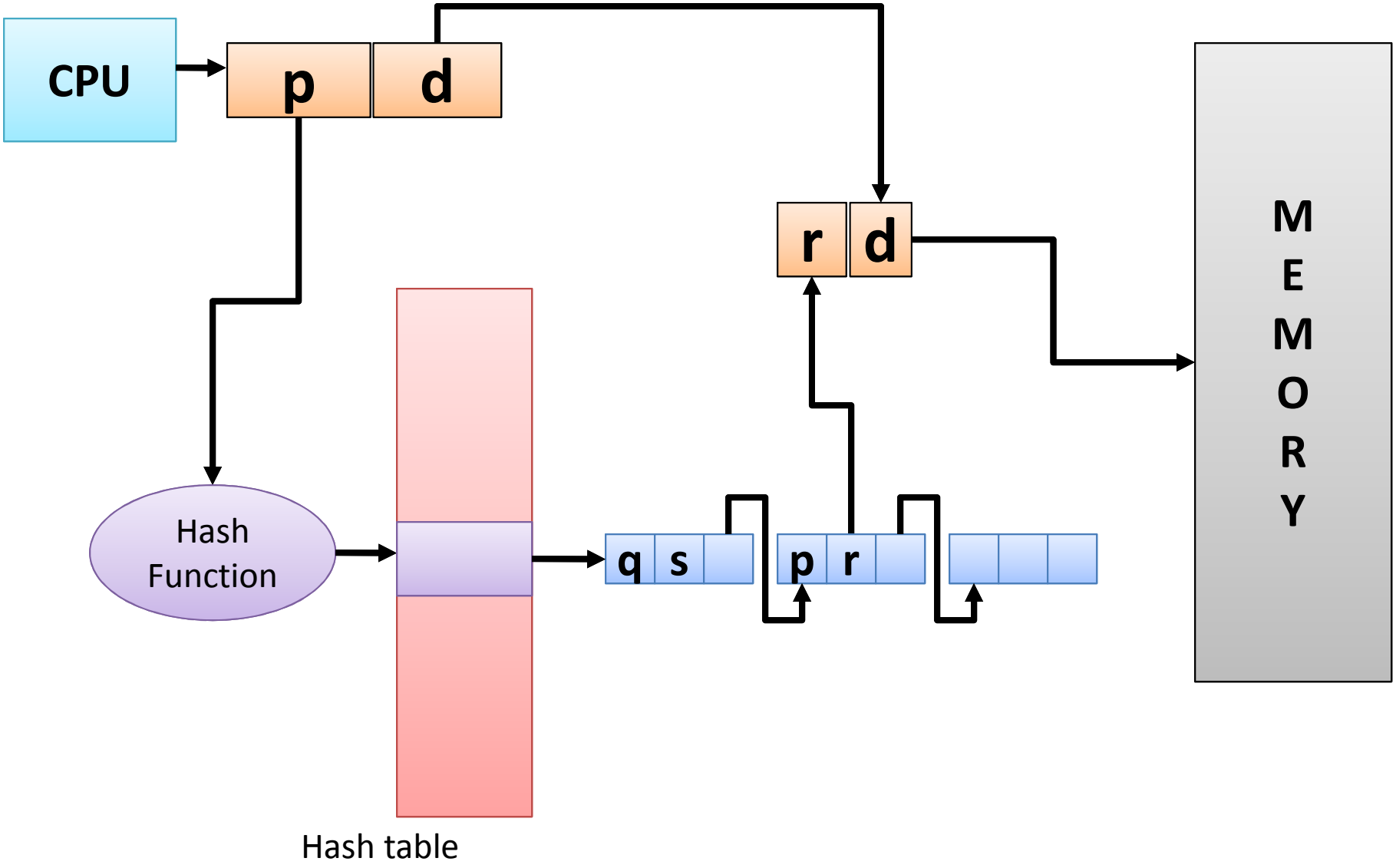
# Hashed Page Tables

- Common in address spaces > 32 bits
  - **Modern Ubuntu and Windows uses 48bit**
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  - The virtual page number
  - The value of the mapped page frame
  - A pointer to the next element

# Hashed Page Tables

- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

# Hashed Page Table



# Inverted Page Table

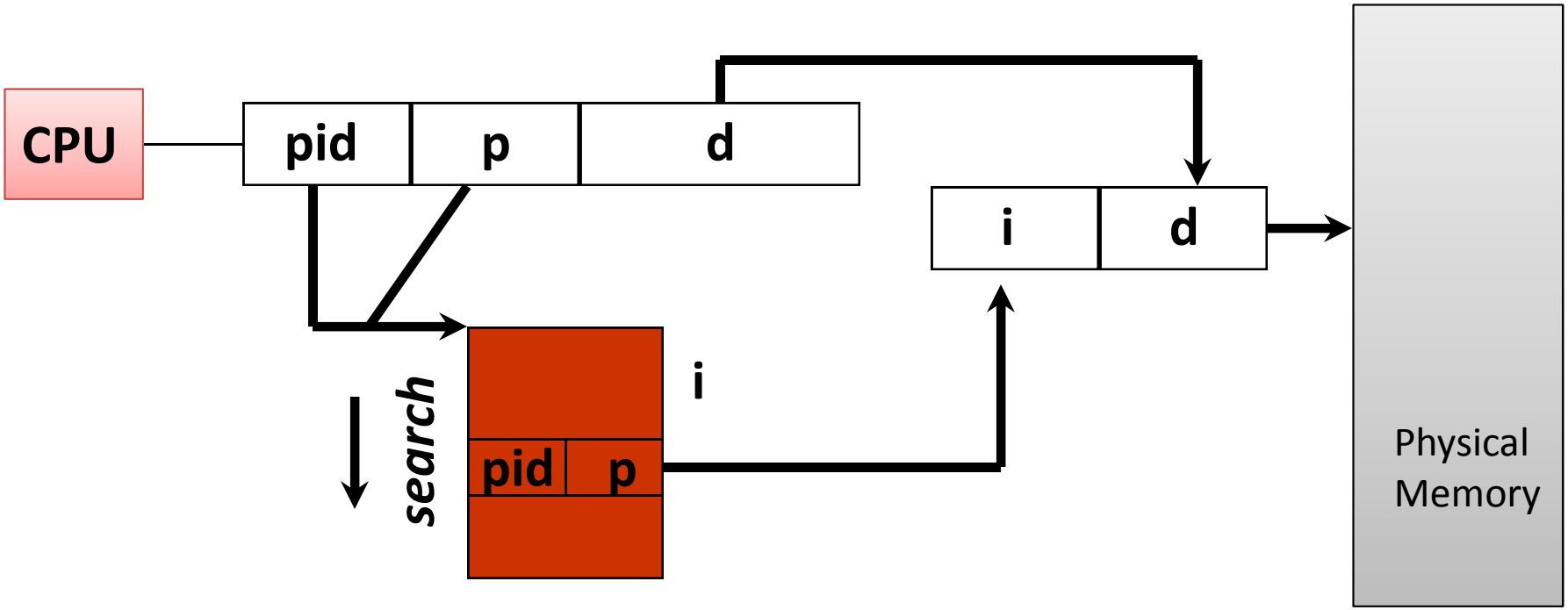
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page/frame of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

# Inverted Page Table

- 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
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

# Inverted Page Table

- Page table maps to physical addresses




- Still need page per process --> backing store
- Memory accesses longer! (search + swap)

# Thanks