OPERATING SYSTEMS

Memory Management

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Structure of the page table

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Hardware Support for Page Table

- The hardware implementation of page table can be done by using dedicated registers.
- But the usage of register for the page table is satisfactory only if page table is small.
- If page table contain large number of entries then we can use TLB(translation Look-aside buffer), a special, small, fast look up hardware cache.

OPERATING SYSTEMS

Hardware Support for Page Table

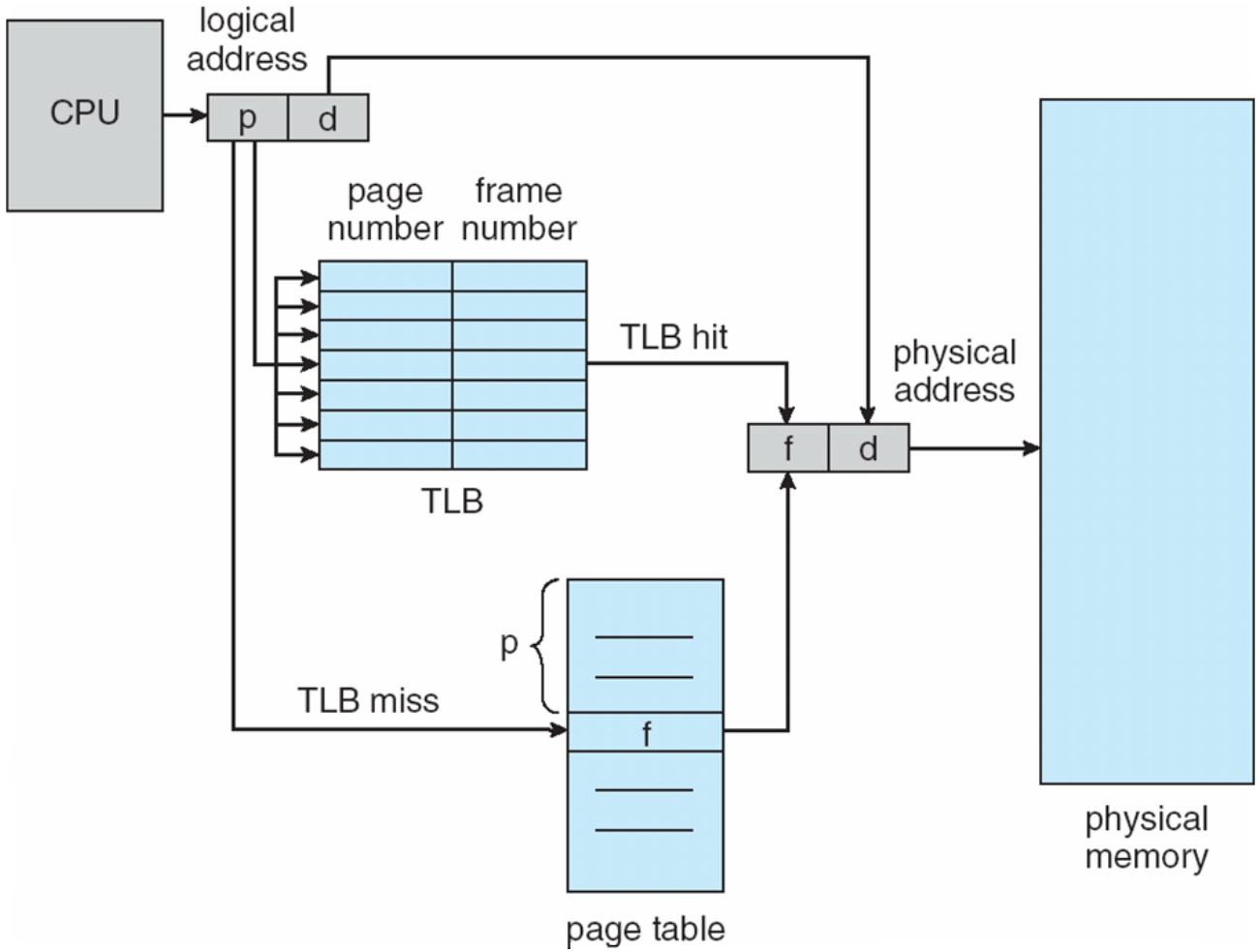
- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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**)

Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry
- This identifier is stored with each TLB entry to distinguish between entries loaded for different processes.
- Otherwise need to flush at every context switch
- 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

OPERATING SYSTEMS

Paging Hardware With TLB



OPERATING SYSTEMS

Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns i.e. if we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 ns) and then access the desired byte in memory (10 nanoseconds), for a total of 20 ns (assuming that a page table lookup takes only one memory access).
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

Translation Look-Aside Buffer

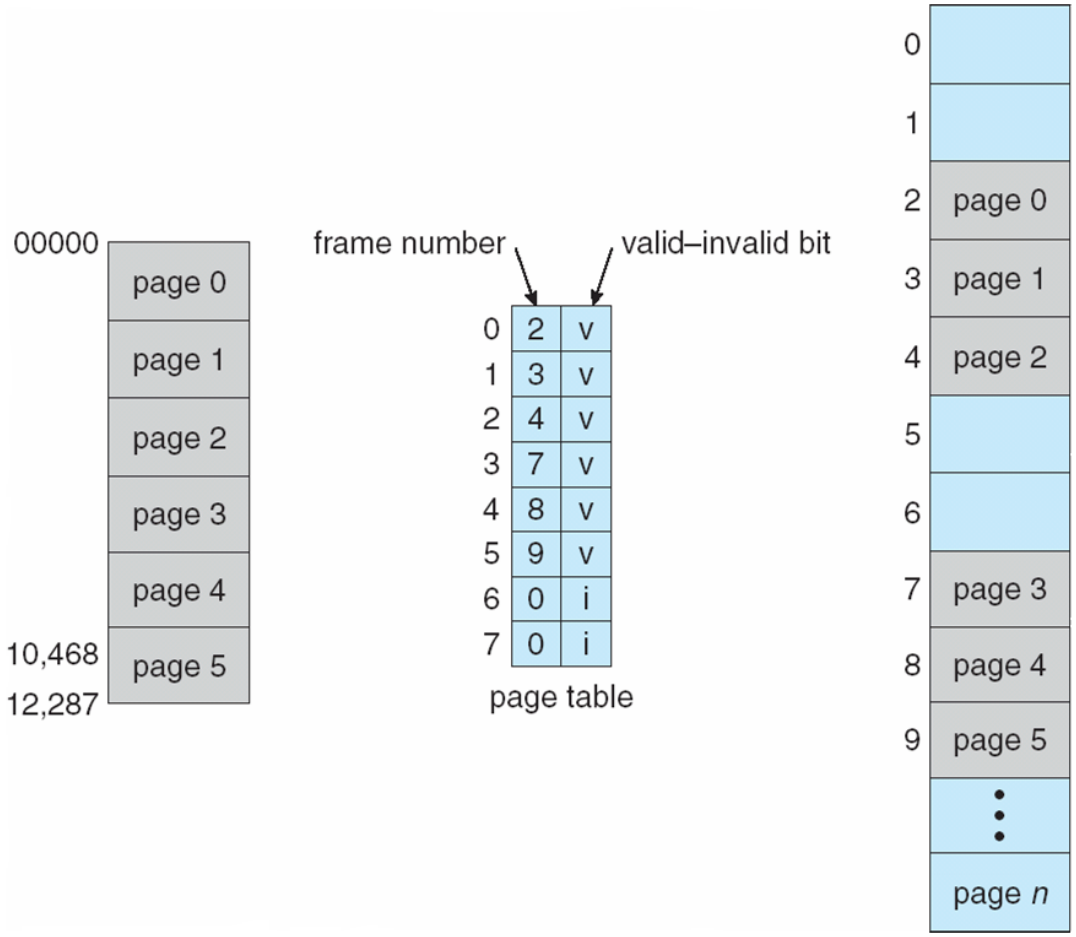
How TLB is different from CPU Cache

- TLB is about ‘speeding up address translation for Virtual/logical memory’ so that page-table needn’t to be accessed for every address.
- CPU Cache is about ‘speeding up main memory access latency’ so that RAM isn’t accessed always by CPU.
- TLB operation comes at the time of address translation by MMU while CPU cache operation comes at the time of memory access by CPU.

Memory Protection

- Memory protection implemented by 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, and is thus 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

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



OPERATING SYSTEMS

Shared Pages

- An advantage of paging is the possibility of sharing common code.
  - important in a time-sharing environment
- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150 KB of code and 50 KB of data space.
- we need 8,000 KB(150 KBX40+50KB X40) to support the 40 users.
- Can this memory space usage be reduced?
  - YES
- If the code is shared, then a total space required will be 2,150 KB

Shared Pages

Shared code

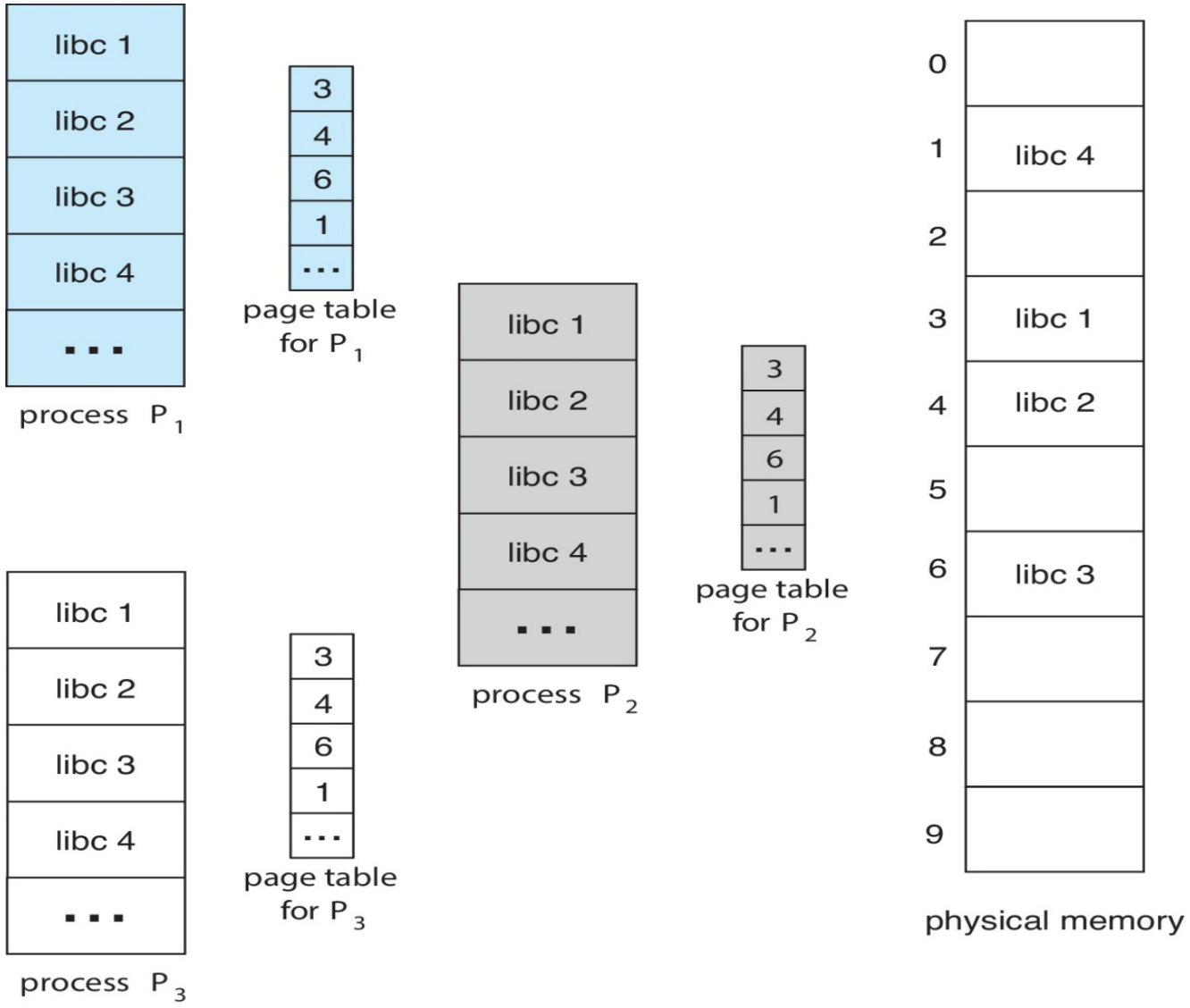
- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Reentrant (multi-instance) code is a reusable routine that multiple programs can invoke, interrupt, and reinvoke simultaneously.
- Reentrant code is non-self-modifying code: it never changes during execution.
- Similar to multiple threads sharing the same process space
- 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

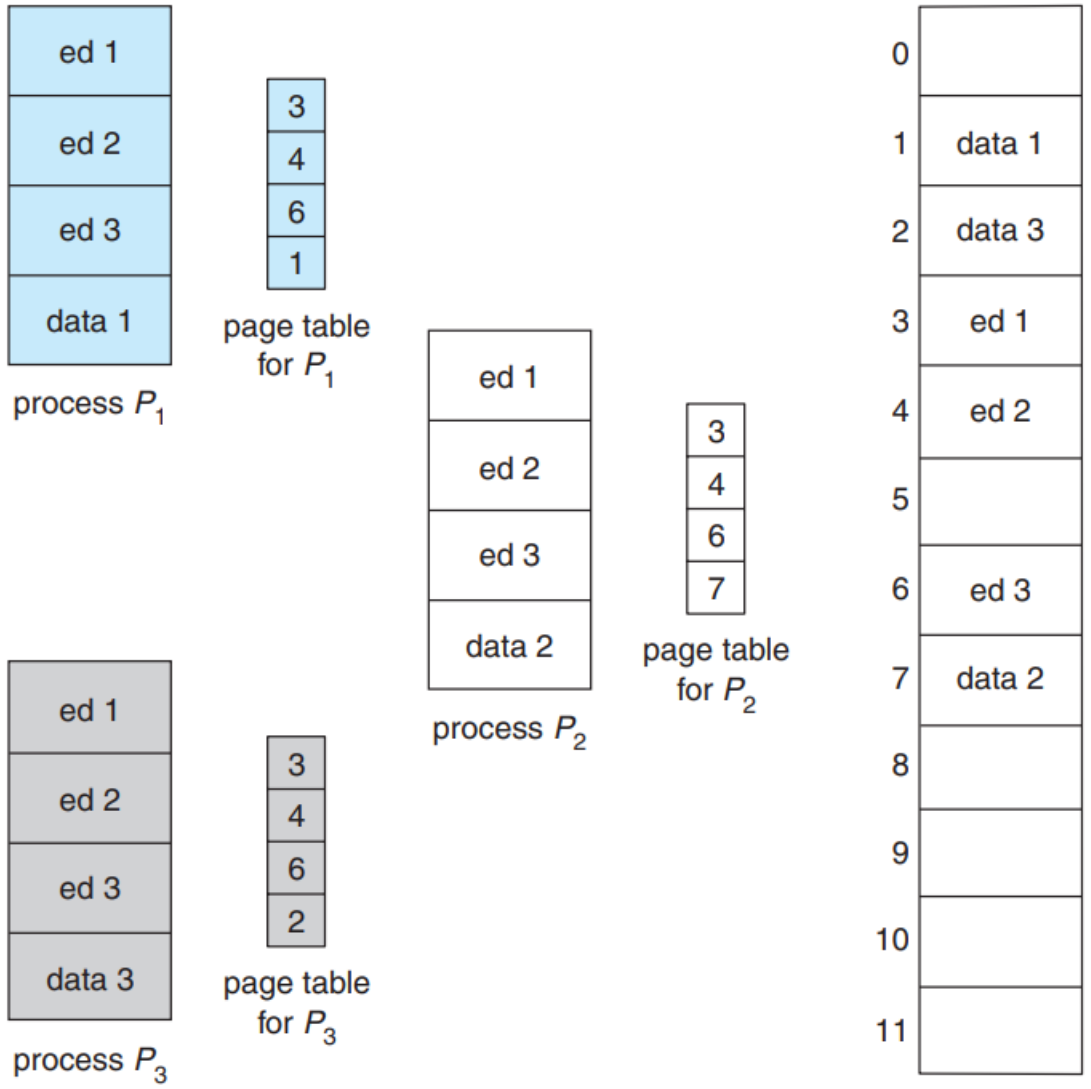
OPERATING SYSTEMS

Shared Pages Example



OPERATING SYSTEMS

Shared Pages Example

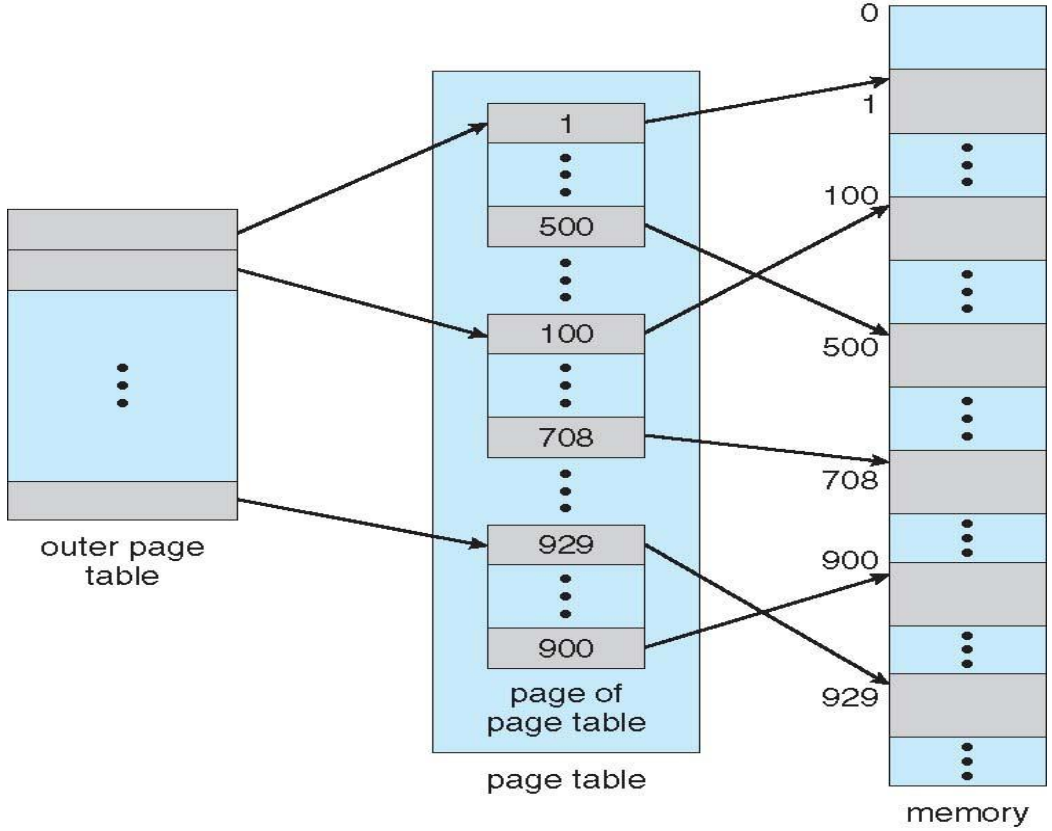


Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

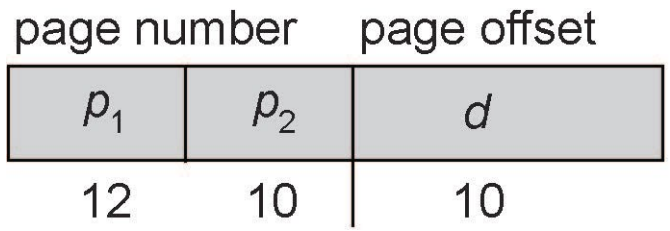
Hierarchical Page Tables

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



Two-Level Paging Example

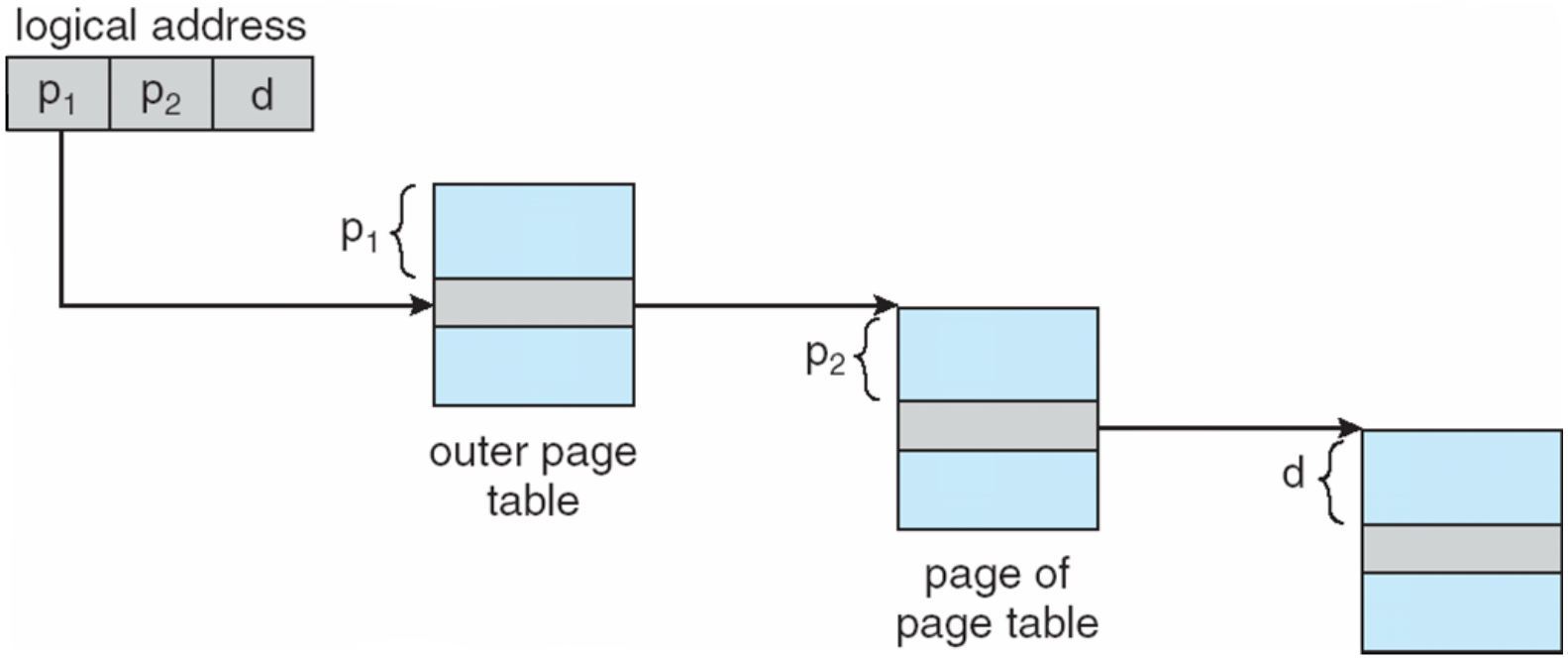
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address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



where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

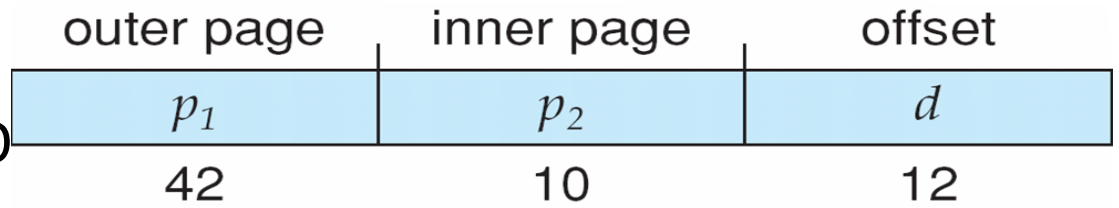
Address-Translation Scheme

- This scheme is known as **forward-mapped page table as** address translation works from the outer page table inward.



64-bit Logical Address Space

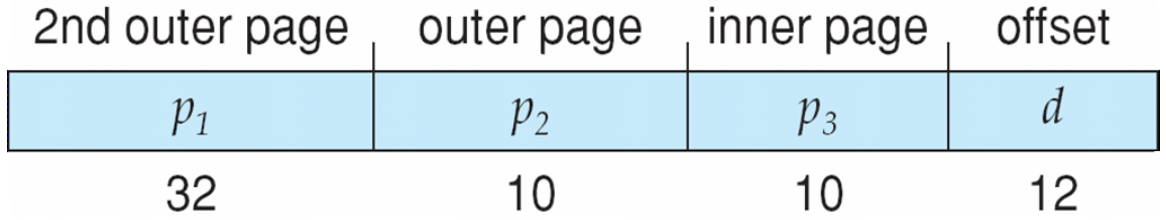
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - O
  - One solution is to add a 2nd outer page table

Three-level Paging Scheme

- We can divide the outer page table in various ways.
- For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes).
- In this case, a 64-bit address space is still daunting:



The outer page table is still 2^{34} bytes (16 GB) in size. And possibly 4 memory access to get to one physical memory location

- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.
- The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses— to translate each logical address. So, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

Examples

Consider that the size of main memory (physical memory) is 512 MB and frame size of 4KB. Calculate

a.Page offset(d)

b.Bits to address page number/Frame number

c.What will be the total number of frames

d.Number of level of paging

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number (VPN) is hashed into a page table
  - Each entry in the page table contains a linked list of elements that hash to the same location (to handle collisions)
- Each element contains three fields:

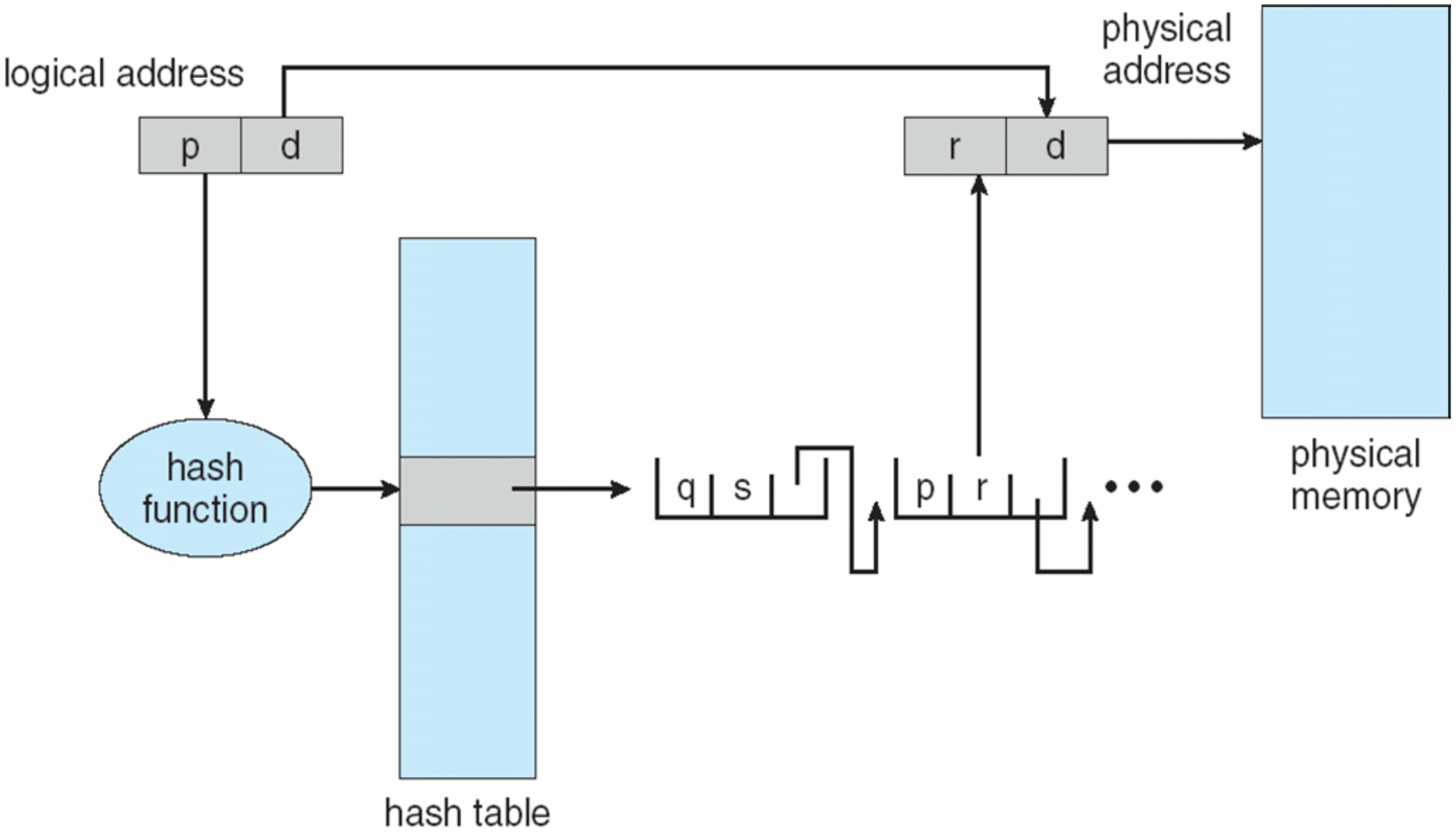
  (1) the virtual page number

  (2) the value of the mapped page frame

  (3) a pointer to the next element
- Virtual page numbers are compared (with field 1) in this chain searching for a match
  - If a match is found, the corresponding physical frame (field 2) is extracted
  - If there is no match, subsequent entries in the linked list are searched for a matching VPN

OPERATING SYSTEMS

Hashed Page Table



OPERATING SYSTEMS

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 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
- 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

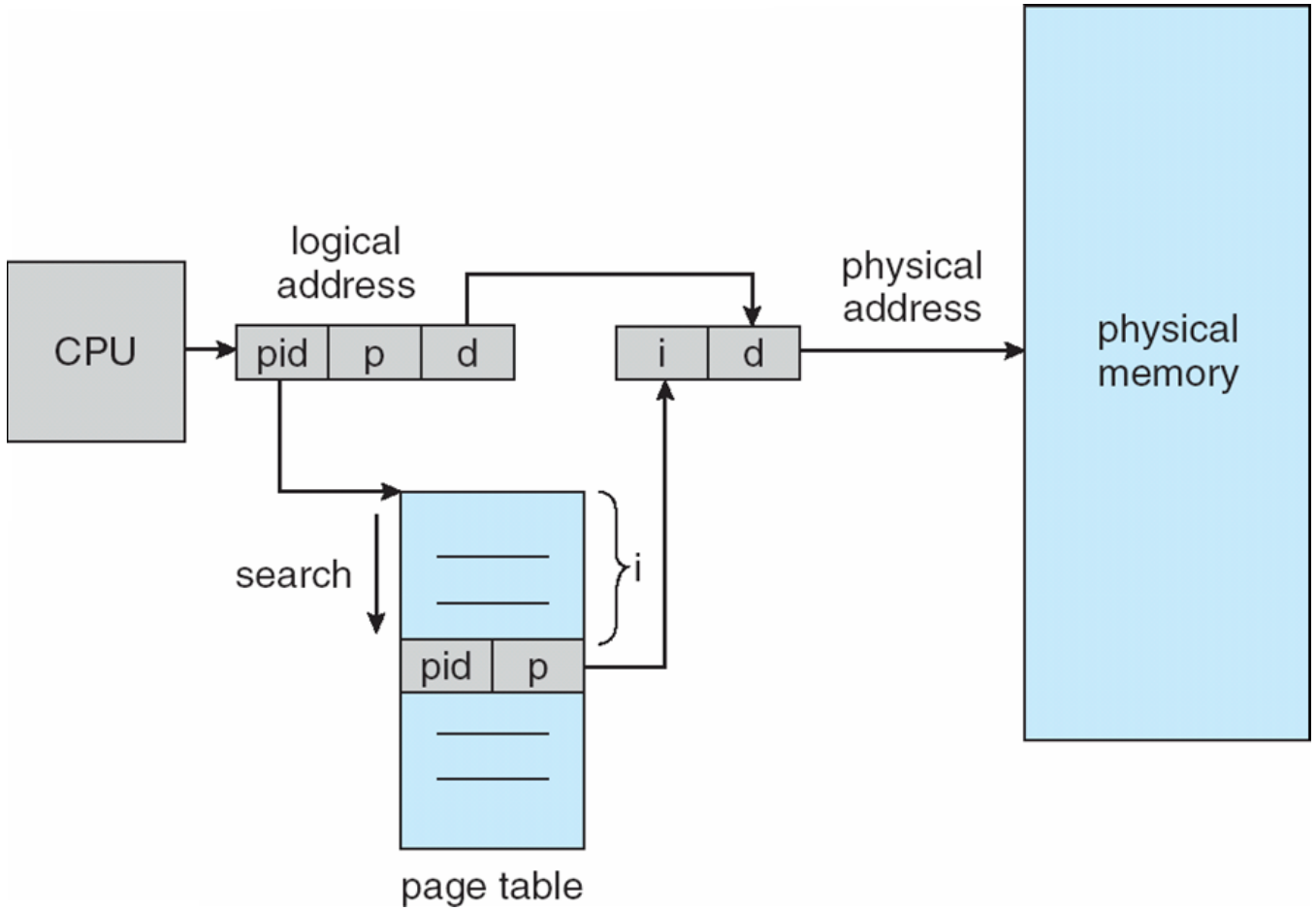
Inverted Page Table

- A simplified version of the inverted page table used in the *IBM RT*.
- IBM was the first major company to use inverted page tables
  - starting with the IBM System 38 to RS/6000 and the current IBM Power CPUs.
  - For the IBM RT, each virtual address in the system consists of a triple:
    - <process-id, page-number, offset>.

Inverted Page Table

- Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier.
- When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, *t*he inverted page table is then searched for a match.
- If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated.
- If no match is found, then an illegal address access has been attempted.

Inverted Page Table Architecture



Used in 64-bit UltraSPARC (from Sun Micro Systems) and PowerPC (created by Apple-IBM-Motorola alliance)

Inverted Page Table

- This scheme decreases the amount of memory needed to store each page table.
- It increases the amount of time needed to search the table when a page reference occurs.
- The inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found.
- Hash table is used to over come this problem where the number of search's are limited one or at most few.
- each access to the hash table adds a memory reference to the procedure.
- One virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table

Inverted Page Table

- Systems that use inverted page tables have difficulty in implementing shared memory.
- Shared memory is usually implemented as multiple virtual addresses are mapped to one physical address.
- There is only one virtual page entry for every physical page
- One physical page cannot have two (or more) shared virtual addresses.
- A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address.
- references to virtual addresses that are not mapped result in page faults

Page table example

Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a. A conventional single-level page table

b. An inverted page table

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu