

CMPT 300
Operating System I
Memory Management - Chapter 9

Dr. Hazra Imran

Summer 2022

Admin notes

Assignment 4 (Due Date July 30, 2022@ 11:59 PM)

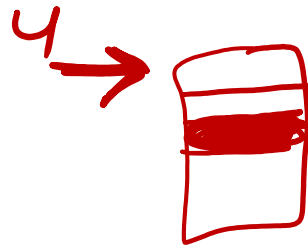
- Step 1: Read the [Assignment 4 instructions \(Links to an external site.\)](#)
- Step 2: [A4 Github Submission Link to submit your files. \(Links to an external site.\)](#)
- Step 3: [Assignment 4 - Reflection Survey + Late claim days \(Aug 4\)](#)

Address Translation with Paging

- For each process, there is an address A

$$\text{page number} = A / \text{page_size}$$

$$\text{offset} = A \bmod \text{page_size}$$



- This is the page number within the process address space.
- e.g., address in the process, A = 10,000 and page size = 4k
- page number
= 10000 / 4k
= 10,000 / 4096
= 2.xxx = truncate to 2

- this is the distance from the beginning of the page
- page offset
= 10000 mod 4k
= 10,000 mod 4096
= 1908 ✓

Think

Ques. What if the page size is very large?

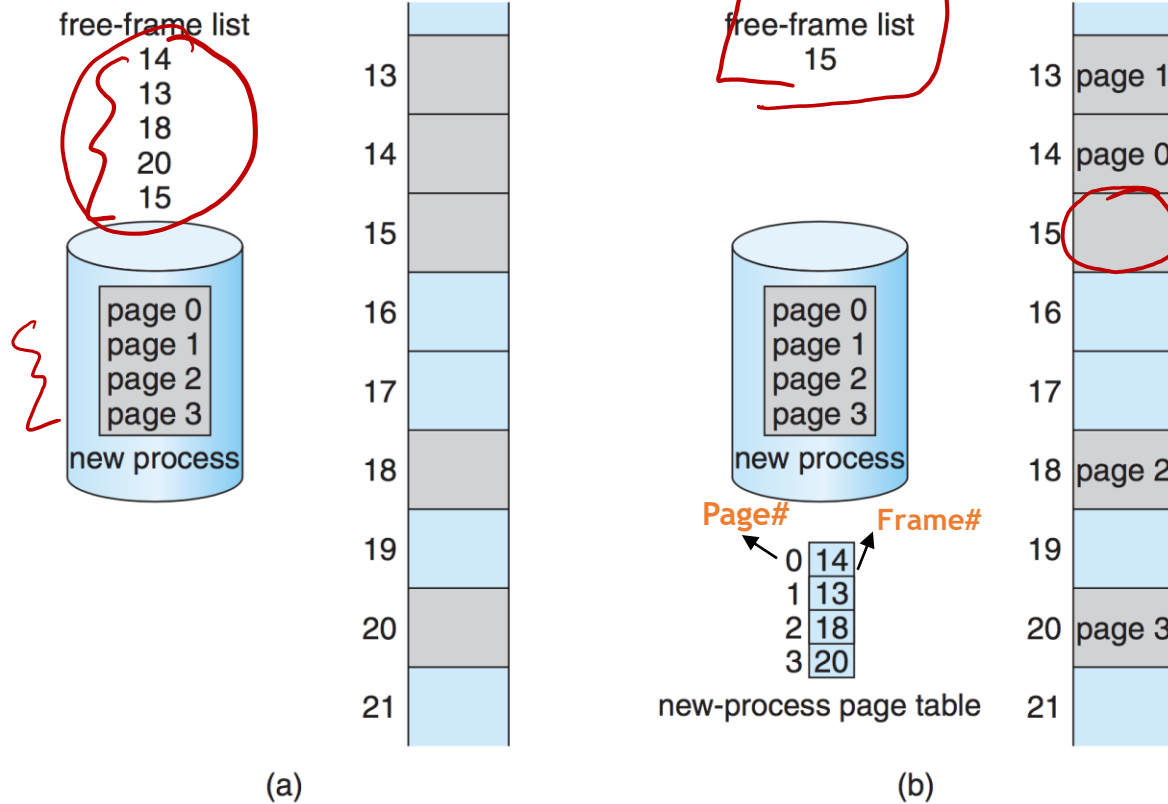
Internal fragmentation

Ques. What if the page size is very small?

Many page table entries, which consume memory

Free Page Frames

- The OS maintains a list of free frames




Before allocation

After allocation

Implementation of Page Table (The Slow Way)

Page table is kept in main memory

- **Page-table base register (PTBR)** points to page table
- **Page-table Length Register (PTLR)** indicates size of page table
- Both, PTBR and PTLR are also stored in the process's PCB
-  Most processes do not use all their address space ranges - waste memory space to create page table entries for each possible page
- Downside of keeping page table in memory?
 - Every data/instruction access requires two memory accesses
 - One for page table and one for data/instruction → **Slow**

Implementation of Page Table (The Slow Way)

Solution (A better way): Translation Look-aside Buffer (TLB)

- Fast-lookup hardware cache called associative memory, typically 32 to 1024 entries.
- Caches page table entries (of currently running process)
- Part of the MMU (hardware)
- Associative memory - parallel search

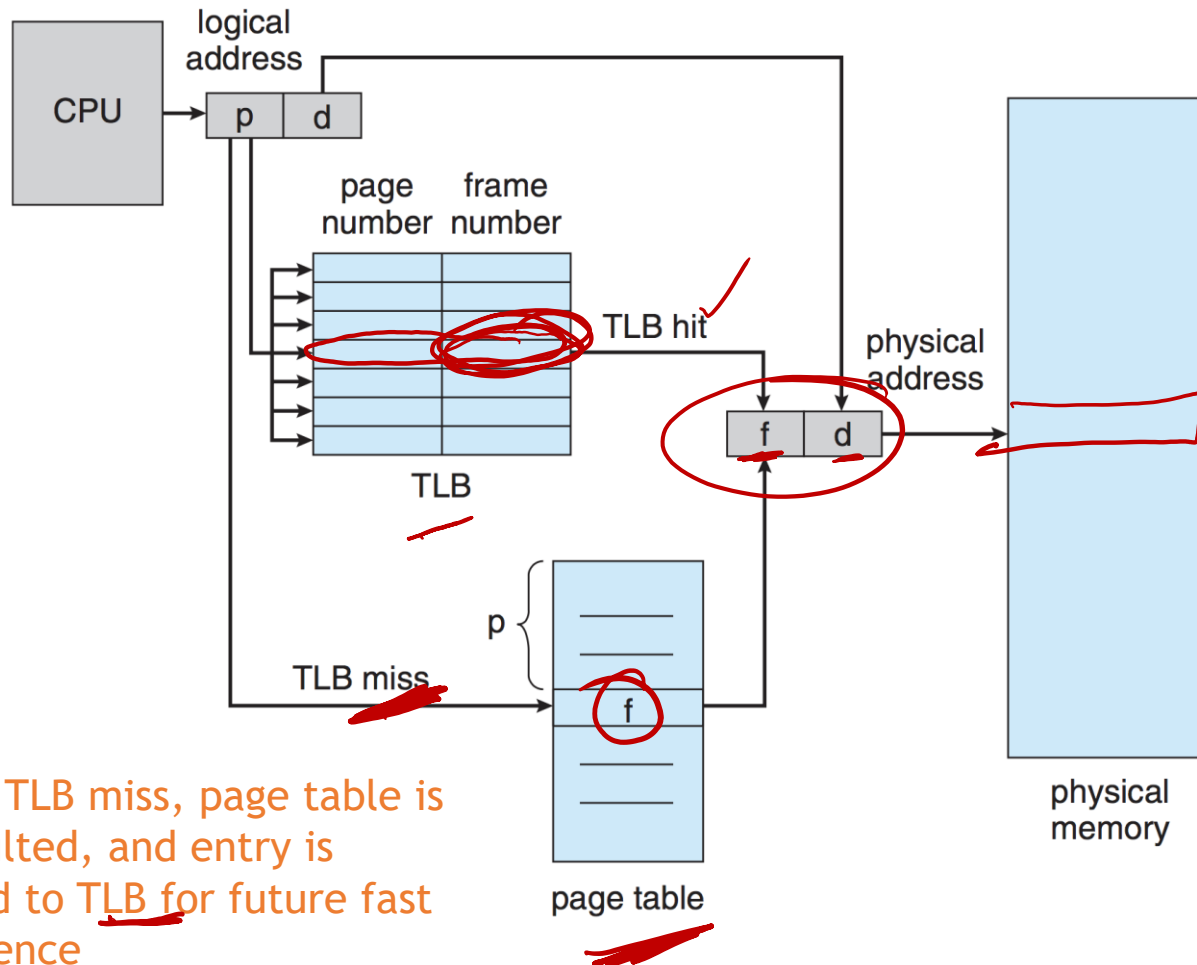
Address translation give (p, d)

- If p is in associative memory, then retrieve the corresponding frame #
- Otherwise retrieve the corresponding frame # from the page table in memory

Page #	Frame #

Paging With TLB

highspeed memory



Hit = 1 search + 1 memory reference

Miss = 1 search + 1 mem reference(of page table) + 1 mem reference

Paging With TLB

- Upon TLB miss, page table is consulted, and entry is added to TLB for future fast reference
- If TLB is full, replace an existing entry
 - OS may participate (mostly hardware)
 - Some cannot be removed (“wired down”)
 - E.g., pages containing kernel code
- Each entry stores Address Space Identifier (ASID)
 - Identifier for process ✓
 - Address translation checks ASID for protection
 - Allow TLB to hold page table entries for multiple processes simultaneously
 - No TLB flush (erase) during a context switch

Effective Access Time

- Associative lookup in TLB takes e time unit
- Memory access time is tm time unit
- Hit ratio α
 - % of times the requested page frame number is found in TLB

• Effective Access Time

$$= \alpha (e + tm) + (1 - \alpha) (e + tm + tm)$$

Handwritten annotations:
- α is labeled "hit"
- $(e + tm)$ is labeled "hit memory time"
- $(1 - \alpha)$ is labeled "miss"
- $(e + tm + tm)$ is labeled "miss memory time"

- As α approaches 1, effective access time approaches tm

The higher the TLB hit ratio, the lower the overhead of address translation.

Effective Access Time

$$\text{Effective Access Time} = \alpha(e + t_m) + (1 - \alpha)(e + t_m + t_m)$$

Consider $\alpha = 80\%$, e (TLB search) = 20 ns , t_m (memory access time) = 100 ns

With 80% TLB hit rate

$$\text{EAT} = 0.80 \times (100 + 20) + 0.20 \times (200 + 20) = 140 \text{ ns}$$

More realistic hit ratio $\alpha = \underline{99\%}$

$$\text{EAT} = 0.99 \times (100 + 20) + 0.01 \times (200 + 20) = 121 \text{ ns}$$

Modern systems have multiple levels of TLBs

- L1, L2, data, instruction
- Computing effective access time gets complicated

correct

Effective Access Time - Try!

$$\text{Effective Access Time} = \alpha(e + t_m) + (1 - \alpha)(e + t_m + t_m)$$

Consider $\alpha = .98\%$, e (TLB search) = .1 ns, t_m (memory access time) = 1 ns

EAT?

$$= .98 (.1+1) + (1-.98)(.1+1+1)$$

$$= .98 (1.1) + 0.02(2.1)$$

$$= 1.078 + 0.042$$

$$= 1.12 \text{ ns (With TLB)}$$

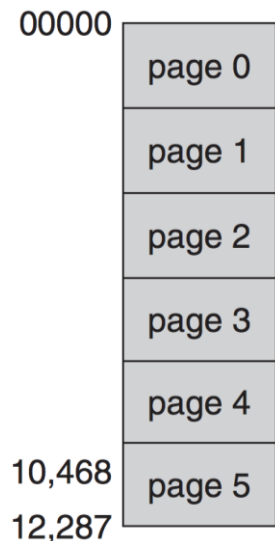
Memory Protection

- Associate protection bits with each frame
 - Specify whether a page is read-only, read-write, execute-only
 - One bit for each to allow combinations
- Another bit (valid-invalid) may be used
 - Valid indicates whether a page is in the process's address space, i.e., a legal page to access
 - Invalid indicates that the page is not in the process's address space
 - Illegal memory accesses trapped to OS - segfaults!

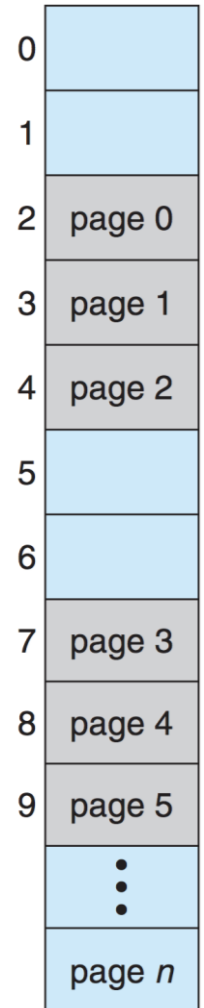
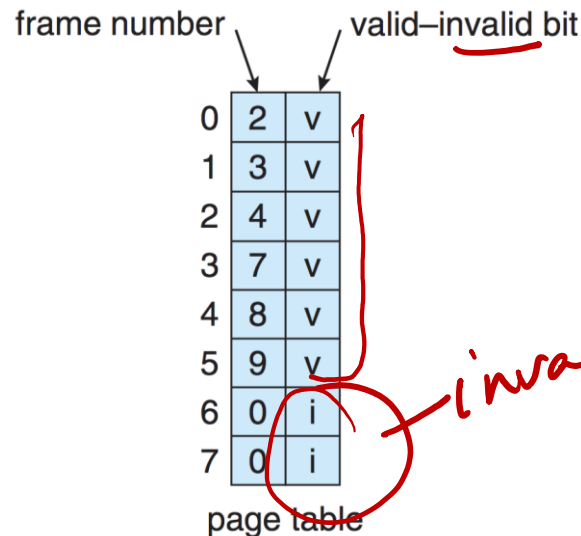
Valid (v) or Invalid (i) Bit in Page Table

Example: 14-bit address space

- Valid address range: 0 - 10468
- Pages 0 - 5 are valid
- Addresses for pages 6 - 7 are not accessible



**Logical
memory**



**Physical
memory**

Shared Pages

shared code

Ques. Suppose that you have code (e.g., emacs) that are being used by several processes at the same time. Does every process need to have a separate copy of that code in memory?

NO. Put the code in “Shared Pages”

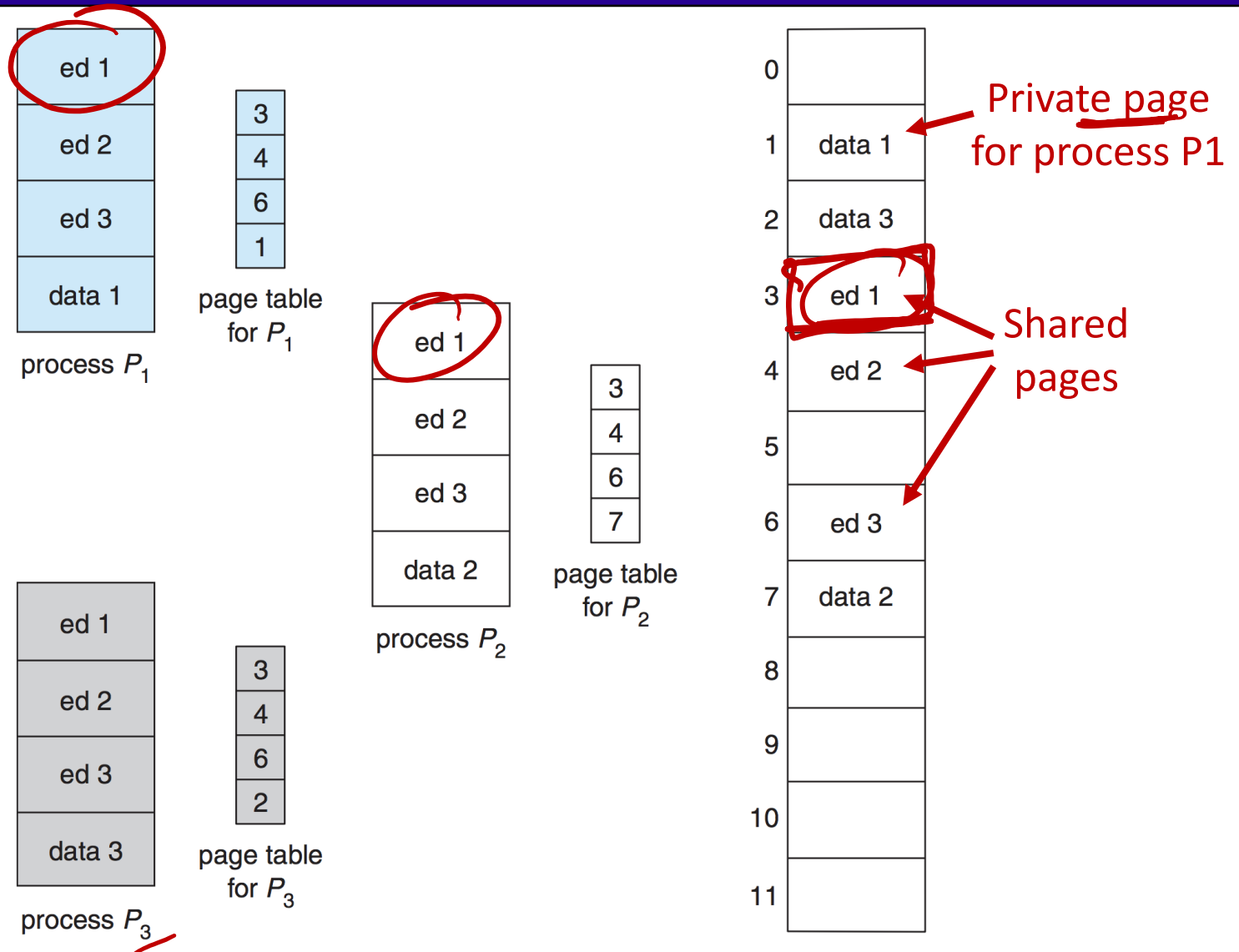
- One copy of read-only (reentrant) code shared among processes (e.g., editors, compilers, window systems)
- Also useful for inter-process communication if sharing of read-write pages is allowed
- Shared code must appear in the same location in the logical address space of all processes

Ques. What if the shared code creates per-process data?

Private data

- Each process keeps “private pages” for data
- Private pages can appear anywhere in address space

Example: Shared/Private Pages



Page Table Structure

Assume 32-bit address space, page size 4 KB (2^{12})

- How many pages can a process have?
 - $2^{32} / 2^{12} = 2^{20}$ pages
- If each entry in page table takes 4 bytes, what would be the size of page table?
 - $2^{20} \times 4 = 4 \text{ MB}$
- Anything wrong with this number? ✓
 - Huge size (and must be contiguous in memory)
- Solution
 - Option 1: Use larger pages (e.g., 4MB instead of 4KB)
 - Option 2: divide page table into smaller pieces in memory
 - Hierarchical Paging, Hashed Page Tables, Inverted Page Tables

Page Table Structure

The “large page table problem”

Page table structures:

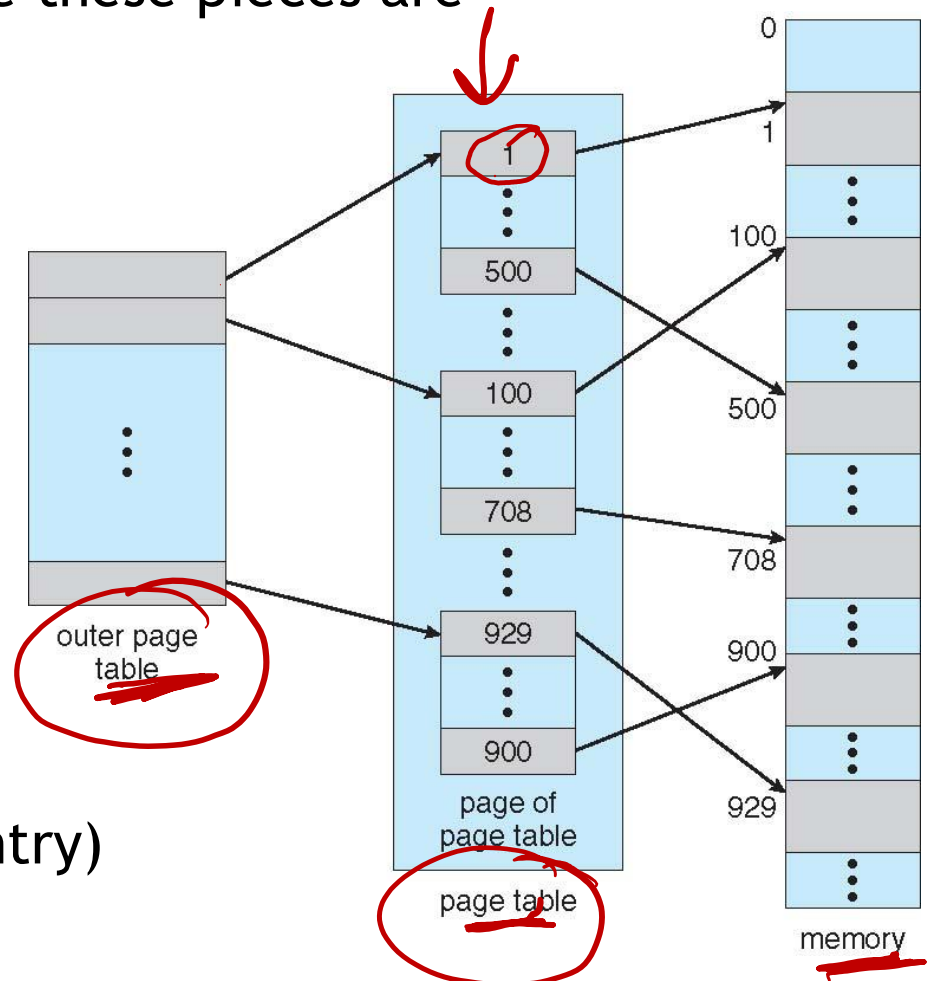
- Hierarchical Paging: “page the page table”
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

“Page” the page table itself: break it into pieces and then create another table to indicate where these pieces are

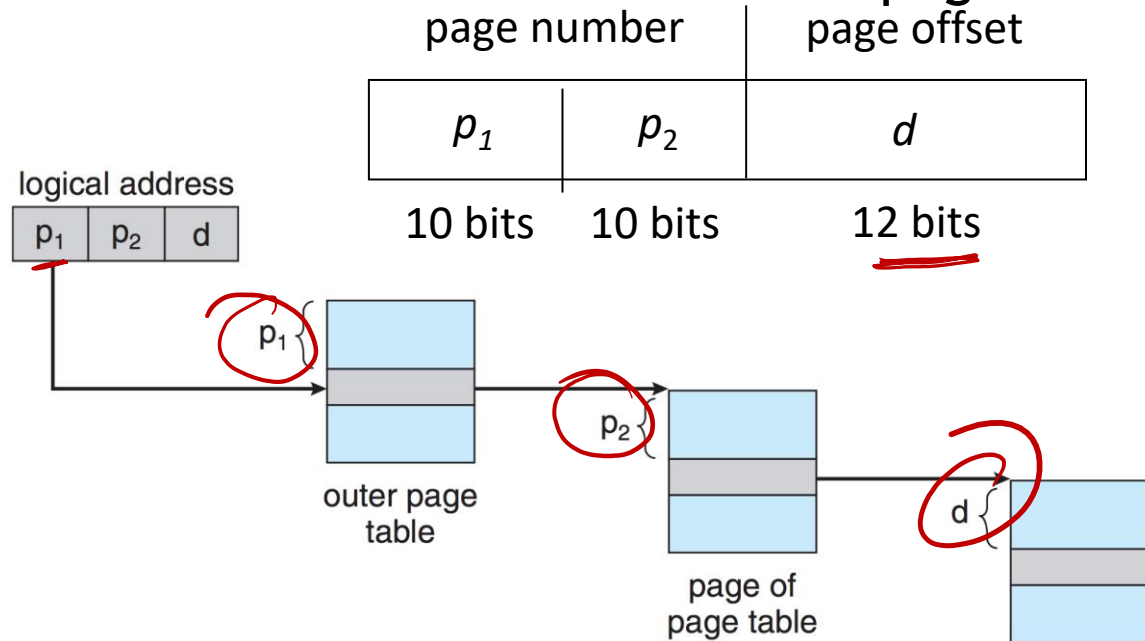
Example: two-level page table

- Benefits
 - Page table pieces can be anywhere
 - Also, pieces can be created on demand (as process needs)
- Downside: memory overhead (2 memory accesses for PT entry)



Two-Level Paging Example

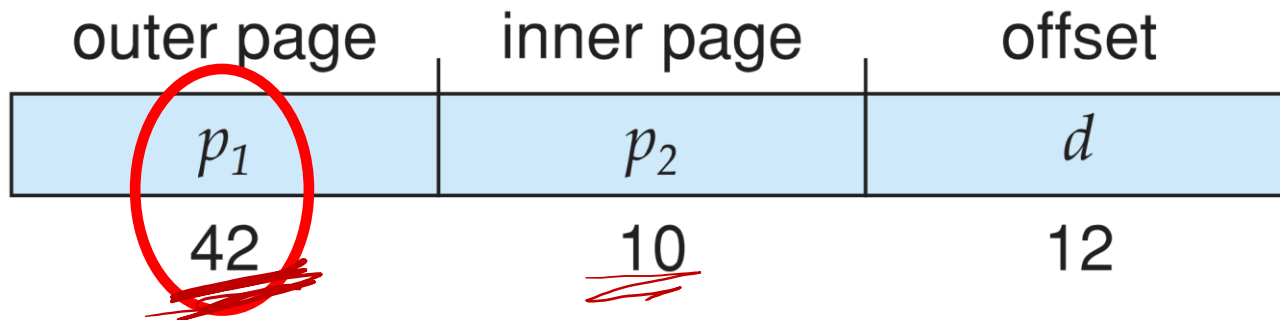
- Address on 32-bit machine with 4KB page size looks like:



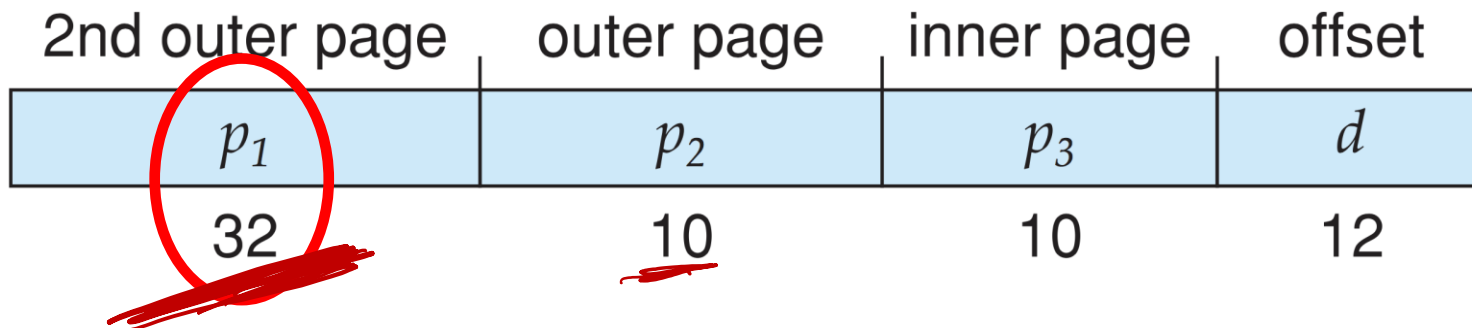
- OS creates the outer page table and one page of the inner page table
- More pages of the inner table are created on-demand

Three-level Paging Scheme

- For 64-bit address space, addresses look like with 2-level page tables:

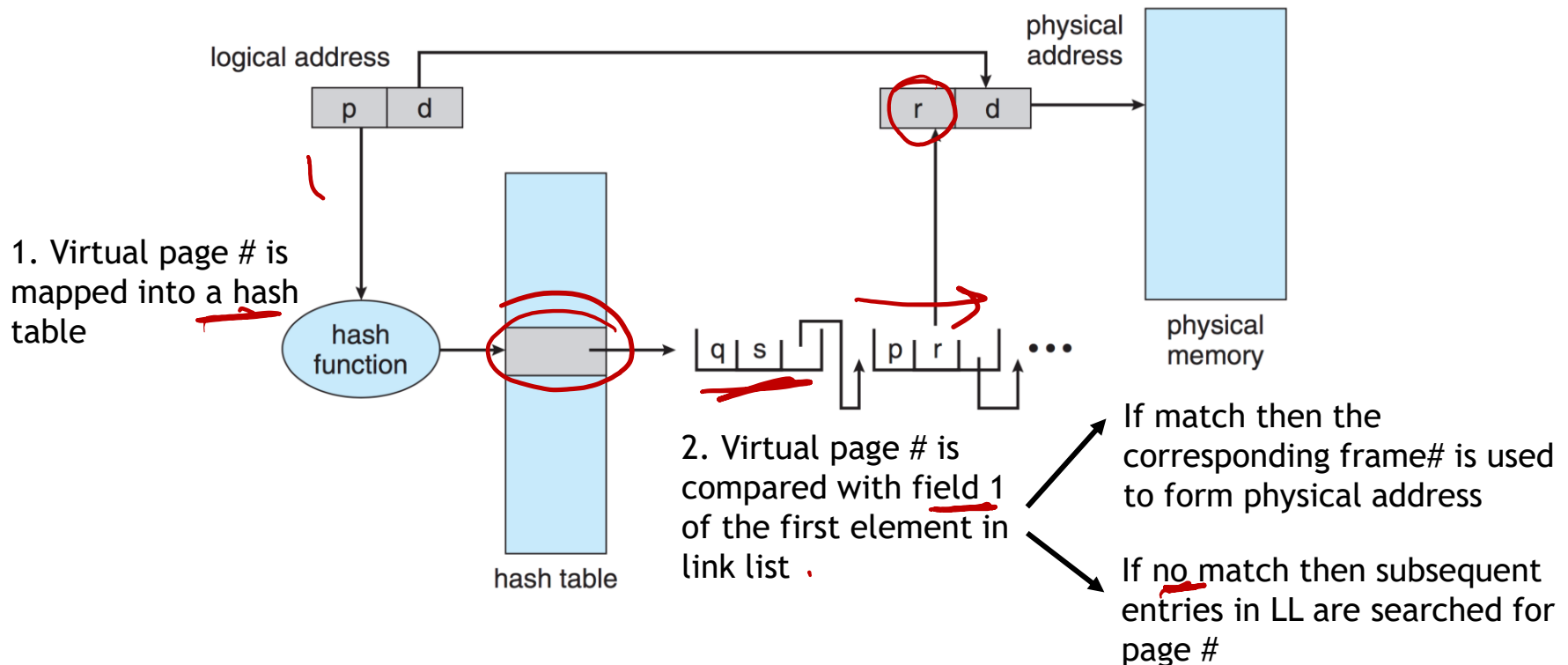


- Using 2 levels, assuming 4-byte entries \rightarrow total 16TB (2^{42})
- To reduce size, we may use 3 levels:



Hashed Page Tables

- Fixed-size hash table
- Page number is hashed into a page table
 - Collisions result in a chain of elements
 - Each element has (1) virtual page number, (2) mapped page frame number and (3) pointer to next element in linked list



Inverted Page Tables

- One entry for each frame of physical memory
 - Limits the size to number of physical frames
- Entry has: page # stored in that frame, and info about process (PID) that owns that page

Inverted Page Table

Ques 1. If we have 20 processes and each process's pages are in MM, how many process tables, we will need in the main memory?

20

Page table of P1

0	2
1	1
2	3

Ques 2. How many frames, we will need to store these PT

20

Page #

0	2	1
1	2	2
2	3	1

Process id

Frame#

linear search time

Global page table - IPT

Frame #

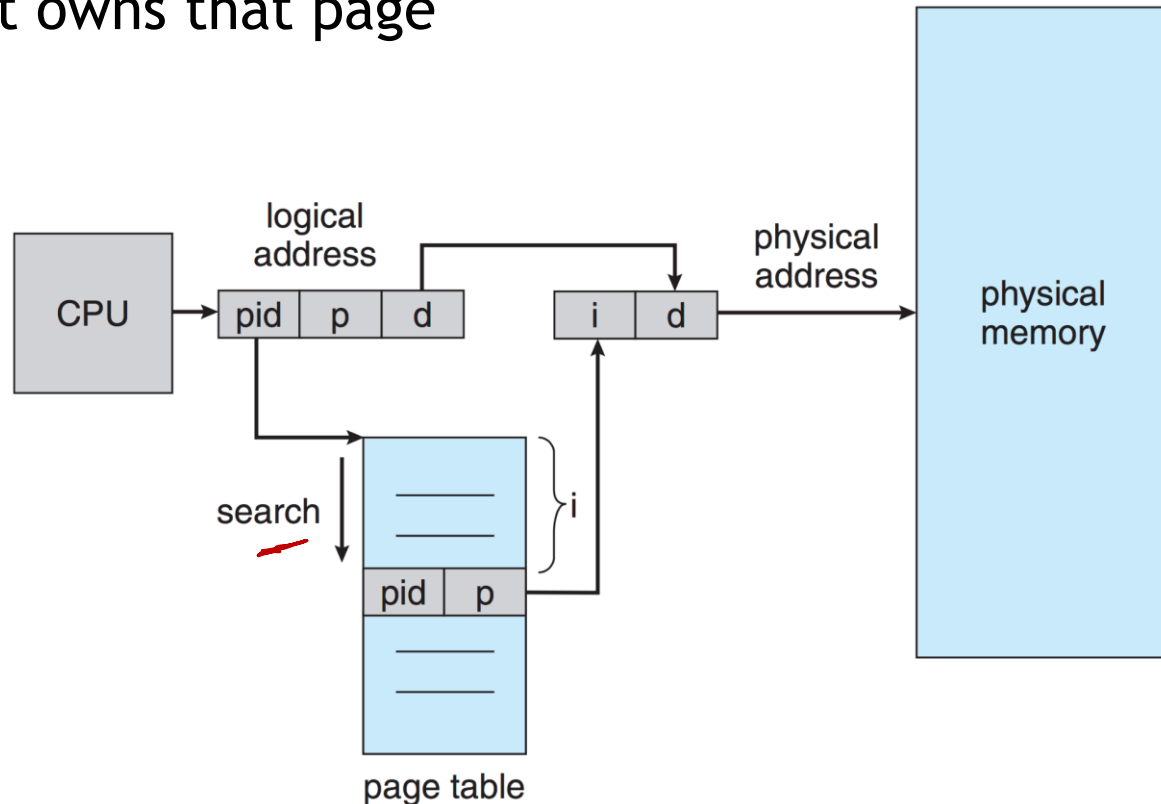
0	2
1	3
2	1

Page #

Page table of P2

Inverted Page Tables

- One entry for each frame of physical memory
 - Limits the size to number of physical frames
- Entry has: page # stored in that frame, and info about process (PID) that owns that page



Inverted Page Tables

Pros:

- Decreases memory needed to store page table

Cons:

- Increases time needed to search the table when a page reference occurs
 - Can be alleviated by using a hash table to reduce search time
 - Adds a memory reference, but TLB can help
- Difficult to implement shared pages
 - Because we use only one entry for each frame in the page table, i.e., one virtual address for each frame

Summary

- Segmentation: variable size, user's view of memory
- Contiguous memory allocation
 - First-, Best-, and worst-fit
 - Mechanisms: splitting, coalescing, tracking of free and allocated memory
 - Fragmentation (external, internal)
- Paging: non-contiguous memory allocation
 - Logical address space is divided into pages, which are mapped using page table to memory frames
 - Page table (one table for each process)
 - Access time: can be reduced by caching page table entries in TLB
 - Different types: multiple levels, hashed page tables, inverted page tables

↓ virtual memory