Misc

- **Exercise 2 updated with Part III.**
  - Due on next Tuesday 12:30pm.
- **Project 2 (Suggestion)**
  - Write a small test for each call.
  - Start from file system calls with simple Linux translation.
  - Or start from Exec()
- **Midterm (May 7)**
  - Close book. Bring 3 pages of double-sided notes.
  - Next Monday: Project 2 or review?

More on Address Translation

CS170 Fall 2015. T. Yang
Based on Slides from John Kubiatowicz
http://cs162.eecs.Berkeley.edu

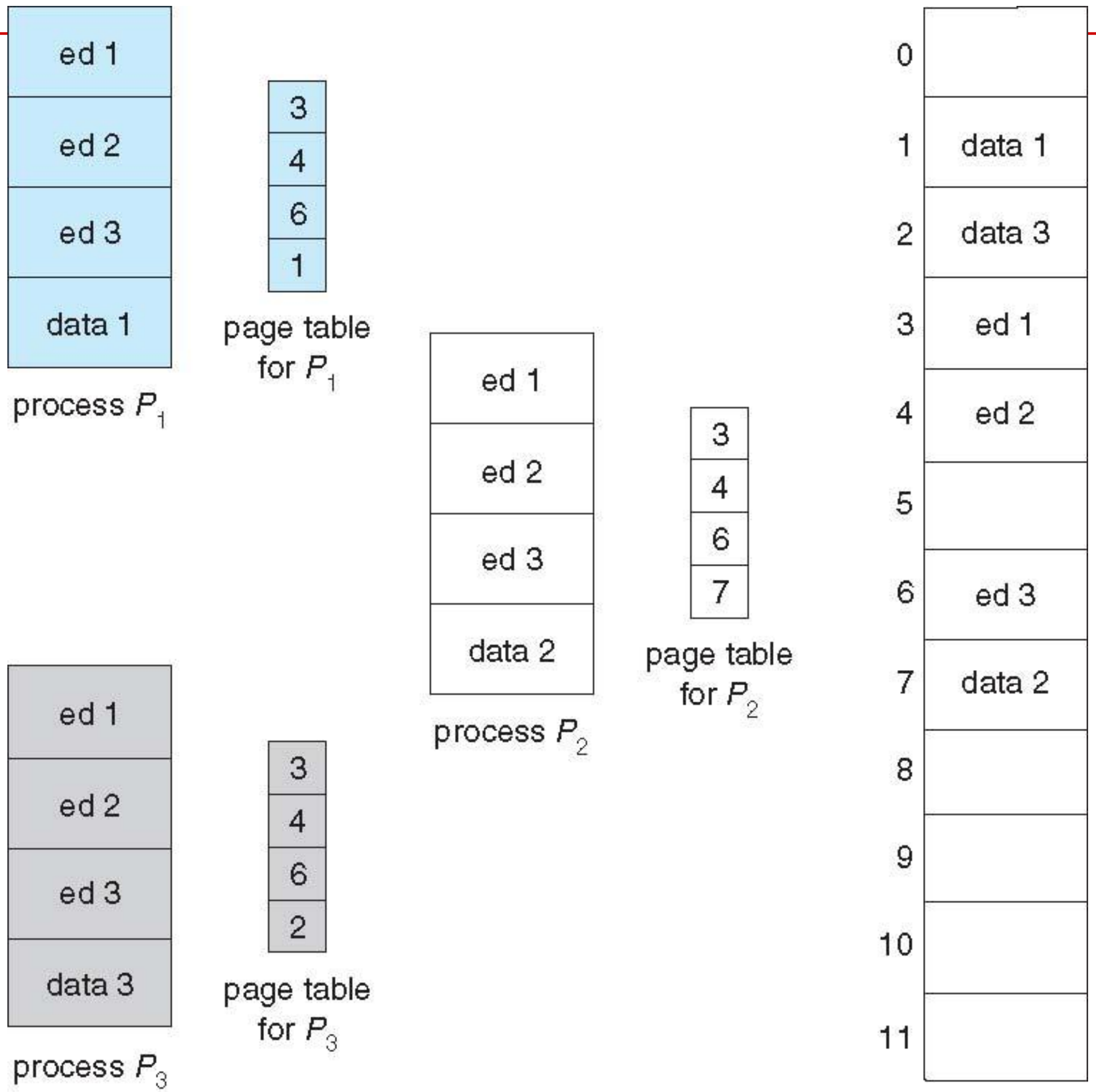
Implementation Options for Page Table

- **Page sharing among process**
- **What can page table entries be utilized?**
- **Page table implementation**
  - One-level page table
  - Muti-level paging
  - Inverted page tables

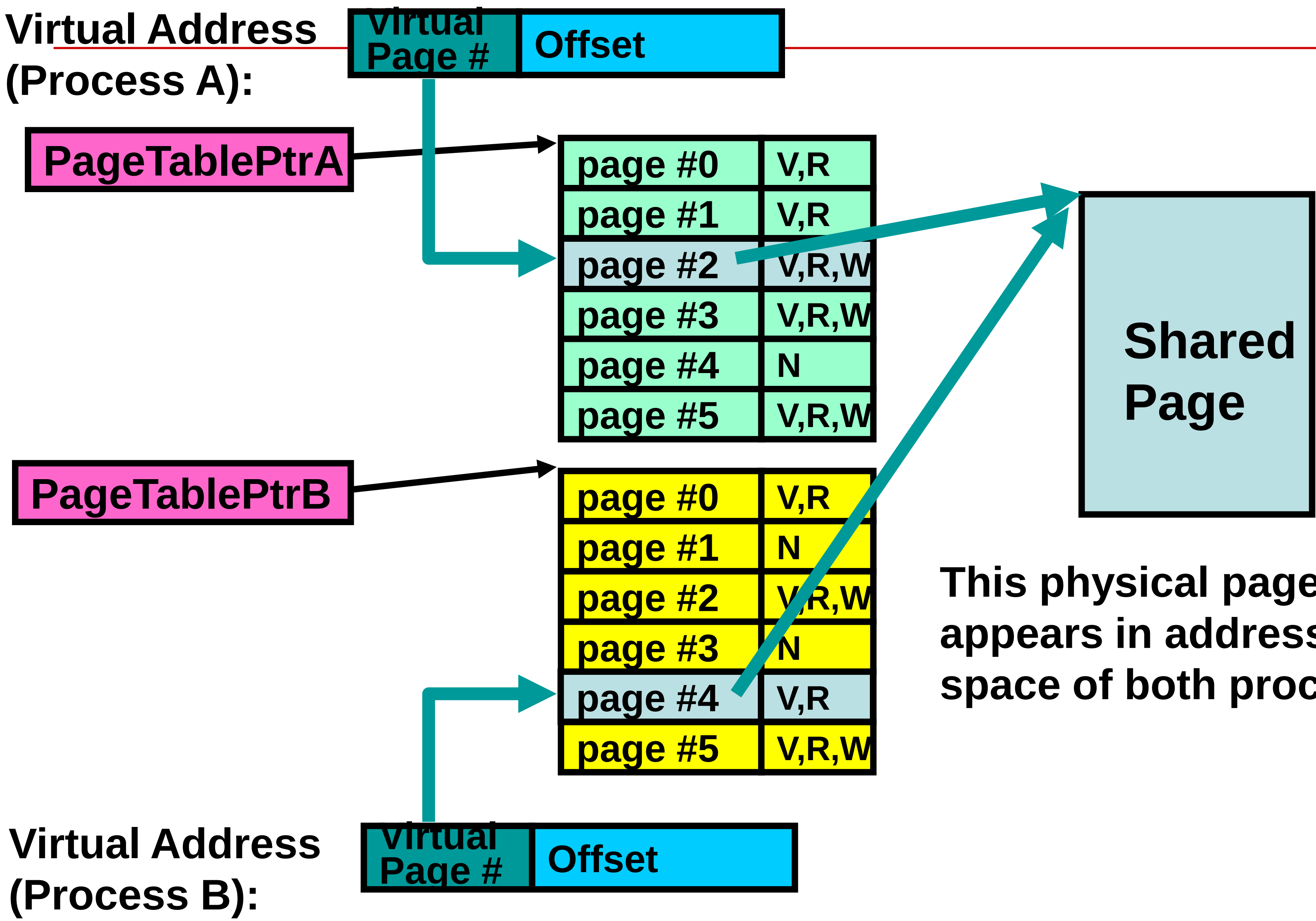
Shared Pages through Paging

- **Shared code**
  - One copy of read-only code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **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



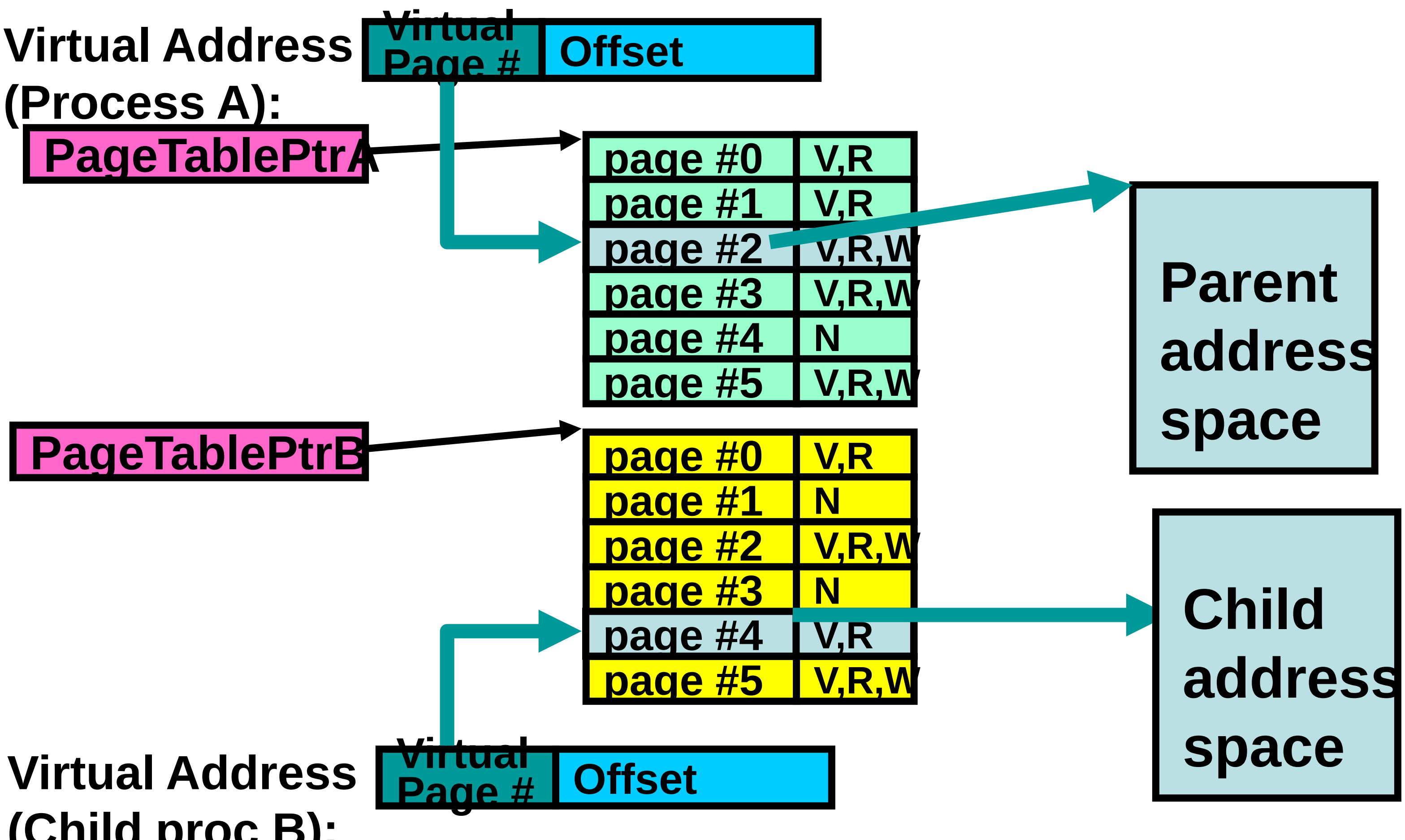
Example



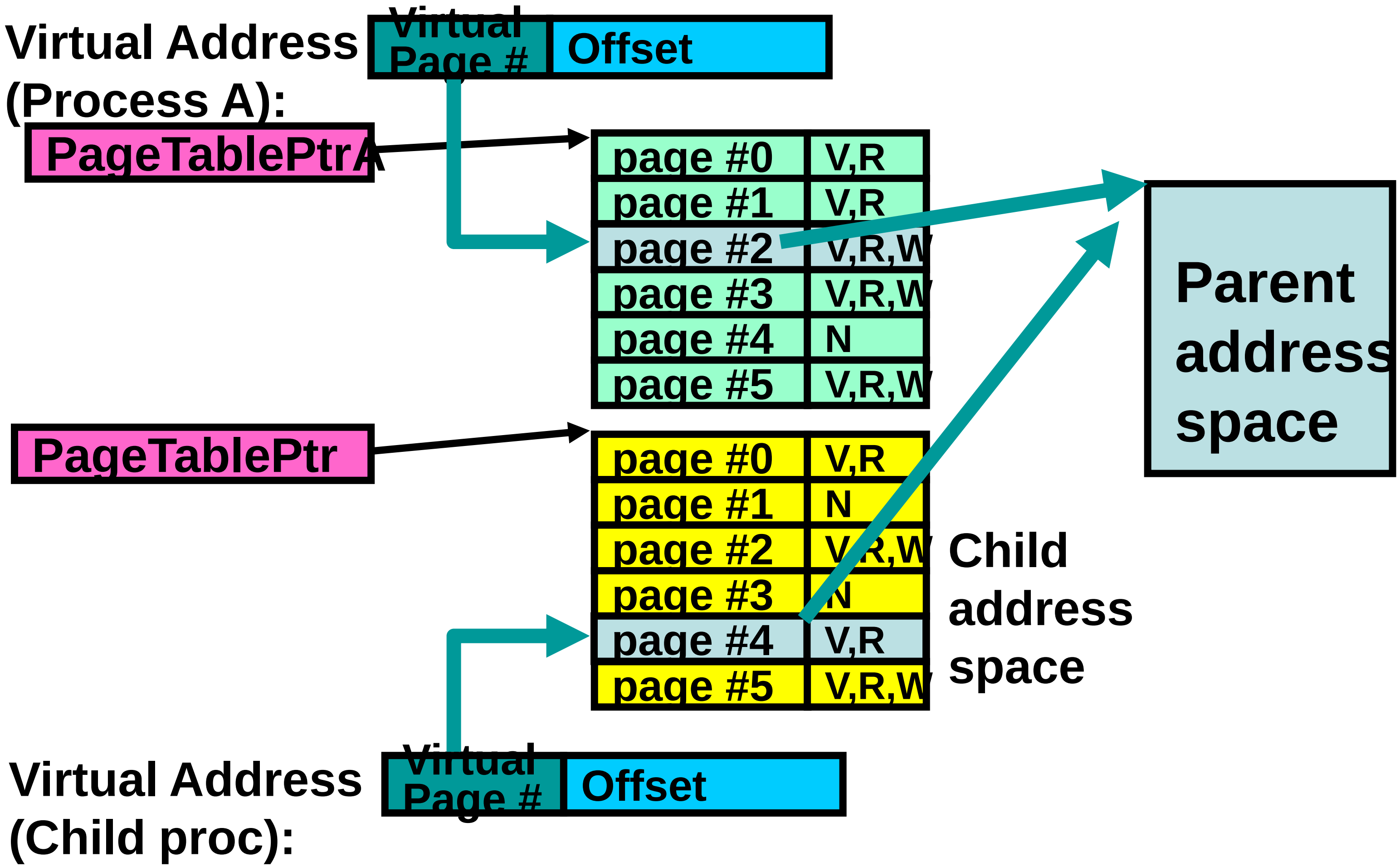
This physical page appears in address space of both processes

Optimization of Unix System Call Fork()

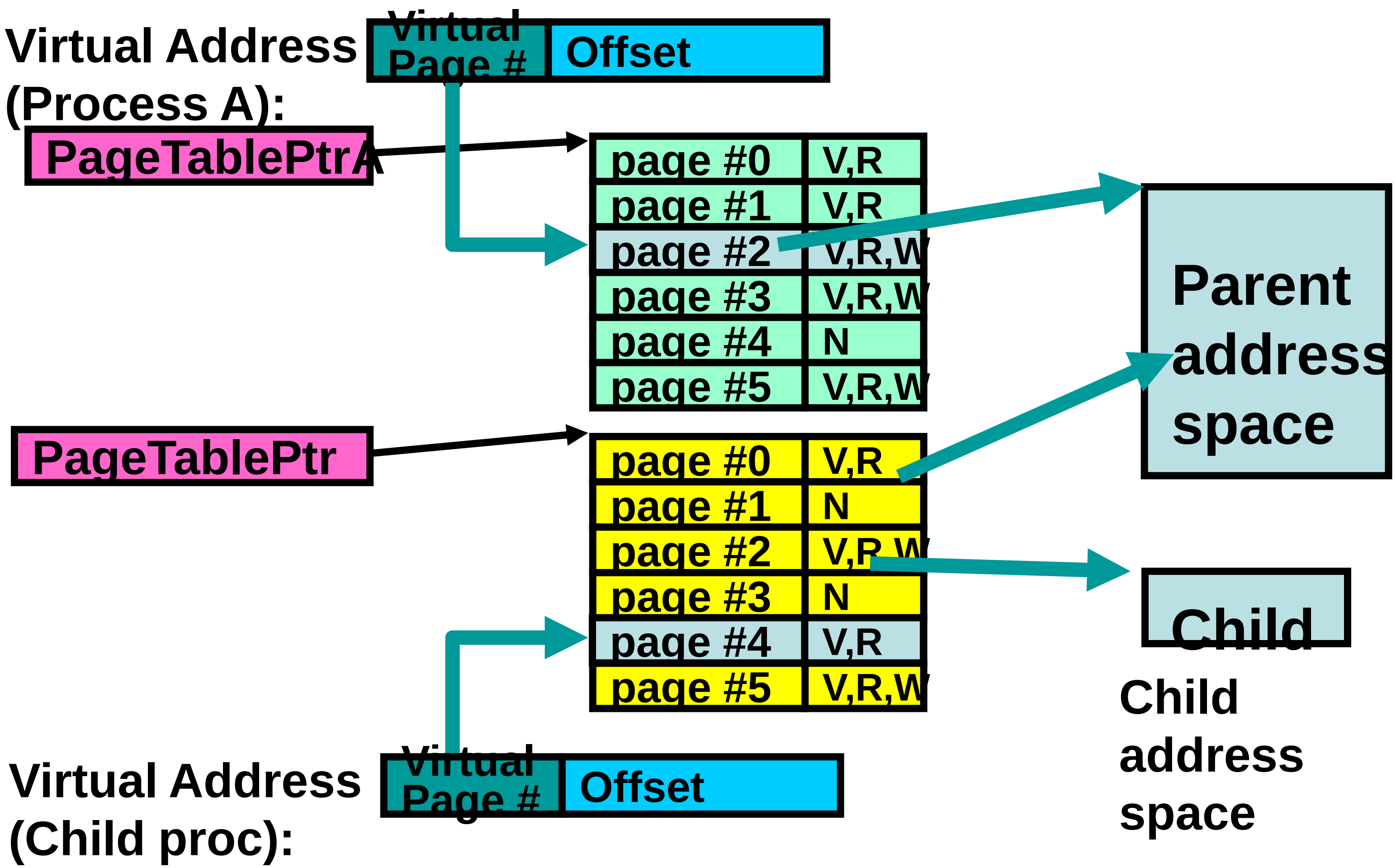
- A child process copies address space of parent.
  - Most of time it is wasted as the child performs exec().
  - Can we avoid doing copying on a fork()?



Unix fork() optimization

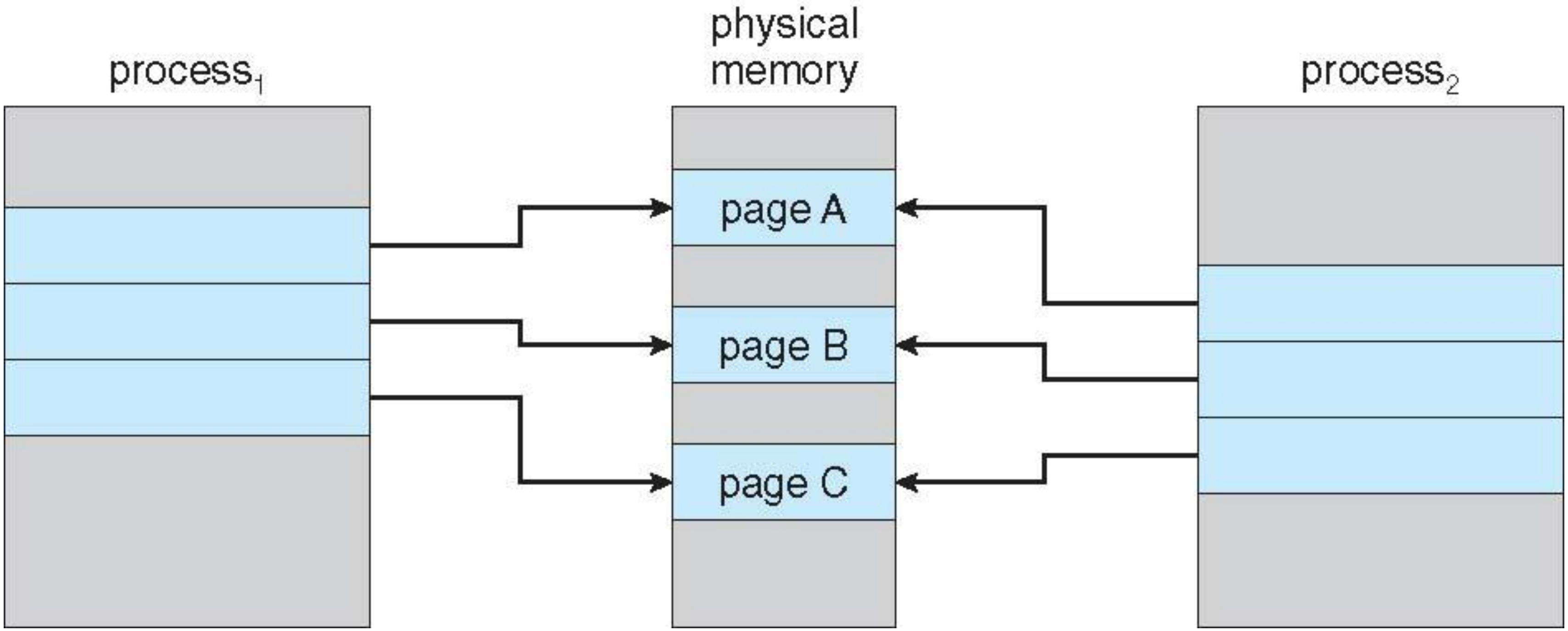


Unix fork() optimization

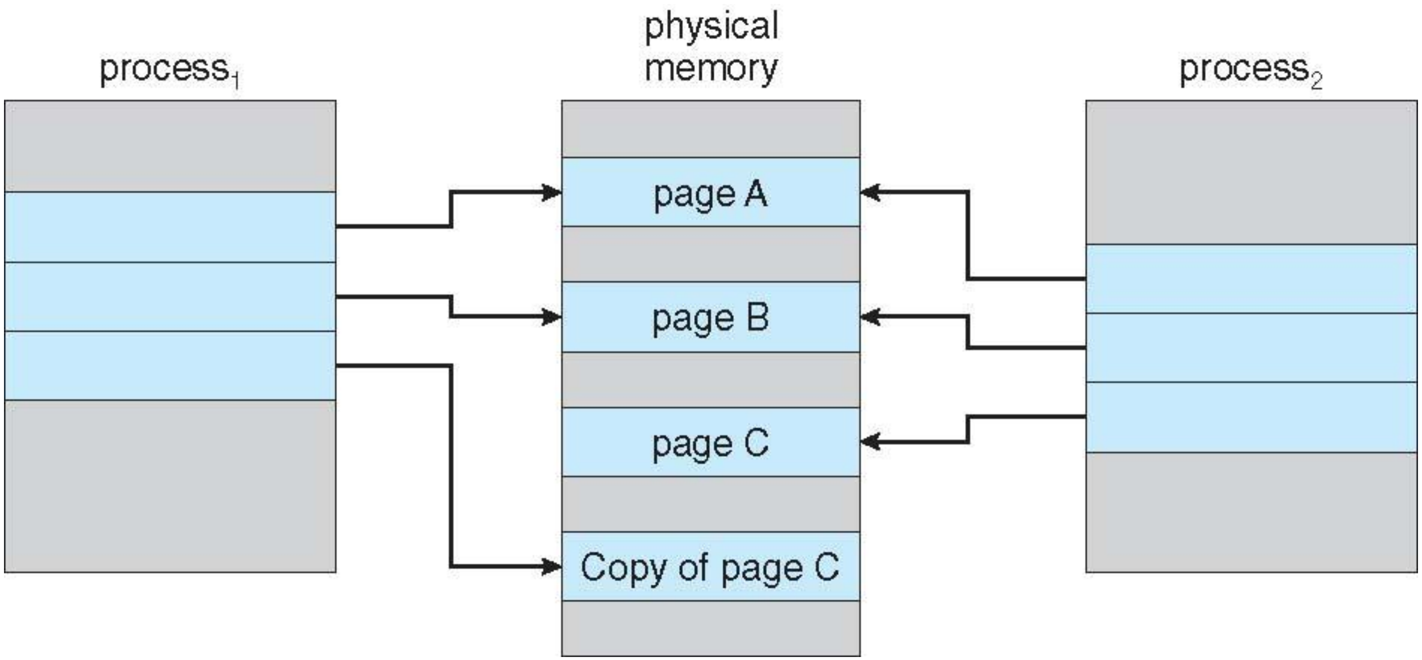


Copy-on-Write: Lazy copy during process creation

- **COW** allows both parent and child processes to initially *share* the same pages in memory**.**
- A shared page is duplicated only when modified
- COW allows more efficient process creation as only modified pages are copied



Copy on Write: After Process 1 Modifies Page C



How to memorize a page is shared?

When to detect the need for duplication?

Need a page table entry bit

Page table entry	Physical page number			

More examples of utilizing page table entries

Page table entry

Physical page number			

- **How do we use the PTE?**
  - Invalid PTE can imply different things:
    - Region of address space is actually invalid or
    - Page/directory is just somewhere else than memory
  - Validity checked first
    - OS can use other bits for location info
- **Usage Example: Copy on Write**
  - Indicate a page is shared with a parent
- **Usage Example: Demand Paging**
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid

Example: Intel x86 architecture PTE

- Address format (10, 10, 12-bit offset)
- Intermediate page tables called “Directories”

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as “valid” bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty (PTE only): page has been modified recently

L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

More examples of utilizing page table entries

- **Usage Example: Zero Fill On Demand**
  - Security and performance advantages
    - New pages carry no information
  - Give new pages to a process initially with PTEs marked as invalid.
    - During access time, page fault → physical frames are allocated and filled with zeros
  - Often, OS creates zeroed pages in background

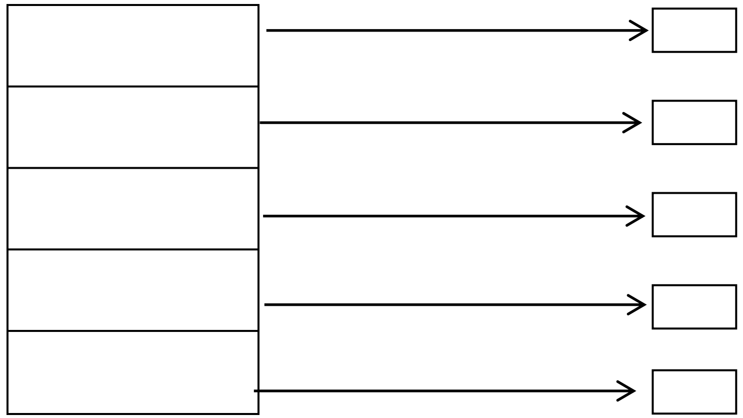
- **Can a process modify its own translation tables?**
  - NO!
  - If it could, could get access to all of physical memory
  - Has to be restricted

Implementation Options for Page Table

- **Page sharing among process**
- **What can page table entries be utilized?**
- **Page table implementation**
  - One-level page table
  - Muti-level paging
  - Inverted page tables

One-Level Page Table

- **What is the maximum size of logical space?**



Mapping of pages

Constraint:
Each page table needs to fit into a physical memory page!

Why? A page table needs consecutive space.
Memory allocated to a process
is a sparse set of nonconsecutive pages

Maximum size = # entry * page size

One-level page table cannot handle large space

- **Example:**
  - 32 -bit address space with 4KB per page.
  - Page table would contain $2^{32}/ 2^{12}$= 1 million entries.
    - 4 bytes per entry
- **Need a 4MB page table with contiguous space.**
    - Is there 4MB contiguous space for each process?
- **Maximum size with 4KB per page**

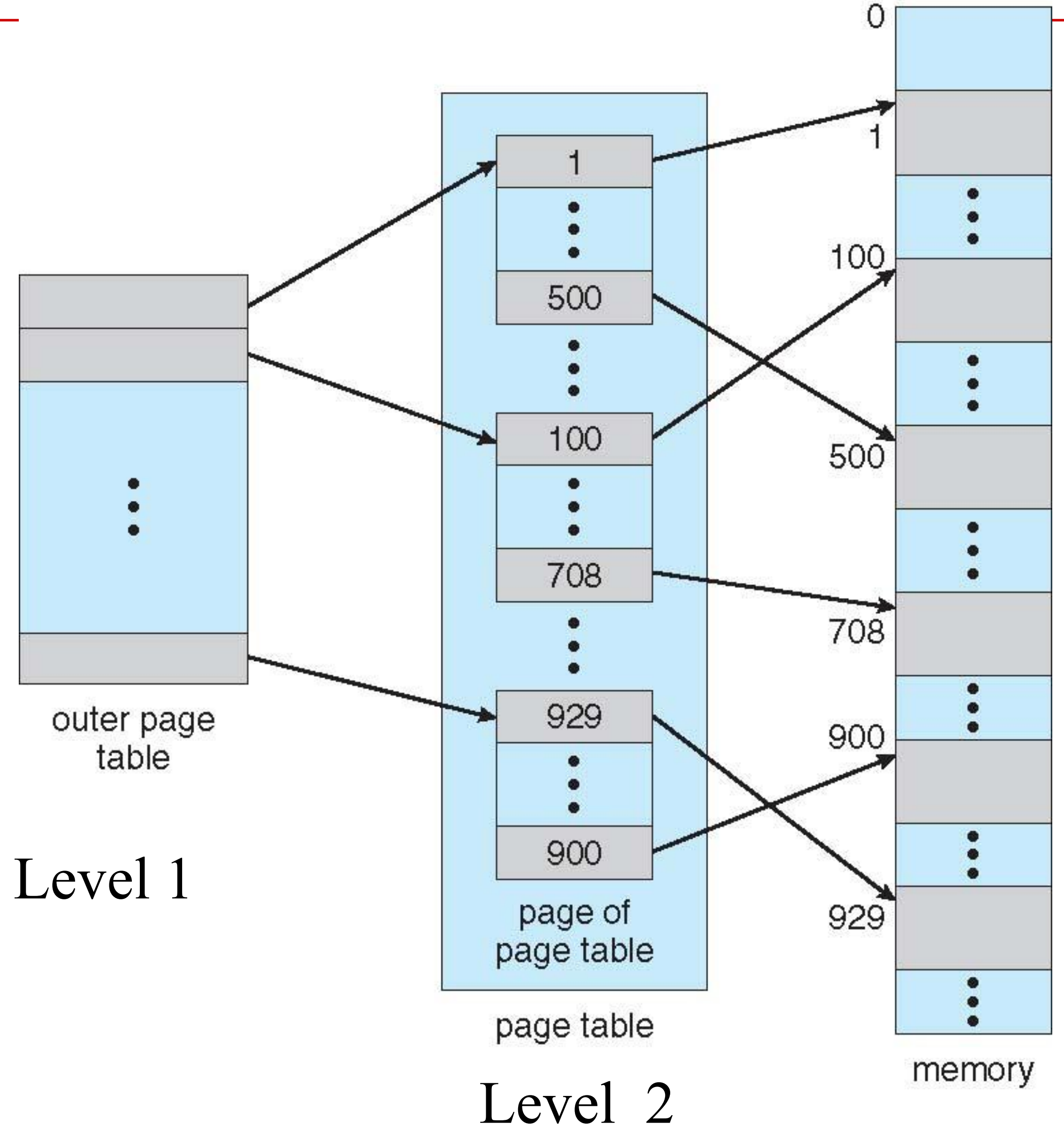
  #entry= 4KB/4B = 1K.

  Maximum logical space=1K*4KB= 4MB.

One-level Page Table: Advantage/Disadvantage

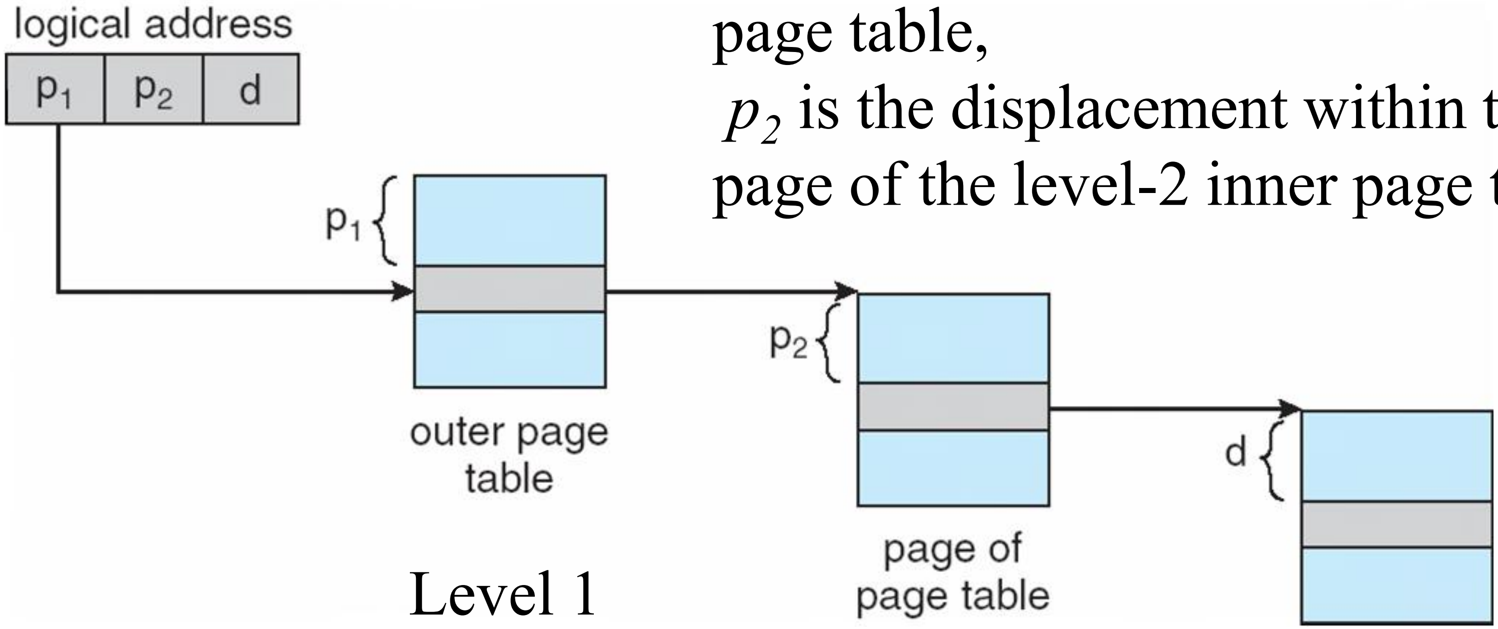
- **Pros**
  - Simple memory allocation
  - Easy to Share
- **Con: What if address space is sparse?**
  - E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
  - Cannot handle a large virtual address space
- **Con: What if table really big?**
  - Not all pages used all the time $\Rightarrow$ would be nice to have working set of page table in memory
- **How about combining paging and segmentation?**
  - Segments with pages inside them?
  - Need some sort of multi-level translation

Two-Level Page-Table Scheme



Address-Translation Scheme

p_1 is an index into the outer level-1 page table,
p_2 is the displacement within the page of the level-2 inner page table

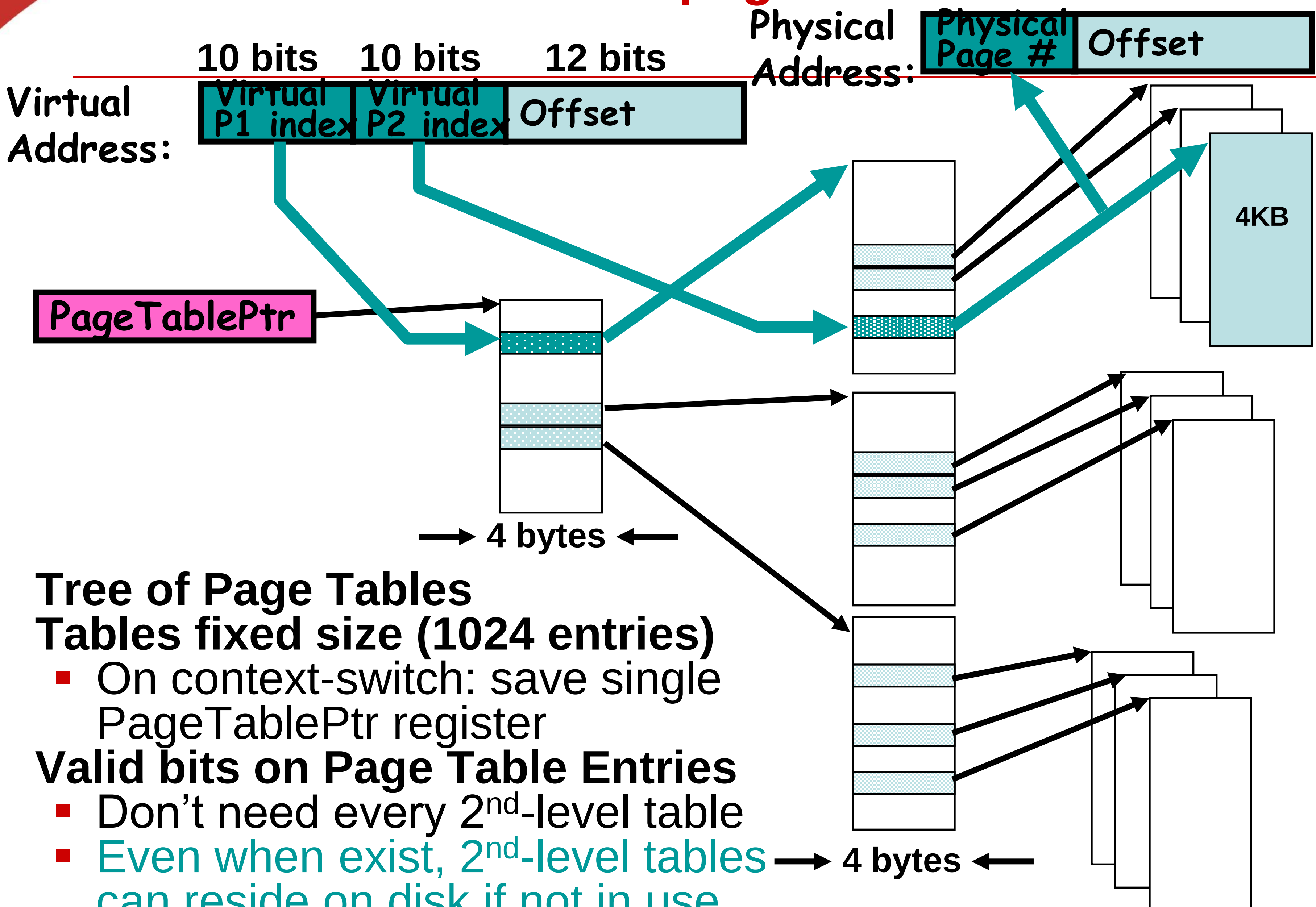


Level 1

Level 2

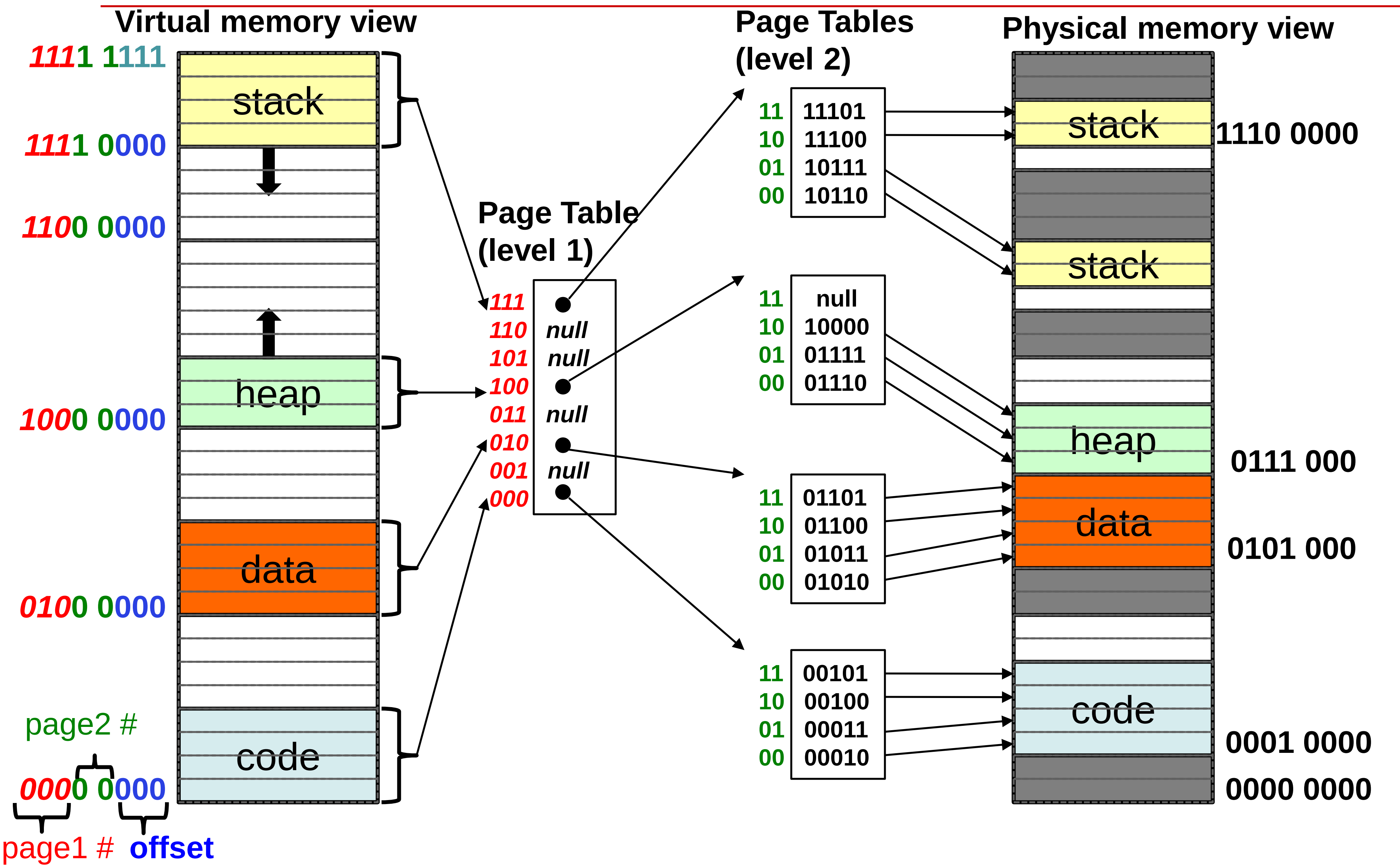
Level-2 page table gives the final physical page ID

Flow of the two-level page table

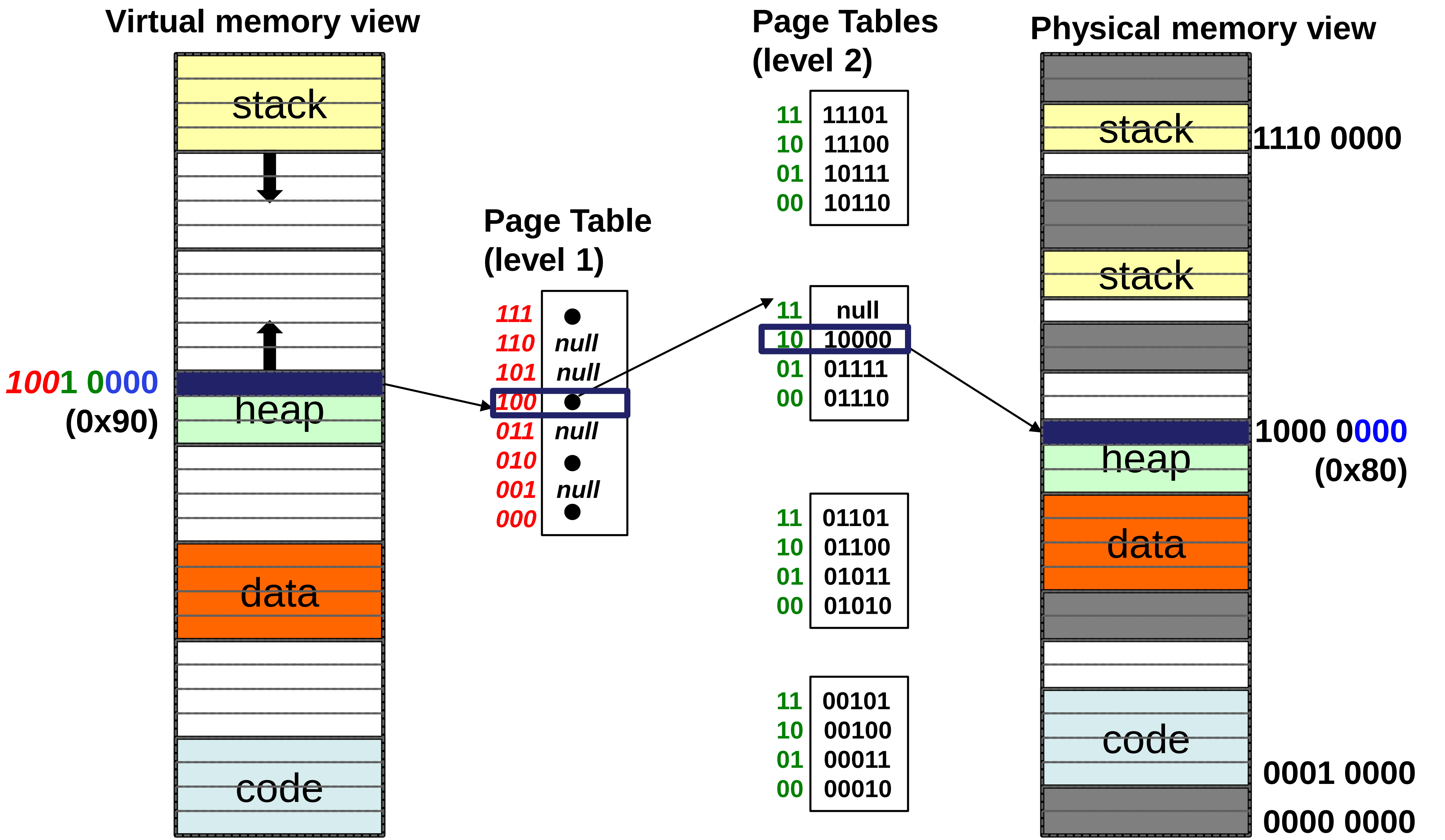


- **Tree of Page Tables**
- **Tables fixed size (1024 entries)**
  - On context-switch: save single PageTablePtr register
- **Valid bits on Page Table Entries**
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Summary: Two-Level Paging

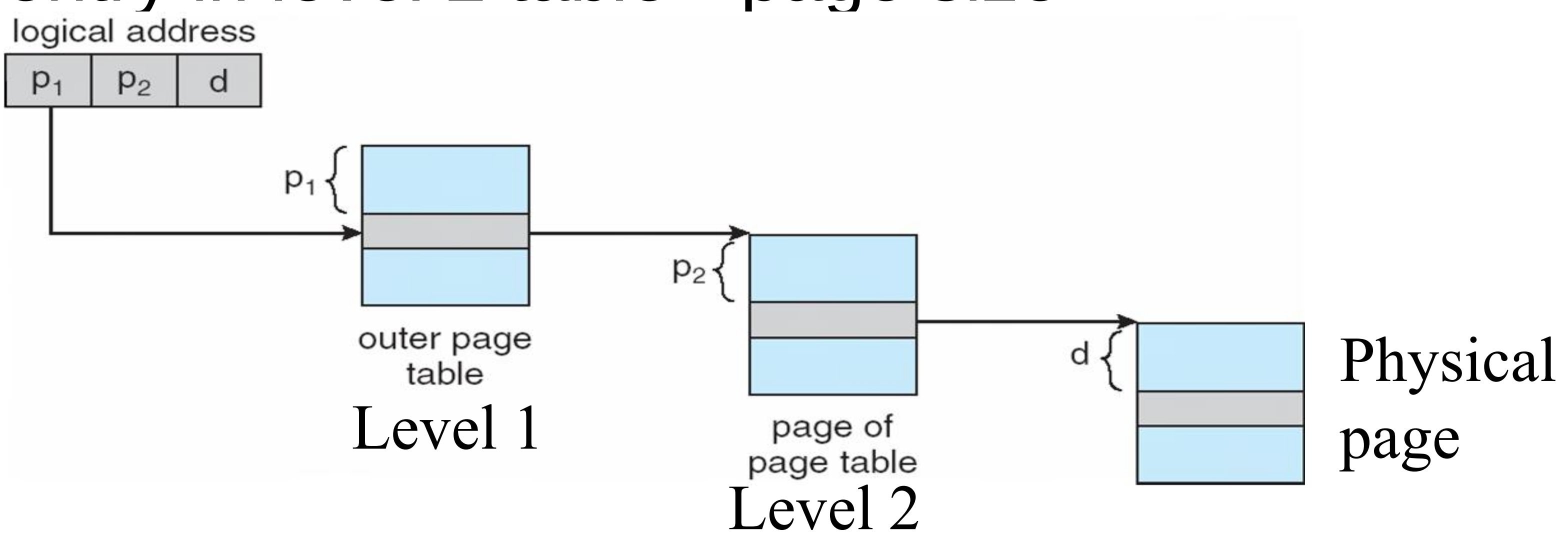


Summary: Two-Level Paging



Constraint of paging

- Bits of d = log (page size)
- Bits of p1 >= log (# entries in level-1 table)
- Bits of p2 >= log (# entries in level-2 table)
- Physical page number is limited by entry size of level-2 table.
- logical space size = # entry in level-1 table * # entry in level 2 table * page size



Analysis of a 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
- **Each entry uses 4 bytes**
- **How to build a two-level paging scheme?**
  - How many entries can a single-page table hold?
  - What are p_1, p_2?

page number		page offset
p_i	p_2	d
?	?	12

Analysis of a Two-Level Paging Example

- **A 1-page table with 4KB contains 1K entries and each uses 4B.**
- **1K entries require 10 bits for P_1 and P_2 offset**
- **The page number is further divided into:**
  - a 10-bit level-1 index
  - a 10-bit level-2 index

page number		page offset
p_i	p_2	d
10	10	12

What if we use 2 bytes for each table entry?

- Increased logical space size?
- Increased physical space size?

Example of maximum logical space size

- Maximum logical space size

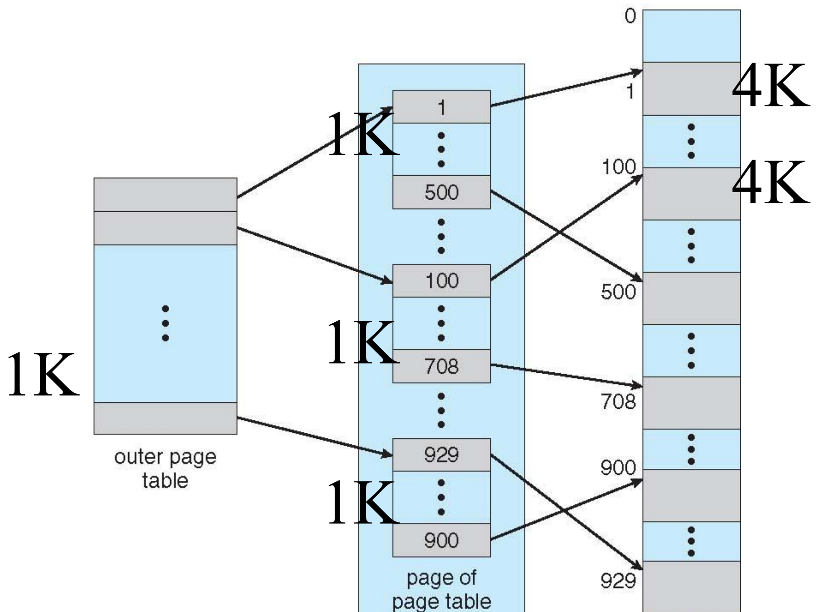
entry in level-1 page table * # entry in level-2 page table * page size

page number		page offset
p_i	p_2	d
10	10	12

= 1K * 1K * 4KB

= 2^{32} bytes

= 4GB



An example of three-level paging in a 64-bit address space

outer page	inner page	offset
p_1	p_2	d
42	10	12

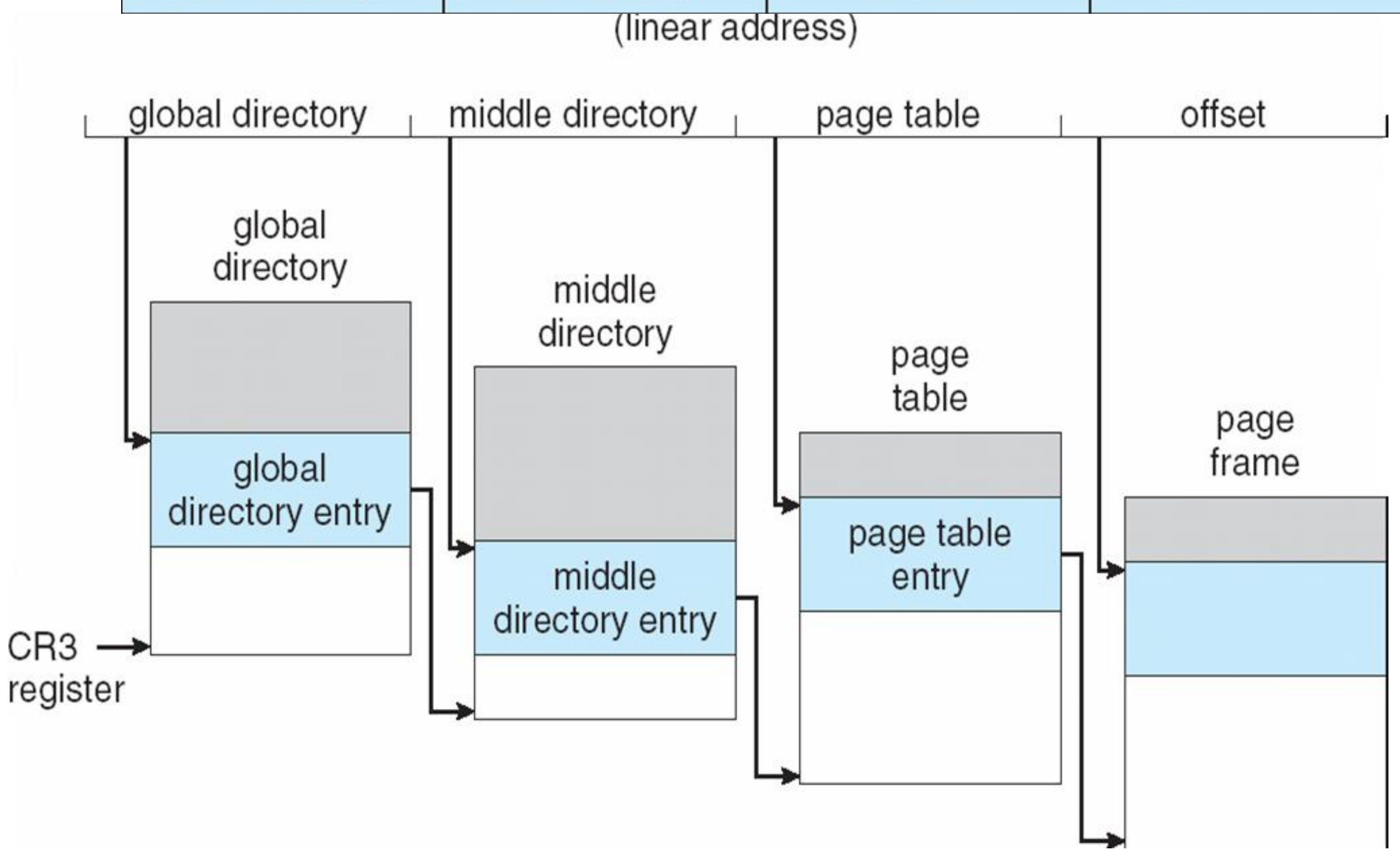
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

What is the maximum logical and physical space size?

Three-level Paging in Linux

Address is divided into 4 parts

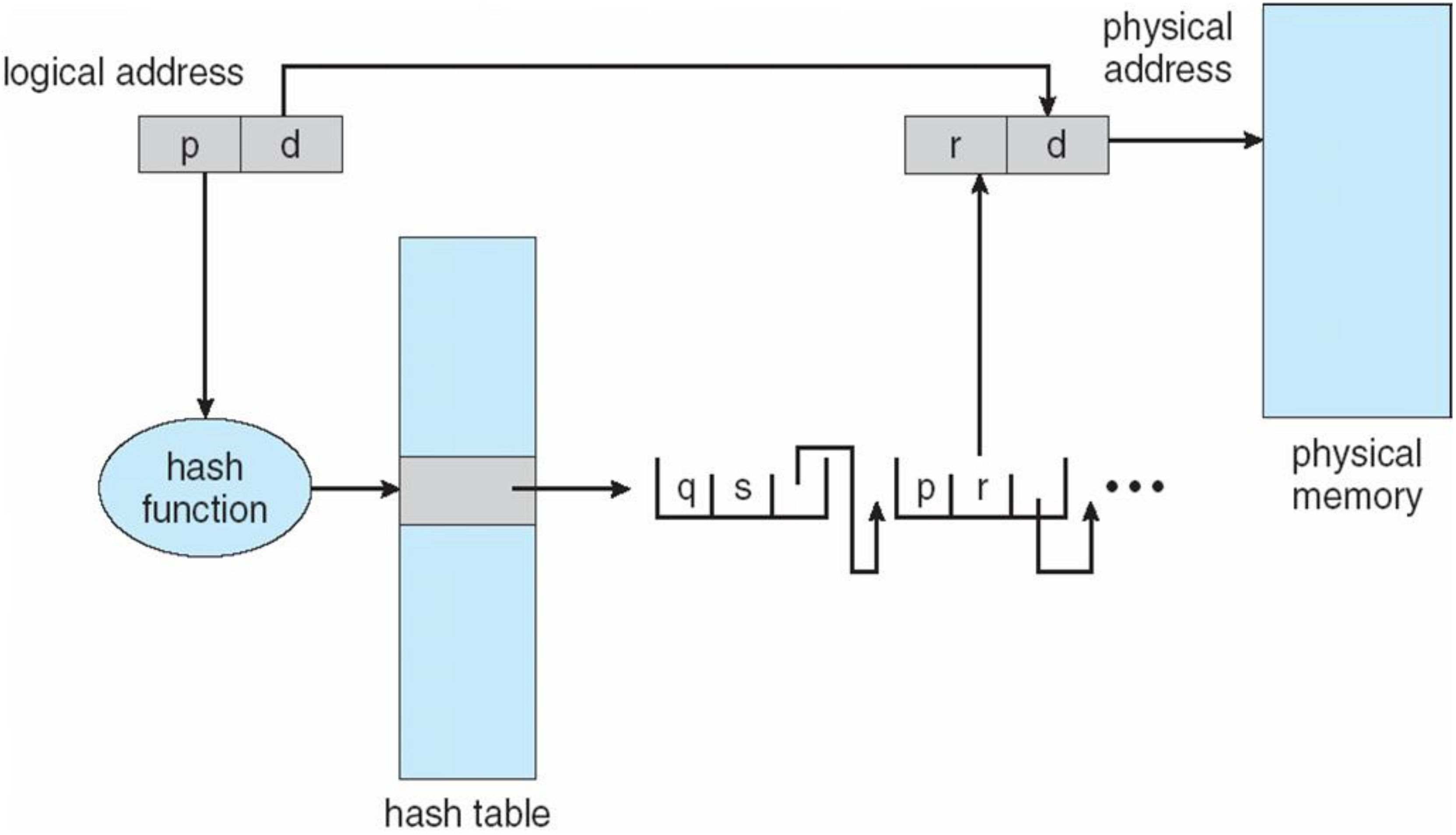
global directory	middle directory	page table	offset



Hashed Page Tables

- **Common in address spaces > 32 bits**
- Size of page table grows proportionally as large as amount of virtual memory allocated to processes
- **Use hash table to limit the cost of search**
  - to one — or at most a few — page-table entries
  - One hash table per process
  - This page table contains a chain of elements hashing to the same location
- **Use this hash table to find the physical page of each logical page**
  - If a match is found, the corresponding physical frame is extracted

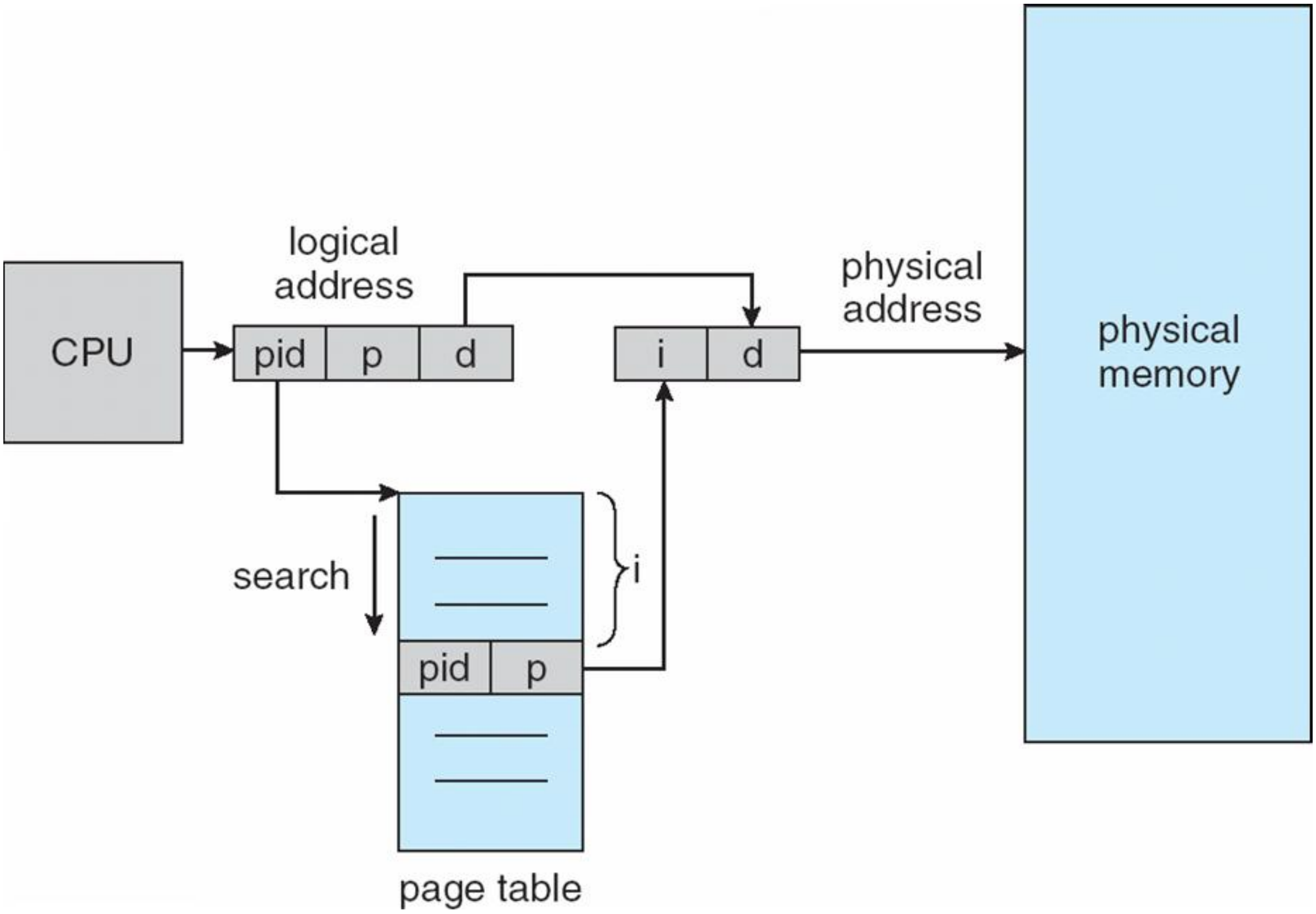
Hashed Page Table



Inverted Page Table

- **One hash table for all processes**
  - 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**

Inverted Page Table Architecture



Address Translation Comparison

<table>
<tr><th></th><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>Segmentation</td><td>Fast context switching: Segment mapping maintained by CPU</td><td>External fragmentation</td></tr>
<tr><td>Paging (single-level page)</td><td>No external fragmentation, fast easy allocation</td><td>Large table size ~ virtual memory
Internal fragmentation</td></tr>
<tr><td>Paged segmentation</td><td rowspan="2">Table size ~ # of pages in virtual memory, fast easy allocation</td><td rowspan="2">Multiple memory references per page access</td></tr>
<tr><td>Two-level pages</td></tr>
<tr><td>Inverted Table</td><td>Table size ~ # of pages in physical memory</td><td>Hash function more complex</td></tr>
</table>

Summary

- **Page Tables**
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- **Usage of page table entries**
  - Page sharing.
  - Copy on write
  - Pages on demand
  - Zero fill on demand
- **Multi-Level Tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- **Inverted page table**
  - Size of page table related to physical memory size