# INFO 324 Operating Systems II

Ahmad FAOUR

# Virtual Memory (1)

# Page Tables

When and why do we access a page table?

- On every instruction to translate virtual to physical addresses?

# Page Tables

When and why do we access a page table?

- On every instruction to translate virtual to physical addresses?

In real machines NO!

In real machines it is only accessed

- *On TLB miss faults to refill the TLB*
- *During process creation and destruction*
- *When a process allocates or frees memory?*

# Translation Lookaside Buffer

Problem: MMU can’t keep up with the CPU if it goes to the page table on every memory access!

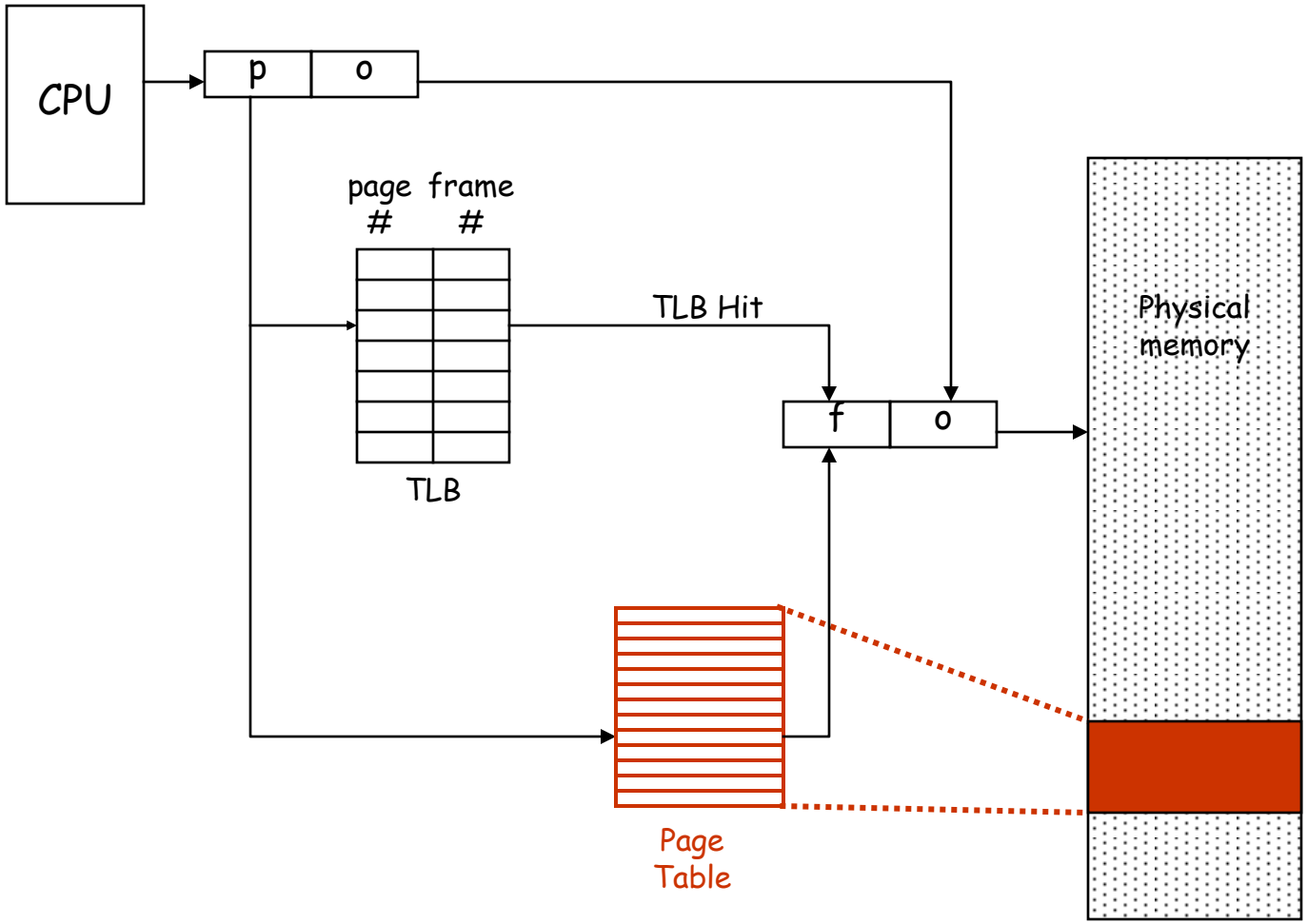
# Translation Lookaside Buffer

Solution:

- Cache the page table entries in a hardware cache
- Small number of entries (e.g., 64)
- Each entry contains page number and other stuff from page table entry
- Associatively indexed on page number

  ie. You can do a lookup in a single cycle

# Translation Lookaside Buffer



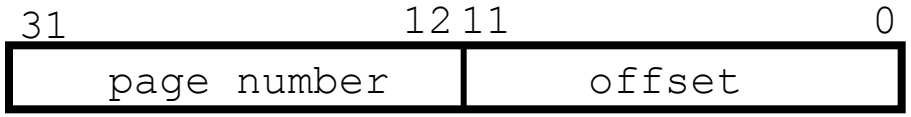
# Hardware Operation of TLB

Key

| Page Number | Frame Number | Other | | | | |
|---|---|---|---|---|---|---|
| 23 | 37 | unused | D | R | W | V |
| 17 | 50 | unused | D | R | W | V |
| 92 | 24 | unused | D | R | W | V |
| 5 | 19 | unused | D | R | W | V |
| 12 | 6 | unused | D | R | W | V |

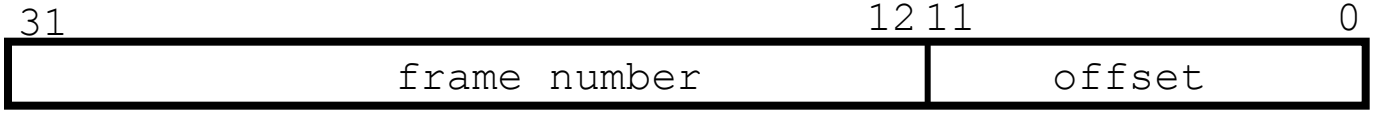
# Hardware Operation of TLB

virtual address



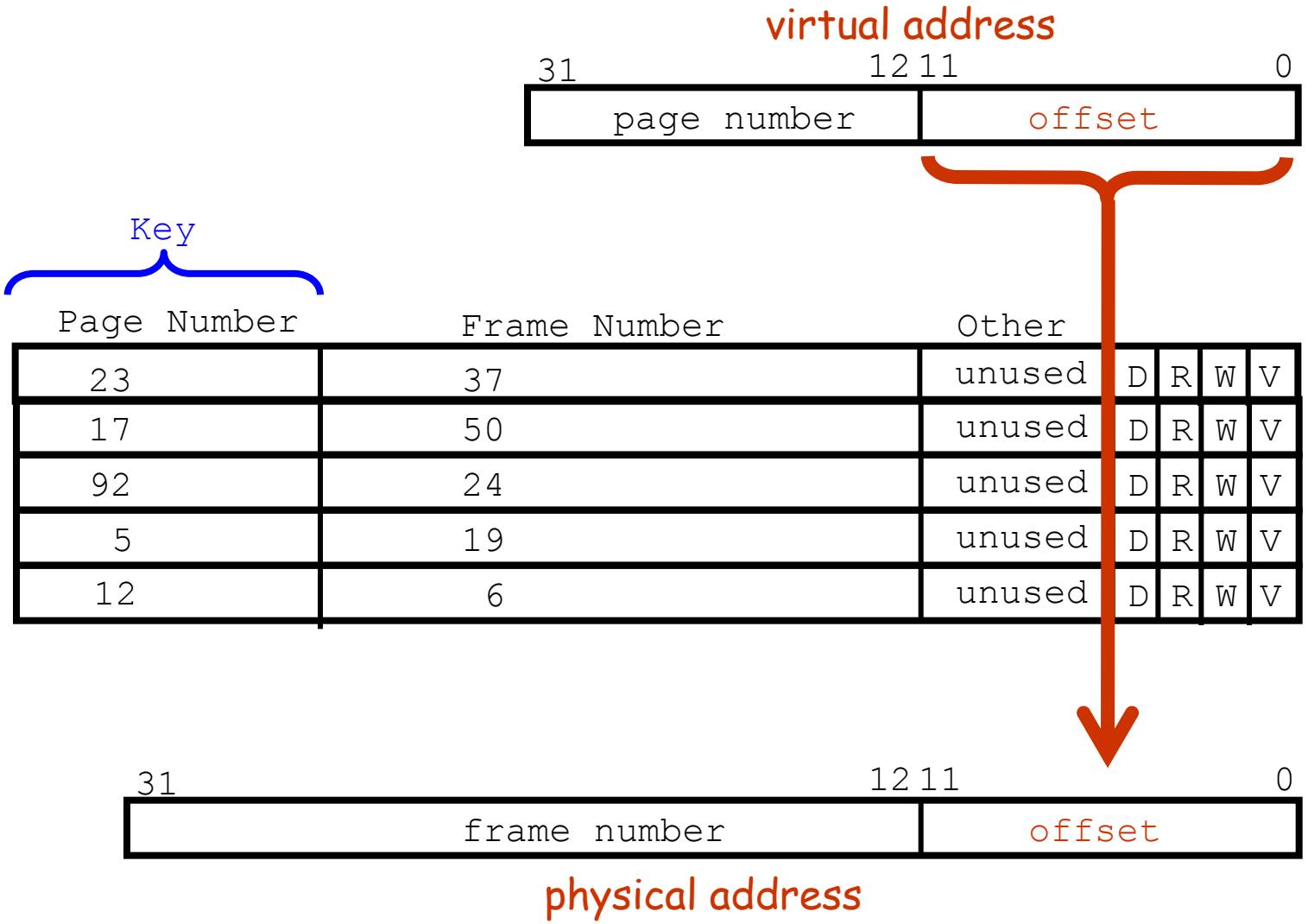

Key

| Page Number | Frame Number | Other | | | | |
|---|---|---|---|---|---|---|
| 23 | 37 | unused | D | R | W | V |
| 17 | 50 | unused | D | R | W | V |
| 92 | 24 | unused | D | R | W | V |
| 5 | 19 | unused | D | R | W | V |
| 12 | 6 | unused | D | R | W | V |

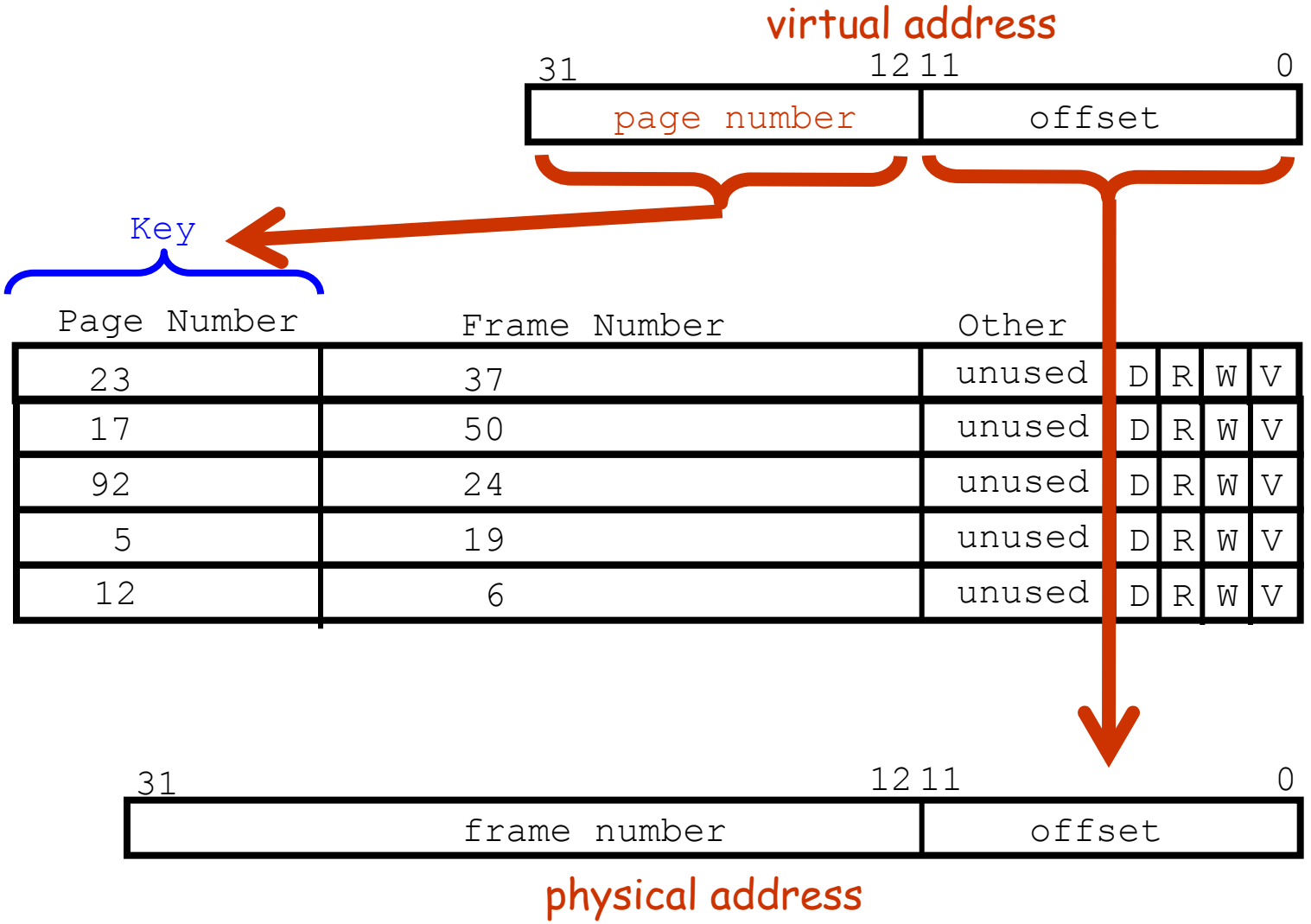



physical address

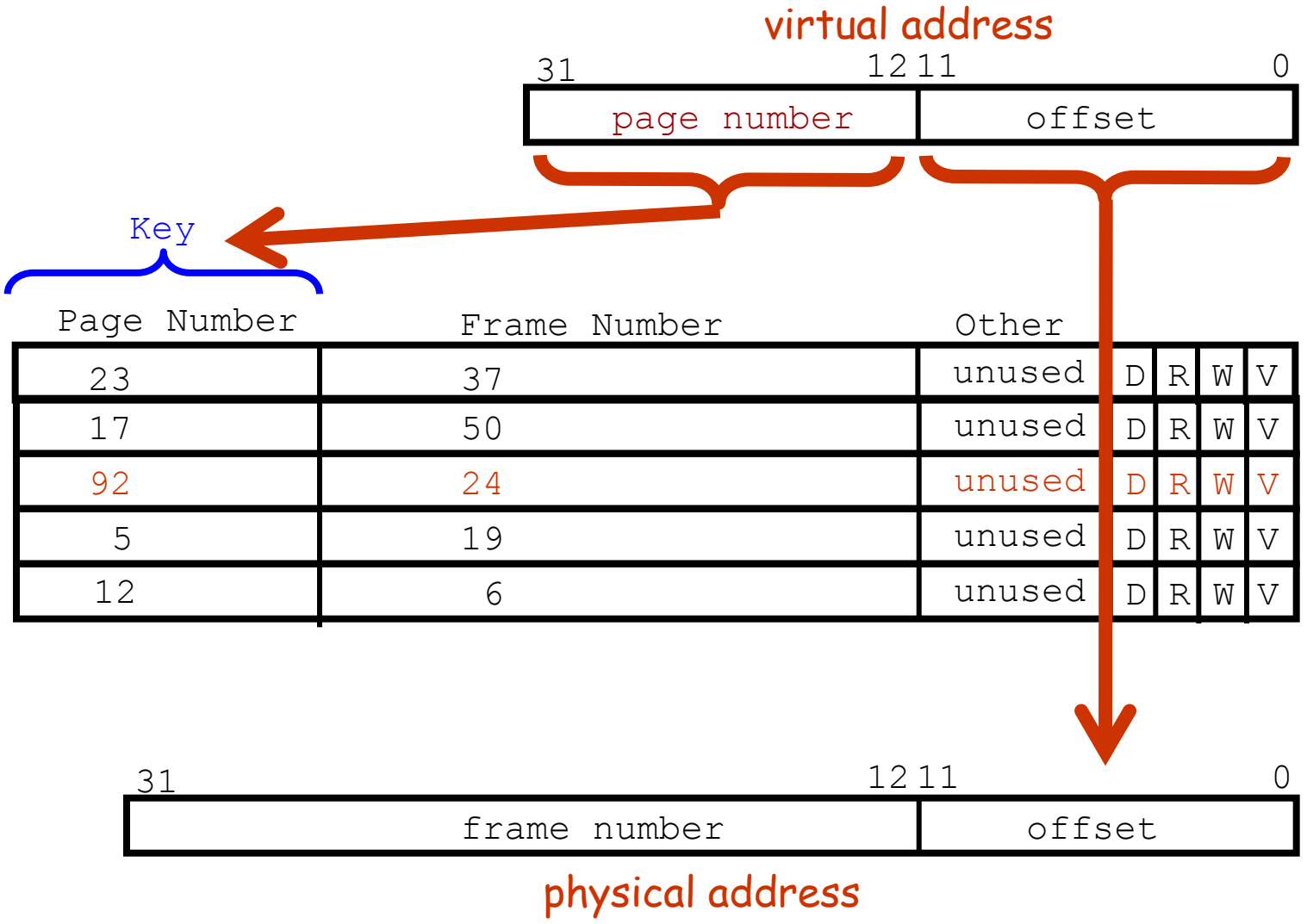
# Hardware Operation of TLB



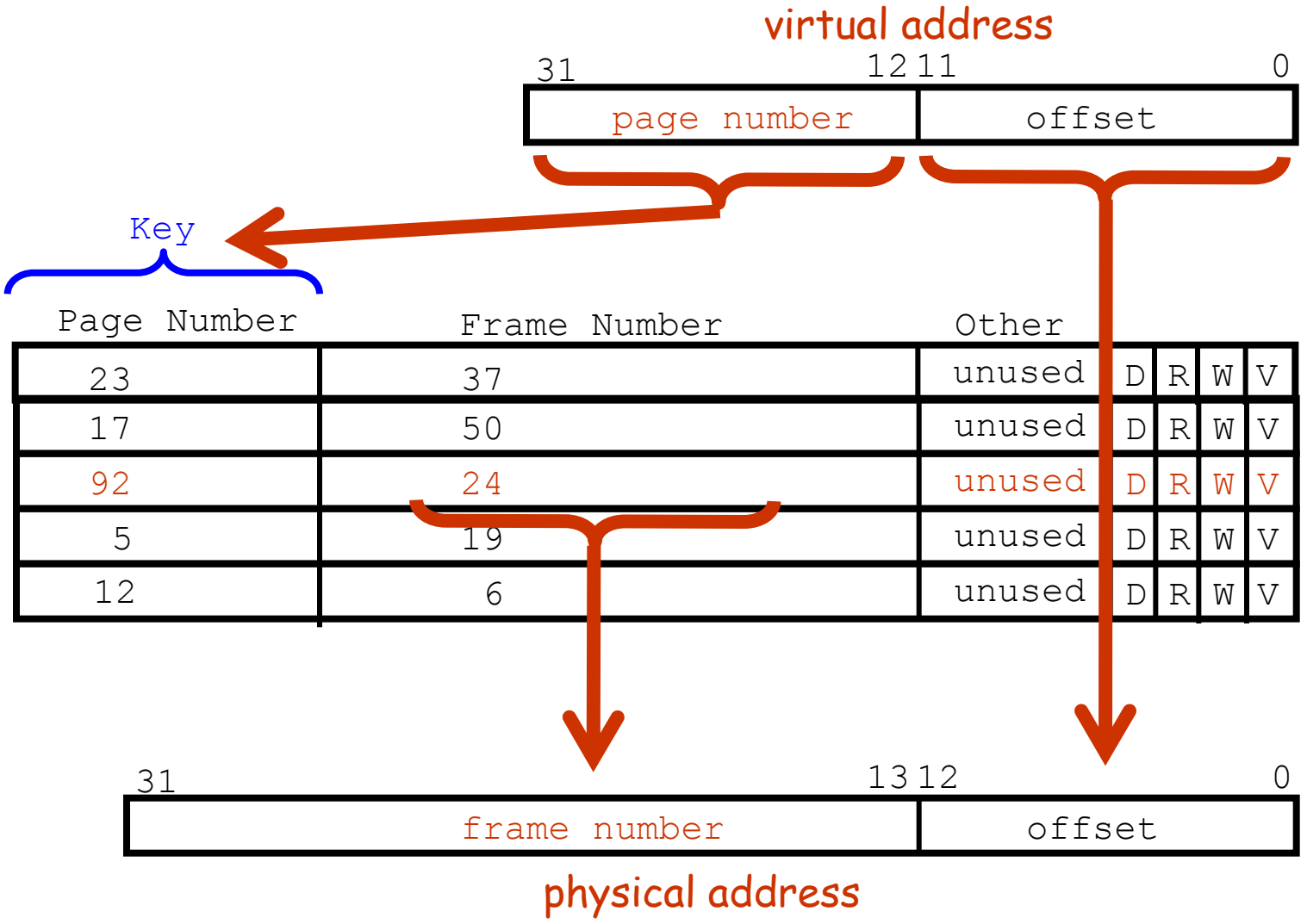
# Hardware Operation of TLB



# Hardware Operation of TLB



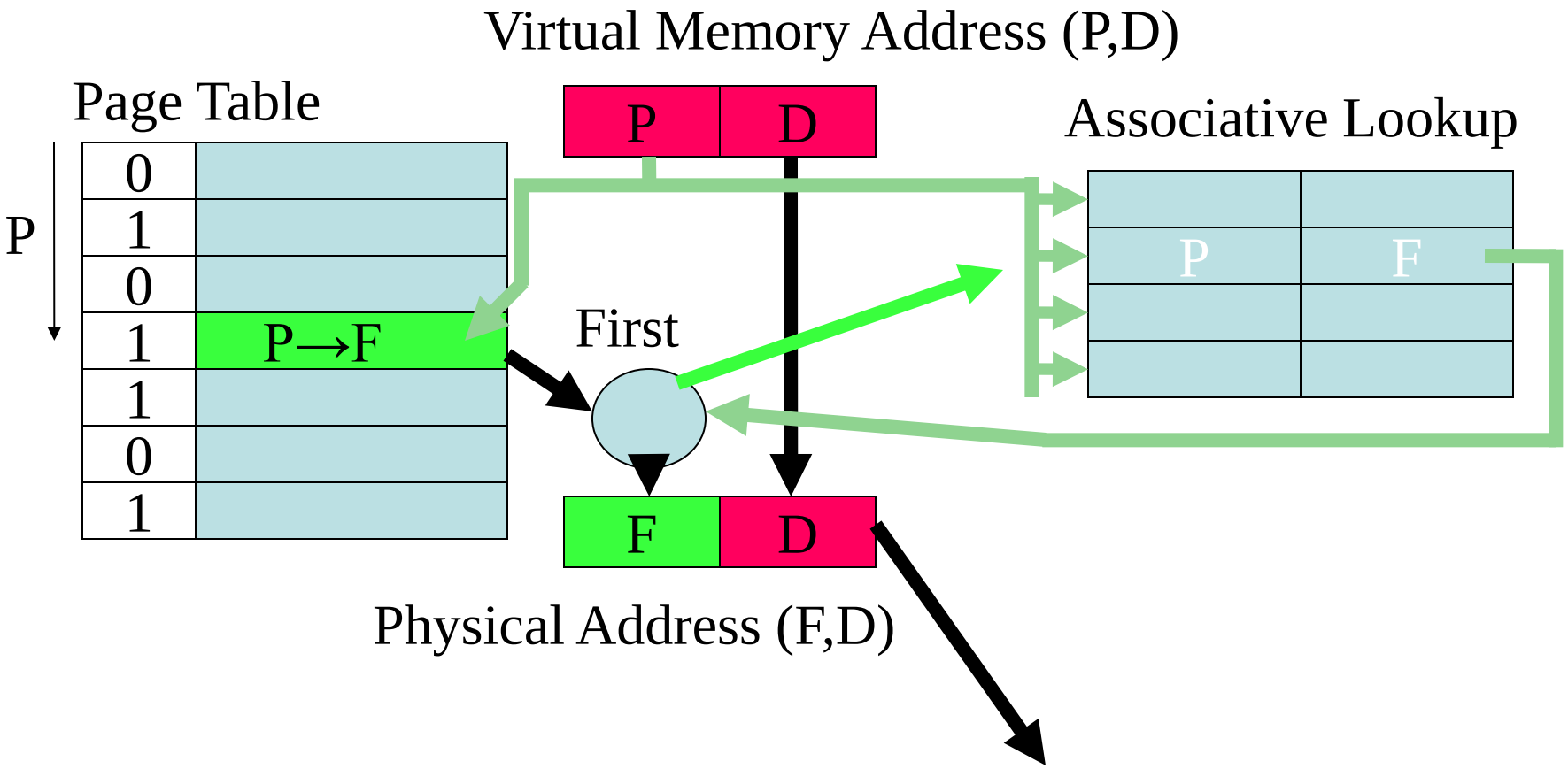
# Hardware Operation of TLB



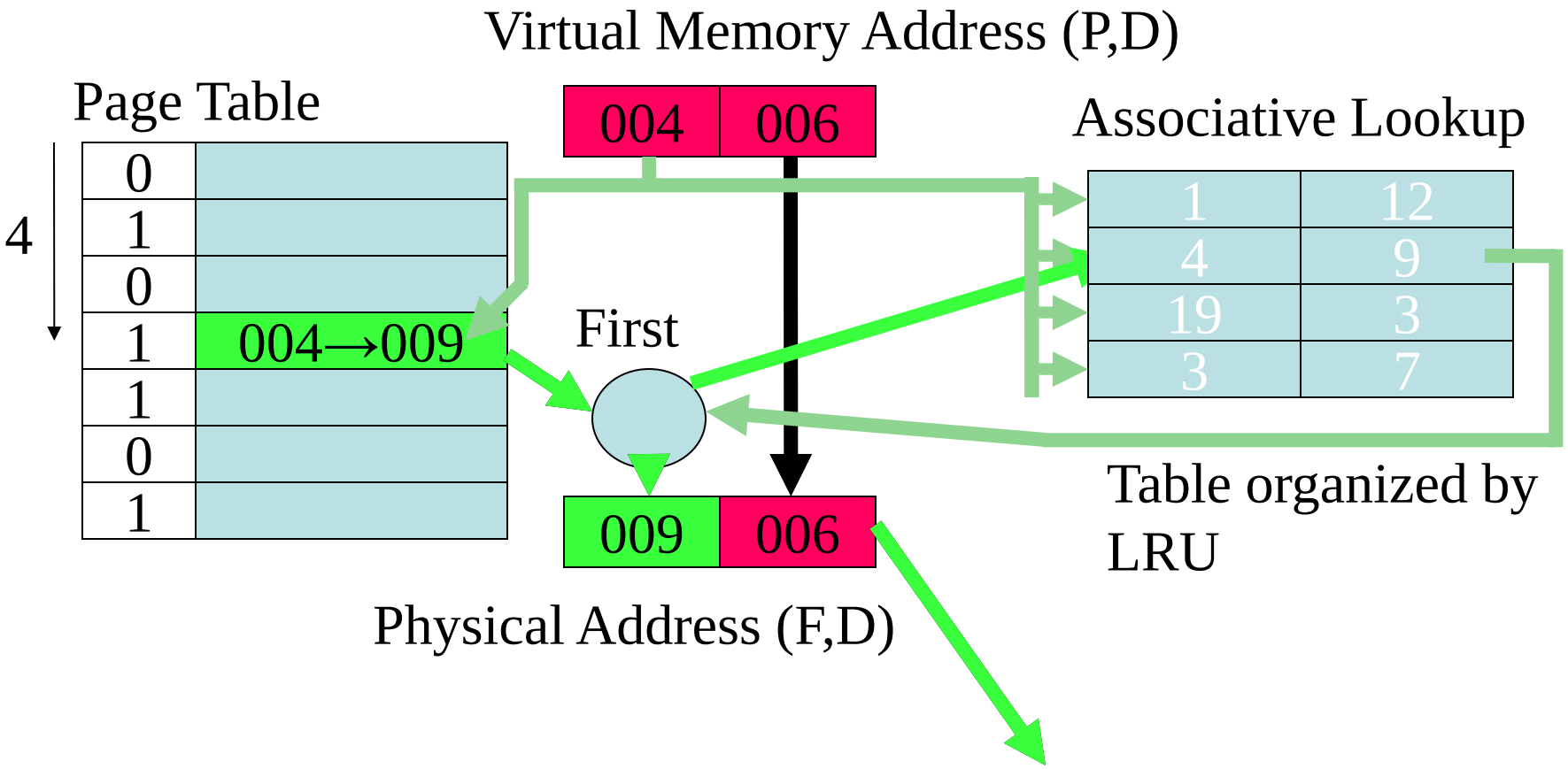
# TLB Function

- If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).
- If match is valid, the page is taken from TLB without going through page table.
- If match is not valid
  - MMU detects miss and does an ordinary page table lookup.
  - It then evicts one page out of TLB and replaces it with the new entry, so that next time that page is found in TLB.

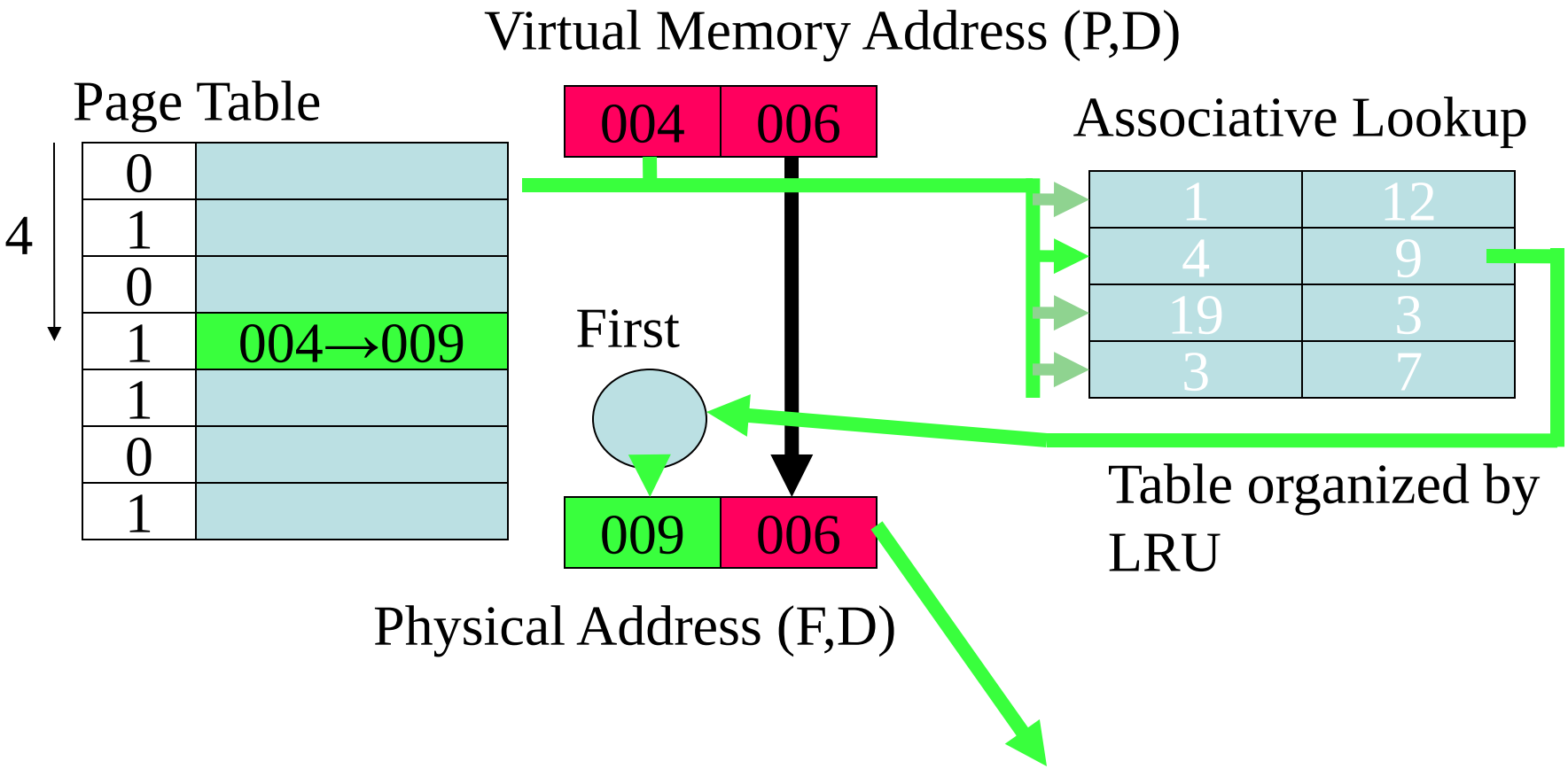
# Page Mapping Hardware



# Page Mapping Example

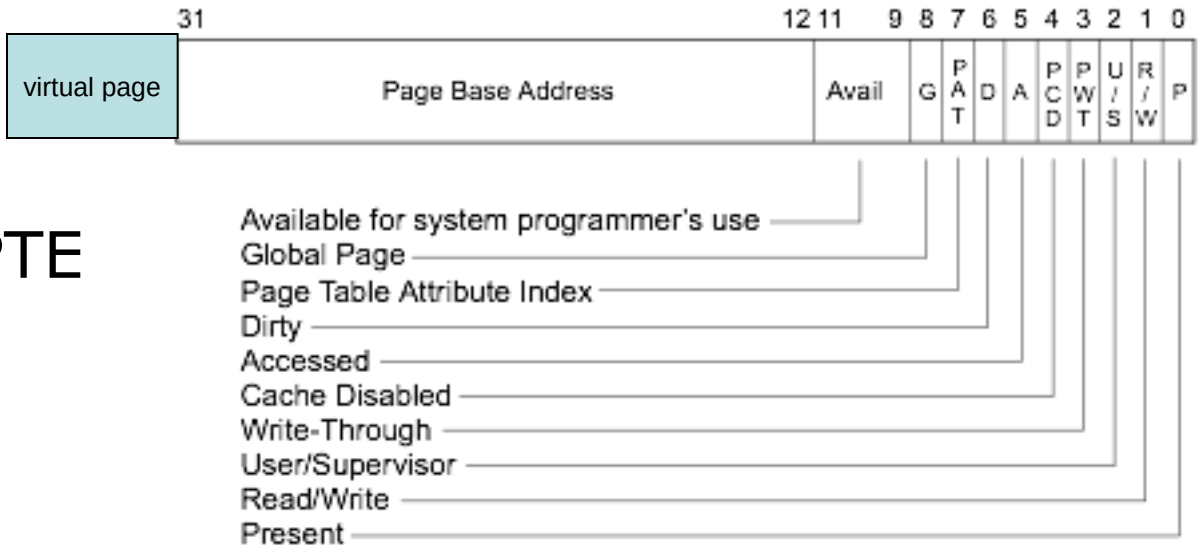


# Page Mapping Example: next reference



# Bits in a TLB Entry

- Common (necessary) bits
  - Virtual page number: match with the virtual address
  - Physical page number: translated address
  - Valid
  - Access bits: kernel and user (nil, read, write)
- Optional (useful) bits
  - Process tag
  - Reference
  - Modify
  - Cacheable
- Includes parts of PTE
  - Example: x86



# Page Table Design

Page table size depends on

- Page size
- Virtual address length

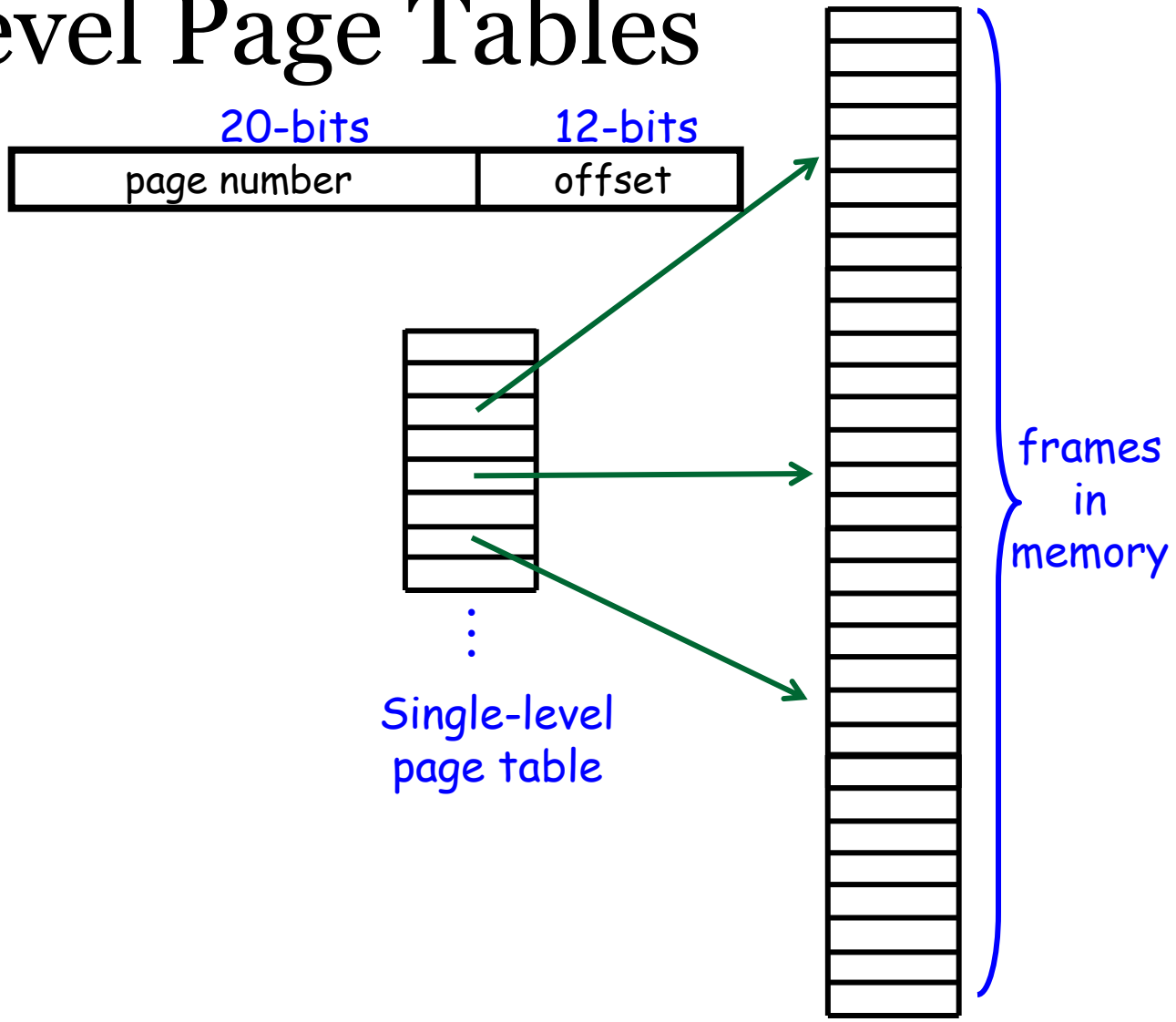
Memory used for page tables is overhead!

- How can we save space? … and still find entries quickly?

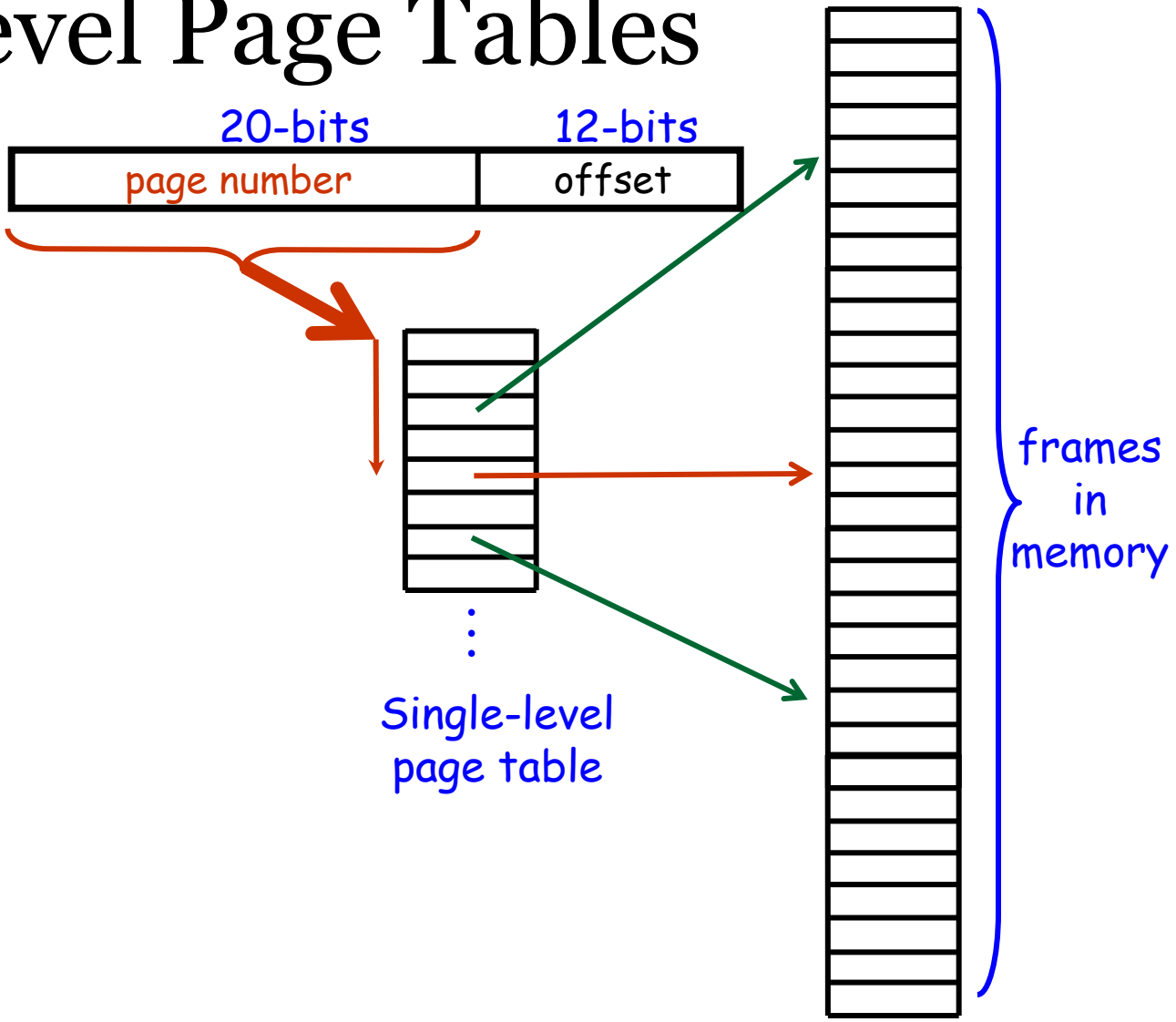
Three options

- Single-level page tables
- Multi-level page tables
- Inverted page tables

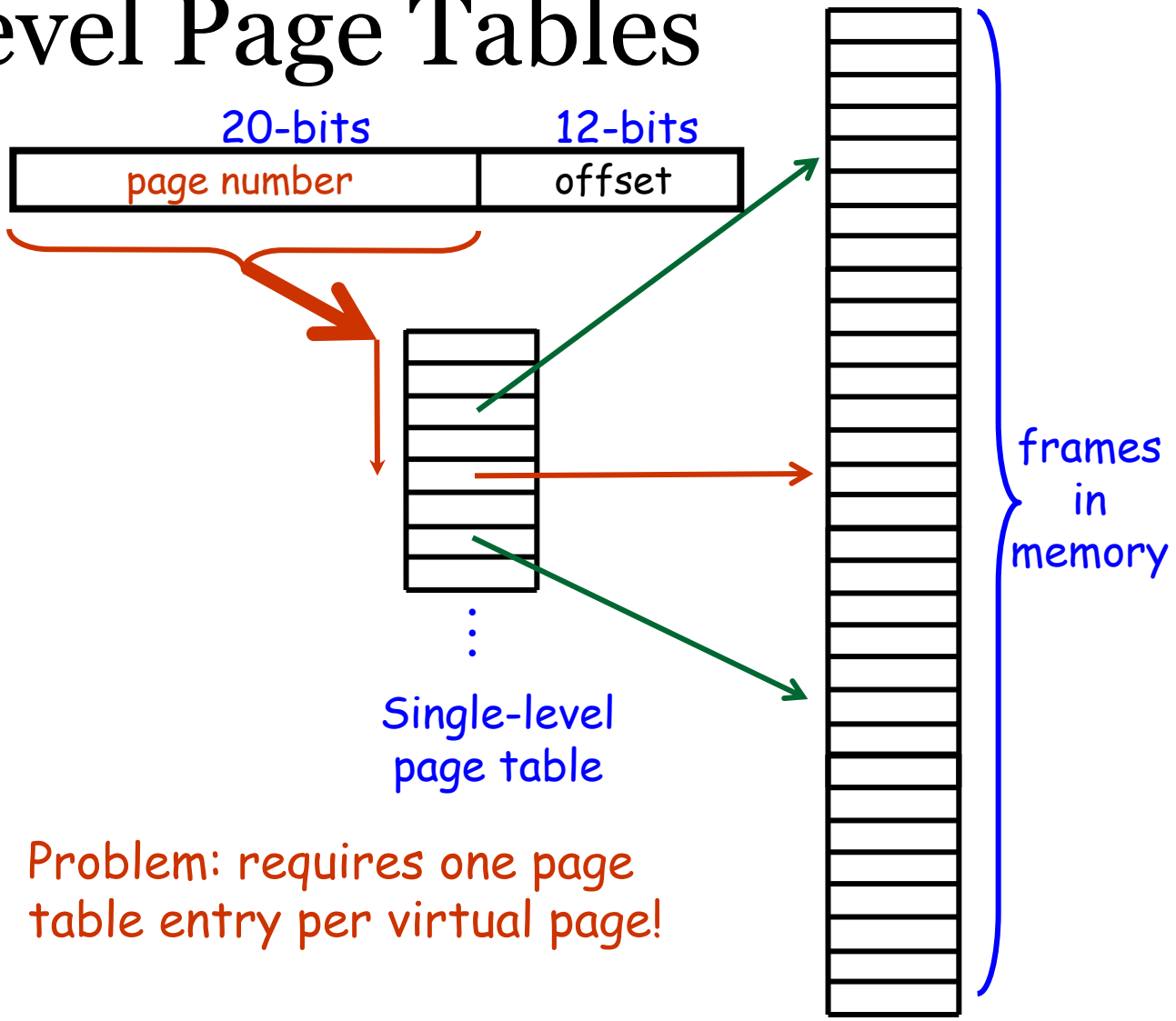
# Single-Level Page Tables



# Single-Level Page Tables

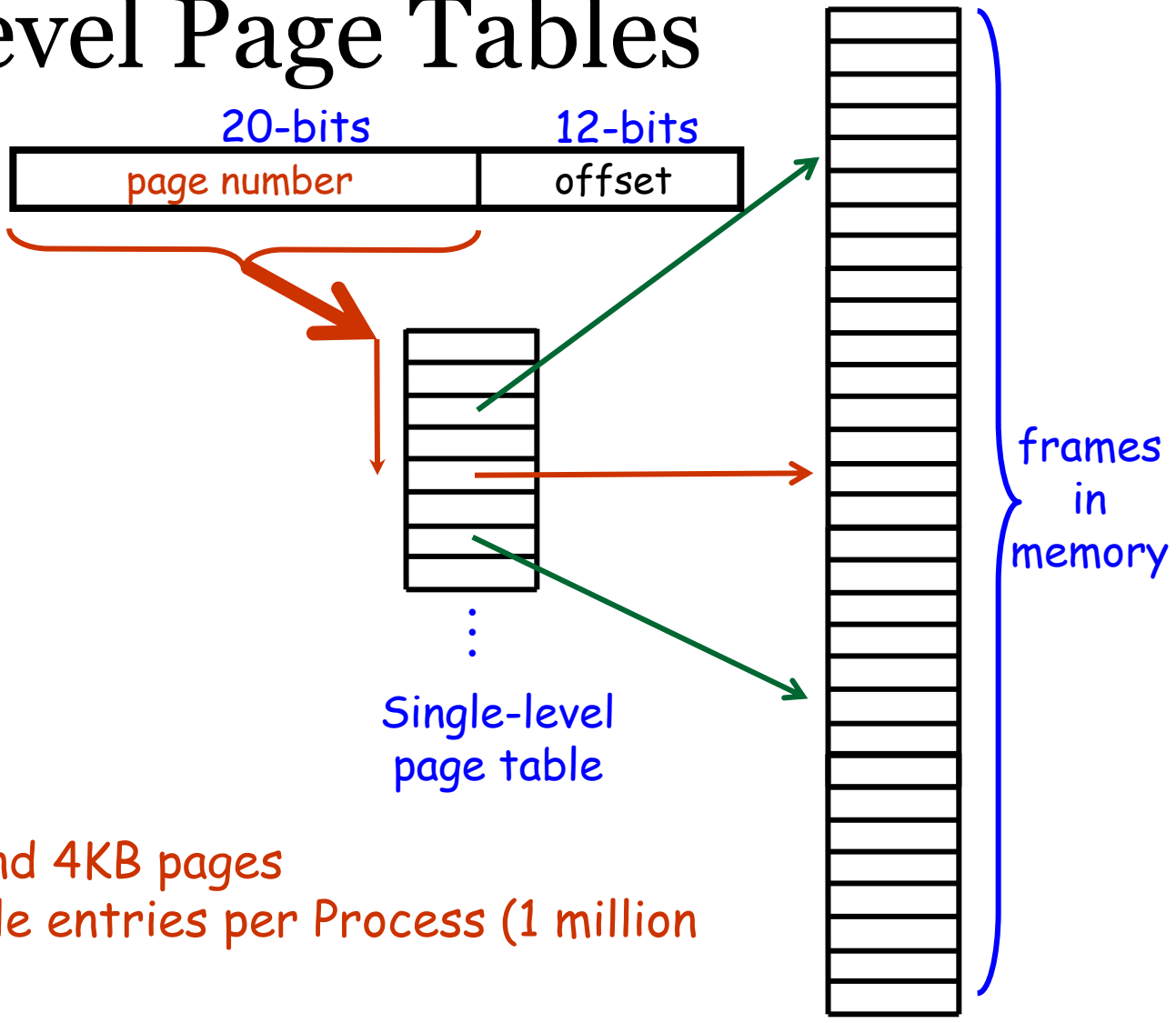


# Single-Level Page Tables



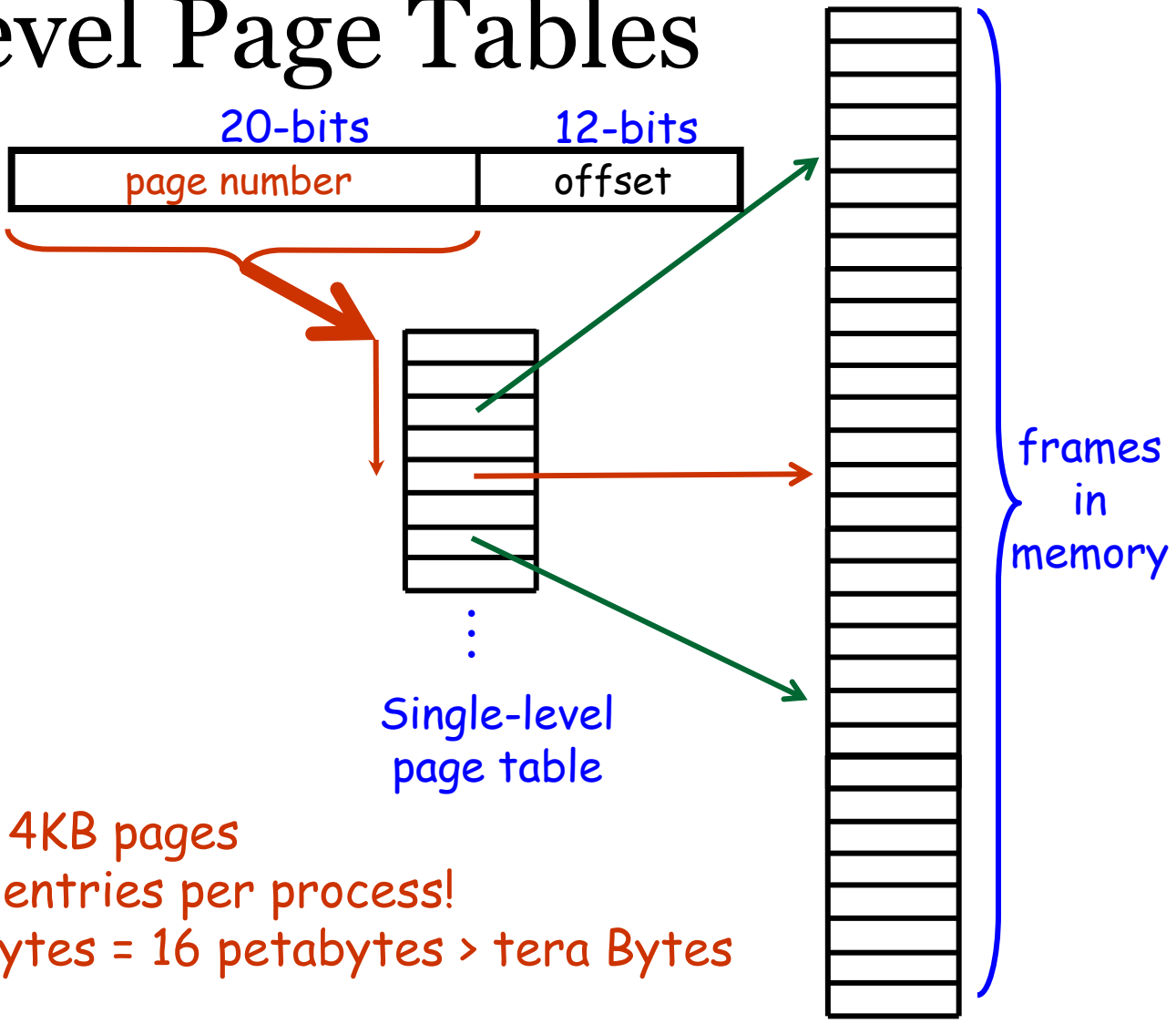

Problem: requires one page table entry per virtual page!

# Single-Level Page Tables




32 bit addresses and 4KB pages
means $2^{20}$ page table entries per Process (1 million entries ~4MB)

# Single-Level Page Tables

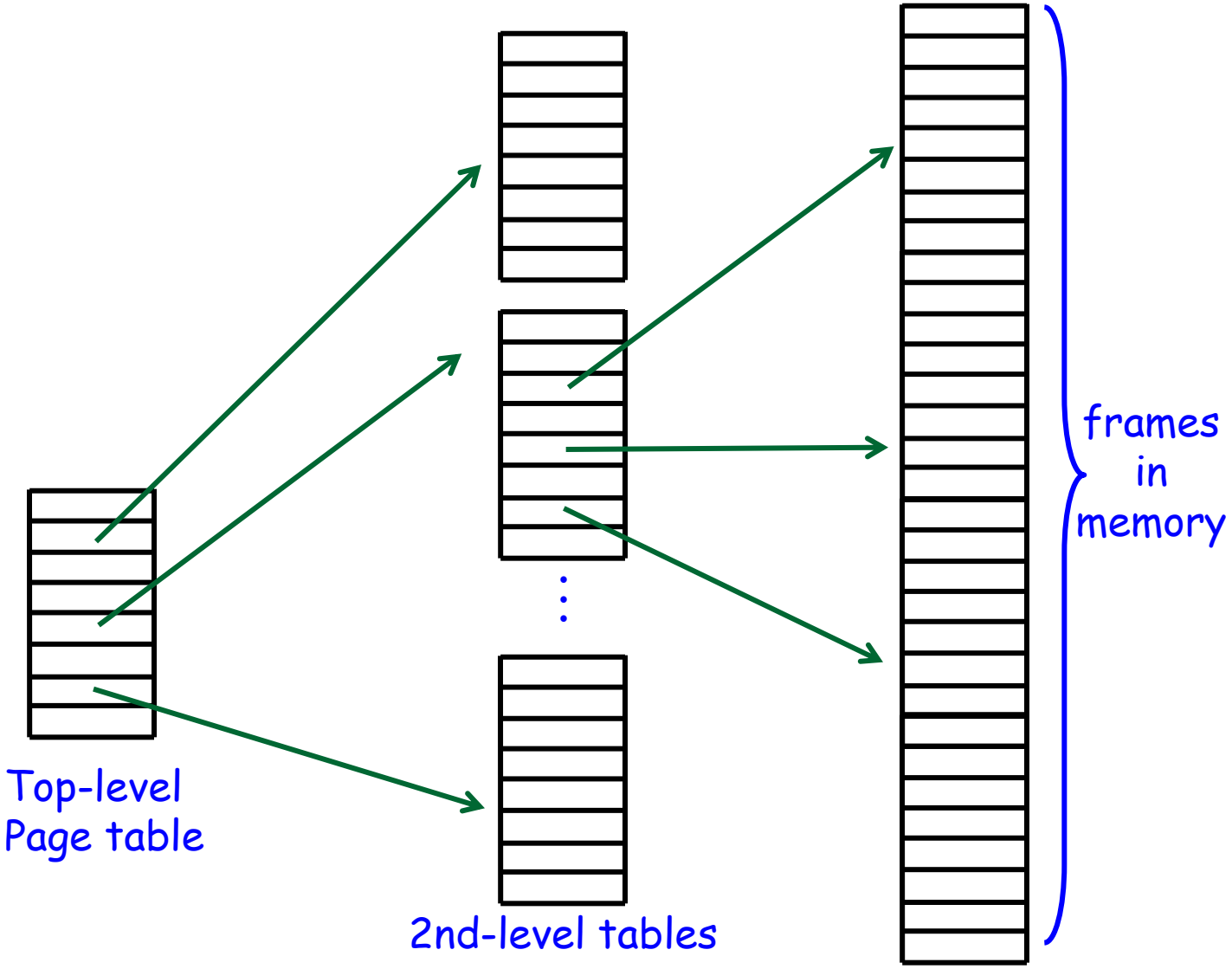



64 bit addresses and 4KB pages
means $2^{52}$ page table entries per process!
$2^{52}$ * 4 bytes = $2^{54}$ bytes = 16 petabytes > tera Bytes

# Multilevel Page Tables

- Since the page table can be very large, one solution is to **page** the page table
- Divide the page number into
  - An index into a page table of second level page tables
  - A page within a second level page table
- Advantage
  - No need to keep all page tables in memory all the time
  - Only recently accessed memory's mapping need to be kept in memory, the rest can be fetched on demand

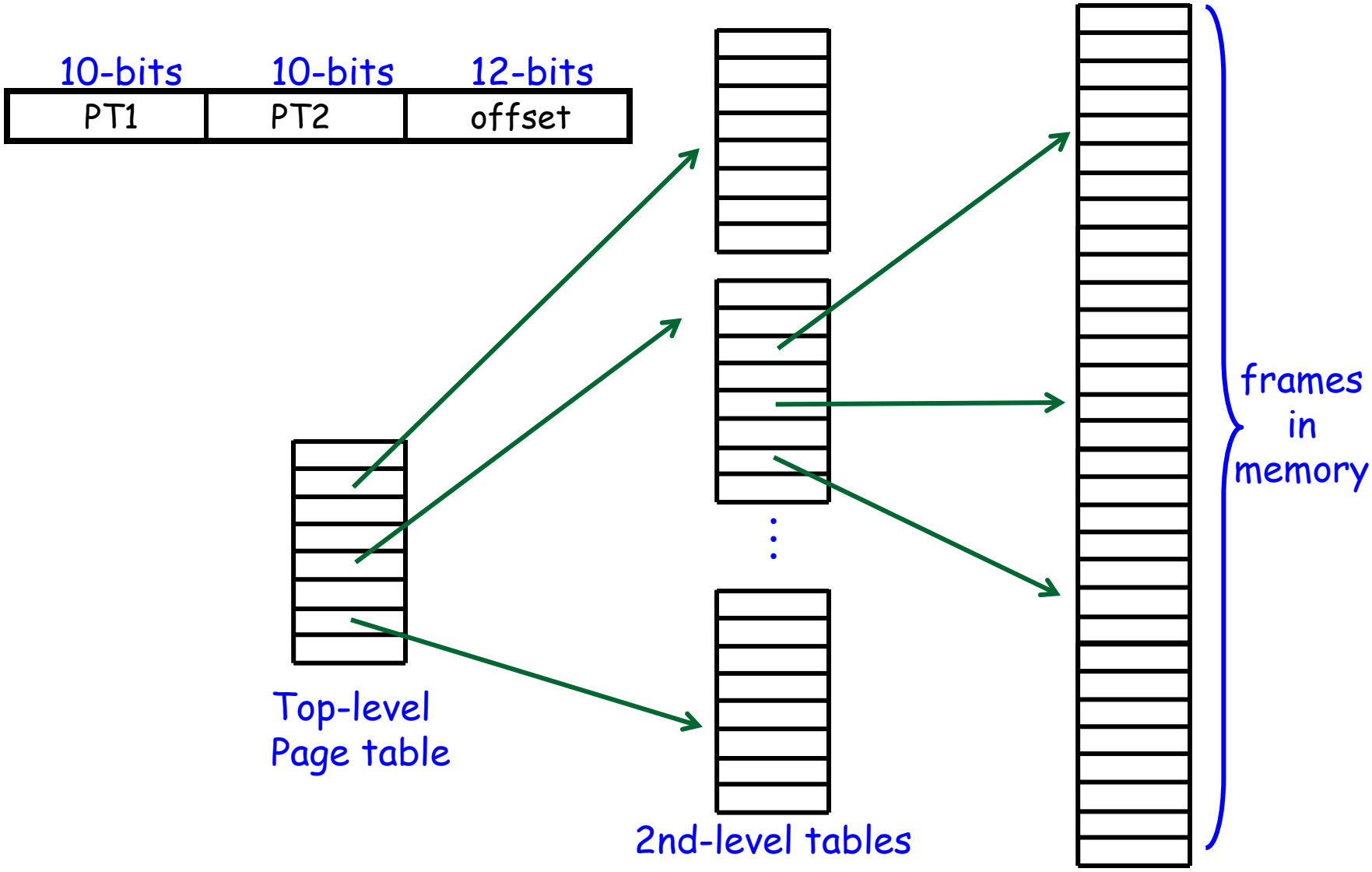
# Multi-Level Page Tables



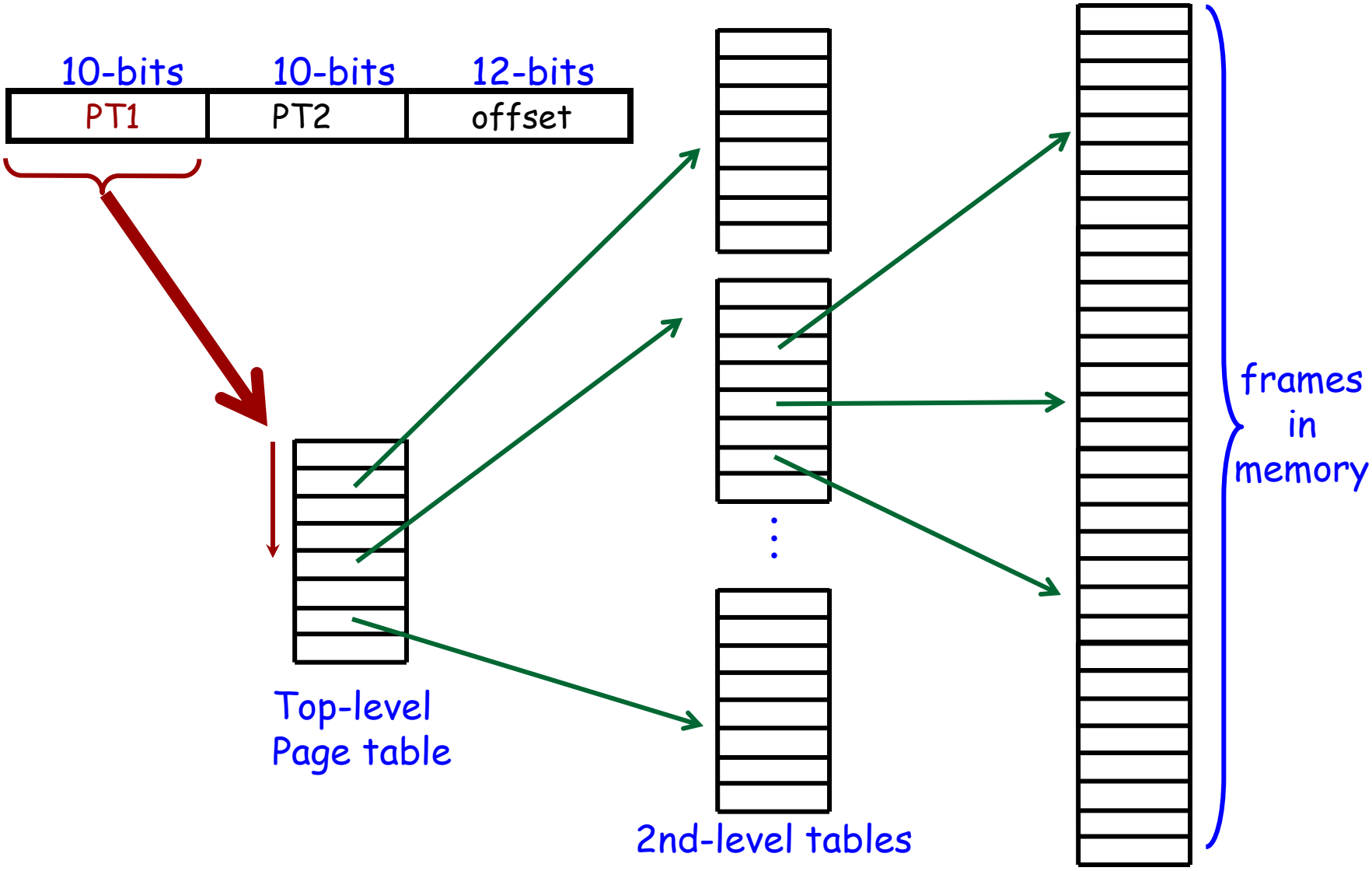
# Multilevel page tables

- **Multilevel page tables** avoid keeping one huge page table in memory all the time.
- This works because most processes use only a few of its pages frequently and the rest, seldom if at all. Scheme: the page table itself is paged.
- EX. Using 32 bit addressing:
  - The top-level table contains 1,024 pages (indices).
  - The entry at each index contains the page frame number of a 2nd-level page table.
  - This index (or page number) is found in the 10 highest (leftmost) bits in the virtual address generated by the CPU.
  - The next 10 bits in the address hold the index into the 2nd-level page table. This location holds the page frame number of the page itself.
  - The lowest 12 bits of the address is the offset, as usual.

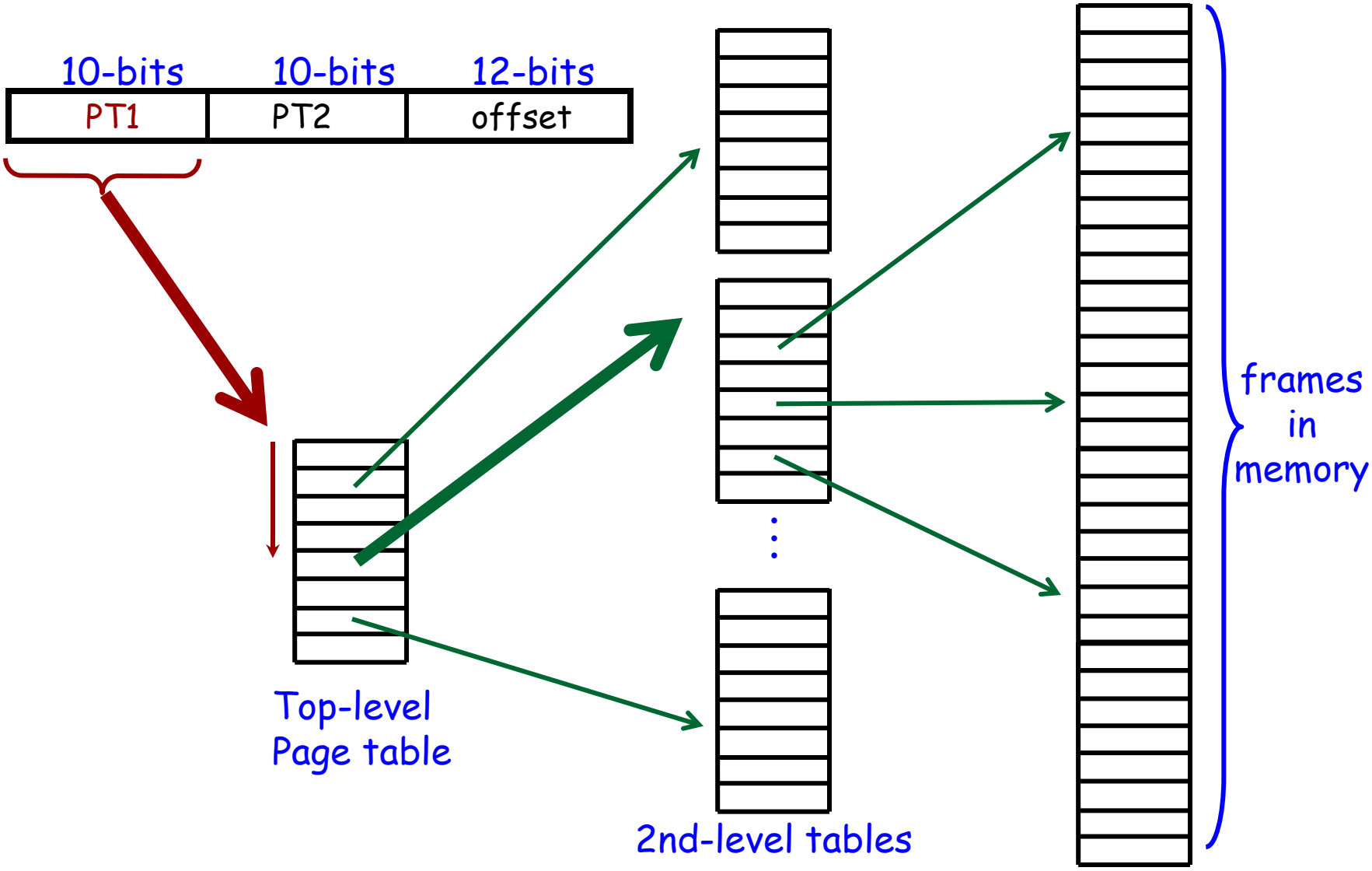
# Multi-Level Page Tables



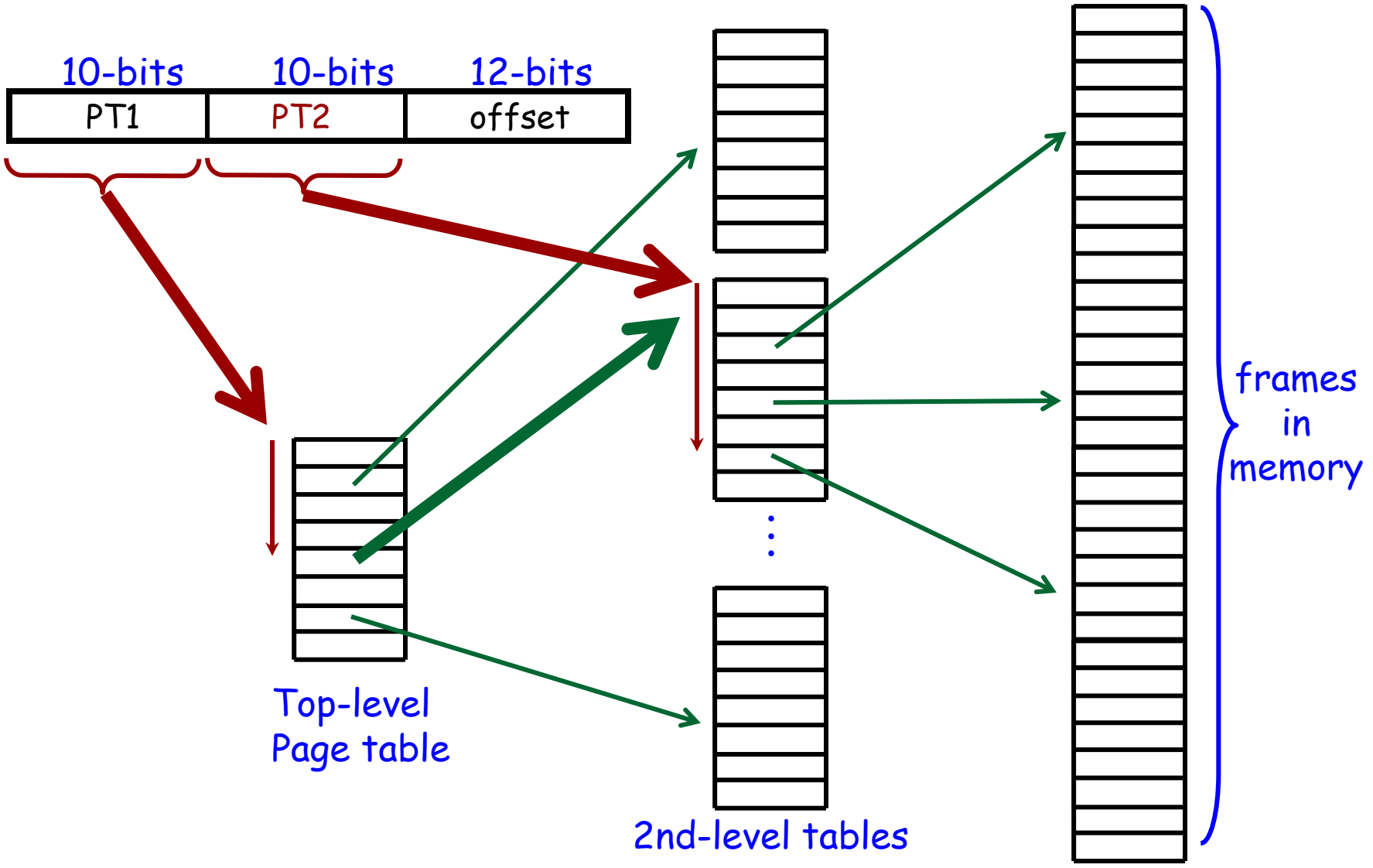
# Multi-Level Page Tables



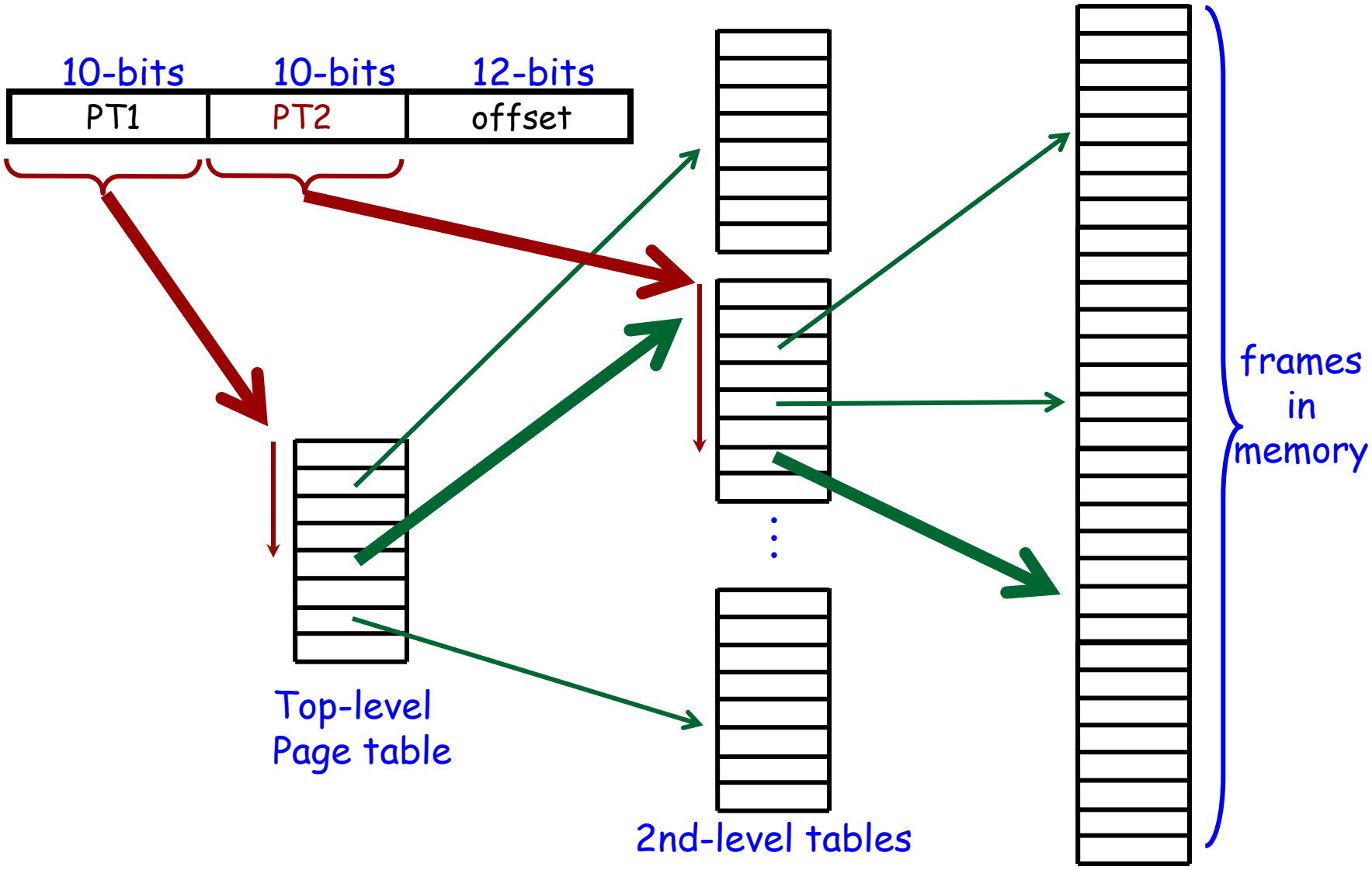
# Multi-Level Page Tables



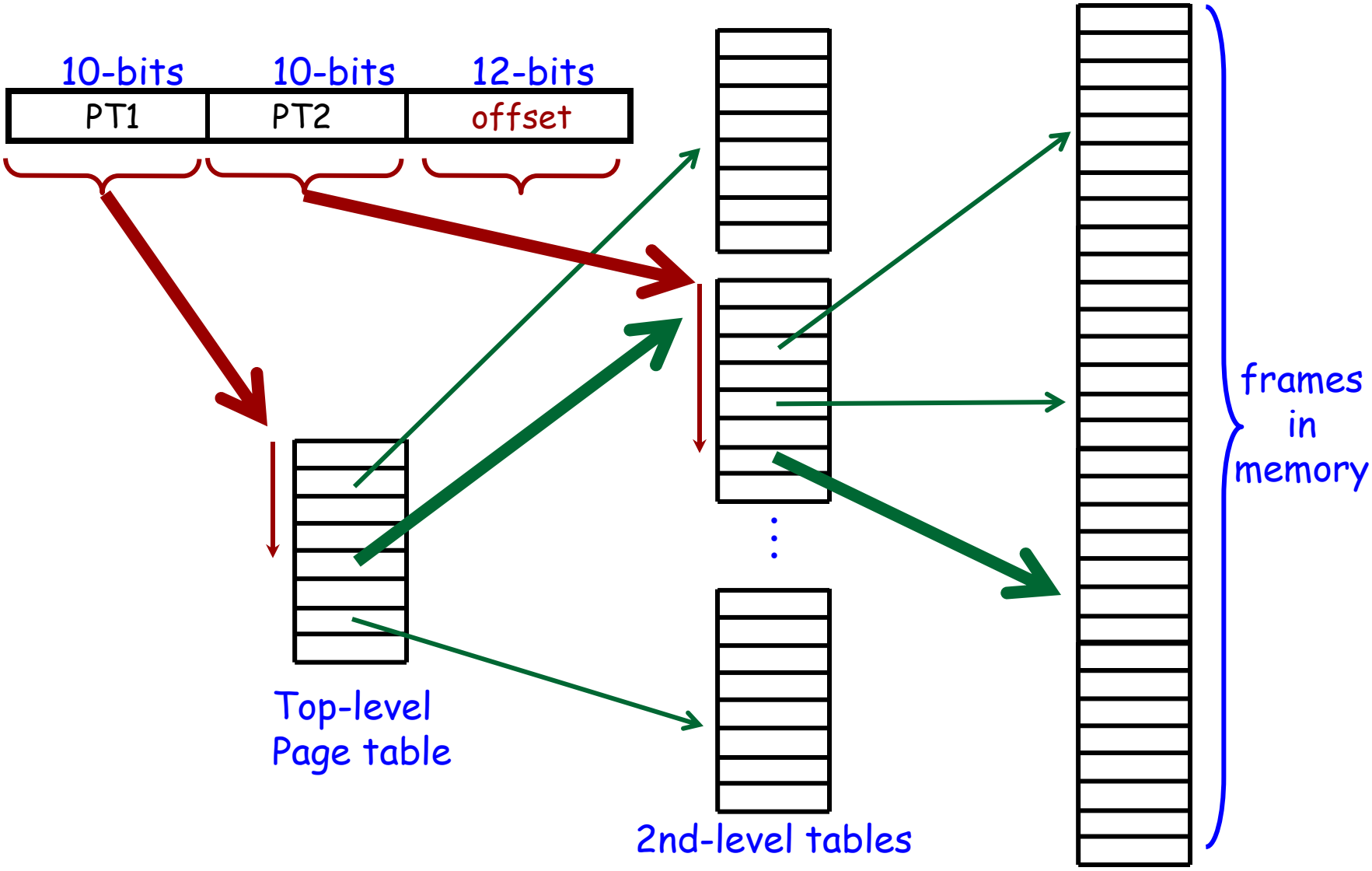
# Multi-Level Page Tables



# Multi-Level Page Tables



# Multi-Level Page Tables



# Two-level Page Tables (Example)

**Ex. Given 32 bit virtual address 00403004 (hex) = 4,206,596 (dec)**

converting to binary we have:

0000 0000 0100 0000 0011 0000 0000 0100

regrouping 10 highest bits, next 10 bits, remaining 12 bits:

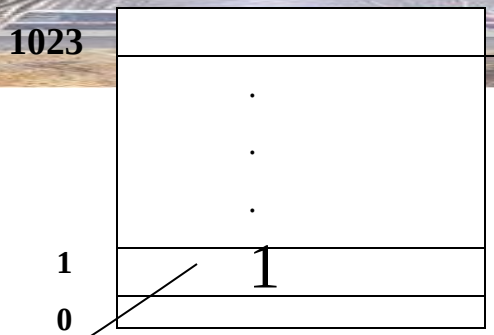
| | | |
|---|---|---|
| 0000 0000 01 | 00 0000 0011 | 0000 0000 0100 |
| PT1 = 1 | PT2 = 3 | offset = 4 |

PT1 = 1 => go to index 1 in top-level page table. Entry here is the page frame number of the 2nd-level page table. (entry =1 in this ex.)

PT2 = 3 => go to index 3 of 2nd-level table 1. Entry here is the no. of the page frame that actually contains the address in physical memory. (entry=3 in this ex.) The address is found using the offset from the beginning of this page frame. (Remember each page frame corresponds to 4096 addresses of bytes of memory.)

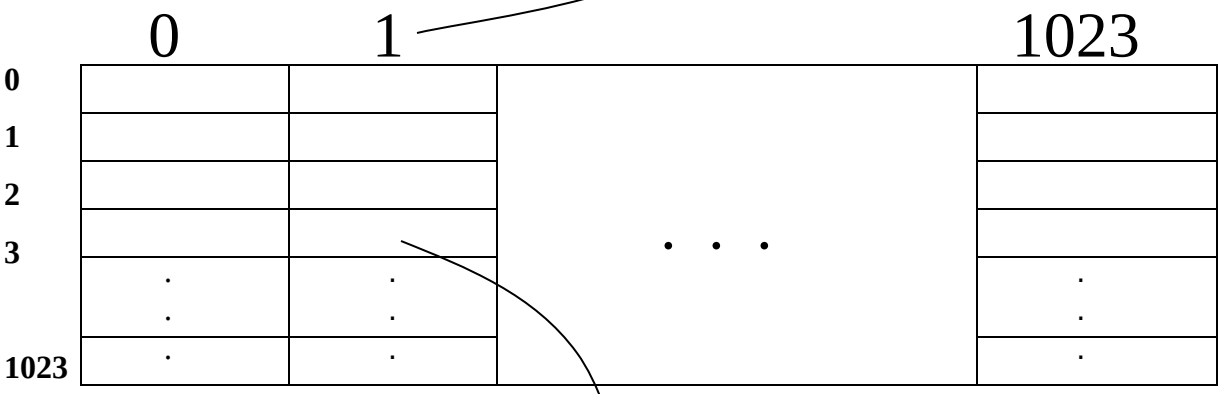
# Diagram of previous example:

Top-level page table:




Corresponds to all possible virtual addresses with 32bit addresses:
0 - 4,294,967,295(dec)

2nd-level page table:




Each page ~ 4K

each chunk ~ 4M

(~4000*1000)

**Corresponds to addresses 0 - 4, 194, 303**

**Corresponds to addresses 4, 194, 304 - 8,388,608**

3

**12,292**

**Corresponds to bytes 12,288 -**

**16,384 from beginning of page table 1**

**Offset 4 + 12,288 = 12,292**
**(corresponds to absolute address 4, 206, 596)**

# Multi-Level Page Tables

Ok, but how exactly does this save space?

# Multi-Level Page Tables

Ok, but how exactly does this save space?

Not all pages within a virtual address space are allocated

- Not only do they not have a page frame, but that range of virtual addresses is not being used
- So no need to maintain complete information about it
- Some intermediate page tables are empty and not needed

We could also page the page table

- This saves space but slows access … a lot!