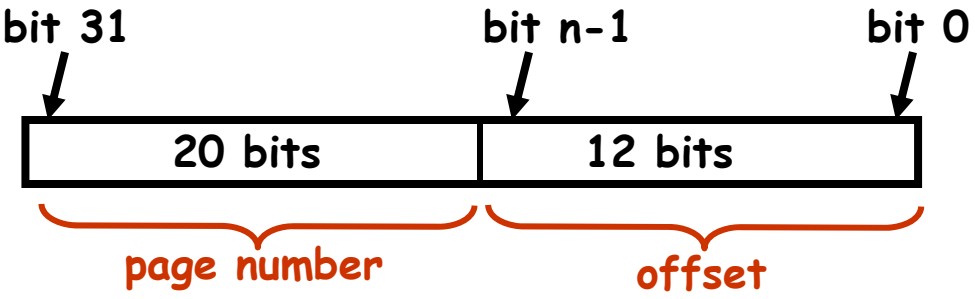
CS 333
Introduction to Operating Systems

Class 11 – Virtual Memory (1)

Jonathan Walpole
Computer Science
Portland State University

Virtual addresses

- **Virtual memory addresses (what the process uses)**
  - Page number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page number



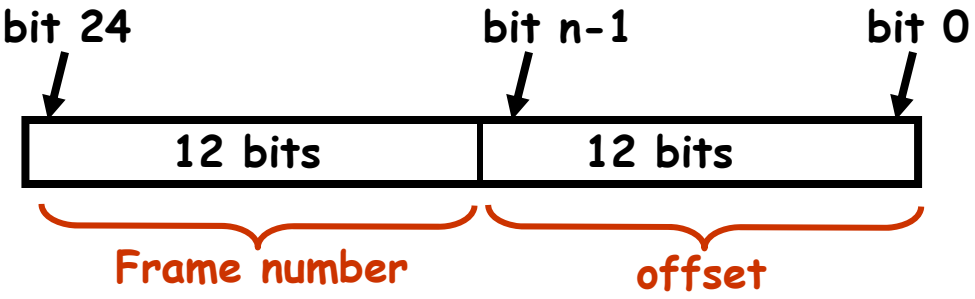
Example: 32 bit virtual address

Page size = 2^{12} = 4KB

Address space size = 2^{32} bytes = 4GB

Physical addresses

- **Physical memory addresses (what memory uses)**
  - Frame number plus byte offset in frame
  - Low order n bits are the byte offset
  - Remaining high order bits are the frame number



Example: 24 bit physical address

Frame size = 2^{12} = 4KB

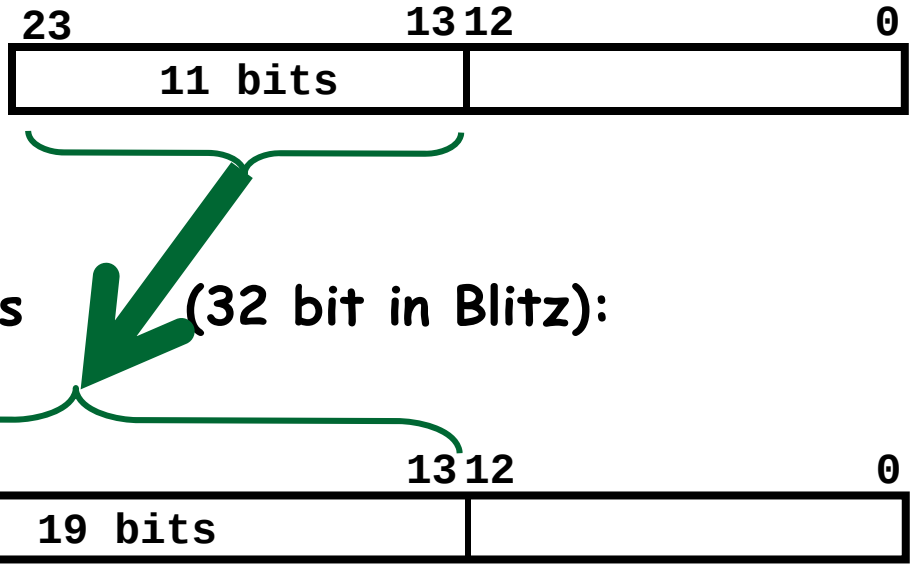
Max physical memory size = 2^{24} bytes = 16MB

Address translation

- **Complete set of address mappings for a process are stored in a page table in memory**
  - But accessing the table for every address translation is too expensive
  - So hardware support is used to map page numbers to frame numbers at full CPU speed
    - *Memory management unit (MMU) has multiple registers for multiple pages and knows how to access page tables*
    - *Also called a translation look aside buffer (TLB)*
    - *Essentially a cache of page table entries*

The BLITZ architecture

- **The page table mapping:**
  - Page --> Frame
- **Virtual Address (24 bit in Blitz):**



- **Physical Address (32 bit in Blitz):**

The BLITZ page table

- **An array of "*page table entries*"**
  - Kept in memory
- **2^{11} pages in a virtual address space**
  - ---> 2K entries in the table
- **Each entry is 4 bytes long**
  - 19 bits The Frame Number
  - 1 bit Valid Bit
  - 1 bit Writable Bit
  - 1 bit Dirty Bit
  - 1 bit Referenced Bit
  - 9 bits Unused (and available for OS algorithms)

The BLITZ page table

- **Two page table related registers in the CPU**
  - Page Table Base Register
  - Page Table Length Register
- **These define the page table for the "current" process**
  - Must be saved and restored on process context switch
- **Bits in the CPU "status register"**

  "System Mode"

  "Interrupts Enabled"

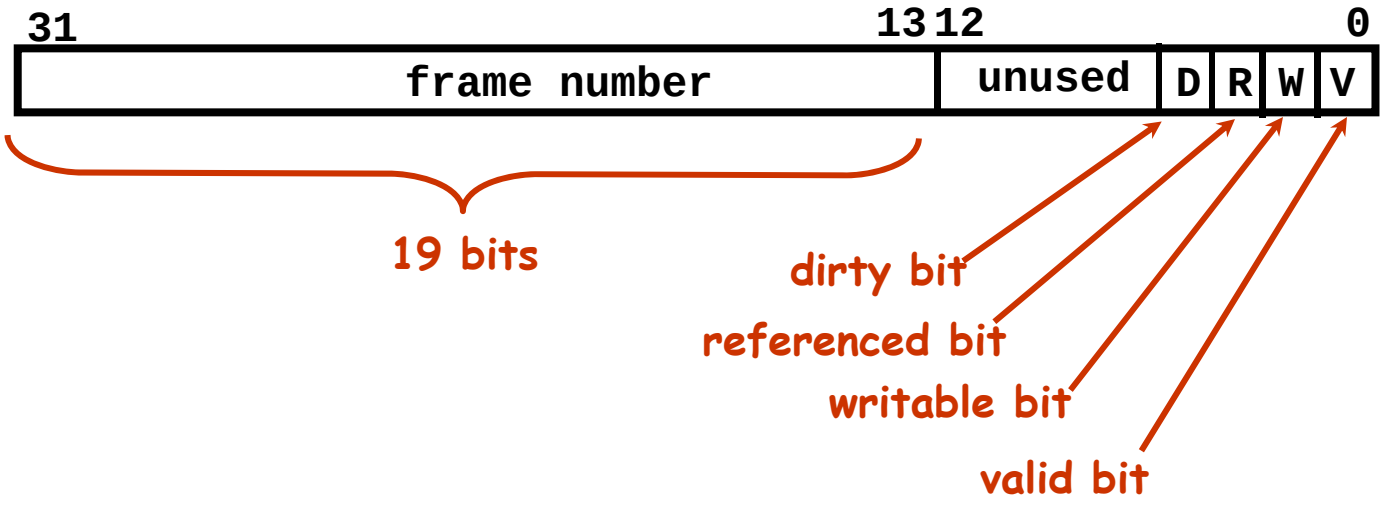
  "Paging Enabled"

  1 = Perform page table translation for every memory access

  0 = Do not do translation

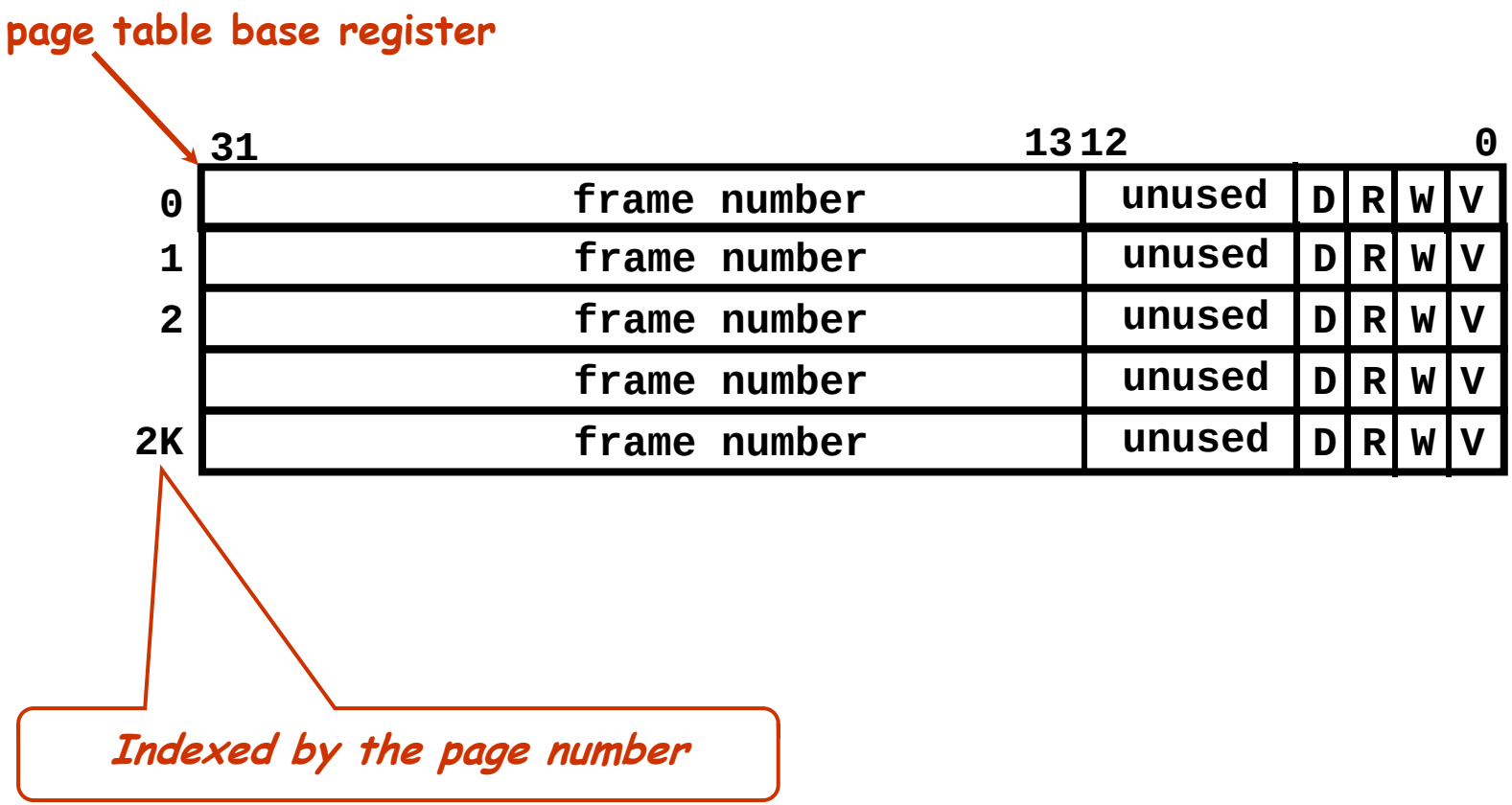
The BLITZ page table

- A page table entry

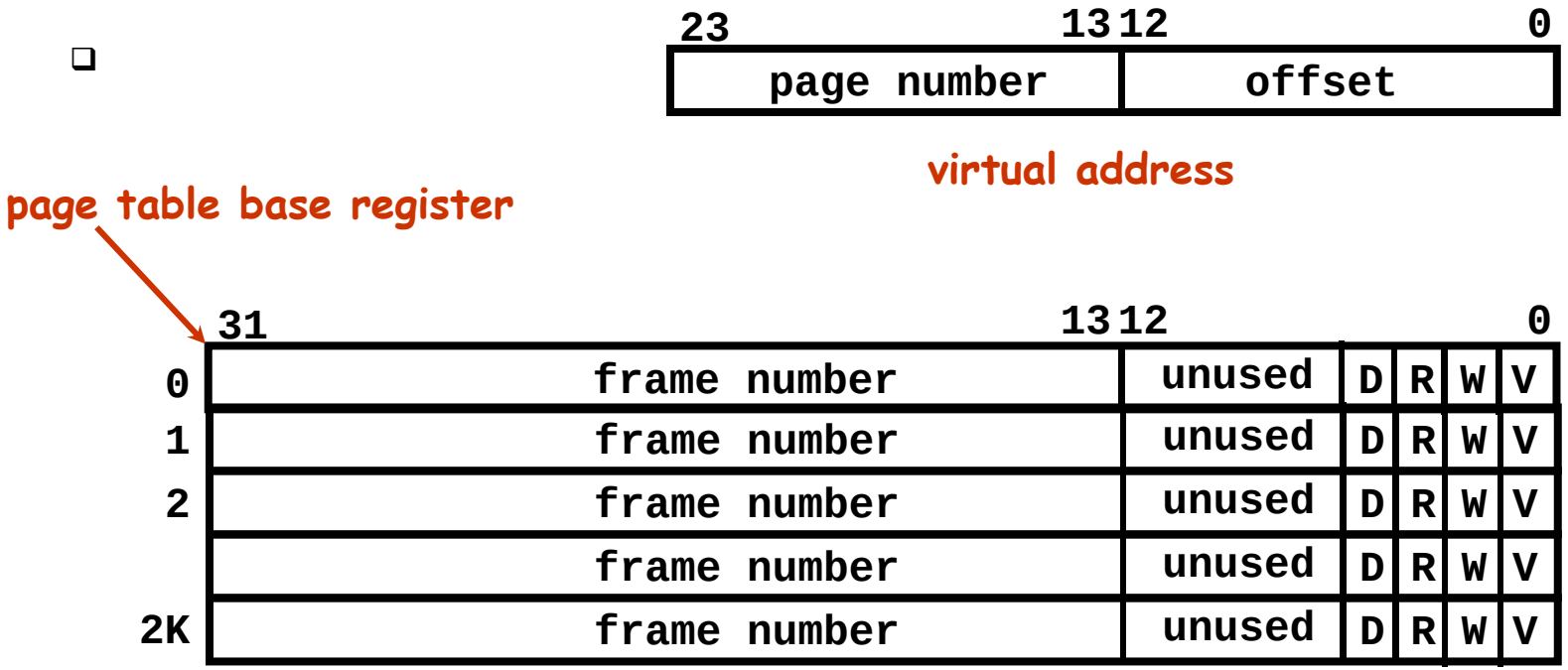


The BLITZ page table

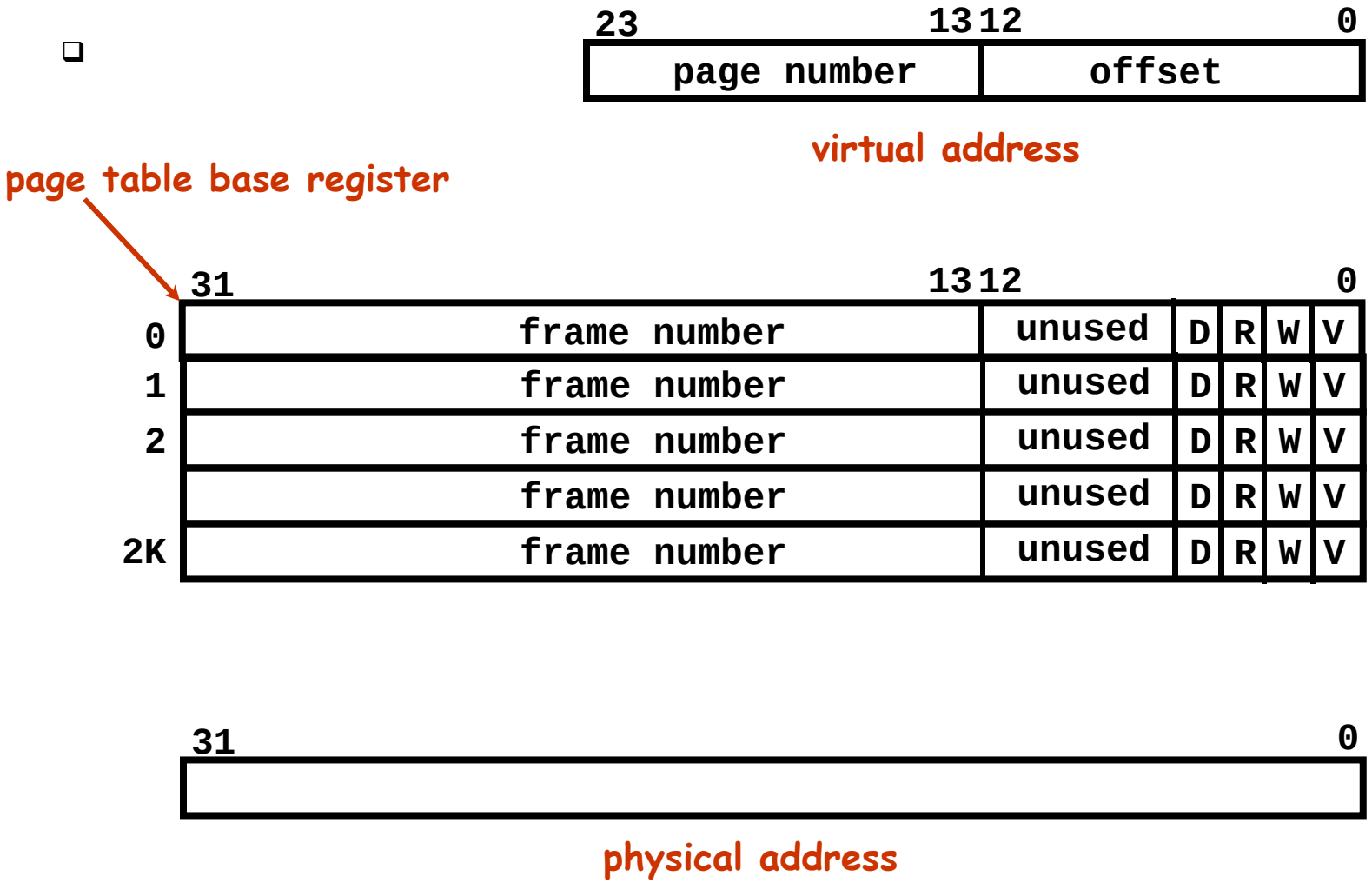
- The full page table



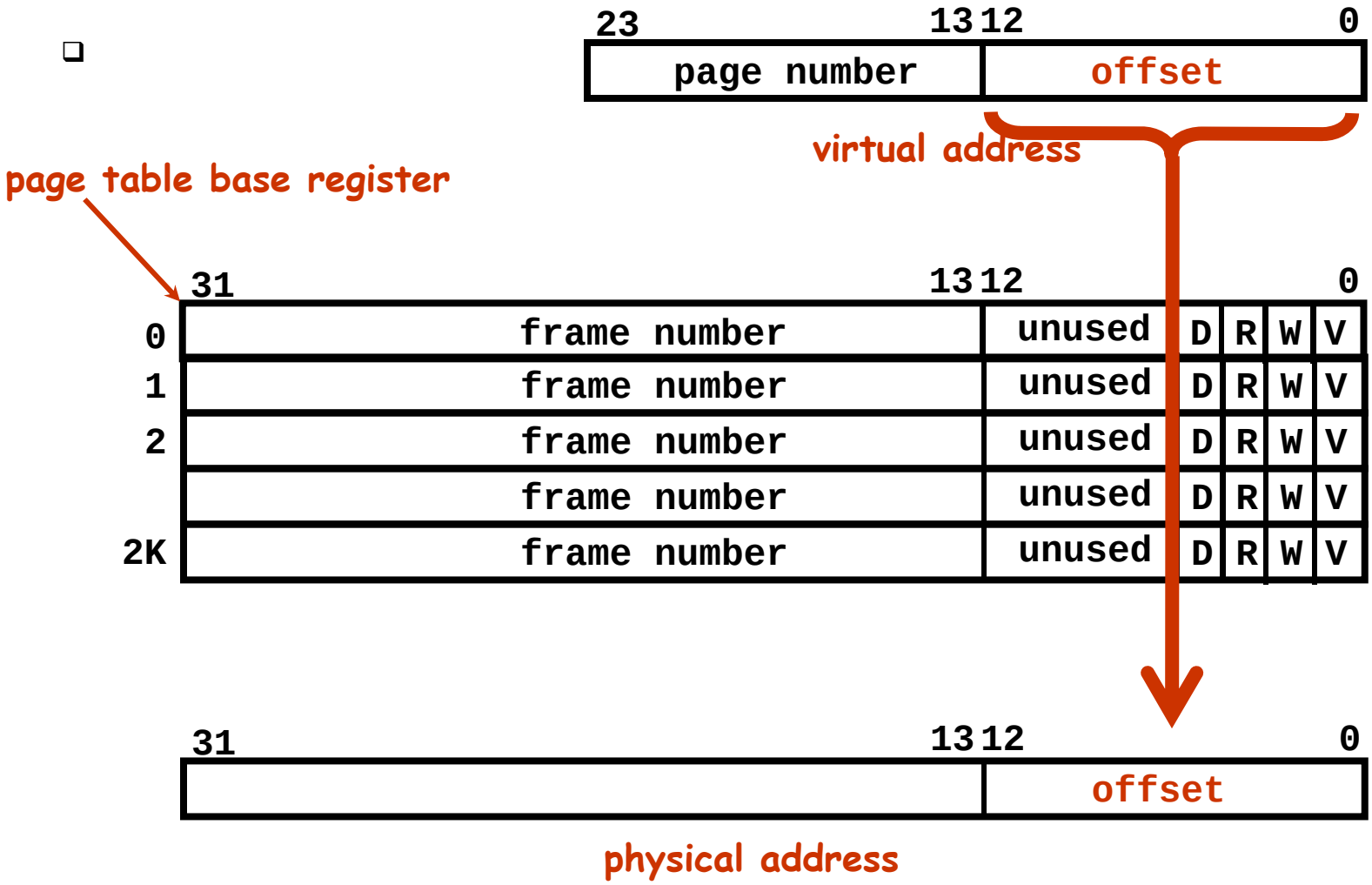
The BLITZ page table



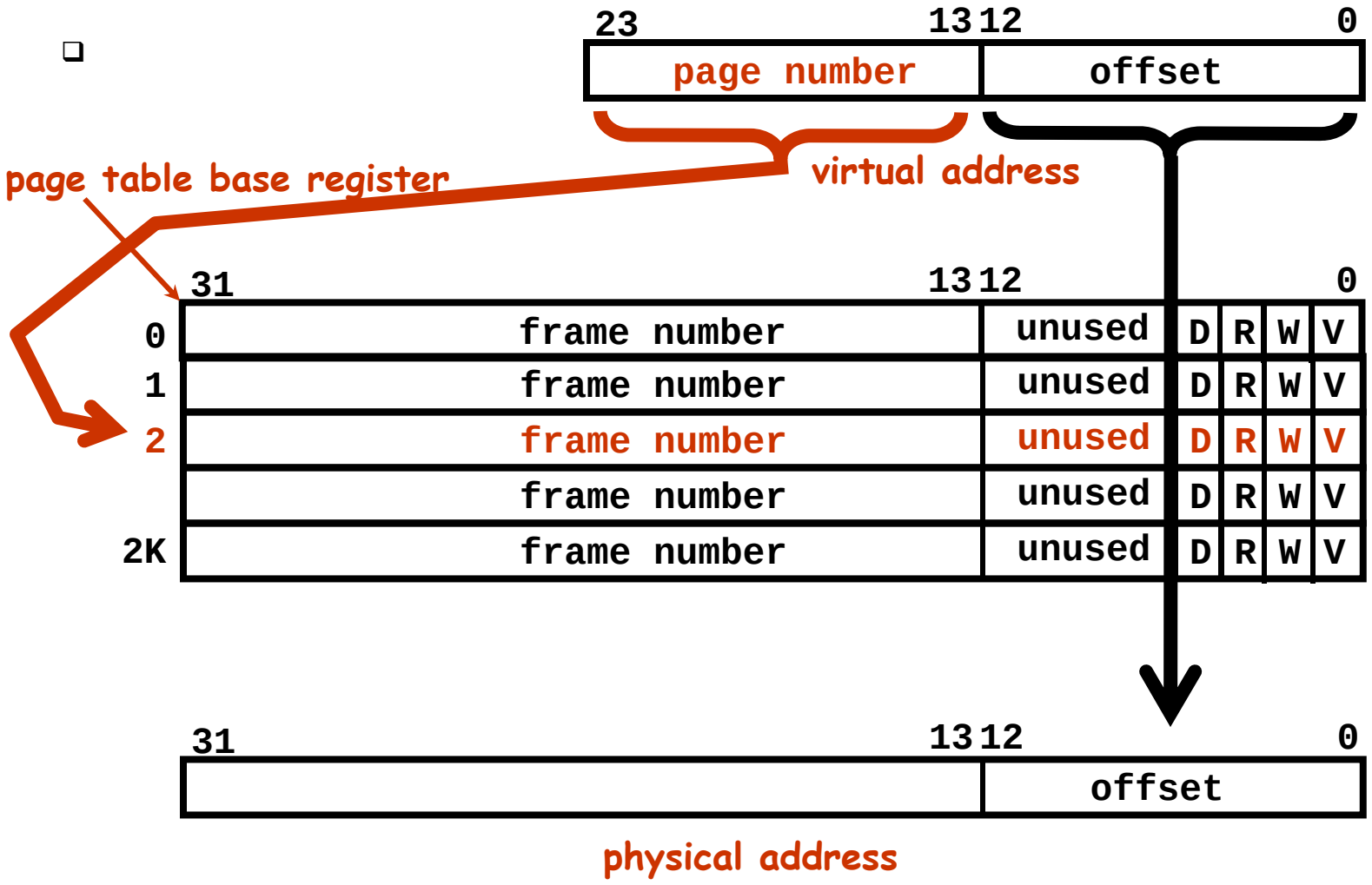
The BLITZ page table



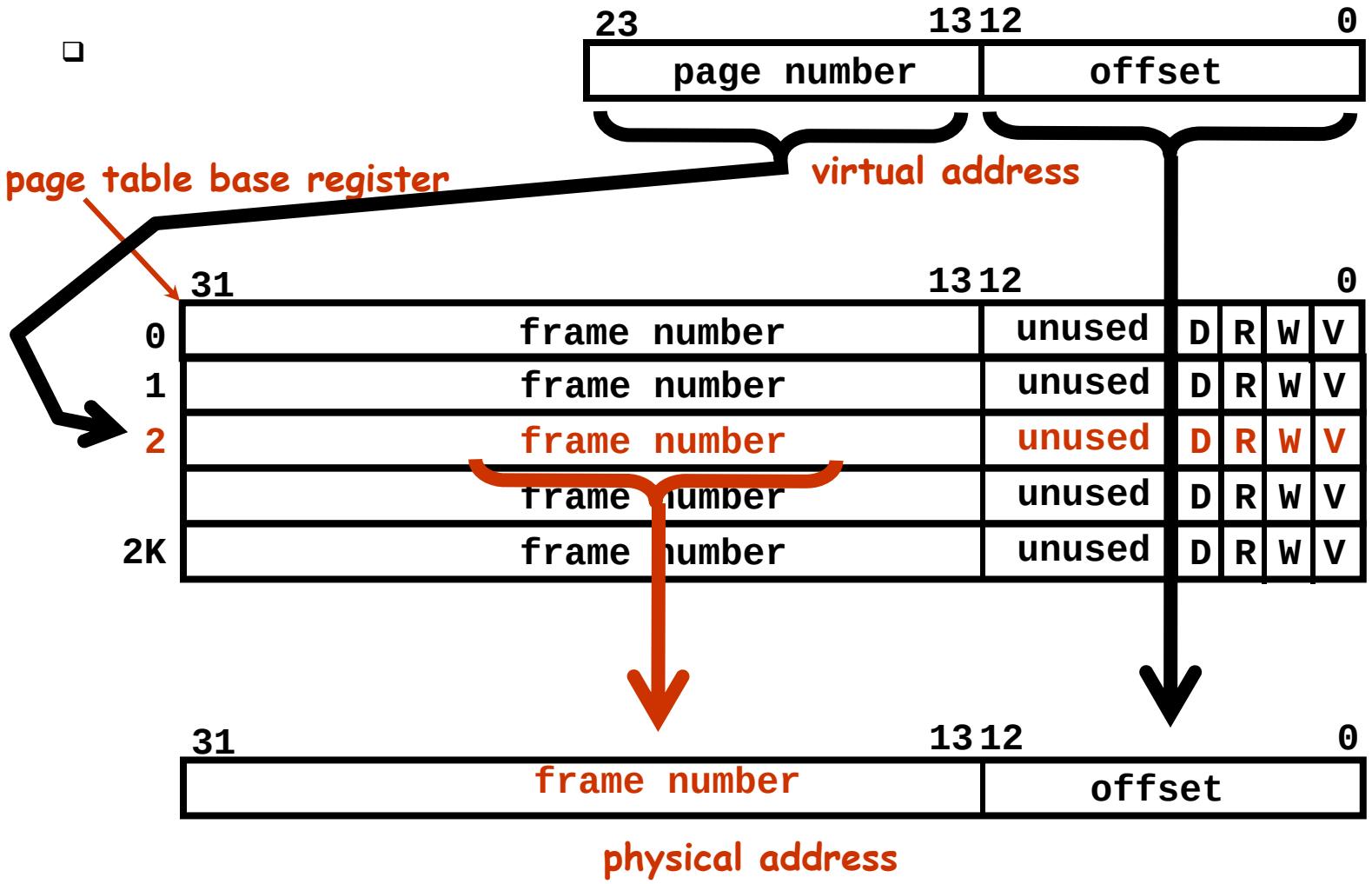
The BLITZ page table



The BLITZ page table



The BLITZ page table



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 Blitz, YES, but 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?*

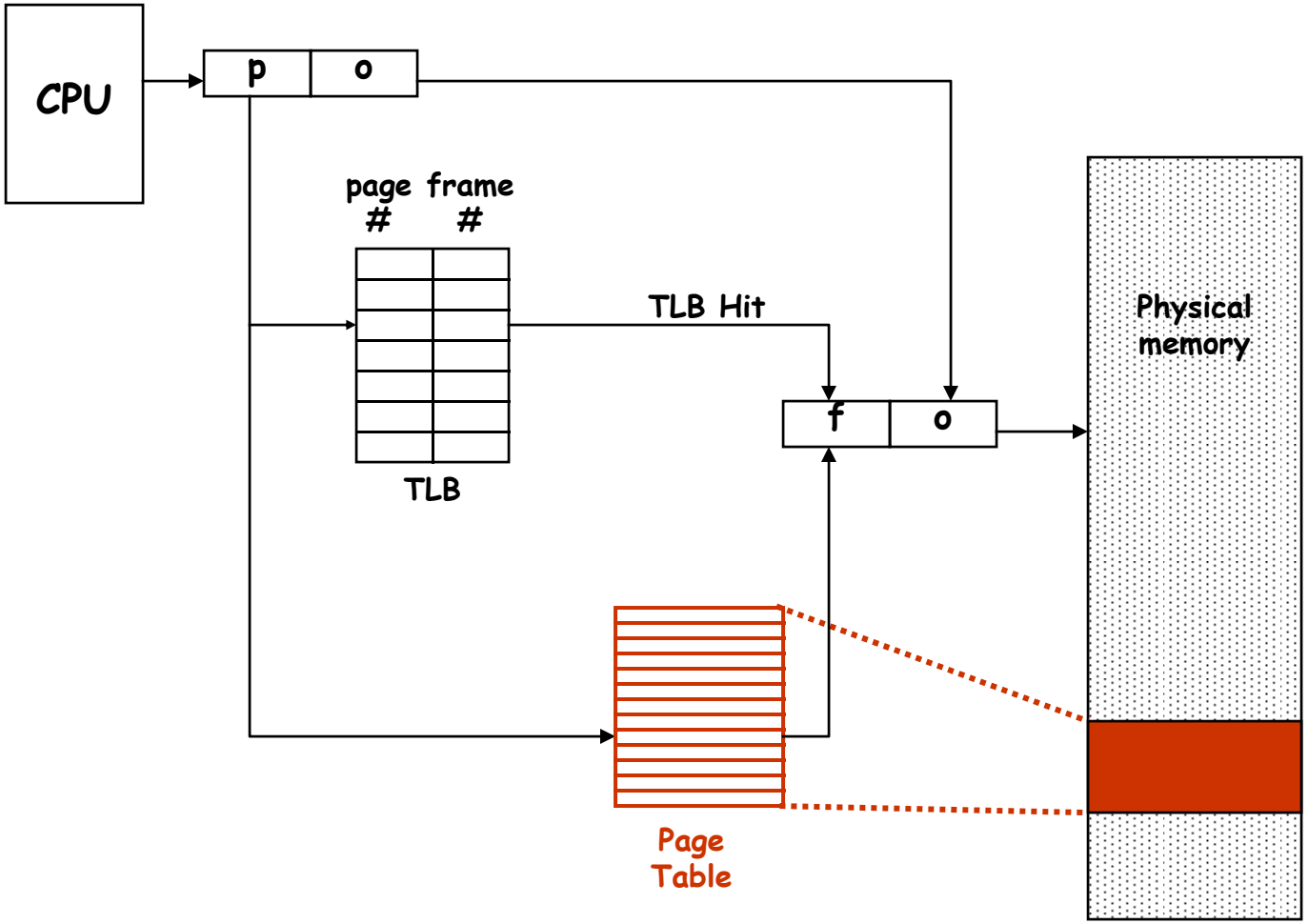
Translation Lookaside Buffer (TLB)

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

Translation Lookaside Buffer (TLB)

- **Problem:**
  - MMU can't keep up with the CPU if it goes to the page table on every memory access!
- **Solution:**
  - Cache the page table entries in a hardware cache
  - Small number of entries (e.g., 64)
  - Each entry contains
    - **Page number**
    - **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

-

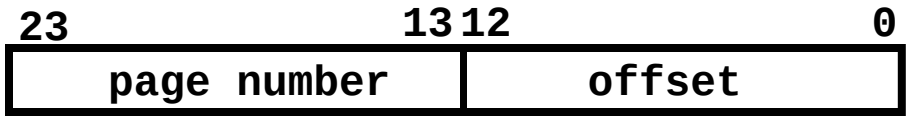
Key

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

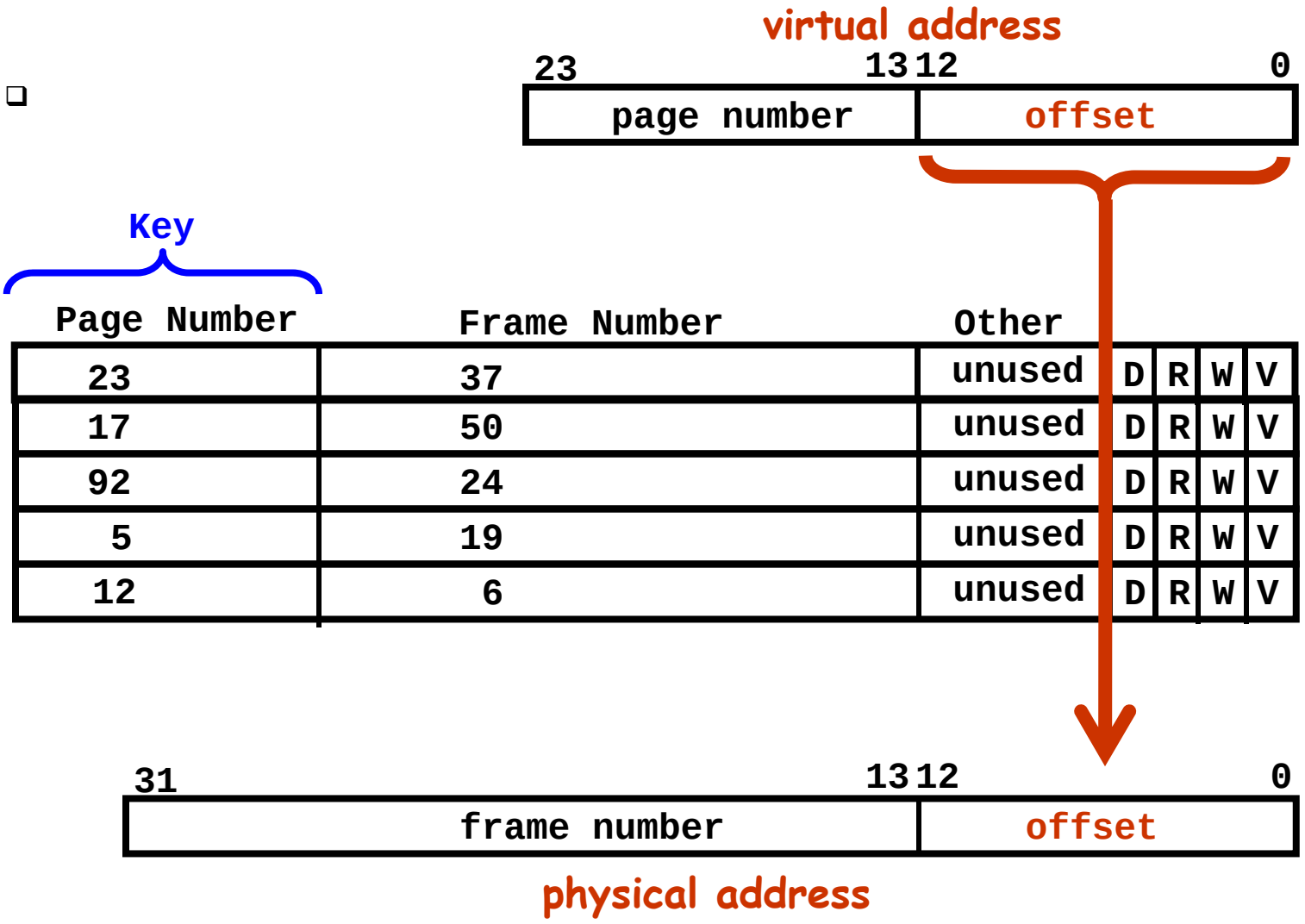


Key

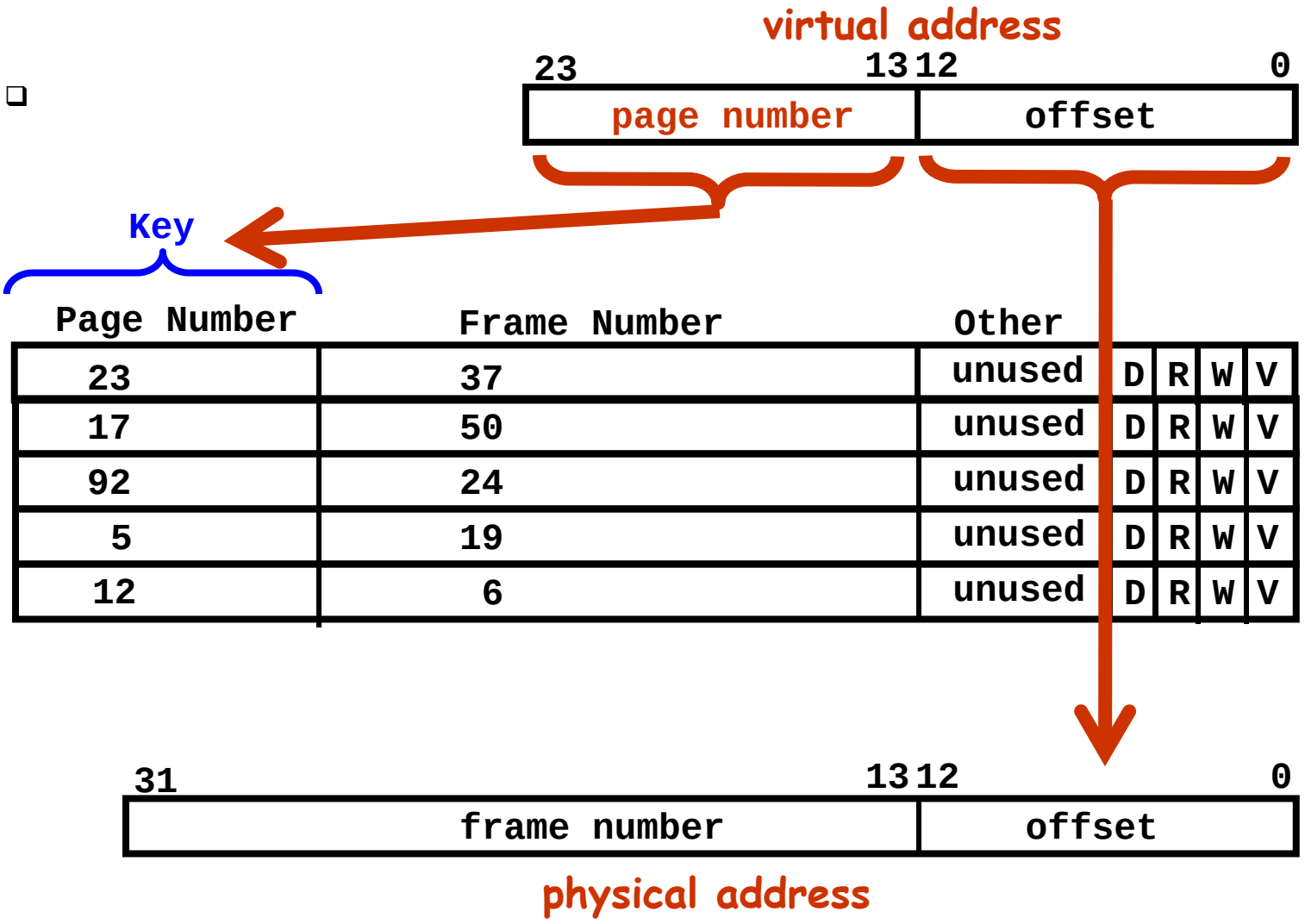
Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

physical address

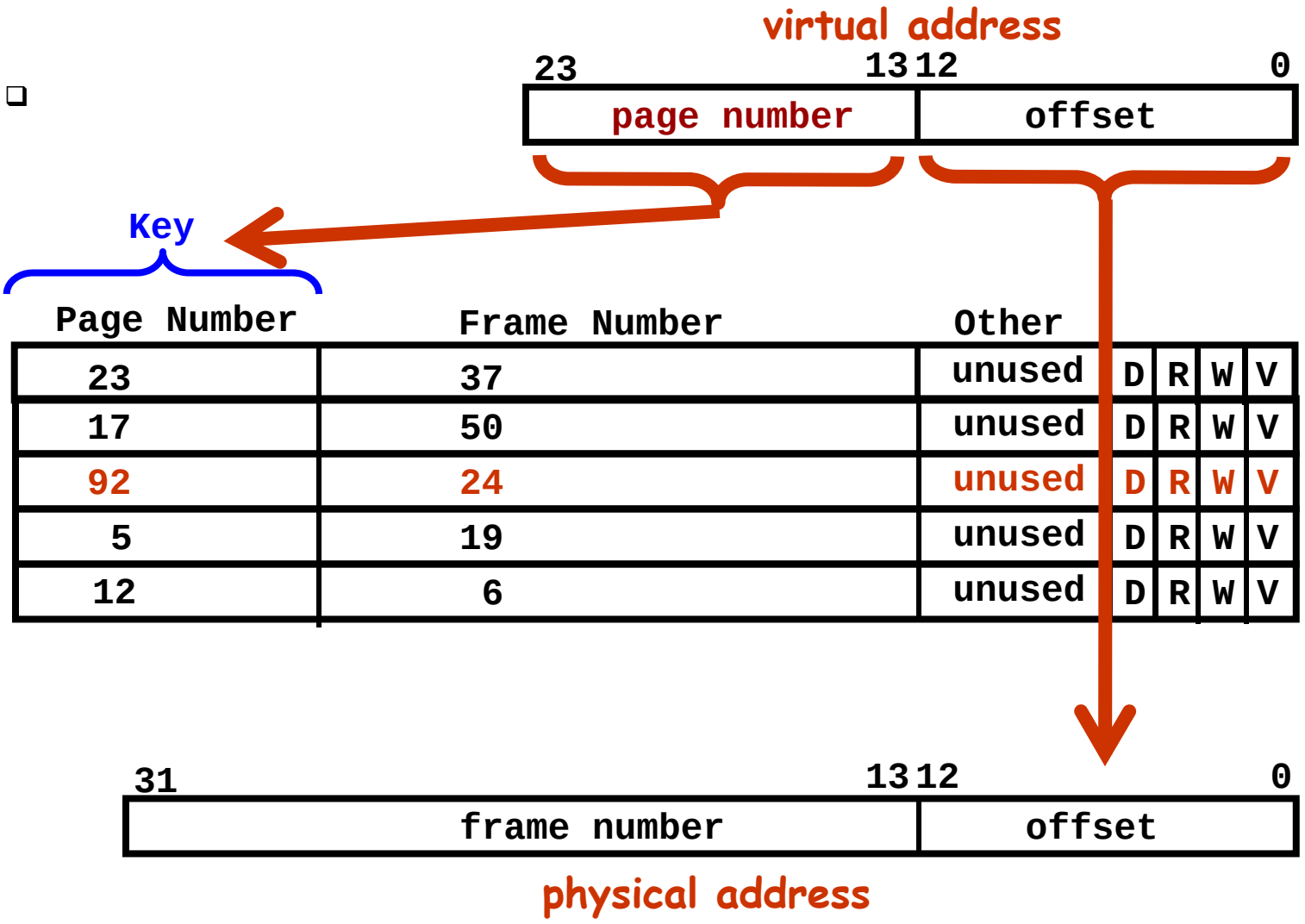
Hardware operation of TLB



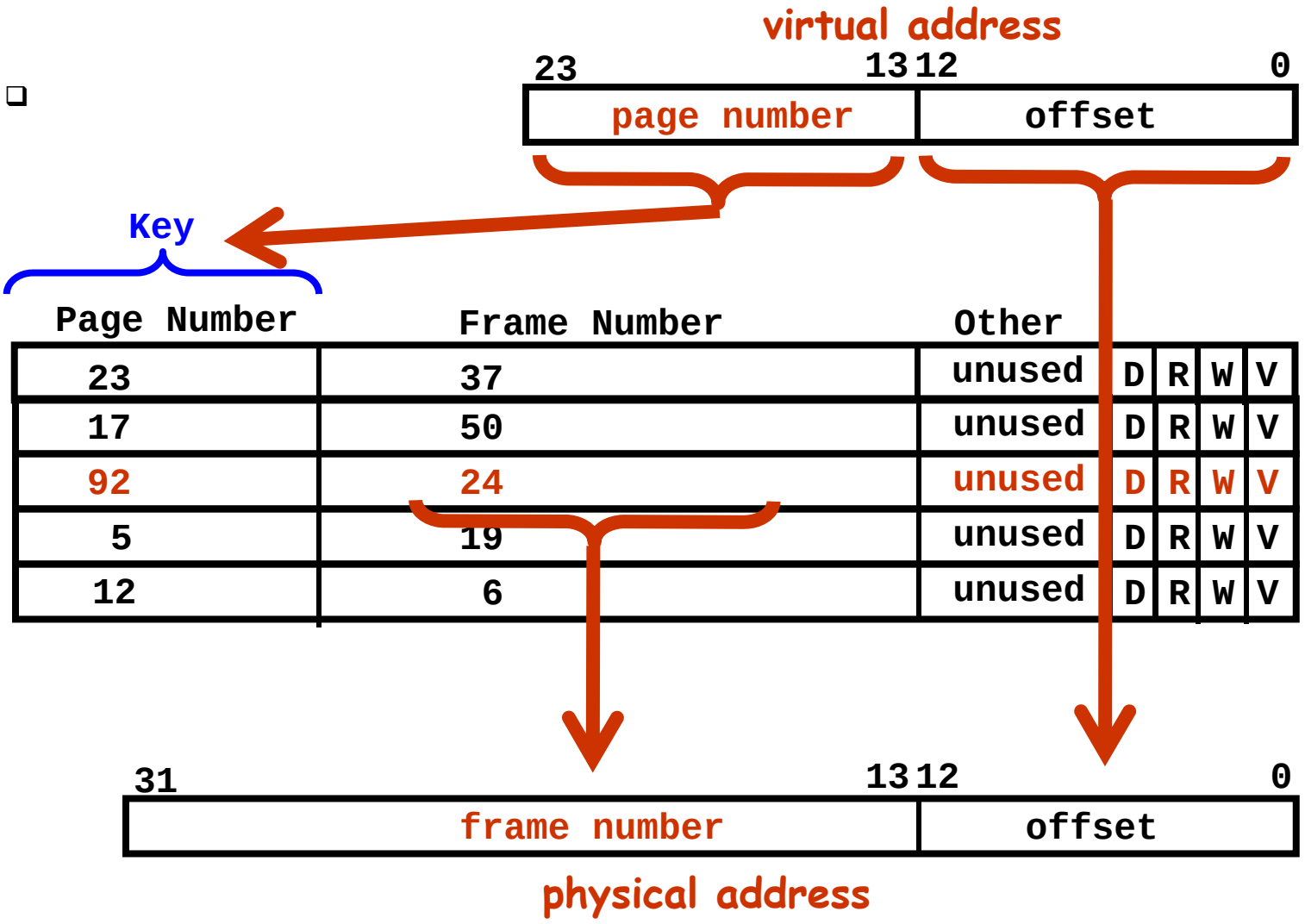
Hardware operation of TLB



Hardware operation of TLB



Hardware operation of TLB



Software operation of TLB

- **What if the entry is not in the TLB?**
  - Go look in the page table in memory
  - Find the right entry
  - Move it into the TLB
  - But which TLB entry should be replaced?

Software operation of TLB

- **Hardware TLB refill**
  - Page tables in specific location and format
  - TLB hardware handles its own misses
  - Replacement policy fixed by hardware
- **Software refill**
  - Hardware generates trap (TLB miss fault)
  - Lets the OS deal with the problem
  - Page tables become entirely a OS data structure!
  - Replacement policy managed in software

Software operation of TLB

- **What should we do with the TLB on a context switch?**
- **How can we prevent the next process from using the last process's address mappings?**
  - Option 1: empty the TLB
    - **New process will generate faults until its pulls enough of its own entries into the TLB**
  - Option 2: just clear the "Valid Bit"
    - **New process will generate faults until its pulls enough of its own entries into the TLB**
  - Option 3: the hardware maintains a process id tag on each TLB entry
    - **Hardware compares this to a process id held in a specific register … on every translation**

Page tables

- Do we access a page table when a process allocates or frees memory?

Page tables

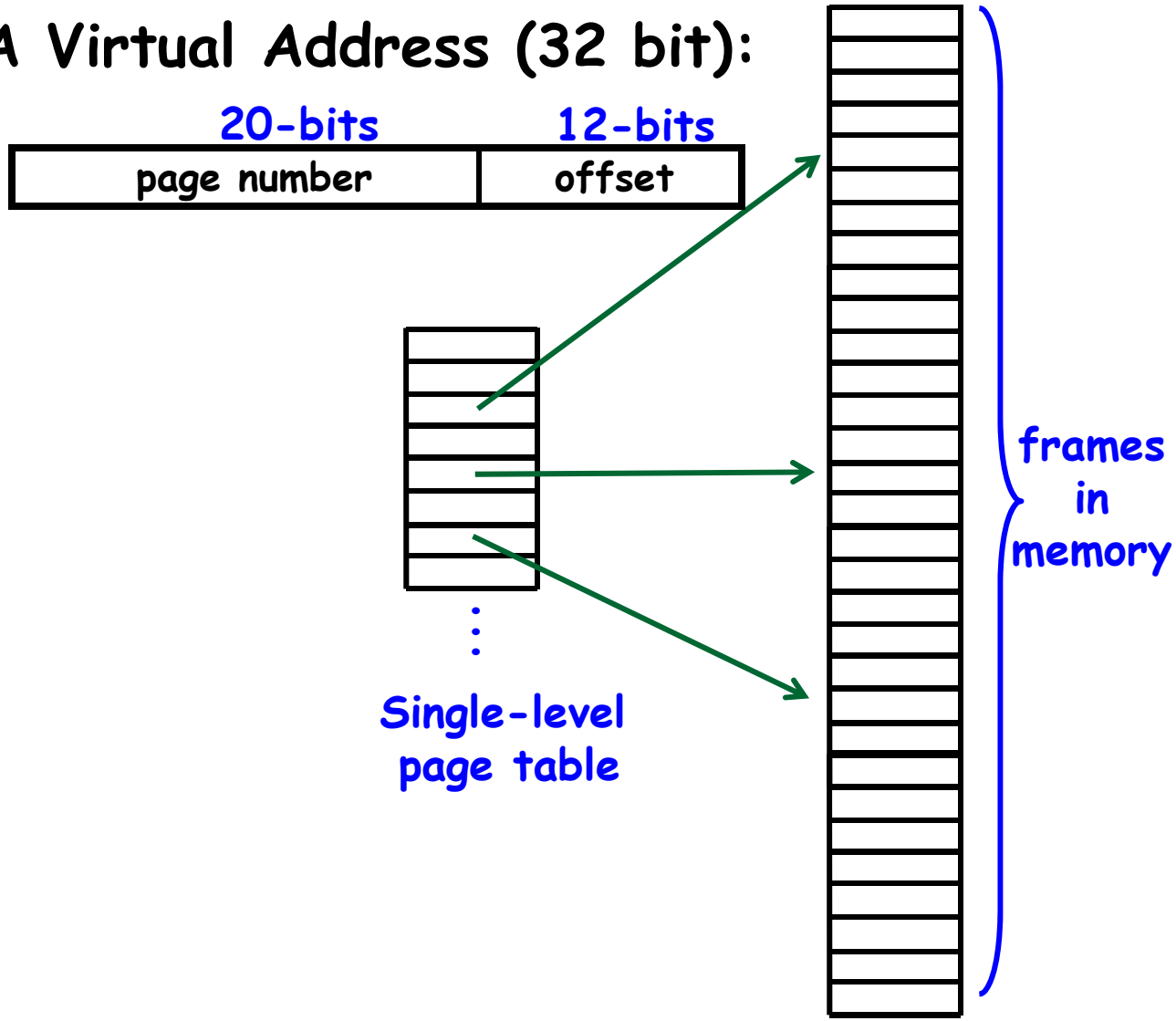
- **Do we access a page table when a process allocates or frees memory?**
  - Not necessarily
  - Library routines (malloc) can service small requests from a pool of free memory already allocated within a process address space
  - When these routines run out of space a new page must be allocated and its entry inserted into the page table
    - **This allocation is requested using a system call**

Page table design issues

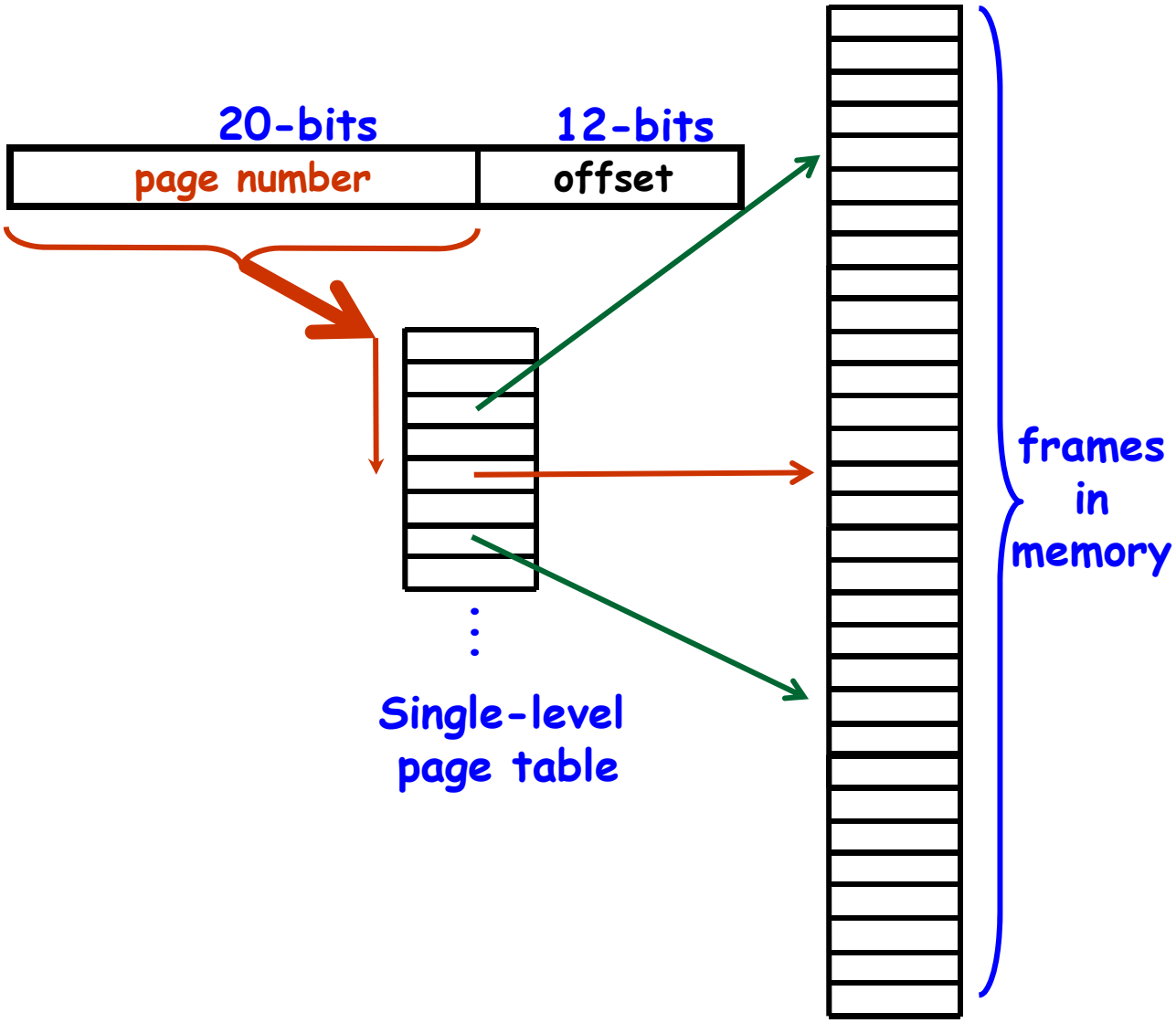
- **Page table size depends on**
  - Page size
  - Virtual address length
- **Memory used for page tables is overhead!**
  - How can we save space?
  - … and still find entries quickly?
- **Three options**
  - Single-level page tables
  - Multi-level page tables
  - Inverted page tables

Single-level page tables

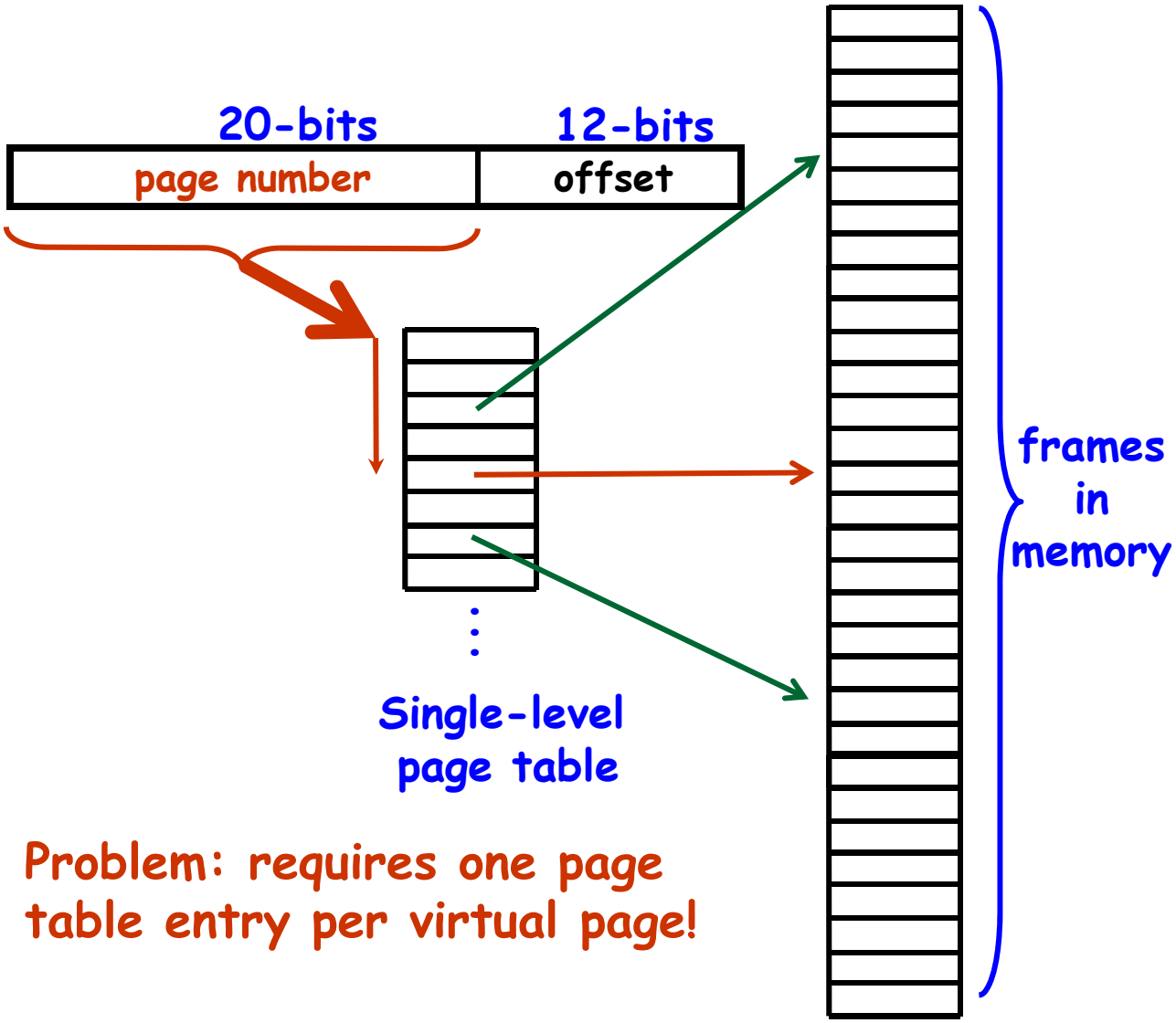
A Virtual Address (32 bit):



Single-level page tables

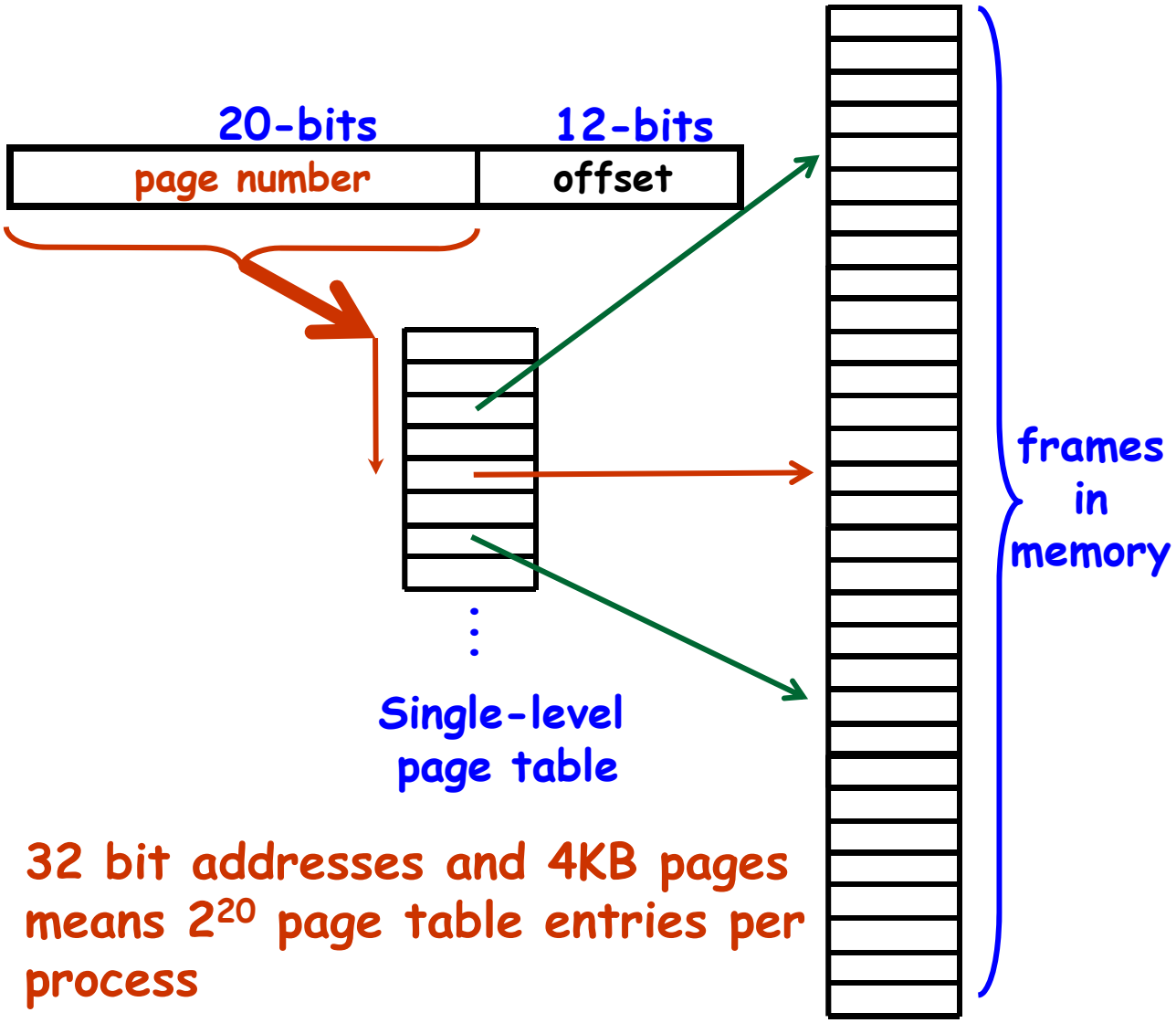


Single-level page tables



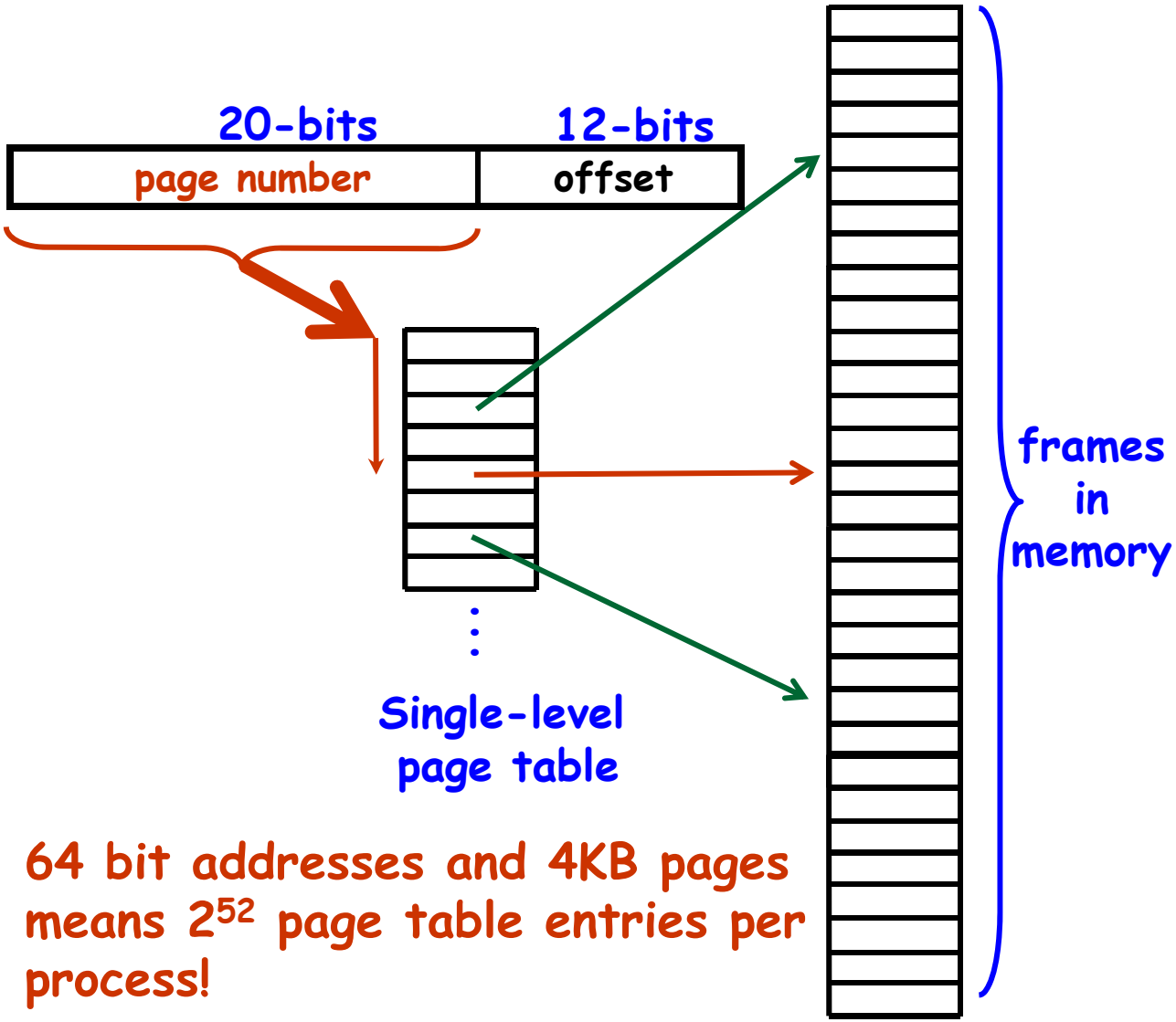
Problem: requires one page table entry per virtual page!

Single-level page tables



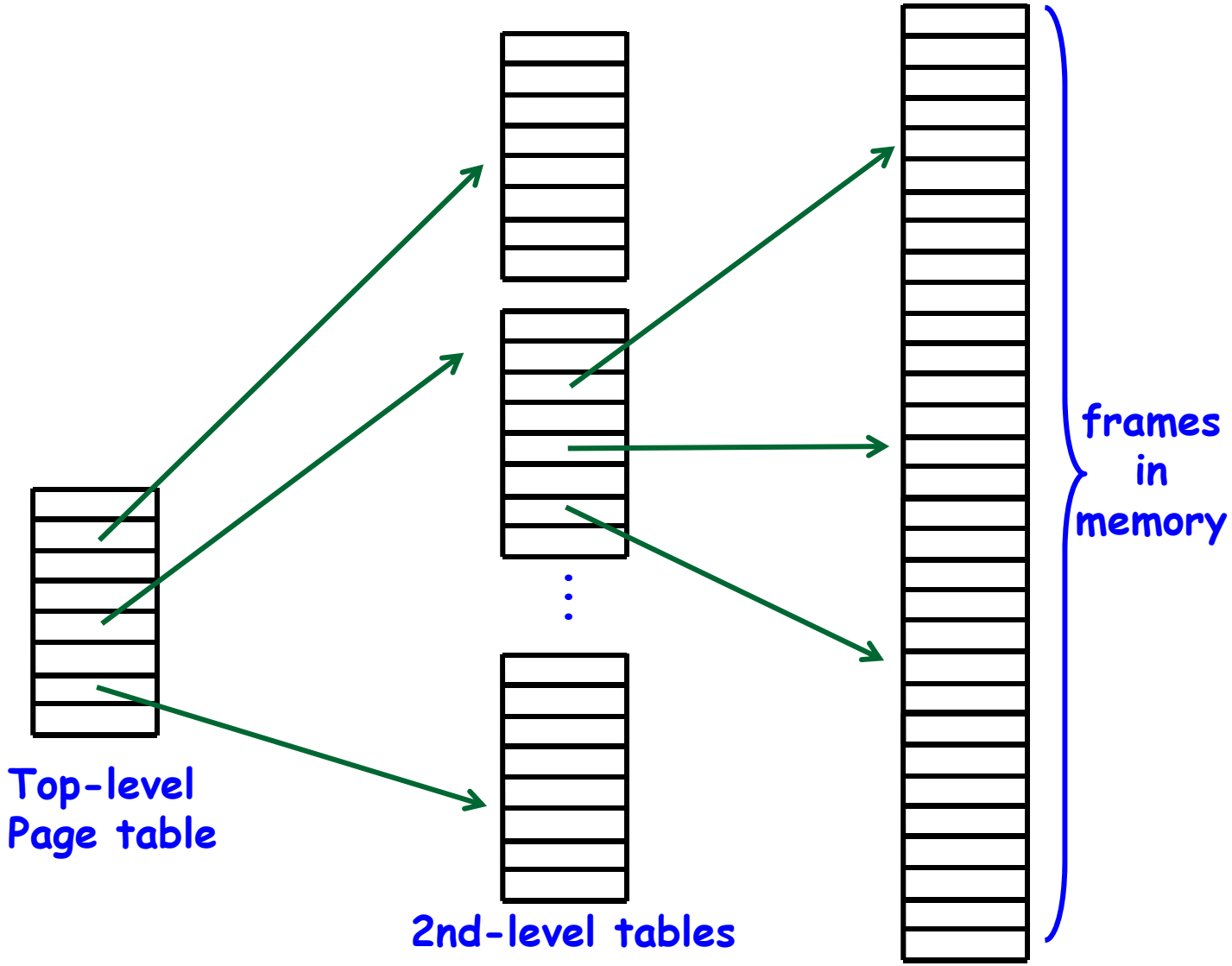
32 bit addresses and 4KB pages means 2^{20} page table entries per process

Single-level page tables



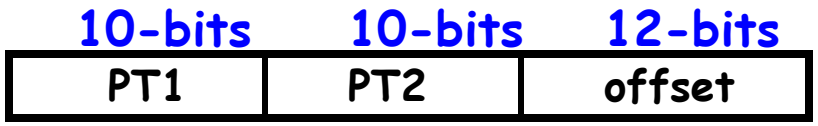
64 bit addresses and 4KB pages means 2^{52} page table entries per process!

Multi-level page tables



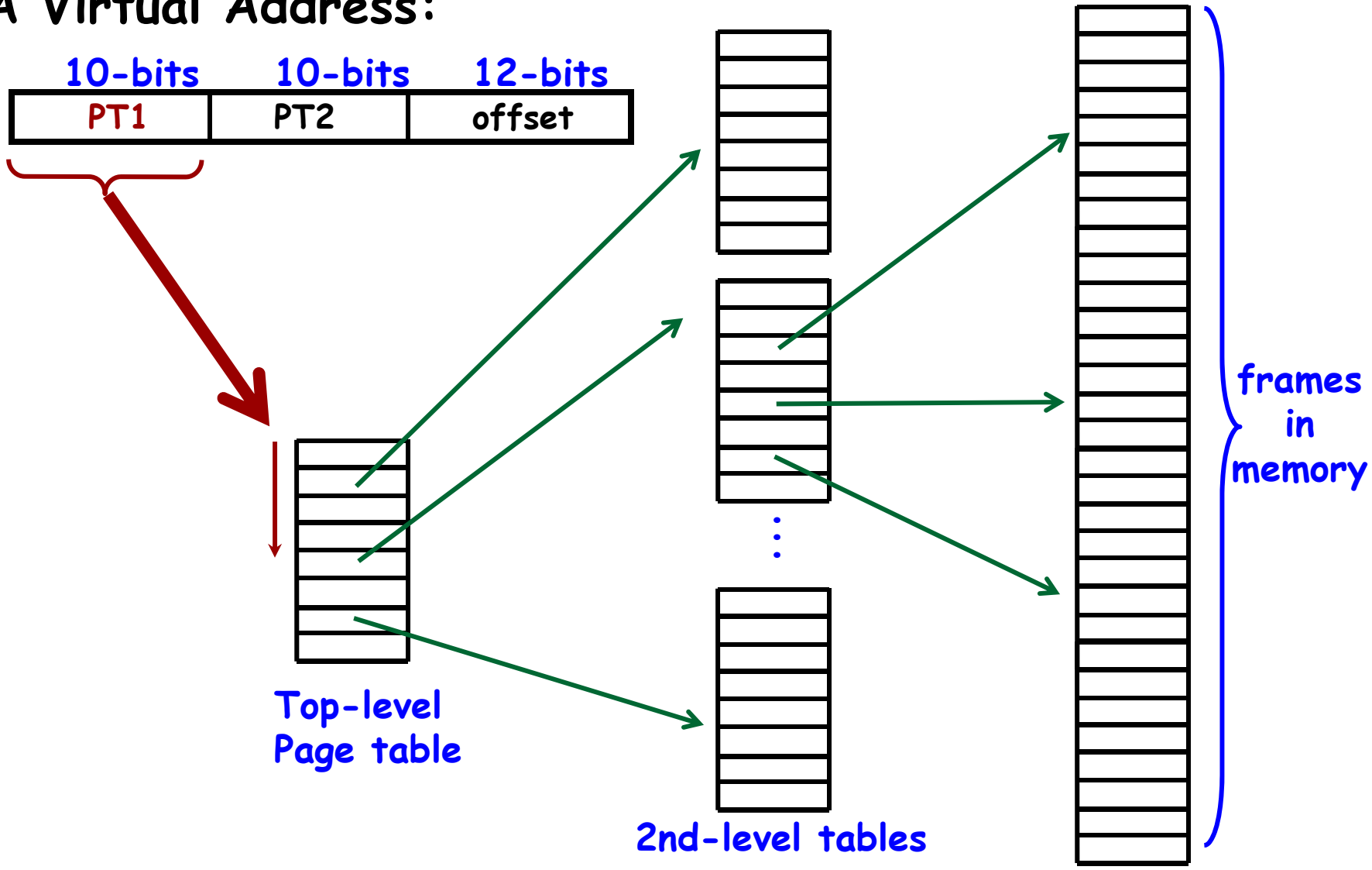
Multi-level page tables

- A Virtual Address:



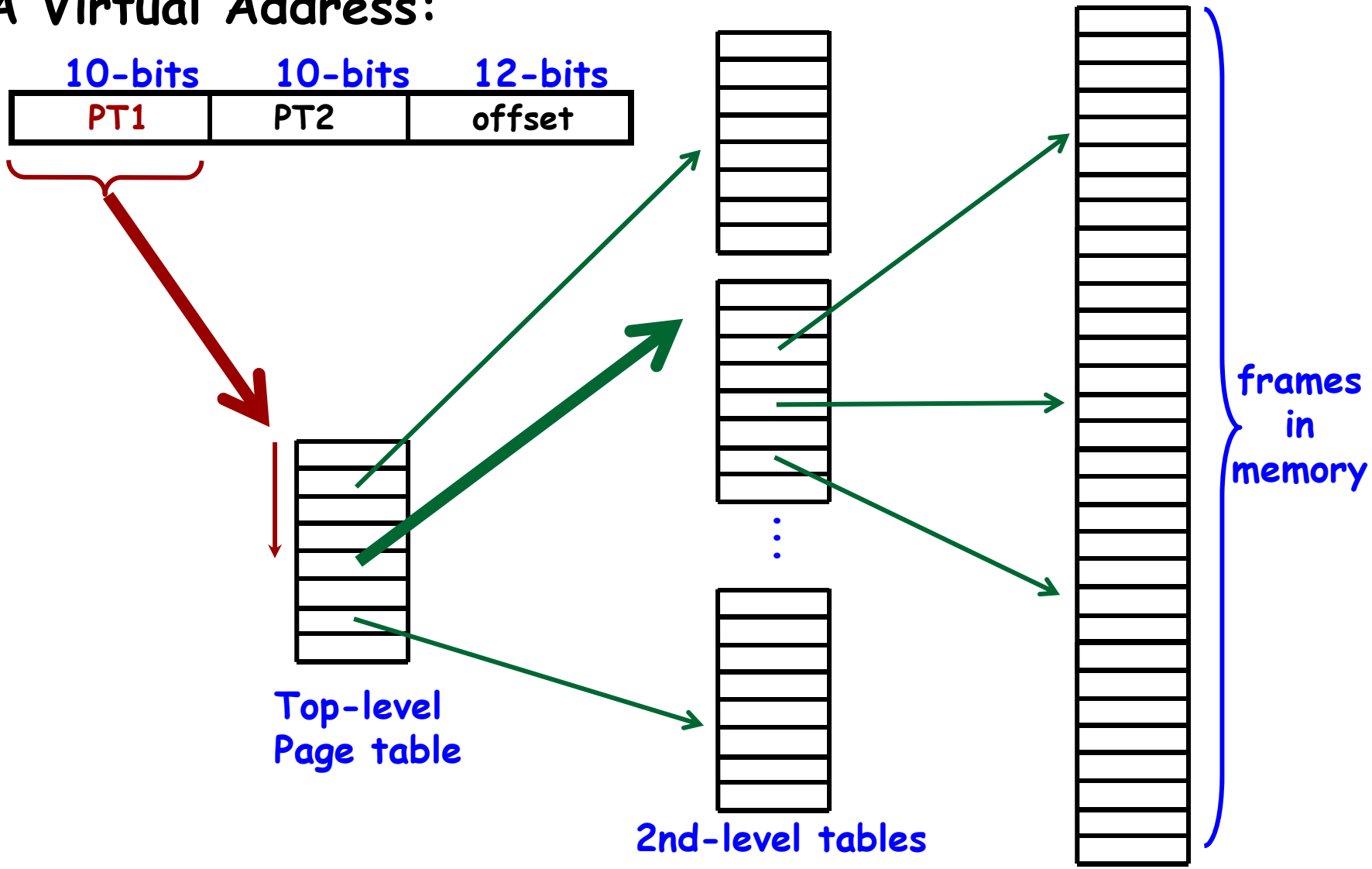
Multi-level page tables

- A Virtual Address:



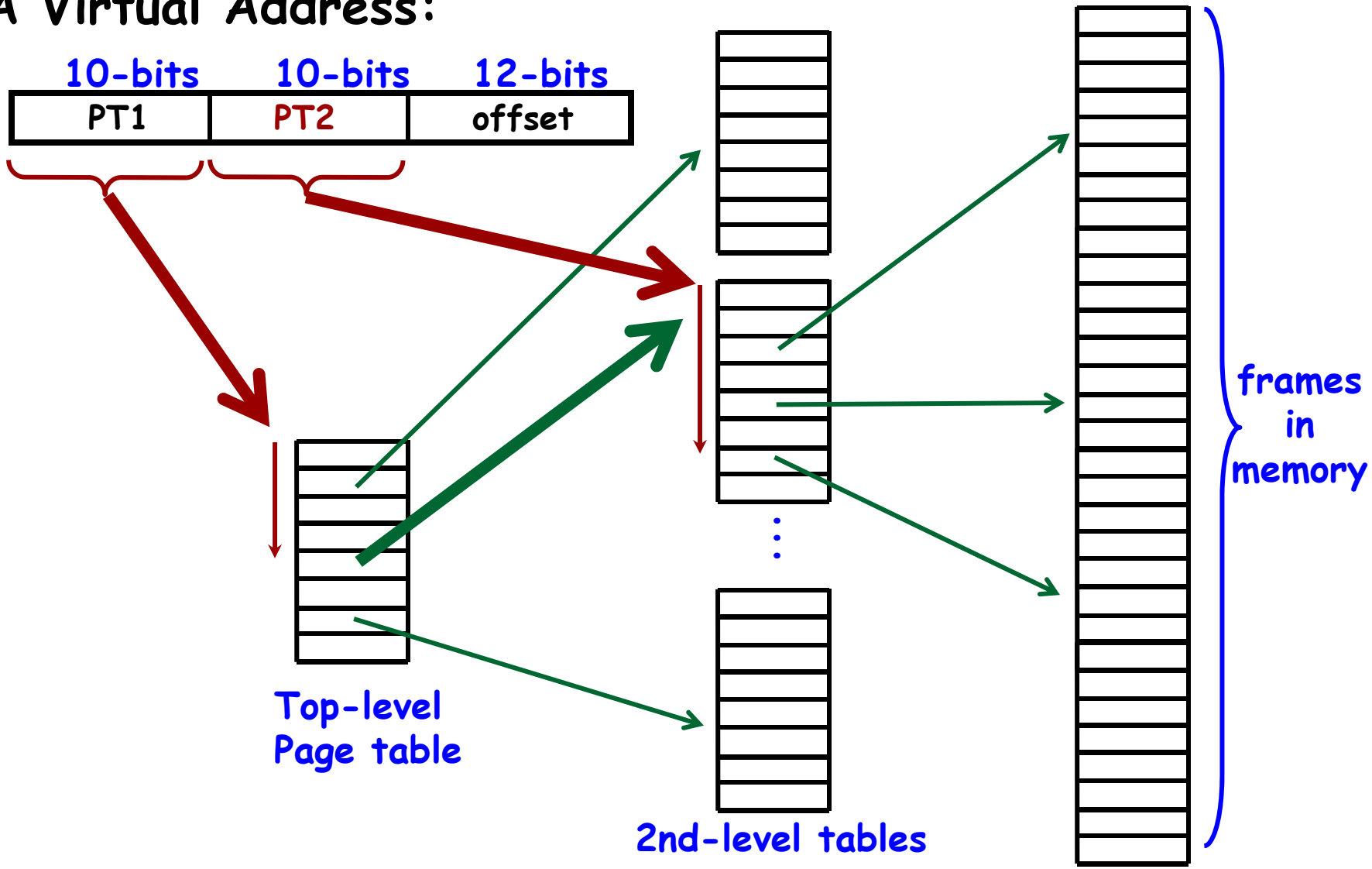
Multi-level page tables

- A Virtual Address:



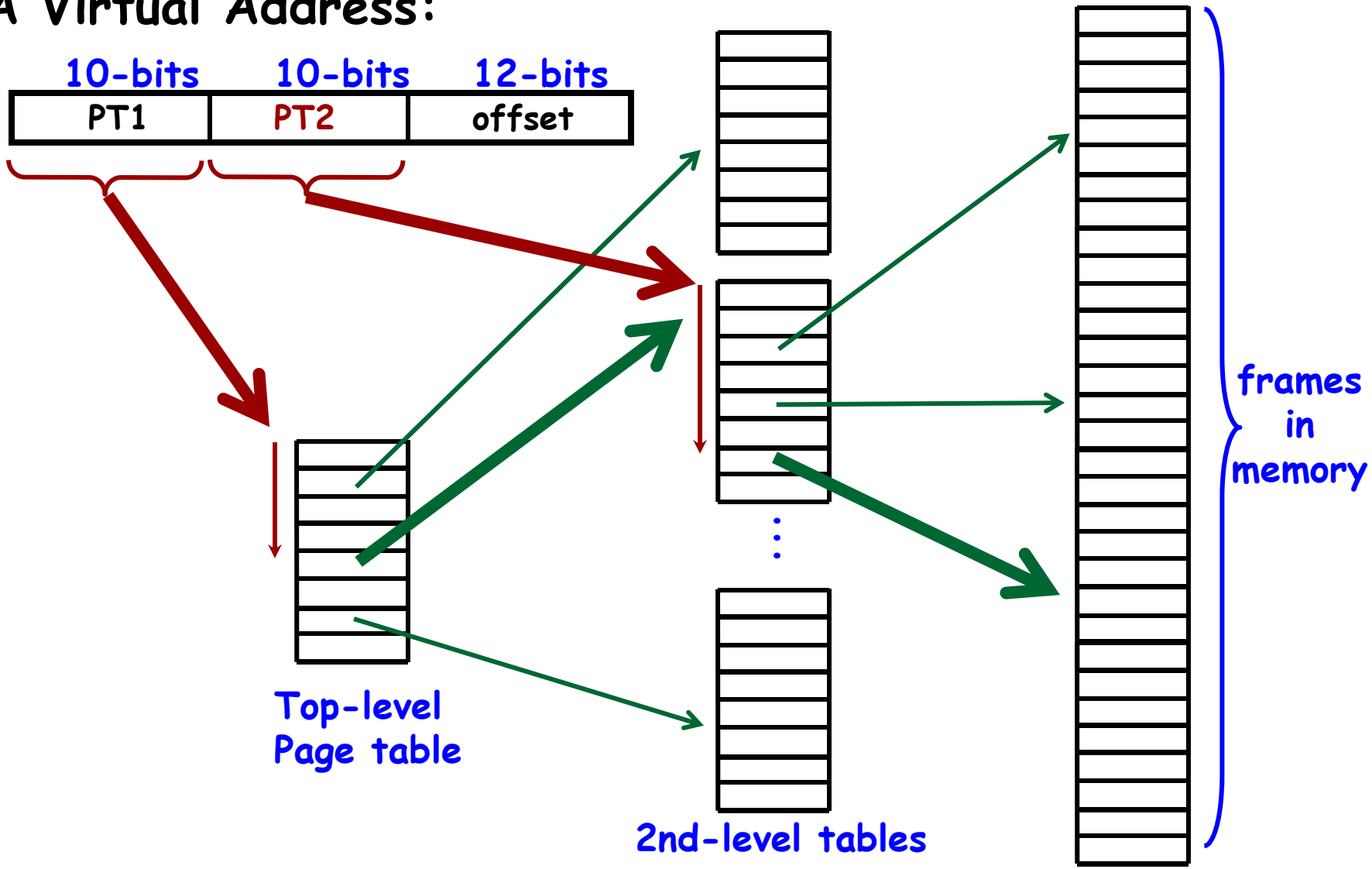
Multi-level page tables

- A Virtual Address:



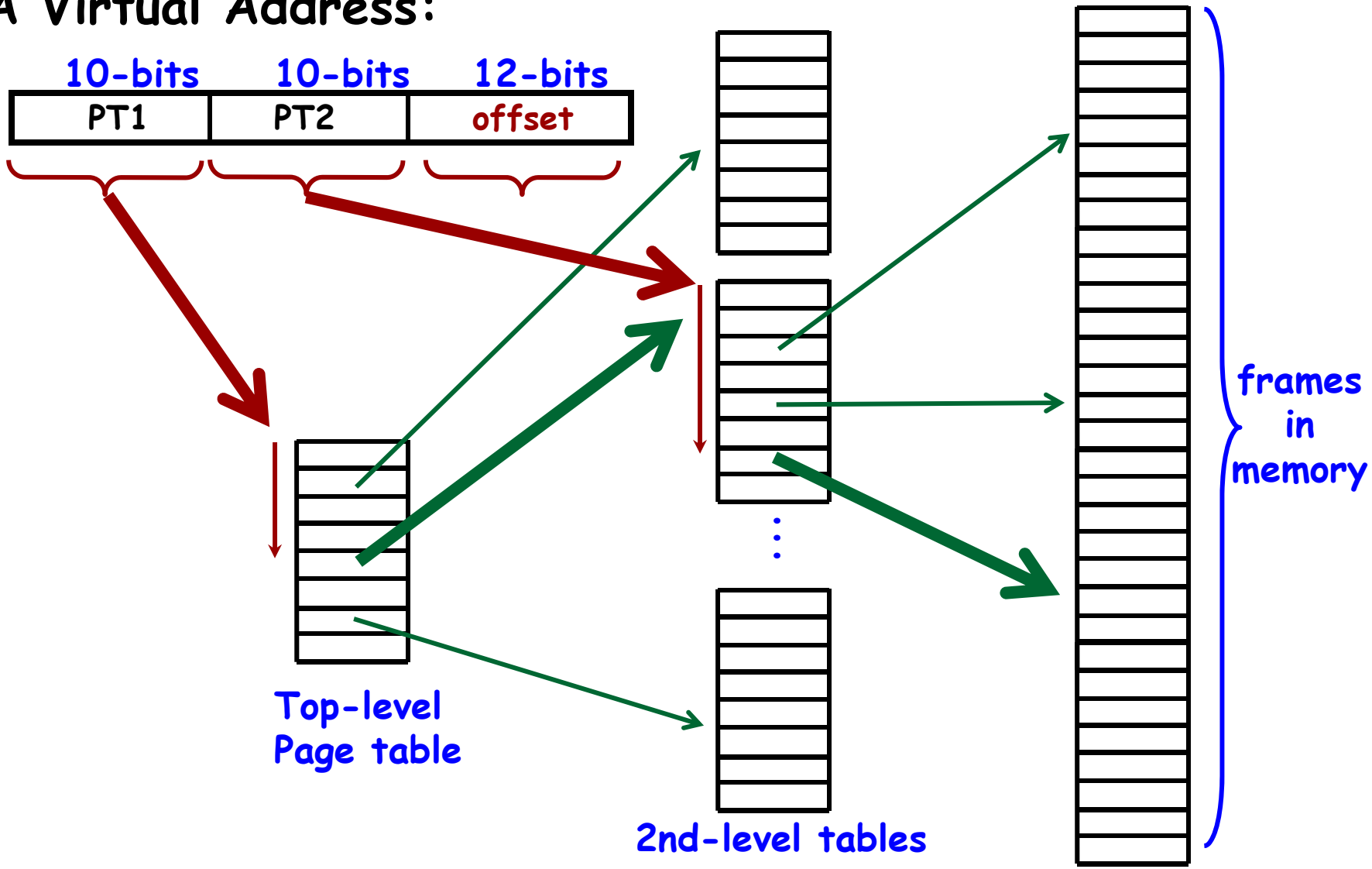
Multi-level page tables

- A Virtual Address:



Multi-level page tables

- A Virtual Address:



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!

VM puzzle