

R

بسم الله الرحمن الرحيم

سیستم عامل

جلسه سیزدهم – مدیریت حافظه (جدول صفحه)

جلسه‌ی گذشته

مدیریت حافظه - شروع

Memory Management

- Memory – a linear array of bytes
 - *Holds O.S. and programs (processes)*
 - *Each cell (byte) is named by a unique memory address*
- Recall, processes are defined by an *address space*, consisting of text, data, and stack regions
- Process execution
 - *CPU fetches instructions from the text region according to the value of the program counter (PC)*
 - *Each instruction may request additional operands from the data or stack region*

Address Binding

Address binding of instructions and data to memory addresses can happen at three different stages:

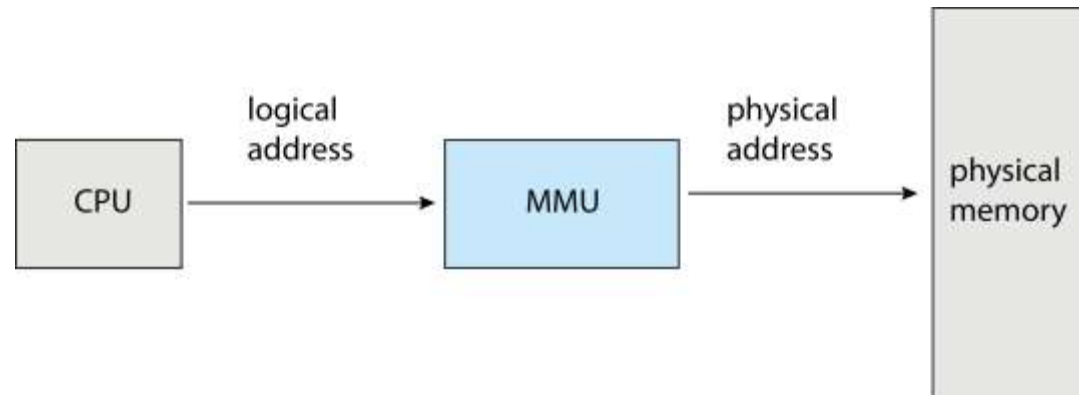
- **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time:** Must generate **relocatable code** if memory location is not known at compile time
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - Need hardware support for address maps (e.g., base and limit registers)

Base and Limit Registers

- Simple runtime relocation scheme
 - *Use 2 registers to describe a partition*
- For every address generated, at runtime...
 - *Compare to the **limit** register (& abort if larger)*
 - *Add to the **base** register to give **physical** memory address*

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this lecture

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Solutions to Fragmentation

- Compaction requires high copying overhead
- Why not allocate memory in non-contiguous equal fixed size units?
 - *No external fragmentation!*
 - *Internal fragmentation < 1 unit per process*
- How big should the units be?
 - *The smaller the better for internal fragmentation*
 - *The larger the better for management overhead*
- The key challenge for this approach

How can we do secure dynamic address translation?

جلسه‌ی جدید

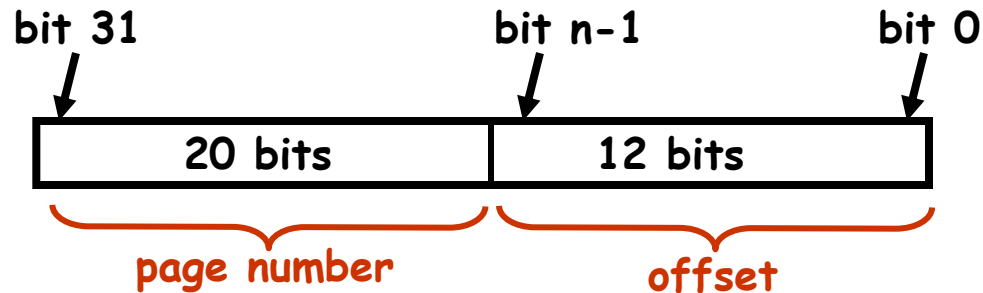
مدیریت حافظه - جدول صفحه

Non-Contiguous Allocation (Pages)

- Memory divided into fixed size **page frames**
 - *Page frame size = 2^n bytes*
 - *Lowest n bits of an address specify byte offset in a page*
- But how do we associate page frames with processes?
 - *And how do we map memory addresses within a process to the correct memory byte in a page frame?*
- Solution – address translation
 - *Processes use **virtual addresses***
 - *CPU uses **physical addresses***
 - *Hardware support for virtual to physical **address translation***

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*



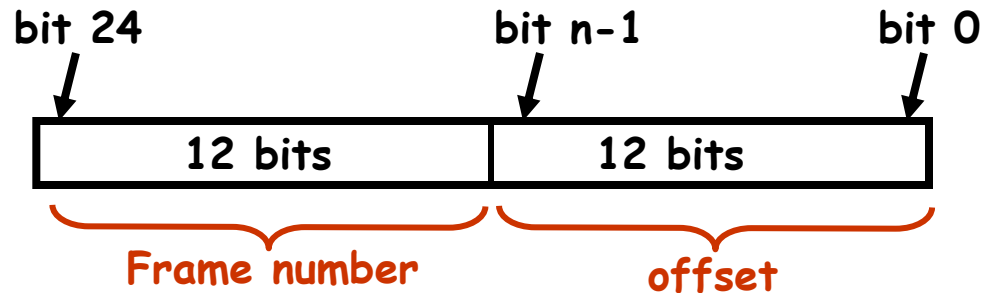
Example: 32 bit virtual address

Page size = 2^{12} = 4KB

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

Physical Addresses

- Physical memory addresses (what the CPU uses)
 - *Page “frame” number plus byte offset in page*
 - *Low order n bits are the byte offset*
 - *Remaining high order bits are the *frame* number*



Example: 24 bit physical address

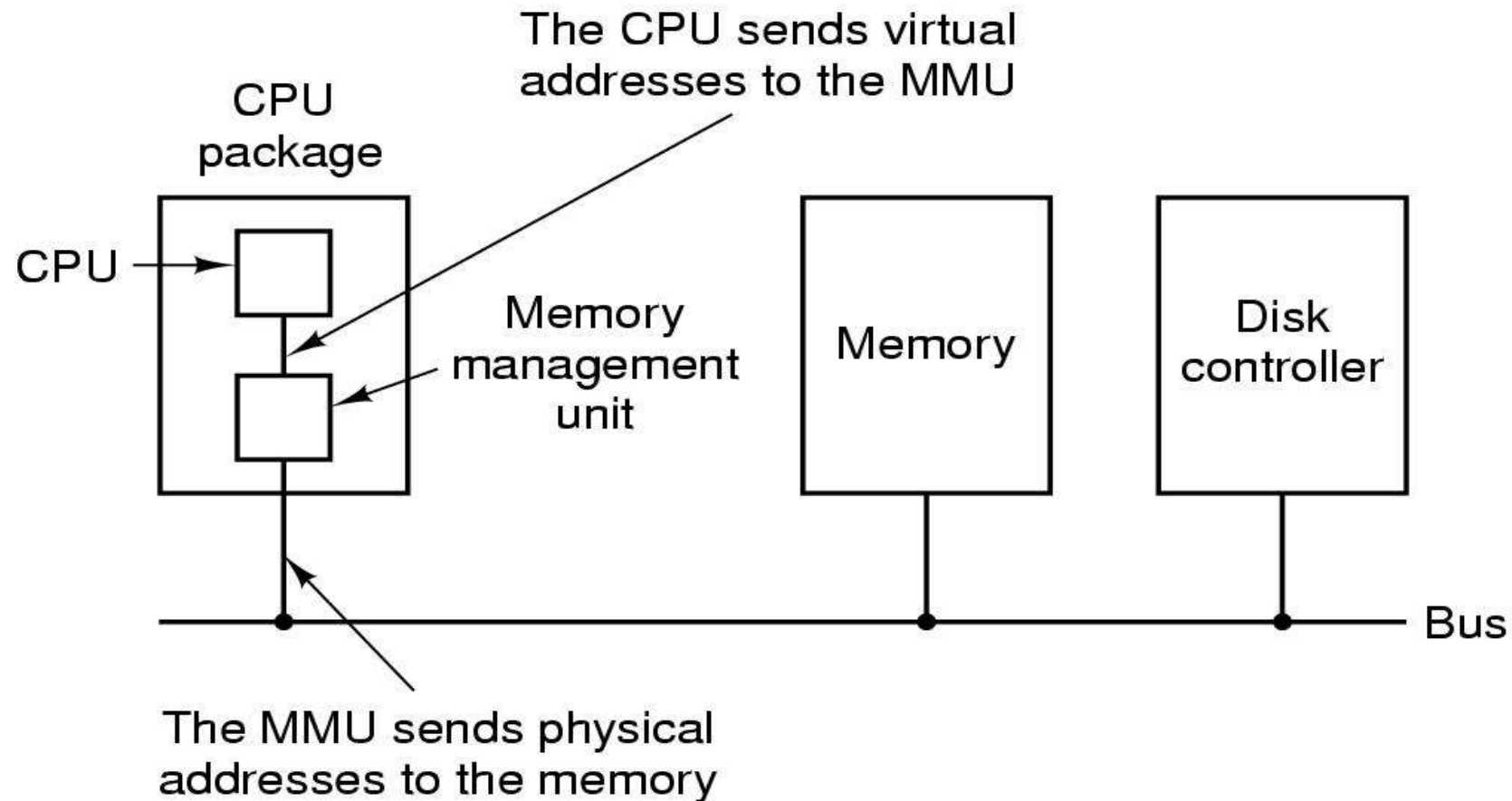
Frame size = 2^{12} = 4KB

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

Address Translation

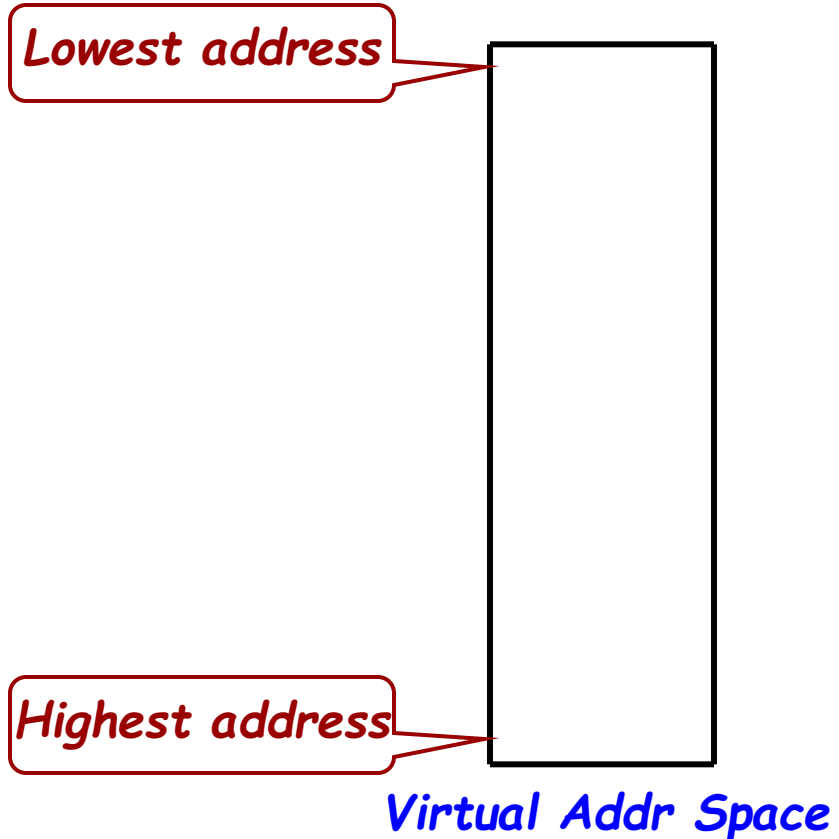
- Hardware maps **page** numbers to **frame** numbers
- Memory management unit (MMU) has multiple registers for multiple pages
 - *Like a base register except its value is substituted for the page number rather than added to it*
 - *Why don't we need a limit register for each page?*

Memory Management Unit (MMU)



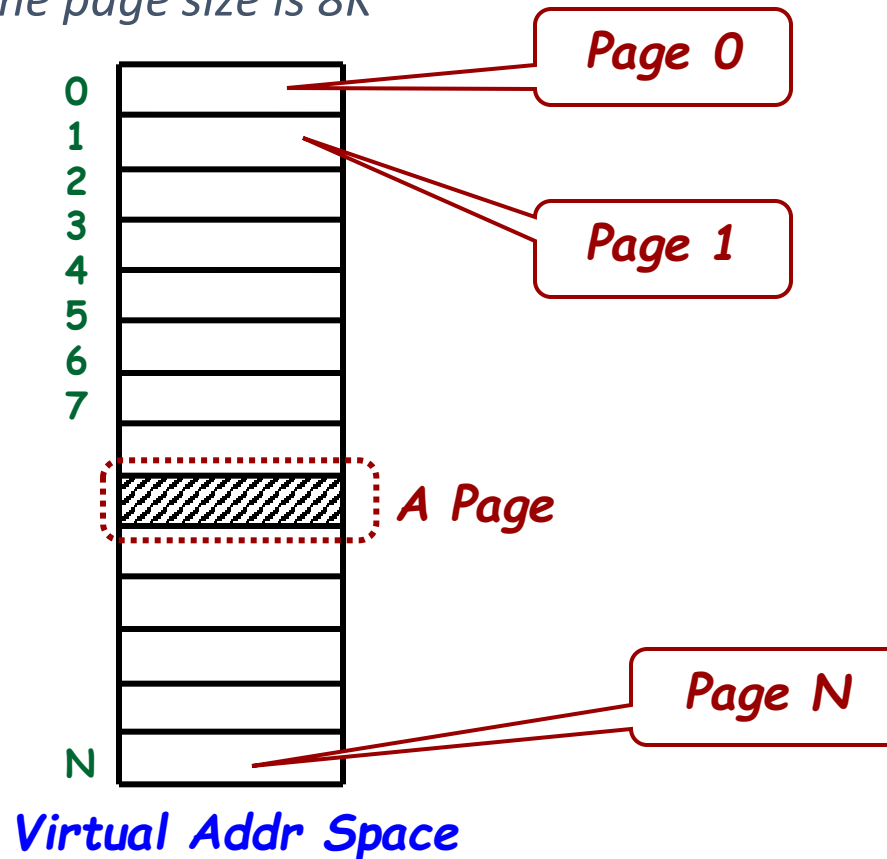
Virtual Address Spaces

- Here is the virtual address space (as seen by the process)



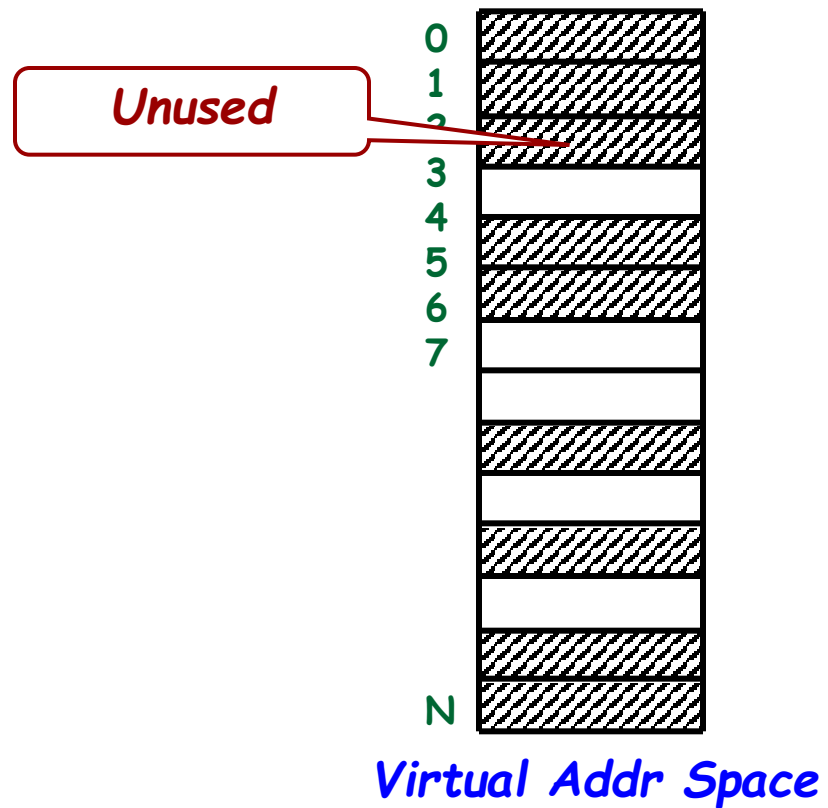
Virtual Address Spaces

- The address space is divided into “pages”
 - In BLITZ, the page size is 8K



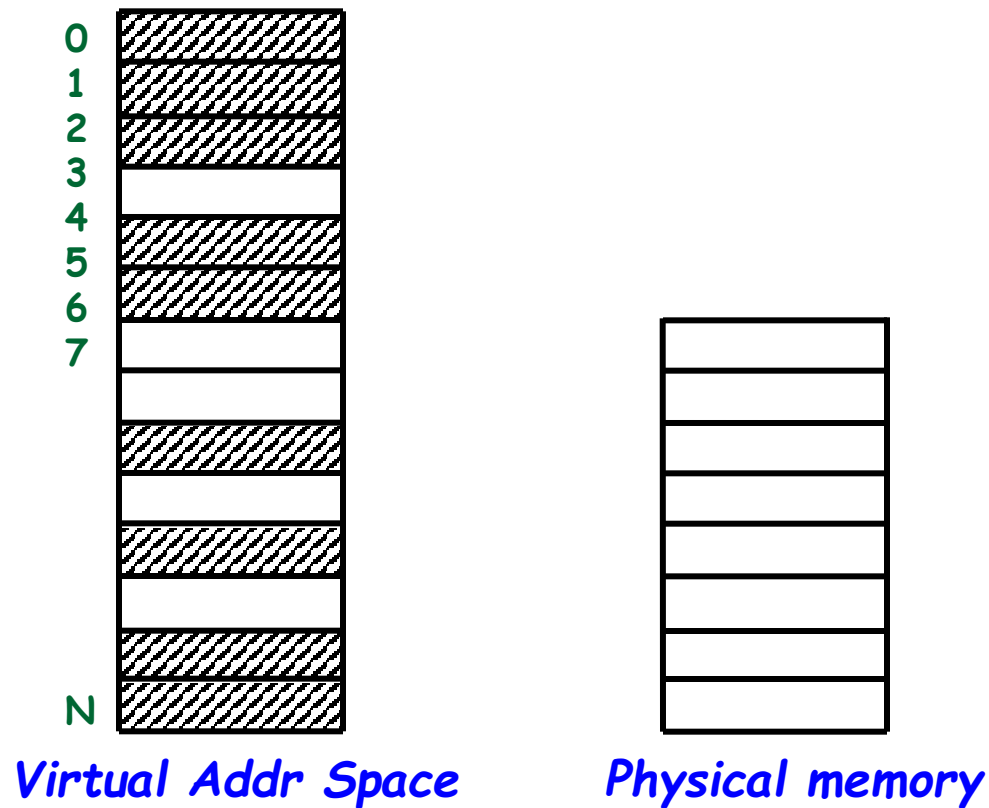
Virtual Address Spaces

- In reality, only some of the pages are used



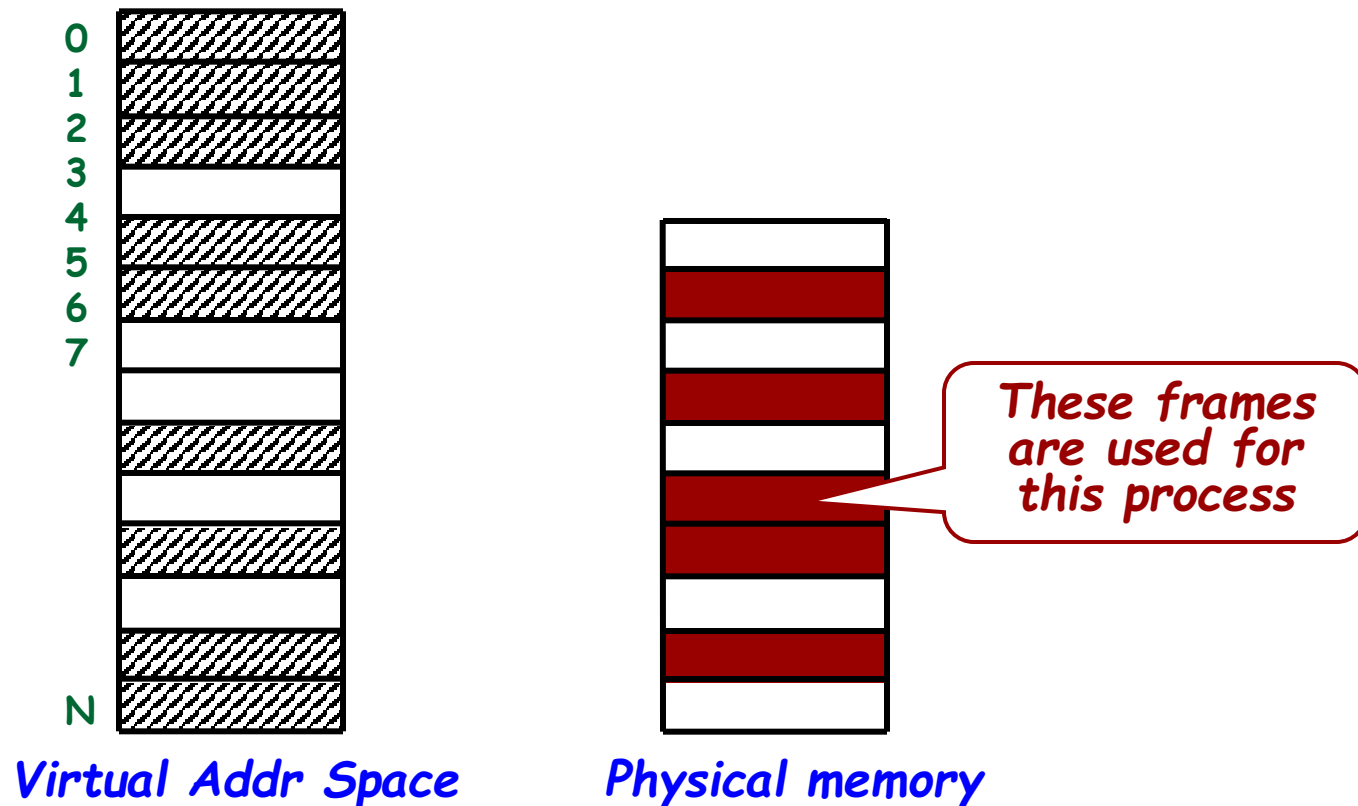
Physical Memory

- Physical memory is divided into “*page frames*”
 - (*Page size = frame size*)



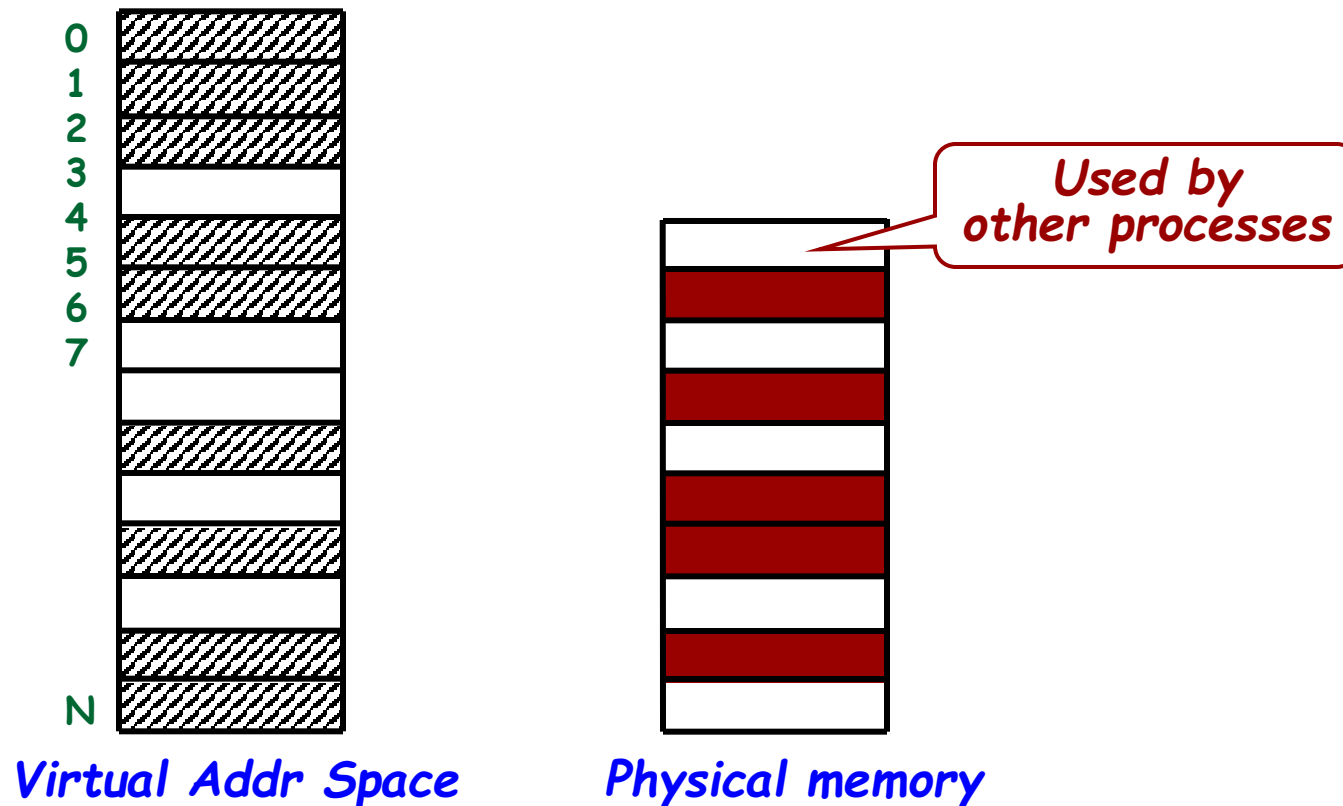
Virtual & Physical Address Spaces

- Some frames are used to hold the pages of this process



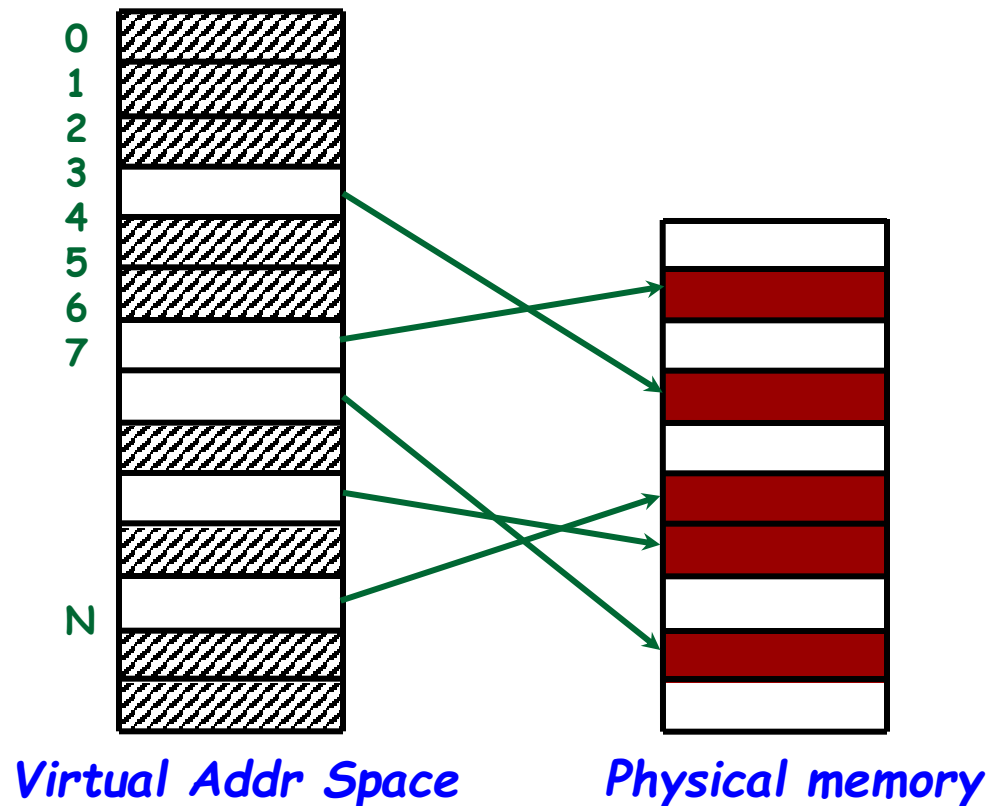
Virtual & Physical Address Spaces

- Some frames are used for other processes



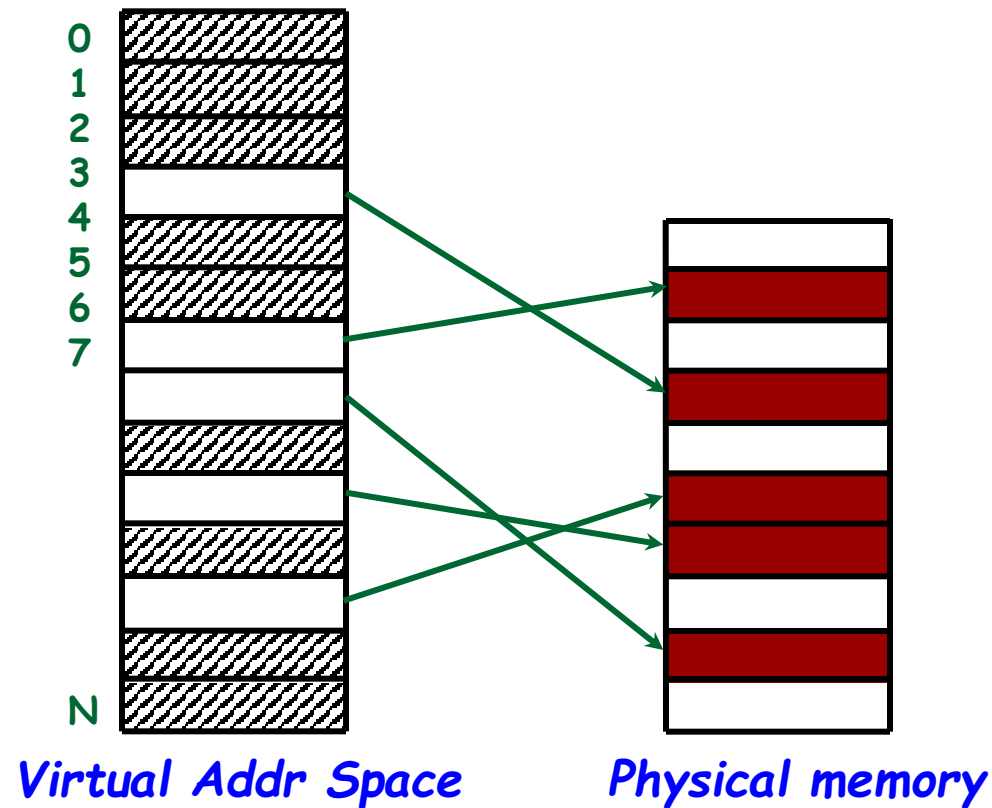
Virtual & Physical Address Spaces

- Address **mappings** say which frame has which page



Page Tables

- Address mappings are stored in a *page table* in memory
- 1 entry/page: is page in memory? If so, which frame is it in?



Address Mappings

- Address mappings are stored in a page table in memory
 - Typically one page table for each process
- Address translation is done by hardware (ie the MMU)

PAGE TABLE IN BLITZ

بررسی پیاده‌سازی Page Table در Blitz

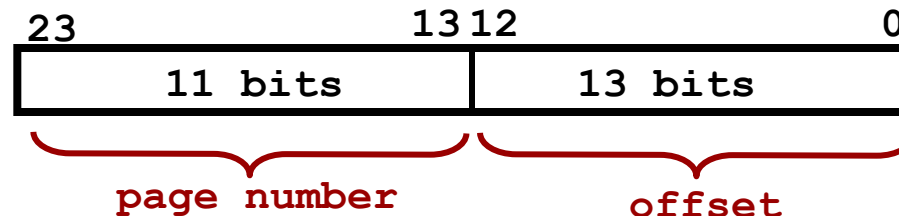
The BLITZ Architecture

- Page size
 - *8 Kbytes (13 bits for offset)*
- Virtual addresses (“logical addresses”)
 - *24 bits --> 16 Mbyte virtual address space*
 - *2^{11} Pages --> 11 bits for page number*

The BLITZ Architecture

- Page size
 - 8 Kbytes
- Virtual addresses (“logical addresses”)
 - 24 bits --> 16 Mbyte virtual address space
 - 2^{11} Pages --> 11 bits for page number

- An address:



The BLITZ Architecture

- Physical addresses

- *32 bits --> 4 Gbyte installed memory (max)*
- *2^{19} Frames --> 19 bits for frame number*

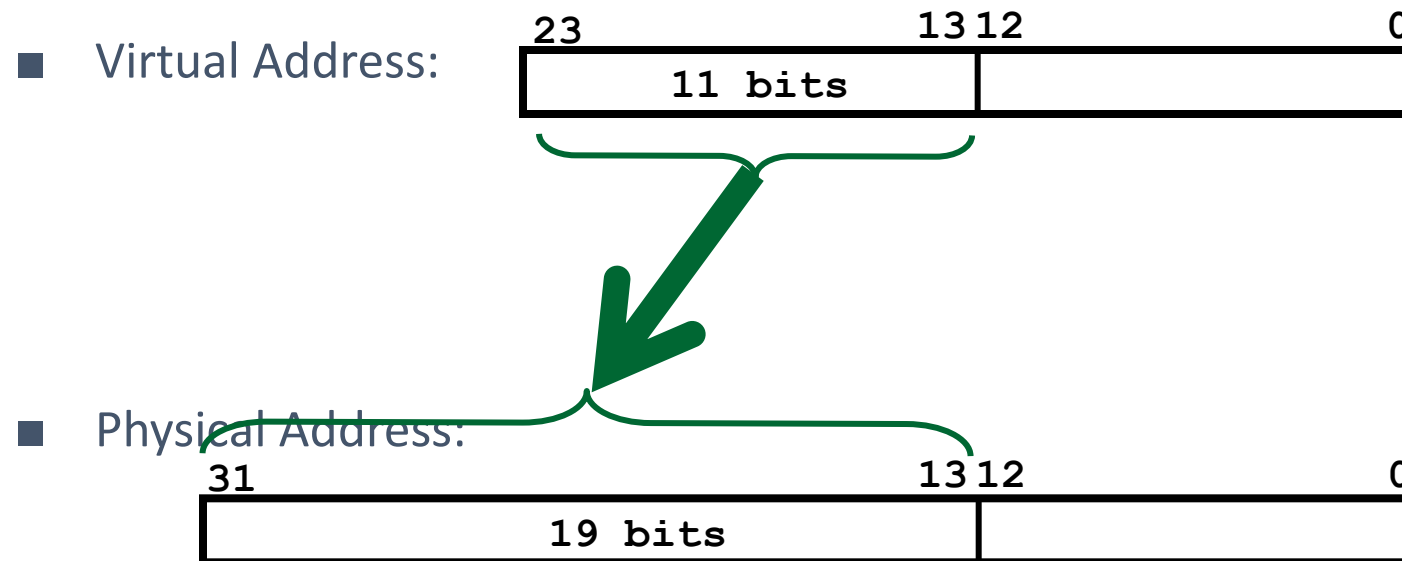
The BLITZ Architecture

- Physical addresses
 - 32 bits --> 4 Gbyte installed memory (max)
 - 2^{19} Frames --> 19 bits for frame number



The BLITZ Architecture

- The page table mapping:
 - *Page --> Frame*



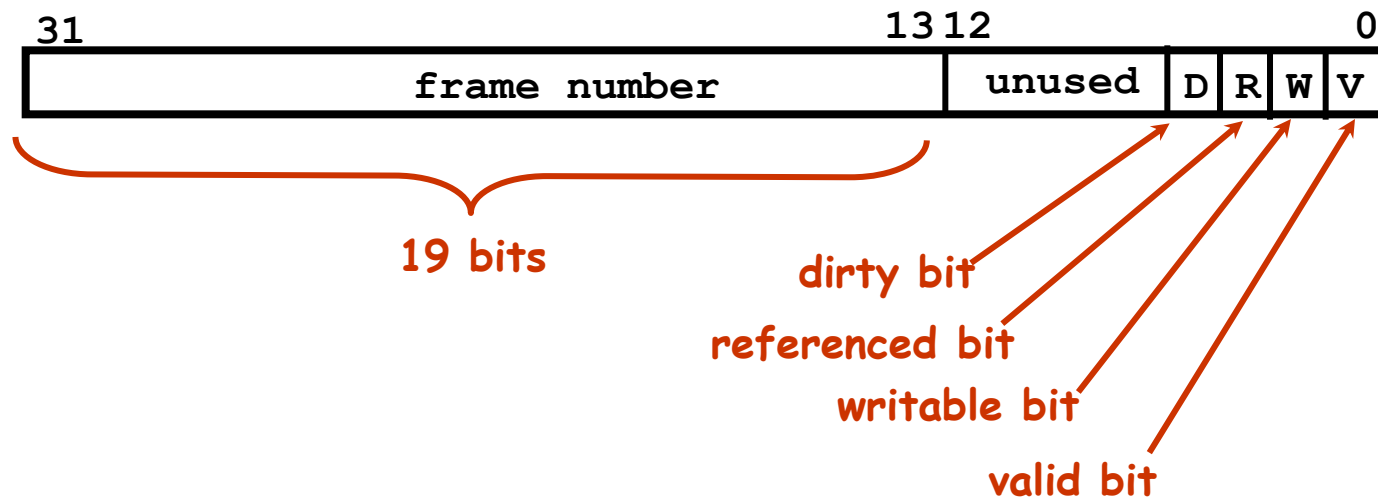
The BLITZ Page Table

- 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 “current” page table
 - *This is how the CPU knows which page table to use*
 - *Must be saved and restored on context switch*
 - *They are essentially the Blitz MMU*
- 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



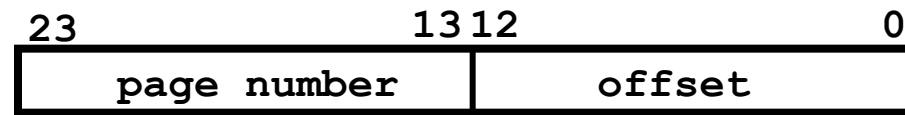
The BLITZ Page Table

page table base register

	31	13 12	0
0	frame number	unused	D R W V
1	frame number	unused	D R W V
2	frame number	unused	D R W V
	frame number	unused	D R W V
2K	frame number	unused	D R W V


Indexed by the page number

The BLITZ Page Table



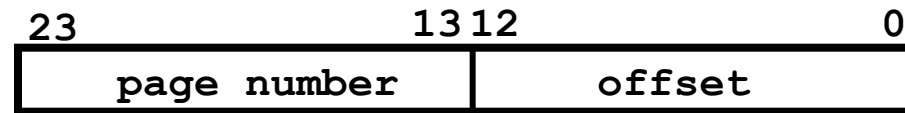
virtual address

page table base register




	31	13	12	0		
0	frame number	unused	D	R	W	V
1	frame number	unused	D	R	W	V
2	frame number	unused	D	R	W	V
	frame number	unused	D	R	W	V
2K	frame number	unused	D	R	W	V

The BLITZ Page Table



virtual address

page table base register

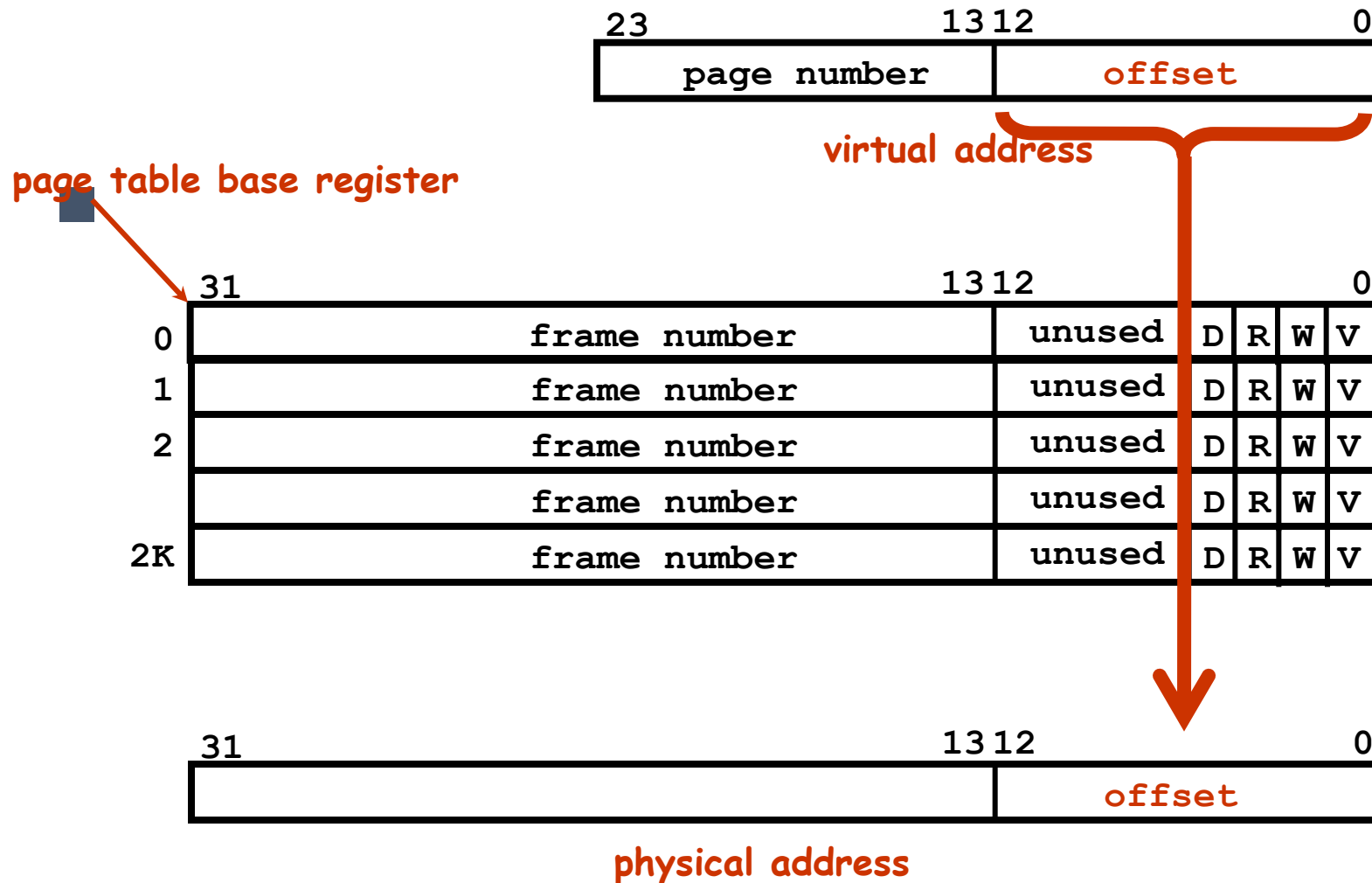


	31	13	12	0		
0	frame number	unused	D	R	W	V
1	frame number	unused	D	R	W	V
2	frame number	unused	D	R	W	V
	frame number	unused	D	R	W	V
2K	frame number	unused	D	R	W	V

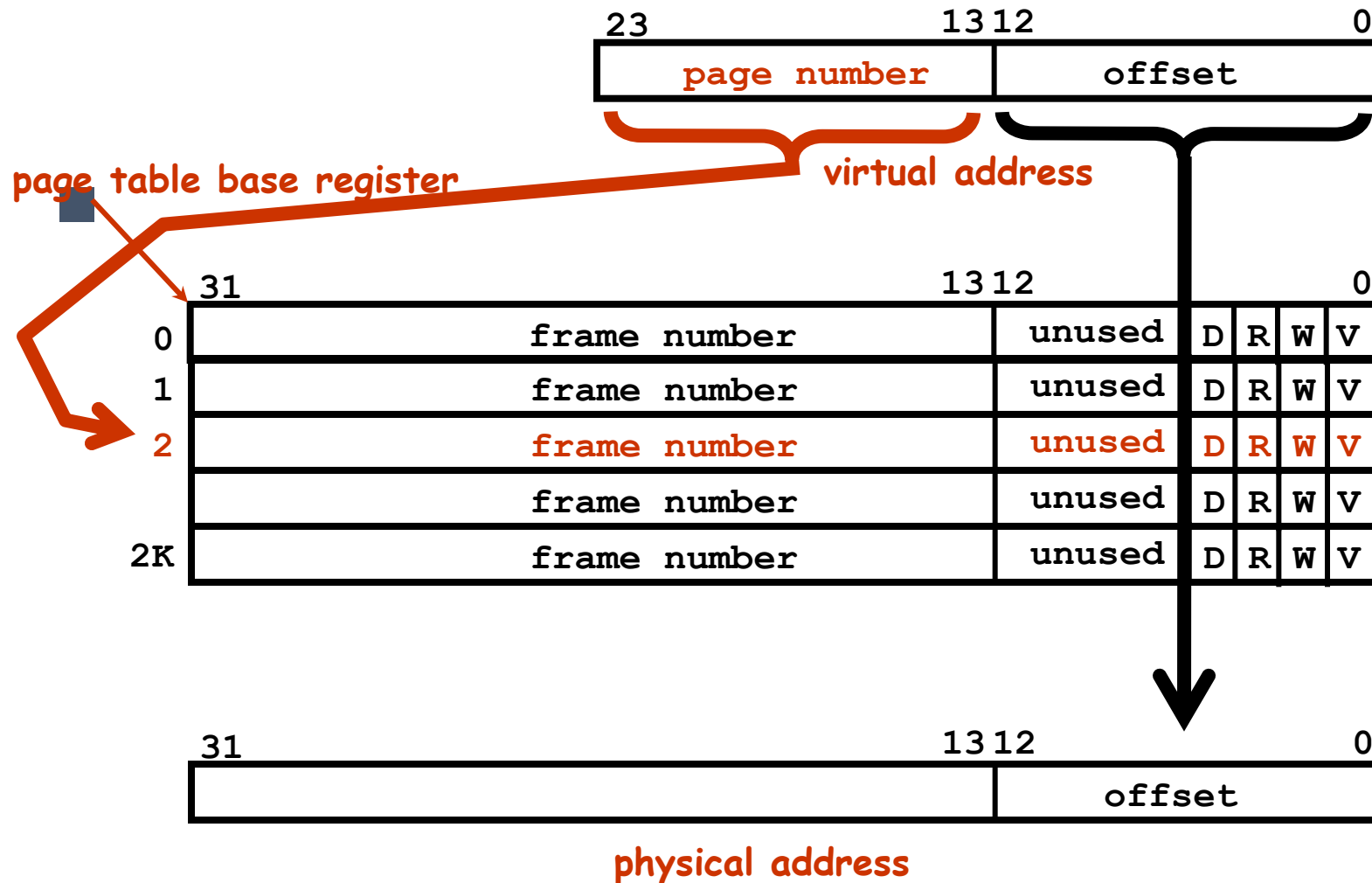


physical address

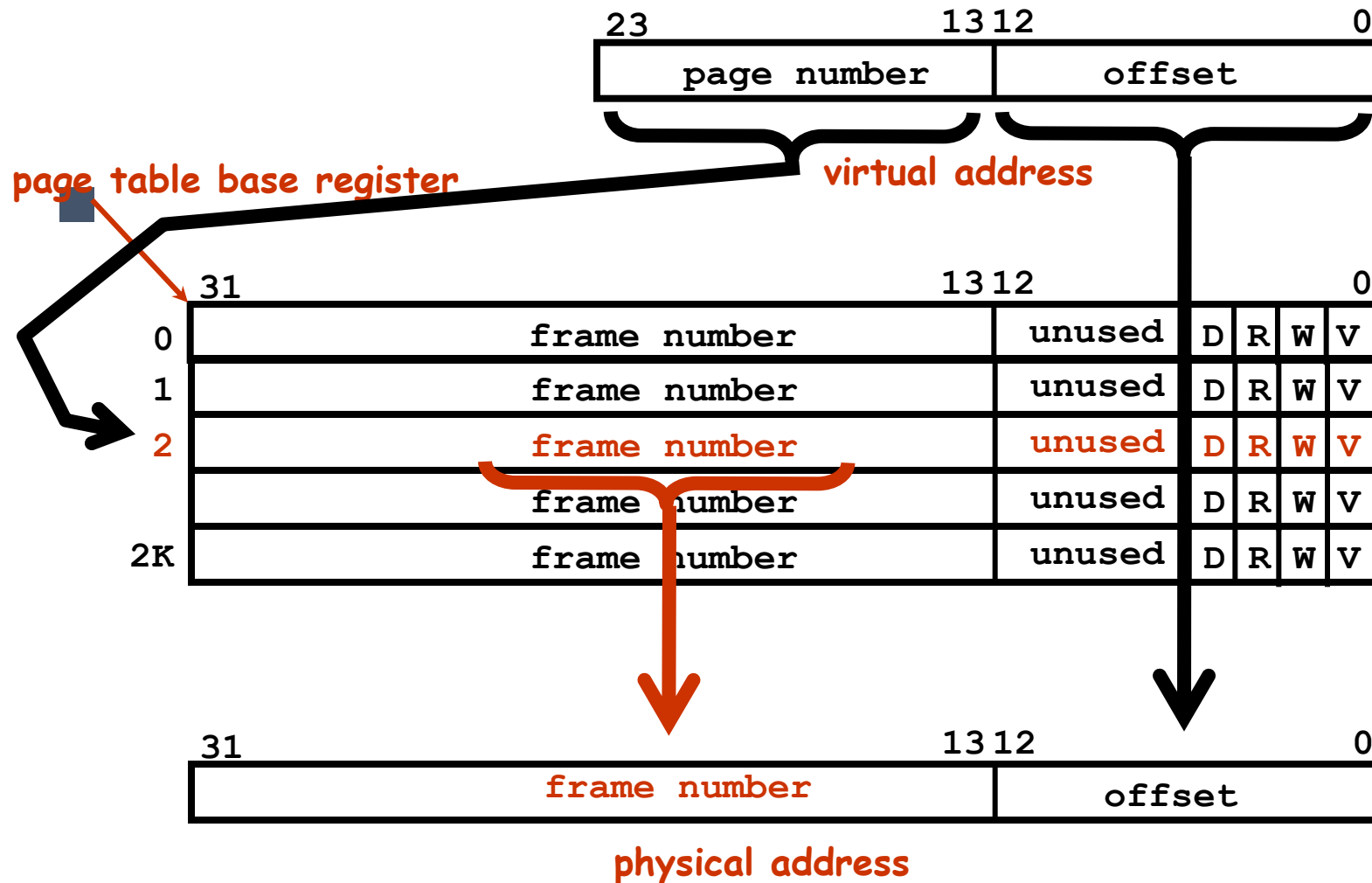
The BLITZ Page Table



The BLITZ Page Table



The BLITZ Page Table



TLB

Translation Look-Aside Buffer

Address Mappings

- Address mappings are stored in a page table in memory
 - Typically one page table for each process
- Address translation is done by hardware (ie the MMU)
- How does the MMU get the address mappings?
 - Either the MMU holds the entire page table (too expensive) or it knows where it is in physical memory and goes there for every translation (too slow)
 - Or the MMU holds a portion of the page table and knows how to deal with TLB misses
 - MMU caches page table entries
 - Cache is called a translation look-aside buffer (TLB)

Address Mappings & TLB

- What if the TLB needs a mapping it doesn't have?
- Software managed TLB
 - *It generates a **TLB-miss fault** which is handled by the operating system (like interrupt or trap handling)*
 - *The operating system looks in the page tables, gets the mapping from the right entry, and puts it in the TLB*
- Hardware managed TLB
 - *It looks in a pre-specified physical memory location for the appropriate entry in the page table*
 - *The hardware architecture defines where page tables must be stored in physical memory*
 - *OS must load current process page table there on context switch!*

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

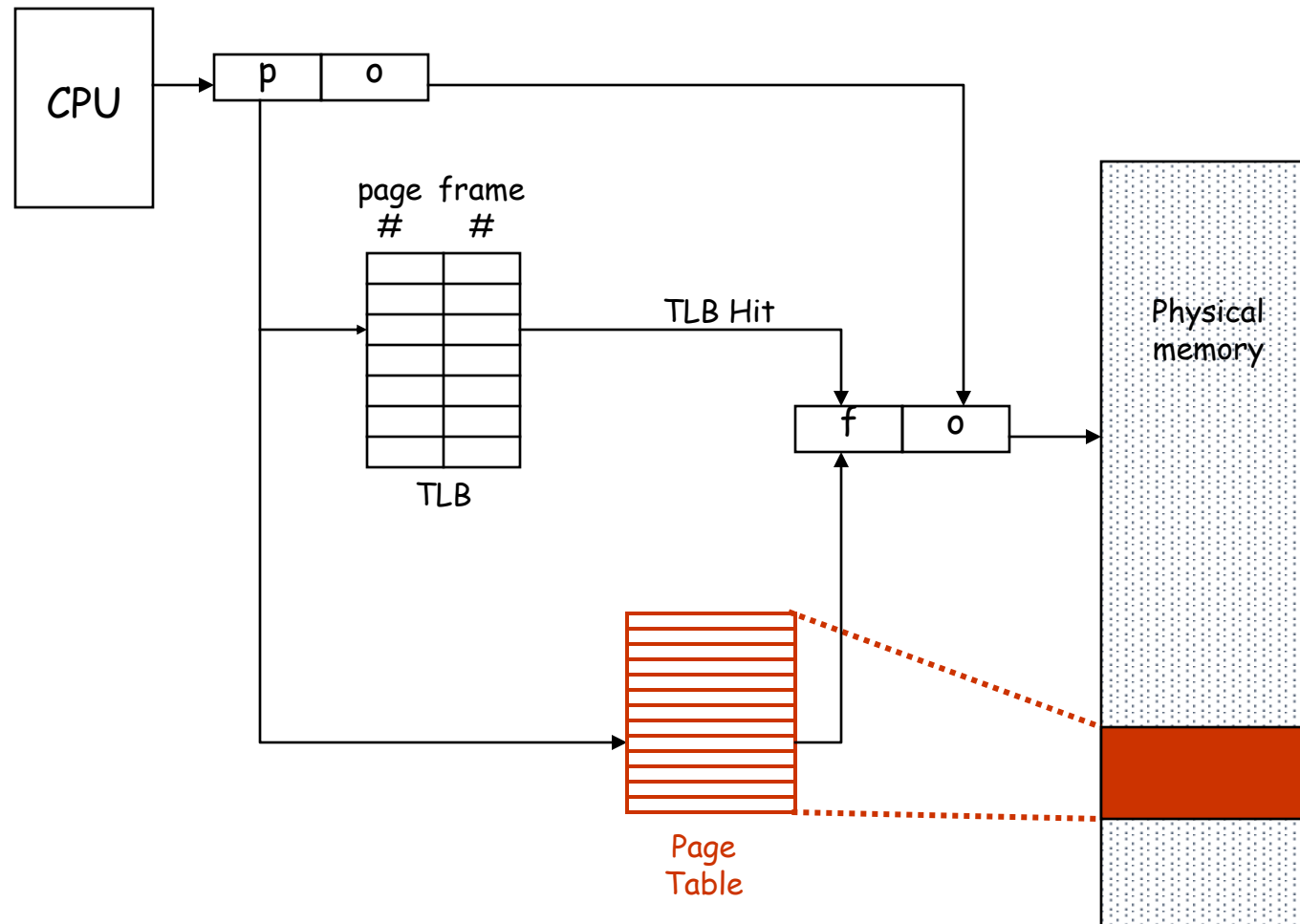
- **Problem:** MMU can't keep up with the CPU if it goes to the page table on every memory access!

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



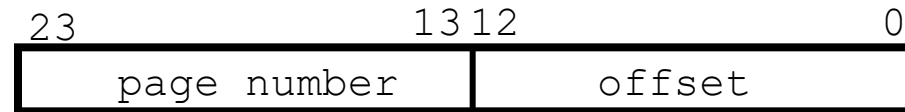
Hardware Operation of TLB

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

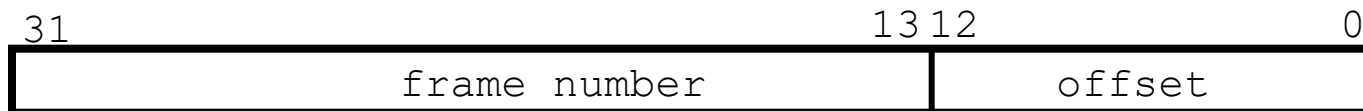
Hardware Operation of TLB

virtual address



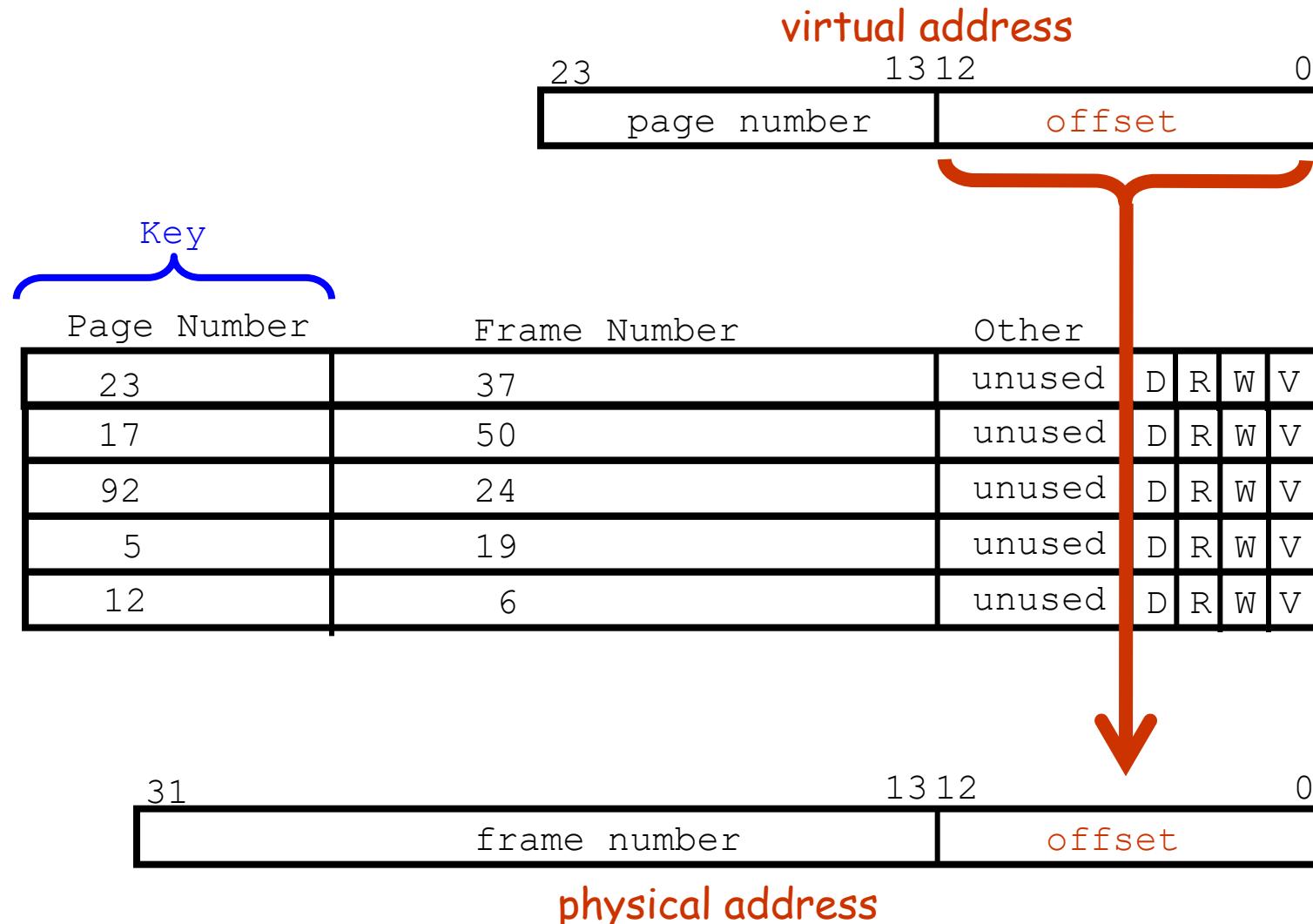
Key

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

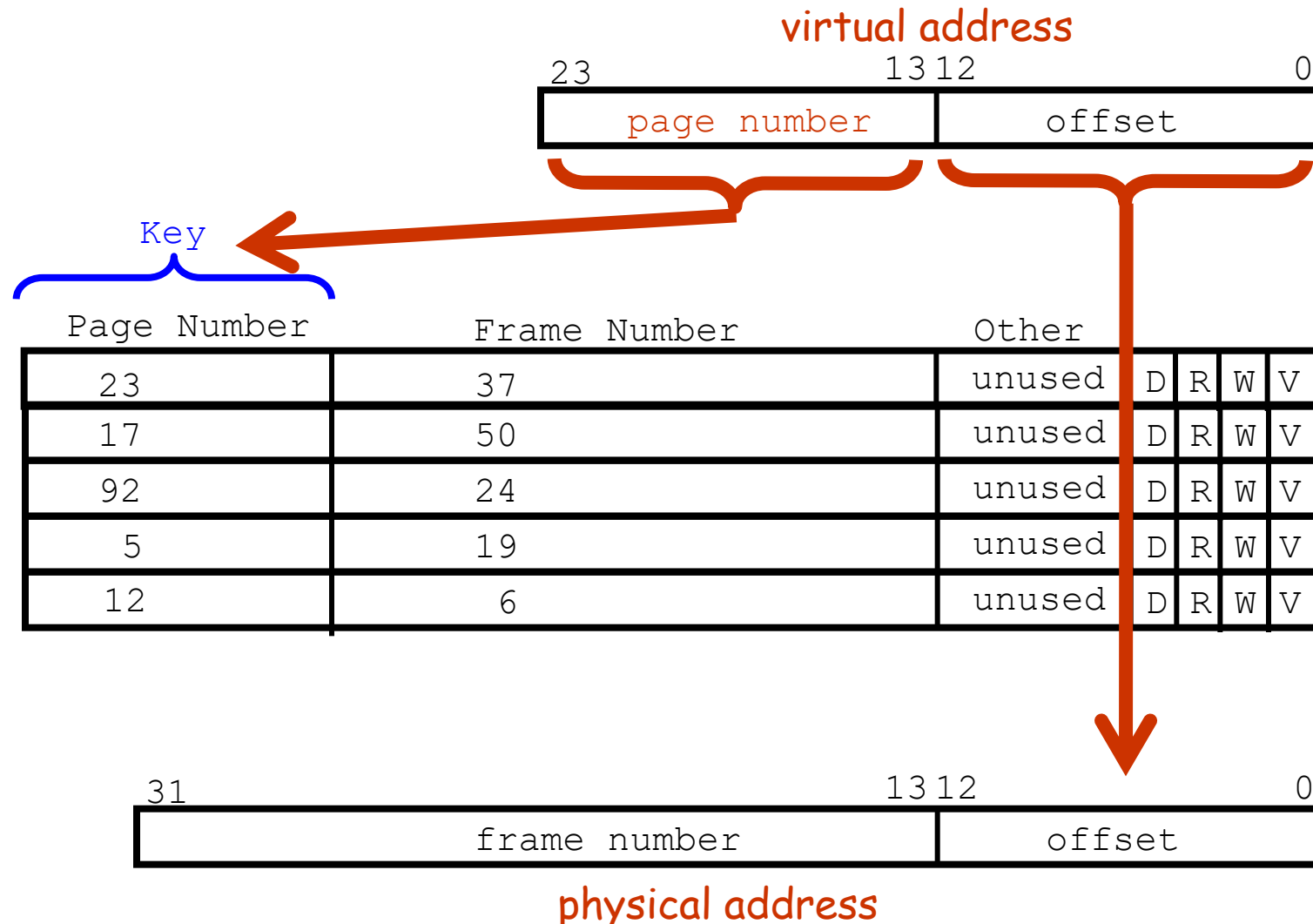


physical address

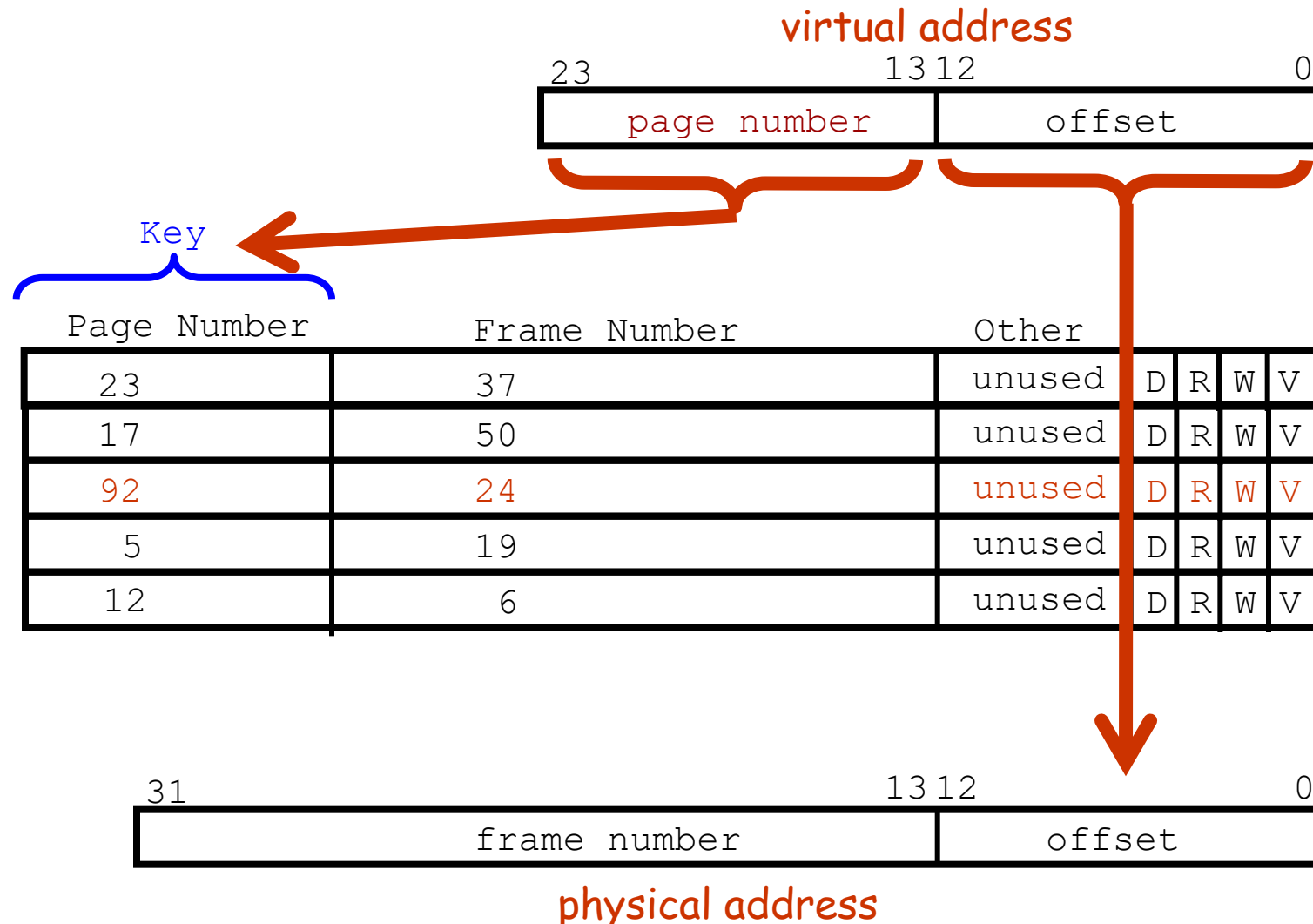
Hardware Operation of TLB



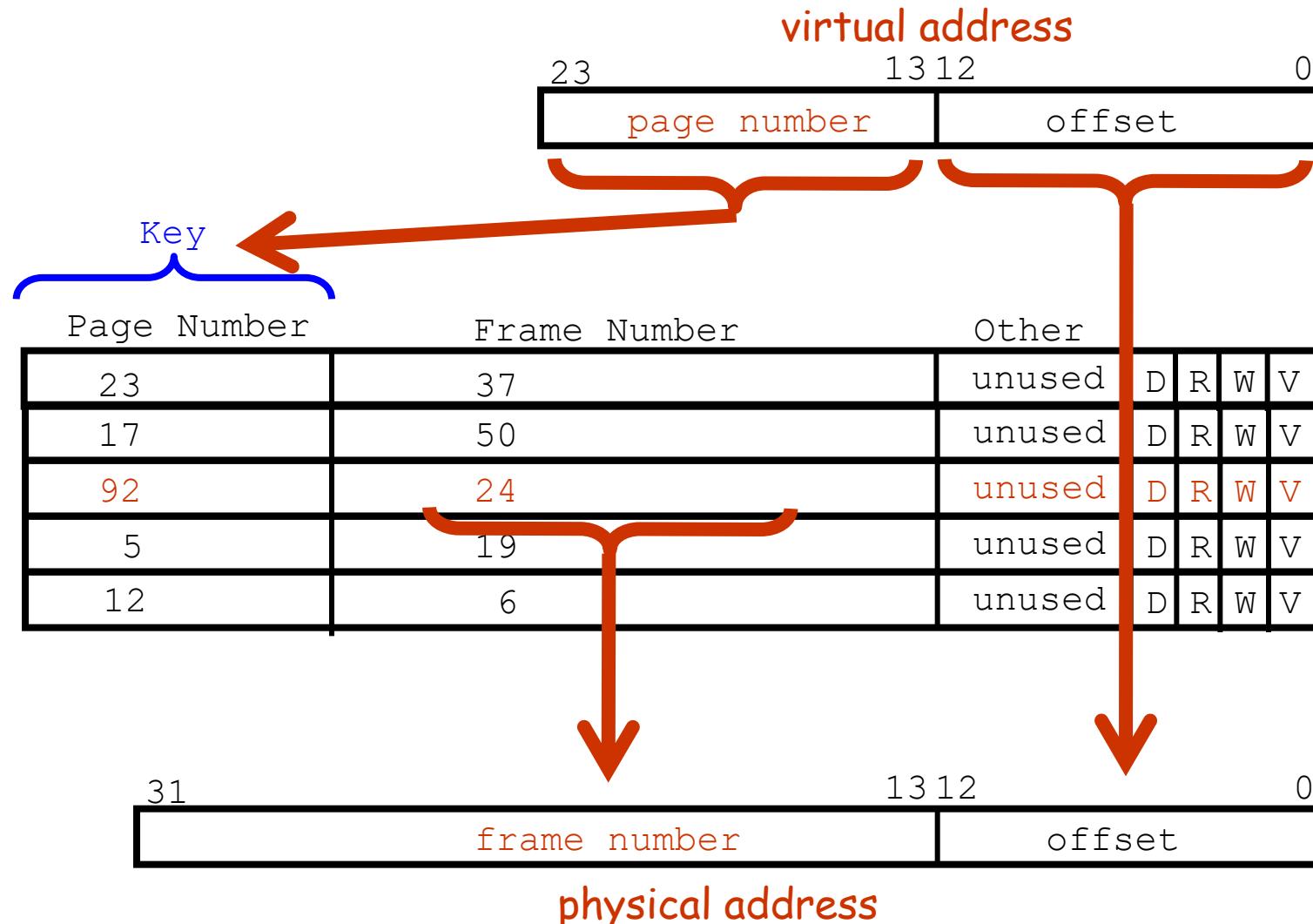
Hardware Operation of TLB



Hardware Operation of TLB



Hardware Operation of TLB



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

- How can we prevent the next process from using the last process's address mappings?
 - **Option 1:** *empty the TLB on context switch*
 - *New process will generate faults until its pulls enough of its own entries into the TLB*
 - **Option 2:** *just clear the “Valid Bit” on context switch*
 - *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*

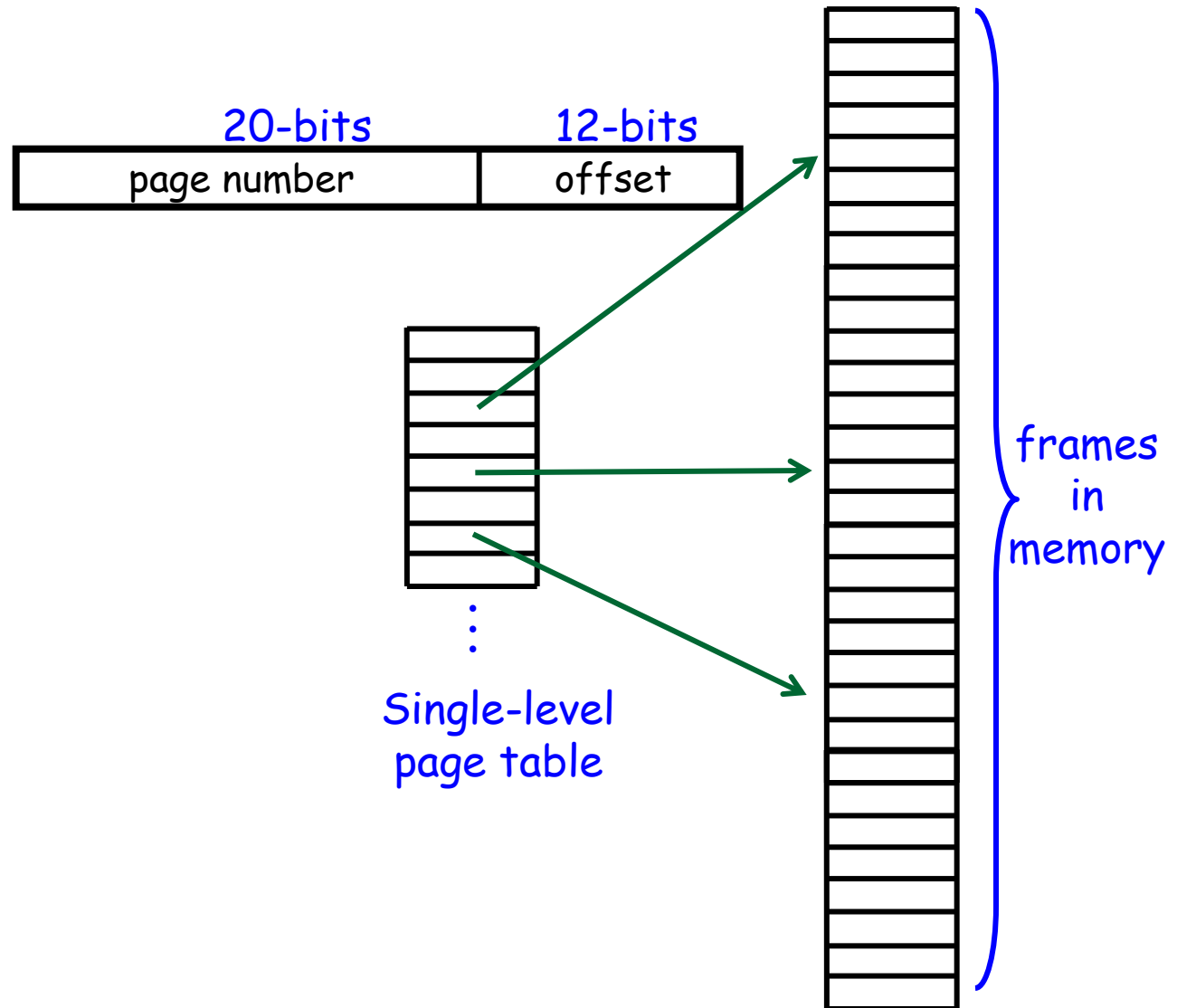
Page Table Usage

- 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*

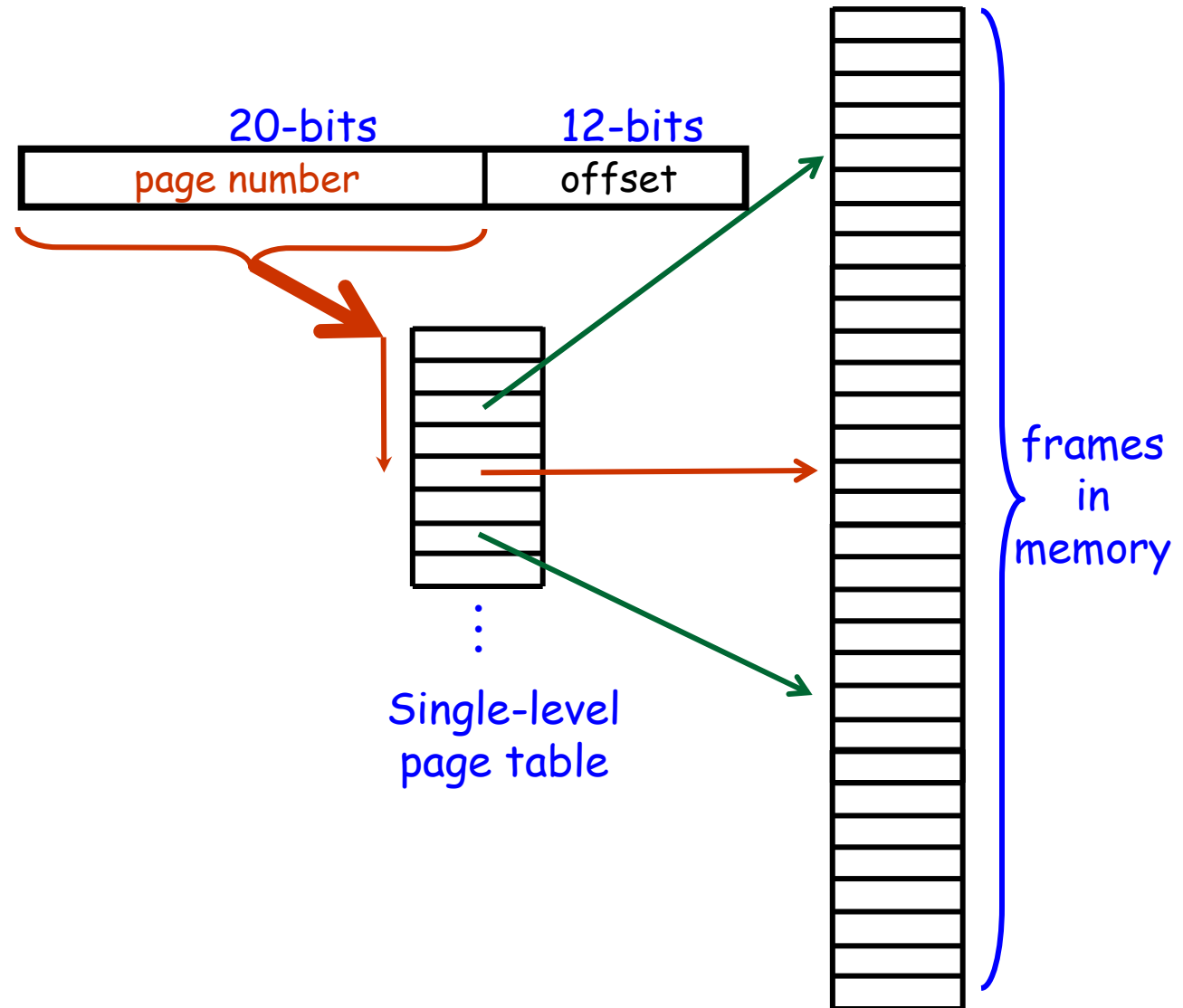
Page Table Design

- 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*

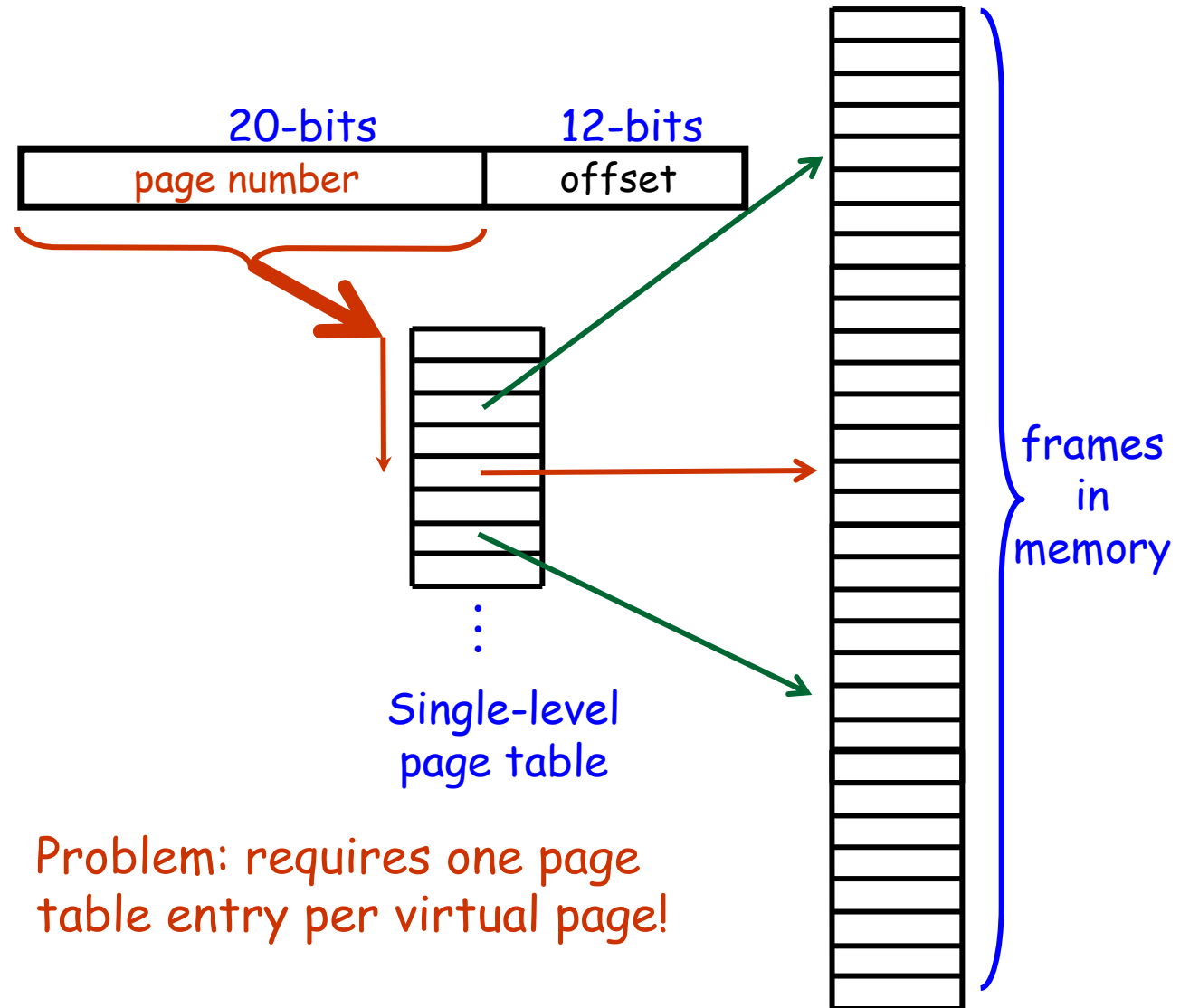
Single-Level Page Tables



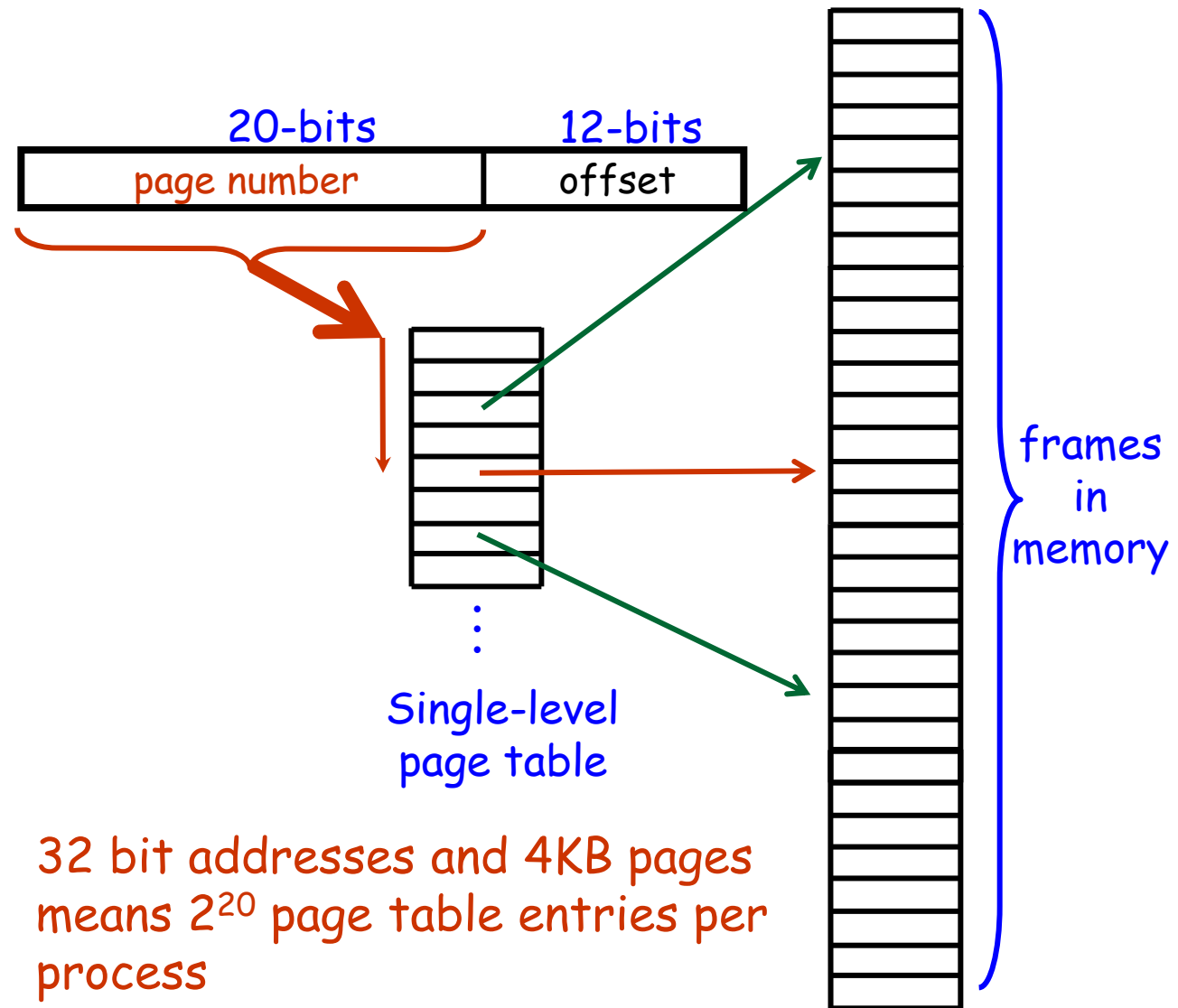
Single-Level Page Tables



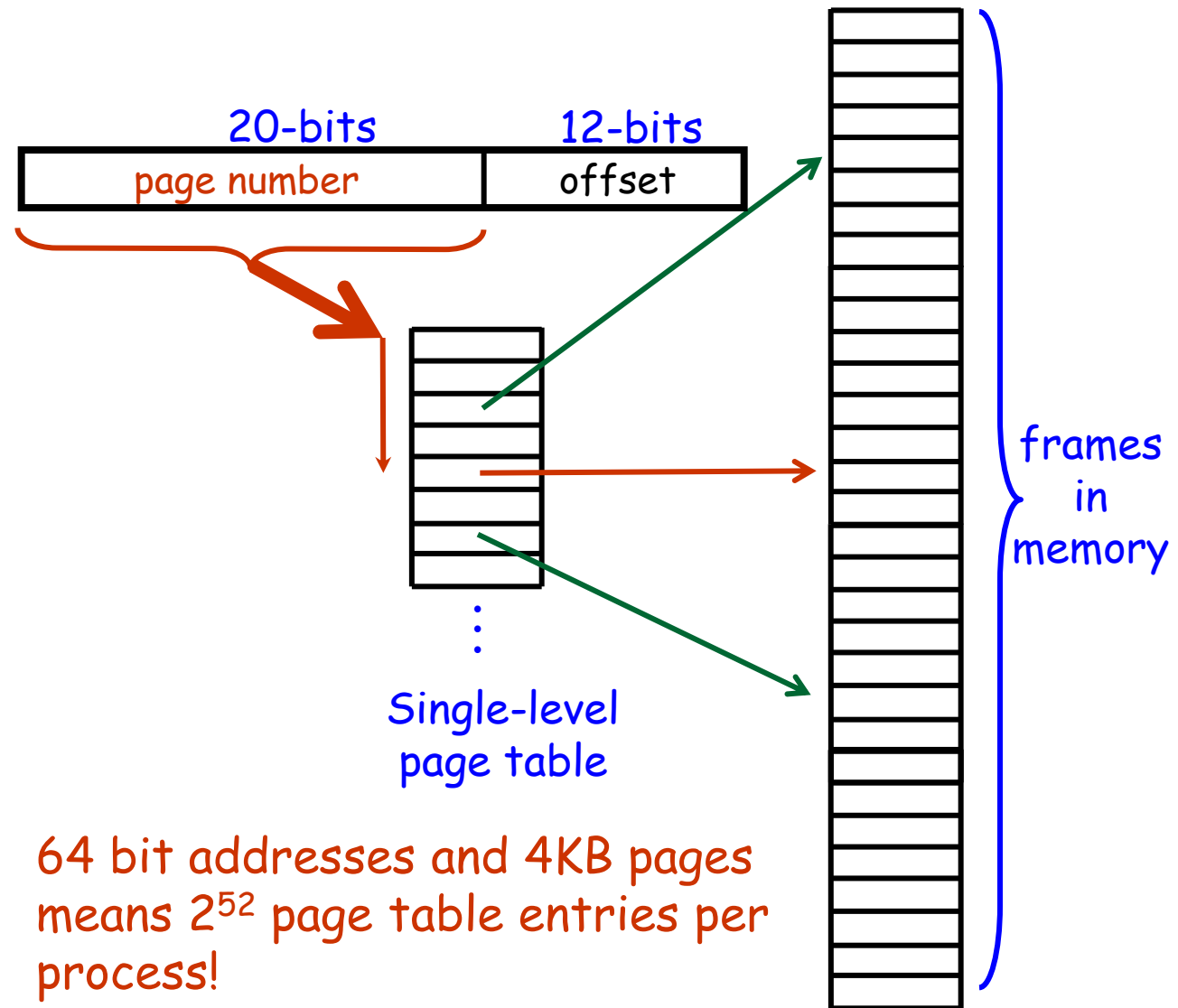
Single-Level Page Tables



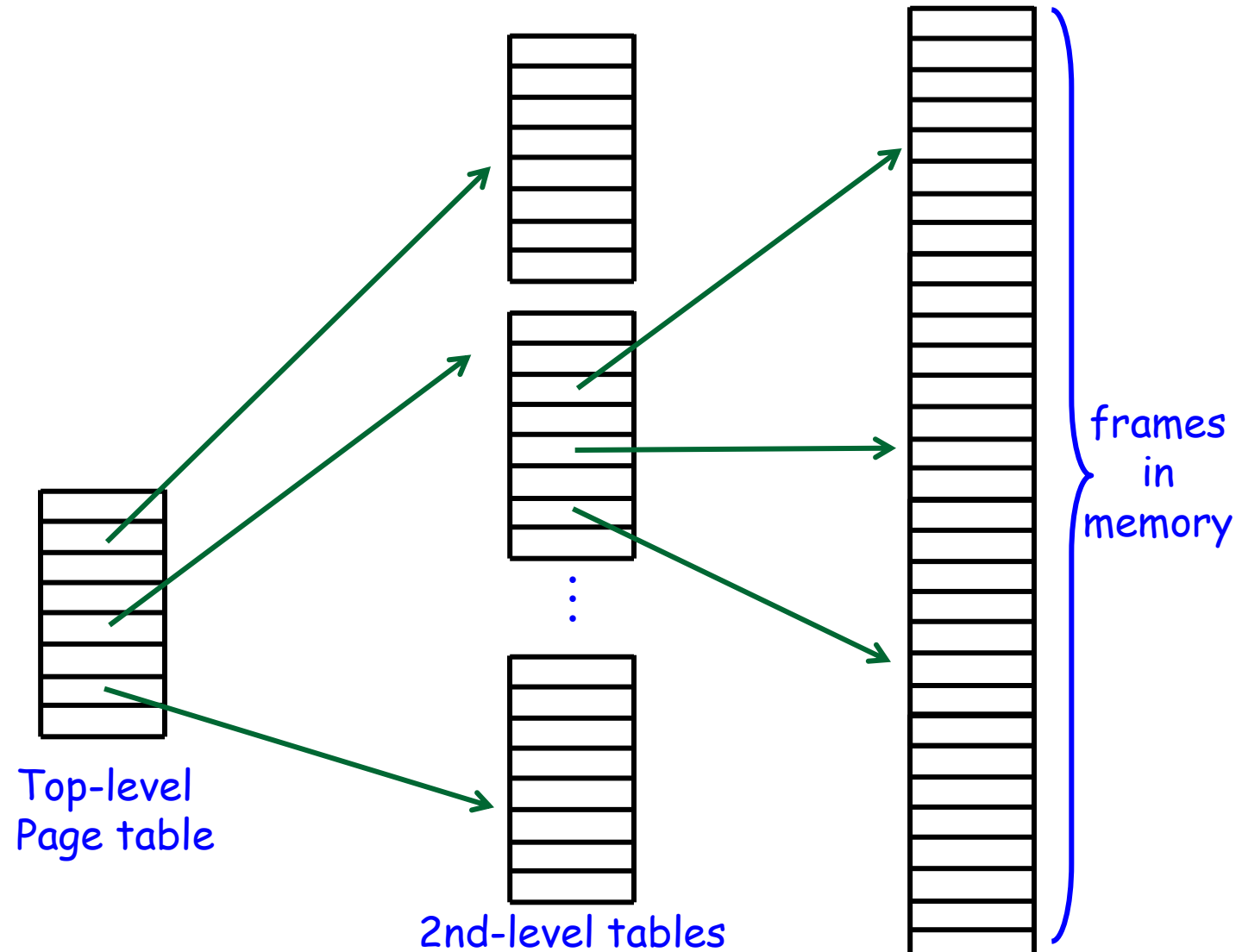
Single-Level Page Tables



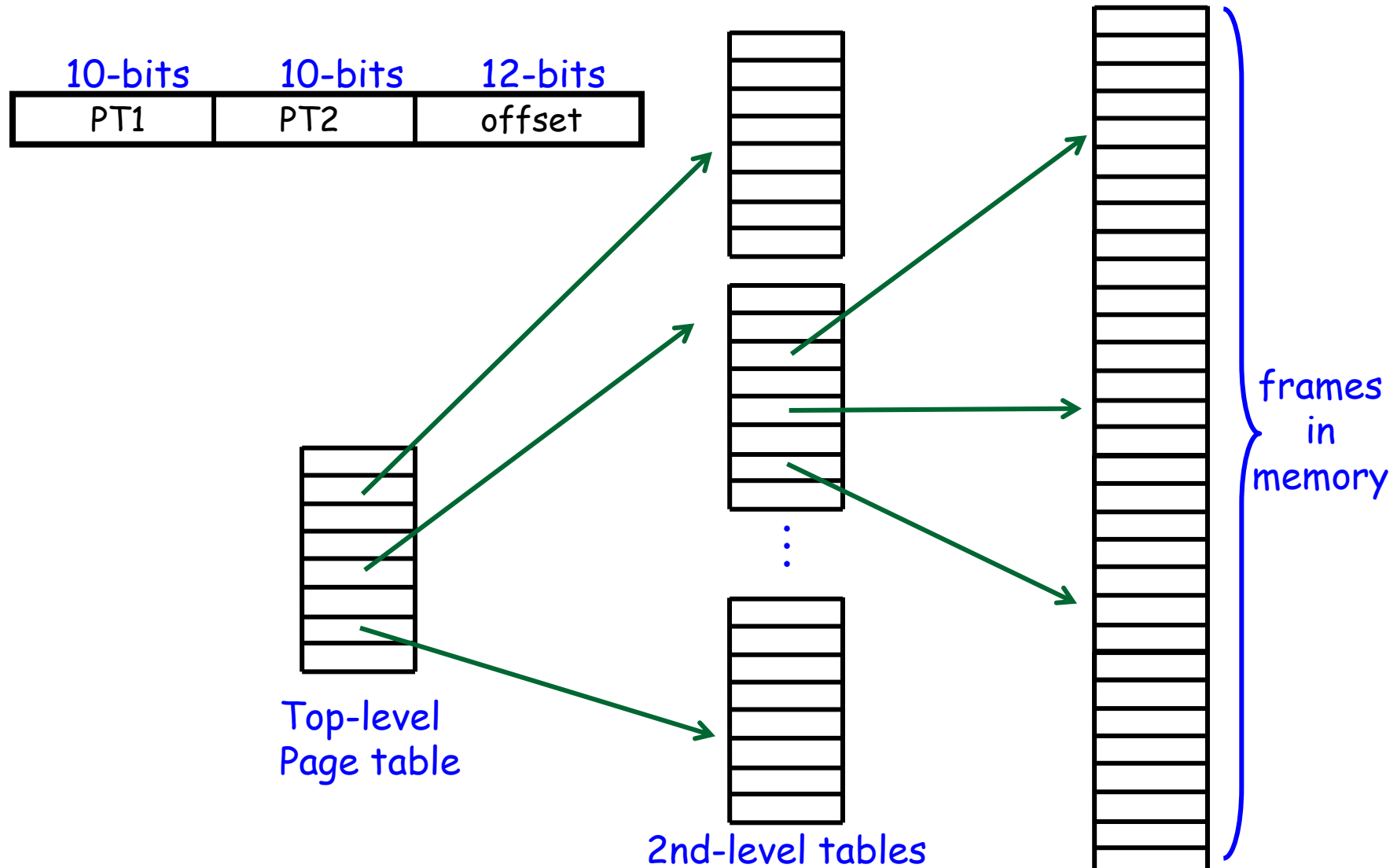
Single-Level Page Tables



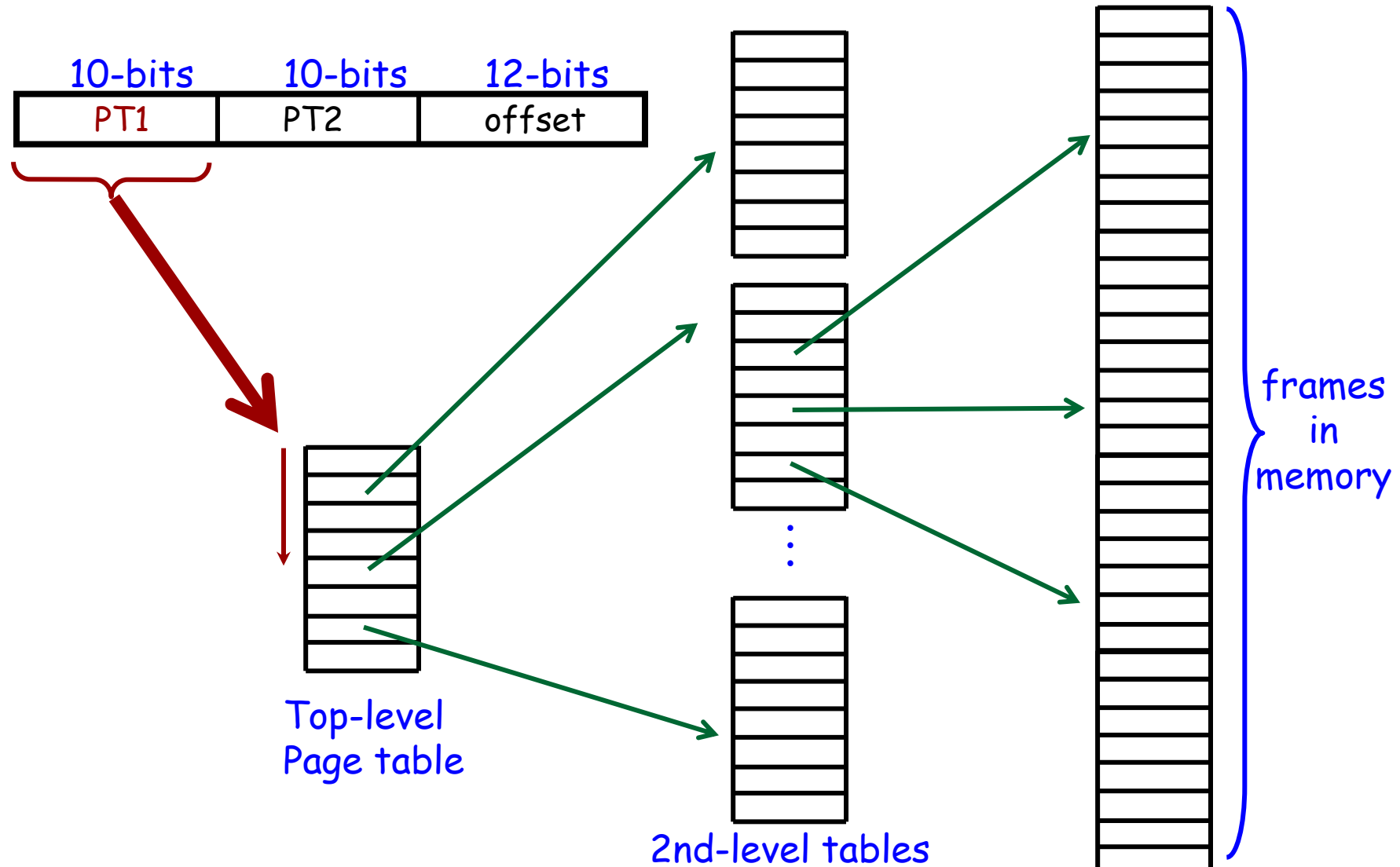
Multi-Level Page Tables



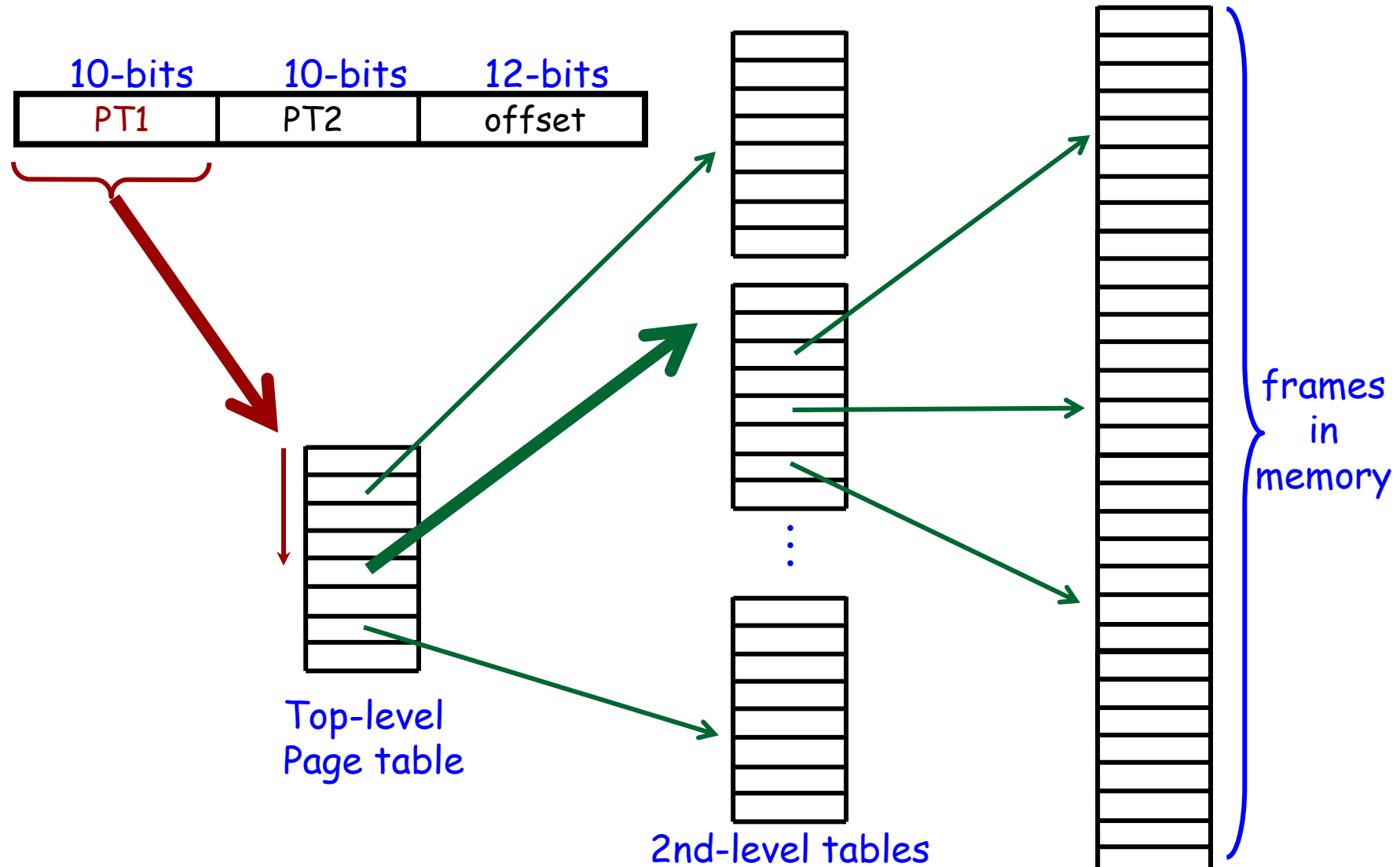
Multi-Level Page Tables



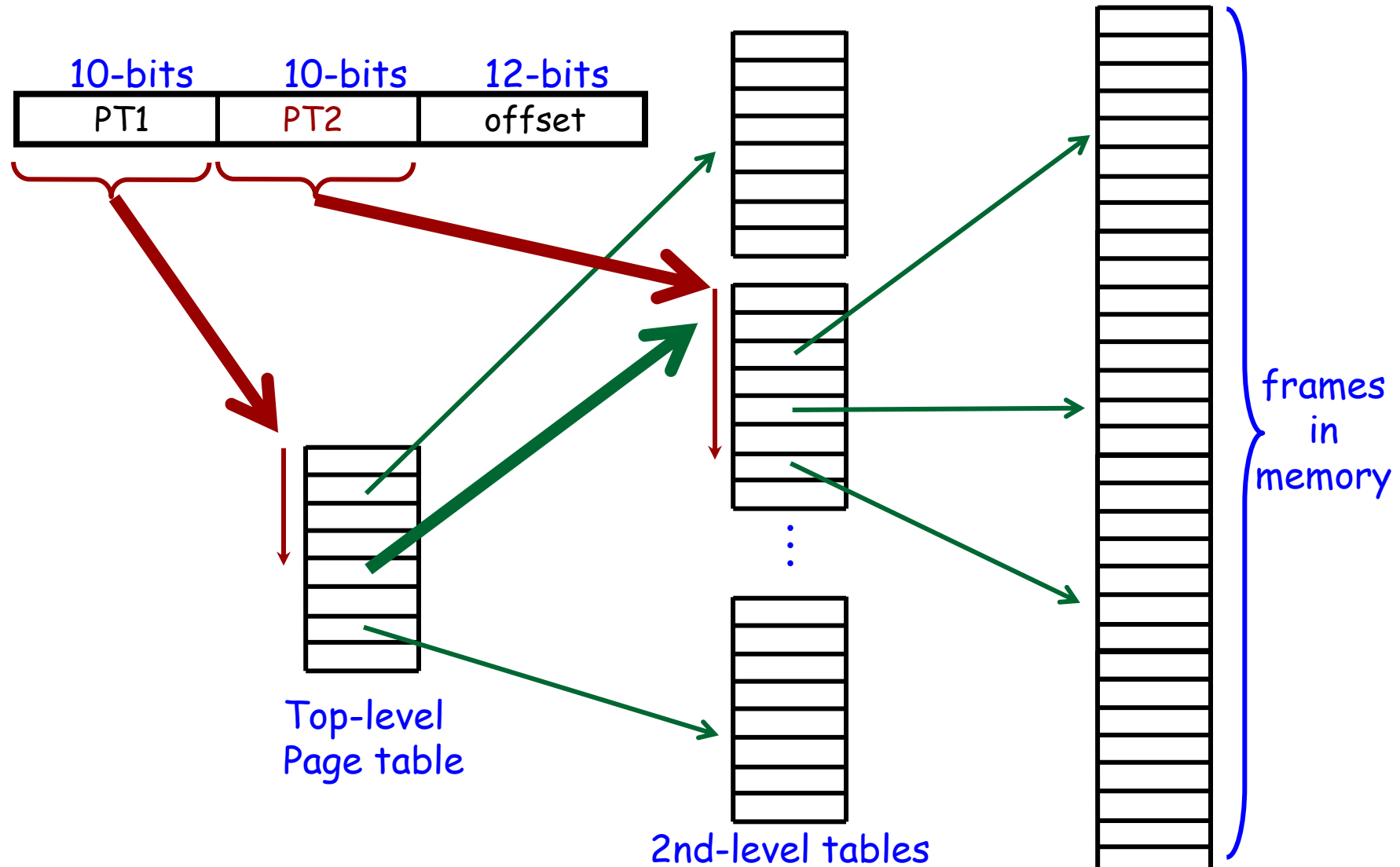
Multi-Level Page Tables



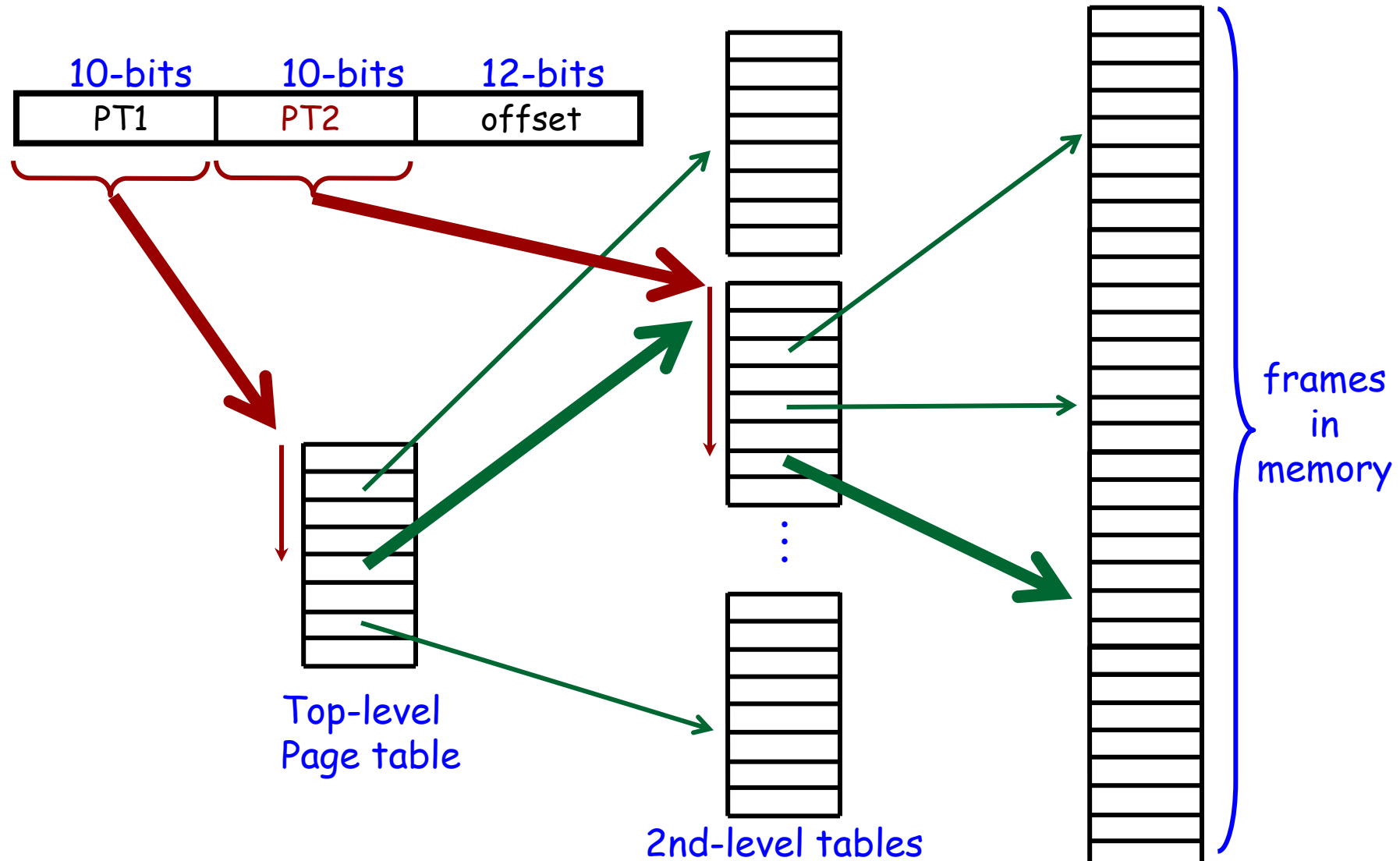
Multi-Level Page Tables



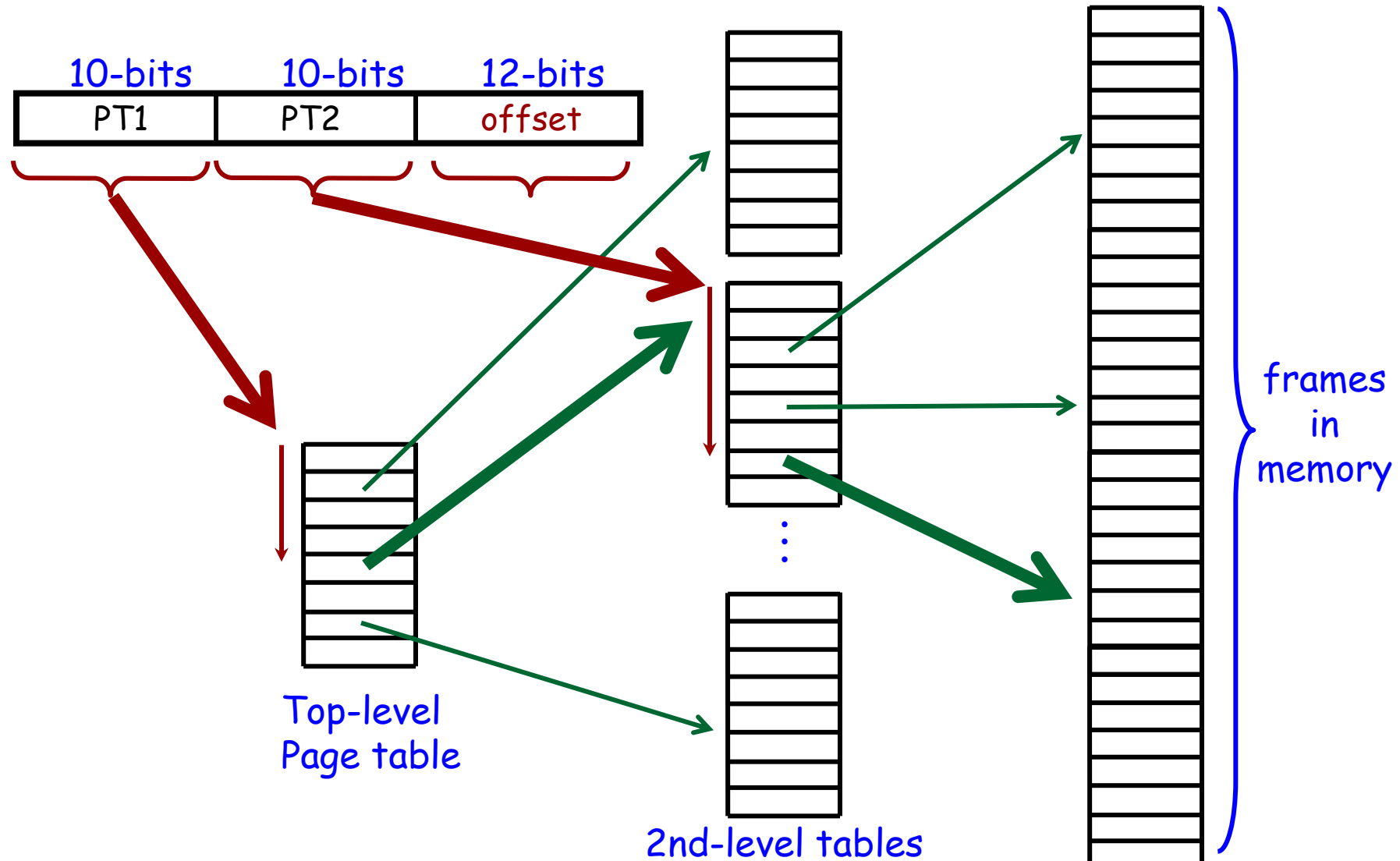
Multi-Level Page Tables



Multi-Level Page Tables



Multi-Level Page Tables



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!*

جلسه ی بعد...

ادامه ی موضوع: Inverted Page Tables ،Memory Protection ،Page Sharing