R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه چهاردهم – مدیریت حافظه (جدول صفحه)

جلسه‌ی گذشته

مدیریت حافظه – جدول صفحه

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*

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

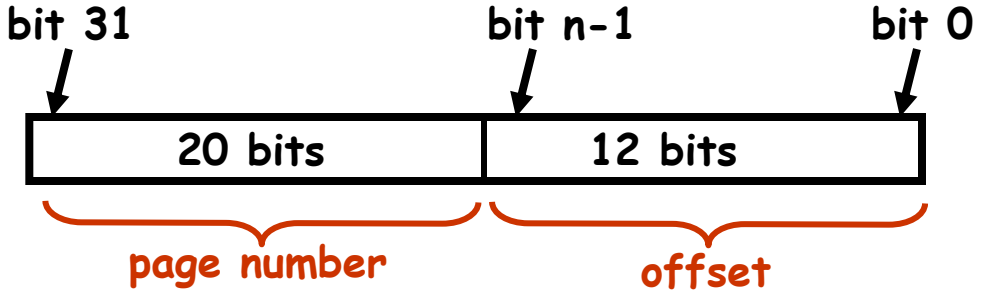
  How can we do secure dynamic address translation?

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

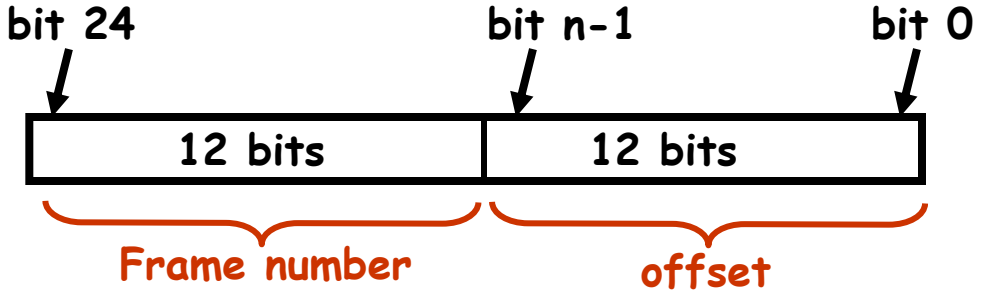
- Virtual memory addresses (what the process uses)
  - *Page number plus byte offset in page*
  - *Low order n bits are the byte offset*
  - *Remaining high order bits are the page number*



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*

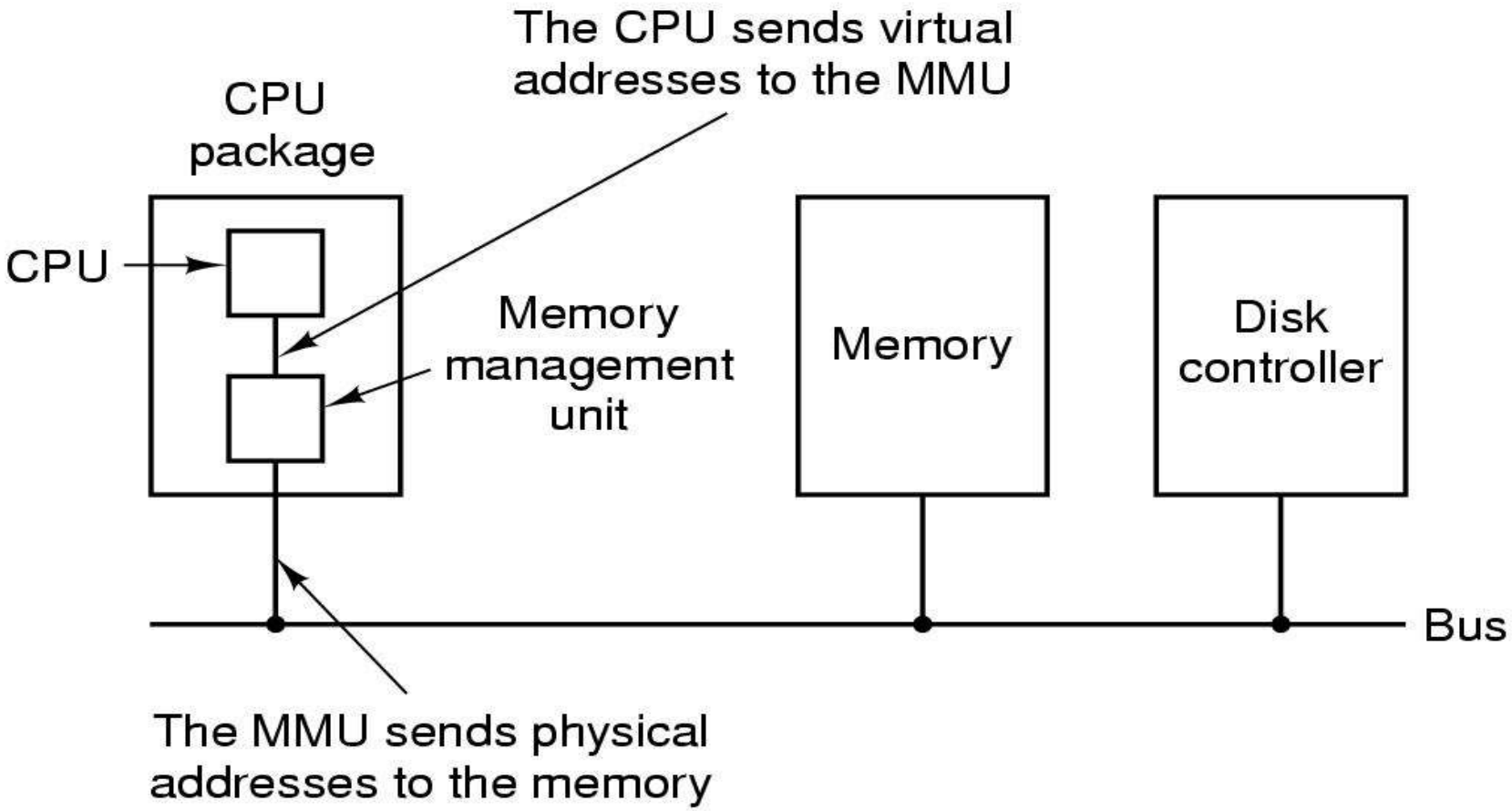


Example: 24 bit physical address
Frame size = 2^{12} = 4KB
Max physical memory size = 2^{24} bytes = 16MB

Address Translation

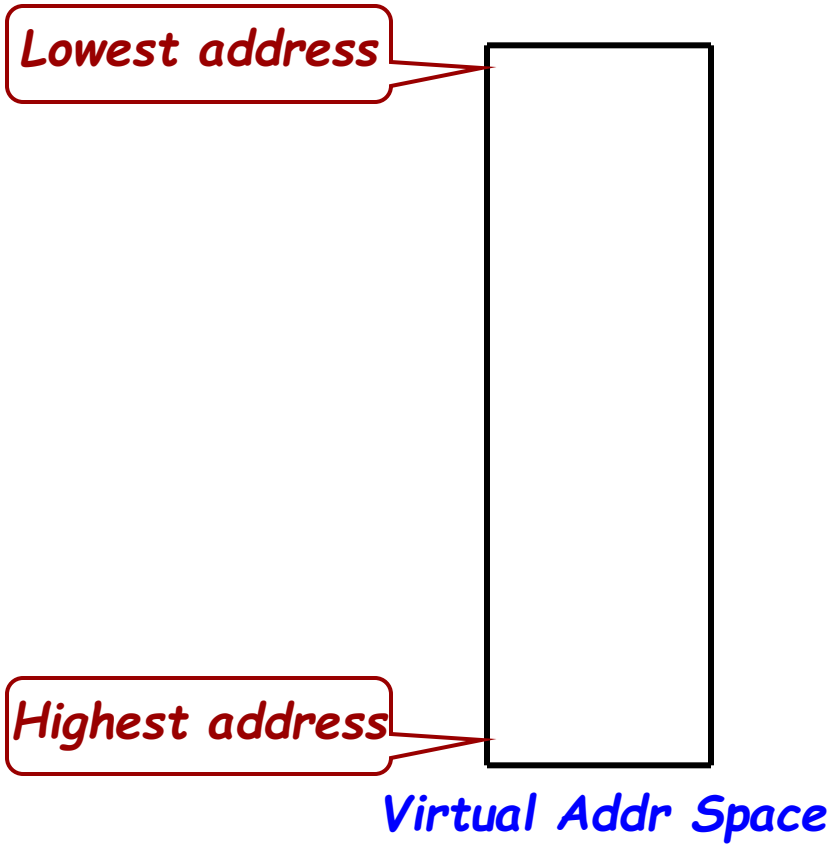
- Hardware maps page numbers to frame numbers
- Memory management unit (MMU) has multiple registers for multiple pages
  - *Like a base register except its value is substituted for the page number rather than added to it*
  - *Why don't we need a limit register for each page?*

Memory Management Unit (MMU)



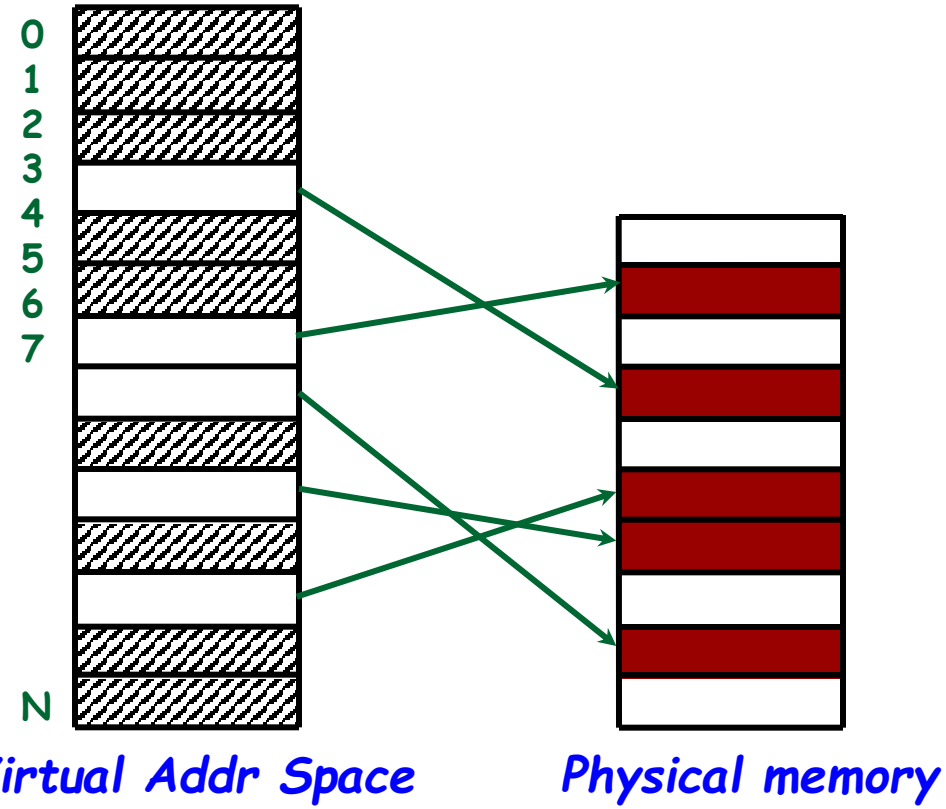
Virtual Address Spaces

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



Page Tables

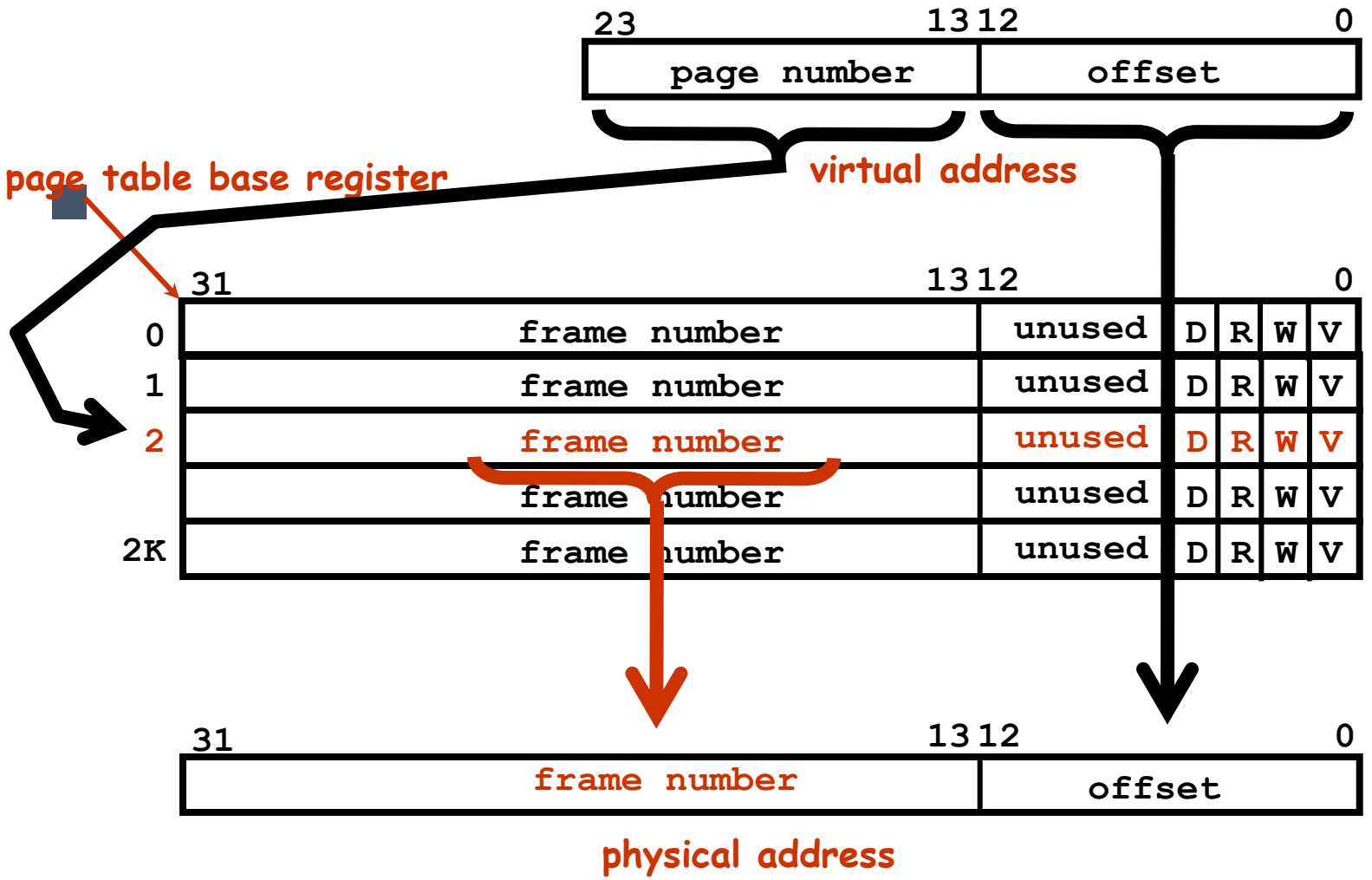
- Address mappings are stored in a *page table* in memory
- 1 entry/page: is page in memory? If so, which frame is it in?



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)

The BLITZ Page Table



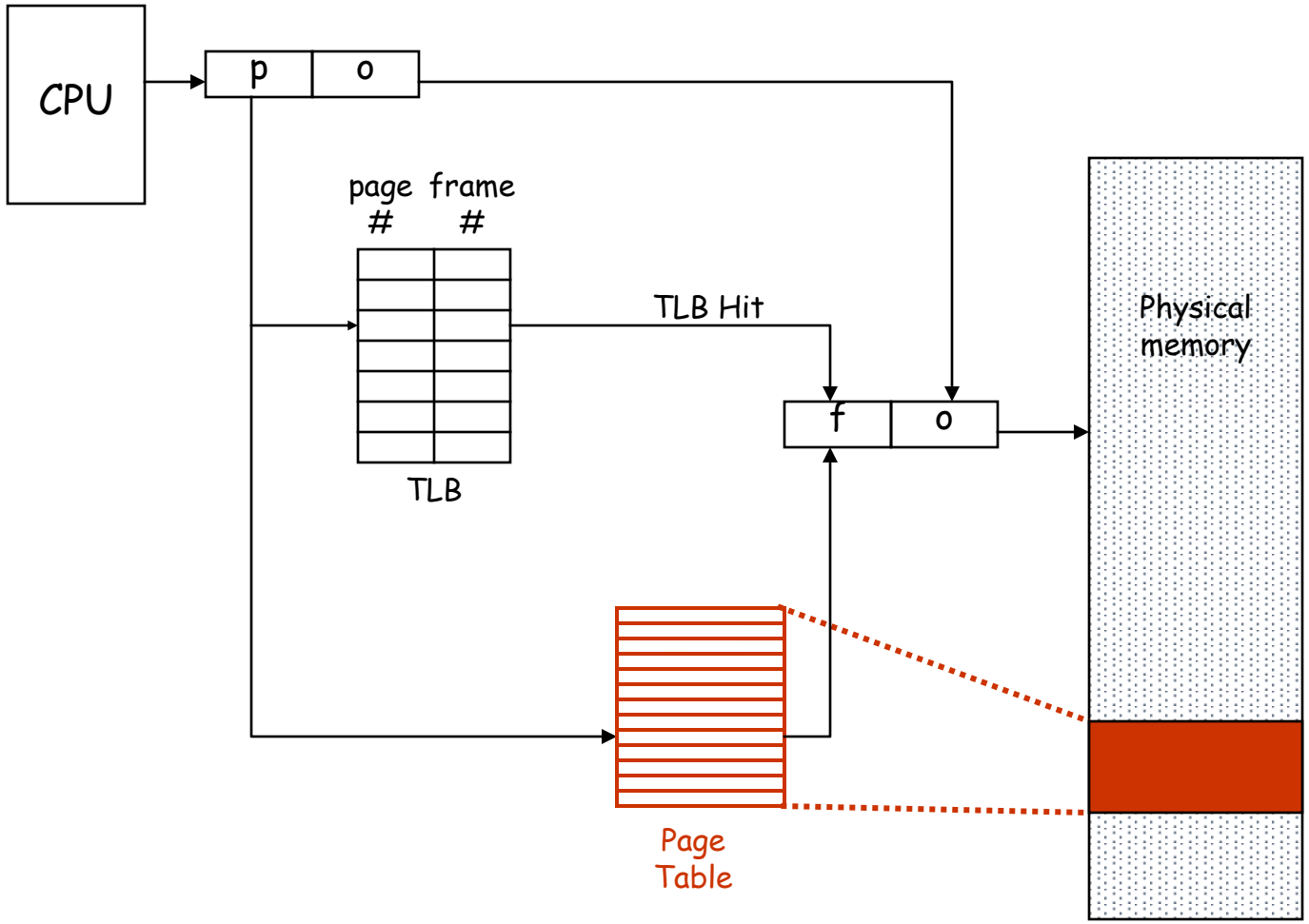
Translation Lookaside Buffer

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

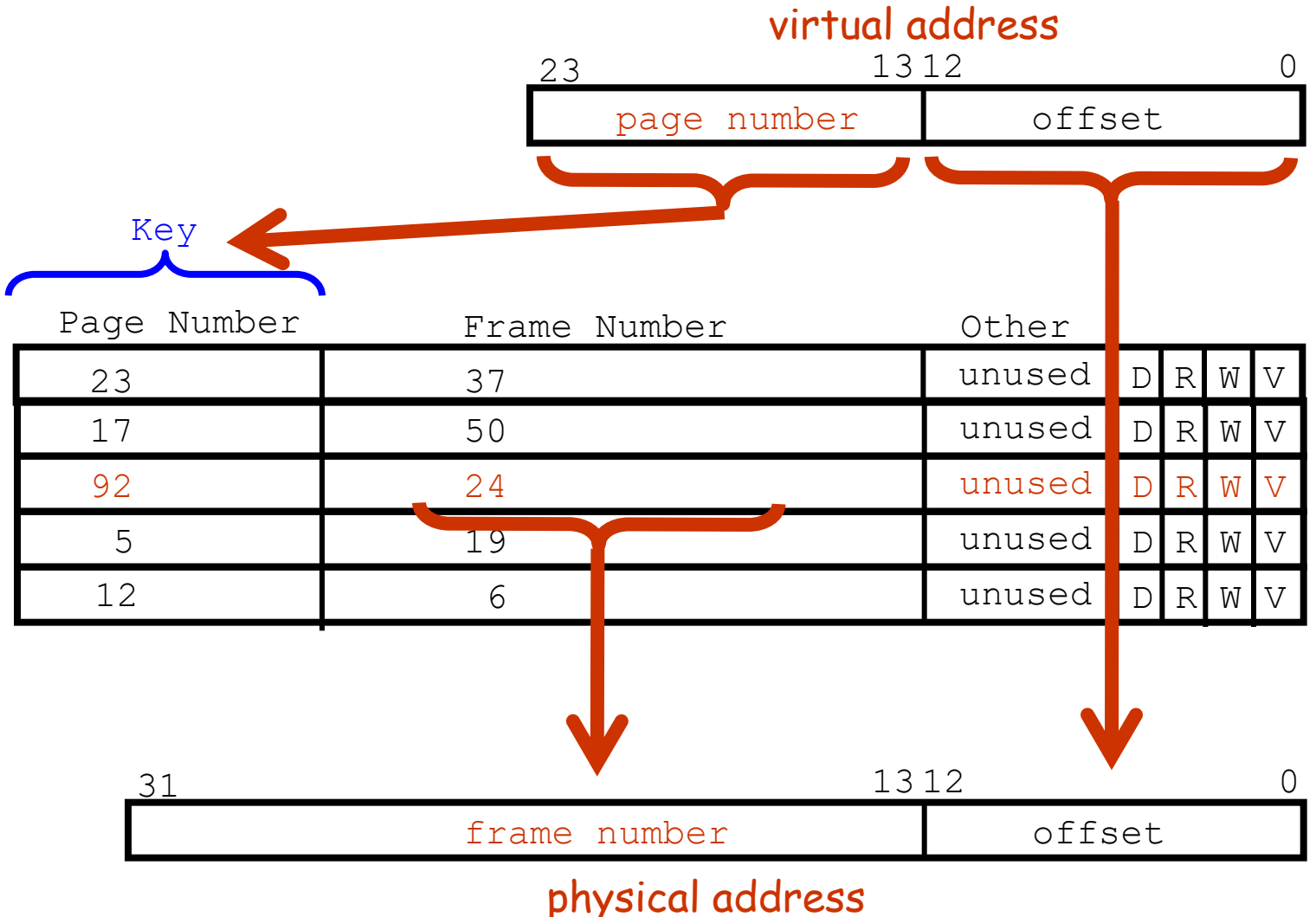
Translation Lookaside Buffer

- 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



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*

جلسه‌ی جدید

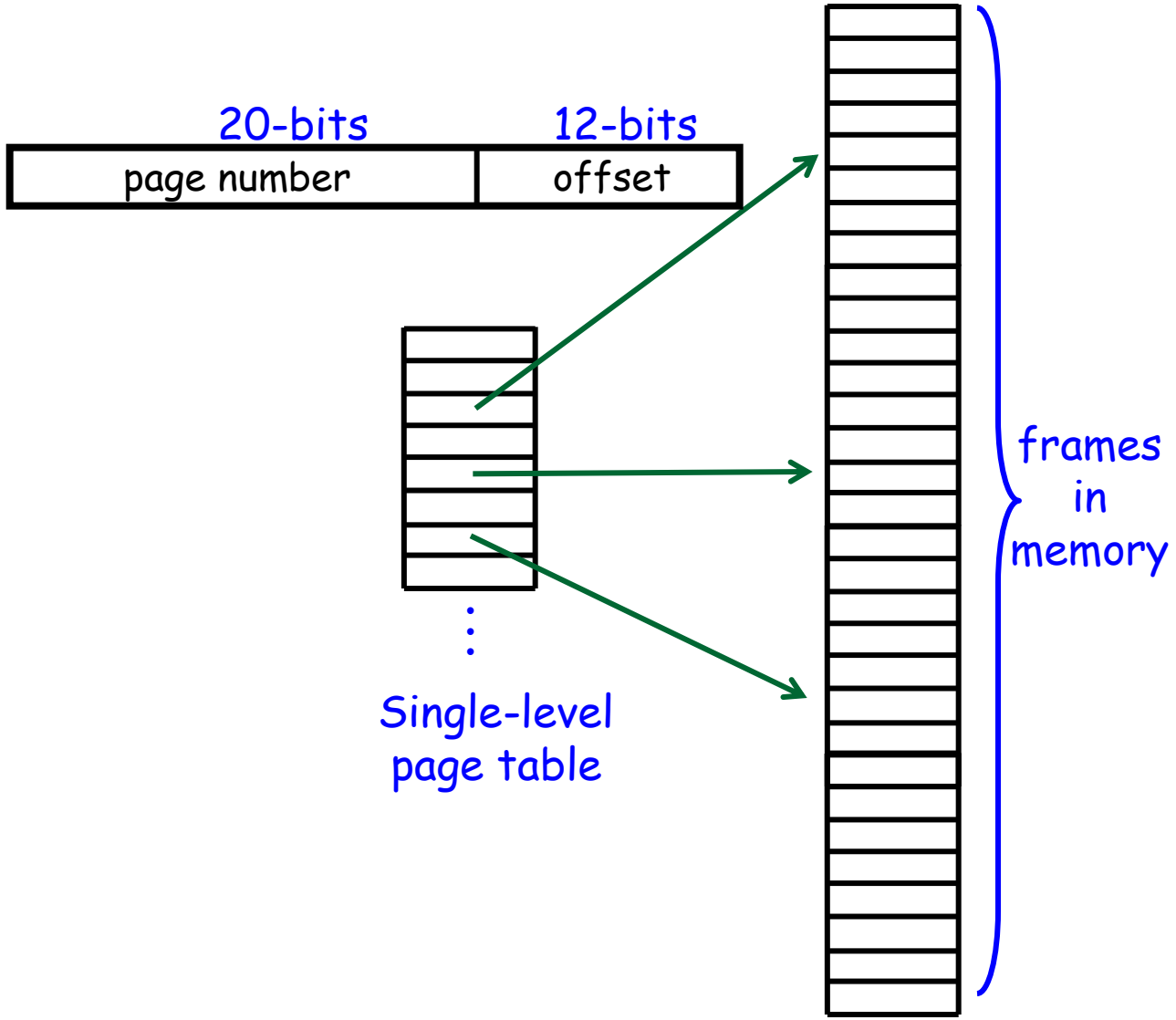
جدول صفحه – طراحی جدول صفحه، محافظت از حافظه

طراحی جدول صفحه

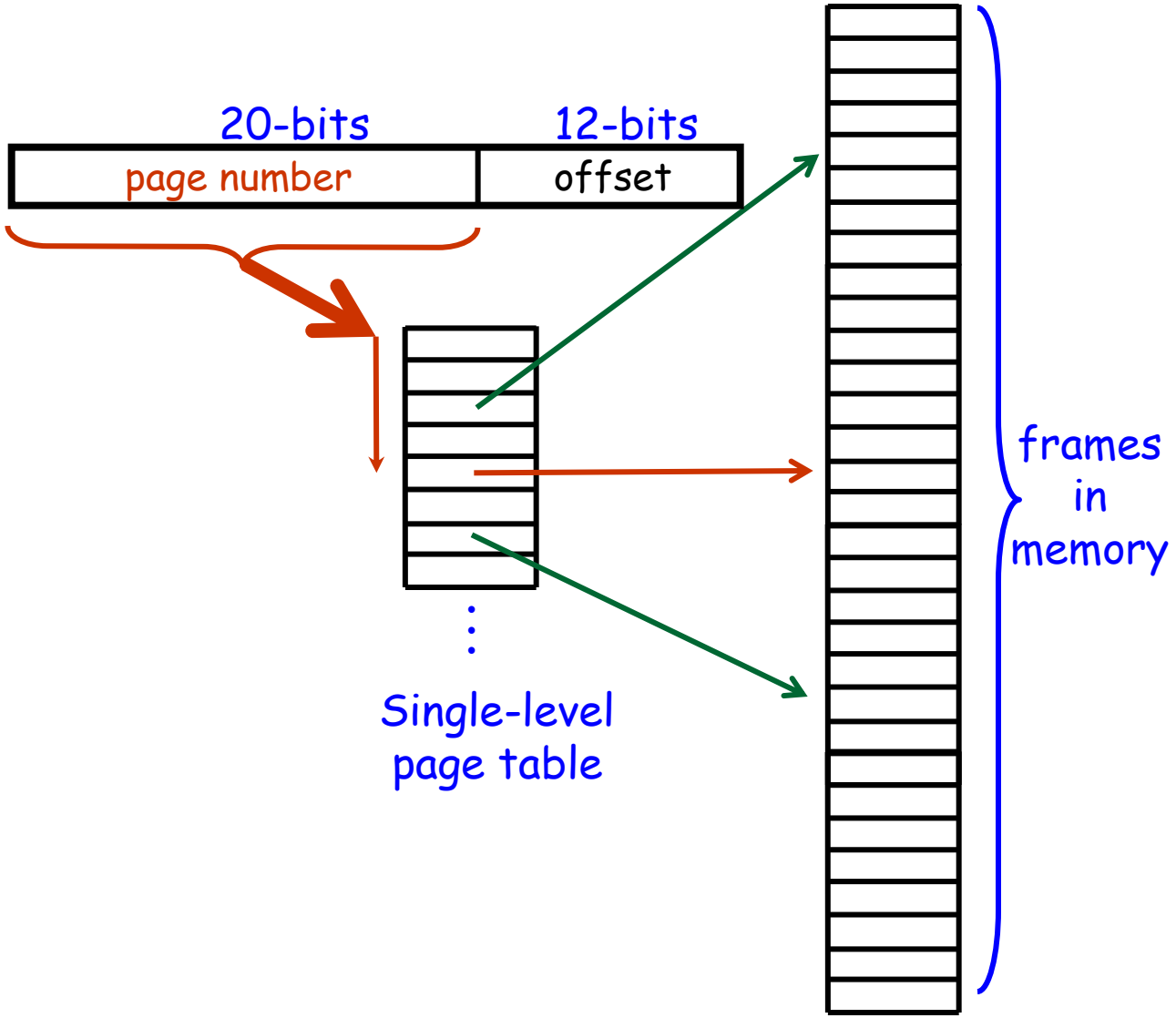
Page Table Design

- 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*

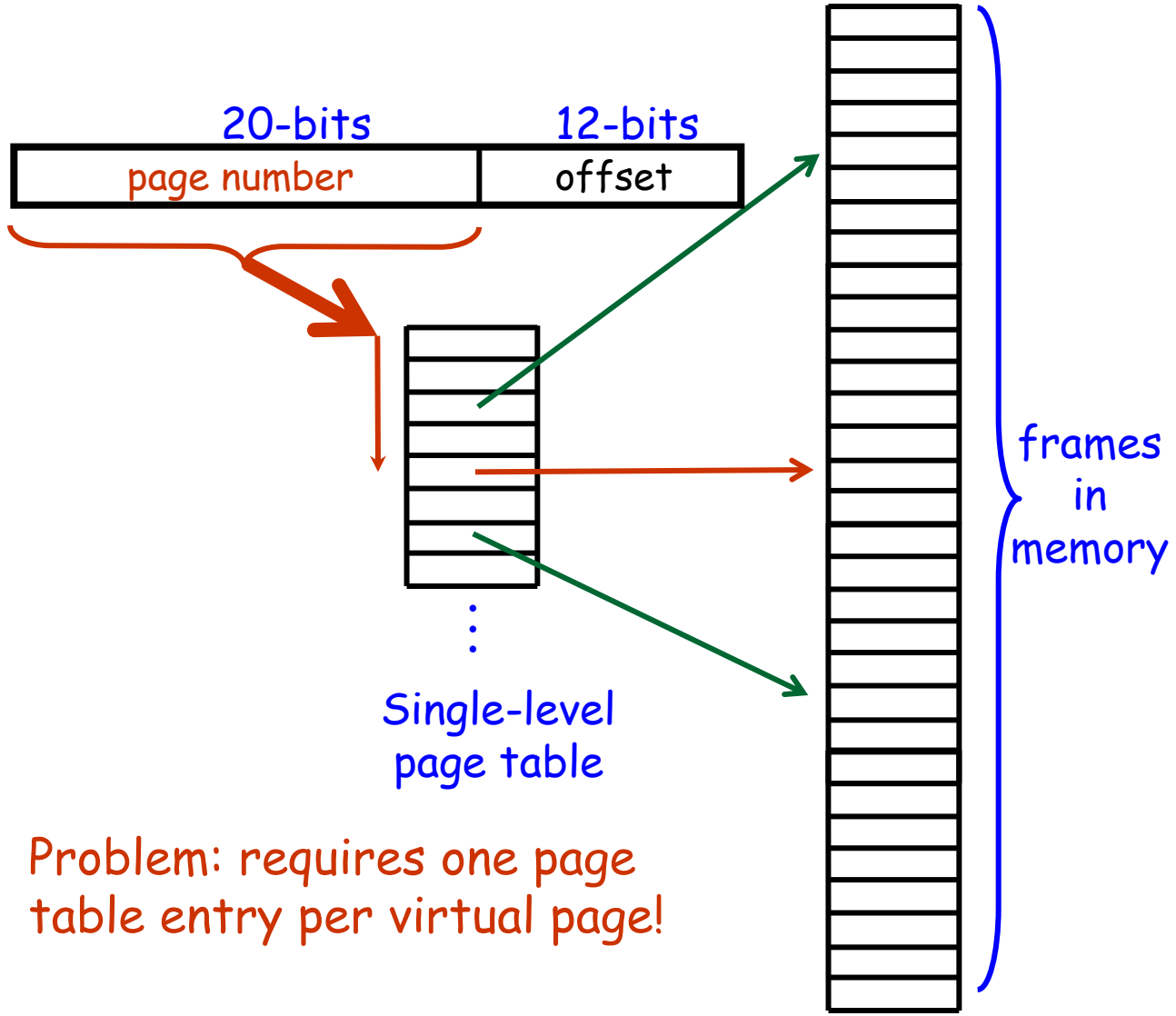
Single-Level Page Tables



Single-Level Page Tables

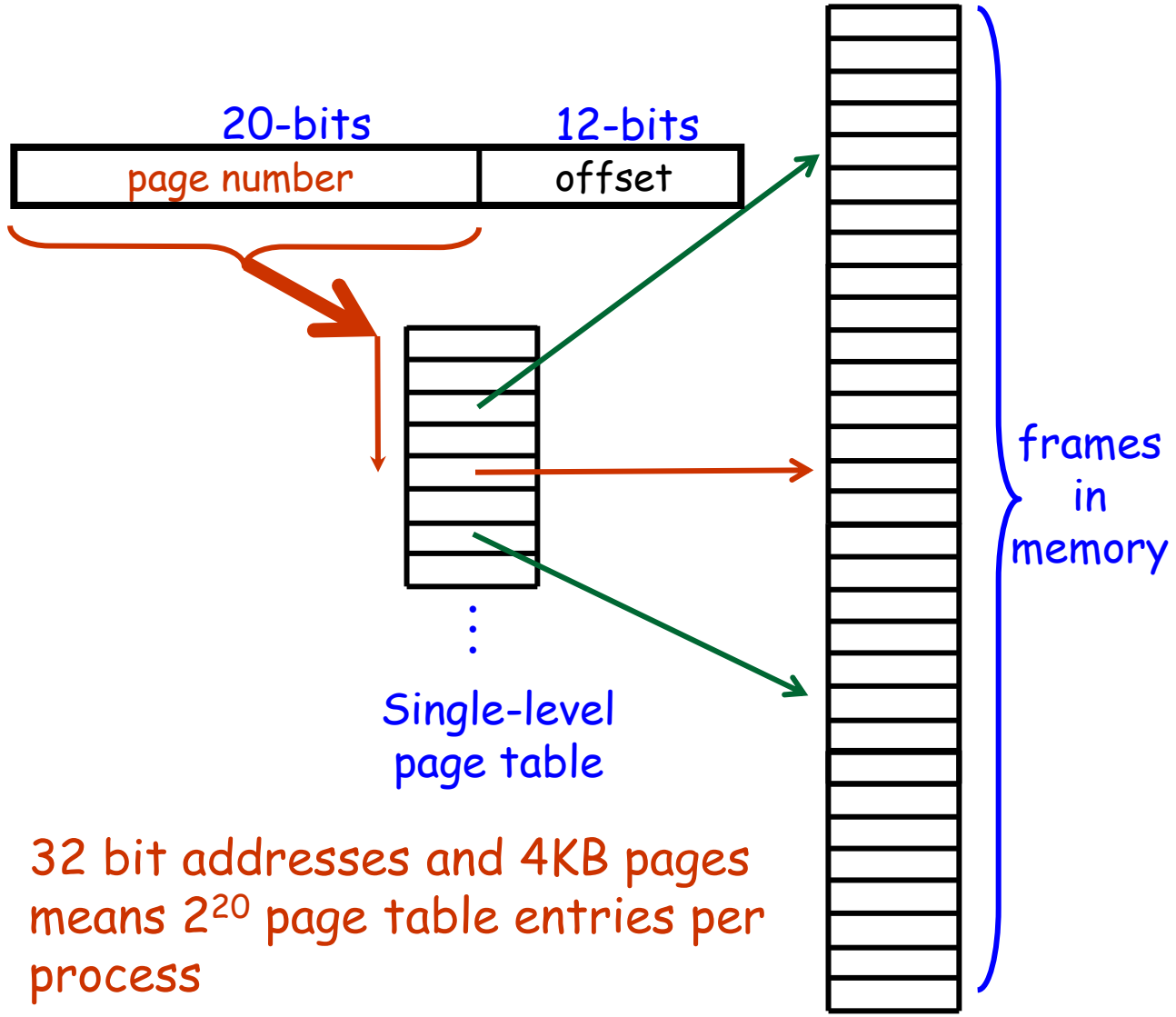


Single-Level Page Tables



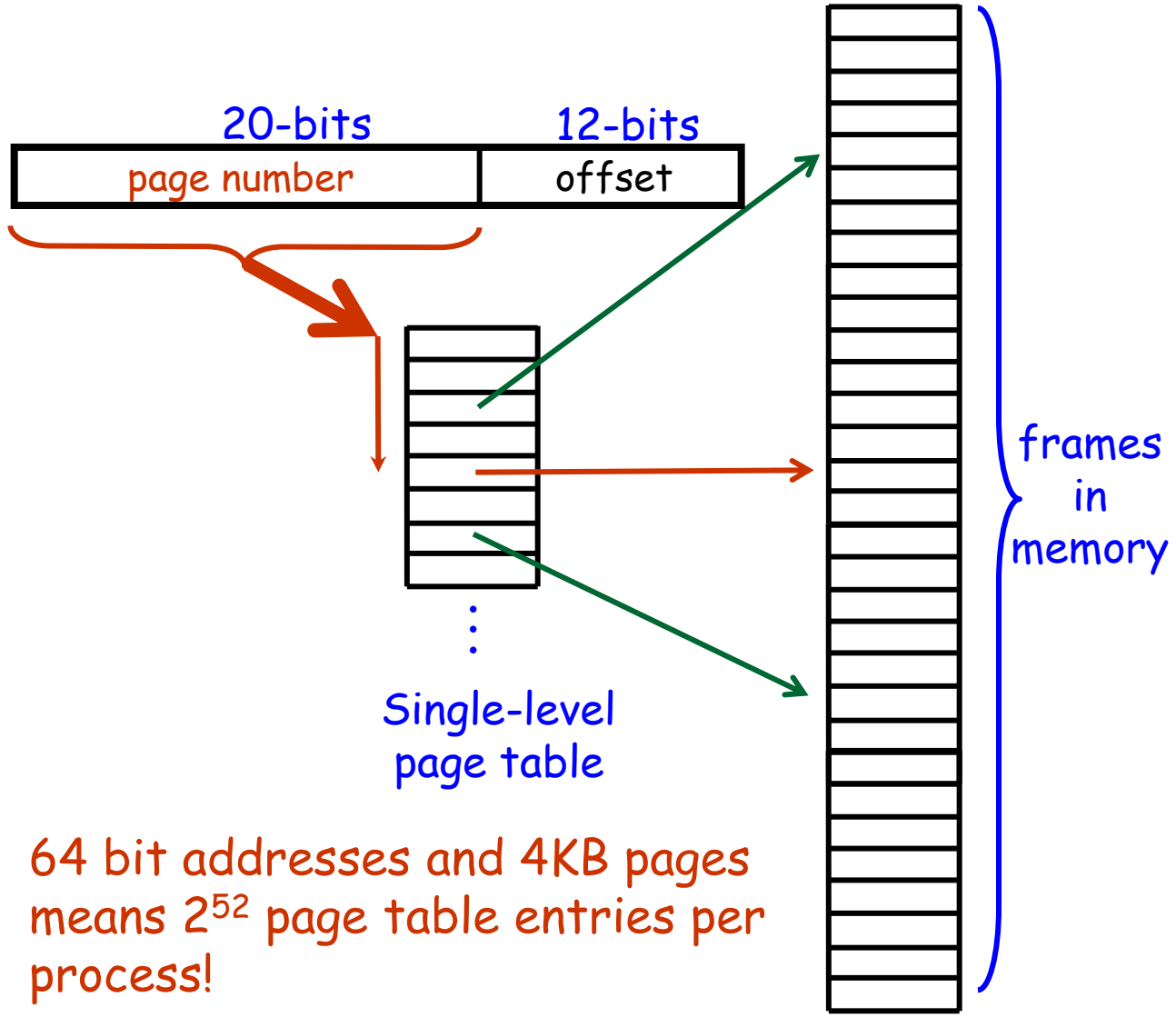
Problem: requires one page table entry per virtual page!

Single-Level Page Tables



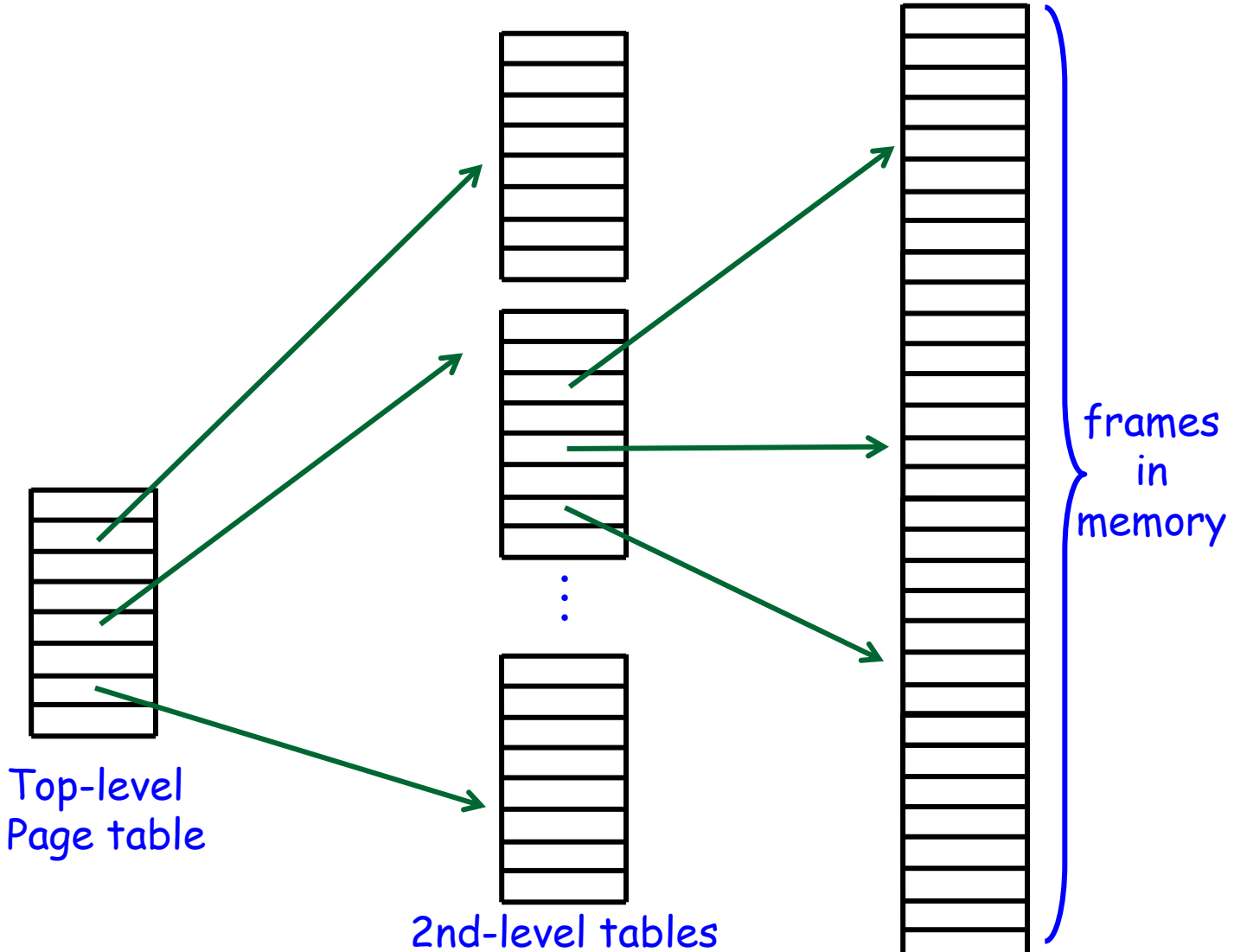
32 bit addresses and 4KB pages means 2^{20} page table entries per process

Single-Level Page Tables

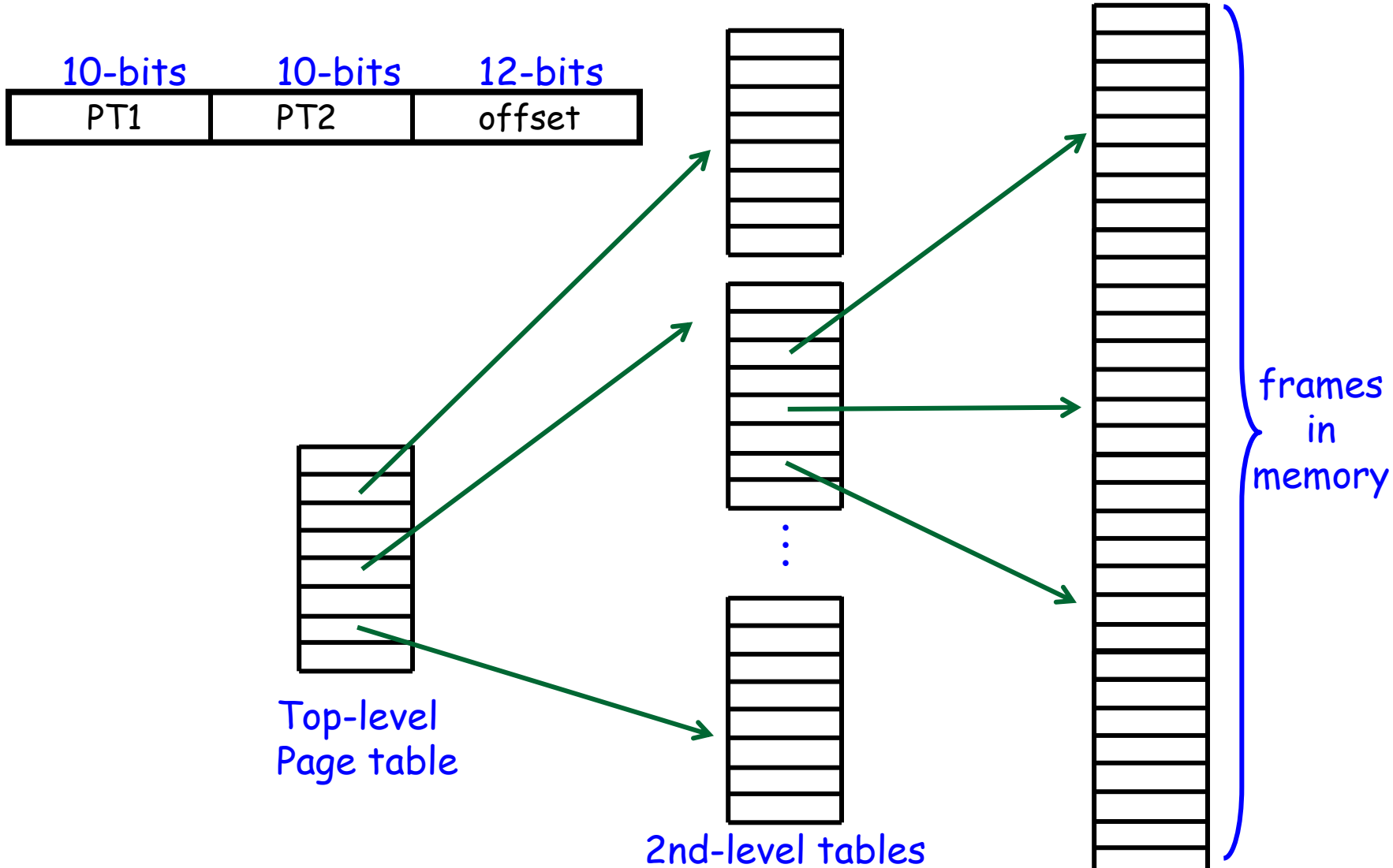


64 bit addresses and 4KB pages means 2^{52} page table entries per process!

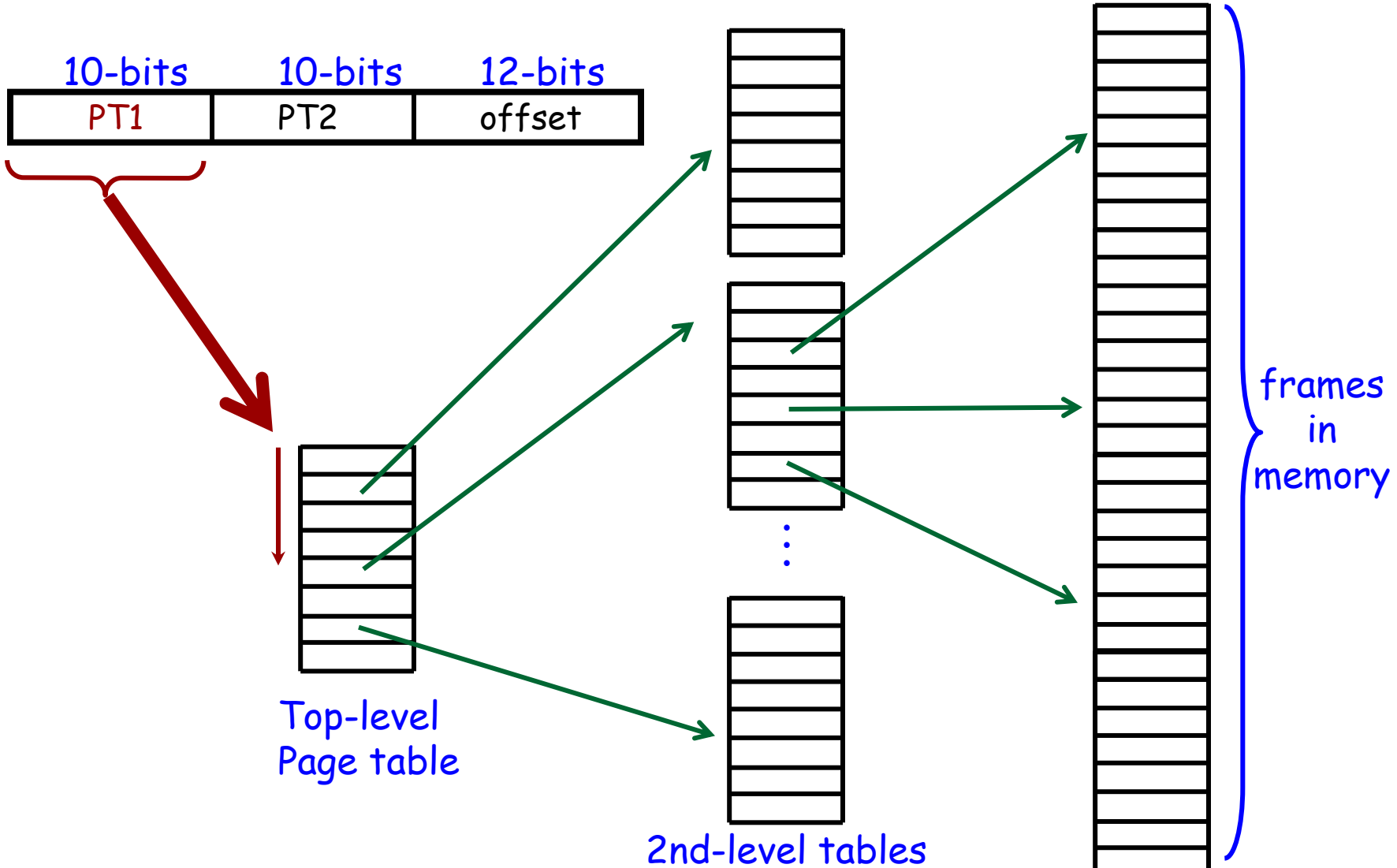
Multi-Level Page Tables



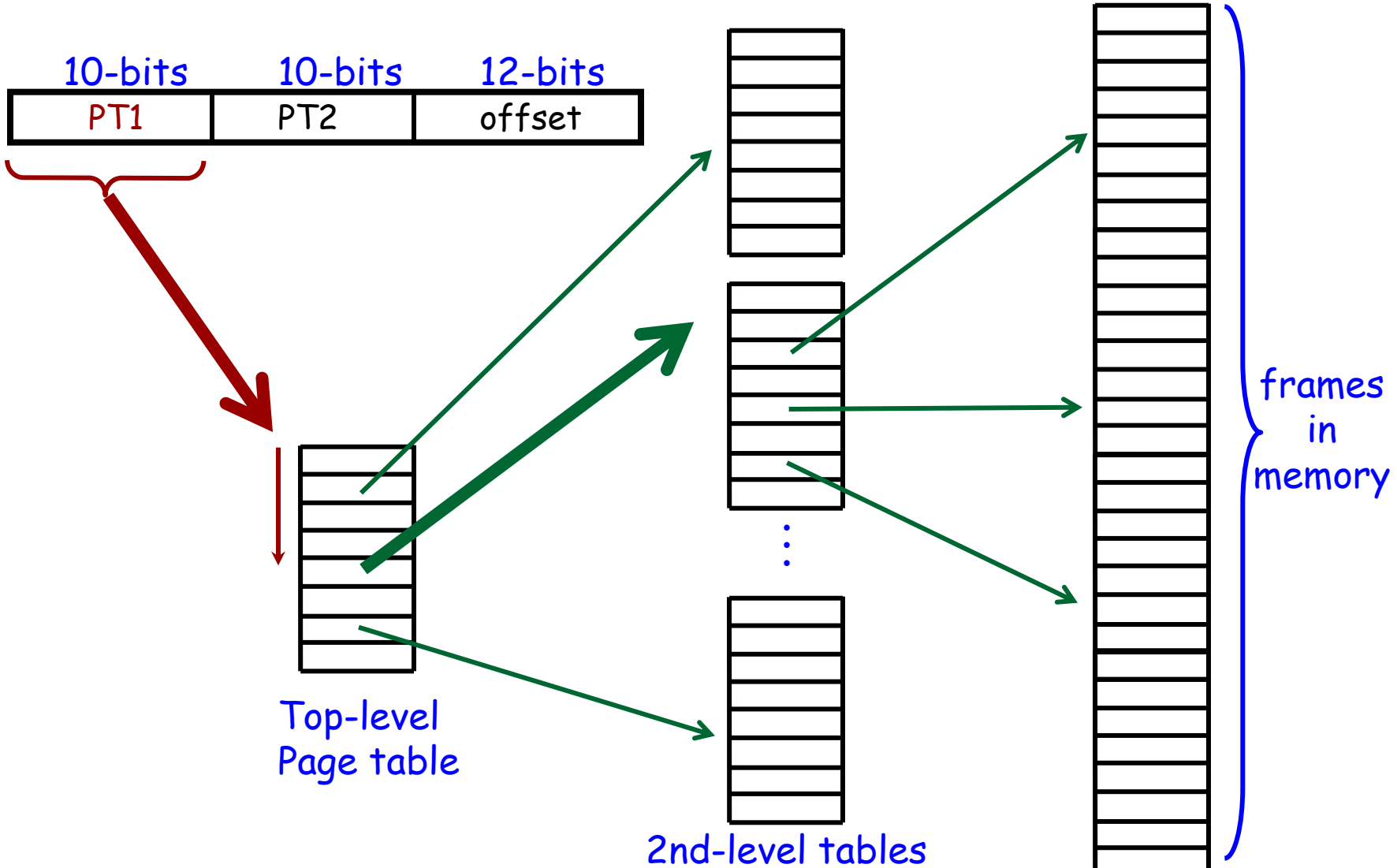
Multi-Level Page Tables



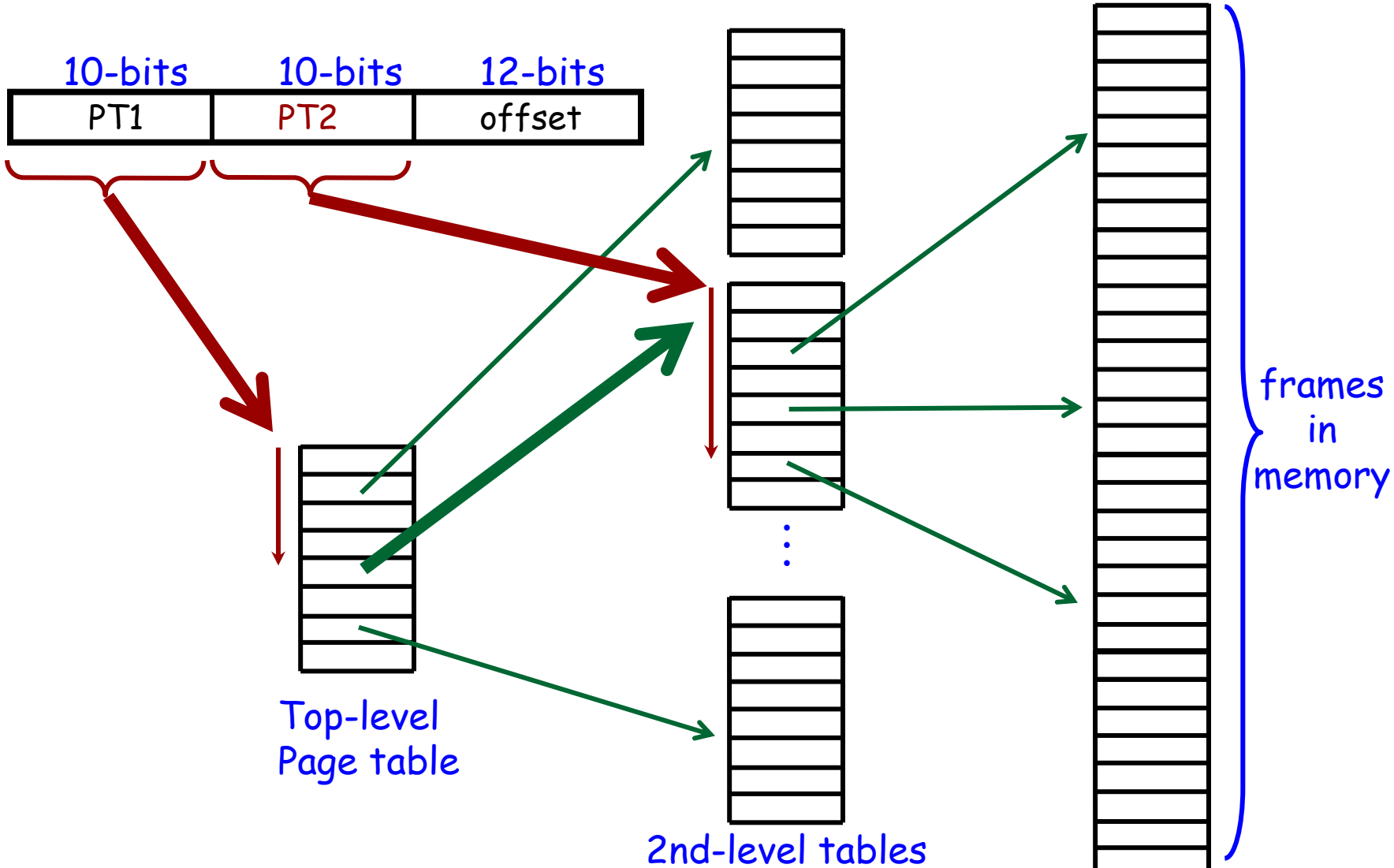
Multi-Level Page Tables



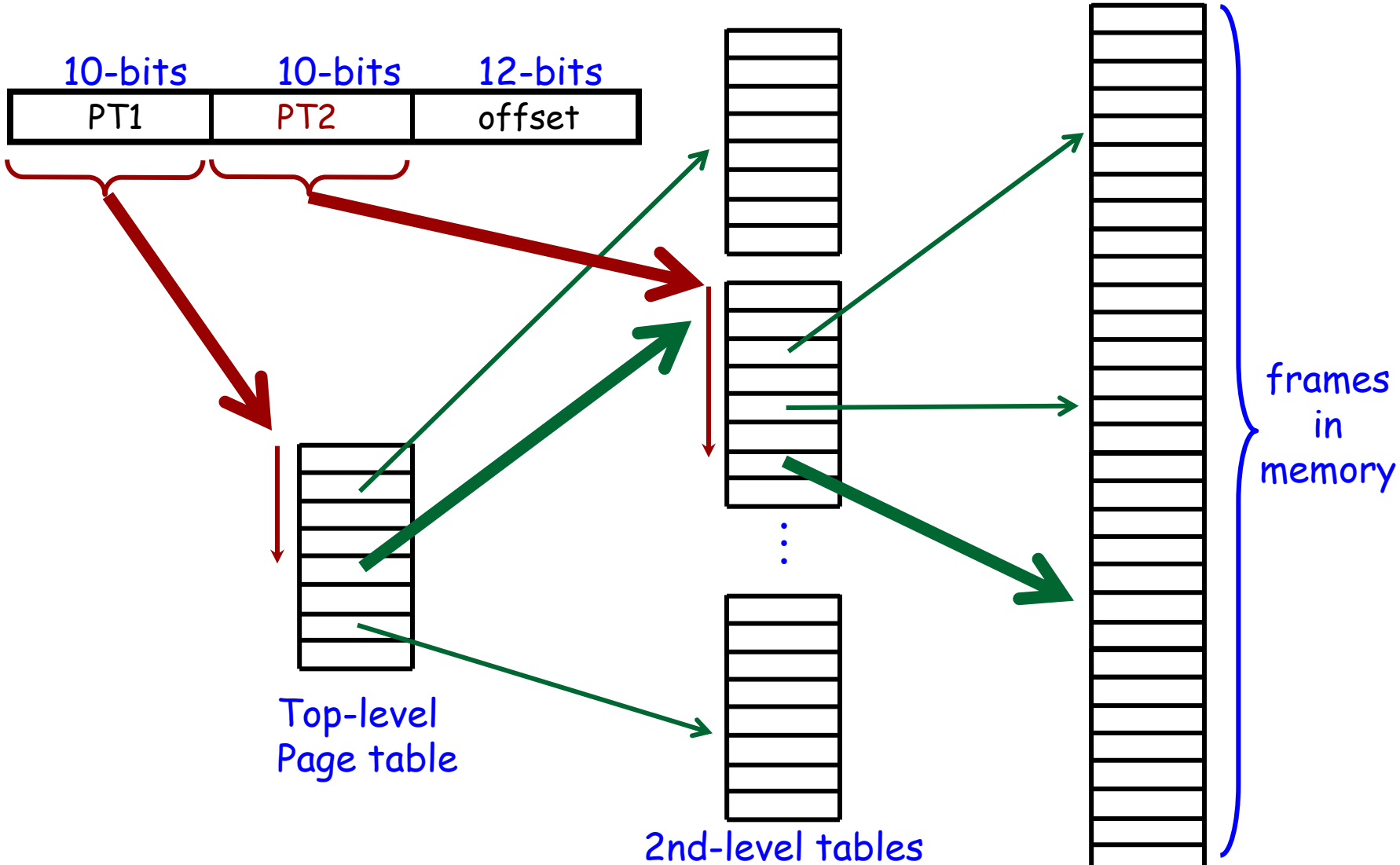
Multi-Level Page Tables



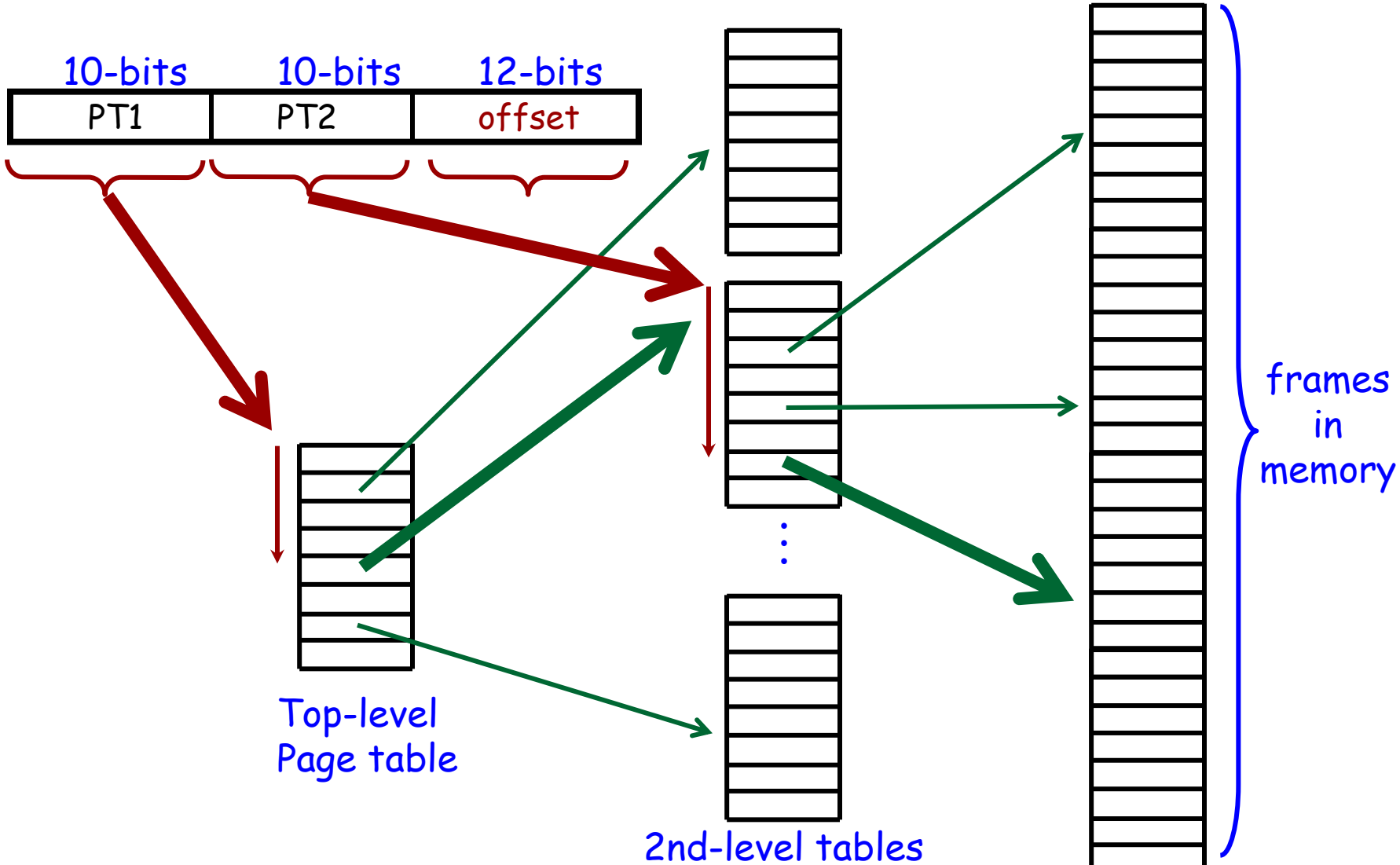
Multi-Level Page Tables



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!*

Inverted Page Tables

- Problem:
  - *Page table overhead increases with address space size*
  - *Page tables get too big to fit in memory!*
- Consider a computer with 64 bit addresses
  - *Assume 4 Kbyte pages (12 bits for the offset)*
  - *Virtual address space = 2^{52} pages!*
  - *Page table needs 2^{52} entries!*
  - *This page table is much too large for memory!*

Inverted Page Tables

- How many mappings do we need (maximum) at any time?
- We only need mappings for pages that are in memory!

Inverted Page Tables

- An inverted page table
  - *Has one entry for every resident memory page*
  - *Roughly speaking, one for each frame of memory*
  - *Records which page is in that frame*
  - *Can not be indexed by page number*
- So how can we search an inverted page table on a TLB miss fault?

Inverted Page Tables

- Given a page number (from a faulting address), do we exhaustively search all entries to find its mapping?

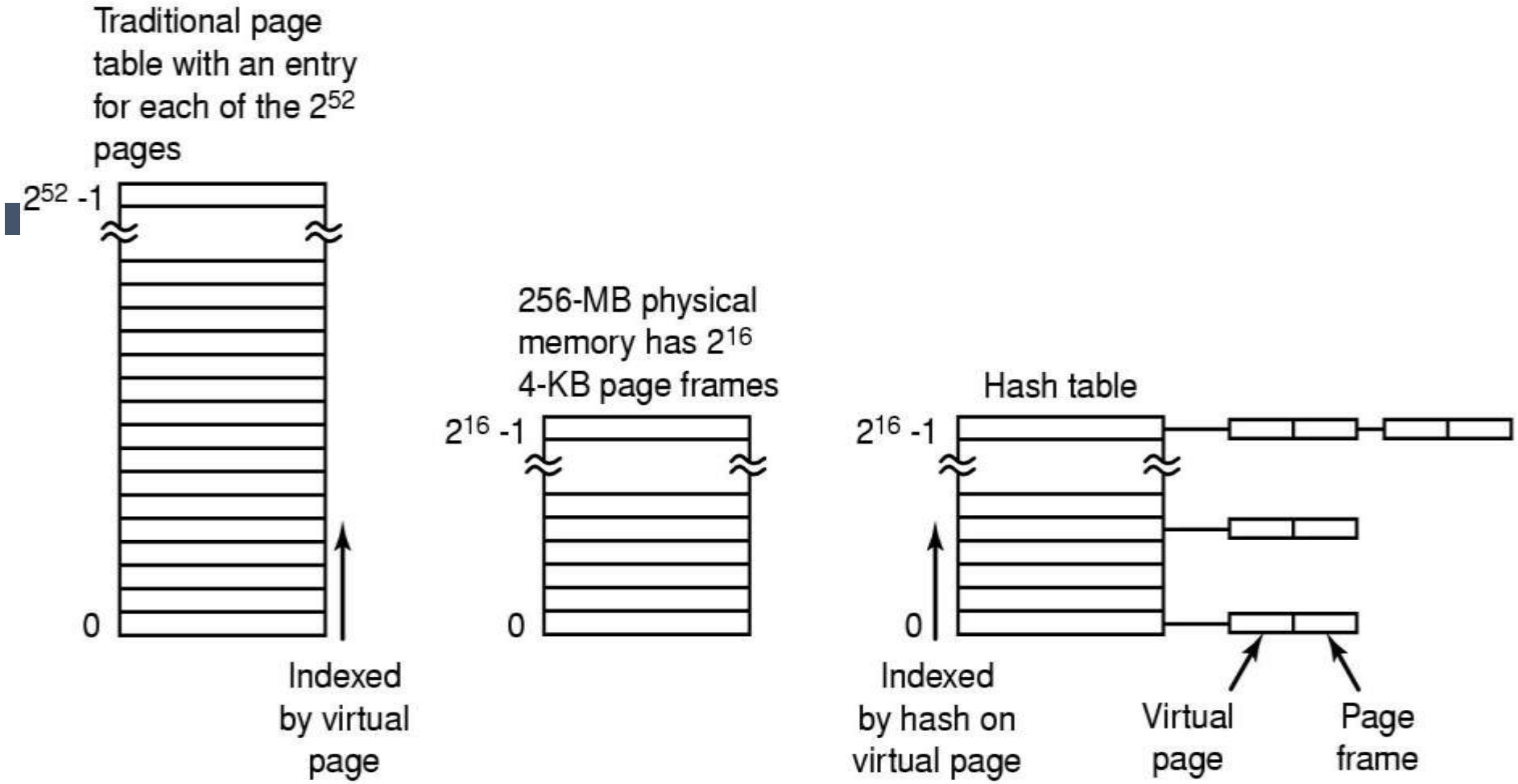
Inverted Page Tables

- Given a page number (from a faulting address), do we exhaustively search all entries to find its mapping?
  - *No, that's too slow!*
  - *A hash table could allow fast access given a page number*
  - *O(1) lookup time with a good hash function*

Hash Tables

- Data structure for associating a key with a value
  - *Apply hash function to key to produce a hash*
  - *Hash is a number that is used as an array index*
  - *Each element of the array can be a linked list of entries (to handle collisions)*
  - *The list must be searched to find the required entry for the key (entry's key matches the search key)*
  - *With a good hash function the average list length will be short*

Inverted page table



Which Page Table Design is Best?

- The best choice depends on CPU architecture
- 64 bit systems need inverted page tables
- Some systems use a combination of regular page tables together with segmentation (later)

محافظت از حافظه

Memory Protection

- Protection though addressability
  - *If address translation only allows a process to access its own pages, it is implementing memory protection*
- But what if you want a process to be able to read and execute some pages but not write them?
  - *eg. the text segment*
- Or read and write them but not execute them?
  - *eg. the stack*
- Can we implement protection based on access type?

Memory Protection

- How is protection checking implemented?
  - *Compare page protection bits with process capabilities and operation types on every load/store*
  - *That sounds expensive!*
  - *Requires hardware support!*
- How can protection checking be done efficiently?
  - *Use the TLB as a "protection" look-aside buffer as well as a translation lookaside buffer*
  - *Use special segment registers*

Protection Lookaside Buffer

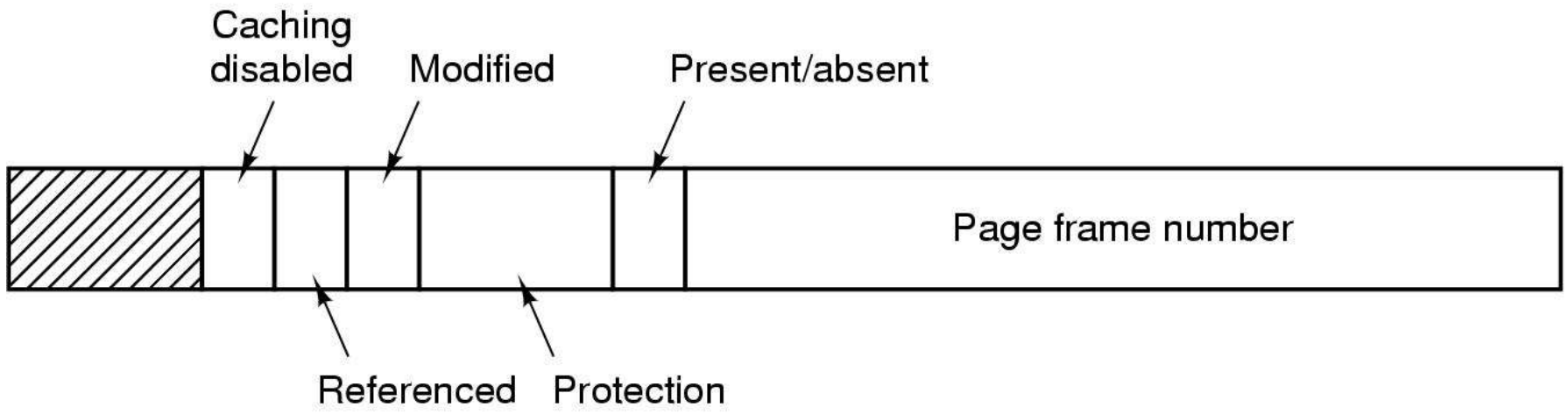
- A TLB is often used for more than just “translation”
- Memory accesses need to be checked for validity
  - *Does the address refer to an allocated segment of the address space?*
    - *If not: segmentation fault!*
  - *Is this process allowed to access this memory segment?*
    - *If not: segmentation/protection fault!*
  - *Is the type of access valid for this segment?*
    - Read, write, execute …?
    - *If not: protection fault!*

Protection Checking With a TLB

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Page Grain Protection

A typical page table entry with support for page grain protection



Memory Protection Granularity

- At what granularity should protection be implemented?
- Page-level?
  - *A lot of overhead for storing protection information for non-resident pages*
- Segment level?
  - *Coarser grain than pages*
  - *Makes sense if contiguous groups of pages share the same protection status*

Segment-Granularity Protection

- All pages within a segment usually share the same protection status
  - *So we should be able to batch the protection information*
- Then why not just use segment-size pages?

Segment-Granularity Protection

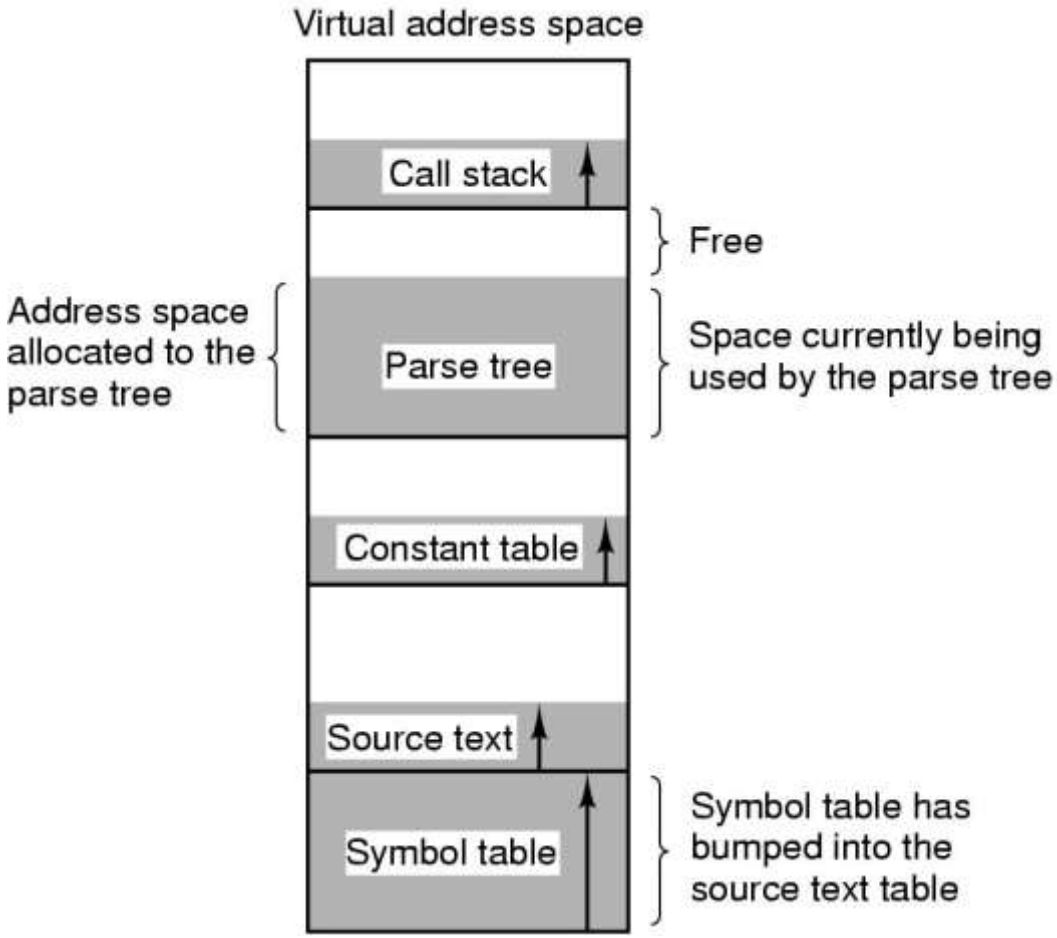
- Segments vary in size from process to process
- Segments change size dynamically (stack, heap)
- Need to manage addressability, access-based protection and memory allocation separately
- Coarse-grain protection can be implemented simply through addressability

Segmented Address Spaces

- *Traditional Virtual Address Space*
  - *"flat" address space (1 dimensional)*
- *Segmented Address Space*
  - *Program made of several pieces or "segments"*
  - *Each segment is its own mini-address space*
  - *Addresses within a segment start at zero*
  - *The program must always say which segment it means*
    - either embed a segment id in an address
    - or load a value into a segment register and refer to the register
  - *Addresses:*
    - Segment + Offset
  - *Each segment can grow independently of others*

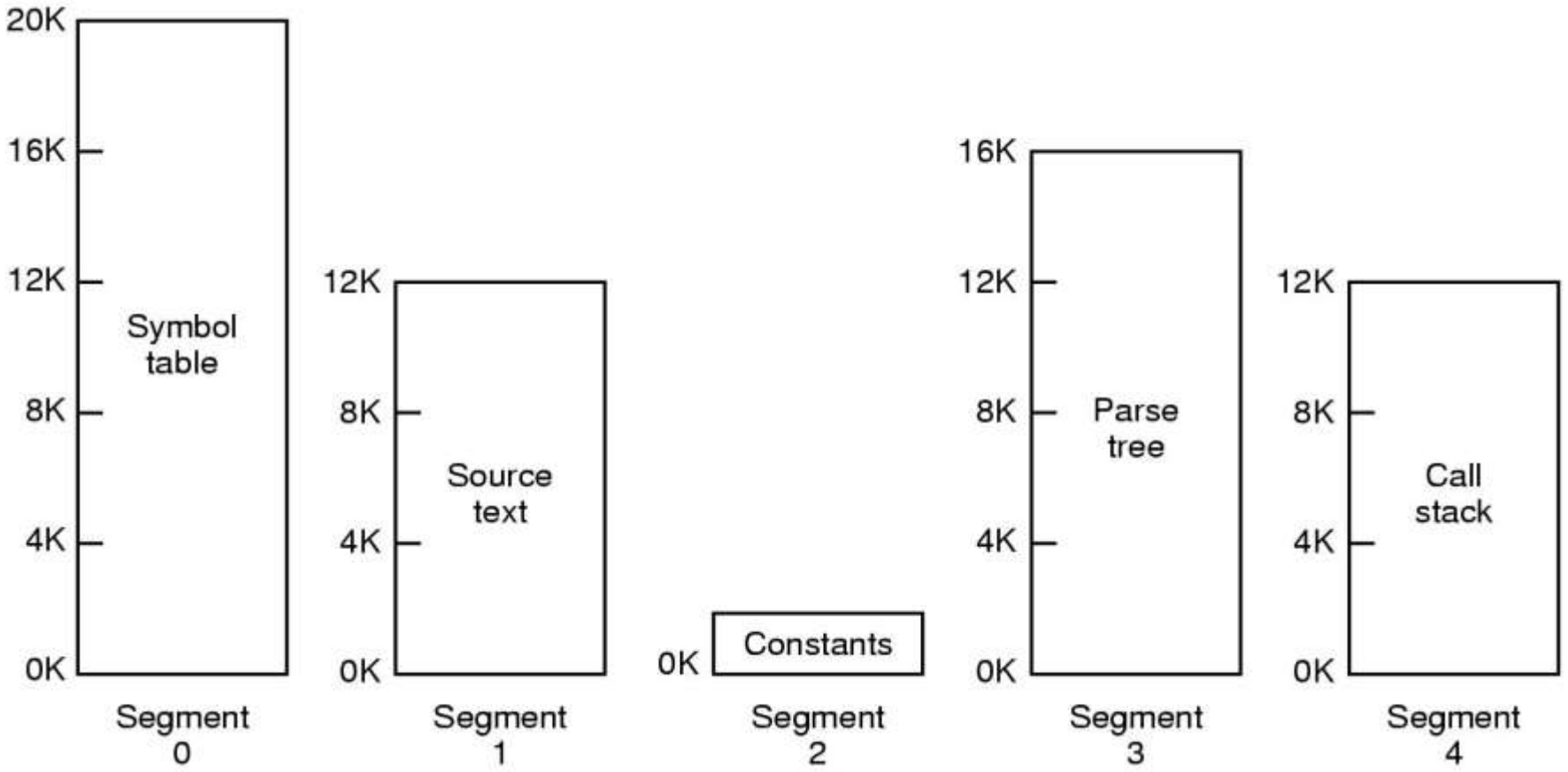
Segmented Address Spaces

Example: *A compiler*



Segmented Memory

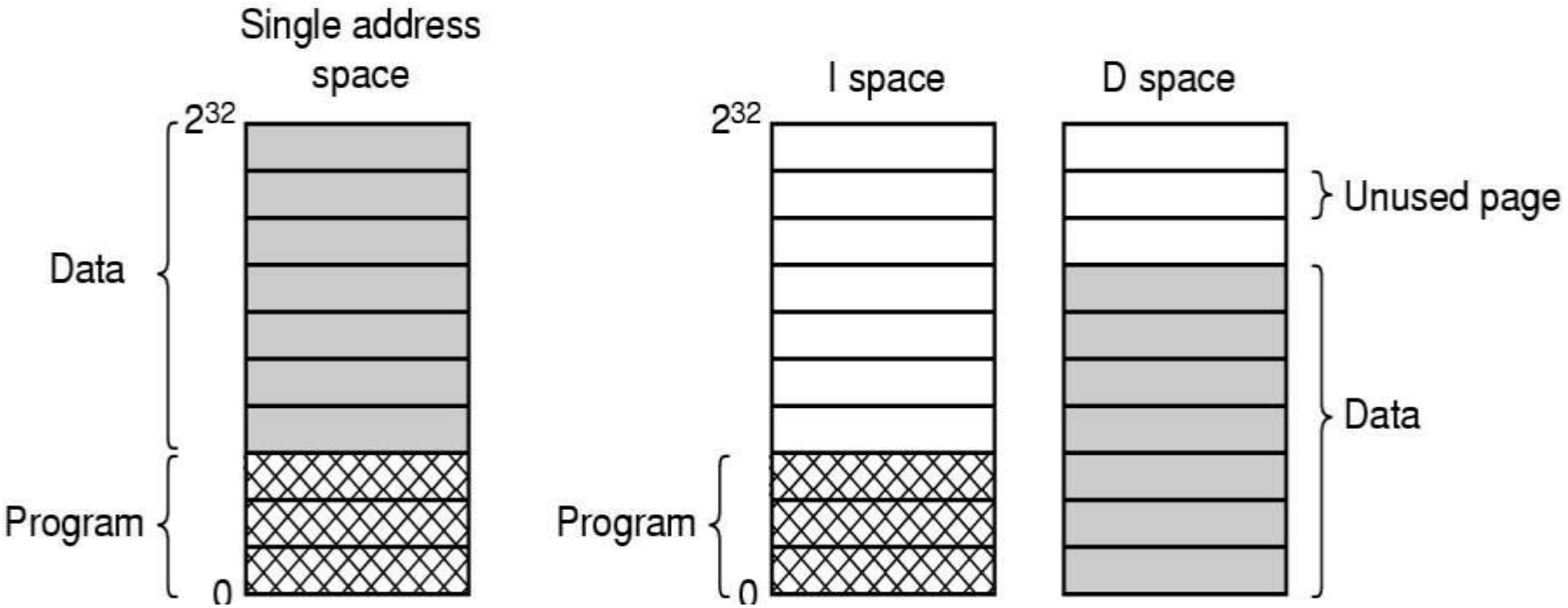
- Each space grows, shrinks independently!



Instruction and Data Spaces

* One address space

* Separate I and D spaces



Pure Paging vs. Pure Segmentation

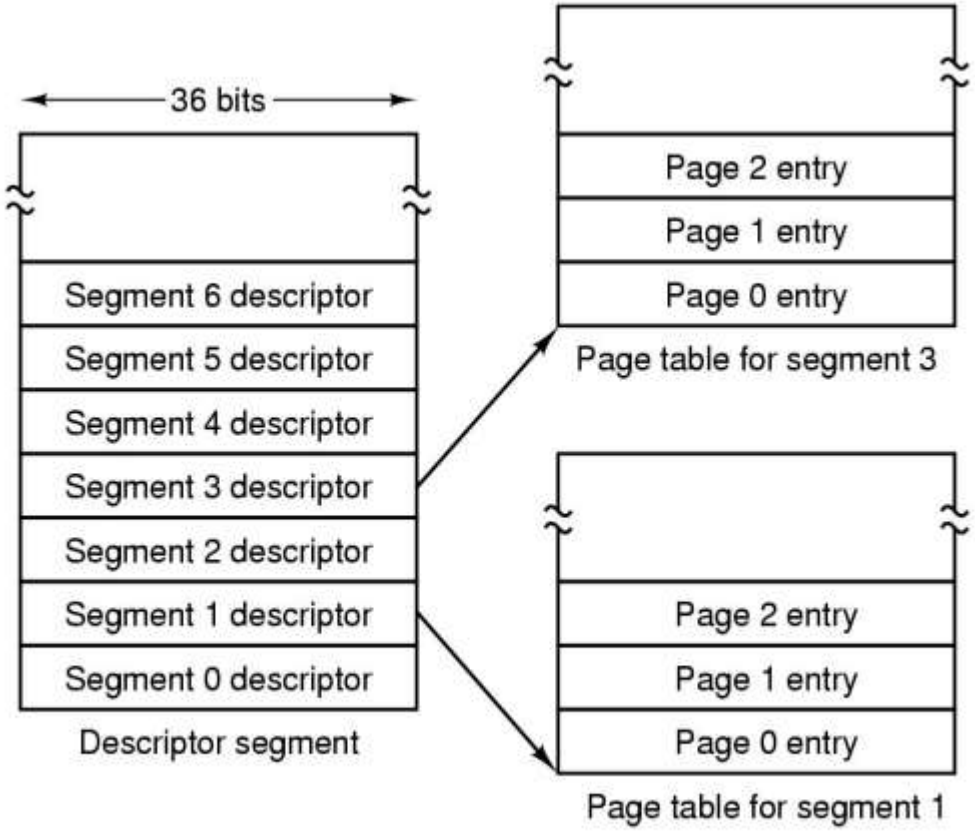
Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Segmentation vs. Paging

- Do we need to choose one or the other?
- Why not use both together
  - *Paged segments*
  - *Paging for memory allocation*
  - *Segmentation for maintaining protection information at a coarse granularity*
  - *Segmentation and paging in combination for translation*

Paged Segments in MULTICS

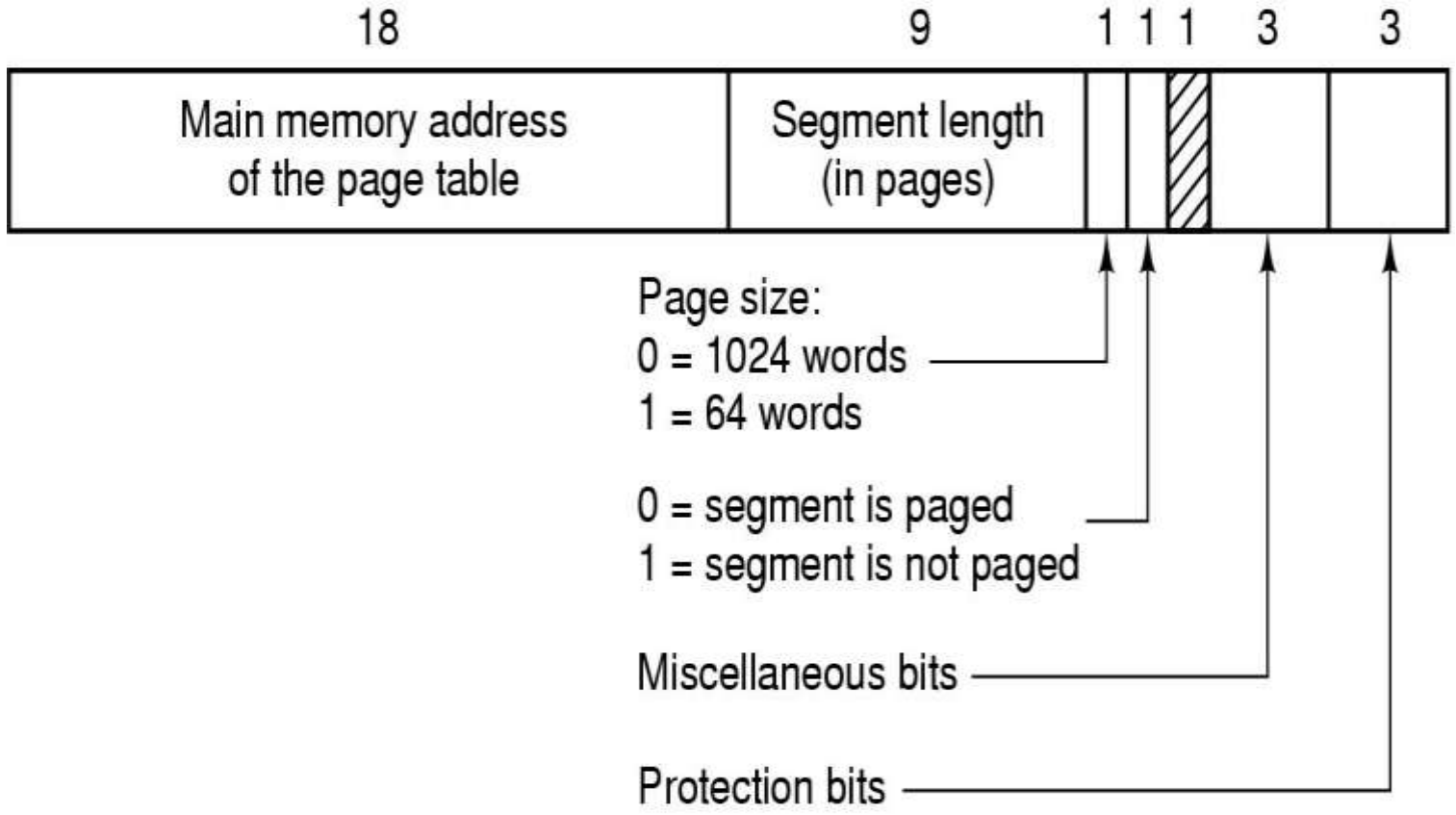
- Each segment is divided up into pages.
- Each segment descriptor points to a page table.



Paged Segments in MULTICS

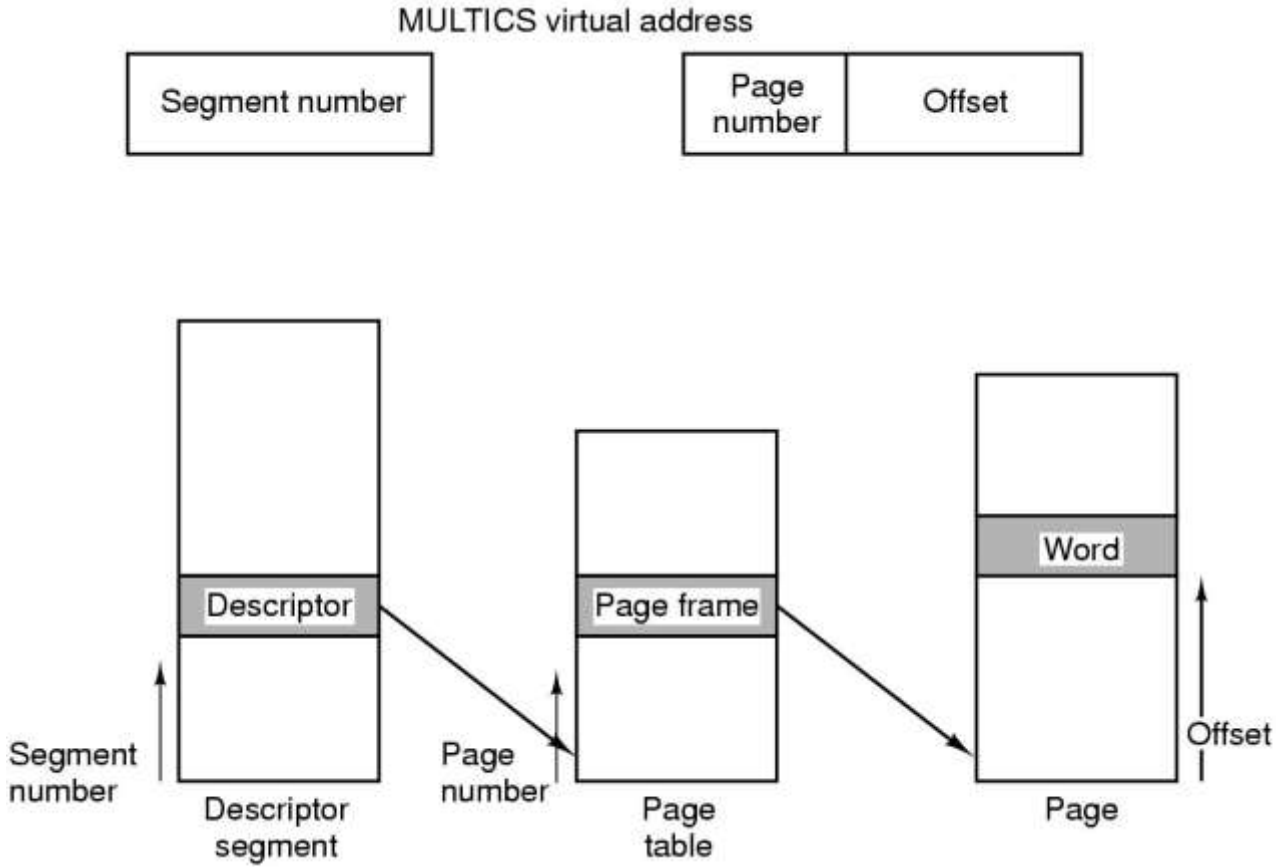
(Multiplexed Information and Computing Service)

- Each entry in segment table



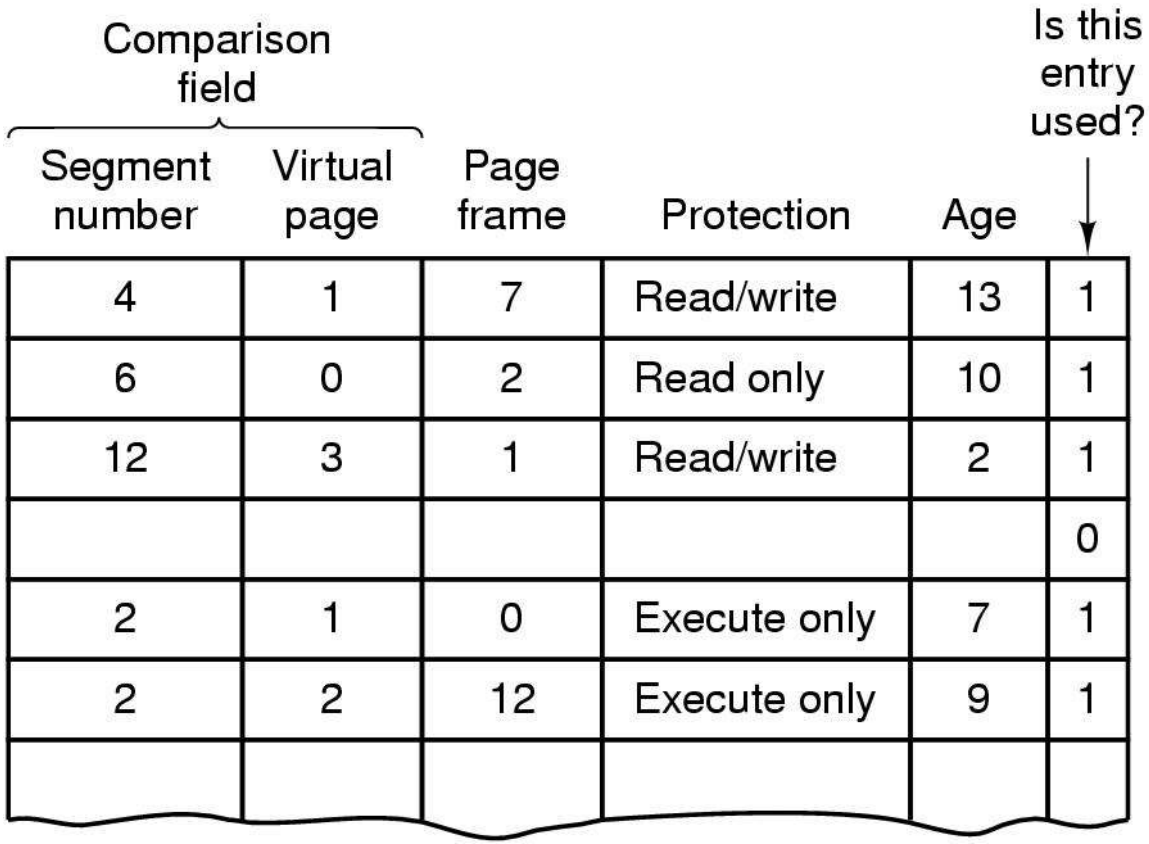
Paged Segments in MULTICS

Conversion of a 2-part MULTICS address into a main memory address



The MULTICS TLB

- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated



Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

PAGE TABLE IN MODERN OSES

مثلا در لینوکس

Memory Management in Linux

- 64-bit addressable space, with only 48 bits currently used.
- No segmentation, with protection handled at the page level.
- Separate user and kernel spaces through negative addressing.
- Support for variable page sizes (4 KB, 2 MB, 1 GB).
- A 4-level page table hierarchy.

جلسه‌ی بعد

مسئله‌ی Page Sharing, Page Fault