

Operating Systems

Lecture 6

Address Translation

Prof. Mengwei Xu



Recap of Last Course

Goals for Today

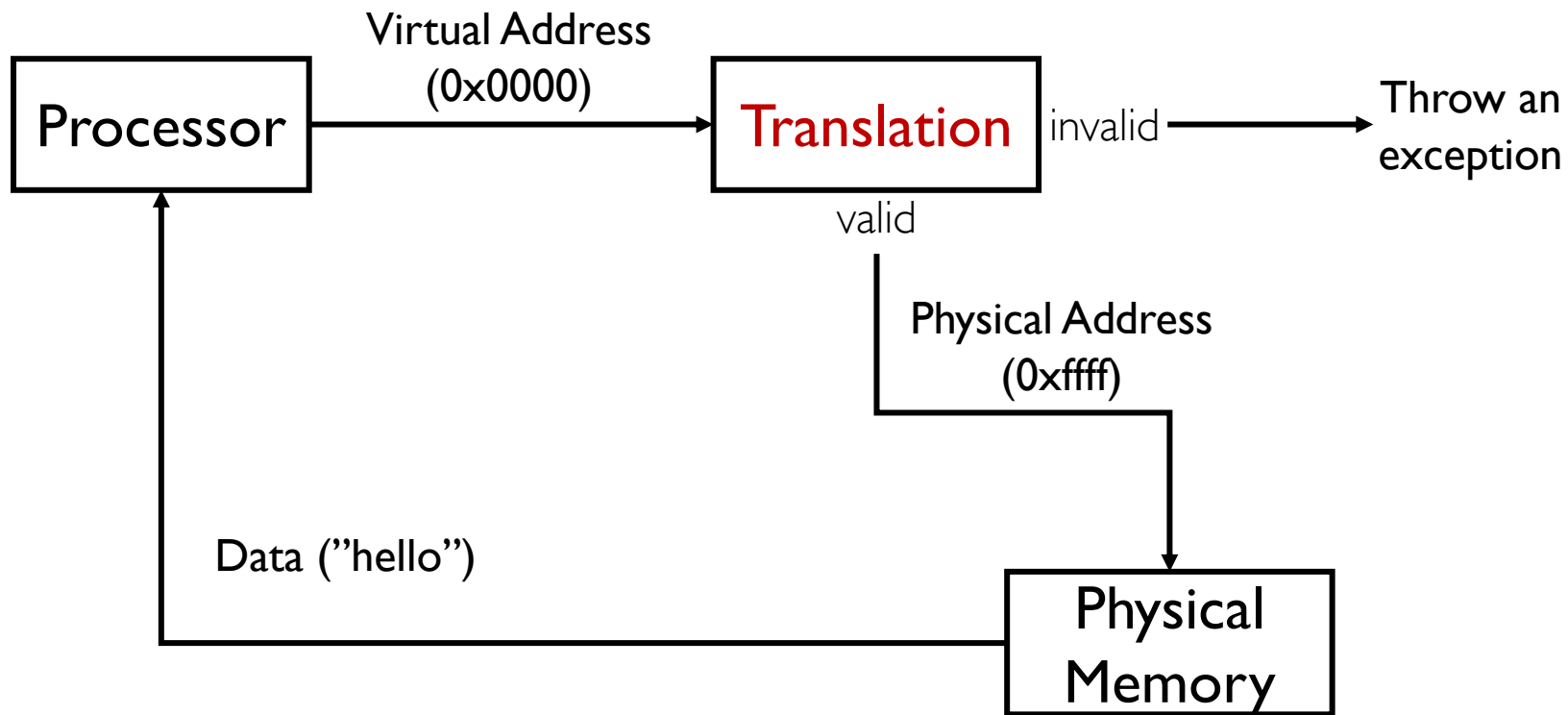
- Address Translation Concept
- Segmentation (分段)
- Paging (分页)

Goals for Today

- Address Translation Concept
- Segmentation (分段)
- Paging (分页)

Address Translation

- From virtual memory address (虚拟内存地址) to physical memory address (物理内存地址)



Address Translation

- From virtual memory address (虚拟内存地址) to physical memory address (物理内存地址)
- The goals and motivations of address translation
 - Memory protection
 - Memory sharing
 - Flexible memory placement
 - Sparse addresses
 - Runtime lookup efficiency
 - Compact translation tables
 - Portability

Address Translation

- From virtual memory address (虚拟内存地址) to physical memory address (物理内存地址)
- The goals and motivations of address translation
- When translation exists, processor uses virtual addresses, physical memory uses physical addresses
 - Not every processor/OS has address translation, e.g., certain embedded chips.

Address Translation

- From virtual memory address (虚拟内存地址) to physical memory address (物理内存地址)
- The goals and motivations of address translation
- When translation exists, processor uses virtual addresses, physical memory uses physical addresses
- Address translation involves intensive hardware-OS cooperation

Address Translation

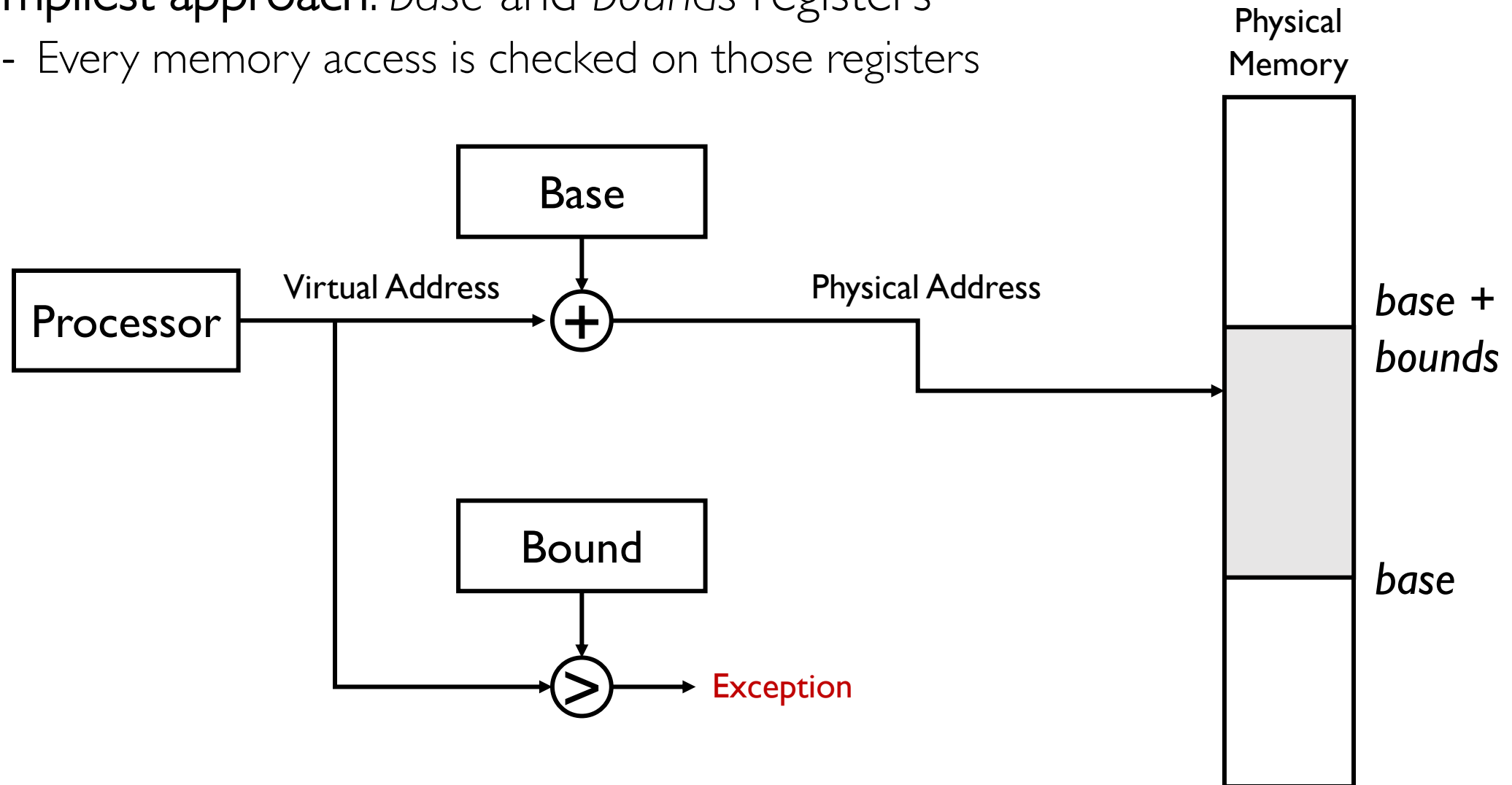
- From virtual memory address (虚拟内存地址) to physical memory address (物理内存地址)
- The goals and motivations of address translation
- When translation exists, processor uses virtual addresses, physical memory uses physical addresses
- Address translation involves intensive hardware-OS cooperation
- Address space: all the addresses and state a process can touch
 - Each process and kernel has different address space

Goals for Today

- Address Translation Concept
- Segmentation (分段)
- Paging (分页)

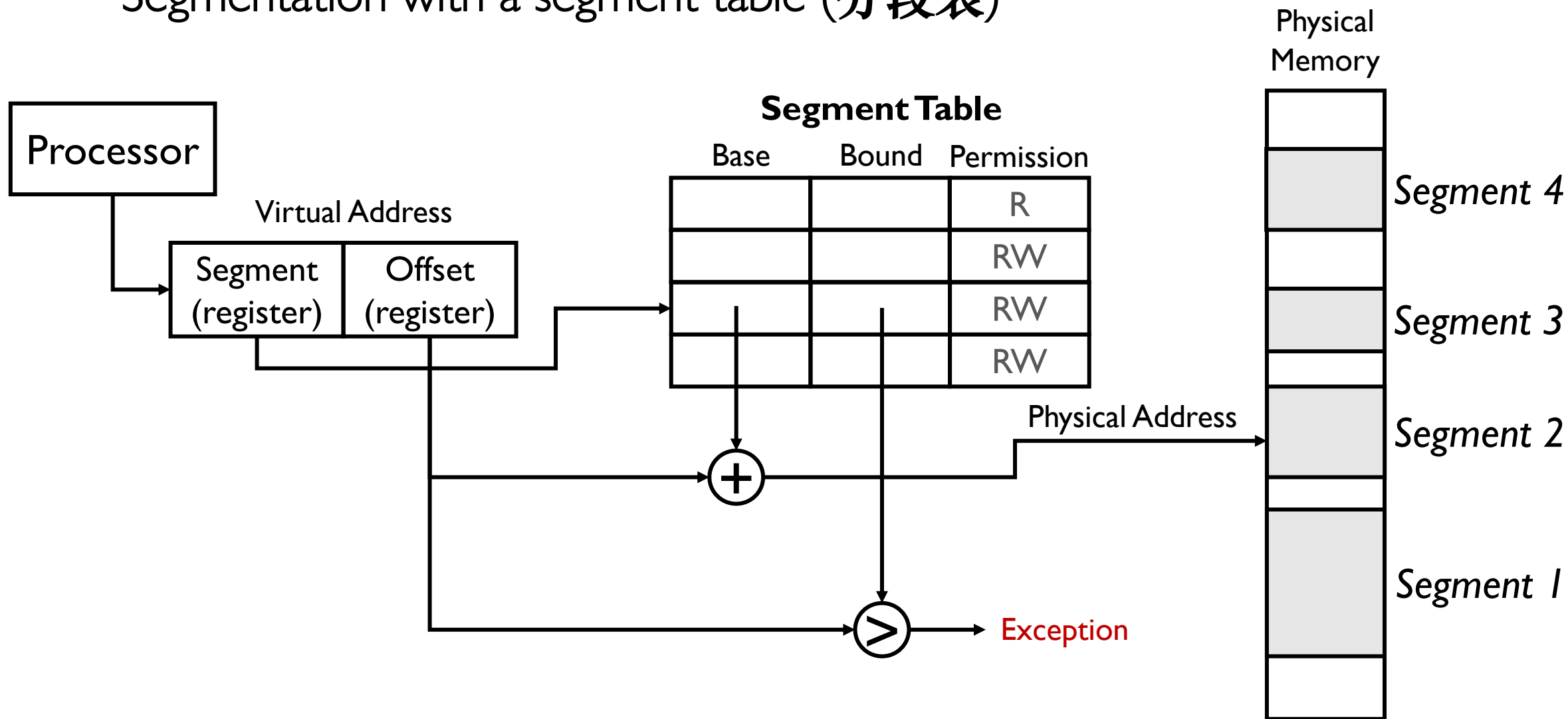
Segmented Memory

- Simplest approach: *base* and *bounds* registers
 - Every memory access is checked on those registers



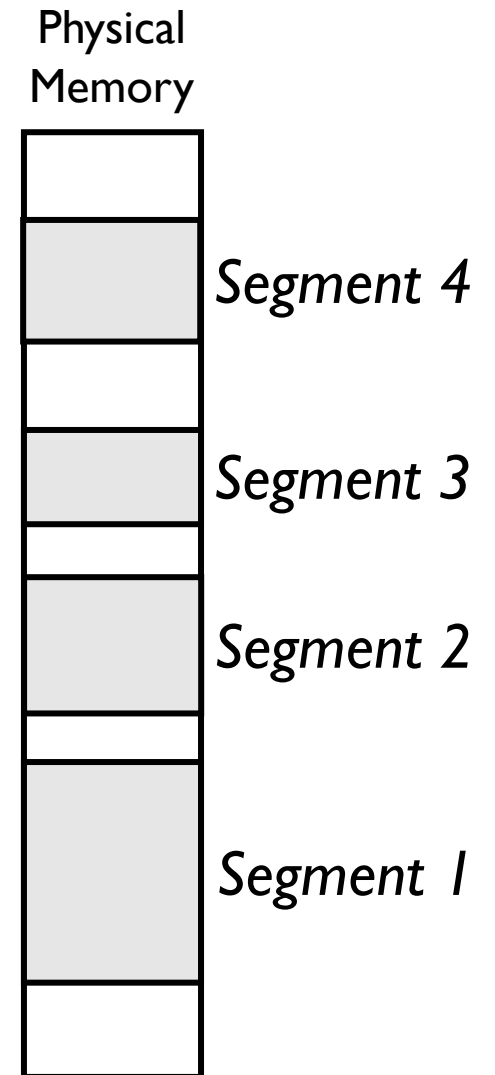
Segmented Memory

- Segmentation with a segment table (分段表)



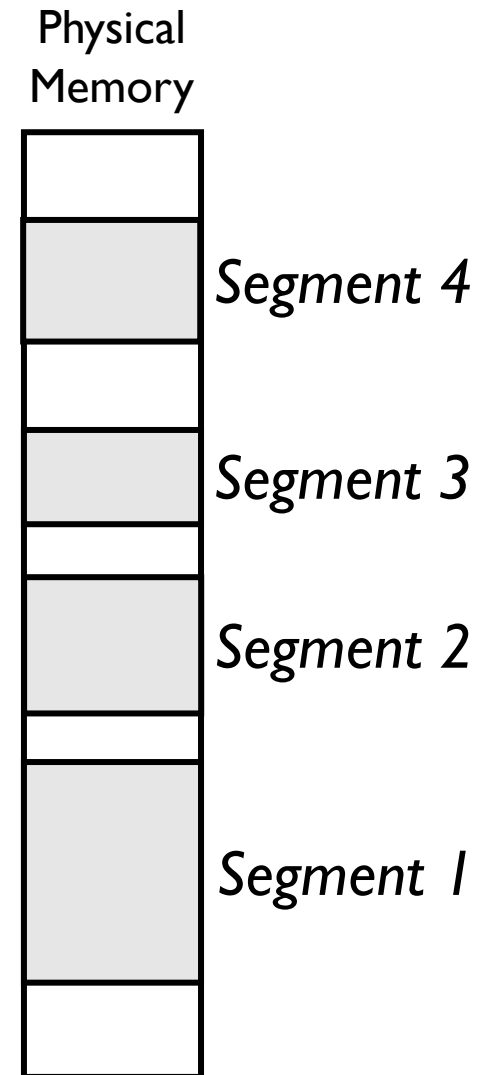
Segmented Memory

- Segmentation with a segment table (分段表)
 - *Why there are “holes” in the physical memory*
 - *What if a program branches into those “holes”?*



Segmented Memory

- Segmentation with a segment table (分段表)
 - Why there are “holes” in the physical memory
 - Processes come and go..
 - What if a program branches into those “holes”?
 - Segmentation error..



Segmented Memory

- The real segmentation implementation could vary a lot
 - Some Oses like Multics allocates a segment for each data structure to allow fine-grained protection and sharing between processes
 - Most modern systems use segments only for coarse-grained memory regions

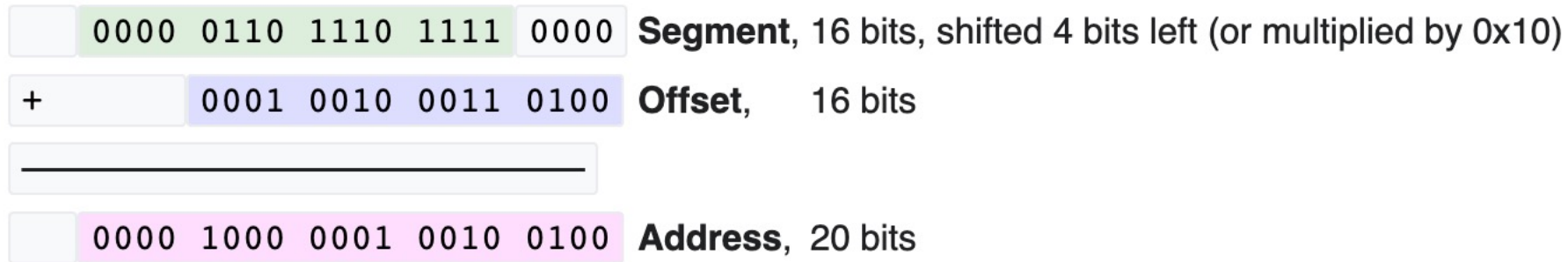
Segmented Memory

- An 80386 view of memory segmentation (each 16-bits long)
 - Code segment: CS
 - Data segment: DS
 - Stack segment: SS
 - Extra segment: ES, FS GS
- Developer practice
 - All CPU instructions are implicitly fetched from the code segment (CS register).
 - Most memory references come from the data segment specified by the segment selector held in the DS register. These may also come from the extra segment specified by the segment selector held in the ES register, if a segment-override prefix precedes the instruction that makes the memory reference.
 - Processor stack references, either implicitly (e.g. push and pop instructions) or explicitly (memory accesses using (E)SP or (E)BP registers) use the stack segment (SS register).
 - String instructions (e.g. stos, movs), along with data segment, also use the extra segment specified by the segment selector held in the ES register.

```
movl $foo, 0x10(%esp)
=
movl $foo, %ss:0x10(%esp)
```


Segmented Memory

- An x86 view of memory segmentation
 - In real mode, there is no segment table

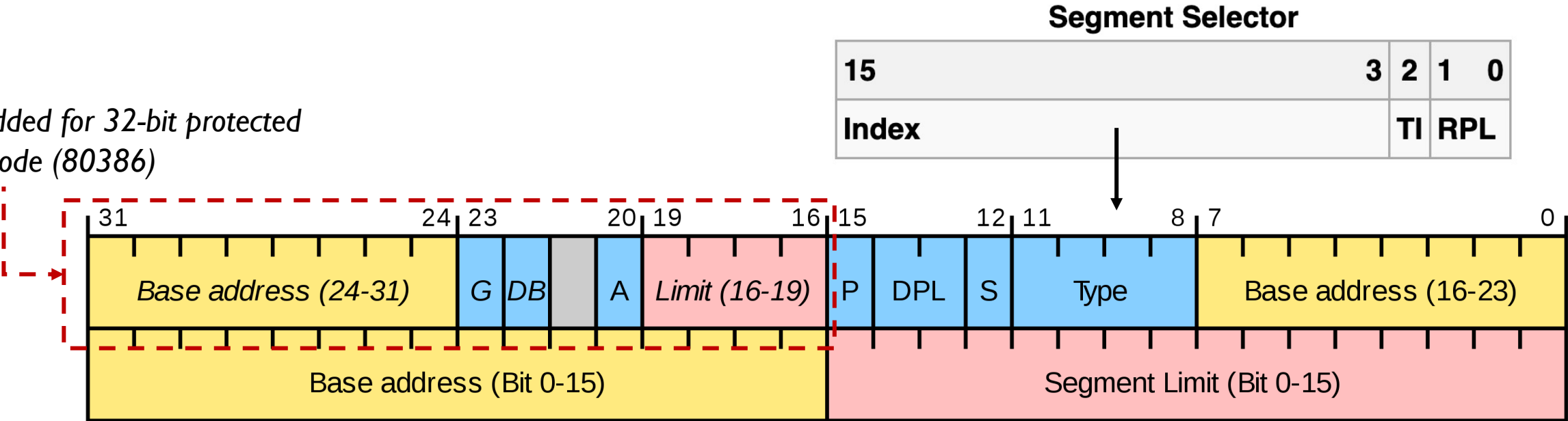


In real mode
no segment table!

Segmented Memory

- An x86 view of memory segmentation
 - In protected mode, the segment table is called global descriptor table (GDT, 全局描述符表) or local descriptor table (LDT, 局部描述符表)
 - Linear address = base address + offset

Added for 32-bit protected mode (80386)



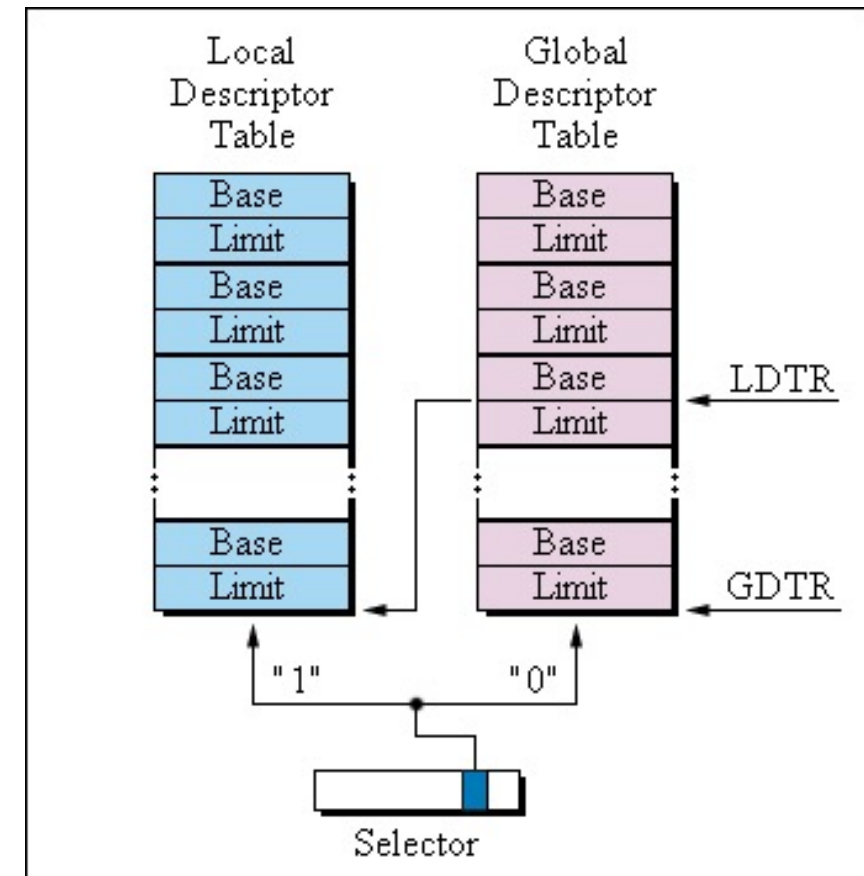
A segment descriptor

Segmented Memory

- An x86 view of memory segmentation
 - In protected mode, the segment table is called global descriptor table (GDT, 全局描述符表) or local descriptor table (LDT, 局部描述符表)
 - Linear address = base address + offset

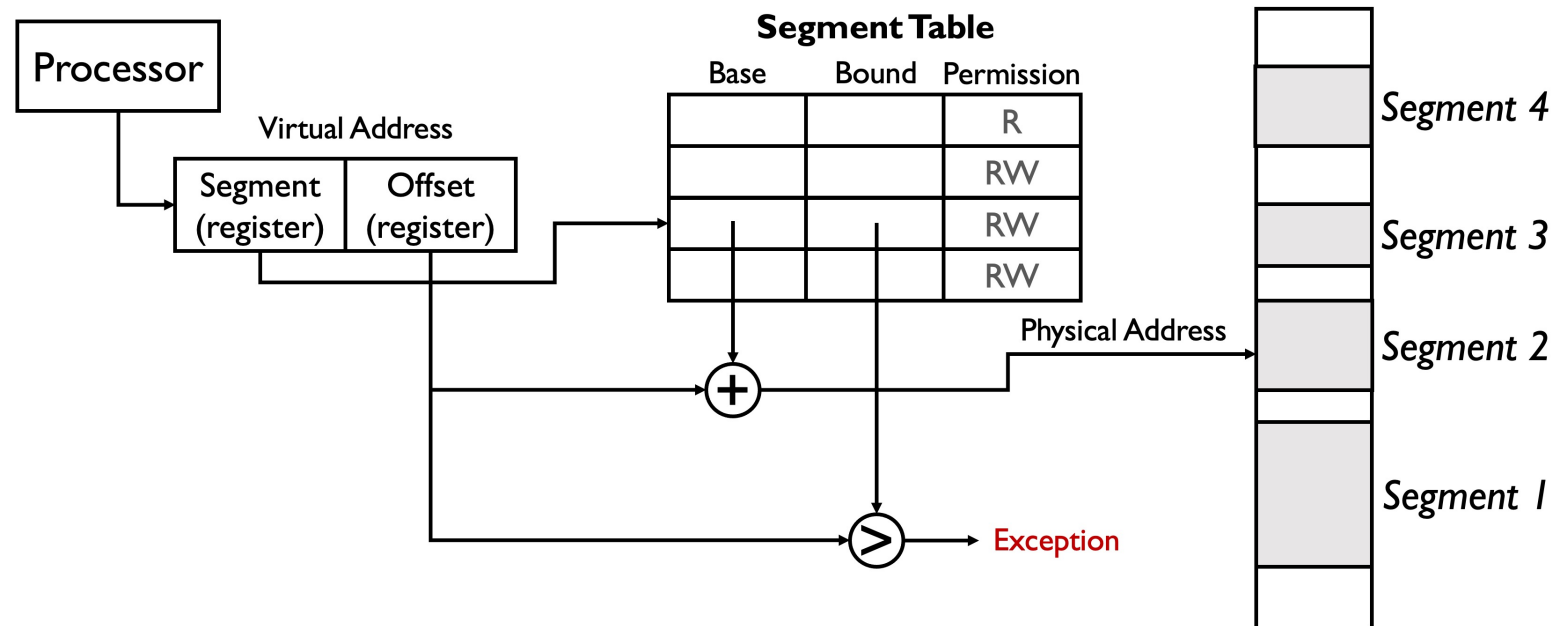
```

151 // Segment Descriptors
152 struct Segdesc {
153     unsigned sd_lim_15_0 : 16; // Low bits of segment limit
154     unsigned sd_base_15_0 : 16; // Low bits of segment base address
155     unsigned sd_base_23_16 : 8; // Middle bits of segment base address
156     unsigned sd_type : 4; // Segment type (see STS_constants)
157     unsigned sd_s : 1; // 0 = system, 1 = application
158     unsigned sd_dpl : 2; // Descriptor Privilege Level
159     unsigned sd_p : 1; // Present
160     unsigned sd_lim_19_16 : 4; // High bits of segment limit
161     unsigned sd_avl : 1; // Unused (available for software use)
162     unsigned sd_rsv1 : 1; // Reserved
163     unsigned sd_db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
164     unsigned sd_g : 1; // Granularity: limit scaled by 4K when set
165     unsigned sd_base_31_24 : 8; // High bits of segment base address
166 };
  
```



Segmented Memory

- The power of segmentation
 - Access control
 - Code sharing (library routines)
 - Inter-process communication
 - Efficient management of dynamically allocated memory



Segmented Memory

- The principle downside of segmentation: overhead of managing a large number of variable size and dynamically growing memory segments.
 - *External fragmentation*: free space becomes noncontiguous
 - Compacting the memory is very slow
 - It becomes even more complex if the segments can grow (like heap)

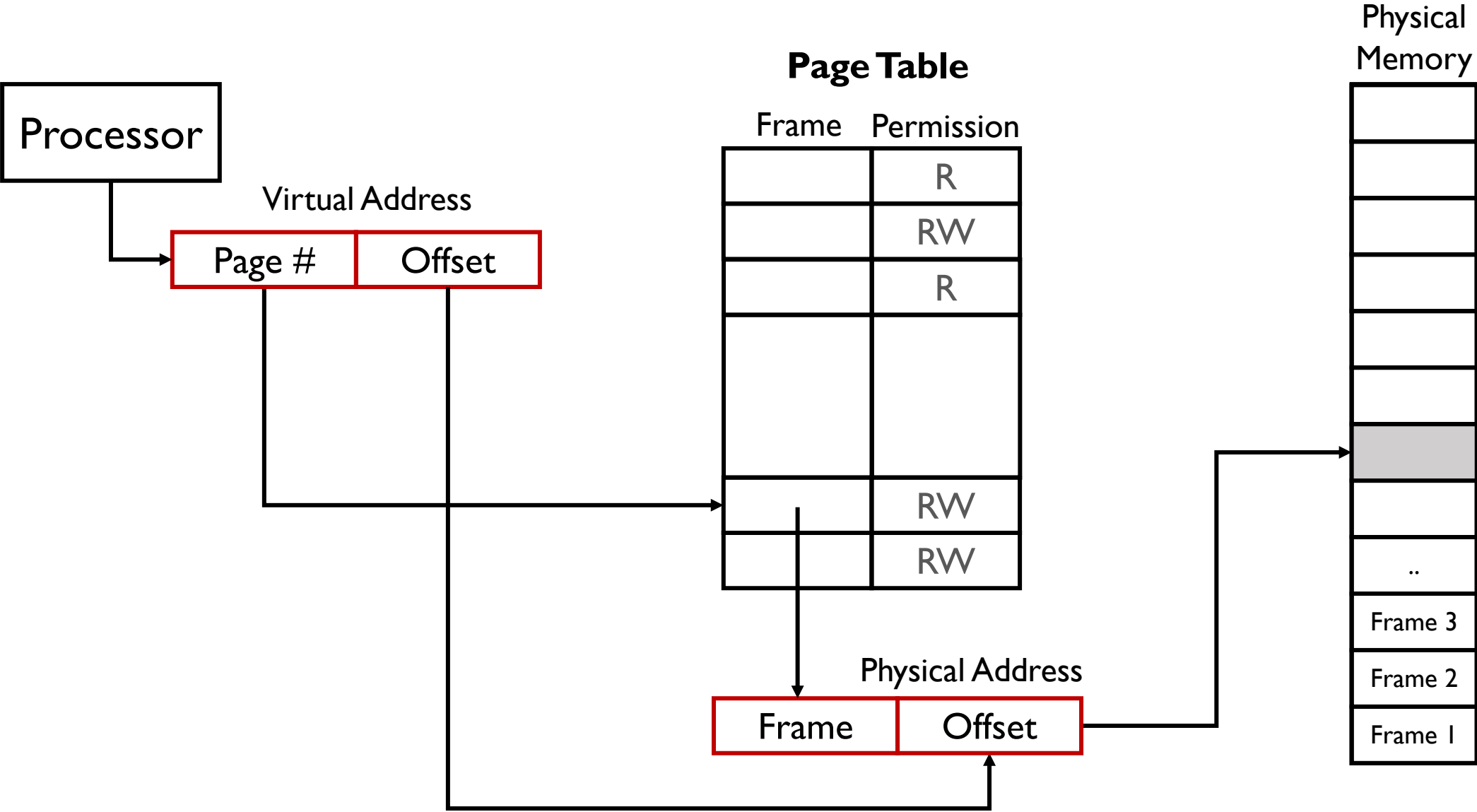
Goals for Today

- Address Translation Concept
- Segmentation (分段)
- Paging (分页)

Paged Memory

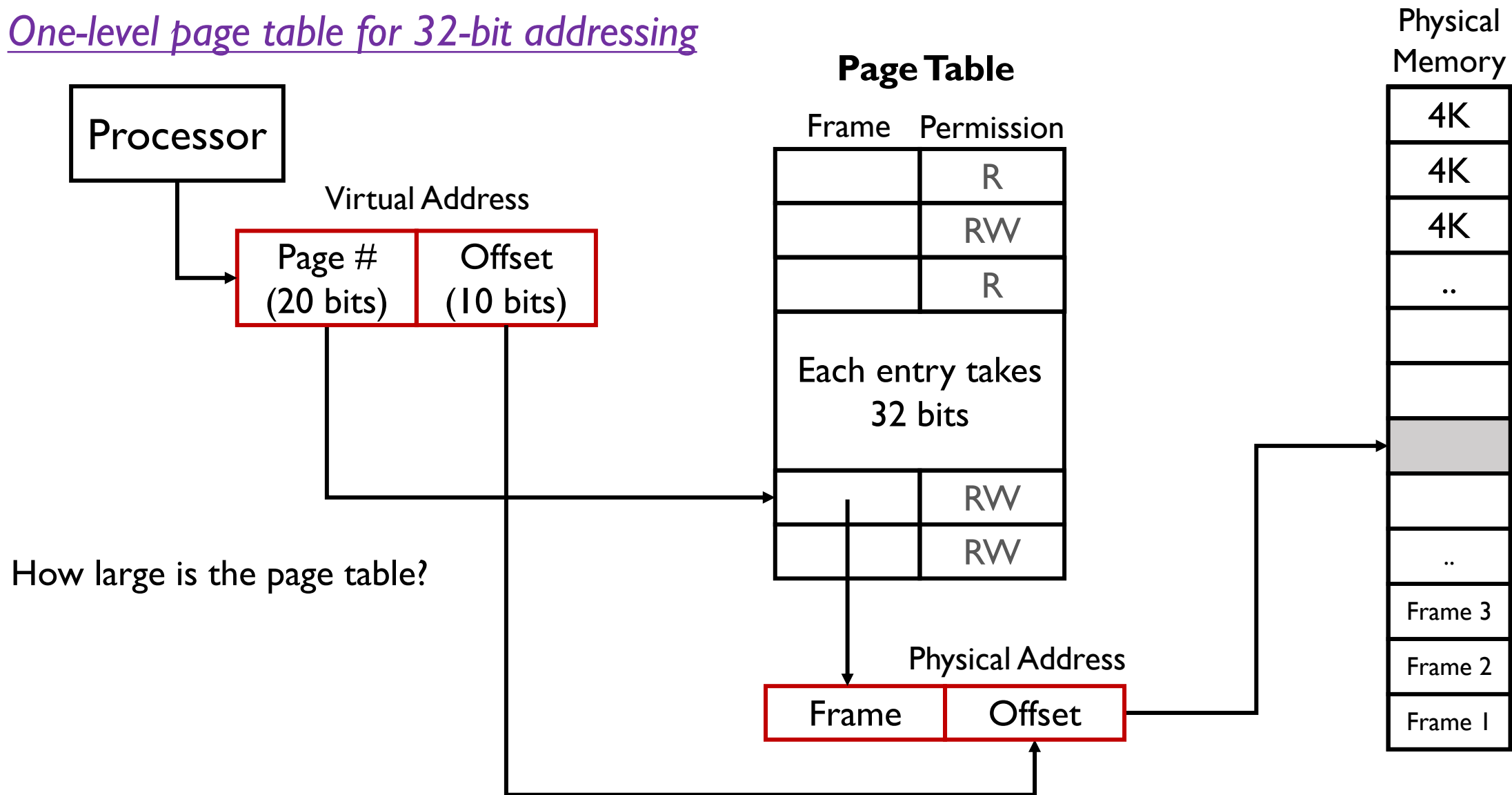
- Paging (分页): allocating memory in fixed-sized chunks called page frames (页框)
- A page table (页表) stores for each process whose entries contain pointers to the page frames.
 - More compact than segment table because it does not need to store "bound"
- What's cool: the pages are scattered across physical memory regions
 - Yet within a page, the memory access is contiguous
 - For instance, a large matrix might span many pages
- Memory allocation becomes very simple: find a page frame.

Paged Memory



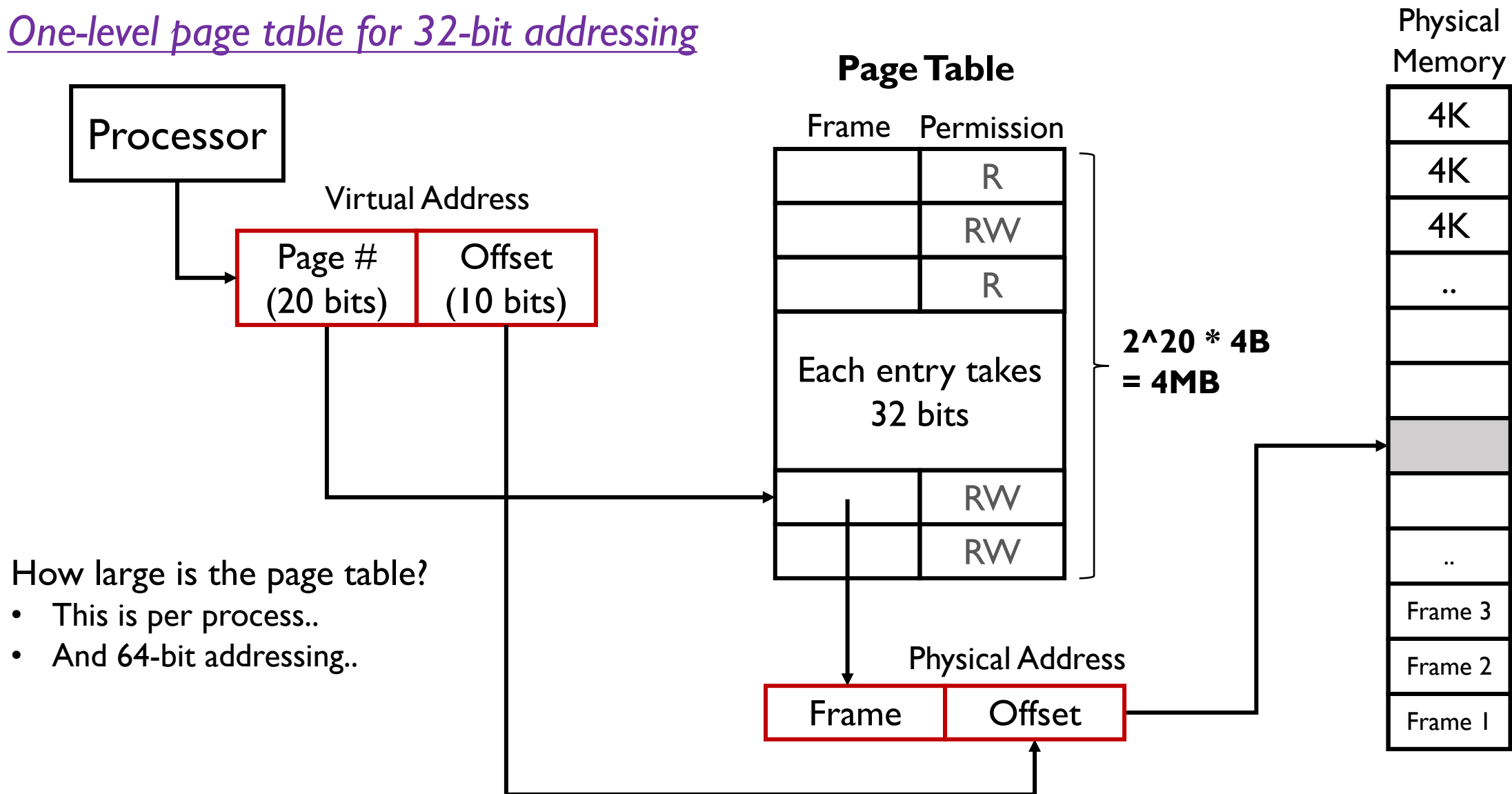
Paged Memory

One-level page table for 32-bit addressing



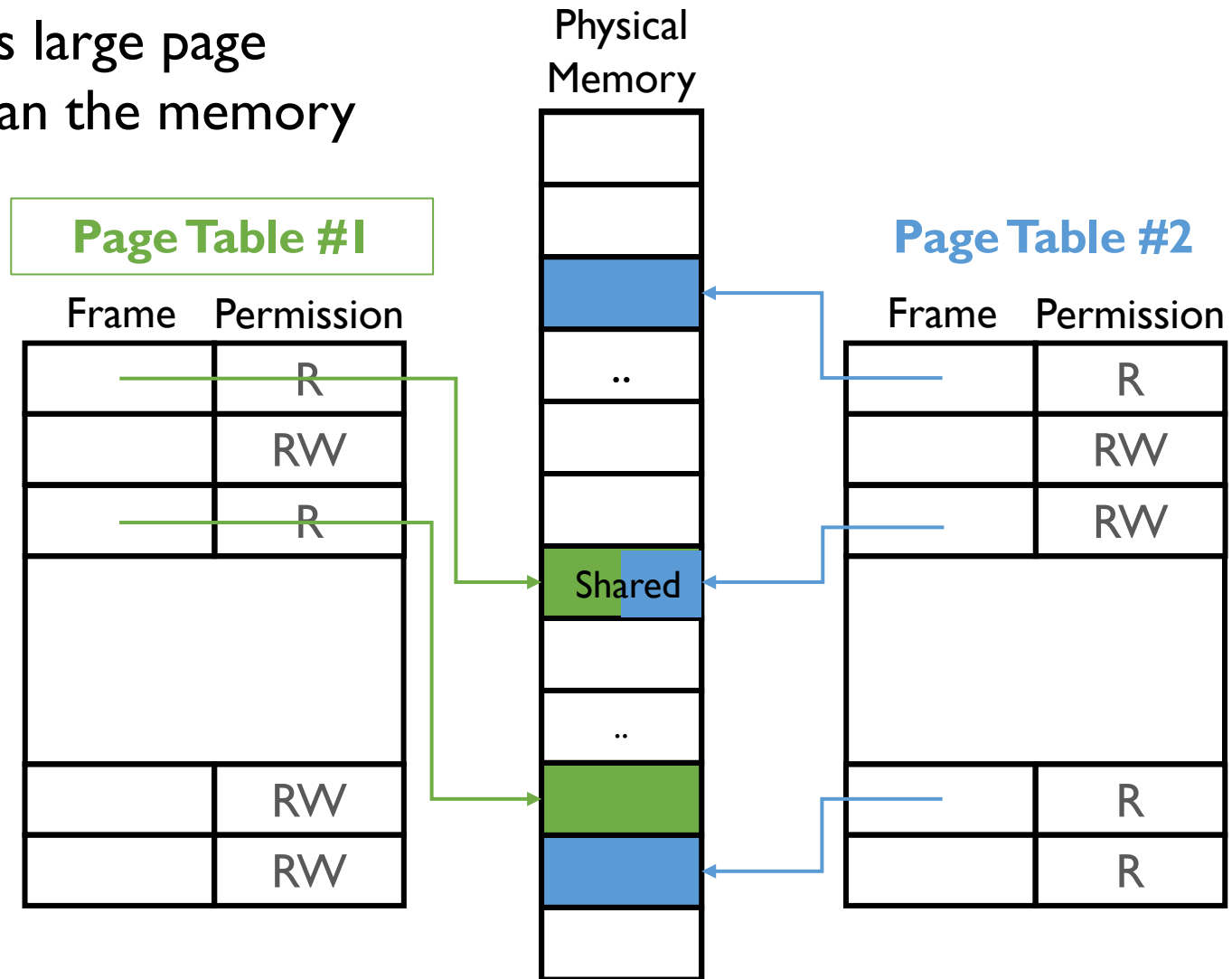
Paged Memory

One-level page table for 32-bit addressing

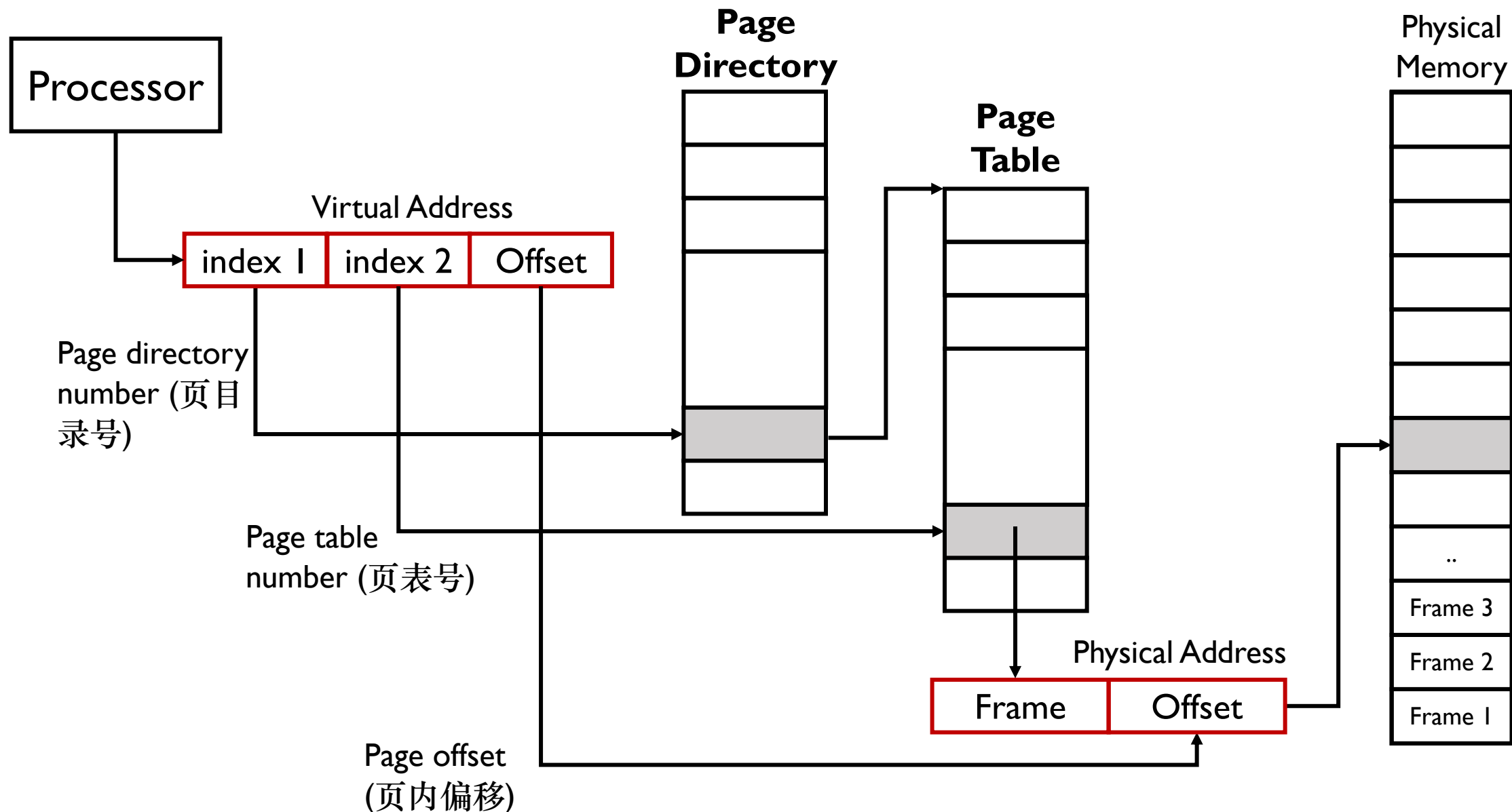


Paged Memory

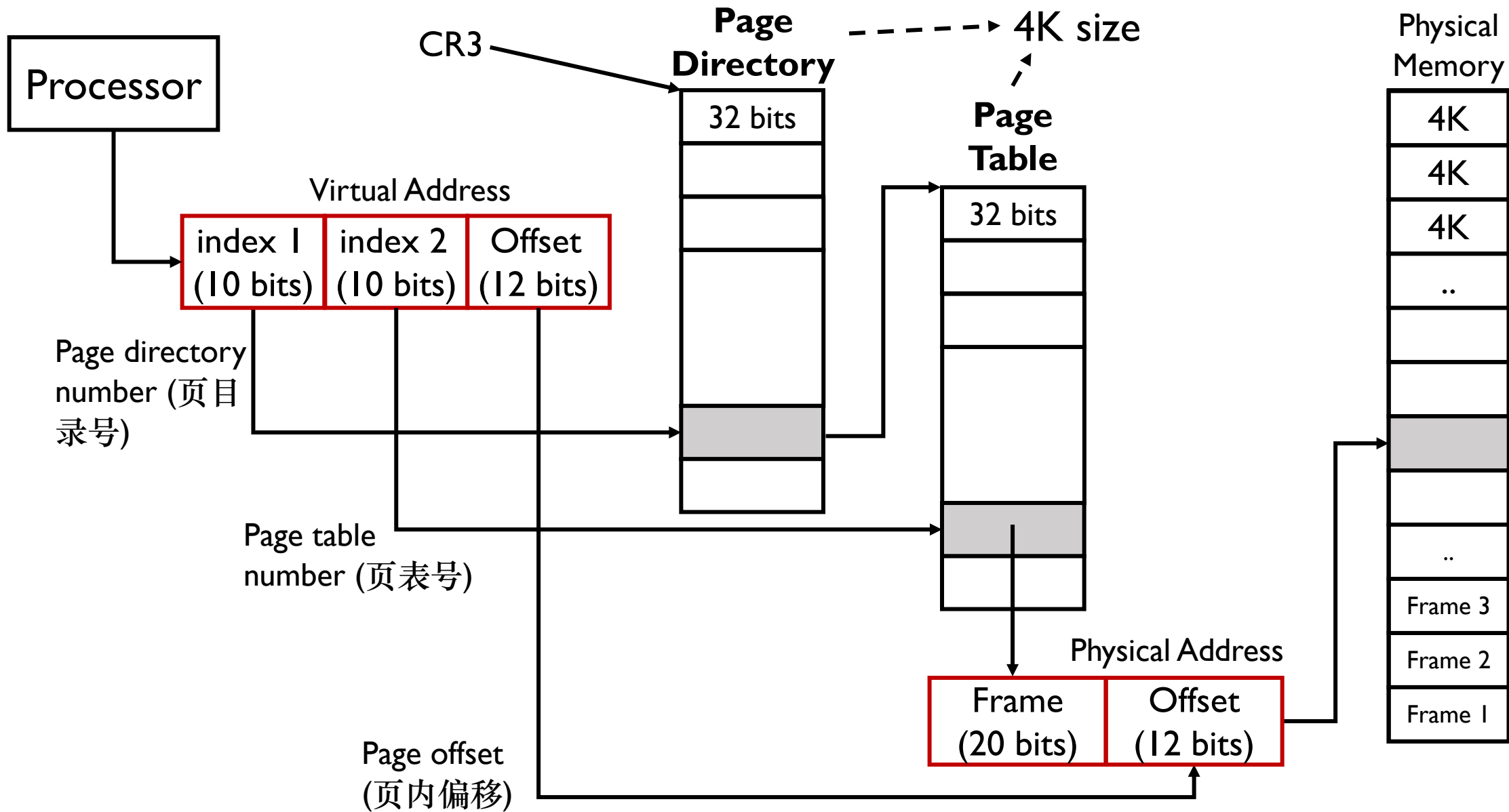
Single-level paging solves most of the issues (e.g., sharing as shown), but has large page table, which could be larger than the memory usage of the process itself!



Multi-level Paging

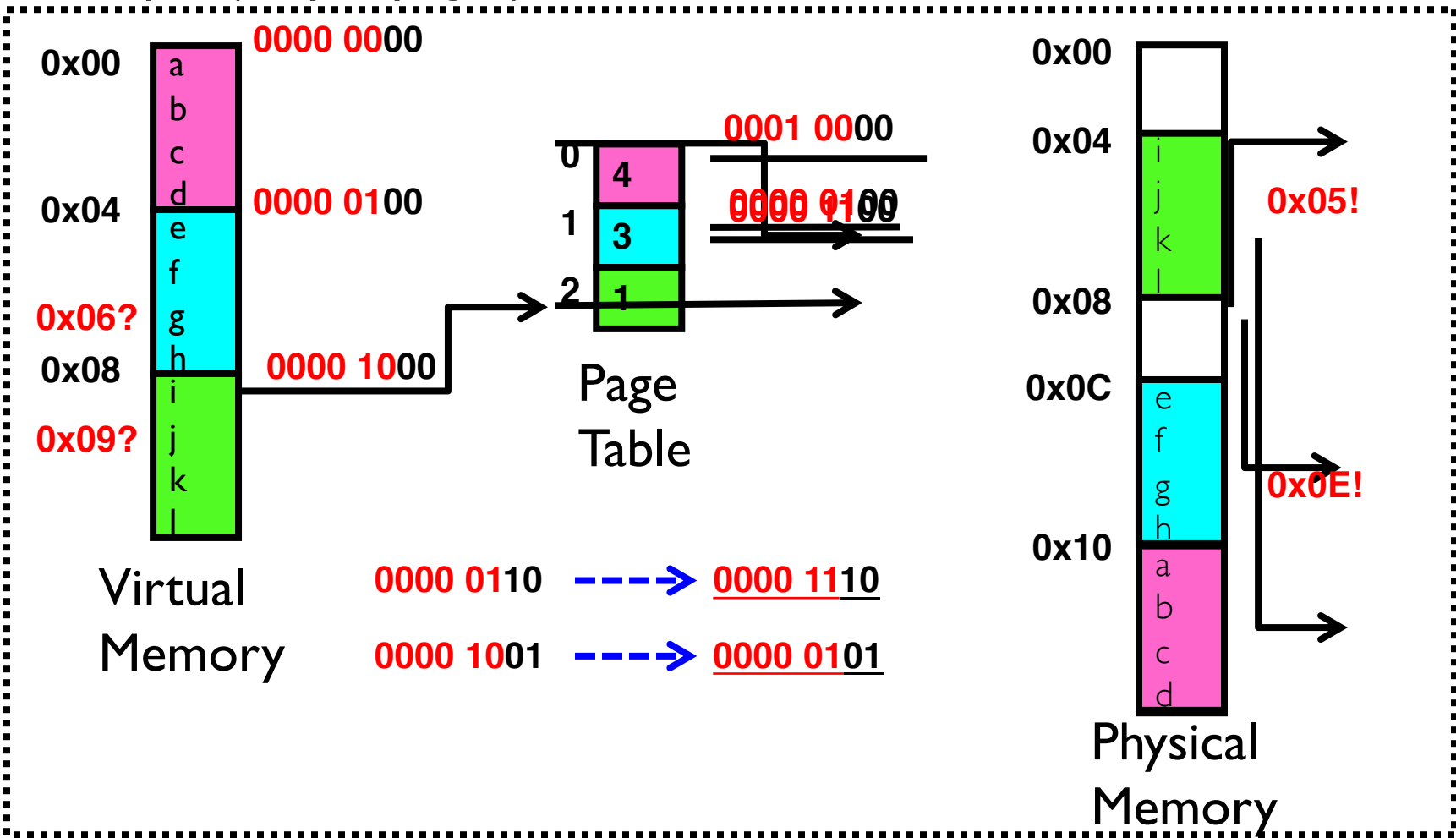


x86 Multi-level Paging



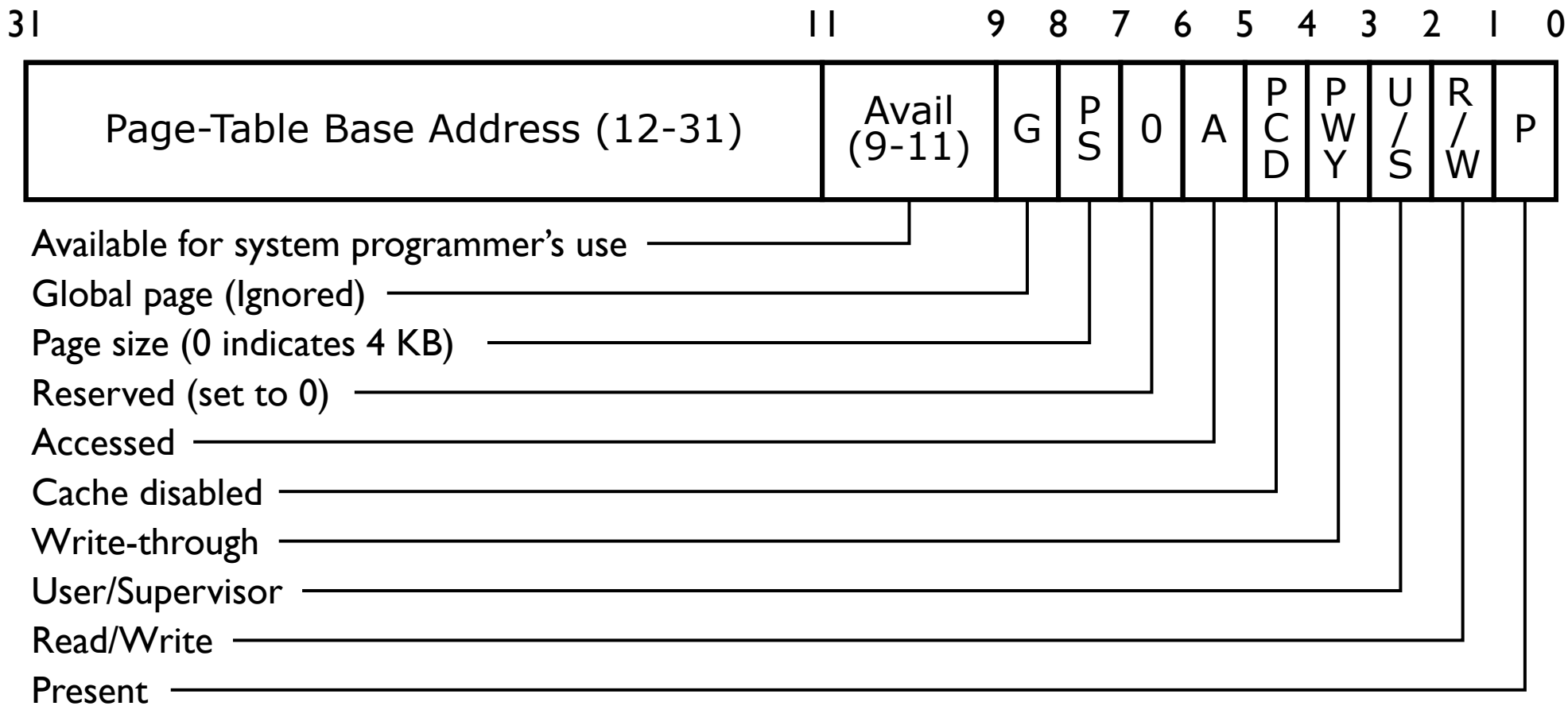
x86 Multi-level Paging

Example (4 byte pages)



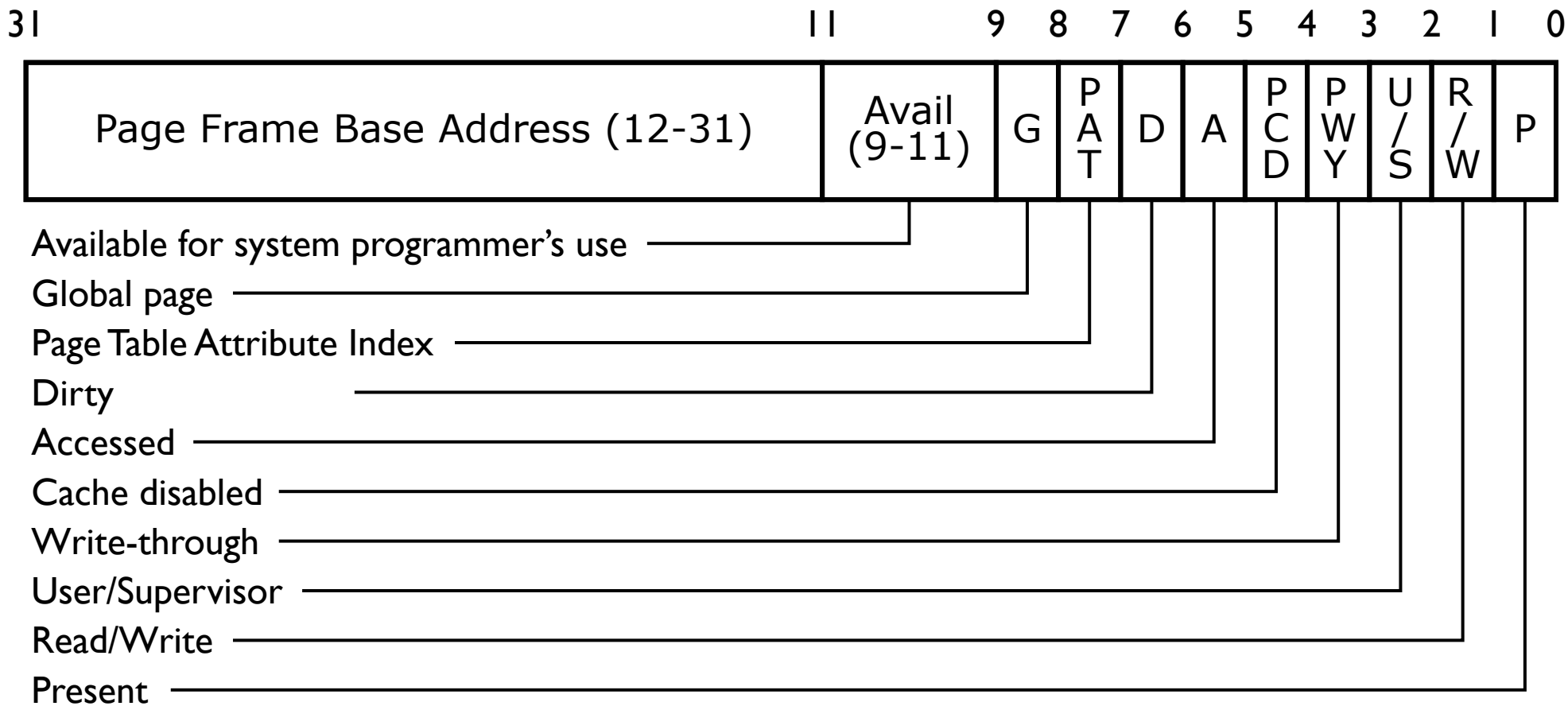
x86 Multi-level Paging

- Each page directory entry (PDE, 页目录项) is 32-bits long.



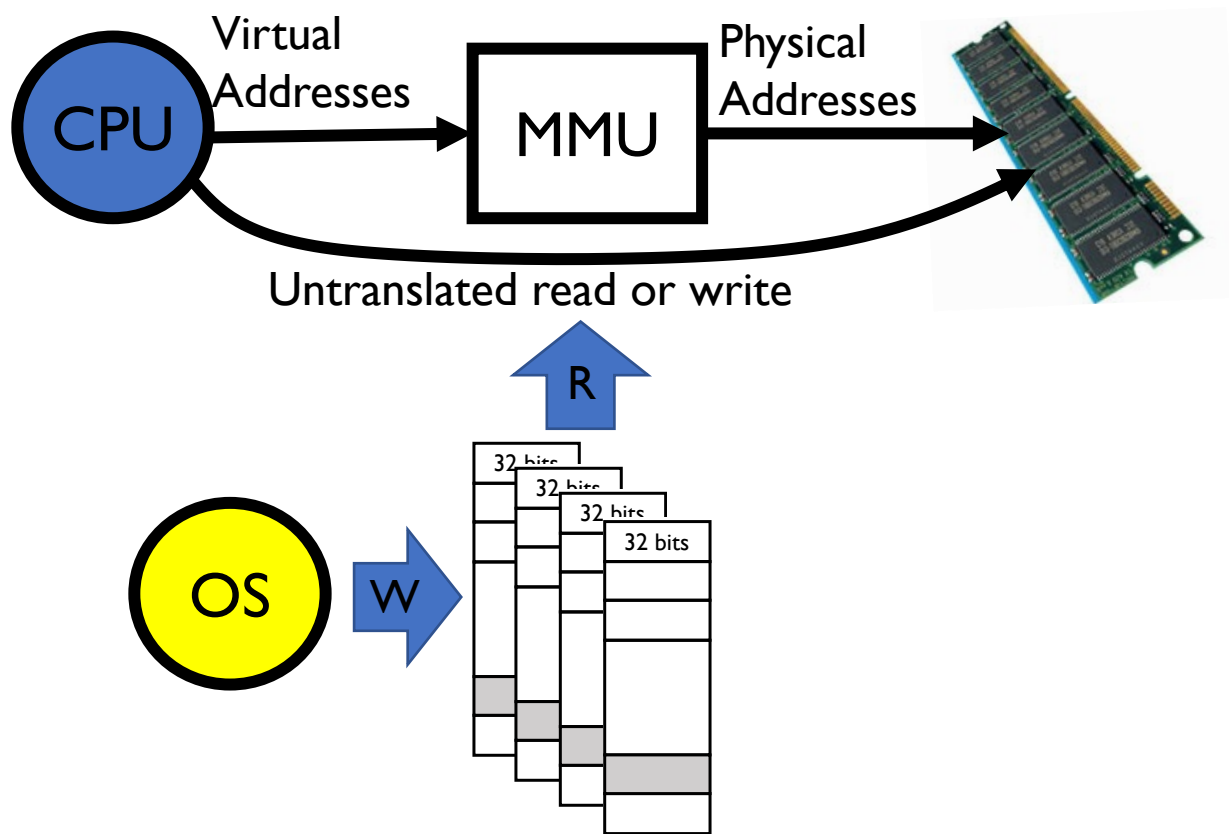
x86 Multi-level Paging

- Each page table entry (PTE, 页表项) is 32-bits long.



x86 Multi-level Paging

- Memory management unit (MMU, 分页内存管理单元): the hardware that actually does the translation
 - Usually located in CPU



x86 Multi-level Paging

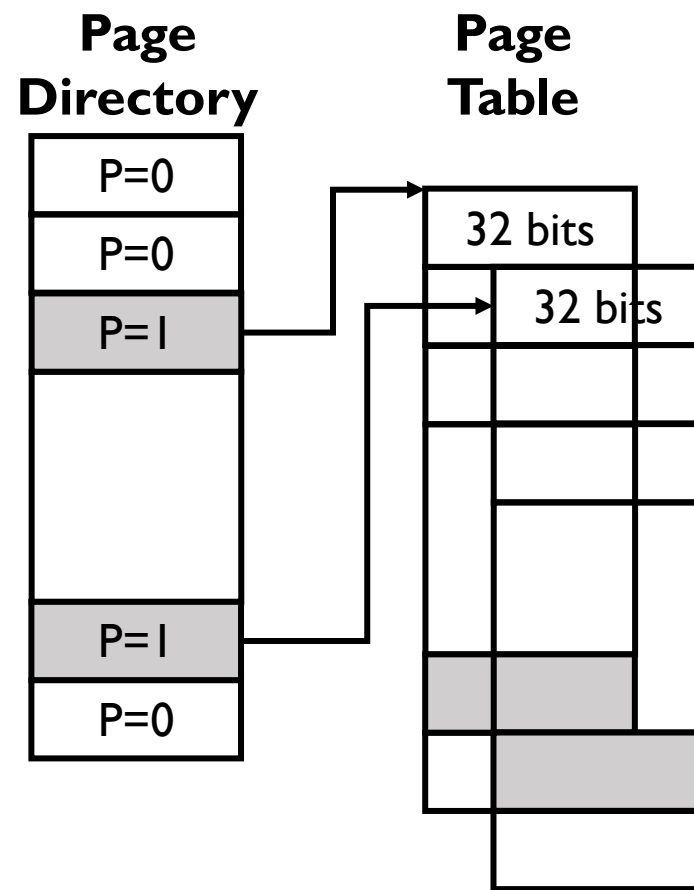
- Memory management unit (MMU, 分页内存管理单元): the hardware that actually does the translation
- Page size shall be neither too small or too large
 - Too small: large page table sizes; low cache hit ratio
 - Too large: memory waste
 - Typical range: 512B to 8192B; default 4KB on Linux.

x86 Multi-level Paging

- Memory management unit (MMU, 分页内存管理单元): the hardware that actually does the translation
- Page size shall be neither too small or too large
- Each process and kernel has their own page table!
 - Not threads
 - The same address of different processes translate to different physical locations, unless the page is shared
 - A process can only access/modify its own page table! Otherwise..

x86 Multi-level Paging

- Memory management unit (MMU, 分页内存管理单元): the hardware that actually does the translation
- Page size shall be neither too small or too large
- Each process and kernel has their own page table!
- Page tables can be **sparse** (vs. single-level paging)
 - Not every PDE has a corresponding page table.
 - Saves a lot of space.
 - It's good to fit page table into one page.



Page Fault

- Page Fault (缺页中断) happens when CPU/MMU accesses a memory location that is not readily mapped
 - Pure: memory swapped out; shared pages; etc.
 - ☐ After handled, the access will be performed again
 - Invalid: write to read-only pages; access to pages not allocated; etc.
 - ☐ Segmentation fault!

x86 Multi-level Paging

- Why PDE/PTE use 20 bits for addressing the next-level table or page?
- What needs to be switched on a context switch?
- If a process needs 1 page for its data, how many it will actually take?
- The largest address can be accessed in 2-level paging (32 bits address)?

x86 Multi-level Paging

- Why PDE/PTE use 20 bits for addressing the next-level table or page?
 - Page directory/tables are always page-aligned ($\% 4k = 0$).
- What needs to be switched on a context switch?
 - The page directory, stored in CR3
- If a process needs 1 page for its data, how many it will actually take?
 - 3 in total (1 page directory + 1 page table)
- The largest address can be accessed in 2-level paging (32 bits address)?
 - $4K * 2^{10} * 2^{10} = 4G$

Virtual or Physical??

- CR3 stores the virtual or physical address of the page directory?
- How about the PDE/PTE?
- How can OS manipulate the page directory/tables?

<https://wiki.osdev.org/Paging#Manipulation>

x86_64 Multi-level Paging

- 4-level: 48 bits

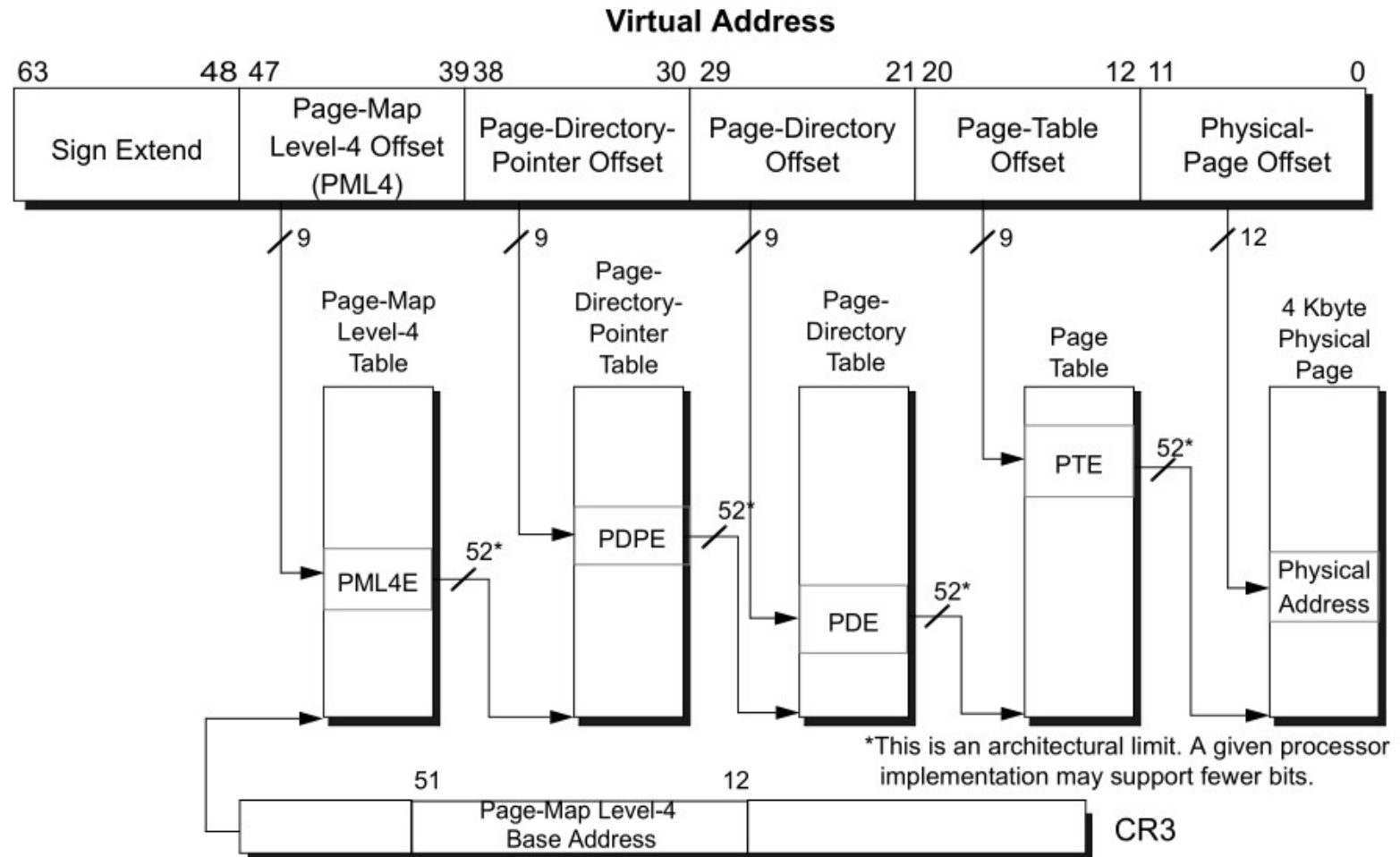
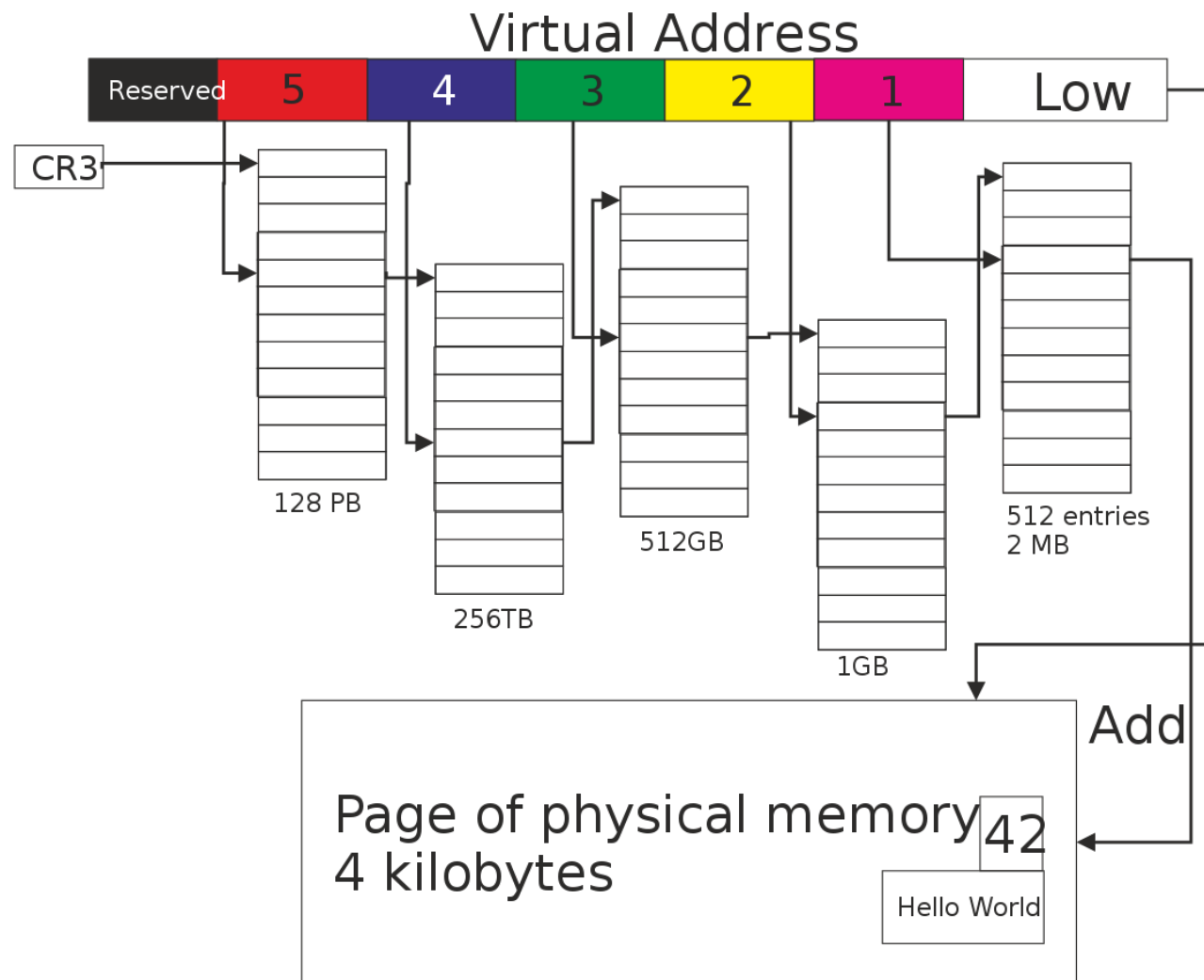


Figure 5-17. 4-Kbyte Page Translation—Long Mode

x86_64 Multi-level Paging

- 4-level: 48 bits
- 5-level: 64 bits

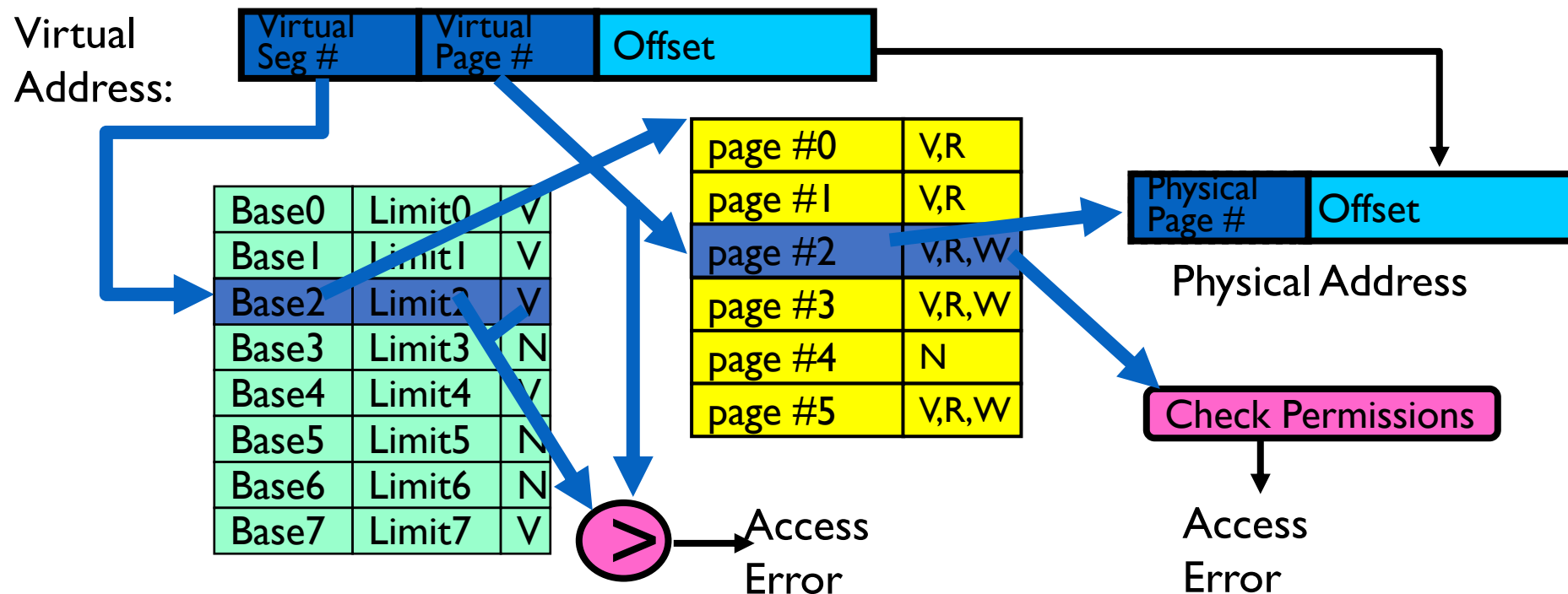


Multi-level Paging Summary

- Pros:
 - Only need to allocate as many page table entries as we need for application
 - ❑ In other words, sparse address spaces are easy
 - Easy memory allocation
 - Easy Sharing
 - ❑ Share at segment or page level (need additional reference counting)
- Cons:
 - One pointer per page (typically 4K – 16K pages today)
 - Page tables need to be contiguous
 - ❑ However, previous example keeps tables to exactly one page in size
 - Two (or more, if >2 levels) lookups per reference
 - ❑ Seems very expensive!

Segments + Paging

- What about a tree of tables?
 - Lowest level page table \Rightarrow memory still allocated with bitmap
 - Higher levels often segmented
- Could have any number of levels. Example (top segment):



Segmentation vs. Paging

- Intel x86 and Linux
 - 8086 era: segmentation and paging are both used
 - 80386 era: the segmentation is not really used
 - ☐ The processor provides 4 modes: none; paging only; segmentation only; both.
 - ☐ The `CS` is always set to 0 and the limit is 2^{32} .
 - x86_64 era: segmentation is considered as a legacy and not used in most OSes
- Now, everyone uses paging, few make any real use of segmentation.

<https://softwareengineering.stackexchange.com/questions/100047/why-not-segmentation>



Homework
