Operating Systems
Lecture 6

Address Translation

Prof. Mengwei Xu

Recap: Thread Abstraction

- Thread: a single <u>execution sequence</u> that represents a <u>separately schedulable task</u>

Each thread executes a sequence of instructions (assignments, conditionals, loops, procedures, etc) just as in the sequential programming model

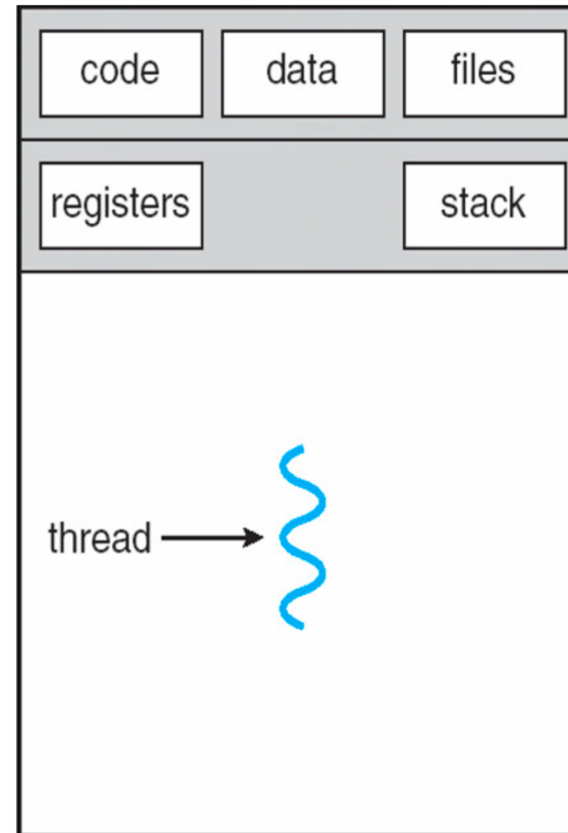The OS can run, suspend, or resume a thread at any time

The minimal scheduling unit in OS!
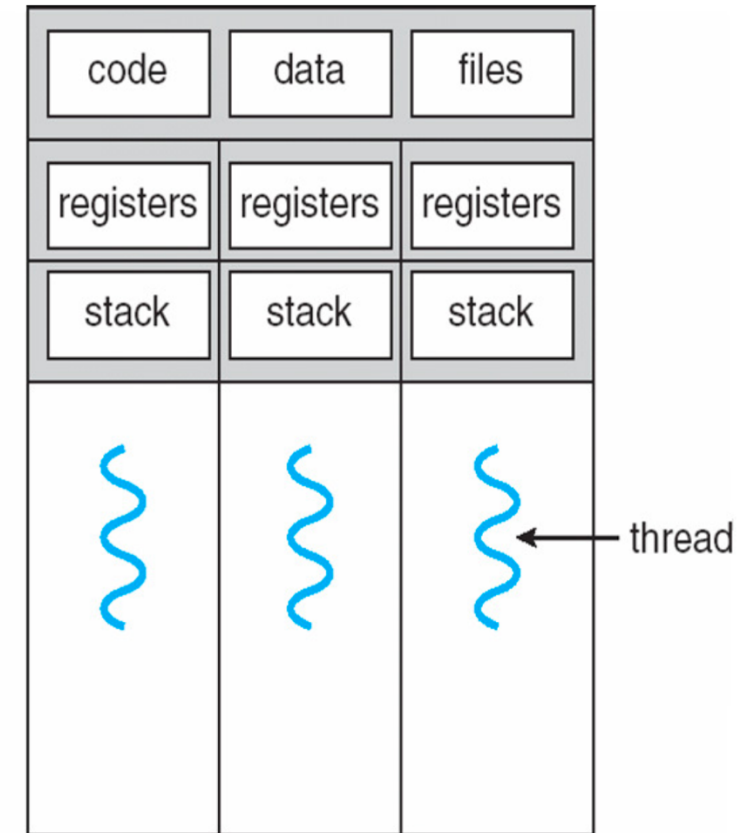
Recap: Thread Abstraction

- Thread: a single <u>execution sequence</u> that represents a <u>separately schedulable task</u>

Threads in the same process share memory space, but not execution context
- There will be thread context switch

single-threaded process

multithreaded process

Recap: Thread vs. Process

	Thread	**Process**
Currency	Both of them can be scheduled by OS.	
Context	Different threads/processes have their dedicated execution contexts (registers values and stacks). Scheduling them incurs context switching.	
Definition	A single execution sequence that represents a separately schedulable task	An execution of any program
	The minimal scheduling unit "a lightweight process"	The minimal dedicated memory space
Resources	Consume less resources	Consume more resources
Memory	Threads in the same process share memory space	Processors do not share memory space
Communications	Easier and faster for threads in the same process to communicate with each other	More complex and slow for different processes to communicate with each other

Recap: POSIX Thread Example

```c
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <pthread.h>
4
5    void *print_message_function( void *ptr );
6
7    main()
8    {
9         pthread_t thread1, thread2;
10        char *message1 = "Thread 1";
11        char *message2 = "Thread 2";
12        int  iret1, iret2;
13
14        iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
15        iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
16
17        pthread_join( thread1, NULL);
18        pthread_join( thread2, NULL);
19
20        printf("Thread 1 returns: %d\n",iret1);
21        printf("Thread 2 returns: %d\n",iret2);
22        exit(0);
23    }
24
25    void *print_message_function( void *ptr )
26    {
27        char *message;
28        message = (char *) ptr;
29        printf("%s \n", message);
30    }
```

What's the possible output?

Recap: Thread Data Structures

- ## Thread Control Block (TCB)
  - Stack pointer: each thread needs their own stack
  - Copy of processor registers
    - ❑ General-purpose registers for storing intermediate values
    - ❑ Special-purpose registers for storing instruction pointer and stack pointer
  - Metadata
    - ❑ Thread ID
    - ❑ Scheduling priority
    - ❑ Status
  - What's different from PCB??

Recap: Thread Data Structures

- Thread Control Block (TCB)

- Shared state

- OS does not enforce physical division on threads' own separated states

  - If thread A has a pointer to the stack location of thread B, can A access/modify the variables on the stack of thread B?

Recap: Thread Implementation

- Kernel threads
  - What are the use cases?

- User-level threads
  - Can be implemented with or without kernel help

Recap: Thread Implementation

- ## Create a thread
  - Allocate per-thread state: the TCB and stack
  - Initialize per-thread state: registers (args)
  - Put TCB on ready list

- ## Delete a thread
  - Remove the thread from the ready list so it will never run again
  - Free the per-thread state allocated for the thread
  - Can a thread delete itself?

- ## Context Switch
  - Voluntary: thread_yield
  - Involuntary: interrupts and exceptions

Implementing Multi-threaded Processes

- Implementing user-level multi-threaded processes through
  1. Kernel threads (each thread op traps into kernel)
  2. User-level libraries (no kernel support)
  3. Hybrid mode

Implementing Multi-threaded Processes

- **Implementing multi-threaded processes through kernel threads**
  - Each thread operation invokes the corresponding kernel thread syscall

Create a kernel thread	Create a user-level thread
- Allocate per-thread state in kernel: the TCB and stack - Initialize per-thread state: registers (args) - Put TCB on ready list	- User lib allocates a user-level stack - Invokes thread_create() syscall - Stores a pointer to the TCB in the PCB (why?)

How about join, yield, exit?

Implementing Multi-threaded Processes

- **Implementing multi-threaded processes in user libraries**
  - The library maintains everything in user space
    - ❑ TCBs, stacks, ready list, finished list
  - The library determines which thread to run
  - A thread op is just a procedure call

Implementing Multi-threaded Processes

- **Implementing multi-threaded processes in user libraries**
  - The library maintains everything in user space
    - ❑ TCBs, stacks, ready list, finished list
  - The library determines which thread to run
  - A thread op is just a procedure call

- **How can we make user-level threads run currently, as kernel is not aware of their existence?**

- **How can program change the PC and stack pointer?**

Implementing Multi-threaded Processes

- **Implementing multi-threaded processes in user libraries**
  - The library maintains everything in user space
    - ❑ TCBs, stacks, ready list, finished list
  - The library determines which thread to run
  - A thread op is just a procedure call

- **How can we make user-level threads run currently, as kernel is not aware of their existence?**
  - The preemptive way: timer interrupts (upcall) from kernel
  - The cooperative way: threads yield voluntarily

- **How can program change the PC and stack pointer?**
  - jmp and esp

Threads in Kernel vs. User

	User-level Threads	**Kernel Threads**
Currency	Both of them run currently	
Context	Share heap/code, but have separated stack/registers	
Role of kernel	No kernel assistance at all	Each thread operation invokes kernel syscall
Speed (context switch, creating, etc)	Fast	Slow
Memory cost	Small	Large
I/O waiting time	Cannot avoid the I/O waiting time (though there are certain optimizations to do so)	Kernel can schedule another thread when I/O blocks
Multi-core processor	No parallel on multi-core processors	Can schedule many threads in the same process at the same time on multi-core processors

Implementing Multi-threaded Processes

- Implementing multi-threaded processes in hybrid way: optimizations based on kernel threads
  - Hybrid thread join: for example, no need for syscall if the thread to be joined is already finished (with exit value saved in memory)
  - Per-processor kernel thread with user-level thread implementation
  - Scheduler activations: in recent Windows, the user-level scheduler can be notified when a thread blocks in a syscall, so it can schedule another thread to fully utilize the processor.
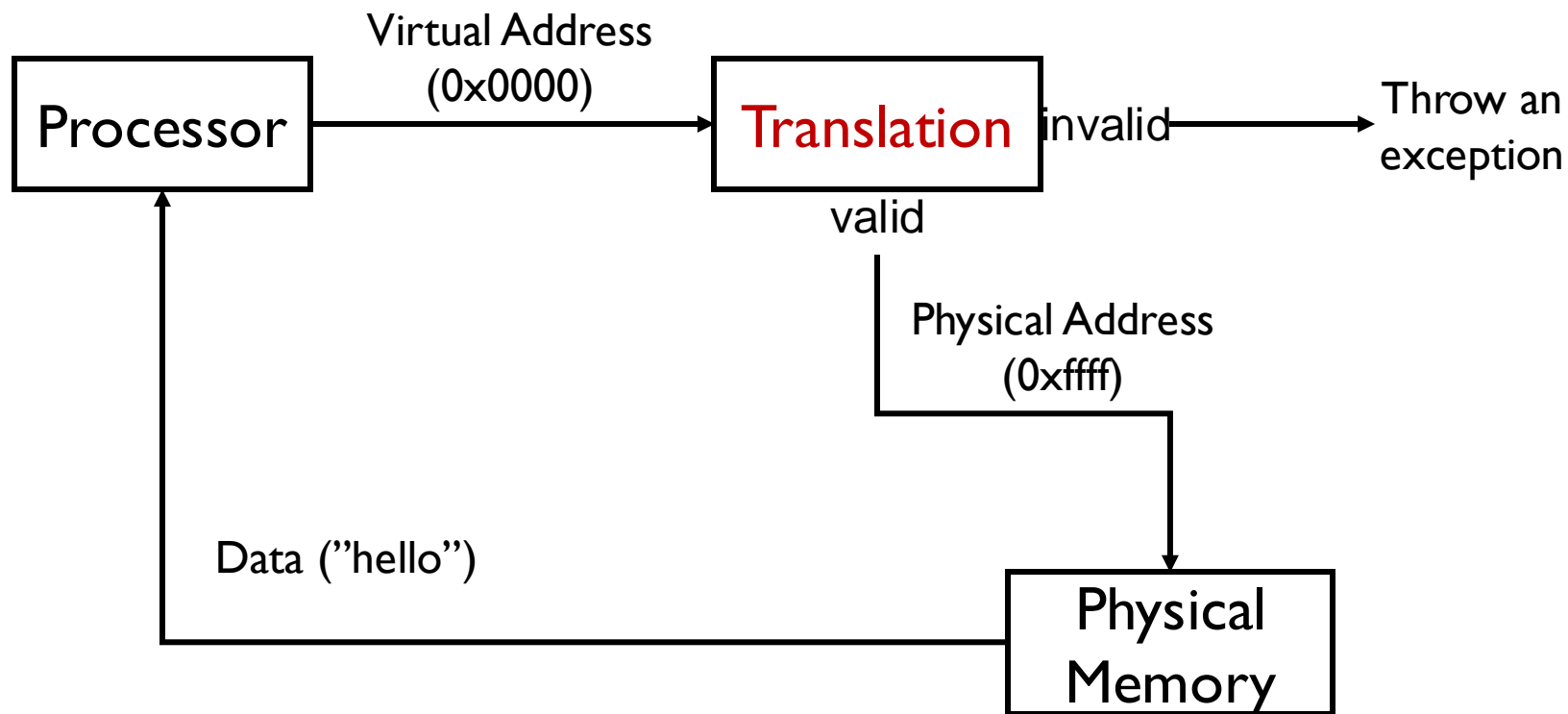
- Address Translation Concept
- Segmentation (分段)
- Paging (分页)

- **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
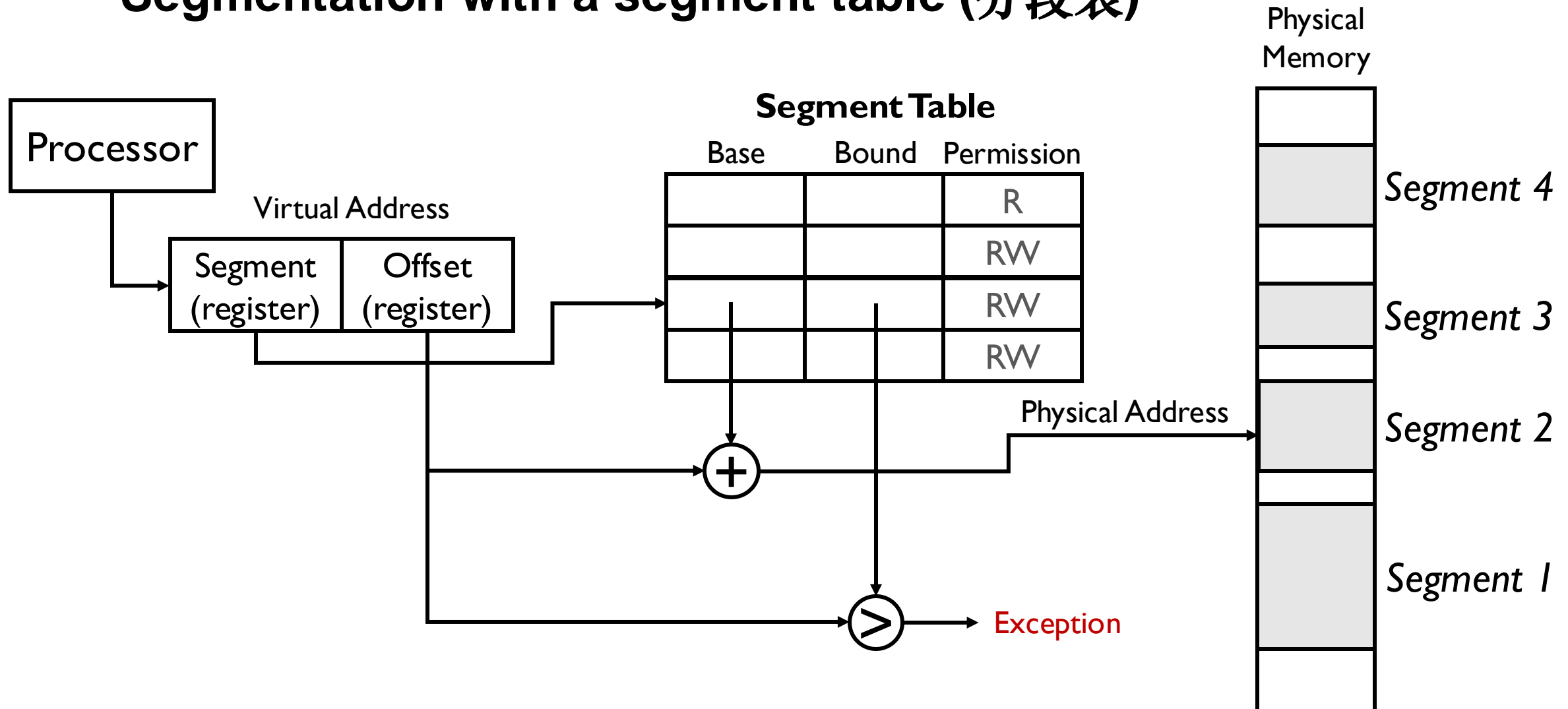- **Segmentation (分段)**
- Paging (分页)

Segmented Memory

- **Simpliest approach**: *base* and *bounds* registers
  - Every memory access is checked on those registers
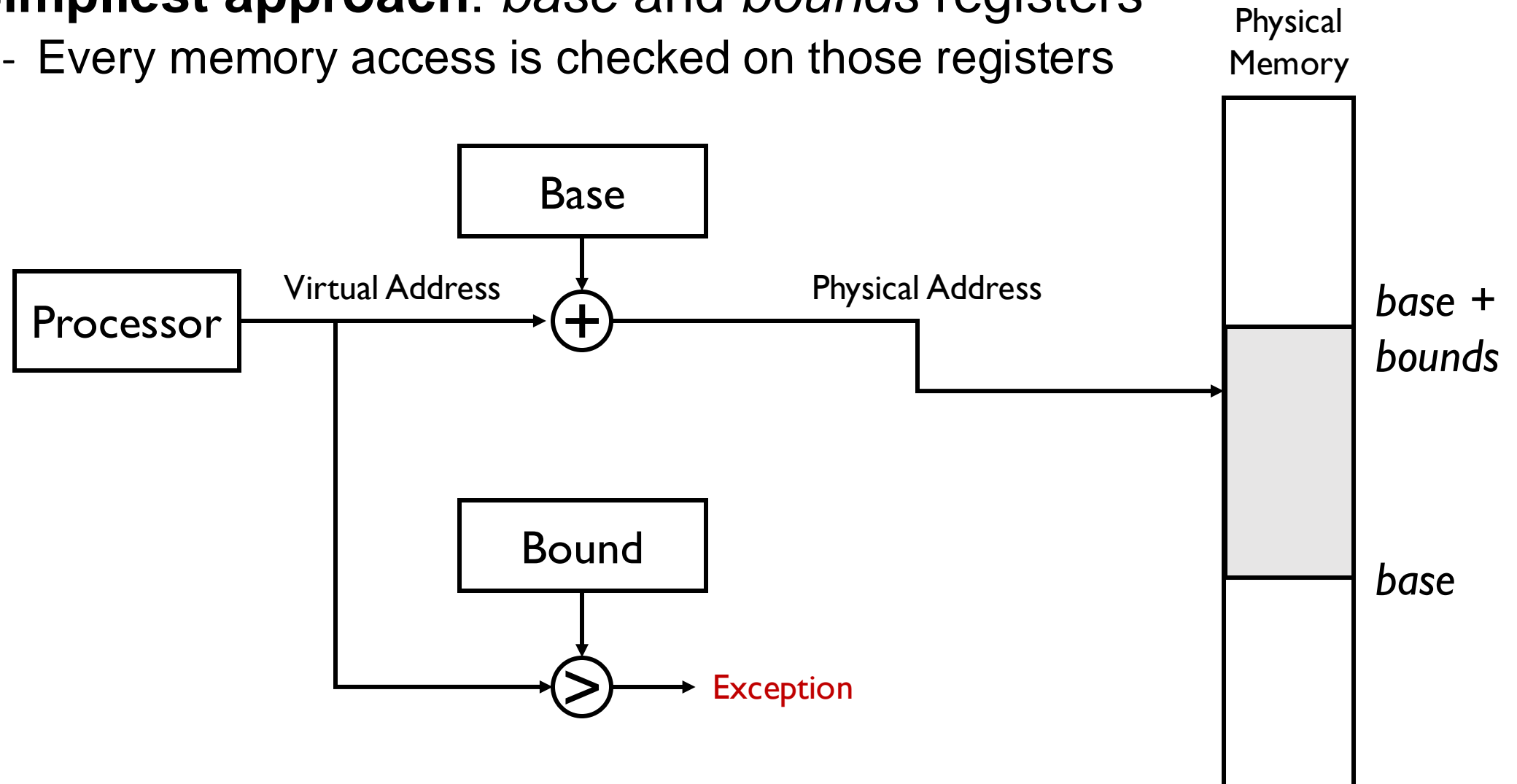
Segmented Memory

- **Segmentation with a segment table (分段表)**

- **Segmentation with a segment table (分段表)**

Physical Memory

  - *Why there are "holes" in the physical memory*
  - *What if a program branches into those "holes"?*

Segment 4

Segment 3

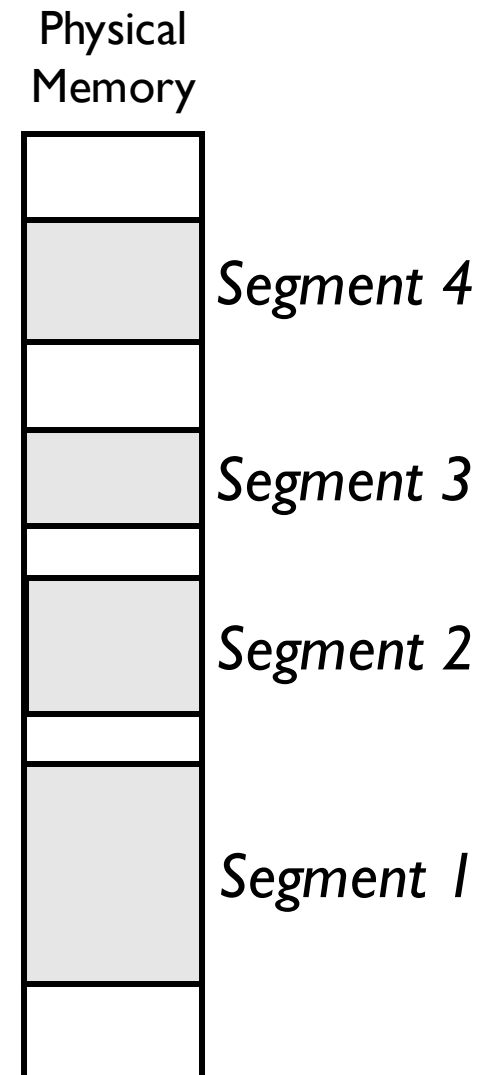*Segment 2*

Segment 1

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..

Physical Memory

Segment 4

Segment 3
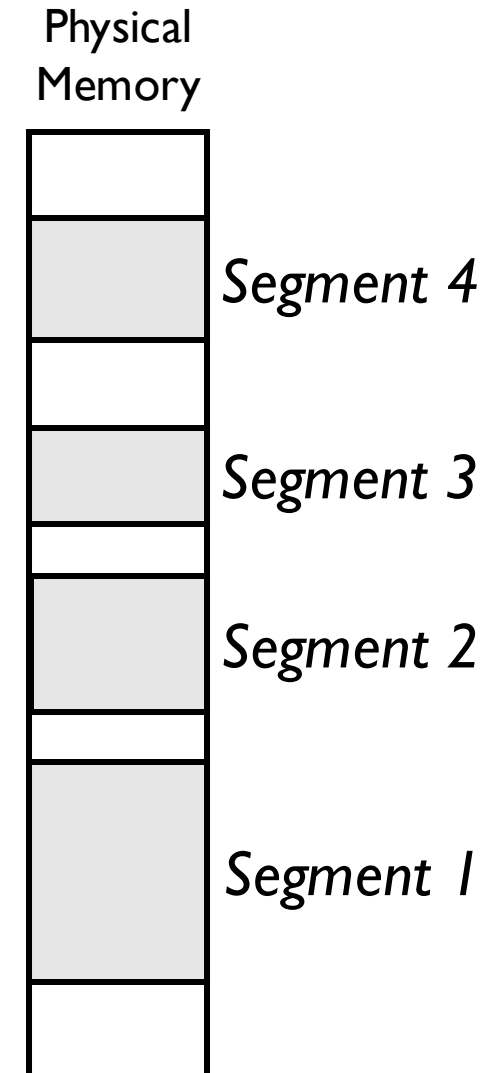
Segment 2

Segment 1

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 x86 view of memory segmentation (each 16-bits long)
  - Code segment: CS
  - Data segment: DS
  - Stack segment: SS
  - Extra segment: ES, FS GS

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

- 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

Segmented Memory

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

`0000 0110 1110 1111 0000`	**Segment**, 16 bits, shifted 4 bits left (or multiplied by 0x10)
+ `0001 0010 0011 0100`	**Offset**, 16 bits
`0000 1000 0001 0010 0100`	**Address**, 20 bits

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

Segment Selector

15		3	2	1	0
Index			TI	RPL	

Added for 32-bit protected mode (80386)

31	24	23			20	19	16	15		12	11		8	7	0
Base address (24-31)		G	DB		A	Limit (16-19)		P	DPL	S	Type			Base address (16-23)	
Base address (Bit 0-15)								Segment Limit (Bit 0-15)							

A segment descriptor
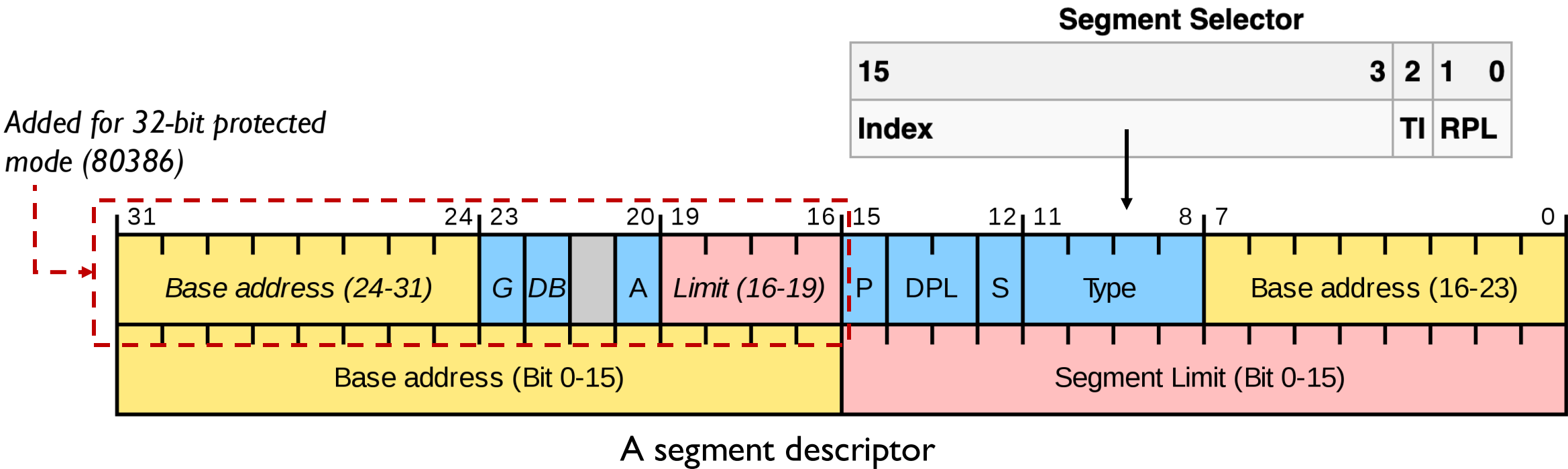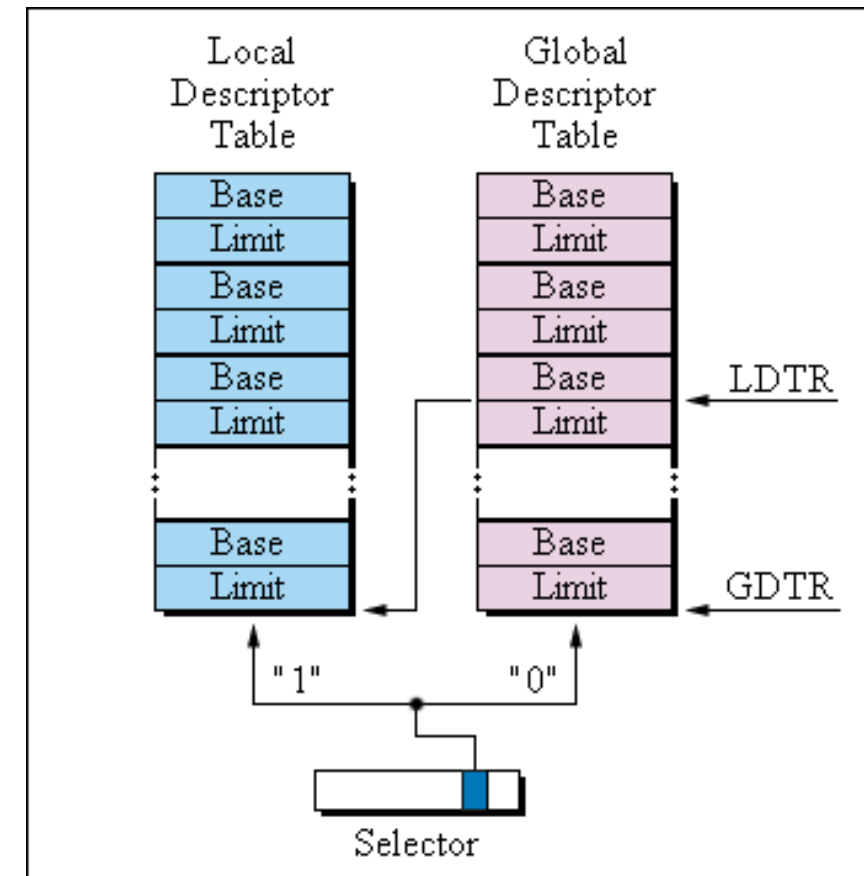
Segmented Memory

- 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
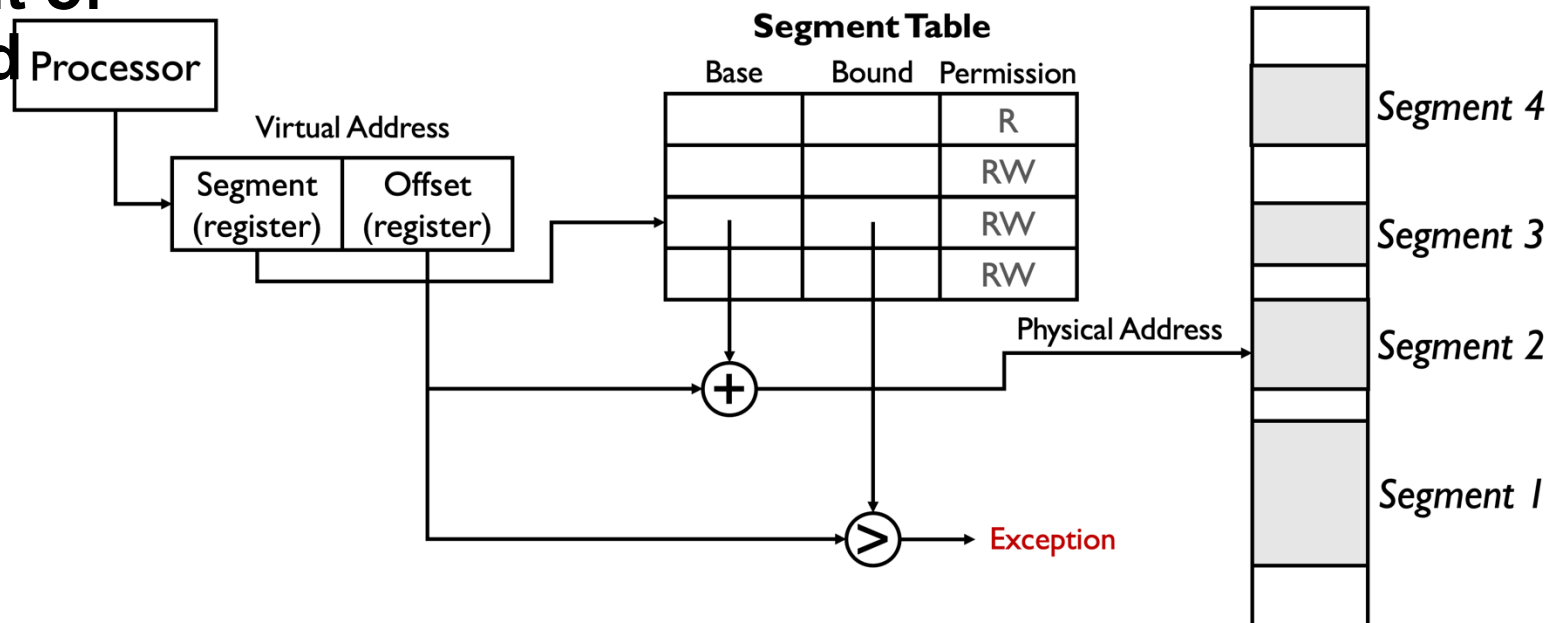
Segmented Memory

- **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
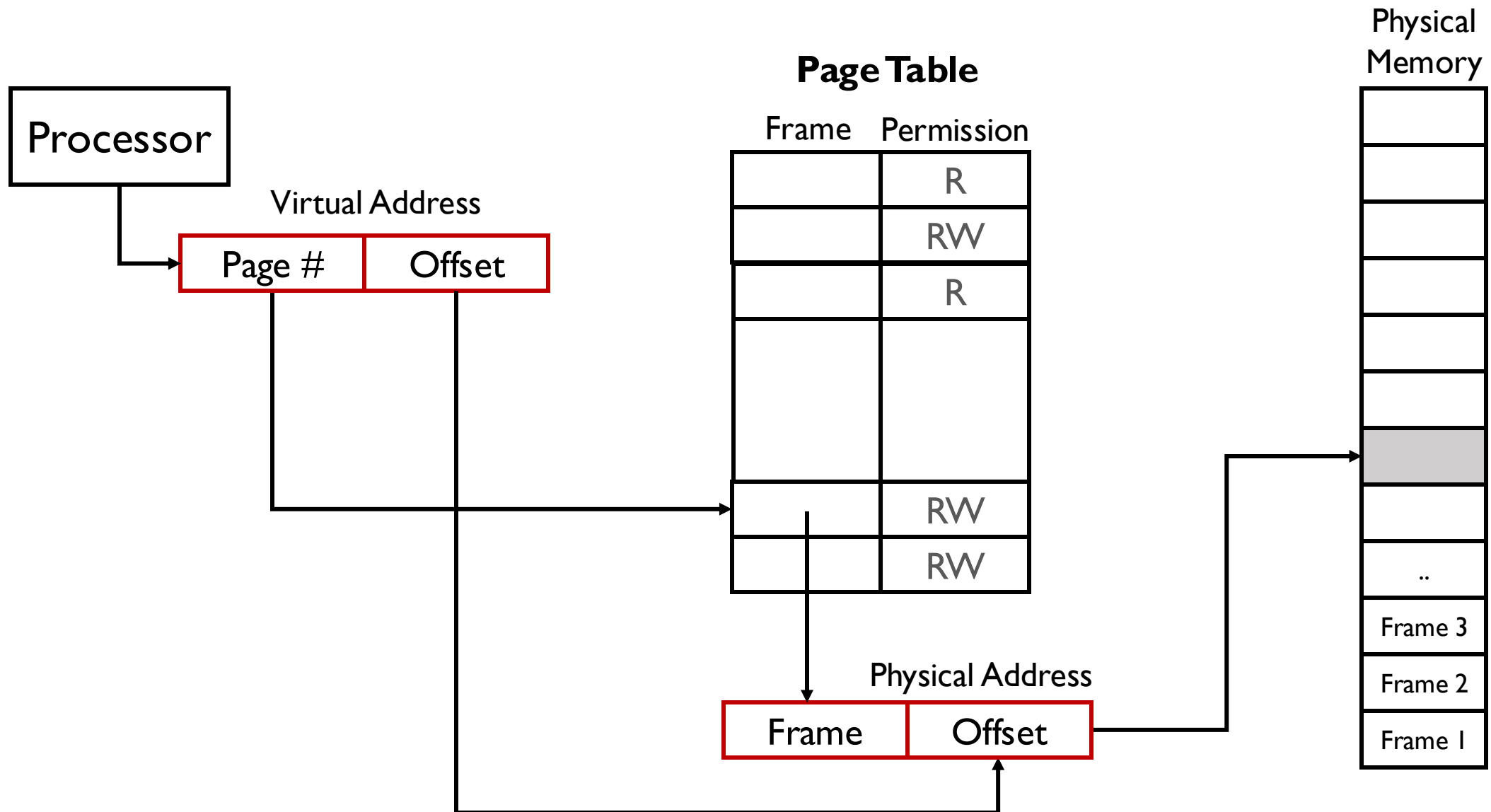 - It becomes even more complex if the segments can grow (like heap)

- 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

Processor

Virtual Address

Page # (20 bits)	Offset (12 bits)

How large is the page table?

Page Table

Frame   Permission

	R
	RW
	R
Each entry takes 32 bits	
	RW
	RW

Physical Address

Frame	Offset

Physical Memory

4K
4K
4K
..
..
Frame 3
Frame 2
Frame 1

Paged Memory

One-level page table for 32-bit addressing

Processor

Virtual Address

Page # (20 bits)	Offset (12 bits)

Page Table

Frame   Permission

	R
	RW
	R
Each entry takes 32 bits	
	RW
	RW

2^20 * 4B = 4MB

Physical Address

Frame	Offset

Physical Memory

4K
4K
4K
..
..
Frame 3
Frame 2
Frame 1
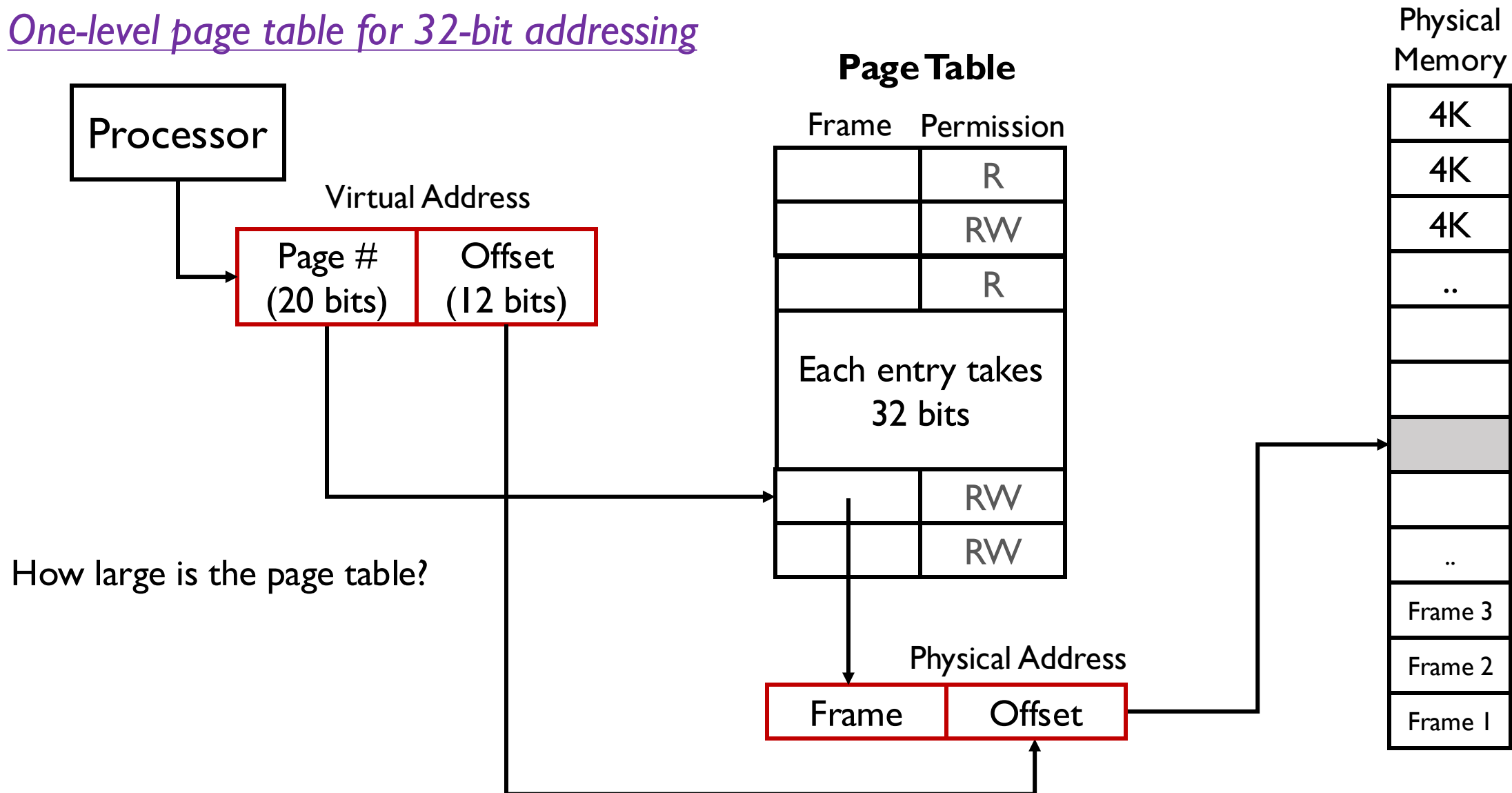
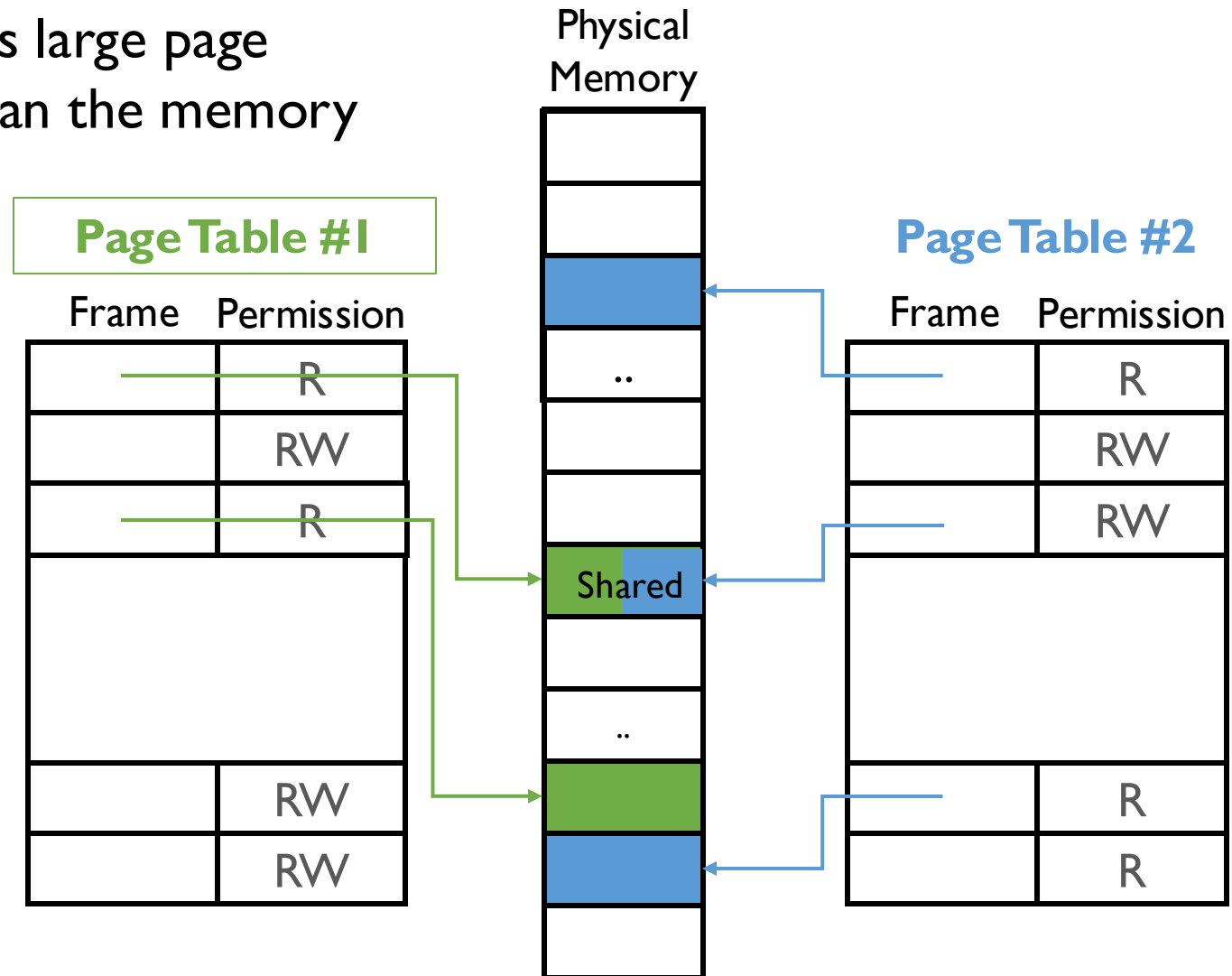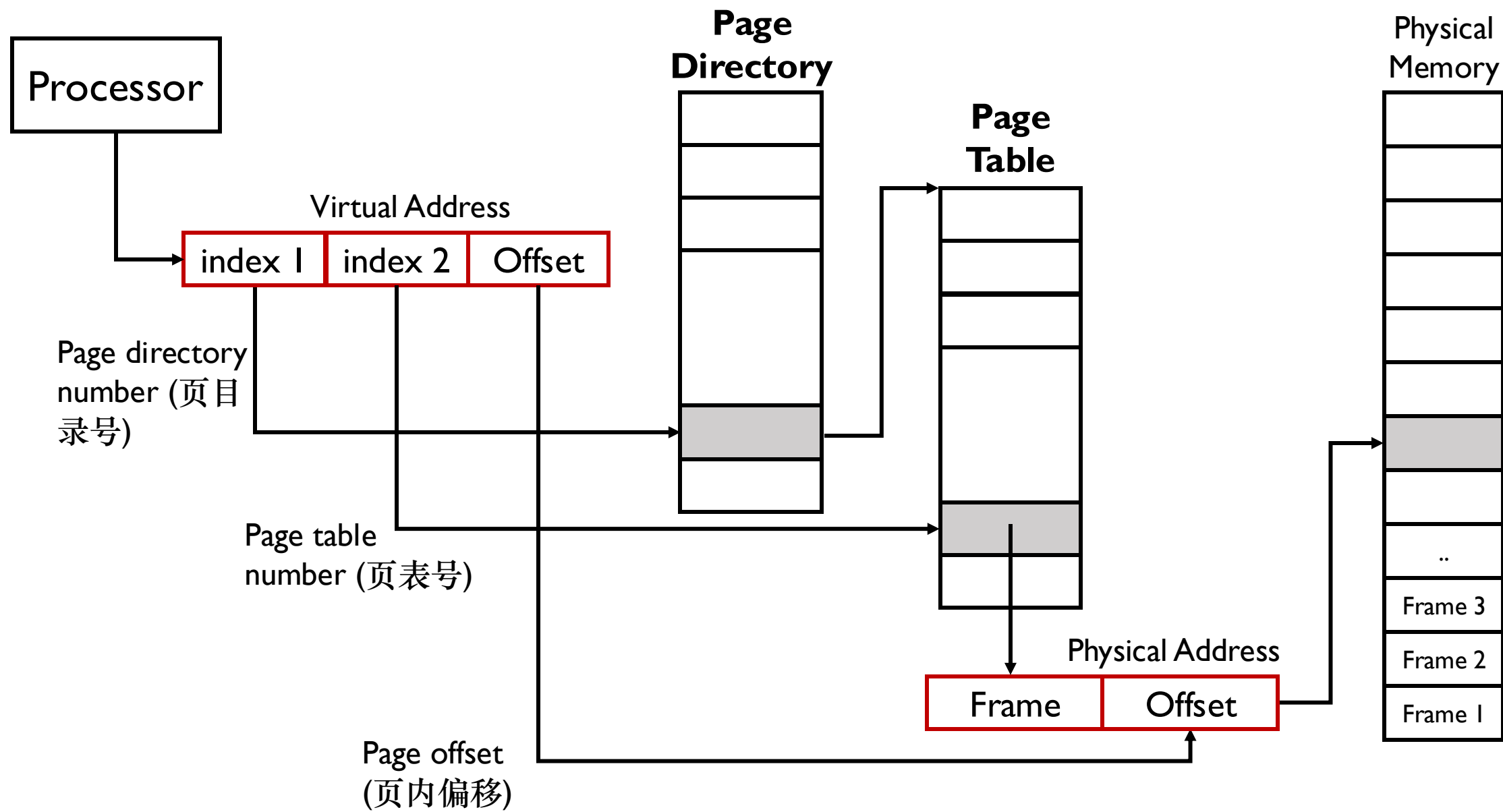How large is the page table?
- This is per process..
- And 64-bit addressing..

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

Processor

Page Directory

Page Table

Physical Memory

Virtual Address

index 1	index 2	Offset

Page directory number (页目录号)

Page table number (页表号)

Physical Address

Frame	Offset

Page offset (页内偏移)

..

Frame 3

Frame 2

Frame 1

x86 Multi-level Paging

Processor

CR3

Page Directory

- - - - → 4K size

Page Table

Physical Memory

Virtual Address

| index 1 (10 bits) | index 2 (10 bits) | Offset (12 bits) |

Page directory number (页目录号)

Page table number (页表号)

Page offset (页内偏移)

32 bits

32 bits

| 4K |
| 4K |
| 4K |
| .. |
| |
| |
| |
| |
| .. |
| Frame 3 |
| Frame 2 |
| Frame 1 |

Physical Address

| Frame (20 bits) | Offset (12 bits) |

x86 Multi-level Paging

Example (4 byte pages)

Virtual Memory

0x00	a
	b
	c
	d
0x04	e
	f
0x06?	g
	h
0x08	i
0x09?	j
	k
	l

0000 0000

0000 0100

0000 1000

Page Table

0	4
1	3
2	1

0001 0000

0000 0100

Physical Memory

0x00	
0x04	i
	j
	k
	l
0x08	
0x0C	e
	f
	g
	h
0x10	a
	b
	c
	d

0x05!

0x0E!

0000 0110 ----> 0000 1110

0000 1001 ----> 0000 0101
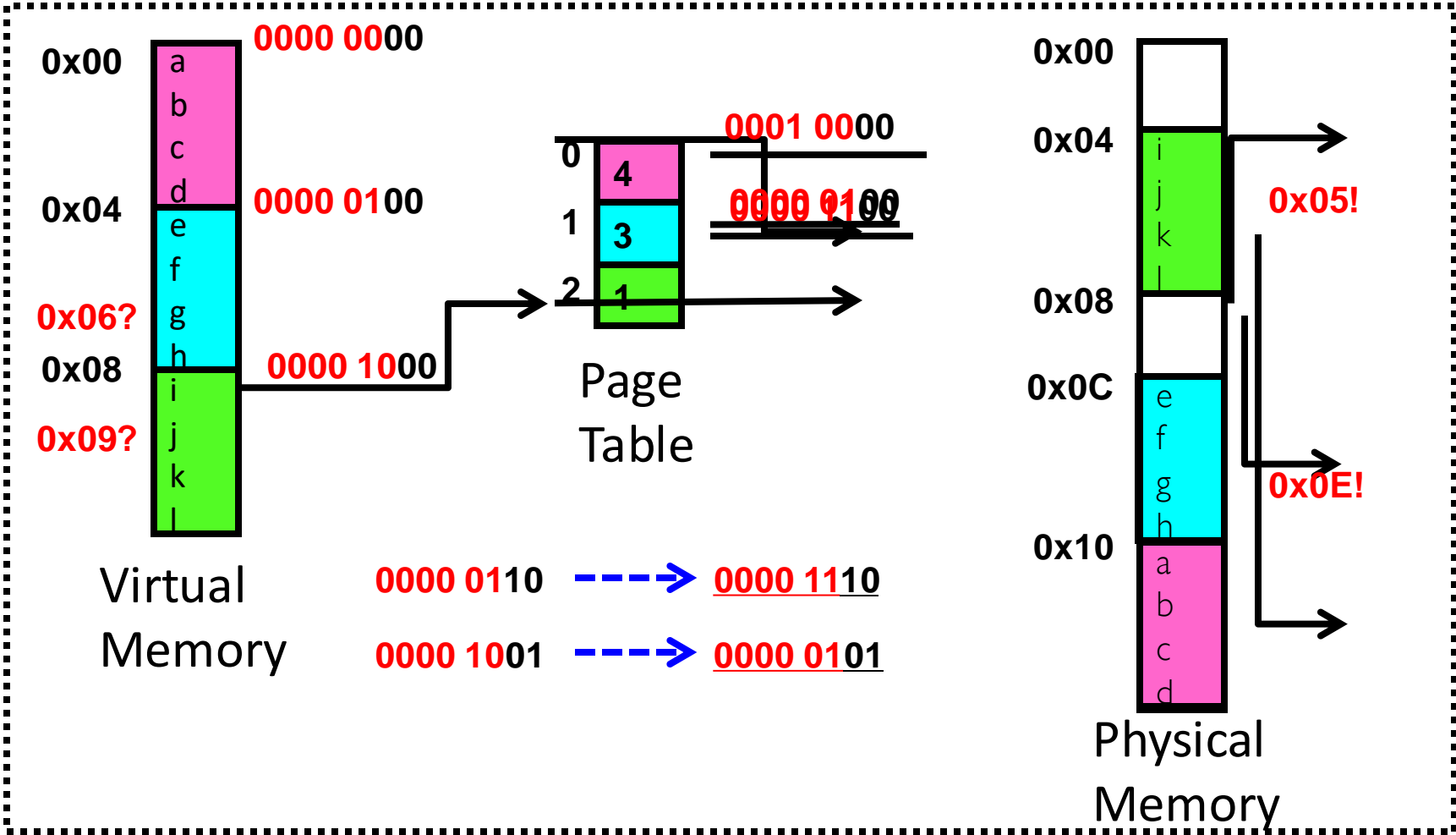
x86 Multi-level Paging

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

31	11	9	8	7	6	5	4	3	2	1	0
Page-Table Base Address (12-31)	Avail (9-11)	G	P S	0	A	P C D	P W Y	U / S	R / W	P

Available for system programmer's use
Global page (Ignored)
Page size (0 indicates 4 KB)
Reserved (set to 0)
Accessed
Cache disabled
Write-through
User/Supervisor
Read/Write
Present
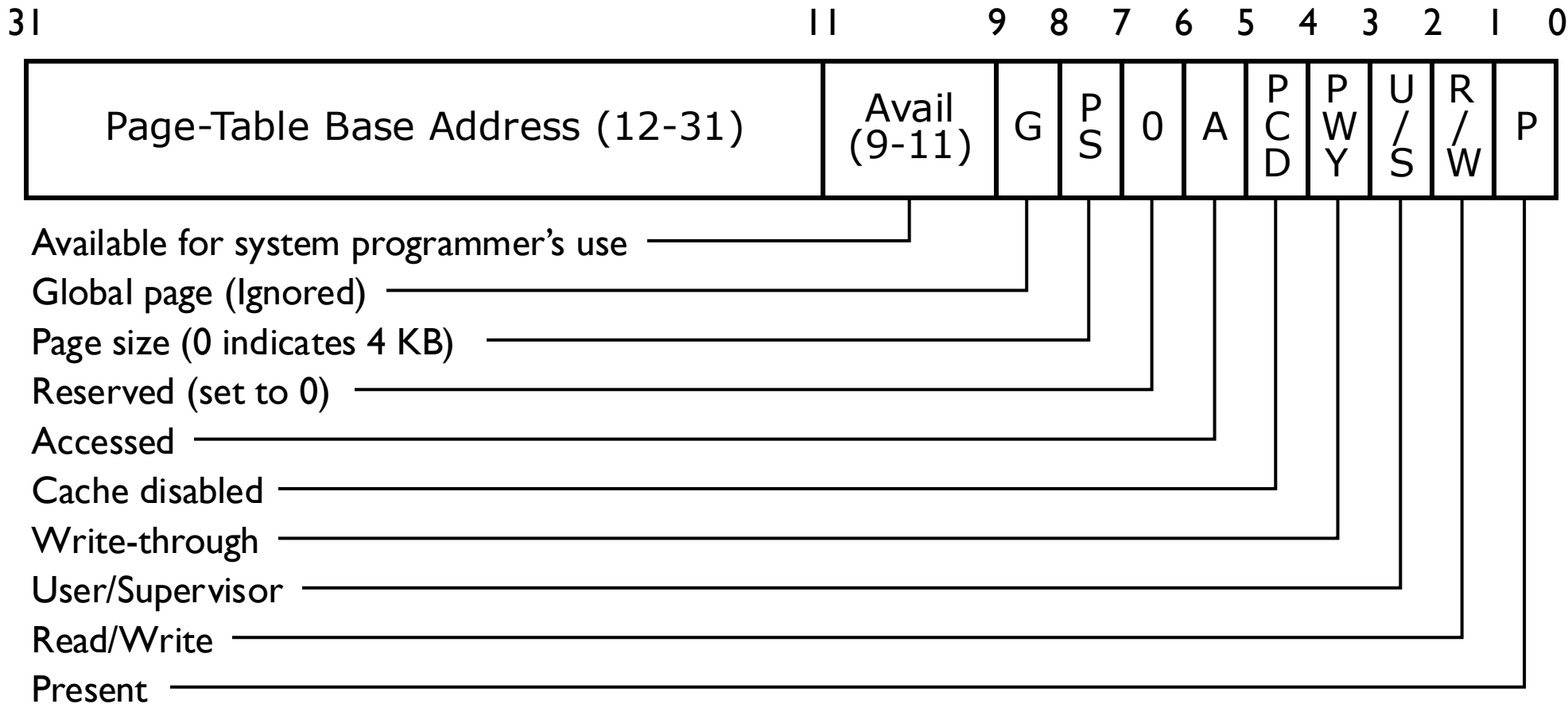
x86 Multi-level Paging

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

31	11	9	8	7	6	5	4	3	2	1	0
Page Frame Base Address (12-31)	Avail (9-11)	G	PAT	D	A	PCD	PWY	U/S	R/W		P

Available for system programmer's use
Global page
Page Table Attribute Index
Dirty
Accessed
Cache disabled
Write-through
User/Supervisor
Read/Write
Present

- Memory management unit (MMU, 分页内存管理单元): the hardware that actually does the translation
 - Usually located in CPU
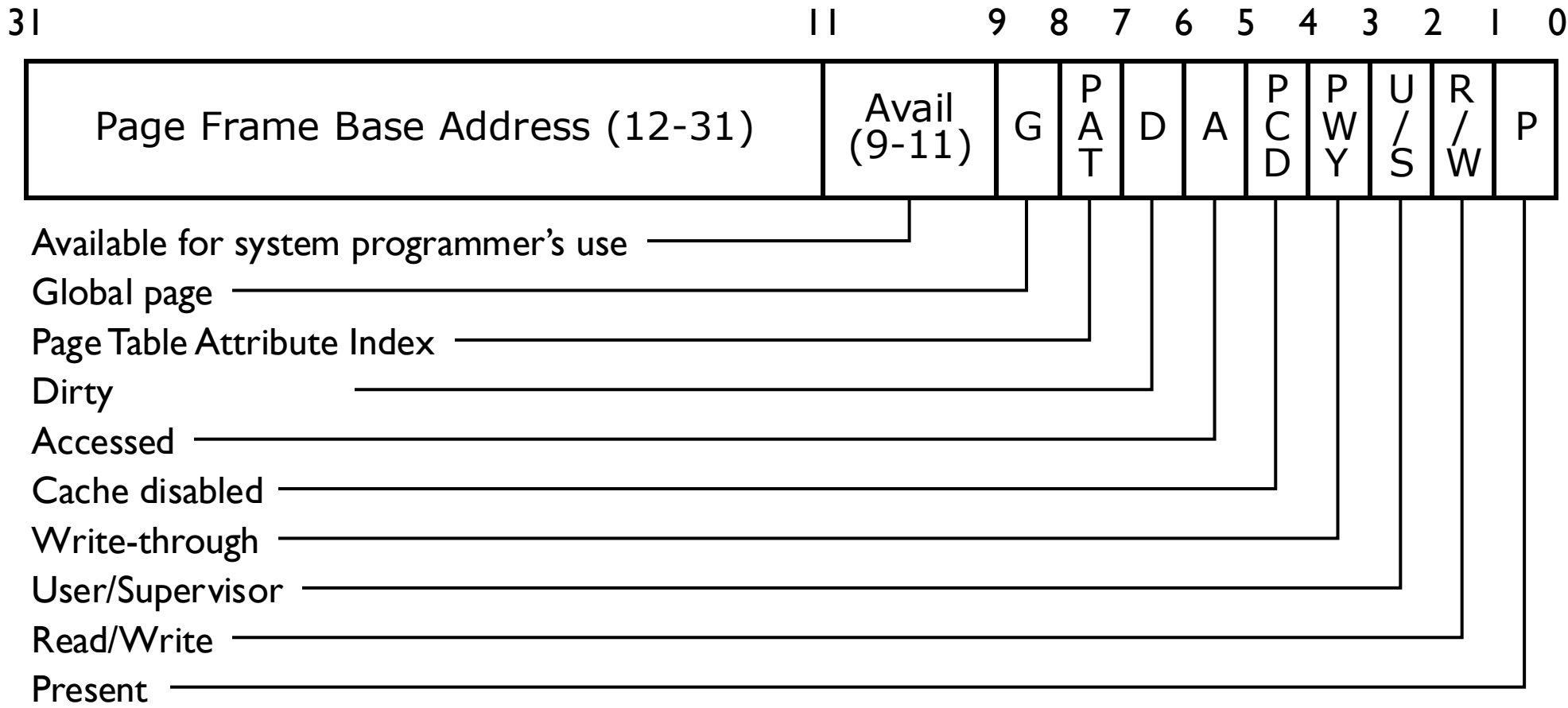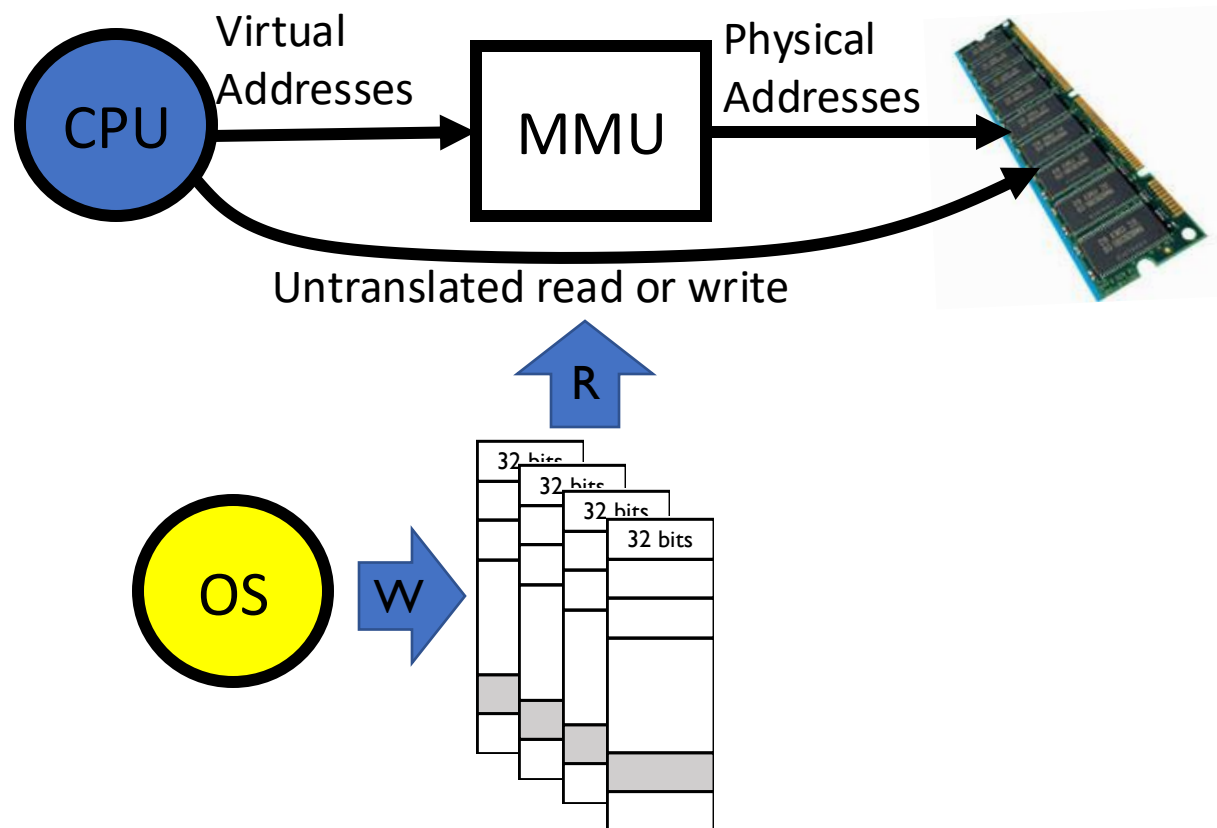
x86 Multi-level Paging

- 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.

- 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..
  - In Linux, there is only one kernel space for all process

- 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 Directory

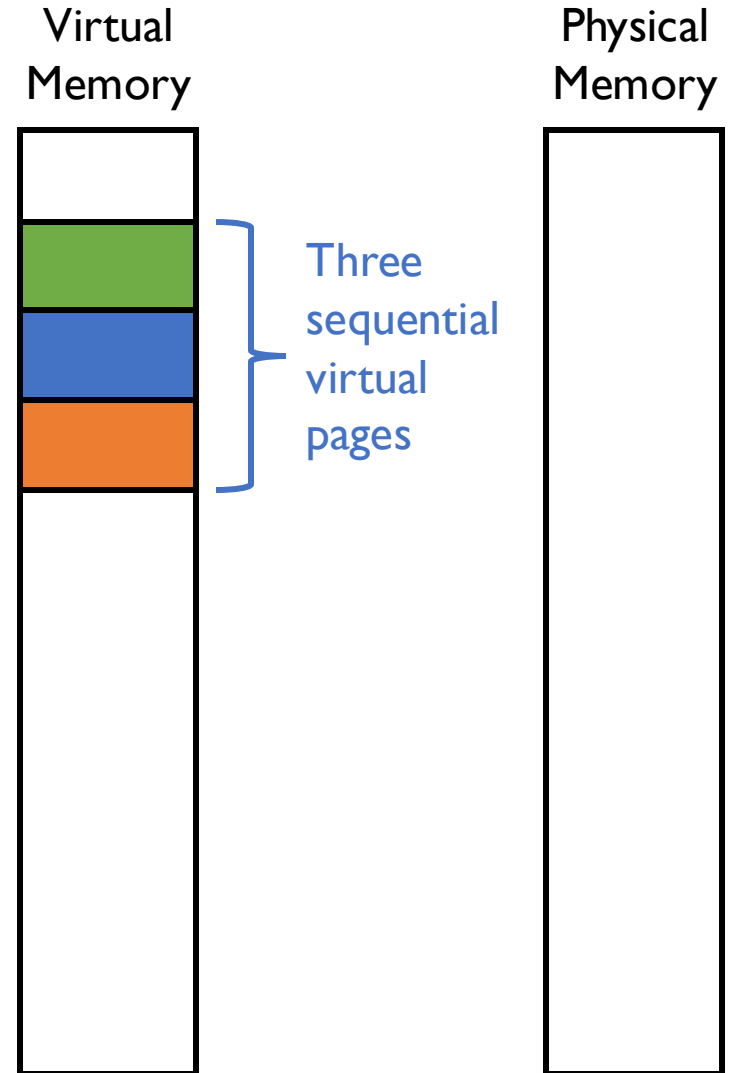| |
|---|
| P=0 |
| P=0 |
| P=1 |
| |
| |
| P=1 |
| P=0 |

Page Table

32 bits

32 bits
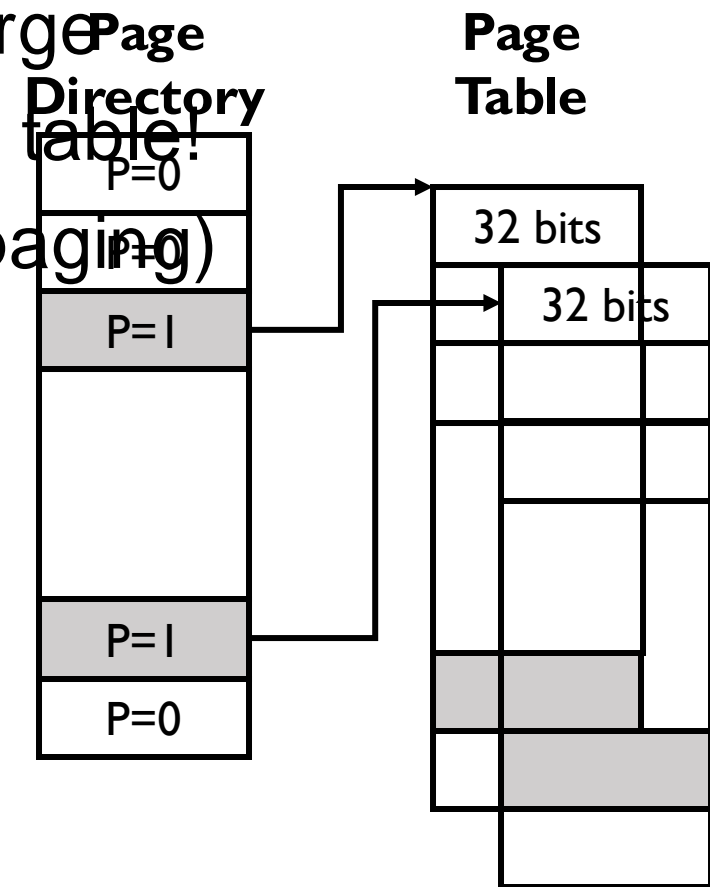
Page Fault

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

- In modern OSes, malloc does memory allocation "lazily"
  - It allocates virtual memory immediately
  - The physical memory is allocated only when program accesses that memory through page fault handler
  - Why?

Page Fault

```c
#include <stdio.h>

void main() {
    int* x = (int *) malloc(4096*3);
    // ..
    x[0] = 1;
    // ..
    x[4096] = 2;
    // ..
    return;
}
```
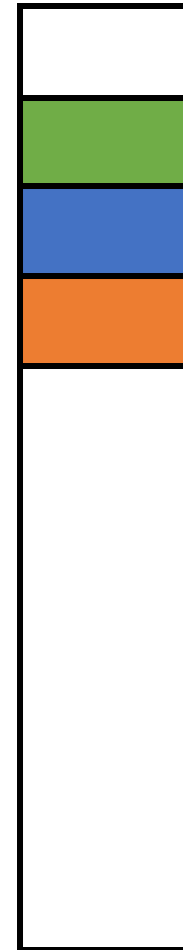
Virtual Memory

Physical Memory

Three sequential virtual pages

Page Fault

```
#include <stdio.h>

void main() {
    int* x = (int *) malloc(4096*3);
    // ..
▸   x[0] = 1;
    // ..
    x[4096] = 2;
    // ..
    return;
}
```

Virtual Memory

Physical Memory

1. Page fault
2. Physical page allocation
3. Page mapping

Page Fault

```c
#include <stdio.h>

void main() {
    int* x = (int *) malloc(4096*3);
    // ..
    x[0] = 1;
    // ..
▸   x[4096] = 2;
    // ..
    return;
}
```
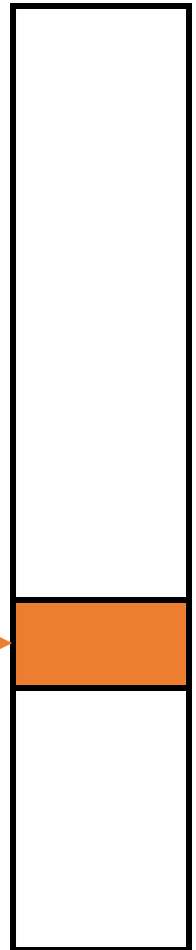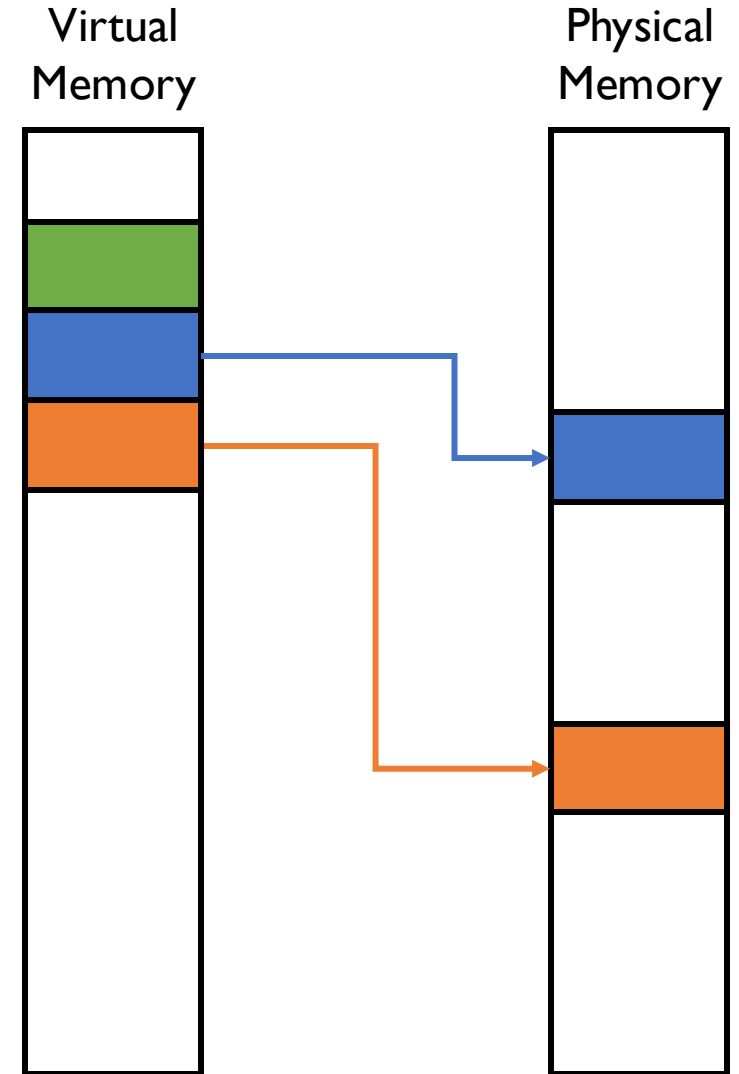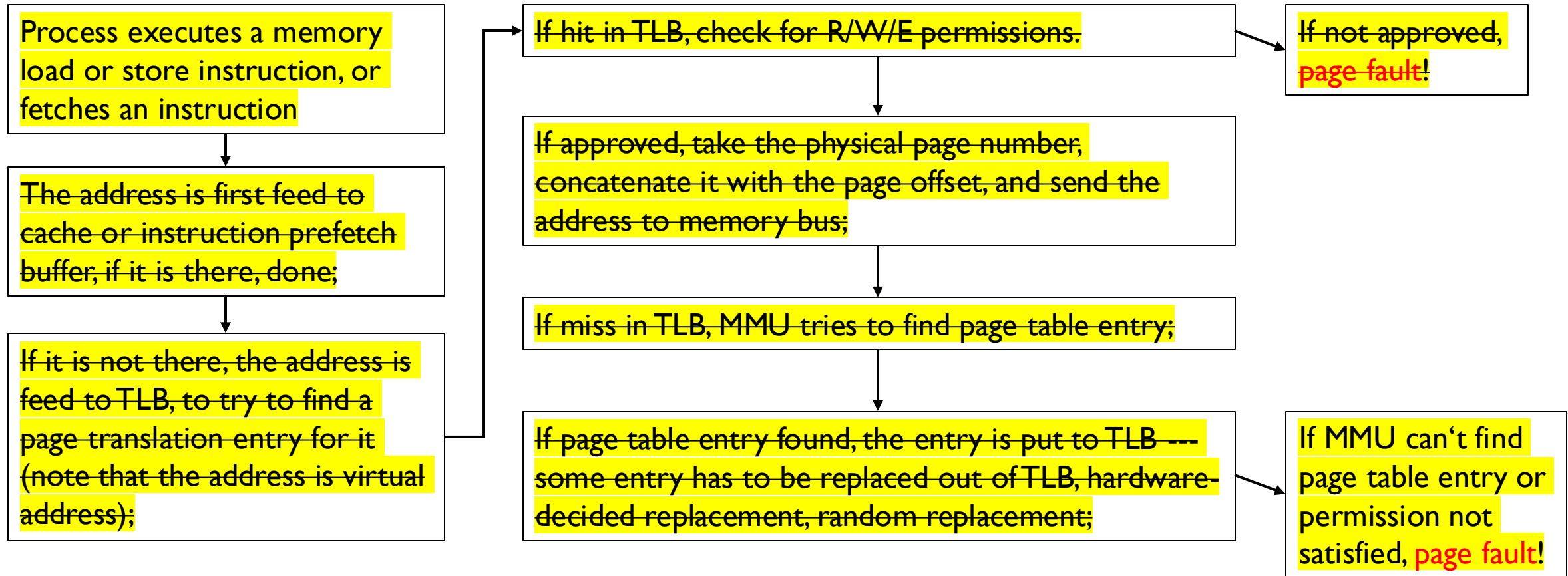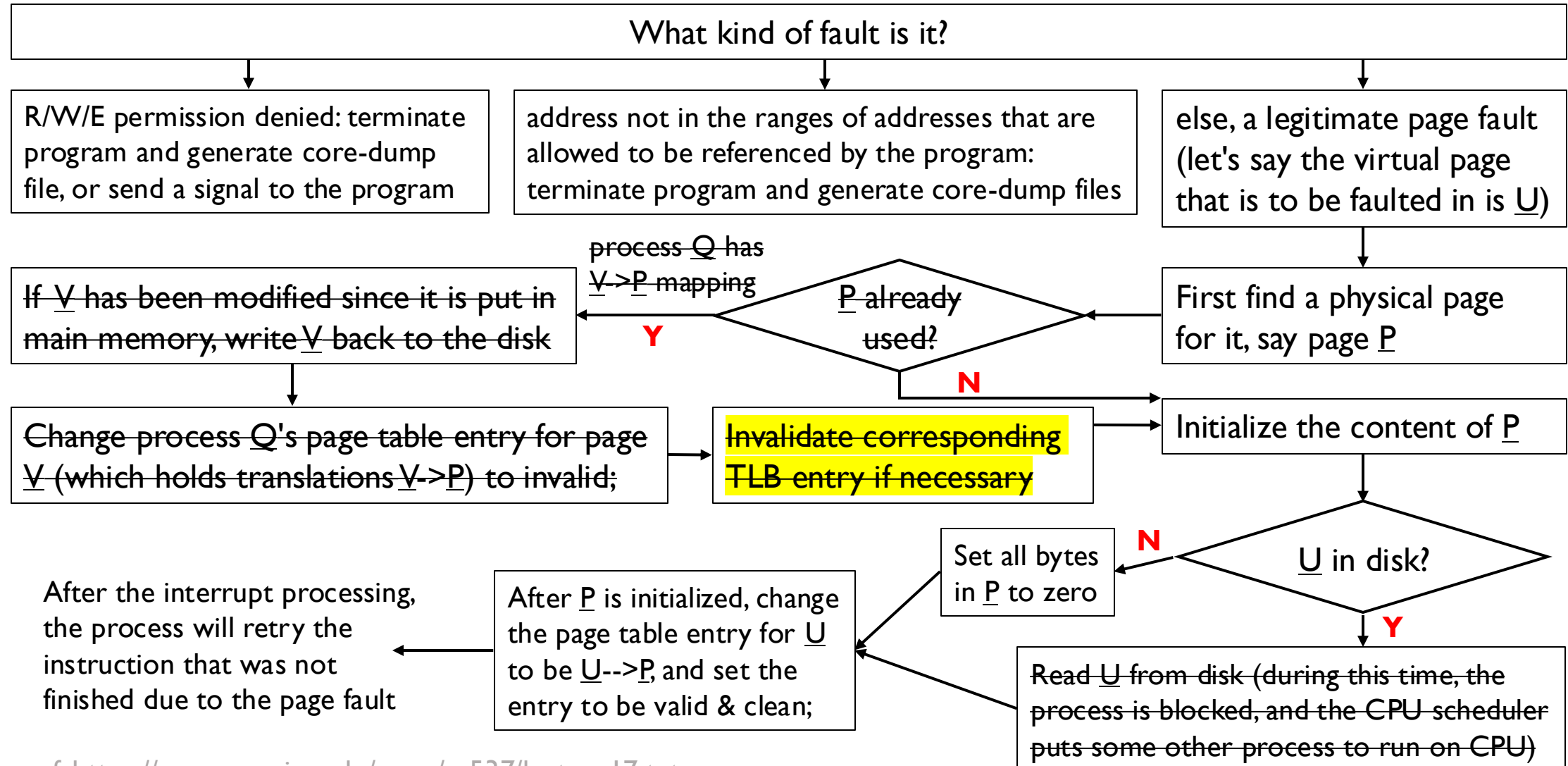
Virtual Memory

Physical Memory

Detailed Page Fault Process

Before Page Fault (done by hardware)

Process executes a memory load or store instruction, or fetches an instruction

The address is first feed to cache or instruction prefetch buffer, if it is there, done;

If it is not there, the address is feed to TLB, to try to find a page translation entry for it (note that the address is virtual address);

If hit in TLB, check for R/W/E permissions.

If approved, take the physical page number, concatenate it with the page offset, and send the address to memory bus;

If miss in TLB, MMU tries to find page table entry;

If page table entry found, the entry is put to TLB --- some entry has to be replaced out of TLB, hardware-decided replacement, random replacement;

If not approved, page fault!

If MMU can't find page table entry or permission not satisfied, page fault!

Detailed Page Fault Process

Handling Page Fault (done by hardware)

What kind of fault is it?

R/W/E permission denied: terminate program and generate core-dump file, or send a signal to the program

address not in the ranges of addresses that are allowed to be referenced by the program: terminate program and generate core-dump files

else, a legitimate page fault (let's say the virtual page that is to be faulted in is U)

First find a physical page for it, say page P

~~process Q has V->P mapping~~

P already used?

Y ~~If V has been modified since it is put in main memory, write V back to the disk~~

N

~~Change process Q's page table entry for page V (which holds translations V->P) to invalid;~~

Invalidate corresponding TLB entry if necessary

Initialize the content of P

U in disk?

N Set all bytes in P to zero

Y

~~Read U from disk (during this time, the process is blocked, and the CPU scheduler puts some other process to run on CPU)~~

After P is initialized, change the page table entry for U to be U-->P, and set the entry to be valid & clean;

After the interrupt processing, the process will retry the instruction that was not finished due to the page fault

ref: https://pages.cs.wisc.edu/~cao/cs537/lecture17.txt
Mengwei Xu @ BUPT

Detailed Page Fault Process

During Page Fault (done by hardware)

- for a legitimate page fault,.
    - First find a physical page for it, say page P; If page P currently holds virtual page V of process Q, is the virtual page V modified since it is put in main memory? if so, write V back to the disk (topic 3); otherwise, do nothing; Change process Q's page table entry for page V (which holds translations V->P) to invalid; * invalidate the corresponding TLB entry if necessary;

- 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 + 1 page for its data)

- 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?

- The pointers used by kernel is virtual or physical?

- How can kernel manipulate the page directory/tables?

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

- Since OS only sees the virtual address, how can it manipulate the page table, e.g., getting the physical address of a given virtual address
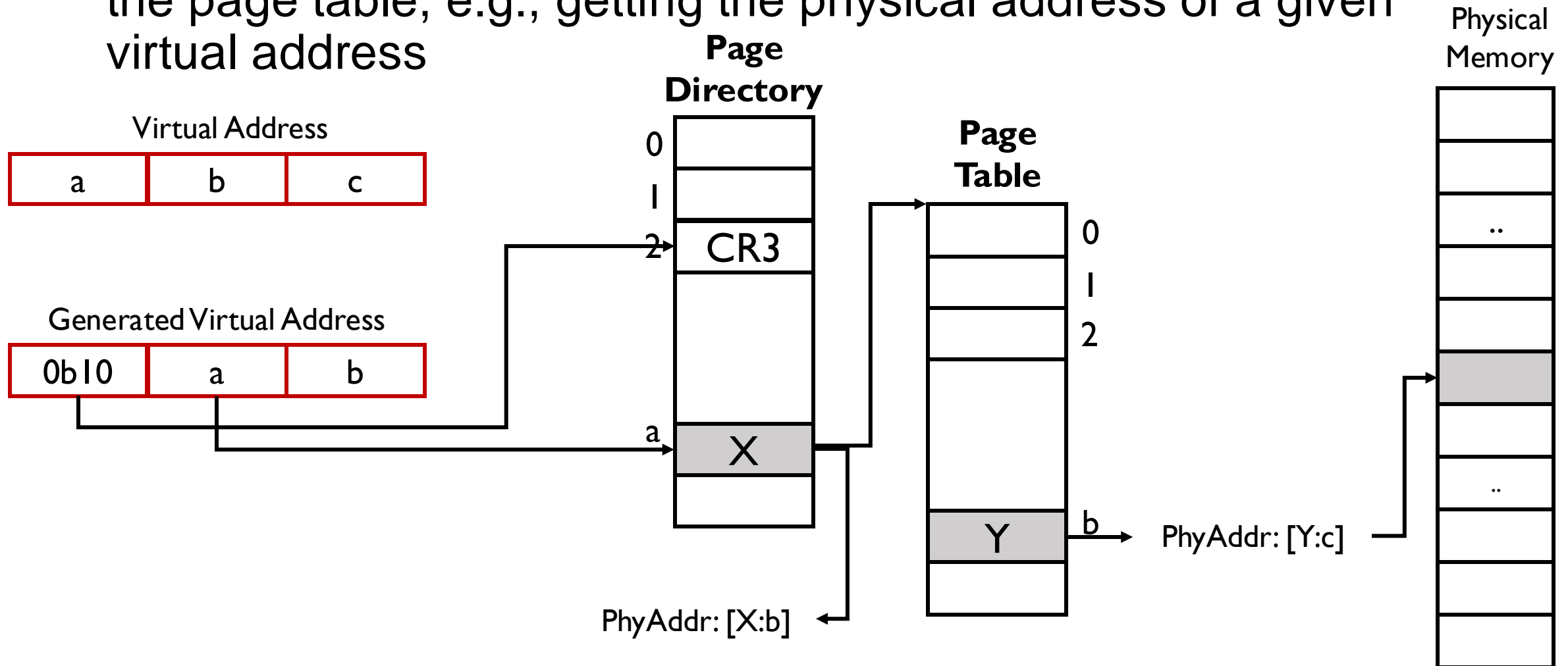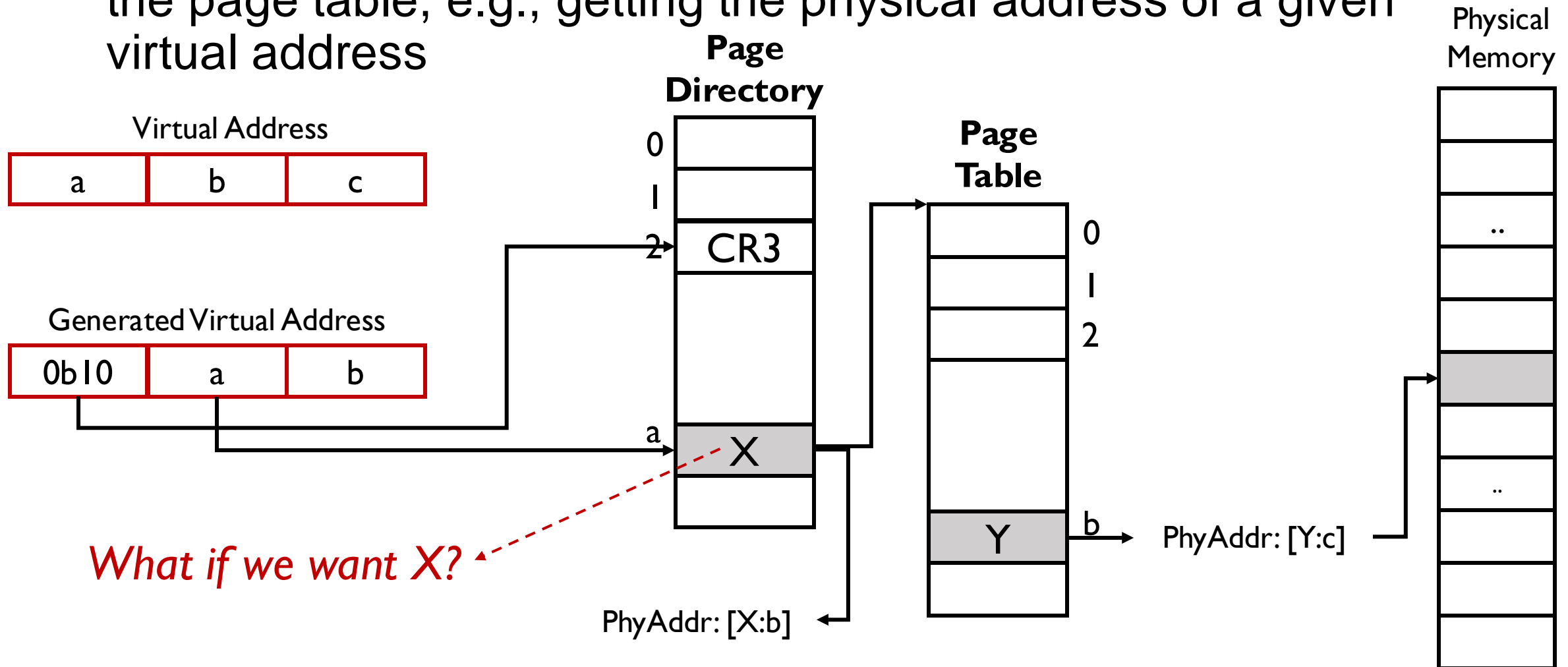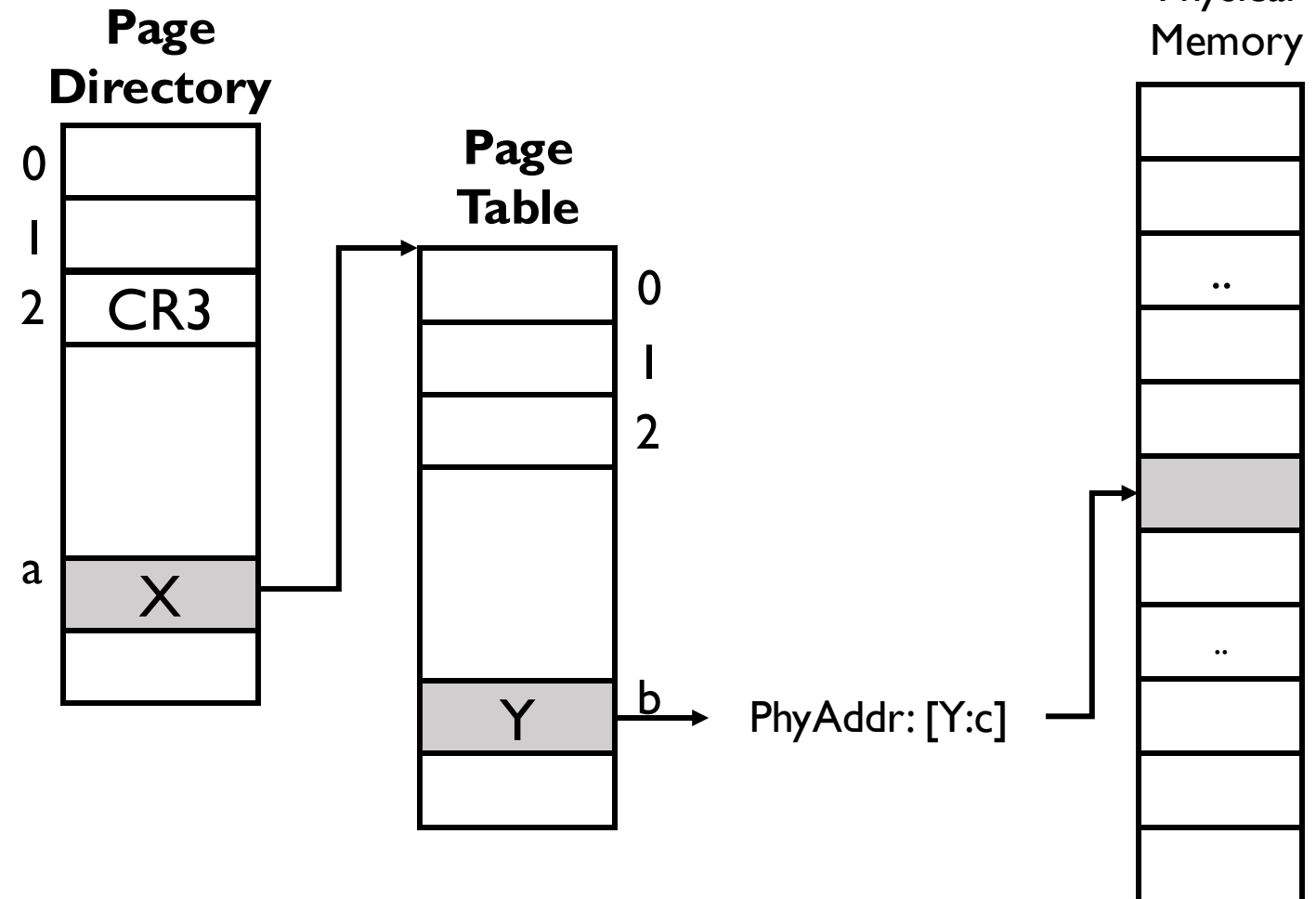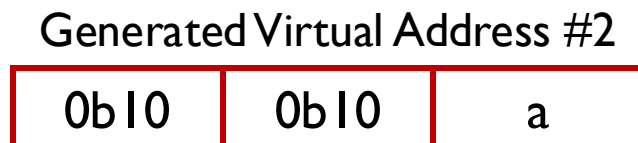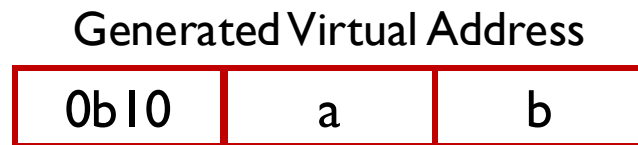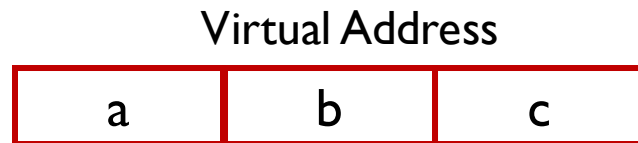
Manipulating Page Table

- Since OS only sees the virtual address, how can it manipulate the page table, e.g., getting the physical address of a given virtual address

Self-mapping

Virtual Address

a	b	c

Page Directory

0
1
2 CR3

a X

Page Table

0
1
2

b Y

PhyAddr: [Y:c]

Physical Memory

..

..

- Since OS only sees the virtual address, how can it manipulate the page table, e.g., getting the physical address of a given virtual address

Page Directory

Page Table

Physical Memory

Virtual Address

| a | b | c |

Generated Virtual Address

| 0b10 | a | b |

PhyAddr: [X:b]

PhyAddr: [Y:c]

- Since OS only sees the virtual address, how can it manipulate the page table, e.g., getting the physical address of a given virtual address

Physical Memory

Page Directory

Virtual Address

a	b	c

0

1

2 CR3

Generated Virtual Address

0b10	a	b

Page Table

0

1

2

a X

Y b → PhyAddr: [Y:c]

What if we want X?

PhyAddr: [X:b]

- Since OS only sees the virtual address, how can it manipulate the page table, e.g., getting the physical address of a given virtual address

Page Directory

Physical Memory

Page Table

Virtual Address

a	b	c

Generated Virtual Address

0b10	a	b

Generated Virtual Address #2

0b10	0b10	a

Page Directory:
0
1
2 CR3
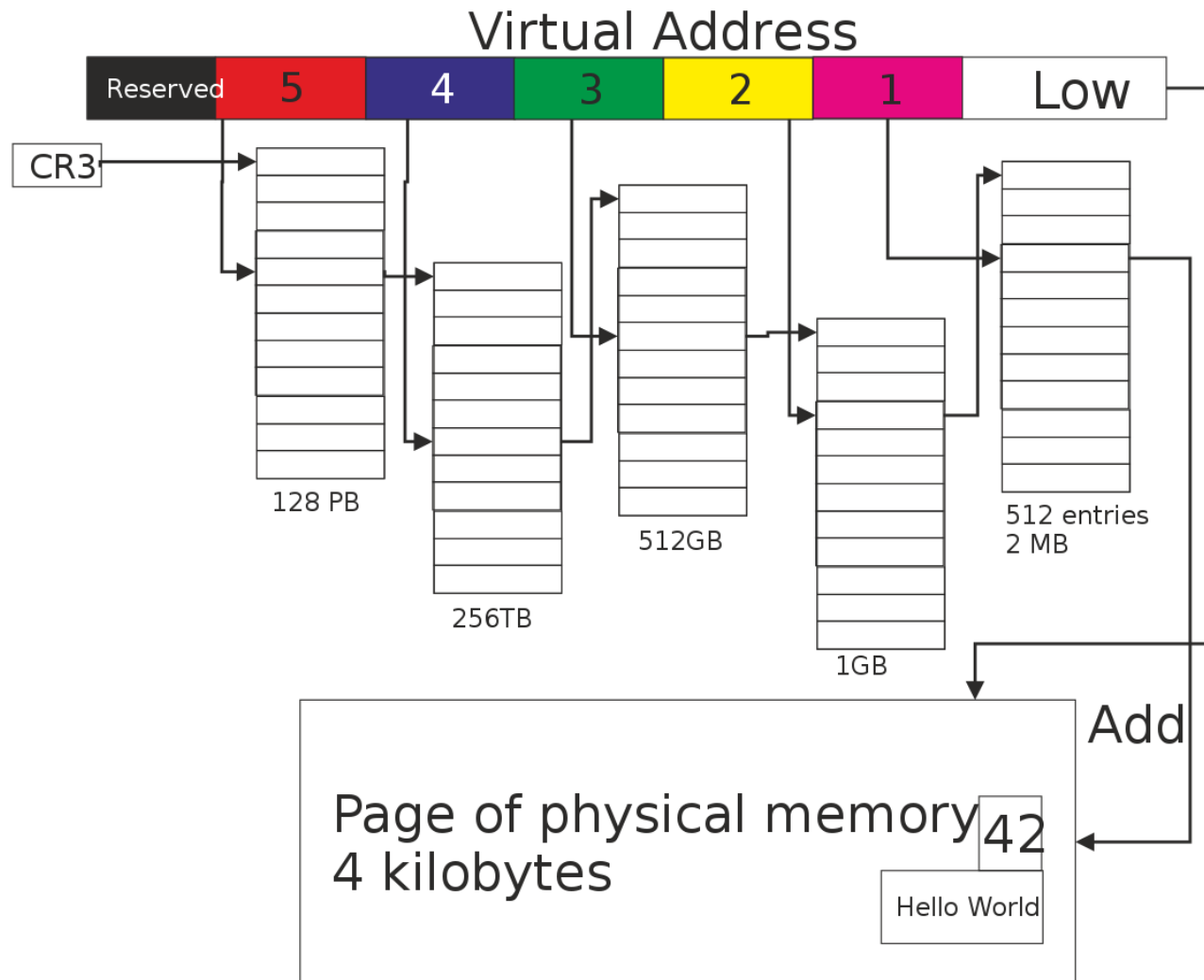a X

Page Table:
0
1
2
Y b → PhyAddr: [Y:c]

- 4-level: 48 bits

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

- 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 wards, 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):
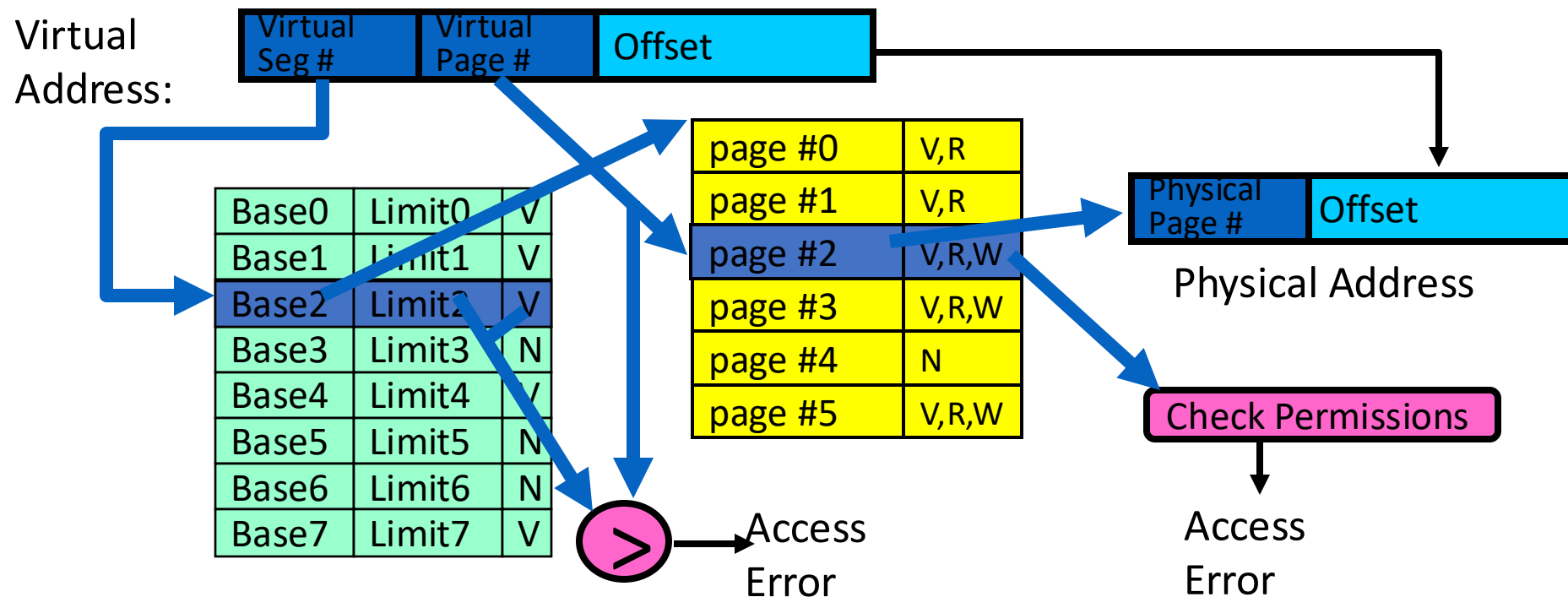
Segmentation vs. Paging

- ## 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

Copy-on-Write (COW)

- How to implement an efficient fork()?
  - Do not copy all contents immediately, but mark the page/segment tables of both child and parent processes as "read-only"
  - When a write (from either child or parent) happens, it traps into kernel through page fault, and a private page is copied.

- A fork() followed immediately by a exec(), how many pages are really copied?

Homework

- Look at the function get_physaddr in https://wiki.osdev.org/Paging#Manipulation, and explain how it works line by line.

- If the page size is 8K, how the address translation will work? Explain step by step in details, e.g., how the virtual address is splited, how large is the page directory and page table, etc..