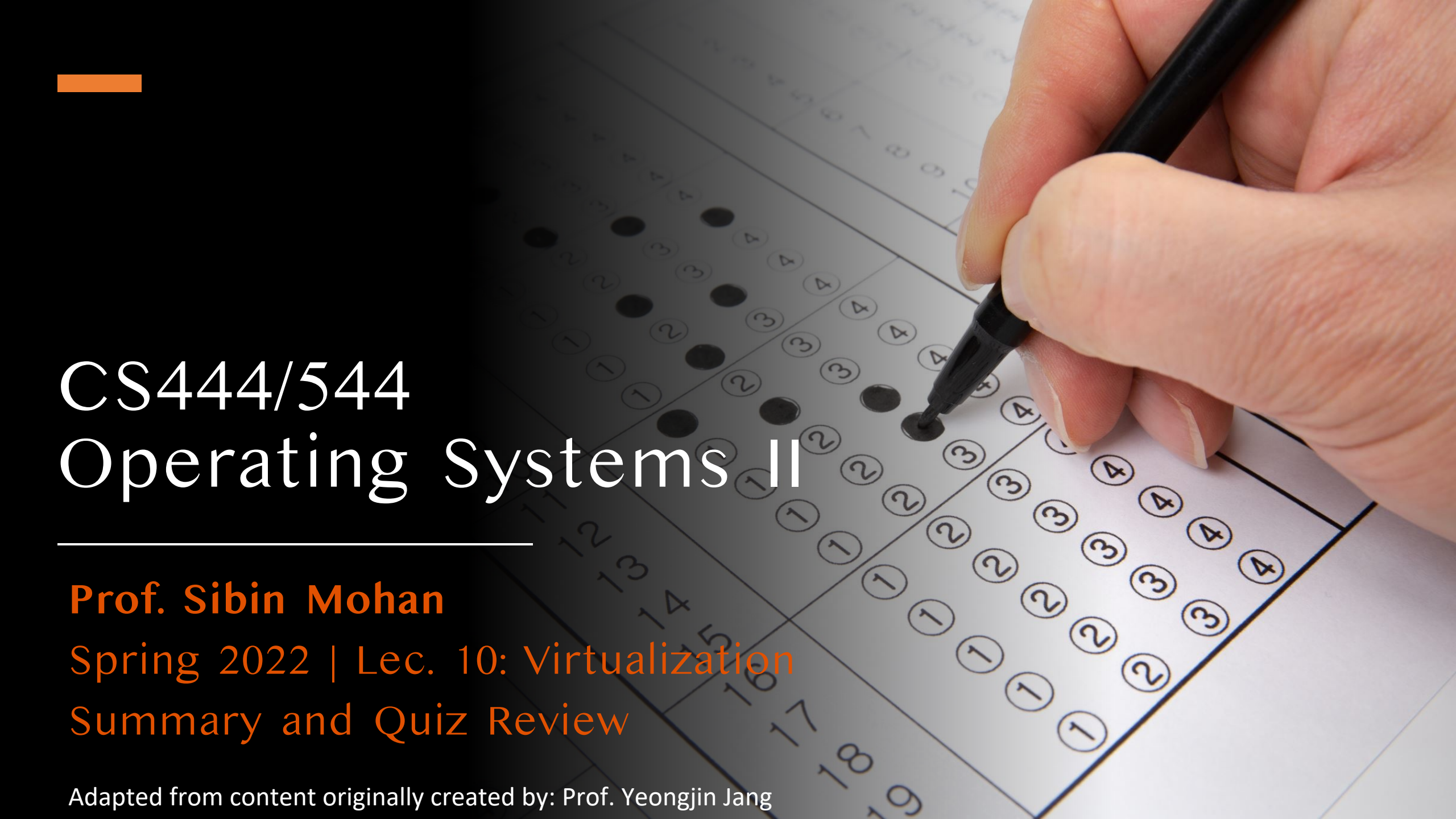


CS444/544 Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec. 10: Virtualization Summary and Quiz Review

Adapted from content originally created by: Prof. Yeongjin Jang

Administrivia

- Lab 2 due date: **May 6, 2022** [**Friday**] at **11:59 PM**!
  - Lots of office hours this week → USE THEM WELL!
- Quiz 2 on **May 5, 2022** [**Thursday**] at **8:30 AM**!
  - Available until **May 6, 2022** [**Friday**], **11:59 PM**
- Lab 3 announced [User Environment]
  - Due **May 13, 2022** [**Friday**] at **11:59 PM**
- Watch all **Tutorials** and go through the slides/textbook

Today's Topic

Virtualization → Concurrency

OS has **three** purposes

- **Virtualization**
  - memory, process, user/kernel, etc.
- **Concurrency**
  - multi-threading, multi-process, scheduling, synchronization
- **Persistence**
  - disk, file, snapshot, etc.

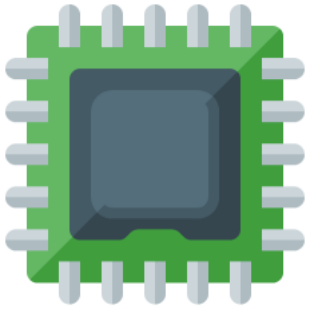
Recap on Virtualization

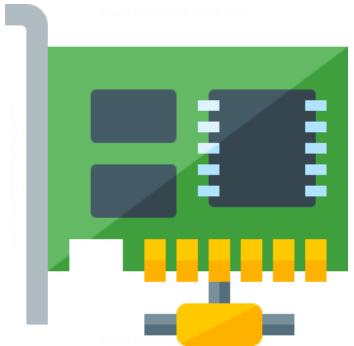
What is an OS?

Applications

OS

Hardware





Memory

- 8086 Segmentation – Real Mode
  - Address = seg * 16 + offset
- 80386 Segmentation – Protected Mode
  - GDT defines base and limit
  - seg selects a GDT entry
  - Address = GDT[seg].base + offset

31 16			15 0	
Base 0:15			Limit 0:15	
63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

Backward Compatibility

- BIOS assumes your CPU is an **i8086**
  - A 40+ year old CPU!
- Start with 16-bit mode → enable 32-bit mode → paging, etc.

- **Unified Extensible Firmware Interface** [**UEFI**]
  - Publicly available **interface** between OS and hardware written in C
  - BIOS replacement
  - Can boot **large disk partitions** [> 2 TB!]
  - **Network access, GUI, multi language**, etc.
  - **32-** or **64-bit**

Virtual Memory Recap

Paging

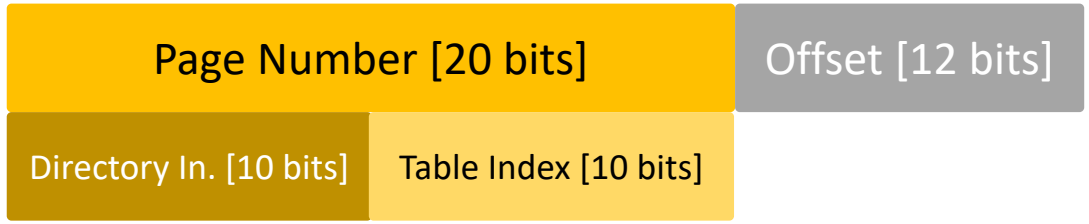
When enabled, all memory address will be translated

Memory Address Translation

Virtual Address

Memory Address Translation

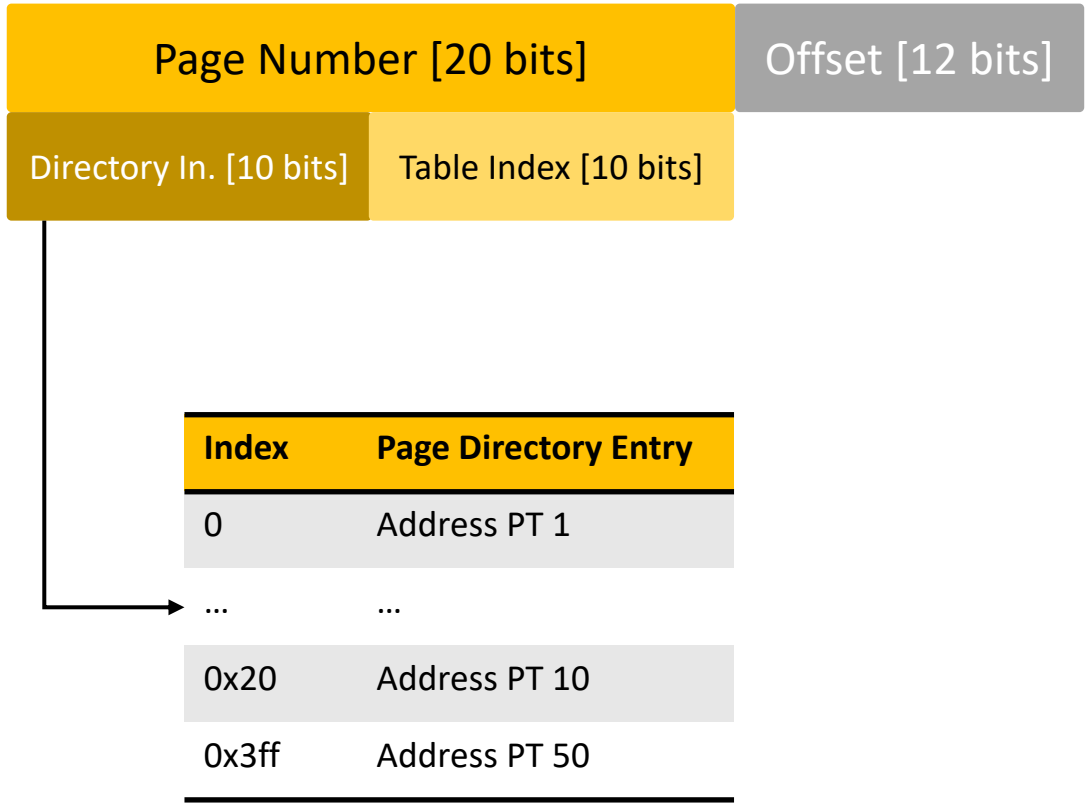
Virtual Address



Page Directory

Memory Address Translation

Virtual Address

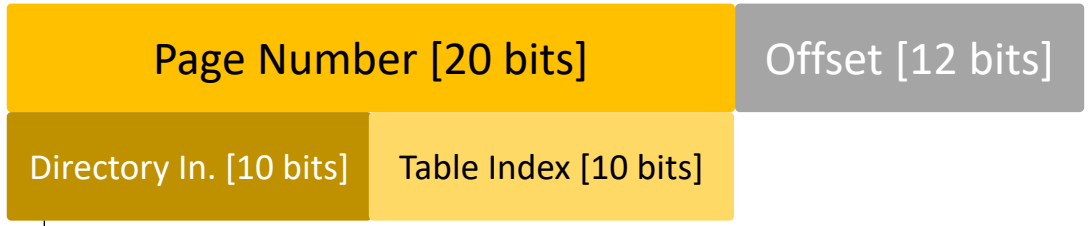


Index	Page Directory Entry
0	Address PT 1
…	…
0x20	Address PT 10
0x3ff	Address PT 50

Page Directory

Memory Address Translation

Virtual Address



Index	Page Directory Entry
0	Address PT 1
…	…
0x20	Address PT 10
0x3ff	Address PT 50

Page Directory

PT 1	Page Table Entry
0	NO MAPPING
…	…
0x30	Page Address x
0x49	Page Address y

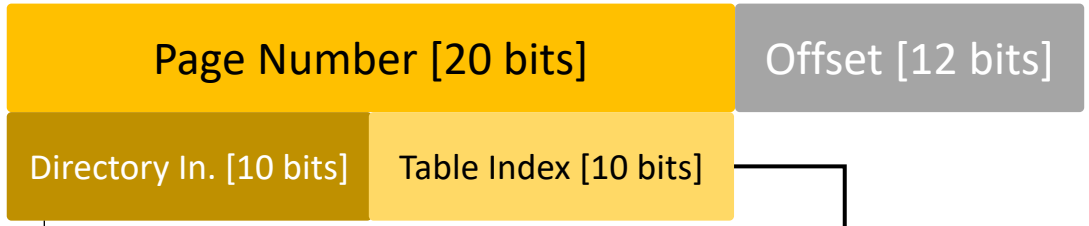
PT 10	Page Table Entry
0	NO MAPPING
…	…
0x48	0x10000
0x49	ox11000

PT 10	Page Table Entry
0	NO MAPPING
…	…
0xaa	Page address p
0xbb	Page address q

Page Table

Memory Address Translation

Virtual Address



Index	Page Directory Entry
0	Address PT 1
…	…
0x20	Address PT 10
0x3ff	Address PT 50

Page Directory

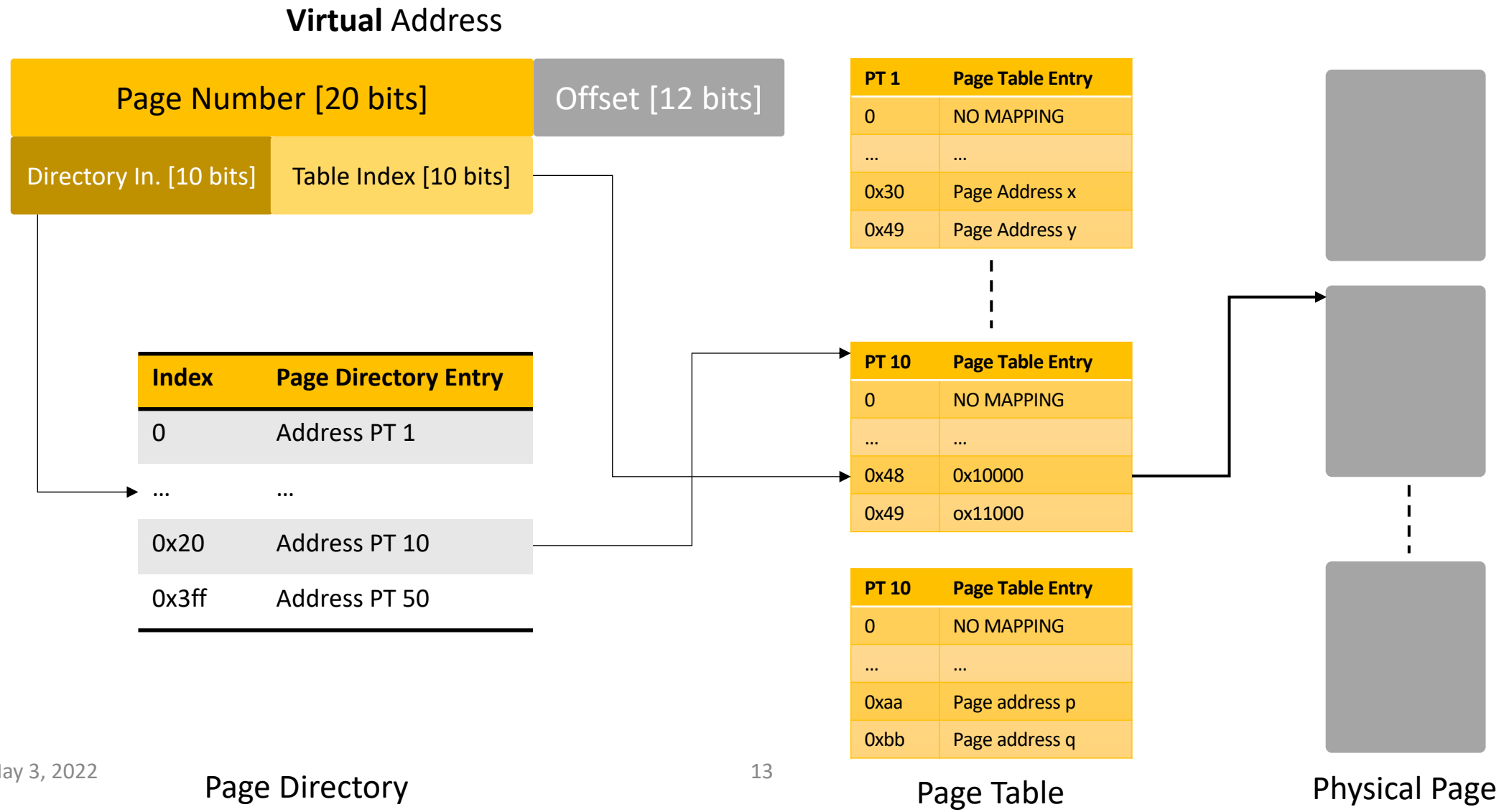
PT 1	Page Table Entry
0	NO MAPPING
…	…
0x30	Page Address x
0x49	Page Address y

PT 10	Page Table Entry
0	NO MAPPING
…	…
0x48	0x10000
0x49	ox11000

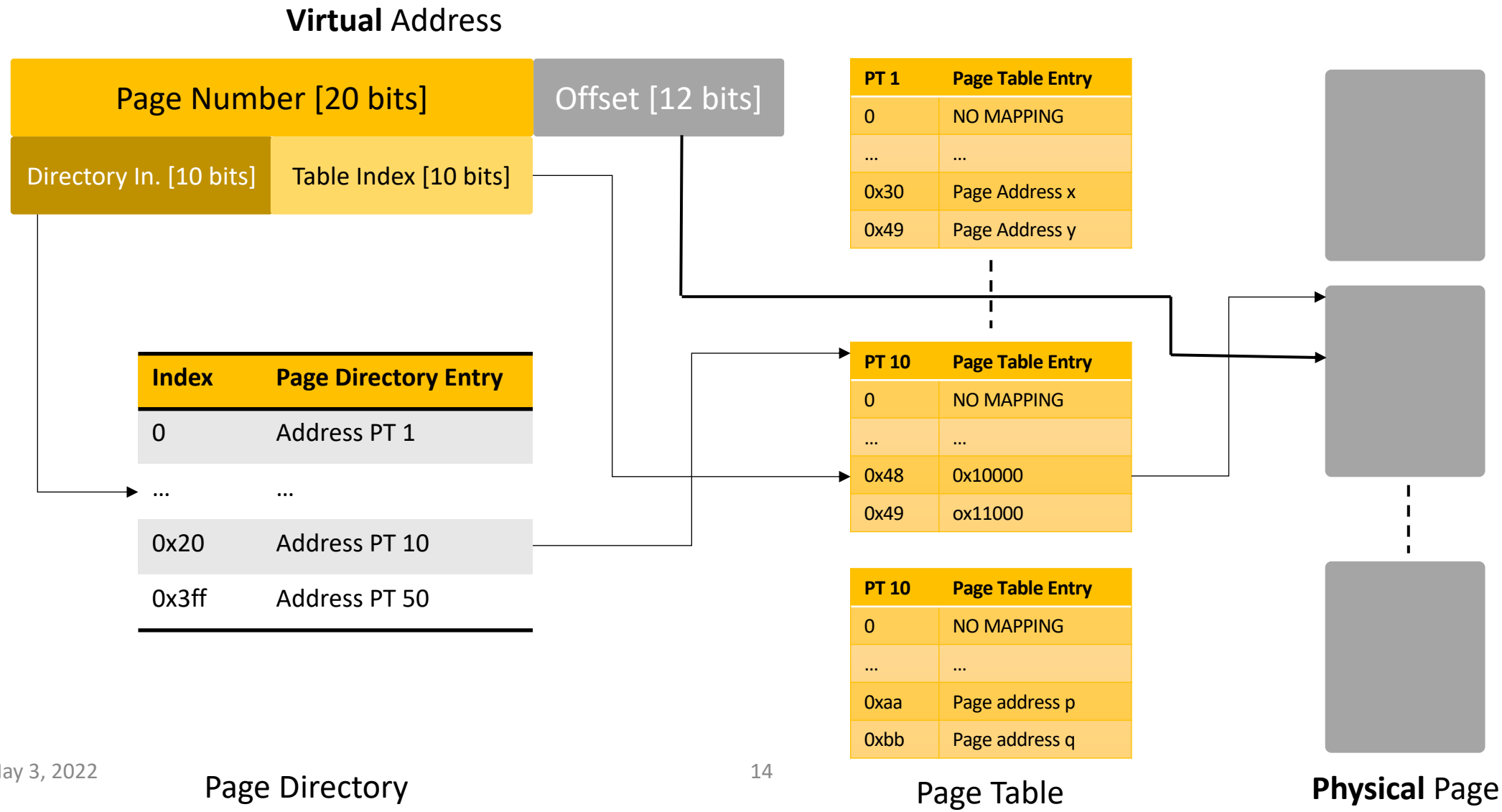
PT 10	Page Table Entry
0	NO MAPPING
…	…
0xaa	Page address p
0xbb	Page address q

Page Table

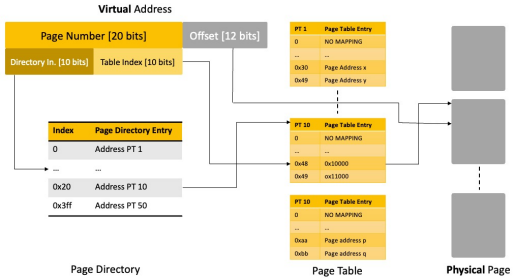
Memory Address Translation



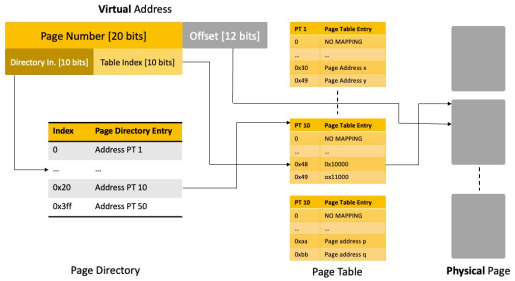
Memory Address Translation



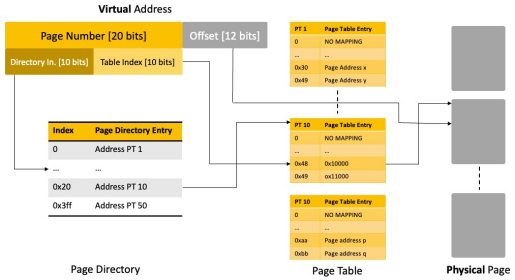
Memory Address Translation



Process 1 Page Table



Process 2 Page Table

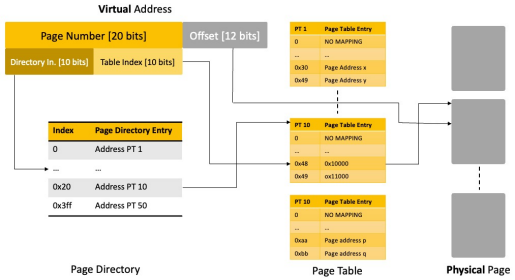


Process N Page Table

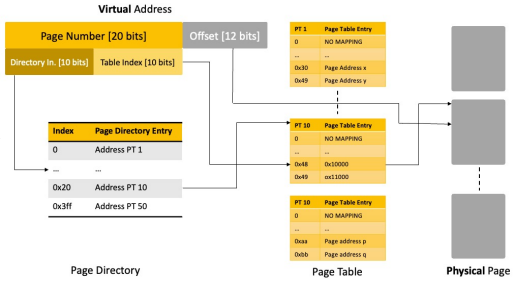
Memory Address Translation

CR3 REGISTER

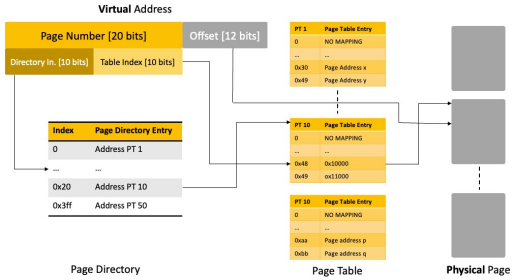
Point to the root of the PDE



Process 1 Page Table



Process 2 Page Table

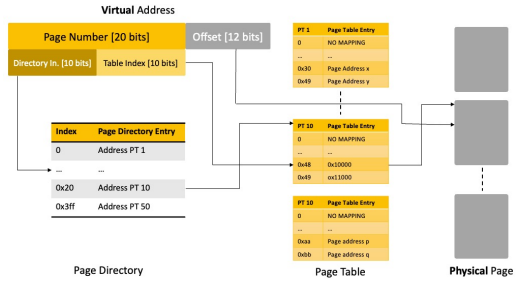


Process N Page Table

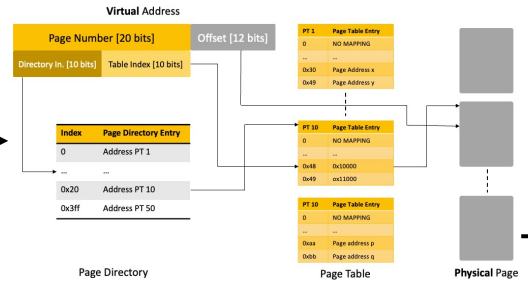
Memory Address Translation

CR3 REGISTER

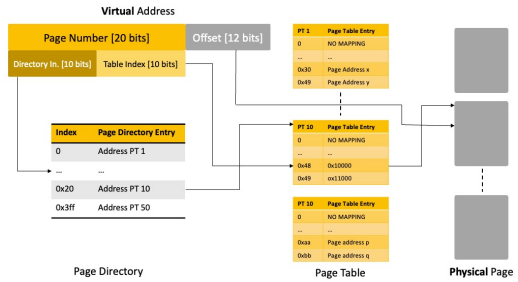
Point to the root of the PDE



Process 1 Page Table



Process 2 Page Table



Process N Page Table

VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0	0	0

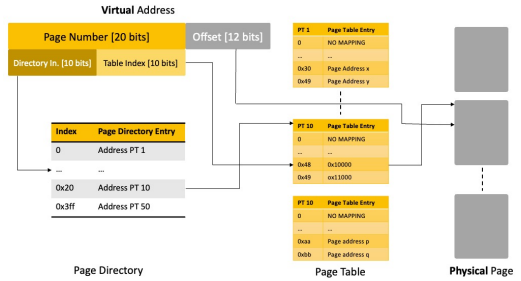
TLB

Caches most recent translation

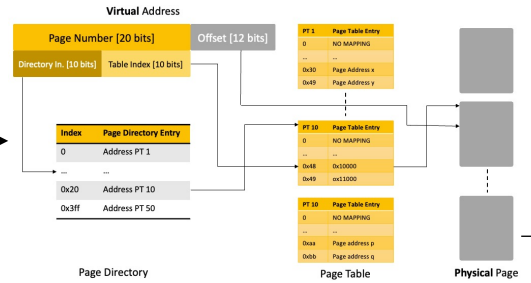
Memory Address Translation

CR3 REGISTER

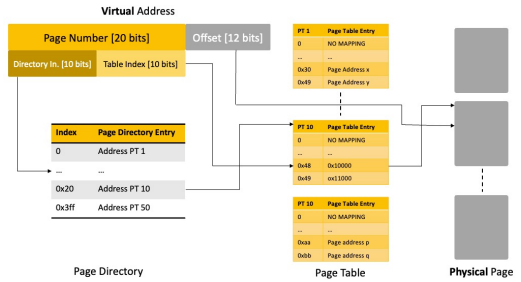
Point to the root of the PDE



Process 1 Page Table



Process 2 Page Table



Process N Page Table

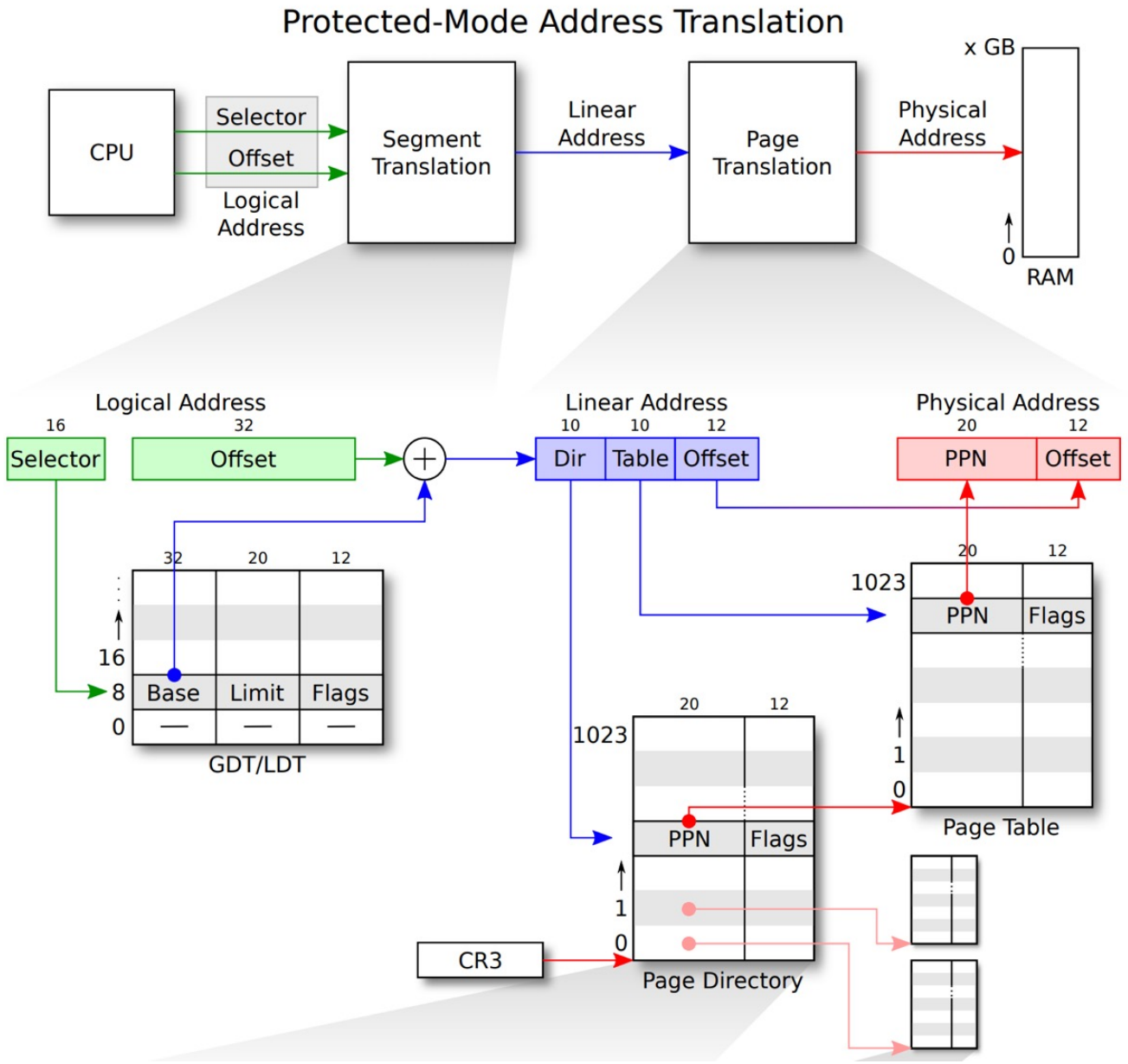
VPN	PPN	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1
0	0	0

TLB

Caches most recent translation

Can avoid expensive address translation process for cached addresses!

x86 Memory Access



Virtual Memory

Three Goals!

TRANSPARENCY

EFFICIENCY

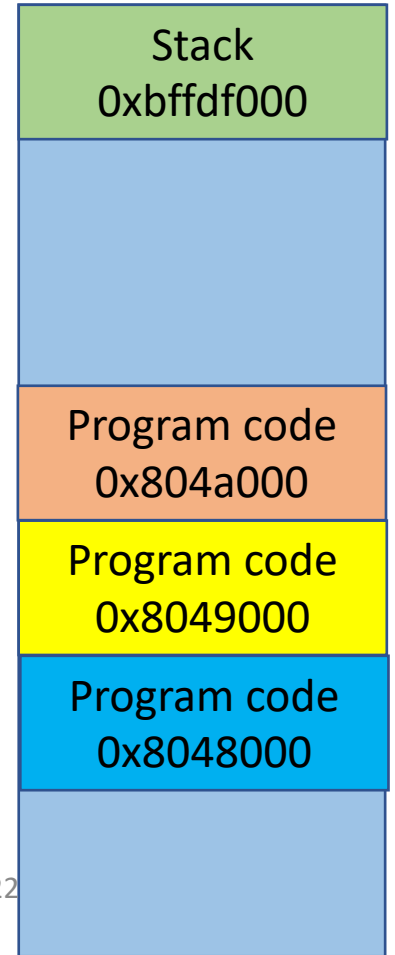
PROTECTION

Three Goals | Details

- **Transparency**: programs shouldn't need to know system's **internal state**
  - Program A is loaded at **0x8048000**. Can Program B be loaded at **0x8048000**?
- **Efficiency**: do not waste memory; avoid **memory fragmentation**
  - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of its allocation?
- **Protection**: **isolate** program's execution environment
  - Can we prevent an overflow from Program A from overwriting Program B's data?

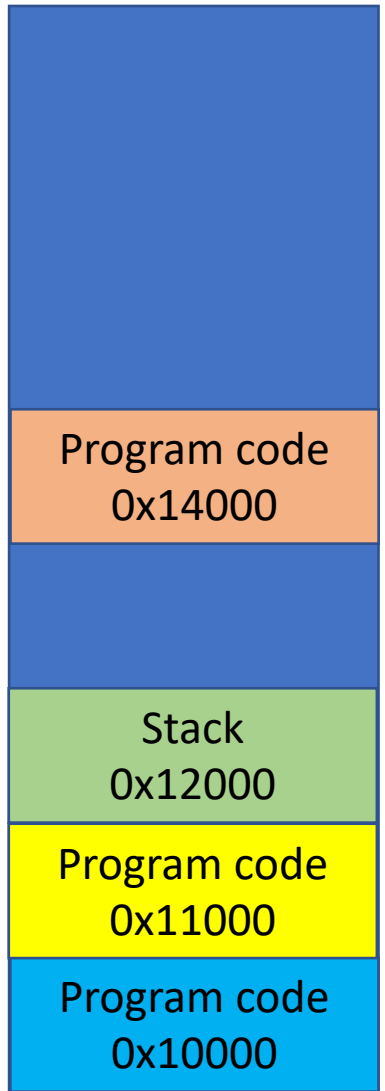
Virtual Memory | Paging

- Uses an **indirect table** that **maps virt-addr → phys-addr**



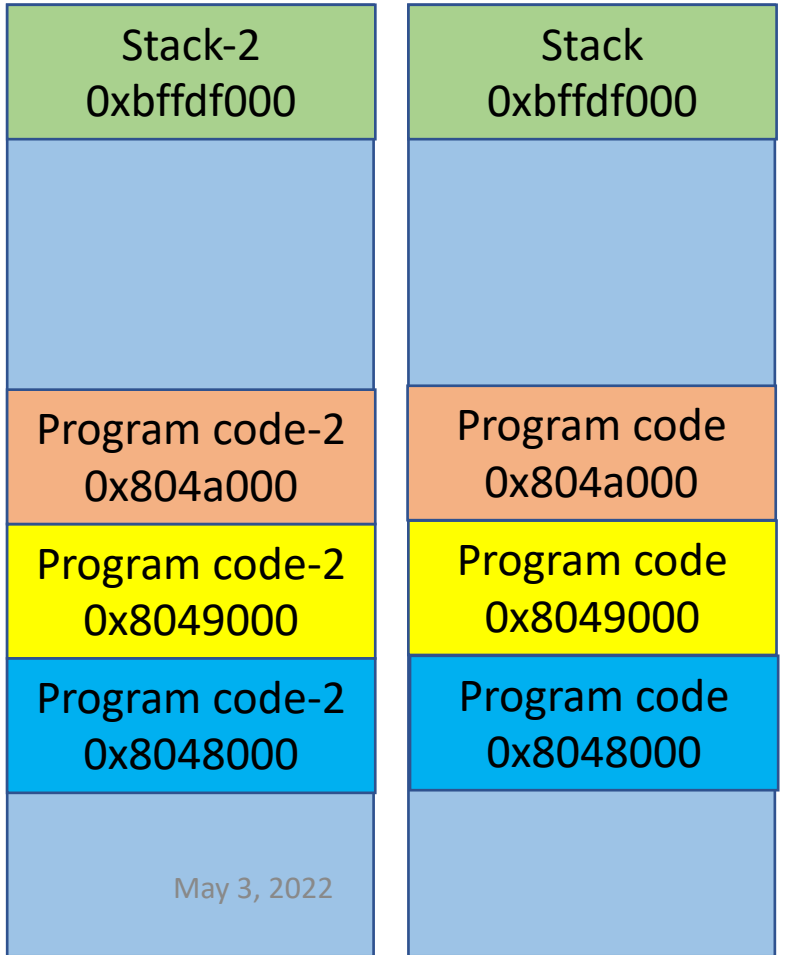
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

Physical Memory



Paging: Virtual Memory

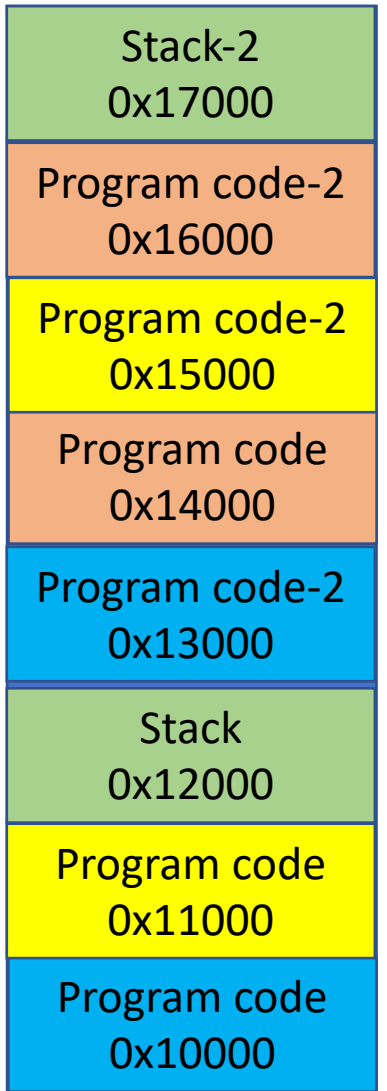
- Uses an **indirect table** that **maps virt-addr → phys-addr**



Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

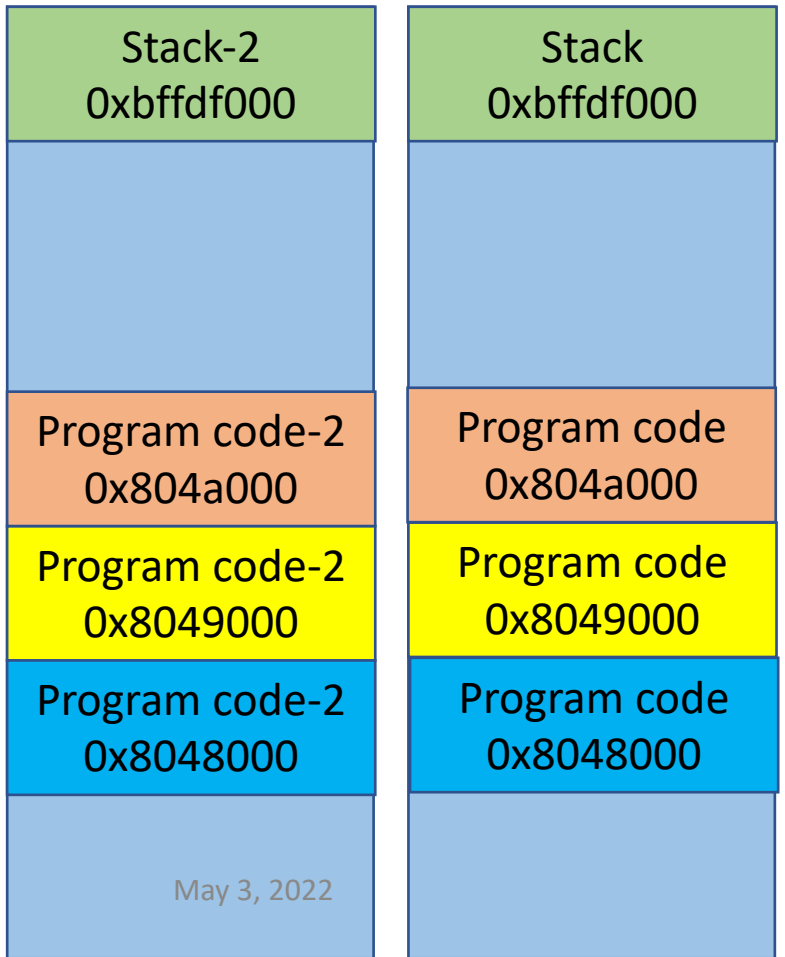
Physical Memory



Paging: Virtual Memory

Transparency: does not need to know system's internal state

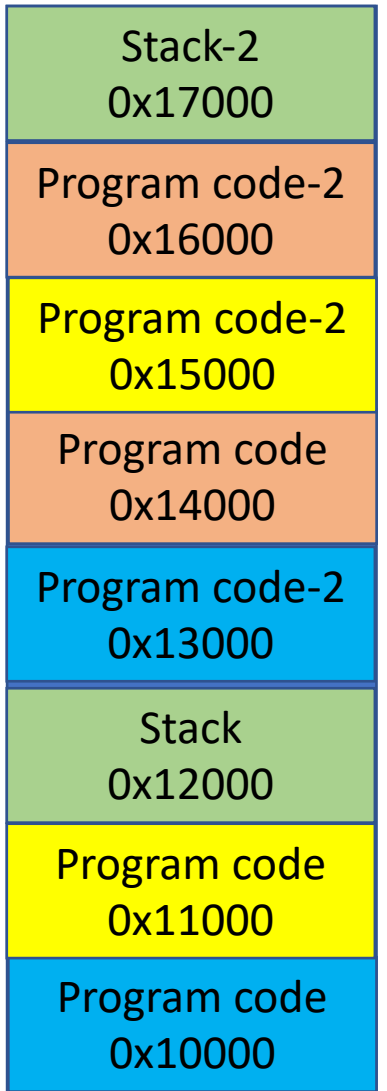
Program A & B loaded at **0x8048000?**



Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

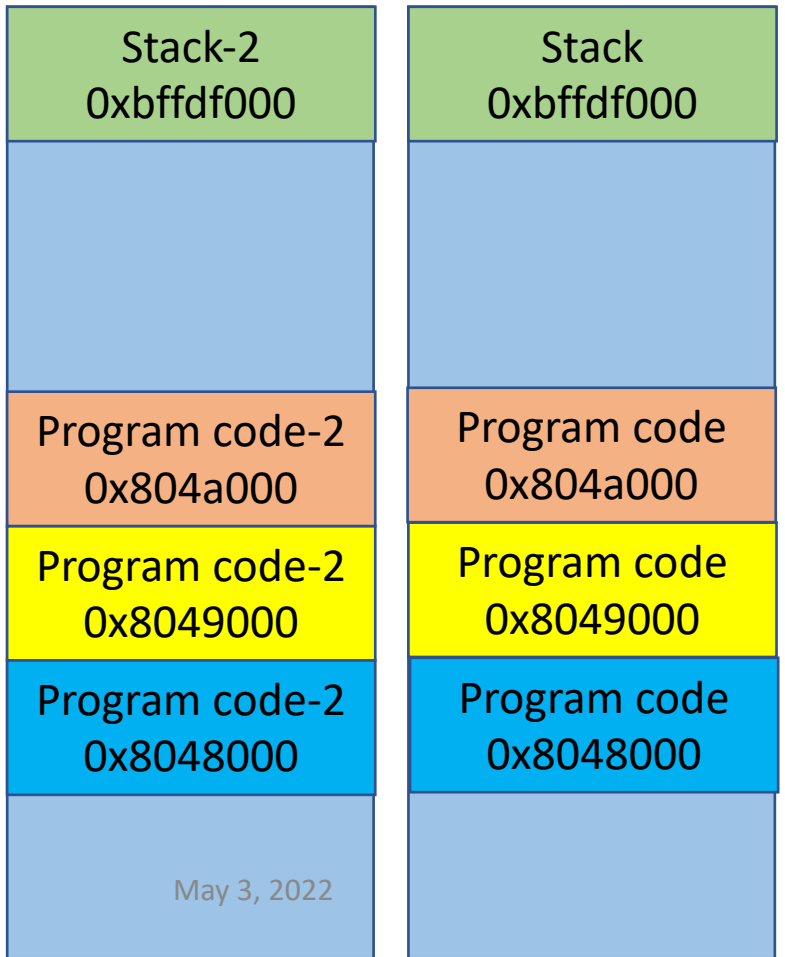
Physical Memory



Efficiency: do not waste memory

Can Program B (288KB) be loaded if only 288 KB of memory is free, regardless of its allocation?

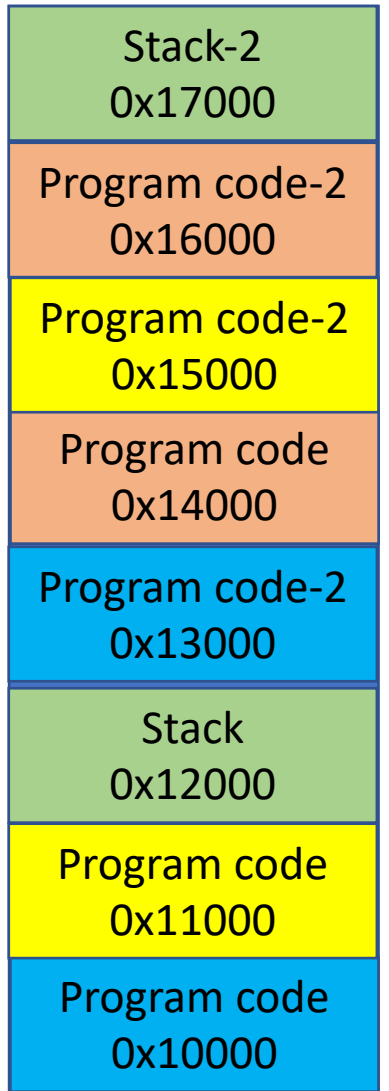
- Uses an **indirect table** that **maps virt-addr → phys-addr**



Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

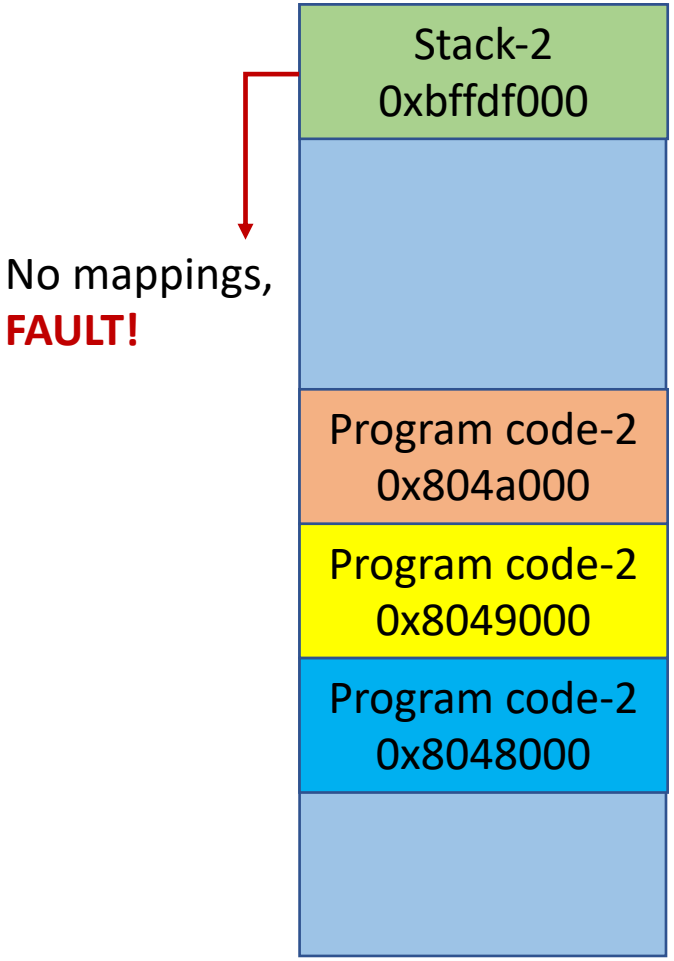
Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

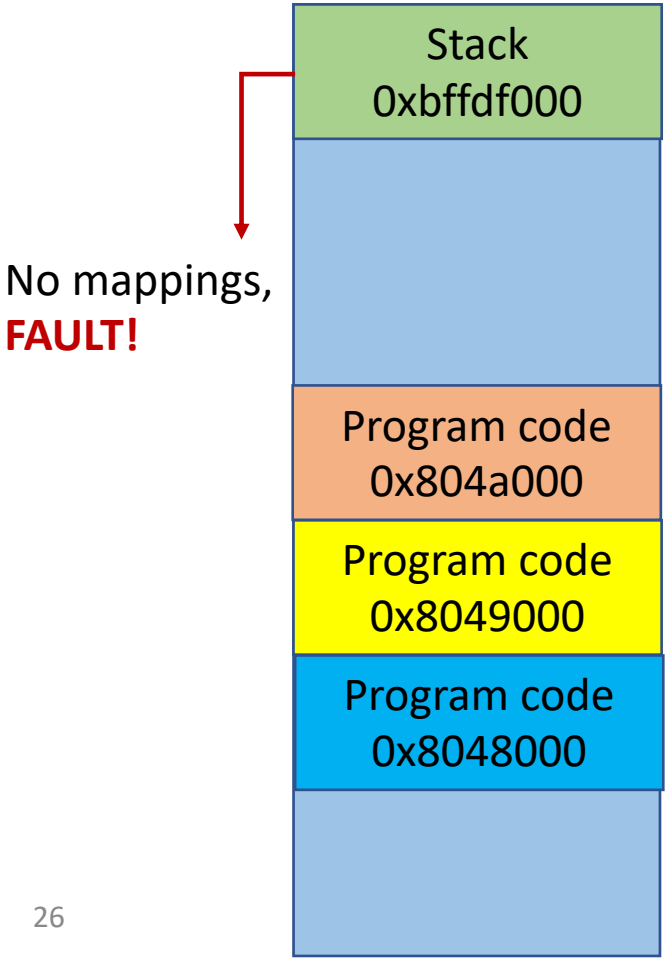
Physical Memory



Paging: Virtual Memory

Protection: **isolate** program's execution environment
Prevent overflow/overwriting between multiple programs?





User Space

- Boot sequence

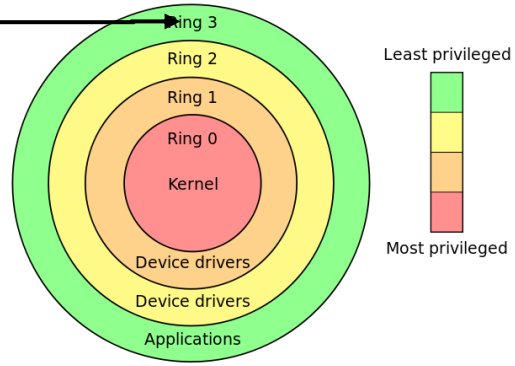
Bootloader

Kernel

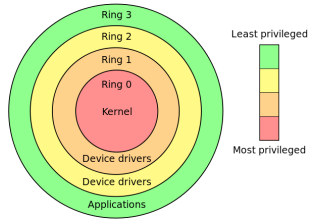
- Memory Management

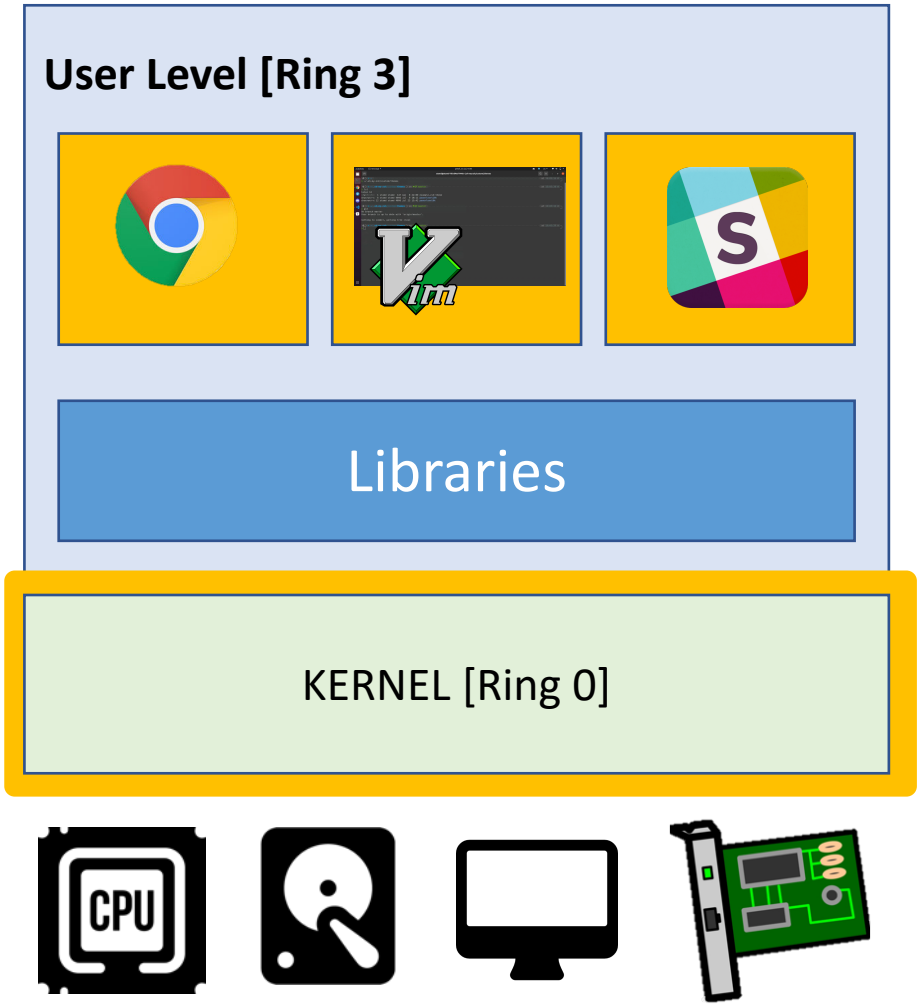
- Chrome
- Vim
- Slack…

User Space



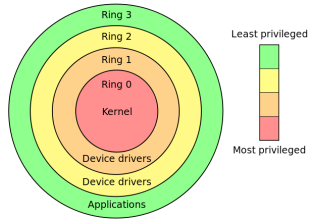
Kernel [Ring 0]

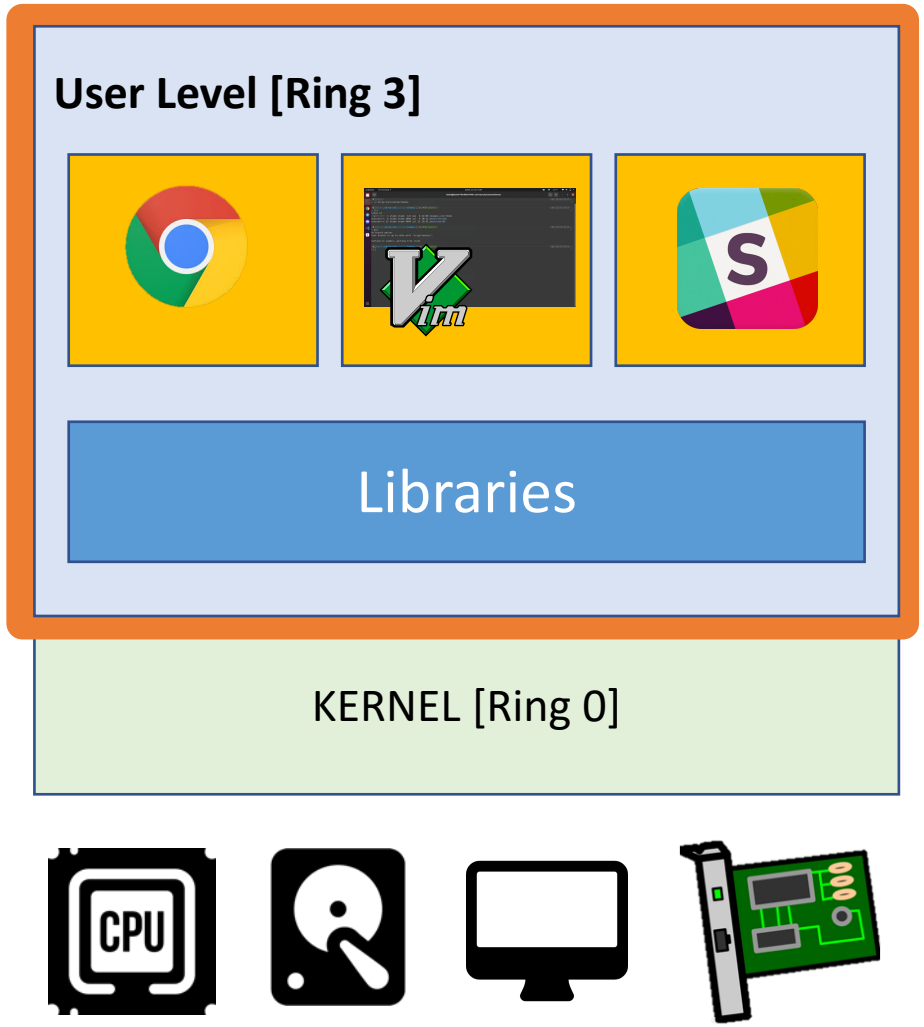




- Executes with **highest privilege level** (Ring 0)
- Configures system (devices, memory, etc.)
- Manages hardware resources
  - Disk, memory, network, video, keyboard, etc.
- Manages **other jobs**
  - Processes and threads
- Serves as **trusted computing base** (TCB)
  - Sets privilege
  - Restrict other jobs from doing something bad

User [**Ring 3**]

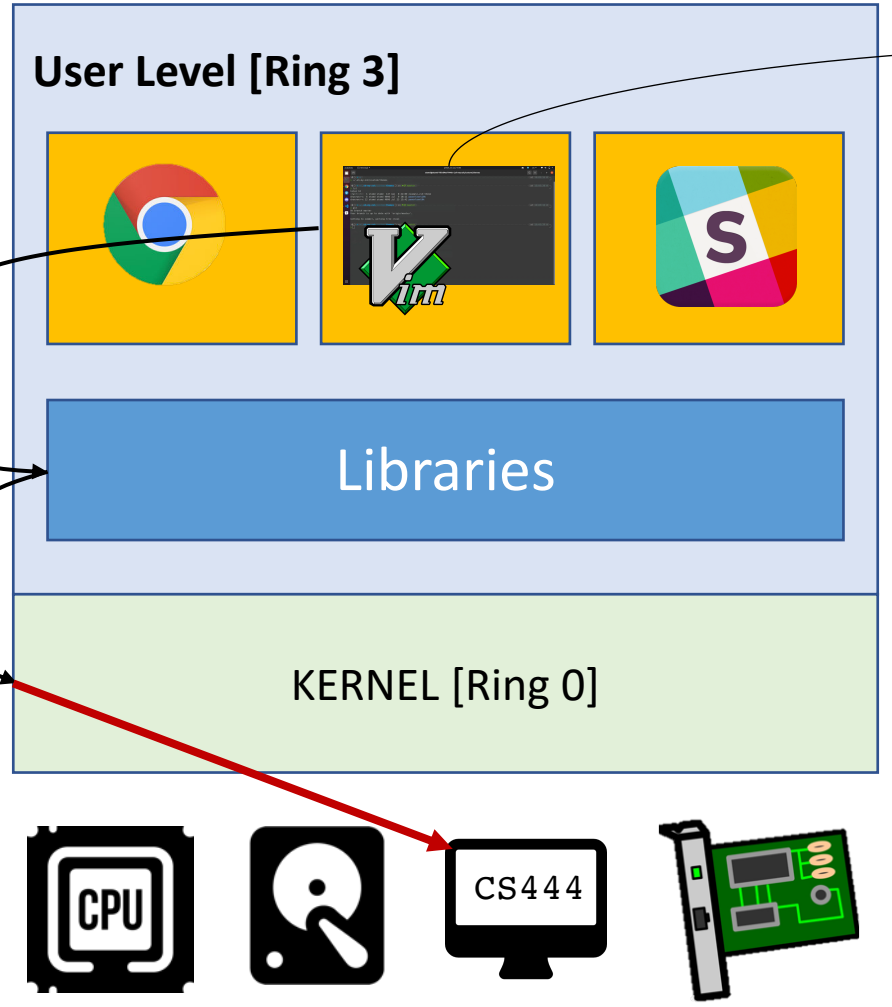




- Runs with a **restricted** privilege [**Ring 3**]
  - The privilege level for running an application
- Most regular applications run at this level
- **Cannot** access kernel memory
  - Can only access pages set with **PTE_U**
- **Cannot talk directly to hardware devices**
  - Kernel must mediate the access

Overview of User/Kernel Execution

```
int main() {
    printf("CS444");
}
```



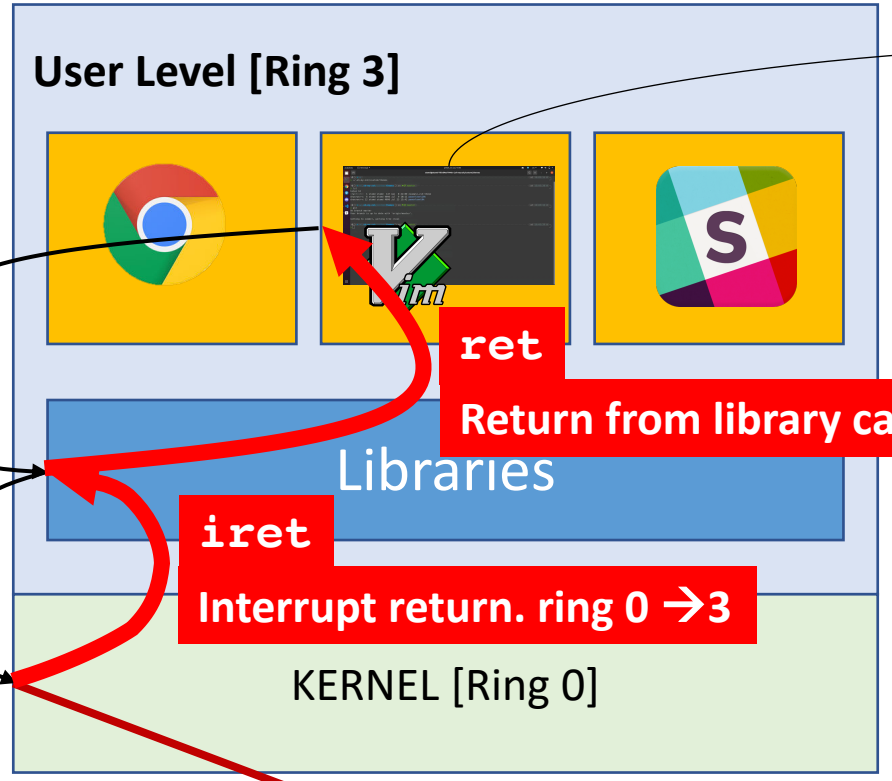
`printf``("CS444")`
library call in **ring 3**

`sys_write``(1,"CS444",5)`
system call from **ring 3**
Interrupt! Switch from ring 3 →0

`do_sys_write``(1,"CS444",5)`
kernel function in **ring 0**

Overview of User/Kernel Execution [contd.]

```
int main() {
    printf("CS444");
}
```



`printf``("CS444")`
library call in **ring 3**

`sys_write``(1,"CS444",5)`
system call from **ring 3**
Interrupt! Switch from ring 3 →0

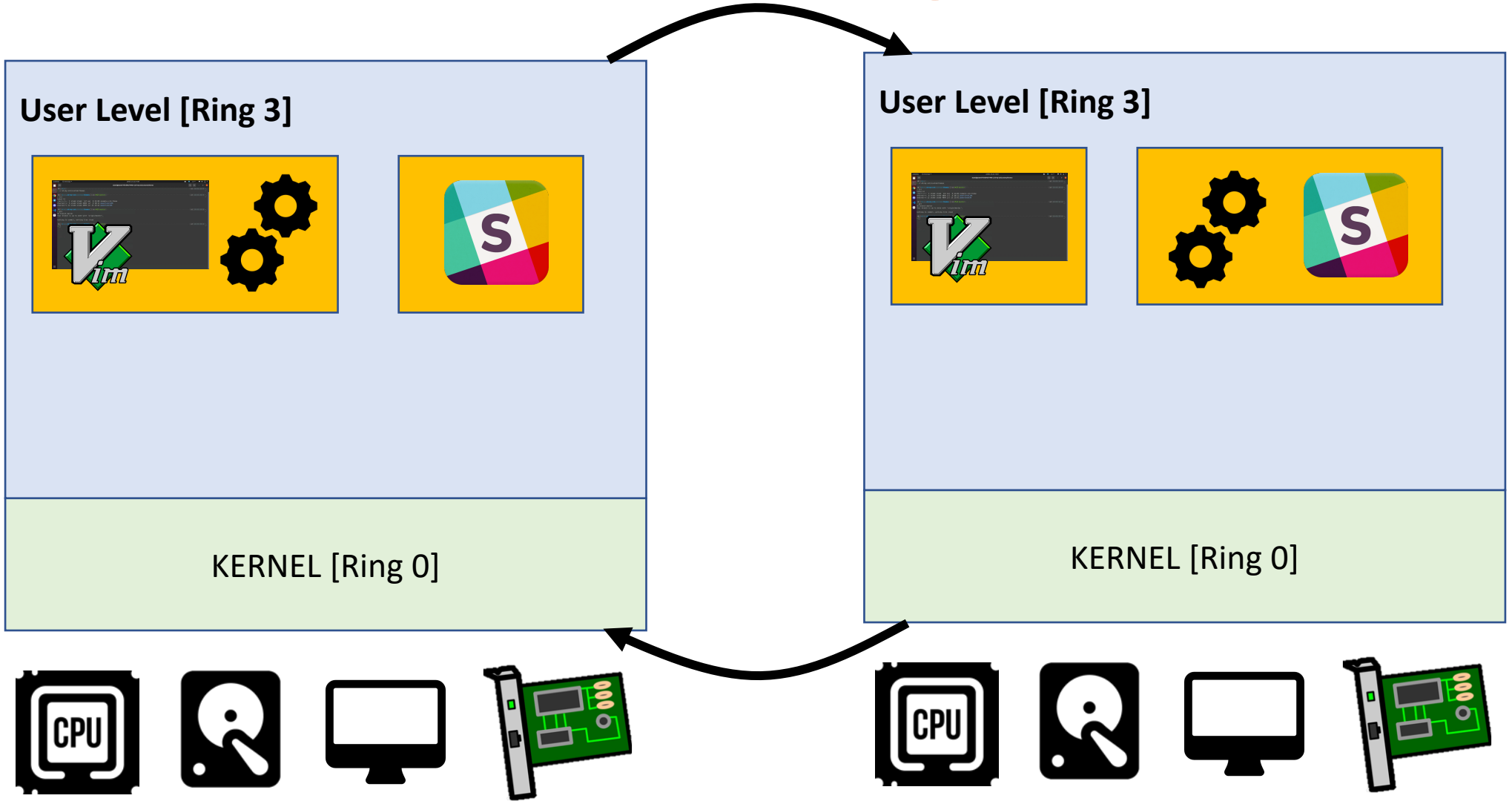
`do_sys_write``(1,"CS444",5)`
kernel function in **ring 0**

`ret`
Return from library call. ring 3 [no switch]

`iret`
Interrupt return. ring 0 →3

```c
int main() {
    while(1);
}
```

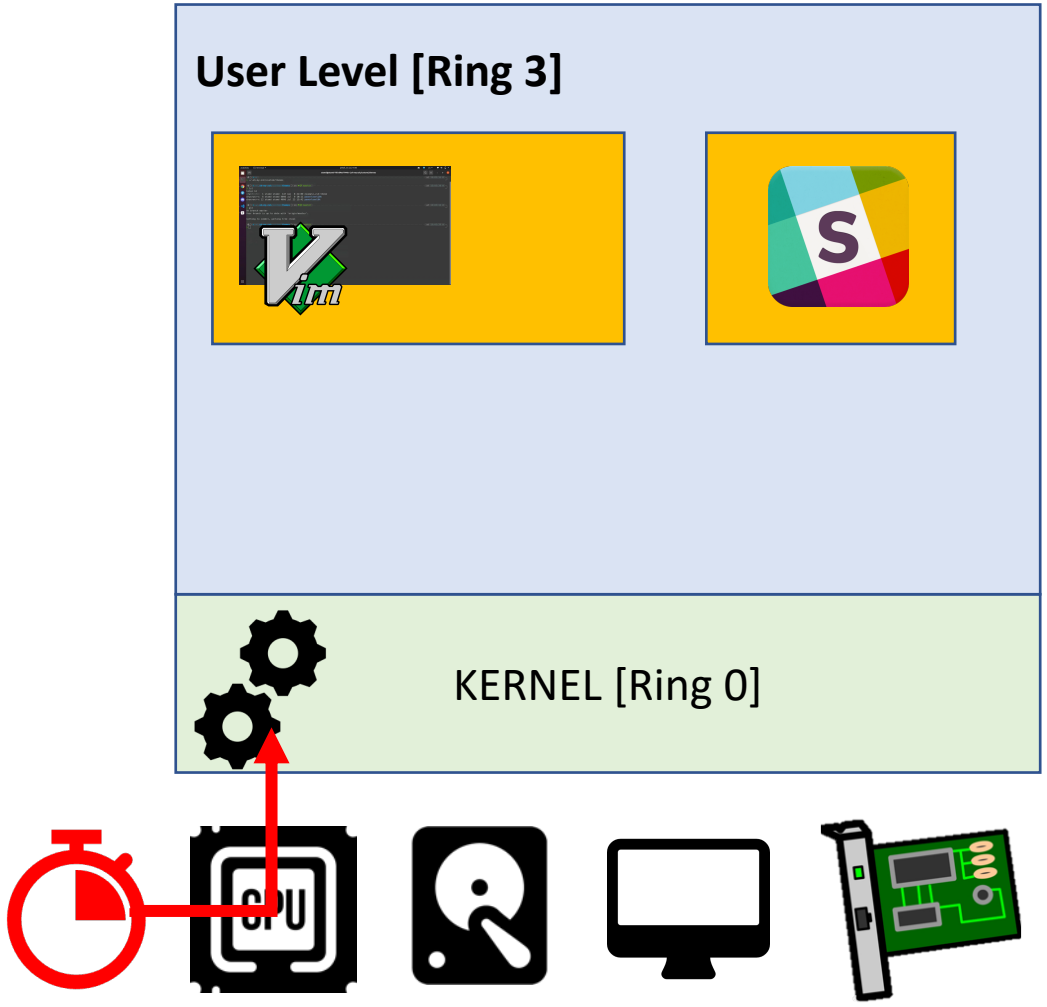
Preemptive Multitasking



Preemptive Multitasking | Timers

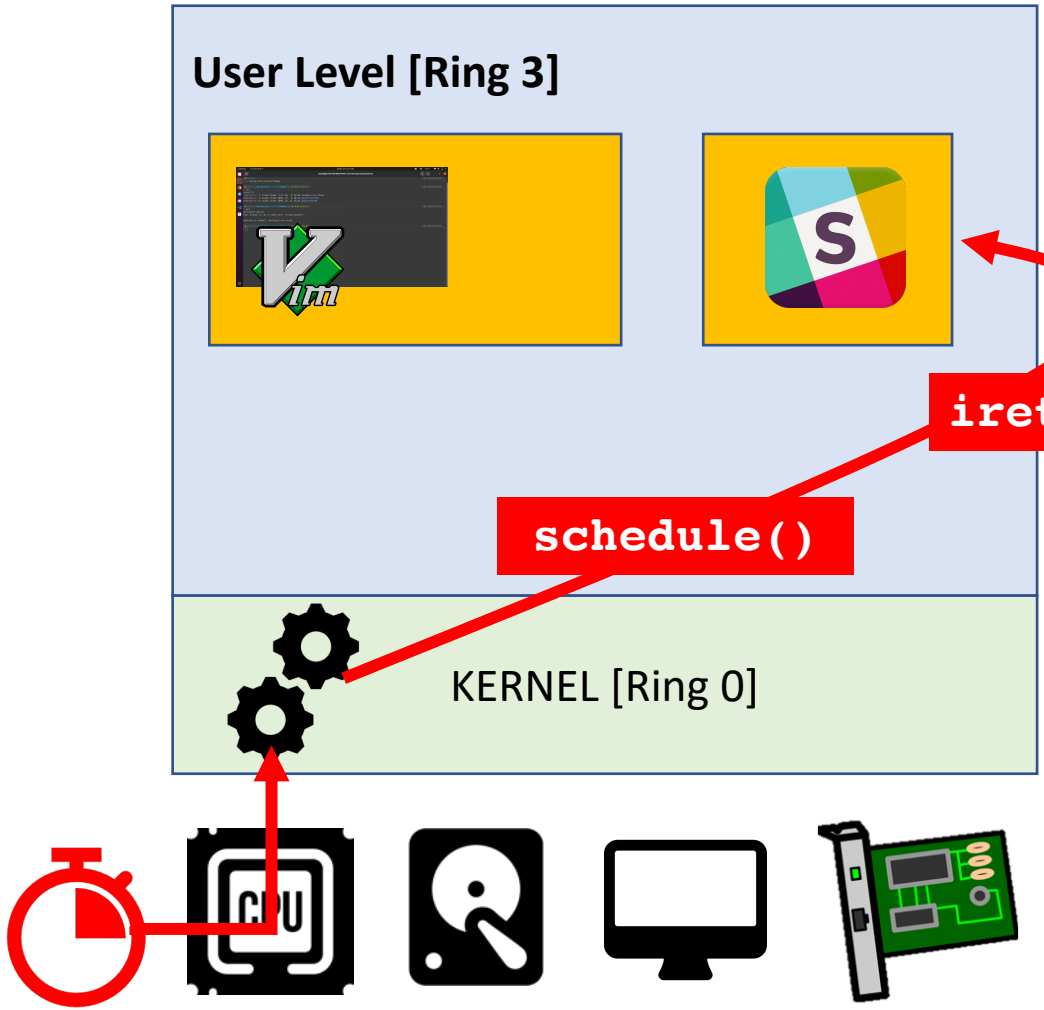
- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**

Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

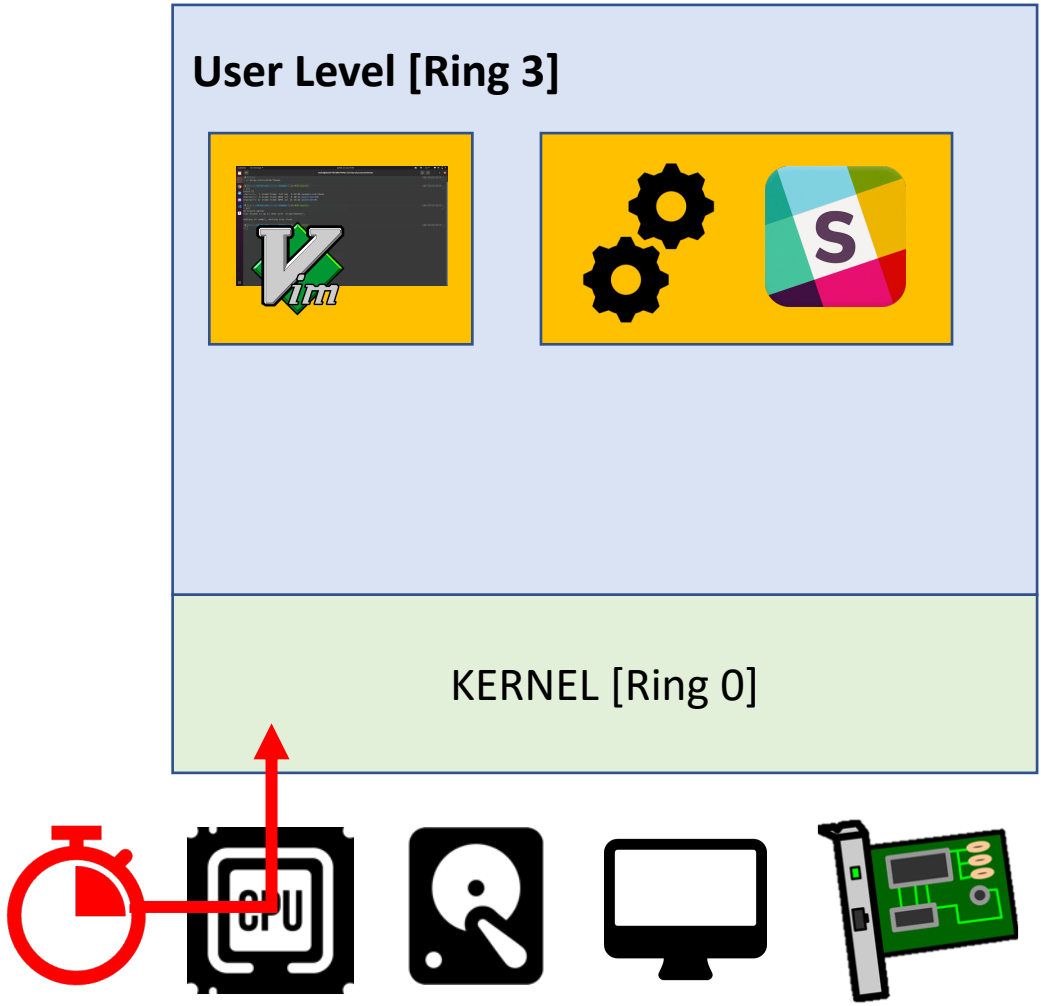
Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

Preemptive Multitasking | Timers

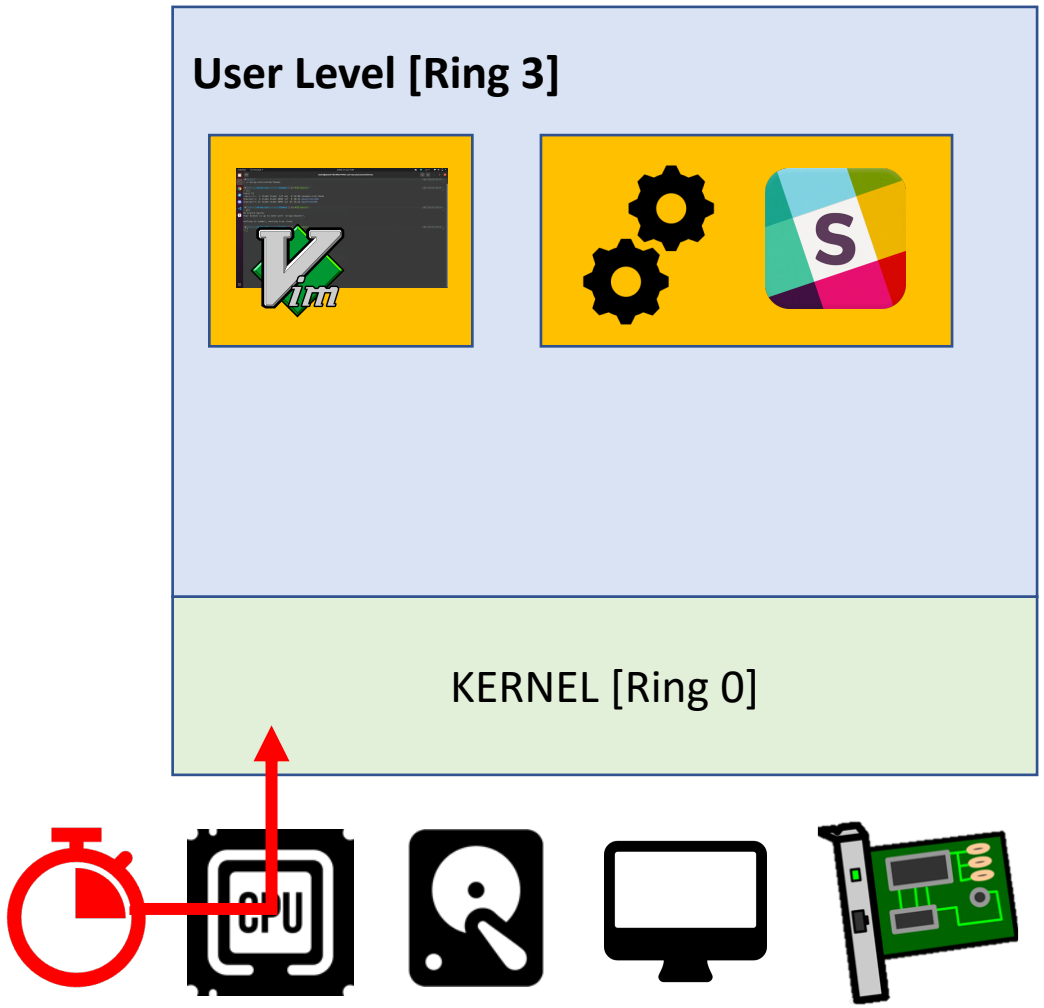


- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

time quantum

Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

time quantum

- **Timer guarantees execution in kernel**

Traps

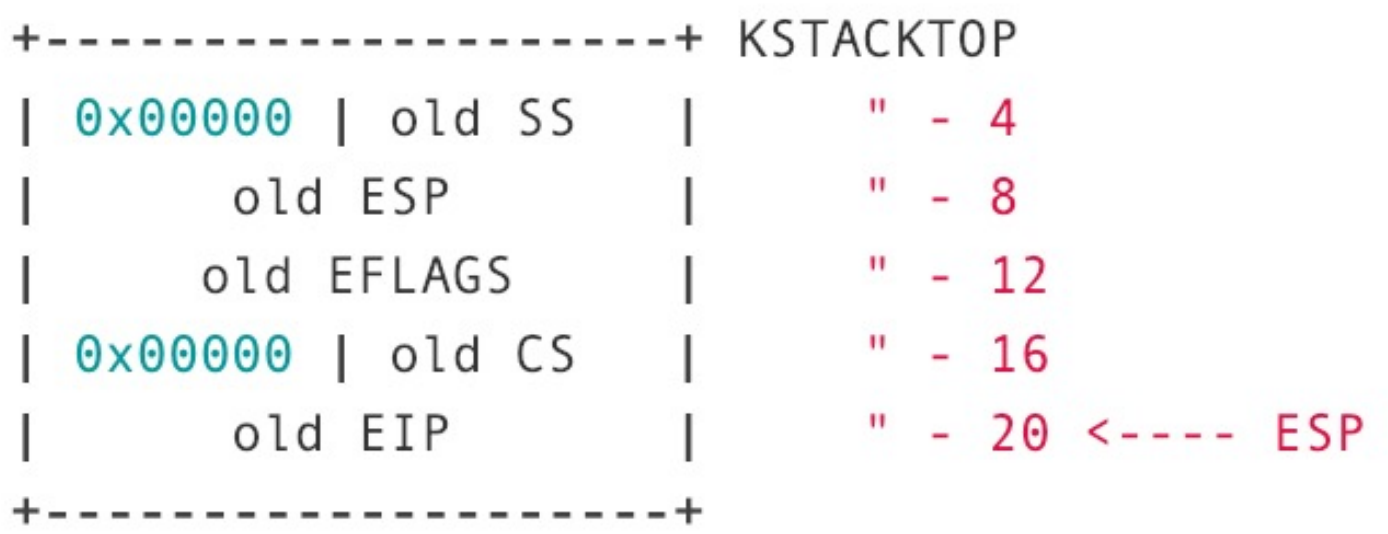
- Any event that forces CPU to stop and **execute kernel code**
- **trap handler**

Types of Traps

Interrupts	Faults	Exceptions
• **Hardware interrupt** [clock timer, network packet, etc.] • **Software interrupt** [System calls]	• An error that OS **can recover from** and continue execution [e.g., page fault]	• An error that **OS cannot recover from** • must stop the current execution [e.g., divide by zero]

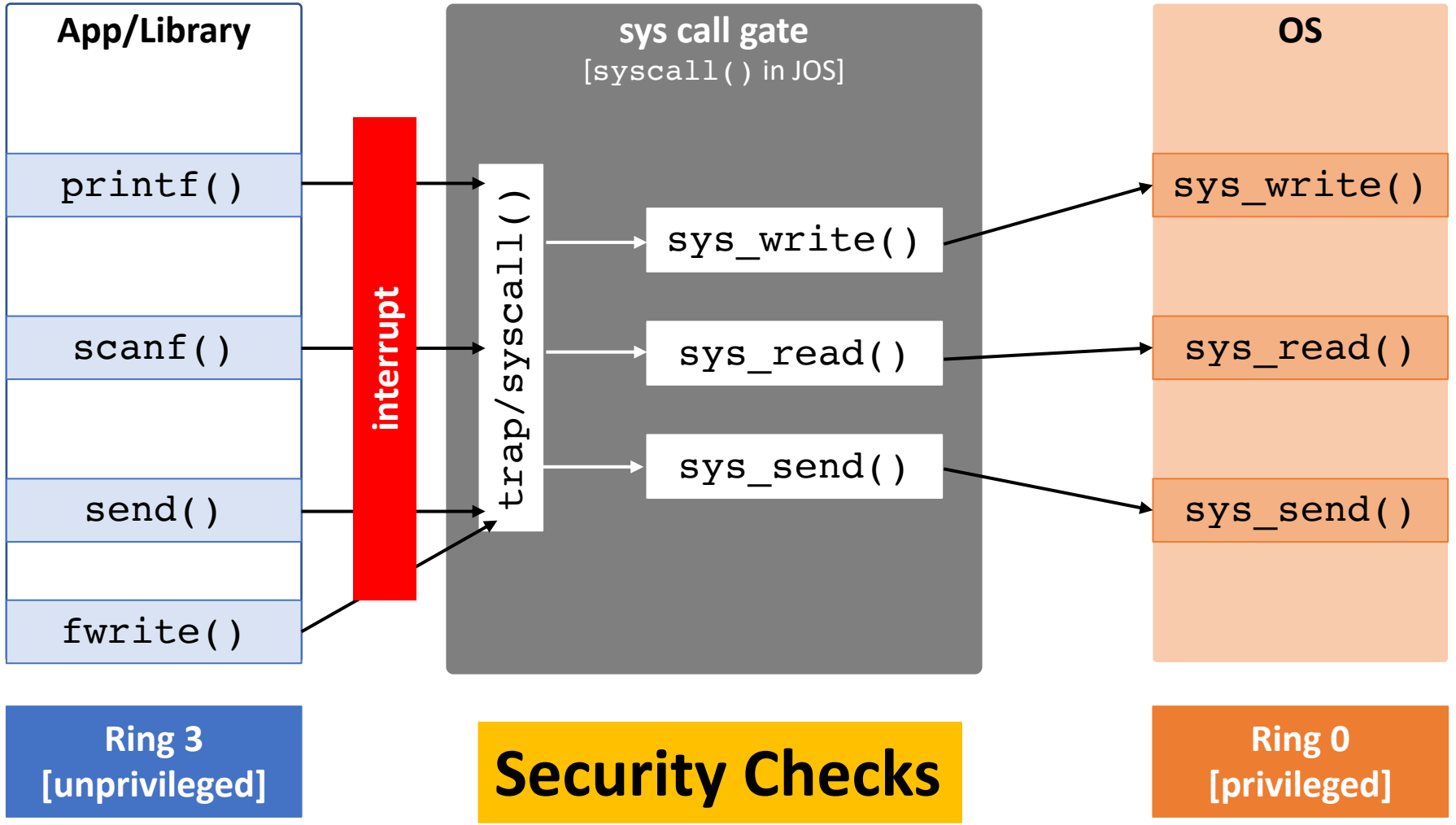
Many others, please refer to the Intel Manual Chapter 6 (https://os.unexploitable.systems/r/ia32/IA32-3A.pdf)

Trapframe!


+---------------------+ KSTACKTOP
| 0x00000 | old SS | " - 4
| old ESP | " - 8
| old EFLAGS | " - 12
| 0x00000 | old CS | " - 16
| old EIP | " - 20 <---- ESP
+---------------------+

Secure System Call Design: Call Gate via Interrupt Handling

- Call gate: a secure method to control access to Ring 0!



A Program's Memory Layout [ELF]

`.text`

- Code area. Read-only and executable

`.rodata`

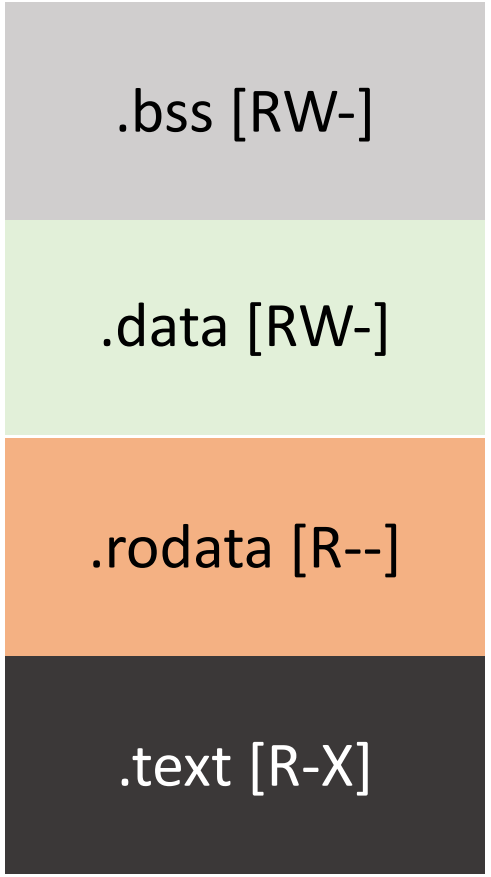
- Data area, Read-only and not executable

`.data`

- Data area, Read/Writable (not executable)
- Initialized by some values

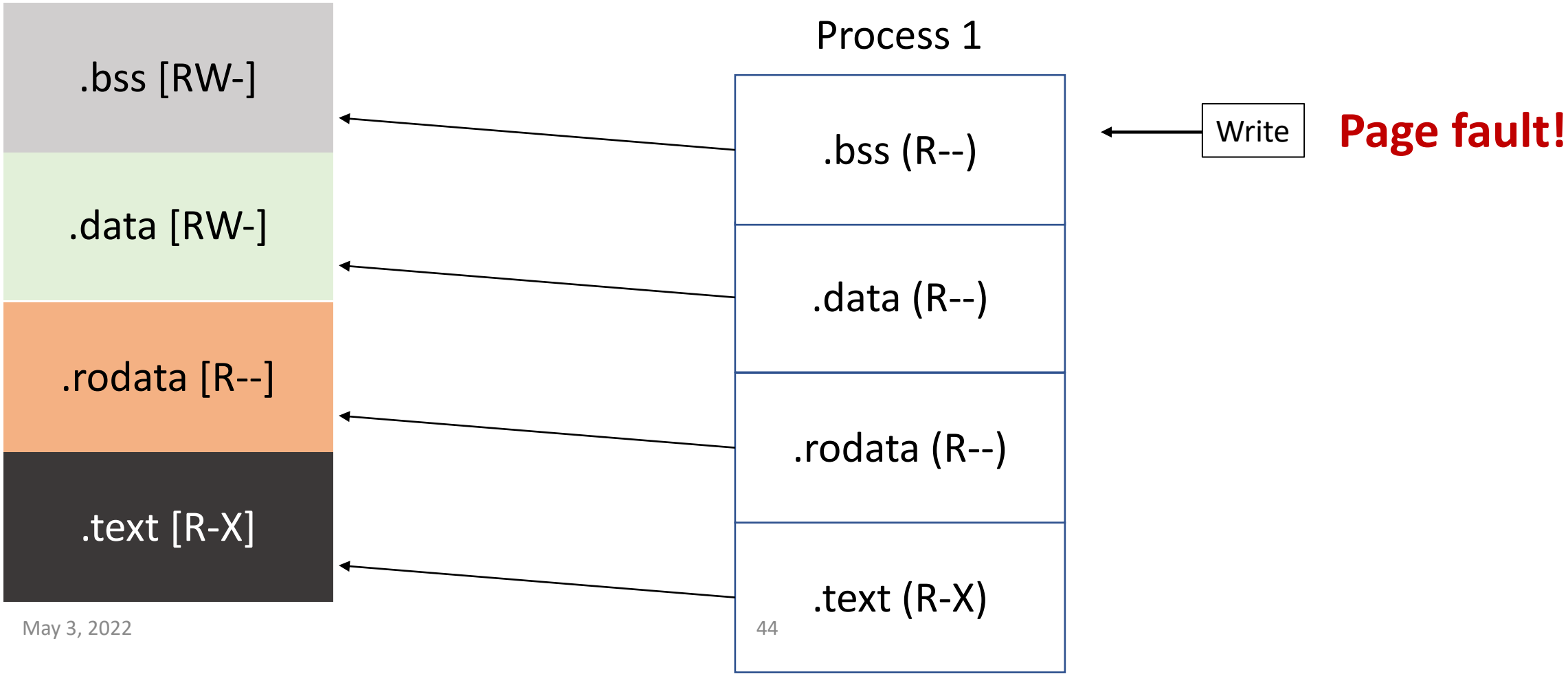
`.bss`

- uninitialized data
- Data area, Read/Writable (not executable)
- Initialized as 0



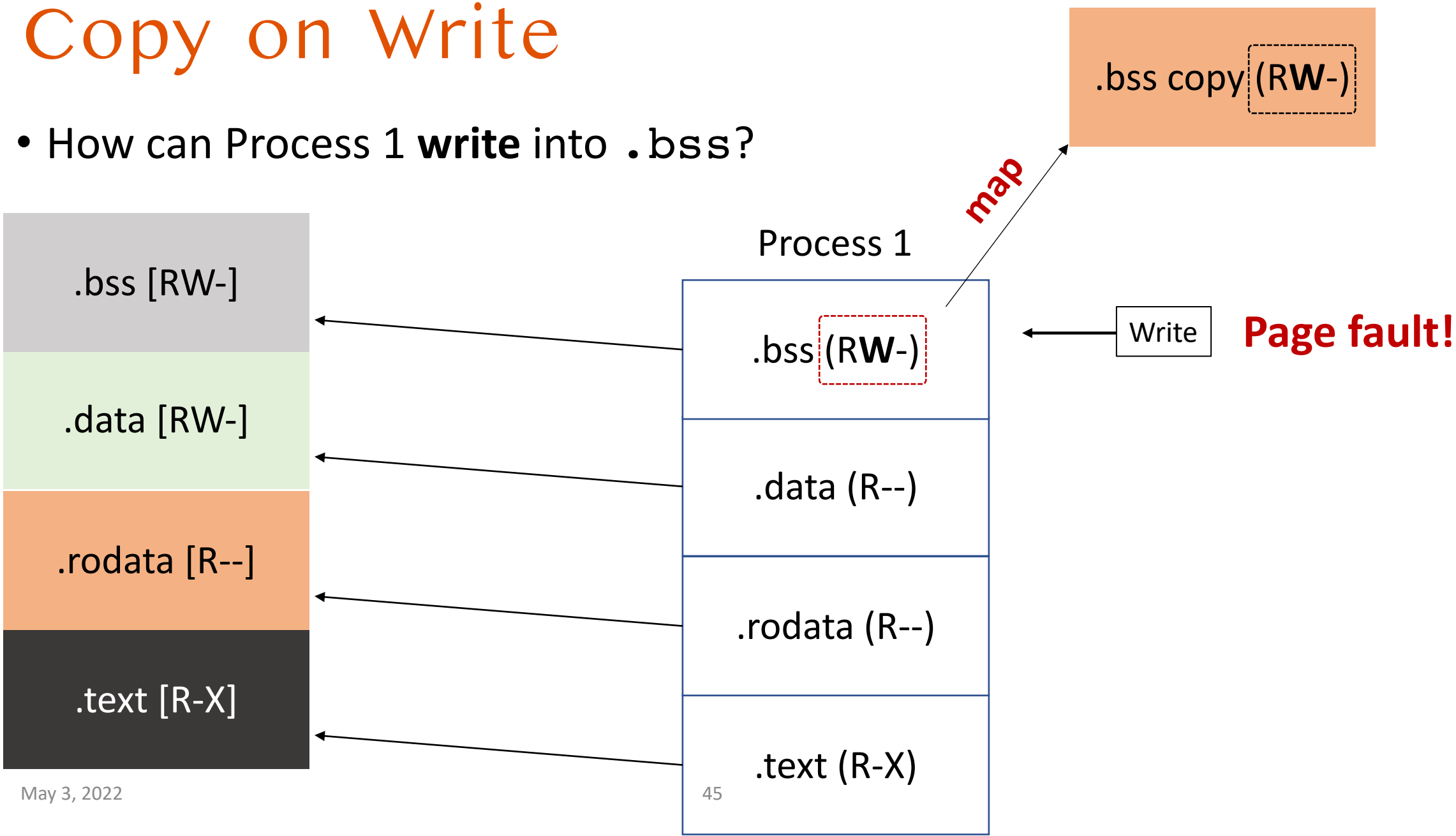
Copy on Write

- How can Process 1 **write** into `.bss`?



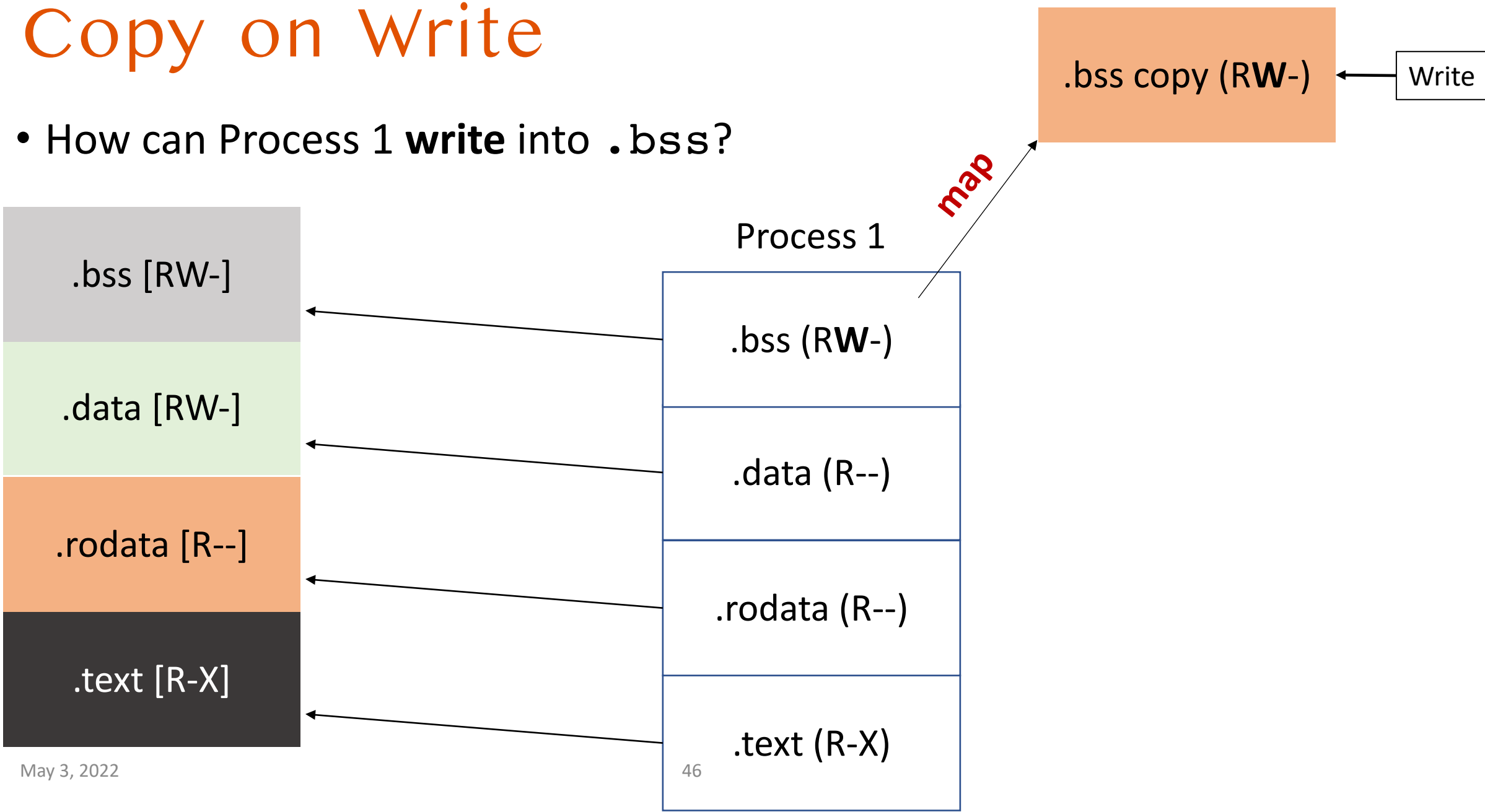
Copy on Write

- How can Process 1 **write** into `.bss`?



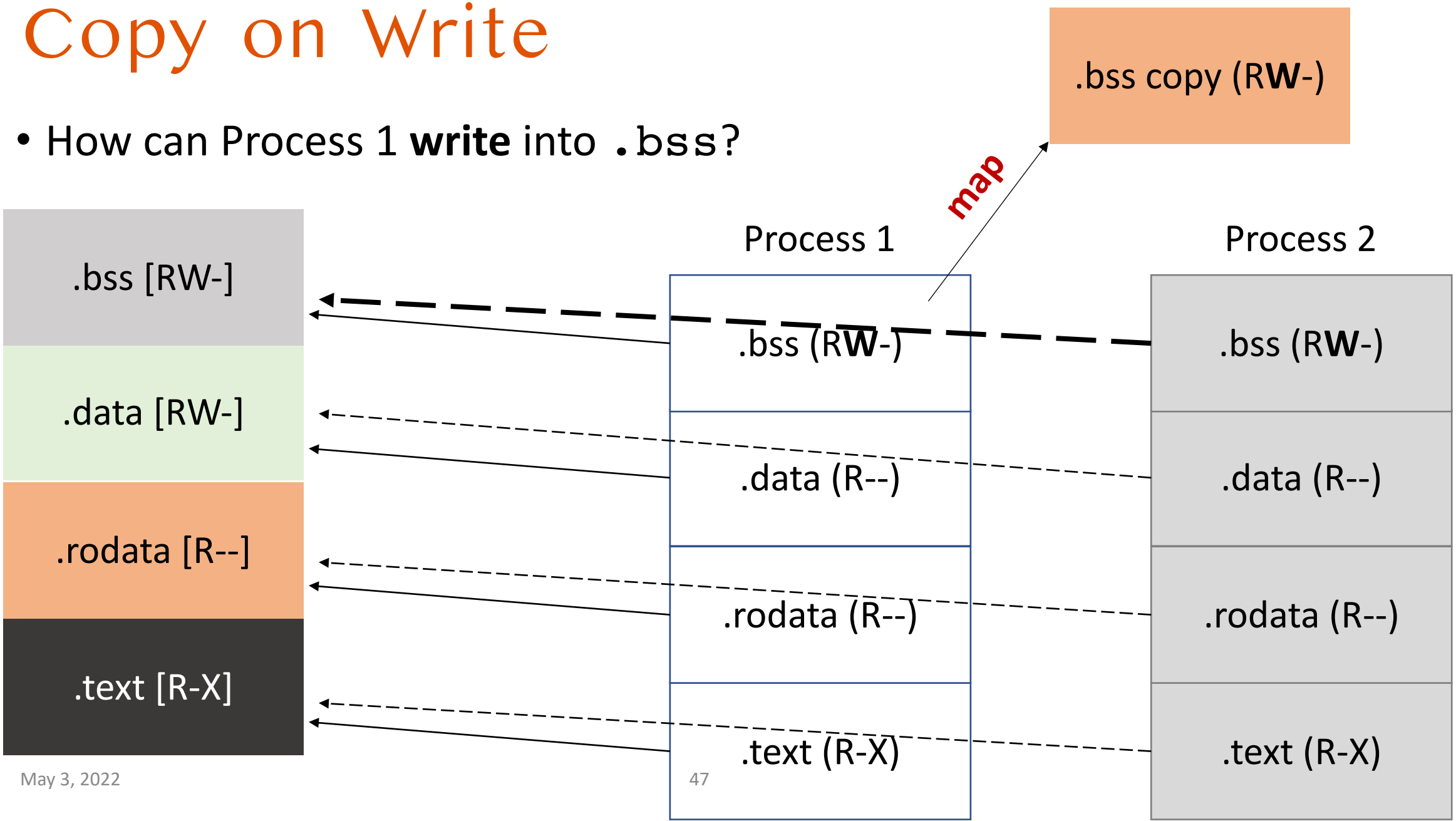
Copy on Write

- How can Process 1 **write** into `.bss`?



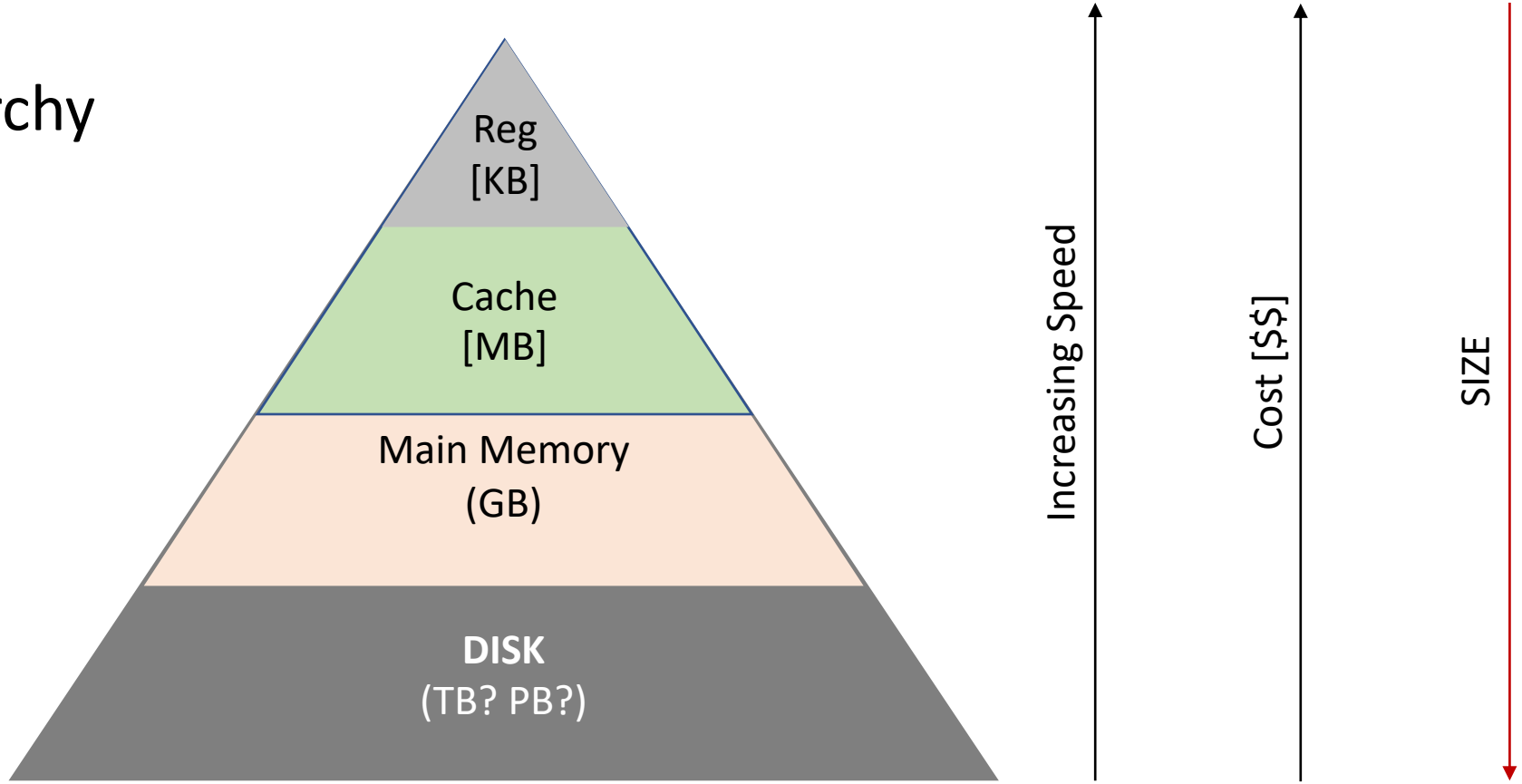
Copy on Write

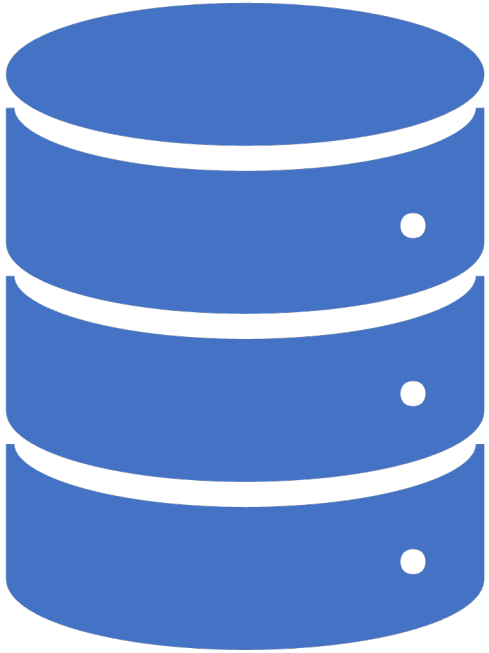
- How can Process 1 **write** into `.bss`?



Memory Swapping

- Memory Hierarchy





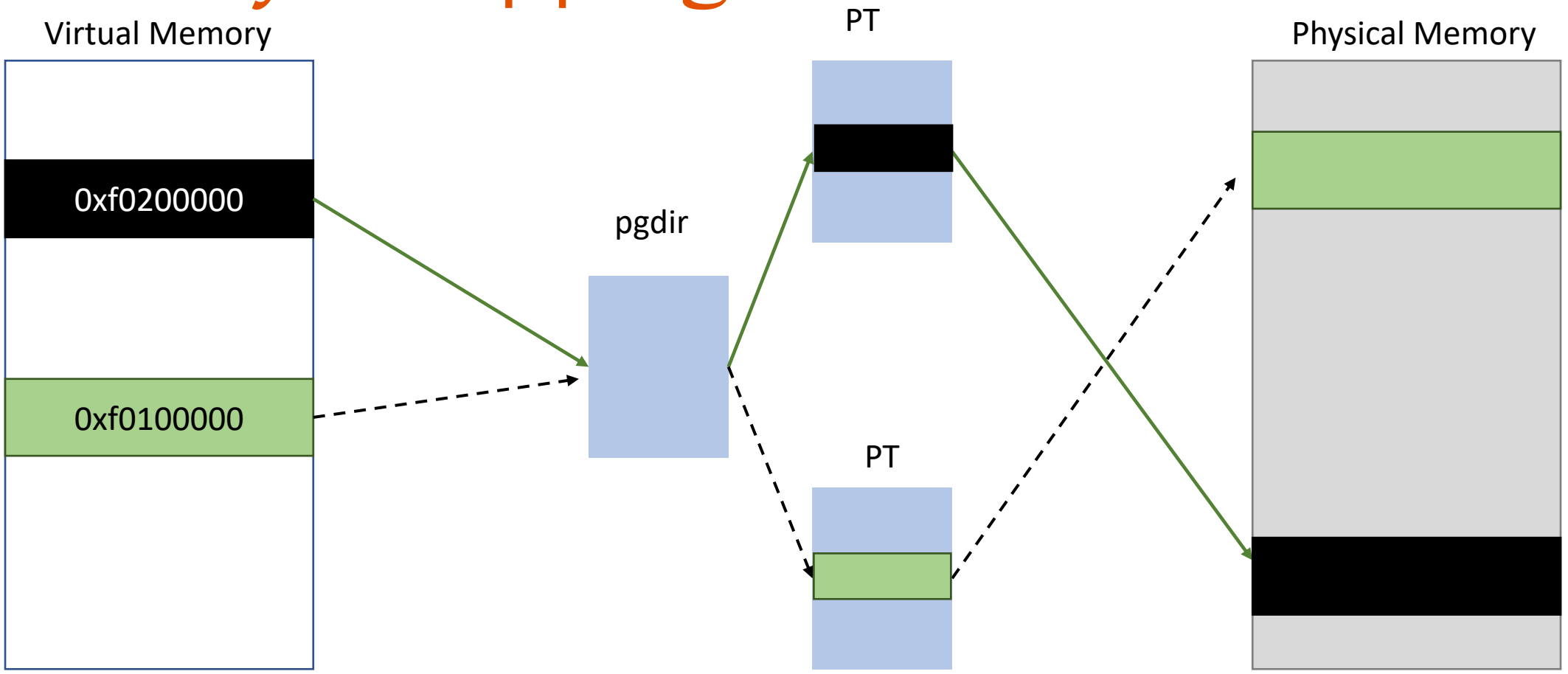
Challenge

- Suppose you have 8GB of main memory

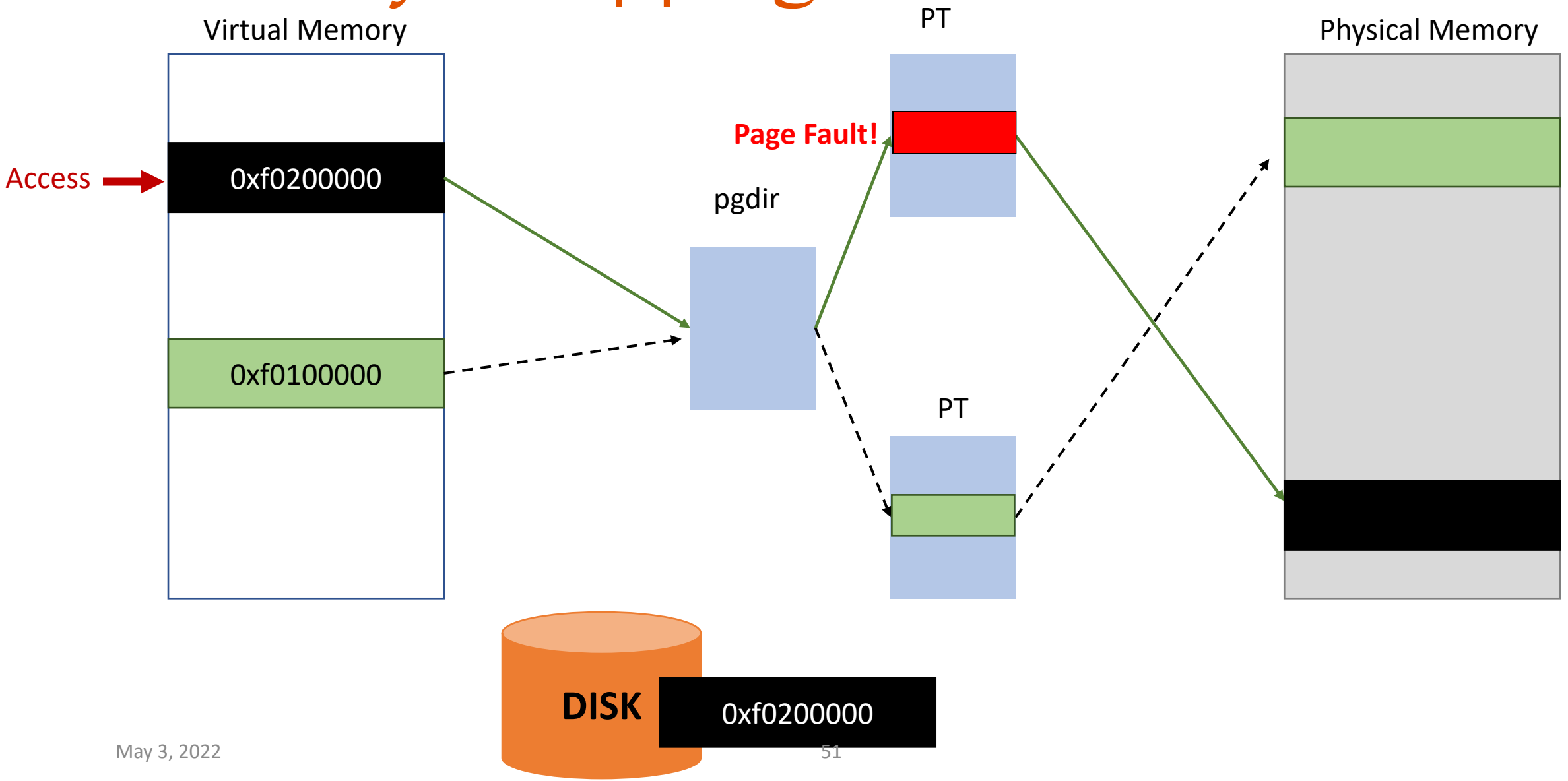
- Can you run a program that is 16GB in size?
  - Yes, you can manually **load it one part at a time**
  - we **do not use all of data at the same time**

- OS do this **seamlessly** [**transparently**] for application?

Memory Swapping



Memory Swapping

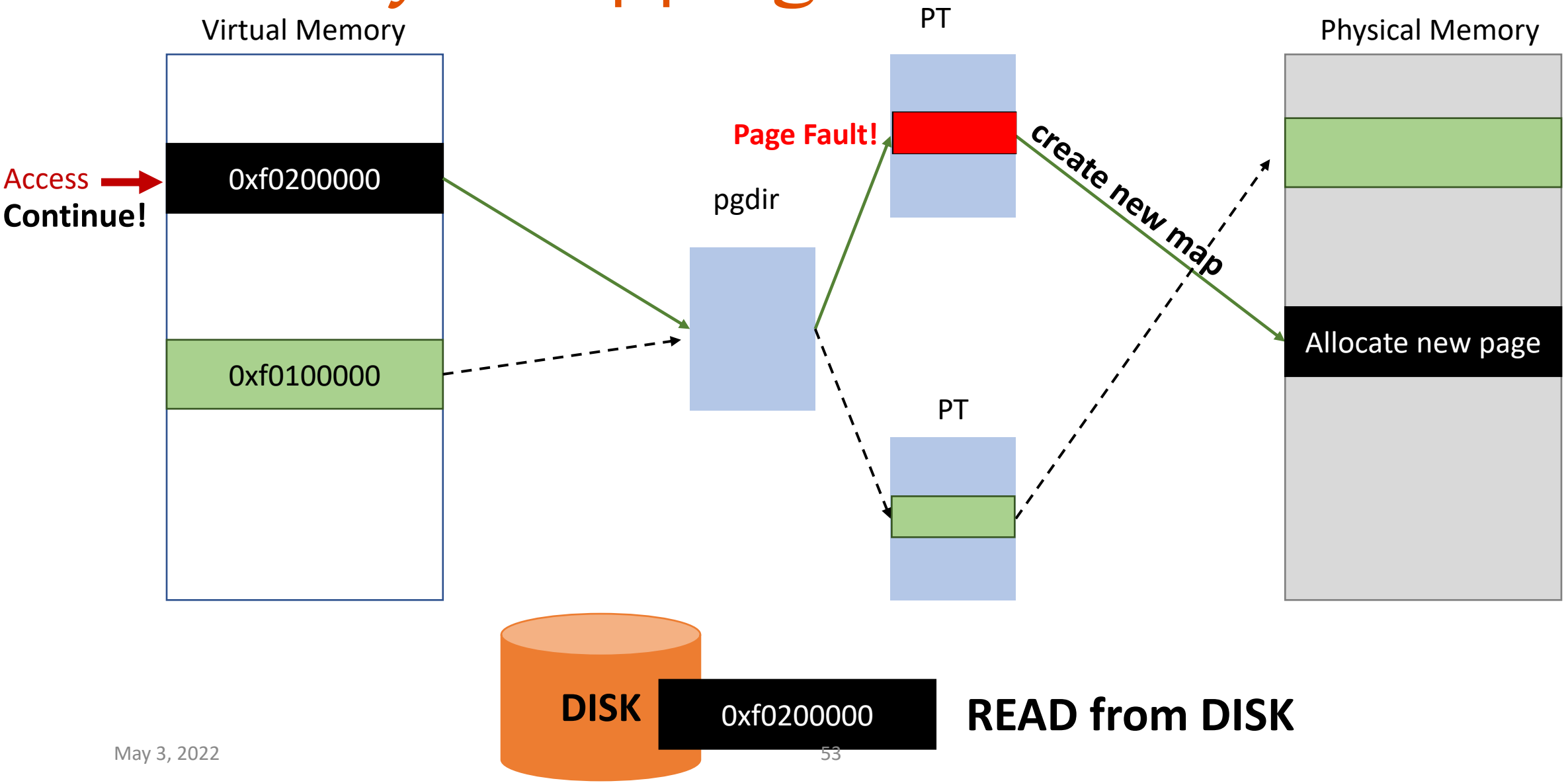


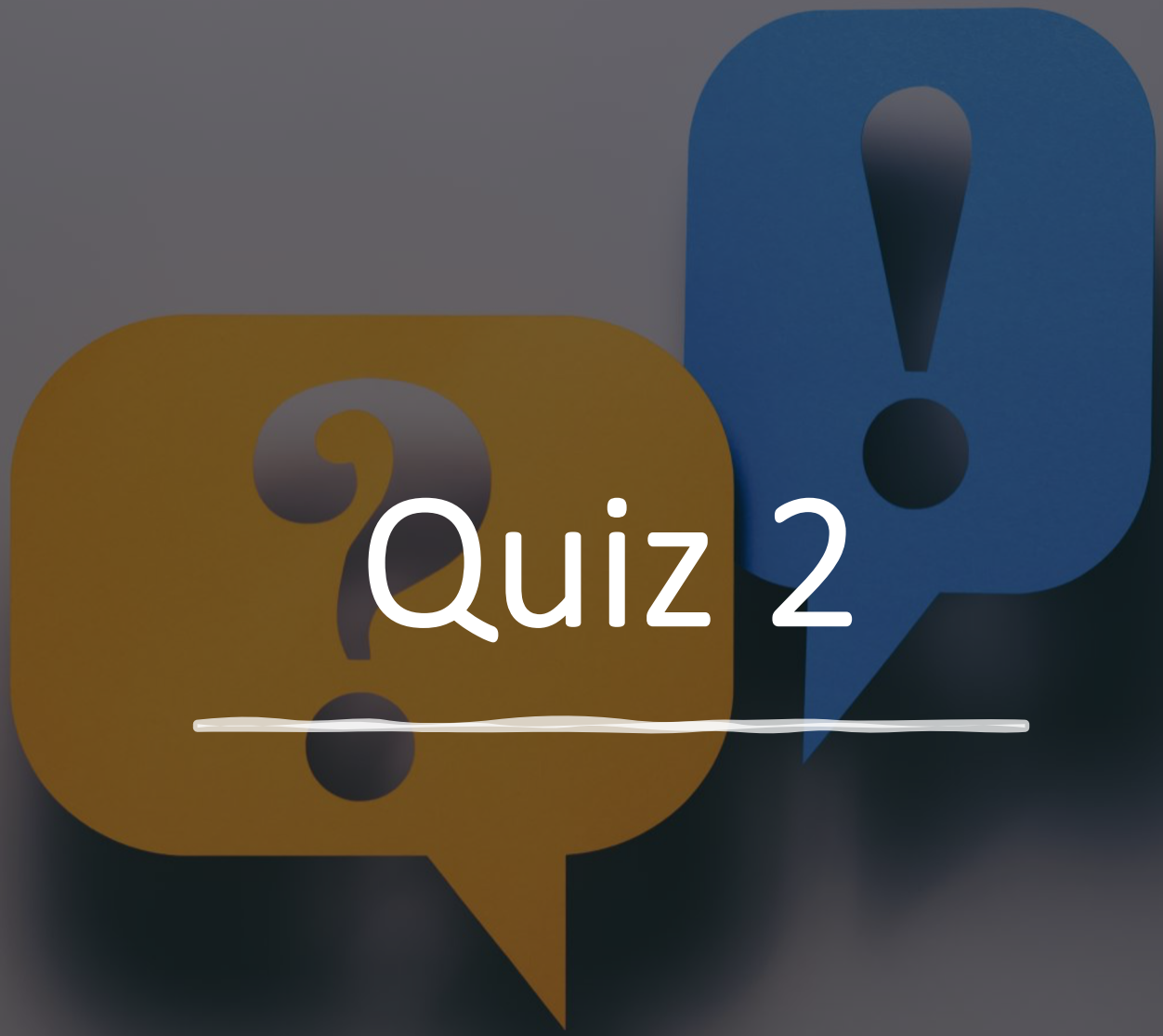
Swapping | OS

- Page fault handler
  - Read CR2
  - get address [`0xf0200000`]
  - Read error code
- If error code → page not present fault **and**
- **faulting page is stored in the disk**
- Lookup disk if it swapped out 0xf0200000 of this environment [process]
  - This must be **per process** because **virtual address is per-process resource**

- **Load that page into physical memory**
- Map it and then continue!

Memory Swapping





Quiz 2

Quiz 2

- **May 5, 2022** [**Thursday**] at **8:30 AM**!
- Available until **May 6, 2022** [**Friday**], **11:59 PM**

- **Open** materials [slides, videos, code, and textbook]
- Unlimited time
- **Two attempts**!

- Don’t forget about the lab3 due date: **May 13, 2022**

Quiz 2 Topics

JOS Lab 2: Memory Management

JOS Lab 3: User Environment

Lecture 7: User and Kernel Spaces

Lecture 8: Handling Interrupt & Exceptions

Lecture 9: System Calls and Page Faults

Sample Questions

Memory Protection

- How does an OS/CPU apply access control to memory?
- **Protected mode (DPL), Page directory / page table (permission flags, PTE_W & PTE_U)**

- How does an OS kernel protect itself against attacks from application code?
- **Removing PTE_U from PDE or PTE**

- How does an OS protect a read-only memory area from write attempts?
- **Removing PTE_W from PDE or PTE**

- How does an OS isolate the memory space of a process from other processes?
- **Separate page directory / page tables**

Sample Questions

Memory Overhead Calculations

- Consider the following mapping for a program. How much physical memory is required to support virtual to physical address translation for this program (compute the minimal total size of the page directory and page tables that enables this allocation)?

Area	Start virtual addr	End virtual addr	Size
.text (code)	0x800000	0x804000	0x4000
.data (read/write)	0x900000	0x902000	0x2000
.bss	0xc00000	0xd00000	0x100000

Sample Questions

Memory Overhead Calculation

Area	Start virtual addr	End virtual addr	Size
.text (code)	0x800000	0x804000	0x4000
.data (read/write)	0x900000	0x902000	0x2000
.bss	0xc00000	0xd00000	0x100000

Index	Address range	PTE
0	0x0 ~ 0x400000	invalid
1	0x400000 ~ 0x800000	invalid
2	0x800000 ~ 0xc00000	valid
3	0xc00000 ~ 0x1000000	valid
…	…	Invalid
0x3ff	0xffc00000 ~ 0xffffffff	Invalid

1 page directory: **4KB**
2 page tables: **2 * 4KB = 8KB**

4KB + 8KB = **12KB**

Sample Questions

User / Kernel Switch

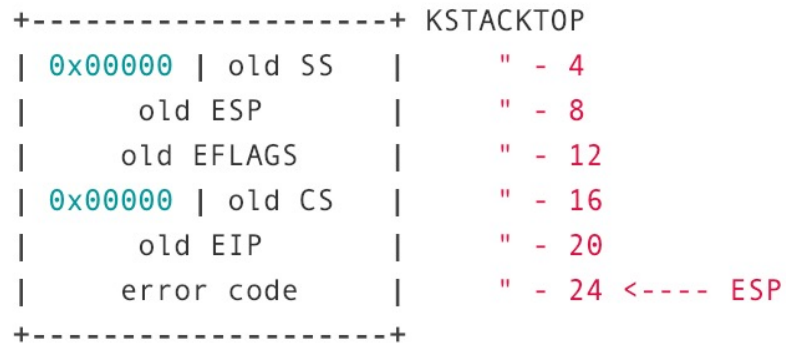
- How does OS get back CPU execution if user runs while(1);?
- **Timer interrupt will preempt the execution from user to kernel**

- How does a user program access hardware? What does OS do for this?
- **OS offers system calls (APIs available in OS)**
- **User program invokes system call via generating a software interrupt**
- **OS checks access to resources**
  - **File, network, memory, etc.**

Sample Questions

For an interrupt that has an error code,

1. which part of TrapFrame is prepared by the CPU?
   - **tf_ss, tf_esp, tf_eflags, tf_cs, tf_eip and tf_err**
2. which part of TrapFrame is prepared by JOS?
   - **All others: tf_trapno, tf_ds, tf_es, tf_regs**




```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when c
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Sample Questions

Page Fault

- We run 1,000,000 instances of `/bin/bash` on our os2 server
- How many copies of the code (read-only part) of /bin/bash exist in the physical memory?
- **One**
- **shared via copy-on-write**

- How does an OS run a program that requires more memory than what is available a machine's physical memory?
- **Store currently unused memory pages in the disk (swap-out)**
- **Accessing to swapped-out pages will generate a page fault**
- **The OS can search for swapped-out pages, and fill a page in if exists (swap-in)**
- **Resumes user execution!**