

# CS444/544

# Operating Systems II

Lecture 10

System Calls and Page Fault

5/6/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

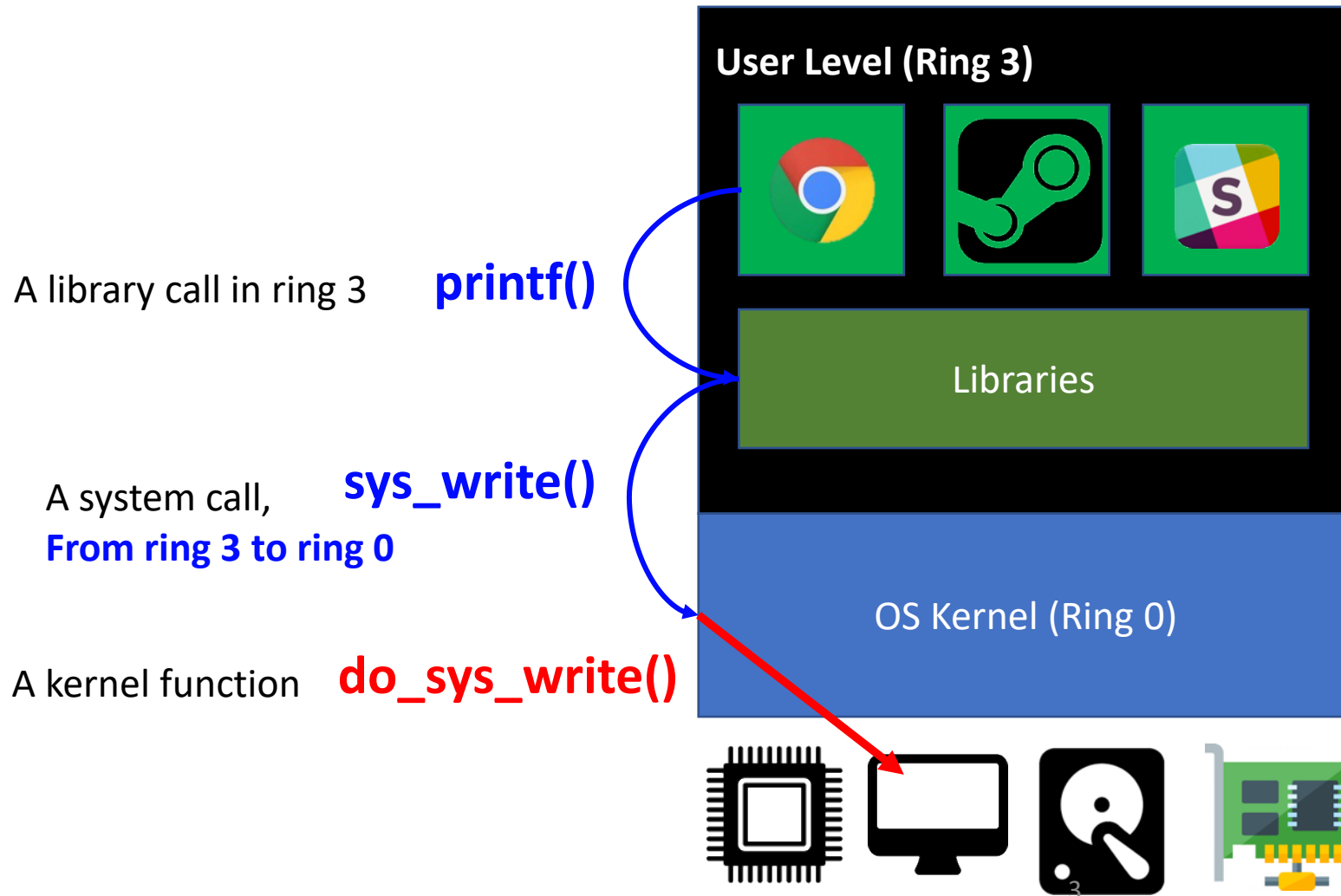


**Oregon State**  
**University**

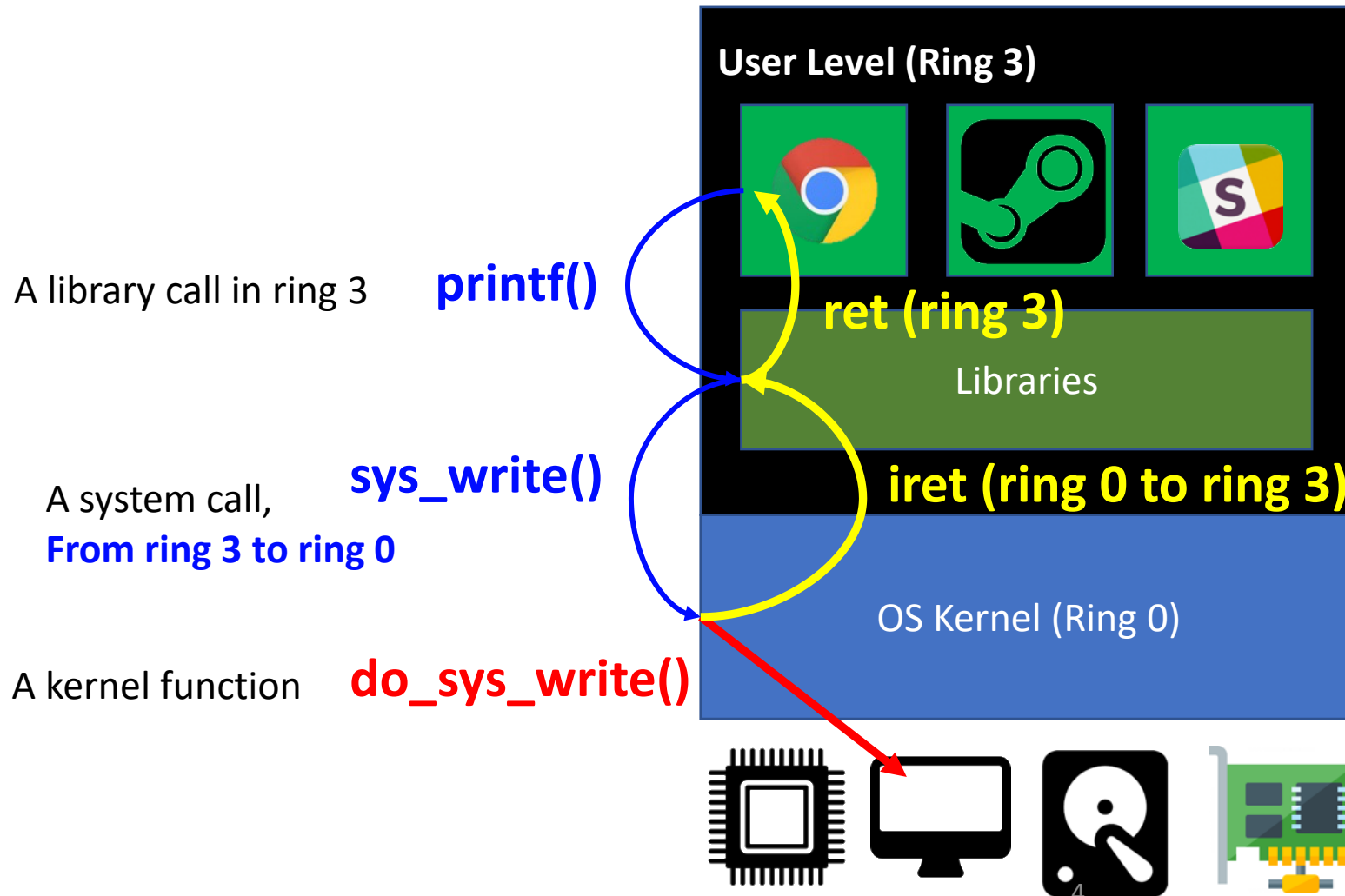
# Reminders

- 75% due for lab 2: today's midnight
- Quiz 2 next Monday
  - Review and prep. on Wednesday's lecture

# Recap: A High-level Overview of User/Kernel Execution



# Recap: A High-level Overview of User/Kernel Execution



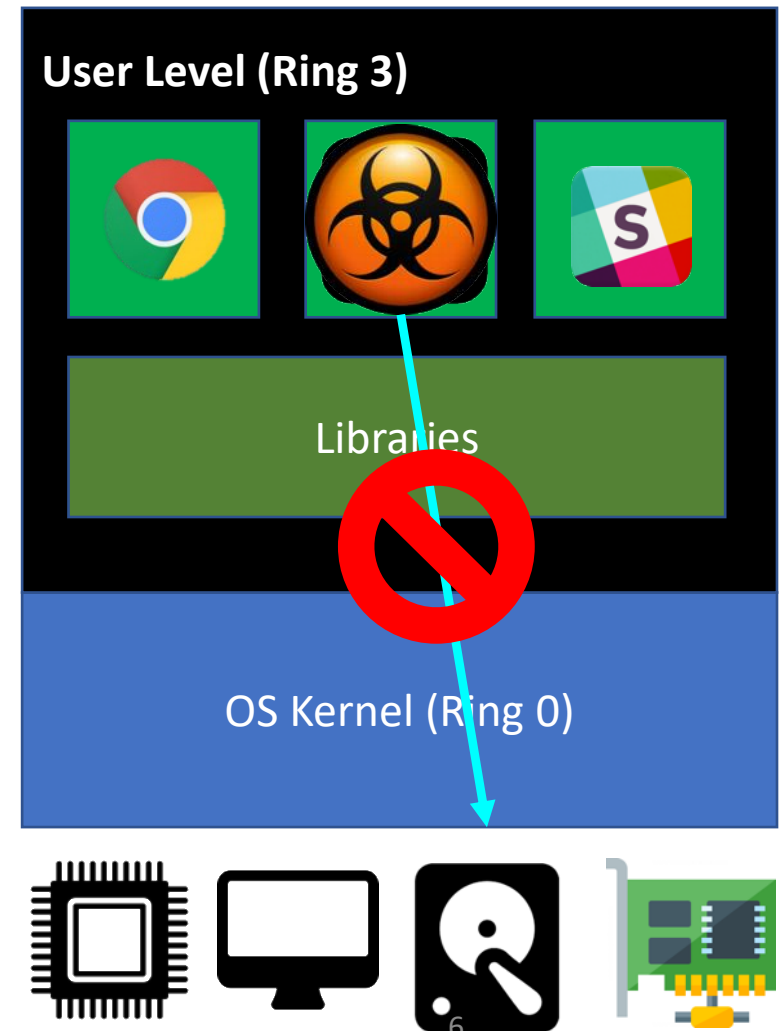
# Today's Topic

- More about System Call
  - Privilege separation and call gate
- Page Fault
  - How does an OS handle a fault and resume the execution?
  - For what purpose?
    - Automatic stack allocation
    - Copy-on-write
    - Swap

# Ring 3 (User) and Ring 0 (Kernel)

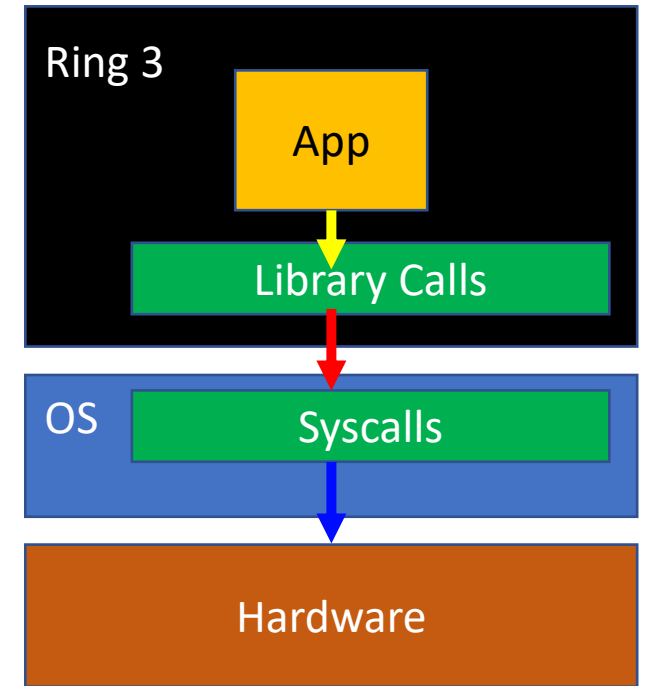
- Why do we have privilege separation?
  - Security!
- We do not know what application will do
  - Do not allow dangerous operations to system
    - Flash BIOS, format disk, deleting system files, etc.
  - Only the OS can access hardware
    - Apply access control on accessing hardware resources!
    - E.g., only the administrator can format disk

OS must mediate hardware access request from userspace, and we handle this via system calls



# Library Calls vs. System Calls

- Library Calls
  - APIs in Ring 3
  - DO NOT include operations in Ring 0
    - **Cannot access hardware directly**
  - Could be a wrapper for some computation or
  - Could be a wrapper for system calls
    - E.g., `printf()` internally uses `write()`, which is a system call
- Some system calls are available as library calls
  - As wrappers in Ring 3



```
NAME
    read - read from a file descriptor

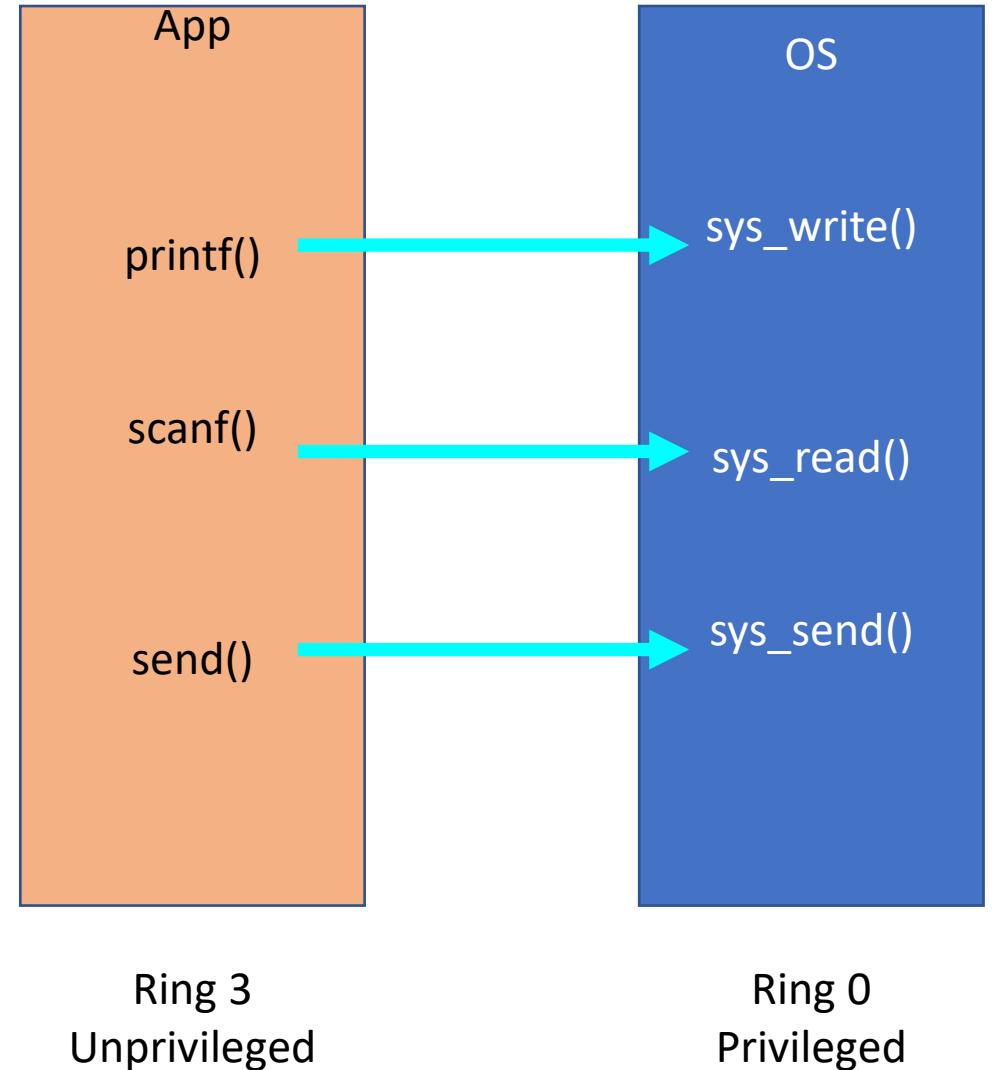
SYNOPSIS
    #include <unistd.h>

    ssize_t read(int fd, void *buf, size_t count);
```

# Library Calls vs. System Calls

- System Calls

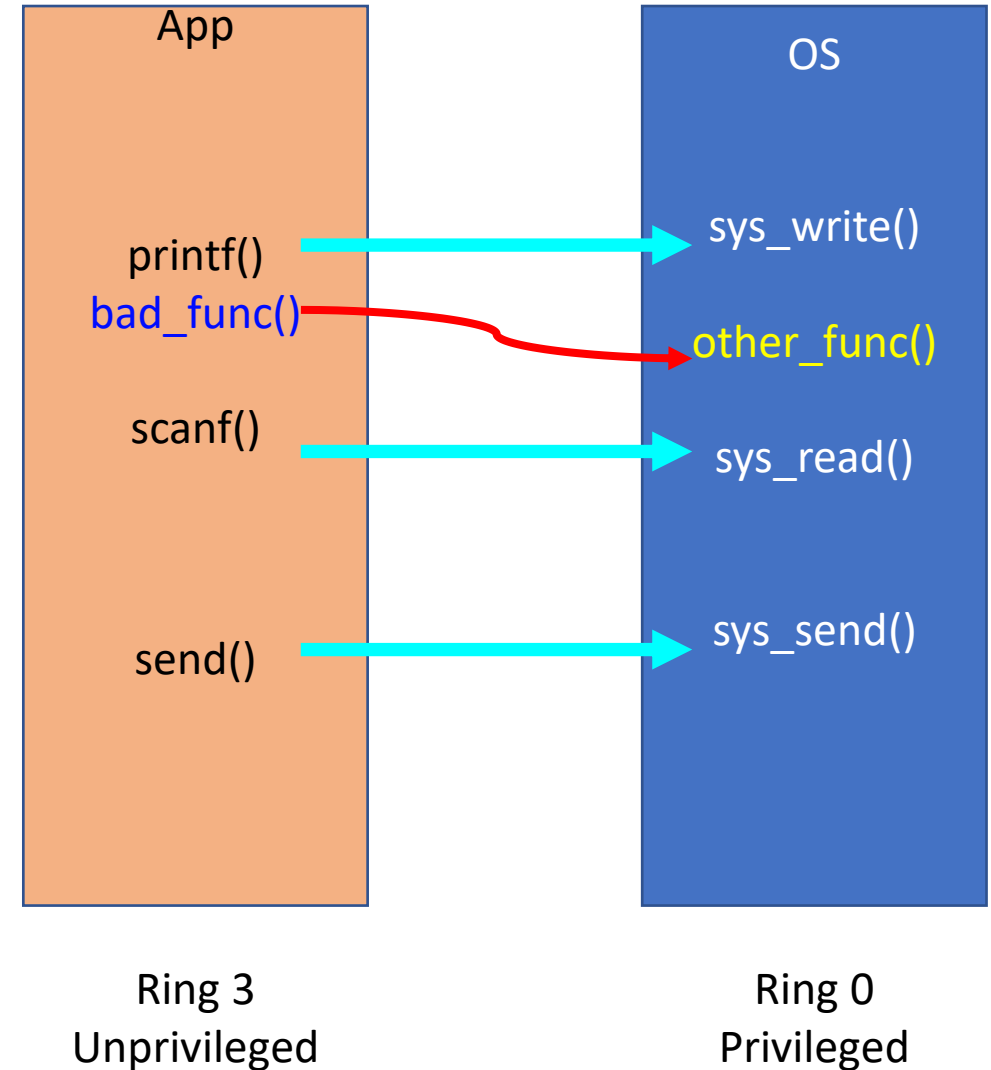
- APIs in Ring 0
- OS's abstraction for hardware interface for user space
- Called when Ring 3 application need to perform Ring 0 operations





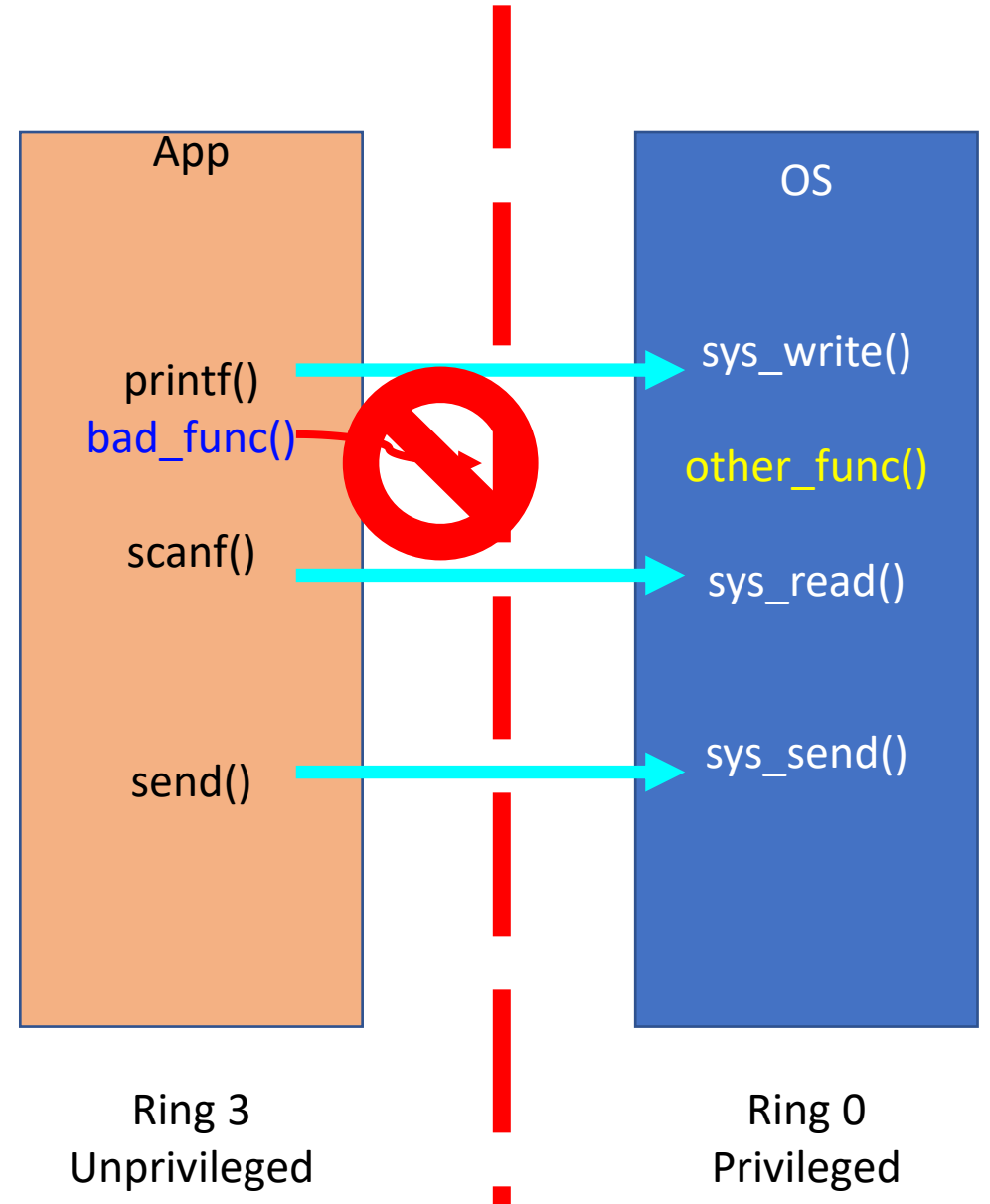
# System Call Design

- Application should not call arbitrary function
  - If so, app can do all operations that OS can do; privilege separation is meaningless!
- How can we avoid this, in other words, how can we **restrict apps to invoke system calls only** but **not other OS functions**?



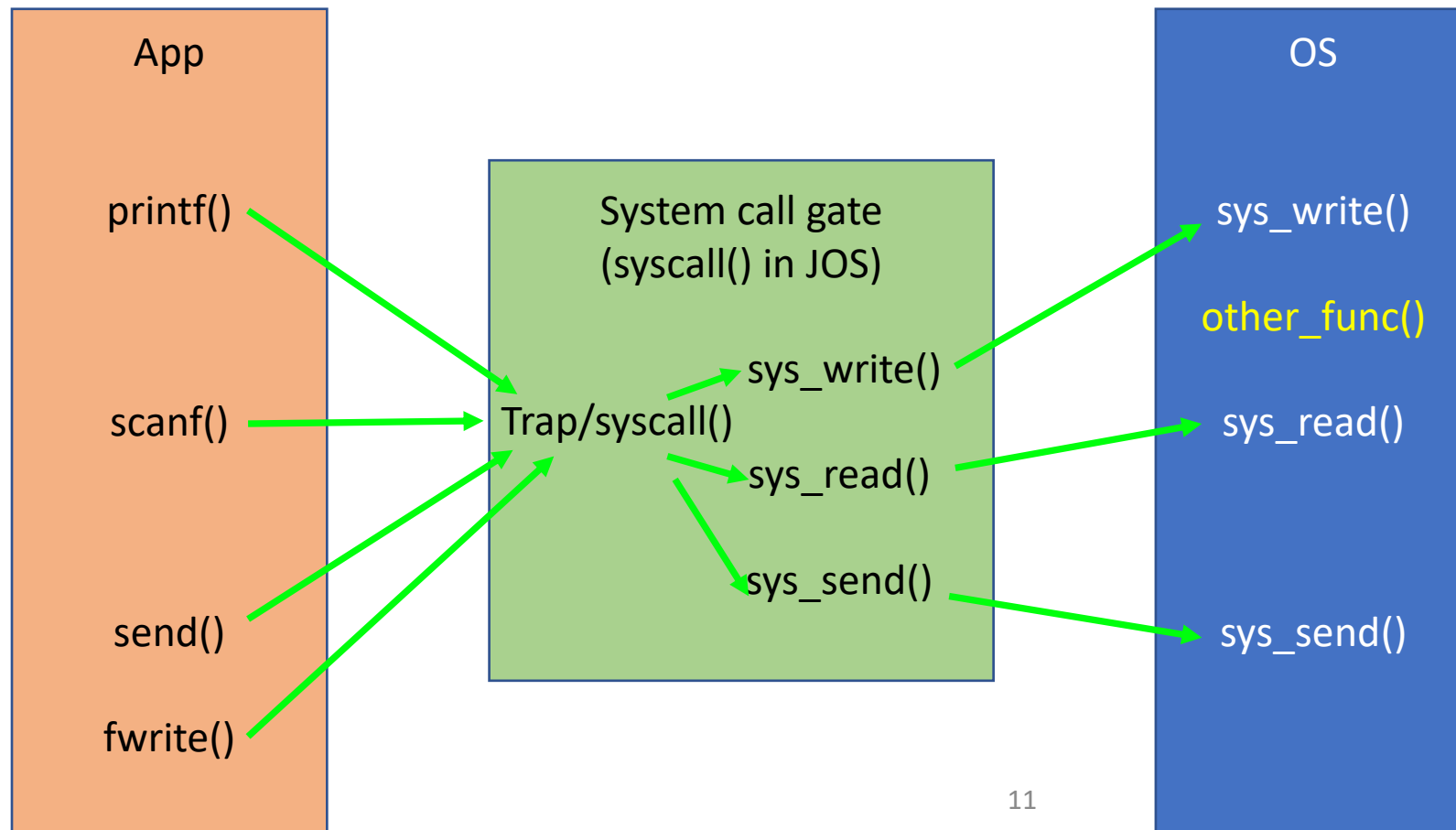
# System Call Design

- Application should not call arbitrary function
  - If so, app can do all operations that OS can do; privilege separation is meaningless!
- How can we avoid this, in other words, how can we **restrict apps to invoke system calls only** but **not other OS functions**?



# Secure System Call Design: Call Gate via Interrupt Handling

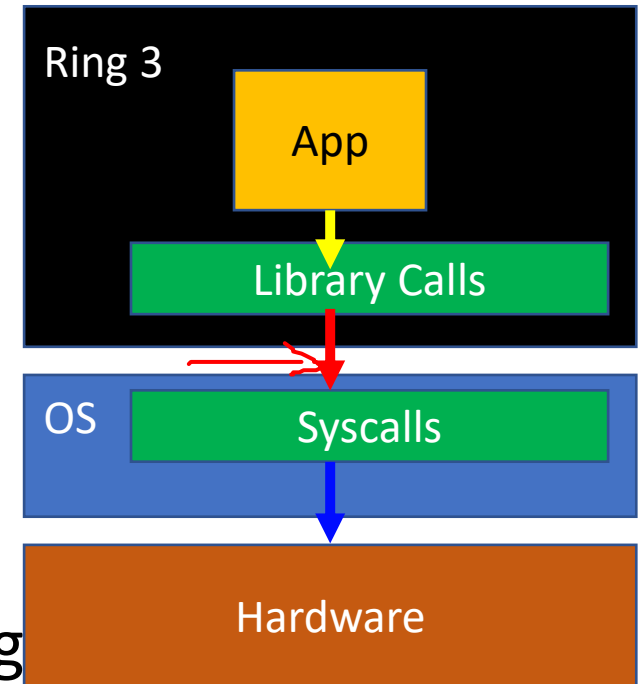
- Call gate: a secure method to control access to Ring 0!



# Call Gate via Interrupt Handling

- Call gate
  - System call can be invoked only with trap handler
    - `int $0x30` – in JOS
    - `int $0x80` – in Linux (32-bit)
    - `int $0x2e` – in Windows (32-bit)
    - `sysenter/sysexit` (32-bit)
    - `syscall/sysret` (64-bit)
- OS performs checks if user space is doing a right thing
  - Before performing important ring 0 operations
  - E.g., accessing hardware..

`int $0x30`  
**CHECK!!**



# An Example of Protecting Syscalls via Call Gate

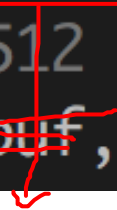
- How can we protect 'read()' system call?
  - `read(int fd, void *buf, size_t count)`
  - Read `count` bytes from a file pointed by `fd` and store those in `buf`
- Usage

```
// buffer at the stack
char buf[512];
// read 512 bytes from standard input
read(0, buf, 512);
```

# An Example of Protecting Syscalls via Call Gate

- Problem: what will happen if we call...

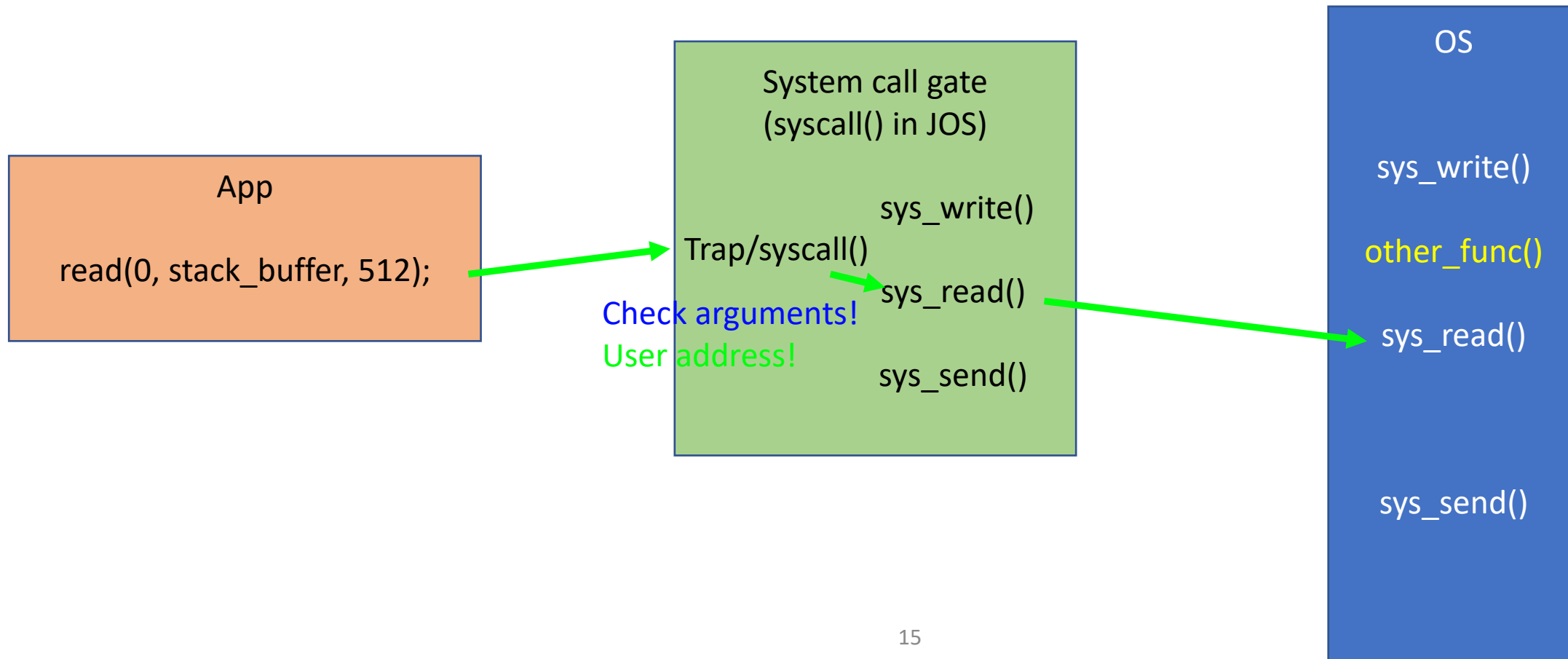
```
// kernel address will points to a dirmap of
// the physical address at 0x100000
char kernel_address = KERNBASE + 0x100000;
// read 512 bytes from standard input
read(0, buf, 512);
```



- This is trying to **overwrite kernel code** with your keystroke typing..
  - If this was allowed, changing kernel code from Ring 3 is possible!

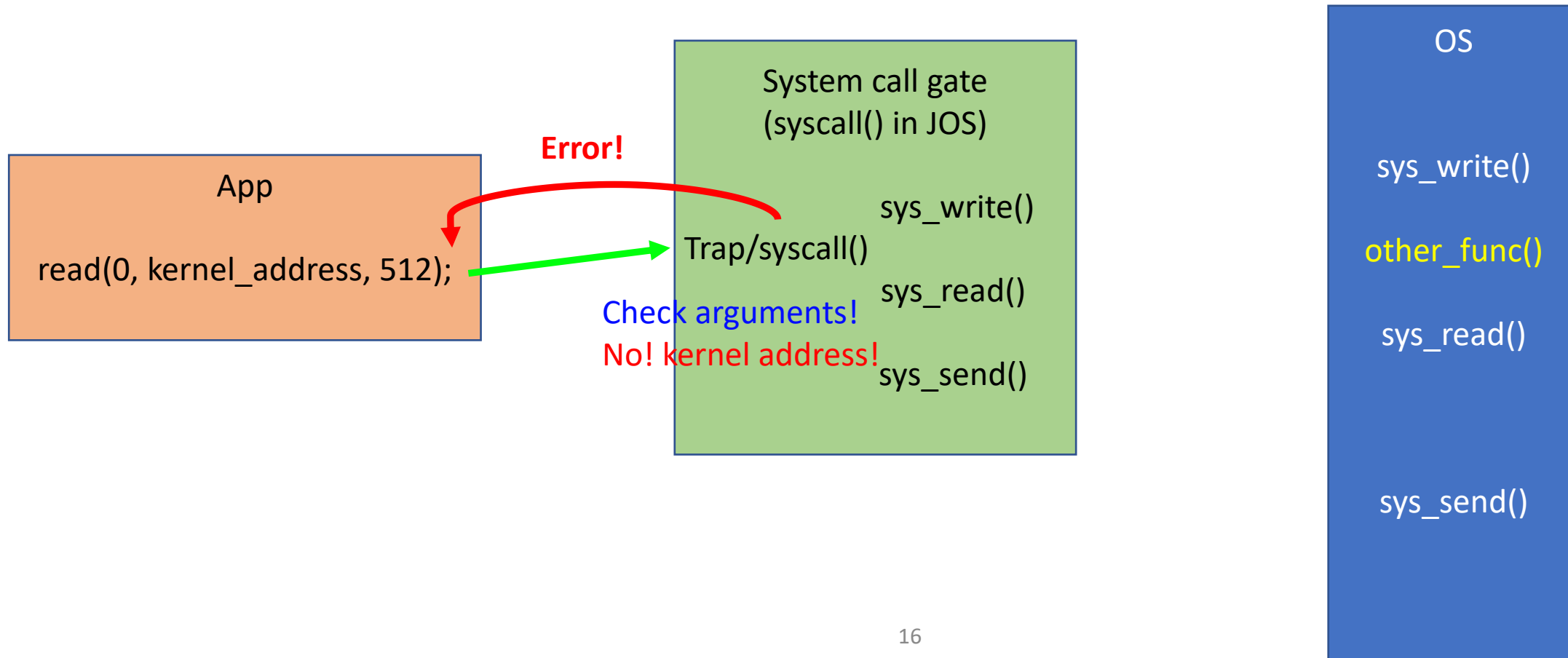
# How Call Gate Works?

- We can hook all syscalls from Ring 3 at our syscall trap handler



# Call Gate

- We can hook all syscalls from Ring 3 at our syscall trap handler





# Test

```
1 #include <stdio.h>
2
3 int main() {
4     // stack buffer
5     char buf[512];
6
7     // read 512 bytes from console into stack buffer
8     int ret = read(0, buf, 512);
9
10    printf("Read to stack memory returns: %d\n", ret);
11
12    // read 512 bytes from console into kernel addr
13    ret = read(0, (void*) 0xffffffff01000000, 512);
14
15    printf("Read to kernel memory returns: %d\n", ret);
16    perror("Reason for the error:");
17
18    return 0;
19 }
```

```
os2 ~/cs444/s21 173% a.out
abcd
Read to stack memory returns: 5
Read to kernel memory returns: -1
Reason for the error:: Bad address
```

# Check How System Calls are Invoked in Linux Kernel

- Use `strace` in Linux, e.g., `$ strace /bin/ls`

```
read(0, "asdfzxcv\n", 512) = 9
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
brk(NULL) = 0x18c5000
brk(0x18e6000) = 0x18e6000
write(1, "Read to stack memory returns: 9\n", 32) = 32
read(0, 0xffffffff01000000, 512) = -1 EFAULT (Bad address)
write(1, "Read to kernel memory returns: -"..., 34) = 34
dup(2) = 3
fcntl(3, F_GETFL) = 0x8001 (flags O_WRONLY|O_LARGEFILE)
close(3) = 0
write(2, "Reason for the error:: Bad addre"..., 35Reason for the error:: Bad address
```

# Summary: System Call / Call Gate

- Prevent Ring 3 from accessing hardware directly
  - Security reasons!
  - OS mediates hardware access via system calls
- You may regard system calls as APIs of an OS
- How to prevent an application from running arbitrary ring 0 operation?
  - Call gate
- Modern OS use call gate to protect system calls
  - At trap handler, an OS can apply access control to system call request

# Handling Fault: Page Fault

- Faults
  - Faulting instruction has not executed (e.g., page fault)
  - Resume the execution after handling the fault
- Resume the execution after handling the fault

# Page Fault: A Case of Handling Faults

- Occurs when paging (address translation) fails
  - `!(pde&PTE_P) or !(pte&PTE_P)` : invalid translation
  - Write access but `!(pte&PTE_W)` : access violation
  - Access from user but `!(pte&PTE_U)` : protection violation

# Page Fault: an Example

- Accessing a Kernel address from User

```
int main() {  
    char *kernel_memory = (char*)0xf0100000;  
    // I am a bad guy, and I would like to change  
    // some contents in kernel memory  
    kernel_memory[100] = '!';  
}
```

```
0x00800039 ? movb    $0x21,0xf0100064
```

# Page Fault: an Example

- Accessing a Kernel address from User

```
int main() {  
    char *kernel_memory = (char*)0xf01  
    // I am a bad guy, and I would lik  
    // some contents in kernel memory  
    kernel_memory[100] = '!';  
}
```

```
0x00800039 ? movb $0x21, 0x
```

TRAP frame at 0xf01c0000

edi 0x00000000

esi 0x00000000

ebp 0xeebdfd0

oesp 0xefffffffdc

ebx 0x00000000

edx 0x00000000

ecx 0x00000000

eax 0xeec00000

es 0x----0023

ds 0x----0023

trap 0x0000000e Page Fault

cr2 0xf0100064

err 0x00000007 [user, write, protection]

eip 0x00800039

cs 0x----001b

flag 0x00000096

esp 0xeebdfdb8

ss 0x----0023

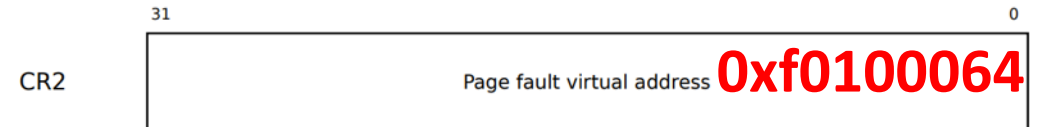
[00001000] free env 00001000

# Page Fault: What Does CPU Do?

- CPU let OS know why and where such a page fault happened
  - CR2: stores the address of the fault
  - Error code: stores the reason of the fault

```
TRAP frame at 0xf01c0000
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xeffffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0xeec00000
es 0x---0023
ds 0x---0023
trap 0x0000000e Page Fault
cr2 0xf0100064
err 0x00000007 [user, write, protection]
eip 0x00800039
cs 0x---001b
flag 0x00000096
esp 0xeebdfb8
```

```
kernel_memory[100] = '!'; 00001000
```



31	15	5	4	3	2	1	0		
Reserved		SGX	Reserved		PK	I/D	U/S	W/R	P

**111**

P 0 The fault was caused by a non-present page.  
1 The fault was caused by a page-level protection violation.

W/R 0 The access causing the fault was a read.  
1 The access causing the fault was a write.

U/S 0 A supervisor-mode access caused the fault.  
1 A user-mode access caused the fault.

RSVD 0 The fault was not caused by reserved bit violation.  
1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.

I/D 0 The fault was not caused by an instruction fetch.  
1 The fault was caused by an instruction fetch.

PK 0 The fault was not caused by protection keys.  
1 There was a protection-key violation.

SGX 0 The fault is not related to SGX.  
1 The fault resulted from violation of SGX-specific access-control requirements.

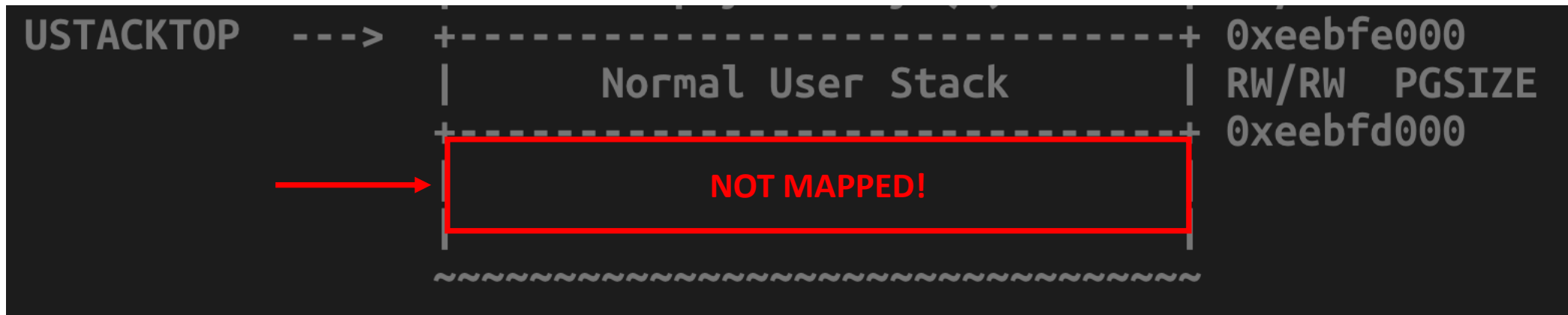


# CPU/OS Execution Example

- User program accesses 0xf0100064
- CPU generates page fault (pte&PTE\_U == 0)
  - Put the faulting address on CR2
  - Put an error code
  - Calls page fault handler in IDT
- OS: page\_fault\_handler
  - Read CR2 (address of the fault, 0xf0100064)
  - Read error code (contains the reason of the fault)
  - Resolve error (if not, destroy the environment)
  - Continue user execution
- User: resume on that instruction (or destroyed by the OS)

# Fault Resume Example: Stack Overflow

- inc/memlayout.h
- We allocate one (1) page for the user stack



- If you use a large local variable on the stack
  - Stack overflow (stack grows down...)

```
int func() {  
    char buf[8192];  
    buf[0] = '1';  
}
```

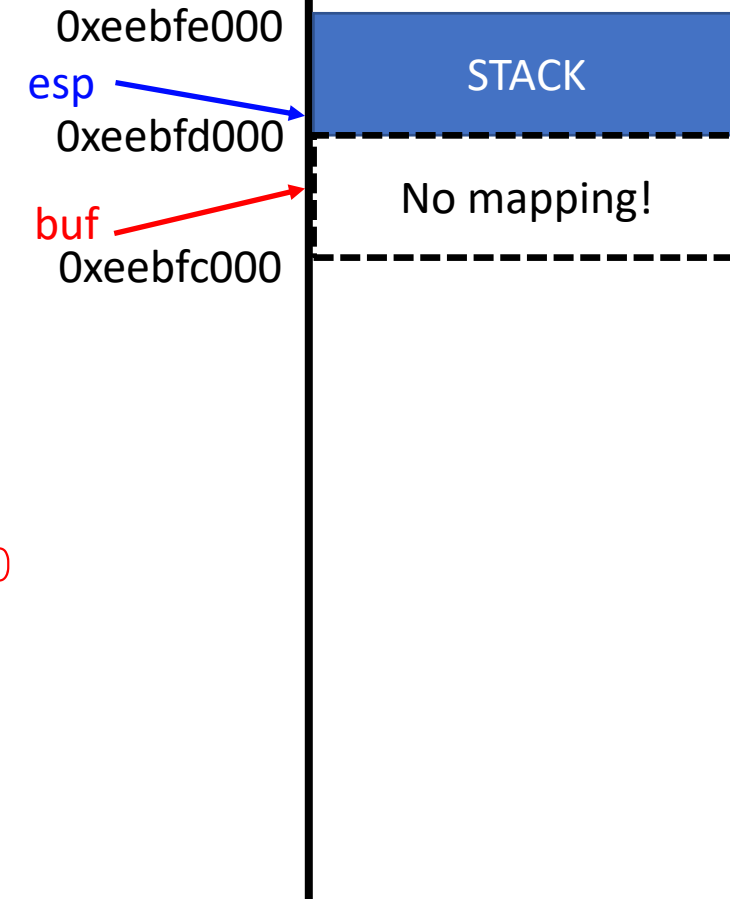
# Some Idea: Allocating New Stack Automatically

- Can we detect such an access and allocate a new page for the stack automatically?
  - Yes
- We will utilize 'Page Fault'
- Observations
  - Stack overflow would be sequential (access pages adjacent to the stack)
  - We should catch both read/write access (both should fault)

# Example: New Stack Allocation by Fault (User)

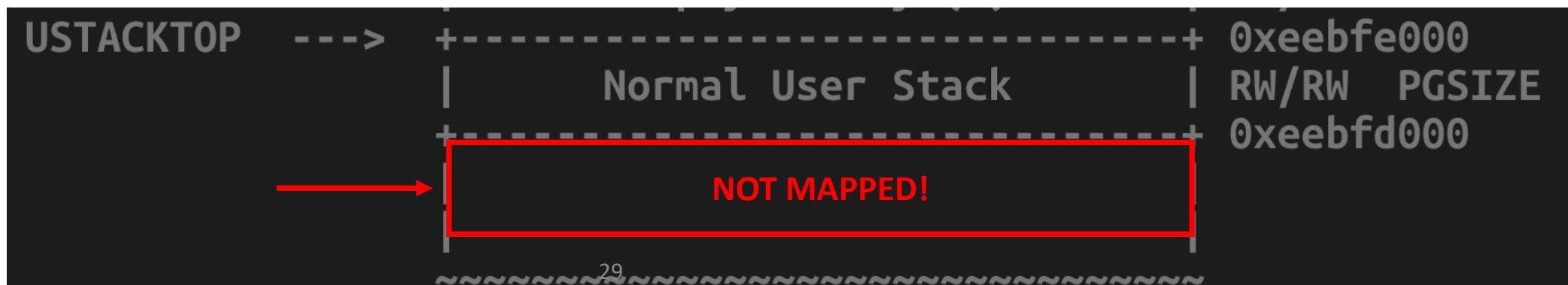
```
int func() {  
    char buf[32];  
    for(int i=0; i<32; ++i) {  
        → buf[i] = '1' + i;  
    }  
}
```

- Stack ends at 0xeebfd000
- Suppose the current value of esp (stack) is
  - 0xeebfd010
- User program creates a new variable: char buf[32]
  - buf = 0xeebfcff0
  - Buffer range: 0xeebfcff0 ~ 0xeebfd010
- On accessing buf[0] = '1';
  - movb \$0x31, (%eax)
  - eax = 0xeebfcff0 No translation for 0xeebfc000
  - **Need to allocate 0xeebfc000 ~ 0xeebfd000**



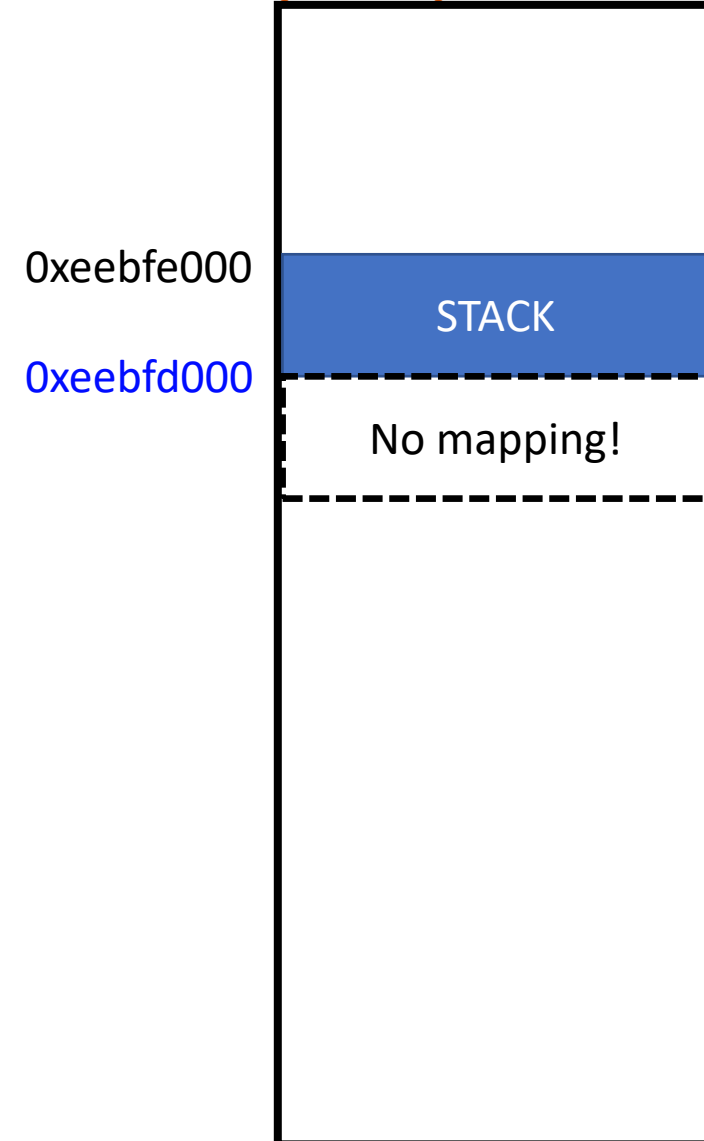
# Example: New Stack Allocation by Fault (CPU)

- Lookup page table
  - No translation!
- Store `0xeebfcff0` to CR2
- Set error code
  - “The fault was caused by a non-present page!”
- Raise page fault exception (interrupt #14) -> call page fault handler



# Example: New Stack Allocation by Fault (OS)

- Interrupt will make CPU invoke the `page_fault_handler()`
- Read CR2
  - `0xeebfcff0`, it seems like the page right next to current stack end
  - The current stack end is: `0xeebfd000`
- Read error code
  - “The fault was caused by a non-present page!”
- Let’s allocate a new page for the stack!



# Example: New Stack Allocation by Fault (OS)

- Allocate a new page for the stack

- `Struct PageInfo *pp = page_alloc(ALLOC_ZERO);`

- Get a new page, and wipe it to have all zero as its contents

- `page_insert(env_pgdir, pp, 0xeebfc000, PTE_U|PTE_W);`

- Map a new page to that address!

- `iret!`

0xeebfe000

0xeebfd000

0xeebfc000

STACK

STACK

# Example: New Stack Allocation by Fault (User-Return)

- On accessing `buf[0] = '1';`
  - `movb $0x31, (%eax)`
  - `eax = 0xeebfcff0` No translation for `0xeebfc000`
- Execute the faulting instruction again: `buf[0] = '1';`
  - `movb $0x31, (%eax)`
  - `eax = 0xeebfcff0` Now translation is valid!
- Continue to execute the loop..

0xeebfe000

STACK

0xeebfd000

STACK

0xeebfc000

By exploiting **page fault and its handler**, we can implement **automatic allocation of user stack!**

```
int func() {  
    char buf[32];  
    for(int i=0; i<32; ++i) {  
        buf[i] = '1' + i;  
    }  
}
```



# Other Useful Examples of Using Page Fault (in Modern OSes)

- Copy-on-Write (CoW)
  - Technique to reduce memory footprint
  - Share pages read-only
  - Create a private copy when the first write access happens
- Memory Swapping
  - Use disk as extra space for physical memory
  - Limited RAM Size: 16GB?
  - We have a bigger storage: 1T SSD, Hard Disk, online storage, etc.
  - Can we store some 'currently unused but will be used later' part into the disk?
    - Then we can store only the active part of data in memory

# Copy-on-Write (CoW) to Reduce Memory Footprint

- Think about our os2 server

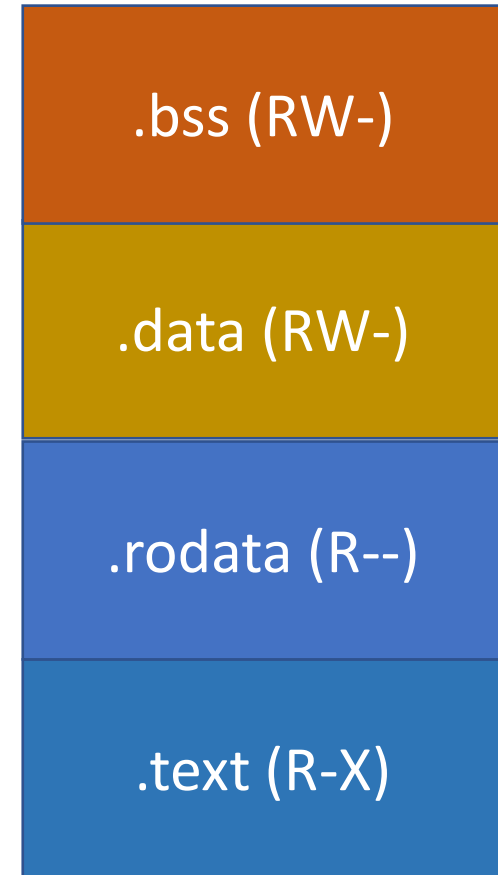
```
os2 ~/cs444/s21 186% ps aux | grep bash | wc -l
110
os2 ~/cs444/s21 187% ps aux | grep tmux | wc -l
23
os2 ~/cs444/s21 188% ps aux | grep gdb | wc -l
13
```

Count number of processes running bash, tmux, and gdb

- Will run many /bin/bash, /usr/bin/gdb, /usr/bin/tmux, etc.
  - Each of you will run those programs!!
  - Do we need to have 110 copies of the same program in memory?
- How can we build an OS to efficiently load them and minimize memory usage?
  - Share physical pages of the same program!

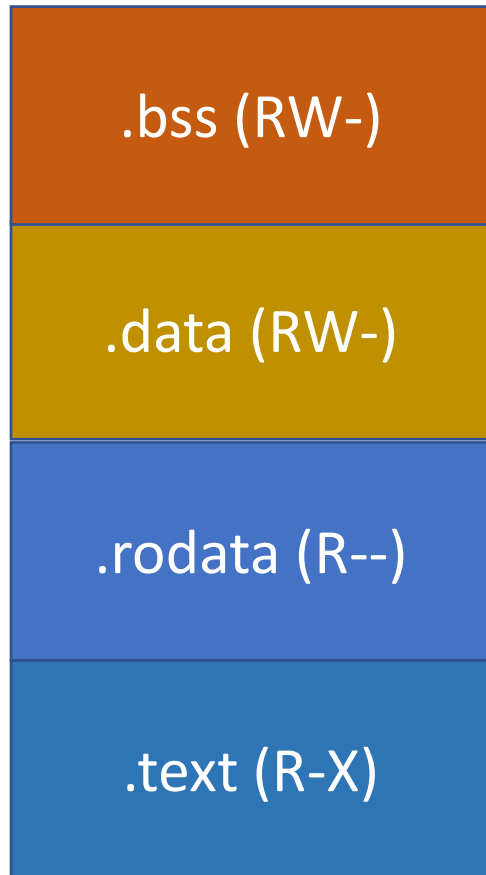
# A Program

- .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

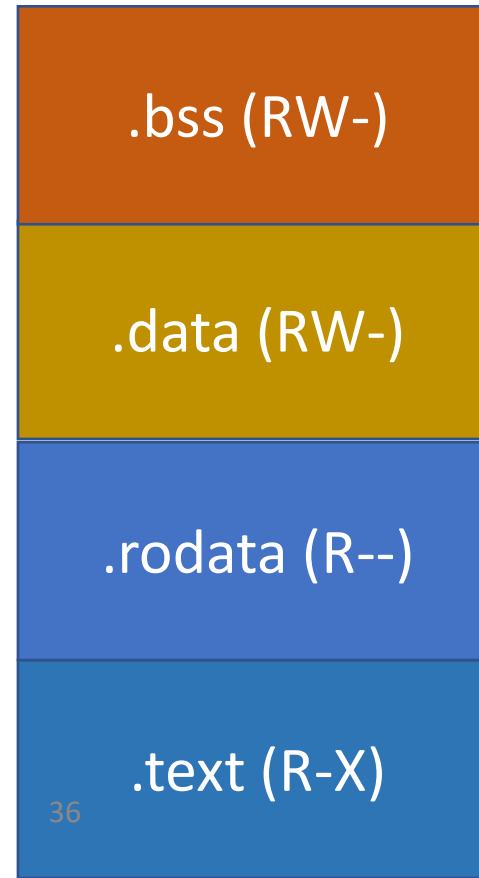


# Running the Same Program...

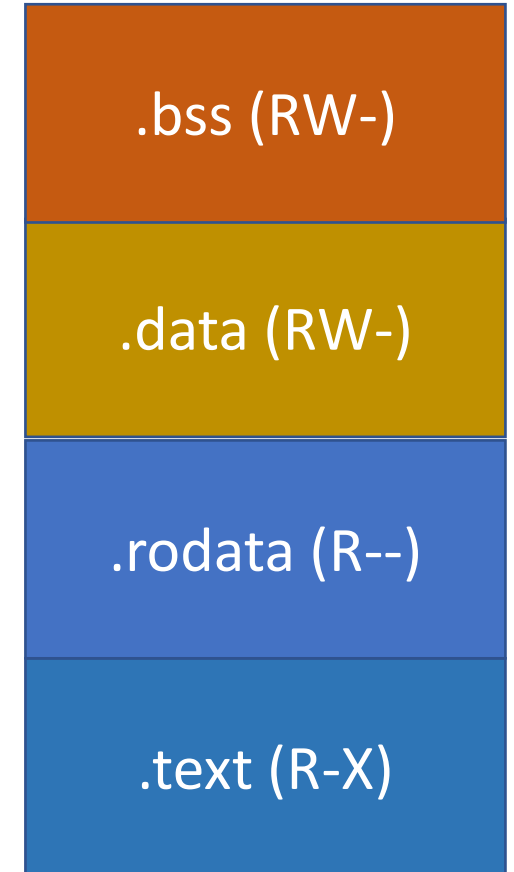
Do we need to copy the same data for each process creation?



Process 1

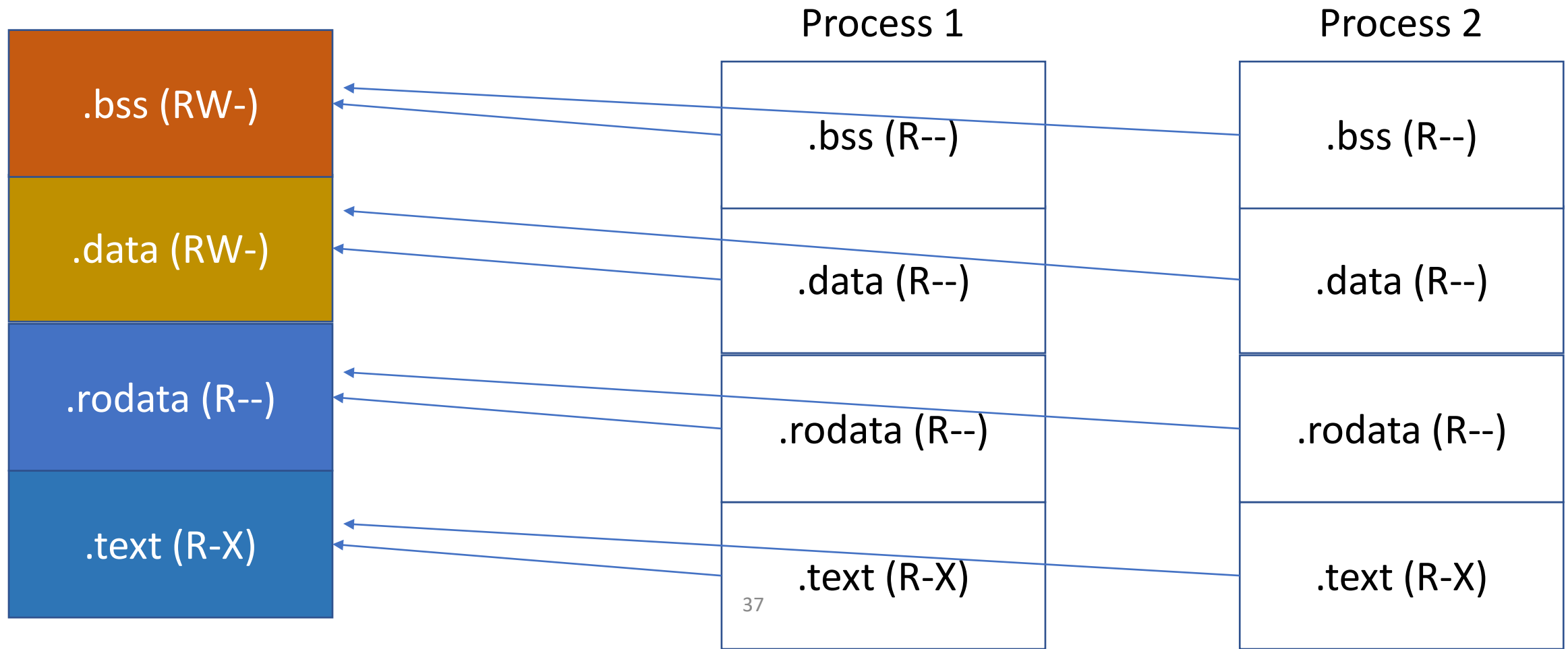


Process 2



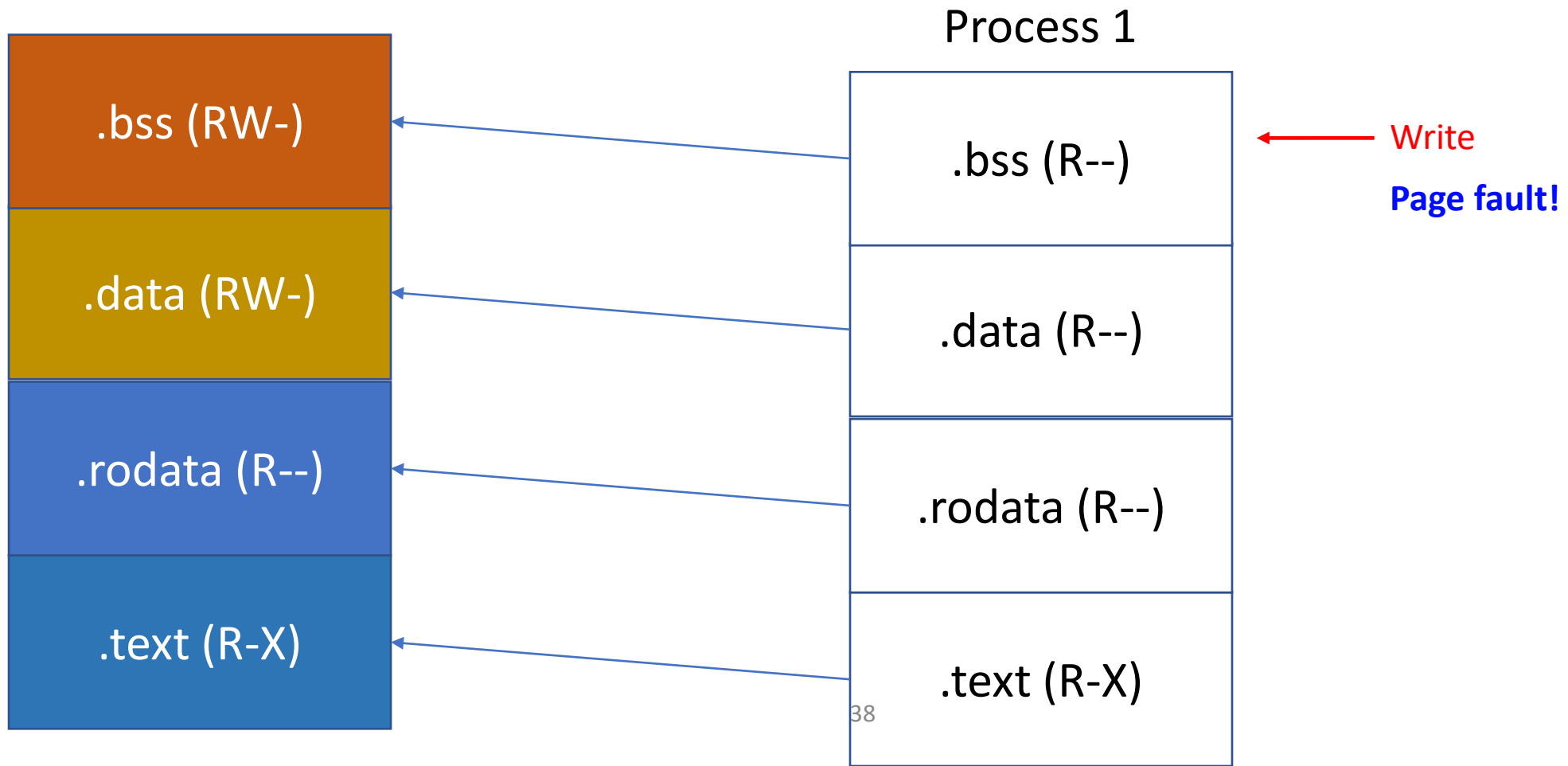
# Sharing by Read-only

- Set page table to map the same physical address to share contents



# OK for Read-only Sections

- How can Process 1 write on .bss???

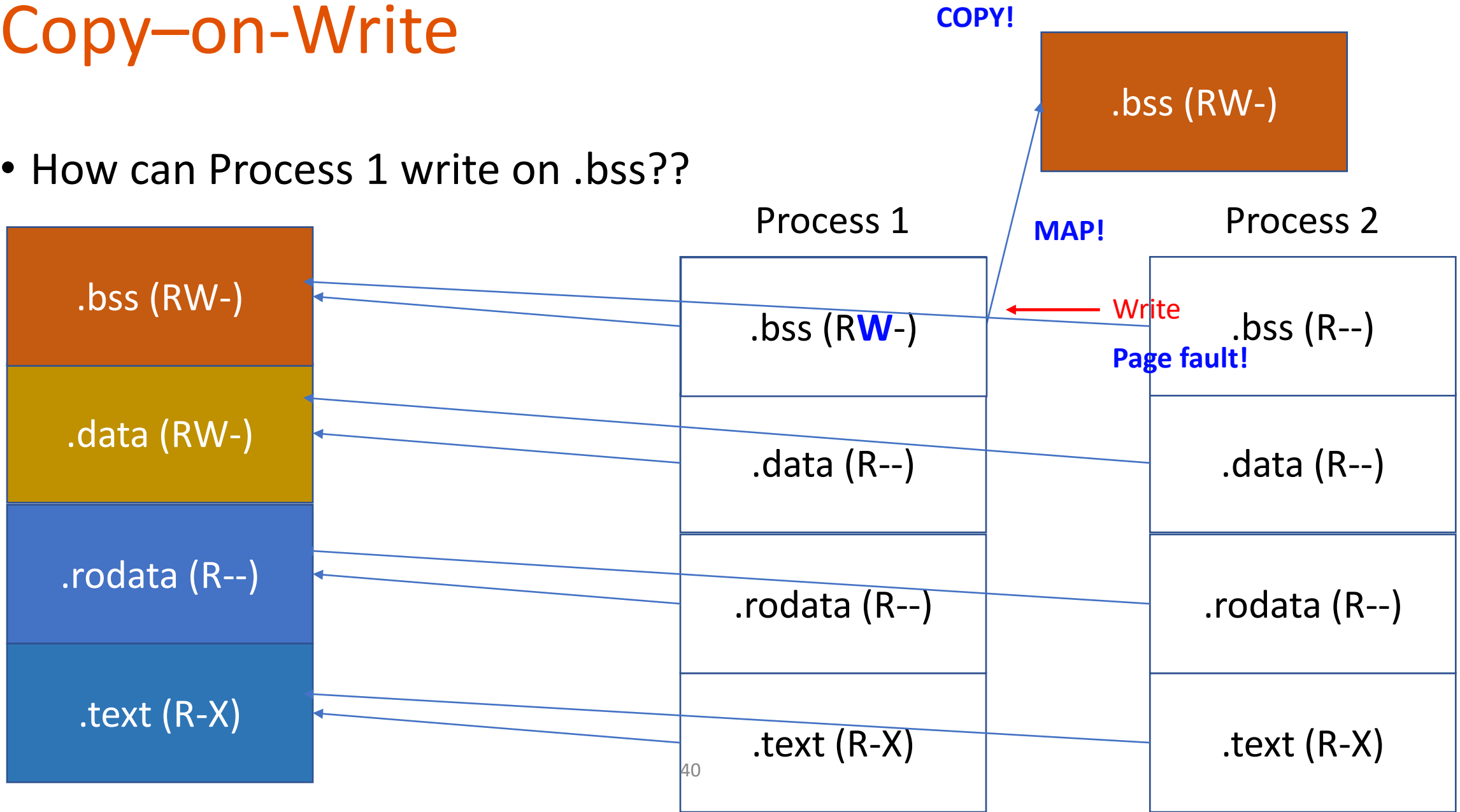


# Page Fault Handler

- Read CR2
  - An address that is in the page cache
    - Hmm... a fault from one of the shared location!
- Read Error code
  - Write on read-only memory
    - Hmm... the process requires a private copy! (we actually mark if COW is required in PTE)
- ToDo: create a writable, private copy for that process!
  - Map a new physical page (page\_alloc, page\_insert)
  - Copy the contents
  - Mark it read/write
  - Resume...

# Copy-on-Write

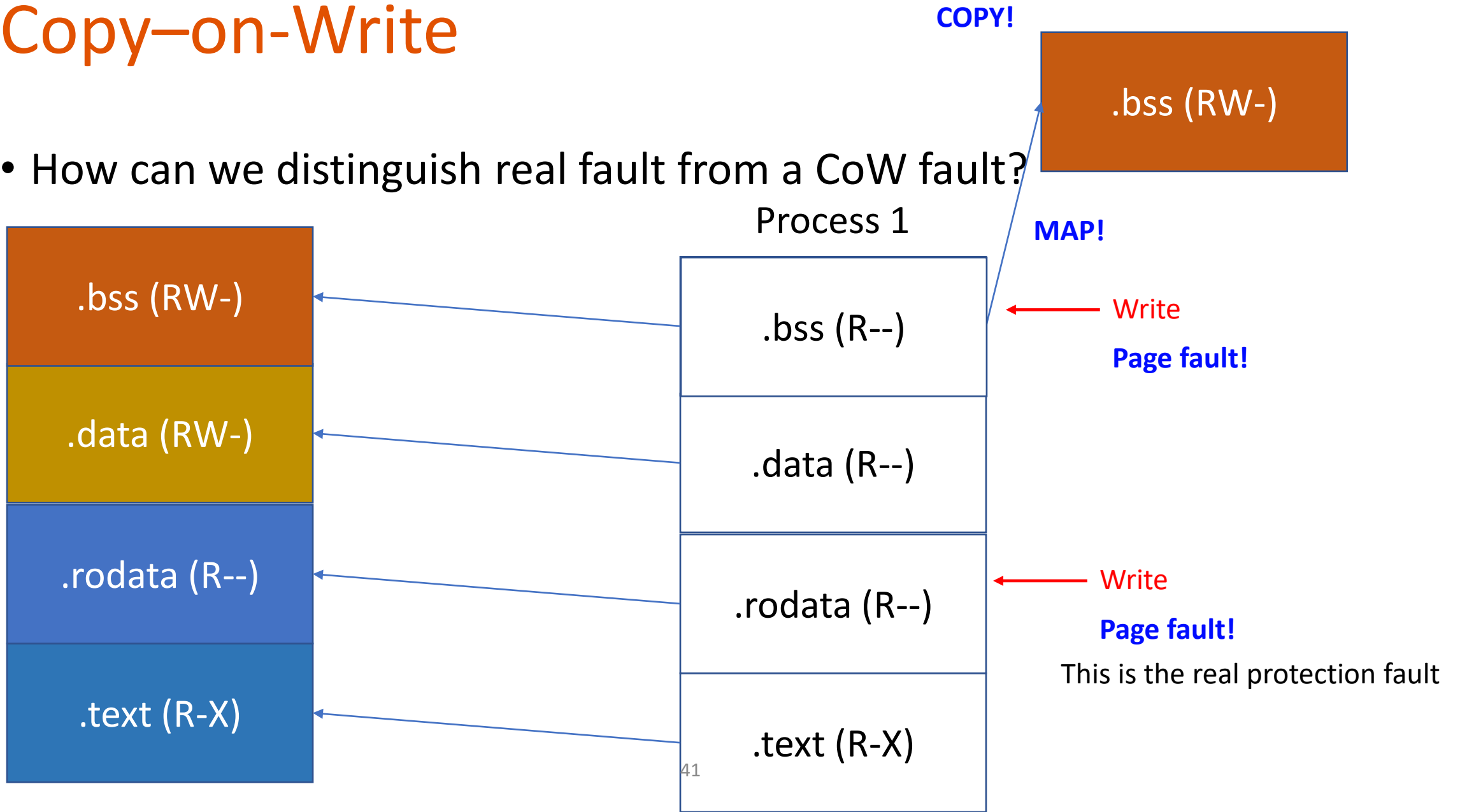
- How can Process 1 write on .bss???





# Copy-on-Write

- How can we distinguish real fault from a CoW fault?



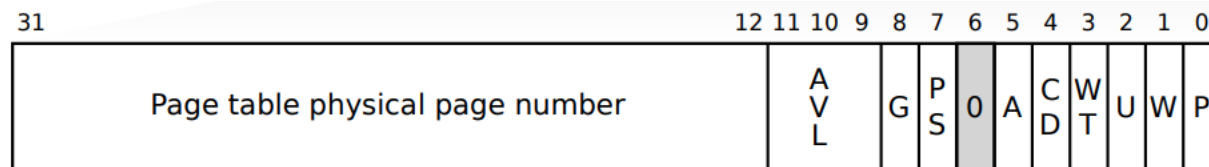
# Use Available Flags in PTE

- PTE\_COW

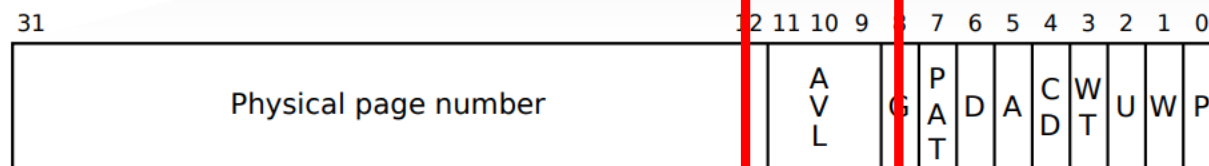
```
// PTE_COW marks copy-on-write page table entries.  
// It is one of the bits explicitly allocated to user processes (PTE_AVAIL).  
#define PTE_COW      0x800
```

- 1000 0000 0000

- 11<sup>th</sup>-bit is 1



PDE

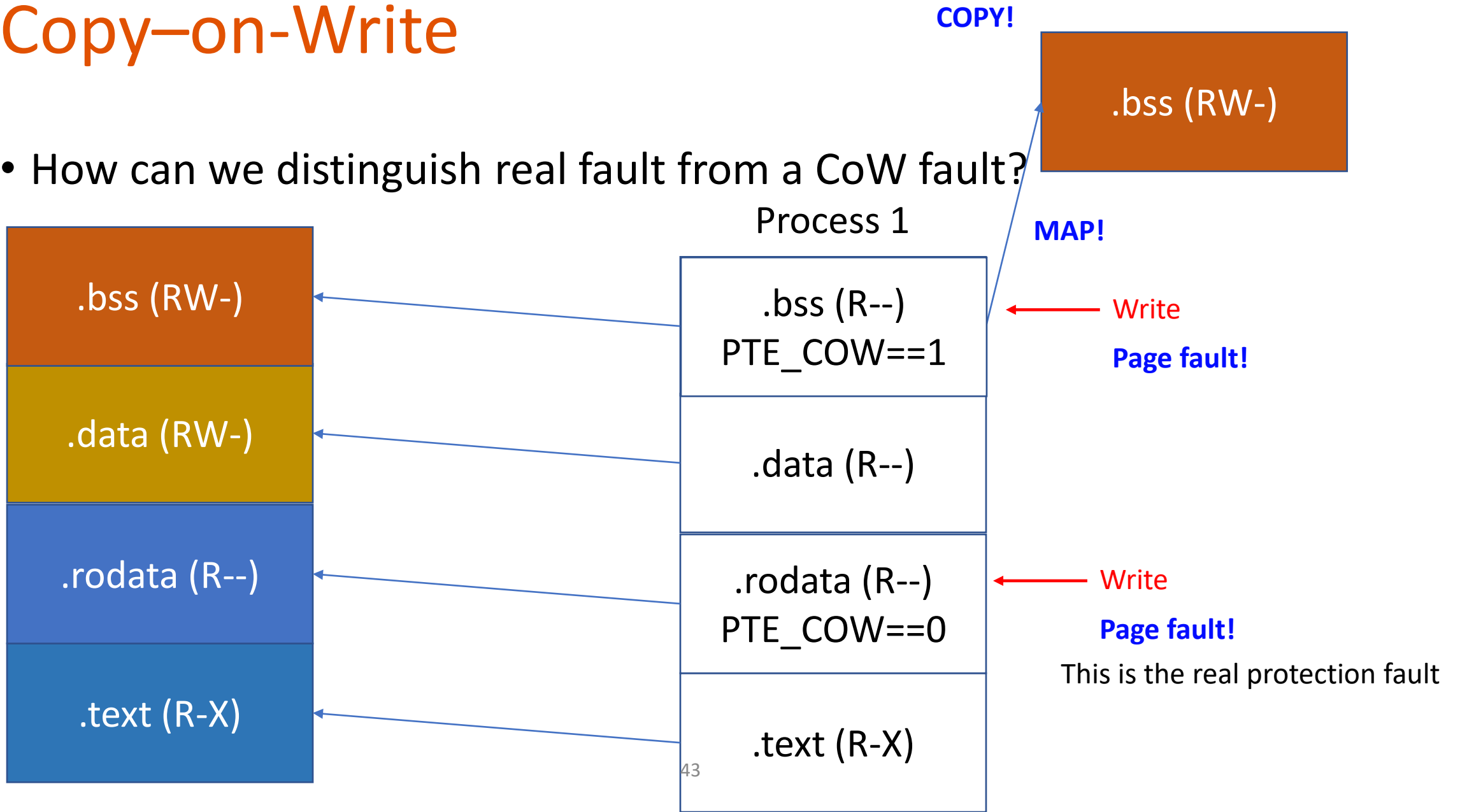


PTE

- P Present
- W Writable
- U User
- WT 1=Write-through, 0=Write-back
- CD Cache disabled
- A Accessed
- D Dirty
- PS Page size (0=4KB, 1=4MB)
- PAT Page table attribute index
- G Global page
- AVL Available for system use

# Copy-on-Write

- How can we distinguish real fault from a CoW fault?



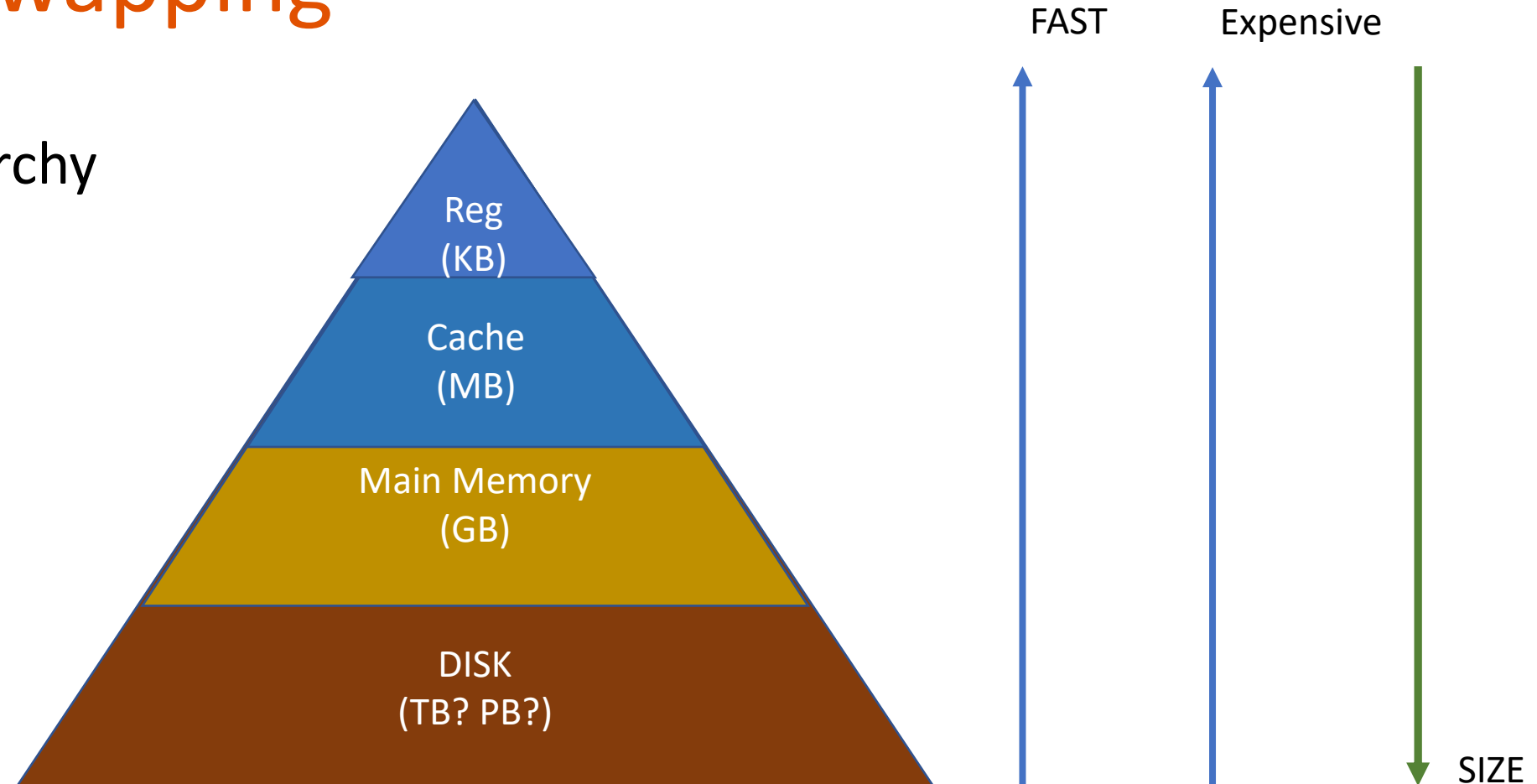
# Benefits?

By exploiting **page fault and its handler**, we can implement **copy-on-write**, a mechanism that can **reduce physical memory usage** by **sharing pages of same contents** among multiple processes.

- Can reduce time for copying contents that is already in some physical memory (page cache)
- Can reduce actual use of physical memory by sharing code/read-only data among multiple processes
  - 1,000,000 processes, requiring only 1 copy of .text/.rodata
- At the same time
  - Can support sharing of writable pages (if not written at all)
  - Can create private pages seamlessly on write

# Memory Swapping

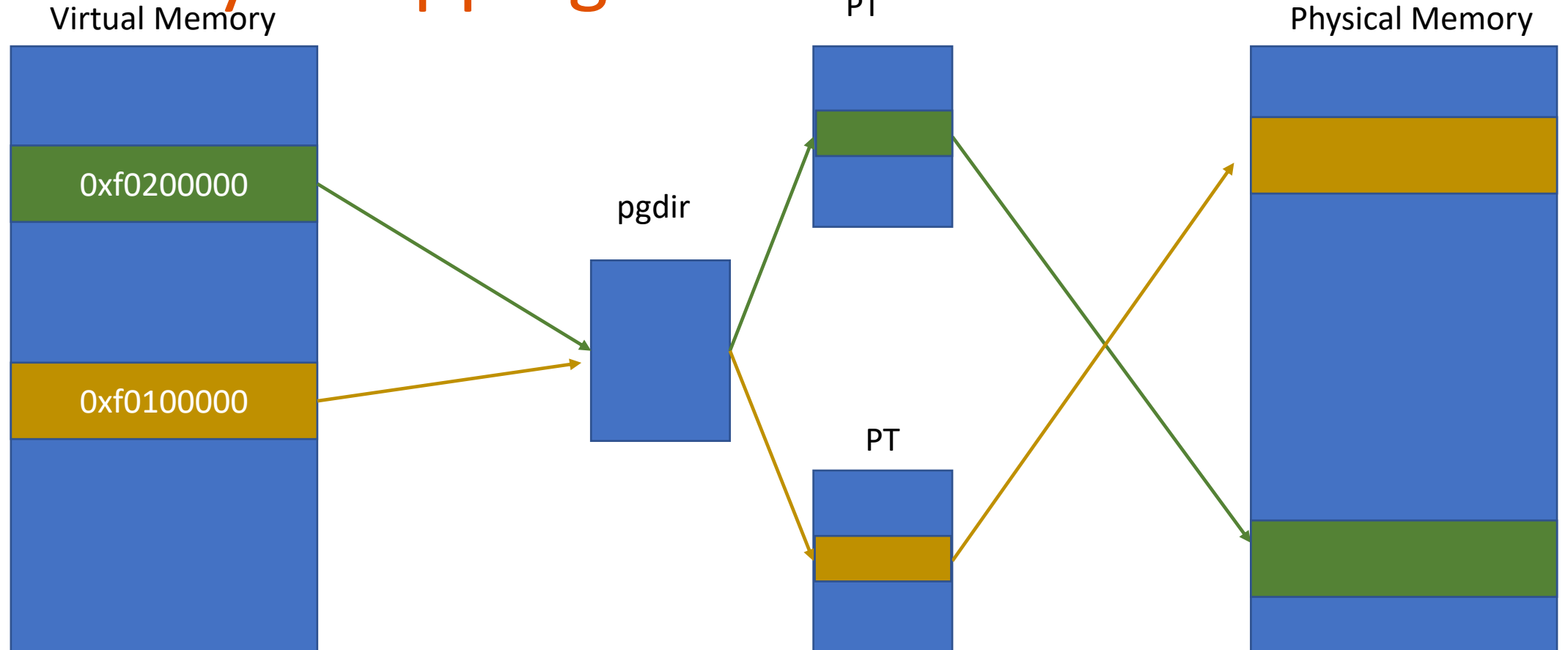
- Memory Hierarchy



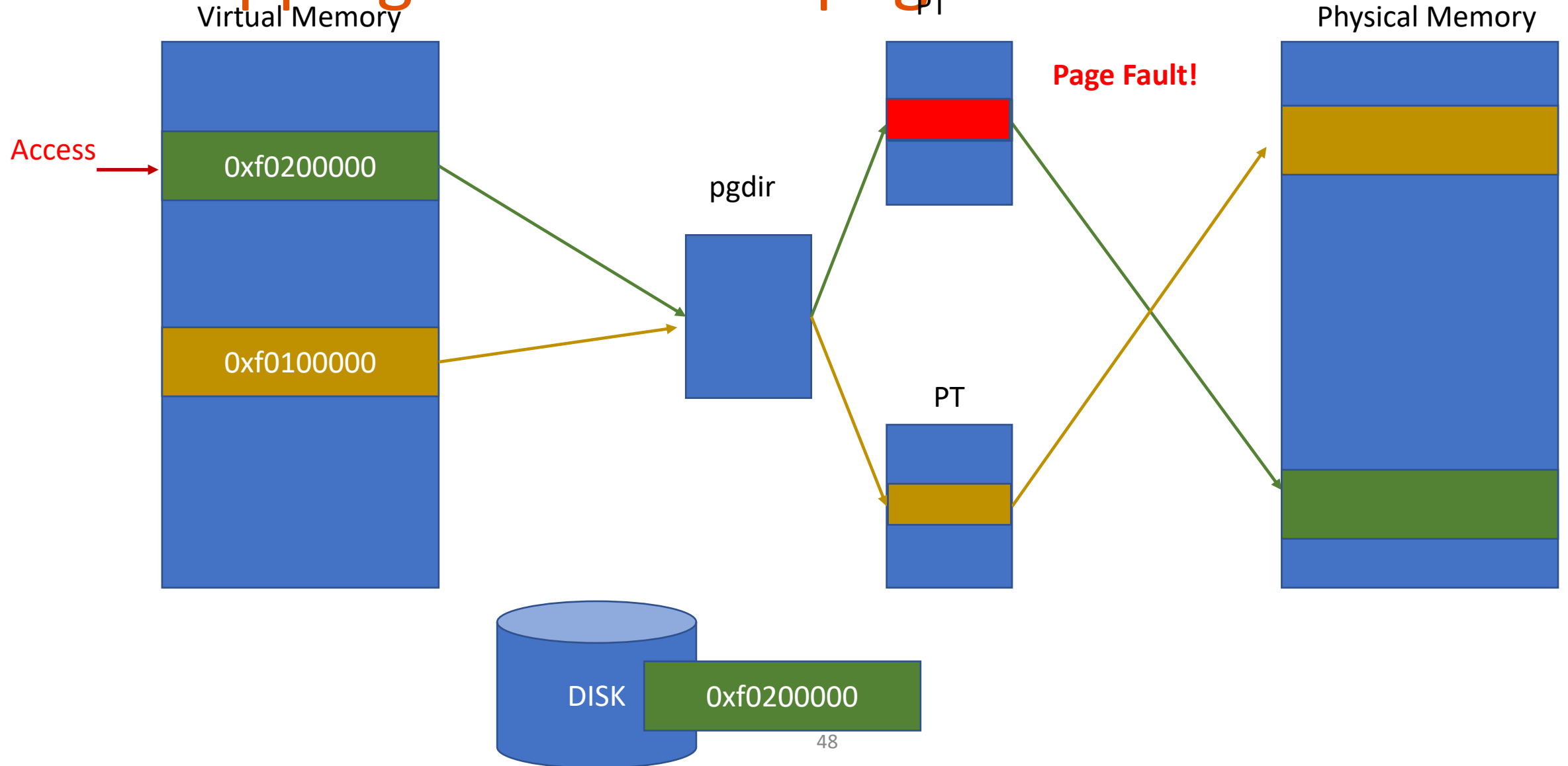
# Challenge

- Suppose you have 8GB of main memory
- Can you run a program that its program size is 16GB?
  - Yes, you can load them part by part
  - This is because we do not use all of data at the same time
- Can your OS do this execution seamlessly to your application?

# Memory Swapping



# Swapping – Remove a page...

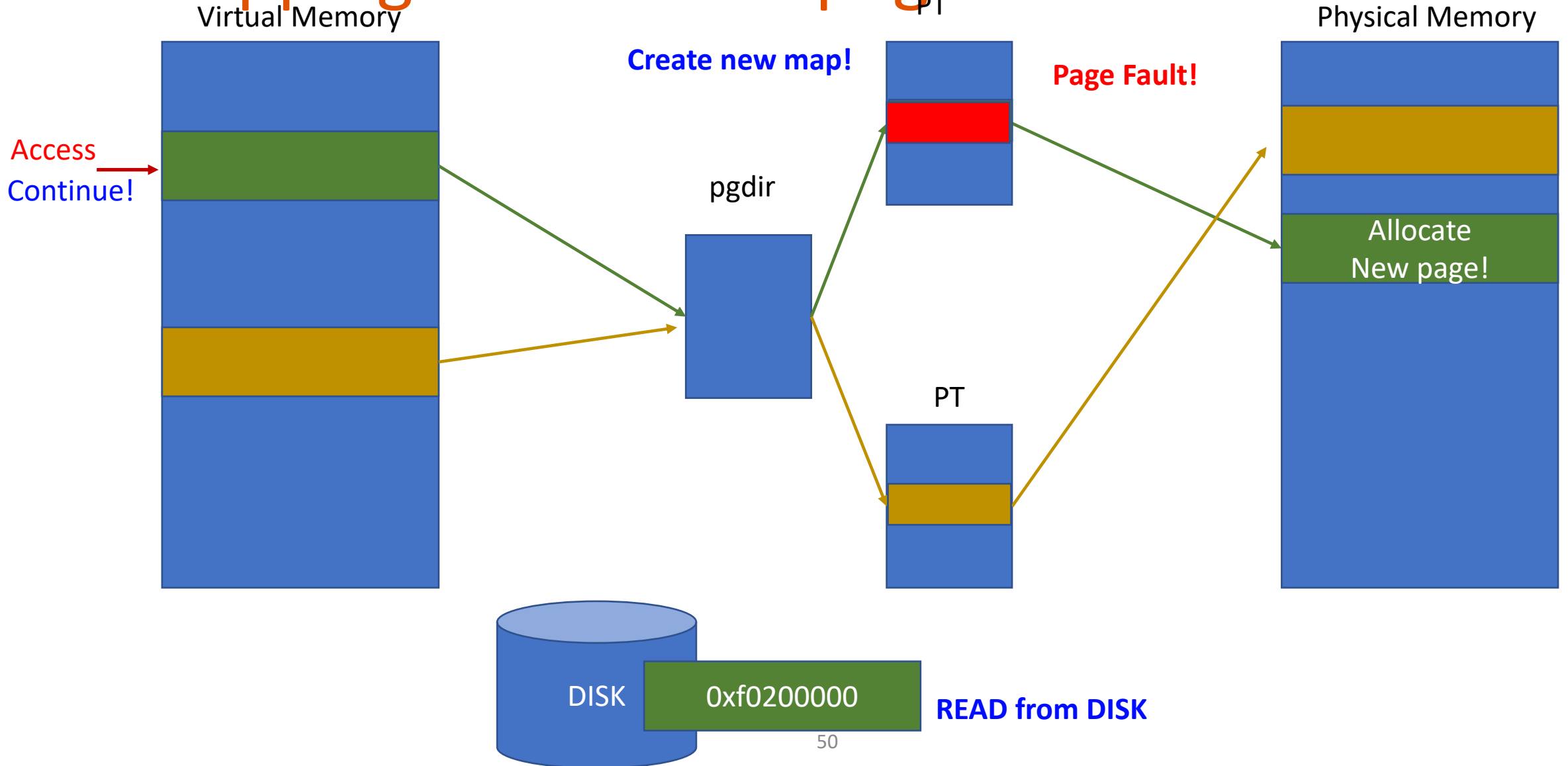




# Swapping - OS

- Page fault handler
  - Read CR2 (get address, `0xf0200000`)
  - Read error code
- If error code says that the fault is caused by non-present page and
- The faulting page of the current process is stored in the disk
  - Lookup disk if it swapped put `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!

# Swapping – Remove a page...



# Page Fault

- Is generated when there is a memory error (regarding paging)
- Is an exception that can be recovered
  - And user program may resume the execution
- Is useful for implementing
  - Automatic stack allocation
  - Copy-on-write (will do in Lab4)
  - Memory Swapping