CS444/544 Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec 9: System Calls and Page Faults

Adapted from content originally created by: Prof. Yeongjin Jang

Recap | System Calls, printf()?

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

`printf``("CS444")`

library call in **ring 3**

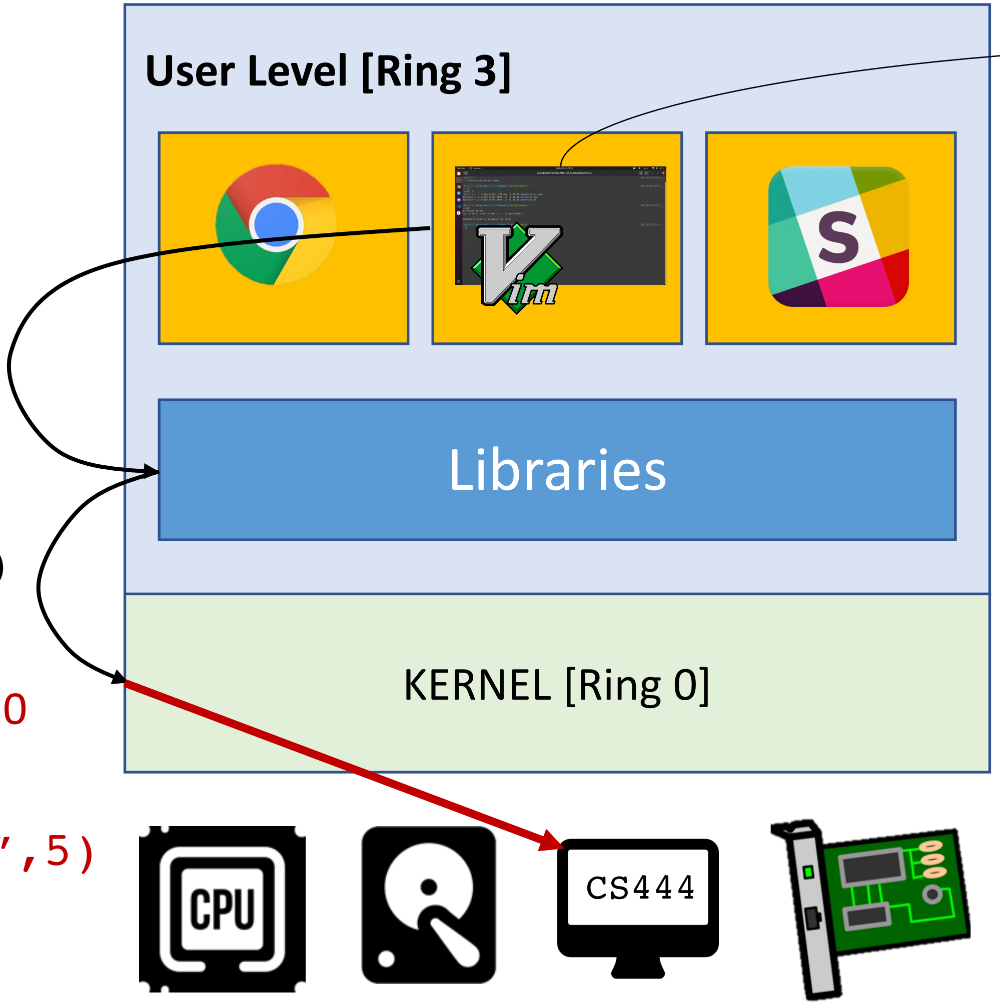
`sys_write``(1,"CS444",5)`

system call from **ring 3**

Interrupt! Switch from ring 3 →0

`do_sys_write``(1,"CS444",5)`

kernel function in **ring 0**



We're not done with printf() though!

`printf("CS444")`

library call in **ring 3**

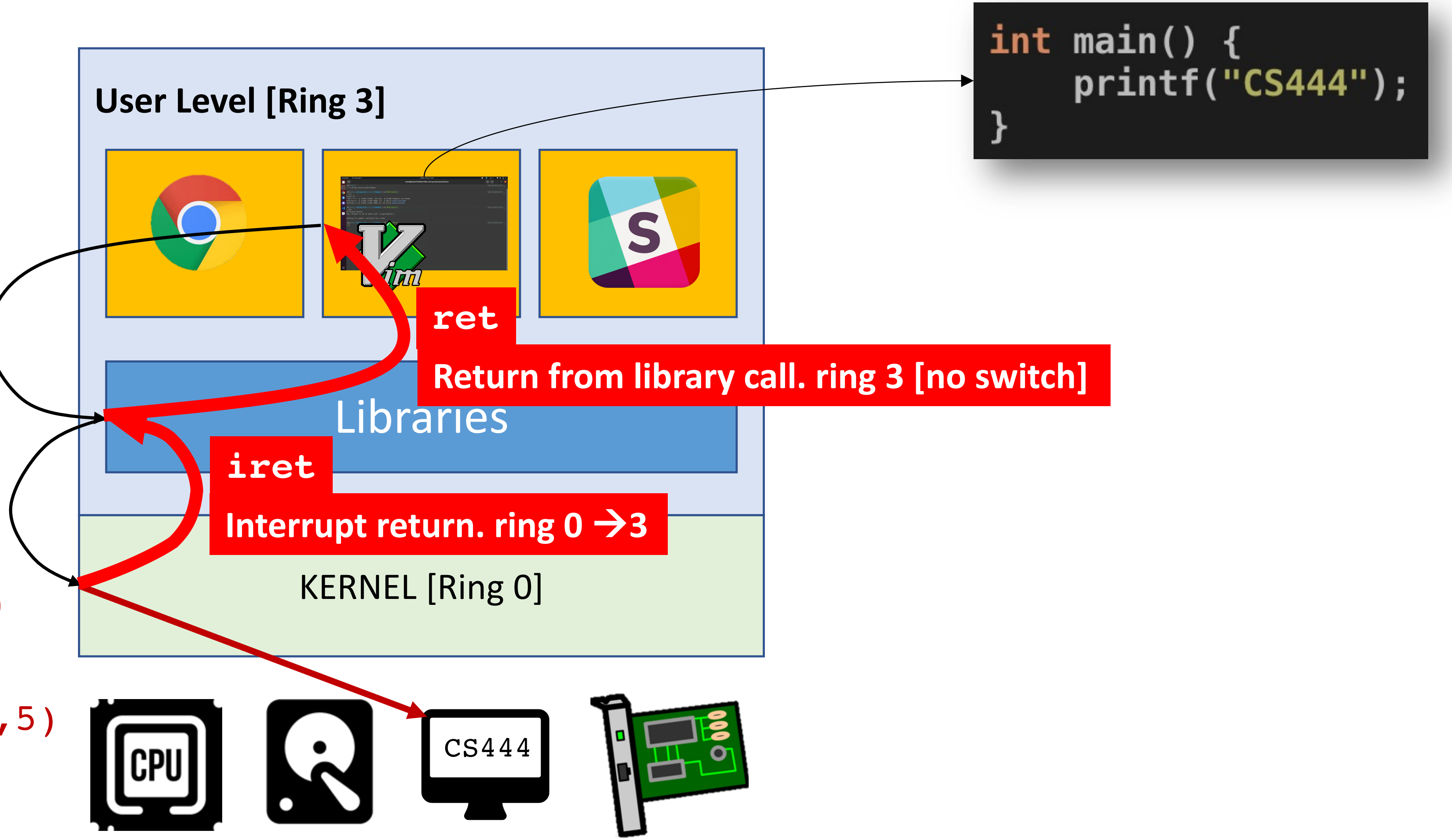
`sys_write(1,"CS444",5)`

system call from **ring 3**

Interrupt! Switch from ring 3 →0

`do_sys_write(1,"CS444",5)`

kernel function in **ring 0**



Today's Topic

More about System Calls

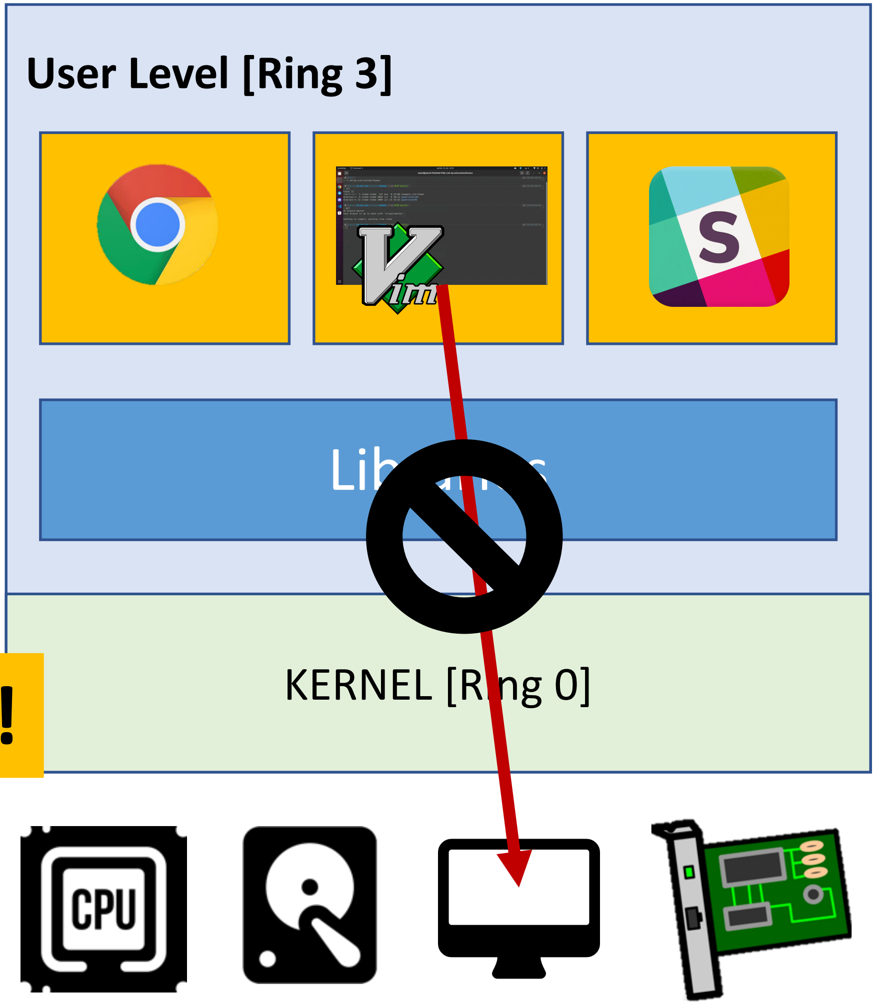
- **Privilege separation** and **call gate**

Page Faults

- OS **fault handling** and **resume execution**?
- For what purpose?
  - Automatic stack allocation
  - Copy-on-write
  - Swap

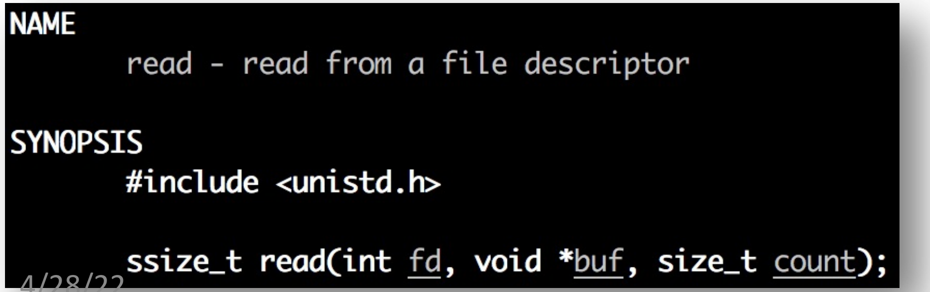
Why have Privilege Separation?

- **Security**!
- We do not know what application will do
  - **Dangerous operations**
    - Flash BIOS, format disk, deleting system files, etc.
  - Let only OS access hardware
    - Apply **access control** for hardware resources!
    - E.g., only the administrator can format disk
- OS mediates hardware access **System Calls!**



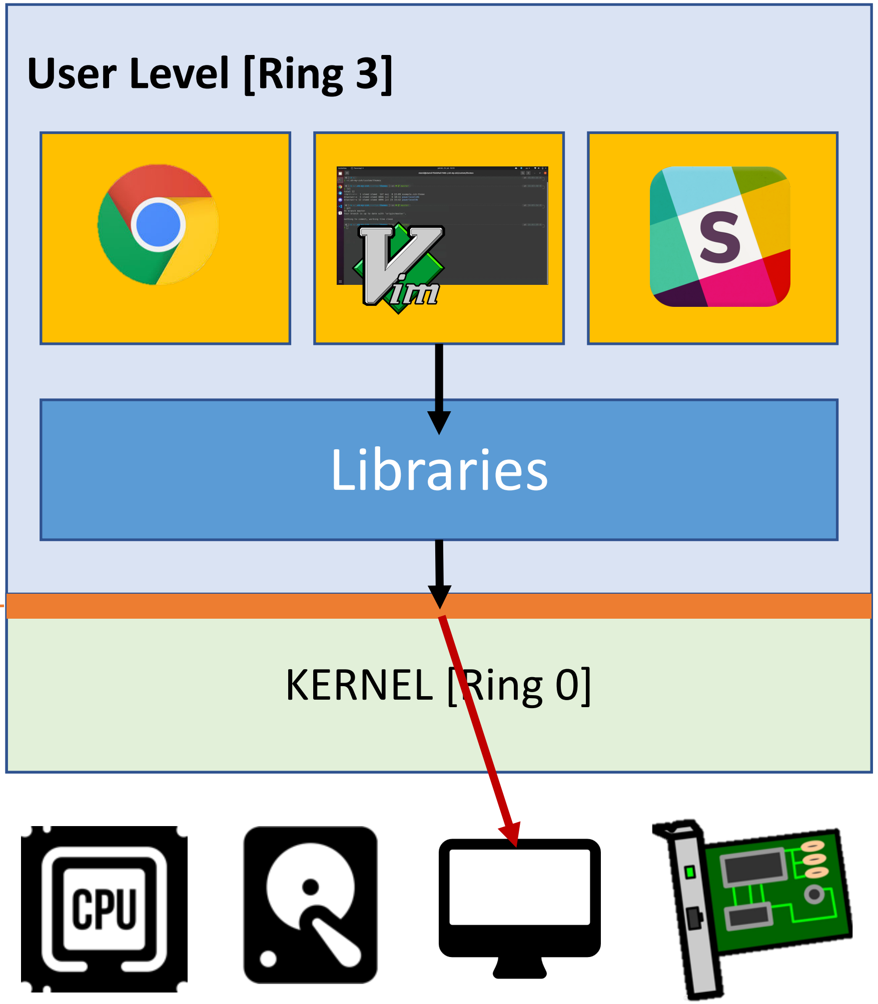
Library vs System Calls

- **Library Calls**
  - APIs available in Ring 3
  - **Do not** include operations in Ring 0
  - **Cannot access hardware directly**
- Could be a **wrapper** for
  - some computation or
  - for system calls
  - E.g., `printf()` internally uses `write()`



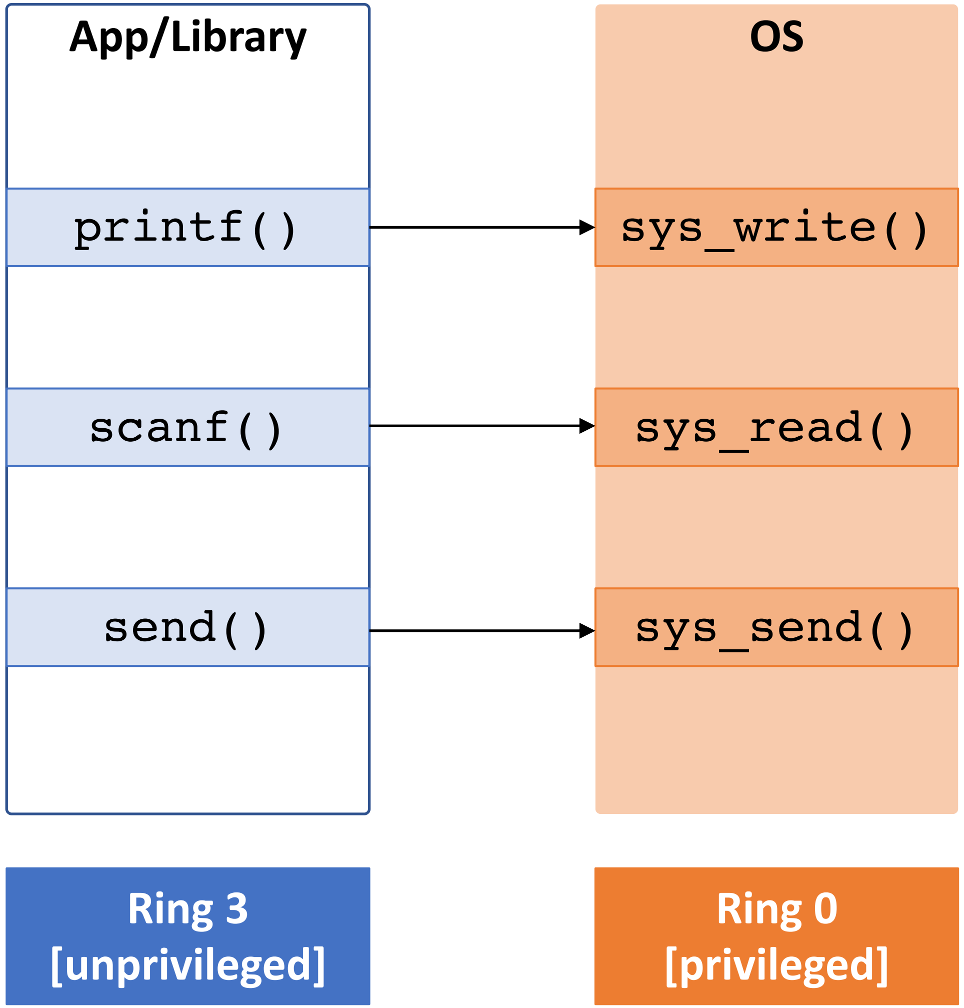
```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

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

system calls



Library Calls → System Calls

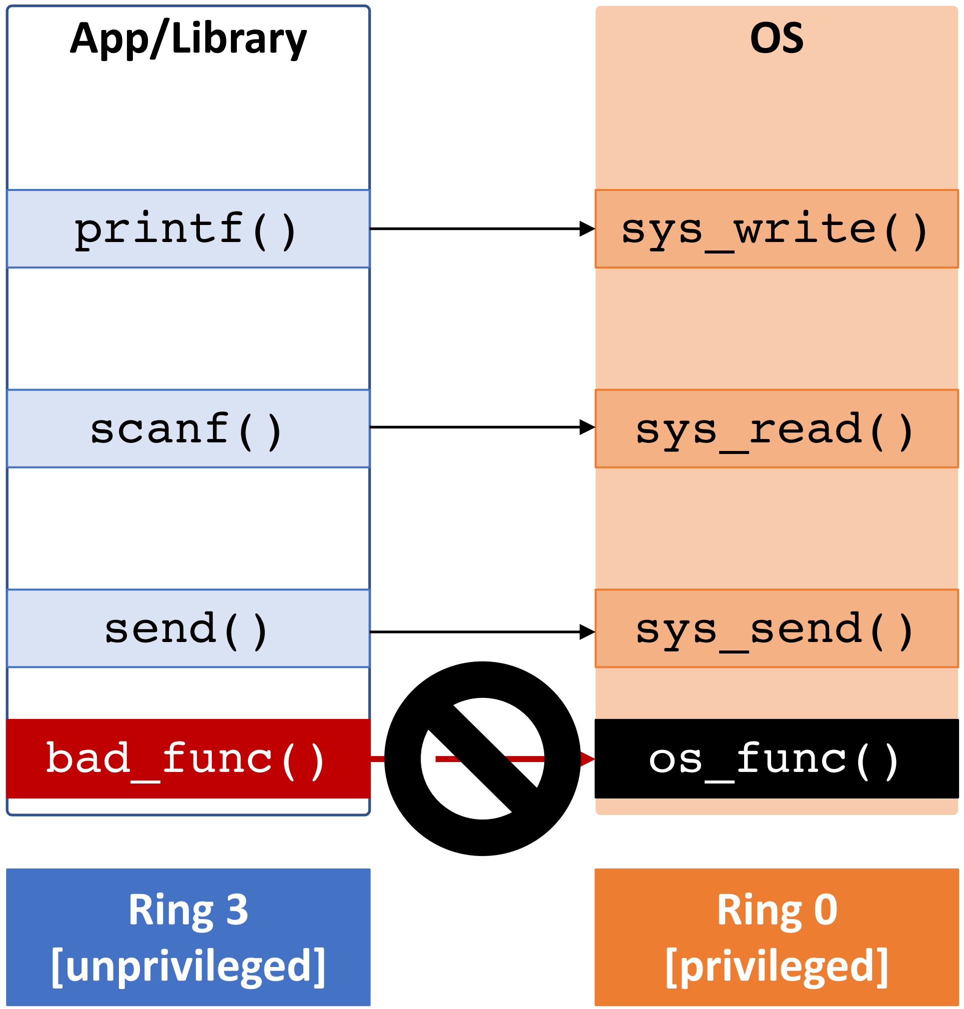
- **System Calls**
  - APIs available in **Ring 0**
  - OS **abstraction** for **hardware interface**
  - Ring 3 application → Ring 0 operations



Library Calls → System Calls

- App **shouldn't call arbitrary function**!
- Else privilege separation meaningless

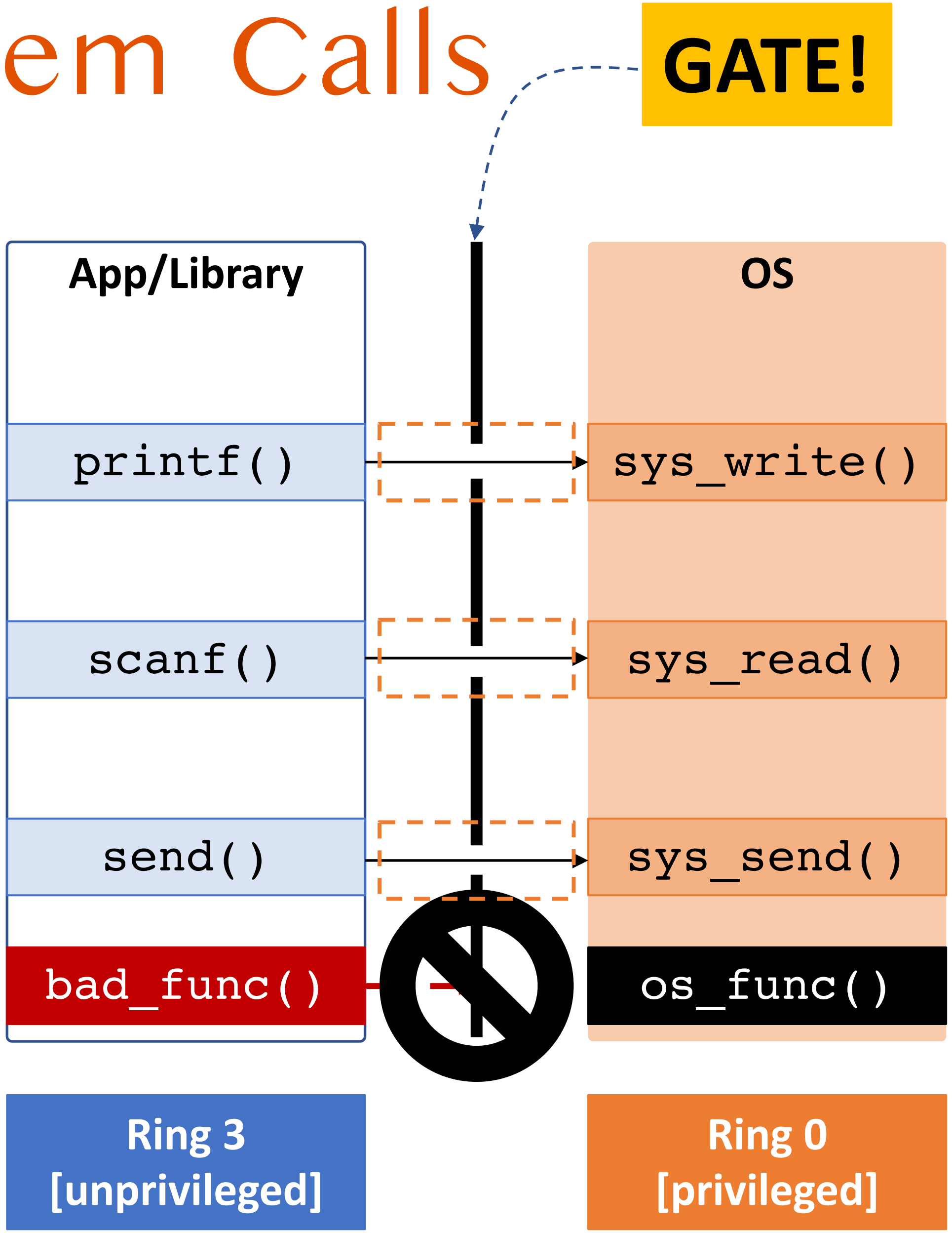
- Apps/libraries can invoke system calls
- **But no other kernel functions!**



Library Calls → System Calls

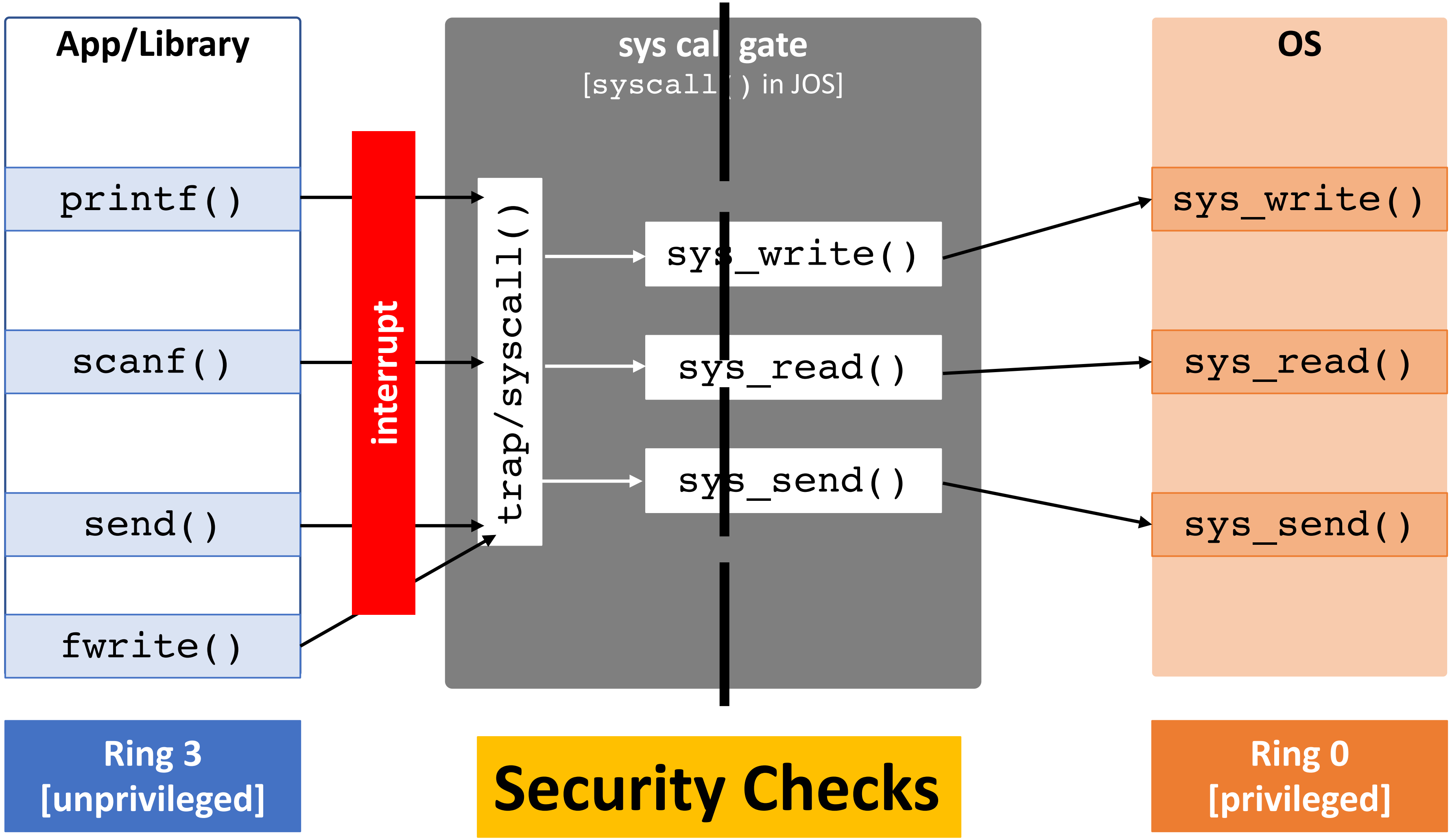
- App **shouldn't call arbitrary function**!
- Else privilege separation meaningless

- Apps/libraries can invoke system calls
- **But no other kernel functions!**



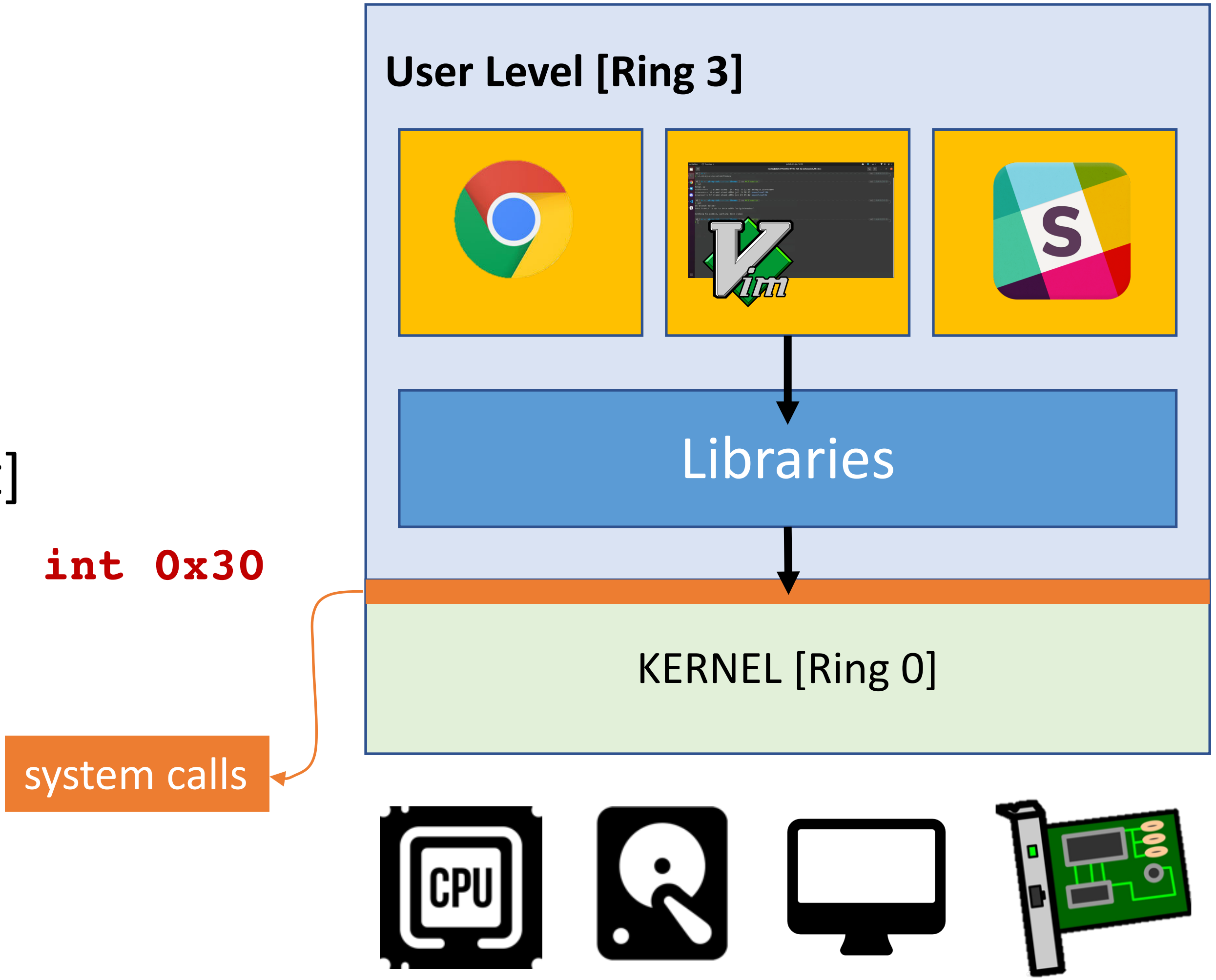
Secure System Call Design: Call Gate via Interrupt Handling

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



Call Gates, syscall handling

- Call Gate
- System call invoked
  - only with trap handler
  - `int $0x30` – in JOS
  - `int $0x80` – in Linux [32-bit]
  - `int $0x2e` – in Windows [32-bit]



Call Gates, syscall handling

- Call Gate
- System call invoked
  - only with trap handler
  - `int $0x30` – in JOS
  - `int $0x80` – in Linux [32-bit]
  - `int $0x2e` – in Windows [32-bit]

program execution → interrupt → call trap gate → create/store trap frame → load trap handler in mem → execute

Pop context → iret → resume execution

Overheads!
10s of thousands of cycles!

Call Gates, syscall handling

- Call Gate
- System call 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)
10x faster than ints

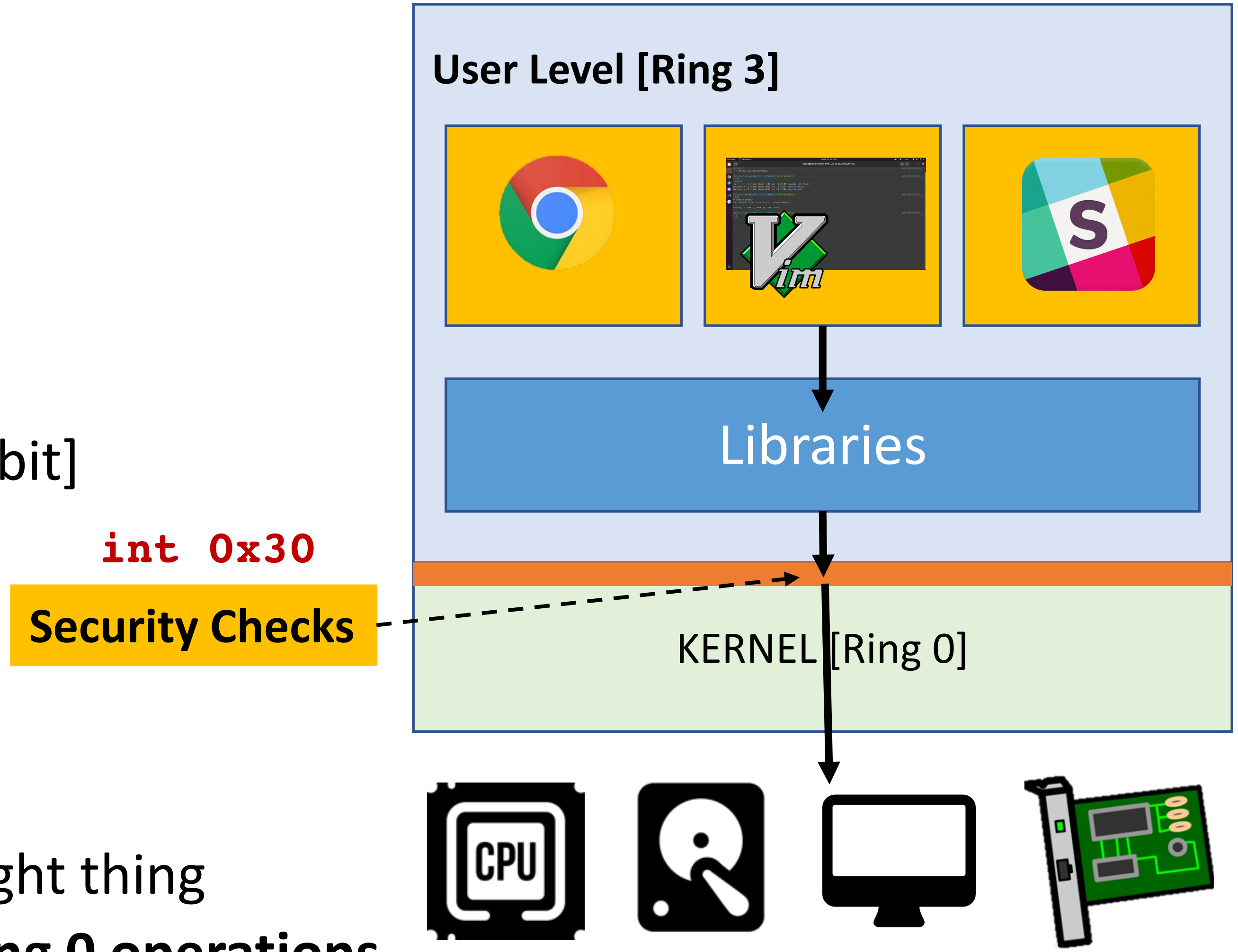
program execution → interrupt → call trap gate → create/store trap frame → load trap handler in mem → execute

Pop context → iret → resume execution

Overheads!
10s of thousands of cycles!

Call Gates, syscall handling

- Call Gate
- System call 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 userspace app/lib is doing a right thing
  - **Before performing important ring 0 operations**



Protecting Syscalls via Call Gate

- Consider the '`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);
```

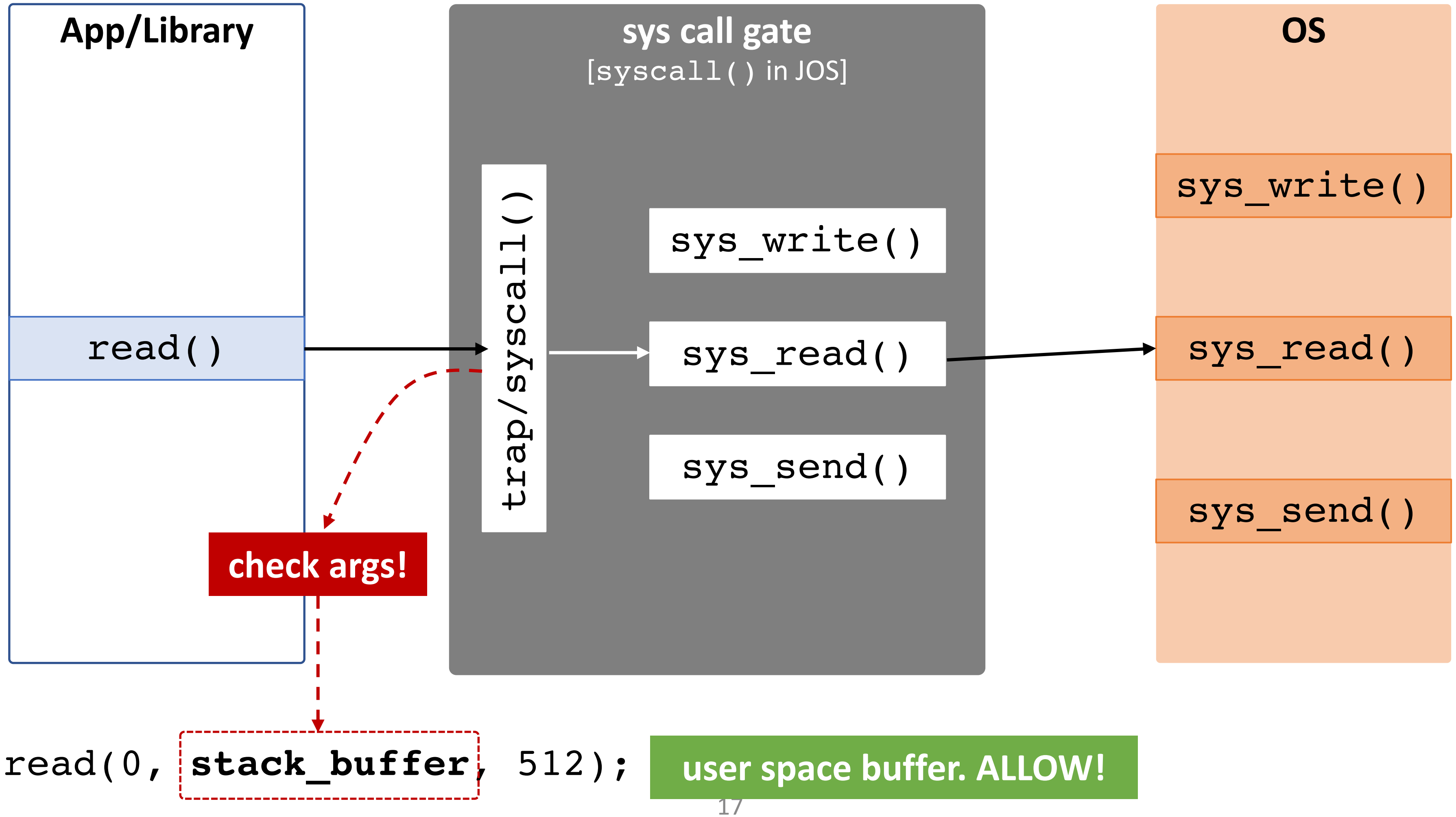
Protecting Syscalls via Call Gate

```
// 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 will **overwrite kernel code** with your keyboard inputs!!!
  - Changing kernel code from Ring 3 is possible!

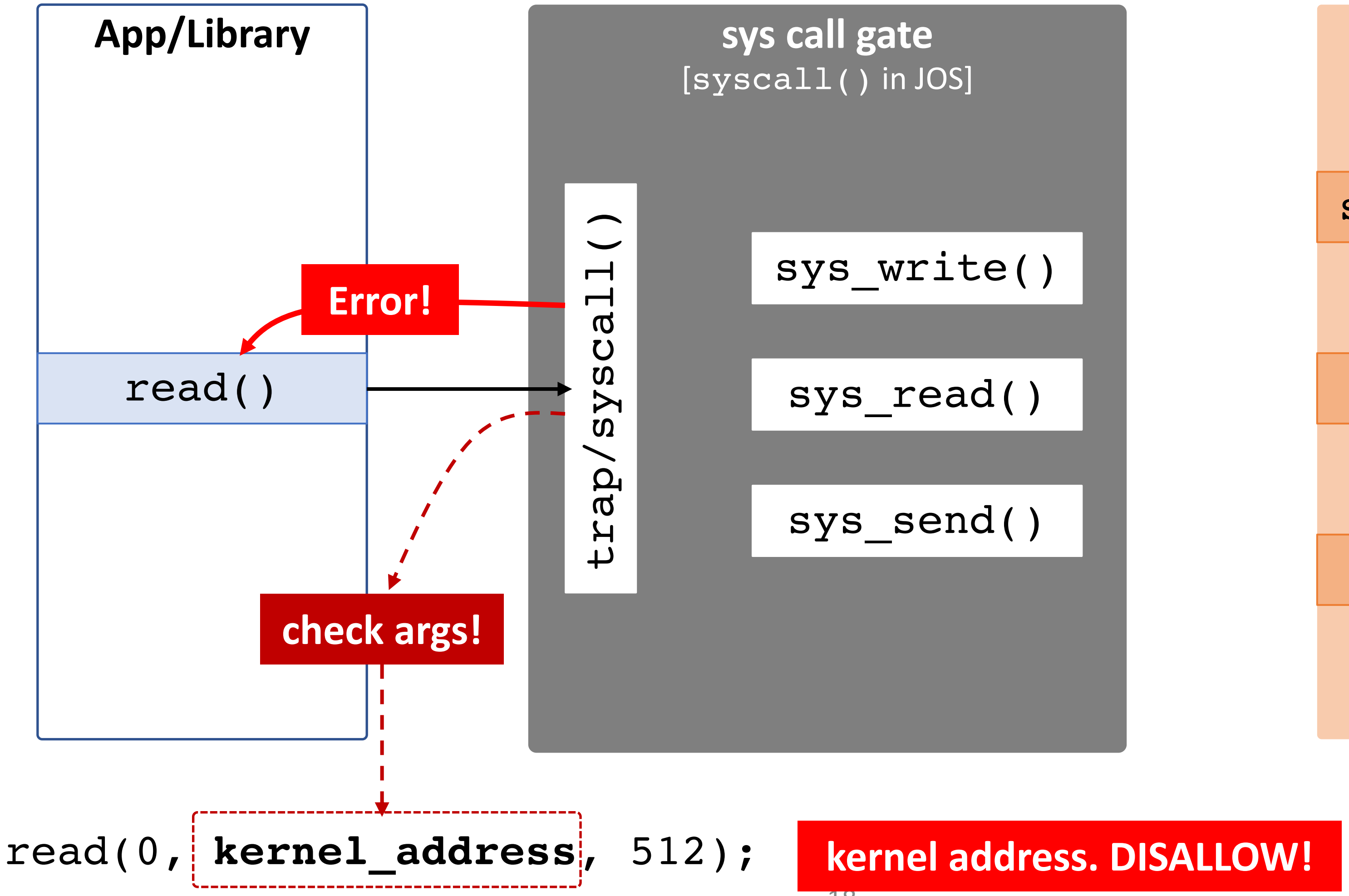
Use the Call Gate!

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



Use the Call Gate!

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



Test

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

printf("Read to stack memory returns: %d\n", ret);

// read 512 bytes from standard input to kernel memory
ret = read(0, (void *) 0xffffffff01000000, 512);

printf("Read to kernel memory returns: %d\n", ret);
perror("Reason for the error:");
```

```
[blue9057@blue9057-vm-jos ~/test$] ./a
asdfzxcv
Read to stack memory returns: 9
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**

- System calls → **APIs of an OS**

- Prevent application from running arbitrary ring 0 operation?
  - Call gate
- Modern OSes use call gates to protect system calls
  - trap handler → OS **applies access control** for system call

Page Faults

Handling Faults | Page Fault

- Faulting instruction has not executed [e.g., page fault]
- **Resume 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

```c
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
```

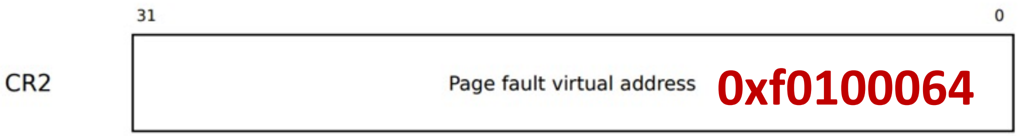
```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xeffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xf0100064
  err  0x00000007 [user, write, protection]
  eip  0x00800039
  cs   0x----001b
  flag 0x00000096
  esp  0xeebfdfb8
  ss   0x----0023
[00001000] free env 00001000
```

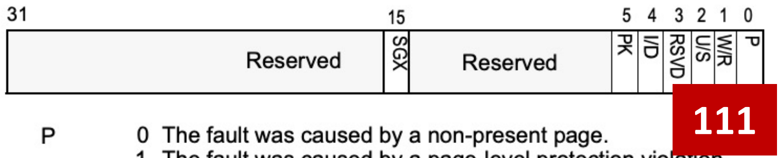
Page Fault: What Does CPU Do?

- CPU informs OS → **why** and **where** 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  0xeebfdfd0
  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  0xeebfdfb8
  ss   0x----0023
[00001000] free env 00001000
```

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





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)`
  - Stores the faulting address in **CR2**
  - Put out an error code
  - **Calls page fault handler in IDT**

- OS: calls **page_fault_handler**
  - Read CR2 [address of the fault, `0xf0100064`]
  - Read error code [reason of the fault]
  - Resolve error [if not possible, **destroy the environment**]
  - Continue user execution
- User: **resume** instruction in CR2 [or destroyed by the OS]

Fault Resume Example: Stack Overflow

- `inc/memlayout.h`
- We (initially) **allocate one [1] page [4 kb] for the user stack**

```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW  PGSIZE
                 +------------------------------+ 0xeebfd000
                 |         NOT MAPPED!          |
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- If you use a large local variable on the stack
- **Stack overflow!** **Page Fault!**

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

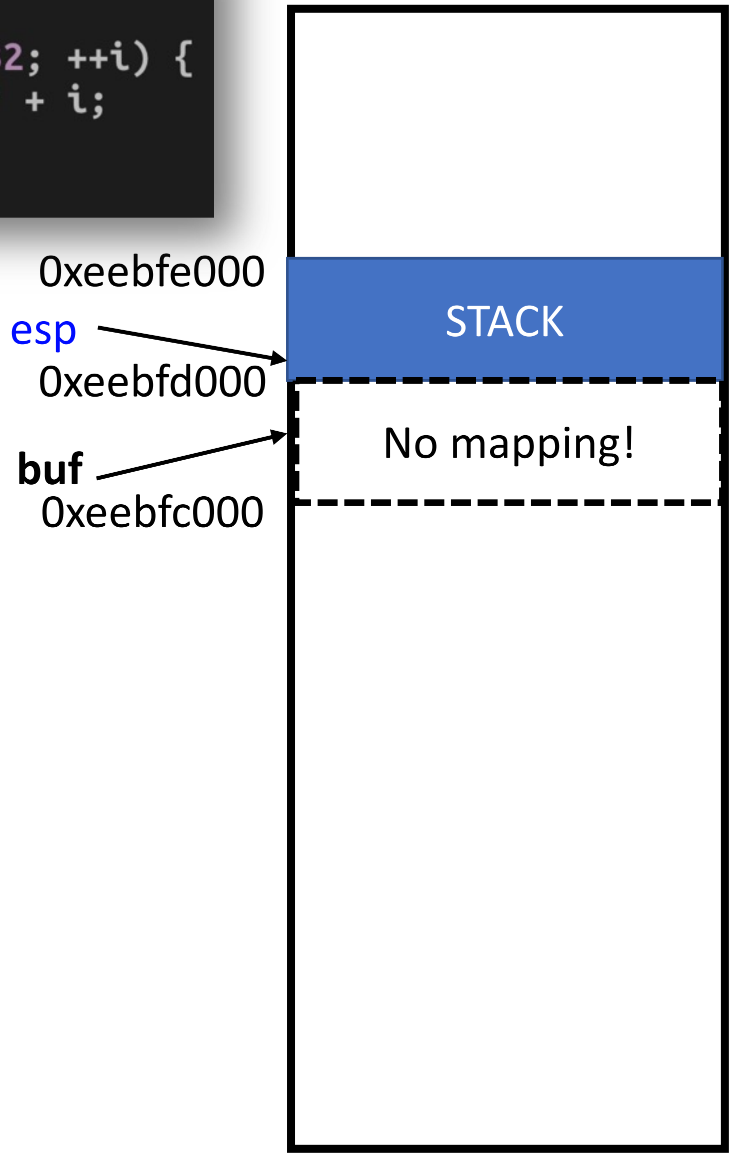
Allocating New Stack Automatically

- **Detect** such an access?
- Allocate a **new page** for the stack **automatically**?
- **Yes!**
- **‘Page Fault’**
- Observations
  - Stack overflow is sequential → access pages adjacent to stack
  - We should catch both read/write access → both should fault

Example: New Stack Allocation by Fault (User)

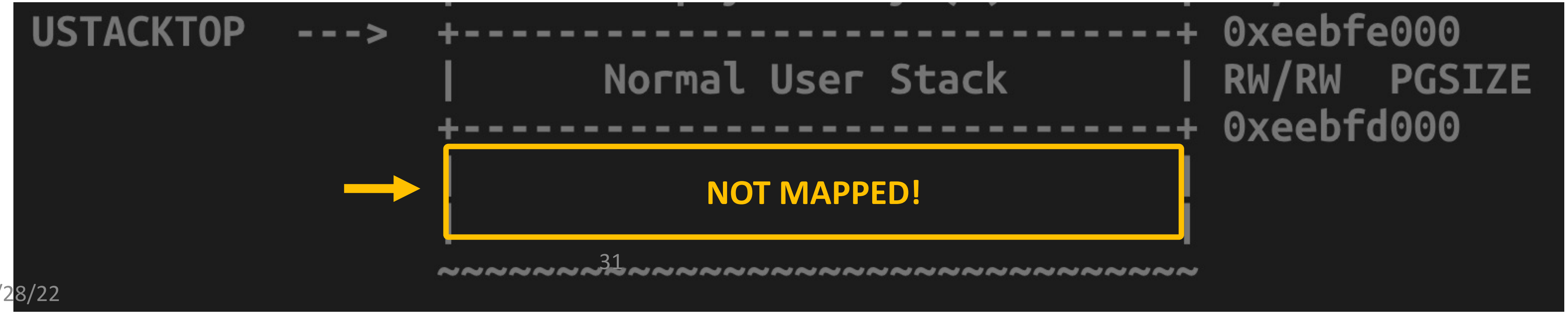
```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

- 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`**
  - **Must allocate `0xeebfc000 ~ 0xeebfd000`**



Example: New Stack Allocation by Fault (CPU)

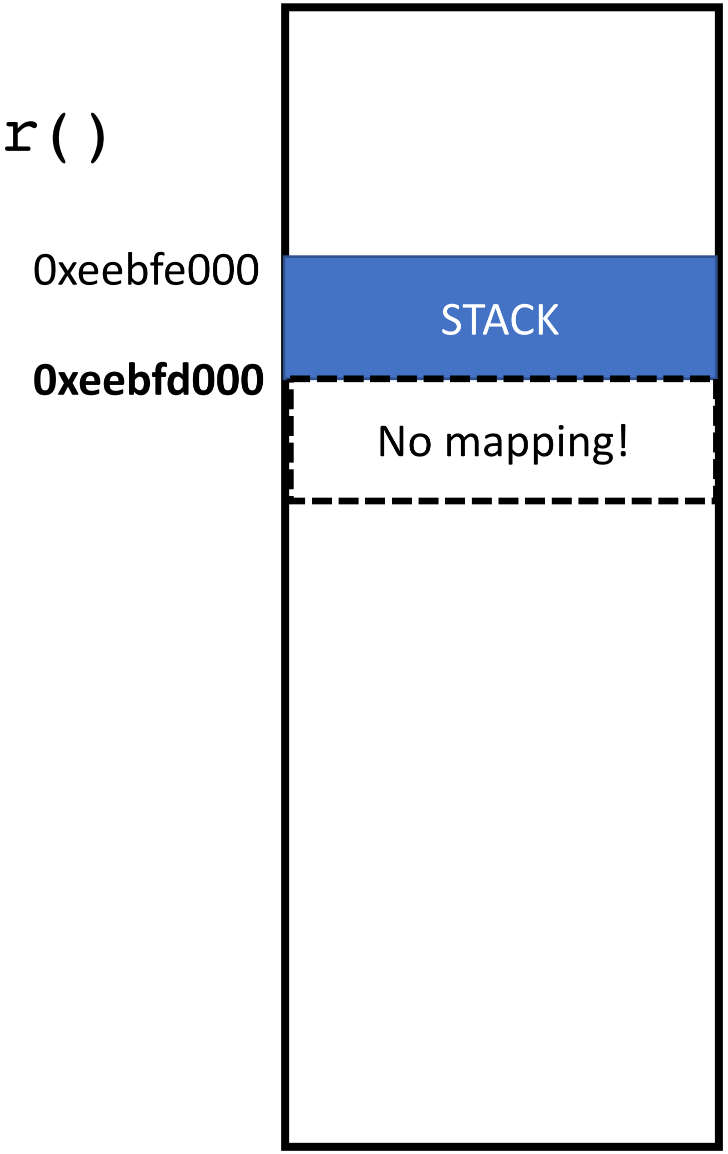
- Lookup page table → **no translation!**
- Store `0xeebfcff0` in the CR2 register
- Set **error code**
  - *"The fault was caused by a page that wasn't present!"*
- Raise page fault exception [**interrupt #14**] → call **page fault handler**



Example: New Stack Allocation by Fault (OS)

- Interrupt will force CPU to invoke the `page_fault_handler()`
- Read CR2
  - `0xeebfcff0`, the page right next to current stack end
  - The current stack end is: **0xeebfd000**
- Read error code
  - *"The fault was caused by a page that wasn't present!"*

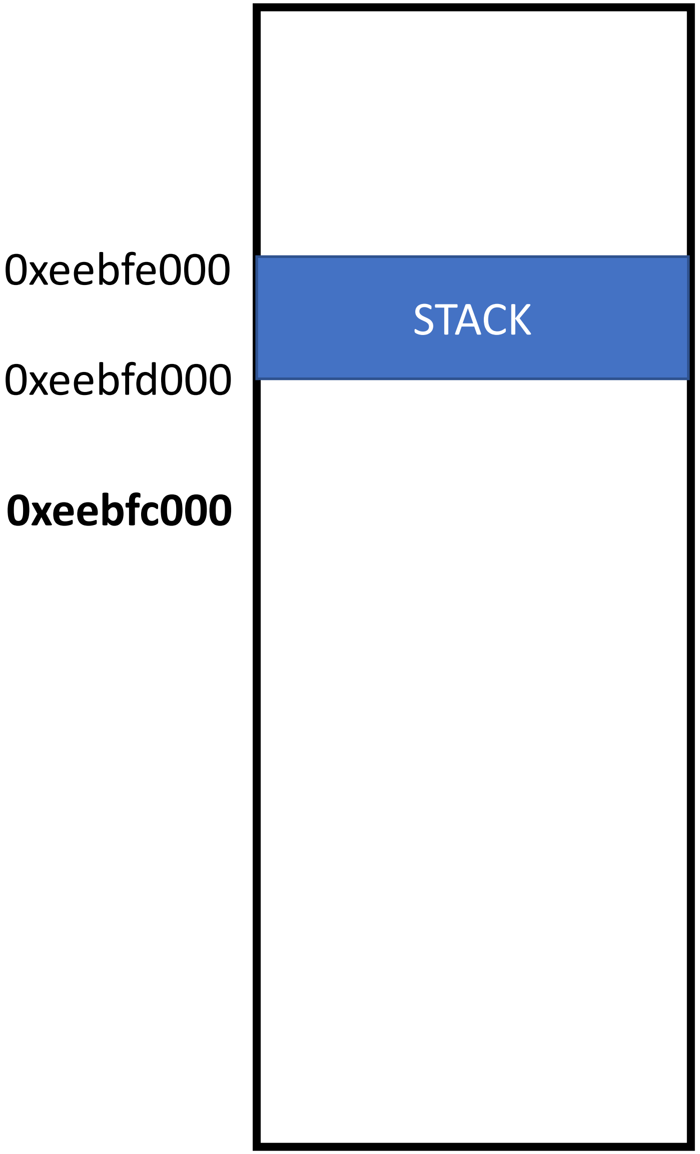
- Let's **allocate a new page for the stack**!



Example: New Stack Allocation using Fault

- Allocate a new page for the stack
- `struct PageInfo *pp = page_alloc(ALLOC_ZERO);`
  - Get a new page, and wipe it to zero all its contents
- `page_insert(env_pgdir, pp, `**`0xeebfc000`**`, PTE_U|PTE_W);`
  - Map a new page to that address!
- `iret!`

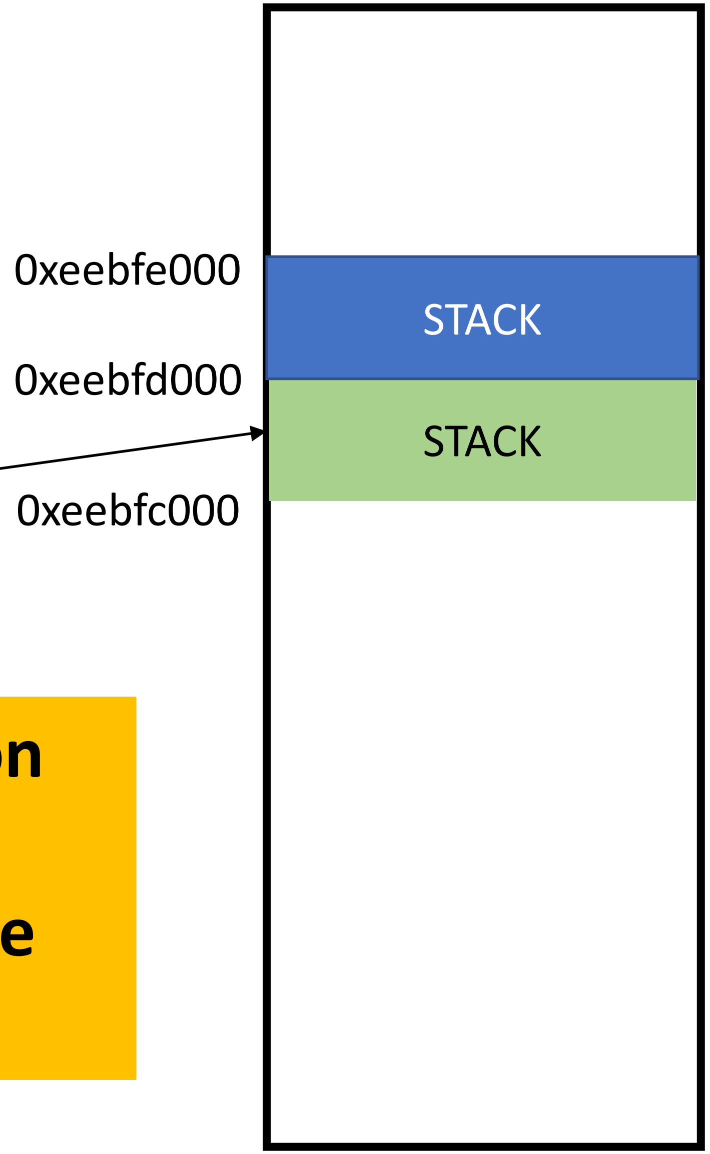
NEW STACK PAGE



New Stack Allocation Using Fault (User-Return)

- **Original** access: `buf[0] = '1';`
  - `movb $0x31, (%eax)`
  - `eax =` **`0xeebfcff0`** **[No translation for `0xeebfc000`]**
- **Execute instruction after page fault handled**: `buf[0] = '1';`
  - `movb $0x31, (%eax)`
  - `eax =` **`0xeebfcff0`** **Translation is Valid!**
- Continue to execute rest of the loop

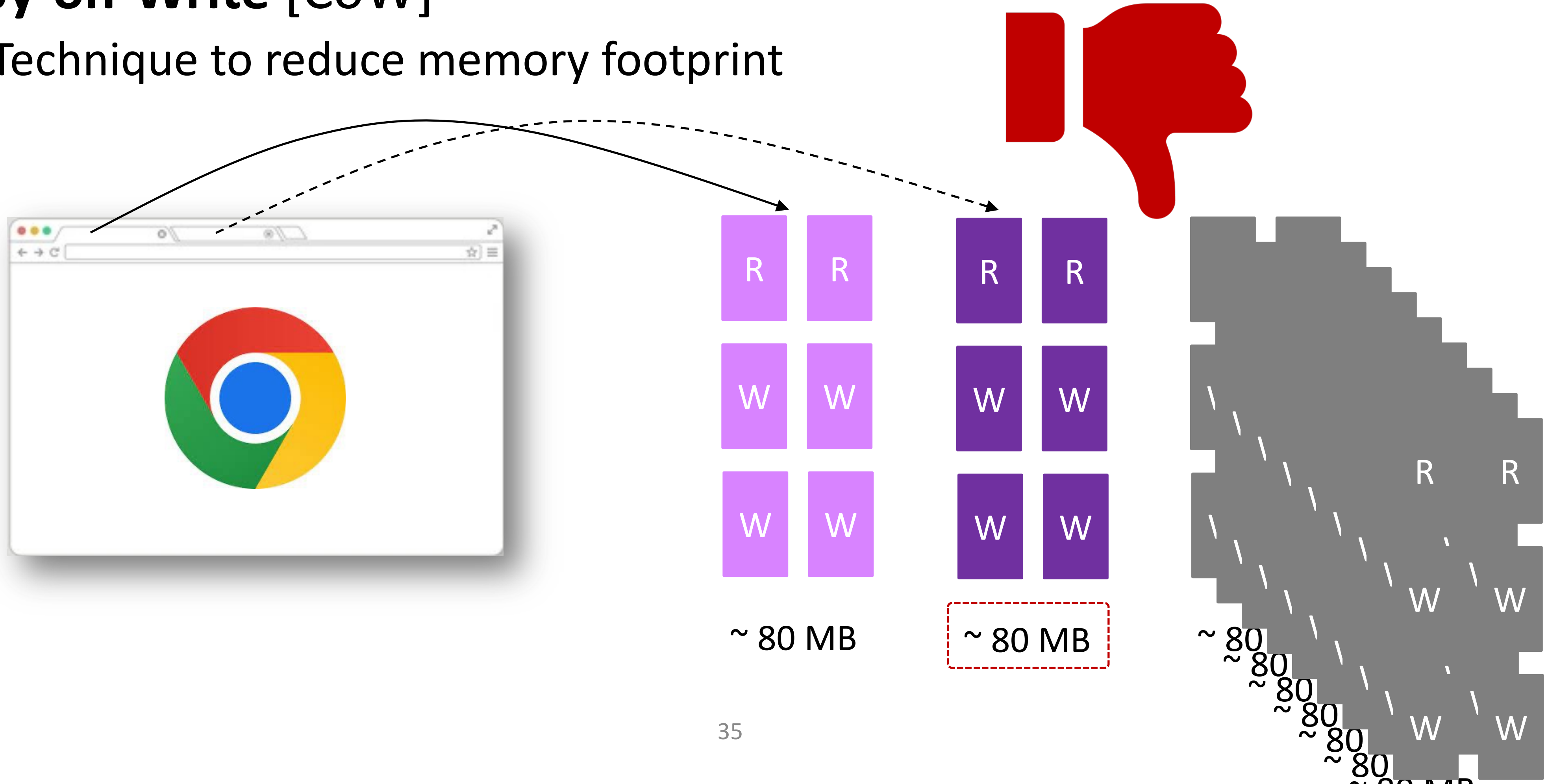
```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```



Automatic allocation of user stack using **page faults** and **page fault handlers**!

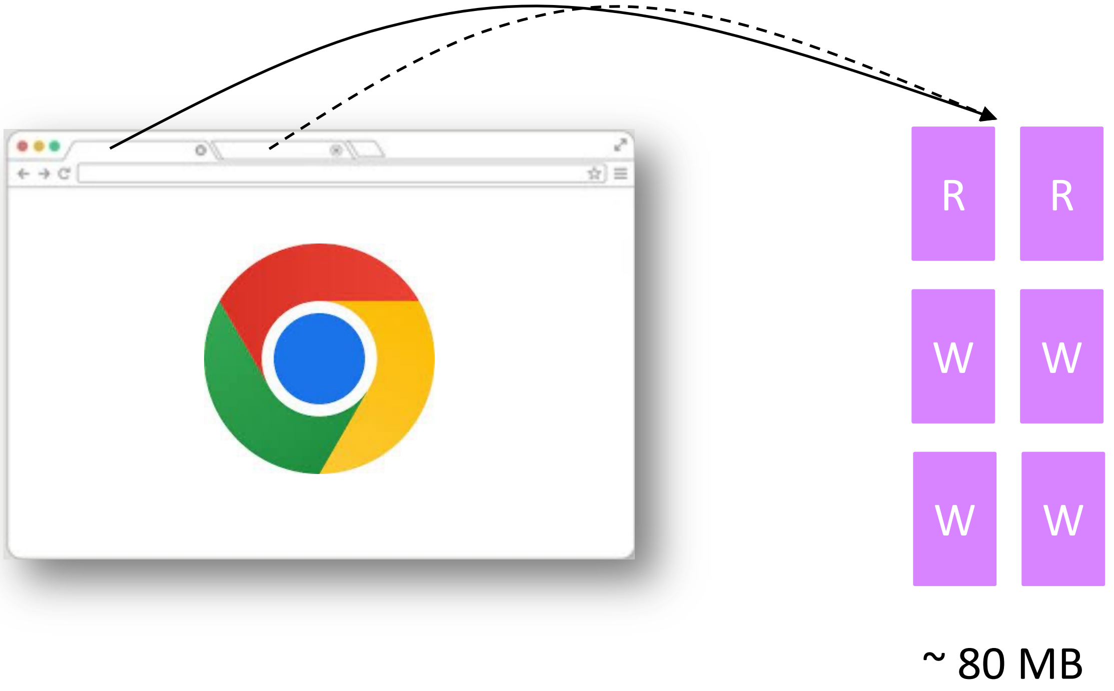
Other Useful Examples of Using Page Fault (in Modern OSes)

- **Copy-on-Write** [CoW]
  - Technique to reduce memory footprint



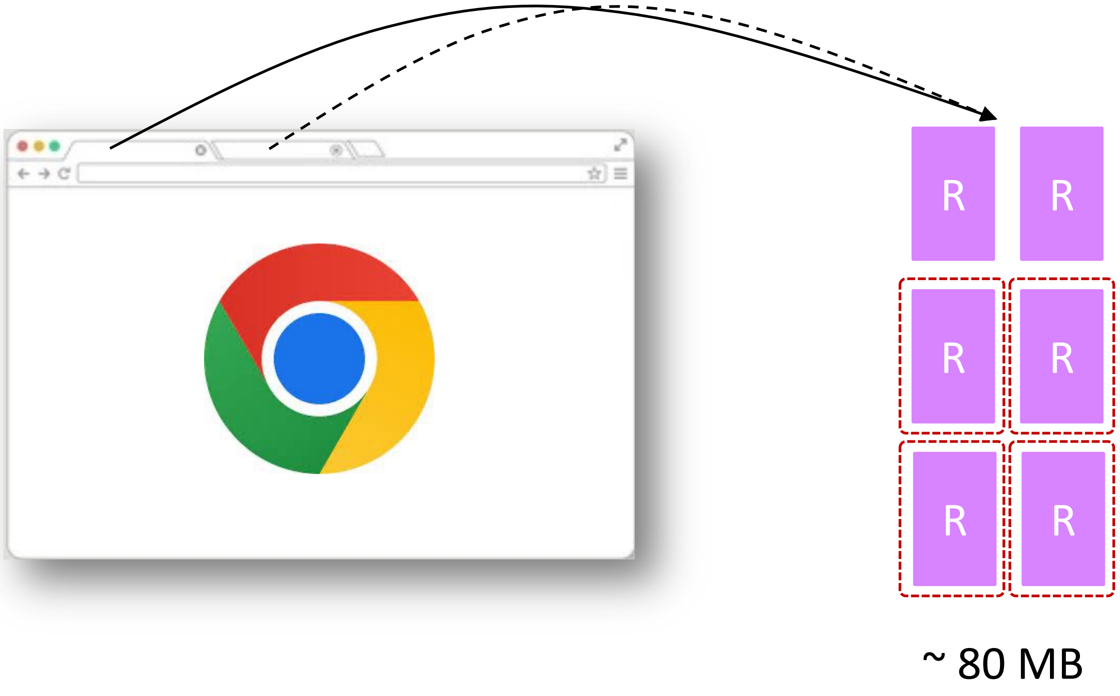
Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



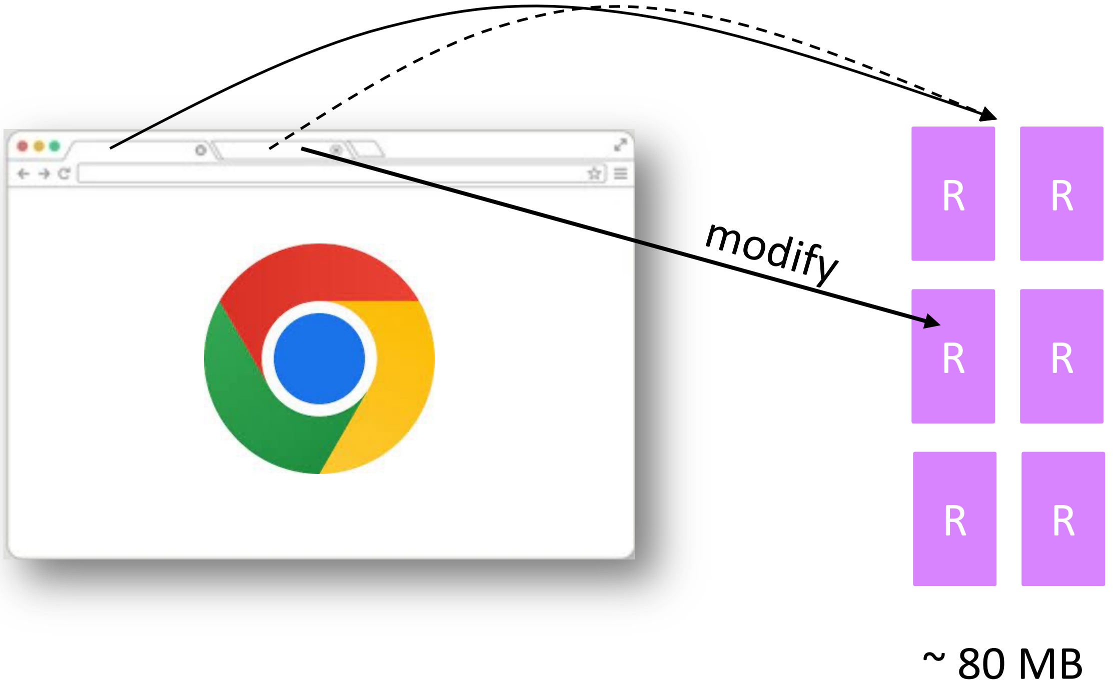
Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



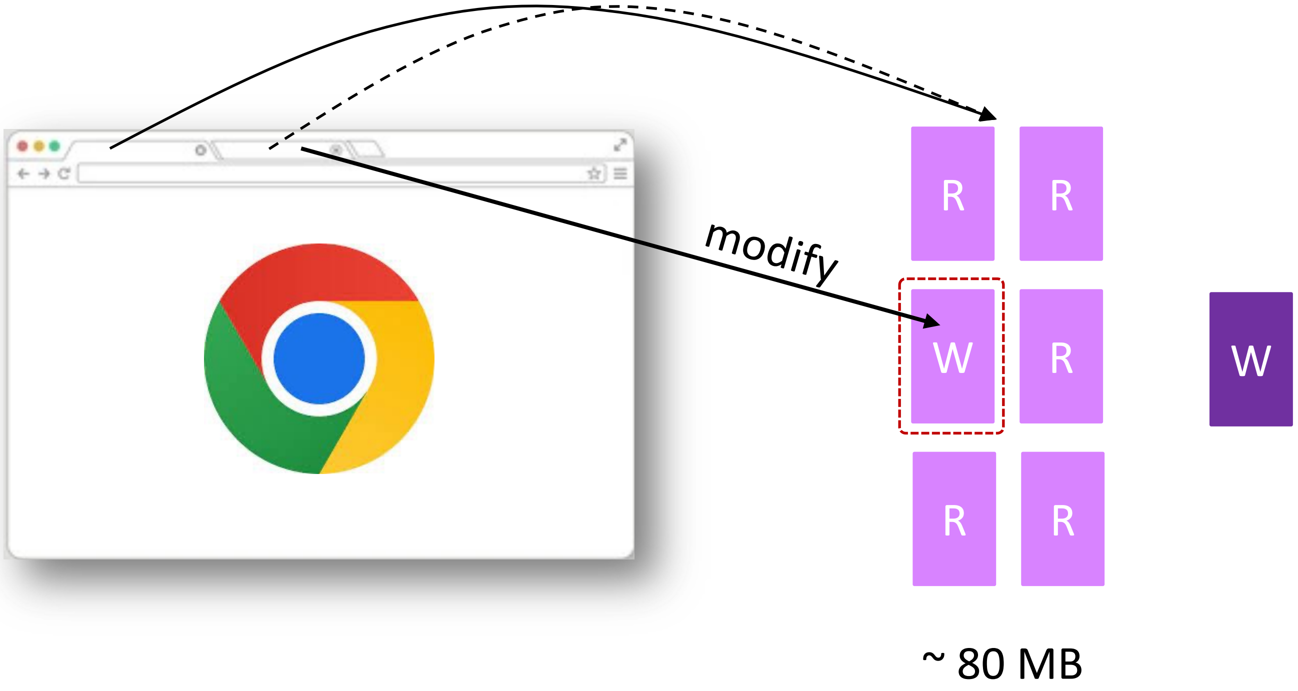
Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



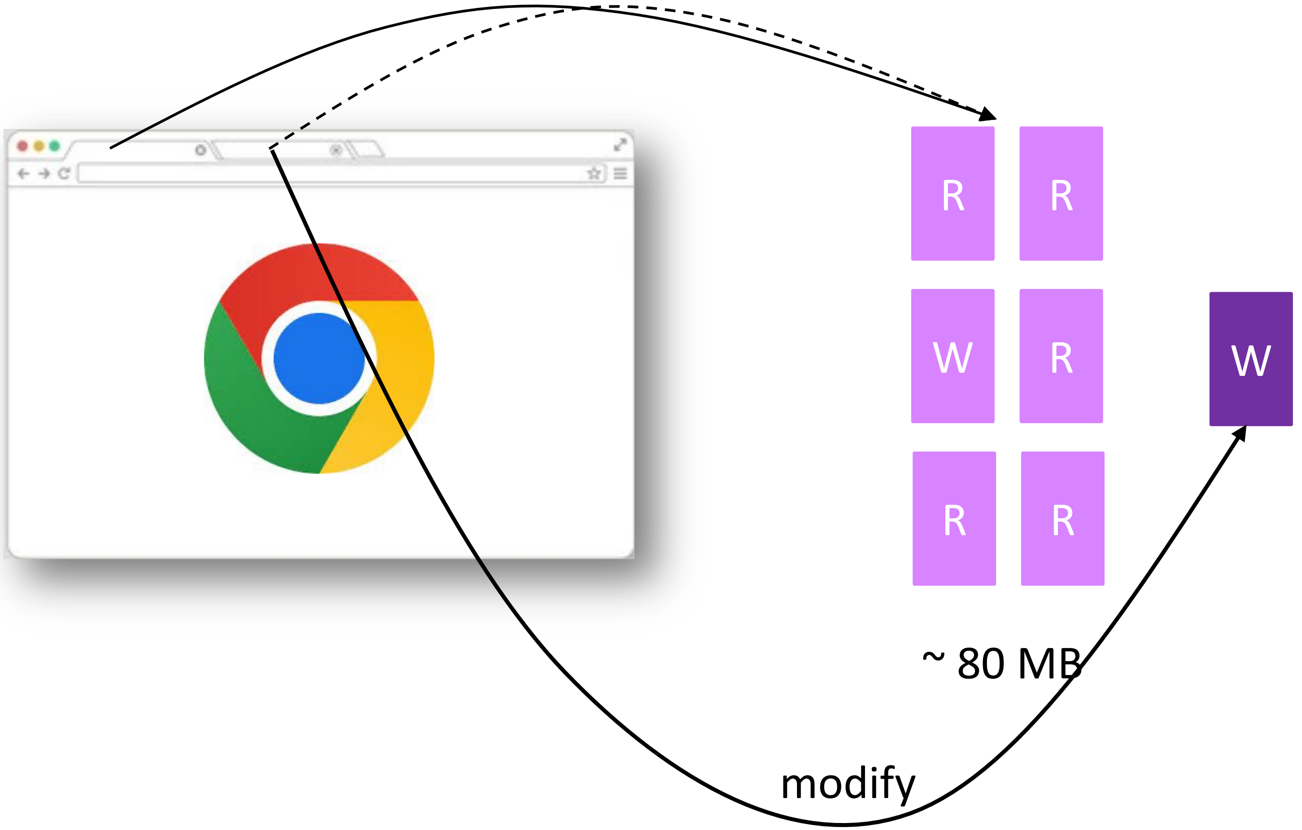
Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



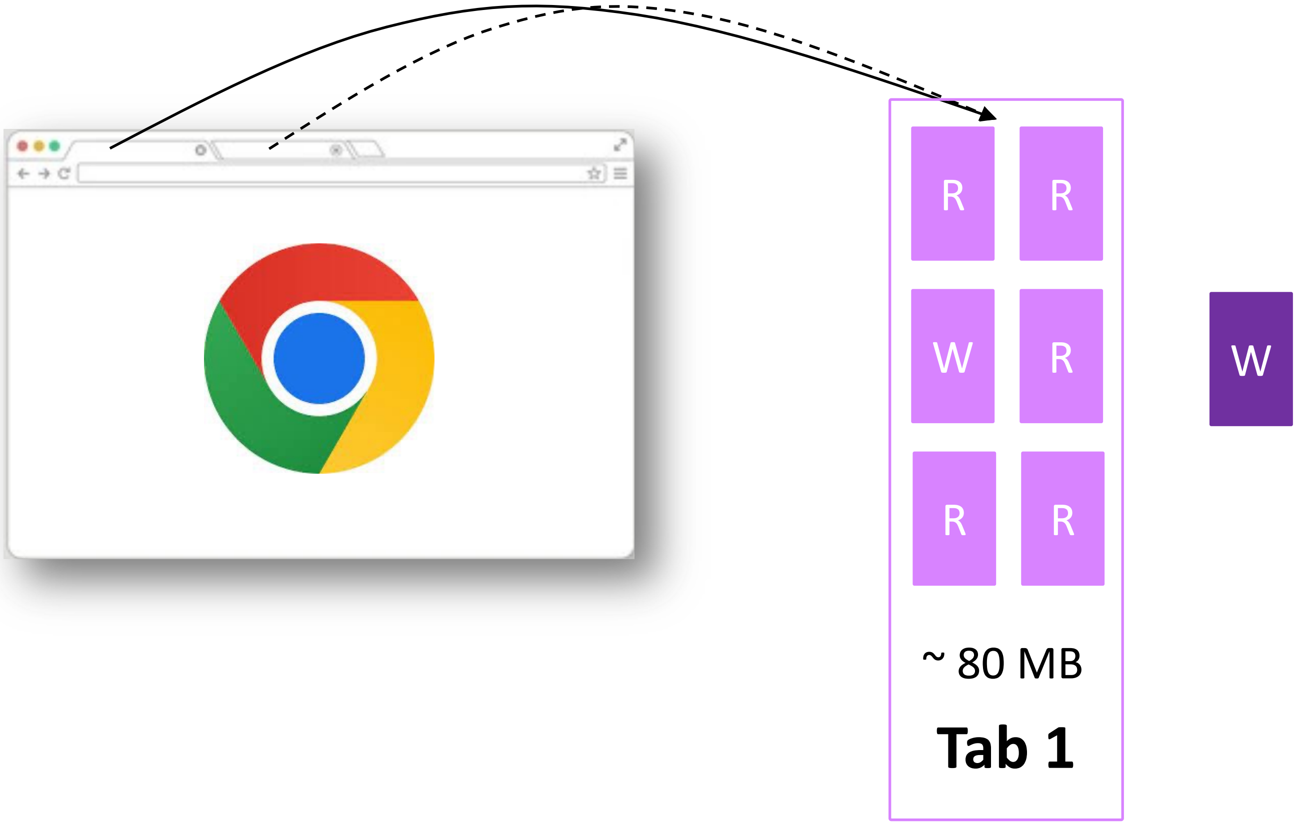
Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



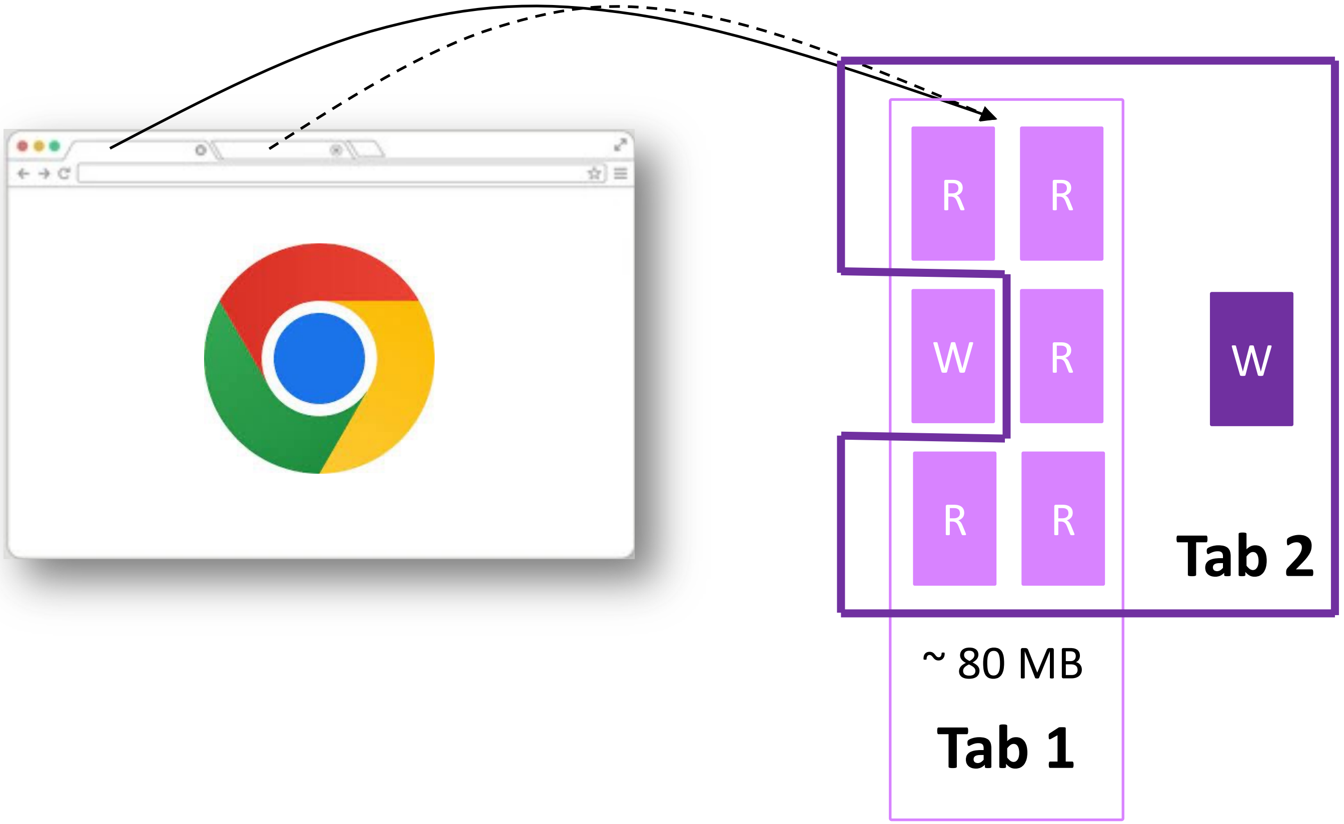
Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



Copy On Write [CoW]

- **Share** pages **read-only**
- **Create a private copy when the first write access happens**



Fork!

Memory Swapping

- **Use disk as extra space for physical memory**
- Limited RAM Size: 16/32/64 GB?
- We have a bigger storage: 1T SSD Hard Disk, cloud storage, etc.
- Store some **'currently unused but will be used later'** pages in the disk
- Store **only the active part of data in memory**

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

- os2 server
- Will run many **`/bin/bash, /usr/bin/gdb, /usr/bin/tmux`**, etc.
  - Each of you will run those programs!
  - Do we need to have 826 copies of the same program in memory?
- Build an OS to **efficiently** manage programs and **minimize** memory usage?
  - **Share physical pages of the same program!**

```
[jangye@os2 ~$] ps aux | grep bash | wc -l
826
[jangye@os2 ~$] ps aux | grep tmux | wc -l
128
[jangye@os2 ~$] ps aux | grep gdb | wc -l
90
```

Count number of processes running bash, tmux, and gdb

A Program’s Memory Layout [ELF]

`.text`

- Code area. Read-only and executable

`.rodata`

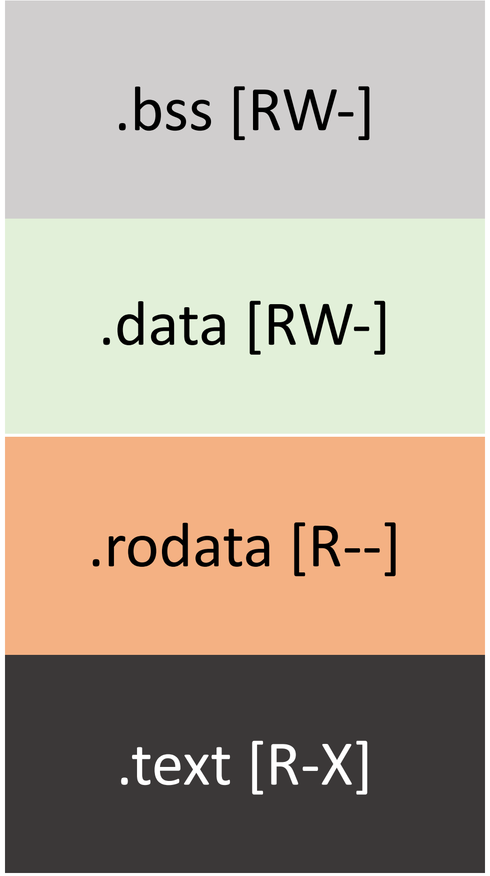
- Data area, Read-only and not executable

`.data`

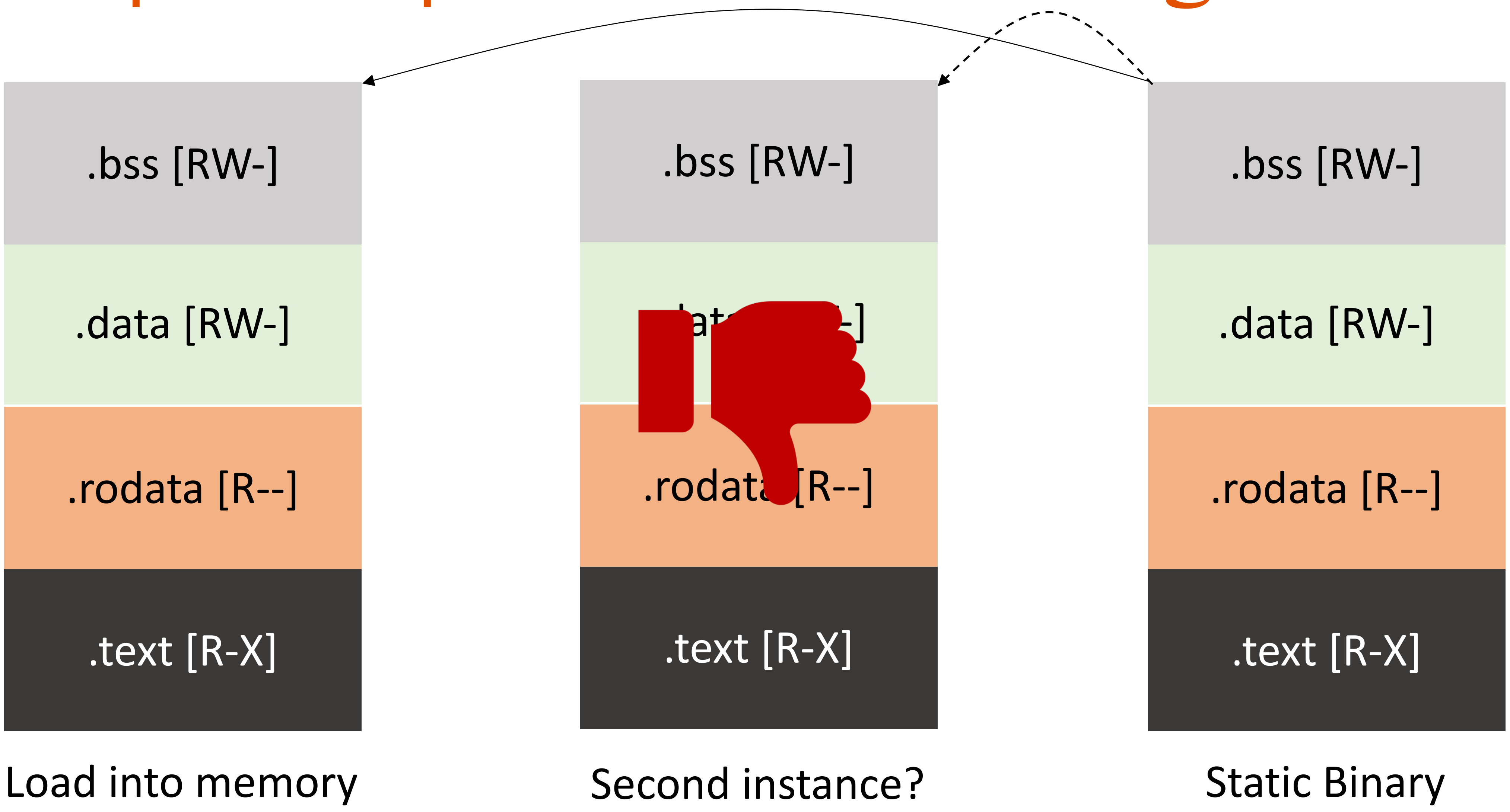
- Data area, Read/Writable (not executable)
- Initialized by some values

`.bss`

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



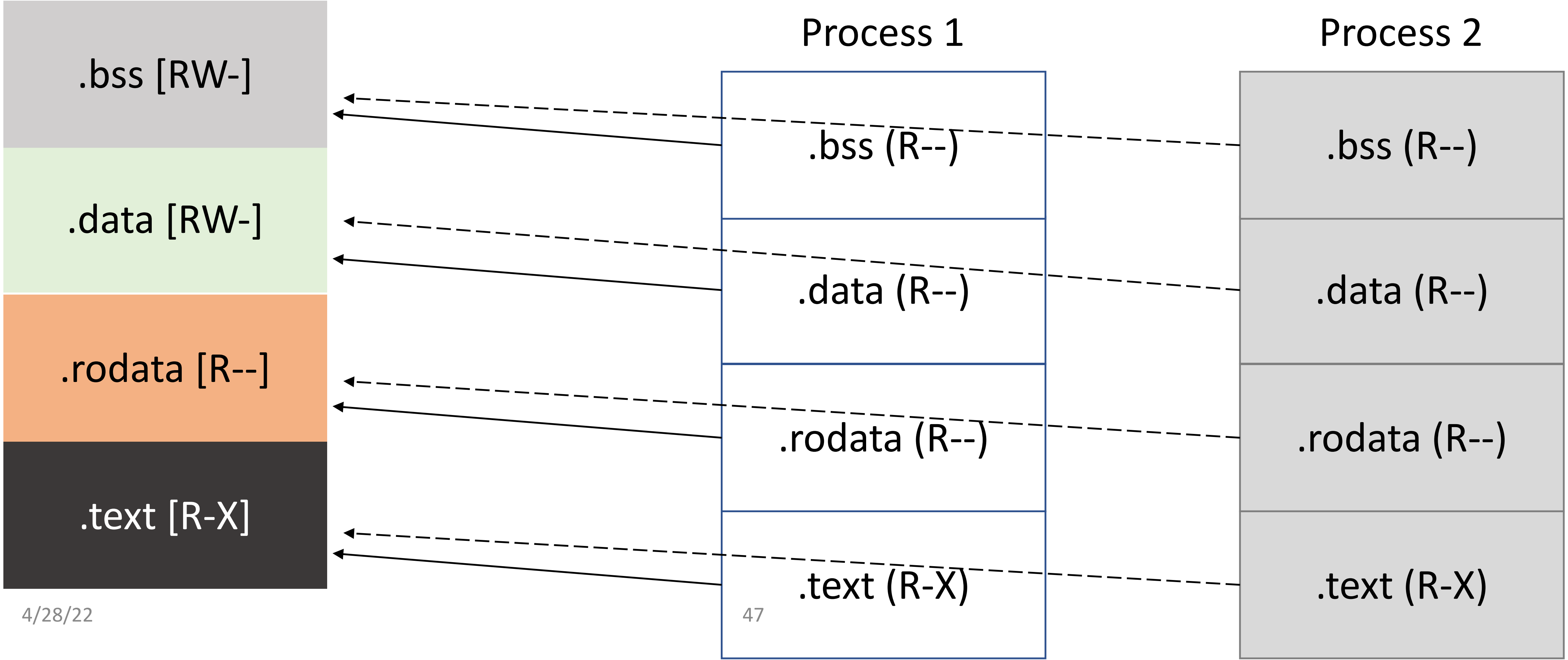
Multiple Copies of Same Program



Sharing by Read-only

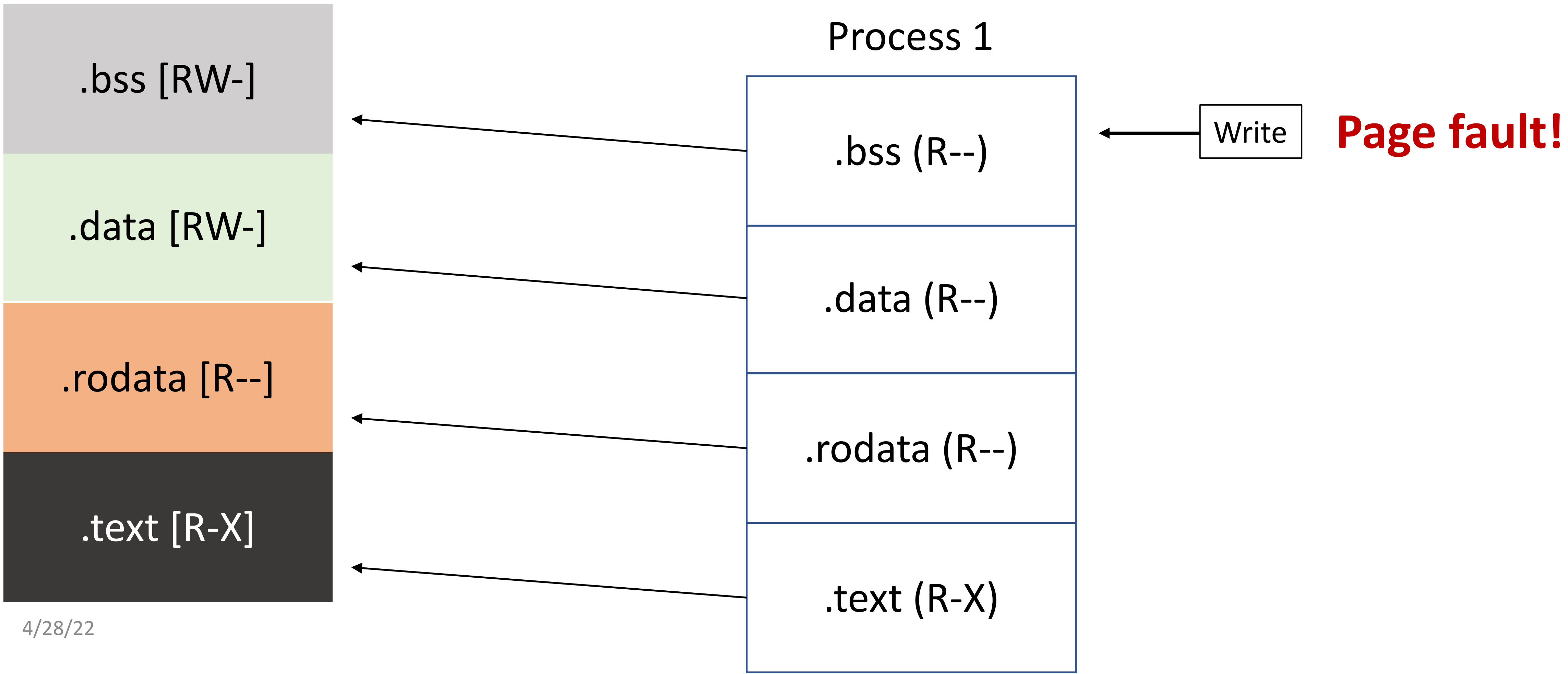
Create **Page Directory** and **copy entries**!

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



OK for Read-only Sections

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



Page Fault Handler

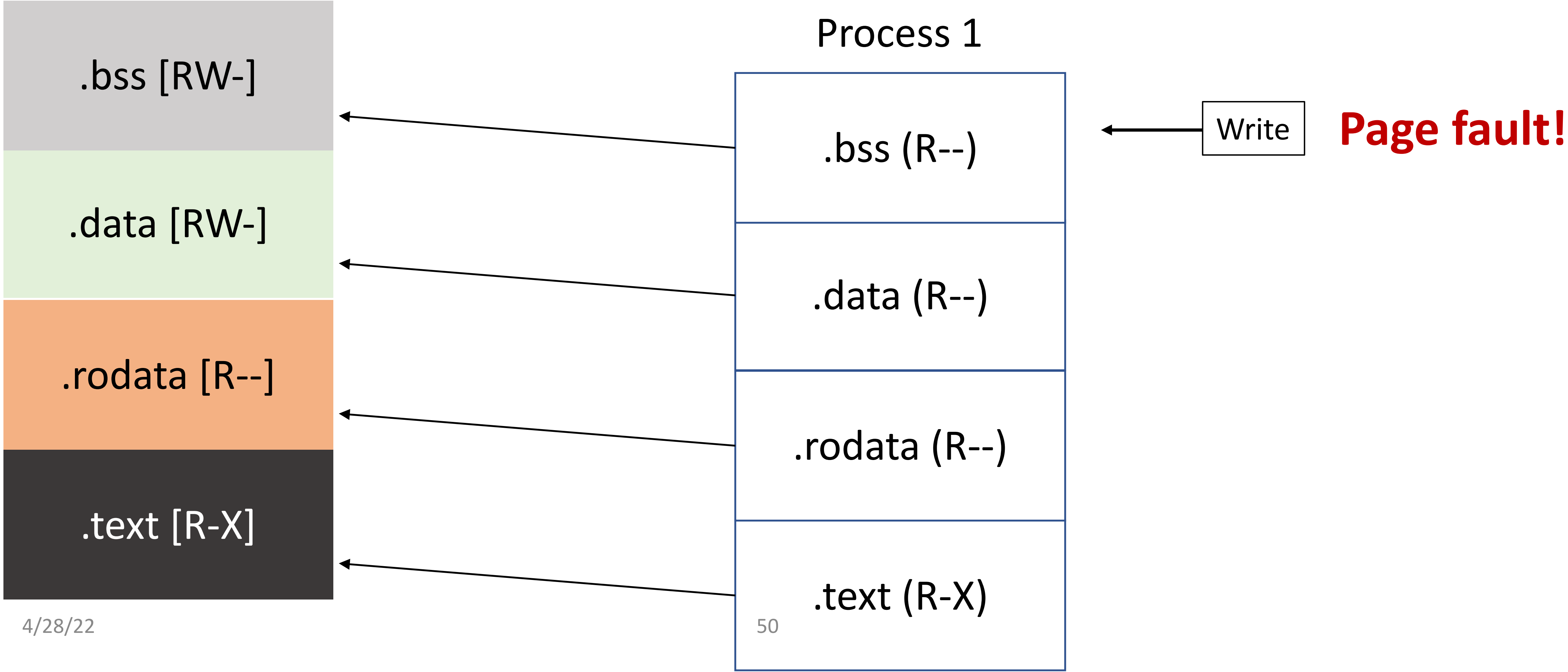
- Read CR2
- An address that is in the page cache
- **Fault from a shared location!**
- Read Error code
    - Write on read-only memory
    - **Process requires a private copy**!

      [we 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 contents
    - Mark as read/write
    - Resume

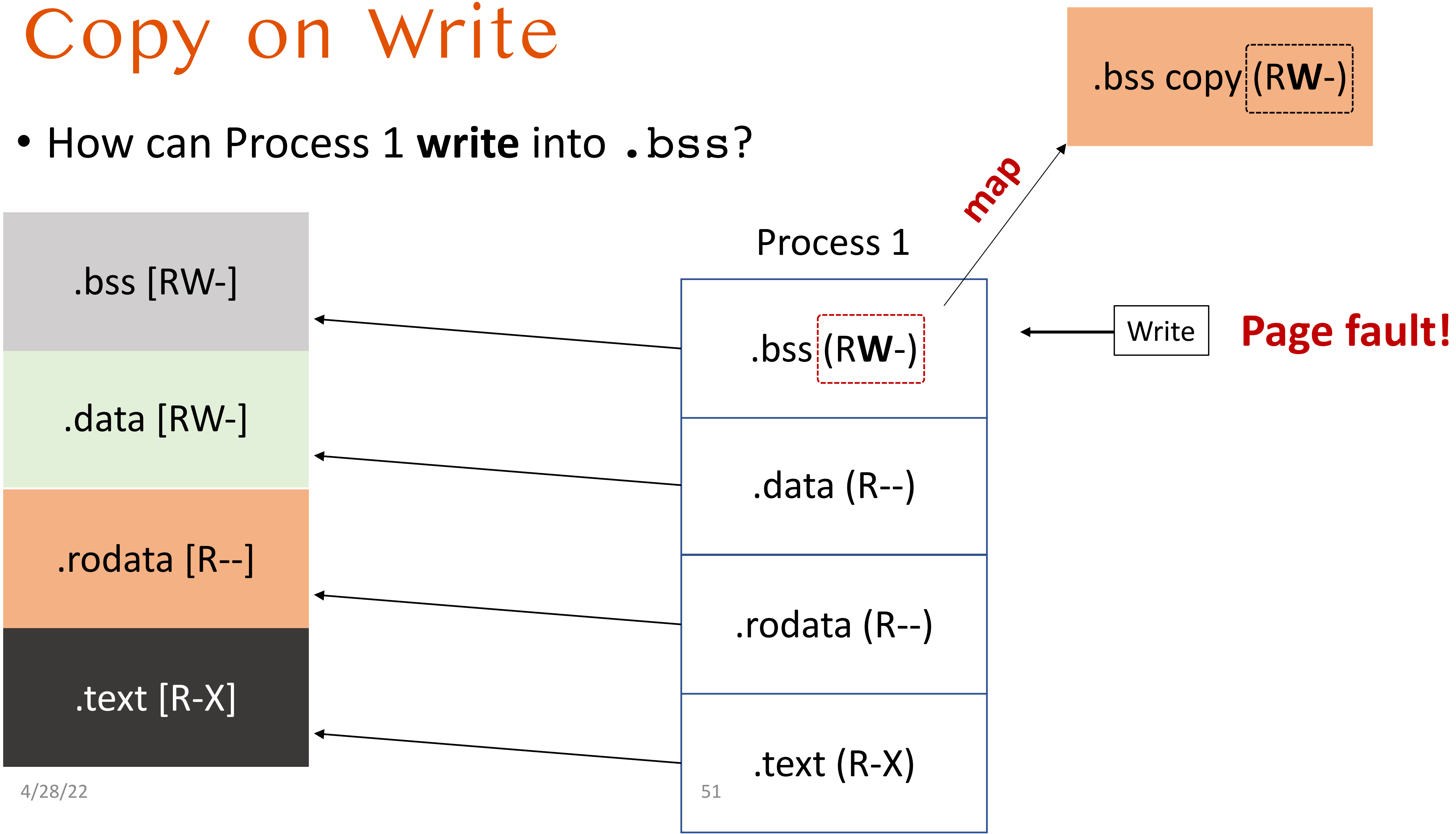
Copy on Write

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



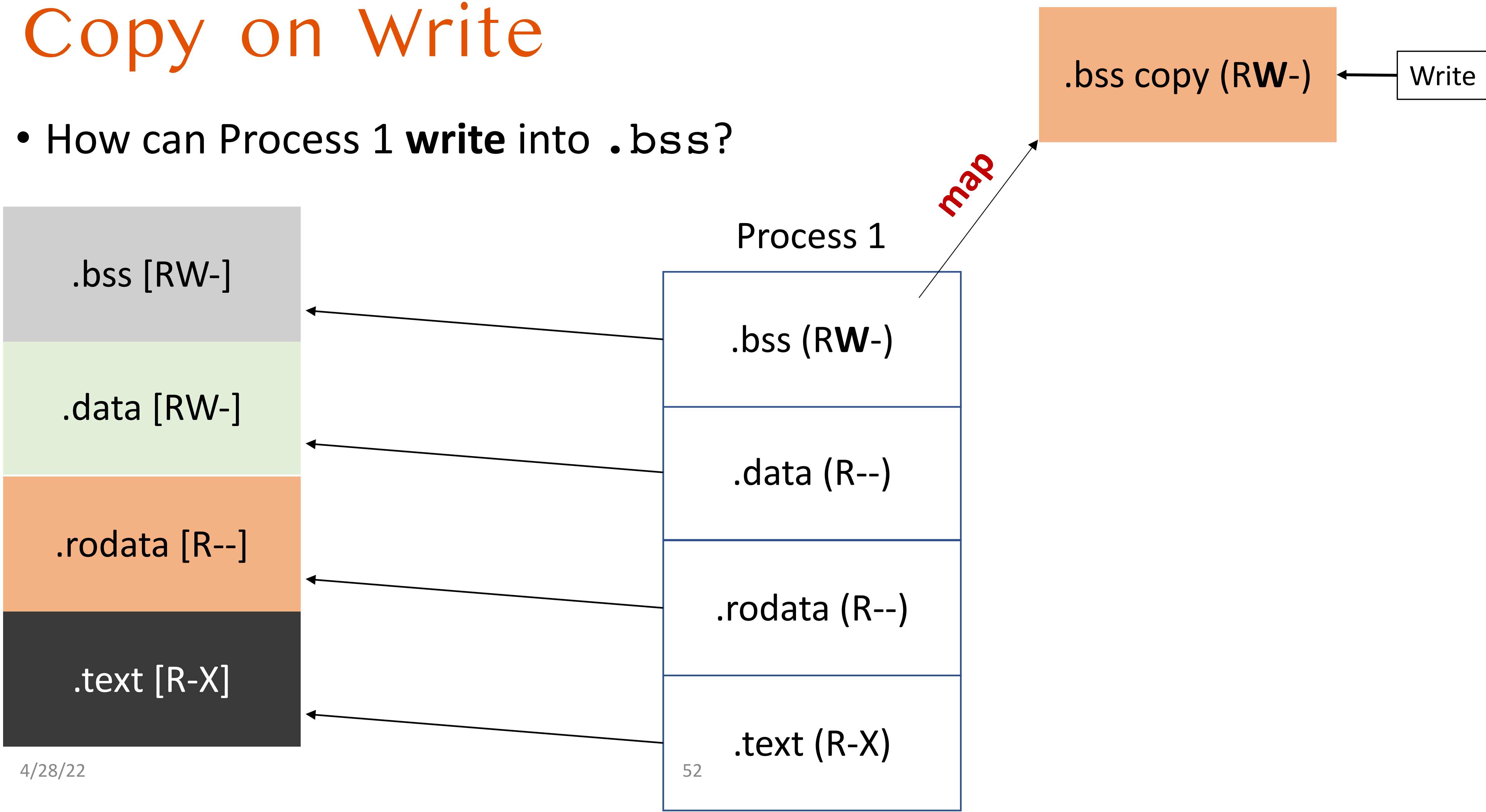
Copy on Write

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



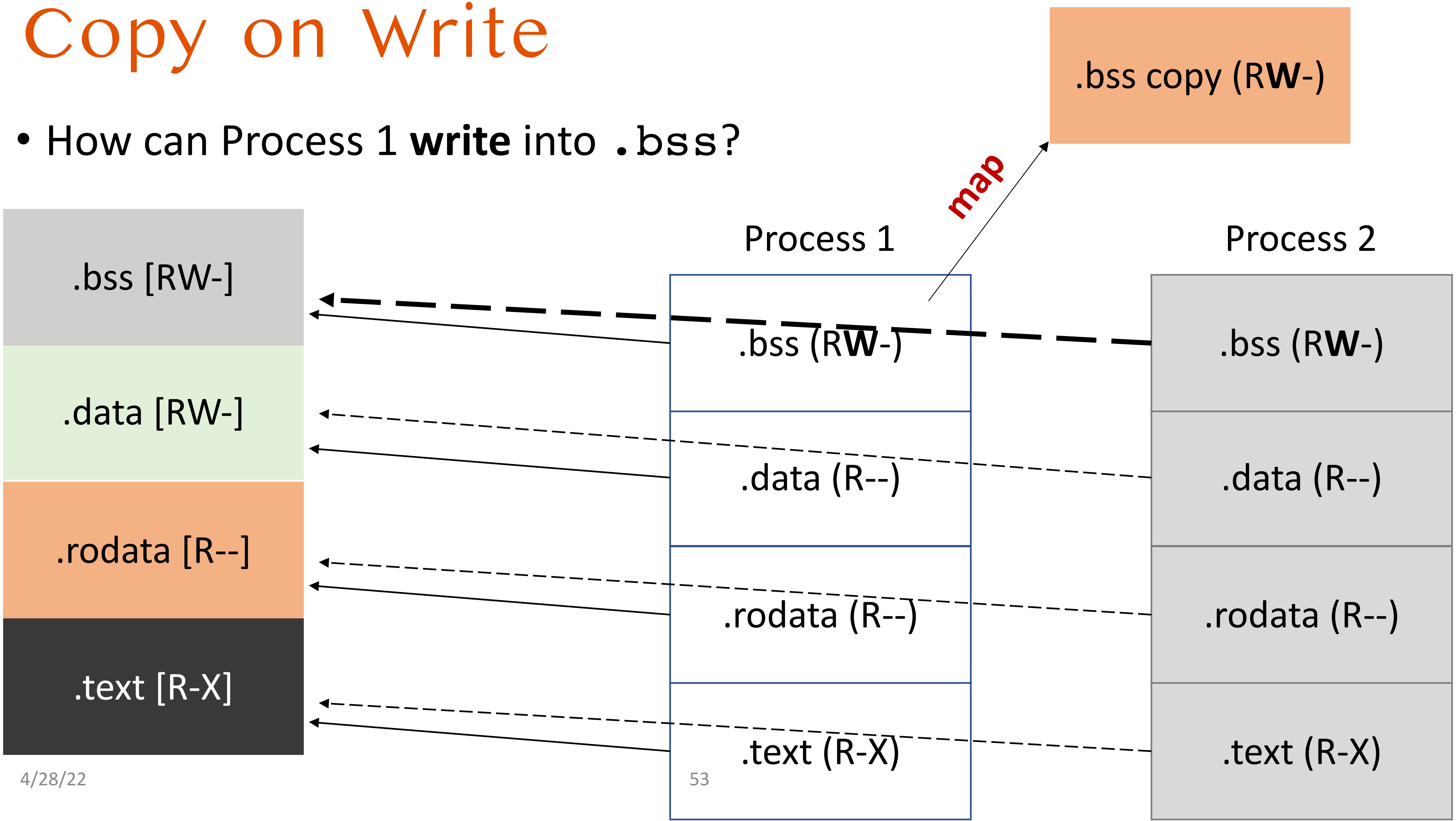
Copy on Write

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



Copy on Write

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

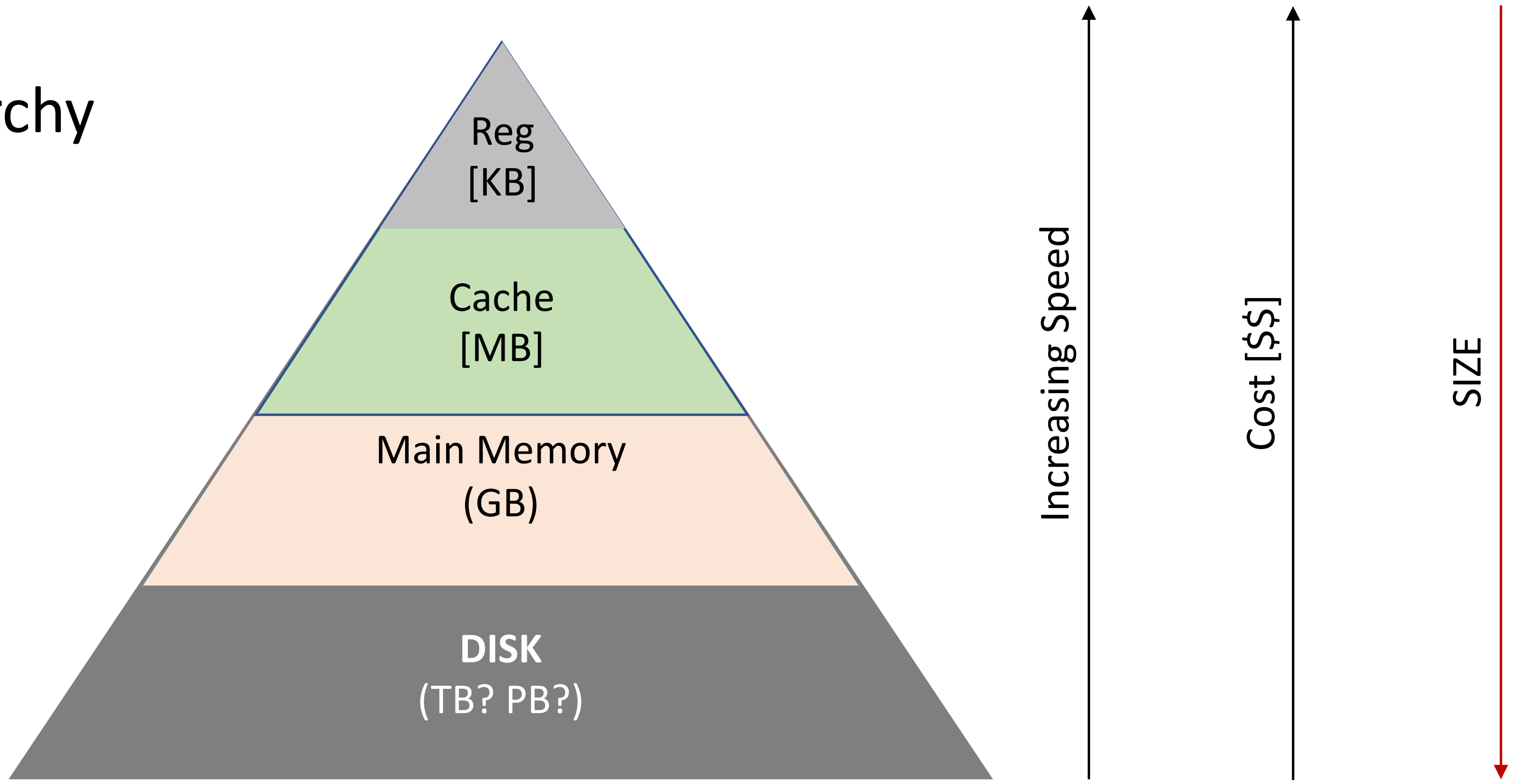


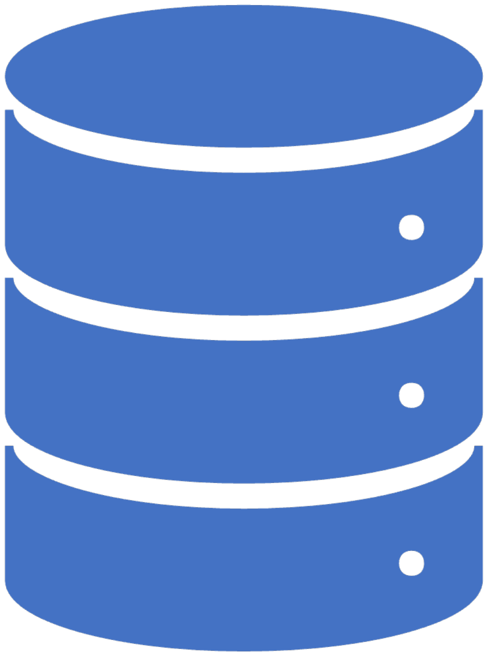
Benefits? using **page faults**!

Better performance!	More **efficient**!	Additional benefits
• reduce time for copying contents already in physical memory (page cache)	• reduce physical memory use • sharing code/read-only data among multiple processes • 1,000,000 processes, requiring only 1 copy of .text/.rodata	• Can support **sharing of writable pages** [unless modified] • Can **create private pages** seamlessly on write

Memory Swapping

- Memory Hierarchy

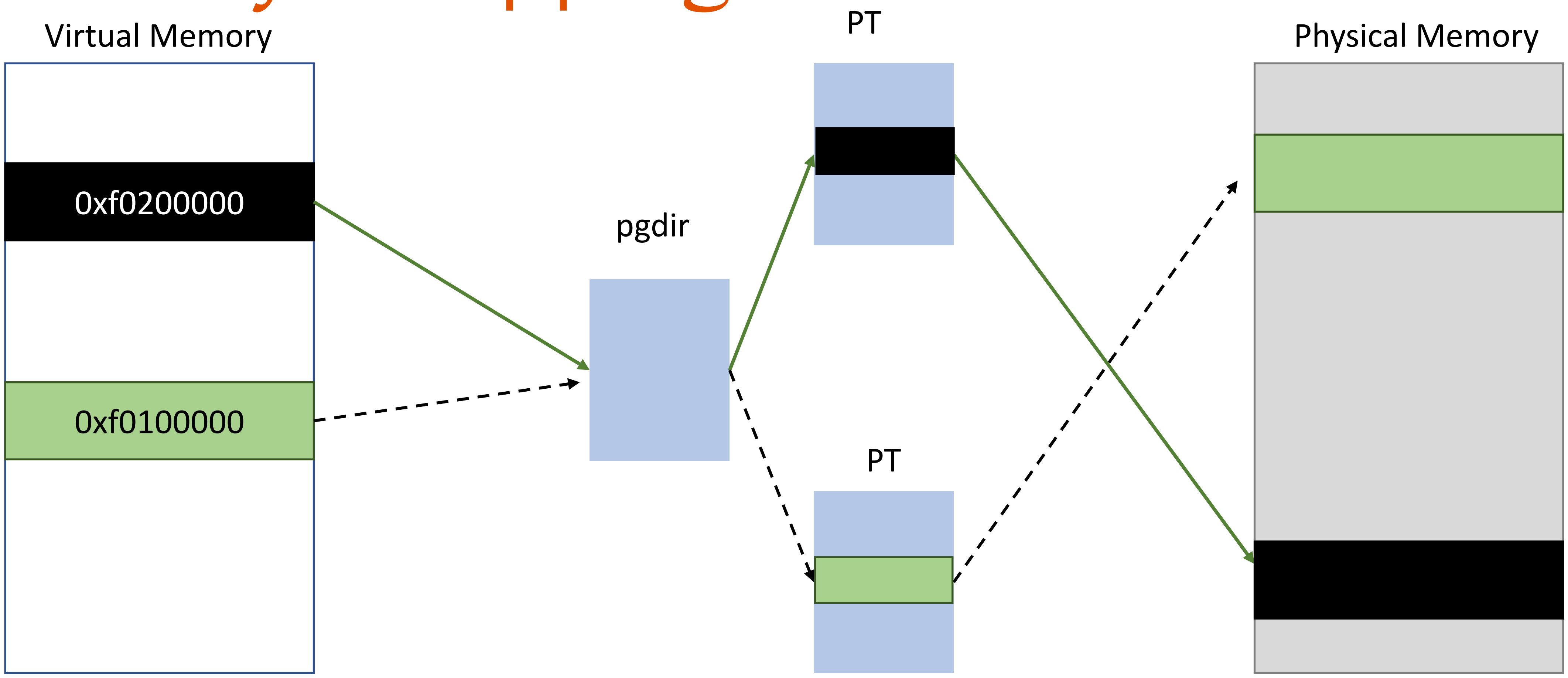




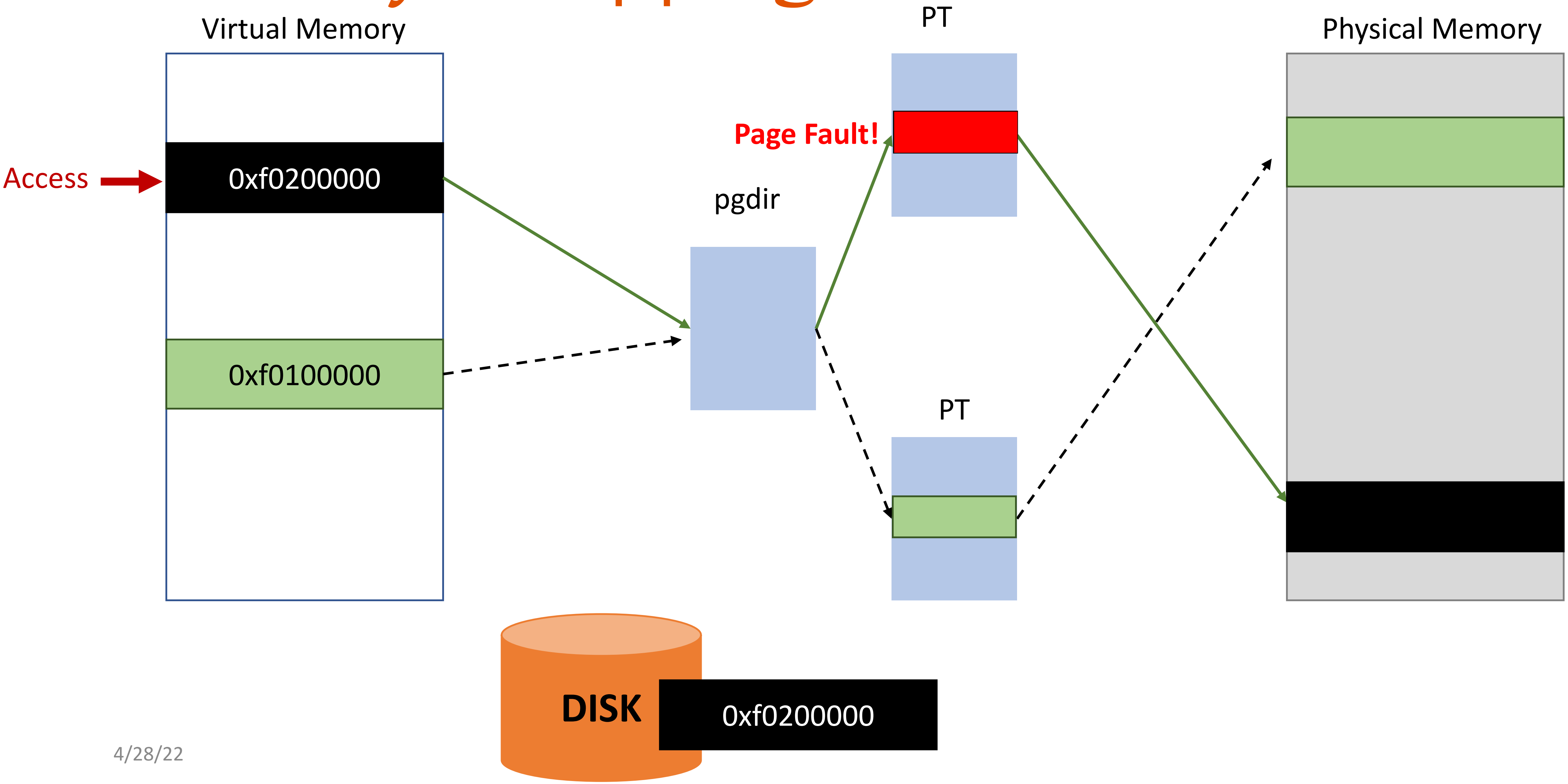
Challenge

- Suppose you have 8GB of main memory
- Can you run a program that 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

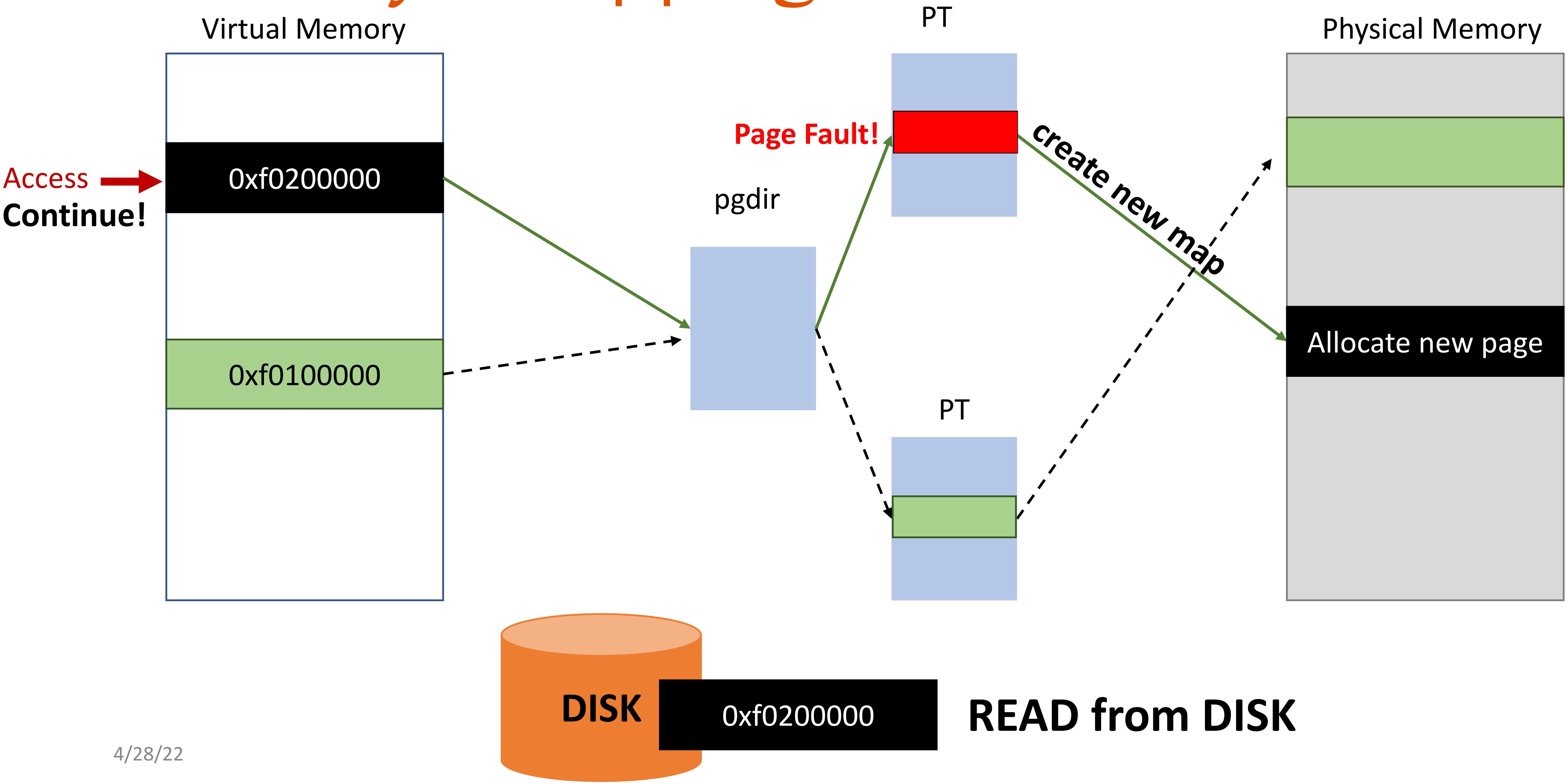


Swapping | OS

- 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



Page Fault | Summary

- Generated for memory errors [during paging]
- A **recoverable exception**
- User program may resume the execution

- Is useful for implementing
  - Automatic stack allocation
  - Copy-on-write (will do in Lab4)
  - Swapping

Backup Slides

Check How Library Calls are Invoked

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

```
read(0, "asdfzxcv\n", 512)                                      = 9
printf("Read to stack memory returns: %d"..., 9)                = 32
read(0 <no return ...>
error: maximum array length seems negative
, "", 512)                                                      = -1
printf("Read to kernel memory returns: %"..., -1)               = 34
perror("Reason for the error:"Reason for the error:: Bad address
)                                                  = <void>
+++ exited (status 0) +++
```