

# CS444/544

# Operating Systems II

Lecture 11

Virtualization Summary and Quiz 2 Prep

5/8/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang



**Oregon State**  
**University**

# Odds and Ends

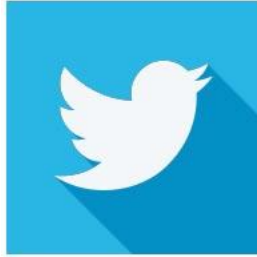
- Lab 2 will be graded by this weekend
- Quiz 2 on Monday (more info later)

# Today's Topic

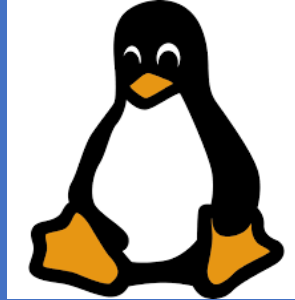
- Virtualization -> Concurrency
- OS Three Easy Pieces
  - Virtualization (memory, process, user/kernel, etc.)
  - Concurrency (multi-threading, multi-process, scheduling, synchronization)
  - Persistence (disk, file, snapshot, etc.)
- Do recap on Virtualization
- Quiz 2 Prep.

# What is an OS?

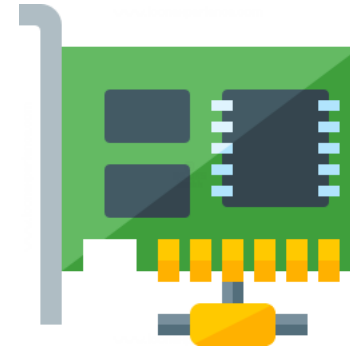
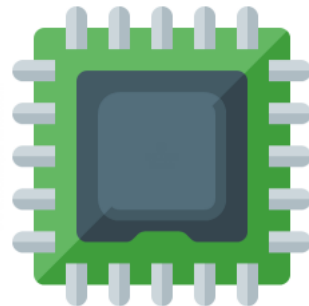
Applications



OS



Hardware



# Memory

Backward compatibility: BIOS will put the code that assumes your CPU as **i8086** (a **43-years old 16-bit** CPU). So we need to start with **16-bit mode** and then enable **32-bit protected mode**, **paging**, etc...

**UEFI** does not go through this process; they directly starts with 32-bit or 64-bit mode, so OSes do not have to handle these things..

- 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

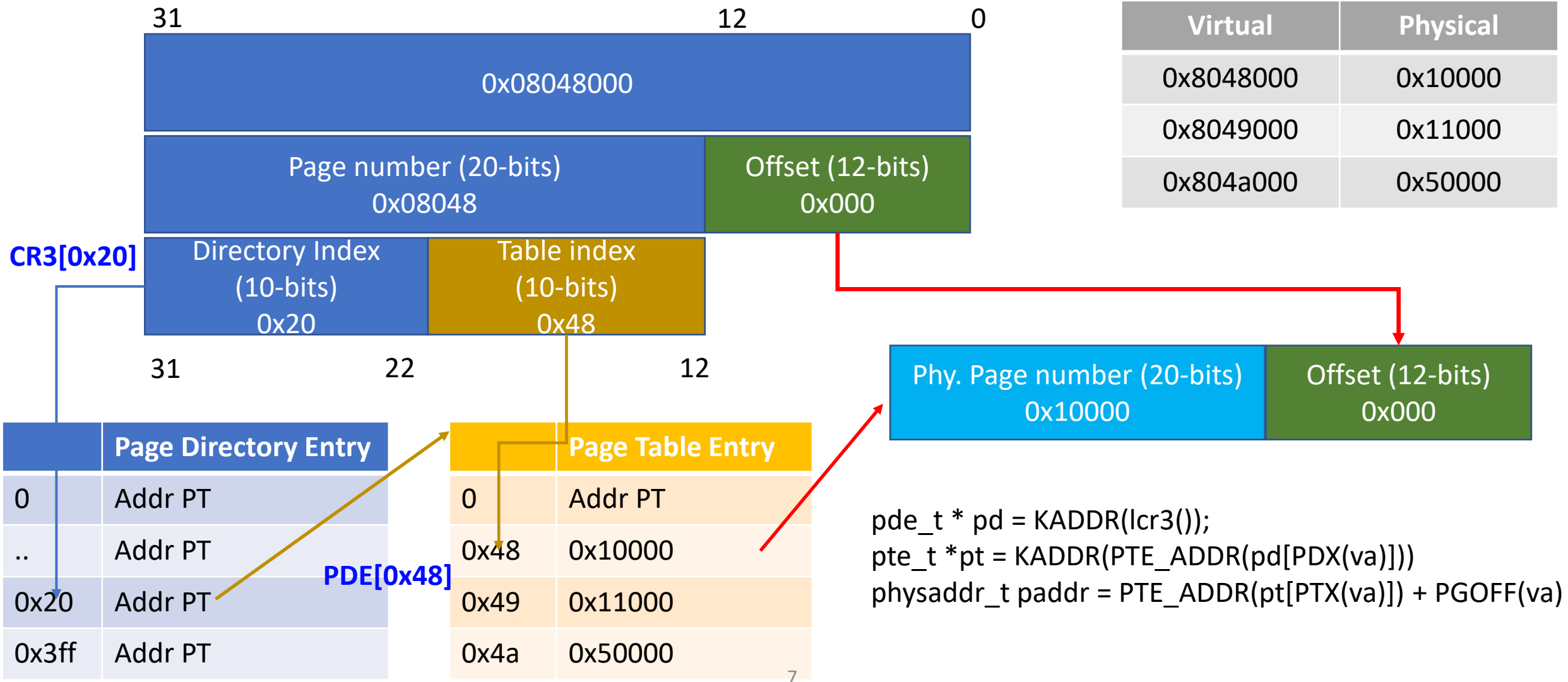
31				16				15				0							
Base 0:15								Limit 0:15											
63		56		55		52		51		48		47		40		39		32	
Base 24:31				Flags				Limit 16:19				Access Byte				Base 16:23			



# Virtual Memory

- Paging
  - When enabled, all memory address will be translated via
  - CR3 -> PDE -> PTE -> Physical Page!

# Recap – Page Table & Addr Translation

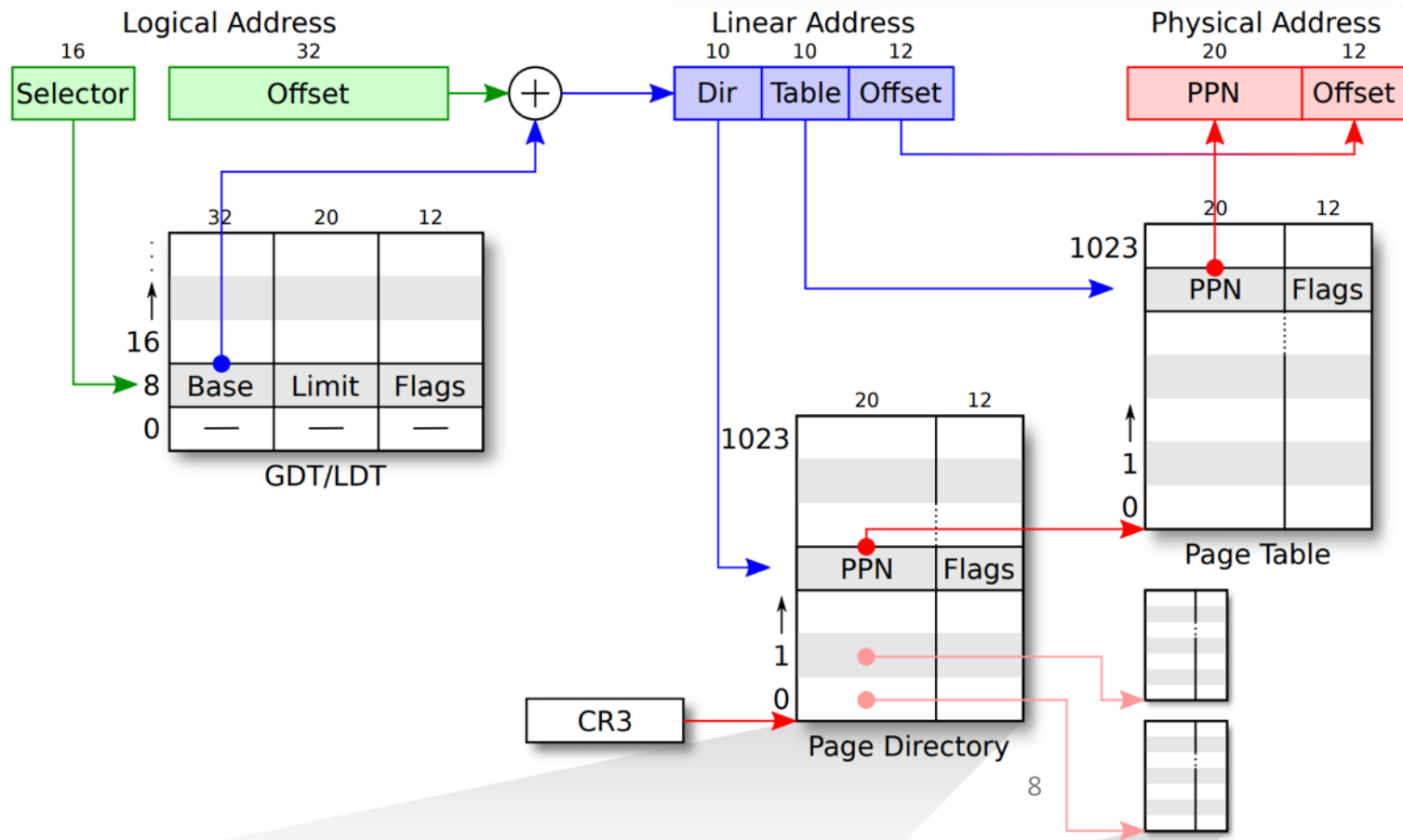
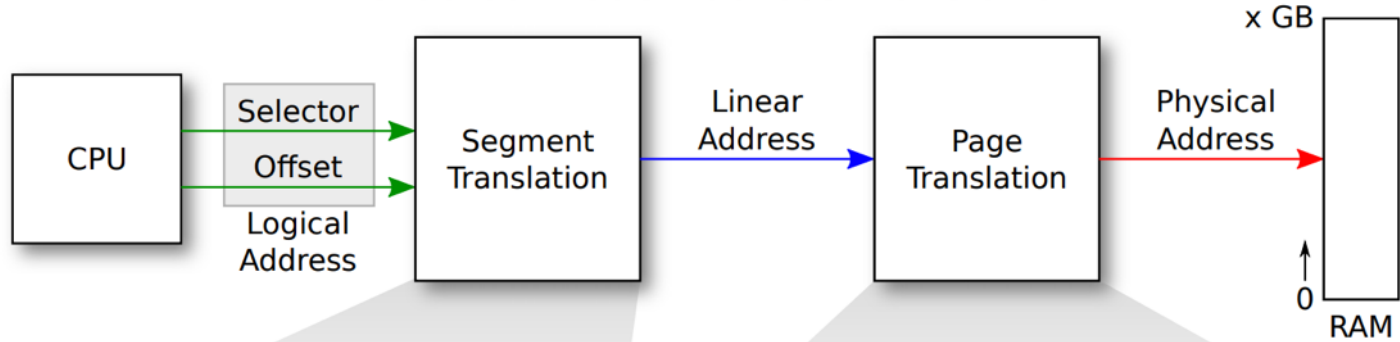


Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x50000

```

pde_t * pd = KADDR(lcr3());
pte_t * pt = KADDR(PTE_ADDR(pd[PDX(va)]))
physaddr_t paddr = PTE_ADDR(pt[PTX(va)]) + PGOFF(va)
    
```

# Protected-Mode Address Translation





# Why Virtual Memory?

- Three goals
  - Transparency: does not 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; manage 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?

# Paging: Virtual Memory

- Having an indirect table that maps virt-addr to phys-addr

Physical Memory

Stack-2 0xbffdf000	Stack 0xbffdf000
Program code-2 0x804a000	Program code 0x804a000
Program code-2 0x8049000	Program code 0x8049000
Program code-2 0x8048000	Program code 0x8048000

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

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

Stack-2 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000

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

Program A is loaded at **0x8048000**.

Can Program B be loaded at **0x8048000**?

- Having an indirect table that maps virt-addr to phys-addr

Physical Memory

Stack-2 0xbffdf000
Program code-2 0x804a000
Program code-2 0x8049000
Program code-2 0x8048000

Stack 0xbffdf000
Program code 0x804a000
Program code 0x8049000
Program code 0x8048000

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

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

Stack-2 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000

**Efficiency:** do not waste memory  
 Can Program B (288KB) be loaded if  
 only **288 KB** of memory is free, regardless of its allocation?

- Having an indirect table that maps virt-addr to phys-addr

Physical Memory

Stack-2 0xbffdf000
Program code-2 0x804a000
Program code-2 0x8049000
Program code-2 0x8048000

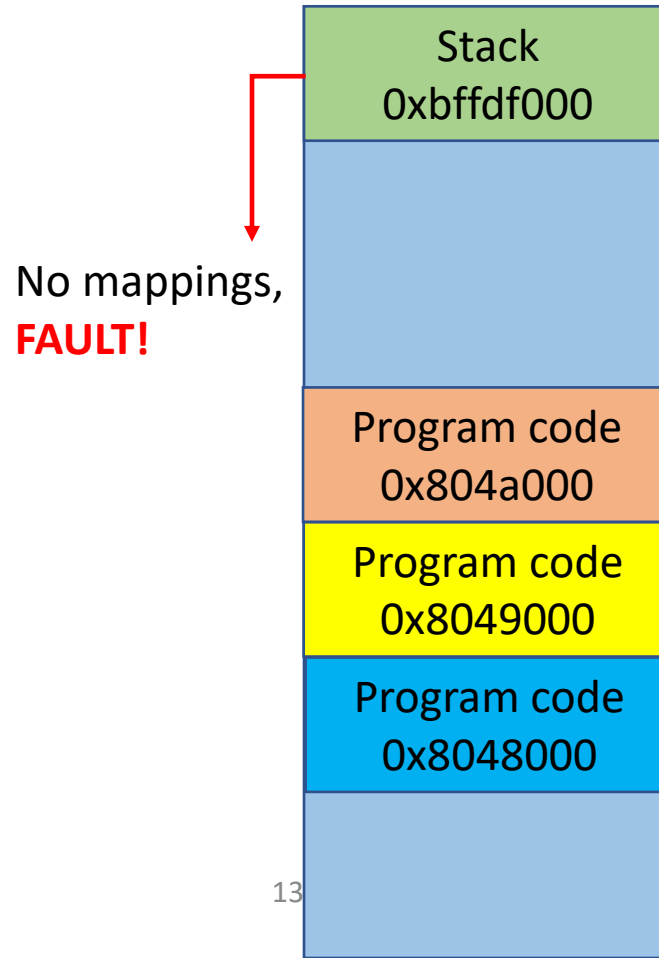
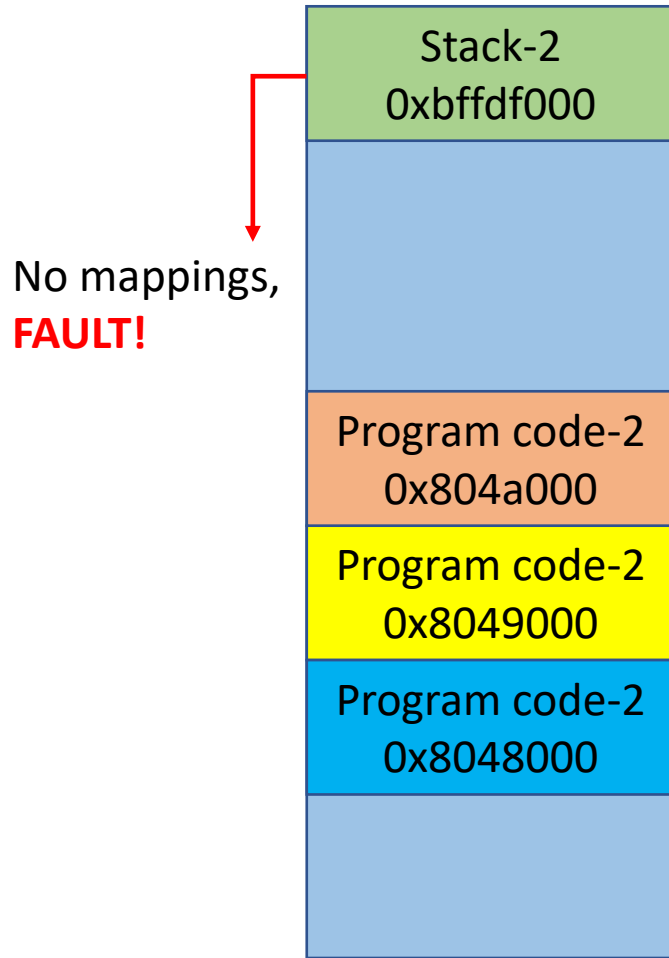
Stack 0xbffdf000
Program code 0x804a000
Program code 0x8049000
Program code 0x8048000

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

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

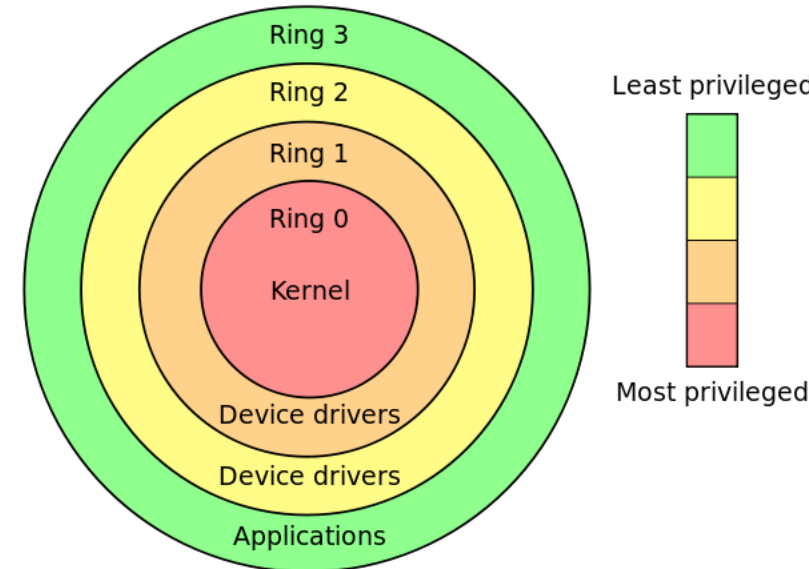
Stack-2 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000

**Protection:** isolate program's execution environment  
Can we prevent an **overflow from Program A** from  
overwriting **Program B's data**?



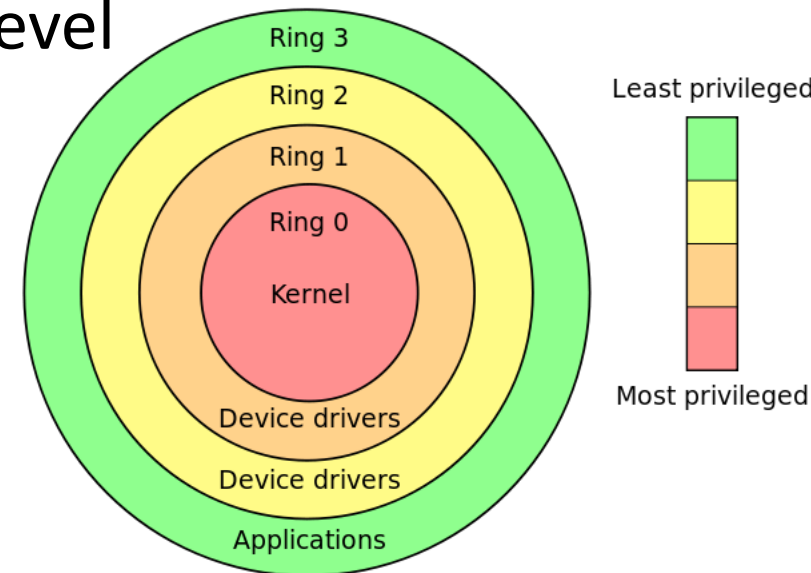
# Kernel (Ring 0)

- Runs with the 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)
  - Set privilege, restrict other jobs from doing something bad, etc.

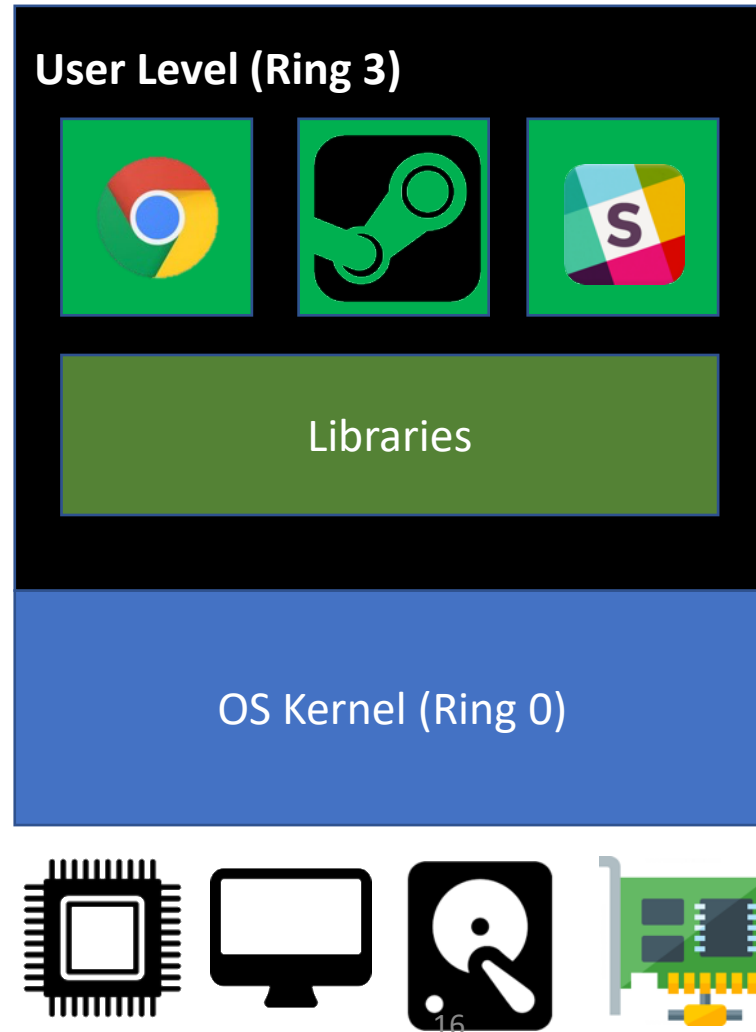


# User Level (Ring 3)

- Runs with a restricted privilege (Ring 3)
  - The privilege level for running an application...
- Most of our regular applications runs in this ring level
- Cannot access kernel memory
  - Can only access pages set with PTE\_U
- Cannot talk directly to hardware devices
  - Kernel must mediate the access

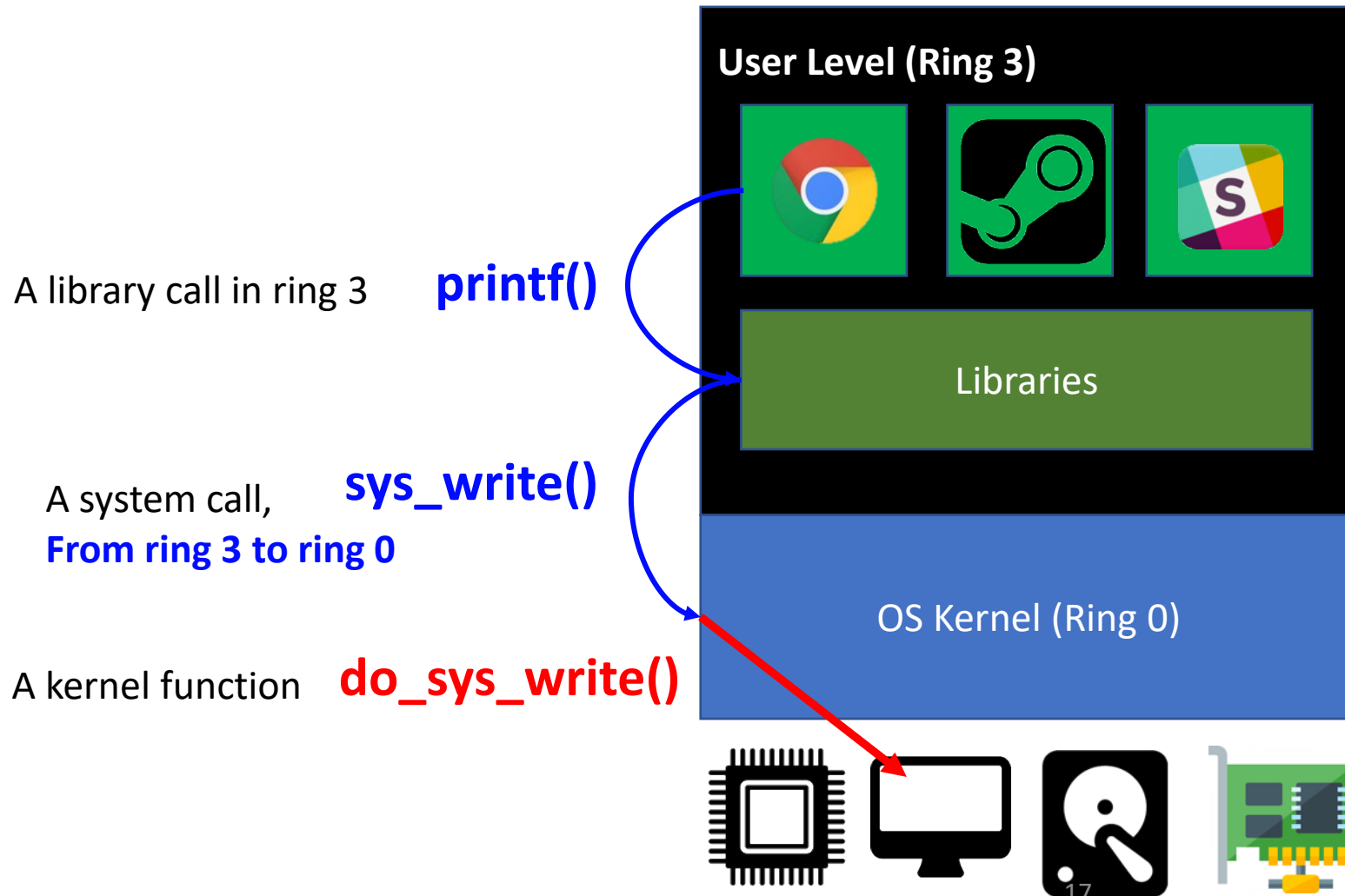


# A High-level Overview of User/Kernel Execution

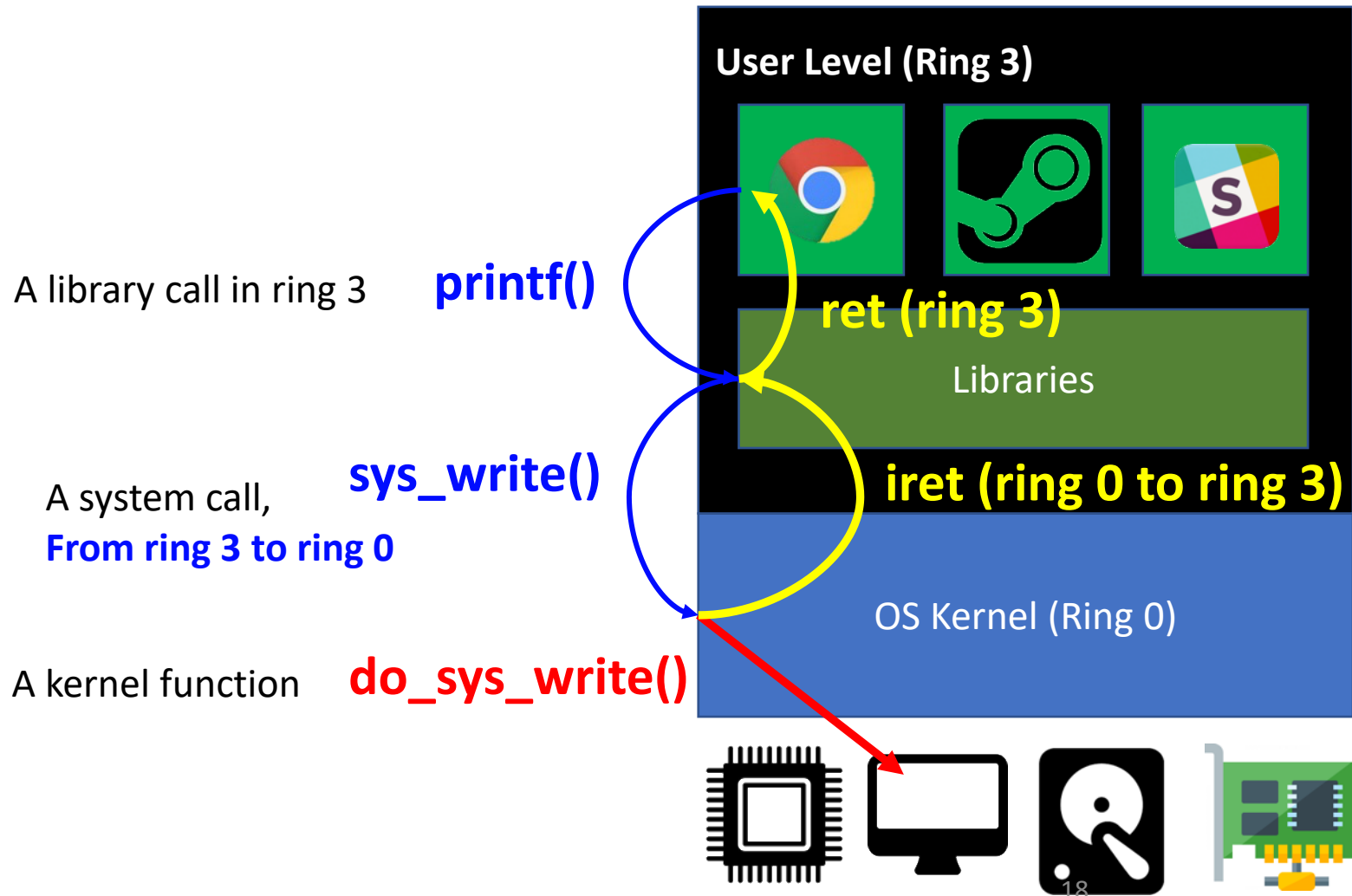




# A High-level Overview of User/Kernel Execution



# A High-level Overview of User/Kernel Execution



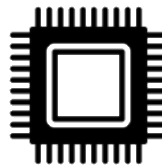
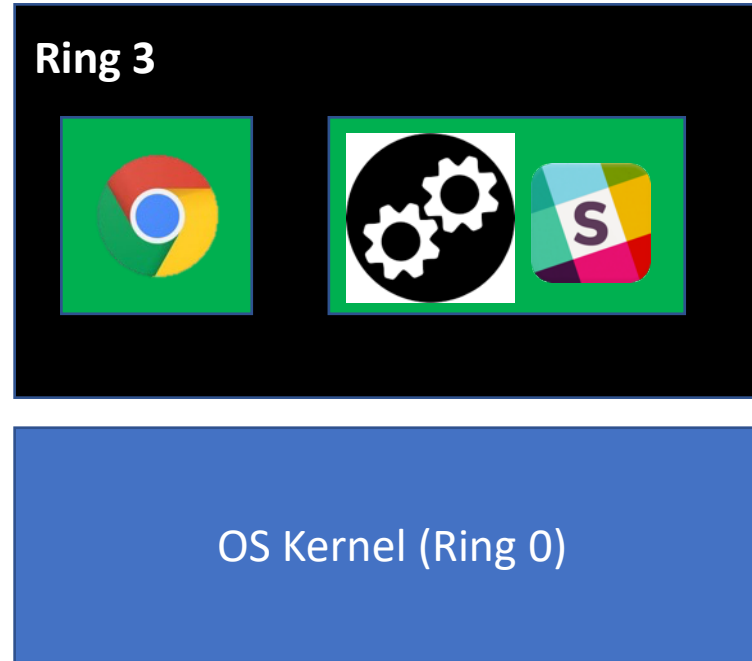
# User Execution Strawman 2'

- What if a process runs

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

Much wait

No such yield()  
Too long



# User Execution Strawman 2'

- What if a process runs

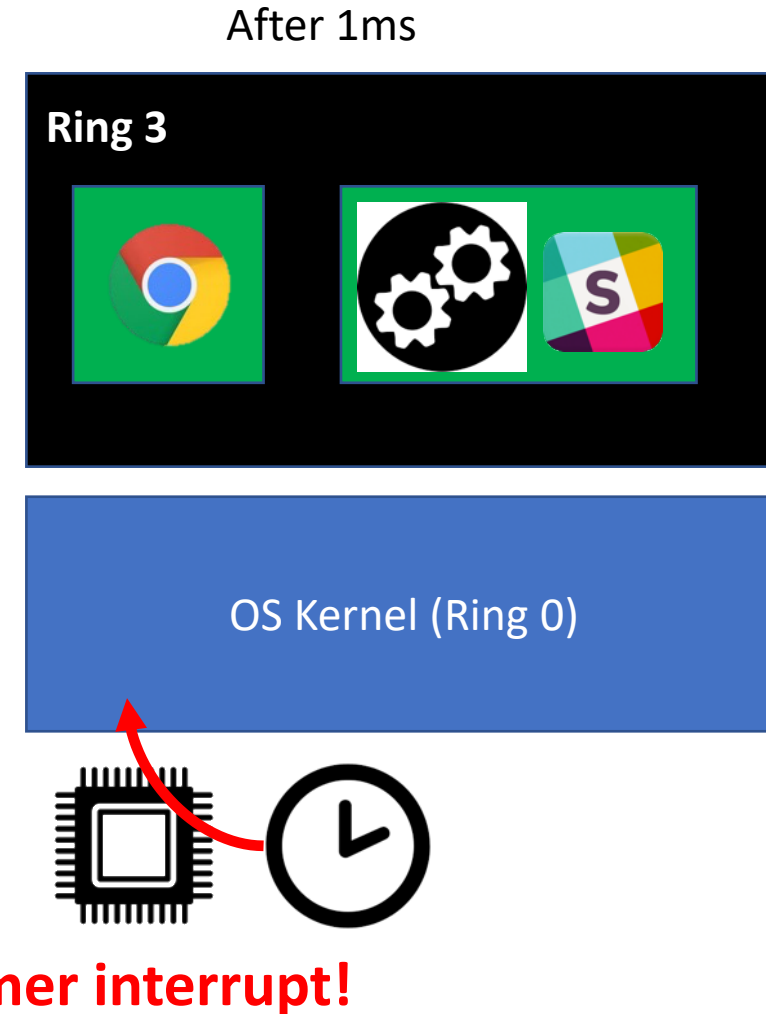
```
int main() {  
    while(1);  
}
```

Too long



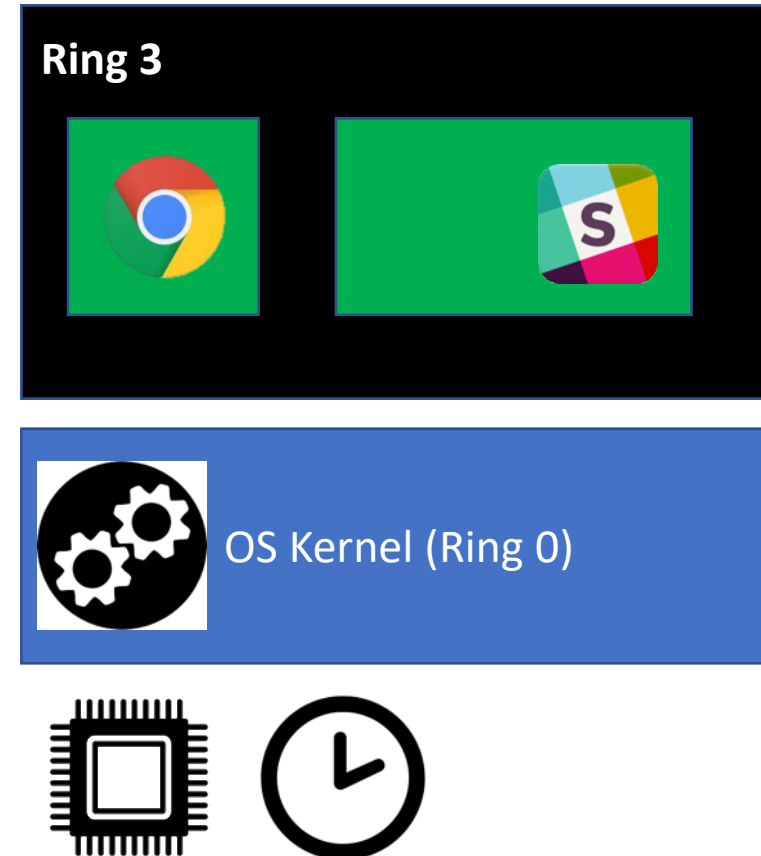
# Recap: Timer Interrupt and Multitasking

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..



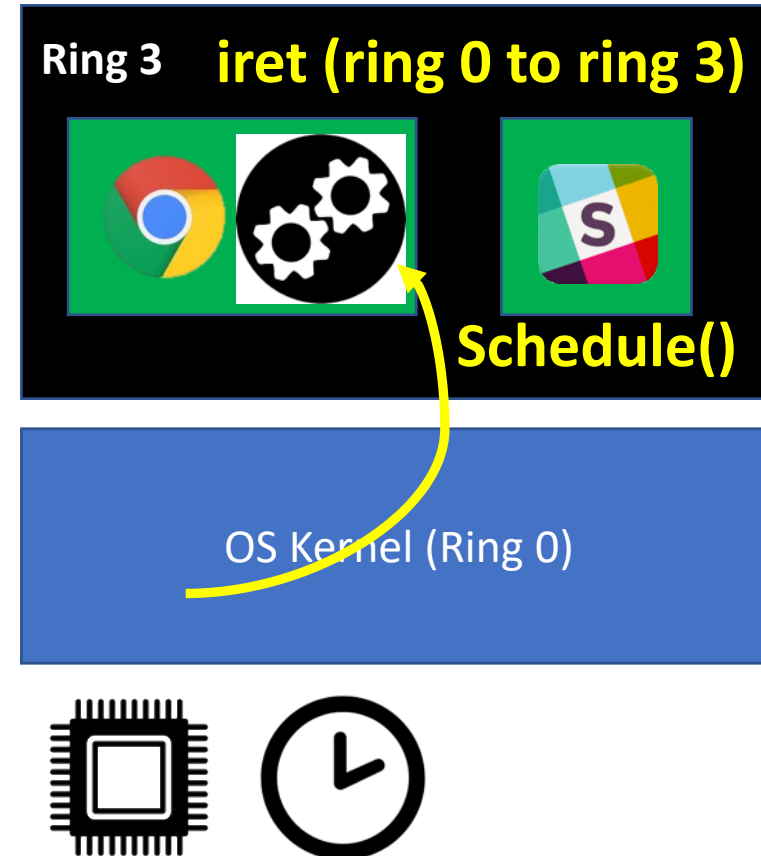
# Recap: Timer Interrupt and Multitasking

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



# Recap: Timer Interrupt and Multitasking

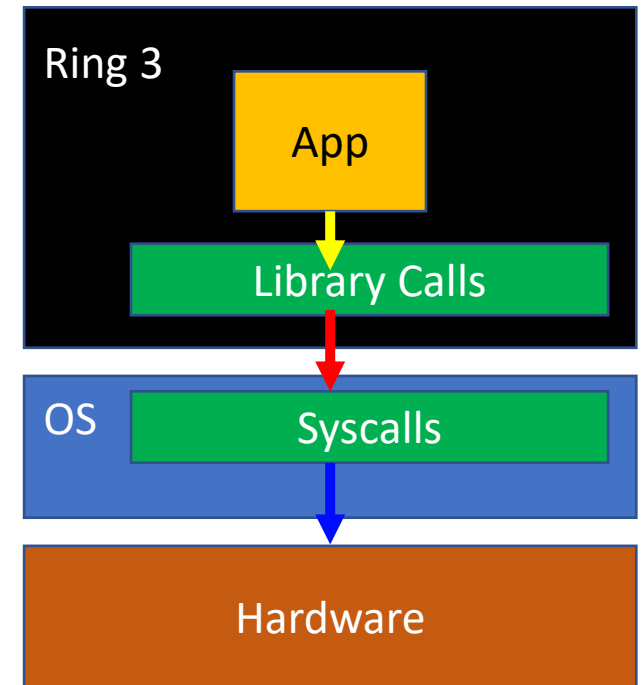
- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



# User/Kernel Switch

- System call
  - User calls Kernel APIs
  - Kernel mediates API access (checks legitimacy at call gate)
- How switch?
  - At the call gate, store trap frame
    - Stores all registers, and other states
  - On returning to user (iret)
    - Restore all information from trap frame

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





# User/Kernel Switch

- Interrupts
  - Could come from hardware (when it is not a software interrupt)
  - Think about the timer interrupt
    - Let OS do context switch!
- Steps
  - Stops current process (stores trapframe)
  - Runs kernel for handling the interrupt (refer to IDT)
  - Resumes previous (or new) process (iret)

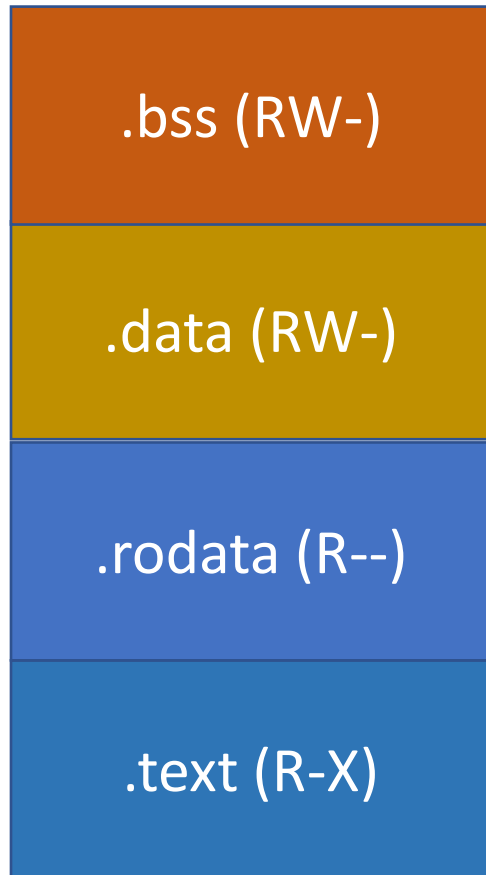
# Faults

- An error that OS can recover
- Example
  - Page fault
    - Copy-on-write fork
    - Memory Swapping

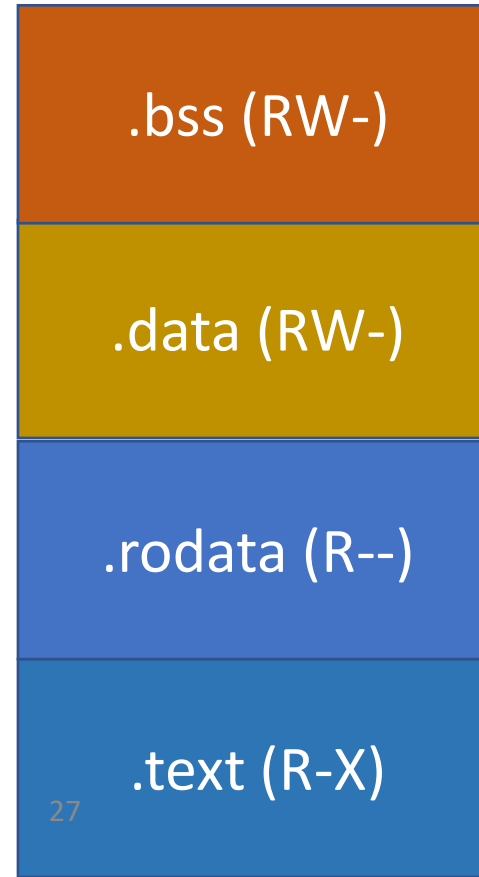
# Copy-on-Write Sharing

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

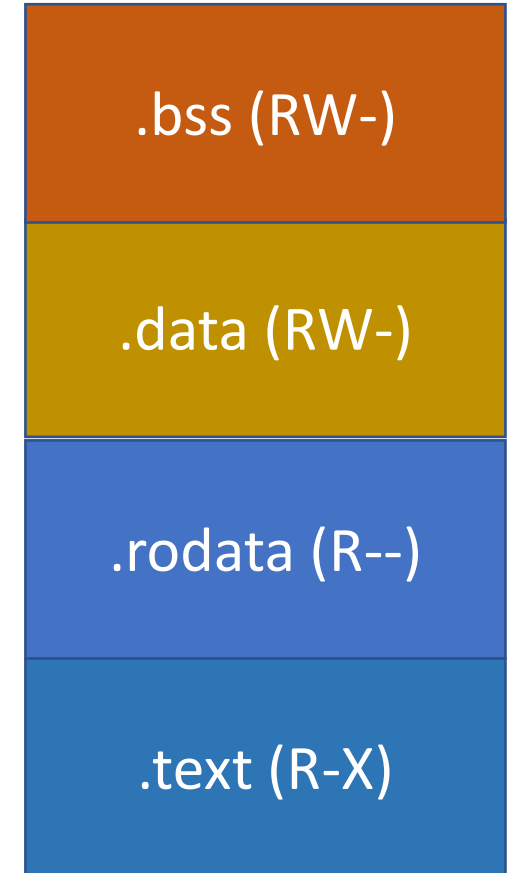
- Store one copy of file in the memory



Process 1

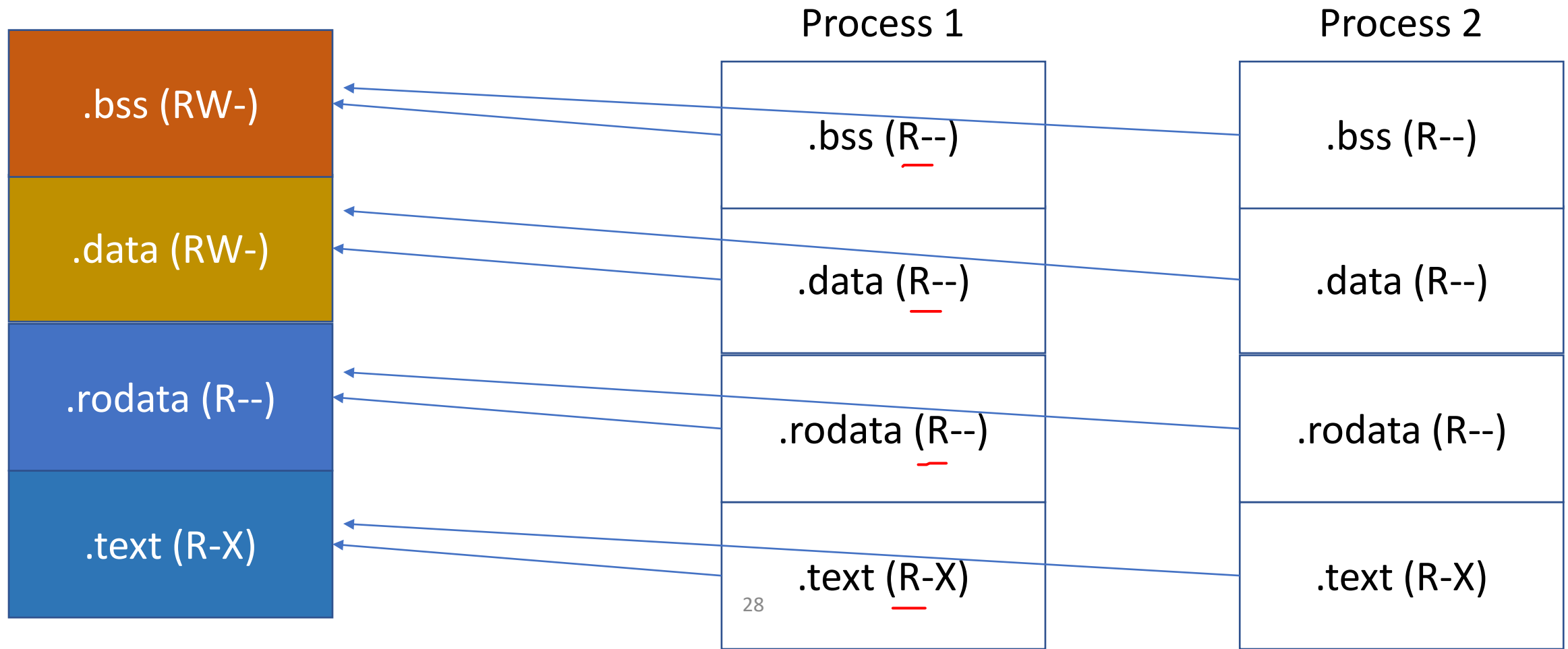


Process 2



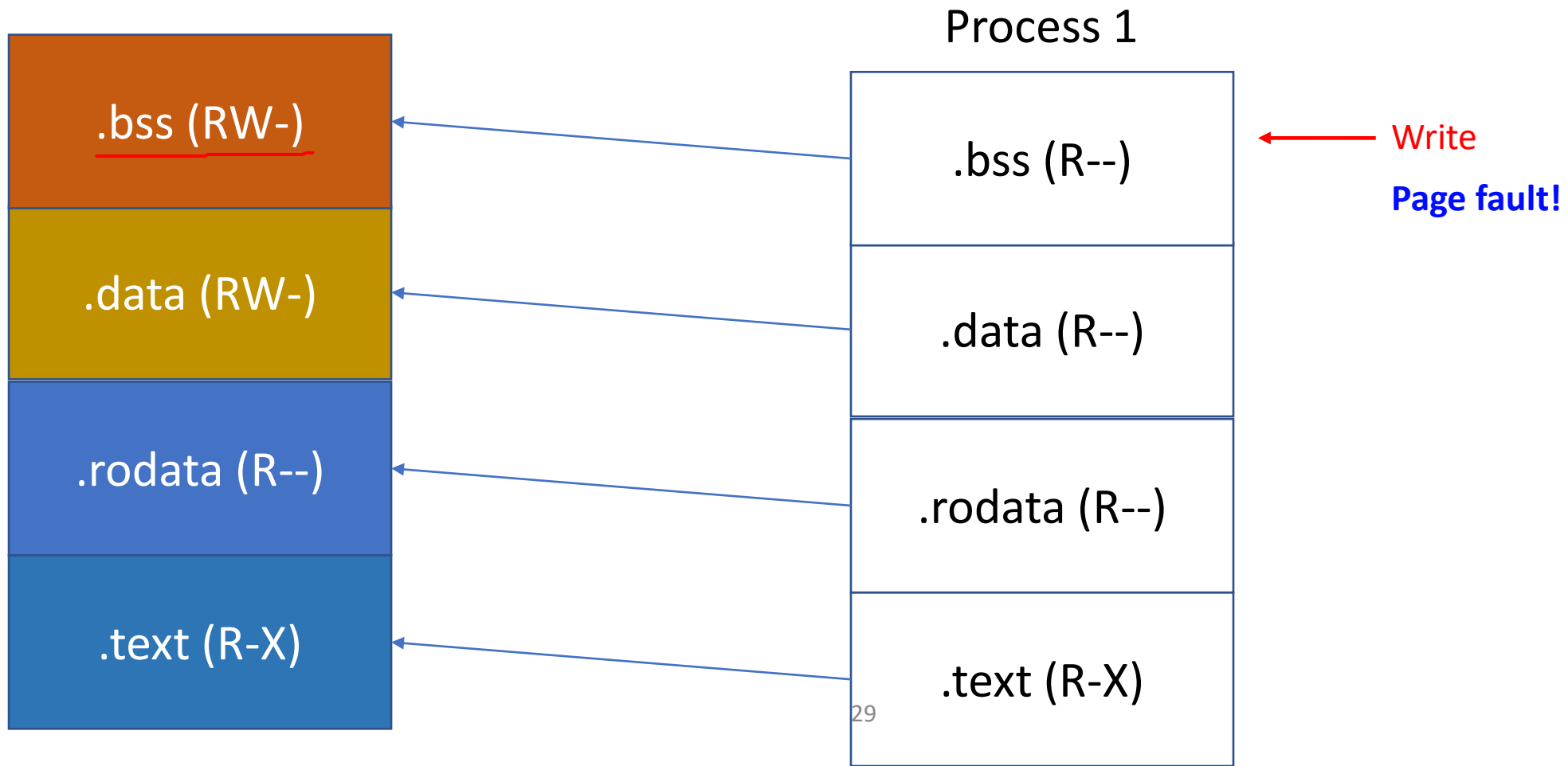
# Sharing by Read-only

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



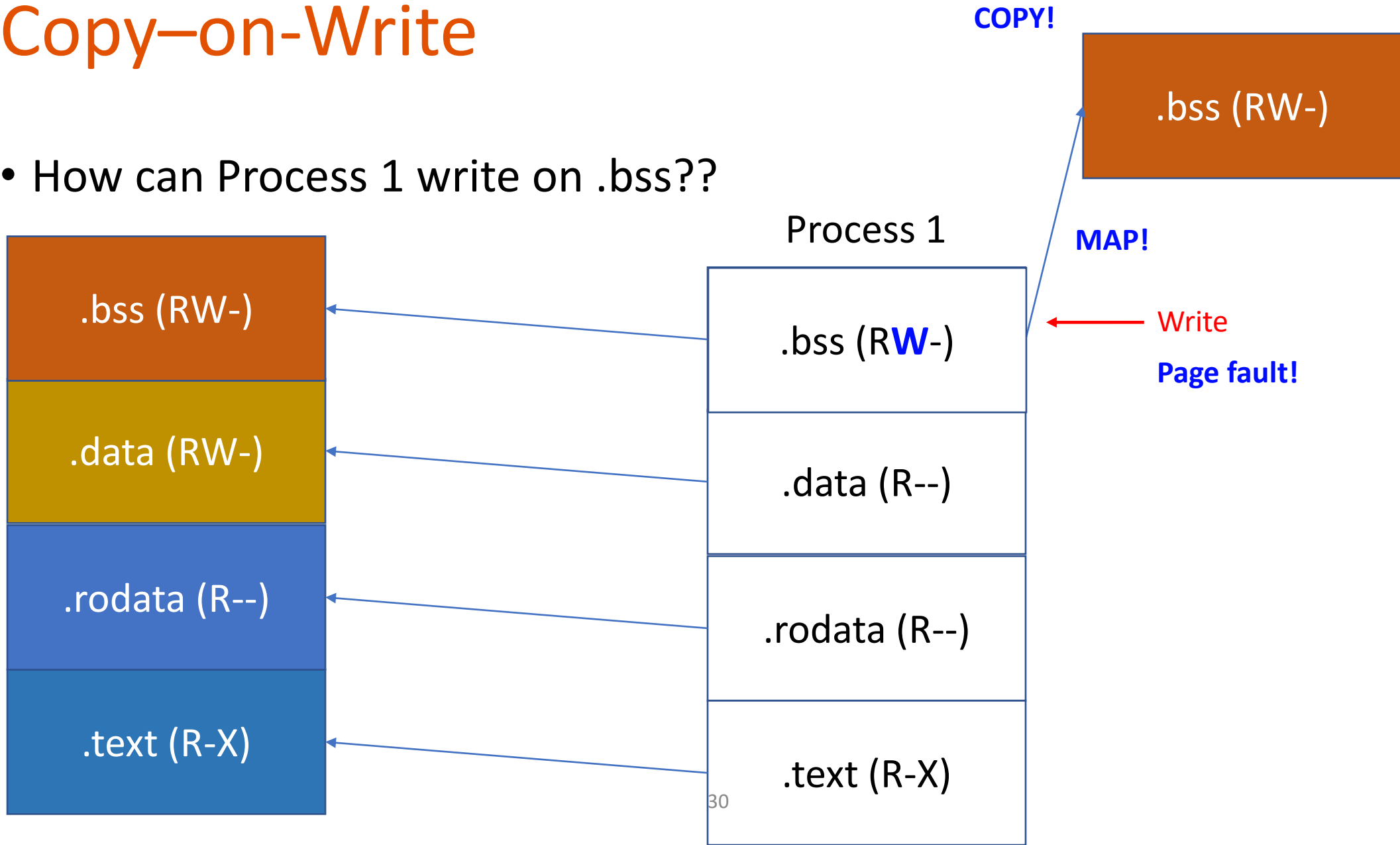
# Generate a Page Fault on Writing Attempt

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



# Copy-on-Write

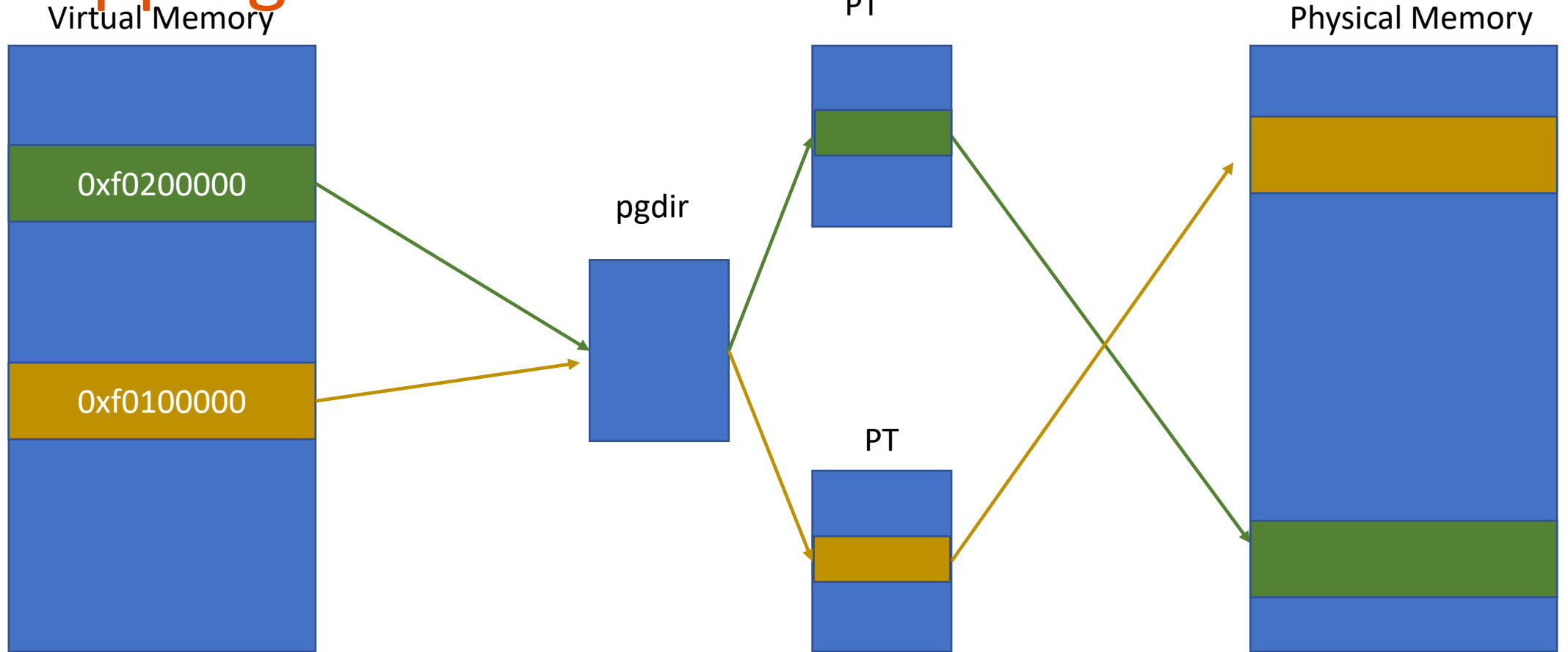
- How can Process 1 write on .bss??



# A Challenge of Having Small Physical Memory

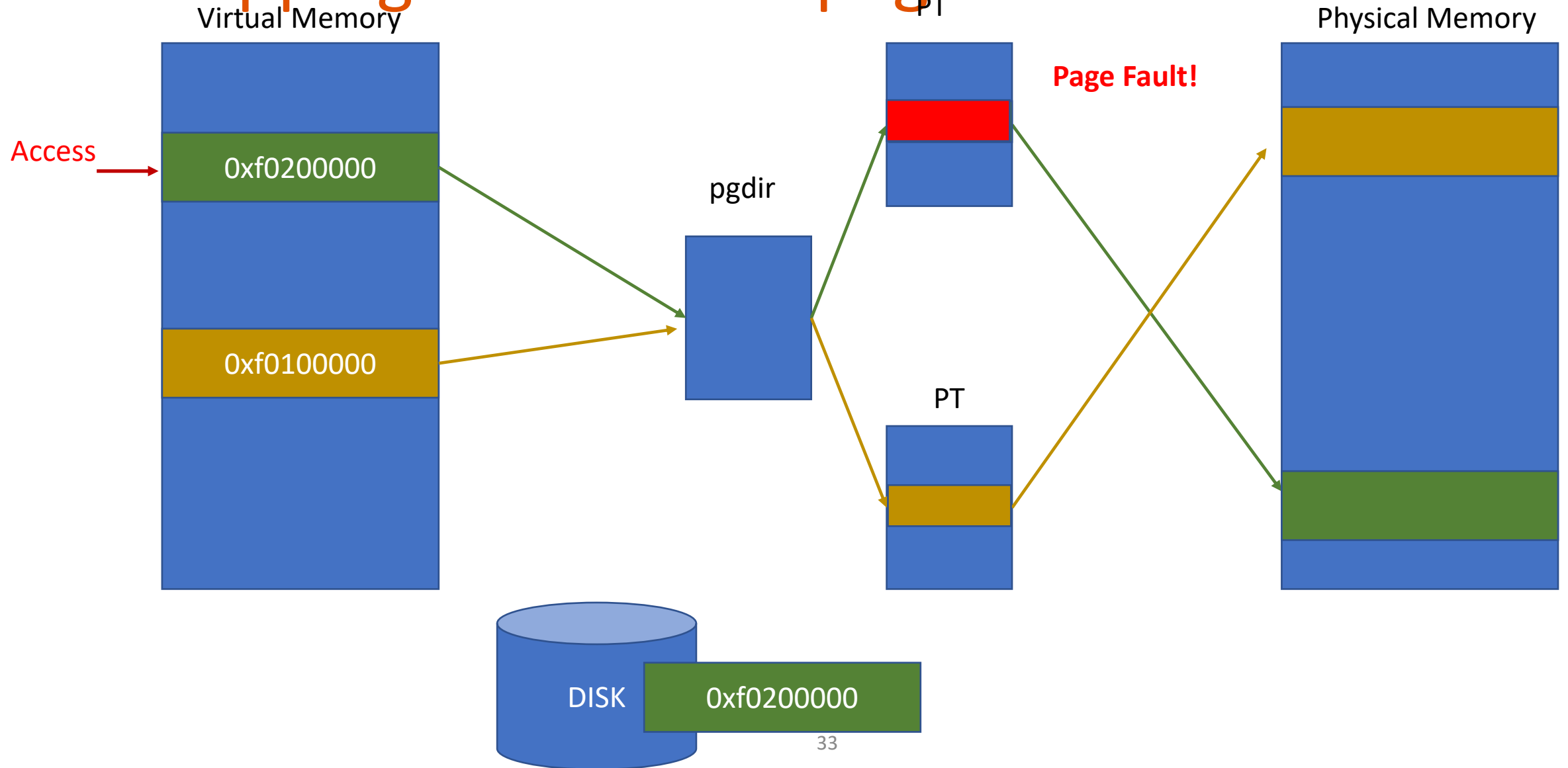
- Suppose you have 8GB of main memory
- Can you run a program that its program size is 16GB?
  - Yes, you can load them part by part
  - This is because we do not use all of data at the same time
- Can your OS do this execution seamlessly to your application?

# Swapping

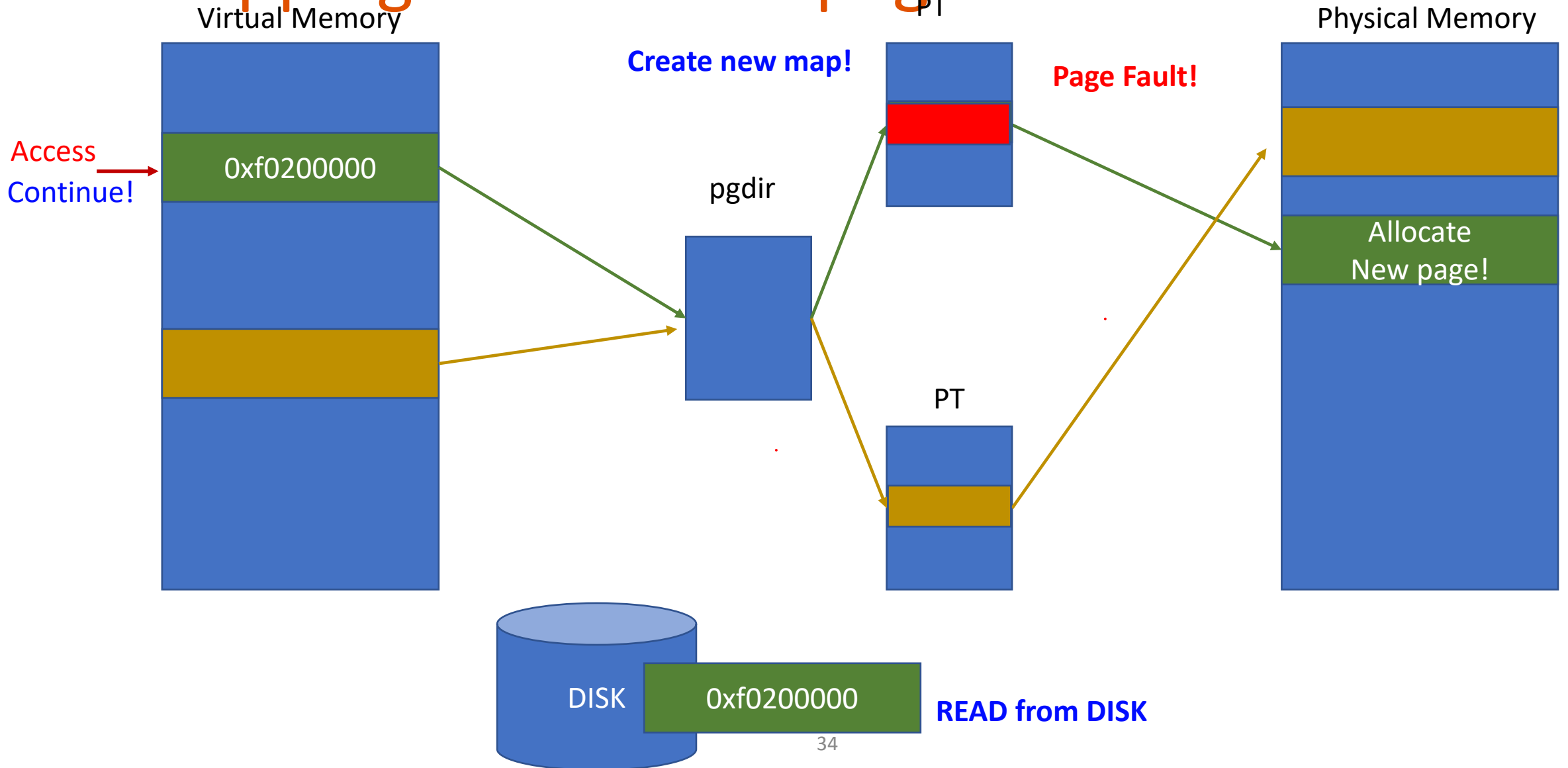




# Swapping – Remove a page...



# Swapping – Remove a page...



# Quiz 2

- Monday (5/13 from 8:00am to 11:59pm, 90 mins, 2 attempts)
  - Open materials (slides, videos, code, and textbook)
- You will have 2 attempts for the quiz

# Quiz 2 Coverage

- JOS Lab 2 (Virtual Memory Management)
- JOS Lab 3 (User/Kernel, System Call and Interrupt Handling)
- Lecture 8: User/Kernel Context Switch
- Lecture 9: Handling Interrupt & Exceptions
- Lecture 10: System Calls, Call Gate, and Page Fault
- Lecture 11: Virtualization Review

# Example Questions

- Memory Protection
  - How does an OS/CPU apply access control to the virtual memory system?
    - Protected mode (DPL), Page directory / page table (permission flags, PTE\_W & PTE\_U)
  - How does an OS protect itself against the attacks from application code?
    - Removing PTE\_U from PDE or PTE
  - How does an OS protect the memory area which is supposed to be read-only from write attempts?
    - Removing PTE\_W from PDE or PTE
  - How does an OS isolate the memory space of a process from others?
    - Having a new page directory / page tables

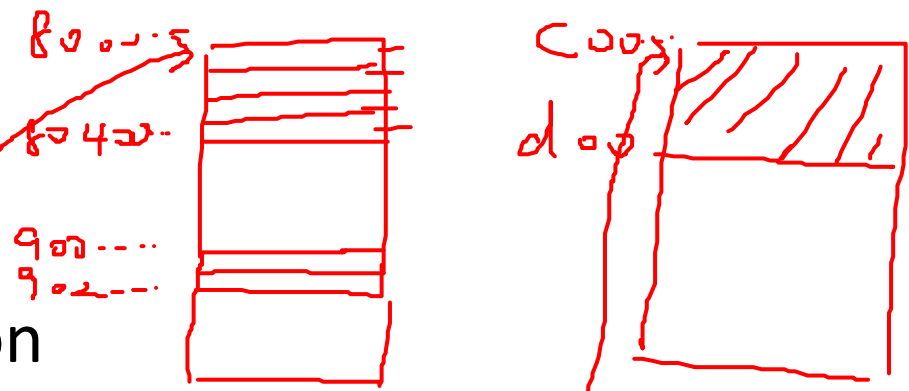
# Example Questions

- Memory Overhead Calculation
  - We have the following mapping for a program. How much of physical memory is required to support virtual to physical address translation for this program (get the minimal total size of 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

# Example 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	<u>0x100000</u>

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

0x11000000

1 page directory: 4KB

2 page tables: 2 \* 4KB = 8KB

4KB + 8KB = 12KB

# Example Questions

- User / Kernel Switch
  - How does an OS get back the execution from the user while the user runs `while(1);`?
    - Timer interrupt will preempt the execution from user to kernel
  - How does the user program access the hardware? What's the job of the OS in this process?
    - 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.



# Example Questions

- For an interrupt that has an error code,
- Which part of TrapFrame is prepared by the CPU?
  - `tf_ss`, `tf_esp`, `tf_eflags`, `tf_cs`, `tf_eip` and `tf_err`
- Which part of TrapFrame is prepared by JOS?
  - All others: `tf_trapno`, `tf_ds`, `tf_es`, `tf_regs`

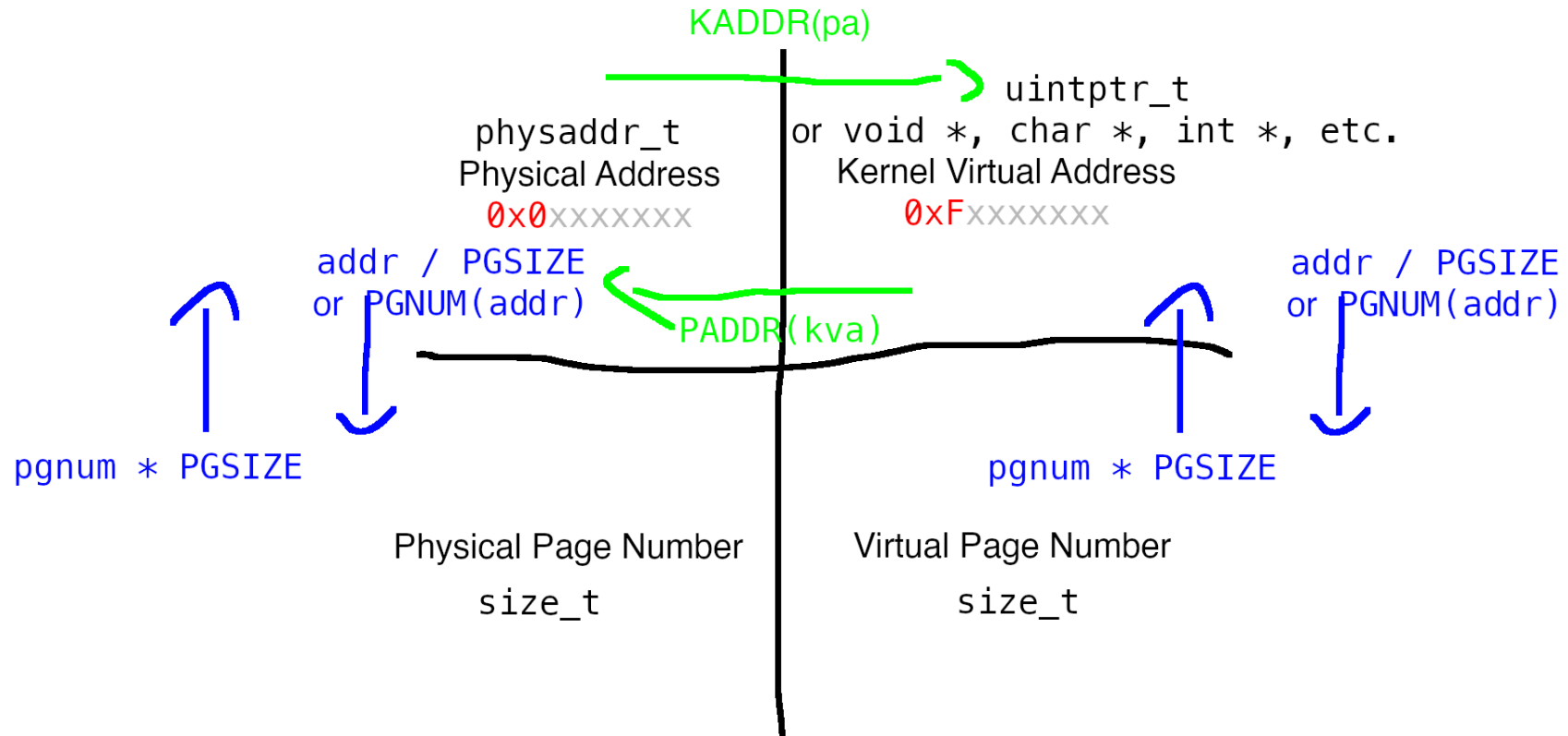
```
+-----+ KSTACKTOP
| 0x000000 | old SS   |      " - 4
|         | old ESP  |      " - 8
|         | old EFLAGS |     " - 12
| 0x000000 | old CS   |     " - 16
|         | old EIP  |     " - 20
|         | error code |     " - 24 <----- ESP
+-----+
```

```
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));
```

# Example Questions

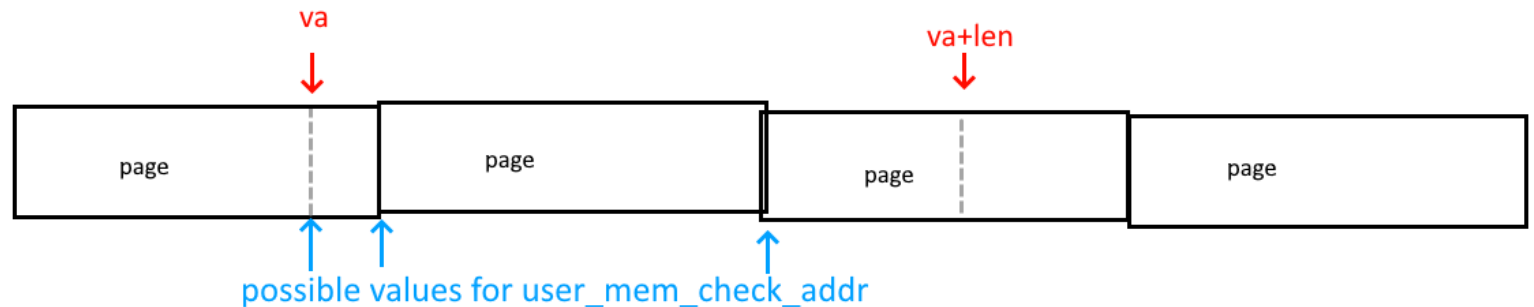
- Page Fault
  - We run 1,000,000 instances of /bin/bash in our os2 server, running Linux enabled with copy-on-write fork(). How many copies of the code (the read-only part) of /bin/bash exist in the physical memory?
    - One, shared via copy-on-write
  - How can an OS run a program that requires more memory than a machine's physical memory?
    - We can 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!

# Tips for Lab 3



# Tips for Lab 3 weird edge cases with user\_mem\_check

user\_mem\_check\_addr must be the FIRST invalid address in the range [va, va+len]



Because all addresses in a given page have the SAME permissions, an address can only go from valid to invalid at the start of a page.

This means that the FIRST invalid address must either be:

- The first address in a given page (i.e. page-aligned)
- **va** itself (for when the first page is invalid) (not necessarily page-aligned)

Remember that user\_mem\_check\_addr must still be within the [va, va+len] range!