More System calls and Page faults

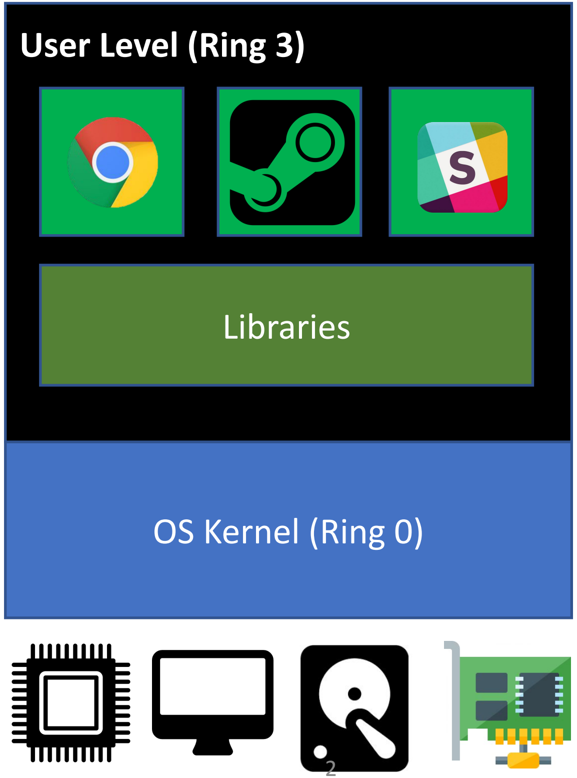
ECE 469, Feb 13

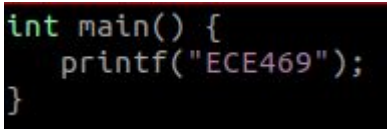
Aravind Machiry

Recap: Syscalls

- An API of an OS
- User-level Application calls functions in kernel
  - Open
  - Read
  - Write
  - Exec
  - Send
  - Recv
  - Socket
  - Etc...

Syscall: User/Kernel communication



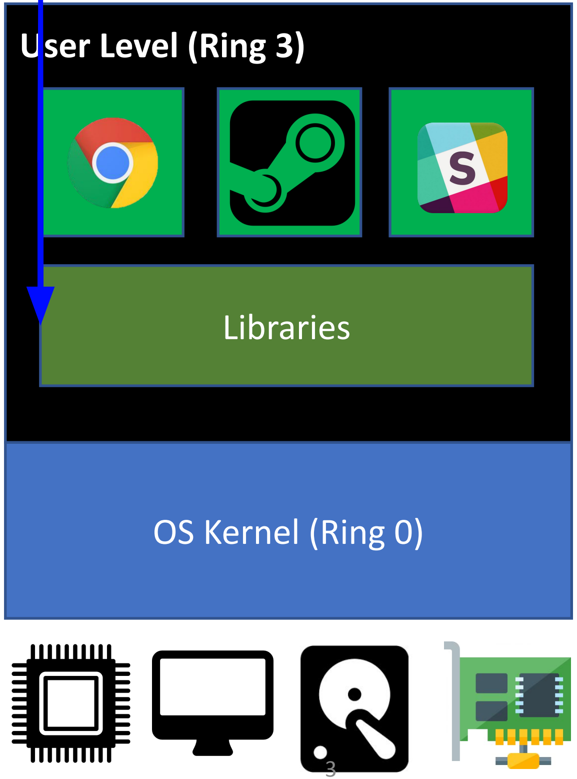


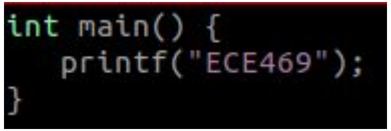
```
int main() {
   printf("ECE469");
}
```

Syscall: User/Kernel communication

printf("ECE469")

A library call in ring 3





```
int main() {
   printf("ECE469");
}
```

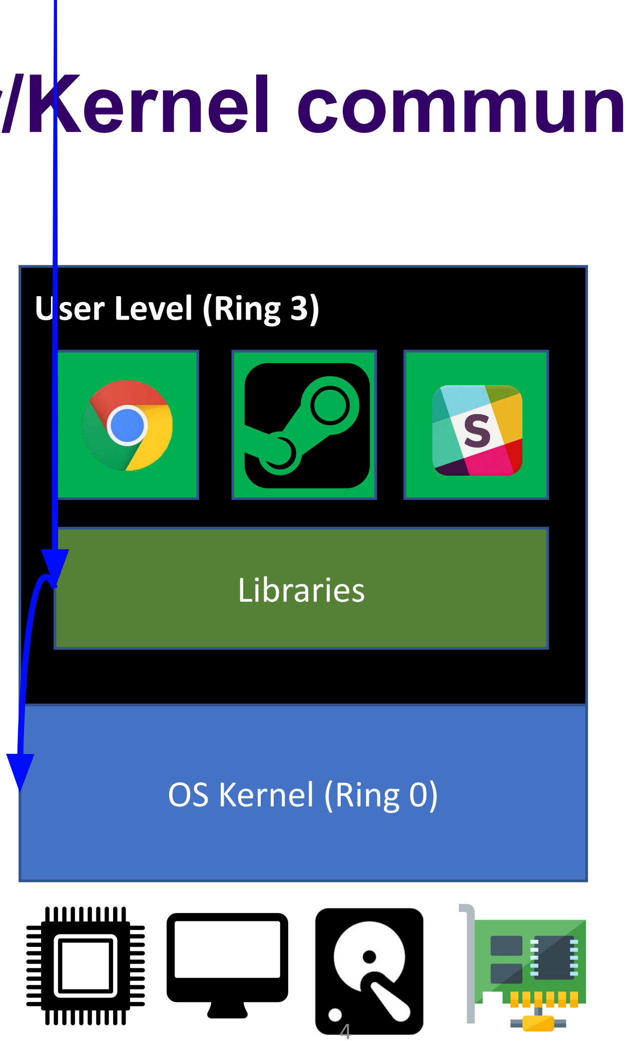
Syscall: User/Kernel communication

printf(“ECE469”)

A library call in ring 3

sys_write(1, “ECE469”, 6);

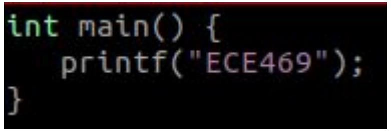
A system call, **From ring 3**



User Level (Ring 3)

Libraries

OS Kernel (Ring 0)



```
int main() {
   printf("ECE469");
}
```

Syscall: User/Kernel communication

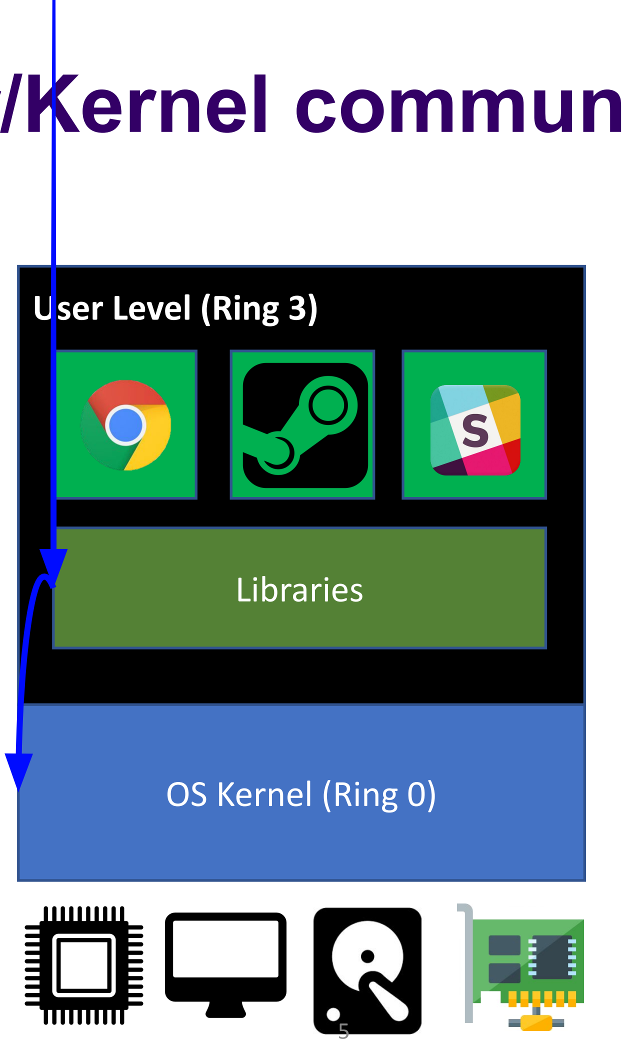
printf(“ECE469”)

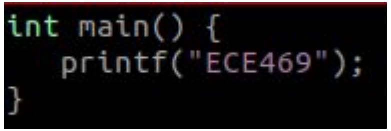
A library call in ring 3

sys_write(1, “ECE469”, 6);

A system call, **From ring 3**

Interrupt!, switch from ring3 to ring0





```
int main() {
    printf("ECE469");
}
```

Syscall: User/Kernel communication

printf(“ECE469”)

A library call in ring 3

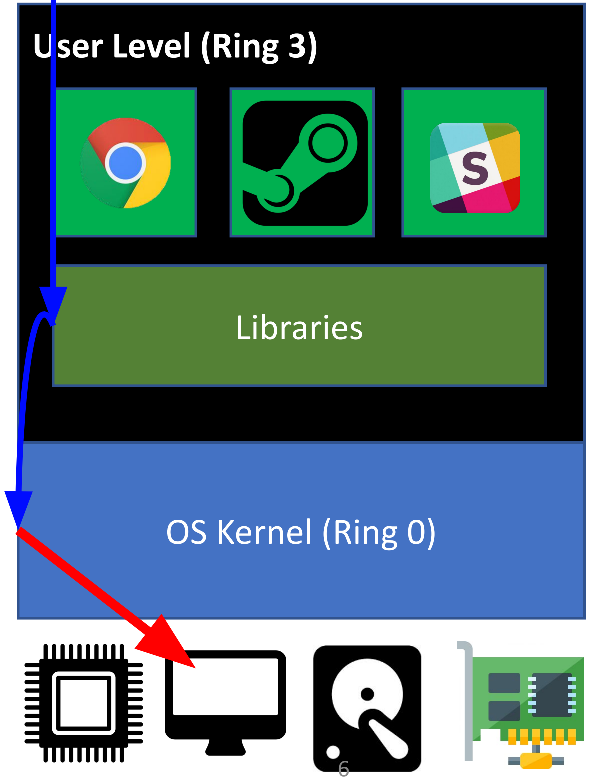
sys_write(1, “ECE469”, 6);

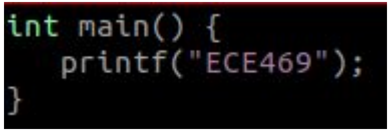
A system call, **From ring 3**

Interrupt!, switch from ring3 to ring0

A kernel function

do_sys_write(1, “ECE469”, 6)





```
int main() {
    printf("ECE469");
}
```

Syscall: User/Kernel communication

printf("ECE469")

A library call in ring 3

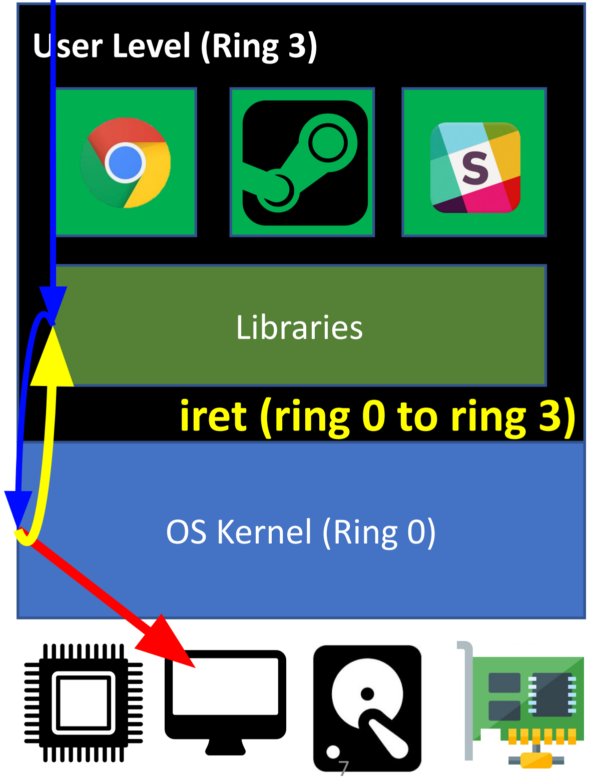
sys_write(1, "ECE469", 6);

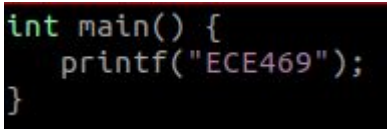
A system call, **From ring 3**

Interrupt!, switch from ring3 to ring0

A kernel function

do_sys_write(1, "ECE469", 6)





```
int main() {
   printf("ECE469");
}
```

Syscall: User/Kernel communication

printf(“ECE469”)

A library call in ring 3

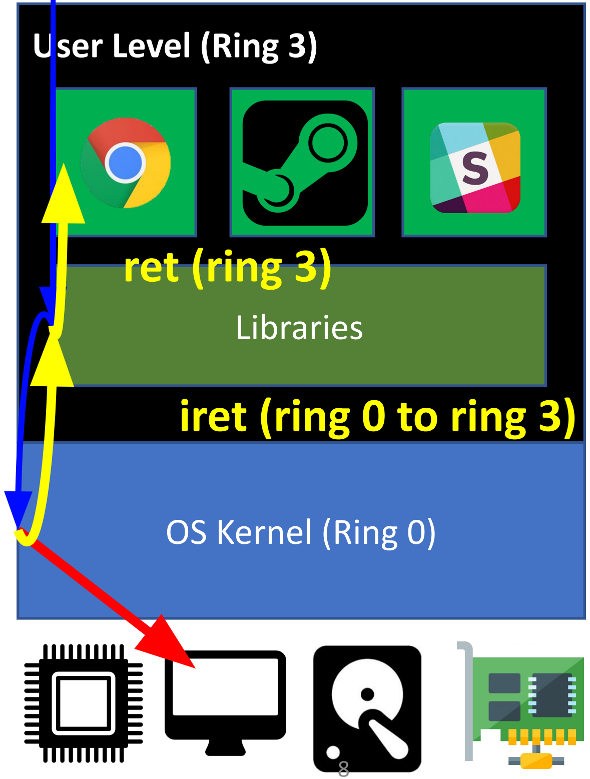
sys_write(1, “ECE469”, 6);

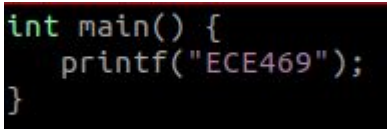
A system call, **From ring 3**

Interrupt!, switch from ring3 to ring0

A kernel function

do_sys_write(1, “ECE469”, 6)

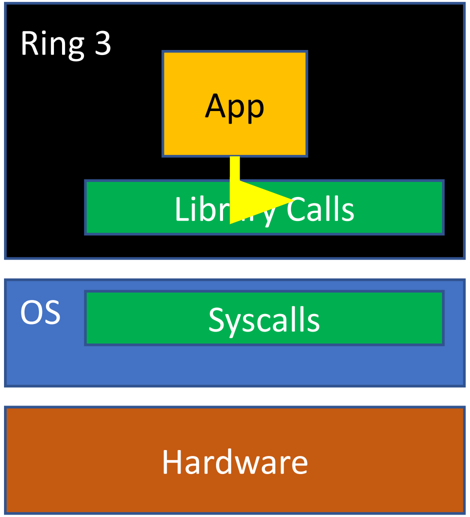




```
int main() {
   printf("ECE469");
}
```

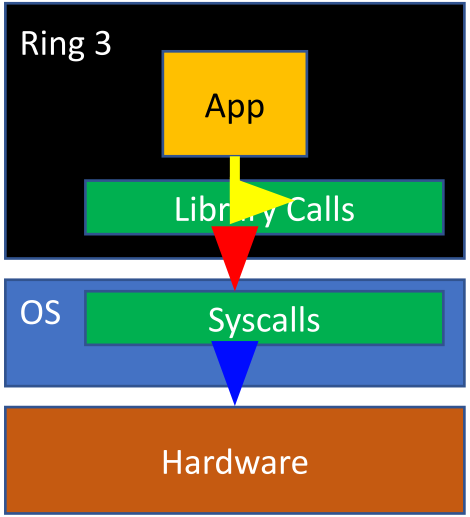
System calls via Interrupt Handler

- 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 userspace is doing a right thing
  - Before performing important ring 0 operations
  - E.g., accessing hardware..



System calls via Interrupt Handler

- 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)



`int $0x30`

`CHECK!!`

- OS performs checks if userspace is doing a right thing
  - Before performing important ring 0 operations
  - E.g., accessing hardware..

Implementing Syscalls in JOS

- See kern/syscall.c
- `void sys_cputs(const char *s, size_t len)`
  - Print a string in s to the console
- `int sys_cgetc(void)`
  - Get a character from the keyboard
- `envid_t sys_getenvid(void)`
  - Get the current environment ID (process ID)
- `int sys_env_destroy(envid_t)`
  - Kill the current environment (process)

Implementing Syscalls in JOS

- See kern/syscall.c
- `void sys_cputs(const char *s, size_t len)`
  - Print a string in s to the console
- `int sys_cgetc(void)`
  - Get a character from the keyboard
- `envid_t sys_getenvid(void)`
  - Get the current environment ID (process ID)
- `int sys_env_destroy(envid_t)`
  - Kill the current environment (process)

Required for Implementing scanf, printf, etc…

Passing arguments to Syscalls

- How can we pass arguments to syscalls?
  - Remember syscalls are implemented as interrupts!

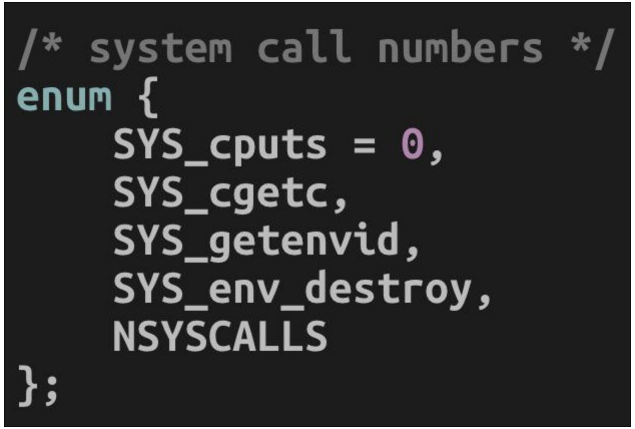
Passing arguments to Syscalls

- How can we pass arguments to syscalls?
    - Remember syscalls are implemented as interrupts!

General Purpose Registers!!!

Passing arguments to Syscalls

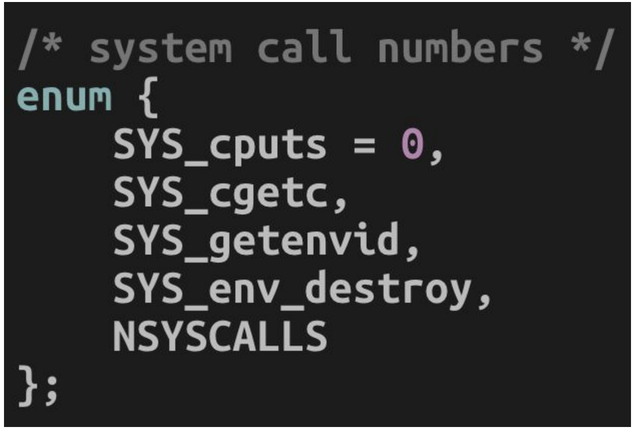
- In JOS
  - eax = system call number
  - edx = 1st argument
  - ecx = 2nd argument
  - ebx = 3rd argument
  - edi = 4th argument
  - esi = 5th argument
- E.g., calling sys_cputs(“asdf”, 4);
  - eax = 0
  - edx = address of “asdf”
  - ecx = 4
  - ebx, edi, esi = not used
- And then
  - Run `int $0x30`



```
/* system call numbers */
enum {
	SYS_cputs = 0,
	SYS_cgetc,
	SYS_getenvid,
	SYS_env_destroy,
	NSYSCALLS
};
```

Passing arguments to Syscalls

- In JOS
  - eax = system call number
  - edx = 1st argument
  - ecx = 2nd argument
  - ebx = 3rd argument
  - edi = 4th argument
  - esi = 5th argument
- E.g., calling sys_cputs(“asdf”, 4);
  - eax = 0
  - edx = address of “asdf”
  - ecx = 4
  - ebx, edi, esi = not used
- And then
  - Run `int $0x30`

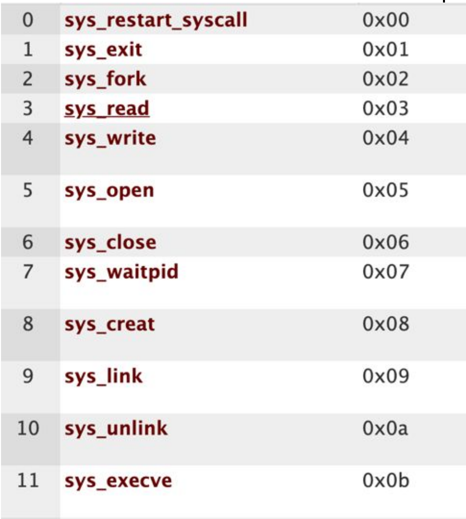


```
/* system call numbers */
enum {
	SYS_cputs = 0,
	SYS_cgetc,
	SYS_getenvid,
	SYS_env_destroy,
	NSYSCALLS
};
```

Will add more as
our lab implementation progresses

Passing arguments to Syscalls

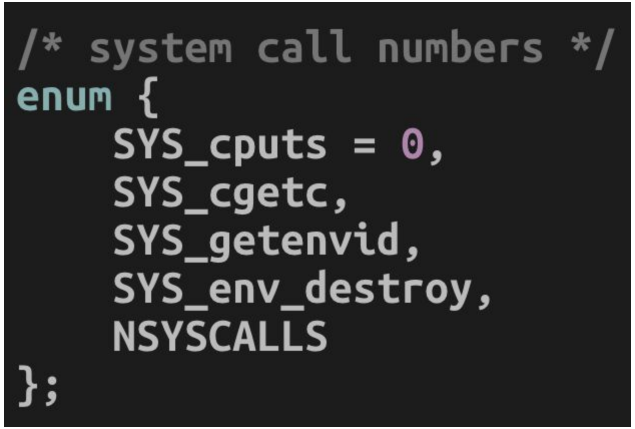
- In Linux x86 (32-bit)
  - eax = system call number
  - ebx = 1st argument
  - ecx = 2nd argument
  - edx = 3rd argument
  - esi = 4th argument
  - edi = 5th argument
- See table
  - https://syscalls.kernelgrok.com/ : lists 337 system calls…



0	sys_restart_syscall	0x00
1	sys_exit	0x01
2	sys_fork	0x02
3	sys_read	0x03
4	sys_write	0x04
5	sys_open	0x05
6	sys_close	0x06
7	sys_waitpid	0x07
8	sys_creat	0x08
9	sys_link	0x09
10	sys_unlink	0x0a
11	sys_execve	0x0b

Handling arguments to Syscalls

- E.g., calling sys_cputs(“asdf”, 4);
  - eax = 0
  - edx = address of “asdf”
  - ecx = 4
  - ebx, edi, esi = not used
- And then
  - Run `int $0x30`
- At interrupt handler
  - Read syscall number from the eax of tf
    - syscall number is 0 -> calling SYS_cputs
  - Read 1st argument from the edx of tf
    - Address of “adsf”
  - Read 2nd argument from ecx of tf
    - 4
  - call sys_cputs(“asdf”, 4) // in kernel



```
/* system call numbers */
enum {
	SYS_cputs = 0,
	SYS_cgetc,
	SYS_getenvid,
	SYS_env_destroy,
	NSYSCALLS
};
```

Invoking Syscalls

- Set all arguments in the registers
    - Order: edx ecx ebx edi esi

- int $0x30 (in JOS)
    - Software interrupt 48

- int $0x80 (in 32bit Linux)
    - Software interrupt 128

Invoking Syscalls in User mode

- User calls a function

Invoking Syscalls in User mode

- User calls a function
  - cprintf -> calls sys_cputs()

Invoking Syscalls in User mode

- User calls a function
  - cprintf -> calls sys_cputs()
- sys_cputs() at user code will call syscall() (lib/syscall.c)
  - This syscall() is at lib/syscall.c

Invoking Syscalls in User mode

- User calls a function
  - cprintf -> calls sys_cputs()
- sys_cputs() at user code will call syscall() (lib/syscall.c)
  - This syscall() is at lib/syscall.c
  - Set args in the register and then

Invoking Syscalls in User mode

- User calls a function
  - cprintf -> calls sys_cputs()
- sys_cputs() at user code will call syscall() (lib/syscall.c)
  - This syscall() is at lib/syscall.c
  - Set args in the register and then

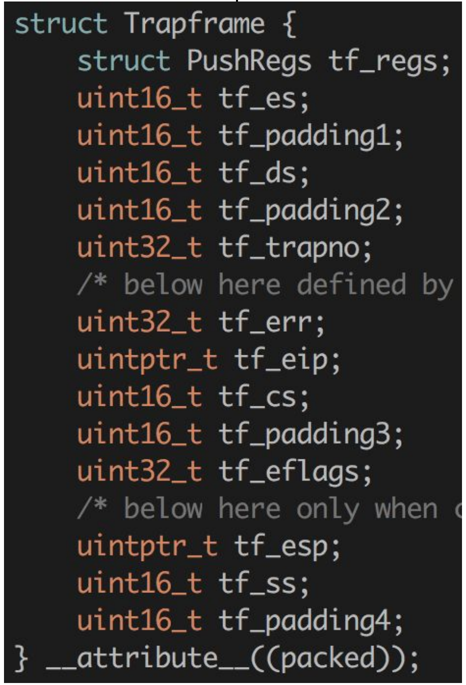
- int $0x30

Invoking Syscalls in User mode

- User calls a function
  - cprintf -> calls sys_cputs()
- sys_cputs() at user code will call syscall() (lib/syscall.c)
  - This syscall() is at lib/syscall.c
  - Set args in the register and then
- int $0x30
- Now kernel execution starts…

Invoking Syscalls in User mode

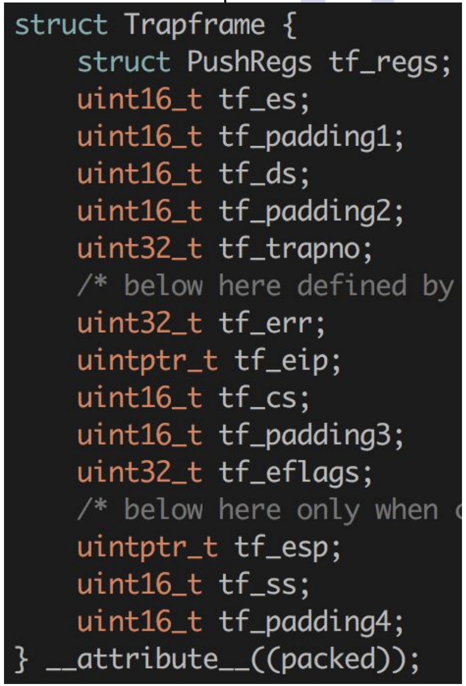
- User calls a function
  - cprintf -> calls sys_cputs()
- sys_cputs() at user code will call syscall() (lib/syscall.c)
  - This syscall() is at lib/syscall.c
  - Set args in the register and then
- int $0x30
- Now kernel execution starts…



```
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
	uintptr_t tf_esp;
	uint16_t tf_ss;
	uint16_t tf_padding4;
} __attribute__((packed));
```

Handling Syscalls in Kernel

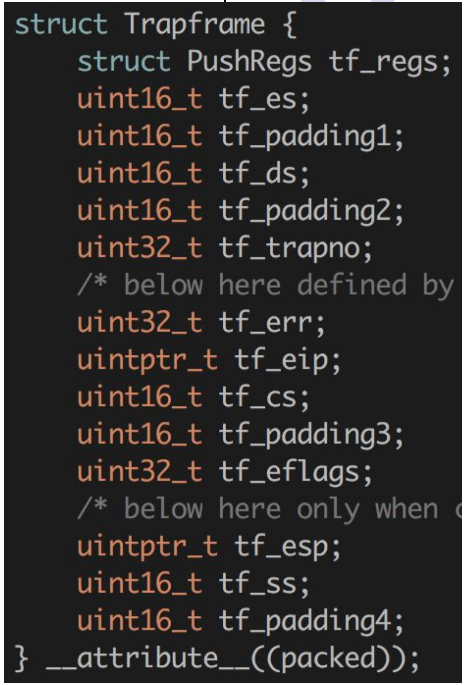
- CPU gets software interrupt



```
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
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Handling Syscalls in Kernel

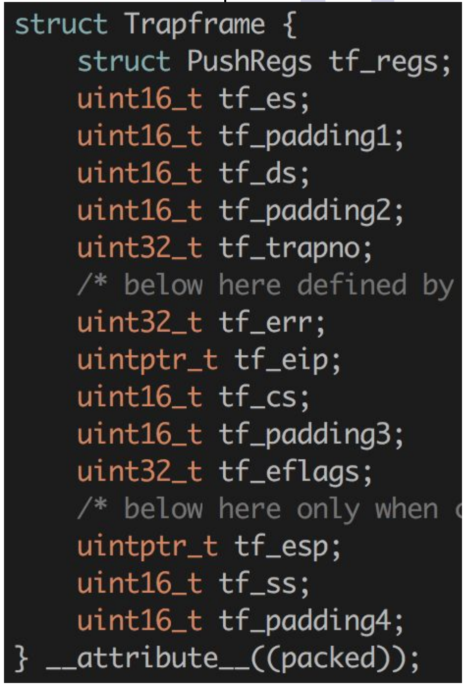
- CPU gets software interrupt
- TRAPHANDLER_NOEC(T_SYSCALL…)



```
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
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Handling Syscalls in Kernel

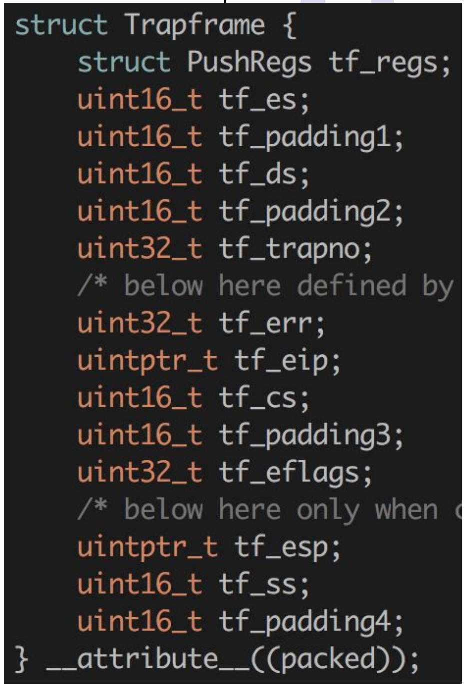
- CPU gets software interrupt
- TRAPHANDLER_NOEC(T_SYSCALL…)
- _alltraps()



```
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
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Handling Syscalls in Kernel

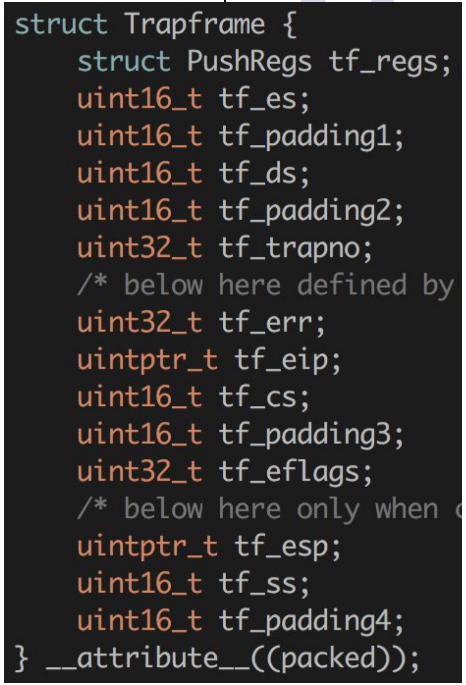
- CPU gets software interrupt
- TRAPHANDLER_NOEC(T_SYSCALL…)
- _alltraps()
- trap()



```
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
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Handling Syscalls in Kernel

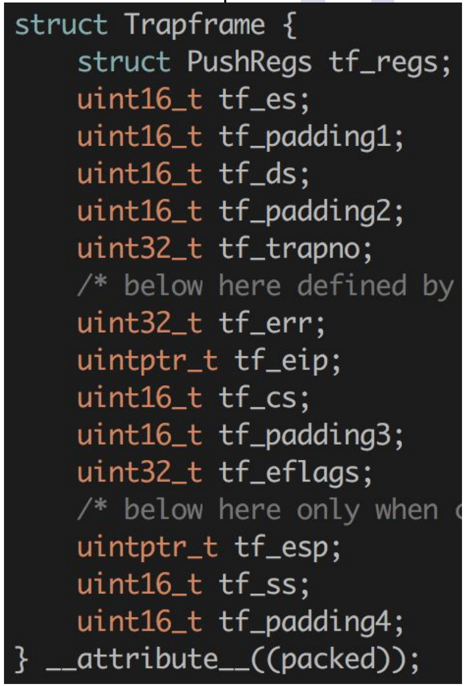
- CPU gets software interrupt
- TRAPHANDLER_NOEC(T_SYSCALL…)
- _alltraps()
- trap()
- trap_dispatch()



```
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));
```

Handling Syscalls in Kernel

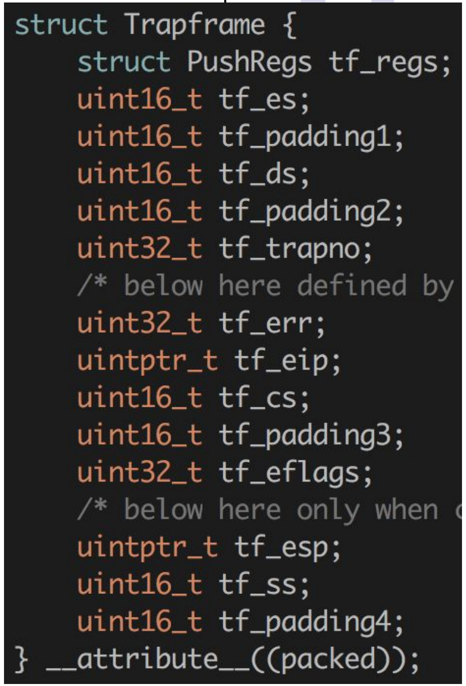
- CPU gets software interrupt
- TRAPHANDLER_NOEC(T_SYSCALL…)
- _alltraps()
- trap()
- trap_dispatch()
    - Get registers that store arguments from struct Trapframe *tf



```
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
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Handling Syscalls in Kernel

- CPU gets software interrupt
- TRAPHANDLER_NOEC(T_SYSCALL…)
- _alltraps()
- trap()
- trap_dispatch()
  - Get registers that store arguments from struct Trapframe *tf
  - Call syscall() using those registers
    - This syscall() is at kern/syscall.c



```
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
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Syscall Return from Kernel

- Finishing handling of syscall (return of syscall())

Syscall Return from Kernel

- Finishing handling of syscall (return of syscall())

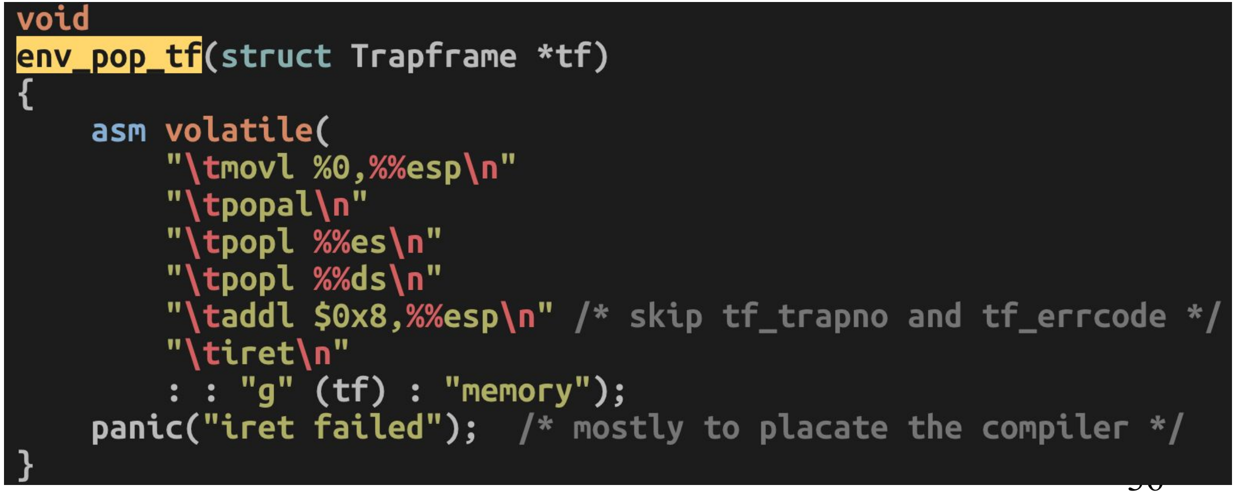
- trap() calls env_pop_tf()
  - Get back to the user environment!

Syscall Return from Kernel

- Finishing handling of syscall (return of syscall())

- trap() calls env_pop_tf()
  - Get back to the user environment!

- env_pop_tf()



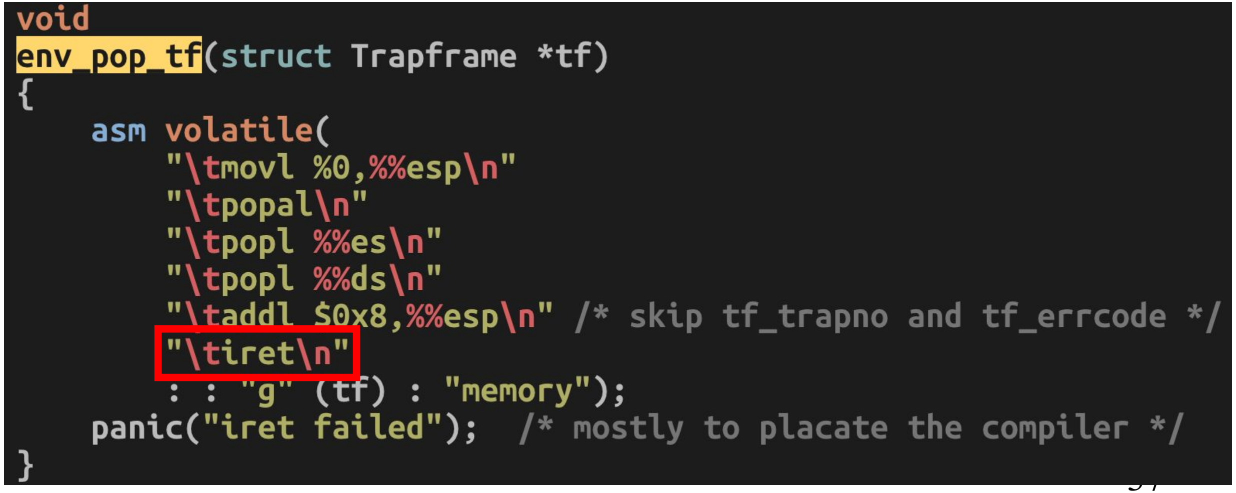
```
void
env_pop_tf(struct Trapframe *tf)
{
	asm volatile(
		"\tmovl %0,%%esp\n"
		"\tpopal\n"
		"\tpopl %%es\n"
		"\tpopl %%ds\n"
		"\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
		"\tiret\n"
		: : "g" (tf) : "memory");
	panic("iret failed");  /* mostly to placate the compiler */
}
```

Syscall Return from Kernel

- Finishing handling of syscall (return of syscall())

- trap() calls env_pop_tf()
  - Get back to the user environment!

- env_pop_tf()
  - Runs `iret`



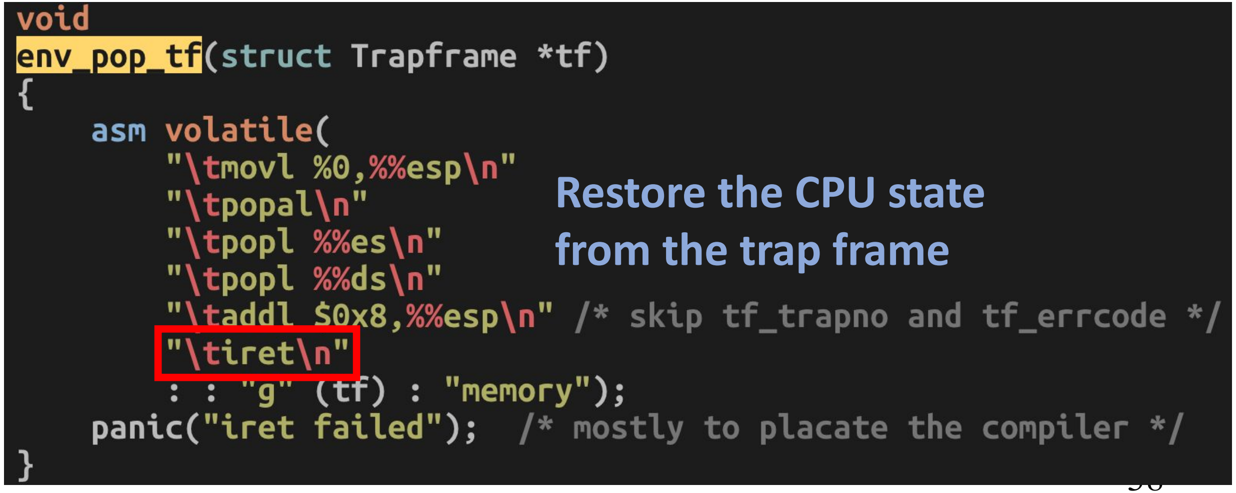
```
void
env_pop_tf(struct Trapframe *tf)
{
    asm volatile(
        "\tmovl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret\n"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

Syscall Return from Kernel

- Finishing handling of syscall (return of syscall())

- trap() calls env_pop_tf()
  - Get back to the user environment!

- env_pop_tf()
  - Runs `iret`

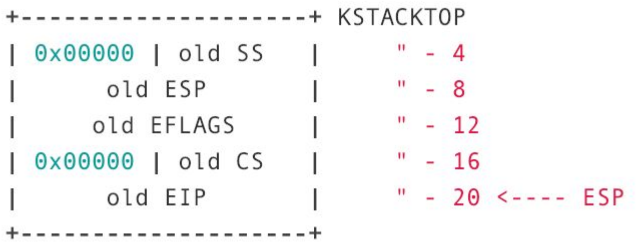


```
void
env_pop_tf(struct Trapframe *tf)
{
    asm volatile(
        "\tmovl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret\n"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

Restore the CPU state from the trap frame

Syscall Return from Kernel

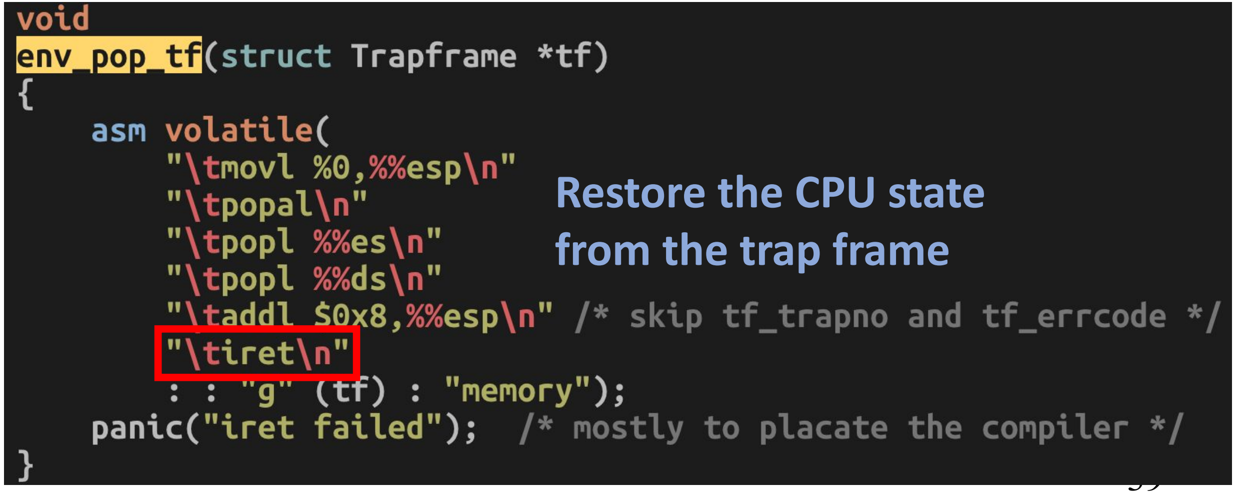
- Finishing handling of syscall (return of syscall())



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

- trap() calls env_pop_tf()
  - Get back to the user environment!

- env_pop_tf()
  - Runs `iret`

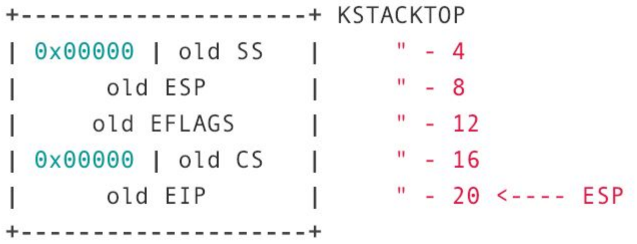


```
void
env_pop_tf(struct Trapframe *tf)
{
	asm volatile(
		"\tmovl %0,%%esp\n"
		"\tpopal\n"
		"\tpopl %%es\n"
		"\tpopl %%ds\n"
		"\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
		"\tiret\n"
		: : "g" (tf) : "memory");
	panic("iret failed");  /* mostly to placate the compiler */
}
```

Restore the CPU state from the trap frame

Syscall Return from Kernel

- Finishing handling of syscall (return of syscall())

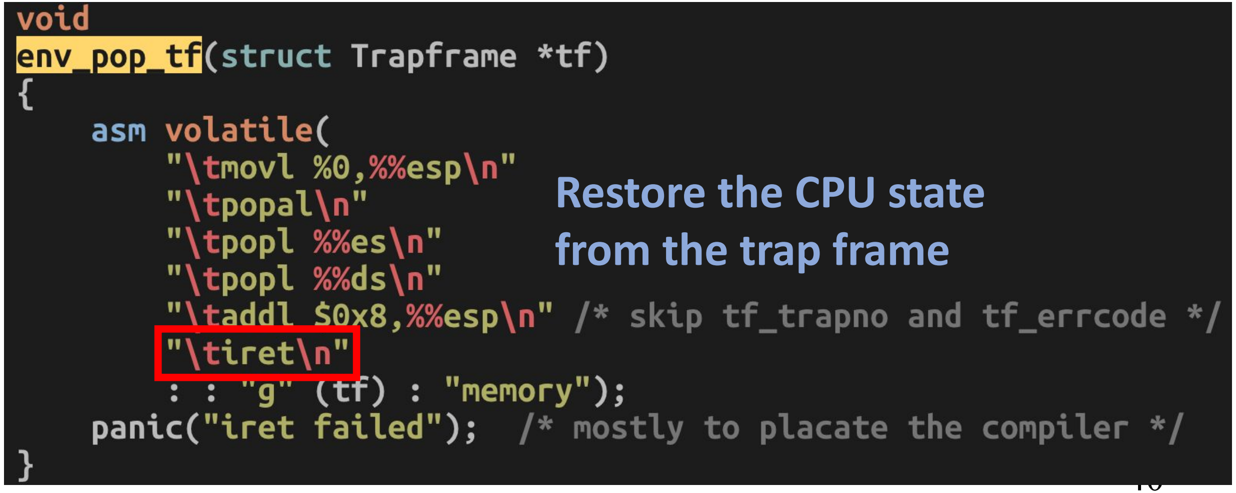


```
+--------------------+ KSTACKTOP
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+
```

- trap() calls env_pop_tf()
  - Get back to the user environment!

- env_pop_tf()
  - Runs `iret`

- Back to Ring 3!



```
void
env_pop_tf(struct Trapframe *tf)
{
	asm volatile(
		"\tmovl %0,%%esp\n"
		"\tpopal\n"
		"\tpopl %%es\n"
		"\tpopl %%ds\n"
		"\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
		"\tiret\n"
		: : "g" (tf) : "memory");
	panic("iret failed");  /* mostly to placate the compiler */
}
```

Restore the CPU state from the trap frame

System call Execution

- Execution…
- int $0x30
- Call trap gate
- Handle trap!
- Pop context
- iret
- Execution resumes…

System call Execution

- Execution…
- int $0x30

- Call trap gate

- Handle trap!

- Pop context

- iret

- Execution resumes…

Ring 3

System call Execution

- Execution…
- int $0x30

Ring 3

- Call trap gate
- Handle trap!

Ring 0

- Pop context
- iret
- Execution resumes…

System call Execution

- Execution…
- int $0x30

Ring 3

- Call trap gate
- Handle trap!
- Pop context
- iret

Ring 0

- Execution resumes…

Ring 3

Page faults

- Occurs when paging (address translation) fails
  - !(pde&PTE_P) or !(pte&PTE_P): Present bit not set
    - **Automated extension of runtime stack**
  - Write access but !(pte&PTE_W): access violation
  - Access from user but !(pte&PTE_U): protection violation

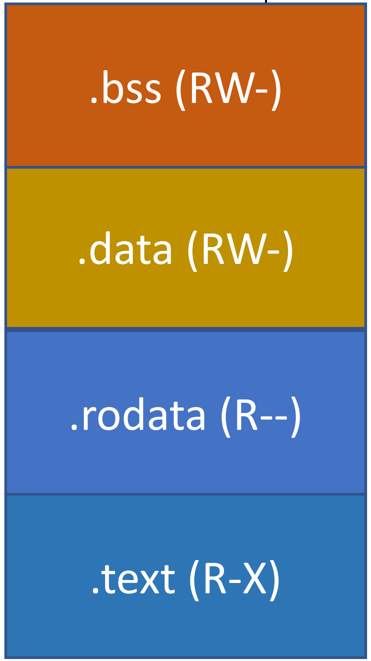
Page faults Handling (2): Copy-On-Write (CoW)

- 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

Program in Memory

- .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



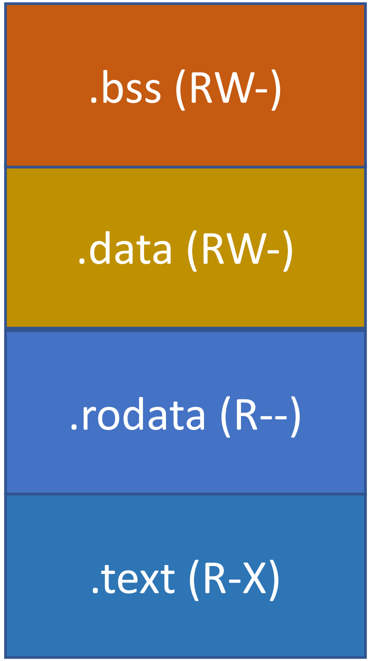
.bss (RW-)

.data (RW-)

.rodata (R--)

.text (R-X)

Running same program multiple times



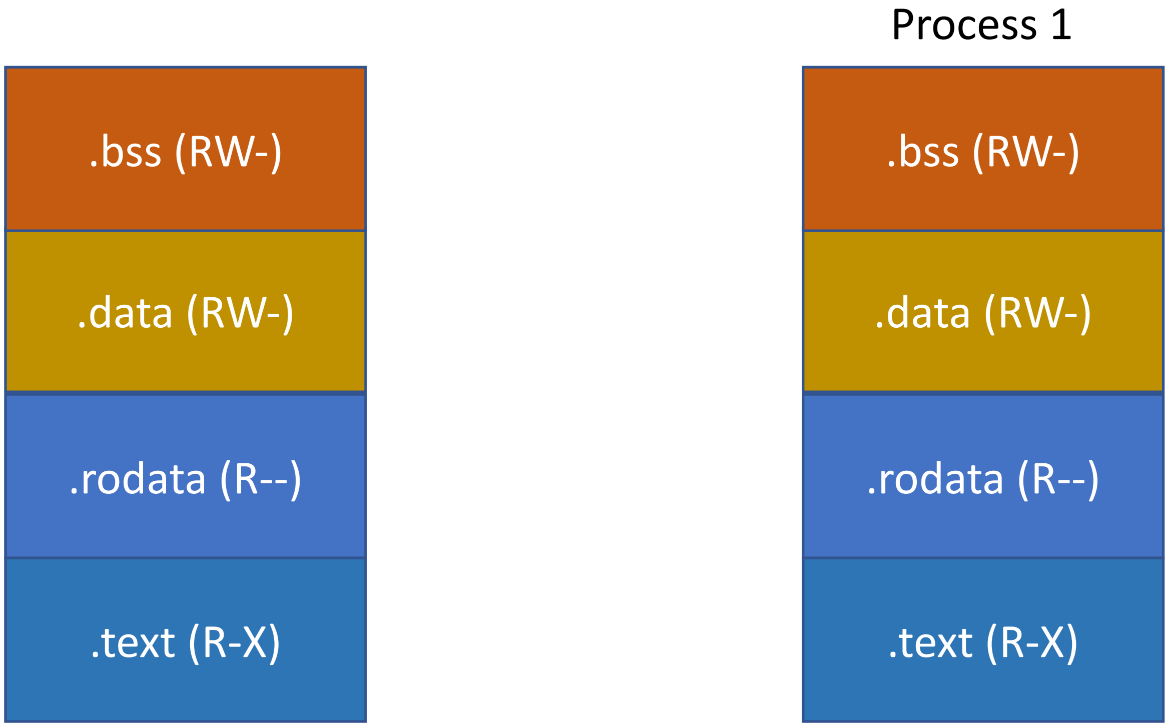
.bss (RW-)

.data (RW-)

.rodata (R--)

.text (R-X)

Running same program multiple times



Process 1

.bss (RW-)

.data (RW-)

.rodata (R--)

.text (R-X)

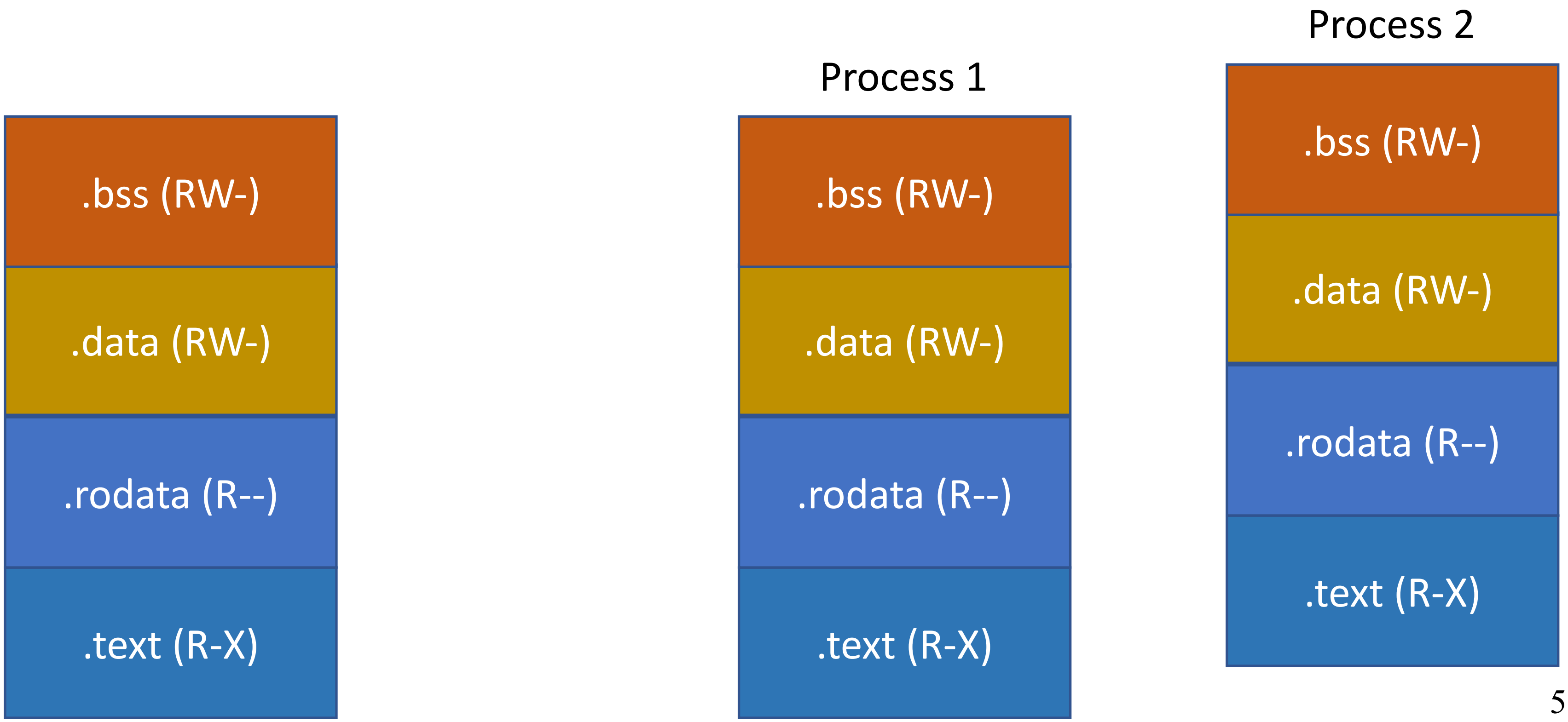
.bss (RW-)

.data (RW-)

.rodata (R--)

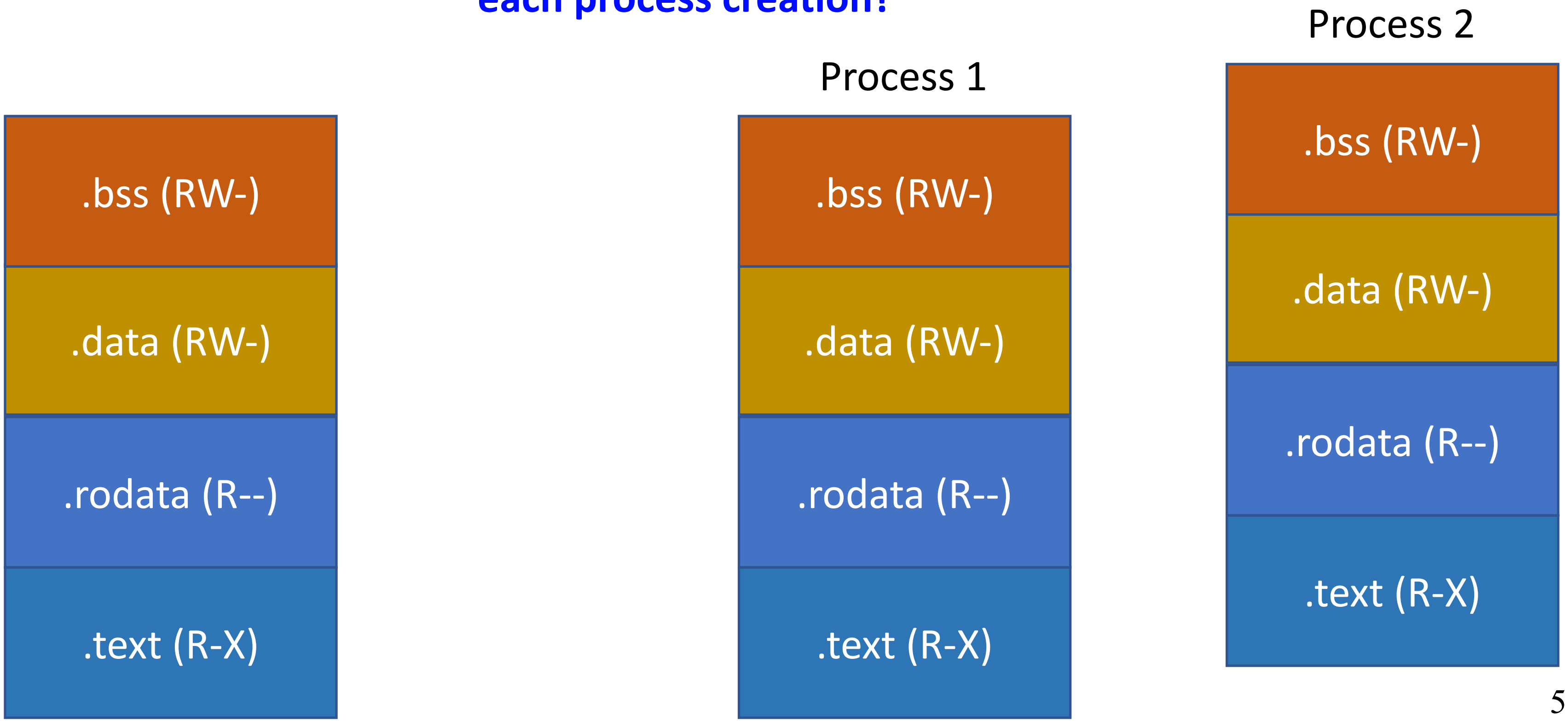
.text (R-X)

Running same program multiple times



Running same program multiple times

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



.bss (RW-)

.data (RW-)

.rodata (R--)

.text (R-X)

Process 1

.bss (RW-)

.data (RW-)

.rodata (R--)

.text (R-X)

Process 2

.bss (RW-)

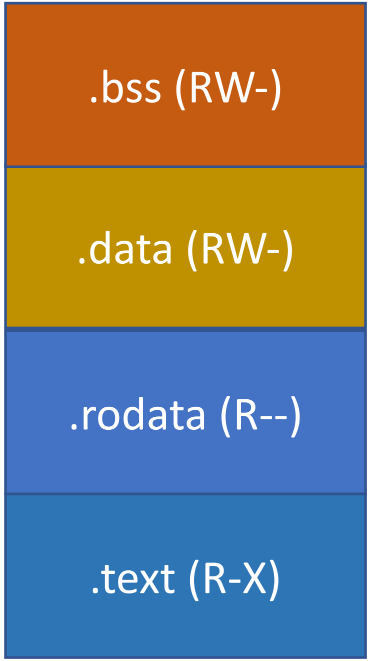
.data (RW-)

.rodata (R--)

.text (R-X)

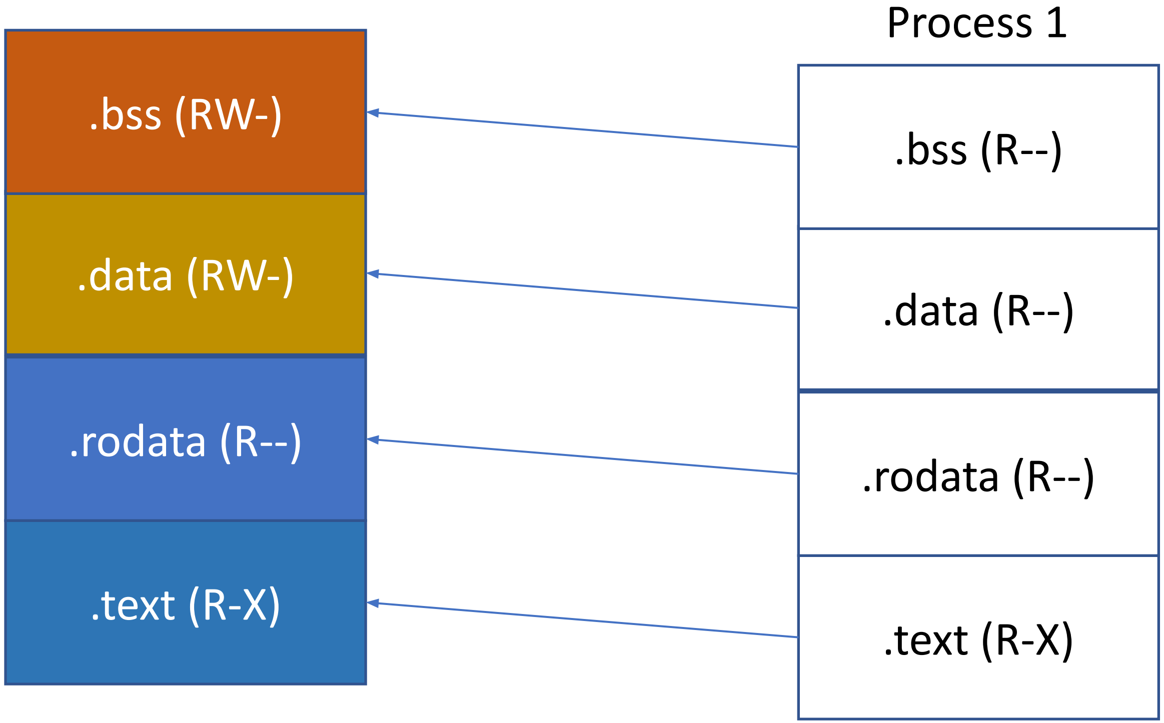
Sharing pages by marking read-only

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



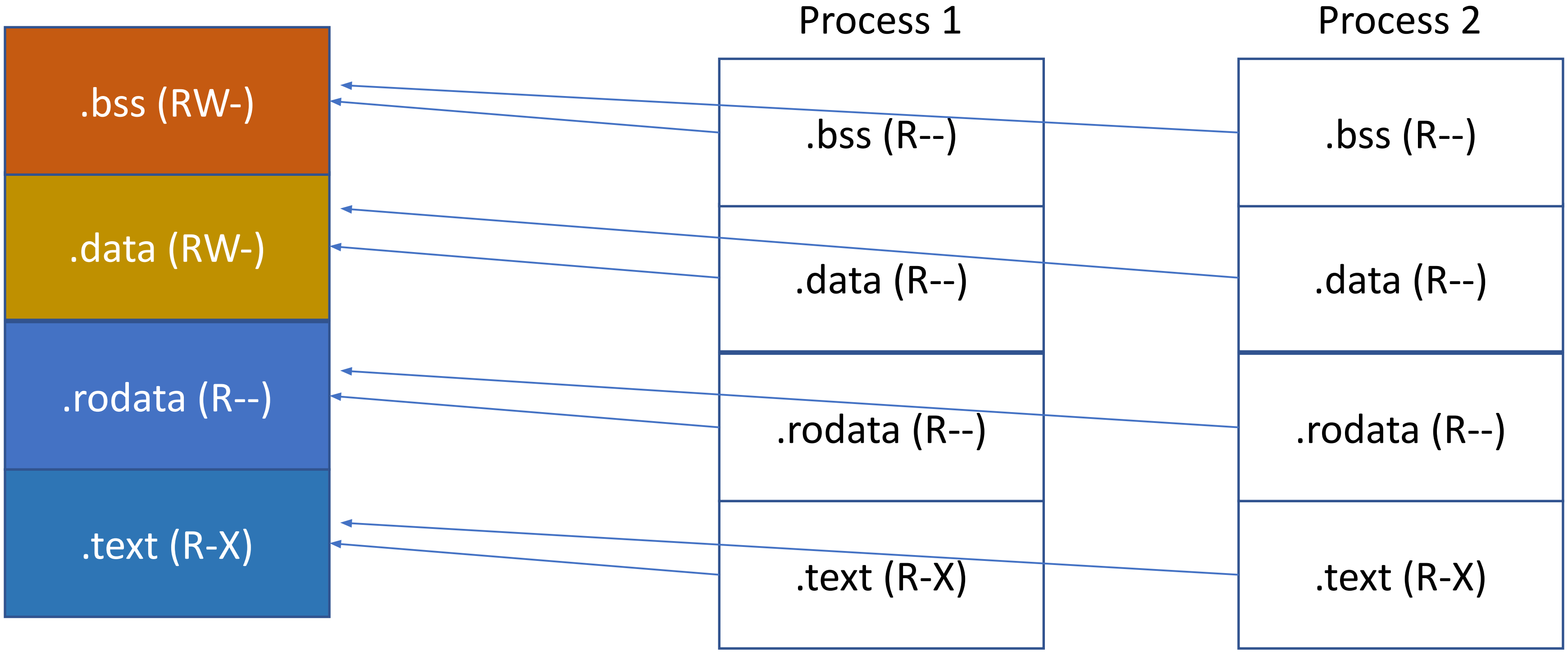
Sharing pages by marking read-only

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



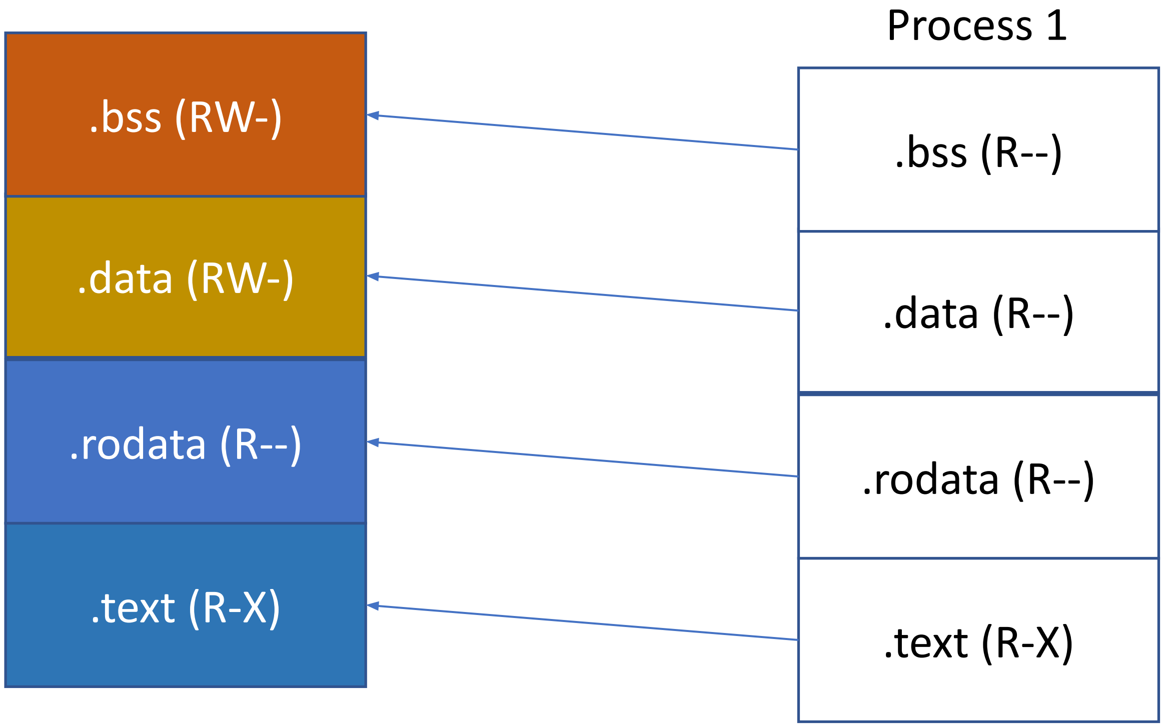
Sharing pages by marking read-only

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



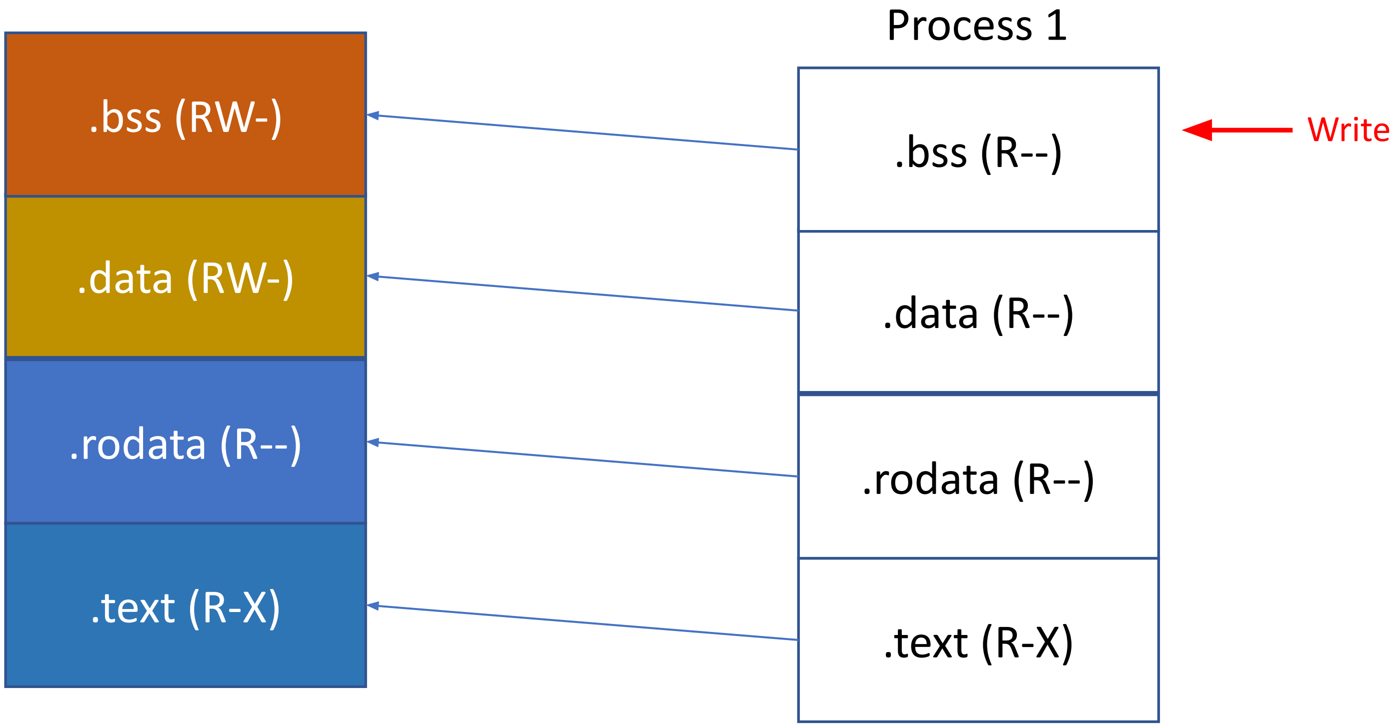
What about writes?

- How can Process 1 write on .bss??



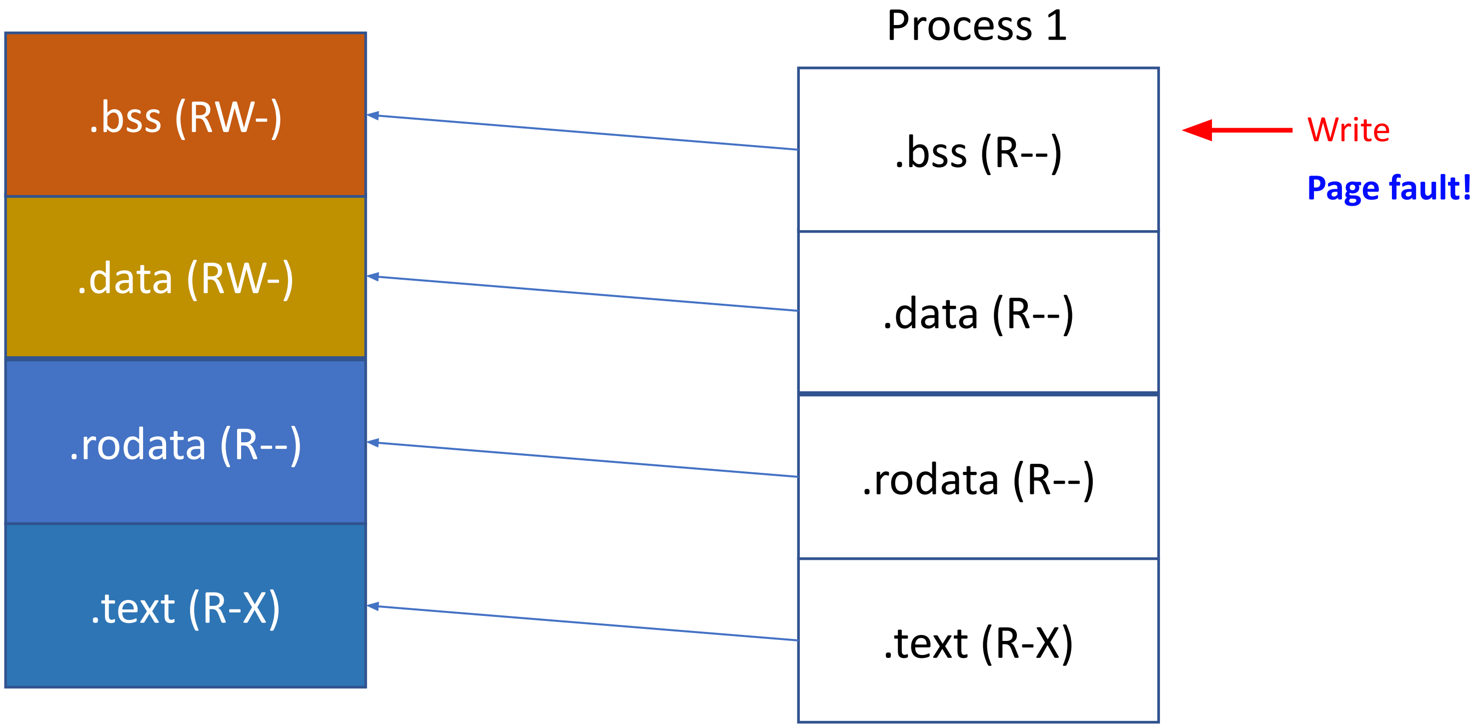
What about writes?

- How can Process 1 write on .bss??



What about writes?

- How can Process 1 write on .bss??



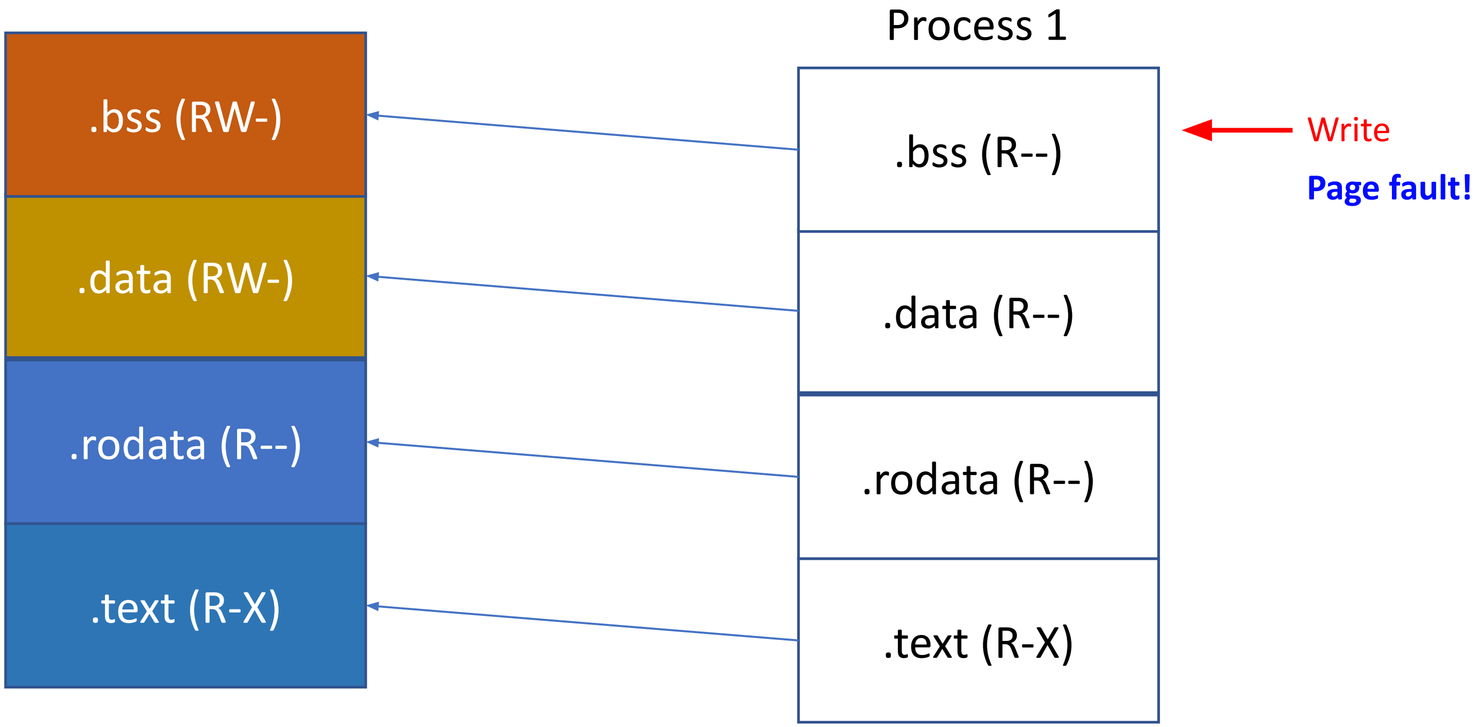
Handling writes: Copy-on-Write

- Read CR2
  - 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…

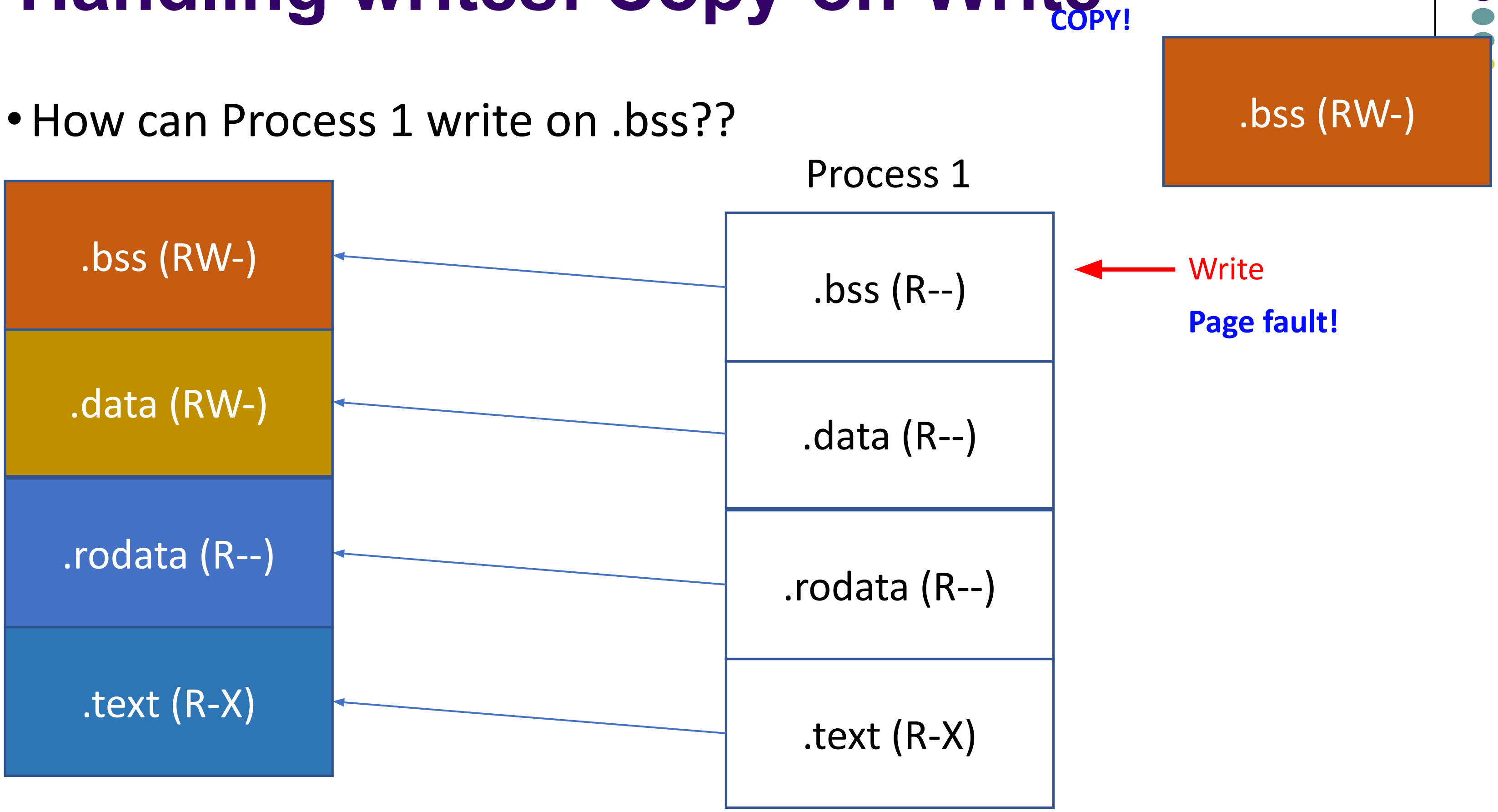
Handling writes: Copy-on-Write

- How can Process 1 write on .bss??



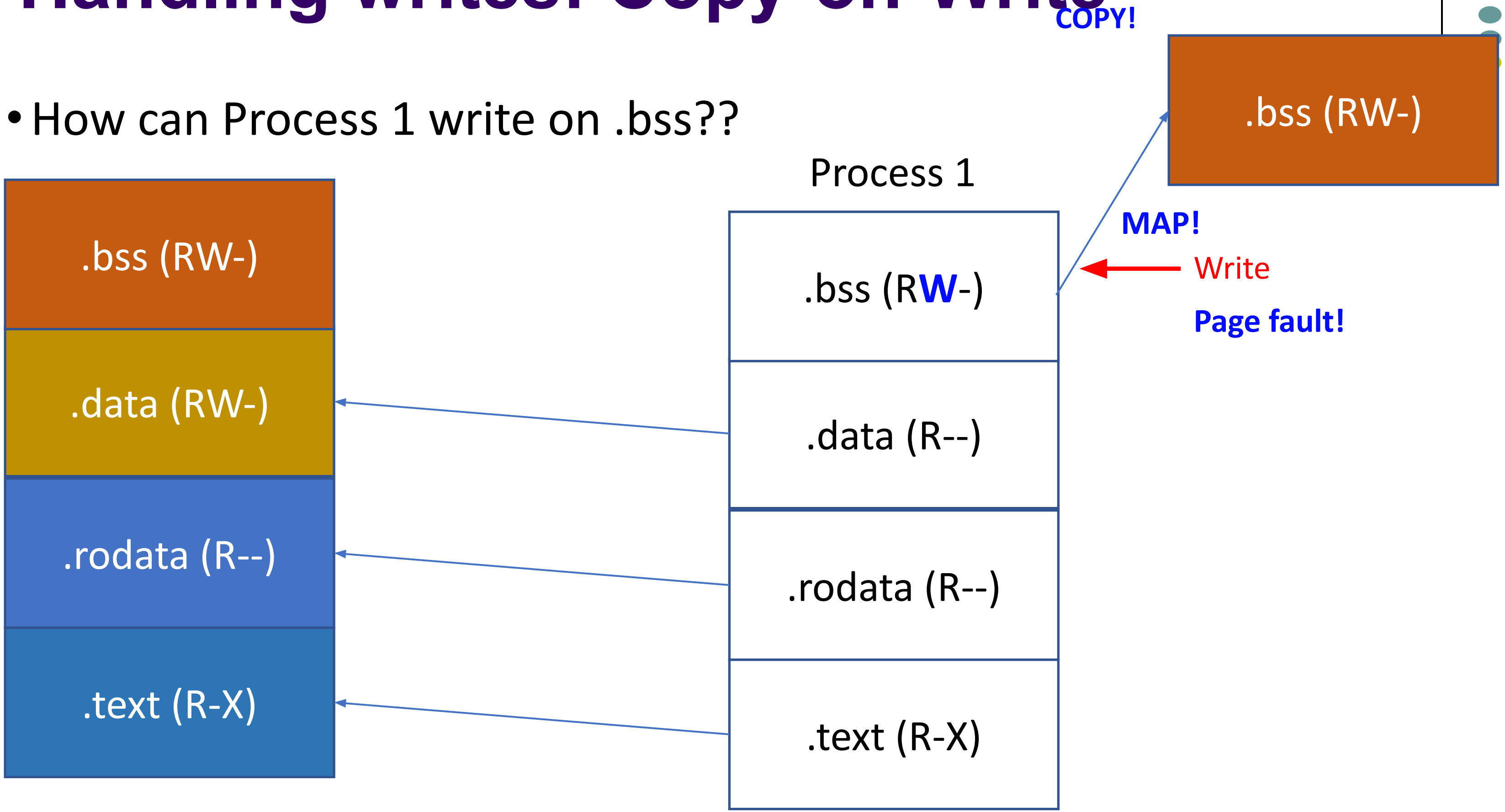
Handling writes: Copy-on-Write

- How can Process 1 write on .bss??



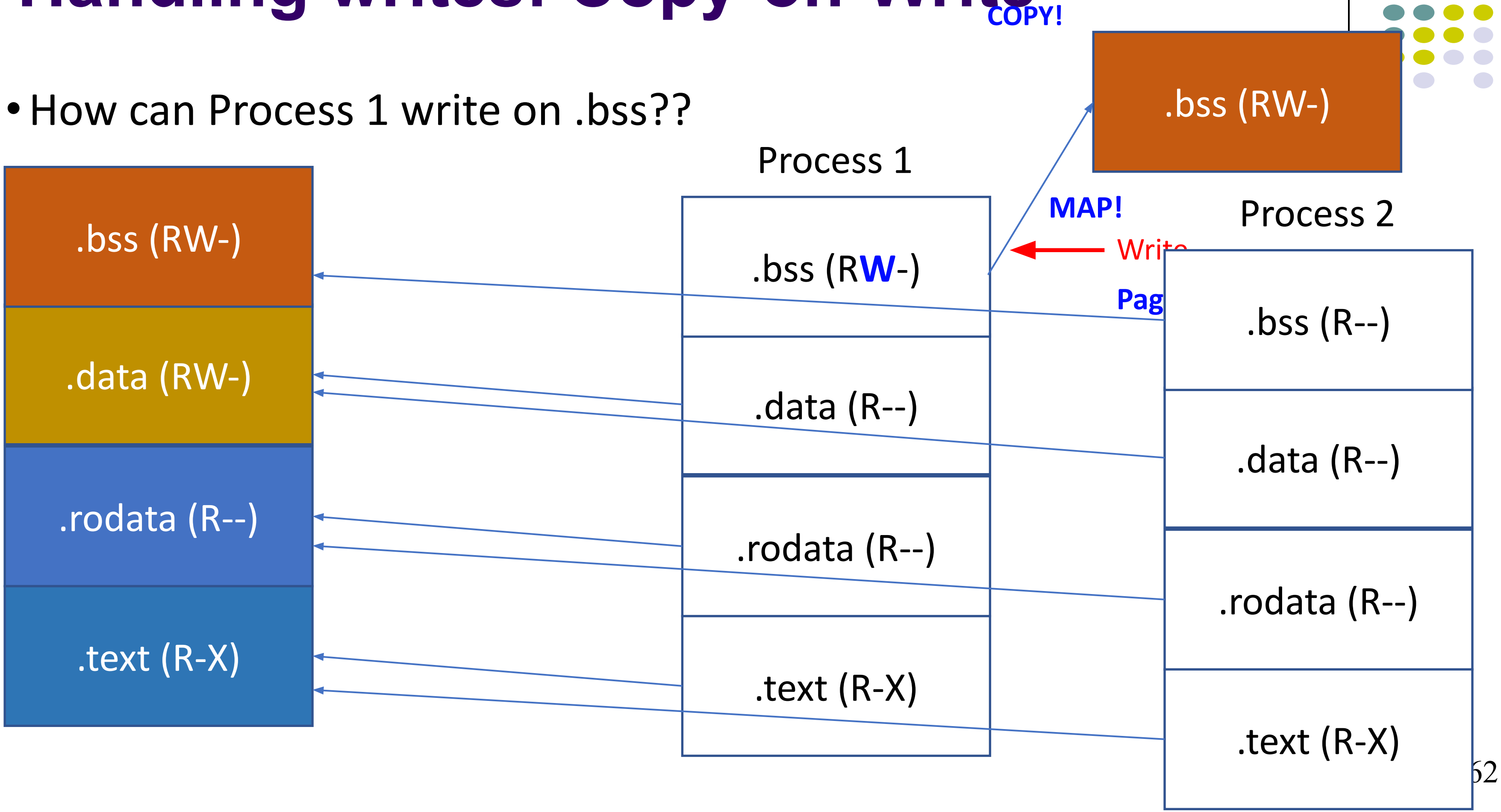
Handling writes: Copy-on-Write

- How can Process 1 write on .bss??



Handling writes: Copy-on-Write

- How can Process 1 write on .bss??



Benefits

- 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 nothing has written at all)
  - Can create private pages seamlessly on write

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 has written at all)
  - Can create private pages seamlessly on write

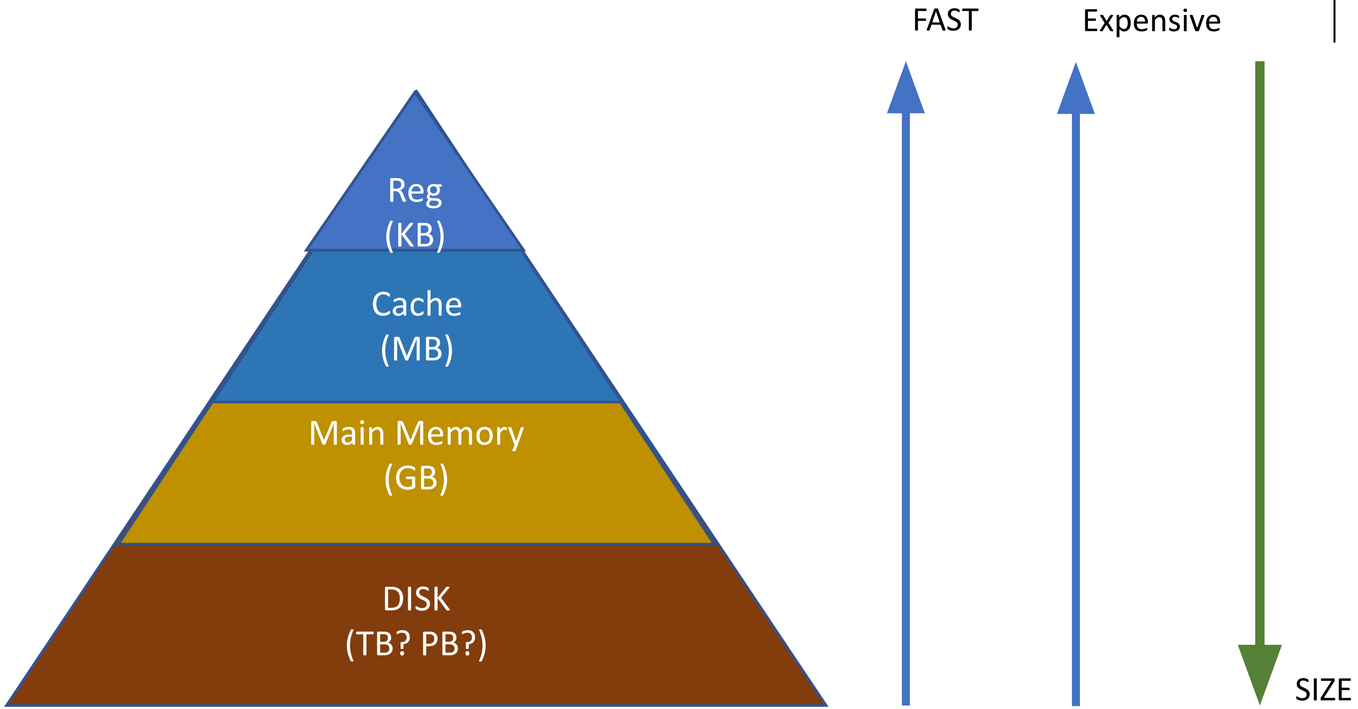
Handling low memory

- 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?

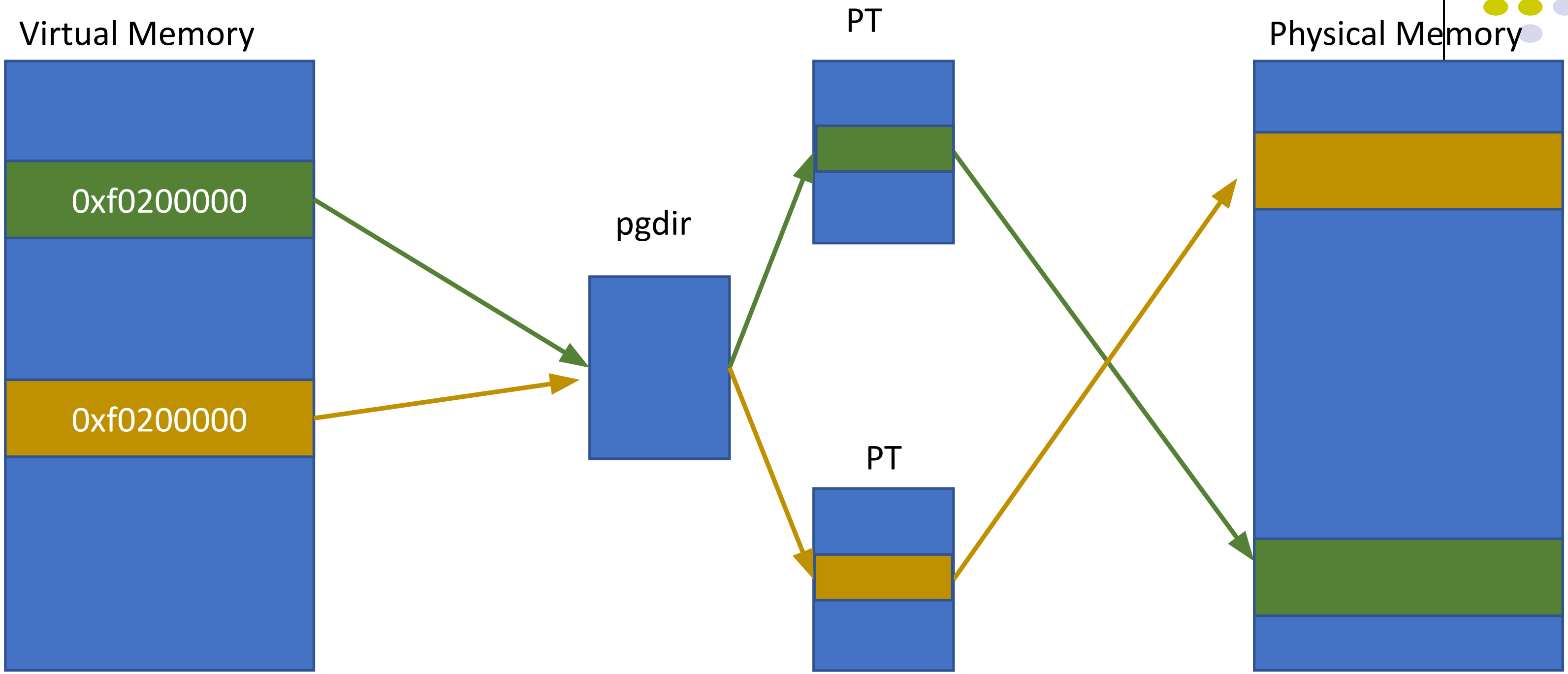
Memory Hierarchy



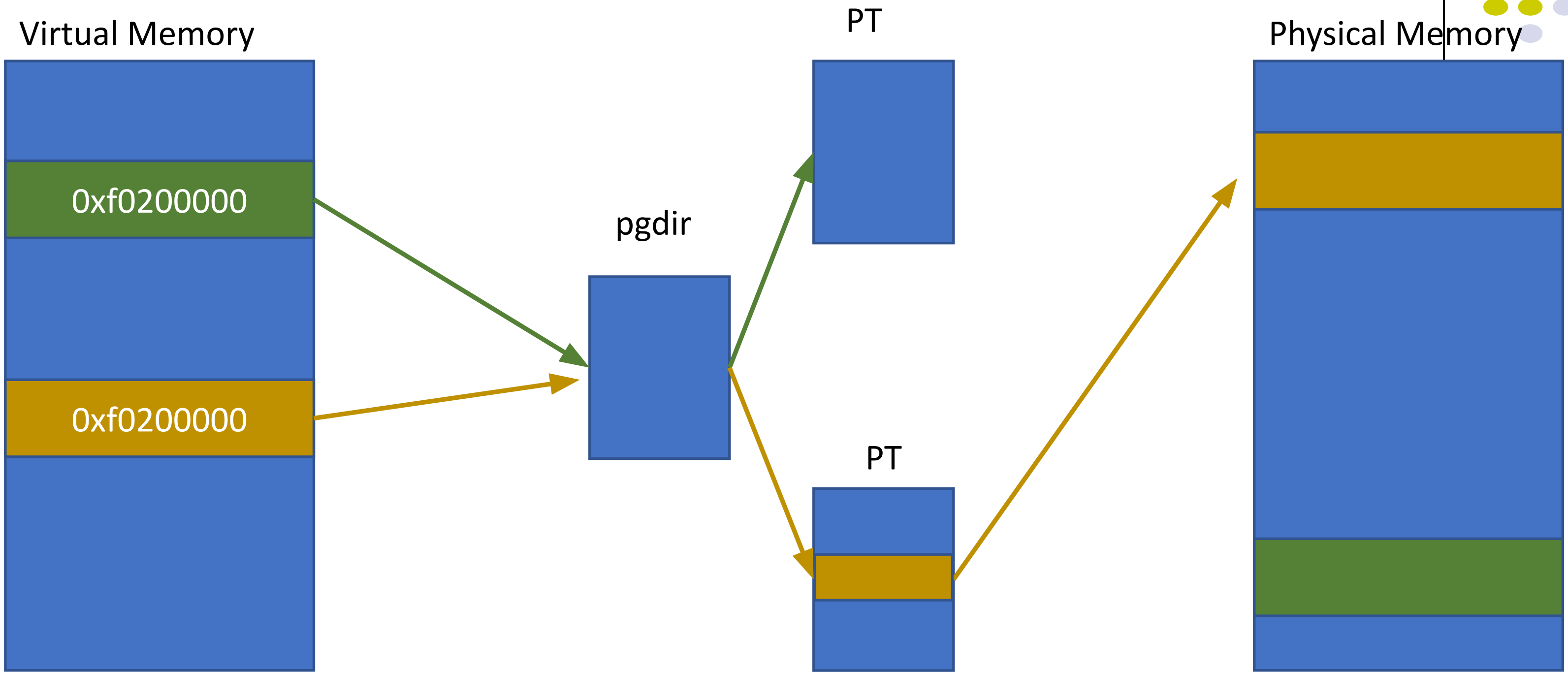
Memory Swapping

- Use disk as backing store under memory pressure

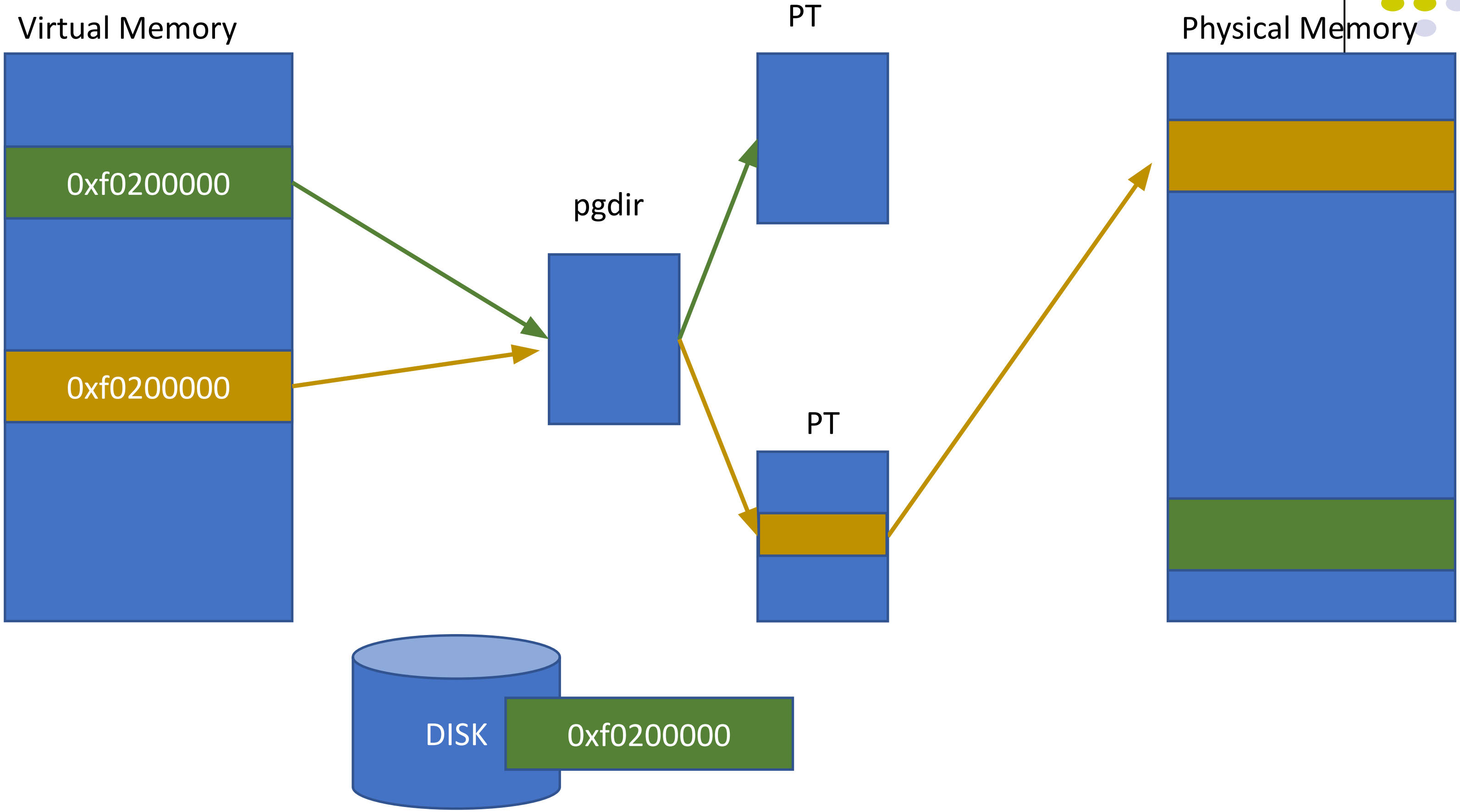
Memory Swapping



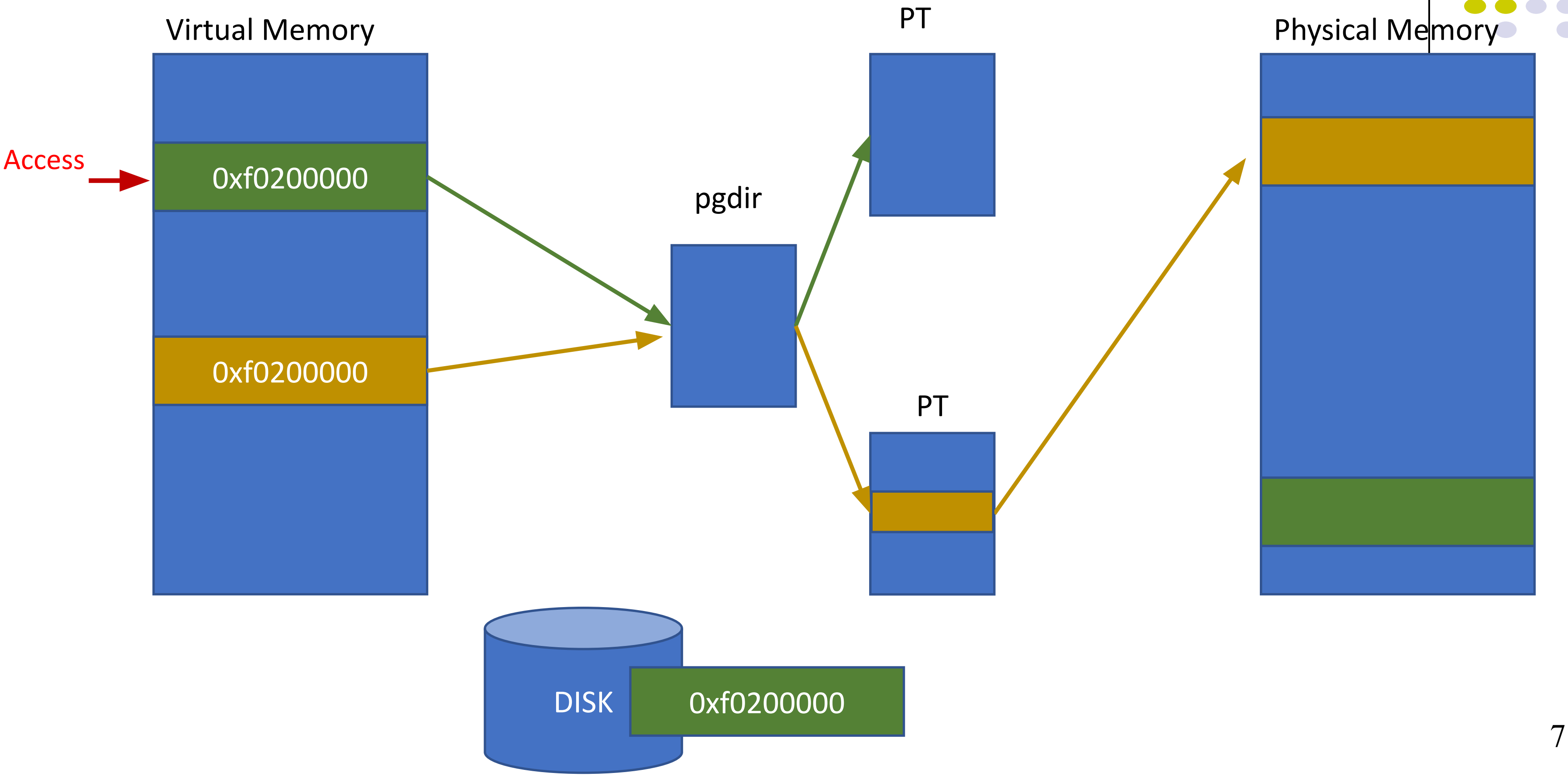
Memory Swapping - Removing a page



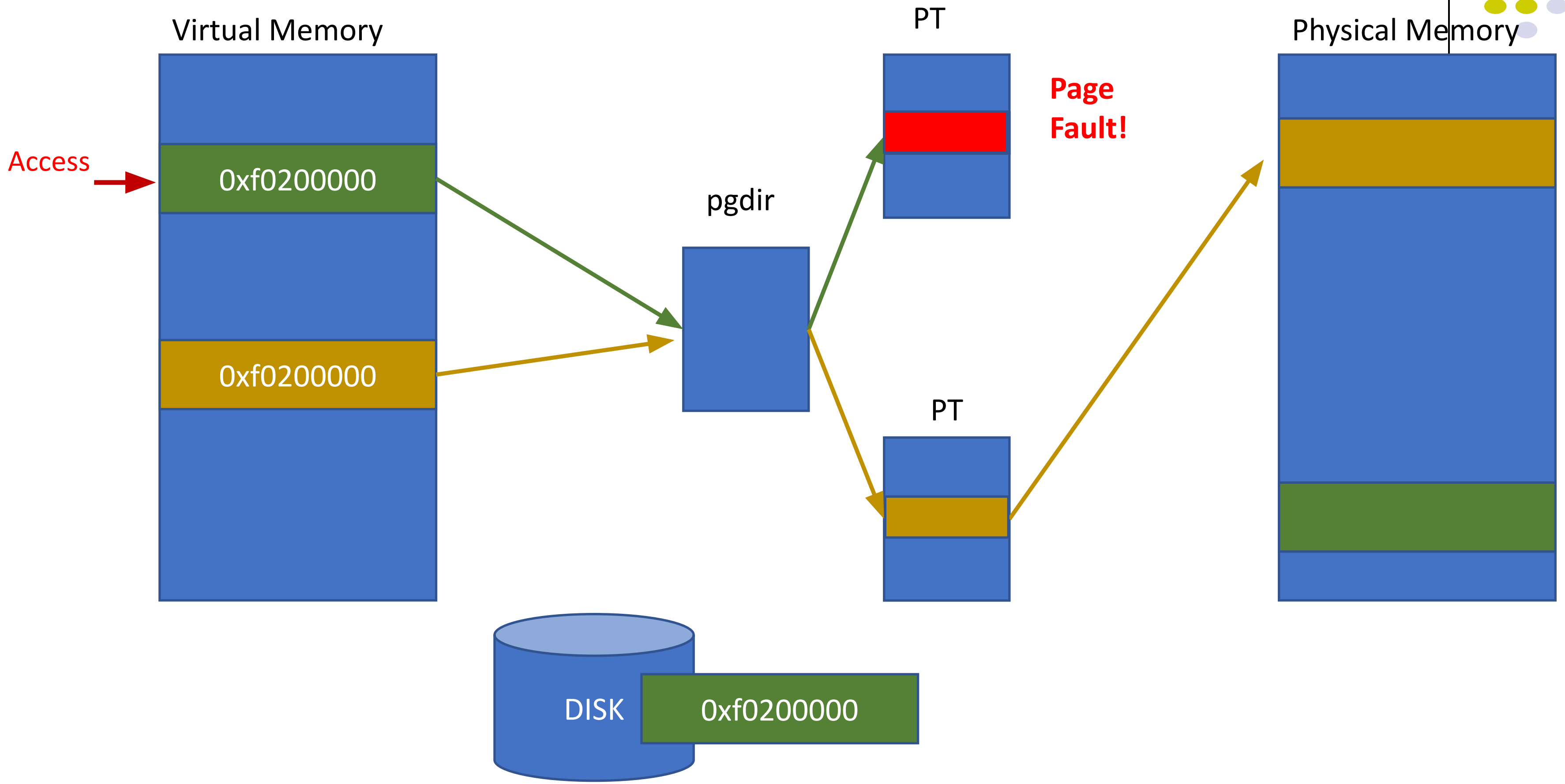
Memory Swapping - Removing a page



Memory Swapping - Removing a page



Memory Swapping - Removing a page



Swapping - Transparently load page from disk

- Page fault handler

Swapping - Transparently load page from disk

- Page fault handler
  - Read CR2 (get address, `0xf0200000`)

Swapping - Transparently load page from disk

- Page fault handler
  - Read CR2 (get address, `0xf0200000`)
  - Read error code

Swapping - Transparently load page from disk

- 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

Swapping - Transparently load page from disk

- 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

Swapping - Transparently load page from disk

- 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)

Swapping - Transparently load page from disk

- 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

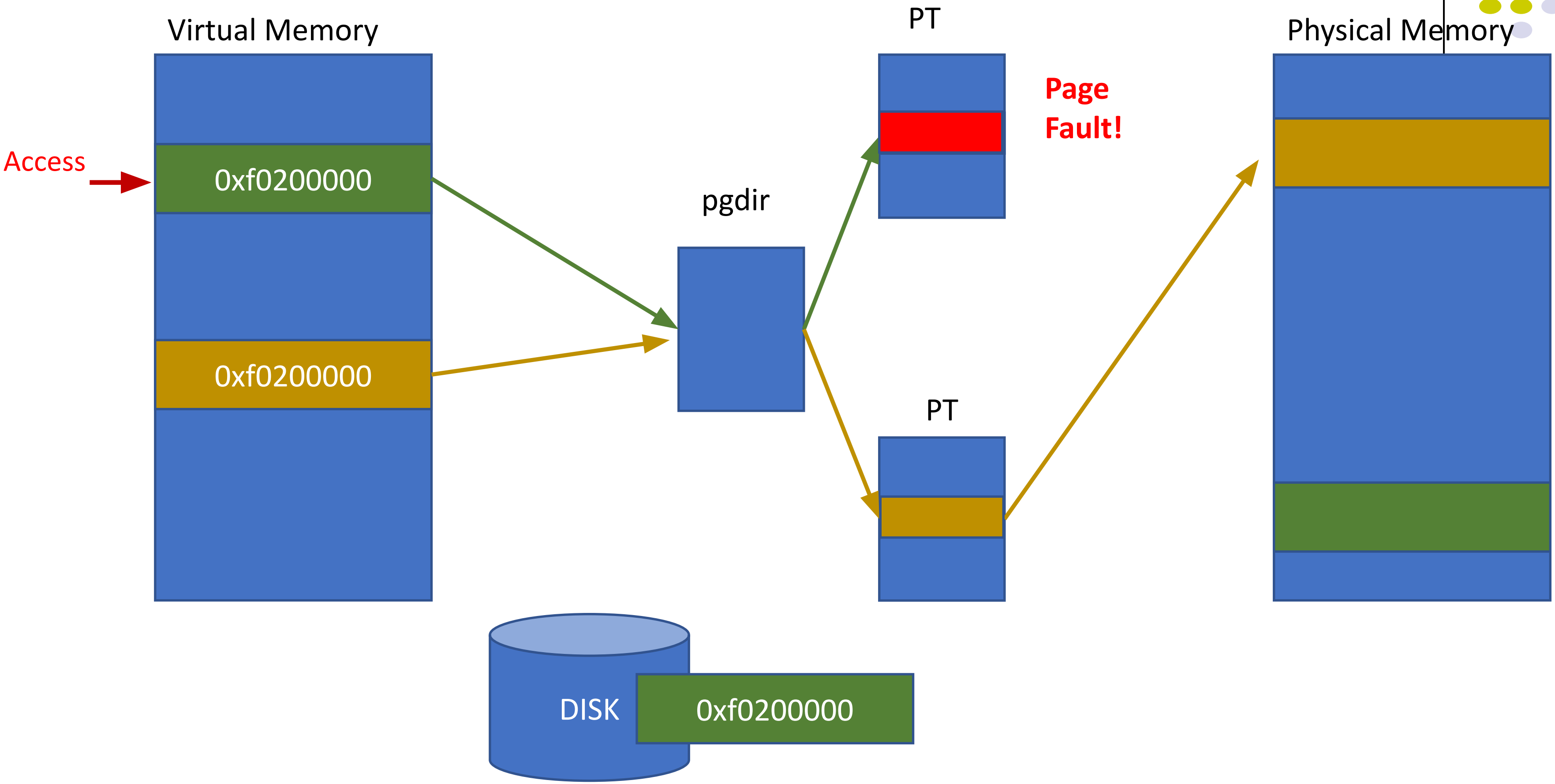
Swapping - Transparently load page from disk

- 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

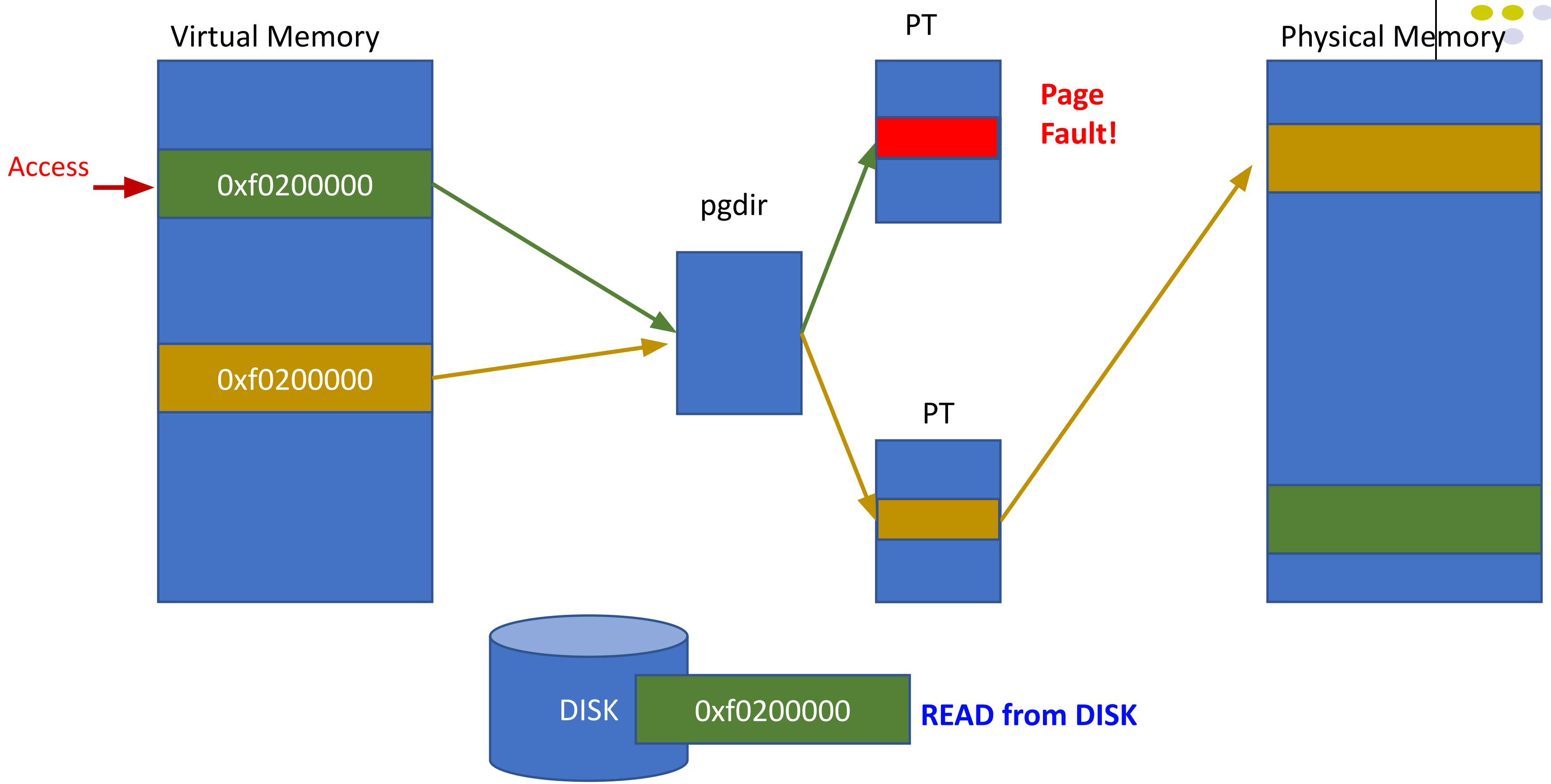
Swapping - Transparently load page from disk

- 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!

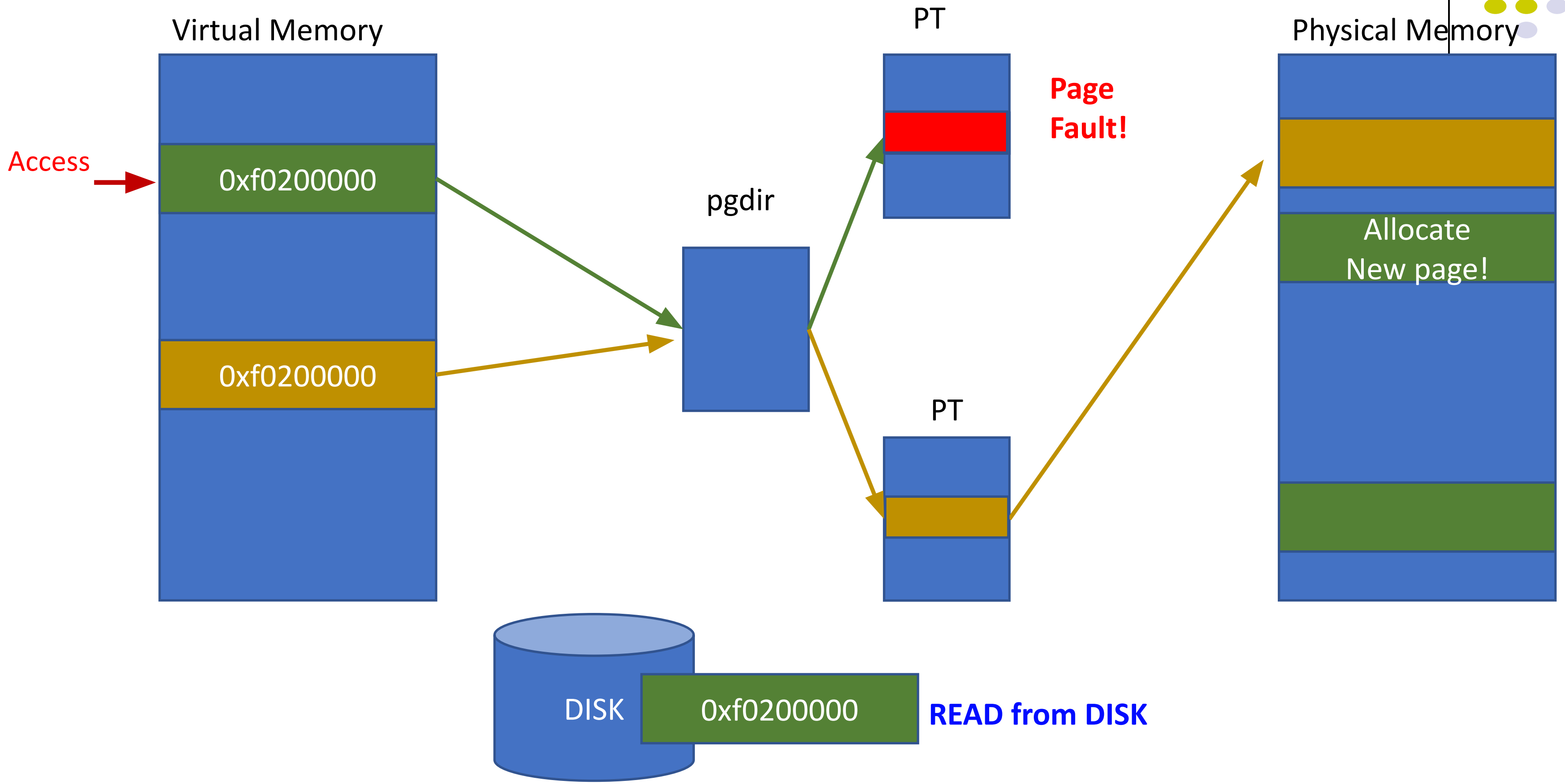
Swapping - Transparently load page from disk



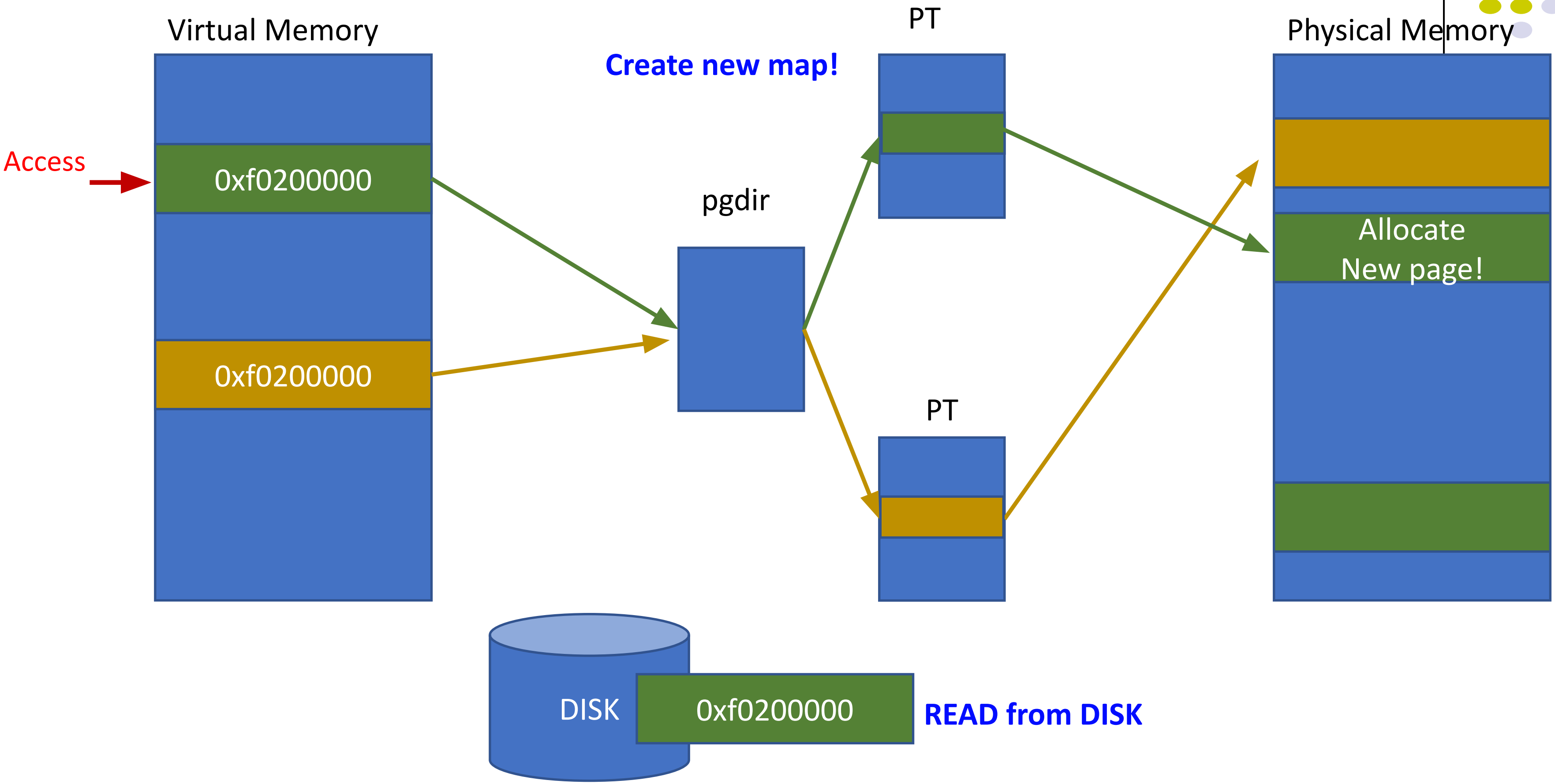
Swapping - Transparently load page from disk



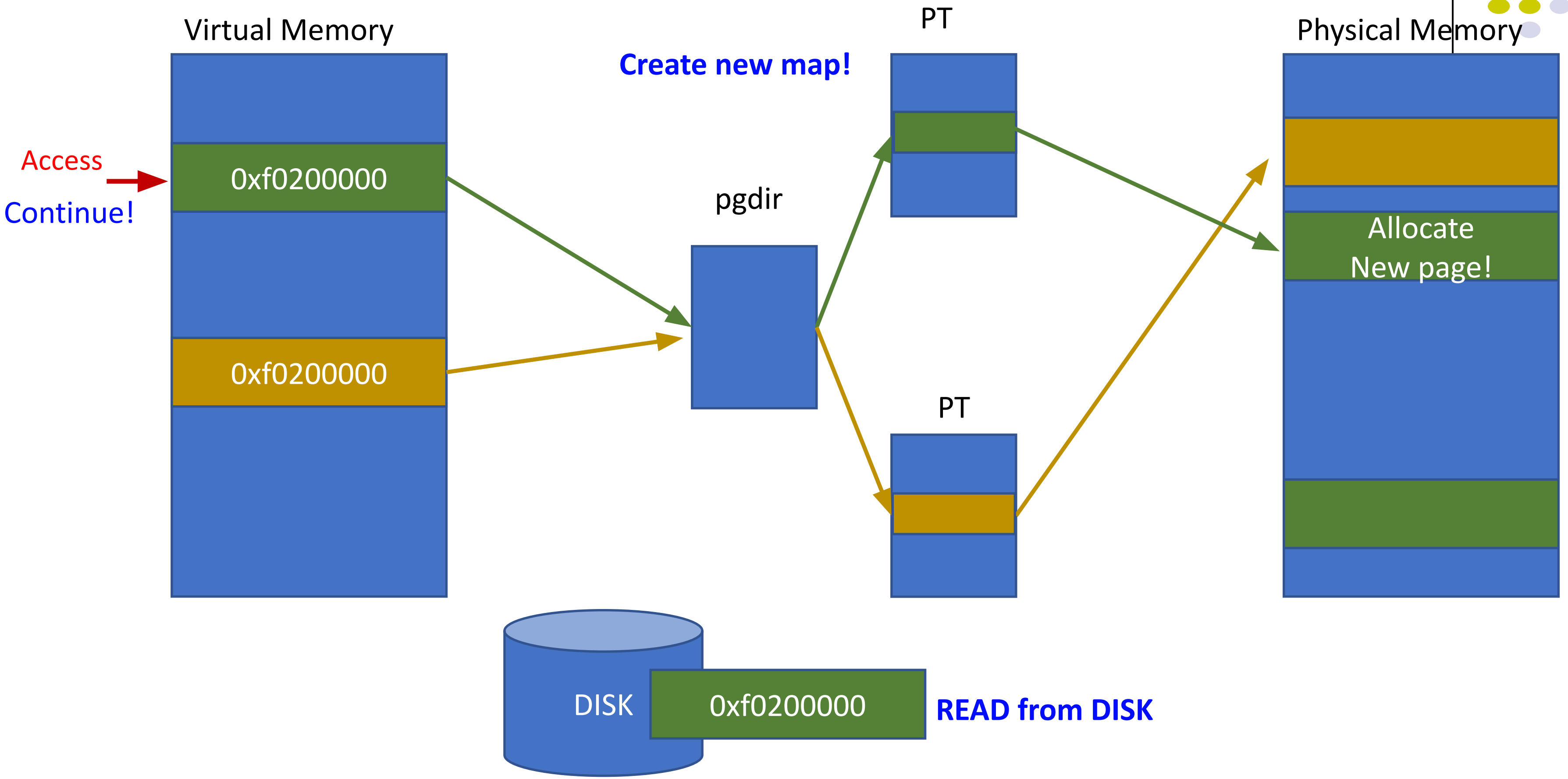
Swapping - Transparently load page from disk



Swapping - Transparently load page from disk



Swapping - Transparently load page from disk



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)
  - Swapping