System calls and Page faults

ECE 469, Feb 10

Aravind Machiry

Recap: Interrupts

- Hardware Interrupts

- Software Interrupts

Recap: Hardware Interrupts

- A way of hardware interacting with CPU

- Example: a network device
  - NIC: “Hey, CPU, I have a packet received for the OS, so please wake up the OS to handle the data”
  - CPU: call the interrupt handler for network device in ring 0 (set by the OS)

- Asynchronous (can happen at any time of execution)
  - It’s a request from a hardware, so it comes at any time of CPU’s execution

- Read
  - https://en.wikipedia.org/wiki/Intel_8259
  - https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller

Recap: Software Interrupts / exceptions

- A software mean to run code in ring 0 (e.g., int $0x30)
  - Telling CPU that ”Please run the interrupt handler at 0x30”

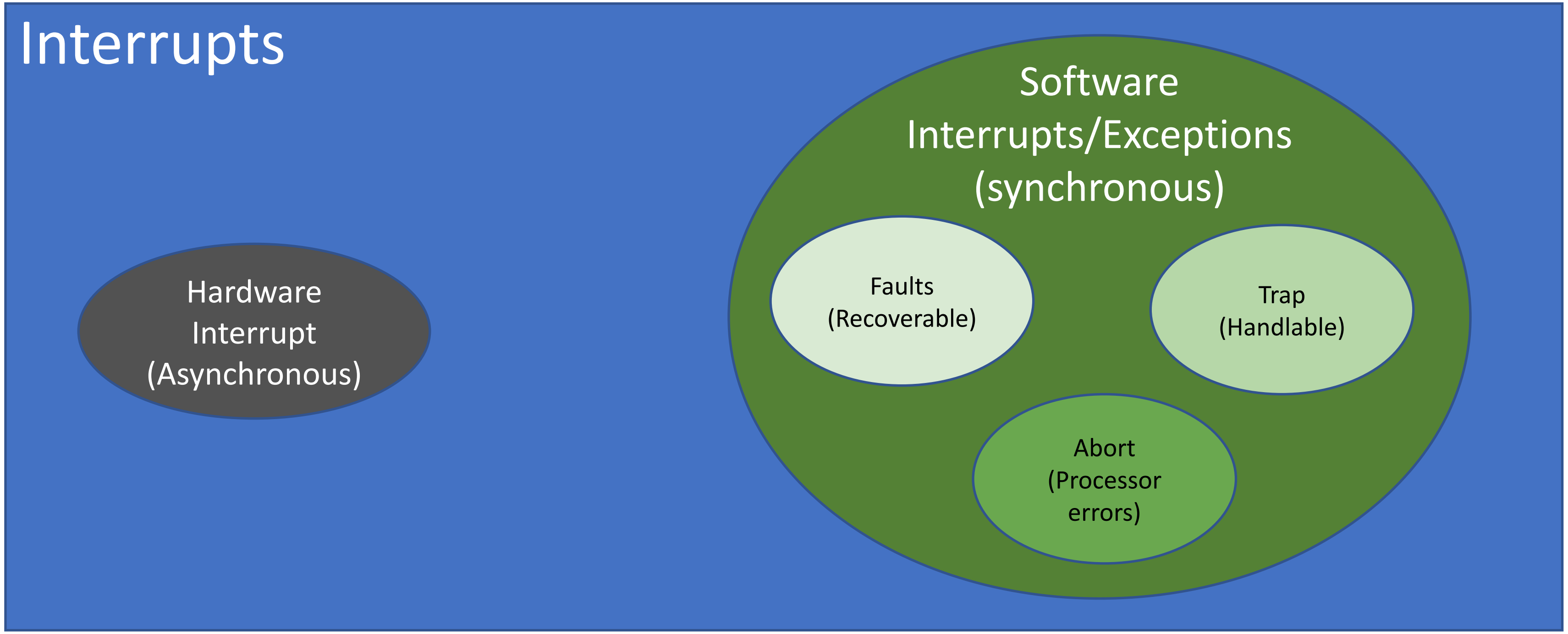
- Synchronous (caused by running an instruction, e.g., int $0x30)

- System call
  - int $0x30 □ system call in JOS

Recap: Types of exceptions

- Classification based on how they are handled:
    - Fault
        - Exception occurred but can be fixed
        - IP points to the current instruction
    - Trap
        - Exception occurred but the program could continue execution
        - IP points to next instruction
    - Abort
        - Unhandlable exception
        - Hardware failures in processor

Recap: Interrupts classification


Interrupts
Hardware Interrupt (Asynchronous)
Software Interrupts/Exceptions (synchronous)
Faults (Recoverable)
Trap (Handlable)
Abort (Processor errors)

Recap: Handling Interrupts

- Setting an Interrupt Descriptor Table (IDT)

Interrupt Number	Code address
0 (Divide error)	0xf0130304
1 (Debug)	0xf0153333
2 (NMI, Non-maskable Interrupt)	0xf0183273
3 (Breakpoint)	0xf0223933
4 (Overflow)	0xf0333333
…	
8 (Double Fault)	0xf0222293
…	
14 (Page Fault)	0xf0133390
…	…
0x30 (syscall in JOS)	0xf0222222

Recap: Handling Interrupts

- Setting an Interrupt Descriptor Table (IDT)

Interrupt Number	Code address
0 (Divide error)	0xf0130304
1 (Debug)	0xf0153333
2 (NMI, Non-maskable Interrupt)	0xf0183273
3 (Breakpoint)	0xf0223933
4 (Overflow)	0xf0333333
…	
8 (Double Fault)	0xf0222293
…	
14 (Page Fault)	0xf0133390
…	…
0x30 (syscall in JOS)	0xf0222222

Load the base address into IDTR

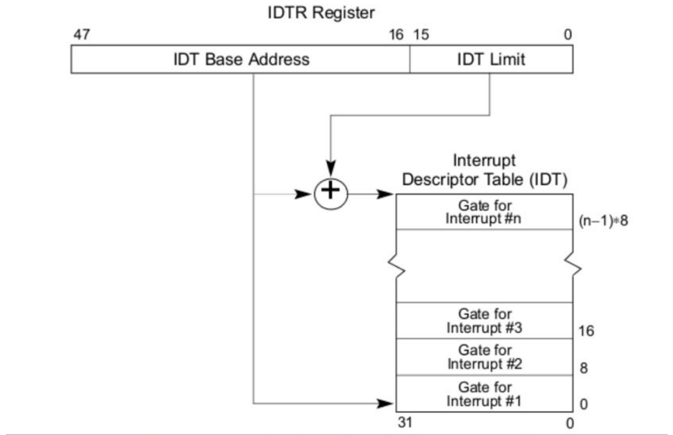
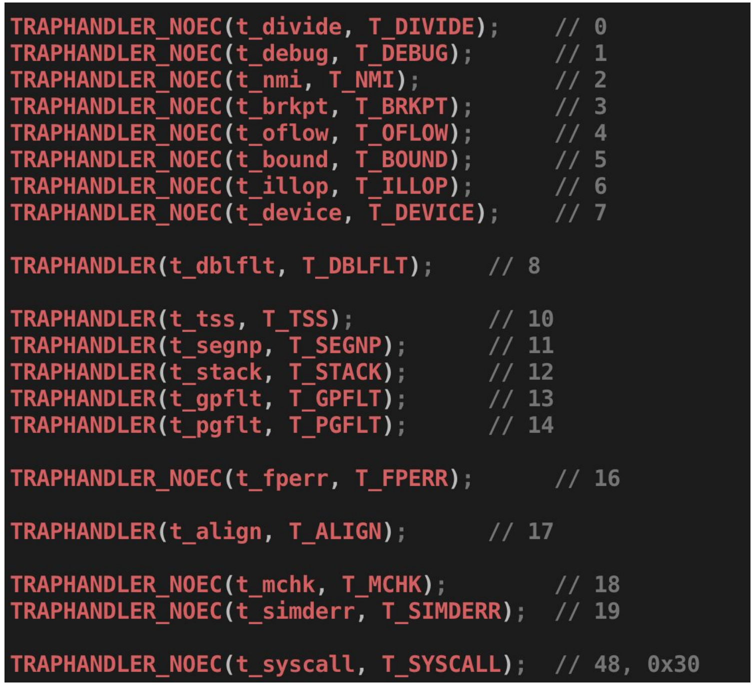


Figure 6-1. Relationship of the IDTR and IDT

Recap: Handling Interrupts

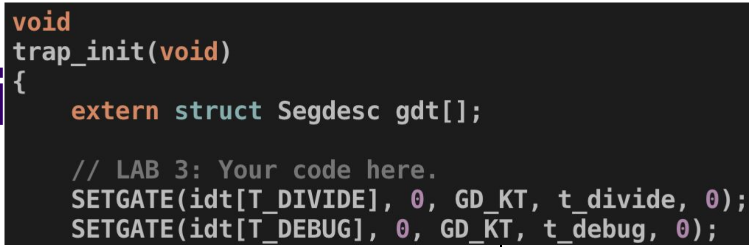
- Setting an Interrupt Descriptor Table (IDT)

Interrupt Number	Code address
0 (Divide error)	t_divide
1 (Debug)	t_debug
2 (NMI, Non-maskable Interrupt)	t_nmi
3 (Breakpoint)	t_brkpt
4 (Overflow)	t_oflow
…	
8 (Double Fault)	t_dblflt
…	
14 (Page Fault)	t_pgflt
…	…
0x30 (syscall in JOS)	t_syscall



```
TRAPHANDLER_NOEC(t_divide, T_DIVIDE);     // 0
TRAPHANDLER_NOEC(t_debug, T_DEBUG);       // 1
TRAPHANDLER_NOEC(t_nmi, T_NMI);           // 2
TRAPHANDLER_NOEC(t_brkpt, T_BRKPT);       // 3
TRAPHANDLER_NOEC(t_oflow, T_OFLOW);       // 4
TRAPHANDLER_NOEC(t_bound, T_BOUND);       // 5
TRAPHANDLER_NOEC(t_illop, T_ILLOP);       // 6
TRAPHANDLER_NOEC(t_device, T_DEVICE);     // 7

TRAPHANDLER(t_dblflt, T_DBLFLT);     // 8

TRAPHANDLER(t_tss, T_TSS);           // 10
TRAPHANDLER(t_segnp, T_SEGNP);       // 11
TRAPHANDLER(t_stack, T_STACK);       // 12
TRAPHANDLER(t_gpflt, T_GPFLT);       // 13
TRAPHANDLER(t_pgflt, T_PGFLT);       // 14

TRAPHANDLER_NOEC(t_fperr, T_FPERR);       // 16

TRAPHANDLER(t_align, T_ALIGN);       // 17

TRAPHANDLER_NOEC(t_mchk, T_MCHK);         // 18
TRAPHANDLER_NOEC(t_simderr, T_SIMDERR);   // 19

TRAPHANDLER_NOEC(t_syscall, T_SYSCALL);   // 48, 0x30
```

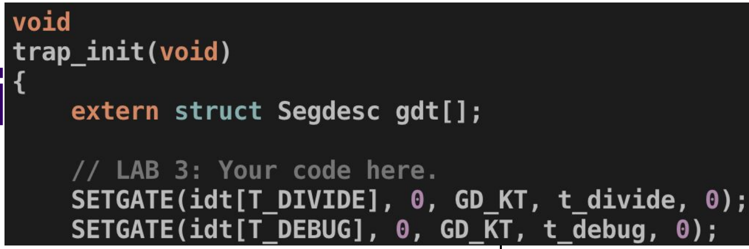
Recap: JOS Interrupt Handli



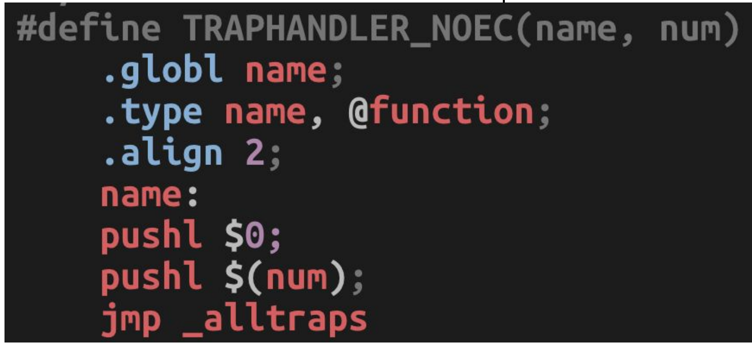
```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```

- Setup the IDT at trap_init() in kern/trap.c

Recap: JOS Interrupt Handli



```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```



```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

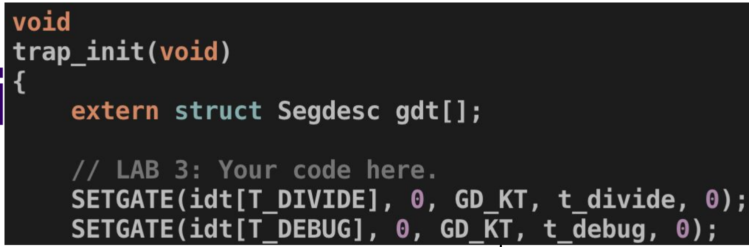
- Setup the IDT at trap_init() in kern/trap.c

- Interrupt arrives to CPU!
- Call interrupt hander in IDT
- Call _alltraps (in kern/trapentry.S)

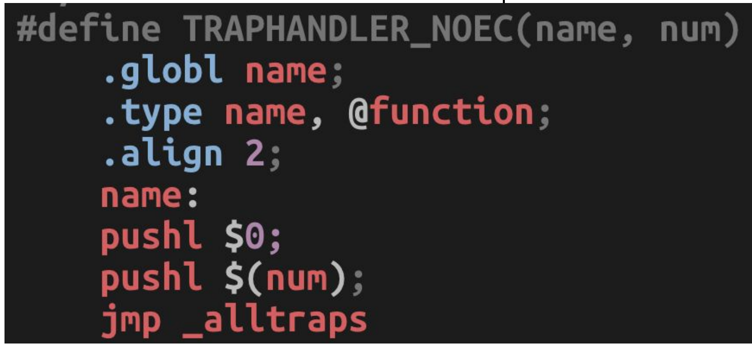
Recap: JOS Interrupt Handli

- Setup the IDT at trap_init() in kern/trap.c

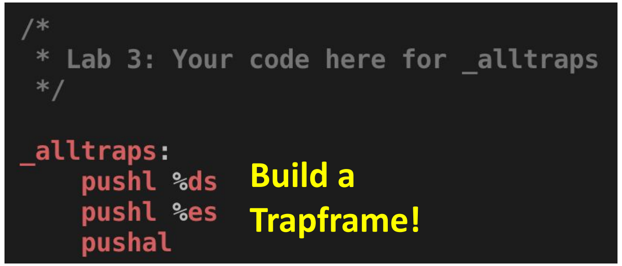
- Interrupt arrives to CPU!
- Call interrupt hander in IDT
- Call _alltraps (in kern/trapentry.S)
- Call trap() in kern/trap.c



```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```



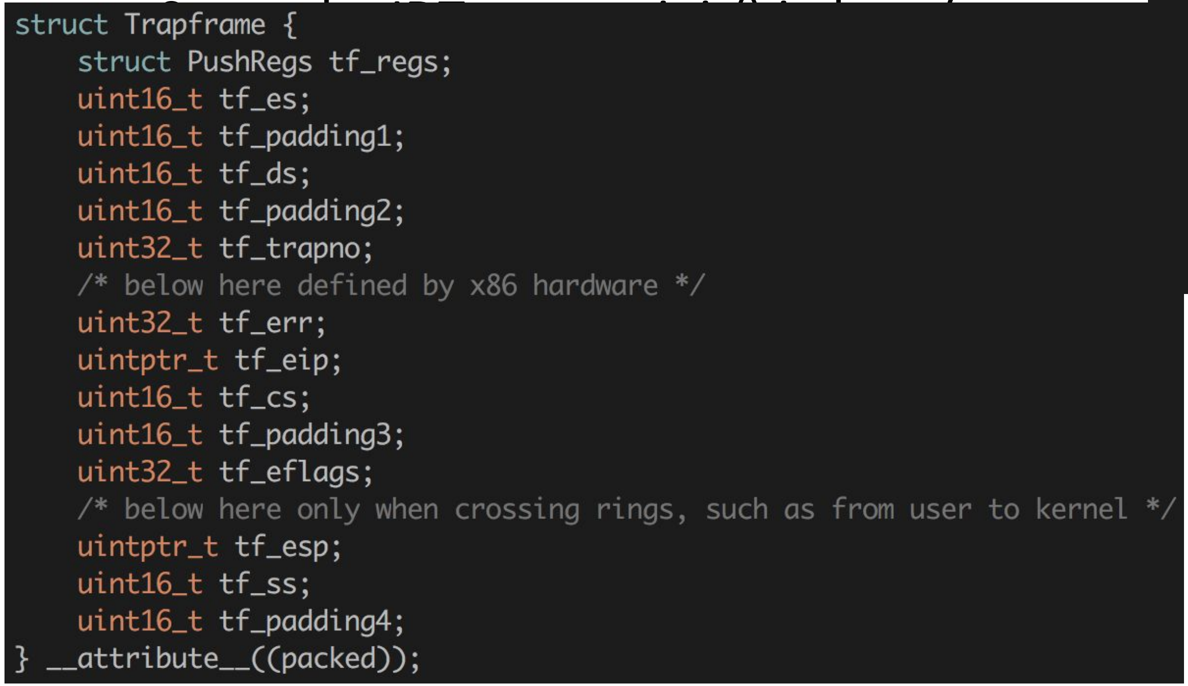
```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```



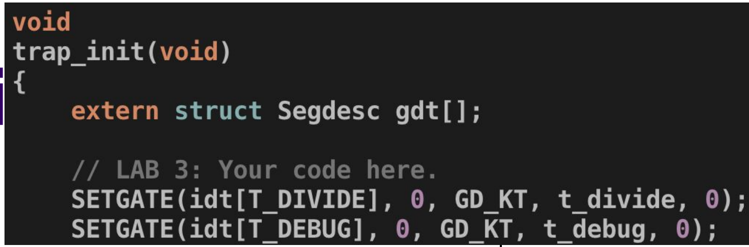
```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds
    pushl %es
    pushal
```

Build a Trapframe!

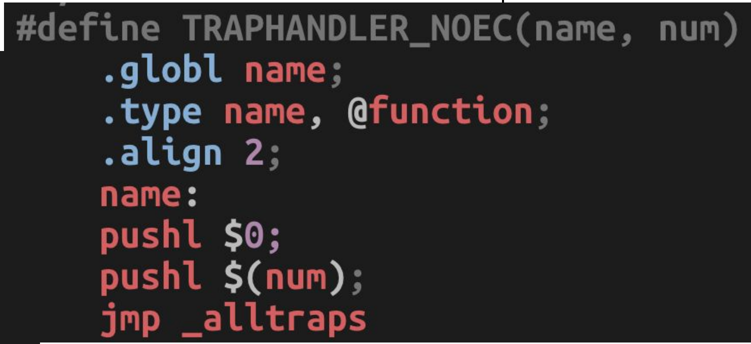
Recap: JOS Interrupt Handli



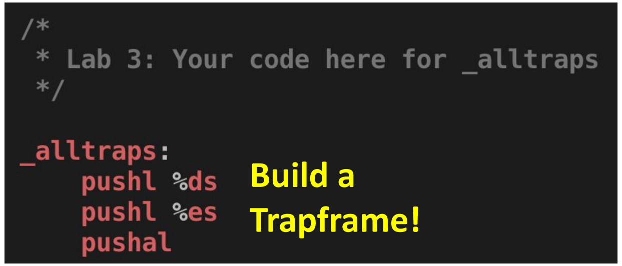
```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```



```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```



```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```



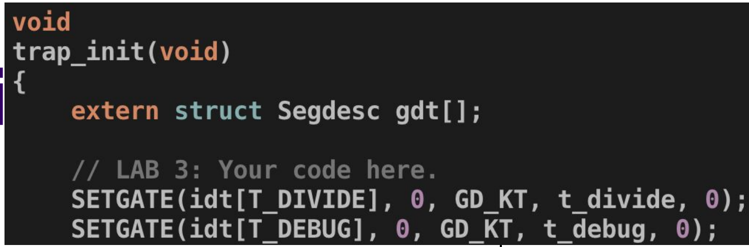
```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds
    pushl %es
    pushal
```

Build a Trapframe!

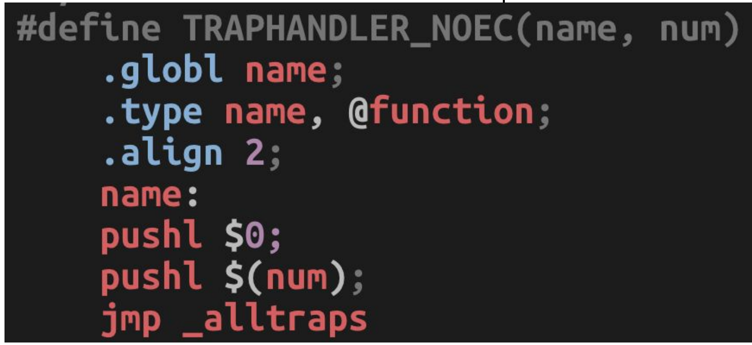
Recap: JOS Interrupt Handli

- Setup the IDT at trap_init() in kern/trap.c

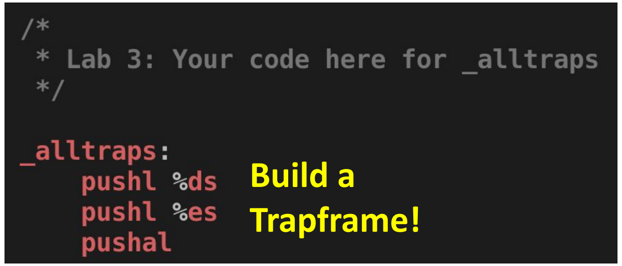
- Interrupt arrives to CPU!
- Call interrupt hander in IDT
- Call _alltraps (in kern/trapentry.S)
- Call trap() in kern/trap.c



```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```

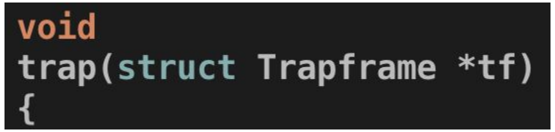


```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```



```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds
    pushl %es
    pushal
```

Build a Trapframe!

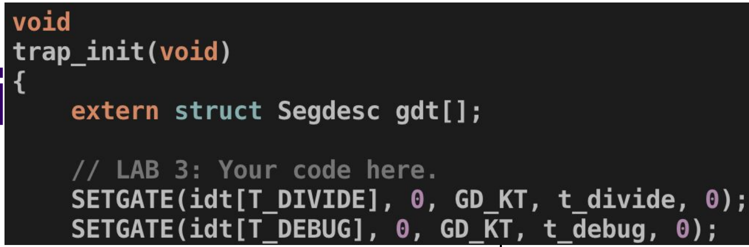


```
void
trap(struct Trapframe *tf)
{
```

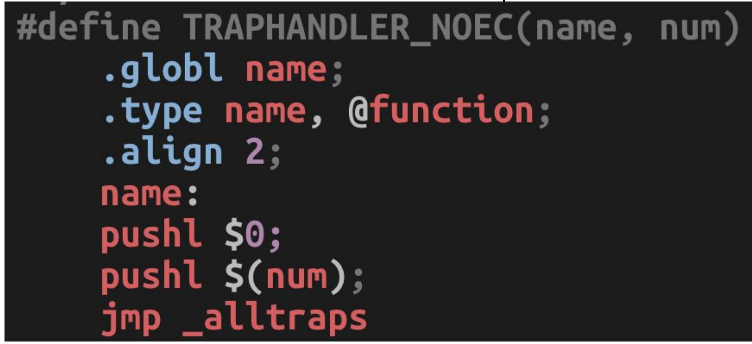
Recap: JOS Interrupt Handli

- Setup the IDT at trap_init() in kern/trap.c

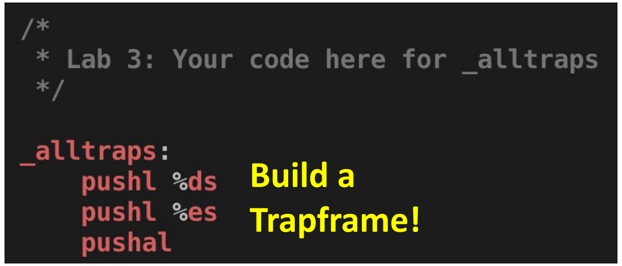
- Interrupt arrives to CPU!
- Call interrupt hander in IDT
- Call _alltraps (in kern/trapentry.S)
- Call trap() in kern/trap.c
- Call trap_dispatch() in kern/trap.c



```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```

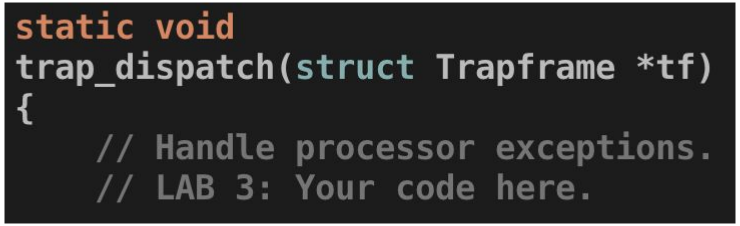


```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

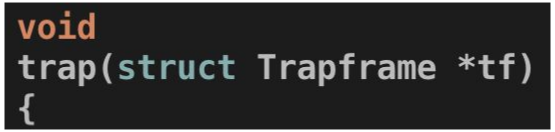


```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds
    pushl %es
    pushal
```

Build a Trapframe!



```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
```

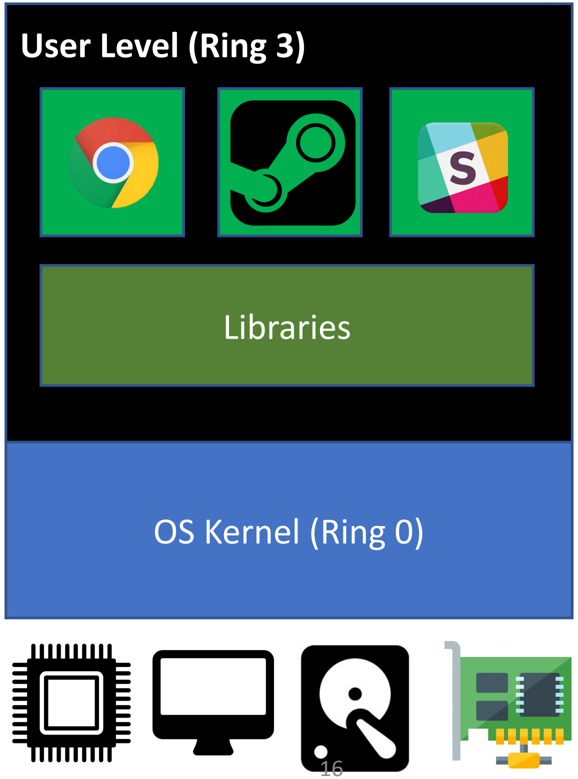


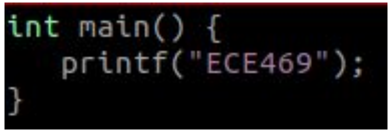
```
void
trap(struct Trapframe *tf)
{
```

Today

- Syscalls
- Page fault

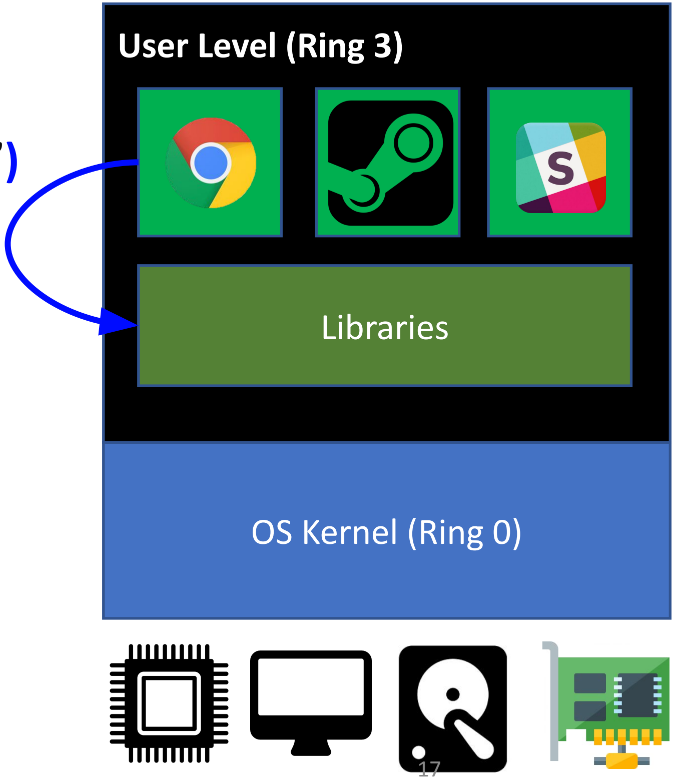
Syscall: User/Kernel communication



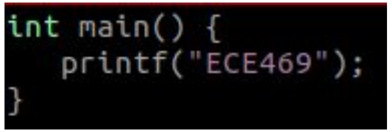


```
int main() {
   printf("ECE469");
}
```

Syscall: User/Kernel communication



A library call in ring 3



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

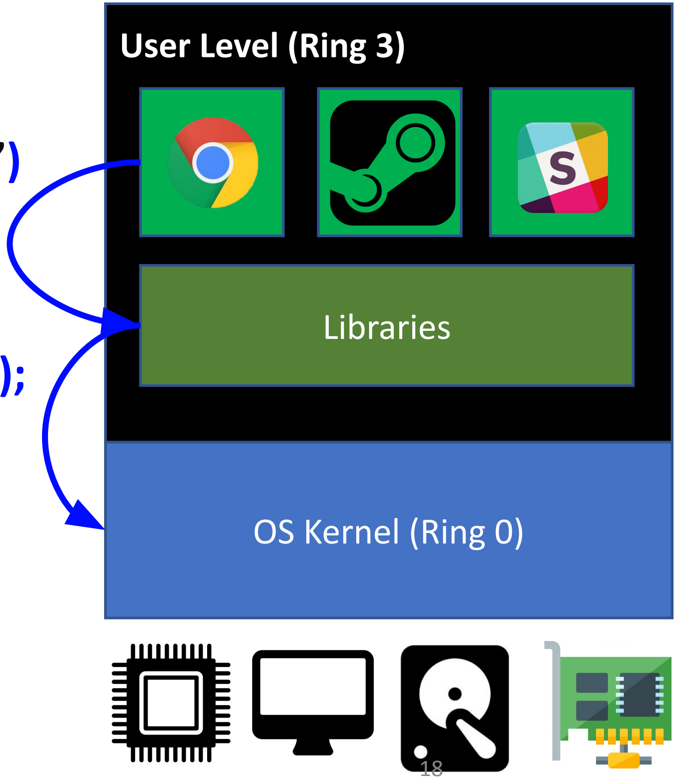
Syscall: User/Kernel communication

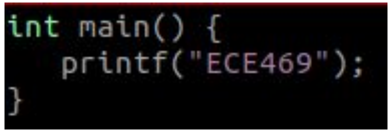
printf("ECE469")

A library call in ring 3

sys_write(1, "ECE469", 6);

A system call, **From ring 3**





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

Syscall: User/Kernel communication

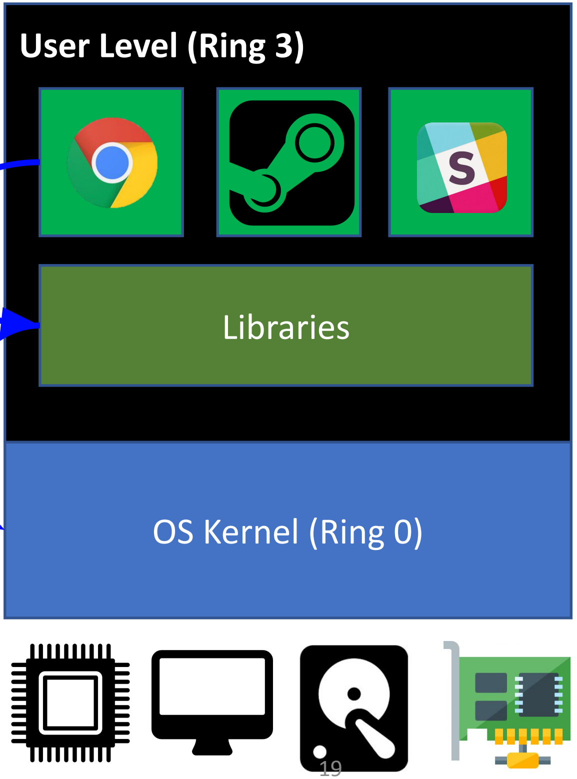
printf("ECE469")

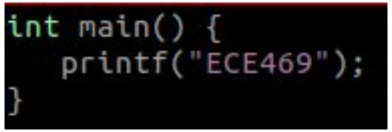
A library call in ring 3

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

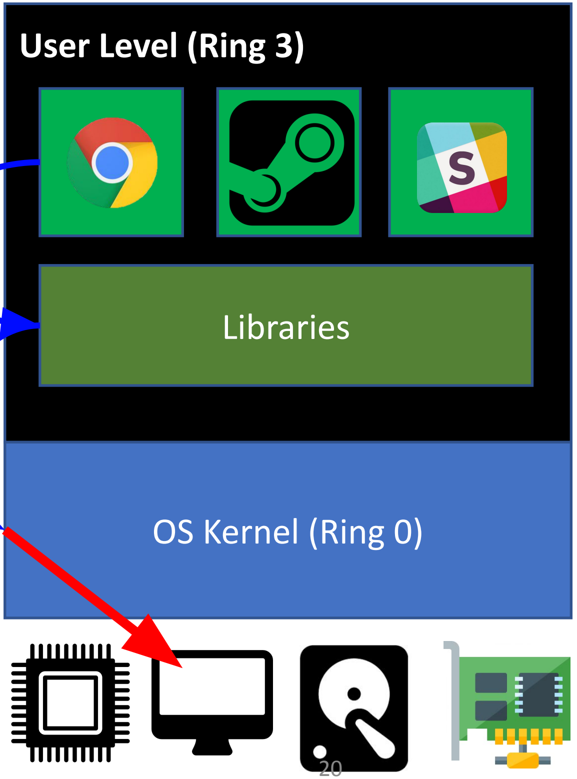
sys_write(1, "ECE469", 6);

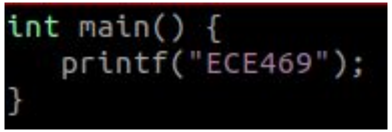
A system call, **From ring 3**

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

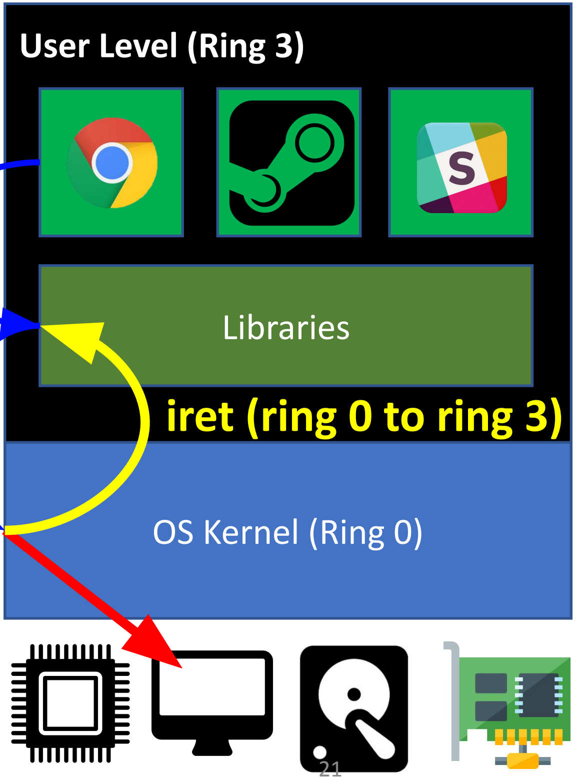
sys_write(1, "ECE469", 6);

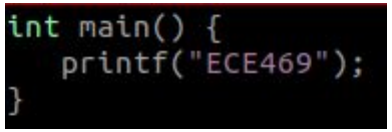
A system call, **From ring 3**

Interrupt!, switch from ring3 to ring0

A kernel function

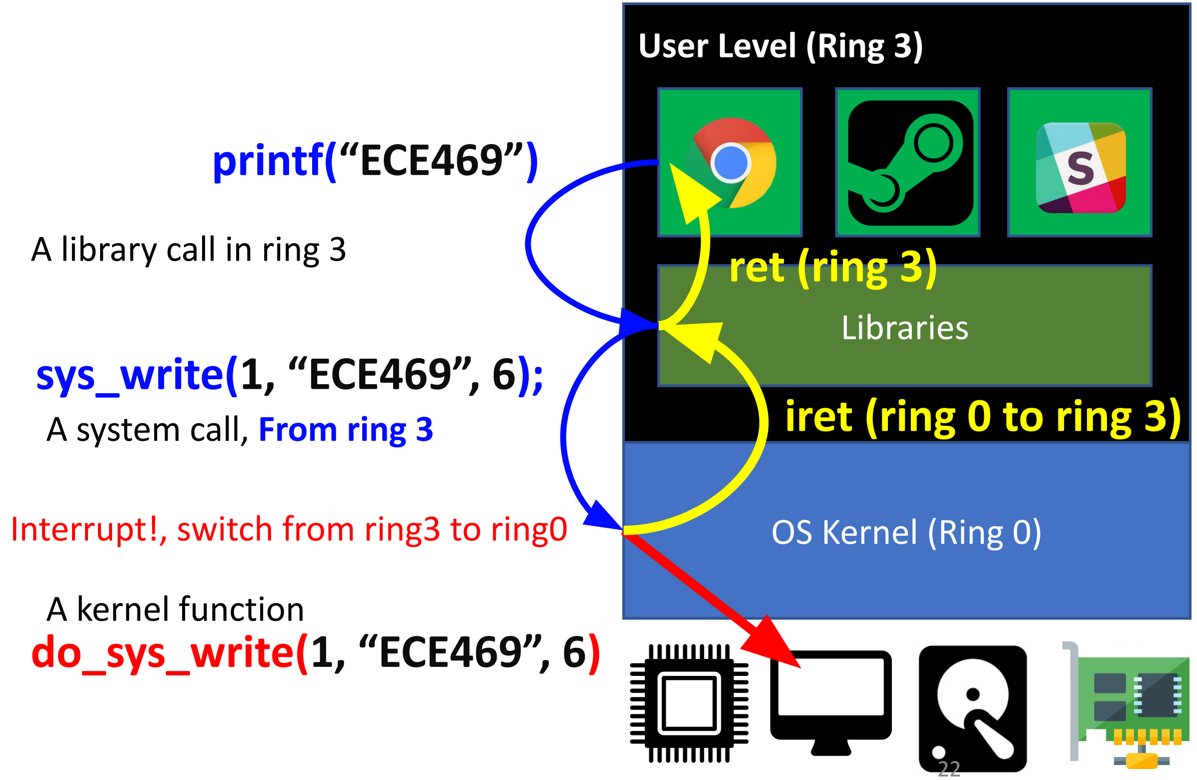
do_sys_write(1, "ECE469", 6)

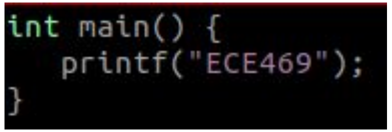




```
int main() {
    printf("ECE469");
}
```

Syscall: User/Kernel communication

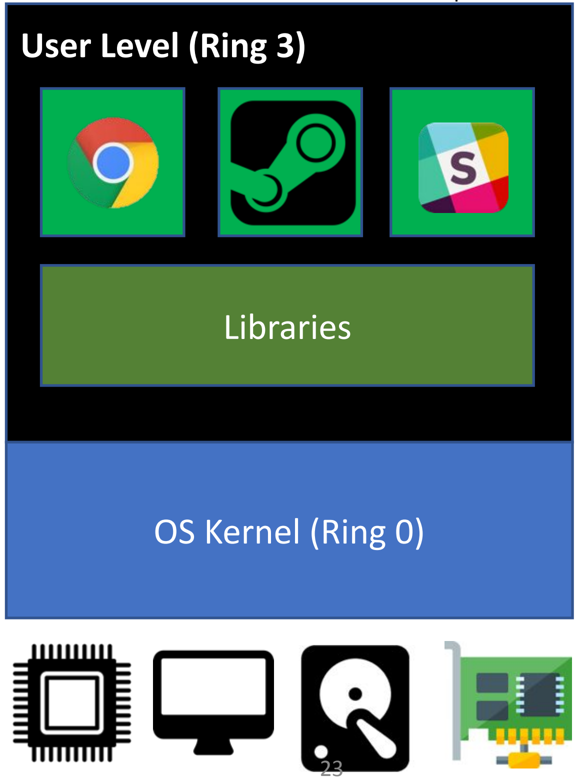




```
int main() {
    printf("ECE469");
}
```

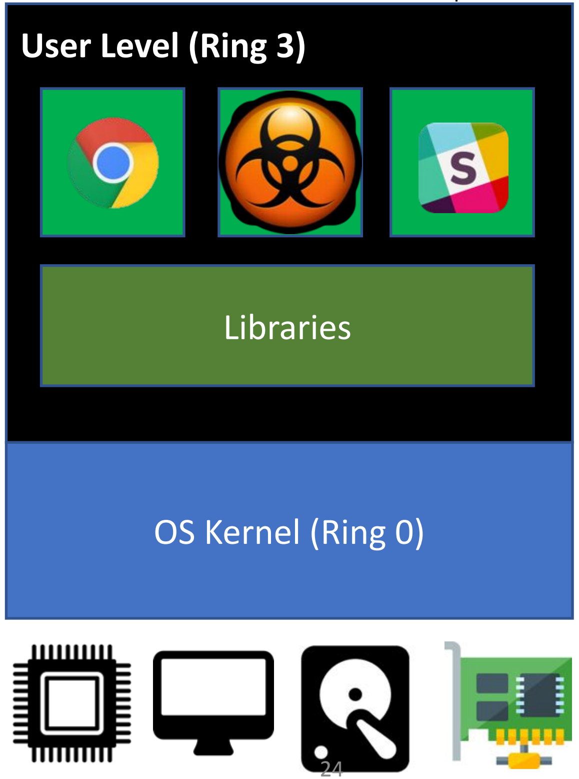
The need for syscalls?

- We cannot let a process access peripherals.



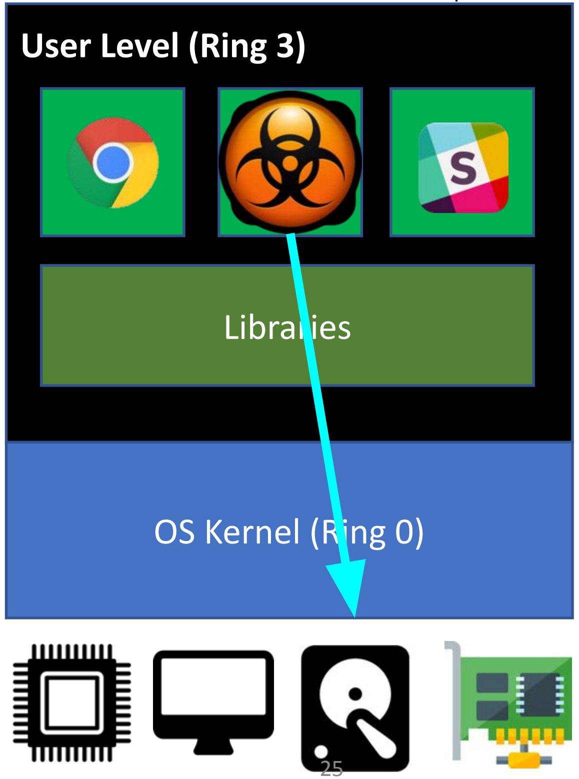
The need for syscalls?

- We cannot let a process access peripherals.



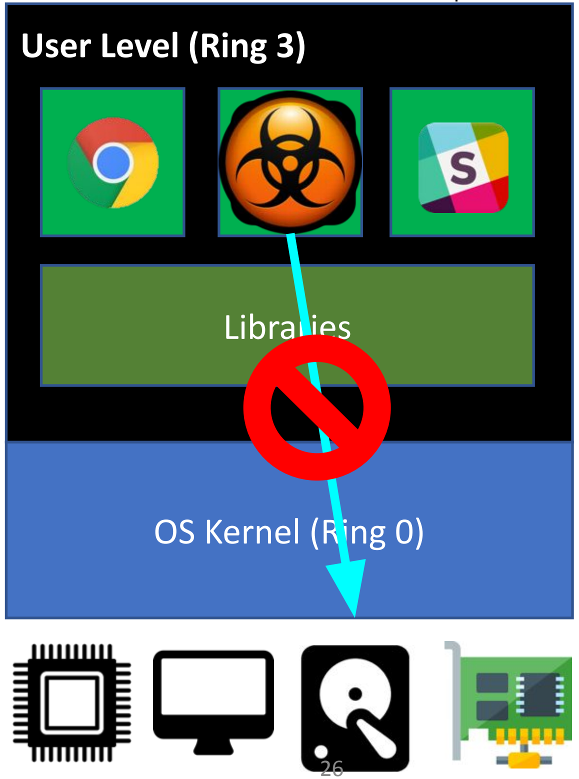
The need for syscalls?

- We cannot let a process access peripherals.



The need for syscalls?

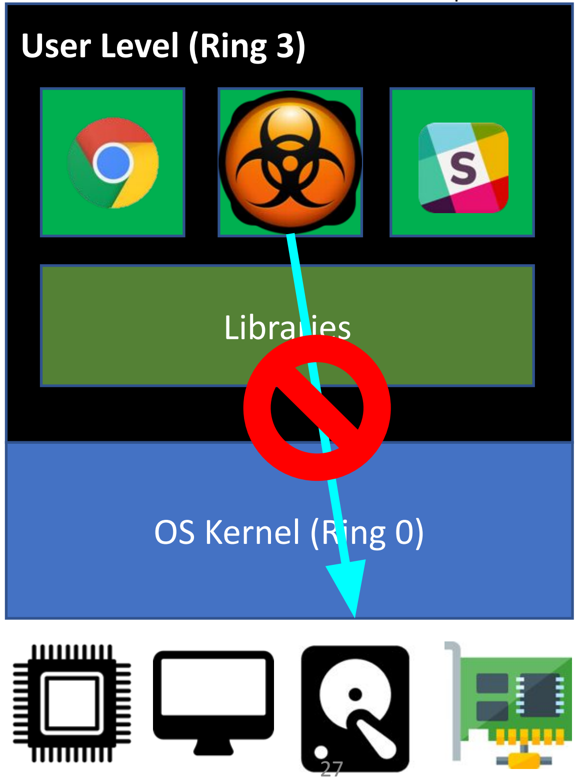
- We cannot let a process access peripherals.
- Why do we have privilege separation?
  - Security!
- We do not know what application will do
  - Do not allow dangerous operations to system
    - Flash BIOS, format disk, deleting system files, etc.
  - Let only the OS can access hardware
    - Apply access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



The need for syscalls?

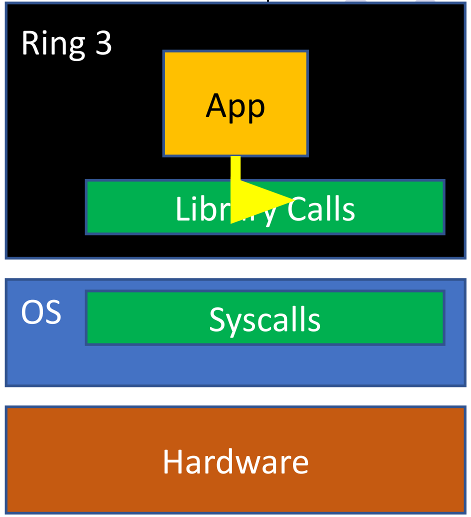
- We cannot let a process access peripherals.
- Why do we have privilege separation?
  - Security!
- We do not know what application will do
  - Do not allow dangerous operations to system
    - Flash BIOS, format disk, deleting system files, etc.
  - Let only the OS can access hardware
    - Apply access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

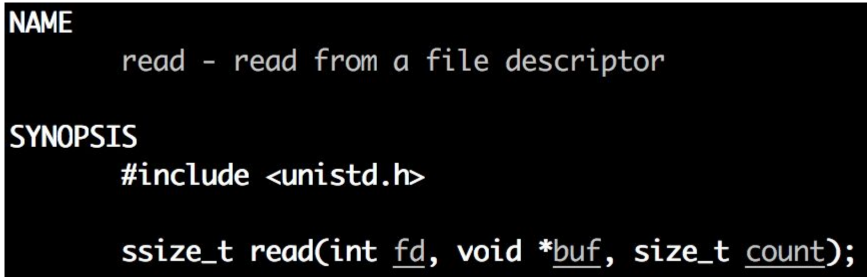
OS must mediate hardware access request from userspace, and we handle this via system calls



Library Calls v/s 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
  - 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

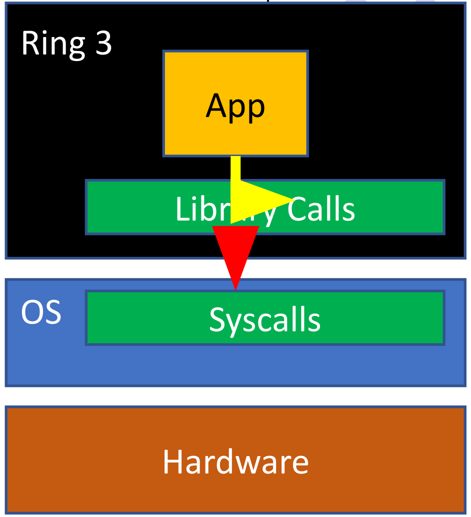


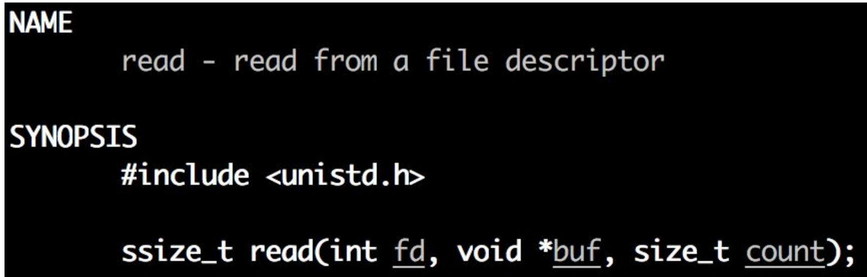


```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

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

Library Calls v/s 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
  - 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

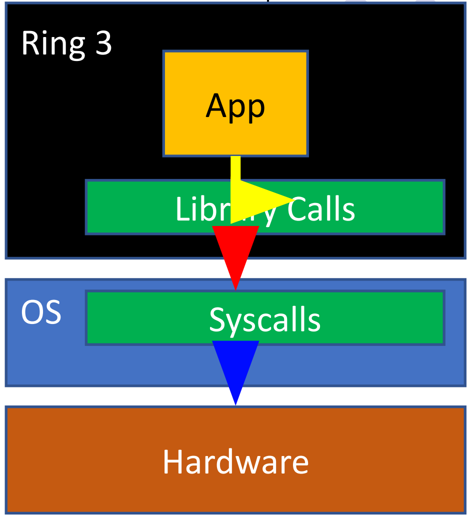


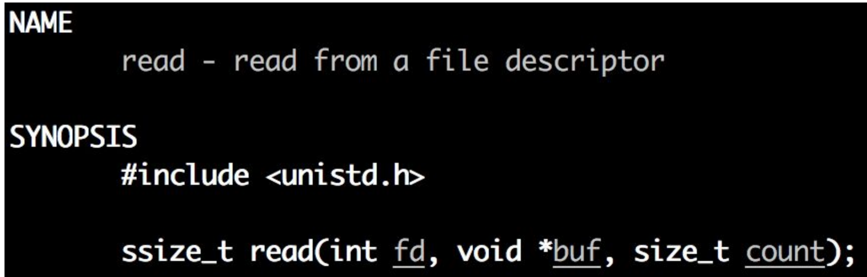


```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

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

Library Calls v/s 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
  - 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

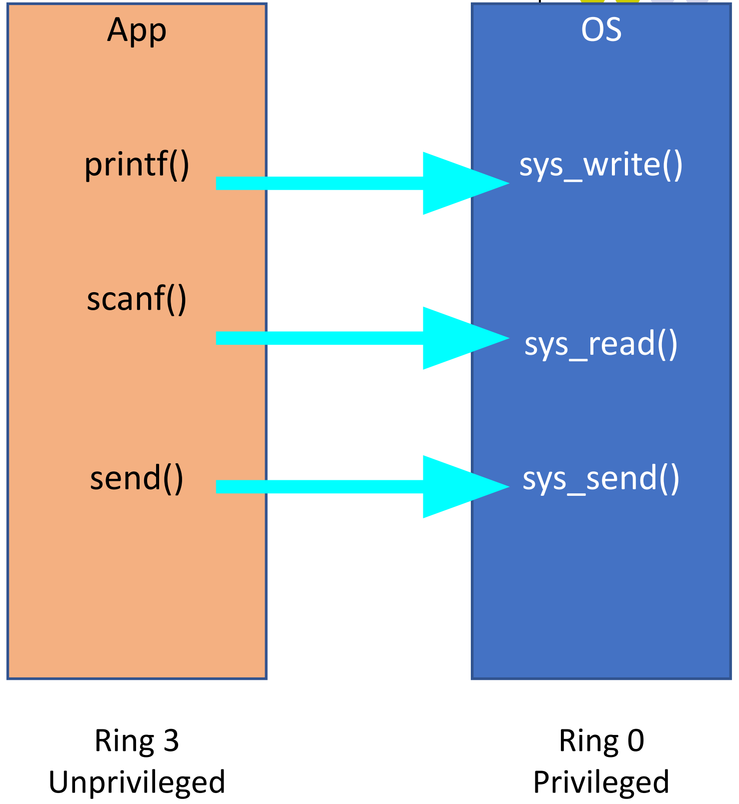




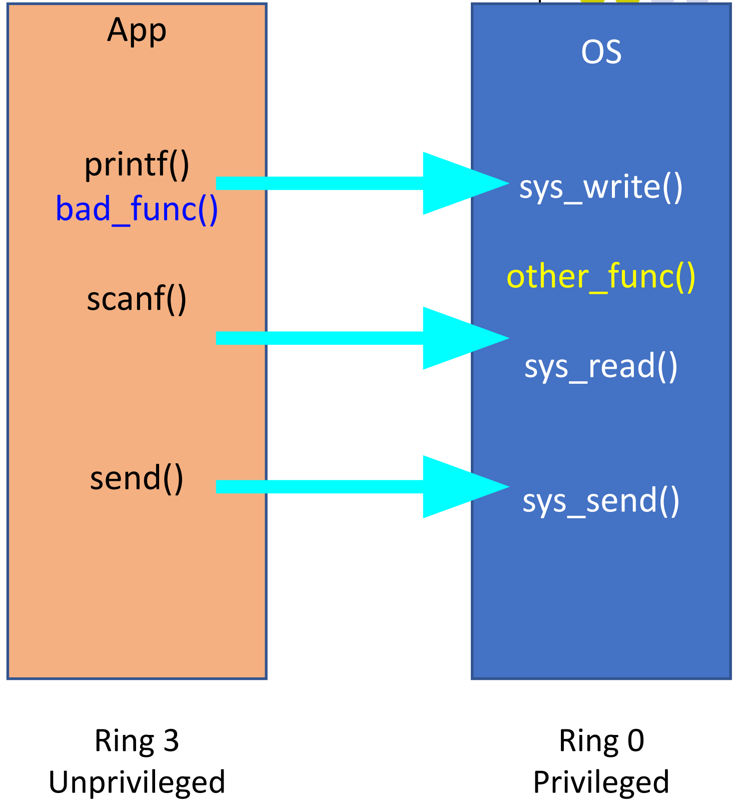
```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

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

Library Calls v/s System calls

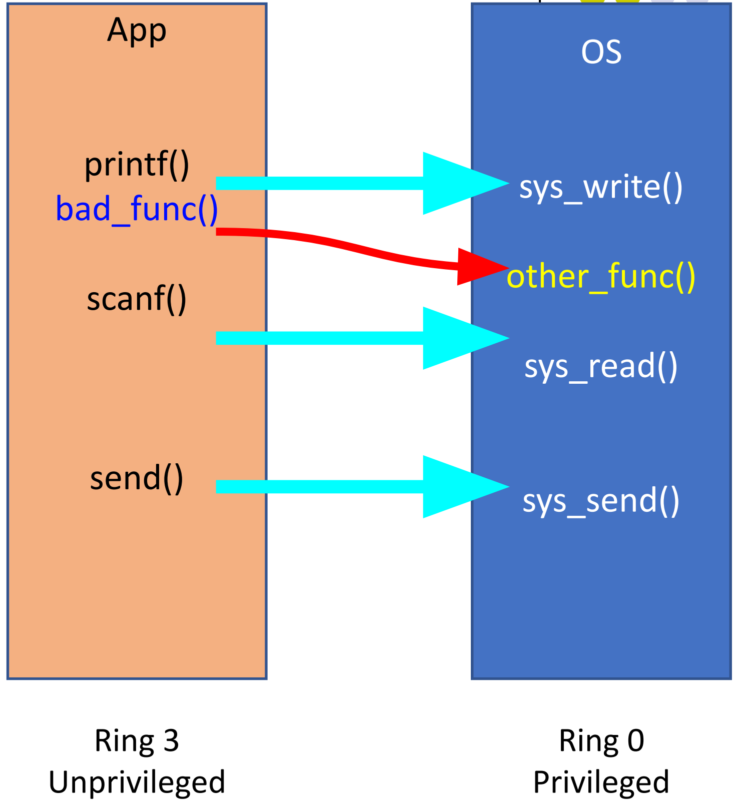
- System Calls
  - APIs available in Ring 0
  - OS’s abstraction for hardware interface for userspace
  - Called when Ring 3 application need to perform Ring 0 operations



System calls are not function calls!

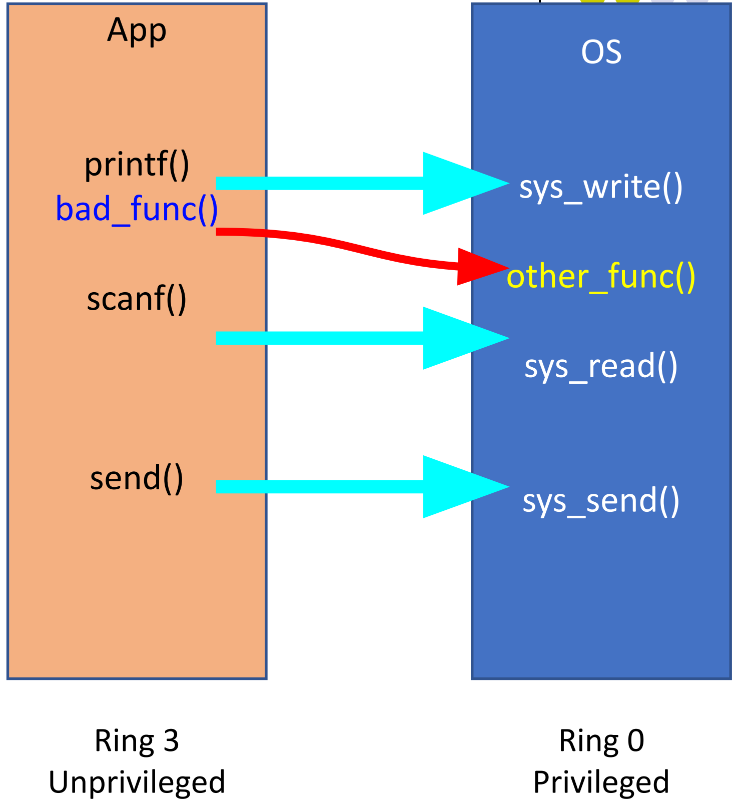


System calls are not function calls!



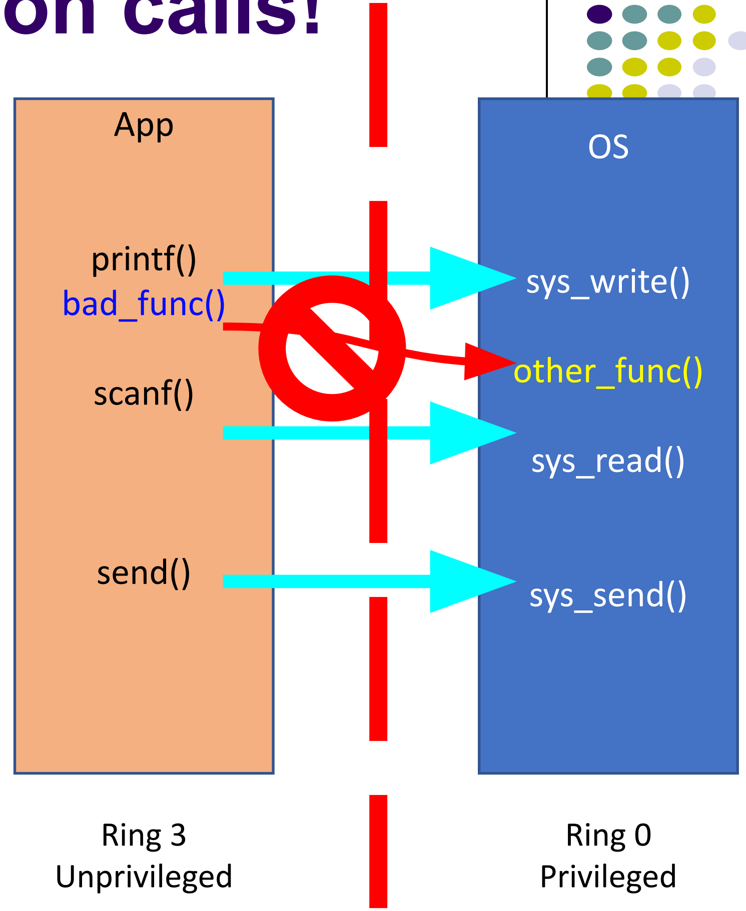
System calls are not function calls!

- Application should not call arbitrary OS functions
  - If so, app can do all operations that OS can do; privilege separation is meaningless!



System calls are not function calls!

- Application should not call arbitrary OS functions
  - If so, app can do all operations that OS can do; privilege separation is meaningless!
- How can we protect this, in other words, how can we let apps invoke system calls but no other OS functions?

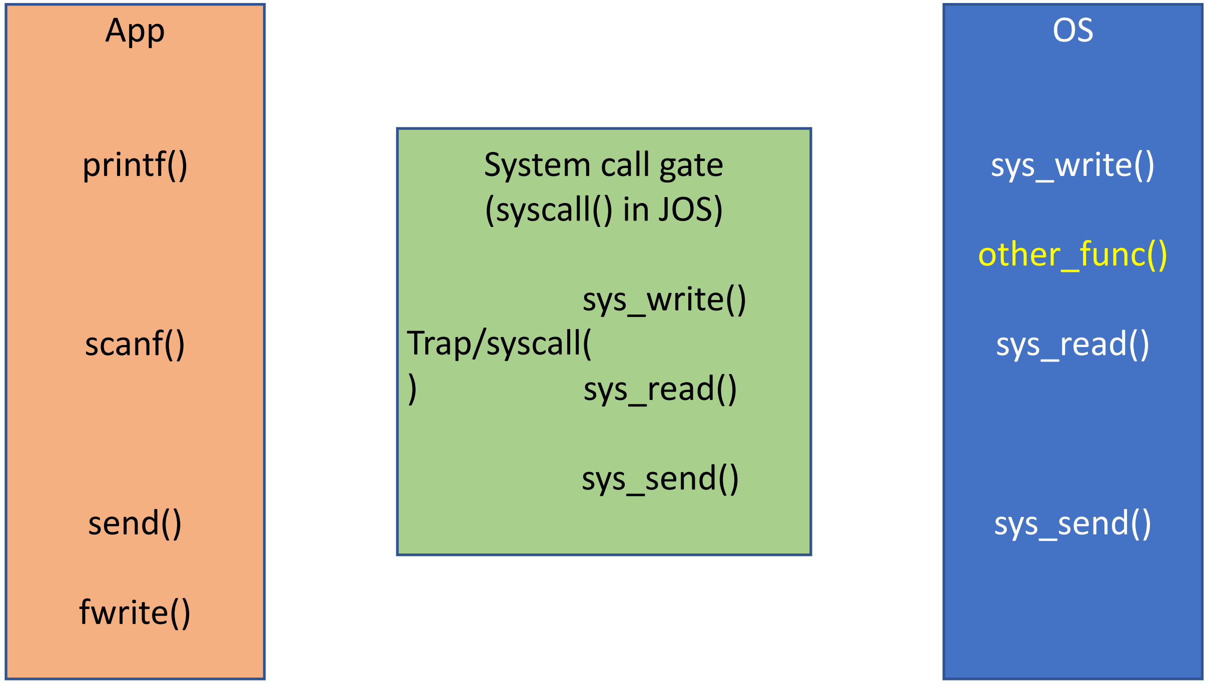


System call Design via call gate

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

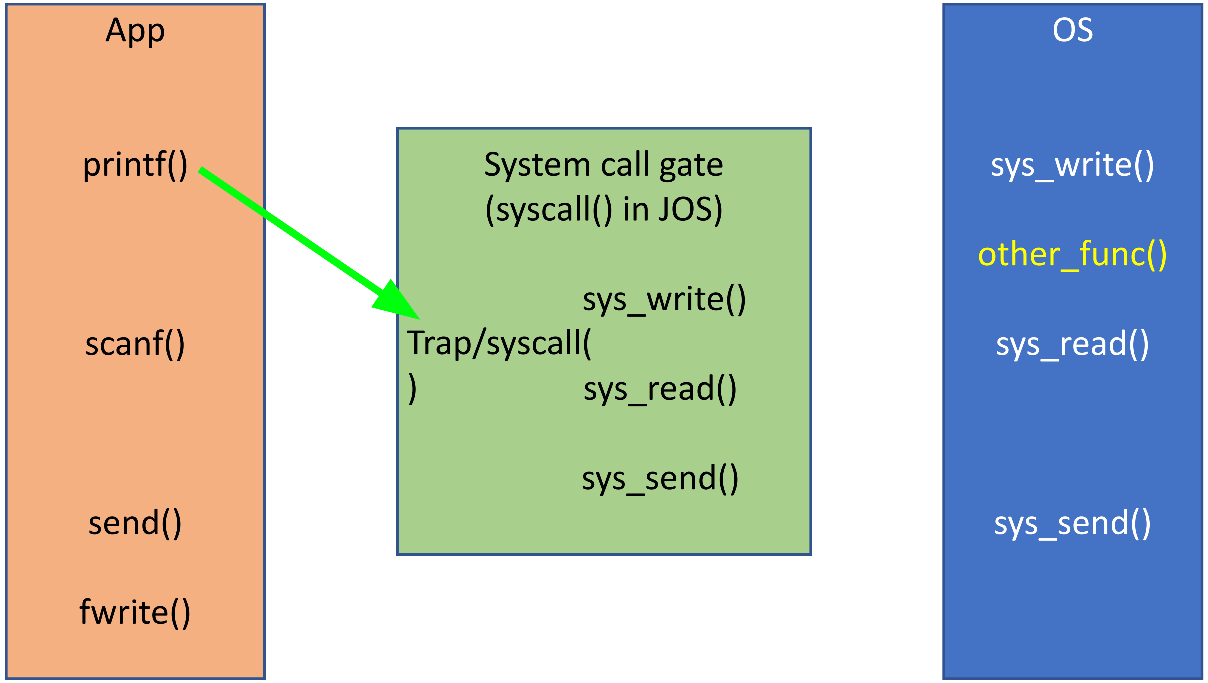
System call Design via call gate

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



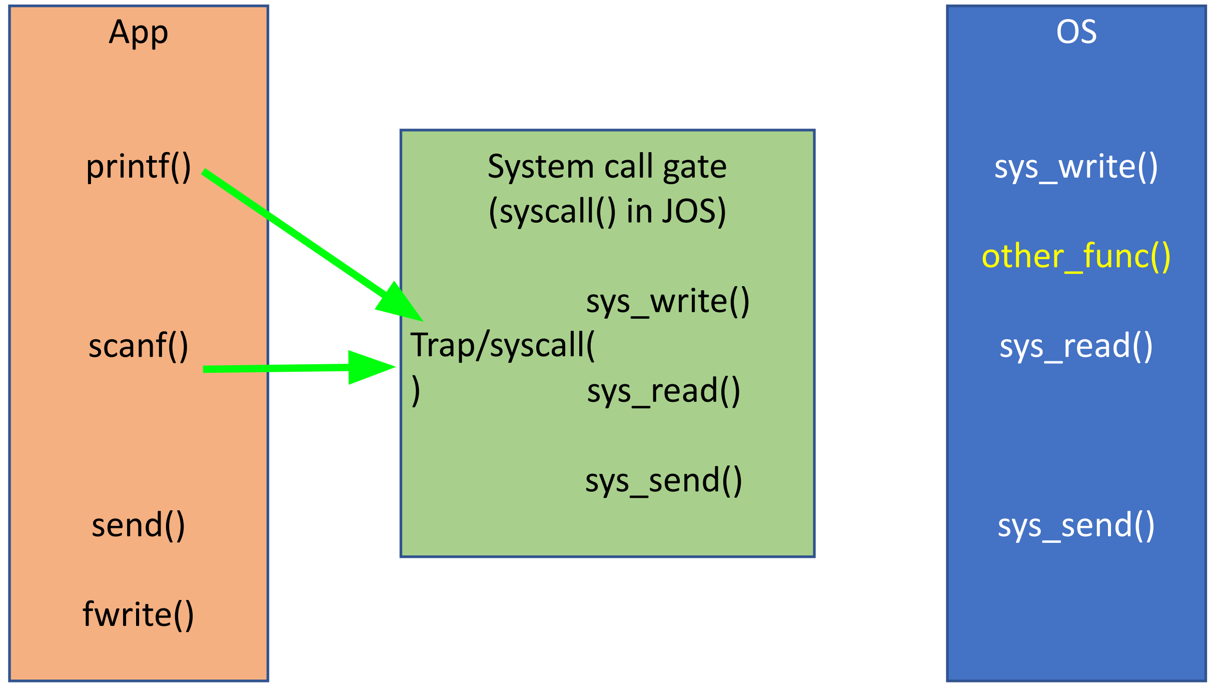
System call Design via call gate

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



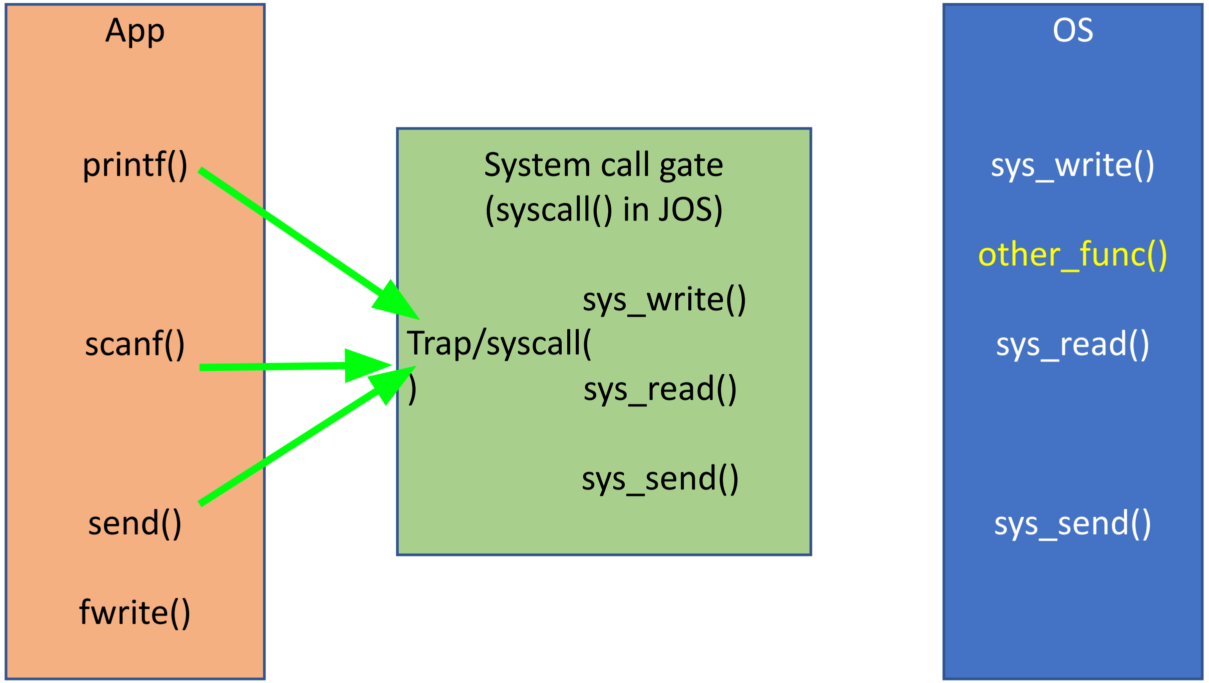
System call Design via call gate

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



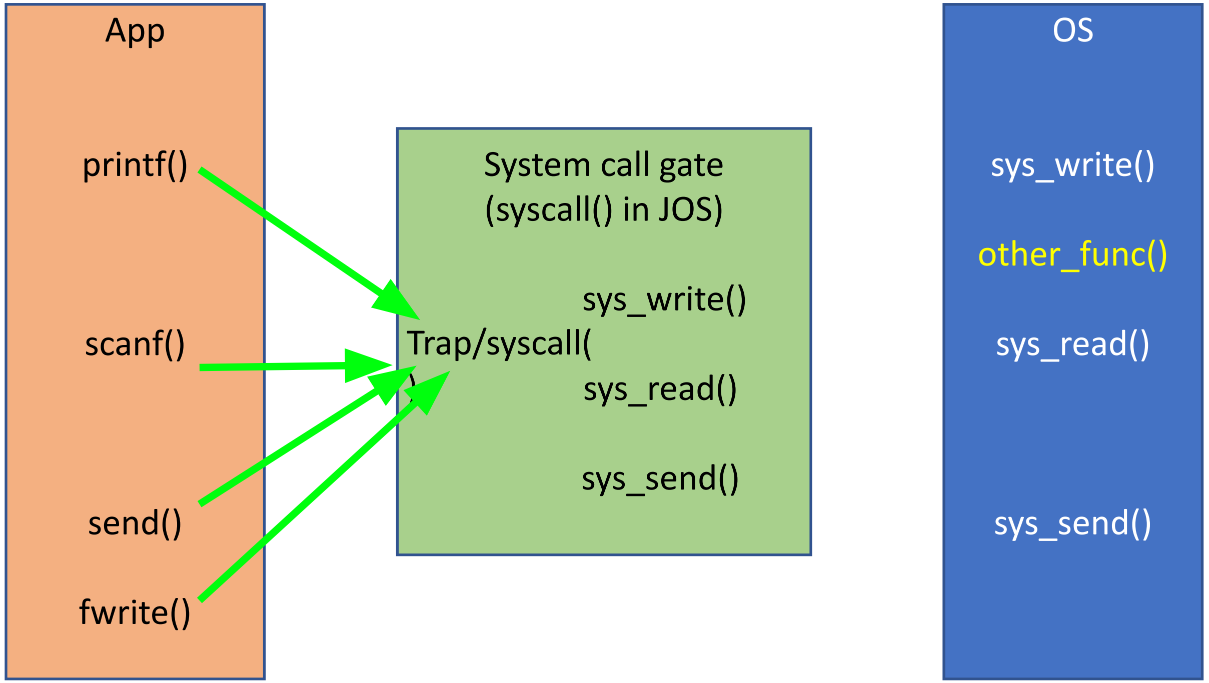
System call Design via call gate

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



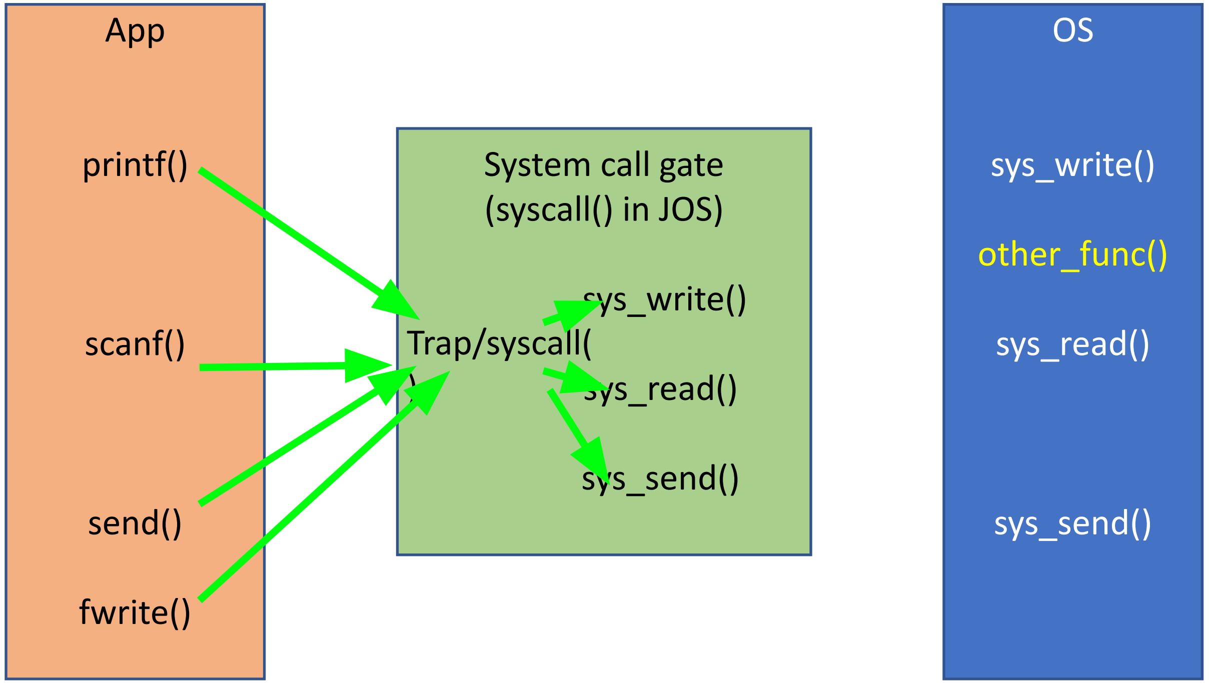
System call Design via call gate

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



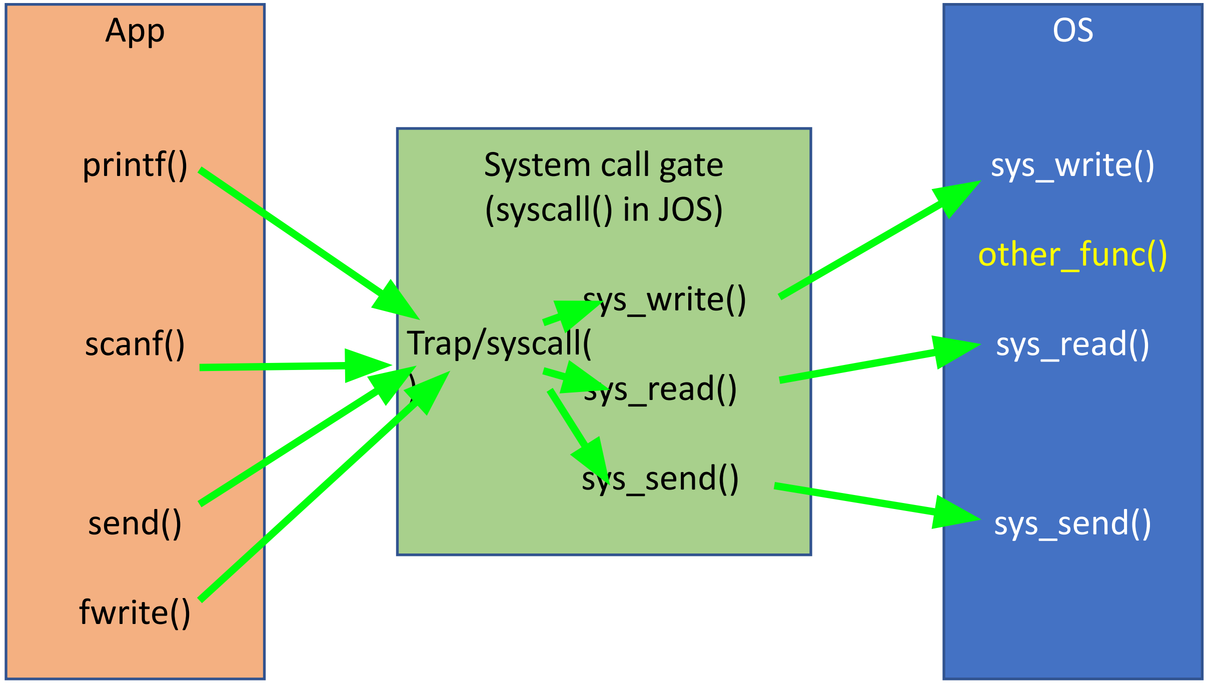
System call Design via call gate

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



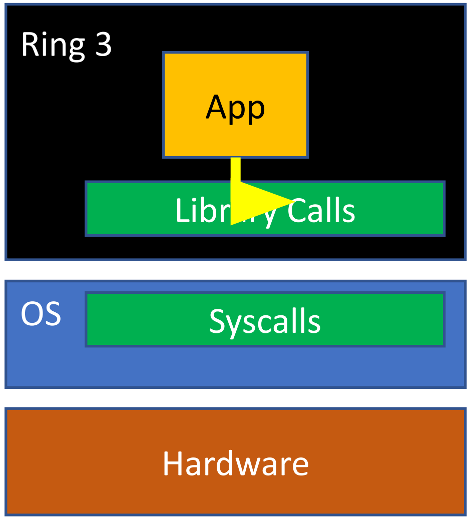
System call Design via call gate

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



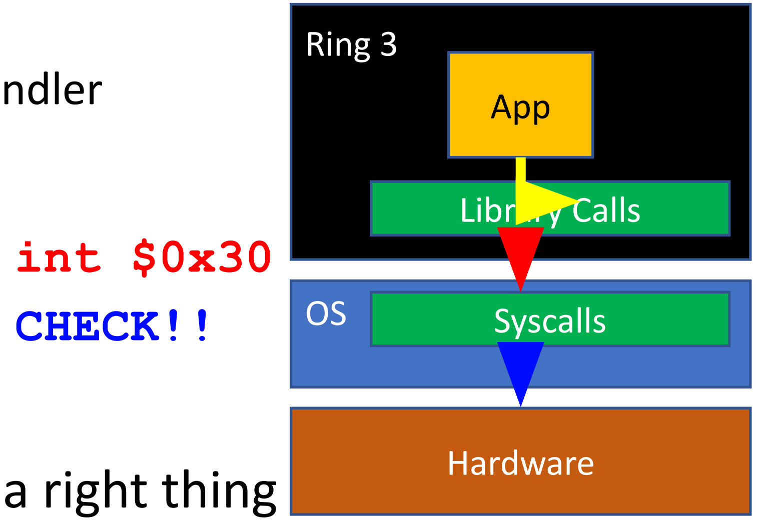
Call gate via Interrupt Handler

- 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..



Call gate via Interrupt Handler

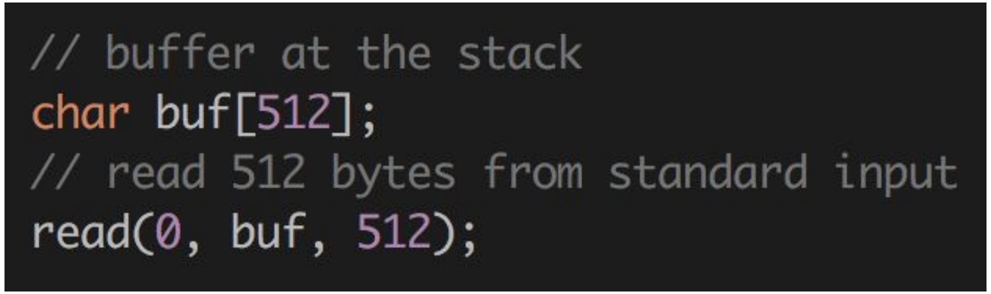
- Call gate
  - System call can be invoked only with trap handler
    - `int $0x30` – in JOS
    - `int $0x80` – in Linux (32-bit)
    - `int $0x2e` – in Windows (32-bit)
    - `sysenter/sysexit` (32-bit)
    - `syscall/sysret` (64-bit)



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

Why should we check arguments?

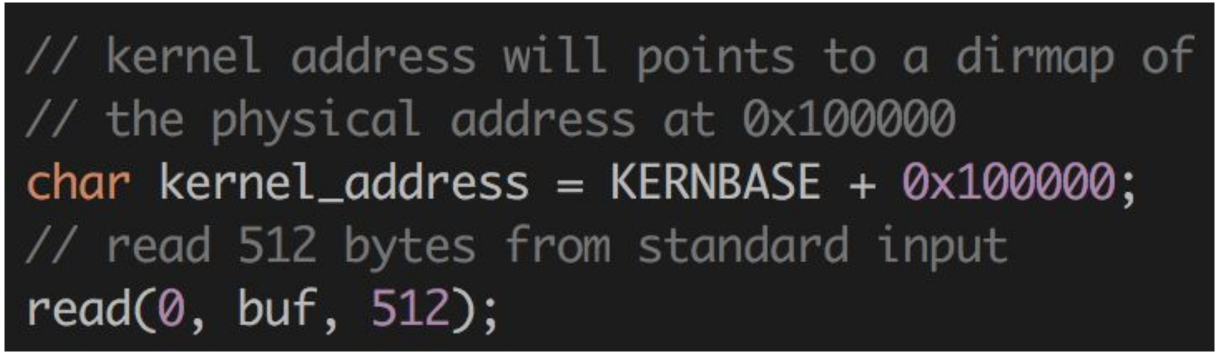
- How can we protect ‘`read()`’ system call?
  - `read(int fd, void *buf, size_t count)`
  - Read `count` bytes from a file pointed by `fd` and store those in `buf`
- Usage



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

Why should we check arguments?

- Problem: what will happen if we call…

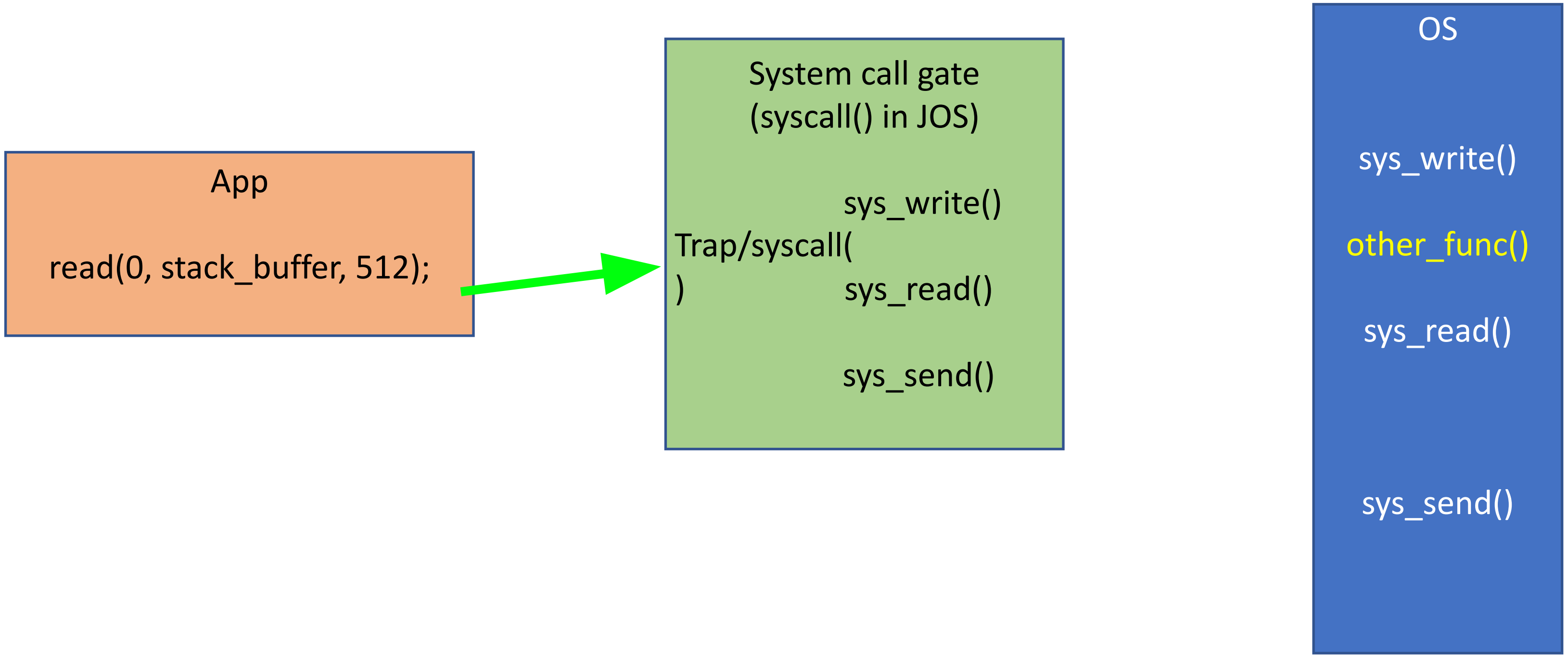


```c
// kernel address will points to a dirmap of
// the physical address at 0x100000
char kernel_address = KERNBASE + 0x100000;
// read 512 bytes from standard input
read(0, buf, 512);
```

- This will **overwrite kernel code** with your keystroke typing..
  - Changing kernel code from Ring 3 is possible!

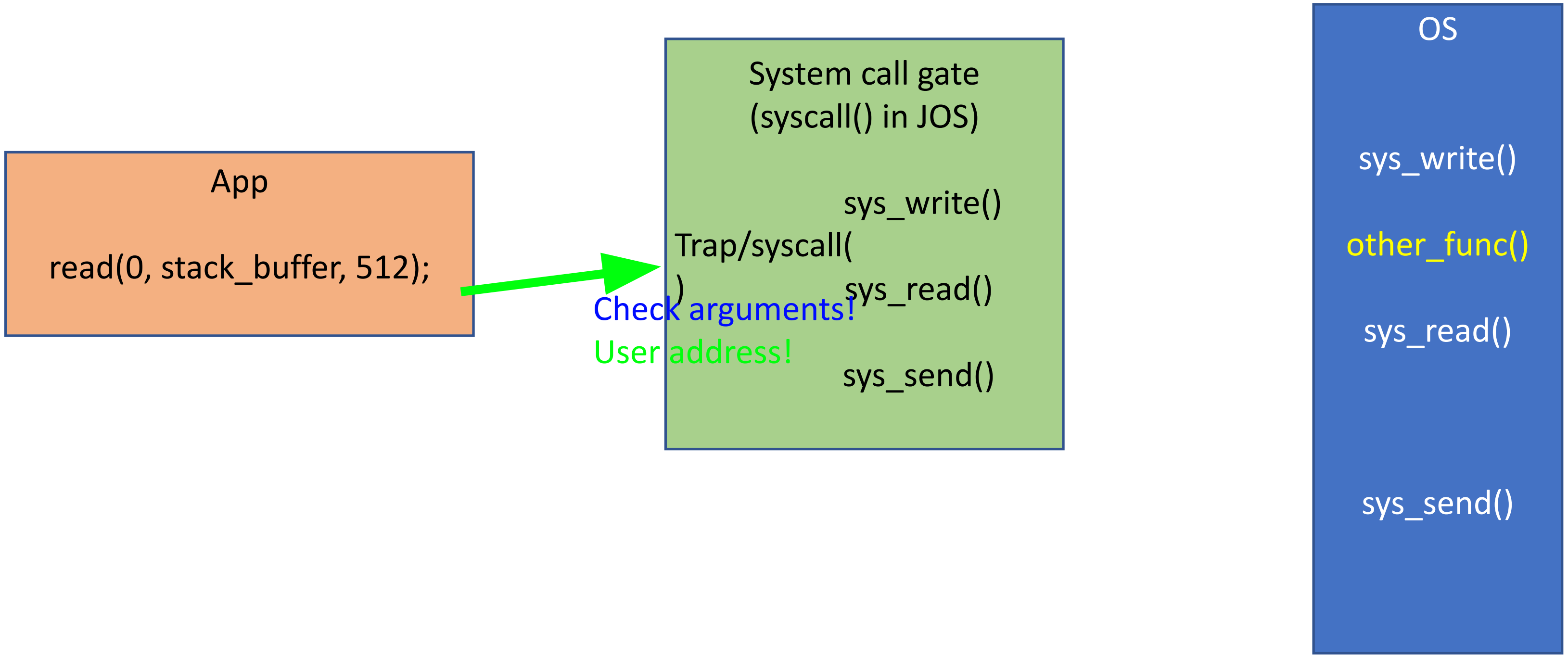
Checking arguments for syscalls

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



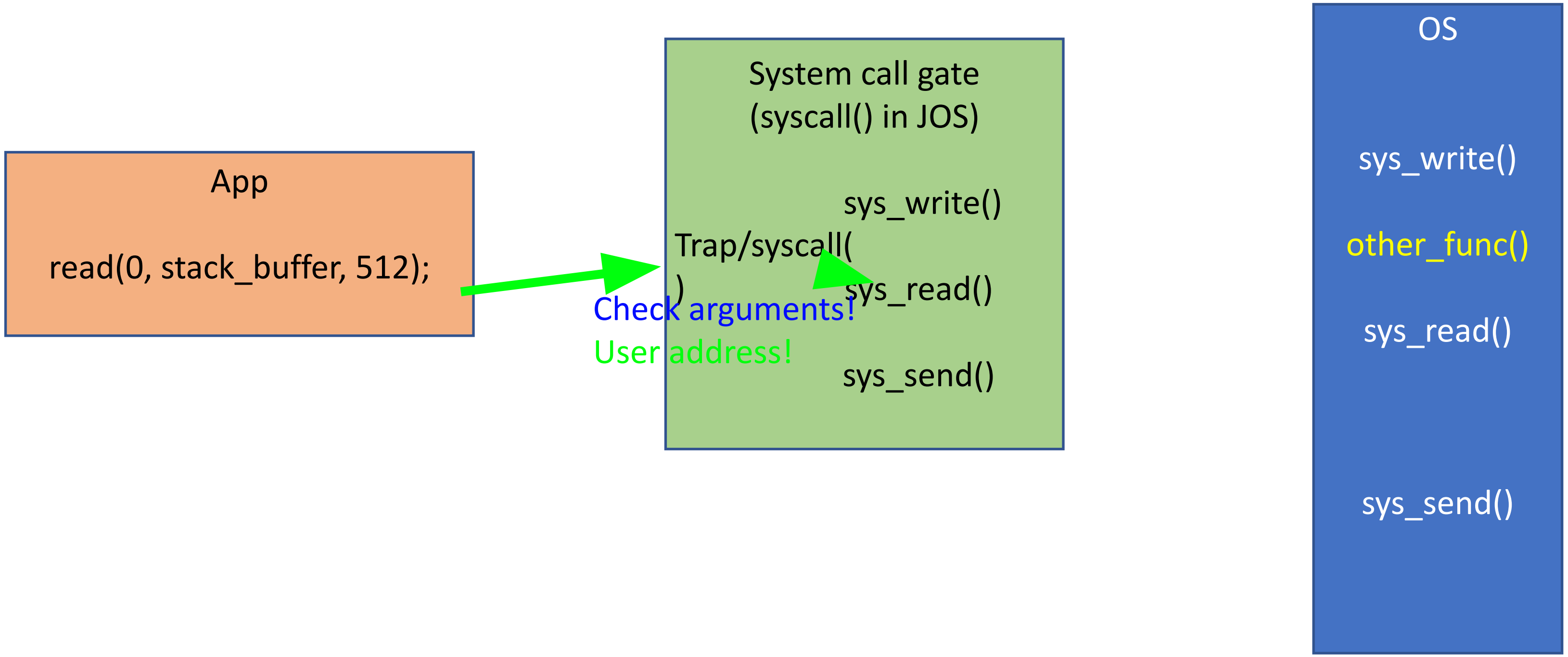
Checking arguments for syscalls

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



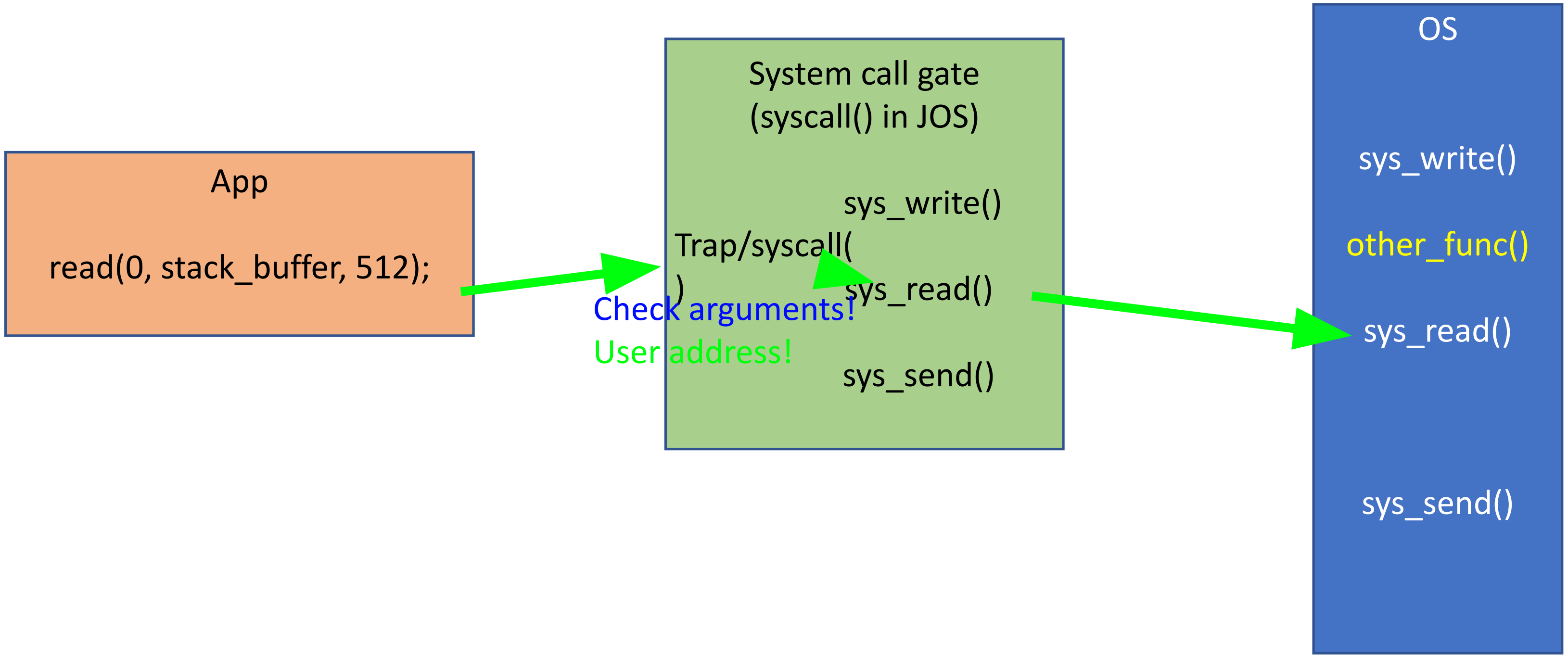
Checking arguments for syscalls

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



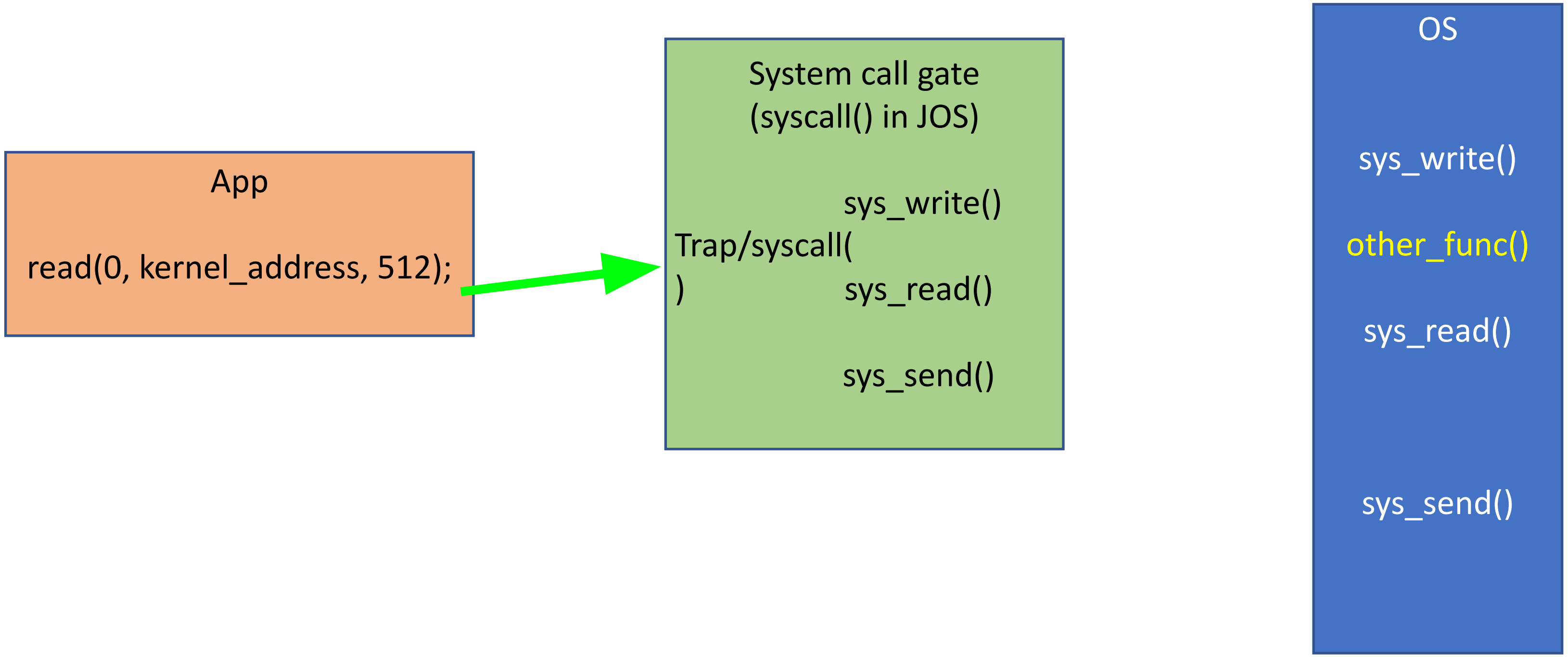
Checking arguments for syscalls

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



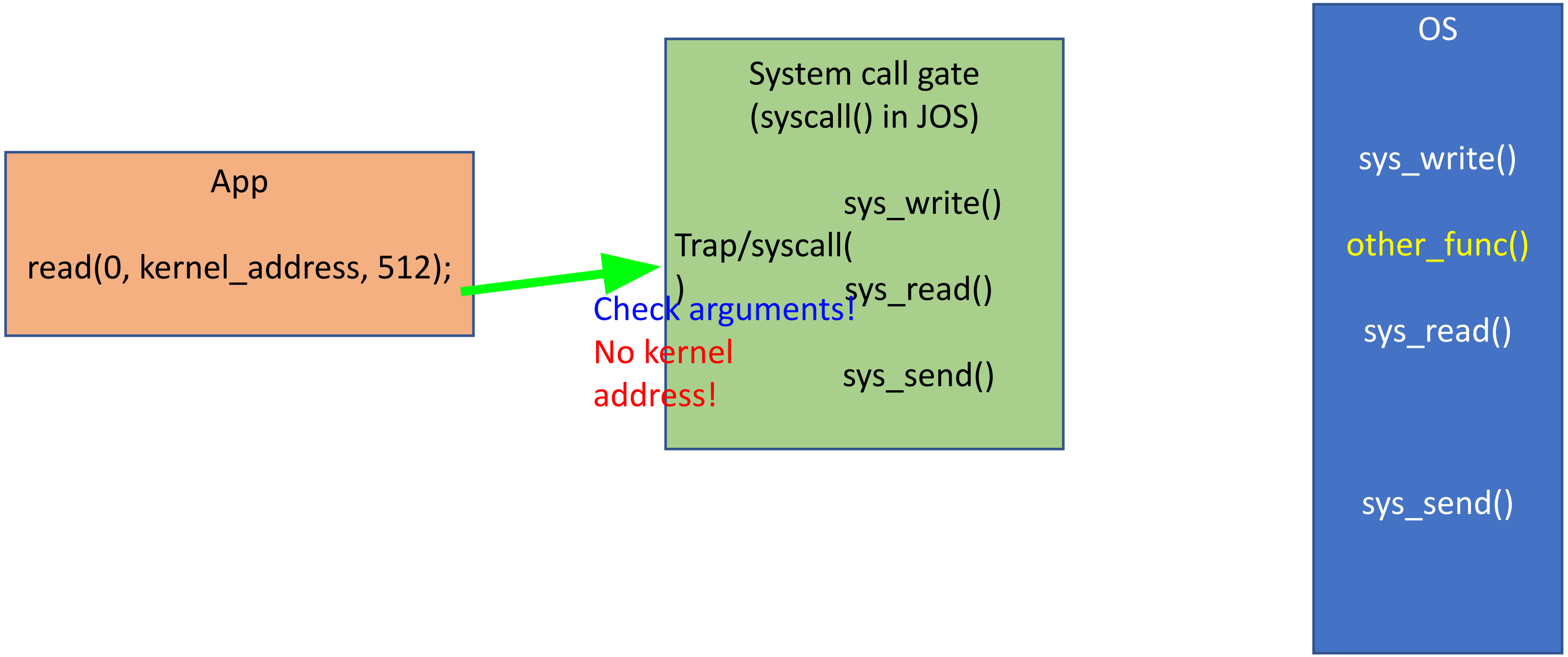
Checking arguments for syscalls

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



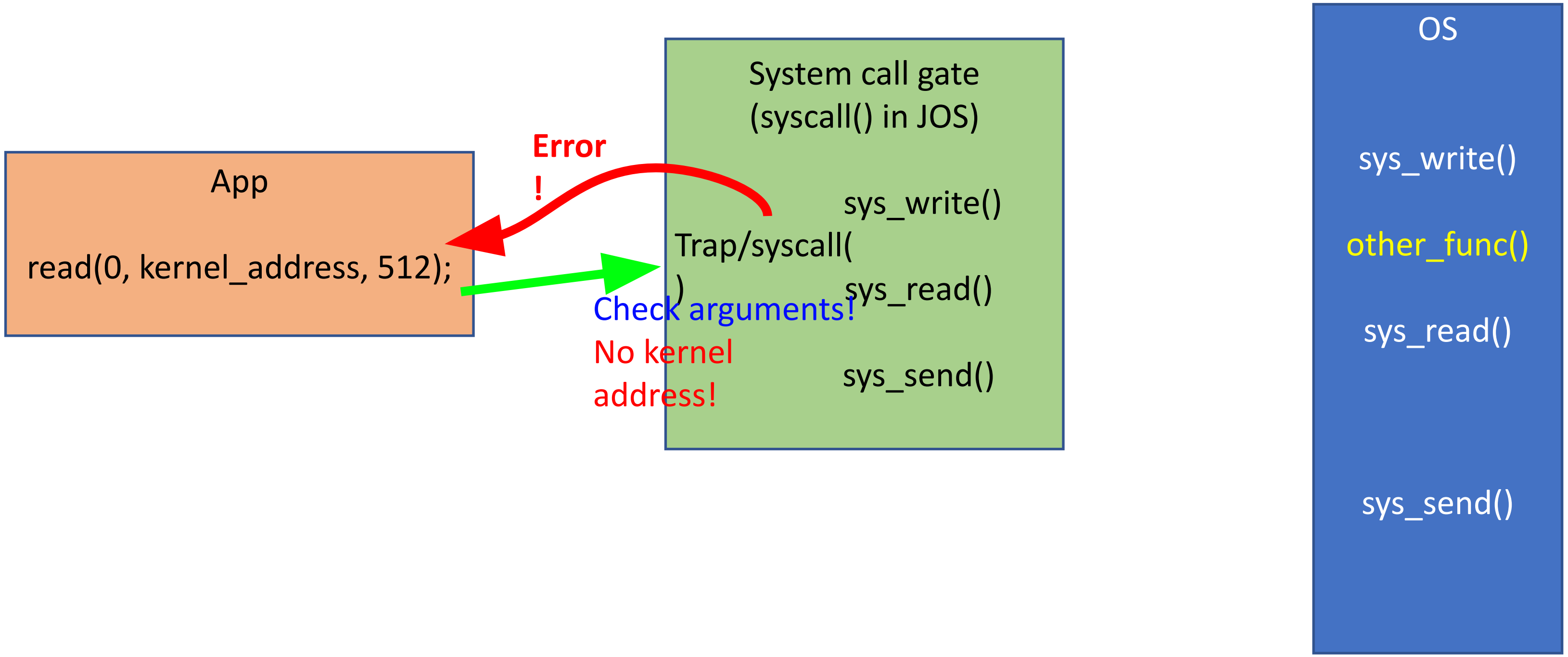
Checking arguments for syscalls

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



Checking arguments for syscalls

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



Test: using ltrace and strace

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

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

// read 512 bytes from stdin to kernel.
ret = read(0, (void*)0xffffffff01000000,512);

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

Summary: Syscalls

- 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

Faults

- 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 faults

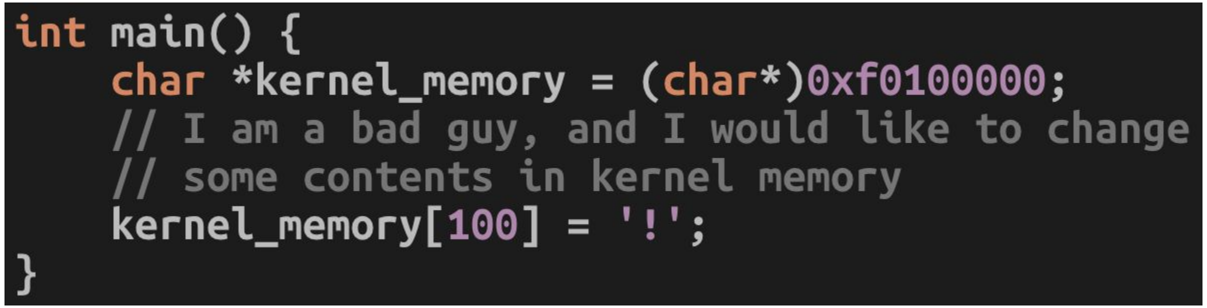
- Occurs when paging (address translation) fails

Page faults

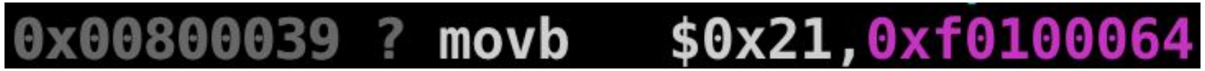
- Occurs when paging (address translation) fails
  - Access from user but `!(pte&PTE_U)`: protection violation

Page faults

- Occurs when paging (address translation) fails
  - Access from user but `!(pte&PTE_U)`: protection violation



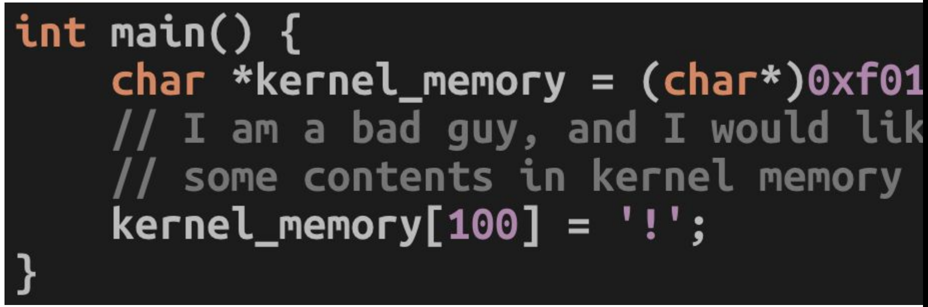
```
int main() {
    char *kernel_memory = (char*)0xf0100000;
    // I am a bad guy, and I would like to change
    // some contents in kernel memory
    kernel_memory[100] = '!';
}
```



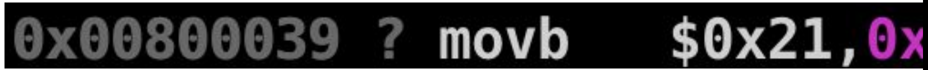
```
0x00800039 ? movb    $0x21,0xf0100064
```

Page faults

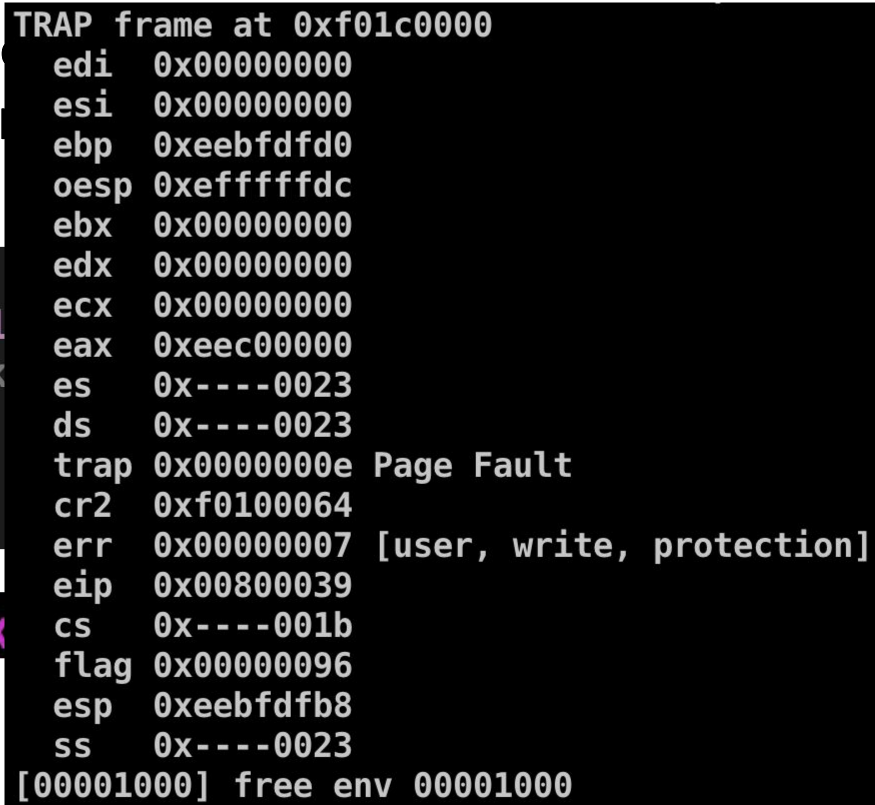
- Occurs when paging (address translati
  - Access from user but `!(pte&PTE_U)`:



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



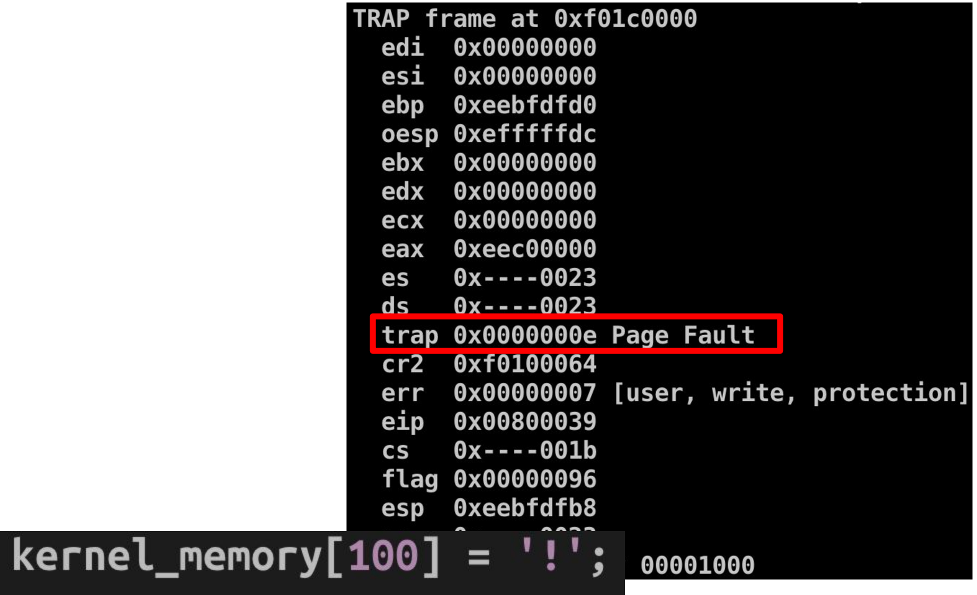
```
0x00800039 ? movb    $0x21,0x
```



```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  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
```

What does CPU do on a page fault?

- CPU let OS know why and where such a page fault happened

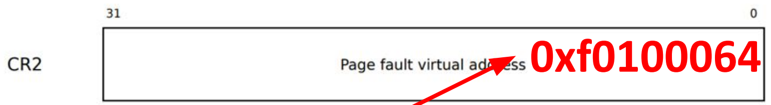


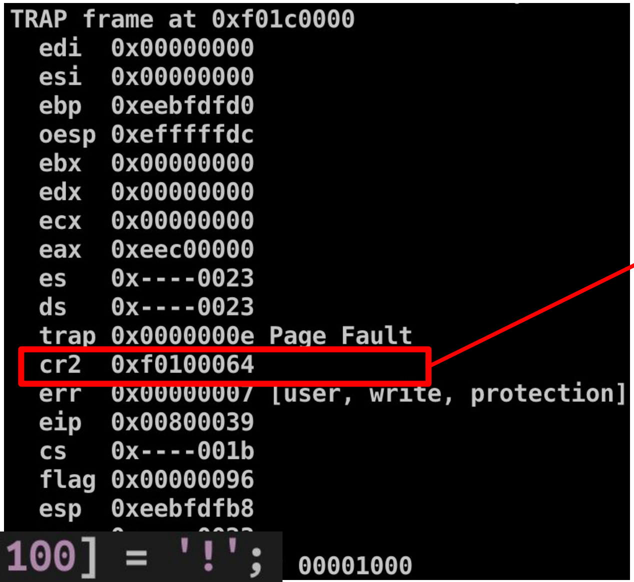
```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xf0100064
  err  0x00000007 [user, write, protection]
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

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

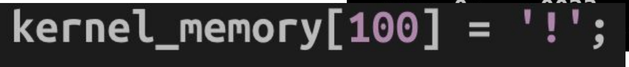
What does CPU do on a page fault?

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





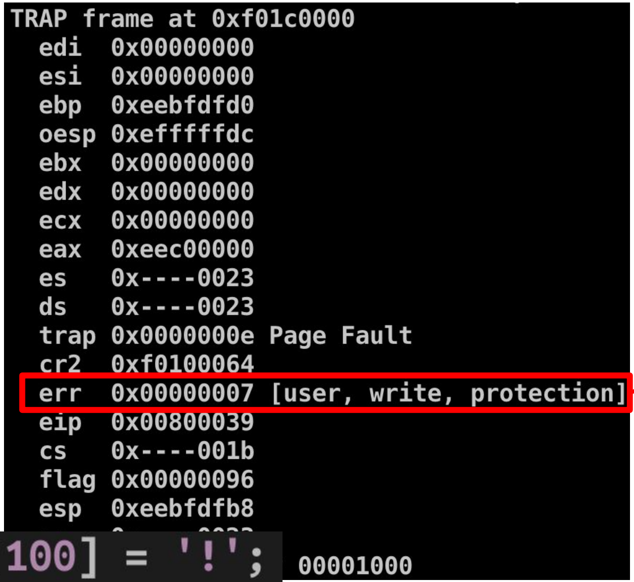
```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xf0100064
  err  0x00000007 [user, write, protection]
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



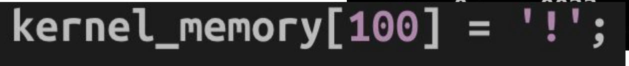
```
kernel_memory[100] = '!';
```

What does CPU do on a page fault?

- 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

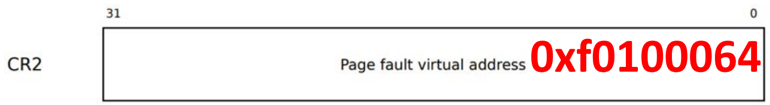


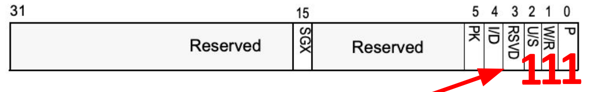
```
TRAP frame at 0xf01c0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0xf0100064
  err  0x00000007 [user, write, protection]
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



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

```
00001000
```





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.

How does OS handle page fault?

- User program accesses 0xf0100064

How does OS handle page fault?

- 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

How does OS handle page fault?

- 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

How does OS handle page fault?

- 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)

How does OS handle page fault?

- 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)

How does OS handle page fault?

- 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)

How does OS handle page fault?

- 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

How does OS handle page fault?

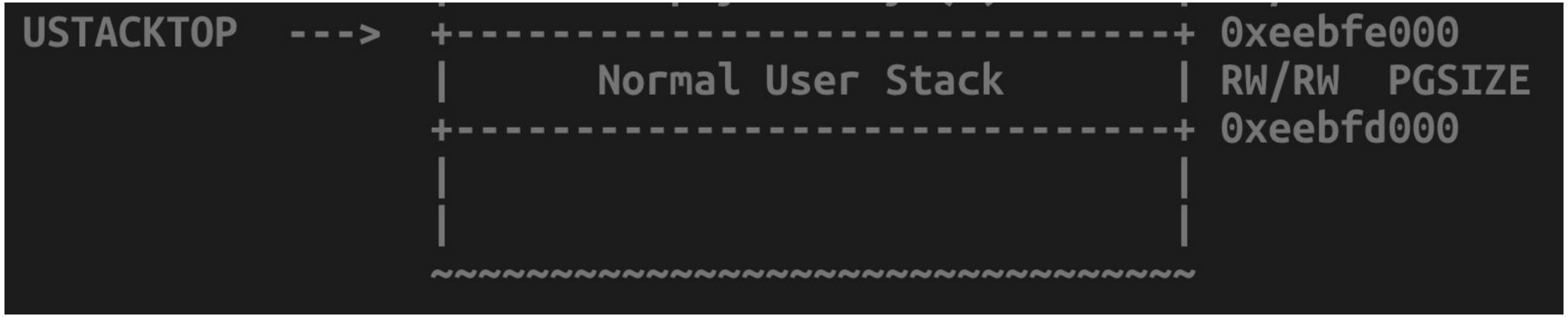
- User program accesses 0xf0100064

- 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)

Page fault example (2): Handling call stack

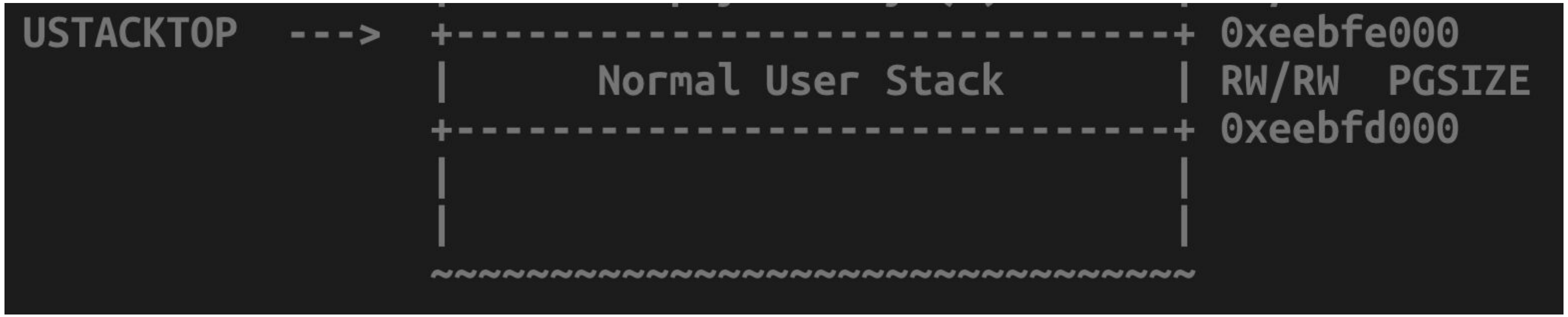
- inc/memlayout.h
- We allocate one (1) page for the user stack



```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW  PGSIZE
                 +------------------------------+ 0xeebfd000
                 |                              |
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

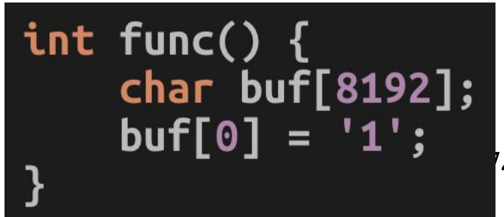
Page fault example: Handling call stack

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



```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW   PGSIZE
                 +------------------------------+ 0xeebfd000
                 |                              |
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

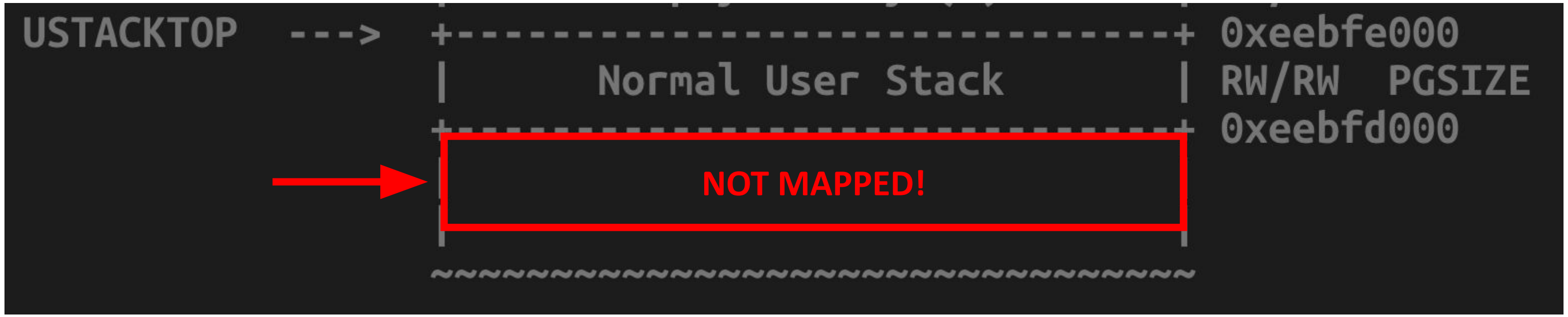
- If you use a large local variable on the stack
  - Stack overflow (stack grows down…)



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

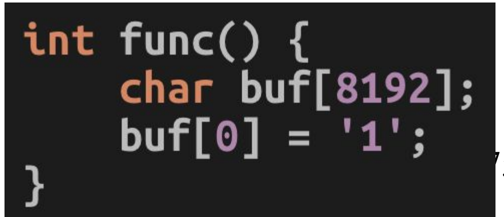
Page fault example: Handling call stack

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



```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW  PGSIZE
                 +------------------------------+ 0xeebfd000
                 |                              |
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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



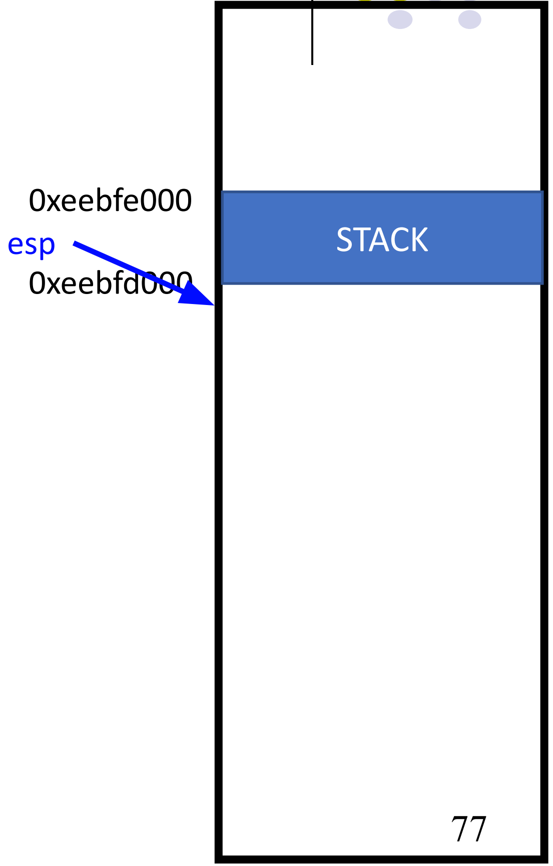
```
int func() {
    char buf[8192];
    buf[0] = '1';
}
```

Expand stack automatically

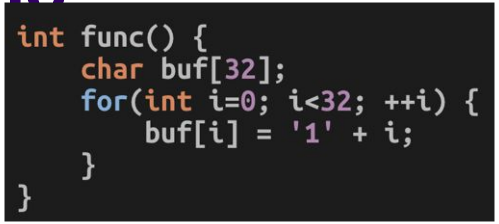
- Can we detect such an access and allocate a new page for the stack automatically?
  - Yes
- We will utilize ‘Page Fault’
- Observations
  - Stack overflow would be sequential (access pages adjacent to the stack)
  - We should catch both read/write access (both should fault)

Expand stack automatically

- Stack ends at `0xeebfd000`
- Suppose the current value of `esp` (stack) is
  - `0xeebfd010`

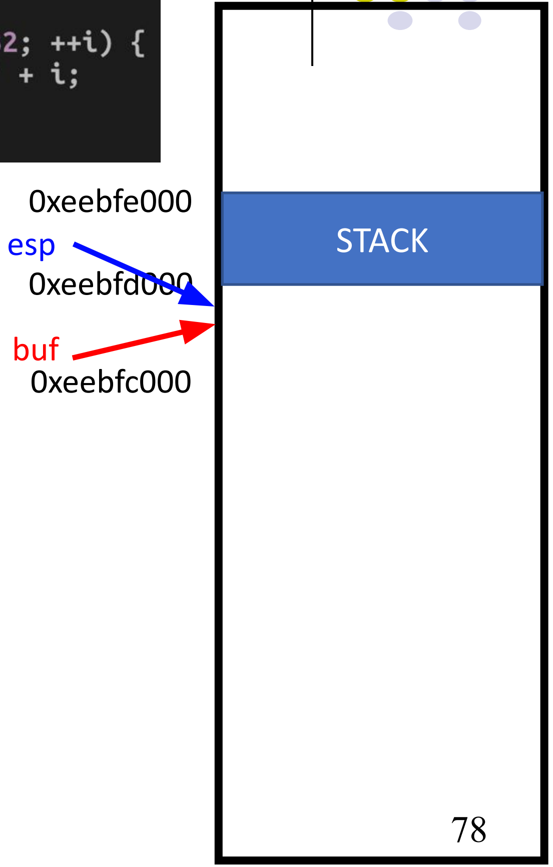


Expand stack automatically

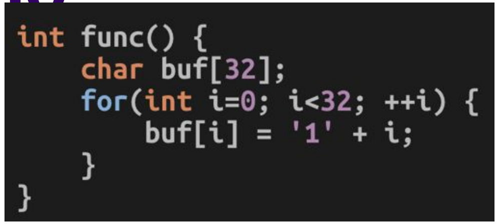


```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

- 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`

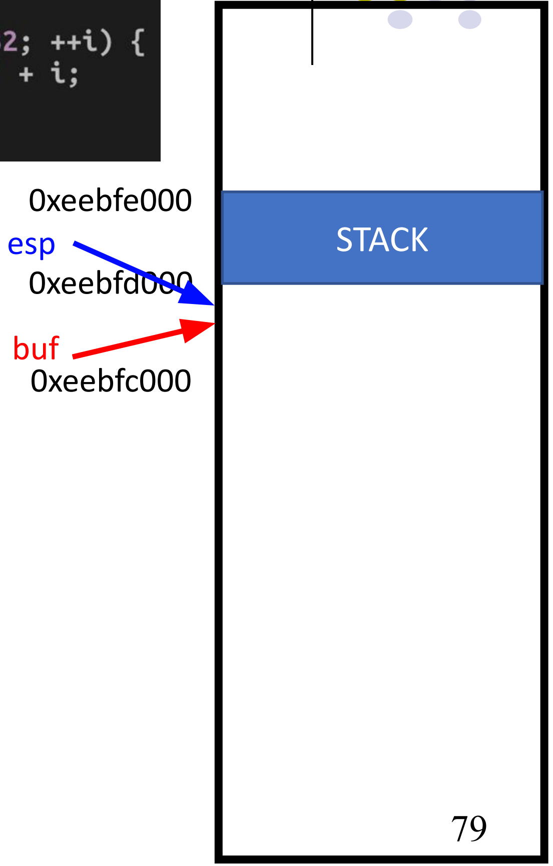


Expand stack automatically

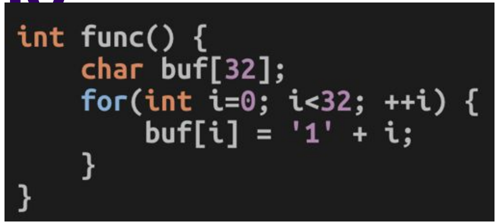


```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

- 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)`

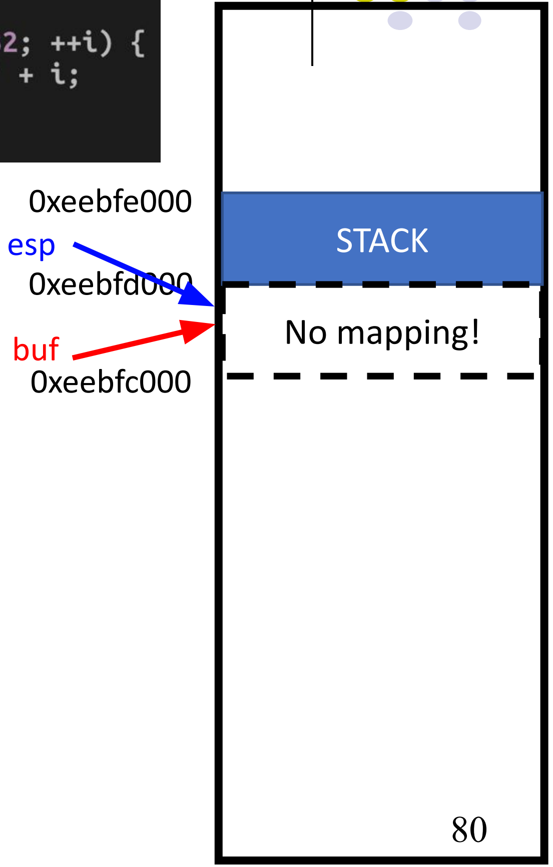


Expand stack automatically

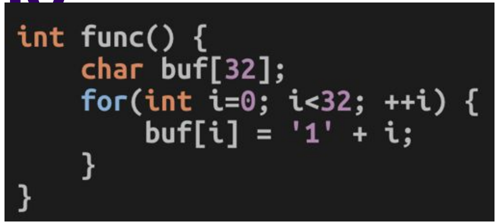


```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

- 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`

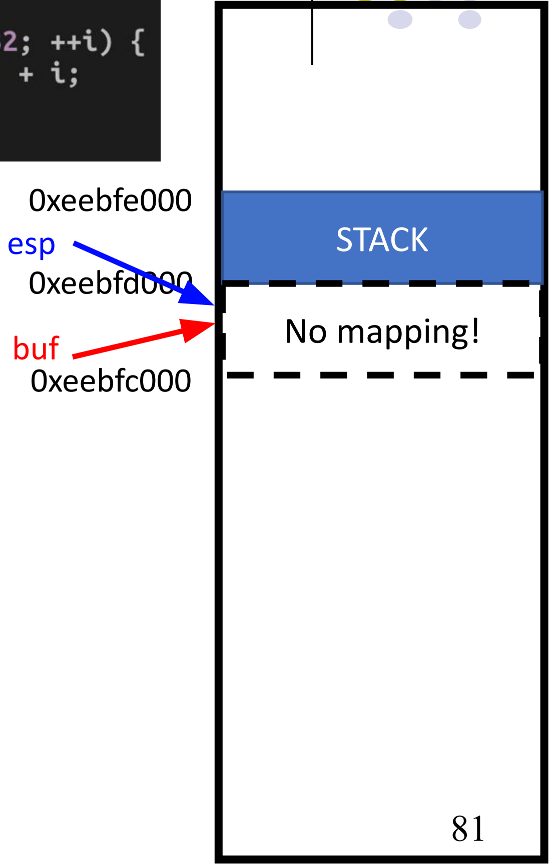


Expand stack automatically



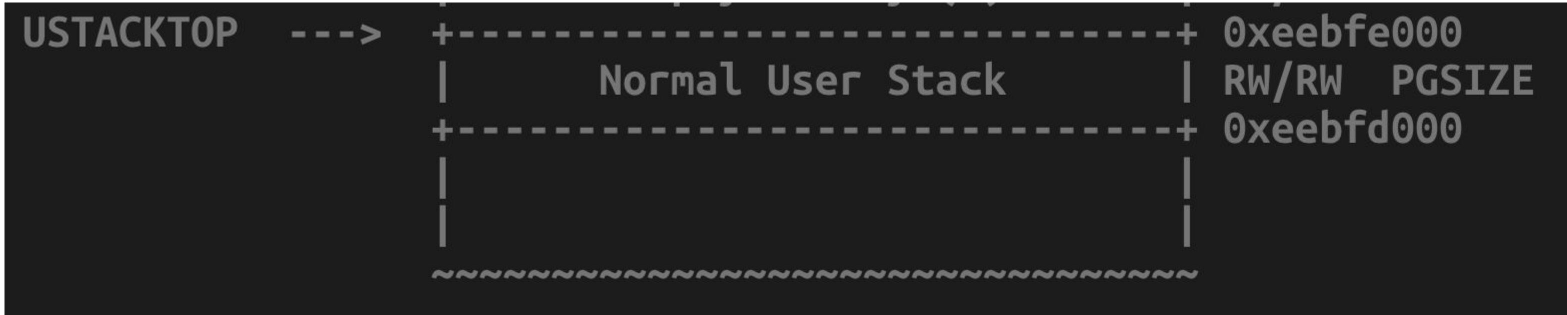
```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

- 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`**



What does processor do?

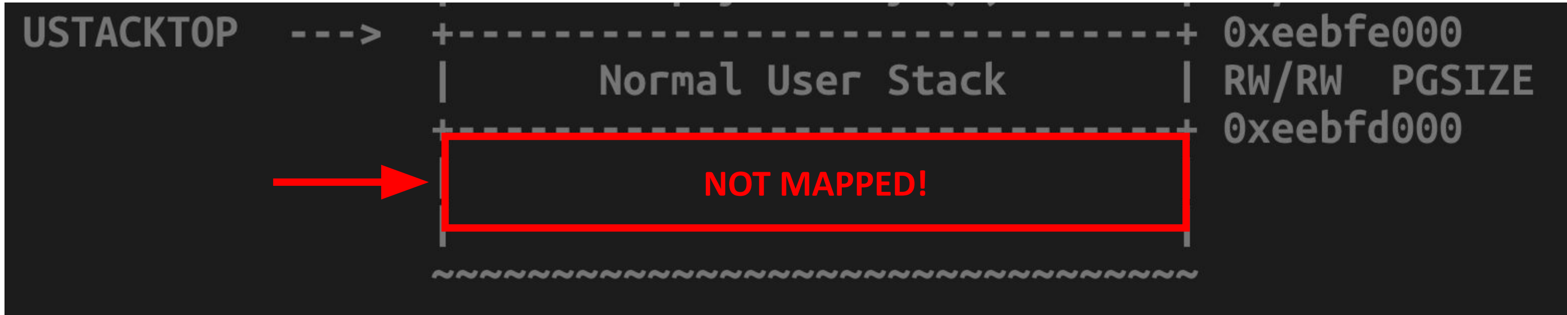
- Lookup page table
  - No translation!



```
USTACKTOP   --->  +--------------------------------+ 0xeebfe000
                  |       Normal User Stack        | RW/RW   PGSIZE
                  +--------------------------------+ 0xeebfd000
                  |                                |
                  |                                |
                  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

What does processor do?

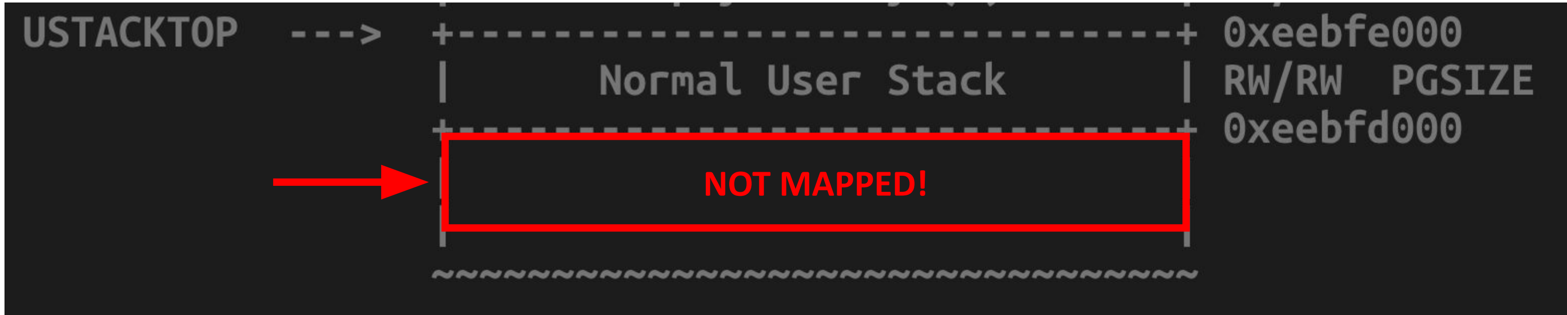
- Lookup page table
  - No translation!



```
USTACKTOP  --->  +--------------------------------+ 0xeebfe000
                 |       Normal User Stack        | RW/RW  PGSIZE
                 +--------------------------------+ 0xeebfd000
                 |                                |
                 |                                |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

What does processor do?

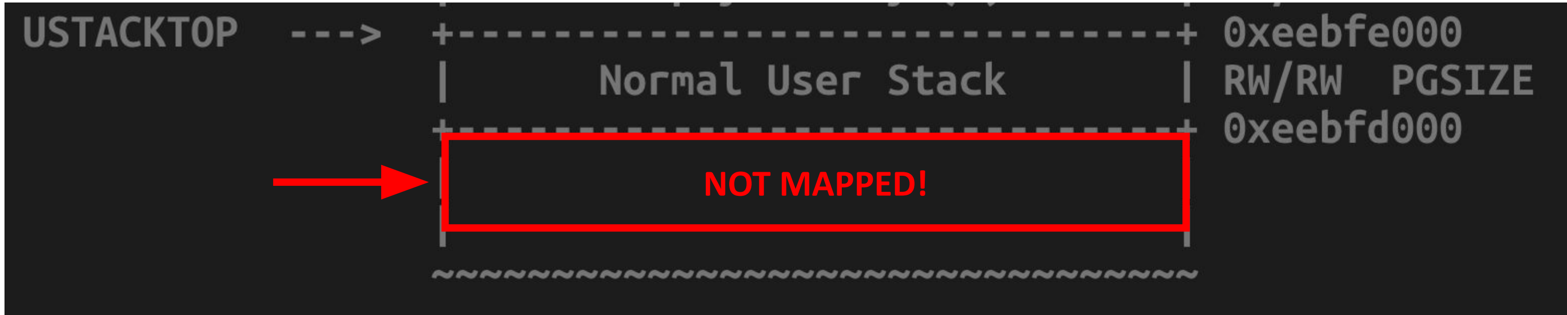
- Lookup page table
  - No translation!
- Store `0xeebfcff0` to CR2



```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW  PGSIZE
                 +------------------------------+ 0xeebfd000
                 |                              |
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

What does processor do?

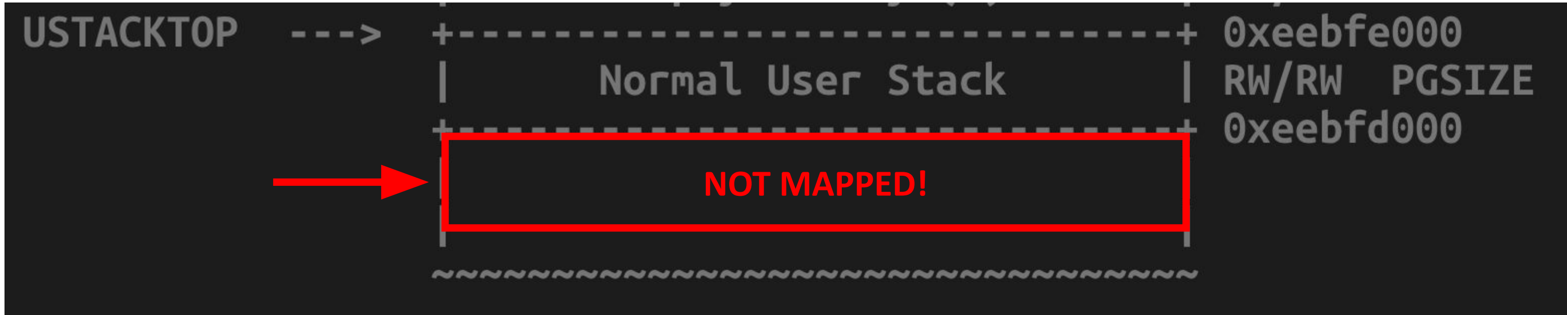
- Lookup page table
  - No translation!
- Store 0xeebfcff0 to CR2
- Set error code
  - “The fault was caused by a non-present page!”



```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW  PGSIZE
                 +------------------------------+ 0xeebfd000
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

What does processor do?

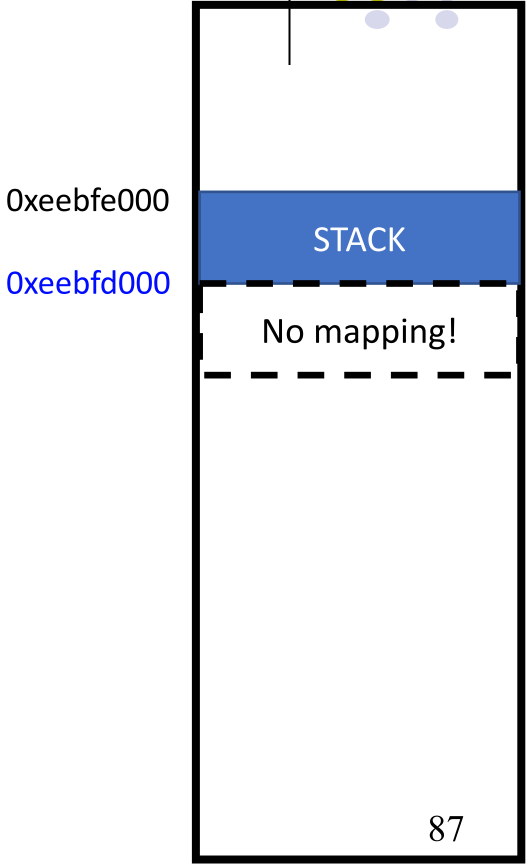
- Lookup page table
  - No translation!
- Store 0xeebfcff0 to CR2
- Set error code
  - "The fault was caused by a non-present page!"
- Raise page fault exception (interrupt #14) -> call page fault handler



```
USTACKTOP  --->  +------------------------------+ 0xeebfe000
                 |      Normal User Stack       | RW/RW  PGSIZE
                 +------------------------------+ 0xeebfd000
                 |                              |
                 |                              |
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

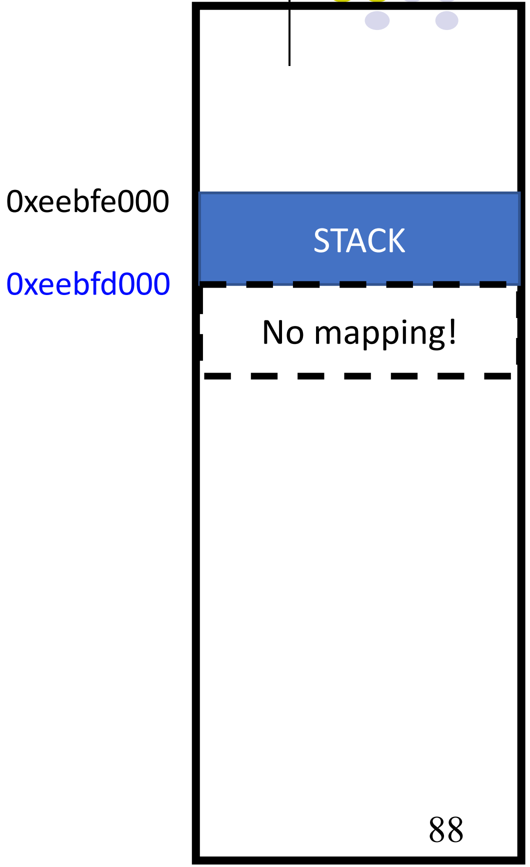
Handling page fault on Stack access

- Interrupt will make CPU invoke the page_fault_handler()



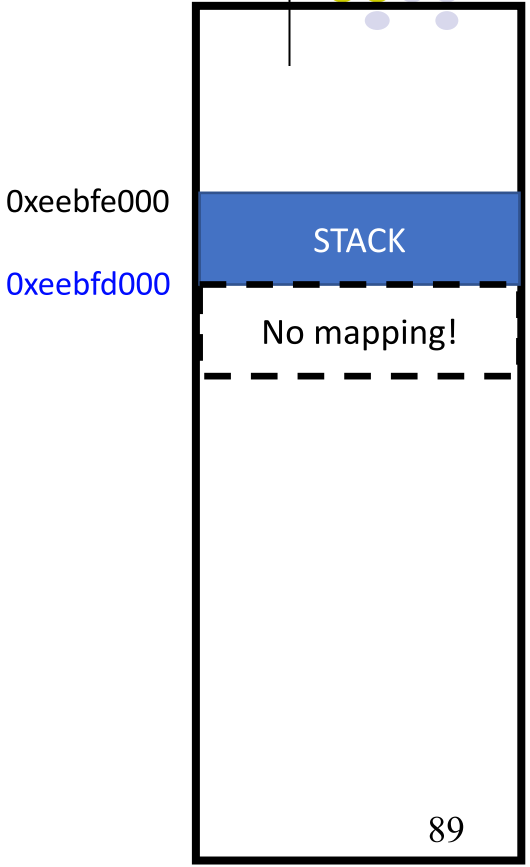
Handling page fault on Stack access

- Interrupt will make CPU invoke the page_fault_handler()
- Read CR2
    - `0xeebfcff0`



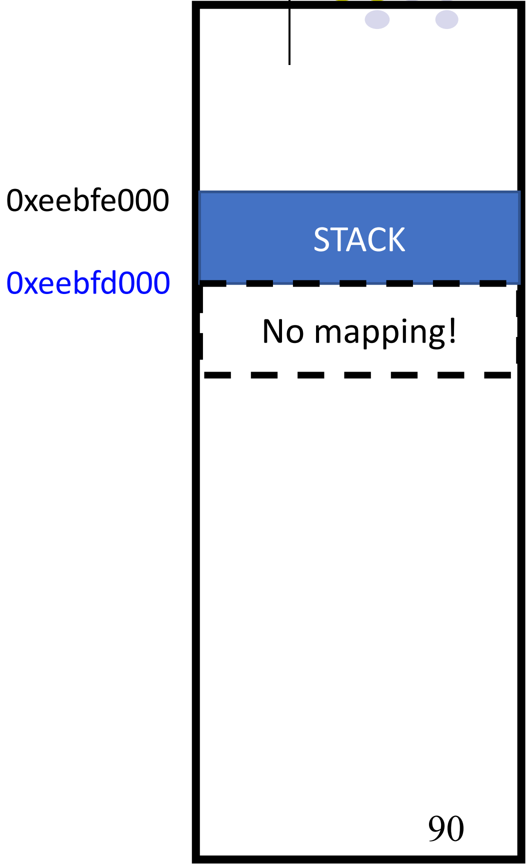
Handling page fault on Stack access

- 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



Handling page fault on Stack access

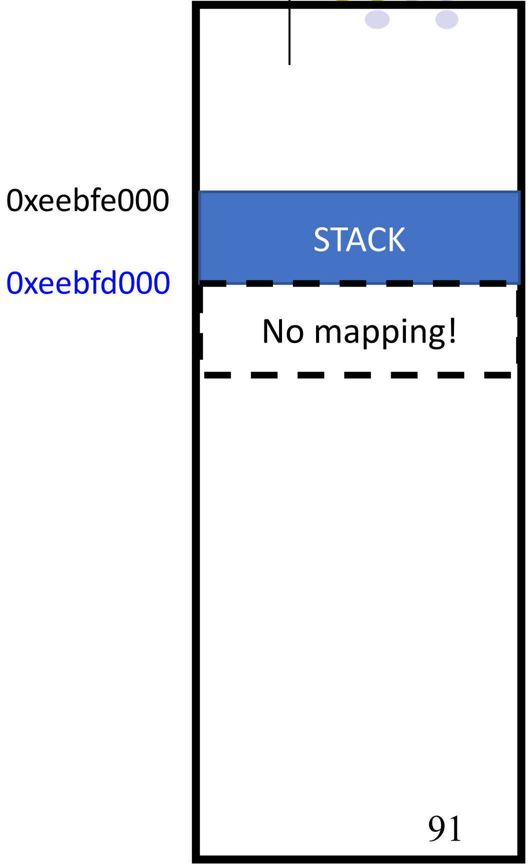
- Interrupt will make CPU invoke the page_fault_handler()
- Read CR2
  - `0xeebfcff0`, it seems like the page right next to current stack end
  - The current stack end is: 0xeebfd000
- Read error code
  - "The fault was caused by a non-present page!"



Handling page fault on Stack access

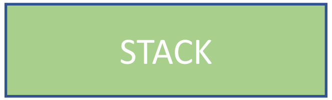
- Interrupt will make CPU invoke the page_fault_handler()
- Read CR2
  - `0xeebfcff0`, it seems like the page right next to current stack end
  - The current stack end is: 0xeebfd000
- Read error code
  - "The fault was caused by a non-present page!"

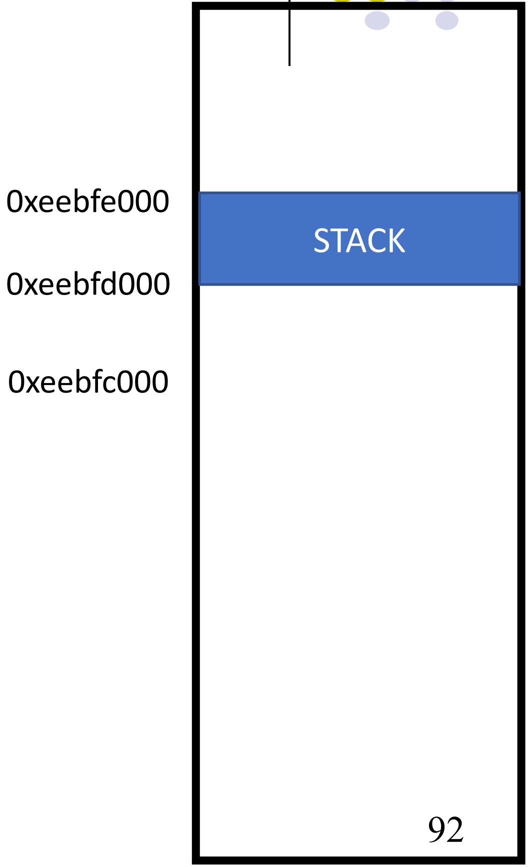
- Let's allocate a new page for the stack!



Adding new page for stack

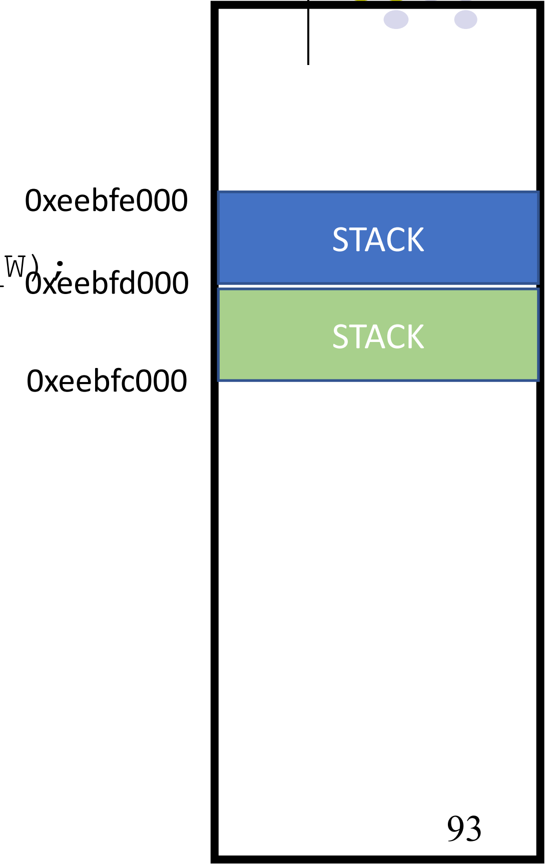
- Allocate a new page for the stack
  - `Struct PageInfo *pp = page_alloc(ALLOC_ZERO);`
    - Get a new page, and wipe it to have all zero as its contents





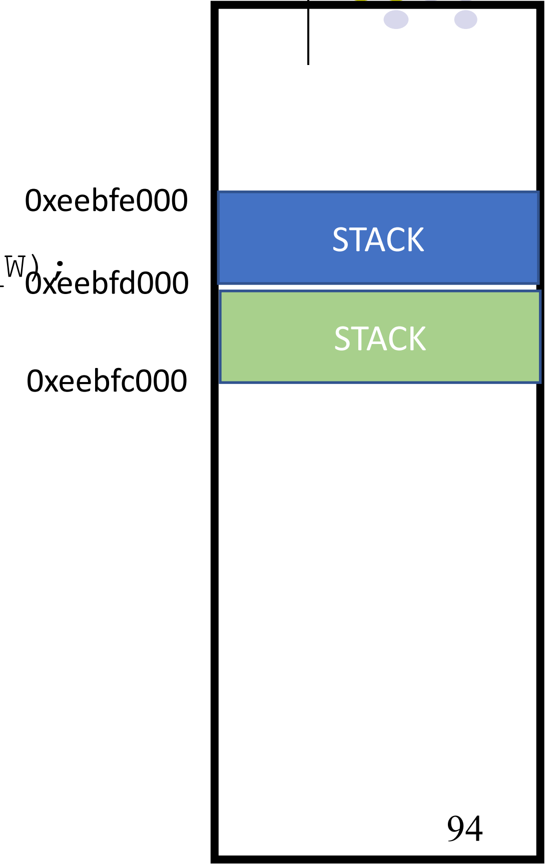
Adding new page for stack

- 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!



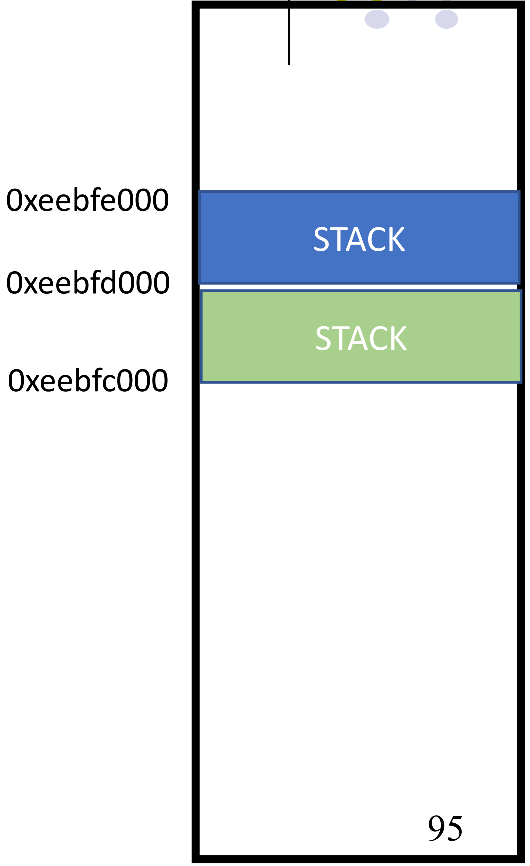
Adding new page for stack

- 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!



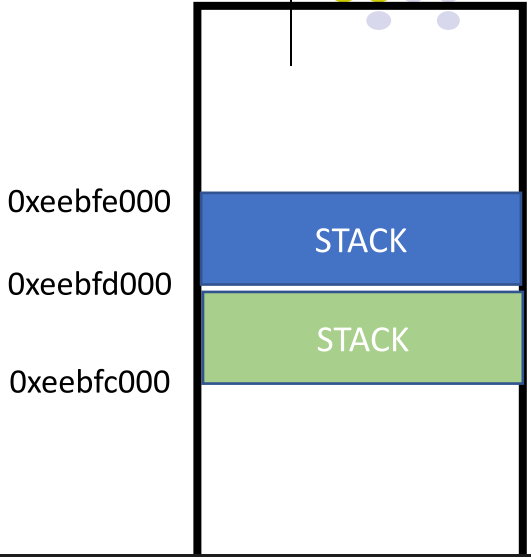
Resuming execution of user process

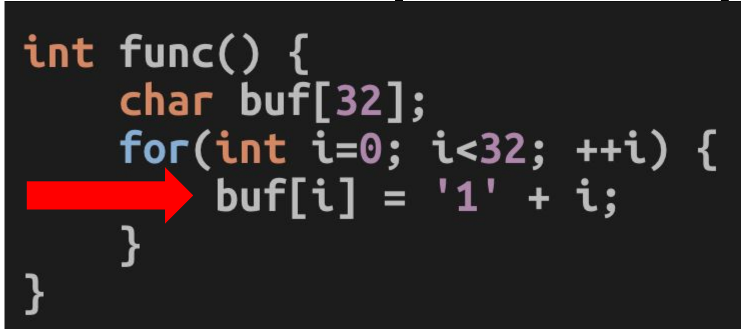
- On accessing `buf[0] = '1';`
  - `movb $0x31, (%eax)`
  - `eax = 0xeebfcff0 No translation for 0xeebfc000`



Resuming execution of user process

- 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`

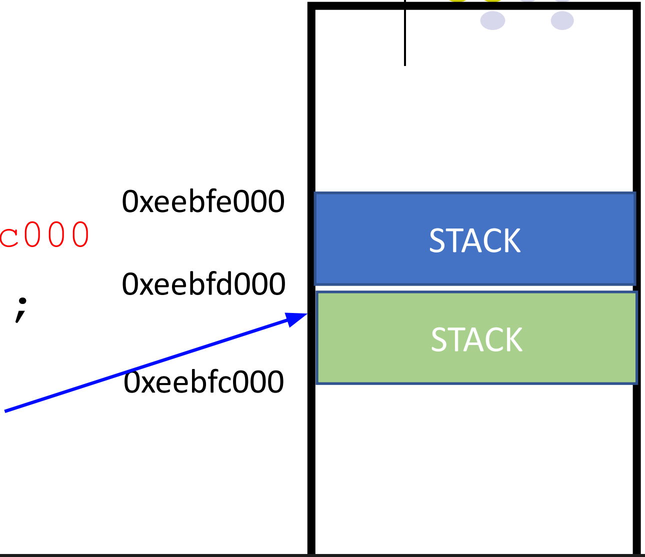


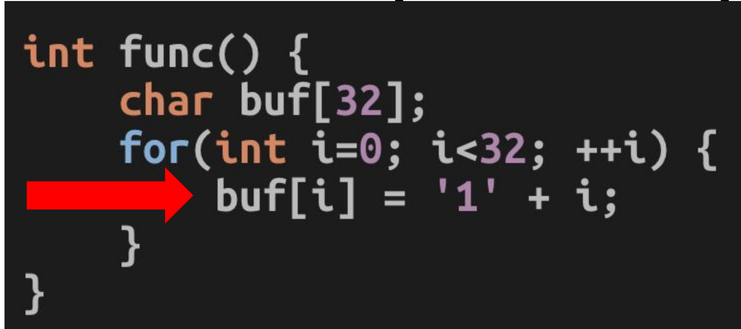


```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

Resuming execution of user process

- 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!`

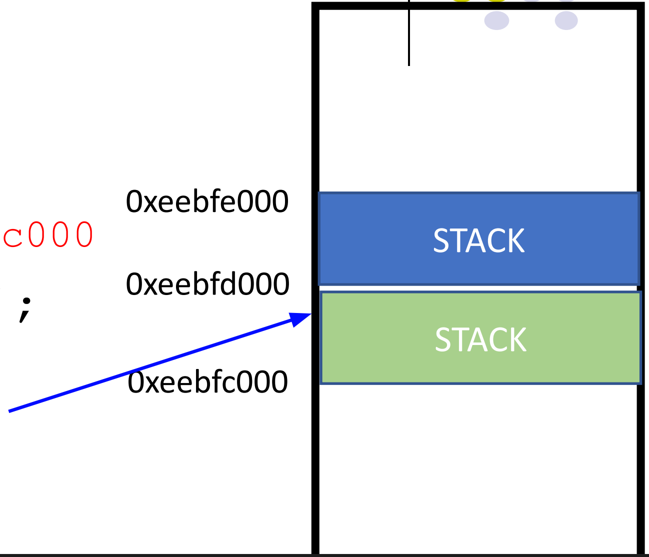


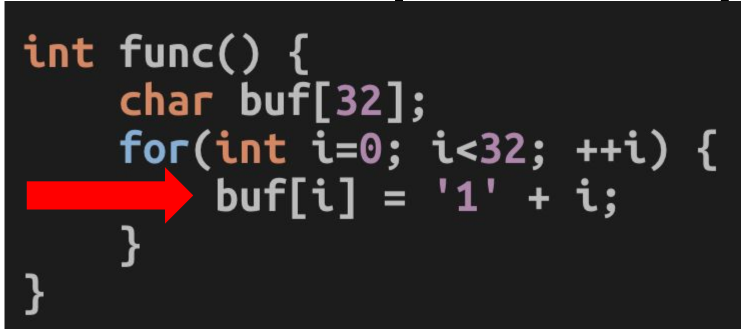


```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

Resuming execution of user process

- 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..

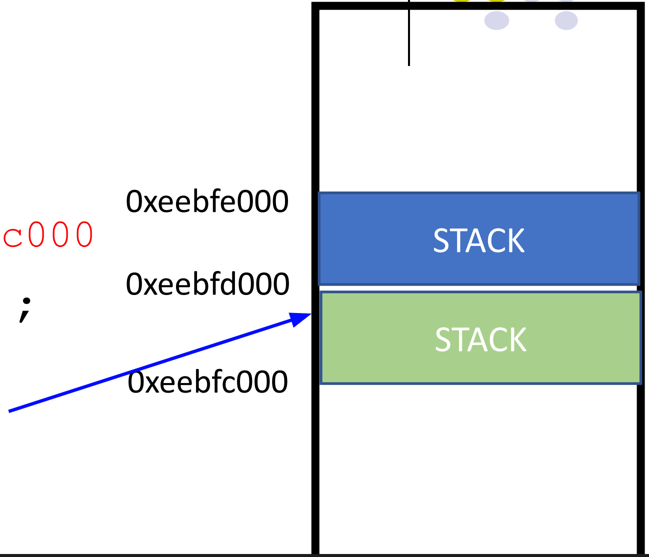




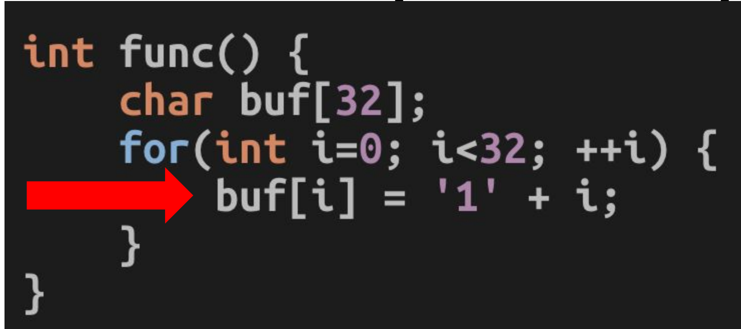
```
int func() {
    char buf[32];
    for(int i=0; i<32; ++i) {
        buf[i] = '1' + i;
    }
}
```

Resuming execution of user process

- 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..



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;
    }
}
```