



CSE 451: Operating Systems, 26au

L3: The Kernel and its Abstractions

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

Sagar Saby

Andrew Wood

Adapted from slides by Tom Anderson

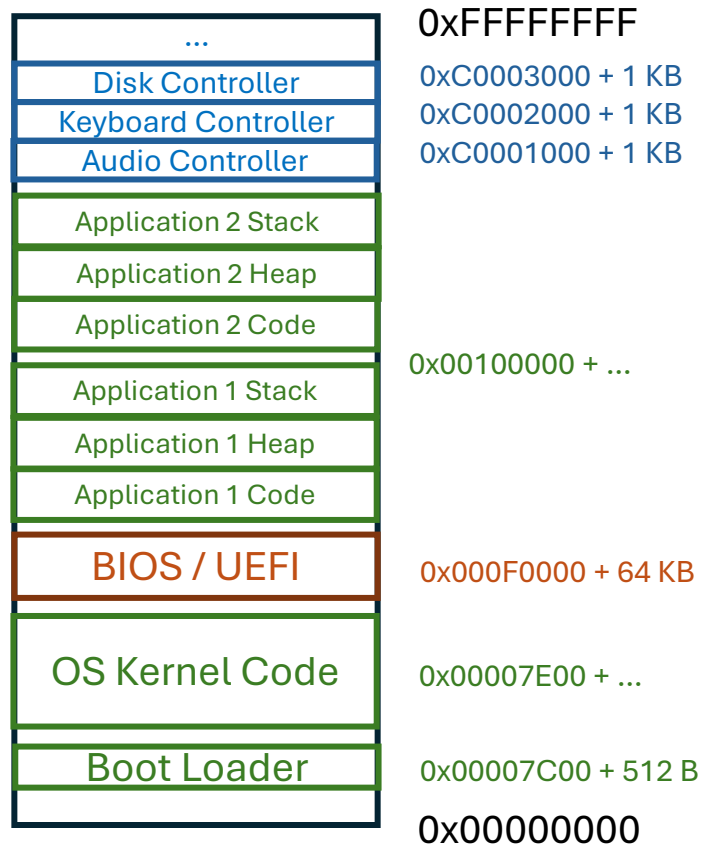
Administrivia

- Lab 1 out
 - Lab groups and repo links emailed to you later today
 - Due 10/12 (next Monday)
- Pset 1 out on Gradescope
 - Due 10/16 (a little under 2 weeks)

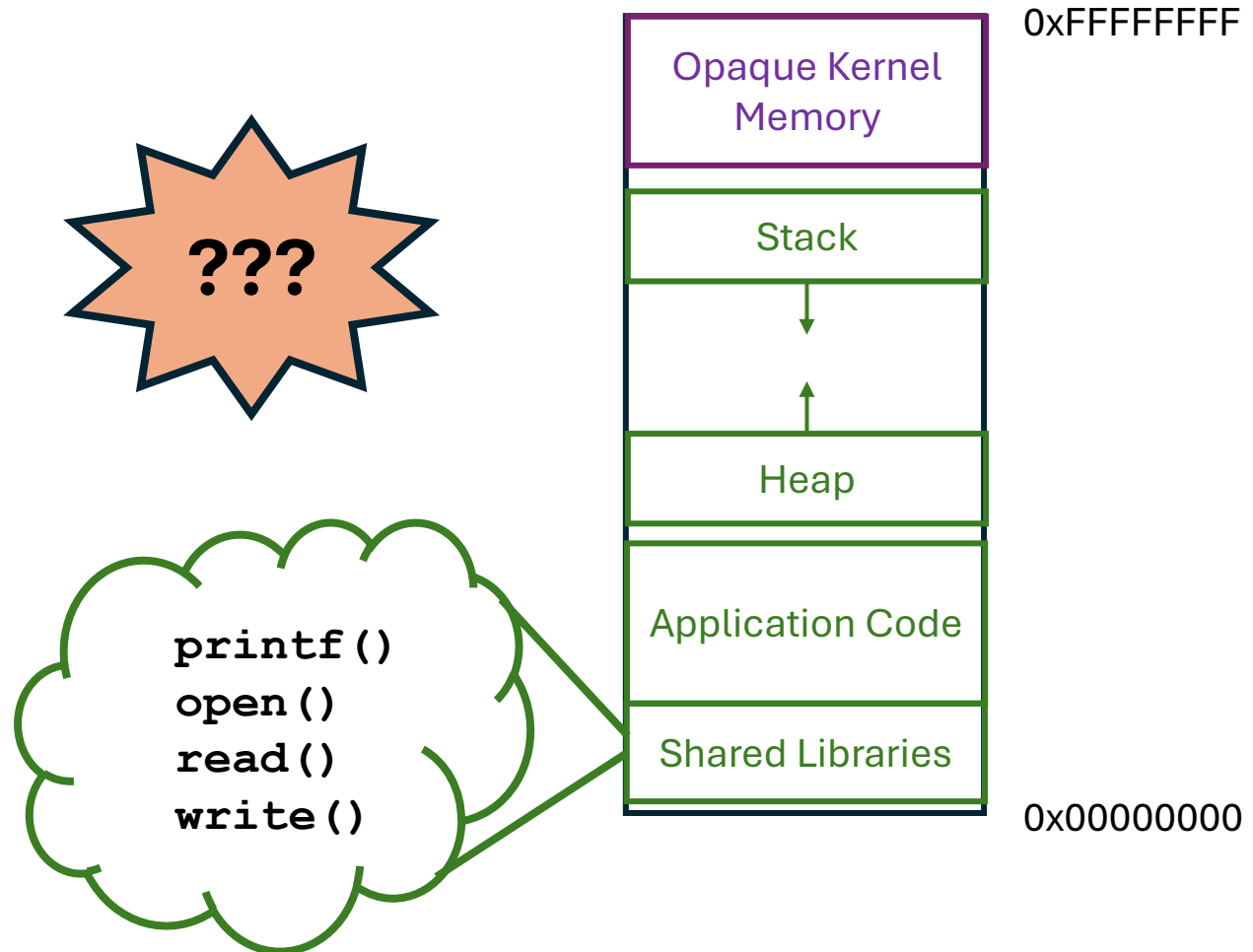
Kernels and Processes

Protecting the computer from itself

Physical Address Space



The Illusion of a Computer



Design goal: protection

- Code tends to be **buggy**
- Code can be **malicious**
- What are some things we want to **keep user code from doing?**

Process Abstraction

- Process: a *single running instance* of a program on a computer
- Consists of:
 - Memory arranged into an address space (stacks, a heap, program code, the classics)
 - Threads (paths of execution through the program code)
 - Private metadata structures in the kernel to keep track of all that stuff
- Has limited rights!
 - Cannot modify or read operating system kernel code or data
 - Cannot modify or read I/O controller memory
 - Cannot modify its own rights without asking the kernel
 - Can be “kicked off” of the CPU against its will
 - Etc.

The process abstraction



Referee



Illusionist



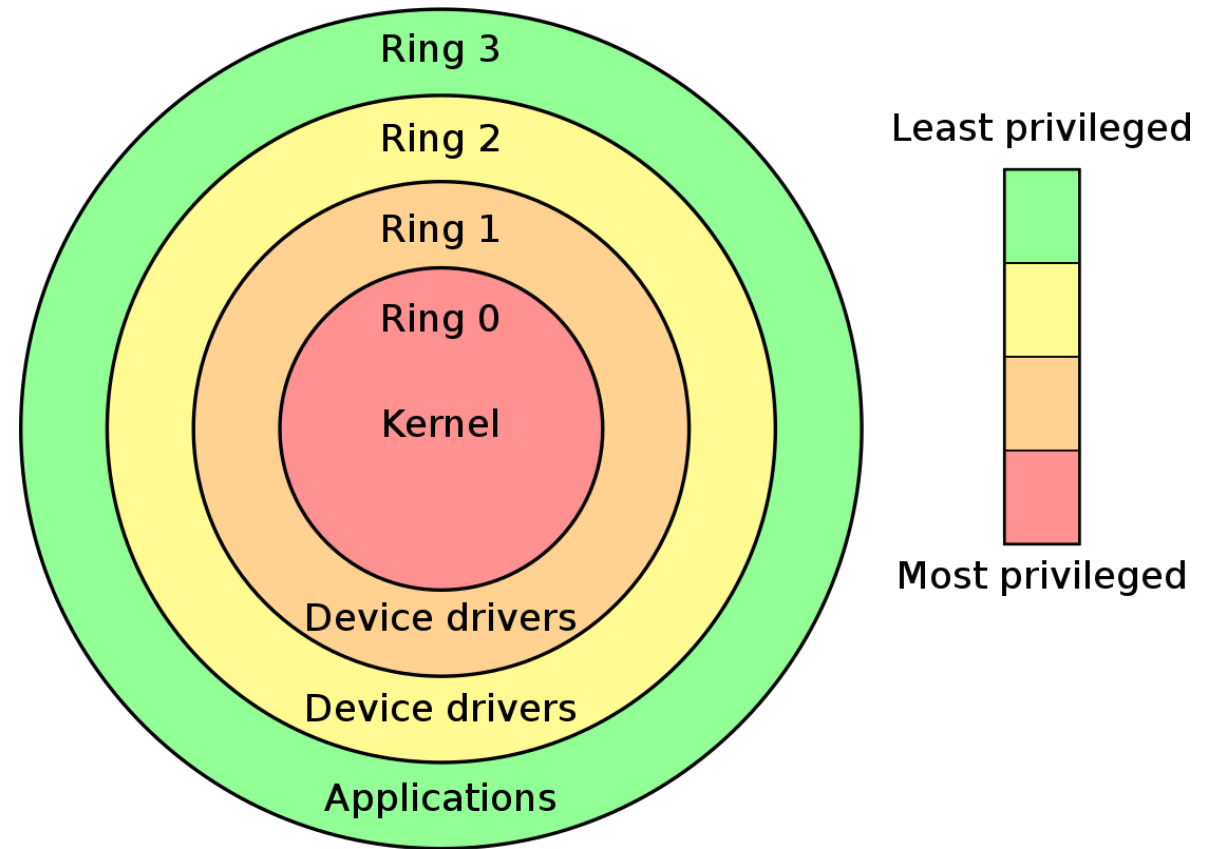
Glue

CPU “Dual-mode Operation”

- Kernel mode
 - Privilege to do absolutely whatever
- User mode
 - Only privileged to execute a subset of instructions
 - Only privileged to access a subset of physical memory
 - Violating these privileges causes the processor to immediately triggers an interrupt
- On the x86, mode stored in the CPU's EFLAGS register

Privilege levels in x86

- x86 privileges are actually configurable in up eight levels, called “**rings**”
- Most commonly:
 - ring 0 (kernel mode)
 - rings 1 and 2 (elevated system processes)
 - ring 3 (user mode)
- Extra rings help support **virtual machines** (more later)



Hardware Timers

- Hardware components that sit next to the CPU core and periodically interrupt it
 - Whatever the process is doing, control jumps to the kernel's timer interrupt handler
 - Handler checks if current user process has been running too long
 - Puts it to sleep if so
 - Returns control to program if not
- Interrupt frequency set by the kernel
 - Not by user code!
- Interrupts can be temporarily deferred by kernel
 - Not by user code!
 - Crucial for implementing mutual exclusion

Kernel Mode Transfer

Hopping between user space and kernel space

Kernel Mode Transfer

- Happens in many scenarios, but all with the same underlying hardware mechanism:
 - **Hardware Interrupts**: triggered by timer and I/O devices
 - **“Traps”** / Software Interrupts:
 - **Exceptions**: triggered by unexpected or malicious application behavior (eg, divide-by-zero)
 - **System calls (“syscalls”)**: “protected procedure call,” user-space software requests the kernel to do something

Kernel Mode Transfer (2)

- Motivation: the number of places code can actually transition from user-to-kernel is **very small** and written **very carefully**
- Mode change is **atomic**, with hardware changing all of this at once:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
- In general, interrupts cannot be interrupted
 - They do the minimum amount of code possible to get CPU ready to be interrupted again
 - Usually: add work descriptor to a queue and signal another kernel thread to do the real work

Some Traps

Table 6-1. Exceptions and Interrupts

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. ¹
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.

More Traps

Table 6-1. Exceptions and Interrupts (Contd.)

Vector	Mnemonic	Description	Source
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. ²
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ³
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁴
20	#VE	Virtualization Exception	EPT violations ⁵
21	#CP	Control Protection Exception	The RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at the target of an indirect call or jump.
22-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

[Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1](#) (p 169)

UNIX systems calls (1972)

- Mount, unmount filesystem
- File open/create
- Seek to a file offset
- Read/write file
- Close, delete file
- Change file owner/mode
- Create/delete directory
- Add link to file
- ioctl for I/O device operations
- Process fork
- Process exec
- Exit process; wait for exit
- Create pipe between processes
- Send signal to another process
- Set/mask signal handler
- Extend heap memory region

(nifty trick: “Trap and Emulate”)

- OS technique for providing functionality beyond what hardware provides by itself
- Primary tool for  **OS-as-Illusionist** 
- Examples:
 - Handle a divide by zero error
 - Emulate floating point unit on hardware without one
 - Interacting with memory regions that haven't been used yet

Nitty-gritty: the x86 INT instruction

- Hardware saves current stack pointer
- Saves current program counter
- Saves current processor status word (condition codes)
- Switches to kernel stack
- Puts SP, PC, PSW on stack
- Switches to kernel mode
- Indexes into interrupt vector to find interrupt handler
- Jumps to interrupt handler, which saves registers it might clobber

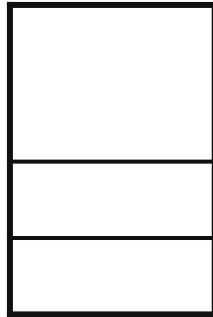


Before Interrupt

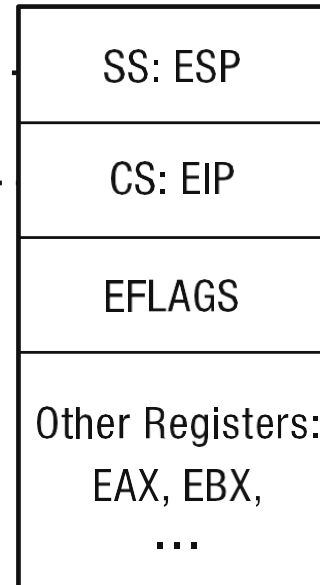
User-level Process

```
foo () {  
    while(...) {  
        x = x+1;  
        y = y-2;  
    }  
}
```

User Stack



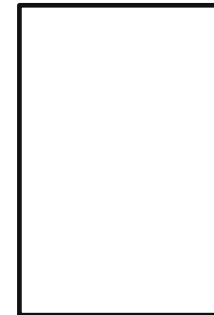
Registers



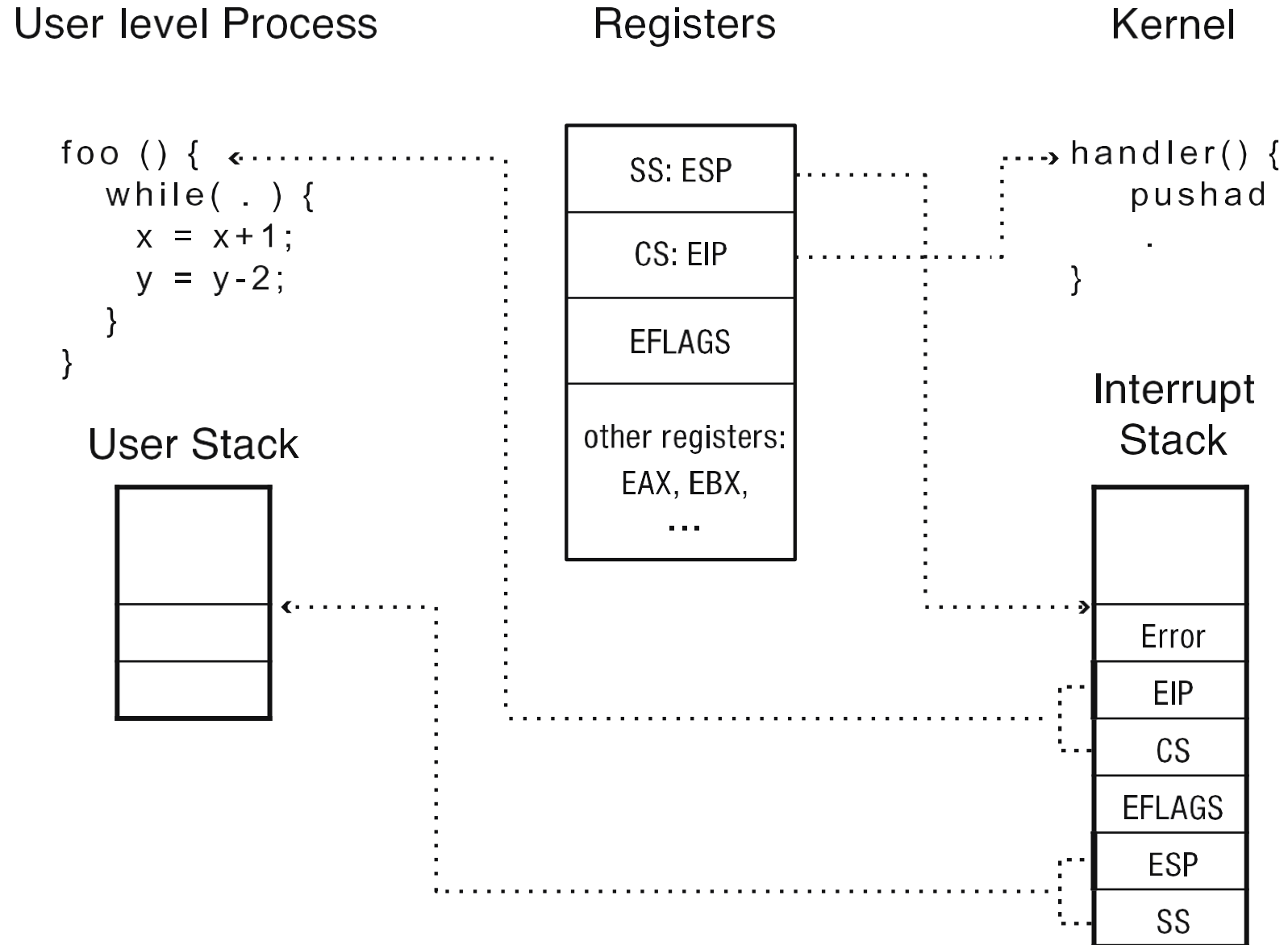
Kernel

```
handler() {  
    pushad  
    ...  
}
```

Interrupt Stack



Right After Interrupt Occurred

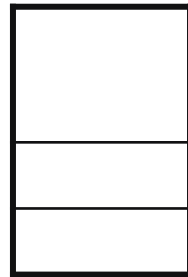


During Interrupt Handler

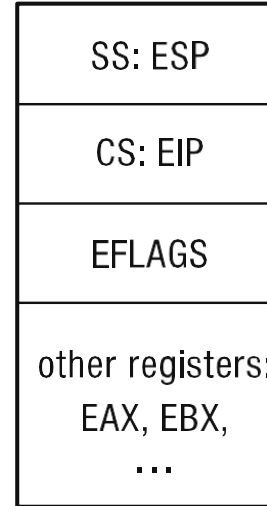
User level Process

```
foo () {  
    while( . ) {  
        x = x+1;  
        y = y-2;  
    }  
}
```

Stack



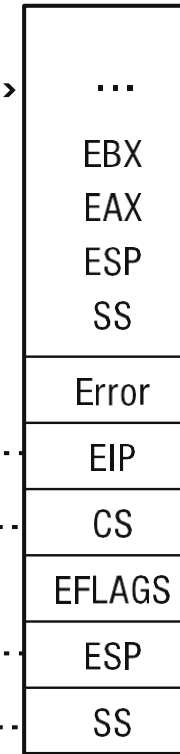
Registers



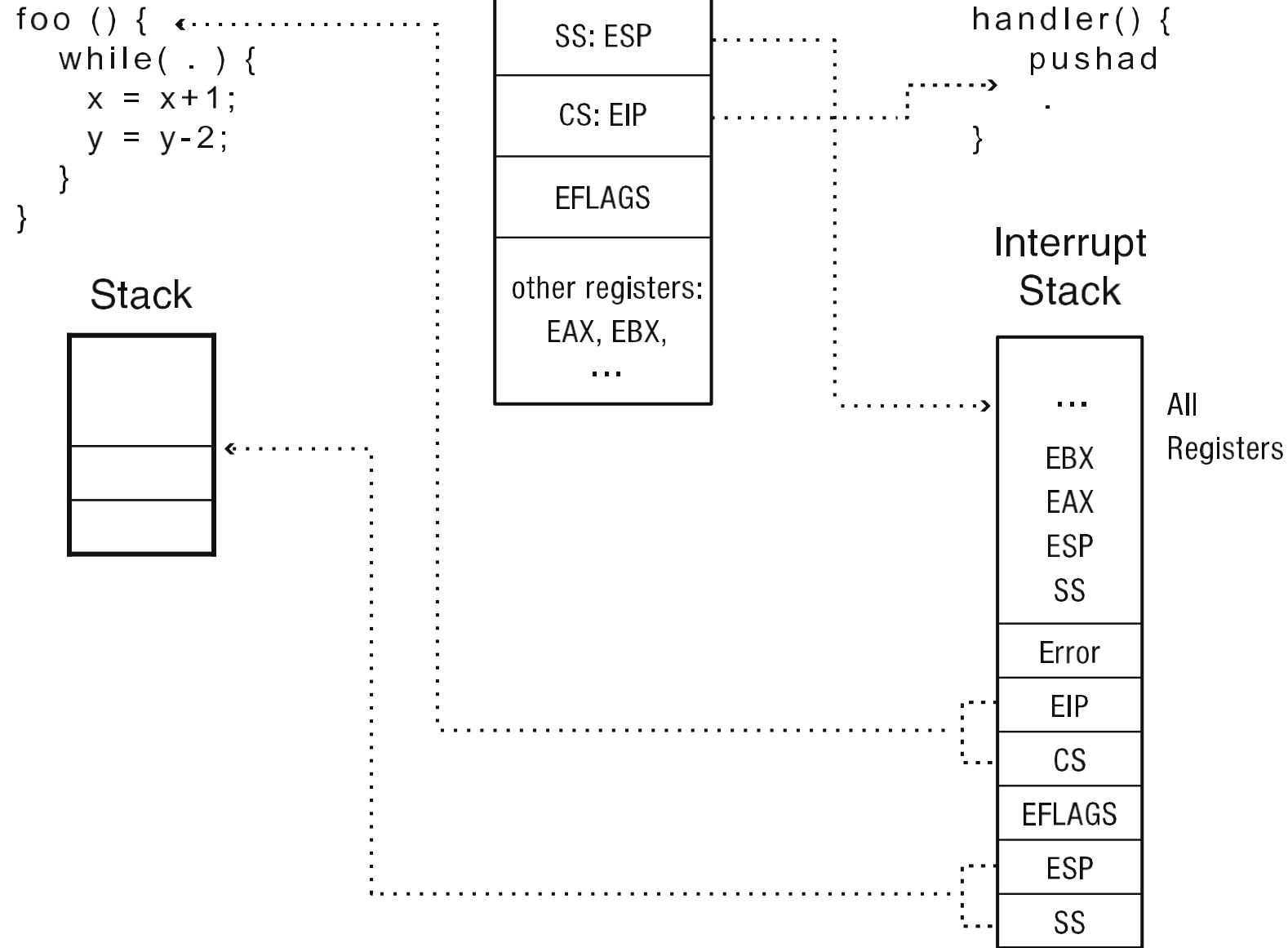
Kernel

```
handler() {  
    pushad  
    .  
}
```

Interrupt
Stack



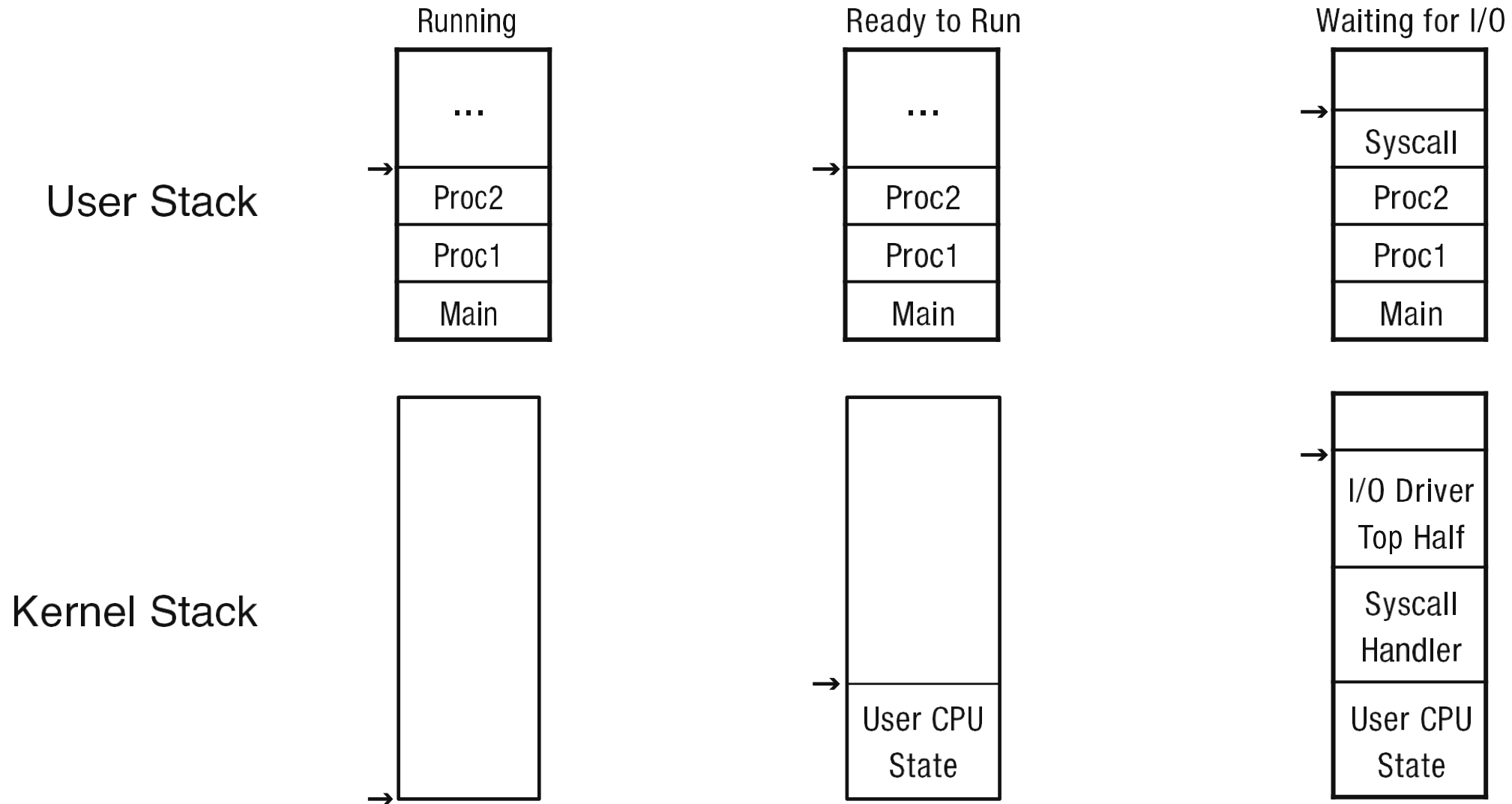
All
Registers



Interrupt Stack

- Per-processor, located in kernel (not user) memory
 - Usually a process/thread has both: kernel and user stack
 - Pointer to the kernel stack is stored in process metadata, along with other context like privileges and address space details
 - Pointer to *that* metadata is configured with the privileged “LTR” instruction
 - 🤔 Why don't we just let the interrupt handler run on the stack of the interrupted user process?
- In general, interrupts cannot be interrupted
 - Minimum amount of code possible to get CPU ready to be interrupted again
 - Usually: add work descriptor to a queue and signal another kernel thread to do the real work

Interrupt Stack Examples



At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
 - Restore program counter
 - Restore program stack
 - Restore processor status word/condition codes
 - Switch to user mode

Kernel->User Mode Switch

- From kernel mode to user mode
 - New process/new thread start
 - Jump to first instruction in program/thread
 - Return from interrupt, exception, system call
 - Resume suspended execution
 - Process/thread context switch
 - Resume some other process
 - User-level upcall (UNIX signal)
 - Asynchronous notification to user program

Bonus: How *exactly* do INT n and IRET work?

- <https://www.felixcloutier.com/x86/intn:into:int3:int1#operation>
- <https://www.felixcloutier.com/x86/iret:iretd:iretq#operation>

Class concluded here on 10/5
and will resume on 10/7



Syscall Implementation

(eg, lab 1)

System Calls

- Functions provided by the kernel to untrusted application code
 - Ex: file open, file read, file write, file close, pipe
- Limited set of entry points into the kernel
 - User code can't just call any kernel procedure
 - Protected procedure calls
- API is system-specific (UNIX != Windows)

User Program

```
main() {  
    open("cats/danni.jpg", "r");  
}
```

User Stub

```
open:  
push #SYSCALL_OPEN  
trap  
return
```

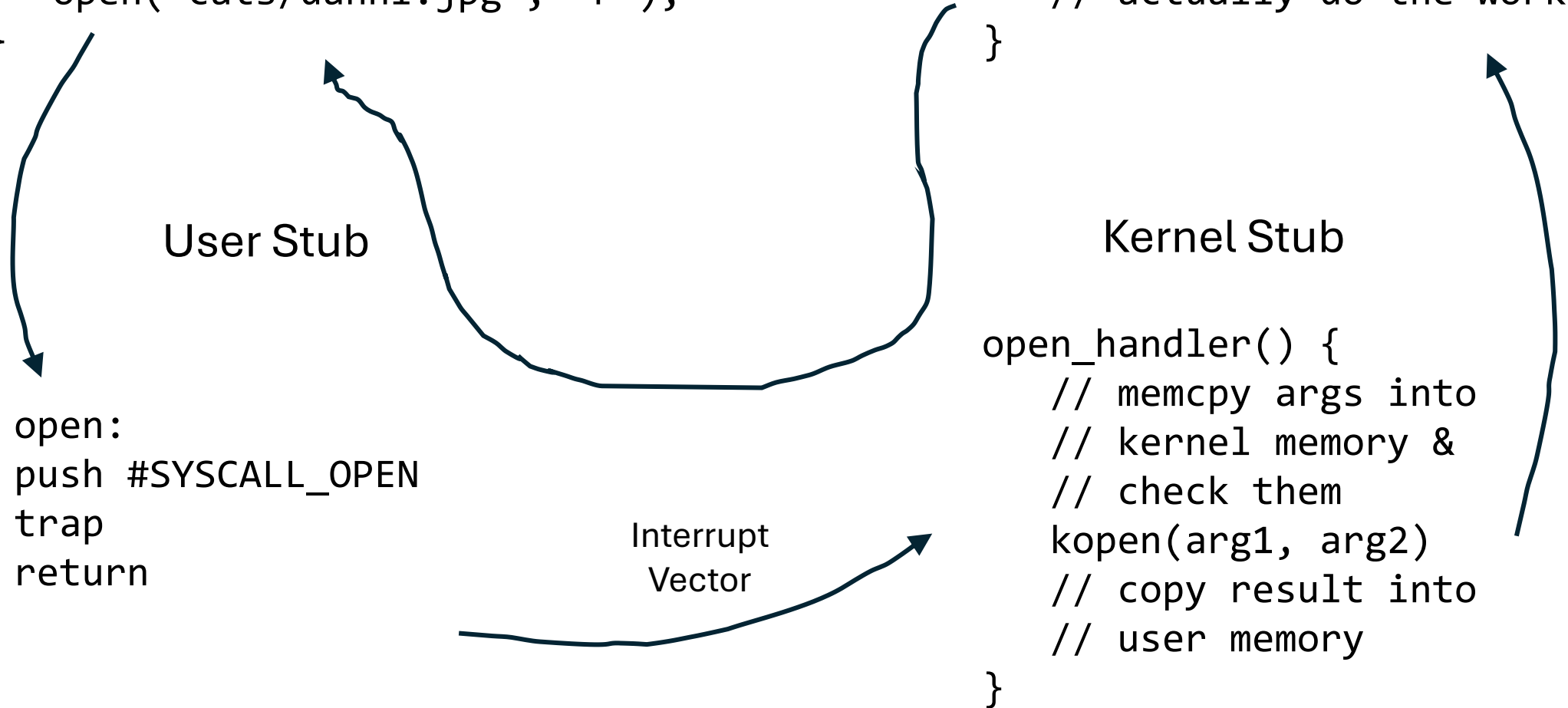
Kernel

```
kopen() {  
    // actually do the work  
}
```

Kernel Stub

```
open_handler() {  
    // memcpy args into  
    // kernel memory &  
    // check them  
    kopen(arg1, arg2)  
    // copy result into  
    // user memory  
}
```

Interrupt
Vector

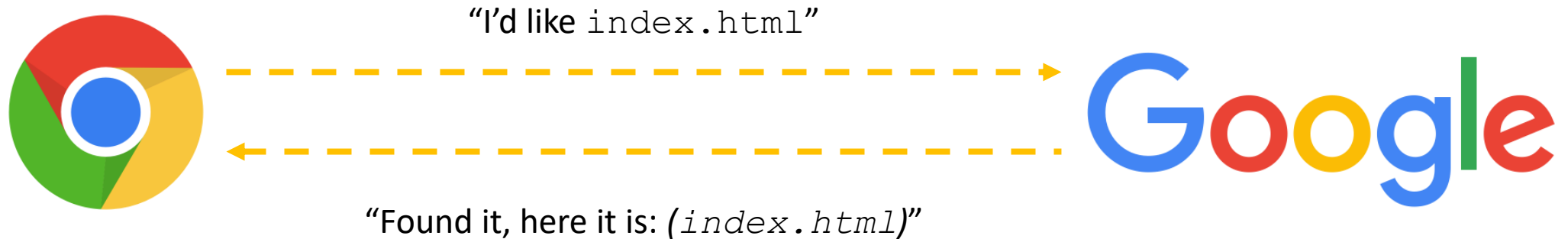


Kernel System Call Handlers

- Locate arguments
 - In registers or on user stack
 - *Translate* user addresses into kernel addresses
- Copy arguments
 - From user memory into kernel memory
 - Protect kernel from malicious code evading checks
- Validate arguments
- Protect kernel from errors in user code
- Copy results back into user memory
 - *Translate* kernel addresses into user addresses

Exercise

- How many user-kernel transitions are needed for a static web server to read an incoming HTTP request and reply with the file data?



Server

Kernel

Hardware

