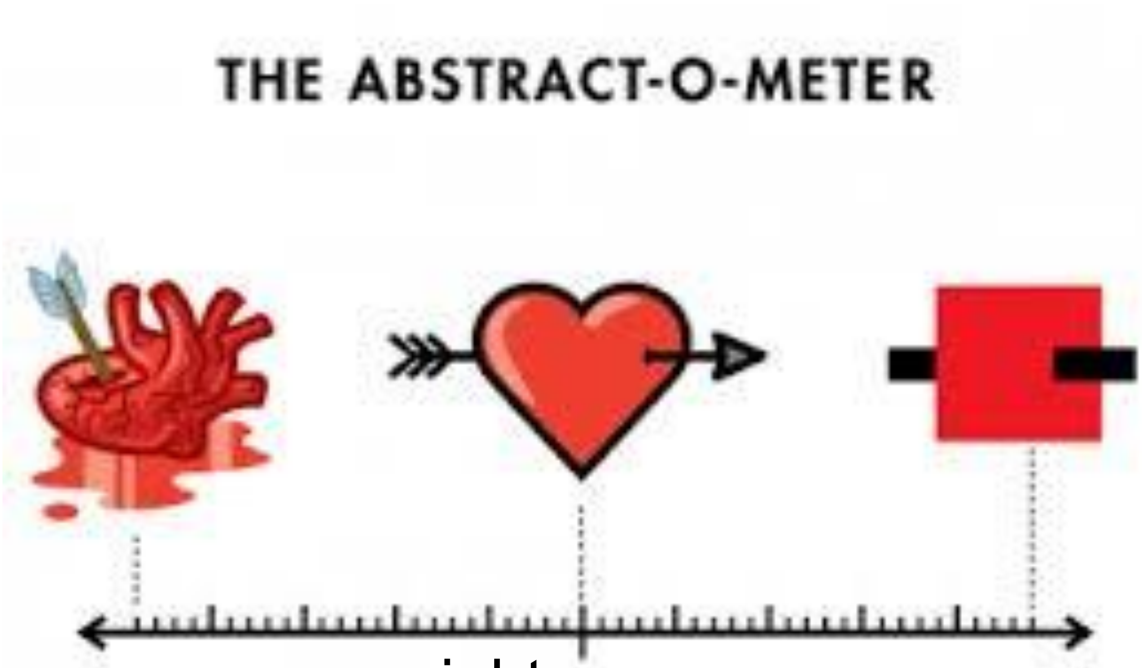
CS330: The Kernel Abstraction

Instructor: Youngjin Kwon

Illusionist role of OS

What is "Abstraction"?

- The **process or outcome** of making something **easier to understand** by **ignoring some of details** that may be unimportant



Too realistic

right abstraction

Too abstract

Question: choose sentence(s) saying abstraction

- Distributing memory among multiple processes X → It's about mechanism

- Supporting different types of monitors O

- Interchangeable accesses to harddisk or SSD O

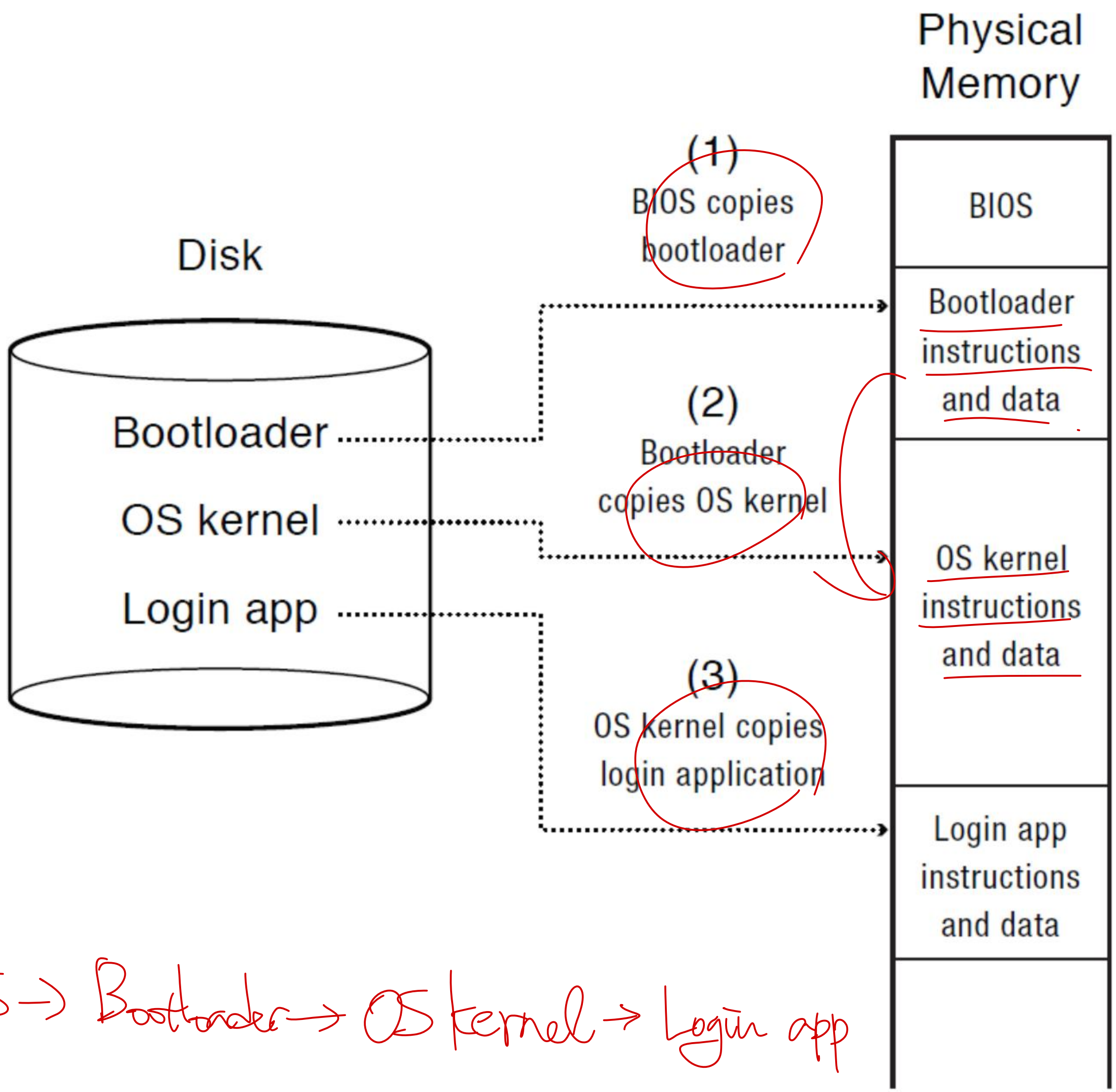
- ….

Why does an operating system is separated from application?

illusion

Booting

MS-DOS
Windows 3.1
→ no seperattion of kernel & app,
∴ app can crash kernel



BIOS → Bootloader → OS kernel → Login app

After booting ….

- You visited eBay to purchase … Then…

Hackers still exploiting eBay's stored XSS vulnerabilities in 2017

Fraudsters are still exploiting eBay's persistent cross-site scripting vulnerabilities to steal account credentials, years after a series of similar attacks took place. Worse still, many of the listings that exploited these vulnerabilities remained on eBay's website for more than a month before they were eventually removed.

7 Votes

ShellShock: All you need to know about the Bash Bug vulnerability

Web servers at risk as new vulnerability potentially affects most versions of Linux and Unix, as well as Mac OS X.

A new vulnerability has been found that potentially affects most versions of the Linux and Unix operating systems, in addition to Mac OS X (which is based around Unix). Known as the "Bash Bug" or "ShellShock," the GNU Bash Remote Code Execution Vulnerability (CVE-2014-6271) could allow an attacker to gain control over a targeted computer if exploited successfully.

Challenge

- Some examples:
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven't tested yet

가두다

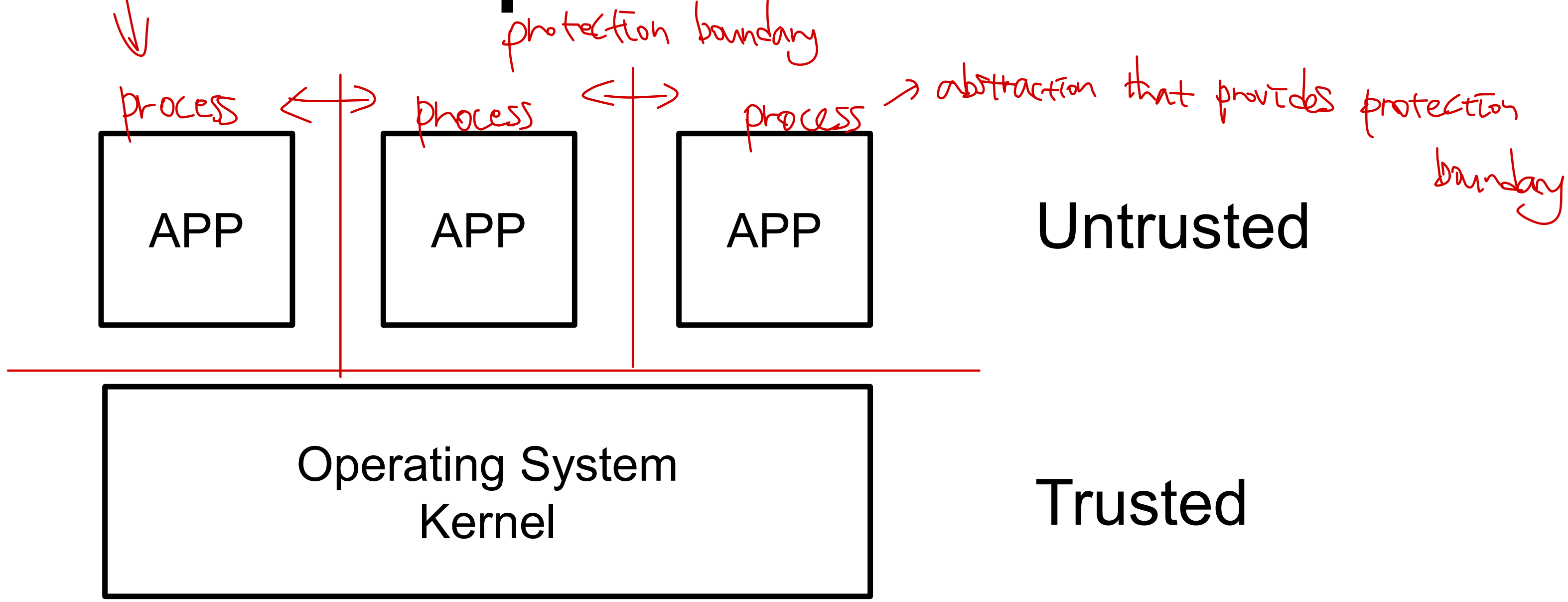
- How to confine the buggy or malicious applications?
  - How do we execute code with restricted privileges?

Why does an operating system is separated from application?

Need an abstraction....

Apps can't crash kernel

For protection



Main Points

- Process concept → Act as whole machine (Illusion)
  - Process is an OS abstraction for executing a program with limited privileges
- Hardware-supported protection
  - Dual-mode operation: user vs. kernel
    - Kernel-mode: execute with complete privileges
    - User-mode: execute with fewer privileges
- Safe control transfer: Mode switch
  - How do we switch from one mode to the other?

Process abstraction

- Process: an *instance* of a program, running with limited rights
  - Thread: a sequence of instructions runs on ***CPU*** within a process
    - Potentially many threads per process (for now 1:1)
  - Address space: set of rights of a process
    - ***Memory*** that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)

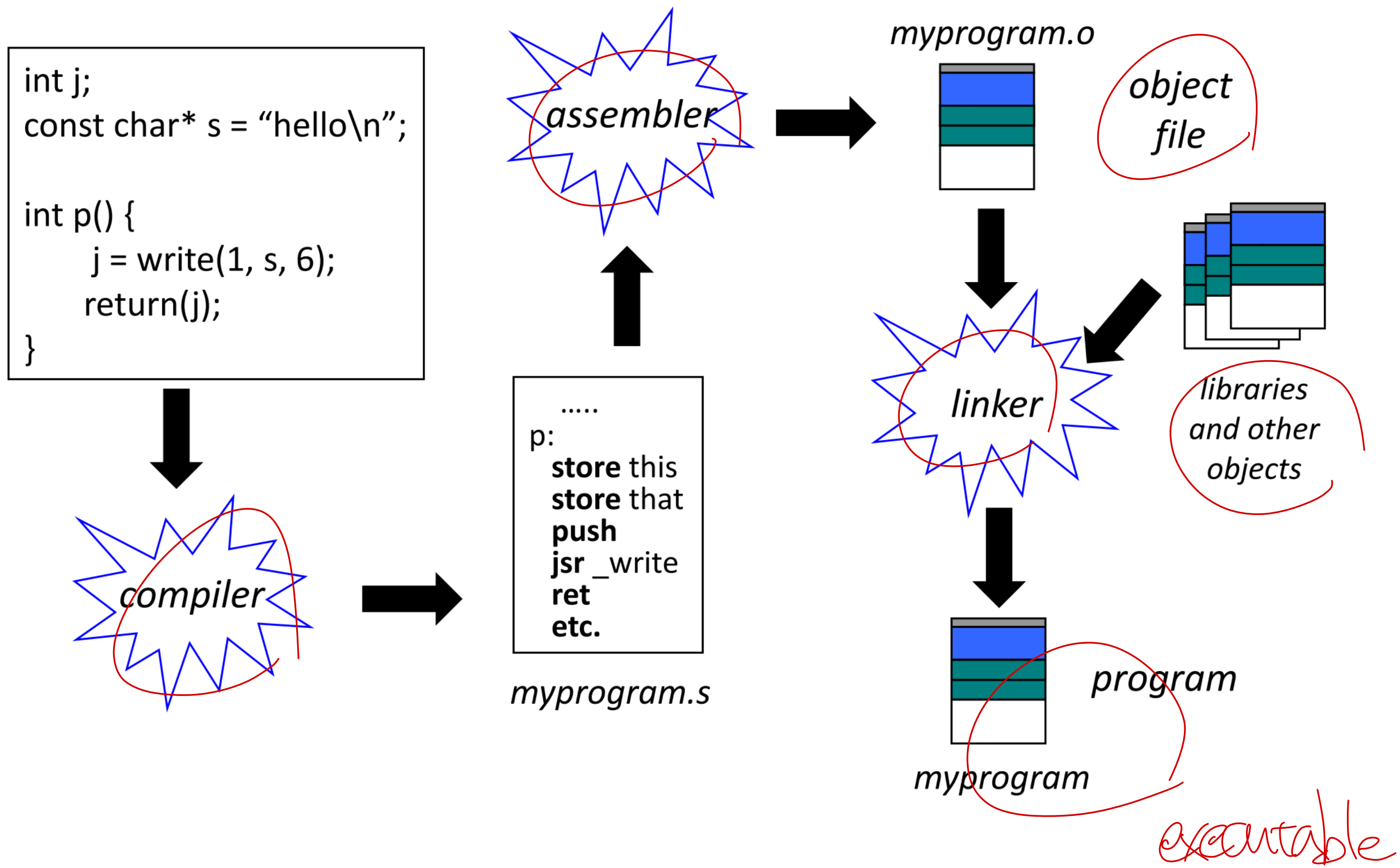
Question

- A process is an abstraction of (machine)

CPU + Memory

(Thread) (Address space)

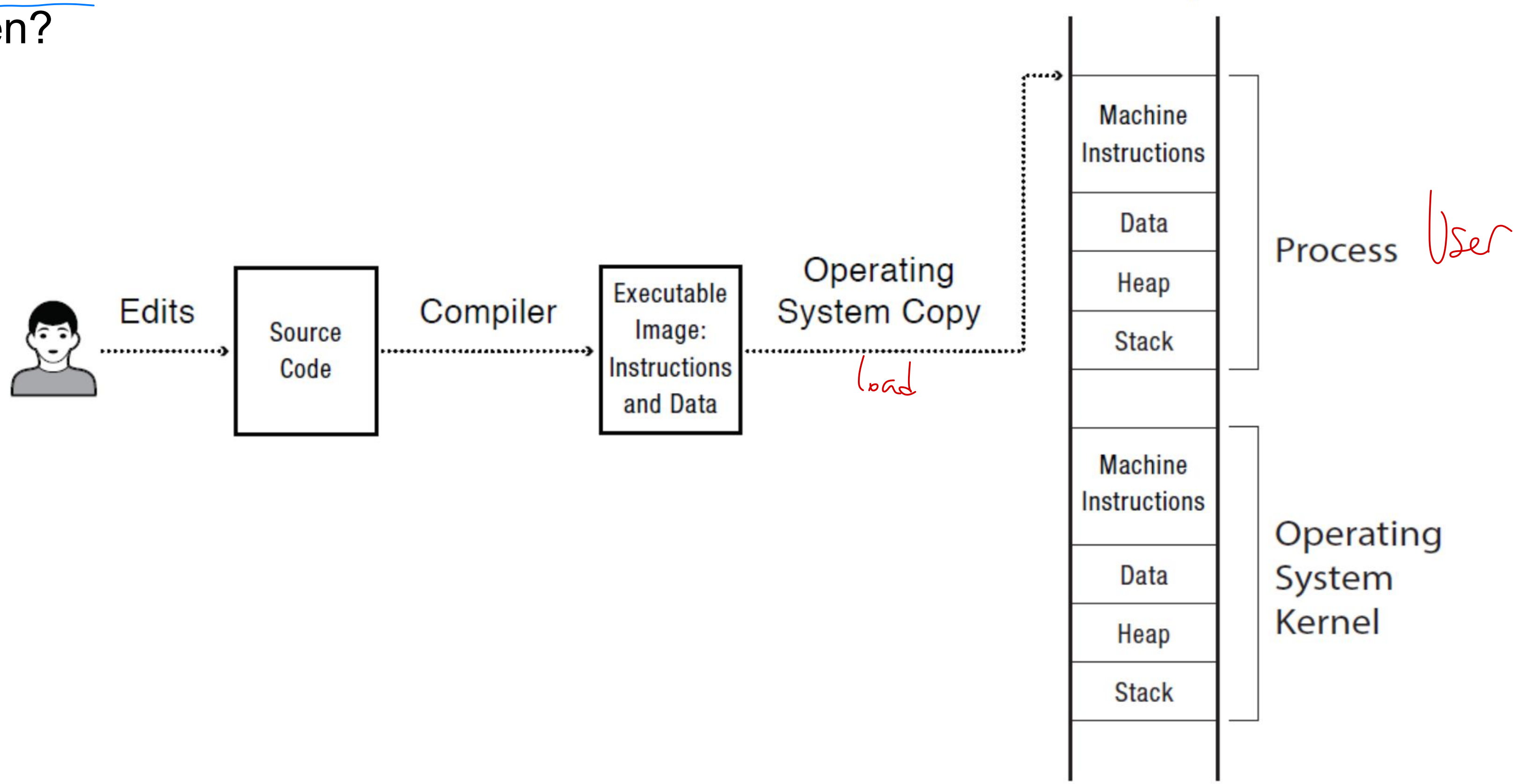
The birth of a program



A Protection problem

달성하다

Archived the separation btw process and OS kernel,
Then?



Design: how to archive protection?

- Preventing applications from executing some important **instructions**

  → limited privileged instructions

- Preventing applications from reading/writing other applications' **memory**

  → memory protection

- OS must **regain control** from applications

  → (Timer) interrupt

Requirements for protection

- Preventing applications from executing some important **instructions**

- Preventing applications from reading/writing other applications' memory

- OS must regain control from applications

move data, memory x

Simulator

x is allowed

go

else

CPU

stop

OS design thinking

- How can we implement execution with limited privilege?
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript and other interpreted languages
- How do we go faster?
  - Run the unprivileged code directly on the CPU!

It is easier to make system secure
if you make everything (super) slow!

- How to archive efficient (without hurting performance) protection?
  - Software only → Slow
  - So?

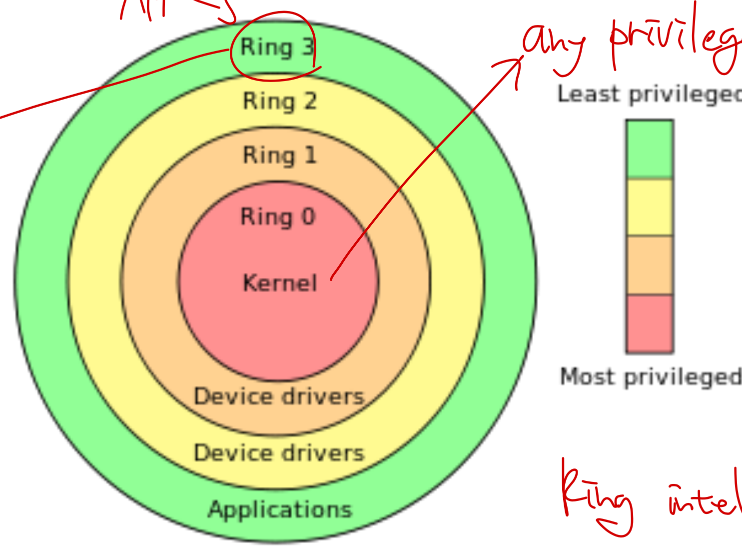
Hardware Protection Mechanisms

- Protection mechanisms
  - Dual mode operation (or ring mode)
    - mode bit is provided by hardware
  - Privilege I/O instructions
  - Memory protection mechanism (later in this semester)

App

any privileged instructions allowed

subset of benign allow



Ring intel system

Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - Limited privileges
  - CPU checks every instructions before executing them
  - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register

0 → Kernel
1 → User

Privileged instructions

- Examples?

interrupt into kernel code

Accessing unauthorized memory

"Segmentation fault"

- What should happen if a user program attempts to execute a privileged instruction?

→ CPU sends an exception to OS.

- What if an application want to run some privileged instructions? → Syscall and make kernel do that.
  - E.g., storing data to HDD (I/O instructions are privileged)

IN—Input from Port

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E4 *ib*	IN AL, *imm8*	I	Valid	Valid	Input byte from *imm8* I/O port address into AL.
E5 *ib*	IN AX, *imm8*	I	Valid	Valid	Input word from *imm8* I/O port address into AX.
E5 *ib*	IN EAX, *imm8*	I	Valid	Valid	Input dword from *imm8* I/O port address into EAX.

Protected Mode Exceptions

User level

Kernel

#GP(0) If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

#UD If the LOCK prefix is used.

- What if an application want to run some privileged instructions?
  - E.g., storing data to HDD (I/O instructions are privileged)

→ Ask kernel to do the privileged instructions

(System call)

Design: how to do it?

- Applications are allowed to use a special instruction called *system call*

- What the applications execute it, the system call causes mode switch to kernel
  - Unprivileged mode to privileged mode

System Calls

→ Used by Application

- Programming interface to the services provided by the OS

- Typically written in a high-level language (C or C++)

  Standard library convert it into systemcall

- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use

  ex. printf → Write system call

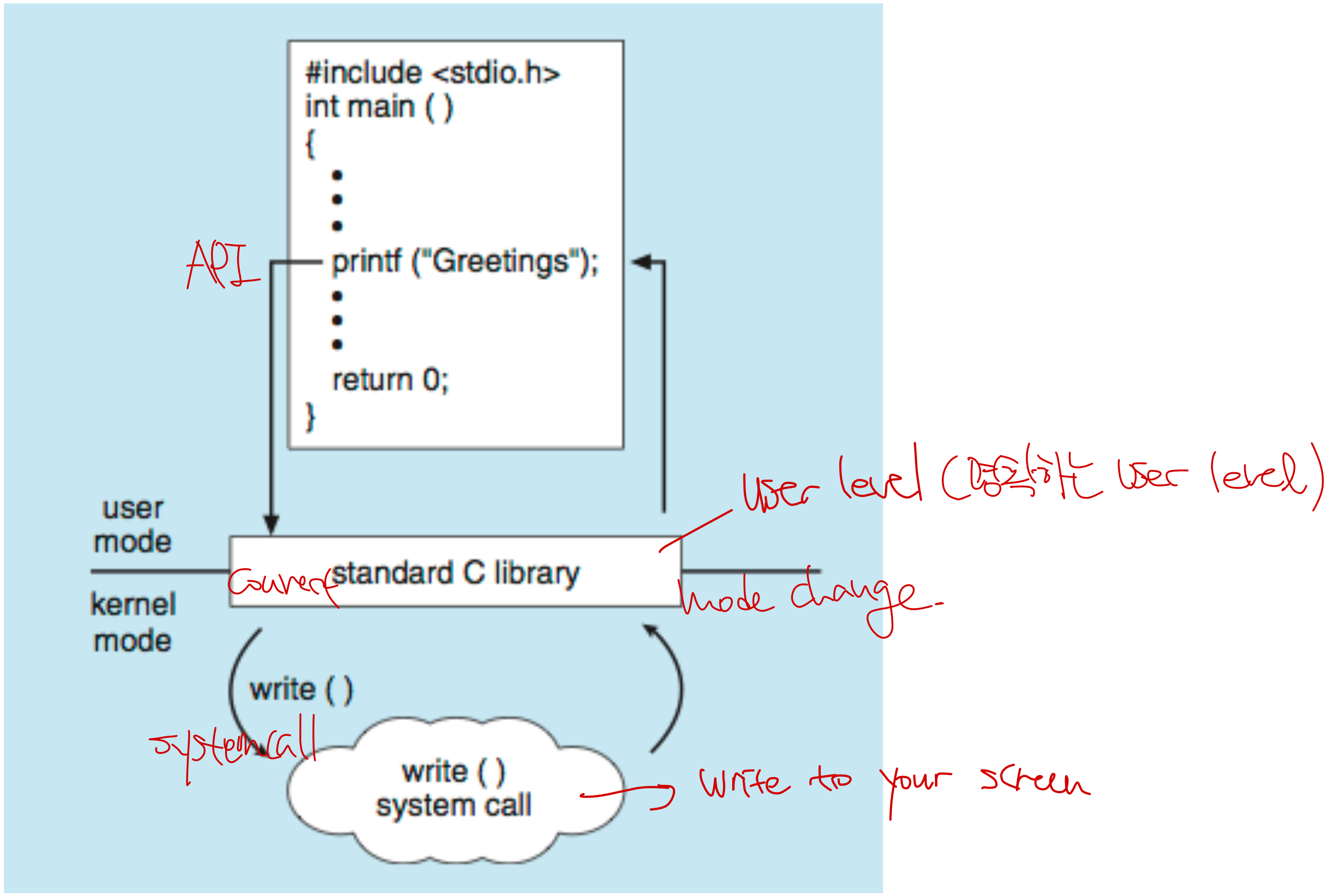
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

- Why use APIs rather than system calls? App programmers

  → easy, hiding HW, abstraction, Don't have to understand all HW mechanism

Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Requirements for protection

- **Privileged instruction**
  - Preventing applications from executing some important **instructions**
- **Memory protection**
  - Preventing applications from reading/writing other applications' **memory**
- **(Timer) interrupt**
  - OS must regain control from applications

Question

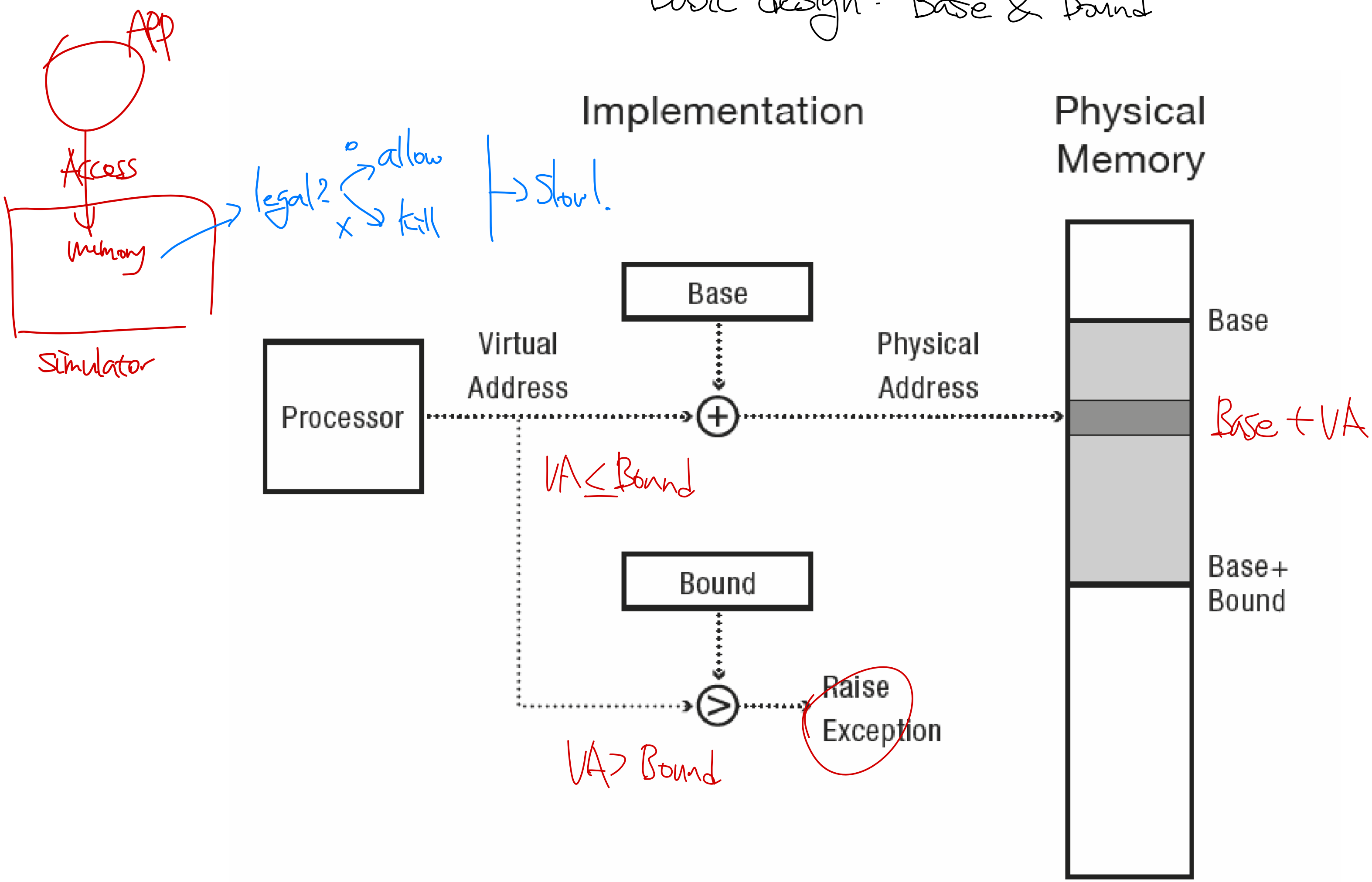
- For a "Hello world" program, the kernel must copy the string from the user program memory into the screen memory.

- Why does OS not allow the application to write directly to the screen's buffer memory?

→ Protection (Might crash the whole system)

Memory Protection using hardware

Basic design: Base & Bound



Towards Virtual Addresses

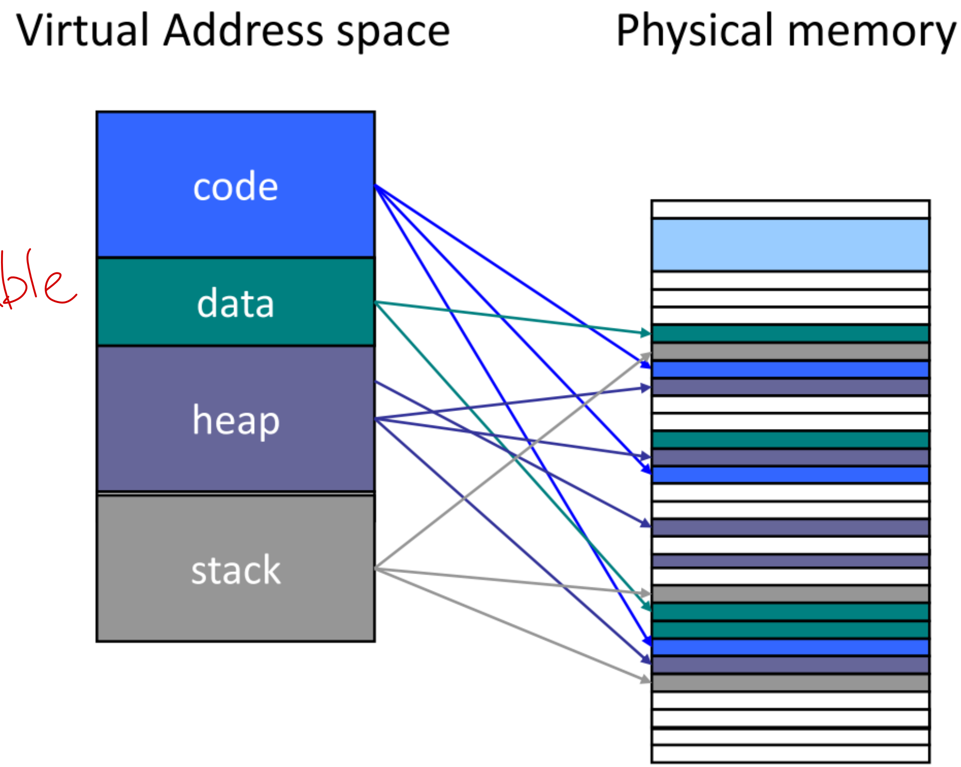
- Problems with base and bounds?

  Expandable heap?
  Expandable stack?
  Memory sharing between processes?
  Memory fragmentation

Why H/W is more privileged(Secure) than SW?: CPU가 execute하는 instruction 모음.

Virtual Addresses

- Translation done in hardware, using a table Page Table
- Table set up by operating system kernel



Quiz: What is proper a virtual address ranges of processes, P1 and P2?

- Assume each process has virtual address of the size of 0x30000

- Case 1:
  - P1: 0x00000 – 0x30000
  - P2: 0x30001 – 0x60001

- Case 2:
  - P1: 0x00000 – 0x30000
  - P2: 0x00000 – 0x30000

  independent

- Case 3:
  - P1: 0x00000 – 0x30000
  - P2: 0x10000 – 0x40000

Example

```
int staticVar = 0;      // a static variable
main() {
    staticVar += 1;
    sleep(10);  // sleep for x seconds
    printf ("static address: %x, value: %d\n",
                              &staticVar, staticVar);
}
```

What happens if we run two instances of this program at the same time?

same

What if we took the address of a procedure local variable in two copies of the same program running at the same time?

same

Summary of Hardware-supported protection

- CPU instructions
  - CPU provides two modes: privileged and unprivileged

Checks instruction before execute it.

- Memory access
  - CPU provides memory protection hardware
    - We will call this paging hardware in later lectures

Requirement for protection

- **Privileged instruction**
  - Preventing applications from executing some important **instructions**
- **Memory protection**
  - Preventing applications from reading/writing other applications' **memory**
- **(Timer) interrupt**
  - OS must regain control from applications

Design: how to regain control?

Case

독차지하다. ex

- A process hogs CPU (~~e.g.~~, infinite loop)
  - No chance to run a kernel code
  - How to regain control?

timer interrupt

- CPU makes kernel code run periodically
  - CPU delivers a hardware signal (interrupt) to kernel
    - Why not the process?

CPU clock?

CPU $\xrightarrow{\text{HW signal}}$ Kernel

  - Then, the process stop its execution and a registered kernel code (interrupt handler) is executed

↳ handling routine

Hardware timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion (will learn later)

Mode switch: User → Kernel

- From user mode to kernel mode
  - Interrupts
    - Triggered by hardware (e.g., timer and I/O)
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (can be classified as exception)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

interrupt, exception, or none of them?

- Keyboard signal → I
- Control + C → I, ~> kernel raise ~~software~~ exception
- Segmentation fault! → E
- Send a signal to a process → N
- A packet arrived to network card → I.
- TLB flush → N

Mode switch: Kernel → User

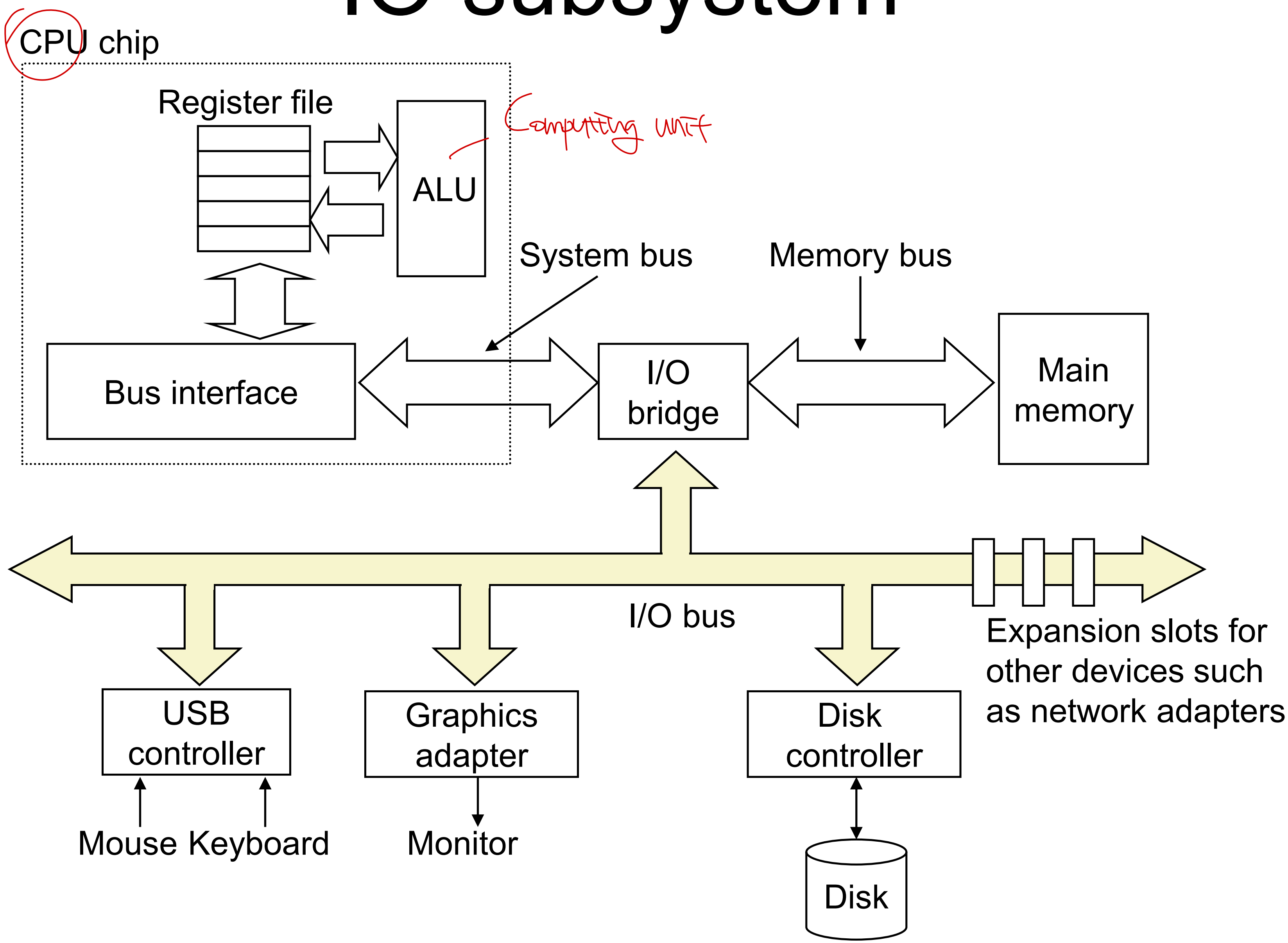
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
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

Wrap-up

- **Privileged instruction**
  – Preventing applications from executing some important instructions
- **Memory protection**
  – Preventing applications from reading/writing other applications' memory
- **(Timer) interrupt**
  – OS must regain control from applications

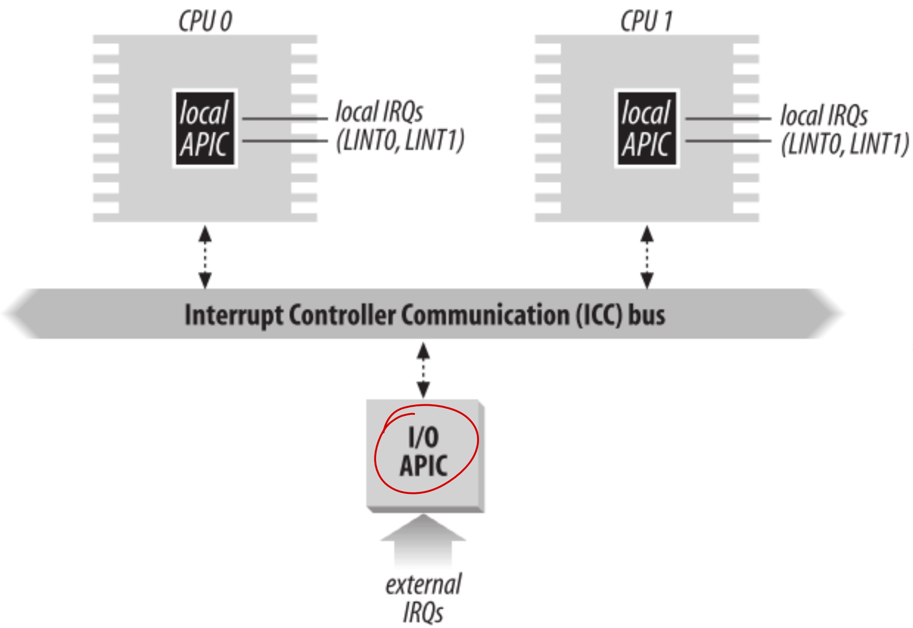
Implementing Mode Switches

IO subsystem



Hardware details: Interrupt and IRQ

Interrupt ReQuest (IRQ): Each hardware has a single output line connected to IRQ line
All IRQ lines are connected to hardware circuit called Programmable Interrupt Controller (PIC)



- Each hardware assigned an IRQ number (0 to 255)
  - Check 'cat /proc/interrupts'
- Also called *interrupt number*, *interrupt vector* (or *exception vector*)

Quick tour of I/O path (System programming)

- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - Polling: Kernel waits until I/O is done
  - Interrupts: Kernel can do other work in the meantime
- Device access to memory

  → CPU 자원 낭비
  - Programmed I/O: CPU reads and writes to device
  - Direct memory access (DMA) by device → CPU 개입 X
  - Buffer descriptor: sequence of DMA's
    - E.g., packet header and packet body
  - Queue of buffer descriptors
    - Buffer descriptor itself is DMA'ed

Device Interrupts

- How do device interrupts work? kernel memory
  - Where is the code to run to handle an interrupt?
  - What stack does it use? → kernel stack
  - Is the work the CPU had been doing before the interrupt lost forever?
    - If not, how does the CPU know how to resume that work?

Push into kernel stack to store it

How do we take interrupts safely?

- Interrupt vector → interrupt number
  - Limited number of entry points into kernel
- "Atomic" transfer of control
  - Single instruction to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode

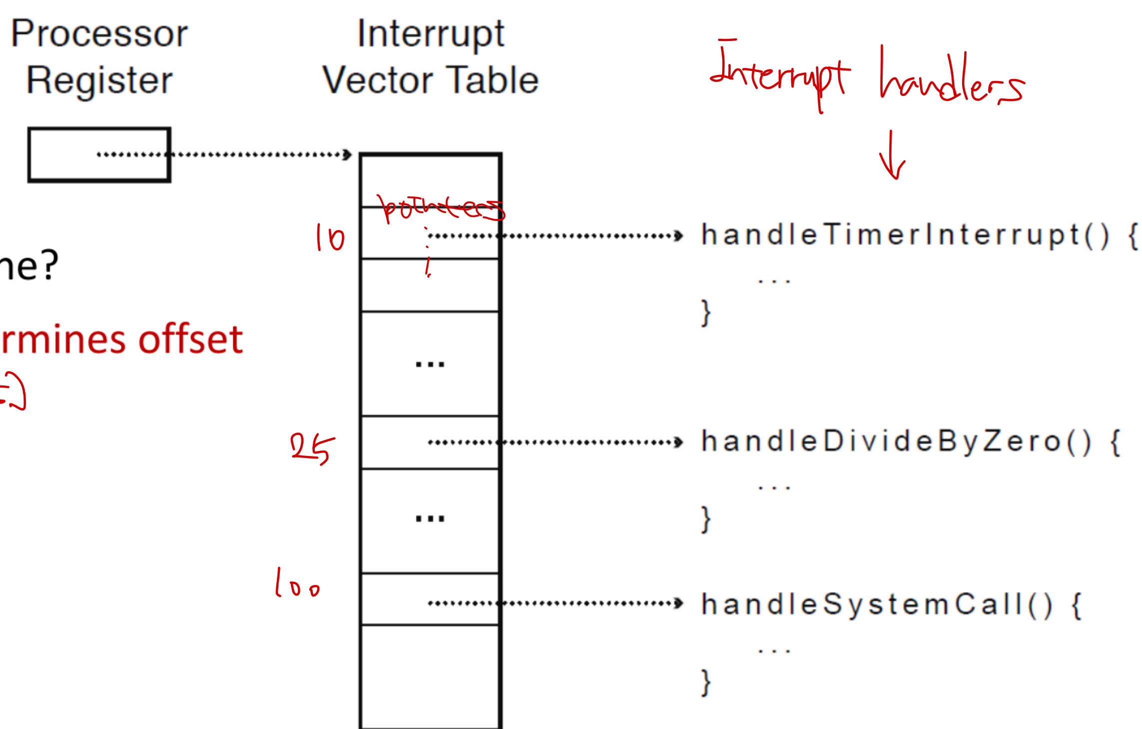
있는데 없는것처럼 보임 ⟷ virtual
없는데 있는것처럼보임.

- Transparent re-startable execution
  - User program does not know interrupt occurred

Interrupt Vector

interrupt 루틴의 주소를 담고있는 테이블

- Table set up by OS kernel; pointers to code to run on different events



Which one?

IRQ determines offset

[0, 255]

A: stack

Q: How do kernel distinguish each interrupt?

Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack

Kernel stack handles interrupts

why interrupt handle in kernel stack instead of user stack

① Reliability ② Security (Protection)

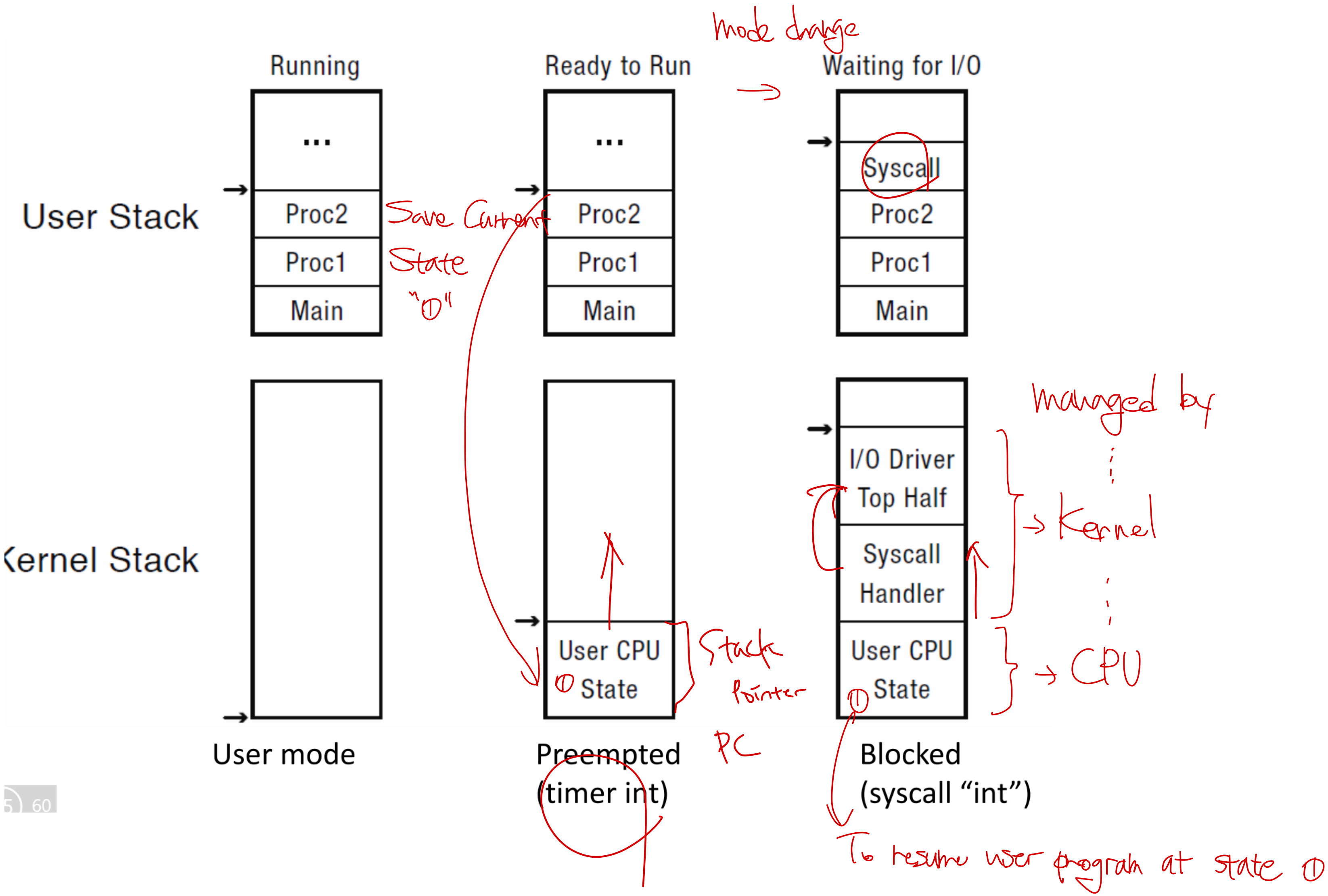
- Why can't the interrupt handler run on the stack of the interrupted user process?

Buggy user program

Kernel states are exposed

CLI, STI

Interrupt Stack



Interrupt Masking

CLI ← privileged Instruction

- Interrupt handler runs with interrupts off
  - STI – Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrupts (Clear Interrupt)
    - STI: enable interrupts (SeT Interrupt)
    - Only applies to the current CPU (on a multicore)

Case Study: x86 Interrupt

CPU

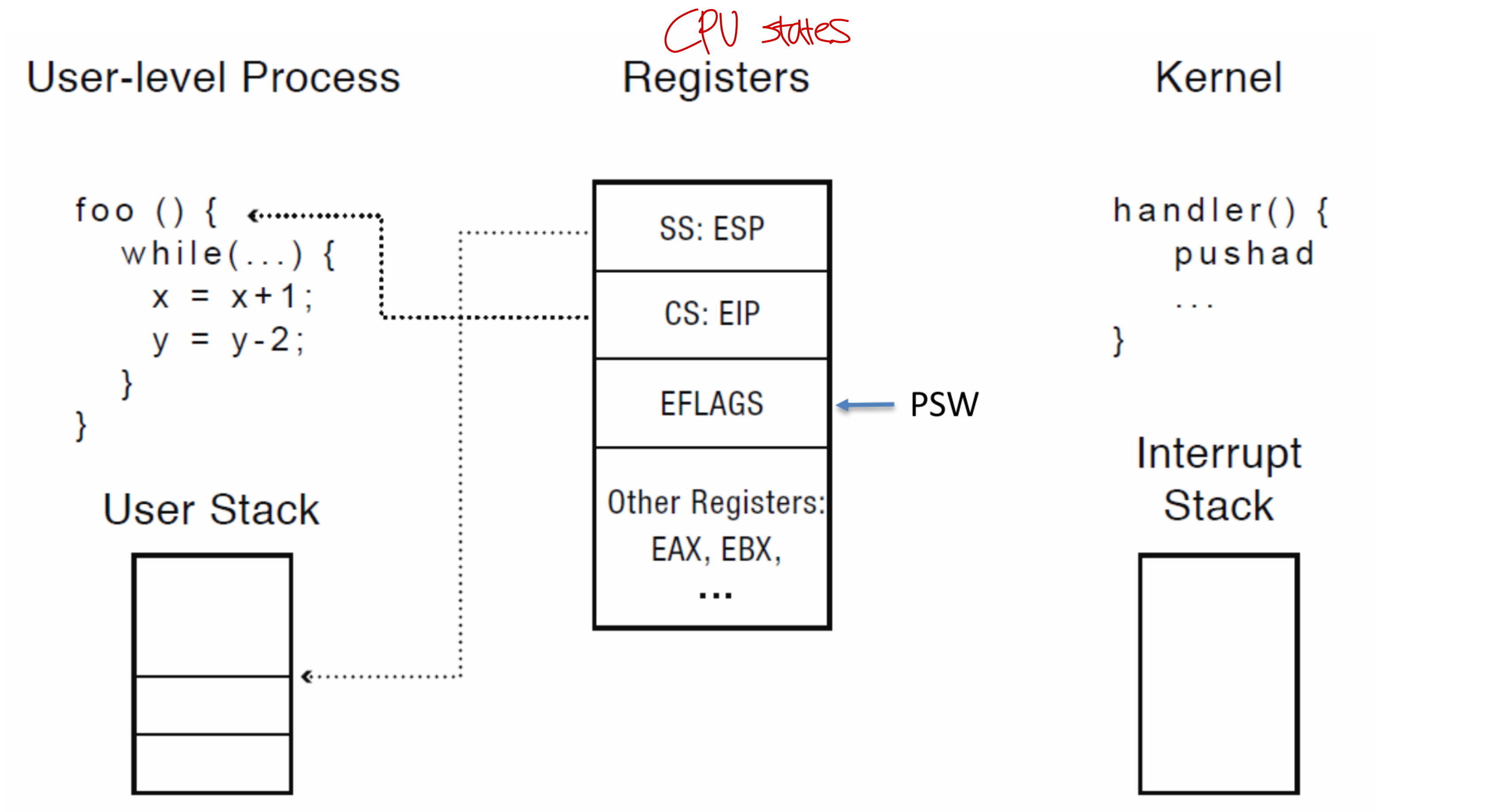
- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)

CPU(?)
HW(?)

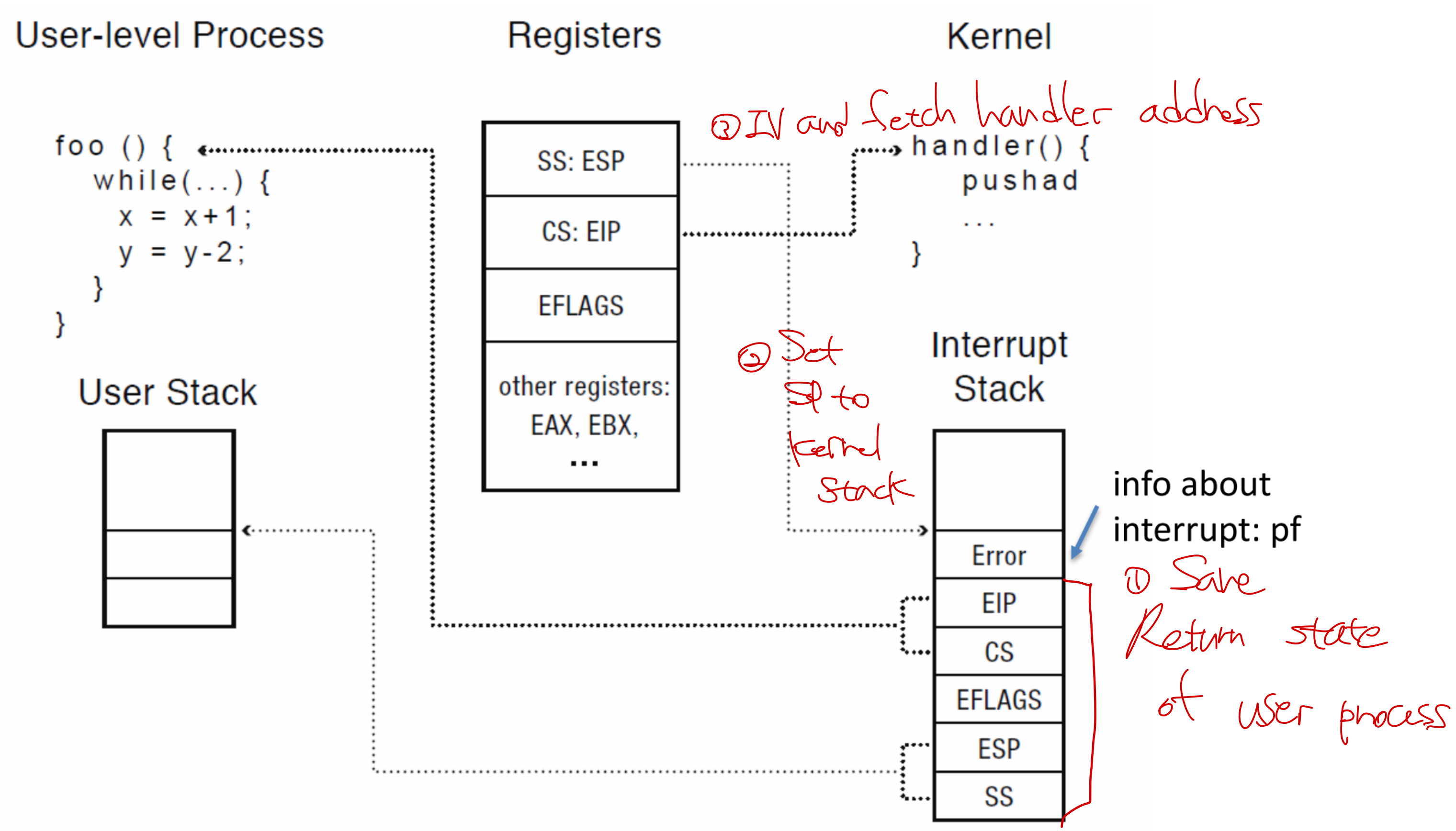
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Interrupt handler saves registers it might clobber

Kernel

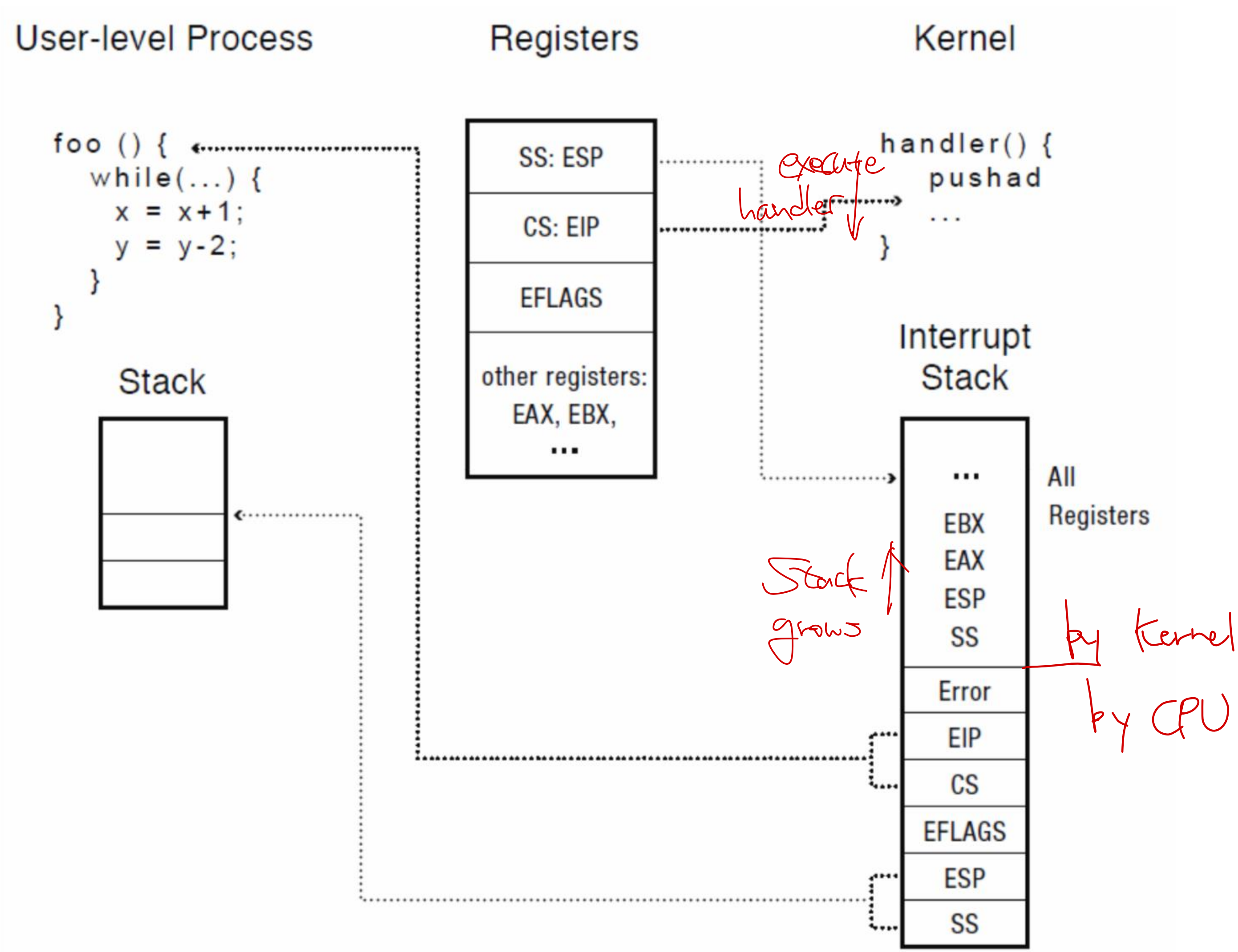
Before Interrupt



Jump to Interrupt Handler

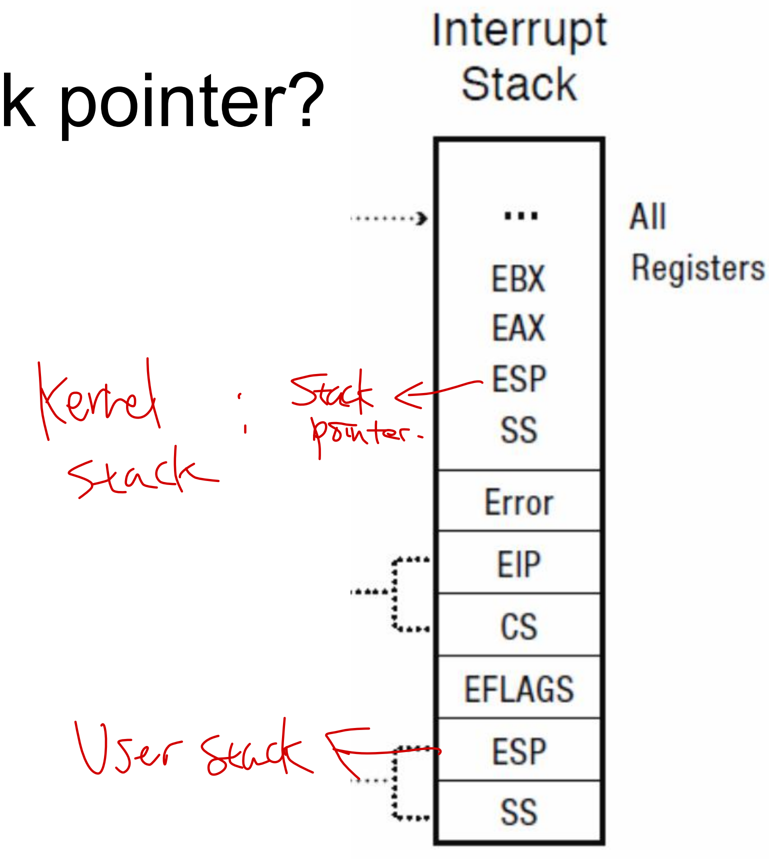


Executing the Interrupt Handler



Question

- Why is the stack pointer saved twice on the interrupt stack?
  - Hint: is it the same stack pointer?



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

How to jump from kernel to user-level?

- X86 does not allow direct jump from kernel to user-level
  - Kernel must use ***iret*** instruction
  - X86 CPU has a protocol when executing iret
- Before executing iret, CPU expects a specific stack frame, called "exception parameter"

	low address	When iret is executed,
RSP →	**RIP**	The instruction to continue execute
	CS	The code segment selector to change to
	EFLAGS	The value eflags register to load
	RSP	The stack pointer to load
	SS	The stack segment selector to change to
	high address	

exception parameter

Kind of exception

Making System Calls Secure

Syscall handler (?)

User Program

```
main () {
  file_open(arg1, arg2);
}
```

(1) (6) Why? → "Secure"

Kernel

```
file_open(arg1, arg2) {
  // do operation
}
```

(3) (4)

e.g. ~ libc code ← **User Stub**

Copied to kernel

```
file_open(arg1, arg2) {
  push #SYSCALL_OPEN
  trap
  return
}
```

(2) exception

Hardware Trap

Trap Return

(5)

Kernel Stub Interrupt handler

```
file_open_handler() {
  // copy arguments
  //   from user memory
  //   check arguments
  file_open(arg1, arg2);
  // copy return value
  //   into user memory
  return;
}
```

Interrupt handling pns)

Time Of Check Time Of Use (TOCTOU)

→ Check와 Use 사이에 user가 arg value를 맬리셔스하게 바꿀 수 있으면 not secure 함.

Pintos: System call

[User-level] lib/usr/syscall.c

```
#define syscall1(NUMBER, ARG0)                                          \
        ({                                                              \
          int retval;                                                   \
          asm volatile                                                  \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                          \
               : [number] "i" (NUMBER),                                 \
                 [arg0] "g" (ARG0)                                      \
               : "memory");                                             \
          retval;                                                       \
        })
```

[Kernel-level] userprog/syscall.c

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}

static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

How does your program get the segmentation fault?

exception
CPU → kernel
↓ signal
User program

```
Closest to 7 is 7
Closest to 8 is 8
Closest to 9 is 9
Iteration 9
Correct: 10/10: 100%
Segmentation fault (core dumped)
```

Who sent this signal?

signal handler for SIGSEGV

Yoohoo. You get the SIGSEGV signal!

/usr/include/bits/signum.h

```
#define SIGKILL    9 /* Kill, unblockable (POSIX).  */
#define SIGUSR1    10  /* User-defined signal 1 (POSIX).  */
#define SIGSEGV    11  /* Segmentation violation (ANSI).  */
#define SIGUSR2    12  /* User-defined signal 2 (POSIX).  */
```

But, Who sent the SIGSEGV? : Kernel

Where is the SIGSEGV handler? → libc (your standard library)

Upcall: User-level event delivery

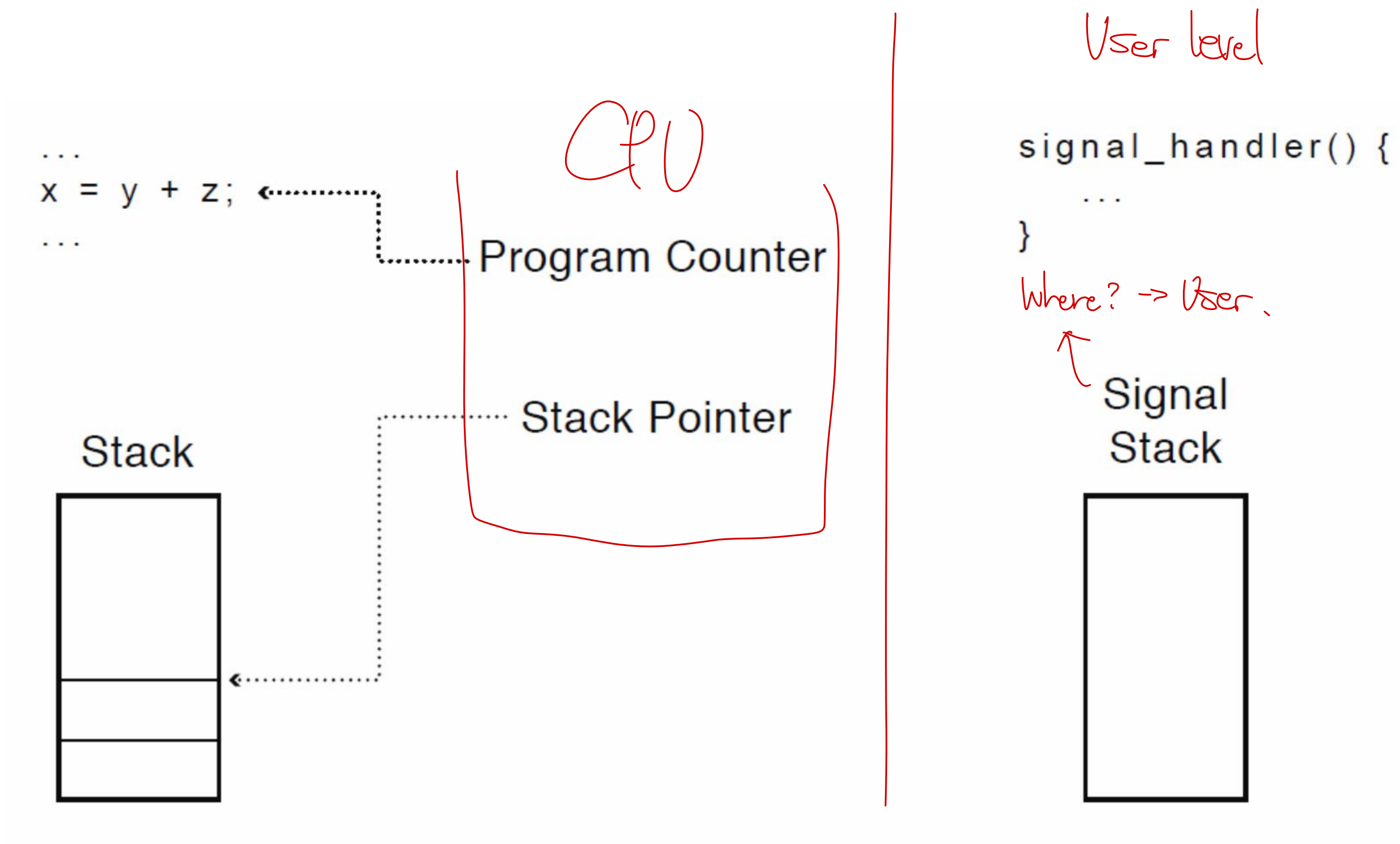
- Notify user process of some event that needs to be handled right away
  - Time expiration 만기
    - E.g., Sleep time is elapsed
    - Time-slice for user-level thread manager
  - Asynchronous I/O completion (async/await)
- AKA UNIX signal

Upcalls vs Interrupts

Similar in design

- Signal handlers = interrupt ~~vector~~ handler
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before



Upcall: During