Operating Systems
Lecture 3

Context Switch

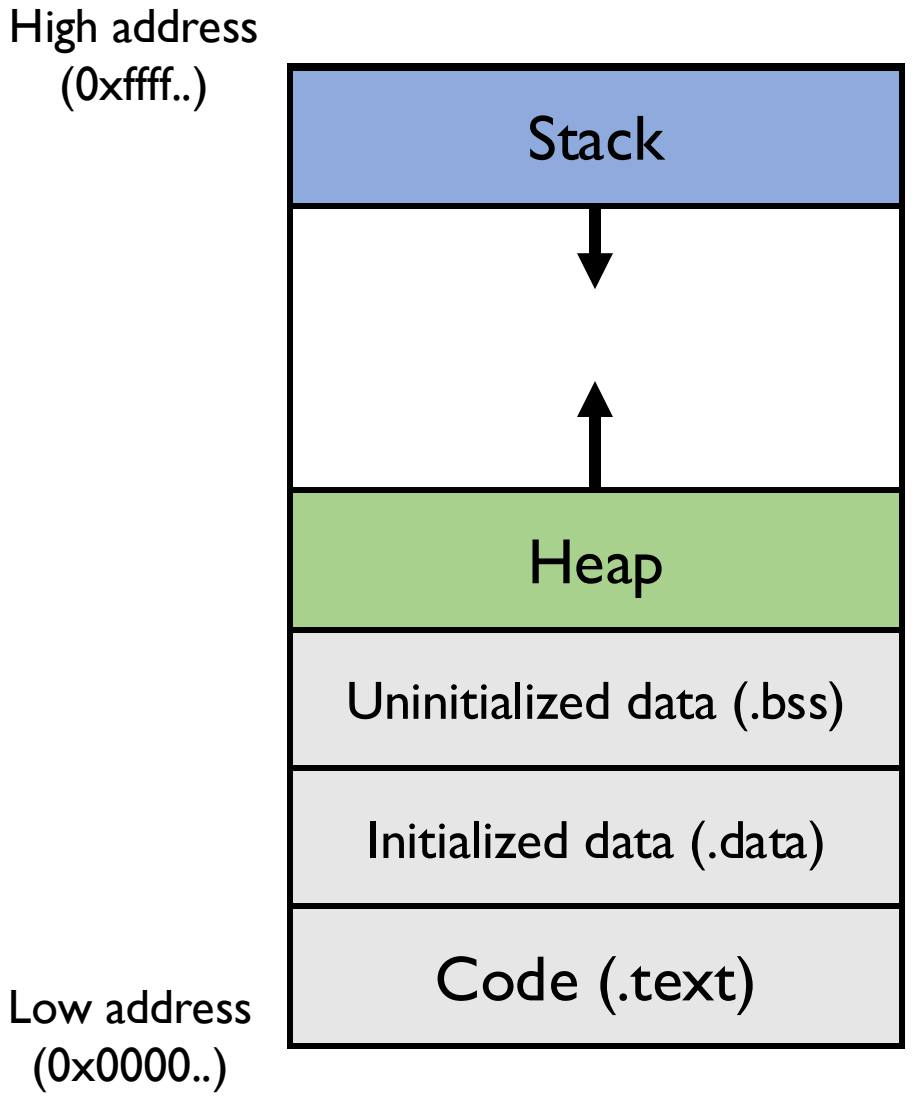
Prof. Mengwei Xu

Recap of Last Course

BIOS	Bootloader
Firmware, comes with HW	Software, comes with (or part of) OS
The first software that runs since power on	The first user-defined or user-changeable software that runs since power on
Usually stored on ROM and not changeable	Stored with OS (hard disk, USB, etc)

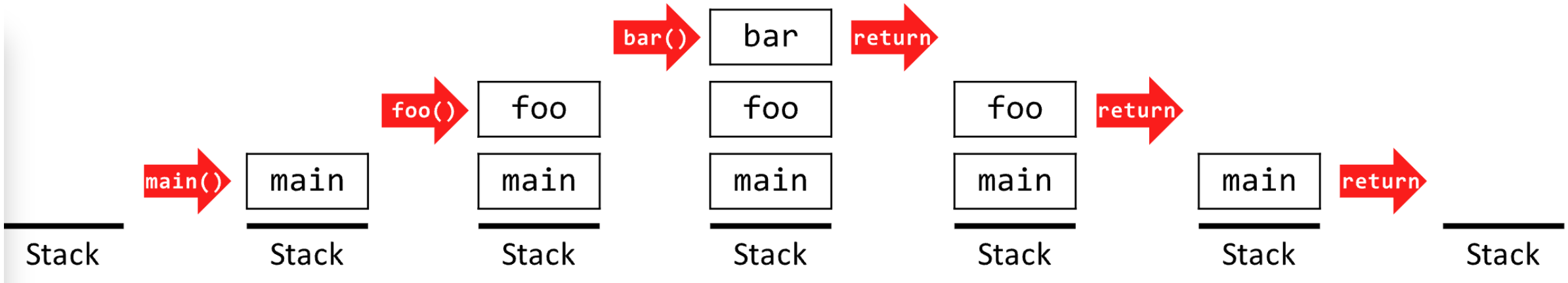
Recap of Last Course

- An executable mainly consists of bss, data, and code regions.
- Remember: this memory address is NOT physical!
  - Will learn how it's translated into physical address later.



Recap of Last Course

- What does a stack store



Dual Mode

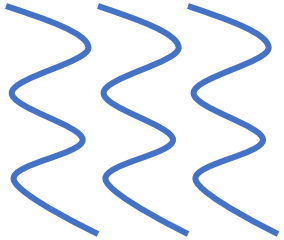
- Hardware-assisted isolation and protection
  - User mode (用户态) vs. kernel mode (内核态)
  - Teachers & TAs are in ?? mode, while students are in ?? mode

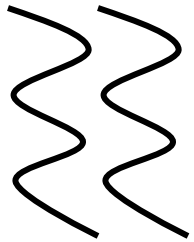
- What hardware needs to provide?
  - Privileged instructions (特权指令)
  - Memory protection
  - Timer interrupts
  - Safe mode transfer (this course)

Concepts

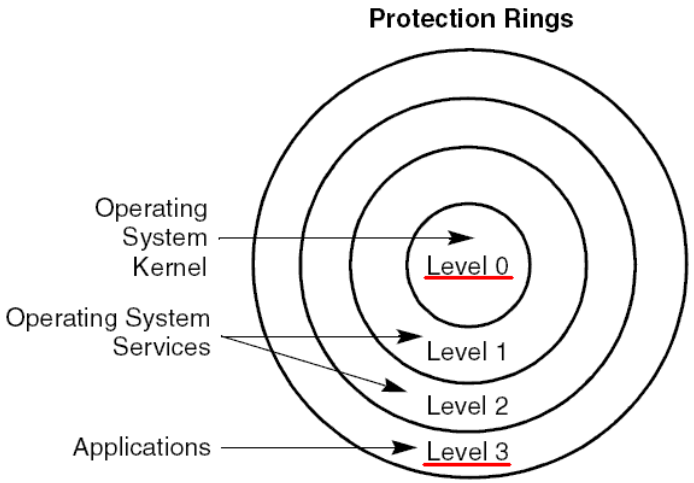
- user/app code vs. user/app process vs. user mode
- OS code vs. system process vs. kernel mode

Code





Process



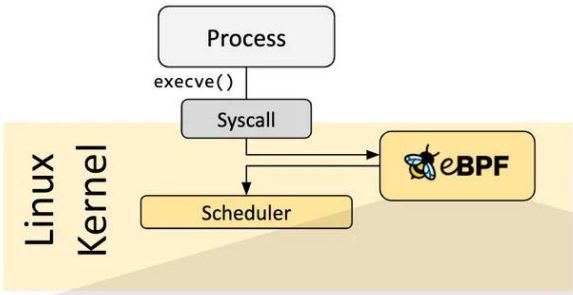
Mode

Some Interesting Questions

- Does user code always run in user process?
- Does user code always run in user mode?
- Does OS code always run in system process?
- Does OS code always run in kernel mode?
- How does code/CPU know if it's in user or kernel mode?

Some Interesting Questions

- Does user code always run in user proce
  - Yes. Third-party drivers?
- Does user code always run in user mode
  - Mostly, except eBPF
- Does OS code always run in system proc
  - No. Interrupt handler
- Does OS code always run in kernel mode
  - No. Shells, UI components, etc..




```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

- How does code/CPU know if it's in user or kernel mode?
  - Last 2 bits in cs segment selector

Goals for Today

- User-kernel mode switch types
  - Exceptions, interrupts, and system calls
- An x86 example of mode transfer

Goals for Today

- **User-kernel mode switch types**
  - Exceptions, interrupts, and system calls
- An x86 example of mode transfer

User-to-kernel Mode Switch

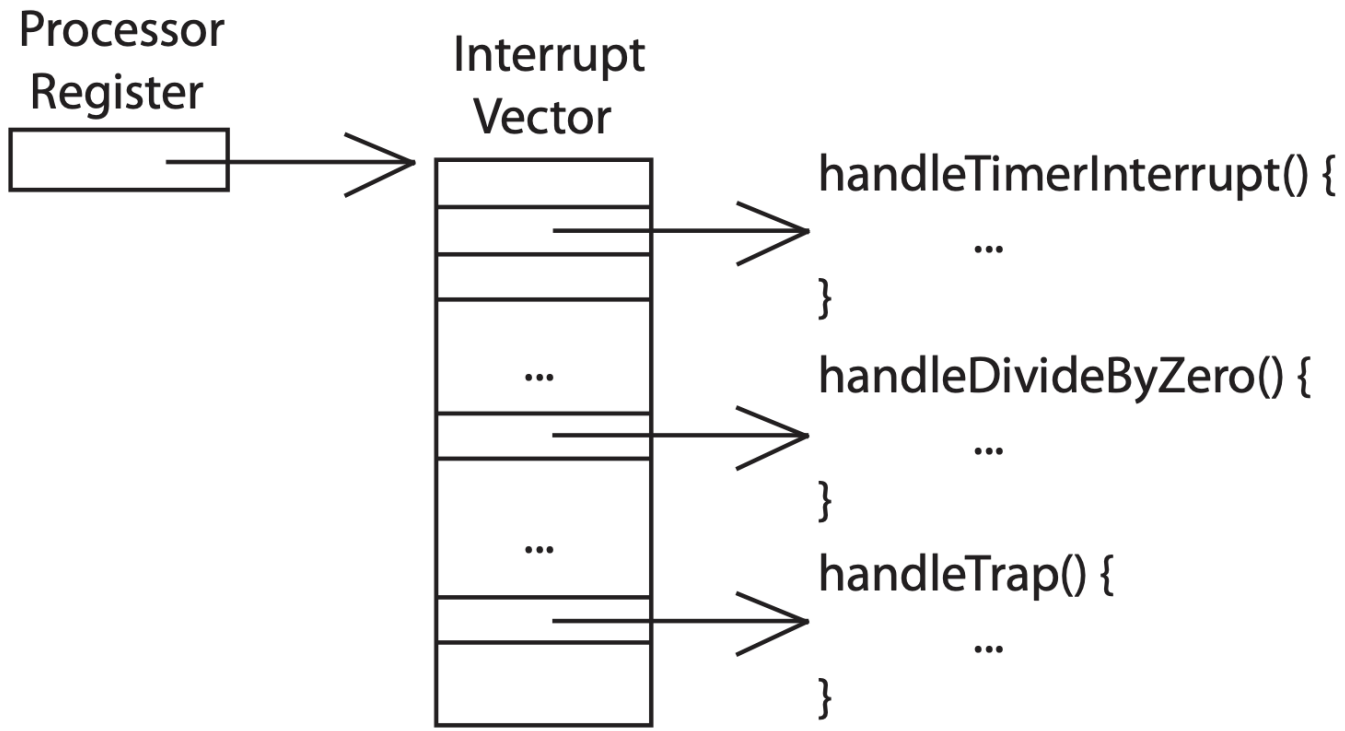
- Exceptions (异常)
  - When the processor encounters unexpected condition.
  - Illegal memory access, divide-by-zero, perform privileged instructions, etc..
- Interrupts (中断)
  - Asynchronous (异步) signal to the processor that some external event has occurred that may require its attention.
  - Timer interrupts, I/O requests such as mouse movement/clicks, etc..
- System calls (系统调用, trap)
  - User processes request the kernel do some operation on the user's behalf.
  - R/W files, create new processes, network connections, etc..

User-to-kernel Mode Switch

- What types of user-to-kernel switch are they (if any)?
  - Inter-processor interrupt (IPI)
  - Invalid opcode
  - Segmentation fault
  - Network card interrupt
  - Divide-by-zero in Python/Java

Interrupt Vector Table

- Interrupt vector table (中断向量表) stores the entries of different handlers for exceptions, interrupts, and traps in real mode.
  - A special register that points to a vector in kernel memory, where each entry points to the first instruction of a different handler procedure in the kernel.



Interrupt Vector Table

- Interrupt vector table (中断向量表) stores the entries of different handlers for exceptions, interrupts, and traps in real mode.
  - In x86, there are 256 entries in total. Each takes 4 bytes.

CPU Interrupt Layout

IVT Offset	INT #	Description
0x0000	0x00	Divide by 0
0x0004	0x01	Reserved
0x0008	0x02	NMI Interrupt
0x000C	0x03	Breakpoint (INT3)
0x0010	0x04	Overflow (INTO)
0x0014	0x05	Bounds range exceeded (BOUND)
0x0018	0x06	Invalid opcode (UD2)
0x001C	0x07	Device not available (WAIT/FWAIT)
0x0020	0x08	Double fault
0x0024	0x09	Coprocessor segment overrun
0x0028	0x0A	Invalid TSS
0x002C	0x0B	Segment not present
0x0030	0x0C	Stack-segment fault
0x0034	0x0D	General protection fault
0x0038	0x0E	Page fault
0x003C	0x0F	Reserved
0x0040	0x10	x87 FPU error
0x0044	0x11	Alignment check
0x0048	0x12	Machine check
0x004C	0x13	SIMD Floating-Point Exception
0x00xx	0x14-0x1F	Reserved
0x0xxx	0x20-0xFF	User definable

```
// Trap numbers
// These are processor defined:
#define T_DIVIDE     0      // divide error
#define T_DEBUG      1      // debug exception
#define T_NMI        2      // non-maskable interrupt
#define T_BRKPT      3      // breakpoint
#define T_OFLOW      4      // overflow
#define T_BOUND      5      // bounds check
#define T_ILLOP      6      // illegal opcode
#define T_DEVICE     7      // device not available
#define T_DBLFLT     8      // double fault
/* #define T_COPROC  9 */   // reserved (not generated by
#define T_TSS       10      // invalid task switch segment
#define T_SEGNP     11      // segment not present
#define T_STACK     12      // stack exception
#define T_GPFLT     13      // general protection fault
#define T_PGFLT     14      // page fault
/* #define T_RES    15 */   // reserved
#define T_FPERR     16      // floating point error
#define T_ALIGN     17      // aligment check
#define T_MCHK      18      // machine check
#define T_SIMDERR   19      // SIMD floating point error
```

Interrupt Descriptor Table

- Interrupt Descriptor Table (IDT, 中断描述符表) tells the CPU where the Interrupt Service Routines (ISR, 中断服务程序) are located
  - Entries are called ”***Gates***”, and there are 256 gates in total
  - Each gate is 8-bytes long on 32-bit processors; or16-bytes long on 64-bit processors
  - Its location is kept in **IDTR** (IDT register), loaded with ***LIDT*** assembly instruction

Interrupt Descriptor Table

- Interrupt Descriptor Table (IDT, 中断描述符表) tells the CPU where the Interrupt Service Routines (ISR, 中断服务程序) are

Gate Descriptor (32-bit):

63 48	47	46 45	44	43 40	39 32
Offset 31 16	**P**	**DPL** 1 0	0	**Gate Type** 3 0	Reserved
31 16	**15**				**0**
Segment Selector 15 0	**Offset** 15				0

- Offset: A 32-bit value, split in two parts, represents the address of the Interrupt Service Routine.
- Selector: A Segment Selector which must point to a valid code segment in your GDT.
- Gate Type: A 4-bit value which defines the type of gate this Interrupt Descriptor represents.
  - Task gate, interrupt gate, trap gate, call gate.. What's the difference?
- DPL: A 2-bit value which defines the CPU Privilege Levels which are allowed to access this interrupt via the *INT*.
- P: Present bit. Must be set (1) for the descriptor to be valid.

Interrupt Descriptor Table

- Interrupt Descriptor Table (IDT, 中断描述符表) tells the CPU where the Interrupt Service Routines (ISR, 中断服务程序) are

Gate Descriptor (32-bit):

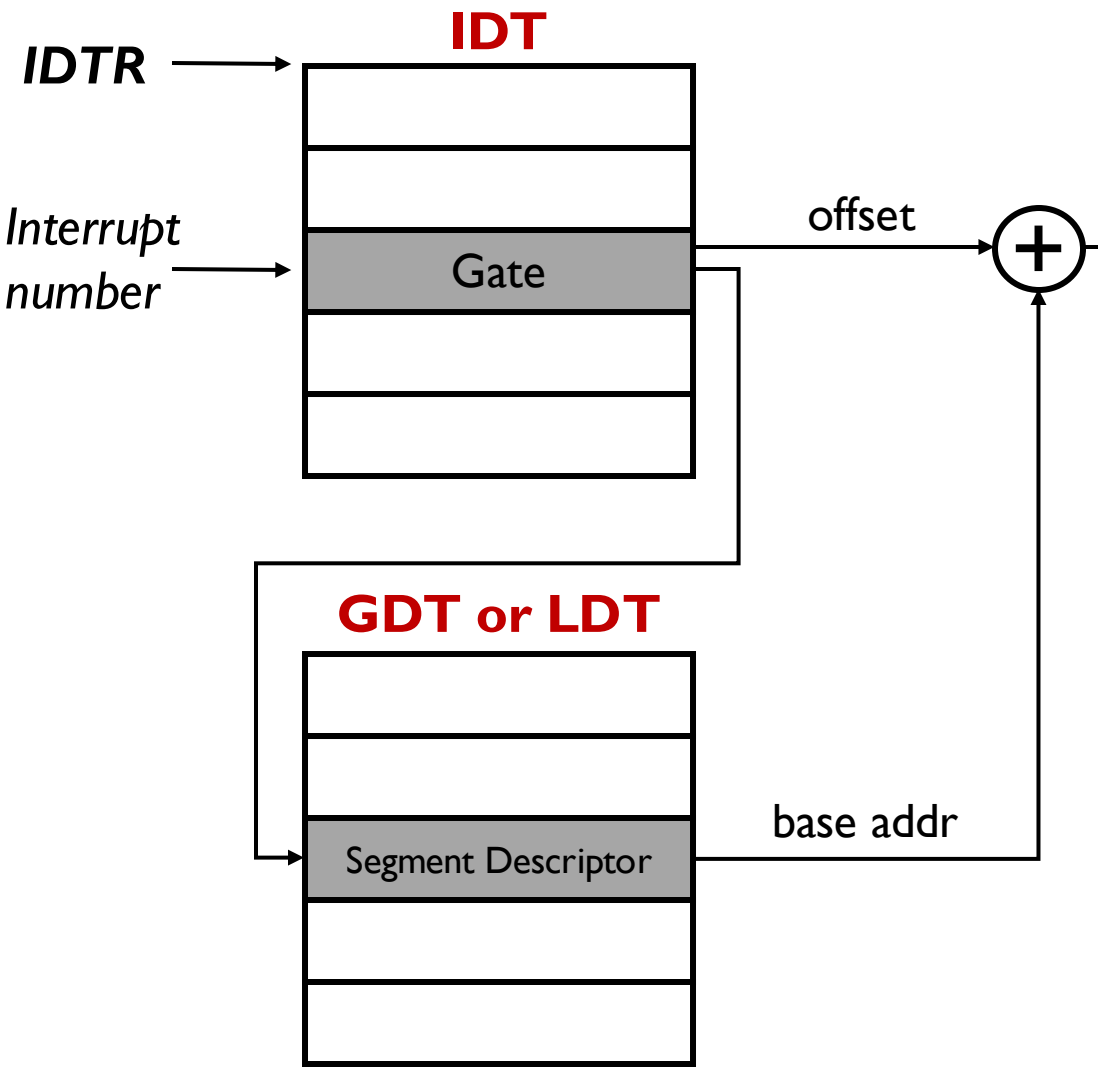
63 48	47	46 45	44	43 40	39 32
Offset 31 16	**P**	**DPL** 1 0	0	**Gate Type** 3 0	Reserved
31 16	**15**				**0**
Segment Selector 15 0	**Offset** 15				0

```
84  struct gate_struct {
85          u16             offset_low;
86          u16             segment;
87          struct idt_bits bits;
88          u16             offset_middle;
89  #ifdef CONFIG_X86_64
90          u32             offset_high;
91          u32             reserved;
92  #endif
93  } __attribute__((packed));
```

```
69  struct idt_bits {
70          u16             ist     : 3,
71                          zero    : 5,
72                          type    : 5,
73                          dpl     : 2,
74                          p       : 1;
75  } __attribute__((packed));
```

Why offset is split into two parts?

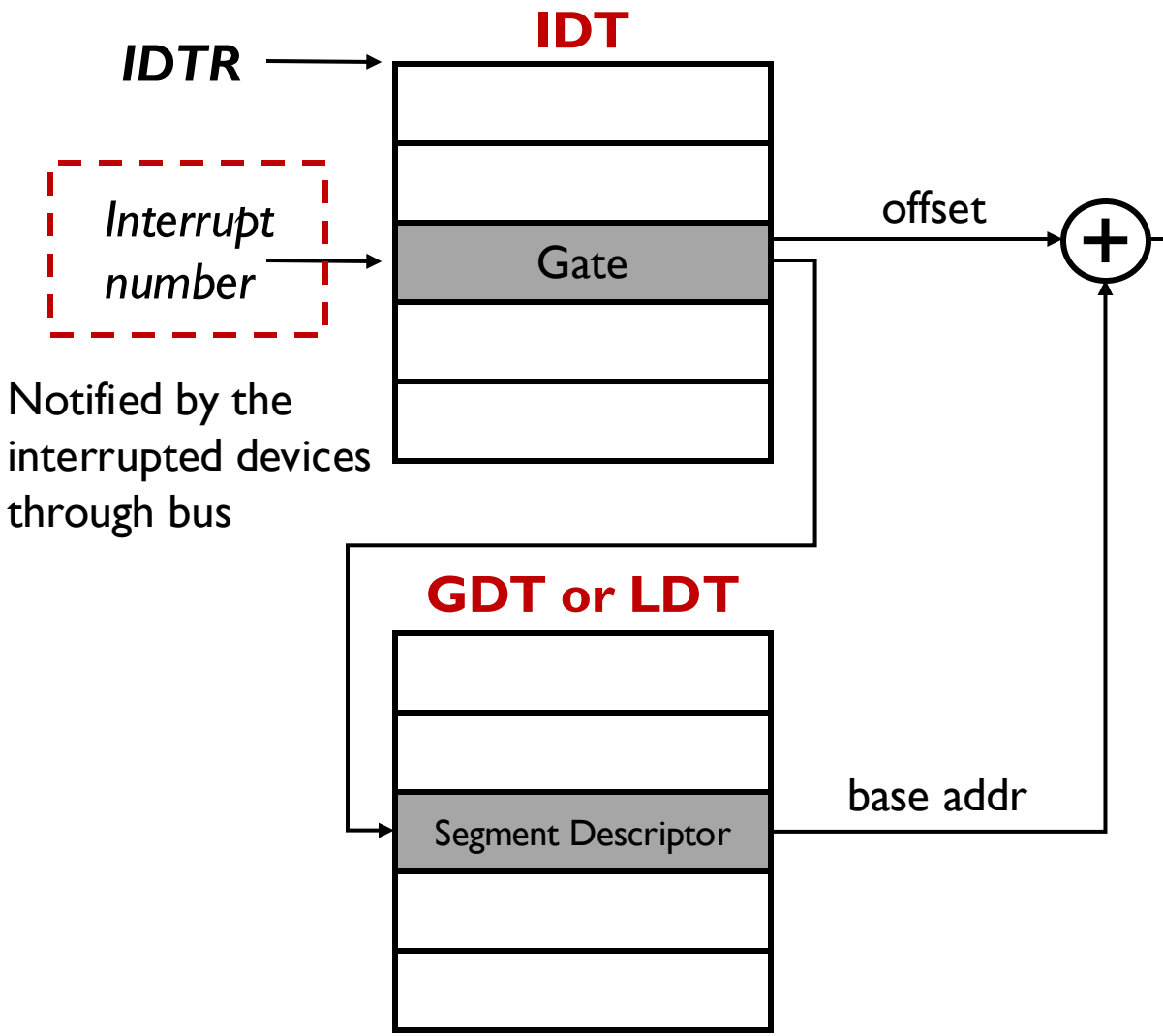
Interrupt Descriptor Table




```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if ((tf->tf_cs & 0x3) == 0)
        panic("Unhanlded page fault in kernel: %08x\n", fault_va);
```

Interrupt Procedure

Interrupt Descriptor Table




```c
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if ((tf->tf_cs & 0x3) == 0)
        panic("Unhanlded page fault in kernel: %08x\n", fault_va);
```

Interrupt Procedure

IVT vs. IDT

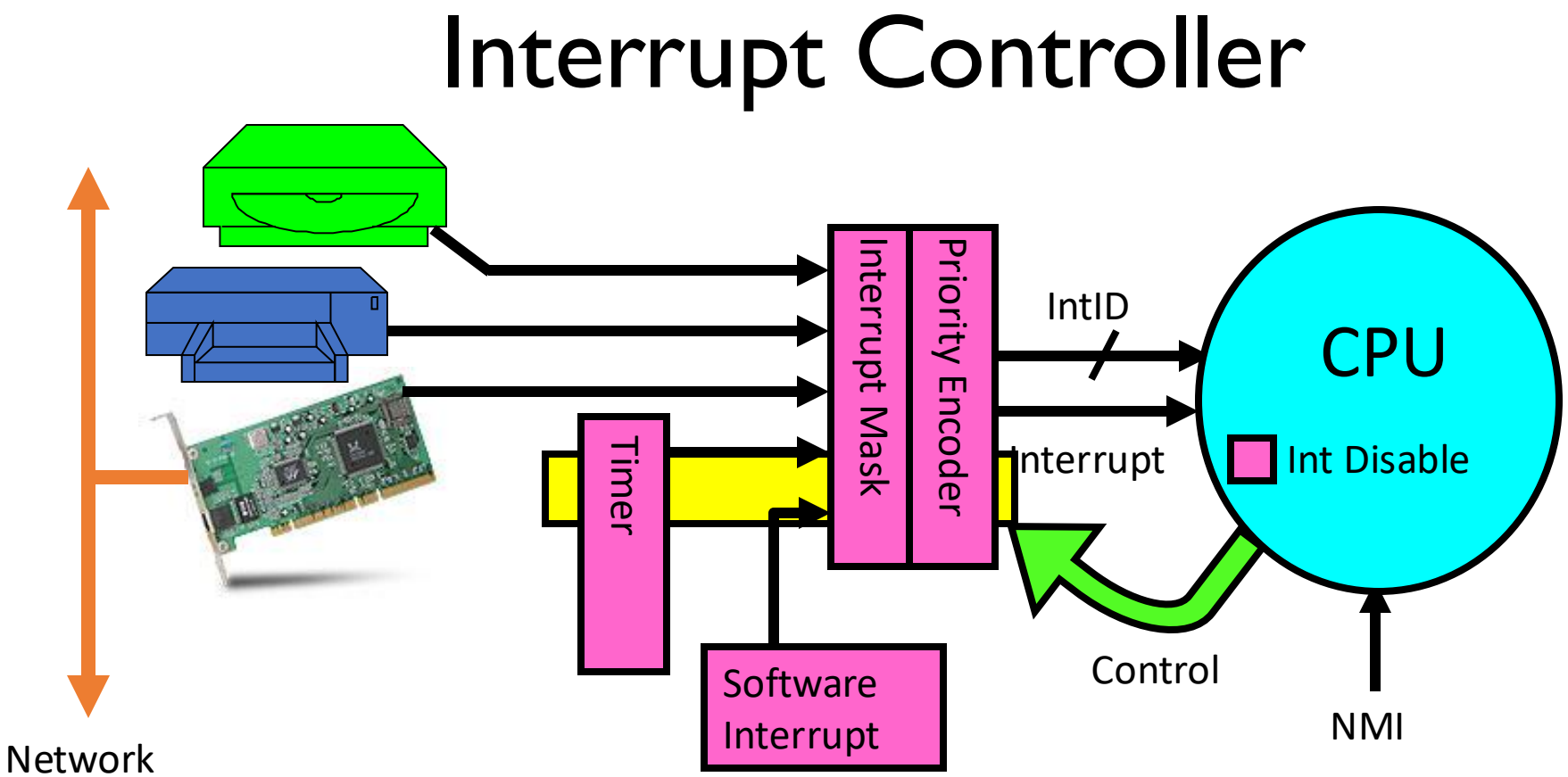
IVT	IDT
Both guarantee a limited number entries from user to kernel space (isolation)	
Used in real mode	Used in protected mode
4-byte entries	8-byte (IA-32) or 16-byte (x86-64) entries
Typically located at 0000:0000H	Anywhere in memory and located through *LIDT* instruction

Interrupt Masking

- *Disable interrupts* and *enable interrupts* are two privileged instructions
  - Maskable interrupts (可屏蔽中断): all software interrupts, all system calls, and partial hardware exceptions
  - Non-maskable interrupts (NMI，不可屏蔽中断): partial hardware exceptions
  - Specified by *eflags* registers

- Interrupts are deferred, but not ignored
  - Given the limited buffer for interrupts, hardware buffers one interrupt of each type

Interrupt Masking



Interrupt Stack

- Interrupt stack (中断栈) is a special stack in kernel memory that saves the interrupt process status.
  - Empty when there is no interrupt (running in user space)

- Why not directly use the user-space stack?
  - For reliability and security

Interrupt Stack

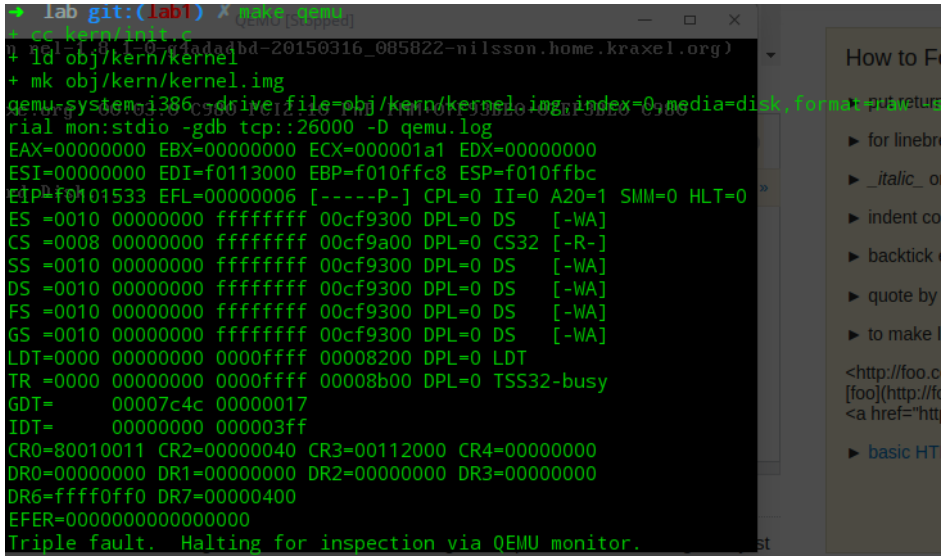
- How many interrupt stacks in kernel?

1 x # of processes (threads)

Interrupt Stack

- How many interrupt stacks in kernel?

- *First fault:* trap from user-space program to kernel-space exception handler
- *Double fault:* trap from exception handler to another handler
- *Triple fault:* reboot



Interrupt Stack

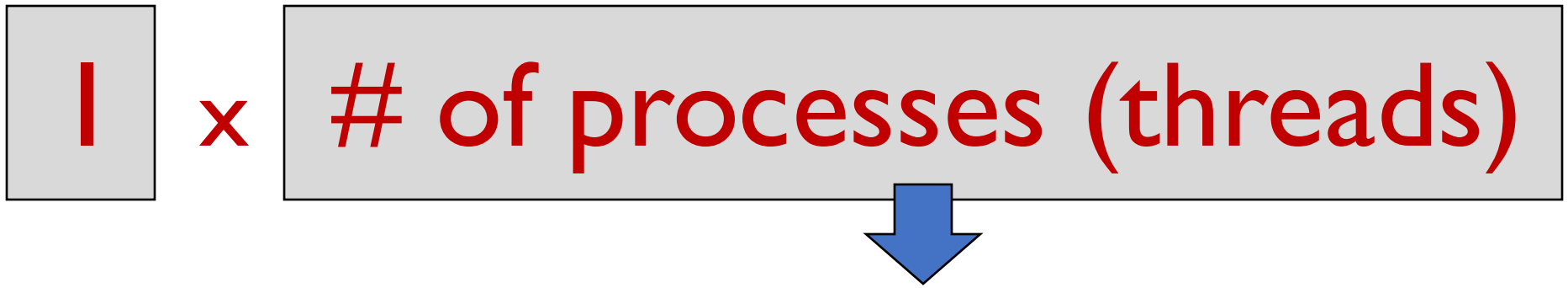
- How many interrupt stacks in kernel?

Things never to do in an OS #1: Swap out the page swapping code (triple-fault here we come).

—Kemp

Interrupt Stack

- How many interrupt stacks in kernel?



Make it easier to switch to a new process inside an interrupt or system call handler.

e.g., a handler might wait for I/O so another process could run

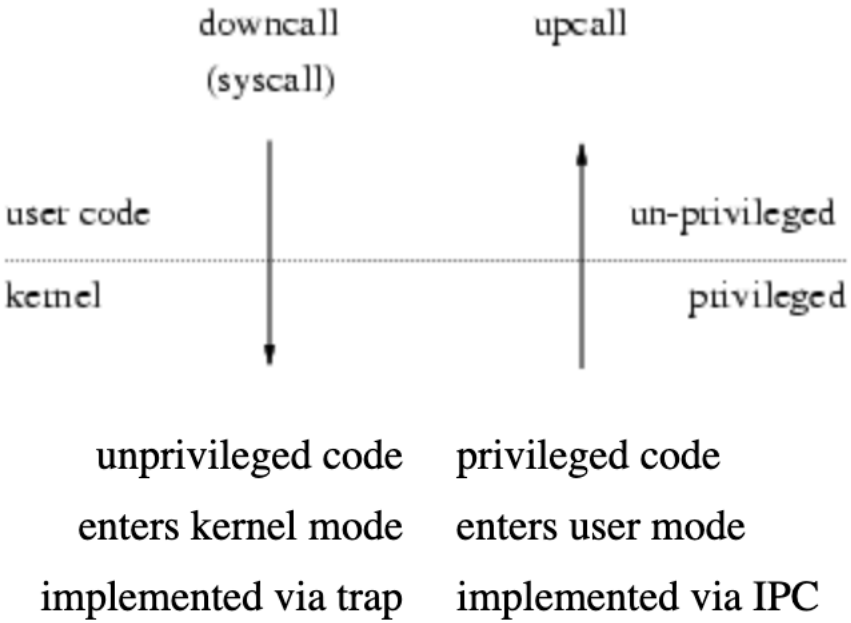
Kernel-to-User Mode Switch

- New process
- Resume after an interrupt/exception/syscall
- Switch to a different process
  - After a timer interrupt
- User-level upcall

Upcalls

- Allow apps to implement OS-like functionality to be invoked by OS

1. Asynchronous I/O notification
   - Wait for I/O completion
2. Inter-process communication
   - Debugger suspends a process
3. User-level exception handling
   - Ensures files are saved before app shuts
4. User-level resource allocation
   - Java garbage collection



Goals for Today

- User-kernel mode switch types
  - Exceptions, interrupts, and system calls
- **An x86 example of mode transfer**

x86 background

- Memory is segmented, so pointers come in two parts: a segment and an offset
  - Program counter: *cs* register and *eip* register
  - Stack pointer: *ss* register and *esp* register
  - CPL is stored as the 2 lower-bits of cs register
- In Intel 8086: cs:eip = cs * 16 + eip
  - Both cs and eip are 16-bits long, therefore CPU can access at most 2*20 (1MB) memory space
- *EFLGAS* register stores the processor status and controls its behavior
  - Whether interrupts are masked or not

x86 background

- Memory is segmented, so pointers come in two parts: a segment and an offset
  - Program counter: *cs* register and *eip* register
  - Stack pointer: *ss* register and *esp* register
  - CPL is stored as the 2 lower-bits of cs register

Why is it possible (2 bits wasted)?

- In Intel 8086: cs:eip = cs * 16 + eip
  - Both cs and eip are 16-bits long, therefore CPU can access at most 2*20 (1MB) memory space

- *EFLGAS* register stores the processor status and controls its behavior
  - Whether interrupts are masked or not

x86 background

- Only a small number of instructions can change the cs register value
  - ljmp (far jump)
  - lcall (far call), which pushes eip *and* cs to the stack, and then far jumps
  - lref (far return), which inverses the far call
  - INT: an assembly language instruction for x86 processors that generates a software interrupt. It takes the interrupt number formatted as a byte value, which reads cs / eip from the Interrupt Vector Table

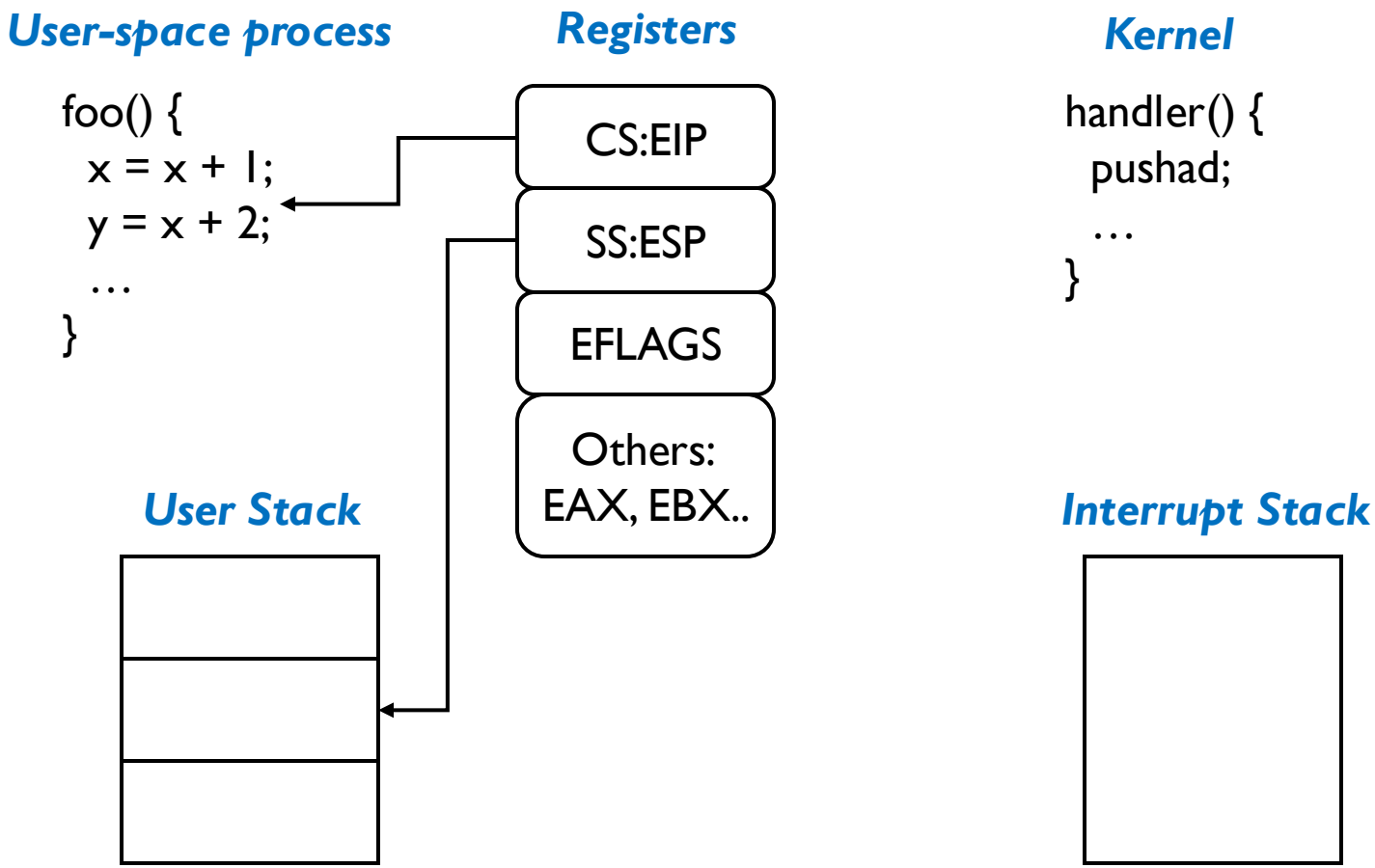
Format: INT X (syscall number)

  - IRET:returns program control from an exception or interrupt handler to a program or procedure that was interrupted previously
    - ❑ It basically reverses an INT

x86 Mode Transfer

- When an interrupt/exception/syscall occurs, the **hardware** will:

1. Mask interrupts
2. Save the special register values to other temporary registers
3. Switch onto the kernel interrupt stack
4. Push the three key values onto the new stack
5. Optionally save an error code
6. Invoke the interrupt handler

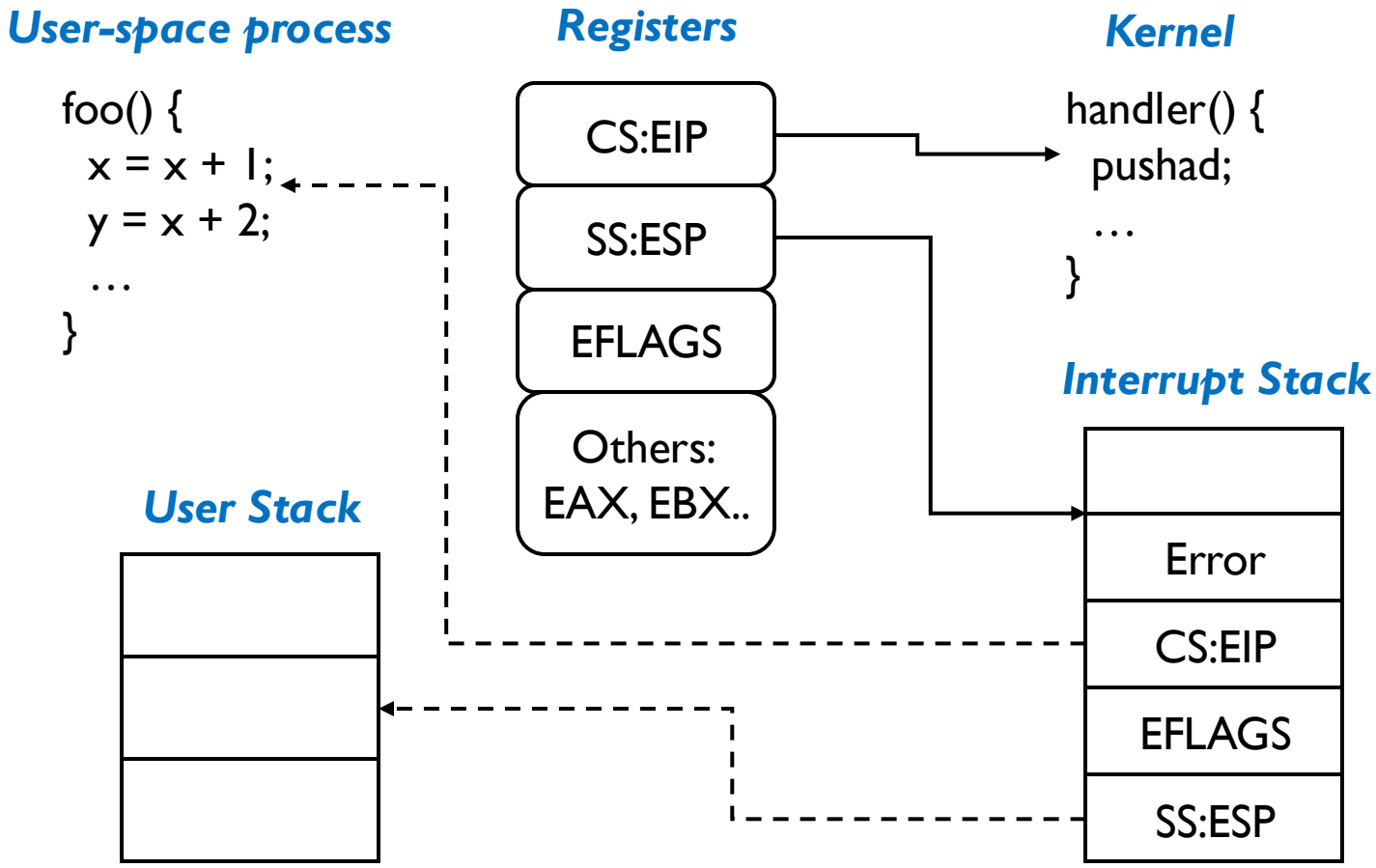


Before interrupt

x86 Mode Transfer

- When an interrupt/exception/syscall occurs, the **hardware** will:

1. Mask interrupts
2. Save the special register values to other temporary registers
3. Switch onto the kernel interrupt stack
4. Push the three key values onto the new stack
5. Optionally save an error code
6. Invoke the interrupt handler



At the beginning of handler

x86 Mode Transfer

- When an interrupt/exception/syscall occurs, the **hardware** will:

1. Mask interrupts
2. Save the special register values to other temporary registers
3. Switch onto the kernel interrupt stack
4. Push the three key values onto the new stack
5. Optionally save an error code
6. Invoke the interrupt handler

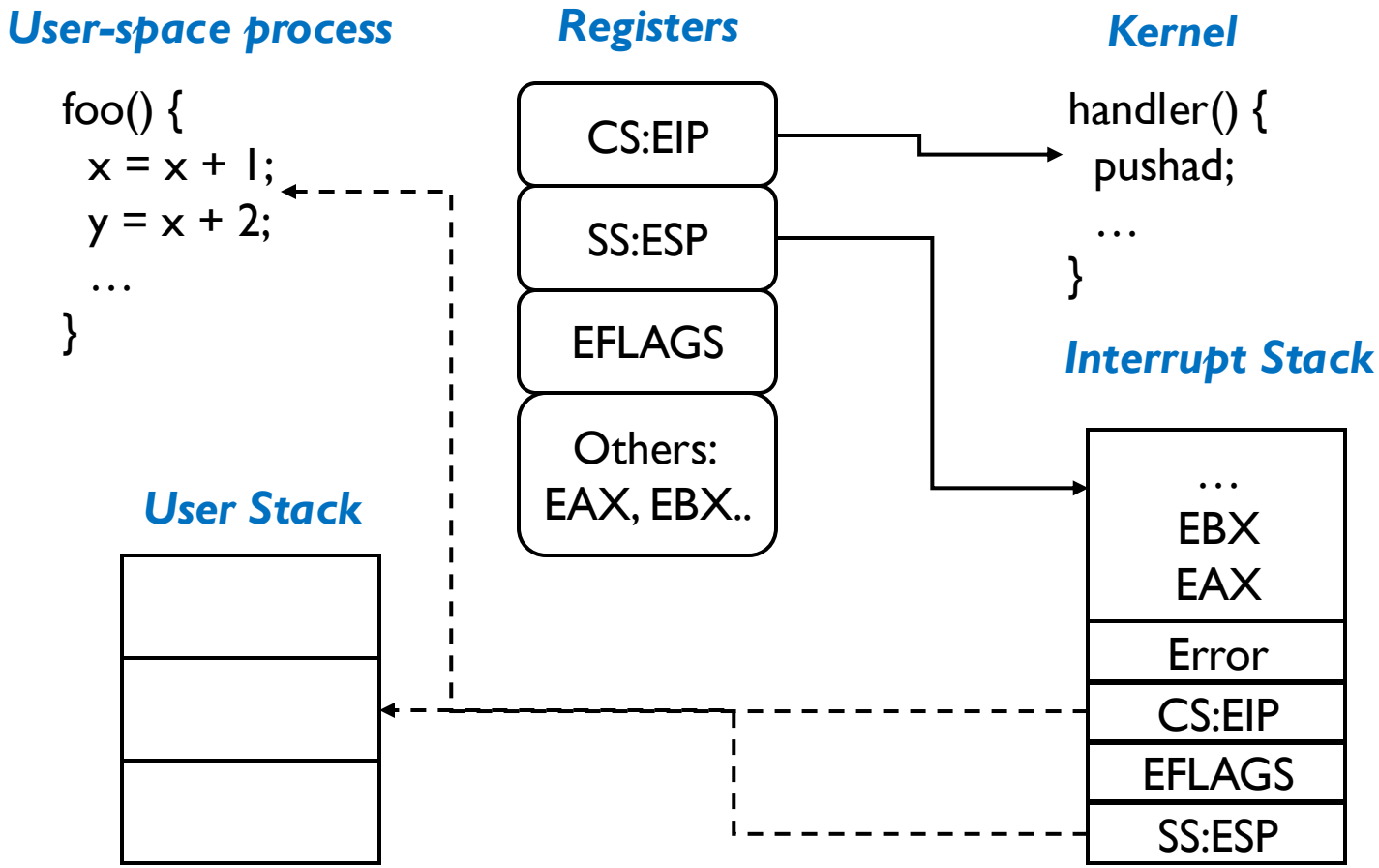
- *Steps 2-4 cannot be reversed. Why?*
- *Can they be done by software (OS)?*
- *Error codes*
  - *Page faults: which page?*
  - *Others: dummy values*

Before interrupt

x86 Mode Transfer

- When an interrupt/exception/syscall occurs, the **OS** will:

1. Save the rest of the interrupted process's state
   - *pusha/pushad*
2. Executes the handler
3. Resume the interrupted process
   - popa/popad + pop error code
4. Resume the interrupted process
   - *iret*



During interrupt handler

x86 Mode Transfer

- Who modifies the CPL (2 bits in CS)? Instructions like:
  - int/SYSCALL
  - iret

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>

int main() {
    const char msg[] = "Hello, Kernel!\n";
    syscall(SYS_write, STDOUT_FILENO, msg, sizeof(msg) - 1);
    return 0;
}
```

x86 Mode Transfer

- Who modifies the CPL (2 bits in CS)? Instructions like:
  - int/SYSCALL
  - iret

```c
#include <stdio.h>

int main() {
    /* Define message and file descriptor */
    const char msg[] = "Hello, Kernel!\n";
    int fd = 1;  // STDOUT

    /* Inline assembly to trap into kernel */
    __asm__ volatile (
        "movl $4, %%eax;"          /* syscall number for sys_write */
        "movl %0, %%ebx;"          /* file descriptor */
        "movl %1, %%ecx;"          /* pointer to message */
        "movl %2, %%edx;"          /* message length */
        "int $0x80;"               /* software interrupt */
        :                         /* no output */
        : "g"(fd), "g"(msg), "g"(sizeof(msg) - 1)
        : "eax", "ebx", "ecx", "edx"
    );

    return 0;
}
```

Stack vs. Heap

Why OS does not track the “heap pointer” as for stack?

Summary

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state

- Interrupts are safely designed
  - Interrupt vector: limited number of entry points into kernel
  - Interrupt stack: kernel handler/user states are decoupled
  - Interrupt masking: handler is non-blocking

- Again, there is hardware-software (OS) cooperation

Homework

- Read the interrupt handler code of xv6, and try to understand what is going on. Check it out on our website.