

# CS444/544

# Operating Systems II

## Handling Interrupt/Exceptions

Yeongjin Jang



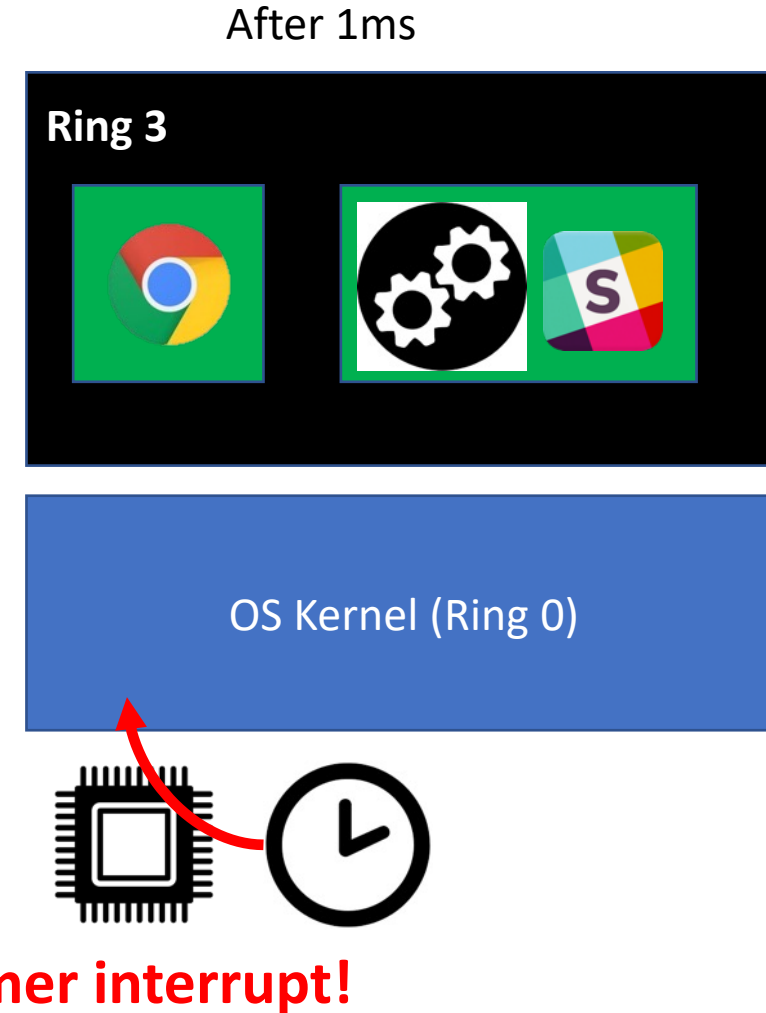
**Oregon State**  
**University**

# Lab 2 Grade

- Will be posted in this weekend
- I will run the grading script once a week (during the weekend)
- For Lab3, ignore part-A due date

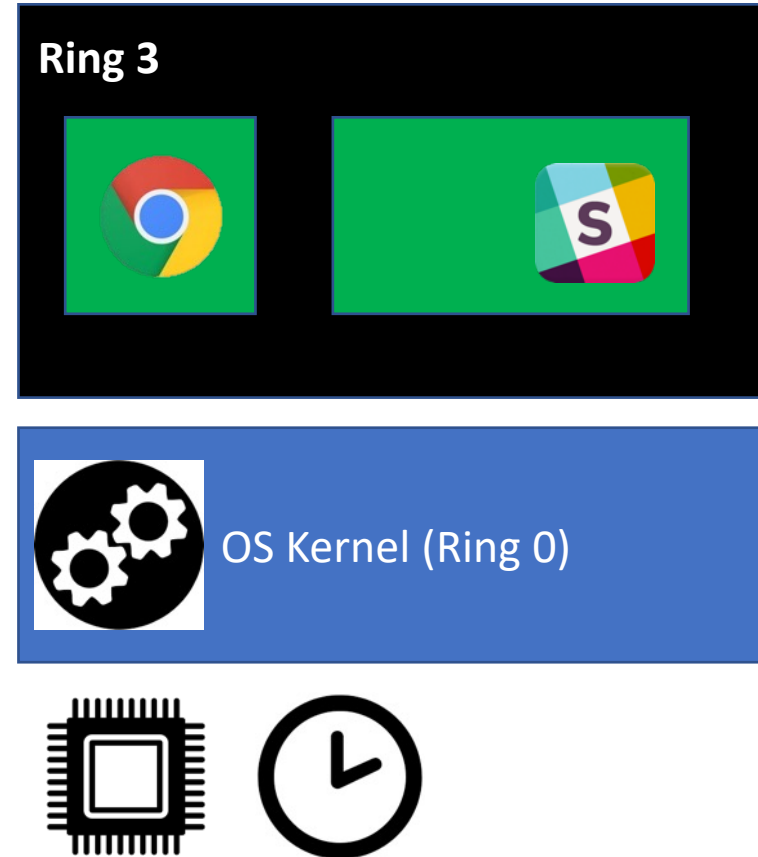
# Recap: Timer Interrupt and Multitasking

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..



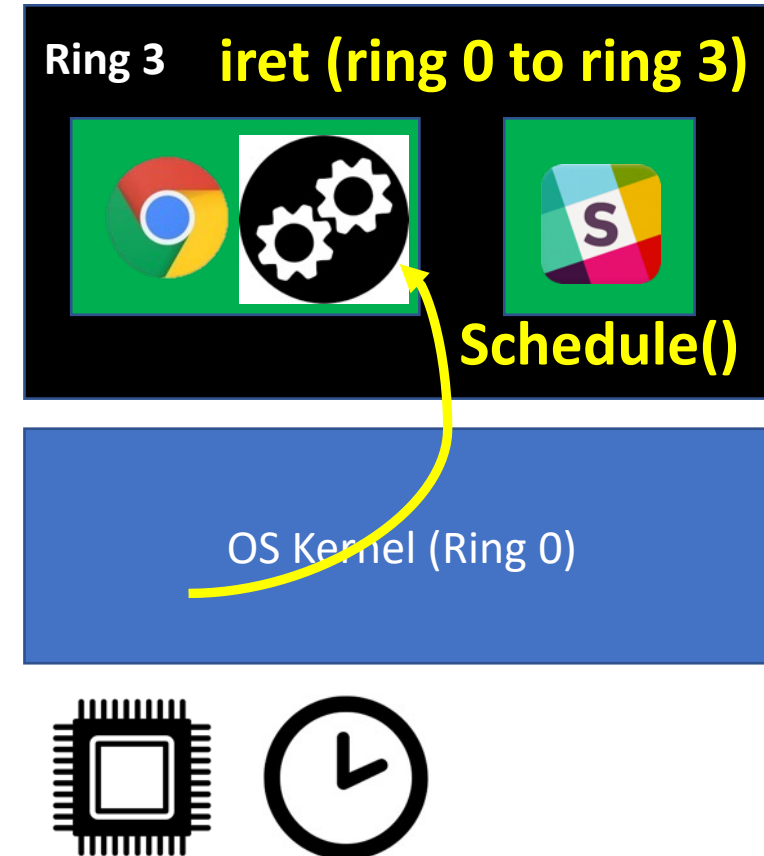
# Recap: Timer Interrupt and Multitasking

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



# Recap: Timer Interrupt and Multitasking

- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



# Recap: Interrupt

- Asynchronous (can happen at any time of execution)
- Mostly caused by external hardware
- Read
  - [https://en.wikipedia.org/wiki/Intel\\_8259](https://en.wikipedia.org/wiki/Intel_8259)
  - [https://en.wikipedia.org/wiki/Advanced\\_Programmable\\_Interrupt\\_Controller](https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller)
- Software interrupt
  - `int $0x30` ← system call in JOS

# Recap: Exceptions

- Synchronous (an execution of an instruction can generate this)
- Faults
  - Faulting instruction has not finished yet (e.g., page fault)
  - Can resume the execution after handling the fault
- Non-fault exceptions
  - The instruction (generated exception) has been executed (e.g., breakpoint)
  - Cannot resume the instruction (if so, it will trap indefinitely...)
- Some exceptions are fatal
  - Triple fault (halts the machine)

# Handling Interrupt/Exceptions

- Set 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 8

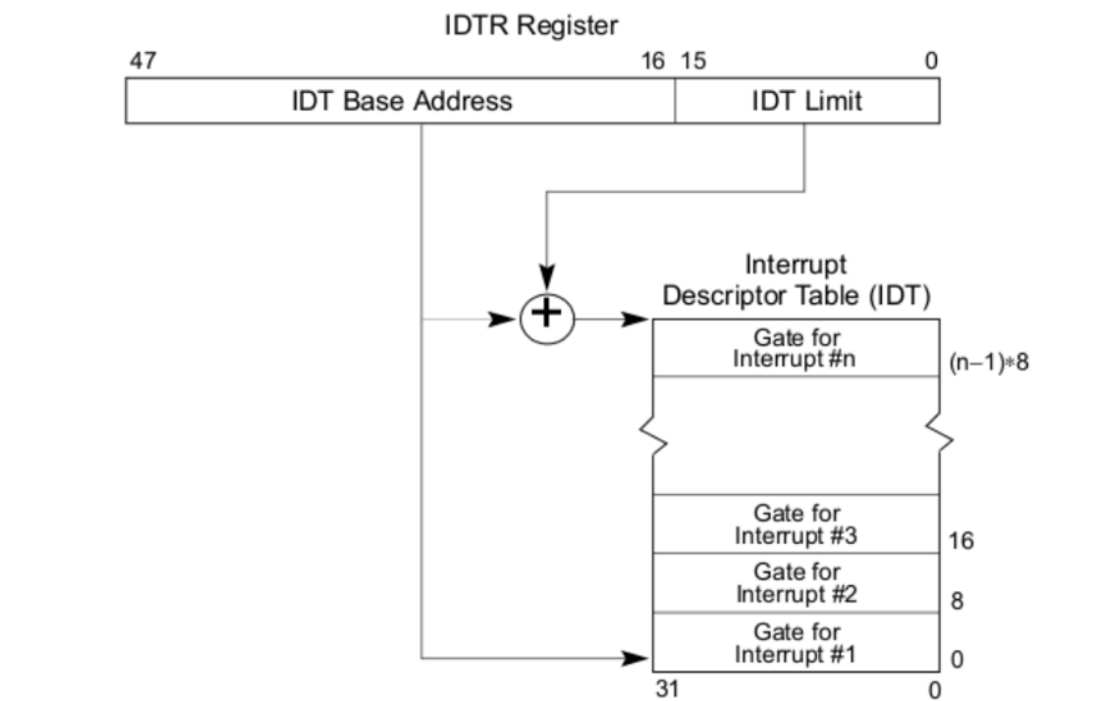


Figure 6-1. Relationship of the IDTR and IDT



# Handling Interrupt/Exceptions

- Set 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 9

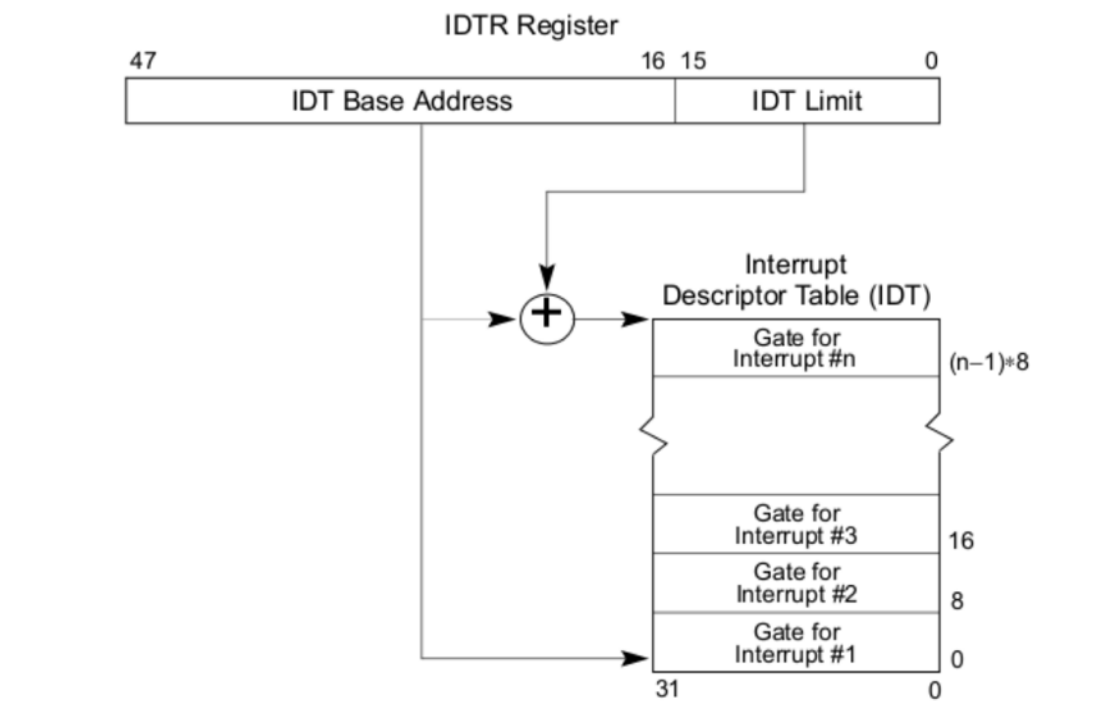


Figure 6-1. Relationship of the IDTR and IDT

# Handling Interrupt/Exceptions

- Set 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 10

```
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

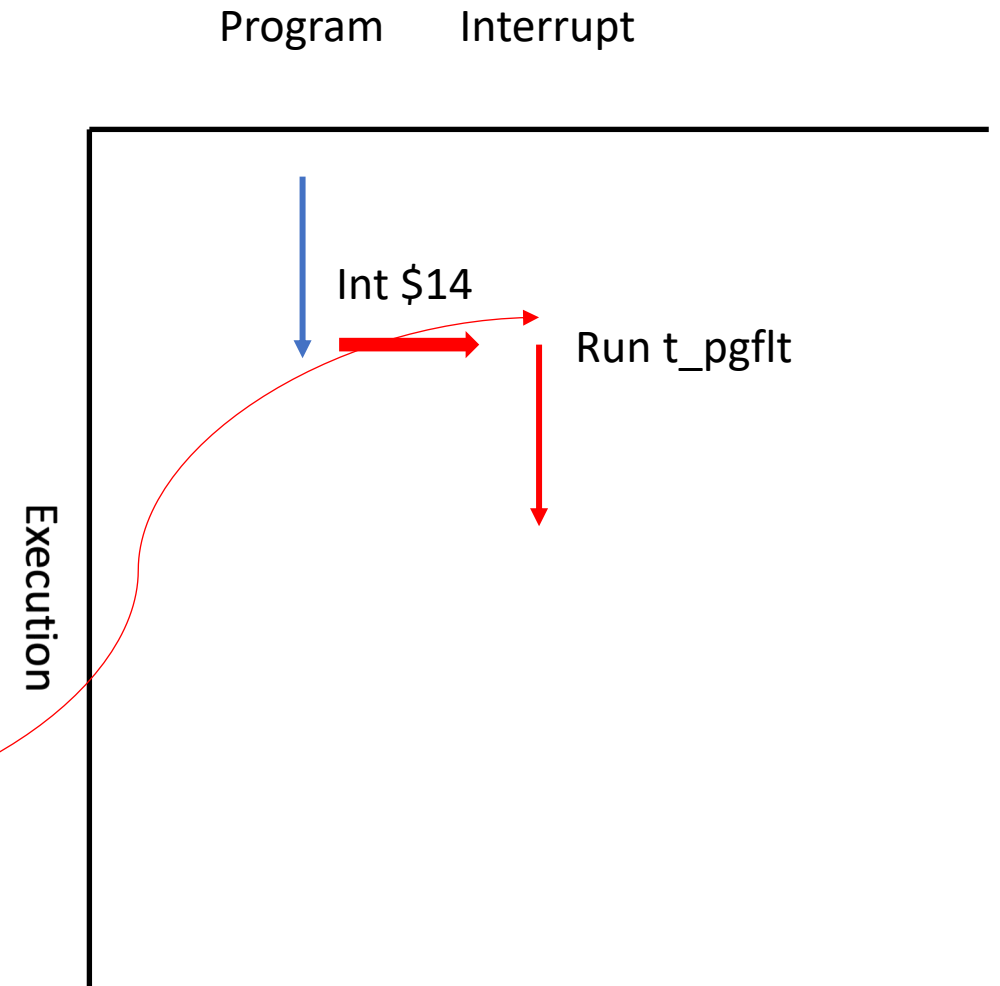
TRAPHANDLER_NOEC(t_mchk, T_MCHK); // 18
TRAPHANDLER_NOEC(t_simderr, T_SIMDERR); // 19

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

# Interrupt/Exception Handler

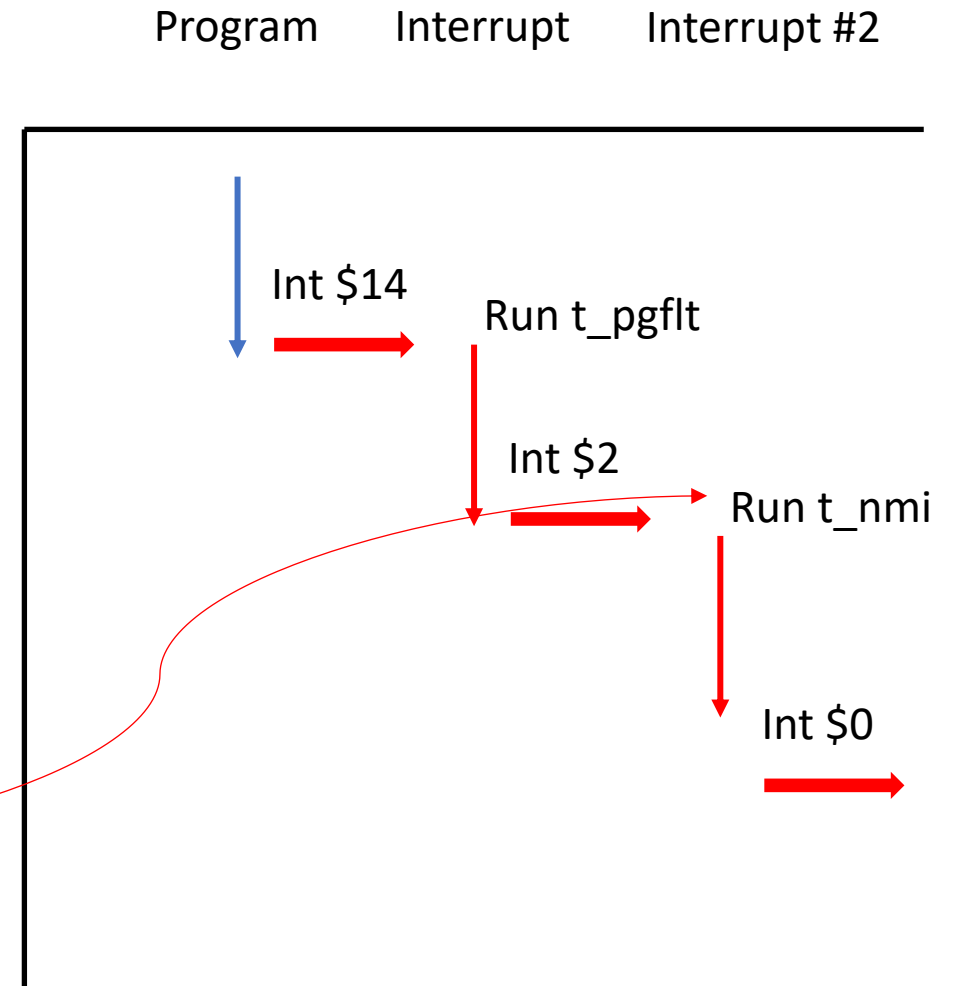
- Processing Interrupt/Exception

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 11



# Interrupt/Exception Handler

- What if another interrupt happens
  - During processing an interrupt?
- Handle interrupts indefinitely...
  - Cannot continue the program execution
  - Even cannot finish an interrupt handler...



Interrupt Number	Code address
0 (Divide error)	t_divide
1 (Debug)	t_debug
2 (NMI, Non-maskable Interrupt)	t_nmi
...	

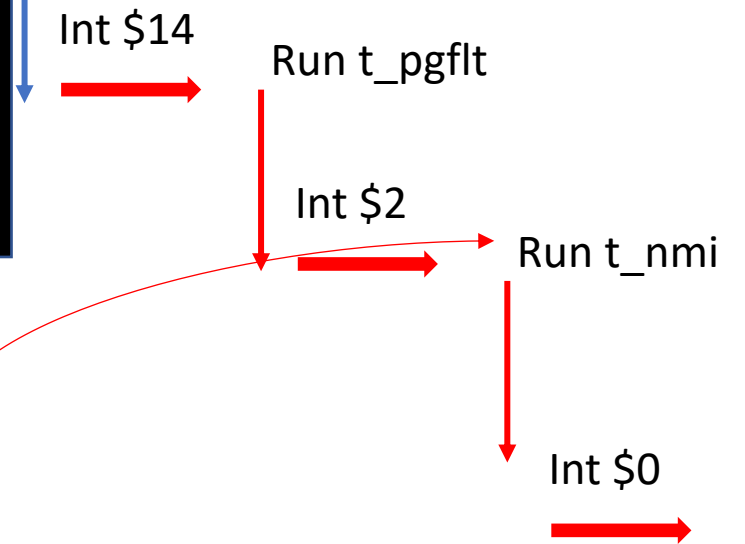
# Interrupt/Exception Handler

- What if another interrupt happens

**Interrupt request** coming during handling an interrupt request could make our interrupt handling **never finish!**

To avoid such an **'infinite' interrupt**,  
We **disable interrupt** while handling interrupt...

Program    Interrupt    Interrupt #2

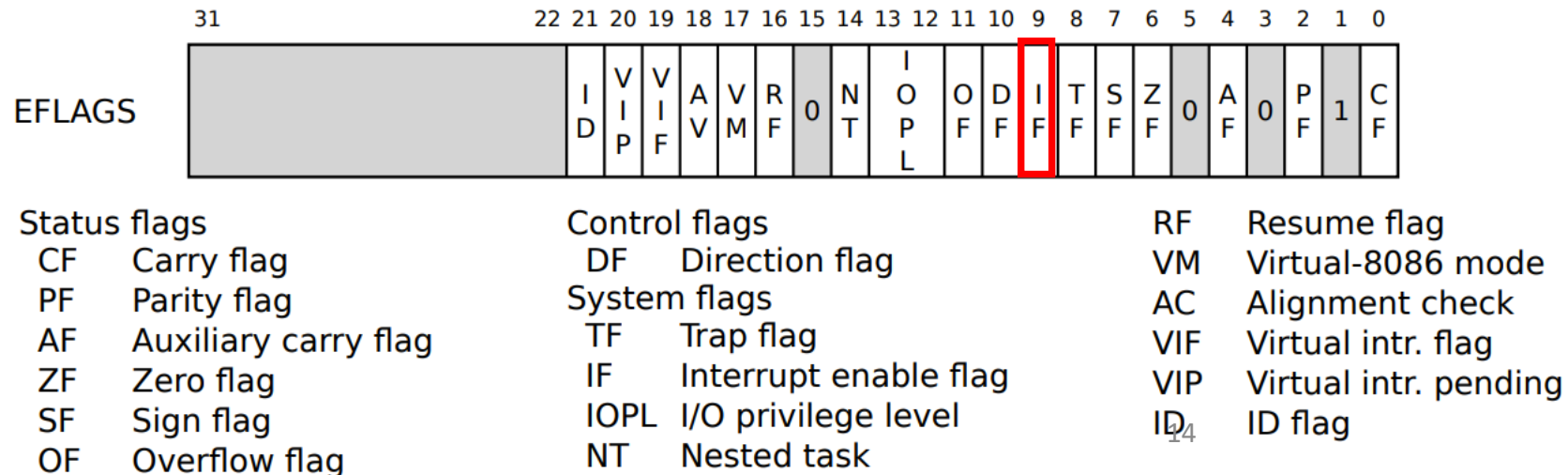


Interrupt Number	Code address
0 (Divide error)	t_divide
1 (Debug)	t_debug
2 (NMI, Non-maskable Interrupt)	t_nmi
...	

# Controlling Hardware Interrupt

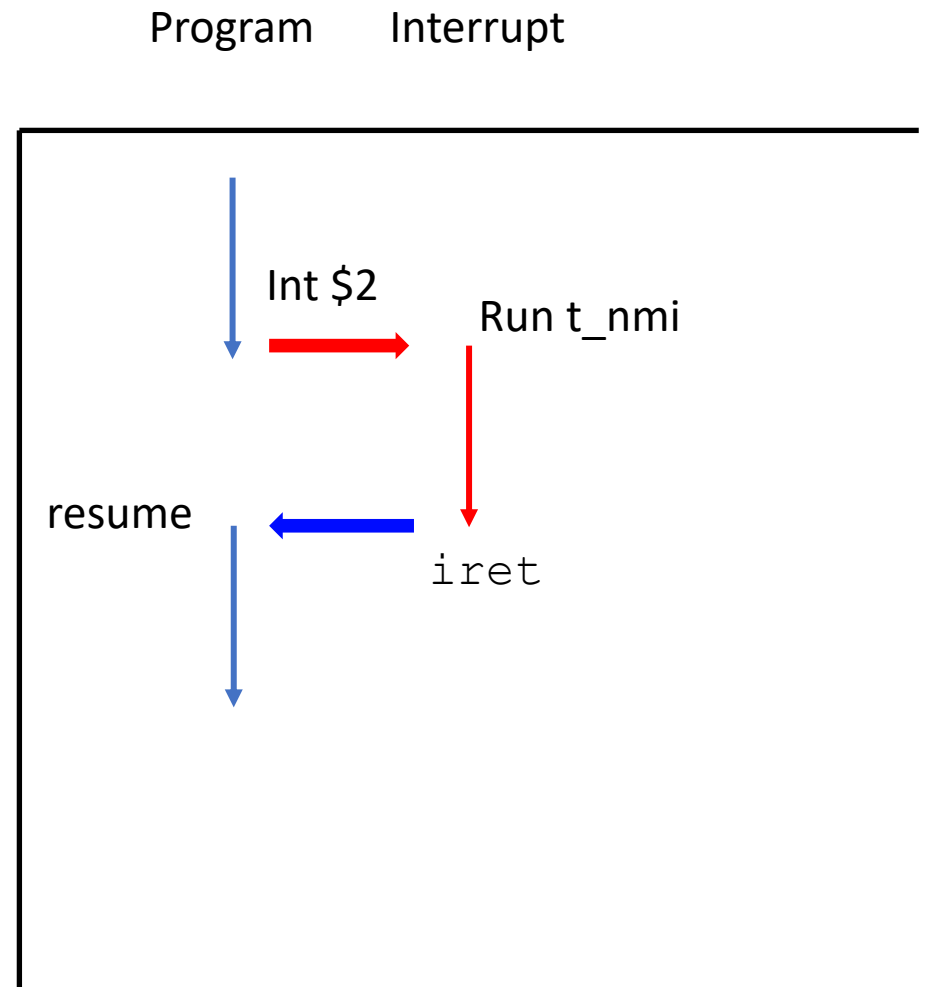
- Enabled/disabled by CPU
- IF flag in EFLAGS indicates this
  - sti (set interrupt flag, turn on)
  - cli (clear interrupt flag, turn off)

```
.globl start
start:
    .code16                # Assemble for 16-bit mode
    cli                   # Disable interrupts
```



# Interrupt/Exceptions Stop Current Execution

- We would like to handle the interrupt/exceptions at the kernel
- After handing that, we would like to go back to the previous execution
- How?
  - Store an execution context



# Storing an Execution Context

```
int global_value; // don't know the value

int main() {

    int i = 3;
    int j = 5;

    int sum = i;
    sum += global_value;

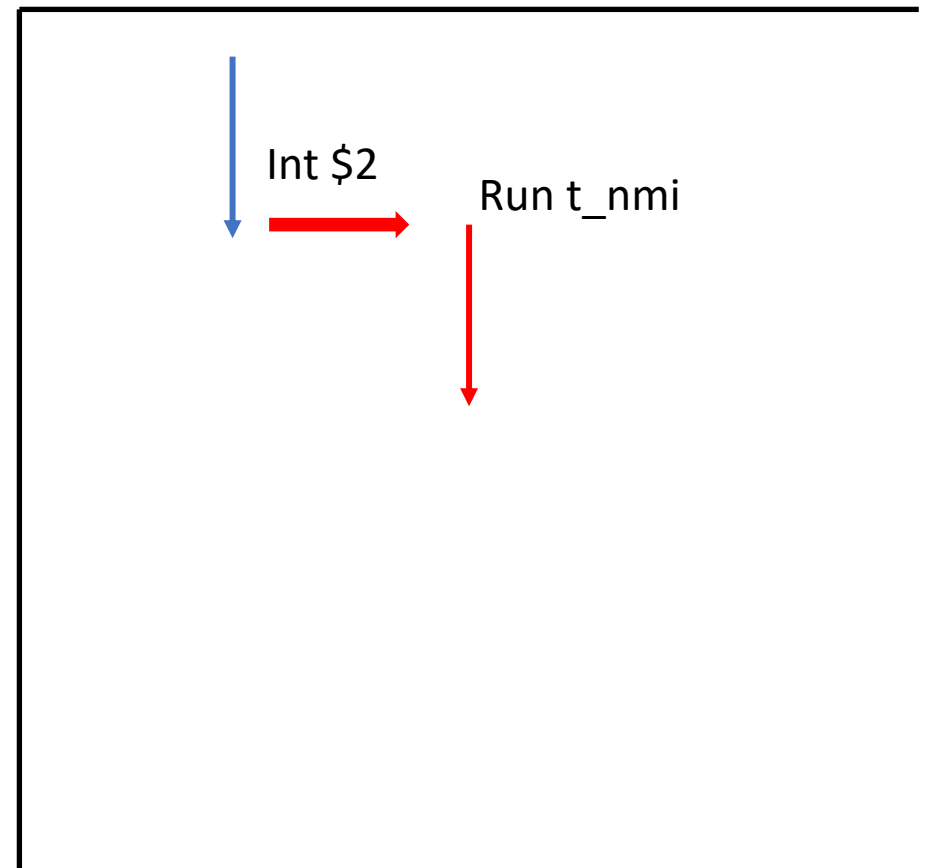
    sum += j;

    return 0;
}
```

Execute

Non-maskable  
Interrupt happened!

Program    Interrupt





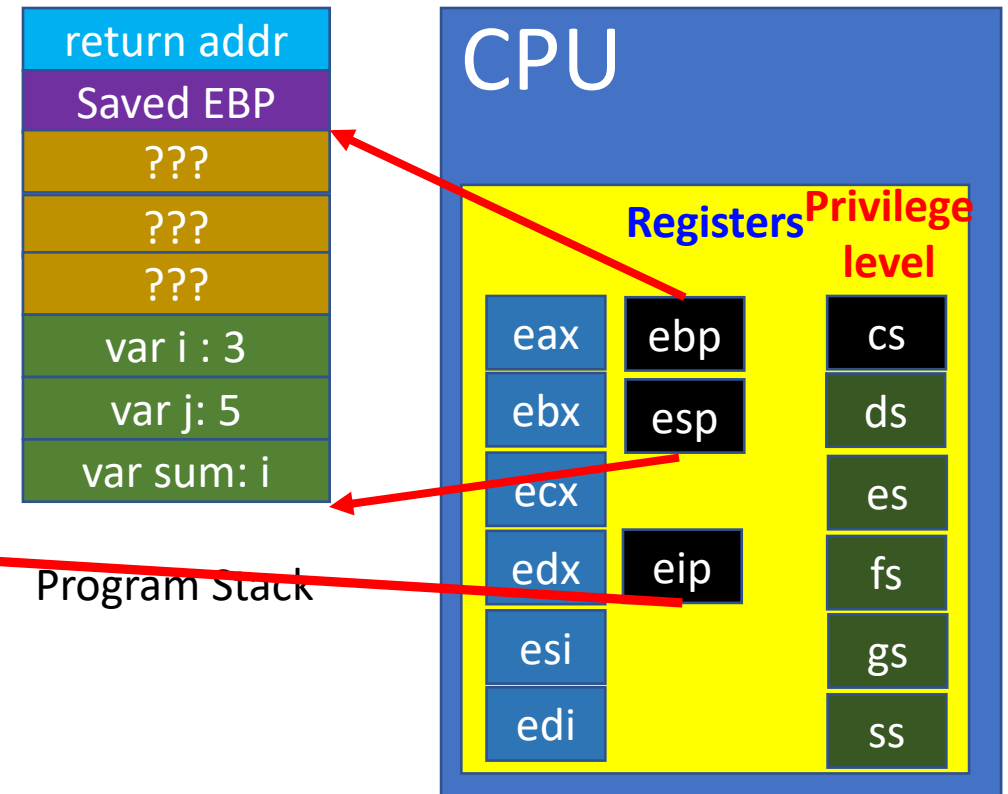
# How to Store an Execution Context?

```
int global_value; // don't know the value
int main() {
    int i = 3;
    int j = 5;

    int sum = i;

    sum += global_value;
    sum += j;

    return 0;
}
```

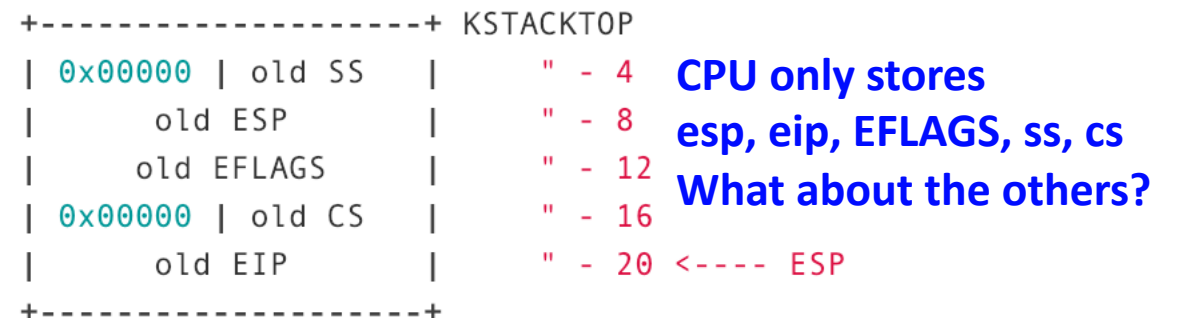


# Storing an Execution Context

- CPU uses registers and memory (stack) for maintaining an execution context

- Let's store them

- Stack (%ebp, %esp)
- Program counter (where our current execution is, %eip)
- All general purpose registers (%eax, %edx, %ecx, %ebx, %esi, %edi)
- EFLAGS
- CS register (why? CPL!)



# TrapFrame in JOS Stores the Context

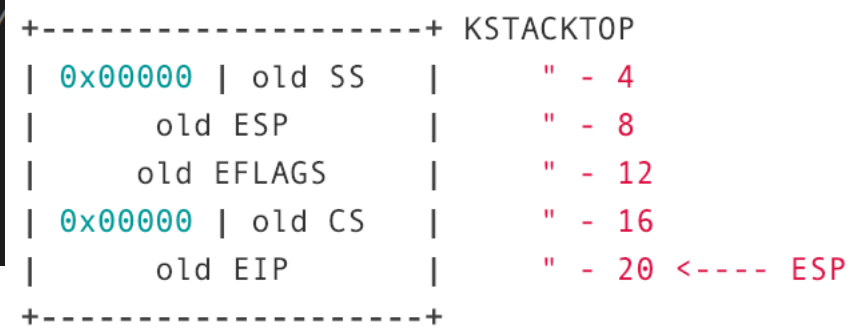
```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    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));
```

**2 byte padding because cs is 16-bit**

**2 byte padding because ss is 16-bit**

```
struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp; /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));
```

**JOS stores additional information as Struct Trapframe**



# How does JOS Handle Interrupt?

- You will setup an interrupt gate per each interrupt/exception

- Using MACROs defined in trapentry.S

- TRAPHANDLER(name, num)
- TRAPHANDLER\_NOEC(name, num)

- Gate generated by this macro should call

- trap() in kern/trap.c
- Implement \_alltraps:

```
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
```

```

#define TRAPHANDLER(name, num)
    .globl name; /* define global symbol for 'name' */
    .type name, @function; /* symbol type is function */
    .align 2; /* align function definition */
    name: /* function starts here */
    pushl $(num);
    jmp _alltraps

```

- Using MACROs defined in trapentry.S
  - TRAPHANDLER(name, num)
  - TRAPHANDLER\_NOEC(name, num)
- Gate generated by this macro should call
  - trap() in kern/trap.c
  - Implement \_alltraps:

```

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

```

# How Can You Know an Interrupt/Exception has EC/NOEC?

- Intel Manual

- <https://os.unexploitable.systems/r/ia32/IA32-3A.pdf> (page 186)

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode. <sup>1</sup>
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction. <sup>2</sup>
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.

# EC? NOEC? Error Code!

```
+-----+ KSTACKTOP
| 0x000000 | old SS | " - 4
|   old ESP   |   " - 8
|   old EFLAGS |   " - 12
| 0x000000 | old CS | " - 16
|   old EIP   |   " - 20 <----- ESP
+-----+
```

**Interrupt context (on the stack)**  
**When there is no error code**

```
+-----+ KSTACKTOP
| 0x000000 | old SS | " - 4
|   old ESP   |   " - 8
|   old EFLAGS |   " - 12
| 0x000000 | old CS | " - 16
|   old EIP   |   " - 20
|   error code |   " - 24 <----- ESP
+-----+
```

**Interrupt context (on the stack)**  
**When there is an error code**

# JOS Implementation

```
#define TRAPHANDLER(name, num)
    .globl name;          /* define global symbol for
    .type name, @function; /* symbol type is function
    .align 2;            /* align function definition */
    name:                /* function starts here */
    pushl $(num);
    jmp _alltraps
```

Pushes the interrupt number!

```
+-----+ KSTACKTOP
| 0x00000 | old SS | " - 4
|         | old ESP | " - 8
|         | old EFLAGS | " - 12
| 0x00000 | old CS | " - 16
|         | old EIP | " - 20
|         | error code | " - 24 <----- ESP
+-----+
```

Push the interrupt number!

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

Pushes the interrupt number!

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

Push 0 as a dummy error code

Push the interrupt number!



# How Can We Store a TrapFrame?

```
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));
```

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

**\_alltraps:**

**pushl %ds**

**pushl %es**

**pushal**

**You need to write  
more code than this!**

+-----+ KSTACKTOP		
0x00000	old SS	" - 4
	old ESP	" - 8
	old EFLAGS	" - 12
0x00000	old CS	" - 16
	old EIP	" - 20
	error code	" - 24 <----- ESP
+-----+		

**Push the interrupt number!**

# JOS Interrupt Handling

```
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));
```

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

```
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);

    fine 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    Build a Trapframe!
    pushl %es
    pushal
```

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

# In trap\_dispatch()

- All Interrupt/Exceptions comes to this function
  - Check trap number from tf->trapno
- Handle the following interrupts
  - T\_PGFLT (page fault, 14)
  - T\_BRKPT (breakpoint, 3)
  - T\_SYSCALL (system call, 48)

```
// Handle processor exceptions.
// LAB 3: Your code here.

switch (tf->tf_trapno) {
    case T_PGFLT:
    {
        // handle page fault here
    }
    case T_BRKPT:
    {
        // handle breakpoint here
    }
    case T_SYSCALL:
    {
        // handle system call here
    }
    default:
    {
    }
}
}
```

# System Call

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

# What Kind of System Call Do We Implement in Lab 3?

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

**Required for  
Implementing scanf, printf, etc...**



# How Can We Pass Arguments to System Calls?

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

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

Will add more as  
our lab implementation progresses

# How Can We Pass Arguments to System Calls?

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

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

# How Can We Pass Arguments to System Calls?

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

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



# How Can We Invoke a System Call?

- Set all arguments in the registers
  - Order: edx ecx ebx edi esi
- int \$0x30 (in JOS)
  - Software interrupt 48
- int \$0x80 (in 32bit Linux)
  - Software interrupt 128

# System Call Handling Routine (User)

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

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

# System Call Handling Routine (Kernel)

- CPU gets software interrupt
- TRAPHANDLER\_NOEC(T\_SYSCALL...)
- \_alltraps()
- trap()
- trap\_dispatch()
  - Get registers that store arguments from struct Trapframe \*tf
  - Call syscall() using those registers
    - This syscall() is at kern/syscall.c

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

# System Call Handling Routine (Return to User)

- Finishing handling of syscall (return of syscall())
- trap() calls env\_run()
  - Get back to the user environment!

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

- env\_pop\_tf()
  - Runs `iret`
- Back to Ring 3!

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

# Software Interrupt Handling (e.g., syscall)

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

Ring 3

Ring 0

Ring 3