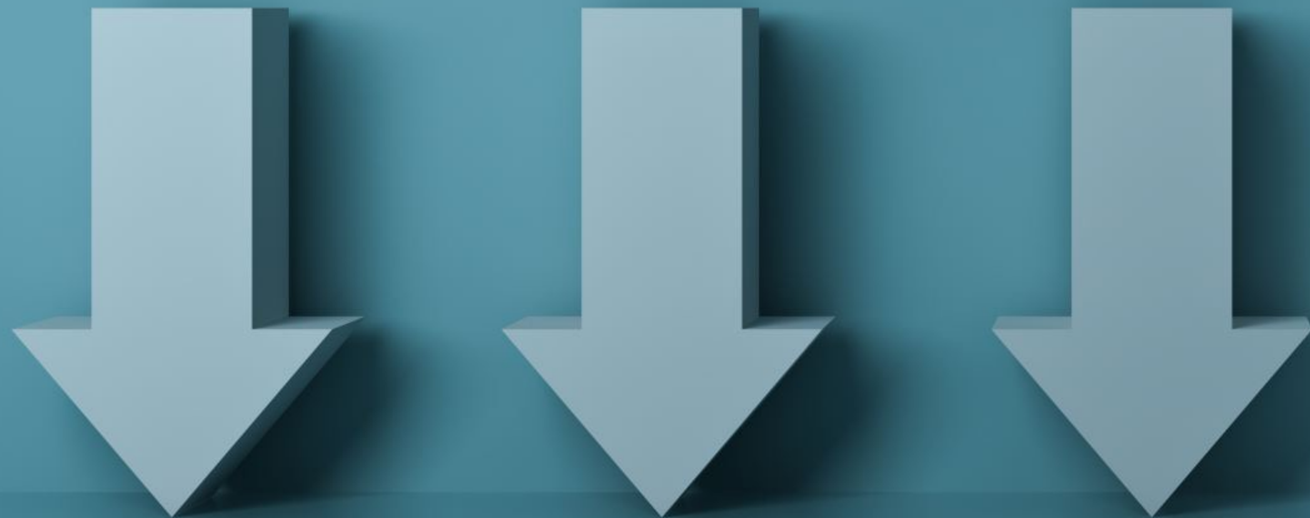
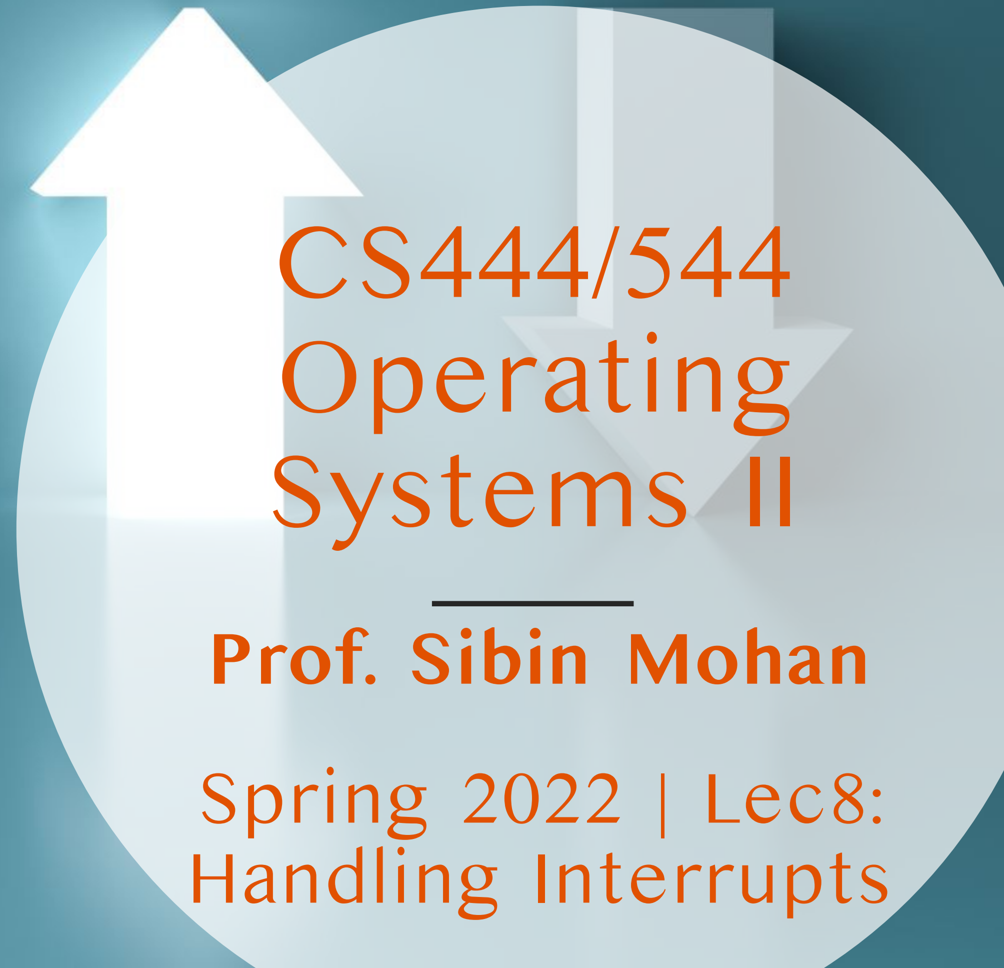



CS444/544 Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec8: Handling Interrupts

Adapted from content originally created by: Prof. Yeongjin Jang

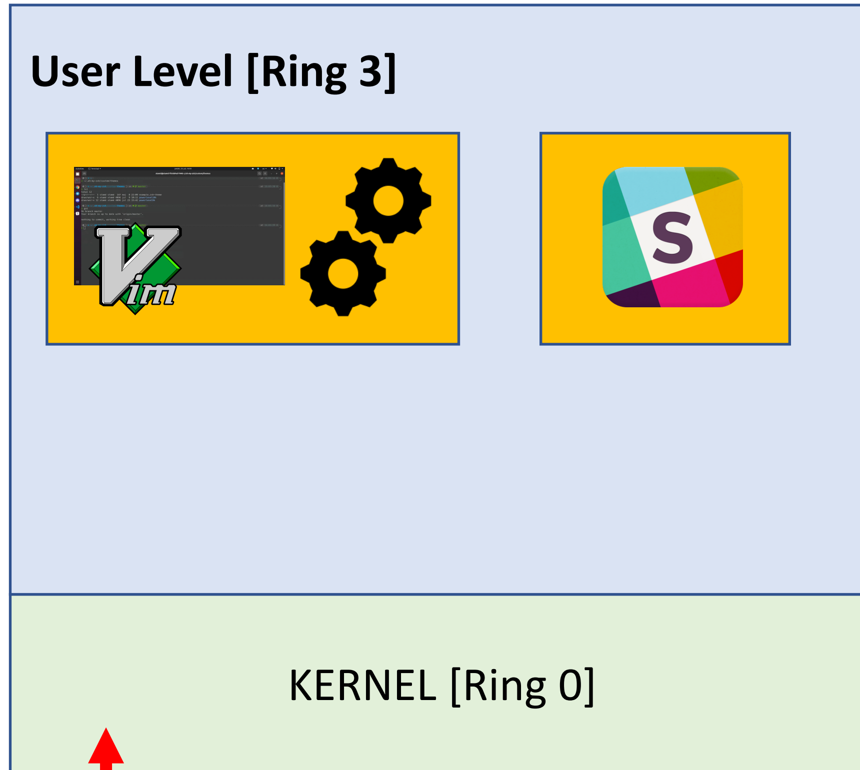
Grading

- Failures on some cases
  - I have provided detailed feedback and screenshots
  - **Make sure you fix all of it!**
  - Use **office hours** and **lab sessions**
- **Missing** repos → **0 pts**
- Deadline: **Wednesday, April 27, 11:59 PM**

Lab 2

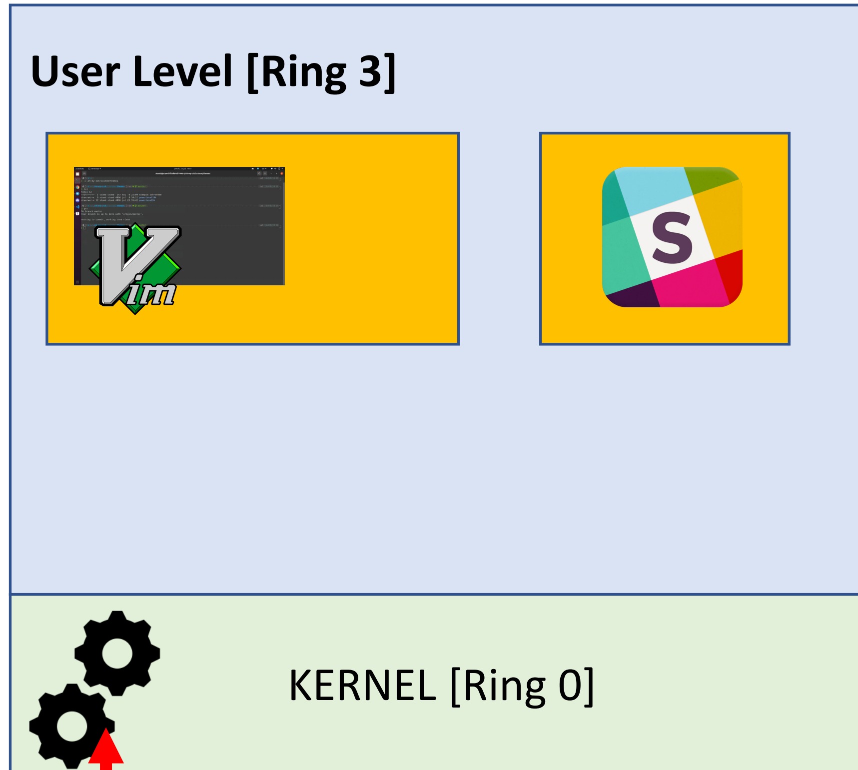
- Steeper learning curve
- Please work on it soon!

Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**

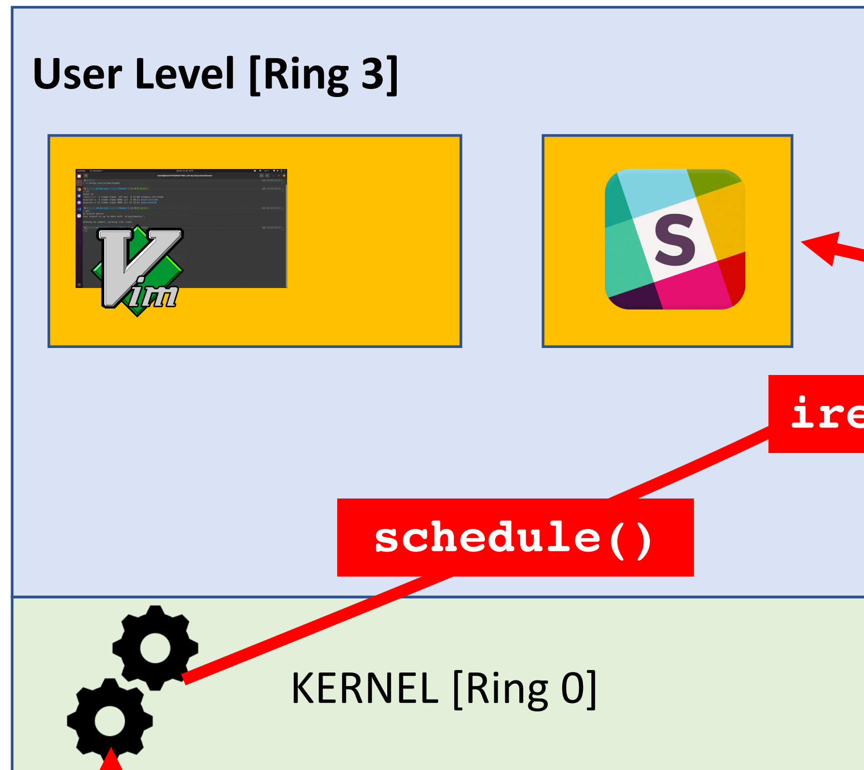
Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

Preemptive Multitasking | Timers



- A (hardware) clock timer
- CPU uses it to generate **periodic interrupts**
- Forces **kernel execution at regular intervals**
- E.g., every 1000 Hz [1 ms]

- Kernel then makes **scheduling decisions**
  - and mediates other resources

Hardware Interrupt

- Method for hardware to **interact** with CPU
- Example: a network device
  - NIC: *"Hey, CPU, I received a new packet, so wake up the OS to handle it"*
  - CPU: calls the **interrupt handler** for network device in ring 0 [set by the OS/driver]
- **Asynchronous** [can happen any time during execution]
  - It's a request from a hardware, so can happen any time
- Read
  - https://en.wikipedia.org/wiki/Intel_8259
  - https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller

Software Interrupt

- A piece of **software** mean to run code in ring 0 [e.g., `int $0x30`]
- Tells CPU, "*run the interrupt handler at 0x30*"

- **Synchronous** [caused by running an instruction, e.g., `int $0x30`]

- E.g.
  - System calls [int $0x30 → system call in JOS]
  - Signals in UNIX/Linux [SIGSEGV, SIGKILL, etc.]

Exceptions/Faults

- **Exceptions**
  - Error caused by the current execution [may or may not be recoverable]
  - Examples of non-recoverable exception [**cannot continue the execution**]
    - Triple fault
    - Divide by zero
    - Breakpoint
- **Fault**
  - An error caused by current execution that may be **recoverable** so **execution can continue**
  - Examples
    - Page fault
    - Double fault

- **Synchronous** [an execution of an instruction can generate this]
  - E.g., divide by 0

Handling Interrupt/Exceptions

- **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

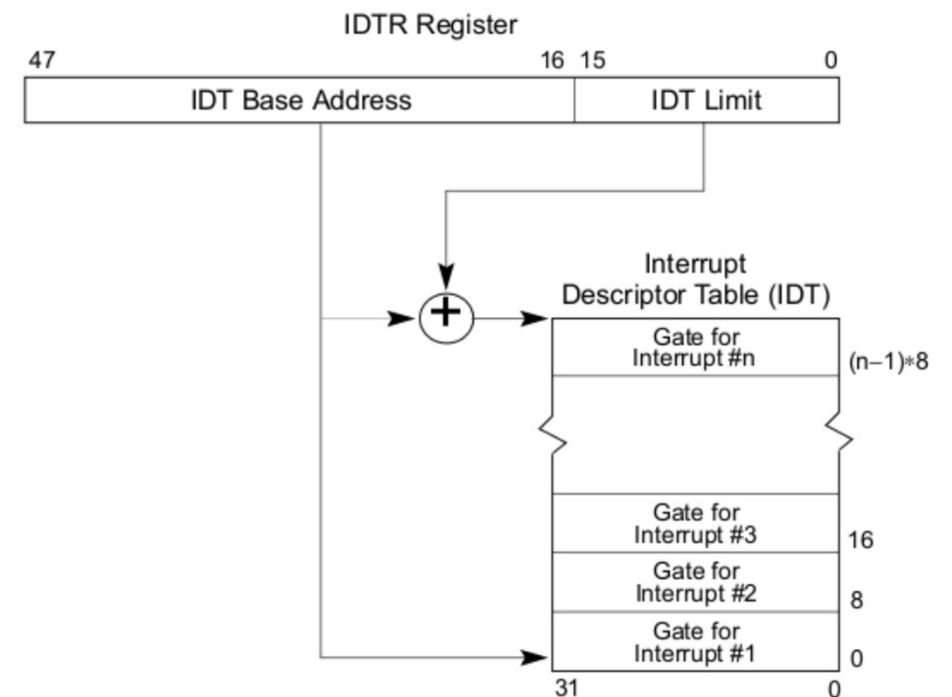


Figure 6-1. Relationship of the IDTR and IDT

Handling Interrupt/Exceptions

- **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**

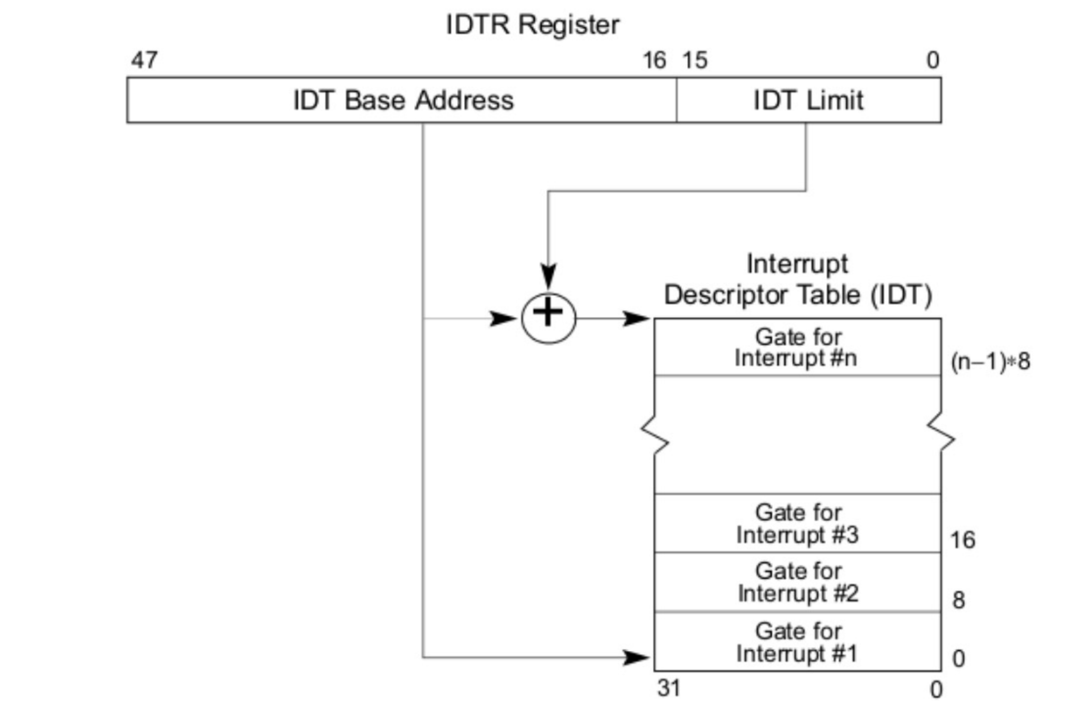


Figure 6-1. Relationship of the IDTR and IDT

Handling Interrupt/Exceptions

- **Interrupt Descriptor Table [IDT]**

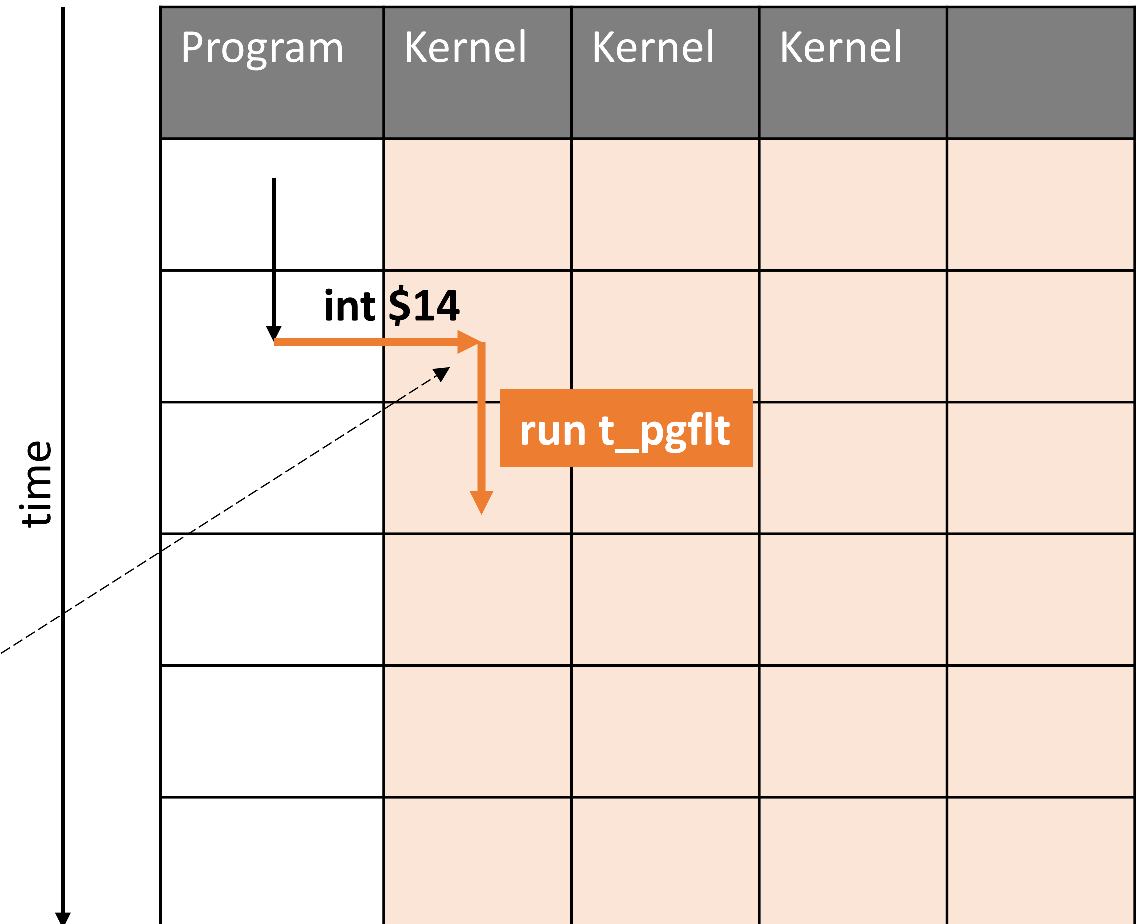
Interrupt Number	Code address
0 (Divide error)	**t_divide**
1 (Debug)	**t_debug**
2 (NMI, Non-maskable Interrupt)	**t_nmi**
3 (Breakpoint)	**t_brkpt**
4 (Overflow)	**t_oflow**
…	
8 (Double Fault)	**t_dblflt**
…	
14 (Page Fault)	**t_pgflt**
…	**…**
0x30 (syscall in JOS)	**t_syscall**

```
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
```

Handling Interrupt/Exceptions

- **Interrupt Descriptor Table [IDT]**

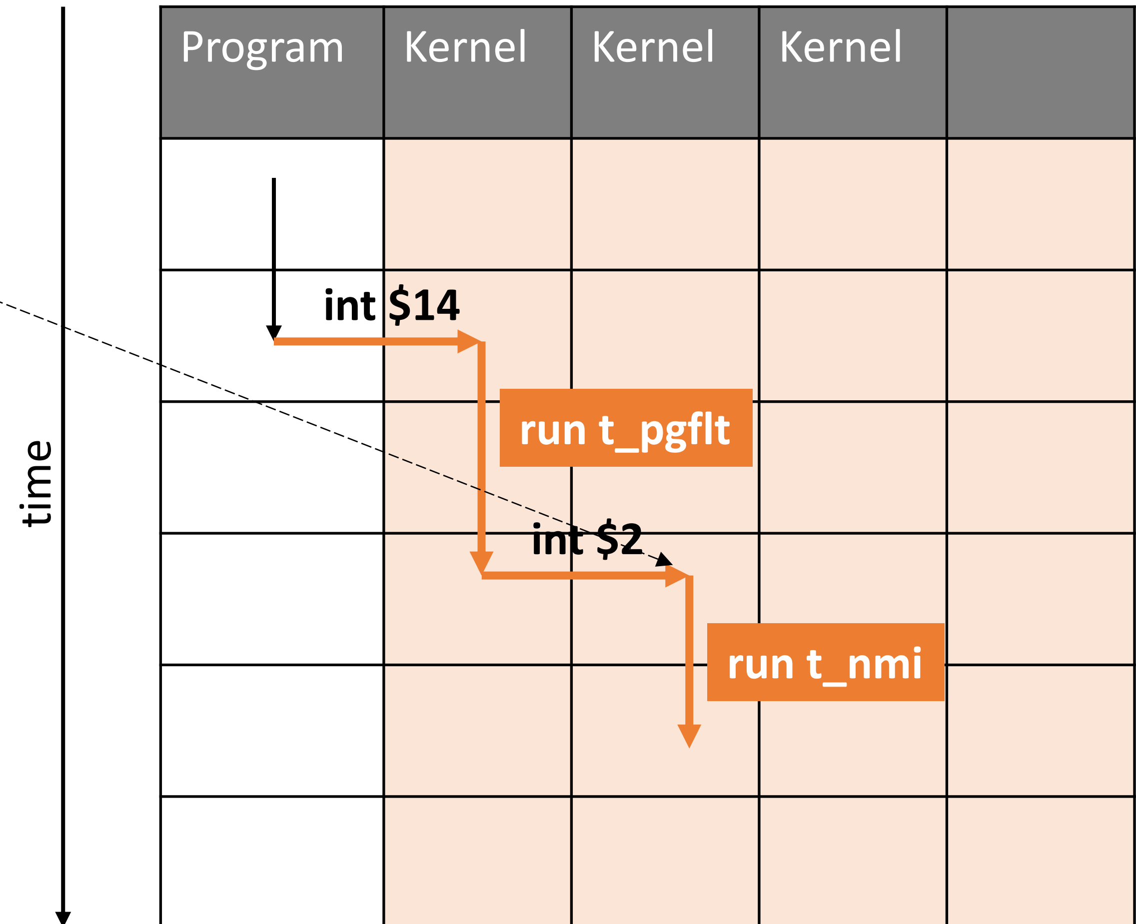
Interrupt Number	Code address
0 (Divide error)	**t_divide**
1 (Debug)	**t_debug**
2 (NMI, Non-maskable Interrupt)	**t_nmi**
3 (Breakpoint)	**t_brkpt**
4 (Overflow)	**t_oflow**
…	
8 (Double Fault)	**t_dblflt**
…	
14 (Page Fault)	**t_pgflt**
…	**…**
0x30 (syscall in JOS)	**t_syscall**



Handling Interrupt/Exceptions

- What if **another interrupt** is raised?
  - While one is being processed

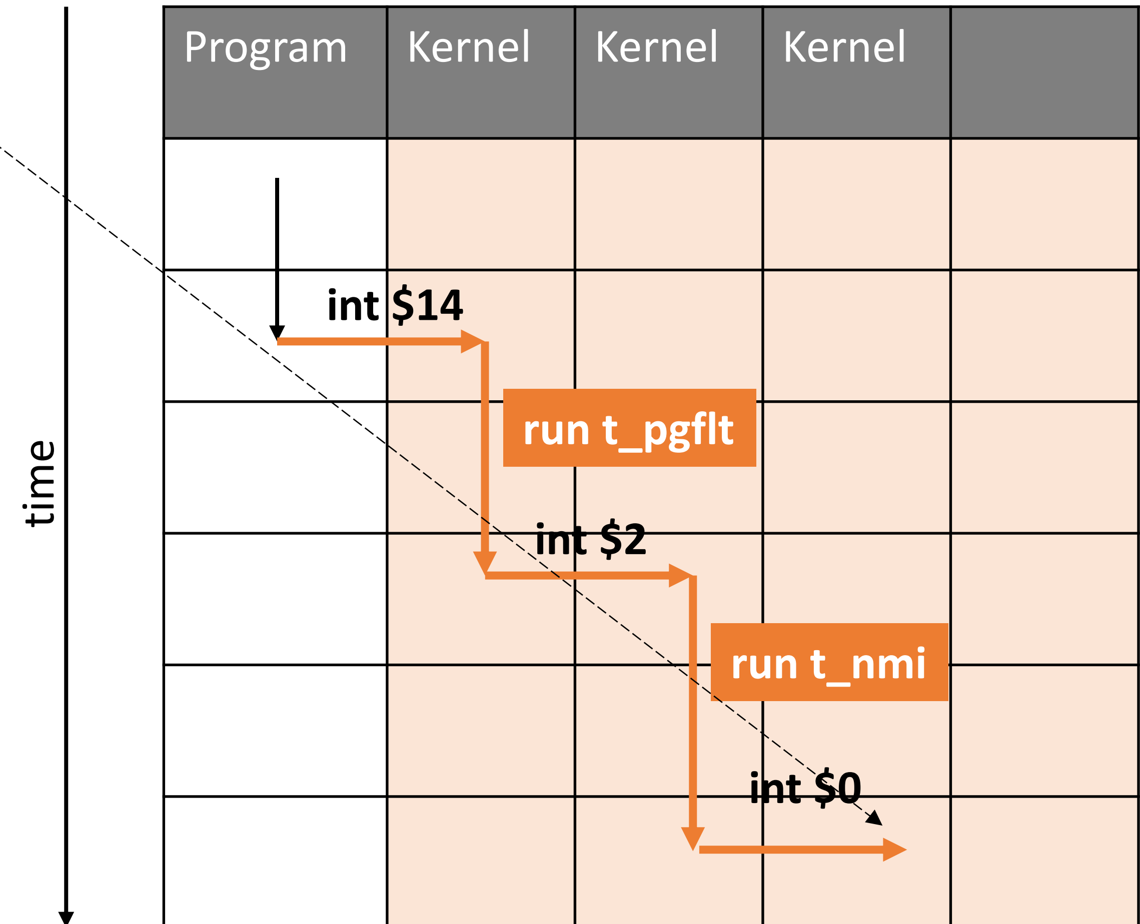
Interrupt Number	Code address
0 (Divide error)	**t_divide**
1 (Debug)	**t_debug**
2 (NMI, Non-maskable Interrupt)	**t_nmi**
3 (Breakpoint)	**t_brkpt**



Handling Interrupt/Exceptions

- What if **another interrupt** is raised?
  - While one is being processed

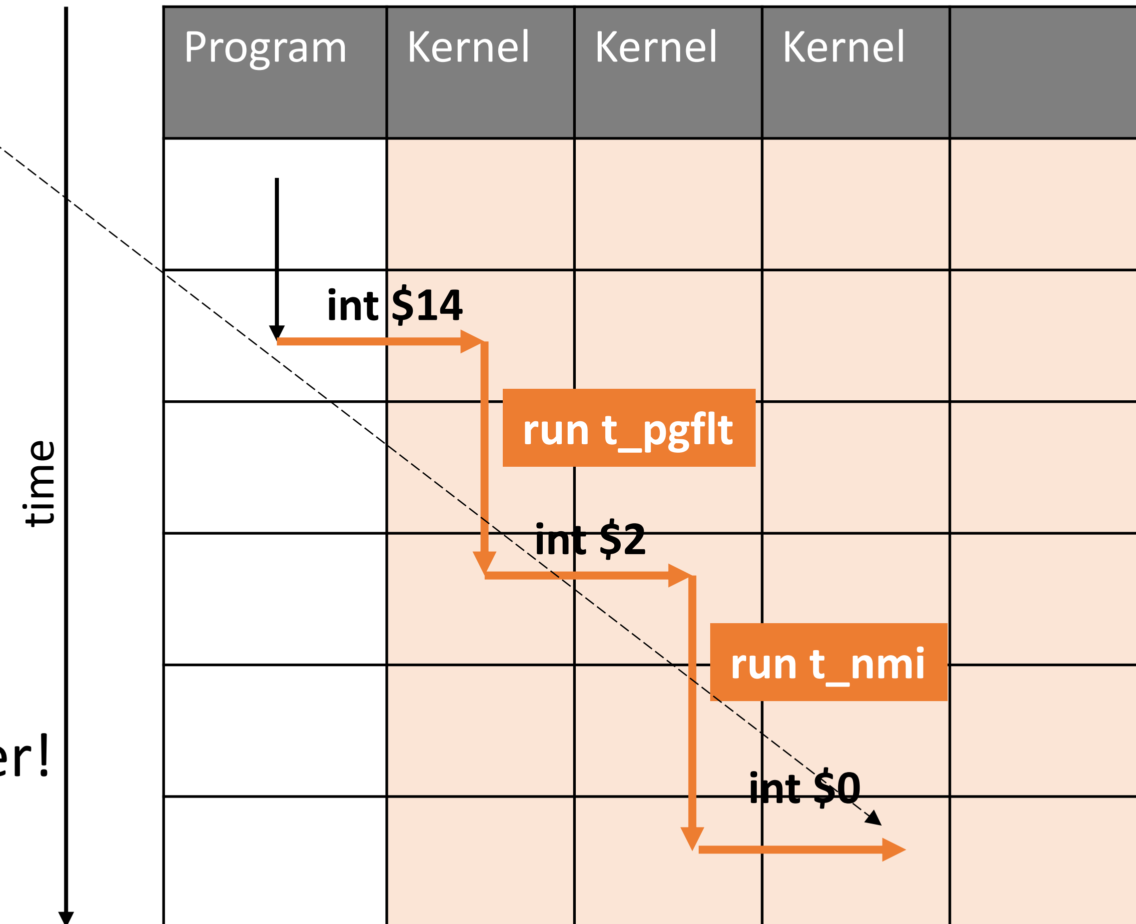
Interrupt Number	Code address
0 (Divide error)	**t_divide**
1 (Debug)	**t_debug**
2 (NMI, Non-maskable Interrupt)	**t_nmi**
3 (Breakpoint)	**t_brkpt**



Handling Interrupt/Exceptions

- What if **another interrupt** is raised?
  - While one is being processed

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



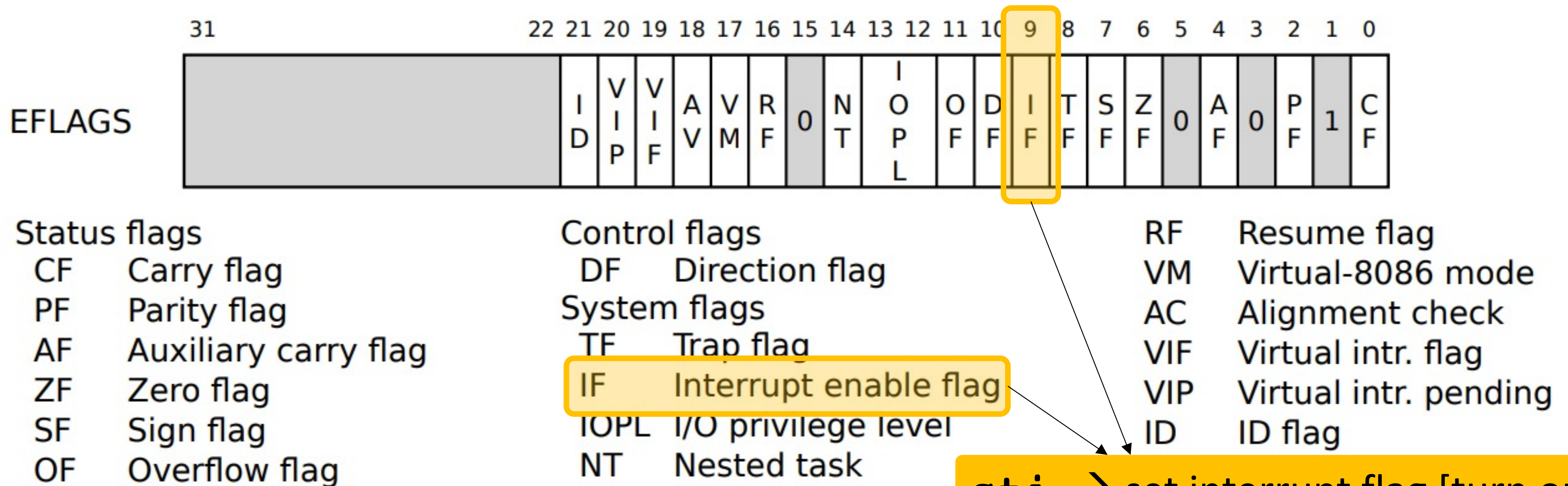
- **Indefinite handling** of interrupts
  - Can't finish program execution
  - Cannot even finish one interrupt handler!

What's the Solution?

- **Disable interrupts** of course!
- While handling other interrupts

Control Hardware Interrupt

- Enabled/disabled by CPU
- Flag in **EFLAGS** regsiter



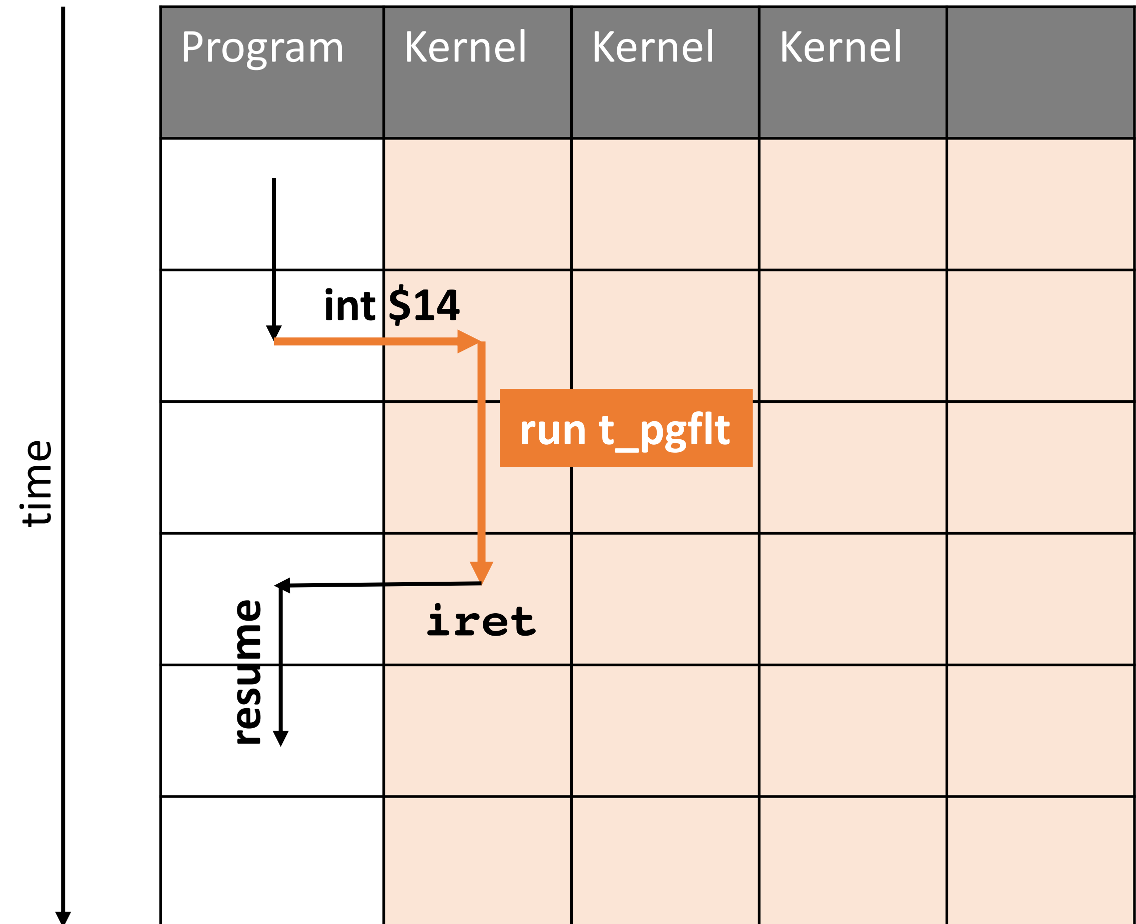
Jos Startup

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

Handling Interrupt/Exceptions

- Interrupts handled in kernel space
- **Return to user space** once done

- **Store user context**



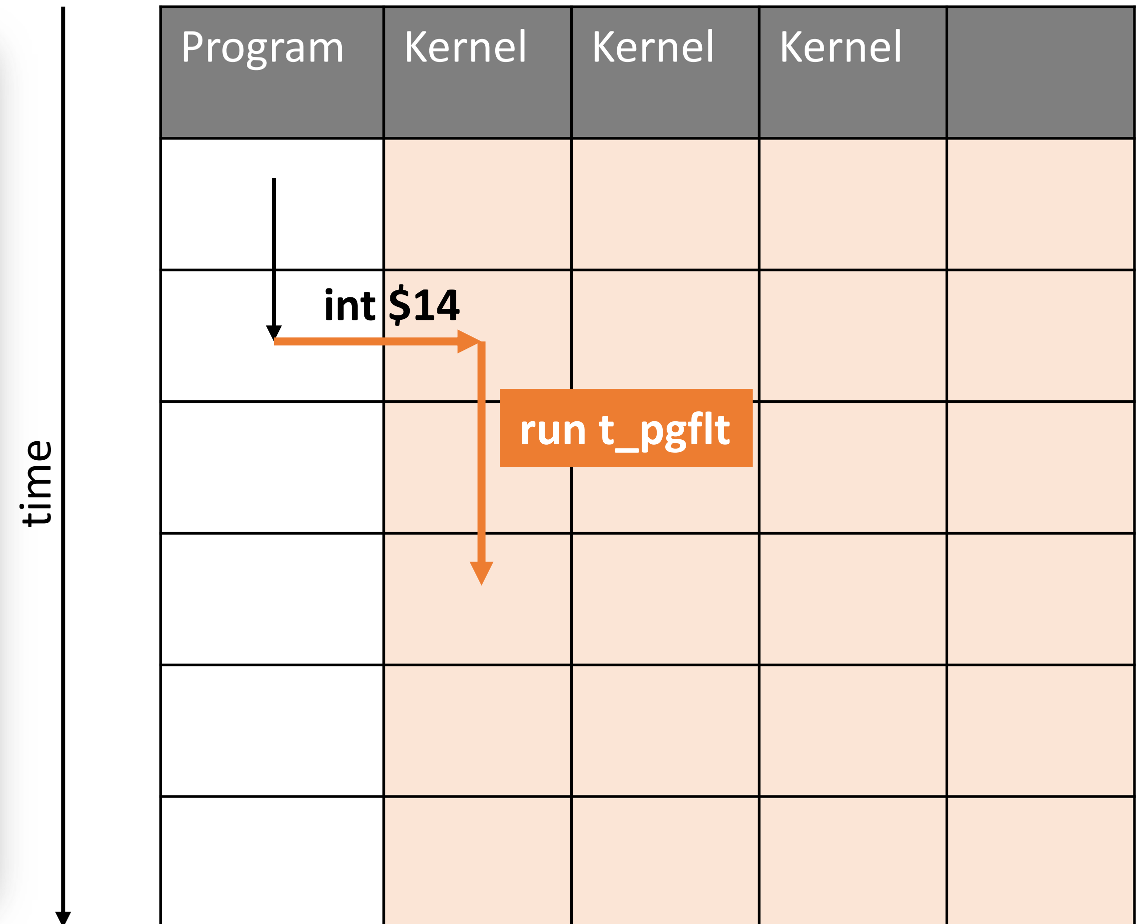
Storing Execution Contexts

```
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

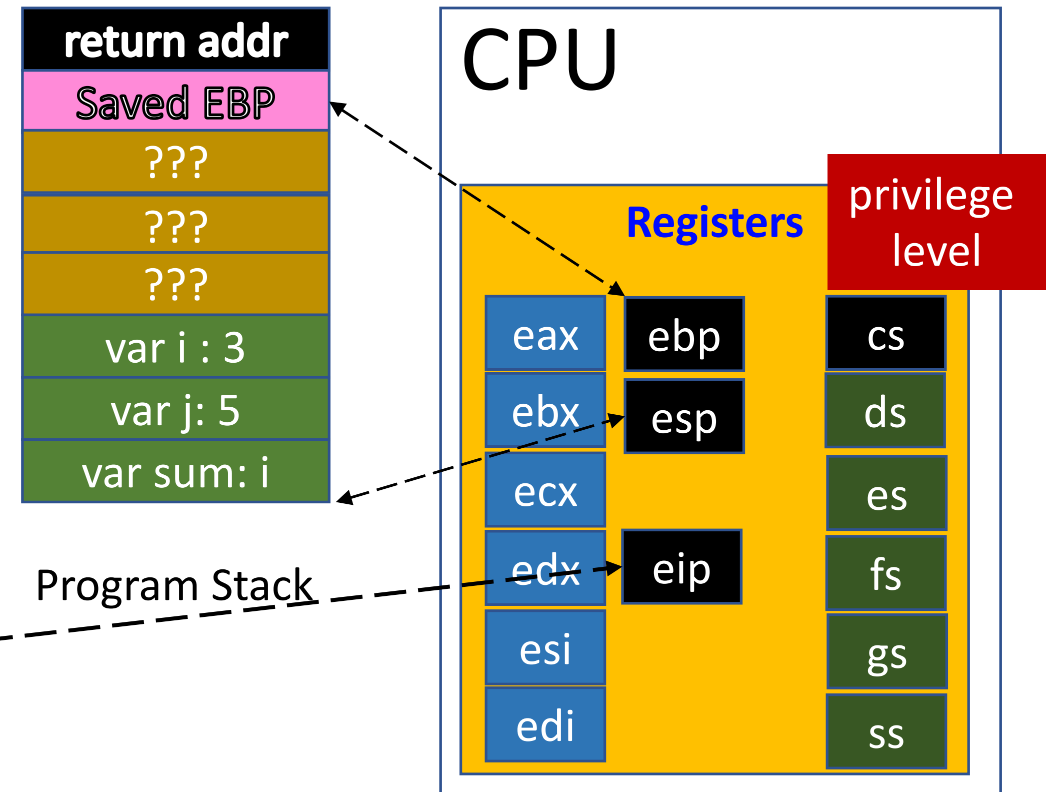
access global variable
PAGE FAULT!

time



Storing Execution Contexts

```
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
- It needs to store:
  - **Stack** (%ebp, %esp)
  - **Program counter** (where our current execution is, %eip)
  - All **general-purpose registers** (%eax, %edx, %ecx, %ebx, %esi, %edi)
  - **EFLAGS**
  - CS register **CPL!**

Trapframe!

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

CPU only stores **`esp, eip, EFLAGS, ss, cs`**
What about the others?

JOS 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));
```

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

```
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));
```

How does JOS Handle Interrupts?

- Set up an **interrupt gate**
- For each interrupt/exception

- Store in **Interrupt Descriptor Table** [IDT]

```
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 does JOS Handle Interrupts?

- 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
```

TRAPHANDLER

- `TRAPHANDLER(name, num)`
- `TRAPHANDLER_NOEC(name, num)`

```
#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
```

Interrupt/Exception with/without 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.[1]
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.[2]
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
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+
```

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

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

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

JOS Implementation

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

Push 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
+--------------------+
```

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

Push 0 as a dummy error code

Push the interrupt number!

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

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

- Setup the IDT at **`trap_init()`** in kern/trap.c
- Interrupt arrives at 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!

JOS Interrupt Handling [contd.]

- Call **`trap()`** in kern/trap.c

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

- Call **`trap_dispatch()`** in kern/trap.c
  - All Interrupt/Exceptions comes to this function
  - Check trap number from `tf->trapno`

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

trap_dispatch()

- Lab 3: 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 Calls

System Call

User-level Application calls functions in kernel

Open	Read	Write	Exec	Send	Recv	Socket	etc.

System Calls 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.

Pass Arguments to System Calls?

- In JOS
  - eax = system call number
  - edx = 1st argument
  - ecx = 2nd argument
  - ebx = 3rd argument
  - edi = 4th argument
  - esi = 5th 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_getenvid,
	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`**
- Interrupt handler
  - **Read syscall number from the eax of traoframe [tf]**
    - **syscall number is 0 -> calling SYS_cputs**
  - **Read 1st argument from the edx of tf**
    - **Address of "adsf"**
  - **Read 2nd argument from ecx of tf**
    - **4**
  - **call sys_cputs("asdf", 4) // in kernel**

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

```
// 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:
    {
```

How Can We Pass Arguments to System Calls?

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

0	sys_restart_syscall	0x00
1	sys_exit	0x01
2	sys_fork	0x02
3	sys_read	0x03
4	sys_write	0x04
5	sys_open	0x05
6	sys_close	0x06
7	sys_waitpid	0x07
8	sys_creat	0x08
9	sys_link	0x09
10	sys_unlink	0x0a
11	sys_execve	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 (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!
- `env_pop_tf()`
  - Runs **`iret`**
- **Back to Ring 3!**

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

```
void
env_pop_tf(struct Trapframe *tf)
{
	asm volatile(
		"\tmovl %0,%%esp\n"
		"\tpopal\n"
		"\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 */
}
```

Restore the CPU state from the trap frame

Software Interrupt Handling (e.g., syscall)

- Execution
- `int $0x30`

Ring 3

- Call trap gate
- Handle trap!
- Pop context

Ring 0

- `iret`
- Execution resumes

Ring 3