Reminders

- Lab 3 posted
  - Check tutorials + slides

- Suggestion: Complete Part A by Tuesday’s lecture!!! 😊
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/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)

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>0xf0130304</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>0xf0153333</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>0xf0183273</td>
</tr>
<tr>
<td>3 (Breakpoint)</td>
<td>0xf0223933</td>
</tr>
<tr>
<td>4 (Overflow)</td>
<td>0xf0333333</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8 (Double Fault)</td>
<td>0xf0222293</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>14 (Page Fault)</td>
<td>0xf0133390</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x30 (syscall in JOS)</td>
<td>0xf0222222</td>
</tr>
</tbody>
</table>
Handling Interrupt/Exceptions

- Set an Interrupt Descriptor Table (IDT)

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>t_divide</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>t_debug</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>t_nmi</td>
</tr>
<tr>
<td>3 (Breakpoint)</td>
<td>t_brkpt</td>
</tr>
<tr>
<td>4 (Overflow)</td>
<td>t_oflow</td>
</tr>
<tr>
<td>...</td>
<td>t_oflow</td>
</tr>
<tr>
<td>8 (Double Fault)</td>
<td>t_dblflt</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>14 (Page Fault)</td>
<td>t_pgflt</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x30 (syscall in JOS)</td>
<td>t_syscall</td>
</tr>
</tbody>
</table>

Figure 6-1. Relationship of the IDTR and IDT
Handling Interrupt/Exceptions

• Set an Interrupt Descriptor Table (IDT)

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>t_divide</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>t_debug</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>t_nmi</td>
</tr>
<tr>
<td>3 (Breakpoint)</td>
<td>t_brkpt</td>
</tr>
<tr>
<td>4 (Overflow)</td>
<td>t_oflow</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8 (Double Fault)</td>
<td>t_dblflt</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>14 (Page Fault)</td>
<td>t_pgflt</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x30 (syscall in JOS)</td>
<td>t_syscall</td>
</tr>
</tbody>
</table>
## Interrupt/Exception Handler

### Processing Interrupt/Exception

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>t_divide</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>t_debug</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>t_nmi</td>
</tr>
<tr>
<td>3 (Breakpoint)</td>
<td>t_brkpt</td>
</tr>
<tr>
<td>4 (Overflow)</td>
<td>t_oflow</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8 (Double Fault)</td>
<td>t_dblflt</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>14 (Page Fault)</td>
<td>t_pgflt</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x30 (syscall in JOS)</td>
<td>t_syscall</td>
</tr>
</tbody>
</table>

![Diagram showing processing of interrupts and exceptions](diagram.png)
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...

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>t_divide</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>t_debug</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>t_nmi</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Program | Interrupt | Interrupt #2
--------|-----------|-------------
Int $14  | Run t_pgflt | Int $2       | Run t_nmi   |
Int $0   |            |             |             |
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 request** coming during handling an interrupt request could make our interrupt handing never finish!

To avoid such an ‘infinite’ interrupt, we disable interrupt while handling interrupt...

<table>
<thead>
<tr>
<th>Interrupt Number</th>
<th>Code address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Divide error)</td>
<td>t_divide</td>
</tr>
<tr>
<td>1 (Debug)</td>
<td>t_debug</td>
</tr>
<tr>
<td>2 (NMI, Non-maskable Interrupt)</td>
<td>t_nmi</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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)
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

Program

Interrupt

Int $14
Run t_pgflt
resume
iret
Storing an Execution Context

```c
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

Accessing a global variable, Page fault!

Int $14

Run t_pgflt
How to Store an Execution Context?

```c
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;
}
```

```
return addr
Saved EBP
???
???
???
var i : 3
var j : 5
var sum : i
```

```
CPU

Program Stack
```

```
Registers

eax
ebx
ebx
ecx
edx
esi
edi
ebp
esp
eip
cs
ds
es
fs
gs
ss
```

Privilege level
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!)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000</td>
<td>old SS</td>
<td>&quot; - 4</td>
</tr>
<tr>
<td></td>
<td>old ESP</td>
<td>&quot; - 8</td>
</tr>
<tr>
<td></td>
<td>old EFLAGS</td>
<td>&quot; - 12</td>
</tr>
<tr>
<td>0x00000</td>
<td>old CS</td>
<td>&quot; - 16</td>
</tr>
<tr>
<td></td>
<td>old EIP</td>
<td>&quot; - 20</td>
</tr>
</tbody>
</table>

CPU only stores esp, eip, EFLAGS, ss, cs

What about the others?
TrapFrame in JOS Stores the Context

```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 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));
```
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_brkp, 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(tdblflt, 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 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:
How Can You Know an Interrupt/Exception has EC/NOEC?

- Intel Manual
  - IA-32 Developer’s Manual
  - (page 186)
EC? NOEC? Error Code!

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

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;       /* function definition */
name:           /* function starts here */
pushl $(num);  
jmp _alltraps

Pushes 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!
How Can We Store a TrapFrame?

```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 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_eip;         ✓
    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 to CPU!
- Call interrupt handler 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);
    fine TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $%(num);
    jmp _alltraps
}

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

Build a Trapframe!
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)
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 = 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

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 1st argument from the edx of tf
    • Address of “asdf”
  • Read 2nd argument from ecx of tf
    • `4`
  • call `sys_cputs("asdf", 4) // in kernel`
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/](https://syscalls.kernelgrok.com/) : lists 337 system calls...
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...
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
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!

```c
void env_pop_tf(struct Trapframe *tf) {
  asm volatile("\tmovl %0,%esp\n"
    "\topal\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 $0xAA

• Call trap gate

• Handle trap!

• Pop context

• iret

• Execution resumes...