Interrupt Service Routines

Interrupt Service Routines, or ISRs, are used to save the current processor state and set up the appropriate segment registers needed for kernel mode before the kernel's C-level interrupt handler is called. This can all be handled in about 15 or 20 lines of assembly language, including calling our handler in C. We need to also point the correct entry in the IDT to the correct ISR in order to handle the right exception.

An Exception is a special case that the processor encounters when it cannot continue normal execution. This could be something like dividing by zero: The result is an unknown or non-real number, so the processor will cause an exception so that the kernel can stop that process or task from causing any problems. If the processor finds that a program is trying to access a piece of memory that it shouldn't, it will cause a General Protection Fault. When you set up paging, the processor causes a Page Fault, but this is recoverable: you can map a page in memory to the faulted address - but that's for another tutorial.

The first 32 entries in the IDT correspond to Exceptions that can possibly be generated by the processor, and therefore need to be handled. Some exceptions will push another value onto the stack: an Error Code value which is specific to the exception caused.

Exception #	Description	Error Code?
0	Division By Zero Exception	No
1	Debug Exception	No
2	Non Maskable Interrupt Exception	No
3	Breakpoint Exception	No
4	Into Detected Overflow Exception	No
5	Out of Bounds Exception	No
6	Invalid Opcode Exception	No
7	No Coprocessor Exception	No
8	Double Fault Exception	Yes
9	Coprocessor Segment Overrun Exception	No
10	Bad TSS Exception	Yes
11	Segment Not Present Exception	Yes
12	Stack Fault Exception	Yes
13	General Protection Fault Exception	Yes
14	Page Fault Exception	Yes
15	Unknown Interrupt Exception	No
16	Coprocessor Fault Exception	No
17	Alignment Check Exception (486+)	No
18	Machine Check Exception (Pentium/586+)	No
19 to 31	Reserved Exceptions	No

As mentioned earlier, some exceptions push an error code onto the stack. To decrease the complexity, we handle this by pushing a dummy error code of 0 onto the stack for any ISR that doesn't push an error code already. This keeps a uniform stack frame. To track which exception is firing, we also push the interrupt number on the stack. We use the assembler opcode 'cli' to disable interrupts and prevent an IRQ from firing, which could possibly otherwise cause conflicts in our kernel. To save space in the kernel and make a smaller binary output file, we get each ISR stub to jump to a common 'isr_common_stub'. The 'isr_common_stub' will save the processor state on the stack, push the current stack address onto the stack (gives our C handler the stack), call our C 'fault_handler' function, and finally restore the state of the stack. Add this code to 'start.asm' in the provided space, filling out all 32 ISRs:

```
; In just a few pages in this tutorial, we will add our Interrupt
; Service Routines (ISRs) right here!
global _isr0
global _isr1
global _isr2
...               ; Fill in the rest here!
global _isr30
global _isr31

;  0: Divide By Zero Exception
_isr0:
    cli
    push byte 0    ; A normal ISR stub that pops a dummy error code to keep a
                   ; uniform stack frame
    push byte 0
    jmp isr_common_stub

;  1: Debug Exception
_isr1:
    cli
    push byte 0
    push byte 1
    jmp isr_common_stub

...               ; Fill in from 2 to 7 here!

;  8: Double Fault Exception (With Error Code!)
_isr8:
    cli
    push byte 8        ; Note that we DON'T push a value on the stack in this one!
                   ; It pushes one already! Use this type of stub for exceptions
                   ; that pop error codes!
    jmp isr_common_stub

...               ; You should fill in from _isr9 to _isr31 here. Remember to
                   ; use the correct stubs to handle error codes and push dummies!

; We call a C function in here. We need to let the assembler know
; that '_fault_handler' exists in another file
extern _fault_handler

; This is our common ISR stub. It saves the processor state, sets
; up for kernel mode segments, calls the C-level fault handler,
; and finally restores the stack frame.
isr_common_stub:
    pusha
    push ds
    push es
    push fs
    push gs
    mov ax, 0x10   ; Load the Kernel Data Segment descriptor!
    mov ds, ax
    mov es, ax
```

```
    mov fs, ax
    mov gs, ax
    mov eax, esp   ; Push us the stack
    push eax
    mov eax, _fault_handler
    call eax       ; A special call, preserves the 'eip' register
    pop eax
    pop gs
    pop fs
    pop es
    pop ds
    popa
    add esp, 8     ; Cleans up the pushed error code and pushed ISR number
    iret           ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP!
```

Add this to 'start.asm' in the spot we indicated in "The Basic Kernel"

Create yourself a new file called 'isrs.c'. Once again, remember to add the appropriate line to get GCC to compile the file in 'build.bat'. Add the file 'isrs.o' to LD's list of files so that it gets linked into the kernel. 'isrs.c' is rather straight-forward: declare our regular #include line, declare the prototypes of each of the ISRs from inside 'start.asm', point the IDT entry to the correct ISR, and finally, create an interrupt handler in C to service all of our exceptions generically. I'll leave it up to you to fill in the holes here:

```
#include < system.h >

/* These are function prototypes for all of the exception
*  handlers: The first 32 entries in the IDT are reserved
*  by Intel, and are designed to service exceptions! */
extern void isr0();
extern void isr1();
extern void isr2();

...                             /* Fill in the rest of the ISR prototypes here */

extern void isr29();
extern void isr30();
extern void isr31();

/* This is a very repetitive function... it's not hard, it's
*  just annoying. As you can see, we set the first 32 entries
*  in the IDT to the first 32 ISRs. We can't use a for loop
*  for this, because there is no way to get the function names
*  that correspond to that given entry. We set the access
*  flags to 0x8E. This means that the entry is present, is
*  running in ring 0 (kernel level), and has the lower 5 bits
*  set to the required '14', which is represented by 'E' in
*  hex. */
void isrs_install()
{
    idt_set_gate(0,  (unsigned)isr0,  0x08,  0x8E);
    idt_set_gate(1,  (unsigned)isr1,  0x08,  0x8E);
    idt_set_gate(2,  (unsigned)isr2,  0x08,  0x8E);
    idt_set_gate(3,  (unsigned)isr3,  0x08,  0x8E);

    ...                         /* Fill in the rest of these ISRs here */

    idt_set_gate(30,  (unsigned)isr30,  0x08,  0x8E);
    idt_set_gate(31,  (unsigned)isr31,  0x08,  0x8E);
}

/* This is a simple string array. It contains the message that
*  corresponds to each and every exception. We get the correct
*  message by accessing like:
*  exception_message[interrupt_number] */
```

```
unsigned char *exception_messages[] =
{
    "Division By Zero",
    "Debug",
    "Non Maskable Interrupt",

    ...                         /* Fill in the rest here from our Exceptions table */

    "Reserved",
    "Reserved"
};

/* All of our Exception handling Interrupt Service Routines will
*  point to this function. This will tell us what exception has
*  happened! Right now, we simply halt the system by hitting an
*  endless loop. All ISRs disable interrupts while they are being
*  serviced as a 'locking' mechanism to prevent an IRQ from
*  happening and messing up kernel data structures */
void fault_handler(struct regs *r)
{
    /* Is this a fault whose number is from 0 to 31? */
    if (r->int_no < 32)
    {
        /* Display the description for the Exception that occurred.
        *  In this tutorial, we will simply halt the system using an
        *  infinite loop */
        puts(exception_messages[r->int_no]);
        puts(" Exception. System Halted!\n");
        for (;;);
    }
}
```

The contents of 'isrs.c'

Wait, we have a new structure here as an argument to 'fault_handler': struct 'regs'. In this case, 'regs' is a way of showing the C code what the stack frame looks like. Remember that in 'start.asm' that we push a pointer to the stack onto the stack itself: this is so that we may be able to retrieve any error codes and interrupt numbers from the handlers themselves. This design is what allows us to use the same C handler for each different ISR and still be able to tell which exception or interrupt actually happened.

```
/* This defines what the stack looks like after an ISR was running */
struct regs
{
    unsigned int gs, fs, es, ds;      /* pushed the segs last */
    unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;  /* pushed by 'pusha' */
    unsigned int int_no, err_code;    /* our 'push byte #' and ecodes do this */
    unsigned int eip, cs, eflags, useresp, ss;   /* pushed by the processor automatically */
};
```

Defines a stack frame pointer argument. Add this to 'system.h'

Open 'system.h' and add the definition to struct 'regs' as well as the function prototype for 'isrs_install' so that we can call it from in 'main.c'. Finally, call 'isrs_install' from in our 'main' function, right after we install our new IDT. It would be a good idea to test out the exception handlers in our kernel now.

OPTIONAL: In 'main', add some tester code that will divide a number by zero. As soon as the processor encounters this, the processor will generate a "Divide By Zero" Exception, and you will see that appear on the screen! When you test that, and it works, you can

delete your exception causing code (the 'putch(myvar / 0);' line, or whatever you decide to write.

You may find the complete solution to 'start.s' here, and the complete solution to 'isrs.c' here.