Interrupt Service Routines

This article refers to its readers using *you* in an unencyclopedic manner.
It should be edited to be in an encyclopedic tone.

The x86 architecture is an interrupt driven system. External events trigger an interrupt — the normal control flow is interrupted and an **Interrupt Service Routine** (ISR) is called.

Such events can be triggered by hardware or software. An example of a hardware interrupt is the keyboard: every time you press a key, the keyboard triggers IRQ1 (Interrupt Request 1), and the corresponding interrupt handler is called. Timers and disk request completion are other possible sources of hardware interrupts.

Software driven interrupts are triggered by the `int` opcode; e.g. the services provided by MS-DOS are called by the software triggering `INT 21h` and passing the applicable parameters in CPU registers.

For the system to know which interrupt service routine to call when a certain interrupt occurs, offsets to the ISRs are stored in the Interrupt Descriptor Table when you're in Protected mode, or in the Interrupt Vector Table when you're in Real Mode.

An ISR is called directly by the CPU, and the protocol for calling an ISR differs from calling e.g. a C function. Most importantly, an ISR has to end with the `iret` opcode (or `iretq` in long mode—yes, even when using intel syntax), whereas usual C functions end with `ret` or `retf`. The obvious but nevertheless wrong solution leads to one of the most "popular" triple-fault errors among OS programmers.

Contents

When the Handlers are Called

x86

When the CPU calls the interrupt handlers, the CPU pushes these values onto the stack in this order:

```
(SS -> ESP) -> EFLAGS -> CS -> EIP
```

The SS:ESP values are only pushed when a privilege-level change occurs. The CS value is padded with two bytes to form a doubleword.

If the gate type is not a trap gate, the CPU will clear the interrupt flag. If the interrupt is an exception, the CPU will push an error code onto the stack, as a doubleword.

The CPU will load the segment-selector value from the associated IDT descriptor into CS.

x86-64

When the CPU calls the interrupt handlers, it changes the value in the RSP register to the value specified in the IST, and if there is none, the stack stays the same. Onto the new stack, the CPU pushes these values in this order:

```
SS:RSP (original RSP) -> RFLAGS -> CS -> RIP
```

CS is padded to form a quadword.

If the interrupt is called from a different ring, SS is set to 0, indicating a null selector. The CPU will modify the RFLAGS register, setting the TF, NT, and RF bits to 0. If the gate type is a trap gate, the CPU will clear the interrupt flag.

If the interrupt is an exception, the CPU will push an error code onto the stack, padded with bytes to form a quadword.

The CPU will load the segment-selector value from the associated IDT descriptor into CS, and check to ensure that CS is a valid code segment selector.

The Problem

Many people shy away from assembly, and want to do as much as possible in their favorite high-level language. GCC (as well as other compilers) allow you to add inline assembly, so many programmers are tempted to write an ISR like this:

```
/* How NOT to write an interrupt handler */
void interrupt_handler(void)
{
    asm("pushad"); /* Save registers. */
    /* do something */
    asm("popad");  /* Restore registers. */
    asm("iret");   /* This will triple-fault! */
}
```

This cannot work. The compiler doesn't understand what is going on. It doesn't understand that the registers and stack are required to be preserved between the asm statements; the optimizer will likely corrupt the function. Additionally, the compiler adds stack handling code before and after your function, which together with the iret results in assembly code resembling this:

```
push   %ebp
mov    %esp,%ebp
sub    $<size of local variables>,%esp
pushad
# C code comes here
popad
iret
# 'leave' if you use local variables, 'pop %ebp' otherwise.
leave
ret
```

It should be obvious how this messes up the stack (ebp gets pushed but never popped). Don't do this. Instead, these are your options.

Solutions

Plain Assembly

Learn enough about assembly to write your interrupt handlers in it. ;-)

Two-Stage Assembly Wrapping

Write an assembly wrapper calling the C function to do the real work, and then doing the iret.

```
; filename: isr_wrapper.s
.globl   isr_wrapper
.align   4

isr_wrapper:
    pushad
    cld    ; C code following the sysV ABI requires DF to be clear on function entry
    call interrupt_handler
    popad
    iret
```

```
/* filename: interrupt_handler.c */
void interrupt_handler(void)
{
    /* do something */
}
```

Generating ISRs (GNU AS macros)

```
/* Simple sequential list generator, ord determines "order" */
.macro SeqGenerator inst:req, ord:req, n=%rsp, va:vararg
  .ifc "\n","%rsp"
    .exitm
  .else
    .ifgt \ord /* ord: Non-zero */
      \inst \n
      SeqGenerator \inst \ord \va
    .else /* ord: Zero */
      SeqGenerator \inst \ord \va
      \inst \n
    .endif
  .endif
.endm

.macro IsrStub n=0
.global AsmInt\n\()Vector
.align 16
AsmInt\n\()Vector:
  /* Swapgs if needed (see below) */
  cmpb $0x08, 0x8(%rsp)
  je 1f
  swapgs
1:
  /* Saving is manually required in x86_64, no pushaq exists */
  SeqGenerator pushq 1 %r15 %r14 %r13 %r12 %r11 %r10 %r9 %r8 %rbp %rdi %rsi %rdx %rcx %rbx %rax
  /* fxsave/xsave here if needed */
  movq %rsp, %rdi /* Argument 1: Interrupt frame */
  movq $\n, %rsi /* Argument 2: IRQ# */
  callq IntDeviceVector
  /* fxrstor/xrstor here if needed */
  SeqGenerator popq 0 %r15 %r14 %r13 %r12 %r11 %r10 %r9 %r8 %rbp %rdi %rsi %rdx %rcx %rbx %rax
  /* Swapgs if needed (see below) */
  cmpb $0x08, 0x8(%rsp)
  je 1f
  swapgs
1:
  iretq
.endm

/* Write out your required stubs */
IsrStub 64
IsrStub 65
IsrStub 66
IsrStub 67
/* etc... */
```

Compiler Specific Interrupt Directives

Some compilers for some processors have directives allowing you to declare a routine interrupt, offer a #pragma interrupt, or a dedicated macro. Clang 3.9, Borland C, Watcom C/C++, Microsoft C 6.0 and Free Pascal Compiler 1.9.* upward and GCC offer this. Visual C++ offers an alternative shown under **Naked Functions**:

Clang

As of version 3.9 it supports interrupt attribute (https://clang.llvm.org/docs/AttributeReference.html#interrupt-x86) for x86/x86-64 targets.

```
struct interrupt_frame
{
    uword_t ip;
    uword_t cs;
    uword_t flags;
    uword_t sp;
    uword_t ss;
};

__attribute__ ((interrupt))
void interrupt_handler(struct interrupt_frame *frame)
{
    /* do something */
}
```

Borland C

```
/* Borland C */
void interrupt interrupt_handler(void)
{
    /* do something */
}
```

Watcom C/C++

```
/* Watcom C/C++ */
void _interrupt interrupt_handler(void)
{
    /* do something */
}
```

Naked Functions

Some compilers can be used to make interrupt routines, but require you to manually handle the stack and return operations. Doing so requires that the function is generated without an epilogue or prologue. This is called making the function *naked* — this is done in Visual C++ by adding the attribute *_declspec(naked)* to the function. You need to verify that you do include a return operation (such as *iretd*) as that is part of the epilogue that the compiler has now been instructed to not include.

If you intend to use local variables, you must set up the stack frame in the manner which the compiler expects; as ISRs are non-reentrant, however, you can simply use static variables.

Visual C++

Visual C++ also supplies the __LOCAL_SIZE assembler macro, which notifies you how much space is required by the objects on the stack for the function.

```
/* Microsoft Visual C++ */
void _declspec(naked) interrupt_handler()
{
    _asm pushad;

    /* do something */

    _asm{
        popad
        iretd
    }
}
```

GCC / G++

The online docs (https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html#x86-Function-Attributes) say that by using GCC's function attributes, they have added the ability to write interrupt handlers in C interface by using __attribute__((interrupt)). So instead of:

```
/* BLACK MAGIC – strongly discouraged! */
void interrupt_handler() {
    __asm__("pushad");
    /* do something */
    __asm__("popad; leave; iret"); /* BLACK MAGIC! */
}
```

You can have the correct (GCC) form:

```
struct interrupt_frame;

__attribute__((interrupt)) void interrupt_handler(struct interrupt_frame* frame)
{
    /* do something */
}
```

The documentation for GCC states that if the interrupt attribute is used, the iret instruction will be used instead of ret on x86 and x86-64 architectures. It also says, "Since GCC doesn't preserve SSE, MMX nor x87 states, the GCC option -mgeneral-regs-only should be used to compile interrupt and exception handlers."

Black Magic

Look at the faulty code above, where the proper C function exit code was skipped, screwing up the stack. Now, consider this code snippet, where the exit code is added manually:

```
/* BLACK MAGIC – strongly discouraged! */
void interrupt_handler() {
    __asm__("pushad");
    /* do something */
    __asm__("popad; leave; iret"); /* BLACK MAGIC! */
}
```

The corresponding output would look somewhat like this:

```
push   %ebp
mov    %esp,%ebp
sub    $<size of local variables>,%esp
pushad
# C code comes here
popad
leave
iret
leave # dead code
ret   # dead code
```

This assumes that `leave` is the correct end-of-function handling — you are doing the function return code "by hand", and leave the compiler-generated handling as "dead code". Needless to say, such assumptions on compiler internals are dangerous. This code can break on a different compiler, or even a different version of the same compiler. It is therefore strongly discouraged, and listed only for completeness.

Assembly Goto

Full article: ISRs, PIC, And Multitasking

Since version 4.5, GCC supports the "asm goto" statement. It can be used to make ISRs as functions which return the correct address of the ISR entry point.

Zig

Since version 0.6.0, zig supports an interrupt calling convention. *However currently, when used non-LLVM backend, this convention does not affect code generation at all and is never working.*

```
export fn isr(ptr: *InterruptStackFrame, errcode: u32) callconv(.Interrupt) void {
}
```