Nano-Kernel : A Bare Metal OS

## Part 5 - Handling Interrupts

First, the Intel x86 architecture defines two types of interrupts:

* Exceptions - which are defined as *synchronous events* generated as a result of the failure to execute an instruction. Examples include division by zero, accessing an illegal memory address, or attempting to execute a privileged instruction.
* Interrupts - which are defined as *asynchronous* *events* that are generated by external devices to get the attention of the system. Examples include typing a key on the keyboard or the arrival of a network packet on a controller.   
  + Hardware interrupts are generated by external hardware events
  + Software interrupts are generated by programs running on the system executing the “INT” instruction.

Both types of interrupts are handled by the same mechanism. When the interrupt is detected the CPU will do the following:

1. If the interrupt is configured as a device interrupt (see the IDT below) then the IF flag is cleared preventing nested interrupts. Exceptions do not clear this flag.
2. If the handler has more privilege than the currently running task the CPU will save the SS, ESP, EFLAGS, CS, and EIP registers internally. The CPU then loads the SS and SP from the tasks’s “[Task State Segment](https://en.wikipedia.org/wiki/Task_state_segment)” (TSS), and the SS, ESP, EFLAGS, CS, and EIP are pushed (in that order) to the tasks TSS.
3. If the handler has the same privilege as the current task, the handler will use the same stack, and EFLAGS, CS, and EIP are pushed to the current stack.
4. Some exceptions have an error code, and this will be pushed to the stack.
5. The segment register will be loaded using the GDT entry of the handler.
6. The CPU will do a jump to the first instruction of the handler.
7. The handler signifies that it is complete by executing the “IRET” instruction.
8. If the handler has the same privilege, then the EIP, CS, and EFLAGS are popped from the call stack. By updating EIP, the next instruction will be back to what was running.
9. If the handler has a higher privilege then the ESP, SS EFLAGS, EIP, and CS are restores from the TSS segment of the original program. By updating EIP, the next instruction will be back to what was running.

All along, we’ve talked about the differences between CISC and RISC. This gives a really good example. Interrupt handling is so un-RISC! Clearly, handling an interrupt is going to involve a series of operations performed by the CPU. These will take time - potentially a variable length of time. The control unit that implements this is clearly quite complex.

### The Stack During an Interrupt

|  |  |
| --- | --- |
| SP (before INT) --> | Current Tasks TOS |
|  | EFLAGS |
|  | CS |
| SP --> | EIP |

The IRET instruction will un-do whatever was pushed onto the stack (except for the error code that is part of some errors).

### Interrupt Handlers

Interrupt handlers “look” like normal functions. In fact, they can *almost* be implemented as a normal C function. The *prologue*, the part of the code that handles the entrance to the function and the *epilogue,* are both different. The call-stack for the interrupt is different (as noted above) and not entirely compatible with the C calling convention.

|  |
| --- |
| static int count = 0; void simple\_fun( ) {  count++;  return; } |

|  |  |
| --- | --- |
| # Normal C Function count .word 0 simple\_fun:  push %eax  mov ($count), %eax  add 1, %eax  mov %eax,($count)  pop %eax  **ret** | # Interrupt Implementation count .word 0 simple\_fun:  push %eax  mov ($count), %eax  add 1, %eax  mov %eax,($count)  pop %eax  **iret** |

Note that the only difference between these two Assembly implementations is that the interrupt handler must execute the special “IRET” instruction. The “ret” or “iret” instruction is part of the *epilogue* of the function (it happens at the end, as opposed to the *prologue*) of the function.

If the handler operates in a different protection level then there is also some additional prolog and epilog that must happen in the lower-ring handler to choose the proper GDT segments.

There is not a standard way to tell the C compiler that a particular function needs this special prolog / epilog. When the compiler translates the code it will treat it like any other function. You may recall that the Microchip C compiler did have a special annotation to do just this.

Instead, we will use a *wrapper* function, written in Assembly, that will handle the special prolog and epilog. Since the rest of the interrupt function is just a normal function we can simply *call* the function like any other C function.

Thus, the CPU will jump into our Assembly handler-wrapper, which will take care to setup the machine, call the real handler, and on the normal return, clean-up the wrapper, and end with the special iret instruction. This wrapper will also remove the error number from the stack of those exceptions that push the error number at the start of the exception:

|  |
| --- |
| # ASM wrapper function for interrupt #10  .type \_excp\_10, @function  \_excp\_10:  cli # clear interrupt flag  pusha # push all registers to stack  mov $0x10, %ax # select GDT segment 0x10 - kernel data  mov %ax,%ds # load the segment regs (like we did)  mov %ax,%es  mov %ax,%fs  mov %ax,%gs   call handle\_excp\_10 # call the C function “handle\_excp\_10”  popa # pop all old values from the stack  add $4, %esp # put stack back  iret # return from interrupt |

### Interrupt Descriptor Table

In protected mode, the Intel x86 architecture uses an [*Interrupt Descriptor Table*](https://wiki.osdev.org/Interrupt_Descriptor_Table) (IDT) that connects a handler to one of the 256 interrupt numbers to a handler. Because the CPU is in protected mode, the protection level and segment for the handler must be associated with each interrupt. The process to setup the IDT is very similar to the GDT table of the previous section.

The example that we will be build here uses an admixture of C and Assembly. We could have done the same with the GDT, but it was relatively simple to manage in Assembly. The IDT will be more complex because we eventually want to have C code called by the interrupt handler:

We would need to have something almost identical for all other interrupts - the handler’s starting address would be unique, and the name of the C function called may/will be different.

### Function-Like Macros to Define Functions

Since each interrupt handler will be *almost* but not *completely* identical, we will need a unique handler for each one. To make this easier, I developed a *function-like macro* that was stored in a file “intr.h”. Using the following macro, we could create the code shown in the previous example by using: EXCP(10, handle\_excp\_10). The C-pre processor will evaluate the macro replacement to be equivalent code. First, consider the macro:

|  |
| --- |
| // this is so that the ## macro works  // EXCP gets the symbolic name  // which is re-written for the EXCP2 macro  #define EXCP(num,handler) EXCP2(num,handler)  #define EXCP2(num,handler) \  .global \_excp\_##num ; \  .type \_excp\_##num, @function ; \  \_excp\_##num: \  cli ; \  push $(num) ; \  pusha ; \  mov $0x10, %ax ; \  mov %ax,%ds ; \  mov %ax,%es ; \  mov %ax,%fs ; \  mov %ax,%gs ; \  call handler ; \  popa ; \  add $4, %esp  iret |

The “##” operator is the *concatenation* operator for the pre-processor. It concatenates whatever value it is given - without further evaluation which will cause problems when the values used in the EXCP call need further evaluation. There is a common trick: define the first “EXCP” macro to be replaced with the “EXCP2” macro. The first macro replacement converts all values to their numeric equivalents and then evaluates the EXCP2 macro with those cooked values.

For example, suppose we have:

#define EXCP\_BASE 32  
#define EXCP\_10 (INTR\_EXCP + 10)  
EXCP(EXCP\_10, handle\_excp\_10)

Then: EXCP(EXCP\_10, handle\_exp\_10) is replaced first with: EXCP2(42, handle\_excp\_10), which is then replaced with the assembly replacement above. Note that because of this “trick”, it will will create a handler: \_excp\_42 and not \_excp\_EXCP\_10 as it would without this trick.

There is another issue regarding multiple lines. The pre-processor will **not** create a multi-line replacement. Therefore each of the assembly statements is separated by a semi-colon. These are needed so that the assembler recognizes the breaks between statements.

One can cause the C compiler to stop processing after the pre-processor and print-out the replaced file by using the “-E” flag:

i686-elf-gcc -E intr.S

Results of the preprocessor:

|  |
| --- |
| .global \_excp\_10 ; .type \_excp\_10, @function ; \_excp\_10: cli ; push $(10) ; pusha ; mov %ds, %ax ; push %eax ; mov $0x10, %ax ; mov %ax,%ds ; mov %ax,%es ; mov %ax,%fs ; mov %ax,%gs ; call int\_handler ; pop %eax ; mov %ax,%ds ; mov %ax,%es ; mov %ax,%fs ; mov %ax,%gs ; popa ; add $4, %esp |

### General Interrupt Handlers

One of the tricks to simplify handling interrupts is to let the C function deal with dispatching the interrupt to the right place. But the **only** time we know which interrupt was generated is when the specific handler is invoked by the CPU, and the only way we know that is because it was the specific function that invoked by the CPU.

The function above pushes the interrupt number onto the call stack just before calling the C function. This makes it appear as the first argument to the C function (it would be like putting the value in the $A0 register in the MIPS). Thus, the *prototype* of the C interrupt handler will be:

void handle(int intr\_num)

This makes it easy to have one handler that can determine which interrupt was detected. It also makes it easy to pay attention or ignore an interrupt based on the current configuration of the machine.

In order to pass an argument to a function, the Intel standard uses the stack. The standard Intel calling conventions say that the stack is in reverse order - last function argument is first PUSHed so that the first argument is at the top of the stack. When the call to “handle” is made, the stack should look like:

|  |  |
| --- | --- |
| SB before INTR | Stack of previous task |
| SP after wrapper -> | EFLAGS, CS, and EIP |
|  | Registers saved by interrupt wrapper |
| SP --> | Interrupt number (pushed by wrapper) |

Or, if its an error code:

|  |  |
| --- | --- |
| SP before INTR | Stack of previous task |
|  | EFLAGS, CS, and EIP (pushed by CPU) |
| SP after *wrapper ->* | Error Code (pushed by CPU) |
|  | Registers saved by interrupt wrapper |
|  | Error Code (pushed by wrapper) |
| SP --> | Interrupt number (pushed by wrapper) |

If you give the interrupt wrapper presented here an examination, you should quickly determine that the stacks do not match. Even worse, the error code will be in the wrong place on the stack!   
  
**Modify the interrupt wrappers to setup the stack in the proper order. Use a piece of paper or whiteboard to trace what goes on to the stack and what comes off. Make sure that the stack pointer is always in the right place at the right time or the machine will crash.**

### 

### Exceptions and Errors

The following table shows the list of exceptions and errors defined by the Intel architecture. In our small, toy OS, it is unlikely that many of these will be generated - but then again, I’ve seen what students’ code looks like, so it is probably best to handle all of these!

The table includes a column Error Code? Which tells us which errors will have their error code number pushed onto the stack. It is the handler’s responsibility to remove this value from the stack before the *iret* instruction. The *epilogue* for these handlers will subtract *8* bytes from the stack, while all of the others will subtract *4* bytes.

One more note: the *triple fault* is one special interrupt that we cannot handle and cannot ignore. If a fault is detected in an exception handler, a *double fault* is generated. If the *double fault* handler is broken then the system has no choice but to generate a triple fault, which will reboot your machine.



|  |
| --- |
| void excp\_handler(int excp\_num)  {  switch(excp\_num) {  case 0: console\_puts("EXP 0 - Divide by zero"); break;  case 1: console\_puts("EXP 1 - Debug"); break;  case 2: console\_puts("EXP 2 - NMI"); break;  case 3: console\_puts("EXP 3 - Breakpoint"); break;  case 4: console\_puts("EXP 4 - Overflow"); break;  case 5: console\_puts("EXP 5 - Bounds reached"); break;  case 6: console\_puts("EXP 6 - Invalid OPCODE"); break;  case 7: console\_puts("EXP 7 - Device not available"); break;  case 16: console\_puts("EXP 16 - FPU Exception"); break;  case 18: console\_puts("EXP 18 - Machine check except."); break;  case 19: console\_puts("EXP 19 - SIMD FPU exception"); break;  default: console\_puts("EXP UNKNOWN"); break;  }  asm("hang\_excp: hlt; jmp hang\_excp;\n");  return;  }  void err\_handler(int err\_num)  {  switch(err\_num) {  case 8: console\_puts("ERR 8 - Double fault"); break;  case 10: console\_puts("ERR 10 - Invalid TSS"); break;  case 11: console\_puts("ERR 11 - Segment not present"); break;  case 12: console\_puts("ERR 12 - Stack segment fault"); break;  case 13: console\_puts("ERR 13 - General protection fault"); break;  case 14: console\_puts("ERR 14 - Page Fault"); break;  case 17: console\_puts("ERR 17 - Alignment check"); break;  default: console\_puts("EXP UNKNOWN"); break;  }  asm("hang\_err: hlt; jmp hang\_err;\n");  return;  } |

These two handlers each of the machine exceptions and errors for the Intel CPU. But at this point, they aren’t actually connected to the interrupt process. For that to happen, there must be an entry in the IDT. The entry in the IDT must point to a valid interrupt handler (not a C function), and then that handler must invoke one of these two functions.

|  |
| --- |
| // The valid task-gate functions  typedef enum {  TASK32=5, INTR16=6, TRAP16=7, INTR32=14, TRAP32=32  } idt\_gate\_t;  // The IDT entry  typedef struct {  uint16\_t offset\_lo;  uint16\_t selector;  uint8\_t zero;  uint8\_t type;  uint16\_t offset\_hi;  } \_\_attribute\_\_((packed)) idt\_entry\_t;  // The IDT description entry  typedef struct {  uint16\_t limit;  uint32\_t base;  } \_\_attribute\_\_((packed)) idt\_desc\_t;  // The IDT table itself static idt\_entry\_t idt\_entries[NUM\_IRQ];  // Add an entry to the IDT table, linking the interrupt number,  // the (index into the IDT), the gate type, and the **ASSEMBLY HANDLER** // that will call the C handler.  static void idt\_register\_handler(uint16\_t intnum, idt\_gate\_t type,   void (\*handler)(void))  {  uint32\_t handler\_address\_asint = (uint32\_t) handler;  idt\_entries[intnum].offset\_lo = (handler\_address\_asint & 0xffff);  idt\_entries[intnum].selector = 0x08;  idt\_entries[intnum].zero = 0;  idt\_entries[intnum].type = 0x8e;  idt\_entries[intnum].offset\_hi = (handler\_address\_asint >> 16) & 0xffff;  } |

There is one more bridge that needs to be constructed - the assembly language functions themselves. There are two parts that must be constructed - the assembly code (and its label) and a C language prototype that will make that assembly label “visible” to C.

The first part of the bridge: use the pre-processor macro to create the Assembly wrapper (in a file intr.S:

|  |
| --- |
| EXCP(0, excp\_handler)  EXCP(1, excp\_handler)  EXCP(2, excp\_handler)  EXCP(3, excp\_handler)  EXCP(4, excp\_handler)  EXCP(5, excp\_handler)  EXCP(6, excp\_handler)  EXCP(7, excp\_handler)  ERR(8, err\_handler)  ERR(10, err\_handler)  ERR(11, err\_handler)  ERR(12, err\_handler)  ERR(13, err\_handler)  ERR(14, err\_handler)  EXCP(16, excp\_handler)  ERR(17, err\_handler)  EXCP(18, excp\_handler)  EXCP(19, excp\_handler) |

And then the C prototypes (in handler.h):

|  |
| --- |
| void \_excp\_0( );  void \_excp\_1( );  void \_excp\_2( );  void \_excp\_3( );  void \_excp\_4( );  void \_excp\_5( );  void \_excp\_6( );  void \_excp\_7( );  void \_err\_8( );  void \_err\_10( );  void \_err\_11( );  void \_err\_12( );  void \_err\_13( );  void \_err\_14( );  void \_excp\_16( );  void \_err\_17( );  void \_excp\_18( );  void \_excp\_19( ); |

And, then, the actual linking:

|  |
| --- |
| void init\_handler( )  {  idt\_clear( );    idt\_register\_handler(0, TRAP32, \_excp\_0);  idt\_register\_handler(1, TRAP32, \_excp\_1);  idt\_register\_handler(2, TRAP32, \_excp\_2);  idt\_register\_handler(3, TRAP32, \_excp\_3);  idt\_register\_handler(4, TRAP32, \_excp\_4);  idt\_register\_handler(5, TRAP32, \_excp\_5);  idt\_register\_handler(6, TRAP32, \_excp\_6);  idt\_register\_handler(7, TRAP32, \_excp\_7);  idt\_register\_handler(8, TRAP32, \_err\_8);  idt\_register\_handler(10, TRAP32, \_err\_10);  idt\_register\_handler(11, TRAP32, \_err\_11);  idt\_register\_handler(12, TRAP32, \_err\_12);  idt\_register\_handler(13, TRAP32, \_err\_13);  idt\_register\_handler(14, TRAP32, \_err\_14);  ... |

After the interrupts are loaded into the IDT, we need to actually load the IDT into the machine using an IDT description and the LIDT instruction. The example loads the *limit* field with the 50 entries (they can be zero), and loads the *base* as the memory address of the idt\_entries table (from previous example).

|  |
| --- |
| idt\_desc\_t idt\_desc = { .limit = sizeof(idt\_entry\_t) \* 50 - 1,  .base = (uint32\_t) &idt\_entries };    intr\_loaddt(&idt\_desc); |

The function “intr\_loaddt” is actually implemented in assembly. There is a function prototype in “handler.h”:

|  |
| --- |
| extern void intr\_loaddt( ); |

And the actual implementation in: “intr.S”

|  |
| --- |
| .global intr\_loaddt  .type intr\_loaddt, @function    intr\_loaddt:  mov 4(%esp),%eax  lidt (%eax)  ret |

### Testing the Interrupt Handler

There are two ways to test the interrupt handler logic. The first is to generate one of the errors and see if your handler is executed. For example, we can add to “kernel.c”:

|  |
| --- |
| int x = 34; x = 34 / 0; console\_puts(“Error - exception was not detected!”); |

Or, we can just force the interrupt:

|  |
| --- |
| asm(“int 0x01;\r”);  console\_puts(“Error - exception was not detected!”); |

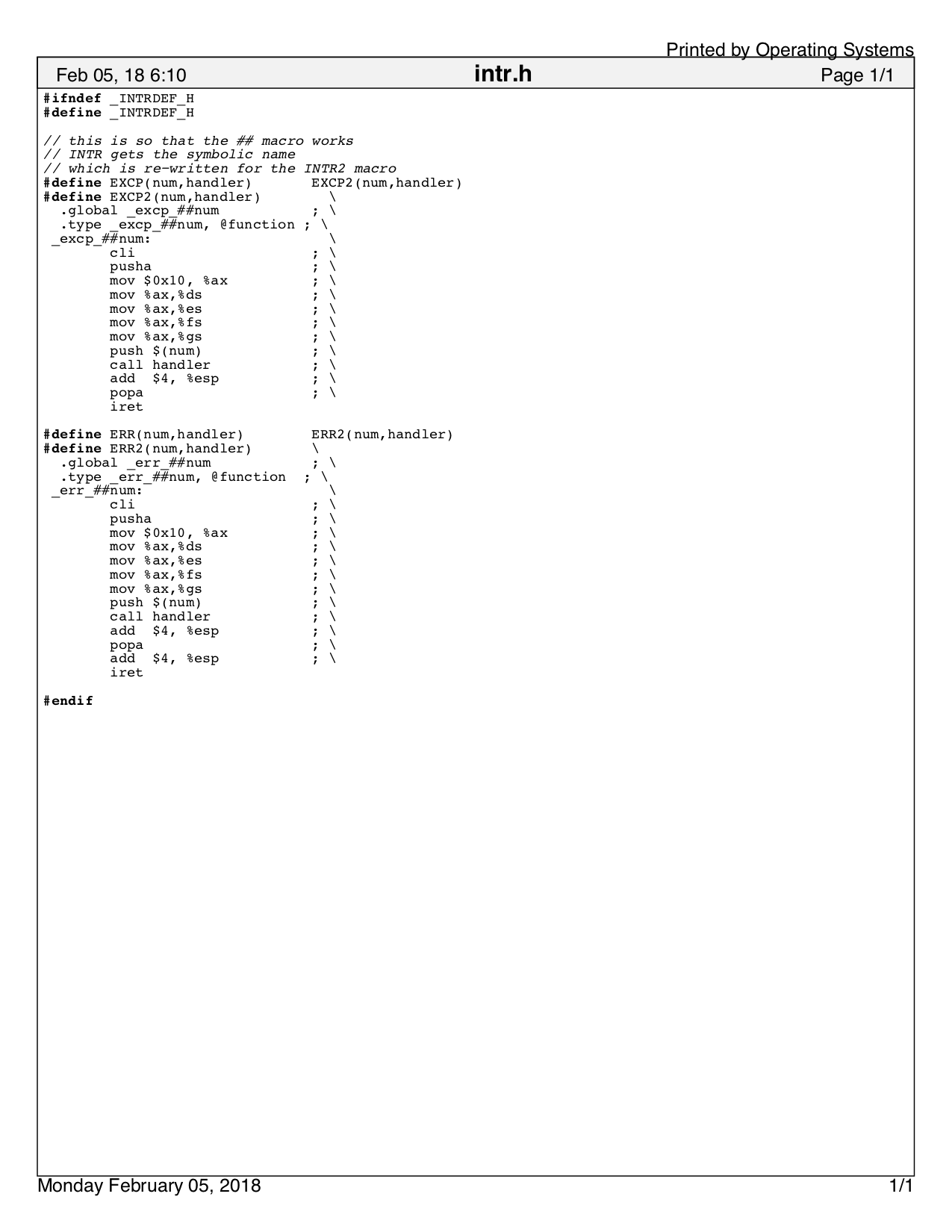
### 

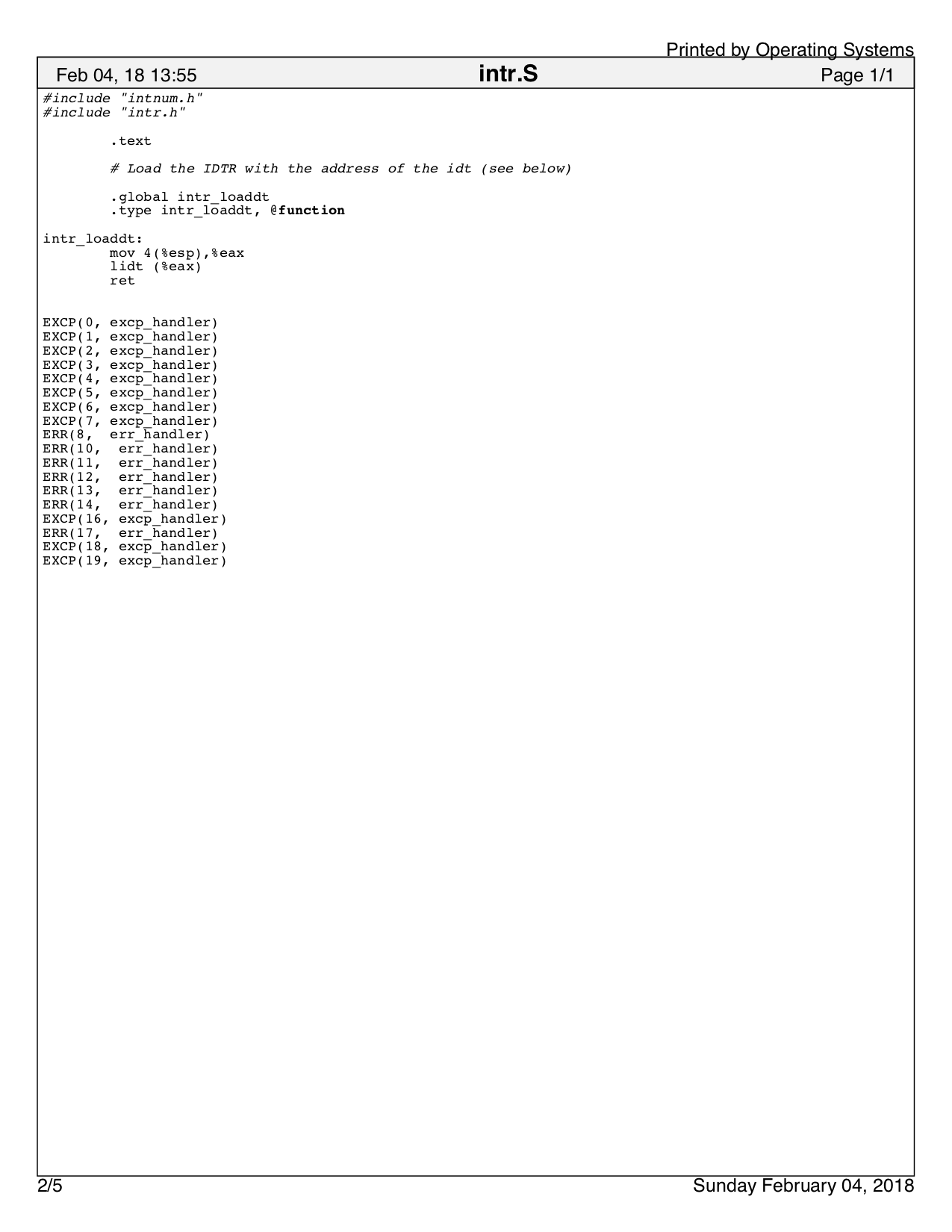
Note: you can use GDB to put break-points at the labels that were created by the wrapper macros. For example, put a break-point at “\_excp\_0” and generate the divide by zero. Use the “disassemble” command to make GDB display the instructions and “stepi” to step through them. If you didn’t fix the stack in the previous part of the lab this is probably where things will go wrong. Very, very wrong.

### 

### Interrupt Code So Far…

With the “broken” stack:











# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. Since I’ve given you most of the code for this section, I just want to see that you’ve implemented
2. Give me a demo of your system handling different types of interrupts

Deliver to me answers to the following questions (1 answer per group):

1. Lookup the “ret” and “iret” instruction in an Intel reference manual. Do they take any arguments? Are there other related instructions?
2. What would happen if you “ret” from an interrupt or “iret” from a function?

Points: \_\_\_\_\_\_\_\_\_ / 30