作業二：觀察中斷

學習目標：

- 了解 Linux 怎樣設定中斷向量表
- 了解驅動程式中，關於中斷的部分
- 了解如何追蹤 Linux、反組譯等技巧

題目：

1. 設定你的 Linux，執行 dbg-Linux5.0-in-QEMU.sh 及 Eclipse debugger，將中斷點設定在下列位置，讓 Linux 執行到中斷點的位置（紅色字標起來的部分即可），並附上螢幕截圖

   截圖 1. b trap_init

   - 目的：由 start_kernel（相當於 Linux 的 main）呼叫，初始化 Intel 處理器的 trap，共 19 個（請參考附一）
   - 動手操作及觀察：trap_init-> idt_setup_traps-> idt_setup_from_table
     - t->addr 就是中斷服務
     - size 會等於 19（請參考附一）
     - 反組譯第一個中斷服務常式（interrupt service routine，ISR）

```
(gdb) disass 0xffffffff82200b70
Dump of assembler code for function divide_error:
```

```
0xffffffff82200b70 <+0>:     nop
0xffffffff82200b71 <+1>:     nop
0xffffffff82200b72 <+2>:     nop
0xffffffff82200b73 <+3>:     pushq  $0xffffffffffffffff
0xffffffff82200b75 <+5>:     callq  0xffffffff82200fe0 <error_entry>
0xffffffff82200b7a <+10>:    mov    %rsp,%rdi
0xffffffff82200b7d <+13>:    xor    %esi,%esi
0xffffffff82200b7f <+15>:    callq  0xffffffff8103e235 <do_divide_error>
0xffffffff82200b84 <+20>:    jmpq   0xffffffff822010d0 <error_exit>
```

截圖 2.　b init_IRQ

- 目的：由 start_kernel（相當於 Linux 的 main）呼叫，設定外部中斷的中段向量表
- 動手操作及觀察：程式碼如下，留意一下中文註解的部分

```
void __init init_IRQ(void)
{
	int i;

	/*
	 * On cpu 0, Assign ISA_IRQ_VECTOR(irq) to IRQ 0..15.
	 * If these IRQ's are handled by legacy interrupt-controllers like PIC,
	 * then this configuration will likely be static after the boot. If
	 * these IRQ's are handled by more mordern controllers like IO-APIC,
	 * then this vector space can be freed and re-used dynamically as the
	 * irq's migrate etc.
	 */
	//初始化傳統的「外部中斷」的『陣列』，共16個
	for (i = 0; i < nr_legacy_irqs(); i++)
		per_cpu(vector_irq, 0)[ISA_IRQ_VECTOR(i)] = irq_to_desc(i);
	//會呼叫native_init_IRQ，真正設定中斷向量表
	x86_init.irqs.intr_init();
}
```

截圖 3. b native_init_IRQ

- 目的：init_IRQ 呼叫此函數「真正」去設定中斷向量表
- 動手操作及觀察：init_IRQ->native_init_IRQ -> idt_setup_apic_and_irq_gates -> idt_setup_from_table -> idt_init_desc
  - 初始化傳統的 16 個外部中斷

```
(gdb) p d->addr
$11 = (const void *) 0xffffffff822015d0 <reschedule_interrupt>
```

- 動手操作及觀察：init_IRQ->native_init_IRQ -> idt_setup_apic_and_irq_gates -> set_intr_gate
  - 初始化其餘的外部中斷

```
(gdb) p addr
$14 = (const void *) 0xffffffff82200218 <irq_entries_start+8>
(gdb) disassemble 0xffffffff82200218
Dump of assembler code for function irq_entries_start:
   0xffffffff82200210 <+0>:     pushq   $0x5f
   0xffffffff82200212 <+2>:     jmpq    0xffffffff82200940 <common_interrupt>
   0xffffffff82200217 <+7>:     nop
   0xffffffff82200218 <+8>:     pushq   $0x5e
   0xffffffff8220021a <+10>:    jmpq    0xffffffff82200940 <common_interrupt>
   0xffffffff8220021f <+15>:    nop
```

  - 到此可以發現外部中斷的處理方式是
    - 在每一個外部中斷處理函式（interrupt service routine，ISR，將一個特殊的編號放入堆疊，然後跳到 common_interrupt，因為是跳過去，所以

common_interrup 應該不會 return)

截圖 4. b serial_link_irq_chain

- 目的：UART 驅動程式向 Linux kernel 註冊當 serial port 裝置發生中斷時，應該呼叫哪個函數，此函數屬於驅動程式的一部分

```
(gdb) b serial_link_irq_chain
Breakpoint 9 at 0xffffffff81a018d2: file drivers/tty/serial/8250/8250_core.c, line 172.
(gdb) c
Continuing.

Breakpoint 9, serial_link_irq_chain (up=0x100000021) at
drivers/tty/serial/8250/8250_core.c:172
172	{
(gdb) l 212
207			} else {
208				INIT_LIST_HEAD(&up->list);
209				i->head = &up->list;
210				spin_unlock_irq(&i->lock);
211				irq_flags |= up->port.irqflags;
212				ret = request_irq(up->port.irq, serial8250_interrupt,
213						  irq_flags, up->port.name, i);
214				if (ret < 0)
215					serial_do_unlink(i, up);
216			}
```

- 動手操作及觀察：註冊 serial port 的中斷
  - serial_link_irq_chain->request_irq->request_thread_irq
  - 在 request_thread_irq 中可以用 gdb 印出相關訊息如

下

```
(gdb) p irq
$20 = 4
(gdb) p handler
$21 = (irq_handler_t) 0xffffffff81a016b2
<serial8250_interrupt>
(gdb) p devname
$22 = 0xffff88800f349540 "ttyS0"
```

截圖 5.　b common_interrupt

- 目的：所有的中斷服務函數都會「跳到」這段組合語言，他的主要功能是將所有的暫存器儲存下來，然後呼叫 do_IRQ，從 do_IRQ 開始就是 C 語言。
- 動手操作及觀察：可以在 common_interrupt 印出這個「外部中斷的『編號』」

```
Breakpoint 13, common_interrupt () at arch/x86/entry/entry_64.S:580
580		addq	$-0x80, (%rsp)			/* Adjust vector to [-256, -1] range */
(gdb) p *((int *) $rsp)
$24 = 88   //請特別注意，數字 88，16 進位碼是 58
```

- 動手操作及觀察：do_IRQ 使用 orig_ax 取得該中斷向量的 C 處理函數，此處理函數包裹在資料結構「irq_desc」中。orig_ax 是在 irq_entries 放入堆疊中（請參考「截圖 7.」）

```
/*
do_IRQ handles all normal device IRQ's (the special
SMP cross-CPU interrupts have their own specific
```

```
handlers).
*/
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    struct irq_desc * desc;
    /* high bit used in ret_from_ code   */
    //提取外部中斷的編號
    unsigned vector = ~regs->orig_ax;

    entering_irq();

    /* entering_irq() tells RCU that we're not quiescent.   Check it. */
    RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");
    //提取外部中斷的相關資料結構，尤其是函數指標
    desc = __this_cpu_read(vector_irq[vector]);
    //開始真正的處理中斷
    if (!handle_irq(desc, regs)) {
        ack_APIC_irq();
        if (desc != VECTOR_RETRIGGERED) {
            pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
            __func__, smp_processor_id(),
            vector);
        } else {
            __this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
        }
    }
    exiting_irq();
    set_irq_regs(old_regs);
    return 1;
}
```

截圖 6. b serial8250_interrupt

- 目的：如果這個裝置會發出中斷，那麼這樣的函數就是開發驅動程式的人必須撰寫的「其中一部分」。此部分稱之為 top halve，由 common_ingterrupt->do_IRQ 開始一層層呼叫，直到此函數

- 動手操作及觀察：使用 bt 可以發現 serial8250_interrupt（serial port、UART）的驅動程式是由 common_interrupt 一路呼叫下去

```
(gdb) bt
#0  serial8250_interrupt (irq=4351, dev_id=0x1 <irq_stack_union+1>) at drivers/tty/serial/8250/8250_core.c:108
#1  0xffffffff811bbe3e in __handle_irq_event_percpu (desc=0xffff88800f02b000, flags=0xffff88800f603d08) at kernel/irq/handle.c:149
#2  0xffffffff811bbfcf in handle_irq_event_percpu (desc=0xffff88800f02b000) at kernel/irq/handle.c:189
#3  0xffffffff811bc075 in handle_irq_event (desc=0xffff88800f02b000) at kernel/irq/handle.c:206
#4  0xffffffff811c42a8 in handle_edge_irq (desc=0xffff88800f02b000) at kernel/irq/chip.c:791
#5  0xffffffff81043c66 in generic_handle_irq_desc (desc=0xffff88800f02b000) at ./include/linux/irqdesc.h:154
#6  0xffffffff81043f64 in handle_irq (desc=0xffff88800f02b000, regs=0xffff88800f603dd8) at arch/x86/kernel/irq_64.c:78
#7  0xffffffff822016c1 in do_IRQ (regs=0xffff88800f603dd8) at arch/x86/kernel/irq.c:246
#8  0xffffffff8220094f in common_interrupt () at arch/x86/entry/entry_64.S:583
#9  0xffff88800f603dd8 in ?? ()
#10 0xffffffff8220094a in common_interrupt () at arch/x86/entry/entry_64.S:581
#11 0x0000000000000000 in ?? ()
```

截圖 7. disass irq_entries_start

- 目的：這裡列的程式碼就是中斷向量表（interrupt vector table 或 interrupt descript table）所指向的程式碼。
- 動手操作及觀察：仔細看一下程式碼是否都很類似？將「編號」放到堆疊以後，就 jump 到 common_interrupt

```
(gdb) disass irq_entries_start
Dump of assembler code for function irq_entries_start:
```

繳交的檔案：

```
0xffffffff82200210 <+0>:      pushq   $0x5f
0xffffffff82200212 <+2>:      jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff82200217 <+7>:      nop
0xffffffff82200218 <+8>:      pushq   $0x5e
0xffffffff8220021a <+10>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff8220021f <+15>:     nop
0xffffffff82200220 <+16>:     pushq   $0x5d
0xffffffff82200222 <+18>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff82200227 <+23>:     nop
0xffffffff82200228 <+24>:     pushq   $0x5c
0xffffffff8220022a <+26>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff8220022f <+31>:     nop
0xffffffff82200230 <+32>:     pushq   $0x5b
0xffffffff82200232 <+34>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff82200237 <+39>:     nop
0xffffffff82200238 <+40>:     pushq   $0x5a
0xffffffff8220023a <+42>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff8220023f <+47>:     nop
0xffffffff82200240 <+48>:     pushq   $0x59
0xffffffff82200242 <+50>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff82200247 <+55>:     nop
//從上往下數，是第八號組合語言
0xffffffff82200248 <+56>:     pushq   $0x58
0xffffffff8220024a <+58>:     jmpq    0xffffffff82200940 <common_interrupt>
0xffffffff8220024f <+63>:     nop
```

截圖 8.　b *(irq_entries_start+56)

- 目的：從中斷的進入點，一直追蹤到 do_IRQ
- 動手操作及觀察：不斷的輸入「si」(step into next instruction) 直到 do_IRQ

問題1.　一份簡單的報告，請將題目所說的八個中斷點予以截圖

問題2.　在（問題 1.）的報告中，說明 Linux 如何設定中斷向量

- 提示（請增加至少十個字）：

1. CPU 內建的中斷。CPU 內建的「中斷事件」，也稱作「software interrupt」或「trap」。Linux 在 start_kernel 中，先呼叫 trap_init，將 CPU 內部中斷的中斷處理函數寫入到「中斷向量表」（interrupt vector table）
2. 外部中斷的部分，由 init_IRQ 一路呼叫到 set_intr_gate，外部中斷的 ISR 的程式碼如下

```
Dump of assembler code for function irq_entries_start:
   0xffffffff82200210 <+0>:      pushq   $0x5f
   0xffffffff82200212 <+2>:      jmpq    0xffffffff82200940 <common_interrupt>
   0xffffffff82200217 <+7>:      nop
   0xffffffff82200218 <+8>:      pushq   $0x5e
   0xffffffff8220021a <+10>:     jmpq    0xffffffff82200940 <common_interrupt>
   0xffffffff8220021f <+15>:     nop
```

問題3.　在（問題 1.）的報告中，說明 Linux 如何從中斷向量的組合語言部分（interrupt service routine，這裡只討論外部中斷）跳躍到特定的中斷函數

- 提示（請增加至少十個字）：

1. 以 serial port 的中斷為例，他是第四號外部中斷（可以從 /proc/irq/4 中得知），程式碼如下

```
   0xffffffff82200248 <+56>:     pushq   $0x58
```

```
    0xffffffff8220024a <+58>:     jmpq    0xffffffff82200940
<common_interrupt>
    0xffffffff8220024f <+63>:     nop
```

特別要注意的是，第四號中斷在程式碼中到底是 irq_entries_start 中的第 X 號組合語言並非一對一的對應，例如在這個例子中是「第八號組合語言」。Linux 對這個中斷的「軟體編號」是「0x58」

2. common_interrupt 的程式碼如下

```
common_interrupt:
addq   $-0x80, (%rsp)              /* Adjust vector to [-256, -1] range */
call    interrupt_entry
UNWIND_HINT_REGS indirect=1
call    do_IRQ        /* rdi points to pt_regs */
```

其中 call interrupt_entry 的目的是將所有暫存器放入到堆疊，這部分的重點是『製造堆疊』，堆疊內的資料型態為「pt_regs」，並且將『中斷的「軟體編號」(即「0x58」)』放到 pr_regs 的 orig_ax

```
/*
 * Interrupt entry helper function.
 *
 * Entry runs with interrupts off. Stack layout at entry:
 * +----------------------------------------------------+
 * | regs->ss                                           |
 * | regs->rsp                                          |
 * | regs->eflags                                       |
 * | regs->cs                                           |
 * | regs->ip                                           |
 * +----------------------------------------------------+
 * | regs->orig_ax = ~(interrupt number)                |
```

```
 * +-------------------------------------------------------+
 * | return address                                        |
 * +-------------------------------------------------------+
 */
ENTRY(interrupt_entry)
        UNWIND_HINT_FUNC
        ASM_CLAC
        cld
        testb   $3, CS-ORIG_RAX+8(%rsp)
        jz      1f
        SWAPGS
        /*
         * Switch to the thread stack. The IRET frame and orig_ax are
         * on the stack, as well as the return address. RDI..R12 are
         * not (yet) on the stack and space has not (yet) been
         * allocated for them.
         */
        pushq   %rdi
        /* Need to switch before accessing the thread stack. */
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rdi
        movq    %rsp, %rdi
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp
          /*
          * We have RDI, return address, and orig_ax on the stack on
          * top of the IRET frame. That means offset=24
          */
        UNWIND_HINT_IRET_REGS base=%rdi offset=24
        pushq   7*8(%rdi)               /* regs->ss */
        pushq   6*8(%rdi)               /* regs->rsp */
        pushq   5*8(%rdi)               /* regs->eflags */
        pushq   4*8(%rdi)               /* regs->cs */
        pushq   3*8(%rdi)               /* regs->ip */
        pushq   2*8(%rdi)               /* regs->orig_ax */
        pushq   8(%rdi)                 /* return address */
        UNWIND_HINT_FUNC
        movq    (%rdi), %rdi
1:
        PUSH_AND_CLEAR_REGS save_ret=1
```

```
        ENCODE_FRAME_POINTER 8

        testb   $3, CS+8(%rsp)

        jz      1f

        /*

         * IRQ from user mode.

         *

         * We need to tell lockdep that IRQs are off.  We can't do this until

         * we fix gsbase, and we should do it before enter_from_user_mode

         * (which can take locks).  Since TRACE_IRQS_OFF is idempotent,

         * the simplest way to handle it is to just call it twice if

         * we enter from user mode.  There's no reason to optimize this since

         * TRACE_IRQS_OFF is a no-op if lockdep is off.

         */

        TRACE_IRQS_OFF

        CALL_enter_from_user_mode

1:

        ENTER_IRQ_STACK old_rsp=%rdi save_ret=1

        /* We entered an interrupt context - irqs are off: */

        TRACE_IRQS_OFF

        ret

END(interrupt_entry)
```

其中 do_IRQ 為 C 語言，do_IRQ 的程式如下，這部分的重點是：由於 orig_ax 放的是「中斷的軟體編號」，因此將這個編號作為「中斷向量物件」的索引，即 __this_cpu_read(vector_irq[vector])，並在 handle_irq 中呼叫該函數。在這個例子中的函數即 serial8250_interrupt。約略等同於下列程式碼

desc = __this_cpu_read(vector_irq[vector]);

desc->action->handler(…);

其中 handler 為函數指標，即：serial8250_interrupt

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
	struct pt_regs *old_regs = set_irq_regs(regs);
	struct irq_desc * desc;
	/* high bit used in ret_from_ code  */
	unsigned vector = ~regs->orig_ax;
	entering_irq();
	/* entering_irq() tells RCU that we're not quiescent.  Check it. */
	RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");
	desc = __this_cpu_read(vector_irq[vector]);
	if (!handle_irq(desc, regs)) {
		ack_APIC_irq();
		if (desc != VECTOR_RETRIGGERED) {
			pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
				__func__, smp_processor_id(), vector);
		} else {
			__this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
		}
	}
	exiting_irq();
	set_irq_regs(old_regs);
	return 1;
}
```

其他：

1. 報告格式

   甲、必須是 pdf 檔案，裡面放入八張截圖，並回答問題（問題 2.）及（問題 3.）

   乙、報告的名稱為：hw2.pdf

丙、學號、姓名（請隱藏個人資訊，例如：學號 687410007，姓名：羅 X 五）

2. 繳交期限：請參考課程網頁

3. 如果真的不會寫，記得去請教朋友。在你的報告上寫你請教了誰即可。

附一：

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.