獨孤派作業系統
期中，上機考

中正大學 作業系統實驗室

指導教授：羅習五

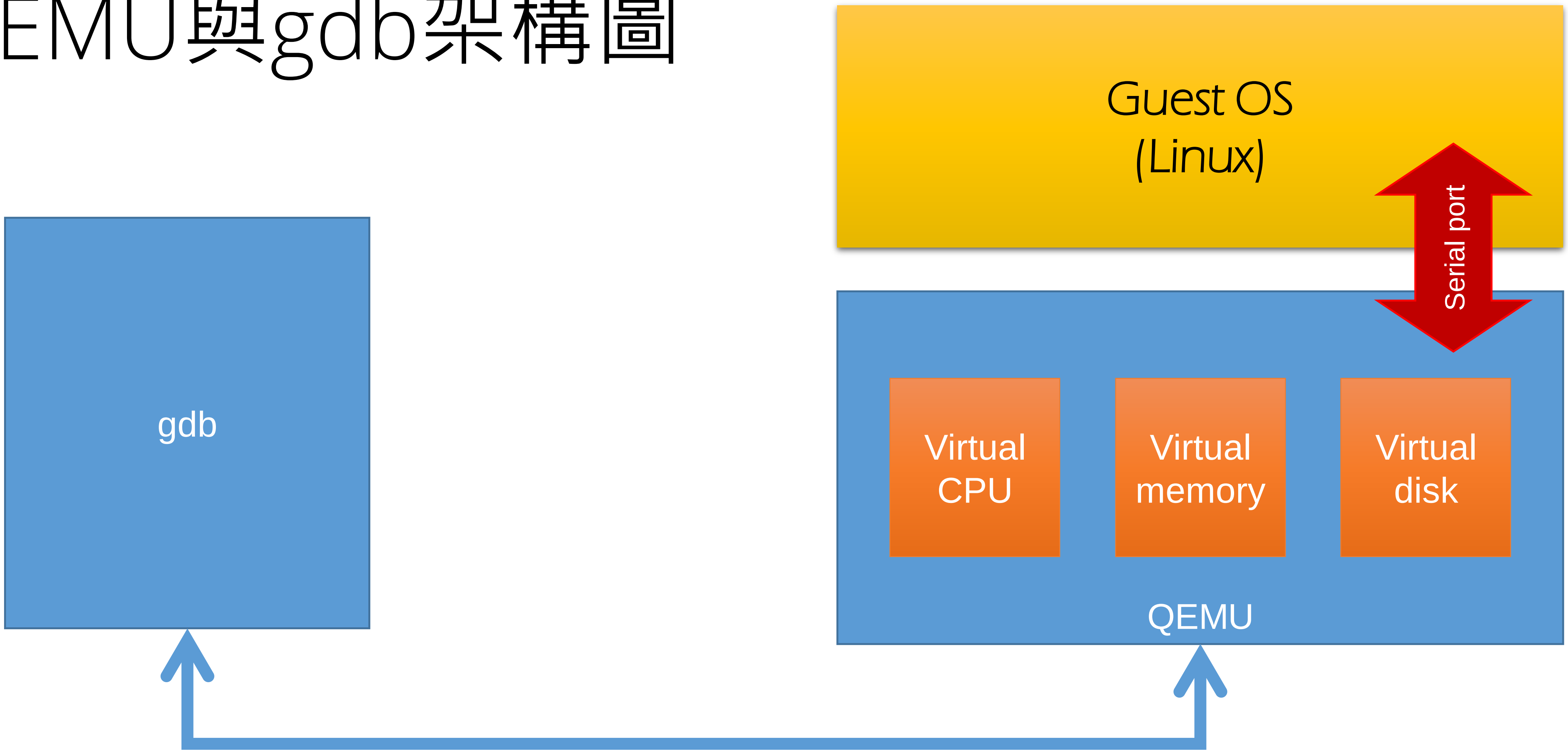
下載已經做好的qemu相關檔案

- 自dropbox下載home_shiwulo-qemu+eclipse+oracleJava.tar.xz
- 解壓縮
  - sudo tar Jxvf home_shiwulo-qemu+eclipse+oracleJava.tar.xz
  - 注意，這個地方要用sudo，因為壓縮檔中有特殊檔案，需要用mknode產生
  - 注意，解壓縮以後，這些檔案的owner和group都是「1000」，如果你的id不是1000請使用chown及chgrp修改

目錄內容介紹

- 所有相關檔案都放在/home/yourname/qemu-linux-4.0底下
- 『dbg-qemu.sh』使用qemu執行Linux，qemu會等待debugger連入
- 『vmlinux』是帶有debug資訊的Linux的核心
- 『bzImage、 initramfs_data.cpio.gz 』啟動用的Linux kernel及initramfs
- 『system.map』使用『nm』自『vmlinux』匯出所有的symbol的可讀檔案

QEMU與gdb架構圖



很特別，這個地方是走TCP/IP，因此可以（如果需要的話）可以遠端除錯

開始本次上機考

啟動debuggee

- $ ./ dbg-qemu.sh
- 啟動以後，QEMU會停住，等待gdb的連線

啟動debugger

- 另外開一個terminal，輸入
- $ gdb ./vmlinux
- gdb會讀取vmlinux的symbol table，進入gdb後...
- (gdb) target remote localhost:666

第一部分，評分方式

1. （15pt）請問你是用懶人包，或者自行設定trace kernel的環境？
   a) （15pt）完全自行設定
   b) （10pt）使用/home/shiwulo的懶人包
   c) （5pt）使用virtual machine懶人包
2. （10pt）設定breakpoint於start_kernel，並印出start_kernel的call stack，及call stack中各個函數的位址
3. （10pt）找出中斷向量表的開始位址
4. （10pt）找出system call的進入點
5. （10pt）印出CPU初始化的時候，「除以零」這個中斷的interrupt service routine的位址

2. start_kernel

這個號碼就是第#題的答案

#0 start_kernel () at init/main.c:490

#1 0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200

#2 0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189

#3 0x0000000000000000 in ?? ()

3. IDT進入點（即中斷向量表開始位址）

```
(gdb) p idt_table
$3 = 0xffffffff825fb000 <idt_table>
(gdb) bt
#0  _set_gate (gate=<optimized out>, type=<optimized out>, addr=0xffffffff81d0f850 <divide_error>, dpl=<optimized out>, ist=<optimized out>, seg=<optimized out>) at ./arch/x86/include/asm/desc.h:371
#1  0xffffffff822df55c in trap_init () at arch/x86/kernel/traps.c:956
#2  0xffffffff822dc07b in start_kernel () at init/main.c:550
#3  0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200
#4  0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189
#5  0x0000000000000000 in ?? ()
(gdb) p/x *(struct gate_struct64*)idt_table
$6 = {offset_low = 0xb000, segment = 0x10, ist = 0x0, zero0 = 0x0, type = 0xe, dpl = 0x0, p = 0x1, offset_middle = 0x822d, offset_high = 0xffffffff, zero1 = 0x0}
```

TSS (cpu0)

```
(gdb) p t
$4 = (struct tss_struct *) 0xffff88000fa10ec0
(gdb) p *t
$5 = {x86_tss = {reserved1 = 0, sp0 = 18446744071597735936, sp1 = 0, sp2 = 0, reserved2 = 0, ist = {0, 0, 0, 0, 0, 0, 0}, reserved3 = 0, reserved4 = 0, reserved5 = 0, io_bitmap_base = _bitmap = {0 <repeats 1025 times>}, stack = {0 <repeats 64 times>}}
(gdb) bt
#0  cpu_init () at arch/x86/kernel/cpu
#1  0xffffffff822dfb80 in trap_init () at arch/x86/kernel/traps.c:1006
#2  0xffffffff822dc07b in start_kernel () at init/main.c:550
#3  0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200
#4  0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189
#5  0x0000000000000000 in ?? ()
```

這個值很重要，每個CPU有自己的TSS

0xFFFFFFFF82204000

每個task有自己的stack即sp0

idt descriptor

```
(gdb) p/x idt_descr
$13 = {size = 0xfff, address = 0xffffffffff57b000}
(gdb) bt
#0  load_current_idt () at ./arch/x86/include/asm/desc.h:513
#1  0xffffffff810230d8 in cpu_init () at arch/x86/kernel/cpu/common.c:1333
#2  0xffffffff822dfb80 in trap_init () at arch/x86/kernel/traps.c:1006
#3  0xffffffff822dc07b in start_kernel () at init/main.c:550
#4  0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200
#5  0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189
#6  0x0000000000000000 in ?? ()
```

4. sytem call

```
(gdb) p system_call
$15 = {<text variable, no debug info>} 0xffffffff81d0e100 <system_call>
(gdb) bt
#0  syscall_init () at arch/x86/kernel/cpu/common.c:1174
#1  0xffffffff810230f9 in cpu_init () at arch/x86/kernel/cpu/common.c:1336
#2  0xffffffff822dfb80 in trap_init () at arch/x86/kernel/traps.c:1006
#3  0xffffffff822dc07b in start_kernel () at init/main.c:550
#4  0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200
#5  0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189
#6  0x0000000000000000 in ?? ()
```

interrupt stack

```
(gdb) p/x *t
$21 = {x86_tss = {reserved1 = 0x0, sp0 = 0xffffffff82204000, sp1 = 0x0, sp2 = 0x0, reserved2 = 0x0, ist = {0xffff88000fa06000, 0xffff88000fa07000, 0xffff88000fa09000, 0xffff88000fa0a000, 0x0, 0x0, 0x0}, reserved3 = 0x0, reserved4 = 0x0, reserved5 = 0x0, io_bitmap_base = 0x0}, io_bitmap = {0x0 <repeats 1025 times>}, stack = {0x0 <repeats 64 times>}}
(gdb) bt
#0  cpu_init () at arch/x86/kernel/cpu/common.c:1360
#1  0xffffffff822dfb80 in trap_init () at arch/x86/kernel/traps.c:1006
#2  0xffffffff822dc07b in start_kernel () at init/main.c:550
#3  0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200
#4  0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189
#5  0x0000000000000000 in ?? ()
```

interrupt stack

(gdb) p/x *(struct tss_struct*) 0xffff88000fa10ec0

$13 = {x86_tss = {reserved1 = 0x0, sp0 = 0xffffffff82204000, sp1 = 0x0, sp2 = 0x0, reserved2 = 0x0, ist = {0xffff88000fa06000, 0xffff88000fa07000, 0xffff88000fa09000, 0xffff88000fa0a000, 0x0, 0x0, 0x0}, reserved3 = 0x0, reserved4 = 0x0, reserved5 = 0x0, io_bitmap_base = 0x80}, io_bitmap = {0xffffffffffffffff <repeats 1025 times>}, stack = {0x0 <repeats 64 times>}}

(gdb) bt

#0 0xffffffff81004416 in __switch_to (prev_p=0xffffffff8220e180 <init_task>, next_p=0xffff88000ecc0910) at arch/x86/kernel/process_64.c:280

#1 0xffffffff810c8e52 in context_switch (rq=0x0 <irq_stack_union>, prev=0x0 <irq_stack_union>, next=0x0 <irq_stack_union>) at kernel/sched/core.c:2317

(gdb) p/x *(struct gate_struct64*) 0xffffffff825fb000

$14 = {offset_low = 0xf850, segment = 0x10, ist = 0x0, zero0 = 0x0, type = 0xe, dpl = 0x0, p = 0x1, offset_middle = 0x81d0, offset_high = 0xffffffff, zero1 = 0x0}

5. 印出第零號中斷 即「除以零」

「除以零的ISR位址」ffffffff,81d0,f850

(gdb) p/x *(struct gate_struct64*) 0xffffffff825fb000

$14 = {offset_low = 0xf850, segment = 0x10, ist = 0x0, zero0 = 0x0, type = 0xe, dpl = 0x0, p = 0x1, offset_middle = 0x81d0, offset_high = 0xffffffff, zero1 = 0x0}

資料結構的定義請見下一頁

gate_struct64 /arch/x86/include/asm/desc_defs.h

```c
/* 16byte gate */
struct gate_struct64 {
      u16 offset_low;
      u16 segment;
      unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
      u16 offset_middle;
      u32 offset_high;
      u32 zero1;
} __attribute__((packed));
```

system call所用的kernel stack位置 每一個task會有自己的kernel stack

(gdb) p/x thread->sp0

$24 = 0xffffffff82204000

(gdb) bt

跟第12頁的值一樣
0xFFFFFFFF82204000

#0 r … 0ec0, thread=0xffffffff8220e778 <init_task+1528>)
at ./a … 58

#1 0 … xffff88000fa10ec0, thread=0xffffffff8220e778 <init_task+1528>)
at ./a … 76

#2 0xffffffff810232a3 in cpu_init () at arch/x86/kernel/cpu/common.c:1374

#3 0xffffffff822dfb80 in trap_init () at arch/x86/kernel/traps.c:1006

#4 0xffffffff822dc07b in start_kernel () at init/main.c:550

#5 0xffffffff822db72d in x86_64_start_reservations (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:200

#6 0xffffffff822db6f0 in x86_64_start_kernel (real_mode_data=0x13f60 <runqueues+1440> <error: Cannot access memory at address 0x13f60>) at arch/x86/kernel/head64.c:189

#7 0x0000000000000000 in ?? ()

第二部分，評分方式

6. （30pt）於「執行」 divideByZero時，使用gdb對中斷處理常式（interrupt service routine，ISR的位址）進行除錯
   a) （10pt）印出當下正在執行的ISR的位址
   b) （10pt）印出ISR的程式碼（組合語言）
   c) （10pt）印出該ISR所使用的堆疊的位址，請問此時ISR所使用的stack是從哪邊載入
7. （15pt）（延續問題5.a）請解釋ISR組合語言的意義
   a) （10pt）error_entry
   b) （5pt） do_divide_error

b divide_error

(gdb) # 按下ctr-c

(gdb) b divide_error

Breakpoint 12 at 0xffffffff81d0f850: file arch/x86/kernel/entry_64.S, line 1020.

(gdb) c

在QEMU中

/ # ./divideByZero

6.a、6.b 第一次攔截「除以零」ISR如下所示

```
(gdb) disassemble /m divide_error
Dump of assembler code for function divide_error:
1143    idtentry divide_error do_divide_error has_error_code=0
   0xffffffff81d0f853 <+3>:     pushq  $0xffffffffffffffff
   0xffffffff81d0f855 <+5>:     sub    $0x78,%rsp
   0xffffffff81d0f859 <+9>:     callq  0xffffffff81d0fd20 <error_entry>
   0xffffffff81d0f85e <+14>:    mov    %rsp,%rdi
   0xffffffff81d0f861 <+17>:    xor    %esi,%esi
   0xffffffff81d0f863 <+19>:    callq  0xffffffff8100688f <do_divide_error>
   0xffffffff81d0f868 <+24>:    jmpq   0xffffffff81d0fdd0 <error_exit>
   0xffffffff81d0f86d:  nopl   (%rax)
   0xffffffff81d0f870 <+0>:     data16 xchg %ax,%ax

End of assembler dump.
```

第一次攔截「除以零」印出暫存器

```
(gdb) info registers
rax            0xa      10
rbx            0x400400 4195328
rcx            0x44b9c0 4504000
rdx            0x0      0
rsi            0x7fff252c40a8
rdi            0x1      1
rbp            0x7fff252c3f80
rsp            0xffff88000002bfd8
r8             0x1000000 16777216
r9             0x6ba8e0 7055584
r10            0x15     21
r11            0x0      0
r12            0x401900 4200704
r13            0x0      0
r14            0x6b8018 7045144
r15            0x0      0
rip            0xffffffff81d0f850
<divide_error>
eflags         0x46     [ PF ZF ]
cs             0x10     16
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x63     99
gs             0x0      0
```

6.c 第二次攔截「除以零」

```
(gdb) info registers
rax            0xa       10
rbx            0x400400  4195328
rcx            0x44b9c0  4504000
rdx            0x0       0
rsi            0x7fff7fba26b8
rdi            0x1       1
rbp            0x7fff7fba2590
rsp            0xffff88000e59bfd8
r8             0x1000000 16777216
r9             0x6ba8e0  7055584
r10            0x15      21
r11            0x0       0
r12            0x401900  4200704
r13            0x0       0
r14            0x6b8018  7045144
r15            0x0       0
rip            0xffffffff81d0f850
<divide_error>
eflags         0x46      [ PF ZF ]
cs             0x10      16
                         0
                         0
es             0x0       0
fs             0x63      99
gs             0x0       0
```

rsp不一樣是因為每個task
有自己的kernel stack

6.c 第二次攔截「除以零」觀察TSS的結構

0xffff88000e59bfd8 - 0xffff88000e59c000 = 0x28

「除以零」和「system call」都屬於exception，在x86中堆疊會自動切換到sp0

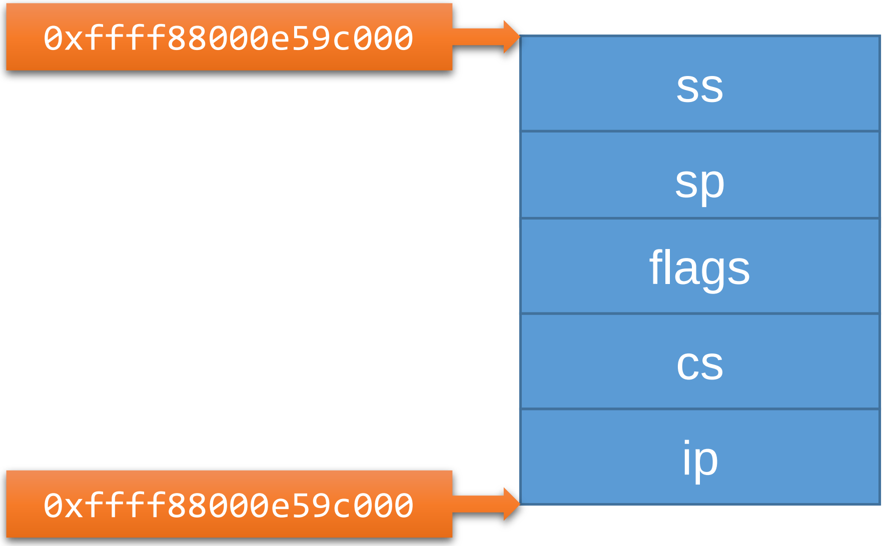
(gdb) p/x *(struct tss_struct *) 0xffff88000fa10ec0

$15 = {x86_tss = {reserved1 = 0x0, sp0 = 0xffff88000e59c000, sp1 = 0x0, sp2 = 0x0, reserved2 = 0x0, ist = {0xffff88000fa06000, 0xffff88000fa07000, 0xffff88000fa09000, 0xffff88000fa0a000, 0x0, 0x0, 0x0}, reserved3 = 0x0, reserved4 = 0x0, reserved5 = 0x0, io_bitmap_base = 0x80}, io_bitmap = {0xffffffffffffffff <repeats 1025 times>}, stack = {0x0 <repeats 64 times>}}

7.a 關於x86-64如何處理exception的文章

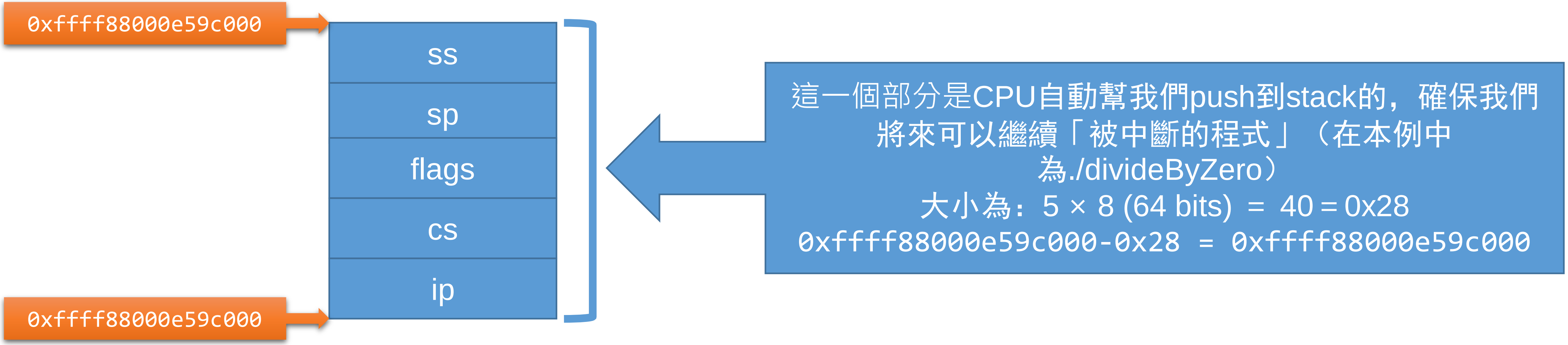
- **Aligning the stack pointer**: An interrupt can occur at any instructions, so the stack pointer can have any value, too. However, some CPU instructions (e.g. some SSE instructions) require that the stack pointer is aligned on a 16 byte boundary, therefore the CPU performs such an alignment right after the interrupt.
- **Switching stacks** (in some cases): A stack switch occurs when the CPU privilege level changes, for example when a CPU exception occurs in an user mode program. It is also possible to configure stack switches for specific interrupts using the so-called *Interrupt Stack Table* (described in the next post).
- **Pushing the old stack pointer**: The CPU pushes the values of the stack pointer (rsp) and the stack segment (ss) registers at the time when the interrupt occured (before the alignment). This makes it possible to restore the original stack pointer when returning from an interrupt handler.
- **Pushing and updating the RFLAGS register**: The RFLAGS register contains various control and status bits. On interrupt entry, the CPU changes some bits and pushes the old value.
- **Pushing the instruction pointer**: Before jumping to the interrupt handler function, the CPU pushes the instruction pointer (rip) and the code segment (cs). This is comparable to the return address push of a normal function call.
- **Pushing an error code** (for some exceptions): For some specific exceptions such as page faults, the CPU pushes an error code, which describes the cause of the exception.
- **Invoking the interrupt handler**: The CPU reads the address and the segment descriptor of the interrupt handler function from the corresponding field in the IDT. It then invokes this handler by loading the values into the rip and cs registers.

7.a 堆疊的宣告




```
struct pt_regs {
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long bp;
        unsigned long bx;
    /* arguments: non interrupts/non tracing syscalls only save up to
here*/
        unsigned long r11;
        unsigned long r10;
        unsigned long r9;
        unsigned long r8;
        unsigned long ax;
        unsigned long cx;
        unsigned long dx;
        unsigned long si;
        unsigned long di;
        unsigned long orig_ax;
    /* end of arguments */
    /* cpu exception frame or undefined */
        unsigned long ip;
        unsigned long cs;
        unsigned long flags;
        unsigned long sp;
        unsigned long ss;
    /* top of stack page */
};
```

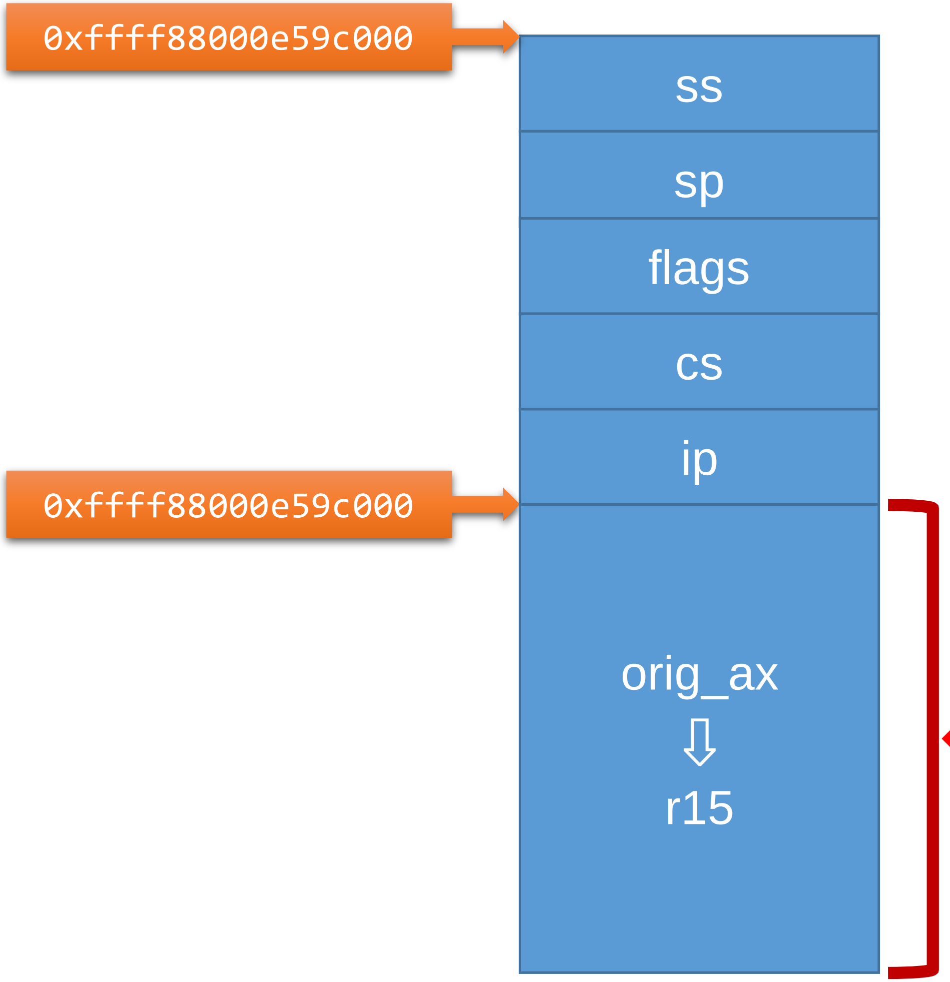
7.a 堆疊的宣告


0xffff88000e59c000
ss
sp
flags
cs
ip
0xffff88000e59c000
這一個部分是CPU自動幫我們push到stack的，確保我們將來可以繼續「被中斷的程式」（在本例中為./divideByZero）
大小為：5 × 8 (64 bits) ＝ 40＝0x28
0xffff88000e59c000-0x28 = 0xffff88000e59c000

7.a 堆疊的宣告

```
pushq  $0xffffffffffffffff
sub    $0x78,%rsp
callq  0xffffffff81d0fd20 <error_entry>
mov    %rsp,%rdi
xor    %esi,%esi
callq  0xffffffff8100688f <do_divide_error>
jmpq   0xffffffff81d0fdd0 <error_exit>
nopl   (%rax)
data16 xchg %ax,%ax
```

如果是system call這邊會放進去AX，exception則放「錯誤代碼」



紅色的組合語言會在堆疊中放入這一個部分

```
cld
mov      %rdi,0x78(%rsp)
mov      %rsi,0x70(%rsp)
mov      %rdx,0x68(%rsp)
mov      %rcx,0x60(%rsp)
mov      %rax,0x58(%rsp)
mov      %r8,0x50(%rsp)
mov      %r9,0x48(%rsp)
mov      %r10,0x40(%rsp)
mov      %r11,0x38(%rsp)
mov      %rbx,0x30(%rsp)
mov      %rbp,0x28(%rsp)
mov      %r12,0x20(%rsp)
mov      %r13,0x18(%rsp)
mov      %r14,0x10(%rsp)
mov      %r15,0x8(%rsp)
```

見下頁解釋

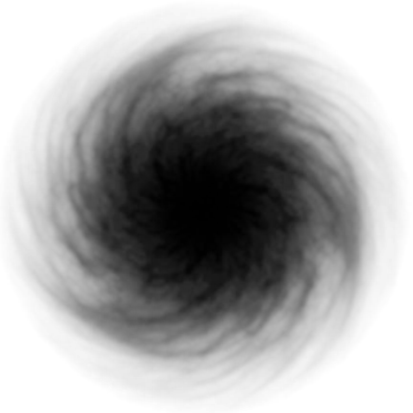
7.a x86指令，cld的意義

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI).

7.b 自行解釋「 do_divide_error 」的部分

其實我在後面有寫...自己再稍加整理就是答案了

傳送門
至33頁



繳交方式

- 請照著這份投影片，將所有的題目做過一次，每個考題都要「截圖」或者用「手機拍照」
- 繳交期限11/21（星期三），晚上11:59:59
- 繳交方式：助教將在星期六之前公布

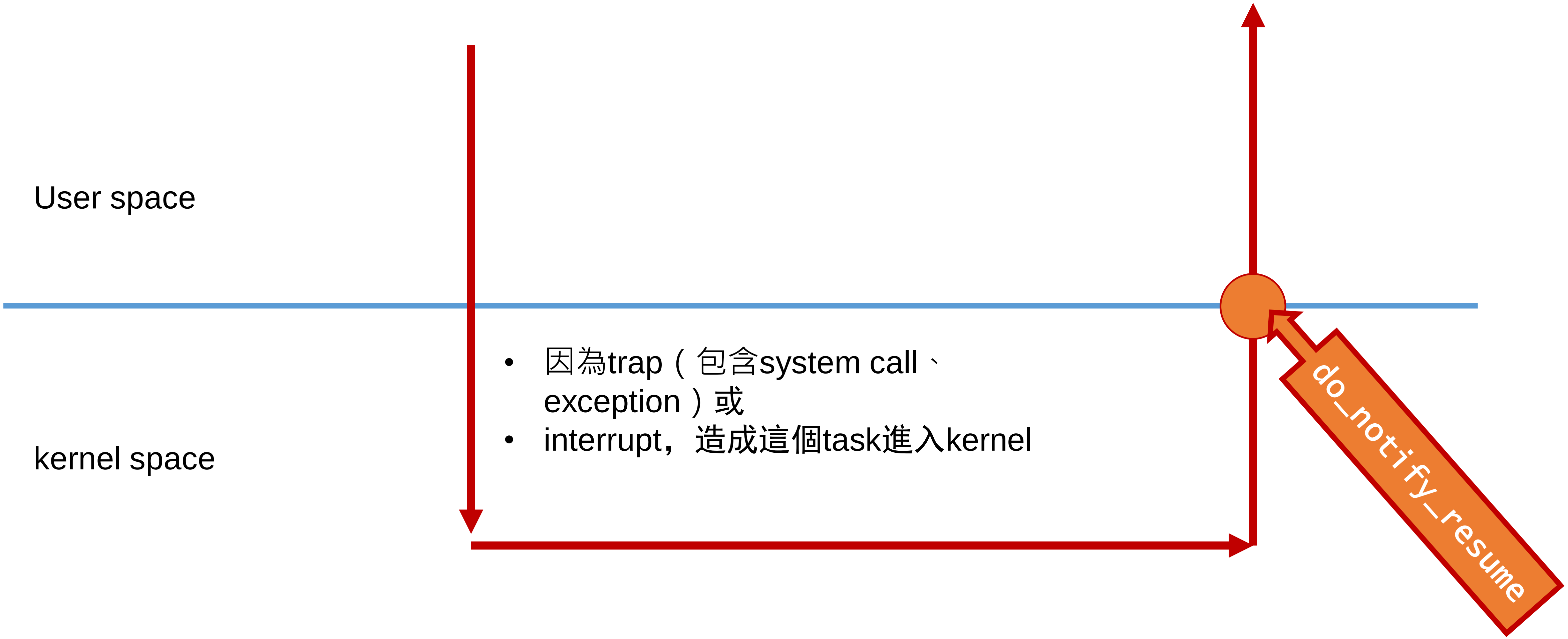
補充教材：
將「除以零」轉成signal的方式

7.b 通知，在task_struct內設定「發生了signal」

divide_error ➔ do_divide_error ➔ do_error_trap

```
do_error_trap
        fill_trap_info
        do_trap
                unhandled_signal
                 force_sig_info
                        specific_send_sig_info
                                __send_signal（在這個地方會設定flag）
```

7.b 示意圖


User space
kernel space
• 因為trap（包含system call、exception）或
• interrupt，造成這個task進入kernel
do_notify_resume

do_notify_resmue的呼叫點

- ENTRY(native_iret)
  - 從中斷返回時呼叫do_notify_resmue
- ENTRY(system_call)
  - 從system call返回時呼叫do_notify_resmue
- error_exit
  - Divide by zero，最後就是「跳到」error_exit

ENTRY(native_iret) /arch/x86/kernel/entry_64.S

```
ENTRY(native_iret)
/*...*/
retint_signal:
testl $_TIF_DO_NOTIFY_MASK,%edx
jz     retint_swapgs
TRACE_IRQS_ON
ENABLE_INTERRUPTS(CLBR_NONE)
SAVE_REST
movq $-1,ORIG_RAX(%rsp)
xorl %esi,%esi          # oldset
movq %rsp,%rdi          # &pt_regs
call do_notify_resume
RESTORE_REST
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
GET_THREAD_INFO(%rcx)
jmp retint_with_reschedule
/*...*/
```

7.b ENTRY(error_exit)
error_exit ➔ retint_careful

```
retint_careful:
CFI_RESTORE_STATE
bt     $TIF_NEED_RESCHED,%edx
jnc    retint_signal
TRACE_IRQS_ON
ENABLE_INTERRUPTS(CLBR_NONE)
pushq_cfi %rdi
SCHEDULE_USER
popq_cfi %rdi
GET_THREAD_INFO(%rcx)
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
jmp retint_check

retint_signal:
testl $_TIF_DO_NOTIFY_MASK,%edx
jz     retint_swapgs
TRACE_IRQS_ON
ENABLE_INTERRUPTS(CLBR_NONE)
SAVE_REST
movq $-1,ORIG_RAX(%rsp)
xorl %esi,%esi         # oldset
movq %rsp,%rdi         # &pt_regs
call do_notify_resume
RESTORE_REST
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
GET_THREAD_INFO(%rcx)
jmp retint_with_reschedule
```

ENTRY(system_call) /arch/x86/kernel/entry_64.S

```
ENTRY(system_call)
/* Either reschedule or signal or syscall exit tracking needed. */
/* First do a reschedule test. */
/* edx:    work, edi: workmask */
/*...*/
int_signal:
testl $_TIF_DO_NOTIFY_MASK,%edx
jz 1f
movq %rsp,%rdi          # &ptregs -> arg1
xorl %esi,%esi          # oldset -> arg2
call do_notify_resume
1:    movl $_TIF_WORK_MASK,%edi
/*...*/
```

```
/*  /arch/x86/include/asm/thread_info.h  */
/* Only used for 64 bit */
#define _TIF_DO_NOTIFY_MASK \
(_TIF_SIGPENDING |
_TIF_NOTIFY_RESUME | \
_TIF_USER_RETURN_NOTIFY |
_TIF_UPROBE)
```

分析「do_notify_resume」
do_notify_resume➔do_signal➔get_signal

(gdb) p ksig->info.si_signo

$1 = 8 # 8) SIGFPE

(gdb) bt

#0 get_signal (ksig=0xffff88000e5afe50) at kernel/signal.c:2358

#1 0xffffffff81005349 in do_signal (regs=0xffff88000e5aff58) at arch/x86/kernel/signal.c:703

#2 0xffffffff81005984 in do_notify_resume (regs=<optimized out>, unused=<optimized out>, thread_info_flags=<optimized out>) at arch/x86/kernel/signal.c:748

#3 <signal handler called>

附錄：關於x86硬體簡介

認識x86

- x86處理器，起始於1978年，8086的定址空間只有16bit（65K），大部分的電腦只有4～16K的記憶體
- 程式主要使用組合語言，例如：著名的DOS（Microsoft的作業系統）
- 為了方便程式設計師撰寫程式，組合語言的功能非常的強大，例如：有專門給字串處理用的組語
- 在記憶體方面也給程式設計師很大的方便，例如：
  - cs:0x800 ;代表程式區段第800道指令
  - ds:0x700 ;代表資料區段第800個位置的地方
  - ss:0x600 ;代表堆疊裡面第600號的位置

認識x86

- 下面位置中，cs、ds、ss稱之為區段，segment
  - cs:0x800 ;代表程式區段第800道指令
  - ds:0x700 ;代表資料區段第800個位置的地方
  - ss:0x600 ;代表堆疊裡面第600號的位置

segment

- segment是一個古老的設計，但x86-64傳承了自8086起的這個古老設計
- 在x86-64原則上將segment剔除，但是還是會看到
    - Segment selector（一個數字，用來選擇目前要用哪一個Segment descriptor）
    - Segment descriptor（一個資料結構，主要記載segment的開始位置、大小、存取的屬性）

x86-64中斷向量表

IA-32e system level registers

紅色部分是我們會用到的部分

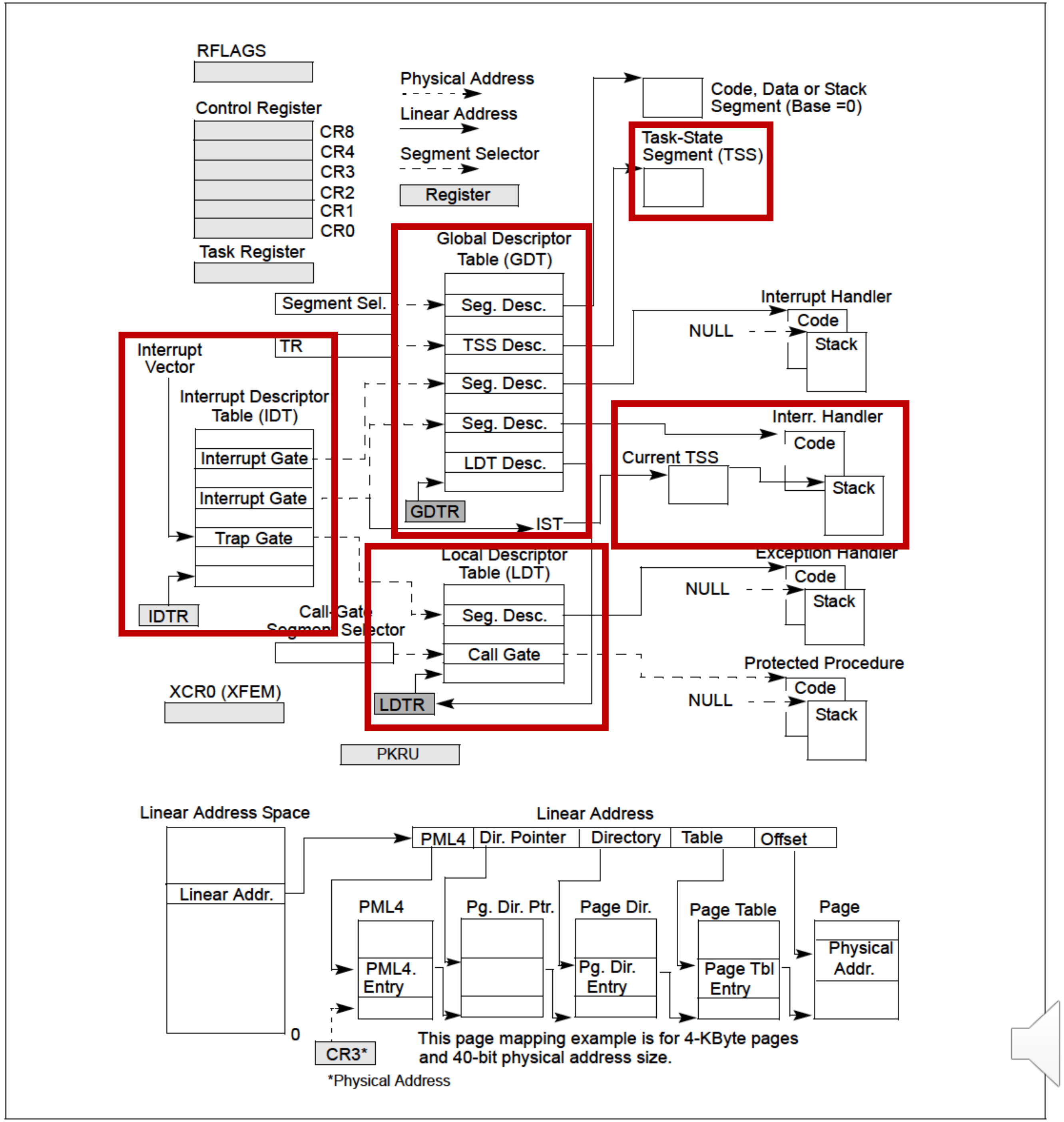


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

IA-32e system level registers

- 藍色部分就是segment descriptor，系統有很多segment descriptor，以陣列的方式存在
- 在x86-64中有三個segment descriptor「陣列」，分別用IDT LDTR、GDTR指向這三個陣列
- segment selector就是index用來指向LDTR、GDTR，而IDTR陣列，就是中斷向量表

嚴格來說IDT內部放的東西應該要叫做gate，但其樣式跟seg. descriptor很像

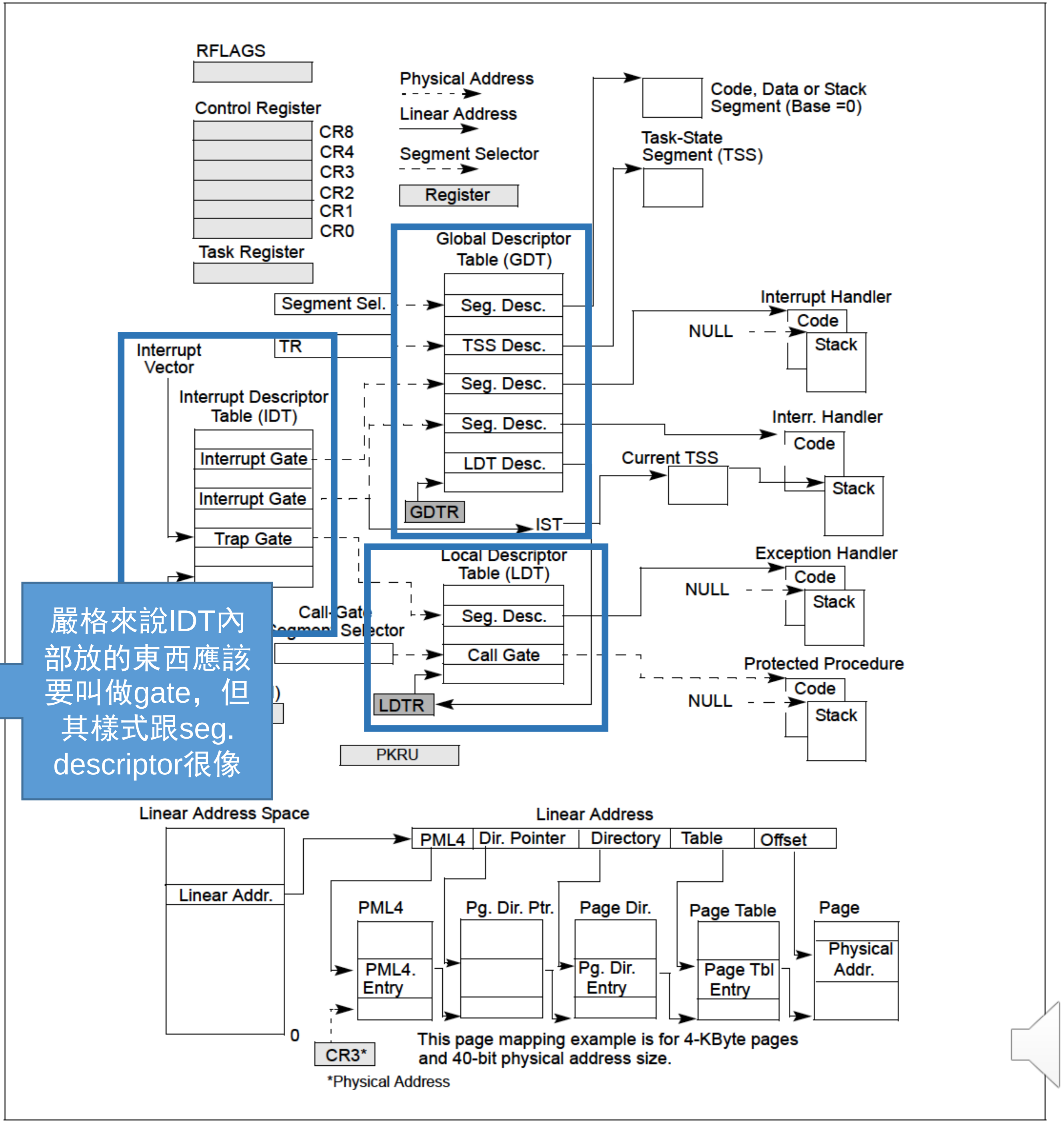


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

IDT Gate Descriptor

中斷向量表的其中一個entry，大小為128 bits，紅色的部分共64 bits指向ISR

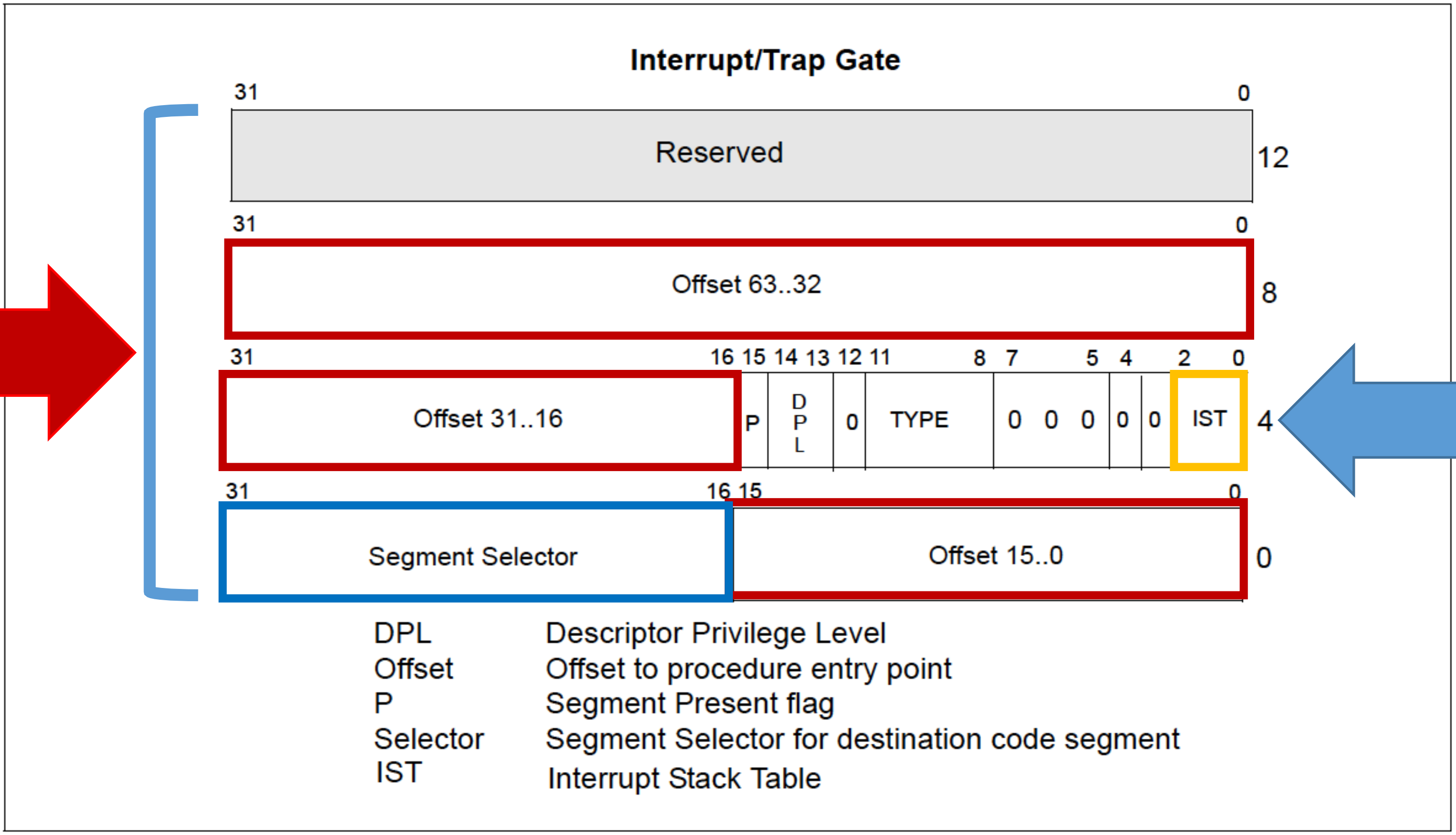


Figure 6-7. 64-Bit IDT Gate Descriptors

IST，值介於1~7，可以讓OS選擇發生中斷的時候要切換到哪一個stack

ISR：interrupt server routine

Hardware-generated exceptions

INT_NUM	Short Description PM
0x00	**Division by zero**
0x01	Single-step interrupt (see trap flag)
0x02	NMI
0x03	Breakpoint (callable by the special 1-byte instruction 0xCC, used by debuggers)
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor Segment Overrun *(386 or earlier only)*
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Fault
0x0D	General protection fault
0x0E	Page fault
0x0F	*reserved*
0x10	Math Fault
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception

/arch/x86/include/asm/traps.h

```
enum {
        X86_TRAP_DE = 0,    /*  0, Divide-by-zero */
        X86_TRAP_DB,        /*  1, Debug */
        X86_TRAP_NMI,        /*  2, Non-maskable Interrupt */
        X86_TRAP_BP,        /*  3, Breakpoint */
        X86_TRAP_OF,        /*  4, Overflow */
        X86_TRAP_BR,        /*  5, Bound Range Exceeded */
        X86_TRAP_UD,        /*  6, Invalid Opcode */
        X86_TRAP_NM,        /*  7, Device Not Available */
        X86_TRAP_DF,        /*  8, Double Fault */
        X86_TRAP_OLD_MF,    /*  9, Coprocessor Segment Overrun */
        X86_TRAP_TS,        /* 10, Invalid TSS */
        X86_TRAP_NP,        /* 11, Segment Not Present */
        X86_TRAP_SS,        /* 12, Stack Segment Fault */
        X86_TRAP_GP,        /* 13, General Protection Fault */
        X86_TRAP_PF,        /* 14, Page Fault */
        X86_TRAP_SPURIOUS,    /* 15, Spurious Interrupt */
        X86_TRAP_MF,        /* 16, x87 Floating-Point Exception */
        X86_TRAP_AC,        /* 17, Alignment Check */
        X86_TRAP_MC,        /* 18, Machine Check */
        X86_TRAP_XF,        /* 19, SIMD Floating-Point Exception */
        X86_TRAP_IRET = 32,    /* 32, IRET Exception */
};
```

對IDT進一步解釋

先解釋segment

gate_struct64 /arch/x86/include/asm/desc_defs.h

```
/* 16byte gate */
struct gate_struct64 {
        u16 offset_low;
        u16 segment;
        unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
        u16 offset_middle;
        u32 offset_high;
        u32 zero1;
} __attribute__((packed));
```

segments in x64

In 64-bit mode: CS, DS, ES, SS are treated as if each **segment base is 0**, regardless of the value of the associated segment descriptor base. This creates a **flat address space for code, data, and stack**. **FS and GS are exceptions**. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures).

segments in x64

- 大致上意思是說：
  - segment原則上沒有用處
  - GS和FS可能會用到

segment descriptor

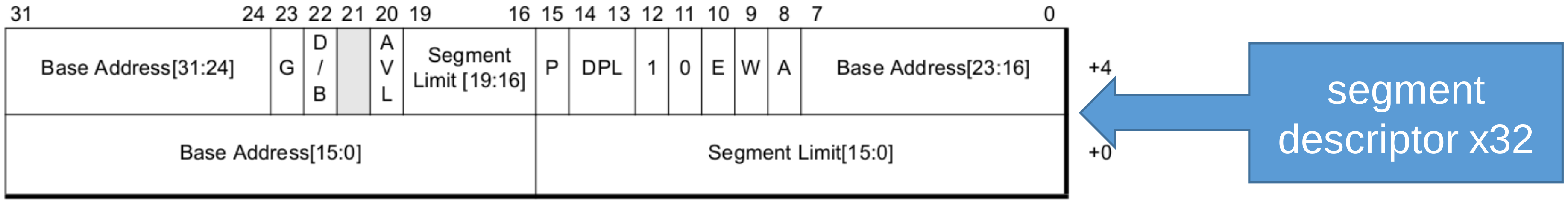


Figure 4-15. Data-Segment Descriptor—Legacy Mode

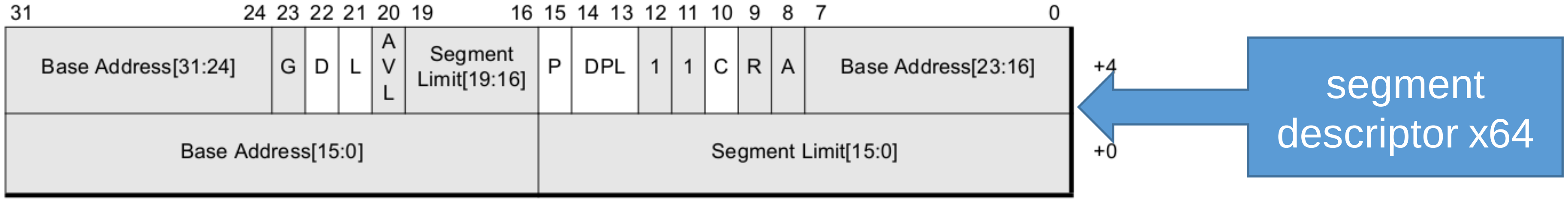


Figure 4-20. Code-Segment Descriptor—Long Mode

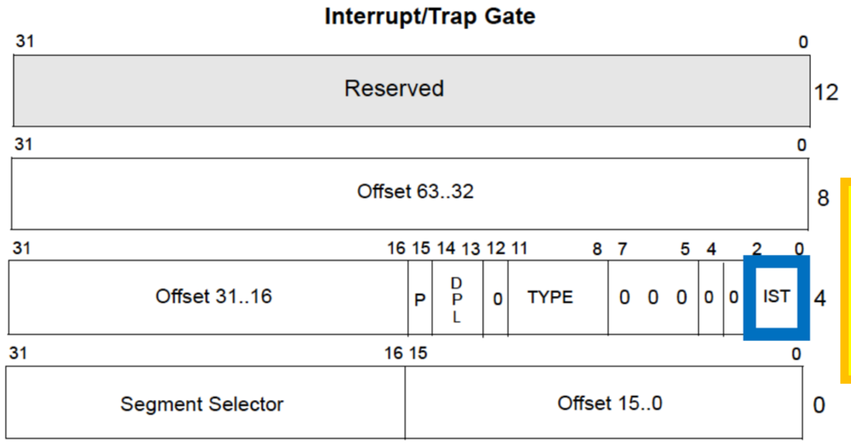
desc_struct /arch/x86/include/asm/desc_defs.h

segment descriptor in x64 mode

```
struct desc_struct {
    union {
        struct {
            unsigned int a;
            unsigned int b;
        };
        struct {
            u16 limit0;
            u16 base0;
            unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
            unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
        };
    };
} __attribute__((packed));
```

對IDT進一步解釋
再解釋IST

Task state segment (tss)



IST 3是Linux中ISR使用到的stack位置

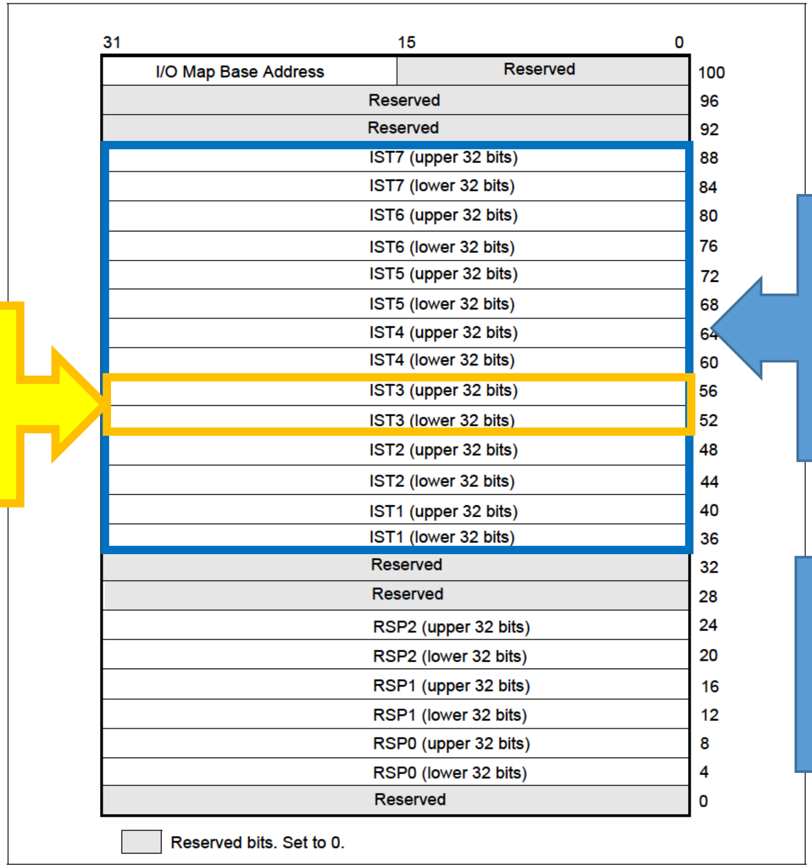


Figure 7-11. 64-Bit TSS Format

「IST」記錄在TSS七種stack的選擇

藉此，Linux給中斷一個獨立stack

tss_struct
/arch/x86/include/asm/processor.h

```c
struct tss_struct {
	/*
	 * The hardware state:
	 */
	struct x86_hw_tss	x86_tss;
	unsigned long		io_bitmap[IO_BITMAP_LONGS + 1];
	unsigned long		stack[64];
} ____cacheline_aligned;
```

tss : initialnization
/arch/x86/include/asm/processor.h

```
/*
 * per-CPU TSS segments. Threads are completely 'soft' on Linux,
 * no more per-task TSS's. The TSS size is kept cacheline-aligned
 * so they are allowed to end up in the .data..cacheline_aligned
 * section. Since TSS's are completely CPU-local, we want them
 * on exact cacheline boundaries, to eliminate cacheline ping-pong.
 */
__visible DEFINE_PER_CPU_SHARED_ALIGNED(struct tss_struct, init_tss) = INIT_TSS;
```

tss : initialnization
/arch/x86/include/asm/processor.h

```
#define INIT_TSS {                                          \
    .x86_tss = {                                            \
        .sp0         = sizeof(init_stack) + (long)&init_stack, \
        .ss0         = __KERNEL_DS,                         \
        .ss1         = __KERNEL_CS,                         \
        .io_bitmap_base = INVALID_IO_BITMAP_OFFSET,         \
    },                                                      \
    .io_bitmap       = { [0 ... IO_BITMAP_LONGS] = ~0 },    \
}
```

tss in ctw_sw

/arch/x86/kernel/process_64.c

```
__visible __notrace_funcgraph struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
        int cpu = smp_processor_id();
        struct tss_struct *tss = &per_cpu(init_tss, cpu);
/*...*/
        if (unlikely(task_thread_info(next_p)->flags & _TIF_WORK_CTXSW_NEXT ||
                task_thread_info(prev_p)->flags & _TIF_WORK_CTXSW_PREV))
                __switch_to_xtra(prev_p, next_p, tss);
/*...*/
}
```

tss in ctw_sw
/arch/x86/kernel/process.c

```c
void __switch_to_xtra(struct task_struct *prev_p, struct task_struct *next_p,
              struct tss_struct *tss)
/*...*/
if (test_tsk_thread_flag(next_p, TIF_IO_BITMAP)) {
            memcpy(tss->io_bitmap, next->io_bitmap_ptr,
             max(prev->io_bitmap_max, next->io_bitmap_max));
      } else if (test_tsk_thread_flag(prev_p, TIF_IO_BITMAP)) {
            memset(tss->io_bitmap, 0xff, prev->io_bitmap_max);
      }
/*...*/
}
```

trap_init

- Call cpu_init() to do:
- initialize per-CPU state
- reload the GDT and IDT
- mask off the eflags NT (Nested Task) bit
- set up and load the per-CPU TSS and LDT
- clear 6 debug registers (0, 1, 2, 3, 6, and 7)
- stts(): set the 0x08 bit (TS: Task Switched) in CR0 to enable lazy
- register saves on context switches

trap_init ➜ cpu_init /arch/x86/kernel/cpu/common.c

```c
void cpu_init(void) {
        me = current;
        syscall_init();
        barrier();
        //set up and load the per-CPU TSS
        if (!oist->ist[0]) {
            char *estacks = per_cpu(exception_stacks, cpu);

            for (v = 0; v < N_EXCEPTION_STACKS; v++) {
                estacks += exception_stack_sizes[v];
                oist->ist[v] = t->x86_tss.ist[v] =
                        (unsigned long)estacks;
                if (v == DEBUG_STACK-1)
                    per_cpu(debug_stack_addr, cpu) = (unsigned long)estacks;
            }
        }
        t->x86_tss.io_bitmap_base = offsetof(struct tss_struct, io_bitmap);
}
```

syscall & sysret

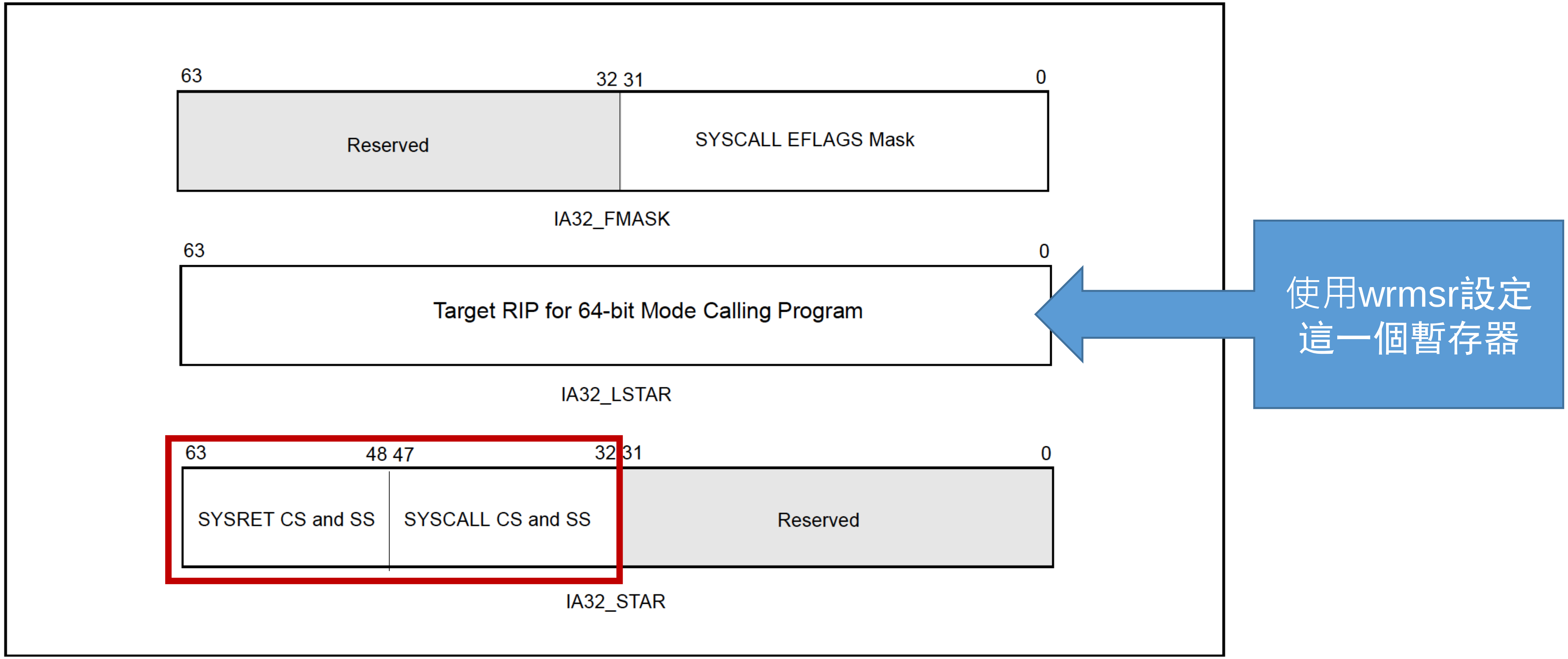


Figure 5-14. MSRs Used by SYSCALL and SYSRET

trap_init ➜ cpu_init ➜ syscall_init
/arch/x86/kernel/cpu/common.c

```c
void syscall_init(void)
{
        wrmsrl(MSR_STAR,  ((u64)__USER32_CS)<<48  | ((u64)__KERNEL_CS)<<32);
        wrmsrl(MSR_LSTAR, system_call);
        wrmsrl(MSR_CSTAR, ignore_sysret);
        /* Flags to clear on syscall */
        wrmsrl(MSR_SYSCALL_MASK,
                X86_EFLAGS_TF|X86_EFLAGS_DF|X86_EFLAGS_IF|
                X86_EFLAGS_IOPL|X86_EFLAGS_AC|X86_EFLAGS_NT);
}
```