https://play.google.com/store/apps/details?id=jp.co.namura.magnifier&hl=zh_HK
https://www.pngfind.com/mpng/iRhTJTh_tux-linux-logo-start-menu-linux-icons-hd/

作業二：
觀察中斷

中正大學 作業系統實驗室
指導教授： 羅習五

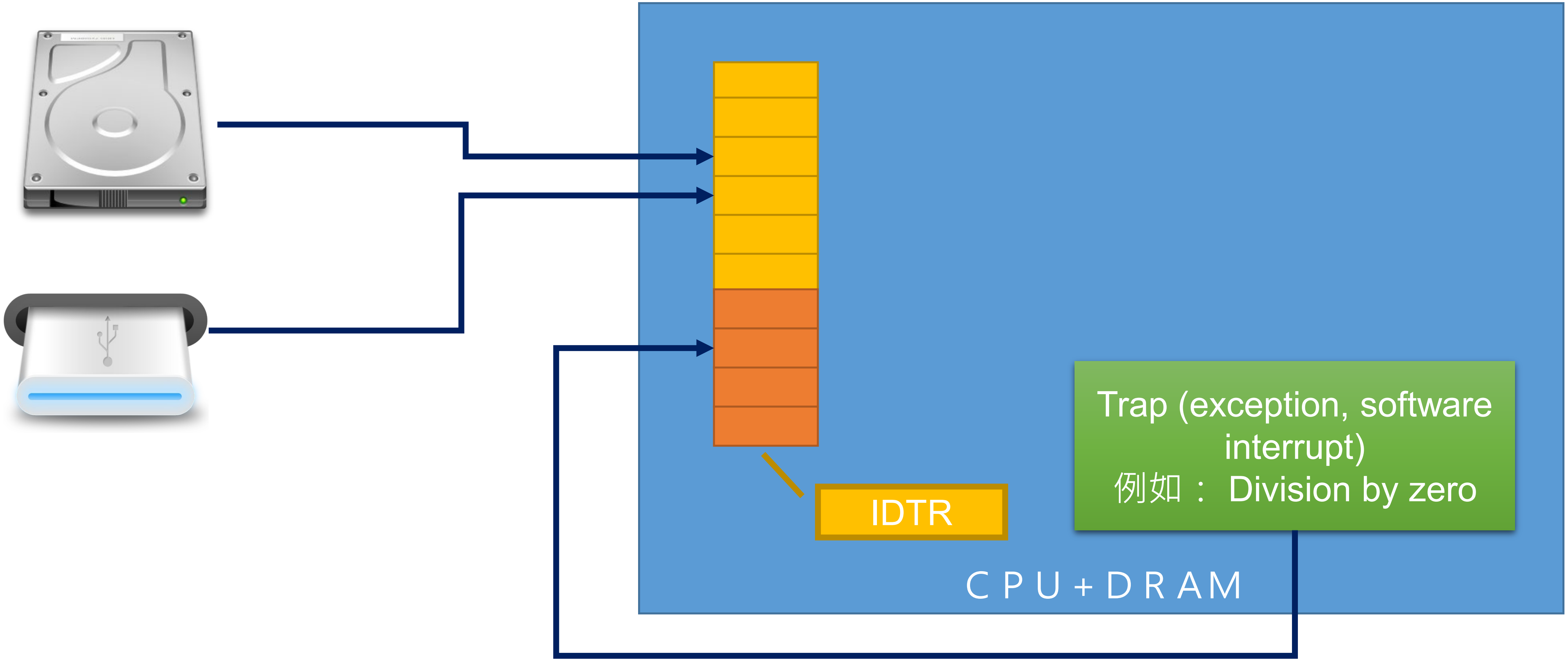
圖片來源

- 新垣結衣
  - https://makey.asia/column.php?id=532
  - http://pic.haibao.com/image/14284778.html?kw=%E6%96%B0%E5%9E%A3%E7%BB%93%E8%A1%A3
  - https://huaban.com/pins/835412722/

作業目標及負責助教

- 作業目標：
  - 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
- 負責助教：
  - 請看網頁

從硬體看x86的中斷

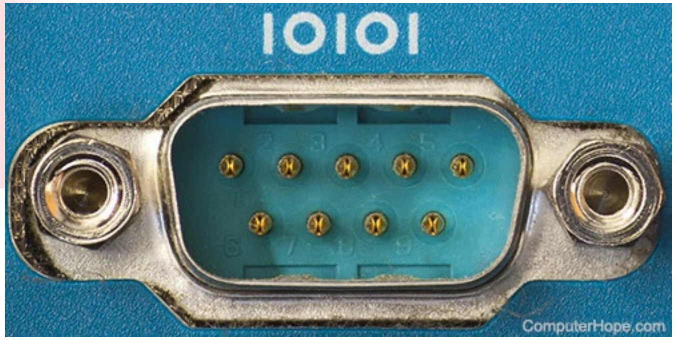

IDTR
Trap (exception, software interrupt)
例如： Division by zero
CPU+DRAM

從「軟體」看x86的中斷

1. 怎麼樣向OS註冊中斷?

2. 中斷向量的運作流程?

1. 怎麼樣向OS註冊中斷?

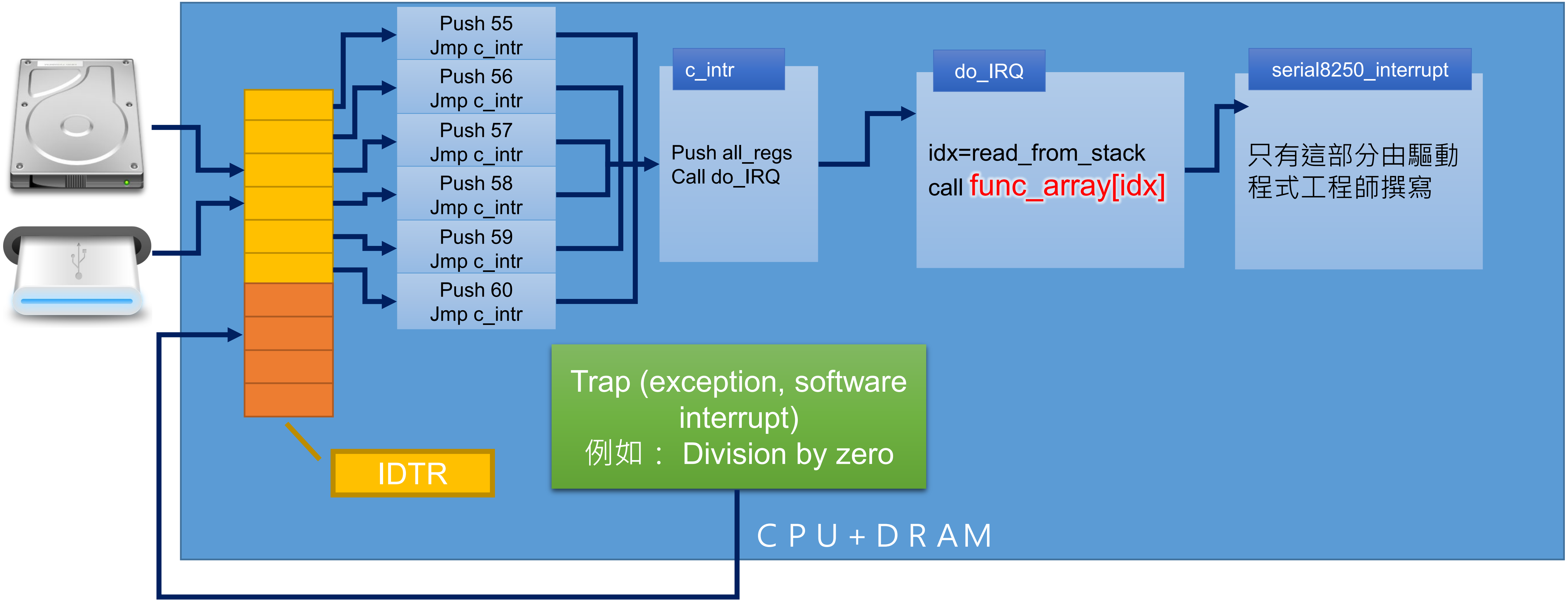


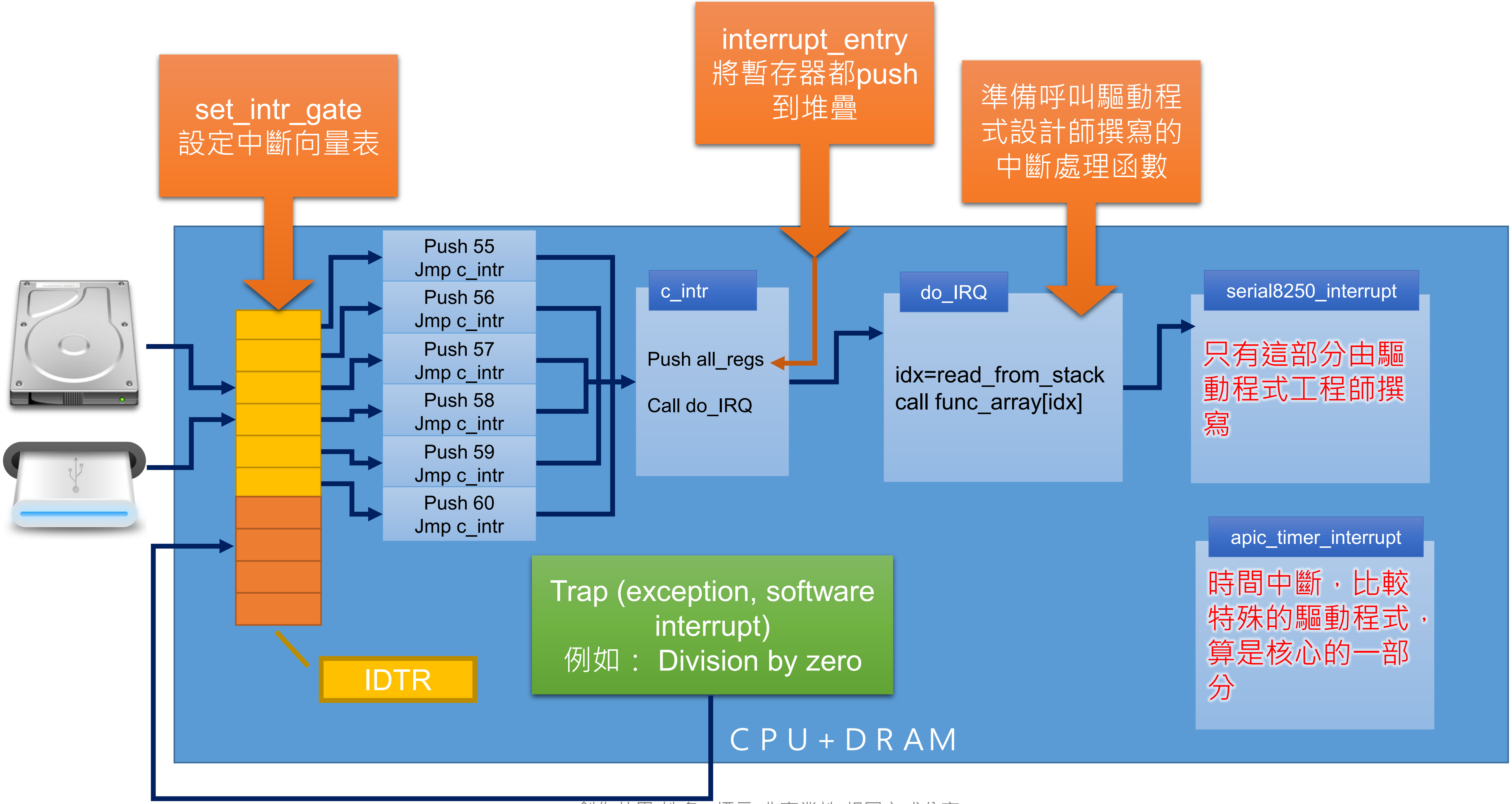
- API：request_IRQ
  - 這部分的投影片都以serial8250為例，這是很簡單的驅動程式
  - 軟硬體都很簡單，幾乎不會出錯，因此常常用來做OS的除錯工具
  - ret = request_irq(up->port.irq, serial8250_interrupt,
                                        irq_flags, up->port.name, i);
  - 在這個例子中，最重要的參數是 serial8250_interrupt ，請注意這是一個C 函數

2. 中斷向量的運作流程?

- 對驅動程式設計師而言，只要硬體發生中斷，那麼OS就會呼叫「 serial8250_interrupt 」，大部分的驅動程式都是用「C」撰寫而成
- 幾乎沒人用組合語言，組合語言的部分、CPU直接相關的部分，都由Linux代為處理

從OS開發者看「註冊中斷」





set_intr_gate
設定中斷向量表

interrupt_entry
將暫存器都push
到堆疊

準備呼叫驅動程
式設計師撰寫的
中斷處理函數

Push 55
Jmp c_intr

Push 56
Jmp c_intr

Push 57
Jmp c_intr

Push 58
Jmp c_intr

Push 59
Jmp c_intr

Push 60
Jmp c_intr

c_intr

Push all_regs

Call do_IRQ

do_IRQ

idx=read_from_stack
call func_array[idx]

serial8250_interrupt

只有這部分由驅
動程式工程師撰
寫

apic_timer_interrupt

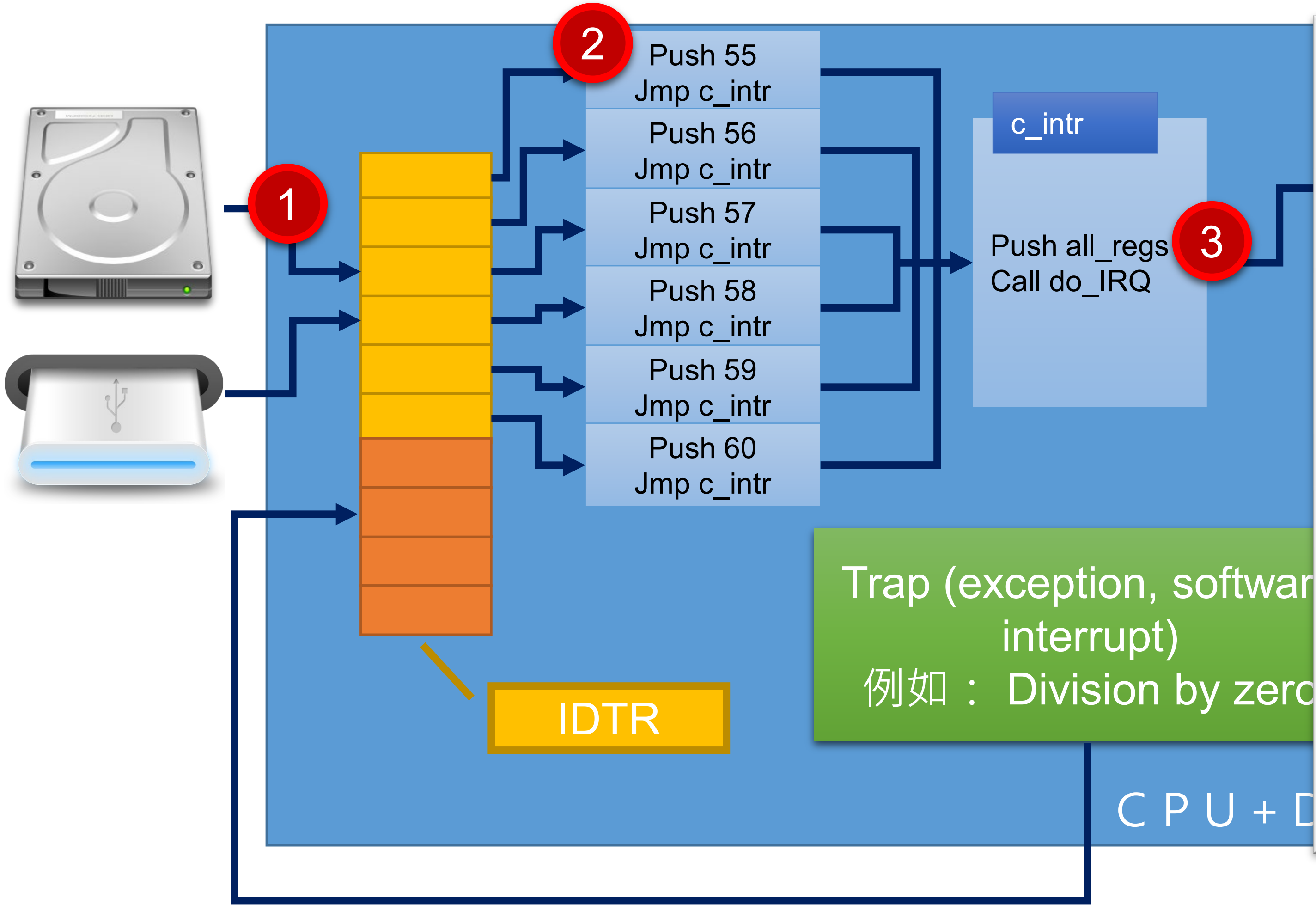
時間中斷，比較
特殊的驅動程式，
算是核心的一部
分

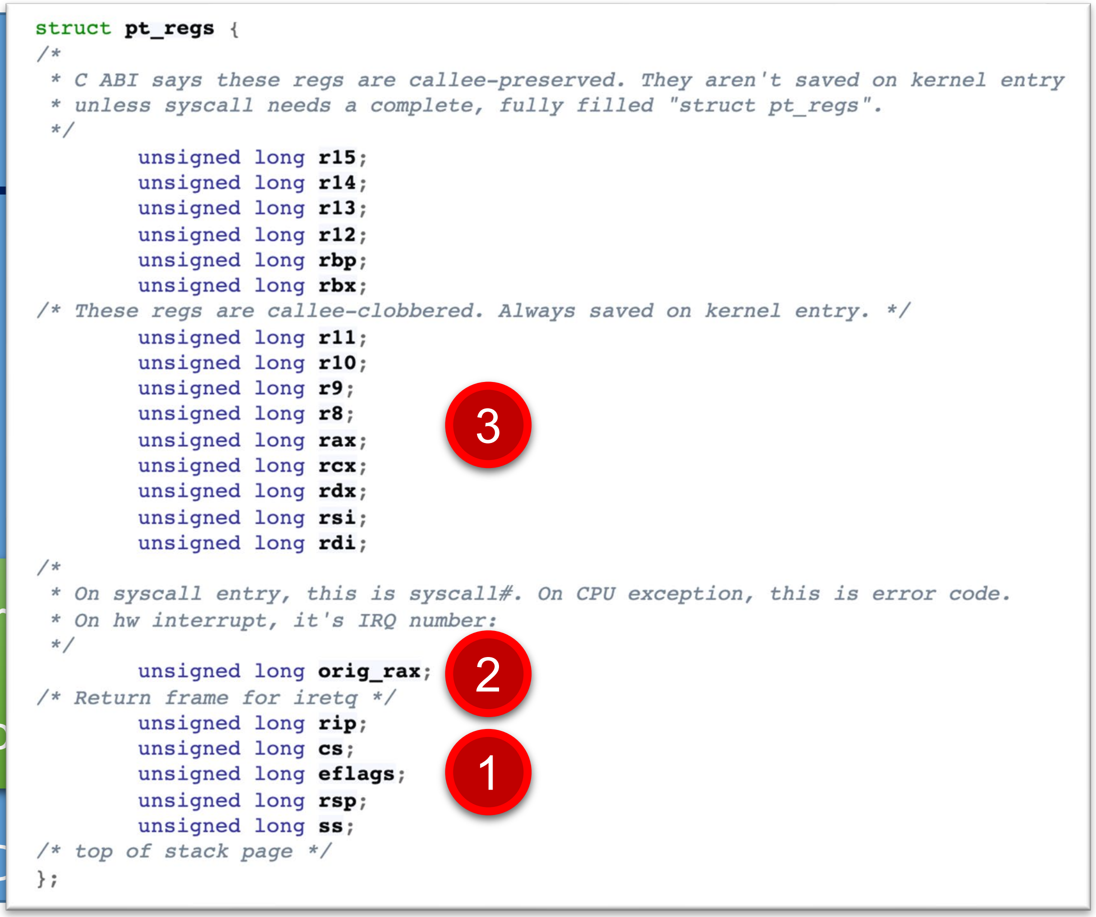
IDTR

Trap (exception, software
interrupt)
例如： Division by zero

C P U + D R A M

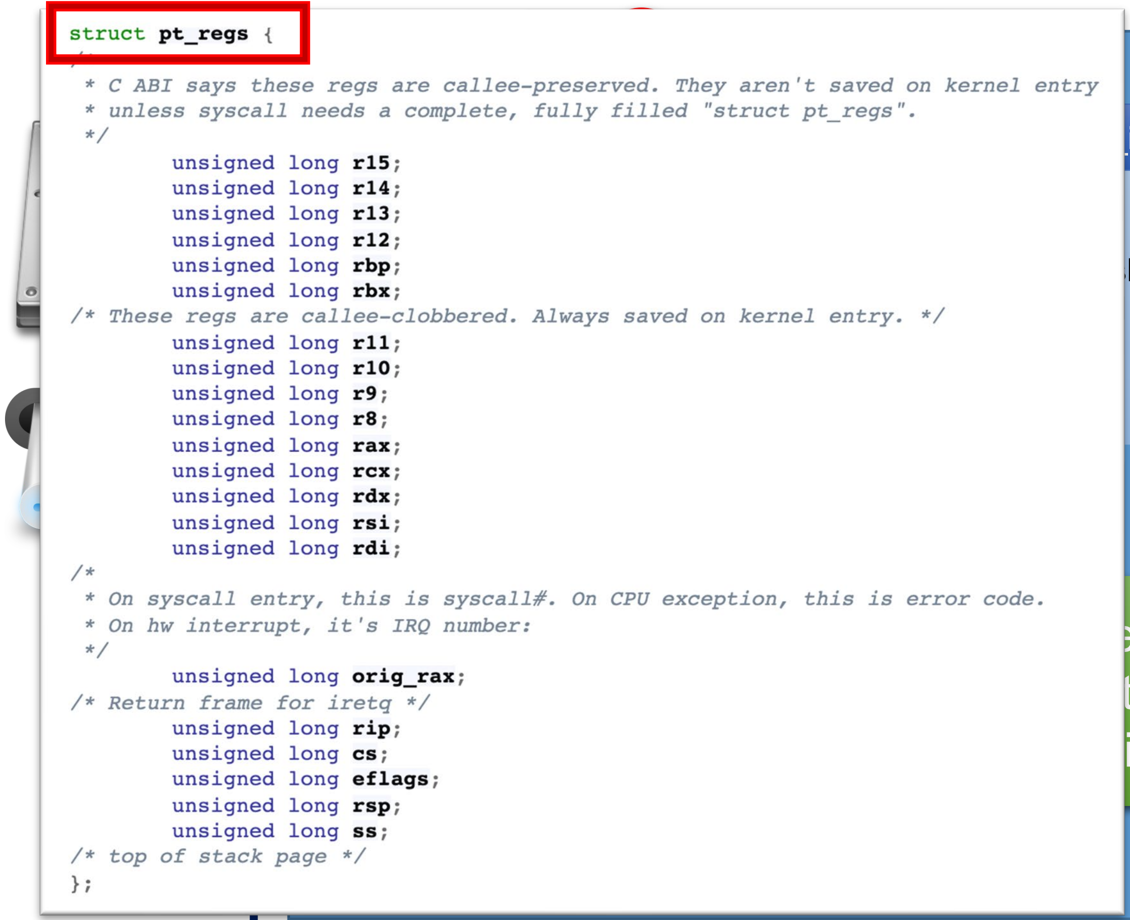
從OS開發者看「堆疊」



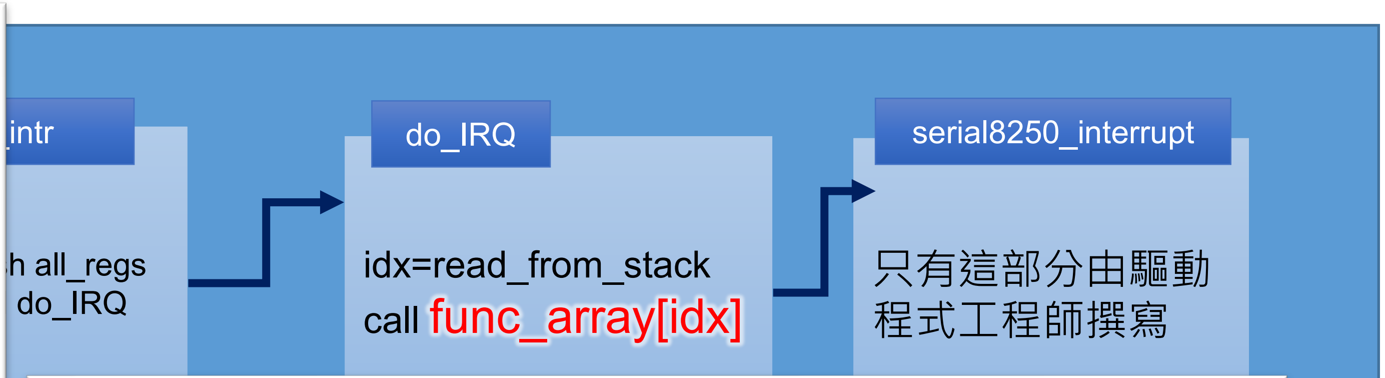


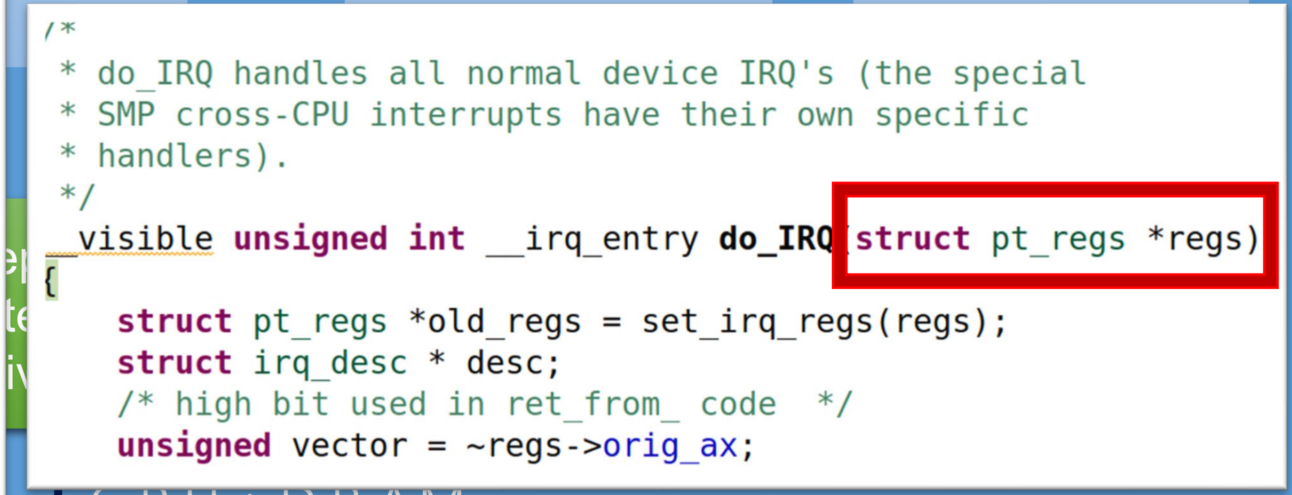
```
struct pt_regs {
/*
 * C ABI says these regs are callee-preserved. They aren't saved on kernel entry
 * unless syscall needs a complete, fully filled "struct pt_regs".
 */
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long rbp;
        unsigned long rbx;
/* These regs are callee-clobbered. Always saved on kernel entry. */
        unsigned long r11;
        unsigned long r10;
        unsigned long r9;
        unsigned long r8;
        unsigned long rax;
        unsigned long rcx;
        unsigned long rdx;
        unsigned long rsi;
        unsigned long rdi;
/*
 * On syscall entry, this is syscall#. On CPU exception, this is error code.
 * On hw interrupt, it's IRQ number:
 */
        unsigned long orig_rax;
/* Return frame for iretq */
        unsigned long rip;
        unsigned long cs;
        unsigned long eflags;
        unsigned long rsp;
        unsigned long ss;
/* top of stack page */
};
```

從OS開發者看「堆疊」



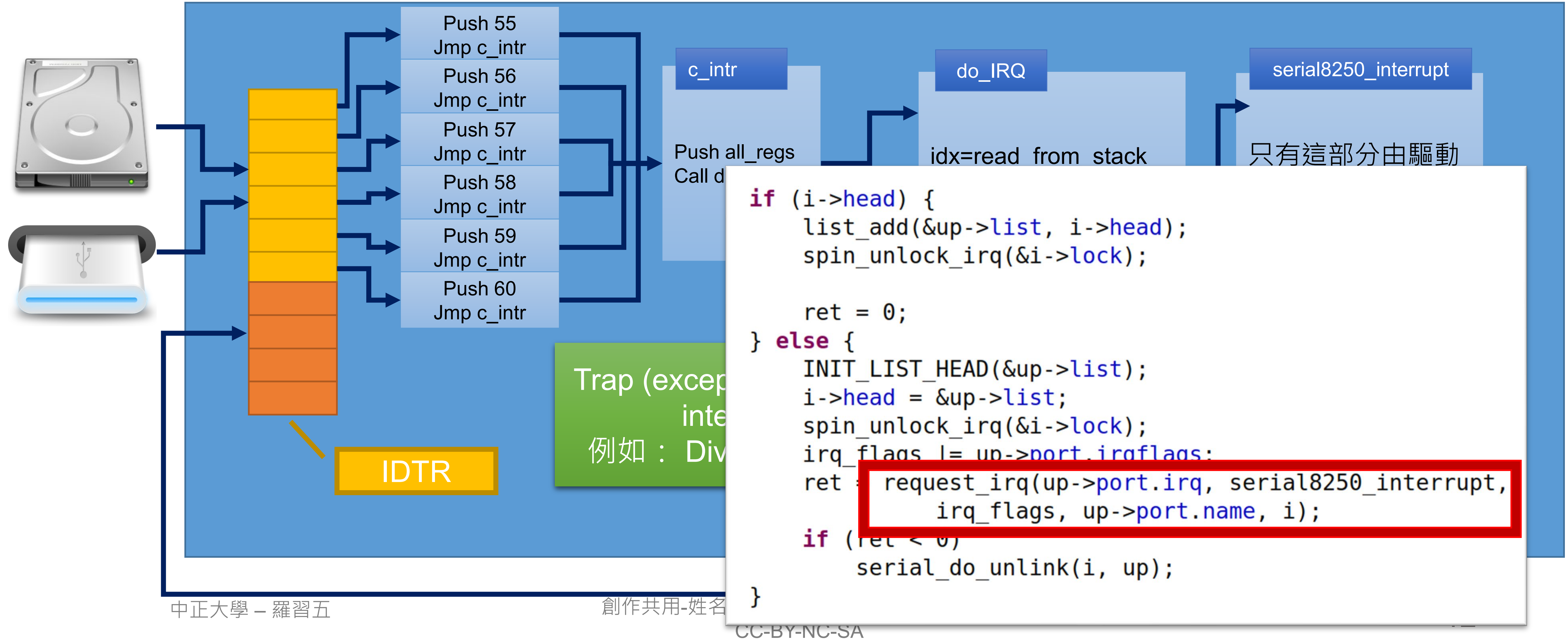
```
struct pt_regs {
/*
 * C ABI says these regs are callee-preserved. They aren't saved on kernel entry
 * unless syscall needs a complete, fully filled "struct pt_regs".
 */
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long rbp;
        unsigned long rbx;
/* These regs are callee-clobbered. Always saved on kernel entry. */
        unsigned long r11;
        unsigned long r10;
        unsigned long r9;
        unsigned long r8;
        unsigned long rax;
        unsigned long rcx;
        unsigned long rdx;
        unsigned long rsi;
        unsigned long rdi;
/*
 * On syscall entry, this is syscall#. On CPU exception, this is error code.
 * On hw interrupt, it's IRQ number:
 */
        unsigned long orig_rax;
/* Return frame for iretq */
        unsigned long rip;
        unsigned long cs;
        unsigned long eflags;
        unsigned long rsp;
        unsigned long ss;
/* top of stack page */
};
```





```
/*
 * 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  */
    unsigned vector = ~regs->orig_ax;
```

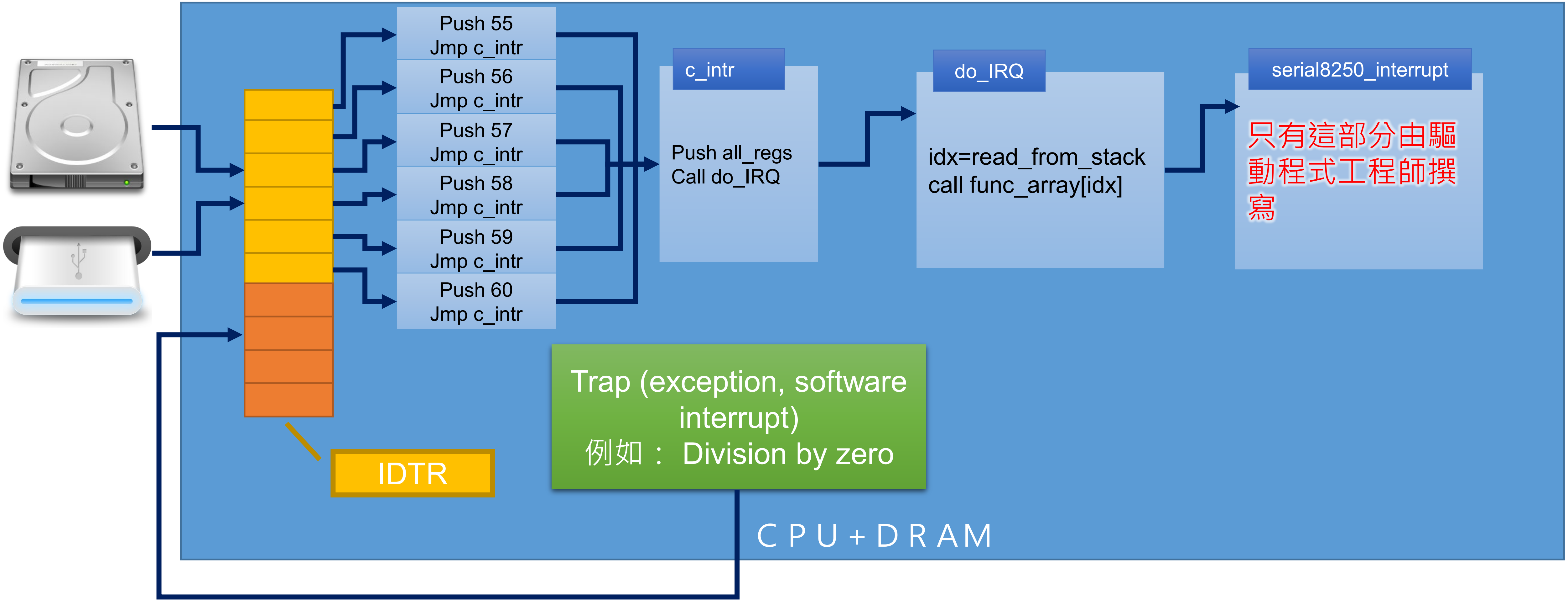
從『驅動程式開發者』看「註冊中斷」



```
if (i->head) {
    list_add(&up->list, i->head);
    spin_unlock_irq(&i->lock);

    ret = 0;
} else {
    INIT_LIST_HEAD(&up->list);
    i->head = &up->list;
    spin_unlock_irq(&i->lock);
    irq_flags |= up->port.irqflags;
    ret = request_irq(up->port.irq, serial8250_interrupt,
              irq_flags, up->port.name, i);
    if (ret < 0)
        serial_do_unlink(i, up);
}
```

從『驅動程式開發者』看中斷流程



只有這部分由驅動程式工程師撰寫

Trap (exception, software interrupt)
例如： Division by zero

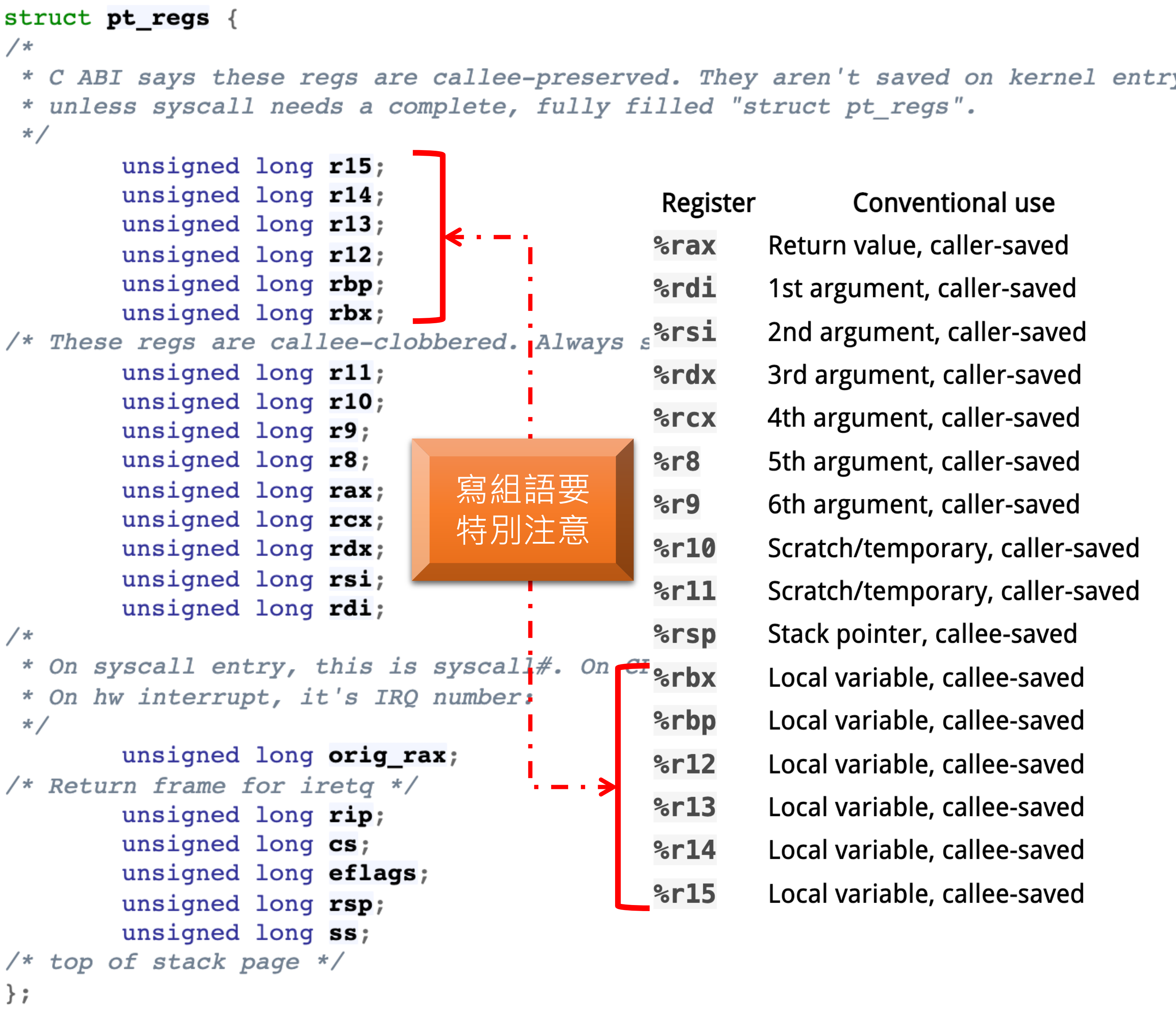
作業系統概論基於GNU/Linux

中正大學，資工系，作業系統實驗室，副教授 羅習五，shiwulo@gmail.com

附錄

關於堆疊

- 不管是system call、trap、interrupt，在呼叫第一個C函數時，堆疊一定是長這樣子

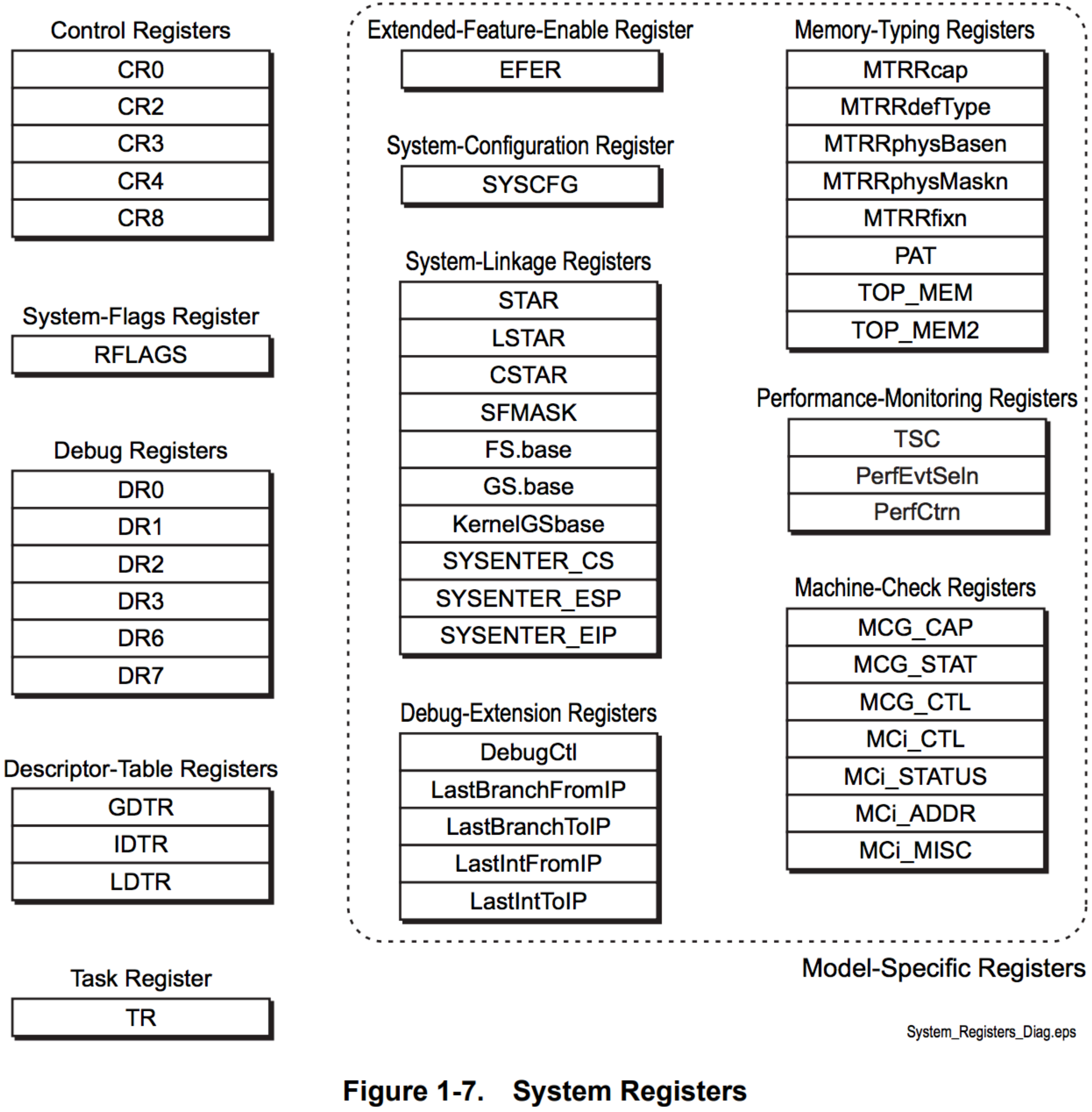


```
struct pt_regs {
/*
 * C ABI says these regs are callee-preserved. They aren't saved on kernel entr
 * unless syscall needs a complete, fully filled "struct pt_regs".
 */
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long rbp;
        unsigned long rbx;
/* These regs are callee-clobbered. Always s
        unsigned long r11;
        unsigned long r10;
        unsigned long r9;
        unsigned long r8;
        unsigned long rax;
        unsigned long rcx;
        unsigned long rdx;
        unsigned long rsi;
        unsigned long rdi;
/*
 * On syscall entry, this is syscall#. On CI
 * On hw interrupt, it's IRQ number:
 */
        unsigned long orig_rax;
/* Return frame for iretq */
        unsigned long rip;
        unsigned long cs;
        unsigned long eflags;
        unsigned long rsp;
        unsigned long ss;
/* top of stack page */
};
```

寫組語要特別注意

Register	Conventional use
%rax	Return value, caller-saved
%rdi	1st argument, caller-saved
%rsi	2nd argument, caller-saved
%rdx	3rd argument, caller-saved
%rcx	4th argument, caller-saved
%r8	5th argument, caller-saved
%r9	6th argument, caller-saved
%r10	Scratch/temporary, caller-saved
%r11	Scratch/temporary, caller-saved
%rsp	Stack pointer, callee-saved
%rbx	Local variable, callee-saved
%rbp	Local variable, callee-saved
%r12	Local variable, callee-saved
%r13	Local variable, callee-saved
%r14	Local variable, callee-saved
%r15	Local variable, callee-saved

Intel的系統暫存器



Control Registers
- CR0
- CR2
- CR3
- CR4
- CR8

System-Flags Register
- RFLAGS

Debug Registers
- DR0
- DR1
- DR2
- DR3
- DR6
- DR7

Descriptor-Table Registers
- GDTR
- IDTR
- LDTR

Task Register
- TR

Extended-Feature-Enable Register
- EFER

System-Configuration Register
- SYSCFG

System-Linkage Registers
- STAR
- LSTAR
- CSTAR
- SFMASK
- FS.base
- GS.base
- KernelGSbase
- SYSENTER_CS
- SYSENTER_ESP
- SYSENTER_EIP

Debug-Extension Registers
- DebugCtl
- LastBranchFromIP
- LastBranchToIP
- LastIntFromIP
- LastIntToIP

Memory-Typing Registers
- MTRRcap
- MTRRdefType
- MTRRphysBasen
- MTRRphysMaskn
- MTRRfixn
- PAT
- TOP_MEM
- TOP_MEM2

Performance-Monitoring Registers
- TSC
- PerfEvtSeln
- PerfCtrn

Machine-Check Registers
- MCG_CAP
- MCG_STAT
- MCG_CTL
- MCi_CTL
- MCi_STATUS
- MCi_ADDR
- MCi_MISC

Model-Specific Registers

System_Registers_Diag.eps

Figure 1-7. System Registers

- 大概介紹86的語法
  - https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html?wapkw=