# 作業二：
# 瞭解作業系統的進入點

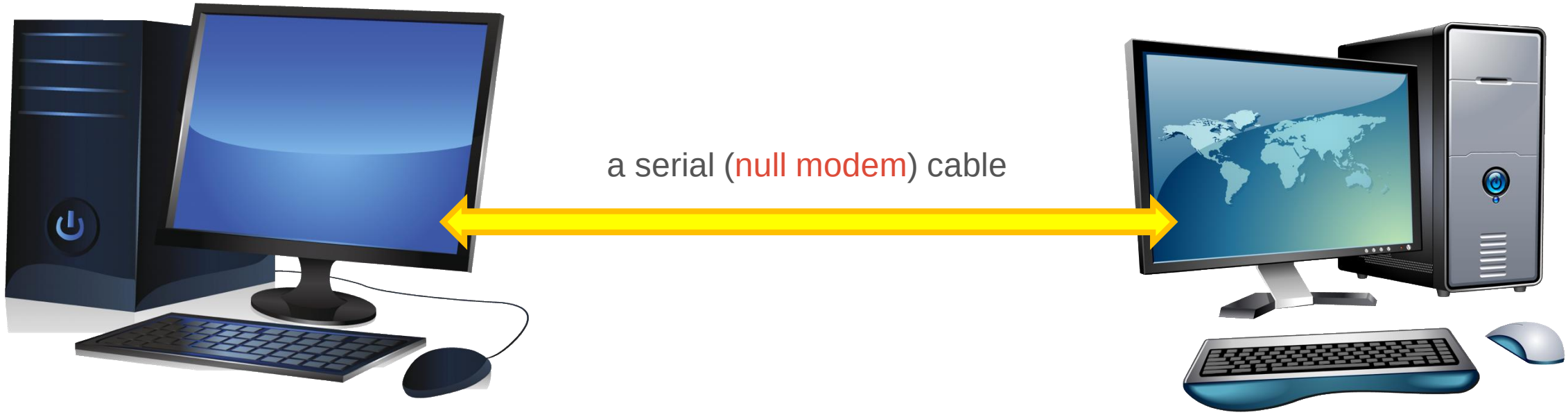
中正大學 作業系統實驗室
指導教授：羅習五



# 作業目標及負責助教

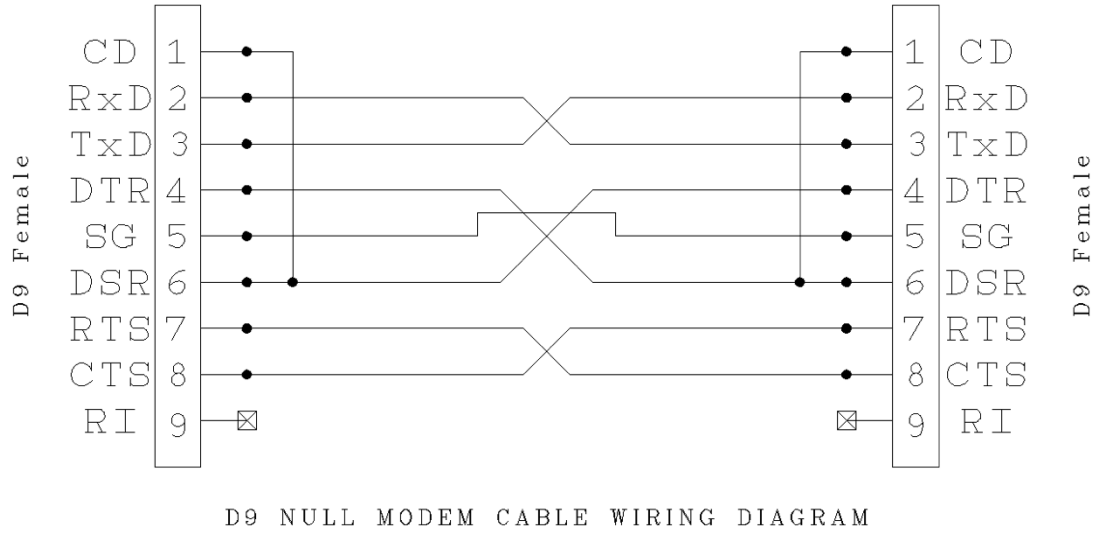
- 作業目標：
  - 了解Linux kernel的系統呼叫處理機制
  - 了解x86的除錯機制
  - 了解如何於PC上對kernel進行追蹤（trace）
  - trace二個system call
- 負責助教：張皓翔
  - Email：f26227279@gmail.com

# Debug Linux kernel的方式



# 什麼是a serial (null modem) cable

## 硬體配線圖

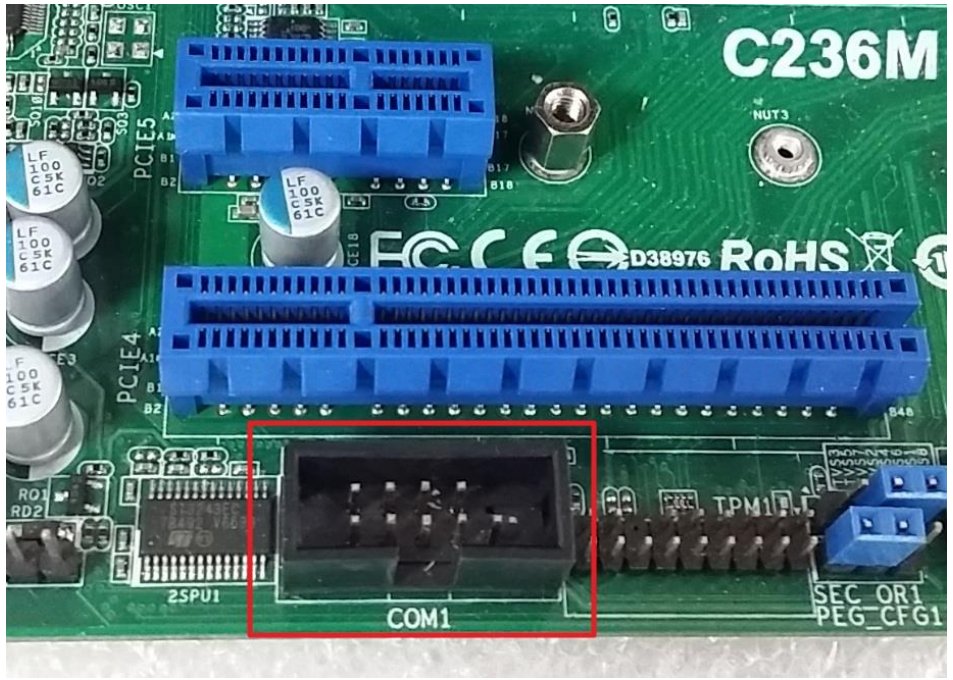


D9 NULL MODEM CABLE WIRING DIAGRAM

## 實際的照片

# 為什麼我的電腦上沒有這樣的接頭

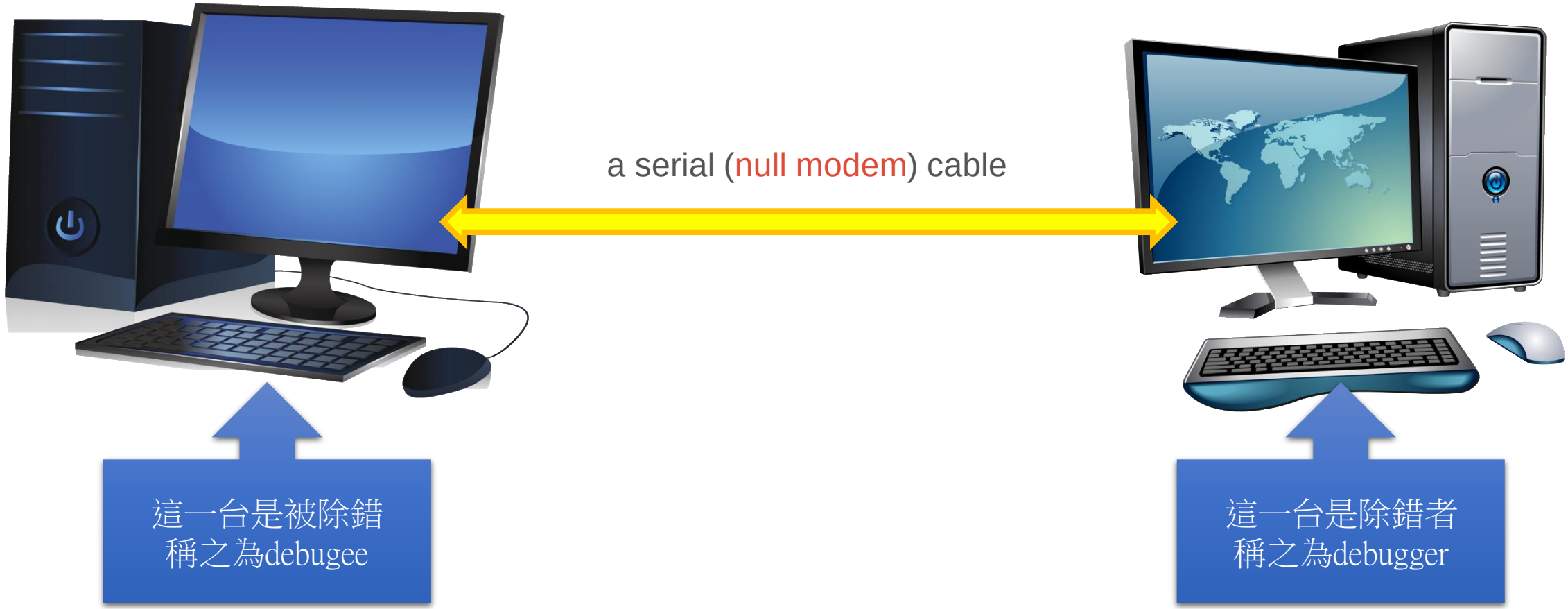
- 通常主機板上面還是會有，但是沒有接出來




# 為什麼使用a serial (null modem) cable

- 因為這一個硬體很簡單，driver也很簡單，幾乎不會出錯
- 假設使用乙太網路或者USB作為傳輸介面，那麼這二個本身可能就是個錯
- 真正在除錯的時候，「被除錯的」通常要真的接上serial cable，「除錯者」可以接USB轉serial cable，這樣較為方便（例如：筆記型電腦通常不支援serial cable）

# 因此通常需要二台機器來進行kernel的除錯



# Kernel除錯與傳統除錯機制的不同

- 如果我們對一般的應用程式進行除錯，由Linux kernel負責將debugger和debuggee連接上。方式是透過ptrace()這個system call
- 當debugger（例如：gdb）透過ptrace()連上debuggee（例如：a.out）後，kernel會將它所接收到的除錯訊息交付給gdb

# x86-64對除錯的支援

- 幾乎所有的處理器都支援除錯機制
- x86-64支援特殊指令int3，這是一道組合語言，當把這一道組合語言插入到程式中，會觸發「軟體中斷」，而這道中斷的意思就是，「執行到中斷點囉，請OS替我呼叫gdb進行後續處理」

# x86-64對除錯的支援

```
1.  #include <stdio.h>
2.  int main() {
3.      __asm__("int3");          //產生breakpoint中斷
4.      printf("before\n");
5.      __asm__("int $0x03");     //自己產生第三號中斷
6.      printf("after\n");
7.  }
```

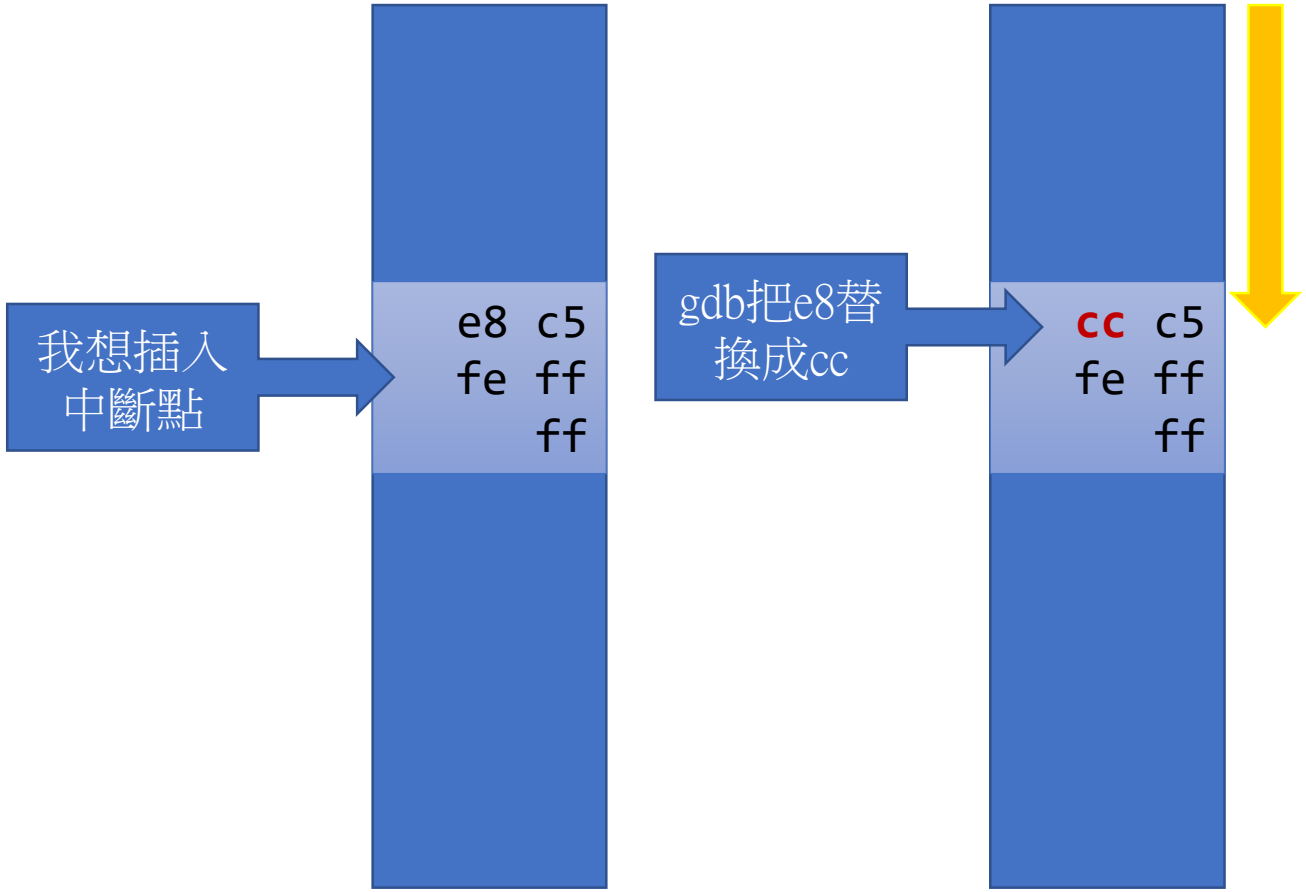
# 執行結果

```
Reading symbols from ./a.out...done.
(gdb) r     /*沒有設定任何中斷點就開始執行*/
Starting program: /home/shiwulo/a.out
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at hello.c:4
4               printf( “before\n” );          /*在before的地方觸發了breakpoint*/
(gdb) c
Continuing.
before
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at hello.c:6
6               printf("after\n");                /*在after的地方觸發了breakpoint*/
(gdb)
```

# 結論

- 不管是 “int 3”或者是 “int3”都觸發breakpoint，而且gdb也抓到了這二個breakpoint
- “int 3”可能被編譯為「 0xCD03 」或「0xCC 」
- “int3”會被編譯為「0xCC 」

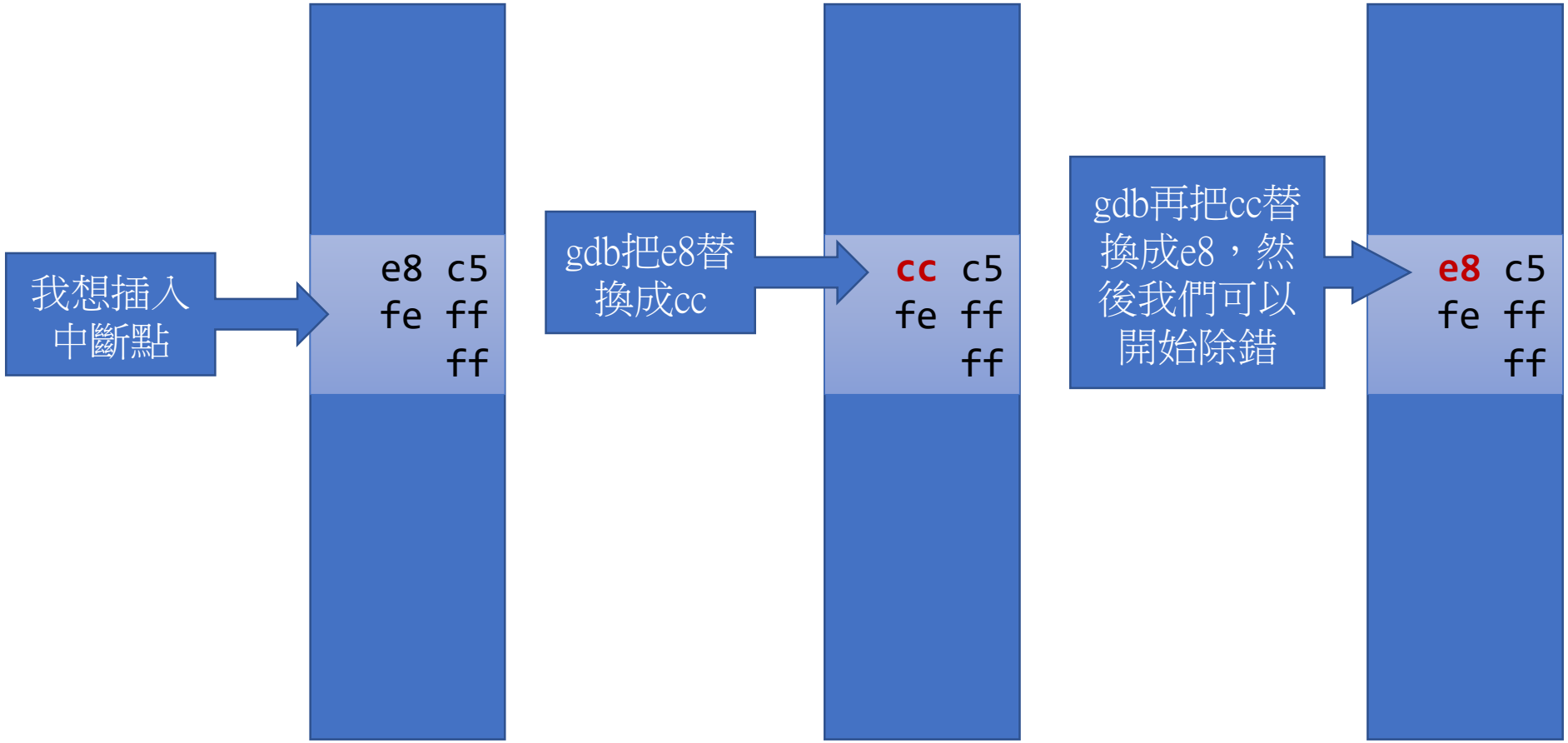
# gdb怎麼做？




程式往下執行遇到breakpoint（即：cc），
觸發軟體中斷，OS kernel接手，OS將控制
權交給gdb

「e8 c5 fe ff ff」的意義是：callq  0x510 <puts@plt>

# gdb怎麼做？




「e8 c5 fe ff ff」的意義是：callq  0x510 <puts@plt>

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

```
/* Interrupts/Exceptions */
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 */
};
```

Linux的確將第3號中斷當成breakpoint

# 逐步追蹤

- 在x86的FLAGS暫存器中有一個bit稱之為TF（ Trap flag，single step ）
- 若這個暫存器設定為1，那麼CPU每執行一道指令以後，立即產生trap
- 同樣的OS kernel會收到這個trap，然後交給gdb處理，這就是gdb的逐步追蹤

https://en.wikipedia.org/wiki/FLAGS_register

# Hardware breakpoint

- 在x86上，還可以使用hardware breakpoint，其指令如下

  (gdb) hb ？？？？
- Hardware breakpoint通常用於該程式區段無法設定中斷點，例如：唯讀區域，ROM等等。
- x86支援最多6組hardware breakpoint

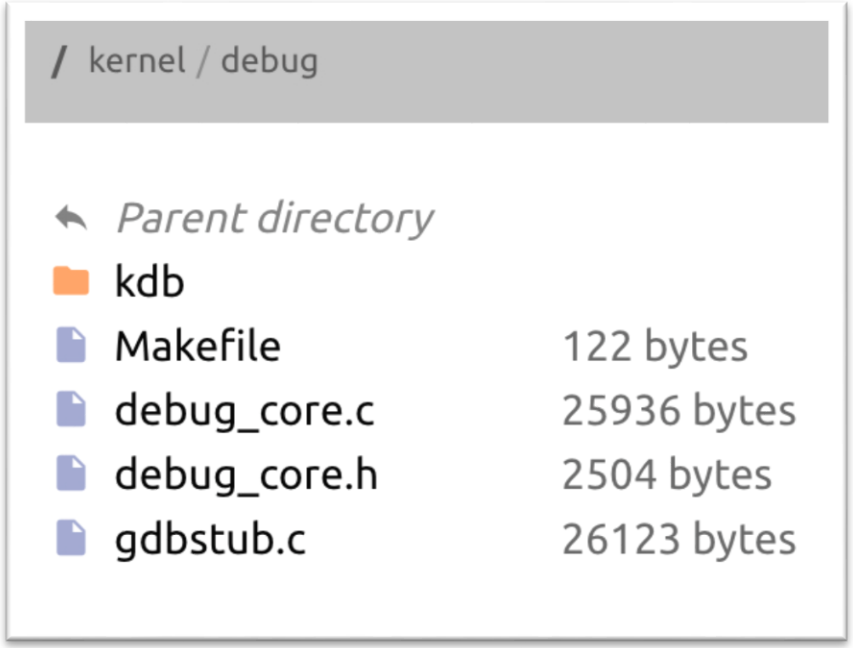
https://en.wikipedia.org/wiki/X86_debug_register

# 小結論

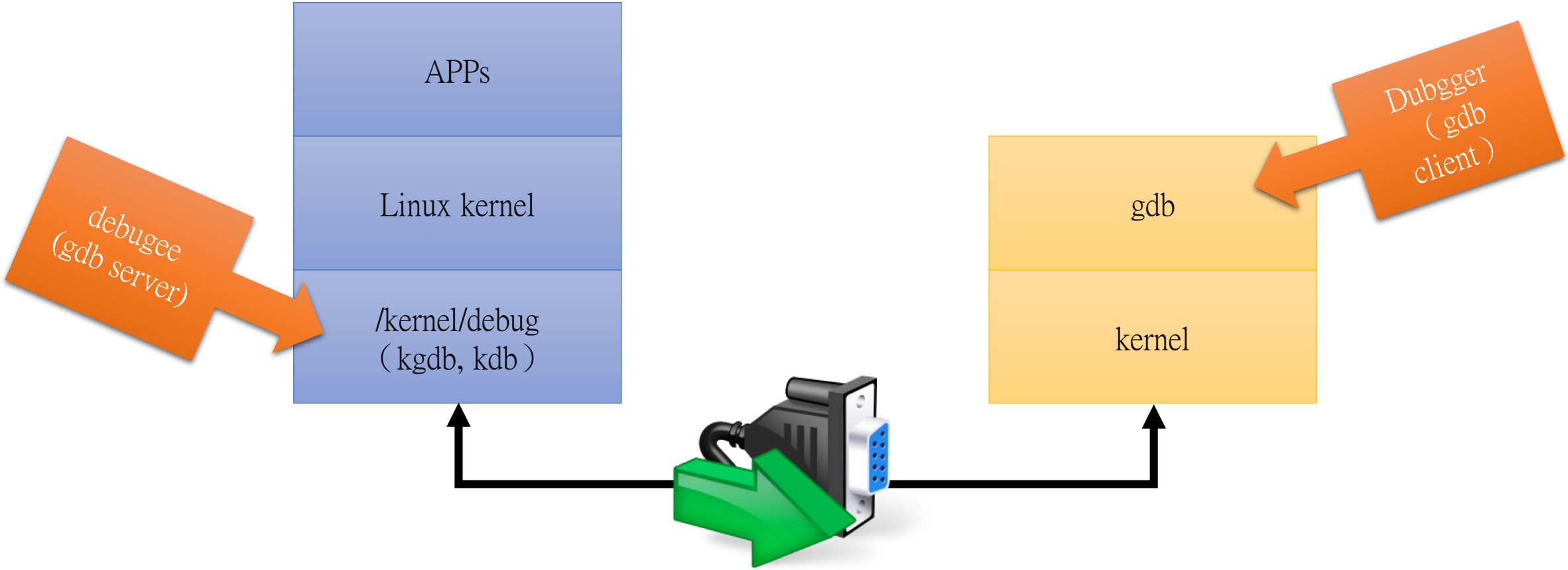
- 如果是user space的除錯，硬體提供三種機制，而這三種機制都依賴OS kernel處理software interrupt。
- 但現在我們要對OS kernel除錯，應該怎樣做呢？

# Linux kernel對除錯的支援

- Linux kernel的程式碼網站
  - https://elixir.bootlin.com
- Linux內部附有除錯功能，如下圖所示：



# 軟體的除錯架構



# 小結論

- Linux內部有一隻小程式叫做kgdb，可以傳遞特殊的命令啟動kgdb，之後如果kernel觸發了breakpoint等等情況（投影片後面會介紹），kgdb會將除錯訊息透過serial port傳遞給另一台機器上的gdb
- 限制：
  1. 必須等到核心執行到某個階段才可以開始除錯，例如：我們無法除錯核心解壓縮部分的程式碼，在serial port的driver起來前，也無法除錯
  2. 如果kernel暫停中斷（disable interrupt），在這個情況下也無法除錯

# 克服無法除錯的窘境

下面方法都可以用來對kgdb無法除錯的部分進行除錯

1. 使用ICE（這是一種特殊的硬體， In-circuit emulation）
2. 使用軟體模擬器，由於每一道指令都是模擬的，因此可以進行全系統的除錯，最著名的軟體是QEMU
   （ https://www.qemu.org/ ）

# 這次作業的目標

- 更精準來講，本次作業希望模擬「進行實體除錯」會遇到的問題，讓大家知道核心如何除錯（例如：我們要替一個新的硬體撰寫driver，這時候就無法靠模擬器了）
- 雖然有前面所說的總總限制，但我們依然可以對系統呼叫進行追蹤、除錯

# 使用virtual machine進行除錯

# 使用virtual machine除錯

- 正式的除錯方法需要二台電腦，使用上較為不方便
- 由於我們是想要了解OS的運作方式，因此可以使用virtual machine除錯
- 在這裡我們選用vmware為例子，說明如何除錯

# 軟體架構圖


在左圖中，使用「pipe」代替serial cable，連接二個virtual machine

# 預備編譯核心的環境

$ sudo apt install git flex bison bc libssl-dev gawk libudev-dev ocl-icd-opencl-dev libpci-dev libelf-dev python2.7 libncurses-dev fakeroot kernel-wedge binfmt-support ksh lsscsi binfmt-support libpcre16-3 libpcre3-dev libpcre32-3 libpcrecpp0v5 libsepol1-dev libattr1-dev libblkid-dev libpcre16-3 libpcre3-dev libpcre32-3 libpcrecpp0v5 libselinux1-dev libsepol1-dev uuid-dev debugedit libarchive13 libdw1 liblua5.2-0 liblzo2-2 libnspr4 libnss3 librpm8 librpmbuild8 librpmio8 librpmsign8 rpm rpm-common rpm2cpio spl-dkms kernel-package

# 編譯核心

- 下載核心。sudo apt install linux-source。或從www.kernel.org下載。
- 從/boot複製config到Linux kernel source裡面
  - 例如：cp /boot/config-???? ~/linux-source-dir/.config
- 修改kernel的設定
  - make menuconfig /*看一下作業系統有哪些新的功能*/
  - 由於設定breakpoint需要動態修改程式碼，因此確定註解掉這一行

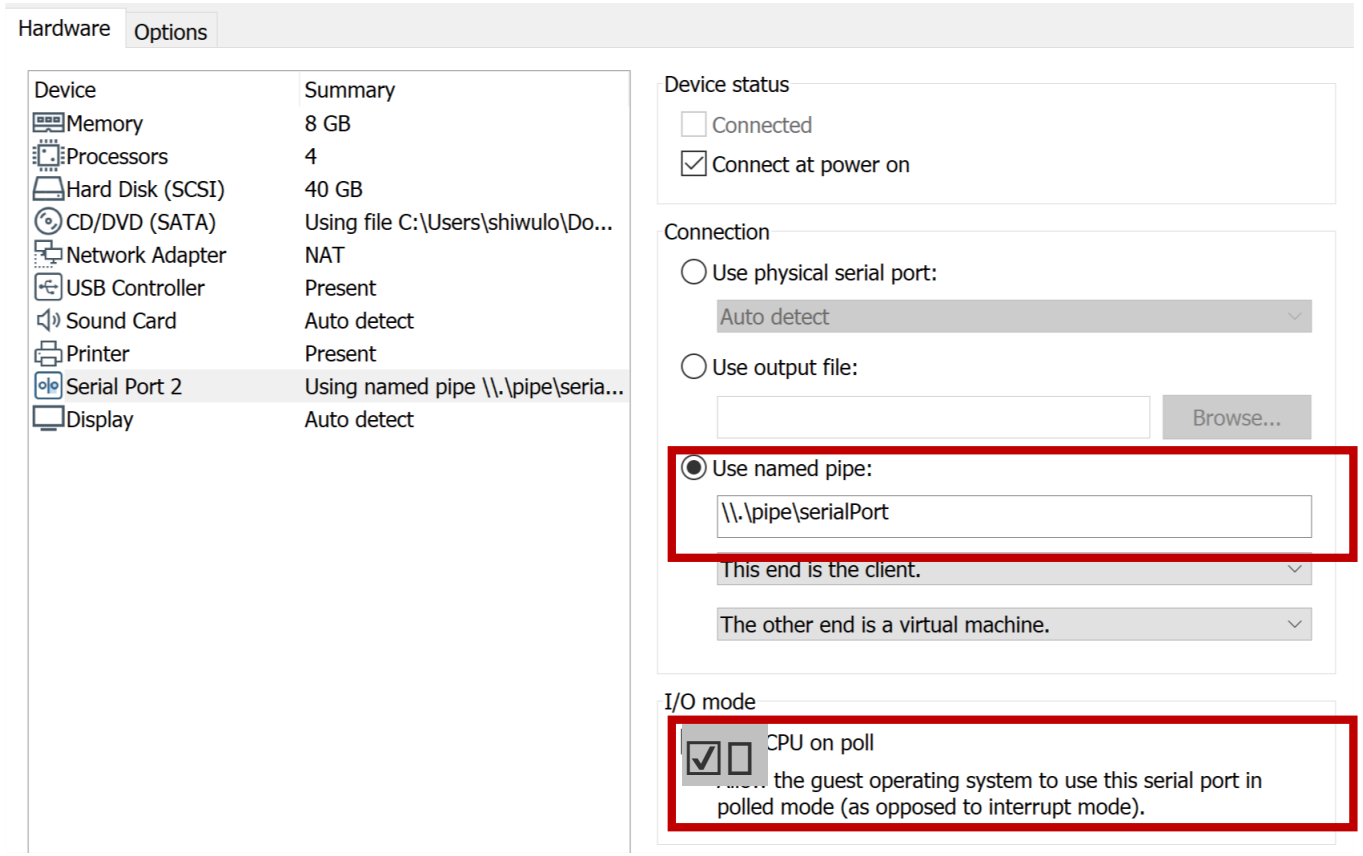
    *# CONFIG_STRICT_KERNEL_RWX is not set*
  - 編譯核心

    make -j 8
- 安裝
  - sudo make modules -j 8
  - sudo make modules_install
  - sudo make install

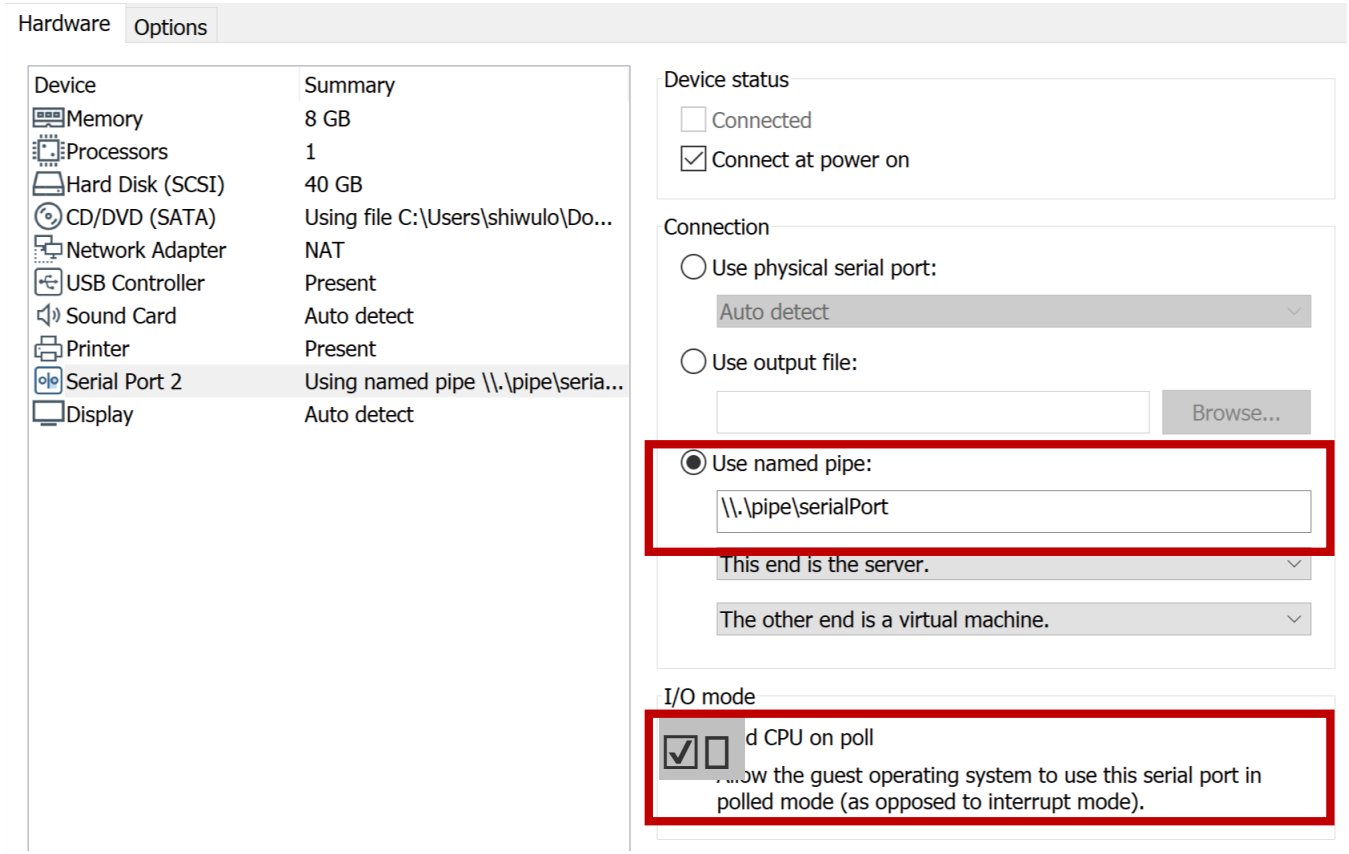
# 複製virtual machine

- 選擇Linked clone
- 建議將debuggee的virtual machine的CPU數量設定為1

# Debugger的設定



# Debuggee的設定



# debuggee

- 從開機開始攔截(建議使用這一個)
  - sudo vim /etc/default/grub
  - 對grub檔案做如下的設定

    GRUB_CMDLINE_LINUX_DEFAULT="quiet splash nokaslr kgdboc=ttyS1, 115200 kgdbwait"

    在你的系統可能不是ttyS1，也有可能是ttyS0，視你的vmware的組態而定
  - sudo update-grub
- 開機之後再開始除錯
  - GRUB的設定為：GRUB_CMDLINE_LINUX_DEFAULT="quiet splash nokaslr kgdboc=ttyS1, 115200"
  - sudo -s
  - sudo echo "1" >/proc/sys/kernel/sysrq
  - sudo echo g > /proc/sysrq-trigger

# debugger

- vim ~/.gdbinit

add-auto-load-safe-path /home/os2018/linux-4.18.14/scripts/gdb/vmlinux-gdb.py

set auto-load safe-path /

- os2018@vm:~/linux-4.18.14$ sudo gdb ./vmlinux
- (gdb) set serial baud 115200
- (gdb) target remote /dev/ttyS1
- (gdb) b ???(這裡填入你想要觀察的函數)

# 怎樣知道我可以設定哪些函數

- $ sudo less /boot/System.map-???
- 上述檔案是用工具「nm」作成的
- 屬性是T的都可以設定breakpoint
  - $ sudo less /boot/System.map-4.15.0-36-generic | grep do_fork
  - ffffffff8108b590 T _do_fork
  - 由上可以知道我們要trace fork，就必須將breakpoint設定在_do_fork
- System.map詳細介紹，請參考底下網頁
  - https://zh.wikipedia.org/wiki/System.map

# debugger (範例：追蹤_do_fork)

```
(gdb) set serial baud 115200
(gdb) target remote /dev/ttyS1
Remote debugging using /dev/ttyS1
kgdb_breakpoint () at kernel/debug/debug_core.c:1073
1073            wmb(); /* Sync point after breakpoint */
(gdb) b _do_fork
Breakpoint 1 at 0xffffffff810f5390: file kernel/fork.c, line 2098.
(gdb) c
Continuing.
[Switching to Thread 2]
Thread 3 hit Breakpoint 1, _do_fork (clone_flags=8390417,
    stack_start=18446744071580010144, stack_size=18446612139860579136,
    parent_tidptr=0x0 <irq_stack_union>, child_tidptr=0x0 <irq_stack_union>, tls=0)
    at kernel/fork.c:2098
2098    {
```

# debugger (範例：追蹤_do_fork的caller)

```
(gdb) bt
#0  _do_fork (clone_flags=18874385, stack_start=0, stack_size=0,
    parent_tidptr=0x0 <irq_stack_union>, child_tidptr=0x7ff24e25d9d0,
    tls=140678669915904) at kernel/fork.c:2098
#1  0xffffffff810f5777 in __do_sys_clone (tls=<optimized out>,
    child_tidptr=<optimized out>, parent_tidptr=<optimized out>,
    newsp=<optimized out>, clone_flags=<optimized out>) at kernel/fork.c:2230
#2  __se_sys_clone (tls=<optimized out>, child_tidptr=<optimized out>,
    parent_tidptr=<optimized out>, newsp=<optimized out>, clone_flags=<optimized out>)
    at kernel/fork.c:2224
#3  __x64_sys_clone (regs=<optimized out>) at kernel/fork.c:2224
#4  0xffffffff810043b5 in do_syscall_64 (nr=56, regs=0xffffc900022f3f58)
    at arch/x86/entry/common.c:290
#5  0xffffffff81c0008d in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:238
#6  0x0000000000000000 in ?? ()
```

# debugger (範例：追蹤_do_fork)

(gdb) d

Delete all breakpoints? (y or n) y

(gdb) b entry_SYSCALL_64

Breakpoint 2 at 0xffffffff81c00020: file arch/x86/entry/entry_64.S, line 214.

# 不能除錯，看原始程式碼

arch/x86/entry/entry_64.S

```
ENTRY(entry_SYSCALL_64)
        UNWIND_HINT_EMPTY
        /*
         * Interrupts are off on entry.
         * We do not frame this tiny irq-off block with TRACE_IRQS_OFF/ON,
         * it is too small to ever cause noticeable irq latency.
         */

        swapgs
        /*
         * This path is only taken when PAGE_TABLE_ISOLATION is disabled so it
         * is not required to switch CR3.
         */
        movq    %rsp, PER_CPU_VAR(rsp_scratch)
        movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                      /* pt_regs->ss */
        pushq   PER_CPU_VAR(rsp_scratch)        /* pt_regs->sp */
        pushq   %r11                            /* pt_regs->flags */
        pushq   $__USER_CS                      /* pt_regs->cs */
        pushq   %rcx                            /* pt_regs->ip */
GLOBAL(entry_SYSCALL_64_after_hwframe)
        pushq   %rax                            /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        TRACE_IRQS_OFF

        /* IRQs are off. */
        movq    %rax, %rdi
        movq    %rsp, %rsi
        call    do_syscall_64           /* returns with IRQs disabled */
```

# 這就是我們說的問題：breakpoint是「軟體中斷」，有時候無法除錯

- https://en.wikipedia.org/wiki/INT_(x86_instruction)
- Breakpoint是使用INT3實現，INT3會觸發軟體中斷，但所有的中斷已經被disable

# 試試看do_syscall_64

```
(gdb) b do_syscall_64
Breakpoint 5 at 0xffffffff81004350: file arch/x86/entry/common.c, line 273.
(gdb) c
Continuing.
[Switching to Thread 1584]
Thread 340 hit Breakpoint 5, do_syscall_64 (nr=39, regs=0xffffc9000279ff58)
    at arch/x86/entry/common.c:273
273     {
(gdb)
```

# do_syscall_64

```
271  #ifdef CONFIG_X86_64
272  __visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
273  {
274          struct thread_info *ti;
275
276          enter_from_user_mode();
277          local_irq_enable();
278          ti = current_thread_info();
279          if (READ_ONCE(ti->flags) & _TIF_WORK_SYSCALL_ENTRY)
280                  nr = syscall_trace_enter(regs);
281
282          /*
283           * NB: Native and x32 syscalls are dispatched from the same
284           * table.  The only functional difference is the x32 bit in
285           * regs->orig_ax, which changes the behavior of some syscalls.
286           */
287          nr &= __SYSCALL_MASK;
288          if (likely(nr < NR_syscalls)) {
289                  nr = array_index_nospec(nr, NR_syscalls);
290                  regs->ax = sys_call_table[nr](regs);
291          }
292
293          syscall_return_slowpath(regs);
294  }
295  #endif
```

# 承接上頁

```
(gdb) b 290
Breakpoint 11 at 0xffffffff810043a2: file arch/x86/entry/common.c, line 290.
(gdb) c
Continuing.
[Switching to Thread 1584]

Thread 340 hit Breakpoint 11, do_syscall_64 (nr=39, regs=0xffffc9000279ff58)
    at arch/x86/entry/common.c:290
290                     regs->ax = sys_call_table[nr](regs);
(gdb) s
__x86_indirect_thunk_rax () at arch/x86/lib/retpoline.S:32
32      GENERATE_THUNK(_ASM_AX)
```

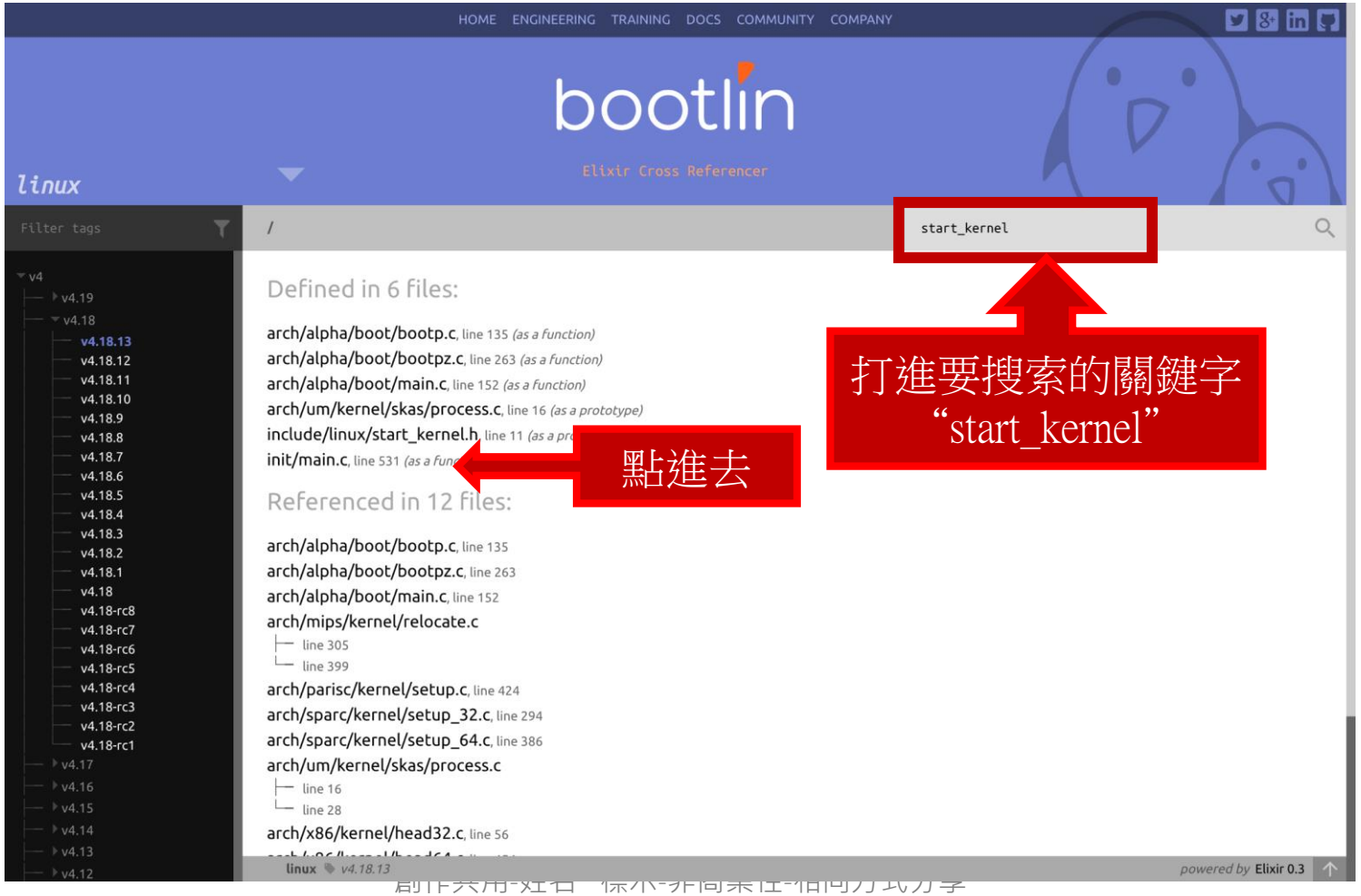
# 作業

- 請完成_do_fork的完整追蹤
- 請追蹤getpid()這個system call的完整追蹤
- 上述二個作業只需要列出函數呼叫圖即可
  - 請同學們自發性的看一下Linux kernel的註解，提升自己的能力
  - 在追蹤的時候有時候必須參考原始程式碼，請google「lxr linux」

# Linux kernel的網頁相關資訊

- https://elixir.bootlin.com

# 使用LXR查閱程式碼



# 附錄：有辦法追蹤系統初始化嗎？

```
(gdb) b start_kernel
Breakpoint 2 at 0xffffffff82938ca6: file init/main.c, line 532.
(gdb) l start_kernel
527             /* Should be run after espfix64 is set up. */
528             pti_init();
529     }
530
531     asmlinkage __visible void __init start_kernel(void)
532     {
533             char *command_line;
534             char *after_dashes;
535
536             set_task_stack_end_magic(&init_task);
```

# 使用kgdb搭配/kernel/debug無法辦到

- 由於/kernel/debug的程式碼是在start_kernel之後才初始化，因此無法追蹤start_kernel這個函數
- 如果要追蹤start_kernel這個函數，需要使用特殊的硬體(例如：ICE)，或者使用純軟體的模擬(例如：Qemu)

# 評分方式

- 60pt 完成_do_fork的函數呼叫圖
- 30pt 完成getpid的函數呼叫圖
- 10pt 指出_do_fork和getpid的返回路徑是否不同，如果不同，為什麼不同
- 繳交期限：2018/11/01 23:59:59以前
- 繳交方式：助教於星期六以前指定

# 這一頁是祕技

只有聰明的人才看得出來 ^_^