LAB5 Report:

0256823 呂東融

2.1

Ans: unlock()的動作是 enable interrupt，此處我們不需要特別在 env_pop_tf()前 enable interrupt 仍可使 keyboard、timer 的 interrupt 的動作執行，因為當執行 env_pop_tf()後，回到 user space，是使 kernel register 的值（或是 global variables ）回復的動作，所以在 env_pop_tf() 前，即使產生了 interrupt routine 也並不會造成 kernel registers 的值有任何的污染或是即使有也立即回復（因為一進入 env_pop_tf()就是做回復的動作）

env_pop_tf() 的目的是使在 trapframe 中的 register values 回復為進入 kernel mode 前的值，因為當 iret 發生時，processor 會 pushes：

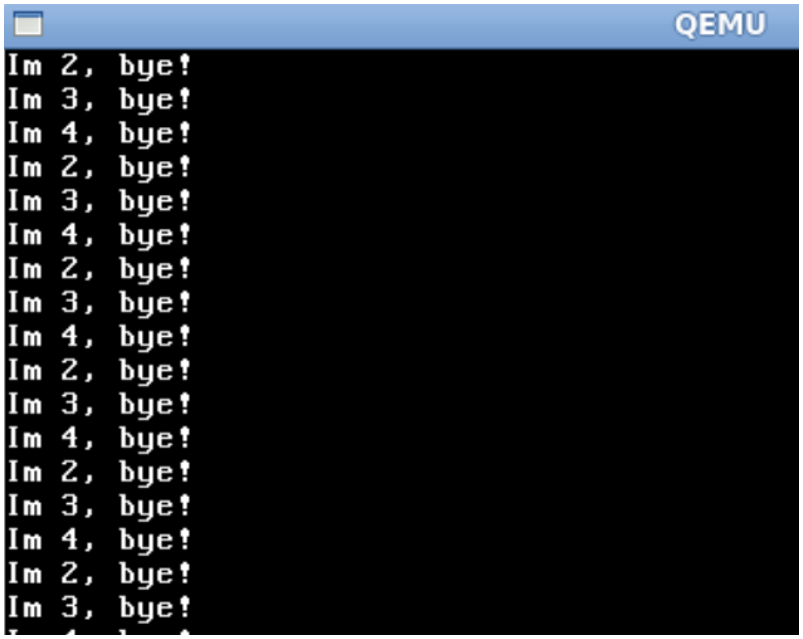
1. Stack segment and pointer (ss:esp)
2. EFLAGS
3. The return code segment and instruction pointer (cs:eipi)
4. Error code(我們的 code 沒有)

Iret 的工作就是 undoing setps 1-3, 藉由回復的動作: pushing the required information to the stack segmnets，重回到 ring 3(user mode)，即使 iret 前面並沒有一個真正的中斷引起的存入 stack 的動作，不過等到執行 iret 時，傳回的效果是一樣的

2.2

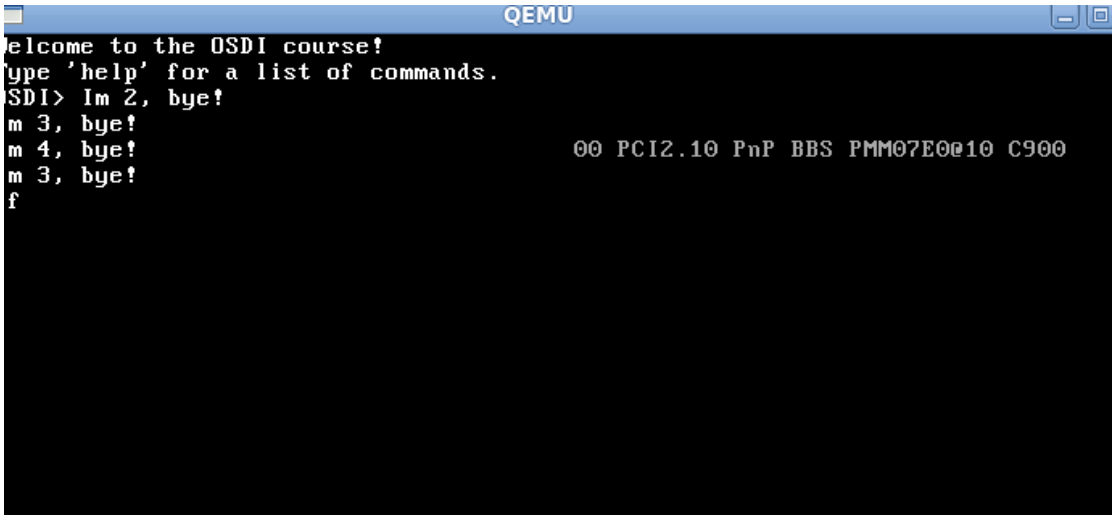
Ans: **當 Uncomment(1) lock， 在 fork() trap 進入 kernel section 時無法執行 timer 或 keyboard 的 interrupt。此時每一個 fork 出來的 task 在每一個 tcik 被 kill 掉，在這過程會不斷的 trap 到 kernel section，同時，fork() trap 進 kernel section 時會做 copy trapframe 的動作，而 kill 會再呼叫 sched_yield()，替換 task 的行程，而產生 tasks 不斷交替執行 printf，總共有五個 task，1 個執行 shell、1 個 parent、fork 出一個 child，仍有兩個 available 的 tasks 可以被 scheduling。**

而由於 lock 住整個 kernel section 所以，會在每次 fork()時 trap，當執行到 trap_dispatch()會在 if ((tf->tf_cs & 3) == 3)的判斷式下去 copy 在 user mode 的 trap frame 值，都能被正確的保存到 fork()行程的 entry point 狀態。所以保證了任一個 fork() trap 到 kernel section 到返回 user mode 的行程中，不會被其它的 interrupt 中斷而污染到 kernel variables。



2.3

Ans: 當 **Uncomment (2) lock**，**and comment (1) lock**，我們如隨意輸入字元，其使 **schedule** 每一次所產生的 **task**，隨即被 **task2()kill** 的完整行為被中斷掉，因 **lock(2)** 的位置，是在已經做完 **ISR** 的情況下，且也在做恢復 **trapframe** 的動作前（即返回 **user mode** 前），即被 **keyboard** 的 **interrupt** 所中斷掉。所以 **currently trapframe** 會變為 **keyboard** 或 **timer** 中斷所產生的 **trapframe**，執行對應的 **ISR**。，不過這 **lock()**只有 **partial kernel section**，所以在中斷剛好發生在 **fork trap** 到 **kernel** 的時間點，才有這樣的情形，所以仍有一段時間內，**round robin scheduling** 的動作可順利執行，但一但無法正常 **fork** 即會跳至 **shell()**，而無法再進入由 **child** 所執行 **task2()**的執行畫面。



2.4

Ans: 要儘可能的用最小的範圍做 **lock()**的動作**(add lock mechanism)**

找到每個 **task** 所 **shard** 相同的 **kernel stack**，並在使用前 **lock** 住。

在每 **task2()**裡面，每次呼叫一次 **fork()**，都會使 **CPU** 進行 **trap process(check IDT,dpl,…)** 並跳到 **sys_fork()**中，而每次的 **sys_fork()** 都會 **copy parent's trapframe to new task's trapframe**。但是由於 **TIME_QUANT** 設為 **1**，會使每 **1** 個 **tick** 即呼叫 **sched_yield()**，找出 **running able state** 的 **task** 設為 **running**，而將原先 **running** 的 **cur_task** 指向另一個原本是 **runnable** 的 **task**

。在原先 **fork()**時，必須使用到 **copy cur_task** 的 **trapframe**，但 **timmer interrupt** 會造成 **cur_task** 的指向的位置改變，而使得有各種不可預期的錯誤: 例

有可能會使原先 **parent fork** 出來的 **child**，**pid** 指向 **parent** 以外的 **task**，或者造成在計算新 **task** 的 **esp** 時發生錯誤，使 **esp** 的估算錯誤，**return** 回 **user space** 時指向不同的 **entry point**。

因此在做 **copy global kernel stack:** 即 **copy trapframe** 時都應予以 **lock**

實作在 **trap.c** 上加一個 **lock()**、**task.c** 加 **lock()** 與 **unlock()**

```
      // Copy trap frame (which is currently on the stack)
      // into 'cur_task->tf', so that running the environment
      // will restart at the trap point.
     lock();
      cur_task->tf = *tf;
//      lock();
      tf = &(cur_task->tf);


    }
    // Do ISR
    trap_hnd[tf->tf_trapno](tf);

    // Pop the kernel stack
//    lock();
    env_pop_tf(tf);

    return;
  }
```

```
   /* Lab4 TODO: Copy parent's tf to new task's tf */
lock();
 tasks[pid].tf = cur_task->tf;
 /* Lab4 TODO: Copy parent's usr_stack to new task's usr_stack and reset the new task's esp pointer. */
   memcpy(tasks[pid].usr_stack, cur_task->usr_stack, USR_STACK_SIZE );
   tasks[pid].tf.tf_esp = tasks[pid].usr_stack + cur_task->tf.tf_esp - cur_task->usr_stack;
   tasks[pid].tf.tf_regs.reg_eax = pid;
 unlock();
```

LAB1~4: I have learned from the OSDI

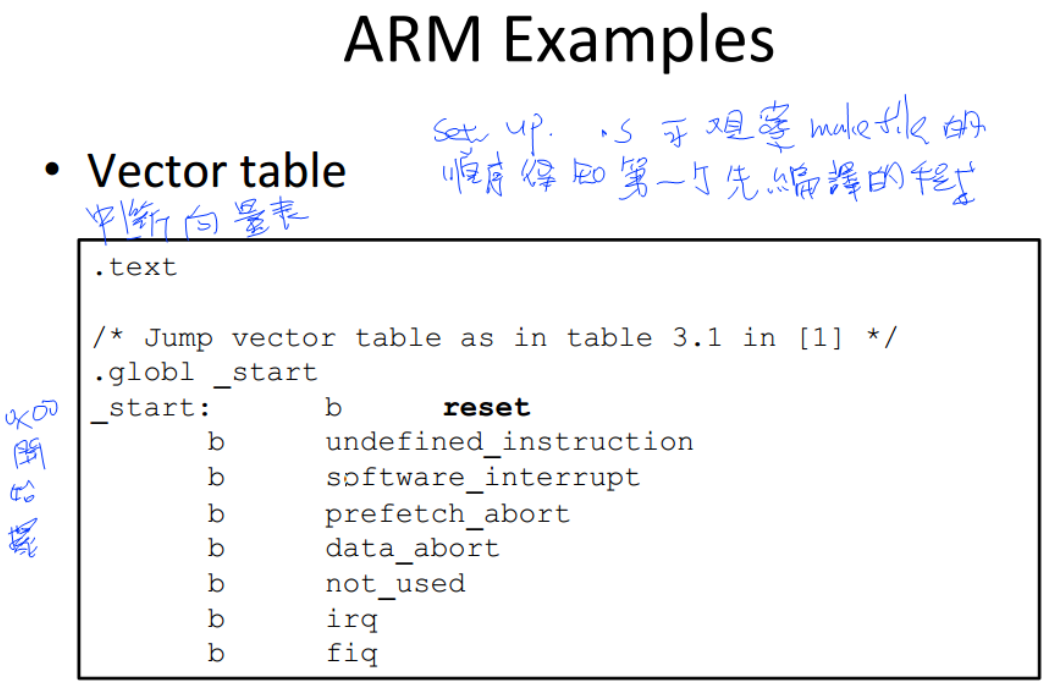
LAB1:

- **GDB debugging**
- **Makefile**

LAB2:

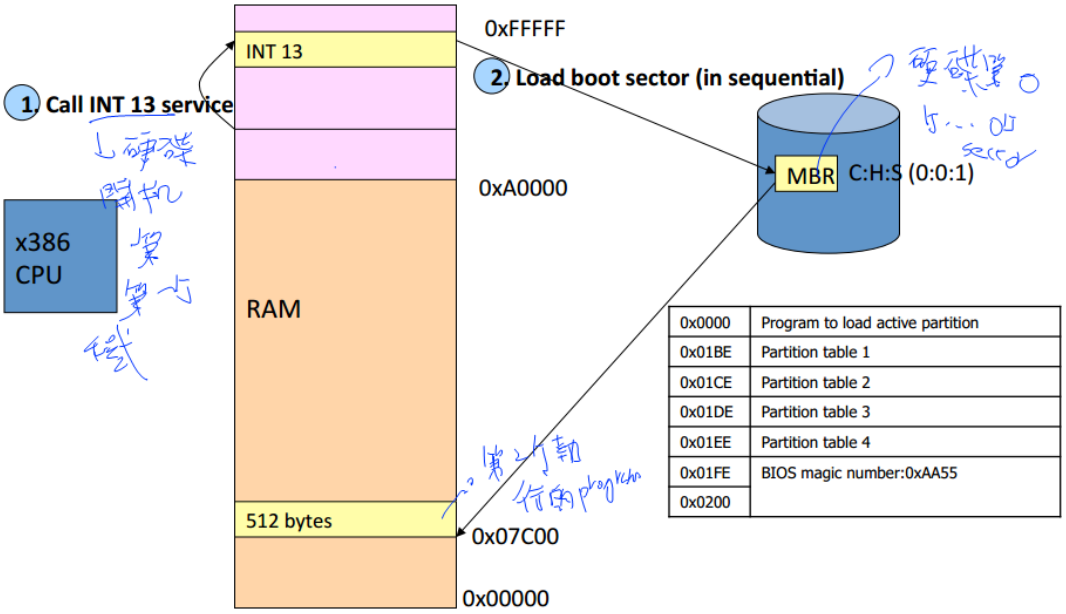
- **Booting process:**
- **multi_boot:**

當電腦電源開啟時，boot-loader 會 loads OS into memory，是 CPU 執行的第一組程式，CPU 會從 bios 去 fetch、execute 此程式：需參照 CS 所存取的 address，CS:IP 是指向目前執行程式在記憶體中所在的區域，讓 BIOS 可從 0XFFFF0 的第一條指令位址開始執行，而在載入 OS 的工作前，必須先準備 Vector table 和 ISR。



，當執行完畢 PC counter 會指向別的 address 進行開機程式（第二組程式）。[BIOS 是開機只執行一次的程式，所以不必放在 cache 裡。Boot loader 的流程如下圖。] 此時 OS 會把 disk 中的 bootsec [在硬碟的第 C:H:S(0:0:1) 第 0 磁軌第 0 磁頭第 1 磁區:稱為開機磁區]copy 置記憶體 0x07C00 處[512bytes]，

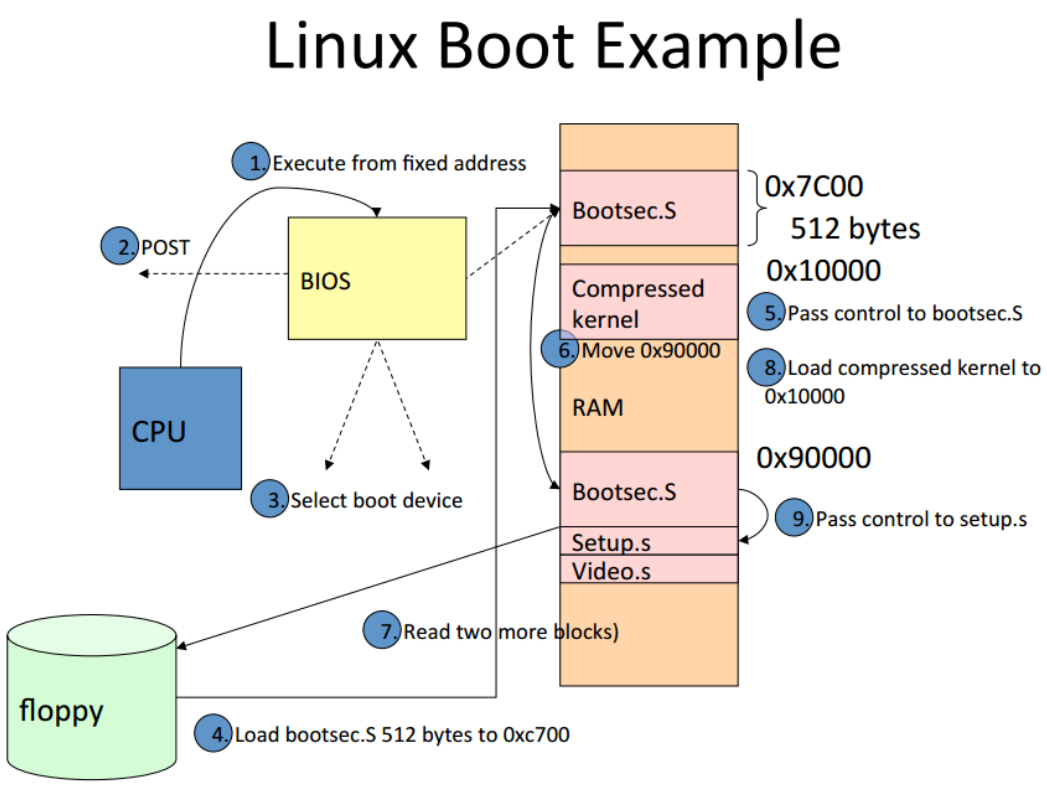
PC Booting (Cont)



當 os 被 boot-loader load 進去時，會一邊讀 kernel image，一邊自行解壓縮。
而在 Linux kernel 中，為避免 OS 過於肥大，會將 kernel image、和 root disk（含 driver）分開存放。

1. Bootsec 第一個工作是做記憶體的規劃
2. 複制 bootsec，從 0x07c00 複製至 0x90000 處：0x07c00 是為了不同 manufacture 有識別作用，當 bootsec 將自已移至 0x90000 處，OS 即可依照自已需求安排記憶體
3. 將 Setup 程式載入到記憶體

等 bootsect 執行完，setup 就開始工作: 載入系統資料

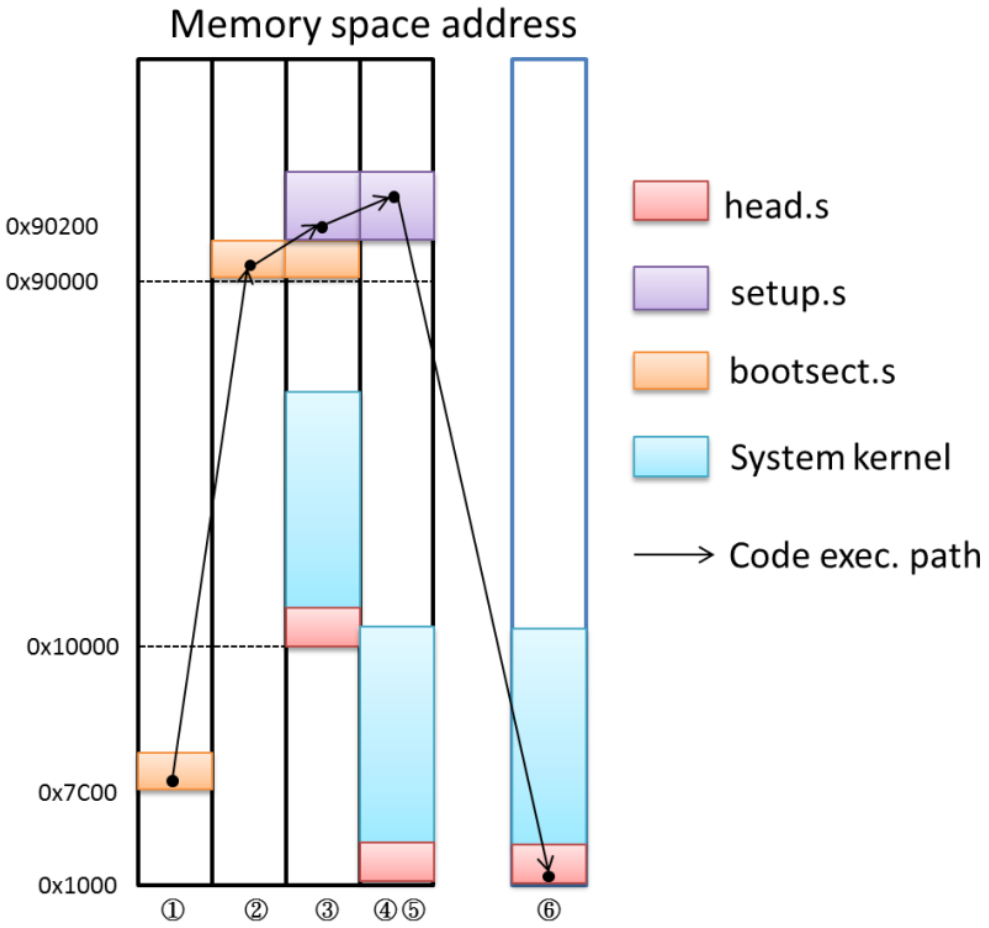


關中斷並將 system 移動到記憶體位址起始位置 0x00000
:在 32bit mode 工作，並為呼叫 main 作準備

Setup 將 0x10000 的核心程式 copy 至 0x00000 處
當 system 復制到 0x00000 後，將 disable BIOS 的中斷表
Kernel 即可開始使用自已的中斷表。

執行 head.s

Kernel booting



LAB3

- **Interrupt & System call**
- **generating entry points for the different traps**
- **set IDT table operation**
- **Check the trap number and call the interrupt handler**
- **push registers to build a trap frame therefore make the stack look like a struct trapframie**
    - **setup the contents of IDT, you will let CPU know where is the IDT by using 'lidt' instruction**
- **ld script**

背景知識：

同一行指令在 kernel-space speed > user-space

CPU 根據 trap 的原因，調用適當的 trap handler function：

interrupt 被 enable 時，os 會經由查表找到相對應的處理程式(ISR: Intrrupt Service Rountine, 即 interrupt handler)，並中斷原本在執行的程式，讓 cpu 執行 ISR 的內

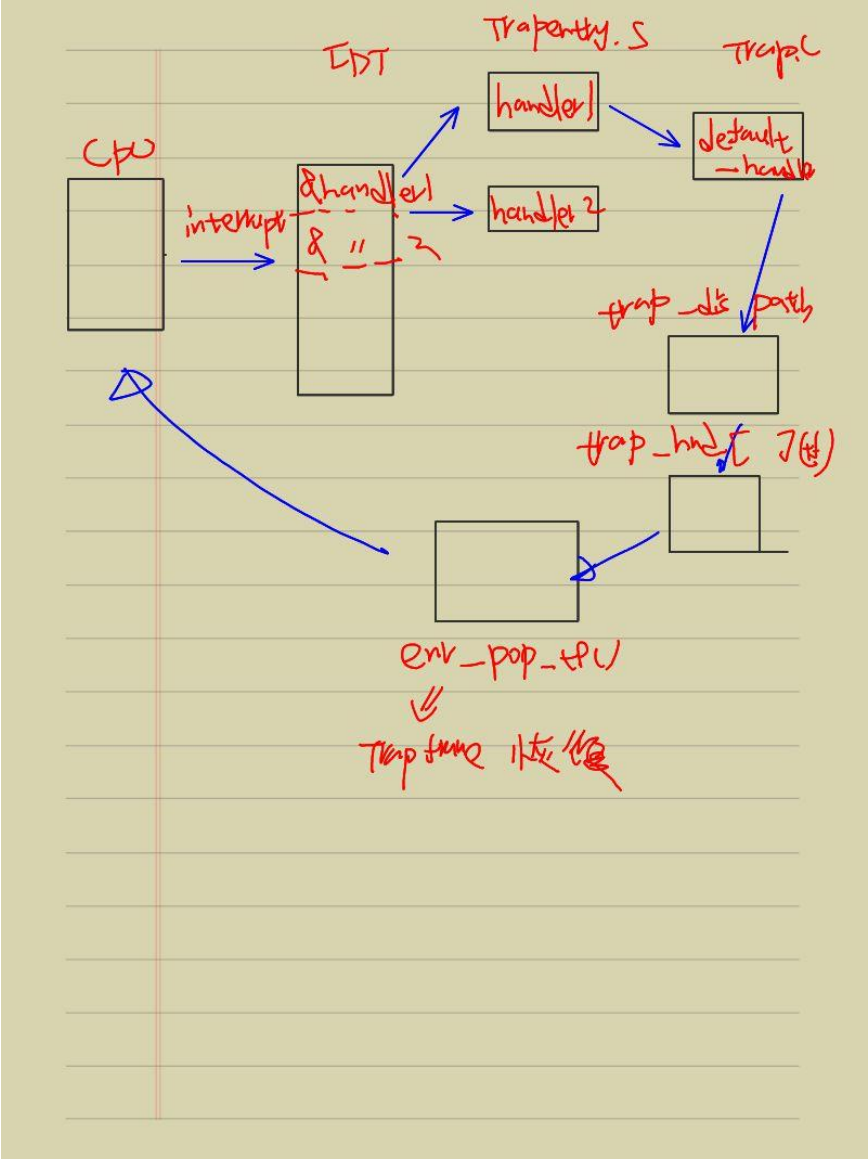
容，之後才再轉回原來的程式中。過程如下：
當中斷發生時，CPU 會自動切換到一個新的 stack，然後自動地將當前運行程式的 EFLAGS registers push 到 stack 中，然後壓入 CS 和 IP ，之後才會轉到 ISR 程序的第一道指令：即 handler 中

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

我們並用 TRAPHANDLER(name, num) 來為不同的 trap handler 命名，和傳入 trap number (defines a globally-visible function for handling a trap
, it pushes a trap number onto the stack, then jumps to _alltraps)
此處我們使用 TRAPHANDLER_NOEC：handler 不推入 Error_code (須推入一個 0 用來佔位，才推入 trap number), 執行到 __alltraps 會進入 default_trap_handler.. ，這是任一種 trap handler 都一樣的程式，作用是依然繼續補充 trap frame 的結構，push stack 的順序和 trap.h 所定義的 trapframe 結構一樣。至此保存現場的工作完成

```
/* ISRs */
TRAPHANDLER_NOEC(Default_ISR, T_DEFAULT)

/* Lab3 TODO: Setup timer and keyboard's ISR entry point */
TRAPHANDLER_NOEC(t_irqtimer_lbl,IRQ_OFFSET+IRQ_TIMER)
TRAPHANDLER_NOEC(t_irqkbd_lbl,IRQ_OFFSET+IRQ_KBD)
```



在 physical memory 中，會使用 MMU 來記錄 memory 的使用情況，並由於記憶體無法片斷的擺放，會產生 internal fragment[要五毛給一塊]. 所以要額外 design 一個機制，來處理此情形發生：少人用小的會議室，多人用大的會議室

Virtual memory：

1． 當 fetch instruction →decode→知道去一個 virtual address 去取值，MMU 會有個 mapping logical: physical address 的表，並由 MMU 告知 CPU 。
由此可知 fetch instruction，要先 access logical entry 再 access physical memory 至少要 2 次 MMU[TLB]的存取動作
→可提供改善的方法：先找 cache[TLB]再找 memory
TLB hit. If the requested address is not in the TLB, it is a miss, and the translation

proceeds by looking up the page table in a process called a *page walk*

On a context switch, some TLB entries can become invalid, since the virtual-to-physical mapping is different→ cpu context-switch 後，TLB 的 entries miss 掉，所以找到正確的 address 會花較久時間→跑較慢

Page fault handler：

Swap page 的策略：1。最不常用的 2。剛用的 做 swap out

Kernel 、user 的 address space

Kernel：不用 page table (physical + logical)

User：要用 page table(logical 連續)

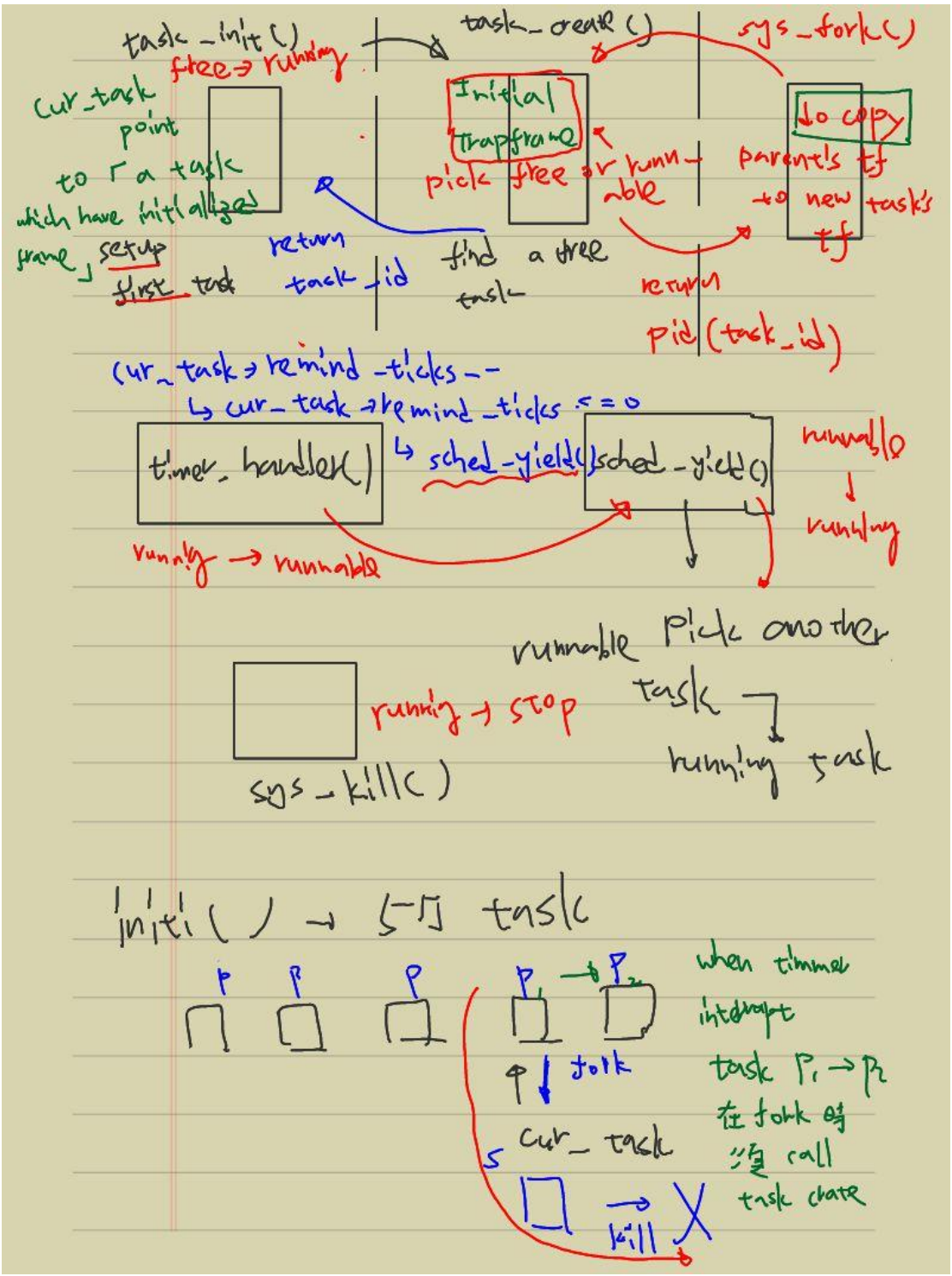
而 linux kernel code 禁止 swap 任何一個

LAB4:

Interrupt 、task create、round robin scheduling、system fork actions

見下圖

背景知識：OS 的 scheduler 並不偏好 context-switch，但 scheduler 記錄不同的 program run 多久，必須的資料結構，會使其 O.S 的 size 增長，當增加一個 process PCB 就會增加。在開檔、由於只能限制於同一 user，為避免 process 轉成 waiting state 無人將其恢復為 busy state，此時就需要 scheduler 的處理。



LAB5

- **Lock: to protect critical section**

  Protected critical section：

  當兩個 cpu 都是 memory intensive 不見得有好處

  CPU speed >> memory speed，而有好處的前提是有 cache

Multi-thread 可使 code 共享，讓 memory 的使用更有效率。
。而由於有共享的變數，需要做 lock 的動作[一次只放行一個 user 使用存取]，且只 lock 部分 code 才有效率。且由於像是 local variable 是在 stack 中不用保護，但在 global variable (heap)中則需要。
而 lock 有十多種，舉例兩種：
Spin lock：等得不久的，沒事的，急的
Sleep lock：等得象的，有事的，不急的