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/

http://wap.yesky.com/gameonline/405/11790405.shtml

作業四：系統呼叫

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

- 可以將Linux kernel當成是運行於privileged mode的『函數庫』，如果要使用這個函數庫，必須使用『硬體提供的特殊指令』
  - X86為例，普通函數庫使用『call』和『ret』
  - X86為例，呼叫Linux kernel要用『syscall』和『sysret』
- 傳遞給核心的參數有哪些？
- 如果是『複雜的參數』（例如：buffer），要怎樣傳遞給核心？
- 負責助教：
  - 請看網頁

Linux的參數傳遞方式

- 直接用暫存器傳遞
  - 例如：system call的編號
  - 簡單的變數（例如：資料長度）
- 使用暫存器傳遞「指標」
  - 例如：指向buffer的指標
  - 指向「其他參數」的指標（通常用於暫存器極少的處理器上，如：x86-32）
- 通常傳遞「system call」編號的暫存器，也是回傳值的暫存器
- 注意：system call的回傳值不一定就是「對應的libc」的回傳值，例如：libc可能會將錯誤進行額外的處理，如：errno

32/64位元對system call的影響

- 很多情況下32位元的system call並無法直接套用到64位元的電腦
- 從32位元邁向64位元的主要目的是：增加定址空間
  - 記憶體的定址空間
  - 檔案的定址空間（例如：lseek）
- 32位元作業系統無法『非常彈性、效率的』使用超過2GB的記憶體（kernel/user space各佔一半），此外也無法『非常有效率的』處理2GB以上的檔案（因為lseek的offset帶正負號）
- 基於上處理由，64位元的作業系統會重新設計一些參數，已加大定址空間等

注意：「int」的長度

- 在C語言中並沒有規定「int」的表示方式（例如：ones' complement、two's complement)
- 也沒規定int的長度
- 某些compiler將「int」定義為：該CPU上「符合暫存器長度」的「帶正負號數」
- gcc在x86-64上，「int」為「32bit」的正負符號整數
  - -2147483648 ~ 2147483647
- 『不可以』假設「pointer」和「int」的長度是一樣的

32位元時代的system call

```
1.  char* hello = "hello world\n";
2.  int len = strlen(hello)+1;
3.  long ret;
4.  printf("使用 'int 0x80' 呼叫system call\n");
5.  __asm__ volatile (
6.    "mov $4, %%rax\n" //write是第4號system call
7.    "mov $2, %%rbx\n" //stdout
8.    "mov %1, %%rcx\n" //buffer
9.    "mov %2, %%rdx\n" //buffer size
10.   "int $0x80\n"    //發出system call
11.   "mov %%rax, %0"  //system call的回傳值放在ax
12.   : "=m"(ret)
13.   : "g" (hello), "g" (len)
14.   : "rax", "rbx", "rcx", "rdx");
15. printf("回傳值是: %ld\n", ret);
```

64位元時代

```
1.  char* hello_tc = "全世界，你好\n";
2.  long len_tc = strlen(hello_tc)+1; //注意我宣告為long，因為long是64位元
3.  printf("使用 'syscall' 呼叫system call\n");
4.  __asm__ volatile (
5.    "mov $1, %%rax\n" //write是第1號system call
6.    "mov $2, %%rdi\n" //stderr
7.    "mov %1, %%rsi\n" //buffer
8.    "mov %2, %%rdx\n" //buffer size
9.    "syscall\n"       //使用syscall比int 0x80快
10.   "mov %%rax, %0"   //system call的回傳值依然放在AX
11.   : "=m"(ret)
12.   : "g" (hello_tc), "g" (len_tc)
13.   : "rax", "rbx", "rcx", "rdx");
14.   printf("回傳值是：%ld\n", ret);
```

輸出結果

```
ubuntu@oslab:~/os-lab/sharedFolder/hw03$ ./syscall
使用 'int 0x80' 呼叫system call
hello world
回傳值是：13
使用 'syscall' 呼叫system call
全世界，你好
回傳值是：20
```

system call在x86-32/64的形式

- 底下是32位元system call的編號及參數順序
  - https://syscalls.kernelgrok.com/
- 底下是64位元的system call的編號和參數傳遞的順序
  - https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- 底下是Google寫的system call table
  - https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit
  - https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

作業

- 使用組合語言呼叫system call，從stdin讀進一個字元
- 繳交項目
  - Makfile 和 必要的 C 檔案，執行檔案名稱為：hw3
  - 一份簡單的報告，請將你的程式碼反組譯，然後「大致」解釋組語的意義（例如：使用gdb內的 disass /m main）
  - 報告的名稱為：hw3.pdf
  - 上述文件請壓縮成「學號.tar.bz2」，例如：687410007.tar.bz2
  - 繳交到ecourse2上

附錄

X86的特權指令

- http://www.brokenthorn.com/Resources/OSDev23.html
- 注意，右表中的RDTSC歸類是：特權指令。
- 存取核心資訊的，『應該要』算是特權指令

Privileged Level Instructions	
Instruction	Description
LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
LTR	Loads a Task Register into TR
MOV *Control Register*	Copy data and store in Control Registers
LMSW	Load a new Machine Status WORD
CLTS	Clear Task Switch Flag in Control Register CR0
MOV *Debug Register*	Copy data and store in debug registers
INVD	Invalidate Cache without writeback
INVLPG	Invalidate TLB Entry
WBINVD	Invalidate Cache with writeback
HLT	Halt Processor
RDMSR	Read Model Specific Registers (MSR)
WRMSR	Write Model Specific Registers (MSR)
RDPMC	Read Performance Monitoring Counter
RDTSC	Read time Stamp Counter

```
1.  #include <stdio.h>
2.  int main() {
3.      int a;
4.      int* ptr = &a;
5.      asm("xor %rax, %rax\n");   //第一種寫法，簡單
6.      __asm("xor %%rax, %%rax\n":::);   //第二種寫法，彈性
7.      asm("hlt\n");          //停止ＣＰＵ運行。這一行會因為「使用者模式」無法執行
8.      __asm(                 //刷掉ＭＭＵ內的快取，這一行也會因為「使用者模式」無法執行
9.          "INVLPG %0\n"
10.         :
11.         :"g" (ptr)
12.         :
13.     );
14. }
```