

教育部顧問室嵌入式軟體聯盟
實驗模組整合與單一平台建置計畫

實驗模組名稱：Linux Exception

開發教師：曹孝櫟、陳鵬升

開發學生：黃彥筑、許展富、王冠人

學校系所：交通大學資訊工程學系

中正大學資訊工程學系

聯絡電話：05-2720411 Ext. 16650

聯絡地址：62102 嘉義縣民雄鄉大學路一段 168

號 資訊工程學系

實驗平台：Raspberry Pi

實驗內容關鍵字：Linux Exception

● 實驗目的：

利用 Linux 來建立 embedded system 是常見的應用。例外處理又稱為軟體中斷，相對於任何時刻可能發生的硬體中斷，軟體中斷是在 CPU 執行指令過程系統發生異常而產生的，也可以透過軟體的控制產生例外。本章目的是讓大家學習在 ARM-Linux 上的例外處理。我們的實驗步驟分成三部份：

- 第一個是了解例外處理是如何被初始化。
- 第二個是了解系統呼叫使用的中斷是如何運作。
- 第三個是要學習寫一個系統呼叫。

● 實驗器材：

1. PC x 1
 - Requirement: any modern PC will do.
 - Purpose: To provide compile and debug environment.
2. Raspberry Pi x 1
 - Development target
3. microSD card x 1

● 實驗所需資源：

1. PC x 1
 - Linux Ubuntu
 - ◆ ARM cross compiler
2. Raspberrry Pi x 1

目錄

Part 1 – 例外處理.....	3
Step 1: 例外初始化.....	3
Step 2: 編譯核心.....	4
Part 2– 系統呼叫運作.....	6
Step 1: 寫一個使用系統呼叫的程式.....	6
Step 2: Cross Compiler.....	6
Step 3: ARM 的 SVC 指令	7
Part 3– 系統呼叫(System Call)	8
Step 1: 修改系統檔案.....	8
Step 2: Add define of system call in unistd.h.....	8
Step 3: 撰寫欲新增的 system call 的內容.....	9
Part 4– 寫一個使用者應用程式，來測試新增的系統呼叫.....	11
Step 1: write user program to call mysyscall.	11
Step 2: 使用 Cross Compiler 編譯使用者程式 mytestsys.c.....	11
Step 3: 在 Raspberry Pi 上執行	12

Part 1 – 例外處理

不管是中斷或是例外的發生都會使得 CPU 改變指令的執行順序。相對於外部產生中斷可能在任何時間發生，例外是隨著 CPU 的時脈，執行過程中，產生需要例外處理的事件，因此跳到例外處理程式做對應的處理，故例外處理也被稱為同步中斷。

例外的產生主要有兩種，一種是因為系統內部執行指令過程中，發生不正常的情況，可能造成 Fault、Trap、甚至是 Abort 的情況；一種是來自程式碼本身產生的例外，例如：系統呼叫還有事件處理的應用。在 Linux 系統上，例外和中斷所使用的是相同的向量表。發生例外的時候，也是像處理中斷一樣，跳到處理例外的處理程式。**Linux 為了讓用戶層可以使用核心層資源，提供系統呼叫介面防止使用者直接做出傷害系統的動作，透過系統呼叫介面使系統提供服務可增加系統的安全性。**

由於使用者無法直接存取核心的定址空間，必須先通知**核心代理**執行想要的功能，必須有通知核心的機制，Linux 使用產生軟體中斷的方式產生例外，使得系統切換到核心層，並且執行例外處理程式。ARM 處理器是透過 SVC（或 SWI）的指令來實作系統呼叫。當這個指令被執行的時候，CPU 會從使用者模式跳到核心模式，而且會執行對應的系統呼叫。**實驗內容主要包含觀察例外初始化及系統呼叫實作。**

Step 1: 例外初始化

<說明>系統初始化

<實驗步驟>

1. 下載 Linux kernel source code

```
$ git clone --depth=1 https://github.com/raspberrypi/linux -
b rpi-4.9.y
```

2. 在<Linux kernel source code>/init/main.c 的 start_kernel 函式中加入一個 printk 敘述。
(Linux Host 的<Linux kernel source code>/init/main.c)

```
/* Modify*/
printk("Initialize traps\n");
/*Modify*/
trap_init();
```

3. 在<Linux kernel source code>/arch/arm/kernel/traps.c 的 trap_init 函式中加入一個 printk 敘述。

(Linux Host 的<Linux kernel source code>/arch/arm/kernel/traps.c)

```
/*Modify*/
printk("arm: system call handler initialization -> see assembly code\n");
/*Modify*/
```

Step 2: 編譯核心

<說明>

```
$ cd <linux kernel source 目錄>
```

```
$ make ARCH=arm bcm2709_defconfig
```

請確認 arm-linux-gnueabihf-gcc 執行檔有在 PATH 路徑裡。

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bzImage
```

你可以在<linux kernel source 目錄>/arch/arm/boot 目錄下，找到 zImage。
將 zImage 複製到 microSD 卡的第一個 partition。複製的方法可以參考“建立簡易 Linux 之實驗模組”之 Part 6。

將系統開機，透過 dmesg 可以看到當系統初始的時候所做的系統初始化的呼叫，其結果如下圖。

<實驗步驟>

```
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.9.80-v7+ (pschen@SmallTurtleLinux1) (gcc version 4.9.3 (crosstool-NG crosstool-ng-1.22.0-88-g8460611) ) #3
SMP Mon Apr 16 14:37:32 CST 2018
[ 0.000000] CPU: ARMv7 Processor [410fd034] revision 4 (ARMv7), cr=10c5383d
[ 0.000000] CPU: div instructions available: patching division code
[ 0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[ 0.000000] OF: fdt:Machine model: Raspberry Pi 3 Model B Rev 1.2
[ 0.000000] cma: Reserved 8 MiB at 0x39400000
[ 0.000000] Memory policy: Data cache writealloc
[ 0.000000] On node 0 totalpages: 236544
[ 0.000000] free_area_init_node: node 0, pgdat 80c6f440, node_mem_map b8bd7000
[ 0.000000] Normal zone: 2079 pages used for memmap
[ 0.000000] Normal zone: 0 pages reserved
[ 0.000000] Normal zone: 236544 pages, LIFO batch:31
[ 0.000000] percpu: Embedded 14 pages/cpu @b8b8c000 s25600 r8192 d23552 u57344
[ 0.000000] pcpu-alloc: s25600 r8192 d23552 u57344 alloc=14*4096
[ 0.000000] pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
[ 0.000000] Built 1 zonelists in Zone order, mobility grouping on. Total pages: 234465
[ 0.000000] Kernel command line: 8250.nr_uarts=1 bcm2708_fb.fbwidth=1920 bcm2708_fb.fbheight=1080 bcm2708_fb.fbswap=1
vc_mem.mem_base=0x3ec00000 vc_mem.mem_size=0x40000000 root=/dev/mmcblk0p2 rootwait console=tty1 console=ttyAMA0,115200
[ 0.000000] PID hash table entries: 4096 (order: 2, 16384 bytes)
[ 0.000000] Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
[ 0.000000] Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
[ 0.000000] Initialize traps
[ 0.000000] arm: system call handler initialization -> see assembly code
[ 0.000000] Memory: 915976K/946176K available (7168K kernel code, 486K rwddata, 2012K rodata, 1024K init, 770K bss, 22008K reserved,
8192K cma-reserved)
[ 0.000000] Virtual kernel memory layout:
```

```

[ 0.000000] vector : 0xffff0000 - 0xffff1000 ( 4 kB)
[ 0.000000] fixmap : 0xffc00000 - 0xffff0000 (3072 kB)
[ 0.000000] vmalloc : 0xba000000 - 0xff800000 (1112 MB)
[ 0.000000] lowmem : 0x80000000 - 0xb9c00000 ( 924 MB)
[ 0.000000] modules : 0x7f000000 - 0x80000000 ( 16 MB)
[ 0.000000] .text : 0x80008000 - 0x80800000 (8160 kB)
[ 0.000000] .init : 0x80b00000 - 0x80c00000 (1024 kB)
[ 0.000000] .data : 0x80c00000 - 0x80c798fc ( 487 kB)
[ 0.000000] .bss : 0x80c7b000 - 0x80d3b9a4 ( 771 kB)
[ 0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
[ 0.000000] Hierarchical RCU implementation.
[ 0.000000] Build-time adjustment of leaf fanout to 32.
[ 0.000000] NR_IRQS:16 nr_irqs:16 16
[ 0.000000] arm_arch_timer: Architected cp15 timer(s) running at 19.20MHz (phys).
...
```

Part 2- 系統呼叫運作

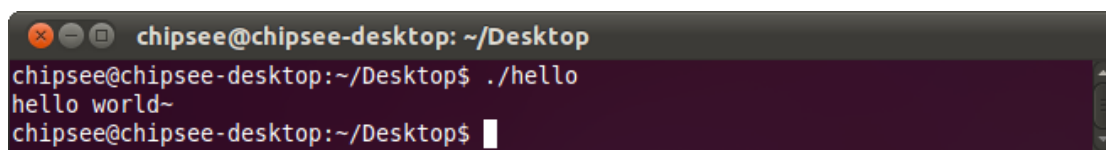
Step 1: 寫一個使用系統呼叫的程式

<說明>

一般 I/O 都是透過系統呼叫，請核心代為操作，所以使用簡單的 printf 程式，觀察其組合語言，找到 SVC (or SWI)指令。

<實驗步驟>

```
#include <stdio.h>
int main (void) {
    printf("hello world~\n");
    return 0;
}
```

A terminal window with a dark purple background. The prompt is 'chipsee@chipsee-desktop: ~/Desktop'. The user enters './hello' and the output is 'hello world~'. The prompt returns to 'chipsee@chipsee-desktop: ~/Desktop\$' with a cursor.

```
chipsee@chipsee-desktop: ~/Desktop
chipsee@chipsee-desktop:~/Desktop$ ./hello
hello world~
chipsee@chipsee-desktop:~/Desktop$
```

Step 2: Cross Compiler

<說明>

由於開發平台在 PC 上，卻要在 ARM 架構上執行，必須透過 Cross compiler 方式使編譯後之程式可以在 ARM 板子上執行。所以我們需要選擇 target 為 ARM 架構的 gcc 來做編譯。因為我們所要觀察的組合語言會使用到其他函式，所以必須使用 static link 方式來編譯程式，以利觀察。

<實驗步驟>

1. 編譯程式

```
$ arm-linux-gnueabihf-gcc -static hello.c -o hello.exe
```

2. 反組譯程式碼

```
$ arm-linux-gnueabihf-objdump -d hello.exe > assembly
```

可以得到一個很大的 assembly 檔案，然而要觀察的只是其中在做 I/O 使用

read/write 前的 open 可以找到 SVC 這個指令。

```

pschen@SmallTurtleLinux1: ~/RaspberryPi3/test
2726c: e49d7004 pop {r7} ; (ldr r7, [sp], #4)
27270: e3700a01 cmn r0, #4096 ; 0x1000
27274: 312fff1e bxcc lr
27278: ea000e18 b 2aae0 <__syscall_error>
2727c: e1a00000 nop ; (mov r0, r0)

00027280 <__libc_open>:
27280: e59fc060 ldr ip, [pc, #96] ; 272e8 <__libc_open+0x68>
27284: e79fc00c ldr ip, [pc, ip]
27288: e33c0000 teq ip, #0
2728c: e52d7004 push {r7} ; (str r7, [sp, #-4]!)
27290: 1a000005 bne 272ac <__libc_open+0x2c>
27294: e3a07005 mov r7, #5
27298: ef000000 svc 0x00000000
2729c: e49d7004 pop {r7} ; (ldr r7, [sp], #4)
272a0: e3700a01 cmn r0, #4096 ; 0x1000
272a4: 312fff1e bxcc lr
272a8: ea000e0c b 2aae0 <__syscall_error>
272ac: e92d400f push {r0, r1, r2, r3, lr}
272b0: eb000816 bl 29310 <__libc_enable_asynccancel>
272b4: e1a0c000 mov ip, r0
272b8: e8bd000f pop {r0, r1, r2, r3}

```

Step 3: ARM 的 SVC 指令

<說明>

在新的 ARM 處理器中，使用 SVC 指令取代原先的 SWI 指令，SVC 是軟體中斷的指令，這個指令會執行記憶體位址 0x08 所存放的指令，改變處理器的執行模式，並且儲存要執行系統呼叫前的暫存器，然後透過後面緊跟著的值，找出對應的系統呼叫。

Part 3— 系統呼叫(System Call)

<說明>

系統呼叫可以擁有系統的權限，存取系統內部保護的資料，所以利用新增系統呼叫的方式，可以做到變更系統內部的事情。以下將介紹如何新增一個系統呼叫。

Step 1: 修改系統檔案

<說明>

為了增加系統呼叫，必須在系統呼叫表定義所欲增加的 system call。修改 calls.S 找到目前的最後一個 system call，在後面加上我們的 system call。

<實驗步驟>

1. (Linux Host 的<Linux kernel source code>/arch/arm/kernel/calls.S)

```
/* 390 */      CALL(sys_mlock2)
                CALL(sys_copy_file_range)
                CALL(sys_preadv2)
                CALL(sys_pwritev2)
                CALL(sys_pkey_mprotect)
/* 395 */      CALL(sys_pkey_alloc)
                CALL(sys_pkey_free)
/* Modify */
                CALL(sys_mysyscall)
/* Modify */
```

Step 2: Add define of system call in unistd.h

<說明>

為了讓使用者程式可以呼叫，我們需要建立一個系統呼叫的序號，修改 unistd.h 找到目前的最後一個 system call，新增 mysyscall，代碼就是原本有的最後一個號碼+1。

<實驗步驟>

(Linux Host 的<Linux kernel source code>/arch/arm/include/uapi/asm/unistd.h)

```
#define __NR_mlock2                ( __NR_SYSCALL_BASE+390)
#define __NR_copy_file_range      ( __NR_SYSCALL_BASE+391)
#define __NR_preadv2              ( __NR_SYSCALL_BASE+392)
#define __NR_pwritev2             ( __NR_SYSCALL_BASE+393)
#define __NR_pkey_mprotect        ( __NR_SYSCALL_BASE+394)
#define __NR_pkey_alloc           ( __NR_SYSCALL_BASE+395)
#define __NR_pkey_free            ( __NR_SYSCALL_BASE+396)

/* Modify */
#define __NR_mysyscall            ( __NR_SYSCALL_BASE+397)
/* Modify */
```

Step 3: 撰寫欲新增的 system call 的內容

<說明>

為了系統呼叫可以適當地被連結，需要在函數前面加上 `asmlinkage`，並在函數名稱前加入 `sys_`。由於要讓編譯器認出 `asmlinkage` 與 `printf`，必須要加入 `#include<linux/linkage>` 及 `#include <linux/kernel.h>`。

以下是放在<Linux kernel source code>/arch/arm/kernel

<實驗步驟>

1.

(Linux Host 的<Linux kernel source code>/arch/arm/kernel/mysyscall.c) (新增)

```
#include<linux/linkage.h>
#include <linux/kernel.h>

asmlinkage void sys_mysyscall(int a, char *b){
    printf("system call ...\n");
    printf("int1:%d, string:%s in kernel\n", a, b);
}
```

2.

(Linux Host 的<Linux kernel source code>/include/linux/syscalls.h) (修改)

將函式宣告加到最後面

```
asm linkage void sys_mysyscall(int a, char* b);
```

3.

修改在<Linux kernel source code>/arch/arm/kernel 下的 Makefile，其中的 obj-y := compat.o entry-armv.o ...，在這行最後面加入 mysyscall.o。

(Linux Host 的<Linux kernel source code>/arch/arm/kernel/Makefile)

```
obj-y      := elf.o entry-common.o irq.o opcodes.o \  
              process.o ptrace.o reboot.o return_address.o \  
              setup.o signal.o sigreturn_codes.o \  
              stacktrace.o sys_arm.o time.o traps.o mysyscall.o
```

最後重新編譯 kernel，並將新產生的 zImage 複製到 Raspberry Pi 的 microSD 卡上。

Part 4- 寫一個使用者應用程式，來測試新增的系統呼叫

<說明>

Step 1: write user program to call mysyscall.

<說明>

<實驗步驟>

```
/* filename: mytestsys.c */  
  
#include <sys/syscall.h>  
#include <linux/unistd.h>  
  
#define __NR_mysyscall (__NR_SYSCALL_BASE+397)  
#define mysyscall(a,b) syscall(__NR_mysyscall, (a), (b))  
  
int main(){  
    mysyscall(14,"system_call_lab");  
    return 0;  
}
```

Step 2: 使用 Cross Compiler 編譯使用者程式 mytestsys.c

<說明>

<實驗步驟>

```
% arm-linux-gnueabihf-gcc -I<linux kernel source code>/include/  
-static -g mytestsys.c -o mytestsys.exe
```

將 mytestsys.exe 複製到 microSD 卡的第二個 partition。

Step 3: 在 Raspberry Pi 上執行

<說明> 使用 dmesg 查看輸出訊息。

<實驗步驟>

```
...
[ 3.521387] input: Wired USB Keyboard as /devices/platform/soc/3f980000.usb/usb1/1-1/1-1.2/1-1.2:1.0/0003:0461:4E6E.0001/input/input0
[ 3.611951] hid-generic 0003:0461:4E6E.0001: input,hidraw0: USB HID v1.10 Keyboard [Wired USB Keyboard] on usb-3f980000.usb-1.2/input0
[ 3.643225] input: Wired USB Keyboard as /devices/platform/soc/3f980000.usb/usb1/1-1/1-1.2/1-1.2:1.1/0003:0461:4E6E.0002/input/input1
[ 3.731491] hid-generic 0003:0461:4E6E.0002: input,hidraw1: USB HID v1.10 Device [Wired USB Keyboard] on usb-3f980000.usb-1.2/input1
[ 4.173736] smsc95xx 1-1.1:1.0 eth0: hardware isn't capable of remote wakeup
[ 5.575891] smsc95xx 1-1.1:1.0 eth0: link up, 100Mbps, full-duplex, lpa 0xC1E1
[ 16.304343] random: crng init done
[ 43.731939] system call ...
[ 43.742471] intl:14, staring: system_call_lab in kernel
```

我們所增加的 system call 有被執行到。