

CSC0016 - Advanced Operating Systems

HW2_Add a System Call in Linux Kernel 2.6.26

NTUST_M11207521 陳俊博

一、系統環境和前言

在軟體環境中，使用 VirtualBox 7.0.18 作為虛擬化平台，並在其中運行 Ubuntu 14.04.6 LTS (Trusty Tahr) 作業系統。虛擬機器配置為使用 4 顆虛擬 CPU 和 8GB 記憶體，並設定了 50GB 的虛擬硬碟。虛擬機環境中使用的 Linux 核心版本為 4.4.0-142-generic，並透過 GCC 4.8.4 和 GNU Make 3.81 進行系統的編譯與建置。在進行修改核心過程中，由於嘗試在 Linux 2.6.32 版本中新增 system call 時多次失敗，最終決定將版本降級至 Linux 2.6.26。這個較早的版本能夠讓我順利完成這次作業，解決了在 2.6.32 版本中遇到的相容性和 system call 實作上的問題。

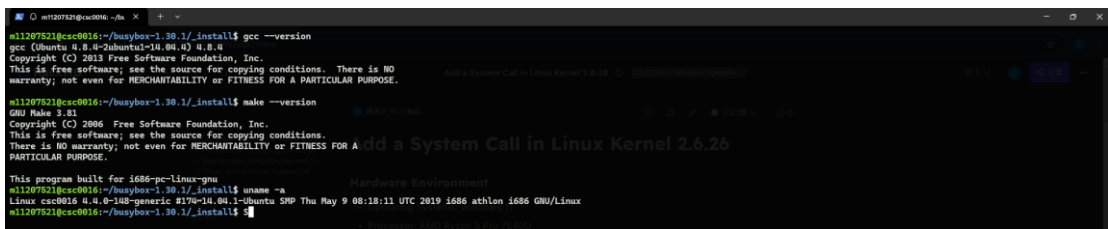


圖 1 VirtualBox 環境

二、核心編譯

在進行 Linux 核心的開發或修改前，首先需要準備開發環境並編譯所需的核版本。以下步驟將說明如何下載並編譯 Linux 2.6.26 核心，同時解決編譯過程中可能會碰到的錯誤。

i. 下載與解壓縮核心

安裝開發工具和相關的必備套件，並下載所需的 Linux 核心源代碼以進行編譯。

1. 首先需要安裝編譯核心所需的套件。執行以下命令來安裝基本的編譯工具集（如 `gcc` 和 `make`），以及核心配置時所需的 `libncurses` 和模擬器 `QEMU`。執行 `sudo apt-get install build-essential libncurses5-dev qemu`。
2. 接下來下載 Linux 2.6.26 的原始碼檔案，執行 `wget https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/linux-2.6.26.tar.gz`。
3. 完成下載後解壓縮檔案，執行 `tar zxvf linux-2.6.26.tar.gz`。

ii. 預設配置與編譯

配置核心並進行編譯。

1. 進入剛剛解壓縮的核心目錄，並使用 `make` 命令為 i386 架構進行預設配置。配置完後開始進行編譯：

```
~$ cd linux-2.6.26
```

```
~$ make ARCH=i386 defconfig
~$ make
```

編譯過程中可能會遇到一些錯誤，以下將介紹如何解決這些錯誤。

iii. 編譯過程中的錯誤及解決方法

Error 1: gcc: error: elf_x86_64: No such file or directory

解決編譯過程中的 ELF 格式錯誤，由於在核心中使用了不正確的編譯目標格式。需要修改核心中 `arch/x86/vdso/Makefile` 文件內的編譯選項：

- 打開 `arch/x86/vdso/Makefile` 文件，找到 `-m elf_x86_64` 和 `-m elf_i386`。
- 將 `-m elf_x86_64` 替換為 `-m64`，將 `-m elf_i386` 替換為 `-m32`。

Error 2: undefined reference to '__mutex_lock_slowpath'

解決核心編譯過程中的未定義引用錯誤。遇到此錯誤時需要修改核心中的 `kernel/mutex.c` 文件：

1. 打開 `kernel/mutex.c` 文件，找到以下兩個函數：

```
static void noinline __sched
__mutex_lock_slowpath(atomic_t *lock_count);

static noinline void __sched __mutex_unlock_slowpath(atomic_t *lock_count);
```

2. 在 `static` 關鍵字後加入 `__used`，修改為：

```
static __used void noinline __sched
__mutex_lock_slowpath(atomic_t *lock_count);

static __used noinline void __sched __mutex_unlock_slowpath(atomic_t
*lock_count);
```

三、 準備根文件系統 BusyBox

在建立核心的測試環境時，通常需要一個基本的根文件系統來模擬作業系統的功能。在這個過程中會使用 BusyBox 來創建一個精簡的根文件系統，並將其包裝為 Ramdisk 映像檔，最終搭配核心在 QEMU 中運行。以下步驟將逐一說明如何完成。

i. 下載與解壓縮 BusyBox

下載 BusyBox 並進行解壓縮：

```
~$ wget https://busybox.net/downloads/busybox-1.30.1.tar.bz2
~$ tar jxvf busybox-1.30.1.tar.bz2
~$ cd busybox-1.30.1
```

ii. 配置編譯選項

設定編譯選項來確保編譯出能夠符合系統架構並滿足要求的 BusyBox 可執行檔案。請在配置介面中確認以下選項已正確設定：

```
## 使用 BusyBox 提供的預設設定
~$ make defconfig
## 然後啟動 menuconfig 介面進行進一步配置
~$ make menuconfig
```

- Settings
 - Build Options :
 - ◆ [*] Build static binary (no shared libs)：確保 BusyBox 使用靜態連結，這樣不依賴外部的共享庫。
 - Additional CFLAGS：設定為 `-m32 -march=i386`，確保 BusyBox 在 32 位元的架構下進行編譯。
 - Additional LDFLAGS：設置為 `-m32`，針對 32 位元系統。
 - What kind of applet links to install：選擇 `(X) as soft-links`。

iii. 編譯與安裝 BusyBox

編譯 BusyBox 並將其安裝到指定的 `_install` 目錄中，準備作為根文件系統使用。

```
~$ make
~$ make install
```

iv. 創建特殊設備文件

為了使 BusyBox 安裝的 `_install` 目錄能夠作為可啟動的根文件系統，需要創建一些設備文件。創建 `dev` 目錄並添加 `console` 和 `ram` 設備：

```
~$ mkdir -p _install/dev
~$ sudo mknod _install/dev/console c 5 1
~$ sudo mknod _install/dev/ram b 1 0
```

v. 創建 init 啟動程式

在根文件系統啟動時需要一個 `init` 程式來初始化系統並進行設定。先創建一個 `init` 文件並撰寫以下內容，最後更改可執行的權限。

```
~$ vim _install/init

#!/bin/sh
echo "### INIT SCRIPT ###"
echo "===== "
mkdir /proc /sys /tmp
mount -t proc none /proc
```

```

mount -t sysfs none /sys
mount -t tmpfs none /tmp
echo -e "\nBoot time: \033[1;32m$(cut -d' ' -f1 /proc/uptime) seconds\033[0m\n"
echo "=====
exec /bin/sh

~$ chmod +x _install/init

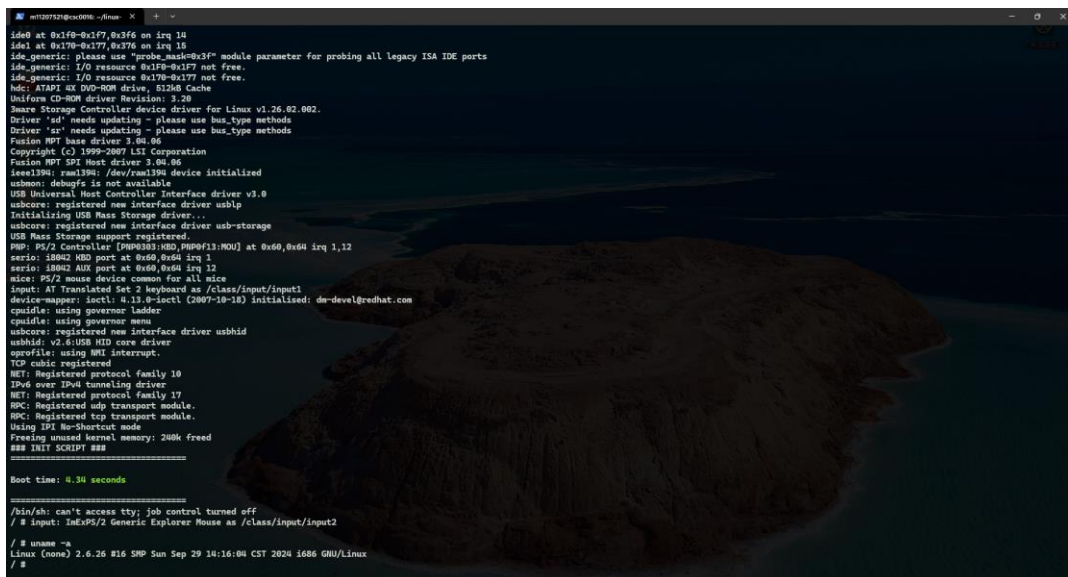
```

vi. 將文件系統打包為 Ramdisk 映像

將 BusyBox 安裝的根文件系統打包為一個可以被核心載入的 Ramdisk 映像檔。在 `_install` 目錄中執行以下命令，將當前文件系統打包為 `initramfs.cpio.gz`，執行 `find . -print0 | cpio --null -ov --format=newc | gzip -9 > ~/initramfs.cpio.gz`。

vii. 使用 QEMU 啟動核心

利用 QEMU 模擬器，搭配編譯好的核心和 Ramdisk 文件系統來啟動模擬系統。如果之前步驟沒有問題，現在可以啟動 QEMU 並載入 BusyBox 根文件系統，執行 `qemu-system-i386 -kernel arch/x86/boot/bzImage -initrd ~/initramfs.cpio.gz -append "console=ttyS0" -m 512 -nographic`。



```

[0] 0x1f0-0x1f7, 0x3f6 on irq 14
ide0 at 0x1f0-0x1f7, 0x3f6 on irq 14
ide_generic: please use "probe_mask=0x3f" module parameter for probing all legacy ISA IDE ports
ide_generic: I/O resource 0x1f0-0x1f7 not free.
ide_generic: I/O resource 0x1f0-0x1f7 not free.
hdc: ATAPI CD-ROM drive, 812KB Cache
Uniform CD-ROM driver Revision: 3.20
Sata Storage Controller device driver for Linux v1.26.02.002.
Driver 'sd' needs updating - please use bus_type methods
Driver 'sr' needs updating - please use bus_type methods
Fusion MPT base driver 3.00.00
Copyright (c) 1999-2007 LSI Corporation
Fusion MPT SPI Host driver 3.00.00
ieee1394: ram1394: /dev/ram1394 device initialized
usbmon: debugfs is not available
USB Universal Host Controller interface driver v3.0
usbcore: registered new interface driver usb-l
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
PMP: PS/2 Controller: [PM0303]HSD, [PM0313]MOU at 0x60, 0x64 irq 1, 12
serio: i8042 i8042 AUX port at 0x60, 0x64 irq 1
serio: i8042 AUX port at 0x60, 0x64 irq 12
mice: PS/2 mouse device common for all mice
input: AT Translated Set 2 keyboard as /class/input/input1
device-mapper: ioctl: 4.13.0-ioctl (2007-10-18) initialised: dm-devel@redhat.com
cpuidle: using governor ladder
cpuidle: using governor menu
usbcore: registered new interface driver usblid
usbhid: v2.6:USB HID core driver
oprofile: using NMI interrupt.
TCP cubic registered
NET: Registered protocol family 10
IPV6 over IP4 tunneling driver
NET: Registered protocol family 17
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
Using IP4 No-Shortcut mode
Freeing unused kernel memory: 240K freed
## INIT SCRIPT ##

Boot time: 4.34 seconds

/bin/sh: can't access tty: job control turned off
# Input: ImExPS/2 Generic Explorer Mouse as /class/input/input2

# uname -a
Linux (none) 2.6.26 #16 SMP Sun Sep 29 14:16:04 CST 2024 i686 GNU/Linux
#

```

圖 2 使用 QEMU 和 BusyBox 成功啟動 Kernel 2.6.26 系統

四、新增 System Call

在這一部分將介紹如何在 Linux 2.6.26 核心中新增一個簡單的 system call，並通過 QEMU 來進行測試。將涉及到對核心程式碼的修改、system call 的實作、對核心重新編譯以及測試程式的撰寫。

i. 新增 System Call 到 System Call Table

編輯 `linux-2.6.26/arch/x86/kernel/syscall_table_32.S`，新增以下：

```
.long sys_stud_id_syscall      /* 327 */
```

ii. 定義 System Call Number

編輯 `linux-2.6.26/include/asm-x86/unistd_32.h`，新增以下：

```
#define __NR_stud_id_syscall    327    /* It's my student id call */
```

注意！確保 system call number 和 system call table 中的編號一樣。

iii. 宣告新的 System Call

編輯 `linux-2.6.26/include/linux/syscalls.h`，新增以下：

```
asmlinkage long sys_stud_id_syscall(void);
```

iv. System Call 實作

在 `linux-2.6.26/kernel/` 目錄新增 `stud_id_syscall.c` 文件，並撰寫程式碼(附錄 A)。

v. 修改核心 Makefile

確保 system call 文件能夠被核心正確編譯，需要編輯 `linux-2.6.26/kernel/Makefile` 文件，並在 `obj-y` 後面加入 `stud_id_syscall.o` (如附錄 B)。

vi. 編譯內核

將核心重新編譯，確保新的 system call 被正確整合到核心中。

```
~$ make ARCH=i386
```

vii. 撰寫使用者層級測試程式

需要撰寫一個簡單的測試程式來呼叫這個新的 system call，驗證它是否正常運行。

1. 先撰寫 `test_syscall.h` (附錄 C) 來宣告測試函數。
2. 再撰寫 `test_syscall.c` (附錄 D) 程式碼來呼叫這個 system call。
3. 最後用這個命令來編譯程式，`gcc -static -o test_syscall test_syscall.c`。

viii. 將測試程式加入到 QEMU

將測試程式加入到 QEMU 的根文件系統中，能夠在模擬環境中運行它。

```
~$ cp test_syscall busybox-1.30.1/_install/bin/
~$ cd busybox-1.30.1/_install
~$ find . -print0 | cpio --null -ov --format=newc | gzip -9 >
~/initramfs.cpio.gz
```

ix. 使用 QEMU 測試 System Call

使用 QEMU 模擬器來啟動核心並測試新增的 system call。

```
~$ cd linux-2.6.26
~$ qemu-system-i386 -kernel arch/x86/boot/bzImage -initrd ~/initramfs.cpio.gz -
append "console=ttyS0" -m 512 -nographic
```

```
/ # /bin/test_syscall
/ # dmesg | tail -n 10
```

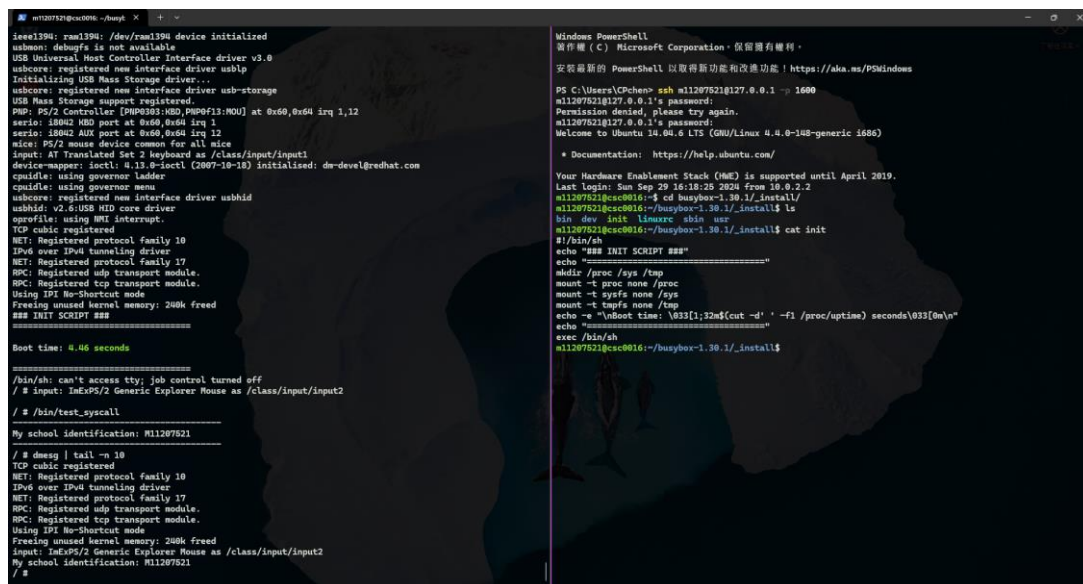


圖 3 執行 system call 成功

五、遇到的困難點

在這次新增系統呼叫的過程中發現一個比較棘手的問題，那就是當虛擬機的架構和 QEMU 開啟的架構不一致時，系統呼叫無法正常運行。雖然在不同的核心版本中所需的修改基本類似，但不同架構之間的差異還是非常明顯的。每個架構都有自己的指令集和核心配置需求，即便程式碼邏輯相同。經過這次作業的實作後，我認為在不同架構上執行的結果可能會完全不同。

六、HackMD 筆記

Build Linux Kernel 2.6.32: https://hackmd.io/@CHUN-PO-CHEN/B1_vQ9ZAA

七、 參考資料

1. Index of /pub/linux/kernel/v2.6/, <https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/>.
2. Online Kernel Code Viewer, <https://elixir.bootlin.com/linux/v2.6.26/source>.
3. Build a minimal Linux system and run it in QEMU, <https://ibug.io/blog/2019/04/os-lab-1/>.
4. HOWTO Add a system call to the 2.6 Linux Kernel, https://userpages.cs.umbc.edu/chettri/421/projects/hello_syscall.html.

附錄 A

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

asmlinkage long sys_stud_id_syscall(void)
{
    printk(KERN_EMERG "My school identification: M11207521\n");
    return 0;
}
```

附錄 B

```
#
# Makefile for the linux kernel.
#

obj-y      =      sched.o fork.o exec_domain.o panic.o printk.o profile.o \
                  exit.o itimer.o time.o softirq.o resource.o \
                  sysctl.o capability.o ptrace.o timer.o user.o \
                  signal.o sys.o kmod.o workqueue.o pid.o \
                  rcupdate.o extable.o params.o posix-timers.o \
                  kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
                  hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
                  notifier.o ksysfs.o pm_qos_params.o sched_clock.o \
                  stud_id_syscall.o # my syscall
```

附錄 C

```
#ifndef TEST_SYSCALL_H
#define TEST_SYSCALL_H

void call_stud_id_syscall(void);

#endif
```


附錄 D

```
#include "test_syscall.h"
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

void call_stud_id_syscall(void)
{
    printf("-----\n");
    syscall(327);
    printf("-----\n");
}

int main()
{
    call_stud_id_syscall();
    return 0;
}
```