

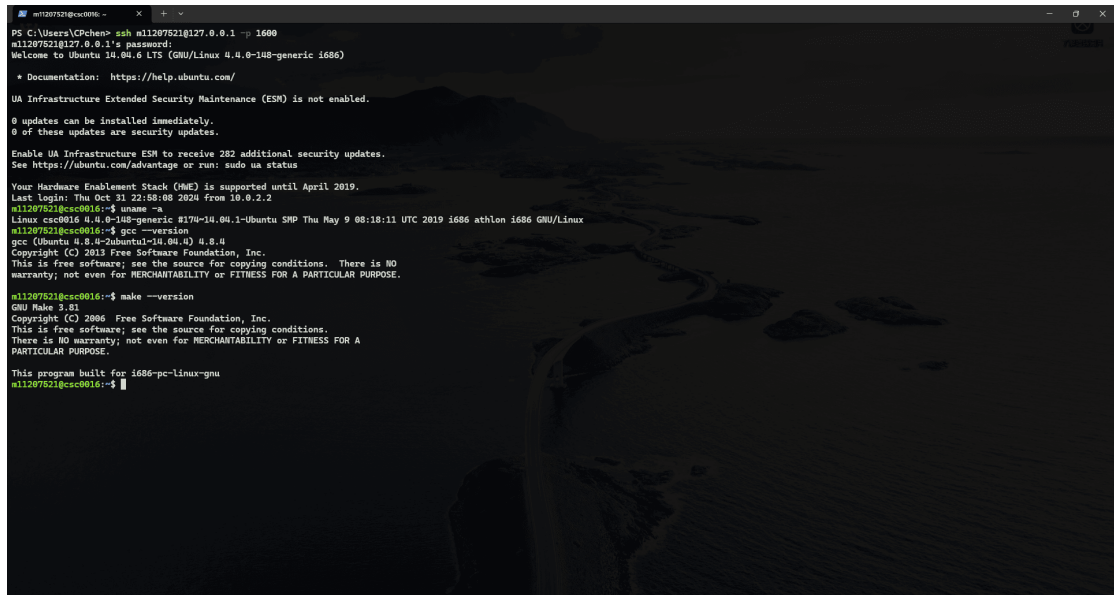
CSC0016 - Advanced Operating Systems

HW3_System Call for Process Information Retrieval

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。



```
PS C:\Users\CPchen> ssh m1207521@127.0.0.1 -p 1600
m1207521@127.0.0.1's password:
Welcome to Ubuntu 14.04.6 LTS (GNU/Linux 4.4.0-142-generic i686)

 * Documentation:  https://help.ubuntu.com/

UA Infrastructure Extended Security Maintenance (ESM) is not enabled.
0 updates can be installed immediately.
0 of these updates are security updates.

Enable UA Infrastructure ESM to receive 282 additional security updates.
See https://ubuntu.com/advantage or run: sudo ua status

Your Hardware Enablement Stack (HWE) is supported until April 2019.
Last login: Thu Oct 31 22:58:08 2024 from 10.0.2.2
m1207521@cs0016:~$ uname -a
Linux cs0016 4.4.0-142-generic #170-14.04.1-Ubuntu SMP Thu May 9 08:18:11 UTC 2019 i686 GNU/Linux
m1207521@cs0016:~$ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04~3) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

m1207521@cs0016:~$ make --version
GNU Make 3.81
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i686-pc-linux-gnu
m1207521@cs0016:~$
```

圖 1 VirtualBox 環境

二、先決條件

請使用先前作業中設定的開發環境進行此項操作。如果尚未完成開發環境的設置可以參考以下教學進行安裝和配置：[Homework 2: Add a System Call in Linux Kernel 2.6.26](#)。該教學詳細說明了如何安裝所需的工具和設定核心環境，以確保能順利進行後續的 system call 開發和測試。

三、定義 prinfo Structure

在 `include/linux/` 中創建 `prinfo.h` 檔案並定義 `prinfo` 結構(程式碼如附錄 A)。此結構允許從核心中提取並組織一個程序的各種資訊，以便將這些資訊傳遞到使用者層。以下是對 `prinfo` 結構中每個成員的說明和設計理由：

`prinfo` 結構設計用於在 system call 中傳遞程序的基本資訊。每個成員的作用如下：

- `state`：程序的當前狀態，如運行或睡眠中。
- `nice`：程序的優先級調整值，影響排程優先級。

- pid：程序 ID，用於唯一標識程序。
- parent_pid：父程序 ID，用於追溯程序來源。
- youngest_child_pid：最年輕的子程序 ID，若無子程序則為 0。
- start_time：程序的啟動時間（毫秒）。
- user_time 和 sys_time：程序在使用者模式和系統模式下消耗的 CPU 時間（毫秒）。
- uid：程序擁有者的使用者 ID。
- comm：程序名稱，最多 15 個字符。

這些成員提供了程序的基本資訊，有助於使用者層監控程序狀態。

四、 新增 System Call

本部分將介紹如何在 Linux 2.6.26 核心中新增一個 `sys_prinfo_syscall` system call，並通過 QEMU 進行測試。此過程包括修改核心程式碼、實現 system call、重新編譯核心以及撰寫使用者層級測試程式。

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_timerfd_settime      /* 325 */
.long sys_timerfd_gettime
.long sys_stud_id_syscall      /* 327 (HW2)*/
.long sys_prinfo_syscall      /* 328 (HW3)*/
```

ii. 定義 System Call Number

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

```
...
#define __NR_timerfd_settime    325
#define __NR_timerfd_gettime    326
#define __NR_stud_id_syscall    327    /* It's my student id call (HW2)*/
#define __NR_prinfo_syscall     328    /* It's prinfo call (HW3)*/
...
```

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：

```
#include <linux/prinfo.h> // <--- new
...
/*-----Assignment 2-----*/
asmlinkage long sys_stud_id_syscall(void);
/*-----*/

/*-----Assignment 3-----*/
asmlinkage long sys_prinfo_syscall(struct prinfo __user *info);
/*-----*/
asmlinkage long sys_time(time_t __user *tloc);
asmlinkage long sys_stime(time_t __user *tptr);
...
```

iv. System Call 實作

在 `linux-2.6.26/kernel/` 目錄下新增 `prinfo_syscall.c` 文件，並撰寫 system call 的程式碼（見附錄 B）。

v. 修改核心 Makefile

為了確保 system call 能正確地被核心編譯，需要編輯 `linux-2.6.26/kernel/Makefile` 文件，在 `obj-y` 後面加入 `prinfo_syscall.o`（如附錄 C）。

vi. 編譯核心

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_prinfo_syscall.c`（附錄 D）程式碼來呼叫這個 system call。
2. 再用這個命令來編譯撰寫完的程式：`gcc -static -o test_prinfo_syscall test_prinfo_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_prinfo_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 模擬器啟動核心，並測試新增的 `sys_prinfo_syscall` 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_prinfo_syscall
/ # dmesg | tail -n 10
```

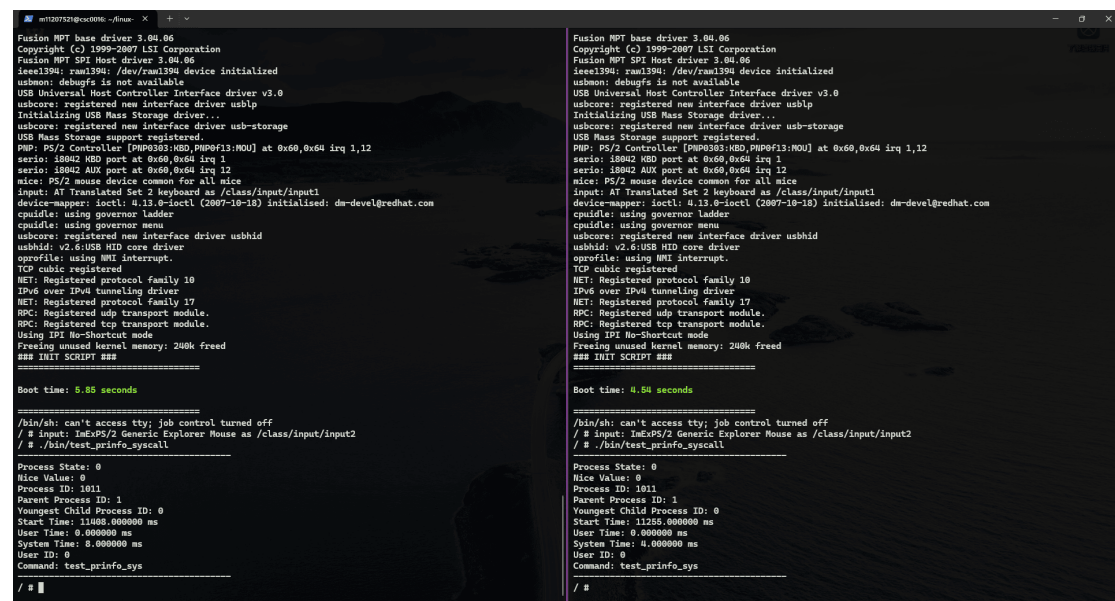


圖 2 執行 system call 成功

五、遇到的困難點

在這次新增 `sys_prinfo_syscall` 的過程中，`struct` 結構的定義和使用是一大挑戰。當我們在核心中定義 `prinfo` 結構時，必須確保每個欄位的型別和對齊方式與使用者層一致。這樣做的原因是，使用者層和核心層之間的數據傳遞需要完全對齊，否則會導致數據錯誤或解析失敗。例如，時間欄位如 `start_time`、`user_time` 和 `sys_time` 需要統一單位，以避免使用者層解析數據時出現誤差。這些細節使得結構定義需要精確且一致，這是此作業中的重要挑戰。

在使用 `struct` 語法時，`.` 操作符用於訪問結構中的成員。當我們有一個結構變數（例如 `pinfo`），可以透過 `pinfo.state` 來訪問 `prinfo` 結構中的 `state` 成員。否

則，如果我們有一個指向結構的指標（例如 `struct prinfo *info`），則需要使用 `->` 操作符（例如 `info->state`）來訪問成員。`.`與`->`操作符的選擇取決於變數是否為指標，這也是在核心程式碼中操作結構體時經常需要注意的基本語法。

六、 HackMD 筆記

[Homework 3: System Call for Process Information Retrieval](#)

七、 參考資料

1. Linux Kernel Documentation, <https://www.kernel.org/doc/Documentation/>.
2. Online Kernel Code Viewer, <https://elixir.bootlin.com/linux/v2.6.26/source>.
3. Adding a New System Call, <https://www.kernel.org/doc/html/next/process/adding-syscalls.html>.
4. C Structures (structs), https://www.w3schools.com/c/c_structs.php.
5. C struct, <https://www.programiz.com/c-programming/c-structures>.
6. 結構體 (Structure), <https://hackmd.io/@metal35x/H1VBKTnQL>.
7. Linux 内核 API task_nice, https://deepinout.com/linux-kernel-api/linux-kernel-api-process-scheduling/linux-kernel-api-task_nice.html.

附錄 A

```
// include/linux/prinfo.h
#ifndef _LINUX_PRINFO_H
#define _LINUX_PRINFO_H

struct prinfo {
    long state;
    long nice;
    pid_t pid;
    pid_t parent_pid;
    pid_t youngest_child_pid;
    unsigned long start_time;
    long user_time;
    long sys_time;
    long uid;
    char comm[16];
};

#endif /* _LINUX_PRINFO_H */
```


附錄 B

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/prinfo.h>
#include <linux/list.h>
#include <linux/jiffies.h>

asmlinkage long sys_prinfo_syscall(struct prinfo __user *info) {
    struct task_struct *task = current;
    struct prinfo pinfo;
    struct task_struct *child;
    pid_t youngest_pid = 0;

    pinfo.state = task->state;
    pinfo.nice = task->static_prio - 120;
    pinfo.pid = task->pid;
    pinfo.parent_pid = task->parent->pid;

    list_for_each_entry(child, &task->children, sibling) {
        youngest_pid = child->pid;
    }
    pinfo.youngest_child_pid = youngest_pid;

    pinfo.start_time = task->start_time.tv_sec * 1000 + task->start_time.tv_nsec / 1000000;

    pinfo.user_time = jiffies_to_msecs(task->utime);
    pinfo.sys_time = jiffies_to_msecs(task->stime);

    pinfo.uid = task->euid;

    strncpy(pinfo.comm, task->comm, sizeof(pinfo.comm) - 1);
    pinfo.comm[sizeof(pinfo.comm) - 1] = '\0';

    if (copy_to_user(info, &pinfo, sizeof(pinfo))) {
        return -EFAULT;
    }

    return 0;
}
```

附錄 C

```
#  
# 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 prinfo_syscall.o # my hw2 and hw3 syscall
```

附錄 D

```
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define SYS_PRINFO 328

struct prinfo {
    long state;
    long nice;
    pid_t pid;
    pid_t parent_pid;
    pid_t youngest_child_pid;
    unsigned long start_time;
    long user_time;
    long sys_time;
    long uid;
    char comm[16];
};

int main() {
    struct prinfo info;

    if (syscall(SYS_PRINFO, &info) == -1) {
        perror("syscall failed");
        return 1;
    }

    else {
        printf("-----\n");
        printf("Process State: %ld\n", info.state);
        printf("Nice Value: %ld\n", info.nice);
        printf("Process ID: %d\n", info.pid);
        printf("Parent Process ID: %d\n", info.parent_pid);
        printf("Youngest Child Process ID: %d\n", info.youngest_child_pid);
        printf("Start Time: %f ms\n", (double)info.start_time);
        printf("User Time: %f ms\n", (double)info.user_time);
        printf("System Time: %f ms\n", (double)info.sys_time);
        printf("User ID: %ld\n", info.uid);
        printf("Command: %s\n", info.comm);
        printf("-----\n");
    }

    return 0;
}
```