微算機系統實習
MICROPROCESSOR SYSTEMS LAB.
SPRING, 2021

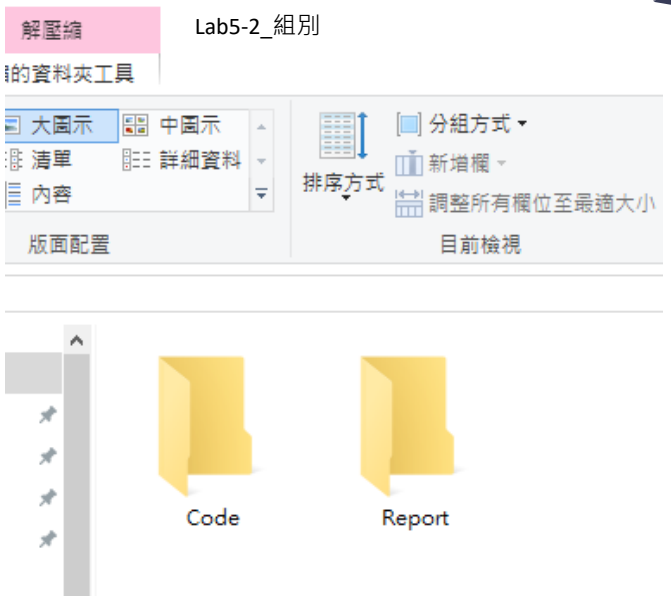
Instructor：Yen-Lin Chen(陳彥霖), Ph.D.
Professor
Dept. Computer Science and Information Engineering
National Taipei University of Technology

實驗五之二

實作 Char Device Driver

作業繳交格式

- 檔名: Lab5-2_組別.zip
- 其zip裡要包含如下資料夾
    1. -Code //存放專案程式碼
    2. -Report //存放報告



*需展示影片，請將影片上傳至Youtube，並附上影片連結，

請勿直接夾帶影片上傳至北科i學園

作業繳交

- 基本繳交時間

  -實驗：以拍攝影片進行驗收(請在報告附上Youtube影片連結)

  - 影片內容需清楚可見各實驗項目完成結果

  -報告：5/31 (23:59)以前上傳

  * 若有因為特殊原因繳交時間有變動助教會另外公布

  超過時間遲交每隔一週（含一週內）分數打8折，採累計連乘方式，實驗與報告打折是分開算的

  - 舉例：

    遲交三天 - 以遲交一週計算 <遲交的項目單獨分數>*0.8 = 該項目得到的分數

    遲交九天 - 以遲交兩週計算 <遲交的項目單獨分數>*0.8*0.8 = 該項目得到的分數

- 以上配分與注意事項，若有問題請聯絡助教

EMBEDDED LINUX DRIVER
進階資料

資料內容

- 此段落主要補充更深入的Linux kernel函式庫與驅動程式編寫上與一般C/C++不同之處與要注意的地方
  - 使用 Linux kernel 的函式庫進行檔案處理
  - 掛載多個驅動注意事項

使用 LINUX KERNEL 的函式庫進行檔案處理

- 檔案處理分為下列四項
  - 開檔
  - 讀檔
  - 寫檔
  - 關檔
- 使用kernel library做檔案處理必須include linux/fs.h、asm/uaccess.h

讀取 USER SPACE 的資料

- 我們是不能直接存取 buf 的內容。因為 Kernel Space 與 User Space 有不同的位址空間，所以不能直接存取他們。我們必須借助 copy_from_user 這個 API。
- 必須先include <asm/uaccess.h>

```
static ssize_t drv_write(struct file *filp, const char *buf, size_t count, loff_t *ppos){
        printk("Enter Write function\n");
        printk("%d\n", iCount);
        printk("W_buf_size: %d\n", (int)count);

        copy_from_user(userChar, buf, count);
        userChar[count - 1] = 0;

        printk("userChar(chr): %s\n", userChar);
        printk("userChar(int): %d\n", (int)sizeof(userChar));
```

userChar 資料型態為char array

使用 LINUX KERNEL 的函式庫進行檔案處理

- 開檔
  - 宣告檔案結構指標
    - struct file *fp;
  - 開啟檔案
    - fp = filp_open(”<檔案路徑>“, <開檔方式>, <權限>);
    - 檔案路徑：要開啟檔案的路徑
    - 開檔方式：有下列三種參數，若要同時使用２個（含）以上的方式可以使用OR(|)的方式來填寫，例如要使用讀寫檔案且當檔案沒有時候要創建就要輸入O_RDWR| O_CREAT
      - O_RDWR：Open for reading and writing. The result is undefined if this flag is applied to a FIFO.
      - O_WRONLY ：Open for writing only.
      - O_RDONLY：Open for reading only.
      - O_CREAT：若沒有檔案時創建檔案
    - 權限：Linux檔案權限（４為數字），預設設０即可

使用 LINUX KERNEL 的函式庫進行檔案處理

- 開檔前要先將記憶體資料存起來當檔案處理完關檔案後要寫回記憶體
- mm_segment_t old_fs; //建立紀錄記憶體中資料的結構
- old_fs = get_fs(); //取得記憶體中資料放
- set_fs(get_ds()); //將記憶體寫檔區段清空讓之後寫檔使用
- ～～～寫檔並關檔後～～～
- set_fs(old_fs); //回存記憶體中資料區段

使用 LINUX KERNEL 的函式庫進行檔案處理

- 關檔
- filp_close(<檔案結構指標>, NULL);

使用 LINUX KERNEL 的函式庫進行檔案處理

- 寫檔
- loff_t pos = 0; //從第幾個char開始寫
- vfs_write(<檔案結構指標>, <資料>, <資料大小>, &pos);

使用 LINUX KERNEL 的函式庫進行檔案處理

- 讀檔
- loff_t pos = 0; //從第幾個char開始讀
- vfs_read(<檔案結構指標>, <資料存放陣列>, <資料大小>, &pos);

使用 LINUX KERNEL 的函式庫進行檔案處理

```
struct file *fp; //建立檔案的結構
loff_t pos = 0; //從第幾個char開始寫
mm_segment_t old_fs; //記錄記憶體內的檔案資料
char buff[10] = { "1234567" };
old_fs = get_fs(); //記錄記憶體內的檔案資料
set_fs(get_ds());

fp = filp_open("/tmp/test.txt", O_WRONLY, 0);

printk("file is open!!\n");

vfs_write(fp, buff, strlen(buff), &pos);//寫檔
vfs_read(fp, buff, strlen(buff), &pos);//讀檔

filp_close(fp, NULL);//關檔
set_fs(old_fs); //之前資料寫回記憶體
```

實驗說明

本次實驗目標

- 練習撰寫一個可掛載於系統的字元驅動
- 用C/C++設計另一個程式來控制掛載中的字元驅動
- 能利用字元驅動來控制GPIO

LAB 5-2 項目要求

使用Linux Kernel的檔案存取方式來處理LED控制

- 請編寫一個字元裝置(character device)驅動程式
- 使用mknod指令在/dev/下建立一個名為demo的node (講義P.31)
- 編寫字元驅動需使用module_init及module_exit (講義P.14)
- 模組須使用Makefile編譯(講義P.12)
- 實驗流程：在/dev/下新增node →編寫字元裝置(建立file_operations資料結構)並編譯出ko檔(驅動模組檔案) →將ko檔(驅動模組檔案)掛載並查看(dmesg)系統訊息→執行LED控制程式(呼叫對應驅動模組) →控制特定LED燈或讀取特定LED點燈狀態

LAB 5-2 項目要求

- **項目一：創建file_operations資料結構 (15%)**
  - **每個function需印出以下規定訊息(講義P.39)**
    - Open: Enter Open function
    - Release: Enter Release function
    - Read: Enter Read function
    - Write: Enter Write function
    - I/O Control: Enter I/O Control function
- **項目二：掛載與卸載驅動程式(15%) (講義P.17)**
  - 初始(掛載)使用register_chrdev function (講義P.40)
  - 離開(卸載)使用unregister_chrdev function (講義P.41)

LAB 5-2 項目要求

- **項目三：控制裝置驅動 (20%)**
  - 必須能讀取命令參數，使指定的LED燈點亮或熄滅(on/off)

    能分別點亮及熄滅4顆LED燈，此動作需於驅動程式中完成(講義P.44)
  - Ex: ./Lab5 LED1 on (第一顆燈狀態為亮，其他LED狀態不變)
- **項目四：讀取裝置驅動狀態 (20%)**
  - 必須在終端機上顯示被點亮的燈號(講義P.43)
  - Ex: ./Lab5 LED1 (讀取LED1點燈狀態)

    顯示“**LED1 Status: 0**”(Terminal回傳LED1 狀態為0，表示未亮)
- **程式請於TX2嵌入式平台上執行**

影片拍攝內容

- 拍攝畫面須讓TX2、螢幕、LED同時入鏡，需看到以下操作:
  - 掛載後查看實驗規定系統訊息(dmesg)畫面
  - 編譯完成後讓4顆LED分別on跟off(項目三)
  - 查看4顆LED的state(項目四)
  - 卸載後再查看系統訊息(dmesg)畫面

    (項目一項目二) *需拍到項目一及項目二的訊息

參考範例

字元裝置驅動模組
drv_write function參考範例(1)

```
static ssize_t drv_write(struct file *flip, const char *buf, size_t count, loff_t *ppos)
{
    printk("device write\n");
    printk("%d\n", iCount);
    printk("W_buf_size: %d\n", (int)count);
    copy_from_user(userChar, buf, count); //利用這個API存取buf內容
    userChar[count - 1] = '\0';
    printk("userChar(char): %s\n", userChar);
    //printk("userChar(int): %d\n", (int)sizeof(userChar));
    char gpio[10] = {0};
    strncpy(gpio, userChar, 3);
    printk("gpio: %s, length: %d\n", gpio, strlen(gpio));
    struct file *io;
    loff_t pos = 0; //從第幾個char開始寫
    mm_segment_t old_fs; //建立紀錄記憶體中資料的結構
    old_fs = get_fs(); //取得記憶體中的資料
    set_fs(get_ds()); //將記憶體寫檔區段清空讓之後寫檔使用
```

因為 Kernel Space 與 User Space 有不同的位址空間，所以不能直接存取他們。必須借助 copy_from_user存取buffer

字元裝置驅動模組
drv_write function參考範例(2)

```
//export
io = filp_open("/sys/class/gpio/export", O_WRONLY, 0);
vfs_write(io, gpio, strlen(gpio), &pos); //寫檔
filp_close(io, NULL);

pos = 0;
//set-dir
if (strcmp(gpio, "396") == 0)
    io = filp_open("/sys/class/gpio/gpio396/direction", O_WRONLY, 0);

vfs_write(io, "out", 3, &pos);
filp_close(io, NULL);
pos = 0;
//set-value
if (strcmp(gpio, "396") == 0)
    io = filp_open("/sys/class/gpio/gpio396/value", O_WRONLY, 0);

vfs_write(io, &userChar[count - 2], 1, &pos); //存1or0(開關燈)
filp_close(io, NULL); //關檔

pos = 0;
set_fs(old_fs); //之前資料寫回記憶體
iCount++;
return count;
}
```

將GPIO腳位開啟

以gpio396為例，將腳位訊號設定為out

以gpio396為例，控制腳位開關

註：初始化要向系統註冊此模組對應的裝置檔案，參考講義P.40

LED控制程式
setGPIO function參考範例

(此程式用作控制/dev/demo裝置檔案，開啟LED燈)

這裡是user space，將user space輸入的資料轉成kenel space儲存的格式寫入

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>
#include <map>
#include <string>

using namespace std;
```

```
void setGPIO(unsigned int gpio, string status)
{
    int io;
    io = open("/dev/demo", O_WRONLY); //對/dev/demo 進行唯寫
    if (io < 0)
    {
        perror("gpio error"); //開檔失敗
        return;
    }
    char buf[10] = {0};
    if (status == "on") {
        strcpy(buf, (to_string(gpio) + "1").c_str()); //gpio跟status放入buf中
    }
    else {
        strcpy(buf, (to_string(gpio) + "0").c_str());
    }
    cout << buf << endl;
    write(io, buf, 5); //寫入
    close(io); //關檔
    return;
}
```

與註冊的裝置檔案作溝通

參考講義P.48