1 Walking through the Assignment 2

In this lab, we will go through the implementation of assignment 2.

1.1 Preparing the base code from Assignment 1

First, after fixing the problems I have mentioned in your assignment 1 code, make a copy of the assignment 1 code into assignment 2 (If you don't have a working copy of assignment 1, you may get a working copy from the teaching staff):

```
$ cp -r asgn1 asgn2
```

Then in the source code, rename all instances of "asgn1" to "asgn2".

Also in the assignment 2, mmap() and lseek() will not be needed. Therefore please remove the implementation of these functions. write() will also not needed in assignment 2, but you may want to use part of the code later in your bottom half. So at the moment, please comment out the body of write(). Don't forget to remove entries of .mmap, .llseek, and .write in:

```
struct file_operations asgn2_fops
```

This will serve as the base code for your assignment 2.

1.2 Dummy port I/O

1.2.1 Preparing your system

Read carefully the instructions of assignment 2 at http://www.cs.otago.ac.nz/cosc440/asgn2.php

Before connecting the dummy port device to your Raspberry Pi (RPI), you should shutdown your RPI and disconnect the power supply.

Carefully unclip and lift up the lid of your RPI. You will see the 26 GPIO pins on the board.

Then you should use jump wires to connect the 11 pins of the dummy port device to the corresponding pins on the board. Make sure you connect the pins with the same label using the same wire, as shown at http://www.cs.otago.ac.nz/cosc440/asgn2.php. There are two GROUND pins on the dummy port device and you can connect any of them to the GND pin on the board. Double check you have connected all pins correctly before turning on power. A wrong connection could DAMAGE the RPI!

Caution: when handling RPI, don't touch anything else except the metals lablled as GROUND in the image of the RPI board as shown at http://www.cs.otago.ac.nz/cosc440/asgn2.php.

In asgn2_init(), we need to initialize GPIO pins used for the dummy port device by calling **gpio_dummy_init**() as shown in the previous lab. All GPIO related code are in the following **gpio.c** file.

```
/**
 * File: gpio.c
 * Date: 04/08/2016
 * Author: Zhiyi Huang
 * Version: 0.2
 *
 * This is a gpio API for the dummy gpio device which
 * generates an interrupt for each half-byte (the most significant
 * bits are generated first.
 *
 * COSC440 assignment 2 in 2016.
 */

/* This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
```

```
 */

#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/version.h>
#include <linux/delay.h>
#if LINUX_VERSION_CODE > KERNEL_VERSION(3, 3, 0)
        #include <asm/switch_to.h>
#else
        #include <asm/system.h>
#endif
#include <mach/platform.h>

static u32 gpio_dummy_base;


/* Define GPIO pins for the dummy device */
static struct gpio gpio_dummy[] = {
                { 7, GPIOF_IN, "GPIO7" },
                { 8, GPIOF_OUT_INIT_HIGH, "GPIO8" },
                { 17, GPIOF_IN, "GPIO17" },
                { 18, GPIOF_OUT_INIT_HIGH, "GPIO18" },
                { 22, GPIOF_IN, "GPIO22" },
                { 23, GPIOF_OUT_INIT_HIGH, "GPIO23" },
                { 24, GPIOF_IN, "GPIO24" },
                { 25, GPIOF_OUT_INIT_HIGH, "GPIO25" },
                { 4, GPIOF_OUT_INIT_LOW, "GPIO4" },
                { 27, GPIOF_IN, "GPIO27" },
};

static int dummy_irq;
extern irqreturn_t dummyport_interrupt(int irq, void *dev_id);

static inline u32
gpio_inw(u32 addr)
{
    u32 data;

    asm volatile("ldr %0,[%1]" : "=r"(data) : "r"(addr));
    return data;
}

static inline void
gpio_outw(u32 addr, u32 data)
{
    asm volatile("str %1,[%0]" : : "r"(addr), "r"(data));
}

void setgpiofunc(u32 func, u32 alt)
{
        u32 sel, data, shift;

        if(func > 53) return;
        sel = 0;
        while (func > 10) {
            func = func - 10;
            sel++;
        }
        sel = (sel << 2) + gpio_dummy_base;
        data = gpio_inw(sel);
```

```
        shift = func + (func << 1);
        data &= ~(7 << shift);
        data |= alt << shift;
        gpio_outw(sel, data);
}

u8 read_half_byte(void)
{
u32 c;
u8 r;

r = 0;
c = gpio_inw(gpio_dummy_base + 0x34);
if (c & (1 << 7)) r |= 1;
if (c & (1 << 17)) r |= 2;
if (c & (1 << 22)) r |= 4;
if (c & (1 << 24)) r |= 8;

return r;
}

static void write_to_gpio(char c)
{
volatile unsigned *gpio_set, *gpio_clear;

gpio_set = (unsigned *)((char *)gpio_dummy_base + 0x1c);
gpio_clear = (unsigned *)((char *)gpio_dummy_base + 0x28);

if(c & 1) *gpio_set = 1 << 8;
else *gpio_clear = 1 << 8;
udelay(1);

if(c & 2) *gpio_set = 1 << 18;
else *gpio_clear = 1 << 18;
udelay(1);

if(c & 4) *gpio_set = 1 << 23;
else *gpio_clear = 1 << 23;
udelay(1);

if(c & 8) *gpio_set = 1 << 25;
else *gpio_clear = 1 << 25;
udelay(1);

}

int gpio_dummy_init(void)
{
    int ret;

    gpio_dummy_base = (u32)ioremap(BCM2708_PERI_BASE + 0x200000, 4096);
    printk(KERN_WARNING "The gpio base is mapped to %x\n", gpio_dummy_base);
    ret = gpio_request_array(gpio_dummy, ARRAY_SIZE(gpio_dummy));

    if (ret) {
        printk(KERN_ERR "Unable to request GPIOs for the dummy device: %d\n", ret);
        goto fail2;
        }
    ret = gpio_to_irq(gpio_dummy[ARRAY_SIZE(gpio_dummy)-1].gpio);
    if(ret < 0) {
        printk(KERN_ERR "Unable to request IRQ for gpio %d: %d\n", gpio_dummy[ARRAY_SIZE( ↩
            gpio_dummy)-1].gpio, ret);
```

```
        goto fail1;
    }
    dummy_irq = ret;
    printk(KERN_WARNING "Successfully requested IRQ# %d for %s\n", dummy_irq, gpio_dummy[ ↩
        ARRAY_SIZE(gpio_dummy)-1].label);

    ret = request_irq(dummy_irq, dummyport_interrupt, IRQF_TRIGGER_RISING | IRQF_ONESHOT, " ↩
        gpio27", NULL);

    if(ret) {
        printk(KERN_ERR "Unable to request IRQ for dummy device: %d\n", ret);
        goto fail1;
    }
    write_to_gpio(15);
return 0;

fail1:
    gpio_free_array(gpio_dummy, ARRAY_SIZE(gpio_dummy));
fail2:
    iounmap((void *)gpio_dummy_base);
    return ret;
}

void gpio_dummy_exit(void)
{
    free_irq(dummy_irq, NULL);
    gpio_free_array(gpio_dummy, ARRAY_SIZE(gpio_dummy));
    iounmap((void *)gpio_dummy_base);
}
```

When the device unloads, the module exit() must call:

```
void gpio_dummy_exit(void);
```

to release the GPIO-related resources acquired by the dummy port device.

1.2.2 Interrupt handler

When there is a half-byte appearing on the dummy port, an interrupt is triggered and we need an interrupt handler (top half) to very quickly copy that half-byte from the dummy port to a temporary byte. Two half-bytes will be assembled into one byte which will be put into a circular buffer.

In **gpio_dummy_init()**, we install the interrupt handler for the dummy port device by calling:

```
int request_irq (unsigned int   irq,
                 irq_handler_t  handler,
                 unsigned long  irqflags,
                 const char *   devname,
                 void *         dev_id);
```

where:

- **irq** is the interrupt line for this handler, which is 7 for our parallel port
- **handler** your IRQ handler function (top half)
- **irqflags** interrupt type flags, in our case, 0
- **devname** the module name
- **dev_id** an identifier, we can pass in **asgn2_deice**

It returns a non-zero value upon failure, which we must check for.

You need to make sure your IRQ handler function is used in the above **gpio_dummy_init()** function.

At cleanup, the IRQ must be released by:

```
void free_irq ( unsigned int irq, void *dev_id);
```

1.3 Overview of the Assignment 2

In this assignment, the supplied binary program **data_generator**

http://www.cs.otago.ac.nz/cosc440/data_generator

will send ascii text files to the dummy port, half-byte at a time, and the end of file is signalled by a *\0* character.

```
$ sudo ./data_generator <ascii file>
```

Your module will be read-only, and will only allow one reader at a time (i.e. each reader will receive a complete ascii file). If there are multiple readers trying to open your device, then your device will only allow one reader in, and queue up the rest of the readers.

1.3.1 Top half

Each half-byte appearing at the dummy port, will trigger an interrupt, which will trigger the interrupt handler (the top half) of the module. interrupt handler needs to quickly copy the half-byte off the dummy port. The tricky thing here is that you need to assemble two half-bytes into one byte. So you need to a toggle variable to know if the half-byte contains the most significant bits or least significant bits of a byte. The most significant bits of a byte are sent first, followed by the least significant bits. Once a byte is assembled, it should be added to the circular buffer - the temporary storage. Then the interrupt handler will call its bottom half, (a tasklet or a workqueue) to get the content of the circular buffer to the multiple page queue.

Since the interrupt handler cannot sleep, there is not enough time to reallocate the circular buffer, therefore when the circular buffer is full, and there are bytes coming in, the only option is to drop the bytes. It is your design decision to decide whether to drop the newest incoming bytes, or the oldest bytes stored in the circular buffer. Also, depending on your choice of the bottom half (tasklet or workqueue), you need to consider whether there will be data race between the top half (interrupt) and the bottom half, and if there may be a data race, you need to find ways to prevent data race, such as spinlocks. These design issues need to be clearly discussed in your report.

1.3.2 Bottom half (producer) and read() (consumer) in the multiple-page queue

The bottom half needs to get all bytes from the circular buffer into the multiple page queue. Here, the multiple page queue has a head and a tail index. Each index has an entry of the page, and the in-page offset. The bottom half is the producer, so it increments the tail; and read() is the consumer, so it increments the head.

After moving the bytes from the circular buffer to the multiple page queue, the bottom half needs to update the tail.

Here, read() is the consumer reading data from the head of the multiple-page queue to the user space. As it reads data, it advances the head, and when a page is finished, it moves to the next page. You need to make a design decision whether to free the used page immediately, or recycle the used page, and explain in your report.

read() needs to stop reading when an EOF of the current file is reached. Also if the multi-page queue is empty but EOF is not yet encountered, read() needs to be blocked, until new data appears in the multi-page queue. This is best handled by a wait queue between the bottom half and read(), and for this reason, after the bottom half adds new data to the multiple page queue, it needs to wake up the consumer queue.