1 Hardware interrupts, tasklets and workqueues

Interrupt handlers usually have two parts: the top halves and the bottom halves.

The top half does what needs to be done immediately, for example, a network driver top half acknowledges the interrupt and gets data off the network card into a buffer for later processing. Basically the top half itself is the interrupt handler.

The bottom half does the rest of the processing that has been deferred, which can be time consuming, or have delays that would otherwise hampers the response of the top half if put in the top half.

1.1 Top half

The top half is the interrupt handler, and it:

- Checks to make sure the interrupt was generated by the right hardware. This check is necessary for interrupt sharing.
- Clears an interrupt pending bit on the interface board.
- Does what needs to be done immediately (usually read or write something to/from the device). This data is usually written to or read from a device-specific bufer, which has been previously allocated.
- Schedules handling the new information later (in the bottom half) if the handling required is not trivial.

An example of an interrupt handler (top half) that schedules the bottom half with tasklets:

```
static struct my_dat { ... } my_fun_data;

/* tasklet bottom half */
static void t_fun (unsigned long t_arg{ ... }

DECLARE_TASKLET_OLD (t_name, t_fun);

/* interrupt handler */
static irqreturn_t my_interrupt (int irq, void *dev_id) {
    top_half_fun();
    tasklet_schedule(&t_name);
    return IRQ_HANDLED;
}
```

An interrupt handler needs to be registered during initialization, and deregistered during cleanup.

To register an interrupt handler:

```
#include <linux/interrupt.h>

int request_irq(unsigned int irq, irq_handler_t handler,
                unsigned long flags, const char *name, void *dev);
```

and to remove an interrupt handler:

```
#include <linux/interrupt.h>

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

Here is an example showing how to set up an interrupt handler:

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <linux/workqueue.h>
#include <linux/kthread.h>
#include <linux/slab.h>

/* IRQ of your network card to be shared */
#define SHARED_IRQ 19
static int irq = SHARED_IRQ;
module_param(irq, int, S_IRUGO);

/* default delay time in top half -- try 10 to get results */
static int delay = 0;
module_param(delay, int, S_IRUGO);

static atomic_t counter_bh, counter_th;

struct my_dat {
    unsigned long jiffies;    /* used for timestamp */
    struct tasklet_struct tsk;    /* used in dynamic tasklet solution */
    struct work_struct work;    /* used in dynamic workqueue solution */
};
static struct my_dat my_data;

static irqreturn_t my_interrupt(int irq, void *dev_id);

static int __init my_generic_init(void)
{
    atomic_set(&counter_bh, 0);
    atomic_set(&counter_th, 0);

    /* use my_data for dev_id */

    if (request_irq(irq, my_interrupt, IRQF_SHARED, "my_int", &my_data))
        return -1;

    printk(KERN_INFO "successfully loaded\n");
    return 0;
}

static void __exit my_generic_exit(void)
{
    synchronize_irq(irq);
    free_irq(irq, &my_data);
    printk(KERN_INFO " counter_th = %d,  counter_bh = %d\n",
           atomic_read(&counter_th), atomic_read(&counter_bh));
    printk(KERN_INFO "successfully unloaded\n");
}
```

1.2 Bottom half

A bottom half is used to process data, letting the top half to deal with new incoming interrupts. Interrupts are enabled when a bottom half runs. Interrupt can be disabled if necessary, but generally this should be avoided as this goes against the basic purpose of having a bottom half - processing data while listening to new interrupts.

There are three main types of bottom halves: namely tasklets, workqueues and kernel threads.

1.2.1 Tasklets

Tasklets are used to queue up work to be done at a later time. Tasklets can be run in parallel, but the same tasklet cannot be run on multiple CPUs at the same time. Also each tasklet will run only on the CPU that schedules it, to optimize cache usage. Since the thread that queued up the tasklet must complete before it can run the tasklet, race conditions are naturally avoided. However, this arrangement can be suboptimal, as other potentially idle CPUs cannot be used to run the tasklet. Therefore workqueues can, and should be used instead, and workqueues will be discussed in the next section.

The tasklet code is explained in **/usr/src/linux/include/linux/interrupt.h**, and the important data structure is:

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

func is a pointer to the function that will be run, with data as its parameter. state is used to determine whether the tasklet has already been scheduled, and if so, then it cannot be done so a second time.

The API of tasklets include:

```
DECLARE_TASKLET_OLD(name, function);
DECLARE_TASKLET_DISABLED(name, function, data);

void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long), unsigned long data);

void tasklet_schedule(struct tasklet_struct *t);
void tasklet_enable  (struct tasklet_struct *t);
void tasklet_disable (struct tasklet_struct *t);
void tasklet_kill    (struct tasklet_struct *t);
```

A tasklet must be initialized before being used, either by dynamically allocating space for the structure and call tasklet_init(), or statically declare and initialize by DECLARE_TASKLET_OLD() or DECLARE_TASKLET(). Alternative, the tasklet can be declared and set at disabled state by DECLARE_TASKLET_DISABLED(), which means that tasklet can be scheduled, but will not be run until the tasklet is specifically enabled.

tasklet_kill() is used to kill tasklets which reschedule themselves.

tasklet_schedule() is called to schedule a tasklet. Please note that if a tasklet has previously been scheduled (but not yet run), the new schedule will be silently discarded.

Here is a trivial example with my_init() scheduling a tasklet:

```
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/init.h>

typedef struct simp_t  {
    int i;
    int j;
} simp;

static simp t_data;
static void t_fun(unsignned long t_arg) {
    simp *datum = &t_data;

    printk(KERN_INFO "Entering t_fun, datum->i = %d, jiffies = %ld\n",
           datum->i, jiffies);
    printk(KERN_INFO "Entering t_fun, datum->j = %d, jiffies = %ld\n",
           datum->j, jiffies);
}

static int __init my_init(void) {
    printk(KERN_INFO "\nHello: my_init loaded at address 0x%p\n",
           my_init);
    t_data.i = 100;
    t_data.j = 200;
    printk(KERN_INFO "scheduling my tasklet, jiffies = %ld\n", jiffies);
    tasklet_schedule(&t_name);
    return 0;
}

static void __exit my_exit(void) {
    printk(KERN_INFO "\nHello: my_exit loaded at address 0x%p\n",
           my_exit);
}

module_init(my_init);
module_exit(my_exit);
```

1.2.2 Workqueues

A workqueue contains a linked list of tasks to be run at a deferred time. Tasks in workqueue:

- run in process context, therefore can sleep, and without inteferring with tasks running in any other queues.
- but still cannot transfer data to and from user space, as this is not a real user context to access.

The important data structure describing the tasks put into the queue is:

```
#include <linux/workqueue.h>

typedef void (*work_func_t)(struct work_struct *work);

struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_funct_t func;
};
```

func points to the function that will be run to get the work done. The other arguments are for internal use.

In order to pass data to a function, one needs to embed the work_struct in a use-defined data structure and then to pointer arithmetic in order to recover it. Here is an example:

```
static struct my_dat {
    int irq;
    struct work_struct work;
};
```

```
static void w_fun(struct work_struct *w_arg) {
    struct my_dat *data = container_of(w_arg, struct my_dat, work);
    atomic_inc(&bhs[data->irq]);
}
```

A work_struct can be declared and initialized at compiled time with:

```
DECLARE_WORK(name, void (*function)(void *));
```

where name is the name of the structure which points to queueing up function() to run. A previously declared work queue can be initialized and loaded with the the two macros:

```
INIT_WORK(   struct work_struct *work, void (*function)(void *));
PREPARE_WORK(struct work_struct *work, void (*function)(void *));
```

where work has already been declared as a work_struct. The INIT_WORK() macro initializes the **list_head** linked-list pointer, and PREPARE_WORK() sets the function pointer. The INIT_WORK() macro needs to be called at least once, and in turn calls PREPARE_WORK(). INIT_WORK() should not be called while a task is alraedy in the work queue.

Alternatively, a workqueue can be statically declared by:

```
DECLARE_WORK(work, void (*function)(void *));
```

In the kernel, there is a default workqueue named events. Tasks are added to amd flushed from this queue with the functions:

```
int schedule_work(struct work_struct *work);
void flush_scheduled_work(void);
```

flush_scheduled_work() is used when one needs to wait until all entries in a work queue have run.

1.2.3 Exercise: Deferred Functions

Write a driver that schedule a deferred function whenever a **write()** to the device takes place.

Pass some data to the driver and have it print out.

Have it print out the **current→pid** field when the tasklet is scheduled, and then again when the queued function is executed.

Implement this using:

- tasklets
- work queues

1.2.4 Exercise: Shared interrupts and bottom halves

Write a module that shares its IRQ with your USB device. You can find the interrupt used by USB by:

```
$ cat /proc/interrupts
```

The first column is the IRQ number and the fourth colummn lists the users of that IRQ number. On RPI, the USB uses IRQ 56. You can see more details about /proc/interrupts at:

http://www.centos.org/docs/5/html/5.1/Deployment_Guide/s2-proc-interrupts.html

Make it use a top half and a bottom half. Implement the bottom half using a tasklet, then repeat this exercise with a workqueue. Note when you share the IRQ 56 you have to make sure your handler returns IRQ_NONE to allow its original handler to work as well. Otherwise the USB devices won't work and the system will crash.

Since the USB interrupts are very frequent, make sure you limit the rate of **printk** in your tasklet using **printk_ratelimit()**. Otherwise, your log files will be soon inundated with your printed messages.

Check **/proc/interrupts** after the module is loaded.

1.3 Hardware I/O

In this part of the lab, we will look at reading from, and writing to hardware I/O ports. Though you will NOT need them for the second assignment, they are useful for hardware I/O in general. At the end of the lab, we will talk about the vitual port you will use in the second assignment.

Operations on I/O registers differs in important ways from normal memory access. In particular, there may be side effects caused by compiler and/or hardware optimizations that reorder instructions. In conventional memory reads and writes, there is no problem, as a write always store a value, and a read always return the last value written. However for I/O ports, there can be a problem because the CPU cannot tell when a process depends on the order of memory access. In another word, because of reading or writing an I/O register, device may initiate or respond to various actions.

Therefore a driver must ensure that no caching or reordering occurs. Otherwise problems which are difficult to diagnose, and only occur intemmitently, may occur.

The solution is to use appropriate memory barrier to prevent re-ordering of some instructions:

```
#include <asm-generic/system.h>

void barrier(void);

void rmb(void);
void wmb(void);
void mb (void);

void smp_rmb(void);
void smp_wmb(void);
void smp_mb (void);
```

barrier() causes the compiler to store in memory all values currently modified in a CPU register, to read them again later when they are needed. This function does not have effects on the hardware itself.

rmb() forces any reads before the barrier to complete before any reads after the barrrier are done; **wmb()** does the same thing for writes and mb() does it for both reads and writes.

Functions with **smp_** prefix insert barriers only on multi-processor systems, and on single CPU systems, just expands to **barrier()**.

Here is a code snippet showing the use of barriers:

```
io32write(direction, dev->base + OFF_DIR);
op32write(size, dev->base + OFF_SIZE);
wmb();
io32write(value, dev->base + OFF_GO);
```

In addition, many architectures provide convenience macros which combine setting a value with invoking a memory barrier. For example:

```
#define set_mb(var, value)  do {var = value; mb();  } while (0)
#define set_wmb(var, value) do {var = value; wmb(); } while (0)
#define set_rmb(var, value) do {var = value; rmb(); } while (0)
```

Memory barriers may cause performance hit, therefore they should be used with care. For example, on x86, the write memory barrier does nothing, as writes are never reordered. However reads may be reordered. So you should noy use **mb()** when **wmb()** will suffice.

1.3.1 REGISTERING I/O PORTS

Before accessing the I/O ports, the driver must register their use (usually during initialization), and the driver must unregister I/O ports during cleanup. These steps are done by:

```
#include <linux/ioport.h>

struct resource *request_region(unsigned long from, unsigned long extent, const char *name) ↩
    ;

void release_region(unsigned long from, unsigned long extent);
```

In these functions, the argument from is the base address of the I/O region. The argument extent is the number of ports (or addresses) and the argument name is the name that will appear in /port/ioports as the entry that claims the region.

Here is a code snippet showing how an I/O port is registered:

```
#include <linux/ioport.h>

static int my_dev_detect(unsigned long port_addr, unsigned long extent) {
    if (!request_region(port_addr, extent, "my_dev"))
        return -EBUSY;      /* the port is occupied */

    if (mydrv_probe(port_addr, extent) != 0) {
        release_region(port_addr, extent);
        return -ENODEV;    /* can't find the device */
    }
    return 0;
}
```

If a region of ports is properly registered, you can find them in **/proc/ioports**.

1.3.2 READING AND WRITING DATA FROM I/O REGISTERS

The following macros give the ability to read and write 8-bit (with suffix b), 16-bit (with suffix w) and 32-bit (with suffix l) once or multiple times:

Reading:

```
#include <linux/io.h>

unsigned char  inb  (unsigned long port_address);
unsigned short inw  (unsigned long port_address);
unsigned long  inl  (unsigned long port_address);
void           insb (unsigned long port_address, void *addr unsigned long count);
void           insw (unsigned long port_address, void *addr unsigned long count);
void           insl (unsigned long port_address, void *addr unsigned long count);
```

Writing:

```
#include <linux/io.h>

void outb  (unsigned char b,  unsigned long port_address);
void outw  (unsigned short w, unsigned long port_address);
void outl  (unsigned long l,  unsigned long port_address);
void outsb (unsigned long port_address, void *addr, unsigned long count);
void outsw (unsigned long port_address, void *addr, unsigned long count);
void outsl (unsigned long port_address, void *addr, unsigned long count);
```

In all architectures, long functions gives only 32-bit operations. Even in 64-bit architectures, there is no 64-bit data path.

The functions above that takes the count arguments do not write to a range of addresses. Instead, they write only to one port address, but they loop efficiently around the operation.

All these functions do I/O in little-endian order regardless of underlying architecture, and do any necessary byte-swapping.

Reading and writing I/O ports may require the use of memory barriers.

Here is an example:

```
/**
 * This function writes from the user buffer to the parallel port
 */
static ssize_t do_parport_write(struct file *filp, const char __user *buf,
                                size_t count, loff_t *f_pos) {
  size_t written = 0;

  while (written < count) {
    outb_p(0x00, parport_base);
    mb();
    outb_p(*(buf + written) | 0x80, parport_base);
    mb();
    udelay(5);

    written++;
    parport_device->total_written++;
  }

  *f_pos += written;
  return written;
}
```

1.3.3 SLOWING I/O CALLS TO THE HARDWARE

Some hardware can only read/write at a slower speed, therefore the kernel provides pausing functions that can be used to handle I/O to these slow devices. These functions have the same form as the I/O functions mentioned above, but with the suffix $_p$ attached to their names (e.g. $outb_p()$)

These functions insert a small delay after the I/O instruction if another such function follows. They should not be necessary except for very old ISA hardware.

1.4 Virtual port simulated with GPIO pins

We use 10 GPIO pins in our second assignment. They work in 5 pairs. Four pairs are used for data transmission of 4 bits (half-byte). Each pair has one GPIO pin as input and the other one as output, where the output pin is connected to the input pin. I will provide the following function for you to read the half-byte from the four input pins.

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

r = 0;
c = gpio_inw((u32)rpi_gpio->base + 0x34); // read the levels of all GPIO pins
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;
}
```

The fifth pair is used for interrupt. The input and output pins are connected. The vitual dummy device uses the output pin to send an interrupt signal to the input pin which triggers an IRQ to the CPU. The following functions are used to bring up and tear down the GPIO-based dummy device.

```
int gpio_dummy_init(void)
{
    int ret;

    gpio_dummy_base = (u32)ioremap_nocache(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);
}
```

The data is generated by a user-space program writing directly to the output pins of the four GPIO pairs for data transmission. The following **main()** function reads from a text file and writes to the virtual port (the output pins) half-byte by half-byte. The final byte of zero indicates the end of a file.

```
int main(int argc, char **argv) {
  FILE *in;
  size_t size_read;
  int i;
  char buf[SIZE];

  setup_io(); // set up base address for GPIO pins

  for (i = 1; i < argc; i++) {
    if (NULL == (in = fopen(argv[i], "r"))) {
      perror(argv[i]);
      continue;
    }

    while (!(feof(in) || ferror(in))) {
      size_read = fread(buf, 1, SIZE, in);
      write_to_port(buf, size_read);
    }

    /* insert '\0' to signal end of file */
    write_to_gpio(0);
    write_to_gpio(0);
    fclose(in);
  }

  return EXIT_SUCCESS;
}
```

The following function writes data to the virtual port half-byte by half-byte.

```
static int write_to_port(const char *buffer, size_t count) {
  size_t written = 0;

  while (written < count) {
        write_to_gpio(*(buffer + written)>>4);
        write_to_gpio(*(buffer + written) & 0xf);
    written++;
  }

  return written;
}
```

The following function writes a half-byte to the GPIO output pins. The bits of the half-byte can then be read by your device driver from the GPIO input pins which connect to the output pins. After the four bits are sent to the output pins, an interrupt signal is sent through pin 4, which brings the level of the pin to high and then brings it down to low.

```
volatile unsigned *gpio;//base address for GPIO pins

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

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

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

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

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

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

// send an interrupt signal to the CPU
*gpio_set = 1 << 4;
usleep(1);
*gpio_clear = 1 << 4;
usleep(1);

}
```

Then your interrupt handler, installed by **request_irq()** in **gpio_dummy_init()**, will be called and you should know how to handle the interrupt and the received half-byte in your device driver:-).

1.5 Reference

1. Writing Linux Device Drivers Chapters 20 and 21, by Jerry Cooperstein