Embedded Systems Programming

Work Queue and Input Processing in Linux
(Module 16)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014
Example of Work Structure and Handler

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/workqueue.h>
MODULE_LICENSE("GPL");

static struct workqueue_struct *my_wq; // work queue
typedef struct {
    struct work_struct my_work; // work
    int x;
} my_work_t;

my_work_t *work, *work2;

static void my_wq_function( struct work_struct *work) // function to be call
{
    my_work_t *my_work = (my_work_t *)work;
    printk("my_work.x %d\n", my_work->x);
    kfree( (void *)work );
    return;
}
```

Example of Work and WorkQueue Creation

```c
int init_module( void )
{
    int ret;
    my_wq = create_workqueue("my_queue");  // create work queue
    if (my_wq) {
        work = (my_work_t *) kmalloc(sizeof(my_work_t), GFP_KERNEL);
        if (work) {  // Queue work (item 1)
            INIT_WORK( (struct work_struct *)work, my_wq_function );
            work->x = 1;
            ret = queue_work( my_wq, (struct work_struct *)work );
        }
    }
    work2 = (my_work_t *) kmalloc(sizeof(my_work_t), GFP_KERNEL);
    if (work2) {  // Queue work (item 2)
        INIT_WORK( (struct work_struct *)work2, my_wq_function );
        work2->x = 2;
        ret = queue_work( my_wq, (struct work_struct *)work2 );
    }
}
return 0;  }
```

Linux Kernel Thread

- A way to implement background tasks inside the kernel

```c
static struct task_struct *tsk;
static int thread_function(void *data) {
    int time_count = 0;
    do {
        printk(KERN_INFO "thread_function: %d times", ++time_count);
        msleep(1000);
    } while(!kthread_should_stop() && time_count<=30);
    return time_count;
}

static int hello_init(void)  {
    tsk = kthread_run(thread_function, NULL, "mythread%d", 1);
    if (IS_ERR(tsk)) { .... }
}
```
Linux Input Systems

- An option: each attached input device is handled by a driver with the details of input port and protocol the device used.

- The other one -- Layers
  - adapter (controller) and port
  - device and driver
  - event interface

(Figure is from ELDD, Chapter 7)
Device drivers, input core, and event handlers

Example:
- i8042 is the driver for 8042 adapter
- psmouse is the driver for ps2 mouse
- mousedev is the event handler for all mice
The adapter – 8042

- from PC-AT, now a part of LPC IO
- PS2 signals: clock, data 5V, and GND.
  - CLOCK and DATA are of "open collector" type
- bidirectional serial protocol (start, data, parity, stop)
  - PC has always a priority and can stop the transmission any time by setting CLOCK low

Example: PS2 Mouse Driver

The driver for the adapter (controller)

When installed –

- create a platform_device
- initialize kbd and aux controllers
- create serio ports with ids (`serio->id.type = SERIO_8042;`)
- request_IRQ and add interrupt handler, and register ports

```c
error = request_irq(I8042_KBD_IRQ, i8042_interrupt, IRQF_SHARED, "i8042", i8042_platform_device);

if (likely(port->exists))
    serio_interrupt(port->serio, data, dfl);
```

Important fields in `struct serio`

```c
struct serio_device_id id;
struct serio_driver *drv;
struct device dev;
```
Threaded interrupt handlers

- isr acknowledges the interrupt to the hardware
- wake the kernel interrupt handler thread

\[ \text{int request_threaded_irq(} \text{unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev)} \]\n
- handler is called in hard interrupt context and checks if the interrupt was from the device
  - if thread_fn is NULL, use the normal handler, no irq thread
- handle_IRQ_event – calls handler (check or normal)
What is done in psmouse-base.c

- **The driver to handle ps2 mouse protocol**
- **When installed,**
  - probe serio bus, connect to serio device via the matching serio_id and create a “psmouse” device
- **psmouse registers itself as an input device to input core**
  - report events: EV_KEY and EV_REL
  - eventually, input_pass_event to handlers
- **When psmouse_interrupt is called**
  - received mouse data and process the protocol
  - `psmouse_process_byte()` analyzes the PS/2 data stream and reports relevant events to the input module once full packet has arrived.
- **What else –**
  - mouse type and protocol – command and response with adapter, mouse state, and data decoding.
Event Handlers

- **evdev:**
  - a generic input event interface to pass the events generated in the kernel straight to the program, with timestamps.
  - a char device to user space
    - when open, an evdev client is created with a buffer for events and is attached to file struct.
    - when read, fetch the events in the buffer and return to the user call.
  - register as a handler of an input device
    - when connected, evdev is created
    - handle is added to the input device
  - input device passes events to handler’s clients via
    - input core’s `input_pass_event`
    - handler’s `evdev_pass_event`

```c
struct input_event {
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};
```