Embedded Systems Programming

ISR and Work Queue
(Module 15)

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

Summer 2014
Interrupt Handling

- Depends on the type of interrupts
  - I/O interrupts
  - Timer interrupts
  - Interprocessor interrupts
- Unlike exceptions, interrupts are “out of context” events
- Generally associated with a specific device that delivers a signal on a specific IRQ
  - IRQs can be shared and several ISRs may be registered for a single IRQ
- ISRs is unable to sleep, or block
  - Critical: to be executed within the ISR immediately, with maskable interrupts disabled
  - Noncritical: should be finished quickly, so they are executed by the ISR immediately, with the interrupts enabled
  - Noncritical deferrable: deferrable actions are performed by means of separate functions
I/O Interrupt Handling

**HARDWARE**

- **Device 1**
- **Device 2**
- **IRQn**
- **PIC**

**SOFTWARE**

(Interrupt Handler)

- **INT**
- **IDT[32+n]**
- **IRQn_interrupt()**
- **do_IRQ(n)**
- **Interrupt service routine 1**
- **Interrupt service routine 2**

**common_interrupt:**
```
SAVE_ALL
movl %esp, %eax
call do_IRQ
jmp ret_from_intr
```

Execute ISRs associated with all the devices that share the IRQ.

*(D. P. Bovet and M. Cesati, “Understanding the Linux Kernel”, 3rd Edition)*
Why ISR Bottom Half?

- To have low interrupt latency -- to split interrupt routines into
  - a `top half`, which receives the hardware interrupt and
  - a `bottom half`, which does the lengthy processing.

- Top halves have following properties (requirements)
  - need to run as quickly as possible
  - run with some (or all) interrupt levels disabled
  - are often time-critical and they deal with HW
  - do not run in process context and cannot block

- Bottom halves are to defer work later
  - “Later” is often simply “not now”
  - Often, bottom halves run immediately after interrupt returns
  - They run with all interrupts enabled

- Code in the Linux kernel runs in one of three contexts:
  - Process context, kernel thread context, and Interrupt.
A World of Bottom Halves

- **Multiple mechanisms are available for bottom halves**

- **softirq: (available since 2.3)**
  - A set of 32 statically defined bottom halves that can run simultaneously on any processor
    - Even 2 of the same type can run concurrently
  - Used when performance is critical
  - Must be registered statically at compile-time

- **tasklet: (available since 2.3)**
  - Are built on top of softirqs
  - Two different tasklets can run simultaneously on different processors
    - But 2 of the same type cannot run simultaneously
  - Used most of the time for its ease and flexibility
  - Code can dynamically register tasklets

- **work queues: (available since 2.5)**
  - Queueing work to be performed in process context
WorkQueues

- To request that a function be called at some future time.
  - tasklets execute quickly, for a short period of time, and in atomic mode
  - workqueue functions may have higher latency but need not be atomic

- Run in the context of a special kernel process (worker thread)
  - more flexibility and workqueue functions can sleep.
  - they are allowed to block (unlike deferred routines)
  - No access to user space

- A workqueue (workqueue_struct) must be explicitly created

- Each workqueue has one or more dedicated “kernel threads”, which run functions submitted to the queue via queue_work().
  - work_struct structure to submit a task to a workqueue
    
    DECLARE_WORK(name, void (*function)(void *), void *data);

- The kernel offers a predefined work queue called events, which can be freely used by every kernel developer
# 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;
    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)) { .... }
}
```