Linux Interrupt Processing and Kernel Thread

Computer Science & Engineering Department
Arizona State University
Tempe, AZ 85287

Dr. Yann-Hang Lee
yhlee@asu.edu
(480) 727-7507
Preemptive Context Switching

Thread A
- Executing in kernel
  - Hardware Interrupt
  - Process Interrupt Request & wake up Thread B
  - Scheduler selects highest priority thread that is ready to run. If not the current thread, the current thread is made ready and the new thread resumed.

Thread B
- Blocked by an event
  - Thread B ready
  - Executing

Nested Execution of Handlers

- Generally nesting of kernel code paths is allowed with certain restrictions
- Exceptions can nest only 2 levels
  - Original exception and possible Page Fault
  - Exception code can block
- Interrupts can nest arbitrarily deep, but the code can never block (nor should it ever take an exception)

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
- INT
- PIC

**SOFTWARE**

*(Interrupt Handler)*

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

**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
Softirqs

- Softirqs are reentrant functions that are serialized on a given CPU, but can run concurrently across CPUs,
  - reduce the amount of locking needed
- Statically allocated at compile-time
- In 2.6.7 kernel, only 6 prioritized softirqs are used
  - HI_SOFTIRQ, TIMER_SOFTIRQ,
  - NET_TX_SOFTIRQ, NET_RX_SOFTIRQ,
  - SCSI_SOFTIRQ, TASKLET_SOFTIRQ
- A softirq often raised from within interrupt handlers and never preempts another softirq
- Pending softirqs are checked for and executed (call \textit{do_softirq()} ) in the following places:
  - After processing a HW interrupt
  - By the ksoftirqd kernel thread
  - By code that explicitly checks and executes pending softirqs
Tasklets are typically functions used by device drivers for deferred processing of interrupts,

- can be statically or dynamically enqueued on either the TASKLET_SOFTIRQ or the HI_SOFTIRQ softirq

Tasklets can only run on one CPU at a time, and are not required to be reentrant (run only once)

- a tasklet does not begin executing before the handler has completed.
- locking between the tasklet and other interrupt handlers may still be required
- locking between multiple tasklets

A more formal mechanism of scheduling software interrupts

- Tasklet struct -- the macro DECLARE_TASKLET(name, func, data)
- tasklet_schedule(&tasklet_struct) schedules a tasklet for execution.
- invokes raise_softirq_irqoff() to activate the softirq
- run in software interrupt context with the result that all tasklet code must be atomic.
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.
  - work_struct structure to submit a task to a workqueue
    DECLARE_WORK(name, void (*function)(void *), void *data);
- A shared, default workqueue provided by the kernel.
Work Queue Activation

- The `queue_work()` routine prepares a `work_struct` descriptor (holding a function) for a work queue and then:
  - Checks whether the function to be inserted is already present in the work queue (`work->pending` field equal to 1); if so, terminates
  - Adds the `work_struct` descriptor to the work queue list, and sets `work->pending` to 1
  - If a worker thread is sleeping in the `more_work` wait queue of the local CPU's `cpu_workqueue_struct` descriptor, this routine wakes it up

- The kernel offers a predefined work queue called `events`, which can be freely used by every kernel developer
  - saves significant system resources when the function is seldom invoked
  - must be careful not to enqueue functions that could block for a long period
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;
}
```