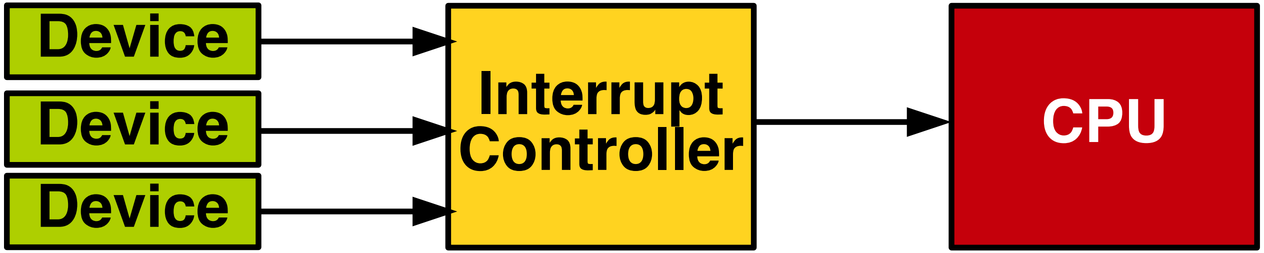
Interrupt Handler: Bottom Half

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
  - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler (top half)

Interrupt controller



- Interrupts are electrical signals multiplexed by the interrupt controller
  - Sent on a specific pin of the CPU
- Once an interrupt is received, a dedicated function is executed
  - **Interrupt handler**
- The kernel/user space can be interrupted at (nearly) any time to process an interrupt

Advanced PIC (APIC, I/O APIC)

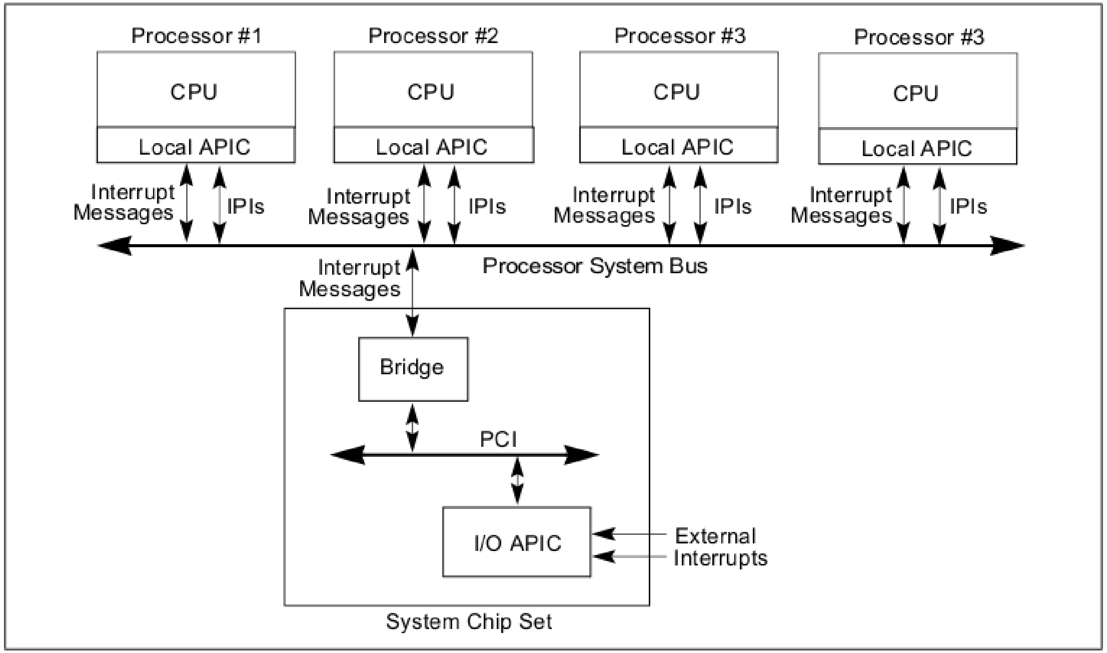


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

Interrupt descriptor table

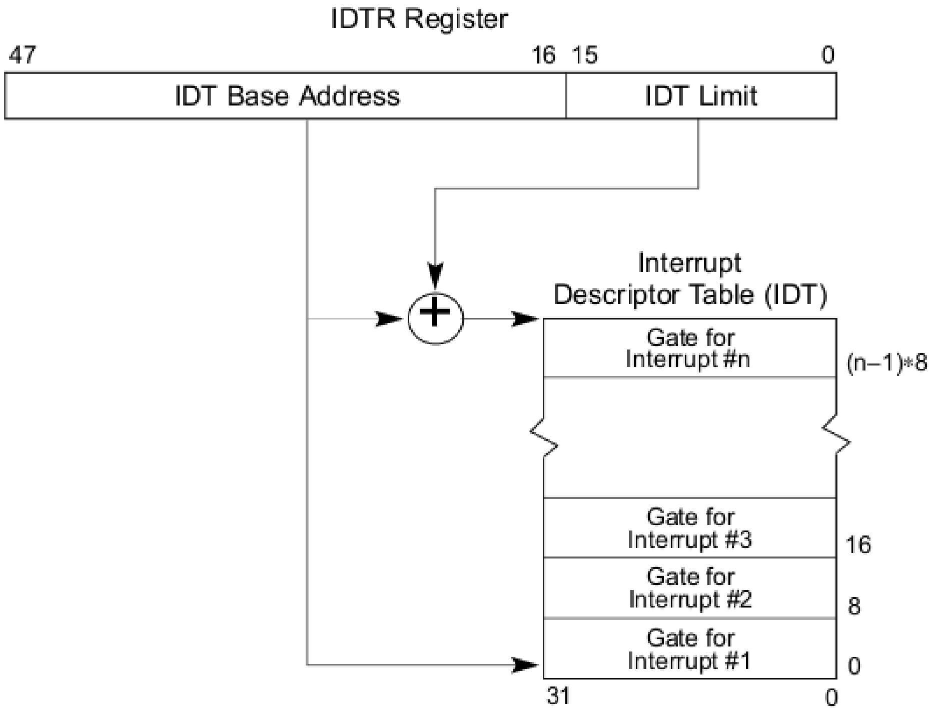


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor table

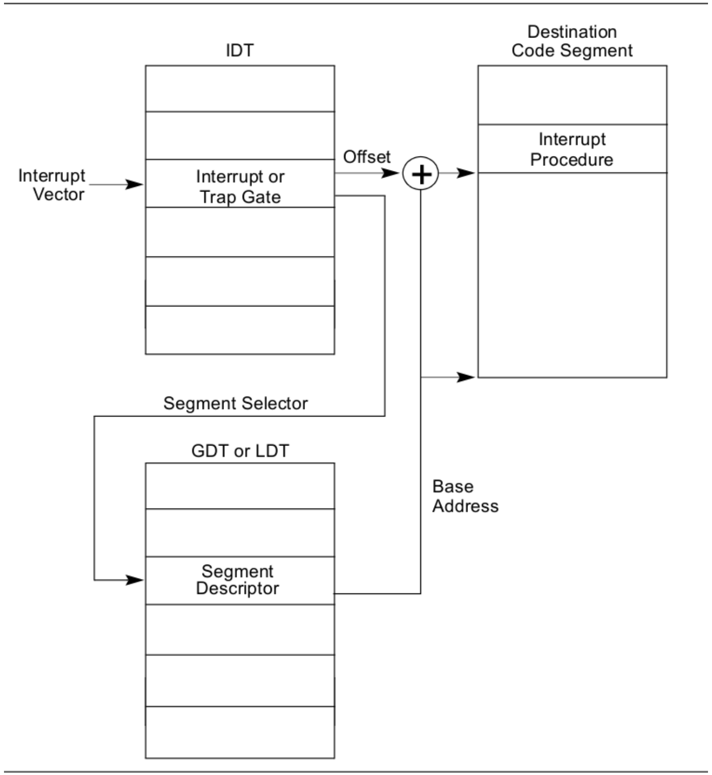
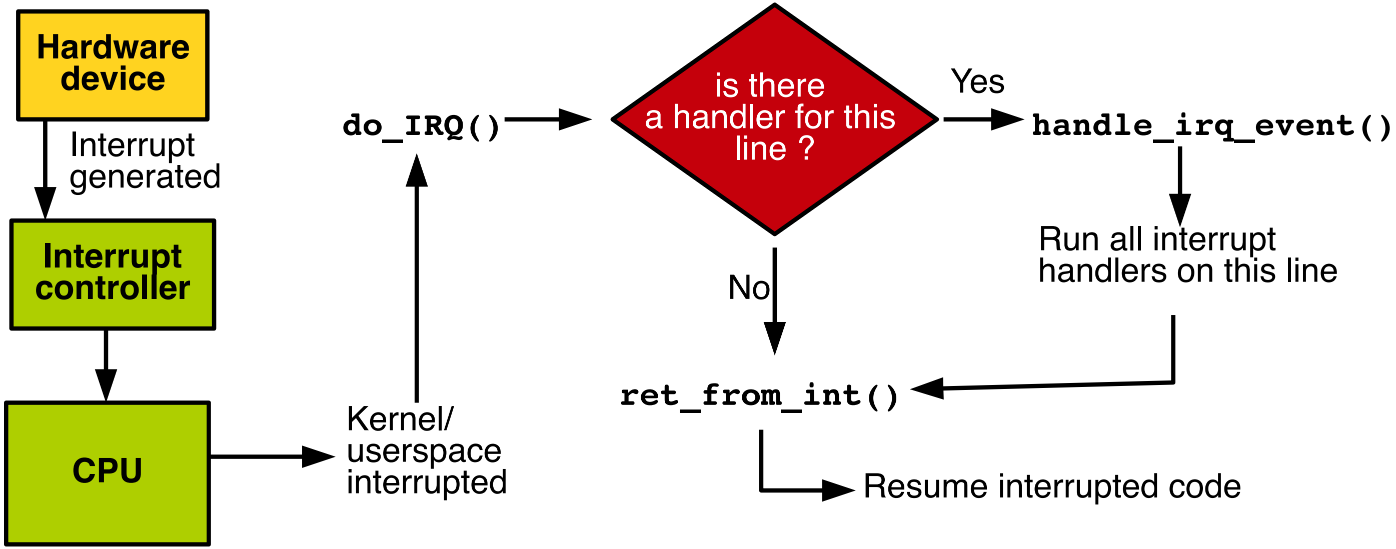


Figure 6-3. Interrupt Procedure Call

Interrupt handling internals in Linux



Today: interrupt handler

- **Top-halves (interrupt handlers)** must run as quickly as possible
  - They are interrupting other kernel/user code
  - They are often timing-critical because they deal with hardware
  - They run in interrupt context: they cannot block
  - One or all interrupt lines are disabled
- Defer the less critical part of interrupt processing to a **bottom-half**

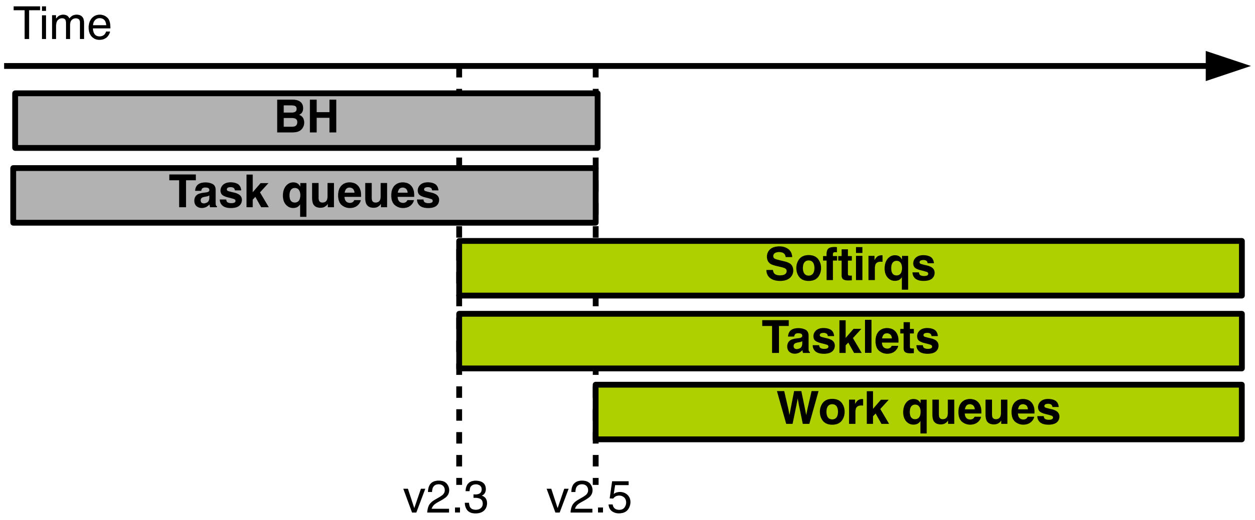
Top-halves vs. bottom-halves

- **When to use top halves?**
  - Work is time sensitive
  - Work is related to controlling the hardware
  - Work should not be interrupted by other interrupt
  - The top half is quick and simple, and runs with **some or all interrupts disabled**
- **When to use bottom halves?**
  - Everything else
  - The bottom half runs later with **all interrupts enabled**

The history of bottom halves in Linux

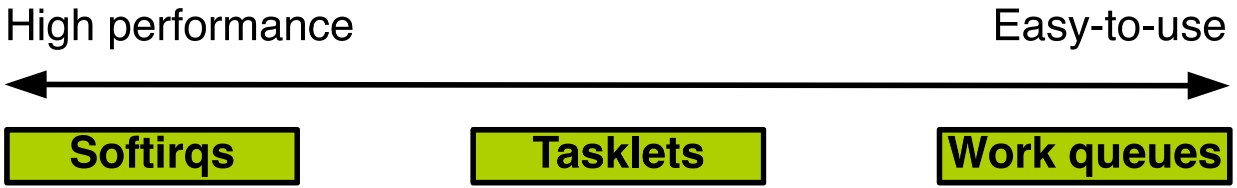
- “Top-half” and “bottom-half” are generic terms not specific to Linux
- Old ”Bottom-Half” (BH) mechanism
  - a statistically created list of 32 bottom halves
  - globally synchronized
  - easy-to-use yet inflexible and a performance bottleneck
- Task queues: queues of function pointers
  - still too inflexible
  - not lightweight enough for performance-critical subsystems (e.g., networking)

The history of bottom halves in Linux



- BH → Softirq, tasklet
- Task queue → work queue

Today's bottom halves in Linux



- All bottom-half mechanisms run with all interrupts enabled
- **Softirqs and tasklets** run in interrupt context
  - Softirq is rarely used directly
  - Tasklet is a simple and easy-to-use softirq (built on softirq)
- **Work queues** run in process context
  - They can block and go to sleep

Softirq

```
/* linux/kernel/softirq.c */
/* Softirq is statically allocated at compile time */
static struct softirq_action softirq_vec[NR_SOFTIRQS]; /* softirq vector */

/* linux/include/linux/interrupt.h */
enum {
    HI_SOFTIRQ=0,     /* [highest priority] high-priority tasklet */
    TIMER_SOFTIRQ,    /* timer */
    NET_TX_SOFTIRQ,   /* send network packets */
    NET_RX_SOFTIRQ,   /* receive network packets */
    BLOCK_SOFTIRQ,    /* block devices */
    IRQ_POLL_SOFTIRQ, /* interrupt-poll handling for block device */
    TASKLET_SOFTIRQ,  /* normal priority tasklet */
    SCHED_SOFTIRQ,    /* scheduler */
    HRTIMER_SOFTIRQ,  /* unused */
    RCU_SOFTIRQ,      /* [lowest priority] RCU locking */

    NR_SOFTIRQS       /* the number of defined softirq (< 32) */
};

struct softirq_action {
    void    (*action)(struct softirq_action *);        /* softirq handler */
};
```

Executing softirqs

- **Raising the softirq**
  - Mark the execution of a particular softirq is needed
  - Usually, a top-half marks its softirq for execution before returning
- **Pending softirqs are checked and executed in following places:**
  - In the return from hardware interrupt code path
  - In the `ksoftirqd` kernel thread
  - In any code that explicitly checks for and executes pending softirqs

Executing softirqs: `do_softirq()`

- Goes over the softirq vector and executes the pending softirq handler

```
/* linux/kernel/softirq.c */
/* do_softirq() calls __do_softirq() */
void __do_softirq(void) /* much simplified version for explanation */
{
    u32 pending;
    pending = local_softirq_pending(); /* 32-bit flags for pending softirq */
    if (pending) {
        struct softirq_action *h;

        set_softirq_pending(0); /* reset the pending bitmask */
        h = softirq_vec;
        do {
            if (pending & 1)
                h->action(h); /* execute the handler of the pending softirq */
            h++;
            pending >>= 1;
        } while (pending);
    }
}
```

Using softirq: assigning an index

```
/* linux/include/linux/interrupt.h */
enum {
    HI_SOFTIRQ=0,     /* [highest priority] high-priority tasklet */
    TIMER_SOFTIRQ,    /* timer */
    NET_TX_SOFTIRQ,   /* send network packets */
    NET_RX_SOFTIRQ,   /* receive network packets */
    BLOCK_SOFTIRQ,    /* block devices */
    IRQ_POLL_SOFTIRQ, /* interrupt-poll handling for block device */
    TASKLET_SOFTIRQ,  /* normal priority tasklet */
    SCHED_SOFTIRQ,    /* scheduler */
    HRTIMER_SOFTIRQ,  /* unused */
    RCU_SOFTIRQ,      /* [lowest priority] RCU locking */

    YOUR_NEW_SOFTIRQ, /* TODO: add your new softirq index */

    NR_SOFTIRQS       /* the number of defined softirq (< 32) */
};
```

Using softirq: registering a handler

```
/* linux/kernel/softirq.c */
/* register a softirq handler for nr */
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}

/* linux/net/core/dev.c */
static int __init net_dev_init(void)
{
    /* ... */
    /* register softirq handler to send messages */
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);

    /* register softirq handler to receive messages */
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
    /* ... */
}

static void net_tx_action(struct softirq_action *h)
{
    /* ... */
}
```

Using softirq: softirq handler

- Run with interrupts enabled and cannot sleep
- The key advantage of softirq over tasklet is scalability
  - **If the same softirq is raised again while it is executing, another processor can run it simultaneously**
- **This means that any shared data needs proper locking**
  - To avoid locking, most softirq handlers resort to per-processor data (data unique to each processor and thus not requiring locking)

Using softirq: raising a softirq

- Softirqs are most often raised from within interrupt handlers (i.e., top halves)
  - The interrupt handler performs the basic hardware-related work, raises the softirq, and then exits

```
/* linux/include/linux/interrupt.h */
/* Disable interrupt and raise a softirq */
extern void raise_softirq(unsigned int nr);

/* Raise a softirq. Interrupt must already be off. */
extern void raise_softirq_irqoff(unsigned int nr);

/* linux/net/core/dev.c */
raise_softirq(NET_TX_SOFTIRQ);
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

Tasklet

- Built on top of softirqs
  - `HI_SOFTIRQ` : high priority tasklet
  - `TASKLET_SOFTIRQ` : normal priority tasklet
- Running in an interrupt context (i.e., cannot sleep)
  - Like softirq, all interrupts are enabled
- Restricted concurrency than softirq
  - The same tasklet cannot run concurrently

tasklet_struct

```
/* linux/linux/include/interrupt.h */
struct tasklet_struct
{
    struct tasklet_struct *next; /* next tasklet in the list */
    unsigned long state;         /* state of a tasklet
        * - TASKLET_STATE_SCHED: a tasklet is scheduled for exeuciton
        * - TASKLET_STATE_RUN: a tasklet is running */
    atomic_t count;              /* disable counter
        * != 0: a tasklet is disabled and cannot run
        * == 0: a tasklet is enabled */
    void (*func)(unsigned long); /* tasklet handler function */
    unsigned long data;          /* argument of the tasklet function */
};
```

Scheduling a tasklet

- Scheduled tasklets are stored in two per-processor linked list:
  - `tasklet_vec`, `tasklet_hi_vec`

```
/* linux/kernel/softirq.c*/
struct tasklet_head {
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};

/* regular tasklet */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
/* high-priority tasklet */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

Scheduling a tasklet

```
/* linux/include/linux/interrupt.h, linux/kernel/softirq.c */

/* Schedule a regular tasklet
 * For high-priority tasklet, use tasklet_hi_schedule() */
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
    unsigned long flags;
    local_irq_save(flags);     /* disable interrupt */
    /* Append this tasklet at the end of list */
    t->next = NULL;
    *__this_cpu_read(tasklet_vec.tail) = t;
    __this_cpu_write(tasklet_vec.tail, &(t->next));
    /* Raise a softirq */
    raise_softirq_irqoff(TASKLET_SOFTIRQ); /* tasklet is a softirq */
    local_irq_restore(flags);  /* enable interrupt */
}
```

Tasklet softirq handlers

```
/* linux/kernel/softirq.c*/
void __init softirq_init(void)
{
    /* ... */
    /* Tasklet softirq handlers are registered at initializing softirq */
    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}

static __latent_entropy void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    /* Clear the list for this processor by setting it equal to NULL */
    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();
```

Tasklet softirq handlers (cont'd)

```
    /* For all tasklets in the list */
    while (list) {
        struct tasklet_struct *t = list;
        list = list->next;
        /* If a tasklet is not processing and it is enabled */
        if (tasklet_trylock(t) && !atomic_read(&t->count)) {
                /* and it is not running */
                if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                    BUG();
                /* then execute the associate tasklet handler */
                t->func(t->data);
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }
        local_irq_disable();
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        __raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_enable();
    }
}
```

Using tasklet: declaring a tasklet

```
/* linux/include/linux/interrupt.h */

/* Static declaration of a tasklet with initially enabled */
#define DECLARE_TASKLET(tasklet_name, handler_func, handler_arg)   \
struct tasklet_struct tasklet_name = { NULL, 0,                    \
                                       ATOMIC_INIT(0) /* disable counter */, \
                                       handler_func, handler_arg }

/* Static declaration of a tasklet with initially disabled */
#define DECLARE_TASKLET_DISABLED(tasklet_name, handler_func, handler_arg) \
struct tasklet_struct tasklet_name = { NULL, 0,                    \
                                       ATOMIC_INIT(1) /* disable counter */,  \
                                       handler_func, handler_arg }

/* Dynamic initialization of a tasklet */
extern void tasklet_init(struct tasklet_struct *tasklet_name,
             void (*handler_func)(unsigned long), unsigned long handler_arg);
```

Using tasklet: tasklet handler

- Run with interrupts enabled and cannot sleep
  - If your tasklet shared data with an interrupt handler, you must task precautions (e.g., disable interrupt or obtain a lock)
- Two of the same tasklets never run concurrently
  - Because `tasklet_action()` checks `TASKLET_STATE_RUN`
- But two different tasklets can run at the same time on two different processors

Using tasklet: scheduling a tasklet

```
/* linux/include/linux/interrupt.h */

void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);

/* Disable a tasklet by increasing the disable counter */
void tasklet_disable(struct tasklet_struct *t)
{
    tasklet_disable_nosync(t);
    tasklet_unlock_wait(t); /* and wait until the tasklet finishes */
    smp_mb();
}

void tasklet_disable_nosync(struct tasklet_struct *t)
{
    atomic_inc(&t->count);
    smp_mb__after_atomic();
}

/* Enable a tasklet by descreasing the disable counter */
void tasklet_enable(struct tasklet_struct *t)
{
    smp_mb__before_atomic();
```

Processing overwhelming softirqs

- System can be flooded by softirqs (and tasklets)
    - Softirq might be raised at high rates (e.g., heavy network traffic)
    - While running, a softirq can raise itself so that it runs again
- How to handle such overwhelming softirqs
    - **Solution 1**: Keep processing softirqs as they come in
        - User-space application can starve
    - **Solution 2**: Process one softirq at a time
        - Should wait until the next interrupt occurrence
        - Sub-optimal on an idle system

`ksoftirqd`

- Per-processor kernel thread to aid processing of softirq
- If the number of softirqs grows excessive, the kernel wakes up `ksoftirqd` with normal priority (nice 0)
  - No starvation of user-space application
  - Running a softirq has the normal priority (nice 0)

```
22:33 $ ps ax -eo pid,nice,stat,cmd | grep ksoftirq
    7   0 S    [ksoftirqd/0]
   18   0 S    [ksoftirqd/1]
   26   0 S    [ksoftirqd/2]
   34   0 S    [ksoftirqd/3]
```

`ksoftirqd`

- Although the work is now being done by ksfotirqd (kernel threads) in process context, ksoftirqd sets up an environment identical to that found in (softirq) interrupt context.
- ksoftirqd executes the softirq handlers with local interrupts enabled (and bottom halves disabled locally).
- A softirq handler code (which runs as a bottom half) does not need to change for ksoftirqd to run it.

Work queues

- Work queues defer work into a kernel thread
    - Always runs in process context
    - Thus, work queues are schedulable and can therefore sleep
- By default, per-cpu kernel thread is created, `kworker/n`
    - You can create additional per-CPU worker thread, if needed
- Workqueues users can also create their own threads for better performance and lighten the load on default threads

Work queue implementation: data structure

```
/* linux/kernel/workqueue.c */
struct worker_pool {
    spinlock_t        lock;       /* the pool lock */
    int               cpu;        /* I: the associated cpu */
    int               node;       /* I: the associated node ID */
    int               id;         /* I: pool ID */
    unsigned int      flags;      /* X: flags */

    struct list_head  worklist;   /* L: list of pending works */
    int               nr_workers; /* L: total number of workers */
    /* ... */
};

/* linux/include/workqueue.h */
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};

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

Work queue implementation: work thread

- Worker threads execute the `worker_thread()` function
- Infinite loop doing the following:

  1. Check if there is some work to do in the current pool
  2. If so, execute all the `work_struct` objects pending in the pool `worklist` by calling `process_scheduled_works()`
     - Call the `work_struct` function pointer `func`
     - `work_struct` objects removed
  3. Go to sleep until a new work is inserted in the work queue

Using work queues: creating work

```
/* linux/include/workqueue.h */

/* Statically creating a work */
DECLARE_WORK(work, handler_func);

/* Dynamically creating a work at runtime */
INIT_WORK(work_ptr, handler_func);

/* Work handler prototype
 * - Runs in process context with interrupts are enabled
 * - How to pass a handler-specific parameter
 *   : embed work_struct and use container_of() macro */
typedef void (*work_func_t)(struct work_struct *work);

/* Create/destory a new work queue in addition to the default queue
 * - One worker thread per process */
struct workqueue_struct *create_workqueue(char *name);
void destroy_workqueue(struct workqueue_struct *wq);
```

Using work queues: scheduling work

```
/* Put work task in global workqueue (kworker/n) */
bool schedule_work(struct work_struct *work);
bool schedule_work_on(int cpu,
         struct work_struct *work); /* on the specified CPU */

/* Queue work on a specified workqueue */
bool queue_work(struct workqueue_struct *wq, struct work_struct *work);
bool queue_work_on(int cpu, struct workqueue_struct *wq,
         struct work_struct *work); /* on the specified CPU */
```

Using work queues: finishing work

```
/* Flush a specific work_struct */
int flush_work(struct work_struct *work);
/* Flush a specific workqueue: */
void flush_workqueue(struct workqueue_struct *);
/* Flush the default workqueue (kworkers): */
void flush_scheduled_work(void);

/* Cancel the work */
void flush_workqueue(struct workqueue_struct *wq);
/* Check if a work is pending  */
work_pending(struct work_struct *work);
```

Work queue example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/workqueue.h>

struct work_item {
    struct work_struct ws;
    int parameter;
};

struct work_item *wi, *wi2;
struct workqueue_struct *my_wq;

static void handler(struct work_struct *work)
{
    int parameter = ((struct work_item *)container_of(
                            work, struct work_item, ws))->parameter;
    printk("doing some work ...\n");
    printk("parameter is: %d\n", parameter);
}
```

Work queue example

```c
static int __init my_mod_init(void)
{
    printk("Entering module.\n");

    my_wq = create_workqueue("lkp_wq");
    wi = kmalloc(sizeof(struct work_item), GFP_KERNEL);
    wi2 = kmalloc(sizeof(struct work_item), GFP_KERNEL);

    INIT_WORK(&wi->ws, handler);
    wi->parameter = 42;
    INIT_WORK(&wi2->ws, handler);
    wi2->parameter = -42;

    schedule_work(&wi->ws);
    queue_work(my_wq, &wi2->ws);

    return 0;
}
```

Work queue example

```
static void __exit my_mod_exit(void)
{
    flush_scheduled_work();
    flush_workqueue(my_wq);
    kfree(wi);
    kfree(wi2);
    destroy_workqueue(my_wq);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
MODULE_LICENSE("GPL");
```

Choosing the right bottom-half

Bottom half	Context	Inherent serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queue	Process	None

- All of these generally run with interrupts enabled
- If there is a shared data with an interrupt handler (top-half), need to disable interrupts or use locks

Disabling softirq and tasklet processing

```
/* Disable softirq and tasklet processing on the local processor */
void local_bh_disable();

/* Eanble softirq and tasklet processing on the local processor */
void local_bh_enable();
```

- The calls can be nested
  - Only the final call to `local_bh_enable()` actually enables bottom halves
- These calls do not disable workqueues processing

Further readings

- 0xAX: Interrupts and Interrupt Handling
- Modernizing the tasklet API
- Moving interrupts to threads

Next lecture

- Kernel synchronization