CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 11

Interrupt Handler: Bottom Half

March 4, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

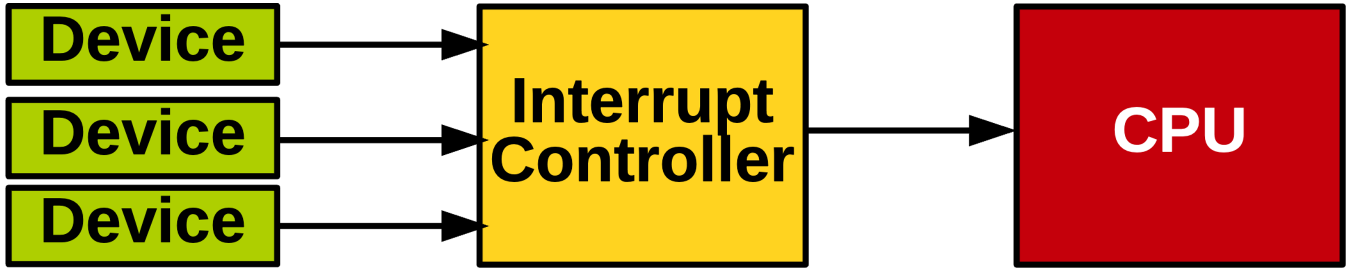
Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Interrupts: What? Why? and How?

- A mechanism to implement abstraction and multiplexing
- Interrupt: asking for a service from the kernel
  - via software (e.g., “int 0x80”) or by hardware (e.g., keyboard)
- Interrupt handling in Linux
  - how to track interrupts
  - how to handle them
    - » top half + bottom half

Interrupt Controller

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



Top-half vs. Bottom-half

- In Linux (and many other OSes), an interrupt process is split into two parts
- Top-half: run immediate upon receiving the interrupt
  - only handle time-critical operations, e.g., ack and reset interrupt
- Bottom-half: less critical & time-consuming work
  - Run later with other interrupts enabled
- An example: network packet processing
  - Top-half
    - » acknowledge the hardware, "hey, I received your signal"
    - » Copy packet to main memory
    - » Set the NIC to a status to receive more packets
    - » Critical: packet buffer on NIC is limited, might lead to packet drop if not processed timely
  - Bottom-half
    - » softirq, tasklet, workqueue
    - » Similar to thread pool in user-space

Registering an Interrupt Handler

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

/**
 *  This call allocates interrupt resources and enables the
 *  interrupt line and IRQ handling.
 *
 *  @irq: Interrupt line to allocate
 *  @handler: Function to be called when the IRQ occurs.
 *        Primary handler for threaded interrupts
 *  @irqflags: Interrupt type flags
 *        IRQF_SHARED - allow sharing the irq among several devices
 *        IRQF_TIMER - Flag to mark this interrupt as timer interrupt
 *        IRQF_TRIGGER_* - Specify active edge(s) or level
 *  @devname: An ascii name for the claiming device
 *  @dev_id: A cookie passed back to the handler function
 *        Normally the address of the device data structure
 *        is used as the cookie.
 */

int request_irq(unsigned int irq, irq_handler_t handler,
    unsigned long irqflags, const char *devname, void *dev_id);
```

Freeing an Interrupt Handler

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

/**
 *  Free an interrupt allocated with request_irq
 *
 *  @irq: Interrupt line to free
 *  @dev_id: Device identity to free
 *
 *  Remove an interrupt handler. The handler is removed and if the
 *  interrupt line is no longer in use by any driver it is disabled.
 *  On a shared IRQ the caller must ensure the interrupt is disabled
 *  on the card it drives before calling this function. The function
 *  does not return until any executing interrupts for this IRQ
 *  have completed.
 *
 *  Returns the devname argument passed to request_irq.
 */
const void *free_irq(unsigned int irq, void *dev_id);
```

Writing an Interrupt Handler

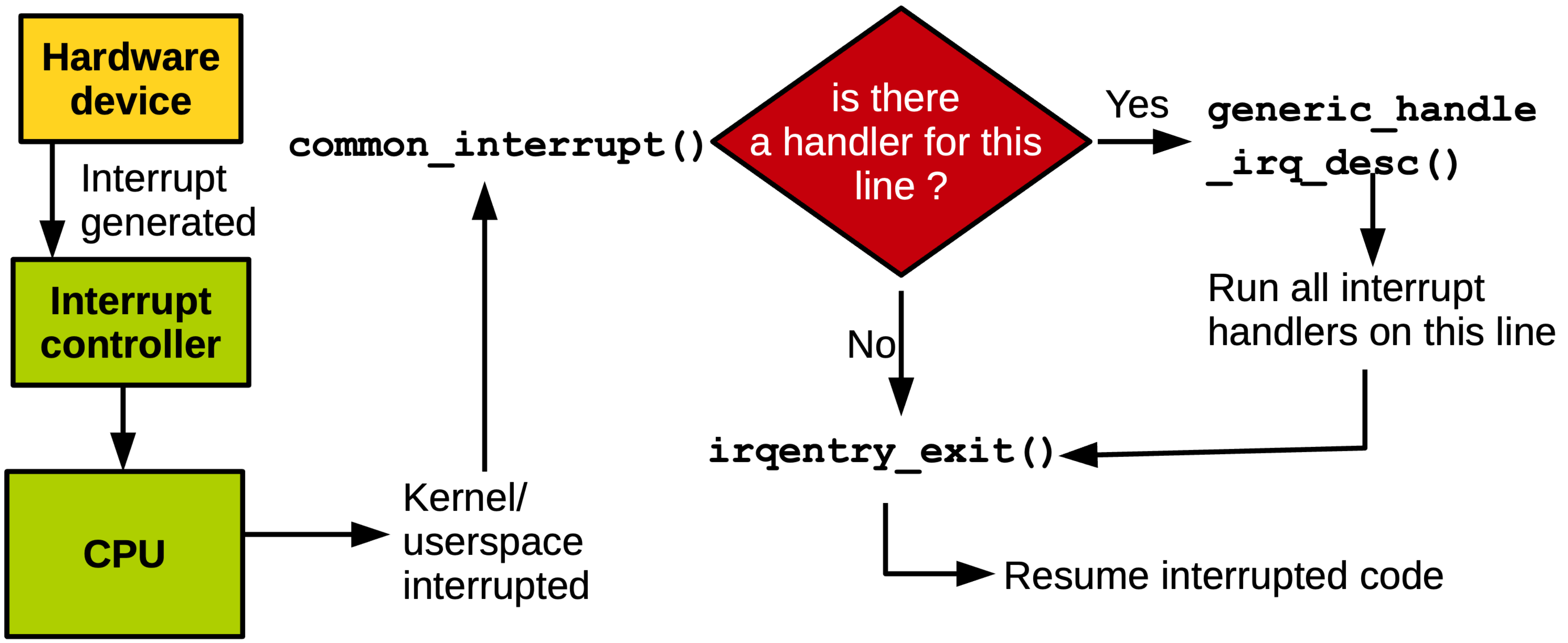
```
/* linux/include/linux/interrupt.h */

/**
 * Interrupt handler prototype
 *
 * @irq: the interrupt line number that the handler is serving
 * @dev_id: a generic pointer that was given to request_irq()
 *      when the interrupt handler is registered
 *
 * Return value:
 *      IRQ_NONE: the interrupt is not handled (i.e., the expected
 *        device was not the source of the interrupt)
 *      IRQ_HANDLED: the interrupt is handled (i.e., the hanlder was
 *        correctly invoked)
 *      #define IRQ_RETVAL(x)   ((x) ? IRQ_HANDLED : IRQ_NONE)
 *
 * NOTE: interrupt handlers need not be reentrant (tread-safe)
 *      - When a given interrupt handler is executing, the corresponding
 *      interrupt line is disabled on all cores while.
 *      - Normally all other interrupts are enables, os other interrupts
 *      are serviced.
 */
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
```

Interrupt Context

- Process context: normal task execution, syscall, and exception
- Interrupt context: ISR
  - Sleeping/blocking is not possible b/c ISR is not a schedulable entity
  - No kmalloc(size, GFP_KERNEL), use "GFP_ATOMIC" instead
  - No blocking locking (e.g., mutex), use "spinlock()" instead
  - No printk(), use trace_printk instead
- Small stack size, one page, e.g., 4KB

Interrupt Handling in Linux



- Specific entry point for each interrupt line
  - Saves the interrupt number and current registers
  - Calls common_interrupt()
- common_interrupt(struct pt_regs *reg, u32 vector)
  - Ack interrupt, disable the line
  - Calls architecture specific functions
- Call chain ends up by calling generic_handle_irq_desc()
  - cal the handler if the line is not shared
  - otherwise iterate over all the handlers registered on that line
  - disable interrupts on the line again if they were previously enabled
- common_interrupt() returns to entry point that call irqentry_exit()
  - checks if reschedul is needed (*need_resched*)
  - restore register values

IDT Initialization

```c
/* linux/arch/x86/include/asm/desc_defs.h */
struct gate_struct {
    u16     offset_low;
    u16     segment;
    struct idt_bits bits;
    u16     offset_middle;
#ifdef CONFIG_X86_64
    u32     offset_high;
    u32     reserved;
#endif
} __attribute__((packed));

typedef struct gate_struct gate_desc;

/* linux/arch/x86/kernel/traps.c */
DECLARE_BITMAP(system_vectors, NR_VECTORS);
```

```
/* linux/arch/x86/include/asm/idtentry.h
/*
 * Build the entry stubs with some assembler magic.
 * We pack 1 stub into every 8-byte block.
 */
    .align 8
SYM_CODE_START(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    UNWIND_HINT_IRET_REGS
0 :
    .byte   0x6a, vector
    jmp asm_common_interrupt
    nop
    /* Ensure that the above is 8 bytes max */
    . = 0b + 8
    vector = vector+1
    .endr
SYM_CODE_END(irq_entries_start)
```

```c
/* linux/init/main.c */

asmlinkage __visible void __init start_kernel(void)
{
    /* ... */
    early_irq_init();
    init_IRQ();
    /* ... */
}

/* linux/arch/x86/kernel/irqinit.c */
void __init init_IRQ(void)
{
    int i;
    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[ISA_IRQ_VECTOR(i)] = irq_to_desc(i);

    BUG_ON(irq_init_percpu_irqstack(smp_processor_id()));

    x86_init.irqs.intr_init();
}
```

```c
/* linux/arch/x86/kernel/idt.c */
void __init idt_setup_apic_and_irq_gates(void)
{
    int i = FIRST_EXTERNAL_VECTOR;
    void *entry;

    idt_setup_from_table(idt_table, apic_idts, ARRAY_SIZE(apic_idts), true);

    for_each_clear_bit_from(i, system_vectors, FIRST_SYSTEM_VECTOR) {
        entry = irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR);
        set_intr_gate(i, entry);
    }
}
```

```c
/* linux/arch/x86/kernel/irqinit.c */
void __init native_init_IRQ(void)
{
    /* Execute any quirks before the call gates are initialised: */
    x86_init.irqs.pre_vector_init();

    idt_setup_apic_and_irq_gates();
    lapic_assign_system_vectors();

    if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
        setup_irq(2, &irq2);
}
```

```c
/* linux/arch/x86/kernel/idt.c */
static void set_intr_gate(unsigned int n, const void *addr)
{
    struct idt_data data;

    BUG_ON(n > 0xFF);

    memset(&data, 0, sizeof(data));
    data.vector = n;
    data.addr   = addr;
    data.segment    = __KERNEL_CS;
    data.bits.type  = GATE_INTERRUPT;
    data.bits.p = 1;

    idt_setup_from_table(idt_table, &data, 1, false);
}
```

Interrupt Control

- Kernel code sometimes need to disable interrupts to ensure atomic execution
  - By disabling interrupts, it guarantees that an interrupt handle will not preempt your code
  - Disabling interrupts also disables kernel preemption
- Disabling interrupts does not protect against concurrent access from other cores
  - Need locking, often used in conjunction with interrupt disabling
- The kernel provides APIs to disable/enable interrupts
  - local_irq_disable()
  - local_irq_enable()
  - can be called multiple times

Disabling Interrupts on the Local Core

- Use local_irq_save()

```
unsigned long flags;
local_irq_save(flags);    /* disables interrupts if needed */
/* ... */
local_irq_restore(flags); /* restore interrupt status to the previous */

/* nesting is okay */
unsigned long flags;
local_irq_save(flags);
{
  unsigned long flags;
  local_irq_save(flags);
  /* ... */
  local_irq_restore(flags);
}
local_irq_restore(flags);
```

Disabling Specific Interrupts

```
/** disable_irq - disable an irq and wait for completion
 *  @irq: Interrupt to disable
 *
 *  Disable the selected interrupt line.  Enables and Disables are
 *  nested.
 *  This function waits for any pending IRQ handlers for this interrupt
 *  to complete before returning. If you use this function while
 *  holding a resource the IRQ handler may need you will deadlock.
 *
 *  This function may be called - with care - from IRQ context. *//
void disable_irq(unsigned int irq);

/** disable_irq_nosync - disable an irq without waiting
 *  @irq: Interrupt to disable */
void disable_irq_nosync(unsigned int irq);

/** enable_irq - enable handling of an irq
 *  @irq: Interrupt to enable
 *
 *  Undoes the effect of one call to disable_irq().  If this
 *  matches the last disable, processing of interrupts on this
 *  IRQ line is re-enabled. */
void enable_irq(unsigned int irq);
```

Interrupt Status

```
/* linux/include/linux/preempt.h */

/*
 * Are we doing bottom half or hardware interrupt processing?
 *
 * in_irq()       - We're in (hard) IRQ context
 * in_interrupt() - We're in NMI,IRQ,SoftIRQ context or have BH disabled
 * in_nmi()       - We're in NMI context
 * in_softirq()   - We have BH disabled, or are processing softirqs
 * in_serving_softirq() - We're in softirq context
 * in_task()      - We're in task context
 */
#define in_irq()             (hardirq_count())
#define in_interrupt()       (irq_count())
#define in_nmi()             (preempt_count() & NMI_MASK)
#define in_softirq()         (softirq_count())
#define in_serving_softirq() (softirq_count() & SOFTIRQ_OFFSET)
#define in_task()            (!(preempt_count() & \
                              (NMI_MASK | HARDIRQ_MASK | SOFTIRQ_OFFSET)))
```

Today's Agenda

- Mechanisms for bottom-half!

Interrupt Handler

- Top-halves (interrupt handlers) must run as fast as possible
  - They are interrupting other kernel/user code
  - They are often timing-critical b/c they deal with hardware
  - They run in interrupt context: no blocking
  - One or all interrupts are disabled
- Defer the less critical part of interrupt processing to a bottom-half

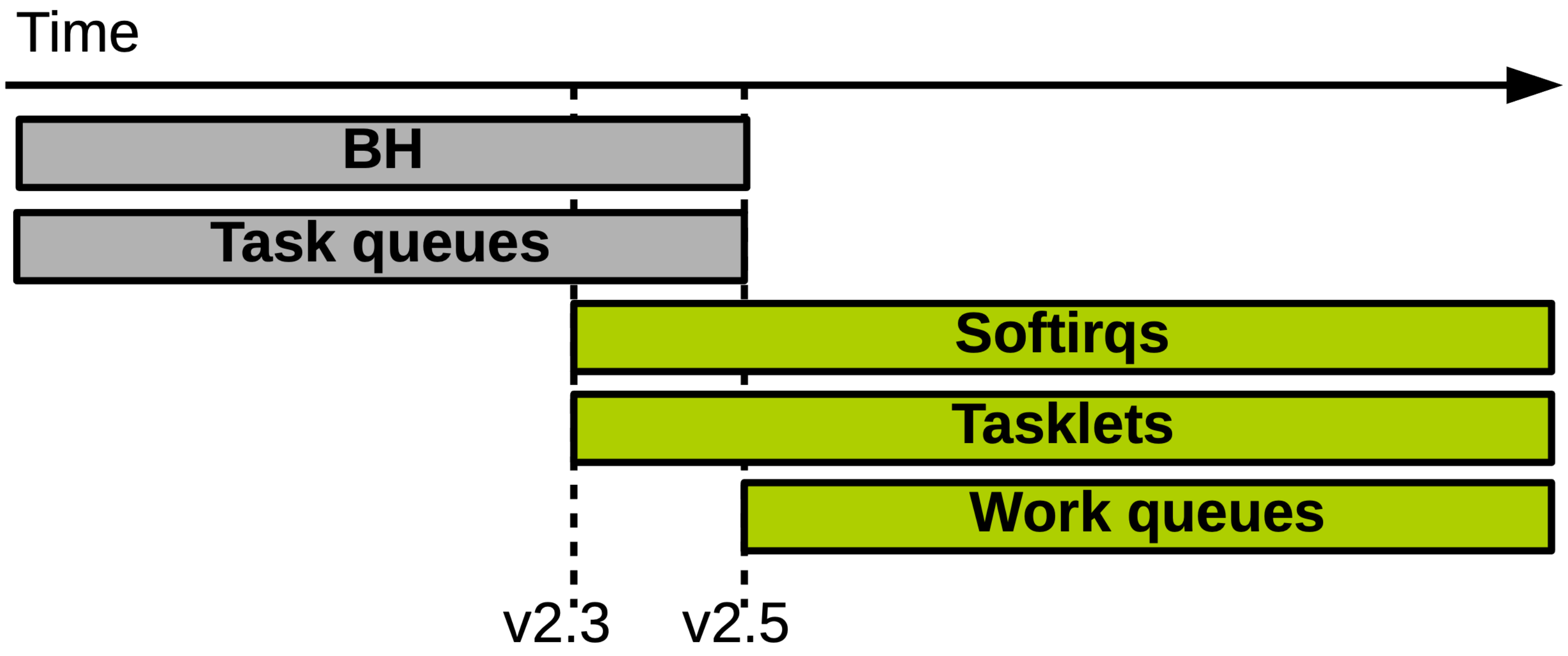
Top-halves vs. Bottom-halves

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

History of Bottom-Half

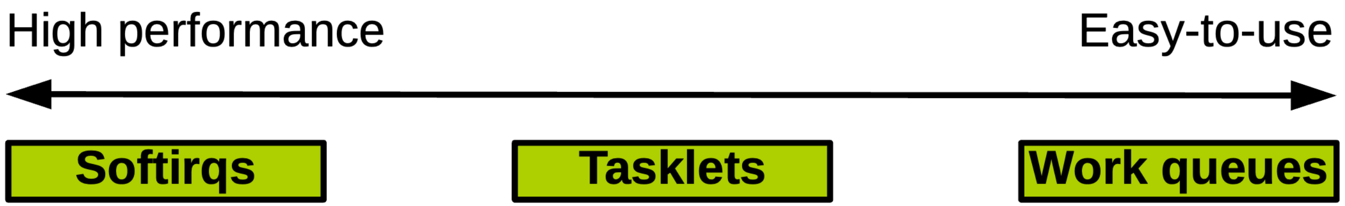
- "Top-half" and "Bottom-half" are generic terms, not specific to Linux
- Old "Bottom-Half" (BH) mechanism
  - a statically allocated list of 32 bottom halves
  - globally synchronized
  - easy-to-use yet inflexible and a performance bottleneck
- Task queues: queues of function pointers
  - still too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

- 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 (bult on softirq)
- Work queues run in process context
  – They can block and go to sleep



Softirq

- include/linux/interrupt.h
- kernel/softirq.c

```c
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

```c
/* PLEASE, avoid to allocate new softirqs, if you need not _really_ high
   frequency threaded job scheduling. For almost all the purposes
   tasklets are more than enough. F.e. all serial device BHs et
   al. should be converted to tasklets, not to softirqs.
 */

enum
{
	HI_SOFTIRQ=0,
	TIMER_SOFTIRQ,
	NET_TX_SOFTIRQ,
	NET_RX_SOFTIRQ,
	BLOCK_SOFTIRQ,
	IRQ_POLL_SOFTIRQ,
	TASKLET_SOFTIRQ,
	SCHED_SOFTIRQ,
	HRTIMER_SOFTIRQ,
	RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

	NR_SOFTIRQS
};
```

```c
/* softirq mask and active fields moved to irq_cpustat_t in
 * asm/hardirq.h to get better cache usage.  KAO
 */

struct softirq_action
{
	void	(*action)(void);
};
```

Executing Softirq

- 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 the following:
  - In the return from hardware interrupt code path
  - In the "ksoftirqd" kernel thread
  - In any code that explicitly checks for and executes pending softirqs

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

```c
/* 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

```
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

Using Softirq

- Softirq registration
  - Done by the driver at initialization phase
- 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's 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-CPU data (data unique to each processor and thus not requiring locking)
- 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

Register softirq

```
/* 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)
{
    /* ... */
}
```

Raise softirq

```
/* 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., no sleep)
  - Like softirq, all interrupts are enabled
- Restricted concurrency than softirq
  - The same tasklet cannot run concurrently

- include/linux/interrupt.h

```
/* 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-CPU linked lists
  - 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);
```

```c
/* 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();
```

```c
/* 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, be cautious
- Two of the same tasklets never run concurrently
  - B/c tasklet_action() checks TASKLET_STATE_RUN
- But two different tasklets can run at the same time on two different processors

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();
```

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?
  - Keep processing softirqs as they come in
    - » May starve userspace applications
  - Process one softirq at a time
    - » Should wait until the next interrupt occurance
    - » Sub-optimal on an idle system

ksoftirqd

- Per-CPU kernel thread to aid processing softirqs
- If the number of softirqs grows excessively, the kernel wakes up ksoftirqd with normal priority (nice 0)
  - No starvation of userspace applications
  - Running a softirq has the normal priority (nice 0)

```
→ linux git:(v6.12) ps ax -eo pid,nice,stat,cmd | grep ksoftirq
   16   0 S    [ksoftirqd/0]
   26   0 S    [ksoftirqd/2]
   32   0 S    [ksoftirqd/4]
   38   0 S    [ksoftirqd/6]
   44   0 S    [ksoftirqd/8]
   50   0 S    [ksoftirqd/10]
   57   0 S    [ksoftirqd/12]
   63   0 S    [ksoftirqd/13]
   69   0 S    [ksoftirqd/14]
   75   0 S    [ksoftirqd/15]
   81   0 S    [ksoftirqd/16]
   87   0 S    [ksoftirqd/17]
   93   0 S    [ksoftirqd/18]
   99   0 S    [ksoftirqd/19]
  105   0 S    [ksoftirqd/1]
  111   0 S    [ksoftirqd/3]
  117   0 S    [ksoftirqd/5]
```

Workqueue

- Workqueue defers work to a kernel thread
  - Always runs in process context
  - workqueues are schedulable and can therefore sleep
- By default, per-cpu kernel thread is created, “kworker/n”
  - The kernel also creates many other additional per-CPU work threads
  - Workqueue users can also create their own threads
    - » e.g., for performance and load balancing

Workqueue 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);
```

Workqueue: Worker Thread

- Worker threads execute the "worker_thread()" function
- Infinite loop doing the following:
  - Check if there is some work to do in the current pool
  - 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"
    - » remove work_struct object
  - Go to sleep until a new work is inserted in the workqueue

Workqueue: 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);
```

Workqueue: 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 */
```

Workqueue: Finishing Work

```c
/* 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);
```

Workqueue Example

```c
#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);
}
```

```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;
}
```

```
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
Workqueue	Process	None

All of these generally run with interrupts enabled

If there is shared data with an interrupt handler (top-half), need to disable interrupts or use locks

Tasklet is depreciated, use workqueue!

Threaded IRQ: Alternative to Top/Bottom-Half

- threaded interrupt handler seeks to reduce the time spent with interrupts disabled to a bare minimum
  - pushing the rest of the processing out into kernel threads
  - low latencies …
- How threaded interrupt helps
  - reducing complexity by simplifying locking between hard and soft parts of interrupt handling
  - threaded handlers will also help the debuggability of the kernel

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id)
```

Disabling softirq and tasklet

- The calls can be nested
  - Only the final call to local_bh_enable() actually enables bottom halves
- These calls do not idsable workqueue 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();
```

```
 *      request_threaded_irq - allocate an interrupt line
 *      @irq: Interrupt line to allocate
 *      @handler: Function to be called when the IRQ occurs.
 *                Primary handler for threaded interrupts.
 *                If handler is NULL and thread_fn != NULL
 *                the default primary handler is installed.
 *      @thread_fn: Function called from the irq handler thread
 *                  If NULL, no irq thread is created
 *      @irqflags: Interrupt type flags
 *      @devname: An ascii name for the claiming device
 *      @dev_id: A cookie passed back to the handler function
 *
 *      This call allocates interrupt resources and enables the
 *      interrupt line and IRQ handling. From the point this
 *      call is made your handler function may be invoked. Since
 *      your handler function must clear any interrupt the board
 *      raises, you must take care both to initialise your hardware
 *      and to set up the interrupt handler in the right order.
 *
 *      If you want to set up a threaded irq handler for your device
 *      then you need to supply @handler and @thread_fn. @handler is
 *      still called in hard interrupt context and has to check
 *      whether the interrupt originates from the device. If yes it
 *      needs to disable the interrupt on the device and return
 *      IRQ_WAKE_THREAD which will wake up the handler thread and run
 *      @thread_fn. This split handler design is necessary to support
 *      shared interrupts.
 *
 *      Dev_id must be globally unique. Normally the address of the
 *      device data structure is used as the cookie. Since the handler
 *      receives this value it makes sense to use it.
 *
 *      If your interrupt is shared you must pass a non NULL dev_id
 *      as this is required when freeing the interrupt.
 *
 *      Flags:
 *
 *      IRQF_SHARED             Interrupt is shared
 *      IRQF_TRIGGER_*          Specify active edge(s) or level
 *      IRQF_ONESHOT            Run thread_fn with interrupt line masked
 */
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id)
```