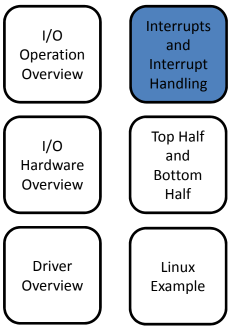
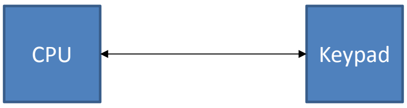
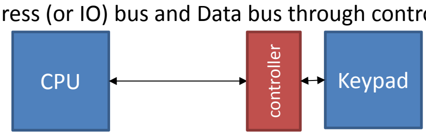


How can we communicate with I/O devices?

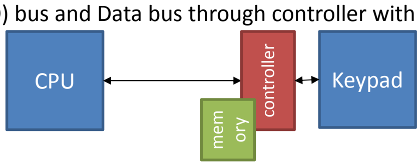
Address (or IO) bus and Data bus



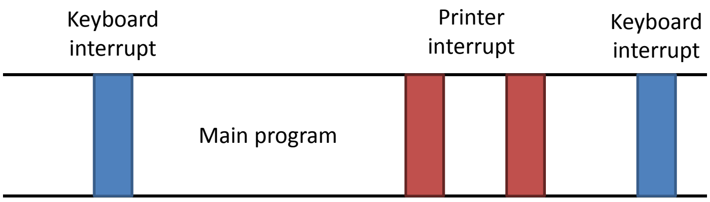
Address (or IO) bus and Data bus through controller



Address (or IO) bus and Data bus through controller with local memory



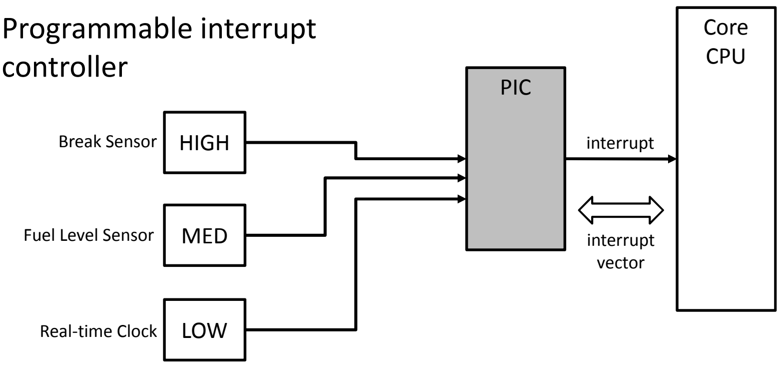
When does an interrupt occur?



3

Exceptions and interrupts

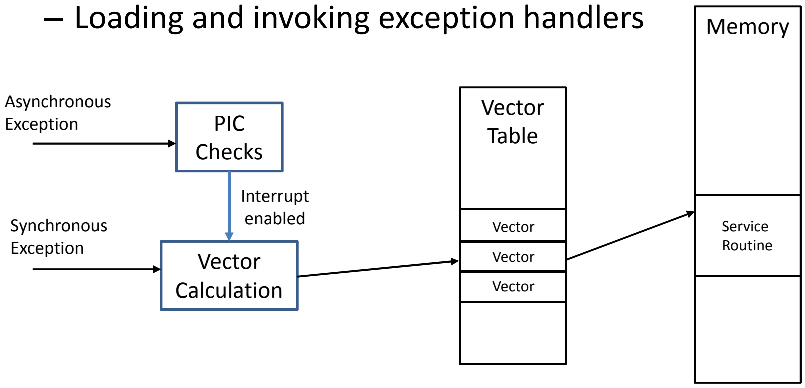
- How does exceptions and interrupts work ?
  - Programmable interrupt controller



Source: Qing Li "real-time concepts for embedded systems"

Exceptions and interrupts (Cont.)

- Processing general exceptions (Cont.)
  - Loading and invoking exception handlers

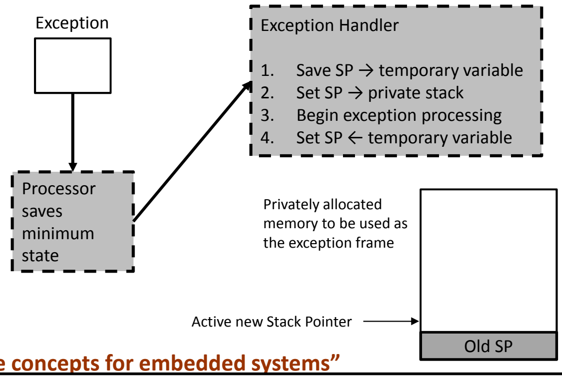


  - Nested exceptions and stack overflow

Source: Qing Li “real-time concepts for embedded systems”

Exceptions and interrupts (Cont.)

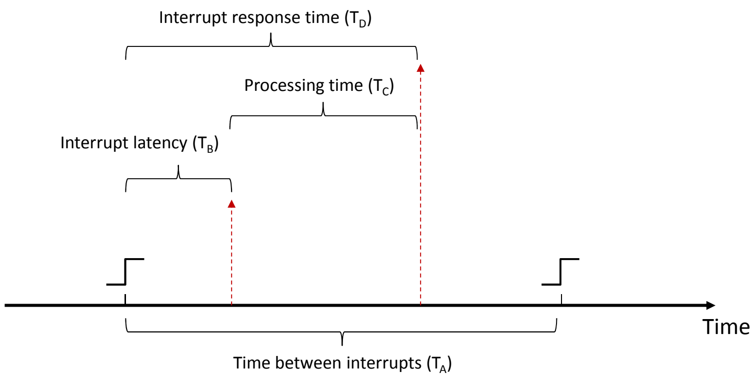
- Exception handlers
  - Exception frame
    - The exception frame is also called the interrupt stack in the context of asynchronous exceptions.



Source: Qing Li “real-time concepts for embedded systems”

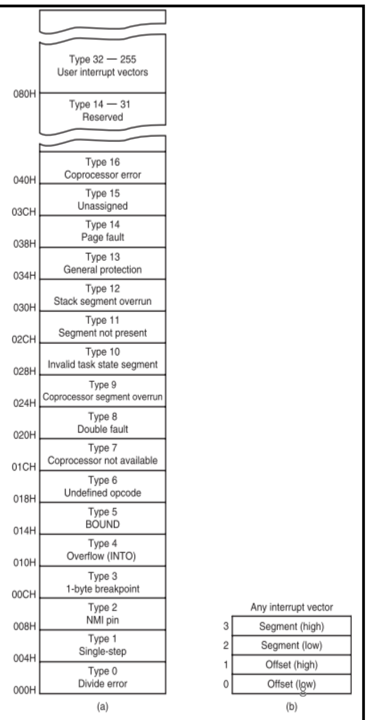
Exceptions and interrupts (Cont.)

- Exception timing



Source: Qing Li "real-time concepts for embedded systems"

Interrupt vector table



Software operations

- Microprocessor completes executing the current instruction
- It determines whether an interrupt is active by checking
  - (1) instruction executions
  - (2) single-step
  - (3) NMI
  - (4) coproces-sor segment overrun
  - (5) INTR
  - (6) INT instructions in the order presented
- The contents of the flag register are pushed onto the stack
- Both the interrupt (IF) and trap (TF) flags are cleared. This disables the INTR pin and the trap or single-step feature
- The contents of the code segment register (CS) are pushed onto the stack
- The contents of the instruction pointer (IP) are pushed onto the stack
- The interrupt vector contents are fetched, and then placed into both IP and CS

9

Important data pushed to stack

- flag register
- CS and IP

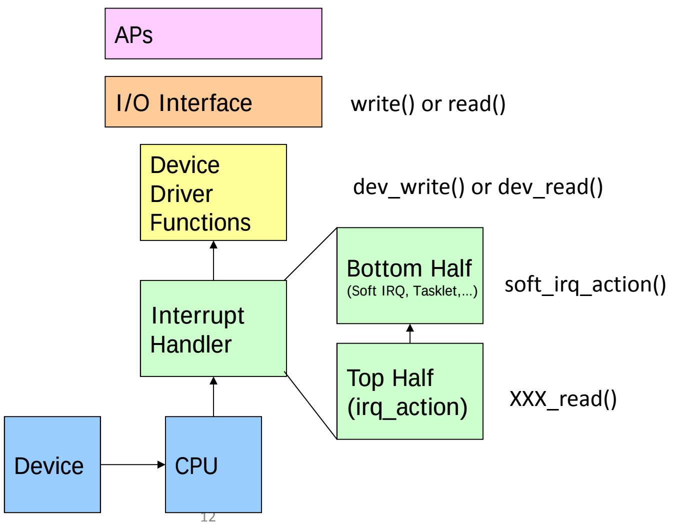
10

Software operations after IRET

- IF and TF are returned to the state prior to the interrupt
- Return address is restored
  - Interrupt type numbers 0, 5, 6, 7, 8, 10, 11, 12, and 13 push a return address that points to the offending instruction, instead of to the next instruction in the program (retry the instruction)

11

Interrupt, Interrupt Handler, and Device Driver



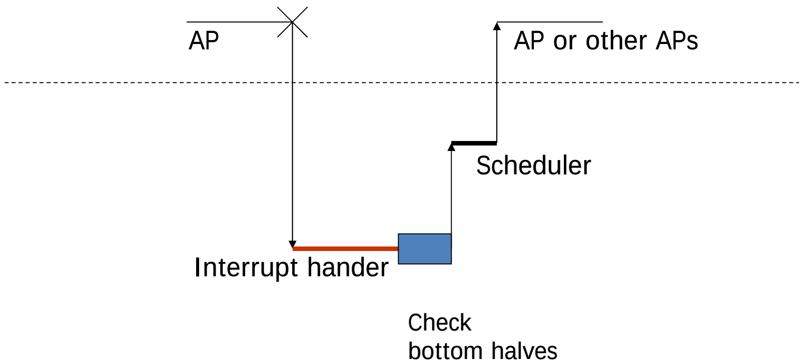
12

Top Halves vs. Bottom Halves

- Top halves
  - interrupt handlers (top halves), are executed by the kernel asynchronously in immediate response to a hardware interrupt
  - ASAP
- Bottom halves
  - to perform any interrupt-related work not performed by the interrupt handler itself
  - Process all interrupt related functions

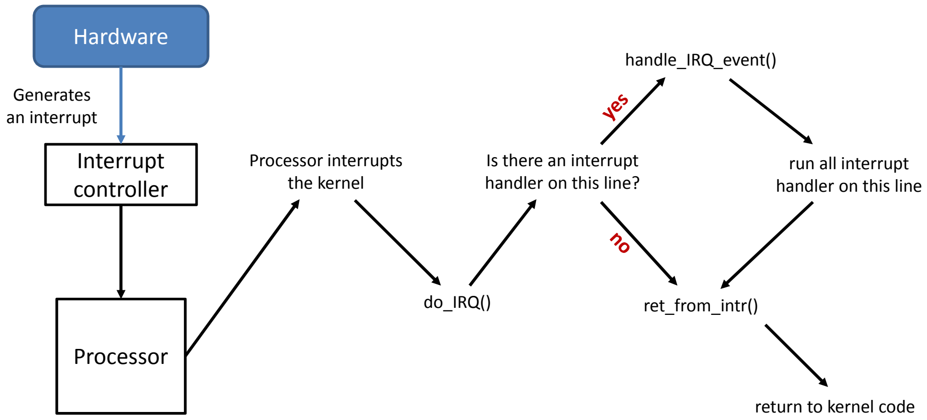
13

Top Halves vs. Bottom Halves



14

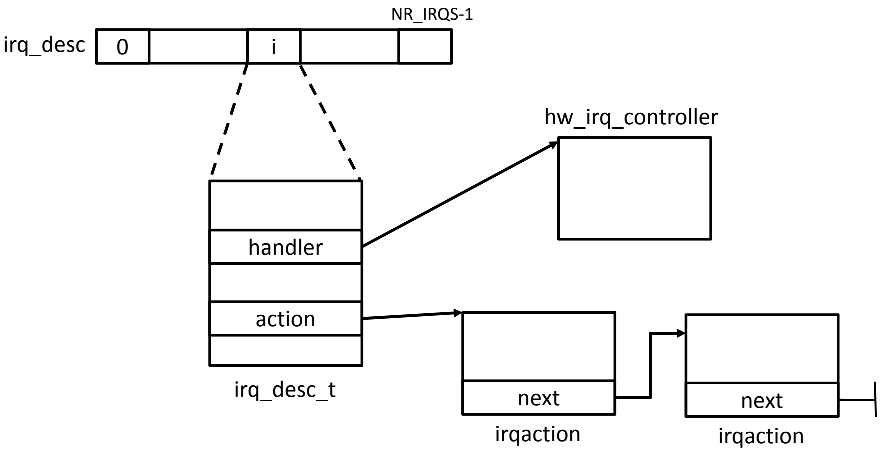
Interrupt Handling



15

Robert Love, **“Linux Kernel Development,” Second Edition**

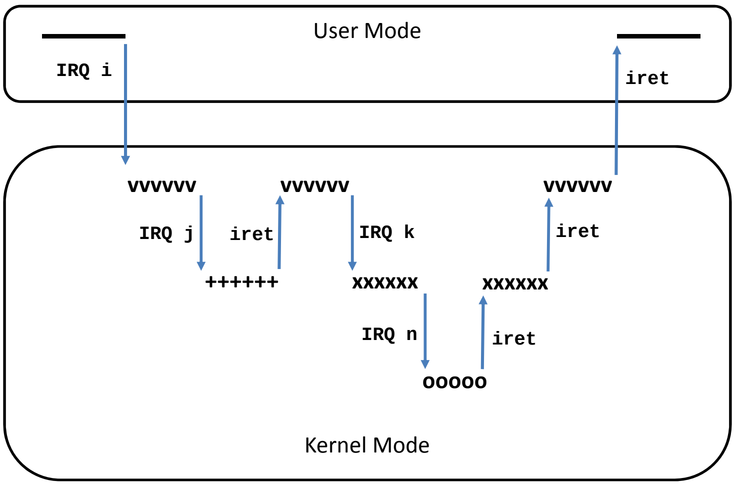
IRQ descriptors



16

Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel,” 3rd Edition

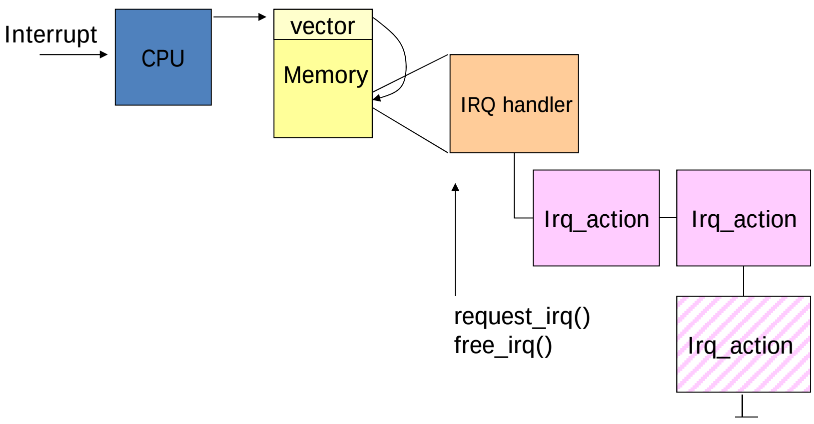
Nested Interrupt



17

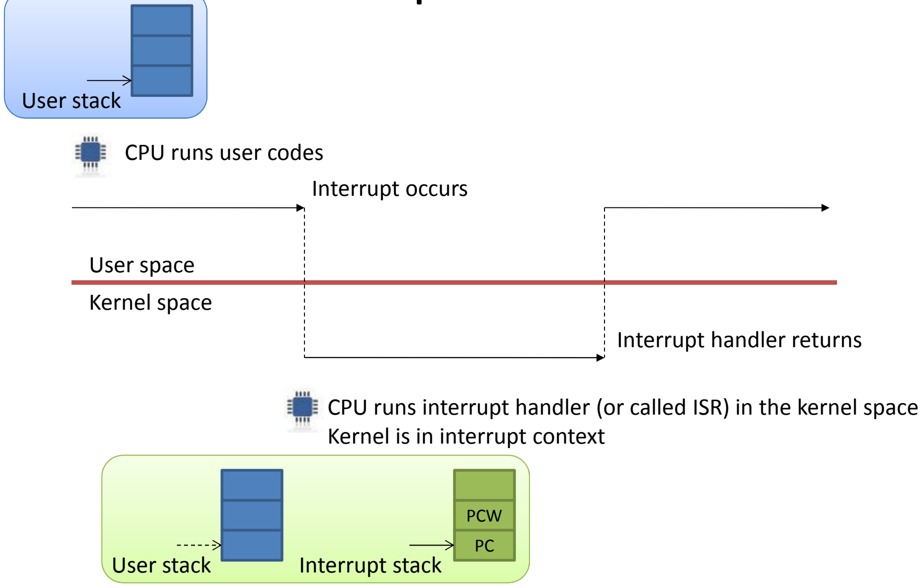
Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel,” 3rd Edition

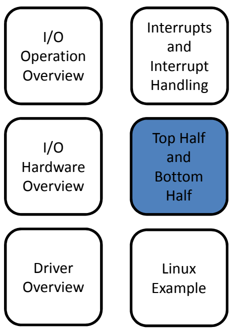
Registering Interrupt Handler



18

Interrupt Context

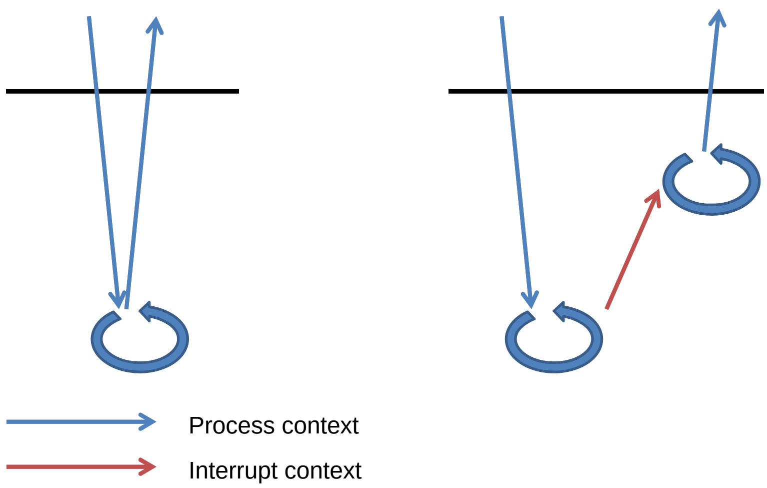




Comparisons of Different Bottom Half Implementation

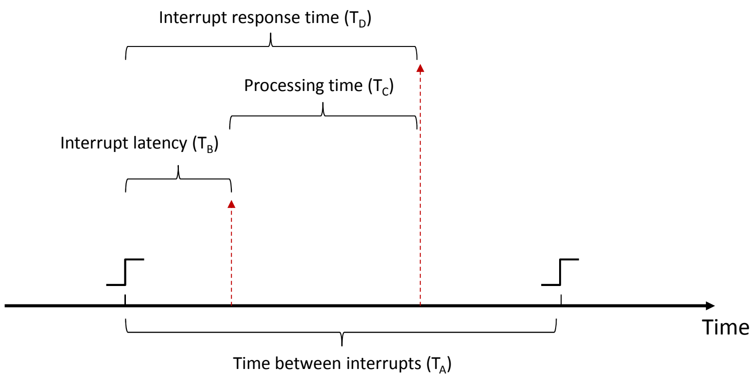
- why we need top half and bottom half
- softirq
- tasklet
- workqueue
- choice of different bottom half implementations

Process Context/Interrupt Context



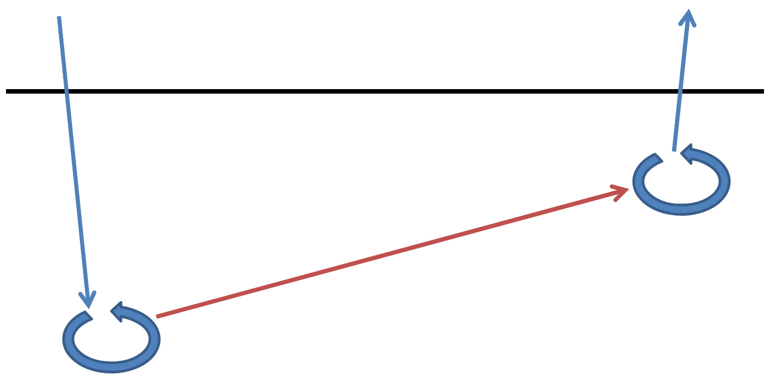
Exceptions and interrupts (Cont.)

- Exception timing



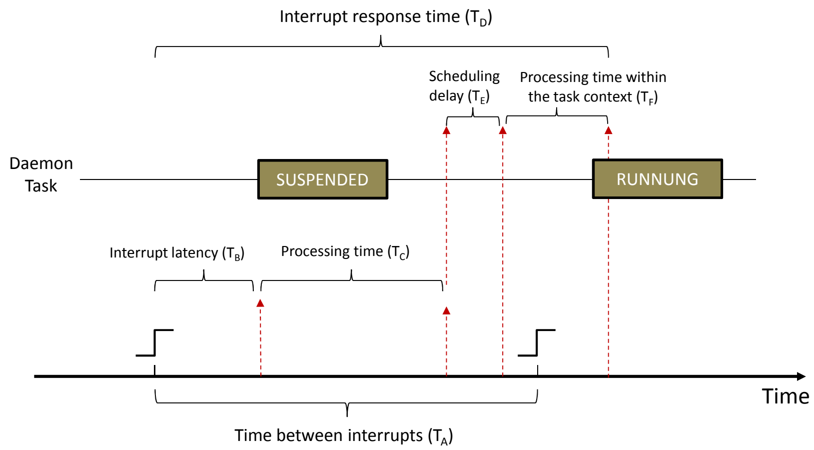
Source: Qing Li "real-time concepts for embedded systems"

Process Context/Interrupt Context



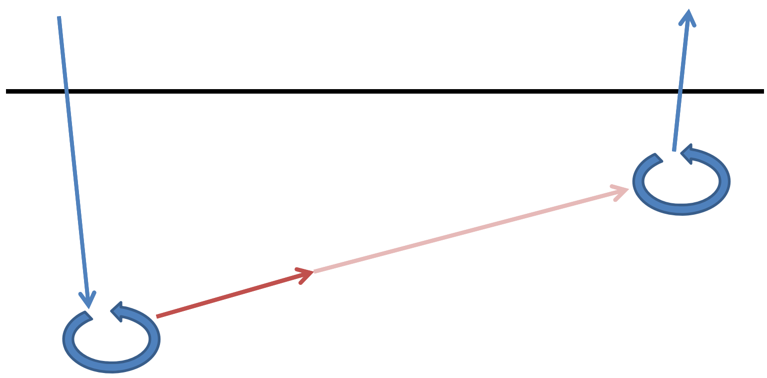
Exceptions and interrupts (Cont.)

- Exception timing (Cont.)

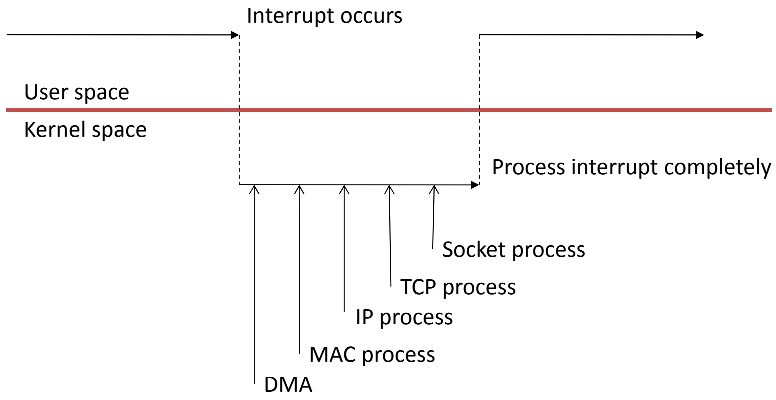


Source: Qing Li "real-time concepts for emb

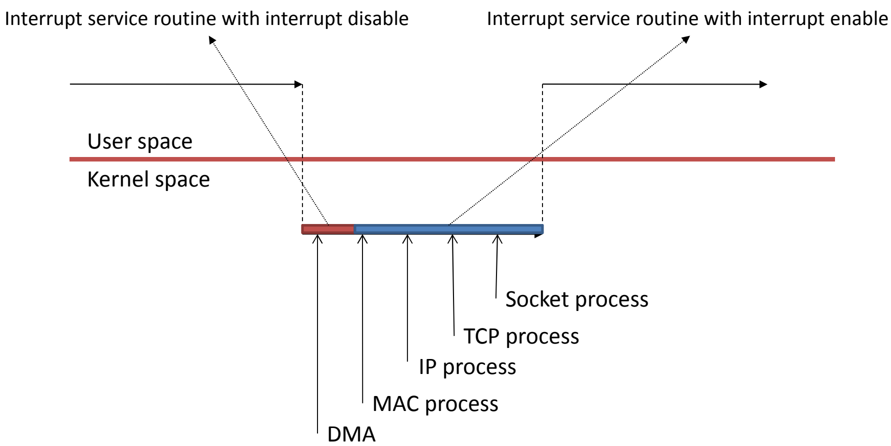
Process Context/Interrupt Context



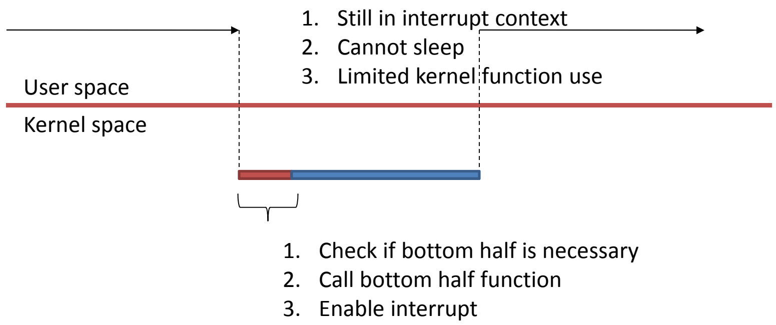
Why we need top half and bottom half



Two halves approaches



How to implement it



Top Halves vs. Bottom Halves

- Top halves
  - interrupt handlers (top halves), are executed by the kernel asynchronously in immediate response to a hardware interrupt
  - ASAP
- Bottom halves
  - to perform any interrupt-related work not performed by the interrupt handler itself
  - Process all interrupt related functions

I/O Operation Overview

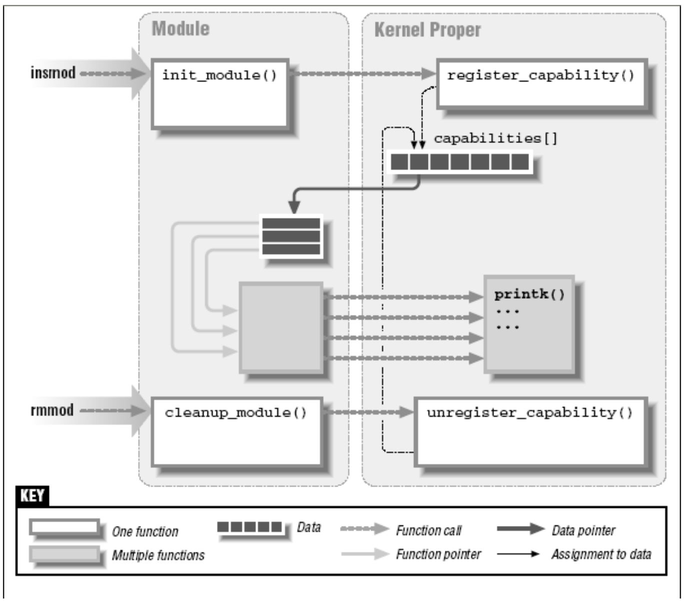
Interrupts and Interrupt Handling

I/O Hardware Overview

Top Half and Bottom Half

Driver Overview

Linux Example



Source: Linux Device Drivers 2nd Edition

Loadable module

```
/*
 * hello.c  Hello, World! As a Kernel Module
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

/*
 * hello_init  the init function, called when the module is loaded.
 * Returns zero if successfully loaded, nonzero otherwise.
 */
static int hello_init(void)
{
        printk(KERN_ALERT "I bear a charmed life.\n");
        return 0;
}

/*
 * hello_exit  the exit function, called when the module is removed.
 */
static void hello_exit(void)
{
        printk(KERN_ALERT "Out, out, brief candle!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Shakespeare");
```

File operations

- struct module *owner
- loff_t (*llseek) (struct file *, loff_t, int);
- ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
- ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
- ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
- ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);
- int (*readdir) (struct file *, void *, filldir_t);
- unsigned int (*poll) (struct file *, struct poll_table_struct *);
- int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
- int (*mmap) (struct file *, struct vm_area_struct *);
- int (*flush) (struct file *);
- int (*release) (struct inode *, struct file *);
- int (*fsync) (struct file *, struct dentry *, int);
- int (*aio_fsync)(struct kiocb *, int);
- int (*fasync) (int, struct file *, int);
- int (*lock) (struct file *, int, struct file_lock *);
- ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
- ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
- ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);
- ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
- unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
- int (*check_flags)(int)
- int (*dir_notify)(struct file *, unsigned long);

34

Device file operations

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

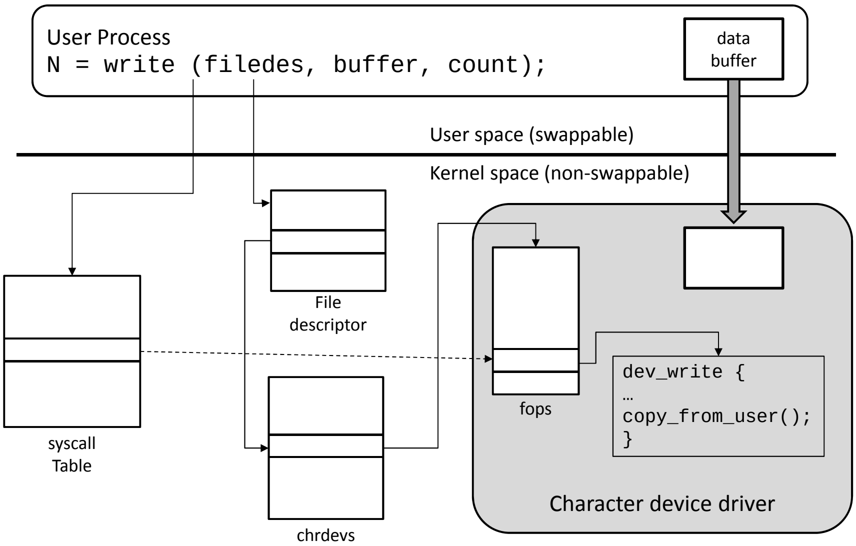
```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;     /* Char device structure      */
};
```

Device file operations

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
    printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        scull_trim(dev); /* ignore errors */
    }
    return 0;          /* success */
}
```

Device write



Linux Implementation

- Top half: implemented entirely via the interrupt handler
- Bottom half
  - multiple implementation
  - mechanisms are different interfaces and subsystems

Bottom Half	**Status**
BH	Removed in 2.5
Task queues	Removed in 2.5
Softirq	Available since 2.3
Tasklet	Available since 2.3
Work queues	Available since 2.5

38

Linux Implementation

- BH
  - The top half could mark whether the bottom half would run by setting a bit in a 32-bit integer
  - Each BH was globally synchronized
  - No two could run at the same time, even on different processors
  - This was easy to use, yet inflexible; simple, yet a bottleneck
  - Remove since 2.5
- Task queues
  - defined a family of queues
  - Each queue contained a linked list of functions to call
  - The queued functions were run at certain times
  - Drivers could register their bottom halves in the appropriate queue
  - It also was not lightweight enough for performance-critical subsystems, such as networking
  - Remove since 2.5

Linux Implementation

- Softirq
  - Since 2.3 development series
  - Softirqs are a set of 32 statically defined bottom halves that can run simultaneously on any processor
  - Softirqs are useful when performance is critical, such as with networking.
  - Two of the same softirq can run at the same time
  - Softirqs must be registered statically at compile-time
- Tasklets
  - Since 2.3 development series
  - dynamically created bottom halves that are built on top of softirqs.
  - Two different tasklets can run concurrently on different processors, but two of the same type of tasklet cannot run simultaneously
  - Tasklets are a good tradeoff between performance and ease of use
  - For most bottom-half processing, the tasklet is sufficient

40

Linux Implementation

```
/*
 * structure representing a single softirq entry
 */
struct softirq_action {
        void (*action)(struct softirq_action *); /* function to run */
        void *data;                              /* data to pass to function */
};
```

```
u32 pending = softirq_pending(cpu);

if (pending) {
    struct softirq_action *h = softirq_vec;

    softirq_pending(cpu) = 0;

    do {
        if (pending & 1)
            h->action(h);
        h++;
        pending >>= 1;
    } while (pending);
}
```

```
raise_softirq(NET_TX_SOFTIRQ);
```

41

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timer bottom half
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
SCSI_SOFTIRQ	4	SCSI bottom half
TASKLET_SOFTIRQ	5	Tasklets

42

ksoftirqd

```
for (;;) {
        if (!softirq_pending(cpu))
                schedule();

        set_current_state(TASK_RUNNING);

        while (softirq_pending(cpu)) {
                do_softirq();
                if (need_resched())
                    schedule();
        }

        set_current_state(TASK_INTERRUPTIBLE);
}
```

43

softirq

- rarely used (normally for interrupts occurs frequently and you really want to fast in SMP)
- statically allocated at compile-time (register and destroy softirqs)

```
static struct softirq_action softirq_vec[32];
```

```
/*
 * structure representing a single softirq entry
 */
struct softirq_action {
        void (*action)(struct softirq_action *); /* function to run */
        void *data;                              /* data to pass to function */
};
```

```
void softirq_handler(struct softirq_action *)
```

softirq

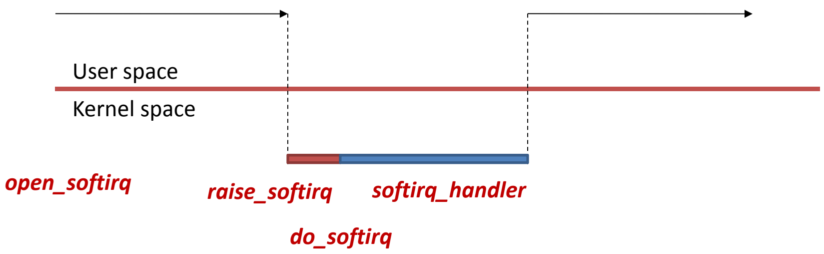
- Execution
  - softirq handler is registered at run-time via open_softirq()
  - interrupt handler marks its softirq for execution before returning (raise_softirq())
  - pending softirqs are checked for and executed in (do_softirq())
    - In the return from hardware interrupt code
    - In the ksoftirqd kernel thread
    - In any code that explicitly checks for and executes pending softirqs

softirq

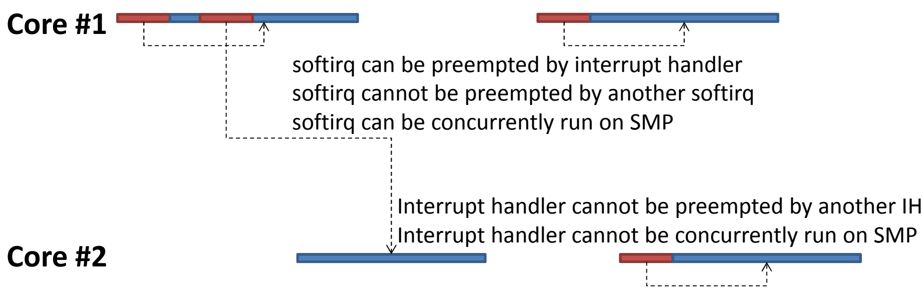
- The softirq handlers run with interrupts enabled and cannot sleep
- While a handler runs, softirqs on the current processor are disabled
- Another processor, however, can execute other softirqs
- Any shared data even global data used only within the softirq handler itself needs proper locking
- ***If a softirq obtained a lock to prevent another instance of itself from running simultaneously, there would be no reason to use a softirq***

softirq

- most softirq handlers resort to per-processor data and provide excellent scalability
- If you do not need to scale to infinitely many processors, then use a tasklet



softirq



softirq priority

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timer bottom half
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
SCSI_SOFTIRQ	4	SCSI bottom half
TASKLET_SOFTIRQ	5	Tasklets

ksoftirqd

- Why
  - Starve user process
  - Starve softirq
- per-processor kernel threads
- Help when the system is overwhelmed with softirqs
- not immediately process reactivated softirqs
- if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load
- The kernel threads run with the lowest possible priority (nice value of 19)

ksoftirqd

```
for (;;) {
        if (!softirq_pending(cpu))
                schedule();

        set_current_state(TASK_RUNNING);

        while (softirq_pending(cpu)) {
                do_softirq();
                if (need_resched())
                    schedule();
        }

        set_current_state(TASK_INTERRUPTIBLE);
}
```

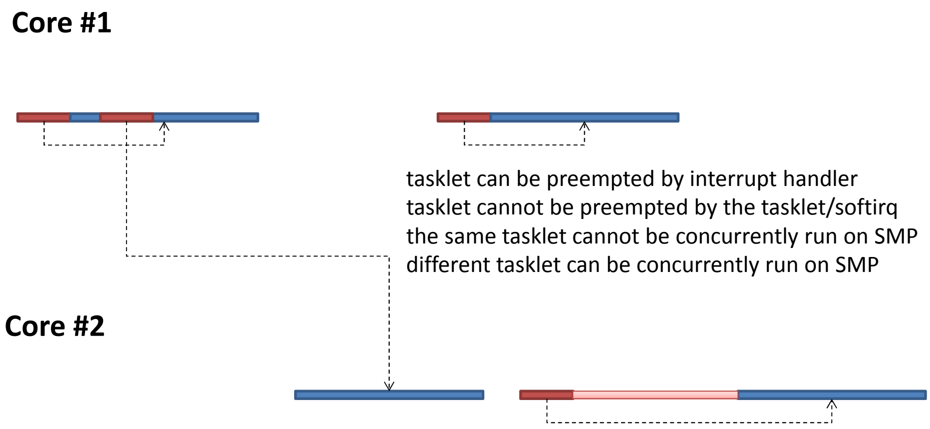
tasklet

- built on top of softirqs (they are softirq)
- Execution
  - Register

```
struct tasklet_struct {
        struct tasklet_struct *next;  /* next tasklet in the list */
        unsigned long state;          /* state of the tasklet */
        atomic_t count;               /* reference counter */
        void (*func)(unsigned long);  /* tasklet handler function */
        unsigned long data;           /* argument to the tasklet function */
};
```

  -
    - tasklet_init
  - Scheduled tasklets (the equivalent of raised softirqs)
    - tasklet_schedule() & tasklet_hi_schedule()
  - do_softirq() and tasklet_handler()

tasklet



tasklet can be preempted by interrupt handler
tasklet cannot be preempted by the tasklet/softirq
the same tasklet cannot be concurrently run on SMP
different tasklet can be concurrently run on SMP

Tasklet

```
struct tasklet_struct {
        struct tasklet_struct *next;  /* next tasklet in the list */
        unsigned long state;          /* state of the tasklet */
        atomic_t count;               /* reference counter */
        void (*func)(unsigned long);  /* tasklet handler function */
        unsigned long data;           /* argument to the tasklet function */
};
```

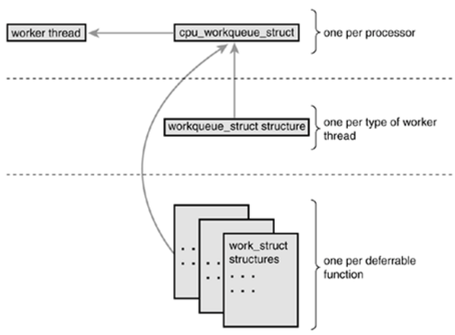
54

Tasklet

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timer bottom half
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
SCSI_SOFTIRQ	4	SCSI bottom half
TASKLET_SOFTIRQ	5	Tasklets

55

workqueue



choice of different bottom half implementations

- Softirqs
  - Concurrency
  - Protect shared data
  - networking subsystem
  - timing-critical and high-frequency uses
  - Good for SMP
- Tasklets
  - if the code is not finely threaded
  - Do not run concurrently
  - Shared with all tasklets
- Workqueue
  - run in process context
  - Can sleep

choice of different bottom half implementations

	Softirqs	**Tasklets**	**Work Queues**
Execution context	Deferred work runs in interrupt context.	Deferred work runs in interrupt context.	Deferred work runs in process context.
Reentrancy	Can run simultaneously on different CPUs.	Cannot run simultaneously on different CPUs. Different CPUs can run different tasklets, however.	Can run simultaneously on different CPUs.
Sleep semantics	Cannot go to sleep.	Cannot go to sleep.	May go to sleep.
Preemption	Cannot be preempted/scheduled.	Cannot be preempted/scheduled.	May be preempted/scheduled.
Ease of use	Not easy to use.	Easy to use.	Easy to use.
When to use	If deferred work will not go to sleep and if you have crucial scalability or speed requirements.	If deferred work will not go to sleep.	If deferred work may go to sleep.