A Deeper Dive into the Linux Kernel

Ahmed Ali-Eldin

1

Previously

- Introduction to the Linux Kernel
- Process Management
- Process Scheduling

Also

- Kernel data-structures

2

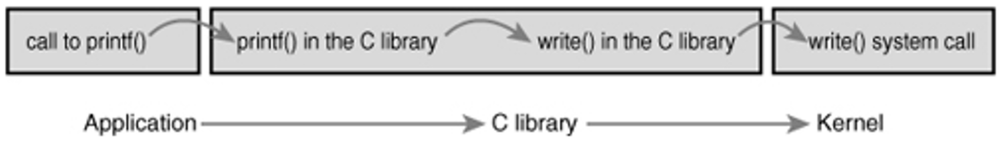
This lecture

- Syscalls
- Interrupts and Interrupt handlers
- Bottom Halves and deferring work

3

System Calls in Linux

- A single common layer between the user and the kernel
- Strives to be POSIX and SUSv3 (Single UNIX Specification) compatible
  - System calls provide a return value of type long
  - Each Syscall has a number like Minix
  - All numbers are in sys_call_table which is architecture dependent
    - Entry to the table for x86 https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscall_64.c



4

SysCall definition

```
SYSCALL_DEFINE0(getpid)
{
        return task_tgid_vnr(current); // returns current->tgid
}
```

- SYSCALL_DEFINE0 is simply a macro that defines a system call with no parameters (hence the *0*).
- Each Syscall has a defined handler

5

Mechanism

- Syscall from user-space results in an interrupt
- Interrupt causes an exception (trap) and the control is switched to kernel space
  - defined software interrupt on x86 is interrupt number 128
- Exception handler for the exception is called
  - The system call handler is named function system_call()
  - Architecture-dependent
    - on x86-64 it is implemented in assembly in entry_64.S
    - For fun: https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_64.S
  - System call number passed into the kernel via the *eax* register
    - User-space puts in the *eax* register the number corresponding to the desired system call.

6

How to implement a Syscall in Linux

http://soc.eurecom.fr/OS/labs_kernel.html

7

Why Not to Implement a System Call

• You need a syscall number, which needs to be officially assigned to you.

• After the system call is in a stable series kernel, it is written in stone. The interface cannot change without breaking user-space applications.

• Each architecture needs to separately register the system call and support it.

• System calls are not easily used from scripts and cannot be accessed directly from the filesystem.

• Because you need an assigned syscall number, it is hard to maintain and use a system call outside of the master kernel tree.

• For simple exchanges of information, a system call is overkill.

8

Other alternatives in Linux

- Linux Kernel Modules
  - More later!

9

Interrupts and Interrupt Handlers

10

Why Interrupts?

- Processors can be orders of magnitudes faster than the hardware they talk to
  - Not ideal for the kernel to issue a request and wait for a response
  - Idea: Deal with the hardware only after that hardware has actually completed its work
- Two ways to solve the variable speed problem
  - Polling
  - Interrupts

11

Interrupts

- Enable hardware to signal to the processor
  - Typing on the keyboard, moving the mouse, data ready from disk, etc
- Hardware devices generate interrupts asynchronously with respect to the processor clock—they can occur at any time
  - Electronic signals originating from hardware devices and directed into input pins on an interrupt controller
  - a simple chip that multiplexes multiple interrupt lines into a single line to the processor
- Different devices can be associated with different interrupts by means of a unique value associated with each interrupt.
- These interrupt values are often called interrupt request (IRQ) lines

12

IRQ

- Each IRQ line is assigned a numeric value
  - IRQ zero is the timer interrupt and IRQ one is the keyboard interrupt.
- Unlike Syscalls
  - Not all IRQ numbers are defined
  - For example, Interrupts associated with devices on the PCI bus are dynamically assigned.
  - The important notion is that a specific interrupt is associated with a specific device

13

Difference between interrupts and exceptions

- Handled mostly similarly by most architectures
- Interrupts origin is HW
- Exceptions usually software

14

Interrupt Handlers

- Each device that generates interrupts has an associated interrupt handler.
- The interrupt handler for a device is part of the device's driver
  - This why you need a driver for each device on your computer
  - The OS might not even know what is this device
  - Device drivers can be implemented in the kernel, or as a Linux Kernel Module
- Interrupt handlers are normal C functions
- They run in a special kernel context, called the *interrupt context* or *atomic context*
  - The handler is atomic, i.e., can not block!
  - Thus must be very quick!
    - A network interrupt on a 10gig card will need a lot of work!

15

Top Halves Versus Bottom Halves

- Split Interrupt handling in two parts: Top and bottom halves
- Interrupt handler is called the top-half
  - Run immediately upon receipt
  - Perform only time-critical work
    - Ack receiving the interrupt
    - Reset the HW to allow new interrupts to come
- Bottom half
  - Do the rest of the work in the future

16

Network Card example

- Network cards receive packets from the network
- Alert the kernel of their availability via interrupt
- Kernel responds by executing the network card’s registered interrupt
- Interrupt runs, acknowledges the hardware, copies the new networking packets from the network buffer into main memory, and readies the network card for more packets
  - Copying data does not need the processor to block as it can be done in the background
  - Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card’s buffer and thus packets being dropped
- Done and wait for bottom half!

17

Registering an Interrupt Handler

- Each device has one associated driver
- Each (interrupt based) driver registers one handler only
- Registered using *request_irq()*
  - Defined in https://github.com/torvalds/linux/blob/master/include/linux/interrupt.h
- On success, request_irq() returns zero.
- A common error is -EBUSY, irq already in use

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

18

Handler flags

- IRQF_DISABLED: instructs the kernel to disable all interrupts when executing this interrupt handler. When unset, interrupt handlers run with all interrupts except their own enabled.
  - What do you think is the common value for this flag?
- IRQF_TIMER: This flag specifies that this handler processes interrupts for the system timer.
- IRQF_SHARED and `dev`: More on this later, enables shared handlers
- IRQF_SAMPLE_RANDOM: Should this interrupt metadata be used for the internal kernel random number generator?

19

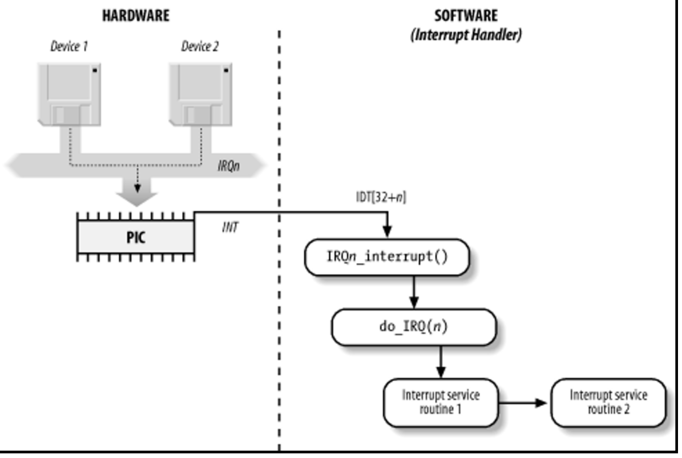
Reentrancy and Interrupt Handlers

- When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors
- This is to prevent the interrupt from being overridden by another interrupt of the same type
- Some interrupts even require masking/disabling all other interrupts
  - Linux is designed to keep these ones to a minimum
- The same interrupt handler is never invoked concurrently to service a nested interrupt.
  - This greatly simplifies writing your interrupt handler.
  - No need to look at concurrency

20

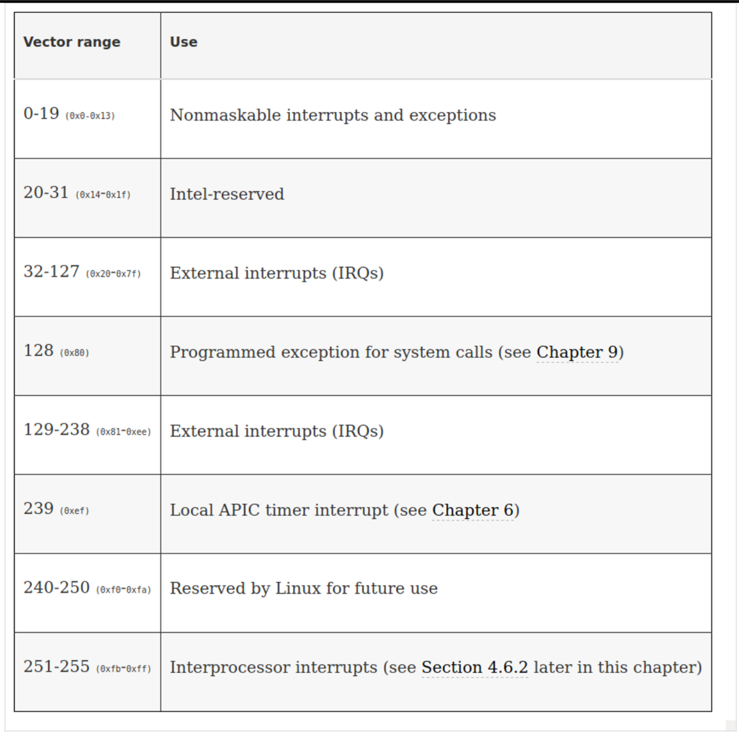
Why interrupt sharing?

- Limited number of pins on the any chip including the interrupt controller
- We need to be able to support tens, if not hundreds of drivers in parallel
- Solution:
    - Share interrupts
    - Must be supported in HW
    - Must be supported in driver



21

Interrupt vector Table

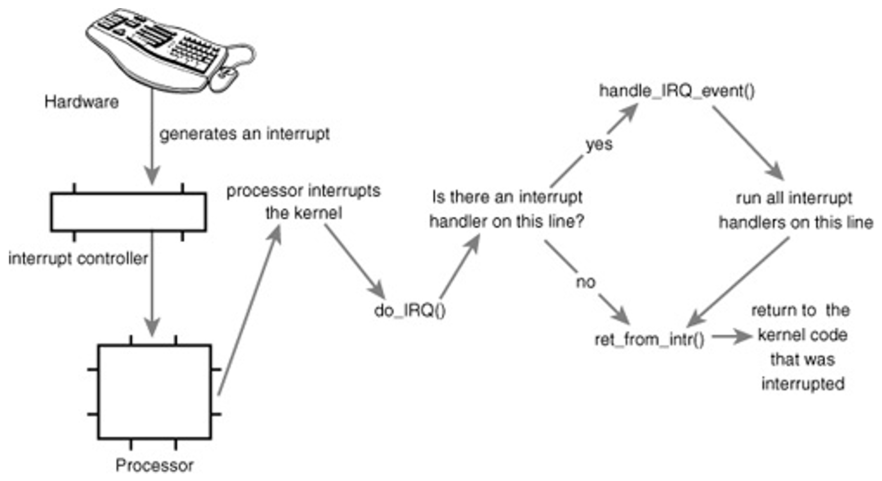


Vector range	Use
0-19 (0x0-0x13)	Nonmaskable interrupts and exceptions
20-31 (0x14-0x1f)	Intel-reserved
32-127 (0x20-0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls (see Chapter 9)
129-238 (0x81-0xee)	External interrupts (IRQs)
239 (0xef)	Local APIC timer interrupt (see Chapter 6)
240-250 (0xf0-0xfa)	Reserved by Linux for future use
251-255 (0xfb-0xff)	Interprocessor interrupts (see Section 4.6.2 later in this chapter)

22

Implementing Interrupt Handlers

- Architecture dependant


Hardware
generates an interrupt
interrupt controller
Processor
processor interrupts the kernel
do_IRQ()
Is there an interrupt handler on this line?
yes
handle_IRQ_event()
run all interrupt handlers on this line
no
ret_from_intr()
return to the kernel code that was interrupted

23

Interrupt control methods in the kernel

For Synchronization purposes as we discuss later

Per processor

No global disabling for all cores

Function	Description
`local_irq_disable()`	Disables local interrupt delivery
`local_irq_enable()`	Enables local interrupt delivery
`local_irq_save()`	Saves the current state of local interrupt delivery and then disables it
`local_irq_restore()`	Restores local interrupt delivery to the given state
`disable_irq()`	Disables the given interrupt line and ensures no handler on the line is executing before returning
`disable_irq_nosync()`	Disables the given interrupt line
`enable_irq()`	Enables the given interrupt line
`irqs_disabled()`	Returns nonzero if local interrupt delivery is disabled; otherwise returns zero
`in_interrupt()`	Returns nonzero if in interrupt context and zero if in process context
`in_irq()`	Returns nonzero if currently executing an interrupt handler and zero otherwise

24

Bottom Halves

25

Bottom Half

- Performs most of the actual work
- Design should offload as much work as possible to the bottom half
- Decision is left entirely up to the device-driver author
    - Good driver designers do not hog your processor
- Point of a bottom half is not to do work at some specific point in the future, but simply to defer work until any point in the future when the system is less busy and interrupts are again enabled.
- Typically runs right after the handler, but with all interrupts enabled

26

Good design

- If the work is time sensitive, perform it in the interrupt handler.
- If the work is related to the hardware, perform it in the interrupt handler.
- If the work needs to ensure that another interrupt (particularly the same interrupt) does not interrupt it, perform it in the interrupt handler.
- Everything else goes to the bottom half.

27

No single way to implement bottom-half

- Old days, all bottom halves were implemented using one single mechanism called (BH)
  - Provided a statically created list of 32 bottom halves for the entire system.
  - But computer evolved to more than these 32 BHs
- Quickly became a bottleneck

28

Method1: Task Queues

- A method of deferring work and a replacement for the BH mechanism was introduced later
- Defines a family of queues
  - Each queue contained a linked list of functions to call
  - queued functions were run at certain times, depending on which queue they were in.
    - A priority based queuing model
- Problem: was not lightweight enough for performance-critical subsystems, such as networking.
- Now replaced by work-queues (more later)

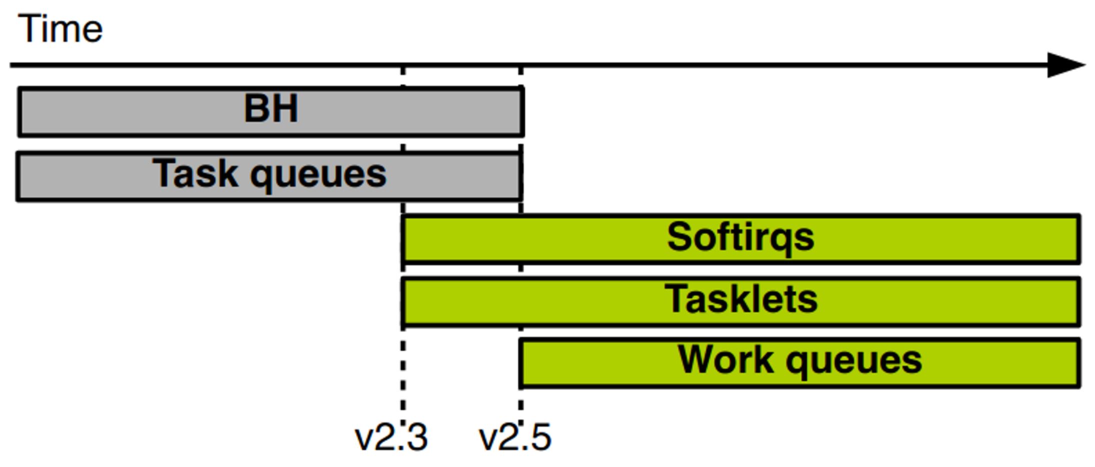
29

Method2: Softirqs and Tasklets

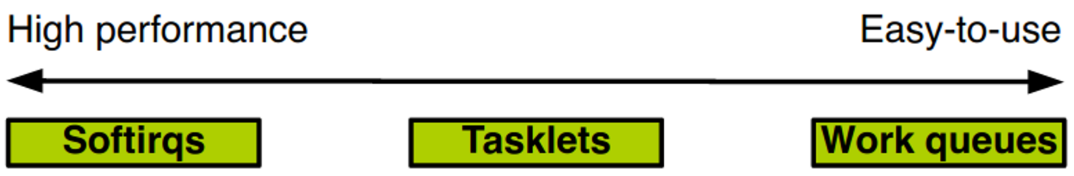
- Introduced in kernel 2.3, and completely replaced BH in v2.5
- Softirqs are a set of **statically defined** bottom halves that can run simultaneously on any processor; even two of the same type can run concurrently.
- Tasklets are flexible, dynamically created bottom halves built on top of softirqs.
  - Two different tasklets can run concurrently on different processors, but two of the same type of tasklet cannot run simultaneously
- Use Softirqs when performance is critical, such as with networking
  - However, more care needed as they can run simultaneously

30

Summary



31



32

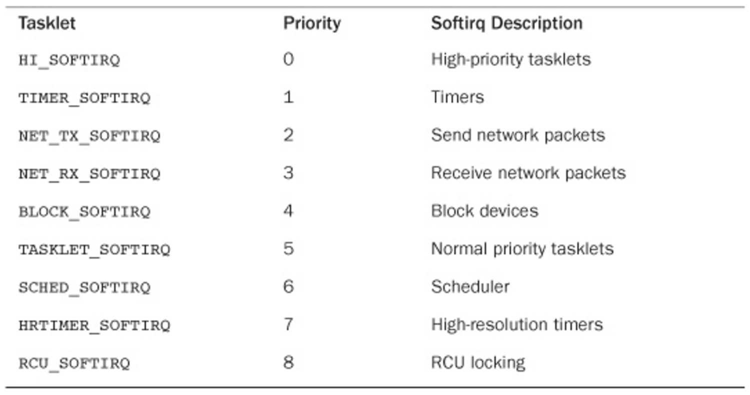
Method3: Threaded interrupts

- The main goal of threaded IRQs is to reduce the time spent with interrupts disabled to a bare minimum
- You do not even have to schedule the bottom half yourself.
- The bottom half is then executed in a dedicated kernel thread
- Not covered more in the lecture!

33

Only eight SoftIRQs

- You can define more, but why?
- Use Tasklets or threaded interrupts!



Tasklet	Priority	Softirq Description
`HI_SOFTIRQ`	0	High-priority tasklets
`TIMER_SOFTIRQ`	1	Timers
`NET_TX_SOFTIRQ`	2	Send network packets
`NET_RX_SOFTIRQ`	3	Receive network packets
`BLOCK_SOFTIRQ`	4	Block devices
`TASKLET_SOFTIRQ`	5	Normal priority tasklets
`SCHED_SOFTIRQ`	6	Scheduler
`HRTIMER_SOFTIRQ`	7	High-resolution timers
`RCU_SOFTIRQ`	8	RCU locking

34

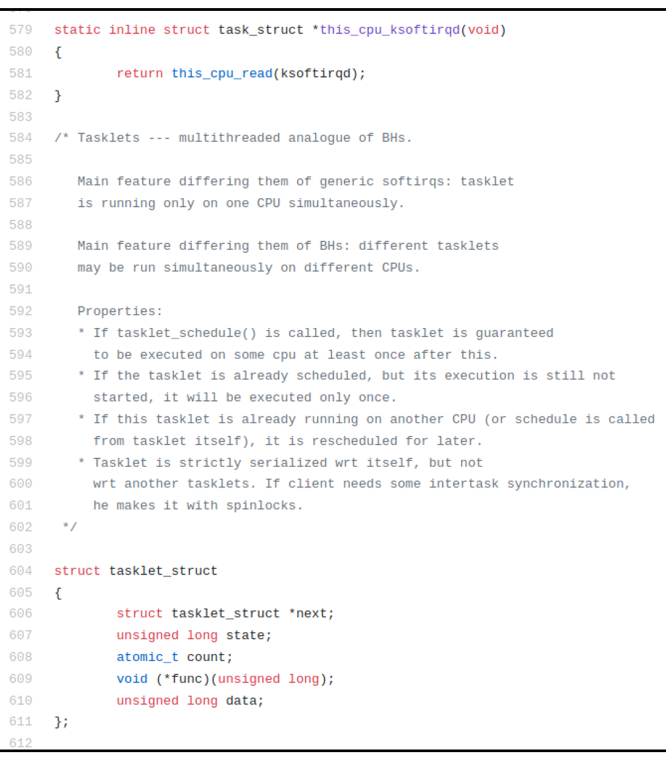
Tasklets

- Built on top of softirqs, thus they are softirqs
- Softirqs are required only for high-frequency and highly threaded uses.
    - Tasklets for everything else
- Tasklets are represented by two softirqs: HI_SOFTIRQ and TASKLET_SOFTIRQ.
    - Difference: HI_SOFTIRQ-based tasklets run prior to the TASKLET_SOFTIRQ-based tasklets.

35

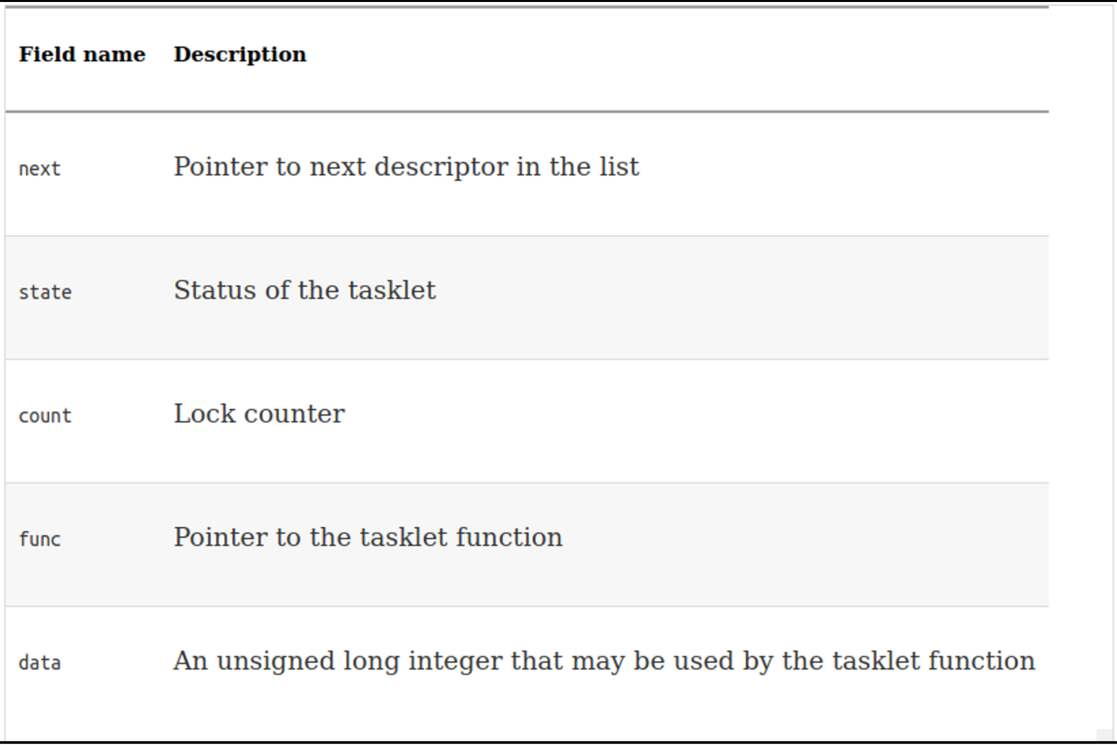
The Tasklet Structure

- *tasklet_struct* structure defined in <



```
static inline struct task_struct *this_cpu_ksoftirqd(void)
{
        return this_cpu_read(ksoftirqd);
}

/* Tasklets --- multithreaded analogue of BHs.

   Main feature differing them of generic softirqs: tasklet
   is running only on one CPU simultaneously.

   Main feature differing them of BHs: different tasklets
   may be run simultaneously on different CPUs.

   Properties:
   * If tasklet_schedule() is called, then tasklet is guaranteed
     to be executed on some cpu at least once after this.
   * If the tasklet is already scheduled, but its execution is still not
     started, it will be executed only once.
   * If this tasklet is already running on another CPU (or schedule is called
     from tasklet itself), it is rescheduled for later.
   * Tasklet is strictly serialized wrt itself, but not
     wrt another tasklets. If client needs some intertask synchronization,
     he makes it with spinlocks.
 */

struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

36



Field name	Description
`next`	Pointer to next descriptor in the list
`state`	Status of the tasklet
`count`	Lock counter
`func`	Pointer to the tasklet function
`data`	An unsigned long integer that may be used by the tasklet function

37

ksoftirqd

- Per-processor kernel thread
- Help in the processing of softirqs when the system is overwhelmed with softirqs
- The kernel processes softirqs in a number of places, most commonly on return from handling an interrupt.
- A softirq can raise itself so that it runs again (for example, the networking subsystem’s softirq raises itself)
    - At high load, this can starve user-processes
    - if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load with the lowest possible priority

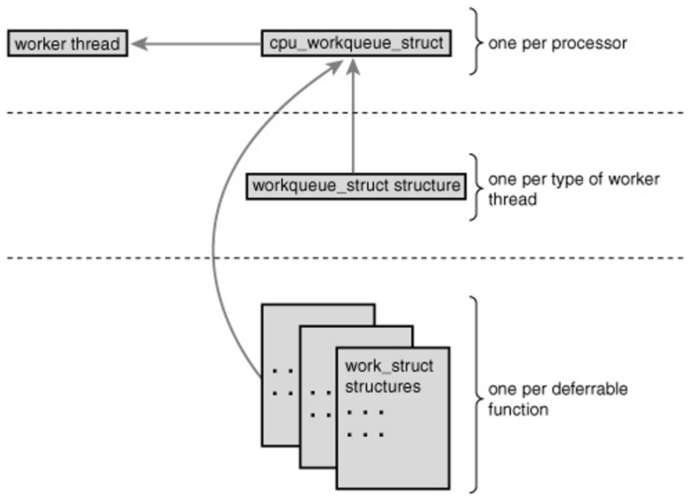
38

Work Queues

- Different form of deferring work into a kernel thread called worked thread that runs in process context, thus can sleep
  - If the deferred work needs to sleep, work queues are used.
  - If the deferred work need not sleep, softirqs or tasklets are used
  - Useful for situations in which you need to allocate a lot of memory, obtain a semaphore, or perform block I/O.

39

Work Queues implementation


worker thread
cpu_workqueue_struct
one per processor
workqueue_struct structure
one per type of worker thread
work_struct structures
one per deferrable function

40

Locking between bottom halves

Something for next lecture :)

41