Kernel Synchronization and Interrupt Handling

Oliver Sengpie, Jan van Esdonk

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2015-01-14

Table of Contents

The timers

- Real time clock (RTC)
- Time stamp counter (TSC)
- Programmable interval timer (PIT)
- CPU Local timer (APIC)
- High precision timer (HPET)
- ACPI Power Management timer

Why do we need Interrupts?

- Getting information from hardware when it's needed (Hardware Interrupts)
- Processing data when it's needed (Software Interrupts)
- One solution would be polling
  - Overhead through unnecessary checks
- Better solution: Interrupts
  - Work is only done when needed

What is an Hardware Interrupt?

- Electrical signal, asynchronously emitted by a hardware device
- Each device is mapped to an Interrupt-Request-Line
- Processed by an Interrupt Controller
- Forces the CPU to handle the interrupt

Dealing with Hardware Interrupts

- Kernel gets interrupted by the CPU
- Each IRQ Line has Interrupt Handlers
    - C function
    - Implements a specific function prototype
    - Executed in its own context (Interrupt Context)

Writing your own Interrupt Handler

- Function prototype:

```
static irqreturn_t intr_handler(int irq, void *dev_id,
 struct pt_regs *regs)
```

- Registering the Handler at the IRQ:

```
int request_irq(unsigned int irq,
irqreturn_t (*handler)(int, void *, struct pt_regs *),
unsigned long irqflags,
const char *devname,
void *dev_id)
```

Kinds of Interrupt Handlers

- `SA_INTERRUPT`
  - Short execution time
  - Not interruptable
- `SA_SAMPLE_RANDOM`
  - Source for Kernel Entropy Pool
- `SA_SHIRQ`
  - Multiple Handlers possible
  - Handler detects which device emitted the interrupt
  - Every registered handler is checked for the emitting device

Interrupt Context

- Not associated to any process
- Running process gets dumped to kernel memory
- No ability to sleep
- Its own stack (1 page)
- Simple rules:
  - Be quick and short
  - Only use the stack if absolutely necessary

Managing Interrupts

- Enabling/Disabling Interrupts
  - local_irq_enable() / local_irq_disable()
- Disabling specific Interrupts
  - disable_irq(unsigned int irq)
- Status of Interrupts (Activation, Context)
  - irqs_disabled()
  - in_interrupt() / in_irq()
- Interrupt statistics
  - /proc/interrupts

Overview of Interrupt Handling

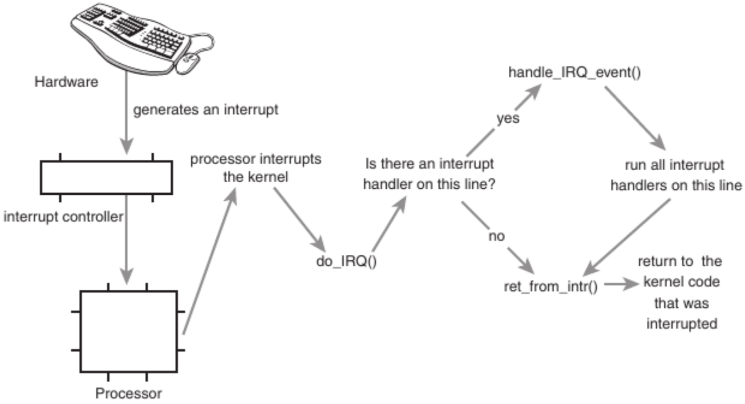
Hardware
generates an interrupt
interrupt controller
Processor
processor interrupts the kernel
do_IRQ()
Is there an interrupt handler on this line?
yes
no
handle_IRQ_event()
run all interrupt handlers on this line
ret_from_intr()
return to the kernel code that was interrupted

Figure: Source: Robert Love: Linux Kernel Development 3rd Edition

Why do we need Bottom Halves

- Interrupt Handlers need to be finished fast
    - All processes on the CPU are blocked by Interrupts
    - Only used for time critical tasks and arrival confirmations
- Bottom Halves are used for data processing
- Interrupts are enabled while executing a BH

Types of Bottom Halves

- Deprecated types
    - BH-Interface
    - Task Queues
- softirq
- Tasklets
- Work-Queues

softirqs

- Statically allocated at compile time
- Maximum of 32 softirqs
- implemented as a structure
  - Points to a function to run
  - softirq handler gets a pointer to the structure
    - `void softirq_handler(struct softirq_action *)`
  - Needs to be marked for execution in a bitmask
  - Gets executed by a looper over the softirq_vec array

Usage of softirqs

1. Assigning a priority index
2. Registering the handler
   - `open_softirq()`
3. Marking it for execution
   - `raise_softirq()`
   - Handler gets executed by the next do_softirq() run

Tasklets

- based on softirqs
- implemented as tasklet_struct
- State field manages its execution status
- Organized in a linked list Tasklet structures
- Gets executed in the do_softirq() cycle

Writing your own Tasklet

1. Declare the Tasklet
    - statically:
      `DECLARE_TASKLET(name, func, data)`
    - dynamically:
      `tasklet_init(t, tasklet_handler, dev)`
2. Implement a Tasklet-Handler
    - function prototype:
      `void tasklet_handler(unsigned long data)`
3. Schedule the Tasklet
    - `tasklet_schedule(&your_tasklet)`

Work Queues

- Executed by their own kernel thread
- Executed in process context
- Organized in linked list of work_queues structures

Usage of Work Queues

1. Declare Work Queue at runtime
    - statically:
      `DECLARE_WORK(name, void (*func)(void *), void *data)`
    - dynamically:
      `INIT_WORK(struct work_struct *work, void (*func)(void *)`
      `void *data)`
2. Implement Handler
    - `void work_handler(void *data)`
3. Schedule the Work Queue
    - `schedule_work(&work)`

Which BH should I choose?

- softirq
  - High priority operations
  - Short execution time
  - Thread safe
- Tasklet
  - Short execution time
  - Doesn't need to be thread safe
- Work Queues
  - Big operations (i/o)
  - Doesn't need to be thread safe

What are synchronization mechanisms?

- Functions and structures provided by the kernel
- Prohibit race conditions in shared memory

Why are these mechanisms necessary?

- The kernel is not running serially
- Most kernels use preemption
- Multicore processor systems
- Kernel control paths are interleaving

Synchronization primitives

There are a few elementary mechanisms to synchronize programs:

- Per-CPU variables
- Local interrupt disabling
- Local softirq disabling
- Optimization and memory barriers
- Atomic operations
- Spin locks
- Semaphores and completions
- Seqlocks
- Read-copy-update (RCU)

Where are them not needed

Before we take a closer look at the mechanisms:
There are cases in which no synchronization mechanisms are needed.

Where are them not needed

- Interrupts
  Interrupts disable their IRQ-lines, cannot be nested or be interleaved by deferrable functions and are nonblocking and nonpreemptable.
- Softirqs and tasklets
  They cannot be interleaved on their CPU and are nonblocking and nonpreemptable
- Unique structures in Tasklets
  They need no synchronization, because they can only be executed with one instance on all CPUs.

Single core

Per-CPU variables

Simplest way to make sure no other CPU can corrupt the memory.
With Interrupthandlers, Softirqs and Tasklets race condition secure.

Interrupt and softirq disabling

Can be used to make interrupt handlers thread secure.

Optimization and memory barriers

Instructions which affect the compiler. They avoid optimization and reorganization of the instructions in the compiler. Optimization barriers assure that the assambly instructions remain in the same order and memory barriers ensure that the instructions before the barrier are finished when the barrier is reached.

Multiple cores

Atomic operations

Atomic operations appear to be instantanious to the hardware.

Spin locks

Make sure that readers and writers don't get into conflicts accessing a ressource. There are different types of spin locks:

- simple spin locks (every action is equal)
- read write spin locks

The waiting processes "spin", i.e. they execute a tiny instruction loop to check whether the lock has yet been released.

Semaphores and completions

Waiting processes are put to sleep and awakened, when they are at the first place of the wait queue and the lock is free.

Seqlocks

Like Spinlock but with priviliges to the writers. The readers have to check after the reading if the data they read is valid.

RCU

Use of pointer - and thus only available for data on the heap.

The Big Kernel Lock

Forces the processor to allow only one process in kernel space.