Linux Interrupt Handlers

The Interrupt Handler API; Concurrency Considerations

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Overview

Roadmap:

- Terminology
- Examples
- Registering, unregistering handlers
- `proc/interrupts`

Overview

Roadmap:

- Shared handlers
- Handler data
- Concurrency considerations

Terminology

An *interrupt*:

"A signal that the hardware can send when it wants the processor's attention..."

-- Alessandro Rubini, Linux Device Drivers. O'Reilly. 2005.

Terminology

Lots of devices:

- Serial ports (UARTS, SSP, I2C, ...)
- Network controllers (Ethernet, USB, CAN, ...)
- Timers and counters, GPIO, ...

Just about everything!

Terminology

An *interrupt handler*:

- Code that responds to a device's interrupt request
- A.k.a. “Interrupt service routine”

Terminology

"Interrupt handler" != "device driver":

- Device drivers *might* involve interrupts, handlers
- It really depends on the device!
- Linux itself doesn't associate the two

Minimal Example

```c
static int gpio_pb29_irqs;

static irqreturn_t
gpio_pb29_irq (int irq, void *unused)
{
  printk("%s: %d\n", __FUNCTION__, gpio_pb29_irqs);
  if (++gpio_pb29_irqs > 10) {
    printk(KERN_ERR "%s: disabling\n", __FUNCTION__);
    disable_irq(gpio_to_irq(AT91_PIN_PB29));
  }
  return IRQ_HANDLED;
}
```

Registering an Interrupt Handler

`request_irq()`

- Registers an interrupt handler
- Returns 0 on success

```
#include <linux/interrupt.h>

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

Registering an Interrupt Handler

`unsigned int irq`

- Interrupt channel identifier
- Enumerations are architecture-specific
- Defined in `<asm/irq.h>`

Registering an Interrupt Handler

```
irq_handler_t handler
```

- Pointer to the handler function
- Handler is invoked when interrupt is detected

```
typedef int irqreturn_t;
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

Registering an Interrupt Handler

`unsigned long irqflags`

- Various channel-specific parameters
- Some channels might not support all flags

Registering an Interrupt Handler

`IRQF_DISABLED`	— Disable interrupts during handling
`IRQF_SAMPLE_RANDOM`	— Interrupt is a source of randomness
`IRQF_SHARED`	— Allow sharing of the interrupt line
`IRQF_TRIGGER_RISING`	— Trigger on rising edge
`IRQF_TRIGGER_FALLING`	— Trigger on falling edge
`IRQF_TRIGGER_HIGH`	— Trigger on high level
`IRQF_TRIGGER_LOW`	— Trigger on low level

Registering an Interrupt Handler

`const char *name`

- Plaintext name, for `/proc/interrupts`

`void *dev_id`

- Pointer to handler's private data
- Returned to handler during interrupts

Registering an Interrupt Handler

```
static irqreturn_t
gpio_pb29_irq(int irq, void *unused);

ret = request_irq(gpio_to_irq(AT91_PIN_PB29),
                  gpio_pb29_irq,
                  IRQF_SAMPLE_RANDOM,
                  "gpio_pb29",
                  (void*)0);
```

Registering an Interrupt Handler

`free_irq()`

- Unregisters an interrupt handler
- Disables the interrupt channel if appropriate
- Use the same `dev_id` passed to `request_irq()`

```
void free_irq(unsigned int irq, void *dev_id);
```

The `/proc/interrupts` Interface

The `/proc` filesystem:

- Pseudo-files of processor-specific information
- Pseudo-directories of process-specific information

```
$ cat /proc/interrupts
          CPU0
 0:  174501757    IO-APIC-edge  timer
 1:         11    IO-APIC-edge  i8042
 7:          2    IO-APIC-edge  parport0
 ...
```

Interrupt Handler Functions

```
irqreturn_t handler(int irq, void *dev_id)
```

- Interrupt handler signature

```
static irqreturn_t
gpio_pb29_irq (int irq, void *unused)
```

Interrupt Handler Functions

`int irq`

- Interrupt channel number

Interrupt Handler Functions

`void *dev_id`

- A.k.a. "device identifier"
- Used to distinguish devices on shared channels
- Provided in `request_irq()`
- Must be unique for `IRQF_SHARED`

Interrupt Handler Functions

```
struct irq_data {
  int data;
};
struct irq_data irq_data;

static int
example_open (struct inode *inode, struct file *pfile)
{
  int ret;
  ret = request_irq(IRQNUM, irq_handler, 0,
                    "example", &irq_data);
  return ret;
}
```

Interrupt Handler Functions

`irqreturn_t`

- Return value from handler

`IRQ_NONE`	— Did not handle the interrupt request
`IRQ_HANDLED`	— Handled the interrupt request
`IRQ_RETVAL(x)`	— Helper macro that returns `NONE` or `HANDLED`

```
#define IRQ_RETVAL(x) ((x) != 0)
```

Interrupt Handler Functions

`"irq255:  nobody cared"`

- All registered handlers returned `IRQ_NONE`, or
- There were no registered handlers
- Followed by an OOPS output

Interrupt Handler Functions

Things to investigate:

- Is my return value correct?
- Is my interrupt handler detecting all possible requests?
- Am I registering with the wrong channel?
- Is it the wrong type of signal (level vs. edge)?

No Sleeping!

Interrupt handlers must be atomic:

- No pending on a semaphore
- No waiting on a completion
- No sleep-based delays

Be especially careful during:

- Memory allocation (!)
- User memory access

No Sleeping!

Memory allocation:

- Avoid it in an interrupt handler!
- Preallocate in your interface code
- Pass a pointer via `dev_id`

No Sleeping!

User memory access:

- Target page might be swapped out
- Use a preallocated intermediate buffer
- The `mmap()` APIs can lock pages
- Consider DMA

Enabling and Disabling Interrupts

```
void enable_irq(unsigned int irq)
```

- Enables an interrupt channel

```
void disable_irq(unsigned int irq)
```

- Disables the requested interrupt request line
- Can be called from an interrupt handler
- Affects all registered handlers
- (You probably won't do this often)

Enabling and Disabling Interrupts

Linux uses “lazy” interrupt disables:

- The `disable_irq()` only sets a flag
- Pending interrupts are still serviced
- Interrupts are physically disabled later
- (This laziness is usually well-hidden)

“When do I enable interrupts?”

`request_irq()`:

- Registers the interrupt handler
- Enables the interrupt channel in the controller
- Does NOT enable the device to assert interrupts!

`enable_irq()`:

- Unmasks an interrupt line
- Does NOT enable the device to assert interrupts!

“When do I enable interrupts?”

Follow this sequence:

- Probe for the device, if possible
- Disable interrupts at the device, if possible
- Initialize the associated driver, if there is one
- Initialize the device

“When do I enable interrupts?”

And then:

- Register the interrupt handler via `request_irq()`
- Enable interrupts at the device, as appropriate

Polling Interrupt Handlers

Polled vs. interrupt-driven:

- Your interrupt handler might support both!
- Done in some ethernet drivers, elsewhere

```
/* set up a transmit */
...;

/* send it out */
while (interrupt_handler(&data) == IRQ_HANDLED)
{
}
```

Shared Data Issues

Sharing data with interrupt handlers:

- Potentially leads to *race conditions* (bad!)
- Demands attention to concurrency issues

Code very cautiously!

- Perfectly safe if done properly

Shared Data Issues

Commonly-used facilities:

- Completions
- Spinlocks

Also:

- Work queues
- Tasklets

Shared Data Issues

Forget about:

- Mutexes, semaphores
- Disabling interrupts

Completions

The *completion* API:

- One-way communication between contexts
- “The requested job is done”

Conceptually similar to a semaphore, but:

- Semaphores are optimized for the “available” case
- Semaphores are expensive during contention
- Semaphores are going away (!)

struct completion

```c
#include <linux/completion.h>

struct completion c;

irqreturn_t
example_irq_handler(int irq, void *unused)
{
  /* get data... */
  ...;

  complete(&c);
  return IRQ_HANDLED;
}
```

struct completion

```
static ssize_t
example_read(struct file *file, char __user *buf,
             size_t count, loff_t *offp)
{
  if (wait_for_completion_interruptible(&c))
    return -EINTR;

  /* data is ready... */
  ...;

  return count;
}
```

Recap

The interrupt API:

- `request_irq()`
- Interrupt handlers
- No sleeping!

Recap

Concurrency:

- Sharing data with interrupt handlers
- Demands use of completions, etc.
- Failure to do so leads to races

Linux Interrupt Handlers

The Interrupt Handler API; Concurrency Considerations

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer