

1. Installing an interrupt handler

The basics:

If there's no handler registered for an interrupt, the kernel will ack it but do nothing with it.
Kernel keeps track of which driver/handler is associated with an interrupt line.

Register interrupt handler:

`int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);`

retval is zero on success, retval is negative on error (EBUSY if already allocated).

=>"irq" the IRQ number being requested

=>"handler" the IRQ callback

=>"flags" bitmask on how to manage interrupt

=>"dev_name" name printed out in `/proc/interrupts`

=>"dev_id" pointer needed for handler freeing when interrupt shared. if the interrupt isn't shared, can be set to NULL.

=>Otherwise, set to some unique pointer within driver.

Interrupt bitmask:

- SA_INTERRUPT:

- "Fast interrupt".

- Execute handler with interrupts disabled on local CPU.

- SA_SHIRQ:

- Interrupt can be "shared".

- SA_SAMPLE_RANDOM:

Unregister handler:

- Interrupts generated by device can contribute to entropy pool for random number generation (`/dev/random` and `/dev/urandom`).

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

- Handler registration can be done on device instantiation or on `open()`.

- Best to register handler on first `open()` / free on last `close()`.

The /proc interface:

- Kernel keeps track of how times each type of interrupt occurs (internal counter).

- Do `"cat /proc/interrupts"` to see:

- Interrupt lines that currently have registered handlers

- The number of times each type of interrupt occurred for each CPU.

- The PIC (Programmable Interrupt Controller) configuration for the interrupt.

- The driver(s) that have registered handlers for the given interrupt, as provided by the "dev_name" parameter of `request_irq()`.

- Do `"cat /proc/stat"` and look for the "intr" line to see:

- The total number of all interrupts that occurred since boot

- The total number of interrupts of a given type that occurred since boot, each entry being separated by a space.

- Number of entries in both files will vary greatly between archs.

-

Autodetecting the IRQ number:

- Sometimes interrupt numbers are known in advance

- Ask PCI config for interrupt number

- Can ask device to generate interrupt and monitor result.
- Can't probe shared interrupts
- Kernel-assisted probing:
- Only for nonshared interrupts
- Usually for ISA only
- `<linux/interrupt.h>`
- unsigned long `probe_irq_on(void);`

- int `probe_irq_off(unsigned long);`
- `retval` is bitmask of unsigned interrupts
- record `retval` for passing to `probe_irq_off()`
- Enable interrupts after this call
- Configure device to emit interrupt

- Call to ask kernel which interrupt occurred
- Disable interrupts before this call
- May need to insert delay prior to calling this function to give time for the interrupt to occur.
- `retval` is > 0 if only one interrupt occurred
- `retval` is 0 if no interrupt occurred
- `retval` is < 0 if more than one interrupt occurred
- Probing can take a lot of time (20 ms for framegrabber)

Do-it-yourself probing:

- Best to do probing only once at module load time
- Most nonPC platforms don't need probing and above
- functions are placeholders (including most PPC, and MIPS implementations).
- Loop on all possible interrupts
- Record interrupt handler for a given interrupt
- Configure device generating an interrupt
- Wait for interrupt to occur

Check to see if handler was called
Free interrupt handler

Fast and slow interrupts:

- Old kernel abstraction
- Currently, the only difference between handlers is those that use the `SA_INTERRUPT` flag and those that don't.
- If `SA_INTERRUPT` is used, handler is called with all local interrupts disabled.
- Most drivers shouldn't use `SA_INTERRUPT` unless absolutely required.
- The internals of interrupt handling on the x86:

Assembly-generating macros in `arch/i386/kernel/entry.S` push
int number on stack and call `do_IRQ` from `arch/i386/kernel/irq.c`.
• `do_IRQ` does:

handle_IRQ_event:

Mask and ack interrupt

- Requests spinlocks for given IRQ number (to avoid other processors from trying to handle it.)
- If handler register, call `handle_IRQ_event` to invoke it Otherwise unlock and return
- If `SA_INTERRUPT` not set, reenables interrupts Invoke handler(s)
- Check for scheduling (processes may have been woken up as a result of interrupt).

Implementing a handler

The basics

Role:

- Interact with device regarding interrupt, usually ACK device.
 - Transfer data to/from device as required
 - Wake up processes waiting for device
 - Defer as much work as possible to tasklets or work queues.
 - Restrictions as to what handler can do
 - Restrictions similar to those of timer:
- => Can't access user-space

Handler arguments and return value:

`int request_irq(unsigned int irq, irqreturn_t (*handler) (int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);`

- 1: "irq", the irq number
- 2: "*dev_id", private data (same as passed to `request_irq()`, can pass internal device "instance" pointer to easily find instance on interrupt).
- 3: "*regs", the CPU registers at interrupt occurrence, seldom used.

- `retval` is status of interrupt handling:
- `IRQ_HANDLED`, an interrupt occurred. Should also be used in no way to determine if interrupt did occur.
- `IRQ_NONE`, no interrupt occurred. Interrupt was spurious or is shared.
- Macro for generating return value depending on variable (nonzero means interrupt handled): `IRQ_RETVAL(var);`

Enabling and disabling interrupts:

- Try to avoid in as much as possible
- No way to disable interrupts on all processors in the same time.

Disabling a single interrupt:

- `<asm/irq.h>`
- `void disable_irq(int irq);`
- `void disable_irq_nosync(int irq);`
- Wait for interrupt handler if it's running and disable it Careful with deadlocks
- Disables interrupt without checking if handler is running
- `void enable_irq(int irq);`
- May play with PIC's mask
- Calls can be nested
- Disabling all interrupts:
- Disable interrupts on local CPU
- `<asm/system.h>`

- void local_irq_save(unsigned long flags);
- void local_irq_disable(void);
- Restores interrupt flags
- void local_irq_enable(void);
- Disables without recording flags
- void local_irq_restore(unsigned long flags);
- Disables interrupts and saves local interrupt flags
- Reenables local interrupts
- No nesting possible: use local_irq_save()

Top and bottom halves

- Interrupt handlers must finish rapidly (top half)
- Lengthy work deferred to later in "bottom half" with all interrupts enabled.

Usually:

- Top half records device data to temporary buffer and scheds BH. Network code pushed packet up stack, while actual processing done in BH.
- BH further does whatever scheduling is needed

Tasklets: Overview

- Will be covered in detail in Ch.8

Reminder: from "Using Linux in Embedded Systems":

- Runs in software interrupt context
- Only one tasklet of a given type will ever be running in the same time in the entire system.
- Even if rescheduled multiple times, will only run once.
- Interrupt may occur while tasklet is running => use appropriate locks.
- Tasklets run on the same CPU where they scheduled

API reminder:

- DECLARE_TASKLET()
- tasklet_init()
- tasklet_schedule()

Workqueues:

- Issue function within work queue process context

Reminder:

- Can sleep
- Can't access userspace
- Can use system default work queue

Interrupt sharing

- Interrupt sharing a must on modern hardware (used by PCI).

Installing a shared handler:

- Use same request_irq()
- Difference with non-shared handlers:
 - Use the SA_SHIRQ flag
 - Pass non-NULL, unique ID in *dev_id, kernel complains otherwise as it may oops at irq freeing.
- Registration fails if other handlers have registered without setting the SA_SHIRQ flag.
- On interrupt occurrence, kernel invokes every registered handler for the given interrupt, passing it

the *dev_id provided on registration.

- Each handler must determine if the device it handles issued the interrupt, and return IRQ_NONE otherwise.
- *dev_id's importance is shown when free_irq() is called since that's the only way to know which handler should be removed from shared list.

Running the handler

- Remember that interrupt is shared => don't disable interrupts.
- The /proc interface and shared interrupts
- /proc/interrupts shows list of driver sharing a given interrupt.

Tasklets

Somewhat similar to timers:

- Run in soft interrupt context
- Run on same CPU where registered
- Received unsigned long argument provided on registration.
- Can reregister themselves
- Tasklet lists are percpu

Difference from timers:

- No specific time for execution
- Just pending work for a later time

Especially useful for interrupt handlers to delay work for "later".

The API:

```
<linux/interrupt.h>
struct tasklet_struct {
    ...
    (*func)(unsigned long);
    unsigned long data;
}
```

Static initialization:

- DECLARE_TASKLET(name, func, data);
- DECLARE_TASKLET_DISABLED(name, func, data);

Dynamic initialization:

void tasklet_init(struct tasklet_struct *t, void (*func) (unsigned long), unsigned long data);

Tasklet features:

- Can disable/enable, even multiple times
- Will only run if enabled as often as it was disabled
- Can specify "high" or "low" priority tasklets, the former being executed first.
- Will run either immediately, if no system load, or at the next system tick at the latest.
- Many tasklets can run in parallel on many CPUs
- Only one tasklet of a given type can run on any CPU at a given time.

Who runs tasklets?

- Per-cpu ksoftirqd

Full API:

void tasklet_disable(struct tasklet_struct *t);

- Disable tasklet execution

- Busy wait if tasklet currently running

```
void tasklet_disable_sync(struct tasklet_struct *t);
```

- Disable tasklet execution
- Don't wait for running tasklet to finish

```
void tasklet_enable(struct tasklet_struct *t);
```

- Enable tasklet execution

```
void tasklet_schedule(struct tasklet_struct *t);
```

- Schedule for execution
- If resched, run once
- If resched while running, rerun

```
void tasklet_hi_schedule(struct tasklet_struct *t);
```

- Schedule with high priority
- Avoid unless absolutely necessary (ex.: media streaming)

```
void tasklet_kill(struct tasklet_struct *t);
```

- Make sure tasklet doesn't run again
- Usually on device close
- Will block if tasklet scheduled
- If tasklet reschedules itself, must make sure it doesn't prior to using tasklet_kill()

Workqueues

Basics:

Not the same thing as previously seen wait queues

Similar to tasklets:

- Run something in the future
- Usually runs on same processor registered
- Cannot access userspace

Different from tasklets:

- Run in special process context, not software interrupt context.
- Can sleep
- Can ask for delayed execution for specific time
- No need for atomic execution

Can tolerate high latency

Each work queue has dedicated "kernel thread"

Basic API:

- struct workqueue_struct: <linux/workqueue.h>
- struct workqueue_struct *create_workqueue(const char *name);
 - One kernel thread per cpu
- struct workqueue_struct *create_singlethread_workqueue(const char *name);
 - One kernel thread for entire system

Submitting a task to a work queue:

Static declaration:

- DECLARE_WORK(name, void (*function)(void *), void *data);●

Dynamic declaration:

- INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
- PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
 - Similar to INIT_WORK() but doesn't initialize pointers to link struct work_struct to actual work queue.

- Useful if structure may have already been submitted to work queue.

Actual submission:

int queue_work(struct workqueue_struct *queue, struct work_struct *work);

- retval is zero if successful add
- retval nonzero if already in queue (not added again)

int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);

About work queue callback sleep:

- Will affect other callbacks queued in work queue.

Rest of API:

int cancel_delayed_work(struct work_struct *work);

- retval nonzero if cancel prior to execution
- retval zero if work may have been running (and could still be running on different CPU).

void **flush_workqueue**(struct workqueue_struct *queue);

- Make sure no scheduled work is running anywhere in the system.

void **destroy_workqueue**(struct workqueue_struct *queue);

- Free work queue.

The share queue:

- Not all drivers need their own work queue
- Kernel provides default shared queue
- Make sure your task doesn't monopolize queue

API:

- int schedule_work(struct work_struct *work);
- int schedule_delayed_work(struct work_struct *work, unsigned long delay);
- void flush_scheduled_work(void);

- Can still use cancel_delayed_work().