Embedded Systems Programming

Device drivers - interrupts and timers

Iwona Kochańska

Katedra Systemów Elektroniki Morskiej
WETI PG

April 8, 2019
Interrupts in Linux drivers

- In Linux, interrupt signals are the distraction which diverts processor to a new activity outside from normal flow of execution.
- This activity is called interrupt handler or interrupt service routine (ISR) and that distraction is Interrupts.
- Interrupts vs Polling
  - In polling the CPU keeps on checking all the hardwares of the availability of any request
  - In interrupt the CPU takes care of the hardware only when the hardware requests for some service
Interrupts in linux drivers

- An interrupt is produced by electronic signals from hardware devices and directed into input pins on an interrupt controller.
- Upon receiving an interrupt, the interrupt controller sends a signal to the processor.
- The processor detects this signal and interrupts its current execution to handle the interrupt.
- The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.

Source: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html
Interrupts in Linux drivers

- Interrupt handling is amongst the most sensitive tasks performed by kernel and it must satisfy following:
  - Hardware devices generate interrupts asynchronously (with respect to the processor clock). Interrupts can come anytime.
  - The kernel might be handling one of interrupts while another one (of a different type) occurs.
  - Some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible.
Interrupts and exceptions

- **Interrupts**: asynchronous interrupts generated by hardware.
- **Exceptions**: synchronous interrupts generated by the processor.
  - exceptions occur synchronously with respect to the processor clock; they are often called “synchronous interrupts”
  - exceptions are produced by the processor while executing instructions either in response to a programming error (e.g. divide by zero) or abnormal conditions that must be handled by the kernel (e.g. a page fault).
  - many processor architectures handle exceptions in a similar manner to interrupts, thus the kernel infrastructure for handling the two is similar.
- **System calls** (one type of exception) on the x86 architecture
  - software interrupt, which traps into the kernel and causes execution of a special system call handler.
  - interrupts work in a similar way, except hardware (not software) issues interrupts.
Maskable/nonmaskable interrupts

- **Maskable interrupts**
  - all Interrupt Requests (IRQs) issued by I/O devices
  - two possible states: masked or unmasked
  - a masked interrupt is ignored by the control unit as long as it remains masked.

- **Nonmaskable interrupts**
  - only a few critical events (such as hardware failures) give rise to nonmaskable interrupts
  - nonmaskable interrupts are always recognized by the CPU.
Exceptions

- **Fails** – like Divide by zero, Page Fault, Segmentation Fault.
- **Traps** – Reported immediately following the execution of the trapping instruction. Like Breakpoints.
- **Aborts** – Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables.
Interrupt handler

- For a device’s each interrupt, its device driver must register an interrupt handler

- **Interrupt handler** or **interrupt service routine (ISR)** is the function that the kernel runs in response to a specific interrupt:
  - each device that generates interrupts has an associated interrupt handler.
  - The interrupt handler for a device is part of the device’s driver (the kernel code that manages the device).

- In Linux, interrupt handlers are normal C functions, which match a specific prototype and thus enables the kernel to pass the handler information in a standard way.

- kernel invokes interrupt handlers in response to interrupts and that they run in a special **interrupt context** (atomic context).
Interrupt handler

- Interrupt can occur at any time -> interrupt handler can be executed at any time.
- It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible.
- It is important that:
  - to the hardware: the operating system services the interrupt without delay. To the rest of the system: the interrupt handler executes in as short a period as possible.
  - at the very least, an interrupt handler’s job is to acknowledge the interrupt’s receipt to the hardware. However, interrupt handlers can often have a large amount of work to perform.
Process Context vs Interrupt Context

- **Process context** - kernel code that services system calls issued by user applications
- **Interrupt context** - interrupt handlers (run asynchronously)
- **Processes contexts** are not tied to any **interrupt context** and vice versa.
- Kernel code running in **process context** is **preemptible**.
- **Interrupt context** always runs to completion and is **not** preemptible.
Process Context and Interrupt Context

- Code executing from **interrupt context** cannot:
  - go to sleep or relinquish the processor
  - acquire a mutex
  - perform time-consuming tasks
  - access user space virtual memory

- Based on our idea, ISR or Interrupt Handler should be execute very quickly and it should not perform time-consuming tasks

- While ISR run, it doesn’t let other interrupts to run (interrupts with higher priority will run). Interrupts with same type can be missed!

- To eliminate that problem, the processing of interrupts is split into two parts, or halves: **top halves** and **bottom halves**.
Processing of interrupts

▶ Top half
  ▶ interrupt handler
  ▶ top half will run immediately upon receipt of the interrupt and performs only the work that is time-critical (such as acknowledging receipt of the interrupt or resetting the hardware)

▶ Bottom half
  ▶ used to process data
  ▶ notifies the top half about new incoming interrupts
  ▶ 4 bottom half mechanisms available in Linux:
    ▶ Work-queue
    ▶ Threaded IRQs
    ▶ Softirqs
    ▶ Tasklets
1. Interrupt handlers can not enter sleep, so to avoid calls to some functions which has sleep.

2. Don’t use mutexes in interrupt handler. Use spinlocks lock.

3. Interrupt handlers can not exchange data with the user space.

4. The interrupt handlers must be executed as soon as possible. Split the implementation into two parts, top half and bottom half.

5. Interrupt handlers can not be called repeatedly. When a handler is already executing, its corresponding IRQ must be disabled until the handler is done.

6. Interrupt handlers can be interrupted by higher authority handlers.
Functions related to interrupt

request_irq() - register an IRQ

- prototype:
  `request_irq ( unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev_id)`

- parameters:
  - irq: IRQ number to allocate
  - handler: interrupt handler function. Returns data type `irq_handler_t`. Return value `IRQ_HANDLED` = the processing is completed successfully; return value `IRQ_NONE` = the processing fails.
  - flags: can be either zero or a bit mask of one or more of the flags defined in `linux/interrupt.h`.
  - name: identify the device name using this IRQ
  - dev_id: IRQ shared by many devices.

  When an interrupt handler is freed, dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line.
  Can be NULL if the line is not shared.
Functions related to interrupt

FLAGS:

- **IRQF_DISABLED**
  - when set, instructs the kernel to disable all interrupts when executing this interrupt handler
  - when unset, interrupt handlers run with all interrupts except their own enabled.
  - most interrupt handlers do not set this flag, as disabling all interrupts is bad form. Its use is reserved for performance-sensitive interrupts that execute quickly.

- **IRQF_SAMPLE_RANDOM**
  - specifies that interrupts generated by this device should contribute to the kernel entropy pool (kernel entropy pool provides truly random numbers derived from various random events)
  - if this flag is specified, the timing of interrupts from this device are fed to the pool as entropy
  - do not set this if your device issues interrupts at a predictable rate (e.g. the system timer) or can be influenced by external attackers (e.g. a networking device).
Functions related to interrupt

FLAGS:

▶ IRQF_TIMER - specifies that this handler processes interrupts for the system timer

▶ IRQF_SHARED
  ▶ specifies that the interrupt line can be shared among multiple interrupt handlers
  ▶ each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line.
free_irq() - release an IRQ registered by request_irq()

- prototype:
  ```c
default free_irq(unsigned int irq, void *dev_id);
```
- parameters:
  - irq: IRQ number
  - dev_id: the last parameter of request_irq.
- If the specified interrupt line is not shared, function removes the handler and disables the line.
- If the interrupt line is shared, the handler identified via dev_id is removed, but the interrupt line is disabled only when the last handler is removed.
- A call to free_irq() must be made from process context.
Functions related to interrupt

- **disable_irq(unsigned int irq)** - disable an IRQ from issuing an interrupt
- **disable_irq_nosync(unsigned int irq)** - disable an IRQ from issuing an interrupt, but wait until there is an interrupt handler being executed
- **enable_irq(unsigned int irq)** - re-enable interrupt disabled by **disable_irq** or **disable_irq_nosync**
- **in_irq()** - returns true when in interrupt handler
- **in_interrupt()** returns true when in interrupt handler or bottom half
Timers

Timer - a specialized type of clock used for measuring specific time intervals.

▶ Timers can be categorized into two main types:
  ▶ stopwatch - timer which counts upwards from zero for measuring elapsed time
  ▶ timer - a device which counts down from a specified time interval

▶ In Linux, kernel keeps track of the flow of time by means of timer interrupts
▶ Timer interrupts are generated at regular timer intervals by using system’s timing hardware
▶ Every time a timer interrupt occurs, the value of an internal kernel counter is incremented
▶ Counter is initialized to 0 at system boot, so it represents the number of clock ticks since last boot.
Timers

- Uses of kernel timers:
  - polling a device by checking its state at regular intervals when the hardware can’t fire interrupts
  - user wants to send some message to other device at regular intervals
  - send error when some action didn’t happened in particular time period
Kernel Timer API

- Include header: `<linux/timer.h>`
- Kernel timers are described by the `timer_list` structure, defined in `<linux/timer.h>`:

```c
struct timer_list {
    /* ... */
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
};
```

- `expires` - expiration time of the timer (in jiffies)
- `function()` will be called on expiration with the given data value
- `jiffy` - defined by a compile-time constant called HZ. Different platforms use different values for HZ.
  - HZ - number of times jiffies is incremented in one second
  - Historically, the kernel used 100 as the value for HZ, yielding a jiffy interval of 10 ms.
  - With 2.4, the HZ value for i386 was changed to 1000, yielding a jiffy interval of 1 ms.
  - Recently (2.6.13) the kernel changed HZ for i386 to 250.
Kernel Timer API

Initialize/Setup Kernel Timer:

- **init_timer**

  ```c
  voidfastcall init_timer ( struct timer_list * timer );
  ```

  - **init_timer** must be done to a timer prior calling any of the other timer functions
  - if it is used to initialize the timer, then the **callback function** and data of the **timer_list** structure have to be set manually.
  - argument **timer** - the timer to be initialized
Kernel Timer API

Initialize/Setup Kernel Timer:

- **setup_timer**

  ```c
  void setup_timer(timer, function, data);
  ```

  - set data and function of `timer_list` structure and **initialize** the timer.
  - recommended to use.
  - arguments:
    - **timer** – the timer to be initialized
    - **function** – Callback function to be called when timer expires
    - **data** – data has to be given to the callback function
Initialize/Setup Kernel Timer:

- `DEFINE_TIMER`

`DEFINE_TIMER(_name, _function, _expires, _data)`

- no need to create the `timer_list` structure. Kernel will create the structure in the name of `_name` and initialize it.

- Arguments:
  - `_name` – name of the timer_list structure to be created
  - `_function` – Callback function to be called when timer expires
  - `_expires` – the expiration time of the timer (in jiffies)
  - `_data` – data has to be given to the callback function
Kernel Timer API - start a timer

add_timer - starts a timer.

```c
void add_timer(struct timer_list *timer);
```

Argument: timer – the timer needs to be start
Modifying Kernel Timer’s timeout

**mod_timer** - modifies a timer’s timeout

```c
int mod_timer (struct timer_list * timer, unsigned long expires);
```

- **Arguments:**
  - `timer` – the timer needs to be modify the timer period
  - `expires` – the updated expiration time of the timer (in jiffies)

- Returns whether it has modified a pending timer or not (0 – mod_timer of an inactive timer, 1 – mod_timer of an active timer)

- **Equivalent to:**
  ```c
del_timer(timer);
timer->expires = expires;
add_timer(timer);
```
Kernel Timer API - stop a timer

- del_timer
  ```c
  int del_timer (struct timer_list * timer);
  ```
  - Deactivate a timer
  - Argument: timer – the timer needs to be deactivate
  - Returns whether it has deactivated a pending timer or not (0 – del_timer of an inactive timer, 1 – del_timer of an active timer)

- del_timer_sync
  ```c
  int del_timer_sync (struct timer_list * timer);
  ```
  - Deactivate a timer and wait for the handler to finish
  - Argument: timer – the timer needs to be deactivate
  - Returns whether it has deactivated a pending timer or not (0 – del_timer_sync of an inactive timer, 1 – del_timersync of an active timer)
Kernel Timer API - check kernel timer status

- `timer_pending` - tells whether a given timer is currently pending, or not.

  ```c
  int timer_pending(const struct timer_list * timer);
  ```

  - Argument: `timer` – the timer needs to check status
  - Returns whether timer is pending or not (0 – timer is not pending, 1 – timer is pending)
Bibliography

- https://lwn.net/Kernel/LDD3/
- http://www.kaltpost.de/?p=1699 (Linux Kernel Module Examples (on the Pi))