3. Interrupts

**General**

* The idea of an interrupt is that a device may say something to the operating system
  + No polling
* 2 types of interrupts
  + Software interrupt
  + Hardware interrupt
    - Physical IRQ line
      * Level active
      * Edge active
    - MSI (Message signaled interrupt)
    - Doorbell (similar to MSI)
* Very simple concept but it is very easy to generate bugs with it
* Interrupt masking
  + May not be supported for all interrupts (especially hardware failures (NMI))
  + I/O line masking
  + Device specific interrupt masking (draw)

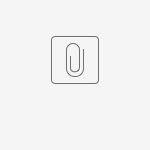
**Interrupts in Linux**

* Interrupts are not executed in process context
  + User space memory cannot be accessed
  + No sleeping
  + No scheduler access
  + All buses may not be accessible from IRQ context
* /proc/interrupts
* Can be enabled and disabled by the kernel
* Top/Bottom half
  + Top half are run in HW interrupt context
    - Need to be as fast as possible
    - The interrupt is disabled while this is running
      * We lose all interrupts happening during the handling (check and clarify this (edge))
      * Fast EOI???
  + Bottom is run as a separate thread/tasklet/work
    - Everything that takes significant amount of time is done here
* Typical interrupt handling example + drawing
* Locking and interrupts (check the ldd book)
  + spin\_lock
  + spin\_lock\_irqsave
    - Needed if the lock is also used in process context
    - Protects from situation where interrupt may fire when the lock is acquired
  + RT kernel specifics (optional - Timo)
* Typical race condition example (edge handler interrupt)
* Interrupt flags
  + IRQ\_F\_ONESHOT
    - The IRQ is disabled when thread is run
  + IRQ\_F\_SHARED
    - Allows multiple handlers for one IRQ line
      * The handler decides if the particular IRQ was interesting
      * When some handler returns IRQ\_HANDLED rest of the handlers are not invoked (check)
* Safety mechanisms (is there more?)
  + Interrupt storm
    - limits the amount of unhandled interrupts (return value IRQ\_NONE)
  + Limits + example

**Interrupt APIs**

* request\_irq
* request\_threaded\_irq
  + Needed when the device cannot be accessed from IRQ context
* free\_irq
* enable\_irq
* disable\_irq
  + blocks until all handlers are done
* disable\_irq\_nosync
  + returns immediately
* Interrupt handler prototype here
  + irq\_return\_t

**Exercises**

[[](https://confluence.int.net.nokia.com/download/attachments/438991887/rtc-cmos.patch?version=2&modificationDate=1486981121000&api=v2)rtc-cmos.patch](https://confluence.int.net.nokia.com/download/attachments/438991887/rtc-cmos.patch?version=2&modificationDate=1486981121000&api=v2)

Modify kernel rtc-cmos.c (patch)

* make menuconfig
* Enable CONFIG\_IRQSOFF\_TRACER
* make
* make modules
* sudo make modules\_install
* sudo make install
* sudo grub2-mkconfig -o /boot/grub2/grub.cfg

**Training 3, lab1**

1. Take the test module from training 1 lab1 as a base.
2. write a irq handler for our interrupt. The handler can just issue a print.
3. Compile and load the module
4. To trigger an interrupt run: sudo hwclock -r
5. See dmesg for results

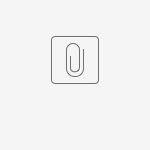
Prototype for handler can be found by investigating the request\_irq() function which you can use from init in order to register the interrupt handler. IRQ number should be the number your kernel is printing from modified rtc-cmos.c Note that we are now using same interrupt as rtc-cmos so we need to state this in the flags when we request the interrupt (IRQF\_SHARED). Remember also to return the IRQ\_HANDLED as return value from handler (why?)

**Training 3, lab2.**

Try doing bad things from interrupt handler.

1. Try calling msleep()
   1. Test similarly as lab1
   2. what do you observe? (Why?)
2. Try disabling irq (using disable\_irq()) from handler. (issue print before and after disabling)
   1. Test similarly as lab1
   2. what do you observe? (Why?)

**Training 3, lab3.**

[[](https://confluence.int.net.nokia.com/download/attachments/438991887/irq_test.c?version=1&modificationDate=1486982238000&api=v2)irq\_test.c](https://confluence.int.net.nokia.com/download/attachments/438991887/irq_test.c?version=1&modificationDate=1486982238000&api=v2)

1. Copy the above module skeleton for locking exercise.
   * This module creates a kernel thread which constantly writes data to an array. Thread fills the whole array first with the character '0'. Then with '1', ...
   * Module also installs the handler for the rtc interrupt. When interrupt occurs the handler copies array and checks if array contains only one type of character. For example array full of '0's or '1's is Ok, but array with '0's and '1's is not. The purpose of this exercise is to try out locking mechanisms which are usable with interrupts.
2. Try using both the spin lock and mutex.
   * Try mutex\_lock/mutex\_unlock. Test.
   * spin\_lock/spin\_unlock.Try running hwclock -r multiple times in a row. What do you observe and why?
   * spin\_lock\_irqsave/spin\_unlock\_irqrestore. Test.
3. Bonus question, why do we have call to schedule() in the thread?

**Training 3, lab4.**

Let's try simple irq disabling as a protection mechanism. Do following modifications to clean version of the irq\_test.c:

1. Add mdelay(10) in irq handler before the call to memcpy.
2. Disable the interrupt for the duration of array filling. Remember to call enable\_irq() to re-enable the interrupt.
   1. Use disable\_irq\_nosync()
   2. disable\_irq().
3. Try out the hwclock -r multiple times in a row. What do you observe and why?

**Training 3, lab5.**

Typically we try to minimize time spent in hw irq context(top half) (why?). Hence irq handlers are traditionally divided to "top half" and "bottom half". Top half is executed in the HW irq context and it should only do minimal amount of work. In many cases top half just check that interrupt is really for the module and everything else is done at bottom half. In cases where timing is important the top half may do actual work.

Let's now divide the irq\_test.c handler in top and bottom half so that top half only copies the data from array to some other location (If you use threaded handler you can for example modify and use the g\_irqdata as storage) and bottom half performs the array "integrity checking".

For division you can use:

* threaded interrupt handler (request\_threaded\_irq()). Primary handler is now your top half and thread function is the bottom half.
* work\_queue (DECLARE\_WORK() or \_\_WORK\_INITIALIZER(), schedule\_work). \_\_WORK\_INITIALIZER() allows you to encapsulate the struct work\_struct inside another struct (which can have a data storage for array copy as other member) and use container\_of() to access the encapsulating struct.
* timer: init\_timer, add\_timer(), set data,function,expires
* tasklet: DECLARE\_TASKLET(), tasklet\_schedule()

Optional: You may store jiffies in the data struct to see how much time was spent between reception of the interrupt and the start of the bottom half handler.