<Slides download>
http://www.pf.is.s.u-tokyo.ac.jp/class.html

Advanced Operating Systems

#13

Shinpei Kato
Associate Professor

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Course Plan

- Multi-core Resource Management
- Many-core Resource Management
- GPU Resource Management
- Virtual Machines
- Distributed File Systems
- High-performance Networking
- Memory Management
- Network on a Chip
- Embedded Real-time OS
- Device Drivers
- Linux Kernel

Schedule

1. 2017.9.27 (No Class)
2. 2017.10.4 Introduction
3. 2017.10.11 (No Class)
4. 2017.10.18 Multi-core & Many-core Resource Management
5. 2017.10.25 GPU Resource Management
6. 2017.11.1 (No Class)
7. 2017.11.8 Memory Management
8. 2017.11.15 Device Drivers
9. 2017.11.22 Linux Kernel
10. 2017.11.29 Linux Kernel
11. 2017.12.6 Linux Kernel
12. 2017.12.13 Virtual Machines & Distributed File Systems
13. 2017.12.20 Network on a Chip
14. 2017.1.10 High-performance Networking
15. 2017.1.17 Embedded Real-time OS
16. 2017.1.24 (Probably No Class)

Device Drivers

Abstracting Devices – Modules and Interrupts

/* The case for Linux */

Acknowledgement:

Prof. Pierre Olivier, ECE 4984, Linux Kernel Programming, Virginia Tech

Outline1

Outline1

Kernel modules: presentation

General information

- **Modules** are pieces of kernel code that can be **dynamically loaded and unloaded at runtime**
  - No need to reboot
- Appeared in Linux 1.2 (1995)
- Numerous Linux features can be compiled as modules
  - Selection in the configuration `.config` file
    - Ex: device/filesystem drivers
    - Generated through the `menuconfig` make target
  - Opposed to **built-in** in the kernel binary executable `vmlinux`

Kernel modules: presentation

Benefits of kernel modules

- Modules benefits:
  - **No reboot**
    - Saves a lot of time when developing/debugging
  - **No need to compile the entire kernel**
  - **Saves memory and CPU time** by running on-demand
  - No performance difference between module and built-in kernel code
  - Help **identifying buggy code**
    - Ex: identifying a buggy driver compiled as a module by selectively running them

Outline

Writing a kernel module

Basic C file for a module

```c
#include <linux/module.h> /* Neededbyallmodules     */
#include <linux/kernel.h> /* KERN_INFO */
#include <linux/init.h>    /* Initandexitmacros     */

staticint answer initdata = 42;

staticint init lkp_init(void)
{
  printk(KERN_INFO"Moduleloaded...¥n");
    printk(KERN_INFO"Theansweris%d...¥n", answer);

    /* Return0onsuccess,somethingelseonerror          */
    return 0;
}

staticvoid exit lkp_exit(void)
{
  printk(KERN_INFO"Moduleexiting...¥n");
}

module_init(lkp_init);
module_exit(lkp_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("PierreOlivier<polivier@vt.edu");
MODULE_DESCRIPTION("Samplekernelmodule");
```

- Create a C file anywhere on the filesystem
  - No need to be inside the kernel sources
- Init. & exit functions
  - Launched at load/unload time
- `MODULE_*` macros
  - General info about the module

Writing a kernel module

Kernel namespace

- Module is linked against the entire kernel:
  - Module has visibility on all of the kernel global variables
  - To avoid namespace pollution and involuntary reuse of variables names:
    - Use a well defined naming convention. Ex:
      `my_module_function_a()`
      `my_module_function_b()`
      `my_module_global_variable`
    - Use `static` as much as possible
- Kernel symbols list is generally present in:
  `/proc/kallsyms`

Outline1

Compiling a kernel module

Kernel sources & module Makefile

- Need to have the kernel sources somewhere on the filesystem
- Create a `Makefile` in the same directory as the module source file

```
#let'sassumethemoduleCfileisnamedlkp.c
obj-m := lkp.o
KDIR := /path/to/kernel/sources/root/directory
#Alternative:Debian/Ubuntuwithkernel-headerspackage:
#KDIR:=/lib/modules/$(shelluname-r)/build
PWD := $(shell pwd)

all: lkp.c
  make -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
  make -C $(KDIR) SUBDIRS=$(PWD) clean
```

- Multiple source files?

```
obj-m += file1.c
obj-m += file2.c
#etc.
```

- After compilation, the compiled module is the file with `.ko` extension

Outline1

Launching a kernel module

`insmod`/`rmmod`

- Needs administrator privileges (root)
  - You are executing kernel code!
- Using `insmod`:

```
sudo insmod file.ko
```

  - Module is loaded and init function is executed
- Note that **a module is compiled against a specific kernel version and will not load on another kernel**
  - This check can be bypassed through a mechanism called `modversions` but it can be dangerous
- Remove the module with `rmmod`:

```
sudo rmmod file
#or:
sudo rmmod file.ko
```

  - Module exit function is called

Launching a kernel module

`modprobe`

- `make modules_install` from the kernel sources installs the modules in a standard location on the filesystem
  - Generally `/lib/modules/<kernel version>/`
- These modules can be inserted through `modprobe`:

```
sudo modprobe <module name>
```

  - No need to point to a file, just give the module name
- Contrary to `insmod`, `modprobe` handles modules dependencies
  - Dependency list generated in `/lib/modules/<kernel version/modules.dep`
- Remove using `modprobe -r <module name>`
- Such installed modules can be loaded automatically at boot time by editing `/etc/modules` or the files in `/etc/modprobe.d`

Outline1

Modules: miscellaneous information

Modules parameters

- **Parameters** can be entered from the command line at launch time

```
#include <linux/module.h>
/* ... */

staticint int_param = 42;
staticchar *string_param ="defaultvalue";

module_param(int_param, int, 0);
MODULE_PARM_DESC(int_param,"Asampleintegerkernelmoduleparameter");
module_param(string_param, charp, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(string_param,"Anotherparameter,astring");

staticint_init lkp_init(void)
{
    printk(KERN_INFO"Intparam:%d¥n", int_param);
    printk(KERN_INFO"Stringparam:%s¥n", string_param);

    /* ... */
}

/* ... */
```

```
sudo insmod lkp.ko int_param=12 string_param="hello"
```

Modules: miscellaneous information

`modinfo`, `lsmod`

- `modinfo`: info about a kernel module
  - Description, kernel version, parameters, author, etc.

```
modinfo my_module.ko
filename:       /tmp/test/my_module.ko
description:    Sample kernel module
author:         Pierre Olivier <polivier@vt.edu>
license:        GPL
srcversion:     A5ADE92B1C81DCC4F774A37
depends:
vermagic:       4.8.0-34-generic SMP mod_unload modversions
parm:           int_param:A sample integer kernel module parameter (int)
parm:           string_param:Another parameter, a string (charp)
```

- `lsmod`: list currently running modules
  - Can also look in `/proc/modules`

Modules: miscellaneous information

Additional sources of information on kernel modules

- The linux kernel module programming guide:
  - `http://www.tldp.org/LDP/lkmpg/2.6/html/index.html`
- Linux loadable kernel module howto
  - `http://www.tldp.org/HOWTO/Module-HOWTO/index.html`
- Linux sources → `Documentation/kbuild/modules.txt`

Outline1

Memory allocation

`kmalloc`

- Allocate memory that is virtually and **physically contiguous**
  - For DMA, memory-mapped I/O, and performance (large pages)
- Because of that property, maximum allocated size through one `kmalloc` invocation is limited
  - 4MB on x86 (architecture dependent)

```c
#include <linux/slab.h>
/* ... */
char *my_string = (char *)kmalloc(128, GFP_KERNEL);
my_struct my_struct_ptr = (my_struct *)kmalloc(sizeof(my_struct), GFP_KERNEL);
/* ... */
kfree(my_string);
kfree(my_struct_ptr);
```

- Returns a pointer to the allocated memory or `NULL` in case of failure
- Mostly used **flags**:
  - `GFP_KERNEL`: ***might sleep***
  - `GFP_ATOMIC`: do not block, but higher chance of failure

Memory allocation

`vmalloc`

- Allocate memory that is **virtually contiguous, but not physically contiguous**
- No size limit other than the amount of free RAM (at least on 64 bit architectures
- ***Might sleep***

```
#include <linux/vmalloc.h>
/* ... */
char *my_string = (char *)vmalloc(128);
my_struct my_struct_ptr = (my_struct *)vmalloc(sizeof(my_struct));
/* ... */
vfree(my_string);
vfree(my_struct_ptr);
```

- Returns a pointer to the allocated memory or `NULL` in case of failure

Outline2

Outline2

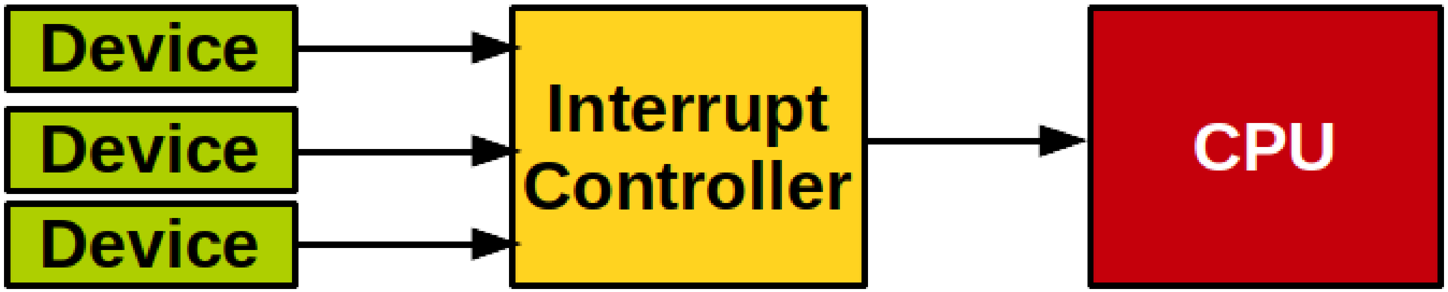
Interrupts: general information

Interrupts

- Compared the the CPU, devices are **slow**
  - Ex: when a read request is issued to the disk, it is sub-optimal to wait, doing nothing until the data is ready (in RAM)
    - Need to know when the hardware is ready
- *Polling* can create a lot of overhead
  - Having the CPU check regularly the status of the hardware
- **The solution is to have** *hardware devices signal the CPU* **that they need attention**
  - **Interrupts**
    - A key has been pressed on the keyboard
    - A packet has been received on the network card
    - etc.

Interrupts: general information

Interrupts (2)



- Interrupts are electrical signals multiplexed by the interrupt controller
  - Sent on a specific pin of the CPU
- Once an interrupt is received, a dedicated function is executed:
  - *Interrupt handler*
- They can be received in a completely non-deterministic way:
  - **The kernel/user space can be interrupted at (nearly) any time to process an interrupt**

Interrupts: general information

Interrupts (3)

- Device identifier: **interrupt line** or **Interrupt ReQuest (IRQ)**
- Function executed by the CPU: **interrupt handler** or **Interrupt Service Routine (ISR)**

- 8259A interrupt lines:
  - IRQ #0: system timer
  - IRQ #1: keyboard controller
  - IRQ #3 and #4: serial port
  - IRQ #5: terminal
  - IRQ #6: floppy controller
  - IRQ #8: RTC
  - IRQ #12: mouse
  - IRQ #14: ATA (disk)
- Source [2]

- *Some interrupt lines can be shared among several devices*
  - True for most modern devices (PCIe)

Interrupts: general information

Exceptions

- **Exception** are interrupt issued by the CPU executing some code
  - *Software* interrupts, as opposed to *hardware* ones (devices)
  - Happen synchronously with respect to the CPU clock
  - Examples:
    - **Program faults**: divide-by-zero, page fault, general protection fault, etc.
    - **Voluntary exceptions**: INT assembly instruction, for example for syscall invocation
    - List: [1]
- Exceptions are managed by the kernel the same way as hardware interrupts

Interrupts: general information

Interrupt handlers

- The **interrupt handlers** (ISR) are kernel C functions associated to interrupt lines
  - Specific prototype
  - Run in **interrupt context**
    - Opposite to process context (system call)
    - Also called atomic context as *one cannot sleep in an ISR*: it is not a schedulable entity
- Managing an interrupt involves two high-level steps:
  1. **Acknowledging the reception** (mandatory, fast)
  2. Potentially **performing additional work** (possibly slow)
     - Ex: processing a network packet available from the Network Interface Card (NIC)

Interrupts: general information

Top-halves vs bottom-halves

- *Interrupt processing must be fast*
  - We are indeed interrupting user processes executing (user/kernel space)
  - In addition, other interrupts may need to be disabled during an interrupt processing
- *However, it sometimes involves performing significant amount of work*
- **Conflicting goals**
  - Thus, processing an interrupt is broken down between:
    1. **Top-half**: time-critical operations (ex: ack), run immediately upon reception
    2. **Bottom-half**: less critical/time-consuming work, run later with other interrupts enabled

Interrupts: general information

Top-half & bottom-half example

- drivers/input/keyboard/omap-keypad.c

```c
/* (block 1) */
static int omap_kp_probe(struct
      platform_device *pdev)
{
  /* ... */
  omap_kp->irq = platform_get_irq(pdev, 0);
  if(omap_kp->irq >= 0) {
    if(request_irq(omap_kp->irq,
      omap_kp_interrupt, 0,
        "omap-keypad", omap_kp) < 0)
      goto err4;
  }
}
```

```c
/* (block 3) */
static DECLARE_TASKLET_DISABLED(
  kp_tasklet, omap_kp_tasklet, 0);
```

```c
/* (block 2) */
/* Top half: interrupt handler (ISR) */
static irqreturn_t omap_kp_interrupt(int
      irq, void *dev_id)
{
  /* disable keyboard interrupt */
  omap_writew(1, /* ... */);

  tasklet_schedule(&kp_tasklet);
  return IRQ_HANDLED;
}
```

```c
/* (block 4) */
/* Bottom half */
static void omap_kp_tasklet(unsigned long
      data)
{
  /* performs lot of work */
}
```

Outline2

1 Interrupts: general information

2 Registering & writing an interrupt handler

3 Interrupt context

4 Interrupt handling internals in Linux

5 /proc/interrupts

6 Interrupt control

Registering & writing an interrupt handler

Interrupt handler registration: request_irq()

- **request_irq()** (in includes/linux/interrupt.h)

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
    const char *name, void *dev)
```

- irq: interrupt number
- handler: function pointer to the actual handler
  - prototype:

```
typedef irqreturn_t (*func)(int irq, void *data);
```

- name: String describing the associated device
  - For example used in /proc/interrupts
- dev: unique value identifying a device among a set of devices sharing an interrupt line

Registering & writing an interrupt handler

Interrupt handler registration: registration flags

Registration flags:

- IRQF_DISABLED: **disables all interrupts when processing this handler**
  - Bad form, reserved for performance sensitive devices
  - Generally handlers run with all interrupts enabled except their own
  - **Removed in 4.1**
- IRQF_SAMPLE_RANDOM: this interrupt frequency will **contribute to the kernel entropy pool**
  - For Random Number Generation
  - **Do not set this on periodic interrupts!** (ex: timer)
    - RNG is used for example for cryptographic key generation
- IRQF_TIMER: system **timer**
- IRQF_SHARED: interrupt line can be **shared**
  - Each of the handlers sharing the line must set this

Registering & writing an interrupt handler

Interrupt handler registration: `irq_request()` (2)

- `irq_request()` returns 0 on success, or standard error code
  - -EBUSY: interrupt line already in use
- `irq_request()` **can sleep**
  - Creating an entry in the `/proc` virtual filesystem
    - `kmalloc()` in the call stack

Registering & writing an interrupt handler

An interrupt example, freeing an interrupt handler

- `omap-keypad` registration and handler:

```c
static int omap_kp_probe(struct
      platform_device *pdev)
{
  /* ... */
  if(request_irq(omap_kp->irq,
      omap_kp_interrupt, 0, "omap-keypad",
       omap_kp) < 0)
    goto err4;
}
```

```c
static irqreturn_t omap_kp_interrupt(int
      irq, void *dev_id)
{
  omap_writew(1, OMAP1_MPUIO_BASE +
      OMAP_MPUIO_KBD_MASKIT);
  tasklet_schedule(&kp_tasklet);
  return IRQ_HANDLED;
}
```

- Freeing an irq is made through `free_irq()`:

```c
void free_irq(unsigned int irq, void
      *dev);
```

- `omap-keypad` example:

```c
static int omap_kp_remove(struct
      platform_device *pdev)
{
  /* ... */
  free_irq(omap_kp->irq, omap_kp);
  /* ... */
  return 0;
}
```

Registering & writing an interrupt handler

Inside the interrupt handler

- **Interrupt handler prototype:**

```
static irqreturn_t handler_name(int irq, void *dev);
```

- **`dev` parameter:**
  - Must be unique between handlers sharing an interrupt line
  - Set when registering the handler and can be accessed by the handler
    - ex: pass a pointer to a data structure representing the device
- **Return value:**
  - `IRQ_NONE`: the expected device was not the source of the interrupt
  - `IRQ_HANDLED`: correct invocation
  - This macro can be used: `IRQ_RETVAL(x)`
    - If `(x != 0)`, expands into `IRQ_HANDLED`, otherwise expands into `IRQ_NONE` (example: `vsc_sata_interrupt` in `drivers/ata/sata_vsc.c`)
- Interrupt handlers do not need to be **reentrant** (thread-safe)
  - The corresponding interrupt is disabled on all cores while its handler is executing

Registering & writing an interrupt handler

Shared handlers

- **Shared handlers**
  - On registration:
    - IRQ_SHARED flag
    - `dev` must be unique (ex: a pointer to a data structure representing the device in question)
  - **Handler must be able to detect that the device actually generated the interrupt it is called from**
    - When an interrupt occurs on a shared line, *the kernel executes sequentially all the handlers sharing this line*
    - Need hardware support at the device level and detection code in the handler

Outline2

Interrupt context

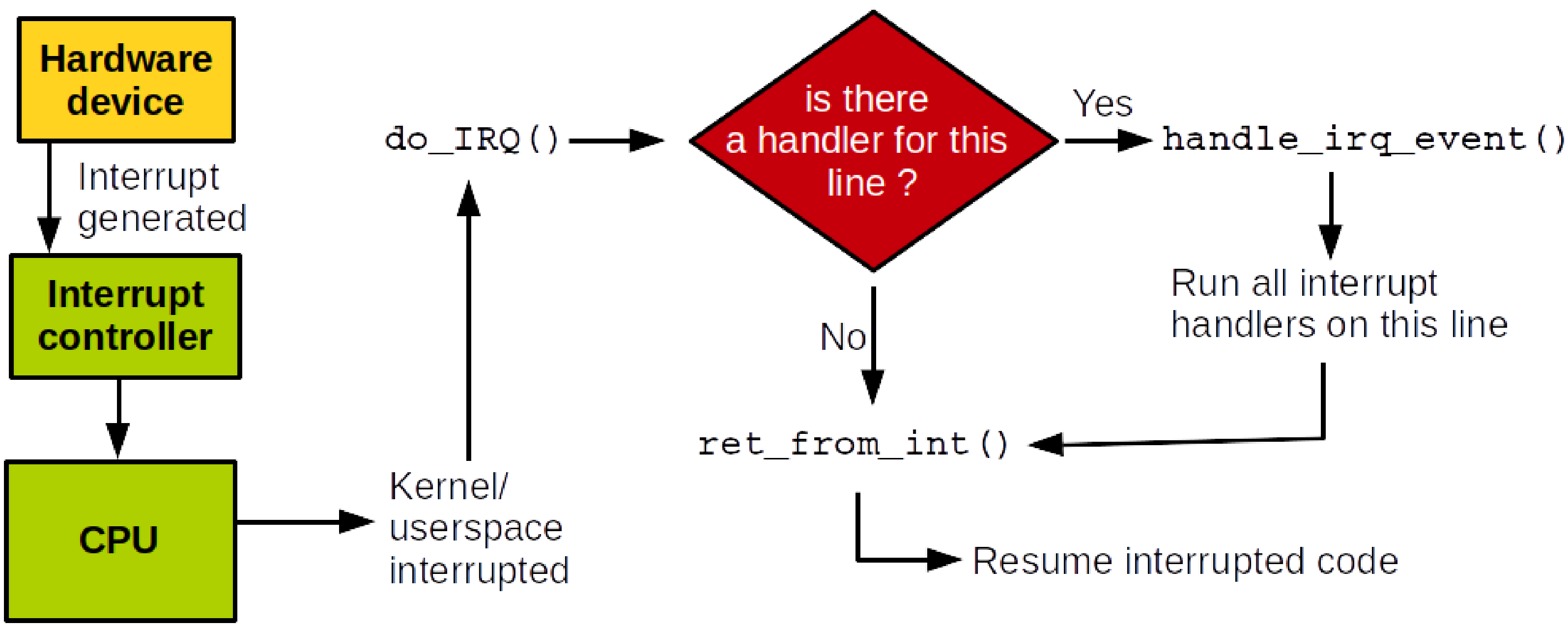
- The kernel can execute in **Interrupt vs process context**
  - In process context following a syscall/an exception
  - In interrupt context upon a hardware interrupt reception
- In interrupt context, **sleeping/blocking is not possible**
  - The handler is not a schedulable entity (user/kernel thread)
  - No kmalloc(x, GFP_KERNEL)
    - Use GFP_ATOMIC
  - No use of blocking synchronization primitives (ex: mutex)
    - Use spinlocks
- Interrupt context is **time-critical**
  - Other code is interrupted
- Interrupt handler stack:
  - Before 2.6: handlers used the kernel stack of the interrupted process
  - Later: 1 dedicated stack per core for handlers (1 page)

Outline2

1. Interrupts: general information
2. Registering & writing an interrupt handler
3. Interrupt context
4. Interrupt handling internals in Linux
5. /proc/interrupts
6. Interrupt control

Interrupt handling internals in Linux

Interrupt processing path



- Taken from the textbook

Interrupt handling internals in Linux

Interrupt processing path (2)

- Specific entry point for each interrupt line
  - Saves the interrupt number and the current registers
  - Calls do_IRQ()
- do_IRQ():
  - Acknowledge interrupt reception and disable the line
  - calls architecture specific functions
- Call chain ends up by calling __handle_irq_event_percpu()
  - Re-enable interrupts on the line if IRQF_DISABLED was not specified during handler registration
  - Call the handler if the line is not shared
  - Otherwise iterate over all the handlers registered on that line
  - Disable interrupts on the line again if they were previously enabled
- do_IRQ() returns to entry point that call ret_from_intr()
  - Checks if reschedule is needed (need_resched)
  - Restore register values

Outline2

1. Interrupts: general information
2. Registering & writing an interrupt handler
3. Interrupt context
4. Interrupt handling internals in Linux
5. /proc/interrupts
6. Interrupt control

/proc/interrupts

```
cat /proc/interrupts
           CPU0    CPU1        ...
  0:         19       0  ...        IR-IO-APIC   2-edge       timer
  1:          5       3  ...        IR-IO-APIC   1-edge       i8042
  8:          1       0  ...        IR-IO-APIC   8-edge       rtc0
  9:        272   13275  ...        IR-IO-APIC   9-fasteoi    acpi
 12:        387      11  ...        IR-IO-APIC  12-edge       i8042
 16:         24       2  ...        IR-IO-APIC  16-fasteoi    ehci_hcd:usb1
 23:         25       2  ...        IR-IO-APIC  23-fasteoi    ehci_hcd:usb2
```

- Columns:
  1. **Interrupt line** (not showed if no handler installed)
  2. **Per-cpu occurrence count**
  3. **Related interrupt controller name**
  4. **Edge/level (fasteoi)**: way the interrupt is triggered
  5. **Associated device name**

Outline2

Interrupt control

- Kernel code sometimes needs to **disable interrupts to ensure atomic execution** of a section of code
  - I.e. we don't want some code section to be interrupted by a handler (as well as kernel preemption)
  - The kernel provides an API to disable/enable interrupts:
    - Disable interrupts for the current CPU
    - Mask an interrupt line for the entire machine
- Note that *disabling interrupts does not protect against concurrent access from other cores*
  - Need locking, often used in conjunction with interrupts disabling

Interrupt control

Disabling interrupts on the local core

```
local_irq_disable();
/* ... */
local_irq_enable();
```

- `local_irq_disable()` **should never be called twice** without a `local_irq_enable()` between them
  - What if that code can be called from two locations:
    1. One with interrupts disabled
    2. One with interrupts enabled
- Need to save the interrupts state in order not to disable them twice:

```
unsigned long flags;

local_irq_save(flags);     /* disables interrupts _if needed_ */
/* .. */
local_irq_restore(flags); /* restores interrupts to the previous state */
/* flags is passed as value but both functions are actually macros */
```

Interrupt control

Disabling / enabling a specific interrupt line

- **Disable / enable a specific interrupt for the entire system**

```
void  disable_irq(unsigned int irq);          /* (1) */
void disable_irq_nosync(unsigned int irq);    /* (2) */
void  enable_irq(unsigned int irq);           /* (3) */
void  synchronize_irq(unsigned  int irq);     /* (4) */
```

1. Does not return until any currently running handler finishes
2. Do not wait for handler termination
3. Enables interrupt line
4. Wait for a specific line handler to terminate before returning

- These enable/disable calls can nest
  - Must enable as much times as the previous disabling call number
- These functions do not sleep
  - They can be called from interrupt context

Interrupt control

Querying the status of the interrupt system

- **in_interrupt()** returns nonzero if the calling code is in interrupt context
  - Handler or bottom-half
- **in_irq()** returns nonzero only if in a handler
- To check if the code is in **process context:**
  - !in_interrupt()

Additional information

- Interrupts:
  - http://www.mathcs.emory.edu/~jallen/Courses/355/Syllabus/6-io/0-External/interupt.html
- More details on Linux interrupt management (v3.18):
  - https://0xax.gitbooks.io/linux-insides/content/interrupts/

Bibliography I

1 Exceptions - osdev wiki
http://wiki.osdev.org/Exceptions
Accessed: 2017-02-08.

2 X86 assembly/programmable interrupt controller.
https://en.wikibooks.org/wiki/X86_Assembly/Programmable_Interrupt_Controller
Accessed: 2017-02-08.