

# Modules



**Mail: [Stephane.Lavirotte@unice.fr](mailto:Stephane.Lavirotte@unice.fr)**  
**Web: <http://stephane.lavirotte.com/>**  
**University of Nice - Sophia Antipolis**



# Introduction

For a good start

# Where does the Modules come from ?

- ✓ Principle based on micro kernel (ex: Mach) :
  - The (micro) kernel only contains generic code
    - Synchronization
    - Scheduling
    - Inter-Process Communications (IPC) ...
  - Each layer/function/driver of the OS is written independently
    - Obligation to define and use interfaces / protocols clearly defined
    - Strong partitioning (MACH module = process)
    - Better management of resources
      - Only active modules consume resources

# Linux Modules

## ✓ Module = object file

- Very partial implementation of the concept of micro-kernel
- Code can be dynamically linked at runtime
  - During boot (rc scripts)
  - At the request (kernel configured with optional CONFIG\_KMOD)
- But the code can also generally be statically linked in kernel code (traditional monolithic approach)

## ✓ Benefits

- Adding features to the kernel (drivers, support FS ...)
- Develop drivers without rebooting: load, test, unload, recompilation, loading ...
- Support incompatibility between drivers
- Useful to keep the kernel image to a minimum size
- Once loaded, any access to the kernel. No special protection.

# Dynamic Linking

## 1/3

- ✓ The code of a library is not copied into the executable at compile
  - It remains in a separate file (file .ko in Linux 2.6)
- ✓ The linker does virtually nothing to compile
  - It noted that libraries need an executable (see [Dependencies Modules](#))
- ✓ Most work is done during loading or execution
  - By the OS loader
- ✓ The loader looks for/loads libraries that a program needs
  - Adds necessary elements for the address space of process

# Dynamic Linking

## 2/3

- ✓ **Using dynamic linking involves relocations**
  - The jumps' addresses are not known at compile time
  - They are only known when the program and libraries are loaded
  - Unable to pre allocate space
    - Conflicts
    - Limitations of space in 32 bit memory
- ✓ **Using a table of indirection**
  - An empty table is added to the program at compile
  - All references go through this table (import directory)
  - Libraries have a table similar to the symbols they export (entry points)
  - This table is filled by loading the loader
- ✓ **Slower than static linking**

# Dynamic Linking

## 3/3

- ✓ **How to find the library at runtime ?**
  
- ✓ **Unix**
  - **Known/defined directories**
  - **Specified in** `/etc/ld.so.conf`
  - **Environment variables** (`LD_PRELOAD`, `LD_LIBRARY_PATH`)
  
- ✓ **Windows**
  - **Using the registry for ActiveX**
  - **Well-known directories** (`System32`, `System...`)
  - **Adding directories** `SetDllDirectory ( )`
  - **Current path and** `PATH`

# Constraints on Linux License

- ✓ **No constraints on redistribution**
  - You can share your changes early, in your own interest, but this is not forced !
- ✓ **Constraints upon distribution**
  - For embedded devices Linux and Free Software, you must cast your sources to the end user. You have no obligation to distribute them to anyone else !
  - The proprietary modules are tolerated (but not recommended) as they are not considered derivatives of GPL code
  - The proprietary drivers can not be linked statically into the kernel
  - No worries for the drivers available under a license compatible with the GPL ([details](#) in the section on writing modules)

# Necessary Modules for Booting

- ✓ **Compiling a kernel**
  - If "everything" is compiled as a module how to start the system?
  - Example:
    - Partition containing the root filesystem as ext3
    - and ext3 is not compiled statically into the kernel
    - How to access the root filesystem ?
  
- ✓ **The solution: initrd**

# initrd: Initial Ram Disk

- ✓ **Initrd (Initial RAM disk)**
  - Root file system (/) with minimalist use of RAM
  - Traditionally used to minimize the numbers of drivers are compiled statically into the kernel.
  - Useful for launching complex initialization scripts
  - Useful for loading proprietary modules (which can be linked statically to the kernel)
  
- ✓ **For more information:**
  - **Read** `Documentation/initrd.txt` **in the kernel sources !**
  - **Also covers changing root file system (`pivot_root`)**

# Manually Create an initrd Image

```
mkdir /mnt/initrd
```

```
dd if=/dev/zero of=initrd.img bs=1k count=2048
```

```
mkfs.ext2 -F initrd.img
```

```
mount -o loop initrd.img /mnt/initrd
```

*<Remplir avec: les modules, le script linuxrc...>*

```
umount /mnt/initrd
```

```
gzip --best -c initrd.img > initrd
```



# Modules

*Adding functionalities to Kernel*

# Dynamic Modules – Principles

- ✓ The kernel can handle a large number of components
  - They may not be active at the same time
- ✓ Modules to load the kernel components when they are needed (and if necessary)
  
- ✓ Benefits:
  - Less memory consumed by the kernel, so more memory for users
  - Avoiding a full kernel compilation when adding code for a device driver
  - Facilitates the debugging of a driver: it allows to unload and reload the module (if it has not crashed the machine ;-)

# « Hello World » Module

## ✓ « Hello World » Module : hello.c

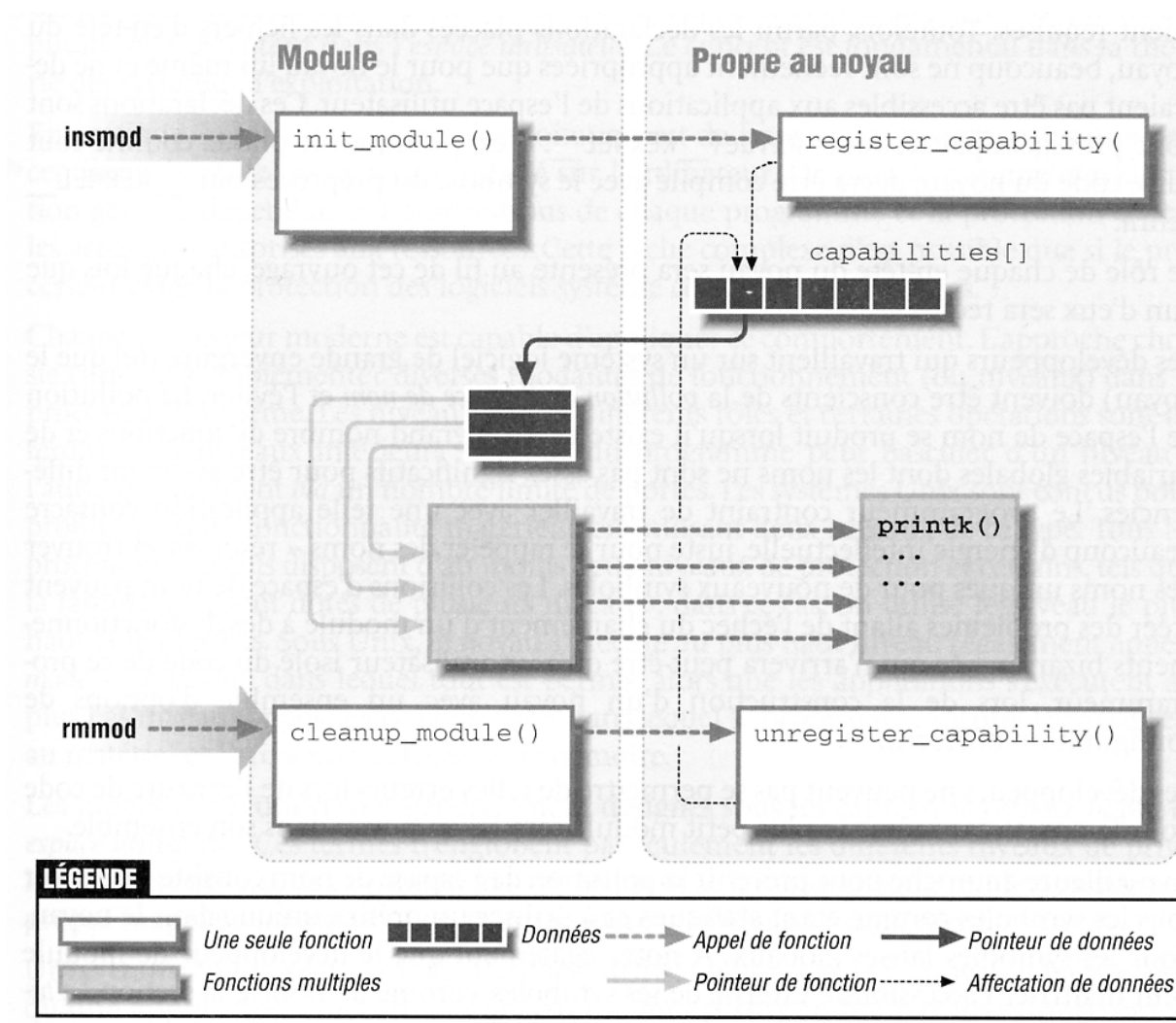
```
#include <linux/module.h> /* Nécessaire pour tout module */

int init_module(void)
{
    printk("Hello World!\n");
    return 0;
}

void cleanup_module(void)
{
    printk("Goodbye, cruel world!\n");
}

MODULE_LICENSE("GPL");
```

# Dynamic Modules – Schema



# Module Initialization

## ✓ From Kernel 2.2:

- Macros `__init` and `__exit`
- `__init`: if « built-in », free memory after initialization

## ✓ From Kernel 2.4:

- Possibility of giving different names to `init|clean_module()`

```
#include <linux/init.h> /* necessary for macros */

static int init_function (void) { ... }

static void exit_function (void) { ... }

module_init(init_function);
module_exit(exit_function);
```

```
static int __init init_module(void)
```

- `__exit`: if « built-in », omit this function

```
static void __exit cleaup_module(void)
```

# Documentation Macros

## ✓ Module's Documentation

- `MODULE_AUTHOR( "..." );`
- `MODULE_DESCRIPTION( "..." );`
- `MODULE_SUPPORTED_DEVICE( "..." );`

## ✓ From Kernel 2.4:

- **Need to define a license for a module, else:**

```
> insmod module.o
```

```
Warning: loading module.o will taint the kernel: no licence
```

```
See http://www.tux.org/lkml/#export-tainted for information about  
tainted modules
```

```
On entre dans le module
```

```
Module module loaded, with warnings
```

# Modules Licenses

```
MODULE_LICENSE( "... " ); voir include/linux/module.h
```

- ✓ **GPL**
  - GNU Public License v2 or greater
- ✓ **GPL v2**
  - GNU Public License v2
- ✓ **GPL and additional rights**
- ✓ **Dual BSD/GPL**
  - Choice between GNU Public License v2 and BSD
- ✓ **Dual MIT/GPL**
  - Choice between GNU Public License v2 and MIT
- ✓ **Dual MPL/GPL**
  - Choice between GNU Public License v2 and Mozilla
- ✓ **Proprietary**
  - Non free products

# Usefulness of Licenses

- ✓ **Used by kernel developers to identify**
  - Problems from proprietary drivers, they will not try to solve
  
- ✓ **Allows users**
  - To verify that their system is 100% free
  
- ✓ **Allows GNU / Linux distributors**
  - To verify compliance with their licensing policy

# Writing a Module: Some Rules

1/2

## ✓ Includes C:

- The C library is implemented above the kernel and not the reverse
- Unable to use the features of the standard C library (`printf()`, `strcat()`, etc.).
- Linux has some useful functions like C `printk()`
  - which has an interface similar to `printf()`
- Therefore, only the header files of the kernel are allowed.

## ✓ Floating Point:

- Never use floating point numbers in the kernel code. Your code can be run on a processor without calculation unit float (like ARM)
- The emulation by the kernel is possible but very slow

# Writing a Module: Some Rules

2/2

- ✓ **Scope:**
  - Define all symbols as local/static, except those that are exported (to prevent pollution of the namespace)
  
- ✓ **Check:**
  - In the source code: Documentation/CodingStyle
  
- ✓ **It is always good to know and to apply the GNU coding rules:**
  - <http://www.gnu.org/prep/standards.html>

# Compiling a Module

✓ **Commande ligne compilation until 2.4:**

```
- gcc -DMODULE -D__KERNEL__ -c hello.c
```

✓ **A Makefile is needed from 2.6:**

```
# Makefile pour le module hello
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

✓ **This will result in building a file hello.ko (Kernel Object)**

# Commands for Modules Management

## ✓ Key Commands:

- `insmod`: **load a module into memory**
- `modprobe`: **load a module and all its dependencies**
- `rmmmod`: **unload a module from memory**
- `lsmod`: **list all loaded modules** (`cat /proc/modules`)
- `modinfo`: **see information on a module**

## ✓ Example:

```
~> lsmod
Module                Size      Used by      Not tainted
mousedev              4308         1      (autoclean)
input                 3648         0      (autoclean) [mousedev]
usb-storage          70048         0      (unused)
usb-uhci              23568         0      (unused)
usbcore               63756         1      [usb-storage usb-uhci]
eepro100             20276         1
mii                   2464         0      [eepro100]
vfat                  10892         0
fat                   32792         0      [vfat]
ext3                  82536         0      (autoclean)
jbd                   42980         0      (autoclean) [ext3]
```

# Modules Dependencies

1/2

- ✓ The dependencies of a module does not need to be specified explicitly by the creator of the module
- ✓ They are automatically deducted when compiling the kernel, through the symbols exported by the module :
  - ModuleB depends on ModuleA if moduleB used a symbol exported by moduleA.
- ✓ The modules dependencies are stored in:
  - `/lib/modules/<version>/modules.dep`
- ✓ This file is updated (as a root user) with:
  - `depmod -a [<version>]`

# Modules Dependencies

2/2

- ✓ To be used by another module, you must export the symbols
- ✓ By default all the symbols are exported
- ✓ Not to export anything:
  - EXPORT\_NO\_SYMBOL;
- ✓ To enable export mechanism
  - EXPORT\_SYMTAB
  - To define before including module.h
- ✓ Using macros in source code to export symbols
  - EXPORT\_SYMBOL(symbol\_name);
  - EXPORT\_SYMBOL\_NOVERS(symbol\_name);
  - EXPORT\_SYMBOL\_GPL(symbol\_name);
- ✓ The symbols are placed in symbol table module

# Using Static Symbols defined in the Kernel

- ✓ In principle, only the exported symbols
  - Declared "extern"
  - The symbols that have been planned for this
    - API provided by the kernel to modules
- ✓ What to do if you need a static symbol ?
  - Reasonable solution 1 : do not use it !
  - Reasonable solution 2 : modify the source of kernel
    - Add "extern"
    - But you need to recompile the kernel (and this become a specific kernel)
  - Bad solution (but efficient one 😊) : cheat
    - Find the address in the symbol table
      - /proc/ksyms
      - /boot/System.mapXXX

# Example of Using Static Symbol

- ✓ Hypothesis: one wishes to use the variable `module_list`
  - Problem : this symbol is not exported by the kernel
  - Solution to avoid (but efficient) :

- Look for this symbol in `System.map`

```
> grep module_list /boot/System.map-2.4.27
c01180d3 T get_module_list
c023edc0 D module_list
```

- Declare this symbol in your module

```
/* /boot/System.map nous donne l'adresse du symbole 'module_list'
 * Comme module_list est un pointeur sur struct module, ce que nous
 * récupérons est de type struct module **
 */
struct module **module_list_ptr = (struct module **)0xc023edc0;
```

- Why "avoid" anyway??

- We must check (and alter) the code of each module to recompile the kernel (the symbols address change)

# Module Configuration

- ✓ **Some modules need information to run**
  - Hardware address for I/O
  - Parameters to modify a driver behavior (DMA...)
- ✓ **2 ways to get these data**
  - Given by the user
  - Auto-detection
- ✓ **Parameters are passed when loading a module** `insmod`  
**or** `modprobe`
  - `insmod hello param1=5 param2="Hello!"`
- ✓ **Possibility to put them in a configuration file (depending on distribution)**
  - Debian: `/etc/modprobe.d/hello`
    - `options hello param1=3 param2="Hello!"`
  - RedHat: `/etc/modules.conf`

# Passing Parameters to Modules

## Linux 2.4

- ✓ **Passing parameters to a module:**
  - The command line parameters are not passed to module as the classical `argc/argv` mechanism
  - In source code, you need to use macro: `MODULE_PARM(name, type)`
    - `name` = parameter name
    - `type` = string containing type
  - Supported types for variables:
    - "b": byte
    - "h": short int (2 bytes)
    - "i": integer
    - "l": long int
    - "s": string (déclaration de la variable comme `char*`)
  - Documenting parameters is strongly recommended
    - `MODULE_PARM_DESC( "... " );`

# Passing Parameters to Modules

## Linux 2.6

### ✓ Passing parameters to a module:

– **Using macro:** `module_param(name, type, permission)`

- `name` = **parameter name**
- `type` = **string specifying the parameter type**
- `permission` = **exposed in sysfs if different than 0 (file permission)**

– **Supported variables types:**

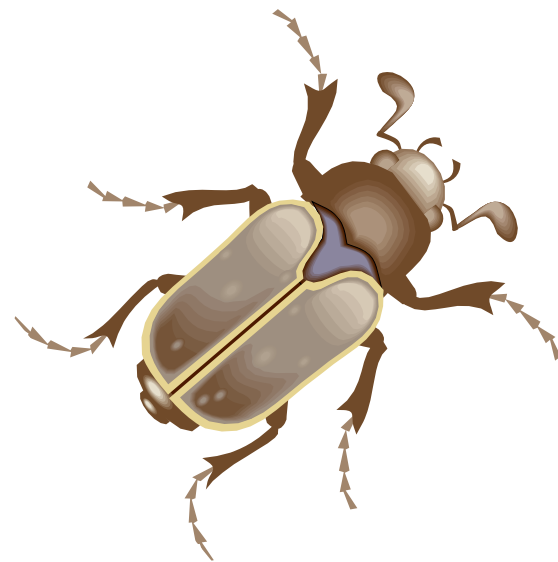
- **bool:** boolean
- **charp:** `char *` (chaîne de caractères)
- **short:** short
- **ushort:** unsigned short
- **int:** integer
- **uint:** unsigned integer
- **long:** long
- **ulong:** unsigned long

– **Macro for importing tables:**

- `module_param_array(name, type, var pointer, permission)`

# Useful References

- ✓ **The Linux Kernel Module Programming Guide**
  - <http://tldp.org/LDP/lkmpg/>



# Kernel Debugging

Or how to remove bugs in your  
kernel developments

# Print Debugging

- ✓ **Universal technique to debug used since the beginning of programming**
  - Printing message at key points in your source code
  - For user programs: `printf`
  - But in kernel space: `printk`
- ✓ **Printed or not in the console or in `/var/log/messages`**
  - Depend on priority `linux/kernel.h` (see next slide)
  - More the printing level is low, more the priority is high
  - Strings "`<[0-7]>`" concatenated to compilation messages
  - If there is no specified level, `DEFAULT_MESSAGE_LOGLEVEL`
  - Else, levels defined in `kernel/printk.c`

# Logging Levels

- ✓ # define KERN\_EMERG "<0>"
  - **Urgent messages just before an unstable system**
- ✓ # define KERN\_ALERT "<1>"
  - **System needing an immediate action**
- ✓ # define KERN\_CRIT "<2>"
  - **Critical situation (software or hardware failure)**
- ✓ # define KERN\_ERR "<3>"
  - **Error situation, hardware problem**
- ✓ # define KERN\_WARNING "<4>"
  - **Problematic situation (not a big problem for the system)**
- ✓ # define KERN\_NOTICE "<5>"
  - **Normal situation but significant**
- ✓ # define KERN\_INFO "<6>"
  - **Information message (for example, name assigned to hardware)**
- ✓ # define KERN\_DEBUG "<7>"
  - **Debugging message**

# Oops

- ✓ **We the kernel detects a problem (illegal operation)**
  - Kernel panic message: “Oops” message
  - Kill the process that made the system fault
  - Even if the system looks like to correctly , side effects might appear due to the stop of the process which generated the problem
- ✓ **A Oops contains**
  - A textual description
  - A Oops number
  - The CPU number
  - The CPU registers
  - The stack
  - Instructions that CPU executed before crash
- ✓ **Do not contain any symbols, just addresses**



# Example of Kernel Oops





# ksymoops

- ✓ **Help to decrypt Oops messages**
  - Convert addresses and code useful information
- ✓ **Easy to use**
  - Copy/Paste the Oops text in a file
- ✓ **Need the following information**
  - System.map of the kernel you use (and which had a problem)
  - List of modules (by default uses `/proc/modules`)
  - List of kernel symbols (`/proc/ksyms`)
  - A copy of the kernel image
- ✓ **With that, you get human readable addresses**
  - EIP; `c88cd018 <[kdb]__memp_fget+158/b54>`
- ✓ **Example of command line**
  - `ksymoops -no-ksyms -m System.map -v vmlinux oops.txt`
- ✓ **See** `Documentation/oops-tracing.txt` **and man** `ksymoops`