



Linux Kernel Modules

Kernel 2.6

Lecturer:

Ing. Roberto Farina

farina@cefriel.it

Summary



- Introduction
 - ▶ Terminology and history
 - ▶ Why to use modules?
- Modules in practice
 - ▶ Behaviour and utilities
 - ▶ Writing and debugging
- News on 2.6 kernel

Terminology



- Base kernel
 - ▶ The part of the kernel that is **bound** into the image you boot
- Linux Kernel Modules (LKM)
 - ▶ AKA: kernel modules, modules
 - Misleading terms!
 - ▶ A **chunk of code** that you add to the kernel **while** it is running
 - They are part of the kernel
 - They does not communicate with the kernel

History



- LKM did not exist since the beginning
 - ▶ Since Linux 1.2 (1995)
- Device drivers and such were **always quite modular**
 - ▶ Small amount of work to make them buildable as LKMs
- Since 2000, virtually everything that makes sense as an LKM has the option to be built this way

Motivation



- Kernel containing support for all devices would be **too big**
- Building **custom** kernels is inconvenient
 - ▶ Adding a new driver would require **rebuilding** the whole kernel and **restarting** the system
- Modules allow to **plug** a new functionality into the kernel at **runtime**
 - ▶ A module is loaded only when it is needed
 - ▶ It is unloaded when it is not needed anymore

LKMs: advantages



- Kernel does not need to be **rebuilt** so often
 - ▶ **Reduce** the likeliness of errors
- Bugs in code do not affect the kernel boot
 - ▶ **Easier** debugging: you know where to look
 - ▶ **Faster** debugging: no need to reboot
- Save memory: loaded when they are needed
 - ▶ The kernel is always in main memory
- Not slower than bound code
 - ▶ A **branch** to the memory location where it resides

What LKMs are used for



- Device drivers
 - ▶ Communicate with specific hardware
- Filesystem drivers
 - ▶ Interpret the content of a filesystem
- System calls
 - ▶ New system calls / Overriding
- Network drivers
 - ▶ Interpret a network protocol
- TTY line discipline
- Executable interpreters
 - ▶ Load and run an executable

Keep an eye...



- It is **NOT** possible to build everything as a module!
 - ▶ E.g.: support for the root filesystem must be bound into the base kernel
- LKMs share lots of properties of user space programs but they are definitely **NOT** user space programs!
- They are **part** of the kernel
 - ▶ They have **free run** of the system
 - ▶ They share the **kernel's code space**
 - ▶ They **can easily crash** it

Initrd



- A way to avoid building the disk device driver into the base kernel
- **INITial RamDisk**
 - ▶ The loader (LILO, GRUB) loads a filesystem into memory as a ramdisk before starting the kernel
 - Mounted as root filesystem
 - It contains the disk device driver and all the needed software
 - ▶ You **MUST** bind
 - The **filesystem driver** for the filesystem in the ramdisk
 - The **executable interpreter** for the programs in the ramdisk

Security Issues (1/2)



- A kernel module has **no limitations** in its privileges
 - ▶ Running somebody else's modules implies giving him full access to the system
- If a module has a security hole, then the kernel and the whole system have too
- Drivers should avoid introducing security bugs
 - ▶ **Buffer overrun** holes are very common
- Drivers should make the **appropriate checks** of a users' privileges

Security Issues (2/2)



- Any input received from user processes should be treated with great suspicion
- Be **careful** with not initialized memory
 - ▶ Memory obtained from the kernel should be **initialized** or **zeroed**, or **information leakage** could result
- **Paranoid** mode
 - ▶ Distrust precompiled kernels: the sources are public and can be maliciously adapted
 - ▶ Disable loading of kernel modules after boot

Summary



- Introduction
 - ▶ History and terminology
 - ▶ Why modules should be used?
- Modules in practice
 - ▶ Behaviour and utilities
 - ▶ Writing and debugging
- News on 2.6 kernel

What an LKM really is



- A single ELF object file
 - ▶ Utilities for loading and unloading
- As part of the Linux Kernel
 - ▶ Built using the **same** kernel build process
 - `make` or `make modules`
 - ▶ Object files **through** the source tree ready to be loaded
 - `make modules_install`
- **Not** part of Linux (not distributed with the kernel)
 - ▶ **Own** build procedures
 - Always end with an **ELF object file**
- The **linking** problem...

Modules in practice



- They are **dependent** on the particular kernel they are built against
 - ▶ “*Couldn’t find kernel version*” error message
 - **.modinfo** section has the version number in it
 - Do we need to recompile them every time?
- They are situated in **/lib/modules/<version>**
- Modules may **export** functions which may be used by other modules
 - ▶ **/lib/modules/<version>/modules.dep**
 - ▶ List of exported symbols in
 - **/usr/src/linux/System.map** (core symbols)
 - **/proc/ksyms** (all symbols)

Modutils



- Displaying loaded modules
 - ▶ `lsmod`
 - ▶ `modinfo` (shows module information)
- Loading a module (as superuser!)
 - ▶ `insmod` (loading only this module)
 - ▶ `modprobe` (loading with dependencies)
 - `/proc/sys/kernel/modprobe`
- Removing a module (as superuser!)
 - ▶ `rmmmod`
 - `--all` option
- Building a map of dependencies (as superuser!)
 - ▶ `depmod -a`

The proc filesystem



- `/proc/modules` holds information about the loaded modules

8139too	18856	1	
mii	4124	1	[8139too]
reiserfs	208272	1	
keybdev	2976	0	(unused)
hid	22404	0	(unused)
input	6208	3	[keybdev mousedev hid]
usb-uhci	27468	0	(unused)
usbcore	82816	1	[hid usb-uhci]
ext3	73376	1	
jbd	56368	1	[ext3]

/proc/modules

- ▶ 0 in column **use count** means it is not used
 - A subroutine returns an indicator for unloading
 - Modules' dependencies
- `/proc/ksyms` lists all the symbols the kernel exports

Linking



- A module is **linked** when it is **loaded**
 - ▶ Symbols get resolved upon insmod'ing
- It **cannot** use library functions
 - ▶ All the symbols it can use are the ones **exported** by the kernel
- It is necessary to use **system calls**

Configuration



- Configured through `/etc/modprobe.conf`
 - ▶ `/etc/modules.conf` for 2.4.x tree
- `modprobe` is passed a string in one of the two forms:
 - ▶ A module name
 - ▶ A more generic identifier

```
alias eth0 8139too
alias sound-slot-0 cs46xx
alias usb-controller usb-uhci
```

`/etc/modprobe.conf`

Kernel version mismatch



- `insmod -f` to force ignoring the version mismatch
- **Symbol versioning**
 - ▶ It allows LKMs to be **sensitive** to the actual content of each kernel subroutine LKMs use
 - ▶ The exported symbols get defined as **macros**
 - Same symbol name plus a **hexadecimal hash value** of the parameter and return value types
 - **gensyms** for the analysis
 - `#define register_chrdev register_chrdev_Rc8dc8350`
 - Both in the source that defines the function and in the source that utilizes it
 - **Modifying the function changes the hash: mismatch!**
 - Load fails if the function changes between two versions
 - ▶ Does not guarantee **compatibility!**

Automatic LKM loading



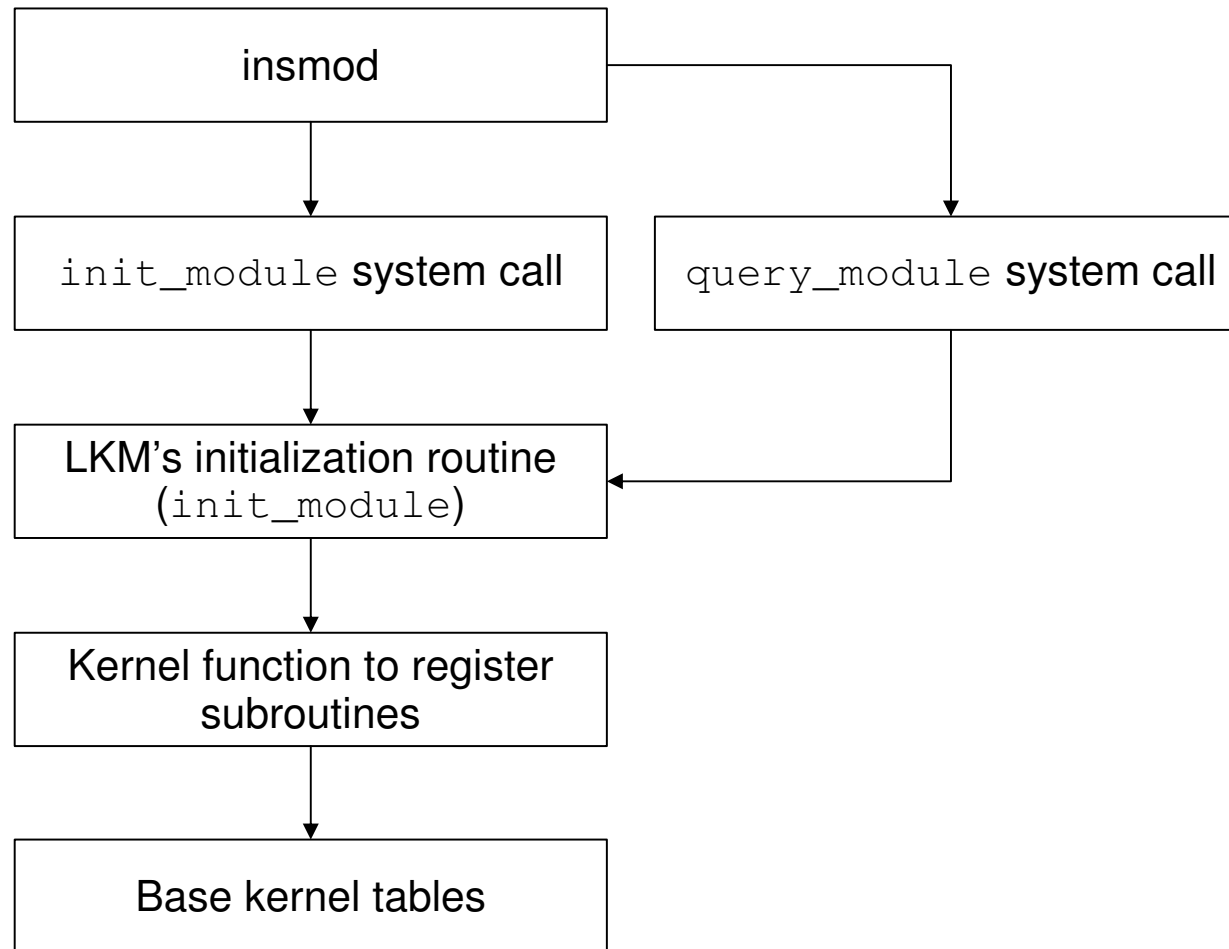
- System can be setup to **automatically** load modules when the kernel first needs it
 - ▶ Kernel module loader (since 2.2)
 - User process (with root rights) that performs modprobe
 - -s (--syslog), -k (--autoclean)
 - ▶ **Kerneld** (older version)
 - IPC message channel with the kernel
 - ▶ In 2.4 the module loading work is submitted to **keventd**
- Significant in systems with **few resources**
 - ▶ Current approach in general-purpose systems is to load all the needed modules at boot time and to leave them in memory

Multiple kernels



- Keeping the old kernel while trying a new one
 - ▶ It is better to keep the old modules until the new kernel is sufficient stable and grants the requested performances
- modprobe “**hunting feature**” loads modules from the appropriate directory
 - ▶ It understands the previously discussed structure
 - ▶ **uname -release** returns the current kernel version

Loading



The first module: minmod



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

int init_module(void)
{
    printk(KERN_INFO "The Minimal Module was
loaded.\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "The Minimal Module was
removed.\n");
}
```

minmod.c

Makefile and compilation



- Makefile

```
obj-m := minmod.o
```

- Compilation

- ▶ A simple command line to be executed from within the directory containing your module's source code

```
make -C /usr/src/linux SUBDIRS=$PWD modules
```

```
make -C /usr/src/linux M=$PWD modules
```

- The compilation step produces the basic object file

- The linker/loader step links in the file

`init/vermagic.o`

- ▶ It **integrates information** about the kernel against which the module was compiled and provides more **stringent checks** used when loading the module

Usage of minmod



- Get a root console
 - ▶ su root
- Compile the module
 - ▶ Be sure to have kernel sources in `/usr/src/linux`
- Load the module
 - ▶ insmod minmod.o
- Check that the module is loaded
 - ▶ cat /proc/modules
- Remove the module
 - ▶ rmmod minmod

First considerations on the code



- `printk` instead of `printf`
 - ▶ But an IO library is never included...
- `init_module` and `cleanup_module` functions
 - ▶ Better ways to do initialization and cleanup in recent kernel versions
- `return 0;`
 - ▶ If init function return a different value the loading fails
- `#include <linux/module.h>`
 - ▶ It is needed by all modules
- `#include <linux/kernel.h>`
 - ▶ It is needed for the `KERN_INFO`

First Help: Logging



- `printk(char *format, ...)`
 - ▶ Similar to `printf` but...
 - ▶ Format is preceded by priority
 - `KERN_EMERG`, `KERN_ALERT`, `KERN_CRIT`, `KERN_ERR`, `KERN_WARNING`, `KERN_NOTICE`, `KERN_INFO`, `KERN_DEBUG`
 - E.g.: `printk(KERN_DEBUG "Just for fun...")`
 - ▶ Writes to a circular buffer
 - ▶ Buffer is accessible in `/proc/kmsg`
 - ▶ `klogd` obtains messages from it and logs them using `syslog` to `/var/log/messages`
 - ▶ Can be used for manual debugging (not the best way...)

Further considerations



```
# insmod ./minmod.o
```

```
Warning: loading ./minmod.o will taint the kernel: no license
```

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

```
Module minmod loaded, with warnings
```

- What this message means?
- The kernel argues that minmod has no licence
 - ▶ In kernel 2.4 and later, a mechanism was devised to **identify** code licensed under the GPL (and friends)
 - People can be **warned** of non open-source code

Module Licensing



- You must specify module licence and info

```
MODULE_AUTHOR("Roberto Farina");  
MODULE_DESCRIPTION("Minimal module");  
MODULE_LICENCE("GPL");  
MODULE_SUPPORTED_DEVICE("testdevice");
```

- The licensor can cause his program to export symbols under a special name

- ▶ Prefix **GPLONLY**

- Modern version of insmod knows to check for `GPLONLY_<symbol_name>` if it cannot find `<symbol_name>`

- ▶ It refuses to load the module if it is not licensed to the public under GPL

Newer methods for initialization



- It is possible to **rename** the functions `init_module` and `cleanup_module`
 - ▶ As of Linux 2.4
 - ▶ `module_init()` and `module_exit()` macros to specify them
 - In 2.6, these macros **MUST** be used to register the init and exit functions
 - Strictly necessary if you want to compile the specified module into the kernel
 - ▶ It is necessary to include `linux/init.h`, that is where those macros are defined
 - ▶ The functions must be defined **BEFORE** calling the macros

Freeing memory



- As of Linux 2.2
- `__init`, `__exit` and `__initdata` macros
 - ▶ They are defined in `linux/init.h`
 - ▶ `__init` causes the `init` function to be **discarded** and its memory **freed** once the `init` function finishes for built-in drivers, but not loadable modules
 - ▶ `__exit` causes the **omission** of the function when the module is built into the kernel
 - It has no effect on loadable modules
 - ▶ `__initdata` works similarly to `__init` but for `init` variables rather than functions
- At boot time:

```
Freeing unused kernel memory: 156k freed
```

A module on multiple files



- Complex modules could be written in multiple files
- Split minmod4 in two files:
 - ▶ minmod_1: init function
 - ▶ minmod_2: exit function
- No changes introduced in the source code!
- Makefile needs some small changes
 - ▶ Module name specification
 - ▶ Files included in that module

```
obj-m := minmod4m.o  
minmod4m-objs := minmod_1.o minmod_2.o
```

Makefile

Passing Parameters - old form



- Useful to specify **init information**
- There are macros in this case too

```
MODULE_PARM(myshort, "h");  
MODULE_PARM(myint, "i");  
MODULE_PARM(mylong, "l");  
MODULE_PARM(mystring, "s");
```

- Several types supported:
 - ▶ b: single byte
 - ▶ h: short int
 - ▶ i : integer
 - ▶ l : long
 - ▶ s : string
 - String should be declared as char * and insmod will allocate memory for them

Passing Parameters - old form



- Array params are also supported
 - ▶ An array of exact 5 integer:
`MODULE_PARM(myarray, "5i");`
 - ▶ An array of int with at least 2 and no more than 5 values:
`MODULE_PARM(myarray2, "2-5i");`
- If it is compiled as a module
 - ▶ Initialization at load-time through the command line

```
$ insmod minmod_param i=5  
$ insmod minmod_param myarray=1,2,3,4,5  
$ insmod minmod_param myarray2=1,2,3,4,5  
$ insmod minmod_param myarray2=1,2,3  
$ insmod minmod_param myarray=1,2,3 -- ERROR
```
- There's no way to know how many values the command line provides

Passing Parameters - new form



- Now there's a new form to specify module parameters:

```
module_param( name, type, perm );
```

- ▶ Name: name of the parameter
- ▶ Type: type of the parameter
- ▶ Perm: permission bits for exposing parameter in sysfs

```
module_param_array( name, type, num, perm );
```

- ▶ Name: name of the parameter
- ▶ Type: type of the parameter
- ▶ Num: pointer to a variable that will store the number of elements of the array initialized by the user at module loading time
- ▶ Perm: permission bits for exposing parameter in sysfs

- With the new form, there's a way to know how many values the command line provides!

Passing Parameters - new form



- Permissions
 - ▶ S_IRUSR: Read right for owner
 - ▶ S_IRGRP: Read right for group
 - ▶ S_IROTH: Read right for other
 - ▶ S_IWUSR: Write right for owner
 - ▶ S_IWGRP: Write right for group
 - ▶ S_IWOTH: Write right for other
- Via sysfs, users could modify parameters value when the module is already loaded
- Permissions could also be specified in octal form
 - ▶ S_IRUSR | S_IRGRP
 - ▶ 0440

Identifying parameters at boot time



- If it is bound into the base kernel
 - ▶ Initialization at boot-time
 - Some problem may rise...
- No problems at load-time
 - ▶ Module invoked with **list** of parameters
- At boot-time there is **only one string** of kernel boot parameters
 - ▶ If there is a module named **xyz**, then the kernel boot parameter **xyz** is for that module
 - ▶ The value of that parameter is an **arbitrary** string that makes sense only to the module

The proc filesystem



- More complex information may be presented using **proc** filesystem
 - ▶ Mounted on **/proc**
- Simple functions to create
 - ▶ Files
 - `create_proc_entry`
 - `remove_proc_entry`
 - ▶ Directories
 - `proc_mkdir`

A Buggy Module



- In kernel a thread
 - ▶ is associated with a user-process
 - E.g.: syscall from userland
 - If a problem occurs, calling process is terminated
 - ▶ is not associated
 - E.g.: IRQ handler, bottom-halves, tasklets
 - If a problem occurs kernel panics
- It generates an oops message to the kernel log
 - ▶ Stuff that the Linux kernel generates when it detects an **internal kernel error**
 - ▶ **ksymoops**

ksymoops



- Program that **interprets** and **displays** “oops” messages
 - ▶ It looks at the **hexadecimal** addresses, looks them up in the **kernel symbol table** and **translates** the addresses in the oops messages to **symbolic** addresses
- If a module crashes, ksymoops can tell:
 - ▶ in **what** LKM is the instruction that crashed
 - ▶ **where** is the instruction relative to an **asm** listing
- ksymoops must be able to get the loadpoints and lengths of the various sections of the LKM
- But ksymoops does not know these information...

Debugging solution



- insmod adds some symbol as it loads the LKM
 - ▶ In `/proc/ksyms`:
 - `__insmod_name_Ssection_Llenght`
 - name : the module name as in `/proc/modules`
 - section : the section name (e.g.: `.text`)
 - length : length of the section, in decimal
 - `__insmod_name_Ofilespec_Mmtime_Vversion`
 - name : the module name as in `/proc/modules`
 - filespec : file specification used to identify the file containing the LKM when it was loaded
 - mtime : modification time of that file (UNIX style)
 - version : kernel version level for which the LKM was built (same as in the `.modinfo` section)

The kdb Kernel Debugger



- <http://oss.sgi.com/projects/kdb>
- Non-official patch to the kernel
- Key combination stops the system and enters the debugger
- The debugger is similar to **gdb**

- Other debuggers
 - ▶ IKD
 - <ftp://ftp.kernel.org/pub/linux/kernel/people/andrea/ikd>

User-Mode Linux Kernel



- <http://user-mode-linux.sourceforge.net/>
- Runs as separate process on Linux machine
- A virtual machine where to run buggy software
- Does not affect the system
 - ▶ Disk storage entirely contained in a single file
 - ▶ Experiments with new Linux kernels or distributions
- Possible to debug it using gdb

Overview



- Introduction
 - ▶ History and terminology
 - ▶ Why modules should be used?
- Modules in practice
 - ▶ Behaviour and utilities
 - ▶ Writing and debugging
- News on 2.6 kernel

2.6: safety features



- Major changes to improve **stability**
- Process of unloading modules has been changed to **reduce** the risk of system crash
 - ▶ It is possible to **disable** module unloading at all
- **Standardization** of the process by which modules determine and **announce** what hardware they support
 - ▶ In previous versions this information was not available **outside** the module

What's new in 2.6



- **Module versioning support**
 - ▶ It adds **extra** versioning information to compiled modules at build-time
 - Designed to help increase module **portability** to kernels other than the one that they were compiled against
- **Module unloading option**
 - ▶ It must be enabled if you want your kernel to be able to unload modules when they are **no longer needed**
 - Especially important in **resource-constrained** and power-sensitive environment such as **embedded systems**
 - ▶ **Forced module unloading** to be able to forcibly unload modules even if the kernel believes they are in use

Device drivers



- 2.6 kernels introduce a **new unified framework** for device drivers
 - ▶ It requires changes to custom device drivers that you may have developed to run under earlier versions of the Linux kernel
 - ▶ **Full and complete** support for Plug and Play and Power Management
 - It defines the **interfaces** that subsystems can use when **communicating** with individual drivers
- **Sysfs** filesystem to provide a **hierarchical** view of each system's device tree

Updating the basic structure



- You must use the `module_init()` and `module_exit()` macros to register the names of your initialization and exit routines
 - ▶ They are only strictly necessary if you intend to compile the specified module into the kernel
- `#define MODULE` statement is no longer needed
 - ▶ Automatically defined and verified by the kernel build system
- Use of the `MODULE_LICENSE` macro is strongly recommended

Other changes in 2.6



- New interface used for modules that take parameters
 - ▶ The `MODULE_PARM()` macro has been replaced by explicit parameter declarations made using the new `module_param()` macro
 - `moduleparam.h`
- Enhanced preemptability and SMP-awareness introduce some **new concerns** for driver writers
 - ▶ Drivers should use a spinlock or mutex to **protect** data that could be accessed from multiple processors
- Module reference counts are managed and manipulated differently

Kernel 2.4



- Compilation of modules for Linux 2.4 is quite different
 - ▶ Not using kernel build infrastructure
 - ▶ Makefile provide all information for a straightforward build process

Compiling minmod: Makefile



```
KERNELDIR=/usr/src/linux

include $(KERNELDIR)/.config

CFLAGS = -c -D__KERNEL__ -DMODULE -
        I$(KERNELDIR)/include -O2 -Wall

ifdef CONFIG_SMP
CFLAGS += -D__SMP__ -DSMP
endif

all: minmod.o
```

Makefile

Options for compilation (1/2)



- `-c` : modules are not independent executables
 - ▶ Does not perform the linking step
- `-Wall` : to turn on compiler warnings
 - ▶ A programming mistake in a module can take the system down
- `-D__KERNEL__` : we are in kernel mode
- `-DMODULE` : a module is being compiled

Options for compilation (2/2)



- `-I$(KERNELDIR)/include` : to specify the headers of the kernel you are compiling against
 - ▶ `-I/lib/modules/'uname -r'/build/include`
 - ▶ `-isystem /lib/modules/'uname -r'/build/include` tells gcc to suppress some “unused variable” warnings that `-Wall` causes including `modules.h`
- `-O2` : optimization flag
 - ▶ Kernel makes extensive use of inline functions
 - ▶ Some of the assembler macros calls will be mistaken by the compiler for function calls (causing loading to fail)

A module on multiple files



- `#define __NO_VERSION__` in all the source files but one
 - ▶ `module.h` normally includes the definition of `kernel_version`
- Compile all the sources as usual
- Combine the object files in a single one

```
ld -m elf_i386 -r -o <module_name> <file1.o> <file2.o>
```

 - `-r` : relocatable output!

Useful documents



- Bryan Henderson - Linux Loadable Kernel Module HOWTO
- Peter Jay Salzman, Ori Pomerantz - The Linux Kernel Module Programming Guide
- Robert Love - Linux kernel development
- Daniel P. Bovet, Marco Cesati - Understanding the Linux Kernel
- LDP : Linux Documentation Project
- Documentation directory in the kernel tree source
- kernelnewbies.org
- The Linux Module Kernel programming guide