Politecnico di Milano

Linux Kernel Modules: an Introduction

Kernel 2.4

Lecturer:
Ing. Luca Pizzamiglio
Politecnico di Milano, DEI
luca.pizzamiglio@gmail.com

Overview

- Introduction
  - Terminology and history
  - Why to use modules?
- Modules in practice
  - Behaviour and utilities
  - Writing and debugging

Terminology

- Base kernel
  - The part of the kernel that is bound into the image you boot
- LKM (Linux Kernel Modules)
  - 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
      - They communicate with the base 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: can be swapped
  - The kernel is always in main memory
- Not slower than bound modules
  - A branch to the memory location where it resides
    - The page might be unloaded from memory

What LKMs are used for

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

Keep an eye…

- It is NOT possible to build everything as a module!
  - Root filesystem support must be bound into the base kernel
    - initrd
- 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 the root privileges
  - Running somebody else's modules is like giving him a root shell
- 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 uninitialized 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

Overview

- Introduction
  - History and terminology
  - Why modules should be used?
- Modules in practice
  - Behaviour and utilities
  - Writing and debugging

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 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
    - `/boot/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!)
  - rmmod
    - --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   0 [8139too]
reiserfs              208272   1 (autoclean)
keybdev                 2976   0 (unused)
hid                    22404   0 (unused)
input                   6208   0 [keybdev mousedev hid]
usb-uhci               27468   0 (unused)
usbcore                82816   1 [hid usb-uhci]
ext3                   73376   1
jbd                    56368   1 [ext3]
```

/proc/modules

  - -1 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/modules.conf`**
  - **`/etc/modprobe.conf`** for 2.6.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/modules.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
      - **genksyms** 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!

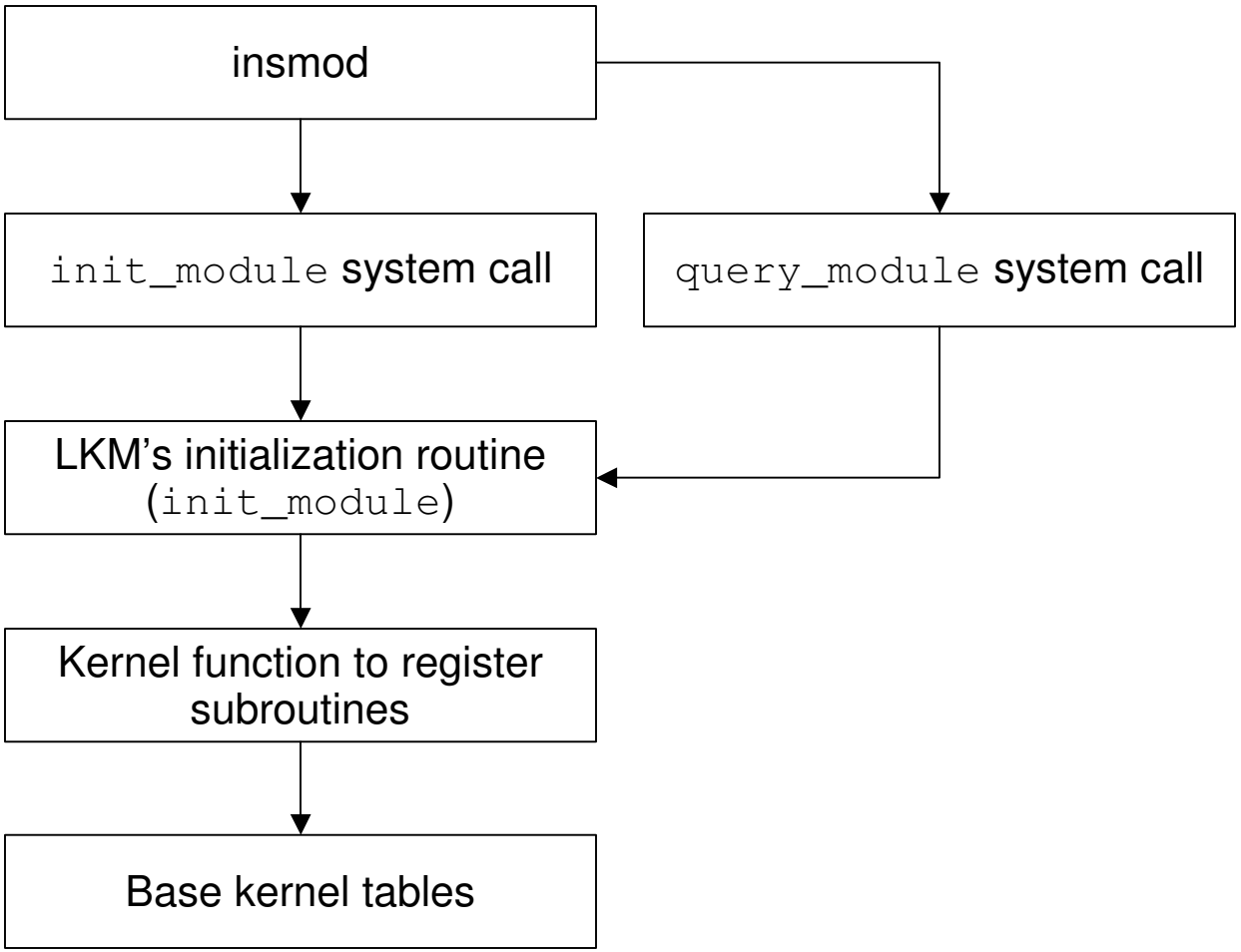
Automatic LKM loading

- 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

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)

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("Luca Pizzamiglio");
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
```


Passing Parameters

- Useful to specify init information
  - IO Addresses, Interrupts, …
- There are macros in this case too

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

- If it is compiled as a module
  - Initialization at load-time
- If it is bound into the base kernel
  - Initialization at boot-time
    - Some problem may rise…

Identifying parameters

- 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 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!

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 lenghts 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)

2.6 new 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

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

Useful documents

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

