Introducing the Linux kernel architecture

Philippe Thierry (philippe.thierry@reseau-libre.net)

École Centrale d’Électronique

16 Nov 2010
Introduction

1. Summary
2. Aim of this document

General presentation

1. What is Linux?
2. The Linux kernel services

Communicate with Linux

1. Through the syscalls
2. Using char devices
3. Through Netlink sockets
4. Using the procfs
5. Using sysctl

The kernel production

1. Bases of the production system
2. Writing a kernel module

Communicate with the user space

1. Through the procfs
2. Through the sysctl
3. Through a device
4. Through new syscalls

Conclusion

1. General conclusion
2. Bibliography
3. Questions
4. Index

Philippe Thierry (philippe.thierry@reseau-libre.net)
This course proposes an introduction to the Linux kernel architecture.

The goal is to give a first vision of the kernel architecture and how kernel modules are integrated in the overall Linux production system.

This course also presents how work the modules you will have to implement in order to apprehend their architecture.
What is Linux?

- Linux is a kernel which has been created in 1991 by Linus Torvald
- This kernel is compliant to most of the POSIX norms
- The current main kernel revision (2.6) contains nearly 9 millions lines of code
  - written in C and assembler (for low level parts)
  - supporting approximatively 10 hardware architectures (IA32, ARM, SPARC, PPC...)
  - containing a huge number of drivers and services
- It is considered that there are around 10,000 usual contributors to the Linux kernel in the world
The Linux kernel is rich and contains a huge number of communicating services and modules.
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
What does Linux contains

The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
- network
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
- network
- USB stack
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
- network
- USB stack
- video and sound
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
- network
- USB stack
- video and sound
- security and cryptography
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
- network
- USB stack
- video and sound
- security and cryptography
- virtualization
The Linux kernel is rich and contains a huge number of communicating services and modules

- base services (processor, memory, interrupt handlers and schedulers)
- file systems
- network
- USB stack
- video and sound
- security and cryptography
- virtualization
- ...

Philippe Thierry (philippe.thierry@reseau-libre.net)
Like all kernels, Linux proposes the four base services which are:

- The processor and board support (internal processor initialization and configuration, cache controller configuration, DRAM and buses initialization...)
- The memory manager
- The interrupt handlers and vector
- The scheduling and tasks management sub-system

These four elements define a kernel base. They are also implemented in micro-kernel architectures.
The file systems

- The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

1. The directory containing the overall system configuration: /etc
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

1. The directory containing the overall system configuration: `/etc`
2. Directories containing the main binaries needed in order to boot the GNU/Linux environment: `/bin` et `/sbin`, and the associated libraries: `/lib`
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

1. The directory containing the overall system configuration: /etc
2. Directories containing the main binaries needed in order to boot the GNU/Linux environment: /bin and /sbin, and the associated libraries: /lib
3. The directory containing variable contents (software caches, logs, etc): /var
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

1. The directory containing the overall system configuration: /etc
2. Directories containing the main binaries needed in order to boot the GNU/Linux environment: /bin et /sbin, and the associated libraries: /lib
3. The directory containing variable contents (software caches, logs, etc): /var
4. The directory containing the users directories: /home
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

- The directory containing the boot sub-system (kernel, boot-loader configuration, etc): `/boot`
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

5. The directory containing the boot sub-system (kernel, boot-loader configuration, etc): /boot

6. The super-user home directory (root): /root
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

5. The directory containing the boot sub-system (kernel, boot-loader configuration, etc): /boot
6. The super-user home directory (root): /root
7. The directory containing all unneccessaries softwares and their dependencies: /usr
The Linux file systems respect the directory structure of the UNIX software platform. In the root file systems, there are various directory families.

- The directory containing the boot sub-system (kernel, boot-loader configuration, etc): /boot
- The super-user home directory (root): /root
- The directory containing all unneccessary softwares and their dependencies: /usr
- The temporary file system, accessible to all users and purged at each system reboot: /tmp
Most of the file systems are usually hosted in a mass storage device like a hard disk drive, a flash memory, etc.

To these directories some more are added, corresponding to virtual file systems *virtuels*, generated by the kernel execution.
Most of the file systems are usually hosted in a mass storage device like a hard disk drive, a flash memory, etc.

To these directories some more are added, corresponding to virtual file systems *virtuels*, generated by the kernel execution.

- A directory containing all informations about the plateform state and all its components, accessible by all softwares: `/proc`
The file systems

- Most of the file systems are usually hosted in a mass storage device like a hard disk drive, a flash memory, etc.
- To these directories some more are added, corresponding to virtual file systems *virtuals*, generated by the kernel execution.

- A directory containing all informations about the plateform state and all its components, accessible by all softwares: `/proc`
- A directory containing all information on the kernel and its modules current execution: `/sys`
The file systems

- Most of the file systems are usually hosted in a mass storage device like a hard disk drive, a flash memory, etc.
- To these directories some more are added, corresponding to virtual file systems *virtuels*, generated by the kernel execution.
  - A directory containing all informations about the plateform state and all its components, accessible by all softwares: `/proc`
  - A directory containing all information on the kernel and its modules current execution: `/sys`
  - A directory given a direct access to the devices drivers through special files named *devices*: `/dev`
The linux file system architecture is stratified. This point is very important to understand correctly how the kernel interfaces works.
The network stacks

- The Linux kernel implements multiple network stacks, supporting various legacy protocols (ATM, FrameRelay, Token Ring...) and also usual protocols (IPv4 et IPv6, ZigBee, 802.11a,b,g,n).

- Most of the kernel implementations are open-source, though some of the network devices stay closed-source
  - This is the case of some of the Wifi controllers drivers.
  - This is also the case of the telephony support (2G, 3G...)
The Linux kernel implements multiple network stacks, supporting various legacy protocols (ATM, FrameRelay, Token Ring...) and also usual protocols (IPv4 et IPv6, ZigBee, 802.11a,b,g,n).

Most of the kernel implementations are open-source, though some of the network devices stay closed-source

- This is the case of some of the Wifi controller drivers.
- This is also the case of the telephony support (2G, 3G...)

In order to integrate the network stacks in a coherent sub-system, Linux support a complete canvas named Netfilter.

- This canvas allows various kernel modules to implement some hooks in layer 2 and 3 of the OSI stack, permitting the implementation various elements outside of the canvas in order to support some added values like connetracking or filtering.
- This is the case of the kernel part of iptables or the IPSec management modules.
The Linux kernel services

USB stack management

- Linux support a complete USB stack
- This stack implements the various USB standardized protocols
  - OHCI et UHCI, of the USB 1 norm
  - EHCI, of the USB 2 norm
  - XHCI, of the USB 3 norm
- Linux contains drivers for a huge number of USB devices like printers, webcam, keyboard and mice, or other classical USB gadgets.
Video and audio support

- Graphic card support and drivers
- Sound support
The Linux kernel services

Video and audio support

- Graphic card support and drivers
  - The kernel-level video drivers allows the usage of basics elements like framebuffers or VGA subsystem, common to all the graphic cards

- Sound support
The Linux kernel services

Video and audio support

- **Graphic card support and drivers**
  - The kernel-level video drivers allows the usage of basics elements like framebuffers or VGA subsystem, common to all the graphic cards
  - The configuration and management of the advanced graphic chipsets are done in userspace by Xorg or X11R6. This server also manage mouse and keyboard events, even if the underlying USB support is done in kernel space

- **Sound support**
Video and audio support

- Graphic card support and drivers
  - The kernel-level video drivers allows the usage of basics elements like framebuffers or VGA subsystem, common to all the graphic cards
  - The configuration and management of the advanced graphic chipsets are done in userspace by Xorg or X11R6. This server also manage mouse and keyboard events, even if the underlying USB support is done in kernel space

- Sound support
  - The Linux sound system, named ALSA (Advanced Linux Sound Architecture) is a complex software architecture which is done explained here
Security and cryptography supports

- Linux implements the security elements proposed by the POSIX norm

Philippe Thierry (philippe.thierry@reseau-libre.net)
Security and cryptography supports

- Linux implements the security elements proposed by the POSIX norm
  - The UNIX rights management, based on a DAC policy (Discretionary Access Control)
Security and cryptography supports

- Linux implements the security elements proposed by the POSIX norm
  - The UNIX rights management, based on a DAC policy (Discretionary Access Control)
  - The UNIX capabilities, managed through properties describing the rights on the network sub-system, administrative tasks and controls, etc and based on inheritance and voluntary loss
Security and cryptography supports

- Linux implements the security elements proposed by the POSIX norm
  - The UNIX rights management, based on a DAC policy (Discretionary Access Control)
  - The UNIX capabilities, managed through properties describing the rights on the network sub-system, administrative tasks and controls, etc and based on inheritance and voluntary loss
- Moreover, Linux supports various MAC and RBAC solution (SMAC, SeLinux, GRSecurity, AppArmor...)

Philippe Thierry (philippe.thierry@reseau-libre.net)
Security and cryptography supports

- Linux implements the security elements proposed by the POSIX norm
  - The UNIX rights management, based on a DAC policy (Discretionary Access Control)
  - The UNIX capabilities, managed through properties describing the rights on the network sub-system, administrative tasks and controls, etc and based on inheritance and voluntary loss

- Moreover, Linux supports various MAC and RBAC solution (SMAC, SeLinux, GRSecurity, AppArmor...)

- Linux also implements some cryptographic algorithms, both symetric and asymetrics like DES, AES, RSA, SHA1, SHA256... used in various components (IPSec, filesystem cryptography, etc)
Focus on the UNIX rights

- In a default configuration, Linux support Discretionary Access Control
- UNIX wrights are manaeged through an octal value between 0000 and 7777, each digit represents a specific wright corresponding to a specific scope

```
ugsrwx rwx rwx
```

<table>
<thead>
<tr>
<th>Rights for others</th>
<th>Rights for the owner’s group</th>
<th>Rights for the owner</th>
<th>Special rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 1 0</td>
<td>1 0 1</td>
<td>1 0 1 1 0</td>
<td>0 7 5 5</td>
</tr>
<tr>
<td>0 0 0 1 1 0 1 0</td>
<td>1 0 0</td>
<td>1 0 0 1</td>
<td>0 6 4 4</td>
</tr>
</tbody>
</table>

Philippe Thierry (philippe.thierry@reseau-libre.net) Linux kernel course 16 Nov 2010
Virtualization support in the mainstrea is relatively new in the Linux kernel. Various existing solution are implemented:
Virtualization support in the mainstream is relatively new in the Linux kernel. Various existing solutions are implemented:

- OpenVZ, which is a solution proposing the virtualization of the network stack, permitting multiple lightweight virtual machines with autonomous network configuration. Only one kernel is executed on the host.
Virtualization support in the mainstream is relatively new in the Linux kernel. Various existing solutions are implemented:

- **OpenVZ**, which is a solution proposing the virtualization of the network stack, permitting multiple lightweight virtual machines with autonomous network configuration. Only one kernel is executed on the host.

- **Xen** is a well-known virtualization architecture based on a lightweight hypervisor (approx 200 KLOC) and supporting paravirtualization and Hardware-based virtualization, allowing the Xen hypervisor to execute Windows on Intel-VT compatible hardware. There is one independent Linux kernel per Linux-based virtual machine.
Virtualization support in the mainstream is relatively new in the Linux kernel. Various existing solutions are implemented:

- OpenVZ, which is a solution proposing the virtualization of the network stack, permitting multiple lightweight virtual machines with autonomous network configuration. Only one kernel is executed on the host.

- Xen is a well-known virtualization architecture based on a lightweight hypervisor (approx 200 KLOC) and supporting paravirtualization and Hardware-based virtualization, allowing the Xen hypervisor to execute Windows on Intel-VT compatible hardware. There is one independent Linux kernel per Linux-based virtual machine.

- KVM is the only official mainstream solution. This virtualization architecture uses a complete Linux kernel as a hypervisor solution, and one running Linux kernel per virtual machine.
Communicate through syscalls

- This is the most logical communication vector
- In the Linux kernel, syscalls are implemented through the software interrupt 80, named the Linux syscall gate.
- Syscall are managed like a vector containing addresses of each function.
Through the syscalls

Communicate through syscalls

- This is the most logical communication vector
- In the Linux kernel, syscalls are implemented through the software interrupt 80, named the Linux syscall gate.
- Syscall are managed like a vector containing addresses of each function.
- Usual syscalls Linux implements are used for two main needs
Communicate through syscalls

- This is the most logical communication vector
- In the Linux kernel, syscalls are implemented through the software interrupt 80, named the Linux syscall gate.
- Syscall are managed like a vector containing addresses of each function.
- Usual syscalls Linux implements are used for two main needs
  - Implementing a part of the POSIX API. This is the case of `open()` or `read()`
Communicate through syscalls

- This is the most logical communication vector
- In the Linux kernel, syscalls are implemented through the software interrupt 80, named the Linux syscall gate.
- Syscall are managed like a vector containing addresses of each function.
- Usual syscalls Linux implements are used for two main needs
  - Implementing a part of the POSIX API. This is the case of `open()` or `read()`
  - Implementing configuration elements and communication with the underlying kernel, like `prctl()`
Communicating through syscalls

- Because syscall are executed through a software interrupt, the syscall vector is hardware dependent.
- Most of the base elements of the kernel core implements syscalls.
Communicating through syscalls

- Because syscall are executed through a software interrupt, the syscall vector is hardware dependent.
- Most of the base elements of the kernel core implements syscalls.
- Today, Linux contains approximatively 250 syscalls
The char device API

- This is the case of serial devices
- Lots of virtual devices also implement char device interface
Using char devices

- This is the case of serial devices
- Lots of virtual devices also implement char device interface
  - In order to configure some drivers
  - In order to define a communication API using simple syscalls like `open()`, `read()`, `write()`, `ioctl()` and `close()`
  - A certain part of the communication interface between userspace and kernel is done through that devices family
Using char devices

- Historically one of the first ways to communicate between userspace and kernel
- Based on a standard char device
- Used in order to communicate in both way between kernel and softwares

Using IOCTL
Historically one of the first ways to communicate between userspace and kernel

Based on a standard char device

Used in order to communicate in both way between kernel and softwares

```
#include <sys/ioctl.h>

int ioctl(int fd, int cmd, void* args);
```
Using Netlink Sockets

- Netlink sockets has been created in order to replace the communication interfaces based on IOCTL.
- The communication is based on a specific socket family: `AF_NETLINK`.
Using Netlink Sockets

- Netlink sockets has been created in order to replace the communication interfaces based on IOCTL.
- The communication is based on a specific socket family: AF_NETLINK.
  - Such interface allows the usage of advanced synchronisation mechanisms.
Using Netlink Sockets

- Netlink sockets has been created in order to replace the communication interfaces based on IOCTL
- The communication is based on a specific socket family: AF_NETLINK
  - Such interface allows the usage of advanced synchronisation mechanisms
  - Using Netlink socket, various protocols are proposed, mostly in order to configure the network stacks
Using Netlink Sockets

- Netlink sockets has been created in order to replace the communication interfaces based on IOCTL
- The communication is based on a specific socket family: AF_NETLINK
  - Such interface allows the usage of advanced synchronisation mechanisms
  - Using Netlink socket, various protocols are proposed, mostly in order to configure the network stacks
  - It is also possible to implement specific protocols in for one’s own needs, and to manage advanced communications like multicast support
    - where the kernel is registered like a source
    - where softwares using the associated netlink socket are targeted
Using Netlink Sockets

- Netlink sockets has been created in order to replace the communication interfaces based on IOCTL.
- The communication is based on a specific socket family: `AF_NETLINK`.
  - Such interface allows the usage of advanced synchronisation mechanisms.
  - Using Netlink socket, various protocols are proposed, mostly in order to configure the network stacks.
  - It is also possible to implement specific protocols in for one’s own needs, and to manage advanced communications like multicast support.
    - where the kernel is registered like a source
    - where softwares using the associated netlink socket are targeted.
  - Such behaviour permits the usage of elaborated flow control, through the usage of kernel-based synchronisation.
The procfs presents various files which can be read and sometimes write in order to communicate.

Most of them are used only to give back some informations:

- `/proc/cpuinfo` gives the processor(s) informations
- `/proc/acpi/thermal_zone/temperature` gives the temperature informations

Some permit userspace softwares to supply some informations to the kernel.

On the kernel side, procfs entries are implemented similarly to char device drivers, with some subtleties.
Using sysctl

- From the user space, `sysctl` are used through the `sysctl` command
- This command permit to configure a lot of kernel variables
- From the kernel space, proposing a new variable to the sysctl subsystem is easy
Linux is based on an advanced Makefile architecture, using a complex and efficient grammar.

- The production system has been implemented specifically in order to simplify the integration of new elements, through the usage of Kconfig files.
- There is different configuration front-ends.
Producing a Linux kernel

- Linux is based on an advanced Makefile architecture, using a complex and efficient grammar.
- The production system has been implemented specifically in order to simplify the integration of new elements, through the usage of Kconfig files.
- There is different configuration front-ends:
  1. using command lines, with `make config`
Bases of the production system

Producing a Linux kernel

- Linux is based on an advanced Makefile architecture, using a complex and efficient grammar.
- The production system has been implemented specifically in order to simplify the integration of new elements, through the usage of Kconfig files.
- There is different configuration front-ends:
  1. using command lines, with `make config`
  2. using a text-based graphical interface with ncurses: `make menuconfig`
Producing a Linux kernel

- Linux is based on an advanced Makefile architecture, using a complex and efficient grammar.
- The production system has been implemented specifically in order to simplify the integration of new elements, through the usage of Kconfig files.
- There is different configuration front-ends:
  1. using command lines, with `make config`
  2. using a text-based graphical interface with ncurses: `make menuconfig`
  3. using a complete graphical user interface: `make xconfig`
What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.
What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the zlib
Writing a kernel module

What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the \textit{zlib}
- These functions are general purpose functions
When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the \textit{zlib}
- These functions are general purpose functions
  \begin{enumerate}
  \item \textit{printk} is the replacement of printf
  \end{enumerate}
What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the zlib
- These functions are general purpose functions
  1. *printk* is the replacement of sprintf
  2. *kmalloc* is the replacement of malloc
Writing a kernel module

What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the zlib
- These functions are general purpose functions
  
  1. printk is the replacement of printf
  2. kmalloc is the replacement of malloc
  3. There is no filesystem support, no accessible file or directory
Writing a kernel module

What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the *zlib*
- These functions are general purpose functions
  1. *printk* is the replacement of printf
  2. *kmalloc* is the replacement of malloc
  3. There is no filesystem support, no accessible file or directory
- To these functions, all locally exported functions, like other drivers functions are also accessible. All these function are added to a *symbol table*. There is two symbol tables
What are the differences between userspace and kernel development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the zlib
- These functions are general purpose functions
  1. `printk` is the replacement of `printf`
  2. `kmalloc` is the replacement of `malloc`
  3. There is no filesystem support, no accessible file or directory
- To these functions, all locally exported functions, like other drivers functions are also accessible. All these function are added to a symbol table. There is two symbol tables
  1. Symbols accessible only to GPL modules
Writing a kernel module

What are the differences between *userspace* and *kernel* development?

When writing a kernel module, there is nor a libc neither syscalls in order to simplify the development. Thus, some functions have been implemented in order to help kernel developers.

- All these functions are regrouped in the *zlib*
- These functions are general purpose functions
  1. *printk* is the replacement of printf
  2. *kmalloc* is the replacement of malloc
  3. There is no filesystem support, no accessible file or directory
- To these functions, all locally exported functions, like other drivers functions are also accessible. All these function are added to a *symbol table*. There is two symbol tables
  1. Symbols accessible only to GPL modules
  2. Symbols accessible to all modules
Developing out of the Linux kernel tree

- This allows a simple and fast implementation of a basic module structure
- The module Makfile is specific, using both standard and kernel-specific commands
- Doesn’t need to access all the kernel sources. Only headers are needed
- Doesn’t allow static module integration
Developing out of the Linux kernel tree

- This allows a simple and fast implementation of a basic module structure
- The module Makfile is specific, using both standard and kernel-specific commands
- Doesn’t need to access all the kernel sources. Only headers are needed
- Doesn’t allow static module integration
- A kernel module always have an entry point and an exit point
  - The entry point is used during the module loading
  - The exit point is used during the module unloading
Writing a kernel module

Developing out of the Linux kernel tree

- This allows a simple and fast implementation of a basic module structure
- The module Makfile is specific, using both standard and kernel-specific commands
- Doesn’t need to access all the kernel sources. Only headers are needed
- Doesn’t allow static module integration
- A kernel module always have an entry point and an exit point
  - The entry point is used during the module loading
  - The exit point is used during the module unloading
- These two functions are easily declared using specific Linux macros

```c
MODULE_INIT(mymodule_init)
MODULE_EXIT(mymodule_exit)
```
Developing in the Linux kernel tree

- The Linux production system need to be updated in order to take into account the new module. This means that Makefile and Kconfig files in the module directory need to be updated.
Developing in the Linux kernel tree

- The Linux production system need to be updated in order to take into account the new module. This means that Makefile and Kconfig files in the module directory need to be updated.
- Le The Makefile is used in order to define the associated module target and to define the associated files when the module is complex.
Developing in the Linux kernel tree

• The Linux production system need to be updated in order to take into account the new module. This means that Makefile and Kconfig files in the module directory need to be updated.
• The Makefile is used in order to define the associated module target and to define the associated files when the module is complex.
• The Kconfig file permit to add a new module configuration entry, specifying dependencies and manual.
The Linux kernel root Makefile is not describes. Its syntax is complex and difficult to apprehend.
The Linux kernel root Makefile is not describes. Its syntax is complex and difficult to apprehend.

All the other Makefiles contain lines like the following:

```make
obj-$(CONFIG_MYMODULE) += mymodule.o

mymodule-y := mymodule_submodule1.o mymodule_submodule2.o
```
Writing a kernel module

Add a new module to the production system

- The Kconfig files generate the configuration tree used by the configuration user interface
- Their syntax is quite simple:

```plaintext
config MYMODULE
  bool "mymodule support" if MYMODULE_DEPENDENCY
  depends on OTHERMODULE
  default n
  ---help---
  This module is a sample module explaining how the Linux system configuration works
```
Writing a procfs interface

- Permit to insert one or more files and directories in the procfs, in order to communicate with the user space
- Need approximatively 100 lines of code in order to propose a correct interface
- Such an interface need to be declared against the procfs manager, using the following functions (e.g.):
  - `create_proc_entry()` in order to ask for a new file creation in `/proc`
  - `proc_mkdir` and `proc_mkdir_mode` in order to create a new directory with specific access rights
- A proc entry must implement a read function, similar to char device driver implementations. Although, some standardized read encapsulation functions exist.
Proposing a new sysctl variable

- Adding a new interface to the sysctl sub-system is fast and easy to communicate with the userspace
- Though, such an interface does have some restrictions
Proposing a new sysctl variable

- Adding a new interface to the sysctl sub-system is fast and easy to communicate with the userspace
- Though, such an interface does have some restrictions
  - The interface is accessible through the `sysctl` command, which need admin rights
Proposing a new sysctl variable

- Adding a new interface to the sysctl sub-system is fast and easy to communicate with the userspace.
- Though, such an interface does have some restrictions:
  1. The interface is accessible through the `sysctl` command, which need admin rights.
  2. It is only possible to read or to set a simple variable (integer, string), not a complex structure.
Proposing a new sysctl variable

- Adding a new interface to the sysctl sub-system is fast and easy to communicate with the userspace
- Though, such an interface does have some restrictions
  1. The interface is accessible through the `sysctl` command, which need admin rights
  2. It is only possible to read or to set a simple variable (integer, string), not a complex structure.
- On the kernel side, implementing a new sysctl just need some lines of code
Proposing a new sysctl variable

- Adding a new interface to the sysctl sub-system is fast and easy to communicate with the userspace.
- Though, such an interface does have some restrictions:
  1. The interface is accessible through the `sysctl` command, which need admin rights.
  2. It is only possible to read or to set a simple variable (integer, string), not a complex structure.
- On the kernel side, implementing a new sysctl just need some lines of code.

To conclude:

Sysctl are a very simple and fast way to give back some information to the userspace. Thus their access may be uneasy. This is the case when implementing a user space application, which need to call the sysctl executable instead of a syscall.
Proposing a new sysctl variable
Proposing a new sysctl variable

```c
#include <linux/kernel.h>
#include <linux/sysctl.h>
```

Philippe Thierry (philippe.thierry@reseau-libre.net)
Proposing a new sysctl variable

```c
#include <linux/kernel.h>
#include <linux/sysctl.h>

int mymodule_var_name = 0;
```
Proposing a new sysctl variable

```c
#include <linux/kernel.h>
#include <linux/sysctl.h>

int mymodule_var_name = 0;

ctl_table my_sysctl_table[] = {
    {
        .ctl_name = CTL_MYMODULE,
        .procname = "my_module_variable",
        .data = &mymodule_var_name,
        .maxlen = sizeof(int),
        .mode = 0600,
        .proc_handler = &proc_dointvec,
    },
    { .ctl_name = 0 }
};
```
Proposing a new sysctl variable

```c
#include <linux/kernel.h>
#include <linux/sysctl.h>

int mymodule_var_name = 0;

ctl_table my_sysctl_table[] = {
  #ifdef CONFIG_MYMODULE
    { .ctl_name = CTL_MYMODULE,
      .procname = "my_module_variable",
      .data = &mymodule_var_name,
      .maxlen = sizeof(int),
      .mode = 0600,
      .proc_handler = &proc_dointvec,
    },
  #endif
  { .ctl_name = 0 }
};
```
Implementing a char device driver

- Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer.
- Such an interface needs to manage various basic functions:
  1. The `open` function, to create the necessary internal structures for the associated context
  2. The `release` function, to desallocate this context
  3. The `read` function, to read the module's internal data
  4. The `write` function, to write some data in the module
  5. The `ioctl` function, to exchange some control structures with the module
  6. The `poll` function, to manage asynchronous events

To conclude, using a char device permits to propose a rich and efficient interface, but needing a complex kernel-side implementation. This interface should be used only when easier interfaces are not rich enough for the needs.
Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer. Such an interface needs to manage various basic functions:

1. The `open` function, to create the necessary internal structures for the associated context
Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer.

Such an interface needs to manage various basic functions:

1. The `open` function, to create the necessary internal structures for the associated context
2. The `release` function, to desallocate this context
Implementing a char device driver

Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer. Such an interface needs to manage various basic functions:

1. The *open* function, to create the necessary internal structures for the associated context
2. The *release* function, to desallocate this context
3. The *read* function, to read the module’s internal data
Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer. Such an interface needs to manage various basic functions:

1. The `open` function, to create the necessary internal structures for the associated context
2. The `release` function, to desallocate this context
3. The `read` function, to read the module’s internal data
4. The `write` function, to write some data in the module
Implementing a char device driver

- Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer.
- Such an interface needs to manage various basic functions:
  1. The `open` function, to create the necessary internal structures for the associated context
  2. The `release` function, to desallocate this context
  3. The `read` function, to read the module’s internal data
  4. The `write` function, to write some data in the module
  5. The `ioctl` function, to exchange some control structures with the module
Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer. Such an interface needs to manage various basic functions:

1. The `open` function, to create the necessary internal structures for the associated context
2. The `release` function, to desallocate this context
3. The `read` function, to read the module’s internal data
4. The `write` function, to write some data in the module
5. The `ioctl` function, to exchange some control structures with the module
6. The `poll` function, to manage asynchronous events
Implementing a char device driver permits to propose a complex and rich interface, but is harder and longer.

Such an interface needs to manage various basic functions:

1. The `open` function, to create the necessary internal structures for the associated context
2. The `release` function, to desallocate this context
3. The `read` function, to read the module’s internal data
4. The `write` function, to write some data in the module
5. The `ioctl` function, to exchange some control structures with the module
6. The `poll` function, to manage asynchronous events

To conclude

Using a char device permits to propose a rich and efficient interface, but needing a complex kernel-side implementation. This interface should be used only when easier interfaces are not rich enough for the needs.
Implementing a char device driver

Let’s begin with the interface methods implementation...
Implementing a char device driver

Let’s begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode,
                  struct file *filep)
{ ... }
```
Implementing a char device driver

Let’s begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode,
                  struct file* filep)
{ ... }

int mymodule_release(void)
{ ... }
```
Implementing a char device driver

Let’s begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode,
                  struct file *filep)
{
    ...
}

int mymodule_release(void)
{
    ...
}

ssize_t mymodule_read(struct file *f,
                       char __user *buf,
                       size_t count,
                       loff_t *ppos)
{
    ...
}
```
Implementing a char device driver

Let’s begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode,
                   struct file* filep)
{
    ...}

int mymodule_release(void)
{
    ...}

ssize_t mymodule_read(struct file* f,
                       char __user* buf,
                       size_t count,
                       loff_t* ppos)
{
    ...}

ssize_t mymodule_write(struct file* f,
                        char __user* buf,
                        size_t count,
                        loff_t* ppos)
{
    ...}
```
Implementing a char device driver

Let's begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode, struct file *filep)
{
    ...
}

int mymodule_release(void)
{
    ...
}

ssize_t mymodule_read(struct file *f, char __user *buf, size_t count, loff_t *ppos)
{
    ...
}

ssize_t mymodule_write(struct file *f, char __user *buf, size_t count, loff_t *ppos)
{
    ...
}

uint32_t mymodule_poll(struct file *f, struct poll_table_struct *pts)
{
    ...
}
```
Implementing a char device driver

Let’s begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode, struct file *filep)
{
    ...
}

int mymodule_release(void)
{
    ...
}

ssize_t mymodule_read(struct file *f,
                      char __user *buf,
                      size_t count,
                      loff_t *ppos)
{
    ...
}

ssize_t mymodule_write(struct file *f,
                       char __user *buf,
                       size_t count,
                       loff_t *ppos)
{
    ...
}

uint32_t mymodule_poll(struct file *f,
                       struct poll_table_struct *pts)
{
    ...
}

int32_t mymodule_ioctl(struct inode *inode, struct file *filep, uint32_t cmd, unsigned long arg)
```
Implementing a char device driver

Let’s begin with the interface methods implementation...

```c
int mymodule_open(struct inode* inode,
                  struct file *filep)
{
    ...
}

int mymodule_release(void)
{
    ...
}

ssize_t mymodule_read(struct file *f,
                       char __user *buf,
                       size_t count,
                       loff_t *ppos)
{
    ...
}

ssize_t mymodule_write(struct file *f,
                        char __user *buf,
                        size_t count,
                        loff_t *ppos)
{
    ...
}

uint32_t mymodule_poll(struct file *f,
                       struct poll_table_struct *pts)
{
    ...
}

int32_t mymodule_ioctl(struct inode *inode,
                        struct file *filep,
                        uint32_t cmd,
                        unsigned long arg)
{
    ...
}

struct file_operations my_fops = {
    .owner = THIS_MODULE;
    .open = mymodule_open;
    .read = mymodule_read;
    .write = mymodule_write;
    .ioctl = mymodule_ioctl;
    .release = mymodule_release;
    .poll = mymodule_poll;
};
```
...which need to be declared with the new virtual device.
...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;
```
...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
}
```
...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
    /* register the char device (MAJOR and MINOR) */
    alloc_chrdev_region(&dev, 0, 1, "mymod");
    mymod_major = MAJOR(dev);
    cdev = cdev_alloc();
    devno = MKDEV(mymod_major, 0);
}
```

Philippe Thierry (philippe.thierry@reseau-libre.net) Linux kernel course 16 Nov 2010
...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
    /* register the char device (MAJOR and MINOR) */
    alloc_chrdev_region(&dev, 0, 1, "mymod");
    mymod_major = MAJOR(dev);
    cdev = cdev_alloc();
    devno = MKDEV(mymod_major, 0);
    /* initialize the corresponding cdev */
    cdev_init(cdev, &my_fops);
    cdev->owner = THIS_MODULE;
    cdev->ops = &my_fops;
    err = cdev_add(cdev, 0, 1);
}
```
Through a device

...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
    /* register the char device (MAJOR and MINOR) */
    alloc_chrdev_region(&dev, 0, 1, "mymod");
    mymod_major = MAJOR(dev);
cdev = cdev_alloc();
devno = MKDEV(mymod_major, 0);
    /* initialize the corresponding cdev */
cdev_init(cdev, &my_fops);
cdev->owner = THIS_MODULE;
cdev->ops = &my_fops;
err = cdev_add(cdev, 0, 1);
}
```
...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
    /* register the char device (MAJOR and MINOR) */
    alloc_chrdev_region(&dev, 0, 1, "mymod");
    mymod_major = MAJOR(dev);
    cdev = cdev_alloc();
    devno = MKDEV(mymod_major, 0);
    /* initialize the corresponding cdev */
    cdev_init(cdev, &my_fops);
    cdev->owner = THIS_MODULE;
    cdev->ops = &my_fops;
    err = cdev_add(cdev, 0, 1);
}
```
...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
    /* register the char device (MAJOR and MINOR) */
    alloc_chrdev_region(&dev, 0, 1, "mymod");
    mymod_major = MAJOR(dev);
cdev = cdev_alloc();
devno = MKDEV(mymod_major, 0);
    /* initialize the corresponding cdev */
cdev_init(cdev, &my_fops);
cdev->owner = THIS_MODULE;
cdev->ops = &my_fops;
err = cdev_add(cdev, 0, 1);
}

int module_release(void)
{
    /* deregistering device and desallocating memory.... */
    ...
}
```
Through a device

...which need to be declared with the new virtual device.

```c
int mymod_major = 0;
dev_t dev;
struct cdev *cdev = NULL;

int mymodule_init(void)
{
    /* register the char device (MAJOR and MINOR) */
    alloc_chrdev_region(&dev, 0, 1, "mymod");
    mymod_major = MAJOR(dev);
    cdev = cdev_alloc();
    devno = MKDEV(mymod_major, 0);
    /* initialize the corresponding cdev */
    cdev_init(cdev, &my_fops);
    cdev->owner = THIS_MODULE;
    cdev->ops = &my_fops;
    err = cdev_add(cdev, 0, 1);
}

int module_release(void)
{
    /* deregistering device and desallocating memory.... */
    ...
}

MODULE_INIT(mymodule_init)
MODULE_EXIT(mymodule_release)
```
Add a new syscall to Linux

- Linux supports approximately 250 syscalls, which is a rich and complex interface.
- During the execution, a syscall is shown like the software interrupt 80 (int 80).
- Depending on the registers content, the specific syscall implementation is selected in the syscall vector.
- Adding a new syscall is done in the arch subsystem (e.g. arch/x86/kernel/syscall_table_32.S file).
Add a new syscall to Linux

- As a consequence, adding a new syscall is difficult

- Adding a new syscall doesn’t follow any specific norm, which generate some difficulties when using existing user space softwares
Add a new syscall to Linux

- As a consequence, adding a new syscall is difficult
  - The implementation is not natively portable to another architecture

- Adding a new syscall doesn’t follow any specific norm, which generate some difficulties when using existing user space softwares
Add a new syscall to Linux

- As a consequence, adding a new syscall is difficult
  - The implementation is not natively portable to another architecture
  - Adding new syscalls to the syscall vector may generate conflicts with other modules having the same behaviour
- Adding a new syscall doesn’t follow any specific norm, which generate some difficulties when using existing user space softwares
Through new syscalls

Add a new syscall to Linux

- As a consequence, adding a new syscall is difficult
  - The implementation is not natively portable to another architecture
  - Adding new syscalls to the syscall vector may generate conflicts with other modules having the same behaviour
- Adding a new syscall doesn’t follow any specific norm, which generate some difficulties when using existing user space softwares
Add a new syscall to Linux

- As a consequence, adding a new syscall is difficult
  - The implementation is not natively portable to another architecture
  - Adding new syscalls to the syscall vector may generate conflicts with other modules having the same behaviour
- Adding a new syscall doesn’t follow any specific norm, which generate some difficulties when using existing user space softwares

To conclude

Using new syscall should be done only for specific modules or when no other interface can be used. You should not have to use such an interface for your own modules.
The Linux kernel is a very big kernel, with a lot of modules, drivers, core elements.

All these parts communicates between them and with the upper layer.

Depending on the studied modules, some will use only procfs interfaces, some will add one or more virtual char devices.

Some will also add some specific sysctl to simplify their management.

Do not hesitate to use these slides when reading some kernel code you will study.
  This book can be downloaded and used under the CreativeCommons license: http://lwn.net/Kernel/LDD3/

• The Linux community, *The Linux kernel 2.6 documentation*
  The official kernel documentation, accessible in the Documentation subdirectory of the kernel sources
Questions
Index

/bin, 15–19
/boot, 20–23
/etc, 15–19
/home, 15–19
/lib, 15–19
/root, 20–23
/sbin, 15–19
/tmp, 20–23
/usr, 20–23
/var, 15–19
directories: dev, 24–27
directories: proc, 24–27
directories: sys, 24–27
interrupt handlers, 5–13
memory, 5–13
POSIX, 4
processor, 5–13
scheduler, 5–13
UNIX, 15–23