



Aricent®

Engineering excellence. **Sourced.**

Linux Device Driver Basics

By

-HEMAKUMAR

# Session-1

## Introduction to LDD

## What to Expect?

- ★ After this session, you would know
  - W's of Linux Drivers
  - Ecosystem of Linux Drivers
  - Types of Linux Drivers
  - Vertical & Horizontal Driver Layering
  - Various Terminologies in vogue
  - Linux Driver related Commands & Configs
  - Using a Linux Driver
  - Our First Linux Driver

# W's of Linux Drivers

- ★ What is a Driver?
- ★ What is a Linux Driver?
- ★ Is Linux Device Driver = Linux Driver?
- ★ Why we need a Driver?
- ★ What are the roles of Linux Driver?

# Device driver basics.

What is a Device driver?

A set of routines which makes an applications to access the HW device functionality on request.

Role of a Device Drivers:

1. Interaction with Applications
2. Interaction with Devices.

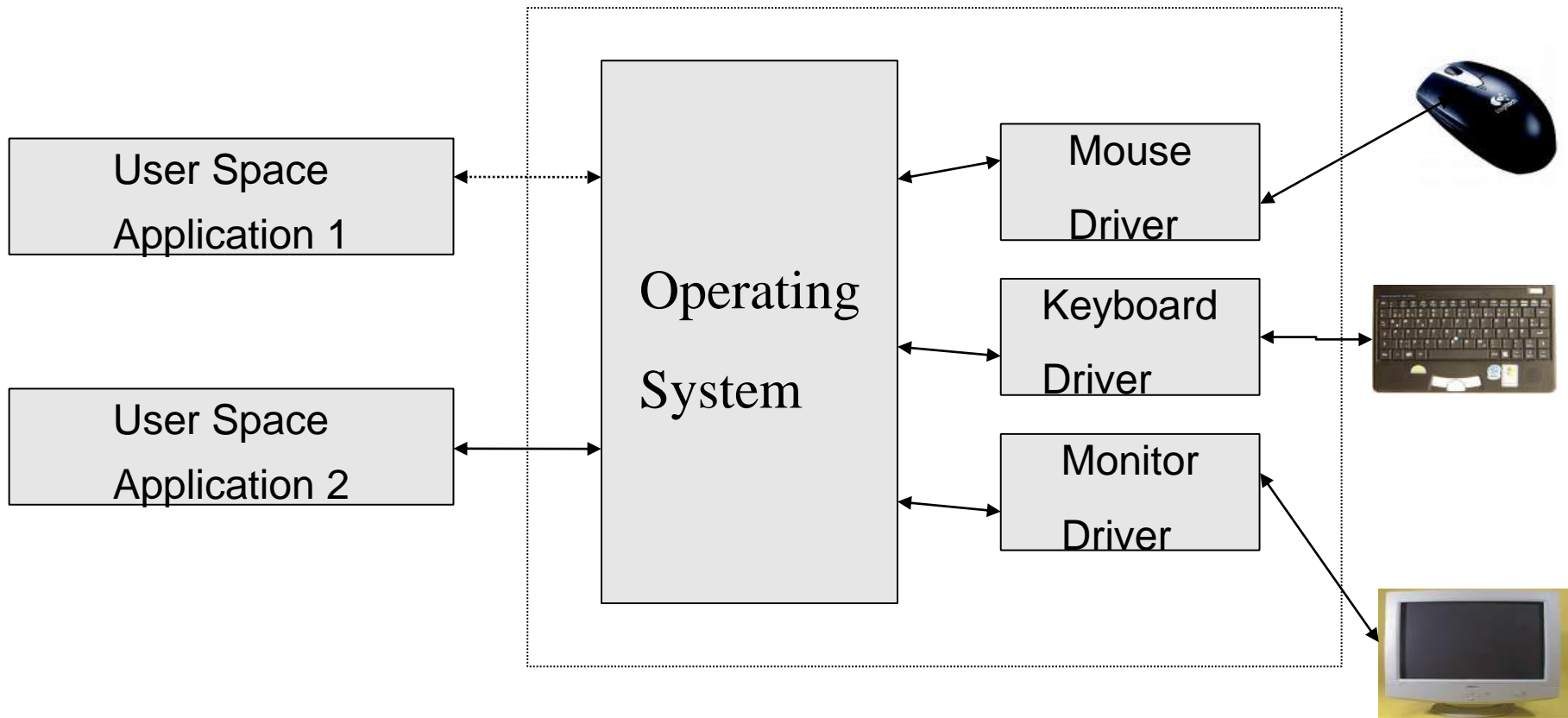
Interaction with applications is OS specific.

Interaction with HW devices is Bus specific.

# Device driver basics..

- Linux device drivers means interaction between Linux applications and Linux drivers.
- Windows device drivers means, interaction between windows apps and windows drivers.
- When we say Device drivers, it means routines for HW interaction without OS (we usually call it as board specific drivers).
- Driver interaction with the HW device will be same for Linux drivers and windows drivers.
- Driver interaction with the applications is OS specific and which can change from OS to OS. In this way Linux drivers and window drivers are different.

# Device Drivers Basics...



- A simple black box that allows a user space application interaction with a hardware, without need for the knowledge of the hardware functioning

# Device driver basics...

## Classification of Linux Device Drivers

Linux device drivers are classified based on how the applications are reachable to the devices.

- Character Devices

A character device can be accessed as a stream of bytes. Applications will reach through the file system to access a device functionality through drivers.(ex: key boards, video cards etc)

- Block Devices

A block device can be accessed as blocks or group of blocks(usually 512 bytes blocks). The transfer of data is done in blocks of varying lengths. (ex: Hard disks, SCSI, all other storage devices)

- Network Devices

Applications will reach through the Network protocol stacks.(ex: Ethernet drivers, and other network controllers). Apps to access the Ethernet drivers should reach through TCP/IP protocol stack.



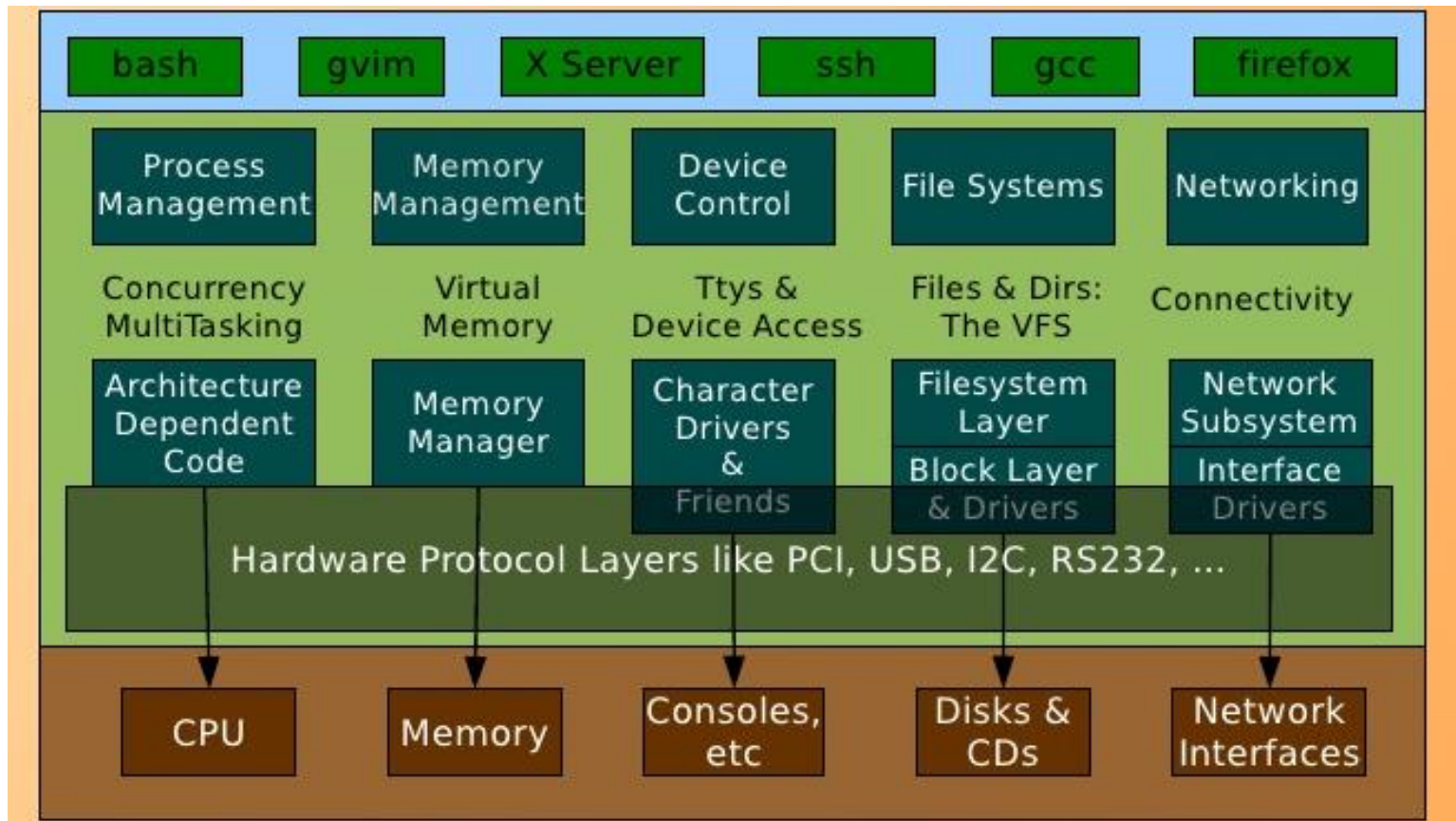
# Functions of an OS

- ✧ Process / Time / Processor Management
- ✧ Memory Management
- ✧ Device I/O Management
- ✧ Storage Management
- ✧ Network Management

## Linux as an OS

- ✧ So, Linux also has the same structure
- ✧ Visually, can be shown as

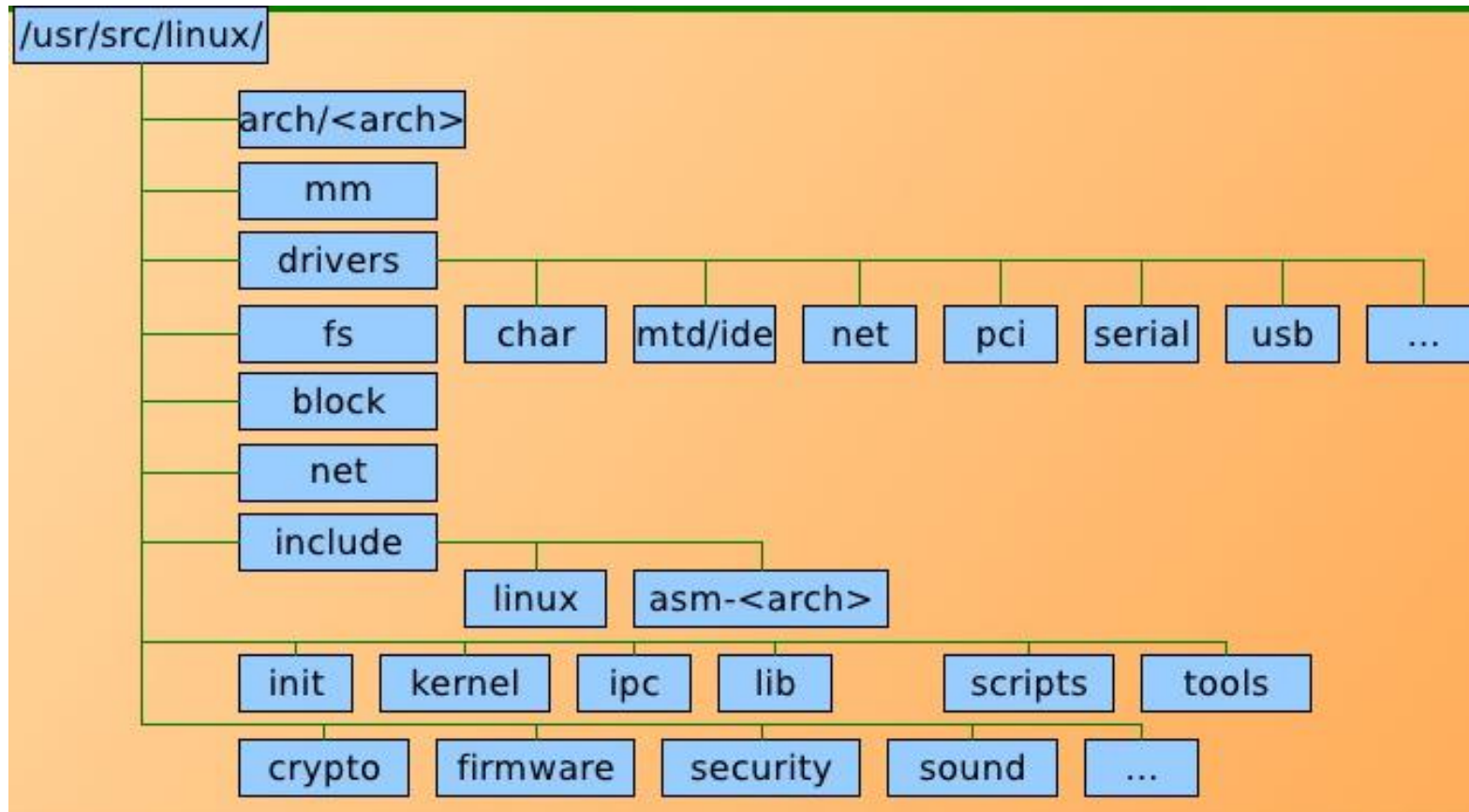
# Linux Driver Ecosystem



# Kernel Architecture overview..

- Kernel Services layers
- Process management: Process creation, destruction, and IPC, process switching, management of process control blocks
- Memory management: managing page tables, memory allocations, address translations, swapping etc.
- Device drivers: Managing interactions between applications and devices.
- Hot plugging subsystem: management of hot plugging devices like USB etc.
- File system layer: Managing kernel housing, virtual file systems etc.
- Network subsystem: Networking and protocol stack management.
- IO subsystem: IO management.

# Kernel Source Organization



# Kernel module programming support

- Linux kernel is designed to support module programming.
- Module programming is a technique offered by the linux kernel where we can implement, build our kernel service/module, and append to the existing kernel code dynamically.
- It provides an abstraction to the existing kernel code , no need to bother how the existing kernel written.
- Kernel module interface provides below tools to append the our kernel modules.

Insmod: To insert a module

Ismod: To list our module

rmmod: To remove the module.

# Rules for writing kernel modules..

- Every module should define its own initialization and exit routines. The function names can be our choice, but signature must follow.

Ex: Init routine: `int init_mod(void);`

Exit routine: `void cleanup_mod(void);`

The below macros are used to register the init and exit routines of my module with Linux kernel.

`module_init(init_mod);`

`module_exit(cleanup_mod);`

- Every module should provide a set of comments with details of author, description and Licensing.



# W's of a Module?

- ✧ Hot plug-n-play Driver
- ✧ Dynamically Loadable & Unloadable
- ✧ Linux – the first OS to have such a feature
- ✧ Later many followed suit
- ✧ Enables fast development cycle
- ✧ File: <module>.ko (Kernel Object)
  - <module>.o wrapped with kernel signature
- ✧ Std Modules Path
  - /lib/modules/<kernel version>/kernel/...
- ✧ Module Configuration: /etc/modprobe.conf



# Rules for writing kernel modules

- Should not use/invoke user space routines(C library, API's IPC's , system calls etc.)
- Should use GNU – C extensions.
- Modules should include header files from kernel header directories only. Kernel header dir:  
**`/lib/modules/'uname -r'/build/include`**
- All modules must include the below header files  
`#include <linux/module.h>`  
`#include <linux/version.h>`  
`#include <linux/kernel.h>`  
`#include <linux/init.h>`

## The First Linux Driver

## The Module Constructor

```
static int __init mfd_init(void)
{
    ...

    return 0;
}
module_init(mfd_init);
```

# The Module Destructor

```
static void __exit mfd_exit(void)
{
    ...
}
module_exit(mfd_exit);
```

# printk – Kernel's printf

- ★ Header: `<linux/kernel.h>`
- ★ Arguments: Same as `printf`
- ★ Format Specifiers: All as in `printf`, except float & double related
- ★ Additionally, a initial 3 character sequence for Log Level
  - `KERN_EMERG`    "`<0>`"    */\* system is unusable \*/*
  - `KERN_ALERT`    "`<1>`"    */\* action must be taken immediately \*/*
  - `KERN_CRIT`      "`<2>`"    */\* critical conditions \*/*
  - `KERN_ERR`       "`<3>`"    */\* error conditions \*/*
  - `KERN_WARNING`   "`<4>`"    */\* warning conditions \*/*
  - `KERN_NOTICE`    "`<5>`"    */\* normal but significant condition \*/*
  - `KERN_INFO`       "`<6>`"    */\* informational \*/*
  - `KERN_DEBUG`     "`<7>`"    */\* debug-level messages \*/*

## The Module Constructor (revisited)

```
static int __init mfd_init(void)
{
    printk(KERN_INFO "mfd registered");
    ...
    return 0;
}
module_init(mfd_init);
```

## The Module Destructor (revisited)

```
static void __exit mfd_exit(void)
{
    printk(KERN_INFO "mfd deregistered");
    ...
}
module_exit(mfd_exit);
```

# The Other Basics & Ornaments

## • Basic Headers

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

• `MODULE_LICENSE("GPL");`

• `MODULE_AUTHOR("WRITE YOUR NAME");`

• `MODULE_DESCRIPTION("First Device Driver");`



# Basic Kernel module Example

## • Simple module

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

```
Static int val = 0;
Module_param(val, int, S_IRUGO);
MODULE_PARAM_DESC(val, "INITIALIZE AT INSERTION TIME");
Void func(void);
```

```
Void func(void)
{
    printk("func invoked \n");
    printk("val = %d", val);
}
Int mymod_init(void)
{
    printk("Module Inserted\n");
    func();
}
```

# Basic Kernel module Example..

```
Void mymod_exit(void)
{
    printk("Module Removed \n");
}

/* register init and cleanup routine with kernel */
Module_init(mymod_init);
Module_exit(mymod_exit);

/*export my module function to global kernel symbol table */
EXPORT_SYMBOL_GPL(func);

/*kernel module comments*/
MODULE_AUTHER("MY TEAM");
MODULE_DESCRIPTION("TEST MODULE");
MODULE_LICENSE("GPL");
```

# Building a module

- ✧ For building our driver, it needs
  - The Kernel Headers for Prototypes
  - The Kernel Functions for Functionality
  - The Kernel Build System & the Makefile for Building
- ✧ Two options to Achieve
  - 1. Building under Kernel Source Tree
    - Put our driver appropriately under drivers folder
    - Edit corresponding Kconfig(s) & Makefile to include our driver
  - 2. Create our own Makefile to do the right invocation

# Building a module..

- To build our module we need to write a makefile(k-build make files) which internally depends on kernel makefile.
- Go to your kernel module source code directory and simply create the Makefile file as follows:

```
$ vi Makefile
```

- Add following text to it:

```
obj-m = mymod.o
```

```
KVERSION = $(shell uname -r)
```

```
all:
```

```
make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules
```

```
clean:
```

```
make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean 4)
```

# Building a module...

- Compile module using make command (module build can be done by any user) :

**\$ make**

- Once module compiled successfully, load it using insmod command. You need to be root user or privileged user to run insmod:

**\$ su**

**#insmod mymod.ko**

- Verify that module loaded:

**# lsmod | less**

- You can see the log messages using 'dmesg'.

**#dmesg**

Module Inserted

func invoked

val = 0

## Make File Example-2

```
ifeq (${KERNELRELEASE},)
    KERNEL_SOURCE := <kernel source directory path>
    PWD := $(shell pwd)
    default:
        $(MAKE) -C ${KERNEL_SOURCE} SUBDIRS=$(PWD) modules
    clean:
        $(MAKE) -C ${KERNEL_SOURCE} SUBDIRS=$(PWD) clean
else
    obj-m += <module>.o
endif
```

# Module Commands

- ★ Typically needs root permission
- ★ Resides in /sbin
- ★ Operates over the kernel-module i/f
- ★ Foundation of Driver Development
- ★ Need to understand thoroughly

# Listing Modules

- ✧ Command: `lsmod`
- ✧ Fields: Module, Size, Used By
- ✧ Kernel Window: `/proc/modules`
- ✧ Are these listed modules static or dynamic?



## Loading Modules

- \* Command: `insmod <module_file>`
- \* Go to modules directory and into fs/vfat
- \* Try: `insmod vfat.ko`

# Unloading Modules

- ★ Command: `rmmod <module_name>`
- ★ Try: `rmmod fat`

## Auto Loading Modules

- ✧ Command: `modprobe <module_name>`
- ✧ Try: `modprobe vfat`

# Kernel Windows

- ★ Through virtual filesystems

- /proc
- /sys

Command: `cat <window_file>`

- ★ System Logs: /var/log/messages

Command:

- `tail /var/log/messages`
- `dmesg | tail`

## Other Useful Commands

- ✧ Disassemble: `objdump -d <object_file>`
- ✧ List symbols: `nm <object_file>`

# Command Summary

- ★ lsmod
- ★ insmod
- ★ modprobe
- ★ rmmod
- ★ dmesg
- ★ objdump
- ★ nm

Try out your First Linux Driver

# What all have we learnt?

- ★ W's of Linux Drivers
- ★ Ecosystem of Linux Drivers
- ★ Types of Linux Drivers
- ★ Vertical & Horizontal Driver Layering
- ★ Various Terminologies in vogue
- ★ Linux Driver related Commands & Configs
- ★ Using a Linux Driver
- ★ Our First Linux Driver



Any Queries?

# Additional Contents of Session-1(Optional)

- Kernel code
- Kernel & User interface
- Compiling Linux
  - Linux kernel sources
  - Kernel configuration
  - Compiling the kernel

# System Calls

**U-Space programs will access the K-space routines using system call interface.**

## User Space steps while executing system call

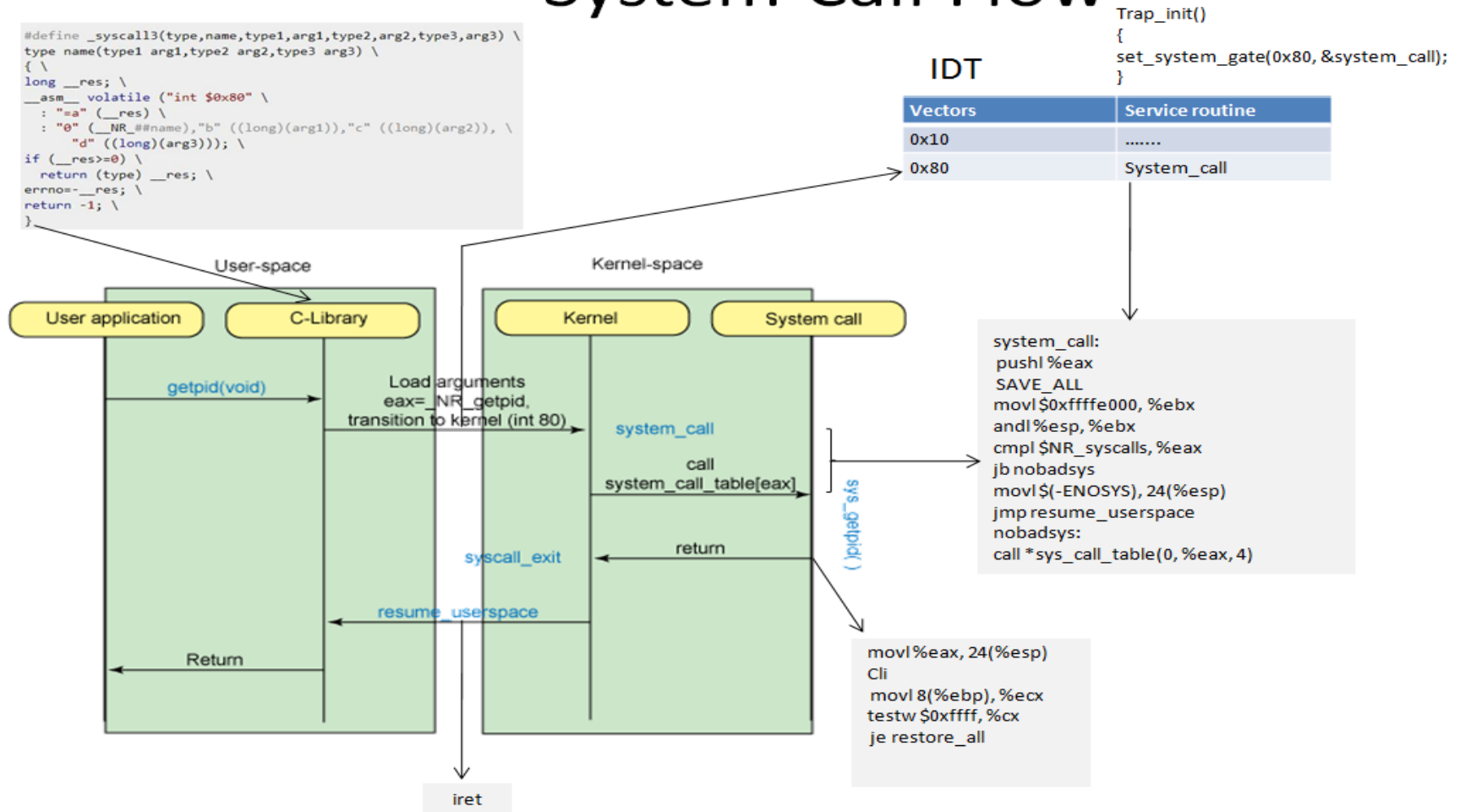
- the C compiler uses a predefined library (the library) that have the names of the system calls, thus resolving the system call reference in the user program.
- The library functions pass the kernel a unique number per system call in a machine-depended way- either as a parameter to operating system trap in a particular register, or on the stack. thus determines the specific system call the user is invoking.
- The library functions pass system call parameters in the CPU registers before issuing the system call.
- The library function invoke an special instruction that changes the process execution user mode to kernel.(int 0x80 or sysenter)

## Kernel space steps while executing sytem call:

- The kernel then copies the parameters stored in the CPU registers onto the Kernel Mode stack before invoking the system call service routine,
- Kernel search respective system call service routine for a unique number supplied from user mode and execute the system call service.
- When the system call service routine terminates, the `system_call()` function gets its return code from `eax` and stores it in the stack location where the User Mode value of the `eax` register is saved.
- Kernel executes an `iret` assembly language instruction to resume the User Mode process

# System Calls..

## System Call Flow



# How to get the Linux source and compile

Compiling and booting Linux &  
Linux kernel sources

2.4->2.6->3.0->4.1->...->4.6

# Linux Distributions.

- Red Hat Linux : One of the original Linux distribution. The commercial, nonfree version is Red Hat Enterprise Linux, which is aimed at big companies using Linux servers and desktops in a big way.
- Debian GNU/Linux : A free software distribution. Popular for use on servers. However, Debian is not what many would consider a distribution for beginners, as it's not designed with ease of use in mind.
- Ubuntu: based on Debian Linux. Ubuntu claims to be most popular desktop version. Many applications and excellent “update mechanism” contribute to its success. Revenue is created by selling technical support.
- SuSE Linux : SuSE was recently purchased by Novell. This distribution is primarily available for pay because it contains many commercial programs, although there's a stripped-down free version that you can download.
- Mandrake Linux : Mandrake is perhaps strongest on the desktop. Originally based off of Red Hat Linux.
- Gentoo Linux : Gentoo is a specialty distribution meant for programmers.

# Linux Packaging Distributions..

- There is no standard package manager in Linux
- Packages Distributed in Binaries or Source Code form
- Main Package Management Standards
  - RPM (RedHat Package Manager) (.rpm)
    - Introduced by RedHat and has been adopted by many other distributions (Fedora, Mandrake, SuSe) .
    - The most popular Linux package format
  - DEB (Debian Package Manager) (.deb)
    - Introduced by Debian distribution
  - Tarball files (.tar.gz/.tar.bz2)
    - The old-fashioned way of distributing software in Linux/Unix
    - Compatible with all distros

# Redhat Packaging Manager..

Using the command line, packages are installed using rpm utility program

- Install a package

```
$ rpm -i <package_name>.rpm
```

- Update an existing package

```
$ rpm -U <package_name>.rpm
```

- Remove a package

```
$ rpm -e <package_name>
```



# Debian Based ...

## Three ways to manage software packages in Debian

1. **dpkg:** Used on .deb files like rpm

**Install:** `$ dpkg -i <package_name>.deb`

-If an older version of the package is installed it updates it automatically by replacing it with the new

**Remove:** `$ dpkg -r <package_name>`

2. **dselect:** dpkg console front-end

3. **apt-get:** The most frequently used way of managing software packages in Debian.

**Install:** `$ apt-get install <package_name>`

e.g. `apt-get install kde` to install KDE Window Manager

**Remove:** `$ apt-get remove <package_name>`

# Tarball Based ...

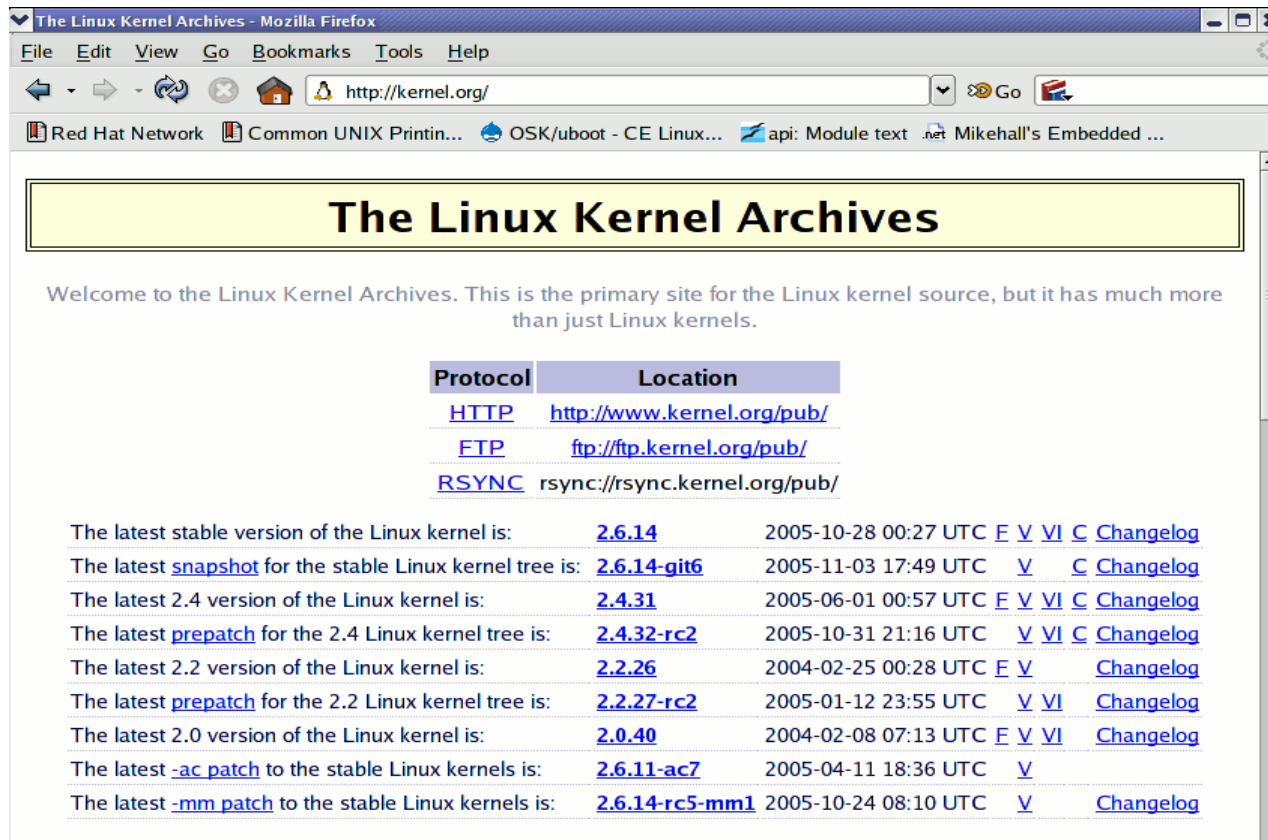
1. Compatible with all Linux distributions
2. Contains a bunch of files of the application, packed in a .tar archive and compressed using GNU Zip (.gz) or BZip2 (.bz2).  
Format : <filename>.tar.gz or <filename>.tar.bz2
3. Can be unzipped and unpacked on a directory using the tar command:
  1. tar xvzf <filename>.tar.gz
  2. tar xvjf <filename>.tar.bz2
4. “INSTALL” or “README” files are also exist in this directory giving application-specific usage information

# Tarball Based ...

- Software Packages coming in source code archives have to be compiled before installed
- Usually come in .tar.gz/.tar.bz2 archives
- Typical compilation/installation steps
  - Unpack the archive:
    - `tar xzvf <package_name>.tar.gz`
    - `tar xvjf <package_name>.tar.bz2`
  - Change to the extracted directory
    - `cd <extracted_dir_name>`
  - Run source configuration script as follows:
    - `./configure`
  - Build the source code using the GNU Make utility as follows:
    - `make`
  - Install the package as follows:
    - `make install`

# Getting Kernel sources (Old)

- Make sure you have installed the Linux on your system(for example Ubuntu 10.10).
- Download the latest kernel source from <http://kernel.org>.



The screenshot shows a Mozilla Firefox browser window titled "The Linux Kernel Archives - Mozilla Firefox". The address bar displays "http://kernel.org/". The browser's menu bar includes File, Edit, View, Go, Bookmarks, Tools, and Help. The page content features a yellow header with the title "The Linux Kernel Archives". Below the header, a welcome message states: "Welcome to the Linux Kernel Archives. This is the primary site for the Linux kernel source, but it has much more than just Linux kernels." A table provides download information:

Protocol	Location
<a href="http://www.kernel.org/pub/">HTTP</a>	<a href="http://www.kernel.org/pub/">http://www.kernel.org/pub/</a>
<a href="ftp://ftp.kernel.org/pub/">FTP</a>	<a href="ftp://ftp.kernel.org/pub/">ftp://ftp.kernel.org/pub/</a>
<a href="rsync://rsync.kernel.org/pub/">RSYNC</a>	<a href="rsync://rsync.kernel.org/pub/">rsync://rsync.kernel.org/pub/</a>

Below the table, a list of kernel versions and their release dates is provided, each with links for further details:

- The latest stable version of the Linux kernel is: [2.6.14](#) 2005-10-28 00:27 UTC [F](#) [V](#) [VI](#) [C](#) [Changelog](#)
- The latest [snapshot](#) for the stable Linux kernel tree is: [2.6.14-git6](#) 2005-11-03 17:49 UTC [V](#) [C](#) [Changelog](#)
- The latest 2.4 version of the Linux kernel is: [2.4.31](#) 2005-06-01 00:57 UTC [F](#) [V](#) [VI](#) [C](#) [Changelog](#)
- The latest [prepatch](#) for the 2.4 Linux kernel tree is: [2.4.32-rc2](#) 2005-10-31 21:16 UTC [V](#) [VI](#) [C](#) [Changelog](#)
- The latest 2.2 version of the Linux kernel is: [2.2.26](#) 2004-02-25 00:28 UTC [F](#) [V](#) [C](#) [Changelog](#)
- The latest [prepatch](#) for the 2.2 Linux kernel tree is: [2.2.27-rc2](#) 2005-01-12 23:55 UTC [V](#) [VI](#) [C](#) [Changelog](#)
- The latest 2.0 version of the Linux kernel is: [2.0.40](#) 2004-02-08 07:13 UTC [F](#) [V](#) [VI](#) [C](#) [Changelog](#)
- The latest [-ac patch](#) to the stable Linux kernels is: [2.6.11-ac7](#) 2005-04-11 18:36 UTC [V](#) [C](#) [Changelog](#)
- The latest [-mm patch](#) to the stable Linux kernels is: [2.6.14-rc5-mm1](#) 2005-10-24 08:10 UTC [V](#) [C](#) [Changelog](#)

# Kernel source structure

arch/<arch>            Architecture specific code  
arch/<arch>/mach-<mach>    Machine / board specific code  
crypto/                Cryptographic libraries  
Documentation/        Kernel documentation.  
drivers/               All drivers (usb, pci...)  
fs/                    Filesystems (fs/ext3/, etc.)  
include/               Kernel headers  
include/asm-<arch>    Architecture dependent headers  
include/linux          Linux kernel core headers  
init/                  Linux initialization (including main.c)  
ipc/                   Code used for process communication

# Kernel source structure..

kernel/	Linux kernel core (very small!)
lib/	Misc library routines (zlib, crc32...) MAINTAINERS
	Maintainers of each kernel part
Makefile	Top Linux makefile (sets arch and version)
mm/	Memory management code (small too!)
net/	Network support code (not drivers)
README	Overview and building instructions
REPORTING-BUGS	Bug report instructions
scripts/	Scripts for internal or external use
security/	Security implementations (SELinux...)
sound/	Sound support code and drivers
usr/	Code to generate an initramfs archive.

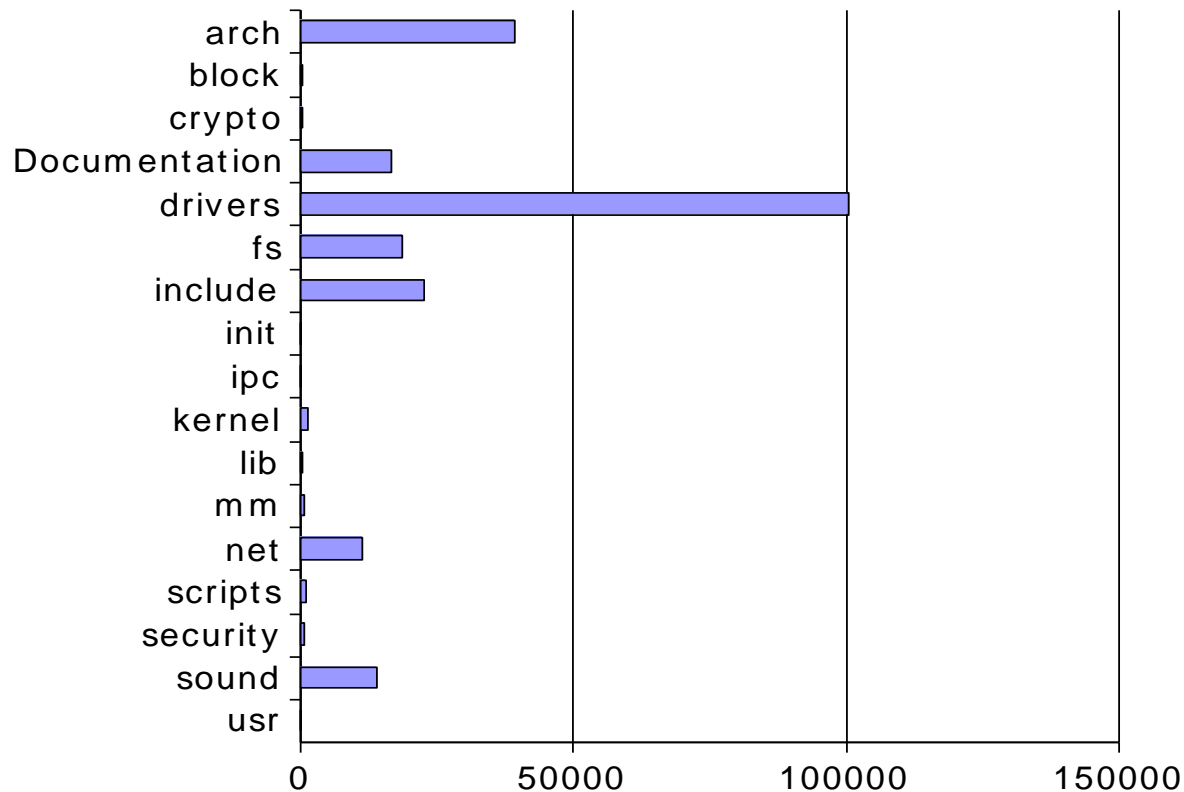
# Linux Kernel size

- Linux 2.6.17 sources:
  - Raw size: 224 MB (20400 files, approx 7 million lines of code)
  - gzip compressed tar archive: 50 MB
  - bzip2 compressed tar archive: 40 MB (better)
  - 7zip compressed tar archive: 33 MB (best)
- Minimum compiled Linux kernel size (with Linux-Tiny patches)  
approx 300 KB (compressed), 800 KB (raw)
- Why are these sources so big?  
Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...
- The Linux core (scheduler, bootup tasks, arch dependent code...) is pretty small!

# Linux Kernel size..

Linux 2.6.17  
Measured with:  
`du -s --apparent-size`

Size of Linux source directories (KB)





# Getting Linux sources: 2 possibilities

## Full sources

- The easiest way, but longer to download.

- Example:

<http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.36.2.tar.bz2>

## Or patch against the previous version

- Assuming you already have the full sources of the previous version

- Example:

<http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.36.bz2> (2.6.35 to 2.6.36)

<http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.36.2.bz2> (2.6.36 to 2.6.36.2)

# Procedure for Kernel download and compilation

## 1 Preliminary Note

I prefer to do all the steps here as the `root` user. So if you haven't already created a root login, you should do so now:

```
sudo passwd root
```

Afterwards, log in as root:

```
su
```

If you would like to work as a normal user instead of root, remember to put `sudo` in front of all the commands shown in this tutorial. So when I run

```
apt-get update
```

you should run

```
sudo apt-get update
```

instead, etc.

# Procedure for Kernel download and compilation..

## 2 Install Required Packages For Kernel Compilation

First we update our package database:

```
apt-get update
```

Then we install all needed packages like this:

```
apt-get install kernel-package libncurses5-dev fakeroot wget bzip2
```

## 3 Download The Kernel Sources

Next we download our desired kernel to `/usr/src`. Go to [www.kernel.org](http://www.kernel.org) and select the kernel you want to install, e.g. `linux-2.6.18.1.tar.bz2` (you can find all 2.6 kernels here: <http://www.kernel.org/pub/linux/kernel/v2.6/>). Then you can download it to `/usr/src` like this:

```
cd /usr/src
wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.1.tar.bz2
```

Then we unpack the kernel sources and create a symlink `linux` to the kernel sources directory:

```
tar xjf linux-2.6.18.1.tar.bz2
ln -s linux-2.6.18.1 linux
cd /usr/src/linux
```

# Kernel compilation..

## 4. Configure The Kernel

- It's a good idea to use the configuration of your current working kernel as a basis for your new kernel. Therefore we copy the existing configuration to /usr/src/linux:
- **cp /boot/config-`uname -r` ./config**
- Then we run
- **make menuconfig**
- which brings up the kernel configuration menu. Go to Load an Alternate Configuration File and choose .config (which contains the configuration of your current working kernel) as the configuration file:

## 6 Build The Kernel

To build the kernel, execute these two commands:

```
make-kpkg clean  
fakeroot make-kpkg --initrd --append-to-version=-custom kernel_image kernel_headers
```

After `--append-to-version=` you can write any string that helps you identify the kernel, but it must begin with a minus (-) and must not contain whitespace.

Now be patient, the kernel compilation can take some hours, depending on your kernel configuration and your processor speed.

## 7 Install The New Kernel

After the successful kernel build, you can find two .deb packages in the `/usr/src` directory.

```
cd /usr/src  
ls -l
```

On my test system they were called `linux-image-2.6.18.1-custom_2.6.18.1-custom-10.00.Custom_i386.deb` (which contains the actual kernel) and `linux-headers-2.6.18.1-custom_2.6.18.1-custom-10.00.Custom_i386.deb` (which contains files needed if you want to compile additional kernel modules later on). I install them like this:

```
dpkg -i linux-image-2.6.18.1-custom_2.6.18.1-custom-10.00.Custom_i386.deb  
dpkg -i linux-headers-2.6.18.1-custom_2.6.18.1-custom-10.00.Custom_i386.deb
```

(You can now even transfer the two .deb files to other Ubuntu systems and install them there exactly the same way, which means you don't have to compile the kernel there again.)

# Kernel compilation....

That's it. You can check `/boot/grub/menu.lst` now, you should find two stanzas for your new kernel there:

```
vi /boot/grub/menu.lst
```

The stanzas that were added on my test system look like these:

```
title          Ubuntu, kernel 2.6.18.1-custom
root           (hd0,0)
kernel         /boot/vmlinuz-2.6.18.1-custom root=
initrd         /boot/initrd.img-2.6.18.1-custom
savedefault
boot

title          Ubuntu, kernel 2.6.18.1-custom (rec
root           (hd0,0)
kernel         /boot/vmlinuz-2.6.18.1-custom root=
initrd         /boot/initrd.img-2.6.18.1-custom
boot
```

Now reboot the system:

```
shutdown -r now
```

If everything goes well, it should come up with the new kernel. You can check if it's really using your new kernel by running

```
uname -r
```

# Downloading kernel source patches

Assuming you already have the linux-x.y.<n-1> version

- Identify the patches you need on <http://kernel.org> with a web browser
- Download the patch files and their signature:
- Check the electronic signature of the archive:

```
gpg --verify linux-2.6.11.12.tar.bz2.sign
```

Example:

- Patch from 2.6.10 to 2.6.11  
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.bz2  
wget <ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.bz2.sign>
- Patch from 2.6.11 to 2.6.11.12 (latest stable fixes)  
wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.12.bz2  
wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.12.bz2.sign

# Apply kernel source patches

- Check the signature of patch files:

```
gpg --verify patch-2.6.11.bz2.sign
```

```
gpg --verify patch-2.6.11.12.bz2.sign
```

- Apply the patches in the right order:

```
cd linux-2.6.10/
```

```
bzcat ../patch-2.6.11.bz2 | patch -p1  
bzcat ../patch-2.6.11.12.bz2 | patch -p1
```

```
cd ..
```

```
mv linux-2.6.10 linux-2.6.11.12
```



# Linux kernel licensing

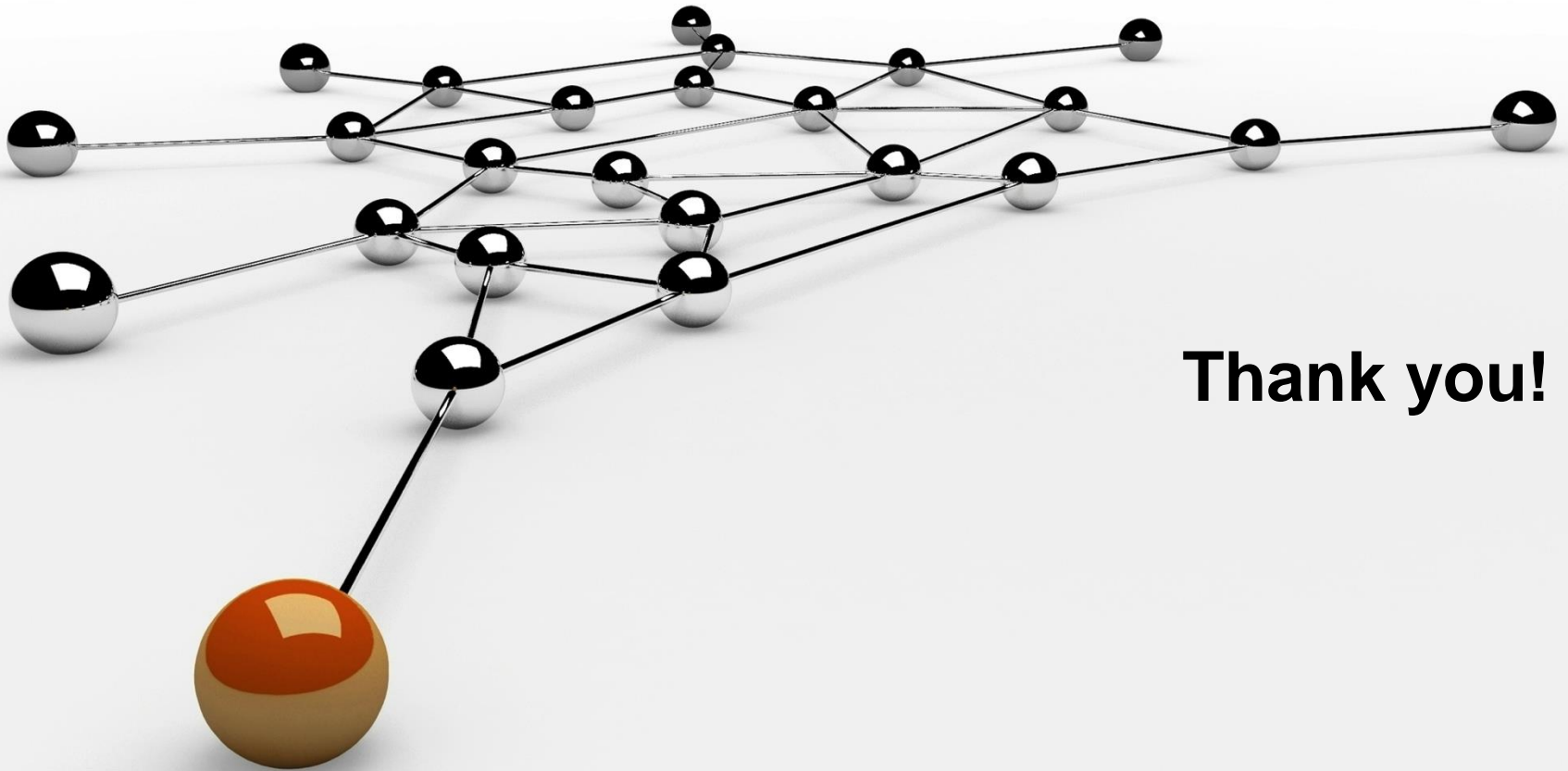
- You don't have to write your driver from scratch. You can reuse code from similar free software drivers.
- You get free community contributions, support, code review and testing. Proprietary drivers (even with sources) don't get any.
- Your drivers can be freely shipped by others (mainly by distributions).
- Closed source drivers often support a given kernel version. A system with closed source drivers from 2 different sources is unmanageable.
- Users and the community get a positive image of your company. Makes it easier to hire talented developers.
- You don't have to supply binary driver releases for each kernel version and patch version (closed source drivers).
- Drivers have all privileges. You need the sources to make sure that a driver is not a security risk.
- Your drivers can be statically compiled in the kernel.

# Books for Ref..

## Books:

- Understanding the Linux Kernel, D. P. Bovet and M. Cesati, O'Reilly & Associates, 2000.
- Linux Core Kernel – Commentary, In-Depth Code Annotation, S. Maxwell, Coriolis Open Press, 1999.
- The Linux Kernel, Version 0.8-3, D. A Rusling, 1998.
- Linux Kernel Internals, 2<sup>nd</sup> edition, M. Beck et al., Addison-Wesley, 1998.
- Linux Kernel, R. Card et al., John Wiley & Sons, 1998.
- Linux Device Drivers, 3rd Edition, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman Published by O'Reilly Media, Inc., 1005

# QUESTIONS ??



**Thank you!**