Introduction to Linux Kernel

Introduction to Linux Kernel

1. Linux History
2. Linux kernel Key features
3. Linux License
4. Linux Kernel Architecture
5. Linux Source tree Overview
6. /proc and /sys virtual file system.
7. What is Linux kernel
8. Two Roles of Kernel.
9. Kernel Programming .
10. Need for kernel programming
11. Kernel programs can be developed in two possible ways.
12. Built Own Kernel (Configuring, Compiling and Booting the Linux Kernel Configuration).

Linux History

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, **Linus Torvalds.**
- Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, more than one thousand people contribute to each kernel release, individuals or companies big and small.

Linux Kernel Features

- **Portability** and hardware support. Runs on most architectures.
- **Scalability.** Can run on super computers as well as on tiny devices (4 MB of RAM is enough).
- **Compliance to standards** and interoperability.
- **Exhaustive networking** support.
- **Security.** It can't hide its flaws. Its code is reviewed by many experts.
- **Stability and reliability.**
- **Modularity.** Can include only what a system needs even at run time.
- **Easy to program.** You can learn from existing code. Many useful resources on the net.

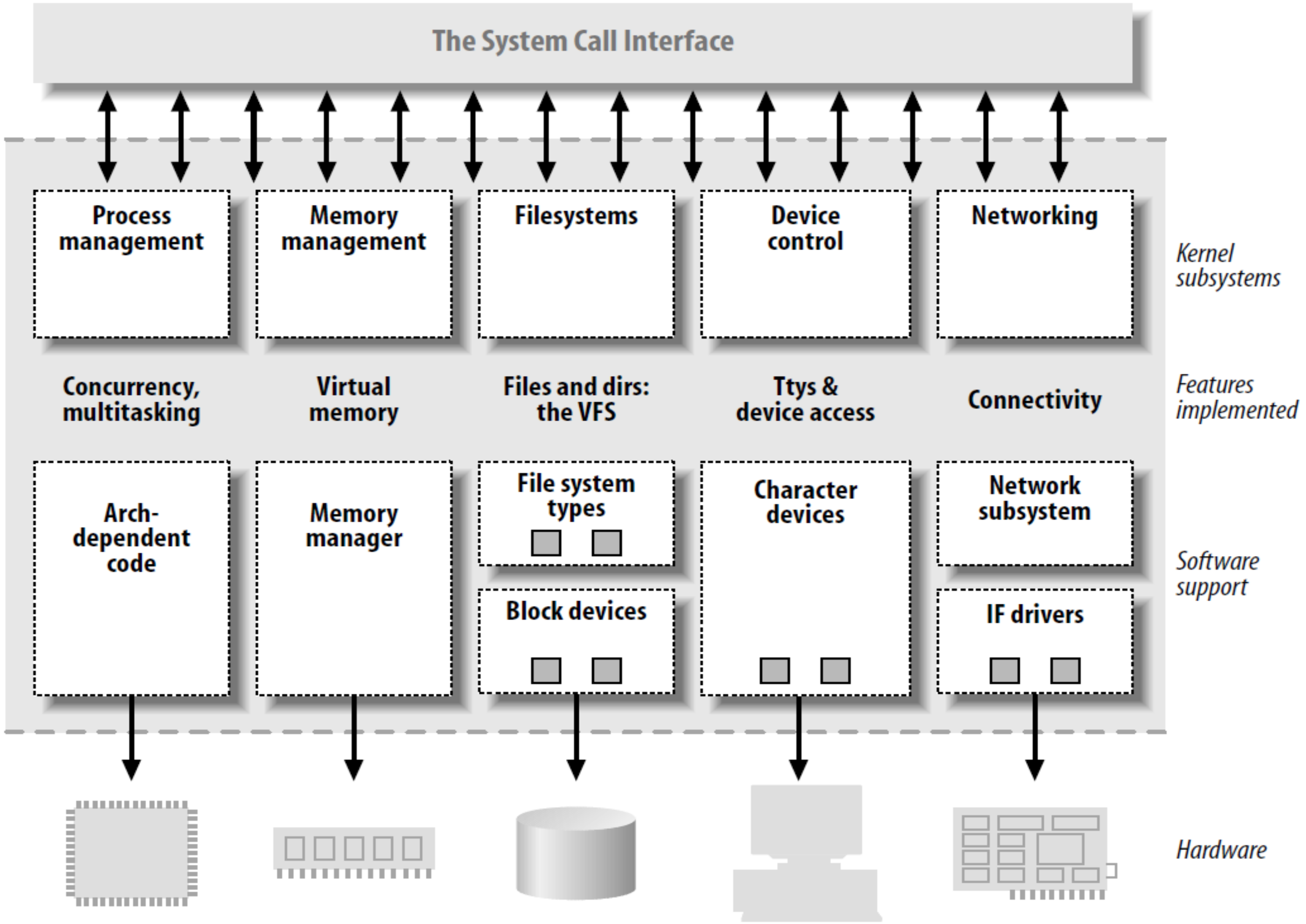
Linux License

- The whole Linux sources are Free Software released under the GNU **General Public License version 2 (GPL v2)**.
- For the Linux kernel, this basically implies that:
  - When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
  - When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction.

Introduction to Linux kernel

Linux Kernel Architecture

Linux Kernel Architecture



Vertical vs Horizontal

- Linux Kernel divided in to 5 verticals.
  1. Process Management
  2. Memory Management
  3. File Management
  4. Device Management
  5. Network Service.
- And 2 horizontals.
  1. High Level Device Drivers
     - Character Device Drivers
     - Block Device Drivers
     - Network Device Drivers
  2. Low Level Device Drivers
     - Platform/Bus Drivers

Introduction to Linux kernel

What is Kernel?

What is Kernel?

- Kernel is the first program that will get loaded in to memory(RAM) of computer.
- As kernel is running in the Supervisor (privileged) mode, only it can access the I/O hardware. Application programs which are running in the user mode (non-privileged) can not access the hardware directly. So application programs has to call the kernel functions (system call) to access the hardware.
- By default kernel and all application programs resided on the storage media like hard disk.

When computer is booted,

- kernel will get loaded first. Kernel remains in the memory till computer is shut down.
- Next kernel loads the user interface application program. This user interface program allows user to run other application programs.
- When application program is started, it is loaded in to RAM from the Hard disk. When application completes, it is removed from the memory.

Introduction to Linux kernel

Two Roles of Kernel

Two roles of Kernel

1. **Kernel acts as a scheduler.** This is active role of the Kernel. As a scheduler, Kernel schedules application programs by allocating CPU time slice for each of them.

2. **Service provider role**. As service provider, Kernel provides lots of services to the application programs. Application programs use these services by calling Kernel functions. Kernel functions are referred as System Calls.

Kernel Programming

- Need for Kernel Programming:
  - The most common need for kernel programming is to write a **device driver**. Device drivers which access h/w must run in the kernel mode.
  - Kernel programming is also required to add new **subsystems** in to the kernel.
  - Kernel programming is required to port kernel to a new h/w board by writing required **BSP (Board Support Packages)**.

Kernel Programming

Parameters	Linux Kernel Programming	Linux Device Drivers Programming
Usage	Kernel Developers focus on interfaces, data structures, algorithms and optimization for the core of the operating system.	Device Drivers use the interfaces and data structures written by the kernel developers to implement device control and IO.
Programming	Kernel programming is done using Module programming technique. There are no standard libraries available. Have to use pure C programming	Device Drivers is done using Module programming technique. There are no standard libraries available. Have to use pure C programming.
Skill Set	A very good kernel programmer may not know a lot about interrupt latency and hardware determinism, but he will know a lot about how locks, queues and Kobjects work.	A device driver programmer will know how to use locks, queues and other kernel interfaces to get their hardware working properly and responsively, but he won't be as likely to fix a page allocation bug or write a new scheduler.

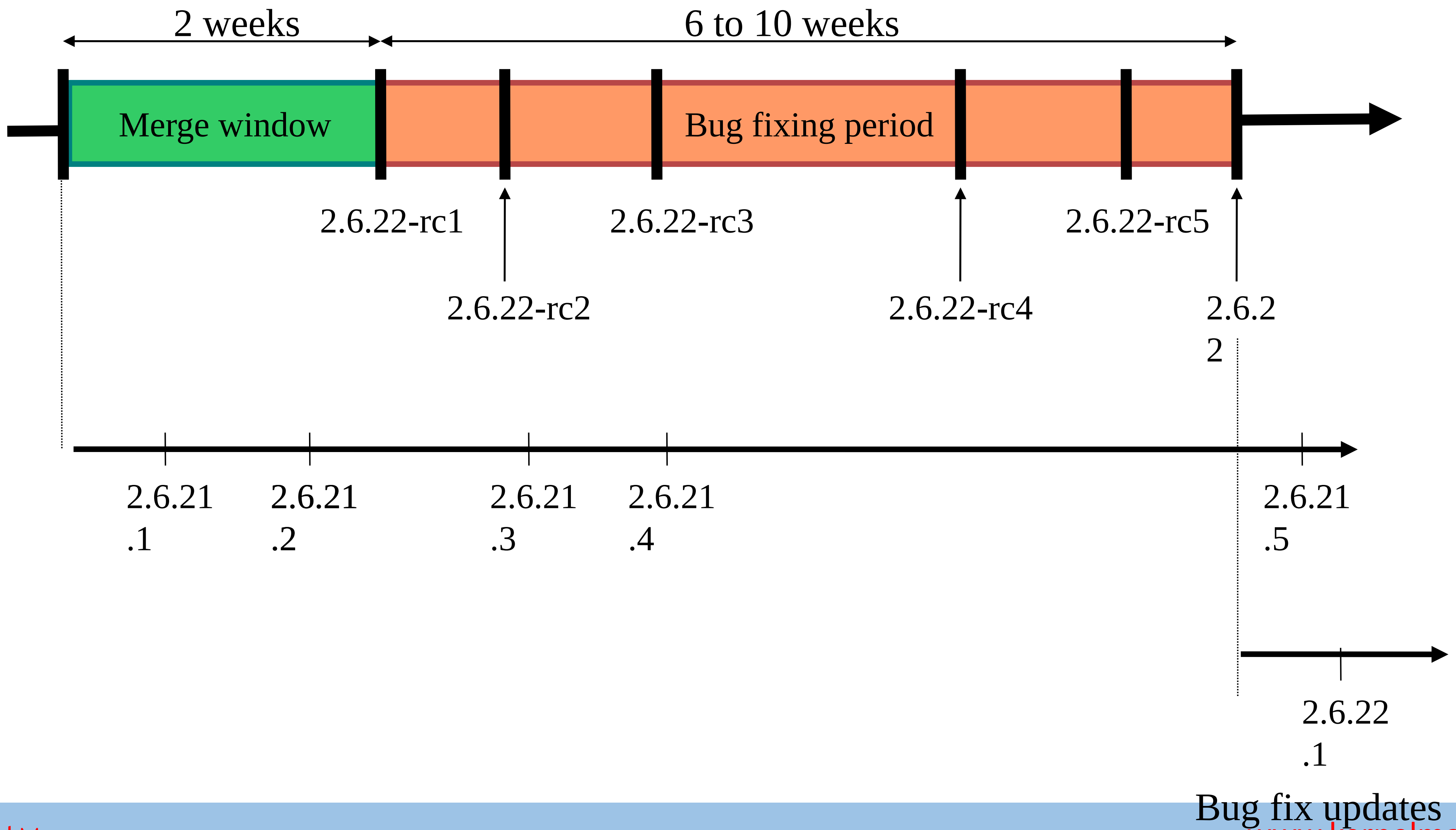
Introduction to Linux kernel

Linux Version

Linux Version

- Linux version numbers follow a longstanding tradition. Each version has three numbers, i.e., **X.Y.Z.**
- The "**X**" is only incremented when a really significant change happens, one that makes software written for one version no longer operate correctly on the other. This happens **very rarely** -- in Linux's history it has happened exactly once.
- The "**Y**" tells you which development "series" you are in. A stable kernel will always have an **even number** in this position, while a development kernel will always have an **odd number**.
- The "**Z**" specifies which **exact version** of the kernel you have, and it is incremented on every release.

Merge and bug fixing windows



Linux Kernel Source

Programming Languages

- Implemented in C like all Unix systems. (C was created to implement the first Unix systems)
- A little Assembly is used too:
  - CPU and machine initialization, exceptions
  - Critical library routines
- No C++ used
- All the code compiled with gcc
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel.

No C library

- The kernel has to be standalone and can't use user space code.
- User space is implemented on top of kernel services, not the opposite.
- Kernel code has to supply its own library implementations (string utilities, cryptography, uncompression ...)
- So, you can't use standard C library functions in kernel code. (printf(), memset(), malloc(),...).
- Fortunately, the kernel provides similar C functions for your convenience, like printk(), memset(), kmalloc(), ...

Location of kernel sources

- The official versions of the Linux kernel, as released by Linus Torvalds, are available at http://www.kernel.org
  - These versions follow the development model of the kernel
  - However, they may not contain the latest development from a specific area yet. Some features in development might not be

    ready for mainline inclusion yet.
- Many chip vendors supply their own kernel sources (https://www.codeaurora.org/)
  - Focusing on hardware support first
  - Can have a very important delta with mainline Linux
  - Useful only when mainline hasn't caught up yet.
- Many kernel sub-communities maintain their own kernel, with usually newer but less stable features (http://www.linux-arm.org/)
  - Architecture communities (ARM, MIPS, PowerPC, etc.), device drivers communities (I2C, SPI, USB, PCI, network, etc.), other communities (real-time, etc.)
  - No official releases, only development trees are available

Getting Linux sources

- The kernel sources are available from http://kernel.org/pub/linux/kernel as **full tarballs** (complete kernel sources) and **patches** (differences between two kernel versions).
- However, more and more people use the git version control system. Absolutely needed for kernel development!
- Fetch the entire kernel sources and history

git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

From Kernel Masters Server:

$ git clone git@192.168.1.6:root/linux-3.12.git

Linux Source code Layout

- arch/<ARCH>
  - Architecture specific code
  - arch/<ARCH>/mach-<machine>, machine/board specific code
  - arch/<ARCH>/include/asm, architecture-specific headers
  - arch/<ARCH>/boot/dts, Device Tree source files, for some architectures
- block/
  - Block layer core
- COPYING
  - Linux copying conditions (GNU GPL)
- CREDITS
  - Linux main contributors
- crypto/
  - Cryptographic libraries

Linux Source code Layout

- Documentation/
  - Kernel documentation. Don't miss it!
- drivers/
  - All device drivers except sound ones (usb, pci...)
- firmware/
  - Legacy: firmware images extracted from old drivers
- fs/
  - Filesystems (fs/ext4/, etc.)
- include/
  - Kernel headers
- include/linux/
  - Linux kernel core headers
- include/uapi/
  - User space API headers
- init/
  - Linux initialization (including init/main.c)
- ipc/
  - Code used for process communication

Linux Source code Layout

- Kbuild
  - Part of the kernel build system
- Kconfig
  - Top level description file for configuration parameters
- kernel/
  - Linux kernel core (very small!)
- lib/
  - Misc library routines (zlib, crc32...)
- MAINTAINERS
  - Maintainers of each kernel part. Very useful!
- Makefile
  - Top Linux Makefile (sets arch and version)
- mm/

Linux Source code Layout

- net/
  - Network support code (not drivers)
- README
  - Overview and building instructions
- REPORTING-BUGS
  - Bug report instructions
- samples/
  - Sample code (markers, kprobes, kobjects...)
- scripts/
  - Scripts for internal or external use
- security/
  - Security model implementations (SELinux...)
- sound/
  - Sound support code and drivers
- tools/
  - Code for various user space tools (mostly C)

Linux Source code Layout

- net/
  - Network support code (not drivers)
- README
  - Overview and building instructions
- REPORTING-BUGS
  - Bug report instructions
- samples/
  - Sample code (markers, kprobes, kobjects...)
- scripts/
  - Scripts for internal or external use
- security/
  - Security model implementations (SELinux...)
- sound/
  - Sound support code and drivers
- tools/
  - Code for various user space tools (mostly C)

Built Own Kernel

Kernel Configuration

Kernel configuration and build system

- The kernel configuration and build system is based on multiple Makefiles
- One only interacts with the main Makefile, present at the **top directory** of the kernel source tree
- Interaction takes place
  - using the make tool, which parses the Makefile
  - through various **targets**, defining which action should be done (configuration, compilation, installation, etc.). Run make help to see all available targets.
- Example
  - cd linux-3.12.13
  - make <target>

Kernel configuration

- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- Thousands of options are available, that are used to selectively compile parts of the kernel source code
- The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled
- The set of options depends
  - On your hardware (for device drivers, etc.)
  - On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.)

Source Management Tools

- ctags
- cscope

Parameters	Linux System Programming	Linux Kernel Programming	Linux Device Drivers Programming
Purpose	System Programmers write daemons, utilities and other tools for automating common or difficult tasks.	Kernel Developers focus on interfaces, data structures, algorithms and optimization for the core of the operating system.	Device Drivers use the interfaces and data structures written by the kernel developers to implement device control and IO.
Programming	All the Standard C libraries are available at system programming level.	Kernel programming is done using Module programming technique. There are no standard libraries available. Have to use pure C programming	Device Drivers is done using Module programming technique. There are no standard libraries available. Have to use pure C programming.
Application	System Programmer should know about various low level functions (system calls) for testing device drivers.	A very good kernel programmer may not know a lot about interrupt latency and hardware determinism, but he will know a lot about how locks, queues and Kobjects work.	A device driver programmer will know how to use locks, queues and other kernel interfaces to get their hardware working properly and responsively, but he won't be as likely to fix a page allocation bug or write a new scheduler.