# Chapter 1. Introduction

## 1.1 Linux

• Linux is an operating system, a software program that controls your computer.

• Linux is a Unix-like and mostly POSIX-compliant computer operating system (OS) assembled under the model of free and open-source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released by Linus Torvalds.The Free Software Foundation uses the name GNU/Linux to describe the operating system.

• Linux was originally developed as a free operating system for personal computers based on the Intel x86 architecture, but has since been ported to more computer hardware platforms than any other operating system. Linux has the largestinstalled base of all general-purpose operating systems.

### 1.1.1 History of Linux

In 1991, while attending the University of Helsinki, Torvalds became curious about operating systems and frustrated by the licensing of MINIX, which at the time limited it to educational use only. He began to work on his own operating system kernel, which eventually became the Linux kernel.

### 1.1.2. Distriution

• A Linux distribution (often abbreviated as distro) is an operating system made from a software collection, which is based upon the Linux kernel and, often, a package management system. Linux users usually obtain their operating system by downloading one of the Linux distributions, which are available for a wide variety of systems ranging from embedded devices (for example, OpenWrt) and personal computers (for example, Linux Mint) to powerful supercomputers (for example, Rocks Cluster Distribution).

• A typical Linux distribution comprises a Linux kernel, GNU tools and libraries, additional software, documentation, a window system (the most common being the X Window System), a window manager, and a desktop environment.

## 1.2 Kernel

• The Linux kernel is a computer operating system kernel.

• The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds. Then Linux quickly started to be used as the kernel for free software operating systems.

• The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.

Role of Linux Kernel?

• Manage all the hardware resources: CPU, memory, I/O.

• Provide a set of portable, architecture and hardware independent APIs (Application Programming Interface) to allow user space applications and libraries to use the hardware resources

• Handle concurrent accesses and usage of hardware resources from different applications. Example: a single network interface is used by multiple user space applications through various network connections. The kernel is responsible to ‘multiplex’ the hardware resource.

# Chapter 2. Linux kernel

## 2.1 Description of Linux kernel architecture

• The Linux kernel is a monolithic kernel, supporting true preemptive multitasking (both in user mode and, since the 2.6 series, in kernel mode), virtual memory, shared libraries, demand loading, shared copy-on-write executables (via KSM), memory management, the Internet protocol suite, and threading.

• Device drivers and kernel extensions run in kernel space (ring 0 in many CPU architectures), with full access to the hardware, although some exceptions run in user space, for example filesystems based on FUSE/CUSE, and parts of UIO. The graphics system most people use with Linux does not run within the kernel. Unlike standard monolithic kernels, device drivers are easily configured as modules, and loaded or unloaded while the system is running. Also, unlike standard monolithic kernels, device drivers can be pre-empted under certain conditions; this feature was added to handle hardware interrupts correctly, and to better support symmetric multiprocessing. By choice, the Linux kernel has no binary kernel interface.

• The hardware is also incorporated into the file hierarchy. Device drivers interface to user applications via an entry in the /dev or /sys directories. Process information as well is mapped to the file system through the /proc directory.

## 2.2 Linux kernel functions

* Portability and hardware support: Run on most architectures
* Scalability: Can run on super computer as well as tiny devices (4MB RAM)
* Compliance to standards and interoperability
* Exhausive networking support
* Security: 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: Many resources on the net

## 2.3 Relationship between Linux Kernel, Hardware, Applications?

## 2.4 Memory-mapped I/O and port-mapped I/O

## 2.5 System call

- The main interface between the kernel and user space is the

set of system calls.

+ About 300 system calls that provide the main kernel services. Example: File and device operations, networking operations, inter-process communication,…

+ Is stable over time: only new system calls can be added by the kernel developers

+ Is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

## 2.6 Interaction between Linux kernel and Users

The answer is that the kernel runs in kernel space, and normal programs run in user space. User space is basically a form of sand-boxing -- it restricts user programs so they can't mess with memory (and other resources) owned by other programs or by the OS kernel. This limits (but usually doesn't entirely eliminate) their ability to do bad things like crashing the machine.

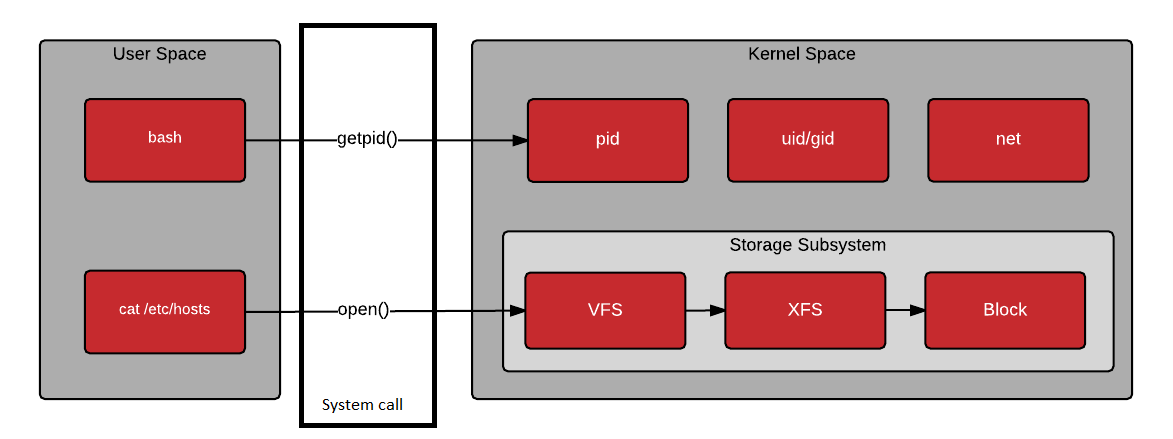
The kernel is the core of the operating system. It normally has full access to all memory and machine hardware (and everything else on the machine). To keep the machine as stable as possible, you normally want only the most trusted, well-tested code to run in kernel mode/kernel space.

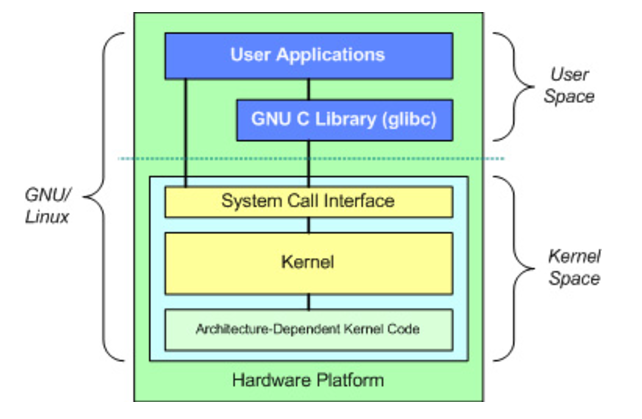
The stack is just another part of memory, so naturally it's segregated right along with the rest of memory.

• The user space, which is a set of locations where normal user processes run (i.e everything other than the kernel). The role of the kernel is to manage applications running in this space from messing with each other, and the machine.

• The kernel space, which is the location where the code of the kernel is stored, and executes under.

* stored, and executes under.





Kernel provides its functions only via special functions, called system calls standard C-library provides them Have strict separation of kernel data and data for user programs ⇒ need explicit copying between user program and kernel

# Chapter 3. Embedded Linux

## 3.1 Overview (what and why)

## 3.2 Boot process

# Chapter 4. Toolchain

## 4.1 Native

A native compiler is an application that converts one language, commonly a high-level language, into the binary or the computer's native language.

A native compiler is a compiler that works on compilation for the same technology on which it runs. It uses the same operating system or platform as the software for which it is assembling machine language.

Developers may recommend different native compiler options for different use cases involving languages like Java and C+. In evaluating a project, programmers might believe that the only benefit of using a native compiler is to prevent reverse engineering or for better code security. Other times, native compilers can have an impact on user experience because code can load more quickly. Within the IT community, professionals often ask one another about whether a native compiler is a good idea, and which specific native compiler options may be best for a development project

One way to understand a native compiler is by contrasting it with a cross compiler, which may compile code for programs running on different platforms. One use of cross compilers is in compiling programs for different hardware devices that may have their own respective platforms. In some cases, using a native compiler can provide significant benefits.

Common benefits of native compilation in some programming languages include better execution or load speed, as well as better inherent security. However, native compiler strategies do limit deployment to a single platform, which may be a drawback in some cases.

## 4.2 Cross

1. Cross compiling the kernel

- Runs on your workstation and generates code for your workstation.

- Compiling a Linux kernel for another CPU architecture:

+ Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.

+ Much easier as development tools for your GNU/Linux workstation are much easier to find.

+ To make the difference with a native compiler, cross compiler executables are prefixed by the name of the target system, architecture and sometimes library.

2. Specifying a cross compiler

The CPU architecture & cross compiler prefix are defined through the ARCH and CROSS\_COMPILE variables in Makefile.

- Makefile defines CC = $(CROSS\_COMPILE)gcc.

- There are 3 solutions:

+ Modify Makefile.

+ Set ARCH and CROSS\_COMPILE through the make command line.

Explanation: any variable set through the make command line overrides any setting in the Makefile.

Drawback: should never forget these settings when run make

+ Set ARCH and CROSS\_COMPILE as environment variables in terminal:

export ARCH=arm

export CROSS\_COMPILE=armlinuxCan

Advantages: Be set in project specific environments.

Not hardcoded in the Makefile.

Not interfere with patches.

Not forget to set them when you run any make command.

Caution: Only apply to shells in which these variables have been set.

3. Configuring the kernel

make xconfig

Configure same as in native compiling and set the right board / machine type.

4. Using readymade

Config files

- Default configuration files available for many boards / machines. Check if one exists in arch/<arch>/configs/ for your target.

- Using arch/<arch>/configs/ is a good way of releasing a default configuration file for a group of users or developers.

- Must run make <machine>\_defconfig.

5. Cross compiling setup

Choosing a toolchain

6. Building the kernel

Step1: Run make

Step2: Copy arch/<platform>/boot/zImage to the target storage

Step3: Customize arch/<arch>/boot/install.sh so that make install does this automatically.

Step4: make INSTALL\_MOD\_PATH=<dir>/ modules\_install then copy <dir>/ to /lib/modules/ on the target storage.

7. Summary

- Edit Makefile: set ARCH and CROSS\_COMPILE

- Get the default configuration: make <machine>\_defconfig (if existing in arch/<arch>/configs)

- Refine the configuration settings according to your requirements: make xconfig

- Add the cross compiler

path to your PATH environment variable

- Compile the kernel: make

- Copy the kernel image from arch/<arch>/boot/ to the target

- Copy modules to a directory which you replicate on the target:

make INSTALL\_MOD\_PATH=<dir> modules\_install

## 4.3 Comparison native vs Cross

|  |  |  |
| --- | --- | --- |
|  | **Cross** | **Native** |
| **Target CPU** | ARM | X86 |
| **Load Speed** | Slower | Faster |
| **Compiling Speed** | Faster | Slower |
| **Target Operating Environment** | For many other systems | Same as running OS or platform |
| **Reverse Engineering** | Normal | Hard |

Table of Contents

[Chapter 1. Introduction 1](#_Toc480839318)

[1.1 Linux 1](#_Toc480839319)

[1.1.1 History of Linux 1](#_Toc480839320)

[1.1.2. Distriution 1](#_Toc480839321)

[1.2 Kernel 1](#_Toc480839322)

[Chapter 2. Linux kernel 1](#_Toc480839323)

[2.1 Description of Linux kernel architecture 2](#_Toc480839324)

[2.2 Linux kernel functions 2](#_Toc480839325)

[2.3 Relationship between Linux Kernel, Hardware, Applications? 3](#_Toc480839326)

[2.4 Memory-mapped I/O and port-mapped I/O 3](#_Toc480839327)

[2.5 System call 3](#_Toc480839328)

[2.6 Interaction between Linux kernel and Users 3](#_Toc480839329)

[Chapter 3. Embedded Linux 5](#_Toc480839330)

[3.1 Overview (what and why) 5](#_Toc480839331)

[3.2 Boot process 5](#_Toc480839332)

[Chapter 4. Toolchain 5](#_Toc480839333)

[4.1 Native 5](#_Toc480839334)

[4.2 Cross 5](#_Toc480839335)

[4.3 Comparison native vs Cross 8](#_Toc480839336)