**OPEN SOURCE SOFTWARE**

**GROUP I**

|  |  |  |
| --- | --- | --- |
| ID | Name | Tasks |
| 1 | Nguyễn Phước Tài  (Group Leader) | Overview about boot process of Embedded Linux  What is Toolchain? Cross-compilation toolchain vs native toolchain? |
| 2 | Nguyễn Bá Linh | - What is Linux? History?  - Relationship between Linux Kernel, Hardware, Applications? |
| 3 | Lê Huỳnh Thiện Nhân | -What is Distributions of Linux?  - Memory-mapped I/O and port-mapped I/O |
| 4 | Trần Võ Phúc Hưng | - What is kernel?  - What is system call? |
| 5 | Trịnh Quang Hưng | - What is Linux kernel? Role of Linux Kernel ?  - Interaction between Linux kernel and Users ? |
| 6 | Nguyễn Xuân Thanh | - Description of Linux kernel architecture  - What and why is Embedded Linux? |
| 7 | Lê Khanh | - Linux kernel functions  - What is Bootloader? Role of BootLoader? |

*Group assignment:*

Contents

[I. OVERVIEW 3](#_Toc528086065)

[1. What is Linux? History? 3](#_Toc528086066)

[1.1. Linux 3](#_Toc528086067)

[1.2. History 4](#_Toc528086068)

[2. Distributions of Linux 5](#_Toc528086069)

[3. What is kernel? 6](#_Toc528086070)

[4. Linux kernel and Role of Linux kernel 6](#_Toc528086071)

[4.1. Linux kernel 6](#_Toc528086072)

[4.2. Feature 7](#_Toc528086073)

[4.3. Role of Linux kernel 8](#_Toc528086074)

[II. LINUX KERNEL 9](#_Toc528086075)

[1. Description of Linux kernel architecture 9](#_Toc528086076)

[2. Functions of Linux Kernel 13](#_Toc528086077)

[3. Relationship between Linux Kernel, Hardware, Applications 14](#_Toc528086078)

[4. Memory-mapped I/O and port-mapped I/O 16](#_Toc528086079)

[5. What is system call? 17](#_Toc528086080)

[6. Interaction between Linux kernel and Users 19](#_Toc528086081)

[III. Embedded Linux 21](#_Toc528086082)

[1. What and why Embedded Linux? 21](#_Toc528086083)

[2. What is Bootloader? Role of Bootloader? 22](#_Toc528086084)

[2.1. What is Bootloader? 22](#_Toc528086085)

[2.2. Role of Bootloader 22](#_Toc528086086)

[3. Overview about boot process of Embedded Linux 24](#_Toc528086087)

[4. What is Toolchain? Cross-compilation toolchain vs native toolchain? 27](#_Toc528086088)

[4.1. What is Toolchain? 27](#_Toc528086089)

[4.2. Cross-compilation toolchain vs native toolchain 27](#_Toc528086090)

1. **OVERVIEW**
2. **What is Linux? History?**
   1. **Linux**

Linux is a family of free and open-source software operating systems built around the Linux kernel. The defining component of a Linux distribution is the Linux kernel an operating system kernel first released on September 17, 1991, by Linus Torvalds. Many Linux distributions use the word "Linux" in their name. The Free Software Foundation uses the name GNU/Linux to refer to the operating system family, as well as specific distributions, to emphasize that most Linux distributions are not just the Linux kernel, and that they have in common not only the kernel, but also numerous utilities and libraries, a large proportion of which are from the GNU project.

The development of Linux is one of the most prominent examples of free and open-source software collaboration. The underlying source code may be used, modified and distributed—commercially or non-commercially—by anyone under the terms of its respective licenses, such as the GNU General Public License.

Just like Windows XP, Windows 7, Windows 8, and Mac OS X, Linux is an operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. To put it simply – the operating system manages the communication between your software and your hardware. Without the operating system (often referred to as the “OS”), the software wouldn’t function.

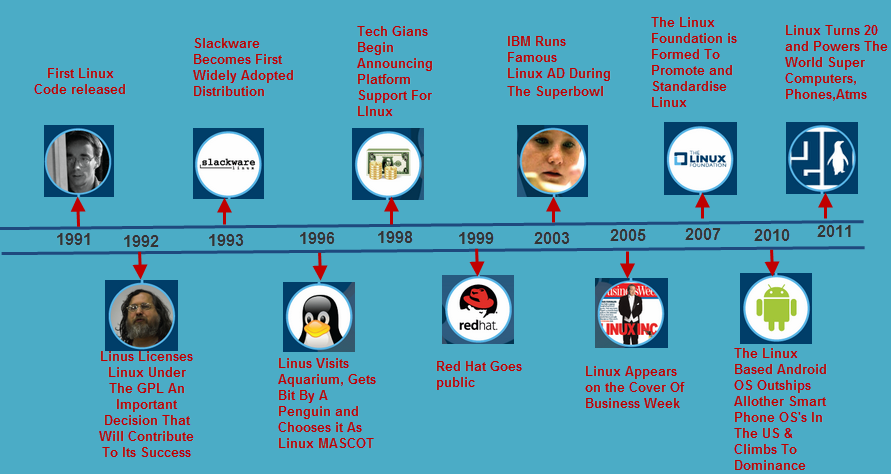
The OS is comprised of a number of pieces:

* **The Bootloader:** The software that manages the boot process of your computer.
* **The kernel:** The kernel is the core of the system and manages the CPU, memory, and peripheral devices. The kernel is the “lowest” level of the OS.
* **Daemons:** These are background services (printing, sound, scheduling, etc) that either start up during boot, or after you log into the desktop.
* **The Shell:** You’ve probably heard mention of the Linux command line
* **Graphical Server:** This is the sub-system that displays the graphics on your monitor
* **Desktop Environment:** There are many desktop environments to choose from (Unity, GNOME, Cinnamon, Enlightenment, KDE, XFCE, etc). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, games, etc).

**Applications:** Most modern Linux distributions (more on this in a moment) include App Store-like tools that centralize and simplify application installation. For example: Ubuntu Linux

* 1. **History**

The history of Linux began in 1991 to create a new free operating system kernel. Since then, the resulting Linux kernel has been marked by constant growth throughout its history. Since the initial release of its source code in 1991, it has grown from a small number of C files under a license prohibiting commercial distribution to the 4.15 version in 2018 with more than 23.3 million lines of source code without comments under the GNU General Public License v2.



1. **Distributions of Linux**

- 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. Most of the included software is free and open-source software made available both as compiled binaries and in source code form, allowing modifications to the original software. Usually, Linux distributions optionally include some proprietary software that may not be available in source code form, such as binary blobs required for some device drivers. A Linux distribution may also be described as a particular assortment of application and utility software (various GNU tools and libraries, for example), packaged together with the Linux kernel in such a way that its capabilities meet the needs of many users. The software is usually adapted to the distribution and then packaged into software packages by the distribution's maintainers. The software packages are available online in so-called repositories, which are storage locations usually distributed around the world. Beside glue components, such as the distribution installers (for example, Debian-Installer and Anaconda) or the package management systems, there are only very few packages that are originally written from the ground up by the maintainers of a Linux distribution.

Almost six hundred Linux distributions exist, with close to five hundred out of those in active development. Because of the huge availability of software, distributions have taken a wide variety of forms, including those suitable for use on desktops, servers, laptops, netbooks, mobile phones and tablets, as well as minimal environments typically for use in embedded systems. There are commercially backed distributions, such as Fedora (Red Hat), openSUSE (SUSE) and Ubuntu (Canonical Ltd.), and entirely community-driven distributions, such as Debian, Slackware, Gentoo and Arch Linux. Most distributions come ready to use and pre-compiled for a specific instruction set, while some distributions (such as Gentoo) are distributed mostly in source code form and compiled locally during installation.

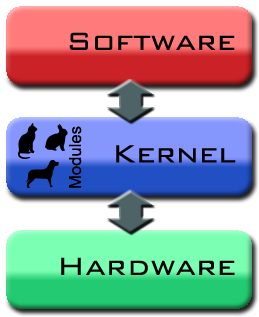
1. **What is kernel?**

A kernel is the core component of an operating system. Using interprocess communication and system calls, it acts as a bridge between applications and the data processing performed at the hardware level.

When an operating system is loaded into memory, the kernel loads first and remains in memory until the operating system is shut down again.

A computer kernel interfaces between the three major computer hardware components, providing services between the application/user interface and the CPU, memory and other hardware I/O devices.

The kernel provides and manages computer resources, allowing other programs to run and use these resources. The kernel also sets up memory address space for applications, loads files with application code into memory, sets up the execution stack for programs and branches out to particular locations inside programs for execution.

****

The kernel is responsible for:

Process management for application execution

Memory management, allocation and I/O

Device management through the use of device drivers

System call control, which is essential for the execution of kernel services

1. **Linux kernel and Role of Linux kernel**
   1. **Linux kernel**

The Linux kernel is a monolithic Unix-like computer operating system kernel.

The Linux kernel not only takes the approach of CPU, memory, and IPC, but it also includes things like device drivers, files system management, and system server calls. Because Linux kernel is monolithic kernel type, so it is better at accessing hardware and multitasking because if a program needs to get information from memory or another process running, it has a more direct line to access it and doesn’t have to wait in a queue to get things done. This however can cause problems because the more things that run in supervisor mode, the more things that can bring down your system if one doesn’t behave properly.

The Linux Kernel is the heart of the operating system. Without the Kernel, we simply cannot perform any task, since it is mainly responsible for the software and hardware of our computer working correctly and can interact with each other.

One of the advantages of the Linux kernel is that it is possible to update it without affecting the rest of the operating system, with a couple of commands (using the root user) in the Terminal. We would achieve this in a couple of minutes or even more simple through the Software Center, although this depends on the distribution that we choose. By updating only the system's kernel, we would have not only a more stable, also safer and faster equipment, all in several minutes.

* 1. **Feature**
* 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
  1. **Role of Linux kernel**

It provides a set of portable hardware and architecture APIs that offer user space applications the possibility to use necessary hardware resources

It helps with the management of hardware resources, such as a CPU, input/output peripherals, and memory

It is used for the management of concurrent accesses and the usage of necessary hardware resources by different applications.

In general terms, the kernel is a software code that serves as a layer between the hardware and main programs that run on a computer. It is the first part to load when the OS boots up. It is loaded in memory and stays there throughout the entire time the computer is in session.

1. **LINUX KERNEL**
2. **Description of Linux kernel architecture**

Kernel is a small and special code which is the core component of Linux OS and directly interacts with hardware.  It is the intermediate level between software and hardware which provides low level service to user mode’s components. It is fully developed in C language and file system architecture Moreover, it has different blocks which manage various operations. .In this tutorial we will learn about kernel architecture of Linux.

Kernel runs a number of processes concurrently and manages the various resources. It is viewed as a resource manager when several programs run concurrently on a system. In this case, the kernel is an instance that shares available resources like CPU time, disk space, network connections etc.

* 1. **Type of Kernel**

Kernel architecture is classified into two types: Monolithic, Micro Kernel.

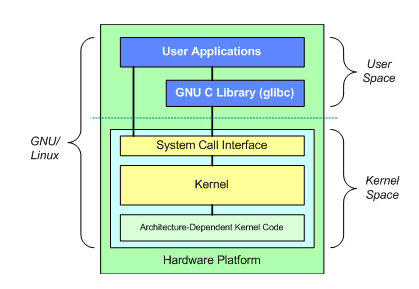
* + 1. **Monolithic Kernels**

In the traditional monolithic kernel architecture, all the basic system services like process and memory management, interrupt handling etc. were packaged into a single module in kernel space. Module is an object file in which the whole code can be linked to kernel at run time. Huge space is consumed by traditional monolithic kernel and maintenance level is very low. It takes few hours for recompilation if some new feature is added because all the services are attached into one single module.  
  
Modern monolithic kernel architecture consists of different modules which can be dynamically loaded and un-loaded. In this way maintainability is very easy because kernel needs to take care of only loaded module. Recompilation is not required upon added feature or some changes. Monolithic kernel is faster than micro kernel.

* + 1. **Micro kernels**

Monolithic kernel architecture supports the modular approach. All service modules are not run in kernel space as compared to monolithic kernel. Device driver management, protocol stack, file system etc are run in user space. This reduces the kernel code size and also increases the security.

* 1. Fundamental Architecture of Linux



Architecture of kernel divided into main two parts: User Space, Kernel Space.

* + 1. **User Space**

All user programs and applications are executed in user space. User Space cannot directly access the memory and hardware.  It accesses the hardware through kernel space. Processes or programs which are running in user space only access some part of memory by system call. Due to full protection, crashes in user mode are recoverable.

GNU C library provides the mechanism switching user space application to kernel space.

* + 1. **Kernel Space**

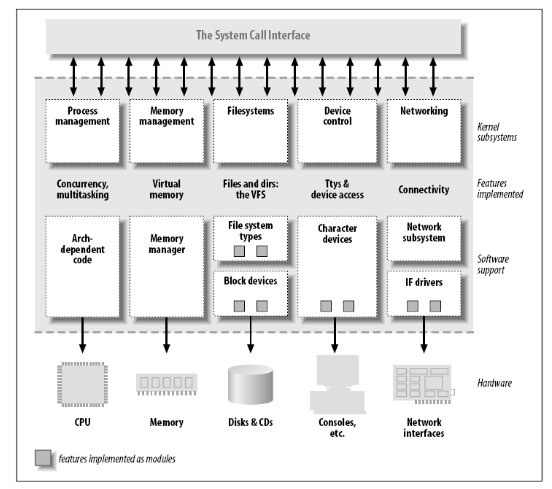
All kernel programs are executed in kernel space. Kernel space accesses full part of memory and directly interacts with hardware like RAM, Hard disk etc It is divided in different blocks and modules which manage all operations (like file management, memory management, process management etc.) in kernel space and applications  running in user space. Kernel space consists of system call interface, Kernel (core component of Linux) and device module.

System call interface is the intermediate layer between user space and kernel space. Each application which is run in user space can interface with kernel through system call interface. For example system call function on file operation are open ( ), write ( ), read ( ) etc.

Kernel is independent from hardware. It is common for all Hardware processors which are supported by Linux. You can run kernel on any processor like Intel, ARM, Atmel etc.  It acts as a resource manager in Kernel space and performs process management, file management, memory management, Interrupt handler, scheduling of process, etc.  It is a powerful structure which handles all kinds of operations.

* 1. Architecture of Kernel

Full Architecture of Kernel



* + 1. **Process management**

Process management is handling the management of various processes which are run at the same time. Process is an instance of ‘program in execution' like open the file, access drive, access external resource (i.e. printer), etc. and can be created and destroyed. Process management gives the information of what’s happening with a process and manages its priorities, such as which address should be assigned to process, file allocated to process, state of process (like running, waiting, stop) etc. I will explain in brief about process in another tutorial.

* + 1. **Memory management**

Memory management is the most important part of kernel which handles assignment of address space to process and application. Basically memory management assigns virtual memory instead of physical memory, whereby latter is the actual address space in RAM.  Don’t be confused between physical and virtual memory. Assignment of virtual address overcomes the limitation of assignment of physical memory. Conversion from physical address to virtual address is performed by MMU (Memory Management Unit which provides the protection of memory interference, sharing of memory and allocation of virtual memory. Physical address space is divided into some block of memory called frame that contains a number of pages.

* + 1. **Linux File System**

The Linux file system is a hierarchically data structured tree which organizes the file system. File is simple data structure (in form of some byte of memory) which can store any kind of data like text, image, video, music and whatever you want in computer. Linux file system is organized as a directory in tree form. Directory is a collection or group of files. Two types of directory are available in Linux system: 1) root 2) Sub directory. Root is main directory and cannot be accessed without permission of super user. It is the parent \ and main directory in file system denoted as forward slash ( / ). Sub directory is under the root directory which can be created or renamed by user. Linux supports number of file types like ext2, ext3, device file, block file, networking file system etc. Each file system is stored in different a disk partition.

1. **Functions of Linux Kernel**
   1. Resource allocation

The kernel's primary function is to manage the computer's resources and allow other programs to run and use these resources. These resources are- CPU, Memory and I/O devices.

* 1. Process Management

A process defines which memory portions the application can access. The main task of a kernel is to allow the execution of applications and support them with features such as hardware abstraction.

To run an application, a kernel first set up an address space for the application, then loads the file containing the application's code into memory, then set up a stack for the program and branches to a given location inside the program, thus finally starting its execution.

* 1. Memory Management

The kernel has full access to the system's memory. It allows processes to safely access this memory as they require it. Virtual addressing helps kernel to create virtual partitions of memory in two disjointed areas, one is reserved for the kernel (kernel space) and the other for the applications (user space).

* 1. I/O Device Management

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through Device Drivers. A device driver is a computer program that enables the operating system to interact with a hardware device. It provides the operating system with information of how to control and communicate with a certain piece of hardware.

A kernel maintains a list of available devices. A device manager first performs a scan on different hardware buses, such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB), to detect installed devices, then searches for the appropriate drivers. The kernel provides the I/O to allow drivers to physically access their devices through some port or memory location.

* 1. Inter- Process Communication

Kernel provides methods for Synchronization and Communication between processes called Inter- Process Communication (IPC). There are various approaches of IPC say, semaphore, shared memory, message queue, pipe (or named fifo), etc.

* 1. Scheduling

In a Multitasking system, the kernel will give every program a slice of time and switch from process to process so quickly that it will appear to the user as if these processes were being executed simultaneously. The kernel uses Scheduling Algorithms to determine which process is running next and how much time it will be given. The algorithm sets priority among the processes.

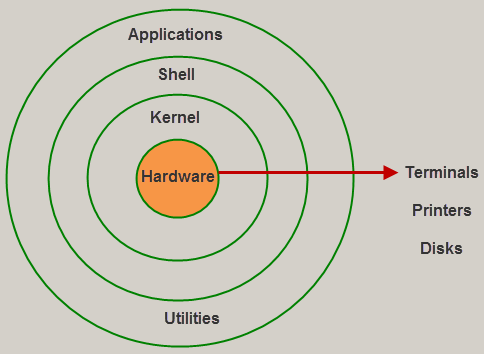
* 1. System Calls and Interrupt Handling

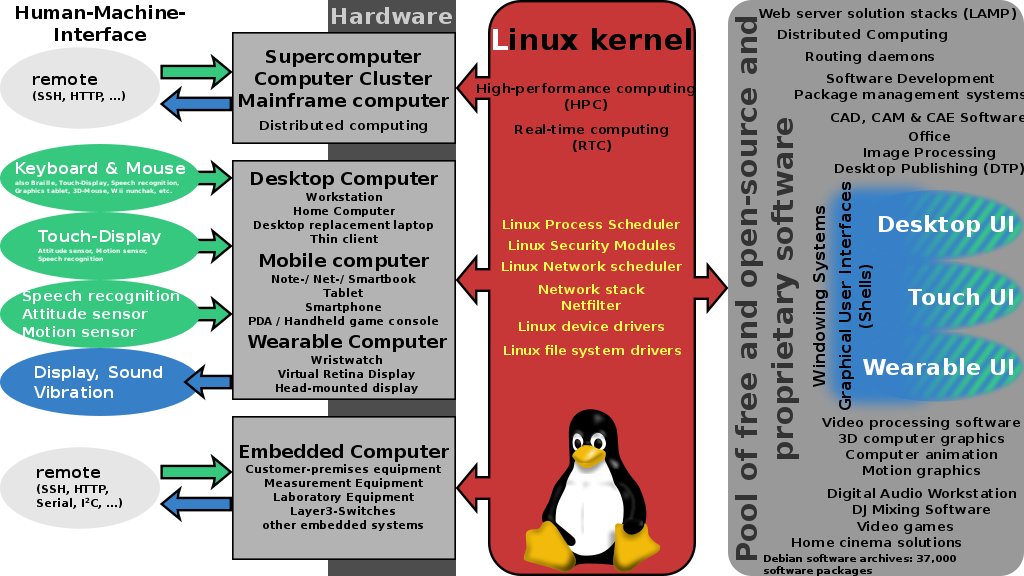
A system call is a mechanism that is used by the application program to request a service from the operating system. System calls include close, open, read, wait and write. To access the services provided by the kernel we need to invoke the related kernel functions. Most kernels provide a C Library or an API, which in turn invokes the related kernel functions. There are few methods by which the respective kernel function can be invoked- using Software- Simulated Interrupt, or using a Gate Call, or by using a Special System Call Instruction and by using a Memory- based Queue.

* 1. Security or Protection Management

Kernel also provides protection from faults (error control) and from malicious behaviors (Security). One approach toward this can be Language based protection system, in which the kernel will only allow code to execute which has been produced by a trusted language compiler.

1. **Relationship between Linux Kernel, Hardware, Applications**





1. **Memory-mapped I/O and port-mapped I/O**

Microprocessors normally use two methods to connect external devices: memory mapped or port mapped I/O. However, as far as the peripheral is concerned, both methods are really identical.

Memory mapped I/O is mapped into the same address space as program memory and/or user memory, and is accessed in the same way.

Port mapped I/O uses a separate, dedicated address space and is accessed via a dedicated set of microprocessor instructions.

The difference between the two schemes occurs within the microprocessor. Intel has, for the most part, used the port mapped scheme for their microprocessors and Motorola has used the memory mapped scheme. As 16-bit processors have become obsolete and replaced with 32-bit and 64-bit in general use, reserving ranges of memory address space for I/O is less of a problem, as the memory address space of the processor is usually much larger than the required space for all memory and I/O devices in a system. Therefore, it has become more frequently practical to take advantage of the benefits of memory-mapped I/O. However, even with address space being no longer a major concern, neither I/O mapping method is universally superior to the other and there will be cases where using port-mapped I/O is still preferable.

*Memory-mapped I/O*

* I/O devices are mapped into the system memory map along with RAM and ROM. To access a hardware device, simply read or write to those 'special' addresses using the normal memory access instructions.
* The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device.
* The disadvantage to this method is that the entire address bus must be fully decoded for every device. For example, a machine with a 32-bit address bus would require logic gates to resolve the state of all 32 address lines to properly decode the specific address of any device. This increases the cost of adding hardware to the machine.

*Port-mapped I/O*

* I/O devices are mapped into a separate address space. This is usually accomplished by having a different set of signal lines to indicate a memory access versus a port access. The address lines are usually shared between the two address spaces, but less of them are used for accessing ports. An example of this is the standard PC which uses 16 bits of port address space, but 32 bits of memory address space.
* The advantage to this system is that less logic is needed to decode a discrete address and therefore less cost to add hardware devices to a machine. On the older PC compatible machines, only 10 bits of address space were decoded for I/O ports and so there were only 1024 unique port locations; modern PC's decode all 16 address lines. To read or write from a hardware device, special port I/O instructions are used.
* From a software perspective, this is a slight disadvantage because more instructions are required to accomplish the same task. For instance, if we wanted to test one bit on a memory mapped port, there is a single instruction to test a bit in memory, but for ports we must read the data into a register, then test the bit.

1. **What is system call?**

System call is the fundamental interface between an application and the linux kernel, the programmatic way in which a program request a service from the kernel of the operating system it is executed on. This may include hardware-related services (for example: accessing a hard disk drive), creation and execution of new processed, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.

Kernel Mode

* When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
* Hence kernel mode is a very privileged and powerful mode.
* If a program crashes in kernel mode, the entire system will be halted.

User Mode

* When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
* In user mode, if any program crashes, only that particular program is halted.
* That means the system will be in a safe state even if a program in user mode crashes.
* Hence, most programs in an OS run in user mode.

System Call

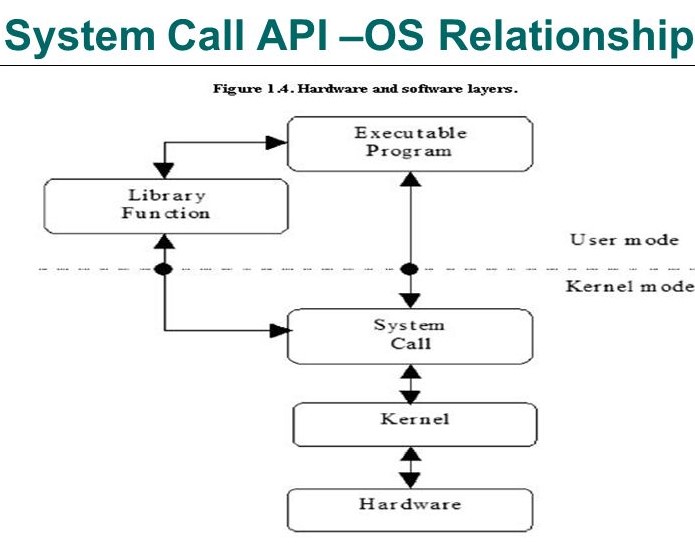
When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

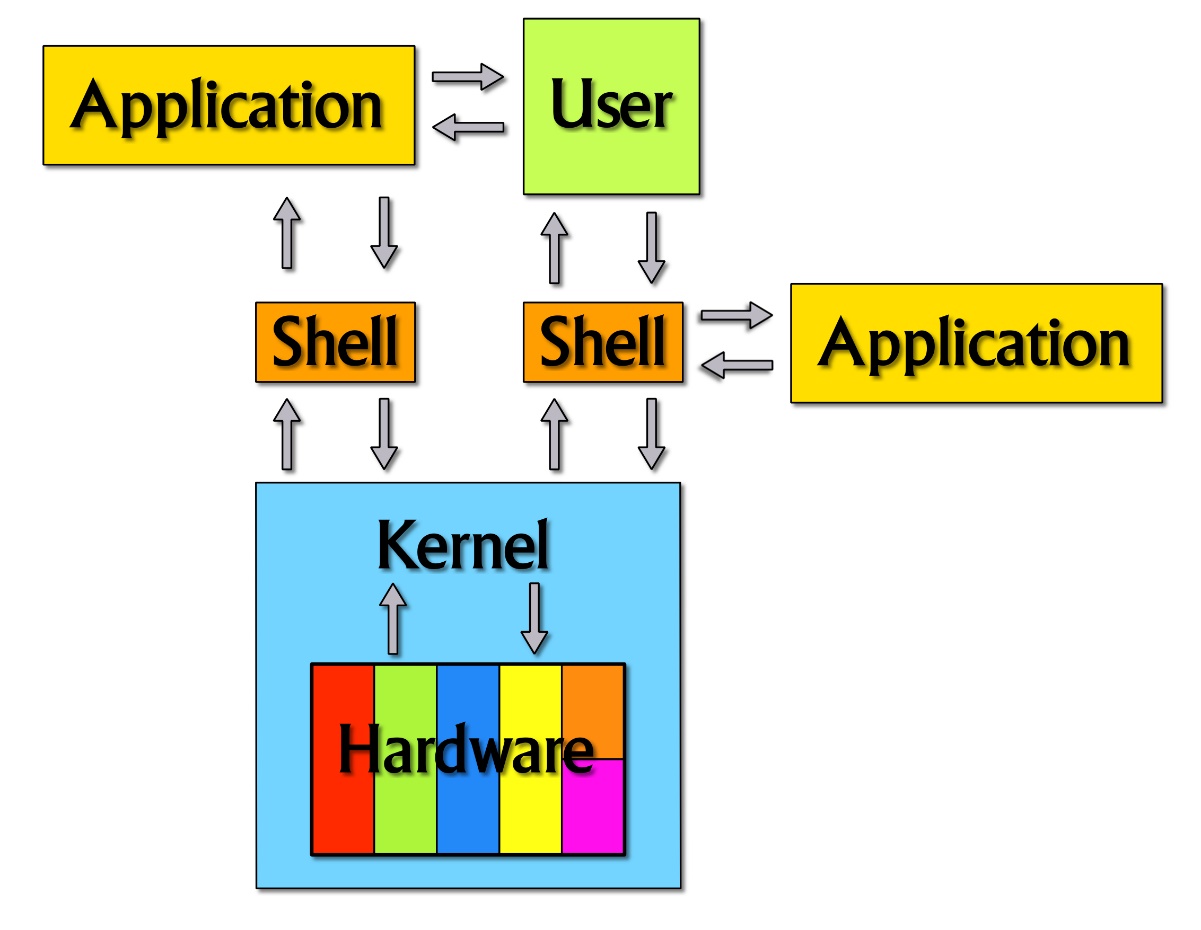
Generally, system calls are made by the user level programs in the following situations:

* Creating, opening, closing and deleting files in the file system.
* Creating and managing new processes.
* Creating a connection in the network, sending and receiving packets.
* Requesting access to a hardware device, like a mouse or a printer.



1. **Interaction between Linux kernel and Users**

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls. The user interface with kernel by the shell. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). As an illustration of the way that the shell and the kernel work together, suppose a user types rm myfile (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program rm, and then requests the kernel, through system calls, to execute the program rm on myfile. When the process rm myfile has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.



1. **Embedded Linux**
2. **What and why Embedded Linux?**

Linux itself is a kernel, but ‘Linux’ in day to day terms rarely means so. Embedded Linux generally refers to a complete Linux distribution targeted at embedded devices.

Why the Linux Embedded is generally?

Linux supports a huge variety of applications and networking protocols.

Linux is scalable can be used in small memory space also. Kernel footprint is less than 500 KB.

Linux is a royalty free operating system.

Linux has attracted a huge number of active developers, enabling rapid support of new hardware architectures, platforms, and devices.

Linux is largely accepted by hardware vendors, chip makers, single board computer maker etc. These vendors regularly release customized OS, drivers, and example programs for their hardware. Purpose is to sell their product by giving ready to use stuff to their clients.

* Linux is very well supported by community and by commercial vendors.

1. **What is Bootloader? Role of Bootloader?**
   1. **What is Bootloader?**

The bootloader is the first code that is executed after a system reset. Its goal is to bring the system to a state in which it can perform its main function. This requires hardware initialization and choosing the correct image to load from flash. Because of its key role, the bootloader is usually placed in a part of the Flash that is protected from accidental erasure or corruption.

Hardware initialization may imply enabling access to RAM, setting up clocks and PLLs, and configuring other key peripherals. However, hardware initialization should be restricted to the essential at this level leaving the rest of the initializations for upper application code.

* 1. **Role of Bootloader**

When power is first applied to a processor board, many elements of hardware must be initialized before even the simplest program can run. Each architecture and processor has a set of predefined actions and configurations upon release of reset, which includes fetching initialization code from an onboard storage device (usually Flash memory). This early initialization code is part of the bootloader and is responsible for breathing life into the processor and related hardware components.

Most processors have a default address from which the first bytes of code are fetched upon application of power and release of reset. Hardware designers use this information to arrange the layout of Flash memory on the board and to select which address range(s) the Flash memory responds to. This way, when power is first applied, code is fetched from a well-known and predictable address, and software control can be established.

The bootloader provides this early initialization code and is responsible for initializing the board so that other programs can run. This early initialization code is almost always written in the processor's native assembly language.

Of course, after the bootloader has performed this basic processor and platform initialization, its primary role is fetching and booting a full-blown operating system. It is responsible for locating, loading, and passing control to the primary operating system. In addition, the bootloader might have advanced features, such as the capability to validate an OS image, upgrade itself or an OS image, or choose from among several OS images based on a developer-defined policy. Unlike the traditional PC-BIOS model, when the OS takes control, the bootloader is overwritten and ceases to exist.

1. **Overview about boot process of Embedded Linux**

Initialize

Kernel

Stage 2 bootloader

Stage 1 bootloader

System Startup

Power-up / Reset

Operation

* **System startup:**

This is the first stage of booting process. When you power on/Restart your machine the power is supplied to SMPS (switched-mode power supply) which converts AC to DC. The DC power is supplied to all the devices connected to that machine such as Motherboard HDD's, CD/DVD-ROM, Mouse, keyboard etc. The most intelligent device in the computer is Processor(CPU), when supplied with power will start running it’s sequence operations stored in it’s memory. The first instruction it will run is to pass control to BIOS(Basic Input/Output System) to do POST(Power On Self Test). Once the control goes to BIOS it will take care of two things

Run POST operation.

Selecting first Boot device.

POST operation: POST is a processes of checking hardware availability. BIOS will have a list of all devices which are present in the previous system boot. In order to check if a hardware is available for the present booting or not it will send an electric pulse to each and every device in the list that it already have. If an electrical pulse is returned from that device it will come to a conclusion the hardware is working fine and ready for use. If it does not receive a single from a particular device it will treat that device as faulty or it was removed from the system. If any new hardware is attached to the system it will do the same operation to find if it’s available or not. The new list will be stored in BIOS memory for next boot.

Selecting First Boot Device: Once the POST is completed BIOS will have the list of devices available. BIOS memory will have the next steps details like what is the first boot device it has to select etc. It will select the first boot device and gives back the control to Processor(CPU). Suppose if it does not find first boot device, it will check for next boot device, if not third and so on. If BIOS do not find any boot device it will alert user stating "No boot device found".

* **Stage 1 bootloader:**

Once the BIOS gives control back to CPU, it will try to load MBR of the first boot device(We will consider it as HDD). MBR is a small part of Hard Disk with just a size of 512 Bytes, I repeat it’s just 512 Bytes. This MBR resides at the starting of HDD or end of HDD depending on manufacturer.

What is MBR?

MBR(Master Boot recorder) is a location on disk which have details about

Primary boot loader code(This is of 446 Bytes)

Partition table information(64 Bytes)

Magic number(2 Bytes)

Which will be equal to 512B (446+64+2)B.

Primary Boot loader code: This code provides boot loader information and location details of actual boot loader code on the hard disk. This is helpful for CPU to load second stage of Boot loader.

Partition table: MBR contains 64 bytes of data which stores Partition table information such as what is the start and end of each partition, size of partition, type of partition(Whether it's a primary or extended etc). As we all know HDD support only 4 partitions, this is because of the limitation of it’s information in MBR. For a partition to represent in MBR, it requires 16 Bytes of space in it so at most we will get 4 partitions.

Magic Number: The magic number service as validation check for MBR. If MBR gets corrupted this magic number is used to retrieve it. What to take backup of your MBR .

Once your CPU knows all these details, it will try to analyse them and read the first portion of MBR to load Second stage of Boot loader

* **Stage 2 bootloader:**

Once the Bootloader stage 1 is completed and able to find the actual bootloader location, Stage 1 bootloader start second stage by loading Bootloader into memory. In this stage GRUB(Grand Unified Bootloader) which is located in the first 30 kilobytes of hard disk immediately following the MBR is loaded into RAM for reading it’s configuration and displays the GRUB boot menu (where the user can manually specify the boot parameters) to the user. GRUB loads the user-selected (or default) kernel into memory and passes control on to the kernel. If user do not select the OS, after a defined timeout GRUB will load the default kernel in the memory for starting it.

* **Kernel:**

Once the control is given to kernel which is the central part of all your OS and act as a mediator of hardware and software components. Kernel once loaded into to RAM it always resides on RAM until the machine is shutdown. Once the Kernel starts it’s operations the first thing it do is executing INIT process.

* **Initialize:**

This is the main stage of Booting Process

init(initialization) process is the root/parent process of all the process which run under Linux/Unix. The first process it runs is a script at /etc/rc.d/rc.sysinit which check all the system properties, hardware, display, SElinux, load kernel modules, file system check, file system mounting etc. Based on the appropriate run-level, scripts are executed to start/stop various processes to run the system and make it functional. INIT process read /etc/inittab which is an initialization table which defines starting of system programs. INIT will start each run level one after the other and start executing scripts corresponds to that runlevel. The script information is stored in different folders in /etc/ folder

1. **What is Toolchain?** **Cross-compilation toolchain vs native toolchain?**
   1. **What is Toolchain?**

In software, a toolchain is the set of programming tools that is used to perform a more complex software development task or to create a software product, which is typically another computer program or a set of related programs.

A toolchain is the set of compiler, linker, librarian and any other tools you need to produce the executable (Shared libraries, etc.) for the target device. A debugger and/or IDE may also count as part of a toolchain.

* 1. **Cross-compilation toolchain vs native toolchain**
* *Native Toolchain:*

- The usual software development tools available on a GNU/Linux workstation is a native toolchain.

- This toolchain runs on workstation and generates code for the same kind workstation, usually x86.

- For embedded system development, it is usually impossible or not interesting to use a native toolchain.

- The target is too restricted in terms of storage and/or memory.

- The target is very slow compared to your workstation. You may not want to install all development tools on your target.

* *Cross compilation Toolchain:*

- Unlike native toolchain, cross-compiling toolchain is a set of software development tools available on any specific workstation platform and can generate for other target device platform.

- Cross-compiling toolchain use the cross-compiler, cross-compiler is a compiler where the target is different from the host. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone.