**Investigation Output**

**EMBEDDED LINUX**

**CHAPTER 1: LINUX**

1. **What is linux?**

* Linux is an operating system, a software program that controls your computer.
* Linux is a [Unix-like](https://en.wikipedia.org/wiki/Unix-like) and mostly [POSIX](https://en.wikipedia.org/wiki/POSIX)-compliant computer [operating system](https://en.wikipedia.org/wiki/Operating_system) (OS) assembled under the model of [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software) development and distribution. The defining component of Linux is the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel), an [operating system kernel](https://en.wikipedia.org/wiki/Kernel_%28computing%29) first released by [Linus Torvalds](https://en.wikipedia.org/wiki/Linus_Torvalds). The [Free Software Foundation](https://en.wikipedia.org/wiki/Free_Software_Foundation) uses the name [*GNU*](https://en.wikipedia.org/wiki/GNU)*/Linux* to describe the operating system.
* Linux was originally developed as a [free operating system](https://en.wikipedia.org/wiki/Free_operating_system) for [personal computers](https://en.wikipedia.org/wiki/Personal_computer) based on the [Intel x86](https://en.wikipedia.org/wiki/Intel_x86) architecture, but has since been [ported](https://en.wikipedia.org/wiki/Porting) to more [computer hardware platforms](https://en.wikipedia.org/wiki/Computer_hardware_platforms) than any other operating system. Linux has the [largest](https://en.wikipedia.org/wiki/Usage_share_of_operating_systems) [installed base](https://en.wikipedia.org/wiki/Installed_base) of all general-purpose operating systems.

1. **History**

In 1991, while attending the [University of Helsinki](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Linux_kernel).

| **Year** | **Version** | **Users** | **Kernel size (Bytes)** | **Milestone(s)** |
| --- | --- | --- | --- | --- |
| 1991 | 0.01 | 100 | 63,362 | Linus Torvalds writes Linux kernel |
| 1992 | 0.99 | 1000 | 431,591 | GNU software integrated with Linux kernel, producing a fully functional operating system |
| 1993 | 0.99 | 20,000 | 937,917 | High rate of code contributions prompts Linus to delegate code review responsibility |
| 1994 | 1.0 | 100,000 | 1,016,601 | First production release |
| 1995 | 1.2 | 500,000 | 1,850,182 | Linux adapted to non-Intel processors |
| 1996 | 2.0 | 1,500,000 | 4,718,270 | Linux supports multiple processors, IP masquerading, and Java |
| 1999 | 2.2 | 7,500,000 | 10,600,000 | Linux growth rate exceeds that of Microsoft Windows NT |

Table 1.1: The History of Linux

1. **Linux distribution**

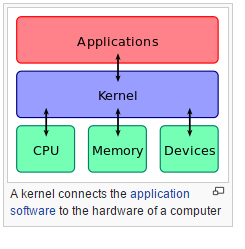
* A Linux distribution (often abbreviated as distro) is an operating system made from a software collection, which is based upon the Linux kernel and, often, a package management system. Linux users usually obtain their operating system by downloading one of the Linux distributions, which are available 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.

**CHAPTER 2: LINUX KERNEL**

# **What is kernel?**

## **Definition**

* The kernel (also called nucleus) is a computer program that constitutes the central core of a computer's operating system. It has complete control over everything that occurs in the system. As such, it is the first program loaded on startup, and then manages the remainder of the startup, as well as input/output requests from software, translating them into data processing instructions for the central processing unit. It is also responsible for managing memory, and for managing and communicating with computing peripherals, like printers, speakers, etc. The kernel is a fundamental part of a modern computer's operating system
* A kernel is the lowest level of easily replaceable software that interfaces with the hardware in your computer. It is responsible for interfacing all of your applications that are running in “user mode” down to the physical hardware, and allowing processes, known as servers, to get information from each other using inter-process communication (IPC).



## **Classification**

## **Microkernel:**

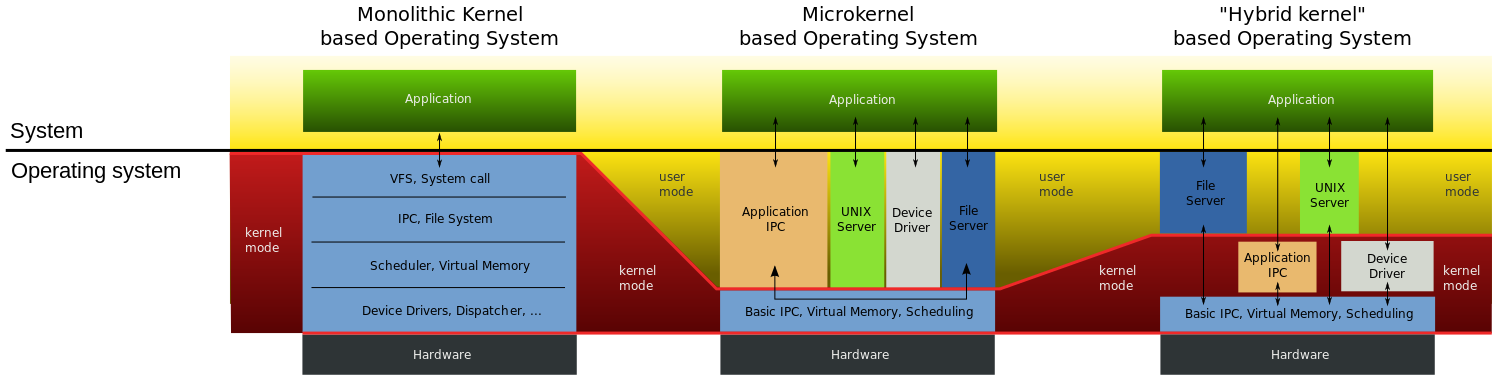
* A microkernel takes the approach of only managing what it has to: CPU, memory, and IPC.
* Pretty much everything else in a computer can be seen as an accessory and can be handled in user mode

## **Monolithic Kernel:**

* Monolithic kernels are the opposite of microkernels because they encompass not only the CPU, memory, and IPC, but they also include things like device drivers, file system management, and system server calls.

## **Hybrid Kernel:**

* + Hybrid kernels have the ability to pick and choose what they want to run in user mode and what they want to run in supervisor mode.
  + In general, most kernels fall into one of three types: monolithic, microkernel, and hybrid. Linux is a monolithic kernel while OS X (XNU) and Windows 7 use hybrid kernels



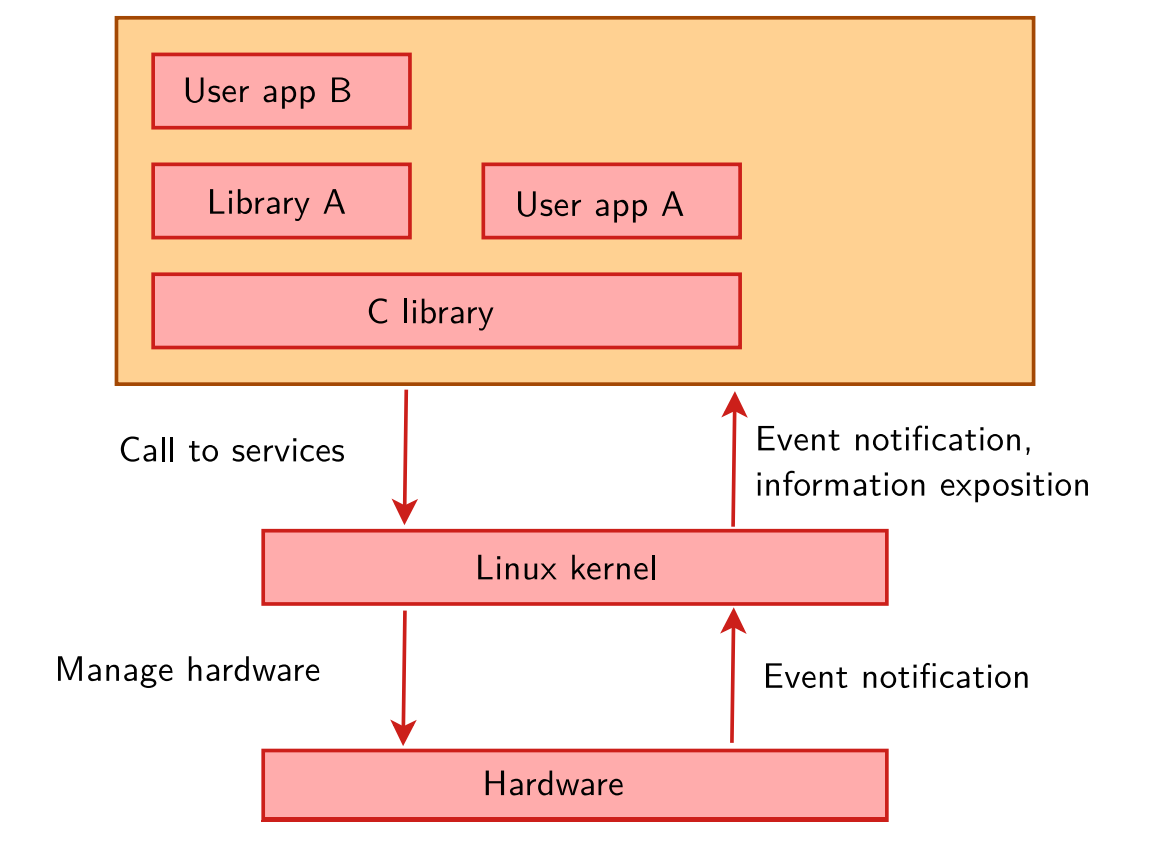
# **What is Linux Kernel? Role of Linux Kernel?**

## **What is Linux 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.

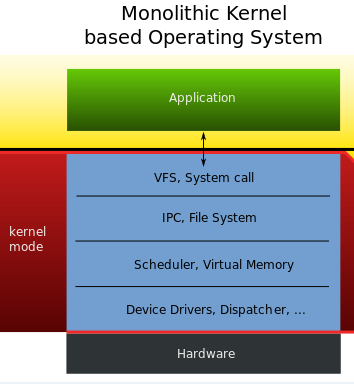
## **Role of Linux Kernel?**

* **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.



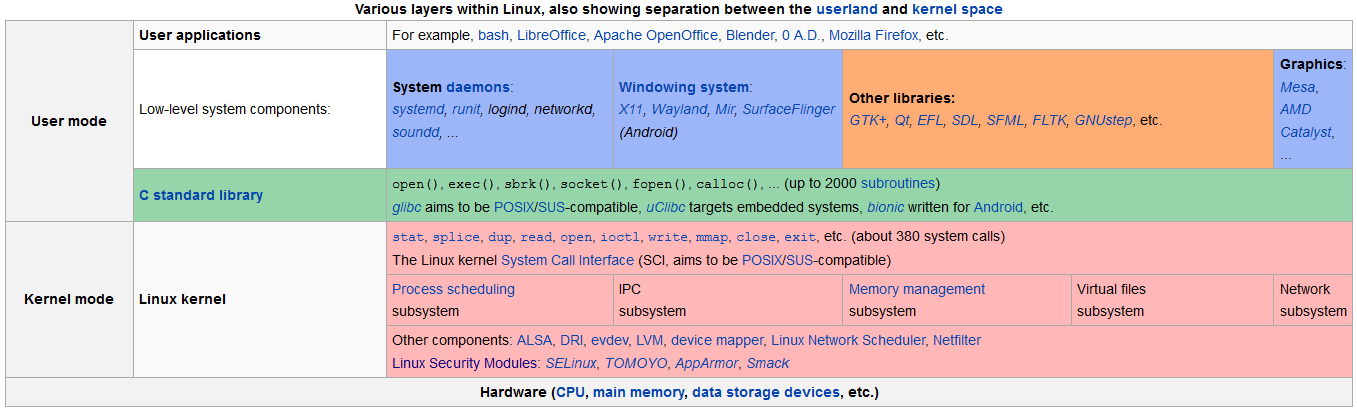
# **Description of Linux Kernel architecture**

## 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.

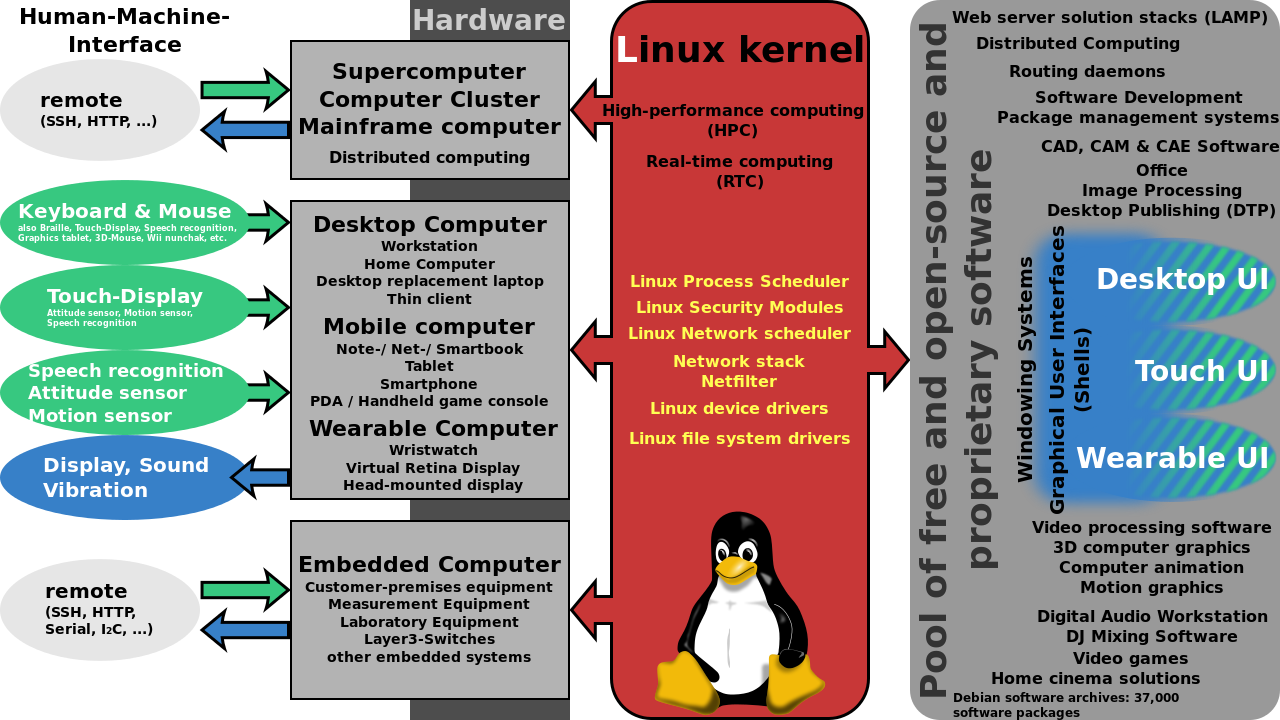


# **Linux Kernel function**

# **Relationship between Linux Kernel, Hardware and Application?**

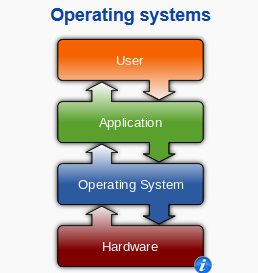
## **Relationship between Linux Kernel and Hardware**

* **Linux kernel can be installed on different computer architectures**
  + Linux is now one of the most widely ported operating system kernels, running on a diverse range of systems from the ARM architecture to IBM Z/Architecture mainframe computers
* **Linux can run on super computers as well as on tiny devices (4 MB of RAM is enough)**

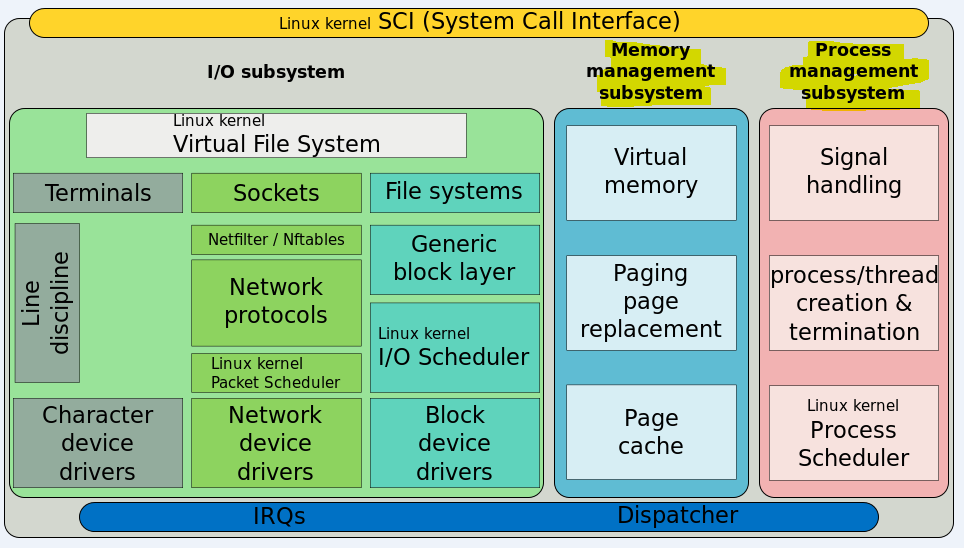
****

## **Relationship between Linux Kernel and Application**

* **Kernel helps application to access to hardware correctly and safely (using device drivers):**
  + 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. The driver is an important and vital piece to a program application. The design goal of a driver is abstraction; the function of the driver is to translate the OS-mandated function calls (programming calls) into device-specific calls. In theory, the device should work correctly with the suitable driver. Device drivers are used for such things as video cards, sound cards, printers, scanners, modems, and LAN cards
  + Example: To show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel



* **Kernel helps applications schedule running time:**
  + This central component of a computer system is responsible for running or executing programs
  + The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors (each of which can usually run only one program at a time)
* **Kernel helps store multiple programs in RAM efficiently:**
  + RAM is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute.
  + Often multiple programs will want access to memory, frequently demanding more memory than the computer has available
  + The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough memory is available.



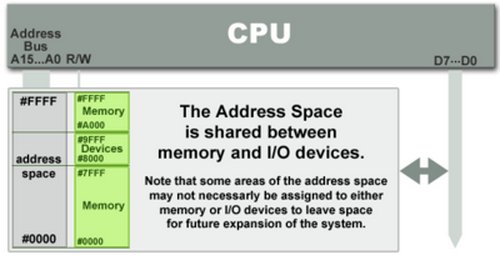
* **Kernel helps applications communicate with I/O devices:**
  + I/O devices include such peripherals as keyboards, mice, disk drives, printers, network adapters, and display devices
  + The kernel allocates requests from applications to perform I/O to an appropriate device and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device)

# **Memory-mapped I/O and port-mapped I/O?**

## **Memory-mapped I/O**

## **Definition:**

* The method to access I/O memory and I/O register with the same address space with memory. So it increase cost if adding more I/O.
* The CPU instructions used to access the memory can also be used for accessing devices by address



## **Method to access:**

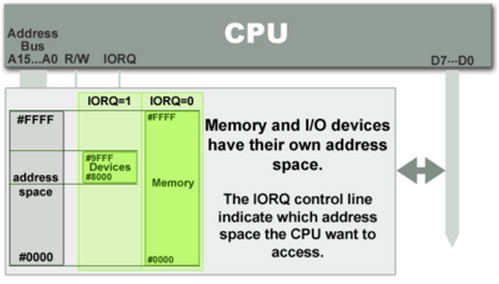
* Can access by address of memory because they have the same address bus. ioread and iowrite instruction usually to access memory



## **Port-mapped I/O**

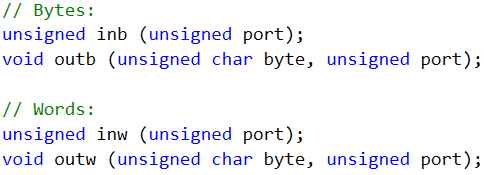
## **Definition:**

* I/O devices have a separate address space from general memory.
* The address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O



## **Method to access:**

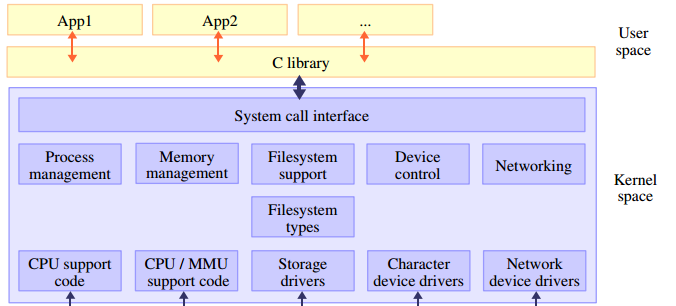
* Port - mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as in and out.(use in intel x86-64 architectures)
* To access: with port is the base address of access port. It control all address of port.



**CHAPTER 3: USERS SPACE AND LINUX KERNEL**

1. **System call :**
   1. **Definition:**

* System call is the programmatic way in which a computer program requests 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 processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.
* In Linux, a system call is an interface between a user-space application and a service that the kernel provides. This system call interface 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.
* System call has to finish before another task can be scheduled. This interface is stable over time: only new system calls can be added by the kernel developers
* It has some advantages:
* Freeing users from studying low-level programming.
* It greatly increases system security.
* These interfaces make programs more portable.
  1. **Services:**



**User space**

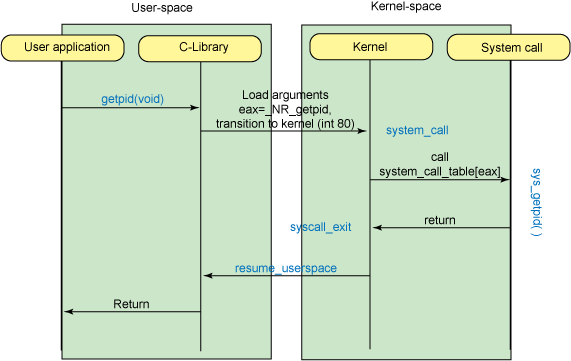
**Linux kernel**

* About **380** system calls that provide the main kernel services
* File and device operations
* Networking operations
* Inter-process communication
* Process management
* Memory mapping
* Timers, threads
* Synchronization primitives
* Etc.
  1. **Spacing :**

The Linux kernel provides a useful way to trace the system calls that a process invokes (as well as those signals that the process receives). The utility is called **strace** and is executed from the command line, using the application you want to trace as its argument. This tracing is accomplished in the kernel when the current system call request has a special field set called syscall\_trace, which causes the function **do\_syscall\_trace** to be invoked.

System call tracer: <http://sourceforge.net/projects/strace/>

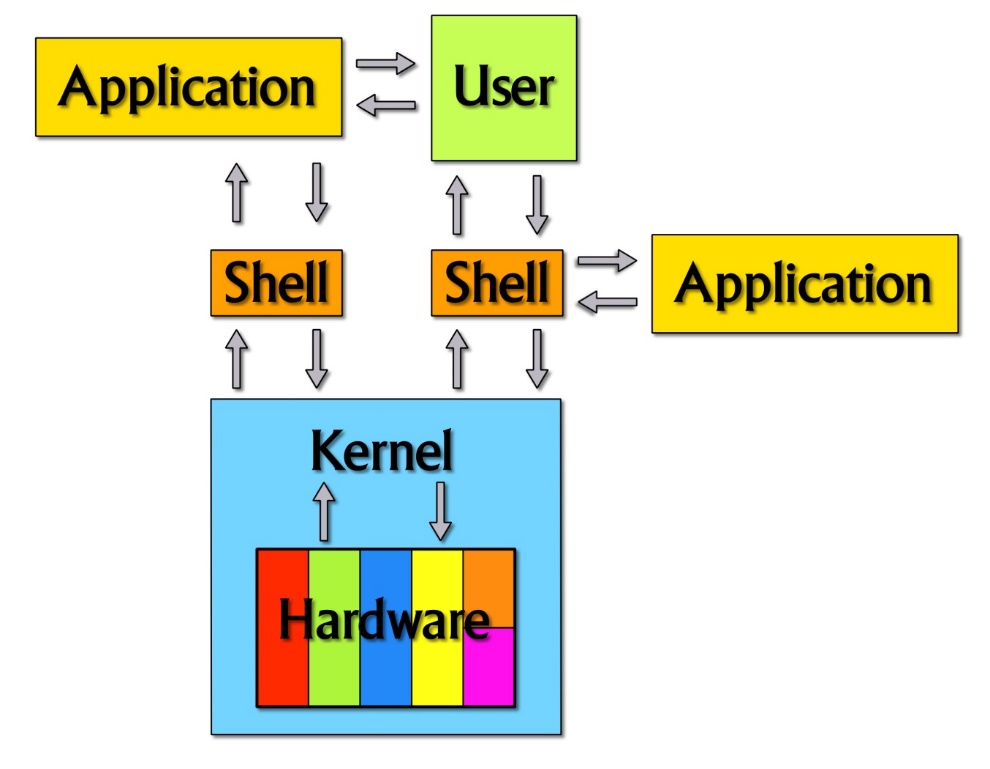
1. **Interaction between users space and linux kernel :**



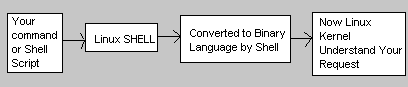
* When users call a function to control hardware , the “call” action mean Users use C library
  + When the C library has loaded the system call index and any arguments, a software interrupt is invoked (interrupt 0x80), which results in execution (through the interrupt handler) of the **system\_call** function. This function handles all system calls, as identified by the contents of eax. After a few simple tests, the actual system call is invoked using the **system\_call\_table** and index contained in eax. Upon return from the system call, **syscall\_exit** is eventually reached, and a call to **resume\_userspace** transitions back to user-space.
  + Some popular system call in <asm/unistd.h>: open, read, write, close, wait, exec, fork, exit, and kill.

**Linux Shell and Linux Kernel:**

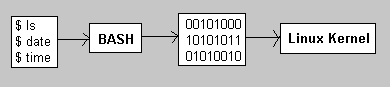
* + The Linux Kernel is the core piece of software that orchestrates the way the Ubuntu operating system works: it manages processes and hardware devices, allocate processor time to each process, etc.
  + There is no real such thing as the Linux Shell. Rather there are a number of "shell" programs, such as bash, ksh, csh (you'll notice they all end up in 'sh'). A shell is run when you open a terminal window and allows you to interact with the operating system using a command line interface
* Simplicity the shell acts as a translator between human language you type in and machine language that the kernel understands and vice versa. The shell in Linux is the command line interface (CLI) whereas in Windows you can think of the shell as the desktop.



* In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English and translate it into computers native binary language.  
    
  This is what Shell Does for US



* You type Your command and shell convert it as



* Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
* Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.
* Several shell available with Linux including

|  |  |  |  |
| --- | --- | --- | --- |
| **Shell Name** | **Developed by** | **Where** | **Remark** |
| BASH ( Bourne-Again SHell ) | Brian Fox and Chet Ramey | Free Software Foundation | Most common shell in Linux. It's Freeware shell. |
| CSH (C SHell) | Bill Joy | University of California (For BSD) | The C shell's syntax and usage are very similar to  the C programming language. |
| KSH (Korn SHell) | David Korn | AT & T Bell Labs |  |
| TCSH | See the man page. Type **$ man tcsh** | -- | TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH). |

* Note that each shell does the same job, but each understand a different command syntax and provides different built-in functions.
* In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are

**CHAPTER 4: EMBEDDED LINUX**

1. **What is embedded Linux? Why is embedded Linux?**
   1. **What is embedded Linux**

* Embedded Linux is the usage of the *Linux kernel* and various *open-source* components in embedded systems.
  1. **Why is embedded Linux?**
* Advantages:
  + Re-using components
  + Low cost
  + Full control
  + Quality
  + Eases testing of new features
  + Community support
  + Taking part into the community
    1. **Re-Using Components**
* The open-source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
* As soon as a hardware device, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.
* Allows to quickly design and develop complicated products, based on existing components.
* Allows to focus on the added value of your product.
  + 1. **Low cost**
* Free software can be duplicated on as many devices as you want, free of charge.
* If your embedded system uses only free software, you can reduce the cost of software licenses to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.
* Allows to have a higher budget for the hardware or to increase the company’s skills and knowledge.
  + 1. **Full control**
* With open-source, you have the source code for all components in your system.
* Allows unlimited modifications, changes, tuning, debugging, and optimization for an unlimited period of time.
  + Without lock-in or dependency from a third-party vendor.
* To be true, none open-source components must be avoided when the system is designed and developed.
* Allows to have full control over the software part of your system.
  + 1. **Quality**
* Many open-source components are widely used, on millions of systems.
* Usually higher quality than what an in-house development can produce, or even proprietary vendors.
* Of course, not all open-source components are of good quality, but most of the widely-used ones are.
* Allows to design your system with high-quality components at the foundations.
  + 1. **Eases testing of new features**
* Open-source being freely available, it is easy to get a piece of software and evaluate it.
* Allows to easily study several options while making a choice.
* Much easier than purchasing and demonstration procedures needed with most proprietary products.
* Allows to easily explore new possibilities and solutions.
  + 1. **Community support**
* Open-source software components are developed by communities of developers and users.
* This community can provide a high-quality support: you can directly contact the main developers of the component you are using. The likely hood of getting an answer doesn’t depend what company you work for.
* Often better than traditional support, but one needs to understand how the community works to properly use the community support possibilities.
* Allows to speed up the resolution of problems when developing your system.

1. **Boot process of Embedded Linux**
2. **Part A**

* Software components Involved in Embedded Linux Boot Process

Bootloader, kernel image, root file system – either an Initrd image or NFS location.

* Steps during Booting process of a conventional PC

System startup – PC\_BIOS/Boot Monitor 🡪 stage 1 bootloader – MBR 🡪 stage 2 bootloader – LILO, GRUB 🡪 kernel-Linux 🡪 Init – the user space.

* Booting process for an Embedded systems

Instead of BIOS you will run program from a fixed location in flash 🡪 the components involved in the first steps of PC boot process are combined in to a single “boot strap firmware”, called “boot loader” 🡪 bootloader also provides additional features useful for development & debugging.

* What is system startup?

CPU starts executing BIOS at address 0xFFFF0 🡪 Power on self-test – is the first step of BIOS 🡪 run time services – involve local deice enumeration and initialization 🡪 after the POST is complete, POST related code is flushed out memory. But BIOS run time services remain in memory and are available to the target OS. 🡪 The run-time searches for devices that are both active and bootable in order of preference defined in CMOS settings. 🡪 The primary boot loader is loaded and BIOS returns control to it.

* The primary boot loader

Perform few optional initializations 🡪 its main job is to load the secondary boot loader.

* Second boot loader

The second stage boot loader loads the Linux & an optional initial RAM disk in to the memory 🡪 on PC, the initrd is used as a temporary root files system, before final root file system gets mounted. However, on embedded systems, the initrd is generally the final root file system. 🡪 The secondary loader passes control to the kernel image – kernel is decompressed & initialized 🡪 the secondary boot loader is the kernel Loader, can also load optional initial RAM disk and then invokes the kernel image.

* Kernel Invocation

As the kernel is invoked, it performs the checks on system hardware, enumerates the attached hardware devices, mounts the root device 🡪 next it loads the necessary kernel modules 🡪 first user-space program now starts and high-level system initialization is performed 🡪 the kernel invocation process is similar on embedded Linux system as well as on PC.

* Kernel image

Is typically a compressed image [zlib compression] 🡪 typial named a zlmage (<512 kb) or bzImage (>512KB) 🡪 at the head of this image is a routine that does some minimal amount of hardware set up and then decompresses the kernel contained tin the kernel image and places in to high memory.

1. **Part B**

(1) Kernel Invocation Process - A Summary

(a) zImage Entry Point

(b) PERFORM BASIC HARDWARE SET UP

(c) PERFORM BASIC ENVIRONMENT SET UP

(d) CLEAR BSS

[Now we have set up the run time environment for the code to be executed next]

(e) DECOMPRESS THE KERNEL IMAGE

(f) Execute the decompressed Kernel Image

- INITIALIZE PAGE TABLES

- ENABLE MMU

- DETECT CPU (& optional FPU) TYPE & SAVE THIS INFO

(g) The First Kernel C function

- DO FURTHER INITIALIZATIONS

- LOAD INITRD

[The above code is being executed by swapper process, the one with pid 0]

(h) The Init Process

- FORK INIT PROCESS

- Init process is with pid 1

- Invoke Scheduler

- RELINQUISH CONTROL TO SCHEDULER

(2) zImage Entry point

(a) This is a call to the absolute physical address by boot loader

- Refer to file arch/\*\*\*/boot/compressed/head.S: start() in kernel source.

- For the ARM process this is "arch/arm/boot/compressed/head.S: start()"

(b) start() performs

- Basic hardware set up

- Basic environment set up

- Clears bss

- Calls the decompress\_kernel()

(3) Decompressing Kernel Image

(a) This is a call to arch/\*\*\*/boot/compressed/misc.c: decompress\_kernel()

This function decompresses the kernel image, stores it in to the RAM & returns the address of decompressed image in RAM.

(b) On ARM processor this maps to "arch/arm/boot/compressed/misc.c: decompress\_kernel()" routine.

(4) Execute the decompressed Kernel Image

(a) After we have got the (uncompressed) kernel image in RAM, we execute it.

(b) Execution starts with call\_kernel() function call [from start()].

(c) call\_kernel() will start executing the kernel code, from Kernel entry point.

(d) arch/\*\*\*/kernel/head.S contains the kernel entry point.

- Separate entry points for Master CPU and Secondary CPUs (for SMP systems).

- This code is in asm

- Page Tables are Initialized & MMU is enabled.

- Type of CPU alongwith optional FPU is detected and stored

- For Master CPU; start\_kernel(), which is the first C function to be executed in kernel, is called.

- For secondary CPUs (on an SMP system); secondary\_start\_kernel() is the first C function to be called.

(e) On ARM process it maps to "arch/arm/kernel/head.S

- Contains kernel ENTRY points for master and secondary CPU.

- For Master CPU "mmap\_switched()" is called as soon as mmu gets enabled. The mmap\_switched() saves the CPU info makes a call to start\_kernel()

- For Secondary CPU "secondary\_start\_kernel()" is called as soon as MMU gets enabled.

(5) The first kernel C function

(a) The start\_kernel() function is being executed by the swapper process.

(b) Refer to init/main.c: start\_kernel() in the kernel source.

(c) start\_kernel():

- A long list of initialization functions are called: this sets up interrupts, performs further memory configuration & loads the initrd.

- Calls rest\_init() in the End.

(d) Secondary\_start\_kernel() for secondary CPUs (on SMP systems).

- Arch/\*\*\*/kernel/smp.c: secondary\_start\_kernel()

- For ARM, arch/arm/kernel/smp.c

- There is not rest\_init() call for secondary CPUs.

(6) Init process

(a) Refer to init/main.c: rest\_init() in kernel source.

(b) Executed only on the Master CPU

(c) rest\_init() forks new process by calling kernel\_thread() function

(d) kernel\_thread(kern\_init,\*,\*); kern\_init has PID-1

(e) kernel\_init() will call the initialization scripts.

(f) kernel\_thread() defined in "arch/\*\*\*/kernel/process.c: kernel\_thread()".

(g) On ARM, arch/arm/kernel/process.c

(7) Invoke scheduler

(a) The rest\_init() calls cpu\_idle() in end [after it is done creating the init process]

(b) For the Secondary CPUs (on SMP systems), cpu\_idle is directly called from secondary\_start\_kernel [no step-5 & hence no init process].

(c) cpu\_idle() defined in "arch/\*\*\*/kernel/process.c: cpu\_idle()".

(d) On ARM, arch/arm/kernel/process.c

(8) Initrd image

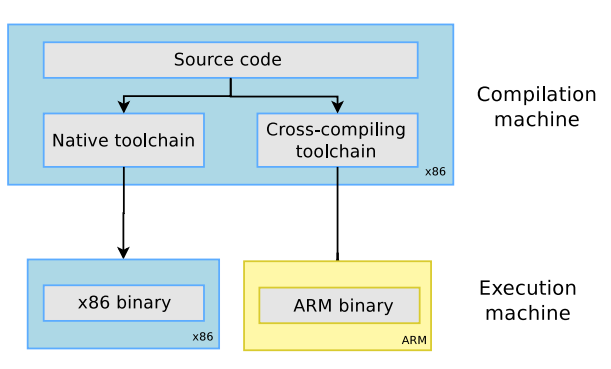
(a) The initrd serves as a temporary root file system in RAM & allows the kernel to fully boot without having to mount and physical disks. Since the necessary modules needed to interface with peripherals can be part of initrd the kernel can be very small.

(b) pivot\_root() routine: the root files system is pivoted where the initrd root file system is unmounted & the real root file system is mounted.

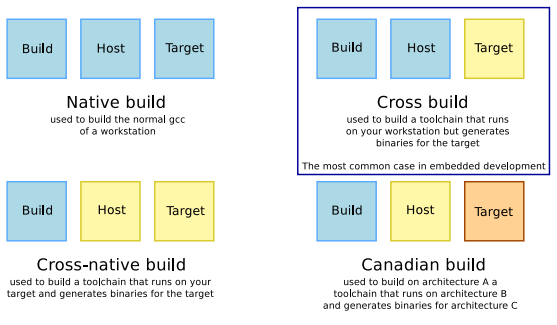
(c) In any embedded system, the initrd could be the root file system.

1. **What is toolchains? Native toolchain vs cross-compilation toolchain?**
   1. **Definition:**

* The usual development tools available on a GNU/Linux workstation is a native toolchain.
* This toolchain runs on your workstation and generates code for your 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.
* Therefore, cross-compiling toolchains are generally used. They run on your workstation but generate code for your target.

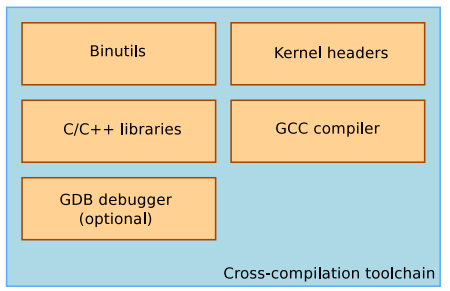


* 1. **Machines in build procedures:**
* Three machines must be distinguished when discussing toolchain creation:
* The **build** machine, where the toolchain is built.
* The **host** machine, where the toolchain will be executed.
* The **target** machine, where the binaries created by the tool chain are executed.
* Four common build types are possible for toolchains.
  1. **Different toolchain build procedure:**



*From these three different machines, we distinguish four different types of toolchain building processes:*

* A native toolchain, as can be found in normal Linux distributions, has usually been compiled on x86, runs on x86 and generates code for x86.
* A cross-compilation toolchain, which is the most interesting toolchain type for embedded development, is typically compiled on x86, runs on x86 and generates code for the target architecture (be it ARM, MIPS, PowerPC or any other architecture supported by the different toolchain components)
* A cross-native toolchain, is a toolchain that has been built on x86, but runs on your target architecture and generates code for your target architecture. It's typically needed when you want a native GCC on your target platform, without building it on your target platform.
* A Canadian build is the process of building a toolchain on machine A, so that it runs on machine B and generates code for machine C. It's usually not really necessary.
  1. **Components:**



1. **Binutils:** is a set of tools to generate and manipulate binaries for a given CPU architecture.
2. **Kernel Headers:**

* The kernel to user space ABI is **backward compatible**
* Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won’t be able to use the new system calls, data structures, etc.
* Binaries generated with a toolchain using kernel headers newer than the running kernel might work on if they don’t use the recent features, otherwise they will break.
* Using the latest kernel headers is not necessary, unless access to the new kernel features is needed.
* The kernel headers are extracted from the kernel sources using the headers\_install kernel Make file target.

1. **C library:**

* The C library is an essential component of a Linux system:
* Interface between the applications and the kernel.
* Provides the well-known standard C API to ease application development.
* Several C libraries are available: *glibc*, *uClibc*, *eglibc*, *dietlibc*, *newlib*, etc.
* The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.