Training Agenda

- Understand how to use Telechip Dolphine Boards (803 series, 805 series etc..)
- Understand hardware details
- Understand How to port OS on Telechip based hardware
- Android porting
- Linux porting

Duration: 40 hours

C1. Linux Fundamentals and BSP Introduction (1/4)

Linux Basics and Commands

- Introduction to Linux and Command-Line Interface (CLI)
- Overview of Linux and its distributions.
- Advantages of using Linux in embedded systems.
- Introduction to the Linux command shell (Bash).
- Basic File Operations
- Navigating the file system: `ls`, `cd`, `pwd`.
- File manipulation commands: `cp`, `mv`, `rm`, `mkdir`, `touch`.
- Process Management and Networking Commands
- Managing processes: `ps`, `top`, `kill`, `jobs`, `bg`, `fg`.
- Networking commands: `ifconfig`, `ping`, `netstat`, `ssh`.
- System Calls Overview
- Introduction to system calls and their importance in Linux.
- Project 1: Basic Linux Environment Setup
- Set up a Linux environment on Telechips, explore the CLI, and execute basic commands

Tools

Hardware: Telechips
Software: Linux OS (Raspbian)

Duration: 40 hours

C1. Linux Fundamentals and BSP Introduction (2/4)

Linux File System and Directory Structure

- Detailed Exploration of Linux File Systems
- Types of file systems: ext2, ext3, ext4, XFS, Btrfs.
- Mounting and unmounting file systems: `mount`, `umount`.
- Understanding the Directory Hierarchy and File Types
- Key directories and their purposes: `/bin`, `/etc`, `/home`, `/var`, `/usr`, `/proc`.
- File types and permissions: Regular files, directories, links, devices.
  Sub-Task: Navigate through the file system, understand directory structure, and permissions.

Tools

Hardware: Telechips
Software: Linux OS (Raspbian)

Duration: 40 hours

C1. Linux Fundamentals and BSP Introduction (3/4)

Linux Kernel Overview

- Overview of the Linux Kernel and Its Role in the System
- What is a kernel? The role of the kernel in managing hardware resources.
- Types of kernels: Monolithic, Microkernel, Hybrid.
- Introduction to Kernel Architecture, Processes, and Modules
- Kernel architecture: User space vs. kernel space.
- Kernel processes, kernel threads, and modules.

  Exploring Kernel Source Code Structure
- Overview of kernel source code: `init`, `drivers`, `fs`, `kernel`.
- Understanding the Linux boot process and key components.
- Sub-Task: Explore the kernel source code structure and understand the boot process.

Tools

Hardware: Telechips Software: Linux Kernel Source

Duration: 40 hours

C1. Linux Fundamentals and BSP Introduction (4/4)

Introduction to Board Support Packages (BSP)

- Understanding Board Support Packages (BSP)
  Role of BSP in embedded systems.
- Components of BSP: Bootloader, Kernel, Root File System, Device Tree.
- Role of Bootloader (U-Boot) in BSP
- Introduction to U-Boot, its configuration, and usage.
- Key files and configurations in the BSP for Telechips.
- Project 2: BSP Analysis
- Analyze the BSP components of Telechips and identify the relevant files and configurations.

Tools

Hardware: Telechips
Software: BSP, U-Boot

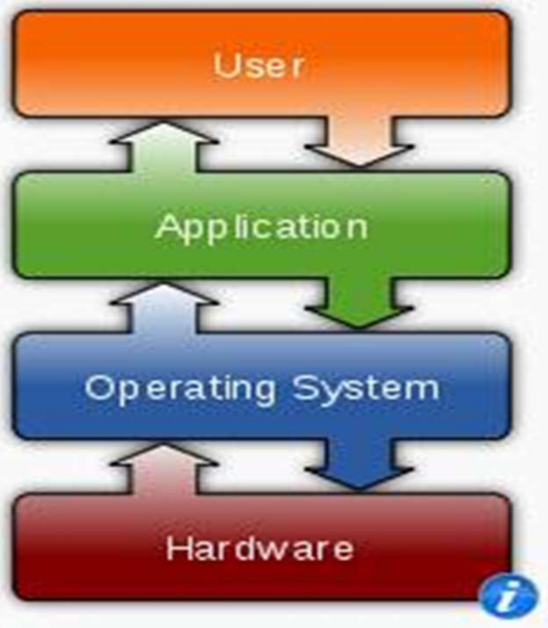
Telechip Dolphin

C1-1 : Linux Fundamentals and BSP Introduction

What is OS

- Each time when we turn on the computer, we can see and perform different activities on the system like write, read, browse the internet and watching a video.
- Well, all these activities are performed by an operating system or kernel.
- A kernel is a program which is the heart of any operating system, used to communicate between hardware and software.
- So, to work on the computer we need an operating system.

Operating systems



Common features

- Process management
- Interrupts
- Memory management
- File system
- Device drivers
- Networking (TCP/IP, UDP)
- Security (Process/Memory protection)
- I/O

Linux History

- Linux and UNIX both are operating systems written in C and Assembly languages. UNIX is a primary operating system introduced in the year 1970 by Ken Thompson and others, while it is a multitasking and multiuser operating system.
- UNIX being a proprietary networking OS, its cost is high for a mainframe system.
- The end-users of UNIX are not authorized to change the kernel in any way as it results in the violation of the license in the terms.
- Later in the year 1991, a Finnish student named Linus Torvalds introduced the initial Linux kernel, which meant a clone of the UNIX.

Why Linux for Embedded Systems?

- Linux is an open-source operating system and free for anyone to use and modify the code.
- It is freely downloaded and distributed over the internet.
- Its open-source nature reduces licensing fees, and its modularity allows for custom configurations.
- Nowadays Linux OS is widely used in day to day life like Supercomputers, desktops, smart phones, web server, home appliances, washing machines, refrigerators, cars, modems etc.

Linux origin

- 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 **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.

Why use Linux ?

- Linux OS gives a great advantage to developers, as they can custom their design in operating systems.
- It is completely trouble-free, highly reliable and stable.

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 Distribution

- Other operating systems like Microsoft combine each bit of codes internally and release it as a single package. You have to choose from one of the version they offer.
- But Linux is different from them. Different parts of Linux are developed by different organizations.
- Different parts include kernel, shell utilities, X server, system environment, graphical programs, etc. If you want you can access the codes of all these parts and assemble them yourself. But its not an easy task seeking a lot of time and all the parts has to be assembled correctly in order to work properly.
- From here on distribution (also called as distros) comes into the picture. They assemble all these parts for us and give us a compiled operating system of Linux to install and use.

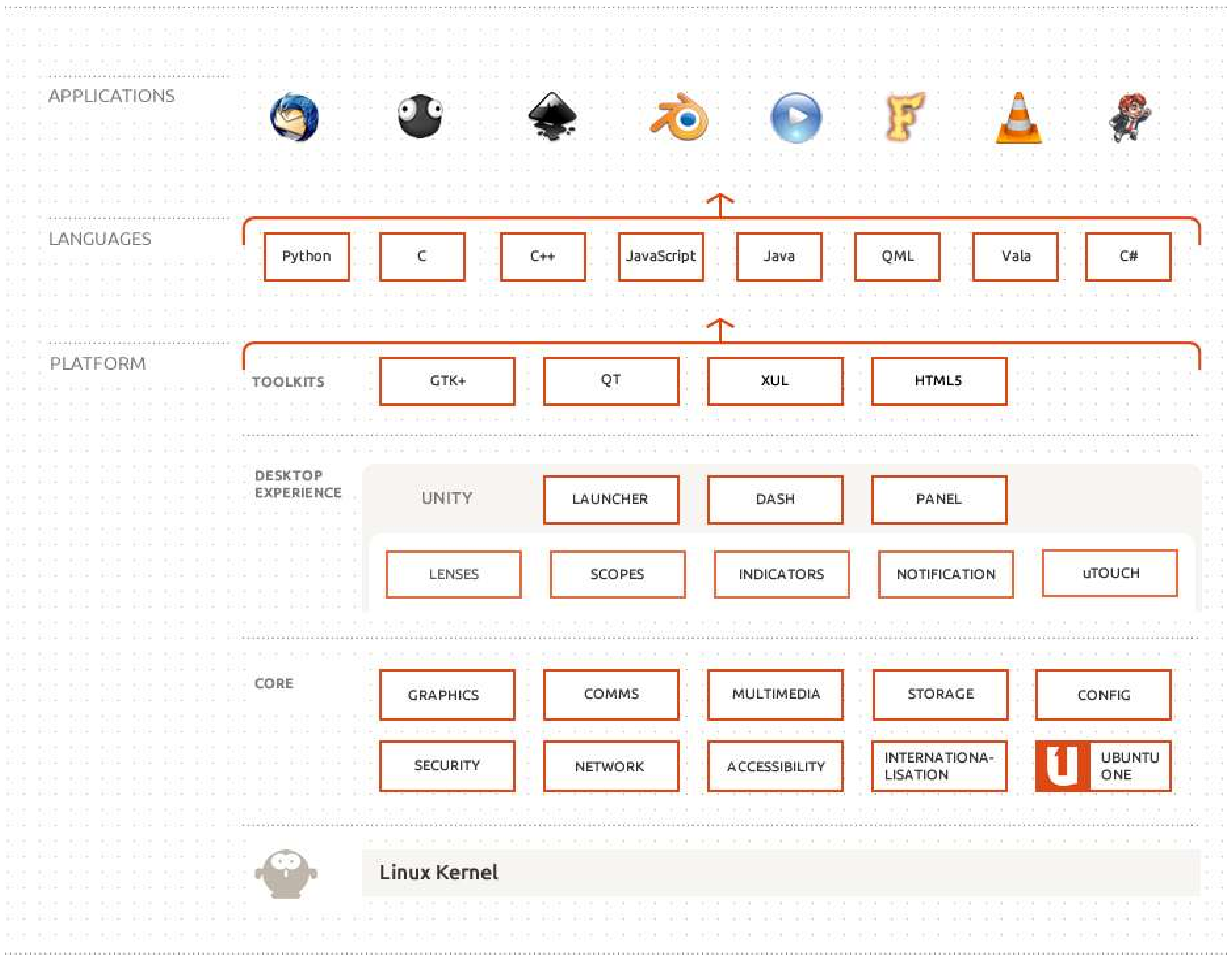
Introduction to Linux Distribution

- A Linux distribution is an OS made through a software collection that contains the Linux kernel and a package management system often.
- Usually, Linux users obtain their OS by downloading a Linux distribution, available for a range of systems from embedded devices (e.g., ***OpenWrt***) to robust supercomputers (e.g., Rocks Cluster Distribution).
- A Linux distribution is composed of a Linux kernel, GNU libraries and tools, other software, a window system, documentation, a desktop environment, and a window manager.

- Almost every added software is open-source and free and becomes available both as in source code and compiled binary form, permitting changes to the actual software.
- Optionally, Linux distributions add a few proprietary software that might not be available in the source code form, like binary blocks needed for a few device drivers.

Some Example of Linux Distro

redhat
MEPIS
turbolinux
LUNAR
EvilEntity
debian
Vine Linux
cAos/CentOS
MiniKazit
UTUTO
archlinux
m0n0wall
jamd LINUX
Knoppix STD
gentoo linux
DeLi Linux
Hiweed
amlug
slackware linux
yellow dog linux
Fedora
LPG
PLD
SLAX
COREL LINUX
Progeny
GEEXBOX
BIGLINUX
FREEDUC
Lycoris
EnGarde
Mandrakelinux
BeatrIX
Linspire
SuSE
中文延伸套件
YOPER
BearOps
ASPLINUX
kalango
Slackintosh
Frugalware
Foresight
Mint
PCLinuxOS
Haydar Linux
sabayon
ubuntu
JULEX
blag


APPLICATIONS
LANGUAGES
Python
C
C++
JavaScript
Java
QML
Vala
C#
PLATFORM
TOOLKITS
GTK+
QT
XUL
HTML5
DESKTOP EXPERIENCE
UNITY
LAUNCHER
DASH
PANEL
LENSES
SCOPES
INDICATORS
NOTIFICATION
uTOUCH
CORE
GRAPHICS
COMMS
MULTIMEDIA
STORAGE
CONFIG
SECURITY
NETWORK
ACCESSIBILITY
INTERNATIONA-LISATION
UBUNTU ONE
Linux Kernel

Linux 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. Linux Basics and Commands

Linux is a powerful and widely used operating system, especially for embedded systems. Its open-source nature, strong community support, and robust performance make it ideal for developers.

Key Features of Linux:

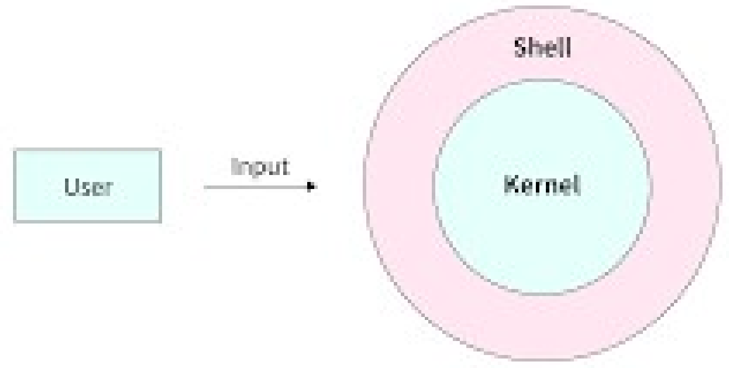
- Multi-user and multitasking capabilities
- Secure file system structure
- Strong process management
- Comprehensive networking support

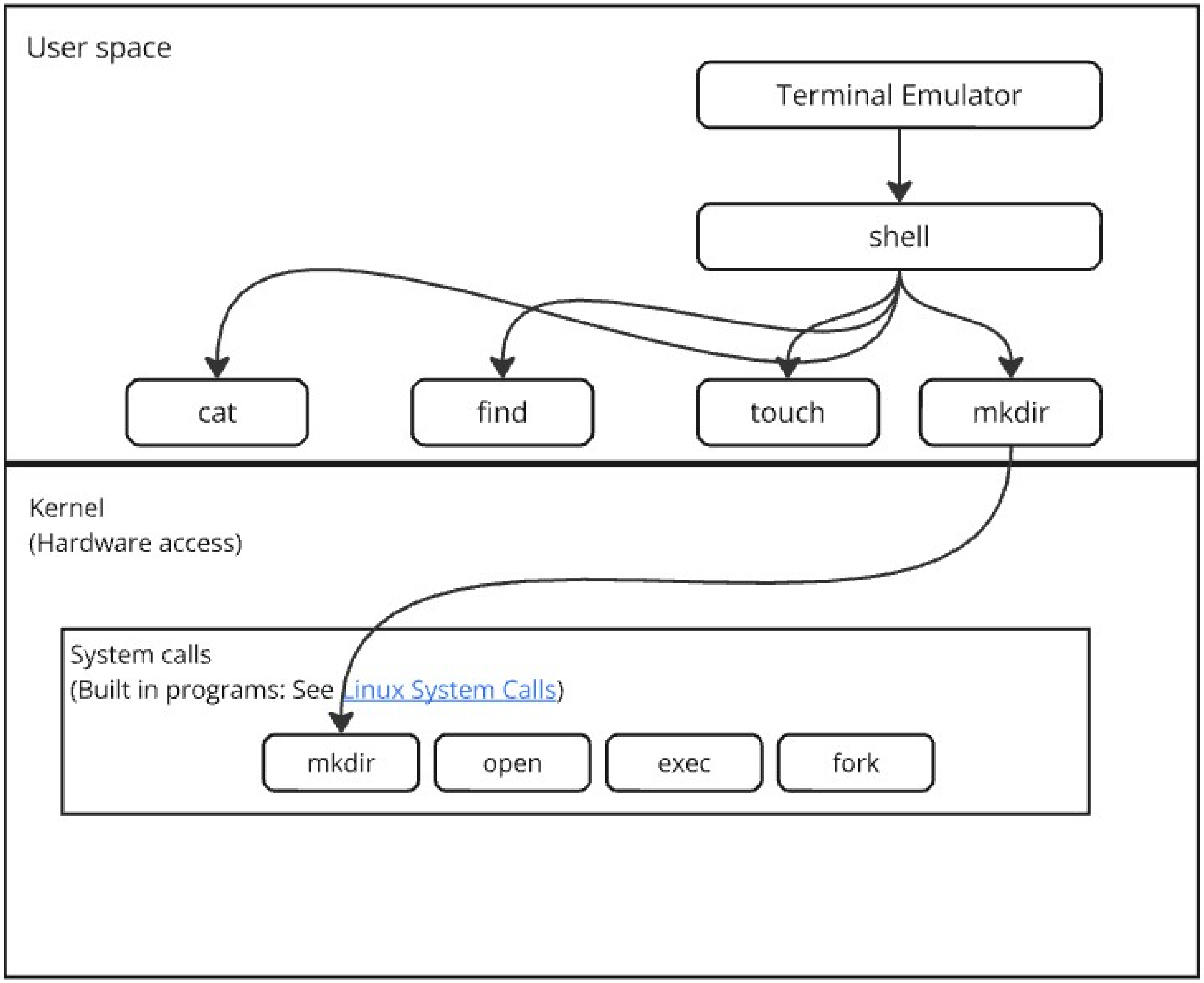
What is shell?

A program that interprets commands

 Allows a user to execute commands by typing them manually at a terminal, or automatically in programs called *shell scripts*. This is also called the command-line interface (CLI)

 A shell is *not* an operating system. It is a way to interface with the operating system and run commands. It acts as a bridge between the user and the kernel and allows the user to execute scripts, issue commands, and monitor running processes.




User space
Terminal Emulator
shell
cat
find
touch
mkdir
Kernel
(Hardware access)
System calls
(Built in programs: See Linux System Calls)
mkdir
open
exec
fork

What is BASH?

BASH = **B**ourne **A**gain **SH**ell

- Bash is a shell written as a free replacement to the standard Bourne Shell (/bin/sh) originally written by Steve Bourne for UNIX systems.

- It has all of the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line.

- since it is Free Software, it has been adopted as the default shell on most Linux systems.

How is BASH different from the DOS command prompt?

Case Sensitivity: In Linux/UNIX, commands and filenames are case sensitive, meaning that typing “EXIT” instead of the proper “exit” is a mistake.

 “\” vs. “/”: In DOS, the forward-slash “/” is the command argument delimiter, while the backslash “\” is a directory separator. In Linux/UNIX, the “/” is the directory separator, and the “\” is an escape character. More about these special characters in a minute!

 Filenames: The DOS world uses the “eight dot three” filename convention, meaning that all files followed a format that allowed up to 8 characters in the filename, followed by a period (“dot”), followed by an option extension, up to 3 characters long (e.g. FILENAME.TXT). In UNIX/Linux, there is no such thing as a file extension. Periods can be placed at any part of the filename,

Why Use Bash?

Bash (Bourne Again Shell) is the default command-line interpreter in many Linux distributions.
Why Use Bash?

1. **Command Execution:** Directly execute system commands to manage files, processes, and services.
2. **Automation:** Write scripts to perform repetitive tasks.
3. **Customization:** Personalize environment settings via configuration files (~/.bashrc).
4. **Efficiency:** Perform complex operations quickly through pipelines and shell utilities.

Key Bash Features

- Command Line Interface (CLI): Users type commands and receive output.
- Built-in Commands: Examples: echo, cd, pwd, exit.
- Environment Variables: Store system and user-specific information, e.g., $PATH, $HOME.
- Scripting Capabilities: Automate tasks using conditional statements, loops, and functions.
- Wildcards & Piping: Wildcards (*, ?) help match files and directories.
- Pipes (|) send the output of one command as input to another.

Special Characters

- Before we continue to learn about Linux shell commands, it is important to know that there are many symbols and characters that the shell interprets in special ways.
- This means that certain typed characters:

a) cannot be used in certain situations,
b) may be used to perform special operations, or,
c) must be “escaped” if you want to use them in a normal way.

Special Characters

\	Escape character. If you want to reference a special character, you must "escape" it with a backslash first. Example: *touch /tmp/filename\\**
/	Directory separator, used to separate a string of directory names. Example: */usr/src/linux*
.	Current directory. Can also "hide" files when it is the first character in a filename.

Special Characters

..	Parent directory
~	User's home directory
*	Represents 0 or more characters in a filename, or by itself, all files in a directory. Example: *pic*2002 can represent the files pic2002, picJanuary2002, picFeb292002, etc*
?	Represents a single character in a filename. Example: *hello?.txt can represent hello1.txt, helloz.txt, but not hello22.txt*
[ ]	Can be used to represent a range of values, e.g. [0-9], [A-Z], etc. Example: *hello[0-2].txt represents the names hello0.txt, hello1.txt, and hello2.txt*
|	"Pipe". Redirect the output of one command into another command. Example: *ls | more*

Special Characters

>	Redirect output of a command into a new file. If the file already exists, over-write it. Example: *ls > myfiles.txt*
>>	Redirect the output of a command onto the end of an existing file. Example: *echo .Mary 555-1234. >> phonenumbers.txt*
<	Redirect a file as input to a program. Example: *more < phonenumbers.txt*
;	Command separator. Allows you to execute multiple commands on a single line. Example: *cd /var/log ; less messages*
&&	Command separator as above, but only runs the second command if the first one finished without errors. Example: *cd /var/logs && less messages*
&	Execute a command in the background, and immediately get your shell back. Example: *find / -name core > /tmp/corefiles.txt &*

Basic File Operation Commands

Command	Description
ls	Lists directory contents
pwd	Prints the current working directory
mkdir	Creates a new directory
rm	Removes files or directories
cp	Copies files or directories
touch	Creates an empty file
echo	Displays text to the terminal

cd -> Changes the current directory .

clear -> Clears the screen.

exit -> Exits the shell or your current session.

Pipes and Re-directions

Piping Commands Together

- The pipe character, “|”, is used to chain two or more commands together. The output of the first command is “piped” into the next program, and if there is a second pipe, the output is sent to the third program, etc. For example:
- *ls -la /usr/bin | less*
- In this example, we run the command “ls -la /usr/bin”, which gives us a long listing of all of the files in /usr/bin. Because the output of this command is typically very long, we pipe the output to a program called “less”, which displays the output for us one screen at a time.

Pipes and Re-directions

Redirecting Program Output to Files

- There are times when it is useful to save the output of a command to a file, instead of displaying it to the screen. For example, if we want to create a file that lists all of the MP3 files in a directory, we can do something like this, using the “>” redirection character:
- *ls -l /home/vic/MP3/*.mp3 > mp3files.txt*
- A similar command can be written so that instead of creating a new file called mp3files.txt, we can append to the end of the original file:
- *ls -l /home/vic/extraMP3s/*.mp3 >> mp3files.txt*

Basic File Operations

Command	Description
cat	Displays file contents
cp	Copies a file
mv	Moves or renames a file
rm	Deletes a file
touch	Creates an empty file
nano	Text editor in CLI
file	Find out what kind of file it is.

Command	Description
head	Display the first few lines of a text file.
tail	Display the last few lines of a text file.
mkdir	Make Directory. *Example: mkdir /tmp/myfiles/*
rmdir	Remove Directory. *Example: rmdir /tmp/myfiles/*

Finding things

Command	Description
which	Shows the full path of shell commands found in your path. *For example, if you want to know exactly where the "grep" command is located on the filesystem, you can type "which grep". The output should be something like: /bin/grep.*
whereis	Locates the program, source code, and manual page for a command (if all information is available). *For example, to find out where "ls" and its man page are, type: "whereis ls". The output will look something like:* *ls: /bin/ls /usr/share/man/man1/ls.1gz*
locate	A quick way to search for files anywhere on the filesystem. *For example. You can find all files and directories that contain the name "mozilla" by typing: locate mozilla*

Finding things

Command	Description
Find	A very powerful command, but sometimes tricky to use. It can be used to search for files matching certain patterns, as well as many other types of searches. *A simple example is:* *find . –name \*mp3*
Grep	The grep command in Linux is a sorting utility used to search for text patterns in files.

Process Management and Informational Commands

ps	Displays current processes
top	Shows real-time system processes
kill	Terminates a process
jobs	Lists background jobs
bg	Resumes a background job
fg	Brings a background job to the foreground
ps	Displays current processes
id	Print your user id and group id’s

w	*Show who is logged on and what they are doing.*
df	(Disk free cmd) Report file system disk space usage.
du	**“Disk Usage”** in a particular directory. “du -s” provides a summary for the current directory.
cat /proc/cpuinfo	Displays lots of information about current memory usage.
uname -a	Prints system information to the screen (kernel version, machine type, etc)

10. Managing processes ps, top, kill, jobs, bg, fg

- **ps (Process Status)**
  The ps command displays information about running process.
  Common Options:
  ps: Lists processes for the current user in the current terminal.
  ps -aux: Displays detailed process information for all users.
  Example Usage:
  ```
  ps -aux | grep firefox
  ```
  This shows all processes containing "firefox" in the process name.

- **kill (Terminate a Process)**
  The kill command sends signals to processes to terminate or control them.
  Common Signals:
  kill <PID>: Sends the SIGTERM (15) signal to terminate a process gracefully.
  kill -9 <PID>: Forcefully kills a process (SIGKILL).
  Example Usage:
  ```
  kill 1234      # Gracefully terminate process with PID 1234
  kill -9 5678  # Forcefully kill process with PID 5678
  ```

.

- **top (Real-Time Process Monitoring)**
  The top command provides a dynamic, real-time view of running processes, including resource usage.
  Key Controls:
  q: Quit top.
  k: Kill a process by entering its PID.
  Shift + P: Sort by CPU usage.
  Shift + M: Sort by memory usage.
  Example Usage:
  ```
  Top
  ```
- **bg (Resume Job in Background)**
  The bg command resumes a stopped job in the background.
  Example Usage:
  ```
  bg %1
  ```
  This resumes job 1 in the background.
- **jobs (List Background Jobs)**
  The jobs command displays background jobs started by the current shell.
  Example Usage:
  ```
  jobs
  ```
  Output example:
  ```
  [1]+ Running    ./script.sh &
  [2]- Stopped    ./long_task.sh
  ```
- **fg (Resume Job in Foreground)**
  The fg command resumes a job in the foreground, bringing it back to the terminal.
  Example Usage:
  ```
  fg %2
  ```
  This brings job 2 to the foreground

Process Management and Networking Commands

Networking Commands:

ifconfig	Displays network interface information
ping	Checks connectivity to a host
netstat	Displays network connections

11. Networking commands: ifconfig, ping, netstat sah

1. ifconfig

Purpose: Displays and configures the network interfaces in Unix/Linux systems.

Common usage:

- ifconfig — Show all network interfaces and their configurations
- ifconfig eth0 up — Enable the eth0 network interface
- ifconfig eth0 down — Disable the eth0 network interface
- ifconfig eth0 192.168.1.10 netmask 255.255.255.0 — Assign an IP address and subnet mask to eth0

2. ping

Purpose: Checks network connectivity by sending ICMP echo requests to a target.

Common usage:

- ping google.com — Send continuous ping requests to Google
- ping -c 4 192.168.1.1 — Send 4 packets to the specified IP
- ping -i 2 example.com — Send ping requests every 2 seconds
- ping -s 100 example.com — Set packet size to 100 bytes

3. netstat

Purpose: Displays active network connections, routing tables, and interface statistics.
Common usage:

- netstat -a — Show all active connections
- netstat -r — Display the kernel routing table
- netstat -tuln — Show listening ports (TCP and UDP) without resolving names
- netstat -i — Display network interface statistics

Other utilities Commands

echo	Display text on the screen. Mostly useful when writing shell scripts. *For example: echo “hello World”*
more	Display a file, or a program output one page at a time. *examples: more mp3files.txt* *ls -la | more*
less	An improved replacement for the “more” command. Allows you to scroll backwards as well as forwards.
sort	Sort a file or program output. *Example: sort mp3files.txt*
su	**“Switch User”.** Allows you to switch to another user’s account temporarily. The default account to switch to is the root/superuser account. Examples: su - Switch the root account. su - - Switch to root, and log in with root’s environment. su kernel - Switch to kernel account.

Practice =>

- Practice Commands
- Practice Simple Shell Script

System Calls Overview

System calls are the interface between user applications and the operating system (OS). They allow programs to request services like hardware access, file manipulation, and process management from the OS kernel.

Types of System Calls:

1. **Process Control:**
   - Create or terminate processes
   - Load, execute programs
   - Example: fork(), exec(), exit()
2. **File Management:**
   - Open, read, write, close files
   - Create or delete files
   - Example: open(), read(), write(), close()

1. **Device Management:**
   - Request and release device access
   - Read/write operations on devices
   - Example: ioctl(), read(), write()
2. **Information Maintenance:**
   - Get/set system data (time, date, etc.)
   - Example: getpid(), gettimeofday(), uname()
3. **Communication:**
   - Data transfer between processes
   - Example: pipe(), shmget(), socket()

Key Features:

- Abstract hardware complexities
- Offer controlled resource access
- Ensure system security and stability

Practice =>

- Check systemcall

Project 1
a. Basic Linux Environment Setup

Basic Linux Environment Setup

A. Download and Install a Linux Distribution:

Common distributions include Ubuntu, Debian, or Fedora.
Download the ISO file from the official website of your chosen distribution and create a bootable USB (use tools like Rufus or BalenaEtcher).

A. Set up the Linux Environment:

Boot your machine using the USB and follow the installation steps.
Select necessary options like partitioning, user credentials, and timezone configuration.

A. Post-Installation Configuration:

Update packages:

```
sudo apt update && sudo apt upgrade
```

Install essential utilities:

```
sudo apt install build-essential git vim curl
```

Project 1
b. Set up a Linux Environment on Telechips

1. **Get the BSP (Board Support Package) for Telechips:**
   Visit the official Telechips support portal to download the appropriate BSP and firmware for your target board.
1. **Flash Linux onto Telechips:**
   Use tools like fastboot or dd to flash Linux images onto the board:
   ```
   sudo dd if=<image_file>.img of=/dev/sdX bs=4M
   ```
   Replace /dev/sdX with the correct device name for your target hardware.
1. **Connect to the Board:**
   Use UART or serial communication tools like minicom to interface with the board:
   ```
   sudo minicom -b 115200 -D /dev/ttyUSB0
   ```

5. **Check System Information**:
   View kernel version:
   ```
   uname -a
   ```
   Disk usage:
   ```
   df -h
   ```

Shortcuts

Up/Down Arrow Keys	Scroll through your most recent commands. You can scroll back to an old command, hit Enter, and execute the command without having to re-type it.
history	Show your complete command history.
history with Ctrl+R	Press CTRL-R and then type and portion of a recent command. It will search the commands for you, and once you find the command you want, just press ENTER.
Shift+PgUp Shift+pgDn	If you type a particular command or file name that the shell recognizes, you can have it automatically completed for you if you press the TAB key. Try typing the first few characters of your favorite Linux command, then hit TAB a couple of times to see what happens.
TAB Completion	**“Switch User”.** Allows you to switch to another user’s account temporarily. The default account to switch to is the root/superuser account. Examples: su - Switch the root account. su - - Switch to root, and log in with root’s environment.

1. **Explore the CLI and Execute Basic Commands:**

   Common commands to explore:

```
pwd          # Print working directory
ls           # List directory contents
cd <path>    # Change directory
mkdir dir    # Create a new directory
touch file   # Create an empty file
cat file     # View file content
sudo reboot  # Reboot the system
```

   Use package management:

```
sudo apt install <package_name>
```

C1-2 : Linux Fundamentals and BSP Introduction

Linux File System And Directory Structure

Linux File System.

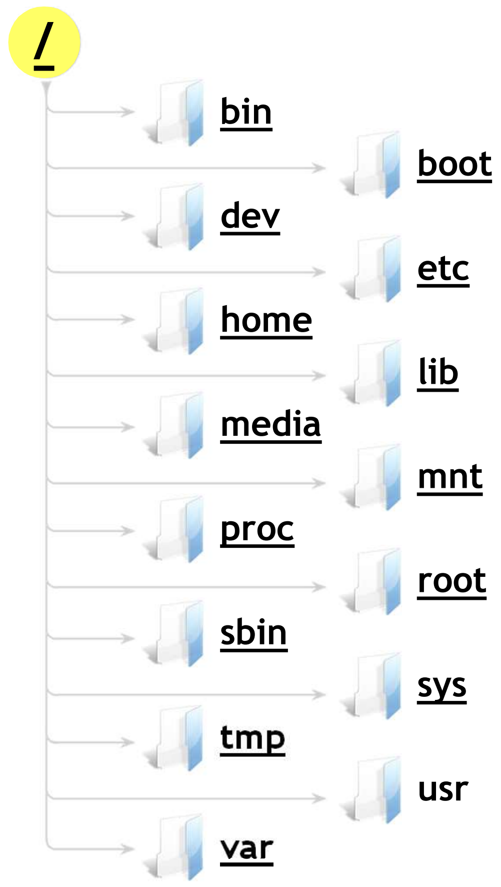
What is a File System?

A **file system** is a method used by the OS to store, organize, and manage data on storage devices.
When it is the matter of data, different operating system uses different approaches to organize it
In unix and linux the file system follows a **hierarchical directory structure** with / as the root.
The access of the data are done using a process called as mounting
The location on where the data should be located called as mount point is to be informed to the OS.

Core Features of Linux File System

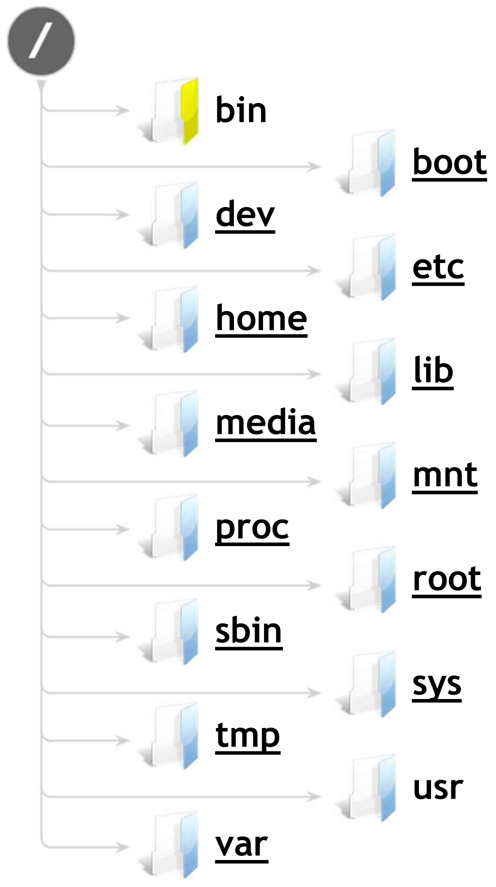
- **Everything is a file** (regular files, directories, devices, processes).
- **Hierarchical structure**: Root (/) at the top, subdirectories below.
- **Multi-user environment**: Each user has a separate home directory and permissions.
- **Journaling**: Prevents data corruption (used in ext3, ext4, XFS).
- **Inodes**: Data structures storing metadata (file size, permissions, timestamps).
- **Symbolic and Hard Links**: Reference files without duplication.

Linux Directory Structure



- Linux organizes files and directories in a hierarchical structure, starting from the root directory (/). This structure is standardized by the Filesystem Hierarchy Standard (FHS), ensuring consistency across Linux distributions. Below is a detailed explanation of the directory hierarchy and file types in Linux.

Linux Directory Structure

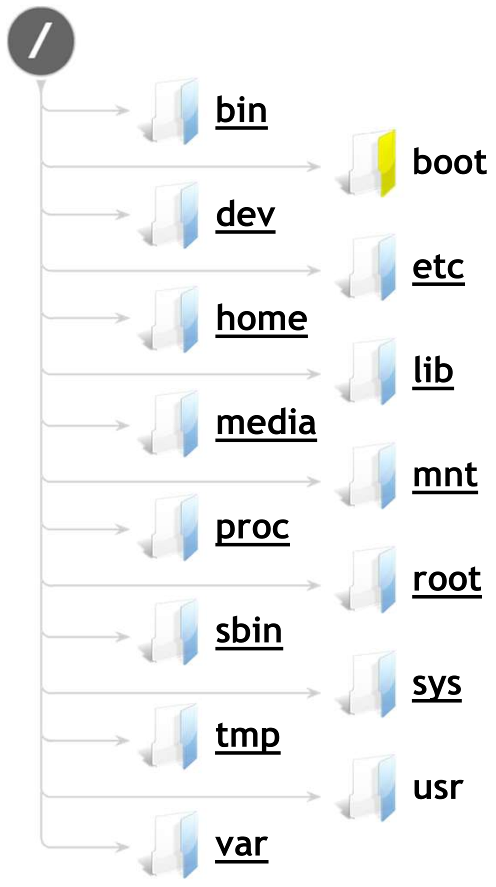


- Place for most commonly used terminal commands
- Common Linux commands you need to use in single-user modes are located under this directory.

- Commands used by all the users of the system are located here.

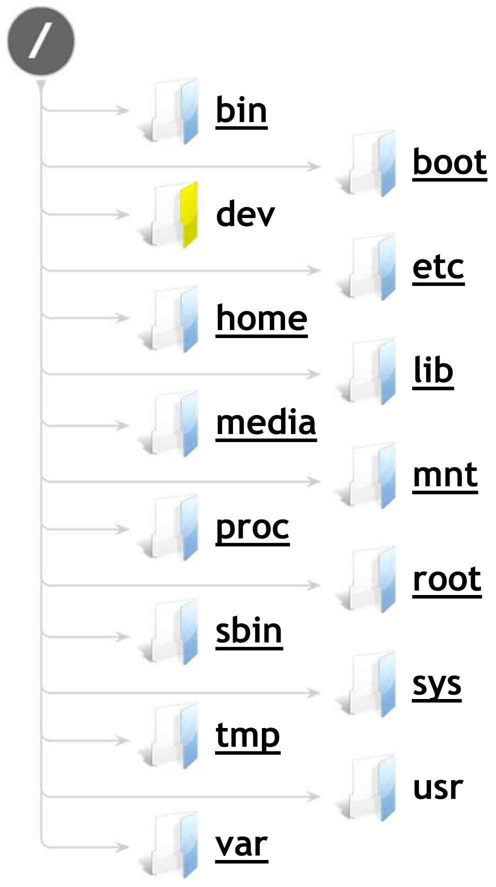
- Examples:
  - ls
  - ping
  - grep
  - cp

Linux Directory Structure



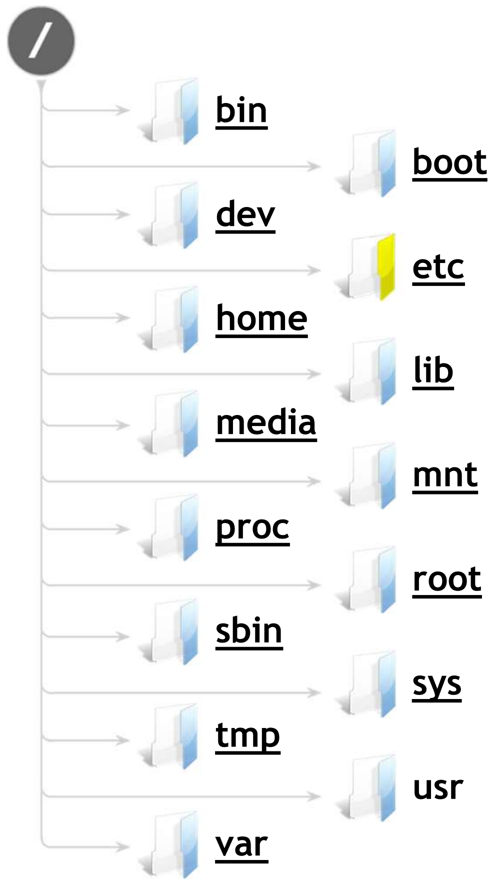
- Contains files needed to start up the system, including the Linux kernel, a RAM disk image and bootloader configuration files
- Kernel image (only when the kernel is loaded from a file system, not common on non-x86 architectures)

Linux Directory Structure



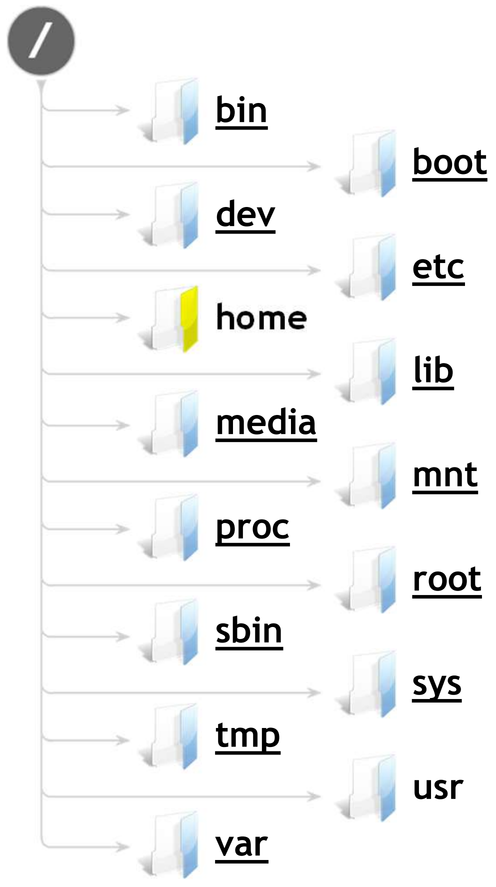
- Device files
- These include terminal devices, usb, or any device attached to the system.
- Examples:
  - /dev/tty1
  - /dev/usbmon0

Linux Directory Structure



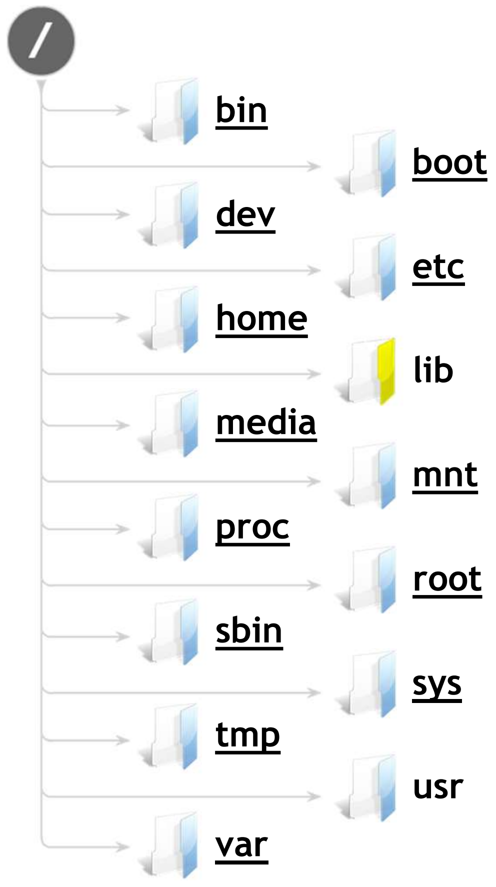
- Contains configuration files required by all programs.
- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- Examples:
  - /etc/resolv.conf
  - /etc/logrotate.conf

Linux Directory Structure



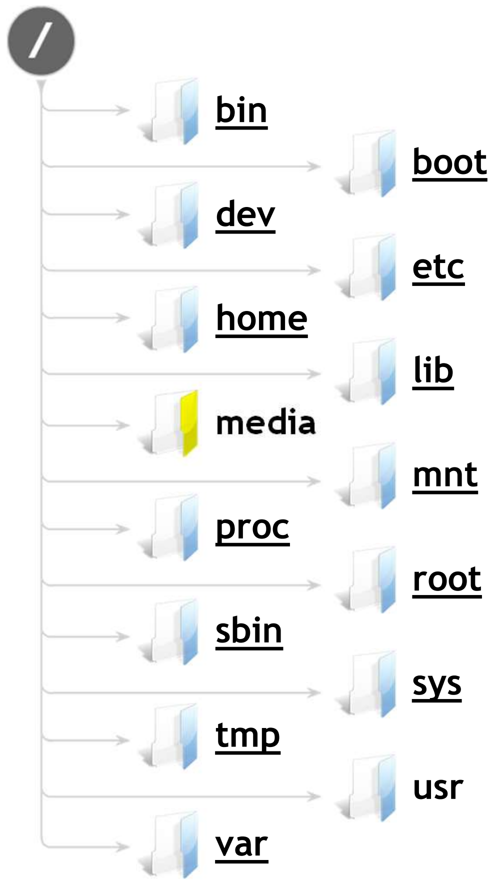
- Home directories for all users to store their personal files.
- Example
  - /home/USER1
  - /home/USER2

Linux Directory Structure



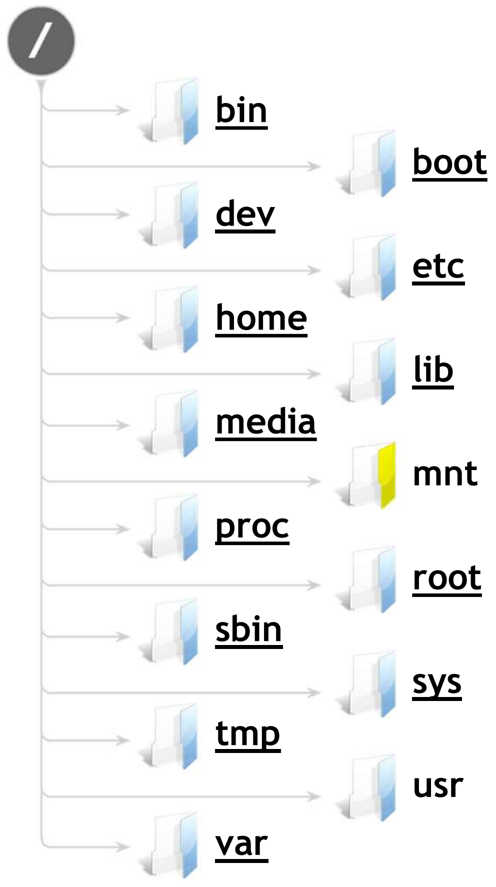
- Contains library files that supports the binaries located under /bin and /sbin
- Library file names are either ld* or lib*.so.*
- Examples
  :
  - ld-2.11.1.so
  - libncurses.so.5.
    7

Linux Directory Structure



- Temporary mount directory for removable devices.
- Examples
  - /media/cdrom
  - /media/floppy
  - /media/cdrecorder

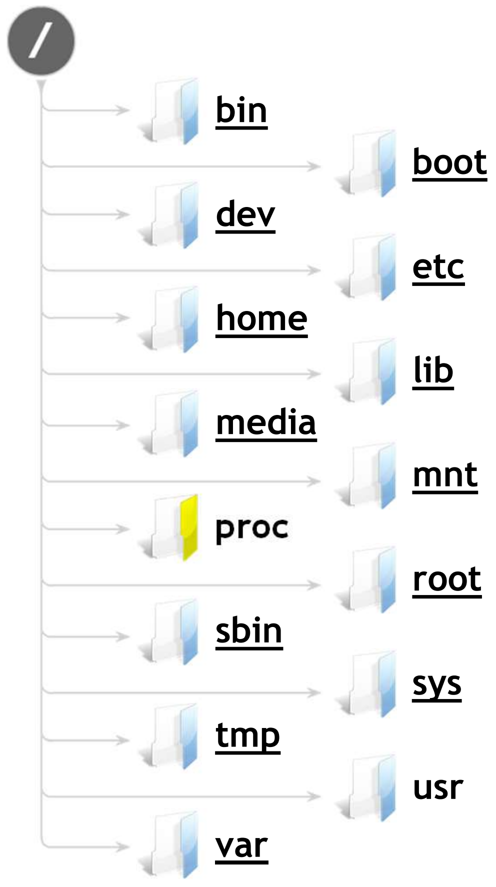
Linux Directory Structure



•

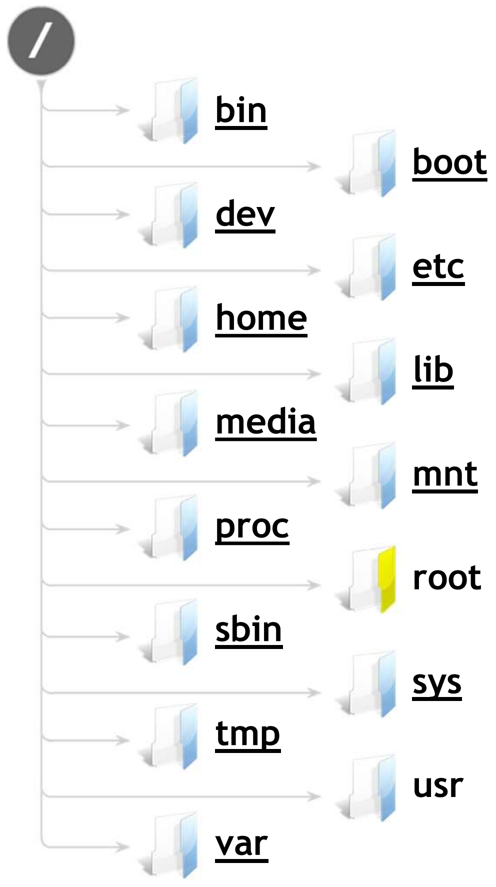
Temporary mount directory where sys admins can mount file systems.

Linux Directory Structure



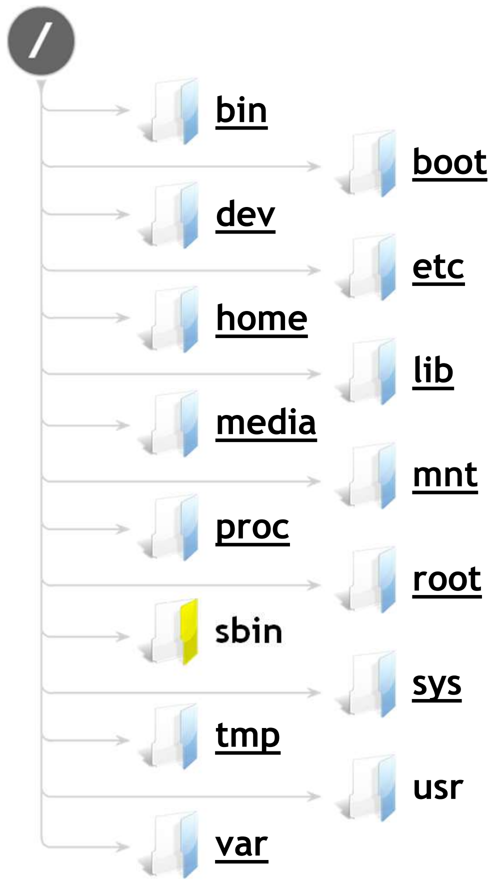
- Contains information about system process.
- This is a pseudo file system contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
- This is a virtual file system with text information about system resources. Examples:
  - /proc/uptime

Linux Directory Structure



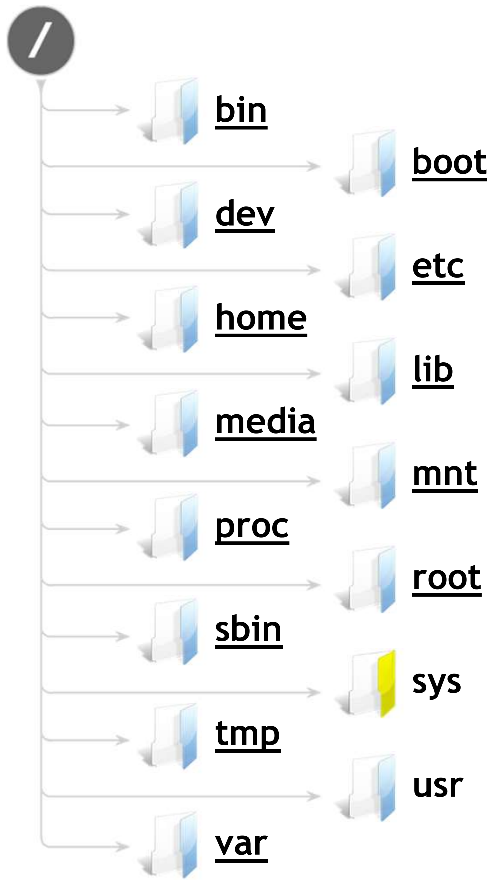
- Root user's home directory

Linux Directory Structure



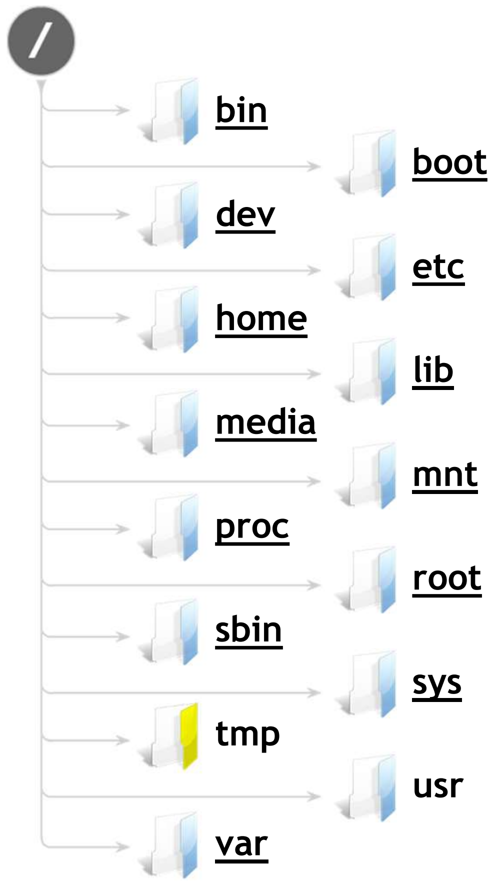
- Just like /bin, /sbin also contains binary executables
- But, the Linux commands located under this directory are used typically by system administrator, for system maintenance purpose.
- Examples
  - iptables reboot fdisk
  - ifconfig swapon
  -
  -
  -

Linux Directory Structure



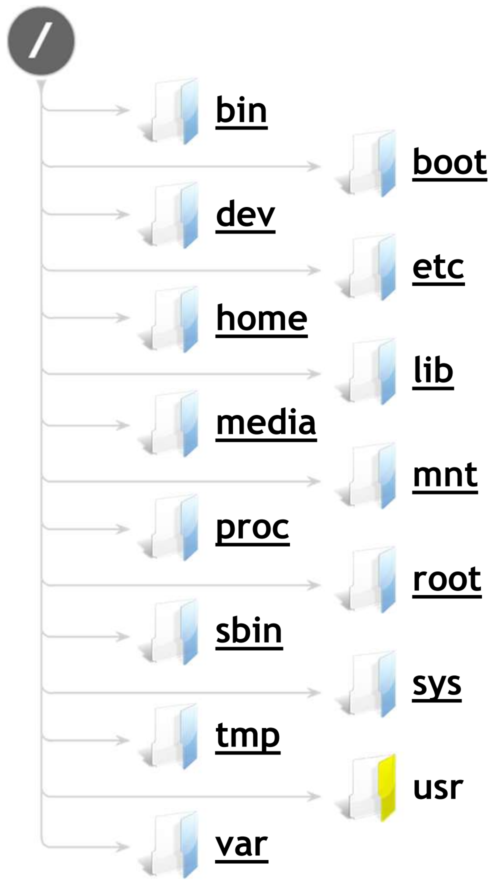
- Mount point of the sysfs virtual file system

Linux Directory Structure



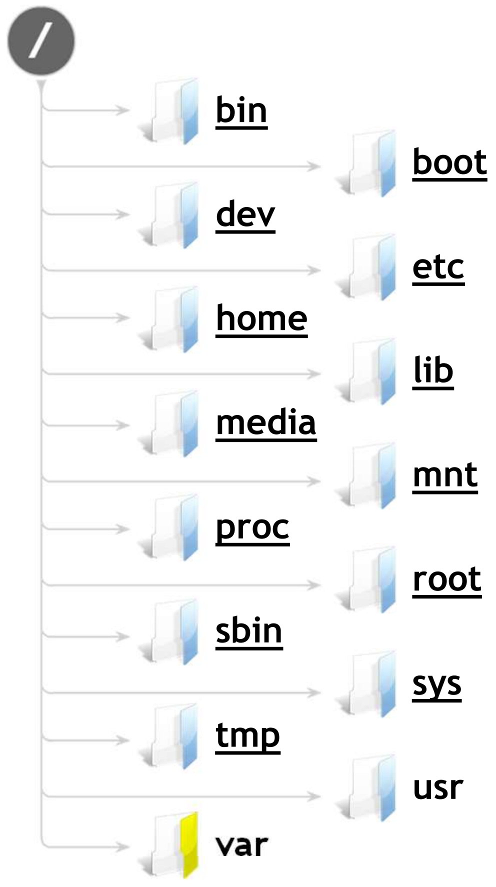
- Directory that contains temporary files created by system and users.
- Files under this directory are deleted when system is rebooted.

Linux Directory Structure



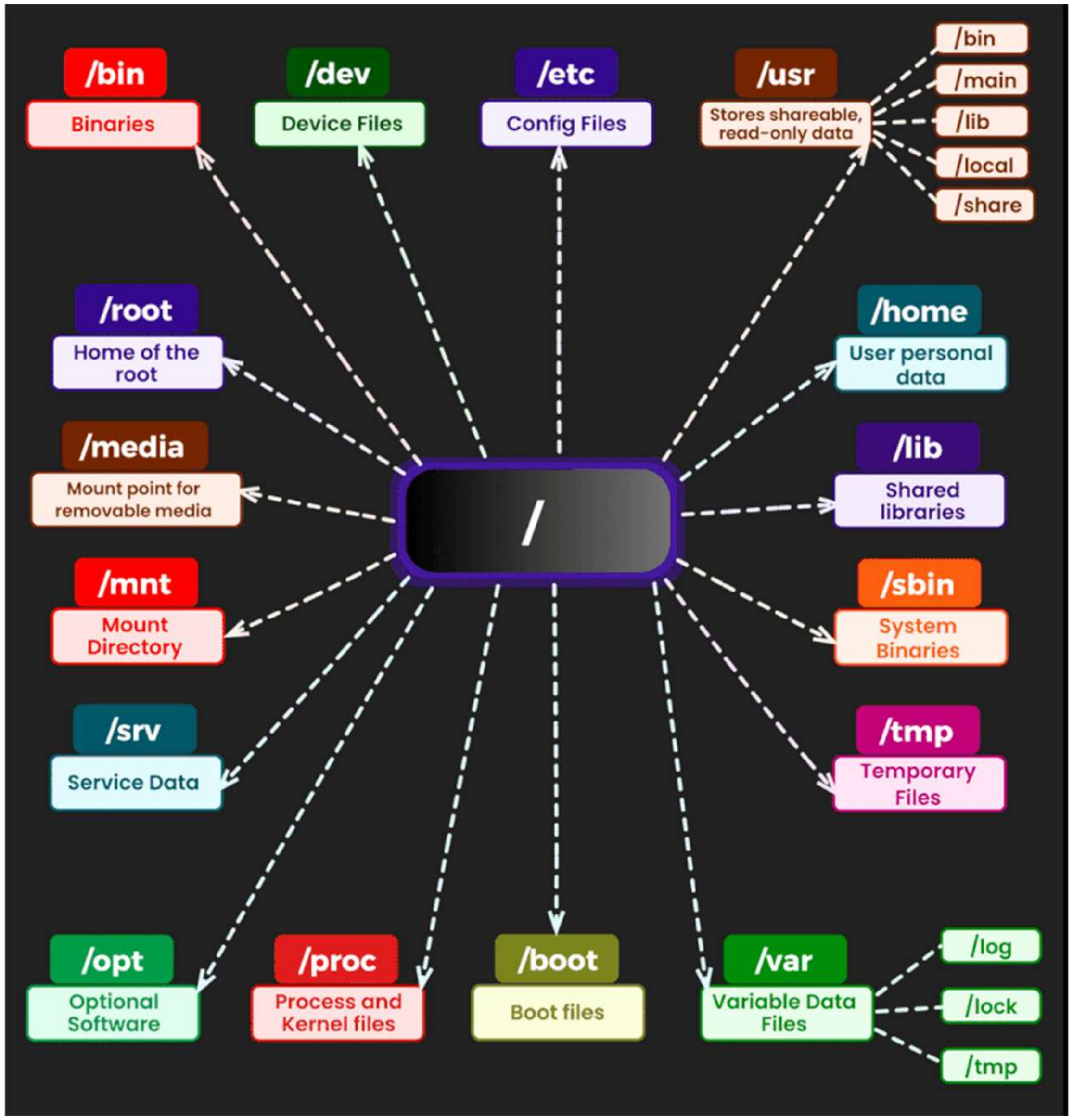
- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can't find a user binary under /bin, look under /usr/bin. For Examples: at, awk, cc, less, scp
- /usr/sbin contains binary files for system administrators. If you can't find a system binary under /sbin, look under /usr/sbin. For Examples: atd, cron, sshd, useradd, userdel
- /usr/lib contains libraries for /usr/bin and /usr/sbin
- /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2

Linux Directory Structure



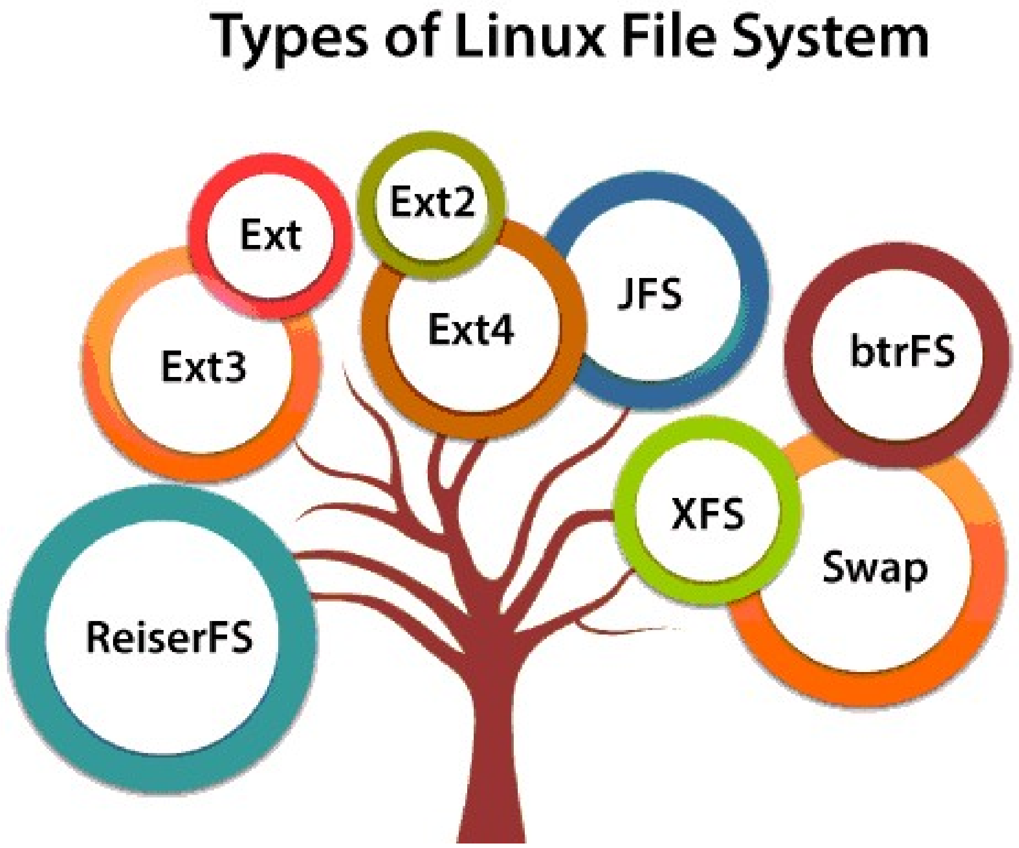
- var stands for variable files.
- Content of the files that are expected to grow can be found under this directory.
- This includes —
  - /var/log - system log files
  - /var/lib - packages and database
  - files
  - /var/mail - emails
  - /var/spool - print queues
  - /var/lock - lock files

    /var/tmp - temp files needed across reboots


/
/bin
Binaries
/dev
Device Files
/etc
Config Files
/usr
Stores shareable, read-only data
/bin
/main
/lib
/local
/share
/root
Home of the root
/home
User personal data
/media
Mount point for removable media
/lib
Shared libraries
/mnt
Mount Directory
/sbin
System Binaries
/srv
Service Data
/tmp
Temporary Files
/opt
Optional Software
/proc
Process and Kernel files
/boot
Boot files
/var
Variable Data Files
/log
/lock
/tmp

File Systems Types in Linux :-

File System Formats Introduction



RAM File Systems - initramfs (Initial RAM File System)

•**initramfs** is a **temporary root file system** loaded into RAM during the Linux boot process.
•It is used to prepare the system for mounting the real root file system.

Purpose
•**Hardware Initialization**: Loads necessary drivers and modules to access the root file system (e.g., for encrypted disks or RAID arrays).
•**Root File System Preparation**: Provides tools to mount the real root file system.

How It Works
1.The bootloader (e.g., GRUB) loads the kernel and initramfs into memory.
2.The kernel unpacks initramfs into a tmpfs file system.
3.The init script inside initramfs performs hardware detection, loads modules, and mounts the real root file system.
4.Once the real root file system is mounted, the system switches to it and continues booting.

RAM File Systems - proc (Process Information File System)

Overview
•**proc** is a **virtual file system** that provides information about running processes and system resources.
•It is mounted at /proc and is dynamically generated by the kernel.

Features
•**Process Information**: Each running process has a directory under /proc/<PID> containing details like memory usage, ope
files, and environment variables.
•**System Information**: Files like /proc/cpuinfo, /proc/meminfo, and /proc/version provide hardware and system details.

•**Kernel Parameters**: Files under /proc/sys allow tuning kernel parameters at runtime.
Use Cases
•**Debugging**: Inspect process behavior and system state.
•**Monitoring**: Monitor system performance and resource usage.
•**Kernel Tuning**: Modify kernel parameters dynamically.

RAM File Systems - sys (sysfs File System)

Overview

- **sysfs** is a **virtual file system** that exports kernel data structures to user space.
- It is mounted at /sys and provides a hierarchical view of kernel objects like devices, drivers, and modules.

Features

- **Device Information**: Provides details about hardware devices and their drivers.
- **Kernel Objects**: Exposes kernel objects like buses, classes, and power management.
- **Hotplug Events**: Used for managing hotplug devices (e.g., USB devices).

Use Cases

- **Device Management**: Inspect and configure hardware devices.
- **Kernel Development**: Debug and interact with kernel objects.
- **Power Management**: Control system power states.

Disk (Block) File System Formats – EXT2

ext2 (Second Extended File System)

•**Description**: The **ext2** file system was the default file system for Linux in the 1990s.

•**Features**:

- Simple and reliable.
- No journaling (see below for explanation).

•**Use Case**: Suitable for small partitions or systems where journaling is not required (e.g., embedded systems)

Disk (Block) File System Formats – EXT3

ext3 (Third Extended File System)

•**Description**: An improvement over **ext2**, **ext3** introduced (logs) **journaling**.

•**Features**:

- **Journaling**: Keeps track of changes in a journal before committing them to the main file system. This improves reliability and reduces the risk of data corruption during crashes.
- Backward compatible with ext2.

•**Use Case**: General-purpose file system for desktops and servers.

Disk (Block) File System Formats – EXT4

ext4 (Fourth Extended File System)

•**Description**: The successor to **ext3**, **ext4** is the most widely used file system in Linux today.

•**Features**:

- Supports larger file sizes (up to 16 TB) and larger volumes (up to 1 EB).
- Improved performance and scalability.
- **Delayed allocation**: Reduces fragmentation by delaying the allocation of blocks until data is written to disk.
- **Journaling**: Like ext3, but more efficient.

•**Use Case**: Default file system for most Linux distributions.

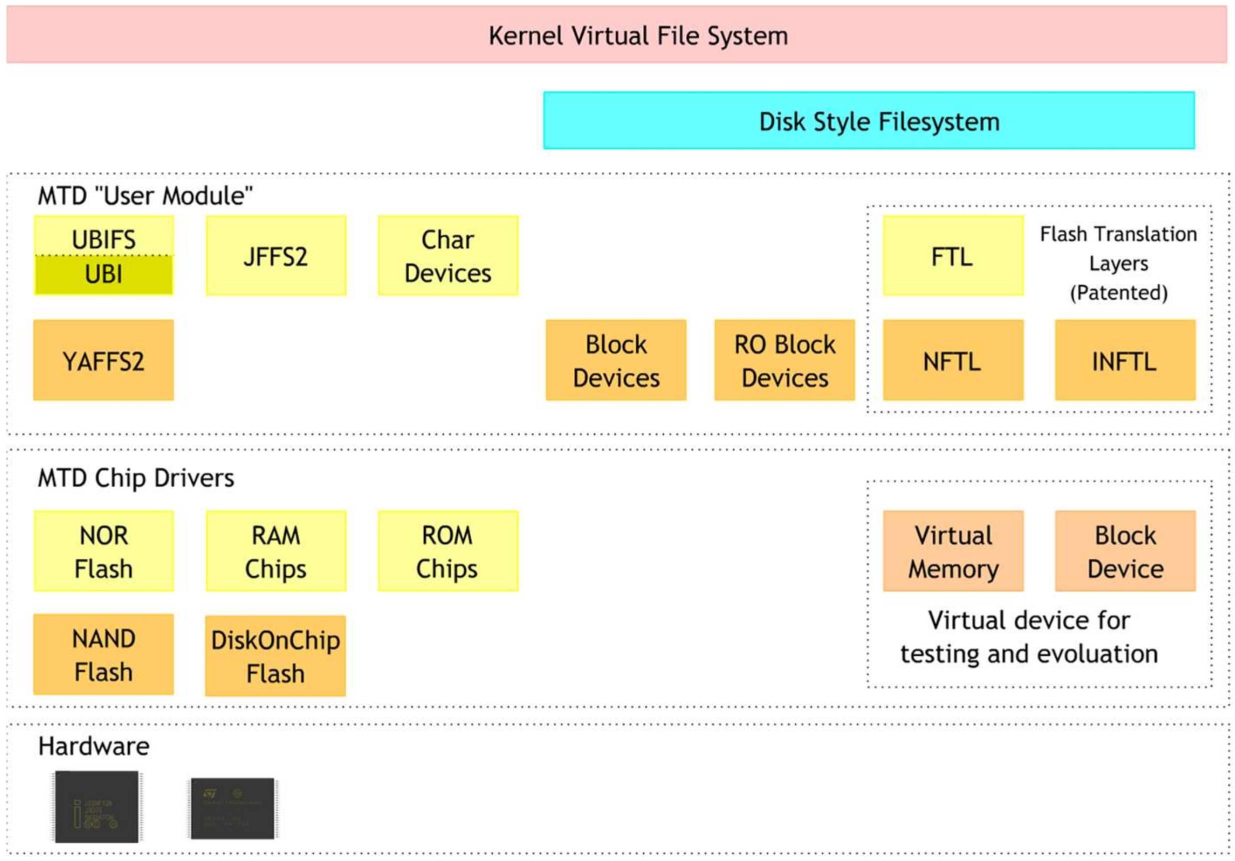
Flash File System

- Flash file system is designed for storing files on flash memory–based storage devices
- They are optimized for the nature and characteristics of flash memory for the following reasons
  - Erasing blocks
  - Random access
  - Wear leveling
- Examples
  - NAND and NOR
- Flashes
  - In Linux the flash memories use a device file called MTD for interaction

Flash File System - MTD

- MTD stands form Memory Technology Device
- Created to have abstraction layer between the hardware specific device drivers and higher-level applications
- MTD helps to use the same API with different flash types and technologies
- MTD does not deal with block devices like MMC, eMMC, SD, Compact Flash etc., Since these are no raw flashes but they have Flash Translation layer. The FTL make these devices look like block devices

Flash File System - MTD



Flash File System - JFFS2 (Mostly for NOR Flash)

JFFS2 (Journaling Flash File System 2)
Overview
- **JFFS2** is the successor to **JFFS** and is widely used in embedded systems.
- It improves upon JFFS with better performance and scalability.

Features
- **Journaling**: Ensures data integrity.
- **Wear Leveling**: Extends flash memory lifespan.
- **Compression**: Supports multiple compression algorithms (e.g., zlib, LZO).
- **Garbage Collection**: Reclaims space from deleted or obsolete data.

Use Cases
- Embedded Linux systems.
- Devices with NAND or NOR flash memory.

Limitations
- Not ideal for large storage devices.
- Requires more RAM for metadata management.

Flash File System - YAFFS (Mostly for NAND Flash)

YAFFS (Yet Another Flash File System)
Overview
- **YAFFS** is designed specifically for **NAND flash memory**.
- It is widely used in embedded systems and mobile devices.

Features
- **Wear Leveling**: Extends the lifespan of NAND flash.
- **Bad Block Management**: Handles bad blocks in NAND flash.
- **No Journaling**: Uses a different mechanism for data integrity.
- **Fast Mount Times**: Optimized for quick startup.

Use Cases
- Mobile devices (e.g., smartphones, tablets).
- Embedded systems with NAND flash.

Limitations
- Primarily designed for NAND flash, not suitable for other storage types.
- Limited support for advanced features like compression.

Special Purpose File System

- Sometimes the way we handle data might depend on the requirements
- There some file system types which can offer different features like
    - Compressions
    - Backups
    - Access etc.

Special Purpose File System - squashfs

- Read-only compressed file system for block devices Can be used for read only data like binaries, kernel etc
- Very good compression rate and read access performance Mostly used in live CDs and live USB distros
- Can support compressions like
  - gzip
  - L0ZM
  - A
  - LZO etc

Special Purpose File System - tmpfs

Overview

- **tmpfs** is a **memory-based file system** that uses RAM (and optionally swap space) for storage.
- It is volatile, meaning all data is lost when the system is rebooted or powered off.

Features

- **Fast Access**: Since it resides in RAM, access speeds are extremely fast.
- **Dynamic Sizing**: Grows and shrinks automatically based on usage.
- **Volatile Storage**: Data is not persistent across reboots.

Use Cases

- **Temporary Files**: Storing temporary files (e.g., /tmp).
- **Caching**: Used for caching data that doesn’t need to persist.
- **Shared Memory**: Used for inter-process communication (IPC).

Practice

Show All the filesystem types in linux

Linux System

- Two major partitions
  - Data partition – Linux data partition
  - A root partition and one pr more data partition
- Swap partition - Expansion of physical memory, extra memory on storage (like virtual RAM)
- Rarely used in embedded system because of the nature of the storage devices used

File Types in Linux

Linux supports several file types, each identified by a specific character in the output of `ls -l`.

Symbol	File Type	Description
-	Regular File	Normal files (e.g., text files, images, binaries).
d	Directory	A folder containing other files or directories.
l	Symbolic Link	A shortcut to another file or directory (similar to a Windows shortcut).
c	Character Device	A device file that communicates byte-by-byte (e.g., /dev/tty).
b	Block Device	A device file that communicates in blocks (e.g., /dev/sda for disks).
p	Named Pipe	A file used for inter-process communication (IPC).
s	Socket	A file used for network communication between processes.

File Types and Permissions in Linux & BSP

Linux treats **everything as a file**, whether it’s a **text file, directory, device, or even a process**.

File Types in Linux :-

1) **Regular Files (-)**

- These are **normal files** that store **text, binary data, programs, scripts, and logs**.
  Command for Checking Regular Files: ls -l file.txt

2) **Directories (d)**

- A **directory** is a special type of file that stores the names and locations of other files.
- It acts like a **container** for files and subdirectories.
  Example: ls -ld /home

3) **Symbolic Links (l)**

- A **symlink (soft link)** is a shortcut or pointer to another file or directory.
- It allows accessing the original file **without duplicating its content**.

Symlinks

Symbolic Links (Symlinks)

What is a Symlink?

- A **symbolic link** (or **symlink**) is a special file that points to another file or directory.
- It acts as a shortcut or reference to the target file/directory.
- If the target file is deleted, the symlink becomes **broken** (dangling).

How to Create a Symlink

Use the ln command with the -s (symbolic) option:

Symlinks

Hard Links

What is a Hard Link?

•A **hard link** is a direct reference to the **inode** of a file.
•It is essentially another name for the same file.
•If the original file is deleted, the hard link still points to the file's data (as long as at least one hard link exists).

How to Create a Hard Link

Use the ln command **without** the -s option:

Device Files (Block & Character Devices)

- Linux represents **hardware devices as files** inside /dev/.

I) Block Devices (b)

- Used for **storage devices** like HDDs, SSDs, USB drives.
- Allows **reading/writing in blocks** (sectors).

II) Character Devices (c)

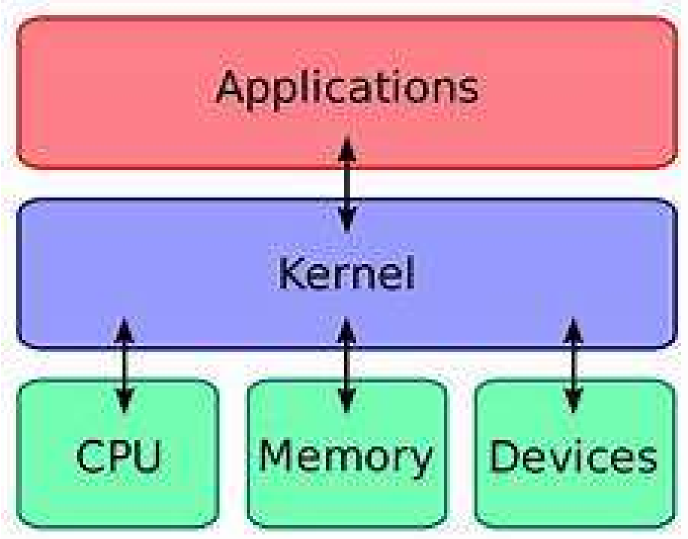
- Used for **serial devices** like keyboards, mice, serial ports.
- Allows **byte-by-byte communication**.

C1-3 : Linux Fundamentals and BSP Introduction

Kernel Architecture, Processes, and Boot Process

What is a Kernel?

- ❖ A set of code which directly interacts with hardware and allocate and manages resources such as CPU time, memory and I/O access .Kernel also contain system calls which provide specific functions.
- ❖ it’s a program that runs in Kernel Mode.
- ❖ CPUs run either in Kernel Mode or in User Mode.
- ❖ when in User Mode, some parts of RAM can’t be addressed, some instructions can’t be executed, and I/O ports can’t be accessed.
- ❖ when in Kernel Mode, no restriction is put on the program
- ❖ besides running in Kernel Mode, kernels have three other peculiarities such as:
    - large size (millions of machine language instructions)
    - machine dependency (some parts of the kernel must be coded in Assembly language)
    - loading into RAM at boot time in a rather primitive way



Types of kernel

Monolithic:

Overview

•A **monolithic kernel** is a single, large program that contains all the core operating system functions, such as memory management, process scheduling, file system management, and device drivers.

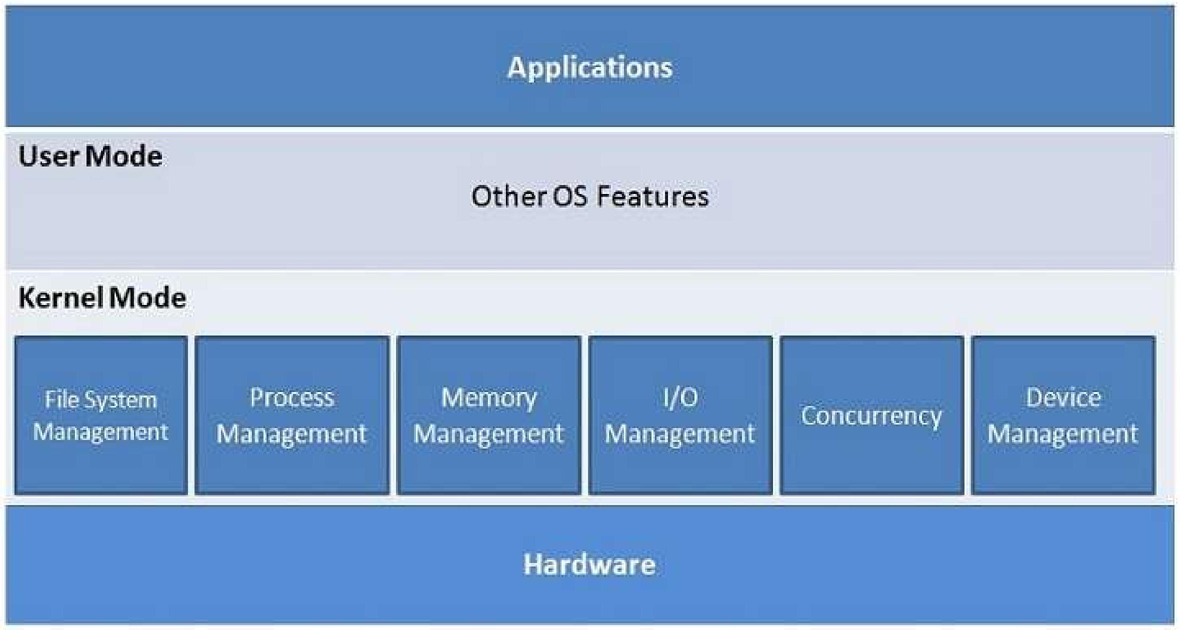
•All these components run in **kernel space**, which provides high performance but can make the kernel complex and harder to maintain.

Features

•**High Performance**: Since all components run in kernel space, there is minimal overhead.

•**Tight Integration**: All parts of the kernel are tightly coupled, which can improve efficiency.

•**Complexity**: The kernel can become large and difficult to maintain.



Monolithic Kernel Architecture

Examples

•**Linux**: Although Linux is often called a monolithic kernel, it supports modularity through loadable kernel modules (LKMs).

•**Unix**: Traditional Unix kernels are monolithic.

Types of kernel

Micro:

•A **microkernel** is a minimal kernel that provides only the most essential services, such as inter-process communication (IPC), memory management, and basic scheduling.

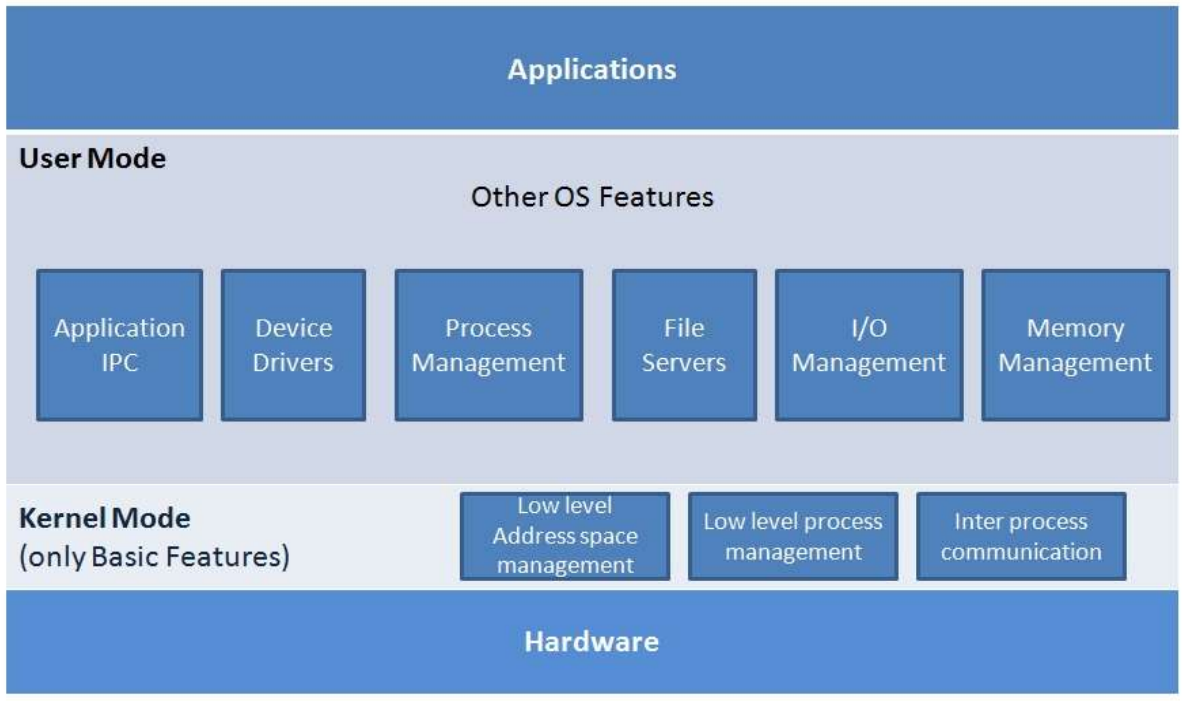
•Other services, such as device drivers and file systems, run in **user space** as separate processes.

Features

•**Modularity**: Easier to maintain and extend since most services run in user space.

•**Stability**: If a service crashes, it does not bring down the entire system.

•**Performance Overhead**: Communication between user-space services and the kernel can introduce latency.

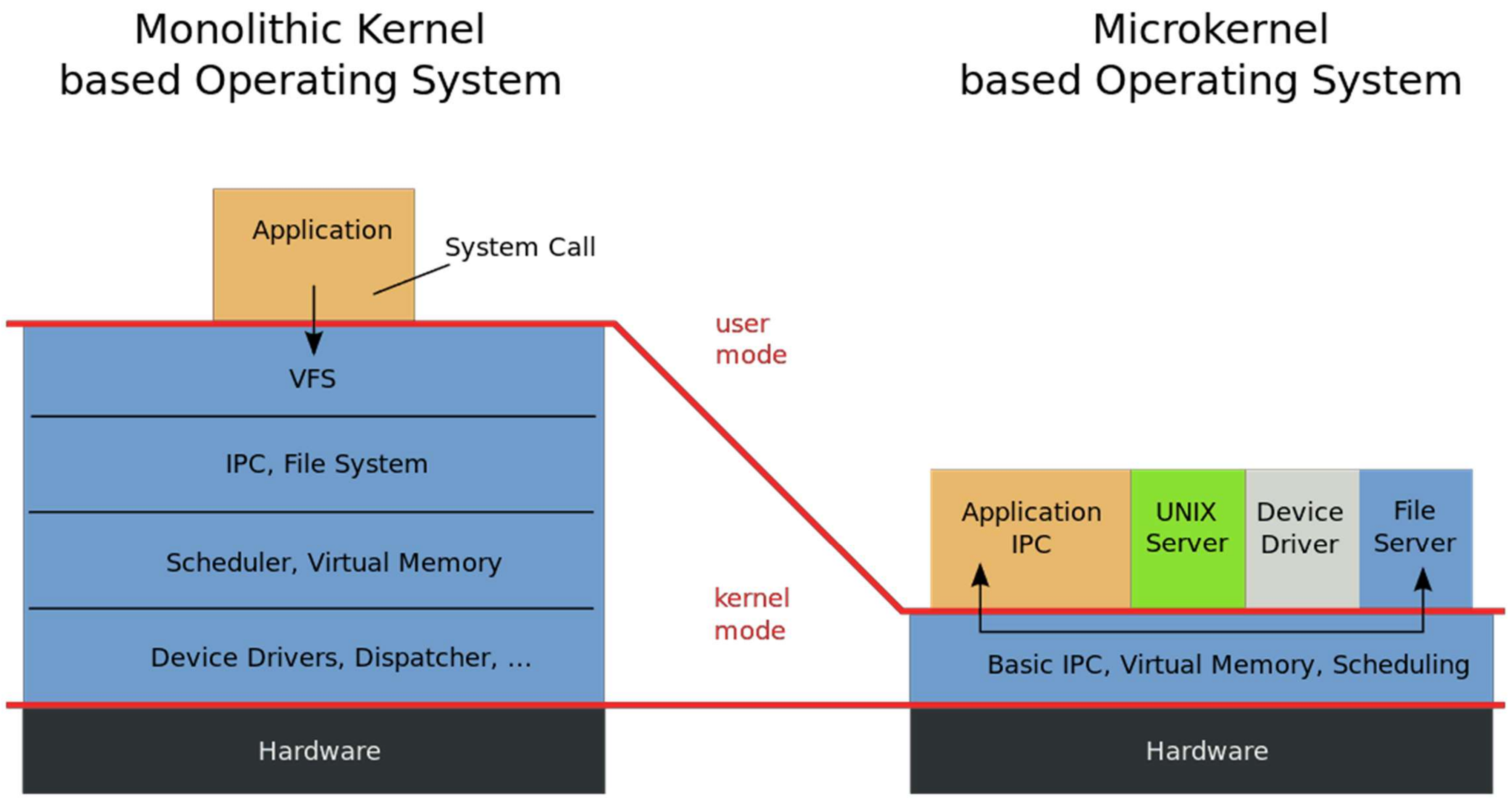


Micro Kernel Architecture

Examples

•**Mach**: The kernel used in early versions of macOS.

•**MINIX**: A Unix-like operating system designed for educational purposes.


Monolithic Kernel
based Operating System
Microkernel
based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user
mode
kernel
mode
Application
IPC
UNIX
Server
Device
Driver
File
Server
Basic IPC, Virtual Memory, Scheduling
Hardware

Types of kernel

Hybrid:

Overview

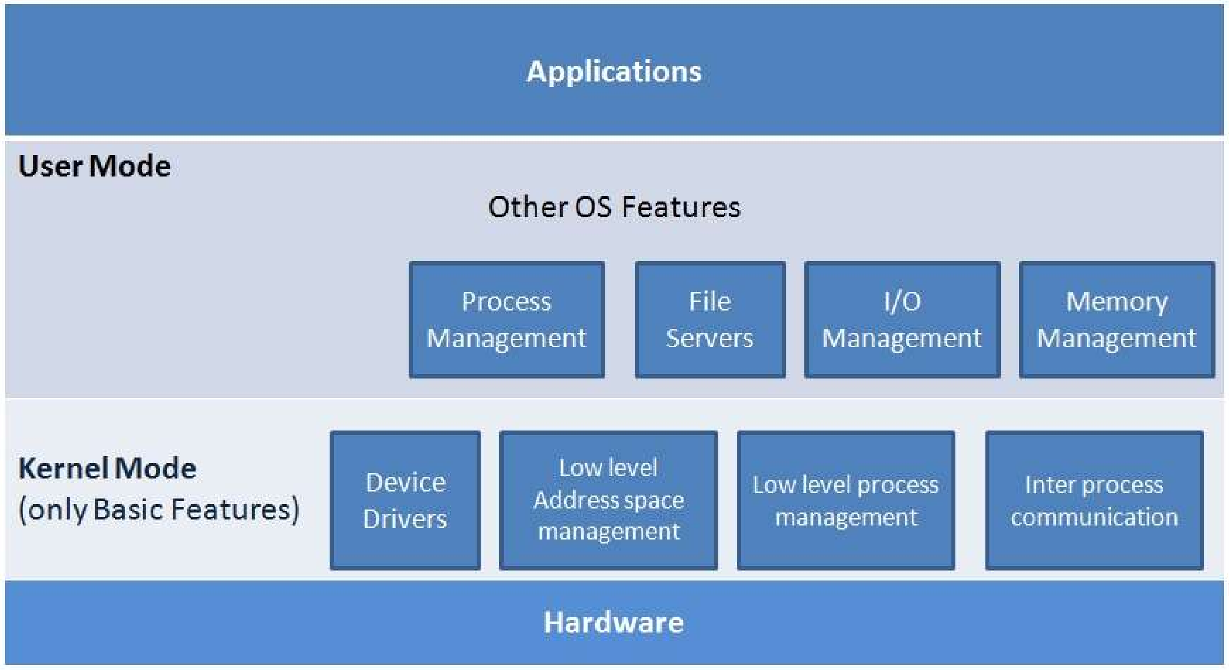
•A **hybrid kernel** combines aspects of both monolithic and microkernel designs.

•It keeps some services in kernel space for performance but moves others to user space for modularity and stability.

Features

•**Balance**: Offers a balance between performance and modularity.

•**Flexibility**: Can be optimized for specific use cases.



Example Hybrid Kernel Architecture. Implementations may differ

Examples

•**Windows NT**: The kernel used in modern Windows operating systems.

•**macOS (XNU)**: The kernel used in macOS and iOS, which combines Mach (microkernel) with BSD components (monolithic).

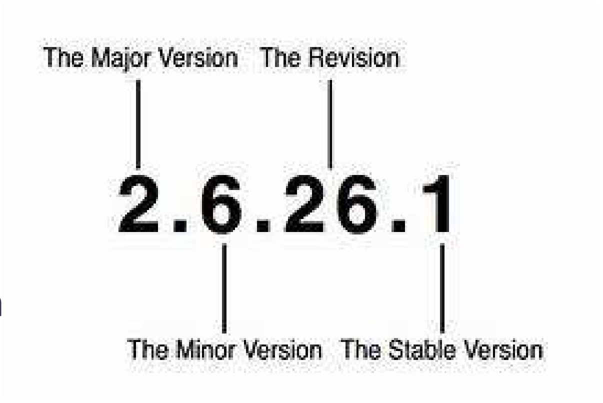
Linux Kernel Versions

- ❖ Linux distinguishes stable kernels from development kernels through a simple numbering scheme. Each version is characterized by three numbers, separated by periods. The first two numbers are used to identify the version; the third number identifies the release.

- ❖ The third field is number of patch. Patches are intended to fix some bug,
they almost never introduce new feature in stable kernel.

- ❖ Patches that do not bring new features (they should be less than 100 lines in length) increase the fourth number.

- ❖ If the fourth number is zero, it's not written: first patch changes supposed 2.2.14 to 2.2.14.1 and the next one to 2.2.14.2 and so on.



- As shown in Figure 1-1, if the second number is even, it denotes a stable kernel; otherwise, it denotes a development kernel. The 2.2 kernel was first released in January 1999, and it differs considerably from the 2.0 kernel, particularly with respect to memory management. Work on the 2.3 development version started in May 1999.

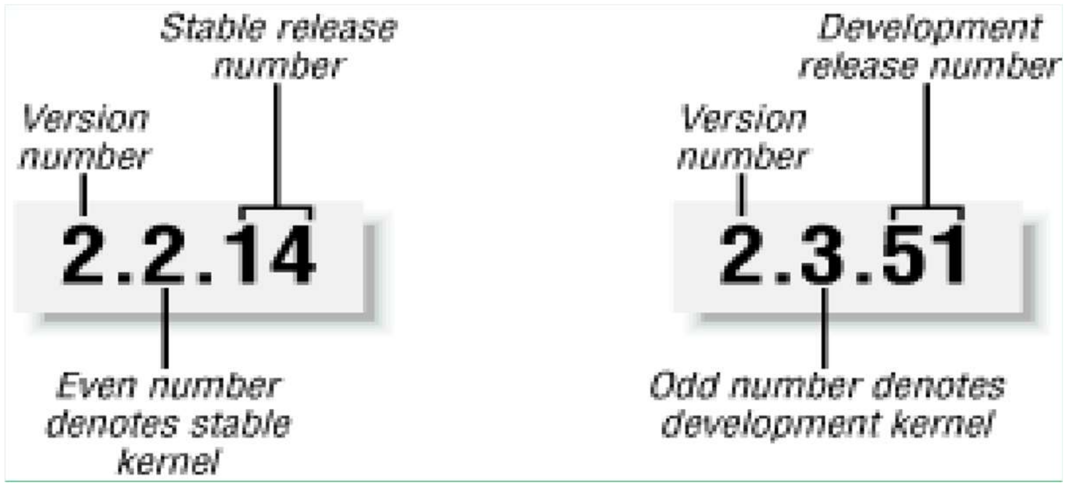

Stable release number
Version number
2.2.14
Even number denotes stable kernel
Development release number
Version number
2.3.51
Odd number denotes development kernel

Figure 1-1. Numbering Linux versions

Kernel Roles and Functionality

A. File System:

- ❖ It is responsible for storing information on disk and retrieving and updating this information.
- ❖ The File System is accessed through system calls such as : open, read, write, …

Example :

- FAT16, FAT32, NTFS

-ext2, ext3…

B. Device Driver:

- ❖ One of the purpose of an OS is to hide the system's hardware from user.
- ❖ Instead of putting code to manage the HW controller into every application, the code is kept in the Linux kernel.
- ❖ It abstracts the handling of devices.

  - All HW devices look like regular files.

C. Process Management:

❖ The Unix OS is a time-sharing system.
❖ Every process is scheduled to run for a period of time (time slice).
❖ Kernel creates, manages and deletes the processes.
❖ Every process (except **init**) in the system is create as the result of a **fork** system call.
❖ The fork system call splits a process into two processes (**Parent** and **Child**).
❖ Each process has a unique identifier (**Process ID**).

D. Memory Management:

❖ Physical memory is limited.
❖ Virtual memory is developed to overcome this limitation such as:

-Large Address space
-Protection
-Memory mapping
-Fair physical memory allocation
-Shared virtual memory

Why we need Kernel

- Each operating system uses a **kernel**. Without a kernel, you can’t have an operating system that actually works. Windows, Mac OS X, and Linux all have kernels, and they’re all different. It’s the kernel that also does the grunt work of the operating system. Besides the kernel, there are a lot of applications that are bundled with the kernel to make the entire package something useful — more on that a bit later.

- The **kernel’s** job is to talk to the hardware and software, and to manage the system’s resources as best as possible. It talks to the hardware via the drivers that are included in the kernel (or additionally installed later on in the form of a kernel module).

- It also aims to avoid deadlocks, which are problems that completely halt the system when one application needs a resource that another application is using. It’s a fairly complicated circus act to coordinate all of those things, but it needs to be done and that’s what the kernel is for.

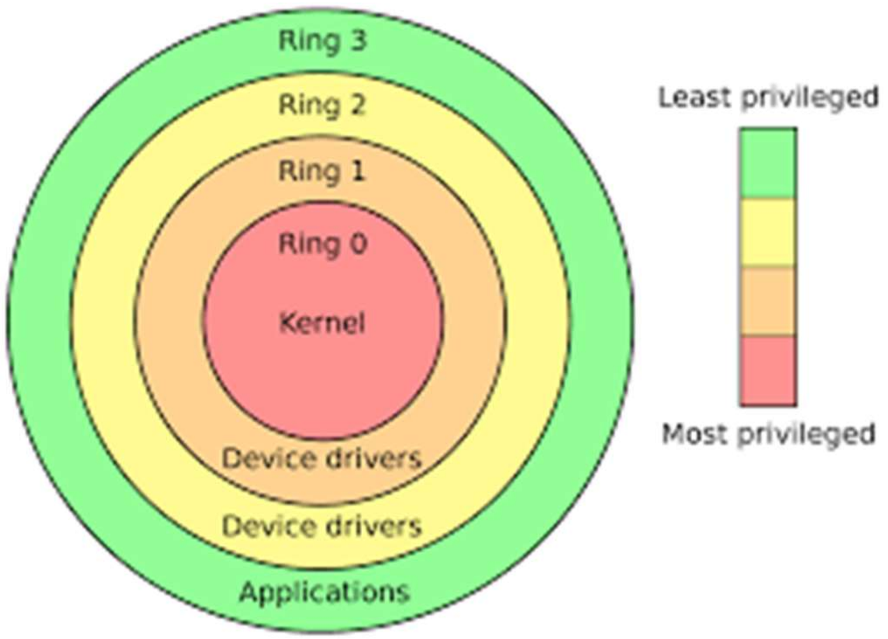
Linux Architecture

User Space: Area where user applications run.

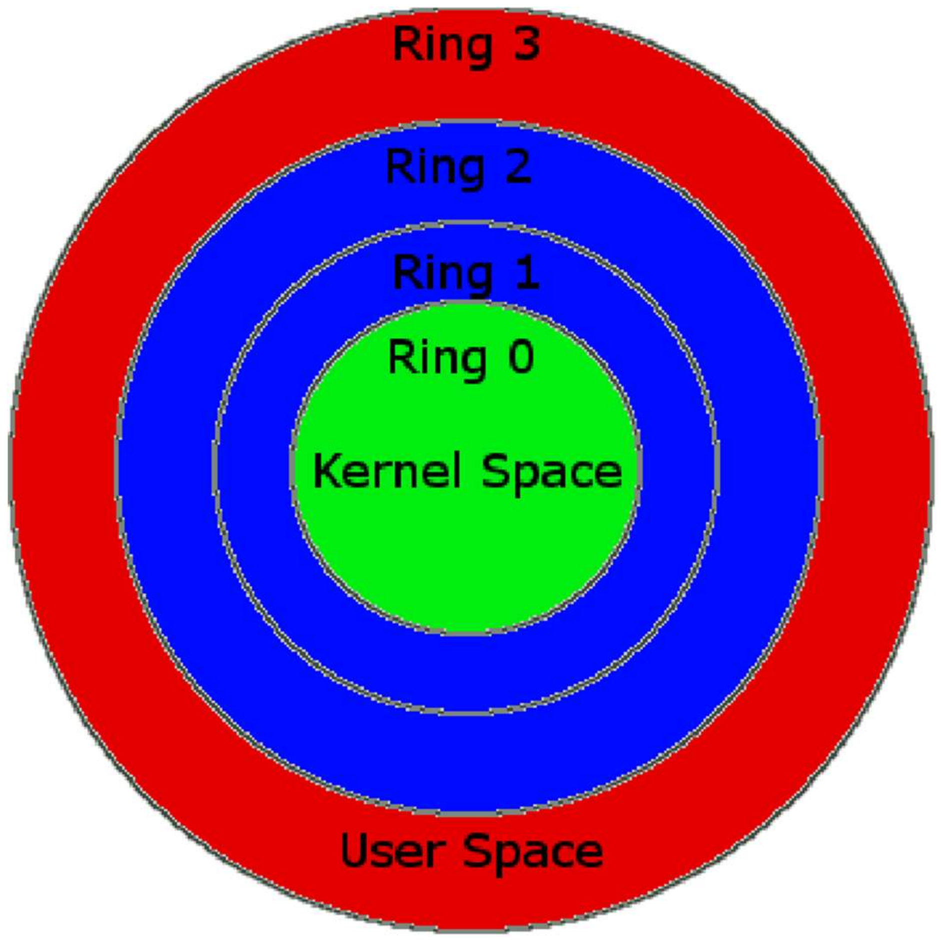
System Call Interface: Mechanism for user applications to request services from the kernel.

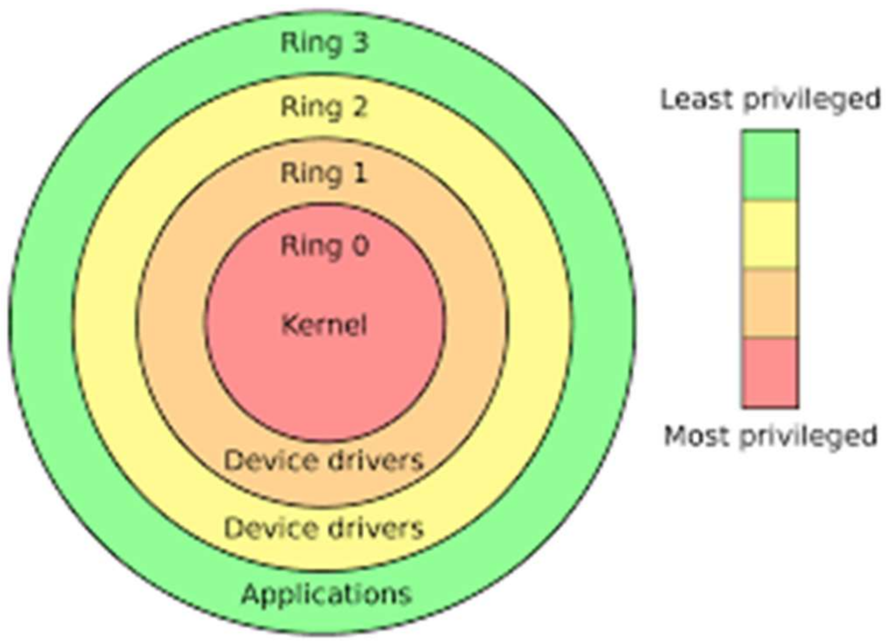
Kernel Space: Protected memory area where the kernel executes, managing system resources.

Hardware: Physical components of the computer system.

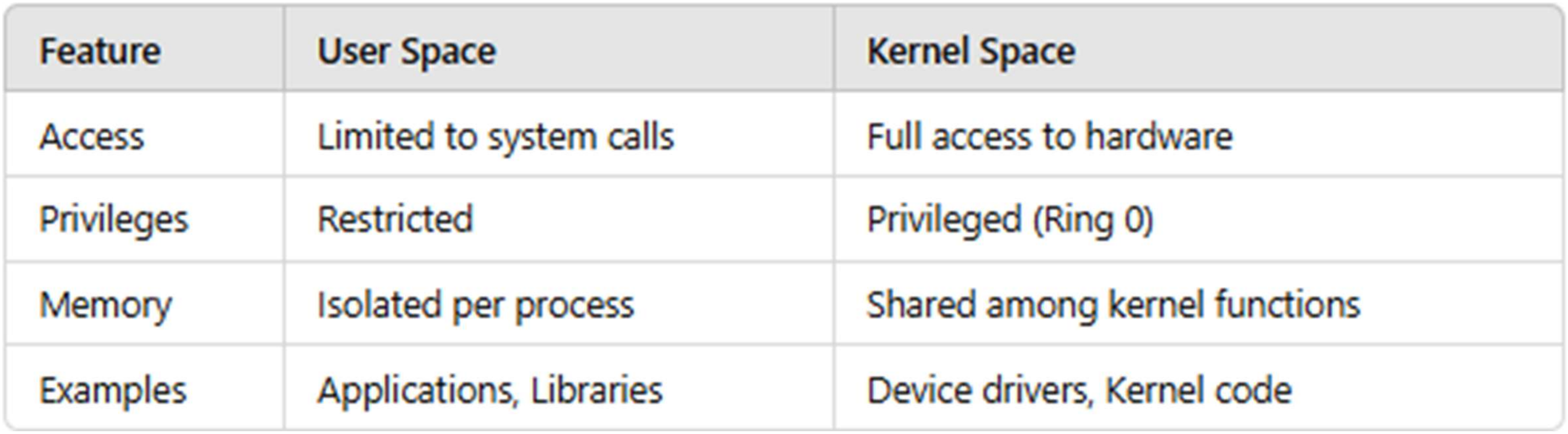


Linux Architecture





User Space vs. Kernel Space



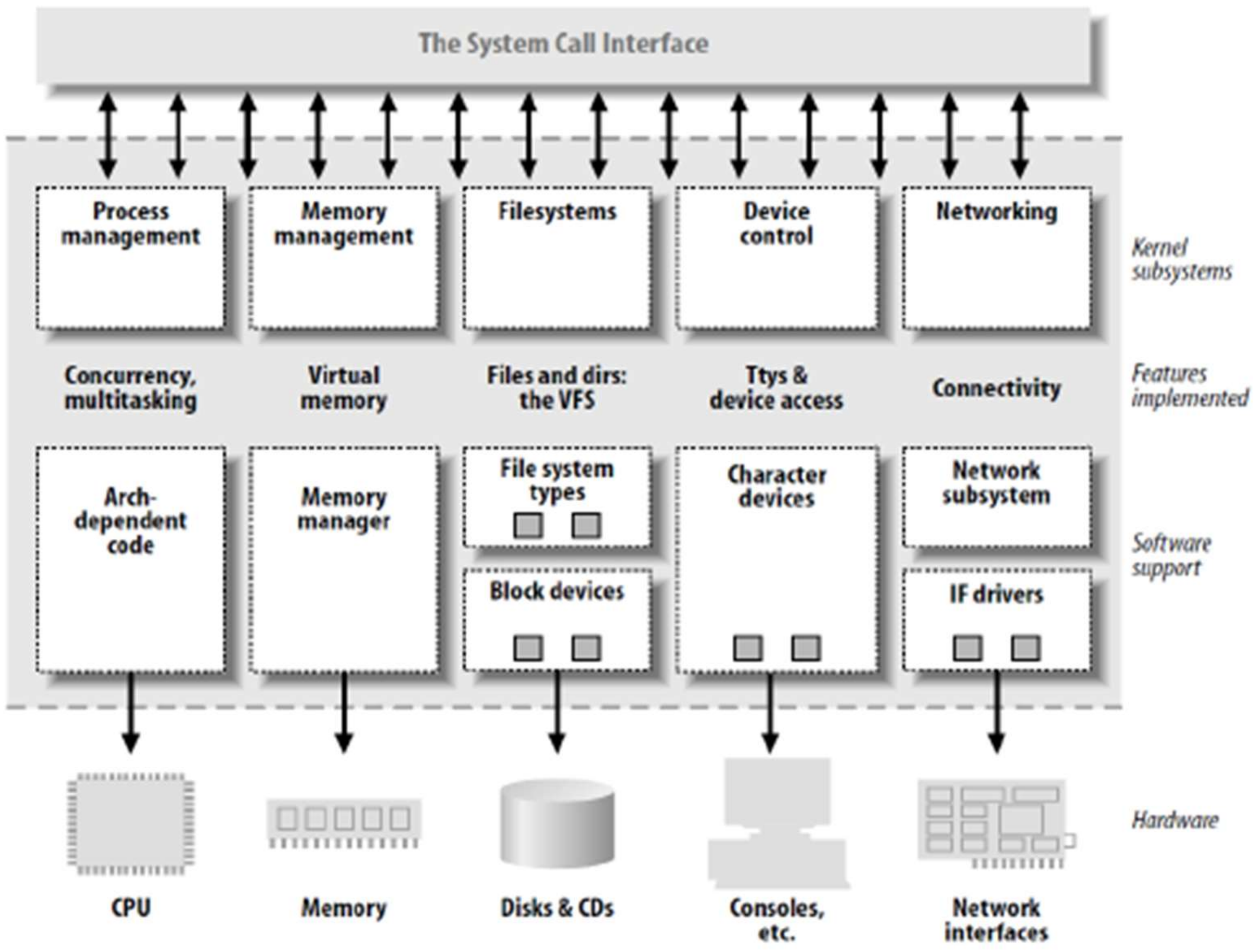
Feature	User Space	Kernel Space
Access	Limited to system calls	Full access to hardware
Privileges	Restricted	Privileged (Ring 0)
Memory	Isolated per process	Shared among kernel functions
Examples	Applications, Libraries	Device drivers, Kernel code

- **System Calls** act as a bridge between User Space & Kernel Space.

Kernel Mode vs. User Mode Execution

- **User Mode:** Runs normal applications with restricted access.
- **Kernel Mode:** Runs privileged operations like hardware control, memory management.
- **Mode Switching:** Transition between User Mode and Kernel Mode via **system calls**.

Linux kernel Architecture



Linux kernel Vertical Vs Horizontal

- **Linux Kernel divided into 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

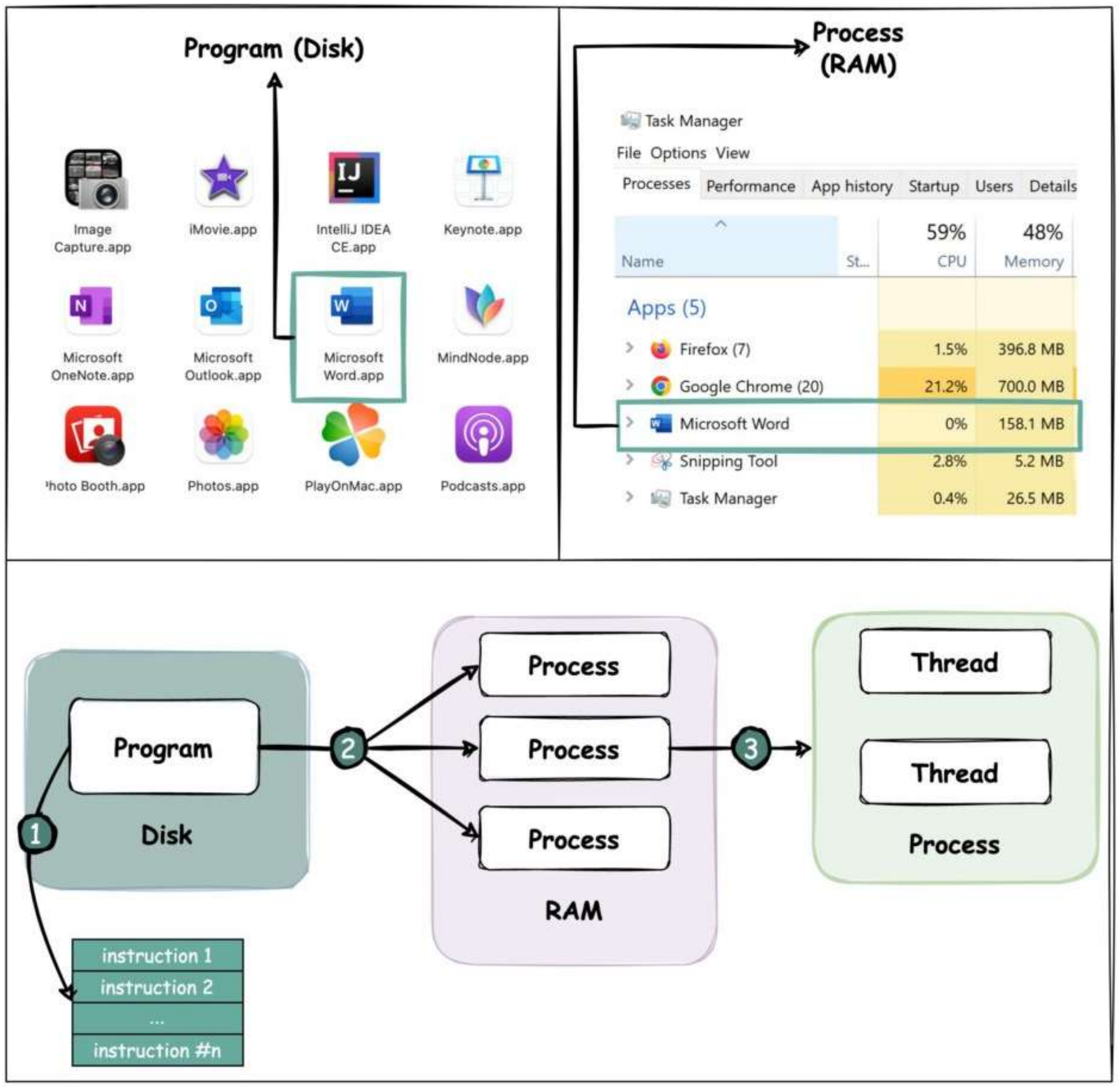
Parameters	Linux System Programming	Linux Kernel Programming	Linux Device Drivers Programming
Purpose	System Programmers write daemons, utilities and other tools for automating common or difficult tasks.	Kernel Developers focus on interfaces, data structures, algorithms and optimization for the core 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.

Introduction to Kernel Processes

Processes are running instances of programs.

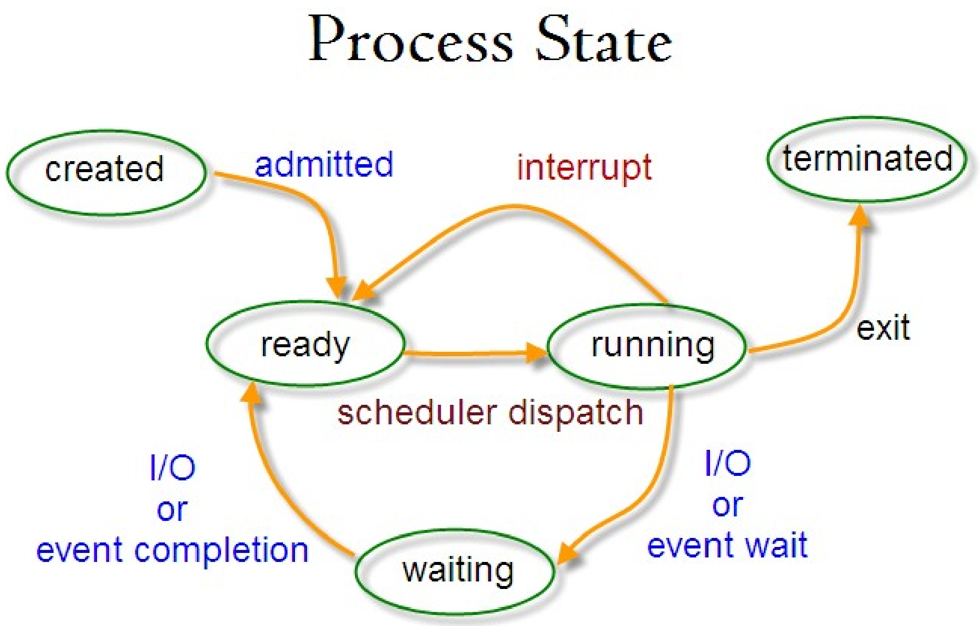
The **kernel manages all system processes**, including:

- Process creation & termination
- CPU scheduling
- Inter-process communication (IPC)



Name	St...	59% CPU	48% Memory
Apps (5)			
Firefox (7)		1.5%	396.8 MB
Google Chrome (20)		21.2%	700.0 MB
Microsoft Word		0%	158.1 MB
Snipping Tool		2.8%	5.2 MB
Task Manager		0.4%	26.5 MB

Process Lifecycle

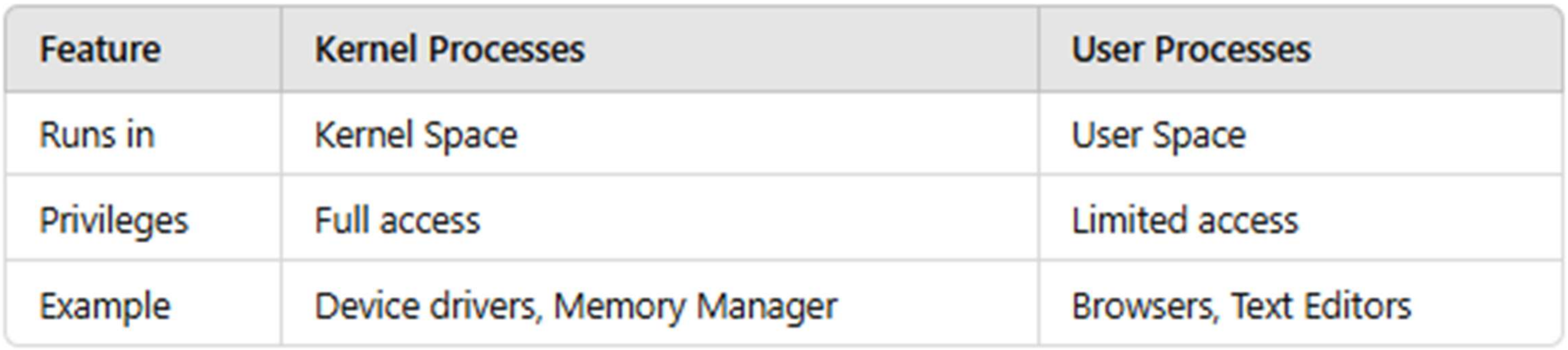


Steps in Process Execution:

1. **Created** – via fork() system call.
2. **Ready** – Waiting in queue for CPU.
3. **Running** – Actively executing on CPU.
4. **Waiting** – Suspended, waiting for an ever
5. **Terminated** – Exits after completion.

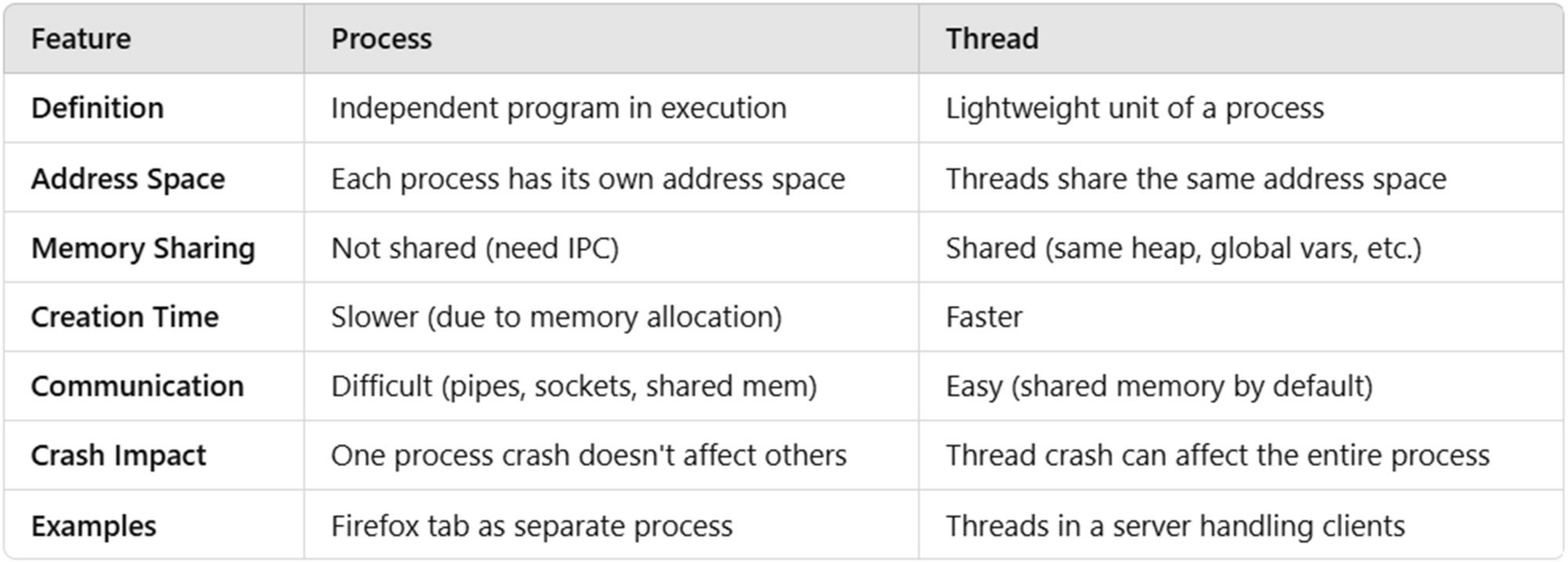
Practical => Create a process and show

Kernel Process vs. User Processes



Feature	Kernel Processes	User Processes
Runs in	Kernel Space	User Space
Privileges	Full access	Limited access
Example	Device drivers, Memory Manager	Browsers, Text Editors

Process vs Threads



Feature	Process	Thread
Definition	Independent program in execution	Lightweight unit of a process
Address Space	Each process has its own address space	Threads share the same address space
Memory Sharing	Not shared (need IPC)	Shared (same heap, global vars, etc.)
Creation Time	Slower (due to memory allocation)	Faster
Communication	Difficult (pipes, sockets, shared mem)	Easy (shared memory by default)
Crash Impact	One process crash doesn't affect others	Thread crash can affect the entire process
Examples	Firefox tab as separate process	Threads in a server handling clients

Practical => Create a Thread and show

Kernel Threads

Kernel threads are lightweight processes that execute in **kernel space**.

Used for:

- Background services (e.g., memory management).
- Hardware event handling.
- Scheduling tasks efficiently.

Practical => Create a kernel Thread and show

Kernel Modules

Kernel modules are dynamically loadable pieces of code extending kernel functionality.

Example: **Device Drivers** for peripherals (USB, Wi-Fi, GPU).
Benefits:

- Modular and flexible
- Reduces kernel size
- Can be loaded/unloaded dynamically

Types of Kernel Modules

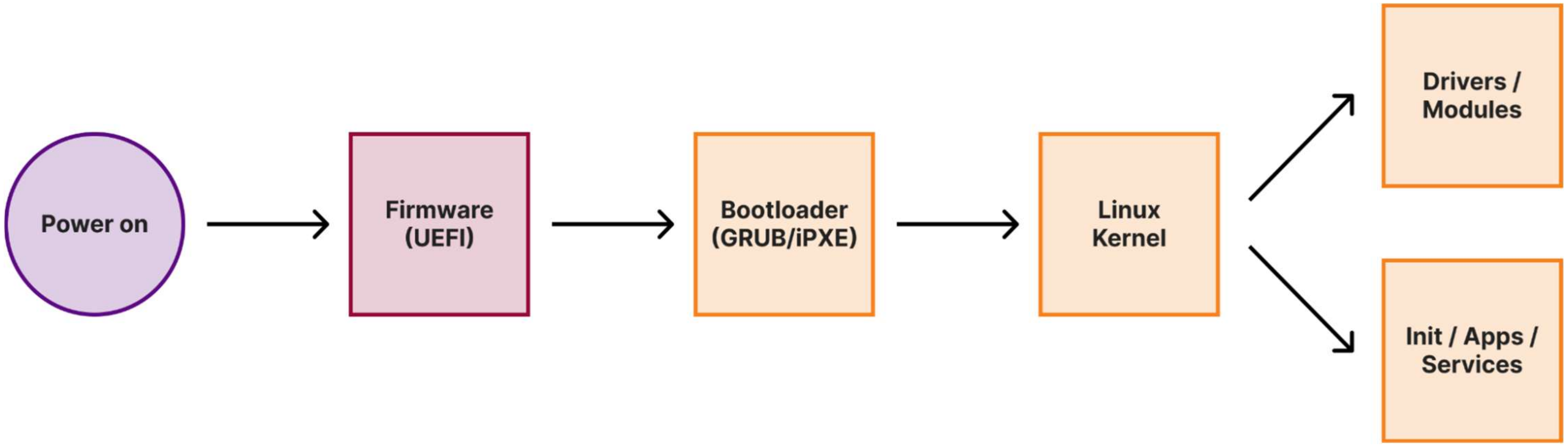
1.**Built-in Modules:** Compiled directly into the kernel.

2.**Loadable Kernel Modules (LKMs):** Can be loaded/unloaded at runtime.

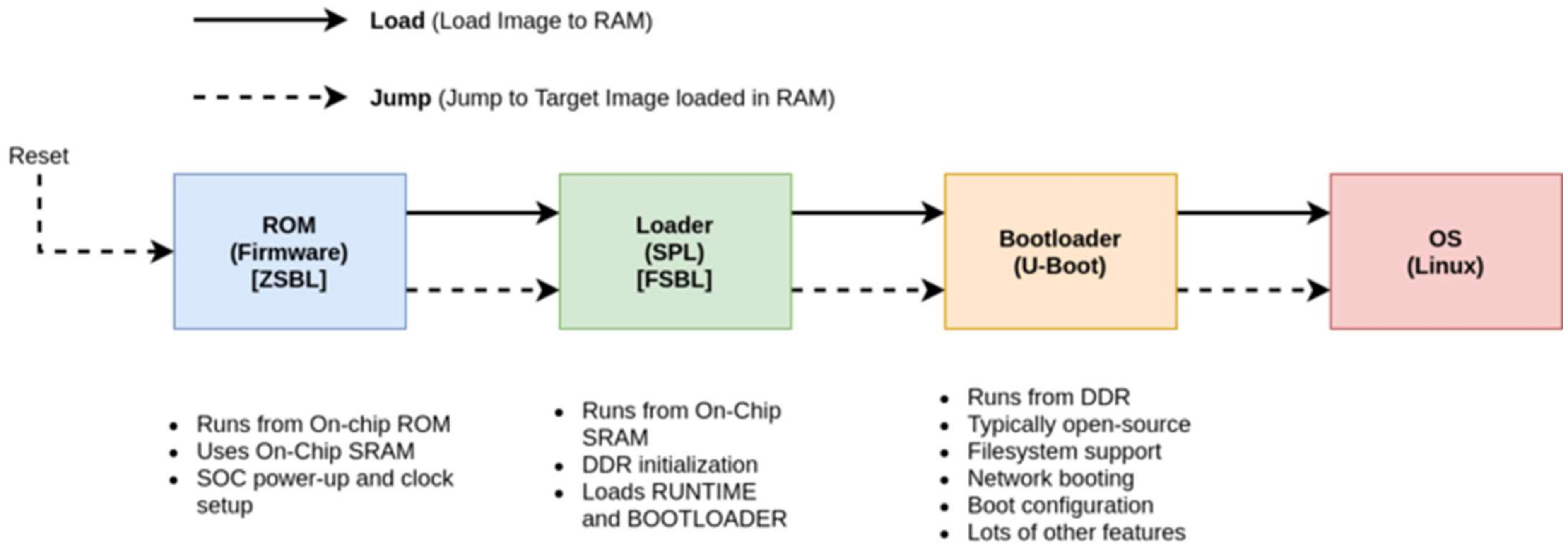
Practical => Create a Kernel Module and show

Practical => kernel source code walkthrough

Linux Boot Process



Embedded Linux Boot Process



- Runs from On-chip ROM
- Uses On-Chip SRAM
- SOC power-up and clock setup

- Runs from On-Chip SRAM
- DDR initialization
- Loads RUNTIME and BOOTLOADER

- Runs from DDR
- Typically open-source
- Filesystem support
- Network booting
- Boot configuration
- Lots of other features

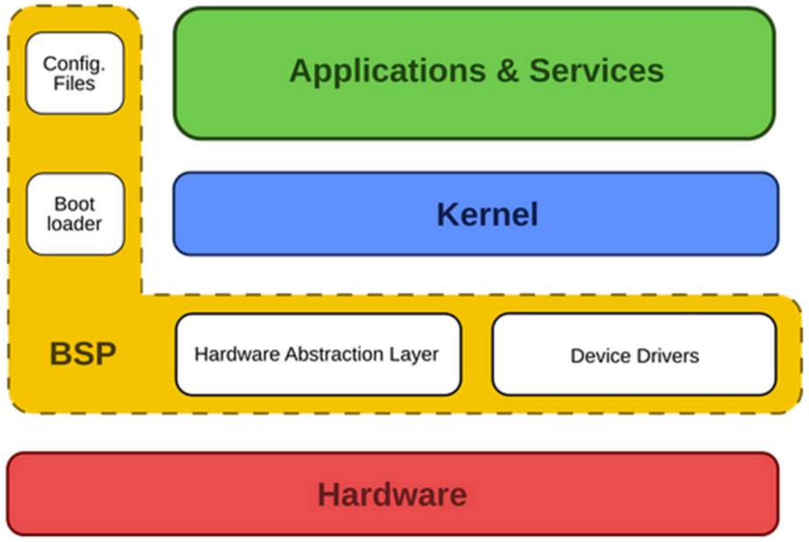
Practical => Telechip kernel source code and Boot process walkthrough

Introduction to BSP

- In embedded systems, a board support package (BSP) is the software layer containing hardware-specific boot firmware and device drivers and other routines that allow a given embedded operating system, (RTOS), to function in a given hardware environment (a motherboard),
- Integrated with the embedded operating system.
- BSP contains essential software a Computer hardware device needs to work with the computer's Operating System (OS). Whenever the hardware device is switched off, the bootloader or boot manager on the BSP will ensure the OS and device drivers are placed into memory.

Introduction to BSP

- A BSP is used to start up and run the embedded target processor.
- The BSP support a minimum number of peripherals on an eval board; so that the user can at least demonstrate that SoC peripherals work and that the SoC can be easily programmed to carry out the main features.
- The BSP has a boot-up program called the “bootloader” or a boot manager, and there are different BSPs for different operating systems.
- BSP contains drivers that enable the peripherals to communicate with the operating system.
- The BSP provides a file structure and will initialize several systems, including the processor, any communication buses, clocks, and memory, as well as start up the boot loader.



BSP Component

The BSP has two components;

1. The microprocessor support: Linux has wide support for all the leading processors in the embedded market such as MIPS (Microprocessor without Interlocked Pipelined Stages), ARM (Advanced RISC Machine), and PowerPC.

2. The board-specific routines: A typical HAL for the board hardware will include:
    1. Bootloader support
    2. Memory map support
    3. System timers
    4. Interrupt controller support
    5. Real-Time Clock (RTC)
    6. Serial support (debug and console)
    7. Bus support (PCI/ISA)
    8. DMA support
    9. Power management

Boot loader Interface

- The boot loader is a software that starts executing immediately after the system is powered on.
- Boot-loading issues are specific to the CPU and the boards.
- Many CPUs, such as the ARM, x86, and MIPS, start execution at specific vectors at reset.
- Some others, such as M68K, fetch the starting location from a boot ROM.
- Eliminating the boot loader and flashing the kernel that bootstraps itself is an approach provided by many RTOSs, which provides boot initialization routines to do POST, set up chip selects, initialize RAM and memory caches, and transfer the image from ROM to RAM.
- Most of the reset initialization is board-specific, and manufacturers of boards give an onboard PROM that does the above.

Boot loader Interface

The mandatory boot loader functionalities are:

1. Initializing the hardware: processor, memory controller, and the hardware devices necessary for loading the kernel such as flash.
2. Loading the kernel: The necessary software to download the kernel and copy it to the appropriate memory location

Boot loader Interface

1. Booting

- Most boot loaders start from the flash.
- They do the processor initialization such as configuring the cache, setting up some basic registers, and verifying the onboard RAM.
- Run the POST routines to do validation of the hardware required for the boot procedure such as validating memory, flash, buses, and so on.

2. Relocation

- The boot loaders relocate themselves to the RAM.
- This is because RAM is faster than flash.
- The relocation step include decompression, as the boot loaders are kept in a compressed format to save storage space.

3. Device initialization

- The boot loader initializes the basic devices necessary for user interaction. (console that a UI is shown for the user).
- It also initializes the devices necessary for picking up the kernel (and maybe the root file system).
- This may include the flash, network card, USB, and so on.

4. UI

- UI is shown for the user to select the kernel image to download onto the target
- There can be a deadline set for the user to enter choice.
- In case of a timeout a default image can be downloaded

5. Image download

- The kernel image is downloaded.
- In case the user has given the choice to download a root file system using the initrd mechanism, the initrd image too gets downloaded to memory

6. Preparing kernel boot

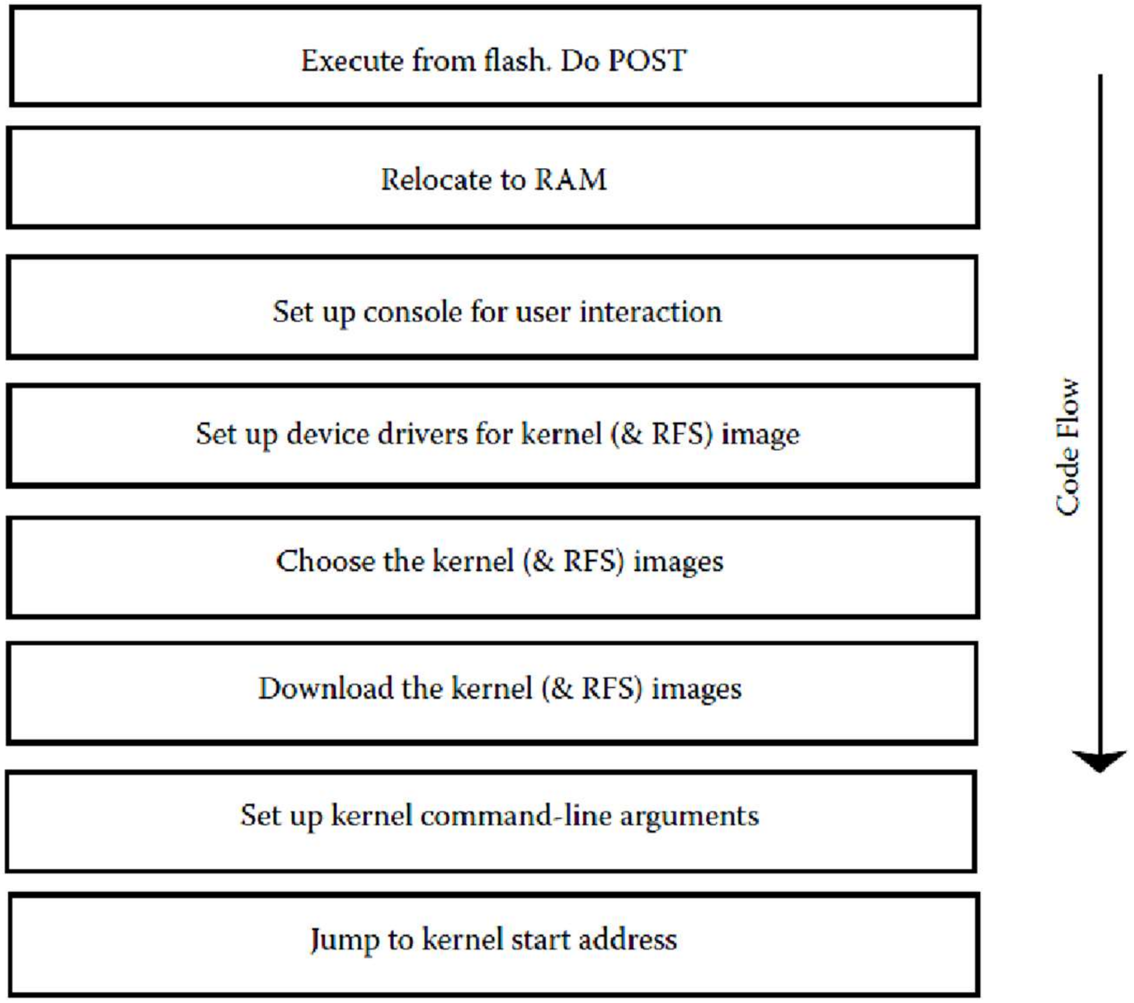
- In case arguments need to be passed to the kernel, the command-line arguments are filled and placed at known locations to the Linux kernel.

7. Booting kernel

- The transfer is given to the kernel.
- Once the Linux kernel starts running, the boot loader is no longer necessary.

Boot loader Interface

Bootloader start-up sequence


Execute from flash. Do POST
Relocate to RAM
Set up console for user interaction
Set up device drivers for kernel (& RFS) image
Choose the kernel (& RFS) images
Download the kernel (& RFS) images
Set up kernel command-line arguments
Jump to kernel start address
Code Flow