Introduction to Computational Science and Basics of Computer Programming (CDSC 601)

Addisu G. Semie, PhD
Asst. Prof.,Computational Data Science Program,
Addis Ababa University
Email: addisu.semie@aau.edu.et

Basics of Linux System

- The Linux Booting process
- Linux System Environment
- Basics of Linux Commands
- I/O Redirection
- Command pipelining
- Linux File System
- Virtual File System (VFS)
- Linux File System Layout
- Linux Special Files
- Naming Files
- The Current Directory
- Absolute Vs. Relative Paths
- Standard Files
- Meta Characters
- Some Special File Names
- Shell Variable and Scripting
- Linux File Category
- File Management Utilities
- File Permissions
- Mounting a new File system
- Maintaining File system
- Finding Files
- Command History
- Linux Shells
- Functions of a Shell
- Shell Variables
- Shell Scripting

The Boot Process

The process begins when the power supply is switched on

The power supply performs a self-test:

- When all voltages and current levels are acceptable (+5v, +3.0 through +6.0 is generally considered acceptable), the supply indicates that the power is stable and sends the "Power Good" signal to the motherboard.
- The "Power Good" signal is received by the microprocessor timer chip, which controls the reset line to the microprocessor.
- The time between turning on the switch to the generation of the "Power Good" signal is usually between 0.1 and 0.5 seconds.

The Boot Process

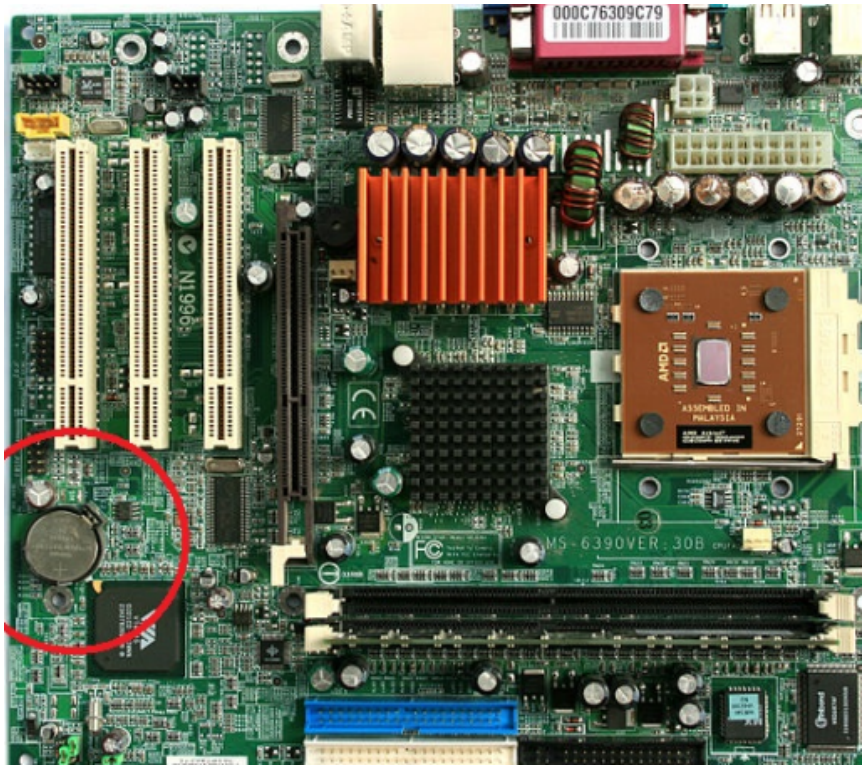
The microprocessor timer chip receives the "Power Good" signal:

- In the absence of the "Power Good" signal, the timer chip continuously resets the microprocessor, which prevents the system from running under bad or unstable power conditions.
- After the reset signal turns off, the CPU begins to operate. Code in RAM cannot be executed since the RAM is empty. The CPU manufacturers pre-program the processor to always begin executing code at address "FFFF:0000" (usually the ROM BIOS) of the ROM.

The Boot Process

The CPU starts executing the ROM BIOS code:

- The CPU loads and executes the ROM BIOS code starting at ROM memory address "FFFF:0000" which is only 16 bytes from the top of ROM memory. As such, it contains only a JMP (jump) instruction that points to the actual address of the ROM BIOS code.



The Boot Process

- **The BIOS locates and reads the configuration information stored in CMOS:**

  CMOS (Complementary Metal-Oxide Semiconductor) is a small area of memory (64 bytes) which is maintained by the current of a small battery attached to the motherboard. Most importantly, for the ROM BIOS startup routines (boot sequence), CMOS determines the order in which drives should be examined for an operating system (floppy disk first, CD-Rom first, or fixed disk first). Furthermore, it holds some essential information such as hard drive size, memory address location, and Date & Time.

The Boot Process

Loading the OS (Operating System):

- When a computer boots, the BIOS transfers control to the first boot device, which can be a hard disk, a floppy disk, a CD-ROM, or any other BIOS-recognized device. We'll concentrate on hard disks, for the sake of simplicity.
- The first sector on a hard is called the Master Boot Record (MBR). This sector is only 512 bytes long and contains a small piece of code (446 bytes) called the primary boot loader and the partition table (64 bytes) describing the primary and extended partitions.
- By default, MBR code looks for the partition marked as active and once such a partition is found, it loads its boot sector into memory and passes control to it.
- Boot loaders replaces the default MBR with their own code. And start booting

The Boot Process

- The boot loader loads the kernel and the kernel configures the system and completes it own loading
- The first thing the kernel does is to execute **init** program. Init is the root/parent of all processes executing on Linux and identified by process id "1".
- init starts to execute scripts in /etc/rc[].d/ directory
- User level applications software will be loaded at last

The Boot Process

Run levels

- The term ***runlevel*** refers to a mode of operation in one of the computer operating systems that implement Unix System V-style initialization.
- A run level describe the system’s state
- Conventionally, seven runlevels exist, numbered from zero to six;
- In standard practice, when a computer enters runlevel zero, it halts, and when it enters runlevel six, it reboots.
- The intermediate runlevels (1-5) differ in terms of which drives are mounted, and which network services are started.
- Lower run levels are useful for maintenance or emergency repairs, since they usually don't offer any network services at all.

The Boot Process

Run levels

- After the Linux kernel has booted, the **init** program reads the /etc/init/rc-sysinit.conf file to determine the behavior for each runlevel.

ID	Name	**Description**
0	Halt	Shuts down the system
1	Single User Mode	Does not: configure network interfaces, start daemons, or allow non-root logins
2	Multi-User Mode	Does not: configure network interfaces or start
3	Multi-User Mode with Networking	Daemons, start the system normally
4	Unused/User defined	For special purposes
5	X11	As runlevel 3 + display manager
6	Reboot	Reboots the system

The Boot Process

- Unless the user specifies another value as a kernel boot parameter, the system will attempt to enter (start) the default runlevel.
- The init process start certain services and kills other in a certain run level
- Depending on the *runlevel* chosen, init runs the scripts in /etc/rc(*runlevel*).d:

Linux System Environment

- The Linux System environment consists of user commands, the shell , kernel, file system, device drivers and the hardware.

- User commands includes executable programs and scripts
- The shell interprets user commands.
  - It is responsible for finding the commands and starting their execution.
  - Several different shells are available. Bash is popular

- The kernel manages the hardware resources for the rest of the system.

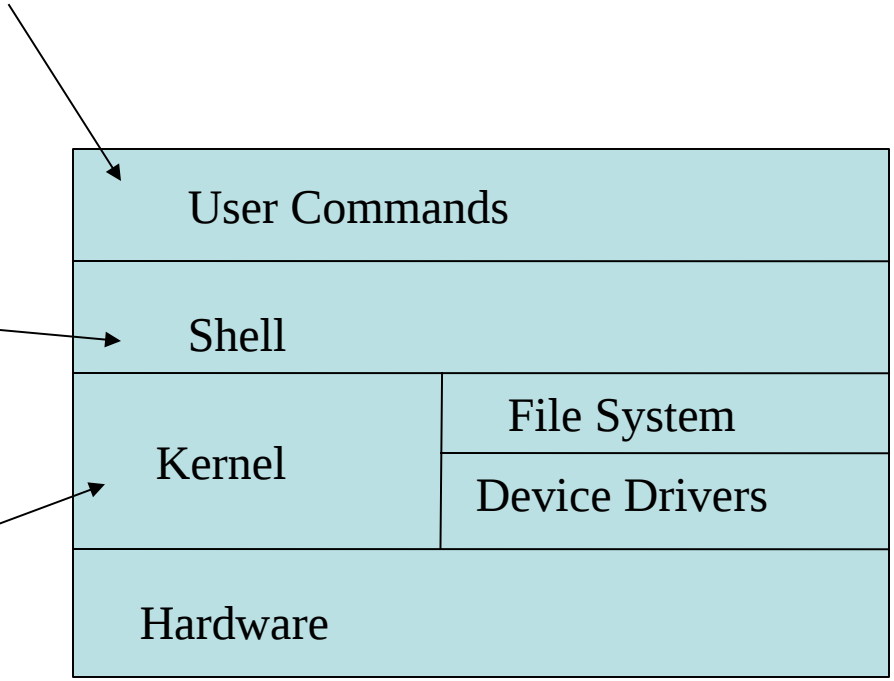
Linux System Environment

The shell interprets user commands.

It is responsible for finding the commands and starting their execution.

Several different shells are available. Bash is popular

The kernel manages the hardware resources for the rest of the system.



Basics of Linux Commands

- The common way that the user could interact with the kernel is via commands.
- The command will be submitted to the shell and the shell interpret the command and gives to the kernel.
- The kernel process the command and give back the output to the shell and the shell present the response to the user.

Basics of Linux Commands

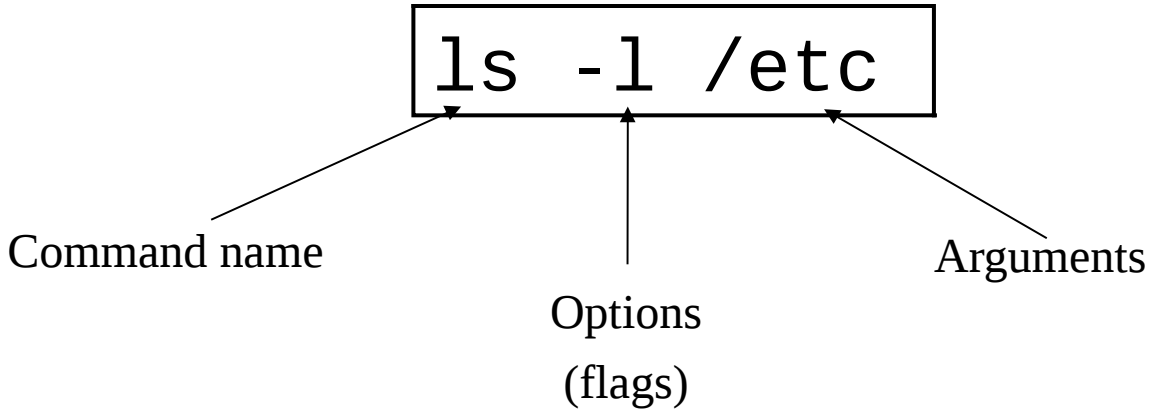
- Generally Linux command has well defined structure as

commandName [options] [arguments]

  - Options is also called flags allow you to control the command to a certain degree.
  - Usually options begin with a single dash and are a single letter (example **-l**).
  - But sometimes have double dashes followed by a keyword (example **--help**).

Basics of Linux Command

- To execute a command, type its name and arguments at the command line



- Command options allow you to control a command to a certain degree
- Conventions:
  - Usually being with a single dash and are a single letter (“-l”)
  - Sometimes have double dashes followed by a keyword (“--help”)
  - Sometimes follow no pattern at all

Basics of Linux Commands

- echo
- date
- pwd
- cd
- ls
- cat
- touch
- chmod
- diff
- head
- tail
- make
- clear
- alias

- ar
- tar
- grep
- awk
- sed
- who
- finger
- whoami
- ps
- find
- locate

- tty
- ifcnfig
- ifup
- ifdown
- foute
- netstat
- arp
- rarp
- ipchains

Basics of Linux Commands

- The most simple way of knowing the options and Linux command is using them and practicing them.
- Moreover, all commands have sufficient documentation.

- One can access the documentation manual via man or info command

  man commandName

  info commandName

  - This enable to access the manual of the stated command

Basics of Linux Commands

- Linux commands are usually located in the path **/bin, /usr/bin, /sbin**.
- However, it is also possible to have them in a user defined path and can be made accessible to users in various ways.
- One of the best option is to include the path of your command into the environment variable called $PATH.

Basics of Linux Commands

- Linux commands can be used to:
  - Navigate through the directory structure **(ls, pwd, cd)**,
  - Manage directory **(rm, rmdir, mkdir)**,
  - Manipulate file **(cp, mv, rm, touch, rename, cat, paste, chmod, chgrp, chown)**,
  - View file **(more, less, head, tail, grep, wc)**,
  - Get system information **(top, ps, kill, du, df)**

I/O Redirection

- The output of any Linux command that will be displayed on the standard output device can be redirected to a file using the operator >
- For example

ls -l > fileList
Will save the output of the command **ls -l** into the filename **fileList**. This command will create the output file if the file doesn't exist previously otherwise the command overwrite the previous content.

If we don't need the command to overwrite our file, we can use the redirection operator >> so that it will append the output at the end of the previous file

ls -l >> fileList
The above command will append the output into the file **fileList**. If the file doesn't exist, this operator create the file and put the output into the file.

I/O Redirection

- It is also possible to have input of a command from a file.
- That is possible to do using the redirection operator <.
- This operator force the command to manipulate the data in the input file and generate the corresponding output.
- For example

grep tts < myfile

This will Filter all the lines from the file **myfile** which has the text **tts**

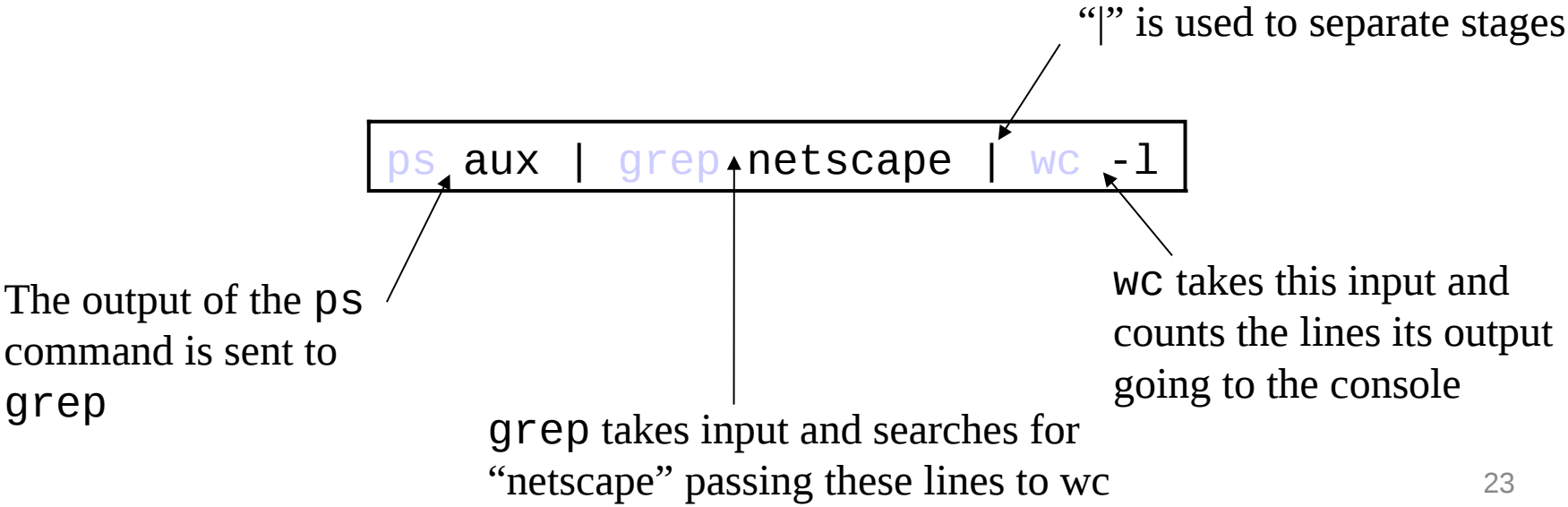
Command pipelining

- Moreover, the output of one command can be used as an input to another command.
- This is referred to as connecting commands with pipes
- We can do this in a single line shell command using piping (|).
- For example

command1 | command2 | command3

This is to mean the output of **command1** is the input to **command2** and output of **command2** is an input to **command3**

- ls -l | grep 5 | grep 6



Linux File System

- In a Linux system, everything is a file.
- A Linux system makes no difference between a file and a directory, since a directory is just a file containing names of other files.
- Programs, services, texts, images, and so forth, are all files.
- Input and output devices, and all other devices are considered to be files in Linux OS.

Linux File System

- Linux support various types of file systems which enable compatibility with different file system devices.
- Most commonly, ext2fs
  - Filenames of 255 characters
  - File sizes up to 2GB
  - Theoretical limit 4TB
- Derived from extfs
- Highly reliable and high performer

Linux File System

- Some other supported file systems include
  - Ext2fs
  - Ext3
  - Ext4
  - Sysv-SCO/Xenix
  - Ufs-SunOS/BSD
  - Vfat-Win9x
  - msdos 0-MS-DOS/Win
  - Umsdos-Linux/DOS
  - ntfs -WinNT (r/o)
  - hpfs-OS/2 (r/o)
  - iso9660 (CD-ROM)
  - nfs – NFS

Virtual File System (VFS)

- VFS is designed to present a consistent view of data as stored on hardware.
- Almost all hardware devices are represented using a generic interface.
- VFS promote compatibility with other operating system standards permitting developers to implement file systems with different policies.
- Most users are familiar with the two primary types of files: text and binary.
- But the /proc/ directory contains another type of file called a virtual file.
- It is for this reason that /proc/ is often referred to as a virtual file system.

Virtual File System (VFS)

- These virtual files have unique qualities unlike either text or binary files.
- Most of them are listed as zero bytes in size and yet when one is viewed, it can contain a large amount of information.
- In addition, most of the time and date settings on virtual files reflect the current time and date, indicative of the fact they are constantly updated.
- Virtual files such as
  - /proc/interrupts,
  - /proc/meminfo,
  - /proc/mounts, and
  - /proc/partitions

  provide an up-to-the-moment glimpse of the system's hardware.
- Others, like
  - /proc/filesystems and
  - /proc/sys/

  directory provide system configuration information and interfaces.

Virtual File System (VFS)

- **virtual file system name** **Content description**

  /proc/cpuinfo CPU Information

  /proc/interrupts Interrupt usage

  /proc/version Kernel version

  /proc/modules Active modules

- By using the **cat**, **more**, or **less** commands on files within the /proc/ directory, users can immediately access an enormous amount of information about the system.
- For example, to display the type of CPU a computer has, type ***cat*** *proc/cpuinfo* to receive output similar to the following (next page)

Virtual File System (VFS)

Check out the size of these files using the command

ls -l filename

ls -l /proc/cpuinfo

ls -l /proc/interrupts

ls -l /proc/version

ls -l /proc/modules

Display the content of these files

cat filename

cat /proc/cpuinfo

cat /proc/interrupts

cat /proc/version

cat /proc/modules

```
cat /proc/cpuinfo
vendor_id       : IBM/S390
# processors    : 1
bogomips per cpu: 86.83
processor 0: version = FF, identification = 045226, machine = 9672
```

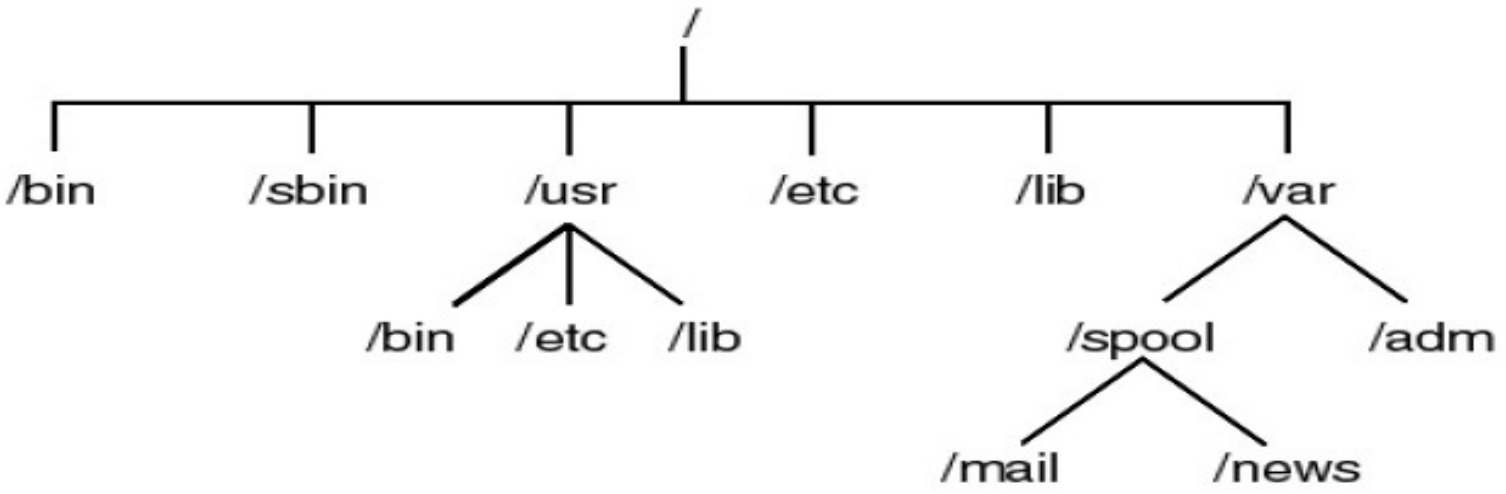
Virtual File System (VFS)

Explore /proc by your self for more

- Display the contents of the files interrupts, devices, cpuinfo, meminfo and uptime using cat.
- Can you see why we say /proc is a pseudo-filesystem which allows access to kernel data structures?

Linux File System Layout

- The Linux file system is laid out as a hierarchical tree structure which is anchored at a special top-level directory known as the **root** (designated /)
- Because of the tree structure, a directory can have many child directories, but only one parent directory.

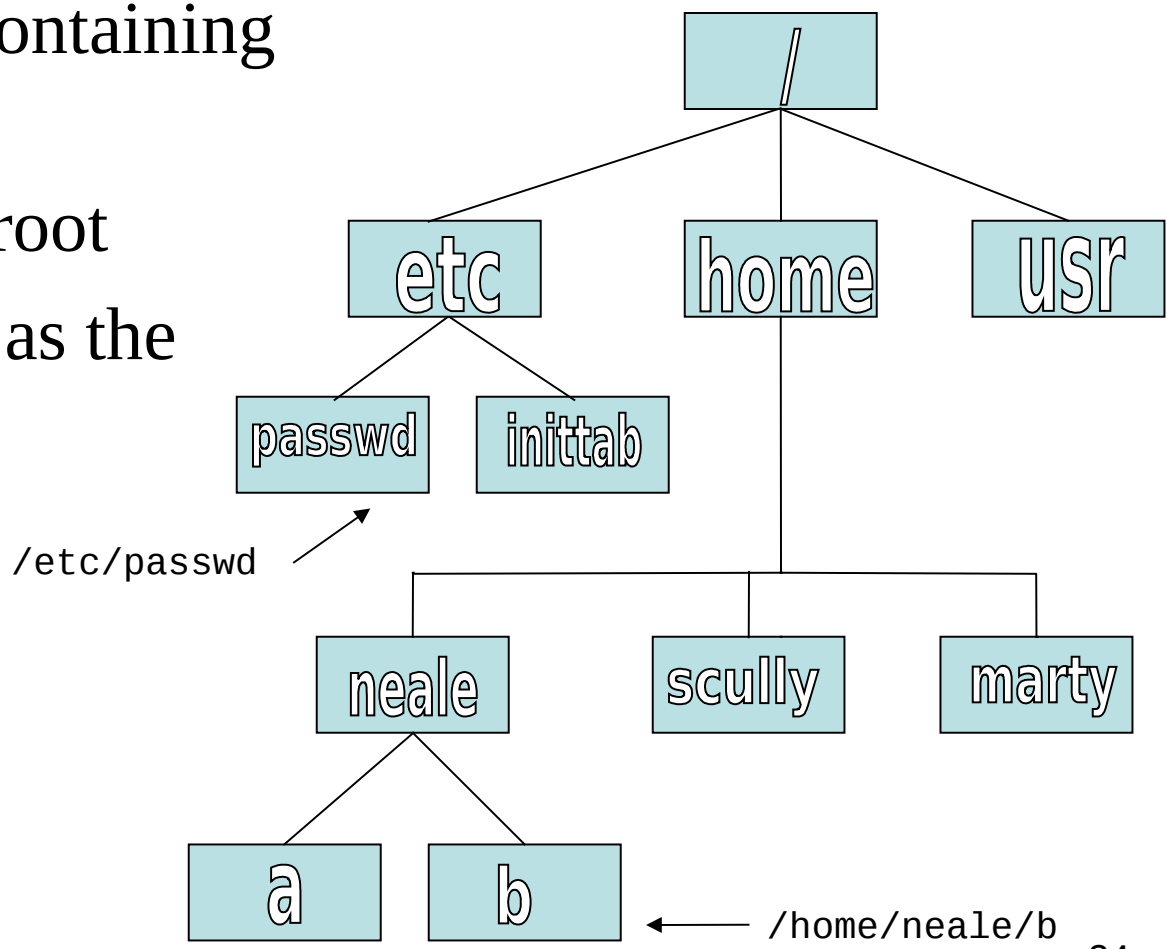


Linux Special Files

- /home - all users' home directories are stored here
- /bin and /usr/bin - system commands
- /sbin and /usr/sbin - commands used by sysadmins
- /etc - all sorts of configuration files
- /var - logs, spool directories etc.
- /dev - device files
- /proc - special system files

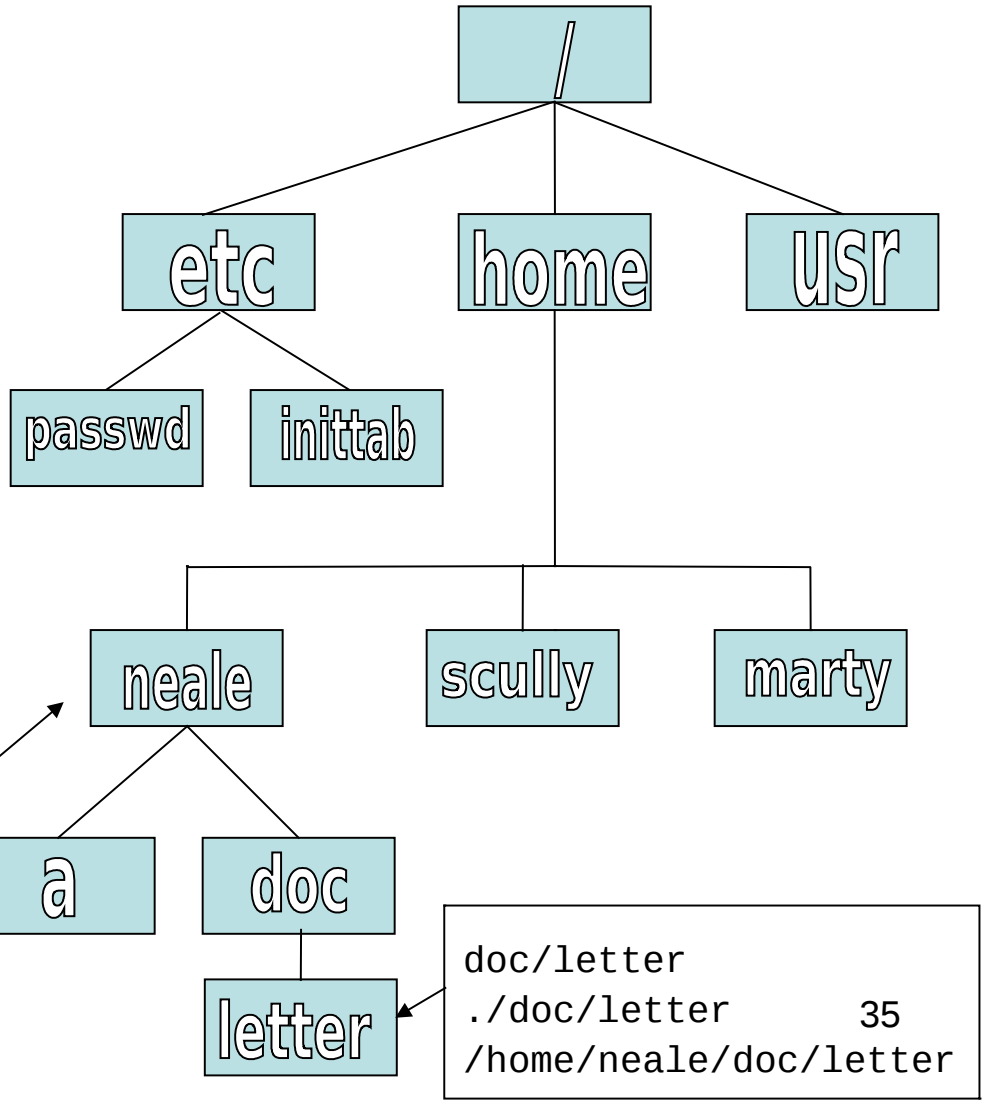
Naming Files

- Files are named by
  - naming each containing directory
  - starting at the root
  - This is known as the *pathname*



The Current Directory

- One directory is designated the *current working directory*
  - if you omit the leading / then path name is relative to the current working directory
  - Use pwd to find out where you are



Absolute Vs. Relative Paths

- To specify a location in the directory hierarchy, you must specify a path through the tree.
- **Absolute:** path to a location defined from the root /
- **Relative:** path to a location defined from the current working directory.
- **cd** command is used to change directory.
- **cd** command can be used with an absolute or a relative path

  e.g [**user@ubuntu ~] $ cd ..** return to parent dir

  [user@ubuntu ~] $ cd − return to previous dir

Exercise 1

Standard Files

- Linux concept of “standard files”
  - standard input (where a command gets its input) - default is the terminal
  - standard output (where a command writes its output) - default is the terminal
  - standard error (where a command writes error messages) - default is the terminal

Meta Characters

- * - matches any string or group of characters.
- ? - matches any single character
- [...] - matches any one of the enclosed characters

  e.g.

  - ls [abc]* - shows all files beginning with letters a,b,or c
  - ls /bin/[a-d]* - shows all files name beginning with letter a, b, c or d
  - ls /bin/[!a-o] or ls /bin/[^a-o] - do not list file name that begin with a,b,c,e...o

Some Special File Names

- Some file names are special:
  - / The root directory (not to be confused with the root user)
  - . The current directory
  - .. The parent (previous) directory
  - ~ My home directory
- Examples:
  - ./a same as a
  - ../jane/x go up one level then look in directory jane for x

Linux File Category

- Four types of file category exist in Linux
  1. Ordinary files
  2. Directories
  3. Links
  4. Devices

Linux File Category

- **Links**
  - A link is a pointer to another file.
  - It can either hard link or soft link

Linux File Category

- **Hard link**
  - Is a link directly points to the data structure of the file called i-node which defined the file.
  - The link and the original filename are the same
  - Syntax

ln linkeName fileName

- **Soft link**
  - Is a link to a file rather than the i-node.
  - If the link file get deleted or renamed, the soft link doesn't work
  - Syntax

ln –s linkeName fileName

Linux File Category

- Devices
  - In Unix-like operating systems, devices can be accessed via special files.
  - A device special file does not use any space on the file system.
  - It is only an access point to the device driver.
  - Two types of special files exist: character and block special files.
  - The former allows I/O operations in character mode while the later requires data to be written in block mode via the buffer cache functions.
  - When an I/O request is made on a special file, it is forwarded to a (pseudo) device driver.

Linux File Category

- The −l option to **ls** displays the file type, using the first character of each input line
  - **e.g. [user@ubuntu ~] $ ls −l**

- This table gives the characters determining the file type:

Symbol	Meaning
-	Regular File
d	Directory
l	Link
C	Character oriented Device file
b	Block oriented device file
p	Named Pipe
s	Socket

Linux File Category

- The **ls -l** command shows **the file type**, permissions on a file, file size, inode number, creation date and time, owners and amount of links to the file.

- The **ls −F** command suffixes file names with one of the characters " /=*|@ " to indicate the file type.

suffix	File Type
nothing	Regular file
/	directory
*	Executable
@	Link
=	Socket
|	Named pipe

File Management Utilities

- **ls -** list files and directories.
- By default **ls** will not list hidden files or directories that start with a leading full stop "."
- options:
  - **ls −l** - long listing, lists permissions, file size, modification date, ownership.
  - **ls −a** - shows all file & directories, including hidden files
  - **ls −d** - list directory entries
  - **ls −F** - append symbols indicate file types
  - **ls −S** - lists the files in descending order sorted by size.
  - **ls −R** - (recursive) to list everything in the current and sub directories

File Management Utilities

- cp <*fromfile*> <*tofile*>
  - Copy from the <fromfile> to the <tofile>
  - **cp –R** is the recursive copy (copy all underlying files and subdirectories)
- mv <*fromfile*> <*tofile*>
  - Move/rename the <fromfile> to the <tofile>
- rm <*file*>
  - Remove the file named <file>
  - **rm −f** -to force removal (useful when you don't want to be prompted).
  - **rm –i** - shell asks for confirmation before deleting
  - **rm –R directory**
- mkdir <*newdir*>
  - Make a new directory called <newdir>
- rmdir <*dir*>
  - Remove an (empty) directory

File Management Utilities

- **file** command determines the type and format of a file

  eg.

  - **[user@ubuntu~]$ file testf**

    testf: directory

  - **[user@ubuntu~]$ file intro-linux.pdf**

    Intro-linux.pdf: PDF document

  - **[user@ubuntu~]$ file myfile.txt**

    myfile.txt: ISO−8859 text

- **pwd** command Print working directory. Prints the absolute path to the working directory.

File Permissions

- Every file
  - Is owned by someone
  - Belongs to a group
  - Has certain access permissions for owner, group, and others
  - Default permissions determined by umask

- Every user:
  - Has a uid (login name), gid (login group) and membership of a "groups" list:
    - The *uid* is who you are (name and number)
    - The *gid* is your initial “login group” you normally belong to
    - The *groups list* is the file groups you can access via group permissions

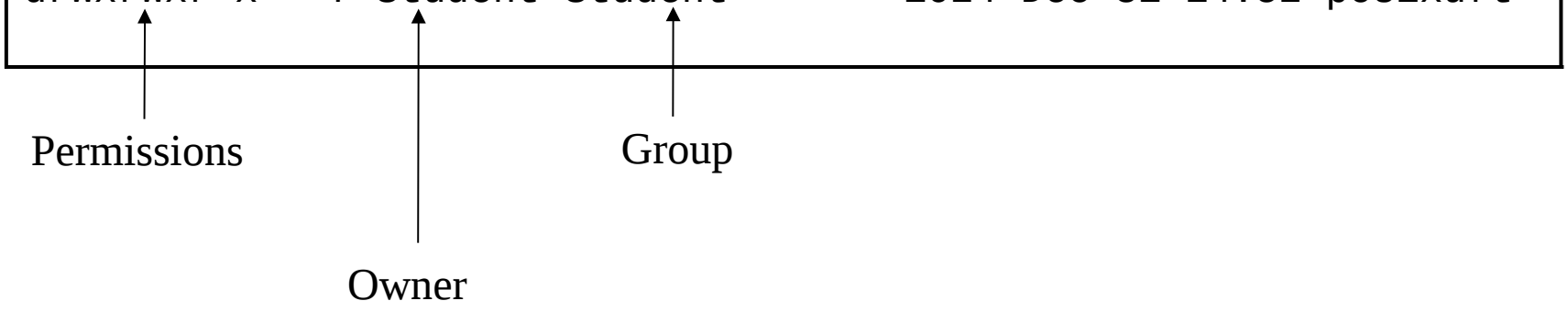
File Permissions

- Linux provides three kinds of permissions:
  - Read - users with read permission may read the file or list the directory
  - Write - users with write permission may write to the file or new files to the directory
  - Execute - users with execute permission may execute the file or lookup a specific file within a directory

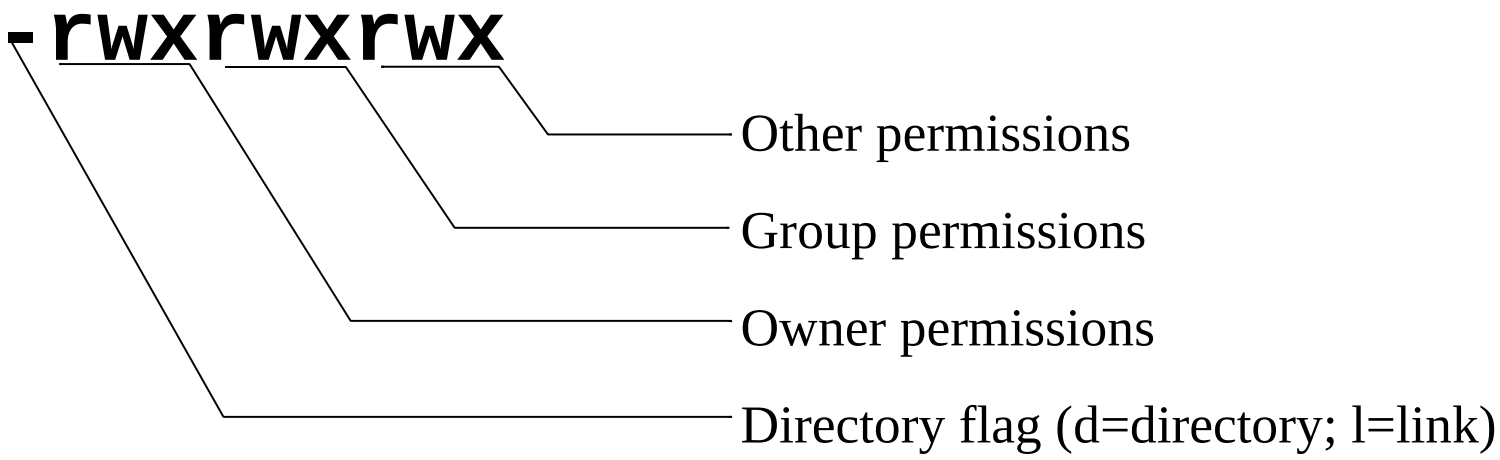
File Permissions

- The long version of a file listing (ls -l) will display the file permissions:

```
-rwxrwxr-x   1 Student Student        5224 Dec 30 03:22 hello
-rw-rw-r--   1 Student Student         221 Dec 30 03:59 hello.c
-rw-rw-r--   1 Student Student        1514 Dec 30 03:59 hello.s
drwxrwxr-x   7 Student Student        1024 Dec 31 14:52 posixuft
```



File Permissions



File Permissions

- Use the chmod command to change file permissions
  - The permissions are encoded as an octal number

```
chmod 755 file  # Owner=rwx Group=r-x Other=r-x
chmod 500 file2 # Owner=r-x Group=--- Other=---
chmod 644 file3 # Owner=rw- Group=r-- Other=r--

chmod +x  file  # Add execute permission to file for all
chmod o-r file  # Remove read permission for others
chmod a+w file  # Add write permission for everyone
```

Permissions

u - User who owns the file.
g - Group that owns the file.
o - Other.
a - All.
r - Read the file.
w - Write or edit the file.
x - Execute or run the file as a program.

File Permissions

- Umask
  - The umask command changes an *environment setting* that controls which permissions newly created files will have.
  - This command will never change the permissions of any existing files.
  - And unless extra steps (not discussed here) are taken, the new setting will be forgotten as soon as you log off.
  - To view the current setting, enter the command
    umask

  This will report the current setting as a set of three octal digits ().

  - To change the setting, enter the command
    umask new_value

    where *new_value* is three *octal digits*.
  - The first digit is the mask for the file owner (or user), the second is the mask for the group, and the third is the mask for all others.
  - Each octal digit is expanded to three binary digits, to set the value for each individual permission bit (0 show grant, 1 shows denial)

Mounting a new File system

- Mount
  - Mounts a file system that lives on a device to the main file tree
  - Start at Root file system
    - Mount to root
    - Mount to points currently mounted to root
  - /etc/fstab used to establish boot time mounting
  - Syntax
    mount [options] devices directory

The standard form of the mount command

mount -t type device dir

mount [-t fstype] something somewhere

Mounting a new File system

- mount command attaches the device to the file-system hierarchy (/)

  to make it accessible

  e.g. [user@ubuntu] $ mount −t ext2 /dev/fd0 /mnt/floppy0

  e.g. [user@ubuntu] $ mount −t iso9660 /dev/hdb /mnt/cdrom

- umount command – removes a device from the file−system hierarchy (the tree (/)).
- Sytax

  umount deviceName
- This needs to be done before you remove a floppy/CDROM or any other removable device

  e.g. [user@ubuntu] $ umount /mnt/floppy0

  e.g. [user@ubuntu] $ umount /mnt/cdrom

Mounting a new File system

- eject – command tells a device to eject the drive.
- Useful for cdrom / DVD drives.

  e.g. [user@ubuntu] $ eject /dev/cdrom

Maintaining File system

- Day-to-day management of the system is performed using the commands
  - df
    - To see the amount of free space on any given file system.
    - Disk usage information on a file system
    - df -h display the free and occupied space in human readable form
  - du:
    - Reports the number of bytes used by a file or a directory.
    - Disk usage for files and directories

Finding Files

- Utilities to find files:
  - locate:
    - *find files by name which is indexed in the locate database. If the database is not updated, it may not locate recent files.*
    - *To update the database use update command*
  - update: *update database tables for use to the locate command*

Finding Files

- Utilities to find files:
  - Which:
    - *determine which version of the program will be executed if you call one program say calc.exe.*
    - *There may be varieties of such code in the computer with the same name and are executable and also accessible through the PATH environment variable.*
    - searches paths to directories containing executable files
    - which looks only in the PATH (environment variable) of a user shell
    - which command is useful when troubleshooting "Command not Found" problems.

    e.g. [user@ubuntu ~] $ which mkdir

    e.g. [user@ubuntu ~] $ which ifconfig

Finding Files

- Utilities to find files:
  - Find:
    - *search for a file in a directory structure*
    - find can accept
      - search file names
      - file size
      - date of last change
      - and other file properties as criteria for a search
    - Syntax

      find [starting point] [criteria]

e.g. [user@ubuntu ~] $ find ~ −name *.txt - find all text files in the current directory or subdirectories

e.g. [user@ubuntu ~] $ find . −size +5000k - find all files in the current directory or subdirectories, that are bigger than 5 MB

[user@ubuntu ~] $ find . –atime +5 -find all files updated five day back in the current directory or subdirectories

Finding Files

- Utilities to find files:
  - whereis – *locates the binary, source, and manual page for a particular program,*
    - *it uses exact matches only*

  e.g. [user@ubuntu ~] $ whereis mkdir

Command History

- Can view previous commands in one display
  - history: displays history of commands, numbered
  - history N : displays only N previous commands
  - history –c: deletes all commands in history
  - !! : execute previous command
  - !N : execute the Nth command from top

Linux Shells

- A shell is any program that:
  - Takes input from the user,
  - Translates it into instructions that the operating system can understand, and
  - Conveys the operating system's output back to the user.

- Shell is
  - an interpreter
  - Interprets and executes commands
  - A user interface between the Linux system and the user
  - Used to call commands and programs
  - Powerful programming language

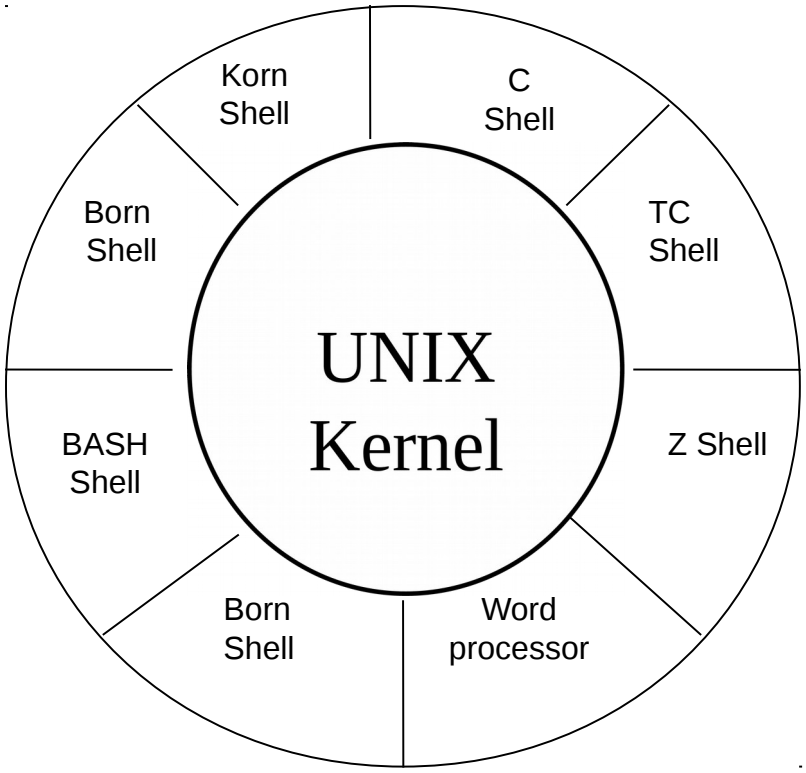
Linux Shells

- Many available shell in linux (bsh; ksh; csh; bash; tcsh)
  - 4 - 5 different shells are installed by default
  - Each shell has slightly different environment & features
- Examples
  - bash - most common, powerful
  - csh - C shell, less features than bash
  - sh - Bourne shell, oldest
  - zsh - new, quite powerful
  - **tcsh**
  - **ksh** - korn shell

Linux Shells

- Many available shell in linux (bsh; ksh; csh; bash; tcsh)
  - 4 - 5 different shells are installed by default
  - Each shell has slightly different environment & features
- Examples
  - bash - most common, powerful
  - csh - C shell, less features than bash
  - sh - Bourne shell, oldest
  - zsh - new, quite powerful
  - **tcsh**
  - **ksh** - korn shell
- to find out the particular shell
  - type echo $SHELL
  - should return something like /bin/bash
- to change shell use the command chsh

Linux Shells



Linux Shells

- Shell is Not Integral Part of OS
  - UNIX Among First to Separate shell from OS
  - Compare to MS-DOS, Mac, Win95, VM/CMS
  - GUI is NOT Required
  - To find all available shells in your system type following command:
    $ cat /etc/shells
  - To find your current shell type following command
    $ echo $SHELL
  - Default Shell Can Be Configured
    - chsh -s /bin/bash
  - Helps To Customize Environment

Linux Shells

- Terminal emulators
  - A **terminal emulator**, **terminal application**, **term**, or **tty** for short, is a program that emulates a "dumb" video terminal within some other display architecture.

    https://linuxhint.com/terminal_emulators_linux/
  - Though typically synonymous with a command line shell or text terminal, the term *terminal* covers all remote terminals, including graphical interfaces.
  - A terminal emulator inside a graphical user interface is often called a **terminal window**.
  - gnome-terminal is GNOME’s most used emulator
  - KDE uses konsole
  - No emulator needed if logging on to terminal directly

Linux Shells

- Start a terminal (without X) by using Alt-F1 to Alt-F8
  - Alt-F7 is X (the GUI)

- Shells:
  - Instead of clicking, you type long, cryptic commands
  - After login, you begin in your home directory
  - tab completion is nice

- Connecting to remote computer:
  - Connect to the Linux boxes via telnet or ssh (ssh preferred)
  - After connection, you have a shell (command prompt)

Linux Shells

ssh

- SSH is a protocol that allows you to connect to a remote computer
- For example, your Web server, your remote PC, etc
- Once connected to remote computer you can type commands to be carried out on that computer, such as moving and copying files, creating directories (folders), and running scripts.
- To talk to your server via SSH,
  - you need an *SSH client* on your computer and you also need three pieces of information about the remote computer which are
    1. IP address or hostname
    2. A username (the username that you'll use to login via SSH).
    3. A password (the password that's associated with the above username).

Linux Shells

- Ssh
  - Syntax

    ssh *username@hostname*
    - *Example* ***ssh sebsibe@10.4.23.168***
  - *This may request you to add in the log file and to enter password for the stated user at the given host if the user and the host exist*
  - *You can also give only the host (as* ***ssh 10.4.23.168)****. In that case it prompt you both the user name and password if the host exist*

Linux Shells

- telnet
  - Enables a user to telnet to another computer (Linux OS) from the command prompt (Windows OS).
- Availability
  - The telnet command is an external command that is available in the Microsoft operating systems.
  - Windows 2000
  - Windows XP
- Syntax
  - telnet [host [port]]
  - Example: telnet 10.4.23.68 8080
  - host specifies the hostname or IP address of the remote computer to connect to.
  - port specifies the port number or service name.

Functions of a Shell

- A shell can be used as
  - Command Line Interpretation
  - Program Initiation
  - Entering multiple commands on one line
  - I/O Redirection
    - Redirecting the standard output
    - The standard input redirection
    - Redirecting Error Message
  - Pipeline Connection
  - Substitution of file Names
  - Maintenance of Variable
  - Environment Control
  - Shell Programming (writing a set of shell commands to be executed in batch)

Shell Variables

- Linux shell variable is a variable that store number, string, filename and any other information which are accessible on a shell and in a script programming environment.
- Shell variables have access scope which is either
  - local or
  - environmental
- Environmental variables are accessible to any shell and shell script.
  - They are initialized when the system started.

- Local variables will be accessible after they are defined by the user on the shell that defines them.

Shell Variables

Local Variables

- Variable definition:
  VAR=value
  MYPATH = /afs/ictp.it/home/s/sdadi/Documents

- Obtain value of variable:
  ${VAR}
  echo $MYPATH
  This prints the value of variable MYHOME
  cd $MYPATH
  This changes your current directory into /afs/ictp.it/home/s/sdadi/Documents

- Unset variable:
  unset VAR
  unset $MYPATH

Shell Variables

Environment Variables

- Create an environment variable:
  export VAR=value
  export MYPATH = /afs/ictp.it/home/s/sdadi/Documents
- This enable you to define the environment variable MYPATH and accessible globally

- To obtain value of variable:
  ${VAR}
  echo $MYPATH

- This prints the value of variable MYHOME

Shell Variables

cd $MYPATH

This changes your current directory into /afs/ictp.it/home/s/sdadi/Documents

- To set the value of a variable to NULL you can use an of the following method:

  $var=

  $var=#

  $var="“

  $unset var

unset $MYPATH

$MYPATH=

$MYPATH=#

$MYPATH=""

Shell Variables

- Readonly variables:

  $readonly variable

  – Example:

  $Readonly pi=3.143

Shell Variables

Environment Variables

- Create an environment variable:

  export VAR=value

export MYPATH = /afs/ictp.it/home/s/sdadi/Documents

  This enable you to define the environment variable MYPATH and accessible globally

- To obtain value of variable:

  ${VAR}

echo $MYPATH

  This prints the value of variable MYHOME

Shell Variables

Environment Variables

- Example of predefined environmental variables

- $HOME
  - User home directory (often be abbreviated as “~”)
- $TERM
  - The type of terminal you are running (for example vt100, xterm, and ansi)
- $PWD
  - Current working directory
- $PATH
  - List of directories to search for commands (search path)

- Type the command bellow and see what the environment variables contains.

echo $PATH
echo $HOME

Shell Variables

- To see all the environment variables type
  env

- You can define your own environment variable that can be accessible all the time you log into the system.

- This can be done by putting them in the file **~/.bash_profile** file and type the command

  source ~/.bash_profile

- This is true is your shell is **bash**

Shell Variables

- There are some environment variables that has predefined meaning

$PAGER

is used by the man command (and others) to see what command should be used to display multiple pages (default setting is

export PAGER=more

You can test by setting

export PAGER=cat

Now see what will happen if you type

man ls

Shell Variables

- There are some environment variables that has predefined meaning

$PS1

This defines the main shell prompt string which you can use to create your own custom prompt

- For example check what happen to the user prompt if you export PS1 as

export PS1="[\u@\h \t \w]"

[username@hostName time path]

Login Scripts

- Used to customize environment before starting the shell
- Every shell has its own startup file; hidden
  - Bash has
    - .bashrc
    - .bash_profile
    - .bash_logout
- Can contain commands and programs

Login Scripts

- One of these scripts get run at startup….
  - .login, .bash_login, .mylogin, .profile, .bash_profile:
- If you use bash, you should create a .bash_profile so that you know what is getting run.
- One of these gets run when you logout or quit a shell
  - .logout, .bash_logout:
- One of these get run when you start a new shell (as opposed to logging in. An example would be when you start a new xterm).
  - .cshrc, .mycshrc, .bashrc:
- .cshrc and .mycshrc are run if you use tcsh and .bashrc is run if you use bash.
- .xsession: This gets run if you login via xdmcp. This is the only thing that gets run.
- It must be set as executable (chmod u+x) to work.

Process

- **Background versus foreground processes execution**
  - As a multitasking operating system, Linux lets you run processes in the background while continue to work in the foreground.

- Background
  - When a command is executed from the prompt with the token “&” at the end of the command line, the prompt immediately returns while the command continues is said to run in the background

- Foreground
  - When a command is executed from the prompt and runs to completion at which time the prompt returns is said to run in the foreground

Process

- **Background Process**
  - Unlike with a foreground process, the shell does not have to wait for a background process to end before it can run more processes.
  - Within the limit of the amount of memory available, you can enter many background commands one after another.
  - To run a command as a background process, type the command and add a space and an ampersand to the end of the command.
  - For example:

    $ *command1 &*
  - Immediately after entering the above command, the shell will execute the command.
  - While that is running in the background, the shell prompt will return.
  - At this point, you can enter another command for either foreground or background process.
  - Background jobs are run at a lower priority to the foreground jobs.

Process

- **Foreground Process Work**
  - A foreground process is different from a background process in two ways:
  - Some foreground processes show the user an interface, through which the user can interact with the program.
  - The user must wait for one foreground process to complete before running another one.
  - To start a foreground process, enter a command at the prompt, e.g.,
    $ *command1*
  - The next prompt will not appear until command1 finishes running.

Process

- **Foreground Process Work**
  - A foreground process is different from a background process in two ways:
  - Some foreground processes show the user an interface, through which the user can interact with the program.
  - The user must wait for one foreground process to complete before running another one.
  - To start a foreground process, enter a command at the prompt, e.g., $ *command1*
  - The next prompt will not appear until command1 finishes running.

Process

- Daemons
  - Is a background processes for system administration
  - These processes are usually started during the boot process
  - The processes are not assigned any terminals

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         5     1  0  1999 ?        00:00:14 [kswapd]
bin        254     1  0  1999 ?        00:00:00 [portmap]
root       307     1  0  1999 ?        00:00:23 syslogd -m 0
root       350     1  0  1999 ?        00:00:34 httpd
```

…Process

- Characteristics of a process:
  - PID
  - The parent PPID
  - Nice number
  - Terminal or TTY
  - User name of the real and effective user (RUID and EUID)
  - Real and effective group owner (RGID and EGID)

Process

- **Viewing process table**
  - **Ps command**
    - Reports the process status.

    [user@efossnet ~]$ps

    [user@efossnet ~]$ps –a

    [user@efossnet ~]$ ps –f

    [user@efossnet ~]$ ps -l

    [user@efossnet ~]$ps -ef | grep *username*
- Some Options
  - **-a** List information about all processes most frequently requested: all those except process group leaders and processes not associated with a terminal.
  - **-A** List information for all processes. Identical to -e, below.
  - **- e** List information about every process now running.
  - **-l** Generate a long listing.
  - **-f** Generate a full listing.

Process

- **Memory and CPU Usage Statistics**
  - **top** command
    - Display top CPU processes

    [user@efossnet ~]$ top

- **Ending a Process**
  - **Kill command**

    [user@efossnet ~]$ kill pid

    http://www.linuxmanpages.com/

Shell Scripting

- A shell script is one of a scripting language like AWK, Python, and PERL
- A shell script is a series of commands incorporated into a text file and executed like a program.
- A shell script is evaluated line by line.
- A shell script don’t require interpreter as the shell act as an interpreter
- The syntax of a shell script statement is the same as a command submitted to the shell on a console.
- Shell scripting language can be used to code more than one command in one line but each command must be separated by semicolon

Why Shell Scripting?

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

Shell Scripting

- The following is a sample shell script

```
#!/bin/bash
# this is a comment
clear
echo "The exit status of the above command is $?"
echo "The number of arguments is $#"
echo "The arguments are $*"
echo "The first is $1"
echo "My process number is $$"
echo "Enter a number from the keyboard:”
read number
echo "The number you entered was $number"
```

Shell Scripting

- The first line indicate which shell to use to execute the commands.
  - In this case the bash shell.
- While executing the shell, you can provide command line arguments and the shell script can interpret them as shown in the echo statements.
- You can also see that, shell scripts can have information the exit status of the previous command ($?), the number of arguments ($#), the list of arguments ($*), the process number ($$),each argument ($k for valid integer k > 0)
- Moreover, it is possible to read values from a keyboard and print into a file/screen.

Shell Scripting

- The following is a sample shell script

```
#!/bin/bash
# Script to print user information who currently login , current date &
  time
clear
echo  -e "Hello $USER"
echo  -e "Today is \c";date
echo  -e "Number of user login : \c       " ; who | wc -l
echo  -e "Calendar"
cal
exit 0
```

Shell Scripting (*echo Command)*

- *Use echo command to display text or value of variable.*

echo [options] [string, variables...]

Displays text or variables value on screen.

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a	*alert (bell)*
\b	*backspace*
\c	*suppress trailing new line*
\n	*new line*
\r	*carriage return*
\t	*horizontal tab*
**	*backslash*

For e.g. $ echo -e "An apple a day keeps away \a\t\tdoctor\n"

Shell Scripting

- Shell scripts are commonly used to automate complex and repetitive tasks.
- To execute a shell script file, there are two approach
  - Set execution mode to the script file (chmod +x filename) and execute the file as
    ./scriptFileName arguments
  - Execute the shell with the script file as an argument as
    bash scriptFileName arguments

Keyboard input

- Reading from keyboard

  $read variable

- Example

```
#!/bin/sh
# Shows how to read a line from stdin
echo "Would you like to exit this script now?“
read answer
echo $answer
```

Command Line Argument

- Passing Arguments to Shell Programs

```
#!/bin/bash
#will print all the arguments passed to it all at once
echo “$0: $# arguments are passed.
echo “the first argument is $1”
echo “the seconded argument is $2”
echo “All the arguments are: $*”
```

Shell Operators

- The Bash shell operators are divided into three groups:
  - defining and evaluating operators,
  - arithmetic operators, and
  - redirecting and piping operators

Quoting

- Backslash (\)
  - Cause single character to be escaped
  - Example

    \$ take $ literally
- Single Quote(‘)
  - used as pair, cause a group of characters to be escaped
  - take all string literally
  - Example:
    - : ‘string’
    - $echo ‘* $HOME’
    - * $HOME

Quoting

- Double Quote(" ")
  - cause a group of character to be escaped but allows some character to be interpreted
  - take string literally, except $,`,\,”,’

Backquote

- It is used to capture the output of a UNIX utility
- A command in backquotes is executed and then replaced by the output of the command
- Backquotes slow down a script if used repeatedly
- **Example**

```
#!/bin/bash
#Illustrates using backquotes
# Output of 'date' stored in a variable
Today="`date`"
echo Today is $Today
```

- Execute the script with

bash backquotes.sh

Shell Script Operator & precedence

- Arithmetic operators: +,-,*,/,%
- comparison operators: -lt, -ge, -eq, -ne, -ge, -gt
  - boolean/logical operators: &&, ||
- Parentheses: (,)
- precedence is the same as C, Java
  - We use operators to form expression

Expression

- Integer expression
  - **expr** command used to perform arithmetic operations on the shell variables
  - Syntax:

    expr var1 operator var2

Expression

- Example: integer expression

```
$expr 10 + 4
$14
$expr 10 \* 4
$40
$int=`expr 10 + 4`
$expr $int / 2
$7
$expr `expr 10 +4 ` / 2
$7
```

Example

```
#!/bin/bash
count=5
count=`expr $count + 1`
echo $count
```

(no space between before and after the symbol =)

- chmod u+x math.sh
- math.sh

Backquote

```
#!/bin/bash
#Perform some arithmetic
x=24
y=4
Result=`expr $x \* $y`
echo "$x times $y is $Result"
```

```
#!/bin/sh
# Illustrates how to change the contents of a variable with tr
Cat_name="Piewacket“
echo "Cat_name is $Cat_name"
Cat_name=`echo $Cat_name | tr 'a' 'i'`
echo "Cat_name has changed to $Cat_name"
```

- One can also specify ranges of characters. This example converts upper case to lower case

```
tr 'A-Z' 'a-z' < file
```

Example

- vi integer_expr1.sh

```
#!/bin/bash
echo "This program will perform arithmetic operation"
read num1
read num2
expr $num1 + $num2
expr $num1 - $num2
expr $num1 \* $num2
expr $num1 / $num2
expr $num1 % $num2
```

Example

vi integer_expr2.sh

```
#!/bin/bash
echo "This program will perform arthemetic operation"
read num1
read num2
result=`expr $num1 + $num2`
echo "$num1 + $num2 = $result"
result=`expr $num1 - $num2`
echo "$num1 - $num2 = $result"
result=`expr $num1 \* $num2`
echo "$num1 \* $num2 = $result"
result=`expr $num1 / $num2`
echo "$num1 + $num2 = $result"
result=`expr $num1 % $num2`
echo "$num1 % $num2 = $result"
```

Test Command

- Examines some conditions and returns a zero exit status if the condition is true and non-zero exit status if the condition is false.
- Test condition is usually used in controlling action in branching and looping structure
- Syntax:

  test condition

  where condition is the expression to be tested.
- The condition that can be tested can be:
  - String operations
  - Integer relationships
  - File operators that test the existence or state of a file.
  - Logical operation that allow for and/or combinations of other operations.

Test Command (Numeric data)

	intvar1 –eq intvar2	Intvar1== intvar2
	intvar1 –ne intvar2	Intvar1!= intvar2
	intvar1 –gt intvar2	Intvar1> intvar2
	intvar1 –ge intvar2	Intvar1>= intvar2
	intvar1 –lt intvar2	Intvar1< intvar2
	intvar1 –le intvar2	Intvar1<= intvar2

Test Command (Character data)

	Strvar1=strvar2	True if strvar1 is the same length as strvar2 and contains the same characters
	Strvar1!=strvar2	True if they are not the same.
	-n strvar	True if not null
	-z strvar	True if strvar is null
	Strvar	True if strvar is not null

"$A" = "$B"	true if string A equals string B
"$A" != "$B"	true if string A not equal to string B
$X ! -gt $Y	true if string X is not greater than Y

Test Command (boolean operands)

$E -a $F	true if expressions E and F are both true (i.e **AND**)
$E -o $F	true if either expression E or expression F is true (i.e. **OR**)

Test Command (for File)

	-r filename	True if the user has read permission
	-w filename	True if the user has writhe permission
	-x filename	True if the user has execute permission
	-f filename	True if the file is regular file
	-d filename	True if the file is directory file
	-c filename	True if the file is character oriented-device file
	-s filename	True if the size of the file is not zero.
	-t fnum	True if the device associated with the file descriptor fnum is terminal device.

Shell Scripting

- Shell script permit branching and looping structure.
- Branching is possible using the **if** or the **case** statement
- Looping is possible using either the **for loop** or **while** statement.

Shell Scripting

if branching

```
if [ test ] then
    commands-if-test-is-true
else
    commands-if-test-is-false
fi
```

- The space before and after the test condition is very important

- Test condition should be evaluated to either true (0) or false (non-zero) value

Example

- vi relational_expr1.sh

```
#!/bin/bash
echo "This program will perform arithmetic
    operation"
read num1
read num2
if [ $num1 -lt $num2 ]
then
    echo "$num1 < $num2"
fi
if  [ $num1 -le $num2 ]
then
    echo "$num1 <= $num2"
fi
if  [ $num1 -eq $num2 ]
then
     echo "$num1 == $num2"
fi
if  [ $num1 -ne $num2 ]
then
    echo "$num1 != $num2"
fi
if  [ $num1 -ge $num2 ]
then
     echo "$num1 >= $num2"
fi
if  [ $num1 -gt $num2 ]
then
      echo "$num1 > $num2"
fi
```

Shell Scripting

Case statement

- The case statements are a convenient way to perform multi-way branches where one input pattern must be compared to several alternatives:

```
case variable in
        pattern1) statement (executed if variable matches with pattern1) ;;
        pattern2) statement ;;
        etc.
esac
```

Shell Scripting

- The following script uses a case statement to have a guess at the type of non-directory non-executable files passed as arguments on the basis of their extensions
- Note
  - how the bitwise "or" operator | can be used to denote multiple patterns,
  - how "*" has been used as a catch-all, and the effect of the forward single quotes (backquote) `):

Shell Scripting

```
#!/bin/sh
read f
        case $f in
            core)                   echo "$f: a core dump file" ;;
            *.c)                    echo "$f: a C program" ;;
            *.cpp|*.cc|*.cxx)       echo "$f: a C++ program" ;;
            *.txt)                  echo "$f: a text file" ;;
            *.pl)                   echo "$f: a PERL script" ;;
            *.html|*.htm)           echo "$f: a web document" ;;
            *)                      echo "$f: appears to be "`file -b $f` ;;
        esac
```

Shell Scripting

Looping

- We can execute the same set of commands repetitively with or without changing the actual argument the command takes

- Looping in shell script can be implemented using for loop or while loop.

Shell Scripting

For loop syntax

for variable **in** list **do**
    statements (referring to $variable)
done

Shell Scripting

```
#!/bin/sh
for f in $* do
    if [ -f $f -a ! -x $f ]
      then
        case $f in
            core) echo "$f: a core dump file" ;;
            *.c) echo "$f: a C program" ;;
            *.cpp|*.cc|*.cxx) echo "$f: a C++ program" ;;
            *.txt) echo "$f: a text file" ;;
            *.pl) echo "$f: a PERL script" ;;
            *.html|*.htm) echo "$f: a web document" ;;
            *) echo "$f: appears to be "`file -b $f` ;;
        esac
    fi
done
```

Shell Scripting

Example1

```
#!/bin/sh
for f in *.txt do
   echo sorting file $f
   cat $f | sort > $f.sorted
   echo "sorted file has been output to $f.sorted "
done
```

Shell Scripting

while loop syntax

while [test] **do**

statements (to be executed while test is true)

done

Shell Scripting

Example2

```
#!/bin/sh
while [ ! -s input.txt ] do
   echo waiting...
   sleep 5
done
Echo " input.txt is ready"
```

Shell Scripting

- Example3

```
#!/bin/sh
while true do
    if [ -s input.txt ]
        echo "input.txt is ready exit"
    fi
    echo waiting...
    sleep 5
done
```